TM1000 Preliminary Data Book

PRELIMINARY INFORMATION

TERMS AND CONDITIONS

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or most work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or most work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from improper use or sale.

Philips Semiconductors and Philips Electronics North America Corporation register eligible circuits under the Semiconductor Chips Protection Act.

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	Formative or in Design	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	Preproduction Product	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	Full Production	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve the design and supply the best possible product.

NOTES

Change Bars: Change bars have been used in this data book to indicate areas that have changed since the April 1997 data book. The change bars appear as heavy vertical lines positioned on the left side of text that has changed.

Product Name Change: The name of the TriMedia Processor was recently changed from TM-1 to TM1000.

TriMedia Product Group, 811 E. Arques Avenue, Sunnyvale, CA 94088

Foreword

by Gert Slavenburg

The Trimedia TM1000 is the first member of an architectural family of programmable multimedia processors. TM1000 contains an ultra-high performance Very Long Instruction Word processor, as well as a complete intelligent video and audio input/output subsystem. The processor has an instruction set that is optimized for processing audio, video and 3-D graphics. It includes powerful SIMD operators for eight- and 16-bit signal datatypes as well as a full complement of 32-bit IEEE compatible floating point operations.

TM1000 is intended as a multi-standard video, audio, graphics accelerator for PCI based personal computers. It can also be used as the master CPU in stand-alone multimedia PCI-bus-based systems.

The architecture of the Trimedia family came about as the result of many years of effort of many dedicated individuals. Going back in history, the origin of Trimedia was laid by the LIFE-1 VLIW processor, designed by Junien Labrousse and myself in 1987. Work continued afterwards in Philips Research Labs, Palo Alto. My special thanks go to the entire Palo Alto research team: Mike Ang, Uzi Bar-Gadda, Peter Donovan, Martin Freeman, Eino Jacobs, Beomsup Kim, Bob Law, Yen Lee, Vijay Mehra, Pieter van der Meulen, Ross Morley, Mariette Parekh, Bill Sommer, Artur Sorkin and Pierre Uszynski.

The Palo Alto period matured the architecture—we ported all video and audio algorithms that we could find to the compiler/simulator and refined the operation set. In addition, we learned how to give the architecture a market direction. In May 1994, Philips management—in particular Cees-Jan Koomen, Eddy Odijk, Theo Claasen and Doug Dunn—decided to develop Trimedia into a major Philips Semiconductors product line.

Under the guidance of Keith Flagler, the Trimedia team was built. All of them contributed to take this from a set of interesting ideas to a reliable and competitive product in a short period of time. The Trimedia team included Fuad Abu Nofal, Karel Allen, Mike Ang, Robert Aquino, Manju Asthana, Patrick de Bakker, Shiv Balakrishnan, Jai Bannur, Marc Berger, Sunil Bhandari, Rusty Biesele, Ahmet Bindal, David Blakely, Hans Bouwmeester, Steve Bowden, Robert Bradfield, Nancy Breede, Shawn Brown, Sujay Chari, Catherine Chen, Howen Chen, Yanming Chen, Yong Cho, Scott Clapper, Matthew Clayson, Paul Coelho, Richard Dodds, Marc Duranton, Darcia Eding, Aaron Emigh, Li Chi Feng, Keith Flagler, Jean Gobert, Sergio Golombek, Mike Grimwood, Yudi Halim, Hari Hampapuram, Carl Hartshorn, Judy Heider, Laura Hrenko, Jim Hsu, Eino Jacobs, Marcel Janssens, Patricia Jones, Hann-Hwan Ju, Jayne Keith, Bhushan Kerur, Ayub Khan, Keith Knowles, Mike Kong, Ashok Krishnamurti, Yen Lee, Patrick Leong, Bill Lin, Laura Ling, Chialun Lu, Naeem Maan, Nahid Mansipur, Mike Maynard, Vijay Mehra, Jun Mejia, Derek Meyer, Prabir Mohanty, Saed Muhssin, Chris Nelson, Stephen Ness, Keith Ngo, Francis Nguyen, Kathleen Nguyen, Derek Noonburg, Ciaran O'Donnel, Sang-Ju Park, Charles Peplinski, Gene Pinkston, Maryam Pirayou, Pardha Potana, Bill Price, Victor Ramamoorthy, Babu Rao Kandamilla, Ehsan Rashid, Selliah Rathnam, Margaret Redmond, Donna Richardson, Alan Rodgers, Tilakray Roychoudhury, Hani Salloum, Chris Salzmann, Bob Seltzer, Ravi Selvaraj, Jim Shimandle, Deepak Singh, Bill Sommer, Juul van der Spek, Manoj Srivastava, Renga Sundararajan, Ken-Sue Tan, Ray Ton, Steve Tran, Cynthia Tripp, Ching-Yih Tseng, Allan Tzeng, Barbara Vendelin, John Vivit, Rudy Wang, Rogier Wester, Wayne Wonchoba, Anthony Wong, Sara Wu, David Wyland, Ken Xie, Vincent Xie, Bettina Yeung, Robert Yin, Charles Young, Grace Yun, Elena Zelayeta and Vivian Zhu.

Expert help and feedback was received from many. In particular, I'd like to mention Kees van Zon of Philips Eindhoven for the help with filtering-related issues, and Craig Clapp of PictureTel for excellent feedback on all aspects of the architecture.

Working with Brian Case has been a joy. He has taken our engineering documents and turned them into a book that is so much clearer than any of us could ever make it. He has the rare talent to both understand, design and explain both hardware and software.

My special thanks go to Joe Kostelec. He made me understand that my ambitions could better be realized in California than in Europe. Furthermore, his vision and his wisdom are credited with keeping this project alive and growing until the 'investment decision.'

The vision of a universal media accelerator is credited to Jaap de Hoog. Jaap, I wish you were here to see it come to fruition.

–Gerrit Slavenburg, January 1996

Table of Contents

3 DSPCPU Architecture

4 Custom Operations for Multimedia

5 Cache Architecture

10 PCI Interface

11 SDRAM Memory System

12 System Boot

13 Image Co Processor

14 VLD Register Interface

15 I2C Interface

16 V.34 Sync Serial Interface

17 JTAG Functional Specification

18 On-Chip Semaphore Assist Device

Index

Pin List Chapter 1

by Fuad Abunofal, Mike Ang, Patrick Leong, Naeem Maan, Gert Slavenburg

1.1 I/O CIRCUIT SUMMARY

TM1000 has a total of 163 functional pins, not counting VDDQ, VSSQ, VREF_PCI and VREF_PERIPH and digital power/ground. TM1000 uses the types of I/O circuits shown in the table below.

I/O Circuit Type	I/O Circuit Description
IN	Pure 3.3-Volt input
IN-5	5-Volt tolerant input
OUT	3.3-Volt output, reflected wave switching, low drive capability
OUT-strong	3.3-Volt output, incident wave switching drive capability into 50-Ohm load
I/O	Pure 3.3-Volt I/O circuit
I/O-5	3.3-Volt output driver combined with 5-Volt tolerant input
I/OD-5	5-Volt tolerant open drain output, with 5-Volt tolerant input (for I^2C)
IN-PCI	3.3- and 5-Volt PCI compliant input (on 3.3 Volt supply)
I/O-PCI	I/O conforming to 3.3- and 5-Volt PCI drive specification (on 3.3-Volt supply). The normal use of this I/O circuit is as PCI 'tri state'. Where the output is used as 'sustained tri state', this is indicated in the pin description.
I/OD-PCI	Open drain conforming to 3.3- and 5-Volt PCI drive specification, combined with PCI compliant input
OD-PCI	Open drain conforming to 3.3- and 5-Volt PCI drive specification (5-Volt tolerant)

1.2 SIGNAL PIN LIST

In the table below, a pin name ending in a '#' designates an active-low signal (the active state of the signal is a low voltage level). All other signals have active-high polarity.

Pin Name	PQFP	Type	Description
Main Clock Interface			
TRI_CLKIN	143	IN	Main Input Clock. The SDRAM clock outputs (MM_CLK) can be set to 2x or 3x this frequency. The on-chip DSPCPU clock (DSPCPU_CLK) can be set to 1x, 5/4, 4/3, 3/2 or 2x the SDRAM clock frequency.
VDDQ	142	PWR	Quiet VDD for the PLL subsystem.
VSSQ	144	GND	Quiet VSS for the PLL subsystem.
Miscellaneous System Interface			
TRI_RESET#	209	IN-PCI	TM1000 RESET input. This pin can be tied to the PCI RST# signal in PCI bus systems. Upon receiving RESET, TM1000 initiates its boot protocol.
BOOT_CLK	146	IN	Used for testing purposes. Must be connected to TRI_CLKIN for normal operation.
RESERVED1	145	IN	Reserved input. Has to be connected to VDDQ for proper operation.
RESERVED2	148	OUT	Reserved test output. Should be left unconnected.
VREF_PCI	240	PWR	VREF_PCI must be connected to 5V for use in a 5 Volt PCI system or to VSS for use in a 3.3 Volt PCI system.
VREF_PERIPH	184	PWR	VREF_PERIPH should be connected to 5V if any of the (non-PCI) inputs provided to TM1000 are 5 Volt inputs. VREF_PERIPH should be connected to 0 Volt if all input signals, with the possible exception of PCI signals are 3.3 Volt inputs.
TRI_USERIRQ	147	IN-5	General purpose level/edge interrupt input. Vectored interrupt source number 4.
TRI_TIMER_CLK	141	IN-5	External general purpose clock source for timers. Max 40 MHz.

Pin Name	PQFP	Type	Description
Main Memory Interface			
MM_CLK0 MM_CLK1	86 83	OUT-strong	SDRAM Output Clock at 2x or 3x TRI_CLKIN frequency. Two identical outputs are provided to reliably drive several small memory configurations without external glue.
MM_MATCHOUT	89	OUT-strong	Phase match clock output. This output must be connected to MM_MATCHIN through a transmission line + load + transmission line structure that mirrors the transmission line characteristics of the SDRAM clock, the SDRAM input load and the SDRAM data return line.
MM_MATCHIN	92	IN	Phase match clock input. Refer to MM_MATCHOUT above.
MM_A00 MM_A01 MM_A02 MM_A03 MM_A04 MM_A05 MM_A06 MM_A07 MM_A08 MM_A09 MM_A10 MM_A11	98 96 95 93 81 80 78 77 76 74 99 101	OUT	Main memory address bus; used for row and column addresses
MM_DQ00 MM_DQ01 MM_DQ02 MM_DQ03 MM_DQ04 MM_DQ05 MM_DQ06 MM_DQ07 MM_DQ08 MM_DQ09 MM_DQ10 MM_DQ11 MM_DQ12 MM_DQ13 MM_DQ14 MM_DQ15 MM_DQ16 MM_DQ17 MM_DQ18 MM_DQ19 MM_DQ20 MM_DQ21 MM_DQ22 MM_DQ23 MM_DQ24 MM_DQ25 MM_DQ26 MM_DQ27 MM_DQ28 MM_DQ29 MM_DQ30 MM_DQ31	121 122 123 125 126 127 130 132 116 115 114 112 111 110 108 107 73 72 70 69 67 66 65 63 51 52 54 55 56 58 59 61	I/O	32-bit data I/O bus
MM_CKE0 MM_CKE1	118 45	OUT	Clock enable output to SDRAM's. Two identical outputs are provided in order to reliably drive several small memory configurations without external glue.
MM_CS0# MM_CS1# MM_CS2# MM_CS3#	47 136 48 133	OUT	Chip select for DRAM rank n; active low
MM_RAS#	102	OUT	Row address strobe; active low
MM_CAS#	104	OUT	Column address strobe; active low
MM_WE#	105	OUT	Write enable; active low
MM_DQM0 MM_DQM1 MM_DQM2 MM_DQM3	138 119 50 62	OUT	MM_DQ Mask Enable; these are byte enable signals for the 32-bit MM_DQ bus

Pin Name	PQFP	Type	Description
PCI Interface (note: current buffer design allows drive/receive from either 3.3 or 5V PCI bus)			
PCI_CLK	39	IN-PCI	All PCI input signals are sampled with respect to the rising edge of this clock. All PCI outputs are generated based on this clock
PCI_AD00 PCI_AD01 PCI_AD02 PCI_AD03 PCI_AD04 PCI_AD05 PCI_AD06 PCI_AD07 PCI_AD08 PCI_AD09 PCI_AD10 PCI_AD11 PCI_AD12 PCI_AD13 PCI_AD14 PCI_AD15 PCI_AD16 PCI_AD17 PCI_AD18 PCI_AD19 PCI_AD20 PCI_AD21 PCI_AD22 PCI_AD23 PCI_AD24 PCI_AD25 PCI_AD26 PCI_AD27 PCI_AD28 PCI_AD29 PCI_AD30 PCI_AD31	44 42 41 38 36 35 33 32 29 27 26 24 23 21 20 18 3 2 1 239 238 236 235 234 230 228 227 225 222 221 219 218	I/O-PCI	Multiplexed address and data.
PCI_C/BE#0 PCI_C/BE#1 PCI_C/BE#2 PCI_C/BE#3	30 17 5 231	I/O-PCI	Multiplexed bus Commands and Byte Enables. High for command, low for byte enable.
PCI_PAR	16	I/O-PCI	Even Parity across AD and C/BE lines.
PCI_FRAME#	6	I/O-PCI	Sustained Tristate. Frame is driven by a master to indicate the beginning and duration of an access.
PCI_IRDY#	7	I/O-PCI	Sustained Tristate. Initiator Ready indicates that the bus master is ready to complete the current data phase.
PCI_TRDY#	9	I/O-PCI	Sustained Tristate. Target Ready indicates that the bus target is ready to complete the current data phase.
PCI_STOP#	12	I/O-PCI	Sustained Tristate. Indicates that the target is requesting that the master stop the current transaction.
PCI_IDSEL	232	IN-PCI	Used as Chip Select during configuration read/write cycles.
PCI_DEVSEL#	10	I/O-PCI	Sustained Tristate. Indicates whether any device on the bus has been selected.
PCI_REQ#	216	I/O-PCI	Driven by TM1000 as PCI bus master to request use of the PCI bus.
PCI_GNT#	224	IN-PCI	Indicates to TM1000 that access to the bus has been granted.
PCI_PERR#	13	I/O-PCI	Sustained Tristate. Parity Error generated/received by TM1000.
PCI_SERR#	14	OD-PCI	System Error. This signal is asserted when operating as target and detecting an address parity error.

Pin Name	PQFP	Type	Description
PCI_INTA# PCI_INTB# PCI_INTC# PCI_INTD#	210 212 213 215	I/OD-PCI	• Can operate as input (power up default) or output, as determined by direction control bits in PCI MMIO register INT_CTL. • As input, a PCI_INT# pin can be used to receive PCI interrupt requests (normal PCI use is active low, level sensitive mode, but the VIC can be set to treat these as positive edge triggered mode). As input, a PCI_INT# pin can also be used as general interrupt request pin if not needed for PCI. • As output, the value of a PCI_INT# can be programmed through PCI MMIO registers to generate interrupts for other PCI masters.
JTAG Interface			
JTAG_TDI	171	IN-5	JTAG Test Data Input
JTAG_TDO	173	OUT	JTAG Test Data Output
JTAG_TCK	172	IN-5	JTAG Test Clock Input
JTAG_TMS	174	IN-5	JTAG Test Mode Select Input
Video In			
VI_CLK	175	I/O-5	• If configured as input (power up default): A positive transition on this incoming video clock pin samples all other VI_DATA input signals below if VI_DVALID is HIGH. If VI_DVALID is LOW, VI_DATA is ignored. Clock and data rates of up to 38 MHz are supported to allow for 16:9 aspect ratio video with 5% clock margin. • If configured as output: Programmable output clock to drive an external video A/D converter. Can be programmed to emit integral dividers of DSPCPU_CLK.
VI_DVALID	190	IN-5	VI_DVALID indicates that valid data is present on the VI_DATA lines. If HIGH, VI_DATA will be accepted on the next VI_CLK positive edge. If LOW, no VI_DATA will be sampled.
VI_DATA0 VI_DATA1 VI_DATA2 VI_DATA3 VI_DATA4 VI_DATA5 VI_DATA6 VI_DATA7	176 178 179 181 182 183 185 186	IN-5	CCIR656 style YUV 4:2:2 data from a digital camera, or general purpose high speed data input pins. Sampled on VI_CLK if VI_DVALID HIGH.
VI_DATA8 VI_DATA9	187 189	IN-5	Extension high speed data input bits to allow use of 10 bit video A/D converters. Sampled on VI_CLK if VI_DVALID HIGH. VI_DATA[8] serves as START and VI_DATA[9] as END message input in message passing mode.
I^2C Interface			
IIC_SDA	160	I/OD-5	I^2C serial data
IIC_SCL	161	I/OD-5	I^2C clock
Video Out			
VO_DATA0 VO_DATA1 VO_DATA2 VO_DATA3 VO_DATA4 VO_DATA5 VO_DATA6 VO_DATA7	192 193 194 196 197 198 200 201	OUT	CCIR656 style YUV 4:2:2 digital output data. Output on positive edge of VO_CLK, and (in external sync mode) synchronized on VO_IO1 and VO_IO2 sync signals from the DENC. Also general purpose high speed data output channel.
VO_IO1	204	I/O-5	• This pin can function as HS (Horizontal Sync) input, HS output or as STMSG (Start Message) output. • If set as HS input, it can be set to respond to positive or negative edge transitions. If the Video Out operates in external sync mode and the selected transition occurs, the VO generates a sequence of a CCIR 656 EAV code, horizontal blanking, an SAV code and YUV 4:2:2 pixel data on VO_DATA. • In message passing mode, this pin acts as STMSG output. A high indicates that the current data presented on VO_DATA[7:0] is the start byte of a message.
VO_IO2	206	I/O-5	• This pin can function as FS (Frame Sync) input, FS output or as ENDMSG output. • If set as FS input, it can be set to respond to positive or negative edge transitions. • If the Video Out operates in external sync mode and the selected transition occurs, the Video Out sends two fields of video data. • In message passing mode, this pin acts as ENDMSG output. A high indicates that the current data presented on VO_DATA[7:0] is the end byte of a message.

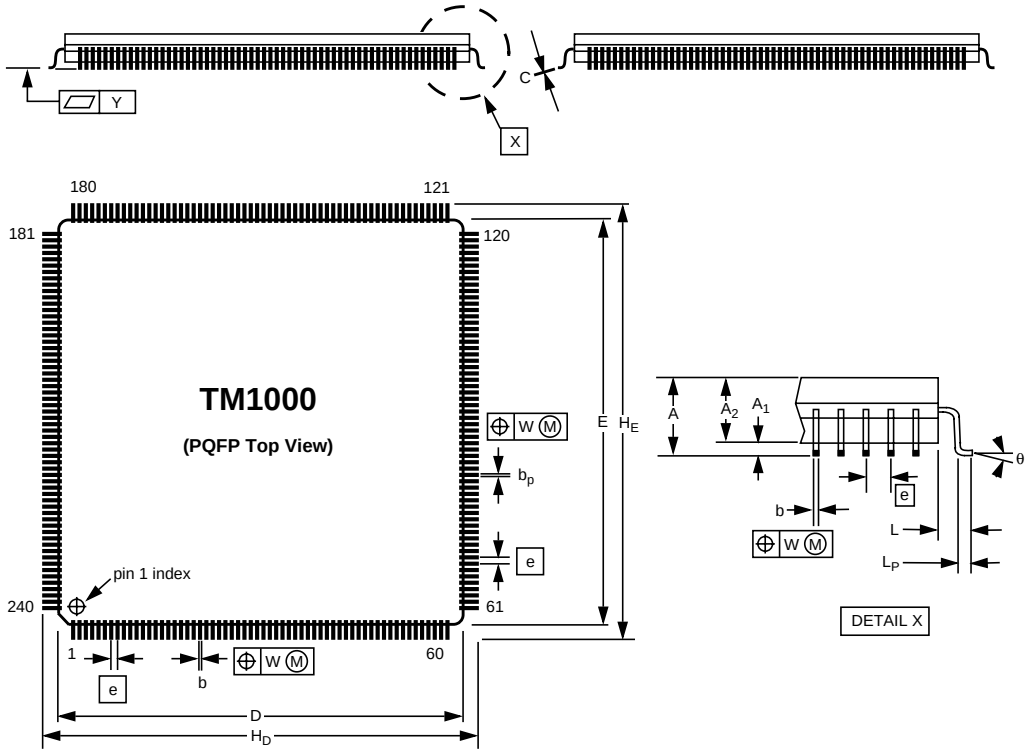
Pin Name	PQFP	Type	Description
VO_CLK	203	I/O-5	• If configured as input (power up default): VO_CLK is received from external display clock master circuitry. • If configured as output, TM1000 emits a programmable clock frequency. The emitted frequency can be set between approx. 4 MHz and 80 MHz with a resolution of 0.07 Hz. The clock generated is frequency accurate and has low jitter properties due to a combination of an on-chip DDS (Direct Digital Synthesizer) and VCO/PLL. • The Video Out unit emits VO_DATA on a positive edge of VO_CLK.
Audio In (always acts as receiver, but can be master or slave for A/D timing)			
AI_OSCLK	153	OUT	Over-Sampling Clock. This output can be programmed to emit any frequency up to 40-MHz with a resolution of 0.07-Hz. It is intended for use as the $256f_s$ or $384f_s$ over sampling clock by external A/D subsystem.
AI_SCK	152	I/O-5	• When Audio-In is programmed as serial-interface timing slave (power-up default), AI_SCK is an input. AI_SCK receives the serial bitclock from the external A/D subsystem. This clock is treated as fully asynchronous to TM1000 main clock. When Audio In is programmed as the serial-interface timing master, AI_SCK is an output. AI_SCK drives the serial clock for the external A/D subsystem. The frequency is a programmable integral divide of the AI_OSCLK frequency. AI_SCK is limited to 20 MHz. The sample rate of valid samples embedded within the serial stream is limited to 100 kHz.
AI_SD	149	IN-5	Serial Data from external A/D subsystem. Data on this pin is sampled on positive or negative edges of AI_SCK as determined by the CLOCK_EDGE bit in the AI_SERIAL register.
AI_WS	150	I/O-5	• When Audio In is programmed as the serial-interface timing slave (power-up default), AI_WS acts as an input. AI_WS is sampled on the same edge as selected for AI_SD. • When Audio In is programmed as the serial-interface timing master, AI_WS acts as an output. It is asserted on the opposite edge of the AI_SD sampling edge. AI_WS is the word-select or frame-synchronization signal from/to the external A/D subsystem.
Audio Out (always acts as sender, but can be master or slave for D/A timing)			
AO_OSCLK	156	OUT	Over Sampling Clock. This output can be programmed to emit any frequency up to 40 MHz, with a resolution of 0.07 Hz. It is intended for use as the 256 or $384f_s$ over sampling clock by the external D/A conversion subsystem.
AO_SCK	158	I/O-5	• When Audio Out is programmed to act as the serial interface timing slave (power up default), AO_SCK acts as input. It receives the Serial Clock from the external audio D/A subsystem. The clock is treated as fully asynchronous to the TM1000 main clock. • When Audio Out is programmed to act as serial interface timing master, AO_SCK acts as output. It drives the Serial Clock for the external audio D/A subsystem. The clock frequency is a programmable integral divide of the AO_OSCLK frequency. AO_SCK is limited to 20 MHz. The sample rate of valid samples embedded within the serial stream is limited to 100 kHz.
AO_SD	159	OUT	Serial Data to external audio D/A subsystem. The timing of transitions on this output is determined by the CLOCK_EDGE bit in the AO_SERIAL register, and can be on positive or negative AO_SCK edges.
AO_WS	155	I/O-5	• When Audio-Out is programmed as the serial-interface timing slave (power-up default), AO_WS acts as an input. AO_WS is sampled on the opposite AO_SCK edge at which AO_SD is asserted. • When Audio Out is programmed as serial-interface timing master, AO_WS acts as an output. AO_WS is asserted on the same AO_SCK edge as AO_SD. AO_WS is the word-select or frame-synchronization signal from/to the external D/A subsystem. Each audio channel receives 1 sample for every WS period.
V.34 interface (synchronous serial interface to an off-chip modem front-end)			
V34_CLK	162	IN-5	Clock signal of the synchronous serial interface to an off-chip modem analog frontend or ISDN terminal adapter. Provided by the receive channel of an external communication device.
V34_RXFSX	164	IN-5	Receive Frame Sync reference of the synchronous serial interface, provided by the receive channel of an external communication device.
V34_RXDATA	165	IN-5	Receive Serial Data input. Provided by the receive channel of an external communication device.
V34_TXDATA	167	OUT	Transmit Serial Data output. Sent to the transmit channel of the external communication device.

Pin Name	PQFP	Type	Description
V34_IO1	168	I/O-5	General purpose programmable I/O. Set to input on powerup.
V34_IO2	170	I/O-5	General purpose programmable I/O. Set to input on powerup. Can also be programmed to function as the transmit channel frame synchronization reference output.

1.3 POWER PIN LIST

PCI Interface			Main Memory Interface			Peripherals, Miscellaneous System Interface	
VSS	VDD		VSS	VDD		VSS	VDD
PQFP	PQFP		PQFP	PQFP		PQFP	PQFP
211	214		49	46		151	154
217	220		57	53		157	163
223	226		64	60		166	169
229	233		71	68		177	180
237	4		82	75		188	191
8	15		85	79		195	199
11	22		88	84		202	205
19	28		91	87		207	208
25	34		97	90			
31	40		103	94			
37			109	100			
43			117	106			
			124	113			
			128	120			
			131	129			
			134	135			
			137	139			
			140				

1.4 PQFP



DIMENSIONS

Unit	A max	A min	A_2	b	c	$D^{(1)}$	$E^{(1)}$	e	H_D	H_E	L	Lp min	W	Y	θ
mm	3.86 3.50	0.25	3.43 3.17	0.27 0.17	0.13	31.79 31.59	31.79 31.59	0.50	34.80 34.40	34.80 34.40	1.30	0.50	0.08	0.08	7° 0°

NOTE: Plastic or metal protrusions of 0.25 mm maximum per side are not included.

1.5 DC/AC CHARACTERISTIC

1.5.1 Maximum Ratings

In Accordance with the Absolute Maximum Rating system (IEC 134)

Symbol	Parameter	Min.	Max	Units	Notes
V_{DD}	Supply voltage	-0.5	4.6	V	
V_{I-5V}	DC input voltage on all 5V pins	-0.5	VDD	V	
$V_{I-3.3V}$	DC input voltage on all 3.3V pins	-0.5	VDD	V	
I_{DD}	Supply current	-	1200	mA	
P_{tot}	Total power dissipation	0	4	W	
T_{stg}	Storage temperature range	-65	150	Deg. C	
T_{amb}	Operating ambient temperature range	0	70	Deg. C	
V_{ESD}	Electrostatic handling for all pins	-	$\pm$2000	V	1

Notes: 1. Equivalent to discharging a 150pF capacitor through a 1.5Kohm series resistor.

1.5.2 DC Characteristics

V_{dd} = 3.13V to 3.46V; T_{amb} = 0 to 70 deg. C, unless otherwise specified

Symbol	Parameter	Condition/Notes	Min.	Max	Units
V_{DD}	Supply voltage		3.135	3.465	V
I_p	Total supply current	Input LOW; no output loads; 100 MHz		1200	mA
I_{pdn}	Total supply current	CPU Power Down mode; 100 MHz		300	mA
V_{IH-5v}	Input HIGH voltage - for I/O-5		2.0	V_{DD} + 0.5	V
$V_{IH-3.3v}$	Input HIGH voltage - for I/O-3.3v		2.0	V_{DD} + 0.3	V
V_{IL-5v}	Input LOW voltage- for I/O-5		-0.5	0.8	V
$V_{IL-3.3v}$	Input LOW voltage - for I/O-3.3v		-0.3	0.8	V
I_{IL-5v}	Input leakage current - for I/O-5v	V_{IN} = 0.5, 2.7V, Note 1	-70	70	uA
$I_{IL--3.3v}$	Input leakage current - for I/O-3.3v	0 < V_{IN} < 2.7V, Note 1	-0	10	uA
V_{OH-5v}	Output HIGH voltage - for I/O-5v	I_{OUT} = -2.0mA	2.4		V
$V_{OH-3.3v}$	Output HIGH voltage - for I/O-3.3v	I_{OUT} = -0.5mA	0.9V_{DD}		V
V_{OL-5v}	Output LOW voltage - for I/O-5v	I_{OUT} = 6.0mA		0.55	V
$V_{OL-3.3v}$	Output LOW voltage - for I/O-3.3v	I_{OUT} = 1.5mA		0.1V_{DD}	V
C_{IN}	Input Pin capacitance			8	pF
3-State Outputs					
$I_{O\ off}$	High-impedance output current				
C_I	High-impedance output capacitance				
I^2C-Bus, SDA/SCL					
V_{IL-I2C}	Input HIGH voltage - for I2C pins		-0.5	0.8	V
V_{IH-I2C}	Input HIGH voltage - for I2C pins		2.0	V_{DD} + 0.5	V
I_{OL}	Low Level output Current	V_{OL} = 0.4V	3		mA
I_{L-I2C}	Leakage Current	V_I = VSS or VDD		10	uA
C_{IN-I2C}	Input Pin capacitance	V_I = VSS		8	pF

Notes: 1. Equivalent to discharging a 150pF capacitor through a 1.5Kohm series resistor

1.5.3 SDRAM Interface Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
T_{CH}	MM_CLK high pulse width	3.5				1
T_{CL}	MM_CLK low pulse width	3.5				1
T_{PD}	Propagation delay of Address		2.4	6.6	ns	2
T_{PD}	Propagation delay of Control		2.9	6.6	ns	2
T_{PD}	Propagation delay of Data			6.6	ns	2
T_{OH}	Output Holdtime of Data, Address and Control	1.0			ns	2
T_{SU}	Input Data Setup Time	1.0			ns	3,4
T_{IH}	Input Data Hold Time			2.5	ns	3,4

Notes:
1. Maximum output load on **MM_CLK0** and **MM_CLK1** is 10pF.
2. **MM_CLK0** or **MM_CLK1** are used as the reference clock.
3. **MM_MATCHIN** is used as a reference clock.
4. **MM_MATCHIN** must be connected to **MM_MATCHOUT** through a transmission line + load + transmission line structure that mirrors the transmission line characteristics of the SDRAM clock, the SDRAM input load and the SDRAM data return line.

1.5.4 PCI Bus Timing

The following specifications were taken from PCI specifications, Rev. 2.1 for the 33MHz bus.

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$T_{val\text{-}PCI\ (Bus)}$	Clk to Signal Valid Delay, Bused signals	2		11	ns	1,2,3
$T_{val\text{-}PCI\ (ptp)}$	Clk to Signal Valid Delay, Point to Point signals	2		12	ns	1,2,3
$T_{on\text{-}PCI}$	Float to Active Delay	2			ns	1
$T_{Off\text{-}PCI}$	Active to Float Delay			28	ns	1
$T_{su\text{-}PCI}$	Input Set up Time to CLK- bused signals	7			ns	3,4
$T_{su\text{-}PCI\ (ptp)}$	Input Set up Time to CLK - point to point signals	12			ns	3,4
$T_{h\text{-}PCI}$	Input Hold Time from CLK	0			ns	4
$T_{rst\text{-}PCI}$	Reset Active Time after power stable	1			ms	5
$T_{rst\text{-}clk\text{-}PCI}$	Reset Active Time after CLK stable	100			ms	5
$T_{rst\text{-}off\text{-}PCI}$	Reset Active to output float delay			40	ns	5,6

Notes:
1. See the timing measurement conditions in Figure 1-1. It is important that all driven signal transitions drive to their V_{oh} or V_{ol} level within one T_{cyc}.
2. Minimum times are measured at the package pin with the load circuit shown in Figure 1-5. Maximum times are measured with the load circuit shown in Figure 1-3 and Figure 1-4.
3. **REG#** and **GNT#** are point-to-point signals and have different input setup times than do bused signals. All other signals are bused.
4. See the Timing measurement conditions in Figure 1-2.
5. **RST#** and is asserted and de-asserted asynchronously with respect to CLK.
6. All output drivers must be floated when **RST#** is active.
7. For the purpose of Active/Float timing measurements, the Hi-Z or “off” state is defined to be when the total current delivered through the component pin is less than or equal to the leakage current specification.

1.5.5 JTAG I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$T_{clk\text{-}TDO}$	JTAG-TCK to JTAG-TDO Valid Delay	2		10	ns	1
$T_{su\text{-}TCK}$	Input Set up Time to JTAG-TCK	10			ns	1
$T_{h\text{-}TCK}$	Input Hold Time from JTAG_TCK	2			ns	1

Notes: 1. See the timing measurement conditions in Figure 1-6.

1.5.6 I^2C I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
f_{SCL}	SCL clock frequency			400	kHz	1
T_{BUF}	Bus Free time	1			us	2
$T_{su\text{-}STA}$	Start condition set up time	1			us	3
$T_{h\text{-}STA}$	Start condition hold time	1			us	3
T_{LOW}	SCL LOW time	1			us	1
T_{HIGH}	SCL HIGH time	1			us	1
T_r	SCL and SDA rise time			0.3	us	1
T_f	SCL and SDA fall time			0.3	us	1
$T_{su\text{-}SDA}$	Data set-up time	100			ns	4
$T_{h\text{-}SDA}$	Data hold time	0			ns	4
$T_{dv\text{-}SDA}$	SCL LOW to data out valid			0.5	us	5
$T_{dv\text{-}STO}$	SCL HIGH to data out	1			ns	5

Notes: 1. See the timing measurement conditions in Figure 1-7.
2. See the timing measurement conditions in Figure 1-8.
3. See the timing measurement conditions in Figure 1-9.
4. See the timing measurement conditions in Figure 1-10.
5. See the timing measurement conditions in Figure 1-11.

1.5.7 VideoIn I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$f_{VI\text{-}CLK}$	VideoIn clock frequency			38	MHz	1,2
$T_{su\text{-}CLK}$	Input Set up Time to VI_CLK	9			ns	1
$T_{h\text{-}CLK}$	Input Hold Time from VI_CLK	3			ns	1

Notes: 1. See the timing measurement conditions in Figure 1-12.
2. 100MHz is supported only for message passing mode

1.5.8 VideoOut I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$f_{VO\text{-}CLK}$	VideoOut clock frequency			80	MHz	1
$T_{CLK\text{-}DV}$	VO_CLK to VO_DATA (or VO_IO*) out	2	8.4	9	ns	1,3
$T_{CLK\text{-}DV}$	VO_CLK to VO_DATA (or VO_IO*) out	2	8.1	9	ns	1,4
$T_{su\text{-}CLK}$	VO_IO* Set up Time to VO_CLK	10			ns	2
$T_{h\text{-}CLK}$	VO_IO* Hold Time from VO_CLK	3			ns	2

Notes: 1. See the timing measurement conditions in Figure 1-13.
2. See the timing measurement conditions in Figure 1-14.
3. CLKOUT asserted, i.e. Video Out is the source of VO_CLK
4. CLKOUT negated, i.e. the external world is the source of VO_CLK

1.5.9 AudioIn I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$f_{AI\text{-}SCK}$	AudioIn AI_SCK clock frequency			20	MHz	1,2
$T_{su\text{-}SCK}$	input Set up Time to AI_SCK	10			ns	1,2
$T_{h\text{-}SCK}$	input Hold Time from AI_SCK	5			ns	1,2
$T_{SCK\text{-}WS}$	AI_SCK to AI_WS			10	ns	1,2

Notes: 1. See the timing measurement conditions in Figure 1-15.

2. The timing measurements are done with respect to the clock edge according to CLOCK_EDGE

1.5.10 AudioOut I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
$f_{AO\text{-}SCK}$	AudioOut AO_SCK clock frequency			20	MHz	
$T_{SCK\text{-}DV}$	AO_SCK to AO_SD valid	2	7.2	10	ns	1,3,4
$T_{SCK\text{-}DV}$	AO_SCK to AO_SD valid	2	7.5	10	ns	1,3,5
$T_{su\text{-}SCK}$	Input Set up Time to AO_SCK	10			ns	1,2
$T_{h\text{-}SCK}$	Input Hold Time from AO_SCK	5			ns	1,2
$T_{SCK\text{-}WS}$	AO_SCK to AO_WS		8.7	10	ns	1,3

Notes: 1. See the timing measurement conditions in Figure 1-16.
2. See the timing measurement conditions in Figure 1-17.
3. The timing measurements are done with respect to the AO_SCK clock edge according to CLOCK_EDGE
4. TM-1 is the serial interface master, i.e. AO_SCK is an output
5. TM-1 is serial interface slave, i.e. AO_SCK is an input

1.5.11 SSI I/O Timing

Symbol	Parameter	Min.	Typ.	Max	Units	Notes
f_{V34_CLK}	V34_CLK clock frequency			20	MHz	1
$T_{CLK\text{-}DV}$	V34_CLK to data valid	2		10	ns	1
$T_{su\text{-}CLK}$	Input Set up Time to V34_CLK	10			ns	1
$T_{h\text{-}CLK}$	Input Hold Time from V34_CLK	5			ns	1

Notes: 1. See the timing measurement conditions in Figure 1-18.
1. See the timing measurement conditions in Figure 1-19.

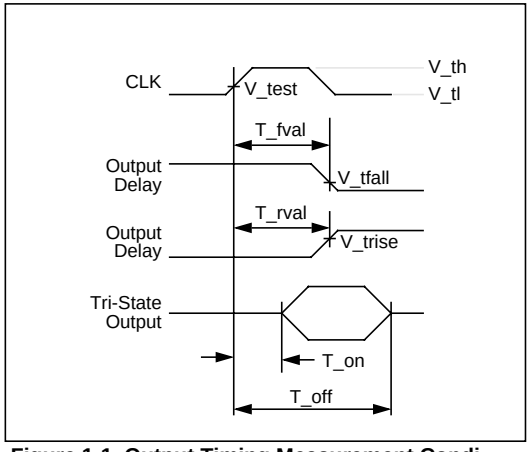


Figure 1-1. Output Timing Measurement Conditions

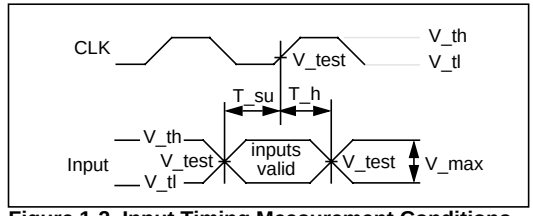


Figure 1-2. Input Timing Measurement Conditions

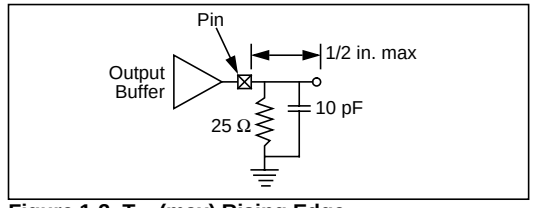


Figure 1-3. T_{val}(max) Rising Edge

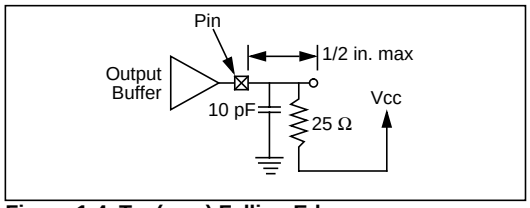


Figure 1-4. T_{val}(max) Falling Edge

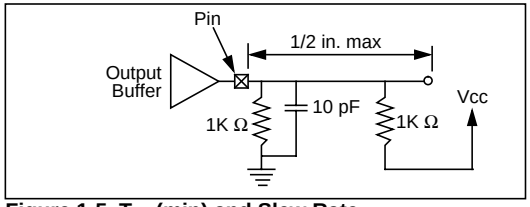


Figure 1-5. T_{val}(min) and Slew Rate

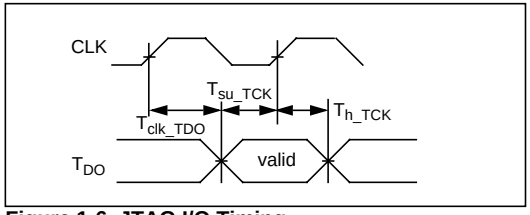


Figure 1-6. JTAG I/O Timing

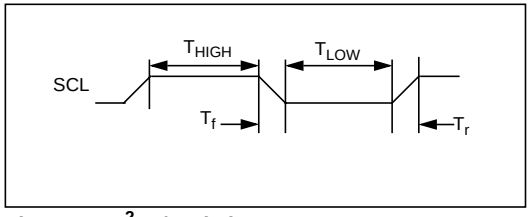


Figure 1-7. I^2C I/O Timing

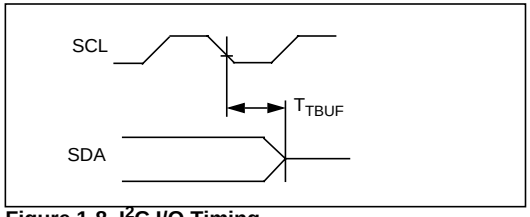


Figure 1-8. I^2C I/O Timing

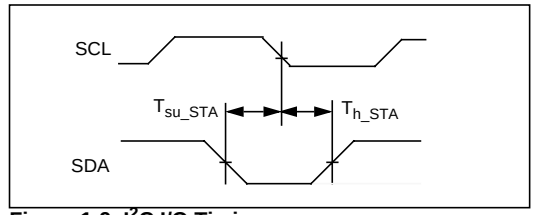


Figure 1-9. I^2C I/O Timing

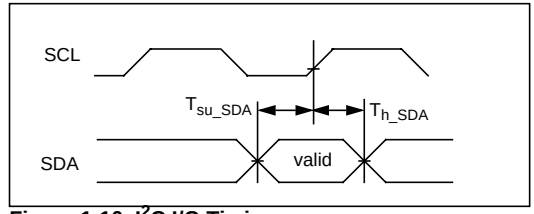


Figure 1-10. I^2C I/O Timing

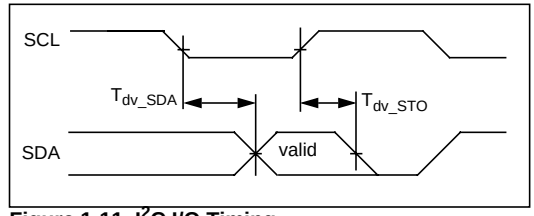


Figure 1-11. I^2C I/O Timing

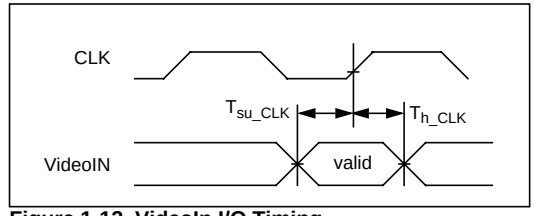


Figure 1-12. VideoIn I/O Timing

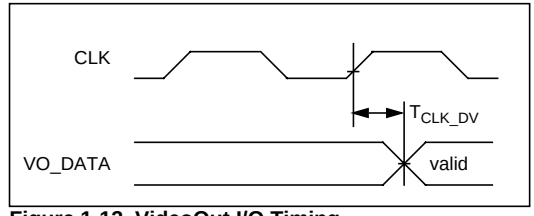


Figure 1-13. VideoOut I/O Timing

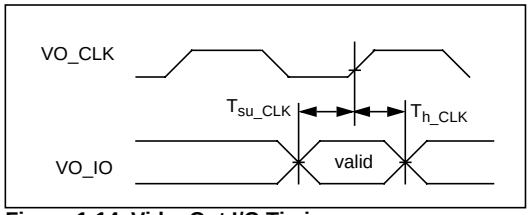


Figure 1-14. VideoOut I/O Timing

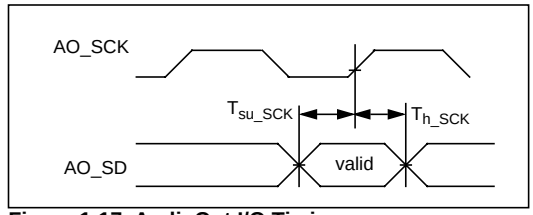


Figure 1-17. AudioOut I/O Timing

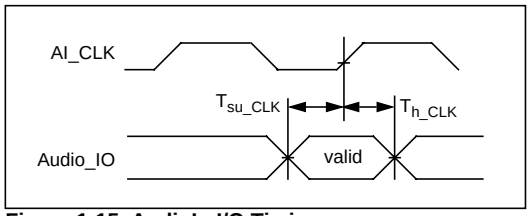


Figure 1-15. AudioIn I/O Timing

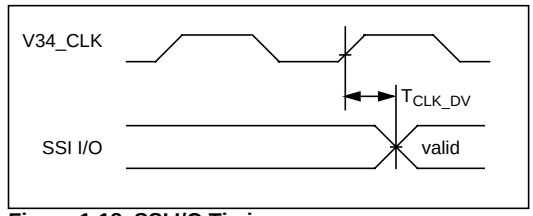


Figure 1-18. SSI I/O Timing

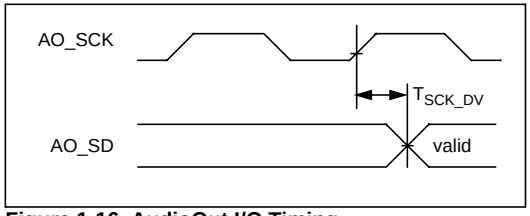


Figure 1-16. AudioOut I/O Timing

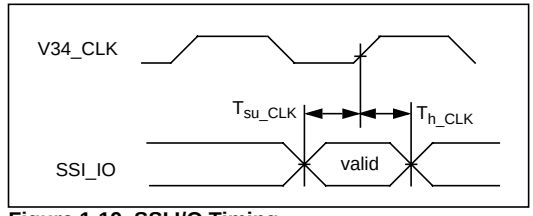


Figure 1-19. SSI I/O Timing

Overview Chapter 2

by Gert Slavenburg

2.1 TM1000 FUNDAMENTALS

TM1000 is a media processor for high-performance multimedia applications that deal with high-quality video and audio. These applications can range from low-cost, single-purpose systems such as video phones to reprogrammable, mutli-purpose plug-in cards for traditional personal computers. TM1000 easily implements popular multimedia standards such as MPEG-1 and MPEG-2, but it's orientation around a powerful general-purpose CPU (called the DSPCPU) makes it capable of implementing a variety of multimedia algorithms, whether open or proprietary.

More than just an integrated microprocessor with unusual peripherals, the TM1000 microprocessor is a fluid computer system controlled by a small real-time OS kernel that runs on the VLIW processor core. TM1000 contains a DSPCPU, a high-bandwidth internal bus, and internal bus-mastering DMA peripherals.

TM1000 is the first member of a family of chips that will carry investments in C/C++ media software forward in time. Compatibility between family members is at the source-code level; binary compatibility between family members is not guaranteed. All family members, however, will be able to perform the most important multimedia functions, such as running MPEG-2 software.

Defining software compatibility at the source-code level gives Philips the freedom to strike the optimum balance between cost and performance for all the chips in the TM1000 family. Powerful compilers ensure that programmers never need to resort to non-portable assembler programming. Programmers use TM1000's multimedia operations from source code; these DSP-like operations are invoked with a familiar function-call syntax.

As the first member of the Trimedia media processor family, TM1000 is designed both for use as accelerator

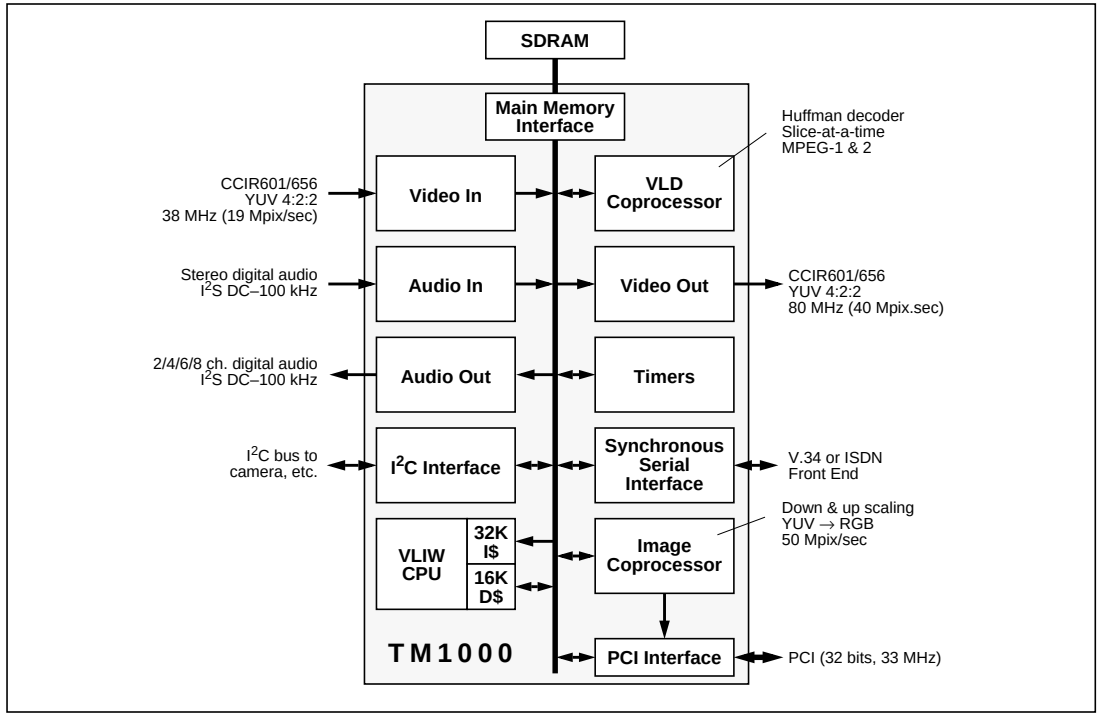


Figure 2-1. TM1000 block diagram.

in a PC environment, or as the sole CPU in stand-alone systems.

Because it is based on a general-purpose CPU, TM1000 can serve as a multi-function PC enhancement vehicle. Typically, a PC must deal with multi-standard video and audio streams, and users desire both decompression and compression, if possible. While the CPU chips used in PCs are becoming capable of low-resolution real-time video decompression, high-quality video decompression of studio resolution video—not to mention compression—is still out of reach. Further, users demand that their systems provide live video and audio without sacrificing the responsiveness of the system.

TM1000 enhances a PC system to provide real-time multimedia, and it does so with the advantages of a special-purpose, embedded solution—low cost and chip count—*and* the advantages of a general-purpose processor—reprogrammability. For PC applications, TM1000 far surpasses the capabilities of fixed-function multimedia chips.

TM1000 is capable of stand-alone operation. In this case it boots from a low-cost attached serial EEPROM. The actual application software is brought in from PCI bus attached ROM or from a peripheral device.

Future media processor family members will have different sets of interfaces appropriate for their intended use.

2.2 TM1000 CHIP OVERVIEW

The key features of TM1000 are:

- A very powerful, general-purpose VLIW processor core (the DSPCPU) that coordinates all on-chip activities. In addition to implementing the non-trivial parts of multimedia algorithms, this processor runs a small real-time operating system that is driven by interrupts from the other units.
- DMA-driven multimedia input/output units that operate independently and that properly format data to make software media processing efficient.
- DMA-driven multimedia coprocessors that operate independently and in parallel with the DSPCPU to perform operations specific to important multimedia algorithms.
- A high-performance bus and memory system that provides communication between TM1000's processing units.

Figure 2-1 shows a block diagram of the TM1000 chip. The bulk of a TM1000 system consists of the TM1000 microprocessor itself, a block of synchronous DRAM (SDRAM), and whatever external circuitry is needed to interface to the incoming and/or outgoing multimedia data streams. TM1000 can gluelessly interface to the standard PCI bus for personal-computer-based applications; thus, TM1000 can be placed directly on the PC mainboard or on a plug-in card.

Figure 2-2 shows a possible TM1000 system application. A video-input stream, if present, might come directly from a CCIR 601-compliant video camera chip in YUV 4:2:2 format; the interface is glueless in this case. A non-

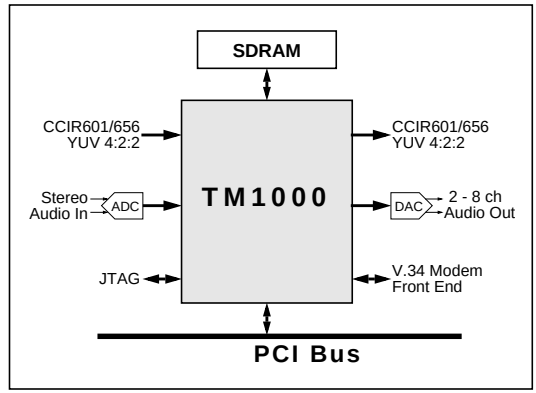


Figure 2-2. TM1000 system connections. A minimal TM1000 system requires few supporting components.

standard camera chip can be connected via a CCIR 601 interface chip (such as the Philips SAA7111). A CCIR 601 output video stream is provided directly from the TM1000 to drive a dedicated video monitor. Stereo audio input and up to 8 channel audio output require external ADC and DAC support. The operation of the video and audio interface units is highly customizable through programmable parameters.

The glueless PCI interface allows the TM1000 to display video via a host PC's video card and to play audio via a host PC's sound hardware. The Image Coprocessor provides display support for live video in an arbitrary number of arbitrarily overlapped windows.

Finally, the V.34/ISDN interface requires only an external front-end chip and phone line interface to provide remote communication support. It can be used to connect TM1000-based systems for video phone or video conferencing applications, or it can be used for general-purpose data communication in PC systems.

2.3 BRIEF EXAMPLES OF OPERATION

The key to understanding TM1000 operation is observing that the DSPCPU and peripherals are time-shared and that communication between units is through SDRAM memory. The DSPCPU switches from one task to the next; first it decompresses a video frame, then it decompresses a slice of the audio stream, then back to video, etc. As necessary, the DSPCPU issues commands to the peripheral function units to orchestrate their operation.

The DSPCPU can enlist the ICP and video-in or video-out units to help with some of the straightforward, tedious tasks associated with video processing. The function of these units is programmable. For example, some video streams need to be scaled horizontally, so the video units can handle the most common cases of horizontal down- and up-scaling on the fly without intervention from the DSPCPU. The ICP is very well suited for arbitrary size horizontal and vertical video resizing and color space conversion.

2.3.1 Video Decompression in a PC

A typical mode of operation for a TM1000 system is to serve as a video-decompression engine on a PCI card in a PC. In this case, the PC doesn't need to know the TM1000 has a powerful, general-purpose CPU; rather, the PC just treats the hardware on the PCI card as a "black-box" engine.

Video decompression begins when the PC operating system hands the TM1000 a pointer to compressed video data in the PC's memory (the details of the communication protocol are typically handled by a software driver installed in the PC's operating system).

The DSPCPU fetches data from the compressed video stream via the PCI bus, decompresses frames from the video stream, and places them into local SDRAM. Decompression may be aided by the VLD (variable-length decoder) unit, which implements Huffman decoding and is controlled by the DSPCPU.

When a frame is ready for display, the DSPCPU gives the ICP (image coprocessor) a display command. The ICP then autonomously fetches the decompressed frame data from SDRAM and transfers it over the PCI bus to the frame buffer in the PC's video display card (or in PC system memory if the PC uses a UMA (Unified Memory Architecture) frame buffer). The ICP accommodates arbitrary window size, position, and overlaps.

Alternately, the Video Out unit can be used to send a single high resolution video stream to Video input ports of PC graphics cards.

2.3.2 Video Compression

Another typical application for TM1000 is in video compression. In this case, uncompressed video is usually supplied directly to the TM1000 system via the video-in unit. A camera chip connected directly to the video-in unit supplies YUV data in eight-bit, 4:2:2 format. The video-in unit takes care of sampling the data from the camera chip and demultiplexing the raw video to SDRAM in three separate areas, one each for Y, U, and V.

When a complete video frame has been read from the camera chip by the video-in unit, it interrupts the DSPCPU. The DSPCPU compresses the video data in software (using a set of powerful data-parallel operations) and writes the compressed data to a separate area of SDRAM.

The compressed video data can now be disposed of in any of several ways. It can be sent to a host system over the PCI bus for archival on local mass storage, or the host can transfer the compressed video over a network,. The data can also be sent to a remote system using the integrated V.34/ISDN interface to create, for example, a video phone or video conferencing system.

Since the powerful, general-purpose DSPCPU is available, the compressed data can be encrypted before being transferred for security.

2.4 TM1000 FUNCTION UNITS

The remainder of this chapter provides a brief introduction to the internal components of TM1000.

2.4.1 Internal "Data Highway" Bus

The internal data bus connects all internal blocks together and provides access to internal control registers (in each function unit), external SDRAM, and the external PCI bus. The internal bus consists of separate 32-bit data and address buses, and transactions on the bus use a block-transfer protocol. On-chip peripheral units and co-processors can be masters or slaves on the bus.

Access to the internal bus is controlled by a central arbiter, which has a request line from each potential bus master. The arbiter is programmable to provide guaranteed bandwidth and latency to requestors so that the arbitration algorithm can be tailored for different applications. Peripheral units make requests to the arbiter for bus access, and depending on the arbitration mode, bus bandwidth is allocated to the units in different amounts. Each mode allocates bandwidth differently, but each mode guarantees each unit a minimum bandwidth and maximum service latency. All unused bandwidth is allocated to the DSPCPU.

The bus allocation mechanism is one of the features of TM1000 that makes it a true real-time system instead of just a highly integrated microprocessor with unusual peripherals.

2.4.2 VLIW Processor Core

The heart of TM1000 is its powerful 32-bit DSPCPU core. The DSPCPU implements a 32-bit linear address space and 128, fully general-purpose 32-bit registers. The registers are not separated into banks; any operation can use any register for any operand.

The core uses a VLIW instruction-set architecture and is fully general-purpose. TM1000 uses a VLIW instruction length that allows up to five simultaneous operations to be issued. These operations can target any five of the 27 functional units in the DSPCPU, including integer and floating-point arithmetic units and data-parallel DSP-like units.

Although the processor core runs a real-time operating system to coordinate all activities in the TM1000 system, the processor core is not intended for true general-purpose use as the only CPU in a computer system. For example, the TM1000 processor core does not implement demand paged virtual memory, memory address translation, or 64 bit floating point - all essential features in a general-purpose computer system.

TM1000 uses a VLIW architecture to maximize processor throughput at the lowest possible cost. VLIW architectures have performance exceeding that of superscalar general-purpose CPUs without the extreme complexity of a superscalar implementation. The hardware saved by eliminating superscalar logic reduces cost and allows the integration of multimedia-specific features that enhance the power of the processor core.

The TM1000 operation set includes all traditional microprocessor operations. In addition, multimedia-specific operations are included that dramatically accelerate standard video compression and decompression algorithms. As just one of the five operations issued in a single TM1000 instruction, a single “custom” or “media” operation can implement up to 11 traditional microprocessor operations. These multimedia-specific operations combined with the VLIW architecture result in tremendous throughput for multimedia applications.

The DSPCPU core is supported by separate 16-KB data and 32-KB instruction caches. The data cache is dual-ported to allow two simultaneous accesses, and both caches are eight-way set-associative with a 64-byte block size.

2.4.3 Video-In Unit

The video-in unit interfaces directly to any CCIR 601/656-compliant device that outputs eight-bit parallel, 4:2:2 YUV time-multiplexed data. Such devices include direct digital camera systems, which can connect gluelessly to TM1000 or through the standard CCIR 656 connector with only the addition of ECL level converters. A single chip external device can be used to convert to/from serial D1 professional video. Non-CCIR-compliant devices can use a digital video decoder chip, such as the Philips SAA7111, to interface to TM1000.

The video-in unit demultiplexes the captured YUV data before writing it into local TM1000 SDRAM. Separate planar data structures are maintained for Y, U, and V.

The video-in unit can be programmed to perform on-the-fly horizontal resolution subsampling by a factor of two if needed. Many camera systems capture a 640-pixel/line or 720-pixel/line image; with subsampling, direct conversion to a 320-pixel/line or a 360-pixel/line image can be performed with no DSPCPU intervention. Further, if subsampling is required eventually, performing this function during data capture reduces initial storage and bus bandwidth requirements.

2.4.4 Video-Out Unit

The video-out unit essentially performs the inverse function of the video-in unit. Video-out generates an eight-bit, multiplexed YUV data stream by gathering bits from the separate Y, U, and V planar data structures in SDRAM. While generating the multiplexed stream, the video-out unit can also up-scale horizontally by a factor of two to convert from CIF/SIF to CCIR 601 resolution.

Since the video-out unit likely drives a separate video monitor—not a PC’s video screen— video out is also capable of generating sophisticated graphics overlays with alpha blending for implementing user interfaces.

2.4.5 Image Coprocessor (ICP)

The image coprocessor (ICP) is used for several purposes to off-load tasks from the DSPCPU, such as copying an image from SDRAM to the host’s video frame buffer. Although these tasks can be easily performed by the DSPCPU, they are a poor use of the relatively expensive CPU resource. When performed in parallel by the ICP, these tasks are performed efficiently by simple hardware, which allows the DSPCPU to continue with more complex tasks.

The ICP can operate as either a memory-to-memory or a memory-to-PCI coprocessor device.

In memory-to-memory mode, the ICP can perform either horizontal or vertical image filtering and resizing. The ICP implements 32 FIR filters of five adjacent pixel input values. The filter coefficients are fully programmable, and the position of the output pixel in the output raster determines which of the 32 FIR filters is applied to generate that output pixel value. Thus, the output raster is on a 32-times finer grid than the input raster. The filtering is done in either the horizontal or vertical direction but not both. Two applications of the ICP are required to filter and scale in both directions.

In memory-to-PCI mode, the ICP can perform horizontal resizing followed by color-space conversion. For example, assume an $n \times m$ pixel array is to be displayed in a window on the PC video screen while the PC is running a graphical user interface. The first step (if necessary) would use the ICP in memory-to-memory mode to perform a vertical resizing. The second step would use the ICP in memory-to-PCI mode to perform horizontal resizing and optional colorspace conversion from YUV to RGB.

While sending the final, resampled and converted pixels over the PCI bus to the video frame buffer, the ICP uses a full, per-pixel occlusion bit mask—accessed in destination coordinates—to determine which pixels are actually written to the graphics card frame buffer for display. Conditioning the transfer with the bit mask allows TM1000 to accommodate an arbitrary arrangement of overlapping windows on the PC video screen.

Figure 2-3 illustrates a possible display situation and the data structures in SDRAM that support the ICP’s operation. On the left in Figure 2-3, the PC’s video screen has four overlapping windows. Two, Image 1 and Image 2, are being used to display video generated by TM1000.

The right side of Figure 2-3 shows a conceptual view of SDRAM contents. Two data structures are present, one for Image 1 and the other for Image 2. Figure 2-3 represents a point in time during which the ICP is displaying Image 2.

When the ICP is displaying an image (i.e., copying it from SDRAM to a frame buffer), it maintains four pointers to the data structures in SDRAM. Three pointers locate the Y, U, and V data arrays, and the fourth locates the per-pixel occlusion bit map. The Y, U, and V arrays are indexed by source coordinates while the occlusion bit map is accessed with screen coordinates.

As the ICP generates pixels for display, it performs horizontal scaling and colorspace conversion. The final RGB pixel value is then copied to the destination address in the screen’s frame buffer only if the corresponding bit in the occlusion bit map is a one.

As shown in the conceptual diagram, the occlusion bit map has a pattern of 1s and 0s that corresponds to the

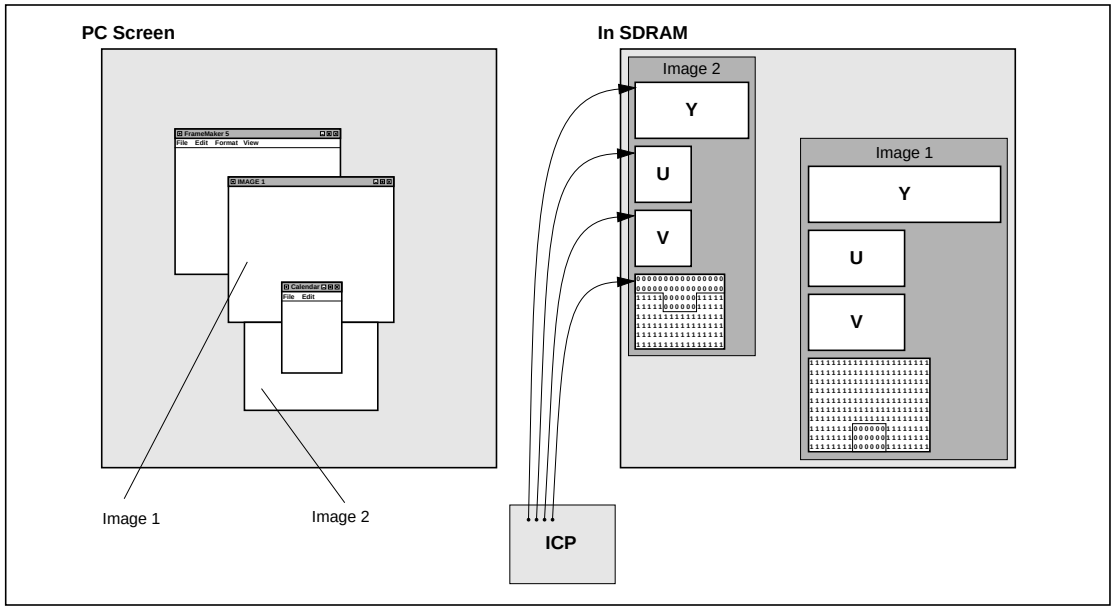


Figure 2-3. ICP operation. Windows on the PC screen and data structures in SDRAM for two live video windows.

shape of the visible area of the destination window in the frame buffer. When the arrangement of windows on the PC screen is changed, modifications to the occlusion bit maps are performed by TM1000 or host resident software.

It is important to note that there is no preset limit on the number and sizes of windows that can be handled by the ICP. The only limit is the available bandwidth. Thus, the ICP can handle a few large windows or many small windows. The ICP can sustain a transfer rate of 50 megapixels per second, which is more than enough to saturate PCI when transferring images to video frame buffers.

2.4.6 Variable-Length Decoder (VLD)

The variable-length decoder (VLD) is included to relieve the DSPCPU of the task of decoding Huffman-encoded video data streams. It can be used to help decode MPEG-1 and MPEG-2 video streams. The lower bitrate of video-conferencing can be adequately handled by DSPCPU software without co-processor.

The VLD is a memory-to-memory coprocessor. The DSPCPU hands the VLD a pointer to a Huffman-encoded bit stream, and the VLD produces a tokenized bit stream that is very convenient for the TM1000 image decompression software to use. The format of the output token stream is optimized for the MPEG-2 decompression software so that communication between the DSPCPU and VLD is minimized.

As with the other processing-intensive coprocessors, the VLD is included mainly to relieve the DSPCPU of a task that wastes its performance potential. When dealing with the high bit rates of MPEG-2 data streams, too much of the DSPCPU's time is devoted to this task, which prevents its special capabilities from being used.

2.4.7 Audio-In and Audio-Out Units

The audio-in and audio-out units are similar to the video units. They connect to most serial ADC and DAC chips, and are programmable enough to handle most reasonable protocols. These units can transfer MSB or LSB first and left or right channel first.

The sampling clock is driven by TM1000 and is software programmable within a wide range from DC to 100 kHz with a resolution of 0.07 Hz. The clock circuit allows the programmer subtle control over the sampling frequency so that audio and video synchronization can be achieved in any system configuration. When changing the frequency, the instantaneous phase does not change, which allows frequency manipulation without introducing distortion.

As with the video units, the audio-in and audio-out units buffer incoming and outgoing audio data in SDRAM. The audio-in unit buffers samples in either eight- or 16-bit format, mono or stereo. The audio-out unit simply transfers sample data from memory to the external DAC; any manipulation of sound data is performed by the DSPCPU since this processing will require at most a few percent of the its processing capacity.

2.4.8 Synchronous Serial Interface

The on-chip synchronous serial interface is specially designed to interface to high integration Analog Modem frontends or ISDN frontend devices. In the analog mo-

dem case, all of the modem signal processing is performed in the TM1000 DSPCPU.

2.4.9 I²C Interface

I2C is a 2 wire multi-master, multi-slave interface capable of transmitting up to 400 kbit/sec. TM1000 implements a I2C master only. This allows TM1000 to configure and inspect status of the peripheral video devices, such as video decoders, video encoders and some camera types.

by Gert Slavenburg, Marcel Janssens

3.1 BASIC ARCHITECTURE CONCEPTS

This section documents the system-programmer or 'bare-machine' view of the TM1000 microprocessor core, also known as the DSPCPU.

3.1.1 Register Model

Figure 3-1illustrates the DSPCPU registers. The DSPCPU provides 128 general purpose registers, named r0..r127. In addition to the hardware program counter PC, there are 4 user-accessible special purpose registers, PCSW, DPC, SPC, and CCCOUNT. Table 3-1 lists the registers and their purposes.

Register r0 always contains the integer value '0', register r1 always contains the integer value '1'. Note that this also corresponds to r0 containing the boolean value 'FALSE' or the single precision floating point value +0.0 and r1 containing 'TRUE'. The programmer is NOT allowed to write to r0 or r1.

> **Note:** Writing to r0 or r1 may cause reads from r0 or r1 scheduled in adjacent clock cycles to return unpredictable values. The standard assembler prevents/forbids the use of r0 or r1 as a destination register.

Registers r2 through r127 are true general purpose registers; the hardware does not in any way imply their use, although compiler or programmer conventions may assign particular roles to particular registers. The DPC (Destination Program Counter) and SPC (Source Program Counter) relate to interrupt and exception handling and are treated in Section 3.1.4, "SPC and DPC—Source and Destination Program Counter." The PCSW (Program Control and Status Word) is treated in Section 3.1.3, "PCSW Overview." CCCOUNT, the 64 bit clock cycle counter is treated in Section 3.1.5, "CCCOUNT—Clock Cycle Counter."

Table 3-1. DSPCPU Registers

Register	Size	Details
r0	32 bits	Always reads as 0x0; must not be used as destination of operations
r1	32 bits	Always reads as 0x1; must not be used as destination of operations
r2–r127	32 bits	126 general-purpose registers
PC	32 bits	Program counter
PCSW	32 bits	Program Control & Status Word
DPC	32 bits	Destination program counter; latches target of taken branch that is interrupted
SPC	32 bits	Source program counter; latches target of taken branch that is not interrupted
CCCOUNT	64 bits	Counts clock cycles since reset

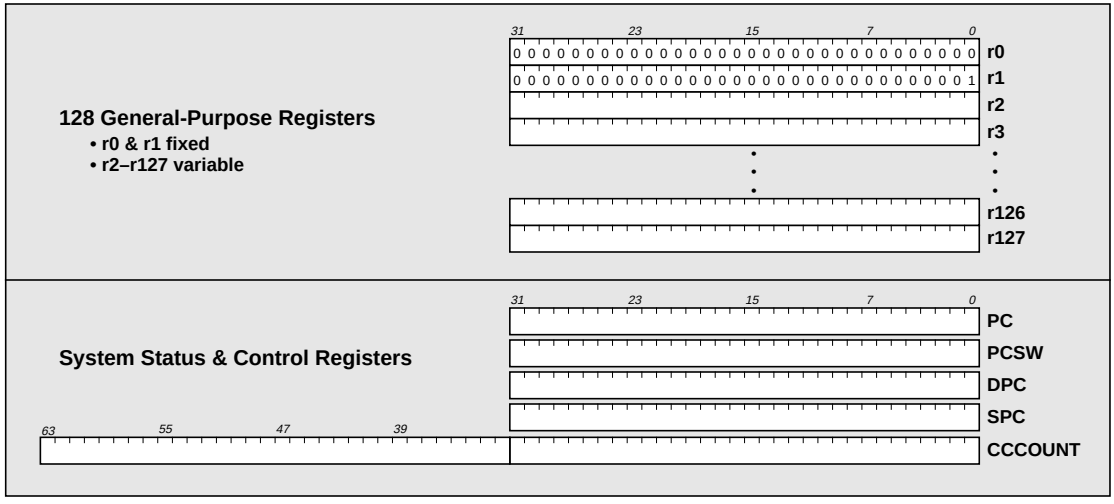


Figure 3-1. TM1000 registers.

3.1.2 Basic TM1000 Execution Model

The DSPCPU issues one 'long instruction' every clock cycle. Each instruction consists of several operations (five operations for the TM1000 microprocessor). Each operation is comparable to a RISC machine instruction, except that the execution of an operation is conditional upon the content of a general purpose register. Examples of operations are:

```
IF r10 iadd r11 r12 → r13
```
(if r10 true, add r11 and r12 and write sum in r13)

```
IF r10 ld32d(4) r15 → r16
```
(if r10 true, load 32 bits from mem[r15+4] into r16)

```
IF r20 jmpf r21 r22
```
(if r20 true and r21 false, jump to address in r22)

Each operation has a specific, known execution time (in clock cycles). For example, iadd takes 1 cycle. This means that the result of an iadd operation started in clock cycle *i* is available for use as an argument to operations issued in cycle *i+1* or later. The other operations issued in cycle *i* cannot use the result of iadd. The ld32d operation takes 3 cycles. The result of an ld32d operation started in cycle *j* is available for use by other operations in cycle *j+3* or later. Branches, such as the jmpf example above have three delay slots. This means that if a branch operation in cycle *k* is taken, all operations in the instructions in cycle *k+1, k+2* and *k+3* are still executed.

In the above examples, r10 and r20 control the conditional execution of the operations. This is also referred to as 'guarding', where r10 and r20 contain the 'guard' of the operation. See Section 3.2.1, "Guarding (Conditional Execution)."

Certain restrictions exist in the choice of what operations can be packed into an instruction. For example, the DSPCPU in TM1000 allows no more than two load/store class operations to be packed into a single instruction. Also, no more than five results (of previously started operations) can be written during any one cycle. The packing of operations is not normally done by the programmer. Instead, the *instruction scheduler* (Trimedia Programmer's Manual) takes care of converting the parallel intermediate format code into packed instructions ready for the assembler. The rules are formally described in the *machine description file* (Trimedia Programmer's Manual, Appendix C) used by the instruction scheduler and other tools.

3.1.3 PCSW Overview

Figure 3-2 shows the PCSW (Program Control and Status Word) register. The value of PCSW on reset is 0. For compatibility, any undefined PCSW fields should never be modified.

Note that the DSPCPU architecture has no integer arithmetic status flags. Integer operations that generate out-of-range results deliver an operation specific bit pattern. For example, see dspiadd in Appendix A, "DSPCPU Operations." Predicate operations exist that take the place of status flags in a classical architecture. Multiword arithmetic is supported by the 'carry' operation, which generates a zero or one depending on the carry that would be generated if its arguments were summed.

FP-Related Fields. The IEEE mode field determines the IEEE rounding mode of all floating point operations, with the exception of a few floating point conversion operations that use fixed rounding mode. For example, see ifixrz, ifloatrz, ifixrz, ifloatrz in Appendix A, "DSPCPU Operations."

The *FP exception flags* are 'sticky bits' that get set as a side effect of floating-point computations. Each floating point operation can set one or more of the flags if it incurs the corresponding exception. The flags can only be reset by direct software manipulation of the PCSW (using the writepcsw operation). The bits have the meanings shown in Table 3-2.

The *FP exception trap enable bits* determine which FP exception flags invoke CPU exception handling. An exception is requested if the intersection of the exception flags and trap enable flags is non-zero. The acceptance and handling of exceptions is described in Section 3.4, "Special Event Handling."

BSX (Bytesex). The DSPCPU has a switchable bytesex. The BSX flag in the PCSW can be written by software. Load/store operations observe little- or big-endian byte ordering based on the current setting of BSX.

IEN (Interrupt Enable). The IEN flag disables or enables interrupt processing for most interrupt sources. Only NMI (non maskable interrupt) bypasses IEN. The acceptance

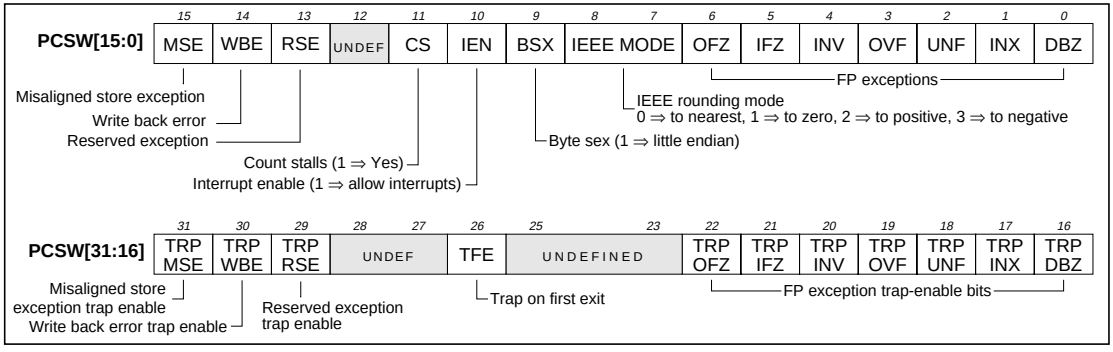


Figure 3-2. TM1000 PCSW (Program Control and Status Word) register format.

Table 3-2. PCSW FP Exception Flag Definitions

Flag	Function
INV	Standard IEEE invalid flag
OVF	Standard IEEE overflow flag
UNF	Standard IEEE underflow flag
INX	Standard IEEE inexact flag
DBZ	Standard IEEE divide-by-zero flag
OFZ	“Output flushed to zero;” set if an operation caused a denormalized result
IFZ	“Input flushed to zero;” set if an operation was applied to one or more denormalized operands

and handling of interrupts is described in Section 3.4.3, “INT and NMI (Maskable and Non-Maskable Interrupts).”

CS (Count Stalls). The CS flag determines the mode of CCCOUNT, the 64 bit clock cycle counter. If CS = ‘1’, the cycle counter increments on stall cycles as well as on normal cycles. If CS = ‘0’, the clock cycle counter only increments on non-stall cycles. See also Section 3.1.5, “CCCOUNT—Clock Cycle Counter.”

MSE and TRPMSE (Misaligned-Store Exception). The MSE bit will be set when the processor detects a store operation to an address that is not aligned. For example, a 32-bit store executed with an address that is not a multiple of four will cause MSE to be set. The TRPMSE bit enables the DSPCPU to raise misaligned address exceptions. An exception is requested if the intersection of MSE and TRPMSE is non-zero. The acceptance and handling of exceptions is described in Section 3.4, “Special Event Handling.”

Unaligned load operations do not cause an exception, because load operations can be speculative (i.e. their result is thrown away).

When the DSPCPU generates an unaligned address, the low order address bit(s) (one bit in the case of a 16-bit load, two bits for a 32-bit load) are forced to zero and the load/store is executed from this aligned address.

WBE and TRPWBE (Write Back Error). The WBE flag will be set whenever a program attempts to write back more than 5 results simultaneously. This is indicative of a programming error, likely caused by the scheduler or assembler. The TRPWBE bit enables the corresponding exception.

RSE, TRPRSE (Reserved Exception). RSE and TRPRSE are reserved for diagnostic purposes and not described here.

TFE (Trap on First Exit). The TFE bits is a support bit for the debugger. The TFE bit is set by the debugger prior to taking a (non-interruptible) jump to the application program. On the next interruptible jump (the first interruptible jump in the application being debugged), an exception is requested because the TFE bit is set. The acceptance and handling of exception processing is described in Section 3.4, “Special Event Handling.”

Corner-case note: Whenever a hardware update (e.g. an exception being raised) and a software update (through writepcsw) of the PCSW coincide, the new value of the PCSW will be the value that is written by the writepcsw instruction, except for those bits that the hardware is currently updating (which will reflect the hardware value).

3.1.4 SPC and DPC—Source and Destination Program Counter

The SPC and DPC registers are support registers for exception processing. The DPC is updated during every interruptible jump with the target address of that interruptible jump. If an exception is taken at an interruptible jump, the value in the DPC register can be used by the exception handling routine as the return address to resume the program at the place of interruption.

The SPC register is updated during every interruptible jump that is not interrupted by an exception. Thus on an interrupted interruptible jump, the SPC register is not updated. The SPC register allows the exception handling routine to determine the start address of the decision tree (a block of uninterruptible, scheduled TM1000 code) that was executing when the exception was taken (see also Section 3.4, “Special Event Handling”).

Corner-case note: Whenever a hardware update (during an interruptible jump) and a software update (through writedpc or writespc) coincide, the software update takes precedence.

3.1.5 CCCOUNT—Clock Cycle Counter

CCCOUNT is a 64 bit counter that counts clock cycles since RESET. Cycle counting can occur in two modes, depending on PCSW.CS. If PCSW.CS = ‘1’, the cycle count increments on stall cycles and normal cycles. If PCSW.CS = ‘0’, the clock cycle count only increments on non-stall cycles.

CCCOUNT is implemented as a master counter/slave register pair. The master 64-bit counter gets updated continuously. The value of the CCCOUNT slave register is updated with the current master cycle count during successful interruptible jumps only. The *cycles* and *hicycles* DSPCPU operations return the content of the 32 LSBs and 32 MSBs, respectively, of the slave register. This ensures that the value returned by hicycles and cycles is coherent, as long as there is no intervening interruptible jump, which makes these operations suitable for 64 bit high resolution timing from C source code programs. The *curcycles* DSPCPU operation returns the 32 LSBs of the master counter. The latter operation can be used for instruction cycle precise timing. When used, it must - of course - be precisely placed, probably at the assembly code level.

3.1.6 Boolean Representation

The bit pattern generated by boolean valued operations (ileq, fleq etc.) is '00...00' (FALSE) or '00...01' (TRUE). When interpreting a bit pattern as a boolean value, only the LSB is taken into account, i.e. 'xx..x0' is interpreted as FALSE and 'xx..x1' is interpreted as TRUE. In particular, wherever a general purpose register is used as a ‘guard’, the LSB determines whether execution of the guarded operation takes place.

3.1.7 Integer Representation

The architecture supports the notion of 'unsigned integers' and 'signed integers.' Signed integers use the standard two's-complement representation.

Arithmetic on integers does not generate traps. If a result is not representable, the bit pattern returned is operation specific, as defined in the individual operation description section. The typical cases are:

- Wrap around for regular add- and subtract-type operations.
- Clamping against the minimum or maximum representable value for DSP-type operations.
- Returning the least significant 32-bit value of a 64-bit result (e.g., integer/unsigned multiply).

3.1.8 Floating Point Representation

The 32-bit version of the TM1000 architecture supports only single precision (32-bit) IEEE-754 floating point arithmetic. On future 64-bit implementations of the architecture, both single and double precision (32- and 64-bit) IEEE-754 floating point will be supported.

All arithmetic conforms to the IEEE-754 standard in flush-to-zero mode.

Most floating point compute operations round according to the current setting of the *PCSW IEEE mode* field. The current setting of the field determines result rounding (to nearest, to zero, to positive infinity, to negative infinity). Conversions from float to integer/unsigned are available in two forms: a rounding-mode-observing form and an ANSI-C-specific-rounding form. The ANSI-C-specific form forces round to zero regardless of the IEEE rounding mode. Conversion from integer/unsigned to float always observes the IEEE rounding mode.

Floating point exceptions are supported with two mechanisms. Each individual floating point operation (e.g. fadd) has a counterpart operation (faddflags) that computes the exception flag values. These operations can be used for precise exception identification[1]. The second mechanism uses the 'sticky' exception bits in the PCSW that collect aggregate exception events. The PCSW exception bits can selectively invoke CPU exception handling. See Section 3.4.2, "EXC (Exceptions)."

The following representation choices were made in TM1000's floating point implementation:

Table 3-3. Special Float Value Representation

Item	Representation
+inf	0x7f800000
-inf	0xff800000
self generated qNaN	0xffffffff
result of operation on any NaN argument	argument | 0x00400000 (forcing the NaN to be quiet)
signalling NaN	never generated by TM1000, accepted as per IEEE-754

3.1.9 Addressing Modes

The addressing modes shown in Table 3-4 are supported by the DSPCPU architecture (store operations allow only displacement mode).

In these addressing modes, R[i] indicates one of the general purpose registers. The scale factor applied (1/2/4) is equal to the size of the item loaded or stored, i.e. 1 for a byte operation, two for a 16-bit operation and four for a 32-bit operation. The range of valid 'i', 'j' and 'k' values may differ between implementations of the architecture; the minimum values for implementation-dependent characteristics are shown in Table 3-5.

Table 3-4. Addressing Modes

Mode	Suffix	Load? Store?	Name
R[i] + scaled(#j)	d	Load & Store	Displacement
R[i] + R[k]	r	Load only	Index
R[i] + scaled(R[k])	x	Load only	Scaled index

Table 3-5. Minimum Values for Implementation-Dependent Addressing Mode Components

Parameter	Minimum Range
'i' and 'k'	0..127 (i.e., each implementation has at least 128 registers)
'j'	-64..63 (i.e., displacements will be at least 7 bits long and signed)

Note that the assembly code specifies the true displacement, and not the value to be scaled. For example 'ld32d(–8) r3' loads a 32 bit value from address (r3 – 8). This is encoded in the binary operation pattern as a –2 in the seven-bit field by the assembler. At runtime, the scale factor four is applied to reconstruct the intended displacement of –8.

3.1.10 Software Compatibility

The DSPCPU architecture expressly does not support binary compatibility between family members, however it is possible to distribute pseudocode (so-called '.t' files) that can be mapped by the instruction scheduler to any processor of the Trimedia family. The ANSI C compiler ensures that all family members are compatible at the source-code level.

3.2 INSTRUCTION SET OVERVIEW

3.2.1 Guarding (Conditional Execution)

In the TM1000 architecture, all operations are optionally 'guarded'. A guarded operation executes conditionally,

1. This mechanism allows precise exception identification in the context of our multi-issue microprocessor core—where many floating point operations may issue simultaneously or speculatively—at the expense of additional operations generated by the compiler.

depending on the value in the ‘guard' register. For example, a guarded add is written as:

```
IF R23 iadd R14 R10 → R13
```

This should be taken to mean

if R23 then R13 ← R14 + R10.

The 'if R23' clause controls the execution of the operation based on the LSB of R23. Hence, depending on the LSB of R23, R13 is either unchanged or set to contain the integer sum of R14 and R10.

Guarding applies to all DSPCPU operations, except the iimm and uimm (load-immediate) operations. Guarding controls the effect on all programmer visible state of the system, i.e. register values, memory content and device state.

3.2.2 Load and Store Operations

Memory is byte addressable. Loads and stores have to be ‘naturally aligned’, i.e. a 16-bit load or store must target an address that is a multiple of two. A 32-bit load or store must target an address that is a multiple of four. The BSX bit in the PCSW determines the byte order of loads and stores. For example, see ld32 and st32 in Appendix A, “DSPCPU Operations.”

Only 32-bit load and store operations are allowed to access MMIO registers in the MMIO address aperture (see Section 3.3, “Memory and MMIO”). The results are undefined for other loads and stores. The state of the BSX bit has no effect on the result of MMIO accesses.

Loads are allowed to be issued speculatively. Loads that are outside the range of valid data memory addresses for the active process return an implementation dependent value and do not generate an exception. Misaligned loads also return an implementation dependent value and do not generate an exception.

If a pair of memory operations involves one or more common bytes in memory, the effect on the common bytes is as defined in Table 3-6.

Table 3-6. Behavior of Loads and Stores with Coincident Addresses

Condition	Behavior
$T_{store} < T_{load}$	If a store is issued before a load, the value loaded contains the new bytes.
$T_{load} < T_{store}$	If a load is issued before a store, the value loaded contains the old bytes.
$T_{store1} < T_{store2}$	If store1 is issued before store2, the resulting value contains the bytes of store2.
$T_{store} = T_{load}$	If a load and store are issued in the same clock cycle, the result is UNDEFINED.
$T_{store1} = T_{store2}$	If two stores are issued in the same clock cycle, the resulting stored value is undefined.

The addressing modes supported are shown in Table 3-4 and the minimum values of implementation-dependent addressing-mode components are shown in Table 3-5.

Note: The index and scaled-index modes are not allowed with store opcodes, due to the hardware restriction that each operation have at most two source operand registers and 1 condition register—stores use one operand register for the value to be stored, which leaves only one register to form an the address.

The scale factor applied (1/2/4) in the scaled addressing modes is equal to the size of the item loaded or stored, i.e. 1 for a byte operation, 2 for a 16-bit operation and 4 for a 32-bit operation.

Table 3-7 lists the available load and store mnemonics for the three addressing modes.

Table 3-7. Load and Store Mnemonics

Operation	Displacement	Index	Scaled-Index
8-bit signed load	ild8d	ild8r	—
8-bit unsigned load	uld8d	uld8r	—
16-bit signed load	ild16d	ild16r	ild16x
16-bit unsigned load	uld16d	uld16r	uld16x
32-bit load	ld32d	ld32r	ld32x
8-bit store	st8d	—	—
16-bit store	st16d	—	—
32-bit store	st32d	—	—

Example usage of load and store operations:

```
IF r10 ild16d(12) r12 → r13
```

If the LSB of r10 is set, load 16 bits starting at address (r12+12) using the byte ordering indicated in PCSW.BSX, sign-extend the value to 32 bits and store the result in r13.

```
IF r10 st32d(40) r12 r13
```

if the LSB of r10 is set, store the 32-bit value from r13 to the address (r12+40) using the byte ordering indicated in PCSW.BSX.

3.2.3 Compute Operations

Compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register.

Immediate Operations. Immediate operations load an immediate constant (specified in the opcode) and produce a result in the destination register.

Floating-Point Compute Operations. Floating-point compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register. Unless otherwise mentioned all floating point operations observe the rounding mode bits defined in the PCSW register. All floating-point operations not ending in "flags" update the PCSW exception flags. All

operations ending in "flags" compute the exception flags as if the operation were executed and return the flag values (in the same format as in the PCSW); the exception flags in the PCSW itself remain unchanged.

Multimedia Operations. These special compute operations are like normal compute operations, but the specified operations are not usually found in general purpose CPU's. These operations provide special support for multi-media applications.

3.2.4 Special-Register Operations

Special register operations operate on the special registers: PCSW, DPC, SPC and CCCOUNT.

3.2.5 Control-Flow Operations

Control-flow operations change the value of the program counter. Conditional jumps test the value in a register, and based on this value, change the program counter to the address contained in a second register or continue execution with the next instruction. Unconditional jumps always change the program counter to the specified immediate address.

Control-flow operations can be interruptible or non-interruptible. The execution of an interruptible jump is the only occasion where the TM1000 allows special event handling to take place (see Section 3.4, "Special Event Handling").

3.3 MEMORY AND MMIO

TM1000 defines four apertures in its 32-bit address space: the memory hole, the DRAM aperture, the MMIO aperture and the PCI apertures (See Figure 3-3).The memory hole covers addresses 0..0xff. A data read from the hole in the default operating mode returns 0. The DRAM and MMIO apertures are defined by the values in MMIO registers; the PCI apertures consist of every address that does not fall in the other three apertures.

3.3.1 Memory Map

DRAM is mapped into an aperture extending from the address in DRAM_BASE to the address in DRAM_LIMIT. The maximum DRAM aperture size is 64 MB.

The MMIO aperture is located at address MMIO_BASE and is fixed 2 MB in size.

In the default operating mode, all memory accesses not going to either the hole, DRAM or MMIO space are interpreted as PCI accesses. This behavior can be overridden as described in Section 5.3.8, "Memory Hole and PCI Aperture Disable."

The MMIO aperture and the DRAM aperture can be at any naturally aligned location, in any order, but should not overlap; if they do, the consequences are undefined (i.e. the processor may deadlock). The values of DRAM_BASE, DRAM_LIMIT, and MMIO_BASE are set during the boot process. In the case of a PCI host assisted boot, the values are determined by the host BIOS. In case of stand-alone boot (i.e., TM1000 is the PCI host), the values are taken from the boot ROM. Refer to Chapter 12, "System Boot" for details.

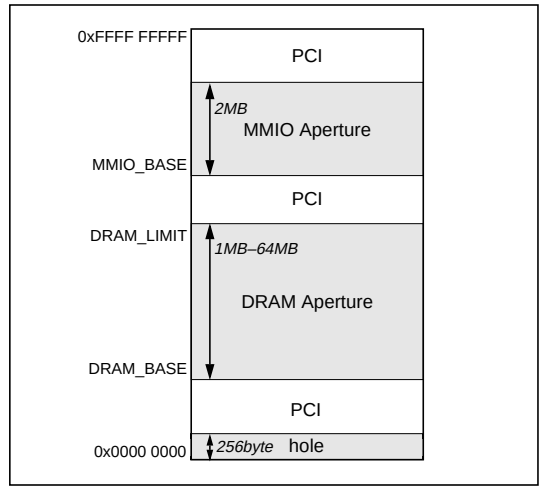


Figure 3-3. TM1000 Memory Map.

3.3.2 The Memory Hole

The memory hole from address 0 to 0xff serves to protect the system from performance loss due to speculative loads. Due to the nature of C program references, most speculative loads issued by the DSPCPU fall in the range covered by the hole. The hole, which is activated by default upon RESET, serves to ensure that these speculative loads do NOT cause PCI read accesses and slow down the system. The value returned by any data load from the hole is 0. The hole only protects loads. Store operations in the hole do cause writes to PCI, SDRAM or MMIO as determined by the aperture base address values.

The hole can be temporarily disabled through the DC_LOCK_CTL register. This is described in Section 5.3.8, "Memory Hole and PCI Aperture Disable."

3.3.3 MMIO Memory Map

Devices are controlled through memory-mapped device registers, referred to as MMIO registers. Devices can autonomously access data memory and can cause CPU interrupts.

The MMIO aperture is 2 MB in size and initially located at address 0xEFE00000 on RESET; it is relocated by the PCI BIOS for PC hosted TM1000 boards; its final location is determined by the boot EEPROM for stand-alone systems. See Chapter 12, "System Boot" for more information. Figure 3-4 gives a detailed overview of the MMIO memory map (addresses used are offsets with respect to the MMIO base). The operating system on TM1000 can change MMIO_BASE by writing to the MMIO_BASE MMIO location. User programs should not attempt this. Refer to the Trimedia programming guide

for safe ways to access the device registers from programs.

Only 32-bit load and store operations are allowed to access MMIO registers in the MMIO address aperture. The results are undefined for other loads and stores. The state of the PCSW BSX bit has no effect on the result of MMIO accesses.

The EXCVEC MMIO location is explained in Section 3.4.2, “EXC (Exceptions).” Section 3.4.3, “INT and NMI (Maskable and Non-Maskable Interrupts),” describes the locations that deal with the setup and handling of interrupts: ISETTING, IPENDING, ICLEAR, IMASK and the interrupt vectors. The timer MMIO locations are described in Section 3.5, “TM1000 Host Interrupts.” The instruction and data breakpoint are described in Section 3.7, “Debug Support.” The MMIO locations of each device are treated in the respective device chapters.

3.4 SPECIAL EVENT HANDLING

The TM1000 microprocessor responds to the special events shown in Table 3-8, ordered by priority.

With the exception of RESET, which is enabled at all times, the architecture of the DSPCPU allows special event handling to begin only during an *interruptible jump* operation (ijmpt, ijmpf or ijmpi) that succeeds (i.e., is a taken jump). EXC, NMI and INT handling can be initiated during handling of an EXC or an INT, but *only* during successful interruptible jumps.

Table 3-8. Special Events and Event Vectors

Event	Vector
RESET	(Highest priority) vector to DRAM_BASE
EXC	(All exceptions) vector to EXCVEC (programmable)
NMI, INT	(Non-maskable interrupt, maskable interrupt) use the programmed vector (one of 32 vectors depending on the interrupt source)

The *instruction scheduler* uses interruptible jumps exclusively for inter-decision tree jumps. Hence, within a decision tree, no special-event processing can be initiated. If a tree-to-tree jump is taken, special-event processing is allowed. Since the only registers live at this point (i.e., that contain useful data) are the *global registers* allocated by the ANSI C compiler, only a subset of the registers needs to be preserved by the event handlers. Refer to the Trimedia Programmer’s Reference Manual to find details on which registers can be in use. The DSPCPU register state can be described by the contents of this subset of the general purpose registers and the contents of the PCSW and the DPC (Destination Program Counter) value (the target of the inter-tree jump).

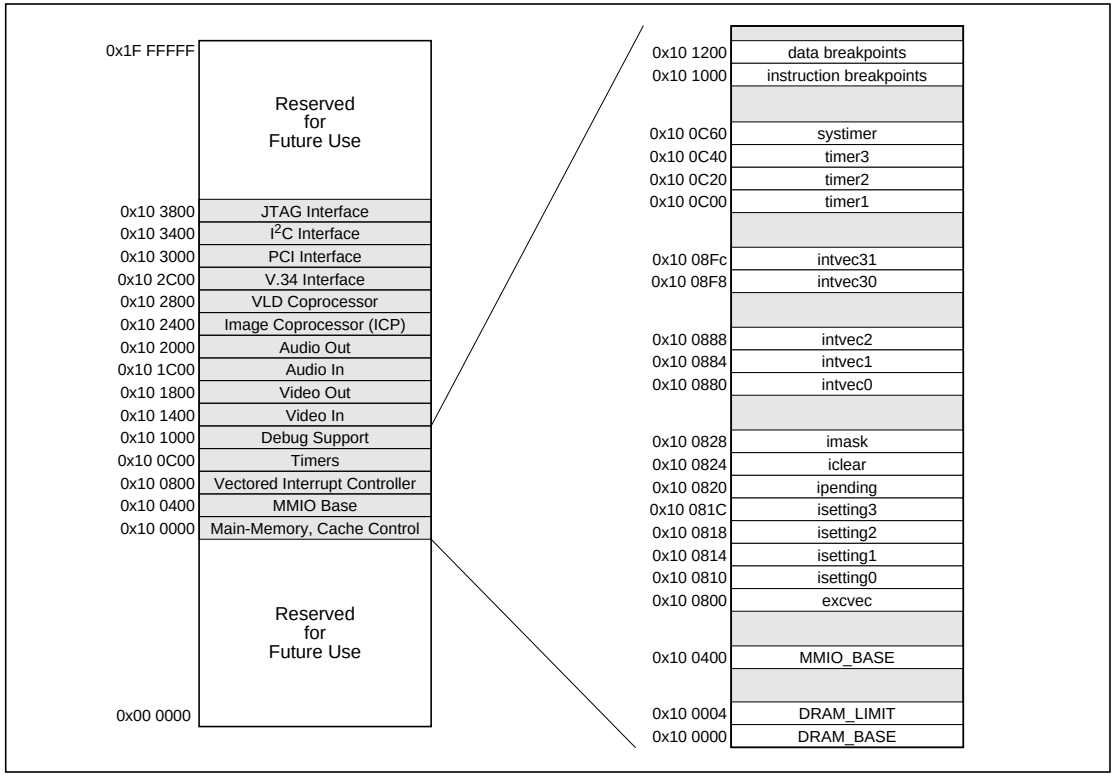


Figure 3-4. Memory map of MMIO address space (addresses are offset from MMIO_BASE).

The priority resolution mechanism built into the DSPCPU hardware dispatches the highest-priority non-masked special event request at the time of a successful interruptible jump operation. In view of the simple, real-time-oriented nature of the mechanisms provided, only limited nesting of events should be allowed.

3.4.1 RESET

RESET is the highest priority special event. It is asserted by external hardware. TM1000 will respond to it at any time. In response to reset assertion, the boot protocol is executed. This causes (a.o.) the current PC value to be lost and instruction execution to start from address DRAM_BASE.

3.4.2 EXC (Exceptions)

The DSPCPU enters EXC special-event processing under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is non-empty or PCSW.TFE is set.
3. A successful interruptible jump is in the final jump execution stage.

DSPCPU hardware takes the following actions on the initiation of EXC processing:

1. DPC gets assigned the intended destination address of the successful jump.
2. Instruction processing starts at EXCVEC.

All other actions are the responsibility of the EXC handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

3.4.3 INT and NMI (Maskable and Non-Maskable Interrupts)

The on-chip Vectored Interrupt Controller (VIC) provides 32 INT request input hardware lines. The interrupt controller prioritizes and maps attention requests from several different peripherals onto successive INT requests to the DSPCPU.

INT special event processing will occur under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is empty and PCSW.TFE is not set.
3. The intersection of IPENDING and IMASK is non-empty.
4. The interrupt is at level NMI or PCSW.IEN = 1.
5. A successful interruptible jump is in the final jump execution stage.

DSPCPU hardware takes the following actions on the initiation of NMI or INT processing:

1. DPC gets assigned the intended destination address of the successful jump.
2. Instruction processing starts at the appropriate interrupt vector.

All other actions are the responsibility of the INT handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

3.4.3.1 Interrupt Vectors

Each of the 32 interrupt sources can be assigned an arbitrary interrupt vector (the address of the first instruction of the interrupt handler). A vector is setup by writing the address to one of the MMIO locations shown in Figure 3-5. The state of the MMIO vector locations is undefined after RESET. (Addresses of the MMIO vector registers are offset with respect to MMIO_BASE.)

Programmer's note: Please see the Trimedia Programmer's Reference Manual for information on writing interrupt handlers.

3.4.3.2 Interrupt Modes

DSPCPU interrupt sources can be programmed to operate in either *level-sensitive* or *edge-triggered* mode. Operation in edge-triggered or level-sensitive mode is determined by a bit in the ISETTING MMIO locations corresponding to the source, as defined in Figure 3-6. On RESET, all ISETTING registers are cleared.

In edge-triggered mode, the leading edge of the signal on the device interrupt request line causes the VIC (Vectored Interrupt Controller) to set the *interrupt pending* flag corresponding to the device source number. Note that, for active high signals, the leading edge is the positive edge, whereas for active low request signals (such as PCI INTA#), the negative edge is the leading edge. The interrupt remains pending until one of two events occurs:

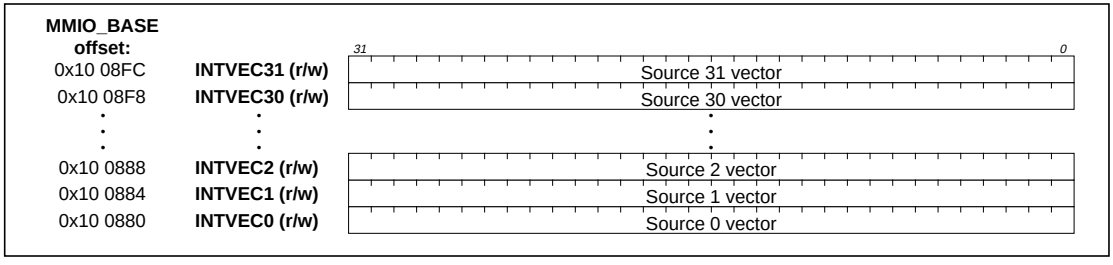


Figure 3-5. Interrupt vector locations in MMIO address space.

- The VIC successfully dispatches the vector corresponding to the source to the TM1000 CPU, or
- TM1000 CPU software clears the interrupt-pending flag by a direct write to the ICLEAR location.

No interrupt acknowledge to ICLEAR is needed for devices operating in edge-triggered mode. The device itself may need a device specific interrupt acknowledge to clear the requesting condition. Edge-triggered mode is *not recommended* for devices that can signal multiple simultaneous interrupt conditions. The on-chip timers can safely be operated in edge triggered mode, which minimizes interrupt service overhead.

In level-sensitive mode, the device requests an interrupt by asserting the VIC source request line. The device holds the request until the device interrupt handler performs a device interrupt acknowledge. It is highly recommended that all off-chip and on-chip sources, with the exception of the timers, are operated in level sensitive mode.

3.4.3.3 Device Interrupt Acknowledge

All devices capable of generating level-triggered interrupts have interrupt acknowledge bits in their memory mapped control registers for this purpose. An interrupt acknowledge is performed by a store to such control register, with a '1' in the bit position(s) corresponding to the desired acknowledge flags.

Programmers note: the store operation that performs the interrupt acknowledge should be issued at least 2 cycles before the (interruptible) jump that ends an interrupt handler. This ensures that the same interrupt is not dispatched twice due to request de-assertion clock delays.

3.4.3.4 Interrupt Priorities

Each interrupt source can be programmed to request one out of eight levels of priorities. The highest priority level (level seven) corresponds to requesting an NMI—an interrupt that cannot be masked by the DSPCPU PCSW.IEN bit. The other levels request regular interrupts, that can be masked as a group by the PCSW.IEN flag. Level six represents the highest priority normal interrupt level and level zero represents the lowest. Refer to Figure 3-6 for details of programming the priority level.

The VIC arbitrates the highest-priority pending interrupt requestor. Sources programmed to request at the same level are treated with a fixed priority, from source number zero (highest) to thirty-one (lowest). At such time as the DSPCPUCPU is willing to process special events, the vector of highest priority NMI source will be dispatched. If no NMI is pending, and the DSPCPU allows regular interrupts (PCSW.IEN is asserted), the vector of the highest priority regular source is dispatched. Once a vector is dispatched, the corresponding interrupt pending flag is de-asserted (edge triggered sources only).

3.4.3.5 Interrupt Masking

A single MMIO register (IMASK in Figure 3-7) allows masking of an arbitrary subset of the interrupt sources. Masking applies to both regular as well as NMI level requestors. Masking is used by software to disable unused devices and/or to implement nested interrupt handling. In the latter case, each interrupt handler can stack the old IMASK content for later restoration and insert a new mask that only allows the interrupts it is willing to handle. For level-triggered device handlers, IMASK should also exclude the device itself to prevent repeated handler activation.

Each interrupt source device typically has its own interrupt enable flag(s), that determine whether certain key device events lead to the request of an interrupt. In addition, the PCSW.IEN flag determines whether the DSPCPU is willing to handle regular interrupts. Non maskable interrupts ignore the state of this flag.

All three mechanisms are necessary: the PCSW.IEN flag is used to implement critical sections of code during which the RTOS (Real-Time Operating System) is unable to handle regular interrupts. The IMASK is used to allow full control over interrupt handler nesting. The device interrupt flags set the operational mode of the device.

When RESET is asserted, IPENDING, ICLEAR, and IMASK are set to all zeroes. (MMIO register addresses shown in Figure 3-7 are offset addresses with respect to MMIO_BASE.)

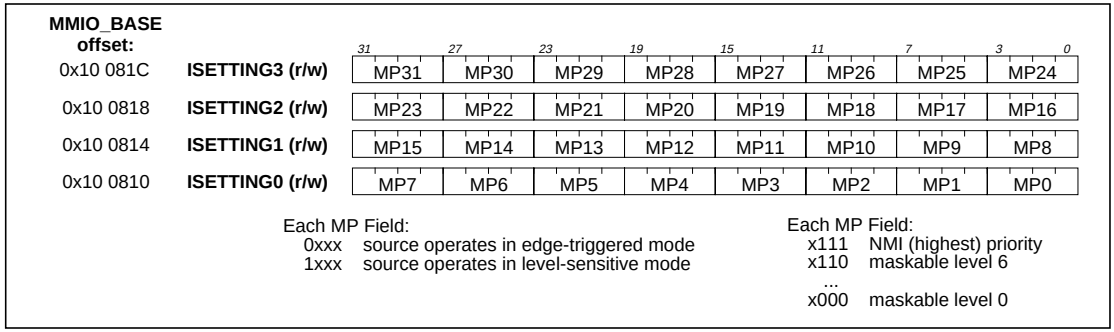


Figure 3-6. Interrupt mode and priority MMIO locations and formats.

3.4.3.6 Software Interrupts and Acknowledgment

The IPENDING register shown in Figure 3-7 can be read to observe the currently pending interrupts. Each bit read depends on the mode of the source:

- For a level-sensitive source, a bit value corresponds to the current state of the device interrupt request line.
- For an edge-triggered interrupt, a '1' is read if and only if an interrupt request occurred and the corresponding vector has not yet been dispatched.

Software can request an interrupt for sources operating in edge-triggered mode. Writes to the IPENDING register assert an interrupt request for all sources where a 1 occurred in the bit position of the written value. Writes have no effect on level-sensitive mode sources. The interrupt request, if not masked, will occur at the next successful interruptible jump. This differs from the conventional software interrupt-like semantics of many architectures. Any of the 32 sources can be requested in software. In normal operation however, software-requested interrupts should be limited to source vectors not allocated for hardware devices. Note that another PCI master can request interrupts by manipulating the IPENDING location in the MMIO aperture. This is useful for inter-processor communication.

The ICLEAR register reads the same as the IPENDING register. Writes to the ICLEAR register serve to clear pending flags for edge-triggered mode sources. All IPENDING flags corresponding to bit positions in which '1's are written are cleared. IPENDING flags corresponding to bit positions in which '0's are written are not affected. Writes have no effect on level-sensitive mode sources. When a pending interrupt bit is being cleared through a write to the ICLEAR register at the same time that the hardware is trying to set that interrupt bit, the hardware takes precedence.

3.4.3.7 NMI Sequentialization

In most applications, it is desirable not to nest NMI's. The NMI interrupt handler can accomplish this by saving the old IMASK content and clearing IMASK before the first interruptible jump is executed by the NMI handler.

Table 3-9. Interrupt Source Assignments

SOURCE NAME	SRC NUM	MODE	SOURCE DESCRIPTION
PCI INTA	0	level	PCI_INTA# pin signal
PCI INTB	1	level	PCI_INTB# pin signal
PCI INTC	2	level	PCI_INTC# pin signal
PCI INTD	3	level	PCI_INTD# pin signal
TRI_USERIRQ	4	either	external general-purpose pin
TIMER1	5	edge	general-purpose timer
TIMER2	6	edge	general-purpose timer
TIMER3	7	edge	general-purpose timer
SYSTIMER	8	edge	reserved for debugger
VIDEOIN	9	level	video in block
VIDEOOUT	10	level	video out block
AUDIOIN	11	level	audio in block
AUDIOOUT	12	level	audio out block
ICP	13	level	image co-processor
VLD	14	level	VLD co-processor
V34	15	level	V.34 interface
PCI	16	level	PCI BIU (DMA, etc.; see Table 10-13 for possible interrupt causes)
IIC	17	level	IIC interface
JTAG	18	level	JTAG interface
t.b.d.	19..27		reserved for future devices
HOSTCOM	28	edge	(software) host communication
APP	29	edge	(software) application
DEBUGGER	30	edge	(software) debugger
RTOS	31	edge	(software) RTOS

3.4.3.8 Interrupt Source Assignment

Table 3-9 shows the assignment of devices to interrupt source numbers, as well as the recommended operating

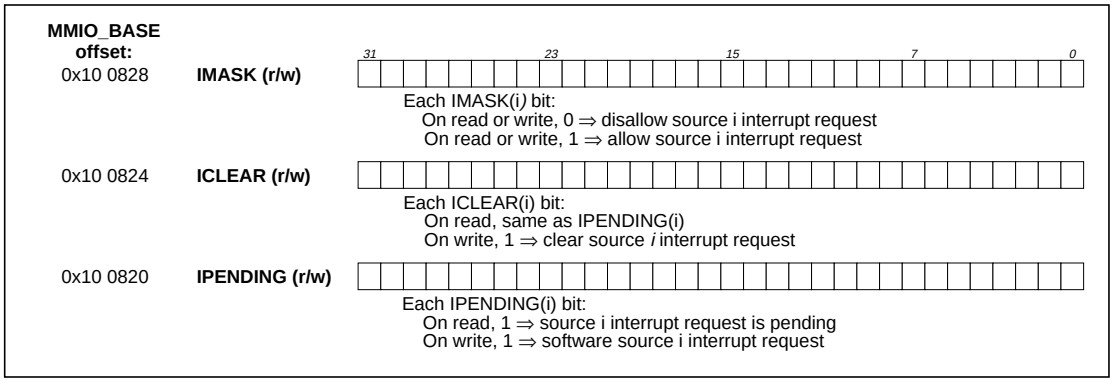


Figure 3-7. Interrupt controller request, clear, and mask MMIO registers.

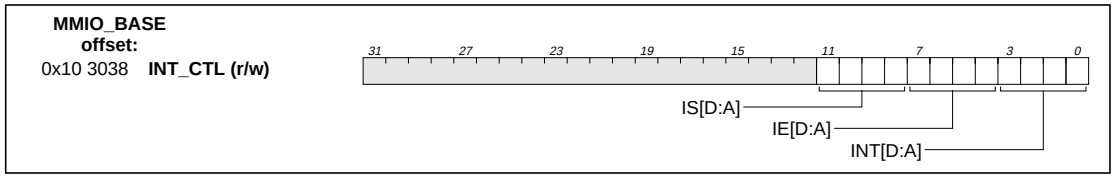


Figure 3-8. Host Interrupt Control register.

mode (edge or level triggered). Note that there are a total of 5 external pins available to assert interrupt requests. The PCI INTA to INTD requests are asserted by active low signal conventions, i.e. a zero level or a negative edge asserts a request. The USERIRQ pin operates with active high signalling conventions.

3.5 TM1000 HOST INTERRUPTS

In systems where TM1000 is operating in the presence of a host CPU on PCI, TM1000 can generate interrupts to the host, using any combination of the four PCI INTA# to INTD# pins. In a typical host system, only one of these pins needs to be wired to the PCI bus interrupt request lines. Any unused pins of this group are then available for use as software programmable I/O pins.

The INT_CTL register (see Figure 3-8) IEx bits, when set, enable the open collector driver of the four INTD#..INTA# pins. The INTx bits determine the output value generated (if enabled). A '1' in INTx causes the corresponding PCI interrupt pin to be asserted (low INTx# pin). The ISx bits can be read and reflect the current active state of the pins, independent of their use as input or output. Note that the actual pins have negative logic (active low) polarity, and are of the open collector output type. Hence the pin voltage is low (active) when the logical value set or seen in the INT_CTL register is a '1'.

The assertion and de-assertion of host interrupts is the responsibility of TM1000 software.

See also Section 10.6.16, "INT_CTL Register."

3.6 TIMERS

The DSPCPU contains four programmable timer/counters. All timer/counters have the same function. The first three (TIMER1, TIMER2, TIMER3) are intended for general use. The fourth timer/counter (SYSTIMER) is reserved for use by the system software and should not be used by applications.

Each timer has three registers as shown in Figure 3-9. The MMIO register addresses shown are offset addresses with respect to the timer's base address (see Figure 3-4).

Each timer/counter can be set to count one of the event types specified in Table 3-10. Note that the DATABREAK event is special, in that the timer/counter may increment by zero, one or two in each clock cycle. For all other event types, increments are by zero or one. The CACHE1 and CACHE2 events serve as cache performance monitoring support. The actual event selected for CACHE1 and CACHE2 is determined by the MEM_EVENTS MMIO register, see Section 5.7, "Performance Evaluation Support." If a TM1000 pin signal (VICLK, etc) is selected as an event, positive-going edges on the signal are counted.

Each timer increments its value until the modulus is reached. On the clock cycle where the incremented value would equal or exceed the modulus, the value wraps around to zero or one (in the case of an increment by two), and an interrupt is generated as defined in Table 3-9. The timer interrupt source mode should be set as edge-sensitive. No software acknowledge to the timer device is necessary.

Counting continues as long as the run bit is set.

Loading a new modulus does not affect the contents of the value register. If a store operation to either the modulus or value register results in value and modulus being the same, no interrupt will be generated. If the run bit is set, the next value will be modulus+1 or modulus+2, and the counter will have to loop around before an interrupt is generated.

A modulus value of zero causes a wrap-around as if the modulus value was 2^{32}.

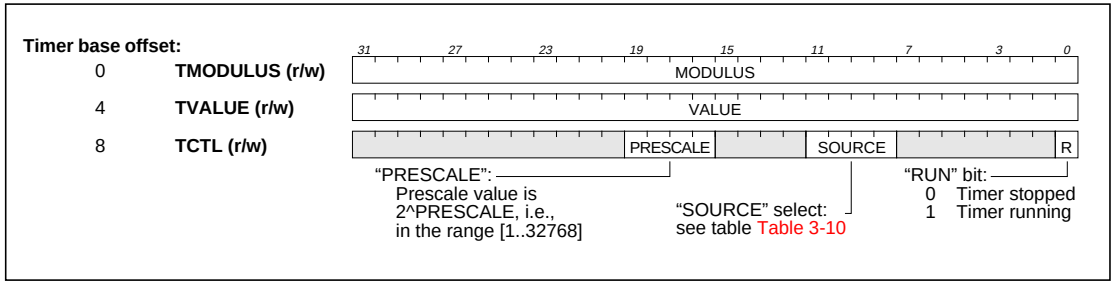


Figure 3-9. Timer register definitions.

Table 3-10. Timer Source Selections

Source Name	Source Bits Value	Source Description
CLOCK	0	CPU clock
PRESCALE	1	prescaled CPU clock
TRI_TIMER_CLK	2	external clock pin
DATABREAK	3	data breakpoints
INSTBREAK	4	instruction breakpoints
CACHE1	5	cache event 1
CACHE2	6	cache event 2
VI-CLK	7	video in clock pin
VO-CLK	8	video out clock pin
AI-WS	9	audio in word strobe pin
AO-WS	10	audio out word strobe pin
V34-RXFSX	11	V34 receive frame sync pin
V34-IO2	12	V34 transmit frame sync pin
—	13-15	undefined

On RESET, the TCTL registers are cleared, and the value of the TMODULUS and TVALUE registers is undefined.

3.7 DEBUG SUPPORT

This section describes the special debug support offered by the DSPCPU. Instruction and data breakpoints can be defined through a set of registers in the MMIO register space. When a breakpoint is matched, an event is generated that can be used as an input clock to a timer (see Section 3.5, “TM1000 Host Interrupts”).

3.7.1 Instruction Breakpoints

The instruction-breakpoint control register is shown in Figure 3-10. On RESET, the BICTL register is cleared. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The instruction-breakpoint address-range registers are shown in Figure 3-11. After RESET, the value of these registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

When the IC bit in the breakpoint control register is set to ‘1’, instruction breakpoints are activated. Any instruction address issued by the TM1000 chip is compared against the low and high address-range values. The IAC bit in the breakpoint control register determines whether the instruction address needs to be inside or outside of the range defined by the low and high address-range registers. A successful comparison takes place when either:

- IAC = ‘0’ and low ≤ iaddr ≤ high, or
- IAC = ‘1’ and iaddr < low or iaddr > high.

On a successful comparison, an instruction breakpoint event is generated, which can be used as a clock input to a timer. After counting the programmed number of instruction breakpoint events, the timer will generate an interrupt request.

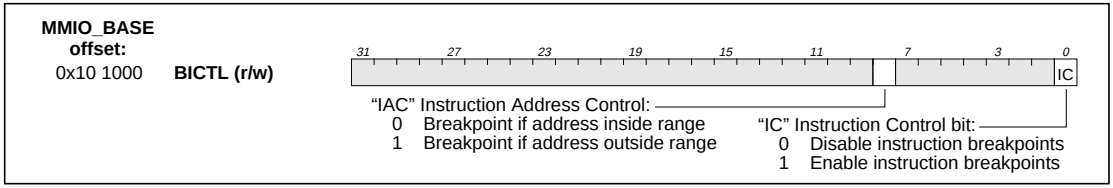


Figure 3-10. Instruction-breakpoint control register.

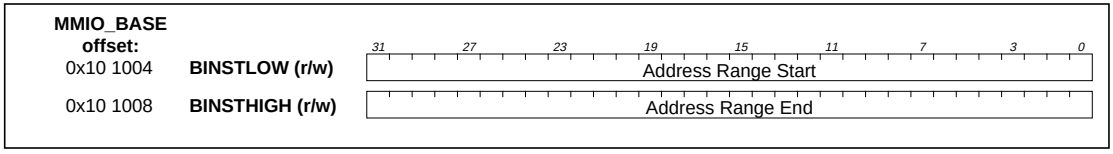


Figure 3-11. Instruction-breakpoint address-range registers.

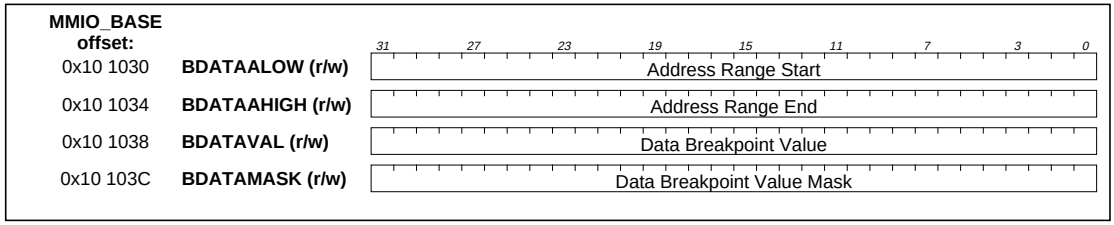


Figure 3-12. Data-breakpoint address-range and value-compare registers.

3.7.2 Data Breakpoints

The data-breakpoint address-range and compare-value registers are shown in Figure 3-12. After RESET, the value of the data breakpoint registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The data-breakpoint control register is shown in Figure 3-13. On RESET, the BDCTL register is cleared. (The register address shown is offset with respect to MMIO_BASE.)

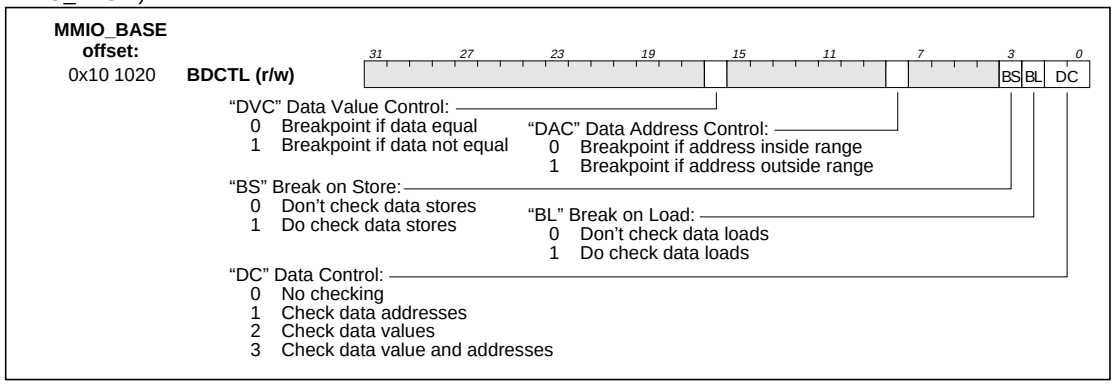


Figure 3-13. Data-breakpoint control register.

When the DC bits in the data breakpoint control register are not set to '0', data breakpoints are activated. When the value of the DC bits is '1' or '3', any data address from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the DSPCPU is compared against the low and high address-range values. The DAC bit in the breakpoint control register determines whether data addresses need to be inside or outside of the range defined by the low and high address-range registers. A successful comparison occurs when either:

- DAC = '0' and low ≤ daddr ≤ high, or
- DAC = '1' and daddr < low or daddr > high.

When the value of the DC bits is '2' or '3', any data value from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the TM1000 CPU is compared against the value in the BDATAVAL register. Only the bits for which the corresponding BDATAMASK register bits are set to '1' will be used in the comparison. The DVC bit in the breakpoint control register determines whether the data value needs to be equal or not equal to the comparison value. A successful comparison occurs when either of the following are true:

- DVC = '0' and (data & BDATAMASK) = (BDATAVAL & BDATAMASK).
- DVC = '1' and (data & BDATAMASK) != (BDATAVAL & BDATAMASK).

Note: use a nonzero datamask or the result is undefined.

When a successful comparison has taken place, a data breakpoint event is generated, which can be used as a clock input to a timer. After counting the set number of data breakpoint events, the timer will generate an interrupt request.

When the value of the DC bits is equal to 3, a data breakpoint event is generated if and only if a successful comparison occurs on both address and data simultaneously.

Note that up to two data breakpoint events can occur per clock cycle, due to the dual load/store capability of the CPU and data cache.

Custom Operations for Multimedia Chapter 4

by Gert Slavenburg, Pieter v.d. Meulen, Yong Cho, Sang-Ju Park

4.1 CUSTOM OPERATION OVERVIEW

Custom operations in the TM1000 CPU architecture are specialized, high-function operations designed to dramatically improve performance in important multimedia applications. When properly incorporated into application source code, custom operations enable an application to take advantage of the highly parallel TM1000 microprocessor implementation. Achieving a similar performance increase through other means—e.g., executing a higher number of traditional microprocessor instructions per cycle—would be prohibitively expensive for TM1000's low-cost target applications.

Custom operations are simple to understand and consistent in their definition, but their unusual functions make it difficult for automatic code generation algorithms to use them effectively. Consequently, custom operations are inserted into source code by the programmer. To make this process as painless as possible, custom operation syntax is consistent with the C programming language, and, just as with all other operations generated by the compiler, the scheduler takes care of register allocation, operation packing, and flow analysis.

4.1.1 Custom Operation Motivation

For both general-purpose and embedded microprocessor-based applications, programming in a high-level language is desirable. To effectively support optimizing compilers and a simple programming model, certain microprocessor architecture features are needed, such as a large, linear address space, general-purpose registers, and register-to-register operations that directly support the manipulation of linear address pointers. A common choice in microprocessor architectures is 32-bit linear addresses, 32-bit registers, and 32-bit integer operations. TM1000 is such a microprocessor architecture.

For the data manipulation in many algorithms, however, 32-bit data and operations are wasteful of expensive silicon resources. Important multimedia applications, such as the decompression of MPEG video streams, spend significant amounts of execution time dealing with eight-bit data items. Using 32-bit operations to manipulate small data items makes inefficient use of 32-bit execution hardware in the implementation. If these 32-bit resources could be used instead to operate on four eight-bit data items simultaneously, performance would be improved by a significant factor with only a tiny increase in implementation cost.

Getting the highest execution rate from standard microprocessor resources is one of the motivations behind custom operations in TM1000. A range of custom operations is provided that each process—simultaneously—four eight-bit or two sixteen-bit data items. There is little cost difference between a standard 32-bit ALU and one that can process either one pair of 32-bit operands or four pairs of eight-bit operands, but there is a big performance difference for TM1000's target applications.

TM1000's custom operations go beyond simply making the best use of standard resources. Custom operations that combine several simple operations are provided. These combinations of operations are tailored specifically to the needs of important multimedia applications. Some high-function custom operations eliminate conditional branches, which helps the scheduler make effective use of all five operation slots in each TM1000 instruction. Filling up all five slots is especially important in the inner loops of computationally intensive multimedia applications.

In short, custom operations help TM1000 reach its goals of extremely high multimedia performance at the lowest possible cost.

4.1.2 Introduction to Custom Operations

Table 4-1 and Table 4-2 contain two listings of the custom operations available in the TM1000 architecture. Table 4-1 groups the custom operations by type of function while Table 4-2 lists the operations by operand size. For more detailed information about the custom operations, Appendix A, "DSPCPU Operations."

Some operations exist in several versions that differ in the treatment of their operands and results, and the mnemonics for these versions make it easy to select the appropriate operation. For example, the sum of products operations all have "fir" in their mnemonics; the prefix and suffix of the mnemonic expresses the treatment of the operands and result. The ifir8ii operation treats both of its operands as signed (ifir8ii) and produces a signed result (ifir8ii). The ifir8iu operation treats its first operand as signed (ifir8iu), the second as unsigned (ifir8iu), and produces a signed result (ifir8iu). The ume8ii operation implements an eight-bit motion-estimation; it treats both operands as signed but produces an unsigned result.

The operations beginning with "dsp" implement a clipping (sometimes called saturating) function before storing the result(s) in the destination register. Otherwise, their naming follows the rules given above where appropriate. For example, the dspuquadaddui operation imple-

Table 4-1. Custom Operations Listed by Function Type

Function	Custom Op	Description
DSP absolute value	dspiabs	Clipped signed 32-bit absolute value
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
DSP add	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspuquadaddui	Quad clipped add of unsigned/ signed bytes
DSP multiply	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
DSP subtract	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
Sum of products	ifir16	Signed sum of products of signed 16-bit halfwords
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	ufir8uu	Unsigned sum of products of unsigned bytes
Merge, pack	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
	packbytes	Pack least-significant bytes
Byte averages	quadavg	Unsigned byte-wise quad average
Byte multiplies	quadumulmsb	Unsigned quad 8-bit multiply most significant
Motion estimation	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

Table 4-2. Custom Operations Listed by Operand Size

Op. Size	Custom Op	Description
32-bit	dspiabs	Clipped signed 32-bit absolute value
	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
16-bit	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
	ifir16	Signed sum of products of signed 16-bit halfwords
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
8-bit	dspuquadaddui	Quad clipped add of unsigned/ signed bytes
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir8uu	Unsigned sum of products of unsigned bytes
	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	packbytes	Pack least-significant bytes
	quadavg	Unsigned byte-wise quad average
	quadumulmsb	Unsigned quad 8-bit multiply most significant
	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

ments four eight-bit additions; it treats the first operand of each addition as unsigned, the second operand as signed, and produces an unsigned result for each addition. Each result, which is computed with no loss of precision, is clipped into the representable range of a byte (0..255).

4.1.3 Example Uses of Custom Ops

The next three sections illustrate the advantages of using custom operations. Also, the more complex examples illustrate how custom operations can be integrated into application code by providing listings of C-language program fragments. The examples progress in complexity from simple to intricate; the most interesting examples

are taken from actual multimedia codes, such as MPEG decompression.

4.2 EXAMPLE 1: BYTE-MATRIX TRANSPOSITION

The goal of this example is to provide a simple, introductory illustration of how custom operations can significantly increase processing speed in small kernels of applications. As in most uses of custom operations, the power of custom operations in this case comes from their ability to operate on multiple data items in parallel.

Imagine that our task is to transpose a packed, four-by-four matrix of bytes in memory; the matrix might, for example, contain eight-bit pixel values. Figure 4-1 illustrates both the organization of the matrix in memory and, in standard mathematical notation, the task to be performed.

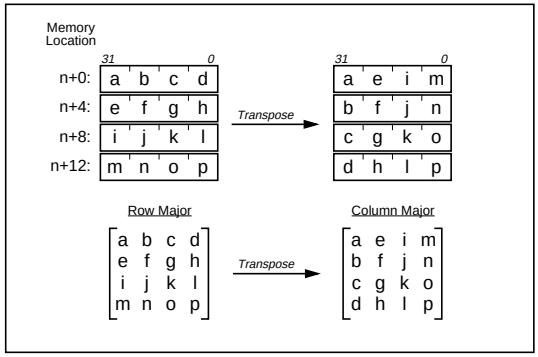


Figure 4-1. Byte-matrix transposition. Top shows byte matrices packed into memory words; bottom shows mathematical matrix representation.

Performing this operation with traditional microprocessor instructions is straight forward but time consuming. One way to perform the manipulation is to perform 12 load-byte instructions (since only 12 of the 16 bytes need to be repositioned) and 12 store-byte instructions that place the bytes back in memory in their new positions. Another way would be to perform four load-word instructions, reposition the bytes in registers, and then perform four store-word instructions. Unfortunately, repositioning the bytes in registers would require a large number of instructions to properly shift and mask the bytes. Performing the 24 loads and stores makes implicit use of the shifting and masking hardware in the load/store units and thus yields a shorter instruction sequence.

The problem with performing 24 loads and stores is that loads and stores are inherently slow operations because they must access at least the cache and possibly slower layers in the memory hierarchy. Further, performing byte loads and stores when 32-bit word-wide accesses run just as fast wastes the power of the cache/memory interface. We would prefer a fast algorithm that takes full advantage of cache/memory bandwidth while not requiring an inordinate number of byte-manipulation instructions.

TM1000 has instructions that merge and pack bytes and 16-bit halfwords directly and in parallel. Four of these instructions can be applied in this case to speed up the manipulation of bytes that are packed into words.

Figure 4-2 shows the application of these instructions to the byte-matrix transposition problem, and the left side of Figure 4-3 shows a list of the operations needed to implement the maxtrix tranpose. When assembled into actual TM1000 instructions, these custom operations would be packed as tightly as dependencies allow, up to five operations per instruction.

Note that a programmer would not need to resort to programming at this level (TM1000 assembler). The matrix transpose would be expressed just as efficiently in C-language source code, as shown on the right side of Figure 4-3. The low-level code is shown here for illustration purposes only.

The first sequence of four load-word operations in Figure 4-3 brings the packed words of the input matrix into registers R10, R11, R12, and R13. The next sequence of four merge operations produces intermediate results into registers R14, R15, R16, and R17. The next sequence of four pack operations could then replace the original operands or place the transposed matrix in separate registers if the original matrix operands were needed for further computations (the TM1000 optimizing C compiler performs this analysis automatically). In this example, the transpose matrix is placed in registers R18, R19, R20, and R21. The final four store-word operations put the transposed matrix back into memory.

Thus, using the TM1000 custom operations, the byte-matrix transposition requires four load-word operations and four store-word operations (the minimum possible) and eight register-to-register data-manipulation operations. The result is 16 operations, or byte-matrix transposition at the rate of one operation per byte.

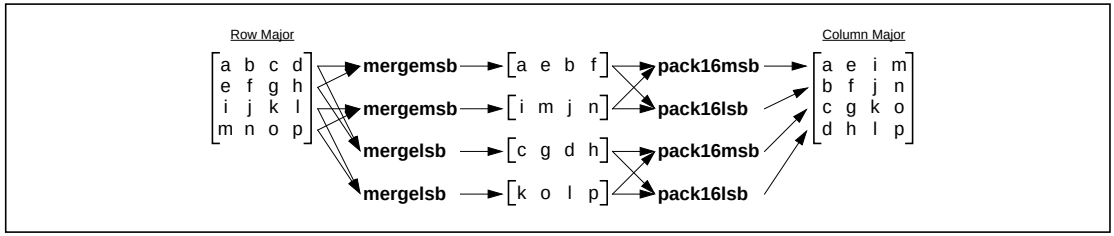


Figure 4-2. Application of merge and pack instructions to the byte-matrix transposition of Figure 4-1.

```
ld32d(0) r100 → r10
ld32d(4) r100 → r11
ld32d(8) r100 → r12
ld32d(12) r100 → r13

mergemsb r10 r11 → r14
mergemsb r12 r13 → r15
mergelsb r10 r11 → r16
mergelsb r12 r13 → r17
pack16msb r14 r15 → r18
pack16lsb r14 r15 → r19
pack16msb r16 r17 → r20
pack16lsb r16 r17 → r21

st32d(0) r101 r18
st32d(4) r101 r19
st32d(8) r101 r20
st32d(12) r101 r21
```

```
char matrix[4][4];
        .
        .
        .
int *m = (int *) matrix;

temp0 = MERGEMSB(m[0], m[1]);
temp1 = MERGEMSB(m[2], m[3]);
temp2 = MERGELSB(m[0], m[1]);
temp3 = MERGELSB(m[2], m[3]);
m[0]  = PACK16MSB(temp0, temp1);
m[1]  = PACK16LSB(temp0, temp1);
m[2]  = PACK16MSB(temp2, temp3);
m[3]  = PACK16LSB(temp2, temp3);
        .
        .
        .
```

Figure 4-3. On the left is a complete list of operations to perform the byte-matrix transposition of Figure 4-1 and Figure 4-2. On the left is an equivalent C-language fragment.

While the advantage of the custom-operation-based algorithm over the brute-force code that uses 24 load- and store-byte instruction seems to be only eight operations (a 33% reduction), the advantage is actually much greater. First, using custom operations, the number of memory references is reduced from 24 to eight (a factor of three). Since memory references are slower than register-to-register operations (such as the custom operations in this example), the reduction in memory references is significant.

Further, the ability of the TM1000 compiling system to exploit the performance potential of the TM1000 microprocessor hardware is enhanced by the custom-operation-based code. This is because it is easier for the compiling system to produce an optimal schedule (arrangement) of the code when the number of memory references is in balance with the number of register-to-register operations. The TM1000 CPU (like all high-performance microprocessors) has a limit on the number of memory references that can be processed in a single cycle (two is the current limit). A long sequence of code that contains only memory references can result in empty operation slots in the long TM1000 instructions. Empty operation slots waste the performance potential of the TM1000 hardware.

As this example has shown, careful use of custom operations has the potential to not only reduce the absolute number of operations needed to perform a computation but can also help the compiling system produce code that fully exploits the performance potential of the TM1000 CPU.

4.3 EXAMPLE 2: MPEG IMAGE RECONSTRUCTION

The complete MPEG video decoding algorithm is composed of many different phases, each with computationally intensive kernels. One important kernel deals with reconstructing a single image frame given that the forward- and backward-predicted frames and the inverse discrete cosine transform (IDCT) results have already been computed. This kernel provides an excellent opportunity to illustrate of the power of TM1000's specialized custom operators.

In the code fragments that follow, the backward-predicted block is assumed to have been computed into an array back[], the forward-predicted block is assumed to have been computed into forward[], and the IDCT results are assumed to have been computed into idct[].

A straightforward coding of the reconstruction algorithm might look as shown in Figure 4-4. This implementation shares many of the undesirable properties of the first example of byte-matrix transposition. The code accesses memory a byte at a time instead of a word at a time, which wastes 75% of the available memory bandwidth. Also, in light of the many quad-byte-parallel operations introduced in Section 4.1.2, "Introduction to Custom Operations," it seems inefficient to spend three separate additions and one shift to process a single eight-bit pixel. Perhaps even more unfortunate for a VLIW processor like TM1000 is the branch-intensive code that performs the saturation testing; eliminating these branches could reap a significant performance gain.

Since MPEG decoding is the kind of task for which TM1000 was created, there are two custom operations—quadavg and dspuquadaddui—that exactly fit this important MPEG kernel (and other kernels). These custom operations process four pairs of eight-bit pixel values in parallel. In addition, dspuquadaddui performs saturation tests in hardware, which eliminates any need to execute explicit tests and branches.

For readers familiar with the details of MPEG algorithms, the use of eight-bit IDCT values later in this example may be confusing. The standard MPEG implementation calls for nine-bit IDCT values, but extensive analysis has shown that values outside the range [–128..127] occur so rarely that they can be considered unimportant. Pursuant to this observation, the IDCT values are clipped into the eight-bit range [–128..127] with saturating arithmetic before the frame reconstruction code runs. The assumption that this saturation occurs permits some of

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 1)
    {
        temp = ((back[i] + forward[i] + 1) >> 1) + idct[i];

        if (temp > 255)
            temp = 255;
        else if (temp < 0)
            temp = 0;

        destination[i+0] = temp;
    }
}
```

Figure 4-4. Straightforward code for MPEG frame reconstruction.

TM1000's custom operations to have clean, simple definitions.

The first step in seeing how custom operations can be of value in this case is to unroll the loop by a factor of four. The unrolled code is shown in Figure 4-5. This creates code that is parallel with respect to the four pixel computations. As is easily seen in the code, the four groups of computations (one group per pixel) do not depend on each other.

After some experience is gained with custom operations, it is not necessary to unroll loops to discover situations where custom operations are useful. Often, a good programmer with knowledge of the function of the custom operations can see by simple inspection opportunities to exploit custom operations.

To understand how quadavg and dspuquadaddui can be used in this code, we examine the function of these custom operations.

The quadavg custom operation performs pixel averaging on four pairs of pixels in parallel. Formally, the operation of quadavg is as follows:

```
quadavg rscr1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [abcd], rsrc2 = [wxyz], and rdest = [pqrs] where a, b, c, d, w, x, y, z, p, q, r, and s are all unsigned eight-bit values. Then, quadavg computes the output vector [pqrs] as follows:

```
p = (a + w + 1) >> 1
q = (b + x + 1) >> 1
r = (c + y + 1) >> 1
s = (d + z + 1) >> 1
```

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 4)
    {
        temp = ((back[i+0] + forward[i+0] + 1) >> 1) + idct[i+0];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+0] = temp;

        temp = ((back[i+1] + forward[i+1] + 1) >> 1) + idct[i+1];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+1] = temp;

        temp = ((back[i+2] + forward[i+2] + 1) >> 1) + idct[i+2];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+2] = temp;

        temp = ((back[i+3] + forward[i+3] + 1) >> 1) + idct[i+3];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+3] = temp;
    }
}
```

Figure 4-5. MPEG frame reconstruction code using TM1000 custom operations; compare with Figure 4-4.

The pixel averaging in Figure 4-5 is evident in the first statement of each of the four groups of statements. The rest of the code—adding idct[i] value and performing the saturation test—can be performed by the dspuquadaddui operation. Formally, its function is as follows:

```
dspuquadaddui rsrc1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [efgh], rsrc2 = [stuv], and rdest = [ijkl] where e, f, g, h, i, j, k, and l are unsigned eight-bit values; s, t, u, and v are signed eight-bit values. Then, dspuquadaddui computes the output vector [ijkl] as follows:

```
i = uclipi(e + s, 255)
j = uclipi(f + t, 255)
k = uclipi(g + u, 255)
l = uclipi(h + v, 255)
```

The uclipi operation is defined in this case as it is for the separate TM1000 operation of the same name described in Chapter 4. Its definition is as follows:

```
uclipi (m, n)
{
    if (m < 0) return 0;
    else if (m > n) return n;
    else return m;
```

To make is easier to see how these operations can subsume all the code in Figure 4-5, Figure 4-6 shows the same code rearranged to group the related functions. Now it should be clear that the quadavg operation can replace the first four lines of the loop assuming that we can get the individual 8-bit elements of the back[] and forward[] arrays positioned correctly into the bytes of a 32-bit word. That, of course, is easy: simply align the byte arrays on word boundaries and access them with word (integer) pointers.

Similarly, it should now be clear that the dspuquadaddui operation can replace the remaining code (except, of course, for storing the result into the destination[] array) assuming, as above, that the 8-bit elements are aligned and packed into 32-bit words.

Figure 4-7 shows the new code. The arrays are now accessed in 32-bit (int-sized) chunks, the loop iteration control has been modified to reflect the “four-at-a-time” operations, and the quadavg and dspuquadaddui operations have replaced the bulk of the loop code. Finally, Figure 4-8 shows a more compact expression of the loop code, eliminating the temporary variable.

Again, note that the code in Figure 4-7 and Figure 4-8 assumes that the character arrays are 32-bit word aligned and padded if necessary to fill an integral number of 32-bit words.

The original code required three additions, one shift, two tests, three loads, and one store per pixel. The new code using custom operations requires only two custom operations, three loads, and one store for *four* pixels, which is more than a factor of six improvement. The actual performance improvement can be even greater depending on how well the compiler is able to deal with the branches in the original version of the code, which depends in part on the surrounding code. Reducing the number of branches

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp0, temp1, temp2, temp3;

    for (i = 0; i < 64; i += 4)
    {
        temp0 = ((back[i+0] + forward[i+0] + 1) >> 1);
        temp1 = ((back[i+1] + forward[i+1] + 1) >> 1);
        temp2 = ((back[i+2] + forward[i+2] + 1) >> 1);
        temp3 = ((back[i+3] + forward[i+3] + 1) >> 1);

        temp0 += idct[i+0];
        if (temp0 > 255) temp = 255;
        else if (temp < 0) temp = 0;

        temp1 += idct[i+1];
        if (temp1 > 255) temp1 = 255;
        else if (temp1 < 0) temp1 = 0;

        temp2 += idct[i+2];
        if (temp2 > 255) temp2 = 255;
        else if (temp2 < 0) temp2 = 0;

        temp3 += idct[i+3];
        if (temp3 > 255) temp3 = 255;
        else if (temp3 < 0) temp3 = 0;

        destination[i+0] = temp;
        destination[i+1] = temp1;
        destination[i+2] = temp2;
        destination[i+3] = temp3;
    }
}
```

Figure 4-6. Re-grouped code of Figure 4-5.

```
                    void reconstruct (unsigned char *back,
                                      unsigned char *forward,
                                               char *idct,
                                      unsigned char *destination)
                    {
                        int i, temp;

                        int *i_back    = (int *) back;
                        int *i_forward = (int *) forward;
                        int *i_idct    = (int *) idct;
                        int *i_dest    = (int *) destination;

                        for (i = 0; i < 16; i += 1)
                        {
                            temp = QUADAVG(i_back[i], i_forward[i]);

                            temp = DSPUQUADADDUI(temp, i_idct[i]);

                            i_dest[i] = temp;
                        }
                    }
```

Figure 4-7. Using the custom operation dspquadaddui to speed up the loop of Figure 4-6.

```
    void reconstruct (unsigned char *back,
                      unsigned char *forward,
                               char *idct,
                      unsigned char *destination)
    {
        int i;

        int *i_back    = (int *) back;
        int *i_forward = (int *) forward;
        int *i_idct    = (int *) idct;
        int *i_dest    = (int *) destination;

        for (i = 0; i < 16; i += 1)
            i_dest[i] = DSPUQUADADDUI(QUADAVG(i_back[i], i_forward[i]), i_idct[i]);
    }
```

Figure 4-8. Final version of the frame-reconstruction code.

almost always improves the chances of realizing maximum performance on the TM1000 CPU.

The code in Figure 4-8 illustrates several aspects of using custom operations in C-language source code. First, the custom operations require no special declarations or syntax; they appear to be simple function calls. Second, there is no need to explicitly specify register assignments for sources, destinations, and intermediate results; the compiler and scheduler assign registers for custom operations just as they would for built-in language operations such as integer addition. Third, the scheduler packs custom operations into TM1000 VLIW instructions as effectively as it packs operations generated by the compiler for native language constructs.

Thus, although the burden of making effective use of custom operations falls on the programmer, that burden consists only of discovering the opportunities for exploiting the operations and then coding them using standard C-language notation. The compiler and scheduler take care of the rest.

4.4 EXAMPLE 3: MOTION-ESTIMATION KERNEL

Another part of the MPEG coding algorithm is motion estimation. The purpose of motion estimation is to reduce the cost of storing a frame of video by expressing the contents of the frame in terms of adjacent frames. A given frame is reduced to small blocks, and a subsequent frame is represented by specifying how these small blocks change position and appearance; usually, storing the difference information is cheaper than storing a whole block. For example, in a video sequence where the camera pans across a static scene, some frames can be expressed simply as displaced versions of their predecessor frames. To create a subsequent frame, most blocks are simply displaced relative to the output screen.

The code in this example is for a match-cost calculation, a small kernel of the complete motion-estimation code. As with the previous example, this code provides an excellent example of how to transform source code to make the best use of TM1000's custom operations.

Figure 4-9 shows the original source code for the match-cost loop. Unlike the previous example, the code is not a self-contained function. Somewhere early in the code, the arrays A[][] and B[][] are declared; somewhere between those declarations and the loop of interest, the arrays are filled with data.

4.4.1 A Simple Transformation

First, we will look at the simplest way to use a TM1000 custom operation.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 1)
        cost += abs(A[row][col] – B[row][col]);
}
```

Figure 4-9. Match-cost loop for MPEG motion estimation.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost += abs(A[row][col+0] – B[row][col+0]);
        cost += abs(A[row][col+1] – B[row][col+1]);
        cost += abs(A[row][col+2] – B[row][col+2]);
        cost += abs(A[row][col+3] – B[row][col+3]);
```

Figure 4-10. Unrolled, but not parallel, version of the loop from Figure 4-9.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(A[row][col+0] – B[row][col+0]);
        cost1 = abs(A[row][col+1] – B[row][col+1]);
        cost2 = abs(A[row][col+2] – B[row][col+2]);
        cost3 = abs(A[row][col+3] – B[row][col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 4-11. Parallel version of Figure 4-10.

We start by noticing that the computation in the loop of Figure 4-9 involves the absolute value of the difference of two unsigned characters (bytes). By now, we are familiar with the fact that TM1000 includes a number of operations that process all four bytes in a 32-bit word simultaneously. Since the match-cost calculation is fundamental to the MPEG algorithm, it is not surprising to find a custom operation—ume8uu—that implements this operation exactly.

To understand how ume8uu can be used in this case, we need to transform the code as in the previous example. Though the steps are presented here in detail, a programmer with a even a little experience can often perform these transformations by visual inspection.

If we hope to use a custom operation that processes four pixel values simultaneously, we first need to create four parallel pixel computations. Figure 4-10 shows the loop of Figure 4-9 unrolled by a factor of four. Unfortunately, the code in the unrolled loop is not parallel because each line depends on the one above it.

Figure 4-11 shows a more parallel version of the code from Figure 4-10. By simply giving each computation its own cost variable and then summing the costs all at once, each cost computation is completely independent.

Excluding the array accesses, the loop body in Figure 4-11 is recognizable now as exactly the function performed by the ume8uu custom operation: the sum of four absolute values of four differences. To use the ume8uu operation, however, the code must access the arrays with 32-bit word pointers instead of with 8-bit byte pointers.

Figure 4-12 shows the loop recoded to access A[][] and B[][] as one-dimensional instead of as two-dimensional arrays. We take advantage of our knowledge of C-lan-

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .

unsigned char *CA = A;
unsigned char *CB = B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 16;

    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(CA[rowoffset + col+0] - CB[rowoffset + col+0]);
        cost1 = abs(CA[rowoffset + col+1] - CB[rowoffset + col+1]);
        cost2 = abs(CA[rowoffset + col+2] - CB[rowoffset + col+2]);
        cost3 = abs(CA[rowoffset + col+3] - CB[rowoffset + col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 4-12. The loop of Figure 4-11 recoded with one-dimensional array accesses.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 4;

    for (col4 = 0; col4 < 4; col4 += 1)
        cost += UME8UU(IA[rowoffset + col4], IB[rowoffset + col4]);
}
```

Figure 4-13. The loop of Figure 4-12 recoded with 32-bit array accesses and the ume8uu custom operation.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (row = 0, rowoffset = 0; row < 16; row += 1, rowoffset += 4)
{
    for (col4 = 0; col4 < 4; col4 += 1)
        cost += UME8UU(IA[rowoffset + col4], IB[rowoffset + col4]);
}
```

Figure 4-14. The loop of Figure 4-13 with strength reduction applied to the rowoffset calculation.

guage array storage conventions to perform this code transformation. Recoding to use one-dimensional arrays prepares the code for the transformation to 32-bit array accesses.

(From here on, until the final code is shown, the declarations of the A and B arrays will be omitted from the code fragments for the sake of brevity.)

Figure 4-13 shows the loop of Figure 4-12 recoded to use ume8uu. Once again taking advantage of our knowledge of the C-language array storage conventions, the one-dimensional byte array is now accessed as a one-dimensional 32-bit-word array. The declarations of the pointers IA and IB as pointers to integers is the key, but also notice that the multiplier in the expression for rowoffset has been scaled from 16 to four to account for the fact that there are four bytes in a 32-bit word.

We can perform another transformation to improve the performance of this code. The outer loop contains a multiplication that can be reduced in strength to an integer addition. Since rowoffset simply tracks the value of row as it increments, we can replace the multiplication with an add of four on each iteration. Figure 4-14 shows the improved code. The rowoffset calculation is now shown as part of the for loop.

Of course, since we are now using one-dimensional arrays to access the pixel data, it is natural to use a single for loop instead of two. Figure 4-15 shows this streamlined version of the code without the inner loop. Since C-language arrays are stored as a linear vector of values, we can simply increase the number of iterations of the outer loop from 16 to 64 to traverse the entire array.

The recoding and use of the ume8uu operation has resulted in a substantial improvement in the performance of the match-cost loop. In the original version, the code executed 1280 operations (including loads, adds, subtracts, and absolute values); in the restructured version, there are only 256 operations—128 loads, 64 ume8uu operations, and 64 additions. This is a factor of five re-

duction in the number of operations executed. Also, the overhead of the inner loop has been eliminated, further increasing the performance advantage.

4.4.2 More Unrolling

The code transformations of the previous section achieved impressive performance improvements, but given the VLIW nature of the TM1000 CPU, more can be done to exploit TM1000's parallelism.

The code in Figure 4-15 has a loop containing only four operations (not counting the loop overhead). Since TM1000's branches have a delay of three instructions and each instruction can contain up to five operations, a fully utilized minimum-sized loop can contain 16 operations (20 minus the loop overhead).

The TM1000 compiling system performs a wide variety of powerful code transformation and scheduling optimizations to ensure that the VLIW capabilities of the CPU are exploited. It is still wise, however, to make program parallelism explicit in source code when possible. Explicit parallelism can only help the compiler produce a fast running program.

To this end, we can unroll the loop of Figure 4-15 some number of times to create explicit parallelism and help the compiler create a fast running loop. In this case, where the number of iterations is a power-of-two, it makes sense to unroll by a factor that is a power-of-two to create the cleanest code.

Figure 4-16 shows the loop unrolled by a factor of eight. Unfortunately, the unrolling has increased the complexity of the array indexing calculation. TM1000 has a memory load operations with a variety of addressing modes, but it does not have a mode that can add three components, which the index calculations Figure 4-16 require. The compiler can apply common subexpression elimination and other optimizations to eliminate extraneous operations, but, again, improvements in the source code can only help the compiler produce the best possible code and fastest-running program.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 1)
    cost += UME8UU(IA[i], IB[i]);
```

Figure 4-15. The loop of Figure 4-14 with the inner loop eliminated.

Figure 4-17 shows one way to modify the code for simpler array indexing.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8)
{
    cost0 = UME8UU(IA[i+0], IB[i+0]);
    cost1 = UME8UU(IA[i+1], IB[i+1]);
    cost2 = UME8UU(IA[i+2], IB[i+2]);
    cost3 = UME8UU(IA[i+3], IB[i+3]);
    cost4 = UME8UU(IA[i+4], IB[i+4]);
    cost5 = UME8UU(IA[i+5], IB[i+5]);
    cost6 = UME8UU(IA[i+6], IB[i+6]);
    cost7 = UME8UU(IA[i+7], IB[i+7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 4-16. Unrolled version of Figure 4-15. This code makes good use of TM1000's VLIW capabilities.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8, IA += 8, IB += 8)
{
    cost0 = UME8UU(IA[0], IB[0]);
    cost1 = UME8UU(IA[1], IB[1]);
    cost2 = UME8UU(IA[2], IB[2]);
    cost3 = UME8UU(IA[3], IB[3]);
    cost4 = UME8UU(IA[4], IB[4]);
    cost5 = UME8UU(IA[5], IB[5]);
    cost6 = UME8UU(IA[6], IB[6]);
    cost7 = UME8UU(IA[7], IB[7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 4-17. Code from Figure 4-16 with simplified array index calculations.

Cache Architecture Chapter 5

by Eino Jacobs

5.1 MEMORY SYSTEM OVERVIEW

The high-performance video and audio throughput of TM1000 is implemented by the DSPCPU and the autonomous I/O and graphics units, but the foundation of this processing is the TM1000 memory hierarchy. To reap the full potential of the chip's processing units, the memory hierarchy must read and write data (and instructions for the DSPCPU) fast enough to keep the units busy.

To meet the requirements of its target applications, TM1000's memory hierarchy must satisfy the conflicting goals of low cost, simple system design (e.g., low parts count), and high performance. Since multimedia video streams can require relatively large temporary storage, a significant amount of external DRAM is required. Keeping the cost of this bulk memory as low as possible is important.

TM1000's memory system achieves a good compromise between cost and performance by coupling substantial on-chip caches with a glueless interface to synchronous DRAM (SDRAM), which provides higher bandwidth than standard DRAM for only a small cost premium. A block diagram of the memory system is shown in Figure 5-1. The high bandwidth of SDRAM permits TM1000 to use a narrower and simpler interface than would be required to achieve similar performance with standard DRAM.

The separate on-chip data and instruction caches serve only the DSPCPU since the data access patterns of the autonomous I/O and graphics units exhibit little or no locality of reference (they access each piece of the multimedia data stream once only in each operation).

Without the caches, the CPU would not be able to achieve its performance potential. SDRAM has enough bandwidth to handle serial streams of multimedia data, but its bandwidth and latency are insufficient to satisfy the CPU's high rate of random data accesses and repeated instruction accesses.

Table 5-1. 100-MHz TM1000 Memory Bandwidth Parameters

Magnitude	Use
2800 MB/s	Instruction bandwidth (224 bits/instruction)
800 MB/s	Data bandwidth (two 32-bit memory ports)
400 MB/s	Main-memory bandwidth (one 32-bit port)

Table 5-1 shows bandwidth parameters for the TM1000 DSPCPU and the main-memory interface. Although 400 MB/s is a lot of bandwidth, it is clear that the SDRAM alone cannot keep up with the CPU's maximum requirements for instructions and data. Luckily, multimedia algorithms resemble other computer programs in terms of locality of reference, so the on-chip caches typically supply the majority of instructions and data to the DSPCPU. The

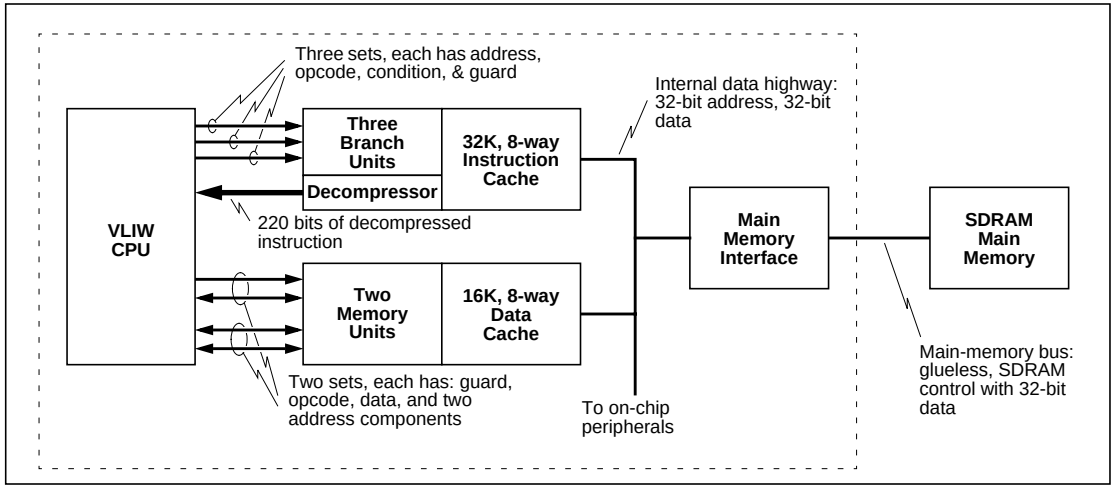


Figure 5-1. The main components of the TM1000 memory system.

wide paths to the caches are matched to the bandwidth requirements of the DSPCPU.

Table 5-2. Summary Of Memory System Characteristics

Unit	Description
Branch units	Branch units execute branch operations. Up to three branch operations can be executed in parallel, but the program must guarantee that only one branch is taken.
Decompression unit	Instructions are stored in memory and in the instruction cache in a space-saving, compressed format. The decompression unit expands instruction to their full, 28-byte size before they are issued to the CPU.
Instruction Cache	The instruction cache holds 32K bytes, is eight-way set-associative, and has a 64-byte block size. A miss in a block causes the entire block to be read from SDRAM. The cache can sustain an issue rate of one instruction per cycle on cache hits.
Memory units	Memory units execute load and store operations. The data cache is dual ported to allow the memory units to operate concurrently.
Data Cache	The data cache holds 16K bytes, is eight-way set-associative, has a 64-byte block size, and implements a copyback, allocate-on-write policy. A miss in a block causes the entire block to be read from SDRAM. The cache supports memory-mapped I/O through non-cacheable address regions.
Data highway	The on-chip data highway bus serves all on-chip units. The highway has separate 32-bit data and address buses. Bandwidth on the bus is allocated by the highway arbiter according to one of several modes.
Main-memory interface	The main-memory interface contains the data-highway access arbiter, the SDRAM controller, and MMIO logic.
SDRAM main memory	External SDRAM connects gluelessly to TM1000 over the 32-bit main-memory bus.

To improve cache behavior and thus program performance, the caches have a locking mechanism. In addition, the instruction cache is coupled with an instruction decompression unit. The compressed instruction format improves the cache hit rate and reduces the bus bandwidth required between main memory and cache. Instructions in main memory and cache use the compressed format.

TM1000's processing units access the external SDRAM through the on-chip central "data highway" bus. The highway consists of separate 32-bit address and data buses, and use of the bus is mediated by the main-memory interface unit. The main-memory interface contains the SDRAM controller and a central arbiter that determines how much of the available SDRAM memory bandwidth is allocated to each unit. Unused bandwidth is always made available to the VLIW CPU for cache refill and memory accesses that bypass the caches.

Table 5-2 gives a summary description of each component of TM1000's memory system.

5.2 DRAM APERTURE

TM1000 implements a 32-bit linear address space of bytes. Within that address space, TM1000 supports several different apertures for specific purposes. The DRAM aperture describes the part of the address space into which the external SDRAM is mapped. SDRAM must consist of a single, contiguous region of memory, which is the most practical configuration for TM1000 systems.

The location and size of the DRAM aperture is defined by two registers, DRAM_BASE and DRAM_LIMIT. These registers are both readable and writeable as MMIO registers and as PCI configuration space registers. The view of the registers in MMIO space is shown in Figure 5-2. The view of the registers in PCI configuration space is described in Chapter 10, "PCI Interface." In normal operation, the base address registers are assigned once during boot, and not changed when the DSPCPU is running. Refer to Chapter 10, "PCI Interface," and Chapter 12, "System Boot," for a description of this process.

DRAM_LIMIT must be set equal to DRAM_BASE plus the actual size of SDRAM present. The amount of the SDRAM is not required to be a power of two, but it must be a multiple of 64 KB. Note that the size of the aperture as set in the PCI configuration space can be larger, because it must be a power of 2.

A memory operation will access SDRAM if its address satisfies:

[dram_base] <= address < [dram_limit]

Any address outside this range cannot access SDRAM.

When TM1000 is reset, DRAM_BASE_FIELD is set to 0x0 and DRAM_LIMIT is set to 0x0010 0000 (1-MB DRAM aperture starting at address 0x0). The boot process described in Chapter 12, "System Boot," overrides these initial settings.

5.3 DATA CACHE

The data cache serves only the DSPCPU and is controlled by two memory units that execute the load and store operations issued by the DSPCPU. The following

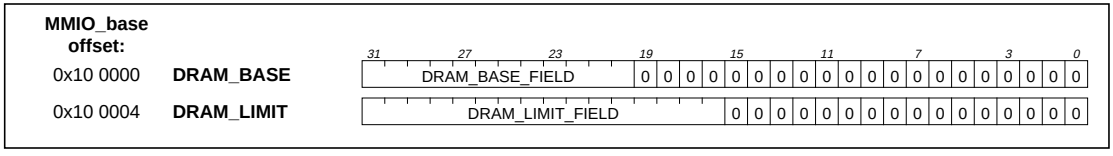


Figure 5-2. Formats of the DRAM_BASE and DRAM_LIMIT registers.

sections describe the data cache and its operation; Table 5-3 summarizes the important characteristics for easy reference.

Table 5-3. Summary Of Data Cache Characteristics

Characteristic	TM1000 Implementation
Cache size	16K bytes
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Dirty bits	One dirty bit per 64-byte block
Miss transfer order	Miss transfers begin with the first word in the block
Replacement policies	Copyback, allocate on write, hierarchical LRU
Endianness	Either little- or big-endian, determined by PCSW bit
Ports	The cache is quasi dual ported; two accesses can proceed concurrently if they reference different banks (determined by bits [4:2] of the computed addresses)
Alignment	Access must be naturally aligned (32-bit words on 32-bit boundaries, 16-bit half-words on 16-bit boundaries); the appropriate number of LSBs of un-naturally aligned addresses are set to zero. For misaligned stores, PCSW.MSE is asserted to generate an exception
Partial word operations	The cache implements byte and 16-bit accesses with the same performance as 32-bit accesses
Operation latency	Three cycles for both load and store operations
Coherency enforcement	Software uses special operations to enforce cache coherency
Cache locking	Up to 1/2 (four out of 8 blocks of each set) of the cache contents can be locked; granularity is 64-byte
Non-cacheable region	One non-cacheable aperture in the DRAM address space is supported.

5.3.1 General Cache Parameters

The data cache on TM1000 is 16 KB in size with a 64-B block size. Thus, the cache contains 256 blocks each with its own address tag. The cache is eight-way set-associative, so there are 32 sets, each containing eight tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid in the cache or invalid; on a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

Each block also contains a dirty bit, which is set whenever a write to the block occurs. Each set contains ten bits to support the hierarchical LRU replacement policy.

The geometry of the data cache is available to software by reading the MMIO register DC_PARAMS, which has the format shown in Figure 5-3. Table 5-10 lists the field values for TM1000's DC_PARAMS register.

Table 5-4. DC_PARAMS Field Values

Field Name	Value
BLOCKSIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	32

The product of the block size, associativity, and number of sets gives the total cache size (16 KB in this case).

5.3.2 Address Mapping

TM1000 data addresses are mapped onto the data cache storage structure as shown in Figure 5-4. A data address is partitioned into four fields as described in Table 5-5.

Table 5-5. Data Address Field Partitioning

Field	Address Bits	Purpose
Byte	1..0	Byte offset within a word for byte or half-word accesses
Word	5..2	Selects one of the words in a set (one of 16 words in the case of TM1000)
Set	10..6	Selects one of the sets in the cache (one of 32 in the case of TM1000)
Tag	31..11	Compared against address tags of set members

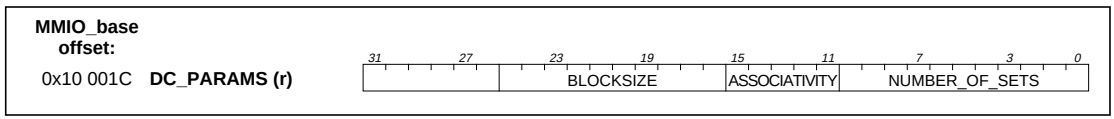


Figure 5-3. Format of the DC_PARAMS register.

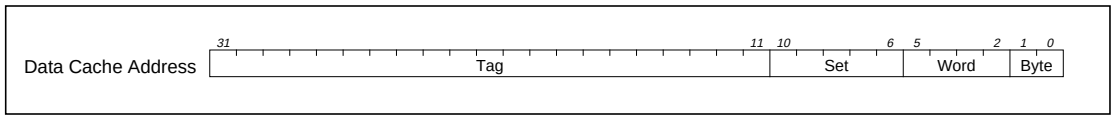


Figure 5-4. Data-Cache address partitioning.

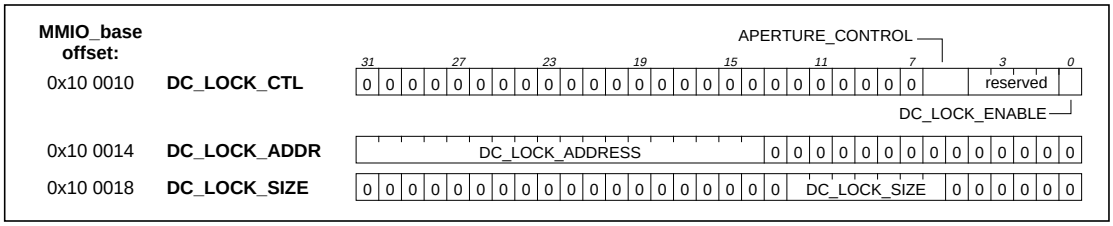


Figure 5-5. Formats of the registers in charge of data-cache locking.

5.3.3 Miss Processing Order

When a miss occurs, the data cache fills the block containing the requested word from the beginning of the block. The CPU is stalled until the entire block is transferred and stored in the cache.

5.3.4 Replacement Policies, Coherency

The cache implements a copyback replacement policy with one dirty bit per 64-B block. Thus, when a miss occurs and the block selected for replacement has its dirty bit set, the dirty block must be written to main memory to preserve its modified contents. On TM1000, the dirty block is written to memory before the needed block is fetched.

Coherency is not maintained in any way by hardware between the data cache, the instruction cache, and main memory. Special operations are available to implement cache coherency in software. See Section 5.6, “Cache Coherency,” for a discussion of coherency issues.

Write misses are handled with an allocate-on-write policy—the write that caused the miss stores its data in the cache after the missing block is fetched into the cache.

The cache implements a hierarchical LRU replacement algorithm to determine which of the eight elements (blocks) in a set is replaced. The algorithm partitions the eight set elements into four groups, each group with two elements. The hierarchical LRU replacement victim is determined by selecting the least-recently used group of two elements and then selecting the least-recently used element in that group. This hierarchical algorithm yields performance close to full LRU but is simpler to implement.

See Section 5.5, “LRU Algorithm,” for a full discussion of the LRU algorithm.

5.3.5 Alignment, Partial-Word Transfers, Endian-ness

The cache implements 32-bit word, 16-bit halfword, and 8-bit byte transfers. All transfers, however, must be to addresses that are naturally aligned; that is, 32-bit words must be aligned on 32-bit boundaries, and 16-bit halfwords must be aligned on 16-bit boundaries.

The CPU uses big-endian byte order for its accesses to memory. The main-memory interface unit and TM1000’s other processing units, however, have the capability to use either big- or little-endian byte order.

5.3.6 Dual Ports

To allow two accesses to proceed in parallel, the data cache is quasi-dual ported. The cache is implemented as eight banks of single-ported memory, but the hardware allows each bank to operate independently. Thus, when the addresses of two simultaneous accesses select two different banks, both accesses can complete simultaneously. Bank selection is determined by the three low-order address bits [4..2] of each address. Thus, the words in a 64-byte cache block are distributed among the eight blocks, which prevents conflicts between two simultaneously issued accesses to adjacent words in a cache block. The TM1000 compiling system attempts to avoid bank conflicts as much as possible.

The dual-ported cache can execute the load and store opcodes (ild8d, uld8d, ild16d, uld16d, ld32d, h_st8d, h_st16d, h_st32d, ild8r, uld8r, ild16r, uld16r, ld32r, ild16x, uld16x, ld32x) in either or both of the two ports.

The special opcodes dcb, dinvalid, rdtag and rdstatus can only be executed in the second port, not in the first port. Whenever any of these special opcodes is issued in the second port, there should not be a concurrent load or store operation in the first. This is a special scheduling constraint.

5.3.7 Cache Locking

The data cache allows the contents of up to one-half of its blocks to be locked. Thus, on TM1000, up to 8K bytes of the cache can be used as a high-speed local data memory. Only four out of eight blocks in any set can be locked.

A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until either (1) the block’s locked status is changed explicitly by software, or (2) a dinvalid operation is executed that targets the locked block.

Cache locking occurs only for the data in the address range described by the MMIO registers DC_LOCK_ADDR and DC_LOCK_SIZE. The granularity of the address range is one 64-byte cache block. The MMIO register DC_LOCK_CTL contains the cache-locking enable bit dcache_lock_enable. Figure 5-5 shows the layout of the data-cache lock registers. Locking will occur for an address if locking is enabled and both of the following are true:

1. The address is greater than or equal to the value in DC_LOCK_ADDR.

2. The address is less than the sum of the values in DC_LOCK_ADDR and DC_LOCK_SIZE.

Programmers (or compilers) must combine all data that needs to be locked into this single linear address range.

Setting dcache_lock_enable to '1' causes the following sequence of events:

1. All blocks that are in cache locations that will be used for locking are copied back to main memory (if they are dirty) and removed from the cache.
2. All blocks in the lock range are fetched from main memory into the cache. If any block in the lock range was already in the cache, it's first copied back (if it's dirty) and invalidated.
3. The LRU status of any set that contains locked blocks is set to the initialization value.
4. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

This sequence of events is triggered by writing '1' to dcache_lock_enable even if the enable is already set to '1'. Setting dcache_lock_enable to '0' causes no action except to allow the previously locked blocks to be replacement victims.

To program a new lock range, the following sequence of operations is used:

1. Disable cache locking by writing '0' to DC_LOCK_ENABLE.
2. Define a new lock range by writing to DC_LOCK_ADDR and DC_LOCK_SIZE.
3. Enable cache locking by writing '1' to DC_LOCK_ENABLE.

Dirty locked blocks can be written back to main memory while locking is enabled by executing copyback operations in software.

Programmer's note: Software should not execute dinvalid operations on a locked block. If it does, the block will be removed from the cache, creating a 'hole' in the lock range (and the dcache) that cannot be reused until locking is deactivated.

Cache locking is disabled by default when TM1000 is reset.

5.3.8 Memory Hole and PCI Aperture Disable

Bits 6 and 5 in DC_LOCK_CTL comprise the APERTURE_CONTROL field. This field can be used to change the memory map as seen by the DSPCPU. The hardware RESET value of the field corresponds to the memory map as described in Section 3.3.1, "Memory Map."

Table 5-6. Aperture Control field

value	Memory Map properties
00 (RESET)	normal operation memory map (Section 3.3.1): • loads to 0..0xff always return 0 and cause no PCI read (memory hole is enabled) • PCI aperture(s) are enabled
01	loads to address 0..0xff cause a PCI read, i.e. the memory hole is disabled
10	PCI apertures are disabled for both loads and stores
11	RESERVED for future extensions

5.3.9 Non-Cacheable Region

The data cache supports one non-cacheable address region within the DRAM address space aperture. The base address of this region is determined by the value in the DRAM_CACHEABLE_LIMIT MMIO register, which is shown in Figure 5-6. SInce uncached memory operations always incur many stall cycles, the non-cacheable region should be used sparingly.

A memory operation is non-cacheable if its target address satisfies:

[dram_cacheable_limit] <= address < [dram_limit]

Thus, the non-cacheable region is at the high end of the DRAM aperture. The format of the DRAM_CACHEABLE_LIMIT register forces the size of the non-cacheable region to be a multiple of 64 KB.

When TM1000 is reset, DRAM_CACHEABLE_LIMIT is set equal to DRAM_LIMIT, which results in a zero-length non-cacheable region.

Programmer's note: When DRAM_CACHEABLE_LIMIT is changed to enlarge the region that is non-cacheable, software must assure coherency. This is accomplished by explicitly copying back dirty data (using dcb operations) and invalidating (using dinvalid operations) the cache blocks in the previously unlocked region.

5.3.10 Special Data Cache Operations

A program can exercise some control over the operation of the data cache by executing special operations. The special operations can cause the data cache to initiate the copyback or invalidation of a block in the cache. These operations are typically used by software to keep the cache coherent with main memory.

In addition, there are special operations that allow a program to read tag and status information from the data cache.

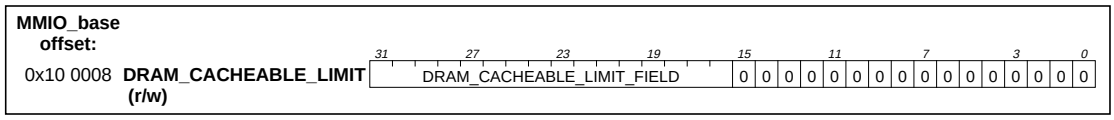


Figure 5-6. Formats of the DRAM_cacheable_limit register.

5.3.10.1 Copyback and Invalidate Operations

The data cache controller recognizes a copyback and an invalidate operation as shown in Table 5-7.

Table 5-7. Copyback And Invalidate Operations

Mnemonic	Description
dcb(offset) r*src1*	Data-cache copyback block. Causes the block that contains the target address to be copied back to main memory if the block is valid and dirty.
dinvalid(offset) r*src1*	Data-cache invalidate block. Causes the block that contains the target address to be invalidated. No copy-back occurs even if the block is dirty.

The dcb and dinvalid operations both compute a target word address that is the sum of a register and seven-bit offset. The offset can be in the range [–256..255] and must be divisible by four.

dcb operation. The dcb operation computes the target address, and if the block containing the address is found in the data cache, its contents are written back to main memory if the block is both valid and dirty. If the block is not present, not valid, or not dirty, no action results from the dcb operation. If the dcb causes a copyback to occur, the CPU is stalled until the copyback completes. If the dcb causes no action, the operation causes no stall cycles.

The dcb operation clears the dirty bit but leaves a valid copy of the written-back block in the cache.

dinvalid operation. The dinvalid operation computes the target address, and if the block containing the address is found in the data cache, its valid and dirty bits are cleared. No copyback operation will occur even if the block is valid and dirty prior to executing the dinvalid operation. The CPU is stalled if the target block is in the cache; otherwise, no stall cycles occur.

The dinvalid and dcb operations affect the LRU replacement status of cache blocks. A dinvalid operation updates the LRU information for the block that is invalidated as if it is accessed, i.e. the invalidated block gets the most recently used status in its set. A dcb operation updates the LRU information as if the block that is copied back is accessed, i.e. the block that is copied back gets the most recently used status in its set.

Programmer's note: Software should not execute dinvalid operations on locked blocks; otherwise, a 'hole' is created that cannot be reused until locking is deactivated.

5.3.10.2 Data-Cache Tag and Status Operations

The data cache controller recognizes two operations for reading cache status as shown in Table 5-8.

The rdtag and rdstatus operations both compute a target word address that is the sum of a register and scaled seven-bit offset. The offset must be divisible by four and in the range [–256..255].

Table 5-8. Cache Read-Status Operations

Mnemonic	Description
rdtag(offset) r*src1*	Read data-cache tag. The target address selects a data-cache block directly; the operation returns a 32-bit result containing the 21-bit cache tag and the valid bit.
rdstatus(offset) r*src1*	Read data-cache status. The target address selects a data-cache set directly; the operation returns a 32-bit result containing the set's eight dirty bits and ten LRU bits.

rdtag operation. The target address computed by rdtag selects the data cache block by specifying the cache set and set element directly. Address bits [10..6] specify the cache set (one of 32), and bits [13..11] specify the set element (one of eight). All other target address bits are ignored. This operation does not cause CPU stall cycles.

The result of the rdtag operation is a full 32-bit word with the format shown in Figure 5-7.

rdstatus operation. The target address computed by rdstatus selects the data cache set by specifying the set number directly. Address bits [10..6] specify the cache set (one of 32); all other target address bits are ignored. This operation causes two CPU stall cycles.

The result of the rdstatus operation is a full 32-bit word with the format shown in Figure 5-7. See Section 5.5.4, "LRU Bit Definitions," for a description of the LRU bits.

A rdtag or rdstatus operation is always executed on the memory port associated with issue slot 5.

5.3.11 Memory Operation Ordering

The TM1000 memory system implements traditional ordering for memory operations that are issued in different clock cycles. That is, the effects of a memory operation

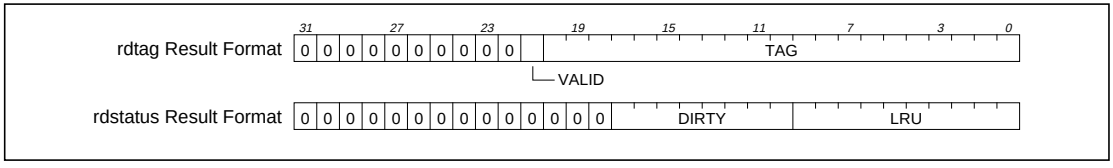


Figure 5-7. Result formats for rdtag and rdstatus operations.

issued in cycle j occur before the effects of a memory operation issued in cycle j+1.

For memory operations issued in the same cycle, however, it is not possible to execute memory operations in a traditional order. So long as the simultaneous memory operations access different addresses (aliasing is not possible in TM1000), no problems can occur. If two simultaneous operations do access the same address, however, TM1000 behavior is undefined. Specifically, two cases are possible:

1. When multiple values are written to the same address in the same cycle, the resulting value in memory is undefined.
2. When a read and a write occur to the same address in the same clock cycle, the value returned by the read is undefined.

The behavior of simultaneous accesses to the same address is undefined regardless of whether one or both memory operations hit in the cache.

Hidden Memory System Concurrency. Some cache operations may be overlapped with CPU execution. In general, a program cannot determine in what order cache misses will complete nor can a program determine when and in what order copyback operations will complete. A program can, however, enforce the completion of copyback transactions to main memory because copyback and invalidate operations can complete only if pending copyback transactions for the same block have completed. Thus, a program can synchronize to the completion of a copyback operation by dirtying a block, issuing a copyback operation for the block, and then issuing an invalidate operation for the block.

Ordering Of Special Memory Operations. The following are special memory operations:

1. Loads or stores to MMIO addresses.
2. Non-cached loads or stores.
3. Any copyback or invalidate operation.
4. Loads or stores that cause a PCI-bus access.

The CPU is stalled while these special memory operations are completed; there is no overlap of CPU execution with these special memory operations. Thus, a programmer can assume that traditional memory operation ordering applies to special memory operations. Note, however, that ordering is undefined for two special memory operations issued in the same cycle.

5.3.12 Operation Latency

Load and store operations have an operation latency of three cycles, regardless of the size of the data transfer.

5.3.13 MMIO Register References

Memory operations that reference MMIO registers are not cached, and the CPU is stalled until the MMIO reference completes. A MMIO register reference occurs when an address is in the range:

[mmio_base] <= address < ([mmio_base] + 0x200000)

The size of the MMIO aperture is hardwired at 2M bytes.

5.3.14 PCI Bus References

Any CPU memory operation that references an address outside the SDRAM and MMIO address apertures is assumed to reference a device or memory on the PCI bus. PCI-bus data transfers are not cached, and the CPU is stalled until the PCI transfer completes.

5.3.15 CPU Stall Conditions

The data cache causes the CPU to stall when:

1. Any cache miss occurs.
2. Two simultaneously issued, cacheable memory operations need to access the same cache bank (bank conflict).
3. An access that references an address in the MMIO aperture is issued.
4. An access to the PCI bus is issued.
5. A non-trivial copyback or invalidate operation is issued.
6. An access to the non-cacheable region in the DRAM aperture is issued.

5.3.16 Data Cache Initialization

When TM1000 is reset, the data cache executes an initialization sequence. The cache asserts the CPU stall signal while it sequentially resets all valid and dirty bits. The cache de-asserts the stall signal after completing the initialization sequence.

5.4 INSTRUCTION CACHE

The instruction cache stores compressed CPU instructions; instructions are decompressed before being delivered to the CPU. The following sections describe the instruction cache and its operation; Table 5-9 summarizes instruction-cache characteristics.

Table 5-9. Summary Of Instruction Cache Characteristics

Characteristic	TM1000 Implementation
Cache size	32K bytes
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Replacement policy	Hierarchical LRU (least-recently used) among the eight blocks in a set
Operation latency	Branch delay is three cycles
Coherency enforcement	Software uses a special operation to enforce cache coherency
Cache locking	Up to 1/2 (four out of eight blocks of each set) of the cache contents can be locked; granularity is 64 bytes

5.4.1 General Cache Parameters

The instruction cache on TM1000 is 32 KB in size with a 64-B block size. Thus, the cache contains 512 blocks each with its own address tag. The cache is eight-way set-associative, so there are 64 sets, each containing eight tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid or invalid; on a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

The geometry of the instruction cache is available to software by reading the MMIO register icache_parameters, which has the format shown in Figure 5-8. Table 5-10 lists the field values for TM1000's IC_PARAMS register.

The product of the block size, associativity, and number of sets gives the total cache size (32 KB in this case).

Table 5-10. IC_PARAMS Field Values

Field Name	Value
BLOCKSIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	64

5.4.2 Address Mapping

TM1000 instruction addresses are mapped onto the data cache storage structure as shown in Figure 5-9. An instruction address is partitioned into three fields as described in Table 5-5.

Table 5-11. Instruction Address Field Partitioning

Field	Address Bits	Purpose
Offset	5..0	Byte offset into a set
Set	11..6	Selects one of the sets in the cache (one of 64 in the case of TM1000)
Tag	31..12	Compared against address tags of set members

5.4.3 Miss Processing Order

When a miss occurs, the instruction cache starts filling the requested block from the beginning of the block. The DSPCPU is stalled until the entire block is fetched and stored in the cache.

5.4.4 Replacement Policy

The hierarchical LRU replacement policy implemented by the instruction cache is identical to that implemented by the data cache. See Section 5.3.4, "Replacement Policies, Coherency," for a description of the hierarchical LRU algorithm.

5.4.5 Location of Program Code

All program code must first be loaded into SDRAM. The instruction cache cannot fetch instructions from other memories or devices. In particular, the cache cannot fetch code from on-chip devices or over the PCI bus.

5.4.6 Branch Units

The instruction cache is closely coupled to three branch units. Each unit can accept a branch independently, so three branches can be processed simultaneously in the same cycle.

Branches in TM1000 are so-called delayed branches because the effect of a successful (taken) branch is not seen in the flow of control until some number of cycles after the successful branch is executed. The number of cycles of latency is called the branch delay, and on TM1000, the branch delay is three cycles.

Although three branches can be executed simultaneously, correct operation of the DSPCPU requires that only one be successful (taken) in any one cycle. DSPCPU operation is undefined if more than one concurrent branch operation is successful.

Each branch unit takes four inputs from the DSPCPU: the branch opcode, a guard bit, a branch condition, and a branch target address. A branch is deemed successful if and only if the opcode is a branch opcode, the guard bit is TRUE (i.e., = 1), and the condition (determined by the opcode) is satisfied.

5.4.7 Coherency: Special iclr Operation

A program can exercise some control over the operation of the instruction cache by executing the special iclr operation. This operation causes the instruction cache to clear the valid bits for all blocks in the cache, including

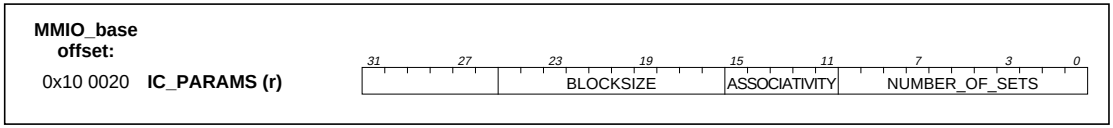


Figure 5-8. Format of the icache_parameters register.

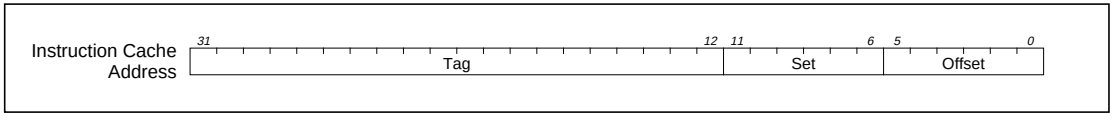


Figure 5-9. Instruction-cache address partitioning.

locked blocks. The LRU replacement status of all blocks is reset to their initialization value. The CPU is stalled while iclr is executing.

See Section 5.6, “Cache Coherency,” for further discussion of coherency issues.

5.4.8 Reading Tags and Cache Status

The instruction cache supports read access to its tag and status bits, but not with special operations as with the data cache. Since the instruction cache and branch units can execute only resultless operations, access to the instruction-cache tags and status bits is implemented using normal load operations that reference a special region in the MMIO address aperture. The region is 64 KB long and starts at MMIO_BASE. Instruction cache tags and status bits are read-only; store operations to this region have no effect. MMIO operations to this special region are only allowed by the DSPCPU, not by any other masters of the on-chip data highway, such as external PCI initiators.

Reading A Tag And Valid Bit. To read the tag and valid bit for a block in the instruction cache, a program can execute a ld32 operation directed at the instruction-cache region in the MMIO aperture. The top of Figure 5-10 shows the required format for the target address. The most-significant 16 bits must be equal to MMIO_BASE, the least-significant 15 bits select the block (by naming the set and set member), and bit 15 must be set to zero to perform a tag read.

A ld32 with an address as specified above returns a 32-bit result with the format shown at the top of Figure 5-11. Bit 20 contains the state of the valid bit, and the least-significant 20 bits contain the tag for the block addressed by the ld32.

Reading The LRU Bits. To read the LRU bits for a set in the instruction cache, a program can execute a ld32 operation as above but using the address format shown at the bottom of Figure 5-10. In this format, bit 15 is set to one to perform the read of the LRU bits, and the tag_i_mux field is set to zeros because it is not needed.

Reading the LRU bits produces a 32-bit result with the format shown at the bottom of Figure 5-11. The least-significant ten bits contain the state of the LRU bits when the ld32 was executed. See Section 5.5.4, “LRU Bit Definitions,” for a description of the LRU bits.

Note that the tag_i_mux and set fields in the address formats of Figure 5-10 are larger than necessary for the instruction cache in TM1000. These fields will allow future implementations with larger instruction caches to use a compatible mechanism for reading instruction cache information. The tag_i_mux field can accommodate a cache of up to 16-way set-associativity, and the set field can accommodate a cache with up to 512 sets. For TM1000, the following constraints of the values of these fields must be observed:

1. 0 <= tag_i_mux <= 7
2. 0 <= set <= 63

5.4.9 Cache Locking

Like the data cache, the instruction cache allows up to one-half of its blocks to be locked. A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until the locked status is changed explicitly by software. Thus, on TM1000, up to 16 KB of the cache can be used as a high-speed instruction ‘ROM.’ Only four out of eight blocks in any set can be locked.

The MMIO registers IC_LOCK_ADDR, IC_LOCK_SIZE, and IC_LOCK_CTL—shown in Figure 5-12—are used to define and enable instruction locking in the same way that the similarly named data-cache locking registers are used. Section 5.3.7, “Cache Locking,” describes the details of cache locking; they are not repeated here.

Setting the icache_lock_enable bit (in IC_LOCK_CTL) to ‘1’ causes the following sequence of events:

1. The instruction cache invalidates all blocks in the cache.
2. The instruction cache fetches all blocks in the lock range (defined by IC_LOCK_ADDR and IC_LOCK_SIZE) from main memory into the cache.
3. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

The only difference between this sequence and the initialization sequence for data-cache locking is that dirty blocks (which cannot exist in the instruction cache) are not first written back.

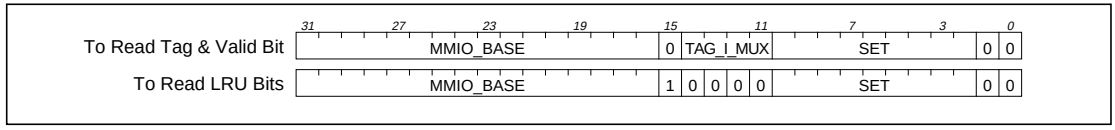


Figure 5-10. Required address format for reading instruction-cache tags and status.

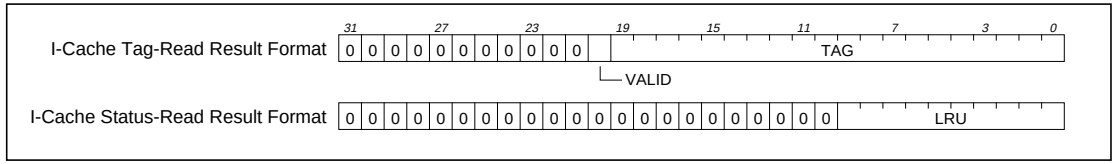


Figure 5-11. Result formats for reads from the instruction-cache region of the MMIO aperture.

Programmer's note: Programmers (or compilers) must combine all instructions that need to be locked into the single linear instruction-locking address range.

Programmer's note: The special iclr operation also removes locked blocks from the cache. If blocks are locked in the icache, then icache locking should be disabled in software (by writing '0' to IC_LOCK_CTL) before an iclr operation is issued.

5.4.10 Instruction Cache Initialization and Boot Sequence

When TM1000 is reset, the instruction cache executes an initialization and processor boot sequence. While reset is asserted, the instruction cache forces NOP operation to the DSPCPU, and the program counter is set to the default value reset_vector. When reset is deasserted, the initialization and boot sequence is as follows.

1. The stall signal is asserted to prevent activity in the DSPCPU and data cache.
2. The valid bits for all blocks in the instruction cache are reset.
3. At the completion of the block invalidation scan, the stall signal to the DSPCPU and data cache are deasserted.
4. The DSPCPU begins normal operation with an instruction fetch from the address reset_vector.

The initialization process takes 512 clock cycles. Reset sets reset_vector equal to DRAM_BASE so that execution starts at the default value of DRAM_BASE. As defined in Section 5.2, "DRAM Aperture," DRAM_BASE is set to 0x0 at reset; thus, after reset, TM1000 executes the boot code located at 0x0 in main memory.

5.5 LRU ALGORITHM

When a cache miss occurs, the block containing the requested data must be brought in to the cache, and this requested block must replace an existing block in the cache. The LRU algorithm is responsible for selecting the replacement victim, and the algorithm attempts to select the least-recently-used block.

The eight-way set-associative caches implement a hierarchical LRU replacement algorithm, which works as follows:

- The eight sets are partitioned into four groups of two elements each. To select the LRU element:
- First, the LRU pair out of the four pairs is selected using a four-way LRU algorithm.
- Second, the LRU element of the pair is selected using a two-way LRU algorithm.

5.5.1 Two-Way Algorithm

The two-way LRU requires an administration of one bit per pair of elements. On every cache hit to one of the two blocks, the cache writes once to this bit (just a write, not a read-modify-write). If the even-numbered block is accessed, the LRU bit is set to one; if the odd-numbered block is accessed, the LRU bit is set to zero. On a miss, the cache replaces the LRU element, i.e. if the LRU bit is zero, the even numbered element will be replaced; if the LRU bit is one, the odd numbered element will be replaced.

5.5.2 Four-Way Algorithm

For administration of the four-way algorithm, the cache maintains an upper-left triangular matrix R of one-bit elements without the diagonal. R contains six bits (in general, n×(n–1)/2 bits for n-way LRU). If set element k is referenced, the cache sets row k to one and column k to zero:

$$R[k, 0..n-1] \leftarrow 1,$$
$$R[0..n-1, k] \leftarrow 0$$

The LRU element is the one for which the entire row is zero (or empty) and the entire column is one (or empty):

$$R[k, 0..n-1] = 0 \textit{ and } R[0..n-1, k] = 1$$

For a four-way set-associative cache, this algorithm requires six bits per set of four cache blocks. On every cache hit, the LRU info is updated by setting three of the six bits to zero or one, depending on the set element that was accessed. The bits need only be written, no read-modify-write is necessary. On a miss, the cache reads the six LRU bits to determine the replacement block.

TM1000 combines the two-way and four-way algorithms into an eight-way hierarchical LRU algorithm. A total of ten administration bits are required: six to maintain the four-way LRU plus four bits maintain the four two-way LRUs.

The hierarchical algorithm has performance close to full eight-way LRU, but it requires far fewer bits—ten instead of 28 bits—and is much simpler to implement.

To update the LRU bits on a cache hit to element j (with 0 <= j <= 7), the cache applies m = (j div 2) to the four-way LRU administration and (j mod 2) is applied to the

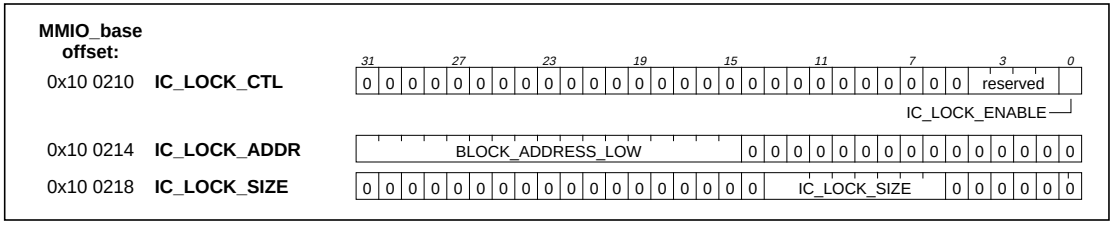


Figure 5-12. Formats of the registers that control instruction-cache locking.

two-way administration of pair m. To select a replacement victim, the cache first determines the pair p from the four-way LRU and then retrieves the LRU bit q of pair p. The overall LRU element is the p×2+q.

5.5.3 LRU Initialization

Reset causes the LRU administration bits to initialized to a legal state:

R[1,0] ← R[2,0] ← R[3,0] ← 1
R[2,1] ← R[3,1] ← R[3,2] ← 0
2_way[3] ← 2_way[2] ← 2_way[1] ← 2_way[0] ← 0

5.5.4 LRU Bit Definitions

The ten LRU bits per set are mapped as shown in Figure 5-13. This is the format of the LRU field as returned by the special operation rdstatus for the data cache and a ld32 from MMIO space (see Section 5.4.8, “Reading Tags and Cache Status”) for the instruction cache.

5.5.5 LRU for the Dual-Ported Cache

For the TM1000 dual-ported data cache, two memory operations to the same set are possible in a single clock cycle. To support this concurrency, two updates of the LRU bits of a single set must be possible.

The following rules are used by TM1000:

1. LRU bits that are changed by exactly one port receive the value according to the algorithm described above.
2. LRU bits that are changed by both ports receive a value as if the algorithm were first applied for the access in port zero and then for the access in port one.

5.6 CACHE COHERENCY

The TM1000 hardware does not implement coherency between the caches and main memory. Generalized coherency is the responsibility of software, which can use the special operations dcb, dinvalid, and iclr to enforce cache/memory synchronization.

5.6.1 Example 1: Data-Cache/Input-Unit Coherency

Before the CPU commands the video-in unit to capture a video frame, the CPU must be sure that the data cache contains no blocks that are in the address region that the video-in unit will use to store the input frame. If the video-in unit performs its input function to an address region and the data cache does hold one or more blocks from that region, any of the following may happen:

- A miss in the data cache may cause a dirty block to be copied back to the address region being used by the video-in unit. If the video-in unit already stored data in the block, the write-back will corrupt the frame data.
- The CPU will read stale data from the cache instead of from the block in main memory. Even though the video-in unit stored new video data in the block in main memory, the cache contents will be used instead because it is still valid in the cache.

To prevent erroneous copybacks or the use of stale data, the CPU must use dinvalid operations to invalidate all blocks in the address region that will be used by the video-in unit.

5.6.2 Example 2: Data-Cache/Output-Unit Coherency

Before the CPU commands the video-out unit to send a frame of video, the CPU must be sure that all the data for the frame has been written from the data cache to the region of main memory that the video-out unit will output. Explicit action is necessary because the data cache—with its copyback write policy—will hold an exclusive copy of the data until it is either replaced by the LRU algorithm or the CPU explicitly forces it to be copied back to main memory.

Before an output command is issued to the video-out unit, the CPU must execute dcb operations to force coherency between cache contents and main memory.

5.6.3 Example 3: Instruction-Cache/Data-Cache Coherency

If code prepared by a program running on the CPU must be subsequently executed, coherency between the instruction and data caches must be enforced. This is accomplished by a two-step process:

1. Coherency between the data cache and main memory must be enforced since the instruction cache can fetch instructions only from main memory.
2. Coherency between the instruction cache and main memory is enforced by executing an iclr operation.

The CPU will now be able to fetch and execute the new instructions.

5.6.4 Example 4: Instruction-Cache/Input-Unit Coherency

When an input unit is used to load program code into main memory, the iclr operation must be issued before attempting to execute the new code.

LRU bit 9	*LRU bit 8*	*LRU bit 7*	*LRU bit 6*	*LRU bit 5*	*LRU bit 4*	*LRU bit 3*	*LRU bit 2*	*LRU bit 1*	*LRU bit 0*
2_way[3]	2_way[2]	2_way[1]	2_way[0]	R[1,0]	R[2,1]	R[2,0]	R[3,2]	R[3,1]	R[3,0]

Figure 5-13. LRU bit definitions; 2_way[k] is the two-way LRU bit of pair k = (j div 2) for set element j.

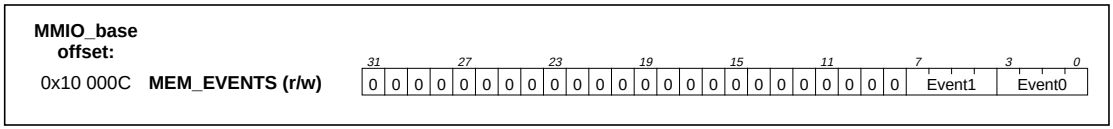


Figure 5-14. Format of the memory_events MMIO register.

5.7 PERFORMANCE EVALUATION SUPPORT

The caches implement support for performance evaluation. Several events that occur in the caches can be counted using the TM1000 timer/counters, by selecting the source CACHE1 and/or CACHE2, as described in Section 3.6, “Timers.” Two different events can be tracked simultaneously by using 2 timers.

The MMIO register MEM_EVENTS determines which events are counted. See Figure 5-14 for the format of MEM_EVENTS. Table 5-12 lists the events that can be tracked and the corresponding values for the MEM_EVENTS fields. Event1 selects the actual source for the TIMER CACHE1 source. Event2 selects the source for TIMER CACHE2.

Table 5-12. Trackable Cache-Performance Events

Encoding	Event
0	No event counted
1	Instruction-cache misses
2	Icache stall cycles (including dcache stall cycles if both icache and dcache are stalled simultaneously)
3	Data-cache bank conflicts
4	Data-cache read misses
5	Data-cache write misses
6	Data-cache stall cycles (that are not also Icache stall cycles)
7	Data-cache copyback to SDRAM
8	Copyback buffer full
9	Dcache write miss with all fetch units occupied
10	Dcache stream miss
11	Prefetch operation started and not discarded
12	Prefetch operation discarded (because it hits in the cache or there is no fetch unit available)
13	Prefetch operation discarded (because it hits in the cache)

Table 5-12. Trackable Cache-Performance Events

Encoding	Event
14–15	Reserved

5.8 MMIO REGISTER SUMMARY

Table Table 5-13 lists the MMIO registers that pertain to the operation of TM1000’s instruction and data caches.

Table 5-13. MMIO Register Summary

Name	Description
DRAM_BASE	Sets location of the DRAM aperture
DRAM_LIMIT	Sets size of the DRAM aperture
DRAM_CACHEABLE_LIMIT	Divides DRAM aperture into cacheable and non-cacheable portions
MEM_EVENTS	Selects which two events will be counted by timer/counters
DC_LOCK_CTL	Data-cache locking enable
DC_LOCK_ADDR	Sets low address of the data-cache address lock aperture
DC_LOCK_SIZE	Sets size of the data-cache address lock aperture
DC_PARAMS	Read-only register with data-cache parameter information
IC_PARAMS	Read-only register with instruction-cache parameter information
IC_LOCK_CTL	Instruction-cache locking enable
IC_LOCK_ADDR	Sets low address of the instruction-cache address lock aperture
IC_LOCK_SIZE	Sets size of the instruction-cache address lock aperture
MMIO_BASE	Sets location of the MMIO aperture

Video In Chapter 6

by Gert Slavenburg

6.1 SUMMARY OF FUNCTIONS

The Video In (VI) unit provides the following functions:

- Digital video input from a digital camera or analog camera (using a video decoder).
- High bandwidth (38 MB/sec) raw input data channel.
- Direct 8-10 bit interface for video A/D converters at up to 38-MHz sample rate.
- Receiver port for TM1000-to-TM1000 unidirectional message passing

The Video In unit operates in one of the modes as per Table 6-1.

Table 6-1. Video In Mode Selection.

Mode	Function	Explanation
0000	fullres capture	YUV 4:2:2 capture without decimation
0001	halfres capture	YUV 4:2:2 capture with decimate by 2
0010	raw8 capture	raw 8 bits data capture, pack 4 bytes to a word
0011	raw10s capture	raw 10 bits data capture, sign extend to 16 bits, pack 2 to a word
0100	raw10u capture	raw 10 bits data capture, zero-extend to 16 bits, pack 2 to a word
0101	message passing	VO to VI message passing
0110 .. 1111	Reserved	

Digital video input is in YUV 4:2:2 with eight-bit resolution multiplexed in CCIR656 format[1] from a digital camera or CCIR656 capable video decoder (such as the Philips SAA7111), across an eight-bit-wide interface. Resolutions up to CCIR601 are accepted at 50 or 60 fields per second. A programmable rectangular image is captured from a video frame and written in *planar format* to TM1000 SDRAM. The video camera or decoder can be programmed using the TM1000 I^2C bus. In *fullres capture* mode, luminance (Y) and chrominance (U, V) pass unmodified. In *halfres capture* mode, luminance and chrominance are horizontally decimated by a factor of two to convert to CIF-like resolution with YUV 4:2:2 or MPEG sampling rules. If vertical subsampling on chrominance is desired, it is performed by software on the DSPCPU or by the on-chip Image Coprocessor (ICP).

When operating as raw input data channel, VI accepts eight-bit-wide data. The operation mode is *raw8 capture.* No data selection or data interpretation is done. Data is written in packed form, four bytes to a word, to local SDRAM. There is no hardware control over the rate at which the source sends data. Instead, VI maintains two pointer/counter registers to ensure that no data is lost when the local SDRAM memory buffer fills. Data is accepted at the clock of the sender. If desired, VI_CLK can be programmed as an output to drive the data transfer at a programmable rate.

VI can accept data from up to 10-bit A/D converters, at sampling rates up to 38 MHz. VI can operate in *raw8, raw10u,* or *raw10s capture* mode for eight-bit, unsigned 10-bit or signed 10-bit data. In the 10-bit modes, data is zero- or sign-extended to 16 bits and stored in packed form in local SDRAM. As with the *raw8-capture* mode, VI maintains two pointer/counter registers to ensure that no data is lost when the local SDRAM memory buffer fills. Data is accepted at the externally set sampling rate. If desired, VI_CLK can be programmed as an output to serve as a programmable sampling clock.

VI can act as receiver from the Video Out unit of another TM1000. One Video Out can broadcast to multiple receiving VI's. In this *message passing* mode, no data selection or data interpretation is done. Each message of the sender is written as byte-packed data to a separate local SDRAM memory buffer. Message start and end is indicated by the sender. The receiving VI will accept data until the sender indicates message end or until the current memory buffer is full. If the memory buffer fills before message end is encountered, the received data is truncated and an error condition is raised.

6.1.1 Interface

Besides the Video-In-specific pins in Table 6-2, the TM1000 I^2C interface is typically used to control the external camera or video decoder.

Figure 6-1 through Figure 6-4 illustrate typical connections for commonly used external sources. Note that VI_DVALID is only used in special circumstances, e.g. when sending data through a channel that results in clock periods both with and without data transfers.

1. Refer to CCIR recommendation 656: Interfaces for digital component video signals in 525 line and 625 line television systems. Recommendation 656 is included in the Philips Desktop Video Data Handbook.

Table 6-2. Video In Interface Pins

VI_CLK	I/O-5	• If configured as input (power up default): A positive transition on this incoming video clock pin samples all other VI_DATA input signals below if VI_DVALID is HIGH. If VI_DVALID is LOW, VI_DATA is ignored. Clock and data rates of up to 38 MHz are supported to allow for 16:9 aspect ratio video with 5% clock margin. • If configured as output: Programmable output clock to drive an external video A/D converter. Can be programmed to emit integral dividers of DSPCPU_CLK. • See section 6.2 for clock programming details.
VI_DVALID	IN-5	VI_DVALID indicates that valid data is present on the VI_DATA lines. If HIGH, VI_DATA will be accepted on the next VI_CLK positive edge. If LOW, no VI_DATA will be sampled.
VI_DATA[7:0]	IN-5	CCIR656 style YUV 4:2:2 data from a digital camera, or general purpose high speed data input pins. Sampled on VI_CLK if VI_DVALID HIGH.
VI_DATA[9:8]	IN-5	Extension high speed data input bits to allow use of 10 bit video A/D converters. Sampled on VI_CLK if VI_DVALID HIGH. VI_DATA[8] serves as START and VI_DATA[9] as END message input in message passing mode.

6.1.2 Diagnostic Mode

The Video-In logic can be set to operate in diagnostic mode, which connects the inputs of VI to the outputs of Video Out. This mode provides boot diagnostics with the ability to verify major operational aspects of the chip before handing control to an operating system.

Diagnostic mode is entered by writing a control word with a '1' in the DIAGMODE bit position to the VI_CTL register (see Figure 6-11). This has to be done after setting the input clock for Video-In (coming from Video-Out). After a Video-In software reset, the DIAGMODE bit has to be set back to '1'. In diagnostic mode, the Video In signals are exactly as shown in Figure 6-2, except that the inputs come from the on-chip Video Out unit. Note that the inputs are truly taken from the TM1000 Video-Out external pins, i.e. if an external (board level) source is driving VO_CLK and Video-Out block is the clock master, diagnostic mode is not capable of testing Video-Out.

Note that the diagnostic mode only controls an input multiplexer. VI can be programmed and operated in all usual modes. The *raw* modes are particularly attractive for diagnostics purposes, since they allow VI to operate almost as an on-chip logic analyzer.

6.1.3 Power Down

The Video In logic participates in global TM1000 chip power down, unless the SLEEPLESS bit in the VI_CTL register is asserted.

6.1.4 Hardware and Software Reset

Video In is reset by a TM1000 hardware reset or by a Video In software reset. The latter is accomplished by writing a control word of 0x00080000 to the VI_CTL register. After a software reset, allow for 5 video clock cycles delay before enabling Video In capture. Upon hardware or software reset, the VI_CTL, VI_STATUS, and VI_CLOCK registers are set to all zeros. Note that the Video-In clock has to be present while applying the software reset.

6.2 CLOCK GENERATOR

The Video In block can operate in two distinct clocking modes, as controlled by the VI_CLOCK control register (see Figure 6-11):

SELFCLOCK = 0: '"External clocking mode". This is the most common mode of operation. In this mode, the VI_CLK pin is an asynchronous clock input. All other inputs are sampled on positive edges of the VI_CLK clock signal. On chip synchronizers ensure reliable asynchronous capture, with an MTBF of 6 months or greater. This mode can be combined with DIAGMODE, in which case the Video Out clock acts as the asynchronous clock source. In external clocking mode, the value of DIVIDER is ignored.

SELFCLOCK = 1: "Internal clocking mode". This mode is typically intended for use with external A/D converters or other sources that require a clock. In this mode, VI_CLK is an output pin. Positive edges of VI_CLK are used to sample all other inputs. The generated clock frequency can be programmed using the DIVIDER field in the VI_CLOCK register.

$$f_{VICLK} = \frac{f_{DSPCPU}}{DIVIDER}$$

On RESET, VI_CLOCK is set to zero, i.e. external clocking mode is the default with DIVIDER ignored.

6.3 FULLRES CAPTURE MODE

In *fullres capture* mode Video In receives all three video components Y, U, and V, as well as synchronization information (SAV and EAV codes) on the VI_DATA[7:0] pins in CCIR656 format. See Figure 6-8. The three video components Y, U, and V are separated into three different streams. Each component is written in packed form into separate Y, U, and V buffers in the SDRAM. This is commonly called a *planar* format[1] (see Figure 6-10).

The CCIR656 standard specifies that the camera has to obey the sampling rules illustrated in Figure 6-5. VI is capable of chrominance resampling, and can produce samples in memory in two ways:

VI_CTL.SC=0. "Co-sited sampling" places luminance and chrominance samples in memory without any modi-

1. The *planar format* is most suitable as input to software compression algorithms.

fication. Hence, a planar format results with sampling positions as per co-sited luminance and chrominance YUV 4:2:2 convention.

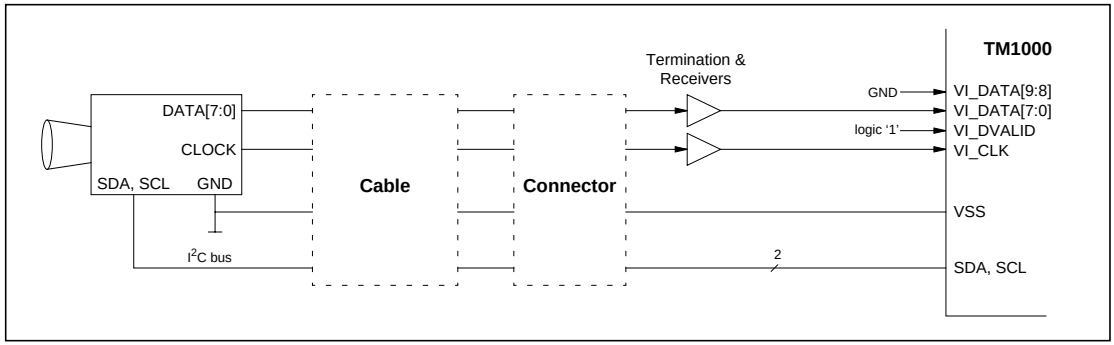


Figure 6-1. Video In connected to an 8-bit CCIR656 digital camera.

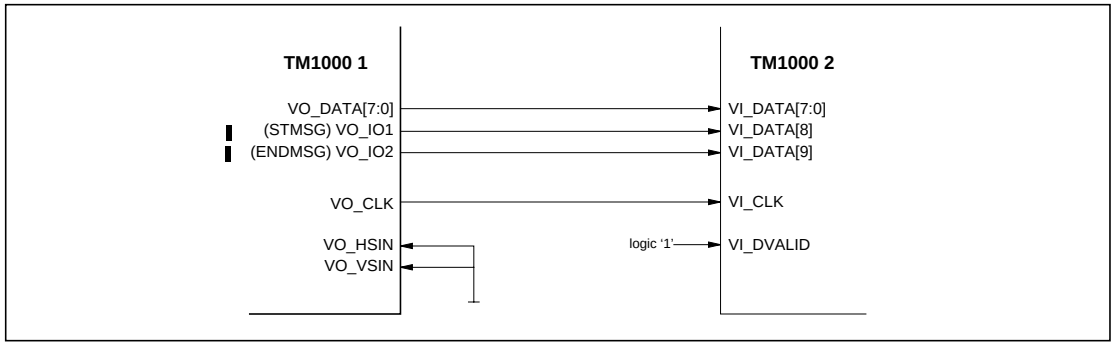


Figure 6-2. Video In connected to Video Out.

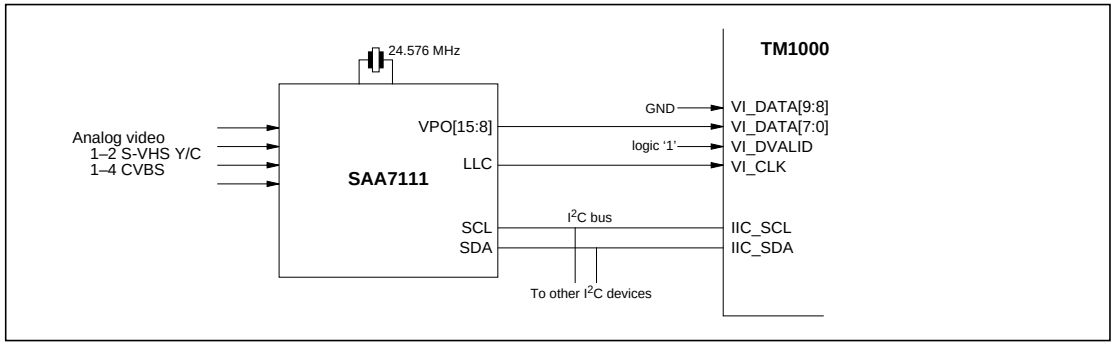


Figure 6-3. Video In connected to a video decoder.

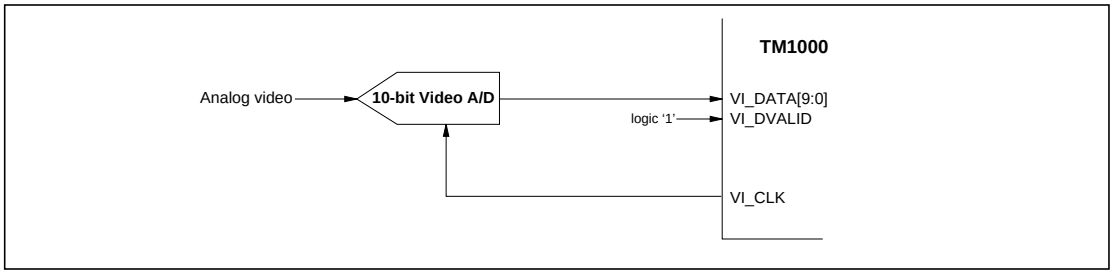


Figure 6-4. Video In connected to a 10-bit video A/D converter.

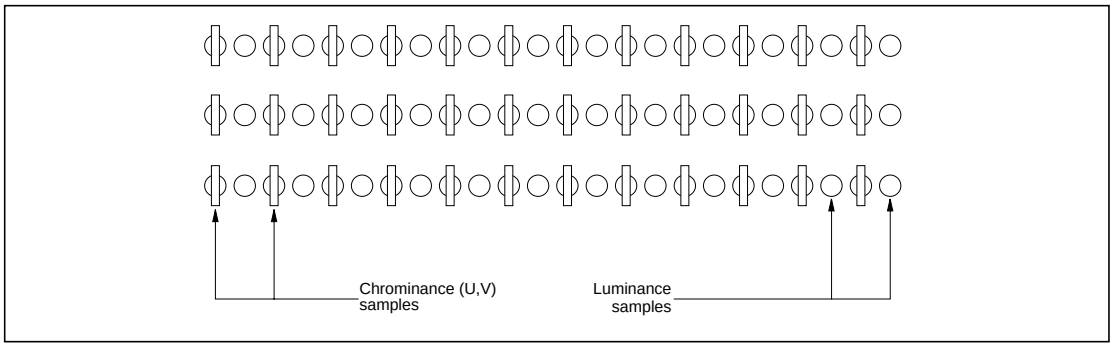


Figure 6-5. Camera YUV 4:2:2 sampling (co-sited luminance/chrominance).

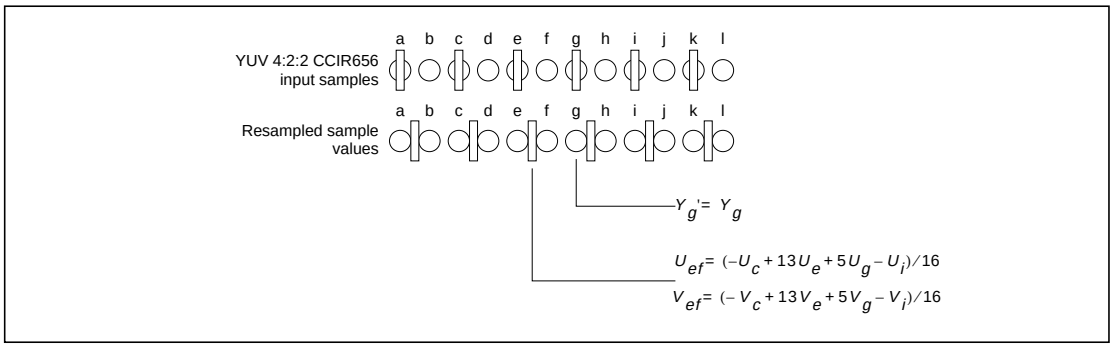


Figure 6-6. Chrominance re-sampling to achieve interspersed sampling.

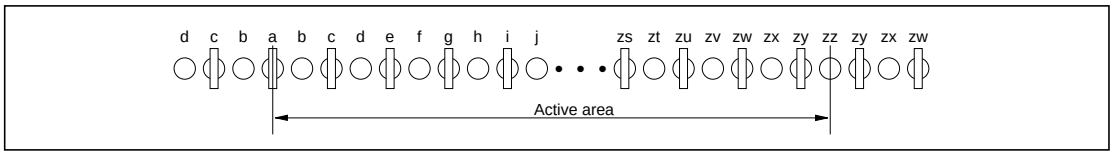


Figure 6-7. Filtering at the edge of the active area.

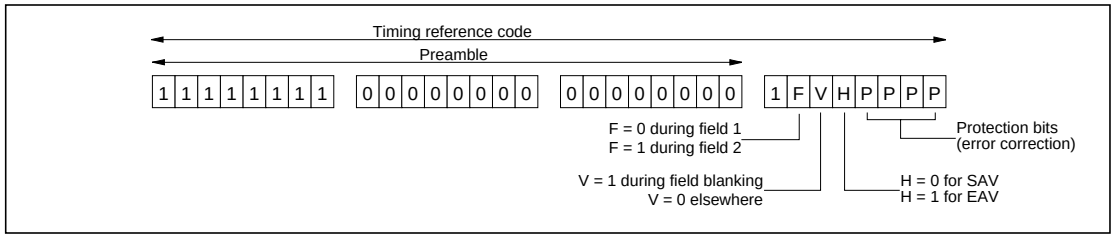


Figure 6-8. Format of CCIR656 SAV and EAV timing reference codes.

VI_CTL.SC=1: "Interspersed sampling" applies a (–1 13 5 –1)/16 filter as illustrated in Figure 6-6 to the chrominance samples before writing them to memory. This filter computes chrominance values at sample points midway between luminance samples[1]. The resulting memory data format is preferred by some video compression standards. The MPEG-1 standard, for example, requires YUV 4:2:0 data with chrominance sampling positions horizontally and vertically midway between luminance samples. This can be achieved from the horizontally interspersed sampling format by vertical subsampling with a (1 1) / 2 or more sophisticated filter. Vertical filtering can be performed by software using the DSPCPU's efficient multi-media operations or by hardware in the Image Coprocessor (ICP).

The filtering process exercises special care at the left and right edges of the active area of the CCIR656 data stream, as defined by the SAV, EAV code positions. See Figure 6-7. Since no pixels exist to the left of the first pixel, nor to the right of the last pixel, filtering can result in artifacts. To minimize artifacts, the image is extended by mirroring pixels around the left-most and right-most pixel. Note that the image is mirrored around pixel 'a', the first pixel after the SAV code and around pixel 'zz', the last pixel before the EAV[2] code. Pixel 'a' in Figure 6-7 is the (chroma, luma) pair defined by the first three camera bytes of the UYVYUYVY... stream after SAV.

Refer to Figure 6-11 for an overview of the memory-mapped I/O (MMIO) registers that are used to control and observe the operation of VI in fullres capture mode.

Upon hardware or software reset (Section 6.1.4, "Hardware and Software Reset"), the VI_CTL, VI_STATUS, and VI_CLOCK registers are set to all zeros.

At any point in time, the VI STATUS register fields (see Figure 6-11) indicate the current camera status:

- CUR_X: The pixel index (0 to M–1) of the most recently received camera pixel. CUR_X gets set to zero for the first pixel following receipt of a SAV code[3], and incremented on every valid Y sample received thereafter.
- CUR_Y: The line index (0 to N–1) of the camera line that is currently being received. CUR_Y gets set to zero upon receipt of a negative edge of V, i.e., upon the first SAV code containing V=0 after one or more SAV codes containing V=1. This is equivalent to the first line after the end of vertical retrace. CUR_Y gets incremented upon every successive SAV code.
- FIELD2: Indicates whether the field currently being received is a field1 or 2. This flag gets updated based on the F field of every received SAV code. Note that field1 is the 'top' field, i.e. the field containing the top-most visible line. Field1 contains lines 1,3,5 etc. Field2 contains lines 2,4,6,8 etc.

Table 6-3 illustrates common digital camera standards and the number of active pixels per line, lines per field and fields per second. Note that any source is acceptable to VI, as long as the maximum VI_CLK rate is not exceeded.

Figure 6-9 shows the details of an incoming field and the captured image. The incoming field consists of N horizontal lines, each line having M pixels labeled 0 through M–1. Lines are numbered from 0 through N–1. The captured image is a subset of the incoming image. It is defined by the capture parameters (START_X, START_Y, WIDTH, HEIGHT) held in the VI_CAP_START and VI_CAP_SIZE MMIO registers (see Figure 6-11).

1. All filters perform full precision intermediate computations and saturation upon generating the result bits.
2. EAV codes with multiple bit errors are accepted and do enable the mirroring function.
3. Note that VI uses the SAV protection bits to implement single error correction and double error detection. An SAV code with double error is ignored.

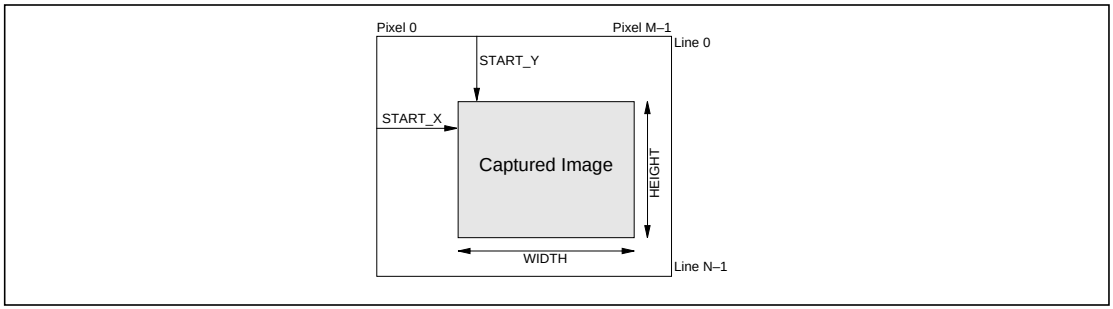


Figure 6-9. Video-in capture parameters.

- START_X: Defines the starting pixel number or (X-coordinate of the starting pixel). START_X must be even.
- START_Y: Defines the starting line number or (Y-coordinate of the starting pixel).
- WIDTH: Defines the width of the captured image in pixels. WIDTH must be even.
- HEIGHT: Defines the height of the captured image in lines.

Table 6-3. Common Video Source Parameters.

Video Source	M (# active pixels)	N (# active lines)	Field Rate (Hz)
CCIR601 50 Hz/625 lines	720	288	50
CCIR601 60 Hz/525 lines	720	240	60
square pixel 50 Hz/625 lines	768	288	50
square pixel 60 Hz/525 lines	640	240	60

Image capture starts after the following conditions are met:

- VI_CTL.CAPTURE ENABLE is asserted.
- VI_STATUS.CAPTURE COMPLETE is de-asserted, indicating that any previously captured image has been acknowledged.
- CUR_Y = START_Y occurs.

Once image capture is started, HEIGHT ‘lines’ are captured. Each ‘line’ capture starts if:

- The previous line capture, if any, is completed.
- CUR_X = START_X

Once line capture starts, it continues for 2*WIDTH pixel clocks[1] in which VI_DVALID is asserted.

Note that capture continues regardless of any horizontal or vertical retrace and associated CUR_Y or CUR_X reset. This provides special applications with the ability to capture information embedded inside the horizontal or vertical blanking interval. If it is desirable to capture ’pixels’ in the horizontal blanking interval, a minimum time separation of 1 µs is required between the last pixel captured on line *y* and the first pixel captured on line *y+1*. An exception to this rule is allowed if and only if the storage parameters below are chosen such that the last and first pixel end up in adjacent memory locations. Note that blanking information capture only makes sense in fullres mode, with co-sited sampling. All other modes apply filtering, which will distort the data.

The captured image is stored in SDRAM at a location defined by the storage parameters in MMIO registers (Y_BASE_ADR, Y_DELTA, U_BASE_ADR, U_DELTA, V_BASE_ADR, V_DELTA). Note that the base-address registers force alignment to 64-byte boundaries (six LSBs are always zero). The default memory packing is big-endian although little-endian packing is also supported by setting the LITTLE_ENDIAN bit in the VI_CTL register.

- Y_BASE_ADR: The desired starting (byte) address in SDRAM memory where the first Y (Luminance) sample of the captured image will be stored. This address is forced to be 64-byte aligned (six LSBs always zero).
- Y_DELTA: The desired address difference between the last sample of a line and the address of the first sample on the next line. Note that the value of Y_DELTA must be chosen so that all line-start addresses are 64-byte aligned.
- U_BASE_ADR, U_DELTA, V_BASE_ADR, V_DELTA: Same functions and alignment restrictions as above, but for chrominance-component samples.

Horizontally-adjacent samples are stored at successive byte addresses, resulting in a packed form (four 8-bit samples are packed into one 32-bit word). Upon horizontal retrace, pixel storage addresses are incremented by the corresponding DELTA to compute the starting byte address for the next line. Note that DELTA is a 16-bit unsigned quantity. This process continues until HEIGHT lines of WIDTH samples have been stored in memory for luminance (Y). For chrominance, HEIGHT lines of half the WIDTH are stored[2]. See Figure 6-10.

Modifications to Y_BASE_ADR, U_BASE_ADR and V_BASE_ADR have no effect until the start of next cap-

1. Four clocks for each C_b,Y,C_r,Y group representing two luminance pixels

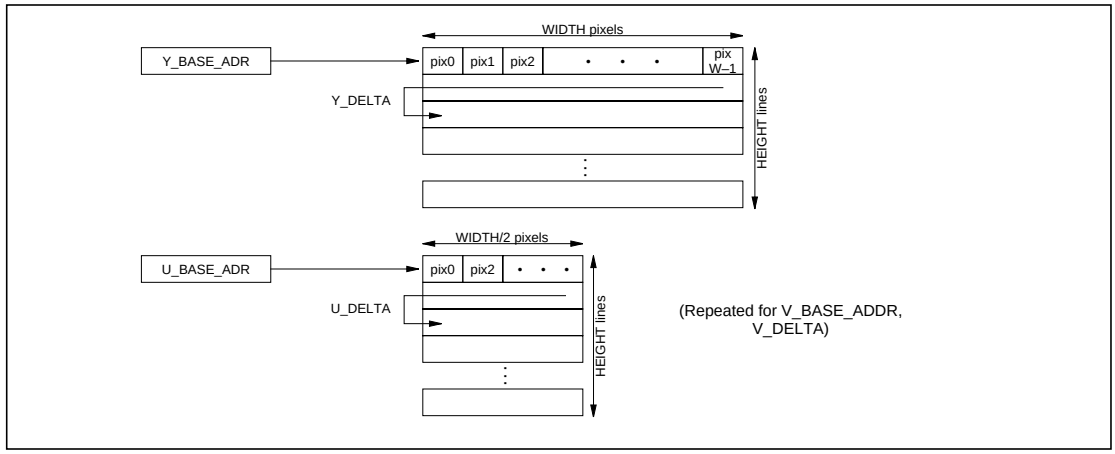


Figure 6-10. Video In YUV 4:2:2 planar memory format.

ture, i.e. VI hardware maintains a separate pointer to track the current address. Modifications to Y_DELTA, U_DELTA and V_DELTA do affect the next horizontal retrace. Hence, under normal circumstances, the DELTA variables should not be changed during capture.

When capture is complete, i.e. any internal VI buffers have been flushed and the entire captured image is in local SDRAM,VI raises the STATUS register flag CAPTURE COMPLETE. If enabled in the VI_CTL register, this event causes a DSPCPU interrupt to be requested.

The programmer can determine whether the captured image is a field1 or field2 by inspection of the FIELD2 flag in VI_STATUS. Note that the FIELD2 flag changes at the start of the vertical blanking interval of the next field.

2. Note that consecutive pixel components of each line are stored in consecutive memory addresses but consecutive lines need not be in consecutive memory addresses

The CAPTURE COMPLETE flag is cleared by writing a word to VI_CTL with a ‘1’ in the CAPTURE COMPLETE ACK bit position.This prepares VI for the capture of the next image.

The user can program the Y_THRESHOLD field to generate pre-completion (or post-completion) interrupts. Whenever CUR_Y reaches Y_THRESHOLD, the THRESHOLD REACHED flag in the STATUS register is set. If enabled in the VI_CTL register, this event causes a DSPCPU interrupt request. The THRESHOLD REACHED flag is cleared by writing a word to VI_CTL with a ‘1’ in the THRESHOLD REACHED ACK bit position. Note that, due to internal buffering in the Video In unit, it is NOT guaranteed that all samples from lines up to and including CUR_Y have been written to local SDRAM upon THRESHOLD REACHED. The implementation guarantees a fixed maximum time of 2 µs between raising the interrupt and completion of all writes to SDRAM. The THRESHOLD interrupt mechanism works regardless of CAPTURE ENABLE. Hence, it can also be used to skip a desired number of fields without constant DSPCPU polling of VI_STATUS.

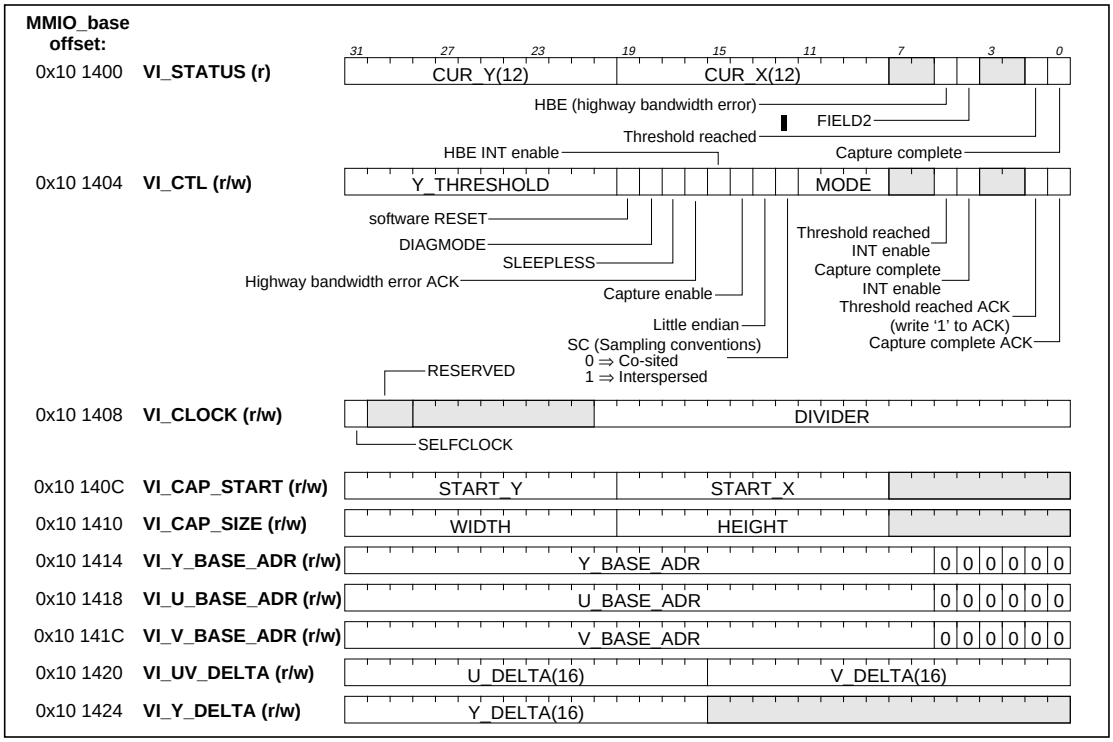


Figure 6-11. YUV capture view of Video In MMIO registers.

If VI internal buffers overflow due to insufficient internal data-highway bandwidth allocation, the HIGHWAY BANDWIDTH ERROR condition is raised in the VI_STATUS register. If enabled, this causes assertion of a VI interrupt request. Capture continues at the correct memory address as soon as the internal buffers can be written to memory, but one or more pixels may have been lost, and the corresponding memory locations are not written. The HBE condition can be cleared by writing a '1' to the HIGHWAY BANDWIDTH ERROR ACK bit in VI_CTL. Refer to Section 6.7, "Highway Latency and HBE" for more information.

Any interrupt event of VI (CAPTURE COMPLETE, THRESHOLD REACHED, HIGHWAY BANDWIDTH ERROR) leads to the assertion of a single VI interrupt (SOURCE 9) to the TM1000 Vectored Interrupt Controller. The interrupt handler routine should check the STATUS register to determine the set of VI events associated with the request. The vectored interrupt controller should always be set to have Video In (SOURCE 9) operate in level sensitive mode. This ensures that each event gets handled.

VI asserts the interrupt request line as long as one or more enabled events are asserted. The interrupt handler clears one or more selected events by writing a '1' to the corresponding ACK field in VI_CTL. The clearing of the last event leads to immediate (next DSPCPU clock edge) de-assertion of the interrupt request line to the Vectored Interrupt Controller. See Section 3.4.3, "INT and NMI (Maskable and Non-Maskable Interrupts)," for information on how to program interrupt handler routines.

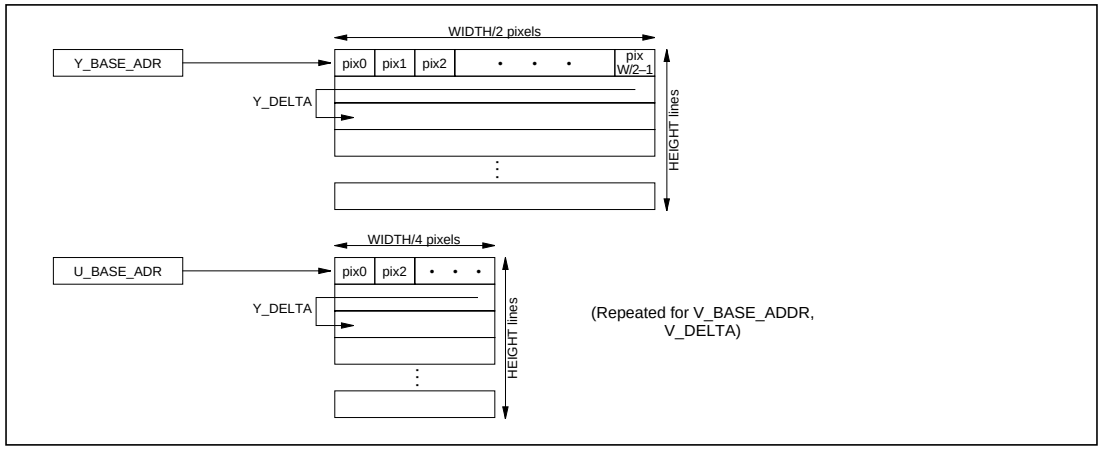


Figure 6-12. Video In halfres planar memory format.

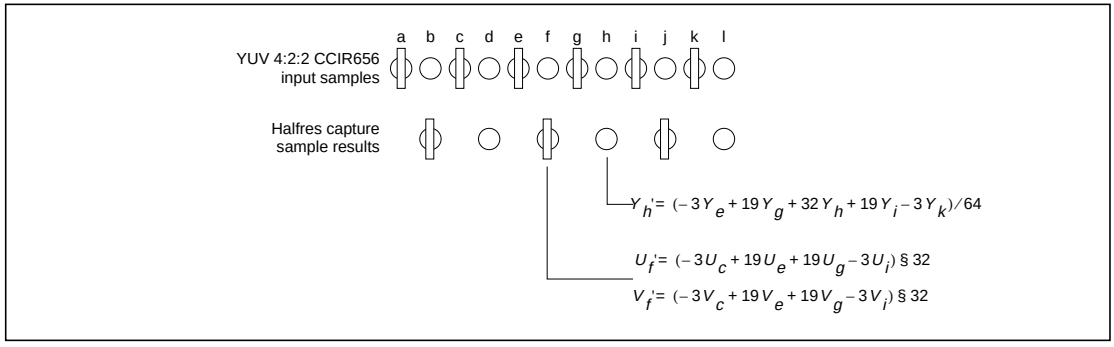


Figure 6-13. Halfres co-sited sample capture.

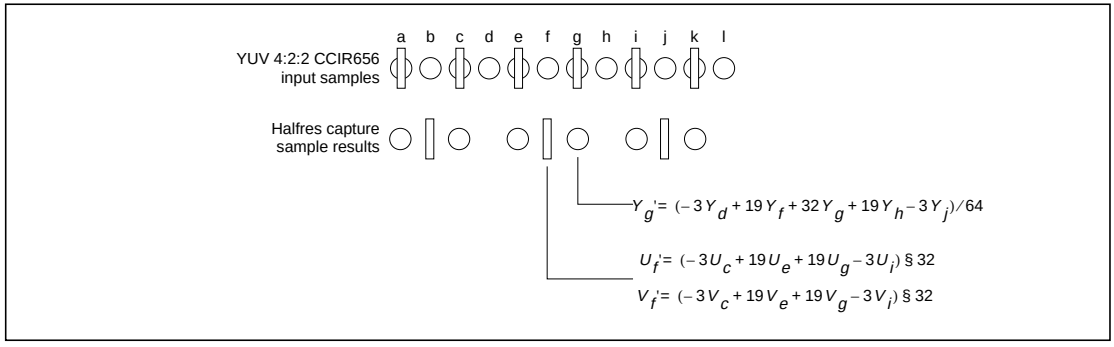


Figure 6-14. Halfres interspersed sample capture.

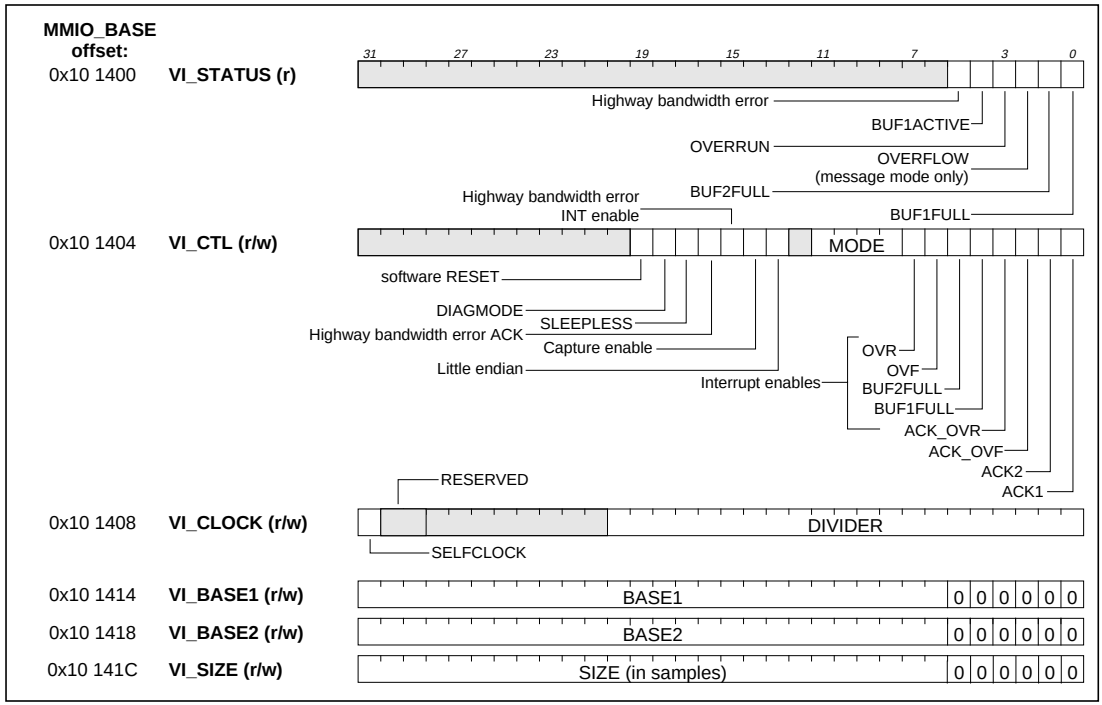


Figure 6-15. Raw & message passing modes view of Video In MMIO registers.

6.4 HALFRES CAPTURE MODE

Halfres capture mode is identical in operation to fullres capture mode except that horizontal resolution is reduced by a factor of two on both luminance and chrominance data.

Referring to Figure 6-9 and Figure 6-11, if VI is programmed to capture HEIGHT lines of WIDTH pixels in halfres mode, the resulting captured planar data is as shown in Figure 6-12. Note that WIDTH/2 luminance and WIDTH/4 chrominance samples are captured. In this mode, START_X and WIDTH must be a multiple of four.

Horizontal-resolution reduction is performed as shown in Figure 6-13 or Figure 6-14. The spatial sampling conventions of the pixels in memory depends on the SC (Sampling Convention) bit in the VI_CTL register. Assuming that the camera sampling positions obey the conventions shown in Figure 6-5, two possible spatial formats are supported in memory:

- If SC=0, co-sited luminance and chrominance samples result as shown in Figure 6-13. This corresponds to the standard YUV 4:2:2 sampling conventions.
- If SC=1, interspersed chrominance samples result, as shown in Figure 6-14. This form is (after vertical subsampling of the chroma components) identical to the MPEG-1 sampling conventions. If vertical subsampling is desired, it can either be performed in software on the DSPCPU, or in hardware using the Image Coprocessor (ICP).

The filtering process applies mirroring at the edge of the active video area, as per Figure 6-7.

6.5 RAW CAPTURE MODES

All raw capture modes (raw8, raw10s and raw10u) behave similarly. VI_DATA information is captured at the rate of the sender's clock, without any interpretation or start/stop of capture on the basis of the data values. Any clock cycle in which VI_DVALID is asserted leads to the capture of one data sample. Samples are eight or 10 bits long (raw8 versus raw10 modes). For the eight-bit capture mode, four samples are packed to a word. For the 10-bit capture modes, two samples (of 16 bits each) are packed to a word. The extension from 10 to 16 bits uses sign extension (raw10s) or zero extension (raw10u).

For 8-bit and 16-bit capture, successive captured values are written to increasing memory addresses. For 16-bit capture, the byte order with which the 16-bit data is written to memory is governed by the LITTLE ENDIAN bit. The VI LITTLE ENDIAN bit should be set the same as the DSPCPU endianness (PCSW.BSX). This ensures that the DSPCPU sees correct 16-bit data.

Figure 6-15 illustrates the 'raw mode' view of the VI MMIO registers. Figure 6-16 shows the major Video In states associated with raw-mode capture. The initial state is reached on software or hardware reset as described in Section 6.1.4, "Hardware and Software Reset". Upon reset, all status and control bits are set to ze-

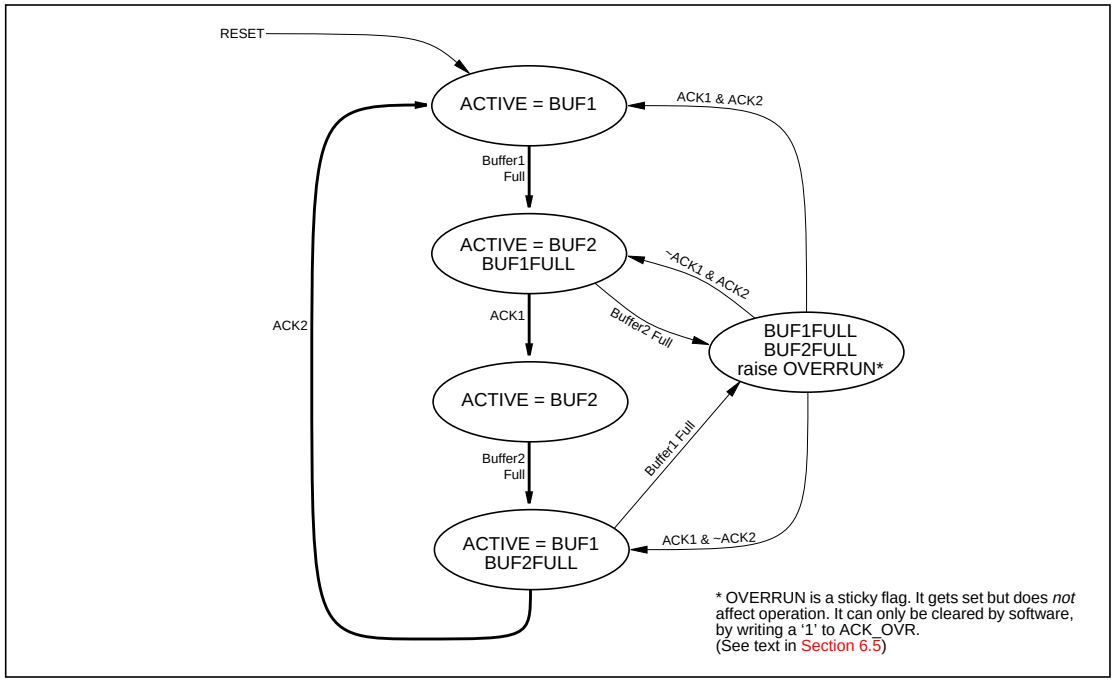


Figure 6-16. Video In raw mode major states.

ro. In particular, CAPTURE_ENABLE is set to 0 and no capture takes place.

Once the software has programmed BASE1 and BASE2 (with the start addresses of two SDRAM buffer areas[1]) and SIZE (in number of samples), it is safe to enable capturing by setting CAPTURE_ENABLE. Note that SIZE is in samples, and must be a multiple of 64, hence setting a minimum buffer size of 64 bytes for raw8 mode and 128 bytes for raw10 modes. At this point, buffer1 is the active capture buffer. Data is captured in buffer1 until capture is disabled or until SIZE samples have been captured. After every sample, a running address pointer is incremented by the sample size (one or two bytes). If SIZE samples have been captured, capture continues (without missing a sample) in buffer2. At the same time, BUF1FULL is asserted. This causes an interrupt on the DSPCPU, if enabled by BUF1FULL INTERRUPT ENABLE.

Buffer2 is now the active capture buffer, and behaves as described above. In normal operation, the DSPCPU will respond to the BUF1FULL event by assigning a new BASE1 and (optionally) SIZE and performing an ACK1. If the DSPCPU fails to assign a new buffer1 and perform an ACK1 before buffer2 also fills up, the OVERRUN condition is raised and capture stops. Capture continues upon receipt of an ACK1, ACK2, or both, regardless of the OVERRUN state. The buffer in which capture resumes is as indicated in Figure 6-16. The OVERRUN condition is 'sticky' and can only be cleared by software, by writing a '1' to the ACK_OVR bit in the VI_CTL register.

If insufficient bandwidth is allocated from the internal data highway, the VI internal buffers may overflow. This leads to assertion of the HIGHWAY BANDWIDTH ERROR condition. One or more data samples are lost. Capture resumes at the correct memory address as soon as the internal buffer is written to memory. The HBE error condition is sticky. It remains asserted until it is cleared by writing a '1' to HIGHWAY BANDWIDTH ERROR ACK. Refer to Section 6.7, "Highway Latency and HBE."

Note that VI hardware uses copies of the BASE and SIZE registers once capture has started. Modifications of BASE or SIZE, therefore, have no effect until the start of the next use of the corresponding buffer.

Note also that the VI_BASE1 and VI_BASE2 addresses must be 64-byte aligned (the six LSBs are always zero).

6.6 MESSAGE-PASSING MODE

In this mode, VI receives eight-bit message data over the VI_DATA[7:0] pins. The message data is written in packed form (four eight-bit message bytes per 32-bit word) to SDRAM. Message data capture starts on receipt of a START event on VI_DATA[8]. Message data is received until EndOfMessage (EOM) is received on VI_DATA[9] or the receive buffer is full. Figure 6-17 illustrates an example of an eight-byte message transfer. The first byte (D0) is sampled on the rising edge of the VI_CLK clock after a valid START was sampled on the

1. SDRAM buffers must start on a 64 byte boundary.

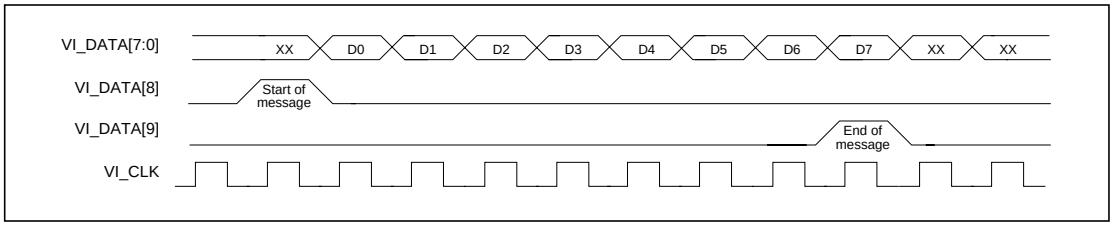


Figure 6-17. Video In message passing signal example.

clock edge before. The last byte (D7) is sampled on the clock during which EOM was asserted.

The message passing mode view of the VI MMIO registers is shown in Figure 6-15. The major states are shown in Figure 6-18. The operation is almost identical to the operation in raw-capture mode, except that transitions to another active buffer occur upon receipt of EOM rather than on buffer full. Overrun is raised if the second buffer receives a complete message before a new buffer is assigned by the DSPCPU.

OVERFLOW is raised if a buffer is full and no EOM has been received. If enabled, it causes a DSPCPU interrupt. Since digital interconnection between devices is reliable, overflow is indicative of a protocol error between the two TM1000's involved in the exchange (failure to agree on message size). Detection of overflow leads to total halt of capture of this message. Capture resumes in the next buffer upon receipt of the next START event on VI_DATA[8]. The OVERFLOW flag is sticky and can only be cleared by writing a '1' to ACK_OVF.

Highway Bandwidth Error behavior in message passing mode is identical to that of raw mode.

6.7 HIGHWAY LATENCY AND HBE

Refer to Section 19.11, "Example" for a description of the arbiter terminology used here. Video In uses internal buffering before writing data to SDRAM. There are two internal buffers, each 64 entries of 32 bits.

In fullres mode, each internal buffer is used for 128 Y samples, 64 U samples and 64 V samples. Once the first internal buffer is filled, 4 highway transactions need to occur before the second buffer fills completely. Hence, the requirement for not loosing samples is:

4*lat+4*T+19 <= 256 Video In clocks

For the typical CCIR601 resolution NTSC or PAL 27 MHz Video In clock rate, TM1 highway clock speed of 100 MHz, and given T=16, latency for the highway should hence be set to less than 216 clock cycles.

In halfres mode,

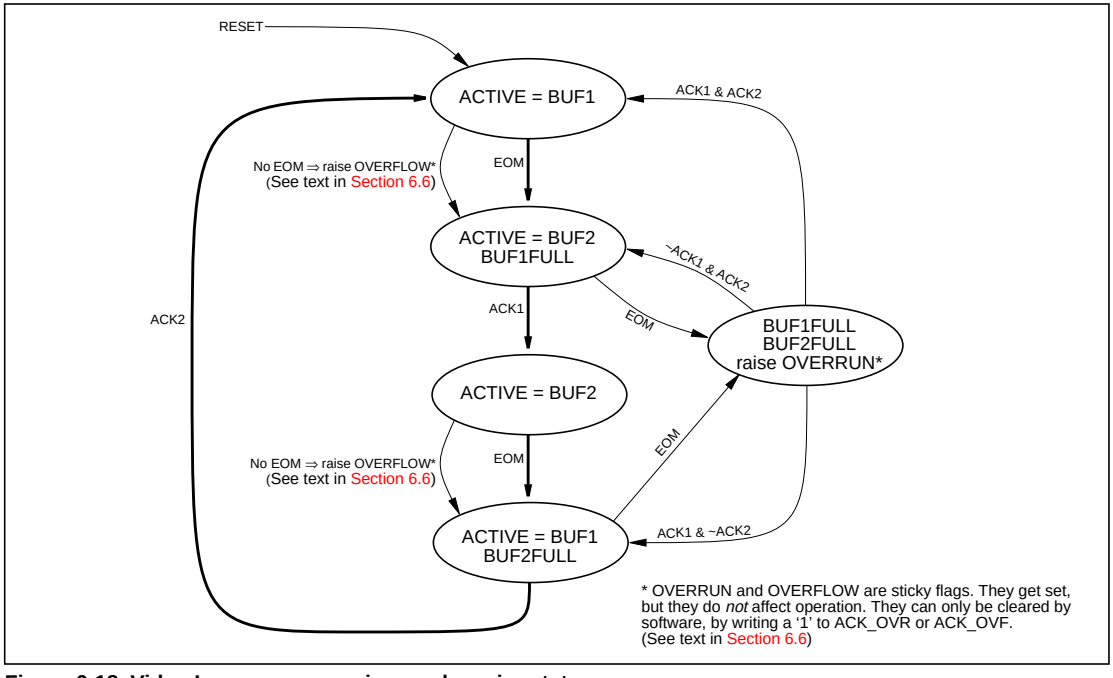


Figure 6-18. Video In message passing mode major states.

Video Out Chapter 7

by Dave Wyland, Gert Slavenburg

7.1 SUMMARY OF FUNCTIONS

The TM1000 Video Out unit (VO) connects to an off chip video subsystem such as a digital video encoder chip (DENC), a digital video recorder or the video input of another TM1000 through a CCIR 656 compatible byte parallel video interface. The VO can either supply or receive video clock and/or synchronizing signals from the external interface. Clock and timing signals can be precisely controlled through programmable registers. The VO assembles planar image data from SDRAM and converts it to a CCIR656 compatible digital video output stream. Programmable interrupts and double buffering allow the VO to generate continuous video output with the DSPCPU programming image pointer information for each field. The VO also provides programmable YUV overlay capability with alpha blending, allowing placement of an alpha blended overlay of arbitrary size and position within the output image.

The VO can also be used to emit raw data or send messages from one TM1000 to another. In the Data Streaming data mode, the VO can generate a continuous stream of byte data using internal or external clocking. Dual buffers facilitate continuous data streaming by allowing the DSPCPU to set up a buffer another is being emptied by the VO. Messages can be sent to one or more TM1000 Video In ports in the Message Passing mode. Start and end-of-message signals are provided in this mode to synchronize message passing to the other TM1000 message receivers.

The Video Out unit provides the following key functionality:

- Continuous digital video output of PAL or NTSC format data according to CCIR601.
- YUV 4:2:2 data output format using CCIR656 8 bits interface with embedded SAV and EAV synchronization codes and separate sync control signals compatible with DENC encoders at a nominal rate of 27 megabytes/second = 13.5 megapixels/second.
- YUV 4:2:2 data rate up to 80 MByte/sec = 40 megapixels/sec.
- Output is compatible with CCIR656-compliant digital VCRs and with Video In of TM1000. The VO can serve as a source of CCIR656 video data for test of TM1000 Video In and for sending CCIR656 video data to other TM1000 devices.
- Output can be generated from planar YUV 4:2:2 cosited, YUV 4:2:2 interspersed, or YUV 4:2:0 memory formats.
- In memory YUV data can be sent with optional horizontal upsampling by 2×.
- YUV graphics data can be overlayed/alpha blended with the image data for simultaneous display of pixel graphics and live video.
- High bandwidth (80 MByte/sec) output data channel in data-streaming and message-passing modes.
- Transmitter port for TM1000-to-TM1000 unidirectional message passing in message-passing mode.

The VO outputs digital video in YUV 4:2:2 co-sited format with 8-bit resolution multiplexed in CCIR656 format[1]. The VO can drive a CCIR656-compatible digital video encoder, or DENC (such as the Philips SAA7185), across an 8-bit wide interface. Digital video output data is sent at a programmable clock rate, typically 27 megabytes/second per the CCIR 656 specification. This corresponds to a pixel rate of 13.5 megapixels per second for YUV 4:2:2 coding. It can also drive other CCIR 656 compatible devices such as digital video tape recorders (VCRs) and the Video In of other TM1000 chips. For example, in Video In Diagnostic Mode, the VO of TM1000 supplies video data to the Video In of TM1000 in internal loopback mode for system diagnostic tests.

The VO normally supplies continuous video data to its outputs. The VO supplies video data from image data stored in YUV 4:2:2 co-sited format, YUV 4:2:2 interspersed format or YUV 4:2:0 format in tables in local SDRAM. The VO is programmed and started by the TM1000 DSPCPU. The VO issues an interrupt to the DSPCPU at the end of each field. The DSPCPU updates the VO image data pointers with pointers to the next field during the vertical blanking interval to maintain continuous video output. During video output, the VO supplies SAV and EAV sync codes and optionally supplies horizontal and frame timing signals. The VO can supply the timing for the pixel clock and for the horizontal and frame timing signals or can genlock to external timing signals such as supplied by a Philips SAA7185 DENC digital encoder or similar timing source.

7.2 INTERFACE

Table 7-1 lists the interface pins for the VO block. Figure 7-1, Figure 7-2, and Figure 7-3 illustrate typical

1. Refer to CCIR recommendation 656: Interfaces for digital component video signals in 525 line and 625 line television systems. Recommendation 656 is included in the Philips Desktop Video Data Handbook.

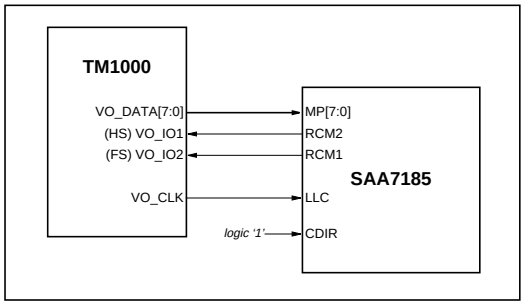


Figure 7-1. Video Out connected to a video encoder (DENC), external sync mode.

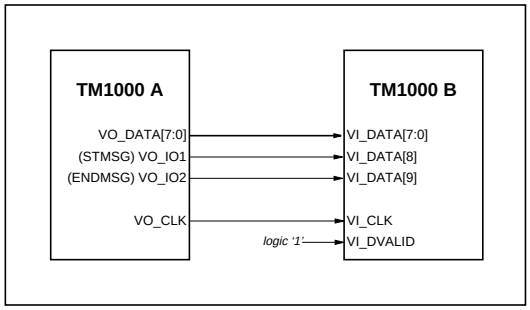


Figure 7-2. Video Out connected to Video In of a second TM1000.

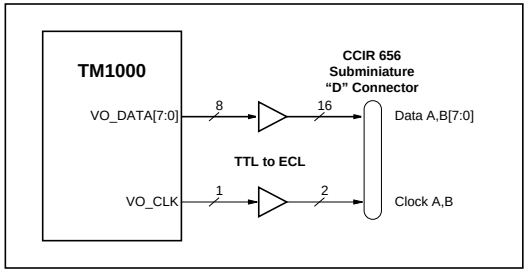


Figure 7-3. Video Out connected to a CCIR 656 video-output connector.

connections for commonly used external devices that interface to the VO. It is also possible to connect VO to a Gennum GS9022 Digital Video Serializer or similar part to generate serial D1 video.

7.3 BLOCK DIAGRAM

Figure 7-4 shows a block diagram of the Video Out block. It consists of a clock generator, a frame timing generator and an image or data generator. The image generator produces either a CCIR 656 digital video data stream with optional YUV overlay or a raw data or message-data stream. It also performs optional format conversions and optional 2:1 horizontal scaling.

Table 7-1. Video Out Interface Pins

Signal Name	Type	Description
VO_DATA[7:0]	OUT	CCIR656 style YUV 4:2:2 digital output data. Output on positive edge of VO_CLK, and (in external sync mode) synchronized on VO_IO1 and VO_IO2 sync signals from the DENC. Also general purpose high speed data output channel.
VO_IO1	I/O-5	This pin can function as HS (Horizontal Sync) input, HS output or as STMSG (Start Message) output. • If set as HS input, it can be set to respond to positive or negative edge transitions. If the Video Out operates in external sync mode and the selected transition occurs, the VO generates a sequence of a CCIR 656 EAV code, horizontal blanking, an SAV code and YUV 4:2:2 pixel data on VO_DATA. • In message passing mode, this pin acts as STMSG output. A high indicates that the current data presented on VO_DATA[7:0] is the start byte of a message.
VO_IO2	I/O-5	This pin can function as FS (Frame Sync) input, FS output or as ENDMSG output. • If set as FS input, it can be set to respond to positive or negative edge transitions. • If the Video Out operates in external sync mode and the selected transition occurs, the Video Out sends two fields of video data. • In message passing mode, this pin acts as ENDMSG output. A high indicates that the current data presented on VO_DATA[7:0] is the end byte of a message.
VO_CLK	I/O-5	• If configured as input (power up default): VO_CLK is received from external display clock master circuitry. • If configured as output, TM1000 emits a programmable clock frequency. The emitted frequency can be set between approx. 4MHz and 80 MHz with a resolution of 0.07 Hz. The clock generated is frequency accurate and has low jitter properties due to a combination of an on-chip DDS (Direct Digital Synthesizer) and VCO/PLL. • The Video Out unit emits VO_DATA on a positive edge of VO_CLK.

The frame timing generator provides programmable image timing including horizontal and vertical blanking, SAV and EAV code insertion, overlay start and end timing, and horizontal and frame timing pulses. It also supplies start-of-message and end-of-message timing in the message passing mode. The sync timing pulses can be generated by the frame timing unit, or the frame timing

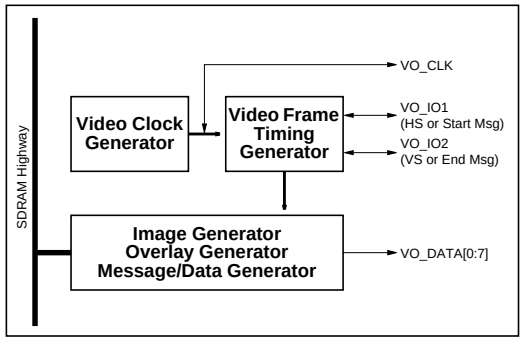


Figure 7-4. Video Out block diagram.

unit can be driven by externally supplied sync timing pulses, as determined by the SYNC_MASTER bit.

The video clock generator produces a programmable video clock. The video clock generator can supply the video clock for the frame timing generator and external devices, or it can be driven by an external clock signal.

7.4 CLOCK SYSTEM

Positive edges of VO_CLK drive all VO output events. A block diagram of the VO clock system is shown in Figure 7-5. The VO clock is either supplied externally or internally generated by the VO, as controlled by the CLKOUT bit in the VO_CTL register. When the CLKOUT bit is zero, the VO clock is supplied by an external source through the VO_CLK pin as an input. This is the default mode, entered at reset. When CLKOUT is a one, an internal clock generator supplies the VO clock and drives the VO_CLK pin as an output.

At the heart of the clock generator system is a *square wave DDS* (Direct Digital Synthesizer). The DDS can be programmed to emit frequencies from 8 MHz to 40 MHz with a resolution of 0.07 Hz. The output of the DDS is sent to a phase locked loop filter, which removes clock jitter from the DDS output signal. The PLL can also be used to divide or double the DDS frequency. The PLL needs to be enabled/programmed, as described in section 7.13. DDS programming is accomplished by setting

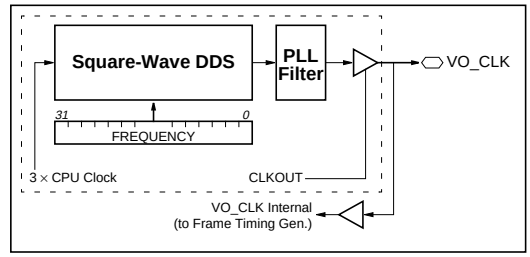


Figure 7-5. Video Out clock system.

the FREQUENCY field in the VO_CLOCK register according to the equation in Figure 7-6:

$$f_{DDS} = \frac{3 \times FREQUENCY \times f_{DSPCPUCLK}}{2^{32}}$$

Figure 7-6. DDS Oscillator Frequency

7.5 IMAGE TIMING

The VO emits a serial data stream used by a CCIR 656 device to generate a displayed image. Figure 7-7 shows an NTSC-compatible, 525-line interlaced image. The field and line numbers are shown for reference.

Interlaced images are generated by the display hardware by controlling the vertical retrace timing. A timing diagram of NTSC compatible interlaced frame timing illustrating the analog vertical retrace signal is shown in Figure 7-8 for reference. The vertical retrace signal for the second field begins in the middle of the horizontal line that ends the first field. This causes the first line of the second field to begin halfway across the display screen and the lines of the second field to be scanned between the lines of the first field (interlaced).

The analog timing to generate the interlaced signal is supplied by the display device. The CCIR 656 digital video signals generated by the VO use frame synchronization timing and do not generate any vertical retrace timing.

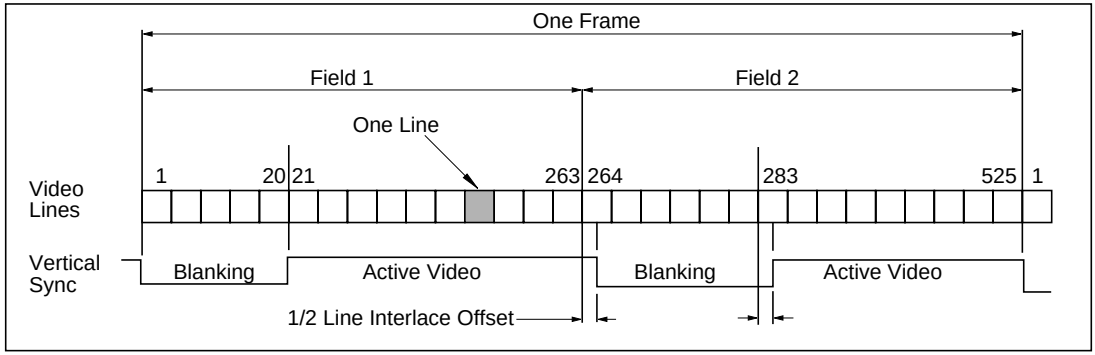


Figure 7-8. Interlaced timing–NTSC analog sync. signals.

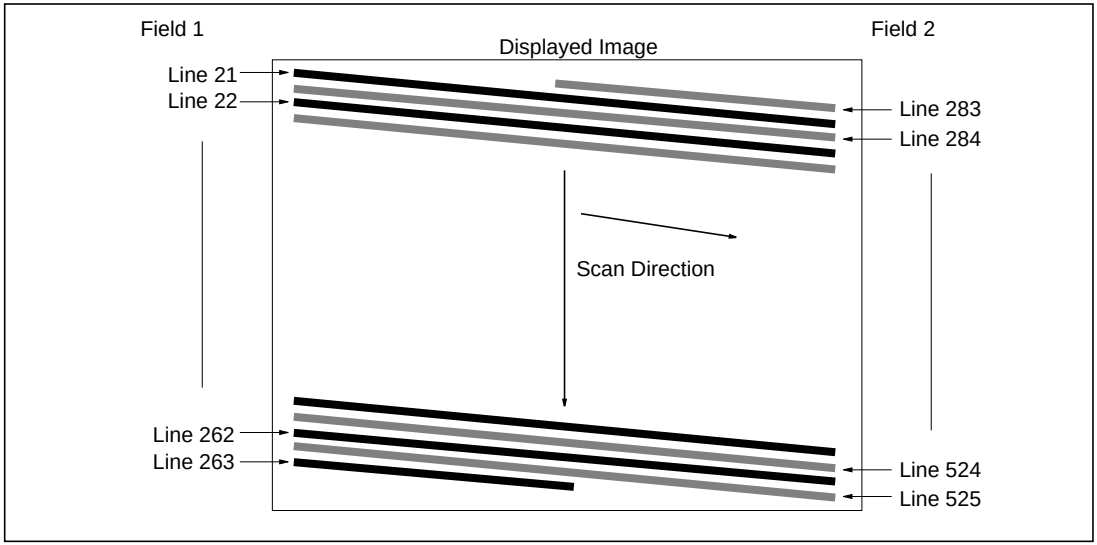


Figure 7-7. Interlaced display: 525-line, 60-Hz image.

7.5.1 CCIR 656 Pixel Timing

The VO generates pixels according to CCIR 656 timing in YUV 4:2:2 co-sited format and outputs these pixels as shown in Figure 7-9. Pixels are generated in groups of two, with four bytes per two pixels. Each pair of pixels has two luminance bytes (Y0, Y1) and one pair of chrominance bytes (U0, V0) arranged in the sequence shown. Pixels are generated at a nominal rate of 13.5 megapixels per second (27 MB/sec), and are clocked out on the positive edge of VO_CLK.

7.5.2 CCIR 656 Line Timing

The CCIR 656 line timing is shown in Figure 7-10. Each line begins with an EAV code, a blanking interval and an SAV code, followed by the line of active video. The EAV code indicates end of active video for the previous line, and the SAV code indicates start of active video for the current line.

7.5.3 SAV and EAV Codes

The EAV (End Active Video) and SAV (Start Active Video) codes are issued at the start of each video line. EAV and SAV codes have a fixed format: a three-byte preamble of FFh, 00h, 00h followed by the SAV or EAV code byte. The EAV and SAV code byte format is shown in Figure 7-11 for reference. The EAV and SAV codes define the start and end of the horizontal blanking interval, and they also indicate the current field number and the vertical blanking interval.

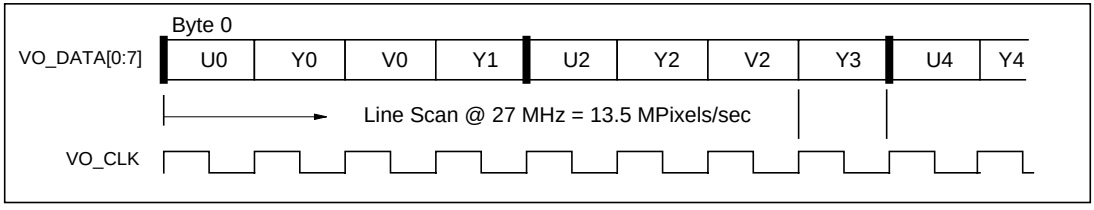


Figure 7-9. CCIR 656 pixel timing.

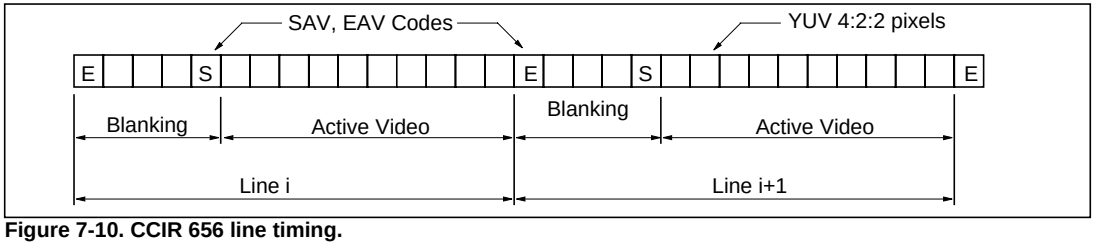


Figure 7-10. CCIR 656 line timing.

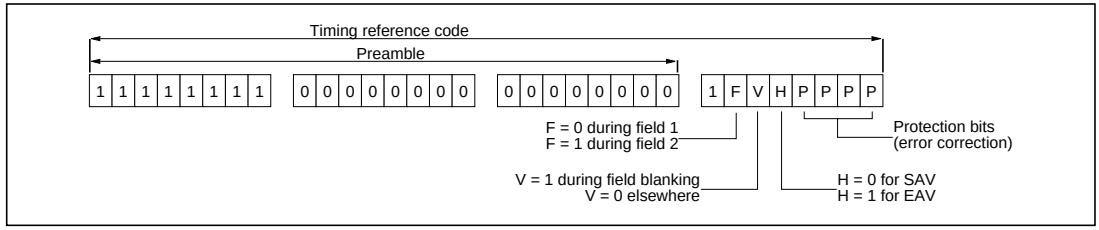


Figure 7-11. Format of SAV and EAV timing codes.

The SAV and EAV codes have a four-bit protection field to insure valid codes. The VO generates these protection bits as part of the SAV and EAV codes as defined by CCIR656. There are eight possible valid SAV and EAV codes. These eight codes with their correct protection bits are shown in Table 7-2. The VO generates SAV and EAV sync codes and inserts them into the video out data stream according to the CCIR656 specification under all conditions, whether it is generating or receiving horizontal and frame timing information.

Table 7-2. SAV and EAV Codes

Code	Binary Value	Field	Vertical Blanking
SAV	1000 0000	1	
EAV	1001 1101	1	
SAV	1010 1011	1	X
EAV	1011 0110	1	X
SAV	1100 0111	2	
EAV	1101 1010	2	
SAV	1110 1100	2	X
EAV	1111 0001	2	X

7.5.4 FFh and 00h Video Clamps

SAV and EAV codes are identified by a three-byte preamble of FFh, 00h and 00h. This combination must be avoided in the video data that the VO sends out to prevent accidental generation of an invalid sync code. The VO includes maximum and minimum value clamps on the video data to prevent this possibility. The VO automatically converts image data values of FFh to FEh and values of 00h to 01h. This clamping action provides protection at the cost of a small limit for extreme values of the video signal. overlap. These extreme values are not valid video signal values in CCIR 601 compatible image data.

7.5.5 CCIR 656 Frame Timing

The frame timing for CCIR 656 is shown in Table 7-3. CCIR 656 defines interlaced frame timing. Lines are numbered from 1 to 525 for 525-line, 60-Hz systems and from 1 to 625 for 625-line, 50-Hz systems. The Field and Vertical Blanking columns indicate whether the field and vertical blanking bits, respectively, are set in the SAV and EAV codes for the indicated lines. The 525 and 625 formats have similar timing but differ in their line number-

Table 7-3. CCIR 656 Frame TIming

Line Number		Field	V. Blank	Comments
525/60	625/50			
1–3	624–625	1	1	Vertical blanking for field 1, SAV/EAV code still indicates field 2
4–19	1–22	0	1	Vertical blanking for field 1, change SAV/ EAV code to field 1
20–263	23–310	0	0	Active video, field 1
264–265	311–312	0	1	Vertical blanking for field 2, SAV/EAV code still indicates field 1
266–282	313–335	1	1	Vertical blanking for field 2, change SAV/ EAV code to field 2
283–525	336–623	1	0	Active video, field 2

7.6 VIDEO OUT TIMING GENERATION

The VO generates timing for frames, active video areas within frames, images within the active video area, and overlays within the image area. The relationship between these four is shown in Figure 7-12. The frame includes the timing for both interlaced fields. The active image area begins after the horizontal and vertical blanking intervals and represents the pixels that are visible on the screen. The image area is the actual displayed image within the active video area. It can be slightly smaller than the active video area to avoid edge effects at the top, bottom and sides of the image. The overlay area is within the image area.

The VO uses two sets of counters to generate and control image timing: frame counters and image counters. The Frame Line Counter and Frame Pixel Counter control the overall timing for the frame and define the total number of pixels per line, lines per frame and interlace timing, including horizontal and vertical blanking intervals. Note that the Frame Line Counter has a starting value of one, not zero, and it counts from 1 to 525 or 625, consistent with CCIR 656 line numbering. The Image Line Counter and Image Pixel Counter define the visible image within the frame.

The geometry of active video is defined by the contents of several MMIO registers; see Figure 7-26. The FIELD 2 START value defines the start of field 2. Field 2 is active when the Field Line Counter contents equal or ex-

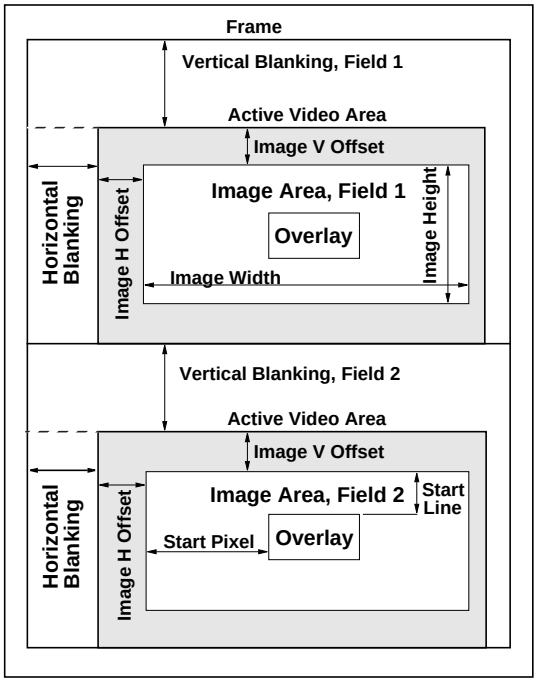


Figure 7-12. Frame, field, active video, image, and overlay definitions.

field of the frame and VIDEO PIXEL START value for each line of the frame. The active video area begins when the contents of the Frame Line Counter and Frame Pixel Counter exceed these values.

The CCIR 656 compliant 525/60 and 625/50 timing specifications define an overlap period where the field number in the SAV and EAV codes from field 1 persist into the vertical blanking interval for field 2, and the codes for field 2 persist into the vertical blanking interval for field 1. The F1 OLAP and F2 OLAP values define these overlap intervals. The overlap interval is two, three, or four lines long, depending on the field and whether 525/60 or 625/50 timing is used. During the overlap interval, the vertical blanking for the next field has begun; however, the field number flag in the SAV and EAV codes still shows the field number for the previous field. The field number is updated to the correct field value at the end of the overlap interval.

F1 OLAP defines the overlap from field 1 to field 2. This overlap occurs during the beginning of vertical blanking for field 2; The SAV and EAV codes continue to show field 1 during this overlap interval, and they change to field 2 at the end of the interval.

F2 OLAP defines the overlap from field 2 to field 1. This overlap occurs during the beginning of vertical blanking for field 1; The SAV and EAV codes continue to show field 2 during this overlap interval, and they change to field 1 at the end of the interval.

F1 OLAP and F2 OLAP are small positive values that indicate the number of lines of prior field overlap following the end of the Active Video Area in the current field. When the last line of the current Active Video Area has been read, a field overlap counter is loaded with the appropriate value of F1OLAP or F2OLAP. This counter is decremented as each line is output. As long as the counter value is positive, the SAV and EAV codes use the field value of the previous field.

The frame and image counters have different start and stop points. The frame counters begin in the vertical blanking interval of the first field and the horizontal blanking interval of the first line. They stop counting when they reach the height and width values of the frame. When the VO generates the frame timing, the frame counters are reset to their start values when they reach their stop values; when the VO receives frame timing signals, the frame counters continue counting until reset by the external signals.

The image area is defined by the IMAGE VOFF and IMAGE HOFF values. These values are added to the Video Line Start and Video Pixel Start values to define the starting line and pixel, respectively of the image area. The image area is active when the contents of the Frame Line Counter and Frame Pixel Counter equal or exceed these values.

The Image Line Counter and Image Pixel Counter start counting at the first active pixel in the image area and the first active line in the image area, respectively. The image counters start at zero and stop counting when they reach their image height and width values. The image counters are reset by frame counter values indicating the start of the image pixel in a line and the start of the image line in a field.

The image counters define the active image area of the frame, the area of interest for image processing. This allows the overlay start address to be defined relative to the active image area, for example. When the VO is not sending out active pixels from the image area, it sends out blanking codes. These blanking codes are (0x80, 0x10, 0x80, 0x10) for each two pixel group in YUV 4:2:2 image data format, as defined by CCIR 656 and shown in Figure 7-9.

7.6.1 Horizontal and Frame Timing Signals

The VO can supply or receive horizontal and frame timing signals. When the SYNC_MASTER bit is set, the VO generates the horizontal and frame timing for the external video device. When the SYNC_MASTER bit is cleared, Video Out operates in external sync mode and an external device, such as a DENC, is responsible for providing horizontal and frame sync.

If SYNC_MASTER is set, VO_IO1 acts as output and generates a horizontal timing signal, and VO_IO2 igenerates a frame timing signals. Figure 7-13 shows how the generated signals relate to the VO line and field timing. The horizontal timing signal corresponds to the horizontal-blanking interval, and the frame timing signal corresponds to the field-2 active interval. The horizontal timing signal is active low from the EAV code at the start

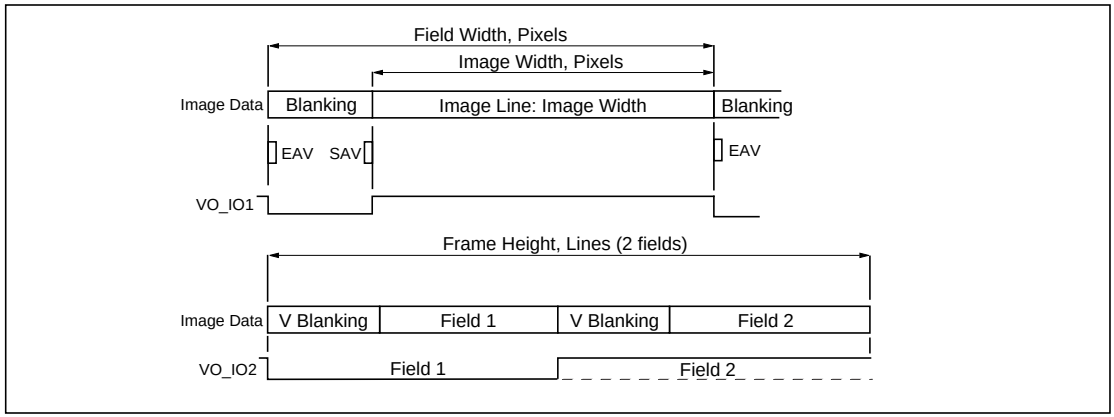


Figure 7-13. Horizontal and vertical timing signals, Video Out as output.

of the line to the SAV code at the start of active video for the line. The frame timing signal is active high from the EAV code that begins the first line of vertical blanking for field 2 to the EAV code that begins the first line of blanking for field 1.

If SYNC_MASTER is clear, VO expects horizontal and frame timing signals on the VO_IO1 and VO_IO2 pins. The active edge of both signals can be programmed using VO_IO1_POS and VO_IO2_POS. The selected polarity transition of the horizontal timing signal on VO_IO1 causes the VO to preset the Frame Pixel Counter to zero. The selected transition of the frame timing signal on VO_IO2 causes the Frame Line Counter to be set to the FRAME PRESET value. This is typically a small value to compensate for the delay in the frame timing source. This is shown in Figure 7-14.

7.7 DATA TRANSFER TIMING

In the *data streaming* and *message passing* modes, the VO supplies a stream of 8-bit, unsigned data at up to 80 MHz data rate. No data selection or data interpretation is done, and data is transferred at one byte per VO_CLK. Data is clocked out on the positive edge of VO_CLK.

The message passing mode issues signals on VO_IO1 and VO_IO2 to indicate the start and end of the message.The timing for these signals is shown in Figure 7-15.

7.8 IMAGE DATA FORMATS

7.8.1 YUV Image Formats

The VO accepts memory-resident video data in three formats: YUV 4:2:2 co-sited, YUV 4:2:2 interspersed and YUV 4:2:0. These formats are shown in Figure 7-16 through Figure 7-18.

7.8.2 Planar Storage of YUV Image Data in Memory

YUV image data is stored in memory with one table for each of the Y, U and V components. This is called planar format. This is shown in Figure 7-19 for YUV 4:2:2 image

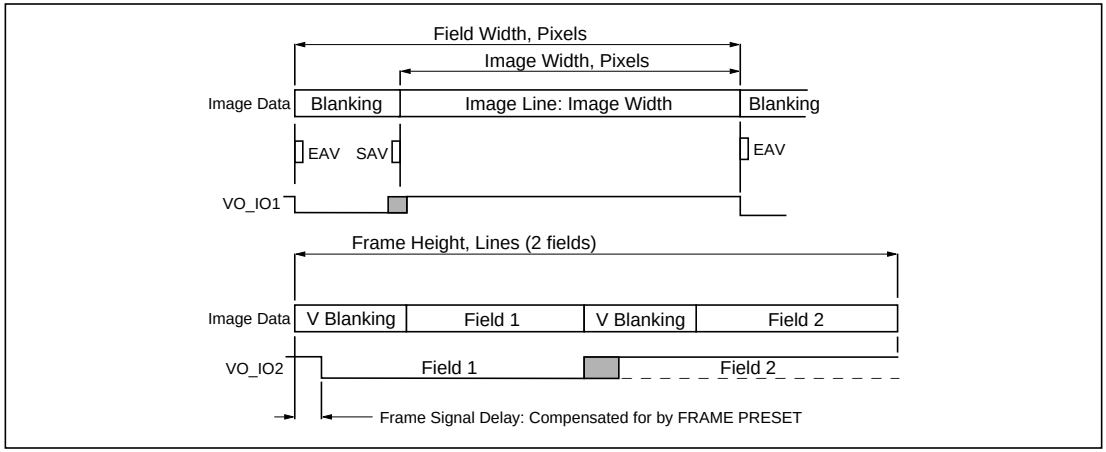


Figure 7-14. Horizontal and vertical timing signals, Video Out as input.

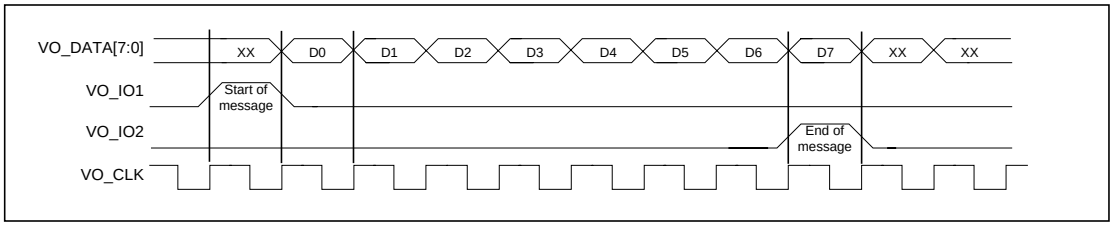


Figure 7-15. Video Out message-passing START and END events.

data. The VO merges bytes from each of the three tables to generate the CCIR 656 compatible output data. The U and V tables have the same number of lines but half the number of pixels per line as the Y table. The transfer is the same for YUV 4:2:0 format except the U and V tables will be 1/4 the size of the Y table. The U and V tables have the half the number of lines and half the number of pixels per line as the Y table.

7.8.3 YUV Overlay Formats

YUV overlay data is stored in a single table in SDRAM. Overlay images are stored in YUV 4:2:2+alpha formats. Figure 7-20 shows this format. The YUV overlay is always in the image output format. The VO does not upscale the overlay image. If the VO is upscaling the output image by 2×, the YUV overlay is provided in upscaled format.

The VO provides alpha blending for the overlay image. No chroma keying is supported.

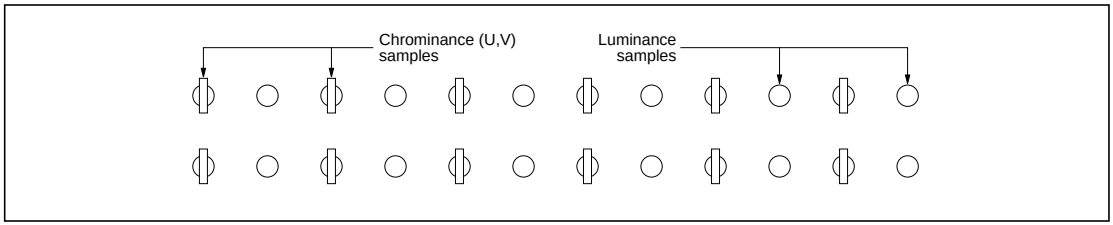


Figure 7-16. YUV 4:2:2 co-sited format.

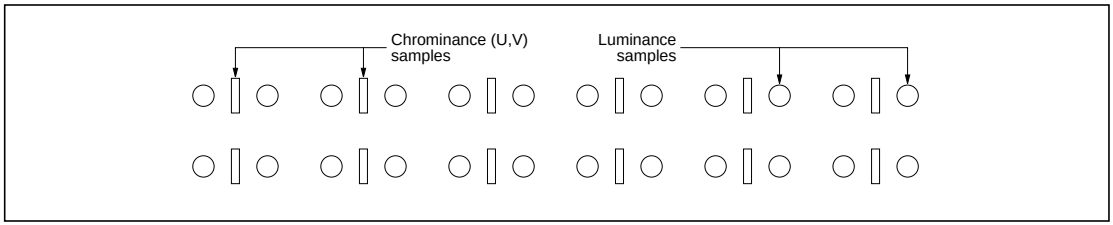


Figure 7-17. YUV 4:2:2 interspersed format.

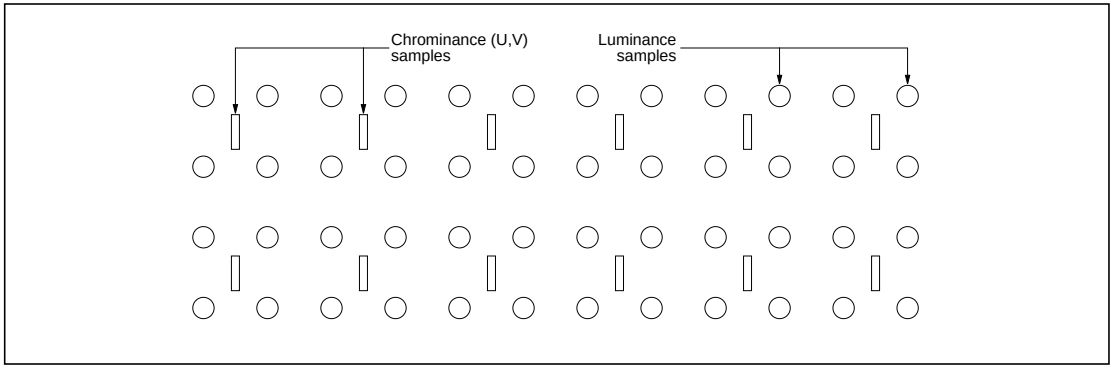


Figure 7-18. YUV 4:2:0 format.

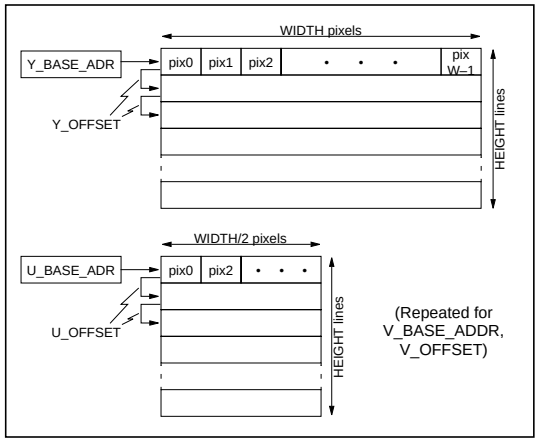


Figure 7-19. Image storage in planar memory format for YUV 4:2:2.

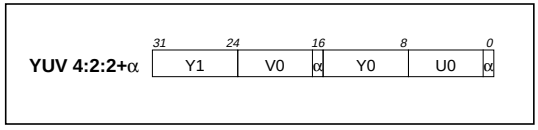


Figure 7-20. YUV 4:2:2+alpha overlay format.

Alpha blending combines the overlay image with the primary image according to an alpha value provided with the overlay pixel. In the YUV 4:2:2+α format, each pixel has a single α-bit supplied as the least significant bit of the U and V pixels for the Y0 and Y1 pixels, respectively. When the α-bit is zero, the ALPHA Zero register supplies the α value. When the α-bit is one, the ALPHA ONE register supplies the α value. These registers are addressed as by ALPHA ZERO field of the VO_OLSTART and ALPHA ONE field of the VO_OLHW register. Alpha blending is provided according to Table 7-4. Although 7 bits of blending resolution are provided for in the architecture, the actual number of bits implemented depends on the TM1000 version. Any TM1000 version implements at least 25% step resolution.

In the YUV 4:2:2 format, only one set of U and V values is supplied for the two Y pixels, Y0 and Y1. The alpha bit in U0 determines the alpha value for U, Y0 and V. The alpha blend bit in V0 only sets the alpha value for Y1 and does not affect the U or V values.

7.9 ALGORITHMS

7.9.1 YUV 4:2:2 Interspersed to YUV 4:2:2 Co-sited Conversion

The VO can accept data from SDRAM in either YUV 4:2:2 co-sited, YUV 4:2:2 interspersed or YUV 4:2:0 interspersed formats. If the input data is in YUV 4:2:2 or YUV 4:2:0 interspersed format, interspersed-to-co-sited conversion is required for co-sited output. The VO uses a four-tap, (–1, 5, 13, –1)/16 filter to perform this conversion on the U and V chroma data. An example of interspersed to co-sited conversion is shown in Figure 7-21.

7.9.2 YUV 4:2:0 to YUV 4:2:2 Co-sited Conversion

YUV 4:2:0 to YUV 4:2:2 conversion is a variation of YUV 4:2:2 interspersed-to-co-sited conversion. The YUV 4:2:0 format has the U and V pixels positioned between lines as well as between pixels within each line. It also has half the number of U and V pixels compared to YUV 4:2:2 formats. The VO converts YUV4:2:0 to YUV 4:2:2 co-sited by using the U and V chrominance pixels values for both surrounding lines and converting the resulting U and V pixels from interspersed to co-sited format. This is shown in Figure 7-22. If true vertical resampling of U and V is desired, the TM1000 image co-processor can be invoked on U and V to convert from YUV 4:2:0 to YUV 4:2:2 interspersed.

7.9.3 YUV-2X Upscaling

In the YUV-2X and YUV 2X2V modes, the VO performs 2× upscaling of the YUV data from SDRAM. The width of the result image (IMAGE WIDTH) should be an even number. Upscaling is performed by four-tap filtering. The Y, luminance data is upscaled using a (–3,19,19,–3)/32 filter to generate the missing output pixels. The output pixels that are at the same location as the input pixels use the corresponding input pixel values. This is shown in Figure 7-23.

The U and V chrominance values are generated in the same way as the Y luminance signal for 2× upscaling, assuming that both the input and output use YUV 4:2:2 co-sited chrominance coding. The U and V output pixels that are at the same location as the U and V input pixels use the corresponding input pixel values. The U and V output pixels that are between the U and V input pixels

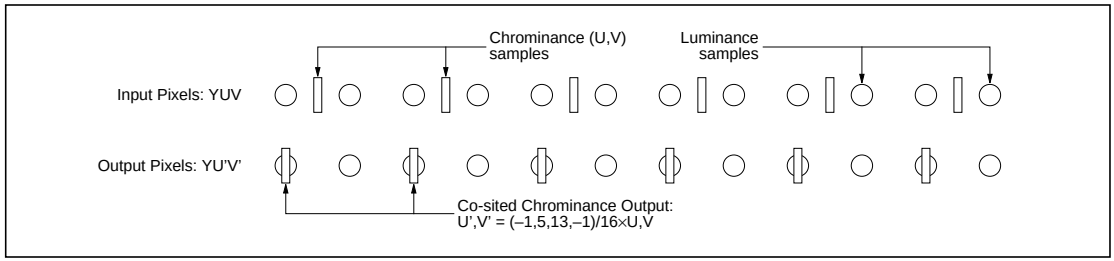


Figure 7-21. YUV interspersed to co-sited conversion.

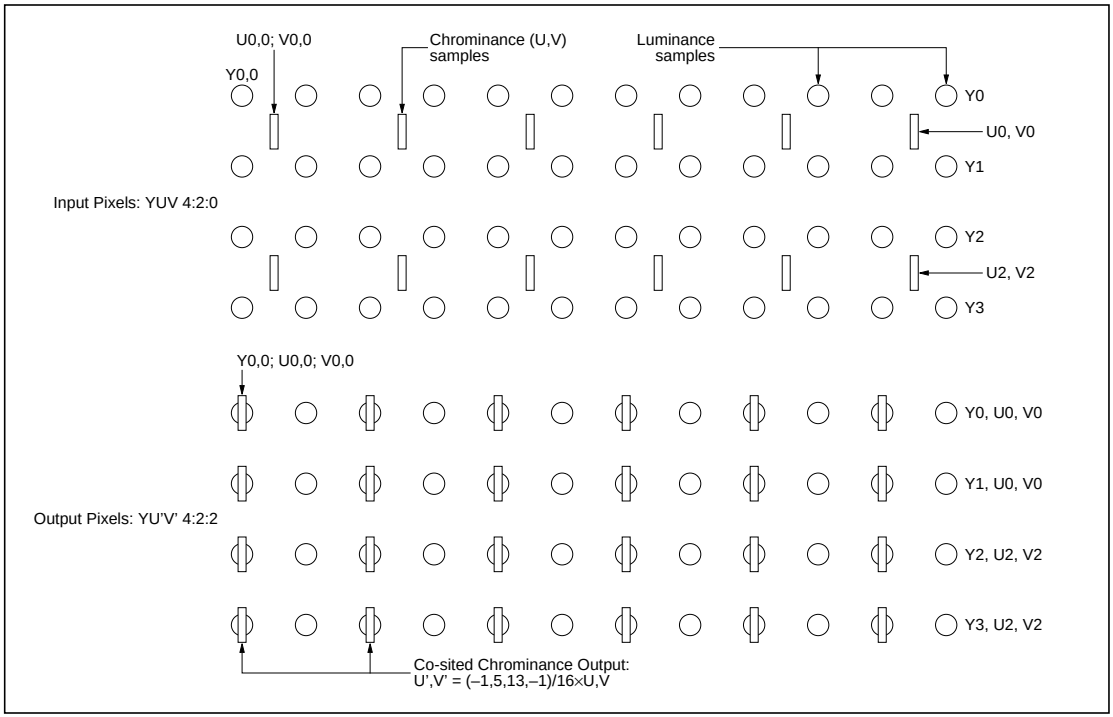


Figure 7-22. YUV 4:2:0 to YUV 4:2:2 co-sited conversion.

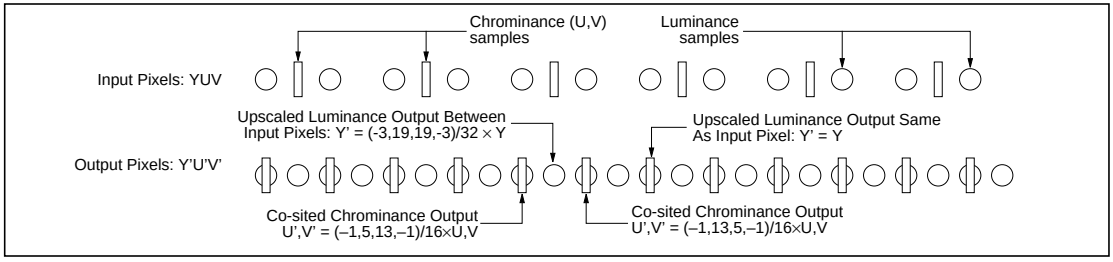


Figure 7-24. 2×-upscaling of U and V with interspersed to co-sited conversion.

are generated using the (–3,19,19,–3)/32 filter. This is shown in Figure 7-23.

If the input chroma is interspersed, a (–1,13,5,–1)/16 filter is used to generate the U and V output pixels that are displaced by half a Y pixel from the U and V input pixels, and a (–1,5,13,–1)/16 filter is used to generate the additional upscaled U and V output pixels that are displaced by 1.5 pixels from the U and V input pixels. This is shown in Figure 7-24.

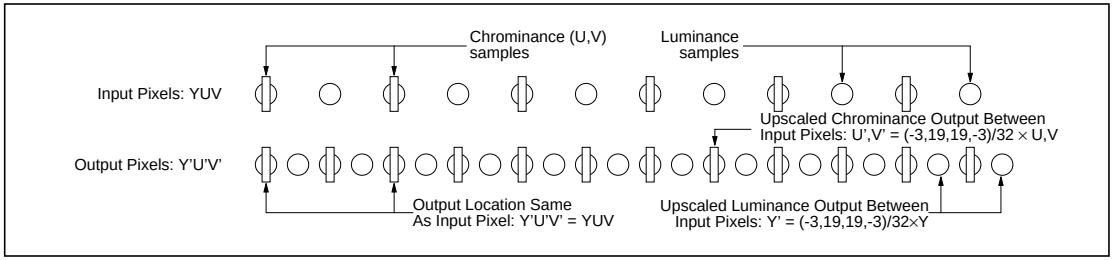


Figure 7-23. 2×-upscaling of Y pixels.

Table 7-4. Alpha Blending Codes

Alpha Code	Alpha Value	Image	Overlay
00h	0	100%	0%
20h	32	75%	25%
40h	64	50%	50%
60h	96	25%	75%
80h–FFh	128–255	0%	100%

7.9.4 Pixel Mirroring for Four-tap filters

The VO uses a four-tap filter for upscaling and for converting from interspersed to co-sited format. One extra pixel is needed at the beginning and two at the end of each line that is processed by this filter. These pixels are supplied automatically by mirroring the first and last pixels of each line. For example:

- Output pixel 1 uses input pixel 1 to generate its value. (same location, no filtering).
- Output pixel 2 uses pixels 1,1, 2 and 3 to generate its value.
- Output pixel 3 uses pixel 2 to generate its value.
- Output pixel 4 pixel uses pixels 1, 2, 3 and 4, etc.
- .....
- Output pixel 2N–2 uses pixels N–2, N–1, N, and N–1 to generate its value.
- Output pixel 2N–1 uses pixel N to generate its value.
- Output pixel 2N uses pixels N–1, N, N, and N–1 to generate its value.

Figure 7-25 shows an example of six pixels upscaled to 12 pixels.

7.10 OPERATING MODES

The Video Out unit operates in one of several image or data transfer modes as determined by the contents of the MODE field in the VO_CTL register. These modes are shown in Table 7-5. The different image transfer modes define input data format and whether horizontal upscaling is performed.

Table 7-5. Video Out Operating Modes

Mode	Function	Explanation
0000	YUV 4:2:2C-1×	YUV 4:2:2 co-sited input, no scaling
0001	YUV 4:2:2I-1×	YUV 4:2:2 interspersed input, no scaling
0010	YUV 4:2:0-1×	YUV 4:2:0 input, no scaling
0011	Reserved	
0100	YUV 4:2:2C-2×	YUV 4:2:2 co-sited input, horizontal 2× upscaling
0101	YUV 4:2:2I-2×	YUV 4:2:2 interspersed input, horizontal 2× upscaling
0110	YUV 4:2:0-2×	YUV 4:2:0 input, horizontal 2× upscaling
0111	Reserved	
1000	Data Streaming	Continuous raw data flow
1001	Message Passing	VO to VI message passing: data streaming with SOM and EOM
1010 thru 1111	Reserved	

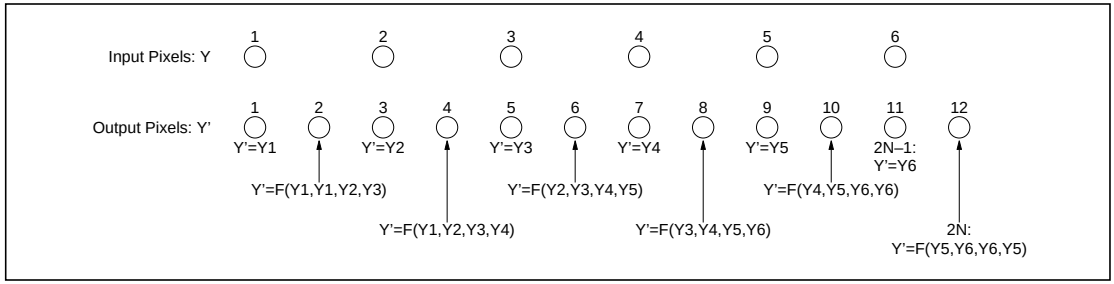


Figure 7-25. Mirroring pixels in 2× upscaling.

7.11 CONTROLS: MMIO REGISTERS

The MMIO Control Registers are shown in Figure 7-26. The register fields are described in Table 7-6, Table 7-7 and Table 7-8.

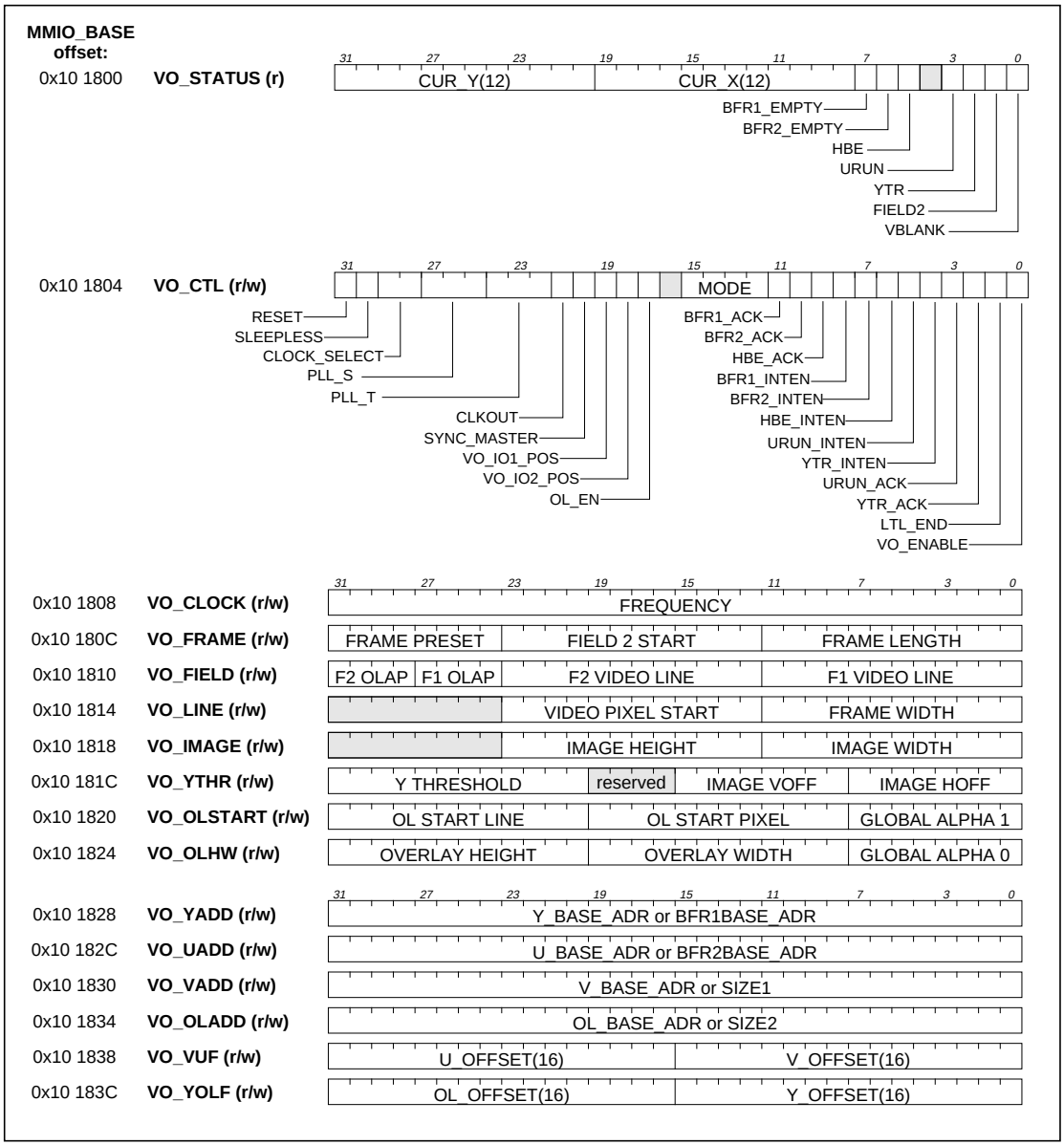


Figure 7-26. Video Out MMIO registers.

7.11.1 Status Register

The VO_STATUS register is a read-only register that shows the current status of the VO. Its fields are shown in Table 7-6.

Table 7-6. Status Register Fields

Field	Description
CUR_Y	Current Y: image line index of the current line being output by VO. CUR_Y reflects current state of the Image Line Counter. CUR_X & CUR_Y form a single 24-bit output data byte counter (CUR_X=counter LSBs) when VO in data-streaming or message-passing mode. This counter reflects the status of the SIZE counter for the currently active buffer. The two LSBs of this counter are not valid for reading during transfers; only the upper 22 bits (the word count) are valid.
CUR_X	Current X: image pixel index of the most-recently output pixel. CUR_X reflects the current state of the Image Pixel Counter.
BFR1_EMPTY BFR2_EMPTY	• Buffers 1 & 2 Empty: these bits are valid in image-transfer, data-streaming & message-passing modes. • In image-transfer modes, only buffer 1 is used. BFR1_EMPTY indicates that the last byte of a field has been transferred. It is actually raised at the completion of the transmission of the Overlap area of the field, as per Figure 7-27. At this point, software should assign a new field of imagery to Y,U,V_BASE_ADR and perform a BFR1_ACK. If BFR1_EMPTY is not cleared by BFR1_ACK before the start of emission of the active video area of the next field, the VO sets the URUN bit. • In data-streaming and message-passing modes, BFR1_EMPTY & BFR2_EMPTY indicate that the last byte in their corresponding buffer has been transferred. When BFR1_EMPTY or BFR2_EMPTY is set, transfer stops from the corresponding buffer. These bits cause an interrupt if their interrupts enables are set, and one interrupt per buffer is signaled. (Only buffer 1 is used in message-passing mode.)
HBE	Highway Bandwidth Error: HBE is set when the SDRAM highway fails to respond in time to a highway read request and data was not ready in time to be set on VO data lines. HBE can be set in both image- and data-transfer modes. HBE indicates insufficient bandwidth was requested from the highway arbiter.
YTR	End Of Field: in image transfer modes, YTR indicates the Image Line Counter value is equal to the Y THRESHOLD value in VO_YTHR. The Y THRESHOLD value can be set to provide an interrupt on any line in the valid image area.
URUN	• Underrun/End Of Transfer: in image- and data-streaming modes, this bit indicates that the CPU did not perform an acknowledge to indicate updated address pointers for the next field or buffer in time for continuous image or data transfer. URUN causes an interrupt if corresponding enable set. • In image-transfer modes, URUN indicates the SAV code marking beginning of active video has been generated without BFR1_ACK resetting BFR1_EMPTY. URUN indicates the CPU did not update the address pointers before the next field transfer had to begin; in this case, image transfer continues with previous address pointers. • In data-streaming mode, URUN indicates the last byte in active buffer was transferred, and no BFR1_ACK or BFR2_ACK occurred to enable next buffer. In this case, image transfer continues with previous address pointers.
FIELD2	• Field 2/Bfr 2 Active: in data streaming modes, zero when buffer 1 is active; one when buffer 2 is active. • In image-transfer modes, FIELD2 indicates that VO is actively sending out a video image for field 2, as defined by Figure 7-27.
VBLANK	Vertical Blanking: indicates VO is in a vertical-blanking interval. VBLANK active only in image-transfer modes.

7.11.2 Control Register

The VO_CTL register sets the operating mode, interrupt enables and clears interrupt flags and initiates VO operations. Its fields are shown in Table 7-7.

Table 7-7. VO_CTL Register Fields

Field	Description
RESET	Software reset of VO. The recommended software reset procedure is to write the desired VO_CTL state, with a '1' bit in the RESET bit position, followed by writing the desired VO_CTL state word. Enabling the newly selected mode by VO_ENABLE should be done last, as a separate transaction. A hardware RESET clears CLKOUT & SYNC_MASTER bits and put VO_CLK, VO_IO1, & VO_IO2 in input state. This results in a VO_CTL value of 32400000h. A software RESET results in a state as specified by the VO_CTL word value written during the above procedure.
SLEEPLESS	Prevents power-down of the VO when TM1000 power-down is active.
CLOCK_SELECT	00 - select PLL VCO output as VO_CLK source. This is the normal mode of operation. 01 - select PLL feedback loop divider output as VO_CLK source 10 - select PLL input divider output as VO_CLK source 11 - (hardware RESET default) select DDS output directly as VO_CLK source, bypass PLL altogether
PLL_S	This field sets the PLL input divider division ratio. A value of *k* selects division by *k+1*. The hardware RESET default for the field value is 1, causing division by 2.
PLL_T	This field sets the PLL feedback loop divider division ratio. A value of *k* selects division by *k+1*. The hardware RESET default for the field value is 1, causing division by 2.
CLKOUT	• When active, CLKOUT enables VO clock generator and makes VO_CLK an output. • When inactive, VO_CLK is input, and VO clock is provided by the external device.
SYNC_MASTER	• When active, VO_IO1 and VO_IO2 are outputs. In image-transfer modes, the VO generates horizontal respectively frame timing signals on VO_IO1 & VO_IO2. In message passing mode, this bit should always be set so that VO_IO1 and VO_IO2 generate START respectively END message signals. • When inactive, VO_IO1 and VO_IO2 are inputs. This is the RESET default. In image-transfer modes VO_IO1 serves as horizontal time reference and VO_IO2 serves as frame time reference. The active edge is selected by VO_IO1_POS resp. VO_IO2_POS.
VO_IO1_POS VO_IO2_POS	• Determines input polarity on VO_IO1 & VO_IO2. • When zero, the corresponding input triggers on the negative (high-to-low) transition of the input signal. • When one, the input triggers on the positive (low-to-high) transition.
OL_EN	Overlay Enable: enables the YUV overlay function in image transfer modes.
MODE	Defines the video output mode, as listed in Table 7-5 on page 7-11.
BRF1_ACK BFR2_ACK	Buffer-1 & buffer-2 acknowledge: when active in data-transfer modes, writing a one to BFR1_ACK clear BFR1_EMPTY and enables buffer 1 for transfer until BFR1_EMPTY is set. Writing a zero to BFR1_ACK has no effect. BRF2_ACK operates similarly for buffer 2. Writing a one to VO_ENABLE in the data-streaming mode is the same as writing a one to both BFR1_ACK & BFR2_ACK and enables both buffers 1 & 2 for transfer. Writing a one to VO_ENABLE in message-passing mode is the same as writing a one to BFR1_ACK and enables buffer 1 for transfer. BFR2_ACK cannot be set in message-passing mode, since only buffer 1 is used.
HBE_ACK URUN_ACK	Clear HBE and URUN flags and reset their corresponding interrupt conditions.
YTR_ACK	Clears the YTR flag and resets its interrupt condition. YTR signals the CPU to set new pointers for the next field. If YTR_ACK is not received by the time the active image area for the next field starts, the URUN flag is set. Data transfer continues with the old pointer values.
BFR1_INTEN BFR2_INTEN HBE_INTEN URUN_INTEN YTR_INTEN	Enable corresponding interrupts when the BFR1_EMPTY, BFR2_EMPTY, HBE, URUN (underrun/end of transfer), and YTR (end of field/buffer) flags are set, respectively.
LTL_END	Little-endian: specifies that data in SDRAM is stored in little-endian format. This only affects the overlay packed image format interpretation in the image transfer modes. Refer to Appendix C, "Endian-ness," for details on Byte Ordering.
VO_ENABLE	Enables the VO to send image data or message data to its output. Setting VO_ENABLE in image-transfer modes starts the VO sending image data beginning with the first pixel in the image. Setting VO_ENABLE in data-streaming and message-passing modes starts the VO sending data beginning with the first byte in buffer 1. In image-transfer and data-streaming modes, VO_ENABLE remains set until cleared by the CPU. In message-passing mode, VO_ENABLE is cleared with BFR1_EMPTY is set indicating the end of message transfer. De-asserting VO_ENABLE in image-transfer modes causes SDRAM reads to stop, but sync framing and BFR1_EMPTY generation/interrupts remain fully operational. Transmitted active image data is undefined. To fully halt Video Out, a software RESET is required.

7.11.3 Video Out Registers

The remaining VO registers and their fields are shown in Table 7-8.

Table 7-8. Video Out Register FIelds

Register	Field	Description
VO_CLOCK	FREQUENCY	VO_CLK frequency. See the equation in Figure 7-6.
VO_FRAME	FRAME LENGTH	Total number of lines per frame, the ending value of Frame Line Counter. Typically set to 525 or 625. Note frame counter counts from 1 to 525 or 625, consistent with CCIR 656 line numbering.
	FIELD 2 START	Start line number in Frame Line Counter where second field of frame begins. If FIELD 2 START is zero, no field 2 is generated, and non-interlaced timing results.
	FRAME PRESET	Value loaded into Frame Line Counter when frame timing edge is received on VO_IO2. This value compensates for delay in arrival of frame timing pulse.
VO_FIELD	F1 VIDEO LINE	Line number in the Frame Line Counter of first active video line of field 1 of the frame.
	F2 VIDEO LINE	Line number in the Frame Line Counter of first active video line of field 2 of the frame.
	F1 OLAP	Overlap of the SAV and EAV codes from field 1 to field 2. Overlap is defined as the delay in lines from start of blanking for field 2 until SAV and EAV codes for field 2 are emitted. Typical values are +3 for 525/60 and +2 for 626/50.
	F2 OLAP	Overlap in lines of the SAV and EAV code from field 2 to field 1. Overlap is defined as the delay in lines from start of blanking for field 1 until the SAV and EAV codes for field 1 are emitted. Typical values are +3 for 525/60 and –2 for 625/50. The negative value means field 1 blanking actually starts two lines before end of field 2 of previous frame. This overlap is described in Table 7-3 on page 7-5, and illustrated in Figure 7-27.
VO_LINE	FRAME WIDTH	Total line length in pixels including blanking. This is also the ending value for the Frame Pixel Counter. Lines always begin with horizontal blanking interval, and image starts after blanking interval and runs to end of the line.
	VIDEO PIXEL START	Pixel number in Frame Pixel Counter of starting pixel of active video area within the line.
VO_IMAGE	IMAGE HEIGHT	Image line height in lines.
	IMAGE WIDTH	Image line width in pixels. Must be even for upscaling by 2x.
VO_YTHR	Y THRESHOLD	Threshold image line number in the Image Line Counter for the YTR interrupt.
	IMAGE VOFF	Image vertical offset in lines from the top of active video window.
	IMAGE HOFF	Image horizontal offset in pixels from the start of active video window.
VO_OLSTART	OL START LINE	Starting image line of YUV overlay within the image. Zero indicates overlay starts at same pixel as the image.
	OL START PIXEL	Starting image pixel of the YUV overlay within the image. Zero indicates overlay starts at same pixel as the image.
	ALPHA ONE	Alpha blend value used for YUV 4:2:2+alpha format overlays when alpha bit = 1.
VO_OLHW	OVERLAY HEIGHT	Height of YUV overlay image in lines. The height of the overlay should be chosen such that it does not extend beyond the image area.
	OVERLAY WIDTH	Width of YUV overlay image in pixels.
	ALPHA ZERO	Alpha blend value used for YUV 4:2:2+alpha format overlays when alpha bit = 0.
VO_YADD	Y_BASE_ADR BFR1BASE_ADR	• In image-transfer modes, Y-component starting byte address. • In data-transfer mode, buffer 1 starting byte address.
VO_UADD	U_BASE_ADR BFR2BASE_ADR	• In image-transfer modes, U-component starting byte address. • In data-transfer mode, buffer 2 starting byte address. • Not used in message-passing mode.
VO_VADD	V_BASE_ADR SIZE1	• In image-transfer modes, V-component starting byte address. • In data-transfer mode, buffer 1 length in bytes.
VO_OLADD	OL_BASE SIZE2	• In image-transfer modes, overlay-image starting byte address. • In data-transfer mode, buffer 2 length in bytes. • Not used in message-passing mode.
VO_YUF	U_OFFSET	Offset in bytes from start of one line to start of next line.
	V_OFFSET	Offset in bytes from start of one line to start of next line.
VO_YOLF	Y_OFFSET	Offset in bytes from start of one line to start of next line.
	OL_OFFSET	Offset in bytes from start of one line to start of next line.

7.11.4 Frame and Field Timing Control

The frame timing for 525/60 and 625/50 cases is shown pictorially in Figure 7-27 for reference. CCIR 656 line definitions are used.

7.11.5 Timing Register Default Values

The default values for the various fields of the timing registers are shown in Table 7-9 for 525/60 and 625/50 timing cases. The FREQUENCY field value shown is for 27.0 MHz assuming a DSPCPU clock of 100.0 MHz.

7.12 VIDEO OUT OPERATION

The VO operates in either image transfer or data transfer modes. The DSPCPU starts the VO by setting the Mode field to the appropriate transfer mode, setting the appropriate addresses, address offsets, image timing registers and the associated control bits in the Control register and setting the VO Enable bit. The VO transfers the image or message as commanded. In the image-transfer and data-streaming modes, the VO runs continuously. In the message-passing mode, the VO runs only until the message has been transferred.

The VO unit is reset by the TM1000 hardware RESET, or by a software VO reset, as described in Table 7-7, RESET bit.

The VO_CLK is normally set as output to drive the data transfer for all modes at a programmable rate. The VO_CLK signal can be an input or output, as controlled by the CLKOUT bit in the VO_CTL register. When CLKOUT is set, VO_CLK is an output, and its frequency is set by the VO_CLOCK register value. When CLKOUT is a

Table 7-9. Timing Register Recommended Values

Register	Field	525/60 Value	625/50 Value
VO_CLOCK	FREQUENCY	170A3D70h	170A3D70h
VO_FRAME	FRAME-LENGTH	525	625
	FIELD 2 START	264	311
	FRAME PRE-SET	1	1
VO_FIELD	F1 VIDEO LINE	20	23
	F2 VIDEO LINE	283	336
	F1 OLAP	2	2
	F2 OLAP	3	-2 (0xE)
VO_LINE	FRAME WIDTH	858	864
	VIDEO PIXEL START	138	144
VO_IMAGE	IMAGE HEIGHT	243	288
	IMAGE WIDTH	720	720 (704 visible)

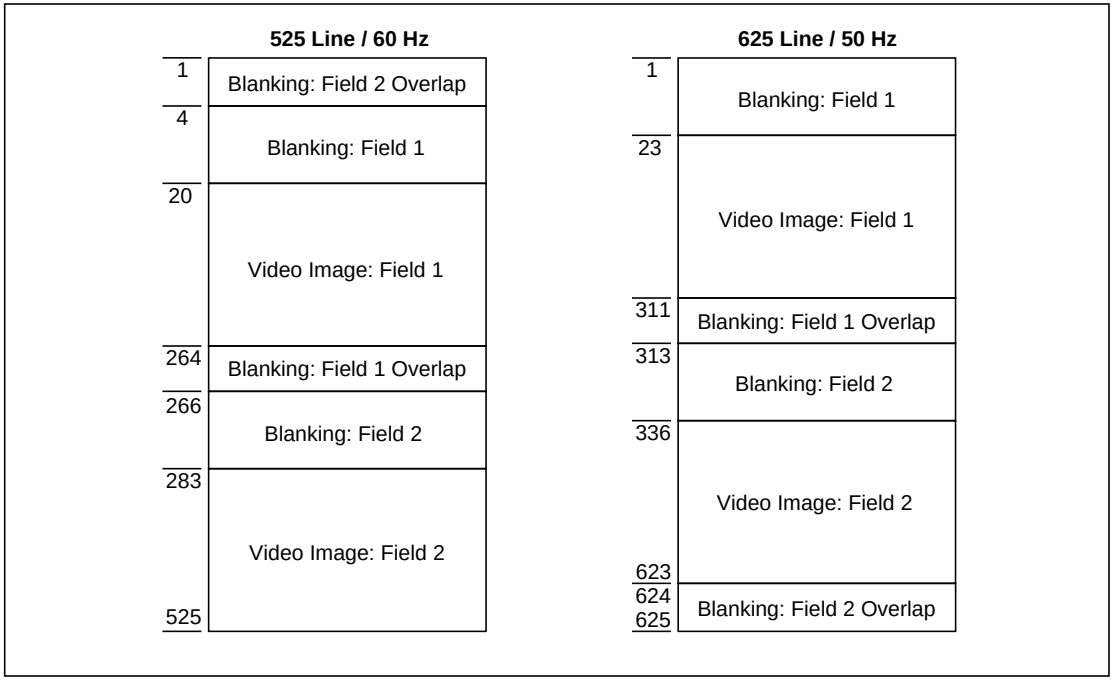


Figure 7-27. Video Out frame timing.

zero, VO_CLK is an input and the VO generates data at the clock rate of the sender.

In image-transfer modes, the VO receives or generates horizontal and frame synchronization signals on the VO_IO1 and VO_IO2 lines, as described in Section 7.6.1, "Horizontal and Frame Timing Signals."

7.12.1 Image Transfer Modes

In the image-transfer modes, the VO transfers an image from SDRAM to the VO port. The Mode field in the VO_CTL register defines the image input data format and whether the VO is to perform horizontal upscaling (see Table 7-5). The VO accepts memory image data in YUV 4:2:2 co-sited, YUV 4:2:2 interspersed and YUV 4:2:0 formats, and generates a CCIR 656 compatible, YUV 4:2:2 co-sited image output stream. Scaling is identified by the *YUV-1×* and *YUV-2×* modes. In *YUV-1×* modes, luminance and chrominance pass unmodified. In *YUV-2×* modes, luminance and chrominance are horizontally upscaled by a factor of two.

During image transfer, the YTR bit is set in the status register when the Image Line Counter reaches the Y THRESHOLD value. When an image field has been transferred, the BFR1_EMPTY bit is set in the status register. The DSPCPU is interrupted when either the YTR or BFR1_EMPTY flag is set and its corresponding interrupt is enabled. To maintain continuous transfer of image fields, the DSP CPU supplies new pointers for the next field following each BFR1_EMPTY interrupt. If the DSPCPU does not supply new pointers before the next field, the URUN bit is set, and the VO uses the same pointer values until they are updated.

YUV Overlay

YUV overlay is enabled by the OL_EN bit in the VO_CTL register. The YUV overlay is typically a computer-generated graphic overlaid onto the output image. The YUV overlay is either generated by the DSPCPU or converted by the DSPCPU from a RGB to a YUV overlay image. The DSPCPU performs RGB to YUV conversion, if required, because this conversion can potentially lose information. Since the DSPCPU typically generates the image, the DSPCPU has the most information about performing this conversion in the most effective manner.

The overlay heigth should be chosen suchthat the overlay does not vertically extend beyond the image area. A heigth greater than this causes undefined results and may result in vertical overlay wraparound.

The YUV overlay logic assembles the U0, Y0, V0, Y1 bytes for a pair of YUV 4:2:2 pixels for both the main image and the overlay image. The alpha bit for pixel 0 (the LSB of the U0 byte of the overlay image) selects ALPHA ZERO or ALPHA ONE as the alpha source, and the alpha blend logic combines U0, Y0, and V0 from the main and overlay images to generate the U0, Y0 and V0 output values. The alpha bit for pixel 1 (the LSB of the V0 byte of the overlay image) selects ALPHA ZERO or ALPHA ONE as the alpha source for blending the Y1 pixels to generate the Y1 output value. The alpha blended U0, Y0, V0 and Y1 bytes are sent to the VO output port in the YUV 422 sequence.

Image Addressing

The output image is read from SDRAM at a location defined by Y_BASE_ADR, Y_OFFSET, U_BASE_ADR, U_OFFSET, V_BASE_ADR, and V_OFFSET. The default memory packing is big-endian although little-endian packing is also supported by setting the LTL_END bit in the VO_CTL register.

Horizontally adjacent samples are stored at successive byte addresses, resulting in a packed form (four 8-bit samples are packed into one 32-bit word). Upon horizontal retrace, the starting byte address for the next line is computed by adding the corresponding OFFSET value to the previous line's starting byte address. Note that OFFSET is a 16-bit unsigned quantity. This process continues until the total image—height in lines and width in pixels per line—have been read from memory for luminance (Y). For chrominance, the same number of lines are read but half the number of pixels per line are read in YUV 4:2:2 and YUV 4:2:0 formats[1]. The YUV 4:2:0 format has half the number of U and V pixels in memory that the YUV 4:2:2 formats have, but each line of U and V data is read twice. See Figure 7-16 through Figure 7-19.

7.12.2 Data Streaming and Message Passing Modes

In the *data streaming* and *message passing* modes, the VO supplies a stream of eight-bit, unsigned data at up to 80 MHz data rate to the VO_DATA[7:0] pins. The data is read from SDRAM in packed form (four 8-bit bytes per 32-bit word). The default packing is big-endian although little-endian packing is also supported by setting the LTL_END bit. No data selection or data interpretation is done, and data is transferred at one byte per VO_CLK.

Data-Streaming Mode. In the *data streaming* mode, data is stored in SDRAM in two buffer tables. When the VO has transferred out the contents of one table, it interrupts the DSPCPU and begins transferring out the contents of the second table. The DSPCPU supplies pointers to both tables. The VO can provide a continuous stream of data to the VO output if the DSPCPU updates the pointer to the next table before the VO starts transferring data from the next table.

When each buffer has been transferred, the corresponding buffer empty bit is set in the status register, and the DSPCPU is interrupted if the buffer empty interrupt is enabled. To maintain continuous transfer of data, the DSPCPU supplies new pointers for the next data buffer following each buffer empty interrupt. If the DSPCPU does not supply new pointers before the next field, the URUN bit is set, and the VO uses the same pointer values until they are updated.

1. Note that consecutive Pixel components of each line are stored in consecutive memory addresses but consecutive lines need not be in consecutive memory addresses

Message-Passing Mode. In the *message passing* mode data is stored in SDRAM in one buffer table. In this mode, it is required that SYNC_MASTER is set to ensure correct operation of VO_IO1 and VO_IO2 as outputs. When message passing is started by setting VO_ENABLE in the VO_CTL register, the VO sends a Start condition on VO_IO1. When the VO has transferred out the contents of the table, it sends an End condition on VO_IO2 as shown in Figure 7-15, sets BFR1_EMPTY, and interrupts the DSPCPU. The VO stops and no further operation takes place until the DSPCPU sets VO_ENABLE for another message or other VO operation.

7.12.3 Interrupts and Error Conditions

The VO has five interrupt conditions defined by bits in the VO_STATUS register. These are BFR1_EMPTY, BFR2_EMPTY, HBE, URUN, and YTR. Each of these conditions has a corresponding interrupt enable flag and interrupt acknowledge action bit in the VO_CTL register.

VO asserts a SOURCE 10 interrupt request to the TM1000 vectored interrupt controller as long as one or more enabled events are asserted. The interrupt controller should always be set such that the Video Out interrupt operates in level triggered mode. This ensures that no event is lost to the interrupt handler. Refer to Section 3.4.3, “INT and NMI (Maskable and Non-Maskable Interrupts),” for a description of setting level triggered mode as well as recommendations on writing interrupt handlers.

The BFR1_EMPTY, BFR2_EMPTY and YTR interrupts are status flags to the DSPCPU indicating that a buffer has been emptied or that the Y threshold has been reached.t

The URUN flag indicates that the DSPCPU did not update the address pointers for the next field or buffer. In this case, the VO uses the old address pointer value and continues image or data transfer. When the DSPCPU updates the pointer, the new pointer value will be used at the start of the next frame or buffer transfer. The URUN flag is therefore a status flag that tells the DSPCPU that the VO is using the old pointer values because it did not receive the new ones in time.

The HBE, Hardware Bandwidth Error flag indicates that the VO did not get data from SDRAM via TM1000’s internal data highway in time to continue the data or image transfer. Data or image transfer will continue, using whatever data is in the VO internal data buffers. The address counter for the failing buffer(s) will continue to count, and the VO will continue to request data from the SDRAM over the highway until the highway can provide the requested data in time.

The VO has no error conditions that cause system hardware problems. The VO is a read only device, transferring data from SDRAM to the VO output port. Unlike Video In, the VO does not modify SDRAM data.

URUN and HBE are the only VO error conditions. In the case of URUN or HBE, the worst that happens is a scrambled image may be displayed for one frame or that incorrect data is sent for one buffer cycle.

Even changing operating modes does not cause a system hardware problem. Changing the MODE bits, the Overlay Enable and Format bits, or the Little Endian bit may cause wrong data to be displayed or transferred. However, the VO does not detect this or stop for it.

In normal operation, the user should not change the mode or transfer control bits while the VO is enabled.The VO should be disabled before changing the MODE bits, the OL_EN bit, or the LTL_END bit. However if these bits are changed while the VO is running, they will take effect at the beginning of the next field or buffer.

7.13 DDS AND PLL FILTER DETAILS

The PLL filter serves to reduce the phase jitter of the DDS output. It can also be used to multiply the DDS output frequency by 2x. The DDS and PLL filter together provide a high quality, accurately programmable output video clock. The complete system is sketched in Figure 7-28. On RESET, the output multiplexer is set in the ‘11’ position, and the PLL system is disabled. To start the system, the following steps are needed:

- Assign a DDS frequency (this starts the DDS). Allow for at least 31 DSPCPU cycles for the DDS frequency setting to take effect.

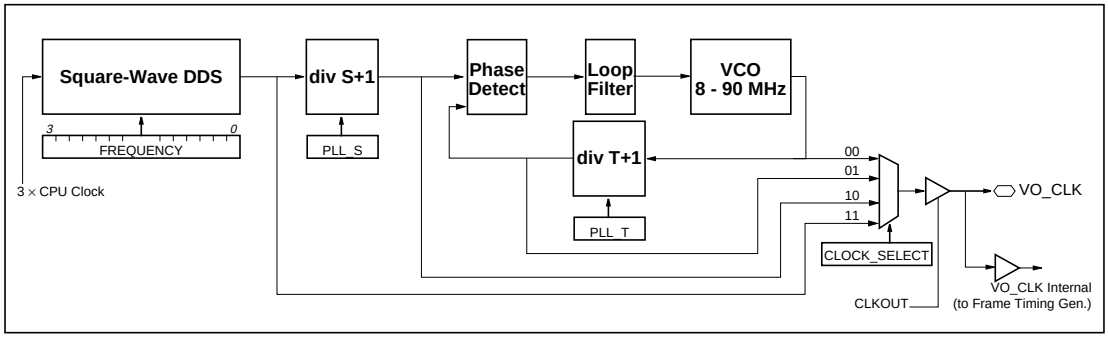


Figure 7-28. PLL filter block diagram

- Choose a value for PLL_S, PLL_T (for 8-40 MHz operation, a value of 1 for division by 2 is recommended).
- Choose a value for CLOCK_SELECT (for 8-40 MHz operation, CLOCK_SELECT=00 is recommended).
- Assign a VO_CTL word containing the above choices. The first assignment with CLOCK_SELECT unequal 11 enables the PLL system. Allow for max. 50 microseconds to achieve lock. The PLL remains enabled until the next RESET.

Once the PLL is locked, small changes to the DDS frequency are allowed, and the VO_CLK output will smoothly track the frequency change.

Note that most consumer electronics equipment imposes *very high precision requirements* on the value of the color burst frequency. In the case of using the VO_CLK to achieve longterm video frame synchronization to a single master reference, special care is required to keep the color burst signal frequency within a tolerance of some 50 ppm. When using a Philips DENC (Digital Encoder), the color burst frequency is derived from the master DENC frequency by a programmable synthesizer on the DENC chip. In this case, VO_CLK changes larger than 50 ppm are allowed by changing the DENC synthesizer to compensate for the VO_CLK change.

A separate application note will describe the best settings for the DDS frequency and PLL filter parameters (PLL_S, PLL_T and CLOCK_SELECT) to achieve minimal jitter for a given frequency.

Table 7-10 illustrates several example settings.

Table 7-10. DDS and PLL example settings

Desired Frequency	DDS frequency	PLL_S	PLL_T	CLOCK_SELECT	Usage
4 - 10 MHz	8 - 20 MHz	1 (divide by 2)	1 (divide by 2)	01 (T divider)	custom low speed video
8 - 45 MHz	8 - 45 MHz	1 (divide by 2)	1 (divide by 2)	00 (VCO)	standard or 16:9 digital video
40 - 80 MHz	20 - 40 MHz	1 (divide by 2)	3 (divide by 4)	00 (VCO)	high pixel rate custom video

Audio In Chapter 8

by Gert Slavenburg

8.1 AUDIO IN OVERVIEW

The TM1000 Audio In unit connects to an off-chip stereo A/D converter subsystem through a flexible bit-serial connection. Audio In provides all signals needed to interface to high quality, low cost oversampling A/D converters, including a generator for a precisely programmable oversampling A/D system clock. The Audio In unit and external A/D together provide the following capabilities:

- One or two channels of audio input.
- Eight- or 16-bit samples per channel.
- Programmable 1-Hz to 100 kHz sampling rate.
- 0.07-Hz frequency resolution oversampling clock.
- Internal or external sampling clock source.
- Audio In autonomously writes sampled audio data to memory using double buffering (DMA).
- Eight-bit mono and stereo as well as 16-bit mono and stereo PC standard memory data formats are supported.
- Little- and big-endian memory formats are supported.

8.2 EXTERNAL INTERFACE

Four TM1000 pins are associated with the Audio In unit. The AI_OSCLK output is an accurately programmable clock output intended to serve as the master system clock for the external A/D subsystem. The other three pins (AI_SCK, AI_WS and AI_SD) constitute a flexible serial input interface. Using the Audio In MMIO registers, these pins can be configured to operate in a variety of serial interface framing modes, including but not limited to:

- Standard stereo I^2S (MSB first, 1-bit delay from AI_WS, left & right data in a frame).[1]
- LSB first, with 1–16 bit data per channel.
- Complex serial frames of up to 512 bits/frame, with ‘valid sample’ qualifier bit.

1. A definition of the Philips I^2S serial interface protocol, among others, can be found in the Philips IC01 databook.

Table 8-1. Audio-In Unit External Signals

Signal	Type	Description
AI_OSCLK	OUT	Over-Sampling Clock. This output can be programmed to emit any frequency up to 40-MHz with a resolution of 0.07-Hz. It is intended for use as the $256f_s$ or $384f_s$ over sampling clock by external A/D subsystem.
AI_SCK	I/O-5	• When Audio-In is programmed as serial-interface timing slave (power-up default), AI_SCK is an input. AI_SCK receives the serial bitclock from the external A/D subsystem. This clock is treated as fully asynchronous to TM1000 main clock. When Audio In is programmed as the serial-interface timing master, AI_SCK is an output. AI_SCK drives the serial clock for the external A/D subsystem. The frequency is a programmable integral divide of the AI_OSCLK frequency. AI_SCK is limited to 20 MHz. The sample rate of valid samples embedded within the serial stream is limited to 100 kHz.
AI_SD	IN-5	Serial Data from external A/D subsystem. Data on this pin is sampled on positive or negative edges of AI_SCK as determined by the CLOCK_EDGE bit in the AI_SERIAL register.
AI_WS	I/O-5	• When Audio In is programmed as the serial-interface timing slave (power-up default), AI_WS acts as an input. AI_WS is sampled on the same edge as selected for AI_SD. • When Audio In is programmed as the serial-interface timing master, AI_WS acts as an output. It is asserted on the opposite edge of the AI_SD sampling edge. AI_WS is the word-select or frame-synchronization signal from/to the external A/D subsystem.

The Audio In can be used with many serial A/D converter devices, including the Philips SAA7366 (stereo A/D), Crystal Semiconductor CS5331, CS5336 (stereo A/D’s), CS4218 (codec), Analog Devices AD1847 (codec).

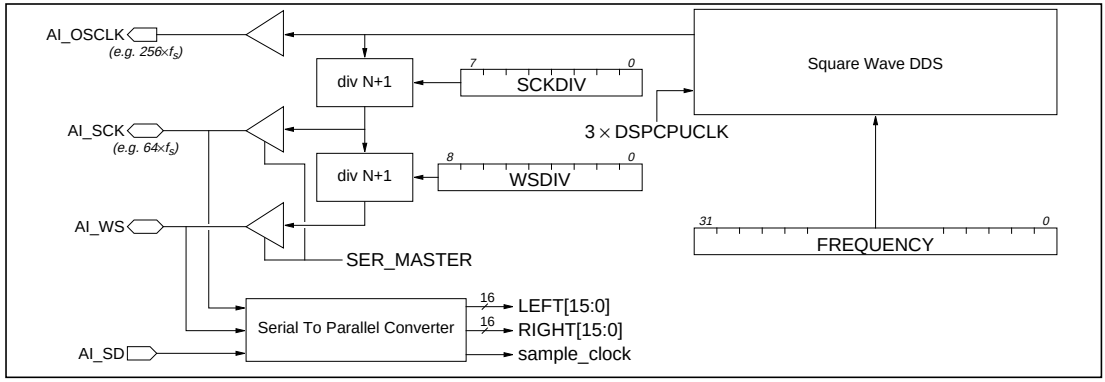


Figure 8-1. Audio In clock system and I/O interface.

8.3 CLOCK SYSTEM

Figure 8-1 illustrates the different clock capabilities of the Audio In unit. At the heart of the clock system is a square wave DDS (Direct Digital Synthesizer). The DDS can be programmed to emit frequencies from ca. 1 Hz to 40 MHz with a resolution of 0.07 Hz. Programming is accomplished through the Audio In FREQUENCY register:

$$f_{OSCLK} = \frac{3 \times FREQUENCY \times f_{DSPCPUCLK}}{2^{32}}$$

The FREQUENCY can be changed by software at any time. The effect of such changes is to pull-in or delay the next clock edge, i.e. the instantaneous phase of the clock is not disturbed. This allows fine control over sample capture rate to longterm track an absolute system timing source.

The output of the DDS is always sent on the AI_OSCLK output pin. This output is typically used as the $256f_s$ or $384f_s$ system clock source instead of a fixed frequency crystal for oversampling A/D converters, such as the Philips SAA7366T, or Analog Devices AD1847.

AI_SCK and AI_WS can be configured as input or output, as determined by the SER_MASTER control field. As output, AI_SCK is a divider of the DDS output frequency. Whether input or output, the AI_SCK pin signal is used as the bit clock for serial-parallel conversion.

$$f_{AISCK} = \frac{f_{AIOSCLK}}{SCKDIV + 1} \qquad SCKDIV \in [0,255]$$

If set as output, AI_WS can similarly be programmed using WSDIV to control the serial frame length from 1 to 512 bits.

Table 8-2. Sample Rate Settings ($f_{DSPCPUCLK}$=100 MHz)

f_s	OSCLK	SCK	FREQUENCY	SCKDIV
44.1 kHz	$256f_s$	$64f_s$	161628209	3
48.0 kHz	$256f_s$	$64f_s$	175921860	3
44.1 kHz	$384f_s$	$64f_s$	242442314	5
48.0 kHz	$384f_s$	$64f_s$	263882791	5

The preferred application of the clock system options is to use AI_OSCLK as A/D master clock, and let the A/D converter be timing master over the serial interface (SER_MASTER=0).

In case of use of an external codec (e.g. the AD1847 or CS4218) for common Audio In and Audio Out use, it may not be possible to independently control the A/D and D/A system clocks. In that case it is recommended that the Audio Out clock system DDS is used to provide a single master A/D and D/A clock. The Audio Out, or the D/A converter, can be used as serial interface timing master, and Audio In is set to be slave to the serial frame determined by Audio Out (Audio In SER_MASTER=0, AI_SCK and AI_WS externally wired to the corresponding Audio Out pins). In such systems, independent software control over A/D and D/A sampling rate is not possible, but component count is minimized.

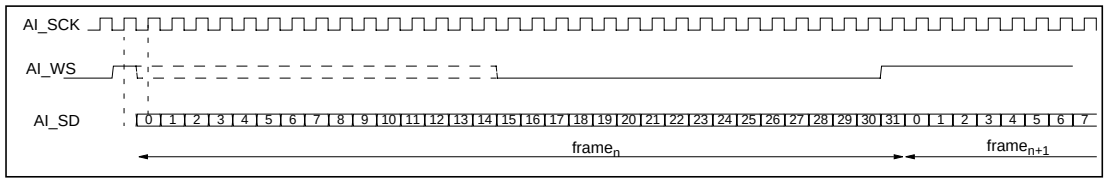


Figure 8-2. Audio In serial frame and bit position definition (POLARITY=1, CLOCK_EDGE=0).

Table 8-3. Audio In MMIO Clock & Interface Control Bits

Field Name	Description
SER_MASTER	0 ⇒ (RESET default), the A/D converter is the timing master over the serial interface. AI_SCK and AI_WS are set to be input. 1 ⇒ TM1000 is the timing master over the Audio In serial interface. The AI_SCK and AI_WS pins are set to be outputs.
FREQUENCY	Sets the clock frequency emitted by the AI_OSCLK output. RESET default 0.
SCKDIV	Sets the divider used to derive AI_SCK from AI_OSCLK. Set to 0..255, for division by 1..256. RESET default 0.
WSDIV	Sets the divider used to derive AI_WS from AI_SCK. Set to 0..511 for a serial frame length of 1..512. RESET default 0.

8.4 SERIAL DATA FRAMING

The Audio In unit can accept data in a wide variety of serial data framing conventions. Figure 8-2 illustrates the notion of a serial frame. If POLARITY=1 and CLOCK_EDGE=0, a frame is defined with respect to the positive transition of the AI_WS signal, as observed by a positive clock transition on AI_SCK. Each data bit sampled on positive AI_SCK transitions has a specific bit position: the data bit sampled on the clock edge after the clock edge on which the AI_WS transition is seen has bit position 0. Each subsequent clock edge defines a new bit position. As defined in Table 8-4, other combinations of POLARITY and CLOCK_EDGE can be used to define a variety of serial frame bitposition definitions.

The capturing of samples is governed by FRAMEMODE. If FRAMEMODE=00, every serial frame results in one sample from the serial-parallel converter. A sample is defined as a left/right pair in stereo modes or a single left channel value in mono modes. If FRAMEMODE=1y, the serial frame data bit in bit position VALIDPOS is examined. If it has value ‘y’, a sample is taken from the data stream (the valid bit is allowed to precede or follow the left or right channel data provided it is in the same serial frame as the data).

The left and right sample data can be in a LSB-first or MSB-first form, at an arbitrary bit position, and with an arbitrary length.

Table 8-4. Audio In MMIO Serial Framing Control Fields

Field Name	Description
POLARITY	0 ⇒ serial frame starts on AI_WS negedge (RESET default) 1 ⇒ serial frame starts on AI_WS posedge
FRAMEMODE	00 ⇒ accept a sample every serial frame (RESET default) 01 ⇒ unused, reserved 10 ⇒ accept sample if valid bit = 0 11 ⇒ accept sample if valid bit = 1

Table 8-4. Audio In MMIO Serial Framing Control Fields

Field Name	Description
VALIDPOS	• Defines the bit position within a serial frame where the valid bit is found. • Default 0.
LEFTPOS	• Defines the bit position within a serial frame where the first data bit of the left channel is found. • Default 0.
RIGHTPOS	• Defines the bit position within a serial frame where the first data bit of the right channel is found. • Default 0.
DATAMODE	0 ⇒ MSB first (RESET default) 1 ⇒ LSB first
SSPOS	• Start/Stop bit position. Default 0. • If DATAMODE=MSB first, SSPOS determines the bit index (0..15) in the parallel word of the last data bit. Bits 15 (MSB) up to/including SSPOS are taken in order from the serial frame data. All other bits are set to zero. • If DATAMODE=LSB first, SSPOS determines the bit index (0..15) in the parallel word of the first data bit. Bits SSPOS up to/including 15 are taken in order from the serial frame data. All other bits are set to zero.
CLOCK_EDGE	• if 0 (RESET default) the AI_SD and AI_WS pins are sampled on positive edges of the AI_SCK pin. If SER_MASTER =1, AI_WS is asserted on negative edges of AI_SCK. • if 1, AI_SD and AI_WS are sampled on negative edges of AI_SCK. As output, AI_WS is asserted on positive edges of AI_SCK.

In MSB-first mode, the serial-to-parallel converter assigns the value of the bit at LEFTPOS to LEFT[15]. Subsequent bits are assigned, in order, to decreasing bit positions in the LEFT data word, up to and including LEFT[SSPOS]. Bits LEFT[SSPOS–1:0] are cleared. Hence, in MSB-first mode, an arbitrary number of bits are captured. They are left-adjusted in the 16-bit parallel output of the converter.

In LSB-first mode, the serial to parallel converter assigns the value of the bit at LEFTPOS to LEFT[SSPOS]. Subsequent bits are assigned, in order, to increasing bit positions in the LEFT data word, up to and including LEFT[15]. Bits LEFT[SSPOS–1:0] are cleared. Hence, in LSB- first mode, an arbitrary number of bits are captured. They are returned left-adjusted in the 16 bit parallel output of the converter.

Refer to Figure 8-3 and Table 8-5 to see an example of how the Audio In unit MMIO registers are set to collect 16 bits samples using the Philips SAA7366 I^2S 18 bit A/D converter. The setup assumes the SAA7366 acts as the serial master.

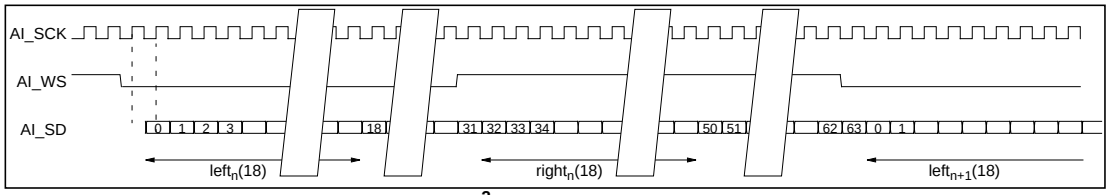


Figure 8-3. Serial frame of the SAA7366 18 bit I^{2}S A/D converter (format 2 SWS).

For the sake of example, if it were desired to use only the 12 MSBs of the A/D converter in Figure 8-3, use the settings of Table 8-5 with SSPOS set to four. This results in LEFT[15:4] being set with data bits 0..11, and LEFT[3:0] being set equal to zero. RIGHT[15:4] is set with data bits 32..43 and RIGHT[3:0] is set to zero.

Table 8-5. Example Setup For SAA7366

Field	Value	Explanation
SER_MASTER	0	SAA7366 is serial master
FREQUENCY	161628209	256f$_s$ 44.1 kHz
SCKDIV	3	AI_SCK set to AI_OSCLK/4 (not needed since SER_MASTER=0)
WSDIV	63	Serial frame length of 64 bits (not needed since SER_MASTER=0)
POLARITY	0	Frame starts with neg. AI_WS
FRAMEMODE	00	Take a sample each ser. frame
VALIDPOS	n/a	Don't care
LEFTPOS	0	Bit position 0 is MSB of left channel and will go to LEFT[15]
RIGHTPOS	32	Bit position 32 is MSB of right channel and will go to RIGHT[15]
DATAMODE	0	MSB first
SSPOS	0	Stop with LEFT/RIGHT[0]
CLOCK_EDGE	0	Sample WS and SD on positive SCK edges for I^{2}S

8.5 MEMORY DATA FORMATS

The Audio In unit autonomously writes samples to memory in mono and stereo 8 and 16 bit per sample formats, as shown in Figure 8-4. Successive samples are always stored at increasing memory address locations. The setting of the LITTLE_ENDIAN bit in the AI_CTL register determines how increasing memory addresses map to byte positions within words. Refer to Appendix C, "Endian-ness," for details on byte ordering conventions.

The Audio In unit hardware implements a double buffering scheme to ensure that no samples are lost, even if the DSPCPU is highly loaded and slow to respond to interrupts. The DSPCPU software assigns buffers by writing a base address and size to the MMIO control fields described in Table 8-6. Refer to section 8.6 for details on hardware/software synchronization.

In eight-bit capture modes, the eight MSBs of the serial parallel converter output data are written to memory. In 16-bit capture modes, all bits of the parallel data are written to memory. If SIGN_CONVERT is set to one, the MSB of the data is inverted, which is equivalent to translating from two's complement to offset binary representation. This allows the use of an external two's complement 16-bit A/D converter to generate eight-bit unsigned samples, which is often used in PC audio.

Table 8-6. Audio In MMIO DMA Control Fields

Field Name	Description
LITTLE_ENDIAN	0 ⇒ capture in big endian memory format (RESET default) 1 ⇒ capture little endian
BASE1	Base Address of buffer1. This must be a 64-byte aligned address in local SDRAM. RESET default 0.
BASE2	Base Address of buffer2. This must be a 64-byte aligned address in local SDRAM. RESET default 0.
SIZE	• Number of samples to be placed in buffer before switching to other buffer. • In stereo modes, a pair of 8- or 16-bit data counts as 1 sample. In mono modes, a single value counts as a sample. • RESET default 0.
CAP_MODE	00 ⇒ mono (left ADC only), 8 bits/sample. (RESET Default). 01 ⇒ stereo, 2 times 8 bits/sample 10 ⇒ mono (left ADC only), 16 bits/sample 11 ⇒ stereo, 2 times 16 bits/sample
SIGN_CONVERT	0 ⇒ leave MSB unchanged (RESET default) 1 ⇒ invert MSB

Note that the Audio In hardware does *not* generate A-law or μ-law 8 bit data formats. If such formats are desired, the DSPCPU can be used to convert from 16-bit linear data to A-law or μ-law data.

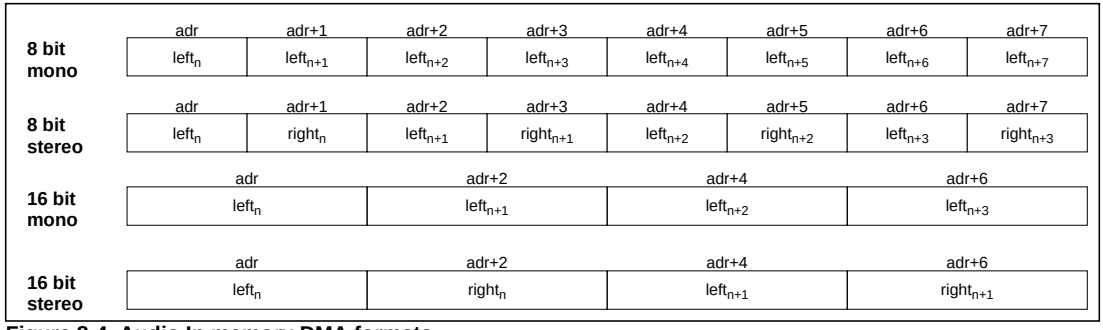


Figure 8-4. Audio In memory DMA formats.

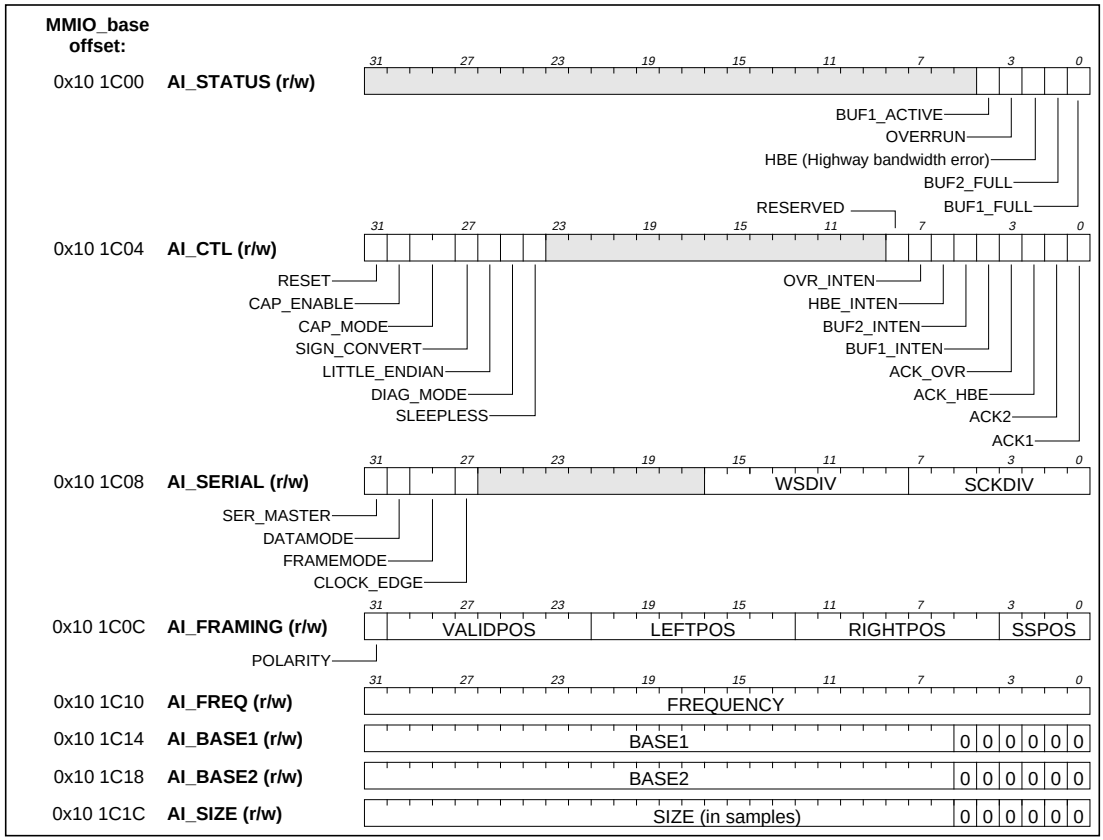


Figure 8-5. Audio In status/control field MMIO layout.

8.6 AUDIO IN OPERATION

Table 8-9 and Table 8-8 describe the function of the control and status fields of the Audio In unit.

The Audio In unit is reset by a TM1000 hardware reset, or by writing 0x80000000 to the AI_CTL register. Upon RESET, capture is disabled (CAP_ENABLE = 0), and buffer1 is the active buffer (BUF1_ACTIVE=1). A minimum of 5 valid AI_SCK clock cycles is required to allow internal Audio In circuitry to stabilize before enabling capture. This can be accomplished by programming AI_FREQ and AI_SERIAL and then delaying for the appropriate time interval.

The DSPCPU initiates capture by providing two equal size empty buffers and putting their base address and

size in the $BASE_n$ and SIZE registers. Once two valid (local memory) buffers are assigned, capture can be enabled by writing a '1' to CAP_ENABLE. The Audio In unit hardware now proceeds to fill buffer 1 with input samples. Once buffer 1 fills up, BUF1_FULL is asserted, and capture continues without interruption in buffer 2. If BUF1INTEN is enabled, a SOURCE 11 interrupt request is generated.

Note that the buffers must be 64-byte aligned, and a multiple of 64 samples in size (the six LSBs of AI_BASE1, AI_BASE2 and AI_SIZE are always zero).

The DSPCPU is required to assign a new, empty buffer to BASE1 and perform an ACK1, before buffer 2 fills up. Capture continues in buffer 2, until it fills up. At that time, BUF2_FULL is asserted, and capture continues in the new buffer 1, etc.

Upon receipt of an ACK, the Audio In hardware removes the related interrupt request line assertion at the next DSPCPU clock edge. Refer to Section 3.4.3, "INT and NMI (Maskable and Non-Maskable Interrupts)," for the rules regarding ACK and interrupt re-enabling. The Audio In interrupt should always be operated in level sensitive mode, since Audio In can signal multiple conditions that each need independent ACKs over the single internal SOURCE 11 request line.

In normal operation, the DSPCPU and Audio In hardware continuously exchange buffers without ever loosing a sample. If the DSPCPU fails to provide a new buffer in time, the OVERRUN error flag is raised. This flag is *not affected* by ACK1 or ACK2; it can only be cleared by an explicit ACK_OVR.

8.7 HIGHWAY LATENCY AND HBE

Audio In uses internal buffering before writing data to SDRAM. The internal buffer consists of a 32-bit input register and 64 bytes of internal buffer memory. Under normal operation, the 64-byte buffer gets written to SDRAM while the 32-bit input register is capable of receiving four, two or one more sample (depending on CAP_MODE). This normal operation is guaranteed to be maintained as long as the highway arbiter is set to guarantee a latency for Audio In that matches the active mode and sampling interval. Given a sample rate f_S, and an associated sample interval T (in DSPCPU clock cycles), the arbiter should be set to have a latency of at most T-2 cycles for stereo 16 bit mode, 2T-2 for mono 16 bit and stereo 8 bit modes and 4T-2 for mono 8 bit mode. Refer to Chapter 19, "Arbiter," for information on arbiter programming. If the requested latency is not adequate, the HBE (Highway Bandwidth Error) condition may result. This error flag gets set when the input register is full, the 64-byte buffer has not yet been written to memory, and a new sample arrives.

8.8 ERROR BEHAVIOR

If either an OVERRUN or HBE error occurs, input sampling is temporarily halted, and samples will be lost. In case of OVERRUN, sampling resumes as soon as the DSPCPU makes one or more new buffers available through an ACK1 or ACK2 operation. In the case of HBE, sampling will resume as soon as the internal buffer can be written to SDRAM.

HBE and OVERRUN are 'sticky' error flags. They will remain set until an explicit ACK_HBE or ACK_OVR.

Table 8-7Audio In Highway Arbiter Latency Requirements (100 MHz)

CapMode	f_S	T	Max. latency (cycles)
stereo 16 bit/sample	44100 Hz	2267	2265
stereo 16 bit/sample	48000 Hz	2083	2081
stereo 16 bit/sample	96000 Hz	1041	1039

Table 8-8. Audio In MMIO Status Fields (Read Only)

Field Name	Description
BUF1_ACTIVE	• If 1, buffer 1 is the buffer that will be used for the next incoming sample. If 0, buffer 2 will receive the next sample. • 1 after RESET.
BUF1_FULL	• If 1, buffer 1 is full. If BUF1_INTEN is also 1, an interrupt request (source 11) is pending. BUF1FUL is cleared by writing a '1' to ACK1, at which point the Audio In hardware will assume that BASE1 and SIZE describe a new empty buffer. • 0 after RESET.
BUF2_FULL	• If 1, buffer 2 is full. If BUF2_INTEN is also 1, an interrupt request (source 11) is pending. BUF2FUL is cleared by writing a '1' to ACK2, at which point the Audio In hardware will assume that BASE2 and SIZE describe a new empty buffer. • 0 after RESET.
HBE	• Highway Bandwidth Error. This error condition is raised when the 64 byte internal Audio In buffer is not yet written to SDRAM when a new input sample arrives. This indicates an insufficient allocation of TM1000 Highway bandwidth for the audio sampling rate/mode. Refer to Chapter 19, "Arbiter." • 0 after RESET.
OVERRUN	• An OVERRUN error has occurred, i.e. the CPU failed to provide an empty buffer in time, and 1 or more samples have been lost. If OVR_INTEN is also 1, an interrupt request (source 11) is pending. The OVERRUN flag can ONLY be cleared by writing a '1' to ACK_OVR. • 0 after RESET.

8.9 DIAGNOSTIC MODE

Diagnostic mode is entered by setting the DIAGMODE bit in the AI_CTL register. In diagnostic mode, the AI_SCK, AI_WS and AI_SD inputs of the serial-parallel converter are taken from the output pins of the TM1000

Table 8-9. Audio In MMIO Control Fields

Field Name	Description
RESET	The Audio In logic is reset by writing a 0x80000000 to AI_CTL. This bit always reads as a '0'. See Section 8.6, "Audio In Operation" for details on software reset.
DIAGMODE	0 ⇒ normal operation (RESET default) 1 ⇒ diagnostic mode (see Section 8.9, "Diagnostic Mode")
SLEEPLESS	0 ⇒ participate in global power-down (RESET default) 1 ⇒ refrain from participating in power-down
CAP_ENABLE	Capture Enable flag. If 1, Audio In captures samples and acts as DMA master to write samples to local SDRAM. If 0 (RESET default), Audio In is inactive.
BUF1_INTEN	Buffer 1 full Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 1 full
BUF2_INTEN	Buffer 2 full interrupt enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 2 full
HBE_INTEN	HBE Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if a highway bandwidth error occurs.
OVR_INTEN	Overrun Interrupt Enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if an overrun error occurs

Table 8-9. Audio In MMIO Control Fields

Field Name	Description
ACK1	Write a 1 to clear the BUF1FUL flag and remove any pending BUF1FUL interrupt request. This bit always reads as 0.
ACK2	Write a 1 to clear the BUF2FUL flag and remove any pending BUF2FUL interrupt request. This bit always reads as 0.
ACK_HBE	Write a 1 to clear the HBE flag and remove any pending HBE interrupt request. This bit always reads as 0.
ACK_OVR	Write a 1 to clear the OVERRUN flag and remove any pending OVERRUN interrupt request. This bit always reads as 0.

Audio Out unit. This mode can be used during the diagnostic phase of system boot to verify correct operation of most of the logic circuitry of the Audio Out and Audio In unit.

Note that the inputs are truly taken from the TM1000 Audio Out external pins, i.e. if an external (board level) source is driving AO_SCK or AO_WS, diagnostic mode is not capable of testing Audio Out.

Special care must be taken to enable diagnostic mode. The recommended way of entering diagnostic mode is to first set Audio Out up such that a AO_SCK is generated and set DIAGMODE bit followed by a 5 (AI_SCK) cycle delay, then do a software reset of Audio In and immediatly set back the DIAGMODE bit.

Audio Out Chapter 9

by Gert Slavenburg, Patrick de Bakker, Charles Peplinski

9.1 AUDIO OUT OVERVIEW

The TM1000 Audio Out unit connects to an off-chip stereo D/A converter subsystem through a flexible bit-serial connection. Audio Out provides all signals to interface to high quality, low cost oversampling D/A converters, including a precisely programmable oversampling D/A system clock. The Audio Out unit and external D/A together provide the following capabilities:

- Up to 8 channels of audio output.
- Eight- or 16-bit samples per channel.
- Programmable 1-Hz to 100-kHz sampling rate, with 0.07-Hz resolution.
- Internal or external sampling clock source.
- Audio Out autonomously reads processed audio data from memory using double buffering (DMA).
- Eight-bit mono and stereo as well as 16-bit mono and stereo PC standard memory data formats are supported.
- Little- and big-endian memory formats are supported.
- Provides control capability for highly integrated PC codecs such as the AD1847 and CS4218.

9.2 EXTERNAL INTERFACE

Four TM1000 pins are associated with the Audio Out unit. The AO_OSCLK output is an accurately programmable clock output intended to be used as the master system clock for the external D/A subsystem. The other three pins (AO_SCK, AO_WS and AO_SD) constitute a flexible serial output interface. Using the Audio Out MMIO registers, these pins can be configured to operate in a variety of serial interface framing modes, including but not limited to:

- Standard stereo I^2S (MSB first, one-bit delay from AO_WS, left & right data in a frame).[1]
- LSB first, with 1–16 bit data per channel.
- Complex serial frames of up to 512 bits/frame.
- Superframes of up to 4 regular frames can be created for 4,6 or 8 channel modes.

1. A definition of the Philips I^2S serial interface protocol, among others, can be found in the Philips IC01 databook.

Table 9-1. Audio-Out Unit External Signals

Signal	Type	Description
AO_OSCLK	OUT	Oversampling Clock. This output can be programmed to emit any frequency up to 40 MHz, with a resolution of 0.07 Hz. It is intended for use as the 256 or $384f_s$ oversampling clock by the external D/A conversion subsystem.
AO_SCK	I/O-5	• When Audio Out is programmed to act as the serial interface timing slave (RESET default), AO_SCK acts as input. It receives the Serial Clock from the external audio D/A subsystem. The clock is treated as fully asynchronous to the TM1000 main clock. • When Audio Out is programmed to act as serial interface timing master, AO_SCK acts as output. It drives the Serial Clock for the external audio D/A subsystem. The clock frequency is a programmable integral divide of the AO_OSCLK frequency. AO_SCK is limited to 20 MHz. The sample rate of valid samples embedded within the serial stream is limited to 100 kHz.
AO_SD	OUT-5	Serial Data to external audio D/A subsystem. The timing of transitions on this output is determined by the CLOCK_EDGE bit in the AO_SERIAL register, and can be on positive or negative AO_SCK edges.
AO_WS	I/O-5	• When Audio-Out is programmed as the serial-interface timing slave (RESET default), AO_WS acts as an input. AO_WS is sampled on the opposite AO_SCK edge at which AO_SD is asserted. • When Audio Out is programmed as serial-interface timing master, AO_WS acts as an output. AO_WS is asserted on the same AO_SCK edge as AO_SD. AO_WS is the word-select or frame-synchronization signal from/to the external D/A subsystem. Each audio channel receives 1 sample for every WS period.

9.3 CLOCK SYSTEM

Figure 9-1 illustrates the different clock capabilities of the Audio Out unit. At the heart of the clock system is a

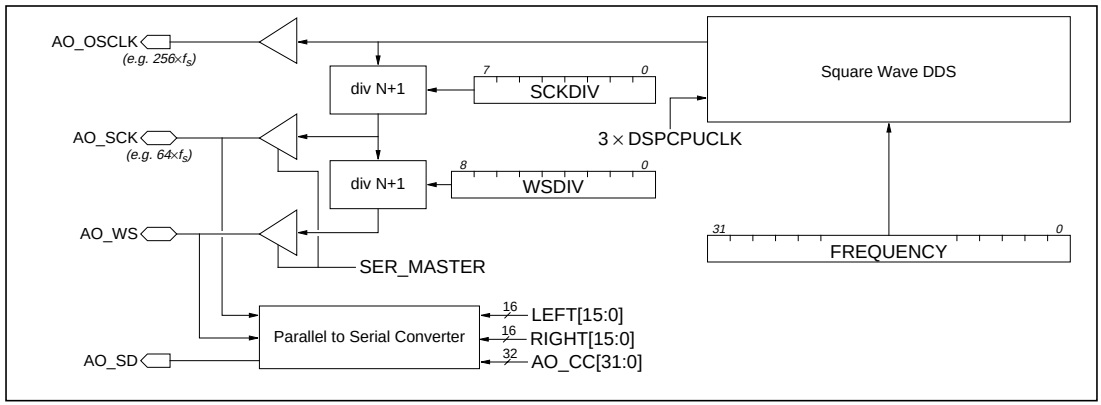


Figure 9-1. Audio out clock system and I/O interface

(Refer to Figure 9-11 for AO_WS detail).

square wave DDS (Direct Digital Synthesizer). The DDS can be programmed to emit frequencies from ca. 1 Hz to 40 MHz with a resolution of 0.07 Hz. Programming is accomplished through the Audio Out FREQUENCY register:

$$f_{OSCLK} = \frac{3 \times FREQUENCY \times f_{DSPCPUCLK}}{2^{32}}$$

The programmer is free to change FREQUENCY, and hence the system sample rate to long-term track any absolute timing source and/or control software buffer fullness. Changes to the FREQUENCY register pull-in or delay the next clock edge and have no instantaneous effect on clock level, i.e. phase speed progression is changed, not phase. The output of the DDS is always sent to the AO_OSCLK output pin. This output is typically used as the $256f_s$ or $384f_s$ system clock source instead of a fixed frequency crystal for oversampling D/A converters, such as the Philips SAA7322, or codecs such as the AD1847 or CS4218.

AO_SCK and AO_WS can be configured as input or output, as determined by the SER_MASTER control field. As output, AO_SCK can be set to a divider of the DDS output frequency.

$$f_{AOSCK} = \frac{f_{AOOSCLK}}{SCKDIV + 1} \qquad SCKDIV \in [0,255]$$

Whether set as input or output, the AO_SCK pin signal is always used as the bit clock for parallel-serial conversion. The AO_WS pin always acts as the trigger to start the generation of a serial frame. AO_WS can similarly be programmed using WSDIV to control the serial frame length. The number of bits per frame (BPF) is equal to WSDIV+1.

The preferred use of the clock system options is to use AO_OSCLK as D/A master clock, and let the D/A converter be timing slave of the serial interface (SER_MASTER=1). Some D/A converters, like the AD1847, provide better SNR properties if they are configured as serial master instead (SER_MASTER=0). As illustrated by Figure 9-1, the internal parallel to serial converter that constructs the serial frame is oblivious to who is serial master, except in the case of superframes of more than 2 audio channels, as described in Section 9.10, “4, 6 and 8 Channel Audio.”

Table 9-2. Sample Rate Settings ($f_{DSPCPUCLK}$=100 MHz)

f_s	OSCK	SCK	FREQUENCY	SCKDIV
44.1 kHz	256fs	64fs	161628209	3
48.0 kHz	256fs	64fs	175921860	3
44.1 kHz	384fs	64fs	242442314	5
48.0 kHz	384fs	64fs	263882791	5

Table 9-3. Audio Out MMIO Clock & Interface Control

Field Name	Description
SER_MASTER	0 ⇒ (RESET default), the D/A subsystem is the timing master over the Audio Out serial interface. AO_SCK and AO_WS act as inputs. 1 ⇒ TM1000 is the timing master over the serial interface. AO_SCK and AO_WS act as outputs. The SER_MASTER bit should only be changed while Audio Out is disabled, i..e. TRANS_ENABLE = 0.
FREQUENCY	Sets the clock frequency emitted by the AO_OSCLK output. RESET default 0.
SCKDIV	Sets the divider used to derive AO_SCK from AO_OSCLK. Set to 0..255, for division by 1..256. RESET default 0.
WSDIV	Sets the divider used to derive AO_WS from AO_SCK. Set to 0..511 for a serial frame length of 1..512. RESET default 0.

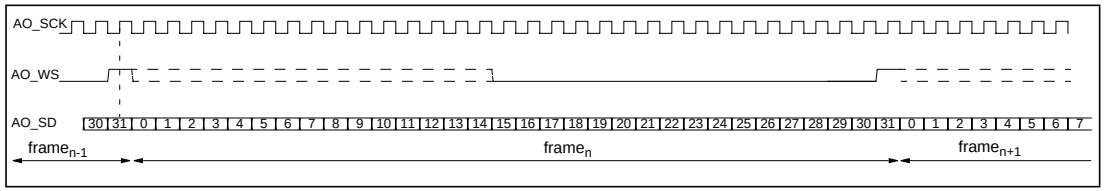


Figure 9-2. Definition of serial frame bit positions (POLARITY = 1, CLOCKEDGE = 0)

9.4 SERIAL DATA FRAMING

The Audio Out unit can generate data in a wide variety of serial data framing conventions. Figure 9-2 illustrates the notion of a serial frame. If POLARITY=1, a frame starts on each positive edge of the AO_WS signal. If CLOCK_EDGE=0, the parallel to serial converter samples AO_WS on a positive clock edge transition, and outputs the first bit (bit 0) of a serial frame on the next falling edge of AO_SCK.

If CLOCK_EDGE = 1, the parallel to serial converter samples AO_WS on the negative edge of AO_SCK, while audio data is output on the positive edge, i.e. the AO_SCK polarity would be reversed with respect to Figure 9-2.

Table 9-4. Audio Out Serial Framing Control Fields

Field Name	Description
POLARITY	0 ⇒ serial frame starts with a AO_WS negedge (RESET default) 1 ⇒ serial frame starts with a AO_WS posedge This bit should NOT be changed during operation of Audio Out, i.e.only update this bit when TRANS_ENABLE = 0.
LEFTPOS(9)	Defines the bit position within a serial frame where the first data bit of the left channel is placed. Default 0.
RIGHTPOS(9)	Defines the bit position within a serial frame where the first data bit of the right channel is placed. Default 0.
DATAMODE	0 ⇒ MSB first (RESET default) 1 ⇒ LSB first
SSPOS	• Start/Stop bit position. Default 0. • If DATAMODE=MSB first, SSPOS determines the bit index (0..15) in the parallel word of the last data bit. Bits 15 (MSB) up to/including SSPOS are generated. All other bits are output as zero. • If DATAMODE=LSB first, SSPOS determines the bit index (0..15) in the parallel word of the first data bit. Bits SSPOS up to/including 15 are generated. All other bits are output as zero.

Table 9-4. Audio Out Serial Framing Control Fields

Field Name	Description
CLOCK_EDGE	0 ⇒ the parallel to serial converter samples AO_WS on positive edges of AO_SCK and outputs data on the negative edge of AO_SCK (RESET default). 1 ⇒ the parallel to serial converter samples AO_WS on negative edges of AO_SCK and outputs data on positive edges of AO_SCK.
WS_PULSE	0 ⇒ emit 50% AO_WS (RESET default). 1 ⇒ emit single AO_SCK cycle AO_WS (this bit is ignored if SER_MASTER=0) In case of 6 channel audio (see Section 9.10, “4, 6 and 8 Channel Audio”), WS_PULSE should be set to ‘1’
SFDIV	See Section 9.10, “4, 6 and 8 Channel Audio,” on superframes.

Every serial frame transmits a single left and right channel sample to the D/A converter. The left and right sample data can be in an LSB first or MSB first form, at an arbitrary bit position, and with an arbitrary length.

In MSB-first mode (DATAMODE = 0), the parallel to serial converter assigns the value of LEFT[15] to the bit at LEFTPOS in the serial frame. Subsequently, bits from decreasing bit positions in the LEFT dataword, up to and including LEFT[SSPOS], are transmitted in order.

In LSB-first mode (DATAMODE = 1), the parallel-to-serial converter assigns the value of LEFT[SSPOS] to the bit at LEFTPOS in the serial frame. Subsequent bits from the LEFT data word, up to and including LEFT[15], are transmitted in order.

Frame bits that do not belong to either LEFT[15:SSPOS] or RIGHT[15:SSPOS] are set to zero. This ensures that TM1000 can be used in combination with a D/A converter which has a higher accuracy than the actual number of transmitted bits.

Refer to Figure 9-3 and Table 9-5 to see how the Audio Out unit MMIO registers would be set to transmit 16 bits of stereo data via an I^2S serial standard to an 18-bit D/A converter with a 64 bit serial frame.

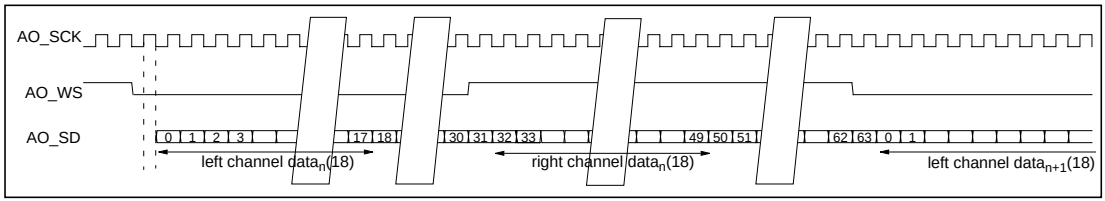


Figure 9-3. Serial frame (64 bit) of a hypothetical 18-bit precision I^{2}S D/A converter.

Table 9-5. Example Setup For I^{2}S

Field	Value	Explanation
POLARITY	0	Frame starts with negedge AO_WS.
LEFTPOS	0	LEFT[15] will go to serial frame position 0.
RIGHTPOS	32	RIGHT[15] will go to serial frame position 32.
DATAMODE	0	MSB first.
SSPOS	0	Stop with LEFT/RIGHT[0], send 0's after.
CLOCK_EDGE	0	AO_SD change on negedge AO_SCK
WSDIV	63	Serial frame length = 64. Only relevant if SER_MASTER=1.
WS_PULSE	0	emit 50% duty cycle AO_WS. Only relevant if SER_MASTER=1.

For the sake of example, if only eight bits were desired to be transmitted, use the settings of Table 9-5 with SSPOS set to 8. This results in LEFT[15:8] being transmitted in data bits 0..7. RIGHT[15:8] is transmitted in data bits 32..39. All other bits in the serial frame are sent as zero.

9.5 CODEC CONTROL

In addition to the left and right data fields that are generated based on autonomous DMA action, a serial frame generated by Audio Out can be set to contain 1 or 2 control fields up to 16 bits in length. Each control field can be independently enabled/disabled by the CC1_EN, CC2_EN bits in AO_CTL. The content shifted into the frame is taken from the CC1 and CC2 field in the AO_CC register. The CC1_POS and CC2_POS fields in the AO_CFC register determine the first bit position in the frame where the control field is emitted. The field is emitted observing the setting of DATAMODE, i.e. LSB or MSB first.

The CC_BUSY bit in AO_STATUS indicates if the Audio Out unit is ready to receive another CC1, CC2 value pair. Writing a new value pair to AO_CC writes the value into a buffer register, and raises the CC_BUSY status. As soon as both CC1 and CC2 values have been copied to a shadow register in preparation for transmission, CC_BUSY is negated, indicating that the Audio Out logic is ready to accept a new codec control pair.

Software always needs to ensure that the CC_BUSY status is deasserted before writing a new CC1, CC2 pair. By busy waiting on CC_BUSY, the DSPCPU can emit a sequence of individual audio frames with distinct control field values reliably. This can for example be used during codec initialization. No provision is made for interrupt driven operation of such a sequence of control values - it is assumed that the value of control fields is rarely changing and can be held constant during the DMA buffer emission of audio.

It is legal to program the control field positions within the frame such that CC1 and CC2 overlap each other and/or left/right data fields. If two fields are defined to start at the same bit position, the priority is left (highest), right, CC1 then CC2. The field with the highest priority will be emitted starting at the conflicting bit position. If a field *f2* is defined to start at a bit position *i* that falls within a field *f1* starting at a lower bit position, *f2* will be emitted starting from *i* and the rest of *f1* will be lost. Any bit positions not belonging to a data or control field will be emitted as zero.

Figure 9-4 shows a 64 bit frame suitable for use with the CS4218 codec. It is obtained by setting POLARITY=1, LEFTPOS=0, RIGHTPOS=32, DATAMODE=0, SSPOS=0, CLOCK_EDGE=1, WS_PULSE=1, CC1_POS = 16, CC1_EN=1, CC2_POS=48, CC2_EN=1.

Note that frames are generated (externally or internally) even when TRANS_ENABLE de-asserted. Writes to CC1 and CC2 should only be done after TRANS_ENABLE is asserted. The 'first' CC values will then go out on the next frame.

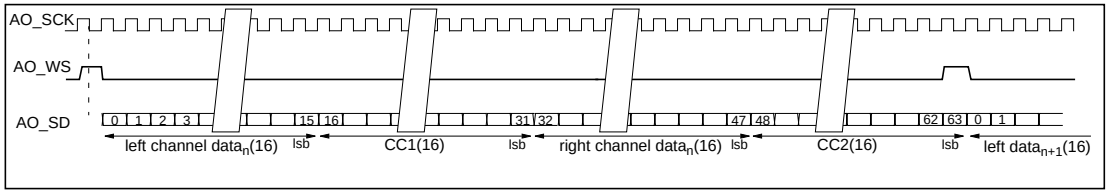


Figure 9-4. Example codec frame layout for a Crystal Semi CS4218.

Table 9-6. Audio Out MMIO Codec Control/Status fields

Field Name	Description
CC1 (16)	The 16 bits value of CC1 is shifted into each emitted serial frame starting at bit position CC1_POS, as long as CC1_EN is asserted.
CC1_POS	Defines the bit position within a serial frame where the first data bit of CC1 is placed. RESET Default 0.
CC1_EN	0 ⇒ CC1 emission disabled (RESET default) 1 ⇒ CC1 emission enabled.
CC2(16)	The 16 bits value of CC2 is shifted into each emitted serial frame starting at bit position CC2_POS, as long as CC2_EN is asserted.
CC2_POS	Defines the bit position within a serial frame where the first data bit of CC2 is placed. Default 0.
CC2_EN	0 ⇒ CC2 emission disabled (RESET default) 1 ⇒ CC2 emission enabled.
CC_BUSY	0 ⇒ Audio Out is ready to receive a CC1, CC2 pair (RESET default). 1 ⇒ Audio Out is not ready to receive a CC1, CC2 pair. Try again in a few SCK clock intervals.

9.6 MEMORY DATA FORMATS

The Audio Out unit autonomously reads samples from memory in mono and stereo eight- and 16-bit-per-sample formats, as shown in Figure 9-5. Successive samples are always read from increasing memory address locations. The setting of the LITTLE_ENDIAN bit in the AO_CTL register determines how increasing memory addresses map to byte positions within words. Refer to Appendix C, “Endian-ness,” for details on byte ordering conventions.

The Audio Out unit hardware implements a double buffering scheme to ensure that there are always samples available to transmit, even if the DSPCPU is highly loaded and slow to respond to interrupts. The DSPCPU software assigns buffers by writing a base address and size to the MMIO control fields described in Figure 9-6. Refer to Section 9.7, “Audio Out Operation,” for details on hardware/software synchronization.

In eight-bit transmit modes, data is MSB-aligned and extended with zeros before it is transmitted to the parallel to serial converter. If SIGN_CONVERT is set to one, the MSB of the data is inverted, which is equivalent to translating from offset binary representation to two’s complement. This allows the use of an external two’s complement 16-bit D/A converter to generate audio from eight-bit unsigned samples.

Note that the Audio Out hardware does *not* generate A-law or μ-law eight-bit data formats. If such formats are desired, the DSPCPU can be used to convert from A-law or μ-law data to 16 bit linear data.

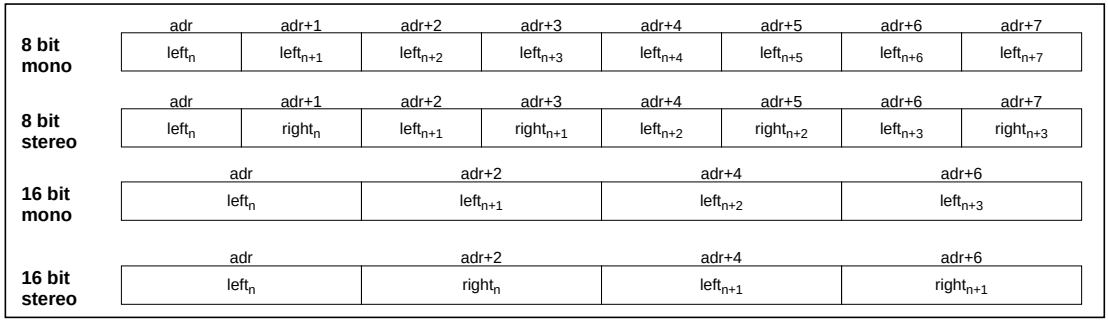


Figure 9-5. Audio Out memory DMA formats.

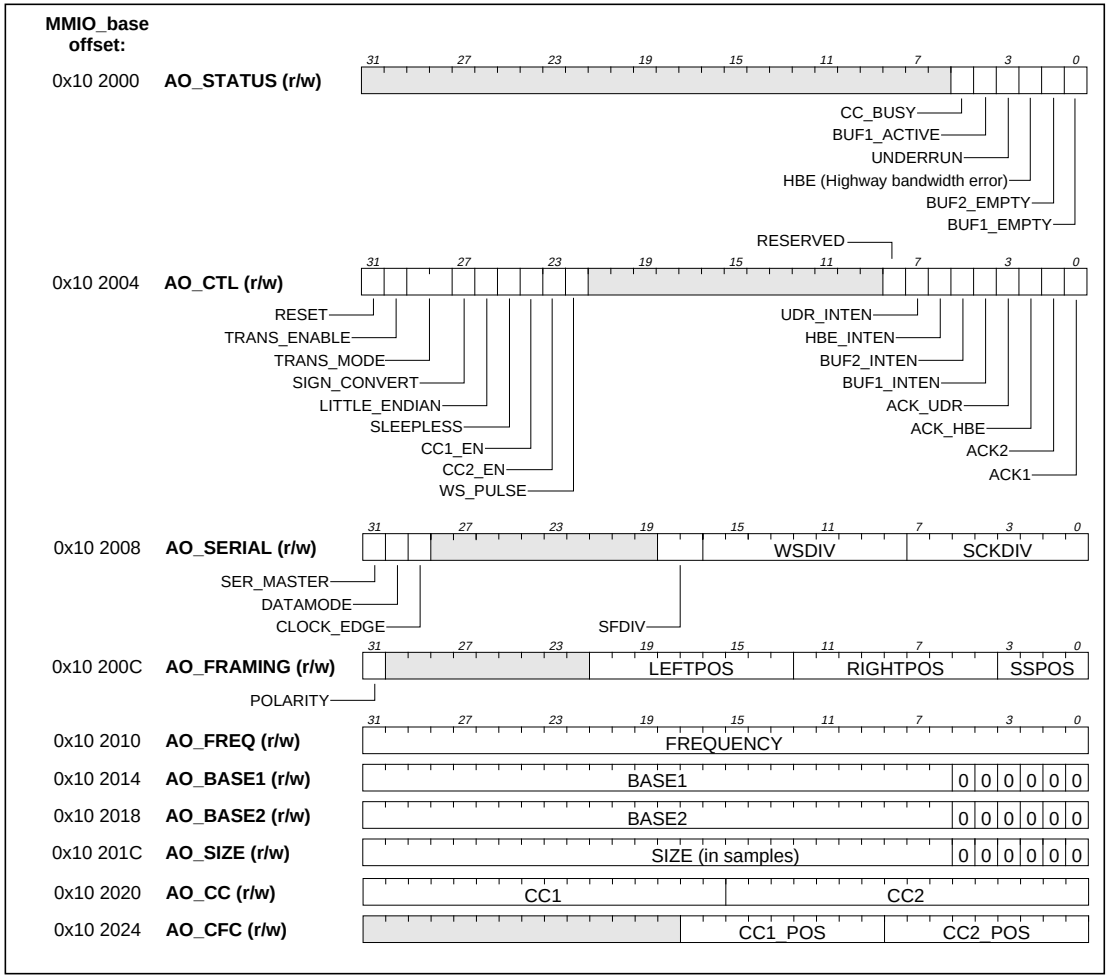


Figure 9-6. Audio Out status/control field MMIO layout.

9.7 AUDIO OUT OPERATION

Table 9-7 and Table 9-8 describe the function of the control and status fields of the Audio Out unit.

Table 9-7. Audio Out MMIO DMA control fields

Field Name	Description
LITTLE_ENDIAN	0 ⇒ big endian memory format (RESET default) 1 ⇒ little endian
BASE1	Base Address of buffer1. Must be a 64-byte aligned address in local SDRAM. RESET default 0.
BASE2	Base Address of buffer2. Must be a 64-byte aligned address in local SDRAM. RESET default 0.
SIZE	Number of samples to be read from a buffer before switching to other buffer. In stereo modes, a left/right pair of eight or 16 bit data counts as a single sample. RESET default 0.
TRANS_MODE	00 ⇒ mono, eight bits/sample. (RESET default). Left data and Right data are the same. 01 ⇒ stereo, two times eight bits/sample 10 ⇒ mono, 16 bits/sample. Left data and Right data are the same. 11 ⇒ stereo, two times 16 bits/sample
SIGN_CONVERT	0 ⇒ leave MSB unchanged (RESET default) 1 ⇒ invert MSB (not applied to codec control fields)

The Audio Out unit is reset by a TM1000 hardware reset, or by writing 0x80000000 to the AO_CTL register. Upon reset, transmission is disabled (TRANS_ENABLE = 0), and buffer1 is the active buffer (BUF1_ACTIVE=1). After a RESET, 5 AO_SCK clock cycles are required to stabilize the internal circuitry and before enabling Audio Out. This can be accomplished by programming the AO_FREQ and AO_SERIAL registers, and then waiting for the appropriate interval.

Table 9-8. Audio Out DMA Status Fields (Read Only)

Field Name	Description
BUF1_ACTIVE	• If 1, buffer 1 will be used for the next sample to be transmitted. • If 0, buffer 2 will contain the next sample (1 after RESET).
BUF1_EMPTY	• If 1, buffer 1 is empty. • If BUF1_INTEN is also 1, an interrupt request (source 12) is asserted. • BUF1_EMPTY is cleared by writing a ‘1’ to ACK1, at which point the Audio Out hardware will assume that BASE1 and SIZE describe a new full buffer. • 0 after RESET.

Table 9-8. Audio Out DMA Status Fields (Read Only)

Field Name	Description
BUF2_EMPTY	• If 1, buffer 2 is empty. • If BUF2_INTEN is also 1, an interrupt request (source 12) is asserted. • BUF2_EMPTY is cleared by writing a ‘1’ to ACK2, at which point the Audio Out hardware will assume that BASE2 and SIZE describe a new full buffer. • 0 after RESET.
HBE	• Highway Bandwidth Error. • 0 after RESET. • Indicates that no data was transmitted due to inability to read the local Audio Out buffer from SDRAM in time. This indicates an insufficient allocation of TM1000 Highway bandwidth for the audio sampling rate/mode.
UNDERRUN	• An UNDERRUN error has occurred, i.e. the CPU failed to provide a full buffer in time, and no samples were transmitted, although requested by the D/A converter. • If UDR_INTEN is also 1, an interrupt request (source 12) is pending. The UNDERRUN flag can ONLY be cleared by writing a ‘1’ to ACK_UDR. 0 after RESET.

The DSPCPU initiates transmission by providing two full equal size buffers and putting their base address and size in the $BASE_n$ and SIZE registers. Once two valid buffers are assigned, transmission can be enabled by writing a one to TRANS_ENABLE. The Audio Out unit hardware now proceeds to empty buffer 1 by transmission of output samples. Once buffer 1 empties, BUF1_EMPTY is asserted, and transmission continues without interruption from buffer 2. If BUF1INTEN is enabled, a SOURCE 12 interrupt request is generated.

Note that the buffers must be 64-byte aligned, and buffersizes must be a multiple of 64 samples (the six LSBs of AO_BASE1, AO_BASE2 and AO_SIZE are zero).

The DSPCPU is required to assign a new, full buffer to BASE1 and perform an ACK1, before buffer 2 empties. Transmission continues from buffer 2, until it is empty. At that time, BUF2_EMPTY is asserted, and transmission continues from the new buffer 1, etc. Upon receipt of an ACK, the Audio Out hardware removes the interrupt request line assertion at the next DSPCPU clock edge. Refer to the interrupt controller documentation for details on interrupt handler programming. The Audio Out interrupt (SOURCE 12) should always be operated in level sensitive mode.

Table 9-9. Audio Out MMIO Control Fields

Field Name	Description
RESET	Resets the audio-out logic. See Section 9.7, "Audio Out Operation" for a description of the recommended procedure.
TRANS_ENABLE	Transmission Enable flag. 0 ⇒ (RESET default) Audio Out inactive. 1 ⇒ Audio Out transmits samples and acts as DMA master to read samples from local SDRAM. Do NOT change the SER_MASTER and POLARITY bits while transmission is enabled.
SLEEPLESS	0 ⇒ (power up default) Audio Out goes into power saving mode if TM1000 goes to power saving mode. 1 ⇒ Audio out continues operation when TM1000 goes to sleep mode. Samples are read from memory as needed, and Audio Out interrupts, when enabled, will wake up the DSPCPU.
BUF1_INTEN	Buffer 1 Empty Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if buffer 1 empty
BUF2_INTEN	Buffer 2 Empty Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if buffer 2 empty
HBE_INTEN	HBE Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if a highway bandwidth error occurs.
UDR_INTEN	UNDERRUN Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if an UNDERRUN error occurs
ACK1	• Write a 1 to clear the BUF1_EMPTY flag and remove any pending BUF1_EMPTY interrupt request. • ACK1 always reads 0.
ACK2	• Write a 1 to clear the BUF2_EMPTYflag and remove any pending BUF2_EMPTY interrupt request. • ACK2 always reads 0.
ACK_HBE	• Write a 1 to clear the HBE flag and • remove any pending HBE interrupt request. • ACK_HBE always reads as 0.
ACK_UDR	• Write a 1 to clear the UNDERRUN flag and remove any pending UNDERRUN interrupt request. • ACK_UDR always reads 0.

9.8 HIGHWAY LATENCY AND HBE

The Audio Out unit uses an internal 64-byte buffer as well as a 32-bit output holding register. Under normal operation, the internal buffer gets refreshed from SDRAM fast enough to avoid any missing samples, while data is being emitted from the holding register. If the highway arbiter is set up with an insufficient latency guarantee, the situation can arise that the 64 byte buffer is not refilled and the holding register is exhausted by the time a new output sample is due. In that case the HBE error is raised. The last sample (or sample pair) will be repeated until the buffer is refreshed. The HBE condition is sticky, and can only be cleared by an explicit ACK_HBE.

Given a sample rate f_S, and an associated sample interval T (in DSPCPU clock cycles), the arbiter should be set to have a latency of at most T-2 cycles for stereo 16 bit mode, 2T-2 for mono 16 bit and stereo 8 bit modes and 4T-2 for mono 8 bit mode. Refer to Chapter 19, "Arbiter," for information on arbiter programming.

Table 9-10. Audio Out Highway Arbiter latency requirements (100 MHz)

TransMode	f_S	T	Max. latency (cycles)
stereo 16 bit/sample	44100 Hz	2267	2265
stereo 16 bit/sample	48000 Hz	2083	2081
stereo 16 bit/sample	96000 Hz	1041	1039

9.9 ERROR BEHAVIOR

In normal operation, the DSPCPU and Audio Out hardware continuously exchange buffers without ever failing to transmit a sample. If the DSPCPU fails to provide a new buffer in time, the UNDERRUN error flag is raised, and the last valid sample or sample pair is repeated until a new buffer of data is assigned by an ACK1 or ACK2. The UNDERRUN flag is *not affected* by ACK1 or ACK2; it can only be cleared by an explicit ACK_UDR.

If an HBE error occurs, the last valid sample or sample pair is repeated until the Audio Out hardware retrieves a new sample buffer across the highway.

9.10 4, 6 AND 8 CHANNEL AUDIO

The TM1000 Audio Out unit is capable of generating a bitstream with 4,6 or 8 channels of audio. This is currently only supported if Audio Out is operating as serial master (SER_MASTER=1). More than two channels of audio are accomplished by creating a superframe consisting of several serial frames. A superframe is created by dividing the internal signal used for parallel-to-serial conversion by 2, 3 or 4 and sending the result of the division as the AO_WS output value.

Modern stereo codecs, such as the CS4218 and AD1847, can easily be set to decode the first, second, third or fourth stereo stream from a superframe of 4 or 8 channels.

Figure 9-11 illustrates the logic that creates a superframe. If SFDIV is set to a value other than 0, a superframe of SFDIV+1 frames is generated. The divider hardware emits a WS edge at the start of each SDRAM buffer, and every superframe thereafter. By setting WS_PULSE=1, a single AO_SCK duration pulse is sent

every superframe. If WS_PULSE=0, AO_WS is a 50% duty cycle signal, except in the case of 6 channel operation, where the duty cycle is undefined.

Note that the software needs to ensure that a SDRAM buffer contains an integral number of superframes. For example, if SFDIV=2, superframes of 3 stereo streams are constructed, and each SDRAM buffer must contain a multiple of 6 16 bit samples. This ensures that the D/A converter set to the first stereo pair of the superframe always receives the first stereo pair from the SDRAM buffer, etc.

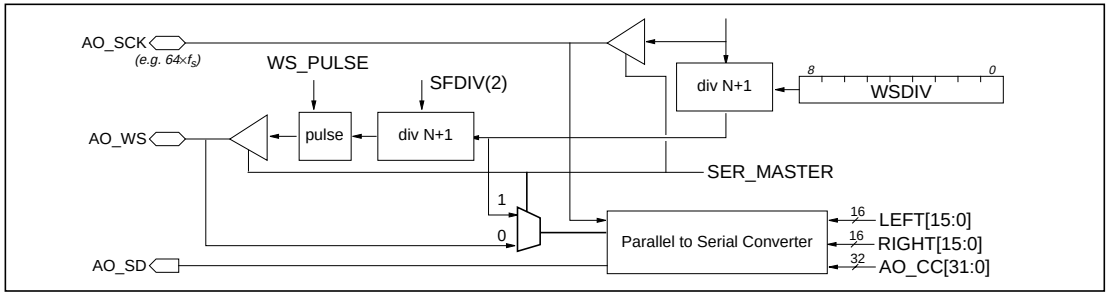


Figure 9-11. Super frame division block diagram

PCI Interface Chapter 10

by Gert Slavenburg, Ken-Sue Tan, Babu Kandimalla

10.1 PCI OVERVIEW

TM1000 includes a PCI interface for easy integration into personal computer applications—where the PCI-bus is the standard for high-speed peripherals. In embedded applications, with TM1000 serving as the main CPU, the PCI bus can interface to peripheral devices that implement functions not provided by the on-chip peripherals. See Figure 10-1.

The main function of the PCI interface is to connect the TM1000 on-chip highway and PCI buses. A bus cycle on the internal highway that targets an address mapped into PCI space will cause the PCI interface to create a PCI bus cycle. Similarly, a bus cycle on PCI that targets an address mapped into TM1000 memory space will cause the PCI interface to create a highway bus cycle targeted at SDRAM. For some operations, the PCI interface is explicitly programmed by the DSPCPU.

From TM1000, only the DSPCPU and the ICP (image co-processor) can cause the PCI interface to create PCI bus cycles; the other on-chip peripherals cannot see external hardware through the PCI interface. From PCI, only SDRAM and a subset of the registers in MMIO space can be accessed by external PCI initiators.

The PCI interface implements DMA (also called block or burst) and non-DMA transfers. DMA transfers are interruptible on 64-byte boundaries. The PCI interface can service outbound (TM1000 → PCI) and inbound (PCI → TM1000) data flows simultaneously.

Table 10-1 lists some of the features of the PCI interface.

Table 10-1. PCI Interface Characteristics

Characteristic	Comments
PCI Compliance	PCI Local Bus Specification Revision 2.1
PCI Speed	Up to 33 MHz
Data bus width	32-bit only
Address space	32 bits (4G bytes)
Voltage levels	Drive & receive at either 3.3V or 5V
Burst mode	Yes, w/ double buffering so maximum transfer rate (132 MB/s) is sustainable
Posted write	Yes, can be disabled
PCI 'special cycle'	Not recognized
PCI 'memory write & invalidate'	Supported for TM1000 as initiator
PCI 'interrupt acknowledge'	Not generated
PCI 'dual-address cycle'	Not generated

10.2 PCI INTERFACE AS AN INITIATOR

The following classes of operations invoked by TM1000 cause the PCI interface to act as a PCI initiator:

- Transparent, single-word (or smaller) transactions caused by DSPCPU loads and stores to the PCI address aperture.
- Explicitly programmed single-word I/O or configuration read or write transactions.
- Explicitly programmed multi-word DMA transactions.
- Image Co-Processor DMA

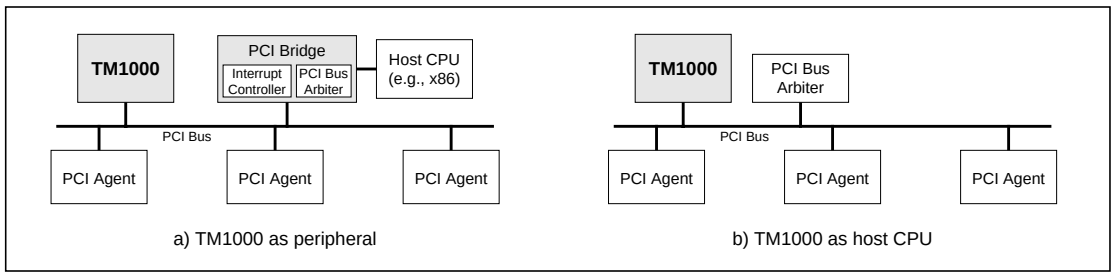


Figure 10-1. Two typical system implementations. (a) shows TM1000 as a PCI peripheral in a desktop PC. (b) shows an embedded system with TM1000 as the host CPU.

10.2.1 DSPCPU Single-Word Loads/Stores

From the point of view of programs executed by TM1000's DSPCPU, there are three apertures into TM1000's 4-GB memory address space:

- SDRAM space (0.5 to 64 MB in size; programmable).
- MMIO space (2 MB in size).
- PCI space.

MMIO registers control the positions and extents of the address-space apertures (see Chapter 3, "DSPCPU Architecture"). The SDRAM aperture begins at the address specified in the MMIO register DRAM_BASE and extends upward to the address in the DRAM_LIMIT register. The 2-MB MMIO aperture begins at the address in MMIO_BASE (defaults to 0xEFE00000 after power-up). All addresses that fall outside these two apertures are assumed to be part of the PCI address aperture. References by DSPCPU loads and stores to the PCI aperture are reflected to external PCI devices by the coordinated action of the data cache and PCI interface.

When a DSPCPU load or store targets the PCI aperture (i.e., neither of the other two apertures), the DSPCPU's data cache automatically carries out a special sequence of events. The data cache writes to the PCI_ADR and (if the DSPCPU operation was a store) PCI_DATA registers in the PCI interface and asserts (load) or deasserts (store) the internal signal pci_read_operation (a direct connection from the data cache to the PCI interface).

While the PCI interface executes the PCI bus transaction, the DSPCPU is held in the stall state by the data cache. When the PCI interface has completed the transaction, it asserts the internal signal pci_ready (a direct connection from the PCI interface to the data cache).

When pci_ready is asserted, the data cache finishes the original DSPCPU operation by reading data from the PCI_DATA register (if the DSPCPU operation was a load) and releasing the DSPCPU from the stall state.

Explicit Writes to PCI_ADR, PCI_DATA

The PCI_ADR and PCI_DATA registers are intended to be used only by the data cache. Explicit writes are not allowed and may cause undetermined results and/or data corruption.

10.2.2 I/O Operations

Explicit programming by DSPCPU software is the only way to perform transactions to PCI I/O space. DSPCPU software writes three MMIO registers in the following sequence:

1. The IO_ADR register.
2. The IO_DATA register (if PCI operation is a write).
3. The IO_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus I/O transaction when software writes to IO_CTL. The interface can raise a DSPCPU interrupt at the completion of the I/O transaction (see BIU_CTL register definition in Section 10.6.4, "BIU_CTL Register") or the DSPCPU can poll the appropriate status bit (see BIU_STATUS register definition in Section 10.6.3, "BIU_STATUS Register"). Note that PCI I/O transactions should NOT be initiated if a PCI configuration transaction described below is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in Section 10.6.12, "IO_CTL Register."

10.2.3 Configuration Operations

As with I/O operations, explicit programming by DSPCPU software is the only way to perform transactions to PCI configuration space. DSPCPU software writes three MMIO registers in the following sequence:

1. The CONFIG_ADR register.
2. The CONFIG_DATA register (if PCI operation is a write).
3. The CONFIG_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus configuration transaction when software writes to CONFIG_CTL. As with the I/O operations, the biu_status and BIU_CTL registers monitor the status of the operation and control interrupt signalling.Note that PCI configuration space transactions should NOT be initiated if a PCI I/O transaction described above is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in Section 10.6.9, "CONFIG_CTL Register."

10.2.4 DMA Operations

The PCI interface can operate as an autonomous DMA engine, executing block-transfer operations at maximum PCI bandwidth. As with I/O and configuration operations, DSPCPU software explicitly programs DMA operations.

General-purpose DMA

For DMA between SDRAM and PCI, DSPCPU software writes three MMIO registers in the following sequence:

1. The SRC_ADR and DEST_ADR registers.
2. The DMA_CTL register (controls direction of data movement and amount of data transferred).

The PCI interface begins the PCI-bus transactions when software writes to DMA_CTL. As with the I/O and configuration operations, the BIU_STATUS and BIU_CTL registers monitor the status of the operation and control interrupt signalling.

The fully detailed description of the steps needed can be found in Section 10.6.15, "DMA_CTL Register."

Image-Coprocessor DMA

The PCI interface also executes DMA transactions for the Image Coprocessor (ICP). The ICP performs rapid post-processing of image data and writes it at PCI DMA speed to a PCI graphics card frame buffer. The ICP cannot perform PCI read transactions. BIU_CTL.IE (ICP DMA Enable) should be asserted before attempting ICP

PCI operation. Programming of ICP DMA is described in Section 13.6, “Operation and Programming.”

10.3 PCI INTERFACE AS A TARGET

The TM1000 PCI interface responds as a target to external initiators for a limited set of PCI transaction types:

- Configuration read/write
- Memory read/write, read line and read multiple to the TM1000 SDRAM or MMIO apertures. See Section 10.8, “Limitations.”

TM1000 ignores PCI transactions other than the above.

10.4 TRANSACTION CONCURRENCY, PRIORITIES, AND ORDERING

The PCI interface can be processing more than one operation at a given time. There are six distinct classes of operations implemented by the PCI interface:

1. DSPCPU load/store to PCI space.
2. PCI I/O read/write, PCI configuration read/write.
3. General-purpose DMA read/write.
4. ICP DMA write.
5. External-PCI-agent-initiated read/write (to TM1000 on-chip resource).

If the active general-purpose DMA transaction is a read, up to five transactions, one from each, can be active simultaneously. If the active general-purpose DMA operation is a write, then only four transactions can be active simultaneously because general-purpose DMA writes force ICP DMA writes to wait until the general-purpose DMA completes. When a general-purpose DMA write is pending, an in-progress ICP DMA operation is suspended at the next 64-byte block boundary and waits until the completion of the DMA write operation. General-purpose DMA reads are interleaved with ICP DMA writes, so both can be active concurrently.

PCI single-data-phase transactions (DSPCPU load/store, I/O read/write, and configuration read/write) are executed in the order they are issued to the PCI interface. Note the strict implementation limitation that PCI I/O and PCI configuration transactions cannot be simultaneously active.

10.5 REGISTERS ADDRESSED IN PCI CONFIGURATION SPACE

Since it is a PCI device, TM1000 has a set of configuration registers to determine PCI behavior. PCI configuration registers allow full relocation of interrupt binding and address mapping by the system’s host processor. This relocatability of PCI-space parameters eases installation, configuration, and system boot.

The PCI standard specifies a 64-byte PCI configuration header region within a reserved 256-byte block. During system initialization, host system software scans the PCI bus, looking for PCI headers, to determine what PCI devices are present in the system. The fields in the header region uniquely identify the PCI device and allow the host to control the device in a generic way. Figure 10-2 shows the layout of the configuration header region.

Figure 10-2 also shows the initial values for the configuration registers. Some registers, such as Device ID, have hardwired values, while others are programmed by software. Still others are set automatically from the external boot ROM during TM1000’s power-up initialization.

10.5.1 Vendor ID Register

For TM1000, the value of the 16-bit Vendor ID field is hardwired to 0x1131 (Philips). This value identifies the manufacturer of a PCI device. Valid vendor identifiers are assigned by the PCI special interest group (PCI SIG) to assure uniqueness. The value 0xFFFF is reserved and must be returned by the host/PCI bridge when an attempt is made to read a non-existent device’s Vendor ID configuration register.

10.5.2 Device ID Register

For TM1000, the value of the 16-bit Device ID field is hardwired to 0x5400. The Device ID is assigned by the manufacturer to uniquely identify each PCI device it makes.

10.5.3 Command Register

The 16-bit command register provides basic control over a PCI device’s ability to generate and/or respond to PCI bus cycles. According to the PCI specification, after reset, all bits in this register are cleared to zero (except for a device that must be initially enabled). Clearing all bits to zero logically disconnects the device from the PCI bus for all accesses except configuration accesses.

The command register format is shown in Figure 10-3. Table 10-2 summarizes the field values. Following are detailed descriptions of the command register fields.

I/O (I/O access enable). This bit controls a device’s ability to respond to I/O-space accesses. A value of zero disables PCI device response; a value of one enables response. This bit is hardwired to zero because all TM1000 internal registers are memory mapped.

MA (Memory Access enable). This bit controls response to memory-space accesses. A value of zero disables TM1000 response; a value of one enables response. This bit is set to zero at power-up; software can set this bit to one with a configuration write.

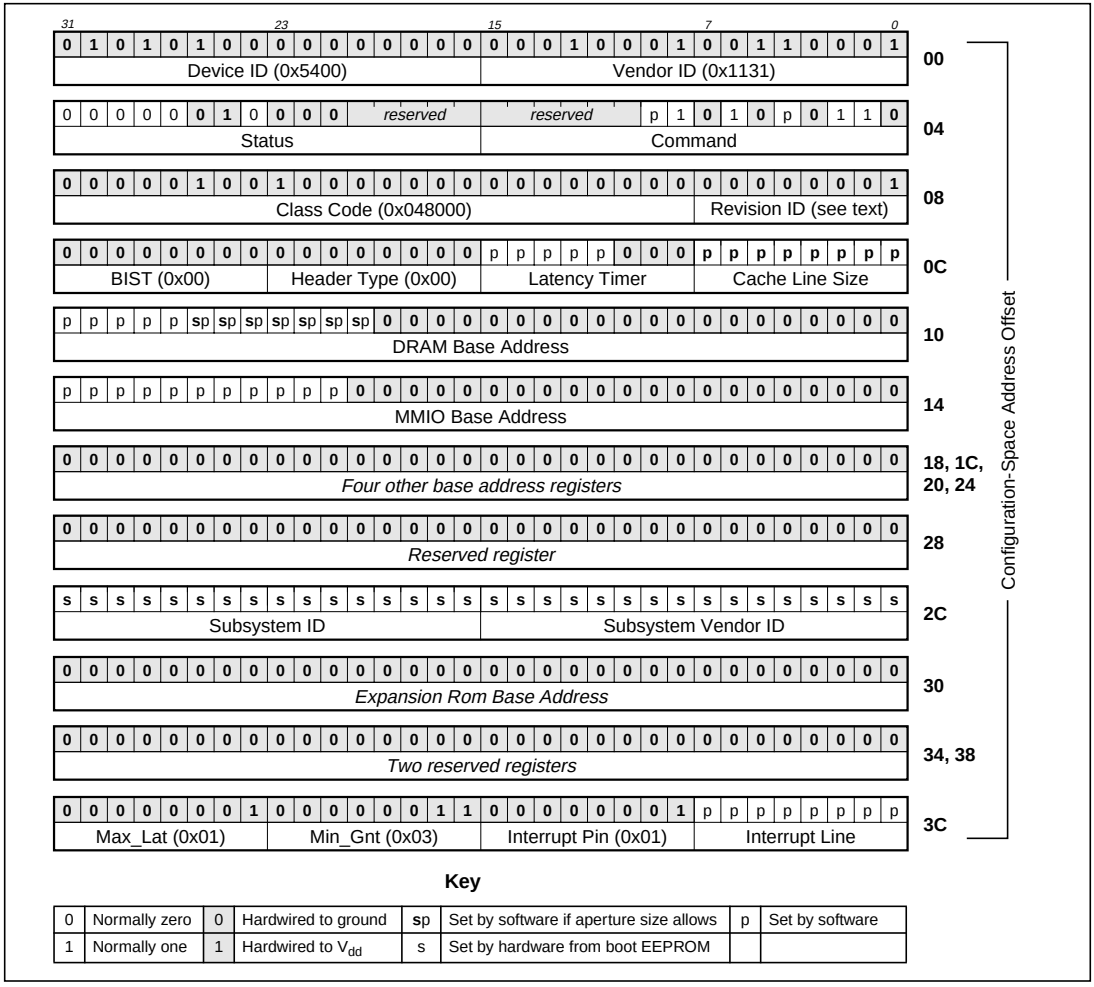


Figure 10-2. PCI configuration header region register layout, initial values and programming. (All values in

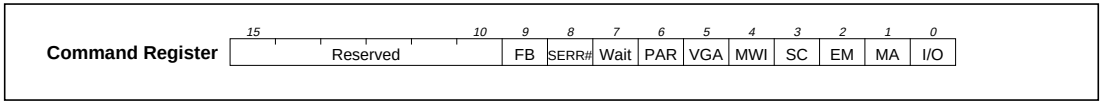


Figure 10-3. Command Register format.

Table 10-2. Field values for Command Register

Field	Value Explanation
I/O	Hardwired to 0 (ignore I/O space accesses)
MA	0 ⇒ no recognition of memory-space accesses 1 ⇒ recognizes memory-space accesses
EM	0 ⇒ cannot act as PCI initiator 1 ⇒ can act as PCI initiator
SC	Hardwired to 0 (ignore special cycle accesses)
MWI	0 ⇒ cannot generate memory write and invalidate 1 ⇒ can generate memory write and invalidate
VGA	Hardwired to 0
Par	0 ⇒ ignore parity errors 1 ⇒ acknowledge parity errors
SERR#	0 ⇒ disable driver for serr# pin 1 ⇒ enable driver for serr# pin
FB	0 ⇒ fast back-to-back only to same agent 1 ⇒ fast back-to-back to different agents
Reserved	Write ignored; reads return 0

EM (Enable Mastering). This bit controls the TM1000 PCI interface's ability to act as a PCI master. A value of zero prevents the PCI interface from initiating PCI accesses; a value of one allows the PCI interface to initiate PCI accesses.

Note that the EM bit is automatically set to one whenever the HE bit in the BIU_CTL register is set to one (see Section 10.6.4, "BIU_CTL Register"). Mastering must be enabled for TM1000 to serve as PCI host processor.

EM is set to zero at power-up. Host system software can set this bit to one with a configuration write.

SC (Special Cycle). This bit controls PCI device recognition of special-cycle operations. A value of zero causes a PCI device to ignore all special cycles; a value of one allows a PCI device to monitor special cycle operations. This bit is hardwired to zero in TM1000.

MWI (Memory Write and Invalidate). This bit determines a PCI devices's ability to generate memory-write-and-invalidate commands. A value of one allows a PCI device to generate memory-write-and-invalidate commands; a value of zero forces the PCI device to use memory-write commands instead. TM1000 implements this bit. The conditions under which TM1000 DMA transactions generate memory-write-and-invalidate are described in Section 10.6.15, "DMA_CTL Register." Image Coprocessor DMA writes always use regular memory-write transactions.

VGA (VGA palette snoop). This bit controls how VGA-compatible PCI devices handle accesses to their palette registers. This bit is hardwired to zero.

PAR (Parity error response). This bit controls signalling of parity errors (data or address). A value of zero causes the PCI interface to ignore parity errors; a value of one causes the PCI interface to report parity errors on the perr# PCI signal. This bit is set to zero at power-up; since the PCI interface checks parity, software can set this bit to one with a configuration write.

Wait (Wait-cycle control). This bit controls whether or not a PCI device does address/data stepping. PCI devices that never do stepping must hardwire this bit to 0. Since TM1000 does not implement stepping, this bit is hardwired to zero.

SERR# (serr# enable). This bit is an enable for the driver of the serr# pin (system error). A value of zero disables the serr# pin; a value of one enables it. All PCI devices that have an serr# pin must implement this bit. This bit is set to zero after reset; this bit can be set to one with a configuration write. SERR# and PAR must both be set to one to allow signalling of address parity errors on the serr# signal.

FB (Fast Back-to-back enable). This bit controls whether or not a PCI master can do fast back-to-back transactions to different devices. A value of zero means fast back-to-back transactions are only allowed when the transactions are to the same agent; a value of one means the master is allowed to generate fast back-to-back transactions to different agents. Initialization software will set this bit if all targets are capable of fast back-to-back transactions.

Reserved. Reads from reserved bits return zero; writes to reserved bits cause no action.

10.5.4 Status Register

The status register is used to record information about PCI bus events. The status register format is shown in Figure 10-4. Table 10-3 lists the Status register fields.

Reserved. Reads from reserved bits return zero; writes to reserved bits cause no action.

66M (66-MHz capable). This bit is hardwired to zero for TM1000 (PCI runs at 33-MHz maximum).

UDF (user Definable Features). Since the TM1000 PCI interface does not implement PCI user-definable features, this bit is hardwired to zero.

FBC (Fast Back-to-back Capable). The TM1000 PCI interface does not support fast back-to-back capability, so this bit is hardwired to zero.

DPD (Data Parity Detected). Since the TM1000 PCI interface can act as a PCI bus initiator, this bit is implemented. DPD is set in the initiator's status register when:

- The PAR (parity-error response) bit in the command register is set, and

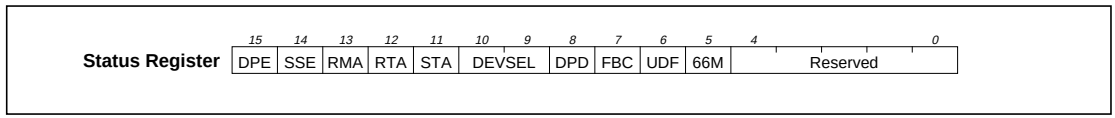


Figure 10-4. Status register format.

- The initiator asserted perr# or detected it asserted by the target (during a write cycle).

Table 10-3. Status Register Fields

Field	Characteristics
Reserved	Writes ignored; reads return 0
66M	PCI bus speed (hardwired to 0 ⇒ 33-MHz)
UDF	User-definable features (hardwired to 0 ⇒ none)
FBC	Fast back-to-back capable (hardwired to 0 ⇒ unsupported)
DPD	Data parity detected
DEVSEL	devsel# signal timing (hardwired to 1 ⇒ 'medium')
STA	Signaled target abort
RTA	Receive target abort
RMA	Receive master abort
SSE	Signaled system error
DPE	Detected parity error

DEVSEL (Device Select timing). This read-only field defines the slowest timing that will be used for the devsel# signal when TM1000 is a target on the PCI bus. Table 10-4 shows the allowable encodings and meanings. These bits are hardwired to '01' to indicate that

Table 10-4. DEVSEL Encodings

DEVSEL	Meaning
00	Fast
01	Medium
10	Slow
11	Reserved

TM1000 uses a 'medium' devsel# timing.

STA (Signalled Target Abort). TM1000's PCI interface sets this bit when it is a target device and aborts a transaction.

RTA (Receive Target Abort). TM1000's PCI interface sets this bit when it is the initiating device and the transaction is aborted by the target device. (All initiating devices must implement this bit.)

RMA (Receive Master Abort). TM1000's PCI interface sets this bit when it is the initiating device and aborts a transaction (except when the transaction is a special cycle). (All initiating devices must implement this bit.)

SSE (Signaled System Error). TM1000's PCI interface sets this bit when it asserts the serr# signal. (TM1000 can generate serr#, so this bit is implemented; devices incapable of generating serr# need not implement SSE.)

DPE (Detected Parity Error). TM1000's PCI interface sets this bit when it detects a parity error, even if parity error handling is disabled. (The PAR bit in the command register enables the handling of parity errors.)

10.5.5 Revision ID Register

The value in the Revision ID register is a read only value chosen by the manufacturer to indicate product revisions. For the TM1000 product family, the two MSB's of the revision ID indicate the fab. The next two bits indicate an all layer revision number, and the 4 lsb's indicate metal layer changes. Each future all layer revision adds 0x10 to the revision ID and resets the 4 lsb's to zero. Non pin or function compatible Trimedia devices will use a revised Device ID.

Table 10-5. Revision Id values

Value (hex)	Product description
00	CTC (CPU Test Chip) - all versions
01	Crolles fab TM1000 0.50 µ original mask version as well as first metal revision
10	Crolles fab TM1000 0.35 µ original mask version

10.5.6 Class Code Register

The value in the Class Code register is read-only. System software uses the Class Code register to identify the generic function of the device, and in some cases, the Class Code can specify a register-level programming interface.

Class Code consists of three one-byte fields as shown in Figure 10-5. The value of the upper byte, Base Class Code, broadly classifies the function of the device. The value of the middle byte, Subclass Code, identifies the function more specifically. The value of the lower byte specifies a register-level programming interface so that device-independent software can interact with the device. The meanings of the Base Class byte values are shown in Table 10-6.

The value of Base Class is hardwired to 0x04 since TM1000 is a multimedia device. Currently, there are no specific register-level programming interfaces defined for multimedia devices.

Table 10-7 lists the defined subclasses of multimedia devices. TM1000 is both a video and audio multimedia device, so its subclass value is hardwired to 0x80.

10.5.7 Cache Line Size Register

The value of the Cache Line Size register specifies the system cache line size in units of 32-bit words Only initi-

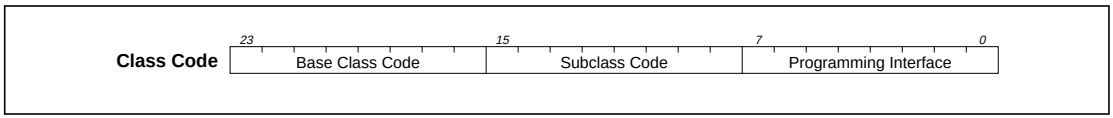


Figure 10-5. Class-code register format.

Table 10-6. Base Class Encodings

Base Class (in hex)	Meaning
00	Device was built before class code definitions were finalized
01	Mass-storage controller
02	Network controller
03	Display controller
04	Multimedia device
05	Memory controller
06	Bridge device
07	Simple communications controller
08	Base system peripheral
0A	Docking station
0B	Processor
0C	Serial bus controller
0D–FE	Reserved
FF	Device does not fit any of the above classes

Table 10-7. Subclass & Programming Interface Fields

Subclass (in hex)	Programming Interface (in hex)	Meaning
00	00	Video Device
01	00	Audio Device
80	00	Other multimedia device

ating devices that can generate memory-write-and-invalidate commands must implement this register. When implemented, the cache line size allows initiators participating in the PCI caching protocol to retry burst accesses at cache-line boundaries.

This register is implemented in TM1000.

10.5.8 Latency Timer Register

The value of the Latency Timer register specifies the minimum number of PCI clock cycles the TM1000 BIU as initiator is allowed to own the PCI bus. This register is readable and writable in PCI configuration space.

This register must be writable in any PCI initiating device that can burst more than two data phases. In the TM1000 PCI interface, the least-significant three bits are hardwired to zero and software can program any value into the most-significant five bits. This permits software to specify the time slice with a minimum granularity of eight PCI clocks. A value of zero signifies maximum latency, i.e. 256 PCI clocks.

10.5.9 Header Type Register

The value of the Header Type register defines the format of words 16 through 63 in configuration space and whether or not the device contains multiple functions. Figure 10-6 shows the format of Header Type.

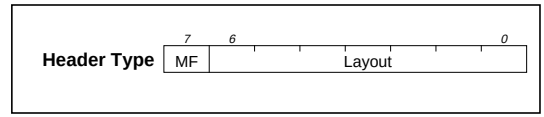


Figure 10-6. Header Type register format.

Bit 7 of Header Type is zero for single-function devices, one for multi-function devices. TM1000 is a single-function device, so bit 7 is zero. Table 10-8 shows the encodings of the Layout field.

10.5.10 Built-In Self Test Register

When implemented, the BIST register is used to control the operation of a device's built-in self testing capability. TM1000 does not implement BIST, so this register is hardwired to return zeros when read.

10.5.11 Base Address Registers

The TM1000 PCI interface implements two memory Base Address registers: DRAM_BASE and MMIO_BASE. DRAM_BASE relocates TM1000's SDRAM within the system address space; MMIO_BASE relocates the 2-MB memory-mapped I/O address aperture.

The values in the Base Address registers determine the address map as seen by both the DSPCPU and external PCI masters. These values are normally set once, and not changed dynamically once the DSPCPU operates.

Hardware RESET initializes DRAM_BASE to 0x0 and MMIO_BASE to 0xefe0,0000, after which the TM1000 boot protocol sets the final value..

In stand-alone systems, the autonomous boot sequence is executed., In this case, the values of SDRAM_BASE and MMIO_BASE are copied from the content of the serial boot EEPROM, as described in Section 12.2.2, "Initial DSPCPU Program Load for Autonomous Bootstrap."

In X86 or other host assisted platforms, the PCI host assisted boot sequence is executed. In this case, the base registers are not set from the EEPROM. Instead, the host BIOS executes a scan for devices on each PCI bus. During this scan, memory apertures needed by each device are determined, and a suitable base is assigned by the host BIOS. The details of this process are described below.

Figure 10-7 shows the formats for DRAM_BASE and MMIO_BASE. Following are descriptions of the register fields.

M (Memory). The value of the M bit indicates whether the desired resource is a memory or PC I/O aperture. The M bit is hardwired to zero, indicating a memory type aperture for both the DRAM_BASE and MMIO_BASE registers.

T (Type). The value of the T field indicates the size of the base address register and constraints on its relocatability. Table 10-9 lists the encodings and meanings of the T field.

Table 10-8. Layout Encodings

Layout (in hex)	Meaning
00	Non-bridge PCI device
01	PCI-to-PCI bridge device

Table 10-9. Type Field Encodings

Type	Meaning
00	Base register is 32 bits wide; mapping can relocate anywhere in 32-bit memory space
01	Base register is 32 bits wide; mapping must relocate below 1MB in memory space
10	Base register is 64 bits wide; mapping can relocate anywhere in 64-bit address space
11	Reserved

TM1000's PCI-interface base registers are 32 bits wide and can be relocated in the 32 bit address space; thus, the value of the T field is 00 for both DRAM_BASE and MMIO_BASE.

P (Prefetchable). The value of the P bit indicates to other devices whether or not prefetching is allowed. Both SDRAM and MMIO are not prefetchable, so the P bit is hardwired to zero for both DRAM_BASE and MMIO_BASE.

(A Base Address register has a P bit set to one if there are no side effects caused by reads. Reads from a prefetchable space return all bytes regardless of byte enables. Host bridges can merge writes to a prefetchable device without causing errors.)

DRAM/MMIO Base Address. The DRAM Base Address and MMIO Base Address fields serve two purposes. First, the host BIOS software can use them to determine the sizes of the SDRAM and MMIO apertures. Second, the BIOS can write to these fields to cause the apertures to be relocated within the PCI memory address space.

To determine the sizes of an aperture, the BIOS first writes all ones (0xFFFFFFFF) to the address field. When the BIOS reads the field immediately after, the value returned will have zeros in all don't-care bits and ones in all required address bits. Required address bits form a left-aligned (i.e., starting at the MSB) contiguous field of ones, thus effectively specifying the size of the aperture.

For example, the MMIO aperture is a fixed 2-MB space. After writing all ones to the MMIO Base Address field, a subsequent read returns the value 0xFFE00000. The M, T, and P fields are all zero indicating the aperture is memory (not I/O), can be relocated anywhere in a 32-bit address space, and is not prefetchable. Since the aperture has 21 address bits (the position of the first one bit), MMIO space is a 2-MB aperture (2^{21} bytes). The host BIOS now assigns a suitable 2 MB aligned base address by writing to the MMIO_BASE register in configuration space.

The DRAM aperture can range in size from 1 MB to 64 MB (but the size must be a power of two). Thus, the number of required address bits can range from 20 to 26. The actual amount of SDRAM present is determined by the content of the first byte of the boot EEPROM, as described in Section 12.4, "Detailed EEPROM Contents." The PCI BIU uses this size to determine which of the bits marked 'sp' in Figure 10-7 are writable and which are set to 0. This causes the BIOS to determine the correct actual DRAM aperture size.

10.5.12 Subsystem ID, Subsystem Vendor ID Register

The subsystem and subsystem vendor ID are new per PCI Rev 2.1. These fields are optional, but their use is highly recommended as a means to have software drivers identify the board rather than the chip on the board.

This register is implemented starting with TM1000 and onwards, and replaces the 'Personality' register functionality in the Trimedia CTC chip.

The board manufacturer chooses the values of both 16 bits fields by modifying the TM1000 Boot EEPROM. The location of these bits is described in. A legal Vendor ID must be obtained from the PCI SIG. The vendor is free to assign subsystem ID's.

10.5.13 Expansion ROM Base Address Register

The Expansion ROM Base Address register is similar in purpose to the SDRAM and MMIO Base Address registers. This register relocates a separate memory aperture for PCI devices that wish to implement additional ROM.

TM1000 does not implement expansion ROM; consequently, the least-significant bit of this register—which indicates whether or not TM1000 responds to expansion ROM accesses—is hardwired to zero. All other bits also read as zeros.

10.5.14 Interrupt Line Register

The value of the Interrupt Line Register determines which input of the system interrupt controller is driven by TM1000's interrupt pin. As it configures the system and assigns resources, host system software writes this reg-

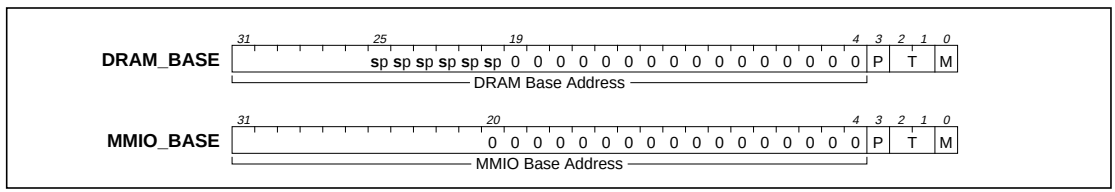


Figure 10-7. Base Address register format.

ister to assign one of the system interrupt lines to TM1000.

10.5.15 Interrupt Pin Register

The value of the Interrupt Pin Register determines which interrupt pin TM1000 uses. Table 10-10 lists the possible values for this register.

Table 10-10. Interrupt Pin Encodings

Interrupt Pin	Meaning
1	Use interrupt pin inta#
2	Use interrupt pin intb#
3	Use interrupt pin intc#
4	Use interrupt pin intd#
all others	Reserved

Since TM1000 uses inta#, the value of this register is hardwired to 1.

10.5.16 Max_Lat, Min_Gnt Registers

The value in the Max_Lat register specifies how often the TM1000 PCI interface needs access to the PCI bus. The value in the Min_Gnt register specifies the minimum length for a burst period on the PCI bus.

Both of these timer values are specified as multiples of 250 ns. Values of zero indicate that a device has no specific requirements for latency and burst-length.

For TM1000, Max_Lat is hardwired to 0x01 (250 ns), and Min_Gnt is hardwired to 0x03 (750 ns).

10.6 REGISTERS IN MMIO SPACE

The TM1000 PCI interface contains 13 MMIO registers; most, except the status bits in BIU_Status, are usually written only by the DSPCPU. Table 10-11 lists the internal cycles sequenced by the PCI interface and the registers each involves.

The MMIO registers are all accessible to DSPCPU software, and all but the PCI_ADR and PCI_DATA registers are accessible to external PCI initiators. The facilities of TM1000's PCI interface can be useful to external initiators in certain circumstances; for example:

- The PCI DMA engine might be useful during host-assisted boot.
- Host-resident diagnostics may want to test the PCI interface during boot.
- The MMIO registers can be used to diagnose malfunctioning parts.

Note, however, that external PCI initiators can access MMIO registers in only one way: as 32-bit words on naturally aligned, 32-bit addresses. If any other type of access is attempted, the results are undefined. Also, the byte order of the external initiator and the PCI interface must be the same; otherwise, the result of an access with disagreeing byte order is undefined.

For easy reference, Table 10-12 lists the MMIO registers together with their offsets from MMIO_BASE and their accessibility by the DSPCPU and external PCI initiators.

Figure 10-8 shows the formats of the PCI interface MMIO registers. Following are detailed descriptions of the MMIO registers.

10.6.1 DRAM_BASE Register

The DRAM_BASE register in MMIO space is a shadow copy of the DRAM_BASE register in PCI Configuration space. See Section 10.5.11, "Base Address Registers," for more details. This shadow copy provides MMIO-space access to this register.

10.6.2 MMIO_BASE Register

The MMIO_BASE register in MMIO space is a shadow copy of the MMIO_BASE register in PCI Configuration space. See Section 10.5.11, "Base Address Registers," for more details. This shadow copy provides MMIO-space access to this register.

10.6.3 BIU_STATUS Register

The BIU_Status register holds bits that track the status of bus cycles initiated by the DSPCPU and bus cycles from external devices that write into SDRAM.Two bits of status are provided for each type of bus cycle: a busy bit and a done bit. The DSPCPU can read both bits; a done bit is cleared by writing a one. The status register also holds two error-flag bits.

DSPCPU software must check the busy bits to avoid issuing a PCI interface bus cycle request while a request of a similar type is in progress. If a bus cycle is issued while a request of similar type is in progress, the PCI interface ignores the second command and sets the appropriate error bit in the status register.

When the DSPCPU issues either an io_cycle or config_cycle request while a previous request of either type is already in progress, the PCI interface sets bit eight in BIU_Status. When the DSPCPU issues a dma_cycle while a previous one is already in progress, the PCI interface sets bit nine in BIU_Status.

RTA (Received Target Abort). This bit gets set when TM1000 initiated a transaction that was aborted by the target. To reset this bit, write a '1' to this bit position. This bit is set simultaneous with the RTA bit in the configuration space status register, but gets cleared independently.

RMA (Received Master Abort). This bit gets set when TM1000 initiated a transaction and aborts it. This usually signals a transaction to a non existent device. To reset this bit, write a '1' to this bit position. This bit is set simultaneous with the RMA bit in the configuration space status register, but gets cleared independently.

TTE (Target Timer Expired). In normal operation, a read of a TM1000 data item is performed on retry basis - TM1000 tells the external master to retry, and meanwhile it fetches the data item across the highway. This bit gets set if an external master did not retry a read of a TM1000

data item within 32768 PCI clocks. The requested data is discarded. To reset this bit, write a '1' to this bit position. This is purely a software information bit. No software action is required when this condition occurs, but it may indicate a non-compliant or defective master on the bus.

10.6.4 BIU_CTL Register

The BIU_CTL register contains bits that control miscellaneous aspects of the PCI interface operation. Following are descriptions of the fields.

Table 10-11. PCI MMIO Registers and Bus Cycles

Internal Cycle	Registers Involved
mmio_cycle (MMIO register R/W)	All registers accessible by external PCI devices
mem_cycle (PCI-space memory R/W)	PCI_ADR, PCI_DATA
dma_cycle (Block data transfer)	SRC_ADR, DEST_ADR, DMA_CTL

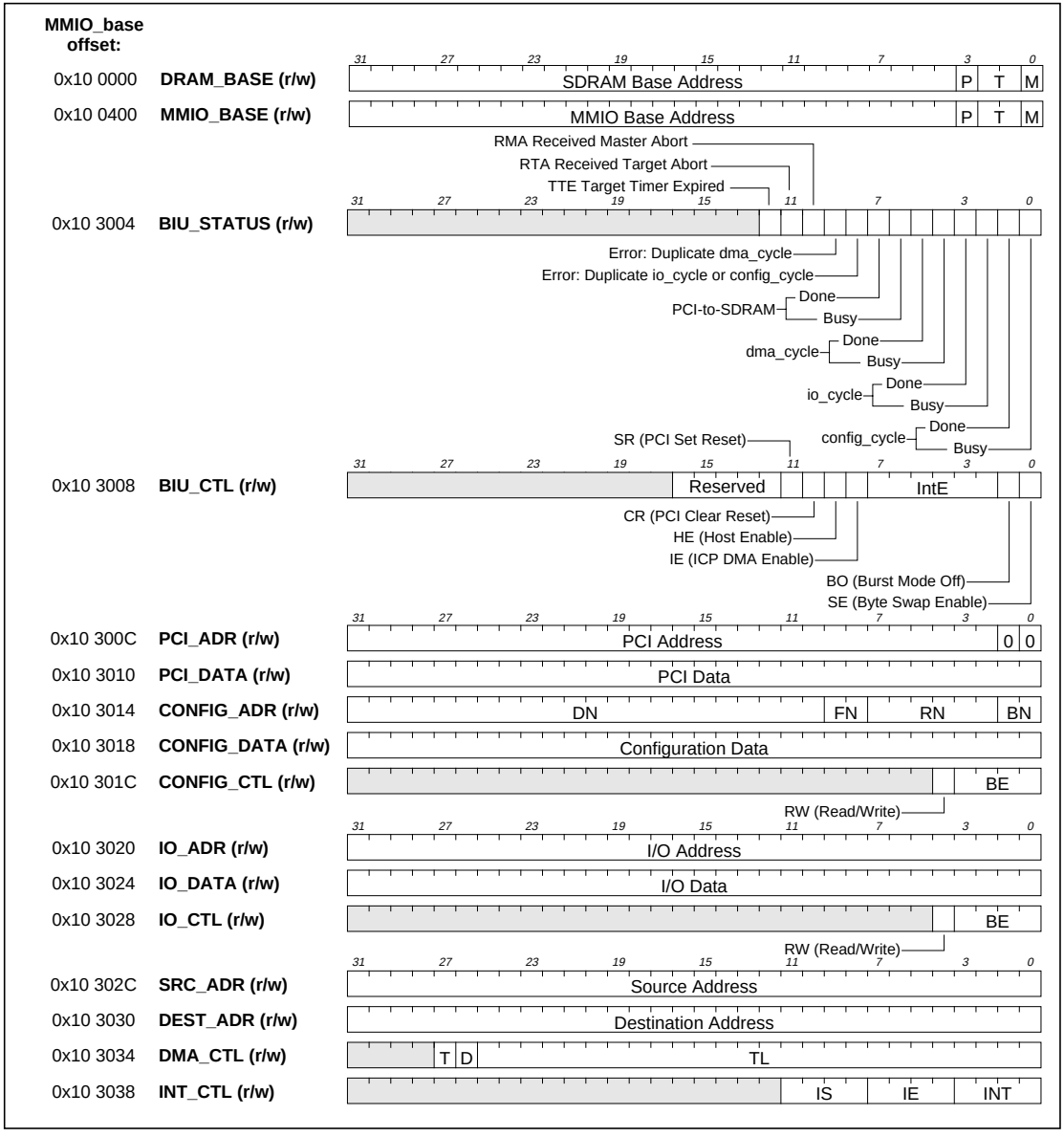


Figure 10-8. PCI interface registers accessible in MMIO address space.

Table 10-11. PCI MMIO Registers and Bus Cycles

Internal Cycle	Registers Involved
IO_cycle (I/O register R/W)	IO_ADR, IO_DATA, IO_CTL
config_cycle (Configuration register R/W)	CONFIG_ADR, CONFIG_DATA, CONFIG_CTL

Table 10-12. PCI MMIO Register Accessibility

Register	MMIO_BASE Offset	Accessibility	
		DSPCPU	External Initiator
DRAM_BASE	0x10 0000	R/W	R/W
MMIO_BASE	0x10 0400	R/W	R/W
BIU_STATUS	0x10 3004	R/W	R/W
BIU_CTL	0x10 3008	R/W	R/W
PCI_ADR	0x10 300C	R/W	–/–
PCI_DATA	0x10 3010	R/W	–/–
CONFIG_ADR	0x10 3014	R/W	R/W
CONFIG_DATA	0x10 3018	R/W	R/W
CONFIG_CTL	0x10 301C	R/W	R/W
IO_ADR	0x10 3020	R/W	R/W
IO_DATA	0x10 3024	R/W	R/W
IO_CTL	0x10 3028	R/W	R/W
SRC_ADR	0x10 302C	R/W	R/W
DEST_ADR	0x10 3030	R/W	R/W
DMA_CTL	0x10 3034	R/W	R/W
INT_CTL	0x10 3038	R/W	R/W

SE (Swap Bytes Enable). This bit is initialized after reset to zero, which causes the PCI interface operate in its default big-endian mode. Writing a one to SE causes the PCI interface operate in little-endian mode.

BO (Burst mode Off). This bit is initialized to zero, which allows the PCI interface to support burst-mode writes as a target on the PCI bus. Setting this bit to one disables burst-mode writes.

With burst mode enabled, the PCI interface buffers as much data as possible into r_buffer before issuing a disconnect to the PCI initiator. With burst mode disabled, the PCI interface buffers only one data phase before issuing a disconnect to the PCI initiator.

IntE (Interrupt Enables). The bits in the IntE field control the signalling of interrupts to the DSPCPU for PCI interface events. These events raise DSPCPU interrupt 16 if enabled. Table 10-13 lists the function of each IntE bit.

IntE is initially set to zeros (interrupts disabled).

Note that the error condition masked by bit 6 (see Section 10.6.3, “BIU_STATUS Register”) occurs when either a config_cycle or an io_cycle is requested and a request of either type is already in progress. That is, the second request need not be of exactly the same type that is already in progress.

Table 10-13. IntE Bit Functions

BIU_CTL Bit	If Set to One, Interrupt DSPCPU When...
2	config_cycle done
3	io_cycle done
4	dma_cycle done
5	pci_dram write cycle done
6	second config_cycle or io_cycle requested
7	second dma_cycle requested

IE (ICP DMA Enable). This bit is must be set to one to allow the Image Coprocessor (ICP) to write pixel data through the PCI interface. If this bit is cleared to zero, the ICP is not allowed to use the PCI interface. Programming of ICP DMA is described in Section 13.6, “Operation and Programming.”

HE (Host enable). This bit is initialized to zero, which prevents the DSPCPU from serving as the host CPU in the PCI system. If this bit is set to one, the Enable Mastering (EM) bit in the PCI Configuration register (see Section 10.5.3, “Command Register”) is also set to one (since TM1000 must be enabled to serve as a PCI bus initiator to perform PCI configuration).

CR (PCI Clear Reset). This bit releases the DSPCPU from its reset state. The TM1000 device driver (executing on an external host CPU) sets this bit to one after it completes TM1000’s configuration.

SR (PCI Set Reset). This bit forces the DSPCPU into its reset state. Writing one to this bit resets the CPU; writing zero causes no action. The TM1000 device driver (executing on an external host CPU) can set this bit to reset the DSPCPU.

10.6.5 PCI_ADR Register

The 30-bit PCI_ADR register is intended to be written only by the data cache. PCI_ADR participates in the special two-cycle data-cache-to-PCI protocol. See Section 10.6.6, “PCI_DATA Register,” for more information.

Only the DSPCPU can write to PCI_ADR. External PCI initiators can neither read nor write this register.

DSPCPU software should not write to this register (by writing to PCI_ADR in MMIO space). This register is intended only to support the special protocol between the data cache and PCI bus. An unexpected write to PCI_ADR via MMIO space will not be prevented by hardware and may result in data corruption on the PCI bus.

10.6.6 PCI_DATA Register

The 32-bit PCI_DATA register is intended to be used only by the data cache. PCI_DATA participates in the special two-cycle data-cache-to-PCI protocol.

The PCI_DATA and PCI_ADR registers are used together by the data cache to perform a single data phase PCI

memory-space read or write. A read operation is triggered when the data cache has written the transaction address into PCI_ADR and asserted the internal signal pci_read_operation (a direct internal connection between the data cache and PCI interface). A write operation is triggered when the data cache has written both PCI_ADR and PCI_DATA with the signal pci_read_operation deasserted.

While the PCI interface is performing the PCI read or write, the DSPCPU is stalled waiting for the completion of the PCI transaction. When the PCI transaction is complete, the PCI interface asserts pci_ready (a direct internal connection between the data cache and PCI interface). To finish a read operation, the data cache reads the PCI_DATA register, forwards the data to the DSPCPU, and then unlocks the DSPCPU. To finish a write, the data cache simply unlocks the DSPCPU.

Note that, if the DSPCPU attempts to access a non-existent PCI address, a RMA condition occurs. In this case, the value in the PCI_DATA register is set to 0. Hence, the DSPCPU always reads non-existent PCI locations as zero.

Normal MMIO write operations to PCI_DATA have no effect. Reads return the register's current value. External PCI initiators can neither read nor write this register.

10.6.7 CONFIG_ADR Register

The CONFIG_ADR register is written by the DSPCPU to set up for a configuration cycle. When TM1000 is acting as the host CPU, it must configure devices on the PCI bus. The DSPCPU writes CONFIG_ADR to select a configuration register within a specific PCI device. See Section 10.6.9, "CONFIG_CTL Register," for more information on initiating configuration cycles.

Following are descriptions of the fields of CONFIG_ADR.

BN (PCI Bus Number). The BN field (the two least-significant bits of CONFIG_ADR) selects one of four possible PCI buses. A value of zero for BN means that the targeted device is on the PCI bus directly connected to TM1000 and that any PCI-to-PCI bridges should ignore the configuration address. Any value for BN other than zero means that the targeted device is on a PCI bus connected to a PCI-to-PCI bridge and that all devices directly connected to TM1000's local PCI bus should ignore the configuration address.

RN (Register Number). The RN field (bits 2..7 of CONFIG_ADR) is used to specify one of the 64 configuration words within the target device's configuration space.

FN (Function Number). The FN field (bits 8..10 of CONFIG_ADR) is used to specify one of up to eight functions of the addressed PCI device.

DN (Device Number). The DN field (bits 11..31 of CONFIG_ADR) is used to select the targeted PCI device. Each bit corresponds to one of the 21 possible PCI devices on a single PCI bus; i.e., each bit corresponds to the idsel signal of one PCI device. Only one idsel signal—and, therefore, only one DN bit—can be asserted during a given configuration cycle.

10.6.8 CONFIG_DATA Register

The 32-bit CONFIG_DATA register is used by the DSPCPU to buffer data for a configuration cycle. When TM1000 is acting as the host CPU, it must configure the PCI bus and devices. The DSPCPU writes or reads CONFIG_DATA depending on whether it is performing a write or read to a PCI device's configuration space. See Section 10.6.9, "CONFIG_CTL Register," for more information on initiating configuration cycles.

10.6.9 CONFIG_CTL Register

The DSPCPU writes to CONFIG_CTL to trigger a configuration read or write cycle on the PCI bus. A PCI configuration read or write should not be performed during an ongoing PCI I/O read or write.

The steps involved in a DSPCPU PCI configuration access are:

1. Wait until BIU_STATUS io_cycle.Busy and config_cycle.Busy are both de-asserted
2. Write to CONFIG_ADR as described above, and (in case of a write operation) write to CONFIG_DATA.
3. Write to CONFIG_CTL to start the read or write.This action sets config_cycle.Busy.
4. Wait (polling or interrupt based) until config_cycle.Done is asserted by the hardware.
5. Retrieve the requested data in CONFIG_DATA (in case of a read)
6. Clear config_cycle.Done by writing a '1' to it.

Following are descriptions of the fields of CONFIG_CTL and a discussion of how a DSPCPU write to CONFIG_CTL triggers configuration cycles.

BE (Byte Enables). The BE field (the four least-significant bits of CONFIG_CTL) determines the state of PCI's four-line c/be# bus during the data phase of a configuration cycle. Since the c/be# bus signals are active low, a zero in a BE field bit means "byte participates;" a one in a BE field bit means "byte does not participate." Table 10-14 shows the correspondence between BE bits and bytes on the PCI bus *assuming little-endian byte order*.

RW (Read/Write). The RW field (bit 4 of CONFIG_CTL) determines whether the configuration cycle will be a read or a write. Table 10-15 shows the interpretation of RW.

Table 10-14. BE Field Interpretation

BE Bit	Interpretation
0	0 ⇒ byte 0 (LSB) participates 1 ⇒ byte 0 (LSB) does not participate
1	0 ⇒ byte 1 participates 1 ⇒ byte 1 does not participate
2	0 ⇒ byte 2 participates 1 ⇒ byte 2 does not participate

Table 10-14. BE Field Interpretation

BE Bit	Interpretation
3	0 ⇒ byte 3 (MSB) participates 1 ⇒ byte 3 (MSB) does not participate

Table 10-15. RW Interpretation

RW	Interpretation
0	Write
1	Read

A write by the DSPCPU to the CONFIG_CTL register starts a configuration cycle on the PCI bus. The CONFIG_DATA (for a write) and CONFIG_ADR registers must be set up before writing to CONFIG_CTL.

During a configuration read, the PCI interface drives the PCI bus with the address from CONFIG_ADR and the BE field from CONFIG_CTL. The returned data is buffered in CONFIG_DATA. When the data is returned, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate “done” status bin in BIU_STATUS. Finally, DSPCPU software reads the CONFIG_DATA register in MMIO space to access the data returned from the configuration cycle.

A write operation proceeds as for a read, except that PCI data is driven from CONFIG_DATA during the transaction and no data is returned in CONFIG_DATA.

10.6.10 IO_ADR Register

The 32-bit IO_ADR register is written by the DSPCPU to set up for an access to a location in PCI I/O space. The DSPCPU writes the address of the I/O register into IO_ADR. See Section 10.6.12, “IO_CTL Register,” for more information on initiating I/O cycles.

10.6.11 IO_DATA Register

The 32-bit IO_DATA register is used by the DSPCPU to set up for an access to a location in PCI I/O space. The DSPCPU writes or reads IO_DATA depending on whether it is performing a write or read from IO space. See Section 10.6.12, “IO_CTL Register,” for more information on initiating I/O cycles.

10.6.12 IO_CTL Register

The DSPCPU writes to IO_CTL to trigger a read or write access to PCI I/O space. The function of this register is similar to that of CONFIG_CTL, and the protocol for an I/O cycle is similar to the configuration cycle protocol. A PCI I/O read or write should not be performed during an ongoing PCI configuration read or write.

The steps involved in a DSPCPU PCI I/O access are:

1. Wait until BIU_STATUS io_cycle.Busy and config_cycle.Busy are both de-asserted
2. Write IO address to IO_ADR, and (in case of a write operation) writedata to IO_DATA.
3. Write to IO_CTL to start the read or write.This action sets io_cycle.Busy.
4. Wait (polling or interrupt based) until io_cycle.Done is asserted by the hardware.
5. Retrieve the requested data in IO_DATA (in case of a read)
6. Clear io_cycle.Done by writing a ‘1’ to it.

Following are descriptions of the fields of IO_CTL and a discussion of how a DSPCPU write to IO_CTL triggers I/O cycles.

BE (Byte Enables). The BE field (the four least-significant bits of IO_CTL) determines the state of PCI’s four-line c/be# bus during the data phase of an I/O cycle. Since the c/be# bus signals are active low, a zero in a BE field bit means “byte participates;” a one in a BE field bit means “byte does not participate.” Table 10-14 shows the correspondence between BE bits and bytes on the PCI bus *assuming little-endian byte order*.

RW (Read/Write). The RW field (bit 4 of IO_CTL) determines whether the I/O cycle will be a read or a write. Table 10-15 shows the interpretation of RW (0 ⇒ write, 1 ⇒ read).

A write by the DSPCPU to the IO_CTL register starts an I/O cycle on the PCI bus. The IO_DATA (for a write) and IO_ADR registers must be set up before writing to IO_CTL.

During an I/O read, the PCI interface drives the PCI bus with the address from IO_ADR and the BE field from IO_CTL. The returned data is buffered in IO_DATA. When the data is returned, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate “done” status bin in BIU_STATUS. Finally, DSPCPU software reads the IO_DATA register in MMIO space to access the data returned from the I/O cycle.

A write operation proceeds as for a read, except that PCI data is driven from IO_DATA during the transaction and no data is returned in IO_DATA.

10.6.13 SRC_ADR Register

The 32-bit SRC_ADR register maintains the source address for a block transfer during a DMA operation. The address is SRC_ADR must be word (4 byte) aligned, i.e. the 2 LSB’s have to be zero. This register is implemented as an incrementer to track the flow of data.

10.6.14 DEST_ADR Register

The 32-bit DEST_ADR register maintains the destination address for a block transfer during a DMA operation. The address is DEST_ADR must be word (4 byte) aligned, i.e. the 2 LSB’s have to be zero. This register is implemented as an incrementer to track the flow of data.

10.6.15 DMA_CTL Register

A write by the DSPCPU to the DMA_CTL register starts a DMA block transfer on the PCI bus. The SRC_ADR

and DEST_ADR registers must be set up before writing to DMA_CTL.

The steps involved in a DMA transfer are:

1. Wait until BIU_STATUS dma_cycle.Busy is de-asserted
2. Write to SRC_ADR and DEST_ADR as described above.
3. Write to DMA_CTL to start the dma transaction.This action sets dma_cycle.Busy.
4. Wait (polling or interrupt based) until dma_cycle.Done is asserted by the hardware.
5. Clear dma_cycle.Done by writing a '1' to it.

The fields of DMA_CTL are described below.

TL (Transfer Length). The TL field (bits 0..25 of DMA_CTL) specifies the number of data bytes to be transferred during the DMA operation. It must be a multiple of 4 bytes. The maximum length of a DMA operation is limited to 64M, the maximum amount of SDRAM supported by TM1000.

D (DMA Direction). The D field (bit 26 of DMA_CTL) determines the direction of data movement during the block transfer. Table 10-16 (shows the interpretation of the D field.

Table 10-16. D Interpretation

D	Data Movement Direction
0	SDRAM → PCI memory space (DMA write)
1	PCI memory space → SDRAM (DMA read)

T (DMA Transaction Type). The T field (bit 27 of DMA_CTL) determines the transaction type of a write, as described below.

Table 10-17. T interpretation

T	DMA Write transaction type
0	memory write
1	memory write-and-invalidate

TM1000 generates memory write-and-invalidate PCI transactions if all conditions below are satisfied, otherwise it generates regular memory write transactions:

- The MWI bit in the Command Register is set.
- The Cache Line Size register is set to 4,8 or 16 32-bit words.
- The DMA source address is 64 byte aligned.
- The DMA destination address is cache line size aligned.
- The T bit is set

During a PCI → SDRAM block transfer, the PCI interface drives the PCI bus with the address from SRC_ADR. The returned data is buffered in r_buffer. The PCI interface then drives the address from DEST_ADR and the data from r_buffer to the SDRAM controller. SRC_ADR and DEST_ADR are incremented, the TL field in DMA_CTL is decremented, and this sequence repeats until TL reaches zero.

At the end of the PCI → SDRAM block transfer, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate "done" status bin in BIU_STATUS.

During an SDRAM → PCI block transfer, the PCI interface drives the address from SRC_ADR to the SDRAM controller. The returned data is buffered in w_buffer. The PCI interface then drives the address from DEST_ADR and the data from w_buffer to the PCI bus. SRC_ADR and DEST_ADR are incremented, the TL field in DMA_CTL is decremented, and this sequence repeats until TL reaches zero.

At the end of the SDRAM → PCI block transfer, the PCI interface can generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate "done" status bit in BIU_STATUS.

10.6.16 INT_CTL Register

The INT_CTL register contains three fields for setting, enabling, and sensing the four PCI interrupt lines. Table 10-18 shows the interpretation of the fields in INT_CTL.

INT (Interrupt bits). The INT field (bits 0..3 of INT_CTL) can force a PCI interrupt to be signalled.

IE (Interrupt Enable). The IE field (bits 4..7 of INT_CTL) enables TM1000 to drive PCI interrupt lines.

IS (Interrupt State). The IS field (bits 8..11 of INT_CTL) senses the state of the PCI interrupt lines.

Table 10-18. INT_CTL Bits

INT_CTL		PCI Signal	Programming
Field	Bit		
INT	0	inta#	0 ⇒ Deassert intx# 1 ⇒ Assert intx# (if enabled); i.e., pull intx# pin to a low logic level
	1	intb#	
	2	intc#	
	3	intd#	
IE	4	inta#	0 ⇒ Disable open-collector output to intx# 1 ⇒ Enable open-collector output to intx#
	5	intb#	
	6	intc#	
	7	intd#	
IS	8	inta#	Reads state of intx# pin: 0 ⇒ No interrupt asserted (intx# is high) 1 ⇒ Interrupt is asserted (intx# is low)
	9	intb#	
	10	intc#	
	11	intd#	

Figure 10-9 shows a conceptual realization of the logic used to implement the control of each intx# pin.

See also Section 3.5, "TM1000 Host Interrupts."

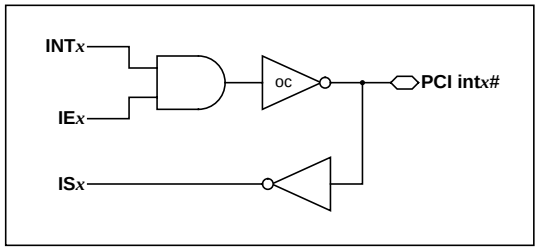


Figure 10-9. Conceptual realization of intx# pin control logic.

10.7 PCI BUS PROTOCOL OVERVIEW

TM1000's PCI interface can generate and respond to several types of PCI bus commands. Table 10-19 lists the 12 possible commands and whether or not TM1000 can generate them.

Table 10-19. TM1000 PCI Commands as Initiator

TM1000 Generates	TM1000 Cannot Generate
Configuration read Configuration write Memory read Memory write Memory write and invalidate Memory read line Memory read multiple I/O read I/O write	Interrupt acknowledge Special cycle Dual address

Table 10-20 lists the 12 possible commands and whether or not TM1000 can respond to them.

Table 10-20. TM1000 PCI Commands as Target

TM1000 Responds To	TM1000 Ignores
Configuration read Configuration write Memory read Memory write Memory read line Memory read multiple	I/O read I/O write Interrupt acknowledge Special cycle Dual address Memory write and invalidate

The basic transfer mechanism on the PCI bus is a burst, which consists of an address phase followed by one or more data phases. In TM1000, the DSPCPU and Image Coprocessor (ICP) are the only two units that can cause TM1000 to become a PCI-bus initiator; i.e., only the DSPCPU and ICP can access external resources.

10.7.1 Single-Data-Phase Operations

When the DSPCPU reads or writes PC memory, the PCI transaction has only a single data phase. A typical single-data-phase read operation is illustrated in Figure 10-10. During the first clock period, the TM1000 asserts the frame# signal to indicate that the transaction has begun and that an address and command are stable on ad and c/be#, respectively.

TM1000 then releases the ad bus, deasserts frame#, asserts irdy#, asserts byte enables on c/be#, and waits for the target to claim the transaction by asserting devsel#. The target asserts trdy# to signal the master that the ad bus contains stable data. The assertion of trdy# causes the initiator (TM1000 in this case) to sample the ad bus data and deassert irdy# to complete the single-data-phase read transaction.

Figure 10-11 shows a typical single-data-phase write operation. The operation begins as with the read: TM1000 asserts the frame# signal and drives the ad bus with the target address and drives the command onto the c/be# bus.

The operation continues when TM1000 deasserts frame#, asserts irdy#, and drives the byte enables as before, but it also drives the data to be written on the ad bus. The target device asserts devsel# to claim the transaction. Eventually, the target asserts trdy# to signal that it is sampling the data on the ad bus. TM1000 continues

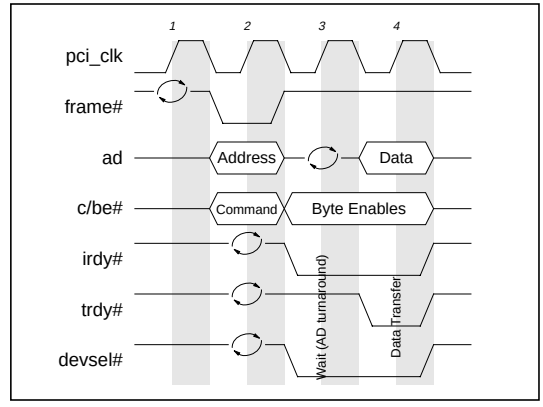


Figure 10-10. Basic single-data-phase read opera-

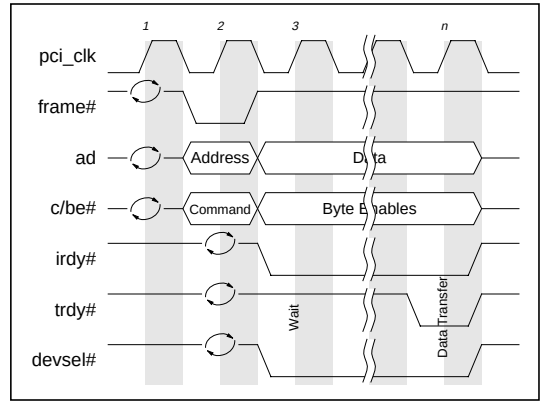


Figure 10-11. Basic single-data-phase write opera-

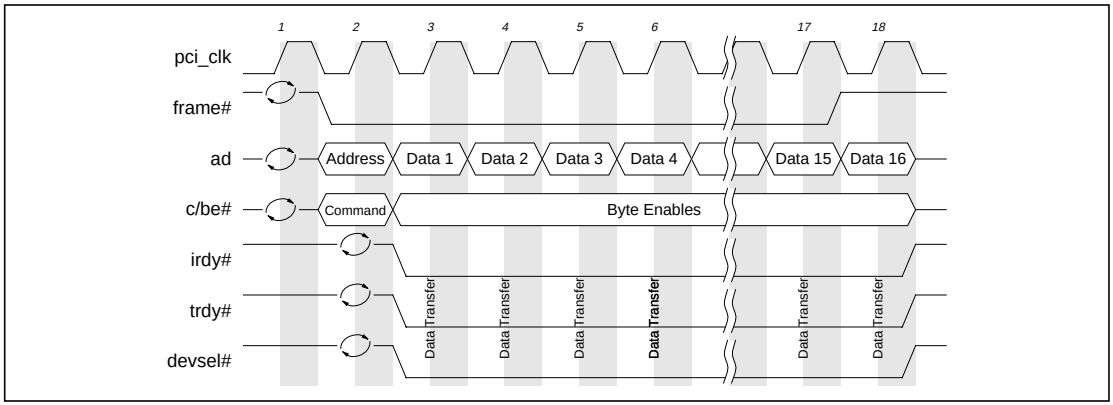


Figure 10-12. PCI burst write operation with 16 data phases.

to drive the data on the ad bus until after the target deasserts trdy#, which completes the write operation.

10.7.2 Multi-Data-Phase Operations

As with the single-data-phase operations, DMA operations begin with the assertion of frame# and valid address and command information. See Figure 10-12. The target knows a burst is requested because frame# remains asserted when irdy# becomes asserted.

In the example timing of Figure 10-12, a fast device is receiving the burst from TM1000. The target asserts devsel# and trdy# simultaneously. The trdy# signal remains asserted while TM1000 sends a new word of data on each PCI clock cycle. The burst operation shown is a 16-word burst transfer. Since only the starting address is sent by the initiator, both initiator and target must increment source and destination addresses during the burst.

The initiator signals the end of the burst of data in Figure 10-12 when it deasserts frame# in clock 17. The last word (or partial word) of data is transferred in the clock cycle after frame# is deasserted. Finally, the target acknowledges the last data phase by deasserting trdy# and devsel#.

Figure 10-13 illustrates back-to-back DMA burst data transfers. The ICP is capable of exploiting the high bandwidth available with back-to-back DMA operations when it is writing image data to a frame buffer on a PCI video card.

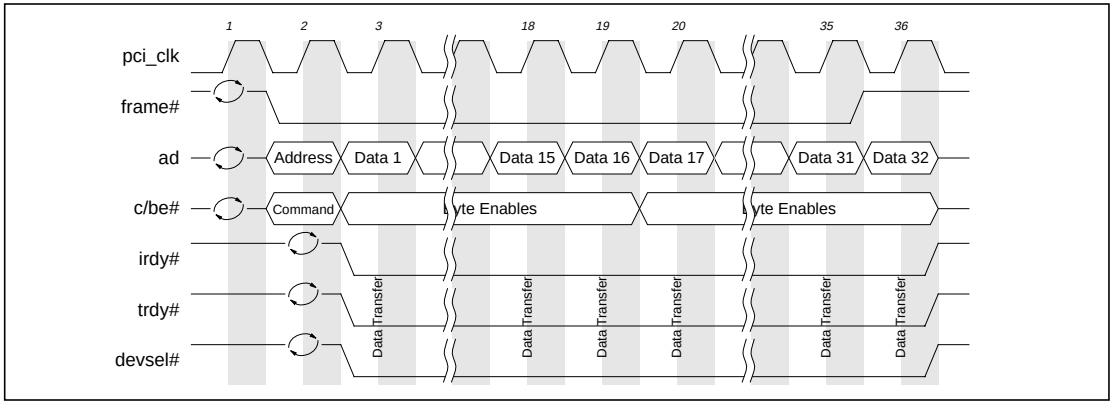


Figure 10-13. Back-to-back PCI burst write operations with 16 data phases such as might be generated by the ICP when writing image data to a PCI-resident video frame buffer.

The timing of Figure 10-13 assumes that the PCI bus is granted to TM1000 until at least the beginning of the second DMA burst operation. For as long as bus ownership is granted to TM1000 and the ICP has queued requests for data transfer, the PCI interface will perform back-to-back DMA operations. If the target eventually becomes unable to accept more data, it signals a disconnect TM1000's PCI interface. The PCI interface remembers where the DMA burst was interrupted and attempts to restart from that point after two bus clocks.

10.8 LIMITATIONS

10.8.1 Bus Locking

The PCI interface does not implement lock#, sbo, and sbone pins. Consequently, it is possible for both the DSPCPU and external PCI initiators to write to a critical memory section simultaneously. Software must implement policies to guarantee memory coherency.

10.8.2 No Expansion ROM

TM1000 does not implement the PCI expansion ROM capability.

10.8.3 No Cacheline Wrap Address Sequence

The PCI interface does not implement the PCI cacheline-wrap address mode for external PCI initiators that access TM1000 SDRAM.

10.8.4 No Burst for I/O or Configuration Space

Only single-data-phase transactions to configuration and I/O spaces are supported. The byte-enable signals select the byte(s) within the addressed word.

10.8.5 Word-Only MMIO Register Access

External initiators can access TM1000 MMIO registers only as full words. The byte-enable signals have no effect on the data transferred. External initiators must read and write all four bytes of MMIO registers.

SDRAM Memory System Chapter 11

by Eino Jacobs, Chris Nelson

11.1 TM1000 MAIN MEMORY OVERVIEW

TM1000 connects to its local memory system with a dedicated memory bus as shown in Figure 11-1. This bus interfaces only with SDRAM (or SGRAM with its DSF pin tied low), and TM1000 is the only master on this bus. For up to four memory chips, the interface is glueless.

A variety of device types, speeds, and rank[1] sizes are supported, which allows a range of TM1000 systems to be built. Table 11-1 summarizes the memory system features.

The interface provides all control and data signals with sufficient drive capacity for a glueless connection to a 100-MHz memory system with up to four memory devices. Note that memory-system speed can be different from TM1000 core speed; the ratio between the memory system clock and TM1000 core clock is programmable.

With current technology, TM1000 supports a glueless 8-MB memory system with four 2×1M×8 SDRAM chips (four devices with 2 banks of one million words, each 8 bits wide). Larger memories require a lower memory system clock frequency (though the TM1000 core clock can be higher), and the largest memory arrays will require external buffers to increase drive capacity.

Table 11-1. Memory System Features

Characteristic	Comments
Data width	32 bits
Number of ranks	Four chip-select signals support up to four ranks
Memory size	From 512KB to 64MB
Devices supported	• Jedec SGRAM (2×128K×32, DSF tied low) • Jedec SDRAM (×4, ×8, ×16, ×32)
Clock rate	Up to 100 MHz SDRAM speed (programmable ratio between TM1000 core clock and memory system clock)
Bandwidth	400 MB/s (@ 100 MHz)
Glueless interface	• Up to four chips @ 100 MHz (e.g., 8 MB memory with 2×1M×8 SDRAM) • More chips with slower clock and/or external buffers
Signal levels	3.3-V LVTTL

11.2 MAIN-MEMORY ADDRESS APERTURE

TM1000's local main memory is just one of three apertures into the 4-GB address space of the DSPCPU:

- SDRAM (0.5 to 64 MB in size),
- MMIO (2 MB in size), and
- PCI (any address not in SDRAM or MMIO).

MMIO registers control the positions of the address-space apertures. The SDRAM aperture begins at the absolute address specified in the MMIO register DRAM_BASE and extends upward to the address specified in the DRAM_LIMIT register. The MMIO aperture begins at the address in MMIO_BASE, which defaults to 0xEFE00000 after power-up, and extends upwards two megabytes. (See Chapter 3, "DSPCPU Architecture," for a detailed discussion.) All addresses that fall outside these two apertures are assumed to be part of the PCI address aperture.

1. In this document, the term "rank" is used to refer to a group of memory devices that are accessed together. Historically, the term "bank" has been used in this context; to avoid confusion, this document uses "bank" to refer to on-chip organization (SDRAM devices have two internal banks) and "rank" to refer to off-chip, system-level organization.

11.3 MEMORY DEVICES SUPPORTED

The devices and organizations supported can be configured as listed in Table 11-2. All devices must have a LVTTL, 3.3-V interface.

Table 11-2. Supported Rank Configurations

Device Size (Mbit)	Device(s)	Rank Size
2	2×64K×16 SDRAM	512 KB
4	2×128K×16 SDRAM	1 MB
8	2×128K×32 SGRAM	1 MB
16	2×256K×32 SDRAM	2 MB
	2×512K×16 SDRAM	4 MB
	2×1M×8 SDRAM	8 MB
	2×2M×4 SDRAM	16 MB

11.3.1 SDRAM

TM1000 is designed to support synchronous DRAM chips directly. SDRAM has a fast, synchronous interface

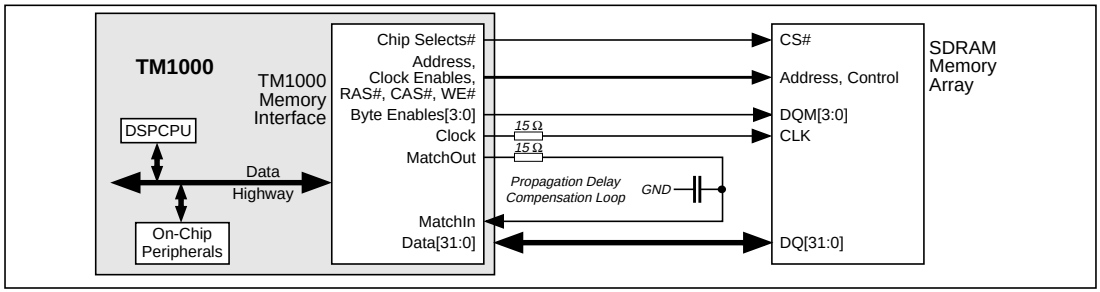


Figure 11-1. TM1000 provides a high-performance memory interface for local main memory. The interface connects the internal highway bus to external SDRAM or SGRAM. The interface is glueless for an array of up to four devices.

that permits burst transfers at a rate of one word per clock cycle. The memory inside an SDRAM device is divided into two banks, and the SDRAM implements interleaved bank access to sustain maximum bandwidth.

SDRAM devices implement a power-down mechanism with self-refresh. TM1000's power management takes advantage of this mechanism.

TM1000 supports only Jedec-compatible SDRAM with two internal banks of memory per device.

11.3.2 SGRAM

Synchronous graphics DRAM (SGRAM) can also be used in a TM1000 system. SGRAM has a 2×128K×32 organization, and is essentially an SDRAM with some additional features for raster graphics functions. The device type is standardized by Jedec and offered by multiple DRAM vendors. SGRAM devices are packaged in a 100-pin QFP and are available in speed grades up to 100-MHz.

By tying the DSF input of an SGRAM low, the device operates like a standard 32-bit-wide SDRAM. Thus, tying DSF low makes SGRAM compatible with TM1000's memory interface.

11.4 MEMORY GRANULARITY AND SIZES

TM1000 supports a variety of memory sizes thanks to:

- The availability of many organizations of SDRAM devices, and
- TM1000's support for up to four memory ranks.

The minimum memory size is 512KB using two 2×64K×16 SDRAM parts on the 32-bit data bus.

Up to four memory devices can be connected to TM1000 without any glue logic and without sacrificing any performance. The maximum memory size with full performance is 8MB using four 2×1M×8 SDRAMs on a 32-bit data bus.

Larger memories can be constructed using more devices, but the frequency of the memory interface must be lowered to account for the extra propagation delay due to the excessive loading on the interface signals (see Section 11.12, "Output Driver Capacity"). When a very large number of chips is connected (more than 16), it is advantageous to add external buffers to the address and control signals.

The following rules apply to memory rank design:

- All devices in a rank must be of the same type.
- All ranks must be a power of two in size.
- All ranks must be equal size.

Table 11-3 lists some example memory system designs. Note that the 64-MB configuration requires external buffers. Note:

- Some of these configurations may not be economically attractive due to the price premium for small-capacity devices.
- "Max. MHz" refers to the memory interface/SDRAM speed, not the TM1000 core operating frequency.

Table 11-3. Example Memory Configurations

Size (MB)	Ranks	Rank Configurations	Max. MHz	Peak MB/s
0.5	1	two 2×64K×16 SDRAM	100	400
1	1	one 2×128K×32 SGRAM	100	400
1	1	two 2×128K×16 SDRAM	100	400
2	1	one 2×256K×32 SDRAM	100	400
4	1	two 2×512K×16 SDRAM	100	400
8	1	four 2×1M×8 SDRAM	100	400
8	2	two 2×512K×16 SDRAM two 2×512K×16 SDRAM	100	400
16	1	eight 2×2M×4 SDRAM	66	264
32	2	eight 2×2M×4 SDRAM eight 2×2M×4 SDRAM	50	200
64	4	eight 2×2M×4 SDRAM eight 2×2M×4 SDRAM eight 2×2M×4 SDRAM eight 2×2M×4 SDRAM	50 (with buffs.)	200

11.5 MEMORY SYSTEM PROGRAMMING

Memory system parameters are determined by the contents of two configuration registers, MM_CONFIG and

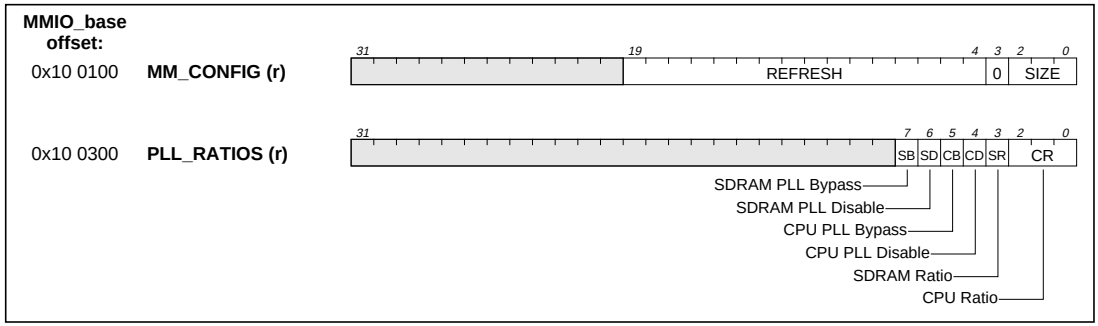


Figure 11-2. Memory interface configuration registers.

PLL_RATIOS. Table 11-4 describes the function of these registers, and Figure 11-2 shows their formats.

Table 11-4. Memory Interface Configuration Registers

Register	Purpose
MM_CONFIG	Describes external memory configuration
PLL_RATIOS	Controls separate memory and CPU PLLs (phase-locked loops)

MM_CONFIG and PLL_RATIOS are loaded from the boot EEPROM, as described in Section 12.4, "Detailed EEPROM Contents." During this boot process, the memory interface is held in reset state. After the memory interface is released from reset, the contents of these registers cannot be altered.

These registers are visible in MMIO space. They can be read, but writes have no effect.

11.5.1 MM_CONFIG Register

The MM_CONFIG register tells the memory interface how to use the local DRAM memory. The fields in this register tell the interface the rank size and the refresh rate of the memory. Table 11-6 summarizes the field functions.

Table 11-5. MM_CONFIG Fields

Field	Function		
REFRESH	Refresh interval in memory clock cycles. Default value 1000 (0x03E8).		
SIZE	Memory rank size	0	Reserved
		1	512KB
		2	1MB
		3	2MB
		4	4MB
		5	8MB
		6	16MB
		7	Reserved

REFRESH (Refresh interval). The 16-bit REFRESH field specifies the number of memory-system clock cycles between refresh operations. The default value of this register is 1000 (0x03E8). See Section 11.10, "Refresh," for more information.

Bit three of MM_CONFIG must be set to zero for normal operation.

SIZE (Rank Size). The three-bit SIZE field specifies the size of each rank of DRAM. Each rank must be the size specified by SIZE. The default is a rank size of 4MB. Refer to Table 11-5 for the interpretation of this field.

11.5.2 PLL_RATIOS Register

The PLL_RATIOS register controls the operation of the separate memory-interface and CPU PLLs. Fields in this register determine if the PLLs are active and what input:output ratio each PLL should generate. Table 11-6 summarizes the field functions. Figure 11-3 shows how the PLLs are connected and how fields in the PLL_RATIOS register control them.

Table 11-6. PLL_RATIOS Fields

Field	Function		
CR	CPU:memory ratio	**0**	1:1
		1	2:1
		2	3:2
		3	4:3
		4	5:4
		5–7	Reserved
SR	Memory:external ratio	**0**	2:1
		1	3:1
CD	CPU PLL Disable	0	CPU PLL on
		1	CPU PLL off
CB	CPU PLL bypass	0	CPU ← PLL
		1	**CPU ← Memory**
SD	SDRAM PLL Disable	0	SDRAM PLL on
		1	SDRAM PLL off
SB	SDRAM PLL bypass	0	Memory ← PLL
		1	**Memory ← external**

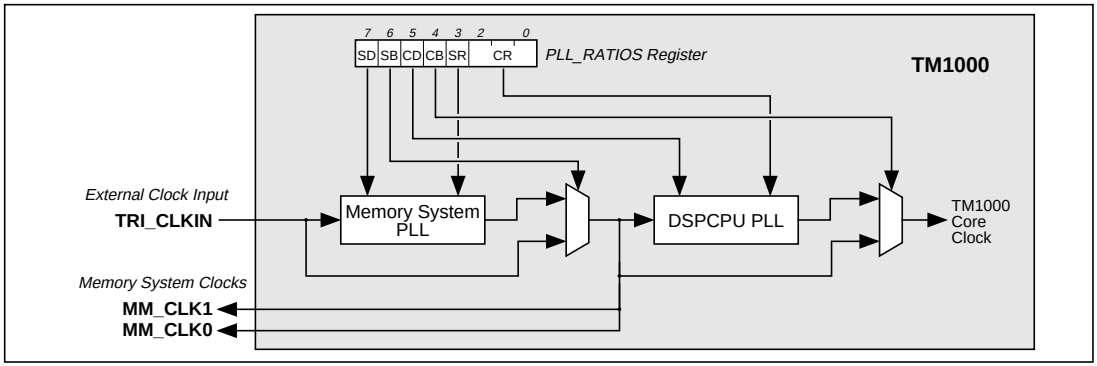


Figure 11-3. TM1000 memory and core PLL connections.

CR (CPU-to-memory PLL Ratio). The three-bit CR field selects one of five input-to-output clock ratios for the CPU PLL. The input clock is the memory system clock; the output clock determines TM1000's core operating frequency. The default value is zero, which implies a 1:1 CPU:memory ratio. See Table 11-6 for other encodings.

SR (Memory-to-external PLL Ratio). The one-bit SR field selects one of two memory-to-external clock ratios for the memory interface PLL. The PLL input is TM1000's external input clock TRI_CLKIN; the PLL output determines the operating frequency of the memory interface and SDRAM devices. The default value is zero, which implies a 2:1 memory:external ratio. A value of one implies a 3:1 ratio.

CD (CPU PLL Disable). The one-bit CD field determines whether or not the CPU PLL is turned on. The reset value is one, which disables operation of the CPU PLL and dissipates almost no power. For normal operation the value should be zero, enabling the CPU PLL.

CB (CPU PLL Bypass). The one-bit CB field determines whether the input or the output of the CPU PLL drives TM1000's core logic. The default value is one, which causes the TM1000 core to be clocked by the input of the CPU PLL (i.e., the memory interface clock). A value of zero causes normal operation, and the core is clocked by the output of the CPU PLL.

Note that if both CB and SB are set to one (bypass the CPU PLL and bypass the SDRAM PLL), TM1000's core logic is effectively clocked at the external input frequency.

Note: it is illegal to use the output of a disabled PLL. For example, it is illegal to have CD set to one while CB is set to zero.

SD (SDRAM PLL Disable). The one-bit SD field determines whether or not the SDRAM PLL is turned on. The default value is one, which disables the SDRAM PLL, and it dissipates almost no power. For normal operation the value should be zero, enabling the SDRAM PLL.

SB (SDRAM PLL Bypass). The one-bit SB field determines whether the input or the output of the SDRAM PLL drives the memory interface and memory devices. The default value is one, which causes the memory system to be clocked by the input of the SDRAM PLL (TM1000's external input clock). A value of zero causes normal operation, and the memory system is clocked by the output of the SDRAM PLL.

11.6 MEMORY INTERFACE PIN LIST

The memory interface consists of 61 signal pins including clocks (but excluding power and ground pins). Table 11-7 lists the interface signal pins.

Table 11-7. Memory Interface Signal Pins

Name	Function	I/O	Active...
MM_CLK[1:0]	Memory bus clock	O	High
MATCHOUT	Clock propagation match-trace output	O	High
MATCHIN	Clock propagation match-trace input	I	High
MM_CS#[3..0]	Chip selects for the four memory ranks	O	Low
MM_RAS#	Row-address strobe	O	Low
MM_CAS#	Column address strobe	O	Low
MM_WE#	Write enable	O	Low
MM_A[11:0]	Address	O	High
MM_CKE[1:0]	Clock enable	O	High
MM_DQM[3:0]	Byte enables for dq bus	O	High
MM_DQ[31:0]	Bi-directional data bus	I/O	High

11.7 ADDRESS MAPPING

Table 11-8 shows how internal address bits from the data highway bus (which connects all internal TM1000 units) are mapped to main-memory address-bus pins (MM_A[11:0]). The mapping is determined by the state of the rank-size bits in the MM_CONFIG register.

Table 11-8. Address Mapping Based on Rank Size

Rank Size	Rank Addr.	Row Address		Column Address		Bank Address	
	H.Way Bits	Pins	H.Way Bits	Pins	H.Way Bits	Pin	H.Way Bit
512 KB	20-19	8, 6–0	18, 17–11	7–0	10–6, 4–2	9	5
1 MB	21-20	8–0	19–11	7–0	10–6, 4–2	9	
2 MB	22-21	9–0	20–11	7–0	10–6, 4–2	10	
4 MB	23–22	10–0	21–11	7–0	10–6, 4–2	11	
8 MB	24-23	10–0	22–12	8–0	11–6, 4–2	11	
16 MB	25-24	10–0	23–13	9–0	12–6, 4–2	11	

The column “Rank Addr./H.Way Bits” specifies which internal data-highway address bits select the preliminary SDRAM rank. The actual rank used is subject to the limitation implied by the relationship between SDRAM aperture size (described in Section 12.2.1) and the rank size. The rank is selected via the chip select bits, MM_CS#[3:0].

The column “Row Address/H.Way Bits” specifies which internal data-highway address bits map to the SDRAM row address. “Row Address/Pins” specifies which lines of TM1000’s MM_A address bus serve as the SDRAM row address.

The column “Column Address/H.Way Bits” specifies which data-highway address bits map to the SDRAM column address. “Column Address/Pins” specifies which lines of TM1000’s MM_A address bus serve as the SDRAM column address.

Bits 5–0 of the highway address are the offset within a 64-byte block; these bits are all zero for an aligned block transfer. The table lists the mapping of bits 5–2 to identify in which SDRAM positions the words of a block are located.

Bit 5 of the highway address is always mapped to the SDRAM internal bank select; thus, each SDRAM bank receives half (32 bytes) of the block transfer.

Bits 4–2 of the highway address are the word offset in a cache block. Bits 1–0 are the byte offset within a 32-bit word.

11.8 MEMORY INTERFACE AND SDRAM INITIALIZATION

Immediately after reset, the main-memory interface is initialized by placing default values in the MM_CONFIG and PLL_RATIOS registers (see Section 11.5, “Memory System Programming”). During the subsequent hardware boot process, when TM1000 reads initial values from an external ROM, these registers can be set to different values.

After TM1000 is released from the reset state, the memory interface automatically executes 10 refresh operations, then initializes the mode register in each SDRAM chip. Table 11-9 shows the settings in the SDRAM mode register(s).

Table 11-9. SDRAM Mode Register Settings

Parameter	Value
Burst Length	4
Wrap type	Interleaved
CAS latency	3

11.9 ON-CHIP SDRAM INTERLEAVING

The main-memory interface takes advantage of the on-chip interleaving of SDRAM devices. Interleaving allows the precharge, RAS, and CAS delays needed to ready one internal bank to be performed while useful data transfer is occurring with the other internal bank. Thus, the overhead of preparing one bank is hidden during data movement to or from the other.

The benefit of on-chip interleaving is sustainable full-bandwidth data transfer (one word per clock cycle). The transition from one internal bank to the other happens on 8-word boundaries; transferring 8 words gives the inac-

tive bank time to prepare (perform precharge, RAS, and CAS) so that when the last word of the 8-word block in the active bank has been transferred, the next word from the just-precharged bank is ready on the next cycle.

The seamless transitions between the two on-chip banks can be sustained for a stream of contiguous addresses with the same direction (read or write). That is, a stream of contiguous reads or contiguous writes can sustain full bandwidth. If a write follows a read, then a small gap between transfers is needed.

Each bank access is terminated with a read or write with automatic precharge, making a separate precharge command before the next RAS unnecessary.

11.10 REFRESH

The main-memory interface performs SDRAM refresh cycles autonomously using the CAS-before-RAS (CBR) mechanism. SDRAMs have a 4K refresh interval: either 4096 rows must be refreshed every 64 ms or 2048 rows every 32 ms.

The main-memory interface performs refresh at timed intervals: one CBR refresh command must be issued every 16 µSec. A counter in the main-memory interface keeps track of the number of SDRAM clock cycles between refresh operations. This counter starts after the CBR operation has completed; this CBR operation take 19 cycles. When the counter reaches a programmed limit, the next refresh operation is due, and the next-in-line data transfer request from the data-highway is delayed until the CBR operation is executed.

All devices in the main-memory system are refreshed simultaneously. The REFRESH field in the MM_CONFIG register determines the number of memory-system clock cycles (as distinguished from TM1000 core clock cycles) between the CBR refresh operations. Table 11-10 lists the number of memory-system clocks for typical SDRAM operation speeds.

Table 11-10. Refresh Intervals

SDRAM Operation Speed	Value For REFRESH Field (decimal)
66 MHz	1000
75 MHz	1140
83 MHz	1270
100 MHz	1540

Each CBR refresh operation takes 19 SDRAM clock cycles. Thus, at 100-MHz, refresh consumes about 1.2% of maximum available SDRAM bandwidth (19 cycles out of 1540). The bandwidth impact is slightly higher at lower frequencies.

11.11 POWER SAVING MODE

When TM1000 is put into sleep mode to reduce power consumption, the main-memory interface responds by putting the SDRAM devices into their power-down mode. In this mode, the SDRAM devices retain their contents through self-refresh.

11.12 OUTPUT DRIVER CAPACITY

TM1000's output driver circuits for the memory address and control signals (output signals in Table 11-7), can drive up to four memory devices when the memory interface is operating at 100 MHz. If more devices are connected, then a lower SDRAM clock frequency must be chosen.

Table 11-11 lists the clock frequency as a function of the number of memory devices connected to unbuffered memory interface signals.

Table 11-11. Glueless Interface Limits

Memory Chips	Maximum Clock Frequency
4	100 Mhz
6	80 MHz
8	66 MHz
16	50 MHz

Two identical outputs are provided for both the MM_CKE (clock-enable) and MM_CLK signals. Each MM_CKE and MM_CLK signal is capable of driving two SDRAM devices at 100MHz, thus the total of four devices.

11.13 SIGNAL PROPAGATION DELAY COMPENSATION

The memory interface has two special pins, matchout and matchin, that help the interface compensate for the propagation delay through circuit-board traces to and from the external SDRAM devices. At high clock frequencies, e.g., 100 MHz, propagation delay becomes significant compared to the clock period, which is as small as 10 ns.

Matchout and matchin are connected through a dedicated trace on the circuit board. This trace forms a "match loop" with an outgoing part and an incoming part. The outgoing part should match the clock trace from the memory interface to the SDRAM(s). The incoming part should match the longest trace between the SDRAM(s) and the memory interface pins.

Since the memory interface uses the matchin signal to sample incoming data, the match-loop trace should estimate the round-trip propagation delay as closely as possible. This can be achieved with careful circuit board layout and some passive components to estimate capacitive loading.

A lumped capacitive load is attached to the middle of the matchout/matchin trace to represent the sum of the clock-input and data-line loads. The lumped load should account for the number of SDRAM devices attached to the clock line. The memory interface provides two clock outputs, each capable of driving one or two memory devices directly.

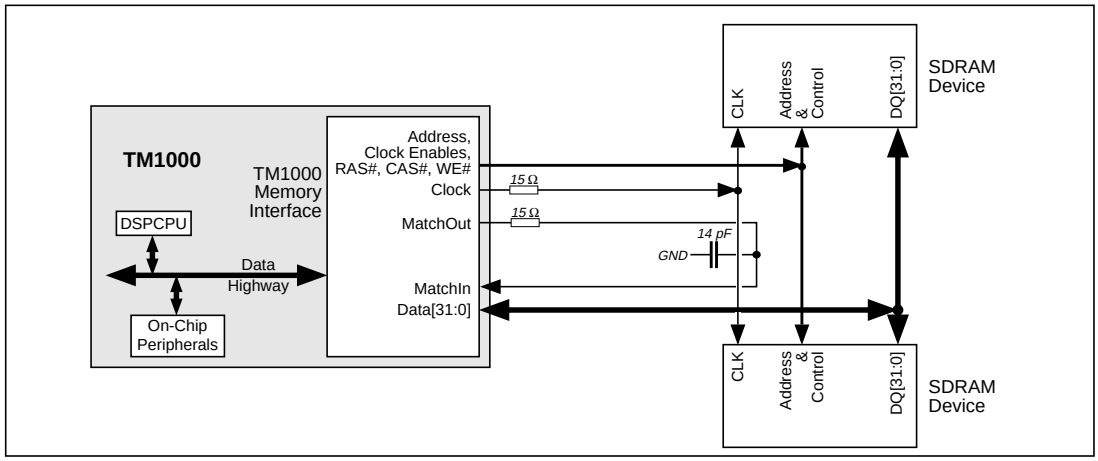


Figure 11-4. Conceptual board layout. The match trace loop should be as close to the sum of the lengths of the clock and data traces as possible.

Finally, to avoid excessive ringing of the clock signals, series termination with a 15-Ohm resistor is advised at the clock and matchout outputs when the memory interface is operating at 100 MHz.

The phase delay of the memory clock with respect to the internal sending and receiving clocks is adjusted inside the memory interface to achieve reliable communication and guarantee correct setup and hold times.

Figure 11-4 shows a conceptual circuit board layout. Two SDRAM devices share a single clock output. The clock and matchout signals have source-series termination. The matchout/matchin trace has a lumped load estimating two SDRAM clock input and data loads.

11.14 CIRCUIT BOARD DESIGN

TM1000 and its memory array form a high-speed digital system. Even though only a small number of chips is involved, this digital system operates at frequencies high enough to make the analog characteristics of the connections between the chips significant. Consequently, the system designer must take care to ensure reliable operation.

11.14.1 General Guidelines

- In general, TM1000 and its memory chips should be as close together as possible to minimize parasitic capacitance. Close proximity is especially important for a 100-MHz memory system.
- SIgnal traces between TM1000 and the memory chips should be matched in length as closely as possible to minimize signal skew.
- The clock-signal trace(s) should be as short as possible.
- Address and control-signal traces should also be short, but their length is less critical than the clock's.
- Data-signal traces should also be short, but their length is less critical than the clock's, especially if only one or two ranks are connected.
- The length of the trace between matchout and matchin should be as close as possible to the sum of the lengths of the longest clock and data traces.

11.14.2 Specific Guidelines

- The maximum length for a signal trace is 10 cm.
- The maximum capacitive load is 30 pF per trace, including loads.
- The signal traces on the TM1000 circuit board must be designed as 50-Ohm transmission lines.
- At 100 MHz, the memory chips should also be soldered to the circuit board.
- At most two SDRAM devices may be connected to each MM_CLK signal at 100 MHz.

11.14.3 Termination

No termination is required for address, data, and control signals. Address and control signals are driven only by TM1000; the output impedance of the drivers is sufficiently matched to prevent excessive ringing. TM1000 design assumes that the output drivers of SDRAM chips, when driving data lines, are also sufficiently impedance matched.

Series termination of the clock and matchout outputs with a 15-Ohm resister is advised when operating the memory system at 100-MHz (see Section 11.13, "Signal Propagation Delay Compensation").

11.15 TIMING BUDGET

The glueless interface of the TM1000 main-memory interface makes the memory system simple and straightforward from one point of view, but to ensure reliable operation at high clock rates, system designers must follow

the match-loop and board design guidelines (see Section 11.13, “Signal Propagation Delay Compensation,” and Section 11.14, “Circuit Board Design”).

The following A.C. timing specifications are provided to help the verification of a memory system design. The timing parameters take into account the following:

- Corners in the fabrication process, temperature, and voltage.
- Ground and V_{DD} bounce.
- Transmission-line reflections.
- Stub mismatch.
- Signal trace wire-length mismatch.
- Imbalance in internal chip wiring.
- Tester accuracy of ± 400 ps.

These timing specifications do not include any other uncorrelated margin. Table 11-12 lists four general timing parameters for the memory bus assuming worst-case conditions for a board designed in compliance with the guidelines of Section 11.14, “Circuit Board Design.”

SDRAM devices must meet the critical specifications listed in Table 11-13 to ensure reliable operation of a 100-MHz memory system. These values leave virtually no margin for the critical timing parameters in a high-speed system.

Table 11-12. Memory-Bus Timing Parameters, Worst-Case Board Design

Timing Parameter	Value
Max. output delay of data, address, and control; (referenced to SDRAM clock input)	6.6 ns
Min. output hold time of data, address, and control; (referenced to SDRAM clock input)	1.0 ns
Min. input setup time of data; (referenced to MatchIn)	0.8 ns
Min. input hold time of data; (referenced to MatchIn)	1.9 ns

Table 11-13. Required SDRAM Performance For 100-MHz Memory System

Timing Parameter	Value
Max. output delay	9.0 ns
Min. output hold time	3.0 ns
Max. input setup time	3.0 ns
Max. input hold time	1.0 ns

11.16 EXAMPLE BLOCK DIAGRAMS

Figure 11-5, Figure 11-6, Figure 11-7, Figure 11-8, and Figure 11-9 illustrate some common memory system designs. Figure 11-5 shows a system with a single SGRAM chip; the others show a variety of SDRAM-based systems.

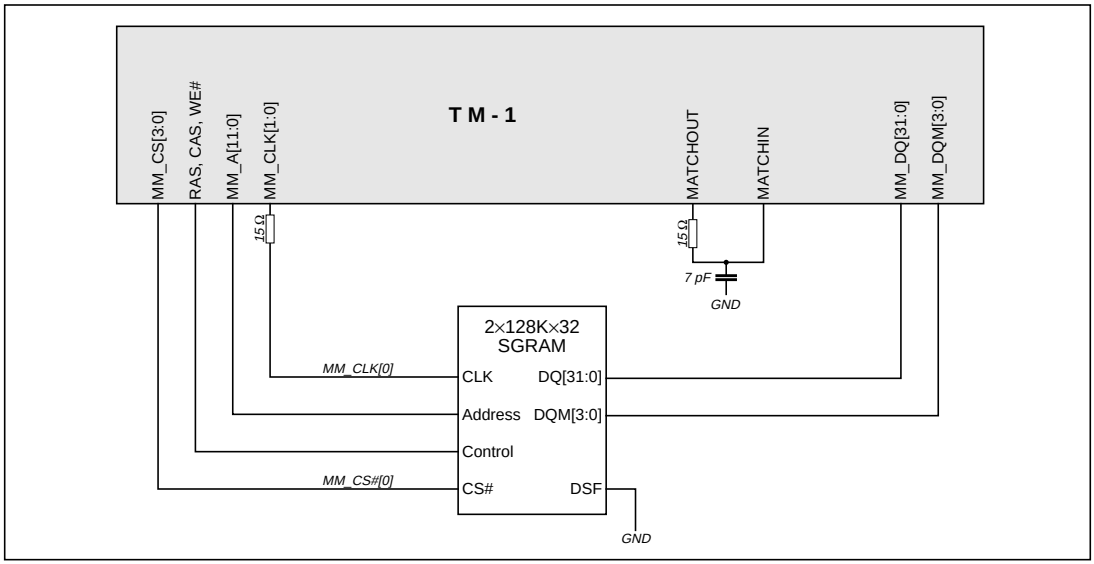


Figure 11-5. Schematic of a 1-MB memory system consisting of one 2×128K×32 SGRAM.

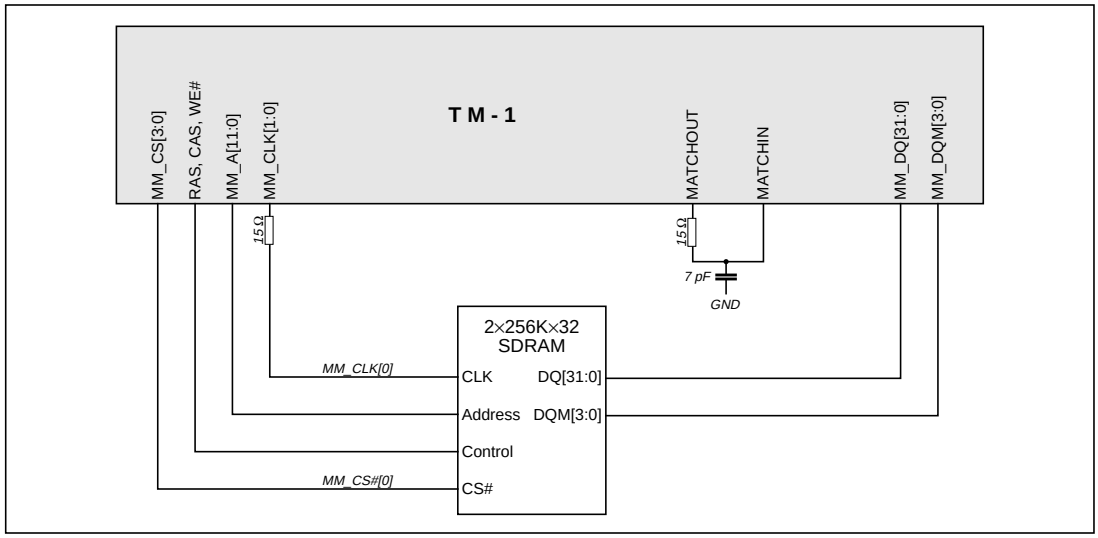


Figure 11-6. Schematic of a 2-MB memory system consisting of one 2×256K×32 SDRAM.

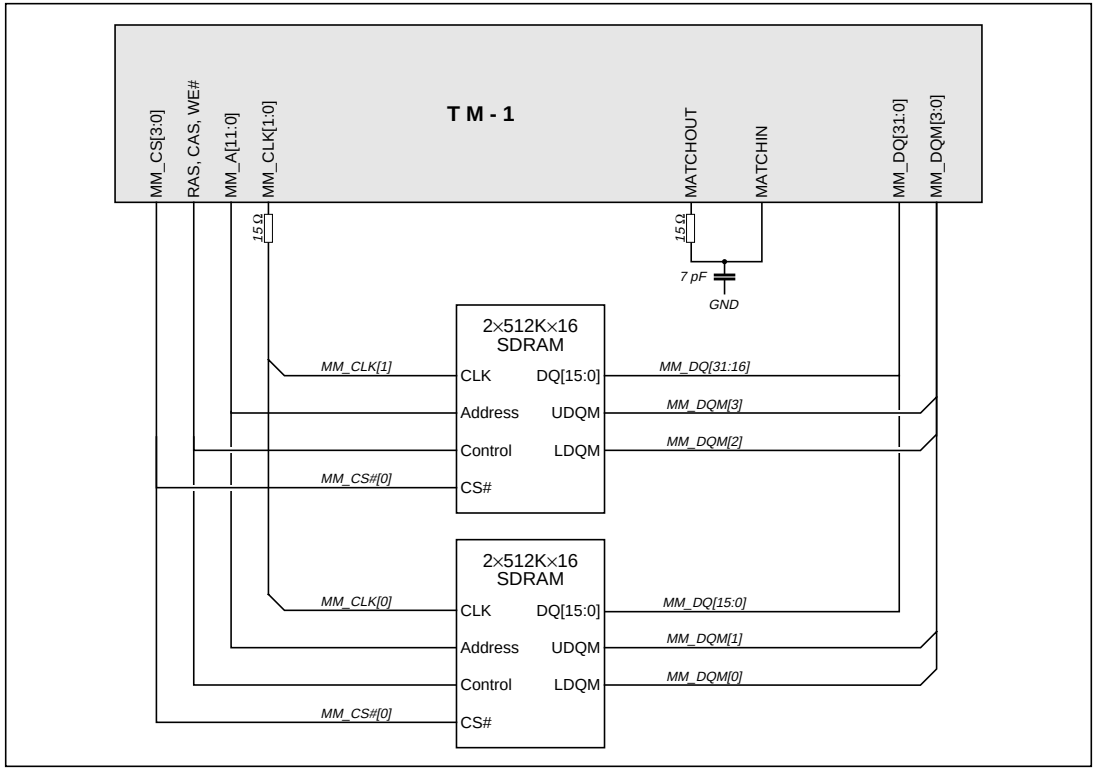


Figure 11-7. Schematic of a 4-MB memory system consisting of two 2×512K×16 SDRAM chips.

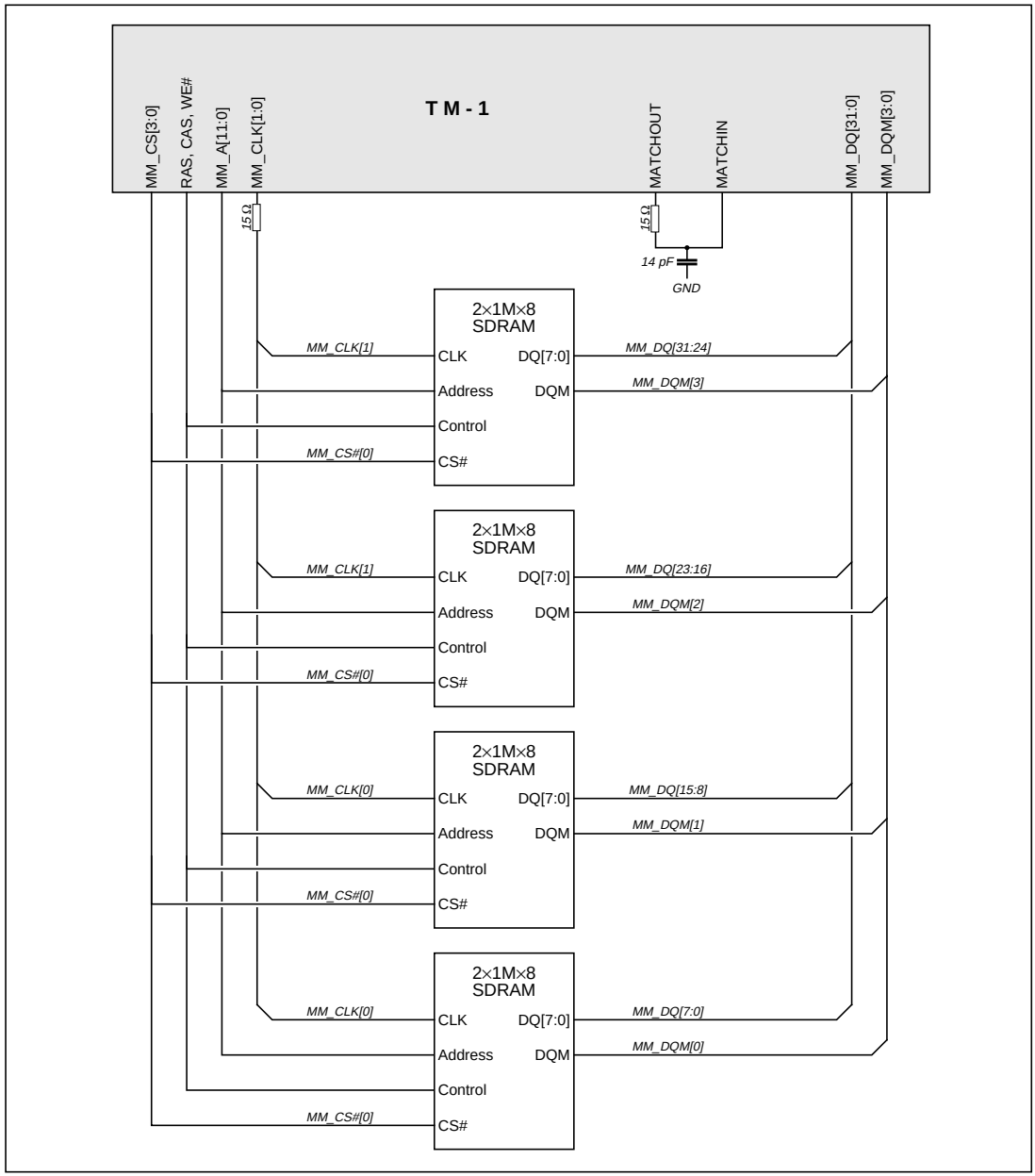


Figure 11-8. Schematic of an 8-MB memory system consisting of four 2×1M×8 SDRAM chips.

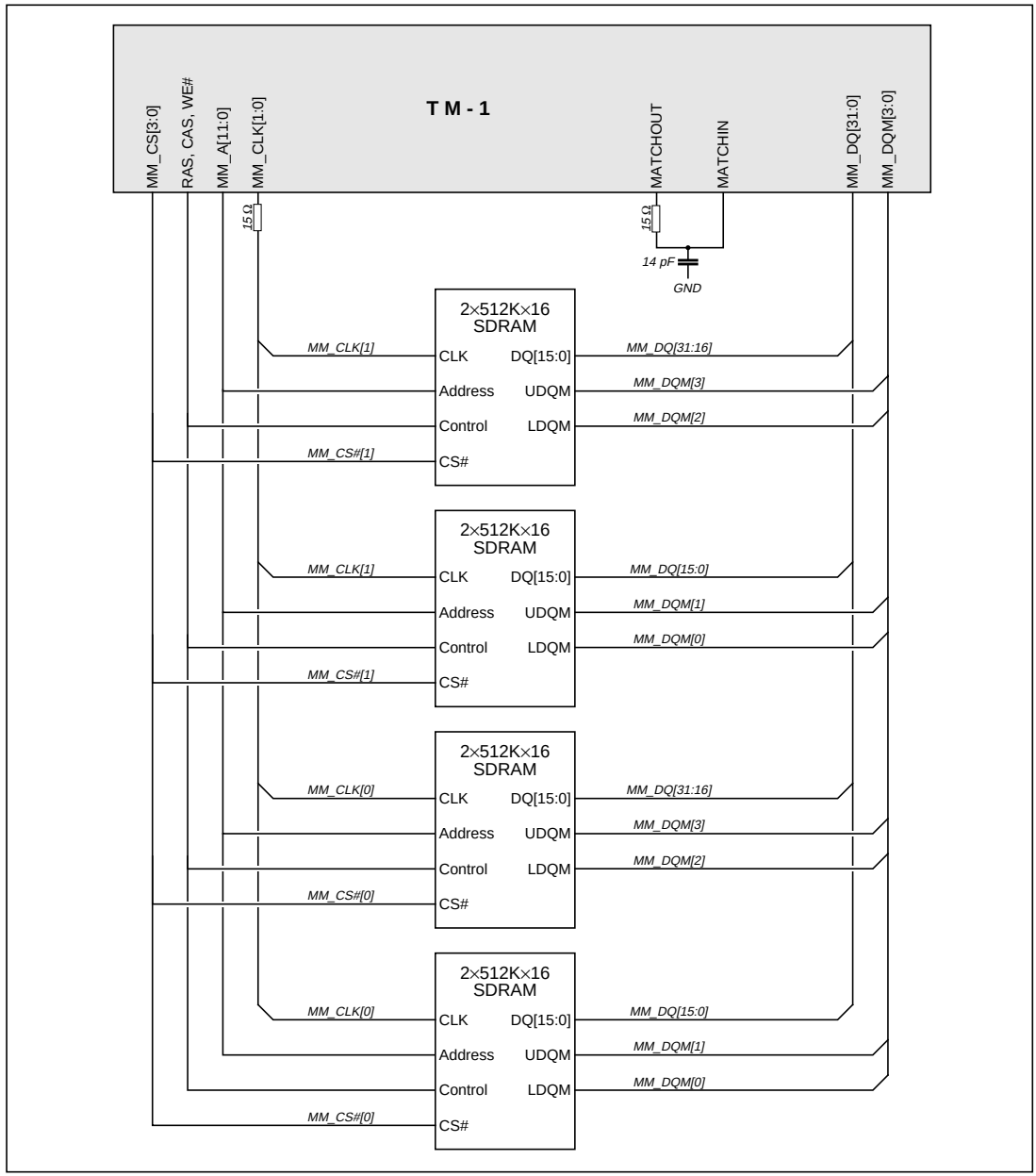


Figure 11-9. Schematic of an 8-MB memory system consisting of four 2×512K×16 SDRAM chips (two ranks)

System Boot Chapter 12

by Gert Slavenburg, Bob Bradfield, and Hani Salloum

12.1 TM1000 BOOT SEQUENCE OVERVIEW

Before a TM1000 system can begin operating, the main-memory interface registers and on-chip clock ratio register must be configured. Since the DSPCPU cannot begin operating until after these registers and circuits are initialized, the DSPCPU cannot be relied upon to initialize these resources. Consequently, TM1000 needs an independent bootstrap facility for the low-level initialization.

TM1000 implements low-level system initialization by combining a small block of on-chip system boot logic with a single external serial boot EEPROM connected to the I^2C interface. See Figure 12-1. Serial EEPROMs with an I^2C interface are slow but have the advantages of being space-efficient and inexpensive. The amount of information needed for initial system boot is small, so speed is not a concern.

The TM1000 system boot block performs differently for each of the two major types of TM1000 system. The most significant bit of the tenth byte in the external EEPROM determines the system boot procedure and must match the system configuration.

In the first type of system, host-assisted bootstrapping takes place. In this configuration, a TM1000 device is integrated into a system where some other processor serves as the host. For example, a TM1000 chip might be part of a PCI card in a standard personal computer (PC). In this case, the TM1000 system boot need only load enough information from the serial EEPROM to configure the on-chip timing circuits and main-memory interface; the host processor can perform all other TM1000 setup chores.

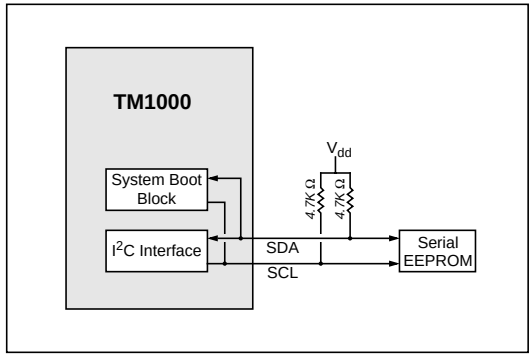


Figure 12-1. The system boot logic uses the I2C interface to access a serial EEPROM that contains main-memory and system timing information.

Table 12-1. System Boot Features

Characteristic	Comments
Boot Configurations Supported	• Host assisted, e.g., TM1000 is a PCI slave in a standard PC. • Autonomous, e.g., TM1000 is the host PCI processor.
ROM Device Types Supported	• Single standard I2C serial EEPROMs from 128 bytes to 2K bytes in size. • EEPROMs connect via the TM1000's built-in two-wire I^2C interface. • The use of EEPROMs with hardware Write Protect (WP) is recommended. A jumper on WP allows user control over in-system reprogramming using the I2C interface. • The EEPROM must respond to I^2C device address 1010.
ROM Device examples	• Atmel 24C01A (128 bytes, WP) • Atmel 24C08 (1Kbytes, WP) • Atmel 24C16 (2Kbytes, WP).
ROM size	• From 128 bytes to 2K bytes (one device) for initial program load.

In the second type of system, autonomous bootstrapping takes place. In this configuration, a TM1000 device serves as the host (main) processor; consequently, the TM1000 system boot must perform more work. In addition to configuring on-chip timing and the main-memory interface, the system boot must set the base addresses of the main-memory and MMIO address apertures and load into main memory a level 1 bootstrap program for the DSPCPU.

Only the first ten bytes of the serial EEPROM are needed when TM1000 is not the host PCI processor; thus, such systems can use a very low-cost 128-byte EEPROM device. When TM1000 serves as the system's host processor, the boot logic permits almost 2K bytes of storage for the level 1 bootstrap DSPCPU program in a single eight-pin EEPROM device.

12.2 BOOT HARDWARE OPERATION

The TM1000 boot sequence begins with the assertion of the reset signal TRI_RESET#. After reset is de-asserted,

only the system boot block, I2C, and PCI interfaces are allowed to operate. In particular, the DSPCPU and the internal data highway bus will remain in the reset state until they are explicitly released during the boot procedure. In autonomous boot, the system boot block is responsible for releasing the DSPCPU and highway from reset. In host-assisted boot, the boot logic releases the highway from reset and the TM1000 software driver (which runs on the host processor) releases the DSPCPU from reset.

The system boot block operation is illustrated in a flow chart shown in Figure 12-2.

12.2.1 Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap

There should be no other I2C master active from reset until boot EEPROM load completes. The system boot procedure begins by loading a few critical pieces of information from the serial EEPROM. This part of the procedure is common to both autonomous and host-assisted bootstrapping. See Table 12-2 for a summary and Table 12-5 for full bit accurate EEPROM layout details.

The first byte of the EEPROM is read using a serial clock equal to BOOT_CLK/1000, which is guaranteed to be less than 100 kHz. After reading the first byte, which contains the actual BOOT_CLK rate as well as the EEPROM speed capability, the boot block proceeds to read subsequent bytes at the highest valid speed.

The number of lines in the EEPROM device should be 0 in case of a 128 byte device and 1 for larger devices.

The SDRAM aperture size should be set to the smallest size that is larger than or equal to the actual size of SDRAM connected to TM1000. The SDRAM aperture size information is forwarded to the PCI interface for use in host BIOS configuration, as described in Section 12.3.2, “Stage 2: Host-System PCI Configuration.”

The BOOT_CLK speed bits should be set to match the closest rounded up frequency of the external clock circuit, i.e. for an external clock of 40 MHz or 50 MHz the value should be 10. This field, together with the EEPROM maximum clock speed bit are used to decide the best possible divider ratio for generation of the I2C clock, as shown in Table 12-3. In addition, the delay actions in Figure 12-2 are taken based on the specified BOOT_CLK value.

The EEPROM maximum clock speed bit is set to match the speed grade of the serial EEPROM device.

The test mode bit should always be set to 0. It is only set to one for factory ATE testing.

The Subsystem ID and Subsystem Vendor ID data has no meaning to the TM1000 hardware; its meaning is entirely software defined. The value is loaded by the system boot block from the EEPROM and published in the PCI configuration space register at offset 0x2C to provide the 16 bit Subsystem ID and Subsystem Vendor ID values. These values are used by driver software to distinguish the board vendor and product revision information for multiple board products based on the TM1000 chip. Refer to Section 10.5.12, “Subsystem ID, Subsystem Vendor ID Register,” for more information on the choice of values.

Table 12-2. Information Loaded During First Part of Bootstrapping Procedure

Information	Size	Interpretation	
Number of lines in EEPROM device	1 bit	0	128 lines
		1	256 or more lines
SDRAM aperture size	3 bits	000	1 MB
		001	1 MB
		010	2 MB
		011	4 MB
		100	8 MB
		101	16 MB
		110	32 MB
		111	64 MB
BOOT_CLK speed	2 bits	00	100 MHz
		01	75 MHz
		10	50 MHz
		11	33 MHz
EEPROM maximum clock speed	1 bit	0	100 KHz
		1	400 KHz
Test mode	1 bit	0	normal operation
		1	rapid ATE testing
Subsystem ID	16 bits	Value is copied to Subsystem ID register in PCI configuration space.	
Subsystem Vendor ID	16 bits	Value is copied to Subsystem Vendor ID register in PCI config space.	
MM_CONFIG register initialization	20 bits	Value is simply written to the MM_CONFIG register; see Section 11.5.1, “MM_CONFIG Register.”	
PLL_RATIOS register initialization	8 bits	Value is simply written to the PLL_RATIOS register; see Section 11.5.2, “PLL_RATIOS Register.”	
Autonomous/host-assisted boot	1 bit	0	host-assisted
		1	autonomous

Table 12-3I2C speed as a function of EEPROM byte 0

BOOT_CLK bits	EEPROM speed bit	divider value	actual I2C speed
00 (100 MHz)	0 (100 kHz)	1040	97 kHz
00	1 (400 kHz)	272	368 kHz
01 (75 MHz)	0 (100 kHz)	784	96 kHz
01	1 (400 kHz)	208	360 kHz
10 (50 MHz)	0 (100 kHz)	528	95 kHz
10	1 (400 kHz)	144	347 kHz
11 (33 MHz)	0 (100 kHz)	352	94 kHz
11	1 (400 kHz)	112	295 kHz

The MM_CONFIG and PLL_RATIOS registers control the hardware of the main-memory interface and TM1000 on-chip clock circuits. These registers are described in detail in Section 11.5, “Memory System Programming.” The boot value should be set to reflect the exact capabilities of the actual SDRAM in the system.

The autonomous/host-assisted boot bit determines whether the system boot logic will continue reading more information from the EEPROM or halt its operation so the host can complete system initialization. After the information listed in Table 12-2 has been loaded into TM1000 registers, an external PCI host processor can finish the initialization of TM1000. If no external PCI host processor is present, the autonomous/host-assisted boot bit should be set to one to allow the system boot logic to load the information described in the next section.

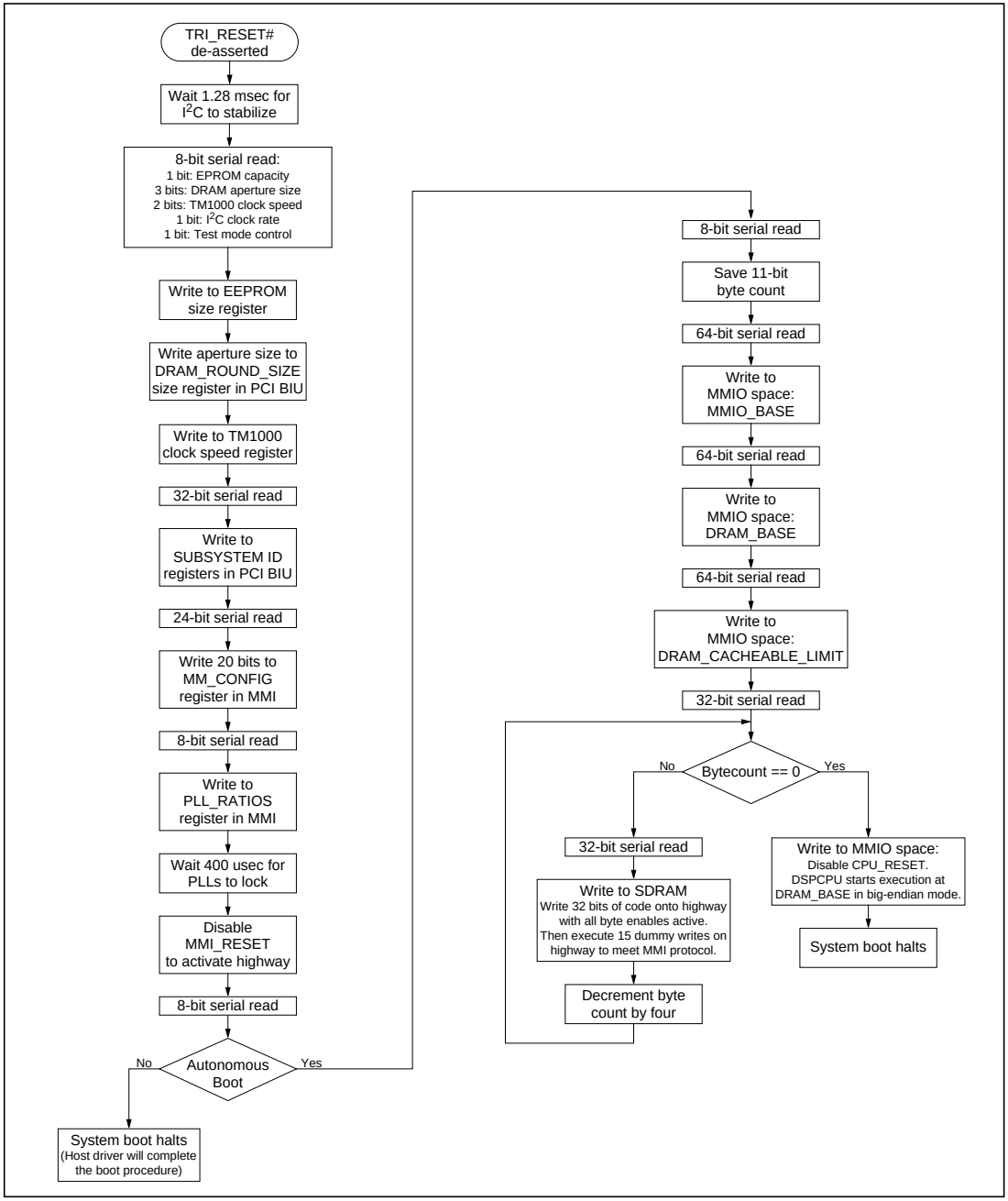


Figure 12-2. Flow chart of system boot procedure for both host-assisted and autonomous configurations.

12.2.2 Initial DSPCPU Program Load for Autonomous Bootstrap

In a system where TM1000 serves as the host CPU, the system boot block performs an autonomous boot procedure. For an autonomous boot, the system boot block reads all the information described in Section 12.2.1, "Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap," and then—because the autonomous boot bit is set—continues reading information from the EEPROM. After this part of the system boot procedure is done, the DSPCPU starts executing. See Table 12-4.

The DSPCPU bootstrap program byte count encodes the number of bytes of DSPCPU program code contained in the EEPROM(s). This eleven-bit unsigned byte count can encode up to 2048 bytes, which is also the maximum amount of EEPROM storage supported. The actual amount of EEPROM available for the DSPCPU bootstrap program is limited to 2000 bytes because the other information consumes 47 bytes and the DSPCPU code must be an integral number of 32-bit words.

Four pairs of 32-bit MMIO-register addresses and values follow the bootstrap program byte count. Each address tells the boot block where in the 32-bit DSPCPU address space to store the corresponding 32-bit value.

The first pair initializes the MMIO_BASE. The MMIO_BASE sets the base address of the 2-MB MMIO-register address aperture within the DSPCPU 32-bit address space. All MMIO registers are addressed using an offset that is relative to the value of MMIO_BASE. For this pair, the address is required to be 0xEFF00400 because that is the default MMIO_BASE enforced when TM1000 is reset. The new value for MMIO_BASE is encoded in the corresponding value.

The DRAM_BASE address/value pair determine the base address of the SDRAM address aperture within the 32-bit DSPCPU address space. The address must be equal to 0x100000 plus the new value of MMIO_BASE set previously in the boot procedure. The DRAM_BASE value must be naturally aligned given the rounded DRAM aperture size, i.e. a 6 MByte DRAM aperture should start on a 8M address multiple.

The DRAM_LIMIT address/value pair determine the extent of the SDRAM address aperture. The address must be equal to 0x100004 plus the new value of MMIO_BASE set previously in the boot procedure. The value in DRAM_LIMIT should be 1 higher than the address of the last valid byte of SDRAM memory, and must be a 64 kByte multiple.

The DRAM_CACHEABLE_LIMIT address/value pair determine the extent of the cacheable aperture of the SDRAM address space. The address must be equal to 0x100008 plus the value of MMIO_BASE set previously in the boot procedure. The cacheable aperture always begins at the address value in DRAM_BASE; the value in DRAM_CACHEABLE_LIMIT is one higher than the address of the last byte of cacheable SDRAM memory, and must be a 64 kByte multiple. It is safe to initially set the value of DRAM_CACHEABLE_LIMIT equal to DRAM_LIMIT. The RTOS can, if desired, change the value later.

Table 12-4. Information Loaded During Second Part of Bootstrapping Procedure for Autonomous Boot

Information	Size	Interpretation
DSPCPU bootstrap program byte count *n*	11 bits	up to 500 32-bit words (2048 bytes less 47 header bytes)
MMIO_BASE address	32 bits	Value must be 0xEFF00400
MMIO_BASE value	32 bits	Value is simply written to 0xEFF00400 to determine new base address of 2-MB MMIO register aperture within 32-bit DSPCPU address space
DRAM_BASE address	32 bits	MMIO_BASE + 0x100000
DRAM_BASE value	32-bits	Value is simply written to DRAM_BASE to determine base address of SDRAM aperture within 32-bit DSPCPU address space
DRAM_LIMIT address	32-bits	MMIO_BASE + 0x100004
DRAM_LIMIT value	32-bits	Value is simply written to DRAM_LIMIT to determine limit address of SDRAM aperture within 32-bit DSPCPU address space
DRAM_CACHEABLE_LIMIIT address	32-bits	MMIO_BASE + 0x100008
DRAM_CACHEABLE_LIMIT value	32-bits	Value is simply written to DRAM_CACHEABLE_LIMIT to determine limit address of cacheable part of SDRAM aperture within 32-bit DSPCPU address space
DRAM_BASE value	32-bits	Copy of the DRAM_BASE; must be equal to value specified above
SDRAM code word 0	32-bits	First 32-bit word of initial DSPCPU bootstrap program
SDRAM code word 1	32-bits	Second 32-bit word of initial DSPCPU bootstrap program
. . .	. . .	. . .
SDRAM code word *n*/4	32 bits	Last 32-bit word of initial DSPCPU bootstrap program

The next 32-bit value in boot EEPROM memory is a copy of the DRAM_BASE value encoded previously. The system boot hardware loads the DSPCPU bootstrap program into SDRAM starting at DRAM_BASE.

The bytes of the DSPCPU bootstrap program follow the copy of the SDRAM_BASE value. The bootstrap program can consist of up to 500 32-bit words of DSPCPU

instructions. The byte count must be a multiple of four. Note that the bytes are stored in the EEPROM in a byte swapped order per group of 4 compared to SDRAM, as detailed in Table 12-5.

After the entire DSPCPU bootstrap program is loaded into SDRAM at DRAM_BASE, the system boot logic releases the DSPCPU from the reset state. At this point, the DSPCPU begins executing the bootstrap program starting at DRAM_BASE and TM1000 is fully operational. At the same time, the boot logic releases the I^2C interface.

12.3 HOST-ASSISTED BOOT DESCRIPTION

For a host-assisted bootstrap, the complete bootstrap process consists of three distinct stages, but the system boot hardware performs only the first stage. The other two stages are the responsibility of the host system.

12.3.1 Stage 1: TM1000 System Boot Hardware

In the first stage, the TM1000 hardware must be initialized enough to allow the host system to query and manipulate TM1000 resources. The system boot hardware, using the procedure described above in Section 12.2.1, "Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap," initializes the Subsystem ID, Subsystem Vendor ID, MM_CONFIG, and PLL_RATIOS registers, waits for the PLLs to lock, enables the internal highway and main-memory interface (MMI), but leaves the DSPCPU in the reset state. After this minimal initialization, the host system can finish the bootstrap process.

At the completion of stage 1, the TM1000 hardware is ready to respond to PCI configuration space accesses, and the boot block has released the I^2C interface.

12.3.2 Stage 2: Host-System PCI Configuration

Stage 2 is carried out either by the host-system PCI BIOS or by a combination of the BIOS and the host operating system (e.g., Windows 95). During this stage, the host system configures all PCI-bus clients.

The PCI-bus configuration consists of querying the bus clients to determine the following:

- The number of PCI base-address registers implemented by each client. For TM1000, the number of PCI base-address registers is always two (MMIO_BASE and DRAM_BASE).
- The size of each aperture associated with the base-address registers. For TM1000, the size of the MMIO aperture is always 2 MB, while the size of the SDRAM aperture can be from 1 MB to 64 MB with the constraint that the size must be a power of two (seven distinct sizes).

Using this information, the host system relocates each address aperture to eliminate overlaps in the PCI address space. The host system accomplishes the relocation by considering each apertures size and then writing an appropriate starting address to each base-address register. For TM1000, the base addresses of the MMIO and SDRAM apertures must be relocated in this way. Note that in the case of autonomous boot, this relocation is done statically by the system boot hardware when it simply copies the values of MMIO_BASE and DRAM_BASE from the serial EEPROM into these registers.

The steps of the PCI protocol for determining the size of an address aperture are as follows (see Section 10.5.11, "Base Address Registers," for a more complete discussion):

- The host writes a 32-bit word of all ones (0xffffffff) to the base-address register.
- The host reads the base-address register immediately after the write. The value returned will have zeros in all don't-care bits and ones in all required address bits. The required address bits form a left-aligned (i.e., starting at the most-significant bit) contiguous field of ones.
- This left-aligned field of ones effectively specifies the size of the address aperture by indicating the bits of the base-address register that are significant for relocation. That is, an address aperture of size 2^n can only begin on a 2^n-byte-aligned boundary.

As an example, consider the case of the MMIO aperture. The host will perform the following steps during stage 2 of the bootstrap process:

- Write 0xffffffff to MMIO_BASE.
- Read from MMIO_BASE, which returns the value 0xffe00000. The host sees that this value has an 11-bit left-aligned field of ones, which indicates that the aperture can only be relocated on 2-MB boundaries; thus, the aperture size is 2 MB.
- Write a new value to MMIO_BASE with the top 11 bits set to relocate the MMIO aperture to a 2-MB region of PCI address space that does not conflict with other PCI address apertures.

At the completion of stage 2, the TM1000 hardware is ready to respond to host configuration space accesses, host MMIO accesses and host SDRAM aperture accesses. The DSPCPU is still in RESET state.

12.3.3 Stage 3: TM1000 Driver Executing on the Host

During the final stage of the bootstrap process, the TM1000 software driver executing on the host system will write to SDRAM a program for the DSPCPU, and set any MMIO registers as it sees fit. When the initial program load is complete, the driver releases the DSPCPU from its reset state by a write to the BIU_CTL register with the CR bit set. See Chapter 10, "PCI Interface." Now, with the DSPCPU and host both running, the TM1000 bootstrap process is complete.

12.4 DETAILED EEPROM CONTENTS

Table 12-5 shows the serial EEPROM contents needed for an autonomous boot procedure. For the host-assisted boot procedure, only the contents up to line nine are needed.

Note that the 32-bit words in the serial EEPROM are not stored on 32-bit word-aligned addresses.

Table 12-5. Serial Boot EEPROM Contents

Line	Data Byte							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	#lines 0: 128 lines 1: 256 or more lines	SDRAM size[2:0] 000: 1MB 001: 1MB 010: 2MB 011: 4MB 100: 8MB 101: 16MB 110: 32MB 111: 64MB			BOOT_CLK[1:0] 00: 100 MHz 01: 75 MHz 10: 50 MHz 11: 33 MHz		EEPROM clock 0: 100 KHz 1: 400 KHz	Test Mode 0: normal 1: rapid ATE
1	Subsystem ID, 8 msb							
2	Subsystem ID, 8 lsb							
3	Subsystem Vendor ID, 8 msb							
4	Subsystem Vendor ID, 8 lsb							
5	—	—	—	—	MM_CONFIG[19:16]			
6	MM_CONFIG[15:8]							
7	MM_CONFIG[7:0]							
8	PLL_RATIOS[7:0}							
	sdram PLL bypass	sdram PLL disable	cpu PLL bypass	cpu PLL disable	sdram ratio	cpu ratio[2:0]		
9	boot type 0: host assist. 1: autonomous	—	—	—	—	byte count [10:8]		
10	byte count [7:0]							
11	MMIO_BASE address [31:24] (must be 0xEF)							
12	MMIO_BASE address [23:16] (must be 0xF0)							
13	MMIO_BASE address [15:8] (must be 0x04)							
14	MMIO_BASE address [15:8] (must be 0x00)							
15	MMIO_BASE value [31:24]							
16	MMIO_BASE value [23:16]							
17	MMIO_BASE value [15:8]							
18	MMIO_BASE value [7:0]							
19	DRAM_BASE address [31:24] (must be byte 3 of MMIO_BASE + 0x100000)							
20	DRAM_BASE address [23:16] (must be byte 2 of MMIO_BASE + 0x100000)							
21	DRAM_BASE address [15:8] (must be byte 1 of MMIO_BASE + 0x100000)							
22	DRAM_BASE address [7:0] (must be byte 0 of MMIO_BASE + 0x100000)							
23	DRAM_BASE value [31:24]							
24	DRAM_BASE value [23:16]							
25	DRAM_BASE value [15:8]							
26	DRAM_BASE value [7:0]							
27	DRAM_LIMIT address [31:24] (must be byte 3 of MMIO_BASE + 0x100004)							
28	DRAM_LIMIT address [23:16] (must be byte 2 of MMIO_BASE + 0x100004)							
29	DRAM_LIMIT address [15:8] (must be byte 1 of MMIO_BASE + 0x100004)							
30	DRAM_LIMIT address [7:0] (must be byte 0 of MMIO_BASE + 0x100004)							
31	DRAM_LIMIT value [31:24]							
32	DRAM_LIMIT value [23:16]							
33	DRAM_LIMIT value [15:8]							
34	DRAM_LIMIT value [7:0]							
35	DRAM_CACHEABLE_LIMIT address [31:24] (must be byte 3 of MMIO_BASE + 0x100008)							
36	DRAM_CACHEABLE_LIMIT address [23:16] (must be byte 2 of MMIO_BASE + 0x100008)							
37	DRAM_CACHEABLE_LIMIT address [15:8] (must be byte 1 of MMIO_BASE + 0x100008)							
38	DRAM_CACHEABLE_LIMIT address [7:0] (must be byte 0 of MMIO_BASE + 0x100008)							

Table 12-5. Serial Boot EEPROM Contents

Line	Data Byte							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
39 40 41 42	DRAM_CACHEABLE_LIMIT value [31:24] DRAM_CACHEABLE_LIMIT value [23:16] DRAM_CACHEABLE_LIMIT value [15:8] DRAM_CACHEABLE_LIMIT value [7:0]							
43 44 45 46	repeat of DRAM_BASE value [31:24] repeat of DRAM_BASE value [23:16] repeat of DRAM_BASE value [15:8] repeat of DRAM_BASE value [7:0]							
47 48 49 50	byte 0 of DSPCPU bootstrap program (stored at DRAM_BASE + 3) byte 1 of DSPCPU bootstrap program (stored at DRAM_BASE + 2) byte 2 of DSPCPU bootstrap program (stored at DRAM_BASE + 1) byte 3 of DSPCPU bootstrap program (stored at DRAM_BASE + 0)							
. . .	. . .							
j+47	byte j of DSPCPU bootstrap program (stored at DRAM_BASE + ((j div 4) + (3 – (j mod 4))))							
. . .	. . .							
(n–1) +47	last byte of DSPCPU bootstrap program (bits [7:0] of last 32-bit word, stored at DRAM_BASE + n – 4)							

12.5 I²C PROTOCOL FOR EEPROM ACCESS

Figure 12-3 shows the SDA (serial data) line protocols for three types of read accesses supported by I²C serial EEPROMs. A read from the address currently latched inside the EEPROM can be for either a single byte or for an arbitrary series of sequential bytes. The master makes the choice by setting the ACK bit after a byte has been transferred.

A random-access read is accomplished by performing a dummy write, which overwrites the latched address stored inside the EEPROM. Once the internal address latch is set to the desired value, one of the other two read protocols can be used to read one or more bytes.

The boot logic inside TM1000 uses a single random read transaction to location 0 of device address 1010000 followed by a sequential read extension to read all required EEPROM bytes in a single pass.

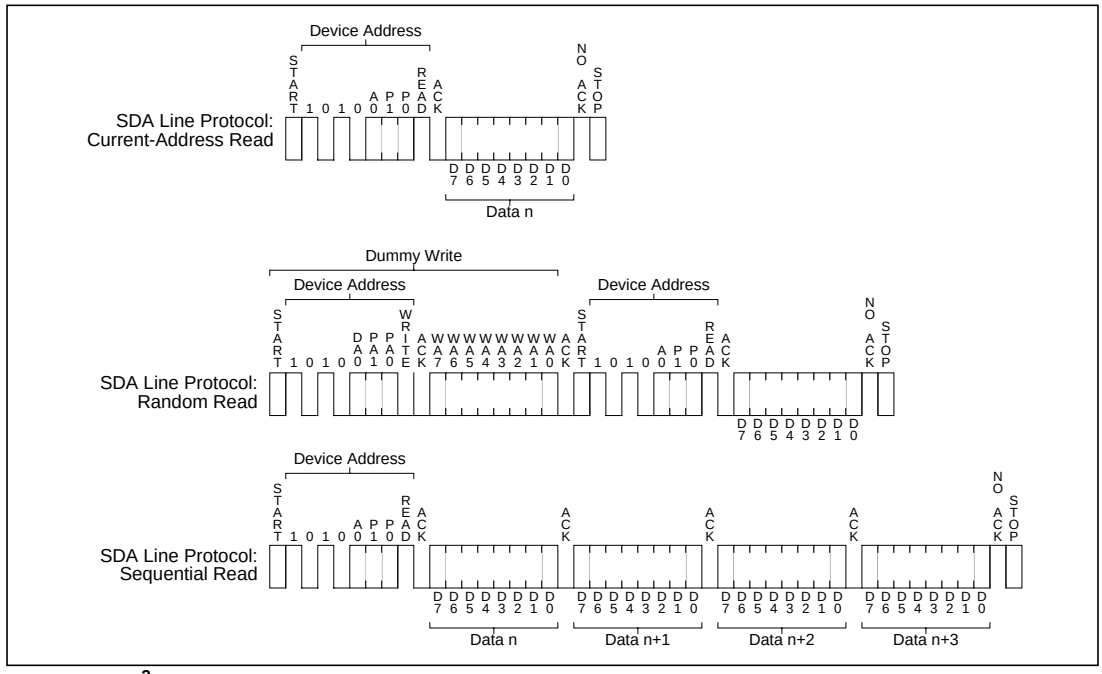


Figure 12-3. I²C protocol for three types of EEPROM access. In the diagrams, a label is shown on top of a data bit window to indicate the SDA line is driven by the master (TM1000), and a label is shown on the bottom to indicate that the SDA line is driven by the EEPROM.

13.1 SUMMARY FUNCTIONALITY

The Image Co-Processor (ICP) connects to the TM1000 on-chip data highway to perform SDRAM block read and write actions. It also connects to the PCI interface to allow block write transactions across PCI.

The major functions of the Image Co-Processor are:

- Move an image by reading the image from SDRAM and writing it back to SDRAM.
- Filer an image by reading the image from SDRAM and writing the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in horizontal direction.
- Filter an image by reading the image from SDRAM and writing the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in vertical direction.
- Filter an image and convert it from planar to RGB or YUV composite by reading the image from SDRAM and writing the image out to PCI bus memory (graphics card) or SDRAM, while performing horizontal scaling and conversion to one of a several RGB and YUV formats. The user can add optional bitmap masking to selectively enable/disable pixel writes to PCI (to refresh only the exposed part of a video window) and an optional image overlay with alpha blending and optional chroma keying (PCI output only).

All of the Image Co-Processor functions move and transform data from memory to memory or memory to the PCI bus. Hence, the DSPCPU can use the ICP in a time-sharing fashion to simultaneously achieve:

1. Vertical and horizontal resizing/subsampling on the stream of images from Video In.
2. Vertical and horizontal resizing/upsampling on the stream of images sent to Video Out.
3. Presentation of a collection of live video windows with programmable up and down scaling and arbitrary overlap configuration on PCI graphics cards.[1]

Full two dimensional scaling and filtering requires two passes over the data: one to do horizontal scaling and filtering and one to do vertical scaling and filtering.

Figure 13-1 shows a block diagram of the TM1000 with the Image Co-Processor (ICP). Figure 13-2 shows a block diagram of the internal structure of the Image Co-Processor. The ICP contains a 5-tap filter, YUV to RGB converter, an overlay and alpha blending unit, and an output formatter. These blocks communicate with each other and communicate with the TM1000 SDRAM Data Highway through a bank of FIFOs. The FIFOs buffer the block data to and from the TM1000 SDRAM Data Highway. The ICP uses a microprogram controlled sequencer to control its internal timing. The program for this sequencer is in a table in SDRAM. The ICP reads the appropriate portion from the SDRAM each time the ICP is commanded to perform a function. Microprogram control simplifies and minimizes the ICP hardware and increases the flexibility of the ICP to do additional tasks without adding hardware.

1. Note that function 2 and 3 don't normally occur simultaneously, and if an application attempts both simultaneous some performance limitations are incurred.

13.2 REQUIREMENTS

13.2.1 Functions

The major functions of the Image Co-Processor are:

1. Read an image from SDRAM and write the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in horizontal direction.
2. Read an image from SDRAM and write the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in vertical direction.
3. Read an image from SDRAM and write the image out to PCI bus memory (graphics card) or SDRAM, while performing horizontal scaling and conversion to one of a several RGB and YUV formats. The PCI output mode includes optional bitmap masking to selectively enable/disable pixel writes to PCI (to refresh only the exposed part of a video window) and optional RGB overlay with alpha blending and optional chroma keying.

13.2.2 Bandwidth

The bandwidth for the ICP can be estimated from the worst case image processing bandwidth. If the worst case image is 1024 x 768 at 30 Hz in YUV 4:2:2 format, the pixel rate is 1024 x 768 x 30 = 23.59 megapixels per second. For YUV 4:2:2 image coding at 2 bytes per pixel, this is 23.59 x 2 = 47.19 megabytes per second. The minimum bandwidth for the ICP function is therefore 47.18 megabytes per second, or approximately 50 megabytes per second.

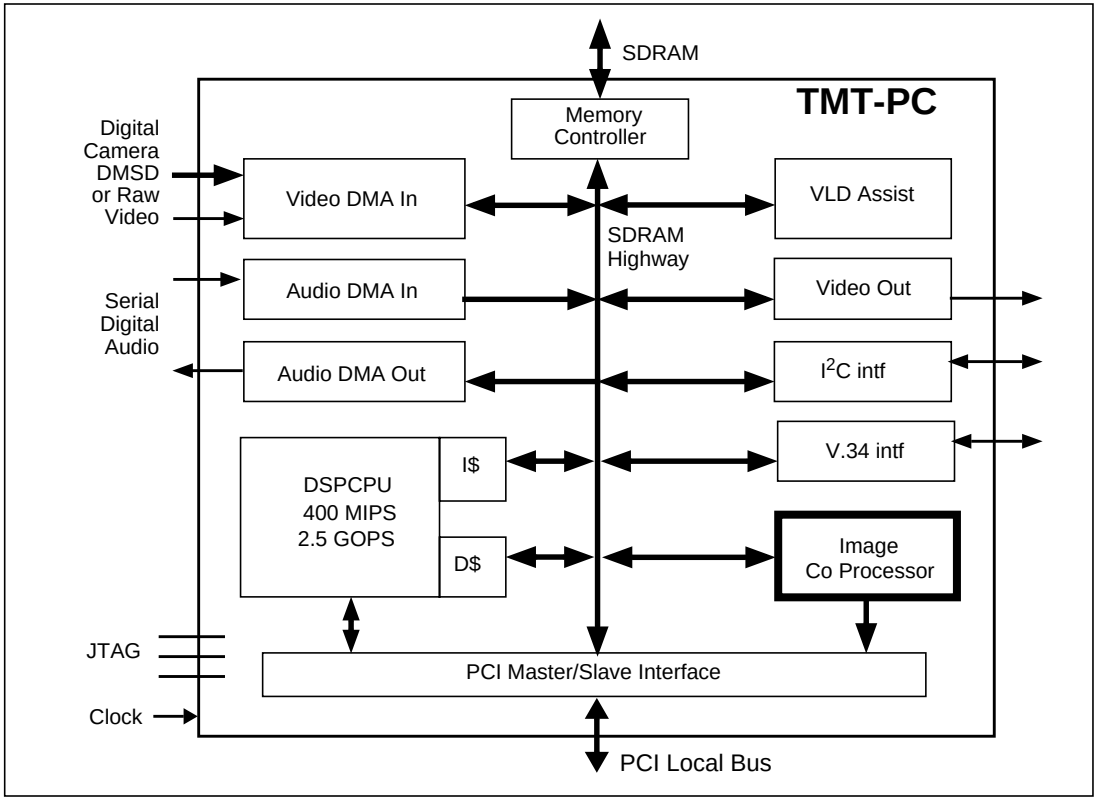


Figure 13-1. TM1000 Chip Block Diagram

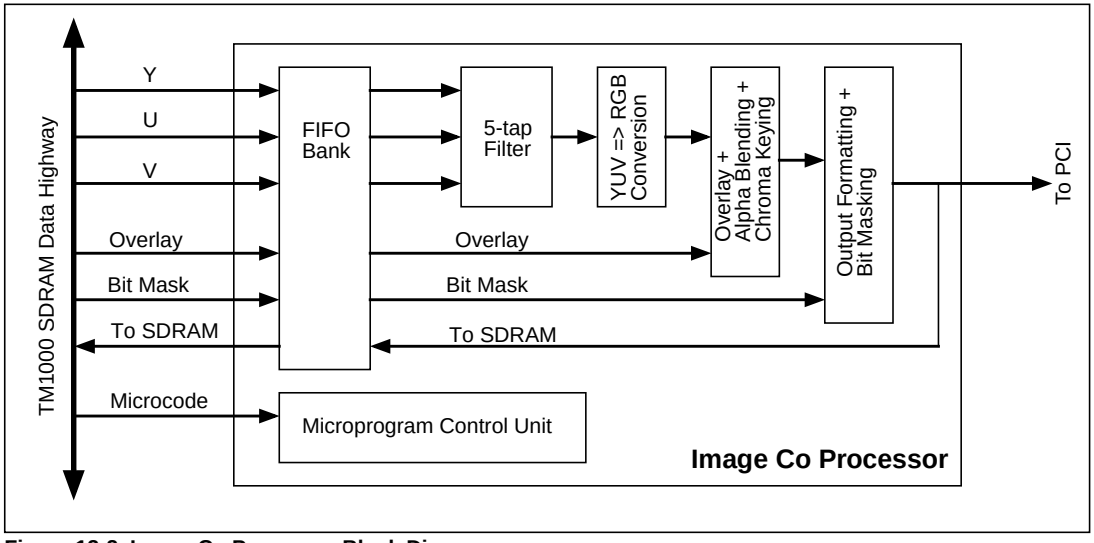


Figure 13-2. Image Co Processor Block Diagram

Scaling and filtering of the two dimensional image requires two passes of the image data through the filter, one for vertical and one for horizontal. Scaling an image and sending it to the PCI bus requires three transfers of the image over the SDRAM bus: one transfer to read the image for vertical filtering, one transfer to write the filtered data back, and one transfer to read the image for horizontal filtering and output to the PCI bus. This means an average of SDRAM bus bandwidth of 3 x 50 = 150 megabytes/second for the 1024 x 768 image case described above, assuming a scaling factor of 1.0. A larger or smaller scaling factor means that either the input or output image will be smaller than 1024 x 768. The bandwidths required are determined by the larger of the two images, input or output. This is because all input pixels must be scanned to generate all the output pixels. Scaling and filtering the image back to the SDRAM requires an additional transfer to write the horizontally filtered image back to SDRAM.

13.2.3 Image Size and Scaling

Image sizes in the TM1000 have a nominal range of 16 x 16 to 1024 x 768. Sizes smaller than 16 x 16 are possible, but are too small to be recognizable images. Images larger than 1024 x 768 (up to 64K x 64K) are possible but cannot be processed in real time. They also require larger SDRAM size to support them. Scaling factors have a nominal range of 1/4 (down scaling by 4) to 4 (upscaling by 4). Larger up and down scaling factors are possible, up to 1000 and beyond; however, very large upscaling factors result in a large magnification of a few pixels, and very large down scaling factors give only a few pixels as a result.

13.3 INTERFACE

The Image Co-Processor block has no TM1000 external pins. It interfaces internally to the SDRAM Data Highway and the PCI output.

13.4 DATA FORMATS

The Image Co-Processor block accepts input and overlay image data to generate output image data. The ICP accommodates a variety of formats for the input, overlay and output data. These image data formats define the relationship between the Y, U and V or the R, G, and B components of the image as they are stored in memory. The ICP accepts input image data in planar format, where the Y, U and V components are in separate tables in SDRAM. The various input image data formats differ in the position of the U and V components relative to the Y component and the amount of U and V data relative to the Y data.

In all modes except the YUV to RGB conversion modes, each ICP operation processes one Y, U or V image component. Three separate commands are required to process all three components of an image. Since each component is scaled and filtered separately, the calling software defines the image format and format conversion by how it scales each component.

In the YUV to RGB conversion to PCI output or SDRAM output mode, each output pixel is a combination of RGB or YUV components as defined by the output format. The YUV input data and the RGB or YUV overlay data are combined by the ICP hardware pixel by pixel to form the RGB or YUV output pixels. Because all three YUV components are simultaneously woven together to create each output pixel, the ICP hardware must know the image data format in SDRAM, defined as how the components of the image data are to be found and combined.

In the YUV to RGB conversion mode, the ICP accepts the following input data formats: YUV 4:2:2 cosited, YUV 4:2:2 interspersed and YUV 4:2:0. In the YUV to RGB conversion mode, the ICP also accepts image overlay data when PCI output is specified. The ICP accepts image overlay data in several combined formats: RGB-24+α, RGB15+α and YUV 4:2:2+α. In this mode, the ICP generates RGB or YUV output data in several RGB and YUV formats. These formats are compatible with a wide variety of PCI frame buffers.

13.4.1 Image Input Formats

The ICP image input formats define the relative positions of the Y component and the U and V components of the input image pixel data. There are three input formats to the ICP: 4:2:2 co-sited, 4:2:2 interspersed, and 4:2:0 interspersed. The 4:2:2 formats have 2 U and 2 V pixels for every 4 Y pixels, so the ratio of Y to U or V is 2:1. The 4:2:0 format has 1 U and 1 V pixel for every 4 Y pixels, so the ratio of Y to U or V is 4:1. The input formats are given below. The input formats have a significant impact on the 2 dimensional scaling operation.

13.4.1.1 YUV 4:2:2 Co-Sited

In the YUV 4:2:2 co-sited format, the U and V pixels coincide with the Y pixel on every other pixel, as shown in Figure 13-3.

13.4.1.2 YUV 4:2:2 Interspersed

In the YUV 4:2:2 interspersed format, the U and V pixels lie between the Y pixels on every other pixel of the horizontal line, as shown in Figure 13-4.

13.4.1.3 YUV 4:2:0 XY Interspersed

In the YUV 4:2:0 interspersed format, the U and V pixels lie between the Y pixels on every other pixel of the horizontal line, as shown in Figure 13-5.

13.4.1.4 YUV 4:1:1 Co-Sited

In the YUV 4:1:1 co-sited format, the U and V pixels coincide with the Y pixel on every fourth pixel, as shown in Figure 13-6.

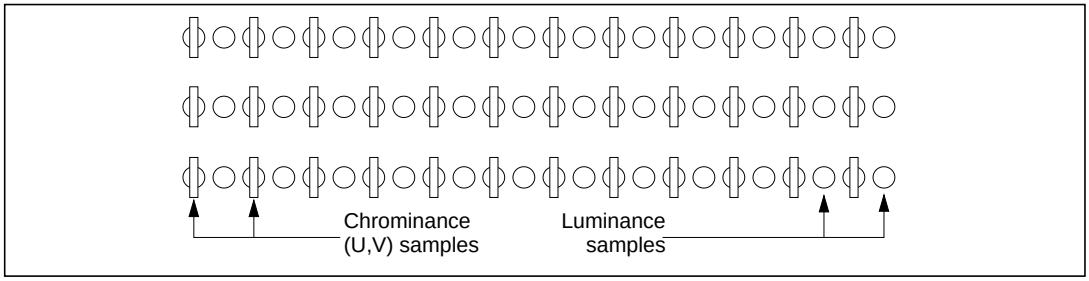


Figure 13-3. 4:2:2 Co-Sited Input Format

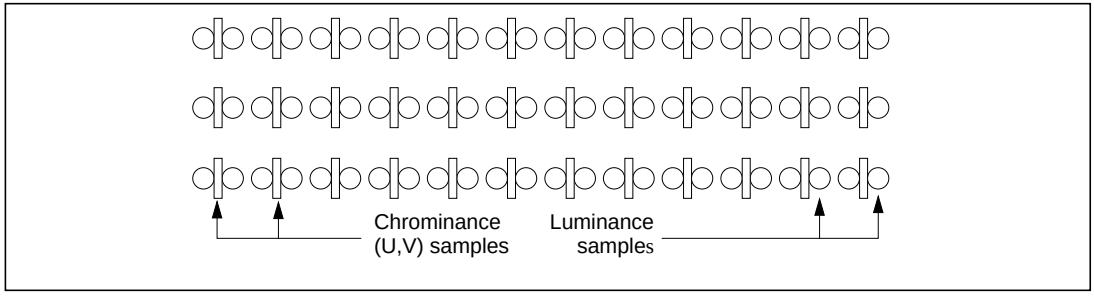


Figure 13-4. 4:2:2 Interspersed Input Format

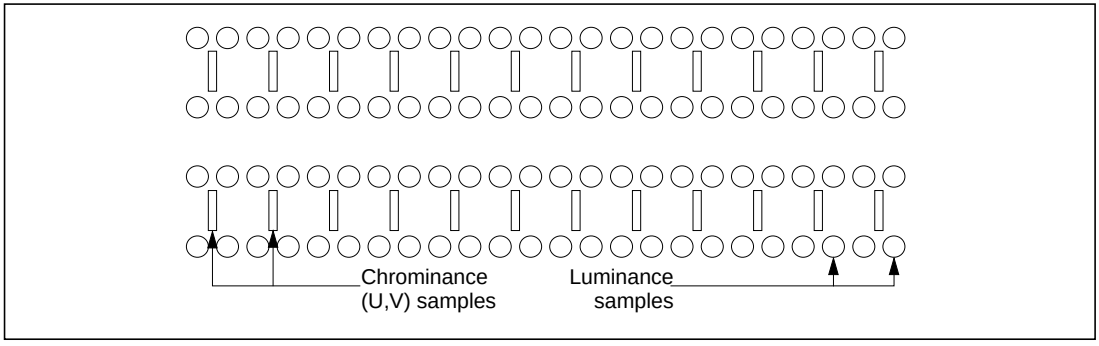


Figure 13-5. 4:2:0 XY Interspersed Input Format

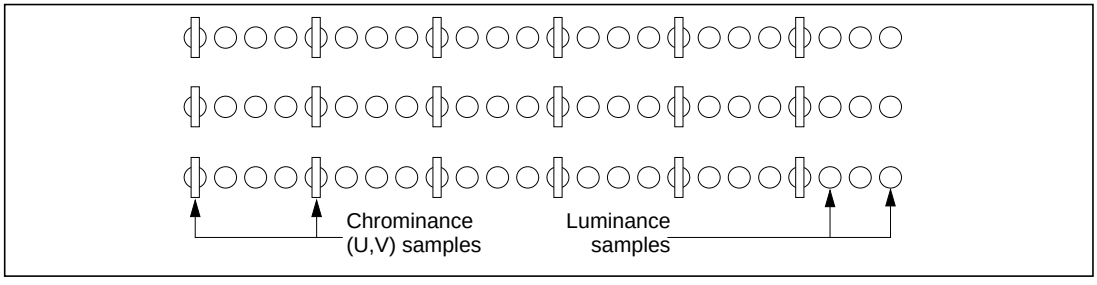


Figure 13-6. 525-60 YUV 4:1:1 Co-Sited Input Format

13.4.2 Image Overlay Formats

The ICP accepts image overlay data in three formats, RGB-24+α, RGB-15+α and YUV-4:2:2+α as shown in Table 13-1. The overlay image format must be the same type as the output image format generated by the ICP for the main image. If the output image is one of the RGB formats, the overlay must be one of the two RGB overlay formats, RGB-24-α and RGB-15+α. If the output image format is YUV, the overlay format must be in YUV-4:2:2+α format. The formats must be of the same type because the ICP does no conversion on the overlay data.

RGB-24+α, a full byte of alpha information is included with each pixel. In RGB-15+α, one bit of alpha is included for each pixel. The pixels are packed 2 pixels per word, and the alpha bit is the most significant bit of each pixel. In the same manner, the YUV-4:2:2+α format packs two pixels into one word, and it has one bit of alpha for each pixel. The least significant bit (LSB) of the U and V components supplies the alpha bit for the Y0 and Y1 pixels, respectively. The alpha bit in these formats selects between two alpha values stored in the ICP, alpha 1 and alpha 0. The alpha 1 and alpha 0 values are loaded from the parameter block when the ICP is started.

Table 13-1. Image Overlay Formats

Format	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
RGB 24+α	a7 - a0	r7 - r0	g7 - g0	b7 - b0
YUV-4:2:2+α	Y1	(v7-v1) + α	Y0	(u7-u1) + α
	Pixel 1		Pixel 0	
RGB 15+α	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0

13.4.3 Alpha Blending Codes

Image overlay uses alpha blending, which combines the overlay image with the main image according to the alpha value. The alpha value is supplied by the alpha byte in RGB 24+α format and by the alpha registers, Alpha Zero and Alpha One in the other formats. The alpha code format is shown in Table 13-2.

Table 13-2. Alpha Blending Codes

Alpha Code	Alpha Value	Image	Overlay
00h	0	100%	0%
20h	32	75%	25%
40h	64	50%	50%
60h	96	25%	75%
80h - FFh	128-255	0%	100%

13.4.4 Output Formats

The output formats are the RGB image formats sent to the PCI interface or SDRAM. These formats are shown in Table 13-3. Note: B1 = Byte 1 of blue = $[b7...b0]_1$.

Table 13-3. Output Data Formats

Format	Word	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
		Pixel 3	Pixel 2	Pixel 1	Pixel 0
RGB 8A: 233	1	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0
RGB 8R: 332	1	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0
		Pixel 1		Pixel 0	
RGB 15+α	1	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0
RGB-16	1	r4 r3 r2 r1 r0 g5 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	r4 r3 r2 r1 r0 g5 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0
		1 Pixel/Word			
RGB 24+α	1	a7 - a0	r7 - r0	g7 - g0	b7 - b0
		Packed 4 Pixels/3 Words			
RGB-24-packed	1	B1	R0	G0	B0
	2	G2	B2	R1	G1
	3	R3	G3	B3	R2
		Packed 2 Pixels/Word			
YUV- 4:2:2	1	Y1	V0	Y0	U0

13.5 ALGORITHMS

13.5.1 Introduction

The ICP provides filtering, resizing (scaling) and YUV to RGB conversion of the source image. Filtering provides image enhancement. Scaling generates a new image that is larger or smaller than the current image. YUV to RGB conversion is used to generate an RGB version of the image for output to an RGB format frame buffer through the PCI interface or to SDRAM.

The filtering, scaling and YUV to RGB conversion algorithms are discussed separately. The ICP uses these algorithms in two ways.

1. It provides one pass horizontal scaling with horizontal 5-tap filtering of Y, U, or V.
2. It provides one pass vertical scaling with vertical 5-tap filtering of Y, U, or V.

13.5.2 Filtering

The ICP provides high quality, 5 tap polyphase filtering, both horizontal and vertical. Horizontal and vertical filtering are done in separate passes as one dimensional filters. Two dimensional filtering of the image requires two passes of the one dimensional filters.

Multi-tap FIR Filtering

In multi-tap FIR filtering of an image, the new filter output (pixel) value is a weighted sum of adjacent pixels. The weighting coefficients determine the type of filtering used. A 5-tap filter generates the new pixel value as a weighted sum of the current value and the two pixels on either side (2 left and 2 right for horizontal filtering, 2 above and 2 below for vertical).

You can use a multi-tap FIR filter to generate values for new pixels that are displaced from the original (“center”) pixel in the same way as linear interpolation. Assume the new pixel location is shifted slightly to the right of the center pixel of the input image. You can use a horizontal filter to estimate the new pixel value by weighting the right pixel filter coefficients more heavily than the left, proportional to the relative position offset of the new pixel. (In this sense, interpolation is a 2-tap filter.) This is shown in Figure 13-7. The ICP horizontal and vertical filter operations use this method to combine scaling with filtering.

Mirroring Pixels at the Start and End of a Line or Window

A line may start and/or end at the edge of the input image. In this case, you are missing the two start and/or end pixels needed for the first and last pixels of the line, respectively. The ICP uses pixel mirroring to solve this problem. In pixel mirroring, you use the two pixels you do have to substitute for the two missing pixels. For the first pixel, you use copies of the two pixels to the right as though they were the two pixels to the left. Specifically, p+2 substitutes for P-2, and P+1 substitutes for P-1. For the last pixel, you use copies of the two pixels to the left as though they were the two pixels to the right. Since the left and right pixels are now the same, this is called pixel mirroring.

There are five states of pixel mirroring: first output pixel, second output pixel, middle pixels, next to last output pixel and last output pixel. The first output pixel uses pixels numbered (2,1,0,1,2). The second pixel uses (1,0,1,2,3). The middle pixels use (P-2, P-1, P, P+1, P+2). The next to last pixel uses (N-3, N-2, N-1,N, N-1), where N is the number of the last input pixel. The last pixel uses (N-2, N-1, N, N-1, N-2).

In some cases of upscaling, one more input pixel may be needed at the end of the line than can be generated by the mirror logic. In this case, the ICP uses a copy the last pixel output as the best estimate of the required output pixel. The mirroring logic which detects the last pixels in the input line also detects this case, and it creates copies of the last pixel generated by preventing any further scaling action (data shifting and scaling counter incrementing) once the end of the input line has been reached.

13.5.3 Scaling

Scaling Overview

Resizing, or scaling the image means generating a new image that is larger or smaller than the original. The new image will have a larger or smaller number of pixels in the horizontal and/or vertical directions than the original image. A larger result image is scaling up (more new pixels); a smaller image is scaling down (fewer newer pixels). A simple case is a 2:1 increase or decrease in size. A 2:1 decrease could be done by throwing away every other pixel (although this simple method results in poor image quality). A 2:1 increase is more interesting. You can generate the new pixels in between the old by:

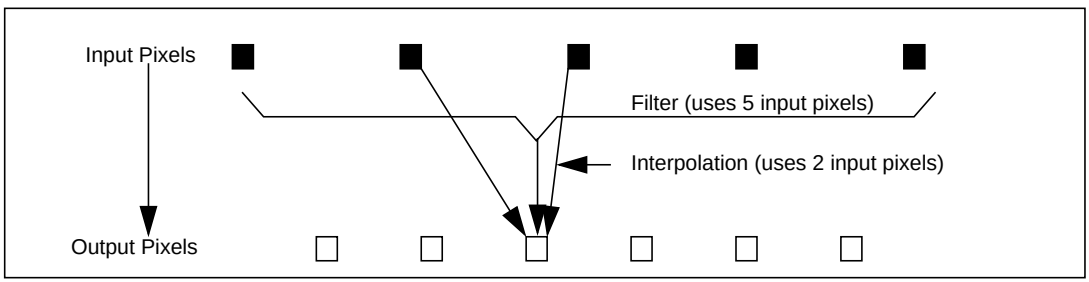


Figure 13-7. Pixel Generation by Interpolation and Filtering

1. Duplicating the original pixels
2. Linear interpolation, where the new in-between pixels are the weighted average of the adjacent input pixels
3. Multi-tap filtering, where the new in-between pixels are multi-pixel filtered version of the adjacent input pixels. This approach results in the best image.

The more general case is where the output image is not an integral multiple of sub-multiple of the input image, such as converting from 640 x 480 to 1024 x 768. In this case, the output pixels have differing positions relative to the input pixels as you move in the horizontal or vertical dimensions. In converting from 640 to 1024, the first output pixel on a line corresponds to the first input pixel. The second output pixel is at 640/1024 of the distance between the first and second input pixels. The third output pixel is at (2*640)/1024 of the distance = 1280/1024 = 1+ 256/1024 = 256/1024 of the distance between the second and third input pixels, etc. The output pixels shift with respect to the input pixel grid as you move along the line in the horizontal or vertical dimensions. This is shown in Figure 13-8.

New pixels are generated by interpolation or filtering of the original pixels. Interpolation is the weighted average of the input pixels adjacent to the output pixel. Filtering extends interpolation to include input pixels beyond the input pair adjacent to the output pixel. The number of pixels used to generate the output defines the filter type. Interpolation is a 2-tap filter. A four tap filter would use the two pixels to the left and the two pixels to the right of the output pixel. A 5 tap filter identifies the single pixel nearest the output as the center pixel, and uses this pixel plus two to the left and two to the right to generate the output.

If the ratio of the output pixel count per line (in H or V) to input pixel count per line is the ratio of small integers, you have a repeating pattern in these relative positions of input to output pixel locations. For 640 to 1024, the ratio is 8/5. The pattern repeats for every 8 output and every 5 input pixels. If the ratio is not a ratio of small integers, the pattern will take a long time to repeat. The worst case would be 640 to 641, for example. There would be no exact repetition for the whole line.

The interpolator or filter coefficients must be weighted according to the relative position of the new pixel relative to the old pixels. The weighting factor is between 0.0 and 1.0, corresponding to the relative position of the new pixel with respect to the old pixel grid. With a repeating pattern, you need fewer weighting factors, and therefore fewer coefficients in the linear interpolator or filter generating the new pixels, since you can reuse them each time the pattern repeats. A filter with a repeating pattern is called polyphase, indicating a repeating pattern in the phase (offset position) of the output pixels relative to the input pixels.

Generating the Output Pixels: Relating the Output Grid to the Input Grid

Scaling is a pixel transformation. You generate an array output pixels from an array of input pixels. The value of each pixel on the output pixel grid is calculated from the values of its adjacent pixels on the input grid. To find these adjacent pixels, you overlay the output grid on the input grid and align the starting pixels, X0Y0, of the two grids. To identify the adjacent input pixels for a given output pixel, you divide the output pixel X (pixel number along the output line) and Y (pixel line number within window) by their corresponding scaling factors:

$X_{in} = X_{out}$ / (Horizontal Scaling Factor)
where: Horizontal Scaling Factor = Output Length / Input Length

$Y_{in} = Y_{out}$ / (Vertical Scaling Factor)
where: Vertical Scaling Factor = Output Height / Input Height

Note that the resulting X_{in} and Y_{in} values will be real numbers, integers plus fractions. This is because the output pixels will usually fall between the input pixels. The fractional value indicates the fractional distance to the next pixel. To calculate the output pixel value, you use the value for the nearest pixel to the left and above and combine it with the value of the other adjacent pixel (s). For example, horizontal interpolation uses the starting pixel to the left interpolated with the next pixel to the right, with the fractional value used to determine the weighting for the interpolation.

ICP Scaling Output Resolution

In the ICP, scaling is forced to have a repeating pattern by limiting the resolution of the new pixel position to 1/32; the new position is forced to be at a location n/32 in H and V relative to the position of the original pixel grid. This results in a worst case error of approximately 1.5% in amplitude relative to calculations using exact output pixel positions. This is comparable to the errors caused by quantizing the amplitude of the pixels. The additional quantization noise can be avoided by choosing an appropriate scale factor which, when inverted, results in fractional values which are expressed in 32nd's, such as the 8/5 scaling factor in the 640 to 1024 example above. A

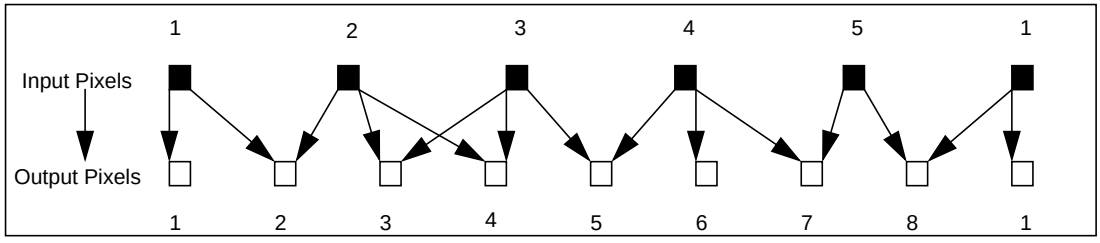


Figure 13-8. 640 to 1024 Upscaling Example

diagram of the input to output pixel relationship and the output fractional X and Y subpixel offset is shown in Figure 13-9.

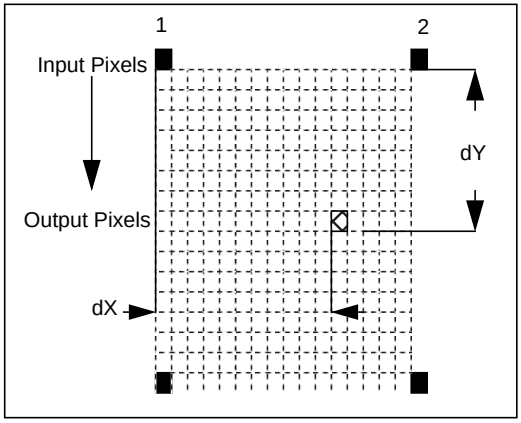


Figure 13-9. ICP 1/32 Output Resolution

Output Scaling Calculation Method

The output pixel distance in H and V in the ICP is calculated to high precision (16 bit fraction) even though the output resolution is fixed at 1/32 of the input grid. Each output pixel's location relative to the input pixel grid in memory is given by:

X location of output pixel = X0 of input line + Output pixel number / X Scale Factor

Y location of output pixel = Y0 of input window + Output line number / Y Scale Factor

The X and Y locations may not be integer values, depending on the scale factor. The resulting X and Y pixel locations can be separated into an integer and a fractional part. The integer part of the X and Y location selects the pixel and line number closest to the output pixel, respectively. The fractional part gives the fractional distance of the output pixel to the next X and Y input pixel values. These fractional parts are the dX and dY values shown in Figure 13-9.

The output pixel's value can be calculated by interpolation between these two pixels, or by 5-tap filtering using the 5 nearest pixels rather than the 2 nearest pixels. Interpolation or filtering uses the fractional position values, ΔX and ΔY, to select the appropriate filter coefficients. In the ICP, these values are limited to 5 bits for a resolution of 1/32, even though the actual position value has much higher resolution. The ICP uses fractional values that are centered around the center pixel with a range of -16/32 to +15/32.

To perform scaling, you must generate the X and Y locations of the output pixel relative to the input pixel grid, including both the integer part to locate the adjacent pixels and the fractional part to choose the filter coefficients which generate the output value from the adjacent pixels. This could be done by generating the output pixel X and Y numbers and dividing each by its associated scale factor. Since dividing is expensive in hardware and time, the ICP effectively multiplies the X and Y pixel numbers by the inverse of the X and Y scaling factors, respectively. The ICP does this by incrementing the X and Y input pixel counters by X and Y increment values that are the inverse of the X and Y scale factors, respectively. If you are at output pixel Xn, you have added the inverse of the scale factor to the X input location n times, equivalent to multiplying n by the inverse of the scale factor.

The ICP uses a 16-bit integer and a16-bit fractional value for the X and Y increment values. This allows a fractional value resolution of 1/64K. This high resolution of the calculated prevents an accumulation of error as you increment along the line. Since you will add the increment value 1024 times in a 1024 pixel line, any error in an individual calculation will be multiplied by 1024.

Only the most significant 5 bits of the fractional value are used by the filter coefficient RAMs. However, the X and Y Counters are incremented by the high resolution X and Y increment values. The result of this truncation is a worst case error of approximately 1.5% in amplitude relative to arbitrary pixel output positions.

The error caused by discrete (1/32) resolution can be reduced to exactly zero if the output image size is adjusted to have a repeating pattern that fits on these 1/32 boundaries. For zero error, this implies that the scaling factor must be of the form of B/A, where B (the output pixel count factor) is a submultiple of 32 [i.e., 1, 2, 4, 8, 16, 32], and A (the input pixel count factor) is an integer determined by the nearest acceptable scale factor for a given B. In the 640 to 1024 conversion case, the B/A ratio was 8/5, meeting this requirement.

The integer values, if accumulated, would be equal to the total number of input pixels when scaling is complete. The integer values for each pixel define the number of pixels to read from memory and shift in to generate the next output pixel. For example, a scaling factor of 1.0 will result in one pixel shifted in for each output pixel generated. Upscaling will have integer increment values of less than one. This means that the integer value will 0 for some pixels and 1 for others. For example, upscaling by 2.0 will result in integer values of 1 half the time and 0 for the other half, depending on the carry out from the fractional increment.

Pixel Shift Bypassing for Large Down Scaling

Down scaling will have integer increment values of greater than one. In this case, the integer value indicates the number of pixels to read to get filter pixels for the next output pixels. There are two ways to read and shift in the pixels in the down scaling case: shift all and shift bypass. In the shift all mode (the default mode) all five pixels are shifted for each input value read and shifted in. The shift all mode uses the five input pixels nearest the output pixel, independent of scaling factor. In the shift bypass case, only the last pixel is shifted in. For example, in a down scaling of 10, nine pixels are read, and the 10th pixel is shifted in to the filter. The shift bypass mode is used for large down scaling, i.e. down scaling factors of 2.0 or greater. The shift bypass mode is selected by setting the GETB bit in the parameter table. The shift by-

pass mode uses input pixels that are nearest the output pixel and those nearest each of the four output pixels adjacent to the output pixel. The shift bypass mode also forces the coefficient RAM inputs to zero, since you are no longer interpolating between adjacent input pixels.

Using Scaling to Convert From YUV 4:2:0 to YUV 4:2:2

YUV information in the 4:2:0 format has the UV pixels offset from the input grid in both X and Y. Also, the U and V pixels are at 1/2 of the horizontal and 1/2 of the vertical frequencies of the Y pixels. This means the UV pixels must be filtered and additionally scaled in both X and Y in order to line up with the output Y pixels even if no initial scaling is done. To generate 4:2:2 interspersed data, you vertically up scale U and V by a factor of 2 with a start offset of -1/4 pixel. Upscaling by 2 generates the additional lines required, and starting with a -1/4 pixel offset (relative to U, V space) moves the output up to the same line as the Y pixels. To generate 4:2:2 co-sited, you then filter horizontally with no scaling factor but with a start offset of -1/4 pixel, moving the output left 1/4 pixel.

13.5.4 YUV to RGB Conversion

In the ICP, YUV to RGB conversion is done by sequentially processing triplets of Y, U, and V pixel data to convert the pixels to an internal YUV 4:4:4 format and applying the YUV to RGB conversion algorithm on the YUV 4:4:4 pixels. The results of this conversion normally go to the PCI bus but can also go back to SDRAM.

YUV to RGB conversion has two steps. First you get the Y, U and a V pixel data to generate an RGB pixel at the output. Second, YUV to RGB conversion is done once the Y, U and V pixels are ready. YUV to RGB conversion uses the following algorithms:

R = Y + 1.375(V)= Y + (1 + 3/8)(V)
G = Y - 0.34375(U) - 0.703125(V)
= Y - (11/32)(U) - (45/64)(V)
B = Y + 1.734375(U)
= Y + (1 + 47/64)(U)

In CCIR601, the U and V values are offset by +128 by inverting the most significant bit of the 8-bit byte. This is the way the U and V values are stored in SDRAM. The above algorithms assume that the U and V values are converted back to normal signed two's complement values by inverting the MSB before being used.

13.5.5 Overlay and Alpha Blending

The ICP has the ability to add an overlay image to the main image when in the horizontal filter to RGB/YUV mode with PCI output. The overlay image is a user defined rectangle within the main image. When the overlay is active, each overlay pixel is combined with each main image pixel to generate the resulting pixel to be displayed. Each pixel combination is controlled by an alpha value which determines the proportions of overlay and main image that contribute to the output pixel. The relation is given by:

Pout = (alpha) * Poverlay + (1-alpha) * Pmain =
(alpha) * (Poverlay-Pmain) + Pmain
where: alpha ranges from 0 to 1

In the ICP, the alpha value range is limited by the hardware to five values: {0.0, 0.25, 0.50, 0.75, 1.0}.

An alpha value is supplied for each overlay pixel. In the RGB 24+alpha overlay data format: the 8-bit alpha value is contained within the overlay data. In all other overlay data formats (RGB 15+alpha, etc.), an alpha bit in the overlay data determines the alpha value. The alpha bit selects between two 8-bit values, alpha 1 and alpha 0, supplied by a pair of internal ICP registers. These registers are loaded from the parameter block when the ICP is started. When the alpha bit is one, alpha 1 value is used as the alpha value; when the alpha bit is zero, alpha 0 is used as the alpha value. The two alpha registers allow translucent images and backgrounds while being restricted to one bit per pixel for alpha selection.

Alpha blending has several uses.

1. Alpha can be used to disable portions of the overlay, called keying. When the alpha for a pixel is zero, there is no overlay. When the alpha is 1, the overlay is 100%, replacing the image. This allows the user to put an irregular shaped object in an image without showing the bounding rectangle of the overlay.
2. Alpha blending allows translucent ("smoky") backgrounds and/or translucent ("ghostly") overlay images
3. Using alpha at the edges of small images such as font characters increases their effective visual resolution.

Chroma Keying

The ICP also provides optional chroma keying. It is a restricted form of chroma keying, sometimes called color keying. When the overlay Y value is zero (an illegal value in the YUV 422+a format) or the RGB values are all zero (RGB15+a format), the alpha value is forced to zero and no overlay or blending occurs. This provides three levels of overlay: no overlay, alpha zero and alpha one. This combination can be used to generate an irregularly shaped menu (an oval shape, for example) which is translucent (with an alpha value of 50%, for example) and containing opaque (alpha = 100%) letters. In a game, this could be a message written on a foggy background in an oval window. The chroma keying provides the definition of the oval shape, the alpha zero value defines the translucent foggy background and the alpha one value defines the opaque characters on the foggy background.

Chroma keying in the ICP is intended for computer generated or modified overlays. Chroma keying turns off the overlay process for selected pixels by forcing an alpha value of zero for those pixels. Chroma keyed pixels use special codes to identify them. These codes must be computer generated in most cases. For example, the DSPCPU or other CPU would process an overlay image and convert the overlay pixels to be turned off into chroma keyed pixels by changing the data for those pixels to the chroma key code.

The ICP does not have full chroma keying. Full chroma keying has adjustable threshold values for the pixel components. Adjustable thresholds allow the user to automatically select an overlay sub-image from a larger overlay background, such as selecting an image of an actor against a bright blue background while inhibiting the blue background.

13.5.6 Dithering

Short output codes, such as RGB 8, have few bits for output value determination. RGB 8R has (2,3,3) bits for (R,G,B). The result is a coarse, patchy image if nothing is done to correct for the limited resolution. Dithering significantly improves the effective resolution of these images. RGB 8 images with dithering look nearly as good as RGB 16, for example.

Dithering works by adding a random dithering value to the pixel before it is truncated by the output formatter. The dither is added to the portion which will be truncated. The carry from this add will occasionally propagate into the most significant portion of the pixel before truncation. The carry from the add thus “dithers” the displayed value. This is shown in Figure 13-10. In the example shown, a random dither value is added to the original data before truncation. The dither value should have a range of from approximately 0 to 1 LSB of the truncated value. The dither value should be symmetrical about 1/2 the LSB of the quantizing error of the truncation. In the example shown, the dither signal has values of (1/8, 3/8, 5/8, 7/8). This set of values has a range of approximately 0 to 1 LSB, and it is symmetrical about 1/2 LSB.

In this example, the input signal has a value of 2.83. Without dithering, this value would be truncated to an output value of 2 in all cases. Averaging the undithered signal over four pixels still gives you a value of 2. By adding the dither signal, the output value is 2 or 3 depending on the value of the added dither signal. Averaging over four pixels, the average output value is 2.75, much closer to the input value that without the dither signal. The dither signal has significantly reduced the error when averaged over four pixels.

Two types of dithering are combined in the ICP: quad pixel and full image dithering. Quad pixel dithering, also known as ordered dithering, adds one of four dithering values to each pixel. The four dithering values correspond to four-pixel quads in the output image. The pixels in each quad have fixed positions in the input image, so the dither values are chosen on the bases of odd or even line number and odd or even pixel number in the line. The dither values of (0/4, 3/4, 2/4, 1/4) are added by line and pixel number: (even line & even pixel, even line & odd pixel, odd line & even pixel, odd line & odd pixel). This gives a four value ordered function for four adjacent pixels in the image. The (0,3,2,1) pattern is chosen specifically to prevent pairs of high or low pixel values from clustering. Spatial dithering provides a significant improvement in effective resolution.

Full image dithering adds a random number to every pixel of the image. The result is that the intensity and color accuracy increases as the size of the sample is enlarged. The random number has a long bit length to prevent repeating patterns in the image. The random number can be static or dynamic. In the static case, the random number generator starts with a fixed seed at the start of the image. The random number spatial pattern is fixed for the image even though the image data may change from frame to frame. In the dynamic case, the random number generator runs continuously, and the dithering pattern changes from frame to frame.

The ICP adds full image dithering to the quad pixel dithering to provide the final dithering signal for each pixel. The quad pixel dither provides the two most significant bits of the dither signal, and the full image dither provides the least significant 4-bits of the dither signal. The combined dither signal is 6 bits.

From 1 to 6 bits of dither signal are used, depending on the output format. If fewer than 6 bits are needed, only the most significant bits (MSBs) of the dither signal are used. For example in the RGB8R output format, the R output value is 3 bits in size. The output uses the 3 MSBs of the R input value and truncates the 5 LSBs. The dither unit adds 5 bits of dither signal (the 5 MSBs) to the 5

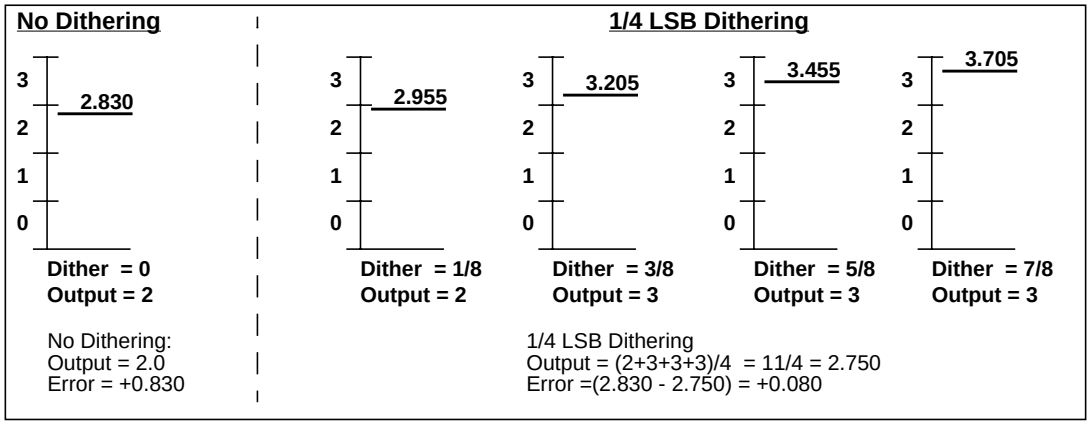


Figure 13-10. Dithering

LSBs of the R input value before truncation, and the RGB formatter truncates the result after adding.

13.5.7 Implementation Overview: Horizontal Scaling and Filtering

Figure 13-11 shows a data flow block diagram of the ICP horizontal scaling algorithm implementation. Blocks of pixels are provided by the input block buffer. Each block of pixels is transferred sequentially to the 5-tap filter. The filter does scaling and filtering of the data and puts the resulting pixels in the output buffer. Completed pixels in the output buffer are written back to SDRAM or to the PCI output. A bypass multiplexer allows bypassing the filter for SDRAM to SDRAM block moves.

Input pixel access is controlled by the Y Counter. The Y Counter selects the word and byte for the current pixel in the Y FIFO buffer. The Y Increment register, Y LSB Register and the Y MSB Counter control the increment of the Y Counter. If the Y MSB Counter contents are not zero, the Y Counter is incremented and the Y MSB register is decremented until the Y MSB Counter is zero.

The Y MSB Counter is loaded with the integer portion of the results of the Y Counter Increment operation. Y Counter Increment means adding the Y Increment fraction and integer values to the Y LSB register and Y MSB Counter, respectively. If there is no scaling (scaling factor = 1.0), the Y Increment integer value will be 1, and the Y Increment fractional value will be 0. Each Y Counter Increment operation will increment the Y Counter by one in this case.

The Y Counter sequentially reads out horizontally indexed pixels to the filter. The Y Counter is incremented once (1.0 for no scaling) for each pixel. For a line of pixels beginning with X_a and ending with X_b, the Y Counter reads pixels from the block buffer beginning with X_{a-2} and ending with X_{b+2}. The extra pixels are required by the 5-tap filter, which uses a total of 5 pixels to generate each output pixel, two pixels before and two pixels after each pixel. The horizontal filter uses the current output from the block buffer and four delayed versions of it to generate the filter output as the weighted sum of the center pixel plus the two on either side. (For the case where the scaling factor = 1.0, the LSBs are always zero.)

For up or down scaling, the Y Increment value is not 1.0, it is the inverse of the scaling factor (See "ICP Scaling Output Resolution," on page 13-7). For up scaling by a factor of 2.0, the effective Y increment value is 0.5, for example. This means you generate two output pixels for each input pixel. The Y Counter effectively increments as 0.0, 0.5, 1.0, 1.5, 2.0, etc. The LSBs of the counter (i.e., the fractional part less than 1) in the Y LSB register are used by to the filter to generate the intermediate values. An LSB value of 0.5 means that the output pixel is half way between X_n and X_{n+1}. The filter contains a set of 5 filter parameter RAMs, one for each coefficient. The 5

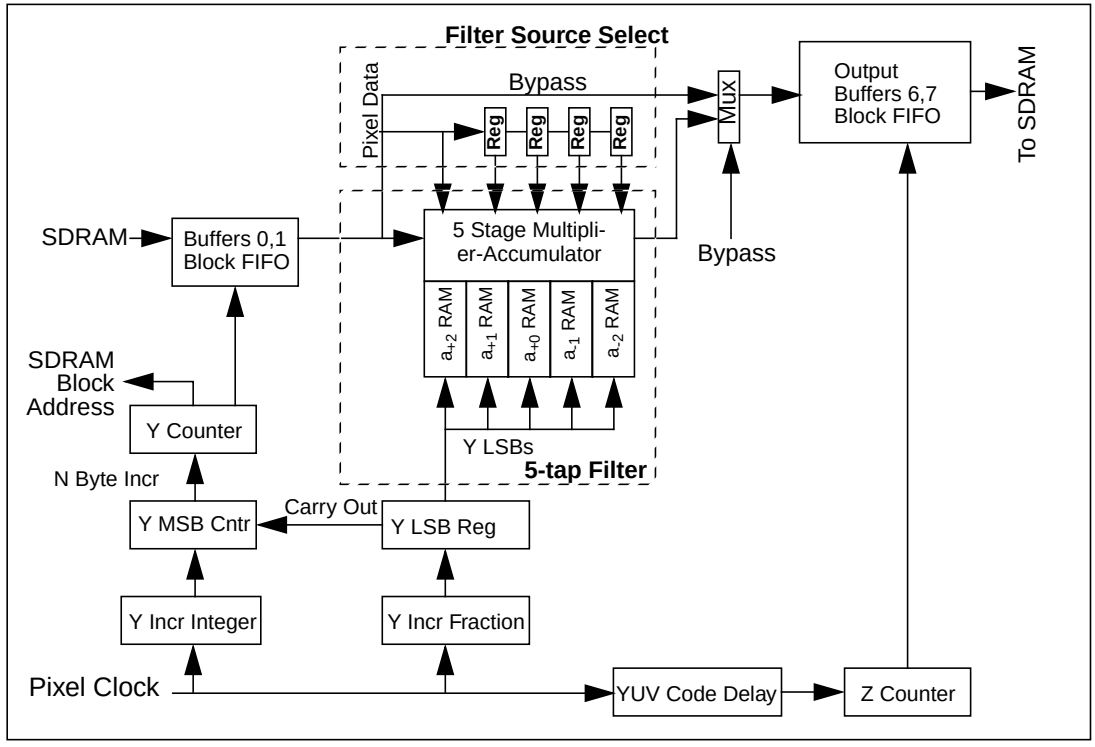


Figure 13-11. ICP Horizontal Scaling Data Flow Block Diagram

most significant LSBs from the counter select the filter coefficients which will generate the correct value for the output pixel at the relative offset from 0.0 indicated by the LSBs.

The Y Counter selects the next pixel from the input buffer. A new pixel is clocked into the filter registers only when the Y Counter contents change. The Y Counter contents change only when the Y MSB Counter is loaded with a value greater than zero. Note that for Y increment values less than 1.0 (up scaling), the change will be caused by carry increment from the Y LSBs, and a new pixel will not be clocked into the filter shift register on every Y clock.

For increment values of 2.0 or for values of 1.0 or greater with carry in (down scaling), multiple new pixels will be clocked into the filter shift register before the filter inputs are ready. The number of new bytes needed for the next pixel is the sum of the Y Increment Integer value and the carry out of the Y LSB adder. This result is loaded into the Y MSB Counter. The filter clock is stalled until the inputs are ready. The integer value of the increment -- including carry -- defines the number of new pixels to be clocked through the shift register before the filter inputs are ready for use.

In this discussion, the Y Counter LSBs form a 16-bit binary number. The upper 5 bits of this 16-bit number form a 5-bit binary number between 0 and 31 representing a fractional distance between Y pixels between 0/32 and 32/31. If the new pixel relative distance is 31/32, it is nearest the right pixel of the two pixels it is in between, and the right 2 pixels will be more heavily weighted than the left 3.

The horizontal filter shown in Figure 13-11 is pipelined to generate a pixel for every integer increment of the Y Counter. The filter input is always 5 clocks ahead of its output. The first stage generates the filter term $a_{n+2}X_{n+2}$ using the data from the input block and the a_{n+2} coefficient from the coefficient RAM driven by the Y LSBs. The second stage registers hold the data for X_{n+1} and its corresponding Y LSBs and generate $a_{n+1}X_{n+1}$. The last stage registers hold the data for X_{n-2} and the X_{n-2} LSBs and generate $a_{n-2}X_{n-2}$.

The LSB Register contents can change on every clock. In the 2:1 scaling example, the LSBs alternated between 0.0 and 0.5. The LSB Counter represents each output pixel's x offset value from the input pixel grid. The LSB Increment value is 16 bits long. The 5 upper bits go to the coefficient RAMs, and the 11 lower bits provide precision increment of the LSB Counter for precision in representing the scaling factor. The 11 lower bits of the LSB Increment value added to the 11 lower bits of the LSB Counter determine when to increment the 5 LSBs that drive the coefficient RAMs and when to clock a new Y pixel into the filter.

13.5.7.1 Loading the Extra Pixels in the Filter

For a 5 tap filter, you need 4 more pixels input to the filter than you generate at the filter output, two before the first pixel and two after the last pixel. In the worst case of a window that is exactly N blocks wide and starts at the first pixel of the first block, you will need to read two extra blocks - one at each end of the window - in order to get these 4 pixels! This is an unavoidable problem with a multi-tap filter. For an n-tap filter, you need n-1 extra pixels. There are two ways to avoid this efficiency hit of fetching extra blocks.

1. Move the window edges so they are not within 2 pixels of a 64 input pixel boundary.
2. Simulate the edge pixels, such as by mirroring the pair of pixels you have on the other side. This is the only solution to the problem of starting (or ending) at the edge of the image, where there are no pixels to the left (or right) of the image window.

The ICP uses automatic mirroring to supply these pixels. Mirroring is used in both horizontal and vertical filter modes.

13.5.7.2 Mirroring Pixels at the Ends of a Line

A line may start and/or end at the edge of the input image. In this case, you are missing the two start and/or end pixels needed for the first and last pixels of the line, respectively. The start mirror uses the two pixels to the right of the first pixel, and the end mirror uses the two pixels to the right of the last pixel. These pixels are supplied by controlling the Y counter.

A mirror multiplexer in the 5-tap filter provides mirroring of one or two pixels at the filter inputs. This mirror multiplexer is used for both horizontal and vertical filtering. In horizontal filtering, the first and last two pixels in the line are mirrored. The mirror multiplexer is set to the appropriate mirror code for the first and last two pixels in the line. The first two pixels are mirrored for the first two clock pulses, and the last two pixels are detected using the pixel counter for the line.

Mirroring is optional, depending on whether the start or end of the line is on a window boundary. The DSP CPU or microprogram must detect this and enable start and/or end mirroring as required.

13.5.7.3 Horizontal Filter SDRAM Timing

Figure 13-13 shows a timing diagram for block data flow between the SDRAM and the filter for a scaling factor of 1.0. The bus block reads and writes are one fourth of the filter processing time because the filter processes data at 100 mega pixels per second, and the SDRAM reads and writes blocks of pixels at 400 megapixels per second. The SDRAM logic reads the next block while the current block is being processed. This also provides the two pixels from the next block required to finish filtering the current block.

If the scaling factor is greater or less than 1.0. the SDRAM bus activity will be different. For scaling factors greater than 1.0, there will be fewer SDRAM reads for the same number of writes generated by the filter. For example, a scale factor of 2.0 means that you need to read only half as many blocks to generate the same number of output blocks. For a scale factor less than one, there will be more reads for the same number of

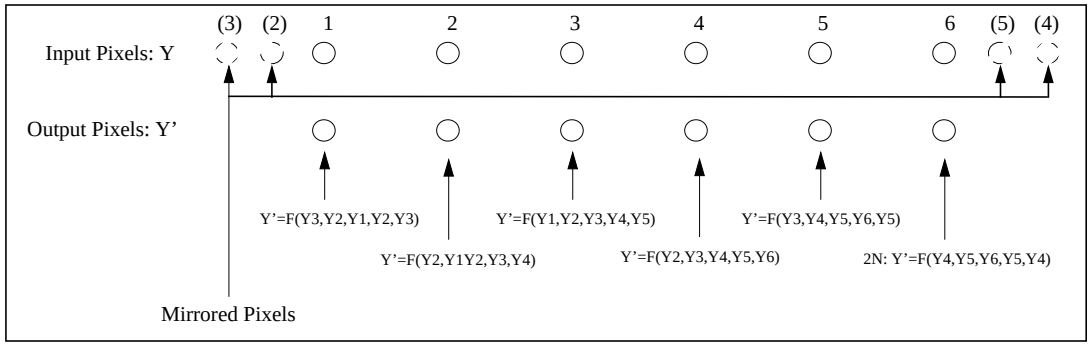


Figure 13-12. Horizontal Pixel Mirroring

writes. For a scale factor of 0.5, you need to read two blocks for every block of output. If the scale factor is less than 1/3, you will spend more time reading and writing SDRAM than filtering.

13.5.8 Implementation Overview: Vertical Scaling and Filtering

Figure 13-14 shows a data flow block diagram of the ICP vertical scaling algorithm implementation. Blocks of pixels are loaded sequentially into five input block buffers, one for each of the 5 terms of the 5-tap filter. Each block of pixels is transferred sequentially to the 5-tap filter. The filter does scaling and filtering of the data and puts the resulting pixels in the output buffer. Completed pixels in the output buffer are written back to SDRAM.

In the vertical scaling case, five separate blocks of pixels, one for each line, are required because the pixels are stored in horizontal sequence in the SDRAM. The Y Counter steps through the 64 horizontal pixels of the five input blocks and writes the resulting pixels into the output block. Four of the five blocks are used on the next pass, so that one block of pixels in generates one block of pixels out except for end conditions. The image is processed in 64 pixel columns. Since the image to be filtered will not generally start or end on a block boundary, the number of horizontal pixels for the first and last columns will be less than 64 in these cases. Also, the data in the columns must be aligned vertically. This results in the requirement that the line to line address offset value must be a multiple of 64 bytes. Note that only the address offset value is modulo 64; the image to be filtered can start and stop anywhere. Block alignment is not required.

Vertical scaling and filtering processes five 64 pixel input line segments to generate one 64 pixel output segment. When input lines Y_{n-2} to Y_{n+2} have been processed to generate one 64 pixel output segment for output line Y_n, five new input segments are needed for the next output line segment in the 64 pixel column, Y_{n+1}. If the vertical scale factor is 1.0 (no scaling), line segments Y_{n-1} to Y_{n+2} are reused, a new block for Y_{n+3} is loaded and the block for line Y_{n-2} is discarded.

To load Y_{n+3}, the MCU adds the Y offset value to the block address (upper 26 bits) of the Y Counter, and the Y Counter selects the next Y block to be read from SDRAM. The Y Counter points to the line block address for last Y block loaded, and the Y offset value is the address difference between the start of one line and the start of the next, X0Y0 to X0Y1. The line offset is always an integral number of SDRAM blocks. The line offset value must be added to the current line address to get the next line address.

Up and down scaling use the U Counter and U Increment value. The U Counter is used to detect how many lines must be read (0 to 5) to generate the next output line and to generate the vertical offset fraction for the 5-tap filter for output lines that fall between the input lines. The U Counter is set to its starting value (typically zero) at the start of the column, and the U Increment value is added to the U Counter for each output line segment generated in the column. For a scaling factor of 1.0, the U Increment value is 1.0, and each line processed will generate a request for one block. If the scaling factor is 1/2, the increment value will be two, corresponding to moving down two lines. In this case, twice the line offset is added to the Y Counter value.

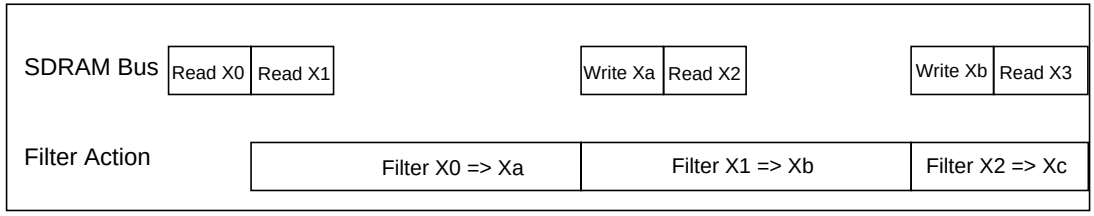


Figure 13-13. SDRAM and Horizontal Filter Block Timing

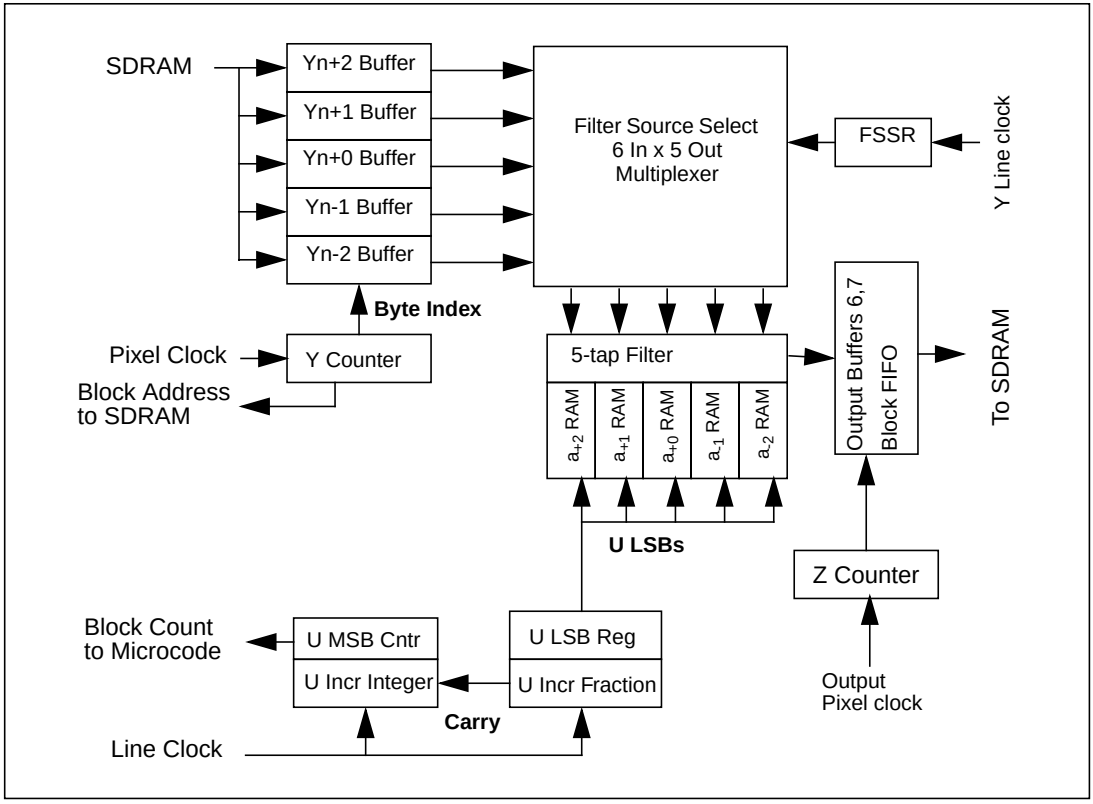


Figure 13-14. ICP Vertical Scaling Data Flow Block Diagram

For up scaling by a factor of 2.0, the Y increment value is 0.5. This means you generate two output lines for each input line. The U Counter increments as 0.0, 0.5, 1.0, 1.5, 2.0, etc. The LSBs of the U Counter (i.e., the fractional part less than 1) are passed along to the filter to generate the intermediate values. An LSB value of 0.5 means that the output line is half way between Y_n and Y_{n+1}. The filter contains a set of 5 filter parameter RAMs, one for each coefficient. The 5 most significant LSBs from the counter select the filter coefficients which will generate the correct value for the output pixel at the relative offset from 0.0 indicated by the LSBs.

For down scaling, the increment factor will be greater than one. If the increment factor is 2.0, two new blocks will have to be loaded before starting the next vertical filter pass. If the increment factor is 5 or greater, all five blocks will have to be loaded. The number of blocks to be loaded for the next line is equal to the integer increment value plus carry out from the LSB portion of the U Counter increment.

Note that the LSB adder carry out is available before the U Counter has been updated. This allows you to use the current U Counter value LSB bits for the filter coefficients while using the carry out for the next value to predict how many blocks to fetch. The integer value from the U increment adder is the number of blocks to be loaded. These blocks must be sequentially loaded (and not skipped) so that the filter has the necessary 5 adjacent lines to perform the filtering. The contents of the integer portion of the U Counter (updated after the add) are not used.

You can only load one new block while the current line is being processed. If two or more blocks are needed to process the next line, you load one in overlap, wait until the current line is done and then load the rest of the blocks. The microprogram only has to make two decisions for the next line: is the increment value zero or greater than zero, and if greater than zero, is it greater than five. If it is zero, do nothing: you will reuse all five blocks. If it is 1-4, load the next block. If it is five or more, calculate the address of the first block -- by adding N times the address offset to the Y counter -- and fetch it.

When a new block is loaded and it is time to process the next line, the block which was Y_{n+2} becomes Y_{n+1}. The Y blocks, in effect, shift up one line as you scan down the image. This shifting action is implemented by shifting the block select codes in the Filter Source Select Register (FSSR). The FSSR contains six 3-bit register fields. These 3-bit fields are rotated by a shift command to the FSSR. The output of five of the FSSR fields go to the input multiplexer, which selects the next block combination

and sends it to the filter. The output of the sixth field is the free block to be filled for the next line while the current line is being processed. The select code is also the block code (0 to 5), so the free block is identified by its block code in the FSSR. The FSSR codes for the six cases of vertical filtering are shown in Table 13-4

Table 13-4. FSSR Codes for Vertical Filtering.

Case	Pn-2	Pn-1	Pn+0	Pn+1	Pn+2	IO Block
1	5	4	3	2	1	0
2	0	5	4	3	2	1
3	1	0	5	4	3	2
4	2	1	0	5	4	3
5	3	2	1	0	5	4
6	4	3	2	1	0	5

13.5.8.1 Mirroring Lines at the Ends of an Image

A widow may start and/or end at the edge of the input image. In this case, you are missing the two start and/or end lines needed for the first and last lines of the window, respectively. These pixels are supplied by the mirror multiplexer at the 5-tap filter which mirrors the input lines.The mirror multiplexer is controlled by the mirror counter and mirror end register in the same manner as in horizontal filtering. The mirror register in vertical filtering is incremented by the output line counter. Mirroring is performed on the first two and last two lines of the column. Mirroring is optional, depending on whether the start or end of the line is on a window boundary. The DSP CPU or microprogram must detect this and enable start and/or end mirroring as required.

13.5.8.2 Vertical Filter SDRAM Block Timing

Figure 13-15 shows a timing diagram for block data flow between the SDRAM and the filter for a scaling factor of 1.0. The bus block reads and writes are one fourth of the filter processing time because the filter processes data at 100 mega pixels per second, and the SDRAM reads and writes blocks of pixels at 400 megapixels per second (peak). The vertical filter starts by reading in the five blocks necessary to generate the next output block. While the current block is being processed, the next block is read from SDRAM to prepare for the next output block.

13.5.9 Horizontal Scaling and Filtering for RGB Output

Figure 13-16 shows a data flow block diagram of the ICP horizontal scaling to RGB output algorithm implementation. The six input block buffers are arranged as three block FIFOs, one each for a Y, U and V pixel streams. These three streams are sequentially filtered, pixel by pixel by the 5-tap filter to generate a scaled and filtered output sequence of Y, U, V, Y, U, V, etc. This YUV stream is fed to the YUV to RGB converter where it is converted to one of several RGB output formats, blended with RGB overlay pixels supplied by the Overlay FIFO and masked by bit mask pixels from the bit mask block. The resulting scaled, filtered, converted, overlay blended and masked RGB stream is sent to the PCI interface -- typically to an RGB format frame buffer on the PCI bus -- or to the SDRAM.

The input pixel streams from the input FIFOs are transferred sequentially to the 5-tap filter. Each stream has its own set of four-stage delay registers used to perform horizontal filtering on the stream. A pair of 3-way multiplexers switch the five filter data inputs and the 5-bit filter coefficient select codes to the 5-tap filter. This set of multiplexers is driven by the YUV Sequence counter, a 2-bit counter that provides the YUV processing sequence.

Horizontal scaling and filtering for RGB output is performed in the same way as for ordinary filtering. The difference is in the format of the output data (RGB), the sequencing of the filtering to create the combined RGB output and the buffering of the YUV output data. In horizontal scaling and filtering from SDRAM to SDRAM, each Y, U and V component is filtered separately as a complete image. In RGB output horizontal scaling and filtering, the image is processed as three interwoven streams of all three YUV components.

In the RGB output mode, the ICP normally generates RGB data and writes it into a frame buffer memory on the RGB bus or to the SDRAM. The frame buffer memory format is RGB with one R, one G and one B value per pixel. This could be called RGB 4:4:4. To generate this image, the ICP generates a YUV 4:4:4 image and converts it to RGB. This process is done one RGB output pixel at a time. The ICP generates a U pixel and saves it in a register, generates a V pixel and saves it in a register, then generates a Y pixel for output. The YUV to RGB converter combines each Y pixel as it is generated with the previously stored U and V pixels to generate the RGB output data. This process is repeated until the whole image has been converted and sent to the PCI bus or SDRAM.

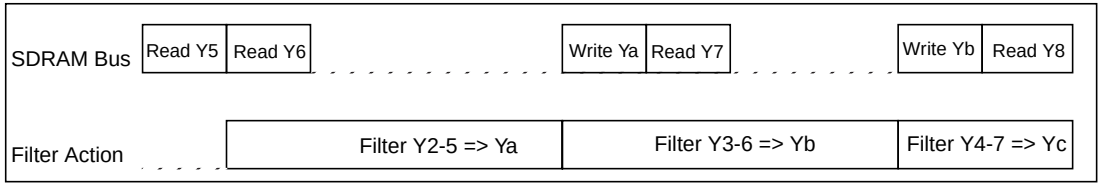


Figure 13-15. SDRAM and Vertical Filter Block Timing

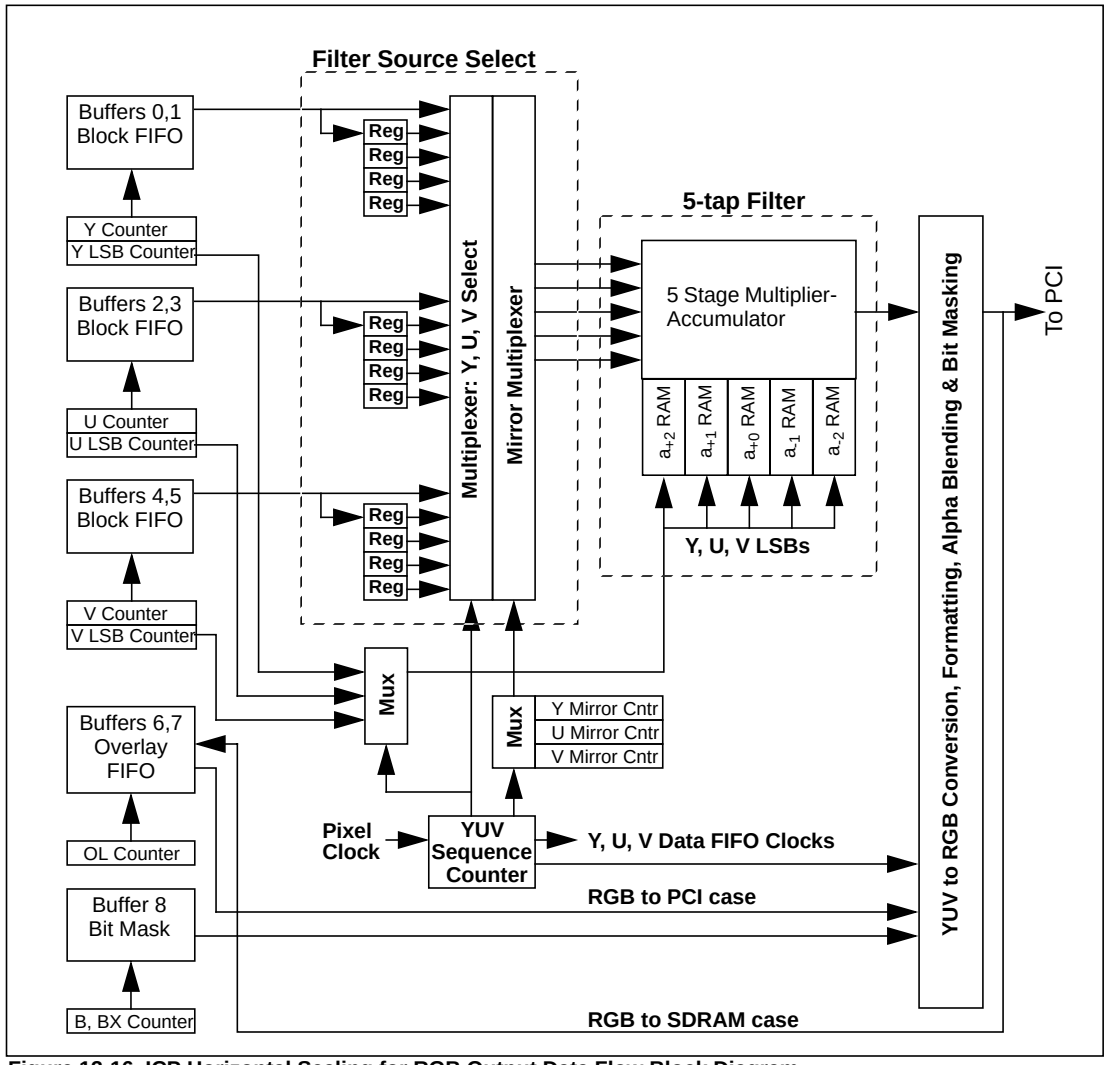


Figure 13-16. ICP Horizontal Scaling for RGB Output Data Flow Block Diagram

13.5.9.1 YUV Sequence Counter in YUV 422 Output Mode

For RGB output formats, the YUV data must be scaled to YUV 4:4:4 format before conversion to RGB. The YUV data in SRAM is typically stored in YUV 4:2:2. This means that the U and V data must be upscaled by 2 relative to the Y data to generate the internal YUV 444 format required for RGB conversion.

For the YUV 4:2:2 output code, the U and V data does not need to be up scaled to 4:4:4. You would be scaling up to YUV 444 only to decimate back to YUV 422. In the YUV 422 output case, you want to use the U and V pixels twice. This is done by having a half-speed mode for the YUV Sequence Counter. In this mode, the sequence is U0, V0, Y0, Y1, U2, V2, Y2, Y3, etc. The U and V are not up scaled by 2 relative to the Y component for YUV 4:4:4 output, although they could be up scaled as part of general up scaling of the image.

The YUV 422 Output mode also provides higher processing bandwidth relative to YUV 4:4:4 up scaling. You are processing half as many U and V pixels. The output pixel rate is one pixel per 20 nanoseconds for the YUV 422 Output mode versus one pixel per 30 for conversion to YUV 4:4:4. This can be used to provide some processing performance improvement for very large images at the expense of some chroma quality.

13.5.9.2 PCI Output Block Timing

The ICP generates pixels to the PCI interface at a peak rate of 33 output megapixels per second in the RGB mode and 50 megapixels per second in the YUV mode using YUV sequencing. For one word per pixel output codes, such as RGB-24, this is a peak rate of 33 mega words or 132 megabytes per second in the RGB sequencing mode. This is the same speed as the 132 megabytes per second peak rate of the PCI interface. (At 50 megapixels per second, the result would be 200 megabytes/second.) The BIU control for the PCI interface has a FIFO for buffering data from the ICP, but this buffer is only 16 words deep. Therefore, the ICP will occasionally have to wait for the PCI to accept more data. The PCI stalls the ICP by deactivating the biu_rdy line. In the PCI output mode, this stalls the ICP clock.

13.6 OPERATION AND PROGRAMMING

The ICP uses a combination of hardware and a Microprogram Control Unit (MCU) to implement its scaling, filtering and conversion functions. The microprogram is a factory supplied state machine that resides in SDRAM. It is read each time the ICP executes an operation. Using an SDRAM resident microprogram controlled state machine minimizes hardware and provides flexibility in handling special conditions without additional hardware.

Important Note: You must set the ICP DMA Enable bit (IE) in the BIO_CTL register of the PCI interface for RGB output to PCI. This bit must be set before initiating RBG to PCI operations, or the ICP will stall waiting for the PCI to become ready. Refer to Section 10.6.4, “BIU_CTL Register.”

13.6.1 ICP Register Model

The ICP is controlled by the DSP CPU through five MMIO registers: the MicroProgram Counter (MPC), the Micro Instruction Register (MIR), the Data Pointer (DP), the Data Register (DR) and the ICP Status register (SR), as shown in Figure 13-17. The MPC, DP and SR are used in normal operations, and the MIR and DR are used in test and debug.

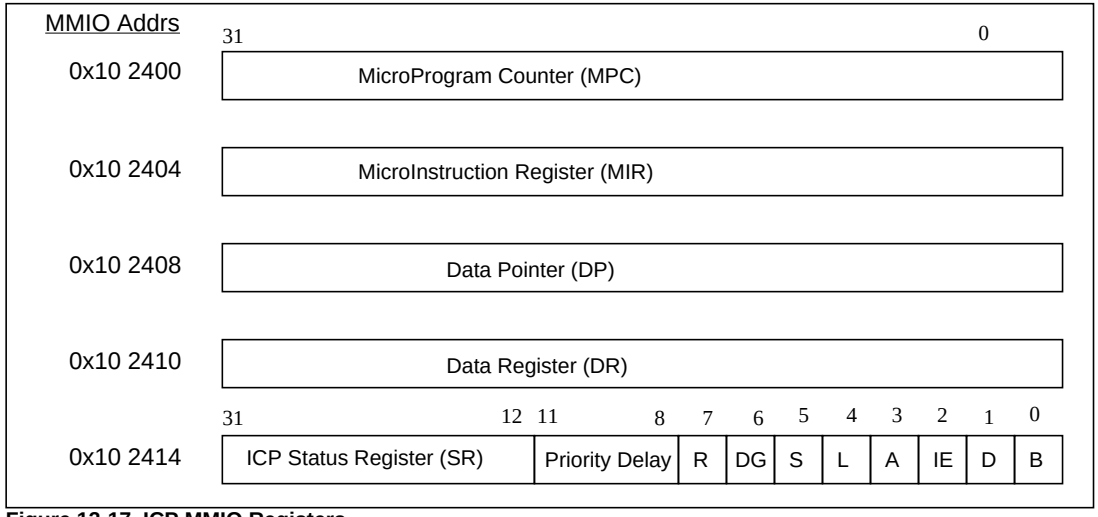


Figure 13-17. ICP MMIO Registers

The MPC is the MCU instruction counter. It points to the next microinstruction to be executed. The entry point in the microprogram defines which ICP operation is to be done.The DP points to the location in SDRAM of a table of parameters used by the ICP to process the image data, such as the image input and output start addresses, scaling factor, etc.

The SR has 12 active bits: Busy (B), Done (D), done Interrupt Enable (IE), ACK_DONE (A), Little Endian (L), Step (S), Diagnostic (DG), Reset (R) and Priority Delay (PD, 4 bits). The 20 most significant bits are reserved.

- Busy indicates the ICP is busy executing microcode.
- Done indicates that the previous requested function is complete, and that the ICP clock is stopped.
- Done causes an interrupt to the DSPCPU when Interrupt Enable is set.
- ACK_DONE clears Done and the corresponding interrupt.
- Little Endian sets the highway endian swap multiplexer to little endian mode for data on the SDRAM bus.
- Step causes the MCU to execute one microinstruction. Step is used for diagnostics to step the ICP through its microinstructions one clock step at a time. Writing a one to Step sets Busy, which is reset at the end of execution of the next microinstruction.
- DG allows SDRAM operations in step mode.
- R is a write-only bit that resets ICP internal registers.

- The PD field sets a timer for bus activity that defines the minimum bus bandwidth available to the ICP.

The ICP Status Register also contains 20 read only status bits. The upper 16 bits of the Status Register can contain a 16-bit code returned by the microprogram upon completion. Bits 15 through 12 are reserved for error flags.

Important Note: You must set the ICP DMA Enable bit (IE) in the BIO_CTL register of the PCI interface for RGB output to PCI. This bit must be set before initiating RBG to PCI operations, or the ICP will stall waiting for the PCI to become ready. Refer to Section 10.6.4, "BIU_CTL Register."

13.6.2 ICP Operation

The DSP CPU commands the ICP to perform an operation by loading the DP with a pointer to a parameter block, loading the MPC with a microprogram start address and setting Busy in the SR. For example to cause the ICP to scale and filter an image, you set up a block of SDRAM with the image and filter parameters, load the MPC with the starting address of the appropriate microprogram entry point in SDRAM, load the DP with the address of the parameter block, and set Busy in the SR by writing a one to it. When the filter operation is complete, the ICP will set Done and issue an interrupt. The DSPCPU clears the interrupt by writing a one to ACK_DONE.

When the DSPCPU sets busy, the MCU begins reading the microprogram from SDRAM. The microinstructions are read in from SDRAM as required by the ICP, and internal pre-fetching is used to eliminate delays. Setting Busy enables the MCU clock, the first block of microinstructions is automatically read in and the MCU begins instruction execution at the current address in the MPC. Clearing Busy stops the MCU clock. Busy can be cleared by hardware reset, by the MCU and by the DSPCPU. Hardware reset clears the Status register, including Busy and Done, and internal registers, such as the TCR. When the MCU completes a microprogram operation, the microprogram typically clears Busy and sets Done, causing an interrupt if IE is enabled.

The DSPCPU performs a software reset by clearing (writing a zero to) Busy and by writing a one to Reset. The DSPCPU can also set Done to force a hardware interrupt, if desired.

13.6.3 ICP Microprogram Set

The ICP comes with a factory generated microprogram set. This microprogram set implements the functions of the ICP. The microprogram set includes the following functions:

1. Loading the filter coefficient RAMs.
2. Horizontal scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
3. Vertical scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
4. Horizontal scaling, filtering and YUV to RGB conversion of an input image from SDRAM to an output image to PCI or SDRAM, with an alpha blended and chroma keyed RGB overlay and a bit mask. The input and output images can be of any size and position that fits in SDRAM and output to the PCI bus or SDRAM, The scaling factors are, in general, limited only by input and output image sizes.

The microprogram is supplied with the ICP as part of the device driver. The entry point in the microprogram defines which ICP operation is to be done. The entry points are given below in terms of word offsets from the beginning of the microprogram:

Offset	Function
0	Load coefficients
1	Horizontal scaling and filtering
2	Vertical scaling and filtering
3	Horizontal scaling, filtering, YUV to RGB conversion, bit masking (PCI) and overlay (PCI) with alpha blending and chroma keying

13.6.4 ICP Processing Time

The time for the ICP to process an image is a function of the processing function and the image. The time required for each image processing function has three components: 1) an overall setup time for the function, 2) a set up time for each line processed, and 3) the processing time for the pixels themselves. The equations shown below estimate the processing time for each of the ICP functions.

13.6.4.1 Horizontal Filter Processing Time

The time required for the horizontal filter to process one Y, U or V component of an image is given by the following equation:

HFTC = B(3 + 2H) + HWC

=Horizontal filter processing time (for one Y, U or V component)

where:

B = the time to load one block of 64 pixels from the TM1000 data highway = 0.16 usec at TM1000 clock = 100 MHz

H = the height of the input or output image in lines, whichever is larger

W = the width of the input or output image in pixels per line, whichever is larger

C = the filter processing time per pixel = TM1000 bus clock time = 10 ns at TM1000 clock = 100 MHz

clock

The B(3 + 2H) term represents the function and line overhead. The horizontal function requires three block times to initialize the function (function overhead). This is the time required to load the microcode and set up the function. The horizontal function requires two block times to initialize each line (line overhead). The line overhead of two block times represents the time to set up the line in microcode plus the time to load the first block of the line into the input FIFO. The ICP must wait for this block to be loaded before beginning processing the line. The HWC term represents the time to process each pixel of the image.

The time required to process all three Y, U and V components of a YUV 4:2:2 image is given by the following equation.

HFT = B(9 + 6H) + 2HWC

= Horizontal filter processing time (for YUV 4:2:2 image)

To process a YUV 4:2:2 image, you must process its three components. The B(9 + 6H) term represents the overhead for three calls to the horizontal filtering function. The 2HWC term represents the processing time for all three components, where the number of U and V pixels are each half the number of Y pixels.

13.6.4.2 Vertical Filter Processing Time

The time required for the vertical filter to process one Y, U or V component of an image is given by the following equation:

VFTC = B(4 + 3(W/64+2) + H(W/64+2)) + HWC

= Vertical filter processing time (for one Y, U or V component)

The B(4 + 3(W/64+2) + H(W/64+2)) term represents the function and line overhead. The vertical function requires four block times to initialize the function. The 3(W/64+2) term represents the time to load the three starting blocks at the beginning of processing each column. Three starting blocks are required to initialize the 5-tap filter. The remaining 2 blocks are mirrored. The H(W/64+2) term represents the time required to flush the block of filtered data for each 64 pixel line segment of each column. The “+2” part of the term represents the ends of the lines assuming that the ends are not block aligned. The HWC term represents the time to process each pixel of the image.

The time required to process all three Y, U and V components of a YUV 4:2:2 image is given by the following equation.

VFT = B(12 + 3(2W/64+6) + H(2W/64+6)) + 2HWC

= B(12 + (3+H)(2W/64+6)) + 2HWC

To process a YUV 4:2:2 image, you must process its three components. The B(12 + 3(2W/64+6) + H(2W/64+6)) term represents the overhead for three calls to the horizontal filtering function. The 2HWC term represents the processing time for all three components, where the number of U and V pixels are each half the number of Y pixels.

13.6.4.3 YUV to RGB Processing Time

The time required to process the three Y, U or V planar components of an image and convert it to RGB is given by the following equation:

RGBT = B(4 + 3H) + 3HWC

= HF with YUV to RGB conversion processing time

The B(4 + 3H) term represents the function and line overhead. The horizontal filter requires four block times to initialize the function, and three block times to initialize each line. The line overhead of three block times represents the time to load the first blocks of each of the Y, U and V components of the line into the input FIFOs. The ICP must wait for these blocks to be loaded before beginning processing the line. The input YUV 4:2:2 (or YUV 4:2:0) image is converted internally to YUV 4:4:4. The 3HWC term represents the time to process each YUV 4:4:4 pixel of the image, convert it to RGB and send it to PCI or the DRAM.

Adding overlay and bit masking adds line setup time to load the overlay and bit mask FIFOs. The time required to process the three Y, U or V components of an image and convert it to RGB with overlay and bit masking is given by the following equation.

RGBVT = B(4 + 5H) + 3HWC

= HF to RGB filter processing time with bit mask & overlay

If the output is YUV instead of RGB, the equation is modified. YUV output uses different internal sequencing. RGB output requires converting the YUV data to YUV 4:4:4 before conversion to RGB. YUV output is in YUV 4:2:2 format, so conversion to YUV 4:4:4 is not required. The YUV422 bit in the control word of the YUV to RGB parameter block indicates YUV 4:2:2 sequencing. Note that the YUV422 sequence bit can also be set for RGB output. This will decrease the processing time required at some expense in image quality, since the U and V values will not be upscaled to YUV 4:4:4 resolution.

The time required to process the three Y, U or V planar components of an image and convert it to YUV composite is given by the following equation:

YUVT = B(4 + 3H) + 2HWC

= HF to YUV 4:2:2 composite processing time

13.6.4.4 ICP Processing Time Examples

The estimated time to process various images using the above equations is given in Table 13-5 below.

13.6.4.5 ICP Bus Bandwidth and Processing Time

The processing time equations assume no bus contention, i.e. that the ICP has full and immediate access to the bus and that no other device is using the bus when the

ICP asks for it. The bandwidth used by the ICP for this case is given in Table 13-6 assuming zero bus latency. The data in the table uses a nominal overhead percentage associated with a 640 by 480 image.

The Pixel Bandwidth column shows the pixel processing rate of the filter in megabytes per second. The pixel processing time is equal to the total number of pixels processed divided by the pixel bandwidth. The Overhead Percent column shows the processing overhead time as a percentage of the pixel processing time as determined by the pixel bandwidth. The total processing time is equal to the pixel processing time plus the overhead time.

The DRAM Input Bandwidth column shows the demand made on the DRAM by the ICP during processing. Note that it is larger than the pixel bandwidth due to bus traffic caused by the overhead. The Overlay Bandwidth is another input bandwidth consumer, and is shown separately. The DRAM Output Bandwidth column shows the output bandwidth from the ICP to the DRAM during operation. Note that it is zero for PCI output. The DRAM Bandwidth column is the total DRAM bandwidth used by the ICP assuming zero bus latency.

Table 13-5. ICP Image Processing Time Examples

Element	Images					
W in pixels	360	640	720	800	800	1024
H in lines	240	480	480	480	600	768
Times for C = 0.010 usec, B = 0.160 usec						
Horizontal Filtering Time for YUV 422, usec	**1,960**	**6,606**	**7,374**	**8,142**	**10,177**	**16,467**
Overhead	12%	7%	6%	6%	6%	4%
Vertical Filtering Time for YUV 422, usec	**2,401**	**8,155**	**9,116**	**10,078**	**12,593**	**20,418**
Overhead	28%	25%	24%	24%	24%	23%
RGB Output Filtering Time, usec	**2,708**	**9,447**	**10,599**	**11,751**	**14,689**	**23,962**
Overhead	4%	2%	2%	2%	2%	2%
RGB Output with Overlay + Bit Mask, usec	**2,785**	**9,601**	**10,573**	**11,905**	**14,881**	**24,208**
Overhead	7%	4%	3%	3%	3%	3%
YUV Output Filtering Time, usec	**1,844**	**6,375**	**7,143**	**7,911**	**9,889**	**16,098**
Overhead	6%	4%	3%	3%	2%	2%
Vertical Filter + RGB Out Time, usec	5,108	17,602	19,715	21829	27,281	44,380
Vertical Filter + YUV Out Time, usec	4,244	14,530	16,259	17,989	22,481	36,516

Table 13-6. ICP Peak Bus Bandwidth Usage

Function	Pixel B/W, MB/s	Overhead Percent	DRAM In B/W, MB/s	Overlay B/W, MB/s	DRAM Out B/W, MB/s	DRAM B/W, MB/s
Horizontal Filter	100	7%	108	0	100	208
Vertical Filter	100	25%	133	0	100	233
RGB to DRAM	75	2%	77	0	100	177
YUV to DRAM	100	4%	104	0	100	204
RGV to PCI	75	2%	77	0	0	77
RGB + 10% Overlay	75	4%	78	20	0	98

The actual case is different. The ICP will not always have immediate access to the bus. If the ICP access is delayed, the ICP processing time will be longer. The actual increase in processing time is a complex function of the interaction of the ICP with the bus. However, the increase in processing time can be estimated by comparing the required bandwidth shown in Table 13-6 with the actual bandwidth available to the ICP. If the available bandwidth is lower, the processing time will be proportionally longer. The bandwidth correction factor is given by the following equation:

Bandwidth Correction Factor: (BCF)

BCF = Actual / Theoretical Processing Time

= ICP Bandwidth / Available Bandwidth

For example, the ICP bus bandwidth for horizontal filtering is 208 megabytes/second. If the available bus bandwidth is 100 megabytes/second, the actual processing time will be (208/100) = 2.08 times as long as the unlimited case. If the image processing time is 10 milliseconds for the unlimited case, it will be 20.8 milliseconds for the

case where the bus bandwidth is limited to 100 megabytes/second.

13.6.4.6 Priority Delay and ICP Minimum Bus Bandwidth

The Priority Delay field in the Status register sets the time the ICP will wait for SDRAM service before shifting from a low priority bus request to a high priority request. The ICP normally requests SDRAM bus service at the lowest priority level, since it is a background processing device. In some cases, service to the ICP could be continuously delayed by other background devices, such as the VLD processor and the DSPCPU. The PD field allows the ICP to change its priority level.

The PD field sets a timer on the currently active bus request. The timer is loaded with the PD value and started each time a bus request is submitted. The timer is incremented once each block time. If the timer times out before the request is serviced, the ICP changes its bus request from a low to a high priority request.

Table 13-7. ICP Bandwidth and Processing Time Bandwidth Correction Factor (BCF) vs. PD Code

PD Code	Timer Value	Minimum Bandwidth	H Filter BCF	V Filter BCF	RGB to DRAM BCF	RGB to PCI BCF	YUV to PCI BCF
0000	0	(400)	1.00	1.00	1.00	1.00	1.00
0001	1	200	1.04	1.17	1.00	1.00	1.02
0010	2	133	1.56	1.75	1.32	1.00	1.53
0011	3	100	2.08	2.33	1.77	1.00	2.04
0100	4	80	2.59	2.92	2.21	1.00	2.55
0101	5	67	3.11	3.50	2.65	1.15	3.06
0110	6	57	3.63	4.08	3.09	1.34	3.57
0111	7	50	4.15	4.67	3.53	1.53	4.08
1000	8	44	4.67	5.25	3.97	1.72	4.59
1001	9	40	5.19	5.83	4.41	1.91	5.10
1010	10	36	5.71	6.42	4.85	2.10	5.61
1011	11	33	6.23	7.00	5.30	2.30	6.13
1100	12	31	6.74	7.58	5.74	2.49	6.64
1101	13	29	7.26	8.17	6.18	2.68	7.15
1110	14	27	7.78	8.75	6.62	2.87	7.66
1111	15	25	8.30	9.33	7.06	3.06	8.17

The PD timer measures time in block times at 16 bus clock times per count. The ICP interprets the PD value in the 4-bit PD field as a number between 0 and 15, as determined by the code in Table 13-7. When the PD timer value is 0, the ICP waits 0 block times before going to high priority. When the PD value is 15, and the ICP waits 15 block times.

The PD value sets the minimum bus bandwidth available to the ICP. The minimum bandwidth is (400 mbyte/sec)/(PD timer value+1). If the PD timer value is 9, the bandwidth available is (400/(1+9)) = 40 Mbytes/second. The PD field allows the user to insure that the ICP has enough bandwidth to do its processing in the required frame time while limiting its bandwidth to allow other devices access to the bus.

13.6.5 ICP Parameter Tables

Each microprogram in the microprogram set has an associated parameter table used by the ICP to process the image data, such as the image input and output start addresses, scaling factor, etc. The DP points to the location in SDRAM of the first word of the parameter table. The parameter table address must be word aligned. The parameter table can be more than one SDRAM block (16 32-bit words) long.

13.6.6 Load Coefficients

This routine loads the filter coefficient RAMs with coefficient data in the parameter table. A total of 32 sets of five, 10-bit coefficients are loaded. Each set of five coefficients forms a 50-bit coefficient word. Two coefficients are stored in each 32-bit word in SDRAM. three 32-bit words are used for each set of five coefficients that form a coefficient word. The parameter table is 96 words (6 SDRAM blocks) long. Each coefficient is stored as the 10 most significant bits of each 16-bit halfword of the 32-bit word.

13.6.6.1 Parameter Table

The parameter table for the coefficient load function contains the coefficient data directly, as shown below. The parameter table is 96 words long.

Table 13-8. Load Coefficients Parameter Table

Parameter Word		Description
Upper 2 bytes	Lower 2 bytes	
a+2	a+1	RAM Coefficient word 0
a+0	a-1	
a-2	0	
a+2	a+1	RAM Coefficient word 1
a+0	a-1	
a-2	0	
a+2	a+1	RAM Coefficient word 31
a+0	a-1	
a-2	0	

13.6.7 Horizontal Filter - SDRAM to SDRAM

This routine performs horizontal scaling and filtering of one component (Y, U or V) of an N x M image from one location in SDRAM to another.

Table 13-9. Horizontal Filter Parameter Table

Parameter Word		Description
Upper 2 bytes	Lower 2 bytes	
Input Image Start Address		Start address of X0Y0 (byte address)
Y Counter Start Fraction	Input Image Line Offset	Starting value: may be 0.5, etc. for interspersed convert; Line offset from X0Y0 to X0Y1
Fraction increment	Integer increment	Increment value for Y = 1/scale factor
Input Image Height	Input Image Width	Height and width in input lines and pixels
Output Image Start Address		Start address of X0Y0 (byte address)
Control	Output Image Line Offset	Control bits; Line offset from X0Y0 to X0Y1
Output Image Height	Output Image Width	Height and width in output lines and pixels

The input and output addresses are the byte addresses of their respective tables. They need not be word or block aligned.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. The line offset allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. There are no restrictions on line offset values other than they must be 16-bit, two's complement integer values. (Note that this allows negative offsets. You can use this to flip an image vertically.)

The input and output image height and width values are the height in lines and width in pixels per line for their respective images. The height and width are 16-bit positive binary numbers between 0 and 64K.

13.6.7.1 Algorithms

The routine reads image data from SDRAM using the Y address counter, scales and filters the data in the horizontal direction and writes it back to the SDRAM using the Z address counter. The 5-tap filter scales and filters the data. The LSB Increment value supplied by the parameter table determines the scaling. The routine reads and writes a line at a time until the full image is transferred. The filter mirrors the ends of each line to provide the extra pixels needed by the filter at the ends of each line.

13.6.7.2 Parameter Table

The parameter table, shown in Table 13-9, supplies the input and output starting addresses and offsets, the image height in lines and width in pixels, and the increment value, which is derived from the scale factor.

The Integer increment and Fraction increment values are the scaling parameters. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If you are upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be 0 and the Fraction increment value will be 0.40. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be 2, and the Fraction increment value will be 0.500.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Width / Output Width
2. Multiply the Increment Value by 65536

3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value.
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and plus 0.49999+. The Start Fraction allows the input data to be offset by up to half a pixel, referred to the input pixel grid. It is zero for Y and for UV cosited data, and is set to minus 0.25 (C000h) for interspersed to cosited conversion of U and V data. The minus 0.25 value effectively shifts the U and V data toward the start of the line by 1/4 pixel, the amount required for conversion.

13.6.7.3 Control Word Format

The Control word provides bit fields which affect the horizontal filtering operation. The format of the Control word is as follows.

Bit	Name	Function
15	Bypass	Bypass filter. Picks nearest input pixel and passes it to output unfiltered. When Bypass is set & scale factor is 1.0, this results in an image block move
9	GETB	Large down scaling bit. Picks nearest input pixels and passes to filter, then passes them to filter. Equivalent to bypass + 5-tap filter of output pixels. LSB value = 0 for filtering.

The Bypass bit causes the data to bypass the 5-tap filter. The scaling operation selects the center pixel, and this pixel is passed to the filter output. No filtering or interpolation is provided. If the scaling factor is 1.0, the result is an image block move where the image is moved from one part of SDRAM to another without modification. If the scaling factor is other than 1.0, the effective algorithm is pixel picking, where the input pixel nearest the output pixel location is used as the output pixel.

The GETB bit is an optional bit for large (> 4) down scaling. When GETB is zero (normal operation), the 5-tap filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent input pixels on either side of this pixel to form the five filter inputs. When GETB is set, the filter receives the pixel nearest the output pixel as its center pixel plus the two pixels nearest the adjacent output pixels on either side of this pixel to form the five filter inputs. The effective algorithm is pixel picking plus 5-tap filtering of the result. GETB also forces the scaling LSB value to zero, since output pixels are being filtered and no interpolation is used. (See Section 13.5.2, "Filtering") This is shown in Figure 13-18.

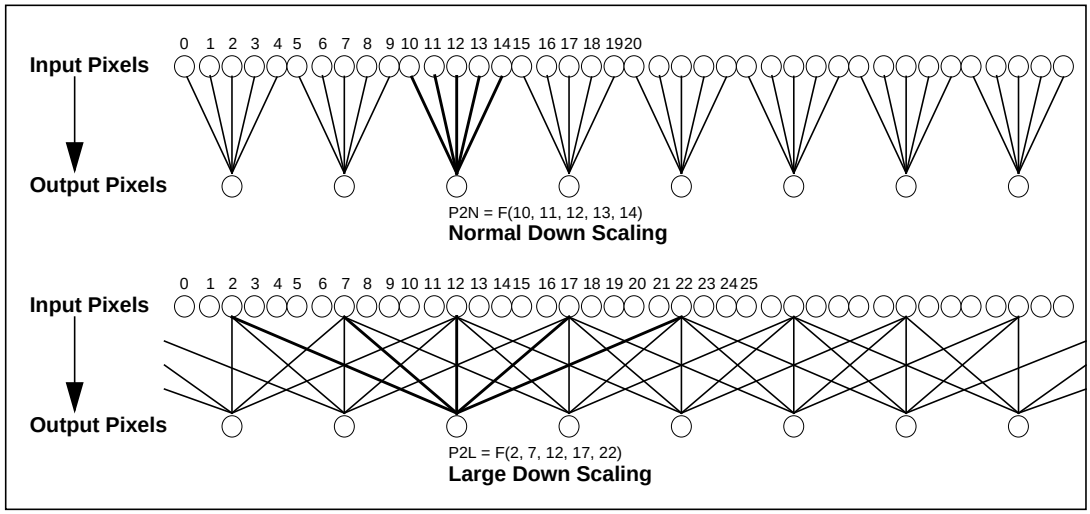


Figure 13-18. Normal vs. Large Down Scaling for Scale Factor = 5.0

13.6.8 Vertical Filter - SDRAM to SDRAM

This routine performs vertical scaling and filtering of one component (Y, U or V) of an N x M image from one location in SDRAM to another.

13.6.8.1 Algorithms

The routine reads image data from SDRAM using the Y address counter, scales and filters the data in the vertical direction and writes it back to the SDRAM using the Z address counter. The 5-tap filter scales and filters the data. The U LSB register is used as the scaling coefficient register. The U LSB Increment value supplied by the param-

eter table determines the scaling. Lines at the top and bottom of the image are mirrored to provide the extra line data needed by the 5-tap filter.

The routine reads and writes data in 64-byte (one SDRAM block) columns of pixels until the entire image is transferred. For each column, line segments of 64 pixels are processed until the entire column has been processed. Each 64 pixel line segment generated requires five vertically adjacent 64 pixel line segments as input to the 5-tap filter. The routine processes the image in pixel columns to eliminate redundant read of input pixel data: each new line segment typically requires reading only one new 64 byte line segment.

The routine processes data in 64 pixel blocks, corresponding to the input block buffer sizes. Five buffers are used in processing the current line segment, while the sixth buffer reads in the next line segment in overlap with current processing.

13.6.9 Parameter Table

The parameter table, as shown in Figure 13-19, supplies the input and output starting addresses and offsets, the image height in lines and width in pixels, and the scale factor.

Figure 13-19. Vertical Filter Parameter Table

Parameter Word		Description
Upper 2 bytes	**Lower 2 bytes**	
Input Image Start Address		Start address of X0Y0 (byte address)
U Counter Start Fraction	Input Image Line Offset	Starting value: may be 0.5, etc. for interspersed convert; Line offset from X0Y0 to X0Y1
Fraction increment	Integer increment	Increment value for U = 1/scale factor
Input Image Height	Input Image Width	Height and width in input lines and pixels
Output Image Start Address		Start address of X0Y0 (byte address)
Control	Output Image Line Offset	Control Word; Line offset from X0Y0 to X0Y1
Output Image Height	Output Image Width	Height and width in output lines and pixels

The input and output addresses are the byte addresses of their respective tables. They need not be word or block aligned.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. The line offset allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. Offset values are 16-bit, two's complement integer values.

Vertical filtering has a restriction on input line offset values: they must be positive, and they must be modulo 64 (i.e., a multiple of 64). Note that this only applies on the line-to-line spacing. Even with this restriction, input images may be any height and any width and may start at any byte address. Also, image subsets of arbitrary height and width can be used. As long as the original image has a modulo 64 line offset, all subsets of that image will also automatically have a modulo 64 line offset, the same as the original image. All images should have modulo 64 line offsets as good programming practice, even though this restriction only applies to vertical filtering. If an image does not have a modulo 64 line offset, it can be converted to modulo 64 by using horizontal filtering in the image block move mode with the output offset value being modulo 64.

The input and output image height and width values are the height in lines and width in pixels per line for their respective images. The height and width are 16-bit positive binary numbers between 0 and 64K.

The Integer increment and Fraction increment values are the scaling parameters. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If you are upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be 0 and the Fraction increment value will be 0.40. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be 2, and the Fraction increment value will be 0.500.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Height / Output Height
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value.

4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and plus 0.49999+. This value is placed in the The Start Fraction allows the input data to be offset by up to half a line, referred to the input pixel grid. It is set to zero for all conventional YUV input data.

13.6.9.1 Control Word Format

The Control word provides bit fields which affect the vertical filtering operation. The format of the Control word is as follows.

Bit	Name	Function
15	Bypass	Bypass filter. Picks nearest input line and passes it to output unfiltered. When Bypass is set & scale factor is 1.0, this results in an image block move

The Bypass bit causes the data to bypass the 5-tap filter. The scaling operation selects the center line, and this line is passed to the filter output. No filtering or interpolation is provided. If the scaling factor is 1.0, the result is an image block move where the image is moved from one part of SDRAM to another without modification. If the scaling factor is other than 1.0, the effective algorithm is line picking, where the input line nearest the output line location is used as the output line.

13.6.10 Horizontal Filter with RGB/YUV Conversion to PCI or SDRAM

This routine moves an N x M image in YUV 4:2:2, YUV 4:2:0 or YUV 4:1:1 format from SDRAM to the PCI bus or to SDRAM. The image is scaled and filtered in the horizontal direction during the move. Optional bit masking and/or RGB overlay may be used during the move when PCI output is specified.

13.6.10.1 Algorithms

The routine reads image data from SDRAM using the Y, U, and V address counters, scales and filters the data in the horizontal direction and writes it to the PCI interface or SDRAM. The 5-tap filter scales and filters the data. The LSB Increment value for each of the Y, U and V components supplied by the parameter table determines the scaling. Separate scaling factors allows YUV 422 interspersed to cosited transformation as the data is being filtered. The scaled and filtered data is converted to RGB or YUV format before being sent to the PCI interface or to SDRAM. In the PCI output case, overlay data with alpha blending and chroma keying can be added to the output image, and the output image can be gated by a bit mask before it is sent to the PCI interface.

The routine reads and writes a line at a time until the full image is transferred. The filter mirrors the ends of each line to provide the extra pixels needed by the filter at the ends of each line.

13.6.10.2 Parameter Table

The parameter table, as shown in Table 13-10, supplies the input and output starting addresses and offsets for Y, U, V, OL, B and Z, the image height in lines and width in pixels, and the scale factors for each component.

The input and output addresses are the byte addresses of their respective tables. They need not be word or block aligned.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. The line offset allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. There are no restrictions on line offset values other than they must be 16-bit, two's complement integer values. (Note that this allows negative offsets. You can use this to flip an image vertically.)

The input and output image height and width values are the height in lines and width in pixels per line for their respective images. The height and width are 16-bit positive binary numbers between 0 and 64K.

Table 13-10. Horizontal Filter to RGB Output Parameter Table

Parameter Word		Description
Upper 2 bytes	**Lower 2 bytes**	
Input Image Y Start Address		Y Start address of X0Y0 (byte address)
Y Counter Start Fraction	Input Image Y Line Offset	Starting value: may be 0.5, etc. for interspersed convert; Y Line offset from X0Y0 to X0Y1
Y Fraction increment	Y Integer increment	Increment value for U = 1/scale factor
Y Input Image Height	Y Input Image Width	Y& Height and width in pixels
Input Image U Start Address		U Start address of X0Y0 (byte address)
U Counter Start Fraction	Input Image U Line Offset	Starting value: may be 0.5, etc. for interspersed convert; U Line offset from X0Y0 to X0Y1

Table 13-10. Horizontal Filter to RGB Output Parameter Table

Parameter Word		Description
Upper 2 bytes	**Lower 2 bytes**	
U Fraction increment	U Integer increment	Increment value for Y = 1/scale factor
U Input Image Height	U Input Image Width	U Height and width in pixels
Input Image V Start Address		V Start address of X0Y0 (byte address)
V Counter Start Fraction	Input Image V Line Offset	Starting value: may be 0.5, etc. for interspersed convert; V Line offset from X0Y0 to X0Y1
V Fraction increment	V Integer increment	Increment value for V = 1/scale factor
V Input Image Height	V Input Image Width	V Height and width in pixels
Output Image Start Address		Start address of X0Y0 (byte address)
Control	Output Image Line Offset	Input & output formats & control bits; Line offset from X0Y0 to X0Y1
Output Image Height	Output Image Width	Height and width in output pixels
Bit Map Image Start Address		Start address of X0Y0 (byte address)
0	Bit Map Image Line Offset	Line offset from X0Y0 to X0Y1
RGB Overlay Start Address		Start address of X0Y0 (byte address)
Alpha 1 & Alpha 0	Overlay Line Offset	Alpha 1 & Alpha 0 blend code for RGB15+alpha, etc.; Line offset from X0Y0 to X0Y1
Overlay End pixel	Overlay Start Pixel	Start and end pixels along line
Overlay End Line	Overlay Start Line	Start and end lines in frame

The Integer increment and Fraction increment values are the scaling parameters. there is a separate scaling parameter for each of the Y, U and V input components. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If you are upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be 0 and the Fraction increment value will be 0.40. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be 2, and the Fraction increment value will be 0.500.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Width / Output Width
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value.
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values.

For YUV 422 or YUV 420 input data and RGB output data, the scaling factor for U and V must be twice the scaling factor for Y, unless YUV422 sequencing is used for speed. In YUV 422 or YUV 420 data, the horizontal components of U and V are half those of Y. The U and V must be upscaled by 2 to generate a YUV 444 format internally for YUV to RGB conversion. For YUV 411 input data, the U and V components must be upscaled by a factor of 4 to generate the required internal YUV 444 format.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and plus 0.49999+. The Start Fraction allows the input data to be offset by up to half a pixel, referred to the input pixel grid. It is zero for Y and for UV cosited data, and is set to minus 0.25 (C000) for interspersed to cosited conversion of U and V data. The minus 0.25 value effectively shifts the U and V data toward the start of the line by 1/4 pixel, the amount required for conversion.

The Alpha 1 and Alpha 0 values are 8-bit fields within the 16-bit Alpha field. These values are loaded into the Alpha 1 and Alpha 0 registers, respectively, for use by RGB 15+alpha and YUV 422+alpha overlay formats in alpha blending.

The Overlay start and end pixels and lines define the start and end pixels and lines within the output image for the overlay. The first pixel of the overlay image will be blended with the pixel at the Overlay Start Pixel and Overlay Start Line in the output image.

13.6.10.3 Control Word Format

The Control word provides bit fields which affect the horizontal filtering operation. The format of the Control word is as follows.

Bits	Name	Function
15	-	Always 0 (reserved)

14	422SEQ	422 Sequence bit. Used with YUV422 output
13	YUV420	YUV 420 input format
12	OEN	Overlay enable. Valid only for PCI output
11	PCI	PCI output enable. Otherwise, DRAM output
10	BEN	Bit mask enable. Valid only for PCI output
9	GETB	Large down scaling bit. Picks five input pixels nearest 5 output pixels and passes to filter. Equivalent to filter bypass + 5-tap filter of output pixels. LSB value = 0 for filtering.
8	OLLE	Overlay little endian enable
7-6	OFRM	Overlay format 0 = RGB 24+alpha 1 = RGB 15+alpha 2 = YUV 422+alpha
5	CHK	Chroma keying enable
4	LE	RGB output little endian enable
3-0	RGB	RGB Output Code 0= YUV 422+alpha 1 = YUV 422 2 = RGB 24 + alpha 3 = RGB 24 packed 4 = RGB 8A = RGB 233 5= RGB 8R = RGB 332 6 = RGB15+alpha 7 = RGB 16

The 422SEQ bit controls the internal sequencing of the YUV to RGB operation. It is set to one when YUV422 output is selected. When 422SEQ is zero, normal RGB output is assumed. In this mode, the input is YUV 422 or YUV 420, and the output is RGB. To generate the RGB output, the YUV 422 or YUV 420 input must be upscaled to YUV 444 before conversion to RGB. This means the scaling factor for U and V must be twice the scaling factor for Y. The internal sequencing of the filter in this case is UVY, UVY, UVY to generate RGB, RGB, RGB. For YUV 422 output formats, no upscaling of U and V is required. In this case, the 422SEQ bit is set to one, and the filter sequence is UVYY, UVYY, UVYY.

The 422SEQ bit can be set in RGB output mode to decrease the processing time for the image at the expense of color bandwidth and some corresponding decrease in picture quality. If the 422SEQ bit is set for RGB output, the filter will perform the UVYY sequence. In this case, the U and V components are not upscaled by 2, and the YUV to RGB converter updates its U and V components every other pixel. In the normal case (422SEQ=0), it takes 6 clocks to generate two RGB pixels. In the 422SEQ=1 case, it takes 4 clocks to generate two RGB pixels, reducing processing time by 33%.

The YUV420 bit indicates that the input data is in YUV 420 format. In YUV 420 format, the U and V components are half the width and half the height of the Y data. YUV 420 data is normally converted to YUV 422 data by a separate vertical upscaling by a factor of 2.0 for best quality. The YUV420 bit allows using YUV 420 data directly but with some quality degradation. When YUV420 is set, the ICP up scales the data vertically by line duplication. Each U and V input line is used twice. The separate vertical scaling step is eliminated at the expense of some quality since the lines are simply duplicated rather than being fully scaled and filtered.

The OEN bit enables overlay. You set it to one if an overlay is used, zero if not. Overlays are only valid for PCI output.

The PCI bit selects PCI as the output port for the ICP data. A one selects PCI output; a zero selects SDRAM output.

The BEN bit enables bit masking. You set it to one if bit masking is used, zero if not. Bit masking is only valid for PCI output.

The GETB bit is an optional bit for large (> 4) down scaling. When GETB is zero (normal operation), the 5-tap filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent input pixels on either side of this pixel to form the five filter inputs. When GETB is set, the filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent output pixels on either side of this pixel to form the five filter inputs. The effective algorithm is pixel picking plus 5-tap filtering of the result. GETB also forces the scaling LSB value to zero, since output pixels are being filtered and no interpolation is used.

The OFRM bit field selects the overlay data format, as shown in the Control word format list.

The CHK bit enables chroma keying. You set it to one if chroma keying is used, zero if not.

The OLLE bit sets the endian-ness of the overlay data input. You set it to one if the overlay data is little-endian, zero if big endian. The OLLE bit is normally set to the same value as the LE bit in the Status register.

The LE bit sets the endian-ness of the RGB/YUV output data. You set it to one if the output data is little-endian, zero if big endian. The LE bit is normally set to the same value as the LE bit in the Status register.

The RGB field defines the output data format, as shown in the Control word format list.

Important Note: You must set the ICP DMA Enable bit (IE) in the BIO_CTL register of the PCI interface for RGB output to PCI. This bit must be set before initiating RBG tp PCI operations, or the ICP will stall waiting for the PCI to become ready.

13.7 ICP PROGRAMMING EXAMPLES

The following examples show how to use the ICP to solve common scaling and filtering problems. The ICP is controlled by its parameter tables. These tables provide a great deal of flexibility by providing user control over the various parameters and starting conditions for each type of scaling, filtering and conversion. These examples show how to set-up the ICP parameter tables for each problem type.

This section contains the following examples:

13.7.1 Load Coefficients

This routine loads the filter coefficient RAMs with coefficient data in the parameter table. A total of 32 sets of five, 10-bit coefficients are loaded. Each set of five coefficients forms a 50-bit coefficient word. Two coefficients are stored in each 32-bit word in SDRAM. three 32-bit words are used for each set of five coefficients that form a coefficient word. The parameter table is 96 words (6 SDRAM blocks) long. Each coefficient is stored as the 10 most significant bits of each 16-bit halfword of the 32-bit word. The values shown in this example are part of the default coefficient table for the ICP.

Parameter	Up 16	Low 16	Param Word	RAM Word
a+2, a+1	0000h	01FFh	0	0
a+0, a-1	0000h	0000h	1	0
a-2, 0	0000h	0000h	2	0
a+2, a+1	FFF6h	601Fh	3	1
a+0, a-1	0000h	0000h	4	1
a-2, 0	0001h	0000h	5	1
a+2, a+1	FFF6h	601Fh	6	2
a+0, a-1	0000h	0000h	7	2
a-2, 0	0001h	0000h	8	2
.....				
.....				
a+2, a+1	000Ch	01FEh	93	31
a+0, a-1	0000h	FFFFh	94	31
a-2, 0	0174h	0000h	95	31

13.7.2 Horizontal Filtering Without Scaling (Scale Factor = 1)

Horizontal filtering without scaling (i.e., scale factor = 1.000) is one of the simpler ICP operations. The parameters for this operation are given below for a 640 x 480 image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

The horizontal filtering operation transfers an image through the ICP filter line by line. Each line is read into the ICP fro DRAM, filtered and written back to DRAM. This memory to memory data transfer is shown in Figure 13-20

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	1	Scale factor = 1 (no scaling)
Fraction Increment	0	Scale factor = 1 (no scaling)

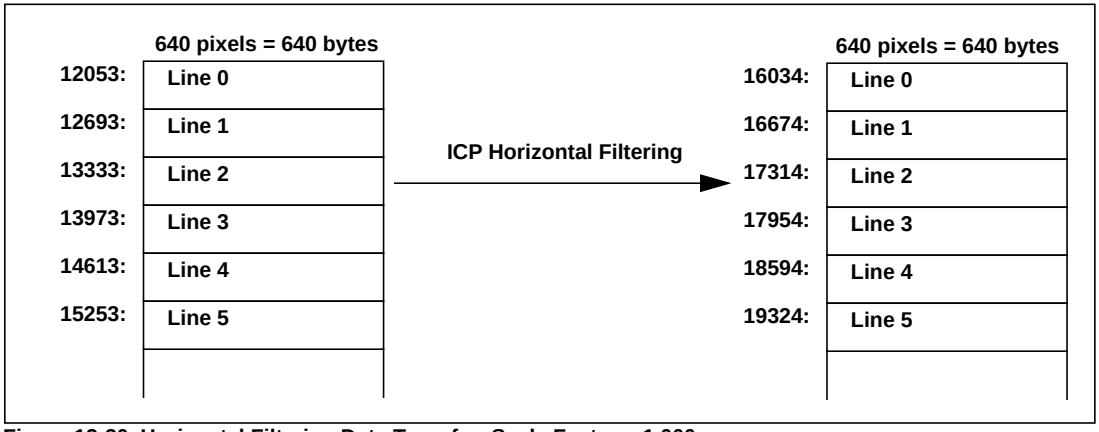


Figure 13-20. Horizontal Filtering Data Transfer: Scale Factor = 1.000

13.7.3 Horizontal Filtering of Sub-image (Windowing)

The ICP will often work on a sub-image, or window, of a larger image. The following example shows the parameter block for horizontal filtering of a sub image without scaling (scale factor = 1.000).

In this case, the sub-image is 100 pixels wide and 20 lines high, and it is contained in an input image that is 640 pixels wide by 480 lines high. The sub-image starts on the third line of the input image and on the 200th pixel of the input image. The input image offset is 640 bytes, the same as the larger, source image. The starting address is offset by 200 bytes corresponding to the 200th pixel starting point.

Note that the input image height in lines is not used, and that the input image width in pixels is used only to control end of line mirroring. The output height and width and the scaling factor (increment value) control the input pixel usage. The ICP uses input pixels as required to generate the output pixels.

The input width controls end of line mirroring. The beginning of the line is always mirrored. You can inhibit of line mirroring in this case because the source image extends beyond the end of the sub-image. To inhibit end of line mirroring, set the input width to at least 2 larger than the actual width: i.e., set the input width to 102 in this case.

Parameter	Value	Comments
Input Image Start Address	13533	Starting byte address of first pixel of input image = 13333 + 200
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	20	Input image height in lines (unused)
Input Image Width	100	Input image width in pixels (used only for end mirroring)
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	100	Offset in bytes from first pixel of one line to the next
Output Image Height	20	Output image height in lines
Output Image Width	100	Output image width in pixels
Integer Increment	1	Scale factor = 1 (no scaling)
Fraction Increment	0	Scale factor = 1 (no scaling)

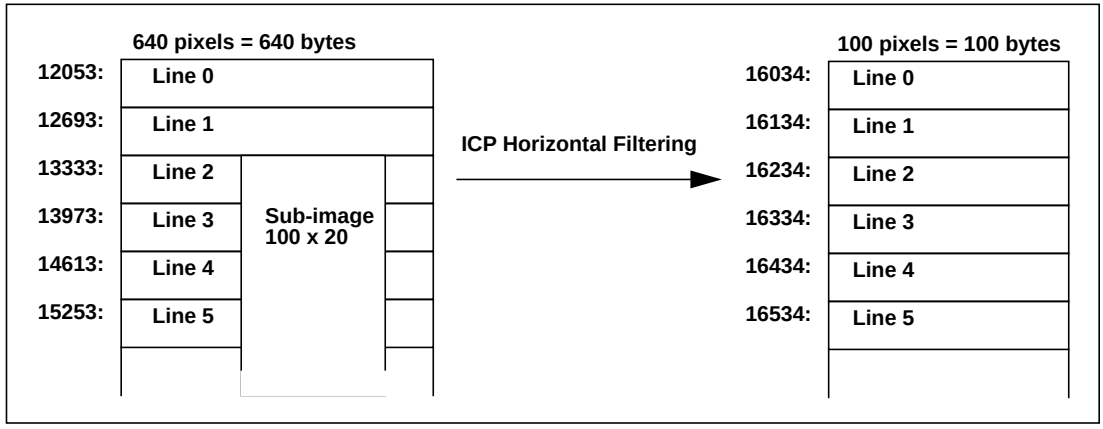


Figure 13-21. Horizontal Filtering Data Transfer of a Small Window in an Image

13.7.4 Image Move Using Horizontal Scaling with Bypass

To move an image without scaling or filtering, use the bypass flag. The bypass flag causes the data to bypass the 5-tap filter. If the scale factor is 1.000, no scaling or filtering will be done. The example below shows the parameter block contents to move an image without scaling or filtering.

The bypass mode can be used with scale factors other than 1.000. No filtering or interpolation is done because the bypass flag causes the data to bypass the 5-tap filter. If the image is scaled up, pixels will be duplicated to make the larger output image. If the image is down scaled, pixels will be skipped to make the smaller output image. This mode of operation would not normally be used except for experimental work, such as comparing filtered with non-filtered data.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	8000h	Bypass on, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	1	Scale factor = 1 (no scaling)
Fraction Increment	0	Scale factor = 1 (no scaling)

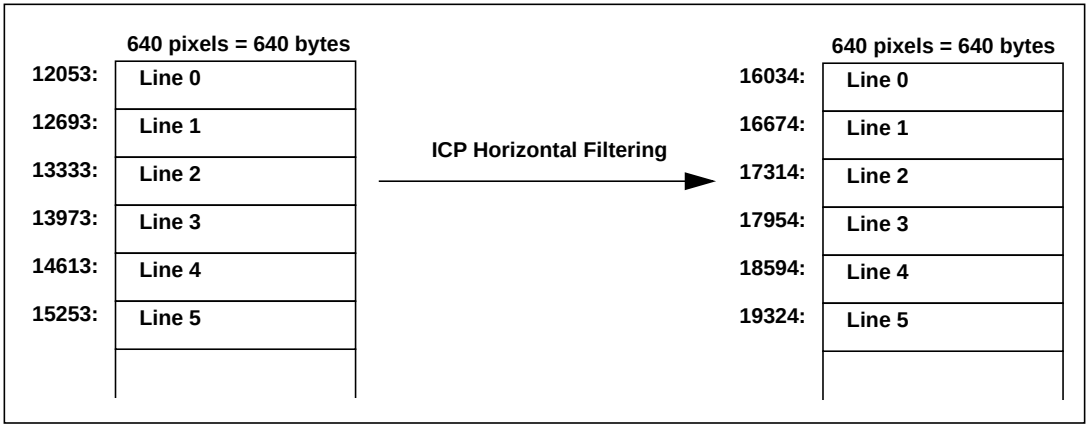


Figure 13-22. Image Move: Horizontal Filtering Data Transfer with Bypass

13.7.5 Horizontal Up Scaling

The parameters for horizontal upscaling by a factor of 2.5 (2.5 times as many output pixels as input pixels) are given below for a 640 x 480 input image upscaled to a 1600 x 480 output image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	1600	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	1600	Output image width in pixels
Integer Increment	0	Increment = 1/ Scale factor for 2.5 up scaling
Fraction Increment	6666h = 0.400	Increment value = 0.400

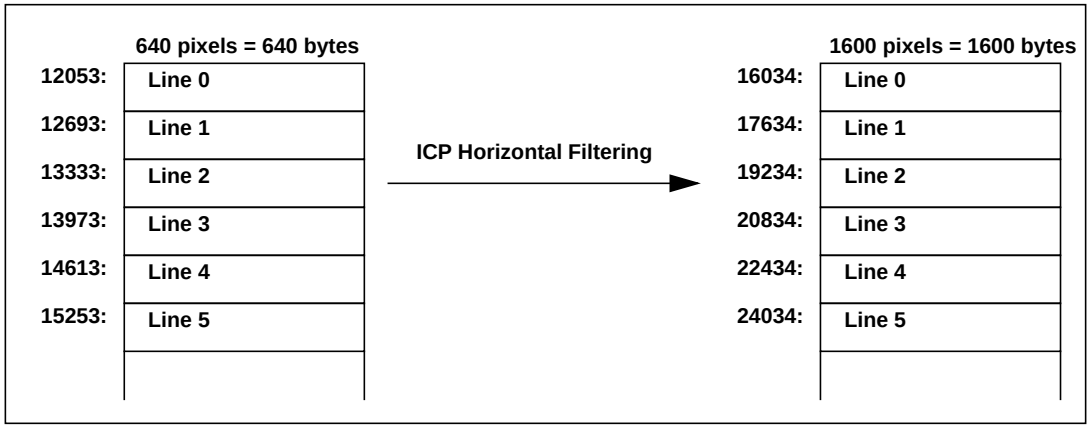


Figure 13-23. Horizontal Filtering: Up Scaling by 2.500

13.7.6 Horizontal Down Scaling

The parameters for horizontal down scaling by a factor of 2.5 (2.5 times as many input pixels as output pixels) are given below for a 640 x 480 input image down scaled to a 256 x 480 output image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	256	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	256	Output image width in pixels
Integer Increment	2	Increment = Scale factor for 2.5 down scaling
Fraction Increment	8000h = 0.500	Increment = 2 + 0.500

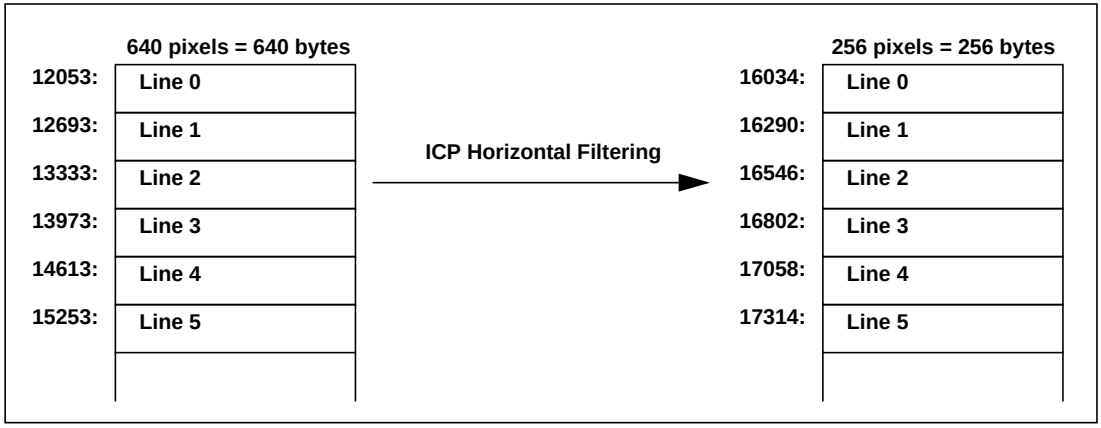


Figure 13-24. Horizontal Filtering: Down Scaling by 2.500

13.7.7 Horizontal Down Scaling by Large Factors

The parameters for large horizontal down scaling by a factor of 5 (5 times as many input pixels as output pixels) are given below for a 640 x 480 input image down scaled to a 128 x 480 output image. The large down scaling (GETB) bit is set in the Control word in this example. Setting the large down scaling bit is optional. When the large down scaling bit is set, the output pixels are filtered rather than the input pixels. It can improve the results on certain images, but not all. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	200h	No Bypass, GETB on for large down scaling)
Output Image Line Offset	128	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	128	Output image width in pixels
Integer Increment	5	Scale factor = 5 down scaling
Fraction Increment	0	Scale factor = 5 down scaling

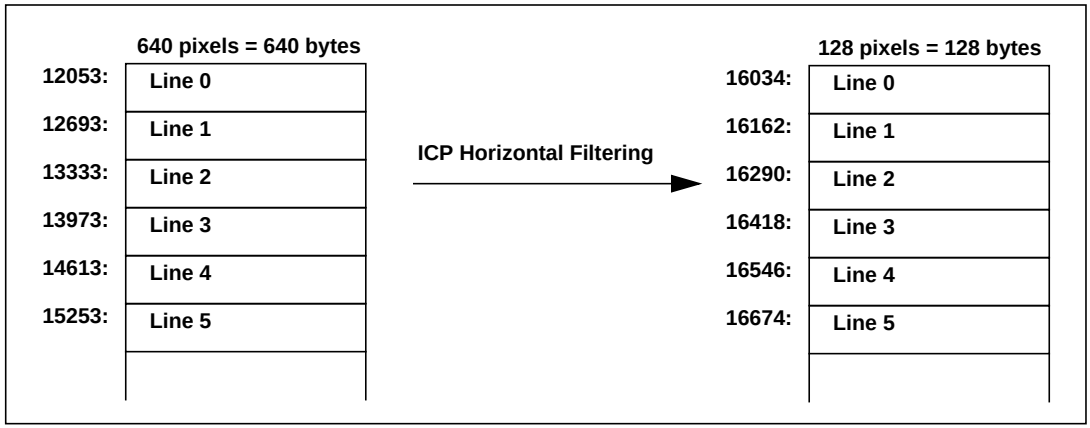


Figure 13-25. Horizontal Filtering: Down Scaling by 5.000

13.7.8 Horizontal Filtering: Interspersed to Co-sited Conversion

The parameters for interspersed to cosited conversion for U and V data are given below for a 640 x 480 input image filtered to a 640 x 480 output image (scale factor = 1.00 in this example). This is simple horizontal filtering with a scale factor of 1.00, but with a - 1/4 pixel offset. This offset corrects for the offset of the interspersed data. interspersed U and V data is offset in the positive direction by + 1/4 pixel with respect to the Y component. Setting the starting fraction to - 1/4 causes the input to effectively select pixels from points moved by 1/4 pixel in the negative (toward the start of the line) direction on the input line. The output line is the same as a filtered version of the input line except that it has been moved 1/4 pixel in the negative direction. By moving U and V by - 1/4 pixel, they now line up with the Y pixels, resulting in cosited YUV422 data.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	C000h	C000h = -0.2500 = - 1/4 pixel offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	1	Scale factor = 1 (no scaling)
Fraction Increment	0	Scale factor = 1 (no scaling)

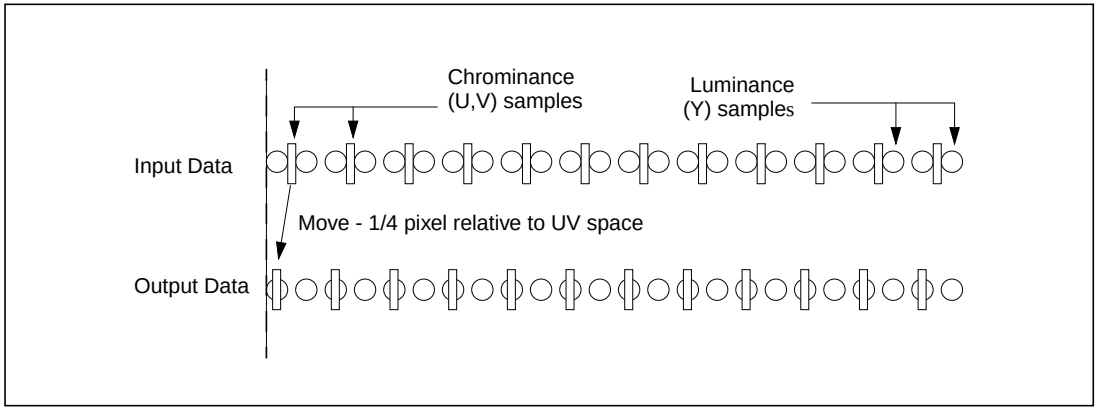


Figure 13-26. Horizontal Filtering: Interspersed to Co-Sited Conversion

13.7.9 Vertical Filtering Without Scaling (Scale Factor = 1)

Vertical filtering without scaling (i.e., scale factor = 1.000) is one of the simpler ICP operations. The parameters for this operation are given below for a 640 x 480 image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

The vertical filtering operation transfers an image through the ICP filter in columns of 64 pixels, and line by line within each column. Each line is read into the ICP fro DRAM, filtered and written back to DRAM. This memory to memory data transfer is shown in Figure 13-20.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	1	Scale factor = 1 (no scaling)
Fraction Increment	0	Scale factor = 1 (no scaling)

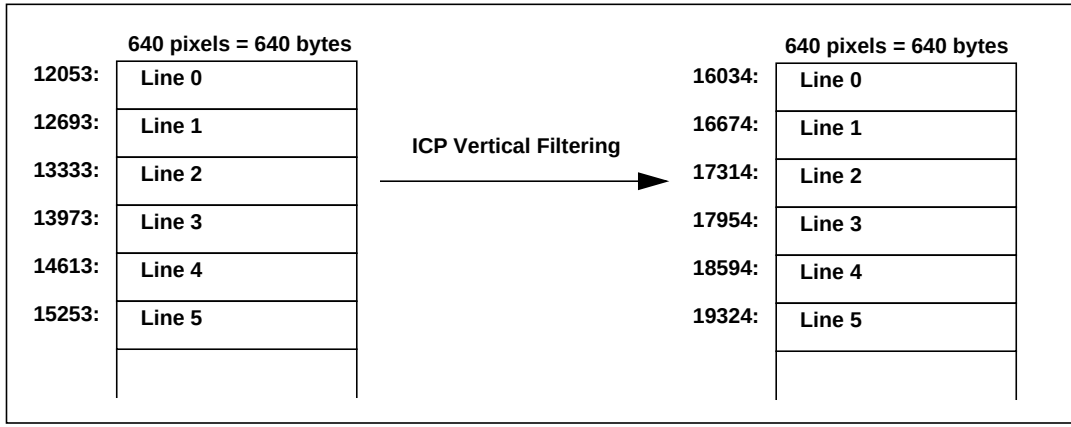


Figure 13-27. Vertical Filtering: Scale Factor = 1.000

13.7.10 Vertical Up Scaling

The parameters for horizontal upscaling by a factor of 2.5 (2.5 times as many output lines as input lines) are given below for a 640 x 480 input image upscaled to a 640 x 1200 output image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	1200	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	0	Increment = 1/ Scale factor for 2.5 up scaling
Fraction Increment	6666h = 0.400	Increment value = 0.400

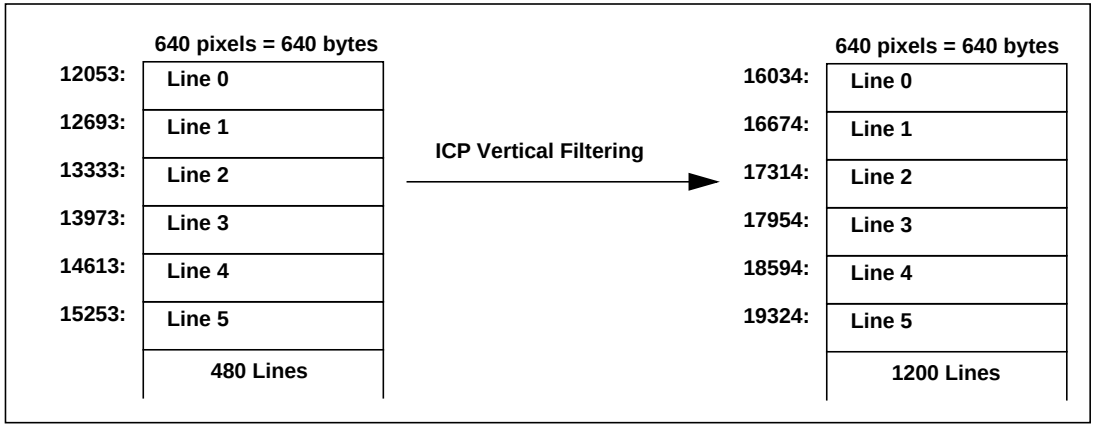


Figure 13-28. Vertical Filtering: Up Scaling by 2.50

13.7.11 Vertical Down Scaling

The parameters for horizontal down scaling by a factor of 2.5 (2.5 times as many input lines as output lines) are given below for a 640 x 480 input image down scaled to a 640 x 192 output image. Nominal values are chosen for the input and output starting addresses for demonstration purposes.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	0	0 = no offset
Input Image LIne Offset	640	Offset in bytes from first pixel of one line to the next
Input Image Height	480	Input image height in lines
Input Image Width	640	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	640	Offset in bytes from first pixel of one line to the next
Output Image Height	192	Output image height in lines
Output Image Width	640	Output image width in pixels
Integer Increment	2	Increment = Scale factor for 2.5 down scaling
Fraction Increment	8000h = 0.500	Increment = 2 + 0.500

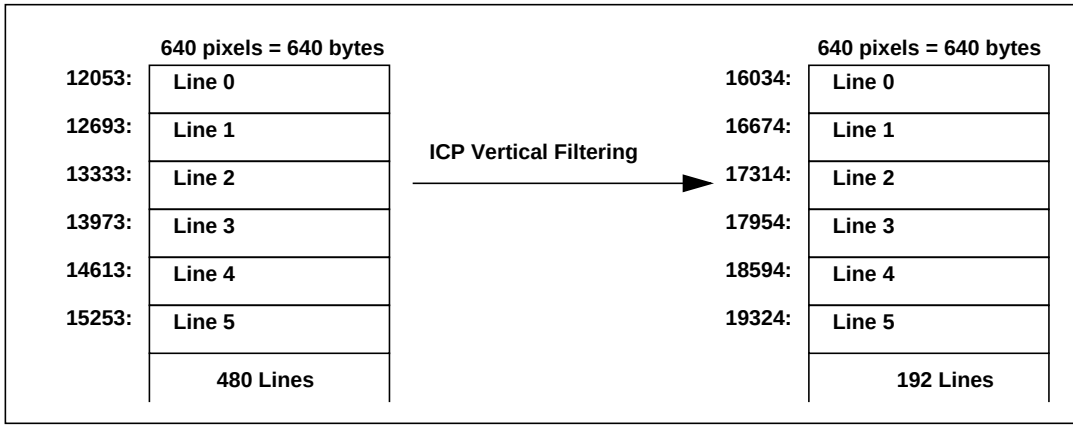


Figure 13-29. Vertical Filtering: Down Scaling by 2.50

13.7.12 YUV 4:2:0 to YUV 4:2:2 Conversion

The YUV 4:2:0 format has half as many chrominance (U and V) lines as YUV 4:2:2, and it positions these lines between the Y lines. To convert YUV 4:2:0 to YUV 4:2:2, you upscale the U and V components by a factor of 2.0 with an negative offset of - 0.250. The up scaling creates the correct number of lines, and the - 1/4 line offset corrects for the UV lines being between the Y lines.

Parameter	Value	Comments
Input Image Start Address	12053	Starting byte address of first pixel of input image
Y Counter Start Fraction	C000h	-0.2500 offset
Input Image LIne Offset	320	Offset in bytes from first pixel of one line to the next
Input Image Height	120	Input image height in lines
Input Image Width	320	Input image width in pixels
Output Image Start Address	16034	Starting byte address of output image
Control	0	Default (no Bypass, no GETB for large down scaling)
Output Image Line Offset	320	Offset in bytes from first pixel of one line to the next
Output Image Height	240	Output image height in lines
Output Image Width	320	Output image width in pixels
Integer Increment	0	Scale factor = 2 up scaling
Fraction Increment	8000h = 0.500	Increment value = 1/2 = 0.500

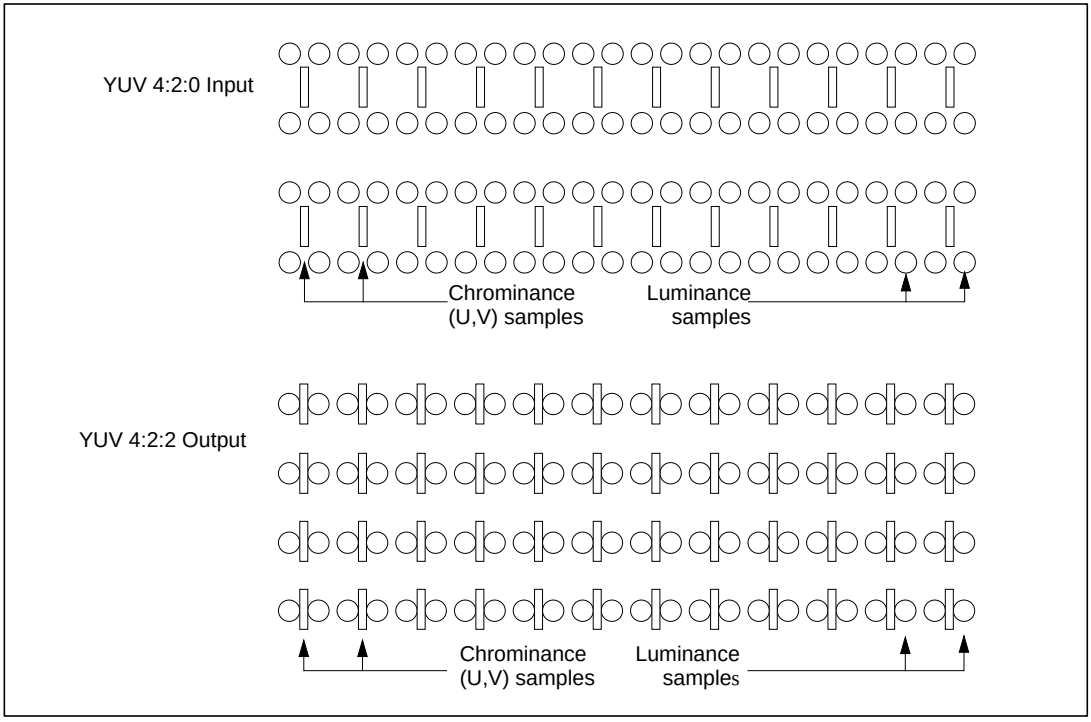


Figure 13-30. Vertical Filtering: YUV 4:2:0 to YUV 4:2:2 Conversion

13.7.13 Horizontal Filtering to YUV 4:2:2 to RGB 16, PCI Out

Horizontal filtering and conversion to RGB output is a common ICP operation. This example shows horizontal filtering of YUV 4:2:2 input data and conversion to RGB 16 output data. No scaling is performed (scale factor = 1.00), and no bit masking or overlay is used. The parameters for this operation are given below for a 640 x 480 image. Note that the output line offset must take into account the number of bytes per pixel of the output format.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image on PCI bus
Control	0807h	PCI Out, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

The horizontal filtering to RGB/YUV operation passes the Y, U and V components of an image through the ICP filter pixel by pixel. After scaling and filtering, each YUV triplet is converted to the selected RGB or YUV output code. The RGB/YUV pixels are created and written out line by line written to the PCI bus or back to DRAM. This data transfer is shown in Figure 13-31.

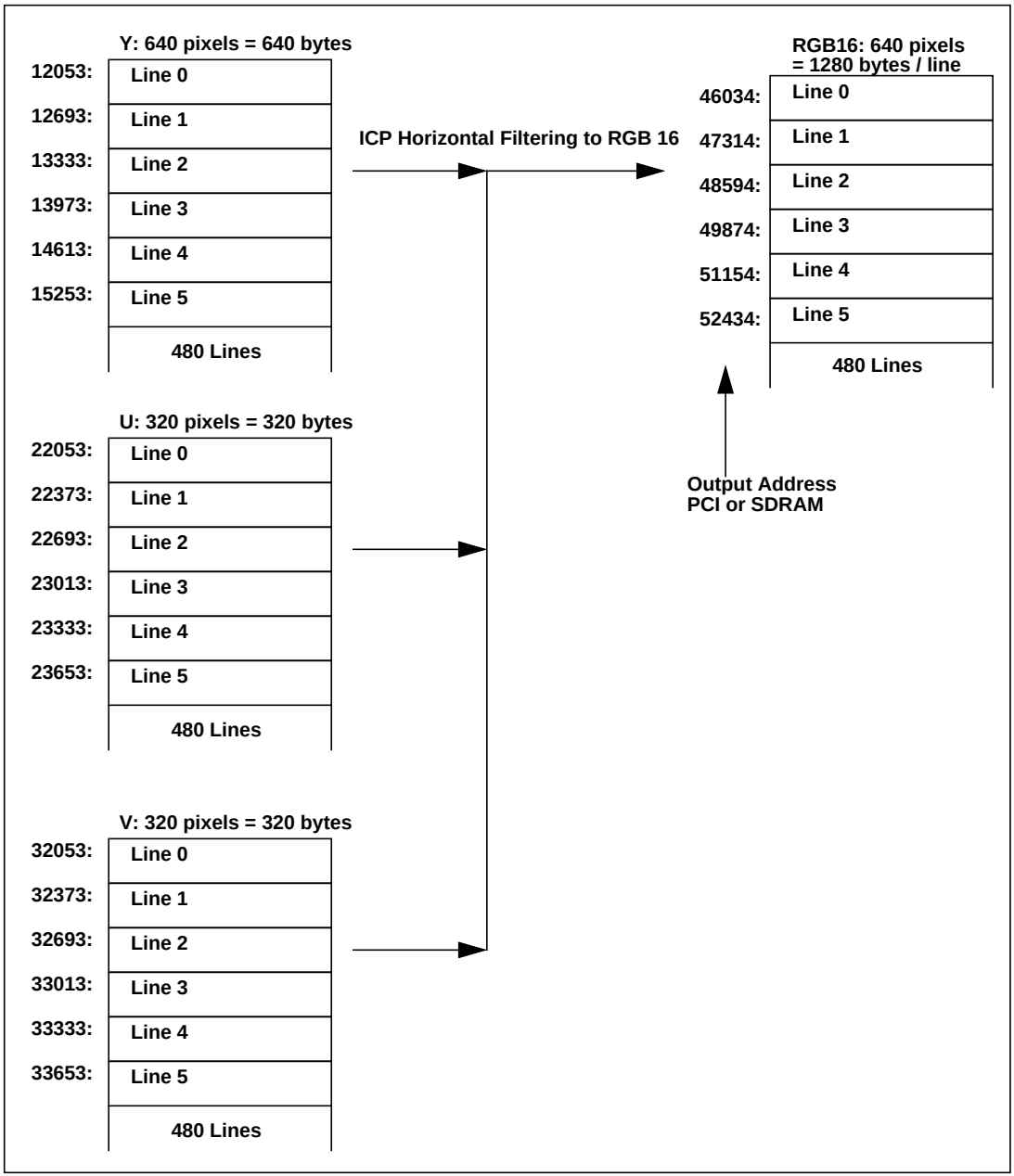


Figure 13-31. Horizontal Filtering to RGB/YUV: YUV 4:2:2 to RGB 16, PCI Out

13.7.14 Horizontal Filtering to YUV 4:2:2 to RGB 16, DRAM Out

This example shows horizontal filtering of YUV 4:2:2 input data and conversion to RGB 16 output data. It is the same as the previous PCI output case except that the output is to SDRAM.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image in SDRAM
Control	0007h	SDRAM Out, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

13.7.15 Horizontal Filtering to YUV 4:2:2 Interspersed to RGB 16

This example shows horizontal filtering of YUV 4:2:2 interspersed input data, conversion to RGB 16 output data and output to the PCI bus. It is the same as the previous original PCI output case except that the U and V components have a - 1/4 pixel offset.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	C000h	-1/4 pixel = YUV interspersed
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	C000h	-1/4 pixel = YUV interspersed
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	0807h	PCI Out, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

13.7.16 Horizontal Filtering to YUV 4:2:0 to RGB 16

This example shows horizontal filtering of YUV 4:2:0 interspersed input data, conversion to RGB 16 output data and output to the PCI bus. YUV 4:2:0 is similar to YUV 4:2:2 interspersed except that YUV 4:2:0 has only half the number of lines of U and V, and these lines are positioned between the Y lines. In this case, the YUV 420 mode bit is set in the Control word. This bit causes the U and V input lines to be used twice. This mode allows processing YUV 4:2:0 input data in one pass, with some loss of quality compared to a separate YUV 4:2:2 to YUV 4:2:0 conversion using vertical filtering. The U and V Start Fractions are also set to -1/4 because the U and V components are offset relative to the Y data.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	C000h	-1/4 pixel = YUV interspersed
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	C000h	-1/4 pixel = YUV interspersed
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	2807h	YUV 420, PCI Out, no overlay or bit mask, RGB16 output, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

The data transfer for the horizontal filtering of YUV 4:2:0 to RGB/YUV operation is shown in Figure 13-32.

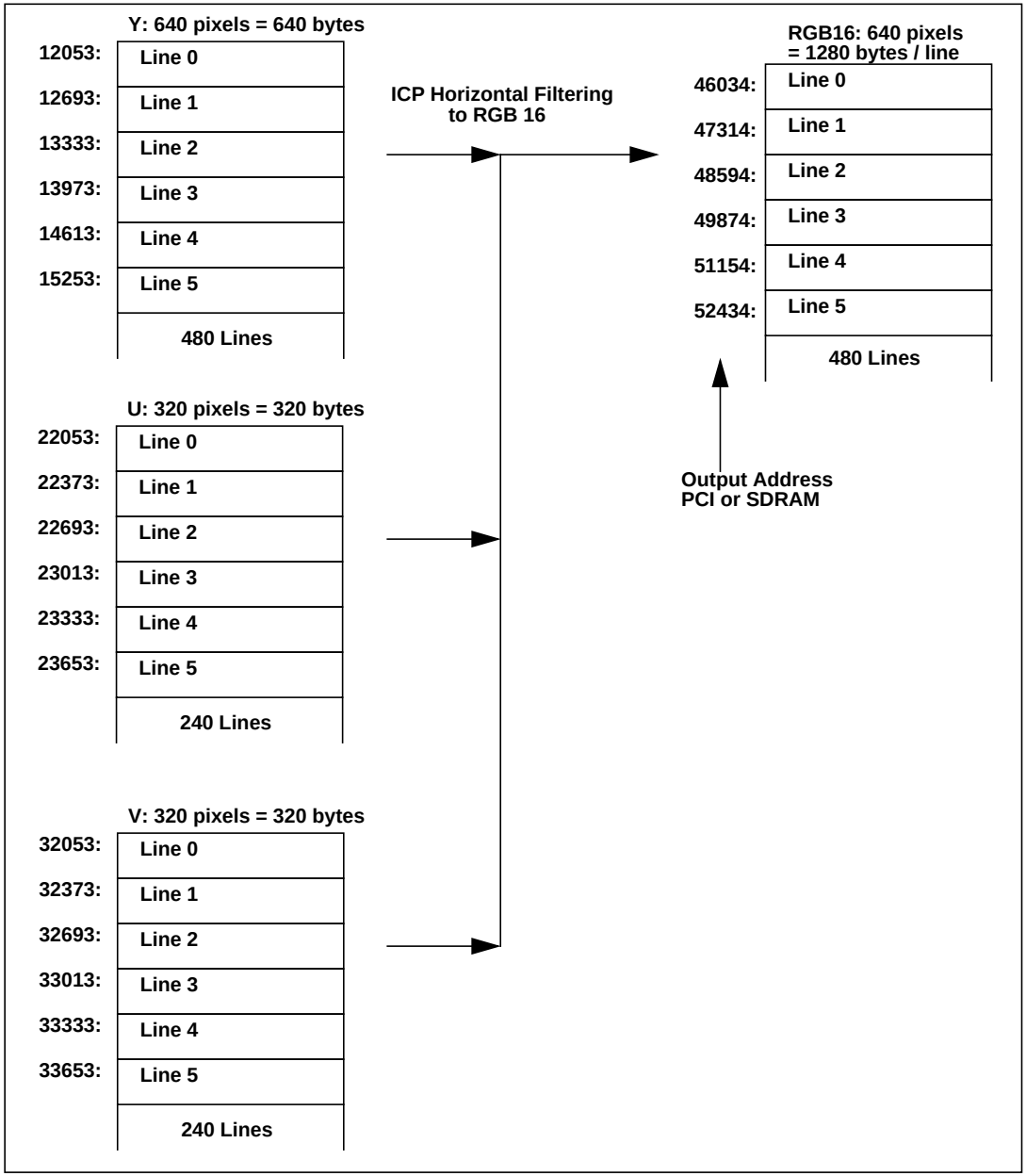


Figure 13-32. Horizontal Filtering to RGB/YUV: YUV 4:2:0 to RGB 16, PCI Out

13.7.17 Horizontal Filtering to YUV 4:1:1 NTSC to RGB 16

This example shows horizontal filtering of 640 x 480 YUV 4:1:1 NTSC cosited input data, conversion to RGB 16 output data and output to the PCI bus. YUV 4:1:1 is similar to YUV 4:2:2 cosited except that YUV 4:1:1 has one pixel of U and V for every 4 Y pixels. This requires a 4x up scaling of U and V. Also, the Y, U and V lines are interlaced. The YUV 4:1:1 data is processed in two passes to handle the interlaced format. In each pass, the offset values are for two lines instead of one line. This causes the ICP to skip every other line. In the first pass, lines 0 through 239 are processed. In the second pass, lines 263 through 502 are processed. The parameter table is shown for the first pass.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	1280	Y Offset in bytes = 2 lines for 4:1:1 interlaced data
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	240	Y Input image height in lines: YUV 4:1:1, first interlaced pass
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	640	U Offset in bytes = 2 lines for 4:1:1 interlaced data
U Integer Increment	0	Scale factor = 4.0 = 4x Y scale factor for YUV 411 input
U Fraction Increment	4000h	Scale factor = 4.0
U Input Image Height	240	U Input image height in lines: YUV 4:1:1, first interlaced pass
U Input Image Width	160	U Input image width in pixels: YUV 4:1:1, first interlaced pass
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	640	V Offset in bytes = 2 lines for 4:1:1 interlaced data
Y Integer Increment	0	Scale factor = 4.0 = 4x Y scale factor for YUV 411 input
V Fraction Increment	4000h	Scale factor = 4.0
V Input Image Height	240	V Input image height in lines: YUV 4:1:1, first interlaced pass
V Input Image Width	160	V Input image width in pixels: YUV 4:1:1, first interlaced pass
Output Image Start Address	46034	Starting byte address of output image
Control	0807h	PCI Out, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

The data transfer for the horizontal filtering of YUV 4:1:1 to RGB/YUV operation is shown in Figure 13-33.

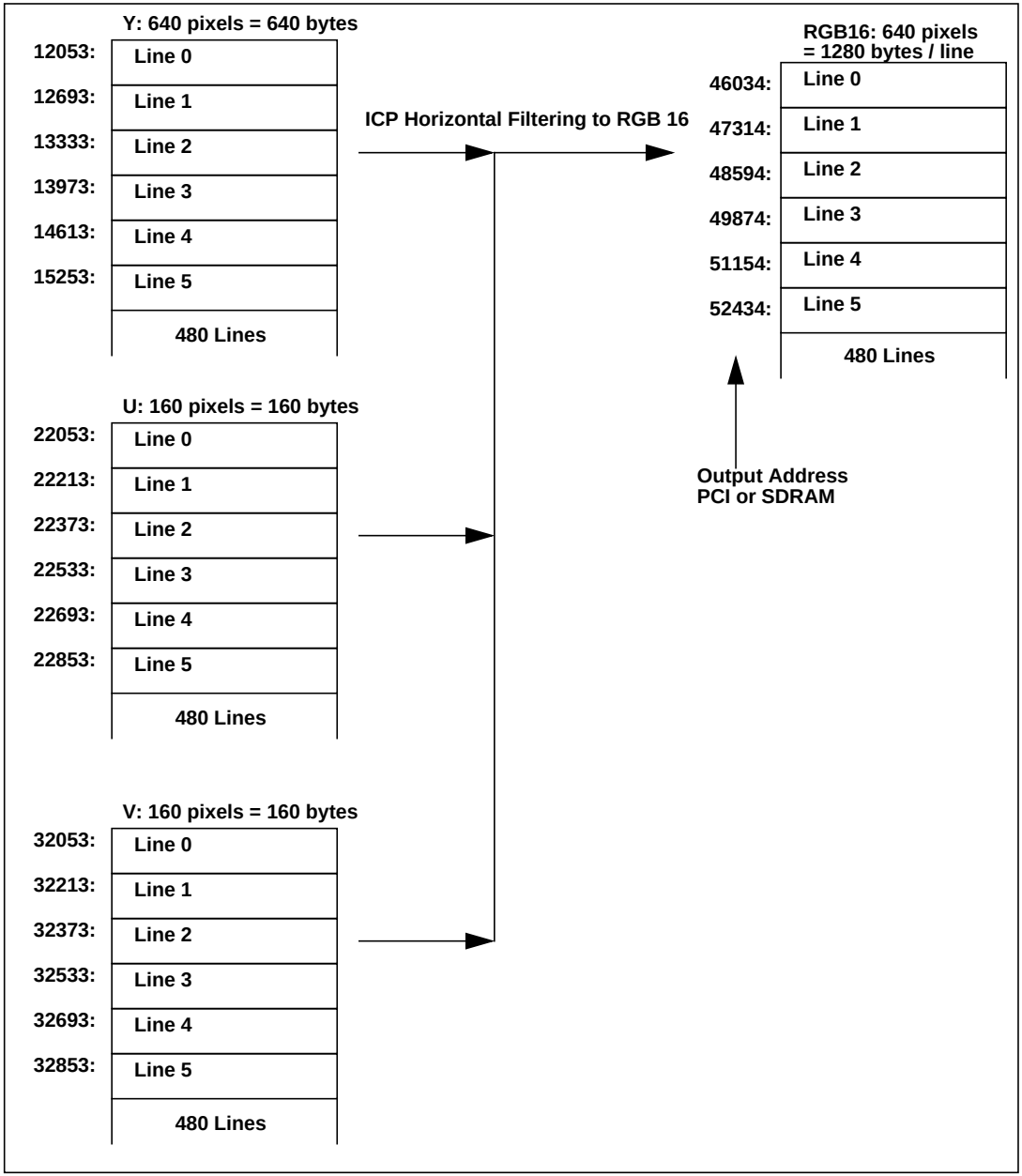


Figure 13-33. Horizontal Filtering to RGB/YUV: YUV 4:1:1 NTSC to RGB 16, PCI Out

13.7.18 Horizontal Filtering to RGB/YUV with RGB 24+a Overlay

This example shows horizontal filtering of YUV 4:2:2 input data, conversion to RGB 16 output data and addition of an RGB 24+a overlay. No scaling is performed (scale factor = 1.00), and no bit masking is used. The overlay is 100 pixels wide by 40 pixels high and begins at pixel 20 and line 10. The parameters for this operation are given below for a 640 x 480 image. Note that the overlay line offset must take into account the number of bytes per pixel of the overlay data, and that the output line offset must take into account the number of bytes per pixel of the output format.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	1802h	PCI Out, overlay RGB24+a, no bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	41536	Starting byte address of overlay
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending (not used)
Overlay Line Offset	400	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	20	Pixel number in output line of first overlay pixel
Overlay Start Line	10	Line number in output image of first overlay line
Overlay End Pixel	119	Pixel number in output line of last overlay pixel
Overlay End Line	49	Line number in output image of last overlay line

The data transfer for the horizontal filtering of YUV 4:2:2 to RGB/YUV with overlay is shown in Figure 13-34.

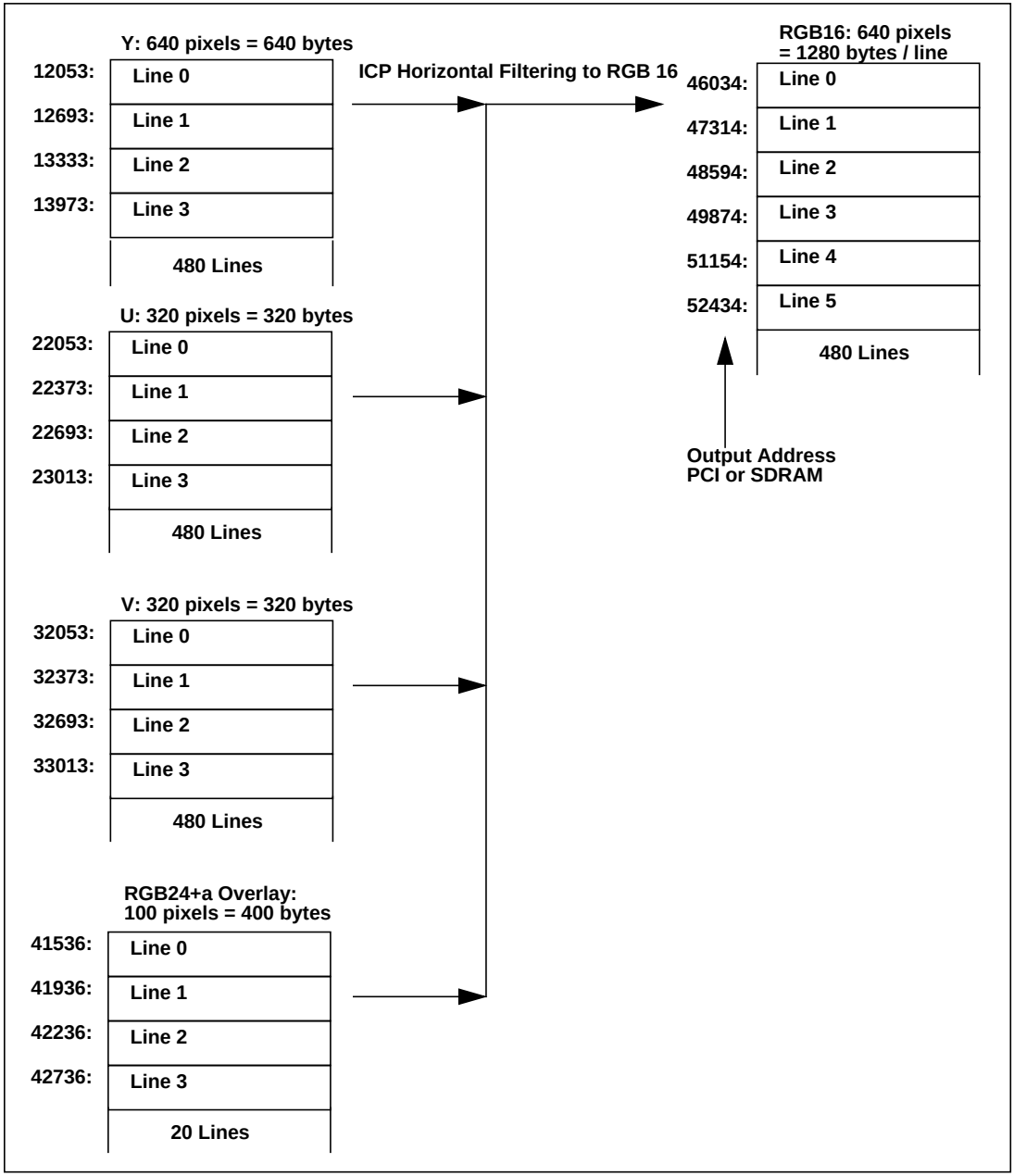


Figure 13-34. Horizontal Filtering to RGB/YUV: YUV 4:2:2 to RGB 16 with Overlay

13.7.19 Horizontal Filtering to RGB/YUV with RGB 15+a Overlay

This example shows horizontal filtering of YUV 4:2:2 input data, conversion to RGB 16 output data and addition of an RGB 15+a overlay. No scaling is performed (scale factor = 1.00), and no bit masking is used. The overlay is 100 pixels wide by 40 pixels high and begins at pixel 20 and line 10. The parameters for this operation are given below for a 640 x 480 image. Note that the overlay line offset must take into account the number of bytes per pixel of the overlay data, and that the output line offset must take into account the number of bytes per pixel of the output format.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	1802h	PCI Out, overlay RGB24+a, no bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	41536	Starting byte address of overlay
Alpha 1 & Alpha 0	4000h	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	200	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	20	Pixel number in output line of first overlay pixel
Overlay Start Line	10	Line number in output image of first overlay line
Overlay End Pixel	119	Pixel number in output line of last overlay pixel
Overlay End Line	49	Line number in output image of last overlay line

13.7.20 Horizontal Filtering to RGB 16 with RGB 15+a Overlay and Bit Masking

This example shows horizontal filtering of YUV 4:2:2 input data, conversion to RGB 16 output data, with an RGB 15+a overlay and bit masking. No scaling is performed (scale factor = 1.00), and no bit masking is used. The overlay is 100 pixels wide by 40 pixels high and begins at pixel 20 and line 10. The parameters for this operation are given below for a 640 x 480 image. Note that the overlay line offset must take into account the number of bytes per pixel of the overlay data, and that the output line offset must take into account the number of bytes per pixel of the output format. Note that the bit mask offset is 1/8 of the output data pixel count because the bit mask packs 8 bits of pixel mask per byte.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
U Fraction Increment	8000h	Scale factor = 2.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	0	Scale factor = 2.0 = 2x Y scale factor for YUV 422 input
V Fraction Increment	8000h	Scale factor = 2.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	1C02h	PCI Out, overlay RGB24+a, bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	68773	Starting byte address of bit mask
Bit Map Line Offset	80	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	41536	Starting byte address of overlay
Alpha 1 & Alpha 0	4000h	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	200	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	20	Pixel number in output line of first overlay pixel
Overlay Start Line	10	Line number in output image of first overlay line
Overlay End Pixel	119	Pixel number in output line of last overlay pixel
Overlay End Line	49	Line number in output image of last overlay line

The data transfer for the horizontal filtering of YUV 4:2:2 to RGB/YUV with overlay is shown in Figure 13-35.

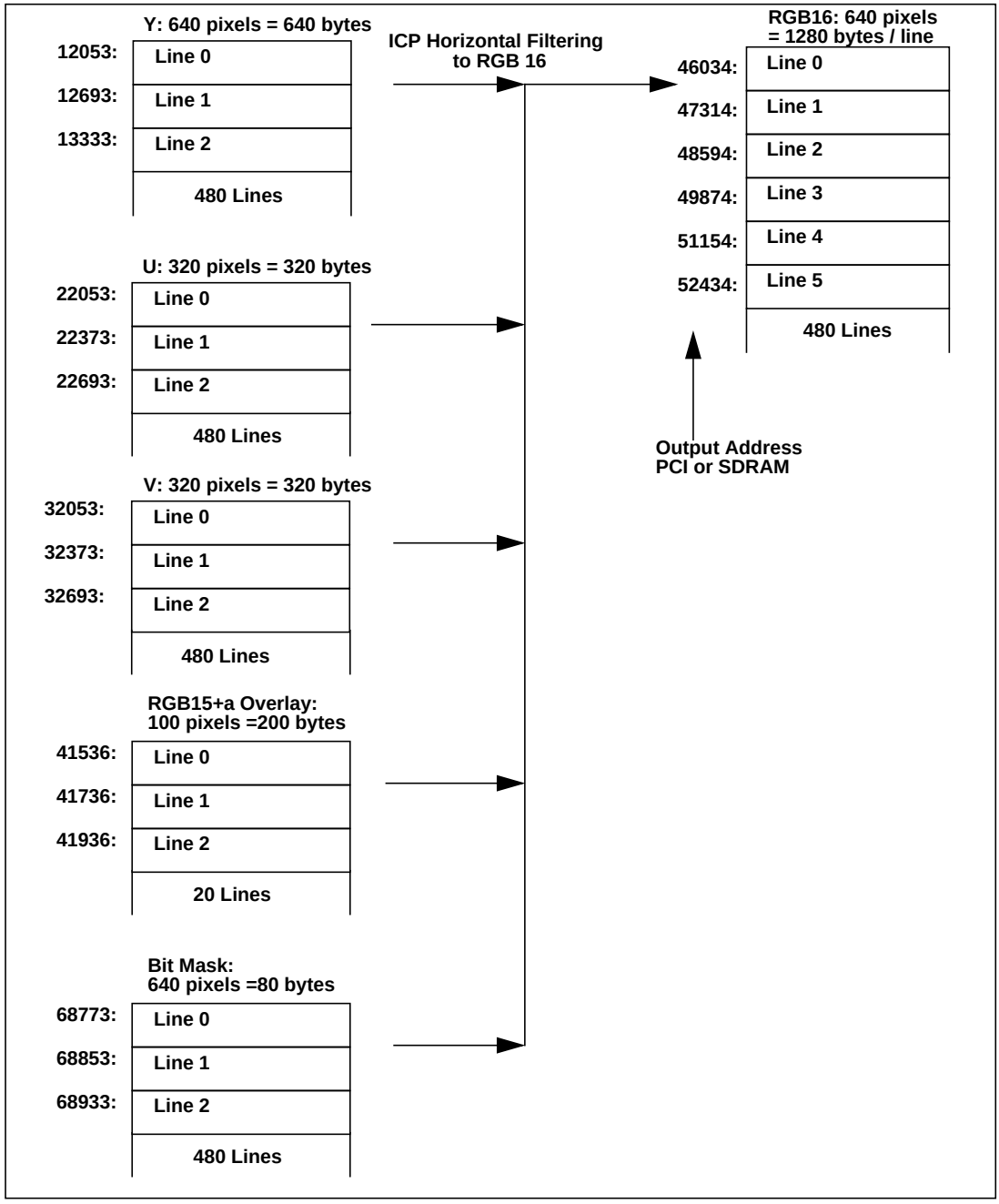


Figure 13-35. Horizontal Filtering to RGB/YUV: YUV 4:2:2 to RGB 15+a with Overlay and Bit Masking

13.7.21 Horizontal Filtering to YUV 4:2:2 Planar to YUV 4:2:2 Composite

This example shows horizontal filtering of YUV 4:2:2 planar input data and conversion to YUV 4:2:2 composite (CCIR 656 style) output data. No scaling is performed (scale factor = 1.00), and no bit masking or overlay is used. The parameters for this operation are given below for a 640 x 480 image. Note that the YUV 422 sequencing is used to generate the output data, and therefore no scaling of U or V is required.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	1	Scale factor = 1.0 = no scaling for YUV 422 sequencing
U Fraction Increment	0	Scale factor = 1.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0 = no scaling for YUV 422 sequencing
V Fraction Increment	0	Scale factor = 1.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	4807h	PCI Out, YUV 422 sequencing, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

13.7.22 Horizontal Filtering to YUV 4:2:2 to RGB 16 with 422 Sequencing

This example shows horizontal filtering of YUV 4:2:2 input data and conversion to RGB 16 output data. No scaling is performed (scale factor = 1.00), and no bit masking or overlay is used. The parameters for this operation are given below for a 640 x 480 image. Note that the YUV 422 sequencing is used to generate the output data, and therefore no scaling of U or V is required. This gives higher throughput (less processing time) at the expense of somewhat lower image quality because the U and V pixels are duplicated, not scaled before conversion to RGB.

Parameter	Value	Comments
Y Input Image Start Address	12053	Starting byte address of first pixel of input image Y component
Y Counter Start Fraction	0	0 = no offset
Y Input Image LIne Offset	640	Y Offset in bytes from first pixel of one line to the next
Y Integer Increment	1	Scale factor = 1.0
Y Fraction Increment	0	Scale factor = 1.0
Y Input Image Height	480	Y Input image height in lines
Y Input Image Width	640	Y Input image width in pixels
U Input Image Start Address	22053	Starting byte address of first pixel of input image U component
U Counter Start Fraction	0	0 = no offset = YUV cosited
U Input Image LIne Offset	320	U Offset in bytes from first pixel of one line to the next
U Integer Increment	1	Scale factor = 1.0 = no scaling for YUV 422 sequencing
U Fraction Increment	0	Scale factor = 1.0
U Input Image Height	480	U Input image height in lines
U Input Image Width	320	U Input image width in pixels
V Input Image Start Address	32053	Starting byte address of first pixel of input image V component
V Counter Start Fraction	0	0 = no offset = YUV cosited
V Input Image LIne Offset	320	V Offset in bytes from first pixel of one line to the next
Y Integer Increment	2	Scale factor = 1.0 = no scaling for YUV 422 sequencing
V Fraction Increment	0	Scale factor = 1.0
V Input Image Height	480	V Input image height in lines
V Input Image Width	320	V Input image width in pixels
Output Image Start Address	46034	Starting byte address of output image
Control	4807h	PCI Out, YUV 422 sequencing, no overlay or bit mask, RGB16 output code, big endian
Output Image Line Offset	1280	Byte offset for RGB 16 @ 2 bytes/pixel = 2 x 640 = 1280
Output Image Height	480	Output image height in lines
Output Image Width	640	Output image width in pixels
Bit Map Start Address	0	Starting byte address of bit mask (not used)
Bit Map Line Offset	0	Offset in bytes from first pixel of one bit map line to the next
Overlay Start Address	0	Starting byte address of overlay (not used)
Alpha 1 & Alpha 0	0	Alpha 1 and Alpha 0 register values for alpha blending
Overlay Line Offset	0	Offset in bytes from first pixel of one overlay line to the next
Overlay Start Pixel	0	Pixel number in output line of first overlay pixel
Overlay Start Line	0	Line number in output image of first overlay line
Overlay End Pixel	0	Pixel number in output line of last overlay pixel
Overlay End Line	0	Line number in output image of last overlay line

14.1 INTRODUCTION

The Variable Length Decoder (VLD) is a coprocessor to TriMedia's DSPCPU which assumes responsibility for the Huffman decoding process in MPEG1 and MPEG2. The VLD receives as input a pointer to an MPEG or MPEG2 bit stream as well as some configuration information, all of which is loaded through MMIO registers. The VLD produces as output a data structure which contains all of the information necessary to complete the video decoding process. A DMA unit inside the VLD fetches the bit stream from SDRAM and writes the VLD output to SDRAM. Control and synchronization of VLD by the DSPCPU is achieved through MMIO registers. This document describes a programmers view of the VLD. Figure 1 is a high level block diagram of the VLD.

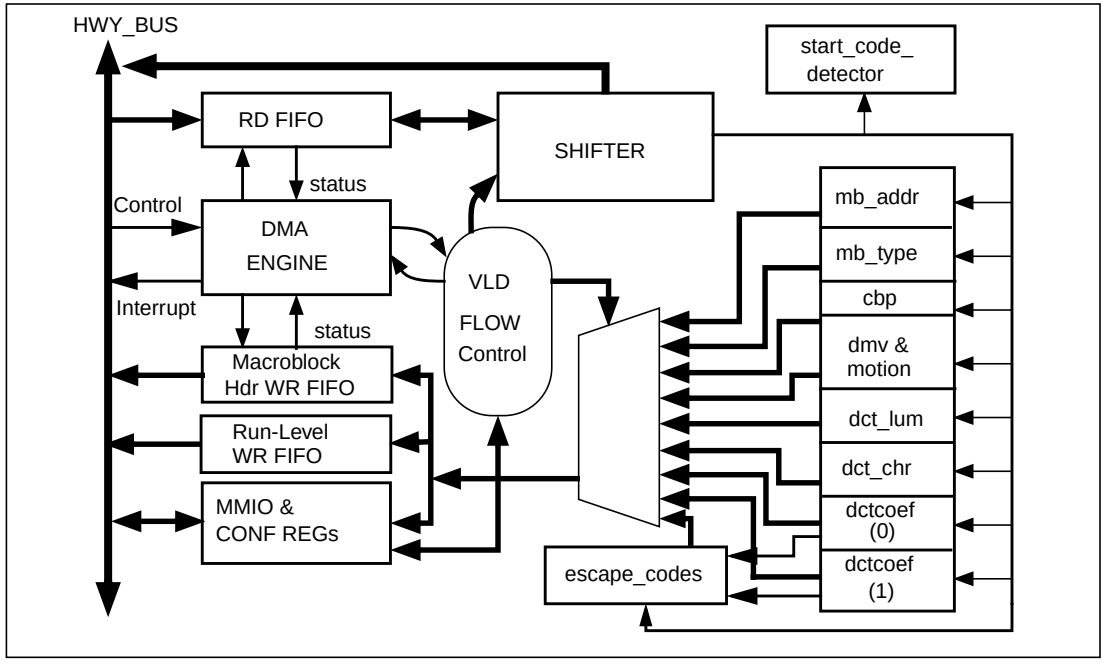


Figure 14-1. VLD Block Diagram

14.2 VLD OPERATION

After initialization, the DSPCPU will control the VLD through the VLD command register. There are currently five commands supported by the VLD:

- Shift the bit stream by some number of bits
- Search for the next start code
- Reset the VLD
- Flush the output fifos (i.e. write all data in the output fifos to SDRAM)
- Parse some number of macroblocks

The normal mode of operation will be for the DSPCPU to request the VLD to parse some number of macroblocks. Once the VLD has begun parsing macroblocks it may stop for any one of the following reasons:

- The command was completed with no exceptions
- A start code was detected
- An error was encountered in the bit stream
- The VLD input DMA completed and the VLD is stalled waiting for more data

- One of the VLD output DMAs has completed and the VLD is stalled because the output FIFO is full

Under normal circumstances, the DSPCPU can be interrupted whenever the VLD halts.

Consider the case in which the VLD has encountered a start code. At this point, the VLD will halt and set the status flag which indicates that a start code has been detected. This flag will generate an interrupt to the DSPCPU. Upon entering the interrupt routine, the DSPCPU will read the VLD status register to determine the source of the interrupt. Once it has been determined that a start code has been encountered, the CPU will read 8 bits from the VLD shift register to determine the type of start code that has been encountered. If a slice start code has been encountered, the DSPCPU will read from the shift register the slice quantization scale and any extra slice information. The slice quantization scale will then be written back to the VLD quantizer scale register. Before exiting the interrupt routine, the VLD will clear the start code bit in the status register and issue a new command to process the remaining macroblocks.

14.3 VLD OUTPUT

The DSPCPU will allocate a section of SDRAM for the VLD to store its output. The VLD will store Macroblock header information and transform coefficients in two separate areas of memory in order to facilitate prefetching the predictors for motion compensated macroblocks. Pointers to these memory areas will be communicated to the VLD DMA through the Current Write Address registers. There are two fifos and associated DMAs for VLD output, one for macroblock header information and the other for run-length encoded DCT coefficients. For each MPEG2 macroblock parsed by the VLD, six 32 bit words of macroblock header information will be output from the VLD. The format of these six words is depicted in the Figure 2 below. For each MPEG macroblock parsed by the VLD, the macroblock header output format will be the same as for an MPEG2 macroblock with the exception that there are no second motion vectors. The DCT coefficients associated with the macroblock are output to a separate memory area and each DCT coefficient is represented as one 32 bit quantity (16 bits of run and 16 bits of level). For intra blocks, the DC term is expressed as 16 bits of DC size and a 16 bit value whose most significant bits (the number of bits used for DC level is determined by DC size) represent the DC level. Each block of DCT coefficients is terminated by a run value of 0xff. The values output by the VLD for each field in the macroblock header output structure are defined by the MPEG2 standard.

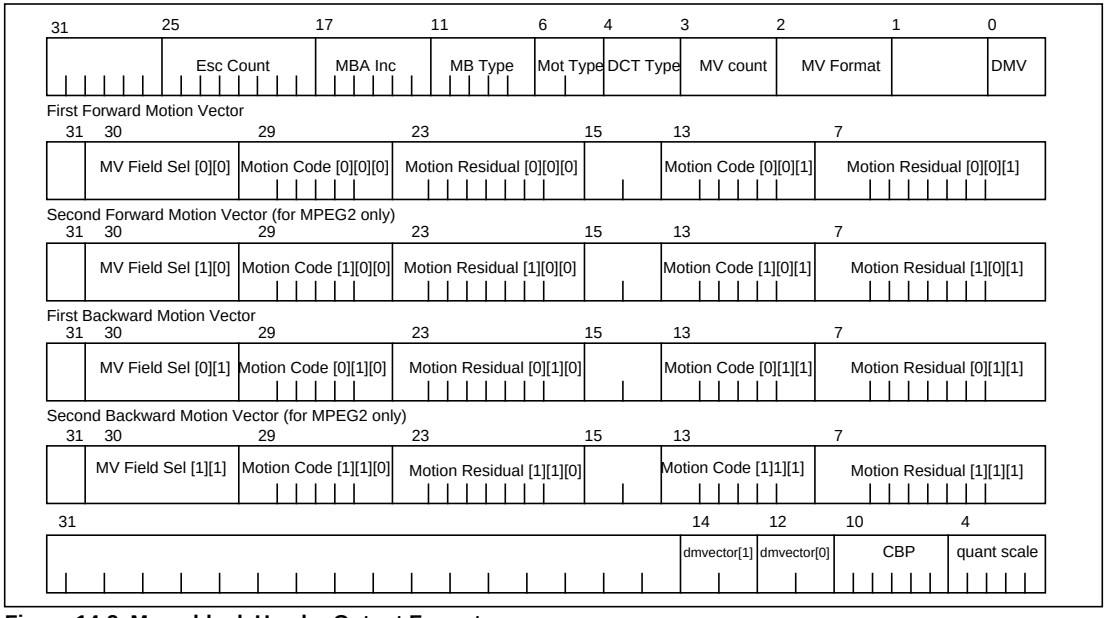


Figure 14-2. Macroblock Header Output Format

14.4 VLD CONTROL AND STATUS REGISTERS

VLD Status

This register contains current status information which is most pertinent to the normal operation of an MPEG video decode application. Writing a one to bits one through five clears that bit. Bit 0 (Command Done) is cleared only by issuing a new command. Writing a one to bit zero of the status register will result in undefined behavior of the VLD. Note that several status bits may be asserted simultaneously. Also note that shadow copies of the VLD command count and the DMA Run/Level write count are included here for programming convenience. Writes to either of these two fields will be ignored????. The VLD STATUS register contains the following fields:

Table 14-1. VLD STATUS (R/W)

Name	Size (bits)	Description
VLD Command Done	1	Indicates successful completion of current command
Start Code Detected	1	VLD encountered 0x000001 while executing current command
Bitstream Error	1	VLD encountered an illegal Huffman code or an unexpected start code
DMA Input Done	1	DMA transfer has completed and VLD is stalled waiting on more input
DMA Macroblock Header Output Done	1	Macroblock Header DMA transfer has completed
DMA Run/Level Output Done	1	Run/Level DMA transfer has completed
Reserved	2	Reserved for future expansion
VLD Macroblock Count	8	Indicates number of macroblocks remaining to be parsed by the VLD
DMA Run/Level Write Count	12	Indicates number of writes remaining before DMA Write transfer completes (a shadow copy of the DMA current write count register)

VLD Interrupt Enable

This 6 bit read/write register allows the DSPCPU to control which of the 6 least significant bits of the VLD Status Register will generate an interrupt. Writing a one to any of these bits enables the interrupt for the corresponding bit in the status register.

VLD Control

This read/write register controls the operation of the VLD and its DMA.

Table 14-2. VLD Control (R/W)

Name	Size (bits)	Description
End of Sequence	1	Force decode regardless of input fifo status
Little Endian	1	Force VLD to operate in Little Endian Mode

14.5 VLD DMA REGISTERS

VLD DMA Current Read Address

This 32 bit read/write register contains the byte address from which the VLD is currently reading.

VLD DMA Current Read Count

This 32 bit read/write register contains the number of bytes remaining to be read before the current DMA is completed. Note that reading this register returns 32 bits, of which the bottom 12 are the VLD Current Read Count and the top 16 are the current VLD DMA Run-Level Current Write Count.

VLD DMA Macroblock Header Current Write Address

This 32 bits read/write register contains the 64 byte block aligned address of the next write to SDRAM from the VLD Macroblock header fifo.

VLD DMA Macroblock Header Current Write Count

This 9 bit read/write register contains the number of SDRAM writes remaining before the current DMA from the macroblock header fifo is completed.

VLD DMA Run-Level Current Write Address

This 32 bits read/write register contains the 64 byte block aligned address of the next write to SDRAM from the VLD run-level fifo.

VLD DMA Run-Level Current Write Count

This 12 bit read/write register contains the number of SDRAM writes remaining before the current DMA from the VLD run-level fifo is completed. Note that reading this register returns 32 bits, of which the bottom 12 are the VLD Current Read Count and the top 16 are the current VLD DMA Run-Level Current Write Count.

14.6 VLD OPERATIONAL REGISTERS

VLD Command

This read/write register indicates the next action to be taken by the VLD. Some commands have an associated count which resides in the least significant 8 bits of this register. There are currently 5 commands which the VLD recognizes:

- Parse “count” macroblocks
- Shift the bitstream “count” bits (“count” must be less than or equal to 16)
- Search for the next start code
- Reset the VLD
- Flush the output fifos to SDRAM

The DSPCPU must wait for the VLD to halt before the next command can be issued. Note that there are several ways in which a command may complete. Only a successful completion is indicated by the command done bit in the status register. A command may complete unsuccessfully if a start code or an error is encountered before the requested number of items has been processed. Note also that expiration of a DMA count does not constitute completion of a command. When a DMA count expires the VLD is stalled waiting for a new DMA to be initiated, it is not halted.

Table 14-3. VLD Command Register

Name	Size (bits)	Description
Count	8	Count for current command
Command	4	VLD command to be executed

Table 14-4. VLD Commands

Command Name	Command Encoding
Shift Bitstream	0x1
Parse Macroblock	0x2
Search for Next Start Code	0x3
Reset VLD	0x4
Flush Write FIFO’s	0x8

VLD Shift Register

This read only register is a shadow of the VLD’s operational shift register and it allows the DSPCPU to access the bitstream through the VLD. Bits 0 through 15 are the current contents of the VLD shift register. Bit 31 to 16 are RESERVED and should be treated as undefined by the programmer.

VLD Quantizer Scale

This 5 bit register read/write register contains the quantization scale code to be output by the VLD until it is overridden by a macroblock quantizer scale code.

VLD Picture Info

This 32 bit read/write register contains the picture layer information necessary for the VLD to parse the macroblocks within that picture. Again, the values of each of these fields is determined by the appropriate standard (MPEG or MPEG2).

Table 14-5. VLD Picture Info Register

Name	Size (bits)	Description
picture type	2	I, P, or B picture
picture structure	2	field or frame picture
frame prediction frame dct	1	specifies that this picture uses only frame prediction and frame dct
intra vlc	1	Use DCT table zero or one
conceal mv	1	concealment vectors present in the bitstream
reserved	6	
mpeg 2 mode	1	switches VLD between mpeg and mpeg2 decoding; 1 = mpeg2 mode
reserved	2	reserved
horizontal forward rsize	4	size of residual motion vector
vertical forward rsize	4	size of residual motion vector
horizontal backward rsize	4	size of residual motion vector
vertical backward rsize	4	size of residual motion vector

14.7 VLD ADDRESS MAP

The following table summarizes the addresses of the memory mapped input/output (MMIO) registers within the VLD.

Table 14-6. VLD Address Map

Register Name	MMIO_base offset
Command	0x102800
Shift Register	0x102804
Quant Scale	0x102808
Picture Info	0x10280C
Status	0x102810
Interrupt Mask	0x102814
Control	0x102818
DMA Input Address	0x10281C
DMA Input Count	0x102820
DMA Macroblock Header Output Address	0x102824
DMA Macroblock Header Output Count	0x102828
DMA Run/Length Output Address	0x10282C
DMA Run/Length Output Count	0x102830

14.8 FUTURE ENHANCEMENTS

The VLD should be restartable at the macroblock (?????) boundary so that it can handle a PES stream in which the PES packet length is not specified (see *section 2.4.3.7 of ISO/IEC 13818-1, definition of PES_packet_length*).

If the VLD ever produces output for an inverse quantizer/ inverse DCT unit, an extra error condition should be added such that if the VLD ever sees a block which contains more than 64 coefficients (including zeroes) an error is flagged.

I²C Interface — Chapter 15

by Robert Bradfield, Robert Nichols

15.1 I²C OVERVIEW

TM1000 includes an I²C interface which can be used to control many different multimedia devices such as:

- DMSDs - Digital Multi-Standard Decoders
- DENCs - Digital Encoders
- Digital Cameras
- I²C - Parallel I/O expanders

The key features of the I²C interface are:

- Supports I²C single master mode
- I²C data rate up to 400 kbits/sec
- Support for both the 7-bit and 10-bit addressing options of the I²C specification
- Provisions for full software use of I²C interface pins for implementing software I²C or similar protocols

Note that the I²C pins are also used to load the initial boot parameters and/or code from a serial EEPROM as described in Section 12, “System Boot”. The boot logic is only active upon TM1000 hardware reset, and quiescent afterwards.

A typical system using the I²C interface is presented in Figure 15-1. The TM1000 is connected as a master to a series of slave devices through SCL and SDA. Note that the bus has one pullup resistor for each of the clock and data lines.

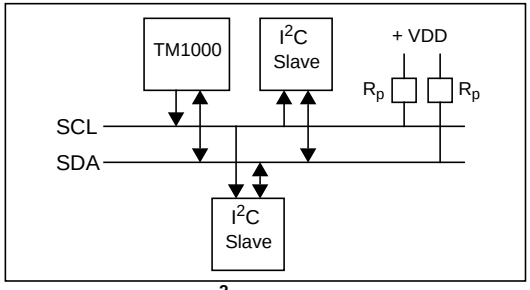


Figure 15-1. Typical I²C System Implementation

15.2 EXTERNAL INTERFACE

The I²C external interface is composed of two signals as shown in Table 15-1.

Table 15-1. I²C External Interface

Signal	Type	Description
IIC-SDA	I/O	I²C serial data
IIC-SCL	O	I²C clock

15.3 I²C REGISTER SET

The I²C user interface consists of four registers visible to the programmer. The registers are mapped into the MMIO address space and are fully accessible to the programmer. Figure 15-2 shows the I²C register set.

15.3.1 IICAR Register

The IICAR is the I²C address register and is used in both master receive and transmit modes. This register is written with the address(es) of the I²C slave device and the bytecount for transmit/receive. Table 15-2 lists the bitfield definitions for the IICAR register.

Table 15-2. IICAR Register

Bits	Field Name	Definition
31:25	ADDRESS	7-bit slave device address.
24	DIRECTION	Read/Write control bit
23:16	ABYTE2	Slave device address byte extension. Used for 10-bit addressing mode only.
15:8	COUNT	Byte count of requested transfer
7:0	reserved	Read as “0”

The ADDRESS bitfield has two modes:

- **7 bit Normal Mode:** 7-bit I²C addressing - ADDRESS must be programmed to contain the 7 bits of the desired slave address
- **10 bit Extended Mode:** 10-bit I²C addressing - ADDRESS must be programmed to contain the binary code 11110xx where ‘xx’ is the two msbits of the slave address. The ABYTE2 must contain the 8 lsbits of the slave address. See Section 15.5, “I2C HARDWARE Operation MODE,” for complete programming details.

The DIRECTION bitfield controls read/write operation on the I²C interface. The bit definition is:

- DIRECTION = 0 –> I²C write

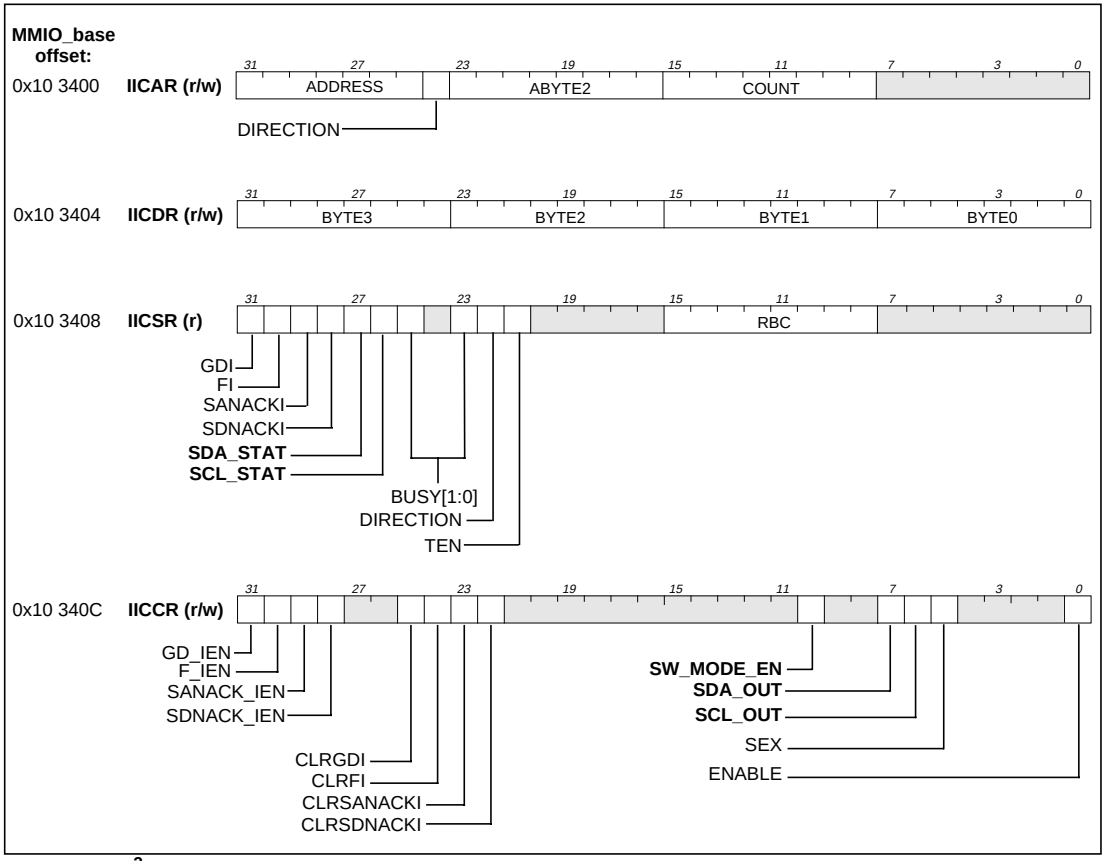


Figure 15-2. I²C Registers

- DIRECTION = 1 –> I²C read

The ABYTE2 field is used for 10-bit I²C addressing only and is unused during 7-bit I²C transfers. The COUNT field must contain the desired bytecount for the current transfer. The COUNT field will decrement by one for each data byte transferred across I²C. The remaining bytecount for the current transfer can be read from the COUNT field at any time. Note that the DSPCPU must refrain from re-writing the IICAR register until the current transfer completes to avoid corrupting the bytecount or address fields.

Note: *For writes, the byte count decrements before the byte is actually transferred over the I²C bus. However, the last byte is saved in an internal register and the DSPCPU can write a new word when COUNT = 0.*

15.3.2 IICDR Register

The IICDR register contains the actual data transferred during I²C operation. For a master transmit operation, data transfer will be initiated when data is written to this register. Transmission will begin with the transfer of the address byte(s) in the IICAR register followed by the data bytes that were written to the IICDR register. The I²C interface will interrupt for more transmit data to be written to the IICDR until the transfer bytecount COUNT in the IICAR register is reached.

In master receive operation, one or more data bytes received are placed in the IICDR register by the I²C interface. Data bytes received are loaded into the IICDR register in the following order:

- If register IICCR bitfield SEX = 0 (RECOMMENDED) then receive data is loaded into IICDR register starting with byte # 3.
- If register IICCR bitfield SEX = 1 (NOT RECMMENDED FOR COMPATIBILITY WITH FUTURE DEVICES) then receive data is loaded into IICDR register starting with byte # 0.

The number of bytes the DSPCPU requests for a transfer is written into the COUNT bitfield of the IICAR register. The I²C interface requests bytes by acknowledging each byte received without a STOP condition on the bus signal lines. The transfer completes when the I²C interface receives the number of bytes indicated by the COUNT bitfield of the IICAR register.

15.3.3 IICSR Register

The I^2C status register contains status information regarding the transfer in progress and the nature of interrupts associated with I^2C operation.

Table 15-3. IICSR Register

Bits	Field Name	Definition
31	GDI	Good Data Interrupt. This is the normal transfer complete interrupt flag. This interrupt may be asserted without the IICSR.FI interrupt bit at the end of an I^2C transfer or after master abort of an I^2C transfer.
30	FI	Full Interrupt. This interrupt indicates the condition of the IICDR register dependent upon whether the I^2C interface is in receive or transmit mode.
29	SANACKI	Slave Address No Acknowledge Interrupt.
28	SDNACKI	Slave Data No Acknowledge Interrupt.
27	SDA_STAT	This bit is used to examine the state of the external I^2C SDA data pin. Bit polarity is: 1 = SDA pad is low 0 = SDA pad floated high
26	SCL_STAT	This bit is used to examine the state of the external I^2C SCL clock pin. Bit polarity is: 1 = SCL pad is low 0 = SCL pad floated high
25	BUSY[1]	The BUSY[1] and BUSY[0] bits indicate the microactivity of the I^2C bus.
23	BUSY[0]	The BUSY[1] and BUSY[0] bits indicate the microactivity of the I^2C bus.
22	DIRECTION	Direction of current data transfer.
21	TEN	I^2C Addressing mode. 7-bit or 10-bit addressing
15:8	RBC	Remaining Byte Count.
7:0	Reserved	Read as '0'

The IICSR register is read only and is intended as the primary source of status regarding current I^2C operation. The IICSR register must be used in conjunction with the IICCR register. The interrupt sources of the IICSR register are individually enabled by writing to the appropriate enable bit in the IICCR register. The bitfield definitions of the IICSR register are presented in Table 15-3. The IICSR provides four sources of interrupts.

- **GDI** interrupt — The GDI bit together with the FI bits provide status about I^2C transfer completion. The interpretation of GDI/FI bit combinations are different depending on whether the I^2C interface is in master transmit or master receive mode. Refer to Table 15-4 and Table 15-6 for GDI/FI interpretation
- **FI** interrupt — See GDI bit definition and GDI/FI transmit and receive definitions in Table 15-4 and Table 15-6.
- **SANACKI** interrupt — This interrupt flag bit indicates that a slave address was transmitted but no slave on the I^2C bus acknowledges the address to claim the transaction. This is an error condition. Once the I^2C interface has set this interrupt flag, the interface is idle. The DSPCPU should clear this interrupt flag by writing a '1' to IICCR.CLRSANACKI before reattempting this transfer or starting another I^2C transfer.
- **SDNACKI** interrupt — This interrupt flag bit indicates that an addressed slave receiver device has refused to acknowledge the current byte of data for an ongoing transfer. This is an error condition. Once the I^2C interface has set this interrupt flag, the interface is idle. The DSPCPU should clear this interrupt flag by writing a '1' to IICCR.CLRSDNACKI before retrying this transfer or starting another.
- **BUSY[1:0]** — These status bits indicate the microactivity of the I^2C interface. The condition codes and their meanings are presented in Table 15-5.

Table 15-4. Master Transmit Mode GDI/FI Status

GDI	FI	Description
0	0	Message is not complete. The IICDR is not empty. No interrupt.
0	1	Message is not complete. The IICDR is empty and the requested transmit byte count is not equal to 0. The DSPCPU must write additional bytes of the current transfer to the IICDR register.
1	X	Message is complete. The IICDR is empty. The byte transmit count = 0.

Table 15-5. BUSY[1:0] Condition Codes

BUSY[1:0]	Meaning
00	I^2C Interface is idle.

Table 15-6. Master Receive GDI/FI Conditions

GDI	FI	Description
0	0	Message is not complete. IICDR is not full. No interrupt.
0	1	IICDR contains received data and needs to be read serviced. More data bytes are expected since the receive byte count is not equal to 0.
1	X	The transfer has been completed and the receive byte count is equal to 0. 0 to 4 valid bytes are in the IICDR register awaiting read servicing by the DSPCPU.

The DIRECTION status bit indicates if the I^2C interface is in transmit or receive mode.

- if DIRECTION = 0 then I^2C is a transmitter.
- if DIRECTION = 1 then I^2C is a receiver.

The TEN bit of the IICSR register indicates if the I^2C interface is in 7-bit address mode or 10-bit address mode.

- if TEN = 0 then I^2C is in 7-bit address mode
- if TEN = 1 then I^2C is in 10-bit address mode.

The RBC bitfield indicates the remaining bytecount for an I^2C transfer in progress. The IICSR.RBC bitfield serves as a read-only “shadow register” for the IICAR.COUNT bitfield. During I^2C transfer, the RBC bitfield will reflect the remaining bytecount. To avoid corrupting an I^2C transfer, the DSPCPU must refrain from writing to the IICAR.COUNT bitfield until a message is complete. Completion is indicated by the RBC bitfield decrementing to zero.

15.3.4 IICCR Register

The I^2C control register contains control information required for enabling I^2C transfers. This register is used to enable and clear interrupt sources which normally occur during I^2C operation. The four interrupt sources described in the section on the IICSR register are enabled and cleared through the IICCR register. The enable bitfields are:

Table 15-7. IICCR Register

Bits	Field Name	Definition
31	GD_IEN	Enable for normal transfer complete interrupt
30	F_IEN	Enable for IICDR data service request interrupt.
29	SANACK_IEN	Enable for slave address not acknowledged interrupt.
28	SDNACK_IEN	Enable for slave data not acknowledged interrupt. An addressed slave receiver has refused to accept the last byte transmitted to it.
27:26	Reserved1	Always write ‘0’s to these bits. (See Note1)
25	CLRGDI	Clear bit for the GDI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the GDI interrupt.
24	CLRFI	Clear bit for the FI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the FI interrupt.
23	CLRSANACKI	Clear bit for the SANACKI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the SANACKI interrupt.
22	CLRSDNACKI	Clear bit for the SDNACKI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the SDNACKI interrupt.
21:6	Reserved2	Always write ‘0’s to these bits. (See Note1)
10	SW_MODE_EN	0 (power-on/reset default) - Normal I2C hardware operating mode.. 1 - Enable software operating mode. The I2C pins are entirely controlled by user writes to the ‘sda_out’ and ‘scl_out’ register bits.

Table 15-7. IICCR Register (Continued)

Bits	Field Name	Definition
7	SDA_OUT	Enabled by sw_mode_en. This bit is used by sw to manually control the external i2c SDA data pin. Bit polarity is: 1 = SDA pad pulled low 0 = SDA pad left open drain
6	SCL_OUT	Enabled by sw_mode_en. This bit is used by sw to manually control the external i2c SCL clock pin. Bit polarity is: 1 = SCL pad pulled low 0 = SCL pad left open drain
5	SEX	Byte order memory format control. This bit controls ordering of bytes within the IICDR register. • 0 - byte 3 of IICDR is transmitted/received first • 1 - byte 0 first For compatibility with future Trimedia devices, it is recommended that this bit always be set to 0.
4:2	Reserved3	Always write ‘0’s to these bits. (See Note1)
1	Reserved4	Always write ‘0’s to these bits. (See Note1)
0	ENABLE	I2C serial interface enable

- **GD_IEN** — Enable for normal transfer complete interrupt.
- **F_IEN** — Enable for IICDR data service request interrupt.
- **SANACK_IEN** — Enable for slave address not acknowledged interrupt. This is an error interrupt.
- **SDNACK_IEN** — Enable for slave data not acknowledged interrupt. An addressed slave receiver has refused to accept the last byte transmitted to it. This is handled as an error interrupt.

In addition to the interrupt enable bits, the IICCR contains interrupt clear bits associated with each of the interrupt sources in the IICSR register. These IICCR interrupt clear bits are defined as:

- **CLRGDI** — Clear bit for the GDI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the GDI interrupt.
- **CLRFI** — Clear bit for the FI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the FI interrupt.
- **CLRSANACKI** — Clear bit for the SANACKI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the SANACKI interrupt.
- **CLRSDNACKI** — Clear bit for the SDNACKI interrupt in the IICSR register. Writing a ‘1’ to this bit clears the SDNACKI interrupt.

The remaining bitfields of the IICCR register are:

- **SEX** — Byte order memory format control. This bit controls ordering of bytes within the IICDR register. If SEX = 0, then Byte3 of IICDR is the first transmitted/ received across I^2C. If SEX = 1, then Byte0 of IICDR is the first transmitted across I^2C. Future Trimedia devices will no longer support the SEX bit in I2C. Instead, they will always transmit byte 3 of IICDR first, corresponding to SEX=0 in TM1000. Hence, for future compatibility, it is strongly recommended that all TM1000 software uses SEX=0 only and perform any required byte swapping in software.

- **ENABLE** — Master enable for I^2C serial interface. ENABLE must be set equal to '1' to transfer any bits from the I^2C interface block. Writing the ENABLE bit to '0' effectively resets the entire I^2C interface, including all status and interrupt flag bits. A transfer in progress is aborted and the byte currently transferred is lost.

  Note 1: *For writes, Reserved1, 2, 3 and 4 bitfields MUST always be written with '0's.*

  Note 2: *During writes, the ENABLE bit must remain active for one byte time (approx. 90 usec with a 100 kHz I^2C clock) after GDI ("end of message" bit) is set in the IICSR register, in order for the write to complete normally.*

15.4 I^2C SOFTWARE OPERATION MODE

I^2C software operation mode is intended for use by software I^2C or similar algorithm implementations. In this case, the SCL and SDA pins are fully controlled and observed by software, and the hardware I^2C interface is disconnected from the SCL and SDA pins. This operation mode is available in the production TM1000 version, and is not present in TM1000 early samples.

Refer to Figure 15-3 for a clarification of the principles involved. Software mode is by default disabled after boot. Software mode is enabled by writing a '1' to IICCR.SW_MODE_EN. At that point, the SCL and SDA pins can be controlled by the IICCR SDA_OUT and SCL_OUT bits. Writing a '1' to either bit causes the corresponding pin to become active, i.e. be pulled low. The SDA and SCL lines are open-collector outputs, and can hence also be pulled low by external devices. The actual pin state can be observed by software by examining IICSR SDA_STAT and SCL_STAT bits. A 1 in these MMIO bits indicates that the corresponding pin is currently pulled low.

By appropriate software, possibly using a timer interrupt, full I^2C functionality can be implemented using this mechanism.

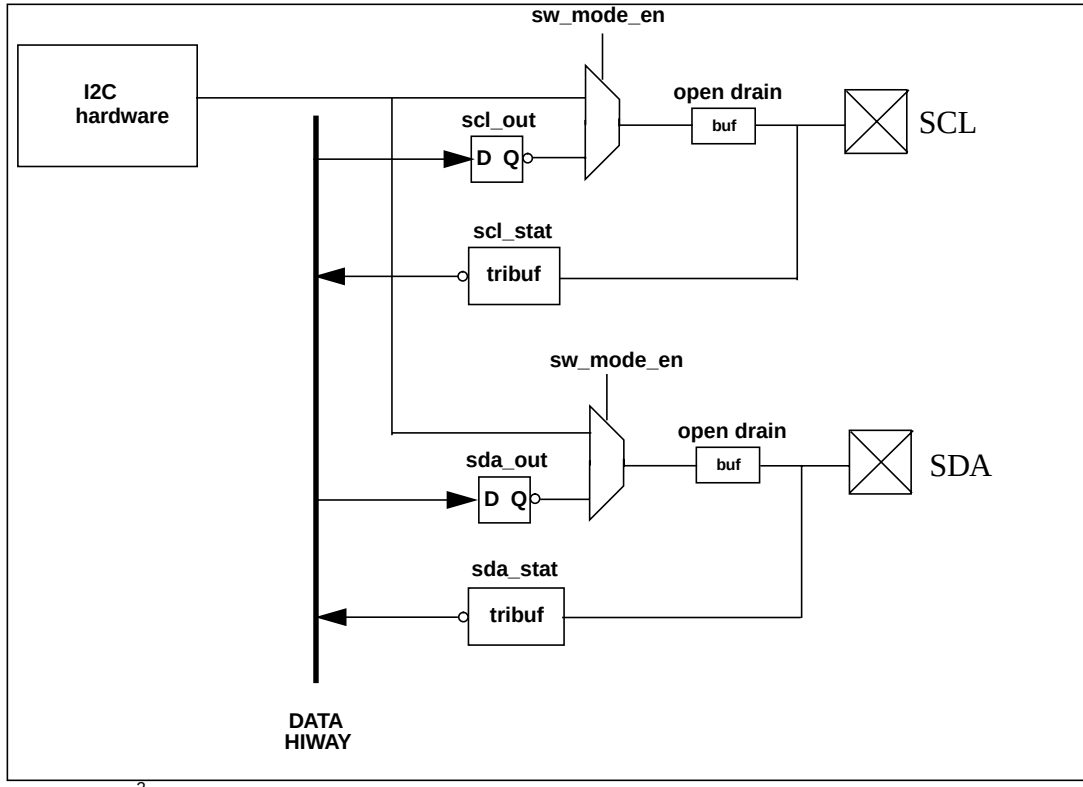


Figure 15-3. I^2C software mode only logic

15.5 I²C HARDWARE OPERATION MODE

Hardware operation of I²C is the default mode after boot. The TM1000 I²C hardware interface operates in one of two modes:

1. Master-Transmitter
2. Master-Receiver

As a master, the I²C logic will generate all the serial clock pulses and the START and STOP bus conditions. The START and STOP bus conditions are shown in Figure 15-4. A transfer is ended with a STOP condition or a repeated START condition. Since a repeated START condition is also the beginning of the next serial transfer, the I²C bus will not be released.

Note: *The I²C interface on TM1000 will operate as a master ONLY!*

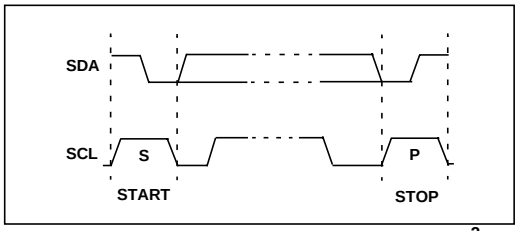


Figure 15-4. START and STOP Conditions on I²C

The number of bytes transferred between the START and STOP conditions from transmitter to receiver is not limited. Each data byte of 8 bits is followed by one acknowledge bit. The transmitter releases the SDA line which will pull-up to a HIGH level during the acknowledge bit time. The receiver acknowledges by pulling the data line LOW during this acknowledge period. The master must always generate an acknowledge related clock pulse.

Two types of data transfers are possible on the I²C bus:

- Data transfer from a master transmitter to a slave receiver. The first byte transmitted by the master is the slave address. For 10-bit address mode, a second sub-address byte follows, otherwise a number of data bytes follow. The slave receiver returns an acknowledge bit after each byte.
- Data transfer from slave transmitter to master receiver. The first byte (the slave address), is transmitted by the master. For 10-bit addressing, the master transmits a second sub-address byte. The slave returns an acknowledge bit after each address byte received. Next follows the data bytes transmitted by the slave to the master. The master generates an acknowledge bit after each byte received, except the last byte. At the end of the last byte, a “not-acknowledge” condition is generated. The slave transmitter then must release the bus so that the master may generate a STOP condition.

The type of transaction is determined by the Lsbit of the first address byte. Data transfer from a master transmitter to a slave receiver is called a WRITE. It is signified by a ‘0’ in the Lsbit of the first address byte. Data transfer from a slave transmitter to a master receiver is called a READ. It is signified by a ‘1’ in the LSBit of the first address byte.

Example steps for successful programming of the I²C interface on TM1000 are outlined as follows for both reads and writes. Enable the I²C interface prior to attempting any accesses to external I²C devices.

To enable the interface:

- Set bit BIU_CTL.CR (0x103008) = 1
- Set bit IICCR.ENABLE (0x10340c) = 1
- Set bit IICCR.SEX (0x10340c) = desired endianness.

For 7-bit write addressing mode:

i) On entry, clear any possible I²C interrupt sources by writing IICCR bits [25:22] = ‘1111’. (Note that programmers must mask and enable high level interrupt sources through the VIC facility in the DSPCPU. See the appropriate TM1000 databook chapter).

ii) Enable desired I²C interrupt sources by setting IICCR[31:28] bits appropriately.

iii) Simultaneously load IICAR[31:25] with 7-bit slave address, IICAR.DIRECTION = 0 and IICAR[15:8] with the appropriate bytecount for the transfer.

iv) Load IICDR[31:0] with data for the write. Note that writing this register triggers the transfer across the I²C bus.

v) Detect I²C resulting condition code in IICSR[31:28] and respond - OR - Detect I²C high level interrupt and respond. (Note that this last step is dependent upon system software requirements).

For 7-bit read addressing mode:

i) On entry, clear any possible I²C interrupt sources by writing IICCR bits [25:22] = ‘1111’. (Note that programmers must mask and enable high level interrupt sources through the VIC facility in the DSPCPU. See the appropriate databook chapter).

ii) Enable desired I²C interrupt sources by setting IICCR[31:28] bits appropriately.

iii) Simultaneously load IICAR[31:25] with 7-bit slave address, IICAR.DIRECTION = 1 and IICAR[15:8] with the appropriate bytecount for the transfer. Note that writing this register triggers the read across the I²C bus.

vi) Detect I²C resulting condition in IICSR[31:28] and respond - OR - Detect I²C interrupt and respond. (Note that this last step is dependent upon system software requirements.)

15.6 I²C CLOCK RATE GENERATION

The I²C hardware block diagram is shown in Figure 15-5 below. In hardware operating mode, the IIC_SCL external clock is derived by division from the BOOT_CLK pin on TM1000. The BOOT_CLK pin is normally connected to TRI_CLKIN. The IIC_SCL clock divider value is determined at boot time, and cannot be changed thereafter. The value chosen depends on the first byte read from the EEPROM, as described in Section 12.2.1, “Boot Proce-

dure Common to Both Autonomous and Host-Assisted Bootstrap."

Table 15-8. I2C speed as a function of EEPROM byte 0

BOOT_CLK bits	EEPROM speed bit	divider value	actual I2C speed
00 (100 MHz)	0 (100 kHz)	1040	97 kHz
00	1 (400 kHz)	272	368 kHz
01 (75 MHz)	0 (100 kHz)	784	96 kHz
01	1 (400 kHz)	208	360 kHz
10 (50 MHz)	0 (100 kHz)	528	95 kHz
10	1 (400 kHz)	144	347 kHz
11 (33 MHz)	0 (100 kHz)	352	94 kHz
11	1 (400 kHz)	112	295 kHz

The TM1000 I^2C block is able to "stretch" the SCL clock in response to slaves that need to slow down byte transfer. This mechanism of slowing SCL in response to a slave is called "clock stretching." This clock stretching is accomplished by the slave by holding the SCL line "low" after completion of a byte transfer and acknowledge sequence. Clock stretching is not by default enabled and must be explicitly enabled by setting ARB_BW_CTL[24] = 1.

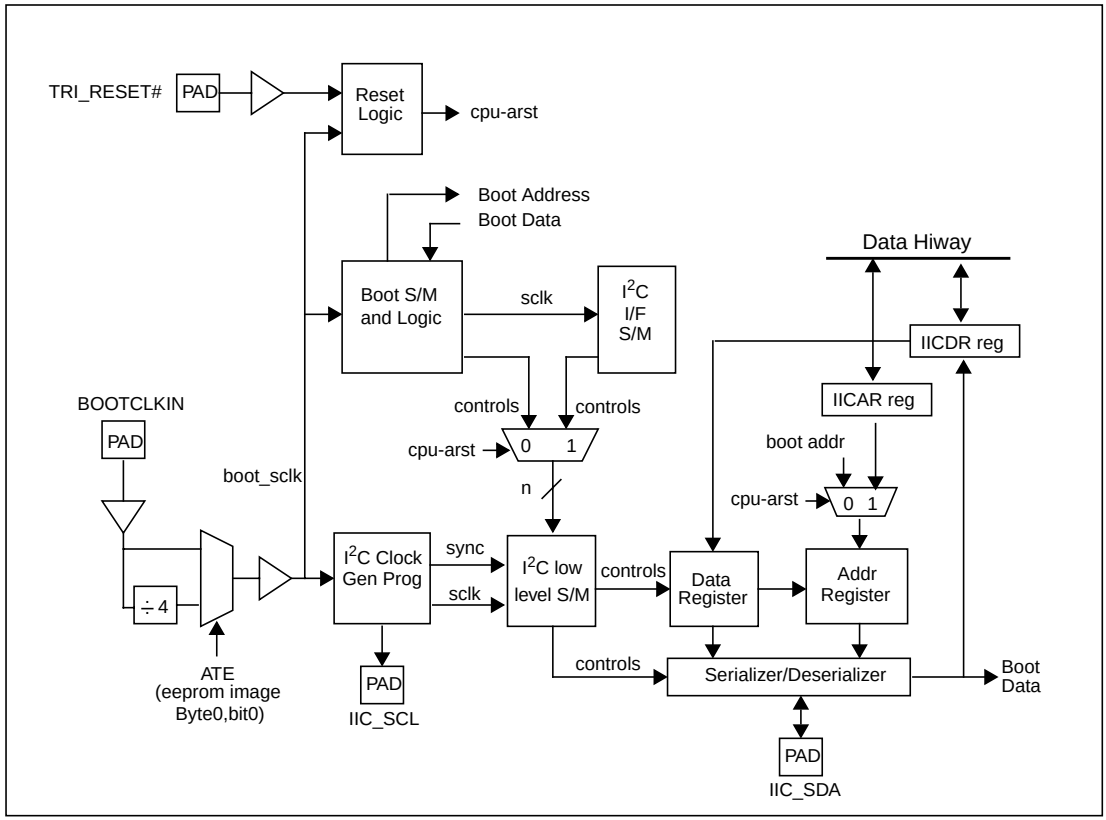


Figure 15-5. I^2C block diagram

16.1 V.34 SYNC SERIAL INTERFACE OVERVIEW

The TM1000 V.34 synchronous serial interface (SSI) unit connects to an off-chip modem analog front end (MAFE) subsystem, Network Terminator, ADC/DAC or Codec through a flexible bit-serial connection. The hardware performs full-duplex serialization/deserialization of a bit steam from any of these devices. Any such front end device to be connected must support Tx, Rx and initialization via a synchronous serial interface.

Since the communication algorithm is implemented in software by the TM1000 DSPCPU and the analog interface is off chip, a wide variety of modem, network and/or FAX protocols may be supported.

V.34 synchronous serial Interface hardware includes:

- A 16-bit receive shift register (RxSR), synchronized by an external receive frame sync pulse (V34_RxFSX) and clocked by an external clock (V34_RxCLK).
- A 32-bit MMIO receive data register (RxDR) to provide data access to internal hiway.
- A 32-depth of 16-bit receive buffer (RxFIFO) to buffer between the receive shift register (RxSR) and MMIO receive data register (RxDR).
- A 16-bit transmit shift register (TxSR), synchronized by an external or internal transmit frame sync pulse and clocked by an external clock (either V34_TxCLK or V34_RxCLK).
- A 32-bit MMIO transmit data register (TxDR) to transmit data from internal hiway.
- A 32-depth of 16-bit transmit buffer (TxFIFO) to buffer between the MMIO transmit data register (TxDR) and transmit shift register (TxSR).
- Transmit frame sync pulse generation logic.
- Control and status logic.
- Interrupt generation logic.

The V.34 SSI is not a hiway bus master. All I/O is completed through MMIO cycles. FIFO service is initiated in response to interrupts generated by the V.34SSI.

16.2 INTERFACE

16.2.1 External

The external interface consists of the 6 pins described in Table 16-1.

Table 16-1. V.34 Synchronous Serial Interface External Pins

Name	Type	Description
V34_CLK	IN-5	Serial I/O interface clock. Provided by the receive channel of an external communication device.
V34_RxFSX	IN-5	Frame synchronization reference. Provided by the receive channel of an external communication device.
V34_RxDATA	IN-5	Receive serial data. Provided by the receive channel of an external communication device.
V34_TxDATA	OUT	Transmit serial data to the transmit channel of an external communication device.
V34_IO1	I/O-5	Serial I/O interface clock. As a transmit clock, it is driven by an external communication device or clock generation circuit. The pin may also serve as a general purpose I/O.
V34_IO2	I/O-5	Frame synchronization reference to the transmit channel of an external communication device. The pin may also serve as a general purpose I/O.

16.2.2 Internal

The V.34SSI contains a standard hiway interface for MMIO registers. The interrupt line for the V.34 synchronous serial interface is interrupt 15d of the Vectored Interrupt Controller (VIC). FIFOs in the V.34SSI have been sized to deal with internal interrupt latencies in excess of 1 ms for sampling rates of 1-16 bit sample at a 9.6 KHz sample rate.

16.3 REGISTERS

Address Map

Table 16-2. MMIO Register Address Map

Register	Address	Mode	Initial Value
V34CR	10 2C00	R/W	0
V34CSR	10 2C04	R/W	0
TxDR	10 2C10	Write	0
RxDR	10 2C20	Read	0
V34UPD	10 2C24	Write	0

TxDR

The TxDR is a 32-bit MMIO transmit data register that accepts two outbound 16-bit words from the hiway.

TxFIFO

The TxFIFO is a 32-depth of 16-bit transmit buffer that buffers thirty-two outbound 16-bit words from the TxDR to the TxSR.

TxSR

The TxSR is a 16-bit transmit shift register. TxSR can be configured to shift out MSB or LSB first. The clock source is external with transfer on either the rising or falling edge under program control. The output pin V34_TxDATA mirrors the state of TxSR MSB or LSB, also under program control.

RxFIFO

The RxFIFO is a 32-depth of 16-bit receive buffer that buffers thirty-two inbound 16-bit words from the RxSR to the RxDR.

RxSR

The RxSR is a 16-bit receive shift register. RxSR can be configured to shift in from MSB or LSB. The clock source is external with transfer on either the rising or falling edge under program control. The input pin V34_RxDATA provides serial shift in data to the RxSR in either the MSB or LSB position, also under program control.

V34UPD

The V34 UPD is a 1-bit MMIO register that is used to signal the SSI receiver state machine that a word has been successfully read from the RxDR. The receiver state machine uses that information to signal the need to update internal status registers. Writing a ‘1’ to the LSB of this register initiates updating. Writing a zero has no effect. The register cannot be read, its effect may be observed in the WAR field of the V34CSR.

16.4 SSI PROGRAMMING MODEL

The SSI can be viewed as one 32-bit control register, one 32-bit control/status register, one 32-bit transmit data register, and one 32-bit receive data register. The control and control/status registers are illustrated in Figure 16-1 and Figure 16-2. The following paragraphs give detailed descriptions of the status and operational controls implemented by each of the bits in the SSI control and control/status registers.

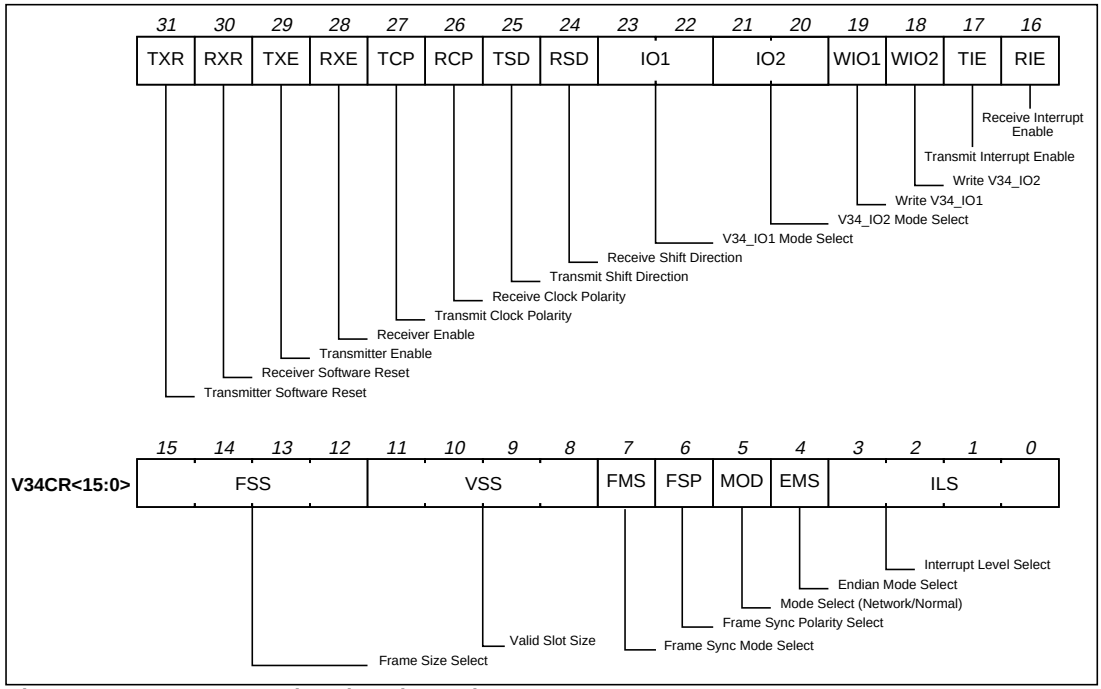


Figure 16-1. V.34 SSI Control Register (V34CR)

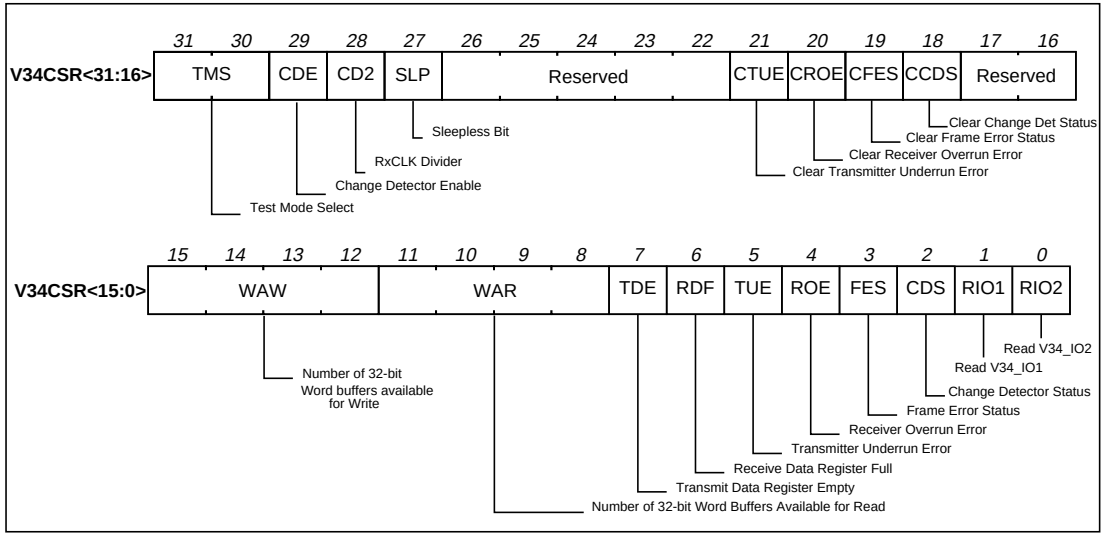


Figure 16-2. V.34 SSI Control/Status Register (V34CSR)

16.4.1 SSI Control Register (V34CR)

V34CR is a 32-bit read/write control register used to direct the operation of the SSI.

V34CR Transmitter Software Reset (TXR) Bit 31

Setting TXR performs the same functions as a hardware reset. Re sets all transmitter functions. A transmission in progress is interrupted and the data remaining in the TxSR is lost. The TxFIFO pointers are reset and the data contained will not be transmitted, but the data in the TxDR and/or TxFIFO is not explicitly deleted. The transmitter status and interrupts are all cleared. This is an action bit. This bit always reads '0'. Writing a '1' initiates a single reset event.

V34CR Receiver Software Reset (RXR) Bit 30

Setting RXR performs the same functions as a hardware reset. Resets all receiver functions. A reception in progress is interrupted and the data collected in the RxSR is lost. The RxFIFO pointers are reset and the V.34 SSI will not generate an interrupt to DSPCPU to retrieve data in the RxDR and/or RxFIFO. The data in the RxDR and/or RxFIFO is not explicitly deleted. The receiver status and interrupts are all cleared.This is an action bit.This bit always reads '0'. Writing a '1' initiates a single reset event.

V34CR Transmitter Enable (TXE) Bit 29

TXE enables the operation of the transmit shift register state machine. When TXE is set and a frame sync is detected, the transmit state machine of the SSI is begins transmission of the frame. When TXE is cleared, the transmitter will be disabled after completing transmission of data currently in the TxSR. The serial output (V34_TxDATA) is three-stated, and any data present in TxDR and/or TxFIFO will not be transmitted (i.e., data can be written to TxDR with TXE cleared; TDE can be cleared, but data will not be transferred to the TxSR).

Status fields updated by the Transmit state machine are not updated or reset when an active transmitter is disabled.

V34CR Receiver Enable (RXE) Bit 28

When RXE is set, the receive state machine of the SSI is enabled. When this bit is cleared, the receiver will be disabled by inhibiting data transfer into RxDR and/or RxFIFO. If data is being received while this bit is cleared, the remainder of that 16-bit word will be shifted in and transferred to the SSI RxFIFO and/or RxDR.

Status fields updated by the Receive state machine are not updated or reset when an active receiver is disabled.

V34CR Transmit Clock Polarity (TCP) Bit 27

The TCP bit value should only be changed when the transmitter is disabled. TCP controls which edge of V34_TxCLK is the sampling edge for an external communication device. This bit causes the data to be sampled at rising edge when TCP equals one or falling edge when TCP equals zero.

V34CR Receive Clock Polarity (RCP) Bit 26

RCP controls which edge of V34_RxCLK samples data. This bit causes the data to be sampled at rising edge when RCP equals one or falling edge when RCP equals zero.

V34CR Transmit Shift Direction (TSD) Bit 25

TSD controls the shift direction of transmit shift register (TxSR). This bit causes the TxSR to shift data out MSB first when TSD equals zero or LSB first when TSD equals one.

V34CR Receive Shift Direction (RSD) Bit 24

The RSD bit value should only be changed when the receiver is disabled. RSD controls the shift direction of receive shift register (RxSR). Receive data is shifted in LSB first when RSD equals zero or MSB first when RSD equals one.

V34CR V34_IO1 Mode Select (IO1) Bits 23-22

The IO1 field value should only be changed when the transmitter and receiver are disabled. The IO1[1:0] bits are used to select the function of V34_TxCLK/V34_IO1 pin. The function may be selected according to the following table.

Table 16-3. V34_IO1 Mode Select

Bit	Mode
00	**General Purpose Output :** Configures the V34_IO1 pin as a general purpose output. The pin follows the state of the WIO1 field of the V34CR.
01	**General Purpose Input :** Change detector may be used. Value can be read in from RIO1 field of the V34CSR.
10	**Enable External TxCLK :** Allows for use of an externally generated TxCLK. The clock is provided via the V34_TxCLK pin. All general purpose I/O functions are unavailable.
11	**Disable :** Pin is not used. Output buffer is tristated and the input is ignored. (RESET default)

V34CR V34_IO2 Mode Select (IO2) Bits 21-20

The IO2 field value should only be changed when the transmitter and receiver are disabled. The IO2[1:0] bits are used to select the function of V34_TxFSX/V34_IO2 pin. The function may be selected according to Table 16-4.

Table 16-4. V34_IO2 Mode Select

Bit	Mode
00	**General Purpose Output :** Configures the V34_IO2 pin as a general purpose output. The pin follows the state of the WIO2 field of the V34CR.

Table 16-4. V34_IO2 Mode Select

Bit	Mode
01	**General Purpose Input :** Value can be read in from RIO2 field of the V34CSR.
10	**Frame Signal TxFSX (Output):** Output the frame signal generated by the internal frame signal generation logic.
11	**Frame Signal TxFSX (Input):** Allows for use of an externally generated TxFSX. The frame sync signal is provided via V34_TxFSX pin. All general purpose I/O functions are unavailable. (RESET default)

V34CR Write V34_IO1 (WIO1) Bit 19

Value written here appears on the V34_TxCLK/V34_IO1 pin when this pin is configured to be a general purpose output.

V34CR Write V34_IO2 (WIO2) Bit 18

Value written here appears on the V34_TxFSX/V34_IO2 pin when this pin is configured to be a general purpose output.

V34CR Transmit Interrupt Enable (TIE) Bit 17

The DSPCPU will be interrupted when TIE and the TDE flag in the SSI status register are both set. When TIE is cleared, this interrupt is disabled. However, the TDE bit will always indicate the transmit data register empty condition even when the transmitter interrupt is disabled.

V34CR Receive Interrupt Enable (RIE) Bit 16

When RIE is set, the DSPCPU will be interrupted when RDF in the SSI status register is set. When RIE is cleared, this interrupt is disabled. However, the RDF bit still indicates the receive data register full condition even when the receiver interrupt is disabled.

V34CR Frame Size Select (FSS) Bits 15-12

The FSS[3:0] bits control the divide ratio for the programmable frame rate divider used to generate the frame sync pulses. The valid setup value ranges from 1 to 16 slot(s). The value 16 is accomplished by storing a 0 in this field.

V34CR Valid Slot Size (VSS) Bits 11-8

The VSS[3:0] bits control the valid slot size starting from slot 1 for different modem analog front end devices. The valid setup value ranges from 1 to 16 slot(s). The value 16 is accomplished by storing a 0 in this field.

V34CR Frame Sync Mode Select (FMS) Bits 7

The FMS bit value should only be changed when the transmitter and receiver are disabled. FMS selects the type of frame sync to be recognized by both Rx and Tx. When FMS equals one, frame sync is word-length bit clock. When this bit equals zero, frame syncis one-bit clock.

V34CR Frame Sync Polarity(FSP) Bits 6

The FSP bit value should only be changed when the transmitter and receiver are disabled. FSP controls which edge of frame sync is the active edge for both Rx and Tx. This bit causes frame signal to be active at rising edge when FSP equals zero, or falling edge when FSP equals one.

V34CR Mode Select (MOD) Bit 5

The MOD bit value should only be changed when the transmitter and receiver are disabled. MOD selects the operational mode of the SSI for ISDN functionality. When MOD is set, the SSI is configured as a U-interface for ISDN NT. Otherwise, set to ‘0’.

V34CR Endian Mode Select (EMS) Bit 4

EMS selects the big- or little-endian mode operation. When EMS is cleared, the big-endian format is selected; when EMS is set, the little-endian format is selected. Explicitly, when EMS is set, the first data byte received in a frame, it will be transferred in bit 7-0 of the RxDR, the fourth byte will be transferred in bits 31-24 of the RxDR. EMS = ‘0’ reverses the order of the bytes in RxDR.

V34CR Interrupt Level Select (ILS) Bit 3-0

Set the point where an interrupt is generated for normal data buffer servicing. The number is ranging from 0 to 15 of 32-bit word(s). This field controls interrupt level of both transmit and receive functions.

16.4.2 SSI Control/Status Register (V34CSR)

V34CSR Test Mode Select (TMS) Bit 31-30

The TMS field value should only be changed when the transmitter and receiver are disabled.

Table 16-5. Test Mode Select

Bit	Mode
0X	**Normal Operation.**
10	**Remote Loopback Test:** Direct connection of receiver serial data to transmitter serial data. Transmitter is clocked with V34_RxCLK. No data loaded to the RxDR register or RxFIFO buffer and no interrupt of the DSPCPU is generated. Useful to allow remote device to test the communication medium and our Rx and Tx front ends.
11	**Local Loopback Test :** Feedback is after TxDR and RxDR register and serializer/deserializer. Allows DSPCPU to test the bulk of the Rx and Tx circuits.

V34CSR Change Detector Enable (CDE) Bit 29

CDE enables the change detector function on the V34_IO1 pin. When CDE is set, the DSPCPU will be interrupted when CDS in the SSI status register is set. When CDE is cleared, this interrupt is disabled. However, the CDS bit will always indicate the change detector condition.

When the change detector is enabled, the V34_CLK samples V34_IO1. The CDS bit will be set for either a '0' –> '1' or a '1' –> '0' change between the current value and the stored value.

V34CSR RxCLK Divider (CD2) Bit 28

When CD2 equals one, the internal RxCLK is divided by two. In the divide by 2 mode, the clock edge that samples the Frame Sync Pulse asserted will resync the RxCLK divider to be a data capture edge. Data samples will occur every other clock thereafter until the end of the valid slots in the frame.

V34CSR Sleepless bit (SLP) Bit 27

When set, this bit allows the V.34 SSI to ignore the global power down signal. If cleared, assertion of the global power down signal will cause the SSI transmitter will finish transmission of the current 16-bit word, then enter a state similar to transmitter disabled, (V34CR.TXE = '0').

In the receiver, a 16-bit word currently being transmitted to RxSR will complete reception and be transferred to the RxFIFO. The receiver will then enter a state similar to receiver disabled, (V34CR.RXE = '0').

V34CSR Reserved bits Bit 26-22.

Reserved for future use.

V34CSR Clear Transmitter Underrun Error (CTUE) Bit 21.

A control bit written by the DSPCPU to indicate that the transmitter underrun error flag should be cleared. This is an action bit. Writing a '1' clears V34CSR.TUE. The bit always reads '0'.

V34CSR Clear Receiver Overrun Error (CROE) Bit 20.

A control bit written by the DSPCPU to indicate that the receiver overrun error flag should be cleared. This is an action bit. Writing a '1' clears V34CSR.TOE. The bit always reads '0'.

V34CSR Clear Framing Error Status(CFES) Bit 19.

A control bit written by the DSPCPU to indicate that the receiver's framing error flag should be cleared. This is an action bit. Writing a '1' clears V34CSR.FES. The bit always reads '0'.

V34CSR Clear Change Detector Status(CCDS) Bit 18.

A control bit written by the DSPCPU to indicate that the change detector status on V34_IO1 flag should be cleared. This is an action bit. Writing a '1' clears V34CSR.CDS. The bit always reads '0'.

V34CSR Number of 32-Bit Word Buffers Available for Write (WAW) Bit 15-12

The WAW[3:0] bits provide the number of 32-bit words available for write in the transmit buffer (TxFIFO).

V34CSR Number of 32-Bit Word Buffers Available for Read (WAR) Bit 11-8

The WAR[3:0] bits provide the number of 32-bit word available for read in the receive buffer (RxFIFO).

V34CSR Transmit Data Register Empty (TDE) Bit 7

In normal operation, this bit will be set when the number of empty words in the TxFIFO is greater than V34CR.ILS. If V34Cr.TIE is set, the SSI will generate an interrupt. When set, it indicates that the TxDR/TxFIFO registers require DSPCPU service for refilling after normal transmission. As the DSPCPU refills the TxFIFO during the interrupt service routine, this bit will be cleared by the V.34 SSI when the number of empty slots drops below the Interrupt Level Select value, V34CR.ILS.

V34CSR Receive Data Register Full (RDF) Bit 6

In normal operation, this bit will be set when the number of words in the RxFIFO is greater than V34CR.ILS. If V34Cr.RIE is set, the SSI will generate an interrupt. When set, this bit indicates that normal received data resides in RxDR register and RxFIFO buffer for reading. DSPCPU must service the RxFIFO before a receiver overrun occurs.

The DSPCPU controls clearing of this bit by explicitly writing to the V34CSR.URS and URC fields after retrieving data from the RxFIFO via the RxDR.

V34CSR Transmitter Underrun Error (TUE) Bit 5

No current data was available from the TxFIFO when a load of the TxSR was scheduled. The transmitted message may have been corrupted.

V34CSR Receive Overrun Error (ROE) Bit 4

Receive data has been received with no RxFIFO slot to store it. These bits have been lost and the message stream is incomplete.

V34CSR Frame Error (FES) Bit 3

A frame sync pulse has been detected where not expected or did not occur as expected. Received data may be invalid.

V34CSR Change Detector Status (CDS) Bit 2

The input change detector on V34_IO1 pin has detected a change in state.

V34CSR Read V34_IO1 (RIO1) Bit 1

RIO1 reflects the value on the V34_IO1 pin.

V34CSR Read V34_IO2 (RIO2) Bit 0

RIO2 reflects the value on the V34_IO2 pin.

16.5 OPERATION DETAILS

16.5.1 Transmit

16.5.1.1 Transmitter Logic Model

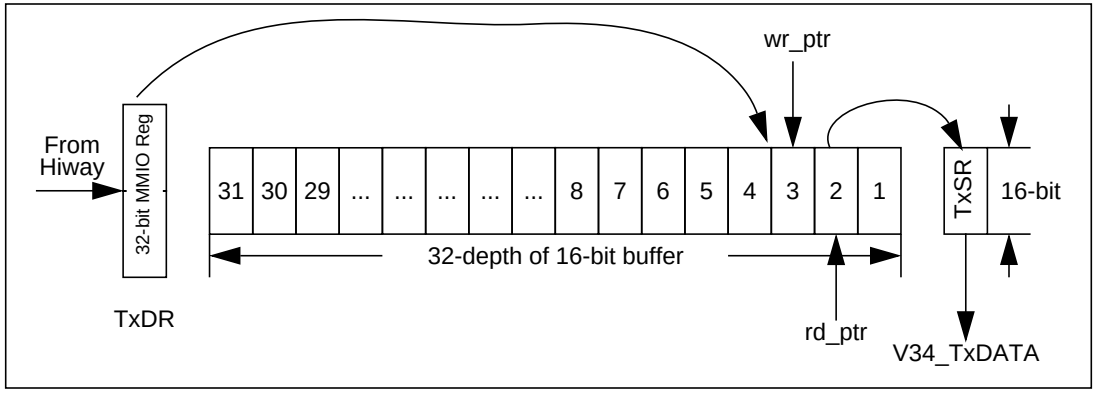


Figure 16-3. The Transmit Buffer

16.5.1.2 Setup V34CR

Write the V34CR to reset and enable the transmitter. The recommended procedure is to set up all transmitter related control bits before performing a TXE assert. In particular, fields TCP, RSD, IO1, IO2, FMS, FSP, MOD and TMS should NOT be changed after enabling the transmitter until after the next transmitter reset.

The TxCLK is normally derived from the V34_CLK pin. The direction of shift in the TxSR and the clock edge to shift on must also be configured in V34CR. If the DSPCPU does not poll the V34 SSI status registers, it should enable the transmitter interrupt and set the ILS field by writing to the V34CR to allow interrupt driven servicing of the V34 SSI. Set the framing controls, slot size, byte-sex and mode required according to the external communication circuit's requirements by writing the V34CR. Finally, set the interrupt level to respond to empty levels in the TxFIFO. Note that the Rx and Tx machines share the framing and clock divide controls. They cannot be set to different values for Rx and Tx.

If the RxCLK used to derive the TxCLK need a divide by two, this must be accomplished by setting V34CSR.CD2.

16.5.1.3 Operation Details

The transmit state machine will wait for transmit data to be written to the TxDR register. As soon as TxDR is written, it will be propagated through one of the TxFIFO and transferred to TxSR, synchronized to TxFSX. Data will begin shifting out of TxSR, on bit for each active edge of the TxCLK, from either bit 31 (MSB first V34CR setting) or from bit 0 (LSB first) until TxSR is empty. When the shift register is empty, the transmit state machine will load the value from the next available TxFIFO location and begin shifting out that data. The transmission continues until the transmit state machine is disabled or reset. If the last available TxFIFO has **not** been updated at the

appropriate time to reload TxSR, the old data is retransmitted and a transmit underrun error (TUE) is indicated in the transmitter status of V34CSR.

16.5.1.4 Interrupt and Status

The refill status of the TxDR register is stored in V34CSR. As the transmit state machine loads a TxFIFO register to the TxSR, it sets the associated status bits. The V.34 SSI will generate an internal interrupt when the number of empty words in the TxFIFO rises above the level set by V34CSR.ILS. If the transmit state machine attempts to read a TxFIFO while the last available TxFIFO has not been updated, it will set the transmit underrun bit. This will usually constitute a protocol error in the transmission.

The WAW and TDE fields of the V34CSR are updated automatically by the V34 SSI.

16.5.2 Receive

16.5.2.1 Receiver Logic Model

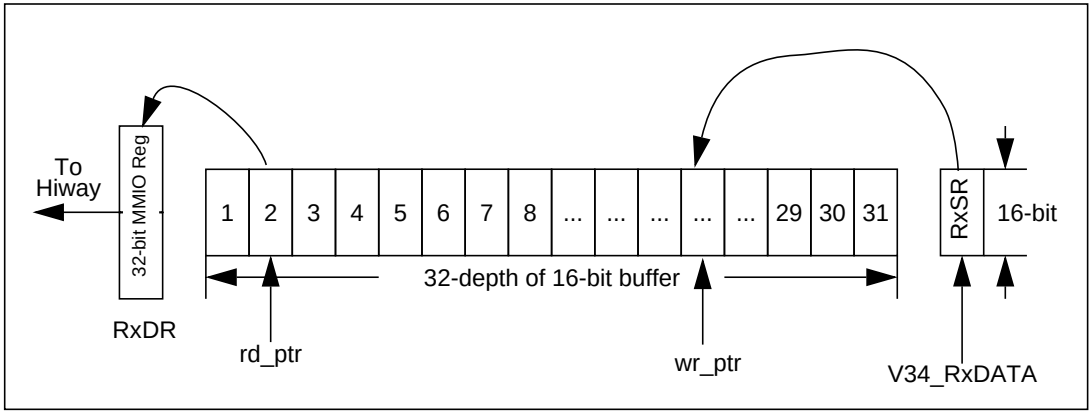


Figure 16-4. The Receive Buffer

16.5.2.2 Setup V34CR

Write the V34CR to reset and enable the receiver. The recommended procedure is to set up all receiver related control bits before performing a RXE assert. In particular, fields TCP, RSD, IO1, IO2, FMS, FSP, MOD and TMS should NOT be changed after enabling the receiver until after the next receiver reset.

The direction of shift in the RxSR, mode, byte-sex and the clock edge polarity must also be configured in V34CR. Set the framing controls according to the external communication circuit's requirements. Note that the Rx and Tx machines share the framing and clock divide controls.

If the DSPCPU does not poll the V34 SSI status registers, it should enable the receiver interrupt and set the ILS field by writing to the V34CR to allow interrupt driven servicing of the V34 SSI receiver.

If the RxCLK used is derived from the V34_CLK by dividing by two, this must be accomplished by setting V34CSR.CD2.

16.5.2.3 Operation Details

The receive state machine will begin shifting V34_RxDATA into the RxSR on the first active edge of RxCLK received after the Rx is enabled. When full, the RxSR is parallel transferred to the first available RxFIFO and possibly RxDR. Reception continues and when RxSR is full again, a parallel load of the next available RxFIFO from RxSR is accomplished. This continues until the receiver is disabled or reset. If the receive state machine must shift RxSR into one of the RxFIFO and none of the RxFIFO is available, the value will be lost and the receive overrun bit will be set.

16.5.2.4 Interrupt and Status

The unload status of the RxDR register is stored in V34CSR. As the receive state machine loads RxFIFO from the RxSR, it sets the associated status bit. The V.34 SSI will generate an internal interrupt when the level of the RxFIFO is full. If the receive state machine attempts to load RxFIFO while none of the RxFIFO is available, it will set the receive overrun bit and generate an interrupt.

Due to the possibility of speculative reading of the RxDR, the DSPCPU must explicitly indicate a successful read of RxDR by writing 'xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxx1 to the V34UPD register. The status fields of the V34CSR will update within 3 TRI_CLKIN cycles after completion of writing to V34 UPD.

16.5.3 GP I/O

The V34_IO1 and V34_IO2 external pins may be used as general purpose I/O by proper configuration of the V34CR. The IO1 function and IO2 function fields of the V34CR control the direction and functionality of these two pins. A hardware reset or a software reset of the transmitter through V34CR.TXR command sets both fields to 11b, a conflict-free initial pin state.

For V34_IO1, a Mode Select value 00b turns the pin into a general purpose output with positive logic polarity, i.e. the pin reflects the WIO1 field value in V34CR.

A Mode Select value 01 turns the V34_IO1 pin into a general purpose input, with optional change detector function. The input state can be read in V34CSR.RIO1. The V34_IO1 pin is fitted with a change detector. The change detector is clocked by the internal RxCLK. The change detector may optionally generate an interrupt, under the control of CDE bit of V34CR.

A mode select value 10b enables the V34_IO1 pin to be used as TxCLK input.

A mode select value 11b puts V34_IO1 in tri-state, with input signal value ignored.

For V34_IO2, a Mode Select value 00b configures the V34_IO2 pin as general pupose positive logic output, reflecting the state of V34CR.WIO2.

A Mode Select value 01b enables V34_IO2 as general purpose input. Its state can be read in V34CSR.RIO2. No change detector is provided for this pin.

A Mode Select value 10b enables V34_IO2 as output, generating V34_TxFSX, i.e. a transmit frame sync signal.

A Mode Select value 11b sets V34_IO2 as V34_TxFSX input. External logic should provide a transmit frame sync signal.

16.5.4 Test Modes

16.5.4.1 Remote Loopback

This test mode allows a remote transmitter to test itself, the intervening transmission media and its associated receiver. In this mode, the data received on V34_RxDATA pin is buffered and transmitted on V34_TxDATA pin. The data is not transferred to TxDR/TxFIFO and the DSPCPU is never interrupted. The transmitter is clocked by V34_RxCLK with a combinatorial clock delay.

16.5.4.2 Local Loopback

This test mode allows the DSPCPU to run local checks of the V.34 SSI. Data written to the TxFIFO is serialized and passed to the receiver via an internal serial connection. The receiver deserializes the data and passes it to the RxFIFO register. Interrupts will be generated if enabled. During local loopback mode, the data on the V34_RxDATA pin is ignored and the V34_TxDATA pin is tristated. An external V34_CLK must be provided during local loopback mode or no transmission or reception will occur.

16.5.5 The V.34 Synchronous Serial Interface

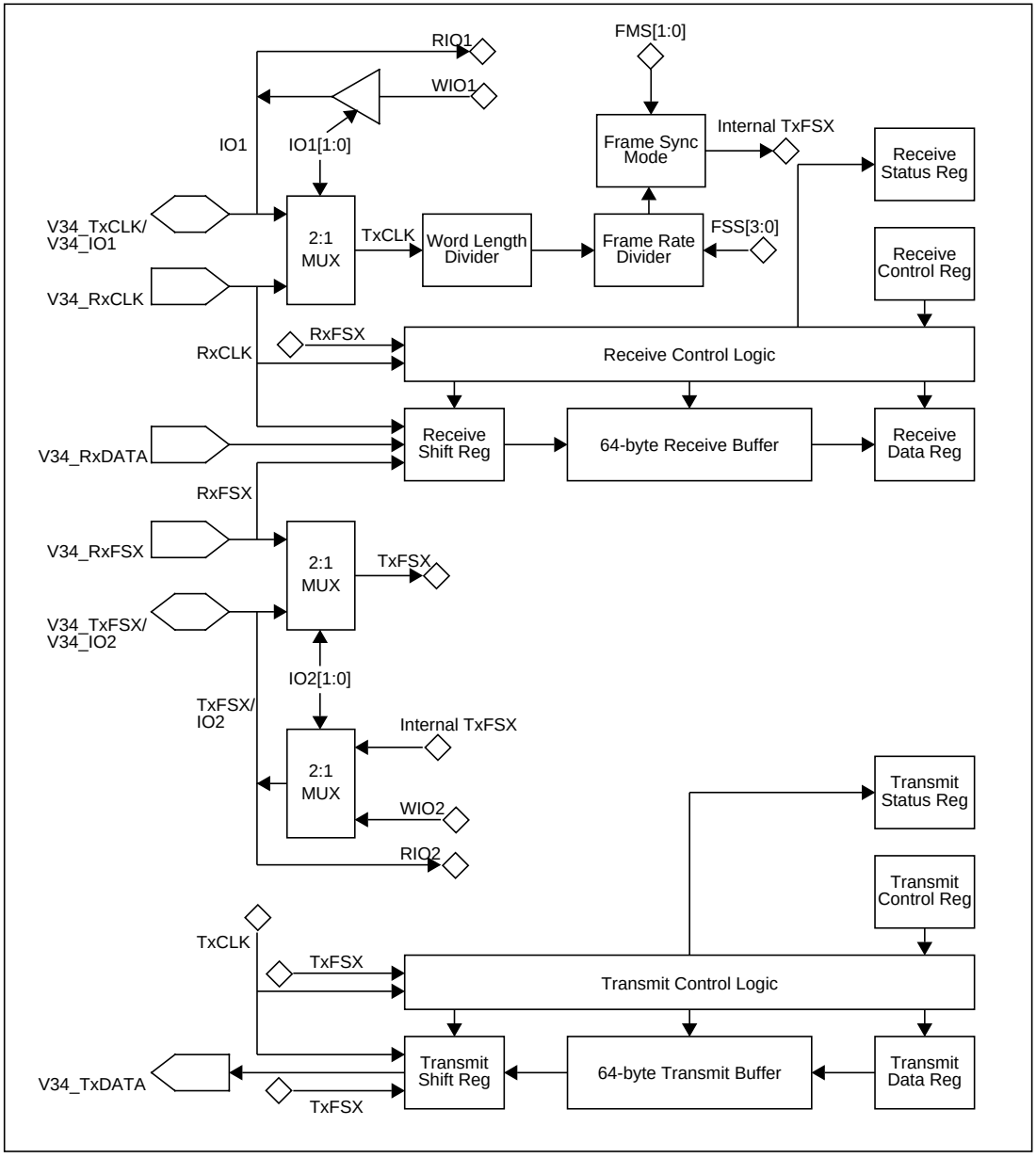


Figure 16-5. The V.34 Sync Serial Interface Block Diagram

JTAG Functional Specification Chapter 17

by Renga Sundararajan and Hans Bouwmeester

17.1 OVERVIEW

The JTAG port on a TriMedia processor is used for communication between a debug monitor running on a TriMedia processor and a debugger front-end running on a host. It is also used for hardware testing which is beyond the scope of this chapter.

The enhancements to the standard functionality of JTAG test logic provide a handshake mechanism for transferring data to and from a TriMedia processor's MMIO registers reserved for this purpose, for posting an interrupt, and for resetting processor state. The actual interpretation of the contents of the MMIO registers is determined by a software protocol used by the debug monitor running on TriMedia processor and the debug front-end running on a host machine.

IEEE 1149.1 (JTAG) standard is used for board level testing of integrated circuits, for testing the internals of the integrated circuits, and for monitoring and modification of a running system. The JTAG standard defines on-chip test logic, which consists of an instruction register, a group of test data registers including a bypass register and a boundary-scan register, four dedicated pins collectively called the Test Access Port (TAP) and a TAP controller. The TAP controller is a finite-state machine. It selects a JTAG instruction or a data register to store the input based on TMS signal, receives instructions and data on the TDI pin, executes the instruction when triggered by TMS, and shifts data out of TDO. The standard defines some instructions that shall be implemented by a TAP controller.The standard allows enhancements to the functionality of a JTAG controller to include, for example, debugging support and still conform to IEEE 1149.1 standard.

Figure 17-1 shows an overview of the JTAG access path from a host machine to a target TriMedia system and a simplified block diagram of the TriMedia processor. The JTAG Interface Module shown separately in the diagram may be a PC add-on card such as PC-1149.1/100F Boundary Scan Controller Board from Corelis Inc or a

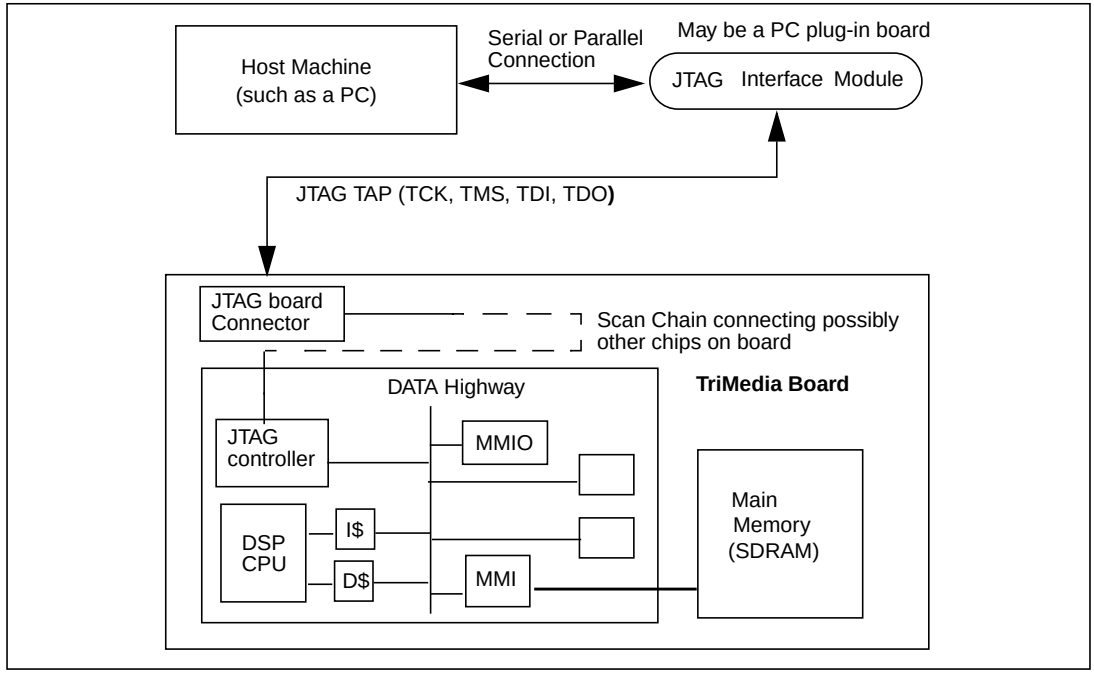


Figure 17-1. TriMedia System with JTAG Test Access

similar module connected to a PC serial or parallel port. The JTAG interface module is necessary only for TriMedia systems that are not plugged into a PC. For PC-hosted TriMedia systems, the host based debugger front-end can communicate with the target resident debug monitor via the PCI bus.

The communication between a host computer and a target TriMedia system via JTAG requires, at a high level of abstraction, the following components.

- **A Host computer with a serial or parallel interface.**

  The host computer transfers data to and from the JTAG interface module, preferably in word-parallel fashion. Also needed is JTAG interface device driver software to access and modify the registers of the JTAG interface module.
- **A JTAG interface module (hardware) that asynchronously transfers data to and from the host computer.**

  The interface module synchronously transfers data to and from the JTAG TAP on a TriMedia processor, supplies the test clock TCK and other signals to the JTAG controller on TriMedia. The interface module may be a PC plug-in board.

  This module may transfer data from and to the host computer in bit-serial or word-parallel fashion. It transfers data from and to the JTAG registers on a TriMedia processor in bit-serial fashion in accordance with the IEEE 1149.1 standard. The JTAG interface module connects to a 4 pin JTAG connector on a TriMedia board which provides a path to the JTAG pins on a TriMedia processor. It is the responsibility of the interface module to scan data in and out of the TriMedia processor into its internal buffers and make them available to the host computer.
- **A JTAG controller on the TriMedia processor which provides a bridge between the external JTAG TAP and the internal system.**

  The controller transfers data from/to the TAP to/from its scannable registers asynchronous to the internal system clock. A monitor running on a TriMedia processor and the debugger front-end running on a host computer exchange data via JTAG by reading/writing the MMIO registers reserved for this purpose, including a control register used for the handshake.

The following sections deal only with the additional JTAG TAP controller registers and functionality necessary for software debugging via JTAG interface.

17.2 TEST ACCESS PORT (TAP)

The Test Access Port includes three dedicated input pins, Test Data In (TDI), Test Mode Select (TMS), and Test Clock (TCK) and one output pin Test Data Out (TDO).

TCK provides the clock for test logic required by the standard. TCK is asynchronous to the system clock. Stored state devices in JTAG controller must retain their state indefinitely when TCK is stopped at 0.

The signal received at TMS is decoded by the TAP controller to control test functions. The test logic is required to sample TMS at the rising edge of TCK.

Serial test instructions and test data are received at TDI. The TDI signal is required to be sampled at the rising edge of TCK. When test data is shifted from TDI to TDO, the data must appear without inversion at TDO after a number of rising and falling edges of TCK determined by the length of the instruction or test data register selected.

TDO is the serial output for test instructions and data from he TAP controller. Changes in the state of TDO must occur after the falling edge of TCK. This is because devices connected to TDO are required to sample TDO at the rising edge of TCK. The TDO driver must be in an inactive state (i.e., TDO line must float) except when the scanning of data is in progress.

17.2.1 TAP Controller

The TAP controller is a finite state machine and it synchronously responds to changes in TCK and TMS signals. The TAP instructions and data are serially scanned into the TAP controller's instruction and data registers via the common input line TDI. The TMS signal tells the TAP controller to select either the TAP instruction register or a TAP data registers as the destination for serial input from the common line TDI. An instruction scanned into the instruction register selects a data register to be connected between TDI and TDO and hence to be the destination for serial data input.

The TAP controller's state changes are determined by the TMS signal which must be sampled at rising edges of TCK. The states are used for scanning in/out TAP instruction and data, updating instruction, and data registers, and for executing instructions.

The TAP controller must be in Test Logic Reset state after power-up. It remains in that state as long as TMS is held at 1. The controller transitions to Run-Test/Idle state from Test Logic Reset state when TMS = 0. The Run-Test/Idle state is an idle state of the controller in between scanning in/out an instruction/data register. The "Run-Test" part of the name refers to start of built-in tests. The "Idle" part of the name refers to all other cases. Note that there are two similar sub-structures in the state diagram, one for scanning in an instruction and another for scanning in data. To scan in/out a data register, one has to scan in an instruction first. Each instruction selects a data register that is connected between TDI and TDO.

The controller's state diagram (Figure 17-2) shows separate states for "capture", "shift" and "update" of data and instructions. The reason is to leave the contents of a data register or an instruction register undisturbed until serial scan in is finished and the update state in entered. By separating the shift and update states, the contents of a register (by that we mean the parallel stage) are not affected *during* scan in/out.

An instruction or data register must have at least two stages, the shift register stage and the parallel input/output stage. When an n-bit register is to be "read", the register is selected by an instruction, the registers contents

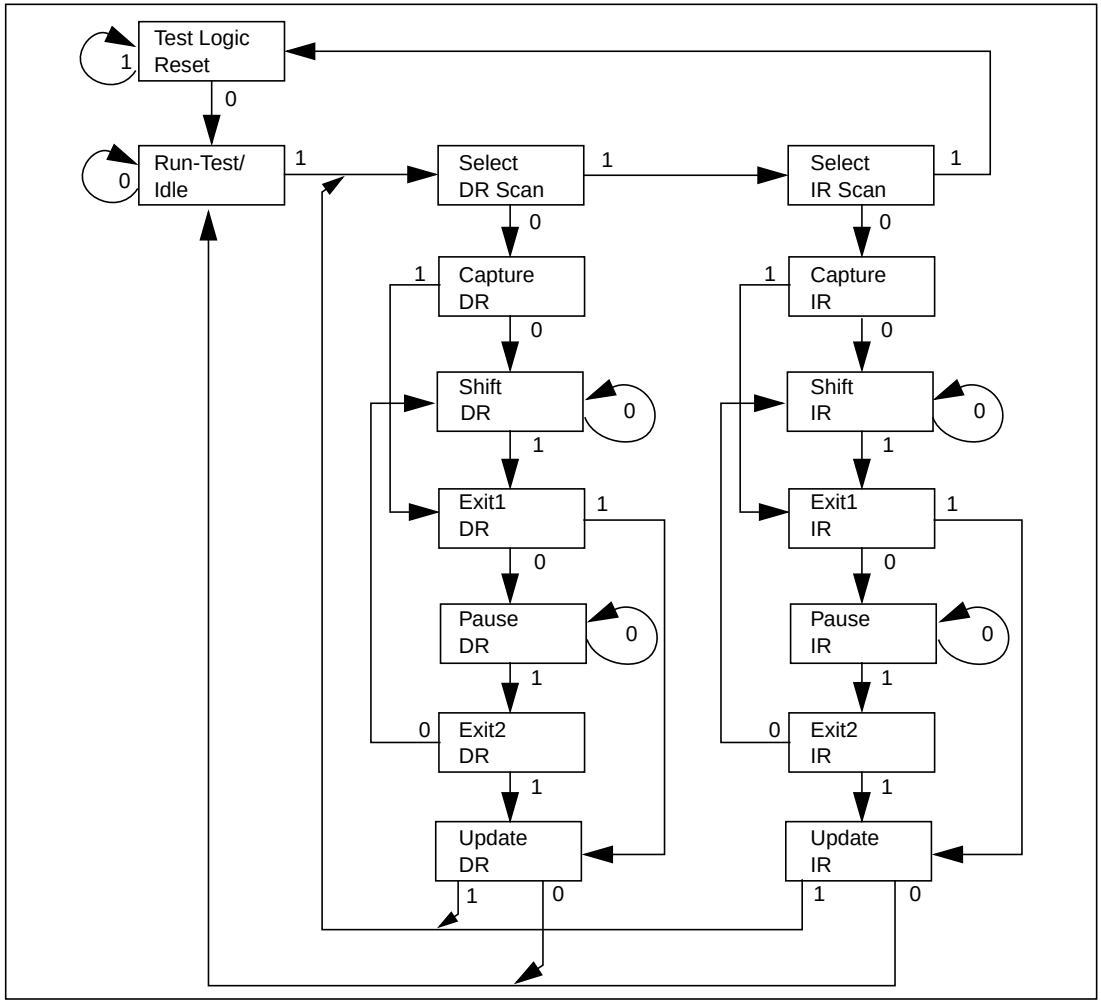


Figure 17-2. State Diagram of TAP controller

are “captured” first (loaded in parallel into shift register stage), n bits are shifted in and at the same time n bits are shifted out, and finally the register is “updated” with the new n bits shifted in.

Note that when a register is scanned, it old value is shifted out of TDO and the new value shifted in via TDI is written to the register at the update state. Hence, scan in/out involve the same steps. This also means that reading a register via JTAG destroys it contents *unless otherwise stated.* We can specify some registers as *read-only* via JTAG so that when the controller transitions to update state for the read-only register, the update has no effect. Some times, we need read-write registers (for example, control registers used for handshake) which must be read non-destructively. In such cases, the value shifted in determines whether the old value is “remembered” or something else happens. The following section specifies additional registers that are *read-only* and *read-write*.

Table 17-1. MMIO Register Assignments

MMIO Offset	JTAG Register
0x 10 3800	JTAG_DATA_IN
0x 10 3804	JTAG_DATA_OUT
0x 10 3808	JTAG_CTRL

17.2.2 JTAG Instruction and Data Registers

The JTAG standard requires a JTAG instruction register and a minimum of two data registers, the bypass and boundary scan registers. Design specific data registers may be added by an implementation. We add two JTAG data and one control registers (see Figure 17-3) in MMIO space and augment the JTAG instruction set. Table 17-1 lists the MMIO addresses of the JTAG data and control registers. The addresses are offsets from MMIO_base. All references to instruction and data registers below are

JTAG instruction and data registers and not TriMedia instruction or data registers.

- **Two new 32-bit data registers, JTAG_DATA_IN and JTAG_DATA_OUT**. They are connected in parallel with the standard Bypass and Boundary Scan registers of JTAG (not shown in Figure 17-3).

  The JTAG_DATA_IN register can be read or written to via the JTAG port. The JTAG_DATA_OUT register is read-only via the JTAG port, so that scanning out JTAG_DATA_OUT is non-destructive.

  The JTAG_DATA_IN and JTAG_DATA_OUT are readable/writable from the TriMedia processor via the usual load/store operations.

- **An 8-bit control register JTAG_CTRL in MMIO space.** The JTAG_CTRL register is used for handshake between a debug monitor running on a TriMedia and a debugger front-end running on a host.

  JTAG_CTRL.ofull = 1 means that JTAG_DATA_OUT has valid data to be scanned out. On power-on reset of the TriMedia Processor, JTAG_CTRL.ofull = 0. JTAG_CTRL.ofull is both readable and writable via JTAG tap. Writing 0 to JTAG_CTRL.ofull via JTAG is a 'remember' operation, i.e., JTAG_CTRL.ofull retains its previous state. Writing 1 to JTAG_CTRL.ofull via JTAG is a 'clear' operation, i.e., JTAG_CTRL.ofull becomes 0.

  JTAG_CTRL.ifull = 0 means that the JTAG_DATA_IN register is empty. JTAG_CTRL.ifull = 1 means that JTAG_DATA_IN has valid data and the debug monitor has not yet copied it to its private area. On power-on reset of the TriMedia processor, JTAG_CTRL.ifull = 0. JTAG_CTRL.ifull is readable and writable via JTAG. Writing 0 to JTAG_CTRL.ifull via JTAG is a 'remember' operation, i.e., JTAG_CTRL.ifull retains it previous state. Writing 1 to JTAG_CTRL.ifull posts an interrupt on hardware line 18.

  The peripheral blocks on a TriMedia processor may enter a "sleep" state to reduce power consumption. The JTAG_CTRL.sleepless bit determines if the JTAG block participates in a power down state. In the power-on RESET state, JTAG_CTRL.sleepless bit is 1 meaning the JTAG block does not go to sleep. It can be read and written to by the TriMedia processor via load/store operations and by the debugger front-end running on a host by scan in/out.

- **Two virtual registers, JTAG_IFULL_IN and JTAG_OFULL_OUT.** The first virtual register JTAG_IFULL_IN connects the registers JTAG_CTRL.ifull and JTAG_DATA_IN in series. Likewise, the virtual register JTAG_OFULL_OUT connects JTAG_CTRL.ofull and JTAG_DATA_OUT in series.

  The reason for the virtual registers is to shorten the time for scanning the JTAG_DATA_IN and JTAG_DATA_OUT registers. Without virtual registers, we must scan in an instruction to select JTAG_DATA_IN, scan in data, scan an instruction to select JTAG_CTRL register and finally scan in the control register. With virtual register, we can scan in an instruction to select JTAG_IFULL_IN and then scan in both control and data bits. Similar savings can be achieved for scan out using virtual registers.

- **A 5 bit instruction register and five new instructions.**
  - Five instructions SEL_DATA_IN, SEL_DATA_OUT, SEL_IFULL_IN, SEL_OFULL_OUT, and SEL_JTAG_CTRL for selecting the registers to be connected between TDI and TDO for serial input/output.
  - An instruction RESET for resetting the TriMedia processor to power on state.
- In the capture-IR state of the TAP controller, the least 2 significant bits (bits 0 and 1) of the shift register stage must be loaded with the '01' as required in the standard. The standard allows the remaining bits of the IR shift stage to be loaded with design specific data. The bits 2, 3 and 4 of the IR shift stage are loaded with bits 0, 1 and 2 of the JTAG_CTRL register. This means that shifting in any instruction allows the 3 least significant bits of the JTAG_CTRL register to be inspected. This reduces the polling overhead for data transfer.

Given that there are three mandatory instructions and five optional instructions specified in the JTAG standard, we need 4 bit wide instruction register to encode 13 instructions (5 new + 3 mandatory + 5 optional). We use a

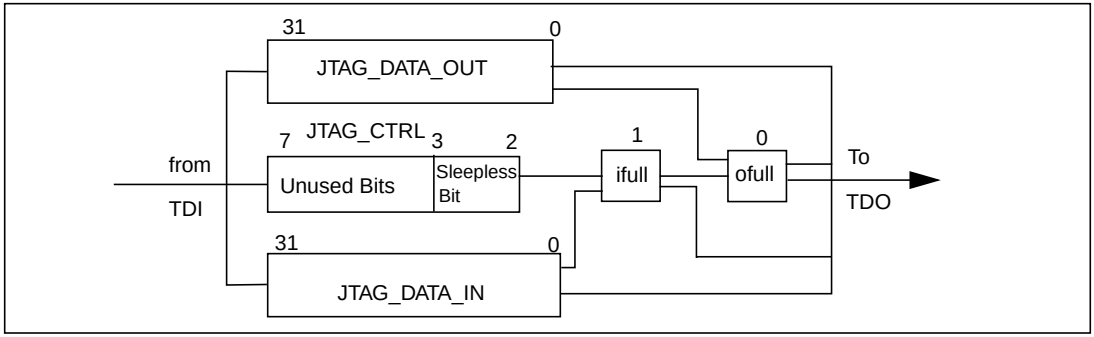


Figure 17-3. Additional JTAG data registers and control register

5 bit instruction register to allow for enhanced debug support via JTAG in future versions of TriMedia. The unused opcodes are private and their effects are undefined.Table 17-2 lists the JTAG instructions.

Table 17-2. JTAG instruction encodings

Encoding	Instruction name	Action
00000	EXTEST	Select (dummy) boundary scan register
00001	SAMPLE/PRELOAD	Select (dummy) boundary scan register
11111	BYPASS	Select bypass register
10000	RESET	Reset TriMedia to power on state
10001	SEL_DATA_IN	Select DATA_IN register
10010	SEL_DATA_OUT	Select DATA_OUT register
10011	SEL_IFULL_IN	Select IFULL_IN register
10100	SEL_OFULL_OUT	Select OFULL_OUT register
10101	SEL_JTAG_CTRL	Select JTAG_CTRL register
11110	MACRO	Hardware test mode select

The JTAG instructions EXTEST, SAMPLE/PRELOAD, and BYPASS are standard instructions and are not discussed here. The MACRO instruction is used for selecting hardware test mode, not discussed here.

Race Conditions

Since the JTAG data registers live in MMIO space and are accessible by both the TriMedia processor and the JTAG controller at the same time, race conditions must not exist either in hardware or in software. The following communication protocol uses a handshake mechanism to avoid software race conditions.

17.2.3 JTAG Communication Protocol

The following describes the handshake mechanism for transferring data via JTAG.

- **Transfer from debug front-end to debug monitor**

  The debugger front-end running on ahost transfers data to a debug monitor via JTAG_DATA_IN register. It must poll JTAG_CTRL.ifull bit to check if JTAG_DATA_IN register can be written to. If the JTAG_CTRL.ifull bit is clear, the front-end may scan data into JTAG_DATA_IFULL_IN register. Note that data and control bits may be shifted in with SEL_IFULL_IN instruction and the bit shifted into JTAG_CTRL.ifull register must be 1. This action triggers an interrupt. The debug monitor must copy the data from JTAG_DATA_IN register into its private area when servicing the interrupt and then clear JTAG_CTRL.ifull bit thus allowing JTAG interface module to write to JTAG_DATA_IN register the next piece of data.
- **Transfer from monitor to front-end**

  The monitor running on TriMedia must check if JTAG_CTRL.ofull is clear and if so, it can write data to JTAG_DATA_OUT. After that, the monitor must set the JTAG_CTRL.ofull bit. The debugger front-end polls the JTAG_CTRL.ofull bit. When that bit is set, it can scan out JTAG_DATA_OUT register and clear JTAG_CTRL.ofull bit. Since JTAG_DATA_OUT is *read-only* via JTAG, the update action at the end of scan out has no effect on JTAG_DATA_OUT. The JTAG_CTRL.ofull bit, however, must be cleared by shifting in the value 1.
- **Controller States**

  In the power-on reset state, JTAG_CTRL.ifull and JTAG_CTRL.ofull must be cleared by the JTAG controller.

17.2.4 Example Data Transfer Via JTAG

Scanning in a 5-bit instruction will take 12 TCK cycles from the Run-Test/Idle state - 4 cycles to reach Shift-IR state, 5 cycles for actual shifting in, 1 cycle to exit1-IR state, 1 cycle to Update-IR state, and 1 cycle back to Run-Test/Idle state. Likewise, scanning in a 32 bit data register will take 38 TCK cycles and transferring an 8-bit JTAG_CTRL data register will take 14 TCK cycles from Idle state. However, if a data transfer follows instruction transfer, then transitioning to DR scan stage can be done without going through Idle state, saving 1 cycle.

17.2.4.1 Transfer of Data to TriMedia Via JTAG

Poll control register to check if input buffer is empty or not and scan in data when it is empty and set the ifull control bit to 1 triggering an interrupt. Note that scanning in any instruction automatically scans out the 3 least significant bits (including ifull and ofull bits) of JTAG_CTRL register.

Table 17-3. Transfer of Data in via JTAG

Action	Number of TCK cycles
IR shift in SEL_IFULL_IN instruction	12
While JTAG_CTRL.ifull = 1, scan in SEL_IFULL_IN instruction	11+
DR scan 33 bits of register JTAG_IFULL_IN	38
TOTAL	61+ cycles

17.2.4.2 Transfer of Data from TriMedia Via JTAG

Poll control register to check if output buffer is full or not and scan out data when it is full and clear the ofull control bit. Note that scanning in any instruction automatically scans out the 3 least significant bits (including ifull and ofull bits) of JTAG_CTRL register.

Note that the above timings do not include the overheads of the JTAG driver software driving the JTAG interface module plugged into a PC.

Table 17-4. Transfer of Data out via JTAG

Action	Number of TCK cycles
IR shift in SEL_OFULL_OUTinstruction	12
While JTAG_CTRL.ofull = 0, scan in SEL_OFULL_OUT instruction	11+
DR scan 33 bits of register JTAG_OFULL_OUT	38
TOTAL	61+ cycles

17.2.5 JTAG Interface Module

It is expected that the interface module will be a programmable JTAG interface module, one end of which is connected to a JTAG tap and the other end is connected to a host computer via a serial line or parallel line or plugged in to a PC. It is up to the JTAG driver software on a host computer to program the JTAG interface module via the serial/parallel interface for transferring data to/from the target. The transfer rates will depend on the interface module.

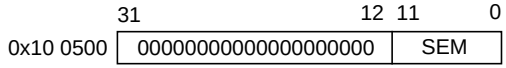
TM1000 has a simple MP semaphore assist device. It is an 32 bits register, accessible through MMIO by either the local TM1000 CPU or by any other CPU on PCI through the aperture made available on PCI. The semaphore “SEM” is located at MMIO offset 0x10 0500.

The operation is as follows: each master in the system constructs a personal nonzero 12 bit ID (see below). To obtain the global semaphore, a master does the following action:

```
write ID to SEM (use 32 bit store, with ID in 12 LSB)
retrieve SEM    (use 32 bit load, it returns 0x00000nnn)
if (SEM = ID) {
    "performs a short critical section action"
    write 0 to SEM
}
else "try again later, or loop back to write"
```

18.1 SEM DEVICE SPECIFICATION

SEM is a 32 bits MMIO location. The 12 LSB consist of storage flip-flops with surrounding logic, the 20 MSB's always return a zero when read.



SEM is RESET to zero by powerup reset.

When SEM is written to, the storage flip-flops behave as follows:

```
if (cur_content == 0)        new_content = write_value;
else if (write_value == 0)   new_content = 0;
/* ELSE NO ACTION ! */
```

18.2 CONSTRUCTING A 12-BIT ID

A TM1000 processor can construct a personal, nonzero 12 bit ID in a variety of ways. Below are some suggestions.

PCI configspace PERSONALITY entry. Each TM1000 receives a 16 bits PERSONALITY value from the EEPROM during boot. This PERSONALITY register is located at offset 0x40 in configuration space. In a MP system, some of the bits of PERSONALITY can be individualized for each CPU involved, giving it a unique 2/3/4 bit ID, as needed given the max. number of CPU's in the design.

In the case of a host-assisted boot of TM1000, the PCI BIOS assigns a unique MMIO_base and DRAM_base to each and every TM1000. In particular, the 11 MSB's of each MMIO_base are unique, since each MMIO aperture is 2 MByte in size. These bits can be used as a personality ID. Use bit 11 (MSB) equal '1' to guarantee a nonzero ID#.

18.3 WHICH SEM TO USE

Each TM1000 in the system adds a SEM device to the mix. The intended use is to treat one of these SEM devices as THE master semaphore in the system. Many methods can be used to determine which SEM is master SEM. Some examples below:

Each DSPCPU can use PCI configurationspace accesses to determine which other TM1000's are present in the system. Then, the TM1000 with the lowest PERSONALITY number, or the lowest MMIO_base is chosen as the TM1000 containing the master semaphore.

18.4 USAGE NOTES

To avoid contention on the master SEM device, it should only be used for inter-processor semaphores. Processes running on a single CPU can use regular memory to implement synchronization primitives.

The critical section associated with SEM should be kept as short as possible. Preferably, SEM should only be used as the basis to make multiple memory resident simple semaphores. In this case, the non-cacheable DRAM area of each TM1000 can be used to implement the semaphore datastructures efficiently.

As described here, SEM does not guarantee starvation-free access to critical resources. Claiming of SEM is purely stochastical. This should work fine as long as SEM is overloaded. Utmost care should be taken in SEM access frequency and duration of the basic critical sections to keep the load conditions reasonable.

Arbiter Chapter 19

by Chris Nelson, Eino Jacobs, Allan Tzeng, Gert Slavenburg

19.1 DOCUMENT STATUS

This document is still under construction (more examples needed).

19.2 ARBITER

The TM1000 highway has a central arbiter, that is embedded in the main memory interface. All traffic on the highway is controlled by this arbiter.

The arbiter has the following primary characteristics:

- round robin arbitration
- hierarchical organization
- programmable allocation of highway bandwidth
- dual priorities with priority raising mechanism

These features are explained in the following chapters.

19.3 DUAL PRIORITIES WITH PRIORITY RAISING MECHANISM

The best CPU performance is obtained if cache misses can take priority over I/O traffic on the highway. However, there needs to be a maximum guaranteed latency that is low enough to satisfy the real time constraints of I/O units.

This is achieved with the following architecture for priorities with priority raising mechanism.

Highway requests can have 2 priorities: low priority and high priority. Within each class there is fair, round-robin arbitration. Requests with high priority take precedence over requests with low priority. Devices can indicate the priority of their requests to be low or high. A device may initially post a request with low priority. If it does not get serviced within a particular waiting time, then the device can raise the priority of the request to high priority. This can be done when the worst case latency at high priority approaches the real time constraint of the device. Thus, the device uses only spare bandwidth without slowing the CPU unless real time constraints require it to claim high priority.

In TM1000, the ICP unit has its own priority raising logic. Refer to Chapter 13, “Image Co Processor,” for more information.

Priority raising for the VLD, PCI, VI and VO units is handled by the MMI’s central priority raising mechanism. The central priority raising mechanism is controlled by the ARB_RAISE MMIO register (see Table 19-1). Each request by the unit is sent first as reql for a programmed length of time, then raised to a reqh. This allows real-time constraints to be met, yet allows other transactions to proceed unimpeded as long as possible. Each unit is allocated five bits in ARB_RAISE. The granularity of the delay is 16 cycles, so the maximum time spent in each reql can be programmed to between 0 and 496 cycles, inclusive, at 16 cycle intervals.

Table 19-1. ARB_RAISE register layout (MMIO offset 0x10010c)

Bits	Value
19:15	VLD_delay
14:10	PCI_delay
9:5	VI_delay
4:0	VO_delay

The default value for the entire ARB_RAISE register is 0. This causes all requests from those units to be handled as high-priority requests until software changes the ARB_RAISE register contents. Note that there is some risk in setting the delay high, then lowering it, as the last request submitted with the high delay might violate the latency constraints of the new real-time domain.

19.4 ROUND ROBIN ARBITRATION ALGORITHM

When requests have the same priority a round-robin arbitration algorithm is used. The purpose of the round-robin arbitration is to assure every device with a high priority request of a maximum latency for gaining access to the highway and a minimum share of bandwidth. In this way it is assured that no starvation of requests can occur and that requests with real-time constraints can be handled in time.

The round-robin algorithm implemented is hierarchical, weighted, programmable round-robin.

Not all devices need to have equal latency and bandwidth. It is preferred to allocate bandwidth to units according to their need. This is achieved with hierarchical, weighted round-robin. The weights can be adjusted by software, allowing to adjust bandwidth allocation depending on application need.

Round robin arbitration works as follows:

Requests are granted according to a priority list. This list is not static. Whenever a device gets a request granted it will be moved to the last position in the priority list and another device will be moved to the first position in the priority list. Priorities are rotated. A device with a waiting request will eventually reach the first place in the priority list.

Hierarchy and weighting are added to this algorithm as follows:

The devices are grouped into several levels of hierarchy. Every level of hierarchy receives a fixed quota of bus transactions, giving it a fixed share of total bus bandwidth.

Within a level of hierarchy the devices can have equal weight, giving them an equal share of bandwidth or they can have different weights, giving them an unequal share of the bandwidth for that level.

There is a programming option that allows to control the allocation of bandwidth to the levels of hierarchy by selecting arbitration weights from a list of possible choices.

To illustrate the arbitration mechanism a few examples of arbitration state machines are given.

In Figure 19-1 an example bubble diagram of a an arbitration state machine is given with 2 requesters. The nodes A and B indicate states A and B. In state A requester A has ownership of the highway, in state B requester B has ownership. The arc from state A to state B indicates that when we're in state A and a request from requester B is asserted, then a transition to state B occurs, i.e. ownership of the highway passes from requester A to requester B. When in a particular state none of the arcs leaving from that node has its condition fulfilled, then the state machine remains in the same state.When both requester A and B have requests asserted, then ownership of the highway switches between A and B, creating fair allocation of ownership. No distinction is made between high priority and low priority requests in this example.

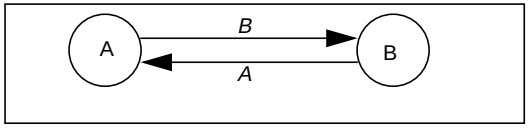


Figure 19-1. State diagram of round robin arbitrator with 2 requesters.

In Figure 19-2 an example is given of a state machine with two requesters A and B with double weight given to requester A. There are now 2 states A1 and A2, and in both of these requester A has ownership of the highway. When both A and B requests are asserted, then requester A will have twice as often ownership of the highway as requester B.

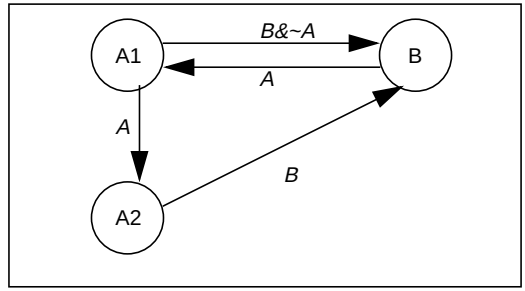


Figure 19-2. State diagram of round robin arbitrator with 2 requesters; requester A has double

In Figure 19-3 an example is given of a state machine with 3 requesters.

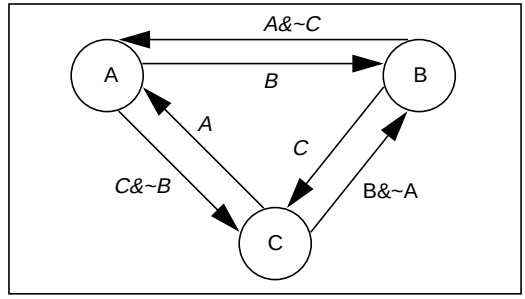


Figure 19-3. State diagram of round robin arbitrator with 3 requesters.

In Figure 19-4 an example is given of a state machine with 3 requesters in which double weight is given to requester A.

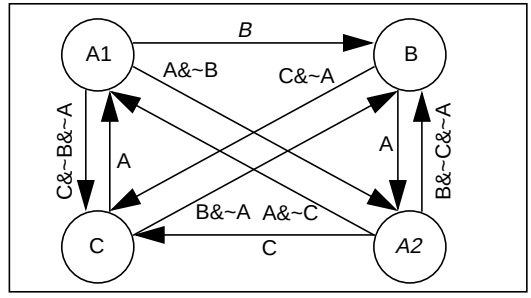


Figure 19-4. State diagram of round robin arbitrator with 3 requesters; request A has double

19.5 PRIORITIES FOR CACHE TRAFFIC

The different types of requests from the DSPCPU caches are arbitrated amongst each other, resulting in a single CPU request to the MMI arbiter.

19.6 ARBITRATION HIERARCHY

19.6.1 Arbitration Levels

The arbitration is split into multiple levels of hierarchy. Each level of hierarchy constitutes an independent arbitration state machine. At the bottom of the hierarchy arbitration is between a group of devices. Whichever of these devices 'wins' is passed to the next level of hierarchy where the selected device competes with other devices at that level for highway access. This is continued until the highest level of arbitration. By splitting arbitration into multiple levels it is easy to support a large number of highway devices while the complexity of the arbitration state machines at each level of hierarchy remains modest. Hierarchy makes it also easy and natural to allocate bus bandwidth to a group of devices. For instance audio devices are grouped together at the bottom of the hierarchy and get a small amount of overall bandwidth.

The arbitration hierarchy consists of 6 levels, as indicated in Figure 19-5.

19.6.2 Arbitration Weights Per Level

The arbitration weights at each level are shown in Table 19-2.

The arbitration weights are implemented by giving each device a number of nodes in the arbitration state machine equal to its weight. For programmable weights only part of the nodes in the state machine are activated.

19.6.3 Programmable Bandwidth Per Level

The allocation of bandwidth is programmable by setting weights.

Bandwidth at level 1 can be allocated between DSPCPU caches and level 2 by programming weights for both.

Bandwidth should be chosen such that enough of it is available for real-time operation of peripheral devices.

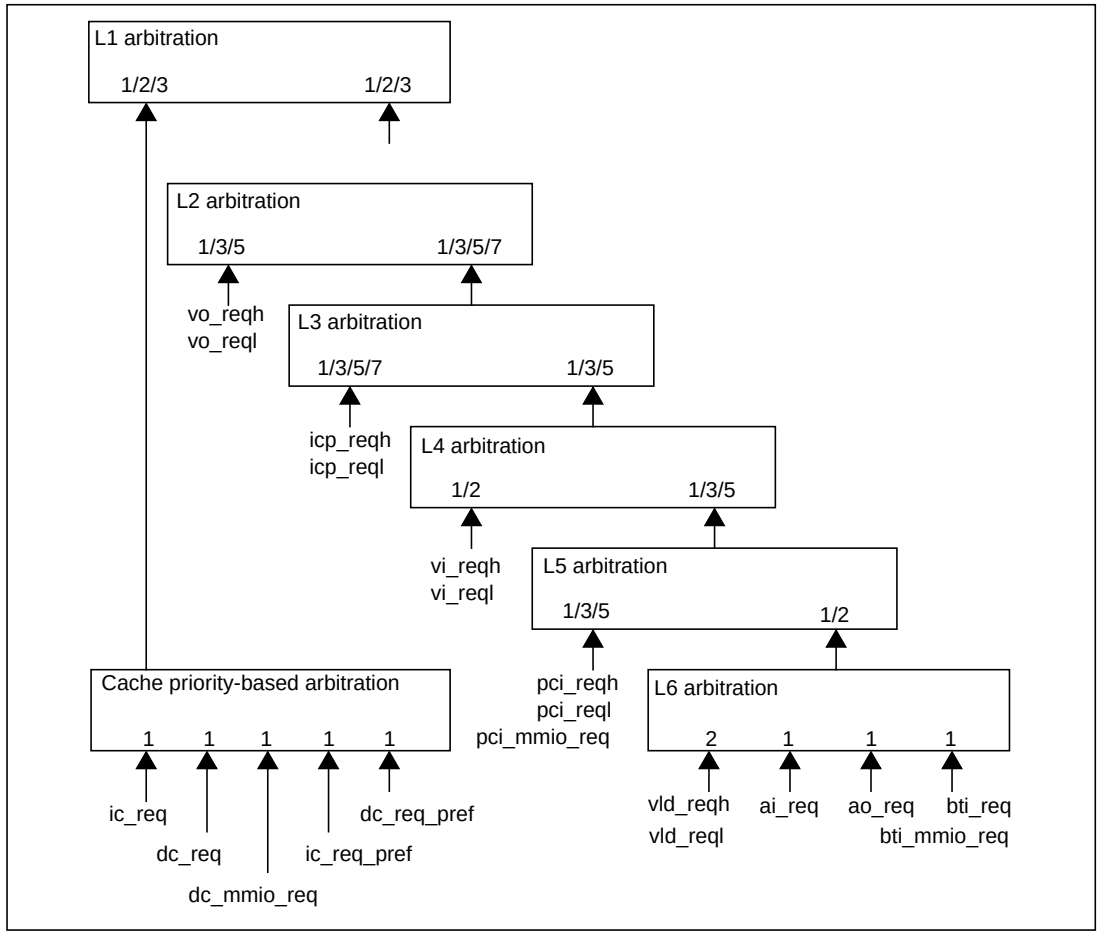


Figure 19-5. Arbitration diagram

Table 19-2. Arbitration Weights at Each Level

Level	Arbitration Weights
level 1:	CPU MMIO, Dcache, Icache are arbitrated with fixed priorities between each other and together have a programmable weight of 1, 2 or 3. Level 2 has a programmable weight of 1, 2 or 3.
level 2:	Video Out has a programmable weight of 1, 3 or 5. Level 3 has a programmable weight of 1, 3, 5 or 7.
level 3:	The ICP has a programmable weight of 1,3,5 or 7. Level 4 has a programmable weight of 1,3 or 5.
level 4	The Video In unit has a programmable weight of 1 or 2. Level 5 has a programmable weight of 1,3 or 5.
level 5:	Level 6 has a programmable weight of 1 or 2. Level 6 has a programmable weight of 1 or 2.
level 6:	Level 6 contains several lower bandwidth, latency tolerant devices. The VLD has a weight of 2. Audio In and Audio Out each have a weight of 1. The boot block (only active during booting) and I2C interface share a fixed weight of 1.

Table 19-3. Bandwidth Allocation Between CPU Caches and Peripheral Units.

weight of CPU and caches	weight of level 2	bandwidth at level 1	bandwidth at level 2
3	1	75%	25%
2	1	67%	33%
3	2	60%	40%
1	1	50%	50%
2	3	40%	60%
1	2	33%	67%
1	3	25%	75%

Otherwise as much bandwidth as possible should be given to the CPU.

Bandwidth allocation for all other levels can be derived from the weights.

??? tables to be inserted here ???

19.7 ARB_BW_CTL MMIO REGISTER

The bandwidth allocation can be selected by programming the MMIO register ARB_BW_CTL.

Table 19-4. ARB_BW_CTL MMIO register (MMIO offset 0x100104)

level of arbitration	field	bits	allowed values
n/a	RESERVED	25:22	
n/a	RESERVED	19:18	

Table 19-4. ARB_BW_CTL MMIO register (MMIO offset 0x100104) (Continued)

level of arbitration	field	bits	allowed values
level 1	CPU weight	17:16	00 = weight 1 01 = weight 2 10 = weight 3
level 1	L2 weight	15:14	00 = weight 1 01 = weight 2 10 = weight 3
level 2	VO weight	13:12	00 = weight 1 01 = weight 3 10 = weight 5
level 2	L3 weight	11:10	00 = weight 1 01 = weight 3 10 = weight 5 11 = weight 7
level 3	ICP weight	9:8	00 = weight 1 01 = weight 3 10 = weight 5 11 = weight 7
level 3	L4 weight	7:6	00 = weight 1 01 = weight 3 10 = weight 5
level 4	VI weight	5	0 = weight 1 1 = weight 2
level 4	L5 weight	4:3	00 = weight 1 01 = weight 3 10 = weight 5
level 5	PCI weight	2:1	00 = weight 1 01 = weight 3 10 = weight 5
level 5	L6 weight	0	0 = weight 1 1 = weight 2

The hardware RESET value of ARB_BW_CTL is 0, resulting in a weight of 1 for all requests. Note that each media processor application needs to carefully review its arbiter settings. The default value is MOST LIKELY NOT a suitable value for high-bandwidth applications.

19.8 ANALYSIS OF BANDWIDTH

Bandwidth is allocated at every level relative to the weights of the devices.

The fraction of bandwidth for a device x is:

$$Fx = Wx / W\_Li$$

with Wx the weight of x and W_Li the sum of the weights of all devices at the level i where device x is connected.

The guaranteed minimum bandwidth for device x is:

$$Bx = Fx * B\_Li$$

with B_Li the total bandwidth available at level i.

Note that expected available bandwidth differs from guaranteed minimum bandwidth, depending on the application. If a particular device does not use all of its bandwidth, then other devices at the same level will get subsequently more bandwidth. If not all bandwidth is

used at a level then higher levels will get more bandwidth.

19.9 ANALYSIS OF LATENCY

The high weighting for MMIO, Icache and Dcache gives low latency to MMIO traffic and cache misses. This assures good CPU performance even at times when the highway is heavily loaded.

Maximum latency is closely related to minimum bandwidth.

The maximum latency Lx (i.e. waiting time till the acknowledgement of a request) for a device x is:

Lx = (ceil(W_Li/Wx)*Btotal/B_Li - 1) * T
(clock cycles)

with the symbols having the same meaning as above. B total is total bus bandwidth. T is the transfer time of one transaction (T=16 if main memory bandwidth is 4B/cycle)

This formula has some inaccuracies for the deeper levels of the hierarchy, but it is adequate for practical purposes. Note that expected latency is normally much lower than worst case latency, because very rarely many devices issue requests at exactly the same time.

The above analysis of latency does not consider the influence of SDRAM refresh and read/write gaps and bus turnaround cycles between SDRAM transactions. These effects can cause an additional increase in worst case latency. For instance, a continuous worst case sequence of read-write-read-write transactions requires has an average transaction time of 20 cycles in stead of 16 cycles (if peak main memory bandwidth is 4B/cycle) due to the read-write gaps. This increases worst-case latency by at most 25%.

The above does not include 19 cycles SDRAM refresh time. SDRAM refresh occurs once per 16 µSec. We only include this overhead once - only units with latency of greater than 1.6 µSec (1600 cycles at 100 MHz) can see the latency more than once.

For any application, the arbiter settings need to be chosen such that buffers of real-time peripherals in use in the application do not over/underflow. This is done by choosing a setting that provides a worst case latency that meets the needs of the unit, given its operational mode. All real-time units have a special exception notification flag that is raised if an overflow/underflow occurs during actual operation.

19.10 WHEN TO USE BANDWIDTH VERSUS LATENCY

On the highway, each request results in a transfer of 64 bytes in length. Different peripherals have different strategies and buffer methods. The ultimate reference is the section on latency in each peripheral chapter.

Latency allocation is required for units that have continuous streams as input or output and that have internal buffers of the same order of magnitude as the highway transfer size. Units with these properties are: Video In, Video Out, Audio In and Audio Out.

For units that have to meet a certain throughput only, or for units that have internal buffers that are an order of magnitude larger than 64 bytes, bandwidth allocation suffices. The ICP is an example of a unit with throughput only requirements. The PCI block mover is typically used to refill large software managed buffers, and hence can be dealt with on a bandwidth allocation basis.

For the TM1000 DSPCPU, latency is of prime importance - CPU power reduces as average latency increases. The design of the arbiter guarantees that the DSPCPU gets all unused bus bandwidth with lowest possible latency. Optimal operation is achieved if the arbiter is set in such a way that the DSPCPU has the best possible latency given the required latency and bandwidth of units active in the application.

In case of doubt, it is best to allocate based on latency.

Table 19-5. Recommended Allocation Method

Video In	allocate required latency
Video Out	allocate required latency
Audio In	allocate required latency
Audio Out	allocate required latency
ICP	allocate bandwidth
PCI	allocate bandwidth
VLD	allocate bandwidth
SSI	not applicable (slave only)
I^2C	not applicable (slave only)

19.11 EXAMPLE

(This Example is Under Construction, Not YET Complete!)

The following illustrates the issues of bandwidth and latency in a practical example.

A TM1000 with 100 MHz SDRAM connected across a 32 bit bus has a 400 MB/s main memory bandwidth and T=16 cycles. We are assuming that T=16 is adequate, and that we do NOT need to take the worst case 20 cycles for continuous read-write-read-write patterns into account. We do take the 19 cycle SDRAM refresh possibility (1 only per request) into account.

On this TM1000, we run a MPEG-2 video and audio playback application. The software decoded YUV 4:2:0 video images are sent out across Video Out. No ICP scaling is used. The software decoded audio is sent across the Audio Out unit. For simplicity, we do NOT use the optional priority raising mechanism in this example.

In Table 19-6 the latency and bandwidth requirements are summarized.

Table 19-6. Requirements for MPEG-2 Video/Audio Decoder

Unit	Operating Mode	Average Bandwidth (MB/s)	Required Latency (cycles)	
Video Out	YUV4:2:0 - no scaling output clock 27.0 MHz overlay disabled	27.0 MB/s	135	refer to Chapter 7, "Video Out" 3*lat+3*T+19 <= 128 out clocks 3*lat+3*T+19 <= 474
Audio Out	stereo, 16 bit/sample 44.1 kHz sample rate	0.18 MB/s	2265	refer to Section 9.8, "Highway Latency and HBE."
VLD	25 MB/sec peak for intra frames	15 MB/sec	n/a	refer to Chapter 14, "VLD Register Interface"
PCI DMA	DMA bitstream from host memory	1.5 MB/s	n/a	we assume a large buffer in SDRAM so that short term latency is not a key concern

An example valid setting is:

L1: CPU weight 3,
L2 weight 1 -> L2 bandwidth=100 MB/sec

L2: VO weight 1,
L3 weight 1 -> L3 bandwidth= 50 MB/sec,
VO latency=112 cycles

L3: ICP weight 1,
L4 weight 5 -> L4 bandwidth= 50 MB/sec
(since ICP is not active)

L4: VI weight 1,
L5 weight 5-> L5 bandwidth=50 MB/sec
(VI not active)

L5: PCI weight 1,
L6 weight 2 -> L6 bandwidth=33 MB/sec
(PCI may temporarily use all BW)

L6: VLD weight (fixed) 2,
AO weight 1 -> VLD bandwidth 22 MB/sec,
AO latency=582 cycles.

The CPU is guaranteed a latency of 32 cycles (plus 19 cycles for SDRAM refresh). It will experience a much lower average latency.

Power Management Chapter 20

by Eino Jacobs

20.1 OVERVIEW

TM1000 supports power management. It has a power down mode in which most clocks on the chip are shut down and the SDRAM main memory is brought into low-power self-refresh mode.

20.2 ENTERING AND EXITING POWER DOWN MODE

Power management is software controlled and is initiated by writing to the MMIO register POWER_DOWN. During execution of this MMIO operation the system is powered down without completing the MMIO operation. Only when the system wakes up from power down mode, the MMIO operation is completed. This means that during the execution of a program on the DSPCPU the moment of power down is defined exactly: any instruction before the instruction that contains the MMIO operation is completed before entering power down mode. The instruction containing the MMIO operation and all subsequent instructions are completed after wake up from power down mode.

Wake-up from power down mode is effected by receiving an interrupt (any interrupt) that passes the acceptance criteria of the interrupt controller.

There is also wake-up from power-down if a peripheral unit asserts a memory request signal on the highway.

During power down mode the whole chip is powered down, except the PLLs, the interrupt logic, the timers, the wake-up logic in the MMI and any logic in the peripheral units and PCI bus interface that is not participating in the power down.

20.3 POWER DOWN OF PERIPHERALS

The peripherals participate in global power down. This can be a programmable option for selected peripherals. These selected peripherals have a programmable MMIO control bit, the SLEEPLESS bit, that can be used to prevent it from participating in the global power down mode. By default every peripheral unit must participate in power down.

The following peripherals have the SLEEPLESS bit: video-in, video-out, audio-in, audio-out, SSI, JTAG.

The following peripherals do not have the SLEEPLESS bit and always participate in powerdown: VLD, boot/I2C and ICP.

The following peripherals do not participate in global powerdown, although they must still power themselves down when they are inactive: VIC, PCI.

When a peripheral does not participate in global power down, it can still do regular main memory traffic. Every time a peripheral unit asserts the highway request signal, the MMI will initiate a wake-up sequence. The CPU must execute software that initiates a new power down of the system. This software can be the wait-loop of the RTOS.

Programmer's note: Since the system is waked up each time there is a transaction on the highway, it may be interresting to make a software loop that does the activation of the POWER_DOWN mode. Then the activation is conditional, and most of the time, done using a global variable usually set by a handler. It becomes then mandatory to be sure that there are no interruptible jumps between the time the value of the global variable is fetched and compared by the DSPCU and the time the conditional write to the MMIO is performed (it is the classical semaphore or test and set issue). Thus it is recommended to use a separate function with the address of the variable as a parameter and this function needs to be compiled specifically without interruptible jumps.

The wake-up from power down mode takes approximately 20 SDRAM clock cycles. This amount of time is added to the worst case latency for memory requests compared to the situation when the system is not in power down mode.

20.4 DETAILED SEQUENCE OF EVENTS

The sequence of events to power down TM1000 is as follows:

- Issue a MMIO write to the POWER_DOWN register
- The main memory interface waits till the completion of the current main memory transfer, if there is one still busy.
- The MMI brings SDRAM into the self refresh state, goes into a wait state and asserts the global signal global_power_down.
- All units that participate in the power down, respond to the global_power_down signal by disabling their clocks.
- Only the PLL, interrupt controller, timers, wake-up logic, the PCI bus interface and any peripherals, that

have their SLEEPLESS bit control bit set, continue to be clocked. Also the SDRAM clock continues.

- An interrupt is detected by the interrupt controller or a unit that didn't participate in the power down requests a memory transfer.
- The MMI deasserts the global_power_down signal, activating all blocks on the chip.
- The MMI recovers SDRAM from self-refresh.
- The MMI causes completion of the MMIO operation that initiated the power down sequence.
- When software takes an interruptible branch operation, the interrupt that caused the wake_up will be serviced (if the wake-up was initiated by an interrupt).

20.5 MMIO REGISTER POWER_DOWN

The register POWER_DOWN has an offset 0x100108 in the MMIO aperture.

The register POWER_DOWN is without content. Writing to this register has the side-effect to power down the chip. Reading from this register returns an undefined value and has no side-effect.

DSPCPU Operations Appendix A

by Gert Slavenburg, Marcel Janssens

A.1 ALPHABETIC OPERATION LIST

The following table lists the complete operation set of TM1000's DSPCPU. Note that this is not an instruction list; a DSPCPU instruction contains from one to five of these operations.

A.2 OPERATION LIST BY FUNCTION

Allocate a cache block pseudo-op for allocd(0)

alloc

SYNTAX

```
[ IF rguard ] alloc(d) rsrc1
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + 0) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocr allocx

DESCRIPTION

The alloc operation is a pseudo operation transformed by the scheduler into an allocd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The alloc operation allocate a cache block with the address computed from **[(rsrc1 + 0) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsiblity of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The alloc operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the alloc operation. If the LSB of rguard is 1, alloc operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`alloc r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 alloc r10`	since guard is false, alloc operation is not executed
r10 = 0xac0f, r11 = 1, cache_block_size = 0x40	`IF r11 alloc r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocd

Allocate a cache block with displacement

SYNTAX

```
[ IF rguard ] allocd(d) rsrc1
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + d) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	7 bits
Modifier range	-255..252 by 4
Latency	-
Issue slots	5

SEE ALSO

allocr allocx

DESCRIPTION

The allocd operation allocate a cache block with the address computed from **[(rsrc1 + d) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsiblity of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocd operation. If the LSB of r*guard* is 1, allocd operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`allocd(0x32) r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 allocd(0x32) r10`	since guard is false, allocd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 allocd(0x4) r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

Allocate a cache block with index

allocr

SYNTAX

```
[ IF rguard ] allocr rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + rsrc2) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	214
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

`allocd allocx`

DESCRIPTION

The allocr operation allocate a cache block with the address computed from **[(rsrc1 + rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsiblity of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocr operation. If the LSB of r*guard* is 1, allocr operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0x32 cache_block_size = 0x40	`allocr r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0x32, cache_block_size = 0x40	`IF r11 allocr r10 r12`	since guard is false, allocr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocr r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocx

Allocate a cache block with scaled index

SYNTAX

```
[ IF rguard ] allocx rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache blockwith [(rsrc1 + 4 x rsrc2) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	215
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocr

DESCRIPTION

The allocx operation allocate a cache block with the address computed from **[(rsrc1 + 4 x rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsiblity of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocx` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocx operation. If the LSB of r*guard* is 1, allocx operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`allocx r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 allocx r10 r12`	since guard is false, allocx operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocx r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

Arithmetic shift left

asl

SYNTAX

```
[ IF rguard ] asl rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  n ← r*src2*<4:0>
  r*dest*<31:n> ← r*src1*<31–n:0>
  r*dest*<n–1:0> ← 0
}

ATTRIBUTES

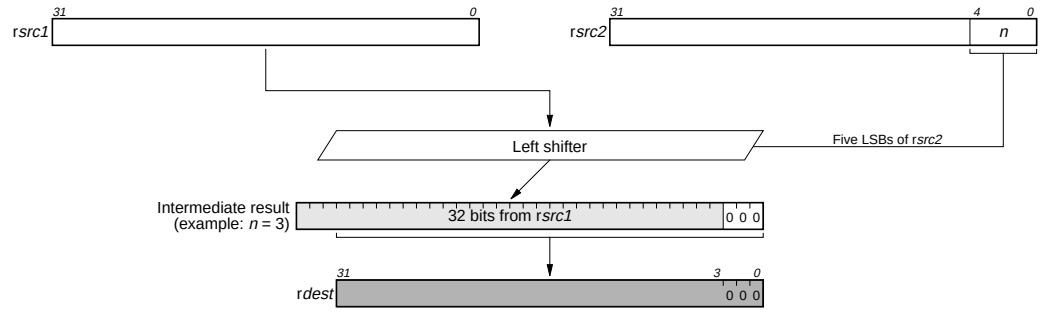
Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asli asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asl` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specify an unsigned shift amount, and r*dest* is set to r*src1* arithmetically shifted left by this amount. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `asl` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`asl r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 asl r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 asl r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`asl r70 r40 → r120`	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	`asl r80 r50 → r125`	r125 ← 0x80000000 (r50 is effectively equal to 0x1e)

asli

Arithmetic shift left immediate

SYNTAX

```
[ IF rguard ] asli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<31:*n*> ← r*src1*<31–*n*:0>
r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

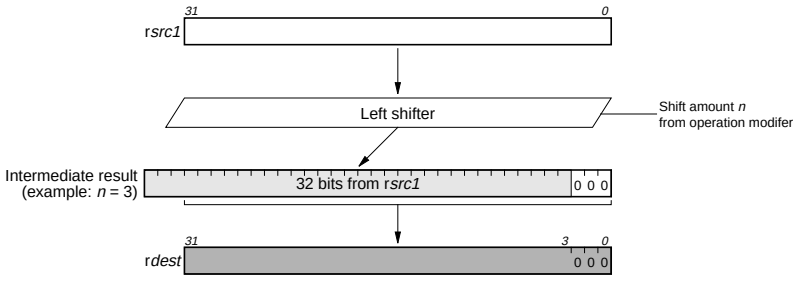
Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* arithmetically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `asli` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`asli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 asli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 asli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`asli(2) r70 → r120`	r120 ← 0xfffffff0
r80 = 0xe	`asli(30) r80 → r125`	r125 ← 0x80000000

Arithmetic shift right

asr

SYNTAX

```
[ IF rguard ] asr rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  n ← r*src2*<4:0>
  r*dest*<31:31–n> ← r*src1*<31>
  r*dest*<30–n:0> ← r*src1*<30:n>
}

ATTRIBUTES

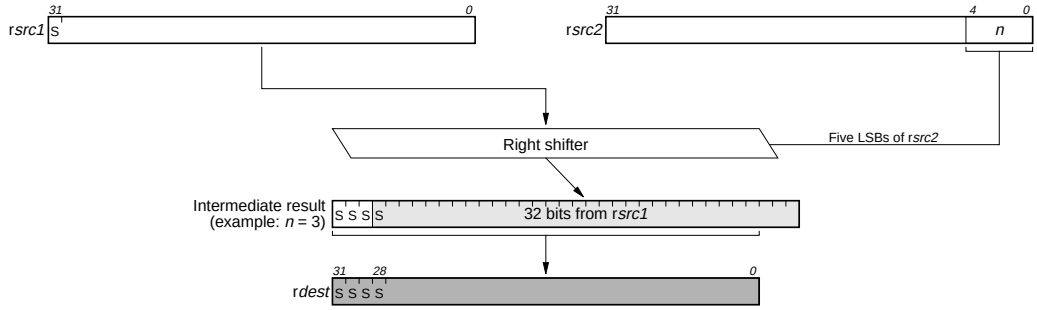
Function unit	shifter
Operation code	18
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asr` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specifies an unsigned shift amount, and r*src1* is arithmetically shifted right by this amount. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The `asr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	`asr r30 r20 → r50`	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	`asr r30 r42 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	`IF r10 asr r30 r44 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	`IF r20 asr r30 r44 → r80`	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	`asr r40 r44 → r90`	r90 ← 0xf8003000
r30 = 0x7008000f, r45 = 0x1f	`asr r30 r45 → r100`	r100 ← 0x00000000

asri

Arithmetic shift right by immediate amount

SYNTAX

```
[ IF rguard ] asri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:31–*n*> ← r*src1*<31>
    r*dest*<30–*n*:0> ← r*src1*<31:*n*>
}

ATTRIBUTES

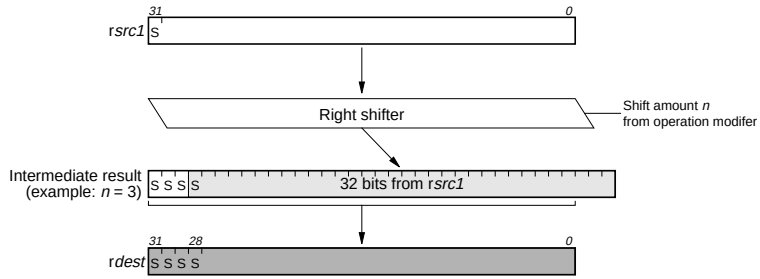
Function unit	shifter
Operation code	10
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* arithmetically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The `asri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`asri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`asri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 asri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 asri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`asri(4) r40 → r90`	r90 ← 0xf8003000
r30 = 0x7008000f	`asri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`asri(31) r40 → r110`	r110 ← 0xffffffff

Bitwise logical AND

bitand

SYNTAX

```
[ IF rguard ] bitand rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* & r*src2*

ATTRIBUTES

Function unit	alu
Operation code	16
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitor bitxor bitandinv

DESCRIPTION

The `bitand` operation computes the bitwise, logical AND of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitand` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitand r30 r40 → r90`	r90 ← 0xf3100000
r10 = 0, r50 = 0x88888888	`IF r10 bitand r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitand r30 r50 → r100`	r100 ← 0x80008888
r60 = 0x11119999, r50 = 0x88888888	`bitand r60 r50 → r110`	r110 ← 0x00008888
r70 = 0x55555555, r30 = 0xf310ffff	`bitand r70 r30 → r120`	r120 ← 0x51105555

bitandinv

Bitwise logical AND NOT

SYNTAX

```
[ IF rguard ] bitandinv rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* & ~r*src2*

ATTRIBUTES

Function unit	alu
Operation code	49
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitor bitxor

DESCRIPTION

The `bitandinv` operation computes the bitwise, logical AND of the first argument, r*src1*, with the 1's complement of the second argument, r*src2*. The result is stored in the destination register, r*dest*.

The `bitandinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitandinv r30 r40 → r90`	r90 ← 0x0000ffff
r10 = 0, r50 = 0x88888888	`IF r10 bitandinv r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitandinv r30 r50 → r100`	r100 ← 0x73107777
r60 = 0x11119999, r50 = 0x88888888	`bitandinv r60 r50 → r110`	r110 ← 0x11111111
r70 = 0x55555555, r30 = 0xf310ffff	`bitandinv r70 r30 → r120`	r120 ← 0x04450000

Bitwise logical NOT

bitinv

SYNTAX

```
[ IF rguard ] bitinv rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ~r*src1*

ATTRIBUTES

Function unit	alu
Operation code	50
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitor bitxor

DESCRIPTION

The `bitinv` operation computes the bitwise, logical NOT of the argument r*src1* and writes the result into r*dest*.

The `bitinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff	`bitinv r30 → r60`	r60 ← 0x0cef0000
r10 = 0, r40 = 0xffff0000	`IF r10 bitinv r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xffff0000	`IF r20 bitinv r40 → r100`	r100 ← 0x0000ffff
r50 = 0x88888888	`bitinv r50 → r110`	r110 ← 0x77777777

bitor

Bitwise logical OR

SYNTAX

```
[ IF rguard ] bitor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* | r*src2*

ATTRIBUTES

Function unit	alu
Operation code	17
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitinv bitxor

DESCRIPTION

The `bitor` operation computes the bitwise, logical OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitor r30 r40 → r90`	r90 ← 0xffffffff
r10 = 0, r50 = 0x88888888	`IF r10 bitor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitor r30 r50 → r100`	r100 ← 0xfb98ffff
r60 = 0x11119999, r50 = 0x88888888	`bitor r60 r50 → r110`	r110 ← 0x99999999
r70 = 0x55555555, r30 = 0xf310ffff	`bitor r70 r30 → r120`	r120 ← 0xf755ffff

Bitwise logical exclusive-OR

bitxor

SYNTAX

```
[ IF rguard ] bitxor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* ⊕ r*src2*

ATTRIBUTES

Function unit	alu
Operation code	48
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitinv bitor

DESCRIPTION

The `bitxor` operation computes the bitwise, logical exclusive-OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitxor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitxor r30 r40 → r90`	r90 ← 0x0ceffff
r10 = 0, r50 = 0x88888888	`IF r10 bitxor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitxor r30 r50 → r100`	r100 ← 0x7b987777
r60 = 0x11119999, r50 = 0x88888888	`bitxor r60 r50 → r110`	r110 ← 0x99991111
r70 = 0x55555555, r30 = 0xf310ffff	`bitxor r70 r30 → r120`	r120 ← 0xa645aaaa

borrow

Compute borrow bit from unsigned subtract

pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] borrow rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < rsrc2 then
      rdest ← 0
   else
      rdest ← 1
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ugtr carry

DESCRIPTION

The `borrow` operation is a pseudo operation transformed by the scheduler into an `ugtr` with reversed arguments. (Note: pseudo operations cannot be used in assembly source files.)

The `borrow` operation computes the unsigned difference of the first and second arguments, r*src1*–r*src2*. If the difference generates a borrow (if r*src2* > r*src1*), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `borrow` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`borrow r70 r30 → r80`	r80 ← 1
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 borrow r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 borrow r70 r30 → r100`	r100 ← 1
r60 = 4, r30 = 0xfffffffc	`borrow r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`borrow r30 r30 → r120`	r120 ← 0

Compute carry bit from unsigned add

carry

SYNTAX

```
[ IF rguard ] carry rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (r*src1*+r*src2*) < 2^{32} **then**
    r*dest* ← 0
  else
    r*dest* ← 1
}

ATTRIBUTES

Function unit	alu
Operation code	45
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

borrow

DESCRIPTION

The `carry` operation computes the unsigned sum of the first and second arguments, r*src1*+r*src2*. If the sum generates a carry (if the sum is greater than 2^{32}-1), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `carry` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`carry r70 r30 → r80`	r80 ← 0
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 carry r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 carry r70 r30 → r100`	r100 ← 0
r60 = 4, r30 = 0xfffffffc	`carry r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`carry r30 r30 → r120`	r120 ← 1

curcycles

Read current clock cycle counter, least-significant word

SYNTAX

```
[ IF rguard ] curcycles → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	162
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

cycles hicycles writepcsw

DESCRIPTION

Refer to Section 3.1.5, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `curcycles` operation copies the current low 32 bits of the master Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*.. The master CCCOUNT increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `curcycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`curcycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 curcycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 curcycles → r100`	r100 ← 0x12345678

Read clock cycle counter, least-significant word

cycles

SYNTAX

```
[ IF rguard ] cycles → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	154
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

hicycles curcycles writepcsw

DESCRIPTION

Refer to Section 3.1.5, "CCCOUNT—Clock Cycle Counter" for a description of the CCCOUNT operation. The `cycles` operation copies the low 32 bits of the slave register of Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if `cycles` and `hicycles` are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `cycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`cycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 cycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 cycles → r100`	r100 ← 0x12345678

dcb

Data cache copy back

SYNTAX

```
[ IF rguard ] dcb(d) rsrc1
```

FUNCTION

```
if rguard then {
   addr ← rsrc1 + d
   if dcache_valid_addr(addr) && dcache_dirty_addr(addr) then {
      dcache_copyback_addr(addr)
      dcache_reset_dirty_addr(addr)
   }
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	205
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

dinvalid

DESCRIPTION

The dcb operation causes a block in the data cache to be copied back to main memory if the block is marked dirty and valid, and the block's dirty bit is reset. The target block of dcb is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

A valid copy of the target block remains in the cache. Stall cycles are taken as necessary to complete the copy-back operation. If the target block is not dirty or if the block is not in the cache, dcb has no effect and no stall cycles are taken.

dcb has no effect on blocks that are in the non-cacheable SDRAM aperture. dcb does not change the replacement status of data-cache blocks.

dcb ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The dcb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`dcb(0) r30`	
r10 = 0	`IF r10 dcb(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dcb(8) r50`	

Invalidate data cache block

dinvalid

SYNTAX

```
[ IF rguard ] dinvalid(d) rsrc1
```

FUNCTION

```
if rguard then {
    addr ← rsrc1 + d
    if dcache_valid_addr(addr) then {
        dcache_reset_valid_addr(addr)
        dcache_reset_dirty_addr(addr)
    }
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	206
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

dcb

DESCRIPTION

The `dinvalid` operation resets the valid and dirty bit of a block in the data cache. Regardless of the block's dirty bit, the block is not written back to main memory. The target block of `dinavlid` is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

Stall cycles are taken as necessary to complete the invalidate operation. If the target block is not in the cache, `dinvalid` has no effect and no stall cycles are taken.

`dinvalid` has no effect on blocks that are in the non-cacheable SDRAM aperture. `dinvalid` does clear the valid bits of locked blocks. `dinvalid` does not change the replacement status of data-cache blocks.

`dinvalid` ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The `dinvalid` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`dinvalid(0) r30`	
r10 = 0	`IF r10 dinvalid(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dinvalid(8) r50`	

dspiabs

Clipped signed absolute value

pseudo-op for h_dspiabs

SYNTAX

```
[ IF rguard ] dspiabs rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= 0 then
      rdest ← rsrc1
   else if rsrc1 = 0x80000000 then
      rdest ← 0x7fffffff
   else
      rdest ← –rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs h_dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The dspiabs operation is a pseudo operation transformed by the scheduler into an h_dspiabs with a constant first argument zero and second argument equal to the dspiabs argument. (Note: pseudo operations cannot be used in assembly source files.)

The dspiabs operation computes the absolute value of rsrc1, clips the result into the range [2^{31}–1..0] (or [0x7fffffff..0]), and stores the clipped value into r*dest*. All values are signed integers.

The dspiabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	dspiabs r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	IF r10 dspiabs r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80000001	IF r20 dspiabs r40 → r100	r100 ← 0x7fffffff
r50 = 0x80000000	dspiabs r50 → r80	r80 ← 0x7fffffff
r90 = 0x7fffffff	dspiabs r90 → r110	r110 ← 0x7fffffff

Clipped signed add

dspiadd

SYNTAX

```
[ IF rguard ] dspiadd rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← sign_ext32to64(rsrc1) + sign_ext32to64(rsrc2)
    if temp < 0xffffffff80000000 then
        rdest ← 0x80000000
    else if temp > 0x000000007fffffff then
        rdest ← 0x7fffffff
    else
        rdest ← temp
}
```

ATTRIBUTES

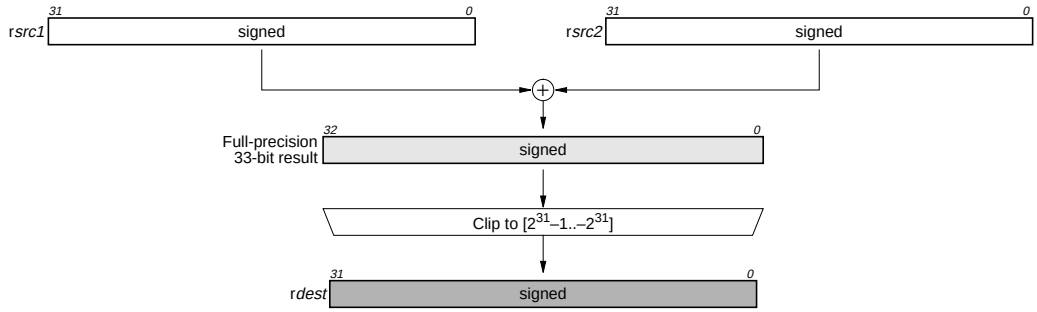
Function unit	dspalu
Operation code	66
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspimul dspisub
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspiadd operation computes the sum r*src1*+r*src2*, clips the result into the 32-bit signed range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The dspiadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	`dspiadd r30 r40 → r60`	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	`IF r10 dspiadd r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	`IF r20 dspiadd r30 r40 → r100`	r100 ← 0x12ff
r50 = 0x7fffffff, r90 = 1	`dspiadd r50 r90 → r110`	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 0xffffffff	`dspiadd r70 r80 → r120`	r120 ← 0x80000000

dspidualabs

Dual clipped absolute value of signed 16-bit halfwords

pseudo-op for h_dspidualabs

SYNTAX

```
[ IF rguard ] dspidualabs rsrc1 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

FUNCTION

```
if rguard then {
    temp1 ← sign_ext16to32(rsrc1<15:0>)
    temp2 ← sign_ext16to32(rsrc1<31:16>)
    if temp1 = 0xffff8000 then temp1 ← 0x7fff
    if temp2 = 0xffff8000 then temp2 ← 0x7fff
    if temp1 < 0 then temp1 ← –temp1
    if temp2 < 0 then temp2 ← –temp2
    rdest<31:16> ← temp2<15:0>
    rdest<15:0> ← temp1<15:0>
}
```

SEE ALSO

h_dspidualabs dspiabs dspidualadd dspidualmul dspidualsub

DESCRIPTION

The dspidualabs operation is a pseudo operation transformed by the scheduler into an h_dspidualabs with a constant zero as first argument and the dspidualabs argument as second argument. (Note: pseudo operations cannot be used in assembly source files.)

The dspidualabs operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of r*src1*. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.

The dspidualabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	dspidualabs r30 → r60	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	IF r10 dspidualabs r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80008001	IF r20 dspidualabs r40 → r100	r100 ← 0x7fff7fff
r50 = 0x0032ffff	dspidualabs r50 → r80	r80 ← 0x00320001
r90 = 0x7fffffff	dspidualabs r90 → r110	r110 ← 0x7fff0001

Dual clipped add of signed 16-bit halfwords

dspidualadd

SYNTAX

```
[ IF rguard ] dspidualadd rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	70
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

FUNCTION

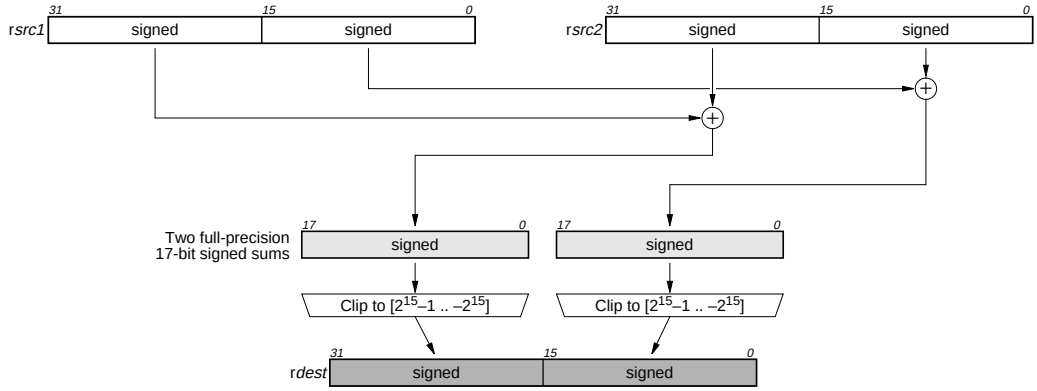
```
if rguard then {
    temp1 ← sign_ext16to32(rsrc1<15:0>) + sign_ext16to32(rsrc2<15:0>)
    temp2 ← sign_ext16to32(rsrc1<31:16>) + sign_ext16to32(rsrc2<31:16>)
    if temp1 < 0xffff8000 then temp1 ← 0x8000
    if temp2 < 0xffff8000 then temp2 ← 0x8000
    if temp1 > 0x7fff then temp1 ← 0x7fff
    if temp2 > 0x7fff then temp2 ← 0x7fff
    rdest<31:16> ← temp2<15:0>
    rdest<15:0> ← temp1<15:0>
}
```

SEE ALSO

dspidualabs dspidualmul dspidualsub dspiabs

DESCRIPTION

As shown below, the dspidualadd operation computes two 16-bit clipped, signed sums separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both sums are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The dspidualadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	`dspidualadd r30 r40 → r60`	r60 ← 0x12350034
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	`IF r10 dspidualadd r30 r40 → r70`	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	`IF r20 dspidualadd r30 r40 → r100`	r100 ← 0x12350034
r50 = 0x80000001, r80 = 0xffff7fff	`dspidualadd r50 r80 → r90`	r90 ← 0x80007fff
r110 = 0x00017fff, r120 = 0x7fff7fff	`dspidualadd r110 r120 → r125`	r125 ← 0x7fff7fff

dspidualmul Dual clipped multiply of signed 16-bit halfwords

SYNTAX

```
[ IF rguard ] dspidualmul rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	95
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

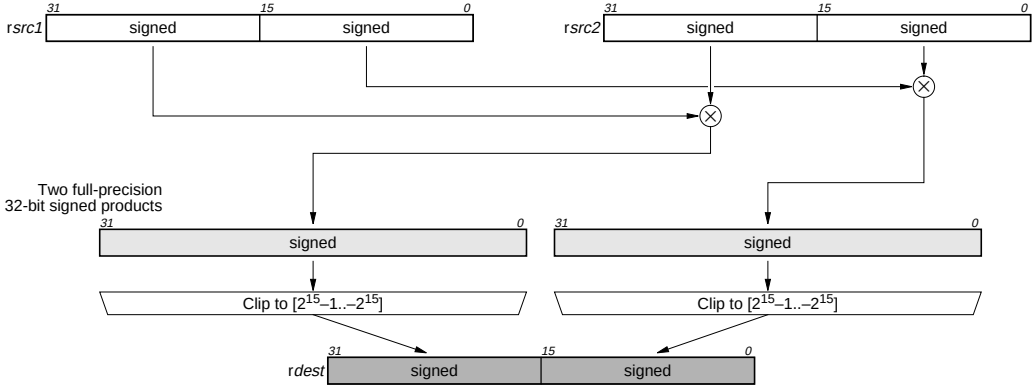
if r*guard* **then {**
  temp1 ← sign_ext16to32(r*src1*<15:0>) × sign_ext16to32(r*src2*<15:0>)
  temp2 ← sign_ext16to32(r*src1*<31:16>) × sign_ext16to32(r*src2*<31:16>)
  if temp1 < 0xffff8000 **then** temp1 ← 0x8000
  if temp2 < 0xffff8000 **then** temp2 ← 0x8000
  if temp1 > 0x7fff **then** temp1 ← 0x7fff
  if temp2 > 0x7fff **then** temp2 ← 0x7fff
  r*dest*<31:16> ← temp2<15:0>
  r*dest*<15:0> ← temp1<15:0>
}

SEE ALSO

dspidualabs dspidualadd dspidualsub dspiabs

DESCRIPTION

As shown below, the `dspidualmul` operation computes two 16-bit clipped, signed products separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both products are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The `dspidualmul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0020010, r40 = 0x00030020	`dspidualmul r30 r40 → r60`	r60 ← 0x00060200
r10 = 0, r30 = 0x0020010, r40 = 0x00030020	`IF r10 dspidualmul r30 r40 → r70`	no change, since guard is false
r20 = 1, r30 = 0x0020010, r40 = 0x00030020	`IF r20 dspidualmul r30 r40 → r100`	r100 ← 0x00060200
r50 = 0x80000002, r80 = 0x00024000	`dspidualmul r50 r80 → r90`	r90 ← 0x80007fff
r110 = 0x08000003, r120 = 0x00108001	`dspidualmul r110 r120 → r125`	r125 ← 0x7fff8000

Dual clipped subtract of signed 16-bit halfwords

dspidualsub

SYNTAX

```
[ IF rguard ] dspidualsub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
  temp1 ← sign_ext16to32(rsrc1<15:0>) – sign_ext16to32(rsrc2<15:0>)
  temp2 ← sign_ext16to32(rsrc1<31:16>) – sign_ext16to32(rsrc2<31:16>)
  if temp1 < 0xffff8000 then temp1 ← 0x8000
  if temp2 < 0xffff8000 then temp2 ← 0x8000
  if temp1 > 0x7fff then temp1 ← 0x7fff
  if temp2 > 0x7fff then temp2 ← 0x7fff
  rdest<31:16> ← temp2<15:0>
  rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

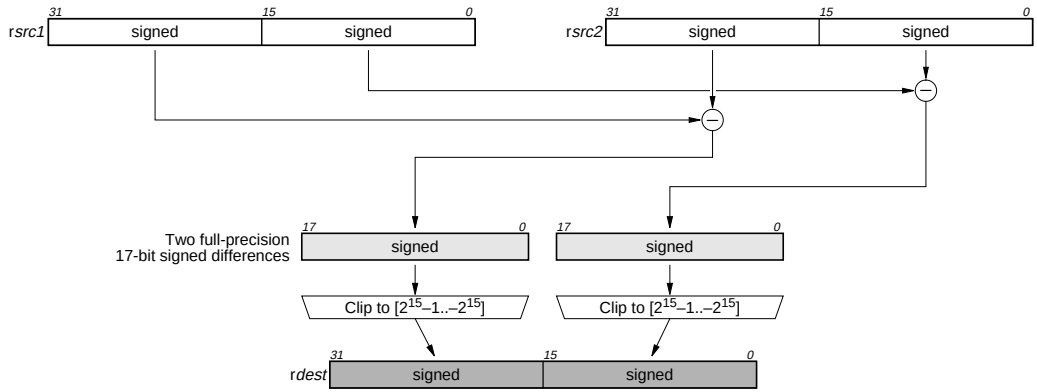
Function unit	dspalu
Operation code	71
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspidualadd dspidualmul dspiabs

DESCRIPTION

As shown below, the dspidualsub operation computes two 16-bit clipped, signed differences separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both differences are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The dspidualsub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	dspidualsub r30 r40 → r60	r60 ← 0x12330030
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	IF r10 dspidualsub r30 r40 → r70	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	IF r20 dspidualsub r30 r40 → r100	r100 ← 0x12330030
r50 = 0x80000001, r80 = 0x00018001	dspidualsub r50 r80 → r90	r90 ← 0x80007fff
r110 = 0x00018001, r120 = 0x80010002	dspidualsub r110 r120 → r125	r125 ← 0x7fff8000

dspimul

Clipped signed multiply

SYNTAX

```
[ IF rguard ] dspimul rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   temp ← sign_ext32to64(rsrc1) × sign_ext32to64(rsrc2)
   if temp < 0xffffffff80000000 then
      rdest ← 0x80000000
   else if temp > 0x000000007fffffff then
      rdest ← 0x7fffffff
   else
      rdest ← temp<31:0>
}
```

ATTRIBUTES

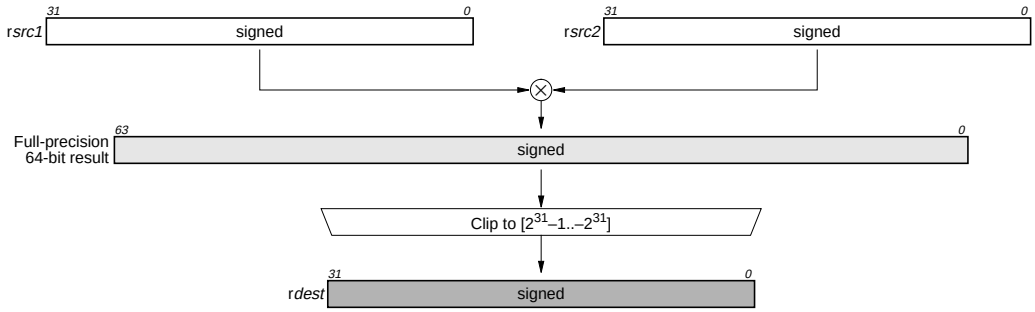
Function unit	ifmul
Operation code	141
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspiabs dspiadd dspisub dspuadd dspumul dspusub

DESCRIPTION

As shown below, the `dspimul` operation computes the product r*src1*×r*src2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The `dspimul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	`dspimul r30 r40 → r60`	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	`IF r10 dspimul r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	`IF r20 dspimul r30 r40 → r100`	r100 ← 0x200
r50 = 0x40000000, r90 = 2	`dspimul r50 r90 → r110`	r110 ← 0x7fffffff
r80 = 0xffffffff	`dspimul r80 r80 → r120`	r120 ← 0x1
r70 = 0x80000000, r90 = 2	`dspimul r70 r90 → r120`	r120 ← 0x80000000

Clipped signed subtract

dspisub

SYNTAX

```
[ IF rguard ] dspisub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← sign_ext32to64(rsrc1) – sign_ext32to64(rsrc2)
    if temp < 0xffffffff80000000 then
        rdest ← 0x80000000
    else if temp > 0x000000007fffffff then
        rdest ← 0x7fffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

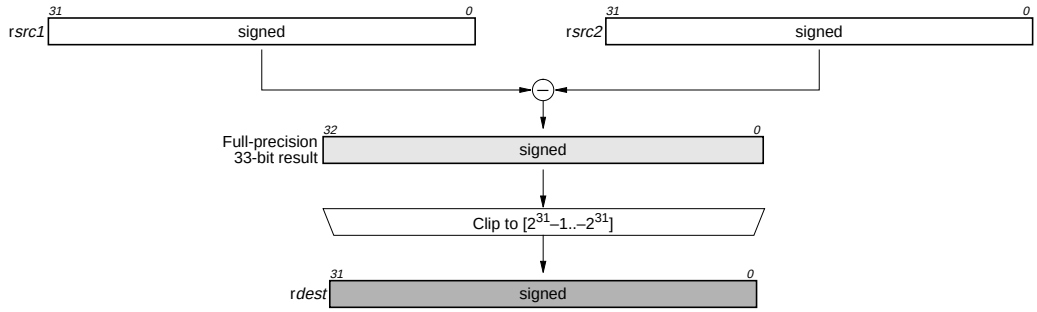
Function unit	dspalu
Operation code	68
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspisub operation computes the difference r*src1*–r*src2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The dspisub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspisub r30 r40 → r60	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspisub r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspisub r30 r40 → r100	r100 ← 0x1101
r50 = 0x7fffffff, r90 = 0xffffffff	dspisub r50 r90 → r110	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 1	dspisub r70 r80 → r120	r120 ← 0x80000000

dspuadd

Clipped unsigned add

SYNTAX

```
[ IF rguard ] dspuadd rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) + zero_ext32to64(rsrc2)
    if (unsigned)temp > 0x00000000ffffffff then
        rdest ← 0xffffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

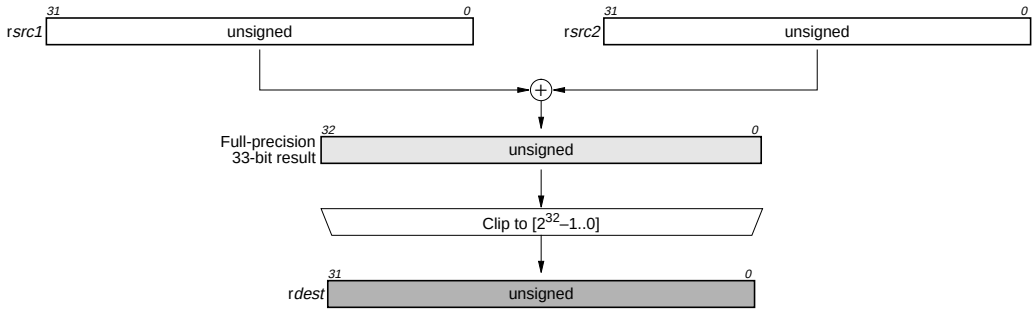
Function unit	dspalu
Operation code	67
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul dspisub dspumul dspusub

DESCRIPTION

As shown below, the dspuadd operation computes unsigned sum r*src1*+r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspuadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspuadd r30 r40 → r60	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspuadd r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspuadd r30 r40 → r100	r100 ← 0x12ff
r50 = 0xffffffff, r90 = 1	dspuadd r50 r90 → r110	r110 ← 0xffffffff
r70 = 0x80000001, r80 = 0x7fffffff	dspuadd r70 r80 → r120	r120 ← 0xffffffff

Clipped unsigned multiply

dspumul

SYNTAX

```
[ IF rguard ] dspumul rsrc1 rsrc2 → rdest
```

OPERATION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) × zero_ext32to64(rsrc2)
    if (unsigned)temp > 0x00000000ffffffff then
        rdest ← 0xffffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

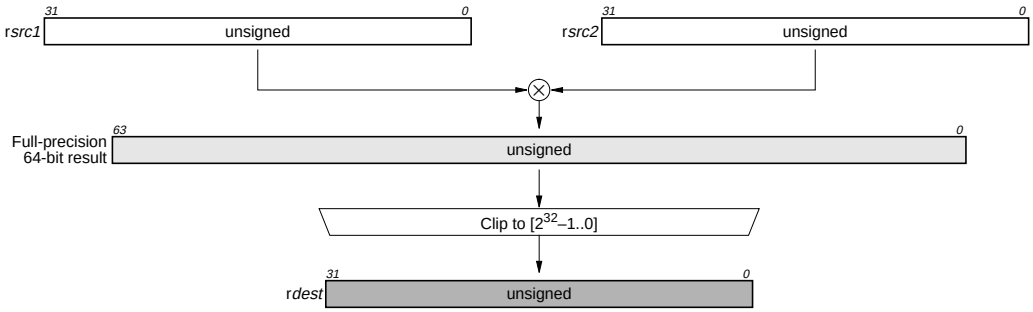
Function unit	ifmul
Operation code	142
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspiabs dspiadd dspisub
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspumul operation computes unsigned product r*src1*×r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspumul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	`dspumul r30 r40 → r60`	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	`IF r10 dspumul r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	`IF r20 dspumul r30 r40 → r100`	r100 ← 0x200
r50 = 0x40000000, r90 = 2	`dspumul r50 r90 → r110`	r110 ← 0x80000000
r80 = 0xffffffff	`dspumul r80 r80 → r120`	r120 ← 0xffffffff
r70 = 0x80000000, r90 = 2	`dspumul r70 r90 → r120`	r120 ← 0xffffffff

dspuquadaddui Quad clipped add of unsigned/signed bytes

SYNTAX

```
[ IF rguard ] dspuquadaddui rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   for (i ← 0, m ← 31, n ← 24; i < 4; i ← i + 1, m ← m – 8, n ← n – 8) {
      temp ← zero_ext8to32(rsrc1<m:n>) + sign_ext8to32(rsrc2<m:n>)
      if temp < 0 then
         rdest<m:n> ← 0
      else if temp > 0xff then
         rdest<m:n> ← 0xff
      else rdest<m:n> ← temp<7:0>
   }
}
```

ATTRIBUTES

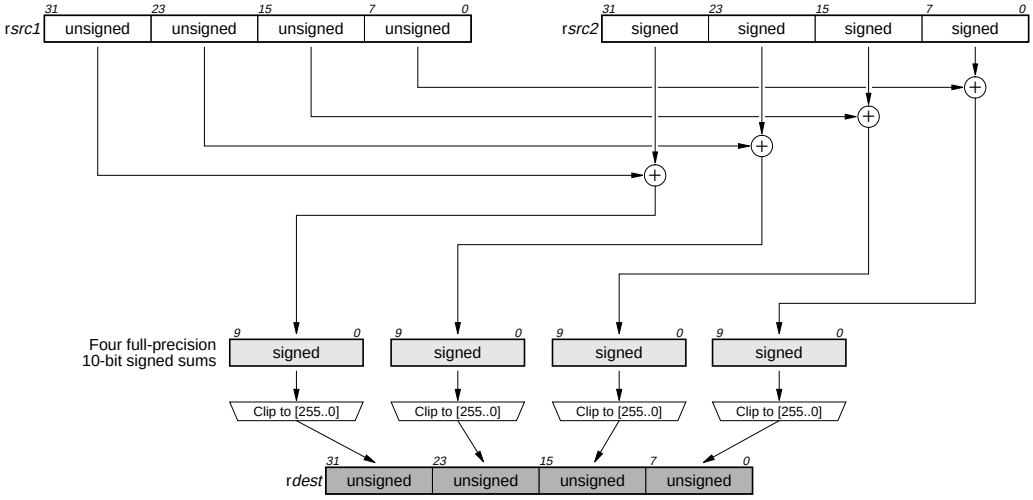
Function unit	dspalu
Operation code	78
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualadd

DESCRIPTION

As shown below, the dspuquadaddui operation computes four separate sums of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. The bytes in r*src1* are considered unsigned values; the bytes in r*src2* are considered signed. The four sums are clipped into the unsigned range [255..0] (or [0xff..0]); thus, the final byte sums are unsigned. All computations are performed without loss of precision.



The dspuquadaddui operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x02010001, r40 = 0xffffff01	dspuquadaddui r30 r40 → r50	r50 ← 0x01000002
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r10 dspuquadaddui r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r20 dspuquadaddui r60 r70 → r90	r90 ← 0xff38c800

Clipped unsigned subtract

dspusub

SYNTAX

```
[ IF rguard ] dspusub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) – zero_ext32to64(rsrc2)
    if (signed)temp < 0 then
        rdest ← 0
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

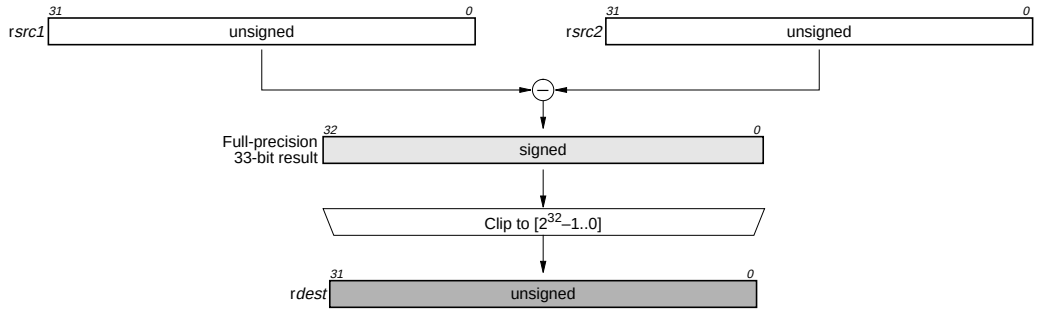
Function unit	dspalu
Operation code	69
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul
dspisub dspuadd dspumul

DESCRIPTION

As shown below, the dspusub operation computes unsigned difference r*src1*–r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspusub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	`dspusub r30 r40 → r60`	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	`IF r10 dspusub r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	`IF r20 dspusub r30 r40 → r100`	r100 ← 0x1101
r50 = 0, r90 = 1	`dspusub r50 r90 → r110`	r110 ← 0
r70 = 0x80000001, r80 = 0xffffffff	`dspusub r70 r80 → r120`	r120 ← 0

fabsval

Floating-point absolute value

SYNTAX

```
[ IF rguard ] fabsval rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 < 0 then
      rdest ← –(float)rsrc1
   else
      rdest ← (float)rsrc1
}
```

ATTRIBUTES

Function unit	falu
Operation code	115
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

iabs dspiabs dspidualabs fabsvalflags readpcsw writepcsw

DESCRIPTION

The `fabsval` operation computes the absolute value of the argument r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. If an argument is denormalized, zero is substituted for the argument before computing the absolute value, and the IFZ flag in the PCSW is set. If `fabsval` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fabsvalflags` operation computes the exception flags that would result from an individual `fabsval`.

The `fabsval` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsval r30 → r90`	r90 ← 0x40400000 (3.0)
r35 = 0xbf800000 (-1.0)	`fabsval r35 → r95`	r95 ← 0x3f800000 (1.0)
r40 = 0x00400000 (5.877471754e-39)	`fabsval r40 → r100`	r100 ← 0x0 (+0.0), IFZ set
r45 = 0xffffffff (QNaN)	`fabsval r45 → r105`	r105 ← 0xffffffff (QNaN)
r50 = 0xffbfffff (SNaN)	`fabsval r50 → r110`	r110 ← 0xffffffff (QNaN), INV set
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsval r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsval r55 → r120`	r120 ← 0x7f7fffff (3.402823466e+38)

IEEE status flags from floating-point absolute value

fabsvalflags

SYNTAX

```
[ IF rguard ] fabsvalflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags(abs_val((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	116
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

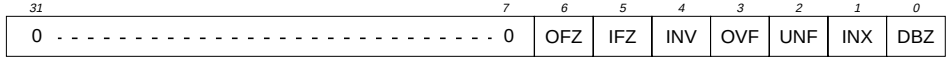
SEE ALSO

fabsval faddflags readpcsw

DESCRIPTION

The `fabsvalflags` operation computes the IEEE exceptions that would result from computing the absolute value of r*src1* and writes a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If r*src1* is denormalized, the IFZ bit in the result is set.

The `fabsvalflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsvalflags r30 → r90`	r90 ← 0x0
r35 = 0xbf800000 (-1.0)	`fabsvalflags r35 → r95`	r95 ← 0x0
r40 = 0x00400000 (5.877471754e-39)	`fabsvalflags r40 → r100`	r100 ← 0x20 (IFZ)
r45 = 0xffffffff (QNaN)	`fabsvalflags r45 → r105`	r105 ← 0x0
r50 = 0xffbfffff (SNaN)	`fabsvalflags r50 → r110`	r110 ← 0x10 (INV)
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsvalflags r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsvalflags r55 → r120`	r120 ← 0x0

fadd

Floating-point add

SYNTAX

```
[ IF rguard ] fadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* + (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	22
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`faddflags` `iadd` `dspiadd` `dspidualadd` `readpcsw` `writepcsw`

DESCRIPTION

The `fadd` operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the sum, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fadd` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `faddflags` operation computes the exception flags that would result from an individual `fadd`.

The `fadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fadd r60 r30 → r90`	r90 ← 0xc0000000 (–2.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fadd r40 r60 → r95`	r95 ← 0x00000000 (0.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fadd r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fadd r40 r80 → r110`	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fadd r40 r81 → r111`	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x80800000 (–1.175494351e-38)	`fadd r82 r83 → r112`	r112 ← 0x00000000 (0.0), OFZ, UNF, INX flags set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fadd r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	`fadd r70 r70 → r120`	r120 ← 0x7f800000 (+INF), OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	`fadd r80 r80 → r125`	r125 ← 0x01000000 (2.350988702e–38)

IEEE status flags from floating-point add

faddflags

SYNTAX

```
[ IF rguard ] faddflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* + (float)r*src2*)

ATTRIBUTES

Function unit	falu
Operation code	112
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

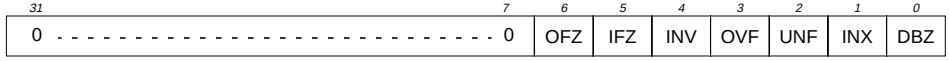
SEE ALSO

`fadd fsubflags readpcsw`

DESCRIPTION

The `faddflags` operation computes the IEEE exceptions that would result from computing the sum r*src1*+r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the sum, and the IFZ bit in the result is set. If the sum would be denormalized, the OFZ bit in the result is set.

The `faddflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r10 = 0x7f7fffff (3.402823466e+38), r20 = 0x3f800000 (1.0)	`faddflags r10 r20 → r60`	r60 ← 0x2 (INX)
r30 = 0, r10 = 0x7f7fffff (3.402823466e+38)	`IF r30 faddflags r10 r10 → r50`	no change, since guard is false
r40 = 1, r10 = 0x7f7fffff (3.402823466e+38)	`IF r40 faddflags r10 r10 → r70`	r70 ← 0xa (OVF INX)
r80 = 0x00a00000 (1.469367939e–38), r81 = 0x80800000 (–1.17549435e–38)	`faddflags r80 r81 → r100`	r100 ← 0x46 (OFZ UNF INX)
r95 = 0x7f800000 (+INF), r96 = 0xff800000 (–INF)	`faddflags r95 r96 → r105`	r105 ← 0x10 (INV)
r98 = 0x40400000 (3.0), r99 = 0x00400000 (5.877471754e–39)	`faddflags r98 r99 → r111`	r111 ← 0x20 (IFZ)

fdiv

Floating-point divide

SYNTAX

```
[ IF rguard ] fdiv rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* / (float)r*src2*

ATTRIBUTES

Function unit	ftough
Operation code	108
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fdivflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fdiv` operation computes the quotient r*src1*÷r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the quotient, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fdiv` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fdivflags` operation computes the exception flags that would result from an individual `fdiv`.

The `fdiv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fdiv r60 r30 → r90`	r90 ← 0xc0400000 (–3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fdiv r40 r60 → r95`	r95 ← 0xbf800000 (–1.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fdiv r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fdiv r40 r80 → r110`	r110 ← 0x7f400000 (2.552117754e38)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fdiv r40 r81 → r111`	r111 ← 0x7f800000 (+INF), IFZ, DBZ flags set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x80800000 (–1.175494351e–38)	`fdiv r82 r83 → r112`	r112 ← 0xbfc00000 (-1.5)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fdiv r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	`fdiv r70 r70 → r120`	r120 ← 0x3f800000 (1.0)
r80 = 0x00800000 (1.763241526e–38)	`fdiv r80 r80 → r125`	r125 ← 0x3f800000 (1.0)
r75 = 0x40400000 (3.0), r76 = 0x0 (0.0)	`fdiv r75 r76 → r126`	r126 ← 0x7f800000 (+INF), DBZ flag set

IEEE status flags from floating-point divide

fdivflags

SYNTAX

```
[ IF rguard ] fdivflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* / (float)r*src2*)

ATTRIBUTES

Function unit	ftough
Operation code	109
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

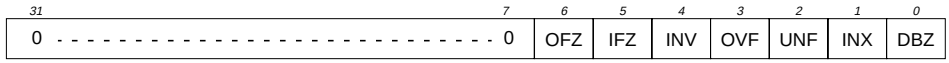
SEE ALSO

fdiv faddflags readpcsw

DESCRIPTION

The `fdivflags` operation computes the IEEE exceptions that would result from computing the quotient r*src1*÷r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the quotient, and the IFZ bit in the result is set. If the quotient would be denormalized, the OFZ bit in the result is set.

The `fdivflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x7f7fffff (3.402823466e+38), r40 = 0x3f800000 (1.0)	`fdivflags r30 r40 → r100`	r100 ← 0
r10 = 0, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r10 fdivflags r50 r60 → r110`	no change, since guard is false
r20 = 1, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r20 fdivflags r50 r60 → r111`	r111 ← 0xa (OVF INX)
r70 = 0x40400000 (3.0), r80 = 0x00400000 (5.877471754e–39)	`fdivflags r70 r80 → r112`	r112 ← 0x21 (IFZ DBZ)
r85 = 0x7f800000 (+INF), r86 = 0xff800000 (–INF)	`fdivflags r85 r86 → r113`	r113 ← 0x10 (INV)

feql

Floating-point compare equal

SYNTAX

```
[ IF rguard ] feql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 = (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	148
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ieql feqlflags fneq readpcsw writepcsw

DESCRIPTION

The `feql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `feql` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `feqlflags` operation computes the exception flags that would result from an individual `feql`.

The `feql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`feql r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`feql r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 feql r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 feql r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`feql r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`feql r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`feql r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`feql r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`feql r50 r50 → r127`	r127 ← 1

IEEE status flags from floating-point compare equal

feqlflags

SYNTAX

```
[ IF rguard ] feqlflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* = (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	149
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

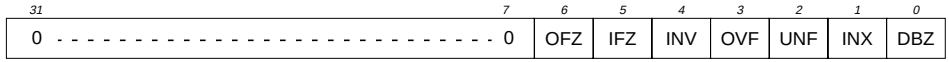
SEE ALSO

feql ieql fgtrflags readpcsw

DESCRIPTION

The `feqlflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `feqlflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`feqlflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`feqlflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 feqlflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 feqlflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`feqlflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`feqlflags r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`feqlflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`feqlflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`feqlflags r50 r50 → r127`	r127 ← 0

fgeq

Floating-point compare greater or equal

SYNTAX

```
[ IF rguard ] fgeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 >= (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igeq fgeqflags fgtr
readpcsw writepcsw

DESCRIPTION

The `fgeq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fgeq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fgeqflags` operation computes the exception flags that would result from an individual `fgeq`.

The `fgeq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeq r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fgeq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeq r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeq r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeq r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeq r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fgeq r50 r50 → r127`	r127 ← 1

IEEE status flags from floating-point compare greater or equal

fgeqflags

SYNTAX

```
[ IF rguard ] fgeqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* >= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

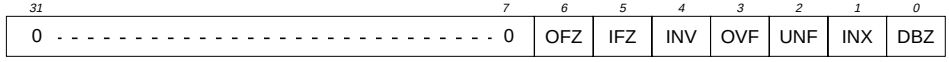
SEE ALSO

fgeq igeq fgtrflags
readpcsw

DESCRIPTION

The `fgeqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgeqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgeqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgeqflags r50 r50 → r127`	r127 ← 0

fgtr

Floating-point compare greater

SYNTAX

```
[ IF rguard ] fgtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 > (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igtr fgtrflags fgeq
readpcsw writepcsw

DESCRIPTION

The `fgtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fgtr` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fgtrflags` operation computes the exception flags that would result from an individual `fgtr`.

The `fgtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgtr r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fgtr r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgtr r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgtr r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgtr r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgtr r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgtr r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgtr r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fgtr r50 r50 → r127`	r127 ← 0

IEEE status flags from floating-point compare greater

fgtrflags

SYNTAX

```
[ IF rguard ] fgtrflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* > (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

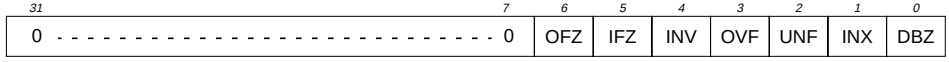
SEE ALSO

fgtr igtr fgeqflags readpcsw

DESCRIPTION

The `fgtrflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgtrflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgtrflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgtrflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgtrflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgtrflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgtrflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgtrflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgtrflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgtrflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgtrflags r50 r50 → r127`	r127 ← 0

fleq

Floating-point compare less-than or equal

pseudo-op for fgeq

SYNTAX

```
[ IF rguard ] fleq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 <= (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ileq fgeq fleqflags readpcsw writepcsw

DESCRIPTION

The `fleq` operation is a pseudo operation transformed by the scheduler into an `fgeq` with the arguments exchanged (`fleq`'s r*src1* is `fgeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fleq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fleqflags` operation computes the exception flags that would result from an individual `fleq`.

The `fleq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleq r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleq r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleq r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleq r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleq r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fleq r50 r50 → r127`	r127 ← 1

IEEE status flags from floating-point compare less-than or equal

fleqflags

pseudo-op for fgeqflags

SYNTAX

```
[ IF rguard ] fleqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* <= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

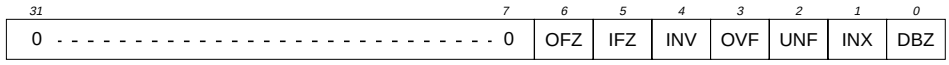
`fleq ileq fgeqflags readpcsw`

DESCRIPTION

The `fleqflags` operation is a pseudo operation transformed by the scheduler into an `fgeqflags` with the arguments exchanged (`fleqflags`'s r*src1* is `fgeqflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fleqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fleqflags r50 r50 → r127`	r127 ← 0

fles

Floating-point compare less-than

pseudo-op for fgtr

SYNTAX

```
[ IF rguard ] fles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 < (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

iles fgtr flesflags readpcsw writepcsw

DESCRIPTION

The `fles` operation is a pseudo operation transformed by the scheduler into an `fgtr` with the arguments exchanged (`fles`'s r*src1* is `fgtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fles` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fles` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `flesflags` operation computes the exception flags that would result from an individual `fles`.

The `fles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fles r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fles r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fles r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fles r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fles r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fles r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fles r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fles r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fles r50 r50 → r127`	r127 ← 0

IEEE status flags from floating-point compare less-than

flesflags

pseudo-op for fgtrflags

SYNTAX

`[ IF rguard ] flesflags rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* < (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

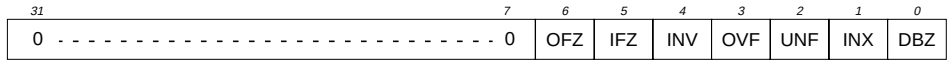
fles iles fleqflags readpcsw

DESCRIPTION

The `flesflags` operation is a pseudo operation transformed by the scheduler into an `fgtrflags` with the arguments exchanged (`flesflags`'s r*src1* is `fgtrflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `flesflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `flesflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`flesflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`flesflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 flesflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 flesflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`flesflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`flesflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`flesflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`flesflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`flesflags r50 r50 → r127`	r127 ← 0

fmul

Floating-point multiply

SYNTAX

```
[ IF rguard ] fmul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* × (float)r*src2*

ATTRIBUTES

Function unit	ifmul
Operation code	28
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

imul umul dspimul dspidualmul fmulflags readpcsw writepcsw

DESCRIPTION

The `fmul` operation computes the product r*src1*×r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the product, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fmul` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fmulflags` operation computes the exception flags that would result from an individual `fmul`.

The `fmul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmul r60 r30 → r90`	r90 ← 0xc0400000 (-3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmul r40 r60 → r95`	r95 ← 0xc1100000 (-9.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmul r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmul r40 r80 → r105`	r105 ← 0x1400000 (3.52648305e-38)
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmul r41 r80 → r110`	r110 ← 0x0, OFZ, UNF, INX flags set
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmul r42 r43 → r106`	r106 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmul r40 r81 → r111`	r111 ← 0, IFZ flag set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmul r82 r83 → r112`	r112 ← 0, UNF, INX flag set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmul r84 r85 → r113`	r113 ← 0xff800000 (-INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fmul r70 r70 → r120`	r120 ← 0x7f800000, OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	`fmul r80 r80 → r125`	r125 ← 0, UNF, INX flag set

IEEE status flags from floating-point multiply

fmulflags

SYNTAX

```
[ IF rguard ] fmulflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* × (float)r*src2*)

ATTRIBUTES

Function unit	ifmul
Operation code	143
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

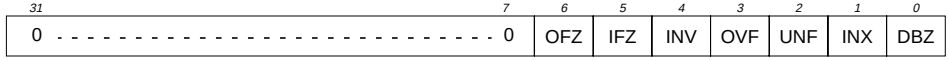
SEE ALSO

fmul faddflags readpcsw

DESCRIPTION

The `fmulflags` operation computes the IEEE exceptions that would result from computing the product r*src1*×r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the product, and the IFZ bit in the result is set. If the product would be denormalized, the OFZ bit in the result is set.

The `fmulflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmulflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmulflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmulflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmulflags r40 r80 → r105`	r105 ← 0
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmulflags r41 r80 → r110`	r110 ← 0x46 (OFZ UNF INX)
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmulflags r42 r43 → r106`	r106 ← 0x10 (INV)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmulflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmulflags r82 r83 → r112`	r112 ← 0x06 (UNF INX)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmulflags r84 r85 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fmulflags r70 r70 → r120`	r120 ← 0x0a (OVF INX)
r80 = 0x00800000 (1.763241526e–38)	`fmulflags r80 r80 → r125`	r125 ← 0x06 (UNF INX)

fneq

Floating-point compare not equal

SYNTAX

```
[ IF rguard ] fneq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 != (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	150
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ineq feql fneqflags readpcsw writepcsw

DESCRIPTION

The `fneq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fneq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fneqflags` operation computes the exception flags that would result from an individual `fneq`.

The `fneq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fneq r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fneq r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fneq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fneq r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fneq r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fneq r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fneq r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fneq r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fneq r50 r50 → r127`	r127 ← 0

IEEE status flags from floating-point compare not equal

fneqflags

SYNTAX

```
[ IF rguard ] fneqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* != (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	151
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

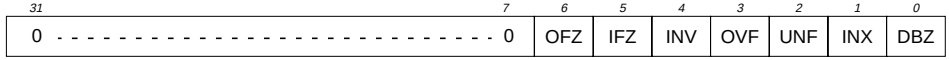
SEE ALSO

fneq ineq fleqflags readpcsw

DESCRIPTION

The `fneqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*!=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fneqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fneqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fneqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fneqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fneqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fneqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fneqflags r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fneqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fneqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fneqflags r50 r50 → r127`	r127 ← 0

fsign

Sign of floating-point value

SYNTAX

```
[ IF rguard ] fsign rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 = 0.0 then
      rdest ← 0
   else if (float)rsrc1 < 0.0 then
      rdest ← 0xffffffff
   else
      rdest ← 1
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	152
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

fsignflags readpcsw writepcsw

DESCRIPTION

The `fsign` operation sets the destination register, r*dest*, to either 0, 1, or –1 depending on the sign of the argument in r*src1*. rdest is set to 0 if r*src1* is equal to zero, to 1 if r*src1* is positive, or to –1 if r*src1* is negative. The argument is treated as an IEEE single-precision floating-point value; the result is an integer. If the argument is denormalized, zero is substituted before computing the comparison, and the IFZ flag in the PCSW is set; thus, the result of `fsign` for a denormalized argument is 0. If `fsign` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsignflags` operation computes the exception flags that would result from an individual `fsign`.

The `fsign` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsign r30 → r100`	r100 ← 1
r40 = 0xbf800000 (-1.0)	`fsign r40 → r105`	r105 ← 0xffffffff (-1)
r50 = 0x80800000 (-1.175494351e-38)	`fsign r50 → r110`	r110 ← 0xffffffff (-1)
r60 = 0x80400000 (-5.877471754e-39)	`fsign r60 → r115`	r115 ← 0, IFZ flag set
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsign r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsign r70 → r117`	r117 ← 0, INV flag set
r80 = 0xff800000 (-INF)	`fsign r80 → r120`	r120 ← 0xffffffff (-1)

IEEE status flags from floating-point sign

fsignflags

SYNTAX

```
[ IF rguard ] fsignflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags(sign((float)r*src1*))

ATTRIBUTES

Function unit	fcomp
Operation code	153
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

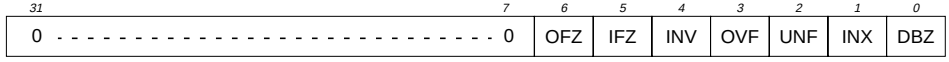
SEE ALSO

fsign readpcsw

DESCRIPTION

The `fsignflags` operation computes the IEEE exceptions that would result from computing the sign of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If the argument is denormalized, zero is substituted before computing the sign, and the IFZ bit in the result is set.

The `fsignflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsignflags r30 → r100`	r100 ← 0
r40 = 0xbf800000 (-1.0)	`fsignflags r40 → r105`	r105 ← 0
r50 = 0x80800000 (-1.175494351e-38)	`fsignflags r50 → r110`	r110 ← 0
r60 = 0x80400000 (-5.877471754e-39)	`fsignflags r60 → r115`	r115 ← 0x20 (IFZ)
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsignflags r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsignflags r70 → r117`	r117 ← 0x10 (INV)
r80 = 0xff800000 (-INF)	`fsignflags r80 → r120`	r120 ← 0

fsqrt

Floating-point square root

SYNTAX

```
[ IF rguard ] fsqrt rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← square_root(r*src1*)

ATTRIBUTES

Function unit	ftough
Operation code	110
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fsqrtflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fsqrt` operation computes the squareroot of r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the squareroot, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fsqrt` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsqrtflags` operation computes the exception flags that would result from an individual `fsqrt`.

The `fsqrt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrt r60 → r90`	r90 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0)	`fsqrt r40 → r95`	r95 ← 0x3fddb3d7 (1.732051), INX flag set
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrt r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrt r40 → r110`	r110 ← 0x3fddb3d7 (1.732051), INX flag set
r82 = 0x00c00000 (1.763241526e–38)	`fsqrt r82 → r112`	r112 ← 0x201cc471 (1.32787105e-19), INX flag set
r84 = 0x7f800000 (+INF)	`fsqrt r84 → r113`	r113 ← 0x7f800000 (+INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrt r70 → r120`	r120 ← 0x5f7fffff (1.8446743e19), INX flag set
r80 = 0x00400000 (5.877471754e-39)	`fsqrt r80 → r125`	r125 ← 0, IFZ flag set

IEEE status flags from floating-point square root

fsqrtflags

SYNTAX

```
[ IF rguard ] fsqrtflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← ieee_flags(square_root((float)r*src1*))

ATTRIBUTES

Function unit	ftough
Operation code	111
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

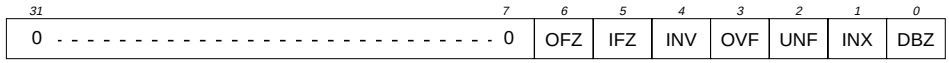
SEE ALSO

`fsqrt` `readpcsw`

DESCRIPTION

The `fsqrtflags` operation computes the IEEE exceptions that would result from computing the squareroot of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If the argument is denormalized, zero is substituted before computing the squareroot, and the IFZ bit in the result is set. If the result is denormalized, and the OFZ flag in the PCSW is set.

The `fsqrtflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrtflags r60 → r90`	r90 ← 0x10 (INV)
r40 = 0x40400000 (3.0)	`fsqrtflags r40 → r95`	r95 ← 0x2 (INX)
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrtflags r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrtflags r40 → r110`	r110 ← 0x2 (INX)
r82 = 0x00c00000 (1.763241526e–38)	`fsqrtflags r82 → r112`	r112 ← 0x2 (INX)
r84 = 0x7f800000 (+INF)	`fsqrtflags r84 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrtflags r70 → r120`	r120 ← 0x2 (INX)
r80 = 0x00400000 (5.877471754e-39)	`fsqrtflags r80 → r125`	r125 ← 0x20 (IFZ)

fsub

Floating-point subtract

SYNTAX

```
[ IF rguard ] fsub rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* – (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	113
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`fsubflags` `isub` `dspisub` `dspidualsub` `readpcsw` `writepcsw`

DESCRIPTION

The `fsub` operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the difference, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fsub` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsubflags` operation computes the exception flags that would result from an individual `fsub`.

The `fsub` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fsub r60 r30 → r90`	r90 ← 0xc0800000 (-4.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fsub r40 r60 → r95`	r95 ← 0x40c00000 (6.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fsub r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fsub r40 r80 → r110`	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fsub r40 r81 → r111`	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	`fsub r82 r83 → r112`	r112 ← 0x0, OFZ flag set
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	`fsub r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	`fsub r70 r86 → r120`	r120 ← 0x7f800000 (+INF), OVF flag set
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	`fsub r87 r30 → r125`	r125 ← 0xffffffff (QNaN)
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	`fsub r87 r30 → r125`	r125 ← 0xffffffff (QNaN), INV flag set
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	`fsub r83 r89 → r126`	r126 ← 0x0, UNF flag set

IEEE status flags from floating-point subtract

fsubflags

SYNTAX

```
[ IF rguard ] fsubflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* – (float)r*src2*)

ATTRIBUTES

Function unit	falu
Operation code	114
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

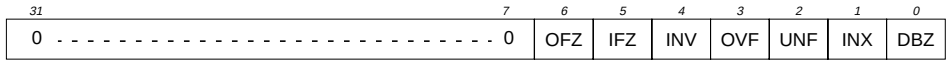
SEE ALSO

fsub faddflags readpcsw

DESCRIPTION

The `fsubflags` operation computes the IEEE exceptions that would result from computing the difference r*src1*–r*src2* and writes a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the difference, and the IFZ bit in the result is set. If the difference would be denormalized, the OFZ bit in the result is set.

The `fsubflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fsubflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fsubflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fsubflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fsubflags r40 r80 → r110`	r110 ← 0x2 (INX)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fsubflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	`fsubflags r82 r83 → r112`	r112 ← 0x40 (OFZ)
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	`fsubflags r84 r85 → r113`	r113 ← 0x10 (INV)
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	`fsubflags r70 r86 → r120`	r120 ← 0x8 (OVF)
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x0
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x10 (INV)
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	`fsubflags r83 r89 → r126`	r126 ← 0x4 (UNF)

funshift1

Funnel-shift 1byte

SYNTAX

```
[ IF rguard ] funshift1 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest*<31:8> ← r*src1*<23:0>
r*dest*<7:0> ← r*src2*<31:24>

ATTRIBUTES

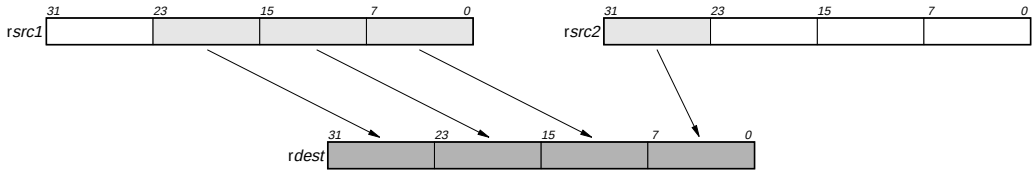
Function unit	shifter
Operation code	99
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift2 funshift3 rol

DESCRIPTION

As shown below, the `funshift1` operation effectively shifts left by one byte the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift1` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift1 r30 r40 → r50`	r50 ← 0xbbccdd11
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift1 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift1 r40 r30 → r70`	r70 ← 0x223344aa

Funnel-shift 2 bytes

funshift2

SYNTAX

```
[ IF rguard ] funshift2 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest*<31:16> ← r*src1*<15:0>
    r*dest*<15:0> ← r*src2*<31:16>

ATTRIBUTES

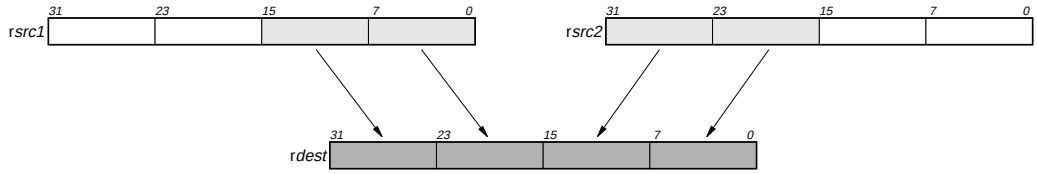
Function unit	shifter
Operation code	100
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift3 rol

DESCRIPTION

As shown below, the `funshift2` operation effectively shifts left by two bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift2` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift2 r30 r40 → r50`	r50 ← 0xccdd1122
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift2 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift2 r40 r30 → r70`	r70 ← 0x3344aabb

funshift3

Funnel-shift 3 bytes

SYNTAX

```
[ IF rguard ] funshift3 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest*<31:24> ← r*src1*<7:0>
r*dest*<23:0> ← r*src2*<31:8>

ATTRIBUTES

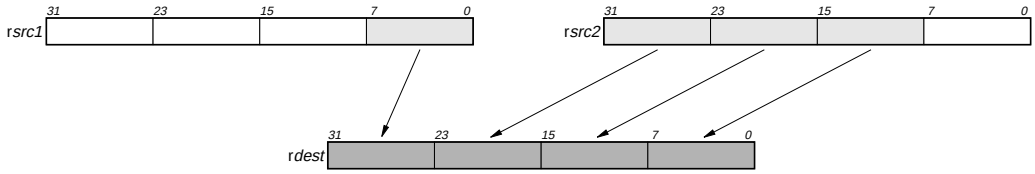
Function unit	shifter
Operation code	101
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift2 rol

DESCRIPTION

As shown below, the `funshift3` operation effectively shifts left by three bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift3` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift3 r30 r40 → r50`	r50 ← 0xdd112233
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift3 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift3 r40 r30 → r70`	r70 ← 0x44aabbcc

Clipped signed absolute value

h_dspiabs

SYNTAX

```
[ IF rguard ] h_dspiabs r0 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src2* >= 0 **then**
    r*dest* ← r*src2*
  else if rsrc2 = 0x80000000 **then**
    r*dest* ← 0x7fffffff
  else
    r*dest* ← –r*src2*
}

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The h_dspiabs operation computes the absolute value of rsrc2, clips the result into the range [0x0..0x7fffffff], and stores the clipped value into r*dest*. All values are signed integers. This operation requires a zero as first argument. The programmer is advised to use the unary pseudo operation dspiabs instead.

The h_dspiabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	h_dspiabs r0 r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	IF r10 h_dspiabs r0 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80000001	IF r20 h_dspiabs r0 r40 → r100	r100 ← 0x7fffffff
r50 = 0x80000000	h_dspiabs r0 r50 → r80	r80 ← 0x7fffffff
r90 = 0x7fffffff	h_dspiabs r0 r90 → r110	r110 ← 0x7fffffff

h_dspidualabs

Dual clipped absolute value of signed 16-bit halfwords

SYNTAX

```
[ IF rguard ] h_dspidualabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   temp1 ← sign_ext16to32(rsrc2<15:0>)
   temp2 ← sign_ext16to32(rsrc2<31:16>)
   if temp1 = 0xffff8000 then temp1 ← 0x7fff
   if temp2 = 0xffff8000 then temp2 ← 0x7fff
   if temp1 < 0 then temp1 ← –temp1
   if temp2 < 0 then temp2 ← –temp2
   rdest<31:16> ← temp2<15:0>
   rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspiabs dspidualadd dspidualmul dspidualsub dspiabs

DESCRIPTION

The h_dspidualabs operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of rsrc2. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers. This operation requires a zero as first argument. The programmer is advised to use the dspidualabs pseudo operation instead.

The h_dspidualabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	`h_dspidualabs r0 r30 → r60`	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	`IF r10 h_dspidualabs r0 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0x80008001	`IF r20 h_dspidualabs r0 r40 → r100`	r100 ← 0x7fff7fff
r50 = 0x0032ffff	`h_dspidualabs r0 r50 → r80`	r80 ← 0x00320001
r90 = 0x7fffffff	`h_dspidualabs r0 r90 → r110`	r110 ← 0x7fff0001

Hardware absolute value

h_iabs

SYNTAX

```
[ IF rguard ] h_iabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc2 < 0 then
      rdest ← –rsrc2
   else
      rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iabs fabsval

DESCRIPTION

The `h_iabs` operation computes the absolute value of rsrc2 and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. This operation requires a zero as first argument. The programmer is advised to use the `iabs` pseudo operation instead.

The `h_iabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`h_iabs r0 r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 h_iabs r0 r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 h_iabs r0 r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`h_iabs r0 r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`h_iabs r0 r60 → r110`	r110 ← 0x80000000
r20 = 1	`h_iabs r0 r20 → r120`	r120 ← 1

h_st16d

Hardware 16-bit store with displacement

SYNTAX

```
[ IF rguard ] h_st16d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc2 + d + (1 ⊕ bs)] ← rsrc1<7:0>
   mem[rsrc2 + d + (0 ⊕ bs)] ← rsrc1<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16 st16d st8 st8d st32 st32d readpcsw ijmpf

DESCRIPTION

The h_st16d operation stores the least-significant 16-bit halfword of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If h_st16d is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 2), the result of h_st16d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The h_st16d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`h_st16d(2) r80 r10`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 h_st16d(-4) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 h_st16d(-4) r70 r30`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

Hardware 32-bit store with displacement

h_st32d

SYNTAX

```
[ IF rguard ] h_st32d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
  if PCSW.bytesex = LITTLE_ENDIAN then
    bs ← 3
  else
    bs ← 0
  mem[rsrc2 + d + (3 ⊕ bs)] ← rsrc1<7:0>
  mem[rsrc2 + d + (2 ⊕ bs)] ← rsrc1<15:8>
  mem[rsrc2 + d + (1 ⊕ bs)] ← rsrc1<24:16>
  mem[rsrc2 + d + (0 ⊕ bs)] ← rsrc1<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

st32 st32d st16 st16d st8 st8d readpcsw ijmpf

DESCRIPTION

The h_st32d operation stores all 32 bits of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If h_st32d is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 4), the result of h_st32d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The h_st32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`h_st32d(4) r80 r10`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 h_st32d(-8) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 h_st32d(-8) r70 r30`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

h_st8d

Hardware 8-bit store with displacement

SYNTAX

```
[ IF rguard ] h_st8d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
mem[r*src2* + *d*] ← r*src1*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

st8 st8d st16 st16d st32 st32d

DESCRIPTION

The h_st8d operation stores the least-significant 8-bit byte of r*src1* into the memory location pointed to by the address formed from the sum r*src2* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The h_st8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	h_st8d(3) r40 r30	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	IF r50 h_st8d(-4) r70 r20	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	IF r60 h_st8d(-4) r70 r30	[0xcfe] ← 0xdd

Read clock cycle counter, most-significant word

hicycles

SYNTAX

```
[ IF rguard ] hicycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<63:32>

ATTRIBUTES

Function unit	fcomp
Operation code	155
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

cycles curcycles writepcsw

DESCRIPTION

Refer to Section 3.1.5, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The hicycles operation copies the high 32 bits of the slave register Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if cycles and hicycles are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The hicycles operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`hicycles → r60`	r60 ← 0xabcdefff
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 hicycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 hicycles → r100`	r100 ← 0xabcdefff

iabs

Absolute value

pseudo-op for h_iabs

SYNTAX

```
[ IF rguard ] iabs rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < 0 then
      rdest ← –rsrc1
   else
      rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

h_iabs dspiabs dspidualabs fabsval

DESCRIPTION

The `iabs` operation is a pseudo operation transformed by the scheduler into an `h_iabs` with zero as the first argument and a second argument equal to the `iabs` argument. (Note: pseudo operations cannot be used in assembly source files.)

The `iabs` operation computes the absolute value of r*src1* and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer.

The `iabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`iabs r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 iabs r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 iabs r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`iabs r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`iabs r60 → r110`	r110 ← 0x80000000
r20 = 1	`iabs r20 → r120`	r120 ← 1

Signed add

iadd

SYNTAX

`[ IF rguard ] iadd rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* + r*src2*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iaddi carry dspiadd
dspidualadd fadd

DESCRIPTION

The `iadd` operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `iadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`iadd r60 r60 → r80`	r80 ← 0x200
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 iadd r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 iadd r60 r30 → r90`	r90 ← 0x1011
r70 = 0xffffff00, r40 = 0xffffff9c	`iadd r70 r40 → r100`	r100 ← 0xfffffe9c

iaddi

Add with immediate

SYNTAX

```
[ IF rguard ] iaddi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* + *n*

ATTRIBUTES

Function unit	alu
Operation code	5
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iadd carry

DESCRIPTION

The `iaddi` operation sums a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The `iaddi` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	`iaddi(127) r30 → r70`	r70 ← 0xf90
r10 = 0, r40 = 0xffffff9c	`IF r10 iaddi(1) r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	`IF r20 iaddi(1) r40 → r90`	r90 ← 0xffffff9d
r50 = 0x1000	`iaddi(15) r50 → r120`	r120 ← 0x100f
r60 = 0xfffffff0	`iaddi(2) r60 → r110`	r110 ← 0xfffffff2
r60 = 0xfffffff0	`iaddi(17) r60 → r120`	r120 ← 1

Signed average

iavgonep

SYNTAX

```
[ IF rguard ] iavgonep rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (sign_ext32to64(r*src1*) + sign_ext32to64(r*src2*) + 1) >> 1;

ATTRIBUTES

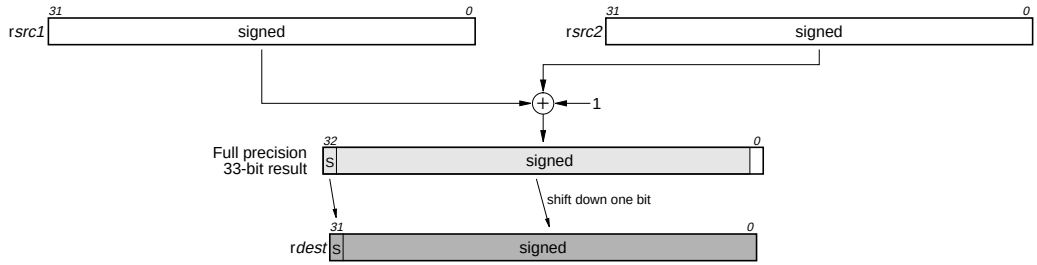
Function unit	dspalu
Operation code	25
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

quadavg iadd

DESCRIPTION

As shown below, the `iavgonep` operation returns the average of the two arguments. This operation computes the sum r*src1*+r*src2*+1, shifts the sum right by 1 bit, and stores the result into r*dest*. The operands are signed integers.



The `iavgonep` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10, r70 = 0x20	`iavgonep r60 r70 → r80`	r80 ← 0x18
r10 = 0, r60 = 0x10, r30 = 0x20	`IF r10 iavgonep r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x9, r30 = 0x20	`IF r20 iavgonep r60 r30 → r90`	r90 ← 0x15
r70 = 0xfffffff7, r40 = 0x2	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd
r70 = 0xfffffff7, r40 = 0x3	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd

ibytesel

Signed select byte

SYNTAX

```
[ IF rguard ] ibytesel rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

FUNCTION

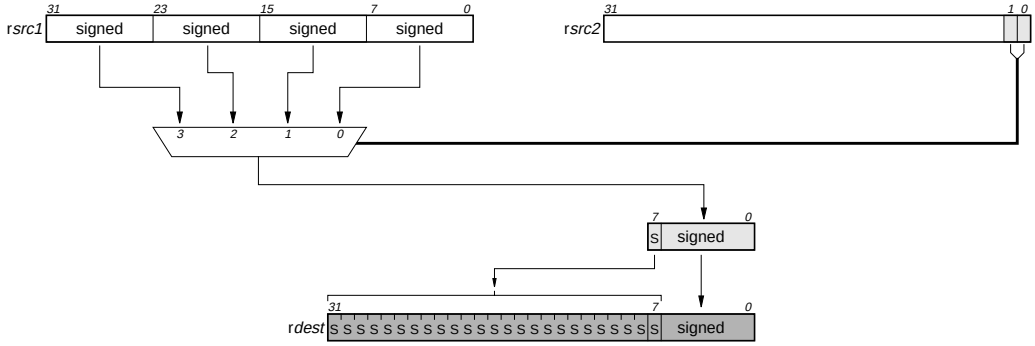
```
if rguard then {
   if rsrc2 = 0 then
      rdest ← sign_ext8to32(rsrc1<7:0>)
   else if rsrc2 = 1 then
      rdest ← sign_ext8to32(rsrc1<15:8>)
   else if rsrc2 = 2 then
      rdest ← sign_ext8to32(rsrc1<23:16>)
   else if rsrc2 = 3 then
      rdest ← sign_ext8to32(rsrc1<31:24>)
}
```

SEE ALSO

ubytesel sex8 packbytes

DESCRIPTION

As shown below, the `ibytesel` operation selects one byte from the argument, r*src1*, sign-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of `ibytesel` is undefined.



The `ibytesel` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	`ibytesel r30 r40 → r50`	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	`IF r10 ibytesel r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	`IF r20 ibytesel r60 r70 → r90`	r90 ← 0xffffffcc
r100 = 0xffffff7f, r110 = 0	`ibytesel r100 r110 → r120`	r120 ← 0x0000007f

Clip signed to signed

iclipi

SYNTAX

```
[ IF rguard ] iclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← min(max(r*src1*, –r*src2*–1), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	74
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

uclipi uclipu imin imax

DESCRIPTION

The `iclipi` operation returns the value of r*src1* clipped into the unsigned integer range (–r*src2*–1) to r*src2*, inclusive. The argument r*src1* is considered a signed integer; rsrc2 is considered an unsigned integer and must have a value between 0 and 0x7fffffff inclusive.

The `iclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`iclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 iclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 iclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`iclipi r100 r110 → r120`	r120 ← 0xffc00000

iclr

Invalidate all instruction cache blocks

SYNTAX

```
[ IF rguard ] iclr
```

FUNCTION

```
if rguard then {
   block ← 0
   for all blocks in instruction cache {
      icache_reset_valid_block(block)
      block ← block + 1
   }
}
```

ATTRIBUTES

Function unit	branch
Operation code	184
Number of operands	0
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	2, 3, 4

SEE ALSO

dcb dinvalid

DESCRIPTION

The `iclr` operation resets the valid bits of all blocks in the instruction cache.

`iclr` does clear the valid bits of locked blocks. `iclr` does not change the replacement status of instruction-cache blocks.

`iclr` ensures coherency between caches and main memory by discarding all pending prefetch operations.

The side effect time behavior of `iclr` for TM1000 is such that if instruction *i* performs an iclr, instructions *i, i+1, i+2* will be included in the discard from the instruction cache, but *i+3* will be retained.

The `iclr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`iclr`	
r10 = 0	`IF r10 iclr`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 iclr`	

Identity

pseudo-op for iadd

ident

SYNTAX

```
[ IF rguard ] ident rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iadd

DESCRIPTION

The `ident` operation is a pseudo operation transformed by the scheduler into an `iadd` with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

The `ident` operation copies the argument r*src1* to r*dest*. It is used by the instruction scheduler to implement register to register copying.

The `ident` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x100	`ident r30 → r40`	r40 ← 0x100
r10 = 0, r50 = 0x12345678	`IF r10 ident r50 → r60`	no change, since guard is false
r20 = 1, r50 = 0x12345678	`IF r20 ident r50 → r70`	r70 ← 0x12345678

ieql

Signed compare equal

SYNTAX

```
[ IF rguard ] ieql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ueql ieqli

DESCRIPTION

The `ieql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ieql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ieql r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ieql r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ieql r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ieql r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ieql r70 r70 → r110`	r110 ← 1

Signed compare equal with immediate

ieqli

SYNTAX

```
[ IF rguard ] ieqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	4
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql igeqi ueqli

DESCRIPTION

The `ieqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ieqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ieqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ieqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ieqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ieqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ieqli(63) r40 → r100`	r100 ← 0
r60 = 0xfffffffc0	`ieqli(-64) r60 → r120`	r120 ← 1

ifir16

Sum of products of signed 16-bit halfwords

SYNTAX

```
[ IF rguard ] ifir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← sign_ext16to32(r*src1*<31:16>) × sign_ext16to32(r*src2*<31:16>) +
sign_ext16to32(r*src1*<15:0>) × sign_ext16to32(r*src2*<15:0>)

ATTRIBUTES

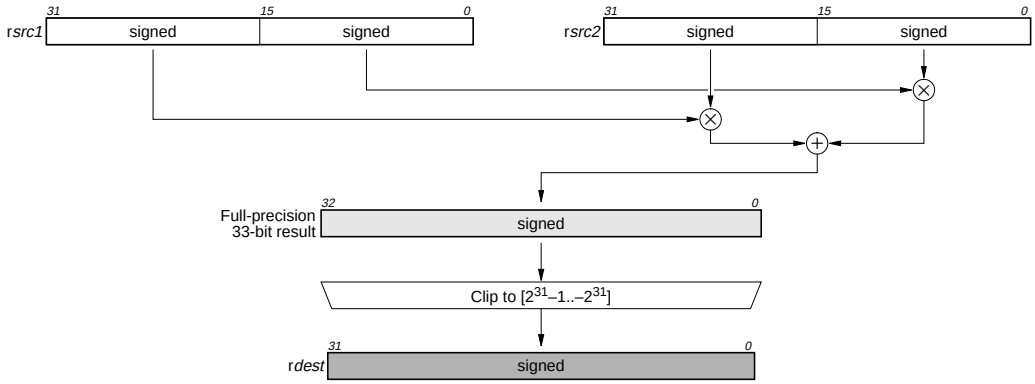
Function unit	dspmul
Operation code	93
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ifir8ui ufir8uu
ifir16

DESCRIPTION

As shown below, the `ifir16` operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0x80000000..0x7fffffff] before being written into r*dest*.



The `ifir16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	`ifir16 r30 r40 → r50`	r50 ← 0x8
r10 = 0, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r10 ifir16 r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r20 ifir16 r60 r70 → r90`	r90 ← 0xffffb1e0
r30 = 0x00020003, r70 = 0x0064ff9c	`ifir16 r30 r70 → r100`	r100 ← 0xffffff9c

Signed sum of products of signed bytes

ifir8ii

SYNTAX

```
[ IF rguard ] ifir8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← sign_ext8to32(r*src1*<31:24>) × sign_ext8to32(r*src2*<31:24>) +
sign_ext8to32(r*src1*<23:16>) × sign_ext8to32(r*src2*<23:16>) +
sign_ext8to32(r*src1*<15:8>) × sign_ext8to32(r*src2*<15:8>) +
sign_ext8to32(r*src1*<7:0>) × sign_ext8to32(r*src2*<7:0>)

ATTRIBUTES

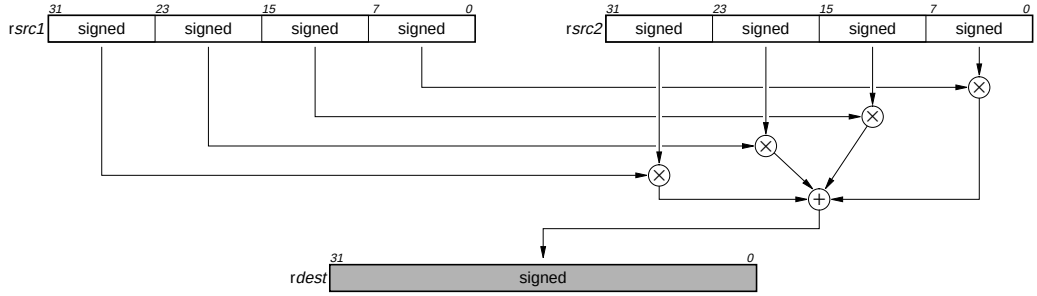
Function unit	dspmul
Operation code	92
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ui ufir8uu ifir16
ufir16

DESCRIPTION

As shown below, the `ifir8ii` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ii` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ii r70 r30 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ii r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ii r80 r40 → r110`	r110 ← 0xffff63c0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ii r50 r60 → r120`	r120 ← 0x200

ifir8ui

Signed sum of products of unsigned/signed bytes

SYNTAX

```
[ IF rguard ] ifir8ui rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← zero_ext8to32(r*src1*<31:24>) × sign_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × sign_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × sign_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × sign_ext8to32(r*src2*<7:0>)

ATTRIBUTES

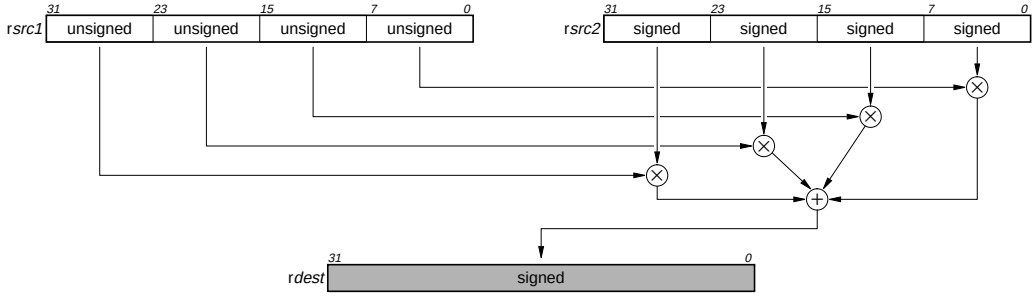
Function unit	dspmul
Operation code	91
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ui` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. The bytes from r*src1* are considered unsigned, but the bytes from r*src2* are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ui` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ui r30 r70 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ui r30 r70 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ui r40 r80 → r110`	r110 ← 0x2bc0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ui r60 r50 → r120`	r120 ← 0xfffe0200

Convert floating-point to integer using PCSW rounding mode

ifixieee

SYNTAX

```
[ IF rguard ] ifixieee rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	121
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ufixieee ifixrz ufixrz

DESCRIPTION

The `ifixieee` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixieee` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixieeeflags` operation computes the exception flags that would result from an individual `ifixieee`.

The `ifixieee` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieee r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixieee r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixieee r50 → r115`	r115 ← -2, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixieee r80 → r122`	r122 ← 0, INV flag set

ifixieeeflags

IEEE status flags from convert floating-point to integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ifixieeeflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	122
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

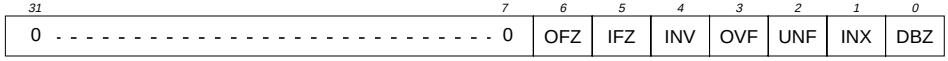
SEE ALSO

ifixieee ufixieeeflags
ifixrzflags ufixrzflags

DESCRIPTION

The `ifixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixieeeflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixieeeflags r80 → r122`	r122 ← 0x10 (INV)

Convert floating-point to integer with round toward zero

ifixrz

SYNTAX

```
[ IF rguard ] ifixrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	21
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ufixieee ufixrz

DESCRIPTION

The `ifixrz` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixrzflags` operation computes the exception flags that would result from an individual `ifixrz`.

The `ifixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrz r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixrz r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixrz r50 → r115`	r115 ← -1, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixrz r80 → r122`	r122 ← 0, INV flag set

ifixrzflags

IEEE status flags from convert floating-point to integer with round toward zero

SYNTAX

```
[ IF rguard ] ifixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	129
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

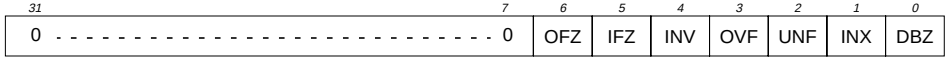
SEE ALSO

ifixrz ufixrzflags
ifixieeeflags
ufixieeeflags

DESCRIPTION

The `ifixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixrzflags r80 → r122`	r122 ← 0x10 (INV)

If non-zero negate

iflip

SYNTAX

```
[ IF rguard ] iflip rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* = 0 **then**
    r*dest* ← r*src2*
  else
    r*dest* ← –r*src2*
}

ATTRIBUTES

Function unit	dspalu
Operation code	77
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

inonzero izero

DESCRIPTION

The `iflip` operation copies r*src2* to r*dest* if r*src1* = 0; otherwise (if r*src1* != 0), r*dest* is set to the two's-complement of r*src2*. All values are signed integers.

The `iflip` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0, r40 = 1	`iflip r30 r40 → r50`	r50 ← 0x1
r10 = 0, r60 = 0xffff0000, r70 = 0xabc	`IF r10 iflip r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xffff0000, r70 = 0xabc	`IF r20 iflip r60 r70 → r90`	r90 ← 0xffff544
r30 = 0, r60 = 0xffffff9c	`iflip r30 r60 → r100`	r100 ← 0xffffff9c

ifloat

Convert signed integer to floating-point

SYNTAX

```
[ IF rguard ] ifloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	20
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ufloat ifloatrz ufloatrz ifixieee ifloatflags

DESCRIPTION

The `ifloat` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ifloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatflags` operation computes the exception flags that would result from an individual `ifloat`.

The `ifloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloat r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloat r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloat r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

IEEE status flags from convert signed integer to floating-point

ifloatflags

SYNTAX

```
[ IF rguard ] ifloatflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	130
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

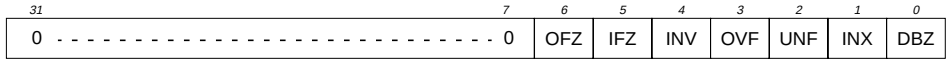
SEE ALSO

ifloat ifloatrzflags
ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatflags` operation computes the IEEE exceptions that would result from converting the signed integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ifloatflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatflags r80 → r122`	r122 ← 0x02 (INX)

ifloatrz

Convert signed integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ifloatrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	117
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloat ufloatrz ifixieee ifloatflags

DESCRIPTION

The `ifloatrz` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ifloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatrzflags` operation computes the exception flags that would result from an individual `ifloatrz`.

The `ifloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloatrz r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrz r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloatrz r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

IEEE status flags from convert signed integer to floating-point with rounding toward zero

ifloatrzflags

SYNTAX

```
[ IF rguard ] ifloatrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	118
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

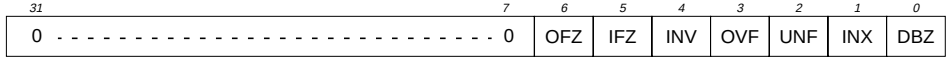
SEE ALSO

ifloatrz ifloatflags ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatrzflags` operation computes the IEEE exceptions that would result from converting the signed integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ifloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatrzflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrzflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatrzflags r80 → r122`	r122 ← 0x02 (INX)

igeq

Signed compare greater or equal

SYNTAX

```
[ IF rguard ] igeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq igeqi

DESCRIPTION

The `igeq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igeq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`igeq r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 igeq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 igeq r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`igeq r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`igeq r70 r70 → r110`	r110 ← 1

Signed compare greater or equal with immediate

igeqi

SYNTAX

```
[ IF rguard ] igeqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	1
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq iles ieqli

DESCRIPTION

The `igeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`igeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`igeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igeqi(-64) r60 → r120`	r120 ← 0

igtr

Signed compare greater

SYNTAX

```
[ IF rguard ] igtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 > rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles igtri

DESCRIPTION

The `igtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`igtr r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 igtr r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 igtr r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`igtr r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`igtr r70 r70 → r110`	r110 ← 0

Signed compare greater with immediate

igtri

SYNTAX

```
[ IF rguard ] igtri(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 > n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	0
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr igeqi

DESCRIPTION

The `igtri` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igtri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igtri(2) r30 → r80`	r80 ← 1
r30 = 3	`igtri(3) r30 → r90`	r90 ← 0
r30 = 3	`igtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igtri(-64) r60 → r120`	r120 ← 0

iimm

Signed immediate

SYNTAX

`iimm(`*n*`)` → r*dest*

FUNCTION

r*dest* ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0x80000000 ..0x7fffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uimm

DESCRIPTION

The `iimm` operation stores the signed 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`iimm(2)` → `r10`	r10 ← 2
	`iimm(0x100)` → `r20`	r20 ← 0x100
	`iimm(0xfffc0000)` → `r30`	r30 ← 0xfffc0000

Interruptible indirect jump on false

ijmpf

SYNTAX

```
[ IF rguard ] ijmpf rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if (rsrc1 & 1) = 0 then {
      DPC ← rsrc2
      if exception  is pending then
         service exception
      elseif interrupt is pending then
         service interrupts
      else
         PC, SPC ← rsrc2
   }
}
```

ATTRIBUTES

Function unit	branch
Operation code	181
Number of operands	2
Modifier	no
Modifier range	—
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpt ijmpi

DESCRIPTION

The `ijmpf` operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If neither interrupts or exceptions are pending and the LSB of r*src1* is 0, the DPC, PC, and SPC registers are set equal to r*src2*. If an interrupt or exception is pending and the LSB of r*src1* is 0, DPC is set equal to r*src2* and the service routine is invoked, where exceptions have priorities over interrupts. If the LSB of r*src1* is 1, program execution continues with the next sequential instruction.

The `ijmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`ijmpf r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 1, r70 = 0x330	`ijmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 ijmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 ijmpf r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

ijmpi

Interruptible jump immediate

SYNTAX

```
[ IF rguard ] ijmpi(address)
```

FUNCTION

if r*guard* **then {**
  DPC ← *address*
  if exception is pending **then**
    service exception
  else if interrupt is pending **then**
    service interrupts
  else
    PC, SPC ← address
}

ATTRIBUTES

Function unit	branch
Operation code	179
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpf ijmpt

DESCRIPTION

The `ijmpi` operation changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending, the DPC, PC, and SPC registers are set equal to *address*. If an exception or interrupts is pending, DPC is set equal to *address* and a service routine is invoked, where exceptions have priorities over interrupts. *address* is an immediate opcode modifier.

The `ijmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified.

EXAMPLES

Initial Values	Operation	Result
	`ijmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 ijmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 ijmpi(0x8000)`	program execution continues at 0x8000

Interruptible indirect jump on true

ijmpt

SYNTAX

```
[ IF rguard ] ijmpt rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
  if (rsrc1 & 1) = 1 then {
    DPC ← rsrc2
    if exception is pending then
      service exception
    elseif interrupt is pending then
      service interrupts
    else
      PC, SPC ← rsrc2
  }
}
```

ATTRIBUTES

Function unit	branch
Operation code	177
Number of operands	2
Modifier	no
Modifier range	—
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpf ijmpi

DESCRIPTION

The `ijmpt` operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending and the LSB of r*src1* is 1, the DPC, PC, and SPC registers are set equal to r*src2*. If an exception or interrupt is pending and the LSB of r*src1* is 1, DPC is set equal to r*src2* and a service routine is invoked, where exceptions have prioriy over interrupts. If the LSB of r*src1* is 0, program execution continues with the next sequential instruction.

The `ijmpt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`ijmpt r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 0, r70 = 0x330	`ijmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 ijmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 ijmpt r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

ild16

Signed 16-bit load

pseudo-op for ild16d(0)

SYNTAX

```
[ IF rguard ] ild16 rsrc1 → rdest
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[(rsrc1 +(1 ⊕ bs)]
   temp<15:8> ← mem[(rsrc1 + (0 ⊕ bs)]
   rdest ← sign_ext16to32(temp<15:0>)
}
```

SEE ALSO

ild16d ild16r ild16x

DESCRIPTION

The `ild16` operation is a pseudo operation transformed by the scheduler into an `ild16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild16` operation loads the 16-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `ild16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`ild16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 ild16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 ild16 r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16 r50 → r90`	r90 undefined, since 0xd01 is not a multiple of 2

Signed 16-bit load with displacement

ild16d

SYNTAX

```
[ IF rguard ] ild16d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 + d + (1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + d + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 uld16d ild16r uld16r ild16x uld16x

DESCRIPTION

The `ild16d` operation loads the 16-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 to 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of `ild16d` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`ild16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 ild16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 ild16d(-4) r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16d(-4) r50 → r90`	r90 undefined, since 0xd01 +(–4) is not a multiple of 2

ild16r

Signed 16-bit load with index

SYNTAX

```
[ IF rguard ] ild16r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[(rsrc1 + rsrc2 +(1 ⊕ bs)]
   temp<15:8> ← mem[(rsrc1 + rsrc2 + (0 ⊕ bs)]
   rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	195
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 ild16d uld16d uld16r ild16x uld16x

DESCRIPTION

The `ild16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `ild16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`ild16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16r r40 r30 → r100`	r100 ← 0xffff8433
r70 = 0xd01, r30 = 0xfffffffc	`ild16r r70 r30 → r110`	r110 undefined, since 0xd01 +(–4) is not a multiple of 2

Signed 16-bit load with scaled index

ild16x

SYNTAX

```
[ IF rguard ] ild16x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 + (2 × rsrc2) + (1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + (2 × rsrc2) + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	196
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 ild16d uld16d ild16r uld16r uld16x

DESCRIPTION

The `ild16x` operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of `ild16x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16x` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`ild16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16x r40 r20 → r90`	r90 ← 0xffff8433
r70 = 0xd01, r30 = 1	`ild16x r70 r30 → r110`	r110 undefined, since 0xd01 + 2×1 is not a multiple of 2

ild8

Signed 8-bit load

pseudo-op for ild8d(0)

SYNTAX

```
[ IF rguard ] ild8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`uld8 ild8d uld8d ild8r uld8r`

DESCRIPTION

The `ild8` operation is a pseudo operation transformed by the scheduler into an `ild8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild8` operation loads the 8-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`ild8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 ild8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 ild8 r20 → r80`	r80 ← 0xffffff84
r50 = 0xd01, [0xd01] = 0x33	`ild8 r50 → r90`	r90 ← 0x00000033

Signed 8-bit load with displacement

ild8d

SYNTAX

```
[ IF rguard ] ild8d(d) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8 uld8d ild8r uld8r

DESCRIPTION

The `ild8d` operation loads the 8-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier in the range -64 to 63, inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`ild8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 ild8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 ild8d(-4) r20 → r80`	r80 ← 0xffffff84
r50 = 0xd05, [0xd01] = 0x33	`ild8d(-4) r50 → r90`	r90 ← 0x00000033

ild8r

Signed 8-bit load with index

SYNTAX

```
[ IF rguard ] ild8r rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	193
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`ild8` `uld8` `ild8d` `uld8d` `uld8r`

DESCRIPTION

The `ild8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`ild8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 ild8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 ild8r r40 r30 → r100`	r100 ← 0xffffff84
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`ild8r r70 r30 → r110`	r110 ← 0x00000033

Signed compare less or equal

pseudo-op for igeq

ileq

SYNTAX

```
[ IF rguard ] ileq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 <= rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ileqi

DESCRIPTION

The `ileq` operation is a pseudo operation transformed by the scheduler into an `igeq` with the arguments exchanged (`ileq`'s r*src1* is `igeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ileq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ileq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ileq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 ileq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ileq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ileq r70 r70 → r110`	r110 ← 1

ileqi

Signed compare less or equal with immediate

SYNTAX

```
[ IF rguard ] ileqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 <= n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	42
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq igeqi

DESCRIPTION

The `ileqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ileqi(2) r30 → r80`	r80 ← 0
r30 = 3	`ileqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ileqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ileqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ileqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ileqi(-64) r60 → r120`	r120 ← 1

Signed compare less

pseudo-op for igtr

iles

SYNTAX

```
[ IF rguard ] iles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ilesi

DESCRIPTION

The `iles` operation is a pseudo operation transformed by the scheduler into an `igtr` with the arguments exchanged (`iles`'s r*src1* is `igtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `iles` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `iles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`iles r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 iles r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 iles r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`iles r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`iles r70 r70 → r110`	r110 ← 0

ilesi

Signed compare less with immediate

SYNTAX

```
[ IF rguard ] ilesi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	2
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles ileqi

DESCRIPTION

The `ilesi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ilesi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ilesi(2) r30 → r80`	r80 ← 0
r30 = 3	`ilesi(3) r30 → r90`	r90 ← 0
r30 = 3	`ilesi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ilesi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ilesi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ilesi(-64) r60 → r120`	r120 ← 1

Signed maximum

imax

SYNTAX

```
[ IF rguard ] imax rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* > r*src2* **then**
    r*dest* ← r*src1*
  else
    r*dest* ← r*src2*
}

ATTRIBUTES

Function unit	dspalu
Operation code	24
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

imin

DESCRIPTION

The `imax` operation sets the destination register, r*dest*, to the contents of r*src1* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src2*. The arguments are treated as signed integers.

The `imax` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imax r30 r20 → r80`	r80 ← 2
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imax r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imax r60 r40 → r90`	r90 ← 0x100
r70 = 0xffffff00, r40 = 0xffffff9c	`imax r70 r40 → r100`	r100 ← 0xffffff9c

imin

Signed minimum

SYNTAX

```
[ IF rguard ] imin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	23
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

imax

DESCRIPTION

The `imin` operation sets the destination register, r*dest*, to the contents of r*src2* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src1*. The arguments are treated as signed integers.

The `imin` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imin r30 r20 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imin r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imin r60 r40 → r90`	r90 ← 0xffffff9c
r70 = 0xffffff00, r40 = 0xffffff9c	`imin r70 r40 → r100`	r100 ← 0xffffff00

Signed multiply

imul

SYNTAX

```
[ IF rguard ] imul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  temp ← (sign_ext32to64(r*src1*) × sign_ext32to64(r*src2*))
  r*dest* ← temp<31:0>

ATTRIBUTES

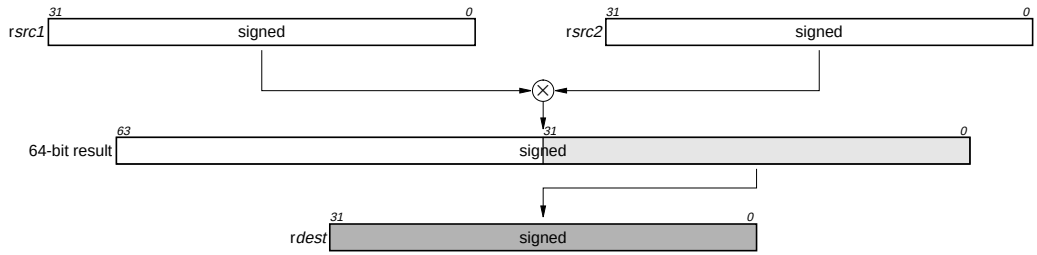
Function unit	ifmul
Operation code	27
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imul` operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered signed integers. No overflow or underflow detection is performed.



The `imul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`imul r60 r60 → r80`	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imul r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 imul r60 r30 → r90`	r90 ← 0xf1100
r70 = 0xffffff00, r40 = 0xffffff9c	`imul r70 r40 → r100`	r100 ← 0x6400

imulm

Signed multiply, return most-significant 32 bits

SYNTAX

```
[ IF rguard ] imulm rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
temp ← (sign_ext32to64(r*src1*) × sign_ext32to64(r*src2*))
r*dest* ← temp<63:32>

ATTRIBUTES

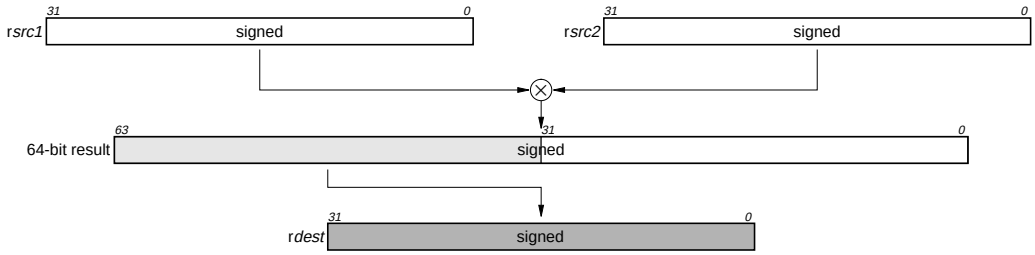
Function unit	ifmul
Operation code	139
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imulm` operation computes the product r*src1*×r*src2* and writes the most-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered signed integers.



The `imulm` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	`imulm r60 r60 → r80`	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imulm r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	`IF r20 imulm r60 r30 → r90`	r90 ← 0xff10ff11
r70 = 0xffffff00, r40 = 0x64	`imulm r70 r40 → r100`	r100 ← 0xffffffff

Signed negate

pseudo-op for isub

ineg

SYNTAX

```
[ IF rguard ] ineg rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← –rsrc1

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isub

DESCRIPTION

The `ineg` operation is a pseudo operation transformed by the scheduler into an `isub` with r0 (always contains 0) as the first argument and r*src1* as the second argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ineg` operation computes the negative of r*src1* and writes the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. If rsrc1 = 0x80000000, then `ineg` returns 0x80000000 since the positive value is not representable.

The `ineg` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`ineg r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 ineg r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 ineg r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`ineg r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`ineg r60 → r110`	r110 ← 0x80000000
r20 = 1	`ineg r20 → r120`	r120 ← 0xffffffff

ineq

Signed compare not equal

SYNTAX

```
[ IF rguard ] ineq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql igtr

DESCRIPTION

The `ineq` operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The `ineq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ineq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	`IF r10 ineq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ineq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ineq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ineq r70 r70 → r110`	r110 ← 0

Signed compare not equal with immediate

ineqi

SYNTAX

```
[ IF rguard ] ineqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 != n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	3
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igeqi

DESCRIPTION

The `ineqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ineqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ineqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ineqi(3) r30 → r90`	r90 ← 0
r30 = 3	`ineqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ineqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ineqi(63) r40 → r100`	r100 ← 1
r60 = 0xfffffffc0	`ineqi(-64) r60 → r120`	r120 ← 0

inonzero

If nonzero select zero

SYNTAX

```
[ IF rguard ] inonzero rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != 0 then
      rdest ← 0
   else
      rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	47
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

izero iflip

DESCRIPTION

The `inonzero` operation writes 0 into r*dest* if the value of r*src1* is not zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The `inonzero` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`inonzero r30 r20 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 inonzero r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 inonzero r60 r40 → r90`	r90 ← 0
r10 = 0, r40 = 0xffffff9c	`inonzero r10 r40 → r100`	r100 ← 0xffffff9c
r20 = 1, r60 = 0x100	`inonzero r20 r60 → r110`	r110 ← 0
r10 = 0, r70 = 0x456789	`inonzero r10 r70 → r120`	r120 ← 0x456789

Subtract

isub

SYNTAX

`[ IF rguard ] isub rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* – r*src2*

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isubi borrow dspisub dspidualsub fsub

DESCRIPTION

The `isub` operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `isub` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`isub r30 r40 → r80`	r80 ← 0xffffffff
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 isub r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 isub r50 r60 → r90`	r90 ← 0xf00
r70 = 0x80000000, r40 = 4	`isub r70 r40 → r100`	r100 ← 0x7ffffffc

isubi

Subtract with immediate

SYNTAX

```
[ IF rguard ] isubi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* – *n*

ATTRIBUTES

Function unit	alu
Operation code	32
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isub borrow

DESCRIPTION

The `isubi` operation computes the difference of a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The `isubi` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	`isubi(127) r30 → r70`	r70 ← 0xe92
r10 = 0, r40 = 0xffffff9c	`IF r10 isubi(1) r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	`IF r20 isubi(1) r40 → r90`	r90 ← 0xffffff9b
r50 = 0x1000	`isubi(15) r50 → r120`	r120 ← 0x0ff1
r60 = 0xfffffff0	`isubi(2) r60 → r110`	r110 ← 0xffffffee
r20 = 1	`isubi(17) r20 → r120`	r120 ← 0xfffffff0

If zero select zero

izero

SYNTAX

```
[ IF rguard ] izero rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = 0 then
      rdest ← 0
   else
      rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	46
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

inonzero iflip

DESCRIPTION

The `izero` operation writes 0 into r*dest* if the value of r*src1* is equal to zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The `izero` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`izero r30 r20 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 izero r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 izero r60 r40 → r90`	r90 ← 0xffffff9c
r10 = 0, r40 = 0xffffff9c	`izero r10 r40 → r100`	r100 ← 0
r20 = 1, r60 = 0x100	`izero r20 r60 → r110`	r110 ← 0x100
r20 = 1, r70 = 0x456789	`izero r20 r70 → r120`	r120 ← 0x456789

jmpf

Indirect jump on false

SYNTAX

```
[ IF rguard ] jmpf rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 0 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	180
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpt jmpi ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpf` operation conditionally changes the program flow. If the LSB of r*src1* is 0, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`jmpf r50 r70`	program execution continues at 0x330
r20 = 1, r70 = 0x330	`jmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 jmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 jmpf r50 r60`	program execution continues at 0x8000

Jump immediate

jmpi

SYNTAX

```
[ IF rguard ] jmpi(address)
```

FUNCTION

if r*guard* **then**
PC ← *address*

ATTRIBUTES

Function unit	branch
Operation code	178
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpi` operation changes the program flow by setting the PC register equal to the immediate opcode modifier *address*.

The `jmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken.

EXAMPLES

Initial Values	Operation	Result
	`jmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 jmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 jmpi(0x8000)`	program execution continues at 0x8000

jmpt

Indirect jump on true

SYNTAX

```
[ IF rguard ] jmpt rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 1 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	176
Number of operands	2
Modifier	no
Modifier range	—
Latency	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpi ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpt` operation conditionally changes the program flow. If the LSB of r*src1* is 1, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`jmpt r50 r70`	program execution continues at 0x330
r20 = 0, r70 = 0x330	`jmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 jmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 jmpt r50 r60`	program execution continues at 0x8000

32-bit load

pseudo-op for ld32d(0)

ld32

SYNTAX

```
[ IF rguard ] ld32 rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 3
  else
    bs ← 0
  r*dest*<7:0> ← mem[r*src1* + (3 ⊕ bs)]
  r*dest*<15:8> ← mem[r*src1* + (2 ⊕ bs)]
  r*dest*<23:16> ← mem[r*src1* + (1 ⊕ bs)]
  r*dest*<31:24> ← mem[r*src1* + (0 ⊕ bs)]
}

ATTRIBUTES

Function unit	dmem
Operation code	7
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32d ld32r ld32x st32 st32d h_st32d

DESCRIPTION

The `ld32` operation is a pseudo operation transformed by the scheduler into an `ld32d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ld32` operation loads the 32-bit memory value from the address contained in r*src1* and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 4, the result of `ld32` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The `ld32` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `ld32`.

The `ld32` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ld32` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32 r10 → r60`	r60 ← 0x84332211
r30 = 0, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r30 ld32 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r40 ld32 r20 → r80`	r80 ← 0x48665544
r50 = 0xd01	`ld32 r50 → r90`	r90 undefined, since 0xd01 is not a multiple of 4

ld32d

32-bit load with displacement

SYNTAX

```
[ IF rguard ] ld32d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + d + (3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + d + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + d + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + d + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	7
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32r ld32x st32 st32d h_st32d

DESCRIPTION

The ld32d operation loads the 32-bit memory value from the address computed by r*src1* + *d* and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4. If the memory address computed by r*src1* + *d* is not a multiple of 4, the result of ld32d is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32d operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32d.

The ld32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32d has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32d(4) r10 → r60	r60 ← 0x84332211
r30 = 0, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r30 ld32d(-8) r20 → r70	no change, since guard is false
r40 = 1, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r40 ld32d(-8) r20 → r80	r80 ← 0x48665544
r50 = 0xd01	ld32d(-8) r50 → r90	r90 undefined, since 0xd01 +(–8) is not a multiple of 4

32-bit load with index

ld32r

SYNTAX

```
[ IF rguard ] ld32r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + rsrc2 + (3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + rsrc2 + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + rsrc2 + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + rsrc2 + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	200
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32d ld32x st32 st32d h_st32d

DESCRIPTION

The ld32r operation loads the 32-bit memory value from the address computed by r*src1* + r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 4, the result of ld32r is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32r operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32r.

The ld32r operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32r has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r20 = 0x4, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32r r10 r20 → r80	r80 ← 0x84332211
r50 = 0, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r50 ld32r r40 r30 → r90	no change, since guard is false
r60 = 1, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r60 ld32r r40 r30 → r100	r100 ← 0x48665544
r50 = 0xd01, r30 = 0xfffffff8	ld32r r70 r30 → r110	r110 undefined, since 0xd01 +(–8) is not a multiple of 2

ld32x

32-bit load with scaled index

SYNTAX

```
[ IF rguard ] ld32x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 3
    else
        bs ← 0
    rdest<7:0> ← mem[rsrc1 + (4 × rsrc2) +(3 ⊕ bs)]
    rdest<15:8> ← mem[rsrc1 + (4 × rsrc2) + (2 ⊕ bs)]
    rdest<23:16> ← mem[rsrc1 + (4 × rsrc2) + (1 ⊕ bs)]
    rdest<31:24> ← mem[rsrc1 + (4 × rsrc2) + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	201
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32d ld32r st32 st32d h_st32d

DESCRIPTION

The ld32x operation loads the 32-bit memory value from the address computed by r*src1* + 4×r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + 4×r*src2* is not a multiple of 4, the result of ld32x is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32x operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32x.

The ld32x operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32x has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r30 = 0x1, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32x r10 r30 → r100`	r100 ← 0x84332211
r50 = 0, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r50 ld32x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r60 ld32x r40 r20 → r90`	r90 ← 0x48665544
r70 = 0xd01, r30 = 0x1	`ld32x r70 r30 → r110`	r110 undefined, since 0xd01 + 4×1 is not a multiple of 4

Logical shift left

pseudo-op for asl

lsl

SYNTAX

```
[ IF rguard ] lsl rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  n ← r*src2*<4:0>
  r*dest*<31:n> ← r*src1*<31–n:0>
  r*dest*<n–1:0> ← 0
}

ATTRIBUTES

Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

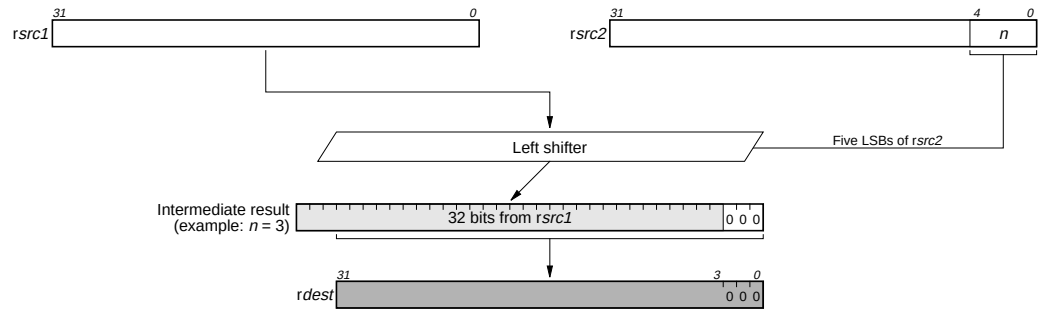
SEE ALSO

asl asli asr asri lsli lsr lsri rol roli

DESCRIPTION

The `lsl` operation is a pseudo operation that is transformed by the scheduler into an `asl` with the same arguments. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `lsl` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specify an unsigned shift amount, and rdest is set to r*src1* arithmetically shifted left by this amount. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `lsl` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`lsl r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 lsl r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 lsl r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`lsl r70 r40 → r120`	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	`lsl r80 r50 → r125`	r125 ← 0x80000000 (r50 is effectively equal to 0x1e)

lsli

Logical shift left immediate

pseudo-op for asli

SYNTAX

```
[ IF rguard ] lsli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:*n*> ← r*src1*<31–*n*:0>
    r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

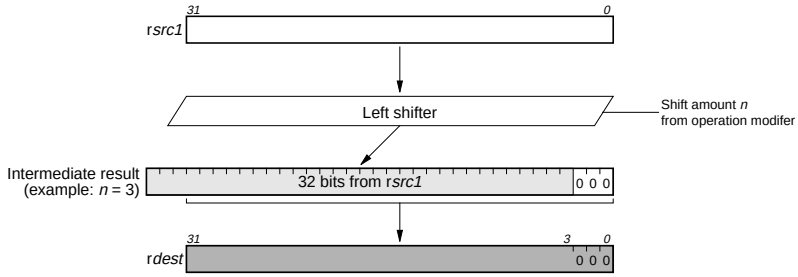
SEE ALSO

asl asli asr asri lsl lsr lsri rol roli

DESCRIPTION

The `lsli` operation is a pseudo operation that is transformed by the scheduler into an `asli` with the same argument and opcode modifier. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `lsli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* logically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `lsli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`lsli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 lsli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 lsli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`lsli(2) r70 → r120`	r120 ← 0xfffffff0
r80 = 0xe	`lsli(30) r80 → r125`	r125 ← 0x80000000

Logical shift right

lsr

SYNTAX

```
[ IF rguard ] lsr rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  n ← r*src2*<4:0>
  r*dest*<31:32–n> ← 0
  r*dest*<31–n:0> ← r*src1*<31:n>
}

ATTRIBUTES

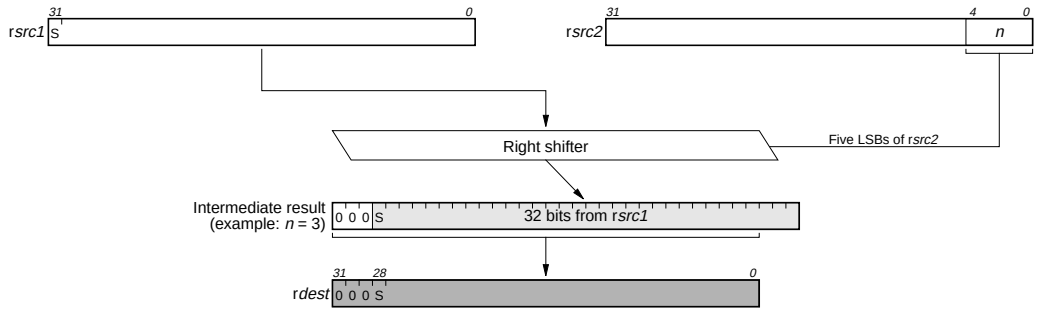
Function unit	shifter
Operation code	96
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsri rol roli

DESCRIPTION

As shown below, the `lsr` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specifies an unsigned shift amount, and r*src1* is arithmetically shifted right by this amount. Zeros fill vacated bits from the left.



The `lsr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	`lsr r30 r20 → r50`	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	`lsr r30 r42 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	`IF r10 lsr r30 r44 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	`IF r20 lsr r30 r44 → r80`	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	`lsr r40 r44 → r90`	r90 ← 0x08003000
r30 = 0x7008000f, r45 = 0x1f	`lsr r30 r45 → r100`	r100 ← 0x00000000

lsri

Logical shift right immediate

SYNTAX

```
[ IF rguard ] lsri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:32–*n*> ← 0
    r*dest*<31–*n*:0> ← r*src1*<31:*n*>
}

ATTRIBUTES

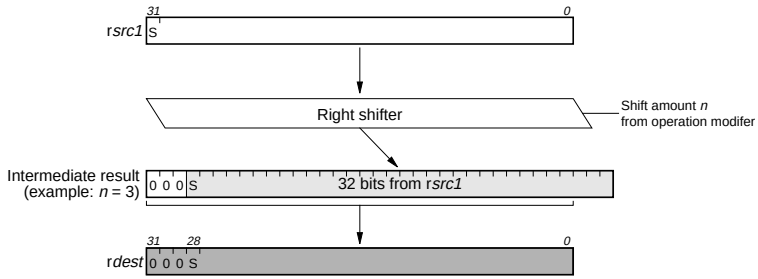
Function unit	shifter
Operation code	9
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsr rol roli

DESCRIPTION

As shown below, the `lsri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* logically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros fill vacated bits from the left.



The `lsri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`lsri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`lsri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 lsri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 lsri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`lsri(4) r40 → r90`	r90 ← 0x08003000
r30 = 0x7008000f	`lsri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`lsri(31) r40 → r110`	r110 ← 0x00000001

Merge least-significant byte

mergelsb

SYNTAX

```
[ IF rguard ] mergelsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← r*src2*<7:0>
r*dest*<15:8> ← r*src1*<7:0>
r*dest*<23:16> ← r*src2*<15:8>
r*dest*<31:24> ← r*src1*<15:8>
}

ATTRIBUTES

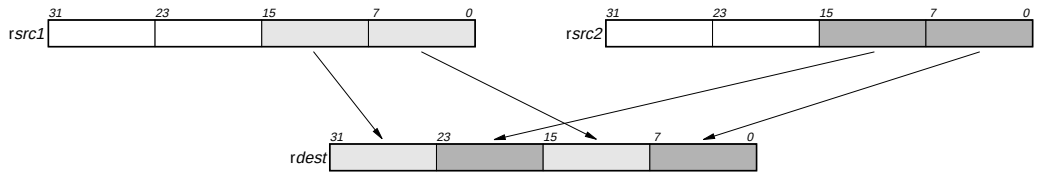
Function unit	alu
Operation code	57
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
packbytes mergemsb

DESCRIPTION

As shown below, the `mergelsb` operation interleaves the two pairs of least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The least-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the least-significant byte from r*src1* is packed into the second-least-significant byte or r*dest*; the second-least-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the second-least-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The `mergelsb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergelsb r30 r40 → r50`	r50 ← 0x56cc78dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergelsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergelsb r40 r30 → r70`	r70 ← 0xcc56dd78

mergemsb

Merge most-significant byte

SYNTAX

```
[ IF rguard ] mergemsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← r*src2*<23:15>
r*dest*<15:8> ← r*src1*<23:15>
r*dest*<23:16> ← r*src2*<31:24>
r*dest*<31:24> ← r*src1*<31:24>
}

ATTRIBUTES

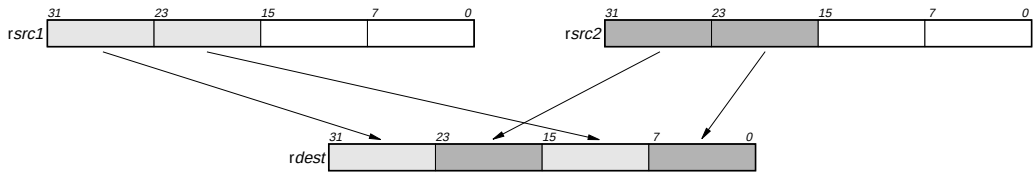
Function unit	alu
Operation code	58
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb packbytes mergelsb

DESCRIPTION

As shown below, the mergemsb operation interleaves the two pairs of most-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The second-most-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the second-most-significant byte from r*src1* is packed into the second-least-significant byte or r*dest*; the most-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the most-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The mergemsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergemsb r30 r40 → r50`	r50 ← 0x12aa34bb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergemsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergemsb r40 r30 → r70`	r70 ← 0xaa12bb34

No operation

nop

SYNTAX

nop

FUNCTION

No operation

ATTRIBUTES

Function unit	-
Operation code	-
Number of operands	-
Modifier	-
Modifier range	-
Latency	1
Issue slots	1-5

SEE ALSO

DESCRIPTION

The NOP operation does not change any DSPCPU state. It is mainly used to fill-up the empty issue slots. Only two bits are used to code the NOP operation.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	nop	No change in any regsiters

pack16lsb

Pack least-significant 16-bit halfwords

SYNTAX

```
[ IF rguard ] pack16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<15:0> ← r*src2*<15:0>
r*dest*<31:16> ← r*src1*<15:0>
}

ATTRIBUTES

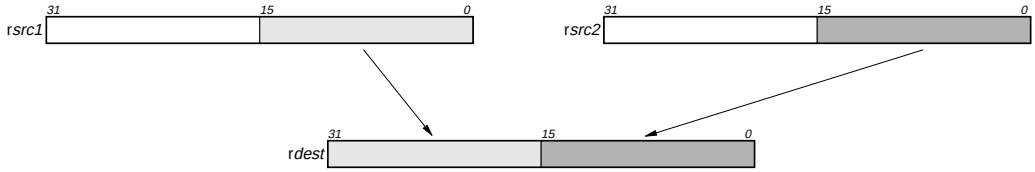
Function unit	alu
Operation code	53
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16msb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the pack16lsb operation packs the two least-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword or r*dest*.



The pack16lsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16lsb r30 r40 → r50`	r50 ← 0x5678ccdd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16lsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16lsb r40 r30 → r70`	r70 ← 0xccdd5678

Pack most-significant 16 bits

pack16msb

SYNTAX

```
[ IF rguard ] pack16msb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<15:0> ← r*src2*<31:16>
    r*dest*<31:16> ← r*src1*<31:16>
}

ATTRIBUTES

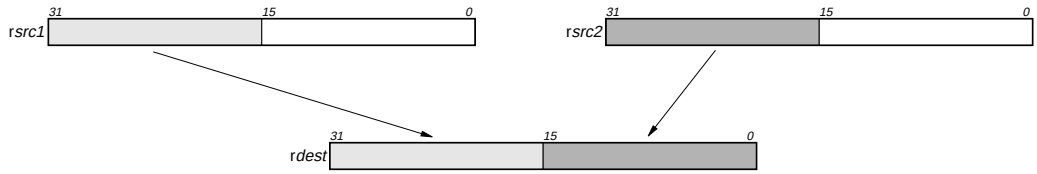
Function unit	alu
Operation code	54
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the pack16msb operation packs the two most-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword or r*dest*.



The pack16msb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	pack16msb r30 r40 → r50	r50 ← 0x1234aabb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	IF r10 pack16msb r40 r30 → r60	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	IF r20 pack16msb r40 r30 → r70	r70 ← 0xaabb1234

packbytes

Pack least-significant byte

SYNTAX

```
[ IF rguard ] packbytes rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← r*src2*<7:0>
r*dest*<15:8> ← r*src1*<7:0>
}

ATTRIBUTES

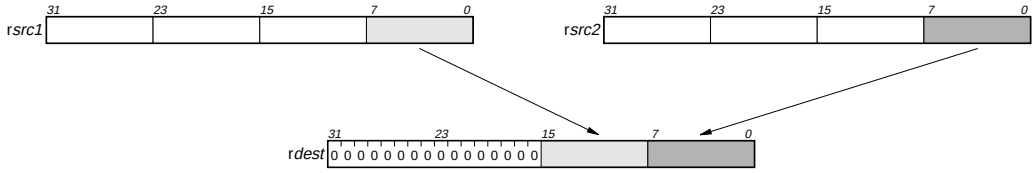
Function unit	alu
Operation code	52
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
mergelsb mergemsb

DESCRIPTION

As shown below, the packbytes operation packs the two least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The byte from r*src1* is packed into the second-least-significant byte of r*dest*; the byte from r*src2* is packed into the least-significant byte or r*dest*. The two most-significant bytes of r*dest* are filled with zeros.



The packbytes operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`packbytes r30 r40 → r50`	r50 ← 0x000078dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 packbytes r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 packbytes r40 r30 → r70`	r70 ← 0x0000dd78

prefetch

pseudo-op for prefd(0)

pref

SYNTAX

```
[ IF rguard ] pref rsrc1
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + 0) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref operation is a pseudo operation transformed by the scheduler into an prefd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The pref operation loads the one full cache block size of memory value from the address computed by ((rsrc1+0) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 pref r10`	since guard is false, pref operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000 and it is not guaranteed to be available in future generations of this product.

pref16x

prefetch with 16-bit scaled index

SYNTAX

```
[ IF rguard ] pref16x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (2 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	211
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref16x operation loads one full cache block from the main memory at the address computed by ((rsrc1+ (2 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. The data cache has hardware to simultaneously sustain two cache misses or prefetches. A pref16x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref16x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`pref16x r10 r12`	Loads a cache line for the address space from 0xabc0 to 0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	since guard is false, pref16x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000 and it is not guaranteed to be available in future generations of this product.

prefetch with 32-bit scaled index

pref32x

SYNTAX

```
[ IF rguard ] pref32x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (4 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	212
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x prefd prefr allocd allocr allocx

DESCRIPTION

The pref32x operation loads the one full cache block size of memory value from the address computed by ((rsrc1+ (4 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref32x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref32x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	since guard is false, pref32x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000 and it is not guaranteed to be available in future generations of this product.

prefd

prefetch with displacement

SYNTAX

```
[ IF rguard ] prefd(d) rsrc1
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + d) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefr allocd allocr allocx

DESCRIPTION

The prefd operation loads the one full cache block size of memory value from the address computed by ((rsrc1+d) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefd operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The prefd operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`prefd(0xd) r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 prefd(0xd) r10`	since guard is false, prefd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 prefd(ox1) r10`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000 and it is not guaranteed to be available in future generations of this product.

prefetch with index

prefr

SYNTAX

```
[ IF rguard ] prefr rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + rscr2) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	210
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd allocd allocr allocx

DESCRIPTION

The prefr operation loads the one full cache block size of memory value from the address computed by ((rsrc1+rscr2) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefr operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The `prefr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of r*guard* is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`prefr r10 r12`	Loads a cache line for the address space from 0xabc0 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 prefr r10 r12`	since guard is false, prefr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 prefr r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000 and it is not guaranteed to be available in future generations of this product.

quadavg

Unsigned byte-wise quad average

SYNTAX

```
[ IF rguard ] quadavg rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  temp ← (zero_ext8to32(r*src1*<7:0>) + zero_ext8to32(r*src2*<7:0>) + 1) / 2
  r*dest*<7:0> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<15:8>) + zero_ext8to32(r*src2*<15:8>) + 1) / 2
  r*dest*<15:8> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<23:16>) + zero_ext8to32(r*src2*<23:16>) + 1) / 2
  r*dest*<23:16> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<31:24>) + zero_ext8to32(r*src2*<31:24>) + 1) / 2
  r*dest*<31:24> ← temp<7:0>
}

ATTRIBUTES

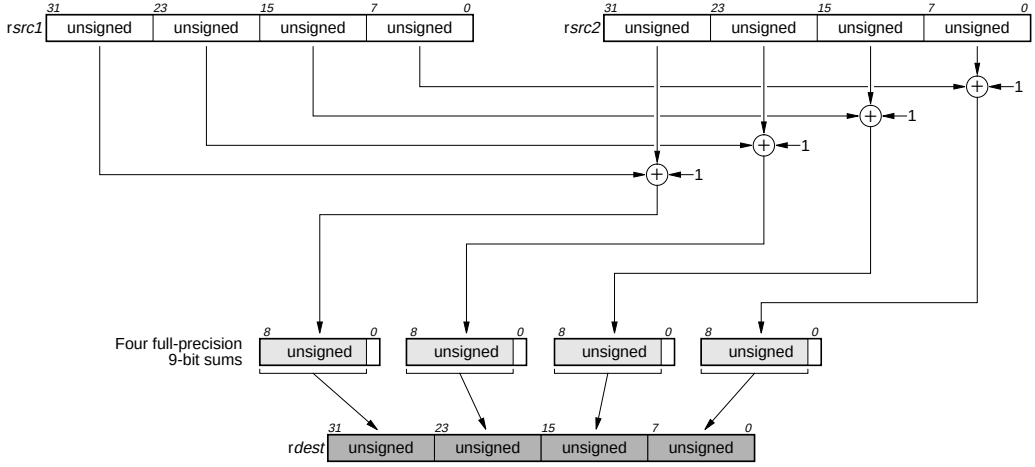
Function unit	dspalu
Operation code	73
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iavgonep dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the quadavg operation computes four separate averages of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The least-significant 8 bits of each average is written to the corresponding byte in r*dest*. No overflow or underflow detection is performed.



The quadavg operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xffffff02	quadavg r30 r40 → r50	r50 ← 0x81808008
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r10 quadavg r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r20 quadavg r60 r70 → r90	r90 ← 0x809c6480

Unsigned quad 8-bit multiply most significant

quadumulmsb

SYNTAX

```
[ IF rguard ] quadumulmsb rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	89
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

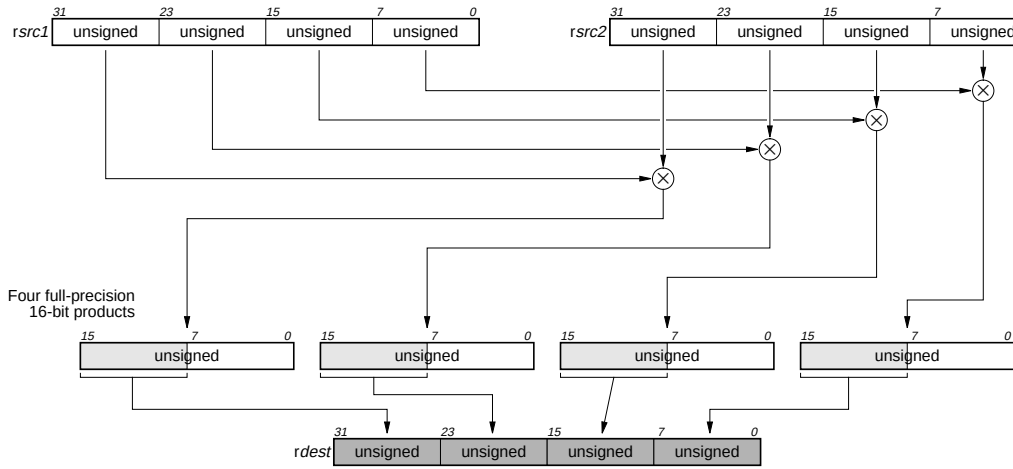
if r*guard* **then {**
  temp ← (zero_ext8to32(r*src1*<7:0>) × zero_ext8to32(r*src2*<7:0>))
  r*dest*<7:0> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<15:8>) × zero_ext8to32(r*src2*<15:8>))
  r*dest*<15:8> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<23:16>) × zero_ext8to32(r*src2*<23:16>))
  r*dest*<23:16> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<31:24>) × zero_ext8to32(r*src2*<31:24>))
  r*dest*<31:24> ← temp<15:8>
}

SEE ALSO

quadavg dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the quadumulmsb operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The most-significant 8 bits of each 16-bit product is written to the corresponding byte in r*dest*.



The quadumulmsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0210800e, r40 = 0xffffff02	quadumulmsb r30 r40 → r50	r50 ← 0x010f7f00
r10 = 0, r60 = 0x80ff1010, r70 = 0x80ff100f	IF r10 quadumulmsb r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x80ff1010, r70 = 0x80ff100f	IF r20 quadumulmsb r60 r70 → r90	r90 ← 0x40fe0100

rdstatus

Read data cache status bits

SYNTAX

```
[ IF rguard ] rdstatus(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   set_addr ← rsrc1 + d

      /* set_addr<10:6> selects set */

   rdest<9:0> ← dcache_LRU_set(set_addr)
   rdest<17:10> ← dcache_dirty_set(set_addr)
   rdest<31:17> ← 0
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	203
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

rdtag

DESCRIPTION

The `rdstatus` operation reads the LRU and dirty bits associated with a set in the data cache and writes these bits into the destination register r*dest*. The target set in the data cache is determined by bits 10..6 of the result of r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

The result of `rdstatus` contains LRU information in bits 9..0 and dirty-bit information in bits 17..10. All other bits of r*dest* are set to zero.

`rdstatus` requires two stall cycles to complete.

The dual-ported cache in TM1000 uses two separate copies of tag and status information. A `rdstatus` operation returns the LRU and dirty information stored in the cache port that corresponds to the operation slot in which the `rdstatus` operation is issued.

The `rdstatus` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`rdstatus(0) r30 → r60`	
r10 = 0	`IF r10 rdstatus(4) r40 → r70`	no change, since guard is false
r20 = 1	`IF r20 rdstatus(8) r50 → r80`	

Read data cache address tag

rdtag

SYNTAX

```
[ IF rguard ] rdtag(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    block_addr ← rsrc1 + d

        /* block_addr<13:11> selects element, block_addr<10:6> selects set */

    rdest<20:0> ← dcache_tag_block(block_addr)
    rdest<31:21> ← 0
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	202
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

rdstatus

DESCRIPTION

The `rdtag` operation reads the address tag associated with a block in the data cache and writes these bits into the destination register r*dest*. The target block in the data cache is determined by bits 13..6 of the result of r*src1* + *d*. Bits 10..6 of r*src1* + *d* select the cache set and 13..11 of r*src1* + *d* select the element within that set. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

`rdtag` writes the address tag for the selected block in bits 20..0 of r*dest*. All other bits of r*dest* are set to zero.

`rdtag` requires no stall cycles to complete.

The dual-ported cache in TM1000 uses two separate copies of tag and status information. A `rdtag` operation returns the address tag information stored in the cache port that corresponds to the operation slot in which the `rdtag` operation is issued.

The `rdtag` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`rdtag(0) r30 → r60`	
r10 = 0	`IF r10 rdtag(4) r40 → r70`	no change, since guard is false
r20 = 1	`IF r20 rdtag(8) r50 → r80`	

readdpc

Read destination program counter

SYNTAX

```
[ IF rguard ] readdpc → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← DPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	156
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`writedpc readspc ijmpf ijmpi ijmpt`

DESCRIPTION

The `readdpc` writes the current value of the DPC (Destination Program Counter) processor register to r*dest*.

Interruptible jumps write their target address to the DPC. If an interrupt or exception is taken at an interruptible jump, execution of the interrupted program can be resumed by jumping to the value contained in DPC. This operation can be used to save state before idling a task in a multi-tasking environment.

The `readdpc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
DPC = 0xbeebee	`readdpc → r100`	r100 ← 0xbeebee
r20 = 0, DPC = 0xabba	`IF r20 readdpc → r101`	no change, since guard is false
r21 = 1, DPC = 0xabba	`IF r21 readdpc → r102`	r102 ← 0xabba

Read program control and status word

readpcsw

SYNTAX

```
[ IF rguard ] readpcsw → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← PCSW
}

ATTRIBUTES

Function unit	fcomp
Operation code	158
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

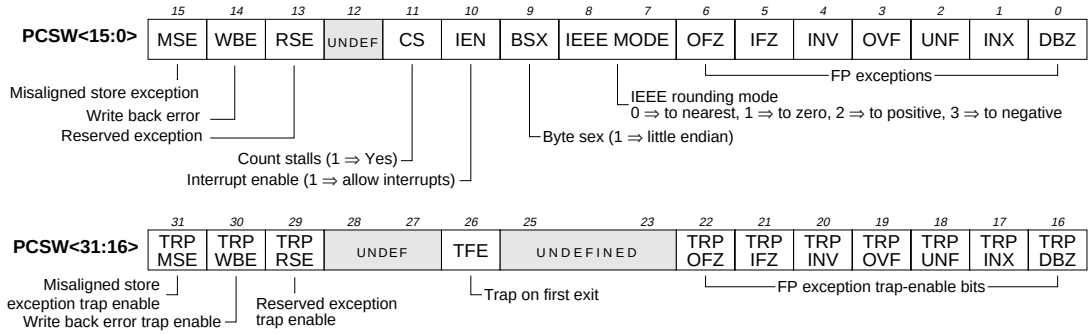
writepcsw

DESCRIPTION

The `readpcsw` writes the current value of the PCSW (Program Control and Status Word) processor register to r*dest*. The layout of PCSW is shown below.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, `readpcsw` can be used to determine current processor operating modes and what events have occurred; this operation can also be used to save state before idling a task in a multi-tasking environment.

The `readpcsw` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.



EXAMPLES

Initial Values	Operation	Result
PCSW = 0x80110642	`readpcsw → r100`	`r100` ← 0x80110642 (trap on MSE, INV and DBZ enabled, IEN=1 - interrupts enabled, BSX=1 - little endian mode of operation, OFZ=1 - a denormalized result was produced somewhere, INX=1 - an inexact result was produced somewhere)
r20 = 0, PCSW = 0x80000000	`IF r20 readpcsw → r101`	no change, since guard is false
r21 = 1, PCSW = 0x80000000	`IF r21 readpcsw → r102`	`r102` ← 0x80000000 (trap on MSE enabled)

readspc

Read source program counter

SYNTAX

```
[ IF rguard ] readspc → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← SPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	157
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

writespc readdpc ijmpf ijmpi ijmpt

DESCRIPTION

The `readspc` writes the current value of the SPC (Source Program Counter) processor register to r*dest*.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC allows an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.This operation can be used to save state before idling a task in a multi-tasking environment.

The `readspc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
SPC = 0xbeebee	`readspc → r100`	r100 ← 0xbeebee
r20 = 0, SPC = 0xabba	`IF r20 readspc → r101`	no change, since guard is false
r21 = 1, SPC = 0xabba	`IF r21 readspc → r102`	r102 ← 0xabba

Rotate left

rol

SYNTAX

```
[ IF rguard ] rol rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  n ← r*src2*<4:0>
  r*dest*<31:n> ← r*src1*<31–n:0>
  r*dest*<n–1:0> ← r*src1*<31:32–n>
}

ATTRIBUTES

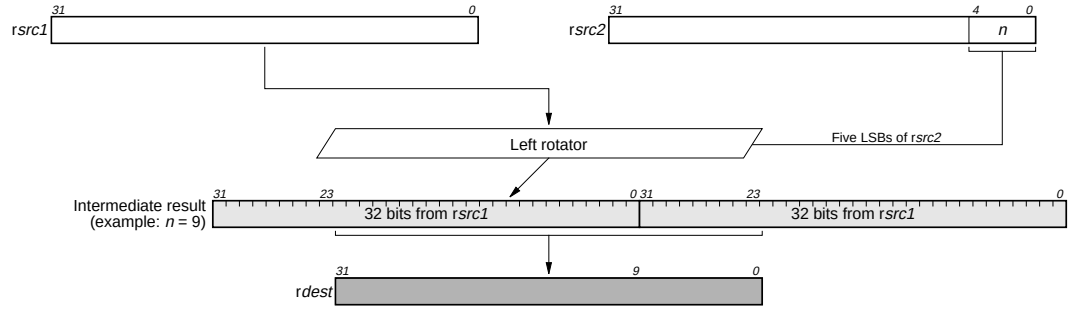
Function unit	shifter
Operation code	97
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

roli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `rol` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specify an unsigned rotate amount, and r*dest* is set to r*src1* rotated left by this amount. The most-significant n bits of r*src1*, where n is the rotate amount, appear as the least-significant n bits in r*dest*.



The `rol` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`rol r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 rol r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 rol r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`rol r70 r40 → r120`	r120 ← 0xffffff3
r80 = 0xe, r50 = 0xfffffffe	`rol r80 r50 → r125`	r125 ← 0x80000003 (r50 is effectively equal to 0x1e)

roli

Rotate left by immediate

SYNTAX

```
[ IF rguard ] roli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<31:*n*> ← r*src1*<31–*n*:0>
r*dest*<*n*–1:0> ← r*src1*<31:32–*n*>
}

ATTRIBUTES

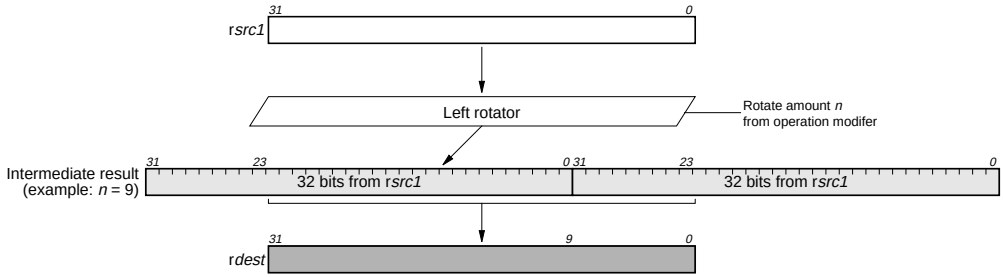
Function unit	shifter
Operation code	98
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

rol asl asli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `roli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* rotated left by *n* bits. The value of *n* must be between 0 and 31, inclusive. The most-significant *n* bits of r*src1* appear as the least-significant *n* bits in r*dest*.



The `roli` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`roli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 roli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 roli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`roli(2) r70 → r120`	r120 ← 0xfffffff3
r80 = 0xe	`roli(30) r80 → r125`	r125 ← 0x80000003

Sign extend 16 bits

sex16

SYNTAX

```
[ IF rguard ] sex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext16to32(r*src1*<15:0>)

ATTRIBUTES

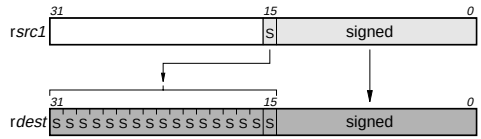
Function unit	alu
Operation code	51
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

zex16 sex8 zex8

DESCRIPTION

As shown below, the sex16 operation sign extends the least-significant 16bit halfword of the argument, r*src1*, to 32 bits and stores the result in r*dest*.



The sex16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of the guard is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	sex16 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 sex16 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 sex16 r40 → r100	r100 ← 0xffffff91
r50 = 0x00000091	sex16 r50 → r110	r110 ← 0x00000091

sex8

Sign extend 8 bits

pseudo-op for ibytesel

SYNTAX

```
[ IF rguard ] sex8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(r*src1*<7:0>)

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

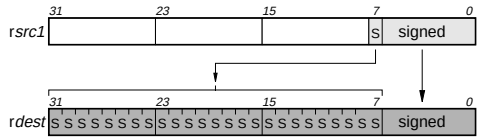
SEE ALSO

ibytesel sex16 zex8 zex16

DESCRIPTION

The `sex8` operation is a pseudo operation transformed by the scheduler into a `ibytesel` with r*src1* as the first argument and r0 (always contains 0) as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `sex8` operation sign extends the least-significant halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The `sex8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	`sex8 r30 → r60`	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	`IF r10 sex8 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	`IF r20 sex8 r40 → r100`	r100 ← 0xffffff91
r50 = 0x00000091	`sex8 r50 → r110`	r110 ← 0xffffff91

16-bit store

pseudo-op for h_st16d(0)

st16

SYNTAX

```
[ IF rguard ] st16 rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc1 + (1 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16d h_st16d st8 st8d st32 st32d

DESCRIPTION

The `st16` operation is a pseudo operation transformed by the scheduler into an `h_st16d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st16` operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1*. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st16` is misaligned (the memory address in r*src1* is not a multiple of 2), the result of `st16` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by `st16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st16 r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st16 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st16 r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

st16d

16-bit store with displacement

pseudo-op for h_st16d

SYNTAX

```
[ IF rguard ] st16d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16 h_st16d st8 st8d st32 st32d

DESCRIPTION

The st16d operation is a pseudo operation transformed by the scheduler into an h_st16d with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st16d operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st16d is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 2), the result of st16d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by st16d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The st16d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`st16d(2) r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 st16d(-4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 st16d(-4) r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

32-bit store

pseudo-op for h_st32d(0)

st32

SYNTAX

```
[ IF rguard ] st32 rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   mem[rsrc1 + (3 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + (2 ⊕ bs)] ← rsrc2<15:8>
   mem[rsrc1 + (1 ⊕ bs)] ← rsrc2<23:16>
   mem[rsrc1 + (0 ⊕ bs)] ← rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32d st16 st16d st8 st8d

DESCRIPTION

The st32 operation is a pseudo operation transformed by the scheduler into an h_st32d(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st32 operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1*. The *d* value is an opcode modifier and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st32 is misaligned (the memory address in r*src1* is not a multiple of 4), the result of st32 is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The st32 operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by st32.

The st32 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	st32 r10 r80	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	IF r50 st32 r20 r70	no change, since guard is false
r60 = 1, r30 = 0xd04, r70 = 0xaabbccdd	IF r60 st32 r30 r70	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

st32d

32-bit store with displacement

pseudo-op for h_st32d

SYNTAX

```
[ IF rguard ] st32d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   mem[rsrc1 + d + (3 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + d + (2 ⊕ bs)] ← rsrc2<15:8>
   mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<23:16>
   mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32 st16 st16d st8 st8d

DESCRIPTION

The st32d operation is a pseudo operation transformed by the scheduler into an h_st32d with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st32d operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st32d is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 4), the result of st32d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The st32d operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by st32d.

The st32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`st32d(4) r10 r80`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 st32d(-8) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 st32d(-8) r30 r70`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

8-bit store

pseudo-op for h_st8d(0)

st8

SYNTAX

```
[ IF rguard ] st8 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
mem[r*src1*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8d st16 st16d st32 st32d

DESCRIPTION

The `st8` operation is a pseudo operation transformed by the scheduler into an `h_st8d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st8` operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address in r*src1*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by `st8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8 r10 r80`	[0xd00] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8 r30 r70`	[0xd02] ← 0xdd

st8d

8-bit store with displacement

pseudo-op for h_st8d

SYNTAX

```
[ IF rguard ] st8d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
    mem[r*src1* + *d*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8 st16 st16d st32 st32d

DESCRIPTION

The st8d operation is a pseudo operation transformed by the scheduler into an h_st8d with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st8d operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address formed from the sum r*src1* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by st8d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The st8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8d(3) r30 r40`	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8d(-4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8d(-4) r30 r70`	[0xcfe] ← 0xdd

Select unsigned byte

ubytesel

SYNTAX

```
[ IF rguard ] ubytesel rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc2 = 0 then
      rdest ← zero_ext8to32(rsrc1<7:0>)
   else if rsrc2 = 1 then
      rdest ← zero_ext8to32(rsrc1<15:8>)
   else if rsrc2 = 2 then
      rdest ← zero_ext8to32(rsrc1<23:15>)
   else if rsrc2 = 3 then
      rdest ← zero_ext8to32(rsrc1<31:24>)
}
```

ATTRIBUTES

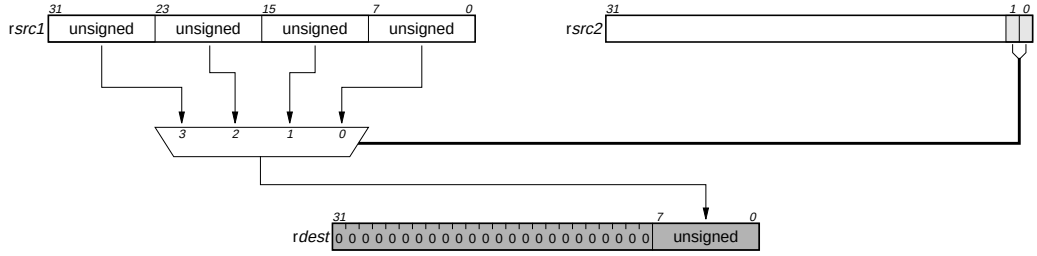
Function unit	alu
Operation code	55
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ibytesel sex8 packbytes

DESCRIPTION

As shown below, the `ubytesel` operation selects one byte from the argument, r*src1*, zero-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of `ubytesel` is undefined.



The `ubytesel` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	`ubytesel r30 r40 → r50`	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	`IF r10 ubytesel r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	`IF r20 ubytesel r60 r70 → r90`	r90 ← 0x000000cc
r100 = 0xffffff7f, r110 = 0	`ubytesel r100 r110 → r120`	r120 ← 0x0000007f

uclipi

Clip signed to unsigned

SYNTAX

```
[ IF rguard ] uclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← min(max(r*src1*, 0), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	75
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipu imin imax

DESCRIPTION

The `uclipi` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The argument r*src1* is considered a signed integer; r*src2* is considered an unsigned integer.

The `uclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipi r100 r110 → r120`	r120 ← 0

Clip unsigned to unsigned

uclipu

SYNTAX

```
[ IF rguard ] uclipu rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	76
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipi imin imax

DESCRIPTION

The `uclipu` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The arguments r*src1* and r*src2* are considered unsigned integers.

The `uclipu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipu r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipu r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipu r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipu r100 r110 → r120`	r120 ← 0x3fffff

ueql

Unsigned compare equal

pseudo-op for ieql

SYNTAX

```
[ IF rguard ] ueql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql ueqli igeq

DESCRIPTION

The `ueql` operation is a pseudo operation transformed by the scheduler into an `ieql` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `ueql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ueql r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ueql r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ueql r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ueql r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ueql r70 r70 → r110`	r110 ← 1

Unsigned compare equal with immediate

ueqli

SYNTAX

```
[ IF rguard ] ueqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	38
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieqli ueql igeqi

DESCRIPTION

The `ueqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ueqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ueqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ueqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ueqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ueqli(63) r40 → r100`	r100 ← 0
r60 = 0x07f	`ueqli(127) r60 → r120`	r120 ← 1

ufir16

Sum of products of unsigned 16-bit halfwords

SYNTAX

```
[ IF rguard ] ufir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← zero_ext16to32(r*src1*<31:16>) × zero_ext16to32(r*src2*<31:16>) +
zero_ext16to32(r*src1*<15:0>) × zero_ext16to32(r*src2*<15:0>)

ATTRIBUTES

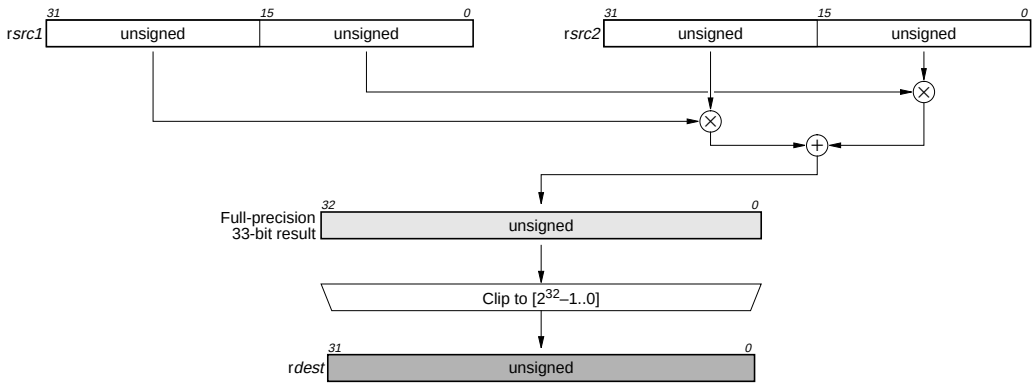
Function unit	dspmul
Operation code	94
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir16 ifir8ii ifir8ui ufir8uu

DESCRIPTION

As shown below, the ufir16 operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All halfwords are considered unsigned; thus, the intermediate products and the final sum of products are unsigned. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0xffffffff..0] before being written into r*dest*.



The ufir16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	ufir16 r30 r40 → r50	r50 ← 8
r10 = 0, r60 = 0x80000064, r70 = 0x00648000	IF r10 ufir16 r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x80000064, r70 = 0x00648000	IF r20 ufir16 r60 r70 → r90	r90 ← 0x00640000
r30 = 0x00020003, r70 = 0x00648000	ufir16 r30 r70 → r100	r100 ← 0x000180c8

Unsigned sum of products of unsigned bytes

ufir8uu

SYNTAX

```
[ IF rguard ] ufir8uu rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← zero_ext8to32(r*src1*<31:24>) × zero_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × zero_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × zero_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × zero_ext8to32(r*src2*<7:0>)

ATTRIBUTES

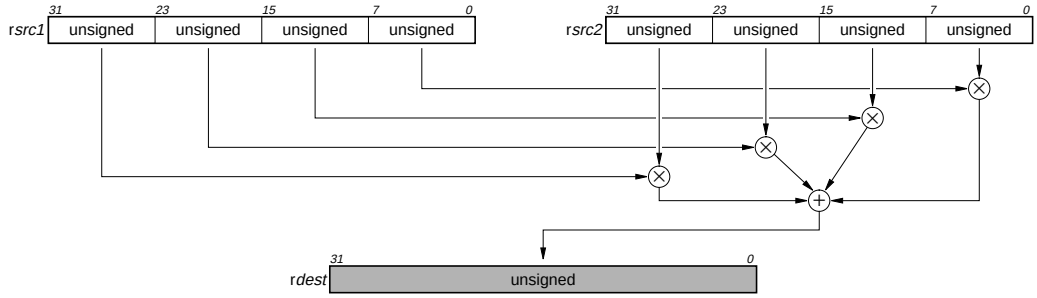
Function unit	dspmul
Operation code	90
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ui ifir8ii ifir16
ufir16

DESCRIPTION

As shown below, the `ufir8uu` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered unsigned. All computations are performed without loss of precision.



The `ufir8uu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ufir8uu r70 r30 → r90`	r90 ← 0x1efa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ufir8uu r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ufir8uu r80 r40 → r110`	r110 ← 0xf3c0
r50 = 0x80808080, r60 = 0xffffffff	`ufir8uu r50 r60 → r120`	r120 ← 0x1fe00

ufixieee

Convert floating-point to unsigned integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ufixieee rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (unsigned long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	123
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ifixrz ufixrz

DESCRIPTION

The `ufixieee` operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ufixieee` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufixieeeflags` operation computes the exception flags that would result from an individual `ufixieee`.

The `ufixieee` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ufixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixieee r40 → r110`	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	`ufixieee r45 → r112`	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ufixieee r50 → r115`	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	`ufixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ufixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ufixieee r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to unsigned integer using PCSW rounding mode

ufixieeeflags

SYNTAX

```
[ IF rguard ] ufixieeeflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	124
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

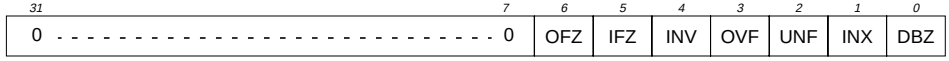
SEE ALSO

ufixieee ifixieeeflags ifixrzflags ufixrzflags

DESCRIPTION

The `ufixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixieeeflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixieeeflags r80 → r122`	r122 ← 0x10 (INV)

ufixrz

Convert floating-point to unsigned integer with round toward zero

SYNTAX

```
[ IF rguard ] ufixrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (unsigned long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	125
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ufixieee ifixrz

DESCRIPTION

The `ufixrz` operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ufixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufixrzflags` operation computes the exception flags that would result from an individual `ufixrz`.

The `ufixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ufixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrz r40 → r110`	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	`ufixrz r45 → r112`	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ufixrz r50 → r115`	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	`ufixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ufixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ufixrz r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to unsigned integer with round toward zero

ufixrzflags

SYNTAX

```
[ IF rguard ] ufixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	126
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

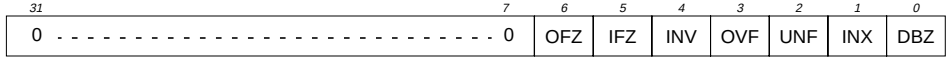
SEE ALSO

ufixrz ifixrzflags
ifixieeeflags
ufixieeeflags

DESCRIPTION

The `ufixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixrzflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixrzflags r80 → r122`	r122 ← 0x10 (INV)

ufloat

Convert unsigned integer to floating-point

SYNTAX

```
[ IF rguard ] ufloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((unsigned long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	127
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloat ifloatrz ufloatrz ifixieee ufloatflags

DESCRIPTION

The `ufloat` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ufloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatflags` operation computes the exception flags that would result from an individual `ufloat`.

The `ufloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloat r40 → r105`	r105 ← 0x4f800000 (4.294967296e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloat r50 → r115`	r115 ← 0x4f800000 (4.294967296e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloat r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

IEEE status flags from convert unsigned integer to floating-point

ufloatflags

SYNTAX

```
[ IF rguard ] ufloatflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	128
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

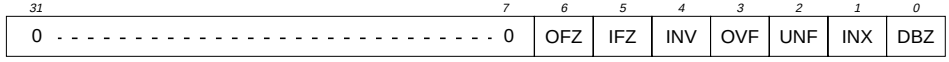
SEE ALSO

ufloat ifloatflags
ifloatrzflags
ufloatrzflags

DESCRIPTION

The `ufloatflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ufloatflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatflags r80 → r122`	r122 ← 0x02 (INX)

ufloatrz

Convert unsigned integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ufloatrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((unsigned long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	119
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloatrz ifloat ufloat ifixieee ufloatflags

DESCRIPTION

The `ufloatrz` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ufloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatrzflags` operation computes the exception flags that would result from an individual `ufloatrz`.

The `ufloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloatrz r40 → r105`	r105 ← 0x4f7fffff (4.294967040e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrz r50 → r115`	r115 ← 0x4f7fffff (4.294967040e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloatrz r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

IEEE status flags from convert unsigned integer to floating-point with rounding toward zero

ufloatrzflags

SYNTAX

```
[ IF rguard ] ufloatrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	120
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

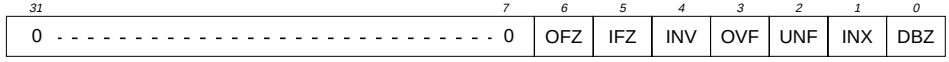
SEE ALSO

ufloatrz ifloatflags
ufloatflags ifloatrzflags

DESCRIPTION

The `ufloatrzflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ufloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatrzflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatrzflags r80 → r122`	r122 ← 0x02 (INX)

ugeq

Unsigned compare greater or equal

SYNTAX

```
[ IF rguard ] ugeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 >= (unsigned)rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ugeqi

DESCRIPTION

The ugeq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The ugeq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ugeq r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ugeq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ugeq r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ugeq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ugeq r70 r70 → r110`	r110 ← 1

Unsigned compare greater or equal with immediate

ugeqi

SYNTAX

```
[ IF rguard ] ugeqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 >= (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	36
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ugeq igeqi

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ugeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ugeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugeqi(127) r60 → r120`	r120 ← 1

ugtr

Unsigned compare greater

SYNTAX

```
[ IF rguard ] ugtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 > (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ugtri

DESCRIPTION

The `ugtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ugtr r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ugtr r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ugtr r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ugtr r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ugtr r70 r70 → r110`	r110 ← 0

Unsigned compare greater with immediate

ugtri

SYNTAX

```
[ IF rguard ] ugtri(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 > (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	34
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtri ugtr

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugtri(2) r30 → r80`	r80 ← 1
r30 = 3	`ugtri(3) r30 → r90`	r90 ← 0
r30 = 3	`ugtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugtri(127) r60 → r120`	r120 ← 1

uimm

Unsigned immediate

SYNTAX

`uimm(`*n*`)` → r*dest*

FUNCTION

r*dest* ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iimm

DESCRIPTION

The `uimm` operation writes the unsigned 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`uimm(2)` → `r10`	r10 ← 2
	`uimm(0x100)` → `r20`	r20 ← 0x100
	`uimm(0xfffc0000)` → `r30`	r30 ← 0xfffc0000

Unsigned 16-bit load

pseudo-op for uld16d(0)

uld16

SYNTAX

```
[ IF rguard ] uld16 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[rsrc1 + (1 ⊕ bs)]
    temp<15:8> ← mem[rsrc1 + (0 ⊕ bs)]
    rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16d ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The `uld16` operation is a pseudo operation transformed by the scheduler into an `uld16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld16` operation loads the 16-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `uld16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`uld16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 uld16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 uld16 r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16 r50 → r90`	r90 undefined (0xd01 is not a multiple of 2)

uld16d

Unsigned 16-bit load with displacement

SYNTAX

```
[ IF rguard ] uld16d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + d + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + d + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The uld16d operation loads the 16-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of uld16d is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by uld16d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The uld16d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and uld16d has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`uld16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 uld16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 uld16d(-4) r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16d(-4) r50 → r90`	r90 undefined (0xd01 +(–4) is not a multiple of 2)

Unsigned 16-bit load with index

uld16r

SYNTAX

```
[ IF rguard ] uld16r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + rsrc2 + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + rsrc2 + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	198
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 uld16d ild16d ild16r uld16x ild16x

DESCRIPTION

The `uld16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `uld16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`uld16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16r r40 r30 → r100`	r100 ← 0x00008433
r70 = 0xd01, r30 = 0xfffffffc	`uld16r r70 r30 → r110`	r110 undefined (0xd01 +(–4) is not a multiple of 2)

uld16x

Unsigned 16-bit load with scaled index

SYNTAX

```
[ IF rguard ] uld16x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + (2 × rsrc2) + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + (2 × rsrc2) + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	199
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 uld16d ild16d uld16r ild16r ild16x

DESCRIPTION

The `uld16x` operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of `uld16x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16x` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`uld16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16x r40 r20 → r90`	r90 ← 0x00008433
r70 = 0xd01, r30 = 1	`uld16x r70 r30 → r110`	r110 undefined (0xd01 + 2×1 is not a multiple of 2)

Unsigned 8-bit load

pseudo-op for uld8d(0)

uld8

SYNTAX

```
[ IF rguard ] uld8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8d ild8d uld8r ild8r

DESCRIPTION

The `uld8` operation is a pseudo operation transformed by the scheduler into an `uld8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld8` operation loads the 8-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`uld8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 uld8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 uld8 r20 → r80`	r80 ← 0x00000084
r50 = 0xd01, [0xd01] = 0x33	`uld8 r50 → r90`	r90 ← 0x00000033

uld8d

Unsigned 8-bit load with displacement

SYNTAX

```
[ IF rguard ] uld8d(d) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 ild8d uld8r ild8r

DESCRIPTION

The `uld8d` operation loads the 8-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier in the range –64 to 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`uld8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 uld8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 uld8d(-4) r20 → r80`	r80 ← 0x00000084
r50 = 0xd05, [0xd01] = 0x33	`uld8d(-4) r50 → r90`	r90 ← 0x00000033

Unsigned 8-bit load with index

uld8r

SYNTAX

```
[ IF rguard ] uld8r rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	194
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 uld8d ild8d ild8r

DESCRIPTION

The `uld8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`uld8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 uld8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 uld8r r40 r30 → r100`	r100 ← 0x00000084
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`uld8r r70 r30 → r110`	r110 ← 0x00000033

uleq

Unsigned compare less or equal

pseudo-op for ugeq

SYNTAX

```
[ IF rguard ] uleq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 <= (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq uleqi

DESCRIPTION

The `uleq` operation is a pseudo operation transformed by the scheduler into an `ugeq` with the arguments exchanged (`uleq`'s r*src1* is `ugeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `uleq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uleq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`uleq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 uleq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 uleq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`uleq r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`uleq r70 r70 → r110`	r110 ← 1

Unsigned compare less or equal with immediate

uleqi

SYNTAX

```
[ IF rguard ] uleqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 <= (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	43
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uleq ileqi

DESCRIPTION

The `uleqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uleqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uleqi(2) r30 → r80`	r80 ← 0
r30 = 3	`uleqi(3) r30 → r90`	r90 ← 1
r30 = 3	`uleqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uleqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uleqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`uleqi(127) r60 → r120`	r120 ← 0

ules

Unsigned compare less

pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] ules rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 < (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles ugtr

DESCRIPTION

The `ules` operation is a pseudo operation transformed by the scheduler into an `ugtr` with the arguments exchanged (`ules`'s r*src1* is `ugtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ules` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ules` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ules r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ules r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ules r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ules r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ules r70 r70 → r110`	r110 ← 0

Unsigned compare less with immediate

ulesi

SYNTAX

```
[ IF rguard ] ulesi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 < (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	41
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ules ilesi

DESCRIPTION

The `ulesi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ulesi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ulesi(2) r30 → r80`	r80 ← 0
r30 = 3	`ulesi(3) r30 → r90`	r90 ← 0
r30 = 3	`ulesi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ulesi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ulesi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ulesi(127) r60 → r120`	r120 ← 0

ume8ii

Unsigned sum of absolute values of signed 8-bit differences

SYNTAX

```
[ IF rguard ] ume8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← abs_val(sign_ext8to32(r*src1*<31:24>) – sign_ext8to32(r*src2*<31:24>)) +
abs_val(sign_ext8to32(r*src1*<23:16>) – sign_ext8to32(r*src2*<23:16>)) +
abs_val(sign_ext8to32(r*src1*<15:8>) – sign_ext8to32(r*src2*<15:8>)) +
abs_val(sign_ext8to32(r*src1*<7:0>) – sign_ext8to32(r*src2*<7:0>))

ATTRIBUTES

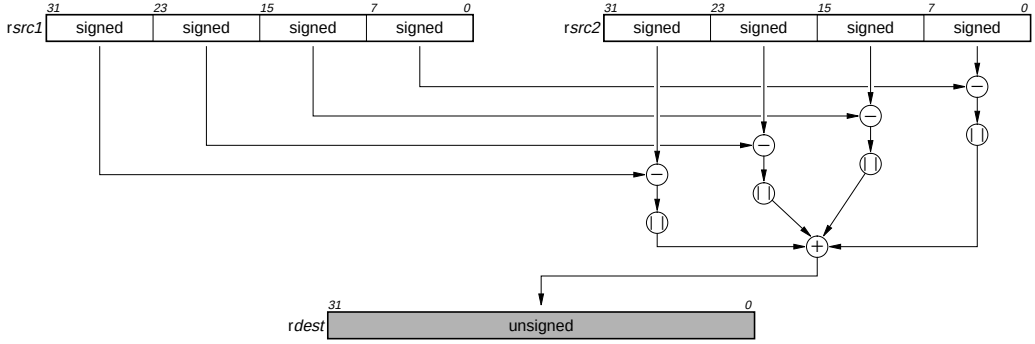
Function unit	dspalu
Operation code	64
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

ume8uu

DESCRIPTION

As shown below, the ume8ii operation computes four separate differences of the four pairs of corresponding signed 8-bit bytes of r*src1* and r*src2*; the absolute values of the four differences are summed, and the sum is written to r*dest*. All computations are performed without loss of precision.



The ume8ii operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8ii r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8ii r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8ii r90 r40 → r110`	r110 ← 0x190
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8ii r40 r90 → r120`	r120 ← 0x190
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8ii r50 r60 → r125`	r125 ← 0x3fc

Sum of absolute values of unsigned 8-bit differences

ume8uu

SYNTAX

```
[ IF rguard ] ume8uu rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← abs_val(zero_ext8to32(r*src1*<31:24>) – zero_ext8to32(r*src2*<31:24>)) +
abs_val(zero_ext8to32(r*src1*<23:16>) – zero_ext8to32(r*src2*<23:16>)) +
abs_val(zero_ext8to32(r*src1*<15:8>) – zero_ext8to32(r*src2*<15:8>)) +
abs_val(zero_ext8to32(r*src1*<7:0>) – zero_ext8to32(r*src2*<7:0>))

ATTRIBUTES

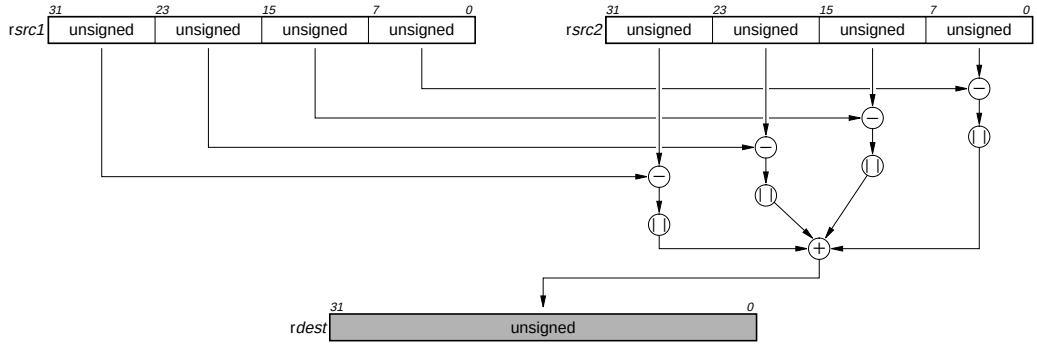
Function unit	dspalu
Operation code	26
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

ume8ii

DESCRIPTION

As shown below, the ume8uu operation computes four separate differences of the four pairs of corresponding unsigned 8-bit bytes of r*src1* and r*src2*. The absolute values of the four differences are summed and the result is written to r*dest*. All computations are performed without loss of precision.



The ume8uu operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	ume8uu r80 r30 → r100	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	IF r10 ume8uu r80 r30 → r70	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	IF r20 ume8uu r90 r40 → r110	r110 ← 0x70
r40 = 0x649c649c, r90 = 0x64649c9c	ume8uu r40 r90 → r120	r120 ← 0x70
r50 = 0x80808080, r60 = 0x7f7f7f7f	ume8uu r50 r60 → r125	r125 ← 0x4

umul

Unsigned multiply

SYNTAX

```
[ IF rguard ] umul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
r*dest* ← temp<31:0>

ATTRIBUTES

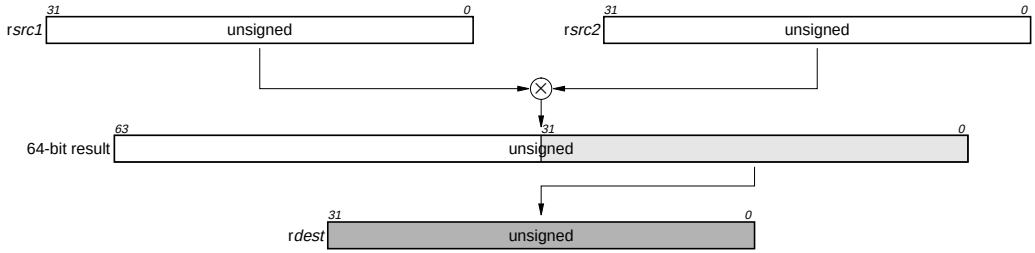
Function unit	ifmul
Operation code	138
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

imul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umul operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered unsigned integers. No overflow or underflow detection is performed.



The umul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	umul r60 r60 → r80	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umul r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	IF r20 umul r60 r30 → r90	r90 ← 0xf1100
r70 = 0x100, r40 = 0xffffff9c	umul r70 r40 → r100	r100 ← 0xffff9c00

Unsigned multiply, return most-significant 32 bits

umulm

SYNTAX

```
[ IF rguard ] umulm rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
  r*dest* ← temp<63:32>

ATTRIBUTES

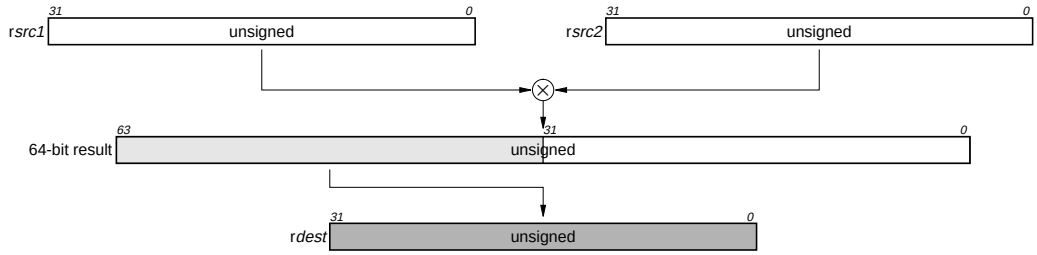
Function unit	ifmul
Operation code	140
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umulm operation computes the product r*src1*×r*src2* and writes the most-significant 32 bits of the 64-bit product into r*dest*. The operands are considered unsigned integers.



The umulm operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	umulm r60 r60 → r80	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umulm r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	IF r20 umulm r60 r30 → r90	r90 ← 0xf110f11
r70 = 0xffffff00, r40 = 0x100	umulm r70 r40 → r100	r100 ← 0xff

uneq

Unsigned compare not equal

pseudo-op for ineq

SYNTAX

```
[ IF rguard ] uneq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igtr

DESCRIPTION

The `uneq` operation is a pseudo operation transformed by the scheduler into an `ineq`. (Note: pseudo operations cannot be used in assembly source files.)

The `uneq` operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The `uneq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`uneq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	`IF r10 uneq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 uneq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`uneq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`uneq r70 r70 → r110`	r110 ← 0

Unsigned compare not equal with immediate

uneqi

SYNTAX

```
[ IF rguard ] uneqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 != (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	40
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uneq ineqi

DESCRIPTION

The `uneqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uneqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uneqi(2) r30 → r80`	r80 ← 1
r30 = 3	`uneqi(3) r30 → r90`	r90 ← 0
r30 = 3	`uneqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uneqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uneqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`uneqi(127) r60 → r120`	r120 ← 1

writedpc

Write destination program counter

SYNTAX

```
[ IF rguard ] writedpc rsrc1
```

FUNCTION

if r*guard* **then {**
DPC ← r*src1*
}

ATTRIBUTES

Function unit	fcomp
Operation code	160
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`readdpc writespc ijmpf ijmpi ijmpt`

DESCRIPTION

The `writedpc` copies the value of r*src1* to the DPC (Destination Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a `writedpc`) coincide, the software update takes precedence.

Interruptible jumps write their target address to the DPC. The value of DPC is intended to be used by an exception-handling routine as a jump address to resume execution of the program that was running before the exception was taken.

The `writedpc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of DPC. If the LSB of r*guard* is 1, DPC is written; otherwise, DPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	`writedpc r30`	DPC ← 0xbeebee
r20 = 0, r31 = 0xabba	`IF r20 writedpc r31`	no change, since guard is false
r21 = 1, r31 = 0xabba	`IF r21 writedpc r31`	DPC ← 0xabba

Write program control and status word

writepcsw

SYNTAX

```
[ IF rguard ] writepcsw rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
PCSW ← (PCSW & ~r*src2*) | (r*src1* & r*src2*)
}

ATTRIBUTES

Function unit	fcomp
Operation code	161
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

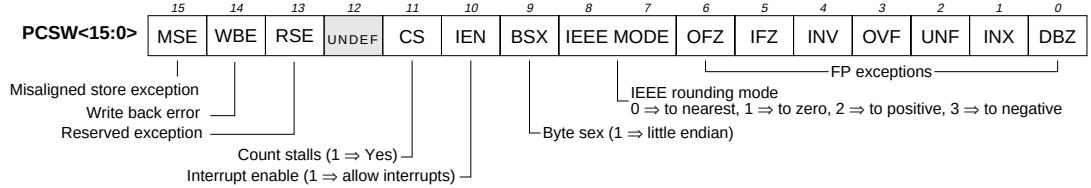
readpcsw fadd faddflags ijmpf cycles hicycles

DESCRIPTION

The `writepcsw` copies the value of r*src1* to the PCSW (Program Control and Status Word) processor register using rsrc2 as a mask. A bit in PCSW is affected by `writepcsw` only if the corresponding bit in rsrc2 is set to 1; the value of any bit in PCSW with a corresponding 0-bit in rsrc2 will not be changed by `writepcsw`. Whenever a hardware update (e.g., when a floating-point exception is raised) and a software update (through a `writepcsw`) coincide, the PCSW bits currently being updated by hardware will reflect the hardware-determined value while the bits not being affected by hardware will reflect the value in the `writepcsw` operand. The layout of PCSW is shown below. The programmer should take care not to alter UNDEF fields in the PCSW.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, `writepcsw` can be used to effect changes in some aspects of processor operation and to clear fields that record events; this operation can also be used to restore state before resuming an idled task in a multi-tasking environment.

The `writepcsw` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of PCSW. If the LSB of r*guard* is 1, PCSW is written; otherwise, PCSW is unchanged.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x100, r40 = 0x180	`writepcsw r30 r40`	PCSW.IEEE MODE = to positive infinity
r20 = 0, r50 = 0x0, r60 = 0x400	`IF r20 writepcsw r50 r60`	no change, since guard is false
r21 = 1, r50 = 0x0, r60 = 0x400	`IF r21 writepcsw r50 r60`	PCSW.IEN = 0 (disable interrupts)
r70 = 0x80110000, r80 = 0xffff0000	`writepcsw r70 r80`	enable trap on MSE, INV and DBZ exclusively

writespc

Write source program counter

SYNTAX

```
[ IF rguard ] writespc rsrc1
```

FUNCTION

if r*guard* **then**
SPC ← r*src1*

ATTRIBUTES

Function unit	fcomp
Operation code	159
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`readspc writedpc ijmpf ijmpi ijmpt`

DESCRIPTION

The `writespc` copies the value of r*src1* to the SPC (Source Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a `writespc`) coincide, the software update takes precedence.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC is intended to allow an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.

The `writespc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of SPC. If the LSB of r*guard* is 1, SPC is written; otherwise, SPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	`writespc r30`	SPC ← 0xbeebee
r20 = 0, r31 = 0xabba	`IF r20 writespc r31`	no change, since guard is false
r21 = 1, r31 = 0xabba	`IF r21 writespc r31`	SPC ← 0xabba

Zero extend 16 bits

pseudo-op for pack16lsb

zex16

SYNTAX

```
[ IF rguard ] zex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext16to32(r*src1*<15:0>)

ATTRIBUTES

Function unit	alu
Operation code	53
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

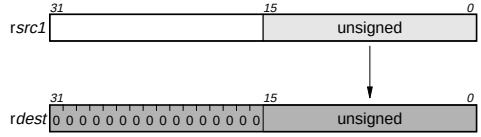
SEE ALSO

sex16 sex8 zex8

DESCRIPTION

The `zex16` operation is a pseudo operation transformed by the scheduler into a `pack16lsb` with 0 as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `zex16` operation zero extends the least-significant 16-bit halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The `zex16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	`zex16 r30 → r60`	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	`IF r10 zex16 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	`IF r20 zex16 r40 → r100`	r100 ← 0x0000ff91
r50 = 0x00000091	`zex16 r50 → r110`	r110 ← 0x00000091

zex8

Zero extend 8 bits

pseudo-op for ubytesel

SYNTAX

```
[ IF rguard ] zex8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(r*src1*<7:0>)

ATTRIBUTES

Function unit	alu
Operation code	55
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

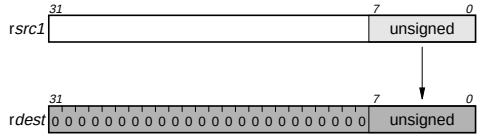
SEE ALSO

ubytesel sex16 sex8 zex16

DESCRIPTION

The zex8 operation is a pseudo operation transformed by the scheduler into a ubytesel with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the zex8 operation zero extends the least-significant byte of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The zex8 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	zex8 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 zex8 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 zex8 r40 → r100	r100 ← 0x00000091
r50 = 0x00000091	zex8 r50 → r110	r110 ← 0x00000091

MMIO Register Summary Chapter B

by Gert Slavenburg

B.1 MMIO REGISTERS

The following table lists all the MMIO registers implemented in TM1000. The registers are grouped according to the unit to which they belong.

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
DSPCPU Registers				
DRAM_BASE	10 0000	R/W	R/W	Start of DRAM address aperture
DRAM_LIMIT	10 0004	R/W	R/W	End of DRAM address aperture
MMIO_BASE	10 0400	R/W	R/W	Start of 2-MB MMIO-register address aperture
EXCVEC	10 0800	R/W	R/W	Interrupt vector (handler start address) for exceptions
ISETTING0	10 0810	R/W	R/W	Interrupt mode & priority settings for sources 24–31
ISETTING1	10 0814	R/W	R/W	Interrupt mode & priority settings for sources 26–23
ISETTING2	10 0818	R/W	R/W	Interrupt mode & priority settings for sources 8–15
ISETTING3	10 081c	R/W	R/W	Interrupt mode & priority settings for sources 0–7
IPENDING	10 0820	R/W	R/W	Interrupt-pending status bit for all 32 sources
ICLEAR	10 0824	R/W	R/W	Interrupt-clear bit for all 32 sources
IMASK	10 0828	R/W	R/W	Interrupt-mask bit for all 32 sources
INTVEC0	10 0880	R/W	R/W	Interrupt vector (handler start address) for source 0
INTVEC1	10 0884	R/W	R/W	Interrupt vector (handler start address) for source 1
INTVEC2	10 0888	R/W	R/W	Interrupt vector (handler start address) for source 2
INTVEC3	10 088c	R/W	R/W	Interrupt vector (handler start address) for source 3
INTVEC4	10 0890	R/W	R/W	Interrupt vector (handler start address) for source 4
INTVEC5	10 0894	R/W	R/W	Interrupt vector (handler start address) for source 5
INTVEC6	10 0898	R/W	R/W	Interrupt vector (handler start address) for source 6
INTVEC7	10 089c	R/W	R/W	Interrupt vector (handler start address) for source 7
INTVEC8	10 08a0	R/W	R/W	Interrupt vector (handler start address) for source 8
INTVEC9	10 08a4	R/W	R/W	Interrupt vector (handler start address) for source 9
INTVEC10	10 08a8	R/W	R/W	Interrupt vector (handler start address) for source 10
INTVEC11	10 08ac	R/W	R/W	Interrupt vector (handler start address) for source 11
INTVEC12	10 08b0	R/W	R/W	Interrupt vector (handler start address) for source 12
INTVEC13	10 08b4	R/W	R/W	Interrupt vector (handler start address) for source 13
INTVEC14	10 08b8	R/W	R/W	Interrupt vector (handler start address) for source 14
INTVEC15	10 08bc	R/W	R/W	Interrupt vector (handler start address) for source 15
INTVEC16	10 08c0	R/W	R/W	Interrupt vector (handler start address) for source 16
INTVEC17	10 08c4	R/W	R/W	Interrupt vector (handler start address) for source 17
INTVEC18	10 08c8	R/W	R/W	Interrupt vector (handler start address) for source 18
INTVEC19	10 08cc	R/W	R/W	Interrupt vector (handler start address) for source 19
INTVEC20	10 08d0	R/W	R/W	Interrupt vector (handler start address) for source 20

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
INTVEC21	10 08d4	R/W	R/W	Interrupt vector (handler start address) for source 21
INTVEC22	10 08d8	R/W	R/W	Interrupt vector (handler start address) for source 22
INTVEC23	10 08dc	R/W	R/W	Interrupt vector (handler start address) for source 23
INTVEC24	10 08e0	R/W	R/W	Interrupt vector (handler start address) for source 24
INTVEC25	10 08e4	R/W	R/W	Interrupt vector (handler start address) for source 25
INTVEC26	10 08e8	R/W	R/W	Interrupt vector (handler start address) for source 26
INTVEC27	10 08ec	R/W	R/W	Interrupt vector (handler start address) for source 27
INTVEC28	10 08f0	R/W	R/W	Interrupt vector (handler start address) for source 28
INTVEC29	10 08f4	R/W	R/W	Interrupt vector (handler start address) for source 29
INTVEC30	10 08f8	R/W	R/W	Interrupt vector (handler start address) for source 30
INTVEC31	10 08fc	R/W	R/W	Interrupt vector (handler start address) for source 31
TIMER1_TMODULUS	10 0c00	R/W	R/W	Contains: (maximum count value for timer 1) + 1
TIMER1_TVALUE	10 0c04	R/W	R/W	Current value of timer 1 counter
TIMER1_TCTL	10 0c08	R/W	R/W	Timer 1 control (prescale value, source select, run bit)
TIMER2_TMODULUS	10 0c20	R/W	R/W	Contains: (maximum count value for timer 2) + 1
TIMER2_TVALUE	10 0c24	R/W	R/W	Current value of timer 2 counter
TIMER2_TCTL	10 0c28	R/W	R/W	Timer 2 control (prescale value, source select, run bit)
TIMER3_TMODULUS	10 0c40	R/W	R/W	Contains: (maximum count value for timer 3) + 1
TIMER3_TVALUE	10 0c44	R/W	R/W	Current value of timer 3 counter
TIMER3_TCTL	10 0c48	R/W	R/W	Timer 3 control (prescale value, source select, run bit)
SYSTIMER_TMODULUS	10 0c60	R/W	R/W	Contains: (maximum count value for system timer) + 1
SYSTIMER_TVALUE	10 0c64	R/W	R/W	Current value of system timer/counter
SYSTIMER_TCTL	10 0c68	R/W	R/W	System timer control (prescale value, source select, run bit)
BICTL	10 1000	R/W	R/W	Instruction breakpoint control
BINSTLOW	10 1004	R/W	R/W	Start of address range that causes instruction breakpoints
BINSTHIGH	10 1008	R/W	R/W	End of address range that causes instruction breakpoints
BDCTL	10 1020	R/W	R/W	Data breakpoint control
BDATAALOW	10 1030	R/W	R/W	Start of address range that causes data breakpoints
BDATAAHIGH	10 1034	R/W	R/W	End of address range that causes data breakpoints
BDATAVAL	10 1038	R/W	R/W	Compare value for data breakpoints
BDATAMASK	10 103c	R/W	R/W	Compare mask for compare value for data breakpoints
Cache And Memory System				
DRAM_CACHEABLE_LIMIT	10 0008	R/W	R/W	Start of non-cacheable region in DRAM
MEM_EVENTS	10 000c	R/W	R/W	Selects two cache-related events for counting
DC_LOCK_CTL	10 0010	R/W	R/W	Enable bit for data-cache locking, also PCI hole disable
DC_LOCK_ADDR	10 0014	R/W	R/W	Start of address range that will be locked into the data cache
DC_LOCK_SIZE	10 0018	R/W	R/W	Size of address range that will be locked into the data cache
DC_PARAMS	10 001c	R/—	R/—	Data-cache geometry (blocksize, associativity, # of sets)
IC_PARAMS	10 0020	R/—	R/—	Instruction-cache geometry (blocksize, assoc., # of sets)
MM_CONFIG	10 0100	R/—	R/—	DRAM settings (rank size, bus width, refresh interval)
ARB_BW_CTL	10 0104	R/W	R/W	Internal bus arbitration control (bandwidth allocation)
ARB_RAISE	10 010C	R/W	R/W	Arbiter Priority Raising timer
POWER_DOWN	10 0108	R/W	R/W	Write to this register to initiate power down
IC_LOCK_CTL	10 0210	R/W	R/W	Enable bit for instruction-cache locking
IC_LOCK_ADDR	10 0214	R/W	R/W	Start of address range that will be locked into the instruction cache

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
IC_LOCK_SIZE	10 0218	R/W	R/W	Size of address range that will be locked into the instruction cache
PLL_RATIOS	10 0300	R/—	R/—	Sets ratios of external and internal clock frequencies
Video In				
VI_STATUS	10 1400	R/—	R/—	Status of video-in unit
VI_CTL	10 1404	R/W	R/W	Sets operation and interrupt modes for video in
VI_CLOCK	10 1408	R/W	R/W	Sets clock source (internal/external), frequency
VI_CAP_START	10 140c	R/W	R/W	Sets capture start x and y offsets
VI_CAP_SIZE	10 1410	R/W	R/W	Sets capture size width and height
VI_BASE1 VI_Y_BASE_ADR	10 1414	R/W	R/W	Capture modes: sets base address of Y-value array Message/raw modes: sets base address of buffer 1
VI_BASE2 VI_U_BASE_ADR	10 1418	R/W	R/W	Capture modes: sets base address of U-value array Message/raw modes: sets base address of buffer 2
VI_SIZE VI_V_BASE_ADR	10 141c	R/W	R/W	Capture modes: sets base address of V-value array Message/raw modes: sets size of buffers
VI_UV_DELTA	10 1420	R/W	R/W	Capture modes: address delta for adjacent U, V lines
VI_Y_DELTA	10 1424	R/W	R/W	Capture modes: address delta for adjacent Y lines
Video Out				
VO_STATUS	10 1800	R/—	R/—	Status of video-out unit
VO_CTL	10 1804	R/W	R/W	Sets operation and interrupt modes for video out
VO_CLOCK	10 1808	R/W	R/W	Sets video-out clock frequency
VO_FRAME	10 180c	R/W	R/W	Sets frame parameters (preset, start, length)
VO_FIELD	10 1810	R/W	R/W	Sets field parameters (overlap, field-1 line, field-2 line)
VO_LINE	10 1814	R/W	R/W	Sets field parameters (starting pixel, frame width)
VO_IMAGE	10 1818	R/W	R/W	Sets image parameters (height, width)
VO_YTHR	10 181c	R/W	R/W	Sets threshold for YTR interrupt, image v/h offsets
VO_OLSTART	10 1820	R/W	R/W	Sets overlay image parameters (start line/pixel, alpha)
VO_OLHW	10 1824	R/W	R/W	Sets overlay image parameters (height, width)
VO_YADD	10 1828	R/W	R/W	Sets Y-component/buffer-1 starting address
VO_UADD	10 182c	R/W	R/W	Sets U-component/buffer-2 starting address
VO_VADD	10 1830	R/W	R/W	Sets V-component address/buffer-1 length
VO_OLADD	10 1834	R/W	R/W	Sets overlay image address/buffer-2 length
VO_VUF	10 1838	R/W	R/W	Sets start-of-line-to-start-of-line address offsets (U, V)
VO_YOLF	10 183c	R/W	R/W	Sets start-of-line-to-start-of-line addr. offsets (Y, overlay)
Audio In				
AI_STATUS	10 1c00	R/—	R/—	Status of audio-in unit
AI_CTL	10 1c04	R/W	R/W	Sets operation and interrupt modes for audio in
AI_SERIAL	10 1c08	R/W	R/W	Sets clock ratios and internal/external clock generation
AI_FRAMING	10 1c0c	R/W	R/W	Sets format of serial data stream
AI_FREQ	10 1c10	R/W	R/W	Sets AI_OSCLK frequency
AI_BASE1	10 1c14	R/W	R/W	Sets base address of buffer 1
AI_BASE2	10 1c18	R/W	R/W	Sets base address of buffer 2
AI_SIZE	10 1c1c	R/W	R/W	Sets number of samples in buffers
Audio Out				
AO_STATUS	10 2000	R/—	R/—	Status of audio-out unit
AO_CTL	10 2004	R/W	R/W	Sets operation and interrupt modes for audio out
AO_SERIAL	10 2008	R/W	R/W	Sets cock ratios and internal/external clock generation

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
AO_FRAMING	10 200c	R/W	R/W	Sets format of serial data stream
AO_FREQ	10 2010	R/W	R/W	Set AO_OSCLK frequency
AO_BASE1	10 2014	R/W	R/W	Sets base address of buffer 1
AO_BASE2	10 2018	R/W	R/W	Sets base address of buffer 2
AO_SIZE	10 201c	R/W	R/W	Sets number of samples in buffers
AO_CC	10 2020	R/W	R/W	Codec control field values
AO_CFC	10 2024	R/W	R/W	Codec Frame Control
PCI Interface				
BIU_STATUS	10 3004	R/—	R/—	Status of PCI interface (done/busy bits, error bits)
BIU_CTL	10 3008	R/W	R/W	Sets operation and interrupt modes for PCI
PCI_ADR	10 300c	R/W	—/—	Holds address for DSPCPU PCI access
PCI_DATA	10 3010	R/W	—/—	Holds data for DSPCPU PCI access
CONFIG_ADR	10 3014	R/W	R/W	Holds address for configuration access
CONFIG_DATA	10 3018	R/W	R/W	Holds data for configuration access
CONFIG_CTL	10 301c	R/W	R/W	Sets read/write, bus number for configuration access
IO_ADR	10 3020	R/W	R/W	Holds address for I/O access
IO_DATA	10 3024	R/W	R/W	Holds data for I/O access
IO_CTL	10 3028	R/W	R/W	Sets read/write, byte-enable for I/O access
SRC_ADR	10 302c	R/W	R/W	Holds source address for DMA operation
DEST_ADR	10 3030	R/W	R/W	Holds destination address for DMA operation
DMA_CTL	10 3034	R/W	R/W	Sets read/write, transfer length for DMA operation
INT_CTL	10 3038	R/W	R/W	Controls interrupt system
JTAG				
JTAG_DATA_IN	10 3800	R/W	R/W	JTAG data input buffer
JTAG_DATA_OUT	10 3804	R/W	R/W	JTAG data output buffer
JTAG_CTL	10 3808	R/W	R/W	JTAG control
Image Co-Processor				
ICP_MPC	10 2400	R/W	R/W	MicroProgram Counter
ICP_MIR	10 2404	R/W	R/W	Micro Instruction Register
ICP_DP	10 2408	R/W	R/W	Data Pointer
ICP_DR	10 2410	R/W	R/W	Data Register
ICP_SR	10 2414	R/W	R/W	Status Register
VLD Co-Processor				
VLD_COMMAND	10 2800	R/W	R/W	Next action to be taken by VLD
VLD_SR	10 2804	R/—	R/—	Bitstream shift register
VLD_QS	10 2808	R/W	R/W	Quantization Scale Code
VLD_PI	10 280C	R/W	R/W	Picture layer Information
VLD_STATUS	10 2810	R/W	R/W	Status Register
VLD_IMASK	10 2814	R/W	R/W	Controls which status bits cause VLD interrupts
VLD_CTL	10 2818	R/W	R/W	Control Register
VLD_BIT_ADR	10 281C	R/W	R/W	Current Bitstream Read Address

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
VLD_BIT_CNT	10 2820	R/W	R/W	Bitstream remaining byte count
VLD_MBH_ADR	10 2824	R/W	R/W	Macro Block Header output address
VLD_MBH_CNT	10 2828	R/W	R/W	Macro Block Header output remaining count
VLD_RL_ADR	10 282C	R/W	R/W	Run/Length output address
VLD_RL_CNT	10 2830	R/W	R/W	Run/Length output remaining count
I^2C Interface				
IIC_AR	10 3400	R/W	R/W	Address, Byte count and Direction
IIC_DR	10 3404	R/W	R/W	Data Register
IIC_STATUS	10 3408	R/—	R/—	Status Register
IIC_CTL	10 340C	R/W	R/W	Control Register
Synchronous Serial Interface				
SSI_CTL	10 2C00	R/W	R/W	Control Register
SSI_CSR	10 2C04	R/W	R/W	Additional Control and Status register
SSI_TXDR	10 2C10	—/W	—/W	Transmit Data Register
SSI_RXDR	10 2C20	R/—	R/—	Receive Data Register
SSI_RXACK	10 2C24	—/W	—/W	Write a ‘1’ here to ACK read of Receive Data Register
SEM Device				
SEM	10 0500	R/W	R/W	

Endian-ness Appendix C

by Selliah Rathnam

C.1 PURPOSE

TM1000 will be used in X86-CPU based PCs and also in the PowerPC-CPU based Power Macintosh systems. The X86-CPU works in Little Endian mode and the PowerPc-CPU works in Big Endian mode. The PCI system bus operates in Little Endian mode in both the systems. This document describes how the Endian-ness feature in TM1000 is handled in Power Macintosh and the X86 systems.

C.2 LITTLE AND BIG ENDIAN ADDRESSING CONVENTIONS

In Big Endian mode, a given word-address base corresponds to the most significant byte (MSB) of the word. Increasing the byte address generally means decreasing the significance of the byte being accessed. In Little Endian mode, the same word-address base refers to the least significant byte (LSB) of that word. Increasing the byte address generally means increasing the significance of the byte being accessed. This addressing convention is shown in Figure C-1.

In Figure C-1, there is a two-line 'C' code which defines a 32-bit constant in hex format to the variable 'w' and its address is copied to the byte (character) pointer variable 'cp'. The value of address referenced by the 'cp' will have a value of "0x04" in Big Endian machine and a value of "0x07" in Little Endian machine.

It is possible to transfer from one Endian-ness to another just by swapping the bytes within a word as shown in Figure C-2.

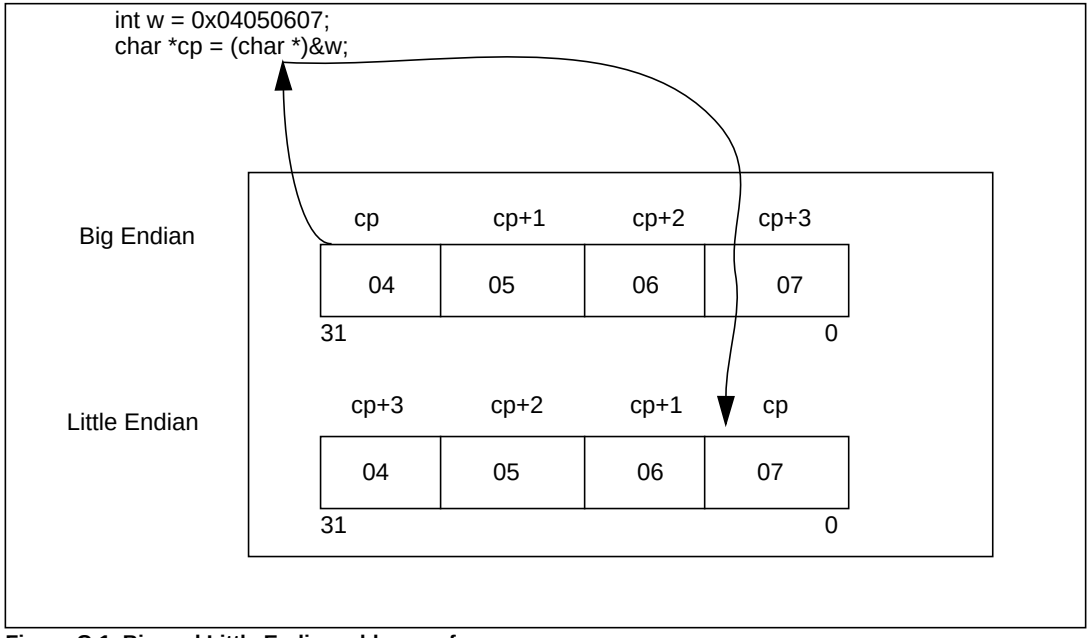


Figure C-1. Big and Little Endian address references

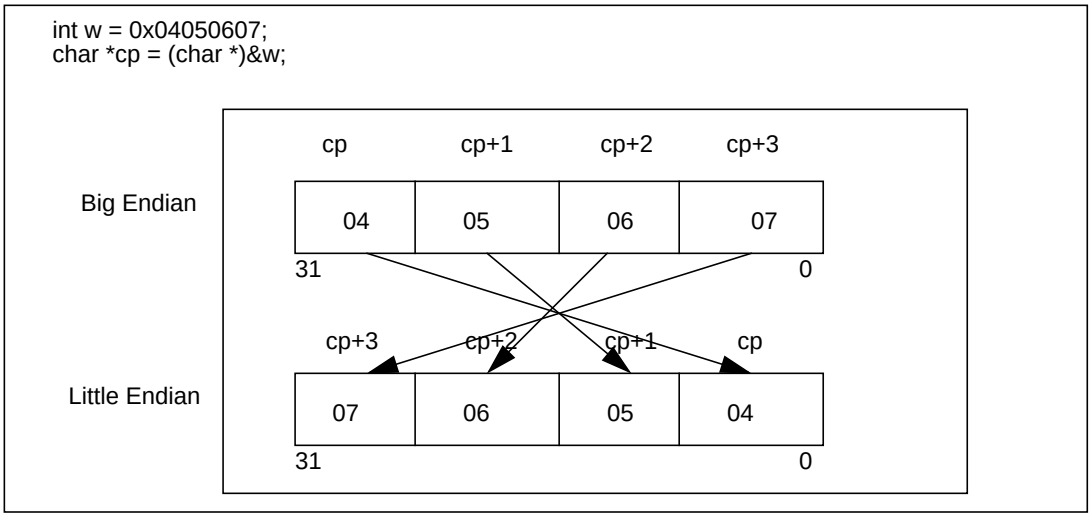


Figure C-2. Data conversion from Big Endian to Little Endian (BSW)

C.3 TEST TO VERIFY THE CORRECT OPERATION OF TM1000 IN X86 AND POWER MACINTOSH SYSTEMS

At the minimum, the following test may be used to verify the correct operation of TM1000 in Little Endian and Big Endian systems.

In X86 system, set the Little Endian flag in TM1000's PCSW register and set the Big Endian flag in TM1000's PCSW register for the Power Macintosh system.

1. Store a 32-bit constant "0x04050607" from the host CPU to the TM1000's SDRAM through PCI interface. Load the word from the same address to one of the TM1000's global register and check for the same value.
2. Store a 32-bit constant "0x04050607" from the host CPU to the TM1000's SDRAM through PCI interface. Load a byte from the same address to one of the TM1000's global register. Check for the value of "0x04" in Power Macintosh system, and check for the value of "0x07" in X86 system.

C.4 REQUIREMENT FOR THE TM1000 TO OPERATE IN EITHER LITTLE ENDIAN OR BIG ENDIAN MODE

The Endian-ness handling in each unit is described in this section. Most of the units use Highway/PCI bus to transfer the data. The data format used in each unit is shown when the data pass through the highway/PCI bus. The highway/PCI bus has four byte lanes. The bit assignment of the highway/PCI bus lanes is shown in Table C-1.

The PCI and TM1000's highway buses are address invariant buses. i.e The data corresponds to address offset 'zero' uses the byte-0 lane of the PCI/Hwy bus, the data corresponds to address offset 'one' uses the byte-1 lane of the PCI/Hwy bus etc.

Table C-1. Bit assignment of the highway/PCI bus lanes

	byte 3	byte 2	byte 1	byte 0
Bits	31:24	23:16	15:8	7:0

C.4.1 Data Cache

TM1000's PCSW register has a byte-sex (BSX) bit to configure the TM1000 in Big Endian or Little Endian mode. This bit needs to be set '1' for the Little Endian mode as defined in Chapter 3, "DSPCPU Architecture." This BSX bit will be used by TM1000's data cache unit for the store/load operation from the data cache. Data Cache performs three categories of data transactions:

- Read/write data from/to CPU register to/from Data Cache or SDRAM memory space (Table C-2).
- Read/write of MMIO data from/to DSPCPU register to/from MMIO registers. and
- Read/write data from/to DSPCPU register to/from PCI address space through special registers in BIU unit.

The DSPCPU's endian-ness of operation is determined by the value of the BSX bit in the PCSW register. Table C-2 and Table C-3 describe the data translation format being used by the Dcache to transfer the data to/from DSPCPU register to/from Data Cache or SDRAM. The Table C-2 and Table C3 formats are restricted to the addresses in between the DRAM_base and DRAM_limit.

There is no byte-swap required for the MMIO data transaction from/to DSPCPU register to the MMIO registers.

Table C-2. Little Endian data format in TM1000 register, Highway, SDRAM memory, PCI bus, Host memory, Host CPU register

PCSW-BSX value	Endian Mode	Data Transaction type	Address	Data in DSPCPU register msb lsb	Data in Hwy/ Dcache/SDRAM/ PCI-bus byte3 byte0 [31:24] [7:0]	Data in Host CPU register msb lsb	Data in Host memory byte3 byte0 [31:24] [7:0]
1	Little	Word r/w	00001000	01020304	01020304	01020304	01020304
1	Little	HalfWord r/w	00001000	xxxx0304	xxxx0304	xxxx0304	xxxx0304
1	Little	HalfWord r/w	00001002	xxxx0304	0304xxxx	xxxx0304	0304xxxx
1	Little	Byte read/write	00001000	xxxxxx04	xxxxxx04	xxxxxx04	xxxxxx04
1	Little	Byte read/write	00001001	xxxxxx04	xxxx04xx	xxxxxx04	xxxx04xx
1	Little	Byte read/write	00001002	xxxxxx04	xx04xxxx	xxxxxx04	xx04xxxx
1	Little	Byte read/write	00001003	xxxxxx04	04xxxxxx	xxxxxx04	04xxxxxx

Table C-3. Big Endian data format in TM1000 register, Highway, SDRAM memory, PCI bus, Host memory, Host CPU register

PCSW-BSX value	Endian Mode	Data Transaction type	Address	Data in DSPCPU register msb lsb	Data in Hwy/ Dcache/SDRAM/ PCI-bus byte3 byte0 [31:24] [7:0]	Data in Host CPU register msb lsb	Data in Host memory byte0 byte3 [31:24] [7:0]
0	Big	Word r/w	00001000	01020304	04030201	01020304	01020304
0	Big	HalfWord r/w	00001000	xxxx0304	xxxx0403	xxxx0304	0304xxxx
0	Big	HalfWord r/w	00001002	xxxx0304	0403xxxx	xxxx0304	xxxx0304
0	Big	Byte read/write	00001000	xxxxxx04	xxxxxx04	xxxxxx04	04xxxxxx
0	Big	Byte read/write	00001001	xxxxxx04	xxxx04xx	xxxxxx04	xx04xxxx
0	Big	Byte read/write	00001002	xxxxxx04	xx04xxxx	xxxxxx04	xxxx04xx
0	Big	Byte read/write	00001003	xxxxxx04	04xxxxxx	xxxxxx04	xxxxxx04

However, one of the special register, pci_data_register, doesn't follow the normal MMIO transactions. The Data Cache will byte-swap the data to/from the pci_data_register using the data translation format as defined in Table C-2 and Table C-3 for the memory cycle.

For the configuration and I/O cycle transactions from DSPCPU, programmer byte-swaps the data in DSPCPU register and write to the pci_data_register using MMIO write operation. There will not be any byte-swap from the pci_data_register in BIU unit to the PCI bus. Software uses the Table C-2 or Table C-3 data to byte-swap the data within the CPU register before writing the data to pci_data_register for the configuration and I/O cycle transactions.

C.4.2 ICache

It is assumed that the ICache will always operate in Little Endian mode in X86 and Power Macintosh systems. ICache will not use the PCSW's byte sex bit (BSX). The compiler supports the loading of instructions in memory differently for Big Endian and Little Endian modes.

C.4.3 TM1000's PCI Interface Unit (BIU)

TM1000's highway bus and the PCI bus are address invariant buses. i.e. a data corresponding to address-zero is always transferred through the byte-zero line irrespective of the endian-ness. This address invariant nature of the PCI and the highway buses allows us to transfer the data from/to PCI bus directly to/from SDRAM without byte-swapping in either big or little endian mode The byte-swapping of data for big endian mode will be performed by the Data Cache unit. However, the MMIO data does not go through the byte swapper in the Data cache. This results in using a byte-swapper in BIU to byte-swap the MMIO data in big endian mode.

TM1000's PCI interface unit (BIU) will have a separate ByteSex (BSX) flag defined in its control register (biu_control). This ByteSex flag will be set from the software,. i.e. MMIO write operation from the host CPU. This ByteSex flag will be used only for MMIO data accesses and non MMIO data accesses will not get affected by this BSX flag. The usage of the BSX flag in BIU unit is given below. Table C-3 shows the byte-swap logic to handle the MMIO accesses from DSPCPU, Host CPU and the Non MMIO data accesses from any source.

BIU has several special registers to handle "mem_cycle", "config_cycle", "I/O cycle" and "dma_cycle". BIU will not byte-swap the in/out data from the special registers.

Table C-4. BIU-BSX bit usage in processing the data in BIU unit

BIU-BSX value	Endian Mode	MMIO access from DSPCPU	MMIO access from PCI side	Non MMIO data
0	Big Endian	No byte-swap	Byte-swap	No byte-swap
1	Little Endian	No byte-swap	No byte-swap	No byte-swap

The Data Cache and software will perform the necessary byte-swap for this data.

When using TM1000 in the X86 based system, the first transaction to the TM1000 is to set the BSX (or SE) bit in BIU's configuration register to avoid unnecessary software byte-swapping in the host CPU for the subsequent MMIO read/write accesses. The BSX bit in BIU_CONTROL register controls the byte swapping of outgoing and incoming data from PCI bus. The default value of BSX is zero, i.e BIU will byte-swap the MMIO data including the write operation to BIU_CONTROL register. Software is required to byte swap the BIU_CONTROL register value within the host CPU before storing the value in BIU_CONTROL register. Once, the BIU-BSX bit has been set, no additional software byte-swap is required for further read/write operations to any MMIO registers.

C.4.4 Image Co-Processor (ICP)

The source data for the image co-processor (ICP) might come from different places such as Video-in, DSPCPU, PCI bus, etc. through the SDRAM. The data consistency needs to be maintained when the TM1000 operates in X86 or in the Power Macintosh system. The ICP needs the capability to operate on the SDRAM source or SDRAM destination data in either Little or Big Endian mode. The ICP will use the following pixel formats in memory as shown in Figure C-3, Figure C-4 and Figure C-5. The ICP's data output format to PCI bus is shown in Figure C-4, Figure C-6 and Figure C-7.

Figure C-3, Figure C-4, Figure C-5 and Figure C-6 illustrate the big and little endian memory image format for the YUV and RGB pixels in SDRAM memory. ICP outputs the data to PCI bus as described in the architecture document for various pixel formats in little endian mode. The RGB 8A and RGB-8R data are byte streams and no swapping is required for the big-endian format. Also, RGB-15+a, RGB-16, RGB-24+a and YUV-4:2:2 pixel formats will be used to output the pixels to PCI in the big endian mode and their format is shown in Figure C-4, Figure C-6, Figure C-7, and Figure C-10. The RGB-24-packed pixel format uses the same format as that of RGB-24+a pixel format and the data value of alpha is undefined. However, space is allocated for the alpha value.

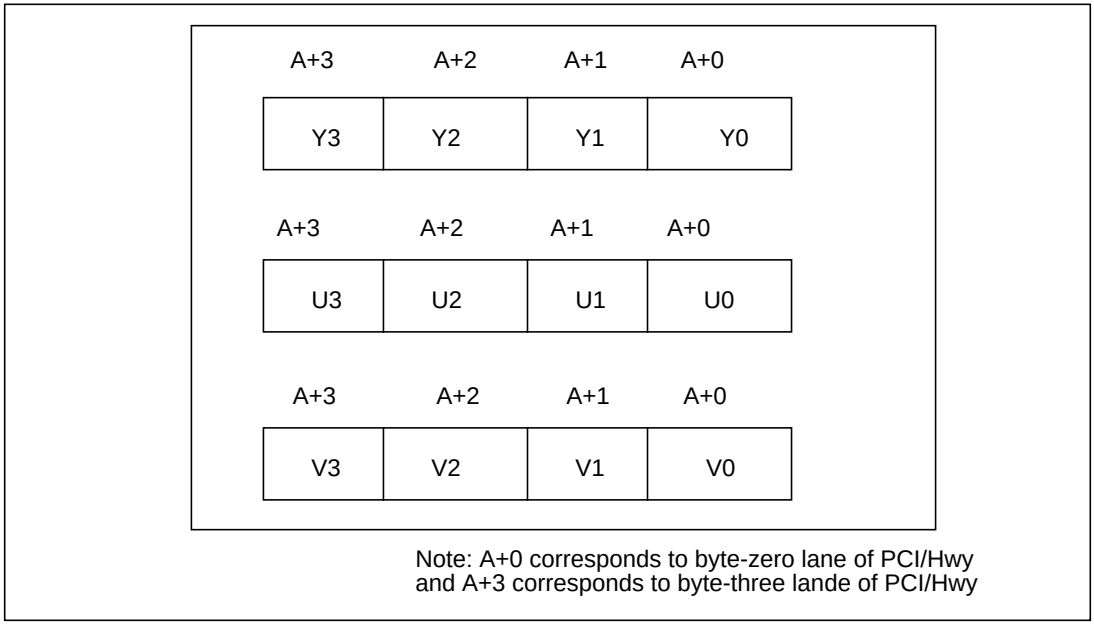


Figure C-3. YUV 4:2:0 and YUV-4:2:2 planar memory images in little and big endian modes

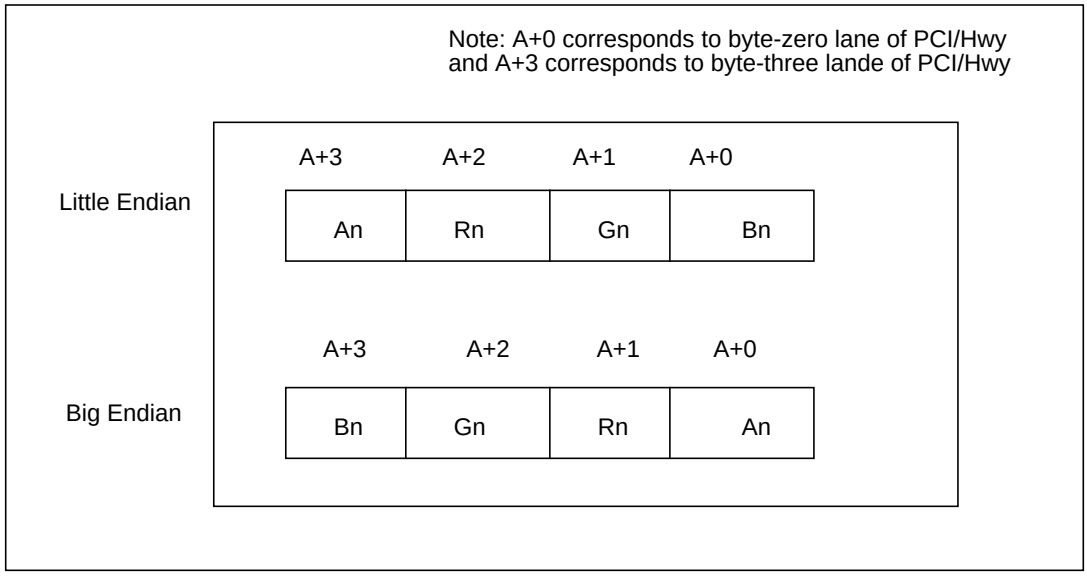


Figure C-4. RBG-24+a memory image

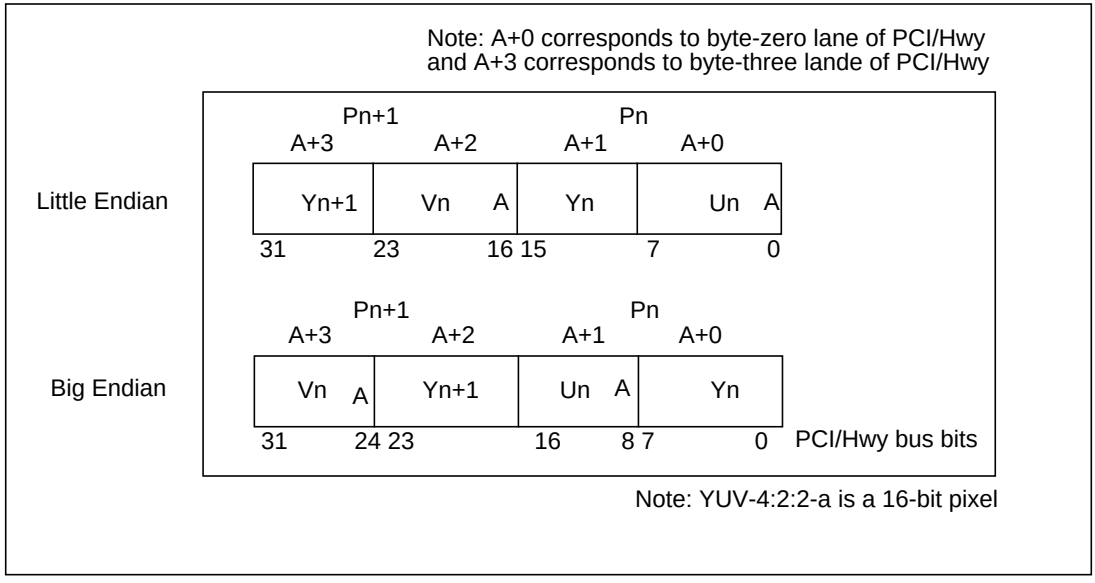


Figure C-5. YUV-4:2:2+a memory image format

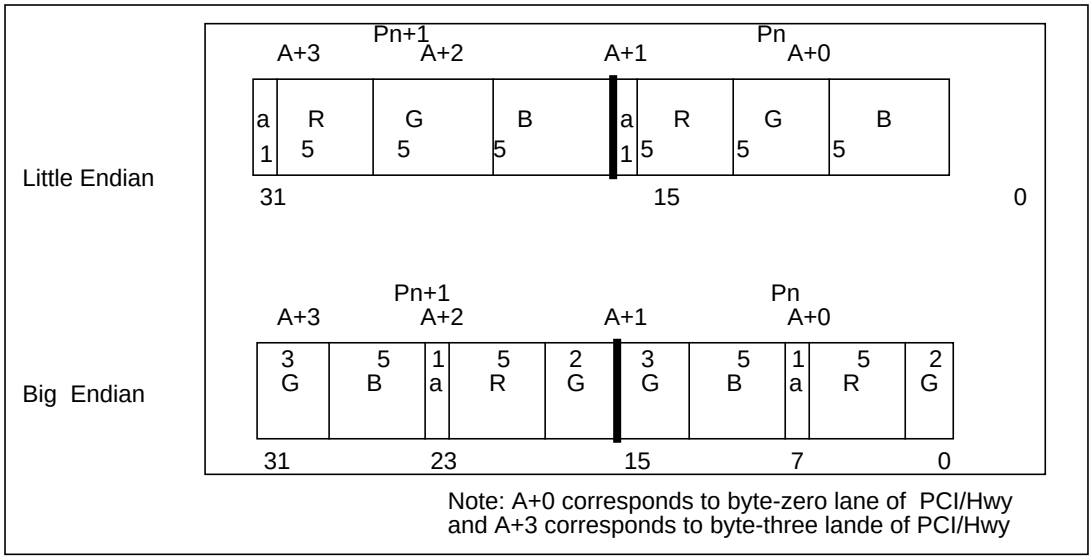


Figure C-6. RGB 15+a data on PCI Bus

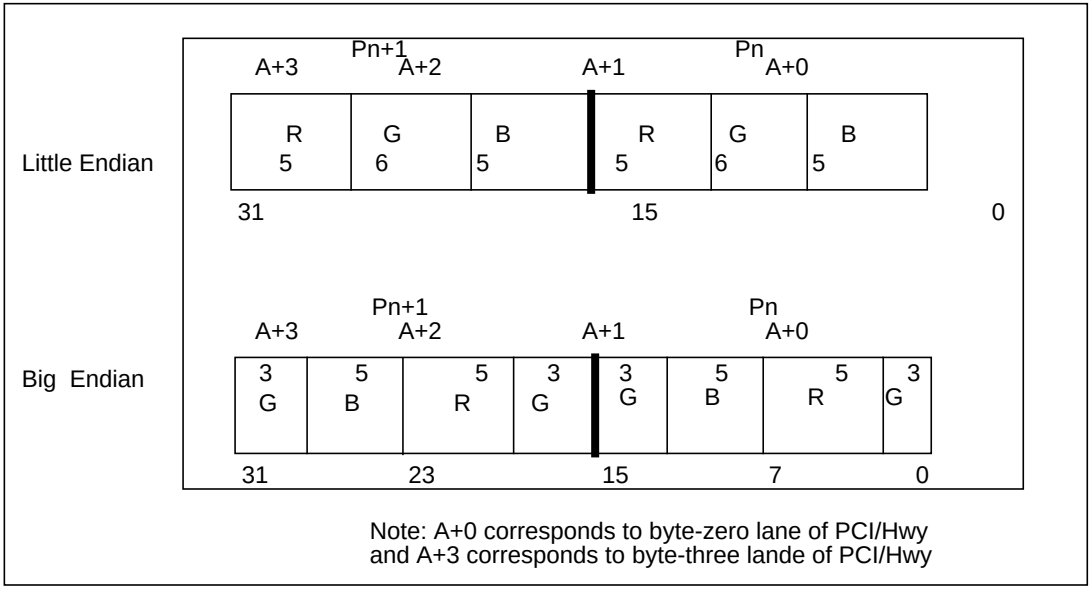


Figure C-7. RGB 16 data on PCI Bus

Image Co-Processor will have a byte sex bit (BSX) defined in its MMIO based configuration register. The setting of this BSX bit and the BSX bit in the PCSW register should be equal. This BSX bit will be set by the software.

The Table C-5 shows the byte-swap implementation of various pixel formats used in the ICP unit. Pl. refer the Swapping type shown in Appendix A for the byte-swap code used in Table C-4 and Table C-5. Byte-swapping is performed only in big endian mode and no swapping is done in the little endian mode.

Table C-5. ICP Byte Swapping Type for Input Data

Endian-ness	BSX-bit	Pixel Type	Swap Type (see Figure C-2 & Figure C-8)
Big Endian	0	Y,U,V Planar	No Swap
Big Endian	0	RGB 24+A Overlay	BSW
Big Endian	0	YUV-4:2:2+A	BSH
Big Endian	0	RGB 15+A	BSH

The RGB-24 packed and YUV-4:2:2 packed data formats are supported only in little endian mode and no byte swapping is done in big endian mode.

C.4.5 Video-In (VI) and Video-Out (VO)

The VI unit stores the YUV pixels in planar 4:2:2 or 4:2:0 image format as shown in Figure C-3 and stores the raw 8 and 10 bit data as shown in Figure C-9.

Table C-6. ICP Byte Swapping Type for Output Data

Endian-ness	BSX-bit	Pixel Type	Swap Type (see Figure C-2 & Figure C-8)
Big Endian	0	RGB 8A: 233	No Swap
Big Endian	0	RGB 8R: 332	No Swap
Big Endian	0	RGB 15+A	BSH
Big Endian	0	RGB 16	BSH
Big Endian	0	RGB 24+A	BSW
Big Endian	0	RGB24 packed	No support for Big Endian
Big Endian	0	YUV- 4:2:2	BSH
Big Endian	0	YUV-4:2:2 packed	No support for Big Endian

The VO unit uses YUV-4:2:2 planar, YUV-4:2:0 planar, and YUV-4:2:2-a packed as input pixel formats. The planar memory image format of the YUV-4:2:2 and YUV-4:2:0 are shown in Figure C-3. The YUV-4:2:2-a memory image format for overlay generation is shown in Figure C-5. The VO unit outputs the pixel as YUV-4:2:2 packed format.Figure C-10 shows the YUV-4:2:2 packed pixel format.

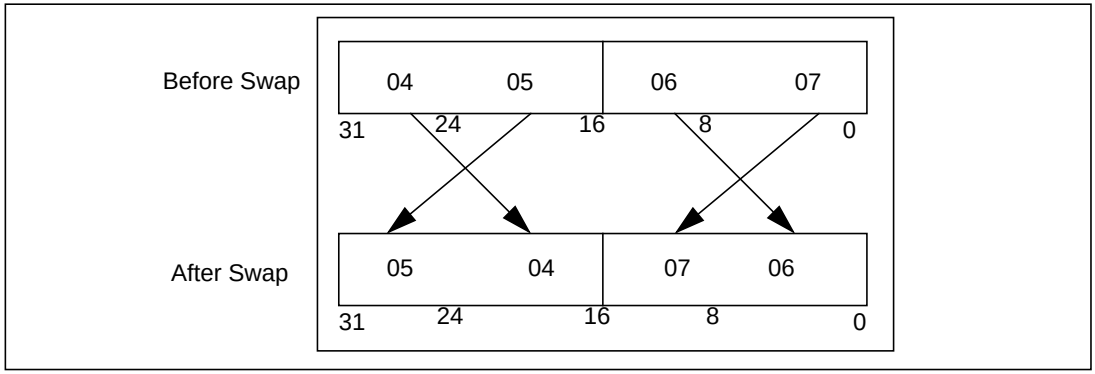


Figure C-8. Half-word Swap Within a Half-word (BSH)

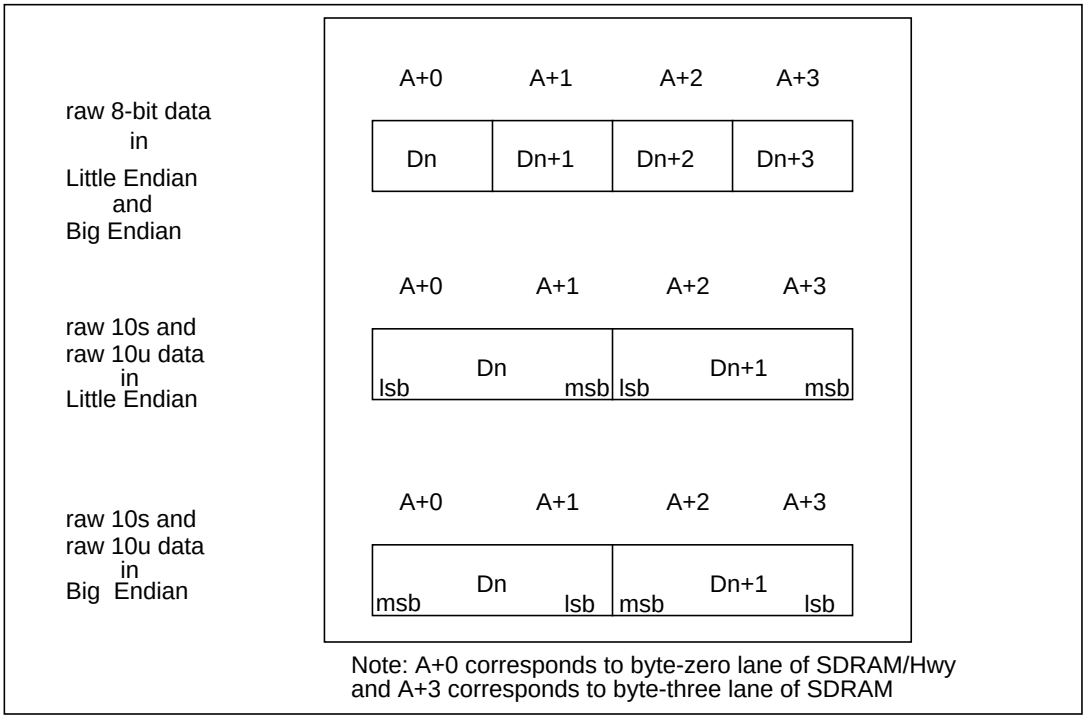


Figure C-9. Memory image format for raw 8-bit and 10-bit data

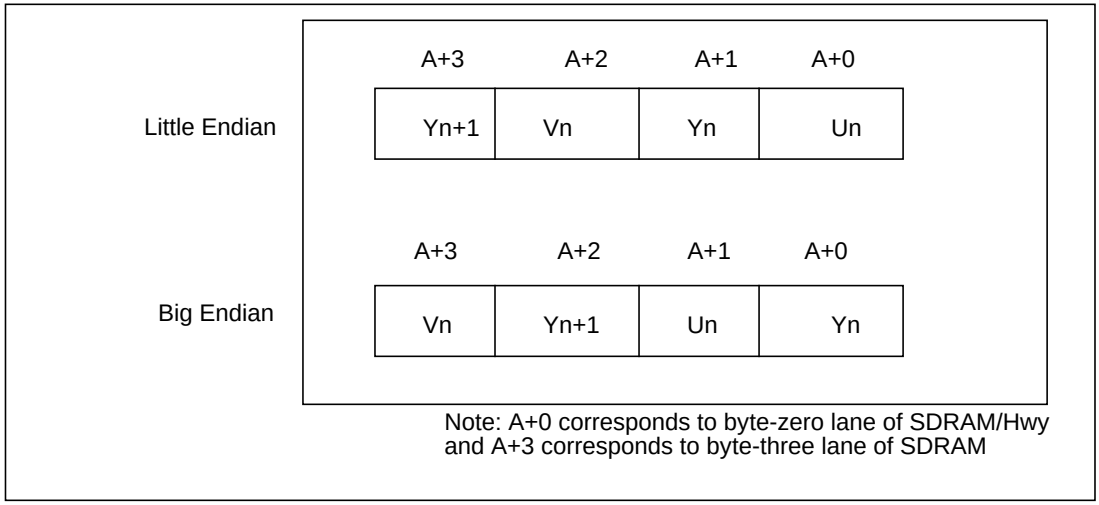


Figure C-10. YUV-4:2:2 memory image format for VO unit

The VI and VO units have a byte sex bit (BSX) defined in VI_CONTROL and VO_CONTROL MMIO based configuration register. The definition of the this BSX bit and the BSX bit in the PCSW register should be treated as same. This BSX bit will be set by the software

C.4.6 Audio-In (AI) and Audio-Out (AO)

The AI and AO units use 8-bit Mono, 8-bit stereo, 16-bit mono and 16-bit stereo data and the memory image format of these data is shown in Figure C-11.

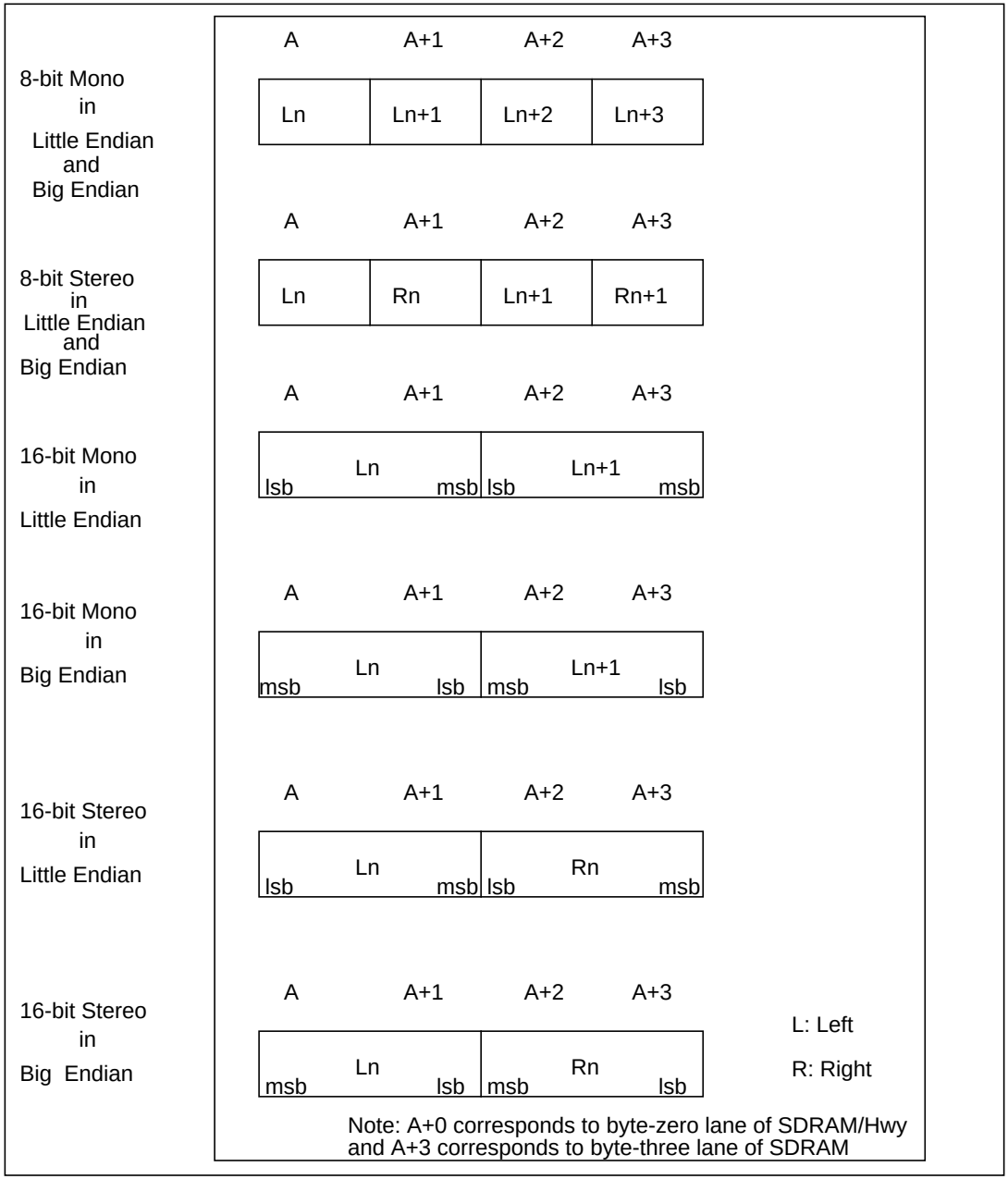


Figure C-11. Memory image format for audio data

The AI and AO units will have byte sex bit (BSX) defined in its MMIO based configuration register. The definition of the this BSX bit and the BSX bit in the PCSW register should be treated as same. This BSX bit will be set by the software

C.4.7 Variable Length Encoder (VLD)

The VLD takes the input from SDRAM in the form of bit stream, with byte aligned starting address and outputs a header stream and a 'run-level' data stream.

The VLD unit will have byte sex bit (BSX) defined in its MMIO based configuration register. The definition of the this BSX bit and the BSX bit in the PCSW register should be treated as same. This BSX bit will be set by the software

Figure C-12 describes the VLD input and output data format as seen in the Highway bus. The input data is byte oriented and no swapping at VLD is required. However, the output data will be read by CPU in terms of word unit. VLD need to swap the output bytes within a word as shown in Figure C-12 to compensate for the CPU swap.

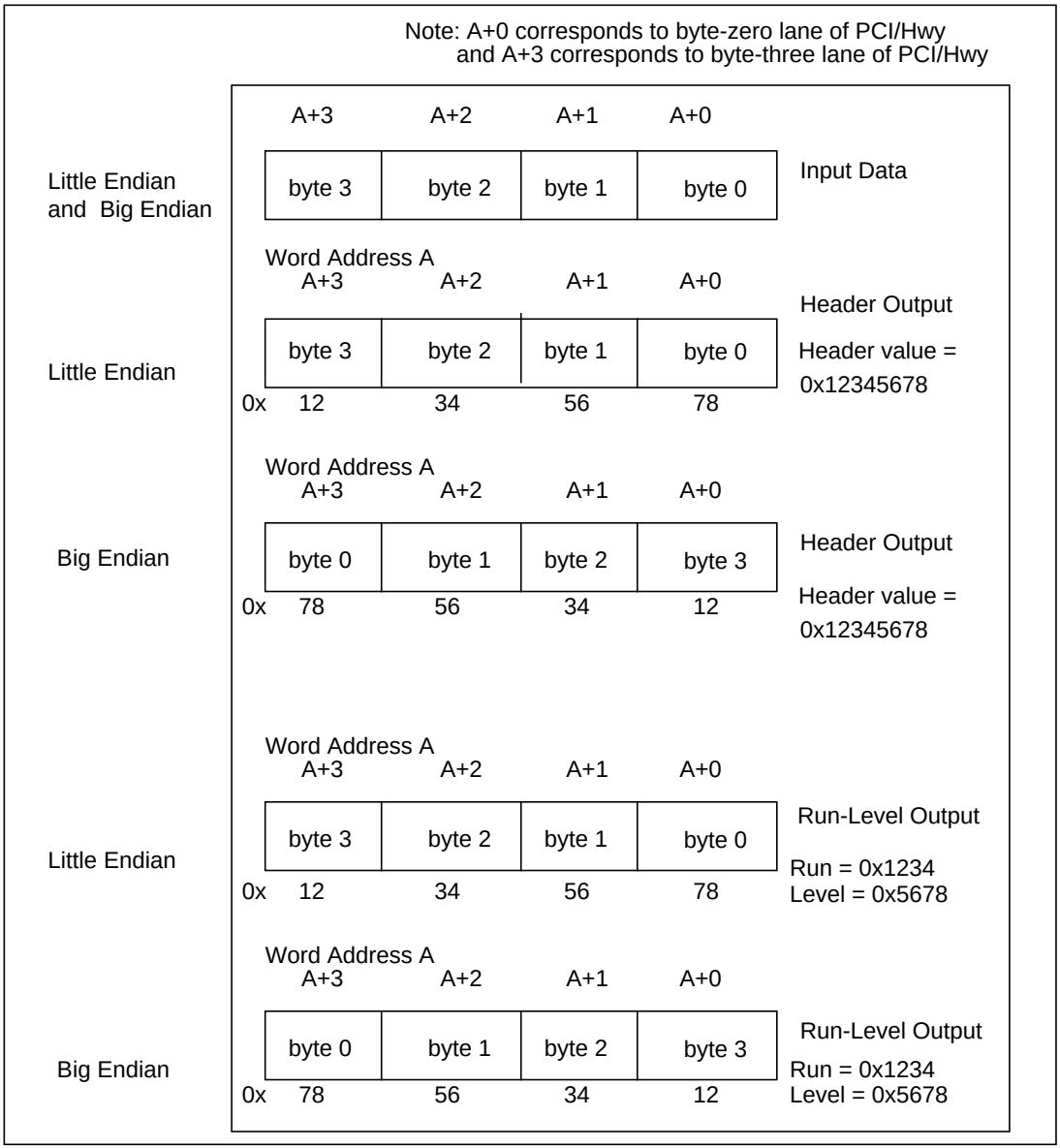


Figure C-12. VLD input and output data format

C.4.8 Synchronous Serial Interface

The synchronous serial interface unit has I/O connections through the external serial pins and also to the internal 32-bit data highway. The minimum quantity of data to be analyzed by CPU is 16-bits (i.e. one half word).

In the receiving mode from the external PIN, the first bit received is moved to the most significant bit location (bit-15) when the 'receive_shift_direction' bit in the control register is set to zero. The first bit is moved to the least significant bit location (bit-0) when the 'receive_shift_direction' bit is set to 1.

In the transmitting mode to the external PIN, the msb bit is sent first when the 'transmit_shift_direction' bit is set to 0 and the lsb bit is sent first when the 'transmit_shift_direction' bit is set to 1.

The highway bus is 32-bit wide, two half-word data is assembled as one word in the SSI and sent to the highway bus (see Figure C-13.). The data format for the received data from highway is also shown in the Figure C-13.

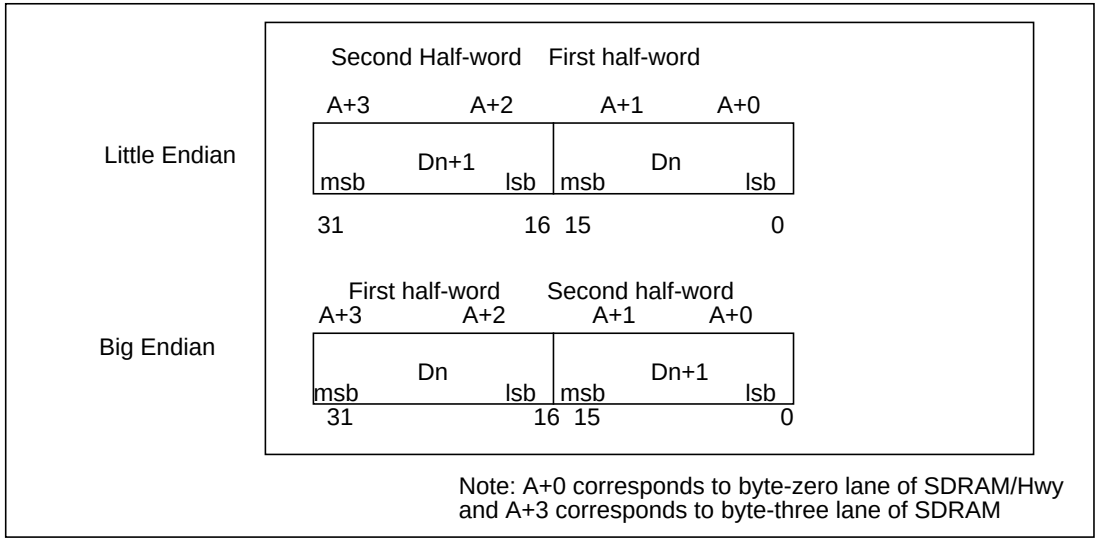


Figure C-13. SSI data format as seen in Highway

C.4.9 Compiler

The compiler will support the loading of instruction in memory differently for Big Endian and Little Endian modes.

C.5 SUMMARY

TM1000 is required to operate in the same Endian-ness as that of the host CPU. TM1000 operates by default in Big Endian mode and no special steps are required to set the Endian bits in the TM1000. When using the TM1000 in X86 system, the first transaction to the TM1000 is to set the BSX bit in the TM1000's BIU_CONTROL register. The second paragraph of Section C.4.3 explains how to set this BIU_CONTROL register from the host CPU.

C.6 REFERENCES

1. *PCI Multimedia Design Guide*, revision 1.0 - dated March 29,1994
2. *Designing PCI Cards and Drivers for Power Macintosh Computers*, By Apple Computer, Inc.; Reference: R0650LL/A; Phone: 1-800-282-2732

Index

A

B

C

D

E

F

G

H

I

J

L

M

Q

R

S

T

U

V

W

Y

Z