

## **M•CORE**

### **Reference Manual with M210/M210S Specifications**

HCMOS  
Microcontroller Unit



# M•CORE with M210/M210S


## Specifications

### Reference Manual

---

---

*Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.*

Motorola and  are registered trademarks of Motorola, Inc.  
DigitalDNA, M•CORE, and OnCE are trademarks of Motorola, Inc.

© Motorola, Inc., 2001

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

<http://www.motorola.com/semiconductors/>

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

### Revision History

Date	Revision Level	Description	Page Number(s)
September, 2001	1.0	Complete reformat adding M210/M210S core specifications	Throughout

List of Sections

Section 1. Overview .....27

Section 2. Registers .....39

Section 3. Instructions .....53

Section 4. Exception Processing.....163

Section 5. Core Interface .....181

Section 6. Interface Operation .....195

Section 7. Hardware Accelerator Interface (HAI) .....219

Section 8. JTAG Test Access Port and OnCE .....247

Appendix A. Nomenclature .....299

Appendix B. M210 and M210S Core Instruction  
Pipeline and Timing .....303

Appendix C. M210/M210S Core Interface .....309

Appendix D. M210/M210S Interface Operation.....329

Index .....371



# Table of Contents

## Section 1. Overview

1.1	Contents . . . . .	27
1.2	Introduction . . . . .	27
1.3	Features . . . . .	28
1.4	Microarchitecture Summary . . . . .	29
1.5	Programming Model . . . . .	30
1.6	Data Format Summary . . . . .	33
1.7	Operand Addressing Capabilities . . . . .	34
1.8	Instruction Set Overview . . . . .	34

## Section 2. Registers

2.1	Contents . . . . .	39
2.2	Introduction . . . . .	39
2.3	User Programming Model . . . . .	40
2.3.1	General-Purpose Registers . . . . .	41
2.3.2	Program Counter . . . . .	41
2.3.3	Condition Code/Carry Bit . . . . .	41
2.4	Supervisor Programming Model . . . . .	41
2.4.1	Alternate Register File . . . . .	43
2.4.2	Processor Status Register . . . . .	43
2.4.2.1	Updates to the PSR . . . . .	48
2.4.2.2	Exception Recognition and Processing Updates . . . . .	48
2.4.2.3	RTE and RFI Instruction Updates . . . . .	48
2.4.2.4	MTCR Instruction Updates . . . . .	49
2.4.3	Vector Base Register . . . . .	49

2.4.4	Supervisor Storage Registers . . . . .	50
2.4.5	Exception Shadow Registers . . . . .	50
2.4.6	Global Control Register . . . . .	51
2.4.7	Global Status Register . . . . .	51

## Section 3. Instructions

3.1	Contents . . . . .	53
3.2	Introduction . . . . .	54
3.3	Instruction Types and Addressing Modes . . . . .	54
3.3.1	Register-to-Register Instructions . . . . .	54
3.3.1.1	Monadic Register Addressing Mode . . . . .	55
3.3.1.2	Dyadic Register Addressing Mode . . . . .	56
3.3.1.3	Register with 5-Bit Immediate Mode . . . . .	57
3.3.1.4	Register with 5-Bit Offset Immediate Mode . . . . .	58
3.3.1.5	Register with 7-Bit Immediate Mode . . . . .	58
3.3.1.6	Control Register Addressing Mode . . . . .	59
3.3.2	Data Memory Access Instructions . . . . .	59
3.3.2.1	Scaled 4-Bit Immediate Addressing Mode . . . . .	59
3.3.2.2	Load/Store Register Quadrant Mode . . . . .	59
3.3.2.3	Load/Store Multiple Register Mode . . . . .	60
3.3.2.4	Load Relative Word Mode . . . . .	60
3.3.3	Flow Control Instructions . . . . .	61
3.3.3.1	Scaled 11-Bit Displacement Mode . . . . .	61
3.3.3.2	Register Addressing Mode . . . . .	61
3.3.3.3	Indirect Mode . . . . .	61
3.3.3.4	Register with 4-Bit Negative Displacement Mode . . . . .	62
3.4	Opcode Map . . . . .	63
3.5	Instruction Set . . . . .	67

## Section 4. Exception Processing

4.1	Contents . . . . .	163
4.2	Introduction . . . . .	164
4.3	Exception Processing Overview . . . . .	164



4.4	Stages of Exception Processing . . . . .	165
4.5	Exception Vectors . . . . .	167
4.6	Exception Types . . . . .	168
4.6.1	Reset Exception (Vector Offset 0x0) . . . . .	169
4.6.2	Misaligned Access Exception (Vector Offset 0x4) . . . . .	169
4.6.3	Access Error Exception (Vector Offset 0x8) . . . . .	170
4.6.4	Divide-by-Zero Exception (Vector Offset 0x0C) . . . . .	170
4.6.5	Illegal Instruction Exception (Vector Offset 0x10) . . . . .	170
4.6.6	Privilege Violation Exception (Vector Offset 0x14) . . . . .	171
4.6.7	Trace Exception (Vector Offset 0x18) . . . . .	171
4.6.8	Breakpoint Exception (Vector Offset 0x1C) . . . . .	173
4.6.9	Unrecoverable Error Exception (Vector Offset 0x20) . . . . .	173
4.6.10	Soft Reset Exception (Vector Offset 0x24) . . . . .	173
4.6.11	Interrupt Exceptions . . . . .	174
4.6.11.1	Normal Interrupt ( $\overline{\text{INT}}$ ) . . . . .	175
4.6.11.2	Fast Interrupt ( $\overline{\text{FINT}}$ ) . . . . .	175
4.6.12	Hardware Accelerator Exception (Vector Offset 0x30) . . . . .	176
4.6.13	Instruction Trap Exception (Vector Offset 0x40-0x5C) . . . . .	176
4.7	Exception Priorities . . . . .	176
4.8	Returning from Exception Handlers . . . . .	178

## Section 5. Core Interface

5.1	Contents . . . . .	181
5.2	Introduction . . . . .	182
5.3	Signal Descriptions . . . . .	182
5.3.1	Address Bus (ADDR[31:0]) . . . . .	185
5.3.2	Data Bus (DATA[31:0]) . . . . .	185
5.3.3	Transfer Request ( $\overline{\text{TREQ}}$ ) . . . . .	185
5.3.4	Transfer Busy ( $\overline{\text{TBUSY}}$ ) . . . . .	185
5.3.5	Transfer Abort ( $\overline{\text{ABORT}}$ ) . . . . .	185
5.4	Transfer Attribute Signals . . . . .	185
5.4.1	Transfer Code (TC[2:0]) . . . . .	186
5.4.2	Read/Write (R/ $\overline{\text{W}}$ ) . . . . .	186

## Table of Contents

5.4.3	Transfer Size (TSIZ[1:0])	186
5.4.4	Sequential Access ( $\overline{\text{SEQ}}$ )	187
5.4.5	Data to Address (D2A)	187
5.5	Transfer Control Signals	187
5.5.1	Transfer Acknowledge ( $\overline{\text{TA}}$ )	187
5.5.2	Transfer Error Acknowledge ( $\overline{\text{TEA}}$ )	188
5.5.3	Breakpoint Request ( $\overline{\text{BRKRQ}}$ )	188
5.6	Memory Management Control Signals	188
5.6.1	Translate Control ( $\overline{\text{TE}}$ )	188
5.6.2	Soft Reset ( $\overline{\text{SRST}}$ )	188
5.7	Interrupt Control Signals	189
5.7.1	Normal Interrupt Request ( $\overline{\text{INT}}$ )	189
5.7.2	Fast Interrupt Request ( $\overline{\text{FINT}}$ )	189
5.7.3	Interrupt Pending Status ( $\overline{\text{IPEND}}$ )	189
5.7.4	Interrupt Vector Number (VEC[6:0])	189
5.7.5	Autovector ( $\overline{\text{AVEC}}$ )	190
5.8	Power Management Control Signals	190
5.9	Status and Clock Signals	191
5.9.1	Processor Status (PSTAT[3:0])	191
5.9.2	M•CORE Processor Clock (CLK)	192
5.10	Global Status and Control Interface	192
5.11	Hardware Accelerator Interface	192
5.12	Debug/Emulation Support Signals	193
5.12.1	Debug Request ( $\overline{\text{DBGREQ}}$ )	193
5.12.2	Debug Acknowledge ( $\overline{\text{DBGACK}}$ )	193
5.13	Test Signals	193
5.14	Power Supply Connections	193
5.15	Signal Summary	193

## Section 6. Interface Operation

6.1	Contents . . . . .	195
6.2	Introduction . . . . .	196
6.3	Bus Characteristics . . . . .	196
6.4	Data Transfer Mechanism . . . . .	197
6.5	Processor Instruction/Data Transfers . . . . .	199
6.5.1	Instruction and Data Read Transfer Cycles . . . . .	200
6.5.2	Read Transfer Cycles with Wait State(s) . . . . .	202
6.5.3	Write Transfer Cycles . . . . .	202
6.5.4	Write Transfer Cycles with Wait State(s) . . . . .	205
6.5.5	Data Bus Hand-Off Between Read and Write Cycles . . . . .	206
6.6	Exception Bus Control Cycles . . . . .	207
6.6.1	Bus Errors . . . . .	208
6.6.2	Breakpoint Requests . . . . .	208
6.7	$\overline{\text{ABORT}}$ Signal Operation . . . . .	209
6.8	D2A Signal Operation . . . . .	210
6.9	Reset Operation . . . . .	211
6.9.1	Hard Reset (Power-On Reset) . . . . .	211
6.9.2	Soft Reset . . . . .	211
6.10	Memory Management Interface Operation . . . . .	212
6.11	Interrupt Interface Operation . . . . .	212
6.12	Global Status and Control Interface Operation . . . . .	214
6.13	Power Management Interface Operation . . . . .	215
6.14	Emulation/Debug Interface Operation . . . . .	217

## Section 7. Hardware Accelerator Interface (HAI)

7.1	Contents . . . . .	219
7.2	Introduction . . . . .	220
7.3	Overview . . . . .	220

7.4	Register Snooping Mechanism . . . . .	221
7.5	Instruction Transfer Mechanism . . . . .	222
7.5.1	Control Handshake . . . . .	222
7.5.2	Driving the $\overline{H\_BUSY}$ and $\overline{H\_EXCP}$ Signals . . . . .	229
7.6	Data Transfer Mechanism . . . . .	230
7.6.1	Register Transfers . . . . .	230
7.6.2	Memory Transfers . . . . .	233
7.6.2.1	H_LD Transfer . . . . .	233
7.6.2.2	H_ST Transfer . . . . .	234
7.7	Instruction Primitives . . . . .	238
7.7.1	H_CALL Primitive . . . . .	238
7.7.2	H_RET Primitive . . . . .	239
7.7.3	H_LD Primitive . . . . .	239
7.7.4	H_ST Primitive . . . . .	240
7.7.5	H_EXEC Primitive . . . . .	241
7.8	Instruction Primitive Glossary . . . . .	241

## Section 8. JTAG Test Access Port and OnCE

8.1	Contents . . . . .	247
8.2	Introduction . . . . .	249
8.3	Top-Level Test Access Port (TAP) . . . . .	251
8.3.1	Test Clock (TCLK) . . . . .	252
8.3.2	Test Mode Select (TMS) . . . . .	252
8.3.3	Test Data Input (TDI) . . . . .	252
8.3.4	Test Data Output (TDO) . . . . .	252
8.3.5	Test Reset ( $\overline{TRST}$ ) . . . . .	252
8.3.6	Debug Event ( $\overline{DE}$ ) . . . . .	252
8.4	Top-Level TAP Controller . . . . .	254
8.5	Instruction Shift Register . . . . .	255
8.5.1	EXTEST Instruction . . . . .	255
8.5.2	IDCODE Instruction . . . . .	256
8.5.3	SAMPLE/PRELOAD Instruction . . . . .	257
8.5.4	ENABLE_MCU_ONCE Instruction . . . . .	257

8.5.5	HIGHZ Instruction . . . . .	258
8.5.6	CLAMP Instruction . . . . .	258
8.5.7	BYPASS Instruction . . . . .	258
8.6	IDCODE Register . . . . .	259
8.7	Bypass Register . . . . .	260
8.8	Boundary SCAN Register . . . . .	260
8.9	Restrictions . . . . .	260
8.10	Non-Scan Chain Operation . . . . .	261
8.11	Boundary Scan . . . . .	261
8.12	Low-Level TAP (OnCE) Module . . . . .	267
8.13	Signal Descriptions . . . . .	269
8.13.1	Debug Serial Input (TDI) . . . . .	269
8.13.2	Debug Serial Clock (TCLK) . . . . .	269
8.13.3	Debug Serial Output (TDO) . . . . .	269
8.13.4	Debug Mode Select (TMS) . . . . .	270
8.13.5	Test Reset ( $\overline{\text{TRST}}$ ) . . . . .	270
8.13.6	Debug Event ( $\overline{\text{DE}}$ ) . . . . .	270
8.14	Functional Description . . . . .	270
8.14.1	Operation . . . . .	271
8.14.2	OnCE Controller and Serial Interface . . . . .	272
8.14.3	OnCE Interface Signals . . . . .	273
8.14.3.1	Internal Debug Request Input ( $\overline{\text{IDR}}$ ) . . . . .	273
8.14.3.2	CPU Debug Request ( $\overline{\text{DBGREQ}}$ ) . . . . .	274
8.14.3.3	CPU Debug Acknowledge ( $\overline{\text{DBGACK}}$ ) . . . . .	274
8.14.3.4	CPU Breakpoint Request ( $\overline{\text{BRKREQ}}$ ) . . . . .	274
8.14.3.5	CPU Address, Attributes (ADDR, ATTR) . . . . .	274
8.14.3.6	CPU Status (PSTAT) . . . . .	274
8.14.3.7	OnCE Debug Output ( $\overline{\text{DEBUG}}$ ) . . . . .	274
8.14.4	OnCE Controller Registers . . . . .	275
8.14.4.1	OnCE Command Register . . . . .	275
8.14.4.2	OnCE Control Register . . . . .	278
8.14.4.3	OnCE Status Register . . . . .	282
8.14.5	OnCE Decoder (ODEC) . . . . .	284

## Table of Contents

8.14.6	Memory Breakpoint Logic	284
8.14.6.1	Memory Address Latch (MAL)	285
8.14.6.2	Breakpoint Address Base Registers	285
8.14.7	Breakpoint Address Mask Registers	285
8.14.7.1	Breakpoint Address Comparators	286
8.14.7.2	Memory Breakpoint Counters	286
8.14.8	OnCE Trace Logic	286
8.14.8.1	OnCE Trace Counter	287
8.14.8.2	Trace Operation	288
8.14.9	Methods of Entering Debug Mode	288
8.14.9.1	Debug Request During $\overline{\text{RESET}}$	288
8.14.9.2	Debug Request During Normal Activity	289
8.14.9.3	Debug Request During Stop, Doze, or Wait Mode	289
8.14.9.4	Software Request During Normal Activity	289
8.14.10	Enabling OnCE Trace Mode	289
8.14.11	Enabling OnCE Memory Breakpoints	290
8.14.12	Pipeline Information and Write-Back Bus Register	290
8.14.12.1	Program Counter Register	291
8.14.12.2	Instruction Register	291
8.14.12.3	Control State Register	291
8.14.12.4	Writeback Bus Register	293
8.14.12.5	Processor Status Register	293
8.14.13	Instruction Address FIFO Buffer (PC FIFO)	294
8.14.14	Reserved Test Control Registers	295
8.14.15	Serial Protocol	295
8.14.16	OnCE Commands	296
8.14.17	Target Site Debug System Requirements	296
8.14.18	Interface Connector for JTAG/OnCE Serial Port	296

## Appendix A. Nomenclature

A.1	Contents	299
A.2	Introduction	299
A.3	References	299
A.4	Units and Measures	299

A.5	Symbology .....	300
A.6	Terminology .....	300

## **Appendix B. M210 and M210S Core Instruction Pipeline and Timing**

B.1	Contents .....	303
B.2	Introduction .....	303
B.3	Instruction Pipeline .....	303
B.4	Instruction Execution Time .....	305

## **Appendix C. M210/M210S Core Interface**

C.1	Contents .....	309
C.2	Introduction .....	310
C.3	M210 Core Interface Overview .....	311
C.4	MLB Signal Descriptions .....	317
C.4.1	Bus Signals .....	317
C.4.1.1	Address Bus (ADDR[22:0]) .....	317
C.4.1.2	Data Bus (DATA[31:0]) .....	317
C.4.1.3	Input Data Bus (DATA <sub>In</sub> [31:0]) .....	317
C.4.1.4	Output Data Bus (DATA <sub>Out</sub> [31:0]) .....	317
C.4.1.5	Data Bus Byte Output Enable (DATA <sub>EN</sub> [3:0]) .....	317
C.4.2	Transfer Control .....	318
C.4.2.1	Transfer Acknowledge ( $\overline{TA}$ ) .....	318
C.4.2.2	Transfer Error Acknowledge ( $\overline{TEA}$ ) .....	318
C.4.2.3	Transfer Request ( $\overline{TREQ}$ ) .....	318
C.4.2.4	Transfer Busy ( $\overline{TBUSY}$ ) .....	318
C.4.2.5	Transfer Busy Output ( $\overline{TBUSYOUT}$ ) .....	318
C.4.2.6	Transfer Busy Input ( $\overline{TBUSYIN}$ ) .....	319
C.4.2.7	Transfer Abort ( $\overline{ABORT}$ ) .....	319
C.4.3	Transfer Attribute Signals .....	319
C.4.3.1	Transfer Code (TC[2:0]) .....	319
C.4.3.2	Read/Write (R/ $\overline{W}$ ) .....	320

## Table of Contents

C.4.3.3	Transfer Size (TSIZ[1:0])	320
C.4.3.4	Sequential Access ( $\overline{\text{SEQ}}$ )	320
C.4.4	Translate Control ( $\overline{\text{TE}}$ )	320
C.4.5	Data to Address Signal (D2A)	320
C.4.6	Processor Status Signals	321
C.4.6.1	Processor Status (PSTAT[3:0])	321
C.5	Other Processor Signals	322
C.5.1	Master Clock (MCLK)	322
C.5.2	Reset Control Signals	322
C.5.2.1	Master Reset ( $\overline{\text{RST}}$ )	322
C.5.2.2	Power-On Reset (POR)	322
C.5.3	Bus Arbitration Control Signals	323
C.5.3.1	Bus Request ( $\overline{\text{BR}}$ )	323
C.5.3.2	Bus Grant ( $\overline{\text{BG}}$ )	323
C.5.3.3	Three-State Control Address ( $\overline{\text{TSCA}}$ )	323
C.5.3.4	Three-State Control Data ( $\overline{\text{TSCD}}$ )	323
C.5.4	Power Management Control Signals	324
C.5.4.1	Low-Power Mode ( $\overline{\text{LPMD}}[1:0]$ )	324
C.5.4.2	Wakeup ( $\overline{\text{WAKEUP}}$ )	325
C.5.5	Global Status and Control Interface Signals	325
C.5.5.1	Global Control (GCB[31:0])	325
C.5.5.2	Global Status (GSB[31:0])	325
C.5.6	Interrupt Control Signals	326
C.5.6.1	Normal Interrupt Request ( $\overline{\text{INT}}$ )	326
C.5.6.2	Raw Normal Interrupt Request ( $\overline{\text{INTRAW}}$ )	326
C.5.6.3	Fast Interrupt Request ( $\overline{\text{FINT}}$ )	326
C.5.6.4	Raw Fast Interrupt Request ( $\overline{\text{FINTRAW}}$ )	326
C.5.6.5	Interrupt Pending ( $\overline{\text{IPEND}}$ )	326
C.5.6.6	Interrupt Vector Number ( $\overline{\text{VEC}}[6:0]$ )	326
C.5.6.7	Autovector ( $\overline{\text{AVEC}}$ )	327
C.5.7	Power Supply Connections	327

## Appendix D. M210/M210S Interface Operation

D.1	Contents	329
D.2	Introduction	330
D.3	Bus Characteristics	330



D.4	Data Transfer Mechanism .....	331
D.5	Processor Instruction/Data Transfers .....	333
D.5.1	Instruction and Data Read Transfer Cycles .....	334
D.5.2	Read Transfer Cycles with Wait State .....	336
D.5.3	Write Transfer Cycles .....	337
D.5.4	Write Transfer Cycles with Wait State .....	339
D.5.5	Data Bus Hand-Off .....	340
D.6	Bidirectional Three-State Data Bus .....	341
D.7	Bus Exception Control Cycles .....	342
D.8	Bus Errors .....	342
D.9	Abort Signal Operation .....	343
D.10	Data to Address Transfer Operation .....	344
D.11	Breakpoint Request Operation .....	345
D.12	Bus Arbitration Operation .....	346
D.12.1	Operation Examples .....	348
D.12.2	Interaction with Low-Power Modes and Debug Operation .....	360
D.12.3	Bus Arbitration and Entry into Low-Power States .....	360
D.13	Reset Operation .....	362
D.13.1	System Issues .....	363
D.13.2	Timing .....	364
D.14	Interrupt Interface Operation .....	365
D.15	Global Status and Control Interface Operation .....	367
D.16	Power Management Interface Operation .....	367
D.17	Emulation/Debug Interface Operation .....	370

## Index

Index .....	371
-------------	-----



## List of Figures

Figure	Title	Page
1-1	Programming Model . . . . .	32
1-2	Data Organization in Memory . . . . .	33
1-3	Data Organization in Registers . . . . .	34
2-1	User Programming Model . . . . .	40
2-2	Supervisor Additional Resources . . . . .	42
2-3	Processor Status Register (PSR) . . . . .	44
2-4	Vector Base Register (VBR) . . . . .	50
3-1	Monadic Format . . . . .	55
3-2	Dyadic Format . . . . .	56
3-3	Register with 5-Bit Immediate Format . . . . .	57
3-4	Register with 5-Bit Offset Immediate Format . . . . .	58
3-5	Register with 7-Bit Immediate Format . . . . .	58
3-6	Control Register Addressing Format . . . . .	59
3-7	Scaled 4-Bit Immediate Format . . . . .	59
3-8	Load/Store Register Quadrant Format . . . . .	60
3-9	Load/Store Multiple Registers Format . . . . .	60
3-10	Load Relative Word Format . . . . .	60
3-11	Scaled 11-Bit Displacement Format . . . . .	61
3-12	Register Addressing Format . . . . .	61
3-13	Indirect Format . . . . .	62
3-14	Register with 4-Bit Displacement Addressing Format . . . . .	62
4-1	Interrupt Interface Signals . . . . .	175
5-1	M•CORE Signal Groups . . . . .	183

## List of Figures

Figure	Title	Page
6-1	Signal Relationships to Clocks . . . . .	197
6-2	External Multiplexer Connections . . . . .	198
6-3	Instruction/Data Read Cycle . . . . .	201
6-4	Read Cycle with Wait States . . . . .	203
6-5	Write Cycle . . . . .	203
6-6	Write Cycle with Wait States. . . . .	205
6-7	Data Bus Hand-Off Operation. . . . .	206
6-8	Data Bus Hand-Off Operation with Wait State . . . . .	207
6-9	$\overline{\text{ABORT}}$ Operation. . . . .	209
6-10	D2A Operation . . . . .	210
6-11	Translation Control Output . . . . .	212
6-12	Interrupt Interface Signals. . . . .	213
6-13	Global Status and Control Signals . . . . .	214
6-14	Power Management Control Signals (Assertion) . . . . .	216
6-15	Power Management Control Signals (Negation) . . . . .	216
6-16	Debug Request Input Control Signal . . . . .	217
6-17	Debug Output Control Signal . . . . .	217
7-1	Register Snoop Operation . . . . .	222
7-2	Basic Instruction Interface Operation, $\overline{\text{H\_BUSY}}$ Negated . . . . .	223
7-3	Basic Instruction Interface Operation, $\overline{\text{H\_BUSY}}$ Asserted . . . . .	224
7-4	Instruction Discard . . . . .	225
7-5	Instruction Pipeline Stall . . . . .	226
7-6	Back-to-Back HAI Instruction Execution . . . . .	226
7-7	Back-to-Back HAI Instruction Execution with Pipeline Stall. . . . .	227
7-8	Back-to-Back HAI Instruction Execution with $\overline{\text{H\_BUSY}}$ Stall . . . . .	228
7-9	$\overline{\text{H\_EXCP}}$ Operation, $\overline{\text{H\_BUSY}}$ Negated . . . . .	228
7-10	$\overline{\text{H\_EXCP}}$ Operation, HAI Busy . . . . .	229
7-11	Register Transfers to External Block with Wait State . . . . .	231
7-12	Register Transfers from External Block with Wait State . . . . .	232
7-13	Memory Transfer to External Block . . . . .	233

<b>Figure</b>	<b>Title</b>	<b>Page</b>
7-14	Memory Transfer to External Block with Access Exception . . . . .	234
7-15	Memory Transfer from External Block . . . . .	235
7-16	Delayed Memory Transfer from External Block . . . . .	236
7-17	Memory Transfer from External Block, Error Termination . . .	237
7-18	H_CALL Primitive Format . . . . .	238
7-19	H_RET Primitive Format . . . . .	239
7-20	H_LD Primitive Format . . . . .	239
7-21	H_ST Primitive Format . . . . .	240
7-22	H_EXEC Primitive Format . . . . .	241
8-1	Top-Level Tap Module and Low-Level (OnCE) TAP Module . . . . .	250
8-2	Top-Level TAP Controller State Machine . . . . .	254
8-3	IDCODE Register Bit Specification . . . . .	259
8-4	OnCE Block Diagram . . . . .	267
8-5	Low-Level (OnCE) Tap Module Data Registers (DRs) . . . . .	268
8-6	OnCE Controller . . . . .	271
8-7	OnCE Controller and Serial Interface . . . . .	273
8-8	OnCE Command Register (OCMR) . . . . .	276
8-9	OnCE Control Register (OCR) . . . . .	278
8-10	OnCE Status Register (OSR) . . . . .	282
8-11	OnCE Memory Breakpoint Logic . . . . .	284
8-12	OnCE Trace Logic Block Diagram . . . . .	287
8-13	CPU Scan Chain Register (CPUSCR) . . . . .	290
8-14	Control State Register (CTL) . . . . .	292
8-15	OnCE PC FIFO . . . . .	294
8-16	Recommended Connector Interface to JTAG/OnCE Port . . . . .	297
B-1	Pipeline Stages . . . . .	304
B-2	Pipeline Flow . . . . .	304
C-1	M210 Core Interface Signals . . . . .	311
C-2	M210S Core Interface Signals . . . . .	312

## List of Figures

Figure	Title	Page
D-1	Mux Byte Organization . . . . .	331
D-2	Internal Multiplexer Connections . . . . .	332
D-3	Instruction/Data Read Cycle . . . . .	334
D-4	Read Cycle with Wait States . . . . .	336
D-5	Write Cycle . . . . .	337
D-6	Write Cycle with Wait States. . . . .	339
D-7	Data Bus Hand-Off Operation. . . . .	340
D-8	Data Bus Hand-Off Operation with Wait State . . . . .	341
D-9	Combining DATA <sub>In</sub> and DATA <sub>Out</sub> Into a Single Bidirectional Data Bus . . . . .	341
D-10	Abort Operation . . . . .	344
D-11	Data to Address Transfer . . . . .	345
D-12	Arbitration Operation, Bus Request →Bus Grant Assertion . . . . .	348
D-13	Arbitration Operation, Bus Request →Bus Grant Assertion, Wait State on Outstanding Cycle Before Assertion, Assertion Delayed . . . . .	349
D-14	Arbitration Operation, Bus Request →Bus Grant Assertion, Wait State on Outstanding Cycle After Assertion . . . . .	351
D-15	Arbitration Operation, Bus Request →Bus Grant Negation . . . . .	352
D-16	Arbitration Operation, Back-to-Back Cycles . . . . .	353
D-17	Arbitration Operation, Bus Request →Bus Grant Negation, No Pending CPU Request. . . . .	354
D-18	Arbitration Operation, Bus Request →Bus Grant Negation, One Wait State on Alternate Master Cycle . . . . .	355
D-19	Arbitration Operation, Bus Request →Bus Grant Negation, Multiple Wait States on Alternate Master Cycle . . . . .	356
D-20	Bus Re-request with Wait State on Alternate Master Cycle . . . . .	357
D-21	Bus Re-request with Multiple Wait States on Alternate Master Cycle . . . . .	358

Figure	Title	Page
D-22	Arbitration Operation, Bus Request →Bus Grant Negation, No Pending CPU Request, Bus Re-Request . . . . .	359
D-23	Arbitration Operation, Entry into Low-Power Mode . . . . .	361
D-24	M210 Clocks and Reset Domains . . . . .	362
D-25	Reset Timing Requirements . . . . .	364
D-26	Interrupt Interface Signals. . . . .	365
D-27	Interrupt Signals . . . . .	366
D-28	Global Status and Control Signals . . . . .	367
D-29	Power Management Signals Assertion. . . . .	368
D-30	Power Management Signals Negation . . . . .	368
D-31	Wakeup Control Signal ( $\overline{\text{WAKEUP}}$ ) . . . . .	369

## List of Figures



## List of Tables

Table	Title	Page
1-1	M•CORE Instruction Set . . . . .	35
3-1	Monadic Instructions . . . . .	55
3-2	Dyadic Instructions . . . . .	56
3-3	5-Bit Immediate Instructions . . . . .	57
3-4	5-Bit Offset Immediate Instructions . . . . .	58
3-5	Opcode Map . . . . .	63
4-1	Exception Vector Assignments . . . . .	167
4-2	Exception Priority Groups . . . . .	177
4-3	Exceptions, Tracing, and $\overline{\text{BRKRQ}}$ Results . . . . .	178
5-1	Signal Index . . . . .	183
5-2	Transfer Code Encoding . . . . .	186
5-3	TSIZx Encoding . . . . .	186
5-4	$\overline{\text{LPMD}}[1:0]$ Encoding . . . . .	190
5-5	PSTATx Encoding . . . . .	191
5-6	Signal Summary . . . . .	194
6-1	Interface Requirements for Read and Write Cycles . . . . .	199
6-2	Termination Result Summary . . . . .	207
8-1	JTAG Instructions . . . . .	256
8-2	List of Pins Not Scanned in JTAG Mode . . . . .	262
8-3	Boundary-Scan Register Definition . . . . .	263
8-4	OnCE Register Addressing . . . . .	277
8-5	Sequential Control Field Settings . . . . .	279
8-6	Memory Breakpoint Control Field Settings . . . . .	281
8-7	Processor Mode Field Settings . . . . .	283

## List of Tables

Table	Title	Page
A-1	Symbols and Operators . . . . .	300
B-1	Instruction Execution Time . . . . .	305
C-1	M210/M210S Signal Descriptions. . . . .	313
C-2	M210/M210S Signal Characteristics. . . . .	315
C-3	Transfer Code Encoding. . . . .	319
C-4	Transfer Size Encoding . . . . .	320
C-5	Processor Status Encoding . . . . .	321
C-6	Low-Power Mode Encoding . . . . .	324
D-1	Interface Requirements for Read and Write Cycles . . . . .	333
D-2	Termination Result Summary . . . . .	342
D-3	M210 Reset and Clock Domains . . . . .	362
D-4	Reset Signals . . . . .	363

## Section 1. Overview

### 1.1 Contents

1.2	Introduction . . . . .	27
1.3	Features . . . . .	28
1.4	Microarchitecture Summary . . . . .	29
1.5	Programming Model . . . . .	30
1.6	Data Format Summary . . . . .	33
1.7	Operand Addressing Capabilities . . . . .	34
1.8	Instruction Set Overview . . . . .	34

### 1.2 Introduction

The 32-bit M•CORE microRISC engine represents a new line of Motorola microprocessor core products. The processor architecture has been designed for high-performance and cost-sensitive embedded control applications, with particular emphasis on reduced system power consumption. This makes the M•CORE suitable for battery-operated, portable products, as well as for highly integrated parts designed for a high temperature environment.

Total system power consumption is dictated by various components in addition to the processor core. In particular, memory power consumption (both on-chip and external) is expected to dominate overall power consumption of the core-plus-memory subsystem. With this factor in mind, the instruction set architecture (ISA) for M•CORE makes the trade-off of absolute performance capability versus total energy consumption in favor of reducing the overall energy consumption, while maintaining an acceptably high level of performance at a given clock frequency.

The M•CORE is a streamlined execution engine that provides many of the same performance enhancements as mainstream reduced instruction set computer (RISC) designs. Fixed length instruction encodings and a strict load/store architecture minimize control complexity and overhead. The goal of minimizing the overhead of memory system energy consumption is achieved by adopting a (relatively) short 16-bit instruction encoding. This choice significantly lowers the memory bandwidth needed to sustain a high rate of instruction execution.

Code density statistics for a number of applications show relative code density competitive in comparison to complex instruction set computer (CISC) designs, and implementation statistics show a large reduction in complexity and overhead relative to a CISC approach.

In addition to substantial cost and performance benefits, M•CORE also offers advantages in power consumption and power management. M•CORE minimizes power dissipation by using a fully static design, dynamic power management, and low-voltage operation. The M•CORE automatically powers down internal functional blocks that are not needed on a clock-by-clock basis. Power conservation modes are also provided for absolute power conservation on a coarser granularity.

### 1.3 Features

The main features of the M•CORE are:

- 32-bit load/store RISC architecture
- Fixed 16-bit instruction length
- 16-entry 32-bit general-purpose register file
- Efficient 4-stage execution pipeline, hidden from application software
- Single-cycle instruction execution for many instructions
- Two cycles for branches and memory access instructions
- Support for byte, half-word, and word memory accesses

- Fast interrupt support with 16-entry dedicated alternate register file
- Vectored and autovectored interrupt support

## 1.4 Microarchitecture Summary

The M•CORE instruction execution pipeline consists of these stages:

- Instruction fetch
- Instruction decode/register file read
- Execute
- Register writeback

These stages operate in an overlapped fashion, allowing single clock instruction execution for most instructions.

Sixteen general-purpose registers are provided for source operands and instruction results. Register R15 is used as the link register to hold the return address for subroutine calls, and register R0 is associated with the current stack pointer value by convention.

The execution unit consists of:

- 32-bit arithmetic/logic unit (ALU)
- 32-bit barrel shifter
- Find-first-one unit (FFO)
- Result feed-forward hardware
- Miscellaneous support hardware for multiplication and multiple register loads and stores

Arithmetic and logical operations are executed in a single cycle with the exception of the multiply, signed divide, and unsigned divide instructions. The multiply instruction is implemented with a 2-bit per clock, overlapped-scan, modified Booth algorithm with early-out capability to reduce execution time for operations with small multiplier values. The signed divide and unsigned divide instructions also have

data-dependent timing. A find-first-one unit operates in a single clock cycle.

The program counter unit has a PC incrementer and a dedicated branch address adder to minimize delays during change of flow operations. Branch target addresses are calculated in parallel with branch instruction decode, with a single pipeline bubble for taken branches and jumps. This results in an execution time of two clocks. Conditional branches that are not taken execute in a single clock.

Memory load and store operations are provided for byte, half-word, and word (32-bit) data with automatic zero extension of byte and half-word load data. These instructions can execute in two clock cycles. Load and store multiple register instructions allow low overhead context save and restore operations. These instructions can execute in  $(N + 1)$  clock cycles, where  $N$  is the numbers of registers to transfer.

A single condition code/carry (C) bit is provided for condition testing and for use in implementing arithmetic and logical operations greater than 32 bits. Typically, the C bit is set only by explicit test/comparison operations, not as a side-effect of normal instruction operation. Exceptions to this rule occur for specialized operations for which it is desirable to combine condition setting with actual computation.

A 16-entry alternate register file is provided to support low overhead interrupt exception processing. The CPU supports both vectored and autovectored interrupts.

## 1.5 Programming Model

The M•CORE programming model is defined separately for two privilege modes: supervisor and user. Certain operations are not available in user mode.

User programs can only access registers specific to the user mode; system software executing in the supervisor mode can access all registers, using the control registers to perform supervisory functions. User programs are thus restricted from accessing privileged information.

The operating system performs management and service tasks for the user programs by coordinating their activities.

Most instructions execute in either mode, but some instructions that have important system effects are privileged and can only execute in the supervisor mode. For instance, user programs cannot execute the STOP, DOZE, or WAIT instructions. To prevent a user program from entering the supervisor mode except in a controlled manner, instructions that can alter the S-bit in the program status register (PSR) are privileged. The TRAP #N instructions provide controlled access to operating system services for user programs. Access to special control registers is also precluded in user mode.

When the S bit in the PSR is set, the processor executes instructions in the supervisor mode. Bus cycles associated with an instruction indicate either supervisor or user access depending on the mode.

The processor uses the user programming model during normal user mode processing. During exception processing, the processor changes from user to supervisor mode. Exception processing saves the current value of the PSR in the EPSR or FPSR shadow control register and then sets the S bit in the PSR, forcing the processor into the supervisor mode. To return to the previous operating mode, a system routine may execute the RTE (return from exception) or RFI (return from fast interrupt) instruction as appropriate, causing the instruction pipeline to be flushed and refilled from the appropriate address space.

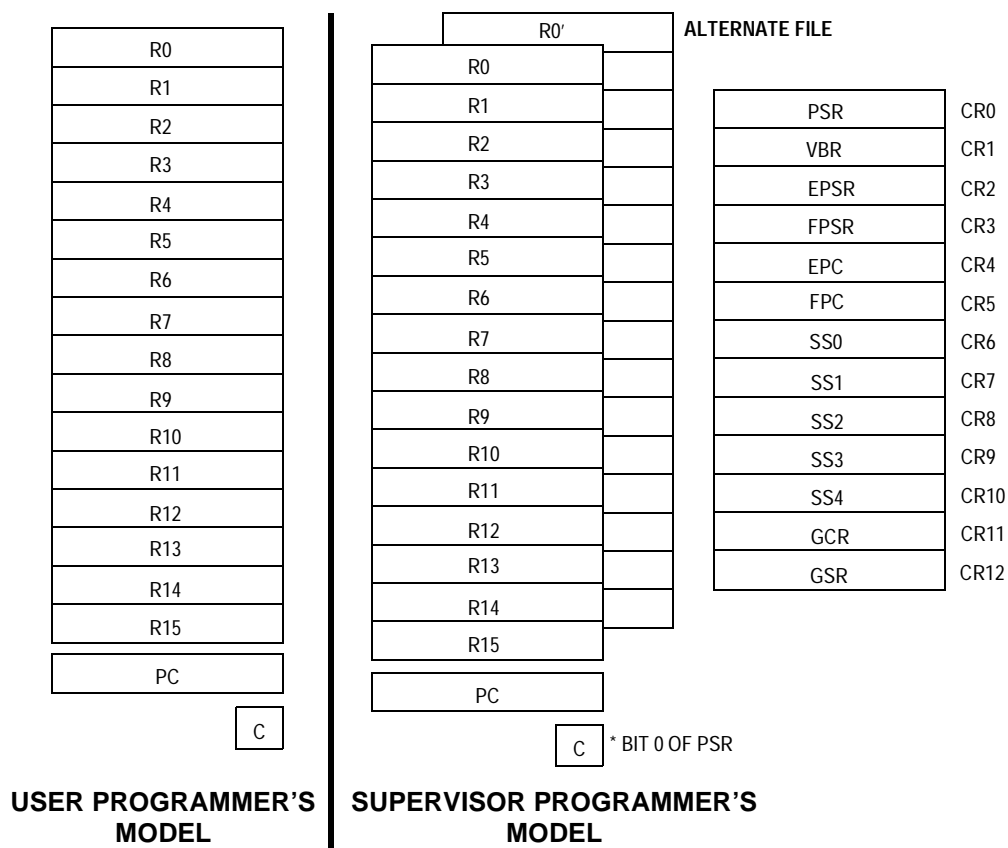
The registers depicted in the programming model (see [Figure 1-1](#)) provide operand storage and control. The registers are partitioned into two levels of privilege: user and supervisor. The user programming model consists of:

- 16 general-purpose 32-bit registers
- 32-bit program counter (PC)
- Condition/carry (C) bit

The C bit is implemented as bit 0 of the PSR. This is the only portion of the PSR accessible by the user. The supervisor programming model consists of sixteen additional 32-bit general-purpose registers (the alternate file), as well as a set of status/control registers and scratch

registers. By convention, register R15 serves as the link register for subroutine calls, and register R0 is typically used as the current stack pointer.

The alternate file is selected for use via a control bit in the PSR. The status, control, and scratch registers are accessed via the move-from-control register (MFCR) and move-to-control register (MTCR) instructions. When the alternate file is selected via the AF bit in the PSR, general-purpose operands are accessed from it. When the AF bit is cleared, operands are accessed from the normal file. This alternate file is provided to allow very low overhead context switching capability for real-time event handling.



**Figure 1-1. Programming Model**



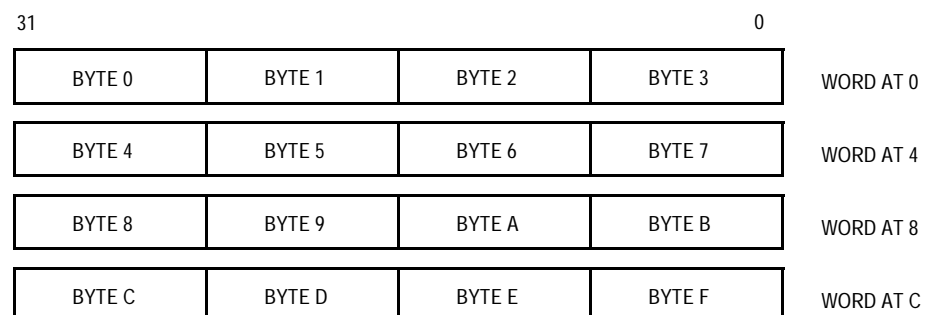
The supervisor programming model includes the PSR, which contains operation control and status information. In addition, a set of exception shadow registers are provided to save the state of the PSR and the program counter at the time an exception occurs. A separate set of shadow registers is provided for fast interrupt support to minimize context saving overhead.

Five scratch registers are provided for supervisor software use in handling exception events. A single register is provided to alter the base address of the exception vector table. Two registers are provided for global control and status.

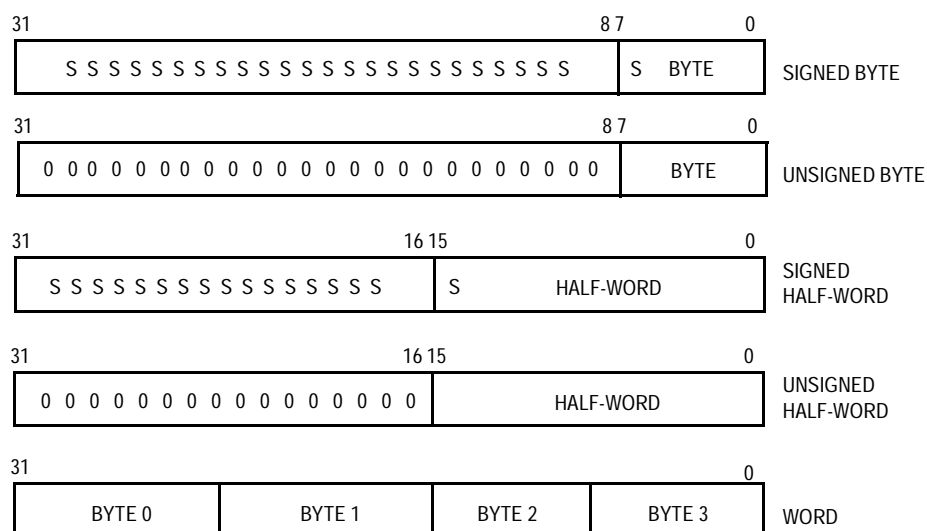
## 1.6 Data Format Summary

The operand data formats supported by the integer unit are standard two's-complement data formats. The operand size for each instruction is either explicitly encoded in the instruction (load/store instructions) or implicitly defined by the instruction operation (index operations, byte extraction). Typically, instructions operate on all 32 bits of the source operand(s) and generate a 32-bit result.

Memory is viewed from a big-endian byte ordering perspective. The most significant byte (byte 0) of word 0 is located at address 0. Bits are numbered within a word starting with bit 31 as the most significant bit.



**Figure 1-2. Data Organization in Memory**



**Figure 1-3. Data Organization in Registers**

## 1.7 Operand Addressing Capabilities

M•CORE accesses all memory operands through load and store instructions, transferring data between the general-purpose registers and memory. Register-plus-four-bit scaled displacement addressing mode is used for the load and store instructions to address byte, half-word, or word (32-bit) data.

Load and store multiple instructions allow a subset of the 16 general-purpose registers to be transferred to or from a base address pointed to by register R0 (the default stack pointer by convention).

Load and store register quadrant instructions use register indirect addressing to transfer a register quadrant to or from memory.

## 1.8 Instruction Set Overview

The instruction set is tailored to support high-level languages and is optimized for those instructions most commonly executed. A standard set of arithmetic and logical instructions is provided, as well as instruction support for bit operations, byte extraction, data movement, control flow modification, and a small set of conditionally executed

instructions which can be useful in eliminating short conditional branches.

**Table 1-1** is an alphabetized listing of the M•CORE instruction set. Refer to **Section 3. Instructions** for more details on instruction operation.

**Table 1-1. M•CORE Instruction Set (Sheet 1 of 3)**

Mnemonic	Description
ABS	Absolute Value
ADDC	Add with C bit
ADDI	Add Immediate
ADDU	Add Unsigned
AND	Logical AND
ANDI	Logical AND Immediate
ANDN	AND NOT
ASR	Arithmetic Shift Right
ASRI	Arithmetic Shift Right Immediate
ASRC	Arithmetic Shift Right, Update C Bit
BCLRI	Clear Bit
BF	Branch on Condition False
BGENI	Bit Generate Immediate
BGENR	Bit Generate Register
BKPT	Breakpoint
BMASKI	Bit Mask Immediate
BR	Branch
BREV	Bit Reverse
BSETI	Bit Set Immediate
BSR	Branch to Subroutine
BT	Branch on Condition True
BTSTI	Bit Test Immediate
CLRF	Clear Register on Condition False
CLRT	Clear Register on Condition True
CMPHS	Compare Higher or Same
CMPLT	Compare Less Than
CMPLTI	Compare Less Than Immediate
CMPNE	Compare Not Equal
CMPNEI	Compare Not Equal Immediate
DECF	Decrement on Condition False
DECGT	Decrement Register and Set Condition if Result Greater Than Zero
DECLT	Decrement Register and Set Condition if Result Less Than Zero
DECNE	Decrement Register and Set Condition if Result Not Equal to Zero
DECT	Decrement on Condition True
DIVS	Divide (Signed)
DIVU	Divide (Unsigned)
DOZE	Doze

**Table 1-1. M•CORE Instruction Set (Sheet 2 of 3)**

Mnemonic	Description
FF1	Find First One
INCF	Increment on Condition False
INCT	Increment on Condition True
IXH	Index Half-Word
IXW	Index Word
JMP	Jump
JMPI	Jump Indirect
JSR	Jump to Subroutine
JSRI	Jump to Subroutine Indirect
LD.[BHW]	Load
LDM	Load Multiple Registers
LDQ	Load Register Quadrant
LOOPT	Decrement with C-Bit Update and Branch if Condition True
LRW	Load Relative Word
LSL, LSR	Logical Shift Left and Right
LSLC, LSRC	Logical Shift Left and Right, Update C Bit
LSLI, LSRI	Logical Shift Left and Right by Immediate
MFCR	Move from Control Register
MOV	Move
MOVI	Move Immediate
MOVF	Move on Condition False
MOVT	Move on Condition True
MTCR	Move to Control Register
MULT	Multiply
MVC	Move C Bit to Register
MVCV	Move Inverted C Bit to Register
NOT	Logical Complement
OR	Logical Inclusive-OR
ROTLI	Rotate Left by Immediate
RSUB	Reverse Subtract
RSUBI	Reverse Subtract Immediate
RTE	Return from Exception
RFI	Return from Interrupt
SEXTB	Sign-Extend Byte
SEXTH	Sign-Extend Half-word
ST.[BHW]	Store
STM	Store Multiple Registers
STQ	Store Register Quadrant
STOP	Stop
SUBC	Subtract with C Bit
SUBU	Subtract
SUBI	Subtract Immediate
SYNC	Synchronize

**Table 1-1. M•CORE Instruction Set (Sheet 3 of 3)**

<b>Mnemonic</b>	<b>Description</b>
TRAP	Trap
TST	Test Operands
TSTNBZ	Test for No Byte Equal Zero
WAIT	Wait
XOR	Exclusive OR
XSR	Extended Shift Right
XTRB0	Extract Byte 0
XTRB1	Extract Byte 1
XTRB2	Extract Byte 2
XTRB3	Extract Byte 3
ZEXTB	Zero-Extend Byte
ZEXTH	Zero-Extend Half-Word



## Section 2. Registers

### 2.1 Contents

2.2	Introduction . . . . .	39
2.3	User Programming Model . . . . .	40
2.3.1	General-Purpose Registers . . . . .	41
2.3.2	Program Counter . . . . .	41
2.3.3	Condition Code/Carry Bit . . . . .	41
2.4	Supervisor Programming Model . . . . .	41
2.4.1	Alternate Register File . . . . .	43
2.4.2	Processor Status Register . . . . .	43
2.4.2.1	Updates to the PSR . . . . .	48
2.4.2.2	Exception Recognition and Processing Updates . . . . .	48
2.4.2.3	RTE and RFI Instruction Updates . . . . .	48
2.4.2.4	MTCR Instruction Updates . . . . .	49
2.4.3	Vector Base Register . . . . .	49
2.4.4	Supervisor Storage Registers . . . . .	50
2.4.5	Exception Shadow Registers . . . . .	50
2.4.6	Global Control Register . . . . .	51
2.4.7	Global Status Register . . . . .	51

### 2.2 Introduction

This section describes the organization of the M•CORE general-purpose registers (GPRs) and control registers in the user and supervisor programming models. Refer to [Section 4. Exception Processing](#) for details on the exception model.

## 2.3 User Programming Model

The user programming model's register usage, as proposed by the *Motorola Applications Binary Interface Standard* (Motorola document number M•COREABISM/AD) and shown in **Figure 2-1**, consists of these registers and the described uses:

- 16 general-purpose 32-bit registers (R[0:15])
- 32-bit program counter (PC)
- Condition code/carry flag (C bit)

R0	STACK POINTER
R1	SCRATCH
R2	FIRST ARGUMENT
R3	SECOND ARGUMENT
R4	THIRD ARGUMENT
R5	FOURTH ARGUMENT
R6	FIFTH ARGUMENT
R7	SIXTH ARGUMENT
R8	LOCAL
R9	LOCAL
R10	LOCAL
R11	LOCAL
R12	LOCAL
R13	LOCAL
R14	LOCAL
R15	LINK/SCRATCH
PC	PROGRAM COUNTER
C	

**Figure 2-1. User Programming Model**

The registers with local usage, as well as R0 (stack pointer), are preserved (by conforming compilers) during a function call. The contents of the argument and scratch registers must be saved by the calling routine if they are to be preserved. More detail about standard register usage can be found in the M•CORE ABI manual cited above.



### 2.3.1 General-Purpose Registers

The general-purpose registers contain instruction operands and results, and provide address information as well. Software and hardware register conventions have been established for subroutine linkage, parameter passing, and for a stack pointer.

### 2.3.2 Program Counter

The program counter (PC) contains the address of the currently executing instruction. During instruction execution and exception processing, the processor automatically increments the PC value or places a new value in the PC, as appropriate. For some instructions, the PC can be used as a pointer for PC-relative addressing. The low order bit of the PC is always forced to 0.

### 2.3.3 Condition Code/Carry Bit

The condition code/carry (C) bit represents a condition generated by a processor operation. The C bit can be set explicitly by comparison operations or implicitly as a result of executing extended precision arithmetic and logical operations. In addition, specialized instructions (such as the decrement, loop, and extract byte instructions) update the C bit as a result of normal execution.

## 2.4 Supervisor Programming Model

System programmers use the supervisor programming model to implement sensitive operating system functions, input/output (I/O) control, and privileged operations.

The supervisor programming model consists of the registers available to the user as well as these registers (see [Figure 2-2](#)):

- 16-entry, 32-bit alternate register file
- Processor status register (PSR)
- Vector base register (VBR)
- Exception saved PSR (EPSR)
- Fast interrupt saved PSR (FPSR)
- Exception saved program counter (EPC)
- Fast interrupt saved program counter (FPC)
- Five 32-bit supervisor scratch registers (SS0–SS4)
- 32-bit global control register (GCR)
- 32-bit global status register (GSR)

R0'	
R1'	
R2'	
R3'	
R4'	
R5'	
R6'	
R7'	
R8'	
R9'	
R10'	
R11'	
R12'	
R13'	
R14'	
R15'	
ALTERNATE FILE	

PSR	CR0
VBR	CR1
EPSR	CR2
FPSR	CR3
EPC	CR4
FPC	CR5
SS0	CR6
SS1	CR7
SS2	CR8
SS3	CR9
SS4	CR10
GCR	CR11
GSR	CR12

**Figure 2-2. Supervisor Additional Resources**

The following paragraphs describe the supervisor programming model registers. Additional information can be found in [Section 4. Exception Processing](#).

### 2.4.1 Alternate Register File

The alternate register file is provided to reduce the overhead associated with context switching and saving/restoring for time critical tasks. When selected, the alternate register file replaces the general register file for all instructions that normally use a general register. The alternate register file is active when the PSR(AF) bit is set. It is disabled and not accessible when the PSR(AF) bit is cleared. Important parameters and pointer values may be retained in the alternate file and thus are readily accessible when a high-priority task is entered.

In addition, register R0 in the alternate file serves as a stack pointer for the task, making independent stack implementation efficient.

Hardware does not prevent software from accessing the alternate file in user mode if the AF bit is set. To prevent this, system software should ensure that the AF bit is cleared before user mode is entered.

**NOTE:** *When an exception occurs, the low-order bit of the exception vector content is copied to the AF bit to select the register file to be used in processing the exception.*

### 2.4.2 Processor Status Register

The processor status register (PSR) stores the processor status (including the C bit) and control data. The control bits indicate these states for the processor:

- Trace mode (TM bits)
- Supervisor or user mode (S bit)
- Normal or alternate file state (AF)

They also indicate whether exception shadow registers are available for context saving, and whether interrupts are enabled. The PSR can be accessed in supervisor mode only. See [Figure 2-3](#).

	Bit 31	30	29	28	27	26	25	Bit 24
Read:	S	0	SP		U3	U2	U1	U0
Write:								
Reset:	1	0	0	0	0	0	0	0
	Bit 23	22	21	20	19	18	17	Bit 16
Read:	0	VEC						
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 15	14	13	12	11	10	9	Bit 8
Read:	TM		TP <sup>(1)</sup>	TC	0	SC	MM	EE
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	IC	IE	0	FE	0	0	AF	C
Write:								
Reset:	0	0	0	0	0	0	0	0

1. This bit exists in the PSR shadow register only. In the PSR, this bit is hardwired to 0.

**Figure 2-3. Processor Status Register (PSR)**

**S** — Supervisor Mode Bit

1 = Processor is operating in supervisor mode

0 = Processor is operating in user mode

This bit is set by reset. The bit is also set by hardware when exception processing is initiated.

**SP[1:0]** — Spare Bits

These bits are spare bits. They are cleared by reset and are currently undefined for other exceptions. These bits should be written only to 0 to avoid undefined behavior.

### U[3:0] — Hardware Accelerator Control Bits

The U[3:0] bits control execution of the hardware accelerator instructions. Each bit corresponds to one encoded value of the UU field of a hardware accelerator opcode. If the appropriate bit is cleared, an attempt to execute a corresponding hardware accelerator instruction is aborted, and a disabled hardware accelerator exception is taken. These bits are cleared by hard reset and are unaffected by other exceptions.

Refer to the appropriate microcontroller user's manual for details of the hardware accelerator instructions. In addition, [4.6.12 Hardware Accelerator Exception \(Vector Offset 0x30\)](#) describes the disabled hardware accelerator exception.

### VEC[6:0] — Vector Number Field

This seven-bit field is written with the vector number used to fetch an exception vector when an exception occurs. This field is cleared by reset.

### TM[1:0] — Trace Mode Field

When this field is non-zero, the M•CORE is placed in trace mode. A trace exception may be taken after the execution of each instruction or only after potential change of flow instructions. This field is cleared by reset and also by hardware when exception processing is initiated. The field is defined as:

- 11 = Change of flow trace mode
- 10 = Reserved
- 01 = Instruction trace mode
- 00 = Normal execution

Refer to [4.6.7 Trace Exception \(Vector Offset 0x18\)](#) for more details on trace operation.

### TP — Trace Pending Bit

This bit is set in the appropriate PSR shadow register as part of exception recognition when the M•CORE is in instruction trace mode, a trace exception is pending, and another exception takes priority over the trace exception at an instruction boundary. Setting this bit in

the PSR has no effect, as the bit is hardwired to 0. A trace exception is taken after the execution of return from exception (RTE) or return from interrupt (RFI) if this bit is set in the appropriate shadow PSR.

### TC — Translation Control Bit

This bit allows control over address translation of instruction and data accesses by an external memory management unit. When this bit is set, the  $\overline{TE}$  output signal is asserted, indicating that access addresses should be translated by an optional external memory management unit. (This signal can also be used for an alternate function.) When an exception occurs, this bit is cleared. This bit is also cleared by reset.

### SC — Spare Control Bit

This bit is a spare control bit. This bit is cleared when an exception occurs and by reset.

### MM — Misalignment Exception Mask Bit

1 = Alignment restrictions are ignored, and the lower address bit(s) are ignored and assumed to be 0.

0 = Loading and storing instructions to a misaligned address causes a misalignment exception to occur instead of a memory access.

This bit does not affect exceptions for the jump indirect (JMPI) or jump-to-subroutine indirect (JSRI) instructions. This bit is cleared by reset and is unaffected by other exceptions.

### EE — Exception Enable Bit

1 = The EPSR and EPC shadow registers are available to save the exception state.

0 = Shadowing of the PSR and PC by the EPSR and EPC registers on an exception is assumed to result in an unrecoverable error.

This bit is cleared by reset. Hardware clears this bit on any exception (including a fast interrupt exception) to indicate that processor context for the exception cannot be overwritten in a recoverable manner.

#### IC — Interrupt Control Bit

1 = Valid pending interrupt ( $\overline{\text{INT}}$  or  $\overline{\text{FINT}}$ ) is allowed to cause a long latency, multi-cycle instruction (divide signed (DIVS), divide unsigned (DIVU), load multiple registers (LDM), load register quadrant (LDQ), multiply (MULT), store multiple registers (STM), or store register quadrant (STQ)) to be interrupted before completion. The instruction will be restarted on return from the interrupt handler. For the load multiple registers (LDM), load register quadrant (LDQ), store multiple registers (STM), and store register quadrant (STQ) instructions, an access in progress will complete prior to interruption.

0 = Interrupts are only recognized on instruction boundaries.

This bit is cleared by soft reset. It is not affected by other exceptions.

#### IE — Interrupt Enable Bit

1 = The  $\overline{\text{INT}}$  interrupt is sampled.

0 = The  $\overline{\text{INT}}$  interrupt input is disabled.

This bit is cleared by soft reset. It is also cleared when any exception occurs to disable interrupts signalled by the  $\overline{\text{INT}}$  input.

#### FE — Fast Interrupt Enable Bit

1 = The FPSR and FPC registers are unfrozen, shadowing by these registers is enabled, and the  $\overline{\text{FINT}}$  interrupt can be sampled.

0 = The FPSR and FPC shadow registers are frozen and the  $\overline{\text{FINT}}$  interrupt input is disabled.

This bit is cleared by reset and soft reset. It is also cleared when a fast interrupt exception occurs. FE is unaffected by other exceptions.

#### AF — Alternate File Enable Bit

1 = The alternate file is enabled.

0 = The general file is enabled.

When an exception occurs, the low-order bit of the exception vector content is copied to this bit to select the file to be used in processing the exception. Although hardware clears this bit on reset, it is overwritten with the low order bit in the fetched reset vector.

### C — Condition Code/Carry Bit

The C bit is used as a condition code or carry bit for certain instructions. This bit is undefined following reset or after being copied into the appropriate shadow PSR register after any other exception.

### Bits 30, 28, 23, 11, 10, 5, 3, and 2

These bits are reserved for future use and must always be written to 0.

#### 2.4.2.1 Updates to the PSR

The content of the PSR can be modified by exception recognition and processing, the return-from-exception (RTE) and return-from-interrupt (RFI) instructions, and the move-to-control register (MTCR) instruction. Each affects the PSR in different ways.

#### 2.4.2.2 Exception Recognition and Processing Updates

Updates to the PSR occur as part of the exception recognition and vectoring process. These updates may affect the S, TM, TC, VEC, IE, FE, EE, and AF bits or fields. Changes to the S, TM, TC, IE, FE and EE bits are effective prior to the fetch of the exception vector. Changes to the VEC and AF bits are effective prior to the execution of the first instruction of a handler.

#### 2.4.2.3 RTE and RFI Instruction Updates

The RTE and RFI instructions update the PSR. These updates may change the state of all bits in the PSR. Changes to the S, DB, TM, TP, TC, IE, FE and EE bits are effective prior to the fetch of the instruction at the return PC location. Changes to the U[3:0], VEC, MM, IC, AF, and C bits are effective prior to the execution of this instruction.



#### 2.4.2.4 MTCR Instruction Updates

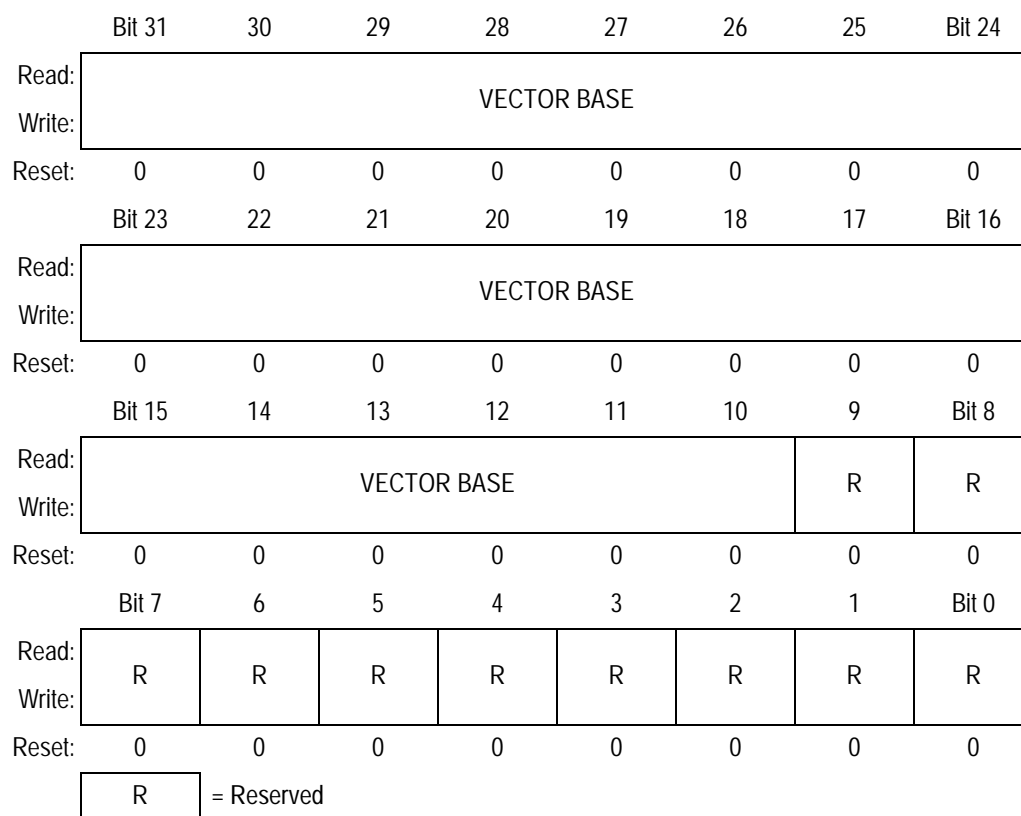
The MTCR instruction updates the PSR when CR0 is the control register destination. These updates may change the state of all bits in the PSR. However, due to the pipelined nature of an implementation, not all changes are reflected immediately. In particular, prefetching and decoding of instructions following the MTCR instruction may involve use of the prior state of the S, DB, TM, TC, EE, and AF bits in the PSR. Changes to these bits may not become effective for several instructions past the MTCR instruction. To minimize the uncertainty of this, the MTCR instruction should be followed with an unconditional branch instruction with a displacement value of 0. All instructions following the branch will be fetched, decoded, and executed with the updates made to the PSR with the MTCR instruction.

An alternative instruction to follow an MTCR to the PSR is a low-power mode instruction (DOZE, WAIT, or STOP).

Interrupt recognition is delayed following an MTCR-to-PSR instruction to allow the following instruction to execute prior to interrupt exception events. This allows a low-power mode instruction which follows an MTCR to the PSR (which enables interrupts) to begin execution before a pending interrupt is recognized.

#### 2.4.3 Vector Base Register

The vector base register holds the base address of the exception vector table. This register contains 22 high-order bits, with the low-order ten bits hardwired to 0. (This allows for possible future expansion of the vector table.) This register is cleared when a reset or soft reset exception occurs.



**Figure 2-4. Vector Base Register (VBR)**

## 2.4.4 Supervisor Storage Registers

The CPU core contains a set of five 32-bit supervisor storage registers. These registers are provided for supervisor software to store data and pointers and to assist in exception state saving, protected from user mode software. Software determines their use and contents. Typically, one of these registers is used as a supervisor stack pointer storage location. These registers are accessed through the MTCR and move-from-control register (MFCR) instructions.

## 2.4.5 Exception Shadow Registers

The EPSR, EPC, FPSR, and FPC registers are used during exceptions to store execution context of the processor. Refer to [Section 4. Exception Processing](#) for details.

## 2.4.6 Global Control Register

The 32-bit global control register (GCR) is used for global control of devices and events external to the core. Thirty-two parallel outputs are provided at the core interface for implementation-defined control purposes. Power management, device control, event scheduling, and other basic control functions can be easily implemented using the GCR. It is beyond the scope of this document to specify the exact functions of the bits in this register. Refer to the appropriate microcontroller user's guide for details. This register can be read and written.

**NOTE:** *Most M210/M210S core devices do not use this register for any special purpose and therefore the register is not updated by hardware.*

## 2.4.7 Global Status Register

The 32-bit global status register (GSR) is used for global status reporting by devices and events external to the core. Thirty-two parallel inputs are provided at the core interface for implementation-defined purposes. Device status and other events can be easily monitored using the GSR. It is beyond the scope of this document to specify the exact meaning of the bits in this register. Refer to the appropriate microcontroller user's manual for details.

This register is read only. Writes to this register while in supervisor mode are ignored. Writes to this register in user mode result in a privilege violation exception.

**NOTE:** *Most M210/M210S core devices do not use this register for any special purpose and therefore the register is not updated by hardware.*



## Section 3. Instructions

### 3.1 Contents

3.2	Introduction . . . . .	54
3.3	Instruction Types and Addressing Modes . . . . .	54
3.3.1	Register-to-Register Instructions . . . . .	54
3.3.1.1	Monadic Register Addressing Mode . . . . .	55
3.3.1.2	Dyadic Register Addressing Mode . . . . .	56
3.3.1.3	Register with 5-Bit Immediate Mode . . . . .	57
3.3.1.4	Register with 5-Bit Offset Immediate Mode . . . . .	58
3.3.1.5	Register with 7-Bit Immediate Mode . . . . .	58
3.3.1.6	Control Register Addressing Mode . . . . .	59
3.3.2	Data Memory Access Instructions . . . . .	59
3.3.2.1	Scaled 4-Bit Immediate Addressing Mode . . . . .	59
3.3.2.2	Load/Store Register Quadrant Mode . . . . .	59
3.3.2.3	Load/Store Multiple Register Mode . . . . .	60
3.3.2.4	Load Relative Word Mode . . . . .	60
3.3.3	Flow Control Instructions . . . . .	61
3.3.3.1	Scaled 11-Bit Displacement Mode . . . . .	61
3.3.3.2	Register Addressing Mode . . . . .	61
3.3.3.3	Indirect Mode . . . . .	61
3.3.3.4	Register with 4-Bit Negative Displacement Mode . . . . .	62
3.4	Opcode Map . . . . .	63
3.5	Instruction Set . . . . .	67

## 3.2 Introduction

This section describes the M•CORE instruction set and provides a reference to M•CORE instructions. An opcode map is provided along with individual instruction pages.

## 3.3 Instruction Types and Addressing Modes

All M•CORE instructions are 16 bits in length. Immediate operands and displacements are encoded in the instruction word. Other operands are located in registers which can be moved to and from memory with load and store instructions.

M•CORE implements three types of instructions:

- Flow control
- Data memory access
- Register-to-register operations

Flow control instructions alter the sequential flow of instruction execution. Data memory access instructions load or store operands to or from the general-purpose registers. Register-to-register instructions perform operations on general-purpose registers, or are used to access control registers. Instruction formats are shown here.

### 3.3.1 Register-to-Register Instructions

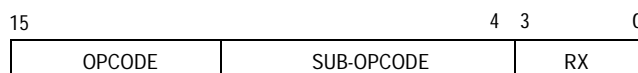
M•CORE supports five addressing modes for register-to-register instructions as described in this subsection.

### 3.3.1.1 Monadic Register Addressing Mode

Monadic register addressing uses a single 4-bit register field to specify the source/destination for an operation. Instructions with this format are shown in [Table 3-1](#). [Figure 3-1](#) shows the monadic format.

**Table 3-1. Monadic Instructions**

Mnemonic	Description
ABS ASRC	Absolute Value Arithmetic Shift Right, Update C Bit
BREV	Bit Reverse
CLRF CLRT	Clear Register on Condition False Clear Register on Condition True
DECF DECGT DECLT DECNE DECT	Decrement on Condition False Decrement Register and Set Condition if Result Greater Than Zero Decrement Register and Set Condition if Result Less Than Zero Decrement Register and Set Condition if Result Not Equal to Zero Decrement on Condition True
FF1	Find First One
INCF INCT	Increment on Condition False Increment on Condition True
LSLC, LSRC	Logical Shift Left and Right, Update C Bit
MVC MVCV	Move C Bit to Register Move Inverted C Bit to Register
NOT	Logical Complement
SEXTB SEXTH	Sign-Extend Byte Sign-Extend Half-Word
TSTNBZ	Test for No Byte Equal Zero
XSR XEXTB XEXTH	Extended Shift Right Zero-Extend Byte Zero-Extend Half-Word



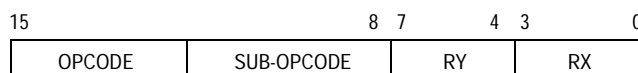
**Figure 3-1. Monadic Format**

## 3.3.1.2 Dyadic Register Addressing Mode

Dyadic register addressing uses two 4-bit register fields encoded in the instruction to specify a source register and a source/destination register. For some instructions, only a single source value is used; the second register specifier is used as a destination specifier only. Instructions with this format are shown in [Table 3-2](#). [Figure 3-2](#) shows the dyadic format.

**Table 3-2. Dyadic Instructions**

Mnemonic	Description
ADDC ADDU AND ANDN ASR	Add with C bit Add Unsigned Logical AND AND NOT Arithmetic Shift Right
BGENR	Bit Generate Register
CMPHS CMPLT CMPNE	Compare Higher or Same Compare Less Than Compare Not Equal
IXH IXW	Index Half-Word Index Word
LSLI, LSRI	Logical Shift Left and Right by Immediate
MOV MOVF MOVT MULT	Move Move on Condition False Move on Condition True Multiply
OR	Logical Inclusive-OR
RSUB	Reverse Subtract
SUBC SUBU	Subtract with C Bit Subtract
TST	Test Operands
XOR	Exclusive OR
ZEXTB ZEXTH	Zero-Extend Byte Zero-Extend Half-word



**Figure 3-2. Dyadic Format**



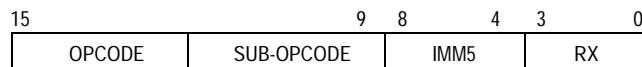
### 3.3.1.3 Register with 5-Bit Immediate Mode

Register with 5-bit immediate addressing uses a 4-bit register field encoded in the instruction to specify a source/destination register and a 5-bit field to specify an unsigned immediate value as the second source operand. Instructions with this format are shown in [Table 3-3](#).

[Figure 3-3](#) shows the 5-bit immediate format.

**Table 3-3. 5-Bit Immediate Instructions**

Mnemonic	Description
ANDI ASRI	Logical AND Immediate Arithmetic Shift Right Immediate
BCLRI BGENI BMASKI BSETI BTSTI	Clear Bit Bit Generate Immediate Bit Mask Immediate Bit Set Immediate Bit Test Immediate
CMPNEI	Compare Not Equal Immediate
LSLI, LSRI	Logical Shift Left and Right by Immediate
ROTLI RSUBI	Rotate Left by Immediate Reverse Subtract Immediate
SUBI	Subtract Immediate



**Figure 3-3. Register with 5-Bit Immediate Format**

## 3.3.1.4 Register with 5-Bit Offset Immediate Mode

Register with 5-bit offset immediate addressing uses a 4-bit register field encoded in the instruction to specify a source/destination register, and a 5-bit field to specify an unsigned immediate value as the second source operand. The binary encoding for the immediate value is offset by one from the actual immediate value, thus, offset immediate values fall in the range from 1 to 32, corresponding to binary encodings of 0 to 31. Instructions with this format are shown in [Table 3-4](#). [Figure 3-4](#) shows the 5-bit offset immediate format.

**Table 3-4. 5-Bit Offset Immediate Instructions**

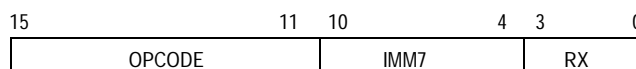
Mnemonic	Description
ADDI	Add Immediate
CMPLTI	Compare Less Than Immediate
SUBI	Subtract Immediate



**Figure 3-4. Register with 5-Bit Offset Immediate Format**

## 3.3.1.5 Register with 7-Bit Immediate Mode

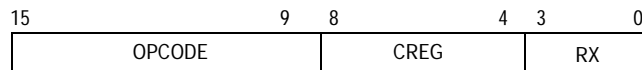
Register with 7-bit immediate addressing uses a 4-bit register field encoded in the instruction to specify a destination register and a 7-bit field to specify an unsigned immediate value as the source operand. Only the move immediate (MOVI) instruction uses this format. [Figure 3-5](#) shows the 7-bit immediate format.



**Figure 3-5. Register with 7-Bit Immediate Format**

### 3.3.1.6 Control Register Addressing Mode

Control register addressing uses a 4-bit register field encoded in the instruction to specify a general-purpose source/destination register and a 5-bit field to specify a control register. Only the move from control register (MFCR) and move to control register (MTCR) instructions use this format. **Figure 3-6** shows the control register addressing format.



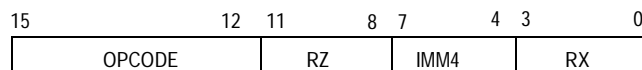
**Figure 3-6. Control Register Addressing Format**

## 3.3.2 Data Memory Access Instructions

M•CORE supports four addressing modes for accessing memory-based operands as described in this subsection.

### 3.3.2.1 Scaled 4-Bit Immediate Addressing Mode

The load (LD) and store (ST) instructions use this addressing mode for effective address calculations. The contents of the general-purpose register specified by the RX instruction field are added to the unsigned 4-bit immediate field which has been scaled (shifted left) according to the size of the memory access to form the effective address for the access. Register RZ serves as the destination register for loads and as the source of store data for stores. **Figure 3-7** shows the scaled 4-bit immediate format.

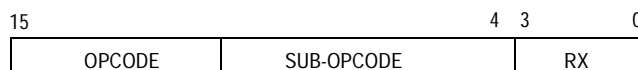


**Figure 3-7. Scaled 4-Bit Immediate Format**

### 3.3.2.2 Load/Store Register Quadrant Mode

The load register quadrant (LDQ) and store register quadrant (STQ) instructions use this mode to transfer a contiguous set of registers to or

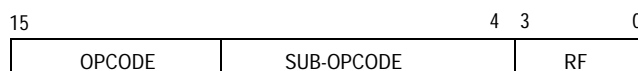
from the memory location pointed to by the contents of general-purpose register RX. Registers R4 through R7 are transferred in ascending order to or from memory. **Figure 3-8** shows the load/store register quadrant format.



**Figure 3-8. Load/Store Register Quadrant Format**

### 3.3.2.3 Load/Store Multiple Register Mode

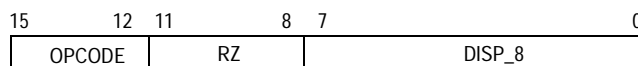
The load multiple registers (LDM) and store multiple register (STM) instructions use this mode to transfer a contiguous set of registers to or from the memory location pointed to by the contents of general-purpose register R0. The RF instruction field specifies the first register in the list to be transferred. Registers RF through R15 are transferred in ascending order to or from memory. **Figure 3-9** shows the load/store multiple registers format.



**Figure 3-9. Load/Store Multiple Registers Format**

### 3.3.2.4 Load Relative Word Mode

The load relative word (LRW) instruction uses this format to address a 32-bit word located relative to the program counter (PC). The effective address is obtained by adding the zero-extended value of the 8-bit displacement field, scaled by four, to the value of PC + 2. The lower two bits of this value are truncated to 00, and a word is fetched from this location into the general-purpose register specified by the RZ instruction field. **Figure 3-10** shows the load relative word format.



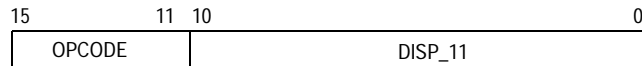
**Figure 3-10. Load Relative Word Format**

### 3.3.3 Flow Control Instructions

M•CORE supports four addressing modes for flow control instructions.

#### 3.3.3.1 Scaled 11-Bit Displacement Mode

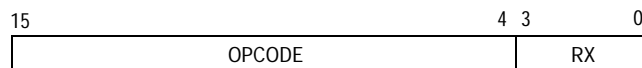
The branch (BR), branch-on-condition false (BF), and branch-to-subroutine (BSR) instructions use this addressing mode for branch target address calculations. The content of the PC plus two (PC + 2) are added to the sign-extended 11-bit displacement field which has been scaled by two (shifted left by one bit). **Figure 3-11** shows the scaled 11-bit displacement format.



**Figure 3-11. Scaled 11-Bit Displacement Format**

#### 3.3.3.2 Register Addressing Mode

The jump (JMP) and jump-to-subroutine (JSR) instructions use this addressing mode for effective address calculations. The target address is contained in the general-purpose register specified by the RX instruction field. **Figure 3-12** shows the register addressing format.

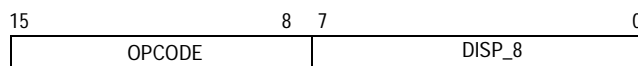


**Figure 3-12. Register Addressing Format**

#### 3.3.3.3 Indirect Mode

The jump indirect (JMPI) and jump-to-subroutine indirect (JSRI) instructions use this format to address a 32-bit word located relative to the PC. The effective address is obtained by adding the zero-extended value of the 8-bit displacement field, scaled by four, to the value of PC + 2. The lower two bits of this value are truncated to 00, and a word is fetched from this location and loaded into the PC. If the value of the

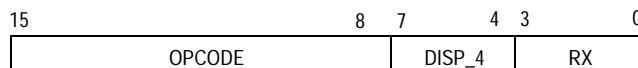
fetches word is even, instruction execution proceeds from the new PC value; otherwise a misaligned access exception is taken. [Figure 3-13](#) shows the indirect format.



**Figure 3-13. Indirect Format**

### 3.3.3.4 Register with 4-Bit Negative Displacement Mode

The decrement with C-bit update and branch if condition true (LOOP) instruction uses this addressing mode for effective address calculations. The target address is formed by extending the DISP\_4 instruction field with ones, shifting this negative number left by one to scale by two, and adding the resultant displacement to PC + 2. A count value is held in the general-purpose register specified by the RX instruction field. [Figure 3-14](#) shows the 4-bit displacement addressing format.



**Figure 3-14. Register with 4-Bit Displacement Addressing Format**

### 3.4 Opcode Map

**Table 3-5** is the opcode map for M•CORE.

**Table 3-5. Opcode Map (Sheet 1 of 5)**

Opcode																Mnemonic
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	BKPT
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SYNC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	RTE
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	RFI
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	STOP
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	WAIT
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	DOZE
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	—
0	0	0	0	0	0	0	0	0	0	0	0	1	0	i	i	TRAP #II
0	0	0	0	0	0	0	0	0	0	0	0	1	1	x	x	—
0	0	0	0	0	0	0	0	0	0	0	1	r	r	r	r	MVC
0	0	0	0	0	0	0	0	0	0	1	1	r	r	r	r	MVCV
0	0	0	0	0	0	0	0	0	1	0	0	r	r	r	r	LDQ
0	0	0	0	0	0	0	0	0	1	0	1	r	r	r	r	STQ
0	0	0	0	0	0	0	0	0	1	1	0	r	r	r	r	LDM
0	0	0	0	0	0	0	0	0	1	1	1	r	r	r	r	STM
0	0	0	0	0	0	0	0	1	0	0	0	r	r	r	r	DECT
0	0	0	0	0	0	0	0	1	0	0	1	r	r	r	r	DECF
0	0	0	0	0	0	0	0	1	0	1	0	r	r	r	r	INCT
0	0	0	0	0	0	0	0	1	0	1	1	r	r	r	r	INCF
0	0	0	0	0	0	0	0	1	1	0	0	r	r	r	r	JMP
0	0	0	0	0	0	0	0	1	1	0	1	r	r	r	r	JSR
0	0	0	0	0	0	0	0	1	1	1	0	r	r	r	r	FF1
0	0	0	0	0	0	0	0	1	1	1	1	r	r	r	r	BREV
0	0	0	0	0	0	0	1	0	0	0	0	r	r	r	r	XTRB3
Key																
rrrr - RX field ssss - RY field zzzz - RZ field ffff - Rfirst field cccc - control register specifier iii..i - one of several immediate fields								ddddddddddd - branch displacement bbbb - loopt displacement uu- accelerator unit ee..e - execution code nnn - register count p - update option x..x - undefined fields								

Table 3-5. Opcode Map (Sheet 2 of 5)

Opcode															Mnemonic
0	0	0	0	0	0	0	1	0	0	0	1	r	r	r	XTRB2
0	0	0	0	0	0	0	1	0	0	1	0	r	r	r	XTRB1
0	0	0	0	0	0	0	1	0	0	1	1	r	r	r	XTRB0
0	0	0	0	0	0	0	1	0	1	0	0	r	r	r	ZEXTB
0	0	0	0	0	0	0	1	0	1	0	1	r	r	r	SEXTB
0	0	0	0	0	0	0	1	0	1	1	0	r	r	r	ZEXTH
0	0	0	0	0	0	0	1	0	1	1	1	r	r	r	SEXTH
0	0	0	0	0	0	0	1	1	0	0	0	r	r	r	DECLT
0	0	0	0	0	0	0	1	1	0	0	1	r	r	r	TSTNBZ
0	0	0	0	0	0	0	1	1	0	1	0	r	r	r	DECGT
0	0	0	0	0	0	0	1	1	0	1	1	r	r	r	DECNE
0	0	0	0	0	0	0	1	1	1	0	0	r	r	r	CLRT
0	0	0	0	0	0	0	1	1	1	0	1	r	r	r	CLRF
0	0	0	0	0	0	0	1	1	1	1	0	r	r	r	ABS
0	0	0	0	0	0	0	1	1	1	1	1	r	r	r	NOT
0	0	0	0	0	0	1	0	s	s	s	s	r	r	r	MOVT
0	0	0	0	0	0	1	1	s	s	s	s	r	r	r	MULT
0	0	0	0	0	1	0	0	s	s	s	s	b	b	b	LOOP
0	0	0	0	0	1	0	1	s	s	s	s	r	r	r	SUBU
0	0	0	0	0	1	1	0	s	s	s	s	r	r	r	ADDC
0	0	0	0	0	1	1	1	s	s	s	s	r	r	r	SUBC
0	0	0	0	1	0	0	0	s	s	s	s	r	r	r	—
0	0	0	0	1	0	0	1	s	s	s	s	r	r	r	—
0	0	0	0	1	0	1	0	s	s	s	s	r	r	r	MOVF
0	0	0	0	1	0	1	1	s	s	s	s	r	r	r	LSR
0	0	0	0	1	1	0	0	s	s	s	s	r	r	r	CMPHS
0	0	0	0	1	1	0	1	s	s	s	s	r	r	r	CMPLT
0	0	0	0	1	1	1	0	s	s	s	s	r	r	r	TST
0	0	0	0	1	1	1	1	s	s	s	s	r	r	r	CMPNE
Key															
rrrr - RX field ssss - RY field zzzz - RZ field ffff - Rfirst field cccc - control register specifier iii.i - one of several immediate fields								dddddddddd - branch displacement bbbb - loopt displacement uu- accelerator unit ee..e - execution code nnn - register count p - update option x..x - undefined fields							



**Table 3-5. Opcode Map (Sheet 3 of 5)**

Opcode																Mnemonic
0	0	0	1	0	0	0	c	c	c	c	c	r	r	r	r	MFCR
0	0	0	1	0	0	1	0	s	s	s	s	r	r	r	r	MOV
0	0	0	1	0	0	1	1	s	s	s	s	r	r	r	r	BGENR
0	0	0	1	0	1	0	0	s	s	s	s	r	r	r	r	RSUB
0	0	0	1	0	1	0	1	s	s	s	s	r	r	r	r	IXW
0	0	0	1	0	1	1	0	s	s	s	s	r	r	r	r	AND
0	0	0	1	0	1	1	1	s	s	s	s	r	r	r	r	XOR
0	0	0	1	1	0	0	c	c	c	c	c	r	r	r	r	MTCR
0	0	0	1	1	0	1	0	s	s	s	s	r	r	r	r	ASR
0	0	0	1	1	0	1	1	s	s	s	s	r	r	r	r	LSL
0	0	0	1	1	1	0	0	s	s	s	s	r	r	r	r	ADDU
0	0	0	1	1	1	0	1	s	s	s	s	r	r	r	r	IXH
0	0	0	1	1	1	1	0	s	s	s	s	r	r	r	r	OR
0	0	0	1	1	1	1	1	s	s	s	s	r	r	r	r	ANDN
0	0	1	0	0	0	0	i	i	i	i	i	r	r	r	r	ADDI
0	0	1	0	0	0	1	i	i	i	i	i	r	r	r	r	CMPLTI
0	0	1	0	0	1	0	i	i	i	i	i	r	r	r	r	SUBI
0	0	1	0	0	1	1	i	i	i	i	i	r	r	r	r	—
0	0	1	0	1	0	0	i	i	i	i	i	r	r	r	r	RSUBI
0	0	1	0	1	0	1	i	i	i	i	i	r	r	r	r	CMPNEI
0	0	1	0	1	1	0	0	0	0	0	0	r	r	r	r	BMASKI #32 (SET)
0	0	1	0	1	1	0	0	0	0	0	1	r	r	r	r	DIVU
0	0	1	0	1	1	0	0	0	0	1	x	r	r	r	r	—
0	0	1	0	1	1	0	0	0	1	x	x	r	r	r	r	—
0	0	1	0	1	1	0	0	1	i	i	i	r	r	r	r	BMASKI
0	0	1	0	1	1	0	1	i	i	i	i	r	r	r	r	BMASKI
0	0	1	0	1	1	1	i	i	i	i	i	r	r	r	r	ANDI
0	0	1	1	0	0	0	i	i	i	i	i	r	r	r	r	BCLRI
0	0	1	1	0	0	1	0	0	0	0	0	r	r	r	r	—
Key																
rrrr - RX field ssss - RY field zzzz - RZ field ffff - Rfirst field ccccc - control register specifier iii.i - one of several immediate fields								ddddddddddd - branch displacement bbbb - loopt displacement uu- accelerator unit ee..e - execution code nnn - register count p - update option x..x - undefined fields								

Table 3-5. Opcode Map (Sheet 4 of 5)

Opcode																Mnemonic
0	0	1	1	0	0	1	0	0	0	0	1	r	r	r	r	DIVS
0	0	1	0	1	1	0	0	0	0	1	x	r	r	r	r	—
0	0	1	0	1	1	0	0	0	1	0	x	r	r	r	r	—
0	0	1	0	1	1	0	0	0	1	1	0	r	r	r	r	—
0	0	1	1	0	0	1	0	0	1	1	1	r	r	r	r	BGENI
0	0	1	1	0	0	1	0	1	i	i	i	r	r	r	r	BGENI
0	0	1	1	0	0	1	1	i	i	i	i	r	r	r	r	BGENI
0	0	1	1	0	1	0	i	i	i	i	i	r	r	r	r	BSETI
0	0	1	1	0	1	1	i	i	i	i	i	r	r	r	r	BTSTI
0	0	1	1	1	0	0	0	0	0	0	0	r	r	r	r	XSR
0	0	1	1	1	0	0	i	i	i	i	i	r	r	r	r	ROTLI
0	0	1	1	1	0	1	0	0	0	0	0	r	r	r	r	ASRC
0	0	1	1	1	0	1	i	i	i	i	i	r	r	r	r	ASRI
0	0	1	1	1	1	0	0	0	0	0	0	r	r	r	r	LSLC
0	0	1	1	1	1	0	i	i	i	i	i	r	r	r	r	LSLI
0	0	1	1	1	1	1	0	0	0	0	0	r	r	r	r	LSRC
0	0	1	1	1	1	1	i	i	i	i	i	r	r	r	r	LSRI
0	1	0	0	u	u	0	0	e	e	e	e	e	e	e	e	H_EXEC
0	1	0	0	u	u	0	1	0	n	n	n	e	e	e	e	H_RET
0	1	0	0	u	u	0	1	1	n	n	n	e	e	e	e	H_CALL
0	1	0	0	u	u	1	0	0	p	i	i	r	r	r	r	H_LD
0	1	0	0	u	u	1	0	1	p	i	i	r	r	r	r	H_ST
0	1	0	0	u	u	1	1	0	p	i	i	r	r	r	r	H_LD.H
0	1	0	0	u	u	1	1	1	p	i	i	r	r	r	r	H_ST.H
0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	—
0	1	1	0	0	i	i	i	i	i	i	i	r	r	r	r	MOVI
0	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	—
0	1	1	1	z	z	z	z	d	d	d	d	d	d	d	d	LRW
0	1	1	1	0	0	0	0	d	d	d	d	d	d	d	d	JMPI
Key																
rrrr - RX field ssss - RY field zzzz - RZ field ffff - Rfirst field cccc - control register specifier iii.i - one of several immediate fields								ddddddddddd - branch displacement bbbb - loopt displacement uu- accelerator unit ee..e - execution code nnn - register count p - update option x..x - undefined fields								

**Table 3-5. Opcode Map (Sheet 5 of 5)**

Opcode																Mnemonic
0	1	1	1	1	1	1	1	d	d	d	d	d	d	d	d	JSRI
1	0	0	0	z	z	z	z	i	i	i	i	r	r	r	r	LD
1	0	0	1	z	z	z	z	i	i	i	i	r	r	r	r	ST
1	0	1	0	z	z	z	z	i	i	i	i	r	r	r	r	LD.B
1	0	1	1	z	z	z	z	i	i	i	i	r	r	r	r	ST.B
1	1	0	0	z	z	z	z	i	i	i	i	r	r	r	r	LD.H
1	1	0	1	z	z	z	z	i	i	i	i	r	r	r	r	ST.H
1	1	1	0	0	d	d	d	d	d	d	d	d	d	d	d	BT
1	1	1	0	1	d	d	d	d	d	d	d	d	d	d	d	BF
1	1	1	1	0	d	d	d	d	d	d	d	d	d	d	d	BR
1	1	1	1	1	d	d	d	d	d	d	d	d	d	d	d	BSR
Key																
rrrr - RX field ssss - RY field zzzz - RZ field ffff - Rfirst field cccc - control register specifier iii..i - one of several immediate fields								ddddddddddd - branch displacement bbbb - loopt displacement uu- accelerator unit ee..e - execution code nnn - register count p - update option x..x - undefined fields								

## 3.5 Instruction Set

This section provides a detailed description of each M•CORE instruction. The descriptions are arranged in alphabetical order according to instruction mnemonic.

## ABS

## Absolute Value

## ABS

**Operation:**  $RX \leftarrow |RX|$

**Syntax:** abs rx

**Description:** Calculate the absolute value of register X, and store the result in register X.

**NOTE:** *An input operand of 0x80000000 yields a result of 0x80000000. No special indication of this is provided.*

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	1	0	Register X			

**Instruction****Fields:**

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# ADDC

## Unsigned Add with C Bit, Update C Bit

# ADDC

**Operation:**  $RX \leftarrow RX + RY + C$ ,  $C \leftarrow \text{carry out}$

**Syntax:** `addc rx,ry`

**Description:** Add the contents of register Y, the C bit, and the contents of register X. Store the result in register X.

**Condition Code:**  $C \leftarrow \text{carry out}$

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

**ADDI****Unsigned Add with Immediate****ADDI**

**Operation:**  $RX \leftarrow RX + \text{unsigned OIMM5}$

**Syntax:** `addi rx,oimm5`

**Description:** Add the immediate value to the content of register X. The immediate value must be in the range 1 to 32.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	OIMM5					Register X			

**Instruction Fields:** Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

OIMM5 Field — Specifies immediate value to be added to RX

**NOTE:** *The binary encoding is offset by one from the actual value to be added.*

00000 — add 1

00001 — add 2

...

11111 — add 32

# ADDU

## Unsigned Add

# ADDU

**Operation:**  $RX \leftarrow RX + RY$

**Syntax:** `addu rx,ry`

**Description:** Add the contents of register Y to the contents of register X and store the result in register X.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# AND

## Logical AND

# AND

**Operation:**  $RX \leftarrow RX \wedge RY$

**Syntax:** and rx,ry

**Description:** Logically AND the value of register Y with register X, and store the result in register X.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	Register Y				Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15



# ANDI

## Logical AND with Immediate

# ANDI

**Operation:**  $RX \leftarrow RX \wedge (\text{unsigned IMM5})$

**Syntax:** `andi rx,imm5`

**Description:** Logically AND the zero-extended IMM5 field with source/destination register X. The result is stored in register X.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	IMM5					Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies the unsigned 5-bit immediate value

00000 — 0

00001 — 1

...

11111 — 31

# ANDN

## Logical AND NOT

# ANDN

**Operation:**  $RX \leftarrow RX \wedge (RY!)$

**Syntax:** andn rx,ry

**Description:** Logically AND the inverted value of register Y with register X. Store the result in register X.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	Register Y				Register X			

**Instruction****Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# ASR

## Arithmetic Shift Right (Dynamic)

# ASR

**Operation:**  $RX \leftarrow \text{asr}(RX)$  by  $RY[5:0]$  bits

**Syntax:** `asr rx,ry`

**Description:**  $RX \leftarrow \text{asr}(RX)$  by  $RY[5:0]$  bits. If  $RY[5:0] > 30$ ,  $RX \leftarrow 0$  or  $-1$ .

Arithmetically shift right the value in register X by the value of  $RY[5:0]$ , and store the result in register X. If the value of register Y [5:0] is greater than 30, RX will be 0 or  $-1$  depending on the original sign of RX.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

ASRC

Arithmetic Shift Right by 1 Bit, Update C Bit

ASRC

**Operation:**  $C \leftarrow RX[0], RX \leftarrow asr(RX)$  by 1

**Syntax:** asrc rx

**Description:**  $RX \leftarrow asr(RX)$  by one bit into the C bit.

**Condition Code:**  $RX[0]$  is copied into the C bit before shifting occurs.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	0	0	0	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# ASRI

## Arithmetic Shift Right Immediate (Static)

# ASRI

**Operation:**  $RX \leftarrow \text{asr}(RX)$  by IMM5 bits

**Syntax:** `asri rx,imm5`

**Description:**  $RX \leftarrow \text{asr}(RX)$  by IMM5 bits (1–31); arithmetically shift right the value in register X by the value of the IMM5 field.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies shift value, must be in the range 1 to 31

00001 — 1

...

11111 — 31

# BCLRI

## Bit Clear Immediate

# BCLRI

**Operation:** Clear bit [IMM5] of RX

**Syntax:** bclri rx,imm5

**Description:** Clear the bit of register RX specified by the immediate field.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	IMM5					Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies bit of RX to be cleared

00000 — Bit 0

00001 — Bit 1

...

11111 — Bit 31

**BF****Conditional Branch if False****BF**

**Operation:** Conditional Branch if False:  
If (C == 0),  
     $PC \leftarrow PC + 2 + (\text{signed-extended 11-bit displacement} \ll 1)$   
else  
     $PC \leftarrow PC + 2$

**Syntax:** bf label

**Description:** If the C bit in the PSR is clear, the program counter is updated by adding its value + 2 to a scaled, sign-extended 11-bit displacement field; otherwise, the program counter is incremented by two to the next instruction. The displacement indicates the destination offset in half-words from the address of the instruction following the branch.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	Branch displacement field										

**Instruction  
Fields:** Branch Displacement Field — Specifies the branch displacement

**BGENI****Bit Generate Immediate (Static)****BGENI**

**Operation:**  $RX \leftarrow (2)^{IMM5}$

**Syntax:** `bgeni rx,imm5`

**Description:** Set the bit of register X specified by the immediate field and clear all other bits of register X.

**NOTE:** *Immediate values of zero to six are implemented using the MOVI instruction.*

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	IMM5					Register X			

**Instruction****Fields:**

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies the single bit to be set in RX

**NOTE:** *Values for the immediate field of 00000 to 00110 are not allowed, as these opcodes are used for other instructions. Assemblers will map the mnemonics for `bgeni rx,#0-6` to the MOVI instruction. Disassembly will indicate the MOVI instruction instead of the original BGENI mnemonic.*

00111 — Bit 7

01000 — Bit 8

01001 — Bit 9

...

11111 — Bit 31



# BGENR

## Bit Generate Register (Dynamic)

# BGENR

**Operation:** If  
 $R_Y[5] = 0, R_X \leftarrow 2^{R_Y[4..0]}$   
 else  
 $R_X \leftarrow 0$

**Syntax:** bgenr rx,ry

**Description:** If  $R_Y[5]$  is clear, set bit in register X specified by the five lower bits (bits 4:0) of register Y and clear all other bits of register X; otherwise clear RX.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

BKPT

Breakpoint

BKPT

**Operation:** Cause a breakpoint instruction exception to be taken

**Syntax:** bkpt

**Description:** Breakpoint

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# BMASKI

## Bit Mask Generate Immediate

# BMASKI

**Operation:**  $RX \leftarrow (2)^{IMM5-1}$

**Syntax:** bmaski rx,imm5

**Description:** Set the low-order IMM5 bits of register RX to 1 and clear the remaining upper bits. From one to 32 bits may be set. An IMM5 value of 32 is encoded in the instruction as an IMM5 field of 00000 by the assembler.

**NOTE:** Immediate values of one to seven are implemented using the MOVI instruction.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies the number of low-order bits to be set in RX. An IMM5 value of 0 is interpreted as a value of 32.

**NOTE:** Values for the immediate field of 00001 to 00111 are not allowed, as these opcodes are used for other instructions. Assemblers will map the mnemonics for BMASKI rx, #1-7 to the MOVI instruction. Disassembly will indicate the MOVI instruction instead of the original BMASKI mnemonic.

00000 — Set bits 0 to 31

00001 to 00111 — Invalid

01000 — Set bits 0 to 7

01001 — Set bits 0 to 8

...

11111 — Set bits 0 to 30

## BR

## Unconditional Branch

## BR

**Operation:** Unconditional Branch;  $PC \leftarrow PC + 2 + (\text{signed-extended 11-bit displacement} \ll 1)$

**Syntax:** br label

**Description:** The program counter is updated by adding its value + 2 to a scaled, sign-extended 11-bit displacement field. The displacement indicates the destination offset in half-words from the address of the instruction following the branch.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	Branch displacement field										

**Instruction Fields:**

Branch Displacement Field — Specifies the branch displacement

# BREV

## Bit Reverse

# BREV

**Operation:** Reverse the bits in RX

**Syntax:** brev rx

**Description:** Reverse the bits in register RX.  
If RX is initially “abcdefghijklmnopqrstuvwxyz012345”,  
it becomes “543210zyxwvutsrqponmlkjihgfedcba”

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	1	Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX  
 0000 — Register R0  
 0001 — Register R1  
 ...  
 1111 — Register R15

# BSETI

## Bit Set Immediate

# BSETI

**Operation:** Set bit [IMM5] of RX

**Syntax:** bseti rx,imm5

**Description:** Set the bit of register RX specified by the immediate field.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	IMM5					Register X			

**Instruction****Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies the bit of RX to be set

00000 — Bit 0

00001 — Bit 1

...

11111 — Bit 31

# BSR

## Branch to Subroutine

# BSR

**Operation:** Branch to Subroutine:  
 $R15 \leftarrow PC + 2$   
 $PC \leftarrow PC + 2 + (\text{signed-extended 11 bit displacement} \ll 1)$

**Syntax:** bsr label

**Description:** Return address is saved in R15. The program counter is updated by adding its value + 2 to a scaled, sign-extended 11-bit displacement field. The displacement indicates the destination offset in half-words from the address of the instruction following the branch.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	Branch displacement field										

### Instruction

#### Fields:

Branch Displacement Field — Specifies the branch displacement

**BT****Conditional Branch if True****BT**

**Operation:** Conditional Branch if True:  
     if (C == 1)  
          $PC \leftarrow PC + 2 + (\text{signed-extended 11 bit displacement} \ll 1)$   
     else  
          $PC \leftarrow PC + 2$

**Syntax:** bt label

**Description:** If the C bit in the PSR is set, the program counter is updated by adding its value + 2 to a scaled, sign-extended 11-bit displacement field, otherwise the program counter is incremented by two to the next instruction. The displacement indicates the destination offset in half-words from the address of the instruction following the branch.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	Branch displacement field										

**Instruction Fields:** Branch Displacement Field — Specifies the branch displacement



# BTSTI

## Bit Test Immediate; Update C Bit

# BTSTI

**Operation:**  $C \leftarrow RX[IMM5]$

**Syntax:** `btsti rx,imm5`

**Description:** Test the bit of register X selected by the IMM5 field, and set the C bit to the value of this bit.

**Condition Code:** Set to the value of the bit of RX pointed to by IMM5

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies which bit of RX is to be tested

00000 — Bit 0

00001 — Bit 1

...

11111 — Bit 31

# CLRF

Clear if Condition False

# CLRF

**Operation:** Conditionally clear RX to 0; if (C == 0),  $RX \leftarrow 0$ **Syntax:** clrf rx**Description:** If (C == 0),  $RX \leftarrow 0$ , move the value 0 to RX when C bit is cleared.**Condition Code:** Unaffected**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	1	Register X			

**Instruction****Fields:**

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# CLRT

## Clear if Condition True

# CLRT

**Operation:** Conditionally clear RX to 0; if (C == 1),  $RX \leftarrow 0$

**Syntax:** clrt rx

**Description:** If (C == 1),  $RX \leftarrow 0$ , move the value 0 to RX if the C bit is set

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	0	Register X			

### Instruction

#### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# CMPHS

Compare for Higher or Same

# CMPHS

**Operation:** Compare register X to register Y. If the value in register X is higher than or the same as the value in register Y,  
     C bit  $\leftarrow$  1  
 else  
     C bit  $\leftarrow$  0

**Syntax:** cmphs rx,ry

**Description:** Subtract the contents of register Y from the contents of register X. Compare the result with 0 and update the C bit appropriately. The CMPHS instruction treats the operands as unsigned. If RX is higher than or the same as RY, the C bit is set; otherwise it is cleared.

**Condition Code:** Set as a result of the comparison operation

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Register Y				Register X			

**Instruction****Fields:**

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# CMPLT

## Compare for Less Than

# CMPLT

**Operation:** Compare register X to register Y. If specified condition is true,  
C bit  $\leftarrow$  1  
else  
C bit  $\leftarrow$  0

**Syntax:** cmplt rx,ry

**Description:** Subtract the contents of register Y from the contents of register X. Compare the result with 0 and update the C bit appropriately. CMPLT treats the operands as signed two's complement integers. If RX is less than RY, the C bit is set; otherwise it is cleared.

**Condition Code:** Set as a result of the comparison operation.

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# CMPLTI

## Compare with Immediate for Less Than

# CMPLTI

**Operation:** Compare register X with immediate value. If specified condition is true,  
                   C bit  $\leftarrow$  1  
 else  
                   C bit  $\leftarrow$  0

**Syntax:** cmplti rx, oimm5

**Description:** Compare the value in register X with the immediate value. The CMPLTI instruction treats the operands as signed two's complement integers (the immediate value is always positive). If RX is less than the immediate value, the C bit is set; otherwise it is cleared. The immediate value must be in the range one to 32.

**Condition Code:** Set as a result of the comparison operation.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	OIMM5					Register X			

**Instruction  
Fields:** Register X Field — Specifies register RX  
           0000 — Register R0  
           0001 — Register R1  
           ...  
           1111 — Register R15

OIMM5 Field — Specifies immediate value to be compared to RX.

**NOTE:** *The binary encoding is offset by one from the actual immediate value to be compared.*

00000 — Compare with 1  
 00001 — Compare with 2  
 ...  
 11111 — Compare with 32

# CMPNE

## Compare for Not Equal

# CMPNE

**Operation:** Compare register X to register Y. If specified condition is true,  
C bit  $\leftarrow$  1  
else  
C bit  $\leftarrow$  0

**Syntax:** cmpne rx,ry

**Description:** Compare the contents of register Y with the contents of register X. CMPNE compares RX and RY, and sets the C bit if the values are not equal; otherwise it clears the C bit.

**Condition Code:** Set as a result of the comparison operation.

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# CMPNEI

Compare with Immediate for Not Equal

# CMPNEI

**Operation:** Compare register X with immediate value. If specified condition is true,  
                   C bit  $\leftarrow$  1  
 else  
                   C bit  $\leftarrow$  0

**Syntax:** cmpnei rx, imm5

**Description:** CMPNEI compares the value in register X with the immediate value. If RX is not equal to the immediate value, the C bit is set; otherwise it is cleared. The immediate value must be in the range zero to 31.

**Condition Code:** Set as a result of the comparison operation.

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	IMM5					Register X			

**Instruction****Fields:**

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies immediate value to be compared with RX

00000 — Compare with 0

00001 — Compare with 1

...

11111 — Compare with 31



# DECF

## Decrement Conditionally on False

# DECF

**Operation:** if C == 0, then  
 $RX \leftarrow RX - 1$   
 else  
 $RX \leftarrow RX$

**Syntax:** decf rx

**Description:** Decrement the value in register RX if the C bit is clear; otherwise the content of register X remains unchanged.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	1	Register X			

### Instruction

#### Fields:

Register X Field — Specifies the source/destination register RX  
 0000 — Register R0  
 0001 — Register R1  
 ...  
 1111 — Register R15

DECGT

Decrement, Set C Bit on Greater Than

DECGT

**Operation:**  $RX \leftarrow RX - 1$   
Update C bit

**Syntax:** decgt rx

**Description:** Decrement the value in register RX and set the C bit if result is greater than 0.

**Condition Code:** C bit is set to 1 if the result in RX is greater than 0; otherwise it is cleared.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	1	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# DECLT

Decrement, Set C Bit on Less Than

# DECLT

**Operation:**  $RX \leftarrow RX - 1$   
Update C bit

**Syntax:** declt rx

**Description:** Decrement the value in register RX and set the C bit if the result is less than 0.

**Condition Code:** C bit is set to 1 if the result in RX is less than 0; otherwise the C bit is cleared.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# DECNE

Decrement, Set C Bit on Not Equal

# DECNE

**Operation:**  $RX \leftarrow RX - 1$ , update C bit**Syntax:** decne rx**Description:** Decrement the value in register RX and set the C bit if the result is not equal to 0.**Condition Code:** C bit is set to 1 if the result in RX is not equal to 0; otherwise the C bit is cleared.**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	1	1	Register X			

**Instruction****Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# DECT

## Decrement Conditionally on True

# DECT

**Operation:** If C == 1 then,  
 $RX \leftarrow RX - 1$   
 else  
 $RX \leftarrow RX$

**Syntax:** dect rx

**Description:** If the C bit is set, decrement the value in register RX; otherwise the content of register X remains unchanged.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX  
 0000 — Register R0  
 0001 — Register R1  
 ...  
 1111 — Register R15

# DIVS

## Signed Divide RX by R1

# DIVS

**Operation:**  $RX \leftarrow \frac{RX}{R1}$

**Syntax:** `divs rx,r1`

**Description:** Divide the contents of register X by the contents of register R1 and store the result in register X. The values in RX and R1 are treated as 32-bit signed integers. For the case of 0x8000 0000 divided by 0xFFFF FFFF, the result is undefined. If the value in R1 is 0, no result is written, and a divide-by-zero exception is generated (vector offset 0x00C).

**Condition Code:** Undefined

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	0	0	0	1	Register X			

**Instruction**

**Fields:**

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# DIVU

## Unsigned Divide RX by R1

# DIVU

**Operation:**  $RX \leftarrow \frac{RX}{R1}$

**Syntax:** `divu rx,r1`

**Description:** Divide the contents of register X by the contents of register R1 and store the result in register X. The values in RX and R1 are treated as 32-bit unsigned integers. For the case of 0x8000 0000 divided by 0xFFFF FFFF, the result is undefined. If the value in R1 is 0, no result is written, and a divide-by-zero exception is generated (vector offset 0x00C).

**Condition Code:** Undefined

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	0	0	0	1	Register X			

**Instruction  
Fields:** Register X Field — Specifies register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

DOZE

Enter Low-Power Doze Mode

DOZE

**Operation:** Enter low-power doze mode

**Syntax:** doze

**Attributes:** Privileged

**Description:** Place the processor in low-power doze mode and wait for an interrupt to exit. The CPU clock is stopped. Which peripherals are stopped is implementation dependent. Refer to the appropriate microcontroller user's manual for details on how this instruction is implemented and how it affects peripherals in a particular implementation.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0



# FF1

## Find First One in RX

# FF1

**Operation:**  $RX \leftarrow \text{ff1}(RX)$

**Syntax:** ff1 rx

**Description:** Find the first set bit in register RX, and return the result into RX. RX is scanned from the most significant bit (MSB) to the least significant bit (LSB), searching for a set bit. The value returned is the offset from the MSB of RX. If bit 31 of RX is set, the value returned is 0. If no bits are set in RX, a value of 32 is returned.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	0	Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# INCF

Increment RX Conditionally on False

# INCF

**Operation:** If C == 0, then  
RX ← RX + 1  
else  
RX ← RX

**Syntax:** incf rx

**Description:** If the C bit is clear, increment the value in register RX; otherwise the content of register X remains unchanged.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	1	1	Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# INCT

## Increment RX Conditionally on True

# INCT

**Operation:** If C == 1, then  
                   $RX \leftarrow RX + 1$   
                  else  
                   $RX \leftarrow RX$

**Syntax:** inct rx

**Description:** If the C bit is set, increment the value in register RX; otherwise the content of register X remains unchanged.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	1	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
          0000 — Register R0  
          0001 — Register R1  
          ...  
          1111 — Register R15

# IXH

## Index Half-Word

# IXH

**Operation:**  $RX \leftarrow RX + [RY \ll 1]$

**Syntax:** `ixh rx,ry`

**Description:** Add the value in register RX to the value in register RY left shifted by one, and store the result in register RX.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	Register Y				Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# IXW

## Index Word

# IXW

**Operation:**  $RX \leftarrow RX + [RY \ll 2]$

**Syntax:** ixw rx,ry

**Description:** Add the value in register RX to the value in register RY shifted left by two, and store the result in register RX.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# JMP

## Unconditional Jump

# JMP

**Operation:** Unconditional jump:  
 $PC \leftarrow (RX)$

**Syntax:** jmp rx

**Description:** Unconditionally jump to the location specified by the content of register RX. The low-order bit of RX is ignored and replaced with a 0.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	0	Register X			

**Instruction****Fields:**

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# JMPI

## Unconditional Jump Indirect

# JMPI

**Operation:** Unconditional jump indirect:  
$$PC \leftarrow \text{MEM}[(PC + 2 + (\text{unsigned disp\_8} \ll 2)) \& 0\text{ffffffc}]$$

**Syntax:** jmp [label]

**Description:** The 8-bit displacement field is zero extended, scaled by two, and added to PC + 2. The low-order two bits of this address are forced to 0, and a word is loaded from this address into the PC. In essence, the destination address is stored in a memory location relative to the current PC (at location label). This word is fetched and loaded into the PC, and instruction execution resumes at the new PC value. Note that only forward offsets from the PC are available for referencing the jump target address. If the value fetched for the destination address is odd, a misaligned access exception is taken.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	DISP_8							

**Instruction Fields:** DISP\_8 Field — Unsigned 8-bit displacement

## JSR

## Unconditional Jump to Subroutine

## JSR

**Operation:** Unconditional jump to subroutine:  
 $R15 \leftarrow PC + 2,$   
 $PC \leftarrow (RX)$

**Syntax:** jsr rx

**Description:** Unconditionally jump to the subroutine location specified by the content of RX, and save the return address in R15. The low-order bit of RX is ignored and replaced with a 0.

**CAUTION:** *RX must not specify R15; this condition is undefined and undetected.*

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	1	Register X			

**Instruction****Fields:**

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1110 — Register R14

1111 — Illegal specifier; do not use



# JSRI

## Unconditional Jump Subroutine Indirect

# JSRI

**Operation:** Unconditional jump subroutine indirect:  
 $R15 \leftarrow PC + 2$ ,  
 $PC \leftarrow MEM[(PC + 2 + (\text{unsigned } disp\_8 \ll 2)) \& 0xffffffc]$

**Syntax:** jsri [label]

**Description:** (PC + 2) is saved in R15, and then the 8-bit displacement field is zero extended, scaled by two, and added to PC + 2. The low-order two bits of the address are forced to 0, and a word is loaded from this address into the PC. In essence, the destination address is stored in a memory location relative to the current PC (at location label). This word is fetched and loaded into the PC, and instruction execution resumes at the new PC value. Note that only forward offsets from the PC are available for referencing the jump target address. If the value fetched for the destination address is odd, a misaligned access exception is taken.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	DISP_8							

### Instruction

#### Fields:

DISP\_8 Field — Unsigned 8-bit displacement

LD.[BHW]

Load Register from Memory

LD.[BHW]

**Operation:** Source/destination register ←Memory location:  
RZ ←MEM[RX + unsigned IMM4 <<{0,1,2}]

**Syntax:** ld.[bhw] rz,(rx,disp)  
[ld, ldb, ldh, ldw] rz,(rx,disp)

**Description:** The load operation has three options: w (word), h (half-word), and b (byte). Disp is obtained by taking the IMM4 field, scaling by the size of the load, and zero-extending. This value is added to the value of register RX, and a load of the specified size is performed from this address, with the result of the load stored in register RZ. For byte and half-word loads, the data fetched is zero-extended before being placed in RZ.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Size		0	register Z				IMM4				Register X			

**Instruction  
Fields:**

Size — Specifies load size  
00 — Word  
01 — Byte  
10 — Half-word

Register Z — Specifies the destination register for the load

IMM4 Field — Specifies a 4-bit scaled immediate value

Register X — Specifies the base address to be added to the scaled immediate field

# LDM

## Load Multiple Registers from Memory

# LDM

**Operation:** Destination Registers  $\leftarrow$  Memory

**Syntax:** `ldm rf-r15,(r0)`

**Description:** The LDM instruction is used to load a contiguous range of registers from the stack. Register 0 (R0) serves as the base address pointer for this form. Registers Rf–R15 are loaded in increasing significance from ascending memory locations. Rf must not specify R0 or R15; these instruction forms are considered illegal, although they are not guaranteed to be detected by hardware. For valid instruction forms, register 0 (R0) is not affected or updated.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	0	register first			

**Instruction  
Fields:** Register First Field — Specifies the first register to be transferred  
Only R1–R14 should be specified.

# LDQ

## Load Register Quadrant from Memory

# LDQ

**Operation:** Destination registers  $\leftarrow$  memory;

**Syntax:** ldq r4–r7,(rx)

**Description:** The LDQ instruction is used to load four registers (R4–R7) from memory. Register X points to the location of the first transfer. Registers are loaded in increasing significance from ascending memory locations. If register X is specified to be R4, R5, R6, or R7, the instruction form is considered invalid, and the results are undefined. For valid instruction forms, register X is not affected or updated.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	Register X			

**Instruction Fields:** Register X — Specifies the base address for the transfers  
Register X should not specify R4, R5, R6, or R7.

# LOOPT

**Branch on True, Decrementing Count,  
Set C Bit on Greater Than**

# LOOPT

**Operation:** If C == 1,  
then  
     $PC \leftarrow PC + 2 + (\text{one's extended 4-bit displacement} \ll 1)$   
     $RY \leftarrow RY - 1$   
    Update C bit  
else  
     $RY \leftarrow RY - 1$   
    Update C bit

**Syntax:** loopt ry,label

**Description:** Decrement the value in register RY setting the C bit if the (signed) result is greater than 0 (clear the C bit otherwise). Branch to label if the C bit was set prior to the decrement.

**Condition Code:** C bit is set to 1 if the (signed) result in RY is greater than 0, and cleared otherwise.

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	Register Y				Disp_4			

## Instruction

### Fields:

Register Y Field — Specifies source/destination register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Disp\_4 Field — Loop displacement from PC + 2

0000 — Displacement of -32

0001 — Displacement of -30

...

1111 — Displacement of -2

## LRW

## Load PC-Relative Word

## LRW

**Operation:**  $RZ \leftarrow \text{MEM}(\text{PC} + 2 + (\text{unsigned imm\_8} \ll 2)) \& 0\text{ffffffc}$

**Syntax:** `lrw rz,[label]`  
 Assembler sets the 8-bit displacement to point to label.

`lrw rz,label`  
 Assembler allocates a literal table entry containing address of label, and sets the 8-bit displacement to point to table entry.

`lrw rz,0x 32-bit value expressed in hexadecimal notation`

`lrw rz,0b 32-bit value expressed in binary notation`

`lrw rz, 32-bit value expressed in decimal notation`

Assembler allocates a literal table entry containing the specified 32-bit value (expressed in hexadecimal, binary, or decimal form) and sets the 8-bit displacement to point to table entry.

**Description:** The DISP\_8 field is zero extended, scaled by four (for example, left-shifted by two), and added to PC + 2. The low-order two bits of this address are forced to 0, and a word is loaded from this address into register RZ. RZ may not be R0 (the stack pointer by convention) or R15 (the link register). The 8-bit displacement field is an unsigned value — forward displacements from PC + 2 in the range 0x4 to 0x3FC are available for referencing load data.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	Register Z				DISP_8							

**Instruction****Fields:**

Register Z — Specifies the destination register for the load  
 May not be R0 (0000) or R15 (1111)

DISP\_8 Field — Specifies an unsigned 8-bit displacement value

# LSL

## Logical Shift Left (Dynamic)

# LSL

**Operation:**  $RX \leftarrow \text{isl}(RX)$  by  $RY[5:0]$  bits

**Syntax:** `isl rx,ry`

**Description:**  $RX \leftarrow \text{isl}(RX)$  by  $RY[5:0]$  bits. If  $RY[5:0] > 31$ ,  $RX \leftarrow 0$ .  
Perform a logical shift left of the value in register X by the value of  $RY[5:0]$ . If the value of register Y[5:0] is greater than 31, RX will be 0.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	Register Y				Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

LSLC

Logical Shift Left by 1, Update C Bit

LSLC

**Operation:**  $C \leftarrow RX[31], RX \leftarrow \text{sl}(RX) \text{ by } 1$

**Syntax:** `lslc rx`

**Description:**  $RX \leftarrow \text{sl}(RX)$  by one bit into the C bit

**Condition Code:** Copy  $RX[31]$  into the C bit before shifting occurs.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	0	0	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15



# LSLI

## Logical Shift Left Immediate (Static)

# LSLI

**Operation:**  $RX \leftarrow \text{LSL}(RX)$  by IMM5 bits (1...31)

**Syntax:** `lsli rx,imm5`

**Description:**  $RX \leftarrow \text{LSL}(RX)$  by IMM5 bits (1...31); logically shift left the value in register X by the value of the IMM5 field.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies shift value. Must be in the range one to 31

00001 — 1

...

11111 — 31

LSR

Logical Shift Right (Dynamic)

LSR

**Operation:**  $RX \leftarrow \text{lsr}(RX)$  by  $RY[5:0]$  bits

**Syntax:** `lsr rx,ry`

**Description:**  $RX \leftarrow \text{lsr}(RX)$  by  $RY[5:0]$  bits. If  $RY[5:0] > 31$ ,  $RX \leftarrow 0$ .  
Perform logical shift right of the value in register X by the value of  $RY[5:0]$ . If the value of register Y[5:0] is greater than 31, RX will be 0.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	Register Y				Register X			

**Instruction**

Register X Field — Specifies source/destination register RX

**Fields:**

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

LSRC

Logical Shift Right by 1 Bit, Update C Bit

LSRC

**Operation:**  $C \leftarrow RX[0], RX \leftarrow \text{lsr}(RX) \text{ by } 1$

**Syntax:** `lsr rX`

**Description:**  $RX \leftarrow \text{lsr}(RX)$  by one bit into the C bit.

**Condition Code:** Copy  $RX[0]$  into the C bit before shifting occurs.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	0	0	0	0	0	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register  $RX$   
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

## LSRI

## Logical Shift Right Immediate (Static)

## LSRI

**Operation:**  $RX \leftarrow \text{lsr}(RX)$  by IMM5 bits (1...31).

**Syntax:** lsri rx,imm5

**Description:**  $RX \leftarrow \text{lsr}(RX)$  by IMM5 bits. Perform logical shift right of the value in register X by the value of the IMM5 field.

**Condition Code:** Unaffected.

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	IMM5					Register X			

**Instruction****Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies shift value, must be in the range one to 31

00001 — 1

...

11111 — 31

# MFCR

## Move from Control Register

# MFCR

**Operation:** Move from control register:  $RX \leftarrow CR_Y$

**Syntax:** mfc r<sub>x</sub>,c<sub>r</sub><sub>y</sub>

**Attributes:** Privileged

**Description:** Move the contents of control register Y to register X.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	CRegister Y					Register X			

### Instruction

#### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

CRegister Y Field — Specifies source control register CR<sub>y</sub>

00000 — Control register CR0

00001 — Control register CR1

...

11111 — Control register CR31

# MOV

## Logical Move

# MOV

**Operation:**  $RX \leftarrow RY$

**Syntax:** `mov rx,ry`

**Description:** Copy the value of register Y to destination register X.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	Register Y				Register X			

**Instruction**

**Fields:**

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# MOVF

Move RY to RX if Condition False

# MOVF

**Operation:** Conditionally move RY to RX; if (C==0),  $RX \leftarrow RY$

**Syntax:** movf rx,ry

**Description:** If (C == 0)  $RX \leftarrow RY$ ; conditionally move RY to RX when C bit is cleared.

**Condition Code:** Unaffected

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	Register Y				Register X			

## Instruction

### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

## MOVI

## Logical Move Immediate

## MOVI

**Operation:**  $RX \leftarrow \text{unsigned IMM7}$

**Syntax:** `movi rx,imm7`

**Description:** Move the zero-extended 7-bit immediate value to destination register X.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	IMM7							Register X			

**Instruction****Fields:**

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM7 Field — Specifies immediate value to be moved to RX

0000000 — 0

0000001 — 1

...

1111111 — 127



# MOVT

Move RY to RX if Condition True

# MOVT

**Operation:** Conditionally move RY to RX; if (C == 1),  $RX \leftarrow RY$

**Syntax:** movt rx,ry

**Description:** If (C == 1),  $RX \leftarrow RY$ ; conditionally move RY to RX when C bit is set.

**Condition Code:** Unaffected

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	Register Y				Register X			

## Instruction

### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

## MTCR

## Move to Control Register

## MTCR

**Operation:** Move to control register:  $CR_Y \leftarrow RX$

**Syntax:** mtcr rx,cry

**Attributes:** Privileged

**Description:** Move the contents of register X to the control register specified by CRegister Y.

**Condition Code:** Unaffected unless CR0 (PSR) specified

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	CRegister Y					Register X			

**Instruction****Fields:**

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

CRegister Y Field — Specifies destination control register CRy

00000 — Control register CR0

00001 — Control register CR1

...

11111 — Control register CR31

# MULT

## Multiply

# MULT

**Operation:**  $RX \leftarrow RX \times RY$

**Syntax:** `mult rx,ry`  
`mul rx,ry`

**Description:** Multiply the contents of register X with the contents of register Y and store the low order 32 bits of the result in register X. 32x32 →32 (low-order product). The result produced is the same regardless of whether the source operands are considered signed or unsigned.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

## MVC

## Move C Bit to Register

## MVC

**Operation:**  $RX \leftarrow C \text{ bit}$

**Syntax:** `mvc rx`

**Description:** Copy the value of the C bit to the low-order bit of destination register X, and clear all other bits of RX.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	0	Register X			

**Instruction Fields:** Register X Field — Specifies destination register RX

- 0000 — Register R0
- 0001 — Register R1
- ...
- 1111 — Register R15

# MVCV

## Move Inverted C Bit to Register

# MVCV

**Operation:**  $RX \leftarrow (C \text{ bit})!$

**Syntax:** mvcv rx

**Description:** Copy the inverted value of the C bit to the low-order bit of destination register X, clear all other bits of RX.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	Register X			

### Instruction

#### Fields:

Register X Field — Specifies destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# NOT

## Logical NOT

# NOT

**Operation:**  $RX \leftarrow (RX)!$

**Syntax:** not rx

**Description:** Logically invert the value of register X.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	1	1	Register X			

**Instruction  
Fields:** Register X Field — Specifies register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# OR

## Logical OR

# OR

**Operation:**  $RX \leftarrow RX \vee RY$

**Syntax:** or rx,ry

**Description:** Logically OR the value of register Y with register X and store the result in register X.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	Register Y				Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# RFI

## Return from Fast Interrupt

# RFI

**Operation:** PC  $\leftarrow$  FPC (CR5);  
PSR  $\leftarrow$  FPSR (CR3)

**Syntax:** rfi

**Attributes:** Privileged

**Description:** The program counter (PC) is loaded with the value saved in control register CR5, the PSR is loaded from the value in CR3, and instruction execution begins with the instruction at the new PC value.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1



# ROTLI

## Rotate Left Immediate (Static)

# ROTLI

**Operation:**  $RX \leftarrow \text{rotl}(RX)$  by IMM5 bits (1..31)

**Syntax:** `rotli rx,imm5`

**Description:**  $RX \leftarrow \text{rotl}(RX)$  by IMM5 bits (1...31). Rotate the value in register X left by the value of the IMM5 field.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies rotate value, must be in the range one to 31

00001 — 1

...

11111 — 31

# RSUB

## Reverse Subtract

# RSUB

**Operation:**  $RX \leftarrow RY - RX$

**Syntax:** `rsub rx,ry`

**Description:** Subtract the contents of register X from the contents of register Y and store the result in register X.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	Register Y				Register X			

**Instruction Fields:**

Register X Field — Specifies source/destination register RX

- 0000 — Register R0
- 0001 — Register R1
- ...
- 1111 — Register R15

Register Y Field — Specifies source register RY

- 0000 — Register R0
- 0001 — Register R1
- ...
- 1111 — Register R15

# RSUBI

## Reverse Subtract with Immediate

# RSUBI

**Operation:**  $RX \leftarrow [\text{unsigned IMM5}] - RX$

**Syntax:** `rsubi rx,imm5`

**Description:** Subtract the contents of register X from the unsigned value specified by the IMM5 field, and store the result in register X.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	IMM5					Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

IMM5 Field — Specifies immediate value to be used

00000 — 0

00001 — 1

...

11111 — 31

RTE

Return from Exception

RTE

**Operation:** PC  $\leftarrow$  EPC (CR4); PSR  $\leftarrow$  EPSR (CR2)

**Syntax:** rte

**Attributes:** Privileged

**Description:** The program counter (PC) is loaded with the value saved in control register CR4, the PSR is loaded from the value in CR2, and instruction execution begins with the instruction at the new PC value.

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

# SXTB

## Sign Extend Byte

# SXTB

**Operation:**  $RX \leftarrow RX[7:0]$  sign-extended to 32 bits

**Syntax:** sxtb rx

**Description:** Sign extend the low-order byte of register RX to 32 bits.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	1	Register X			

**Instruction  
Fields:** Register X Field — Specifies source/destination register RX

- 0000 — Register R0
- 0001 — Register R1
- ...
- 1111 — Register R15

SEXTH

Sign Extend Half-Word

SEXTH

**Operation:**     $RX \leftarrow RX[15:0]$  sign-extended to 32 bits

**Syntax:**        sixth rx

**Description:**    Sign extend the low-order half of register RX to 32 bits.

**Condition Code:**    Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	1	1	Register X			

**Instruction  
Fields:**        Register X Field — Specifies source/destination register RX  
                  0000 — Register R0  
                  0001 — Register R1  
                  ...  
                  1111 — Register R15

**ST.[B,H,W]****Store Register to Memory****ST.[B,H,W]**

**Operation:** Memory  $\leftarrow$  Source Register:  
 $\text{MEM}[\text{RX} + \text{unsigned IMM4} \ll \{0,1,2\}] \leftarrow \text{RZ}$

**Syntax:** st.[b,h,w] rz,(rx,disp)  
 [st, stw sth stb] rz,(rx,disp)

**Description:** Store register contents to memory. The store operation has three options: w (word), h (half-word), and b (byte). Disp is obtained by taking the IMM4 field, scaling by the size of the store, and zero-extending. This value is added to the value of register RX, and a store of the specified size is performed to this address.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Size		1	register Z				IMM4				Register X			

**Instruction  
Fields:**

Size — Specifies store size

00 — Word

01 — Byte

10 — Half-word

Register Z — Specifies the source register for store data

IMM4 Field — Specifies a 4-bit scaled immediate value

Register X — Specifies the base address to be added to the scaled immediate field

# STM

## Store Multiple Registers to Memory

# STM

**Operation:** Memory  $\leftarrow$  Source Registers

**Syntax:** stm rf–r15,(r0)

**Description:** Store multiple registers to memory. The STM instruction is used to transfer a contiguous range of registers to the stack. Register 0 (R0) serves as the base address pointer for this form. Registers Rf –R15 are stored in increasing significance to ascending memory locations. Register 0 (R0) is not affected/updated. Rf may not specify R0 or R15; these instruction forms are considered illegal, although they are not guaranteed to be detected by hardware.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	register first			

**Instruction  
Fields:** Register First Field — Specifies the first register to be transferred. Only R1–R14 should be specified.



# STOP

## Enter Low-Power Stop Mode

# STOP

**Operation:** Enter stop mode

**Syntax:** stop

**Attributes:** Privileged

**Description:** Place the processor in low-power stop mode and wait for an interrupt to exit stop mode. The CPU clock is stopped, and most peripherals cease operation. Refer to the appropriate microcontroller user's manual for details on how this instruction is implemented and how it affects peripherals in a particular implementation.

**Condition Code:** Unaffected

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

STQ

Store Register Quadrant to Memory

STQ

**Operation:** Memory ←Source Registers

**Syntax:** stq r4–r7,(rx)

**Description:** Store register quadrant to memory. The STQ instruction is used to transfer the contents of four registers (R4–R7) to memory. Register X points to the location of the first transfer. Registers are stored in increasing significance to ascending memory locations. Register X is not affected or updated. If register X is part of the quadrant being transferred, the value stored for this register is undefined.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	1	Register X			

**Instruction  
Fields:** Register X — Specifies the base address for the transfers.  
Register X should not specify R4, R5, R6, or R7.

# SUBC

Unsigned Subtract with C Bit; Update C Bit

# SUBC

**Operation:**  $RX \leftarrow RX - RY - (C!)$   
 $C \leftarrow \text{carryout}$

**Syntax:** subc rx,ry

**Description:** Subtract the contents of register Y and the inverted value of the C bit from the contents of register X and store the result in register X. The C bit is updated with the carryout from the subtract. For subtract, this is the inverse of a borrow.

**Condition Code:**  $C \leftarrow \text{carryout}$

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	Register Y				Register X			

## Instruction

### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# SUBI

## Unsigned Subtract with Immediate

# SUBI

**Operation:**  $RX \leftarrow RX - [\text{unsigned OIMM5}]$

**Syntax:** `subi rx,oimm5`

**Description:** Subtract the immediate value from the contents of register X. The immediate value must be in the range of one to 32.

**Condition Code:** Unaffected

**Instruction****Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	OIMM5					Register X			

**Instruction****Fields:**

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

OIMM5 Field — Specifies immediate value to be subtracted from RX.

**NOTE:** *The encoding is offset by one from the actual value to be subtracted.*

00000 — Subtract 1

00001 — Subtract 2

...

11111 — subtract 32

# SUBU

## Unsigned Subtract

# SUBU

**Operation:**  $RX \leftarrow RX - RY$

**Syntax:** subu rx,ry  
sub rx,ry

**Description:** Subtract the contents of register Y from the contents of register X and store the result in register X.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

# SYNC

Synchronize CPU

# SYNC

**Operation:** Cause the CPU to synchronize

**Syntax:** sync

**Description:** When the processor encounters a SYNC instruction, instruction issue is suspended until all outstanding operations are complete, and no pending operations remain.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

# TRAP

## Unconditional Trap to OS

# TRAP

**Operation:** Cause a trap exception to occur

**Syntax:** trap #trap\_number

**Description:** When the processor encounters a TRAP instruction, trap exception processing is initiated.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	vec2	

### Instruction

#### Fields:

Vec2 — 2-bit immediate field to describe trap number

TST

Test with Zero

TST

**Operation:** If (RX & RY)  $\neq$  0, then  
C bit  $\leftarrow$  1  
else  
C bit  $\leftarrow$  0

**Syntax:** tst rx,ry

**Description:** Test the ANDed contents of register X and Y. If the result is non-zero, set the C bit; otherwise clear the C bit.

**Condition Code:** Set if (RX & RY)  $\neq$  0; cleared otherwise.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	Register Y				Register X			

**Instruction  
Fields:**

Register X Field — Specifies register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

Register Y Field — Specifies register RY  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15



# TSTNBZ

Test Register for No Byte Equal to Zero

# TSTNBZ

**Operation:** If no byte of register X is 0, then  
C bit  $\leftarrow$  1  
else  
C bit  $\leftarrow$  0

**Syntax:** tstnbz rx

**Description:** Test whether no byte of register X is equal to 0. If true (no byte equals 0), set the C bit; otherwise clear the C bit.

**Condition Code:** Set to the result of the test operation.

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	1	Register X			

## Instruction

### Fields:

Register X Field — Specifies register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

WAIT

Stop Execution and Wait for Interrupt

WAIT

**Operation:** Enter low-power wait mode

**Syntax:** wait

**Attributes:** Privileged

**Description:** Stop execution and wait for an interrupt. The CPU clock is stopped. Typically, all peripherals continue to run and may generate interrupts, causing the CPU to exit from the wait state. Refer to the appropriate microcontroller user’s manual for details on how this instruction is implemented and how it affects peripherals in a particular implementation.

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

# XOR

## Logical Exclusive OR

# XOR

**Operation:**  $RX \leftarrow RX \otimes RY$

**Syntax:** xor rx,ry

**Description:** Perform logical exclusive OR of register Y with register X; store the result in register X.

**Condition Code:** Unaffected

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	Register Y				Register X			

### Instruction

#### Fields:

Register X Field — Specifies source/destination register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

Register Y Field — Specifies source register RY

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

XSR

Extended Shift Right

XSR

**Operation:** Extended shift right RX by one bit

**Syntax:** xsr rx

**Description:** Shift RX right by one bit through the C bit, for example:  
 $C_{tmp} \leftarrow C$      $C \leftarrow RX[0]$      $lsh(RX, 1)$      $RX[31] \leftarrow C_{tmp}$

**Condition Code:** Set to the original value of RX[0]

**Instruction**

**Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	0	Register X			

**Instruction**

**Fields:**

Register X Field — Specifies source/destination register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# XTRB0

Extract High-Order Byte into R1 and Zero-Extend

# XTRB0

**Operation:**  $R1 \leftarrow \text{byte 0 of RX (bits 31:24) zero extended to 32 bits}$

**Syntax:** `xtrb0 r1,rx`

**Description:** Extract high order byte of RX into R1 and zero-extend;  
 $R1[7:0] \leftarrow \text{RX}[31:24]$ ,  $R1[31:8] \leftarrow 0$ ,  $C \leftarrow (\text{result is } \neq 0?)$

**Condition Code:** The C bit is set to 1 if the result is  $\neq 0$  and cleared otherwise.

## Instruction

### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	1	1	Register X			

## Instruction

### Fields:

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

XTRB1

Extract Byte 1 into R1 and Zero-Extend

XTRB1

**Operation:** R1←byte 1 of RX (bits 23:16) zero extended to 32 bits

**Syntax:** xtrb1 r1,rx

**Description:** Extract bits 23:16 of RX into R1 and zero-extend;  
R1[7:0] ←RX[23:16], R1[31:8] ←0, C ←(result is ≠0?)

**Condition Code:** The C bit is set to 1 if the result is ≠ 0 and cleared otherwise.

Instruction

Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	1	0	Register X			

Instruction

Fields:

Register X Field — Specifies source register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15

# XTRB2

## Extract Byte 2 into R1 and Zero-Extend

# XTRB2

**Operation:**  $R1 \leftarrow \text{byte 2 of RX zero-extended to 32 bits}$

**Syntax:** `xtrb2 r1,rx`

**Description:** Extract bits 15:8 of RX into R1 and zero-extend;  
 $R1[7:0] \leftarrow RX[15:8]$ ,  $R1[31:8] \leftarrow 0$ ,  $C \leftarrow (\text{result is } \neq 0?)$

**Condition Code:** The C bit is set to 1 if the result is  $\neq 0$  and cleared otherwise.

### Instruction

#### Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	1	Register X			

### Instruction

#### Fields:

Register X Field — Specifies source register RX

0000 — Register R0

0001 — Register R1

...

1111 — Register R15

XTRB3

Extract Low-order Byte into R1 and Zero-Extend

XTRB3

**Operation:** R1←byte 3 of RX zero extended to 32 bits

**Syntax:** xtrb3 r1,rx

**Description:** Extract low-order byte into R1 and zero-extend;  
R1[7:0] ←RX[7:0], R1[31:8] ←0, C ←(result is ≠0?)

**Condition Code:** The C bit is set to 1 if the result is ≠ 0 and cleared otherwise.

Instruction

Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	Register X			

Instruction

Fields:

Register X Field — Specifies source register RX  
0000 — Register R0  
0001 — Register R1  
...  
1111 — Register R15



# ZEXTB

## Zero Extend Byte

# ZEXTB

**Operation:**  $RX \leftarrow \text{low-order byte of } RX \text{ zero-extended to 32 bits}$

**Syntax:** `zextb rx`

**Description:** Zero extend the low-order byte of register `RX` to 32 bits.

**Condition Code:** Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	0	Register X			

**Instruction** Register X Field — Specifies source/destination register `RX`.

**Fields:**  
 0000 — Register `R0`  
 0001 — Register `R1`  
 ...  
 1111 — Register `R15`

ZEXTH

Zero Extend Half-Word

ZEXTH

**Operation:**     $RX \leftarrow \text{low-order half of } RX \text{ zero-extended to 32 bits}$

**Syntax:**        `zexth rx`

**Description:**    Zero extend the low-order half of register `RX` to 32 bits.

**Condition Code:**    Unaffected

**Instruction  
Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	1	0	Register X			

**Instruction  
Fields:**        Register X Field — Specifies source/destination register `RX`  
                    0000 — Register `R0`  
                    0001 — Register `R1`  
                    ...  
                    1111 — Register `R15`

## Section 4. Exception Processing

### 4.1 Contents

4.2	Introduction . . . . .	164
4.3	Exception Processing Overview . . . . .	164
4.4	Stages of Exception Processing . . . . .	165
4.5	Exception Vectors . . . . .	167
4.6	Exception Types . . . . .	168
4.6.1	Reset Exception (Vector Offset 0x0) . . . . .	169
4.6.2	Misaligned Access Exception (Vector Offset 0x4) . . . . .	169
4.6.3	Access Error Exception (Vector Offset 0x8) . . . . .	170
4.6.4	Divide-by-Zero Exception (Vector Offset 0x0C) . . . . .	170
4.6.5	Illegal Instruction Exception (Vector Offset 0x10) . . . . .	170
4.6.6	Privilege Violation Exception (Vector Offset 0x14) . . . . .	171
4.6.7	Trace Exception (Vector Offset 0x18) . . . . .	171
4.6.8	Breakpoint Exception (Vector Offset 0x1C) . . . . .	173
4.6.9	Unrecoverable Error Exception (Vector Offset 0x20) . . . . .	173
4.6.10	Soft Reset Exception (Vector Offset 0x24) . . . . .	173
4.6.11	Interrupt Exceptions . . . . .	174
4.6.11.1	Normal Interrupt ( $\overline{\text{INT}}$ ) . . . . .	175
4.6.11.2	Fast Interrupt ( $\overline{\text{FINT}}$ ) . . . . .	175
4.6.12	Hardware Accelerator Exception (Vector Offset 0x30) . . . . .	176
4.6.13	Instruction Trap Exception (Vector Offset 0x40-0x5C) . . . . .	176
4.7	Exception Priorities . . . . .	176
4.8	Returning from Exception Handlers . . . . .	178

### 4.2 Introduction

Exception processing is performed by the processor hardware in preparing to execute a software routine for any condition that causes an exception. This section describes exception processing, exception priorities, returning from an exception, and bus fault recovery. This section also describes the exception vector table.

### 4.3 Exception Processing Overview

Exception processing is the transition from the normal processing of a program to the processing required for any special internal or external condition that pre-empts normal processing. External conditions that cause exceptions are:

- Interrupts from external devices
- Hardware breakpoint requests
- Access errors
- Resets

Internal conditions that cause exceptions are:

- Instructions
- Misalignment errors
- Privilege violations
- Tracing

The TRAP and BKPT instructions generate exceptions as part of their normal execution. In addition, the following cause exceptions:

- Illegal instructions
- Misaligned addresses for LD, LDM, LDQ, ST, STM, and STQ instructions
- Odd valued destination addresses for the JMPL and JSRL instructions
- Privilege violations

Exception processing uses an exception vector table and a set of internal shadow registers to make the transition to an exception handler.

The M•CORE uses an instruction restart exception processing model. Exceptions are recognized at the decode or execution stage of the instruction pipeline and force later instructions that have not yet reached that stage to be aborted. For exceptions detected at the instruction decode stage (unimplemented instructions, traps, privilege violations) and instruction exceptions related to the execute stage (misaligned accesses, access errors), the program counter value saved for an exception points to the instruction that caused the exception. For interrupts and trace exceptions, the program counter points to the next instruction to be executed. For exceptions related to the hardware accelerator interface (HAI) refer to the appropriate microcontroller user's manual.

Any LD and ST instructions that have reached the execute stage of the pipeline are allowed to complete before exception processing begins, unless an access error prevents the instruction from completing. With an interrupt pending, the saved program counter points to the LD or ST instruction if an access error occurred, or points to the next instruction if no access exception occurred. This prevents LD or ST instructions that have completed successfully from being re-executed on returning from the interrupt.

## 4.4 Stages of Exception Processing

Exception processing occurs in these steps:

1. During this step, the processor saves a copy of the status register (PSR) and program counter (PC) in the appropriate set of shadow registers. For the fast interrupt exception they are saved in the FPSR and FPC control registers. For all other exceptions, they are placed in the EPSR and EPC shadow registers. Exceptions (other than fast interrupt exceptions) occurring while PSR(EE) is clear result in an unrecoverable error exception, regardless of their type. After the PSR and PC are saved in the appropriate shadow registers, the PSR(EE) bit is cleared to arm the unrecoverable error exception logic.

2. On an unrecoverable error exception, the EPSR and ECP are still updated. If the unrecoverable error is due to an exception on an exception vector table fetch as part of exception processing (described below), the values saved in the EPSR and EPC are undefined.
3. Next, the processor changes to the supervisor mode by setting the PSR(S) bit and inhibits tracing of the exception handler by clearing the TM field in the PSR. The interrupt enable flag (IE) is also cleared to inhibit normal interrupt recognition. Fast interrupt exceptions and resets clear the fast interrupt enable (FE) bit; it is unaffected by other exceptions.
4. The translation control (TC) bit in the PSR is cleared to disable address translation by an optional external memory management unit for the remainder of exception processing, allowing the following accesses to be performed untranslated. The exception handler may re-enable translation as appropriate.
5. During this step, the processor determines the vector number for the exception. For vectored interrupts, the processor latches the vector number directly from the interrupt controller interface to the core. For all other exceptions, internal logic provides the vector number. This vector number is used in the last step to calculate the address of the exception vector by multiplying it by four to convert it to a vector offset. The vector number associated with the exception is loaded into the VEC field of the PSR to assist shared exception handlers.
6. During this step, the processor determines the address of the first instruction of the exception handler and then passes control to the handler. The processor combines the vector offset with the value contained in the vector base register to obtain the memory address of the exception vector. Next, the processor fetches a word from the vector table entry, loads the new program counter (PC) value from the exception vector table entry with the address of the first instruction of the exception handler, and loads the PSR(AF) control bit from the low-order bit of the vector table entry to determine which register file to use when the exception handler is entered. The processor then resumes execution at the new PC location.

## 4.5 Exception Vectors

The M•CORE supports a 512-byte vector table containing 128 exception vectors (see [Table 4-1](#)). All exception vectors are located in the supervisor address space and are accessed using program relative references. Only the reset and soft reset vectors are fixed in the processor's memory map. Once initialization is complete, the base address of the exception vector table can be relocated after reset by programming the VBR.

**Table 4-1. Exception Vector Assignments**

Vector Number(s)	Vector Offset (Hex)	Assignment
0	000	Reset
1	004	Misaligned access
2	008	Access error
3	00C	Divide by zero
4	010	Illegal instruction
5	014	Privilege violation
6	018	Trace exception
7	01C	Breakpoint exception
8	020	Unrecoverable error
9	024	Soft reset <sup>(1)</sup>
10	028	$\overline{\text{INT}}$ autovector
11	02C	$\overline{\text{FINT}}$ autovector
12	030	Hardware accelerator <sup>(1)</sup>
13	034	(Reserved)
14	038	
15	03C	
16–19	040–04C	TRAP #0–3 instruction vectors
20–31	050–07C	Reserved
32–127	080–1FC	Reserved for vectored interrupt controller use

1. The M210/M210S core does not support the soft reset and hardware accelerator assignments.

The first 32 vectors are used for internally recognized exceptions. External devices such as an interrupt controller are provided with an additional 96 vectors. These vectors are used by supplying a seven-bit vector number along with an interrupt request. The M•CORE latches the interrupt vector number when the interrupt request is accepted, and vectors to the appropriate location. For external devices that cannot provide a vector, an autovector capability is provided. Separate autovectors are provided for normal interrupts ( $\overline{\text{INT}}$ ) and fast interrupts ( $\overline{\text{FINT}}$ ).

### 4.6 Exception Types

This subsection describes the external interrupt exceptions and the different types of exceptions generated internally by the M•CORE. These exceptions are discussed:

- Reset
- Misaligned access
- Access error
- Divide by zero
- Illegal instruction
- Privilege violation
- Trace
- Breakpoint
- Unrecoverable error
- Soft reset<sup>(1)</sup>
- Interrupt
- Fast interrupt
- Hardware accelerator<sup>(1)</sup>
- Trap instructions

---

1. Not offered on M210/M210S core.



#### 4.6.1 Reset Exception (Vector Offset 0x0)

The reset exception has the highest priority of any exception. It provides for system initialization and recovery from catastrophic failure. Reset aborts any processing in progress; processing cannot be recovered.

The reset exception places the processor in the supervisor mode by setting the S bit and disables tracing by clearing the TM field in the PSR. This exception also clears the PSR interrupt enable bits IE and FE, the debug mode bit (DB), and the hardware accelerator enable bits U[3:0]. The vector base register (VBR) is cleared to place the base of the exception table at address zero (0x00000000). The CPU fetches the reset vector from offset \$0 in the exception vector table, which is then loaded into the PC. Reset exception handling proceeds with the transfer of control to the memory location pointed to by the PC.

#### 4.6.2 Misaligned Access Exception (Vector Offset 0x4)

A misaligned access exception occurs when the processor attempts to perform a load or store of an operand which does not lie on a natural boundary consistent with the size of the access. This exception can be masked by setting the PSR(MM) bit to ignore alignment checks for data. The data is accessed from the next lower natural boundary in this mode. On exception handler entry, the EPC points to the instruction that attempted the misaligned access.

In addition to data-related misaligned access exceptions, the JMPI and JSRI instructions cause misaligned access exceptions to occur if the destination address of these change-of-flow instructions is odd. In this case, the EPC contains the value fetched, not the address of the JMPI or JSRI instruction. This is the only condition in which the EPC value is odd when an exception handler is entered.

**NOTE:** *If trace mode is enabled, the TP bit is not set in the EPSR, so the misaligned JMPI or JSRI is not traced automatically.*

*The PSR(MM) bit does not mask JMPI or JSRI related misaligned access exceptions.*

### 4.6.3 Access Error Exception (Vector Offset 0x8)

An access error exception occurs under certain conditions when the transfer error acknowledge ( $\overline{\text{TEA}}$ ) signal is asserted externally to terminate a bus cycle.

A bus error on an operand access always results in an access error exception, causing the processor to begin exception processing.

Bus errors that are signaled during instruction prefetches are deferred until the processor attempts to execute that instruction. At that time, the bus error exception is signaled and exception processing is initiated. If a bus error is encountered during an instruction prefetch cycle, but the corresponding instruction is never executed due to a change-of-flow in the instruction stream, the bus error is discarded. On exception handler entry, the EPC points to the instruction associated with the bus error.

The access error exception can also be used by an external memory management unit to cause exceptions to occur on TLB misses as well as on access violations.

### 4.6.4 Divide-by-Zero Exception (Vector Offset 0x0C)

Exception processing for divide instructions with a divisor of zero is similar to that for instruction traps. When the processor detects a divisor of zero for a divide instruction, it initiates exception processing instead of attempting to execute the instruction. On exception handler entry, the EPC points to the divide instruction.

### 4.6.5 Illegal Instruction Exception (Vector Offset 0x10)

Exception processing for illegal instructions is similar to that for instruction traps. When the processor decodes an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. On exception handler entry, the EPC points to the illegal instruction.

#### 4.6.6 Privilege Violation Exception (Vector Offset 0x14)

To provide system security, certain instructions are privileged. An attempt to execute one of these privileged instructions while in the user mode causes a privilege violation exception:

- Move from control register (MFCR)
- Move to control register (MTCR)
- Return from interrupt (RTI)
- Return from exception (RTE)
- Stop (STOP)
- Wait (WAIT)
- Doze (DOZE)

Exception processing for privilege violations is similar to that for illegal instructions. When the processor identifies a privilege violation, it begins exception processing before executing the instruction. On exception handler entry, the EPC points to the privileged instruction.

#### 4.6.7 Trace Exception (Vector Offset 0x18)

To aid in program development, the M•CORE includes an instruction-by-instruction and instruction change of flow tracing capability. In the instruction trace mode, each instruction generates a trace exception after the instruction completes execution, allowing a debugging program to monitor execution of a program. In change of flow trace mode, a trace exception is taken after each instruction which could cause a change of flow (BRANCH, JMP, etc.) The exception is taken regardless of the outcome of a conditional branch or loop instruction.

These instructions cause trace exceptions to be generated in the change-of-flow trace mode:

- Jump (JMP)
- Jump to subroutine (JSR)
- Jump indirect (JMPI)
- Jump to subroutine indirect (JSRI)

- Branch (BR)
- Branch on condition true (BT)
- Branch on condition false (BF)
- Branch to subroutine (BSR)
- Decrement with C-bit update and branch if condition true (LOOPT)

The trace exception is enabled regardless of the taken/not taken outcome of a conditional change-of-flow instruction.

If an instruction to be traced does not complete due to an instruction-related exception, trace exception processing is deferred, and no trace exception is taken or marked pending. This includes detecting a misaligned condition for the JMPL and JSR instructions. If an interrupt is pending at the completion of an instruction to be traced, trace exception processing is deferred and is marked as pending in the EPSR as part of exception recognition.

The TM field in the PSR controls tracing. The state of the TM field when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. See [2.4.2 Processor Status Register](#) for the definition of the TM field.

Trace exception processing starts at the end of normal processing for the traced instruction and before the start of the next instruction. On exception handler entry, the EPC points to the next instruction to be executed, not the traced instruction.

Certain control related instructions (RTE, RFI, TRAP, STOP, WAIT, DOZE, and BKPT) are never traced, although a trace exception may be taken as part of the normal execution of an RTE or RFI instruction if the EPSR(TP) bit is set. This occurs independent of the setting of the TM field in the PSR or EPSR.

If an interrupt is pending at the completion of an instruction to be traced, it assumes a higher priority, and the trace pending (TP) bit is set in the shadow PSR when the interrupt exception is processed. The execution of an RTE or RFI instruction (as appropriate) at the completion of the interrupt handler causes the pending trace exception to be taken.

#### 4.6.8 Breakpoint Exception (Vector Offset 0x1C)

The breakpoint instruction BKPT and the hardware breakpoint request input ( $\overline{\text{BRKRQ}}$ ) are assigned a unique exception vector. Refer to the appropriate microcontroller user's manual for operation of the  $\overline{\text{BRKRQ}}$  input. To minimize the chance that hardware breakpoint exceptions are lost, this exception has higher priority than interrupts when generated as a result of the  $\overline{\text{BRKRQ}}$  input signal. If the  $\overline{\text{BRKRQ}}$  input is asserted for an instruction prefetch, that instruction will not be executed, and a breakpoint exception will be taken if the instruction normally would begin execution (for instance, is not discarded as the result of a change of instruction flow). If the  $\overline{\text{BRKRQ}}$  input is asserted for a data fetch, a breakpoint exception is taken after the instruction completes. For the BKPT instruction, or for instruction accesses on which the  $\overline{\text{BRKRQ}}$  input is asserted, the EPC points to the instruction on exception handler entry. For data accesses that are marked with a  $\overline{\text{BRKRQ}}$  request, the EPC points to the next instruction.

#### 4.6.9 Unrecoverable Error Exception (Vector Offset 0x20)

Exceptions other than a fast interrupt exception that occur while the PSR(EE) bit is clear cause an unrecoverable exception to be generated, since the context necessary for exception recovery (previously saved in the EPC and EPSR shadow registers) is overwritten as a result of the unrecoverable error.

This error is usually indicative of a system failure, since software should be written in a manner that precludes exceptions while PSR(EE) remains cleared. Since the type of exception causing the unrecoverable error exception is unknown, on entry to the unrecoverable error exception handler, the EPC points to an instruction that may or may not have been executed.

#### 4.6.10 Soft Reset Exception (Vector Offset 0x24)

A soft reset exception is recognized when the  $\overline{\text{SRST}}$  input signal is asserted. This exception is non-maskable. The soft reset exception has the highest priority of any exception below a hard reset; it provides for

system recovery from catastrophic failure. A soft reset also aborts any processing in progress when  $\overline{\text{SRST}}$  is recognized; processing cannot be recovered. Soft reset exception processing begins once the  $\overline{\text{SRST}}$  input is negated.

The soft reset exception places the processor in the supervisor mode by setting the S bit and disables tracing by clearing the TM field in the PSR. This exception also clears the processor's interrupt enable bits IE and FE in the PSR and the exception shadowing enable bit EE. The hardware accelerator enable bits U[3:0] and the misalignment mask bit MM are undefined. The vector base register (VBR) is cleared to place the base of the exception table at address zero (0x00000000). The CPU fetches the soft reset vector from offset 0x24 in the exception vector table, which is then loaded into the PC. Reset exception handling proceeds with the transfer of control to the memory location pointed to by the PC.

**NOTE:** *The M210/M210S core does not support this machine exception.*

### 4.6.11 Interrupt Exceptions

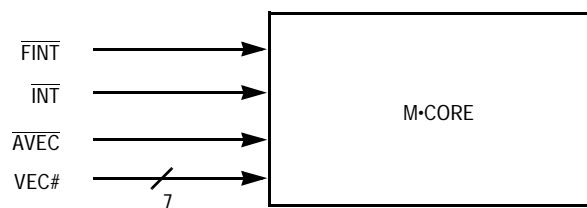
When a peripheral device requires the services of the M•CORE or is ready to send information that the processor requires, it can signal the processor to take an interrupt exception using the interrupt requests and vector signals.

Interrupts are normally recognized on instruction boundaries, although the worst-case interrupt latency may be minimized by allowing certain multi-cycle instructions to be interrupted prior to completion and then later restarted. The PSR(IC) bit may be set to allow interruption of these instructions prior to completion:

- Divide signed (DIVS)
- Divide unsigned (DIVU)
- Load multiple registers (LDM)
- Load register quadrant (LDQ)
- Multiply (MULT)
- Store multiple register (STM)
- Store register quadrant (STQ)

Two signals are provided for requesting interrupts, and both autovectoring and explicit vectoring capability is provided.

**Figure 4-1** shows the interrupt related interface signals to the CPU core. For vectorable interrupts ( $\overline{FINT}$  and  $\overline{INT}$ ), a seven-bit interrupt vector number can be supplied at the time a request is generated, or the autovector input  $\overline{AVEC}$  can be asserted to indicate that the appropriate predefined vector should be used. If  $\overline{AVEC}$  is asserted, the VEC# inputs are ignored.



**Figure 4-1. Interrupt Interface Signals**

#### 4.6.11.1 Normal Interrupt ( $\overline{INT}$ )

$\overline{INT}$  is the normal interrupt request input. It has the lowest priority of the interrupt inputs. The  $\overline{INT}$  input is masked when the PSR(IE) bit is clear. Normal interrupts use the EPSR and EPC exception shadow registers and consequently are also masked when PSR(EE) is cleared. When  $\overline{INT}$  is asserted, either the  $\overline{AVEC}$  input can be asserted to cause autovectoring to occur, or a 7-bit vector number can be provided to select one of vectors 32–127 to be used. (No explicit attempt is made by the processor to preclude use of vectors 0–31.) If a normal interrupt is autovectored, the vector at offset \$28 from the vector table base is used.

#### 4.6.11.2 Fast Interrupt ( $\overline{FINT}$ )

$\overline{FINT}$  is the fast interrupt request input. If enabled (PSR(FE) set), it has higher priority than the normal interrupt input  $\overline{INT}$ . Fast interrupts use the FPSR and FPC exception shadow registers and consequently are not masked when PSR(EE) is cleared. Fast interrupts are masked when PSR(FE) is cleared. When  $\overline{FINT}$  is asserted, either the  $\overline{AVEC}$  input can be asserted to cause autovectoring to occur, or a 7-bit vector number

can be provided to select one of vectors 32–127 to be used. (No explicit attempt is made by the processor to preclude use of vectors 0–31.) If a fast interrupt is autovector, the vector at offset 0x2C from the vector table base is used.

### 4.6.12 Hardware Accelerator Exception (Vector Offset 0x30)

The M•CORE provides a set of enable bits in the PSR (U[3:0]) for controlling execution of hardware accelerator instructions. When an attempt to execute an HAI opcode associated with a disabled accelerator block occurs, the instruction is aborted, and this exception is taken. This exception vector is also used for exceptions reported by a hardware unit as part of HAI instruction execution. Refer to the appropriate microcontroller user's manual for details of the operation of the HAI instruction and the hardware accelerator interface.

**NOTE:** *The M210/M210S core does not support this machine exception.*

### 4.6.13 Instruction Trap Exception (Vector Offset 0x40-0x5C)

Certain instructions are used to explicitly cause trap exceptions. The TRAP #N instruction always forces an exception and is useful for implementing system calls in user programs. On exception handler entry, the EPC points to the TRAP instruction.

## 4.7 Exception Priorities

Exceptions can be classified into five groups defined by specific characteristics and the order in which they are handled, as shown in [Table 4-2](#).

Reset exceptions override all other exceptions that occur at the same time. When exceptions of more than one type, other than reset, occur at the same time, they are handled according to the priority shown. A value of 1.0 represents the highest priority and a value of 6 represents the lowest priority.



**Table 4-2. Exception Priority Groups**

Group Priority	Exception and Relative Priority	Characteristics
1.0 1.1	Reset Soft Reset	The processor aborts all processing (instruction or exception) and does not save old context.
2	Hardware Breakpoint Request	Exception processing begins after completion of the current instruction.
3.0 3.1	Fast Interrupt Normal Interrupt	Exception processing begins when the current instruction is completed (C bit = 0), or certain instructions may be aborted for interrupt recognition.
4	Misaligned Access Access Error	The processor suspends processing and saves the processor context.
5	Illegal Instruction Privilege Violation Disabled Hardware Accelerator Divide by Zero Trap Instruction Hardware Accelerator Exception Breakpoint Instruction	Exception processing begins before the instruction is executed.  Exception processing begins when the instruction is executed.
6	Trace	Exception processing begins when the current instruction is completed.

Groups 4 and 5 are mutually exclusive, thus exceptions in those groups can have the same priority.

An unrecoverable exception may occur in place of an exception in groups 2 to 6 if the PSR(EE) bit is cleared.

When multiple exceptions are pending, the exception with the highest priority is processed first. The remaining exceptions may be regenerated when the original faulting instruction is restarted. [Table 4-3](#) shows the relationships between certain exceptions.

## 4.8 Returning from Exception Handlers

Returning from an exception handler is performed with the RFI or RTE instruction, depending on what type of handler is executing. The RFI instruction is used to return using the context saved in the FSPR and FPC shadow registers, and the RTE instruction uses the context stored in the EPSR and EPC registers.

**Table 4-3. Exceptions, Tracing, and BRKRQ Results**

**1. Exception recognized on decode boundary for instruction N type of exception(s)**

Ill/Priv/ trap/ bkpt	dacc	dbkpt	IACC (tea on N)	IBKPT (brkrq on N)	Interrupt Pending	Misalign	Trace of Previous Instruction Pending	Saved TP in EPSR	Saved EPC	Exception Taken
0	—	—	0	0	0	—	0	—	—	No exception
1	—	—	0	0	0	—	0	0	N	ill / priv / trap / bkpt
x	—	—	x	0	1	—	0	0	N	Interrupt
x	—	—	x	1	x	—	0	0	N	Breakpoint
x	—	—	1	0	0	—	0	0	N	Access error
x	—	—	x	0	0	—	1	0	N	Trace (for previous instruction)
x	—	—	x	0	1	—	1	1	N	Interrupt
x	—	—	x	1	x	—	1	1	N	Breakpoint

**2. Exception recognized during execution of MULT/DIV instruction M type of exception(s)**

Interrupted (IC Bit)	Divide by 0 (for divs and divu)	iacc (tea on M)	ibkpt (brkrq on M)	Interrupt Pending	Misalign	Trace of this Instruction Enabled	Saved TP in EPSR	Saved EPC	Exception Taken
x	1	—	—	0	—	x	0	M	Divide by zero
1	x	—	—	1	—	x	0	M	Interrupt
x	1	—	—	1	—	x	0	M	Interrupt

**Table 4-3. Exceptions, Tracing, and BRKRQ Results (Continued)**

**3. Exception recognized during execution of LD/ST class instruction M type of exception(s)**

Interrupted (IC bit)	dacc (tea on acc)	dbkpt (brkrq on acc)	iacc (tea on M)	ibkpt (brkrq on M)	Interrupt Pending	Misalign	Trace of this Instruction Enabled	Saved TP in EPSR	Saved EPC	Exception Taken
—	—	—	—	—	0	1	x	0	M	Misaligned
—	—	—	—	—	1	1	x	0	M	Interrupt
0	0	1	—	—	0	—	0	0	M+ 2	Breakpoint
0	0	1	—	—	0	—	1	1	M+ 2	Breakpoint
0	0	1	—	—	1	—	0	0	M+ 2	Breakpoint
0	0	1	—	—	1	—	1	1	M+ 2	Breakpoint
0	1	0	—	—	0	—	x	0	M	Access error
0	1	0	—	—	1	—	x	0	M	Interrupt
0	1	1	—	—	x	—	x	0	M	Breakpoint
1	x	0	—	—	x	—	x	0	M	Interrupt
1	x	1	—	—	1	—	x	0	M	Breakpoint

**4. Exception recognized during execution of change of flow class instruction N type of exception(s)**

Instruction	iacc on Table Access	ibkpt on Table Access	iacc on dest Access	ibkpt on dest Fetch	Interrupt Pending	Trace of this Instruction Enabled	Saved TP in EPSR	Saved EPC	Exception Taken
BR, BSR, JMP, JSR, JMPI, JSRI, LOOP	—	—	x	1	0	0	0	Destination	Breakpoint
	—	—	x	1	0	1	1	Destination	Breakpoint
	—	—	x	1	1	0	0	Destination	Breakpoint
	—	—	x	1	1	1	1	Destination	Breakpoint
	—	—	1	0	0	0	0	Destination	Access Error
	—	—	1	0	0	1	1	Destination	Access Error
	—	—	1	0	1	0	0	Destination	Interrupt
	—	—	1	0	1	1	1	Destination	Interrupt
JMPI, JSRI	0	1	—	—	x	0	0	Destination	Breakpoint
	0	1	—	—	x	1	1	Destination	Breakpoint
	1	0	—	—	0	x	0	N	Access Error
	1	0	—	—	1	x	0	N	Interrupt
	1	1	—	—	x	x	0	N	Breakpoint



## Section 5. Core Interface

### 5.1 Contents

5.2	Introduction .....	182
5.3	Signal Descriptions .....	182
5.3.1	Address Bus (ADDR[31:0]) .....	185
5.3.2	Data Bus (DATA[31:0]) .....	185
5.3.3	Transfer Request ( $\overline{\text{TREQ}}$ ) .....	185
5.3.4	Transfer Busy ( $\overline{\text{TBUSY}}$ ) .....	185
5.3.5	Transfer Abort ( $\overline{\text{ABORT}}$ ) .....	185
5.4	Transfer Attribute Signals .....	185
5.4.1	Transfer Code (TC[2:0]) .....	186
5.4.2	Read/Write (R/W) .....	186
5.4.3	Transfer Size (TSIZ[1:0]) .....	186
5.4.4	Sequential Access ( $\overline{\text{SEQ}}$ ) .....	187
5.4.5	Data to Address (D2A) .....	187
5.5	Transfer Control Signals .....	187
5.5.1	Transfer Acknowledge ( $\overline{\text{TA}}$ ) .....	187
5.5.2	Transfer Error Acknowledge ( $\overline{\text{TEA}}$ ) .....	188
5.5.3	Breakpoint Request ( $\overline{\text{BRKRQ}}$ ) .....	188
5.6	Memory Management Control Signals .....	188
5.6.1	Translate Control ( $\overline{\text{TE}}$ ) .....	188
5.6.2	Soft Reset ( $\overline{\text{SRST}}$ ) .....	188
5.7	Interrupt Control Signals .....	189
5.7.1	Normal Interrupt Request ( $\overline{\text{INT}}$ ) .....	189
5.7.2	Fast Interrupt Request ( $\overline{\text{FINT}}$ ) .....	189
5.7.3	Interrupt Pending Status ( $\overline{\text{IPEND}}$ ) .....	189
5.7.4	Interrupt Vector Number (VEC[6:0]) .....	189
5.7.5	Autovector ( $\overline{\text{AVEC}}$ ) .....	190

5.8	Power Management Control Signals .....	190
5.9	Status and Clock Signals .....	191
5.9.1	Processor Status (PSTAT[3:0]) .....	191
5.9.2	M•CORE Processor Clock (CLK) .....	192
5.10	Global Status and Control Interface .....	192
5.11	Hardware Accelerator Interface .....	192
5.12	Debug/Emulation Support Signals .....	193
5.12.1	Debug Request ( $\overline{\text{DBGREQ}}$ ) .....	193
5.12.2	Debug Acknowledge ( $\overline{\text{DBGACK}}$ ) .....	193
5.13	Test Signals .....	193
5.14	Power Supply Connections. ....	193
5.15	Signal Summary .....	193

## 5.2 Introduction

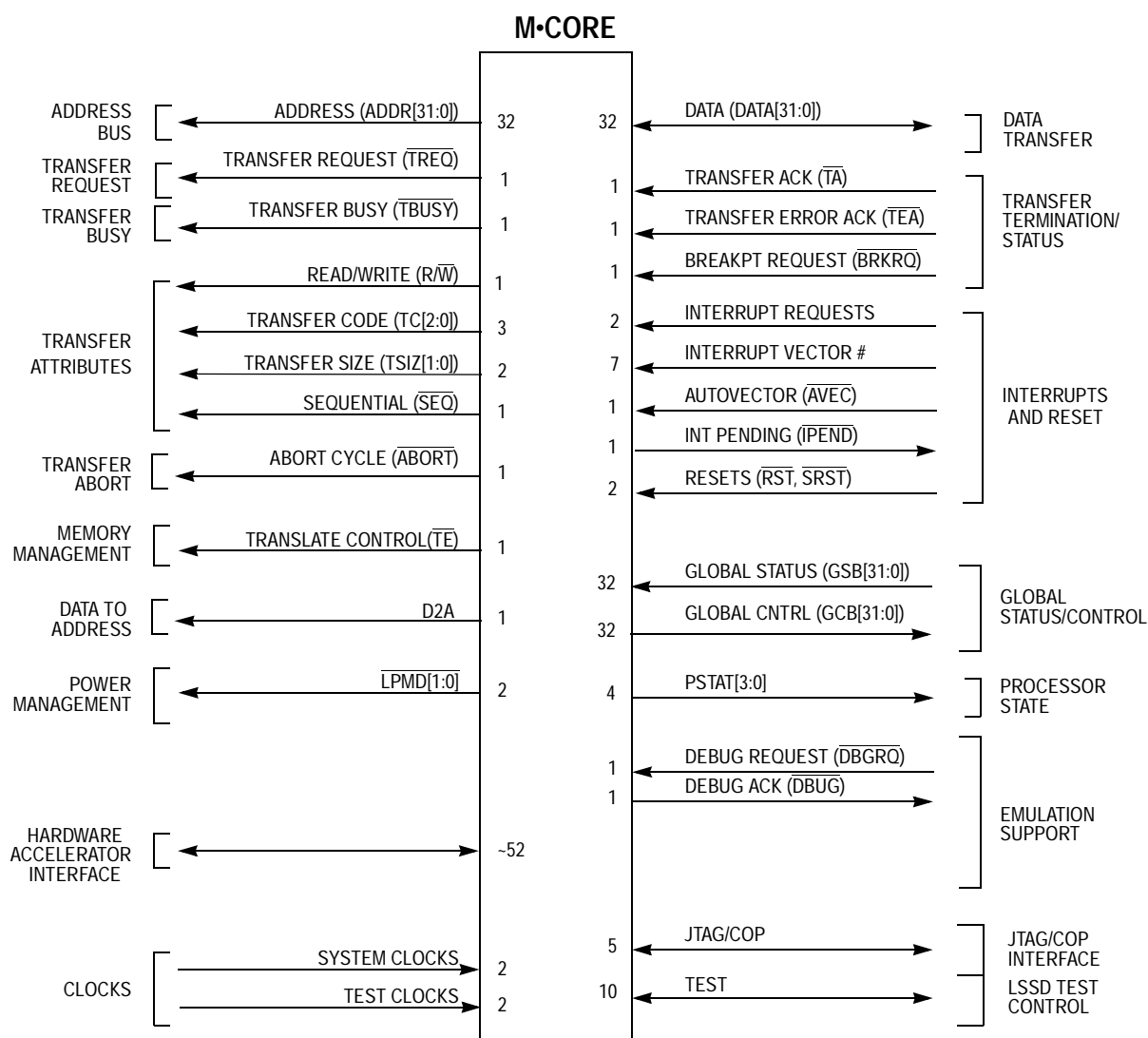
This section describes the interface to the M•CORE. Signals and the data transfer protocols are described.

## 5.3 Signal Descriptions

**Figure 5-1** shows functional grouping of M•CORE signals. **Table 5-1** lists M•CORE signal names, mnemonics, and functional descriptions of the signals.

In this subsection, each signal is described concisely. When necessary, reference is made to detailed information about the signal and related operations.

**NOTE:** See **Appendix C. M210/M210S Core Interface** and **Appendix D. M210/M210S Interface Operation** for specific description of the MM210/M210S core.



**Figure 5-1. M•CORE Signal Groups**

**Table 5-1. Signal Index**

Signal Name	Mnemonic	Function
Address bus	ADDR[31:0]	32-bit address bus
Data bus	DATA[31:0]	32-bit data bus used to transfer up to 32 bits of data per bus transfer
Transfer code	TC[2:0]	Indicate the general transfer type: supervisor/user/instruction/data
Read/write	$R/\overline{W}$	Identifies the transfer as a read or write
Transfer size	TSIZ[1:0]	Indicates the data transfer size — these signals, together with ADDR[0:1] define the active sections of the data bus

Table 5-1. Signal Index (Continued)

Signal Name	Mnemonic	Function
Sequential	$\overline{\text{SEQ}}$	Indicates the next access is sequential
Data to address	D2A	Indicates the next access address must be read from the data bus
Transfer busy	$\overline{\text{TBUSY}}$	Indicates a bus cycle is in progress
Transfer request	$\overline{\text{TREQ}}$	Indicates a request for a bus cycle
Transfer acknowledge	$\overline{\text{TA}}$	Asserted to acknowledge a bus transfer
Transfer error acknowledge	$\overline{\text{TEA}}$	Indicates an error condition exists for a bus transfer
Abort cycle	$\overline{\text{ABORT}}$	Output used to abort a requested access
Break request	$\overline{\text{BRKRQ}}$	Input to signal hardware breakpoint exception for an access
Reset in	$\overline{\text{RST}}$	Processor reset
Interrupt request	$\overline{\text{INT}}$	Normal interrupt request to the processor
Fast interrupt request	$\overline{\text{FINT}}$	Fast interrupt request to the processor
Soft reset	$\overline{\text{SRST}}$	Soft reset input
Interrupt vector number	VEC[6:0]	Interrupt vector number
Autovector	$\overline{\text{AVEC}}$	Used to request internal generation of the interrupt vector number
Interrupt pending	$\overline{\text{IPEND}}$	Indicates an interrupt is pending internally
Processor clock	CLK	Clock input
Low-power mode	$\overline{\text{LPMD}}[1:0]$	Outputs used to indicate low-power mode(s)
Debug request	$\overline{\text{DBGREQ}}$	Input to signal hardware to enter debug mode
Debug acknowledge	$\overline{\text{DBGACK}}$	Output to signal that processor has entered debug mode
Processor status	PSTAT[3:0]	Processor status outputs
Translate control	$\overline{\text{TE}}$	Control address translation or alternate control function
Global control bus	GCB[31:0]	Global control bus outputs
Global status bus	GSB[31:0]	Global status bus inputs
Hardware accumulator interface	HAI	Hardware accelerator interface signals
Test Interface	TBD	Test interface signals to be determined.
Emulator interface	TBD	Emulator interface signals to be determined
Power supply	V <sub>CC</sub>	Power supply
Ground	GND	Ground connection



### 5.3.1 Address Bus (ADDR[31:0])

These signals provide the address for a bus transfer.

### 5.3.2 Data Bus (DATA[31:0])

These three-state bidirectional signals provide the general-purpose data path between the M•CORE and all other devices. The data bus can transfer 8, 16, or 32 bits of data per bus transfer.

### 5.3.3 Transfer Request ( $\overline{\text{TREQ}}$ )

The processor drives this active-low signal to indicate that a new access has been requested. This signal is driven for a single cycle along with address and transfer attribute signals to request a new cycle.

### 5.3.4 Transfer Busy ( $\overline{\text{TBUSY}}$ )

The processor drives this active-low signal to indicate that an access is in progress. This signal is driven for the duration of a cycle, and may be held asserted for multiple transfers.

### 5.3.5 Transfer Abort ( $\overline{\text{ABORT}}$ )

The processor drives this active-low signal to indicate that a requested access must be aborted. This signal may be driven the clock following a valid requested cycle. The processor must receive a termination signal (either  $\overline{\text{TEA}}$  or  $\overline{\text{TA}}$ ) from external logic the same clock cycle  $\overline{\text{ABORT}}$  is asserted.

## 5.4 Transfer Attribute Signals

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer cycle. Refer to [Section 6. Interface Operation](#) for detailed information about the relationship of the transfer attribute signals to bus operation.

### 5.4.1 Transfer Code (TC[2:0])

The processor drives these signals to indicate the type of access for the current bus cycle. [Table 5-2](#) shows the definitions of the TCx encoding.

**Table 5-2. Transfer Code Encoding**

TC[2:0]	Transfer Type
000	User data access <sup>(1)</sup>
001	Reserved
010	User instruction access <sup>(2)</sup>
011	User change of flow instruction access <sup>(3)</sup>
100	Supervisor data access <sup>(1)</sup>
101	Supervisor exception vector access
110	Supervisor instruction access <sup>(2)</sup>
111	Supervisor change of flow instruction access <sup>(3)</sup>

1. Except **LRW** accesses

2. Except change of flow related instruction accesses, includes **LRW** accesses

3. Change of flow related instruction access for taken branches, jumps, and LOOPT instructions (includes table accesses for JMPL, JSRI)

### 5.4.2 Read/Write (R/W)

This output signal defines the data transfer direction for the current bus cycle. Logic level one indicates a read cycle, and a logic level zero indicates a write cycle.

### 5.4.3 Transfer Size (TSIZ[1:0])

These output signals indicate the data size for the bus cycle. [Table 5-3](#) shows the definitions of the TSIZx encoding.

**Table 5-3. TSIZx Encoding**

TSIZ1	TSIZ0	Transfer Size
0	0	Word (4 bytes)
0	1	Byte
1	0	Half-word (2 bytes)
1	1	Reserved

#### 5.4.4 Sequential Access ( $\overline{\text{SEQ}}$ )

This active-low output indicates that the current access is sequential address order from the last access. This signal is driven for sequential instruction fetches as well as for sequential data transfers for the LDM, STM, LDQ, and STQ instructions.

**NOTE:** *This signal is not currently implemented.*

#### 5.4.5 Data to Address (D2A)

This active-high output indicates that the data received for the current read access is driven as the next access address. This signal is driven for table accesses for the JMPL and JSRL instructions as well as for cases where load data is to be used as a jump or jsr destination prefetch in the following access. The timing for this control signal is the same as address timing.

### 5.5 Transfer Control Signals

The following signals provide control functions for bus cycles when the M•CORE is the bus master. Refer to [6.5 Processor Instruction/Data Transfers](#) for detailed information about the relationship of the bus cycle control signals to bus operation.

#### 5.5.1 Transfer Acknowledge ( $\overline{\text{TA}}$ )

This active-low input indicates the completion of a requested data transfer operation. During transfers by the M•CORE,  $\overline{\text{TA}}$  is an input signal from the referenced slave device indicating completion of the transfer. For the M•CORE to accept the transfer as successful with a transfer acknowledge,  $\overline{\text{TEA}}$  must be negated when  $\overline{\text{TA}}$  is asserted.

### 5.5.2 Transfer Error Acknowledge ( $\overline{\text{TEA}}$ )

The current slave asserts this active-low input signal to indicate an error condition for the current transfer to immediately terminate the bus cycle. The assertion of  $\overline{\text{TEA}}$  has precedence over  $\overline{\text{TA}}$ .

### 5.5.3 Breakpoint Request ( $\overline{\text{BRKRQ}}$ )

This active-low input signal is asserted by external logic to request a breakpoint to be associated with the current access. This signal is sampled when the assertion of  $\overline{\text{TEA}}$  or  $\overline{\text{TA}}$  is recognized. A breakpoint is taken if the access is a data access or if the access is an instruction fetch which is not discarded due to a change of flow.

## 5.6 Memory Management Control Signals

These signals can be used to control an optional external memory management unit.

### 5.6.1 Translate Control ( $\overline{\text{TE}}$ )

When asserted, this active-low output signal may be used to indicate that access addresses should be translated by an optional external memory management unit, or may be used for an alternate function. The  $\overline{\text{TE}}$  output is asserted (driven low) while the PSR(TE) bit is set. Refer to [2.4 Supervisor Programming Model](#) for more information on the operation of the PSR(TE) bit.

### 5.6.2 Soft Reset ( $\overline{\text{SRST}}$ )

The assertion of this active-low input signal causes the M•CORE to enter soft reset exception processing. Refer to [6.9 Reset Operation](#) for a description of soft reset operation and to [4.6.10 Soft Reset Exception \(Vector Offset 0x24\)](#) for information about the soft reset exception.

## 5.7 Interrupt Control Signals

These paragraphs describe the signals which control the interrupt functions.

### 5.7.1 Normal Interrupt Request ( $\overline{\text{INT}}$ )

This active-low input signal provides a normal interrupt request condition to the M•CORE. This signal is level sensitive. Refer to [4.6.11 Interrupt Exceptions](#) for information on normal interrupts.

### 5.7.2 Fast Interrupt Request ( $\overline{\text{FINT}}$ )

This active-low input signal provides a fast interrupt request condition to the M•CORE. This signal is level sensitive. Refer to [4.6.11 Interrupt Exceptions](#) for information on fast interrupts.

### 5.7.3 Interrupt Pending Status ( $\overline{\text{IPEND}}$ )

This active-low output signal indicates that an interrupt request has been recognized internally by the processor and is enable by the appropriate bit PSR. External devices (other bus masters or a bus arbiter) can use  $\overline{\text{IPEND}}$  to be alerted of a pending interrupt condition. External power management logic may also use this output to control operation of the core and other logic.

### 5.7.4 Interrupt Vector Number (VEC[6:0])

These input signals provide the vector number to be used when exception processing begins for an incoming interrupt request. These signals are sampled along with the  $\overline{\text{FINT}}$  and  $\overline{\text{INT}}$  inputs, and must be driven to a valid value when either of these signals is asserted, unless the  $\overline{\text{AVEC}}$  signal is asserted. If  $\overline{\text{AVEC}}$  is asserted, these inputs are not used.

### 5.7.5 Autovector ( $\overline{\text{AVEC}}$ )

This active-low input signal is asserted with either  $\overline{\text{INT}}$  or  $\overline{\text{FINT}}$  to request internal generation of the vector number. Refer to [4.6.11 Interrupt Exceptions](#) for more information about automatic vectoring.

## 5.8 Power Management Control Signals

Two low-power mode output signals ( $\overline{\text{LPMD}}[1:0]$ ) are provided for power management by external control.

The  $\overline{\text{LPMD}}[1:0]$  output signals are asserted by the processor when execution of a DOZE, STOP, or WAIT instruction occurs, as shown in [Table 5-4](#). These active-low outputs may be used by external logic to place the processor and system logic in a low-power stopped state. The  $\overline{\text{LPMD}}[1:0]$  outputs assert for one or more clock cycles for execution of a DOZE, STOP, or WAIT instruction until a valid pending interrupt is detected by the core, or until a request to enter debug mode is made via the assertion of the  $\overline{\text{DBGRQ}}$  input signal.

**Table 5-4.  $\overline{\text{LPMD}}[1:0]$  Encoding**

$\overline{\text{LPMD}}1$	$\overline{\text{LPMD}}0$	Mode
0	0	STOP
0	1	WAIT
1	0	DOZE
1	1	Normal

The processor can be placed in a low-power state by forcing the CLK input high.

External logic must detect the asserted edge of these signals to determine that a low-power instruction has been executed. The  $\overline{\text{IPEND}}$  output signal (or other system state) may be monitored to determine when to release the processor (and system if applicable) from the stopped condition. This can be done by re-enabling the processor CLK.

## 5.9 Status and Clock Signals

This subsection describes the signals that provide timing and the internal processor status.

### 5.9.1 Processor Status (PSTAT[3:0])

These outputs indicate the internal execution unit status. The timing is synchronous with the M•CORE processor clock (CLK), and the status may have nothing to do with the current bus transfer. [Table 5-5](#) lists the definition of the PSTATx encodings.

**Table 5-5. PSTATx Encoding**

Hex	PSTAT3	PSTAT2	PSTAT1	PSTAT0	Internal Processor Status
\$0	0	0	0	0	Execution stalled
\$1	0	0	0	1	Execution stalled
\$2	0	0	1	0	Execute exception
\$3	0	0	1	1	Reserved
\$4	0	1	0	0	Processor in stop, wait or doze state
\$5	0	1	0	1	Execution stalled
\$6	0	1	1	0	Processor in debug mode
\$7	0	1	1	1	Reserved
\$8	1	0	0	0	Launch instruction <sup>(1)</sup>
\$9	1	0	0	1	Launch LDM, STM, LDQ, or STQ
\$A	1	0	1	0	Launch hardware accelerator instruction
\$B	1	0	1	1	Launch LRW
\$C	1	1	0	0	Launch change of program flow instruction
\$D	1	1	0	1	Launch RTE or RFI
\$E	1	1	1	0	Reserved
\$F	1	1	1	1	Launch JMPI or JSRI

1. Except RTE, RFI, LDM, STM, LDQ, STQ, LRW, hardware accelerator, or change of flow instructions

### 5.9.2 M•CORE Processor Clock (CLK)

CLK is the synchronous clock of the M•CORE. This signal is used internally to clock the logic of the M•CORE processor core.

Since the M•CORE is designed for static operation, CLK can be gated off (forced high) to lower power dissipation (for example, during low-power stopped states). Refer to [6.11 Interrupt Interface Operation](#) for more information on low-power stopped states.

### 5.10 Global Status and Control Interface

The M•CORE provides two control registers as part of the supervisor programming model to monitor global status in the integrated system as well as to provide global control outputs to the system. Refer to [2.4 Supervisor Programming Model](#) for more information on these registers.

The GCB[31:0] outputs change state when the GCR register is updated by the MTCR instruction.

The GSB[31:0] inputs are sampled by the core and the corresponding values appear in the GSR for transfer to a general register when a MFCR instruction referencing the GSR is executed.

Refer to [6.12 Global Status and Control Interface Operation](#) for more information on the timing for these signals.

### 5.11 Hardware Accelerator Interface

This group of signals is used to interface the M•CORE to one or more external hardware blocks used for task acceleration. Examples include high performance arithmetic units (multiply, multiply/accumulate), CRC generation logic, etc. For a description of this interface and its operation refer to [Section 7. Hardware Accelerator Interface \(HAI\)](#).



## 5.12 Debug/Emulation Support Signals

These interface signals are provided to assist in implementing an on-chip emulation capability with a controller external to the M•CORE.

### 5.12.1 Debug Request ( $\overline{\text{DBGRQ}}$ )

The  $\overline{\text{DBGRQ}}$  input (active-low) is used to place the processor core in debug mode. Refer to [Section 4. Exception Processing](#) and [Section 8. JTAG Test Access Port and OnCE](#) for details of debug mode operation.

### 5.12.2 Debug Acknowledge ( $\overline{\text{DEBUG}}$ )

The  $\overline{\text{DEBUG}}$  output (active-low) is used to indicate that the processor has entered/exited debug mode. Refer to [Section 4. Exception Processing](#) and [Section 8. JTAG Test Access Port and OnCE](#) for details of debug mode operation.

## 5.13 Test Signals

Test signals are currently being defined.

## 5.14 Power Supply Connections

The M•CORE requires connections to a  $V_{CC}$  power supply, positive with respect to ground. The  $V_{CC}$  and ground connections must be planned to supply adequate current to the various sections of the processor.

## 5.15 Signal Summary

[Table 5-6](#) is a summary of the electrical characteristics of the M•CORE signals.

Table 5-6. Signal Summary

Signal Name	Mnemonic	Input/ Output	Active State	Reset State
Address bus	ADDR[31:0]	Output	High	Undefined
Data bus	DATA[31:0]	Input/Output	High	Three-States
Transfer code	TC[2:0]	Output	High	Undefined
Read/write	R/W	Output	High/Low	High
Transfer size	TSIZ[1:0]	Output	High	Undefined
Transfer request	$\overline{\text{TREQ}}$	Output	Low	Negated
Transfer busy	$\overline{\text{TBUSY}}$	Output	Low	Negated
Transfer abort	$\overline{\text{ABORT}}$	Output	Low	Negated
Transfer acknowledge	$\overline{\text{TA}}$	Input	Low	—
Transfer error acknowledge	$\overline{\text{TEA}}$	Input	Low	—
Breakpoint request	$\overline{\text{BRKRQ}}$	Input	Low	—
Reset in	$\overline{\text{RST}}$	Input	Low	—
Normal interrupt request	$\overline{\text{INT}}$	Input	Low	—
Fast interrupt request	$\overline{\text{FINT}}$	Input	Low	—
Soft reset	$\overline{\text{SRST}}$	Input	Low	—
Vector number	VEC[6:0]	Input	High	—
Autovector request	$\overline{\text{AVEC}}$	Input	Low	—
Interrupt pending	$\overline{\text{IPEND}}$	Output	Low	Negated
Processor clock	CLK	Input	—	—
Low-power mode	$\overline{\text{LPMD}}[1:0]$	Output	Low	Negated
Debug request	$\overline{\text{DBGREQ}}$	Input	Low	—
Debug acknowledge	$\overline{\text{DBGACK}}$	Output	Low	Negated
Processor status	PSTAT[3:0]	Output	High	Undefined
Translate control	$\overline{\text{TE}}$	Output	Low	Negated
Global control bus	GCB[31:0]	Output	High	Undefined
Global status bus	GSB[31:0]	Input	High	—
Power supply	V <sub>CC</sub>	Input	—	—
Ground	GND	Input	—	—

## Section 6. Interface Operation

### 6.1 Contents

6.2	Introduction . . . . .	196
6.3	Bus Characteristics . . . . .	196
6.4	Data Transfer Mechanism . . . . .	197
6.5	Processor Instruction/Data Transfers . . . . .	199
6.5.1	Instruction and Data Read Transfer Cycles . . . . .	200
6.5.2	Read Transfer Cycles with Wait State(s) . . . . .	202
6.5.3	Write Transfer Cycles . . . . .	202
6.5.4	Write Transfer Cycles with Wait State(s) . . . . .	205
6.5.5	Data Bus Hand-Off Between Read and Write Cycles . . . . .	206
6.6	Exception Bus Control Cycles . . . . .	207
6.6.1	Bus Errors . . . . .	208
6.6.2	Breakpoint Requests . . . . .	208
6.7	$\overline{\text{ABORT}}$ Signal Operation . . . . .	209
6.8	D2A Signal Operation . . . . .	210
6.9	Reset Operation . . . . .	211
6.9.1	Hard Reset (Power-On Reset) . . . . .	211
6.9.2	Soft Reset . . . . .	211
6.10	Memory Management Interface Operation . . . . .	212
6.11	Interrupt Interface Operation . . . . .	212
6.12	Global Status and Control Interface Operation . . . . .	214
6.13	Power Management Interface Operation . . . . .	215
6.14	Emulation/Debug Interface Operation . . . . .	217

### 6.2 Introduction

The M•CORE interface supports synchronous data transfers between the processor and other devices in the system. This section provides a functional description of the interface, the signals that control the interface, and the bus cycles provided for data transfer operations. Descriptions of the power management signals, external memory management unit control, and the reset operation are also included.

**NOTE:** See [Appendix C. M210/M210S Core Interface](#) and [Appendix D. M210/M210S Interface Operation](#) for specific description of the MM210/M210S core.

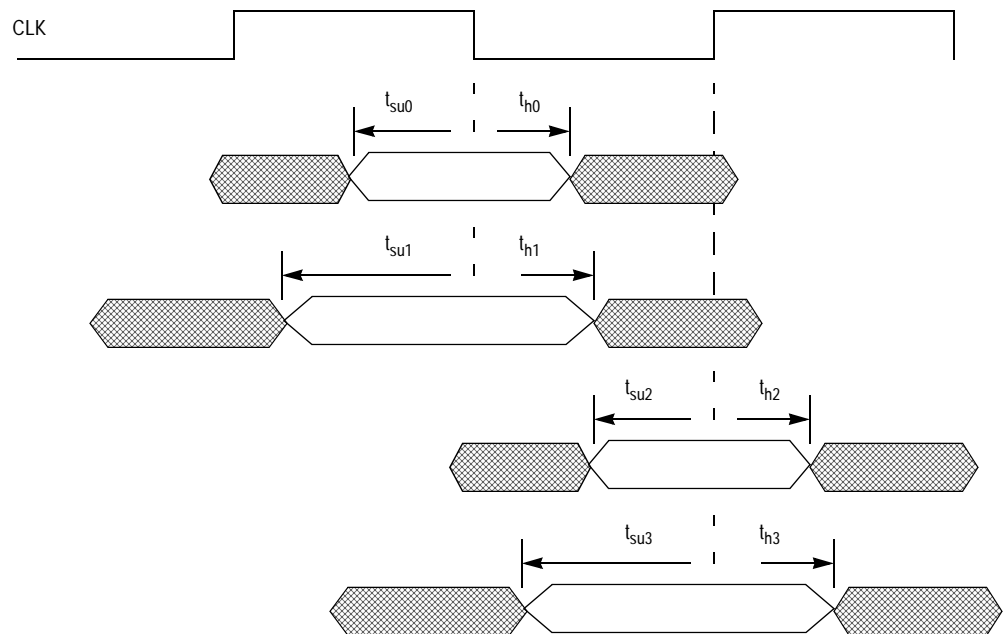
### 6.3 Bus Characteristics

The M•CORE uses the address bus (ADDR[31:0]) to specify the address for a data transfer and the data bus (DATA[31:0]) to transfer the data. Control and attribute signals indicate the beginning and type of a bus cycle as well as the address space and size of the transfer. The selected device then controls the length of the cycle by terminating it using the control signals.

The M•CORE CLK is distributed internally to provide logic timing.

Inputs to the M•CORE (other than the interrupt requests and reset signals) are synchronously sampled and must be stable during the sample window(s) defined by  $t_{su3-0}$  and  $t_{h3-0}$  (see [Figure 6-1](#)) to guarantee proper operation. The  $\overline{INT}$ ,  $\overline{FINT}$ , and  $\overline{SRST}$  signals are sampled on the rising edge of CLK, but are also used in an asynchronous fashion for power management control.

Outputs from the M•CORE transition on one of the two clock edges depending on the signal class.



**Figure 6-1. Signal Relationships to Clocks**

## 6.4 Data Transfer Mechanism

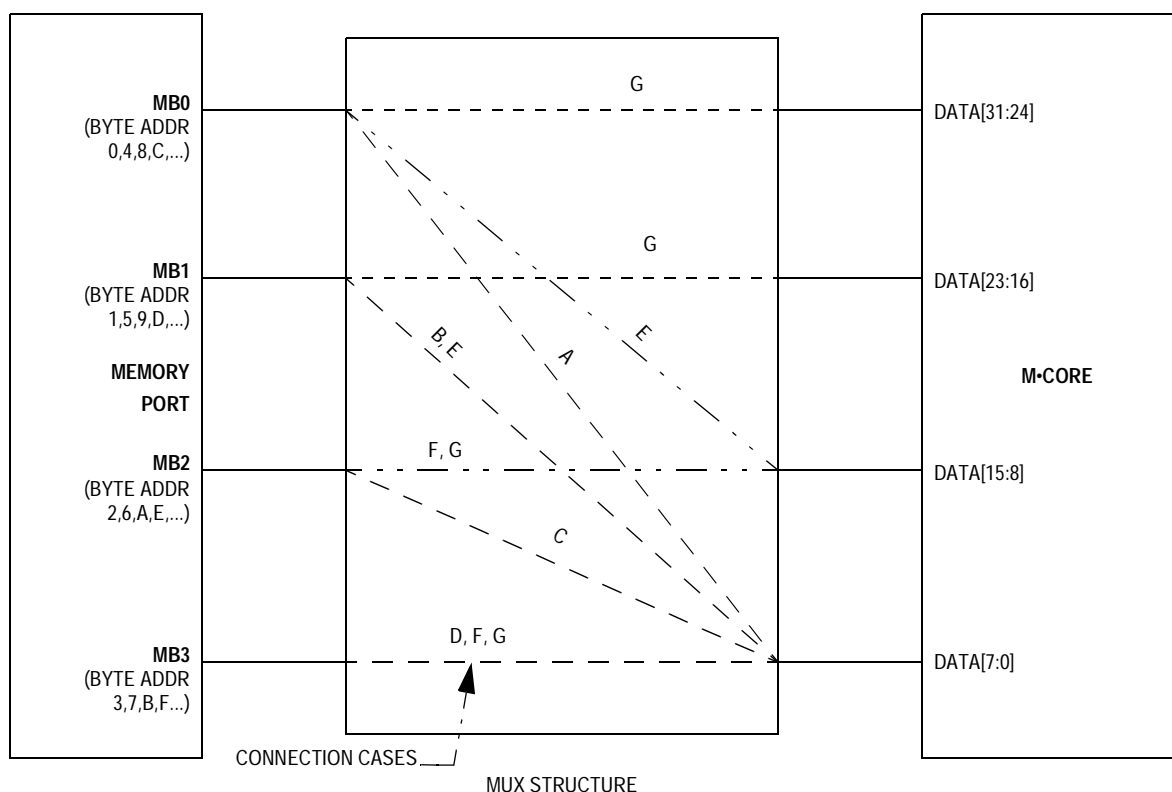
Data transfers occur between an internal register and the external bus. The internal register connects to the external data bus through the internal data bus and a data multiplexer. The data multiplexer establishes the necessary connections for different combinations of address and data sizes. This multiplexer is physically positioned in the overall system to minimize power consumption by minimizing loading and reducing unnecessary signal transitions. Logically, however, it is considered part of the M•CORE.

The M•CORE does not support dynamic bus sizing and expects the referenced device to accept the requested access width. Peripherals with an interface width of N bits should not define internal registers greater than N bits wide.

Additionally, no misaligned transfers are supported. The M•CORE interface may drive the ADDR1 and ADDR0 address lines to a value which is not representative of an aligned transfer, but expects aligned

data to be transferred. ADDR1 and ADDR0 should be selectively ignored by external logic based on the size of the transfer.

The data multiplexer takes the four bytes of the core interface data bus and routes them to their required positions to properly interface to memory and/or peripherals. The external mux connections to memory are controlled on a byte granularity and are referred to as MB[0:3] where MB0 resides at byte address 0 (mod4) and MB3 resides at byte address 3 (mod4). For example, MB0 would normally be routed to DATA[31:24] on a word transfer, but it can also be routed to DATA[7:0] for supporting a byte data transfer. The same is true for any of the other operand bytes. **Figure 6-2** shows the connection requirements for the mux. The transfer size (TSIZ[1:0]) and byte offset (ADDR1 and ADDR0) signals determine the positioning of the bytes.



**Figure 6-2. External Multiplexer Connections**

**Table 6-1** lists the combinations of the SIZx, ADDR1, and ADDR0 signals that are used for each possible transfer size and alignment. In **Table 6-1**, MB[0:3] indicate the portion of the requested operand that is read or written during that bus transfer. For word transfers, all bytes are valid as listed and correspond to portions of the requested operand. The bytes labeled with a dash are not required; they are ignored on read transfers and driven with undefined data on write transfers. Additional information on the encoding for the M•CORE signals can be found in **Section 5. Core Interface**.

**Table 6-1. Interface Requirements for Read and Write Cycles**

Transfer Size	Signal Encoding				Active Interface Bus Sections				Mux Connections
	TSIZ1	TSIZ0	ADDR1	ADDR0	DATA [31:24]	DATA [23:16]	DATA [15:8]	DATA [7:0]	
Byte	0	1	0	0	—	—	—	MB0	a
	0	1	0	1	—	—	—	MB1	b
	0	1	1	0	—	—	—	MB2	c
	0	1	1	1	—	—	—	MB3	d
Half-word	1	0	0	X	—	—	MB0	MB1	e
	1	0	1	X	—	—	MB2	MB3	f
Word	0	0	X	X	MB0	MB1	MB2	MB3	g

## 6.5 Processor Instruction/Data Transfers

The transfer of data between the processor and other devices involves the address bus, data bus, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting aligned byte, halfword, and word transfers. All bus input and output signals are sampled or driven with respect to one of the edges of the CLK signal. The M•CORE moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

Access requests are generated in an overlapped fashion in order to support sustained single cycle transfers. In addition, the M•CORE may choose to change the request address and attribute values if a previous

request is still pending. This might occur if an instruction transfer is not completed in a single cycle, and a data transfer becomes pending. In this case, the data request may replace a pending instruction request. Access requests are assumed to be accepted if there are no accesses in progress ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  negated), or if an access in progress is terminated the same cycle a new request is generated ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  asserted and one of  $\overline{\text{TA}}$ / $\overline{\text{TEA}}$  asserted). Once an access has been accepted, the processor is free to change the current request, thus access information must be latched by a slave device.

The M•CORE may also abort an accepted access the cycle following a valid (taken) request by asserting the  $\overline{\text{ABORT}}$  output signal during the clock cycle following a valid  $\overline{\text{TREQ}}$ . In this case, no access should occur, and external logic must supply a termination by asserting  $\overline{\text{TA}}$  or  $\overline{\text{TEA}}$  the same clock cycle  $\overline{\text{ABORT}}$  asserts. In the case of an aborted access, the address bus and all attributes associated with the aborted request are undefined.

The following paragraphs describe the bus cycles for instruction and data transfers, as well as the overlapped interface operation.

### 6.5.1 Instruction and Data Read Transfer Cycles

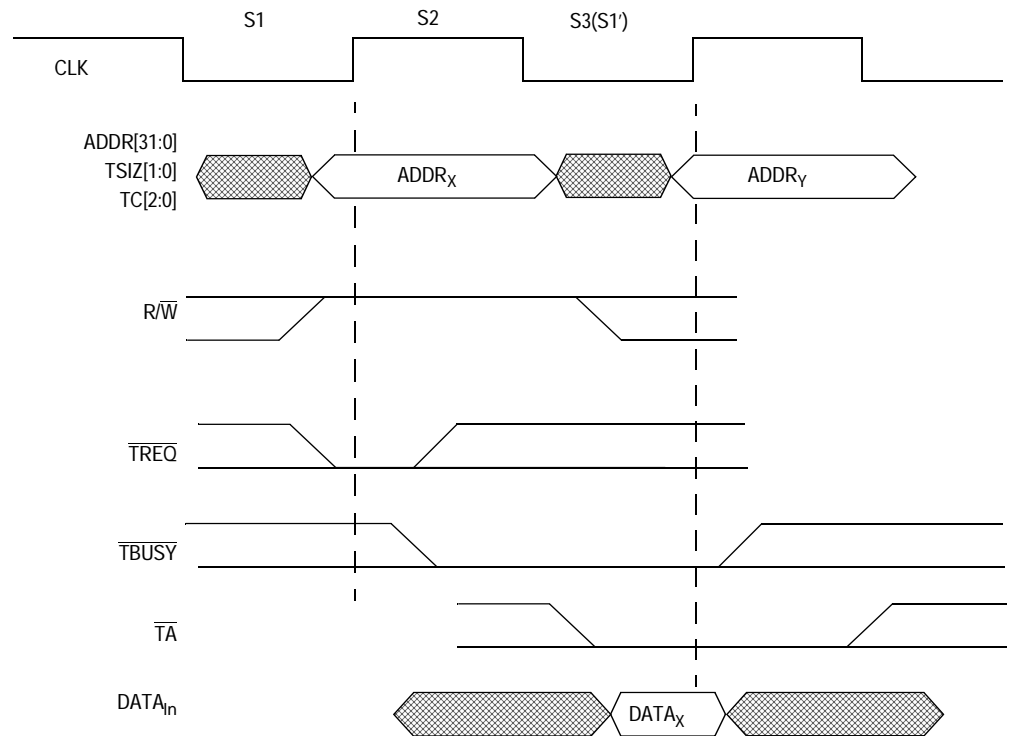
During a read transfer, the processor receives data from a memory or peripheral device. **Figure 6-3** is a functional timing diagram for instruction and data read transfers.

#### State1 (S1)

The read cycle starts in S1. During S1, the processor places valid values on the address bus and transfer attributes. The transfer code (TCx) signals identify the specific access type. The TSIZx pins indicate the size of the transfer. The read/write ( $\text{R}/\overline{\text{W}}$ ) signal is driven high for a read cycle.

The processor asserts transfer request ( $\overline{\text{TREQ}}$ ) during S1 to indicate that a transfer is being requested.





**Figure 6-3. Instruction/Data Read Cycle**

### State2 (S2)

During S2, the memory access takes place using the values of TSIZ<sub>1</sub>, TSIZ<sub>0</sub>, ADDR<sub>1</sub>, and ADDR<sub>0</sub> which are driven during S1 and S2 to enable reading of one or more bytes of memory. The  $\overline{\text{TBUSY}}$  signal is asserted to indicate that an access is in progress.

### State3 (S3)

The memory drives valid data to the core in S3.

The interface control logic uses the values of TSIZ<sub>1</sub>, TSIZ<sub>0</sub>, ADDR<sub>1</sub>, and ADDR<sub>0</sub> which were driven during S1 and S2 to place information on the data bus. If the memory responds without a wait state, then the transfer acknowledge ( $\overline{\text{TA}}$ ) signal is asserted.

The processor samples the level of  $\overline{\text{TA}}$ . If it is asserted, the current value is latched onto the data bus, the bus cycle terminates, and the data is passed to the appropriate unit of the processor. If  $\overline{\text{TA}}$  is not recognized as asserted at the end of the clock cycle, the processor ignores the data and inserts a wait state. The processor continues to

sample  $\overline{TA}$  on successive rising edges of CLK until  $\overline{TA}$  is recognized asserted. Only when  $\overline{TA}$  is recognized asserted is the outstanding transfer terminated.

During S3, the processor may negate  $\overline{TREQ}$  if no further transfers are pending, or may keep  $\overline{TREQ}$  asserted to indicate that another transfer is pending. The  $R/\overline{W}$ ,  $TSIZ$ ,  $TCx$  and  $ADDR[31:0]$  signals are driven with the information for the new pending cycle. If no cycle is pending, the values driven during S3 are undefined.

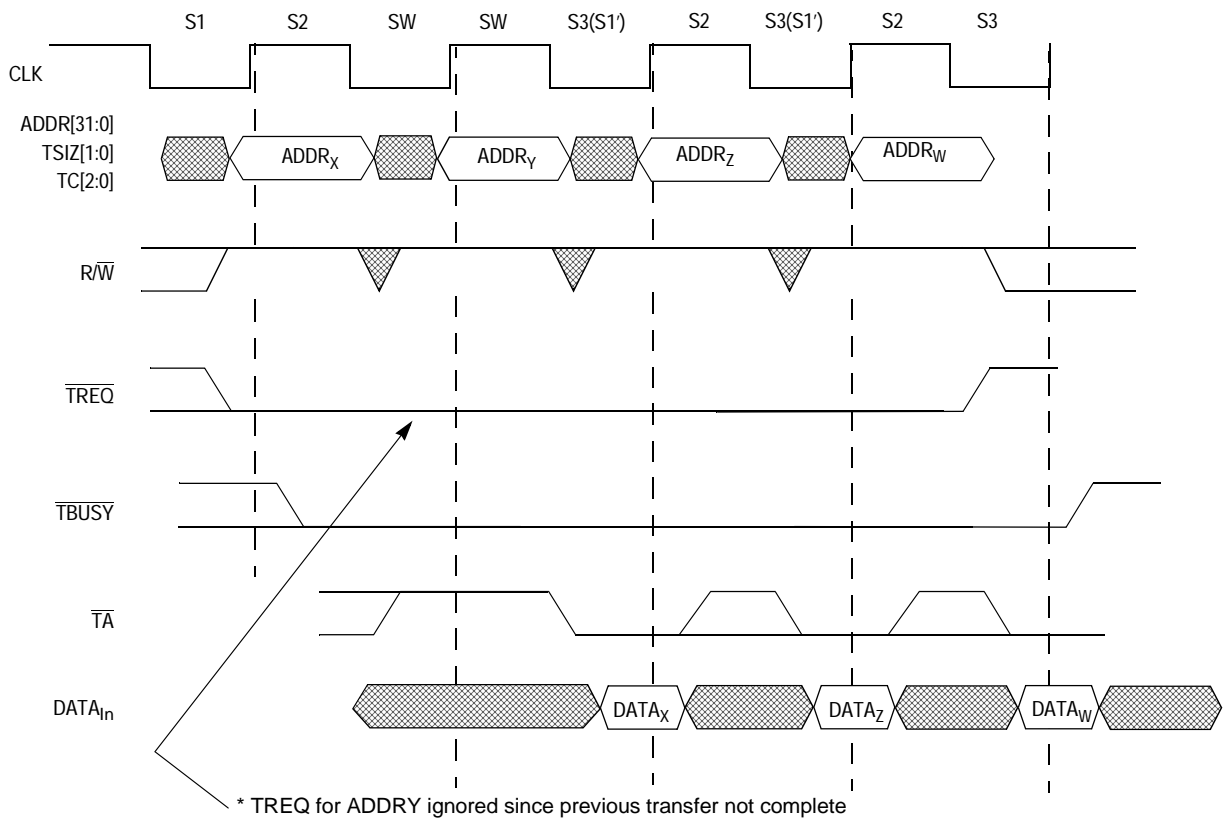
For back-to-back transfers, S3 and the next S1 occur at the same time.

### 6.5.2 Read Transfer Cycles with Wait State(s)

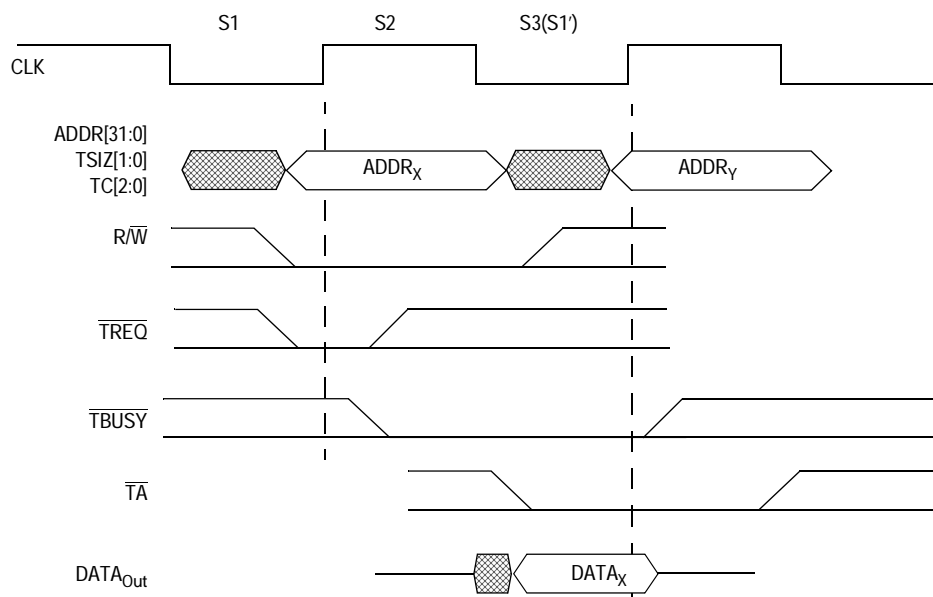
**Figure 6-4** shows an example of wait state operation.  $\overline{TA}$  for the first request ( $ADDR_x$ ) is not asserted following S2, so wait-states (Sw) are inserted until  $\overline{TA}$  is recognized. Meanwhile, another request is generated by the core for  $ADDR_y$ . This request is not considered accepted by the core since the previous transfer has not been terminated, so the processor is free to negate or change the request (in this case it changes the request to  $ADDR_z$ ) on the next cycle. This situation can occur when a data request becomes pending following an instruction prefetch request which has not been accepted, and in other circumstances. Interface control logic must be cognizant of this protocol. With a transfer in progress, the next request is considered accepted only if assertion of  $\overline{TA}$  (or  $\overline{TEA}$ ) and  $\overline{TREQ}$  occur during the same low phase of CLK.

### 6.5.3 Write Transfer Cycles

During a write transfer, the processor drives data to a memory or peripheral device. **Figure 6-5** is a functional timing diagram for write transfers.



**Figure 6-4. Read Cycle with Wait States**



**Figure 6-5. Write Cycle**

### State1 (S1)

The write cycle starts in S1. During S1, the processor places valid values on the address bus and transfer attributes. The transfer code (TCx) signals identify the specific access type. The TSIZx pins indicate the size of the transfer. The read/write ( $\overline{R/\overline{W}}$ ) signal is driven low for a write cycle.

The processor asserts transfer request ( $\overline{TREQ}$ ) during S1 to indicate that a transfer is being requested.

### State2 (S2)

The memory or device access begins in S2. The selected device uses  $\overline{R/\overline{W}}$ , TSIZ1, TSIZ0, ADDR1, and ADDR0 to select the appropriate bytes to be written in S3. The  $\overline{TBUSY}$  signal is asserted to indicate that an access is in progress.

### State3 (S3)

During S2, the processor drives the data bus with the data to be written. The interface control logic uses the values of  $\overline{R/\overline{W}}$ , TSIZ1, TSIZ0, ADDR1, and ADDR0 which were driven during S1 and S2 to align information from the data bus. With the exception of the  $\overline{R/\overline{W}}$  signal, these signals also select any or all of the operand bytes (DATA[31:24], DATA[23:16], DATA[15:8], and DATA[7:0]). If the memory responds without a wait state, then the transfer acknowledge ( $\overline{TA}$ ) signal is asserted.

The processor samples the level of  $\overline{TA}$  and if asserted, terminates the bus cycle. If  $\overline{TA}$  is not recognized as asserted at the end of the clock cycle, the processor inserts a wait state (Sw) instead of terminating the transfer. The processor continues to sample  $\overline{TA}$  on successive rising edges of CLK until  $\overline{TA}$  is recognized asserted. Only when  $\overline{TA}$  is recognized asserted is the outstanding transfer terminated.

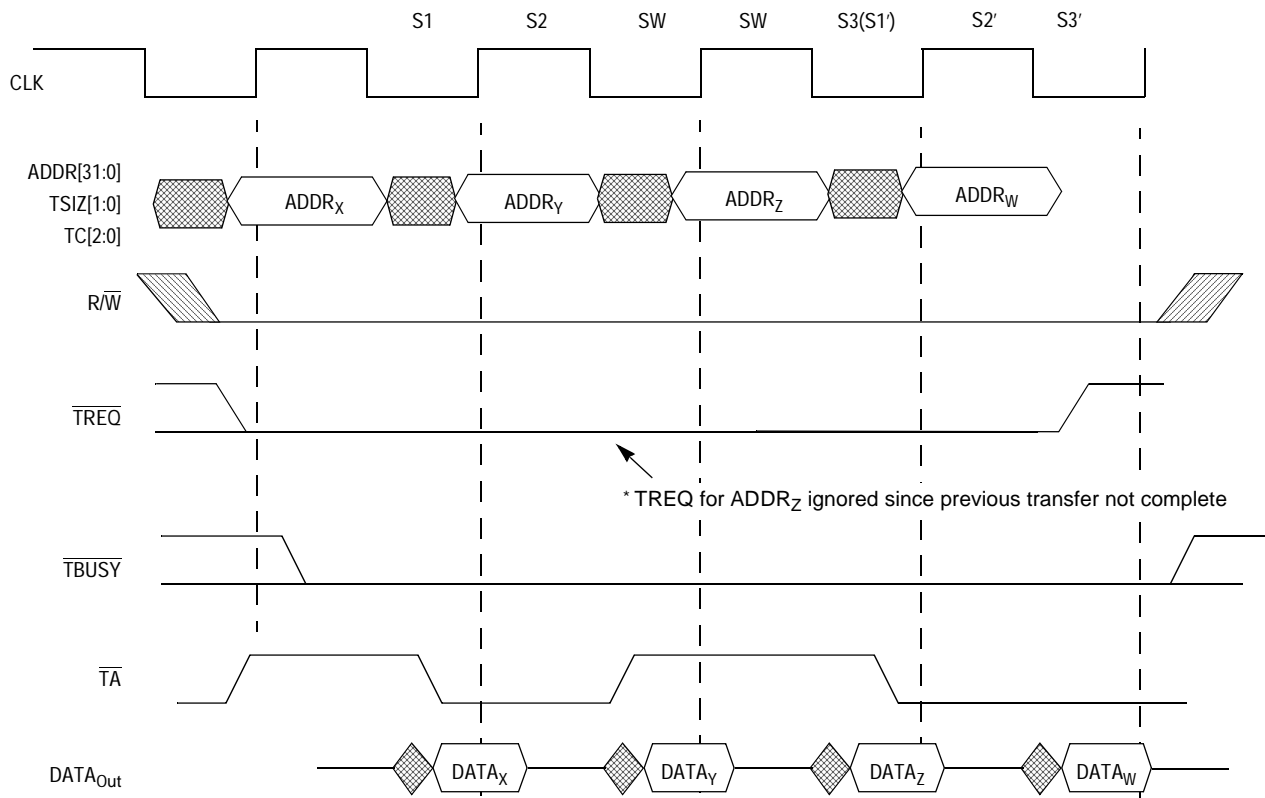
During S3, the processor may negate  $\overline{TREQ}$  if no further transfers are pending, or may keep  $\overline{TREQ}$  asserted to indicate that another transfer is pending. The  $\overline{R/\overline{W}}$ , TSIZ, TCx and ADDR[31:0] signals will be driven with the information for the new pending cycle. If no cycle is pending, the values driven during S3 are undefined.

For back-to-back transfers, S3 and the next S1 occur at the same time.

## 6.5.4 Write Transfer Cycles with Wait State(s)

**Figure 6-6** shows an example of wait state operation.  $\overline{TA}$  for the second write request ( $ADDR_Y$ ) is not asserted following  $S2$ , so wait states ( $Sw$ ) are inserted until  $\overline{TA}$  is recognized. Meanwhile, another request is generated by the core for  $ADDR_Z$ . This request is not considered accepted by the core since the previous transfer has not been terminated, so the processor is free to negate or change the request (in this case it changes the request to  $ADDR_W$ ) on the next cycle. This situation can occur when a data request becomes pending following an instruction prefetch request which has not been accepted, or in other circumstances as well. Logic controlling the interface must be cognizant of this protocol. With a transfer in progress, the next request is considered accepted only if assertion of  $\overline{TA}$  and  $\overline{TREQ}$  occur during the same low phase of  $CLK$ . In this example the request for  $ADDR_Z$  is never accepted nor should it be.

This situation can occur when a data request becomes pending following an instruction prefetch request which has not been accepted, or in other circumstances as well. Logic controlling the interface must be cognizant of this protocol. With a transfer in progress, the next request is considered accepted only if assertion of  $\overline{TA}$  and  $\overline{TREQ}$  occur during the same low phase of  $CLK$ . In this example the request for  $ADDR_Z$  is never accepted nor should it be.

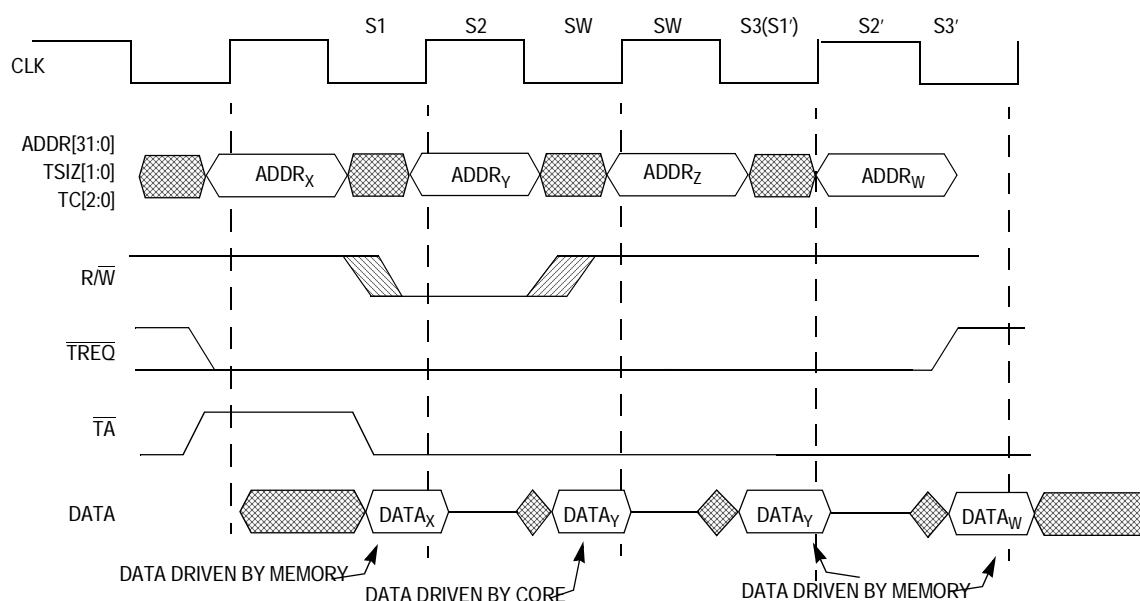


**Figure 6-6. Write Cycle with Wait States**

## 6.5.5 Data Bus Hand-Off Between Read and Write Cycles

Two examples of data bus hand-off operation are:

1. In **Figure 6-7**, the data bus is driven by either memory or the processor core during the clock low phase, and hand-off occurs during the clock high phase. When a requested access changes from a previous read to a write, hand-off is performed by three-stating the data bus drivers of memory during the clock high phase following the assertion of  $\overline{TA}$  for the outstanding read cycle. The processor is then free to drive write data on the bus during S3 (or the first Sw) for the write request. The core will only drive valid data for a single phase when an access is accepted, the memory interface is responsible for either completing the write in this phase, or latching the data.
2. **Figure 6-8** shows a read cycle with wait states followed by a write request. Although the processor has driven the address and attributes for a write cycle to  $ADDR_y$ , the data associated with the write cycle is not driven until after the write cycle has been accepted, in this case with the  $\overline{TA}$  for the  $ADDR_x$  access.



**Figure 6-7. Data Bus Hand-Off Operation**

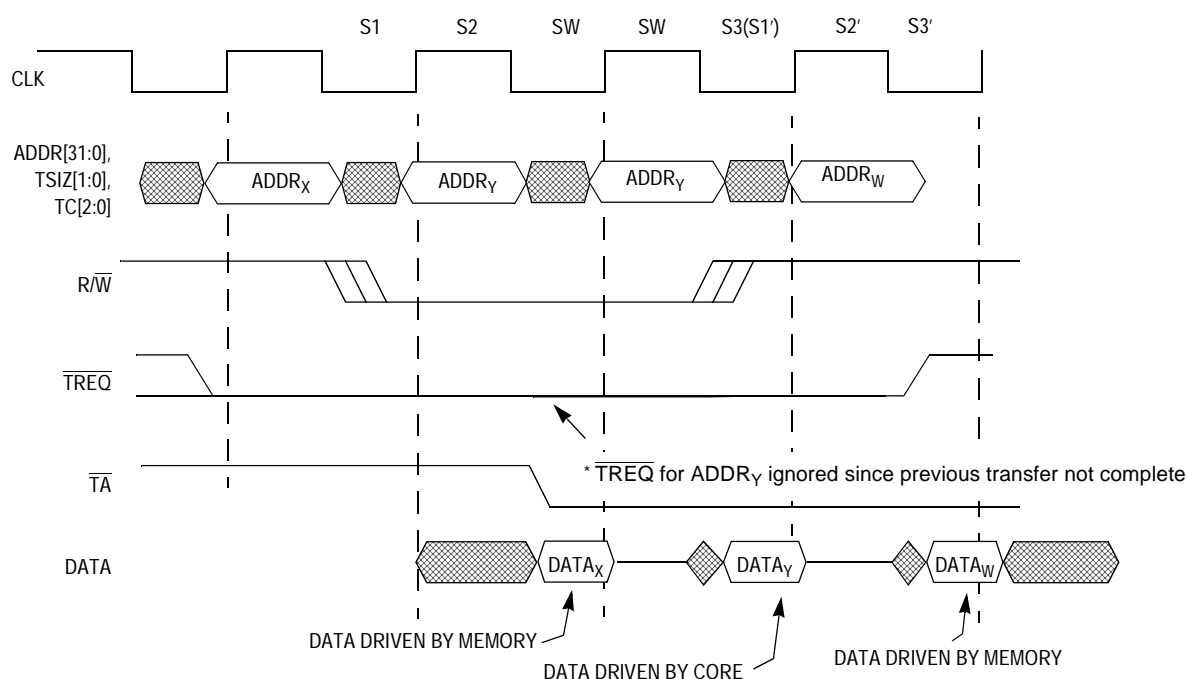


Figure 6-8. Data Bus Hand-Off Operation with Wait State

## 6.6 Exception Bus Control Cycles

The M•CORE bus interface requires assertion of  $\overline{TA}$  from an external device to signal that a bus cycle is complete. External circuitry can provide  $\overline{TEA}$  when no device responds or may indicate that an error condition is associated with an access by asserting  $\overline{TEA}$ . This allows the cycle to terminate and the processor to enter exception processing for the error condition if appropriate.

To properly control termination of a bus cycle for a bus error condition,  $\overline{TA}$  and  $\overline{TEA}$  must be asserted and negated about the same rising edge of CLK. [Table 6-2](#) is a summary of termination results.

Table 6-2. Termination Result Summary

$\overline{TA}$	$\overline{TEA}$	Result
Don't Care	Low	Bus error — Terminate, take bus error exception, if appropriate.
Low	High	Normal cycle — Terminate and continue
High	High	Insert wait states

### 6.6.1 Bus Errors

The system hardware can use the  $\overline{\text{TEA}}$  signal to abort the current bus cycle when a fault is detected. When the processor recognizes a bus error condition for an access, the access is terminated immediately.

When a bus cycle is terminated with a bus error, the M•CORE can enter access error exception processing immediately following the bus cycle, or it can defer processing the exception. The instruction prefetch mechanism requests instruction words from the instruction memory unit before it is ready to execute them. If a bus error occurs on an instruction fetch, the processor does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch, or should a task switch occur, the access error exception for the unused access does not occur. A bus error termination for any write access or read access that reference data specifically requested by the execution unit causes the processor to begin exception processing immediately. Refer to [Section 4. Exception Processing](#) for details of access error exception processing.

### 6.6.2 Breakpoint Requests

The M•CORE bus interface supports an input signal,  $\overline{\text{BRKRQ}}$ , to allow accesses to be tagged with breakpoint requests. The  $\overline{\text{BRKRQ}}$  input signal is sampled when an access is terminated with  $\overline{\text{TA}}$  or  $\overline{\text{TEA}}$  to tag an operand or instruction fetch with a breakpoint request. Operand accesses terminated with  $\overline{\text{BRKRQ}}$  asserted will result in a breakpoint exception being taken following completion of the instruction associated with the access. Instruction accesses terminated with  $\overline{\text{BRKRQ}}$  asserted will result in breakpoint processing when (and if) the instruction attempts execution. Refer to [Section 8. JTAG Test Access Port and OnCE](#) and [Section 4. Exception Processing](#) for other details on breakpoint operation.

The  $\overline{\text{BRKRQ}}$  signal does not terminate a bus cycle, it only provides status associated with a cycle. This signal must be valid when the processor recognizes a  $\overline{\text{TA}}$  or a  $\overline{\text{TEA}}$  termination. Refer to [Section 4. Exception Processing](#) for details of breakpoint exception processing.



## 6.7 $\overline{\text{ABORT}}$ Signal Operation

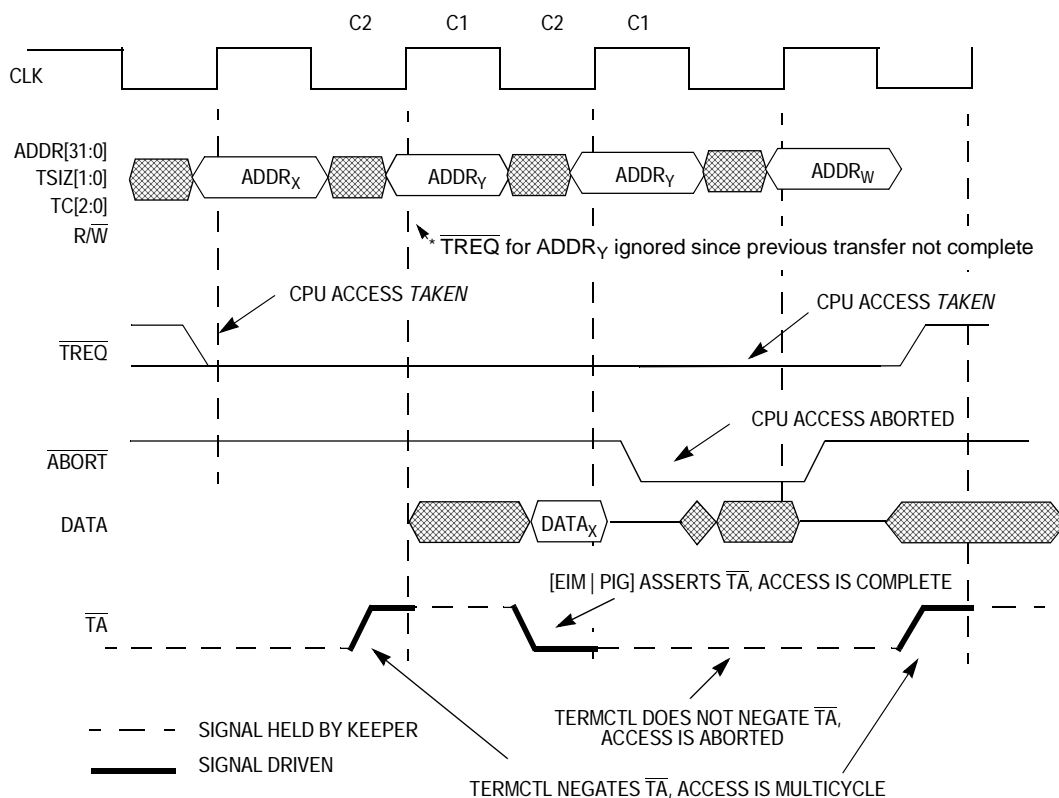
Under certain circumstances involving exception conditions, the CPU will abort an access in the clock following a valid (*taken*)  $\overline{\text{TREQ}}$  in the previous clock. In this event, the access address is an invalid one and must not be used to access devices.

Aborted accesses are indicated by the assertion by the CPU of the  $\overline{\text{ABORT}}$  output early in the clock cycle following a taken access.

Although the CPU asserts the  $\overline{\text{ABORT}}$  output, it still expects a termination signal to be asserted, and expects a no wait-state response.

**Figure 6-9** shows an example of  $\overline{\text{ABORT}}$  operation. In this example, the access for  $\text{addr}_y$  is initially stalled, then aborted.

**NOTE:** The access for  $\text{ADDR}_w$  is valid and taken, even though  $\overline{\text{ABORT}}$  has not yet negated, since it is a C1 to C1 signal.

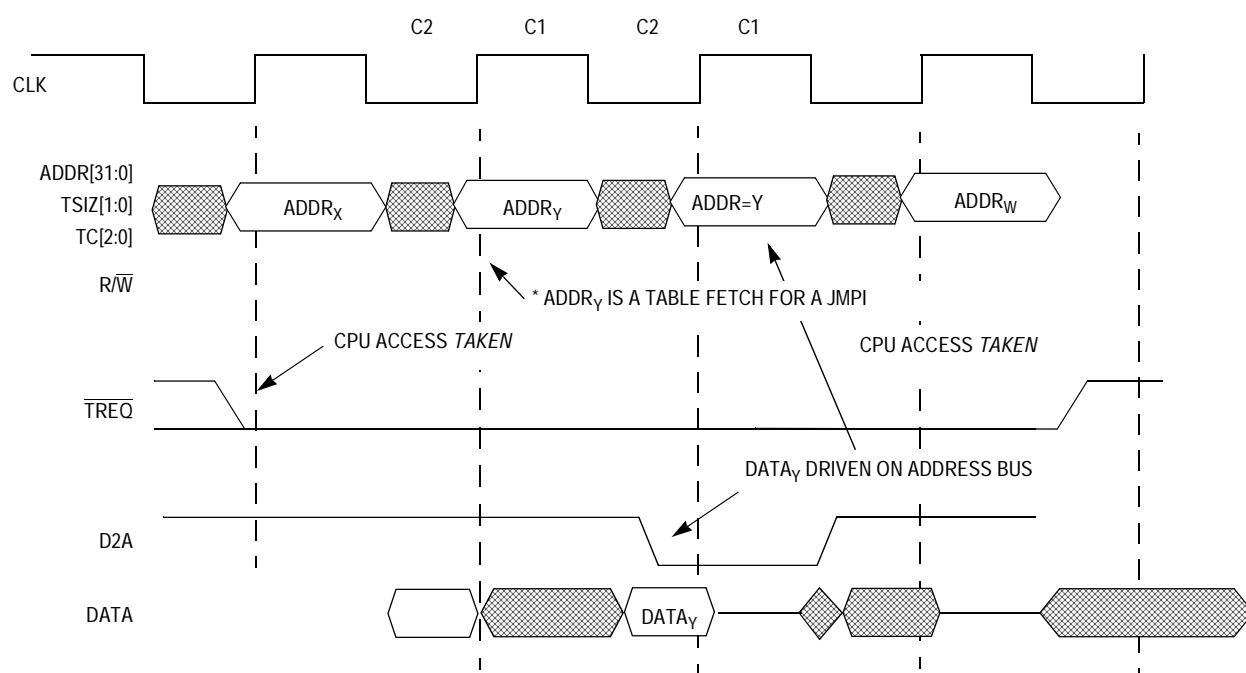


**Figure 6-9.  $\overline{\text{ABORT}}$  Operation**

## 6.8 D2A Signal Operation

Under certain circumstances involving instruction change of flow fetches, the CPU will immediately forward data received on a read access to the address bus for the following access. The D2A signal is provided to indicate this occurrence, and may be used to control system behavior as needed for these cases. Usually, no functionality need be modified, but if timing conditions require it, the signal may be used to delay termination of an access in progress, or the succeeding access.

**Figure 6-10** shows an example of the D2A signal operation. In this example, the access for ADDR<sub>Y</sub> is a table access for a JMPi (or JSRi) instruction. The returning data, DATA<sub>Y</sub>, is then forwarded to the address bus for the destination prefetch.



**Figure 6-10. D2A Operation**

## 6.9 Reset Operation

This subsection describes the reset operation.

### 6.9.1 Hard Reset (Power-On Reset)

To implement the hard reset function, an external block (typically the clock/scan control module for the chip) controls the reset of the processor. This is accomplished by disabling the CLK input and forcing the SCAN1 and SCAN2 test clocks asserted, while providing a low level on the scan data input pin. Exact details of this function will be provided at a later date as part of the definition for this external module. The reset function will be accomplished in approximately 10  $\mu$ s after the core is placed in this state. During the hard reset period, all signals that can be are driven to high impedance, and the remaining signals are driven to their inactive state. Resetting the processor causes any bus cycle in progress to be aborted. In addition, the processor initializes registers appropriately for a reset exception. After the scan clocks have been disabled and the CLK input is restored, reset exception processing is initiated. [Section 4. Exception Processing](#) describes hard reset exception processing.

### 6.9.2 Soft Reset

The soft reset ( $\overline{\text{SRST}}$ ) input signal is provided to cause a non-maskable exception to occur within the CPU to provide for system recovery from catastrophic failure. A soft reset aborts any processing in progress when  $\overline{\text{SRST}}$  is recognized; processing cannot be recovered.

The CPU will abort any activity in progress upon recognition of a soft reset, and will wait for the  $\overline{\text{SRST}}$  input to negate before beginning soft reset exception processing. [Section 4. Exception Processing](#) describes soft reset exception processing.

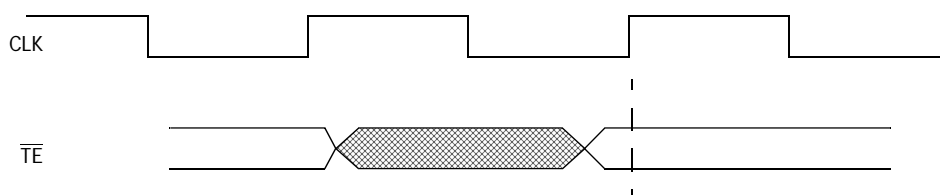
External logic is responsible for holding the  $\overline{\text{SRST}}$  input to the CPU asserted until the rest of the system environment has reached an idle state. The CPU will not wait for outstanding accesses to complete before preparing for soft reset exception processing. Any outstanding bus

access must be terminated (by external means) prior to releasing the CPU, otherwise the potential exists for the CPU to see the termination of a previously initiated access, and confuse it with termination of the first access requested after beginning reset processing.

### 6.10 Memory Management Interface Operation

The M•CORE provides a control signal to an external memory management unit, the translation control ( $\overline{TE}$ ) output. This output changes state as part of the exception recognition process as well as when the PSR is updated by the MTCR, RTE, and RFI instructions.

**Figure 6-11** shows the functional timing of this signal.



**Figure 6-11. Translation Control Output**

If no memory management signal is present in an M•CORE-based system, this output may be used to perform another control function. One possible use is as a logical address extension bit (context) which is controllable independent of the operating mode (supervisor/user) of the CPU.

### 6.11 Interrupt Interface Operation

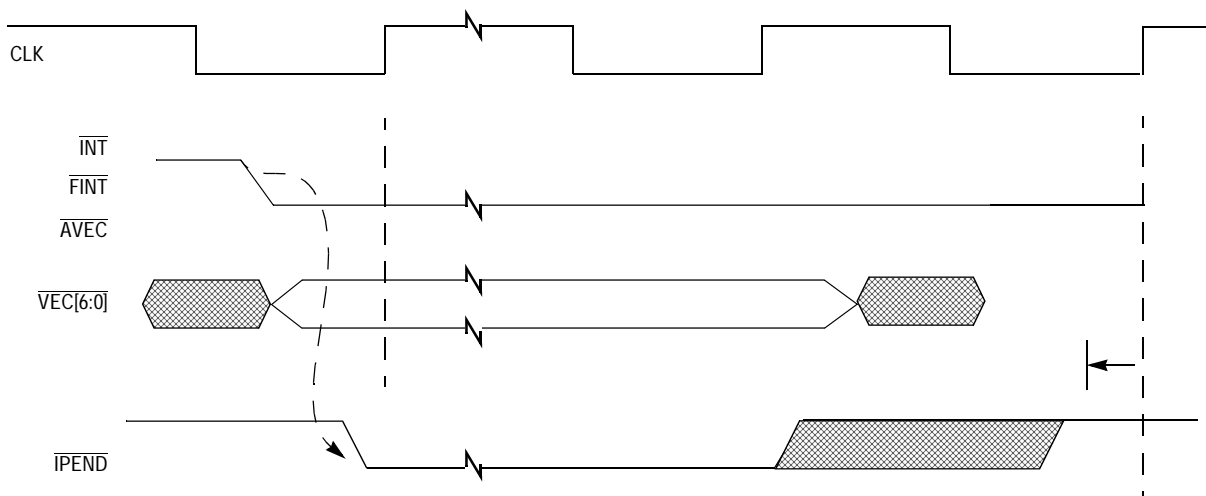
The M•CORE provides a flexible interrupt interface to an external interrupt control module.

The  $\overline{FINT}$  and  $\overline{INT}$  inputs are used to request a particular type of interrupt, and the  $\overline{AVEC}$  and  $VEC\#$  inputs are used to control the interrupt vectoring process. When the  $\overline{FINT}$  or  $\overline{INT}$  input signal is asserted, either the  $\overline{AVEC}$  or  $VEC\#$  inputs must be driven to a valid value as well, in order to properly generate the interrupt exception vector.

Interrupt inputs to the core are all level sensitive, not edge-triggered, thus the interrupt controller module must keep the interrupt request as well as keep the  $\overline{\text{VEC\#}}$  or  $\overline{\text{AVEC}}$  inputs (as appropriate) asserted until the interrupt is serviced to guarantee that the CPU core recognizes the request. On the other hand, once a request is generated, there is no guarantee the CPU will not recognize the interrupt request even if the request is later removed.

The  $\overline{\text{IPEND}}$  output can be used to control power management operation as well as assist in bus arbitration. The  $\overline{\text{IPEND}}$  output is a function of the interrupt request inputs as well as interrupt enable bits in the PSR. The interrupt input signals must meet the set up and hold requirements with respect to the rising edge of the clock, although the  $\overline{\text{IPEND}}$  output is generated combinatorially from these inputs, thus is not referenced to a clock edge. This allows it to be used as a wake-up signal to an external power management/clock generation module when the M•CORE clock input has been disabled in the **LOW** state. The  $\overline{\text{IPEND}}$  output remains asserted until the processor begins exception processing and updates the PSR to mask further interrupts. At this point the  $\overline{\text{IPEND}}$  output can negate with setup to the rising edge of the CLK.

**Figure 6-12** shows the functional timing of these signals.



**Figure 6-12. Interrupt Interface Signals**

The  $\overline{\text{IPEND}}$  output is not guaranteed to be negated if another interrupt which is not masked by recognition of the first ( $\overline{\text{FINT}}$  following  $\overline{\text{INT}}$ ) is presented to the core before the first instruction of the handler for the original interrupt is fetched and decoded, and exception processing begins again for the higher priority interrupt.

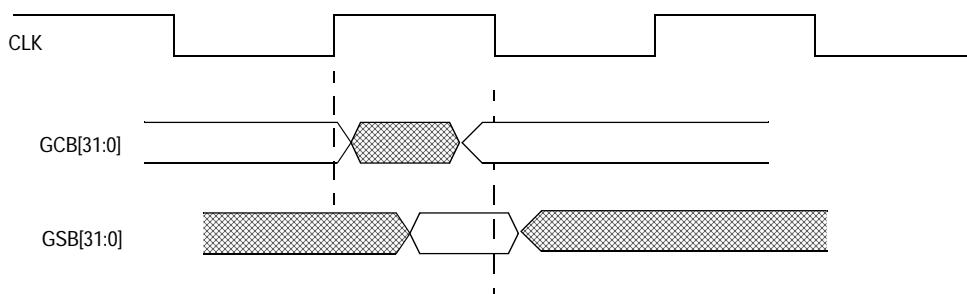
## 6.12 Global Status and Control Interface Operation

The M•CORE provides two control registers as part of the supervisor programming model to monitor global status in the integrated system as well as to provide global control outputs to the system. Refer to [Section 2. Registers](#) for more information on these registers.

The GCB[31:0] outputs change state when the GCR register is updated by the MTCR instruction.

The GSB[31:0] inputs are sampled by the core and the corresponding values appear in the GSR for transfer to a general register when an MFCR instruction referencing the GSR is executed.

**Figure 6-13** shows the functional timing of these signals. The GSB[31:0] inputs are sampled with the falling edge of the CLK, the GCB[31:0] outputs transition following the rising edge of the CLK.



**Figure 6-13. Global Status and Control Signals**

## 6.13 Power Management Interface Operation

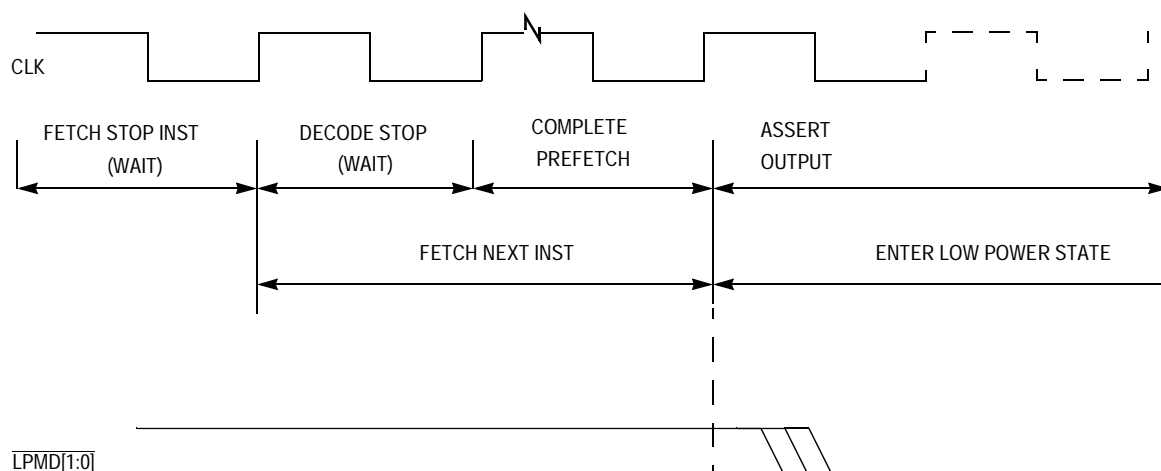
The M•CORE provides three instructions, DOZE, STOP, and WAIT, to enter low-power operating modes. The functionality of these modes is not dictated by the core, but is determined by the design of an external power management module. The M•CORE provides output signals associated with the execution of each of these instructions that may be monitored by external logic to control operation of the core as well as the rest of the system.

When a DOZE, STOP, or WAIT instruction is executed, the appropriate mode is indicated by the core on the  $\overline{\text{LPMD}}[1:0]$  outputs. External logic may then place the core in a low-power state by forcing the CLK input low. The core may be re-enabled by providing the CLK input as system events dictate. Completion of the DOZE, STOP, or WAIT instruction requires recognition of a valid interrupt, and the assertion of the  $\overline{\text{IPEND}}$  output. The processor will remain in a stopped or waiting state until a valid interrupt is pending and the CLK input has been re-enabled.

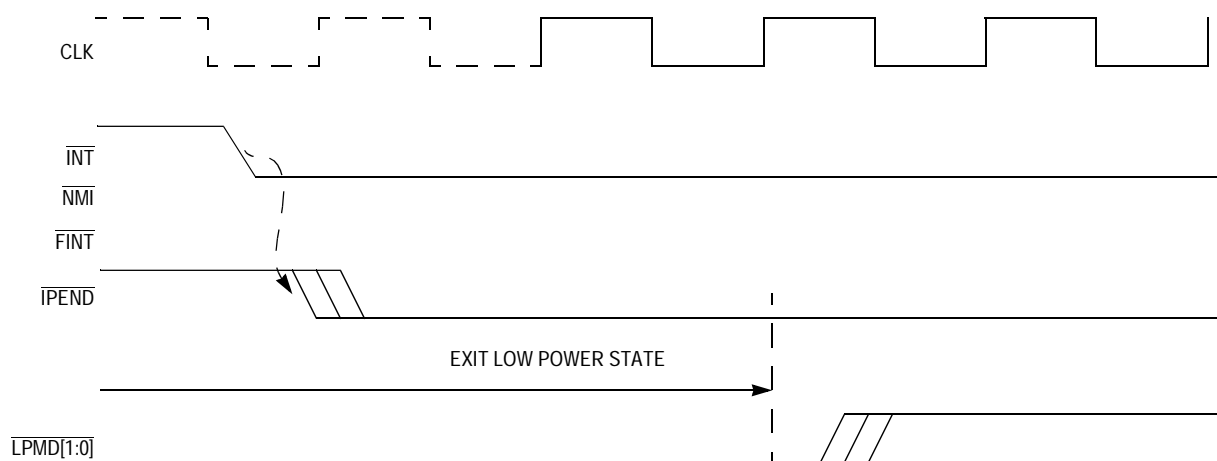
Execution of the DOZE, STOP, or WAIT instruction will be held off until any outstanding prefetch has completed.

**Figure 6-14** and **Figure 6-15** show the functional timing of these signals. The  $\overline{\text{LPMD}}[1:0]$  outputs transition following the rising edge of the CLK after all outstanding fetches have been completed.

Refer to **6.11 Interrupt Interface Operation** for more information on interrupt recognition while in a stopped or waiting state.



**Figure 6-14. Power Management Control Signals (Assertion)**



**Figure 6-15. Power Management Control Signals (Negation)**

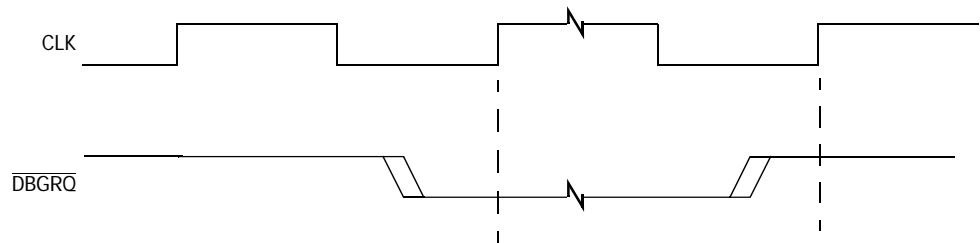


## 6.14 Emulation/Debug Interface Operation

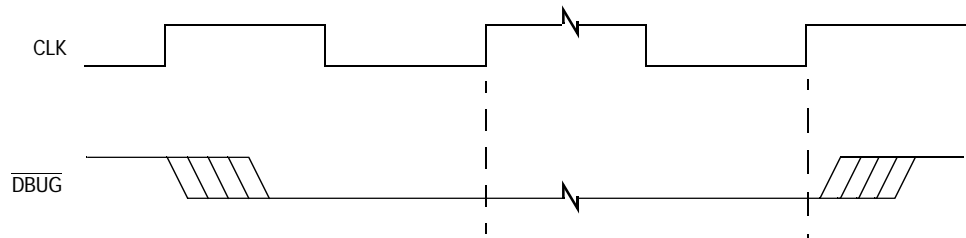
The M•CORE provides an input signal for use by an external emulation/debug module which allows execution of the core to be controlled. The  $\overline{\text{DBGRQ}}$  input is used to force the CPU to enter debug mode at the next instruction boundary.

A dedicated output signal is provided to handshake these requests, the  $\overline{\text{DEBUG}}$  active low output. Refer to [Section 8. JTAG Test Access Port and OnCE](#) for more information on the use of these signals.

[Figure 6-16](#) and [Figure 6-17](#) show the functional timing of these signals. The  $\overline{\text{DBGRQ}}$  input is sampled with the rising edge of the CLK. The  $\overline{\text{DEBUG}}$  output transitions following the rising edge of the clock once the debug state has been entered.



**Figure 6-16. Debug Request Input Control Signal**



**Figure 6-17. Debug Output Control Signal**



## Section 7. Hardware Accelerator Interface (HAI)

### 7.1 Contents

7.2	Introduction . . . . .	220
7.3	Overview . . . . .	220
7.4	Register Snooping Mechanism . . . . .	221
7.5	Instruction Transfer Mechanism . . . . .	222
7.5.1	Control Handshake . . . . .	222
7.5.2	Driving the $\overline{\text{H\_BUSY}}$ and $\overline{\text{H\_EXCP}}$ Signals . . . . .	229
7.6	Data Transfer Mechanism . . . . .	230
7.6.1	Register Transfers . . . . .	230
7.6.2	Memory Transfers . . . . .	233
7.6.2.1	H_LD Transfer . . . . .	233
7.6.2.2	H_ST Transfer . . . . .	234
7.7	Instruction Primitives . . . . .	238
7.7.1	H_CALL Primitive . . . . .	238
7.7.2	H_RET Primitive . . . . .	239
7.7.3	H_LD Primitive . . . . .	239
7.7.4	H_ST Primitive . . . . .	240
7.7.5	H_EXEC Primitive . . . . .	241
7.8	Instruction Primitive Glossary . . . . .	241

**NOTE:** This feature is not offered for the M210/M210S core.

### 7.2 Introduction

The M•CORE hardware accelerator interface (HAI) supports tightly coupled hardware function blocks which are optimized for application specific purposes. An example block might be a DSP arithmetic block (MAC) unit, other examples are simpler blocks such as a population count block (POPC). This section provides a functional description of the interface, the signals that control the interface, and the cycles provided for data transfer operations. The primitives available through the processor core ISA are also described.

Accelerator functions are designed to be implementation specific units, thus the exact functionality of a given unit is free to be changed across different implementations, even though the same instruction mappings may be present.

### 7.3 Overview

The M•CORE provides support for task acceleration by an external hardware block which is optimized for specific application-related operations. These external blocks may be as simple as a block for performing a population count, or a more complicated function such as a DSP acceleration block capable of high speed multiply/accumulate operation.

Data is transferred between the core and an accelerator block by one or more of several mechanisms as appropriate for a particular implementation. These can be divided into transfers to the block and transfers from the block.

One method of transferring data to a block is the register snooping mechanism, which involves no instruction primitive, but is a by-product of normal core operation. This involves reflecting updates to the core's general-purpose registers across the interface such that a block could monitor updates to one or more core registers. This might be appropriate if a block “overlays” a GPR for an internal register or function. In this case, no explicit passing of parameters from the core to a block would be required.

Instruction primitives are provided in the base core for explicit transfer of operands and instructions between external accelerators and the core as well. A handshaking mechanism is provided to allow control over the rate of instruction and data transfer.

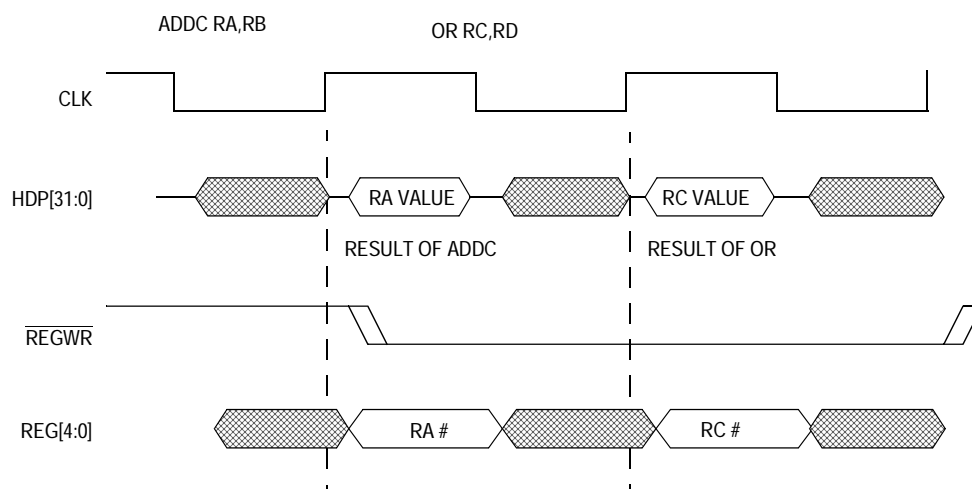
**NOTE:** *Accelerator functions are designed to be implementation specific units, thus the exact functionality of a given unit is free to be changed across different implementations, even though the same instruction mappings may be present.*

## 7.4 Register Snooping Mechanism

To avoid the performance overhead of parameter passing to a hardware block, a register snooping mechanism is provided. This allows a hardware block to implement a shadow copy of one or more of the core's general registers. The capability is implemented by transferring the value being written into one of the core GPRs and an indication of which register is being updated for each GPR update. A strobe signal  $\overline{\text{REGWR}}$  is asserted for each register update. The value is transferred across the 32-bit bidirectional data path HDP[31:0], and a 5-bit register number bus provides a pointer to the actual core register being updated (REG[4:0]). The register number may refer to a register in the normal file or in the alternate file. Alternate file registers are indicated by REG4 == 1, normal file registers by REG4 == 0. Refer to [Section 2. Registers](#) for a description of these registers.

The hardware block may latch the value internally along with an indication of the destination register number to avoid an explicit move later. This functionality may also be used by a debug block to track the state of the register file or a subset of it.

**Figure 7-1** shows an example of the snooping capability. Caveats to using this mechanism in the presence of exceptions are discussed later.



**Figure 7-1. Register Snoop Operation**

## 7.5 Instruction Transfer Mechanism

A dedicated 12-bit instruction bus (H\_OP[11:0]) provides the HAI opcode being issued to the external block. This bus reflects the low order 12 bits of the M•CORE opcode. The high-order four bits are not reflected as they are always 0b0100. A supervisor mode indicator (H\_SUP) is also provided to indicate the current state of the PSR(S) bit, indicating whether the processor is operating in supervisor or user mode. A set of handshake signals between the core and external accelerator blocks coordinate HAI instruction execution.

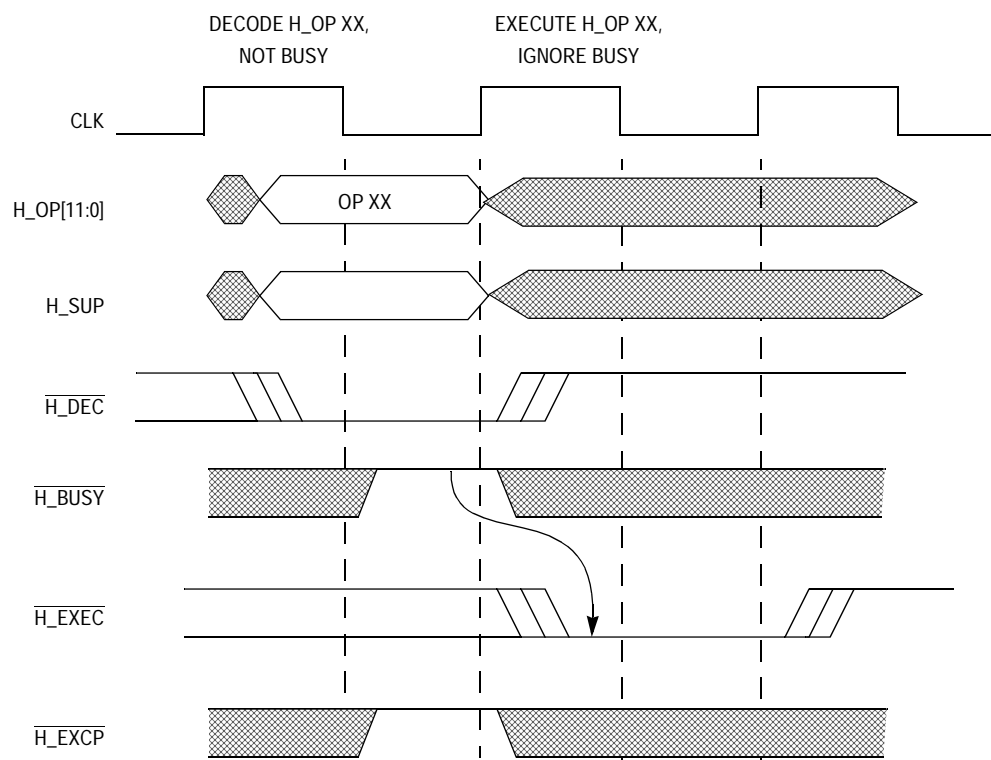
### 7.5.1 Control Handshake

The control signals generated by the core are a reflection of the internal pipeline structure of the processor. The processor pipeline consists of stages for instruction fetch, instruction decode, execution, and result writeback. The processor also contains an instruction prefetch buffer to allow buffering of an instruction prior to the decode stage. Instructions proceed from this buffer to the instruction decode stage by entering the instruction decode register IR.

The instruction decoder receives inputs from the IR, and generates outputs based on the value held in the IR. These decode outputs are not

always valid, and may be discarded due to exception conditions or changes in instruction flow. Even when valid, instructions may be held in the IR until they can proceed to the execute stage of the instruction pipeline. Since this cannot occur until previous instructions have completed execution (which may take multiple clocks), the decoder will continue to decode the value contained in the IR until the IR is updated.

**Figure 7-2** shows the basic instruction interface operation for instruction handshaking.



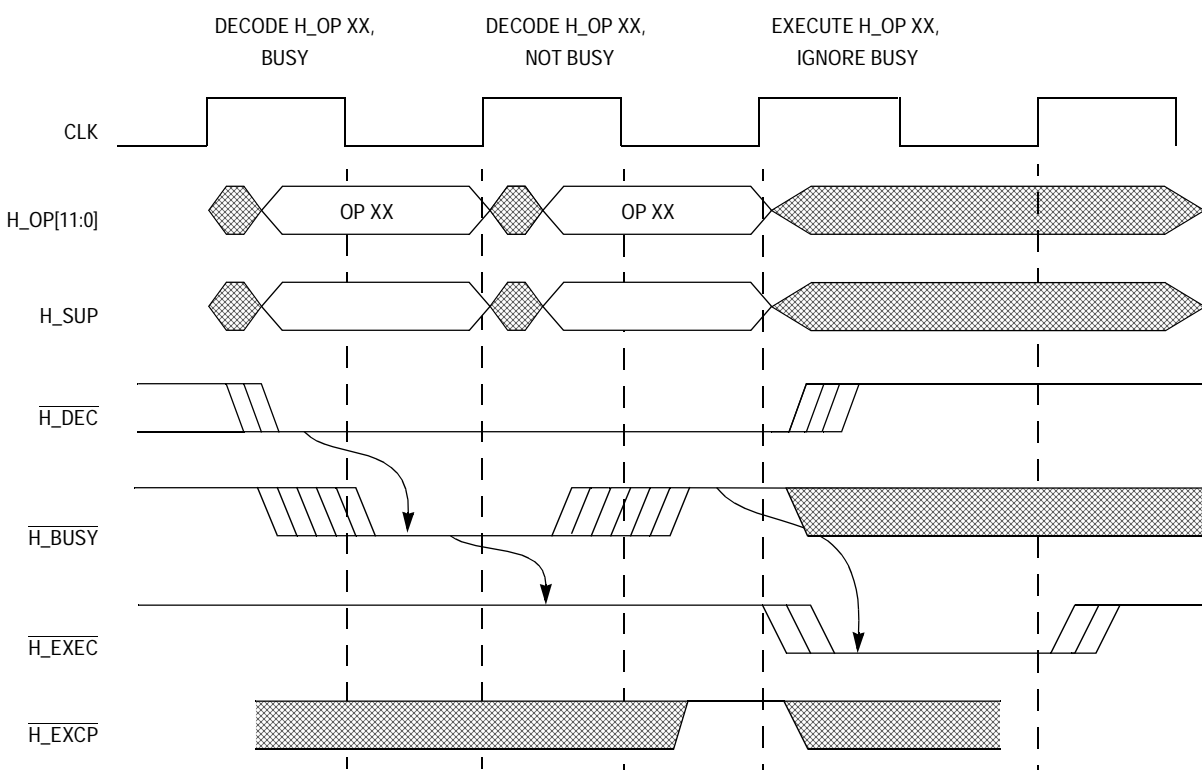
**Figure 7-2. Basic Instruction Interface Operation,  $\overline{H\_BUSY}$  Negated**

An instruction decode strobe ( $\overline{H\_DEC}$ ) signal is provided to indicate the decode of an HAI opcode by the core. This signal will be asserted when an HAI opcode resides in the IR, even if the instruction may be discarded without execution. The  $\overline{H\_DEC}$  output may remain asserted for multiple clocks for the same instruction until the instruction is actually issued or is discarded.

## Hardware Accelerator Interface (HAI)

A busy signal ( $\overline{H\_BUSY}$ ) is monitored by the core to determine if an external block can accept the HAI instruction, and partially controls when issuance of the instruction occurs. If the  $\overline{H\_BUSY}$  signal is negated while  $\overline{H\_DEC}$  is asserted, instruction execution will not be stalled by the interface, and the  $\overline{H\_EXEC}$  signal may assert as soon as instruction execution can proceed. If the  $\overline{H\_BUSY}$  signal is asserted when the core decodes an HAI opcode (indicated by the assertion of  $\overline{H\_DEC}$ ), execution of the HAI opcode will be forced to stall. Once the  $\overline{H\_BUSY}$  signal is negated, the core may issue the instruction by asserting  $\overline{H\_EXEC}$ . If a hardware block is capable of buffering instructions, the  $\overline{H\_BUSY}$  signal may be used to assist filling of the buffer.

**Figure 7-3** shows the instruction interface operation when  $\overline{H\_BUSY}$  is used to control HAI instruction execution.

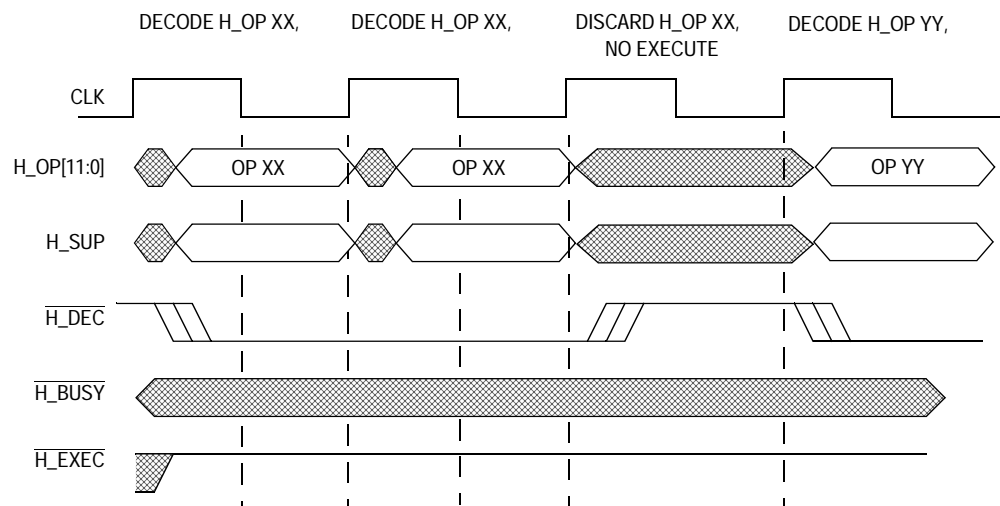


**Figure 7-3. Basic Instruction Interface Operation,  $\overline{H\_BUSY}$  Asserted**



Once any internal stall condition has been resolved, and the  $\overline{H\_BUSY}$  signal has been negated, the processor can assert  $\overline{H\_EXEC}$  to indicate that the HAI instruction has entered the execute stage of the pipeline. An external block should monitor the  $\overline{H\_EXEC}$  signal to control actual execution of the instruction, since it is possible for the processor to discard the instruction prior to execution in certain circumstances. If execution of an earlier instruction results in an exception being taken, the  $\overline{H\_EXEC}$  signal will not be asserted, and the  $\overline{H\_DEC}$  output will be negated. A similar process can occur if the instruction in the IR is discarded as the result of a change in program flow.

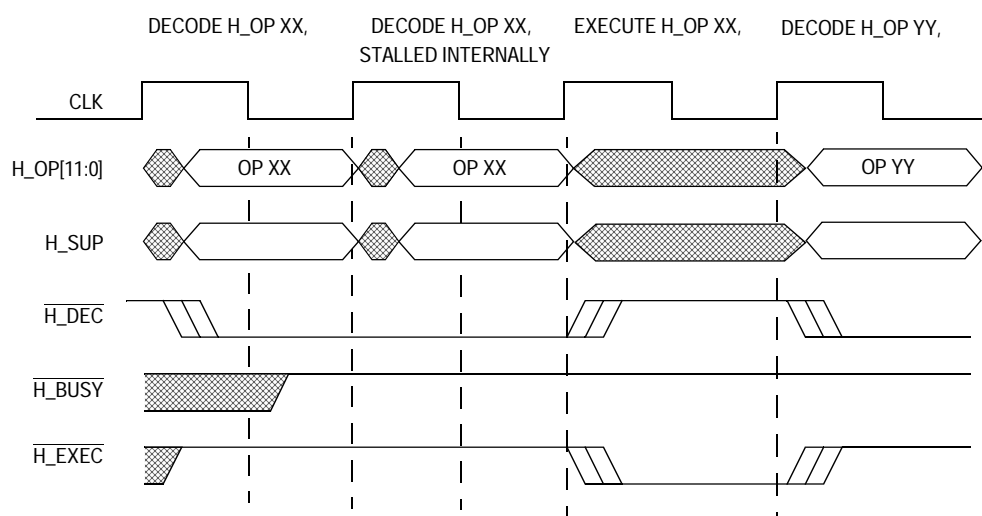
If an instruction is discarded, the  $\overline{H\_DEC}$  signal will be negated before another HAI opcode is placed on the H\_OP bus. [Figure 7-4](#) shows an example of this.



**Figure 7-4. Instruction Discard**

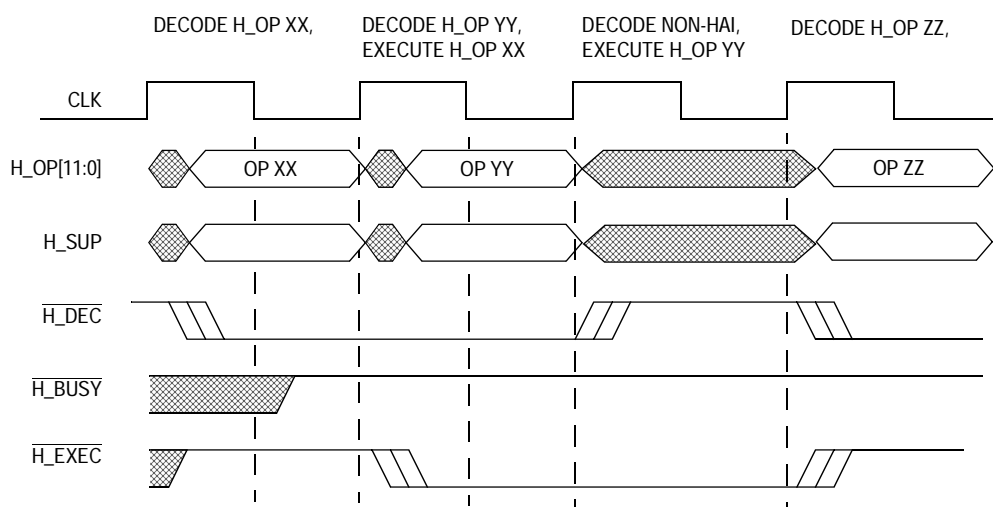
There are circumstances where the core may delay the assertion of  $\overline{H\_EXEC}$  even though  $\overline{H\_DEC}$  is asserted and  $\overline{H\_BUSY}$  is negated. This can occur while waiting for an earlier instruction to complete (see [Figure 7-5](#)).

For back-to-back HAI instructions, the  $\overline{H\_DEC}$  signal can remain asserted without negation, even though the H\_OP bus is updated as new instructions enter the IR.



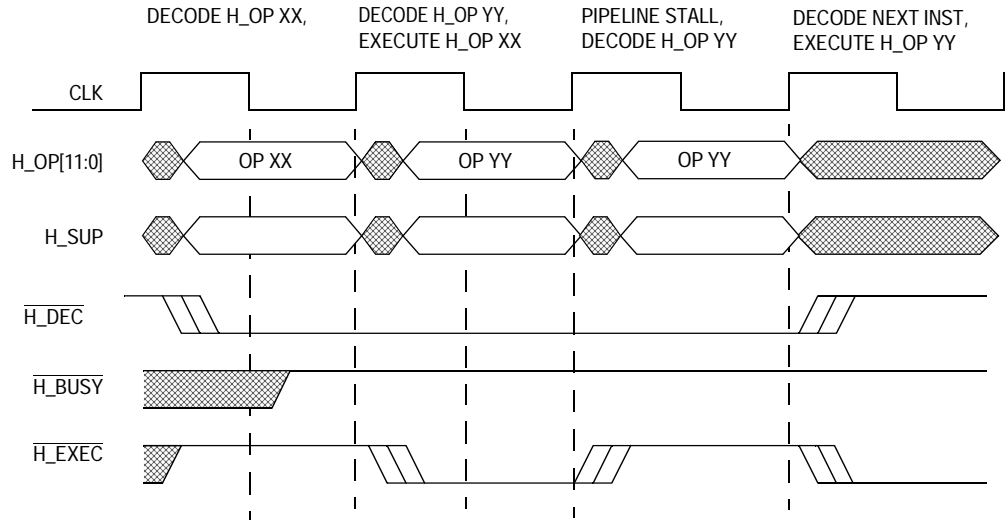
**Figure 7-5. Instruction Pipeline Stall**

**Figure 7-6** shows example of back-to-back execution with no stalls. In general, the assertion of  $\overline{H\_EXEC}$  corresponds to execution of the instruction being decoded on the previous clock.



**Figure 7-6. Back-to-Back HAI Instruction Execution**

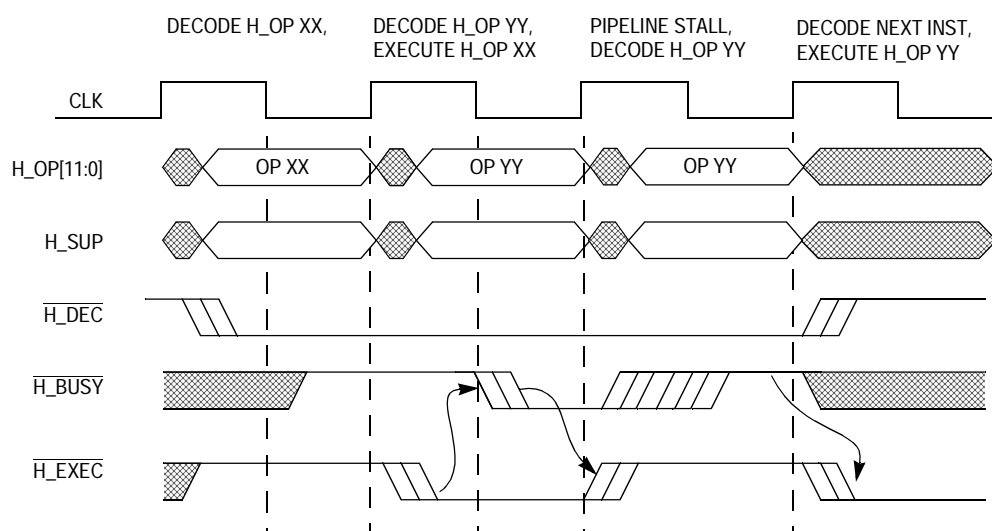
**Figure 7-7** shows back-to-back operation with internal pipeline stalls. In this case,  $\overline{H\_BUSY}$  is negated, but the processor does not assert  $\overline{H\_EXEC}$  for the second HAI instruction until the internal stall condition disappears.



**Figure 7-7. Back-to-Back HAI Instruction Execution with Pipeline Stall**

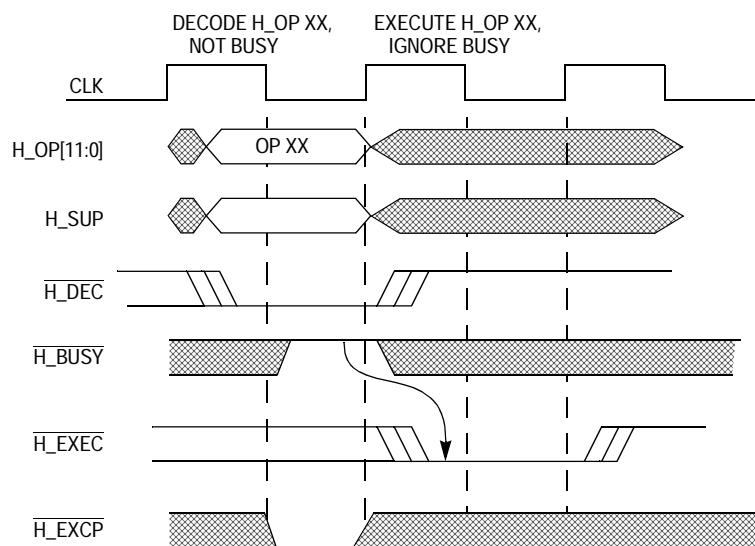
**Figure 7-8** shows back-to-back HAI instructions with  $\overline{H\_BUSY}$  stalls. In this example, the external block is busy, and cannot accept the second instruction immediately.  $\overline{H\_BUSY}$  asserts to prevent the second instruction from being issued by the core. Once the accelerator becomes free,  $\overline{H\_BUSY}$  is negated, and the next HAI instruction advances to the execute stage.

Exceptions related to the decode of an HAI opcode may be signaled by an external block with the  $\overline{H\_EXCP}$  signal. This input to the core is sampled during the clock cycle that  $\overline{H\_DEC}$  is asserted and  $\overline{H\_BUSY}$  is negated, and will result in exception processing for a hardware accelerator exception if the HAI opcode is not discarded as previously described. Details of this exception processing are described in **Section 4. Exception Processing**.



**Figure 7-8. Back-to-Back HAI Instruction Execution with  $\overline{H\_BUSY}$  Stall**

**Figure 7-9** shows an example of the  $\overline{H\_EXCP}$  signal being asserted by an accelerator block in response to the decode and attempted execution of an HAI opcode. The  $\overline{H\_EXCP}$  signal is sampled by the core during the clock that  $\overline{H\_DEC}$  is asserted. The  $\overline{H\_EXEC}$  signal is asserted regardless of whether an exception is signaled by the interface; this distinguishes the exception taken case from the instruction discard case.



**Figure 7-9.  $\overline{H\_EXCP}$  Operation,  $\overline{H\_BUSY}$  Negated**

**NOTE:** The exception corresponds to the instruction being decoded the previous clock cycle, and that no actual execution takes place. An accelerator block must accept the offending instruction and signal an exception prior to the execute stage of the processor pipeline for it to be recognized. The  $\overline{H\_EXCP}$  signal is ignored for all clock cycles where  $\overline{H\_DEC}$  is negated or  $\overline{H\_BUSY}$  is asserted.

Figure 7-10 shows an example where  $\overline{H\_BUSY}$  has been asserted to delay the execution of an HAI opcode which will result in an exception.

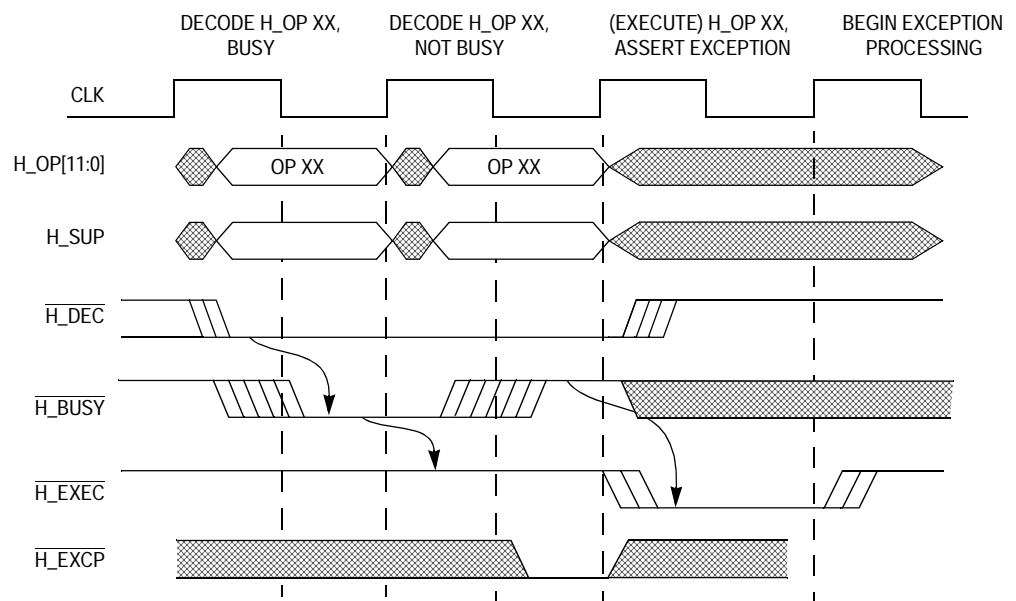


Figure 7-10.  $\overline{H\_EXCP}$  Operation, HAI Busy

## 7.5.2 Driving the $\overline{H\_BUSY}$ and $\overline{H\_EXCP}$ Signals

The  $\overline{H\_BUSY}$  and  $\overline{H\_EXCP}$  signals are shared by all accelerator blocks, thus they must be driven in a coordinated manner.

These signals should be driven (either high or low, whichever is appropriate) by the hardware unit corresponding to H\_OP[11:10] on clock cycles where  $\overline{H\_DEC}$  is asserted. By driving the output only during the low portion of the clock, these signals may be shared by multiple blocks without contention. A holding latch internal to the processor is

provided on this input to hold it in a valid state for the high phase of the clock while no unit is driving it.

### 7.6 Data Transfer Mechanism

Some of the HAI instruction primitives also imply a transfer of data items between the core and an external block. Operands may be transferred across the interface as a function of the particular primitive being executed.

Provisions are made for transferring one or more of the core GPRs either to or from an accelerator block across a 32-bit bidirectional data path. In addition, provisions are also made to load or store a single data item from/to memory with the data sink/source being the interface. The core passes parameters to external blocks via the HDP[31:0] bus during the high portion of CLK; operands are received and latched from the interface by the core during the low phase of the clock. A delay is provided as the clock transitions high before drive occurs to allow for a small period of bus hand off. A block interface must provide the same small delay at the falling clock edge. Handshaking of data items is supported with the data strobe ( $\overline{\text{H\_DS}}$ ) output, the data acknowledge ( $\overline{\text{H\_DA}}$ ) input, and the data error ( $\overline{\text{H\_DERR}}$ ) output signals.

#### 7.6.1 Register Transfers

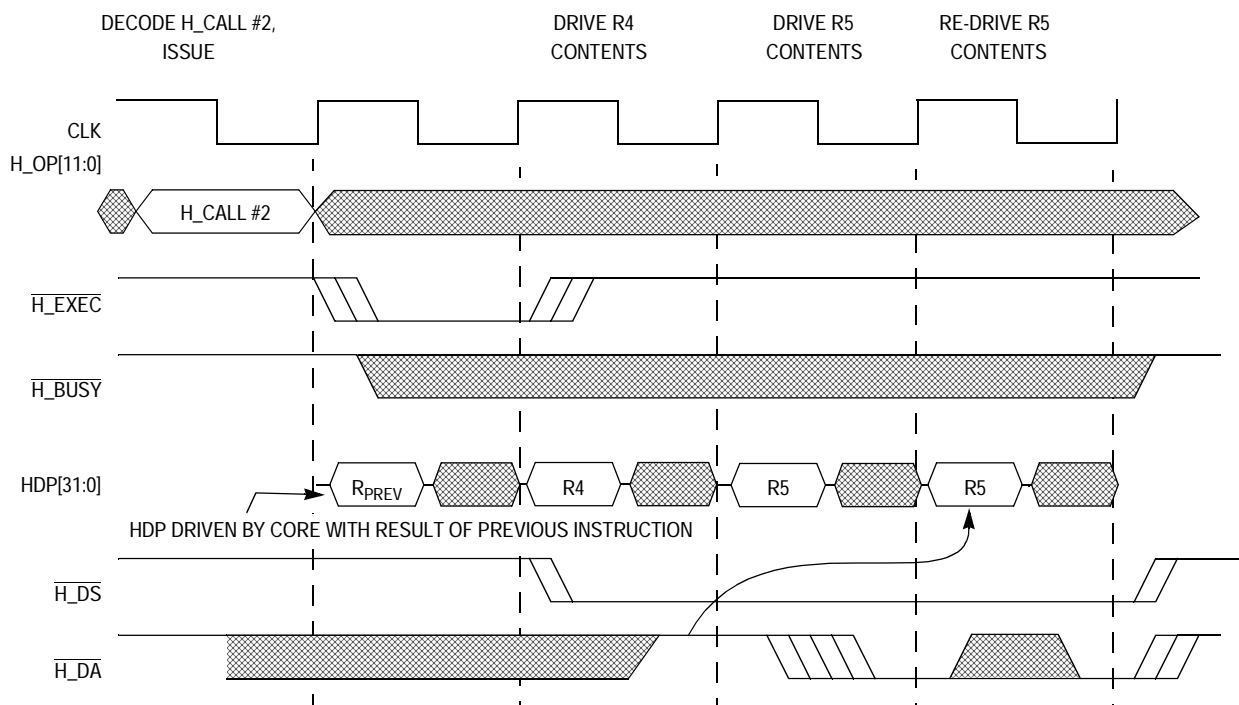
The core provides the capability of transferring a list of call or return parameters to the interface in much the same way as software subroutines are called or returned from. A count of arguments is indicated in the H\_CALL or H\_RET primitive to control the number of parameters passed. Register values beginning with the content of R4 are transferred to (from) the external hardware block as part of the execution of the H\_CALL (H\_RET) primitive. Up to seven register parameters may be passed. This convention is similar to the software subroutine calling convention (see [Section 2. Registers](#)).

Handshaking of the operand transfers are controlled by the data strobe ( $\overline{\text{H\_DS}}$ ) output and data acknowledge ( $\overline{\text{H\_DA}}$ ) input signals. Data strobe will be asserted by the core for the duration of the transfers, and

transfers will occur in an overlapped manner, much the same as the core interface operation. Data acknowledge ( $\overline{H\_DA}$ ) is used to indicate that a data element has been accepted or driven by a hardware block.

Instruction primitives are provided to transfer multiple processor registers and the transfers can ideally occur every clock. For transfers to an external block, the processor will automatically begin driving the next operand (if needed) prior to (or concurrent with) the acknowledge of the current item. External logic must be capable of one level of buffering to ensure no loss of data. **Figure 7-11** shows the sequencing of an H\_CALL transfer to the interface, where two registers are to be transferred. The second transfer is repeated due to a negated data acknowledge ( $\overline{H\_DA}$ ).

For transfers from an external block to processor registers, the processor is capable of accepting values from an external block every clock cycle after  $\overline{H\_DS}$  has been asserted, and these values are written into the register file as they are received, so no buffering is required.



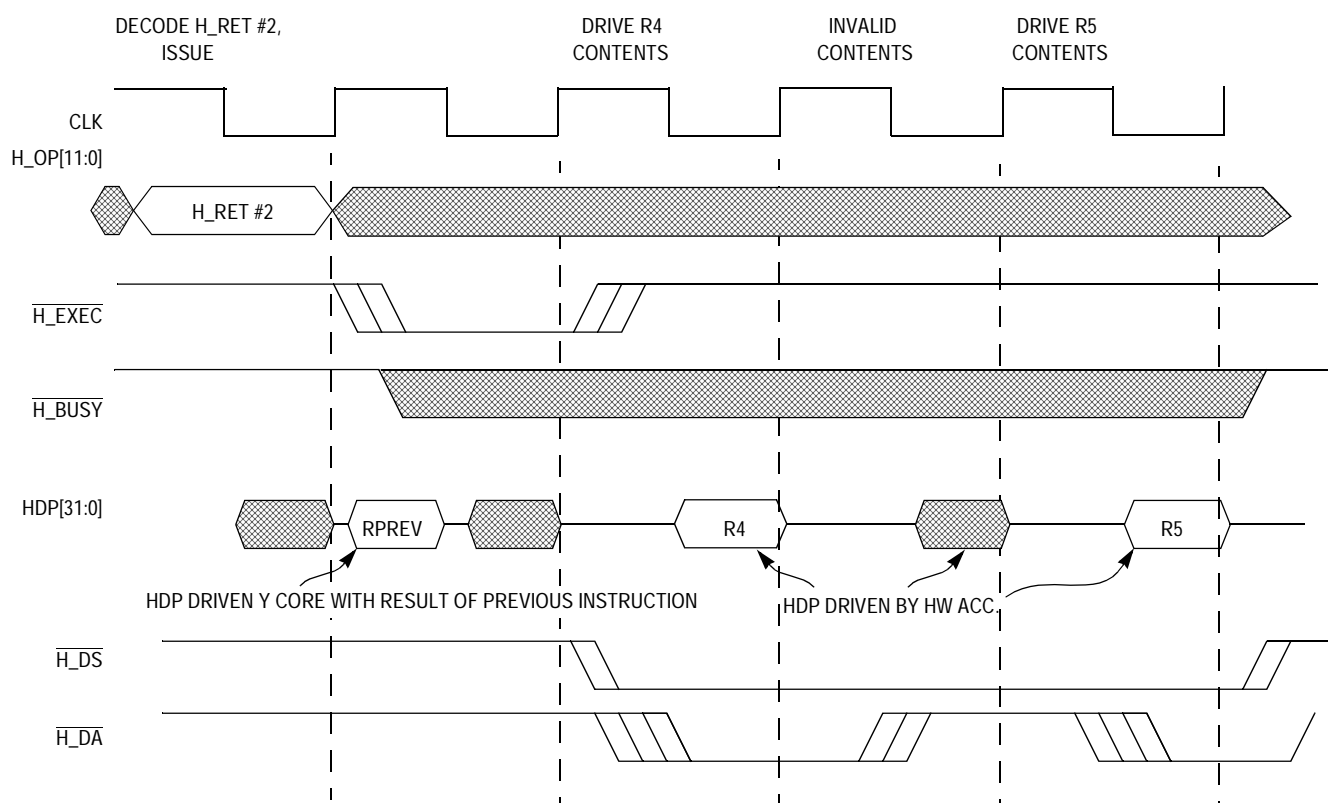
**Figure 7-11** Register Transfers to External Block with Wait State

## Hardware Accelerator Interface (HAI)

An example of register transfers associated with the H\_RET primitive is shown in **Figure 7-12**.

In this example, two register values are transferred. The block may drive data beginning with the clock following the assertion of the  $\overline{H\_EXEC}$  signal, as this is the clock where  $\overline{H\_DS}$  will first be asserted.

The  $\overline{H\_DS}$  output transitions with the rising edge of CLK, while the  $\overline{H\_DA}$  input is sampled during the low phase of CLK. Refer to xxx for more detail on the actual timing of these signals.



**Figure 7-12. Register Transfers from External Block with Wait State**



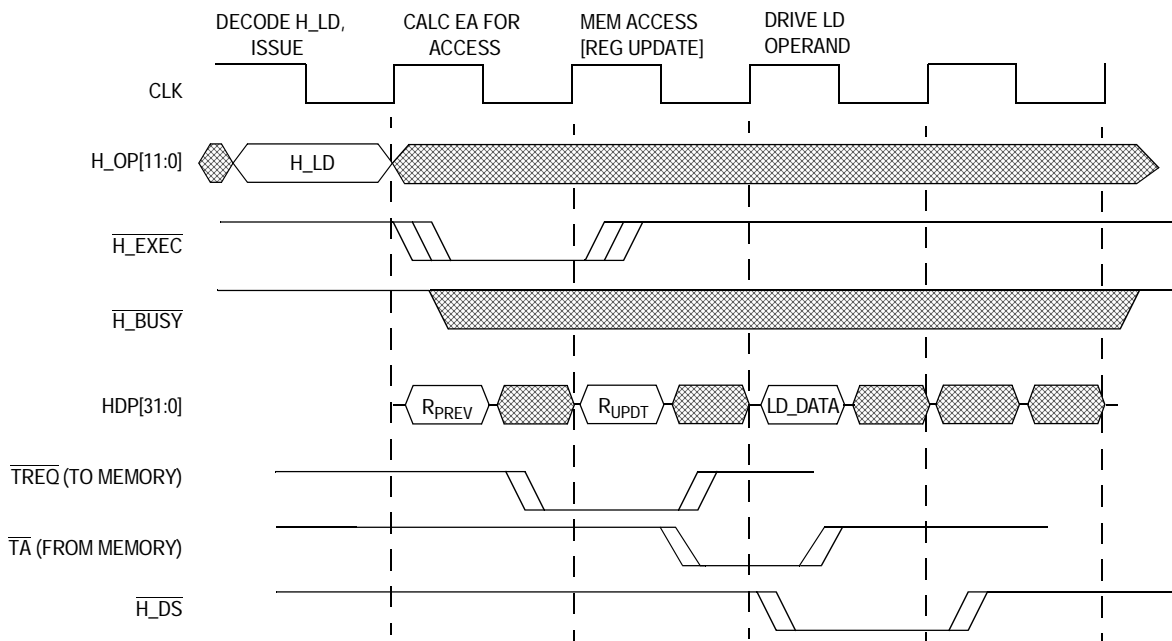
## 7.6.2 Memory Transfers

The core provides the capability of transferring a single memory operand to or from the interface with the **h\_ld** or **h\_st** instruction primitives.

### 7.6.2.1 H\_LD Transfer

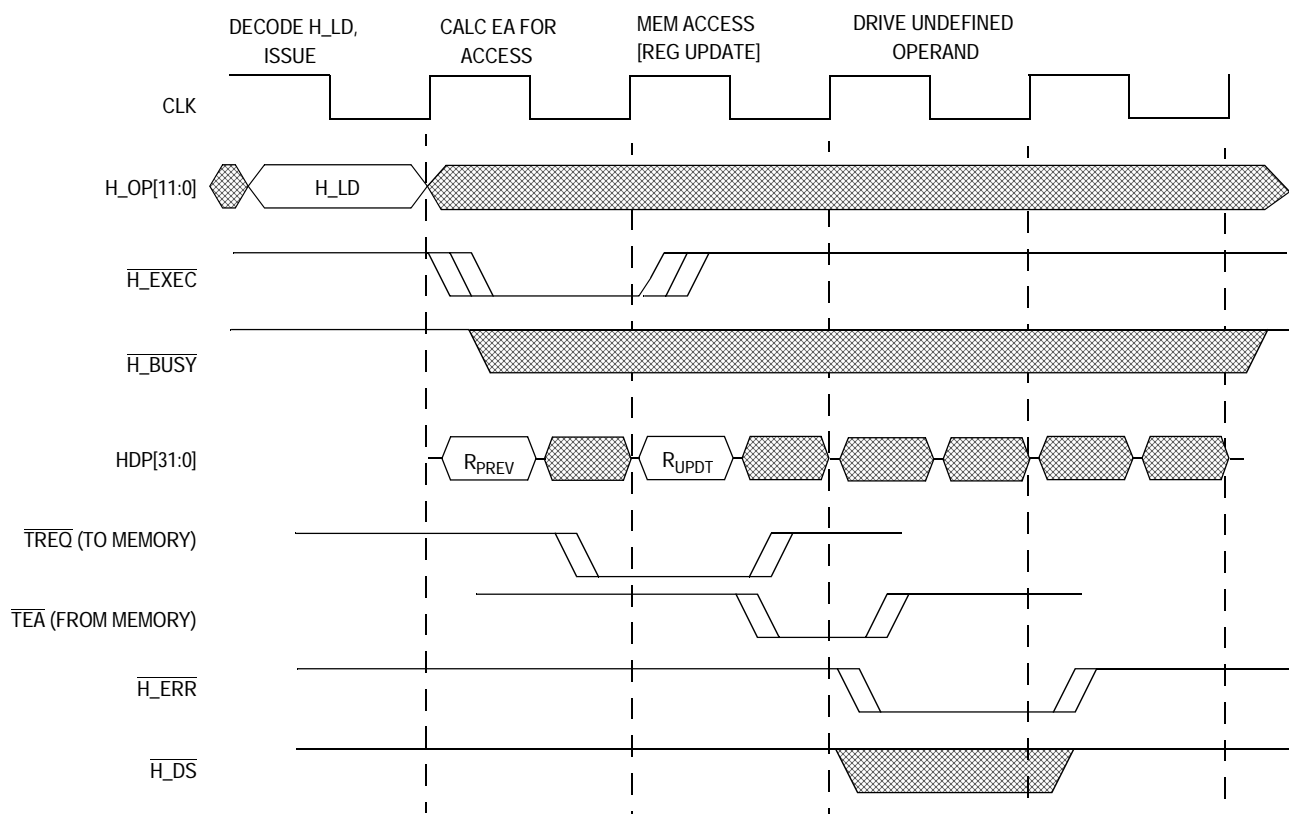
The **h\_ld** primitive is used to transfer data from memory to a hardware block. Handshaking of the operand transfer to the block is controlled by the data strobe ( $\overline{H\_DS}$ ) signal. Data strobe will be asserted by the core to indicate that a valid operand has been placed on the HDP[31:0] bus. The data acknowledge ( $\overline{H\_DA}$ ) input is ignored for this transfer.

**Figure 7-13** shows the sequencing of an **h\_ld** transfer to the interface. In this case, there is a no-wait state memory access. For memory accesses with **n** wait-states, the operand and  $\overline{H\_DS}$  would be driven **n** clocks later. If the option to update the base register with the effective address of the load is selected, the update value is driven on HDP[31:0] the first clock after it has been calculated (the clock following the assertion of  $\overline{H\_EXEC}$ ).



**Figure 7-13. Memory Transfer to External Block**

If the memory access results in an access exception, the  $\overline{H\_ERR}$  signal is asserted back to the external block as shown in [Figure 7-14](#).



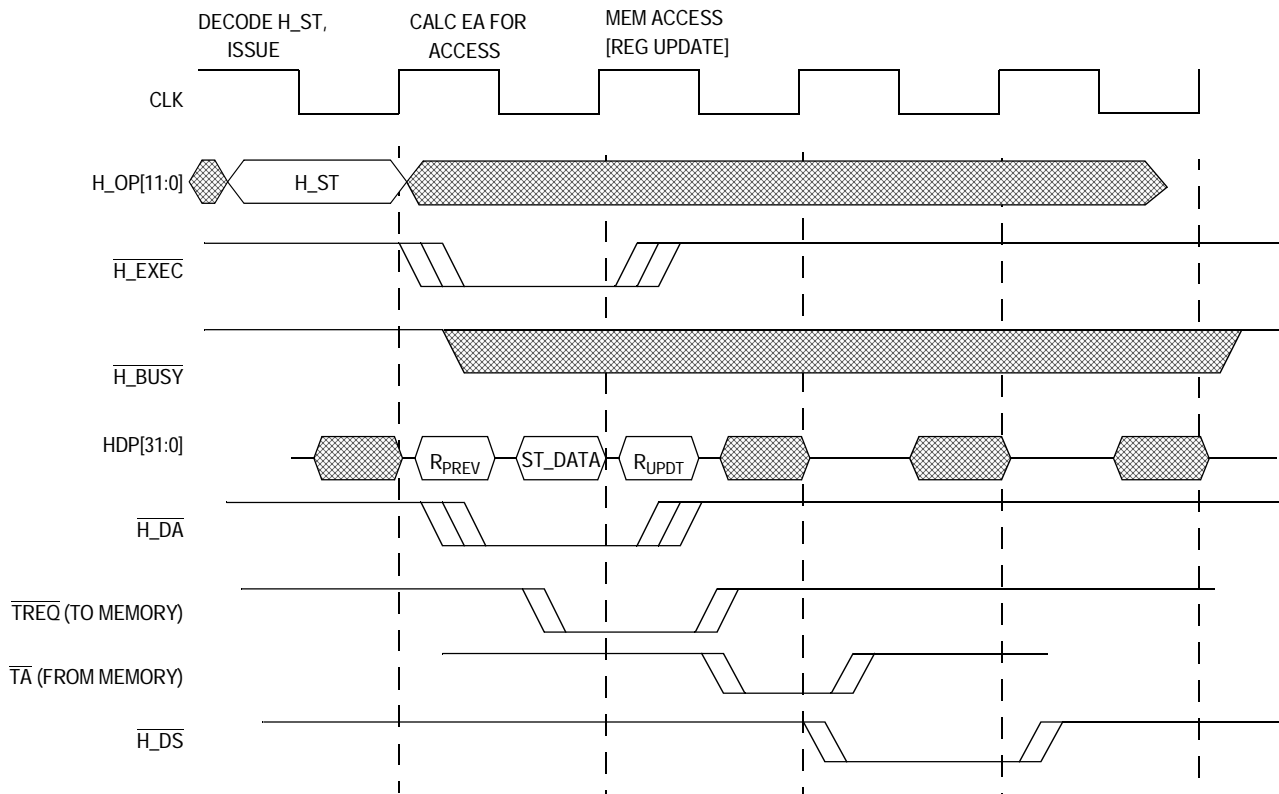
**Figure 7-14. Memory Transfer to External Block with Access Exception**

## 7.6.2.2 H\_ST Transfer

The **h\_st** primitive can be used to transfer data to memory from a hardware block. If the option to update the base register with the effective address of the store is selected, the update value is driven on HDP[31:0] the first clock after it has been calculated (the clock following the assertion of  $\overline{H\_EXEC}$ ).

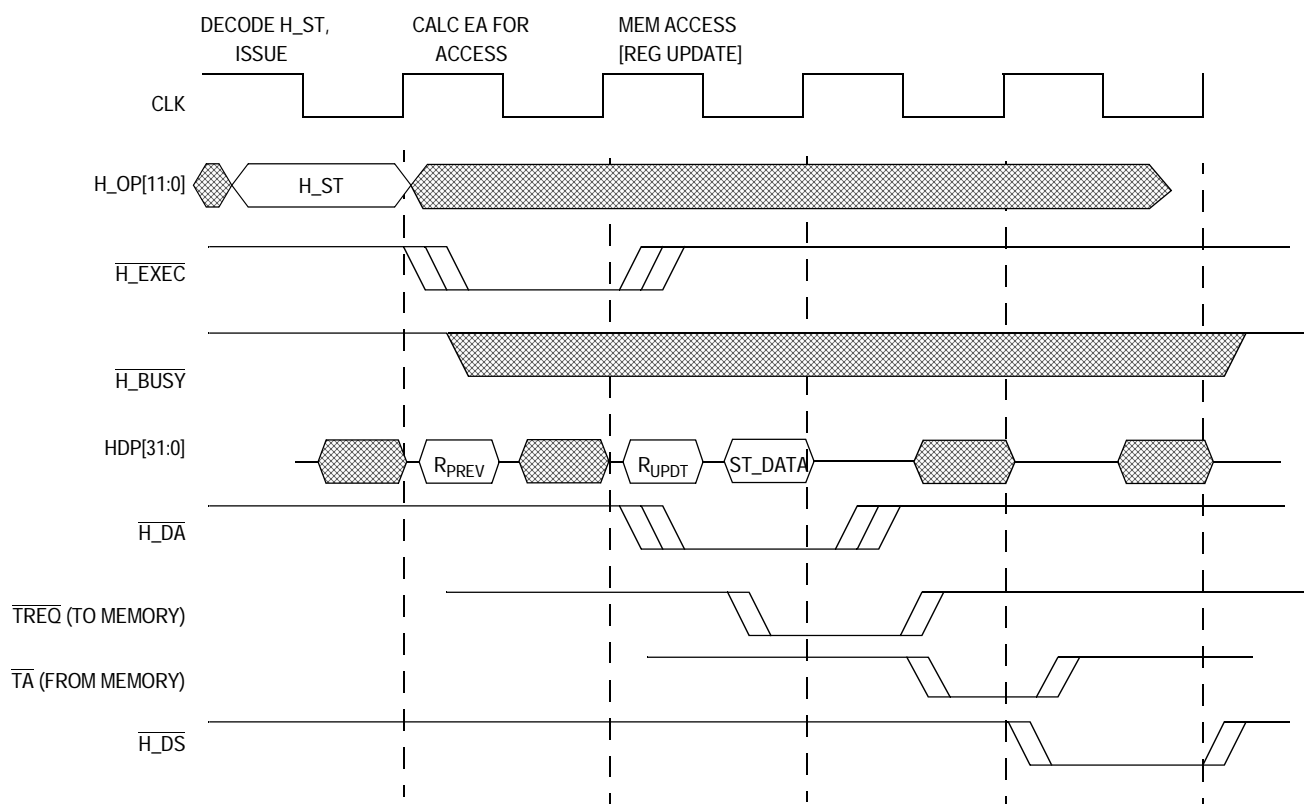
An example of a transfer associated with the **h\_st** primitive is shown in [Figure 7-15](#). The handshake associated with the **h\_st** primitive consists of two parts, an initial handshake from the hardware block, which must provide data for the store, and a completion handshake from the core once the store to memory has completed.

The initial handshake uses the  $\overline{H\_DA}$  input to the core to signal that the hardware block has driven store data to the core. The  $\overline{H\_DA}$  signal is asserted the same clock that data is driven onto the HDP[31:0] bus by the hardware block. The store data is taken from the lower half of the bus for a half-word sized store, the upper 16 bits will not be written into memory. The  $\overline{H\_DA}$  signal will be sampled beginning with the clock the  $\overline{H\_EXEC}$  signal is asserted. The memory cycle is requested during the clock where  $\overline{H\_DA}$  is recognized, and store data will be driven to memory on the following clock. Once the store has completed, the core will assert the  $\overline{H\_DS}$  signal.



**Figure 7-15. Memory Transfer from External Block**

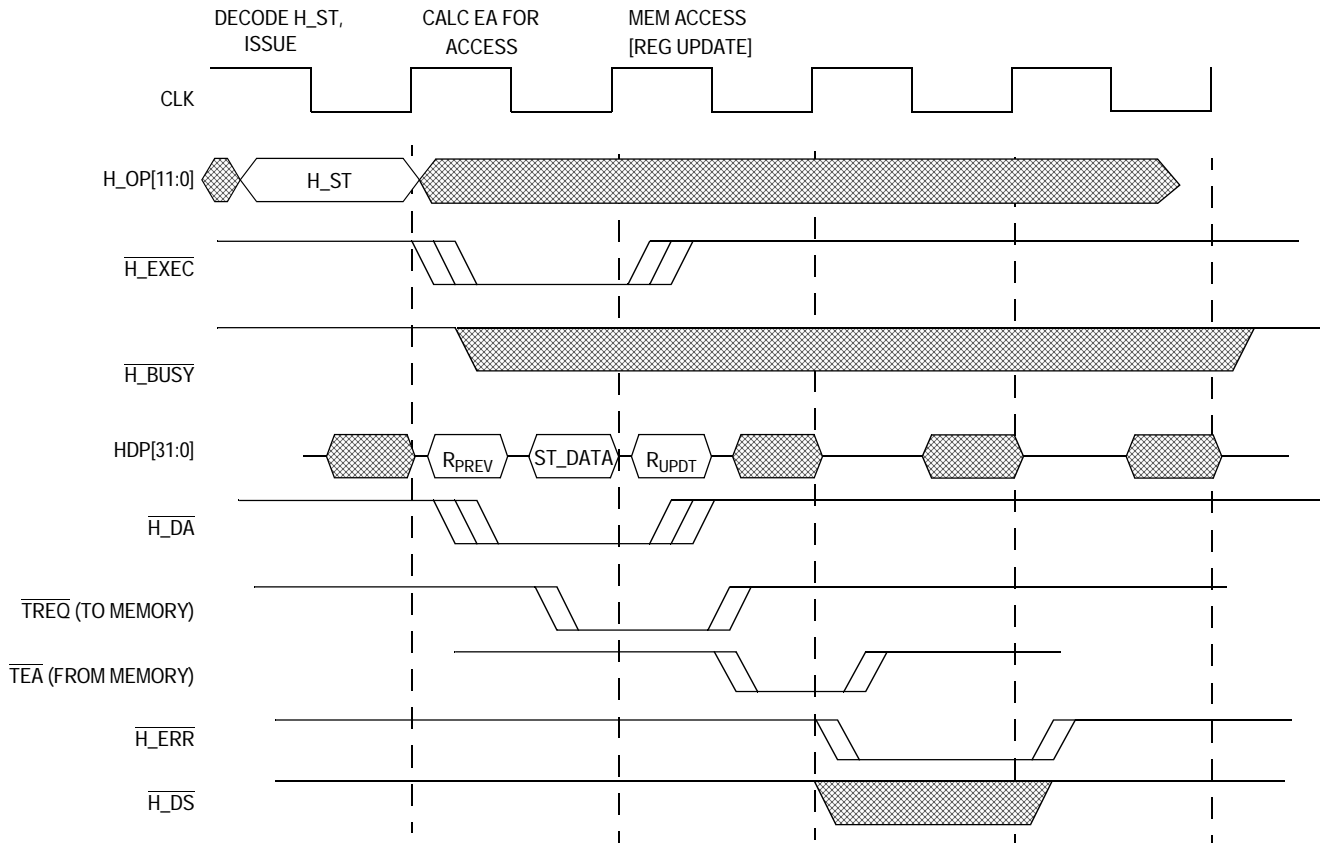
Figure 7-16 shows an example of a transfer with delayed store data.



**Figure 7-16. Delayed Memory Transfer from External Block**

**Figure 7-17** shows how the  $\overline{H\_ERR}$  signal is asserted when the store results in an access error.

If the hardware unit aborts the instruction by asserting  $\overline{H\_EXCP}$  the clock where  $\overline{H\_EXEC}$  is asserted, the  $\overline{H\_DA}$  signal should not be asserted.



**Figure 7-17. Memory Transfer from External Block, Error Termination**

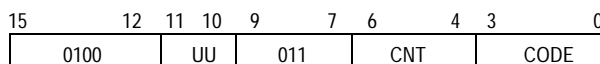
## 7.7 Instruction Primitives

The M•CORE provides several primitives in the core instruction set to interface to a hardware accelerator. This subsection provides a description of the formats of the primitives. The core interprets some of the fields in the primitives, others are interpreted by the accelerator block alone.

**NOTE:** *These primitive definitions are preliminary and subject to change. They are provided to give an overall “feel” for the interface, and have not been finalized.*

### 7.7.1 H\_CALL Primitive

The **h\_call** primitive is used to “call” a function implemented by an accelerator. The paradigm is similar to the software calling convention used by the M•CORE, but in a hardware context. The **h\_call** primitive is interpreted by both the core and the external block to transfer a list of “call parameters” or arguments from the core and initiate a particular function in the block. The instruction format for this primitive is shown in [Figure 7-18](#).



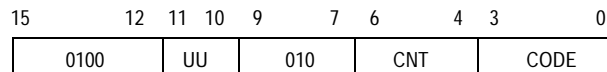
**Figure 7-18. H\_CALL Primitive Format**

The UU and CODE fields of the instruction word are not interpreted by the core, these are used to specify a block specific function. The UU field may specify a hardware unit, and the CODE field may specify a particular operation. The CNT field is interpreted by both the core and the block, and specifies the number of register arguments to pass to the block.

Arguments are passed from the general registers beginning with R4 and continuing through R(4+CNT – 1). Up to seven parameters may be passed. A CNT value of 0b000 is reserved for the **h\_exec** primitive (see [7.7.5 H\\_EXEC Primitive](#)). It would be possible for a block to redefine the meaning of the CODE field for different values of the CNT field if this is appropriate.

### 7.7.2 H\_RET Primitive

The **h\_ret** primitive is used to “return from” a function implemented by an accelerator. The paradigm is similar to the software calling convention used by the M•CORE, but in a hardware context. The **h\_ret** primitive is interpreted by both the core and the external block to transfer a list of “return parameters” or values to the core from a block. The instruction format for this primitive is shown in [Figure 7-19](#).



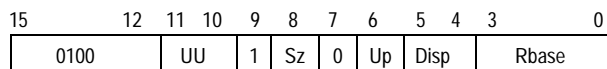
**Figure 7-19. H\_RET Primitive Format**

The UU and CODE fields of the instruction word are not interpreted by the core, these are used to specify a block specific function. The UU field may specify a hardware unit, and the CODE field may specify a particular operation or set of registers in the block to return. The CNT field is interpreted by both the core and the block, and specifies the number of register arguments to pass from the block to the core.

Arguments are passed to the core general registers beginning with R4 and continuing through R(4+CNT – 1). Up to seven parameters may be returned. A CNT value of 0b000 is reserved for the **h\_exec** primitive (see [7.7.5 H\\_EXEC Primitive](#)). It would be possible for a block to redefine the meaning of the CODE field for different values of the CNT field if this is appropriate.

### 7.7.3 H\_LD Primitive

The **h\_ld** primitive is used to pass a value from memory to a block without temporarily storing the memory operand in a core GPR. The memory operand is addressed using a base pointer and an offset. The instruction format for this primitive is shown in [Figure 7-20](#).

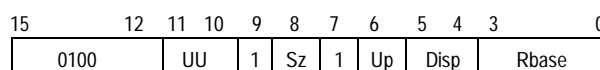


**Figure 7-20. H\_LD Primitive Format**

The UU field of the instruction word is not interpreted by the core, this field may specify a hardware unit. The Sz field specifies the size of the operand (half-word or word only). The Disp field specifies an unsigned offset value to be added to the content of the register specified by the Rbase field to form the effective address for the load. The value of the Disp field is scaled by the size of the operand to be transferred. The Up field specifies whether the Rbase register should be updated with the effective address of the load after it has been calculated. This option allows an “auto-update” addressing mode, although the caveats specified in xxx apply for handling faults on the load data access.

## 7.7.4 H\_ST Primitive

The **h\_st** primitive is used to pass a value from a block to memory without temporarily storing the memory operand in a core GPR. The memory operand is addressed using a base pointer and an offset. The instruction format for this primitive is shown in [Figure 7-21](#).



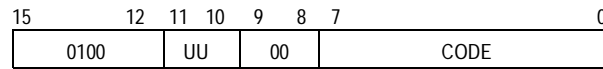
**Figure 7-21. H\_ST Primitive Format**

The UU field of the instruction word is not interpreted by the core, this field may specify a hardware unit. The Sz field specifies the size of the operand (half-word or word only). The Disp field specifies an unsigned offset value to be added to the content of the register specified by the Rbase field to form the effective address for the store. The value of the Disp field is scaled by the size of the operand to be transferred. The Up field specifies whether the Rbase register should be updated with the effective address of the store after it has been calculated. This option allows an “auto-update” addressing mode, although the caveats specified in xxx apply for handling faults on the store data access.



### 7.7.5 H\_EXEC Primitive

The **h\_exec** primitive is used to initiate a function or enter an operating mode implemented by an accelerator. The instruction format for this primitive is shown in [Figure 7-22](#).



**Figure 7-22. H\_EXEC Primitive Format**

The UU and CODE fields of the instruction word are not interpreted by the core, these are used to specify a block specific function. The UU field may specify a hardware unit, and the CODE field may specify a particular operation.

## 7.8 Instruction Primitive Glossary

This section provides a instruction primitive glossary.

H\_CALL

Hardware Accelerator Call Primitive

H\_CALL

**Operation:** Pass parameters to hardware accelerator

**Assembler Syntax:** h\_call #uu, r4-r1ast, #code

**Description:** h\_call passes a set of register-based parameters and a code to hardware block #uu.

**Condition Code:** Unaffected

**Instruction Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	UU		0	1	1	CNT			CODE			

**Instruction Fields:**

- UU field — Specifies hardware block
- 00 — Block 0
  - 01 — Block 1
  - 10 — Block 2
  - 11 — Block 3
- Cnt field — Specifies number of registers to pass, beginning with R4
- 000 — Reserved, do not use
  - 001 — Pass R4
  - ..
  - 111 — Pass R4-R10

# H\_RET

## Hardware Accelerator Return Primitive

# H\_RET

**Operation:** Pass parameters from hardware accelerator

**Assembler Syntax:** `h_ret #uu, r4-r1ast, #code`

**Description:** `h_ret` passes a code to hardware block #uu and receives a set of return parameters to be loaded into CPU registers.

**Condition Code:** Unaffected

**Instruction Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	UU		0	1	0	CNT			CODE			

**Instruction Fields:**

- UU field — Specifies hardware block
- 00 — Block 0
  - 01 — Block 1
  - 10 — Block 2
  - 11 — Block 3
- Cnt field — Specifies number of registers to pass, beginning with R4
- 000 — Reserved, do not use
  - 001 — Pass R4
  - 010 — Pass R4-R5
  - ....
  - 111 — Pass R4-R10

H\_EXEC

Hardware Accelerator Execute Primitive

H\_EXEC

**Operation:** Pass execution code to hardware accelerator

**Assembler Syntax:** h\_exec #uu,#code

**Description:** **h\_exec** is used to control a function in hardware block #uu. The code field is not interpreted by the core

**Condition Code:** Unaffected

**Instruction Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	UU		0	0	CODE							

**Instruction Fields:**

UU field — Specifies hardware block

00 — Block 0

01 — Block 1

10 — Block 2

11 — Block 3

Code field — Specifies an operation code for a hardware block

# H\_LD

## Hardware Accelerator Load Primitive

# H\_LD

**Operation:** Load operand from memory and pass to hardware accelerator

**Assembler Syntax:** h\_ld.[hw][u] #uu, (rx,disp)  
h\_ld.[u] #uu, (rx,disp)

**Description:** **h\_ld** performs a load of a value in memory, and passes the memory operand to the hardware block without storing it in a GPR. The **h\_ld** operation has three options, w (word), h (half-word), and u (update). Disp is obtained by scaling the IMM2 field by the size of the load, and zero-extending. This value is added to the value of register RX and a load of the specified size is performed from this address, with the result of the load passed to the hardware interface. For half-word loads, the data fetched is zero-extended to 32-bits. If the .u option is specified, the effective address of the load is placed in register RX after it is calculated.

**Condition Code:** Unaffected

Instruction Format:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	UU		1	SZ	0	UP	IMM2		Rx			

**Instruction Fields:**

UU field — Specifies hardware block

00 — Block 0

...

11 — Block 3

Size — Specifies load size

0 — Word

1 — Half-word

UP — Specifies whether the base register should be updated

0 — No update

1 — Update base register with effective address

IMM2 Field — Specifies a 2-bit scaled immediate value

Register X — Specifies the base address to be added to the scaled immediate field

H\_ST

Hardware Accelerator Store Primitive

H\_ST

**Operation:** Store operand to memory from hardware accelerator

**Assembler Syntax:** h\_st.[hw][u] #uu, (rx,disp)

**Description:** **h\_st** performs a store to memory, of an operand from a hardware block without storing it in a GPR. The **h\_st** operation has three options, w (word), h (half-word), and u (update). Disp is obtained by scaling the IMM2 field by the size of the store and zero-extending. This value is added to the value of register RX and a store of the specified size is performed to this address, with the data for the store obtained from the hardware interface. If the .u option is specified, the effective address of the load is placed in register RX after it is calculated.

**Condition Code:** Unaffected

Instruction Format:		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		0	1	0	0	UU		1	SZ	1	UP	IMM2		Rx			

**Instruction Fields:**

UU field — Specifies hardware block

00 — Block 0

...

11 — Block 3

Size — Specifies store size

0 — Word

1 — Half-word

UP — Specifies whether the base register should be updated

0 — No update

1 — Update base register with effective address

IMM2 Field — Specifies a 2-bit scaled immediate value

Register X — Specifies the base address to be added to the scaled immediate field

## Section 8. JTAG Test Access Port and OnCE

### 8.1 Contents

8.2	Introduction .....	249
8.3	Top-Level Test Access Port (TAP) .....	251
8.3.1	Test Clock (TCLK) .....	252
8.3.2	Test Mode Select (TMS) .....	252
8.3.3	Test Data Input (TDI) .....	252
8.3.4	Test Data Output (TDO) .....	252
8.3.5	Test Reset ( $\overline{\text{TRST}}$ ) .....	252
8.3.6	Debug Event ( $\overline{\text{DE}}$ ) .....	252
8.4	Top-Level TAP Controller .....	254
8.5	Instruction Shift Register .....	255
8.5.1	EXTEST Instruction .....	255
8.5.2	IDCODE Instruction .....	256
8.5.3	SAMPLE/PRELOAD Instruction .....	257
8.5.4	ENABLE_MCU_ONCE Instruction .....	257
8.5.5	HIGHZ Instruction .....	258
8.5.6	CLAMP Instruction .....	258
8.5.7	BYPASS Instruction .....	258
8.6	IDCODE Register .....	259
8.7	Bypass Register .....	260
8.8	Boundary SCAN Register .....	260
8.9	Restrictions .....	260
8.10	Non-Scan Chain Operation .....	261
8.11	Boundary Scan .....	261
8.12	Low-Level TAP (OnCE) Module .....	267
8.13	Signal Descriptions .....	269
8.13.1	Debug Serial Input (TDI) .....	269
8.13.2	Debug Serial Clock (TCLK) .....	269

8.13.3	Debug Serial Output (TDO) .....	269
8.13.4	Debug Mode Select (TMS).....	270
8.13.5	Test Reset ( $\overline{\text{TRST}}$ ) .....	270
8.13.6	Debug Event ( $\overline{\text{DE}}$ ) .....	270
8.14	Functional Description .....	270
8.14.1	Operation .....	271
8.14.2	OnCE Controller and Serial Interface.....	272
8.14.3	OnCE Interface Signals .....	273
8.14.3.1	Internal Debug Request Input ( $\overline{\text{IDR}}$ ) .....	273
8.14.3.2	CPU Debug Request (DBG $\overline{\text{RQ}}$ ).....	274
8.14.3.3	CPU Debug Acknowledge ( $\overline{\text{DBGACK}}$ ).....	274
8.14.3.4	CPU Breakpoint Request (BRK $\overline{\text{RQ}}$ ).....	274
8.14.3.5	CPU Address, Attributes (ADDR, ATTR).....	274
8.14.3.6	CPU Status (PSTAT) .....	274
8.14.3.7	OnCE Debug Output ( $\overline{\text{DEBUG}}$ ) .....	274
8.14.4	OnCE Controller Registers.....	275
8.14.4.1	OnCE Command Register .....	275
8.14.4.2	OnCE Control Register .....	278
8.14.4.3	OnCE Status Register .....	282
8.14.5	OnCE Decoder (ODEC) .....	284
8.14.6	Memory Breakpoint Logic .....	284
8.14.6.1	Memory Address Latch (MAL) .....	285
8.14.6.2	Breakpoint Address Base Registers .....	285
8.14.7	Breakpoint Address Mask Registers .....	285
8.14.7.1	Breakpoint Address Comparators .....	286
8.14.7.2	Memory Breakpoint Counters .....	286
8.14.8	OnCE Trace Logic .....	286
8.14.8.1	OnCE Trace Counter .....	287
8.14.8.2	Trace Operation .....	288
8.14.9	Methods of Entering Debug Mode .....	288
8.14.9.1	Debug Request During $\overline{\text{RESET}}$ .....	288
8.14.9.2	Debug Request During Normal Activity .....	289
8.14.9.3	Debug Request During Stop, Doze, or Wait Mode ...	289
8.14.9.4	Software Request During Normal Activity .....	289
8.14.10	Enabling OnCE Trace Mode .....	289
8.14.11	Enabling OnCE Memory Breakpoints.....	290
8.14.12	Pipeline Information and Write-Back Bus Register .....	290
8.14.12.1	Program Counter Register .....	291
8.14.12.2	Instruction Register .....	291
8.14.12.3	Control State Register .....	291
8.14.12.4	Writeback Bus Register .....	293



8.14.12.5	Processor Status Register . . . . .	293
8.14.13	Instruction Address FIFO Buffer (PC FIFO) . . . . .	294
8.14.14	Reserved Test Control Registers . . . . .	295
8.14.15	Serial Protocol . . . . .	295
8.14.16	OnCE Commands . . . . .	296
8.14.17	Target Site Debug System Requirements . . . . .	296
8.14.18	Interface Connector for JTAG/OnCE Serial Port . . . . .	296

## 8.2 Introduction

The device has two JTAG (Joint Test Action Group) TAP (test access port) controllers:

1. A top-level controller that allows access to the the device's boundary scan (external pins) register, IDCODE register, and bypass register
2. A low-level OnCE (on-chip emulation) controller that allows access to the device's central processor unit (CPU) and debugger-related registers

At power-up, only the top-level TAP controller will be visible. If desired, a user can then enable the low-level OnCE controller which will in turn disable the top-level TAP controller. The top-level TAP controller will remain disabled until either power is removed and reapplied to the device or until the test reset signal,  $\overline{\text{TRST}}$ , is asserted (logic 0).

The OnCE TAP controller can be enabled in either of two ways:

1. With the top-level TAP controller in its test-logic-reset state:
  - a. Deassert  $\overline{\text{TRST}}$ , test reset (logic1)
  - b. Assert  $\overline{\text{DE}}$ , the debug event (logic 0) for two TCLK, test clock, cycles
2. Shift the ENABLE\_MCU\_ONCE instruction, 0x3, into the top-level TAP controller's instruction register (IR) and pass through the TAP controller state update-IR.

Refer to [Figure 8-1](#).

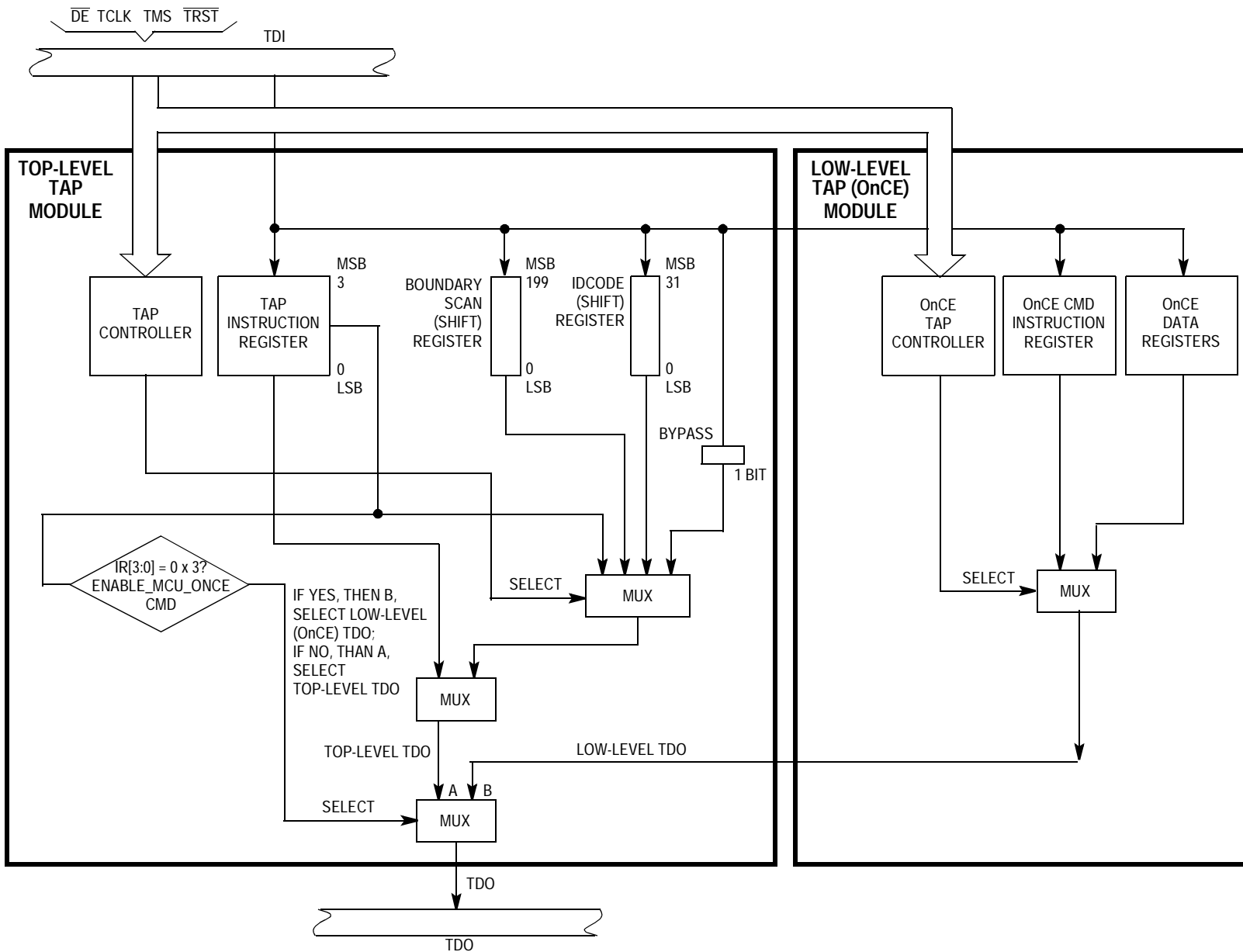


Figure 8-1. Top-Level Tap Module and Low-Level (OnCE) TAP Module

## 8.3 Top-Level Test Access Port (TAP)

The device provides a dedicated user-accessible test access port (TAP) that is fully compatible with the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*. Problems associated with testing high-density circuit boards have led to development of this proposed standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The device implementation supports circuit-board test strategies based on this standard.

The top-level TAP consists of five dedicated signal pins, a 16-state TAP controller, an instruction register, and three data registers, a boundary scan register for monitoring and controlling the device's external pins, a device identification register, and a 1-bit bypass (do nothing) register.

The top-level TAP provides the ability to:

1. Perform boundary scan (external pin) drive and monitor operations to test circuitry external to the device
2. Disable the device's output pins
3. Read the device's IDCODE device identification register

**CAUTION:** *Certain precautions must be observed to ensure that the top-level TAP module does not interfere with non-test operation. See [8.10 Non-Scan Chain Operation](#) for details.*

The device's top-level TAP module includes a TAP controller, a 4-bit instruction register, and three test data registers (a 1-bit bypass register, a 200-bit boundary scan register, and a 32-bit IDCODE register). The top-level tap controller and the low-level (OnCE) TAP controller share the external signals described here.

### 8.3.1 Test Clock (TCLK)

TCLK is a test clock input to synchronize the test logic. TCLK is independent of the device processor clock. It includes an internal pullup resistor.

### 8.3.2 Test Mode Select (TMS)

TMS is a test mode select input (with an internal pullup resistor) that is sampled on the rising edge of TCLK to sequence the TAP controller's state machine.

### 8.3.3 Test Data Input (TDI)

TDI is a serial test data input (with an internal pullup resistor) that is sampled on the rising edge of TCLK.

### 8.3.4 Test Data Output (TDO)

TDO is a three-state test data output that is actively driven in the shift-IR and shift-DR controller states. TDO changes on the falling edge of TCLK.

### 8.3.5 Test Reset ( $\overline{\text{TRST}}$ )

$\overline{\text{TRST}}$  is an active low asynchronous reset with an internal pullup resistor that forces the TAP controller into the test-logic-reset state.

### 8.3.6 Debug Event ( $\overline{\text{DE}}$ )

This is a bidirectional, active-low signal.

As an output, this signal will be asserted for three system clocks, synchronous to the rising CLKOUT edge, to acknowledge that the CPU has entered debug mode as a result of a debug request or a breakpoint condition.

As an input, this signal provides multiple functions such as:

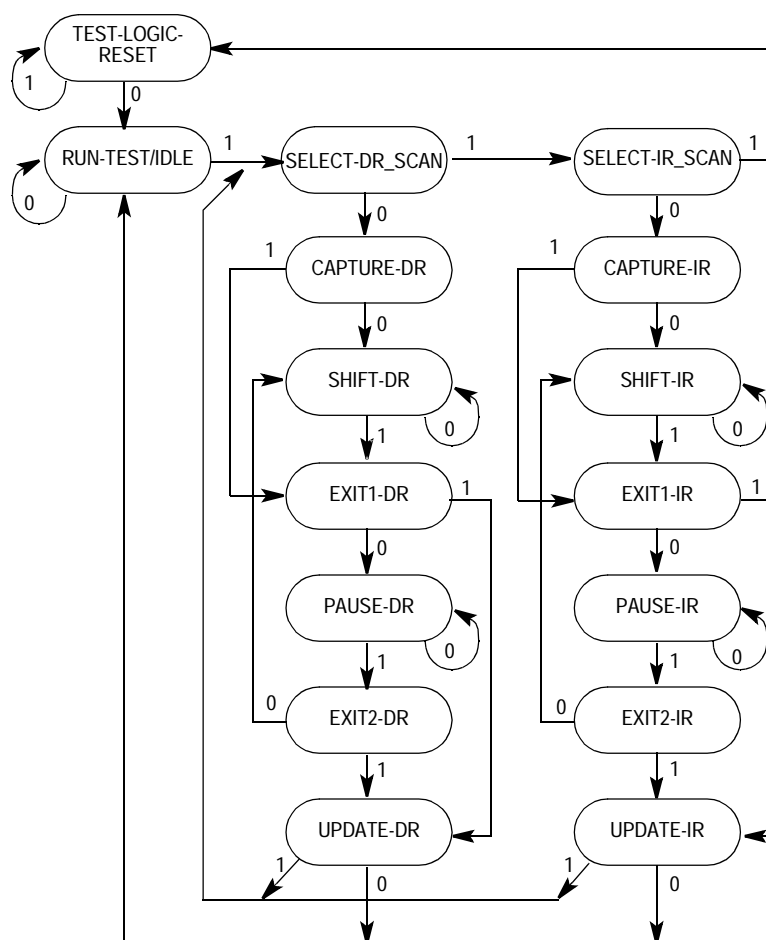
- The main function is a means of entering debug mode from an external command controller. This signal, when asserted, causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for commands to be entered from the serial debug input line. This input must be asserted for at least three system clocks, sampled with the rising CLKOUT edge. This function is ignored during reset. While the processor is in debug mode, this signal is still sampled but has no effect until debug mode is exited.
- Another input function is to enable OnCE. This is an alternate method to the ENABLE\_MCU\_ONCE JTAG command to enable the OnCE logic to be accessible via the JTAG interface. This input signal must be asserted low (while in the test-logic-reset state with POR/TRST not asserted) for at least two TCLK rising edges. Once enabled, the OnCE will remain enabled until the next POR or TRST resets.
- Another input function is as a wake-up event from a low-power mode of operation. Asynchronously asserting this signal will cause the clock controller to restart. This signal must be held asserted until the M•CORE receives three valid rising edges on the system clock. Then the processor will exit the low-power mode and go into debug mode.

**NOTE:** *If used to enter debug mode,  $\overline{DE}$  must be pulled negated before the processor exits debug mode to prevent a still low signal from being unintentionally recognized as another debug request. Also, asserting this signal to enter debug mode may prevent external logic from seeing the processor output acknowledgment since the external pullup may not be able to pull the signal negated before the handshake is asserted. Finally, if using this signal to enable OnCE outside of reset it may be seen as a request to enter debug mode.*

## 8.4 Top-Level TAP Controller

The top-level TAP controller is responsible for interpreting the sequence of logical values on the TMS signal. It is a synchronous state machine that controls the operation of the JTAG logic. The machine's states are shown in [Figure 8-2](#). The value shown adjacent to each arc represents the value of the TMS signal sampled on the rising edge of the TCLK signal.

The top-level TAP controller can be asynchronously reset to the test-logic-reset state by asserting  $\overline{\text{TRST}}$ , test reset. As [Figure 8-2](#) shows, holding TMS high (to logic 1) while clocking TCLK through at least five rising edges will also cause the state machine to enter its test-logic-reset state.



**Figure 8-2. Top-Level TAP Controller State Machine**

## 8.5 Instruction Shift Register

The device top-level TAP module uses a 4-bit instruction shift register with no parity. This register transfers its value to a parallel hold register and applies an instruction on the falling edge of TCLK when the TAP state machine is in the update-IR state. To load the instructions into the shift portion of the register, place the serial data on the TDI pin prior to each rising edge of TCLK. The MSB of the instruction shift register is the bit closest to the TDI pin and the LSB is the bit closest to the TDO pin.

**Table 8-1** lists the instructions supported along with their opcodes, IR3–IR0. The last three instructions in the table are reserved for manufacturing purposes only.

Unused opcodes are currently decoded to perform the BYPASS operation, but Motorola reserves the right to change their decodings in the future.

### 8.5.1 EXTEST Instruction

The external test instruction (EXTEST) selects the boundary-scan register. The EXTEST instruction forces all output pins and bidirectional pins configured as outputs to the preloaded fixed values (with the SAMPLE/PRELOAD instruction) and held in the boundary-scan update registers. The EXTEST instruction can also configure the direction of bidirectional pins and establish high-impedance states on some pins. EXTEST also asserts internal reset for the device system logic to force a predictable internal state while performing external boundary scan operations.

**Table 8-1. JTAG Instructions**

Instruction	IR3–IR0	Instruction Summary
EXTEST	0000	Selects the boundary scan register while applying fixed values to output pins and asserting functional reset
IDCODE	0001	Selects IDCODE register for shift
SAMPLE/PRELOAD	0010	Selects the boundary scan register for shifting, sampling, and preloading without disturbing functional operation
ENABLE_MCU_ONCE	0011	Instruction to enable the M•CORE TAP controller
HIGHZ	1001	Selects the bypass register while three-stating all output pins and asserting functional reset
CLAMP	1100	Selects bypass while applying fixed values to output pins and asserting functional reset
BYPASS	1111	Selects the bypass register for data operations
Reserved	0100 0110	Instruction for chip manufacturing purposes only
Reserved	0101	Instruction for chip manufacturing purposes only <sup>(1)</sup>
Reserved	0111–10001 101–1110 1010–1011	Decoded to select bypass register <sup>(2)</sup>

1. To exit this instruction, the  $\overline{\text{TRST}}$  pin must be asserted or power-on reset.
2. Motorola reserves the right to change the decoding of the unused opcodes in the future.

## 8.5.2 IDCODE Instruction

The IDCODE instruction selects the 32-bit IDCODE register for connection as a shift path between the TDI pin and the TDO pin. This instruction allows interrogation of the device to determine its version number and other part identification data. The IDCODE register has been implemented in accordance with the IEEE 1149.1 standard so that the least significant bit of the shift register stage is set to logic 1 on the rising edge of TCLK following entry into the capture-DR state. Therefore, the first bit to be shifted out after selecting the IDCODE register is always



a logic 1. The remaining 31 bits are also set to fixed values on the rising edge of TCLK following entry into the capture-DR state.

IDCODE is the default instruction placed into the instruction register when the top-level TAP resets. Thus, after a TAP reset, the IDCODE (data) register will be selected automatically.

### 8.5.3 SAMPLE/PRELOAD Instruction

The SAMPLE/PRELOAD instruction provides two separate functions.

First, it obtains a sample of the system data and control signals present at the device input pins and just prior to the boundary scan cell at the output pins. This sampling occurs on the rising edge of TCLK in the capture-DR state when an instruction encoding of hex 2 is resident in the instruction register. The user can observe this sampled data by shifting it through the boundary scan register to the output TDO by using the shift-DR state. Both the data capture and the shift operation are transparent to system operation.

**NOTE:** *The user is responsible for providing some form of external synchronization to achieve meaningful results because there is no internal synchronization between TCLK and the system clock.*

The second function of the SAMPLE/PRELOAD instruction is to initialize the boundary scan register update cells before selecting EXTEST or CLAMP. This is achieved by ignoring the data being shifted out of the TDO pin while shifting in initialization data. The update-DR state in conjunction with the falling edge of TCLK can then transfer this data to the update cells. This data will be applied to the external output pins when EXTEST or CLAMP instruction is applied.

### 8.5.4 ENABLE\_MCU\_ONCE Instruction

The ENABLE\_MCU\_ONCE is a public instruction to enable the M•CORE OnCE TAP controller. When the OnCE TAP controller is enabled, the top-level TAP controller connects the internal OnCE TDO to the pin TDO and remains in the run-test/idle state. It will remain in this

state until  $\overline{\text{TRST}}$  is asserted. While the OnCE TAP controller is enabled, the top-level JTAG remains transparent.

### 8.5.5 HIGHZ Instruction

The HIGHZ instruction is provided as a manufacturer's optional public instruction to prevent having to backdrive the output pins during circuit-board testing. When HIGHZ is invoked, all output drivers, including the 2-state drivers, are turned off (for example, high impedance). The instruction selects the bypass register. HIGHZ also asserts internal reset for the device system logic to force a predictable internal state.

### 8.5.6 CLAMP Instruction

The CLAMP instruction selects the bypass register and asserts internal reset while simultaneously forcing all output pins and bidirectional pins configured as outputs to the fixed values that are preloaded and held in the boundary scan update register. This instruction enhances test efficiency by reducing the overall shift path to a single bit (the bypass register) while conducting an EXTEST type of instruction through the boundary scan register.

### 8.5.7 BYPASS Instruction

The BYPASS instruction selects the single-bit bypass register, creating a single-bit shift register path from the TDI pin to the bypass register to the TDO pin. This instruction enhances test efficiency by reducing the overall shift path when a device other than the device processor becomes the device under test on a board design with multiple chips on the overall IEEE 1149.1 standard defined boundary scan chain. The bypass register has been implemented in accordance with IEEE 1149.1 standard so that the shift register state is set to logic 0 on the rising edge of TCLK following entry into the capture-DR state. Therefore, the first bit to be shifted out after selecting the bypass register is always a logic 0 (to differentiate a part that supports an IDCODE register from a part that supports only the bypass register).

## 8.6 IDCODE Register

An IEEE 1149.1 standard compliant JTAG identification register (IDCODE) has been included on the device.

Bit 31	30	29	28	27	26	25	Bit 24
0	0	0	0	0	1	0	1
Bit 23	22	21	20	19	18	17	Bit 16
1	1	0	0	0	0	0	1
Bit 15	14	13	12	11	10	9	Bit 8
0	1	1	1	0	0	0	0
Bit 7	6	5	4	3	2	1	Bit 0
0	0	0	1	1	1	0	1

**Figure 8-3. IDCODE Register Bit Specification**

### Bits 31–28 — Version Number (Part Revision Number)

This is equivalent to the lower four bits of the PRN of the chip identification register located in the chip configuration module.

### Bits 27–22 — Design Center

Indicates the Motorola Microcontroller Division

### Bits 21–12 — Device Number (Part Identification Number)

Bits 19-12 are equivalent to the PIN of the chip identification register located in the chip configuration module.

### Bits 11–1 — JEDEC ID

Indicates the reduced JEDEC ID for Motorola. JEDEC refers to the Joint Electron Device Engineering Council. Refer to JEDEC publication 106-A and chapter 11 of the IEEE 1149.1 standard for further information on this field.

### Bit 0

Differentiates this register as the JTAG IDCODE register (as opposed to the bypass register), according to the IEEE 1149.1 standard

### 8.7 Bypass Register

The device includes an IEEE 1149.1 standard-compliant bypass register, which creates a single bit shift register path from TDI to the bypass register to TDO when the BYPASS instruction is selected.

### 8.8 Boundary SCAN Register

The device includes an IEEE 1149.1 standard-compliant boundary-scan register. The boundary-scan register is connected between TDI and TDO when the EXTEST or SAMPLE/PRELOAD instructions are selected. This register captures signal pin data on the input pins, forces fixed values on the output signal pins, and selects the direction and drive characteristics (a logic value or high impedance) of the bidirectional and three-state signal pins.

### 8.9 Restrictions

The test logic is implemented using static logic design, and TCLK can be stopped in either a high or low state without loss of data. The system logic, however, operates on a different system clock which is not synchronized to TCLK internally. Any mixed operation requiring the use of the IEEE 1149.1 standard test logic, in conjunction with system functional logic that uses both clocks, must have coordination and synchronization of these clocks done externally.

The control afforded by the output enable signals using the boundary scan register and the EXTEST instruction requires a compatible circuit-board test environment to avoid device-destructive configurations. The user must avoid situations in which the device output drivers are enabled into actively driven networks.

The device features a low-power stop mode. The interaction of the scan chain interface with low-power stop mode is:

1. The TAP controller must be in the test-logic-reset state to either enter or remain in the low-power stop mode. Leaving the test-logic-reset state negates the ability to achieve low-power, but does not otherwise affect device functionality.
2. The TCLK input is not blocked in low-power stop mode. To consume minimal power, the TCLK input should be externally connected to  $V_{DD}$ .
3. The TMS, TDI,  $\overline{TRST}$  pins include on-chip pullup resistors. In low-power stop mode, these three pins should remain either unconnected or connected to  $V_{DD}$  to achieve minimal power consumption.

## 8.10 Non-Scan Chain Operation

Keeping the TAP controller in the test-logic-reset state will ensure that the scan chain test logic is kept transparent to the system logic. It is recommended that TMS, TDI, TCLK, and  $\overline{TRST}$  be pulled up.  $\overline{TRST}$  could be connected to ground. However, since there is a pullup on  $\overline{TRST}$ , some amount of current will result. JTAG will be initialized to the test-logic-reset state on power-up without  $\overline{TRST}$  asserted low due to the JTAG power-on-reset internal input. The low-level TAP module in the M•CORE also has the power-on-reset input.

## 8.11 Boundary Scan

The device boundary-scan register contains 200 bits. This register can be connected between TDI and TDO when EXTEST or SAMPLE/PRELOAD instructions are selected. This register is used for capturing signal pin data on the input pins, forcing fixed values on the output signal pins, and selecting the direction and drive characteristics (a logic value or high impedance) of the bidirectional and three-state signal pins.

This IEEE 1149.1 standard-compliant boundary-scan register contains bits for bonded-out and non-bonded signals excluding JTAG signals, analog signals, power supplies, compliance enable pins, and clock signals. To maintain JTAG compliance, TEST should be held to logic 0 and  $\overline{DE}$  should be held to logic 1. These non-scanned pins are shown in [Table 8-2](#).

**Table 8-2. List of Pins Not Scanned in JTAG Mode**

Pin Name	Pin Type
EXTAL	Clock/analog
XTAL	Clock/analog
$V_{DDSYN}$	Supply
$V_{SSSYN}$	Supply
PQA4–PQA3 and PQA1–PQA0	Analog
PQB3–PQB0	Analog
$V_{RH}$	Supply
$V_{RL}$	Supply
$V_{DDA}$	Supply
$V_{SSA}$	Supply
$V_{DDH}$	Supply
$\overline{TRST}$	JTAG
TCLK	JTAG
TMS	JTAG
TDI	JTAG
TDO	JTAG
DE	JTAG compliance enable
TEST	JTAG compliance enable
$V_{pp}$	Supply
$V_{DDF}$	Supply
$V_{SSF}$	Supply
$V_{STBY}$	Supply
$V_{DD}$	Supply
$V_{SS}$	Supply

**Table 8-3** defines the boundary-scan register.

- The first column shows bit numbers assigned to each of the register's cells. The bit nearest to TDO (the first to be shifted in) is defined as bit 0.
- The second column lists the logical state bit for each device pin alternately with the read/write direction control bit for that pin. The logic state bits are non-inverting with respect to their associated pins, so that a 1 logical state bit equates to a logical high voltage on its corresponding pin. A direction control bit value of 1 causes a pin's logical state to be expressed by its logic state bit, a read of a pin. A direction control bit value of 0 causes a pin's logical voltage to follow the state of its logical state bit, a write to a pin.

**Table 8-3. Boundary-Scan Register Definition (Sheet 1 of 4)**  
(Note: Shaded regions indicate optional pins)

Bit	Logical State and Direction Control Bits for Each Pin	Bit	Logical State and Direction Control Bits for Each Pin
0	D31 logical state	17	A18 direction control
1	D31 direction control	18	A19 logical state
2	A12 logical state	19	A19 direction control
3	A12 direction control	20	$\overline{\text{RSTOUT}}$ logical state
4	A13 logical state	21	$\overline{\text{RSTOUT}}$ direction control
5	A13 direction control	22	A20 logical state
6	A14 logical state	23	A20 direction control
7	A14 direction control	24	$\overline{\text{RESET}}$ logical state
8	A15 logical state	25	$\overline{\text{RESET}}$ direction control
9	A15 direction control	26	A21 logical state
10	A16 logical state	27	A21 direction control
11	A16 direction control	28	A22 logical state
12	A17 logical state	29	A22 direction control
13	A17 direction control	30	$\overline{\text{TEA}}$ logical state
14	CLKOUT logical state	31	$\overline{\text{TEA}}$ direction control
15	CLKOUT direction control	32	$\overline{\text{EB0}}$ logical state
16	A18 logical state	33	$\overline{\text{EB0}}$ direction control
34	$\overline{\text{EB1}}$ logical state	64	$\overline{\text{CS2}}$ logical state

**Table 8-3. Boundary-Scan Register Definition (Sheet 2 of 4)**  
(Note: Shaded regions indicate optional pins)

Bit	Logical State and Direction Control Bits for Each Pin	Bit	Logical State and Direction Control Bits for Each Pin
35	$\overline{EB1}$ direction control	65	$\overline{CS2}$ direction control
36	$\overline{TA}$ logical state	66	$\overline{INT4}$ logical state
37	$\overline{TA}$ direction control	67	$\overline{INT4}$ direction control
38	$\overline{EB2}$ logical state	68	$\overline{CS3}$ logical state
39	$\overline{EB2}$ direction control	69	$\overline{CS3}$ direction control
40	$\overline{SHS}$ logical state	70	TC0 logical state
41	$\overline{SHS}$ direction control	71	TC0 direction control
42	$\overline{EB3}$ logical state	72	$\overline{INT3}$ logical state
43	$\overline{EB3}$ direction control	73	$\overline{INT3}$ direction control
44	$\overline{OE}$ logical state	74	TC1 logical state
45	$\overline{OE}$ direction control	75	TC1 direction control
46	$\overline{SS}$ logical state	76	$\overline{INT2}$ logical state
47	$\overline{SS}$ direction control	77	$\overline{INT2}$ direction control
48	SCK logical state	78	$\overline{INT1}$ logical state
49	SCK direction control	79	$\overline{INT1}$ direction control
50	MISO logical state	80	$\overline{INT0}$ logical state
51	MISO direction control	81	$\overline{INT0}$ direction control
52	MOSI logical state	82	RXD1 logical state
53	MOSI direction control	83	RXD1 direction control
54	$\overline{INT7}$ logical state	84	TXD1 logical state
55	$\overline{INT7}$ direction control	85	TXD1 direction control
56	$\overline{INT6}$ logical state	86	RXD2 logical state
57	$\overline{INT6}$ direction control	87	RXD2 direction control
58	$\overline{CS0}$ logical state	88	TC2 logical state
59	$\overline{CS0}$ direction control	89	TC2 direction control
60	$\overline{CS1}$ logical state	90	TXD2 logical state
61	$\overline{CS1}$ direction control	91	TXD2 direction control
62	$\overline{INT5}$ logical state	92	CSE0 logical state
63	$\overline{INT5}$ direction control	93	CSE0 direction control
94	ICOC1_0 logical state	124	D2 logical state



**Table 8-3. Boundary-Scan Register Definition (Sheet 3 of 4)**  
(Note: Shaded regions indicate optional pins)

Bit	Logical State and Direction Control Bits for Each Pin	Bit	Logical State and Direction Control Bits for Each Pin
95	ICOC1_0 direction control	125	D2 direction control
96	$\overline{\text{CSE1}}$ logical state	126	D3 logical state
97	$\overline{\text{CSE1}}$ direction control	127	D3 direction control
98	$\text{R}/\overline{\text{W}}$ logical state	128	D4 logical state
99	$\text{R}/\overline{\text{W}}$ direction control	129	D4 direction control
100	ICOC1_1 logical state	130	D5 logical state
101	ICOC1_1 direction control	131	D5 direction control
102	ICOC1_2 logical state	132	D6 logical state
103	ICOC1_2 direction control	133	D6 direction control
104	ICOC1_3 logical state	134	D7 logical state
105	ICOC1_3 direction control	135	D7 direction control
106	ICOC2_0 logical state	136	D8 logical state
107	ICOC2_0 direction control	137	D8 direction control
108	ICOC2_1 logical state	138	D9 logical state
109	ICOC2_1 direction control	139	D9 direction control
110	ICOC2_2 logical state	140	D10 logical state
111	ICOC2_2 direction control	141	D10 direction control
112	ICOC2_3 logical state	142	D11 logical state
113	ICOC2_3 direction control	143	D11 direction control
114	D0 logical state	144	D12 logical state
115	D0 direction control	145	D12 direction control
116	A0 logical state	146	D13 logical state
117	A0 direction control	147	D13 direction control
118	A1 logical state	148	D14 logical state
119	A1 direction control	149	D14 direction control
120	D1 logical state	150	A3 logical state
121	D1 direction control	151	A3 direction control
122	A2 logical state	152	A4 logical state
123	A2 direction control	153	A4 direction control
154	D15 logical state	177	A8 direction control

**Table 8-3. Boundary-Scan Register Definition (Sheet 4 of 4)**  
(Note: Shaded regions indicate optional pins)

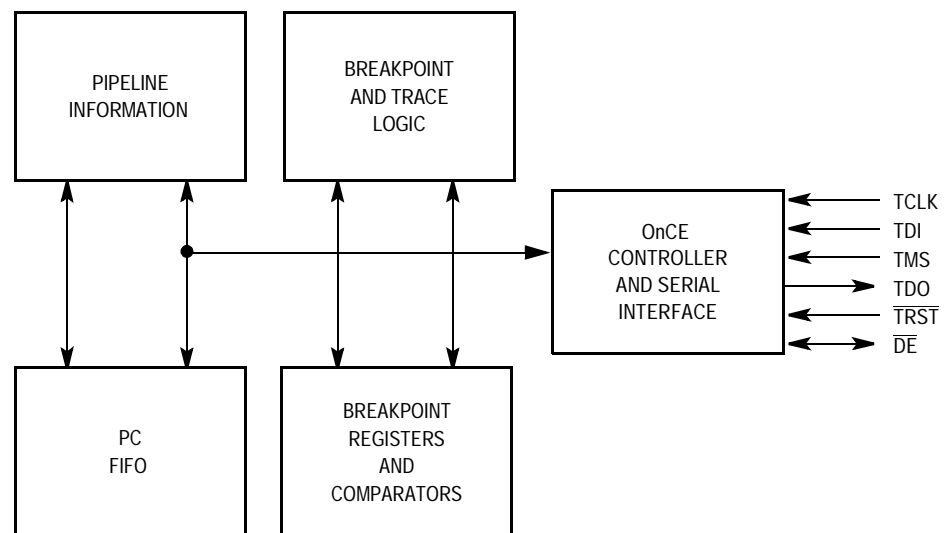
Bit	Logical State and Direction Control Bits for Each Pin	Bit	Logical State and Direction Control Bits for Each Pin
155	D15 direction control	178	A9 logical state
156	A5 logical state	179	A9 direction control
157	A5 direction control	180	D23 logical state
158	D16 logical state	181	D23 direction control
159	D16 direction control	182	A10 logical state
160	A6 logical state	183	A10 direction control
161	A6 direction control	184	D24 logical state
162	A7 logical state	185	D24 direction control
163	A7 direction control	186	D25 logical state
164	D17 logical state	187	D25 direction control
165	D17 direction control	188	A11 logical state
166	D18 logical state	189	A11 direction control
167	D18 direction control	190	D26 logical state
168	D19 logical state	191	D16 direction control
169	D19 direction control	192	D27 logical state
170	D20 logical state	193	D27 direction control
171	D20 direction control	194	D28 logical state
172	D21 logical state	195	D28 direction control
173	D21 direction control	196	D29 logical state
174	D22 logical state	197	D29 direction control
175	D22 direction control	198	D30 logical state
176	A8 logical state	199	D30 direction control

## 8.12 Low-Level TAP (OnCE) Module

The low-level TAP (OnCE, on-chip emulation) circuitry provides a simple, inexpensive debugging interface that allows external access to the processor's internal registers and to memory/peripherals. OnCE capabilities are controlled through a serial interface, mapped onto a JTAG test access port (TAP) protocol.

Refer to [Figure 8-4](#) for a block diagram of the OnCE.

**NOTE:** *The interface to the OnCE controller and its resources is based on the TAP defined for JTAG in the IEEE 1149.1 standard.*



**Figure 8-4. OnCE Block Diagram**

[Figure 8-5](#) shows the OnCE (low-level TAP module) data registers in the device.

DETAILED VIEW OF OnCE DATA REGISTERS BLOCK FOUND IN [FIGURE 8-1](#)

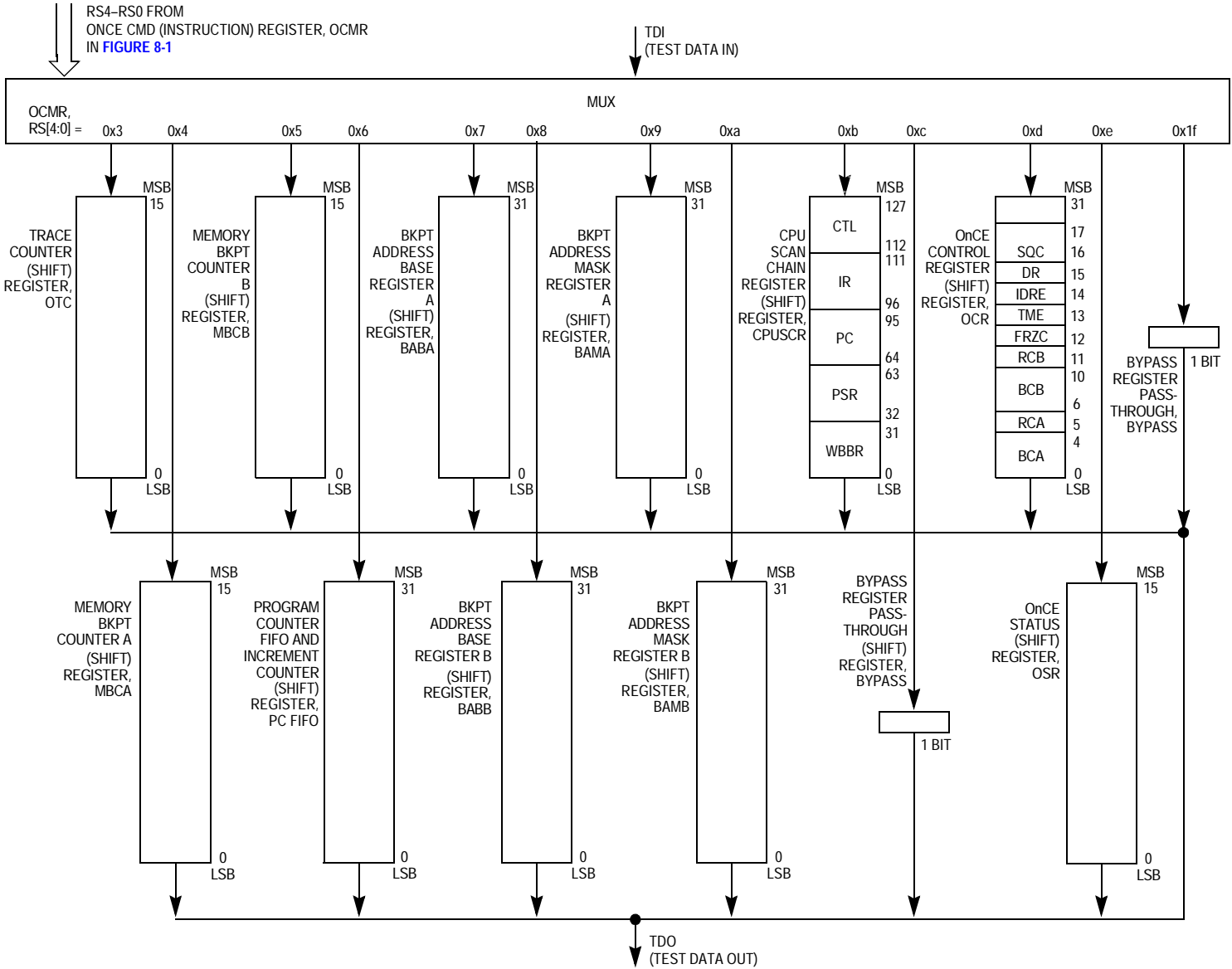


Figure 8-5. Low-Level (OnCE) Tap Module Data Registers (DRs)

## 8.13 Signal Descriptions

The OnCE pin interface is used to transfer OnCE instructions and data to the OnCE control block. Depending on the particular resource being accessed, the CPU may need to be placed in debug mode. For resources outside of the CPU block and contained in the OnCE block, the processor is not disturbed and may continue execution. If a processor resource is required, the OnCE controller may assert a debug request ( $\overline{\text{DBGRQ}}$ ) to the CPU. This causes the CPU to finish the instruction being executed, save the instruction pipeline information, enter debug mode, and wait for further commands. Asserting  $\overline{\text{DBGRQ}}$  causes the device to exit stop, doze, or wait mode.

### 8.13.1 Debug Serial Input (TDI)

Data and commands are provided to the OnCE controller through the TDI pin. Data is latched on the rising edge of the TCLK serial clock. Data is shifted into the OnCE serial port least significant bit (LSB) first.

### 8.13.2 Debug Serial Clock (TCLK)

The TCLK pin supplies the serial clock to the OnCE control block. The serial clock provides pulses required to shift data and commands into and out of the OnCE serial port. (Data is clocked into the OnCE on the rising edge and is clocked out of the OnCE serial port on the falling edge.) The debug serial clock frequency must be no greater than 50 percent of the processor clock frequency.

### 8.13.3 Debug Serial Output (TDO)

Serial data is read from the OnCE block through the TDO pin. Data is always shifted out the OnCE serial port LSB first. Data is clocked out of the OnCE serial port on the falling edge of TCLK. TDO is three-stateable and is actively driven in the shift-IR and shift-DR controller states. TDO changes on the falling edge of TCLK.

### 8.13.4 Debug Mode Select (TMS)

The TMS input is used to cycle through states in the OnCE debug controller. Toggling the TMS pin while clocking with TCLK controls the transitions through the TAP state controller.

### 8.13.5 Test Reset ( $\overline{\text{TRST}}$ )

The  $\overline{\text{TRST}}$  input is used to reset the OnCE controller externally by placing the OnCE control logic in a test logic reset state. OnCE operation is disabled in the reset controller and reserved states.

### 8.13.6 Debug Event ( $\overline{\text{DE}}$ )

The  $\overline{\text{DE}}$  pin is a bidirectional open drain pin. As an input,  $\overline{\text{DE}}$  provides a fast means of entering debug mode from an external command controller. As an output, this pin provides a fast means of acknowledging debug mode entry to an external command controller.

The assertion of this pin by a command controller causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for commands to be entered from the TDI line. If  $\overline{\text{DE}}$  was used to enter debug mode, then  $\overline{\text{DE}}$  must be negated after the OnCE responds with an acknowledgment and before sending the first OnCE command.

The assertion of this pin by the CPU acknowledges that it has entered debug mode and is waiting for commands to be entered from the TDI line.

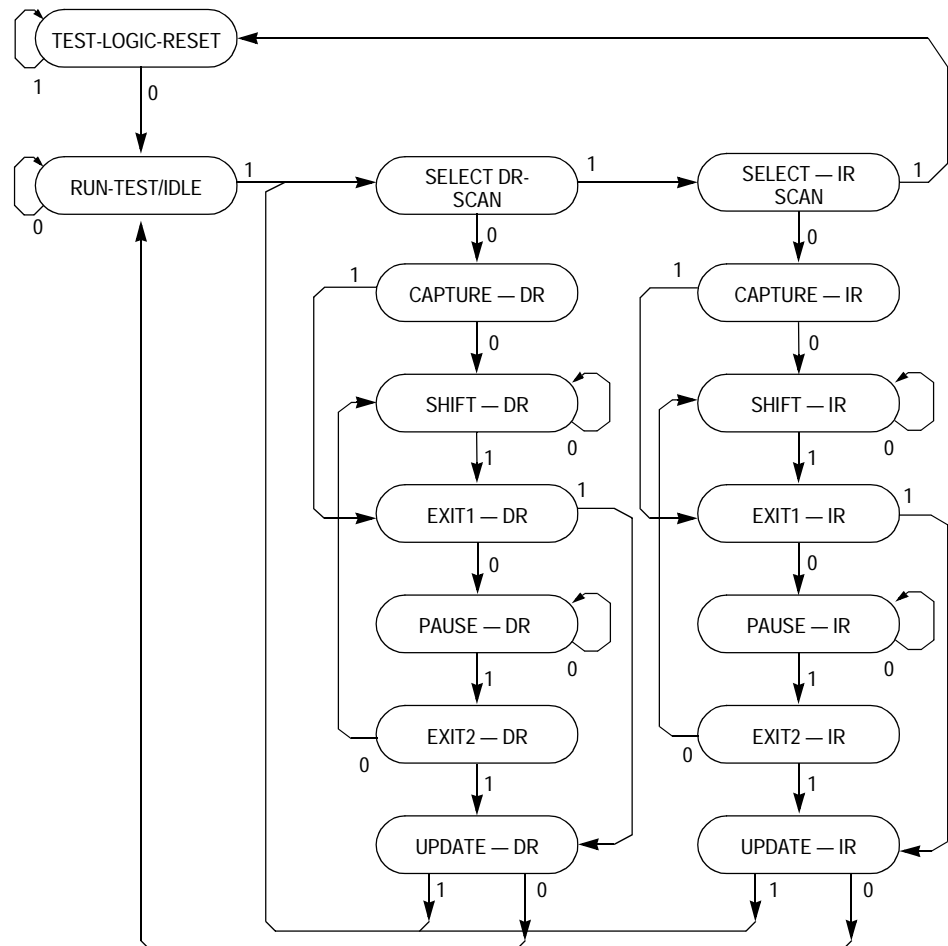
## 8.14 Functional Description

The on-chip emulation (OnCE) circuitry provides a simple, inexpensive debugging interface that allows external access to the processor's internal registers and to memory/peripherals. OnCE capabilities are controlled through a serial interface, mapped onto a JTAG test access port (TAP) protocol. [Figure 8-6](#) shows the components of the OnCE circuitry.

The interface to the OnCE controller and its resources are based on the TAP defined for JTAG in the IEEE 1149.1 standard.

### 8.14.1 Operation

An instruction is scanned into the OnCE module through the serial interface and then decoded. Data may then be scanned in and used to update a register or resource on a write to the resource, or data associated with a resource may be scanned out for a read of the resource.



**Figure 8-6. OnCE Controller**

For accesses to the CPU internal state, the OnCE controller requests the CPU to enter debug mode via the CPU  $\overline{\text{DBG RQ}}$  input. Once the CPU enters debug mode, as indicated by the OnCE status register, the processor state may be accessed through the CPU scan register.

The OnCE controller is implemented as a 16-state finite state machine, with a one-to-one correspondence to the states defined for the JTAG TAP controller.

CPU registers and the contents of memory locations are accessed by scanning instructions and data into and out of the CPU scan chain. Required data is accessed by executing the scanned instructions. Memory locations may be read by scanning in a load instruction to the CPU that references the desired memory location, executing the load instruction, and then scanning out the result of the load. Other resources are accessed in a similar manner.

Resources contained in the OnCE module that do not require the CPU to be halted for access may be controlled while the CPU is executing and do not interfere with normal processor execution. Accesses to certain resources, such as the PC FIFO and the count registers, while not part of the CPU, may require the CPU to be stopped to allow access to avoid synchronization hazards. If it is known that the CPU clock is enabled and running no slower than the TCLK input, there is sufficient synchronization performed to allow reads but not writes of these specific resources. Debug firmware may ensure that it is safe to access these resources by reading the OSR to determine the state of the CPU prior to access. All other cases require the CPU to be in the debug state for deterministic operation.

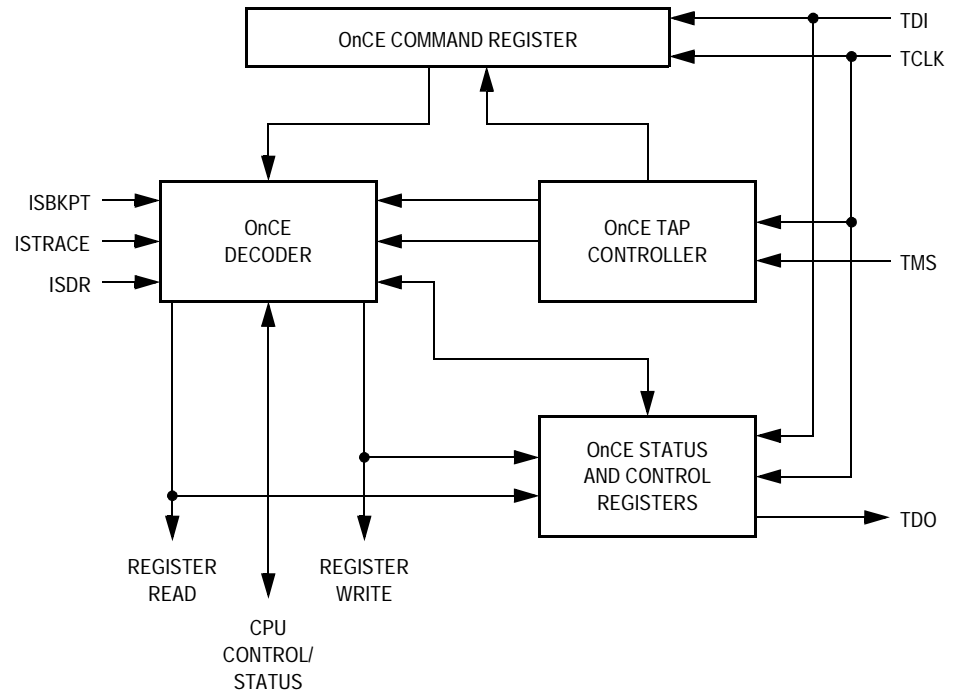
### 8.14.2 OnCE Controller and Serial Interface

**Figure 8-7** is a block diagram of the OnCE controller and serial interface.

The OnCE command register acts as the instruction register (IR) for the TAP controller. All other OnCE resources are treated as data registers (DR) by the TAP controller. The command register is loaded by serially shifting in commands during the TAP controller shift-IR state, and is loaded during the update-IR state. The command register selects a



OnCE resource to be accessed as a DR during the TAP controller capture-DR, shift-DR and update-DR states.



**Figure 8-7. OnCE Controller and Serial Interface**

### 8.14.3 OnCE Interface Signals

The following paragraphs describe the OnCE interface signals to other internal blocks associated with the OnCE controller. These signals are not available externally, and descriptions are provided to improve understanding of OnCE operation.

#### 8.14.3.1 Internal Debug Request Input ( $\overline{IDR}$ )

The internal debug request input is a hardware signal which is used in some implementations to force an immediate debug request to the CPU. If present and enabled, it functions in an identical manner to the control function provided by the DR control bit in the OCR. This input is maskable by a control bit in the OCR.

### 8.14.3.2 CPU Debug Request ( $\overline{DBGRQ}$ )

The  $\overline{DBGRQ}$  signal is asserted by the OnCE control logic to request the CPU to enter the debug state. It may be asserted for a number of different conditions. Assertion of this signal causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for further commands. Asserting  $\overline{DBGRQ}$  causes the device to exit stop, doze, or wait mode.

### 8.14.3.3 CPU Debug Acknowledge ( $\overline{DBGACK}$ )

The CPU asserts the  $\overline{DBGACK}$  signal upon entering the debug state. This signal is part of the handshake mechanism between the OnCE control logic and the CPU.

### 8.14.3.4 CPU Breakpoint Request ( $\overline{BRKRQ}$ )

The  $\overline{BRKRQ}$  signal is asserted by the OnCE control logic to signal that a breakpoint condition has occurred for the current CPU bus access.

### 8.14.3.5 CPU Address, Attributes ( $ADDR$ , $ATTR$ )

The CPU address and attribute information may be used in the memory breakpoint logic to qualify memory breakpoints with access address and cycle type information.

### 8.14.3.6 CPU Status ( $PSTAT$ )

The trace logic uses the  $PSTAT$  signals to qualify trace count decrements with specific CPU activity.

### 8.14.3.7 OnCE Debug Output ( $\overline{DEBUG}$ )

The  $\overline{DEBUG}$  signal is used to indicate to on-chip resources that a debug session is in progress. Peripherals and other units may use this signal to modify normal operation for the duration of a debug session. This may involve the CPU executing a sequence of instructions solely for the purpose of visibility/system control. These instructions are not part of the

normal instruction stream that the CPU would have executed had it not been placed in debug mode.

This signal is asserted the first time the CPU enters the debug state and remains asserted until the CPU is released by a write to the OnCE command register with the GO and EX bits set, and a register specified as either no register selected or the CPUSCR. This signal remains asserted even though the CPU may enter and exit the debug state for each instruction executed under control of the OnCE controller.

#### 8.14.4 OnCE Controller Registers

This section describes the OnCE controller registers:

- OnCE command register (OCMR)
- OnCE control register (OCR)
- OnCE status register (OSR)

All OnCE registers are addressed by means of the RS field in the OCMR, as shown in [Table 8-4](#).

##### 8.14.4.1 OnCE Command Register

The OnCE command register (OCMR) is an 8-bit shift register that receives its serial data from the TDI pin. This register corresponds to the JTAG IR and is loaded when the update-IR TAP controller state is entered. It holds the 8-bit commands shifted in during the shift-IR controller state to be used as input for the OnCE decoder. The OCMR contains fields for controlling access to a OnCE resource, as well as controlling single-step operation, and exit from OnCE mode.

Although the OCMR is updated during the update-IR TAP controller state, the corresponding resource is accessed in the DR scan sequence of the TAP controller, and as such, the update-DR state must be transitioned through in order for an access to occur. In addition, the update-DR state must also be transitioned through in order for the single-step and/or exit functionality to be performed, even though the command appears to have no data resource requirement associated with it.

Bit 7	6	5	4	3	2	1	Bit 0
R/W	G	EX	RS4	RS3	RS2	RS1	RS0

**Figure 8-8. OnCE Command Register (OCMR)**

## R/W — Read/Write Bit

1 = Read the data in the register specified by the RS field.

0 = Write the data associated with the command into the register specified by the RS field.

## GO — Go Bit

When the GO bit is set, the device executes the instruction in the IR register in the CPUSCR. To execute the instruction, the processor leaves debug mode, executes the instruction, and if the EX bit is cleared, returns to debug mode immediately after executing the instruction. The processor resumes normal operation if the EX bit is set. The GO command is executed only if the operation is a read/write to either the CPUSCR or to “no register selected.” Otherwise, the GO bit has no effect. The processor leaves debug mode after the TAP controller update-DR state is entered.

1 = Execute instruction in IR

0 = Inactive (no action taken)

## EX — Exit Bit

When the EX bit is set, the processor leaves debug mode and resumes normal operation until another debug request is generated. The exit command is executed only if the GO bit is set and the operation is a read/write to the CPUSCR or a read/write to “no register selected.” Otherwise, the EX bit has no effect. The processor exits debug mode after the TAP controller update-DR state is entered.

1 = Leave debug mode

0 = Remain in debug mode

## RS4–RS0 — Register Select Field

The RS field defines the source for the read operation or the destination for the write operation. [Table 8-4](#) shows OnCE register addresses.


**Table 8-4. OnCE Register Addressing**

<b>RS4–RS0</b>	<b>Register Selected</b>
00000	Reserved
00001	Reserved
00010	Reserved
00011	OTC — OnCE trace counter
00100	MBCA — memory breakpoint counter A
00101	MBCB — memory breakpoint counter B
00110	PC FIFO — program counter FIFO and increment counter
00111	BABA — breakpoint address base register A
01000	BABB — breakpoint address base register B
01001	BAMA — breakpoint address mask register A
01010	BAMB — breakpoint address mask register B
01011	CPUSCR — CPU scan chain register
01100	Bypass — no register selected
01101	OCR — OnCE control register
01110	OSR — OnCE status register
01111	Reserved (factory test control register — do not access)
10000	Reserved (MEM_BIST — do not access)
10001–10110	Reserved (bypass, do not access)
10111	Reserved (LSRL, do not access)
11000–11110	Reserved (bypass, do not access)
11111	Bypass

## 8.14.4.2 OnCE Control Register

The 32-bit OnCE control register (OCR) selects the events that put the device in debug mode and enables or disables sections of the OnCE logic.

	Bit 31	30	29	28	27	26	25	Bit 24
Read:	0	0	0	0	0	0	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 23	22	21	20	19	18	17	Bit 16
Read:	0	0	0	0	0	0	SQC1	SQC0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 15	14	13	12	11	10	9	Bit 8
Read:	DR	IDRE	TME	FRZC	RCB	BCB4	BCB3	BCB2
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	BCB1	BCB0	RCA	BCA4	BCA3	BCA2	BCA1	BCA0
Write:								
Reset:	0	0	0	0	0	0	0	0

 = Unimplemented or reserved

**Figure 8-9. OnCE Control Register (OCR)**

### SQC1 and SQC0 — Sequential Control Field

The SQC field allows memory breakpoint B and trace occurrences to be suspended until a qualifying event occurs. Test logic reset clears the SQC field. See [Table 8-5](#).

**Table 8-5. Sequential Control Field Settings**

<b>SQC1 and SQC0</b>	<b>Meaning</b>
00	Disable sequential control operation. Memory breakpoints and trace operation are unaffected by this field.
01	Suspend normal trace counter operation until a breakpoint condition occurs for memory breakpoint B. In this mode, memory breakpoint B occurrences no longer cause breakpoint requests to be generated. Instead, trace counter comparisons are suspended until the first memory breakpoint B occurrence. After the first memory breakpoint B occurrence, trace counter control is released to perform normally, assuming TME is set. This allows a sequence of breakpoint conditions to be specified prior to trace counting.
10	Qualify memory breakpoint B matches with a breakpoint occurrence for memory breakpoint A. In this mode, memory breakpoint A occurrences no longer cause breakpoint requests to be generated. Instead, memory breakpoint B comparisons are suspended until the first memory breakpoint A occurrence. After the first memory breakpoint A occurrence, memory breakpoint B is enabled to perform normally. This allows a sequence of breakpoint conditions to be specified.
11	Combine the 01 and 10 qualifications. In this mode, no breakpoint requests are generated, and trace count operation is enabled once a memory breakpoint B occurrence follows a memory breakpoint A occurrence if TME is set.

#### DR — Debug Request Bit

DR requests the CPU to enter debug mode unconditionally. The PM bits in the OnCE status register indicate that the CPU is in debug mode. Once the CPU enters debug mode, it returns there even with a write to the OCMR with GO and EX set until the DR bit is cleared. Test logic reset clears the DR bit.

#### IDRE — Internal Debug Request Enable Bit

The internal debug request ( $\overline{IDR}$ ) input to the OnCE control logic may not be used in all implementations. In some implementations, the  $\overline{IDR}$  control input may be connected and used as an additional hardware debug request. Test logic reset clears the IDRE bit.

- 1 =  $\overline{IDR}$  input enabled
- 0 =  $\overline{IDR}$  input disabled

### TME — Trace Mode Enable Bit

TME enables trace operation. Test logic reset clears the TME bit.

Trace operation is also affected by the SQC field.

1 = Trace operation enabled

0 = Trace operation disabled

### FRZC — Freeze Control Bit

This control bit is used in conjunction with memory breakpoint B registers to select between asserting a breakpoint condition when a memory breakpoint B occurs or freezing the PC FIFO from further updates when memory breakpoint B occurs while allowing the CPU to continue execution. The PC FIFO remains frozen until the FRZO bit in the OSR is cleared.

1 = Memory breakpoint B occurrence freezes PC FIFO and does not assert breakpoint condition.

0 = Memory breakpoint B occurrence asserts breakpoint condition.

### RCB and RCA — Memory Breakpoint B and A Range Control Bits

RCB and RDA condition enabled memory breakpoint occurrences happen when memory breakpoint matches are either within or outside the range defined by memory base address and mask.

1 = Condition breakpoint on access outside of range

0 = Condition breakpoint on access within range

### BCB4–BCB0 and BCA4–BCA0 — Memory Breakpoint B and A Control Fields

The BCB and BCA fields enable memory breakpoints and qualify the access attributes to select whether the breakpoint matches are recognized for read, write, or instruction fetch (program space) accesses. Test logic reset clears BCB4–BCB0 and BCA4–BCA0.




**Table 8-6. Memory Breakpoint Control Field Settings**

<b>BCB4–BCB0 BCA4–BCA0</b>	<b>Description</b>
00000	Breakpoint disabled
00001	Qualify match with any access
00010	Qualify match with any instruction access
00011	Qualify match with any data access
00100	Qualify match with any change of flow instruction access
00101	Qualify match with any data write
00110	Qualify match with any data read
00111	Reserved
01XXX	Reserved
10000	Reserved
10001	Qualify match with any user access
10010	Qualify match with any user instruction access
10011	Qualify match with any user data access
10100	Qualify match with any user change of flow access
10101	Qualify match with any user data write
10110	Qualify match with any user data read
10111	Reserved
11000	Reserved
11001	Qualify match with any supervisor access
11010	Qualify match with any supervisor instruction access
11011	Qualify match with any supervisor data access
11100	Qualify match with any supervisor change of flow access
11101	Qualify match with any supervisor data write
11110	Qualify match with any supervisor data read
11111	Reserved

## 8.14.4.3 OnCE Status Register

The 16-bit OnCE status register (OSR) indicates the reason(s) that debug mode was entered and the current operating mode of the CPU.

	Bit 15	14	13	12	11	10	9	Bit 8
Read:	0	0	0	0	0	0	HDRO	DRO
Write:								
Reset:							0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	MBO	SWO	TO	FRZO	SQB	SQA	PM1	PM0
Write:								
Reset:	0	0	0	0	0	0	0	0

 = Unimplemented or reserved

**Figure 8-10. OnCE Status Register (OSR)**

### HDRO — Hardware Debug Request Occurrence Flag

HDRO is set when the processor enters debug mode as a result of a hardware debug request from the  $\overline{\text{IDR}}$  signal or the  $\overline{\text{DE}}$  pin. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

### DRO — Debug Request Occurrence Flag

DRO is set when the processor enters debug mode and the debug request (DR) control bit in the OnCE control register is set. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

### MBO — Memory Breakpoint Occurrence Flag

MBO is set when a memory breakpoint request has been issued to the CPU via the  $\overline{\text{BRKRQ}}$  input and the CPU enters debug mode. In some situations involving breakpoint requests on instruction prefetches, the CPU may discard the request along with the prefetch. In this case, this bit may become set due to the CPU entering debug mode for another reason. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### SWO — Software Debug Occurrence Flag

SWO bit is set when the processor enters debug mode of operation as a result of the execution of the BKPT instruction. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### TO — Trace Count Occurrence Flag

TO is set when the trace counter reaches zero with the trace mode enabled and the CPU enters debug mode. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### FRZO — FIFO Freeze Occurrence Flag

FRZO is set when a FIFO freeze occurs. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### SQB — Sequential Breakpoint B Arm Occurrence Flag

SQB is set when sequential operation is enabled and a memory breakpoint B event has occurred to enable trace counter operation. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### SQA — Sequential Breakpoint A Arm Occurrence Flag

SQA is set when sequential operation is enabled and a memory breakpoint A event has occurred to enable memory breakpoint B operation. This bit is cleared on test logic reset or when debug mode is exited with the GO and EX bits set.

#### PM1 and PM0 — Processor Mode Field

These flags reflect the processor operating mode. They allow coordination of the OnCE controller with the CPU for synchronization.

**Table 8-7. Processor Mode Field Settings**

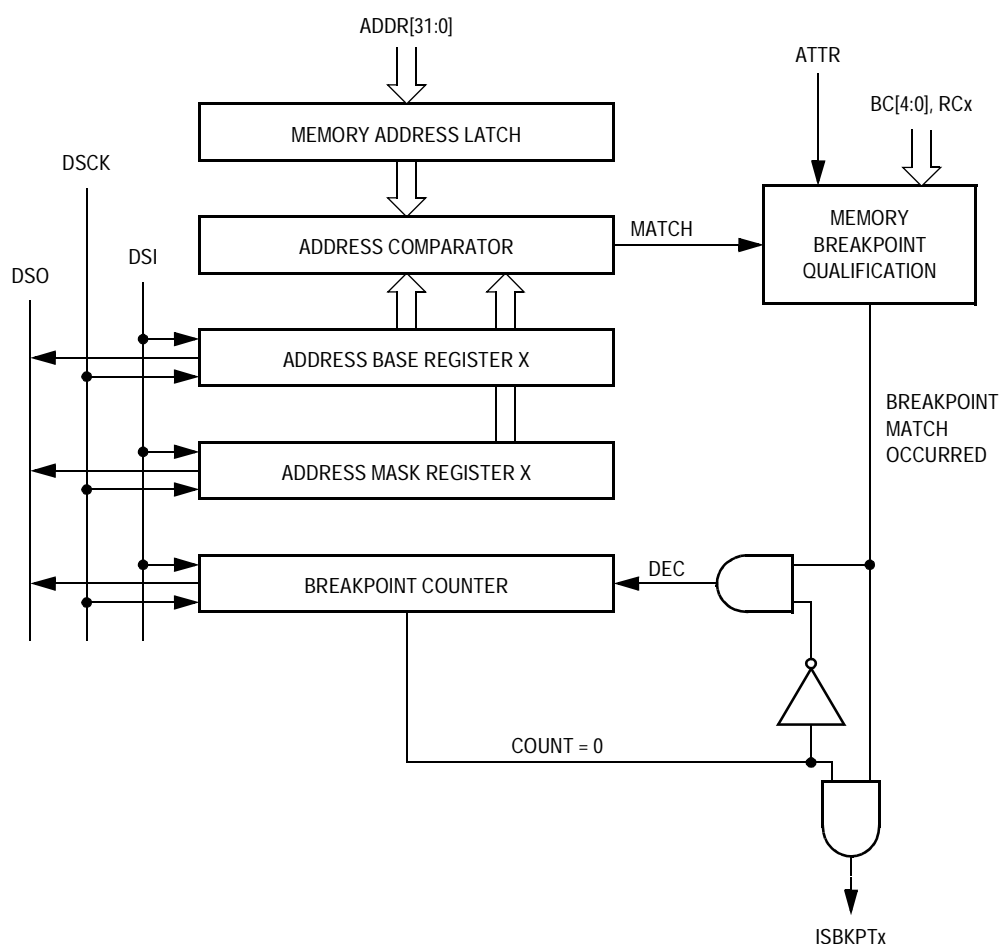
<b>PM1 and PM0</b>	<b>Meaning</b>
00	Processor in normal mode
01	Processor in stop, doze, or wait mode
10	Processor in debug mode
11	Reserved

## 8.14.5 OnCE Decoder (ODEC)

The ODEC receives as input the 8-bit command from the OCMR and status signals from the processor. The ODEC generates all the strobes required for reading and writing the selected OnCE registers.

## 8.14.6 Memory Breakpoint Logic

Memory breakpoints can be set for a particular memory location or on accesses within an address range. The breakpoint logic contains an input latch for addresses, registers that store the base address and address mask, comparators, attribute qualifiers, and a breakpoint counter. **Figure 8-11** illustrates the basic functionality of the OnCE memory breakpoint logic. This logic is duplicated to provide two independent breakpoint resources.



**Figure 8-11. OnCE Memory Breakpoint Logic**

Address comparators can be used to determine where a program may be getting lost or when data is being written to areas which should not be written. They are also useful in halting a program at a specific point to examine or change registers or memory. Using address comparators to set breakpoints enables the user to set breakpoints in RAM or ROM in any operating mode. Memory accesses are monitored according to the contents of the OCR.

The address comparator generates a match signal when the address on the bus matches the address stored in the breakpoint address base register, as masked with individual bit masking capability provided by the breakpoint address mask register. The address match signal and the access attributes are further qualified with the RCx4–RCx0 and BCx4–BCx0 control bits. This qualification is used to decrement the breakpoint counter conditionally if its contents are non-zero. If the contents are zero, the counter is not decremented and the breakpoint event occurs (ISBKPTx asserted).

#### 8.14.6.1 *Memory Address Latch (MAL)*

The MAL is a 32-bit register that latches the address bus on every access.

#### 8.14.6.2 *Breakpoint Address Base Registers*

The 32-bit breakpoint address base registers (BABA and BABB) store memory breakpoint base addresses. BABA and BABB can be read or written through the OnCE serial interface. Before enabling breakpoints, the external command controller should load these registers.

### 8.14.7 **Breakpoint Address Mask Registers**

The 32-bit breakpoint address mask registers (BAMA and BAMB) registers store memory breakpoint base address masks. BAMA and BAMB can be read or written through the OnCE serial interface. Before enabling breakpoints, the external command controller should load these registers.

### 8.14.7.1 Breakpoint Address Comparators

The breakpoint address comparators are not externally accessible. Each compares the memory address stored in MAL with the contents of BABx, as masked by BAMx, and signals the control logic when a match occurs.

### 8.14.7.2 Memory Breakpoint Counters

The 16-bit memory breakpoint counter registers (MBCA and MBCB) are loaded with a value equal to the number of times, minus one, that a memory access event should occur before a memory breakpoint is declared. The memory access event is specified by the RCx4–RCx0 and BCx4–BCx0 bits in the OCR and by the memory base and mask registers. On each occurrence of the memory access event, the breakpoint counter, if currently non-zero, is decremented. When the counter has reached the value of zero and a new occurrence takes place, the  $\overline{\text{ISBKPTx}}$  signal is asserted and causes the CPU's  $\overline{\text{BRKRQ}}$  input to be asserted. The MBCx can be read or written through the OnCE serial interface.

Anytime the breakpoint registers are changed, or a different breakpoint event is selected in the OCR, the breakpoint counter must be written afterward. This assures that the OnCE breakpoint logic is reset and that no previous events will affect the new breakpoint event selected.

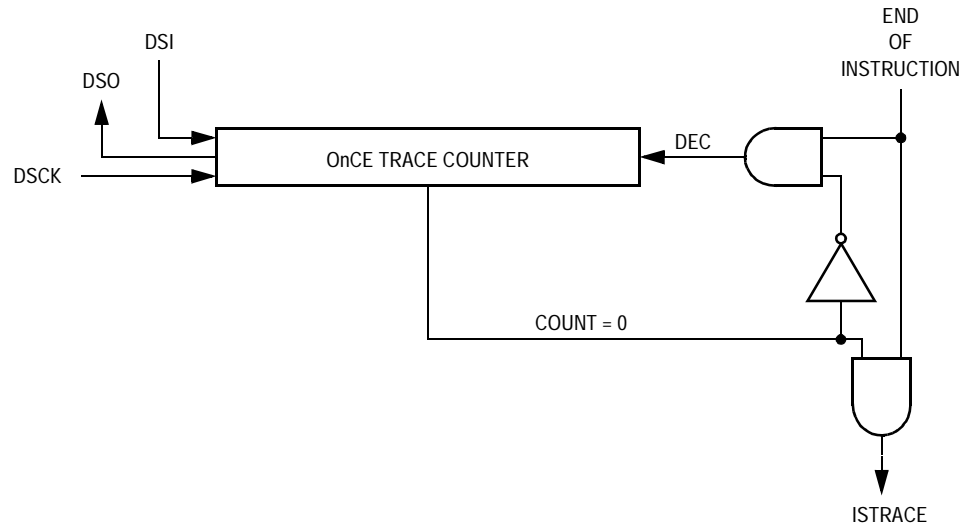
## 8.14.8 OnCE Trace Logic

The OnCE trace logic allows the user to execute instructions in single or multiple steps before the device returns to debug mode and awaits OnCE commands from the debug serial port. The OnCE trace logic is independent of the M•CORE trace facility, which is controlled through the trace mode bits in the M•CORE processor status register. The OnCE trace logic block diagram is shown in [Figure 8-12](#).

### 8.14.8.1 OnCE Trace Counter

The OnCE trace counter register (OTC) is a 16-bit counter that allows more than one instruction to be executed in real time before the device returns to debug mode. This feature helps the software developer debug

sections of code that are time-critical. The trace counter also enables the user to count the number of instructions executed in a code segment.



**Figure 8-12. OnCE Trace Logic Block Diagram**

The OTC register can be read, written, or cleared through the OnCE serial interface. If N instructions are to be executed before entering debug mode, the trace counter should be loaded with  $N - 1$ . N must not equal zero unless the sequential breakpoint control capability is being used. In this case a value of zero (indicating a single instruction) is allowed.

A hardware reset clears the OTC.

#### 8.14.8.2 Trace Operation

To initiate trace mode operation:

1. Load the OTC register with a value. This value must be non-zero, unless sequential breakpoint control operation is enabled in the OCR register. In this case, a value of zero (indicating a single instruction) is allowed.
2. Initialize the program counter and instruction register in the CPUSCR with values corresponding to the start location of the instruction(s) to be executed real time.

3. Set the TME bit in the OCR.
4. Release the processor from debug mode by executing the appropriate command issued by the external command controller.

When debug mode is exited, the counter is decremented after each execution of an instruction. Interrupts can be serviced, and all instructions executed (including interrupt services) will decrement the trace counter.

When the trace counter decrements to zero, the OnCE control logic requests that the processor re-enter debug mode, and the trace occurrence bit TO in the OSR is set to indicate that debug mode has been requested as a result of the trace count function. The trace counter allows a minimum of two instructions to be specified for execution prior to entering trace (specified by a count value of one), unless sequential breakpoint control operation is enabled in the OCR. In this case, a value of zero (indicating a single instruction) is allowed.

### 8.14.9 Methods of Entering Debug Mode

The PM status field in the OSR indicates that the CPU has entered debug mode. The following paragraphs discuss conditions that invoke debug mode.

#### 8.14.9.1 Debug Request During $\overline{\text{RESET}}$

When the DR bit in the OCR is set, assertion of  $\overline{\text{RESET}}$  causes the device to enter debug mode. In this case the device may fetch the reset vector and the first instruction of the reset exception handler but does not execute an instruction before entering debug mode.

#### 8.14.9.2 Debug Request During Normal Activity

Setting the DR bit in the OCR during normal device activity causes the device to finish the execution of the current instruction and then enter debug mode. Note that in this case the device completes the execution of the current instruction and stops after the newly fetched instruction enters the CPU instruction latch. This process is the same for any newly



fetches instruction, including instructions fetched by interrupt processing or those that will be aborted by interrupt processing.

#### 8.14.9.3 Debug Request During Stop, Doze, or Wait Mode

Setting the DR bit in the OCR when the device is in stop, doze, or wait mode (for instance, after execution of a STOP, DOZE, or WAIT instruction) causes the device to exit the low-power state and enter the debug mode. Note that in this case, the device completes the execution of the STOP, DOZE, or WAIT instruction and halts after the next instruction enters the instruction latch.

#### 8.14.9.4 Software Request During Normal Activity

Executing the BKPT instruction when the FDB (force debug enable mode) control bit in the control state register is set causes the CPU to enter debug mode after the instruction following the BKPT instruction has entered the instruction latch.

### 8.14.10 Enabling OnCE Trace Mode

When the OnCE trace mode mechanism is enabled and the trace count is greater than zero, the trace counter is decremented for each instruction executed. Completing execution of an instruction when the trace counter is zero causes the CPU to enter debug mode.

**NOTE:** *Only instructions actually executed cause the trace counter to decrement. An aborted instruction does not decrement the trace counter and does not invoke debug mode.*

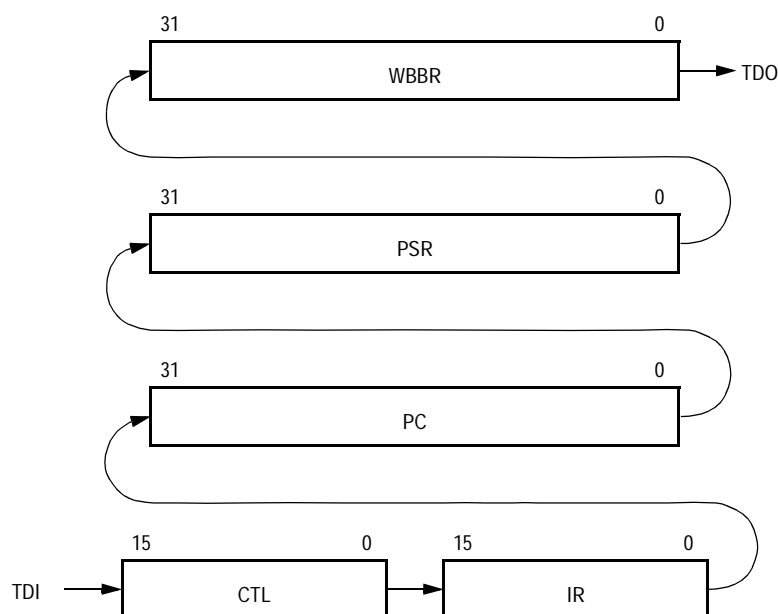
### 8.14.11 Enabling OnCE Memory Breakpoints

When the OnCE memory breakpoint mechanism is enabled with a breakpoint counter value of zero, the device enters debug mode after completing the execution of the instruction that caused the memory breakpoint to occur. In case of breakpoints on instruction fetches, the breakpoint is acknowledged immediately after the execution of the fetched instruction. In case of breakpoints on data memory addresses,

the breakpoint is acknowledged after the completion of the memory access instruction.

## 8.14.12 Pipeline Information and Write-Back Bus Register

A number of on-chip registers store the CPU pipeline status and are configured in the CPU scan chain register (CPUSCR) for access by the OnCE controller. The CPUSCR is used to restore the pipeline and resume normal device activity upon return from debug mode. The CPUSCR also provides a mechanism for the emulator software to access processor and memory contents. **Figure 8-13** shows the block diagram of the pipeline information registers contained in the CPUSCR.



**Figure 8-13. CPU Scan Chain Register (CPUSCR)**

#### 8.14.12.1 Program Counter Register

The program counter register (PC) is a 32-bit latch that stores the value in the CPU program counter when the device enters debug mode. The CPU PC is affected by operations performed during debug mode and must be restored by the external command controller when the CPU returns to normal mode.

#### 8.14.12.2 Instruction Register

The instruction register (IR) provides a mechanism for controlling the debug session. The IR allows the debug control block to execute selected instructions; the debug control module provides single-step capability.

When scan-out begins, the IR contains the opcode of the next instruction to be executed at the time debug mode was entered. This opcode must be saved in order to resume normal execution at the point debug mode was entered.

On scan-in, the IR can be filled with an opcode selected by debug control software in preparation for exiting debug mode. Selecting appropriate instructions allows a user to examine or change memory locations and processor registers.

Once the debug session is complete and normal processing is to be resumed, the IR can be loaded with the value originally scanned out.

#### 8.14.12.3 Control State Register

The control state register (CTL) is used to set control values when debug mode is exited. On scan-in, this register is used to control specific aspects of the CPU. Certain bits reflect internal processor status and should be restored to their original values.

The CTL register is a 16-bit latch that stores the value of certain internal CPU state variables before debug mode is entered. This register is affected by the operations performed during the debug session and should be restored by the external command controller when returning

to normal mode. In addition to saved internal state variables, the bits are used by emulation firmware to control the debug process.

Reserved bits represent the internal processor state. Restore these bits to their original value after a debug session is completed, for example, when a OnCE command is issued with the GO and EX bits set and not ignored. Set these bits to 1s while instructions are executed during a debug session.

	Bit 15	14	13	12	11	10	9	Bit 8
Read:	RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	RSVD	FFY
Write:								
Reset:								0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	FDB	SZ1	SZ0	TC2	TC1	TC0	RSVD	RSVD
Write:								
Reset:	0	0	0	0	0	0		

**Figure 8-14. Control State Register (CTL)**

## FFY — Feed Forward Y Operand Bit

This control bit is used to force the content of the WBBR to be used as the Y operand value of the first instruction to be executed following an update of the CPUSCR. This gives the debug firmware the capability of updating processor registers by initializing the WBBR with the desired value, setting the FFY bit, and executing a MOV instruction to the desired register.

## FDB — Force Debug Enable Mode Bit

Setting this control bit places the processor in debug enable mode. In debug enable mode, execution of the BKPT instruction as well as recognition of the  $\overline{\text{BRKRQ}}$  input causes the processor to enter debug mode, as if the  $\overline{\text{DBGRRQ}}$  input had been asserted.

## SZ1 and SZ0 — Prefetch Size Field

This control field is used to drive the CPU SZ1 and SZ0 outputs on the first instruction pre-fetch caused by issuing a OnCE command with the GO bit set and not ignored. It should be set to indicate a 16-bit size, for example, 0b10. This field should be restored to its

original value after a debug session is completed, for example, when a OnCE command is issued with the GO and EX bits set and not ignored.

#### TC — Prefetch Transfer Code

This control field is used to drive the CPU TC2–TC0 outputs on the first instruction pre-fetch caused by issuing a OnCE command with the GO bit set and not ignored. It should typically be set to indicate a supervisor instruction access, for example, 0b110. This field should be restored to its original value after a debug session is completed, for example, when a OnCE command is issued with the GO and EX bits set and not ignored.

#### 8.14.12.4 Writeback Bus Register

The writeback bus register (WBBR) is a means of passing operand information between the CPU and the external command controller. Whenever the external command controller needs to read the contents of a register or memory location, it forces the device to execute an instruction that brings that information to WBBR.

For example, to read the content of processor register r0, a MOV r0,r0 instruction is executed, and the result value of the instruction is latched into the WBBR. The contents of WBBR can then be delivered serially to the external command controller.

To update a processor resource, this register is initialized with a data value to be written, and a MOV instruction is executed which uses this value as a write-back data value. The FFY bit in the CTL register forces the value of the WBBR to be substituted for the normal source value of a MOV instruction, thus allowing updates to processor registers to be performed.

#### 8.14.12.5 Processor Status Register

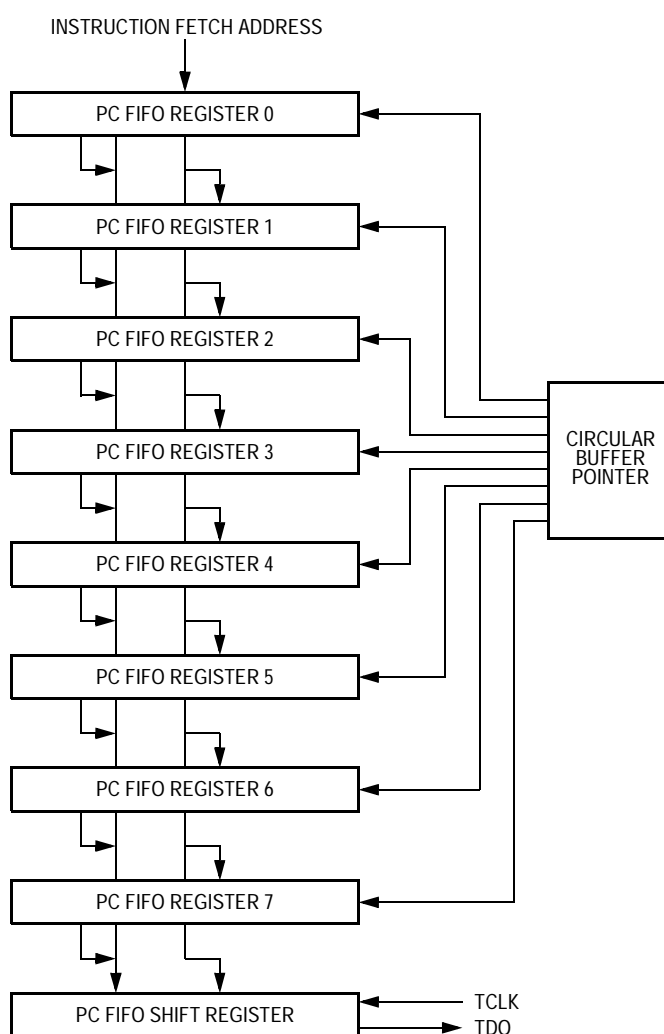
The processor status register (PSR) is a 32-bit latch used to read or write the M•CORE processor status register. Whenever the external command controller needs to save or modify the contents of the M•CORE processor status register, the PSR is used. This register is affected by the operations performed in debug mode and must be restored by the external command controller when returning to normal mode.

## 8.14.13 Instruction Address FIFO Buffer (PC FIFO)

To ease debugging activity and keep track of program flow, a first-in-first-out (FIFO) buffer stores the addresses of the last eight instruction change-of-flow prefetches that were issued.

The FIFO is a circular buffer containing eight 32-bit registers and one 3-bit counter. All the registers have the same address, but any read access to the FIFO address causes the counter to increment and point to the next FIFO register. The registers are serially available to the external command controller through the common FIFO address.

**Figure 8-15** shows the structure of the PC FIFO.



**Figure 8-15. OnCE PC FIFO**

The FIFO is not affected by operations performed in debug mode, except for incrementing the FIFO pointer when the FIFO is read. When debug mode is entered, the FIFO counter points to the FIFO register containing the address of the oldest of the eight change-of-flow pre-fetches. The first FIFO read obtains the oldest address, and the following FIFO reads return the other addresses from the oldest to the newest, in order of execution.

To ensure FIFO coherence, a complete set of eight reads of the FIFO must be performed. Each read increments the FIFO pointer, causing it to point to the next location. After eight reads, the pointer points to the same location as before the start of the read procedure.

#### 8.14.14 Reserved Test Control Registers

The reserved test control registers (MEM\_BIST, FTCCR, and LSRL) are reserved for factory testing.

**CAUTION:** *To prevent damage to the device or system, do not access these registers during normal operation.*

#### 8.14.15 Serial Protocol

The serial protocol permits an efficient means of communication between the OnCE external command controller and the MCU. Before starting any debugging activity, the external command controller must wait for an acknowledgment that the device has entered debug mode. The external command controller communicates with the device by sending 8-bit commands to the OnCE command register and 16 to 128 bits of data to one of the other OnCE registers. Both commands and data are sent or received LSB first. After sending a command, the external command controller must wait for the processor to acknowledge execution of certain commands before it can properly access another OnCE register.

### 8.14.16 OnCE Commands

The OnCE commands can be classified as:

- Read commands (the device delivers the required data)
- Write commands (the device receives data and writes the data in one of the OnCE registers)
- Commands with no associated data transfers

### 8.14.17 Target Site Debug System Requirements

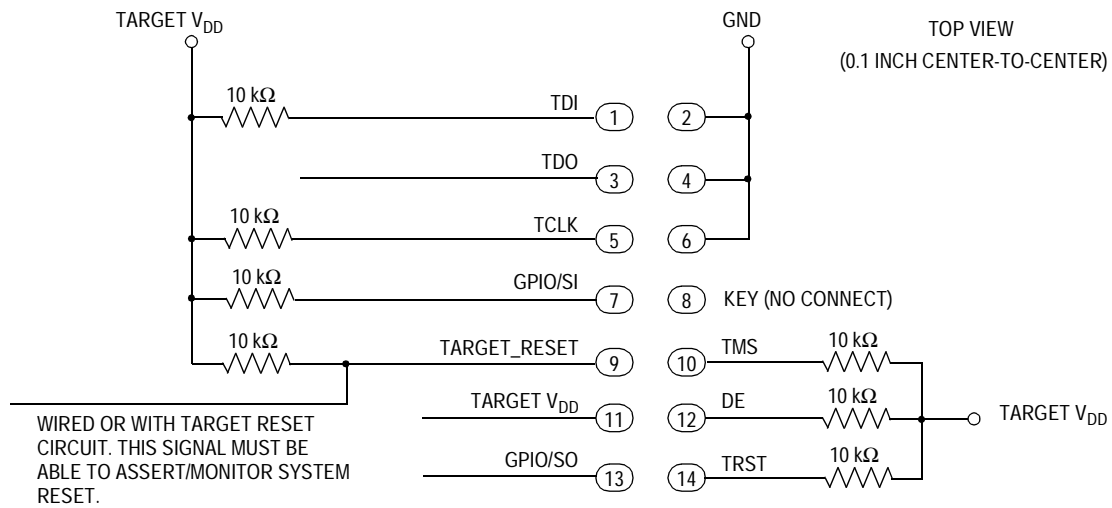
A typical debug environment consists of a target system in which the MCU resides in the user-defined hardware.

The external command controller acts as the medium between the MCU target system and a host computer. The external command controller circuit acts as a serial debug port driver and host computer command interpreter. The controller issues commands based on the host computer inputs from a user interface program which communicates with the user.

### 8.14.18 Interface Connector for JTAG/OnCE Serial Port

**Figure 8-16** shows the recommended connector pinout and interface requirements for debug controllers that access the JTAG/OnCE port. The connector has two rows of seven pins with 0.1-inch center-to-center spacing between pins in each row and each column.





Note: GPIO/SI and GPIO/SO are not required for OnCE operation at this time.  
These pins can be used for high-speed downloads with a recommended interface.

**Figure 8-16. Recommended Connector Interface to JTAG/OnCE Port**



## Appendix A. Nomenclature

### A.1 Contents

A.2	Introduction . . . . .	299
A.3	References . . . . .	299
A.4	Units and Measures . . . . .	299
A.5	Symbology . . . . .	300
A.6	Terminology . . . . .	300

### A.2 Introduction

This section explains the nomenclature used in this manual.

### A.3 References

The M•CORE Technology Center (MTC) uses the *Sematech Official Dictionary* and the JEDEC/EIA *Reference Guide to Letter Symbols for Semiconductor Devices* as references for terminology and symbology.

### A.4 Units and Measures

SIU units and abbreviations are used in MTC technical documentation.

## A.5 Symbology

MTC uses the standard symbols and operators shown in [Table A-1](#).

**Table A-1. Symbols and Operators**

Symbol	Function
+	Addition
-	Subtraction (two's complement) or negation
*	Multiplication
/	Division
>	Greater
<	Less
=	Equal
≥	Equal or greater
≤	Equal or less
≠	Not equal
•	AND
+	Inclusive OR (OR)
⊕	Exclusive OR (EOR)
$\overline{\text{NOT}}$	Complementation
:	Concatenation
⇒	Transferred
↔	Exchanged
±	Tolerance
0b0011	Binary value
0x0F	Hexadecimal value

## A.6 Terminology

**Logic level 1** is a voltage that corresponds to Boolean true (1) state.

**Logic level 0** is a voltage that corresponds to Boolean false (0) state.

To **set** a bit or bits means to establish logic level 1 on them.

To **clear** a bit or bits means to establish logic level 0 on them.

A **signal** is an electronic construct whose state or changes in state convey information.

A **pin** is an external physical connection. The same pin can be used to connect a number of signals.

**Asserted** means that a discrete signal is in active logic state.

- **Active low** signals change from logic level 1 to logic level 0.
- **Active high** signals change from logic level 0 to logic level 1.

**Negated** means that an asserted discrete signal changes logic state.

- **Active low** signals change from logic level 0 to logic level 1.
- **Active high** signals change from logic level 1 to logic level 0.

**LSB** means least significant bit or bits. **MSB** means most significant bit or bits. References to low and high bytes or words are spelled out.

Memory and registers use **Big Endian** ordering. The most significant byte (byte 0) of word 0 is located at address 0.

Bits within a word are numbered downward from the MSB, bit 31.

Signal, bit field, and control bit mnemonics follow this general numbering scheme:

- A **range of mnemonics** is referred to by mnemonic and numbers that define the range, from highest to lowest. For example, ADDR[4:0] are lines four to zero of the address bus.
- A **single mnemonic** stands alone or includes a single numeric designator when appropriate. For example, RST is a unique mnemonic, while ADDR15 represents line 15 of the address bus.



# Appendix B. M210 and M210S Core Instruction Pipeline and Timing

## B.1 Contents

B.2 Introduction . . . . .303

B.3 Instruction Pipeline . . . . .303

B.4 Instruction Execution Time . . . . .305

## B.2 Introduction

This section describes the M210 instruction pipeline and instruction timing information.

## B.3 Instruction Pipeline

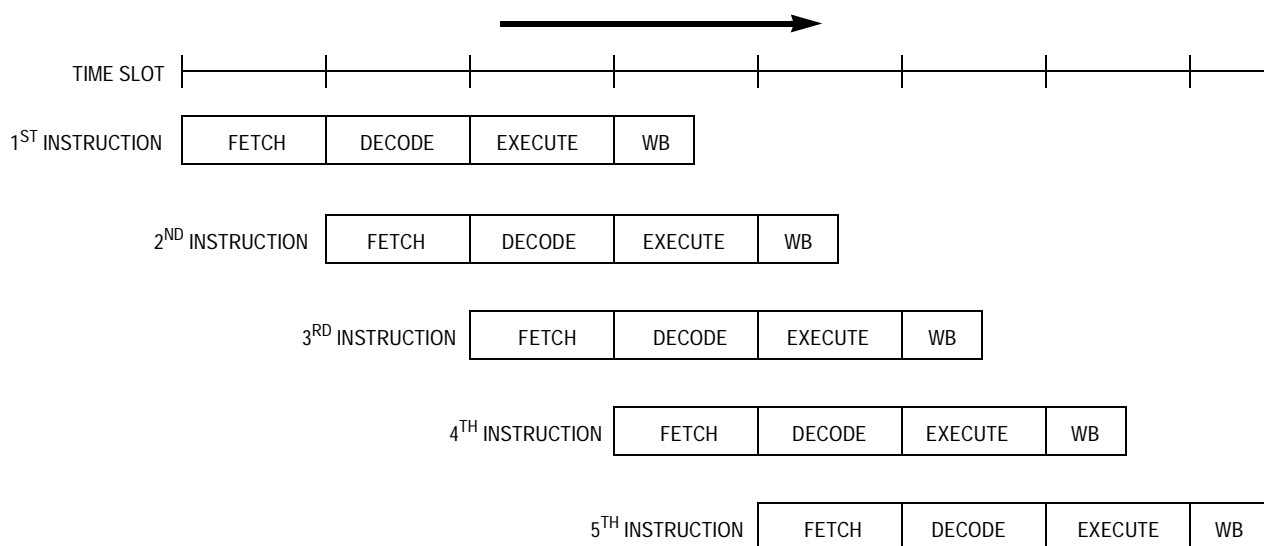
The processor pipeline consists of stages for:

- Instruction fetch
- Instruction decode
- Execution
- Result writeback

Refer to [Figure B-1](#) and [Figure B-2](#). The processor also contains an instruction prefetch buffer to allow buffering of an instruction prior to the decode stage. Instructions proceed from this buffer to the instruction decode stage by entering the instruction decode register IR.

**Figure B-1. Pipeline Stages**

Stage	Description
FETCH	Instruction fetch from memory
DECODE	Instruction decode
EXECUTE / MEM	Instruction execution/memory access
WB	Write back to registers



**Figure B-2. Pipeline Flow**



## B.4 Instruction Execution Time

**Table B-1** contains an instruction execution time and bus access time summary.

**Table B-1. Instruction Execution Time (Sheet 1 of 4)**

Instruction	Instruction Summary	Execution Clocks	Bus Access (Including Instruction Prefetch)
ABS	Absolute value	1	1
ADDC	Unsigned add with c bit, update c bit	1	1
ADDI	Unsigned add with immediate	1	1
ADDU	Unsigned add	1	1
AND	Logical AND	1	1
ANDI	Logical AND with immediate	1	1
ANDN	Logical AND not	1	1
ASR	Arithmetic shift right (dynamic)	1	1
ASRC	Arithmetic shift right by 1 bit, update c bit	1	1
ASRI	Arithmetic shift right immediate (static)	1	1
BCLRI	Bit clear immediate	1	1
BF	Conditional branch if false	1 (not taken)/ 2 (taken)	1 (not taken)/ 2 (taken)
BGENI	Bit generate immediate (static)	1	1
BGENR	Bit generate register (dynamic)	1	1
BKPT	Breakpoint	–	1
BMASKI	Bit mask generate immediate	1	1
BR	Unconditional branch	2	2
BREV	Bit reverse	1	1
BSETI	Bit set immediate	1	1
BSR	Branch to subroutine	2	2
BT	Conditional branch if true	1 (not taken)/ 2 (taken)	1 (not taken)/ 2 (taken)
BTSTI	Bit test immediate; update C bit	1	1
CLRF	Clear if condition false	1	1

# M210 and M210S Core Instruction Pipeline and Timing

**Table B-1. Instruction Execution Time (Sheet 2 of 4)**

Instruction	Instruction Summary	Execution Clocks	Bus Access (Including Instruction Prefetch)
CLRT	Clear if condition true	1	1
CMPHS	Compare for higher or same	1	1
CMPLT	Compare for less than	1	1
CMPLTI	Compare with immediate for less than	1	1
CMPNE	Compare for not equal	1	1
CMPNEI	Compare with immediate for not equal	1	1
DECF	Decrement conditionally on false	1	1
DECGT	Decrement, set C bit on greater than	1	1
DECLT	Decrement, set C bit on less than	1	1
DECNE	Decrement, set C bit on not equal	1	1
DECT	Decrement conditionally on true	1	1
DIVS	Signed divide RX by R1 (32/32)	4 to 38	1
DIVU	Unsigned divide RX by R1 (32/32)	4 to 36	1
DOZE	Enter low power doze mode	—	1
FF1	Find first one in RX	1	1
INCF	Increment RX conditionally on false	1	1
INCT	Increment RX conditionally on true	1	1
IXH	Index half-word	1	1
IXW	Index word	1	1
JMP	Unconditional jump	2	2
JMPI	Unconditional jump indirect	3	3
JSR	Unconditional jump to subroutine	2	2
JSRI	Unconditional jump to subroutine indirect	3	3
LD.[BHW]	Load register from memory	2	2
LDM	Load multiple registers from memory, N = actual number of registers moved	N+1	N+1
LDQ	Load register quadrant from memory	5	5
LOOP	Decrement with C-bit update and branch if condition true	1 (not taken)/ 2 (taken)	1 (not taken)/ 2 (taken)
LRW	Load PC-relative word	2	2

**Table B-1. Instruction Execution Time (Sheet 3 of 4)**

Instruction	Instruction Summary	Execution Clocks	Bus Access (Including Instruction Prefetch)
LSL	Logical shift left (dynamic)	1	1
LSLC	Logical shift left by 1 bit, update C bit	1	1
LSLI	Logical shift left immediate (static)	1	1
LSR	Logical shift right (dynamic)	1	1
LSRC	Logical shift right by 1 bit, update C bit	1	1
LSRI	Logical shift right immediate (static)	1	1
MFCR	Move from control register	1	1
MOV	Logical move	1	1
MOVF	Move RY to RX if condition false	1	1
MOVI	logical move immediate	1	1
MOVT	Move RY to RX if condition true	1	1
MTCR	Move to control register	2	1
MULT	Multiply	3 to 18	1
MVC	Move C bit to register	1	1
MVCV	Move inverted C bit to register	1	1
NOT	Logical NOT	1	1
OR	Logical OR	1	1
RFI	Return from fast interrupt	3	2
ROTLI	Rotate left immediate (static)	1	1
RSUB	Reverse subtract	1	1
RSUBI	Reverse subtract with immediate	1	1
RTE	Return from exception	3	2
SEXTB	Sign extend byte	1	1
SEXTH	Sign extend half-word	1	1
ST.[BHW]	Store register to memory	2	2
STM	Store multiple registers to memory, N = actual number of registers moved	N+1	N+1
STOP	Enter low power stop mode	—	1
STQ	Store register quadrant to memory	5	5
SUBC	Unsigned subtract with C bit; update C bit	1	1

**Table B-1. Instruction Execution Time (Sheet 4 of 4)**

Instruction	Instruction Summary	Execution Clocks	Bus Access (Including Instruction Prefetch)
SUBI	Unsigned subtract with immediate	1	1
SUBU	Unsigned subtract	1	1
SYNC	Synchronize CPU	1	1
TRAP	Unconditional trap to OS	4	3
TST	Test with zero	1	1
TSTNBZ	Test register for no byte equal to zero	1	1
WAIT	Stop execution and wait for interrupt	—	1
XOR	Logical exclusive OR	1	1
XSR	Extended shift right	1	1
XTRB0	Extract byte 0 into R1 and zero-extend	1	1
XTRB1	Extract byte 1 into R1 and zero-extend	1	1
XTRB2	Extract byte 2 into R1 and zero-extend	1	1
XTRB3	Extract byte 3 into R1 and zero-extend	1	1
ZEXTB	Zero extend byte	1	1
ZEXTH	Zero extend half-word	1	1

## Appendix C. M210/M210S Core Interface

### C.1 Contents

C.2	Introduction	310
C.3	M210 Core Interface Overview	311
C.4	MLB Signal Descriptions	317
C.4.1	Bus Signals	317
C.4.1.1	Address Bus (ADDR[22:0])	317
C.4.1.2	Data Bus (DATA[31:0])	317
C.4.1.3	Input Data Bus (DATA <sub>In</sub> [31:0])	317
C.4.1.4	Output Data Bus (DATA <sub>Out</sub> [31:0])	317
C.4.1.5	Data Bus Byte Output Enable (DATA <sub>EN</sub> [3:0])	317
C.4.2	Transfer Control	318
C.4.2.1	Transfer Acknowledge ( $\overline{TA}$ )	318
C.4.2.2	Transfer Error Acknowledge ( $\overline{TEA}$ )	318
C.4.2.3	Transfer Request ( $\overline{TREQ}$ )	318
C.4.2.4	Transfer Busy ( $\overline{TBUSY}$ )	318
C.4.2.5	Transfer Busy Output ( $\overline{TBUSYOUT}$ )	318
C.4.2.6	Transfer Busy Input ( $\overline{TBUSYIN}$ )	319
C.4.2.7	Transfer Abort ( $\overline{ABORT}$ )	319
C.4.3	Transfer Attribute Signals	319
C.4.3.1	Transfer Code (TC[2:0])	319
C.4.3.2	Read/Write (R/ $\overline{W}$ )	320
C.4.3.3	Transfer Size (TSIZ[1:0])	320
C.4.3.4	Sequential Access ( $\overline{SEQ}$ )	320
C.4.4	Translate Control ( $\overline{TE}$ )	320
C.4.5	Data to Address Signal (D2A)	320
C.4.6	Processor Status Signals	321
C.4.6.1	Processor Status (PSTAT[3:0])	321

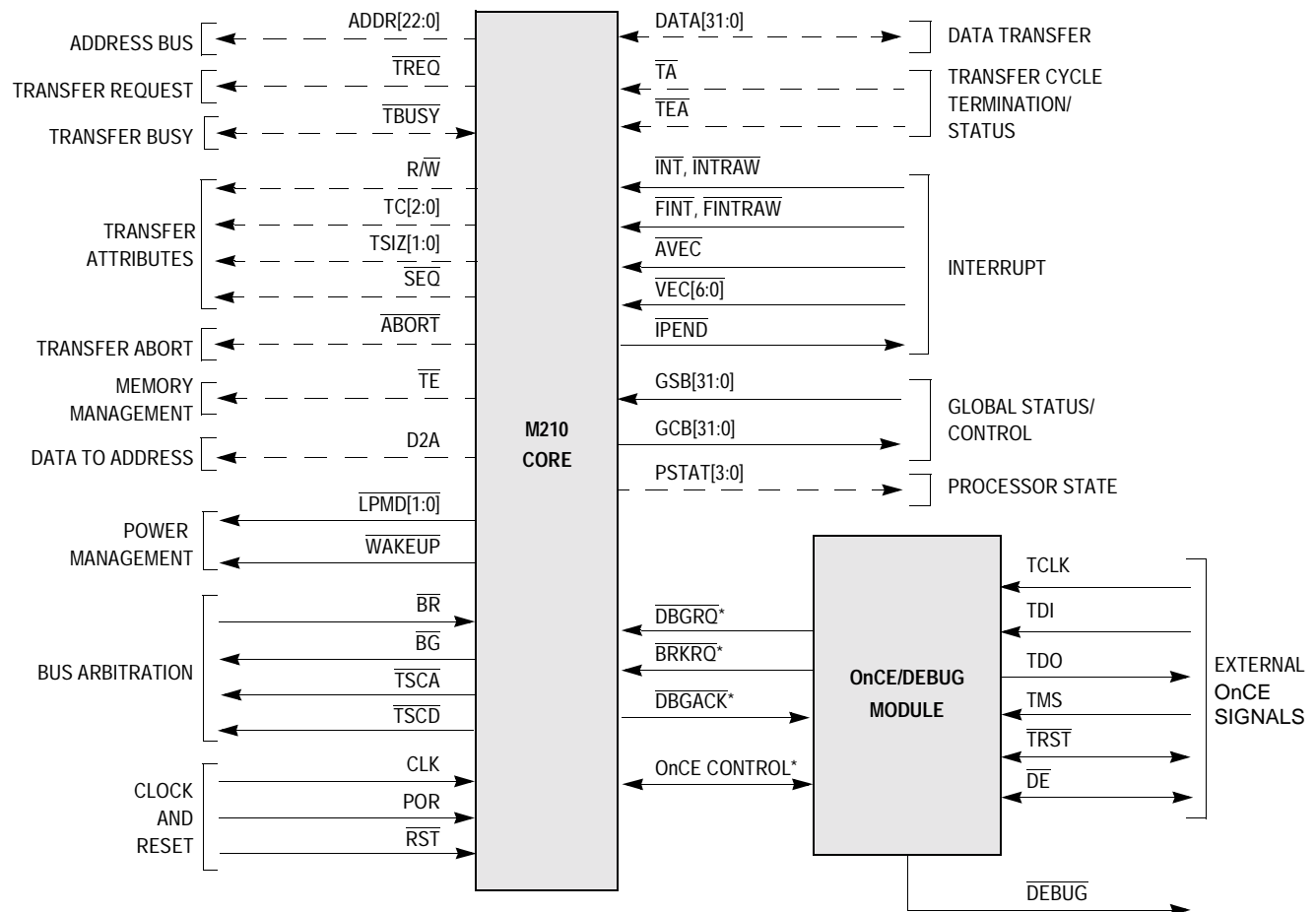
C.5	Other Processor Signals	322
C.5.1	Master Clock (MCLK)	322
C.5.2	Reset Control Signals	322
C.5.2.1	Master Reset ( $\overline{\text{RST}}$ )	322
C.5.2.2	Power-On Reset (POR)	322
C.5.3	Bus Arbitration Control Signals	323
C.5.3.1	Bus Request ( $\overline{\text{BR}}$ )	323
C.5.3.2	Bus Grant ( $\overline{\text{BG}}$ )	323
C.5.3.3	Three-State Control Address ( $\overline{\text{TSCA}}$ )	323
C.5.3.4	Three-State Control Data ( $\overline{\text{TSCD}}$ )	323
C.5.4	Power Management Control Signals	324
C.5.4.1	Low-Power Mode ( $\overline{\text{LPMD}}[1:0]$ )	324
C.5.4.2	Wakeup ( $\overline{\text{WAKEUP}}$ )	325
C.5.5	Global Status and Control Interface Signals	325
C.5.5.1	Global Control ( $\overline{\text{GCB}}[31:0]$ )	325
C.5.5.2	Global Status ( $\overline{\text{GSB}}[31:0]$ )	325
C.5.6	Interrupt Control Signals	326
C.5.6.1	Normal Interrupt Request ( $\overline{\text{INT}}$ )	326
C.5.6.2	Raw Normal Interrupt Request ( $\overline{\text{INTRAW}}$ )	326
C.5.6.3	Fast Interrupt Request ( $\overline{\text{FINT}}$ )	326
C.5.6.4	Raw Fast Interrupt Request ( $\overline{\text{FINTRAW}}$ )	326
C.5.6.5	Interrupt Pending ( $\overline{\text{IPEND}}$ )	326
C.5.6.6	Interrupt Vector Number ( $\overline{\text{VEC}}[6:0]$ )	326
C.5.6.7	Autovector ( $\overline{\text{AVEC}}$ )	327
C.5.7	Power Supply Connections	327

## C.2 Introduction

The M210 core interface consists of the M•CORE local bus (MLB), which supports synchronous data transfers between the processor and other devices in the system, and core-related functions, which include data to address transfer, bus arbitration control, the reset interface, the interrupt interface, the global status and control interface, and the power management interface.

### C.3 M210 Core Interface Overview

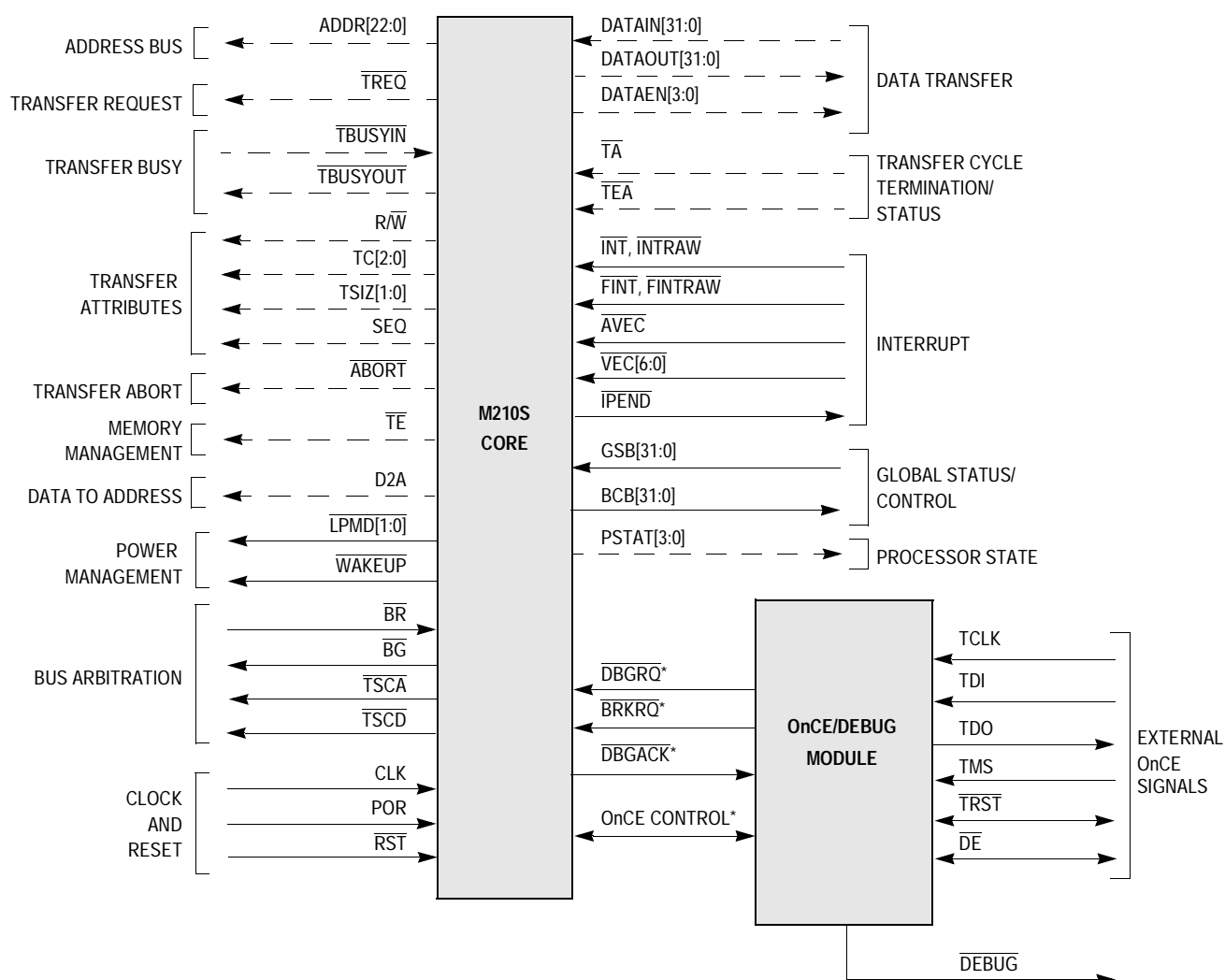
**Figure C-1** shows the functional grouping of M210 core interface signals. **Figure C-2** shows the functional grouping of M210S core interface signals. **Table C-1** provides functional descriptions of M210/M210S signals. **Table C-2** provides a summary of electrical characteristics of M210/M210S signals.



\* These signals are internal to the core  
— — — Indicates MLB signals

**Figure C-1. M210 Core Interface Signals**

# M210/M210S Core Interface



**Figure C-2. M210S Core Interface Signals**



**Table C-1. M210/M210S Signal Descriptions**

Signal Name	Mnemonic	Function
Address bus	ADDR[22:0]	32-bit address bus
Data bus <sup>(1)</sup>	DATA[31:0]	A 32-bit data bus used to transfer up to 32 bits of data per bus transfer.
Input data bus <sup>(2)</sup>	DATA <sub>In</sub> [31:0]	A 32-bit data bus which provides an input data path to the core. It can transfer up to 32 bits of data per bus transfer.
Output data bus <sup>(2)</sup>	DATA <sub>Out</sub> [31:0]	A 32-bit data bus which provides an output data path from the core. It can transfer up to 32 bits of data per bus transfer.
Data bus byte output enable <sup>(2)</sup>	DATA <sub>EN</sub> [3:0]	Byte enables for the output data bus.
Transfer code	TC[2:0]	Indicates the general transfer type: supervisor/user/instruction/data
Read/write	R/ $\overline{W}$	Identifies the transfer as a read or write.
Transfer size	TSIZ[1:0]	Indicates the data transfer size. These signals, together with ADDR[1:0] define the active sections of the data bus.
Data to address	D2A	Indicates the next access address is the value read from the data bus.
Sequential access	$\overline{SEQ}$	Indicates that the next access is sequential.
Transfer busy <sup>(1)</sup>	$\overline{TBUSY}$	Indicates a bus cycle is in progress.
Transfer busy input <sup>(2)</sup>	$\overline{TBUSYIN}$	Alternate master-driven signal indicating that an access is in progress.
Transfer busy output <sup>(2)</sup>	$\overline{TBUSYOUT}$	Core-driven signal indicating that an access is in progress.
Transfer request	$\overline{TREQ}$	Indicates a request for a bus cycle.
Transfer acknowledge	$\overline{TA}$	Asserted to acknowledge a bus transfer.
Transfer error acknowledge	$\overline{TEA}$	Indicates an error condition exists for a bus transfer.
Abort cycle	$\overline{ABORT}$	Aborts a requested access.
Translate control	$\overline{TE}$	Controls address translation or alternate control function.
Bus grant	$\overline{BG}$	Grants interface ownership.
Bus request	$\overline{BR}$	Requests interface ownership.
Three-state address	$\overline{TSCA}$	Can be used to hand off address bus and attributes or as a mux select for these interface signals.

**Table C-1. M210/M210S Signal Descriptions (Continued)**

Signal Name	Mnemonic	Function
Three-state data	$\overline{\text{TSCD}}$	Can be used to hand off data bus and p_tbusy_b or as a mux select for these interface signals.
Reset in	$\overline{\text{RST}}$	Processor reset
	POR	Power-on reset
Normal interrupt request	$\overline{\text{INT}}$	Normal interrupt request to the processor.
Fast interrupt request	$\overline{\text{FINT}}$	Fast interrupt request to the processor.
Normal interrupt request raw	$\overline{\text{INTRAW}}$	Raw normal interrupt request to the processor for $\overline{\text{IPEND}}$ logic.
Fast interrupt request raw	$\overline{\text{FINTRAW}}$	Raw fast interrupt request to the processor for $\overline{\text{IPEND}}$ logic.
Interrupt vector number	$\overline{\text{VEC}}[6:0]$	Interrupt vector number
Autovector	$\overline{\text{AVEC}}$	Requests internal generation of the interrupt vector number.
Interrupt pending	$\overline{\text{IPEND}}$	Indicates an interrupt is pending internally.
System clock	CLK	Clock input
Low-power mode	$\overline{\text{LPMD}}[1:0]$	Indicates low-power mode(s).
Low-power mode wakeup control	$\overline{\text{WAKEUP}}$	Indicates CLK should be activated to exit low-power mode(s) or to allow bus arbitration to occur.
Processor status	PSTAT[3:0]	Processor status outputs
Debug request/open-drain output	$\overline{\text{DE}}$	Signals hardware to enter debug mode.
Internal debug notification	$\overline{\text{DEBUG}}$	Indicates that the processor has entered debug mode.
Debug output	$\overline{\text{DBGACK}}$	CPU debug acknowledge
Internal debug request	$\overline{\text{DBGREQ}}$	Forces an immediate debug request.
Watchpoint event	$\overline{\text{WATCHPOINT}}[1:0]$	Indicates an address watchpoint has been hit.
Hardware breakpoint exception	$\overline{\text{BRKRQ}}$	Internal hardware breakpoint exception
Global control	GCB[31:0]	Global control bus outputs
Global status	GSB[31:0]	Global status bus inputs
Power supply	V <sub>DD</sub>	Power supply
Ground	GND	Ground connection

1. M210 signal only
2. M210S signal only

**Table C-2. M210/M210S Signal Characteristics**

Signal Name	Mnemonic	Input/ Output	Active State	Reset State
Address bus	ADDR[22:0]	Output	High	Undefined
Data bus <sup>(1)</sup>	DATA[31:0]	Input/ output	High	Three-stated
Input data bus <sup>(2)</sup>	DATA <sub>In</sub> [31:0]	Input	High	—
Output data bus <sup>(2)</sup>	DATA <sub>Out</sub> [31:0]	Output	High	Undefined
Data bus byte output enable <sup>(2)</sup>	DATA <sub>EN</sub> [3:0]	Output	High	Negated
Transfer code	TC[2:0]	Output	High	Undefined
Read/write	R/ $\overline{W}$	Output	High/ Low	High
Transfer size	TSIZ[1:0]	Output	High	Undefined
Data to address	D2A	Output	High	Negated
Sequential access	$\overline{SEQ}$	Output	Low	Negated
Transfer busy <sup>(1)</sup>	$\overline{TBUSY}$	Input/ output	Low	Negated
Transfer busy input <sup>(2)</sup>	$\overline{TBUSYIN}$	Input	Low	—
Transfer busy output <sup>(2)</sup>	$\overline{TBUSYOUT}$	Output	Low	Negated
Transfer request	$\overline{TREQ}$	Output	Low	Negated
Transfer acknowledge	$\overline{TA}$	Input	Low	—
Transfer error acknowledge	$\overline{TEA}$	Input	Low	—
Abort cycle	$\overline{ABORT}$	Output	Low	Negated
Translate control	$\overline{TE}$	Output	Low	Negated
Bus grant	$\overline{BG}$	Output	Low	Negated
Bus request	$\overline{BR}$	Input	Low	—
Three-state address	$\overline{TSCA}$	Output	Low	Negated
Three-state data	$\overline{TSCD}$	Output	Low	Negated
Reset in	$\overline{RST}$	Input	Low	—
	POR	Input	High	—
Normal interrupt request	$\overline{INT}$	Input	Low	—

**Table C-2. M210/M210S Signal Characteristics (Continued)**

Signal Name	Mnemonic	Input/ Output	Active State	Reset State
Fast Interrupt Request	$\overline{\text{FINT}}$	Input	Low	—
Normal interrupt request raw	$\overline{\text{INTRAW}}$	Input	Low	—
Fast interrupt request raw	$\overline{\text{FINTRAW}}$	Output	Low	Negated
Interrupt vector number	VEC[6:0]	Input	High	—
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Interrupt pending	$\overline{\text{IPEND}}$	Output	Low	Negated
System clock	CLK	Input	—	—
Low-power mode	$\overline{\text{LPMD}}[1:0]$	Output	Low	Negated
Low-power mode wakeup control	$\overline{\text{WAKEUP}}$	Output	Low	Negated
Processor status	PSTAT[3:0]	Output	High	Undefined
Debug request/open-drain output	$\overline{\text{DE}}$	Input	Low	—
Internal debug notification	$\overline{\text{DEBUG}}$	Input/ output	Low	Negated
CPU debug acknowledge	$\overline{\text{DBGACK}}$	Output	Low	Negated
Internal debug request	$\overline{\text{DBGREQ}}$	Input	Low	—
Watchpoint event	$\overline{\text{WATCHPOINT}}[1:0]$	Output	Low	High
Internal hardware breakpoint exception	$\overline{\text{BRKRQ}}$	Input	Low	—
Global status	GSB[31:0]	Input	High	—
Global control	GCB[31:0]	Output	High	Undefined
Power supply	V <sub>DD</sub>	Input	High	—
Ground	GND	Input	—	—

1. M210 signal only.
2. M210S signal only.

## C.4 MLB Signal Descriptions

This subsection provides descriptions of MLB signals.

### C.4.1 Bus Signals

The address and data bus signals are described here.

#### C.4.1.1 Address Bus ( $ADDR[22:0]$ )

On the M210, these outputs are three-stated. They provide the address for a bus transfer.

On the M210S, these outputs are *not* three-stated. They provide the address for a bus transfer.

#### C.4.1.2 Data Bus ( $DATA[31:0]$ )

On the M210, these three-state bidirectional signals provide the general-purpose data path between the M210 core and peripherals. The data bus can transfer 8, 16, or 32 bits of data per bus transfer.

#### C.4.1.3 Input Data Bus ( $DATA_{In}[31:0]$ )

On the M210S, these input signals provide the data path to the core. The data bus can transfer 8, 16, or 32 bits of data per bus transfer.

#### C.4.1.4 Output Data Bus ( $DATA_{Out}[31:0]$ )

On the M210S, these output signals provide the general-purpose data path from the core. The data bus can transfer 8, 16, or 32 bits of data per bus transfer.

#### C.4.1.5 Data Bus Byte Output Enable ( $DATA_{EN}[3:0]$ )

On the M210S, these output signals are the byte enables for the output data bus. A bidirectional DATA bus can be derived by three-stating

DATA<sub>Out</sub> with DATA<sub>EN</sub> and connecting the output of these three-state buffers to their corresponding DATA<sub>In</sub> signal.

### C.4.2 Transfer Control

The transfer control signals are described here.

#### C.4.2.1 Transfer Acknowledge ( $\overline{TA}$ )

This active-low input signal indicates completion of a requested data transfer operation. An external device asserts  $\overline{TA}$  to terminate the transfer. For the M210 core to accept the transfer,  $\overline{TEA}$  must remain high while  $\overline{TA}$  is asserted.

#### C.4.2.2 Transfer Error Acknowledge ( $\overline{TEA}$ )

This active-low input signal indicates that a transfer error condition has occurred and causes the M210 core to immediately terminate the transfer. An external device asserts  $\overline{TEA}$  to terminate the transfer. The  $\overline{TEA}$  signal has higher precedence than  $\overline{TA}$ .

#### C.4.2.3 Transfer Request ( $\overline{TREQ}$ )

The M210 core normally drives this active-low signal to indicate that a new access has been requested. This signal is driven for a single cycle along with address and transfer attribute signals to request a new cycle.

#### C.4.2.4 Transfer Busy ( $\overline{TBUSY}$ )

On the M210, the core drives this active-low signal to indicate that an access is in progress. This signal is driven for the duration of a cycle, and may be held asserted for multiple transfers.

#### C.4.2.5 Transfer Busy Output ( $\overline{TBUSYOUT}$ )

On the M210S, the core drives this active-low signal to indicate that an access is in progress. This signal is driven for the duration of a cycle, and may be held asserted for multiple transfers.

#### C.4.2.6 Transfer Busy Input ( $\overline{TBUSYIN}$ )

On the M210S, this active-low signal is driven by an alternate master to indicate that an access is in progress. This signal is driven for the duration of a cycle, and may be held asserted for multiple transfers.

#### C.4.2.7 Transfer Abort ( $\overline{ABORT}$ )

The M210 core drives this active-low signal to indicate that a requested access must be aborted. This signal may be driven by the clock following a valid requested cycle. The M210 core must receive the error termination signal ( $\overline{TEA}$ ) from external logic during the same clock cycle  $\overline{ABORT}$  is asserted.

### C.4.3 Transfer Attribute Signals

The transfer attribute signals, which provide additional information about the bus transfer cycle, are described here.

#### C.4.3.1 Transfer Code ( $TC[2:0]$ )

The M210 core drives these signals to indicate the type of access for the current bus cycle. [Table C-3](#) shows the signal encoding.

**Table C-3. Transfer Code Encoding**

TC[2:0]			Transfer Type
0	0	0	User data access <sup>(1)</sup>
0	0	1	Reserved
0	1	0	User instruction access <sup>(2)</sup>
0	1	1	User change of flow instruction access <sup>(3)</sup>
1	0	0	Supervisor data access <sup>(1)</sup>
1	0	1	Supervisor exception vector access
1	1	0	Supervisor instruction access <sup>(2)</sup>
1	1	1	Supervisor change of flow instruction access <sup>(3)</sup>

1. Except **LRW** accesses

2. Except change of flow related instruction accesses, includes **LRW** accesses

3. Change of flow related instruction access for taken branches, jumps, and **LOOPT** instructions (includes table accesses for **JMPI** and **JSRI**)

## C.4.3.2 Read/Write ( $R/\overline{W}$ )

This output signal defines data transfer direction for the current bus cycle. Logic level 1 indicates a read cycle, and logic level 0 indicates a write cycle.

## C.4.3.3 Transfer Size ( $TSIZ[1:0]$ )

These output signals indicate the data size for the bus cycle. [Table C-4](#) shows the definitions of the  $TSIZ[1:0]$  encoding.

**Table C-4. Transfer Size Encoding**

TSIZ[1:0]		Transfer Size
0	0	Word (4 bytes)
0	1	Byte
1	0	Half-word (2 bytes)
1	1	Reserved

## C.4.3.4 Sequential Access ( $\overline{SEQ}$ )

This active-low output signal indicates that the current access is in sequential address order from the last access. This signal is driven for sequential instruction fetches. The timing of this signal is approximately one phase earlier than address timing.

## C.4.4 Translate Control ( $\overline{TE}$ )

This active-low output signal indicates that access addresses can be translated by a memory management unit, or may be used for an alternate function. The  $\overline{TE}$  output is asserted while the PSR TE bit is set.

## C.4.5 Data to Address Signal (D2A)

This active-high output signal indicates that the data received for the current read access will be driven as the next access address. This signal is driven for table accesses for the JMPI and JSRI instructions as well as for cases where load data is to be used as a JUMP or JSR



destination prefetch in the following access. The timing for this control signal is approximately one phase earlier than address timing.

## C.4.6 Processor Status Signals

The signals that provide internal processor status are described here.

### C.4.6.1 Processor Status (PSTAT[3:0])

These outputs indicate the internal execution unit status. The timing is synchronous with the MCLK, so the indicated status may not apply to a current bus transfer. [Table C-5](#) shows PSTAT[3:0] encoding.

**Table C-5. Processor Status Encoding**

PSTAT[3:0]				Internal Processor Status
0	0	0	0	Execution stalled
0	0	0	1	Execution stalled
0	0	1	0	Execute exception
0	0	1	1	Reserved
0	1	0	0	Processor in STOP, WAIT, or DOZE state
0	1	0	1	Execution stalled
0	1	1	0	Processor in debug mode
0	1	1	1	Reserved
1	0	0	0	Launch instruction <sup>(1)</sup>
1	0	0	1	Launch LDM, STM, LDQ, or STQ
1	0	1	0	Reserved
1	0	1	1	Launch LRW
1	1	0	0	Launch branch instruction
1	1	0	1	Launch RTE or RFI
1	1	1	0	Launch JMP or JSR
1	1	1	1	Launch JMPL or JSRL

1. Except **RTE**, **RFI**, **JMPL**, **JSRL**, **LDM**, **STM**, **LDQ**, **STQ**, **LRW**, and change of flow Instructions

### C.5 Other Processor Signals

This subsection provides descriptions of non-MLB core interface signals.

#### C.5.1 Master Clock (MCLK)

The global system clock (MCLK) is used internally to clock the logic of the processor core. The M210 core is a C2, C1-based machine. Since the core is designed for static operation, MCLK can be gated off (forced to logic level one) to lower power dissipation during stopped states. On-chip modules in M•CORE devices use C1 and C2 phase-derivative clocks generated from the master clock signal. The relative timing of these clocks must be the same in all modules.

#### C.5.2 Reset Control Signals

The master, JTAG (Joint Test Action Group), and power-on reset (POR) signals are described here.

##### C.5.2.1 Master Reset ( $\overline{RST}$ )

The  $\overline{RST}$  input is the active-low reset for the M210 processor.  $\overline{RST}$  is considered an asynchronous input and is sampled by the clock control logic in the M•CORE debug module in order to exit from reset gracefully.

##### C.5.2.2 Power-On Reset (POR)

The POR signal is the power-on reset input for the M210 processor. This signal serves two purposes:

1. It prevents M•CORE processor internal bus contention during the short power-on initialization time.
2. POR is “ORed” with TRST and the resulting signal clears the JTAG tap controller, associated registers, and the OnCE state machine. This is an asynchronous clear with a very short assertion time requirement.

### C.5.3 Bus Arbitration Control Signals

The M210 core provides a set of bus arbitration control signals to allow an alternate bus master such as a direct memory access (DMA) device or debug module to perform transfers on the interface. The arbitration protocol uses a simple two wire handshake and also provides three-state control outputs to simplify bus hand off.

#### C.5.3.1 Bus Request ( $\overline{BR}$ )

This active-low input signal is used to request ownership of the interface by an alternate master.

#### C.5.3.2 Bus Grant ( $\overline{BG}$ )

This active-low output signal is used to grant ownership of the interface to an alternate master. If  $\overline{BR}$  is asserted,  $\overline{BG}$  is asserted once the central processor unit (CPU) access pipeline has been drained and the CPU has reached an idle state. Signals which are required by an alternate master are relinquished with or prior to assertion of  $\overline{BG}$ .

#### C.5.3.3 Three-State Control Address ( $\overline{TSCA}$ )

Depending on the implementation, this active-low output signal can be used to enable/disable three-state outputs of an alternate master driving ADDR[22:0] and attributes. It can also be used as a mux select for these interface signals.

#### C.5.3.4 Three-State Control Data ( $\overline{TSCD}$ )

Depending on the implementation, this active-low output signal can be used to enable/disable three-state outputs of an alternate master for driving DATA[31:0] and  $\overline{TBUSY}$ . It can also be used as a mux select for these interface signals.

### C.5.4 Power Management Control Signals

The two output signals which are provided for power management by external control are described here.

#### C.5.4.1 Low-Power Mode ( $\overline{\text{LPMD}}[1:0]$ )

The  $\overline{\text{LPMD}}[1:0]$  outputs are asserted by the processor when execution of a DOZE, STOP, or WAIT instruction occurs, as shown in [Table C-6](#).

**Table C-6. Low-Power Mode Encoding**

$\overline{\text{LPMD}}[1:0]$		Mode
0	0	Stop
0	1	Wait
1	0	Doze
1	1	Normal

The  $\overline{\text{LPMD}}[1:0]$  outputs may assert for one or more clock cycles during the execution of a DOZE, STOP, or WAIT instruction until a valid pending interrupt is detected by the M210 core, or until a request to enter debug mode is made, although a simultaneous debug event or interrupt request may cause the low-power instruction to be exited prior to assertion of the  $\overline{\text{LPMD}}[1:0]$  outputs. External logic can detect the asserted edge of these signals to determine which low-power instruction has been executed and then place the M210 core and peripherals in a low-power consumption state. The  $\overline{\text{IPEND}}$  signal (or  $\overline{\text{WAKEUP}}$ ) can be monitored to determine when to end the stopped condition.

The M210 core can be placed in a low-power state by forcing the CLK input high, and brought out of low-power state by re-enabling CLK.

#### C.5.4.2 Wakeup ( $\overline{\text{WAKEUP}}$ )

This active-low output may be used by external logic to exit the M210 core and system logic from a low-power state.

$\overline{\text{WAKEUP}}$  asserts:

- Whenever a valid pending interrupt is detected by the core
- When a request for bus arbitration is made by an alternate bus master or the alternate bus master still owns the data bus (TSCD) is still asserted)
- When a request to enter debug mode is made via the assertion of the  $\overline{\text{DBUG}}$  input signal.

$\overline{\text{WAKEUP}}$  (or other system state) may be monitored to determine when to release the processor (and system if applicable) from the stopped condition. This can be done by re-enabling CLK.

### C.5.5 Global Status and Control Interface Signals

The core provides two control registers as part of the supervisor programming model to monitor global status in the integrated system as well as to provide global control outputs to the system. This subsection provides information on these registers.

#### C.5.5.1 Global Control ( $\text{GCB}[31:0]$ )

These outputs change state when the global control register (GCR) is updated by the MTCR instruction.

#### C.5.5.2 Global Status ( $\text{GSB}[31:0]$ )

These inputs are sampled by the core and the corresponding values appear in the global status register (GSR) for transfer to a general register when an MFCR instruction referencing the GSR is executed.

### C.5.6 Interrupt Control Signals

This subsection describes the interrupt control function signals.

#### C.5.6.1 Normal Interrupt Request ( $\overline{INT}$ )

This active-low input provides a normal interrupt request condition to the M210 core. This signal is level sensitive.

#### C.5.6.2 Raw Normal Interrupt Request ( $\overline{INTRAW}$ )

This active-low input signal provides an unsynchronized request for normal interrupt service to the M210 core. This signal is level sensitive.

#### C.5.6.3 Fast Interrupt Request ( $\overline{FINT}$ )

This active-low input provides a fast interrupt request condition to the M210 core. This signal is level sensitive.

#### C.5.6.4 Raw Fast Interrupt Request ( $\overline{FINTRAW}$ )

This active-low input signal provides an unsynchronized request for fast interrupt service to the M210 core. This signal is level sensitive.

#### C.5.6.5 Interrupt Pending ( $\overline{IPEND}$ )

This active-low output signal indicates that an interrupt request has been recognized internally by the core and is enabled by the appropriate bit in the PSR. The  $\overline{IPEND}$  signal can be used to signal other bus masters or a bus arbiter that an interrupt condition is pending. External power management logic can use this output to control operation of the core and other logic. External logic uses the wakeup signal similarly.

#### C.5.6.6 Interrupt Vector Number ( $\overline{VEC}[6:0]$ )

These input signals provide the core a vector number to be used when interrupt exception processing begins. These signals are sampled along with the  $\overline{FINT}$  and  $\overline{INT}$  inputs, and must be driven to a valid value when

either of these signals is asserted, unless the  $\overline{AVEC}$  signal is asserted. If  $\overline{AVEC}$  is asserted, these inputs are ignored.

#### C.5.6.7 Autovector ( $\overline{AVEC}$ )

This active-low input signal is asserted with either  $\overline{INT}$  or  $\overline{FINT}$  to request internal generation of the vector number.

### C.5.7 Power Supply Connections

The M210 core requires connections to a  $V_{DD}$  power supply, positive with respect to ground. The  $V_{DD}$  and ground connections must be planned to supply adequate current to the various sections of the processor.





## Appendix D. M210/M210S Interface Operation

### D.1 Contents

D.2	Introduction . . . . .	330
D.3	Bus Characteristics . . . . .	330
D.4	Data Transfer Mechanism . . . . .	331
D.5	Processor Instruction/Data Transfers . . . . .	333
D.5.1	Instruction and Data Read Transfer Cycles . . . . .	334
D.5.2	Read Transfer Cycles with Wait State . . . . .	336
D.5.3	Write Transfer Cycles . . . . .	337
D.5.4	Write Transfer Cycles with Wait State . . . . .	339
D.5.5	Data Bus Hand-Off . . . . .	340
D.6	Bidirectional Three-State Data Bus . . . . .	341
D.7	Bus Exception Control Cycles . . . . .	342
D.8	Bus Errors . . . . .	342
D.9	Abort Signal Operation . . . . .	343
D.10	Data to Address Transfer Operation . . . . .	344
D.11	Breakpoint Request Operation . . . . .	345
D.12	Bus Arbitration Operation . . . . .	346
D.12.1	Operation Examples . . . . .	348
D.12.2	Interaction with Low-Power Modes and Debug Operation . . . . .	360
D.12.3	Bus Arbitration and Entry into Low-Power States . . . . .	360
D.13	Reset Operation . . . . .	362
D.13.1	System Issues . . . . .	363
D.13.2	Timing . . . . .	364
D.14	Interrupt Interface Operation . . . . .	365
D.15	Global Status and Control Interface Operation . . . . .	367
D.16	Power Management Interface Operation . . . . .	367
D.17	Emulation/Debug Interface Operation . . . . .	370

### D.2 Introduction

The M210 core interface supports synchronous data transfers between the processor and other devices in the system. This section provides functional descriptions of:

- The interface
- The signals that control the interface
- The bus cycles provided for data transfer operations
- Power management signals
- Memory management unit control
- Reset operation
- Emulation/debug interface

### D.3 Bus Characteristics

The M210 core interface supports synchronous data transfers between the M210 core and other devices in the system. The CLK is distributed internally to provide logic timing.

The M210 core uses the ADDR[22:0] signals to specify the address for a data transfer and the DATA[31:0] signals to transfer the data.

The M210S core uses the ADDR[22:0] signals to specify the address for a data transfer and the DATA<sub>In</sub>[31:0] or DATA<sub>Out</sub>[31:0] signals to transfer the data.

Control and attribute signals indicate the beginning and type of a bus cycle as well as the address space and size of the transfer. The selected device controls the length of the cycle by terminating it using the control signals.

Inputs to the M210 core (other than the interrupt requests and reset signals) are sampled synchronously and must be stable during the sample windows.

If an input makes a transition during the window time period, the level recognized by the core is not predictable. To guarantee proper

operation, the  $\overline{\text{INT}}$  and  $\overline{\text{FINT}}$  signals are sampled on the rising edge of CLK, but are also used in an asynchronous fashion for power management control. Refer to [7.5.6 Interrupt Control Signals](#) for more information.

Outputs from the M210 core transition on one of the two clock edges depending on the signal class.

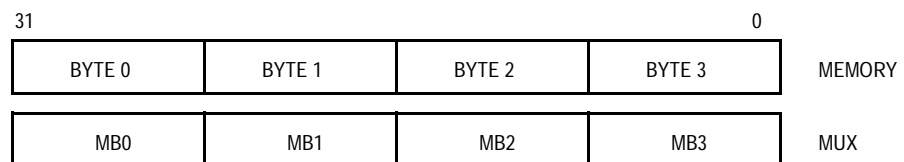
## D.4 Data Transfer Mechanism

Data transfers occur between a core register and the data bus via an internal data multiplexer. The data multiplexer establishes connections for different combinations of address and data sizes.

The core does not support dynamic bus sizing and expects the peripheral being accessed to accept the requested access width. Peripherals with an interface width of N bits must not define internal registers greater than N bits wide.

Additionally, no misaligned transfers are supported. The core can drive the ADDR[1:0] lines to a value which is not representative of an aligned transfer, but expects aligned data to be transferred. For proper function, the TSIZ[1:0] signals must be used to gate ADDR[1:0].

The data multiplexer routes the four bytes of the core data register to properly interface with memory and peripherals connected to the data bus. The mux uses a byte granularity that corresponds to the byte organization in memory, as shown in [Figure D-1](#). Multiplexed core data register bytes are referred to as MB0, MB1, MB2, and MB3

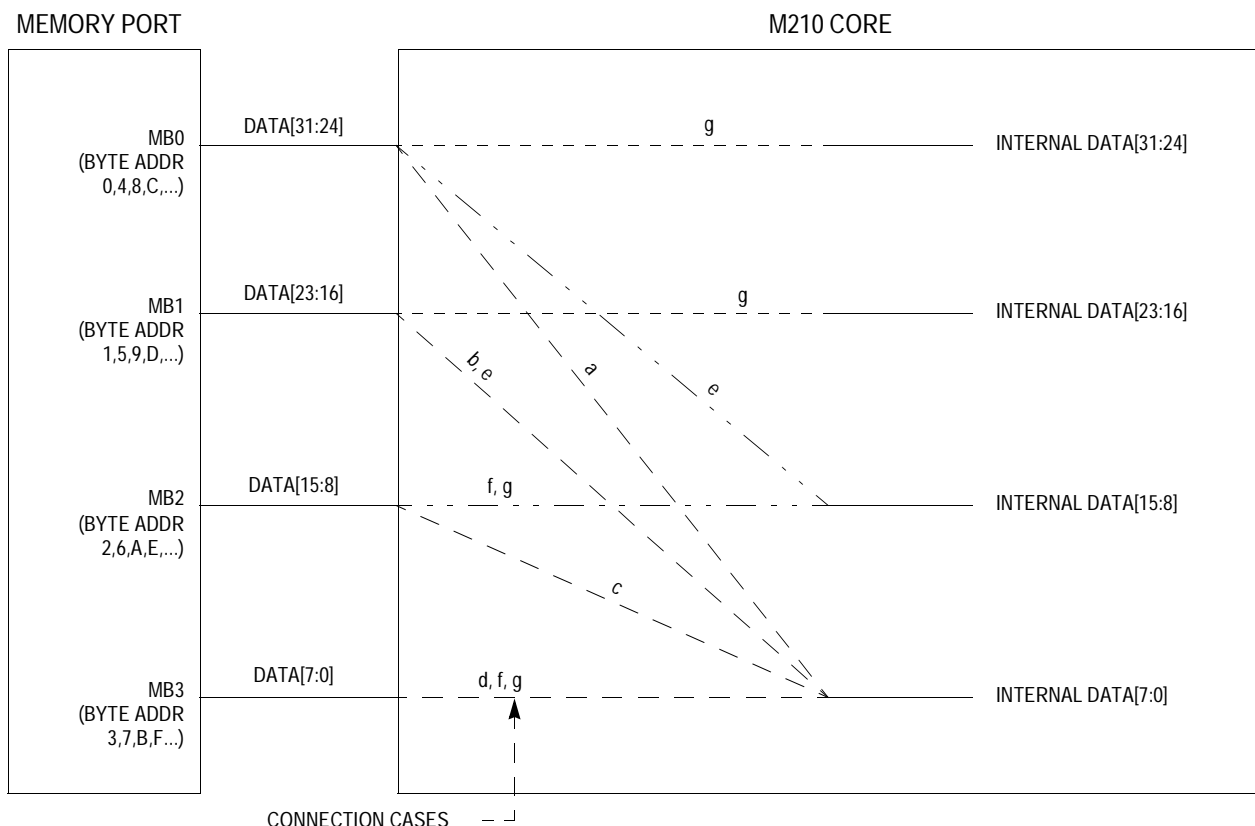


**Figure D-1. Mux Byte Organization**

## M210/M210S Interface Operation

Multiplexed bytes correspond to data bus bytes for 32-bit word transfers, but can be routed differently to support byte and half-word transfers. For instance, MB0 is normally routed to DATA[31:24] on a word transfer, but can be routed to DATA[31:24] to support a byte data transfer.

**Figure D-2** shows the connection requirements for the multiplexer. The transfer size (TSIZ[1:0]) and byte offset (ADDR[1:0]) signals indicate byte position. **Table D-1** shows each possible transfer size and alignment, and the corresponding multiplexer routing. MB0–MB3 indicates the portion of the requested operand that is read or written during that bus transfer. For word transfers, all bytes are valid. Bytes labeled “—” are not required; they are ignored on read transfers and driven with undefined data on write transfers.



**Figure D-2. Internal Multiplexer Connections**

**Table D-1. Interface Requirements for Read and Write Cycles**

Transfer Size	Signal Encoding				Active Interface Bus Sections				Mux Connection
	TSIZ[1:0]		ADDR[1:0]		Internal DATA[31:24]	Internal DATA[23:16]	Internal DATA[15:8]	Internal DATA[7:0]	
Byte	0	1	0	0	—	—	—	MB0	a
	0	1	0	1	—	—	—	MB1	b
	0	1	1	0	—	—	—	MB2	c
	0	1	1	1	—	—	—	MB3	d
Half-word	1	0	0	X	—	—	MB0	MB1	e
	1	0	1	X	—	—	MB2	MB3	f
Word	0	0	X	X	MB0	MB1	MB2	MB3	g

This is perhaps best understood by considering it from a read perspective. Data from 8-bit or 16-bit reads is routed to the same location in the processor data register, regardless of the internal data lines used to access the data. Designers of on-chip peripherals must be aware of these operating constraints.

## D.5 Processor Instruction/Data Transfers

Transfer of data between the M210 core and peripherals involves the address bus, data bus, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting aligned byte, half-word, and word transfers. All bus input and output signals are sampled or driven with respect to one of the edges of the CLK signal. The M210 core moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

Access requests are generated in an overlapped fashion in order to support sustained single-cycle transfers. In addition, the M210 core may choose to change the request address and attribute values if a previous request is still pending. This might occur if an instruction transfer is not completed in a single cycle, and a data transfer becomes pending. In this case, the data request may replace a pending instruction request.

On the M210, access requests are assumed to be accepted if there are no accesses in progress ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  negated), or if an access in progress is terminated during the same cycle a new request is

generated ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  asserted and one of  $\overline{\text{TA}}$  or  $\overline{\text{TEA}}$  asserted).

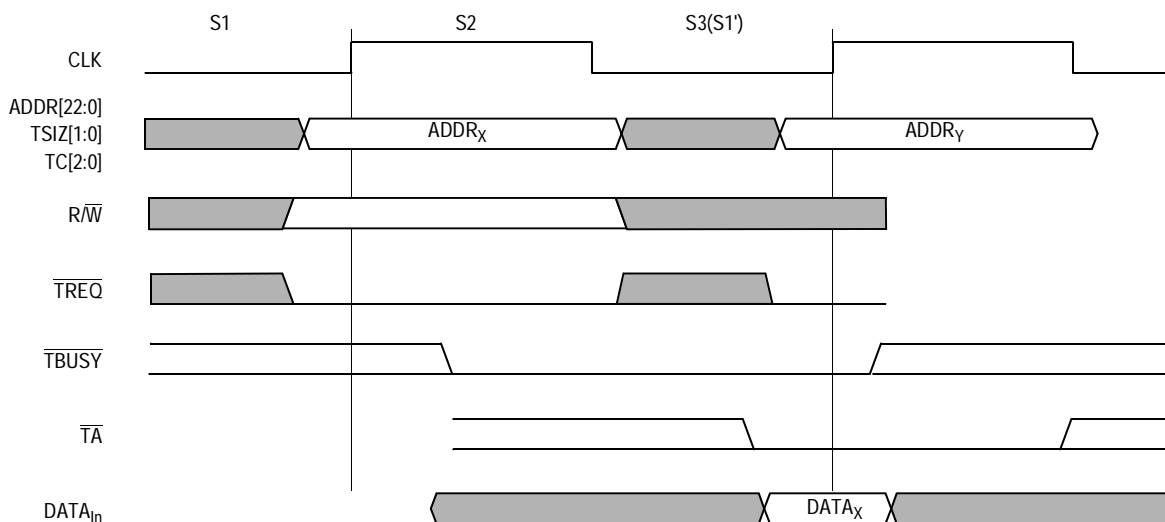
On the M210S, access requests are assumed to be accepted if there are no accesses in progress ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  negated), or if an access in progress is terminated during the same cycle a new request is generated ( $\overline{\text{TREQ}}$  asserted with  $\overline{\text{TBUSY}}$  asserted and one of  $\overline{\text{TA}}$  or  $\overline{\text{TEA}}$  asserted).

Once an access has been accepted, the core is free to change the current request: peripherals may need to latch access information.

The core can also abort an accepted access during the cycle following a valid (taken) request, by asserting  $\overline{\text{ABORT}}$  during the clock cycle following a valid  $\overline{\text{TREQ}}$ . In this case, external logic must terminate the access by asserting  $\overline{\text{TEA}}$  during the same clock cycle  $\overline{\text{ABORT}}$  asserts. In the case of an aborted access, the address bus and all attributes associated with the aborted request are undefined.

## D.5.1 Instruction and Data Read Transfer Cycles

During a read transfer, the core receives data from a memory or peripheral device. **Figure D-3** is a functional timing diagram for instruction and data read transfers



**Figure D-3. Instruction/Data Read Cycle**

### State1 (S1)

The read cycle starts in S1. During S1, the core places valid values on the address bus and transfer attributes. The transfer code (TC[2:0]) signals identify the specific access type. The TSIZ[1:0] signals indicate the size of the transfer. The read/write ( $\overline{R/\overline{W}}$ ) signal is driven high for a read cycle.

The M210 core asserts transfer request ( $\overline{TREQ}$ ) during S1 to indicate that a transfer is being requested.

### State2 (S2)

During S2, the memory access takes place using the values of TSIZ[1:0] and ADDR[1:0], which are driven during S1 and S2 to enable reading of one or more bytes of memory. On the M210, the  $\overline{TBUSY}$  signal is asserted to indicate that an access is in progress. On the M210S, the  $\overline{TBUSY}$  signal is asserted to indicate that an access is in progress.

### State3 (S3)

The memory drives valid data to the core in S3. The interface control logic uses the values of TSIZ[1:0] and ADDR[1:0], which were driven during S1 and S2 to place information on the data bus. If the memory can respond without a wait state, then the transfer acknowledge  $\overline{TA}$  signal is asserted.

The M210 core samples the level of  $\overline{TA}$ . If it is asserted, the current value is latched from the data bus, the bus cycle terminates, and the data is passed to the appropriate unit of the core. If the M210 core does not recognize assertion of  $\overline{TA}$  by the end of the clock cycle, it ignores the data and inserts a wait state. The M210 core continues to sample  $\overline{TA}$  on successive rising edges of CLK until  $\overline{TA}$  is recognized “asserted.” Only when  $\overline{TA}$  is recognized “asserted” is the outstanding transfer terminated.

During S3, the M210 core may negate  $\overline{TREQ}$  if no further transfers are pending, or may keep  $\overline{TREQ}$  asserted to indicate that another transfer is pending. The  $\overline{R/\overline{W}}$ , TSIZ[1:0], TC[2:0] and ADDR[22:0] signals are driven with the information for the new pending cycle. If no cycle is pending, the values driven during S3 are undefined.

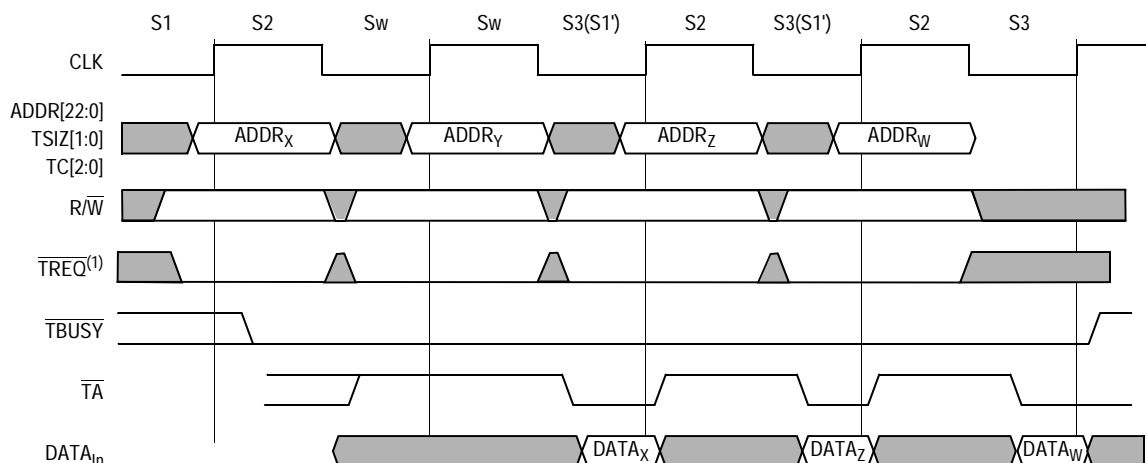
For back-to-back transfers, S3 and the next S1 occur at the same time.

## D.5.2 Read Transfer Cycles with Wait State

**Figure D-4** shows an example of wait state operation. Signal  $\overline{TA}$  for the first request ( $ADDR_X$ ) is not asserted following S2, so wait states ( $Sw$ ) are inserted until  $\overline{TA}$  is recognized.

Meanwhile, another request is generated by the M210 core for  $ADDR_Y$ . This request is not considered accepted by the M210 core since the previous transfer has not been terminated, so the M210 core is free to negate or change the request (in this case it changes the request to  $ADDR_Z$ ) on the next cycle.

This situation can occur when a data request becomes pending following an instruction prefetch request which has not been accepted, and in other circumstances. Interface control logic must be cognizant of this protocol. With a transfer in progress, the next request is considered accepted only if assertion of  $\overline{TA}$  (or  $\overline{TEA}$ ) and  $\overline{TREQ}$  occur during the same low phase of CLK.



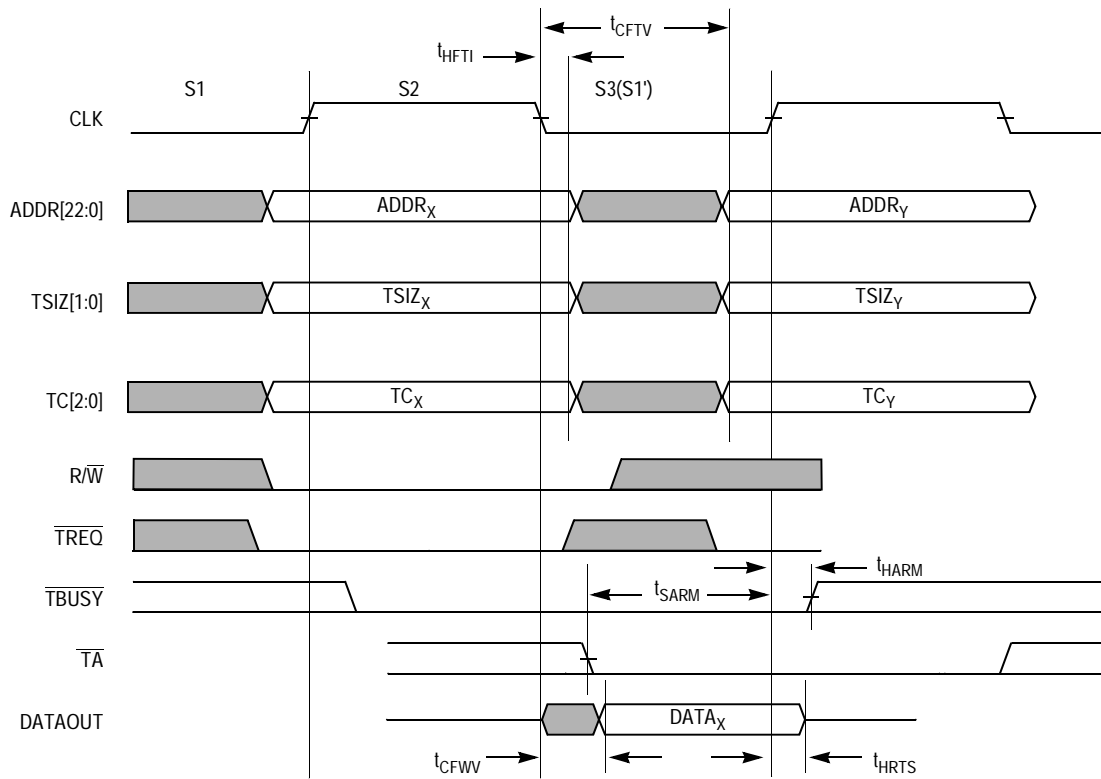
Note 1.  $\overline{TREQ}$  for ADDR<sub>Y</sub> ignored since previous transfer not complete.

**Figure D-4. Read Cycle with Wait States**



### D.5.3 Write Transfer Cycles

During a write transfer, the M210 core drives data to a memory or peripheral device. **Figure D-5** shows an example of a write transfer operation.



**Figure D-5. Write Cycle**

#### State1 (S1)

The write cycle starts in S1. During S1, the M210 core places valid values on the address bus and transfer attributes. The transfer code (TC[2:0]) signals identify the specific access type. The TSIZ[1:0] signals indicate the size of the transfer. The read/write (R/W) signal is driven low for a write cycle.

The M210 core asserts transfer request ( $\overline{TREQ}$ ) during S1 to indicate that a transfer is being requested.

### State2 (S2)

The memory or device access begins in S2. The selected device uses  $R/\overline{W}$ ,  $TSIZ[1:0]$ , and  $ADDR[1:0]$  to select the appropriate bytes to be written in S3. On the M210, the  $\overline{TBUSY}$  signal is asserted to indicate that an access is in progress. On the M210S, the  $\overline{TBUSYOUT}$  signal is asserted to indicate that an access is in progress.

### State3 (S3)

During S3, the M210 core drives the data bus with the data to be written. The interface control logic uses the values of  $R/\overline{W}$ ,  $TSIZ[1:0]$ , and  $ADDR[1:0]$  which were driven during S1 and S2 to align information from the data bus. With the exception of the  $R/\overline{W}$  signal, these signals also select the operand bytes ( $DATA[31:24]$ ,  $DATA[23:16]$ ,  $DATA[15:8]$ , and  $DATA[7:0]$ ). If the memory can respond without a wait state, then it asserts the transfer acknowledge ( $\overline{TA}$ ) signal.

The M210 core samples the level of  $\overline{TA}$  and if it is asserted, terminates the bus cycle. If the M210 core does not recognize assertion of  $\overline{TA}$  by the end of the clock cycle, it inserts a wait state instead of terminating the transfer. The M210 core continues to sample  $\overline{TA}$  on successive rising edges of CLK until  $\overline{TA}$  is recognized asserted. Only when the M210 core recognizes assertion of  $\overline{TA}$  is the outstanding transfer terminated.

During S3, the M210 core may negate  $\overline{TREQ}$  if no further transfers are pending, or may keep  $\overline{TREQ}$  asserted to indicate that another transfer is pending. The  $R/\overline{W}$ ,  $TSIZ[1:0]$ ,  $TC[2:0]$ , and  $ADDR[22:0]$  signals are driven with the information for the new pending cycle. If no cycle is pending, the values driven during S3 are undefined.

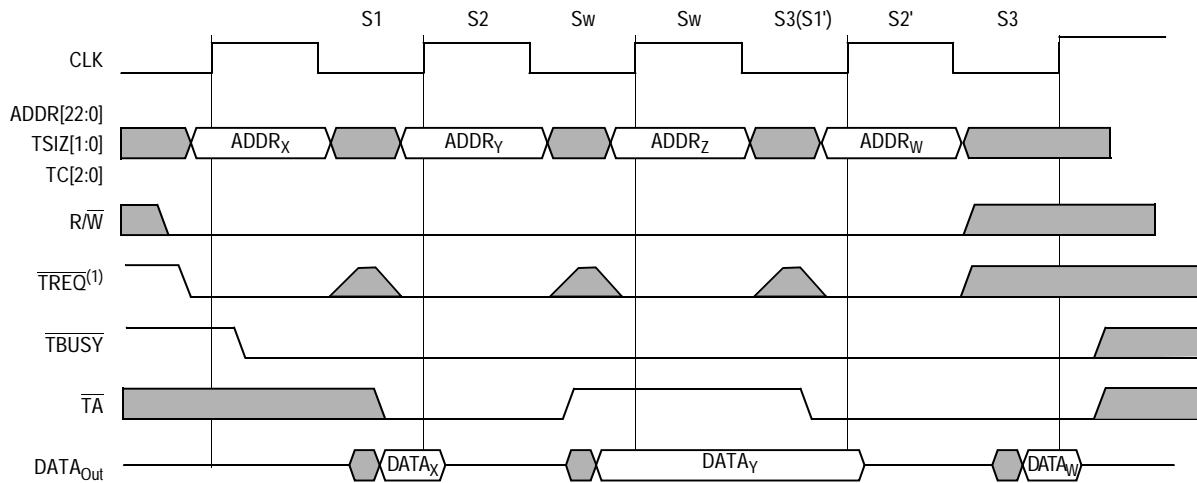
For back-to-back transfers, S3 and the next S1 occur at the same time.

#### D.5.4 Write Transfer Cycles with Wait State

**Figure D-6** shows an example of wait state operation. For the second write request ( $\text{ADDR}_Y$ ),  $\overline{\text{TA}}$  is not asserted following  $\text{S}_2$ , so wait states are inserted until  $\overline{\text{TA}}$  is recognized.

Meanwhile, the core generates another request, for  $\text{ADDR}_Z$ . This request is not considered accepted by the core since the previous transfer has not been terminated, so the core is free to negate or change the request (in this case, the request is changed to  $\text{ADDR}_W$ ) on the next cycle.

This situation can occur when a data request becomes pending following an instruction prefetch request which has not been accepted, and in other circumstances. Logic controlling the interface must be cognizant of this protocol. With a transfer in progress, the next request is considered accepted only if assertion of  $\overline{\text{TA}}$  (or  $\overline{\text{TEA}}$ ) and  $\overline{\text{TREQ}}$  occur during the same low phase of CLK. In this example the request for  $\text{ADDR}_Z$  is never accepted nor should it be.



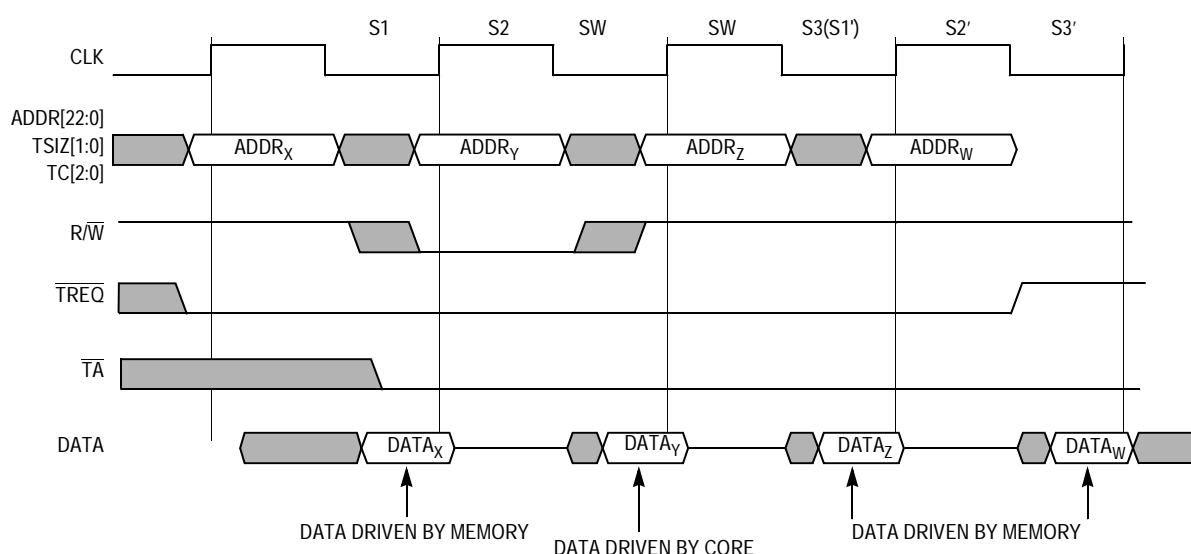
Note 1.  $\overline{\text{TREQ}}$  for  $\text{ADDR}_Z$  ignored since previous transfer not complete.

**Figure D-6. Write Cycle with Wait States**

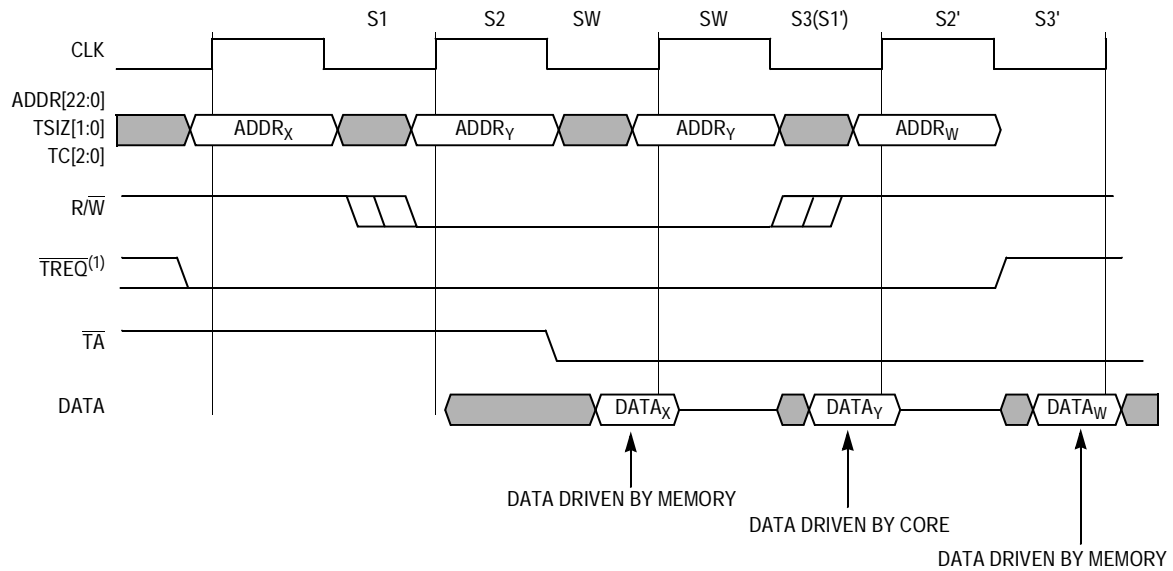
## D.5.5 Data Bus Hand-Off

Two examples of data bus hand-off operation are:

1. In **Figure D-7**, the data bus is driven by either memory or the M210 core during the CLK low phase, and hand off occurs during the CLK high phase. When a requested access changes from a previous read to a write, hand off is performed by three-stating the data bus drivers of memory during the clock high phase following the assertion of  $\overline{TA}$  for the outstanding read cycle. The M210 core is then free to drive write data on the bus during S3 (or the first Sw) for the write request. The M210 core only drives valid data for a single phase when an access is accepted. The memory interface is responsible for either completing the write in this phase, or latching the data.
2. **Figure D-8** shows a read cycle with wait states followed by a write request. Although the M210 core has driven the address and attributes for a write cycle to  $ADDR_Y$ , the data associated with the write cycle is not driven until after the write cycle has been accepted, in this case with the  $\overline{TA}$  for the  $ADDR_X$  access.



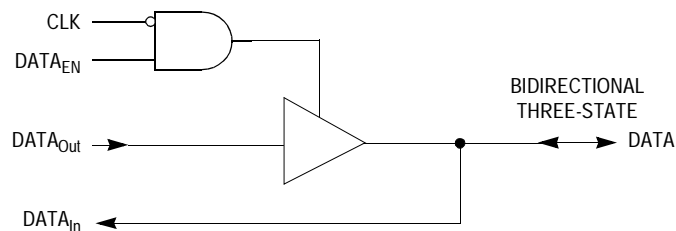
**Figure D-7. Data Bus Hand-Off Operation**



**Figure D-8. Data Bus Hand-Off Operation with Wait State**

## D.6 Bidirectional Three-State Data Bus

The design of the M210S core is well suited for a multiplexed bus implementation with the data bus being split into an input data bus (DATA<sub>In</sub>[31:0]) and an output data bus (DATA<sub>Out</sub>[31:0]). Should the environment in which the core is to be integrated require a single bidirectional data bus, DATA<sub>In</sub>[31:0] and DATA<sub>Out</sub>[31:0] can be combined as shown in [Figure D-9](#).



**Figure D-9. Combining DATA<sub>In</sub> and DATA<sub>Out</sub> Into a Single Bidirectional Data Bus**

## D.7 Bus Exception Control Cycles

The bus interface requires assertion of  $\overline{TA}$  from an external device to signal that a bus cycle is complete.

External circuitry can provide  $\overline{TEA}$  when no device responds or can indicate that an error condition is associated with an access by asserting  $\overline{TEA}$ . This allows the cycle to terminate and the M210 core to enter exception processing for the error condition if appropriate.

To properly control termination of a bus cycle for a bus error condition,  $\overline{TA}$  and  $\overline{TEA}$  must be asserted and negated about the same rising edge of CLK. [Table D-2](#) is a summary of termination results.

**Table D-2. Termination Result Summary**

$\overline{TA}$	$\overline{TEA}$	Result
Don't care	Low	Bus error — terminate, take bus error exception, if appropriate.
Low	High	Normal cycle — terminate and continue
High	High	Insert wait states

## D.8 Bus Errors

The system hardware can use the  $\overline{TEA}$  signal to abort the current bus cycle when a fault is detected. When the M210 core recognizes a bus error condition for an access, the access is terminated immediately.

When a bus cycle is terminated with a bus error, the M210 core can enter access error exception processing immediately following the bus cycle, or it can defer processing the exception.

The instruction prefetch mechanism requests instruction words from the instruction memory unit before it is ready to execute them. If a bus error occurs on an instruction fetch, the M210 core does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch, or should a task switch occur, the access error exception for the unused access does not occur.

A bus error termination for any write access or read access that reference data specifically requested by the execution unit causes the M210 core to begin exception processing immediately.

## D.9 Abort Signal Operation

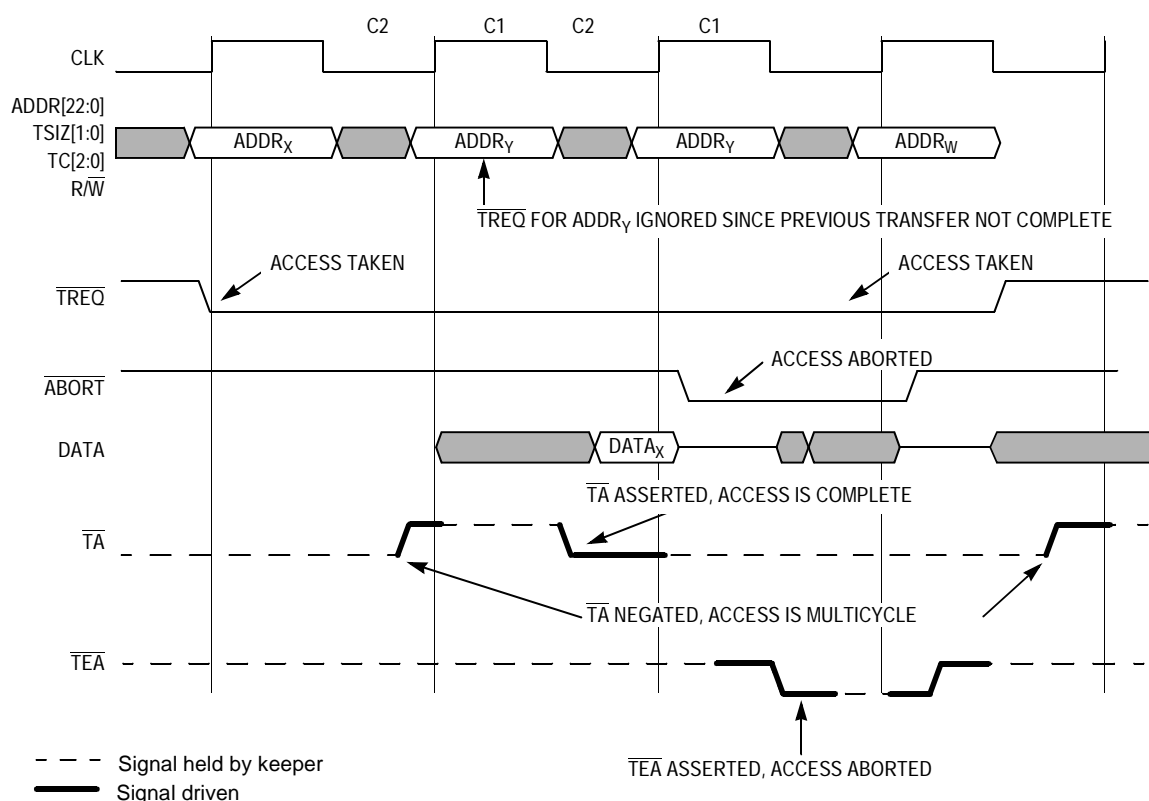
Under certain circumstances involving exception conditions, the central processor unit (CPU) aborts an access in the clock following a valid (taken)  $\overline{TREQ}$  in the previous clock. In this event, the access address is an invalid one and must not be used to access devices. Other circumstances which may cause aborted accesses include:

- When any sort of exception is detected on a taken request
- When  $\overline{TEA}$  occurs on a data access
- When a misaligned exception and the next request is taken
- When a hardware breakpoint occurs during a data transaction with the forced debug enable (FDB) bit in the CPU scan chain control state (CTL) register cleared
- When an interrupt occurs during a load multiple registers (LDM) or store multiple registers (STM) instruction with the interrupt control (IC) bit in the processor status register (PSR) set

Aborted accesses are indicated by the assertion of the  $\overline{ABORT}$  output early in the clock cycle following a taken access. Although the CPU asserts  $\overline{ABORT}$ , it still requires the error termination ( $\overline{TEA}$ ) signal to be asserted, and requires a no wait state response.

**Figure D-10** shows an example of  $\overline{ABORT}$  operation. The access for  $ADDR_Y$  is initially stalled, then aborted.

**NOTE:** *The access for  $ADDR_W$  is valid and taken, even though  $\overline{ABORT}$  has not yet negated, since it is a C1-to-C1 signal.*



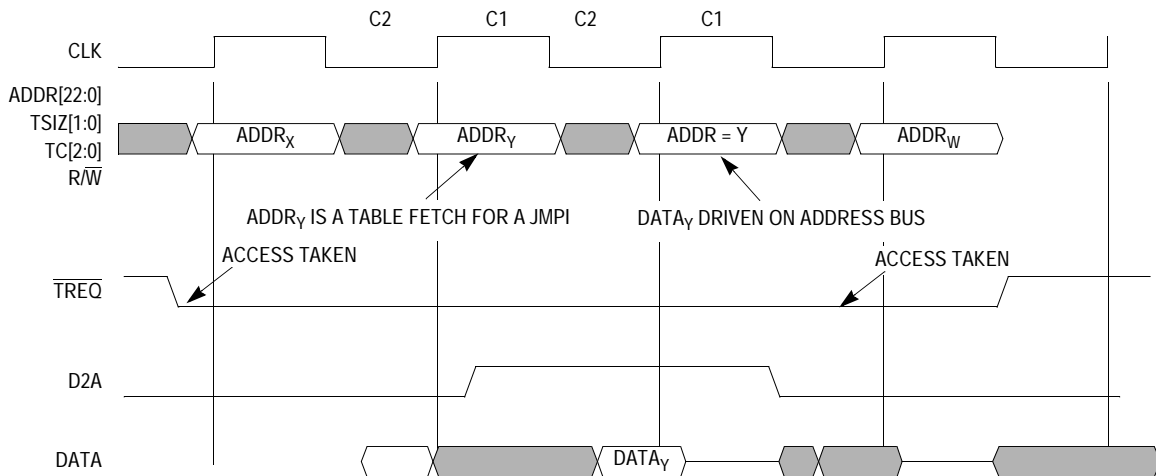
**Figure D-10. Abort Operation**

## D.10 Data to Address Transfer Operation

Under certain circumstances involving instruction change of flow fetches, the CPU immediately forwards data received on a read access to the address bus for the following access. The D2A signal is provided to indicate this occurrence, and may be used to control system behavior as needed for these cases. Usually, no functionality need be modified, but if timing conditions require it, the signal may be used to delay termination of an access in progress, or the succeeding access.

**Figure D-11** shows an example of D2A signal operation. In this example, the access for ADDR<sub>Y</sub> is a table access for a jump indirect (JMPI) or jump to subroutine indirect (JSRI) instruction. The returning data, DATA<sub>Y</sub> is then forwarded to the address bus for the destination prefetch.





**Figure D-11. Data to Address Transfer**

## D.11 Breakpoint Request Operation

The M210 core bus interface supports input signal  $\overline{\text{BRKRQ}}$  to allow accesses to be tagged with breakpoint requests. This signal may also allow the CPU core to enter debug mode via the debug enable (FDB) bit in the CPU scan chain control state (CTL) register. Refer to [Section 9. JTAG Test Access Port and OnCE](#) for more information.  $\overline{\text{BRKRQ}}$  is sampled when an access is terminated with  $\overline{\text{TA}}$  or  $\overline{\text{TEA}}$  to tag an operand or instruction fetch with a breakpoint request.

Operand accesses terminated with  $\overline{\text{BRKRQ}}$  asserted result in a breakpoint exception being taken following completion of the instruction associated with the access. Instruction accesses terminated with  $\overline{\text{BRKRQ}}$  asserted result in breakpoint processing when (and if) the instruction attempts execution.

The  $\overline{\text{BRKRQ}}$  signal does not terminate a bus cycle, it only provides status associated with a cycle. This signal must be valid when the core recognizes a  $\overline{\text{TA}}$  or a  $\overline{\text{TEA}}$  termination.

### D.12 Bus Arbitration Operation

Support for an alternate master to obtain ownership of the interface is provided with the bus arbitration logic block. An external device requests ownership via  $\overline{BR}$ , and is granted ownership by the arbiter with the assertion of  $\overline{BG}$ . Three-state control signals for the address and data buses with their associated signals are provided to allow full bus bandwidth utilization.

Alternate masters request control of the interface by asserting  $\overline{BR}$ . Requests for ownership are granted after initiation of pending CPU accesses. The CPU releases the interface after initiation of outstanding accesses and asserts  $\overline{BG}$ . Two control signals are provided for bus hand off:  $\overline{TSCA}$  (three-state or mux control for ADDR[22:0] and attributes) and  $\overline{TSCD}$  (three-state or mux control for DATA[31:0] and  $\overline{TBUSY}$ ) control signals. These signals indicate the CPU has placed the associated signals in a high-impedance state and are used to enable the three-stateable outputs of the alternate master for low-overhead bus hand off. For a unidirectional bus implementation, these signals can be used to control a mux to select between the core or an alternate master for control of the bus.

The arbitration logic function NOT is available while  $\overline{RST}$  is asserted. Thus, an alternate master may not perform arbitration sequences while the CPU is held in a reset condition. Once  $\overline{RST}$  is released, assertion of  $\overline{BG}$  occurs if  $\overline{BR}$  has been previously asserted and remains asserted.

Maximum latency from assertion of  $\overline{BR}$  to the assertion of  $\overline{BG}$  is one clock cycle, plus the length of any outstanding cycle in progress. Thus, for no-wait state accesses, the maximum latency is two clock cycles, and for an access which requires three cycles to complete, the maximum latency from  $\overline{BR}$  asserted to  $\overline{BG}$  asserted is four clock cycles.

An external master waits for the assertion of  $\overline{BG}$ , after which time the CPU releases the ADDR[22:0],  $R/\overline{W}$ , TSIZ[1:0], TC[2:0],  $\overline{TREQ}$ ,  $\overline{TE}$ ,  $\overline{SEQ}$ , and D2A signals and places these outputs in a high-impedance state.  $\overline{TSCA}$  is asserted one-half clock following assertion of  $\overline{BG}$  to indicate this has occurred. The external master is then responsible for ensuring the correct operation of these signals (including negation, etc. where required). If a cycle is not initiated by the requestor at the time

these signals are handed off, the alternate master must drive  $\overline{TREQ}$  high to negate it.

**NOTE:** *Assertion of  $\overline{BR}$  does not prevent the assertion of  $\overline{TREQ}$  by the CPU during the same clock, and the requested cycle is initiated prior to assertion of  $\overline{BG}$ .*

A requesting device should continue to assert  $\overline{BR}$  past the assertion of  $\overline{BG}$  and  $\overline{TSCA}$  until the alternate cycle is taken. A requested access is taken in a cycle where  $\overline{TREQ}$  is asserted and either  $\overline{TBUSY}$  is negated, or  $\overline{TBUSY}$  is asserted and either  $\overline{TA}$  or  $\overline{TEA}$  asserts to terminate an access in progress.

Since a cycle may still be in progress for the CPU (awaiting  $\overline{TA}$  or  $\overline{TEA}$ , and awaiting returning data for read cycles), the data bus is not relinquished until  $\overline{TSCD}$  is asserted. This may be several cycles for an access in progress with wait states.

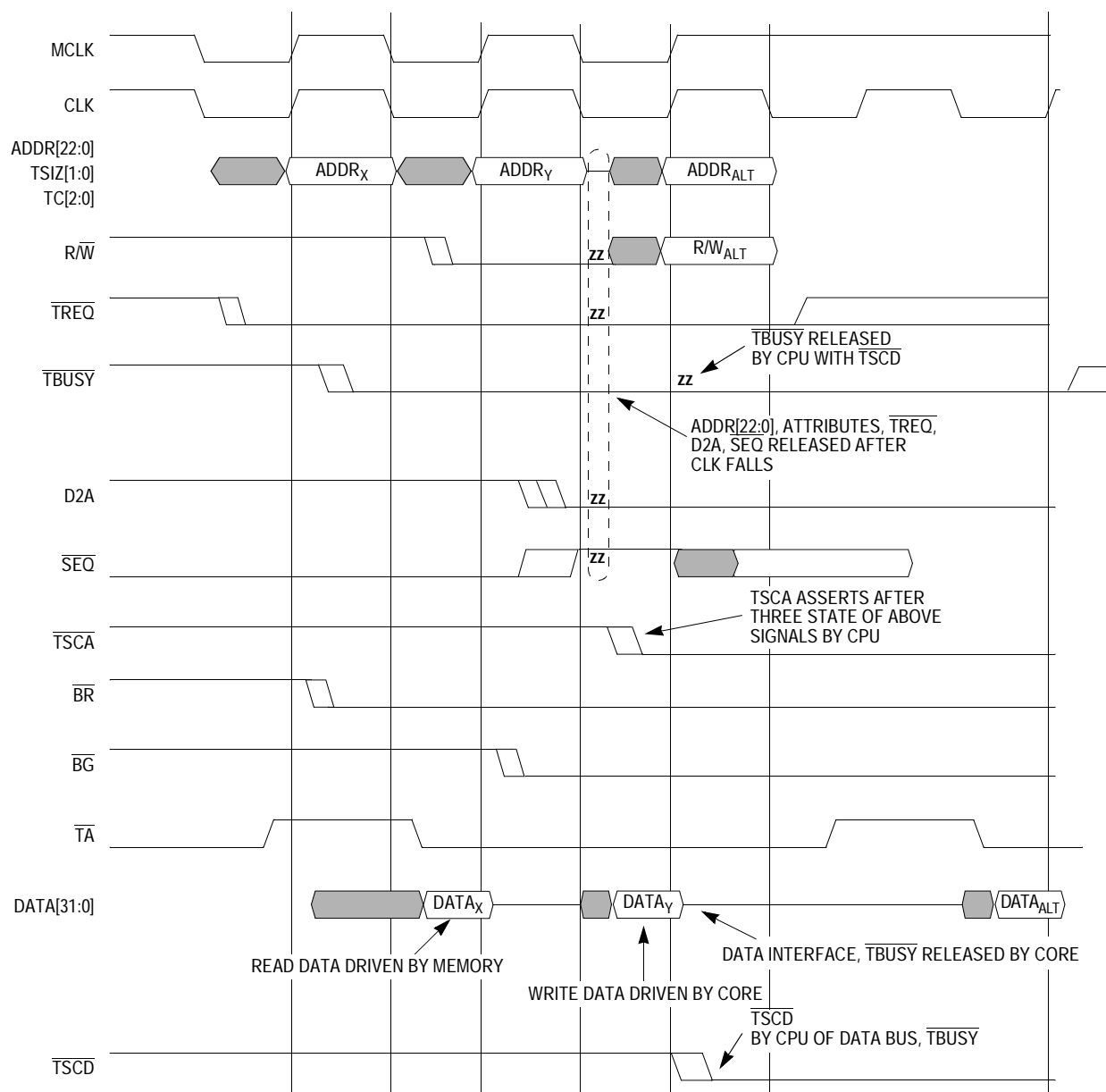
As is the case for  $\overline{TREQ}$  at address bus hand off, the external master is responsible for proper negation of the  $\overline{TBUSY}$  signal at the time of data hand off ( $\overline{TSCD}$  assertion) if an access is not immediately requested.

Once asserted, the bus arbitration logic continues to assert  $\overline{BG}$  until  $\overline{BR}$  is negated. If the  $\overline{BR}$  is negated and then reasserted,  $\overline{BG}$  is negated, and then reasserted after any accesses which might have been initiated by the CPU in the negation interval have completed.

## D.12.1 Operation Examples

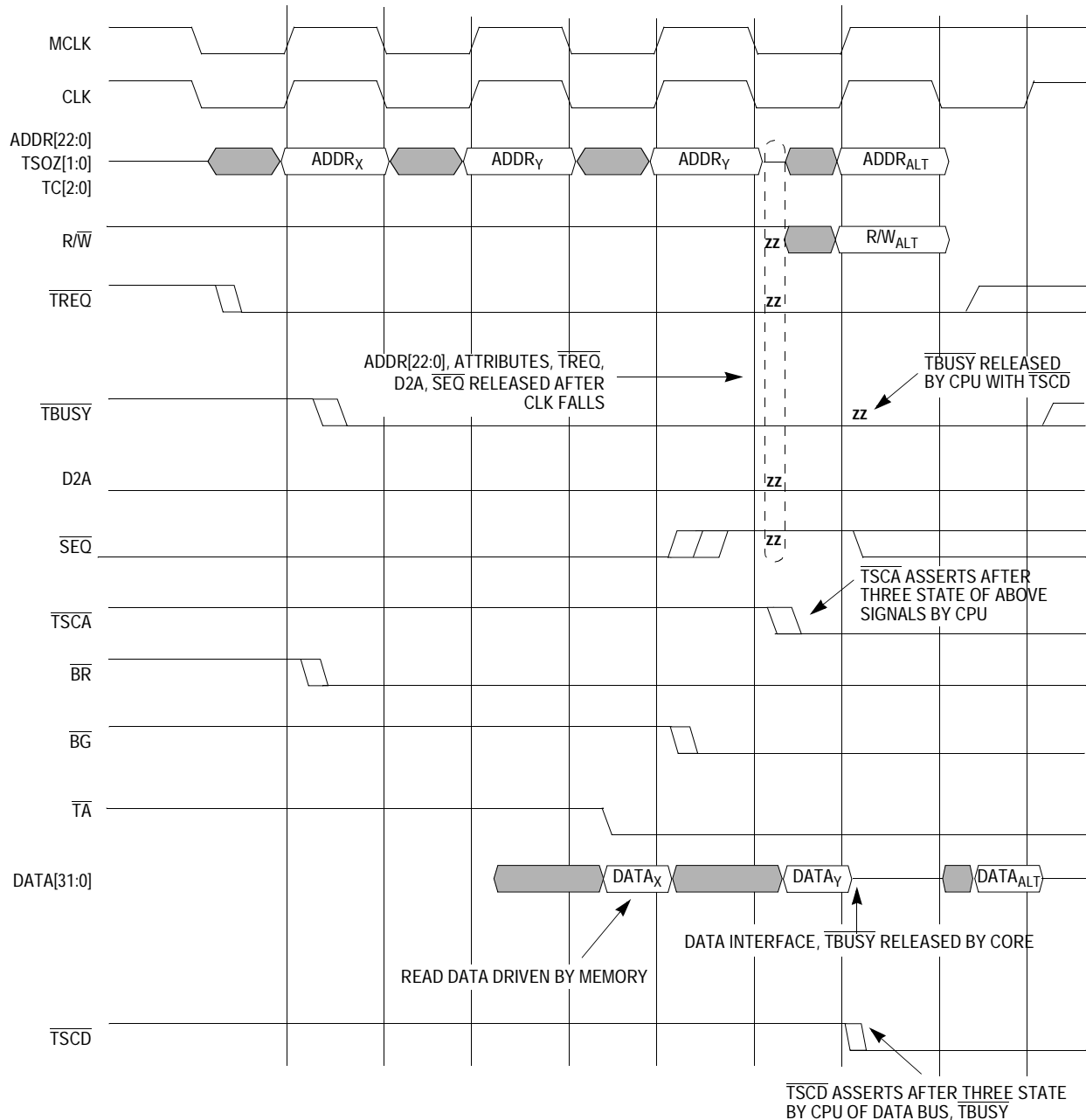
**Figure D-12** shows an example of the CPU relinquishing the interface due to assertion of  $\overline{BR}$ .

**NOTE:** The  $\overline{BG}$  signal is asserted once pending accesses in the CPU pipeline have been initiated.



**Figure D-12. Arbitration Operation, Bus Request → Bus Grant Assertion**

**Figure D-13** shows another example of bus hand off during a read cycle with wait-states. In this case,  $\overline{BG}$  is held off until the pending CPU cycle has been accepted.



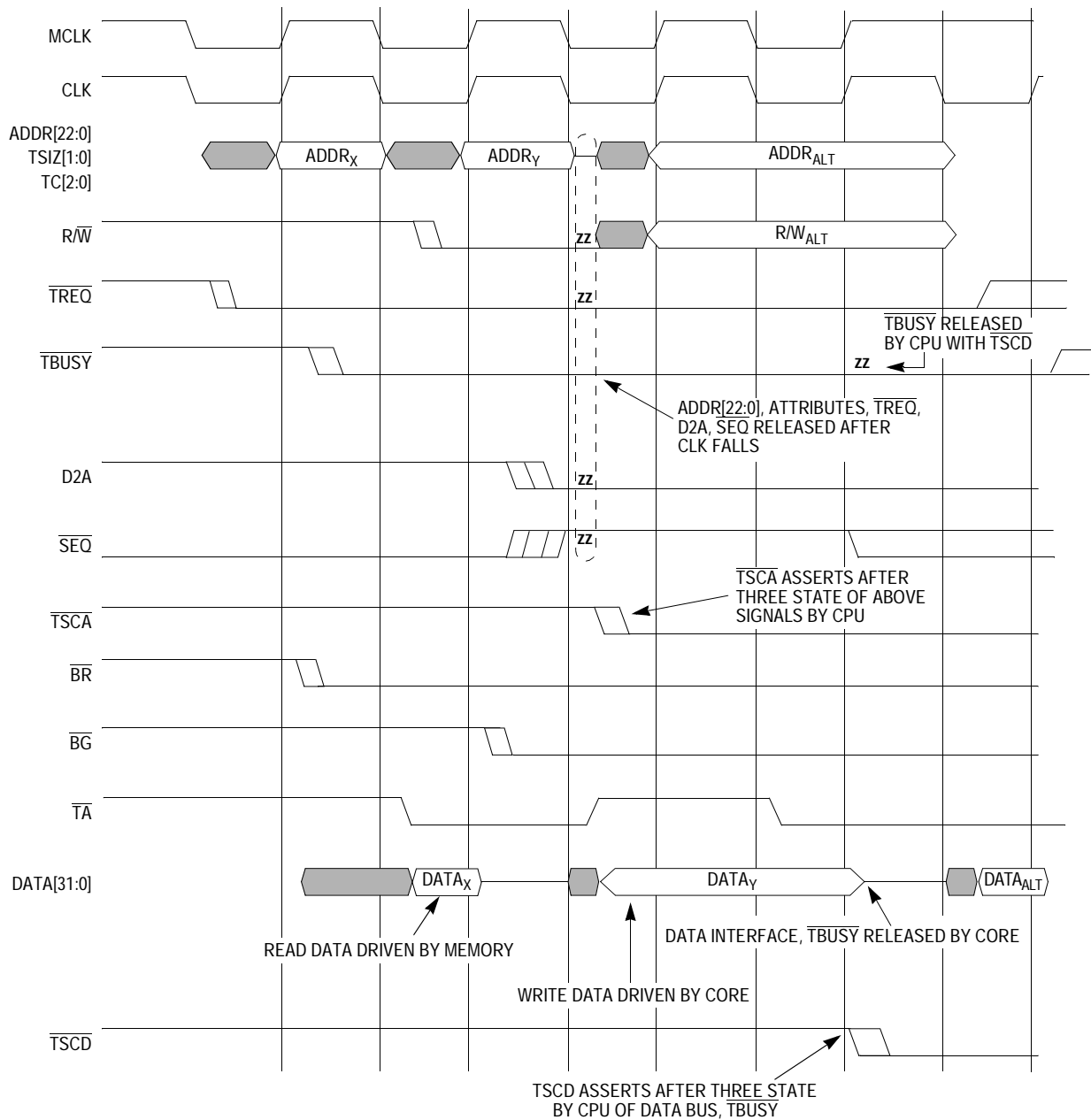
**Figure D-13. Arbitration Operation, Bus Request → Bus Grant Assertion, Wait State on Outstanding Cycle Before Assertion, Assertion Delayed**

**Figure D-14** shows another example of bus hand off during a write cycle with wait states.

**NOTE:** *In this example, since an access is still in progress ( $\overline{TBUSY}$  asserted) when the address and attributes are released to an alternate master (assertion of  $\overline{TSCA}$ ), the alternate master must continue to drive its request and requested address until completion of the data cycle in progress. This is indicated by assertion of  $\overline{TA}$  or  $\overline{TEA}$  while  $\overline{TBUSY}$  is asserted. Alternately, it may be determined by assertion of  $\overline{TSCD}$  if the timing permits.*

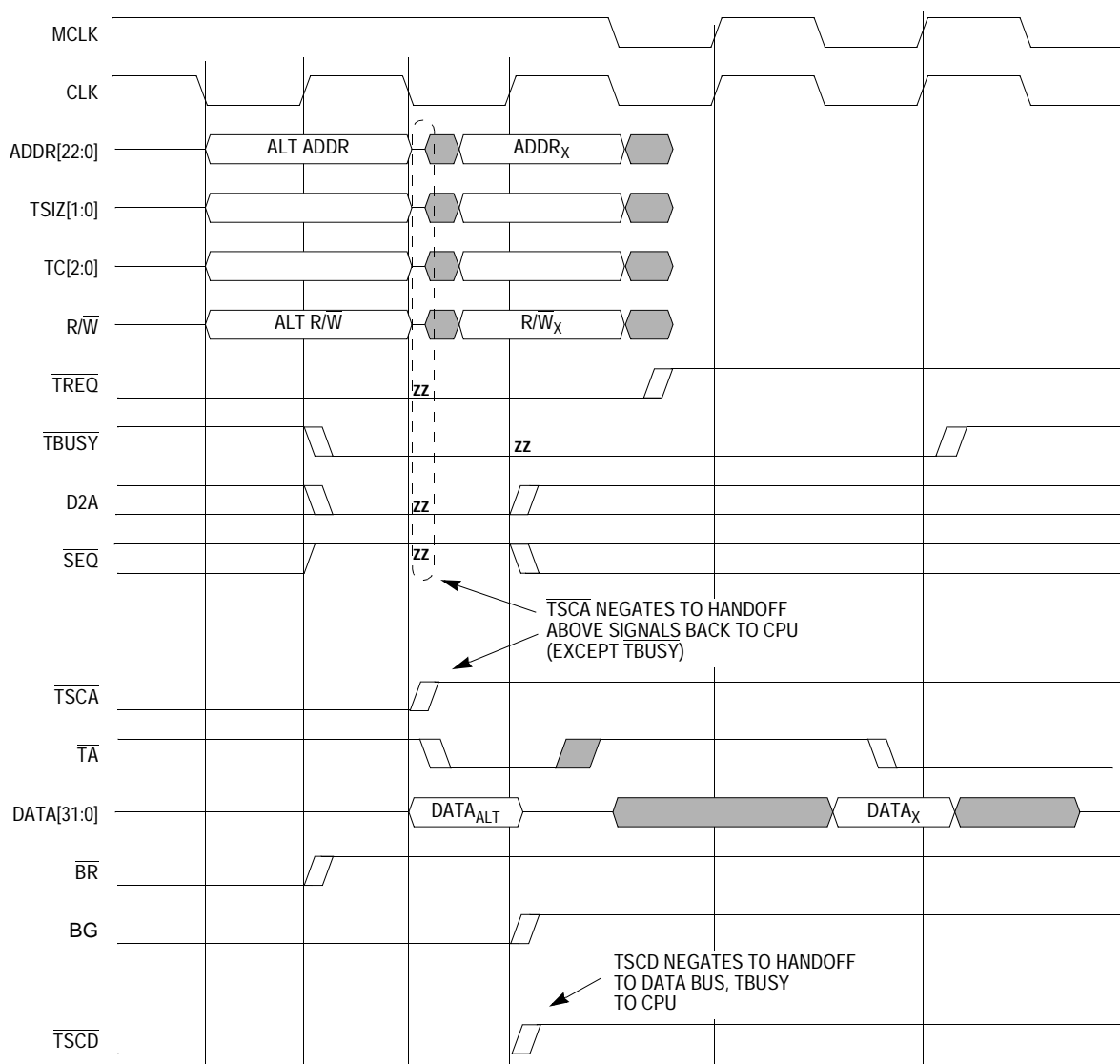
On negation of  $\overline{BR}$ , the CPU negates  $\overline{BG}$  and retakes ownership of the bus. **Figure D-15** through **Figure D-22** show examples of the CPU regaining ownership of the interface due to negation of  $\overline{BR}$ .

**NOTE:** *For proper system operation, the alternate master must ensure that the  $D2A$  and  $\overline{SEQ}$  signals have been negated prior to releasing them.*



**Figure D-14. Arbitration Operation, Bus Request → Bus Grant Assertion, Wait State on Outstanding Cycle After Assertion**

# M210/M210S Interface Operation



**Figure D-15. Arbitration Operation, Bus Request → Bus Grant Negation**



Figure D-16 shows interleaved CPU and alternate master cycles.

**NOTE:** The alternate master negates  $\overline{BR}$  as soon as its requested address has been taken, since it only has a single access to run per request. This ensures no dead time on the address bus.

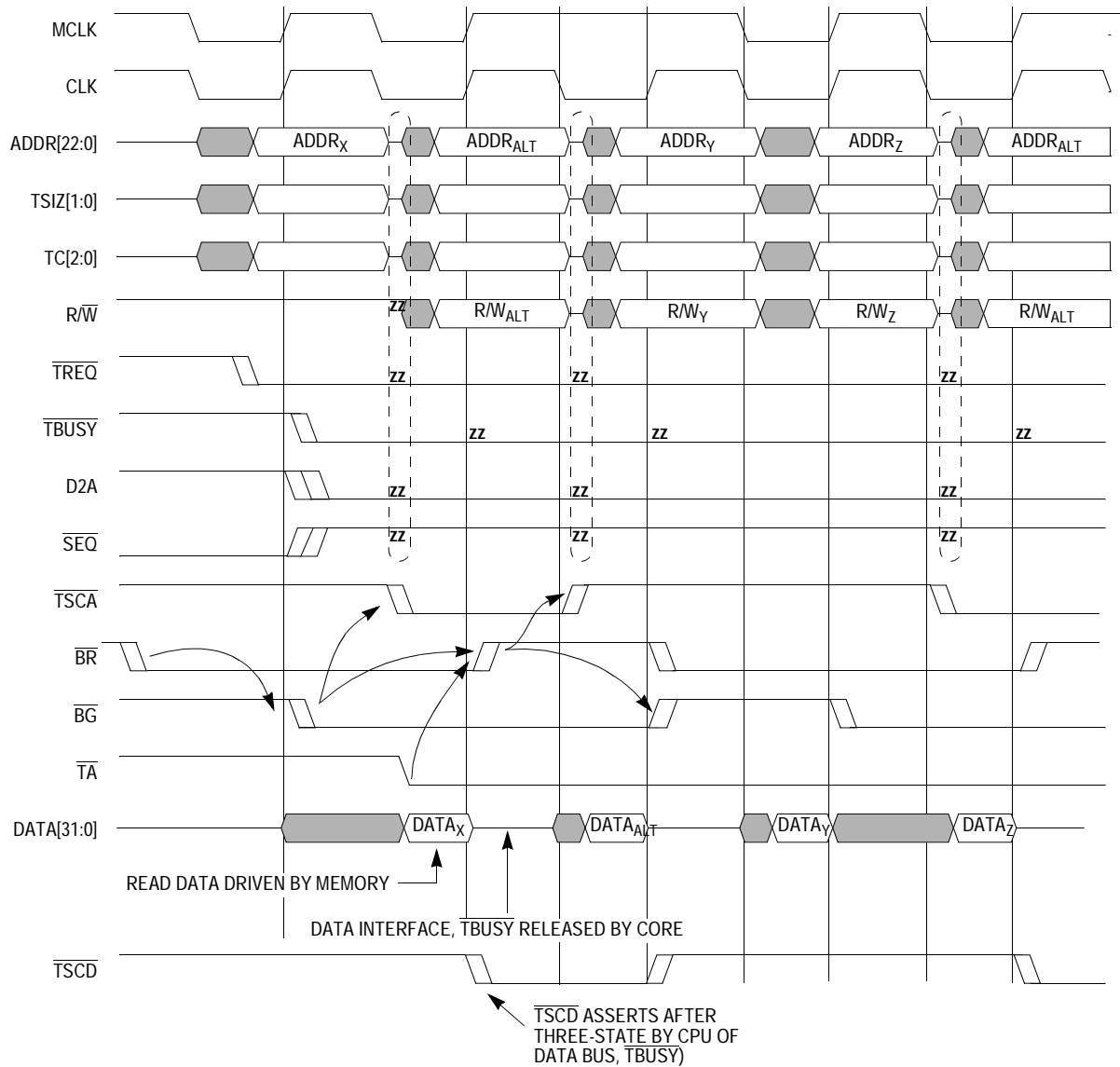
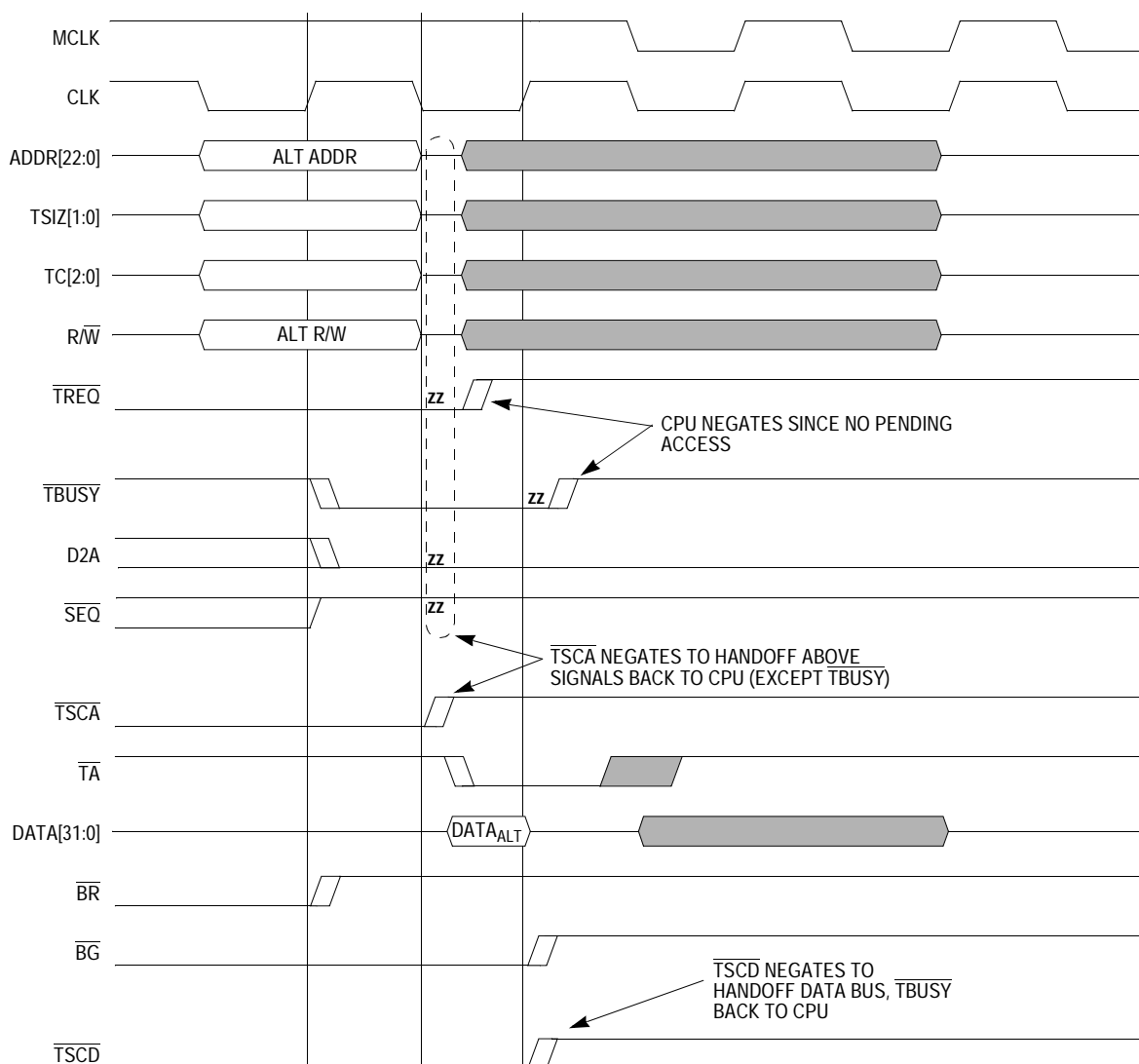


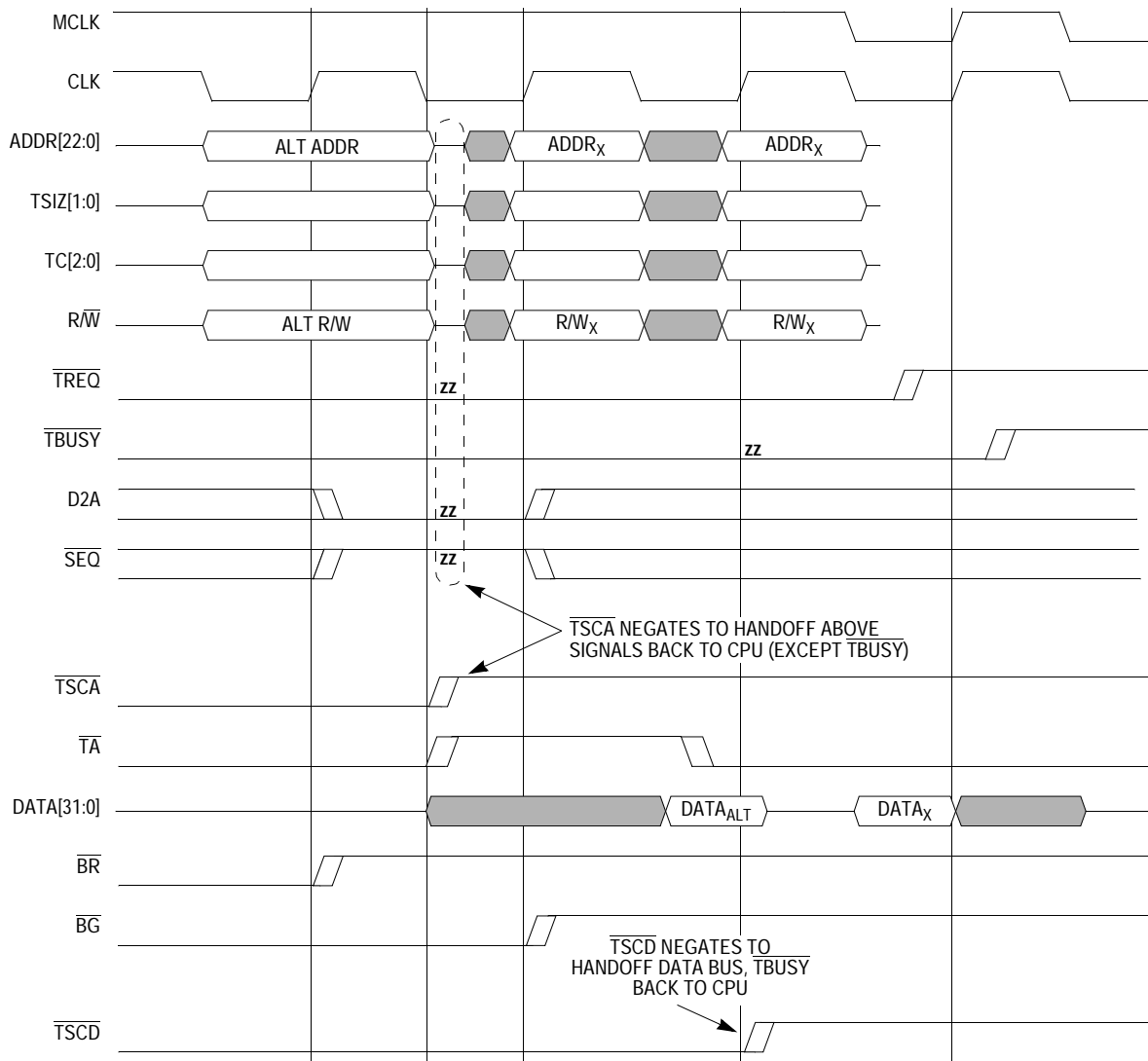
Figure D-16. Arbitration Operation, Back-to-Back Cycles

**Figure D-17** shows an example of release of the bus by an alternate master with no pending CPU request. In this case, the CPU negates the  $\overline{\text{TREQ}}$  and  $\overline{\text{TBUSY}}$  signals after  $\overline{\text{TSCA}}$  negates.



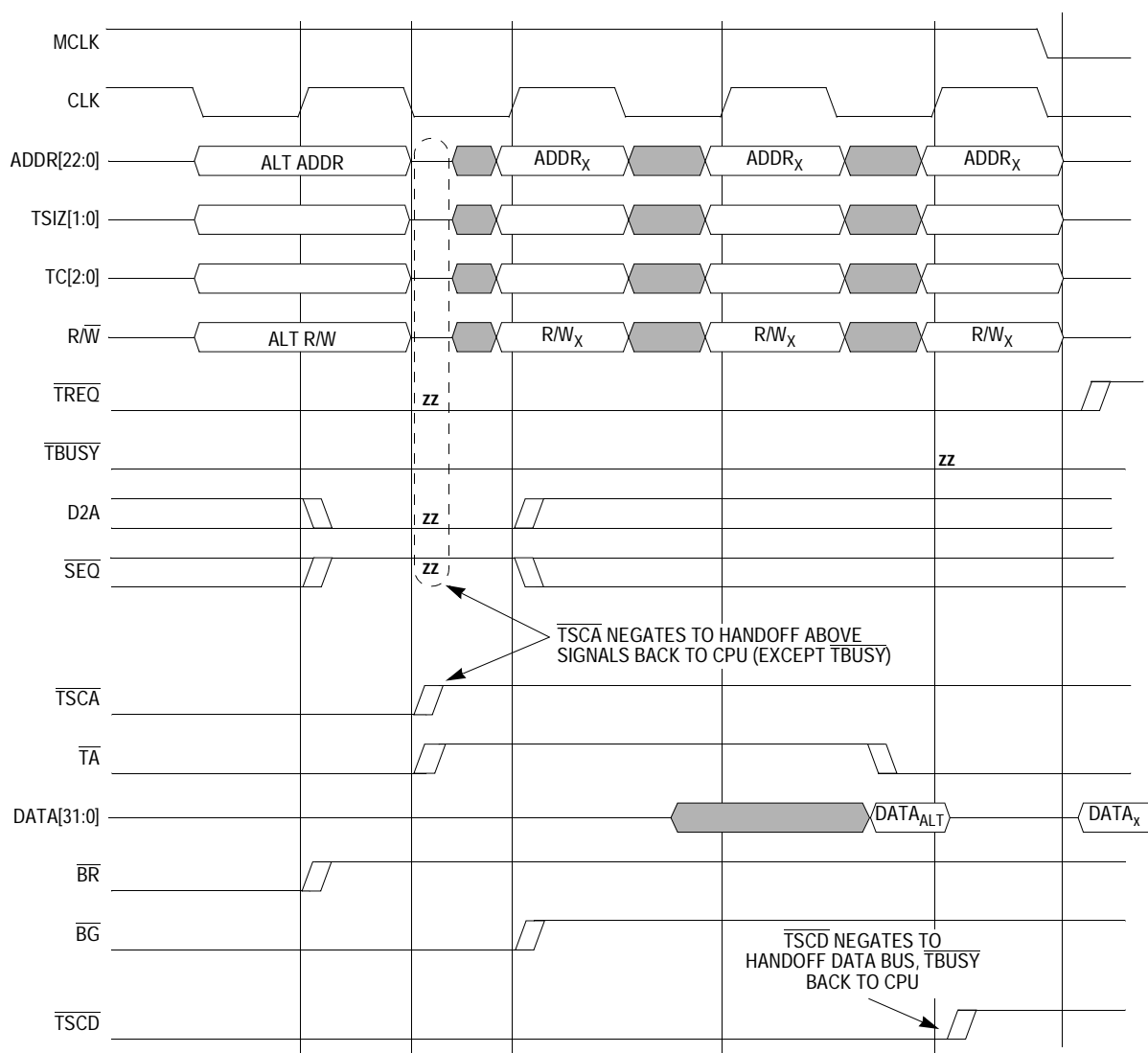
**Figure D-17. Arbitration Operation, Bus Request → Bus Grant Negation, No Pending CPU Request**

**Figure D-18** and **Figure D-19** show examples of one or more wait states following negation of  $\overline{BR}$ . In this case,  $\overline{TSCD}$  is held asserted until completion of the alternate cycle in progress even though  $\overline{BG}$  has already negated.



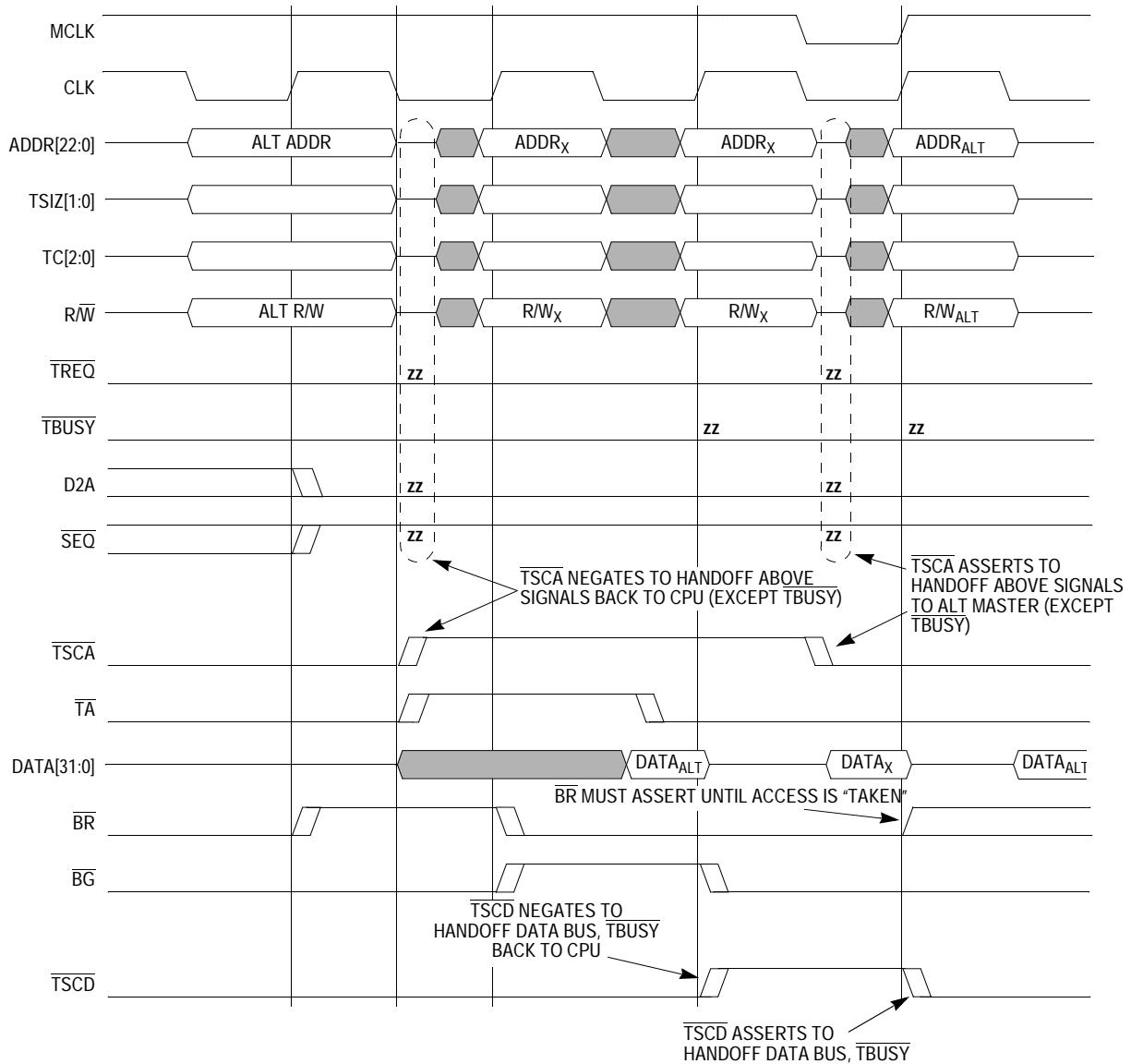
**Figure D-18. Arbitration Operation, Bus Request → Bus Grant Negation, One Wait State on Alternate Master Cycle**

## M210/M210S Interface Operation



**Figure D-19. Arbitration Operation, Bus Request → Bus Grant Negation, Multiple Wait States on Alternate Master Cycle**

**Figure D-20** and **Figure D-21** show examples of bus re-request with an outstanding alternate master cycle. In this case, a pending CPU request is initiated prior to reassertion of  $\overline{BG}$ .



**Figure D-20. Bus Re-request with Wait State on Alternate Master Cycle**

## M210/M210S Interface Operation

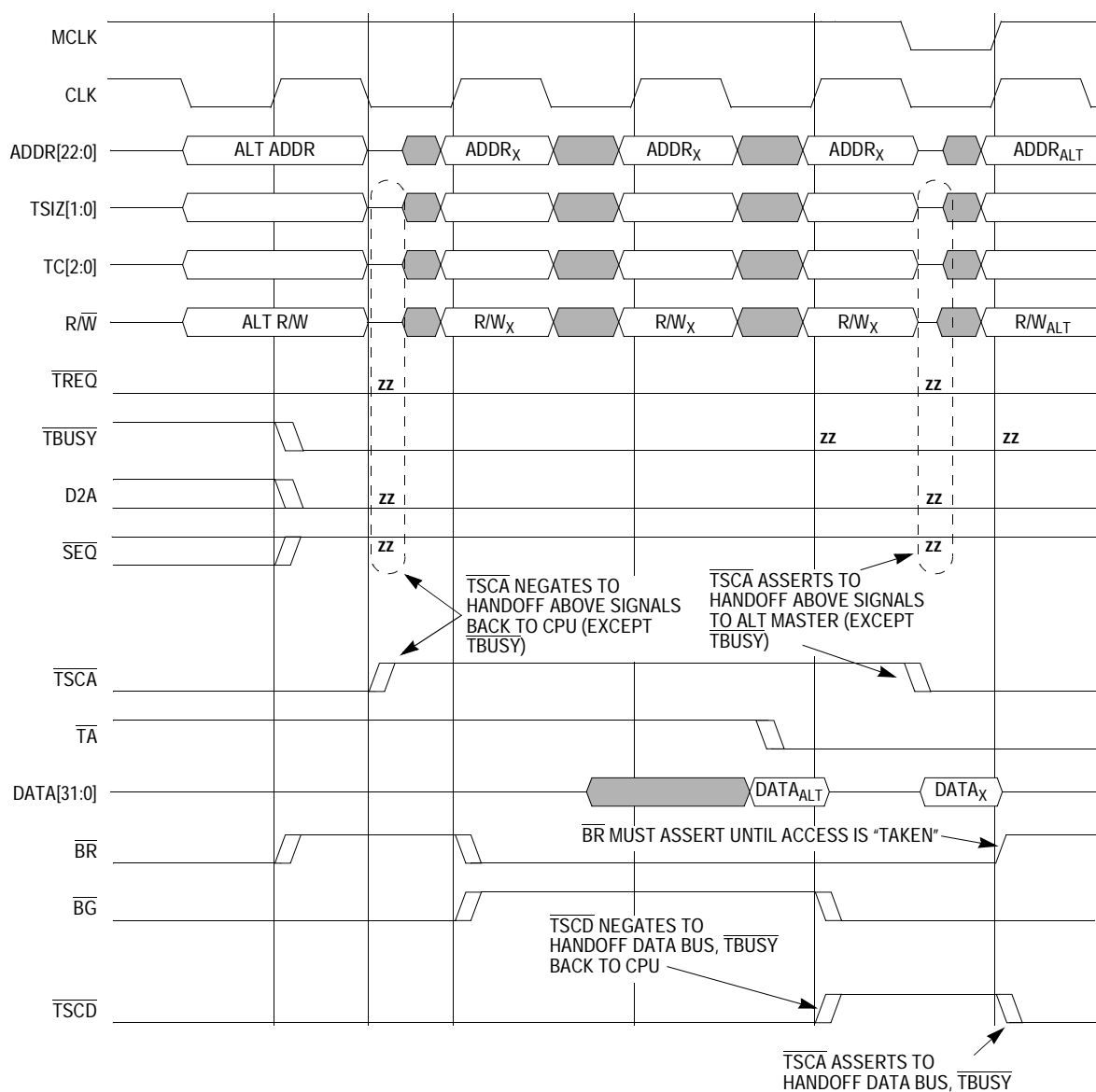
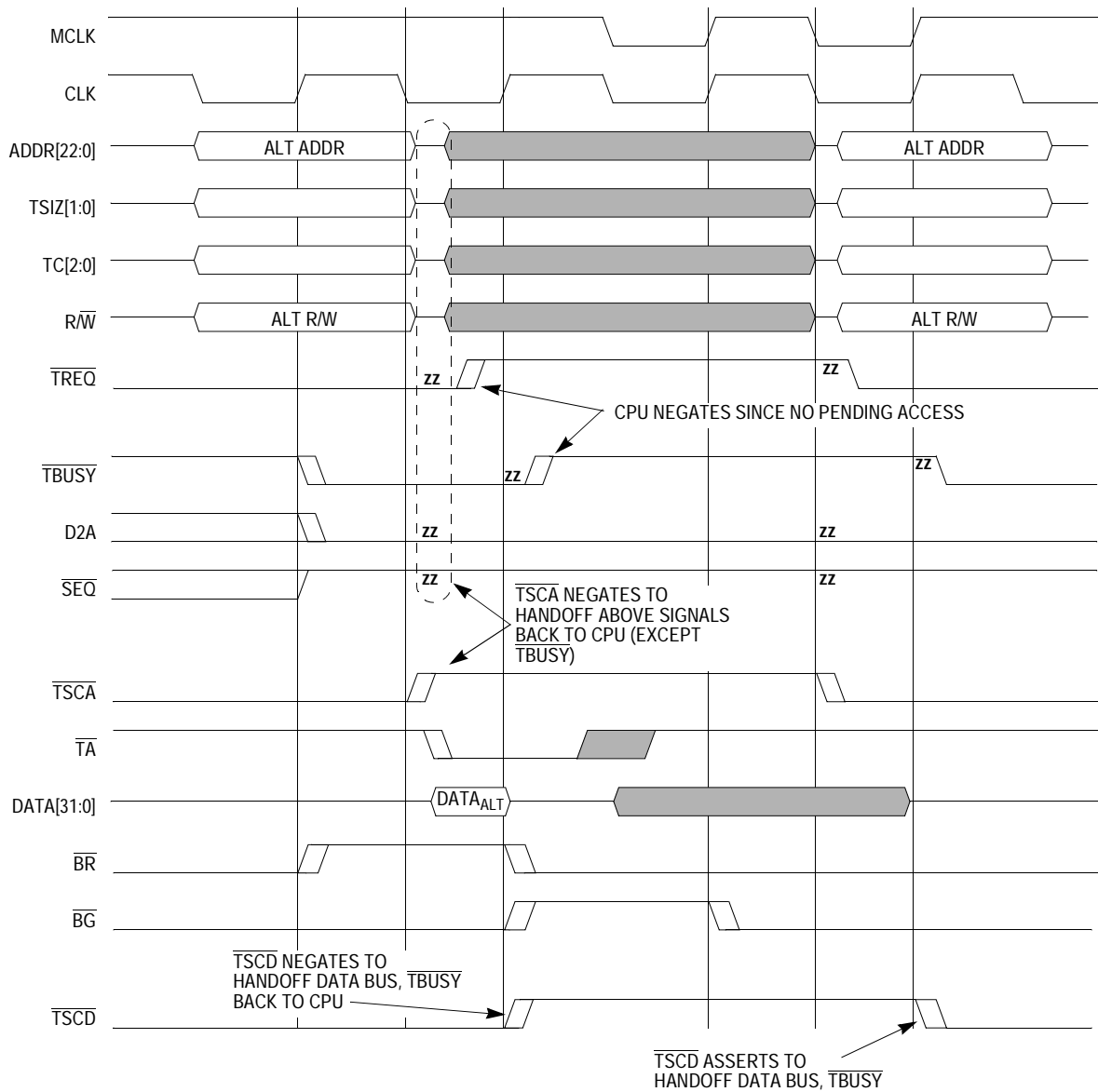


Figure D-21. Bus Re-request with Multiple Wait States on Alternate Master Cycle

**Figure D-22** shows an example of bus re-request with no pending CPU access.



**Figure D-22. Arbitration Operation, Bus Request → Bus Grant Negation, No Pending CPU Request, Bus Re-Request**

### D.12.2 Interaction with Low-Power Modes and Debug Operation

The CPU is capable of signalling the DOZE, STOP, and WAIT low-power modes via  $\overline{\text{LPMD}}[1:0]$  once a low-power state has been entered. Refer to [D.16 Power Management Interface Operation](#) for more information. While in these modes, the clocks to the CPU core may be disabled at the system level. The bus arbitration logic needs to have the clocks re-enabled to allow the bus to be granted, and this is handled in a similar manner to receiving an interrupt while in a low-power state. Upon detecting assertion of  $\overline{\text{BR}}$ , the CPU generates a wakeup signal ( $\overline{\text{WAKEUP}}$ ) to the system level clock module to cause the CPU clocks to be re-enabled for the duration of the alternate master's bus ownership. Following the release of the bus by the alternate master,  $\overline{\text{WAKEUP}}$  is negated (assuming no other wakeup event is pending), allowing the CPU clocks to be disabled again if desired. In this manner, bus arbitration functions normally even with the CPU in a low-power state. Arbitration latency is a function of the system clock controller's response time to  $\overline{\text{WAKEUP}}$ . The M210 remains in a low-power state until a pending interrupt or debug request is detected.

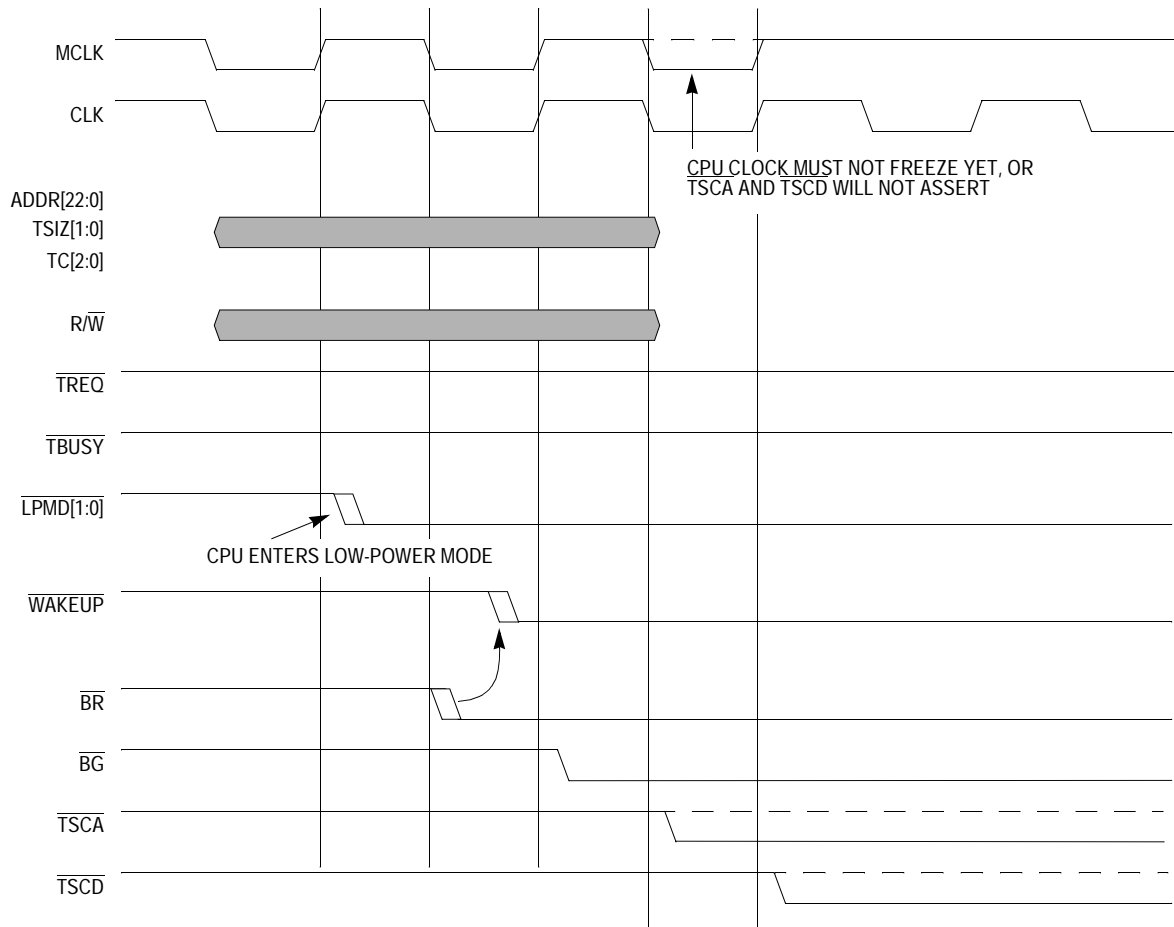
In debug mode, the CPU continues to respond to bus requests and grants the bus with minimal to no additional latency over normal operation. In certain circumstances, such as a pending access or accesses which must be completed, entry into debug mode may be delayed due to arbitration by the alternate master. This has no effect on the alternate master however.

### D.12.3 Bus Arbitration and Entry into Low-Power States

A problem may occur if a bus request is received in the clock cycle just prior to the clock cycle where the system clock controller would normally freeze the CPU clock in an inactive (high) state. In this case, the bus may be granted by assertion of  $\overline{\text{BG}}$ , but since the CPU clock is held high, no  $\overline{\text{TSCA}}$  or  $\overline{\text{TSCD}}$  assertion occurs. Although the  $\overline{\text{WAKEUP}}$  output signal will be asserted, if the system clock control uses a synchronized version (for example, delayed), it will not be seen until after the alternate master has potentially been clocked. Either the  $\overline{\text{BR}}$  input signal or the  $\overline{\text{WAKEUP}}$  output signal (if it is synchronous to the system clock and not delayed)



must be monitored by the system clock controller in a synchronous fashion to ensure the clocks remain running or the alternate master clock must be disabled as well. **Figure D-23** shows the window where this might occur.



**Figure D-23. Arbitration Operation, Entry into Low-Power Mode**

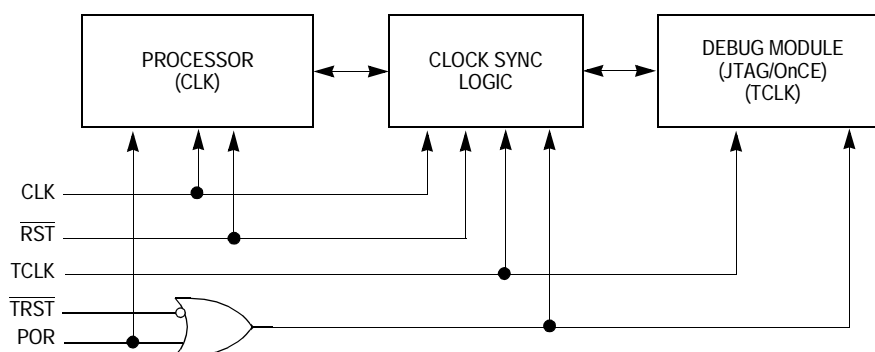
## D.13 Reset Operation

This subsection describes the functionality and requirements for M210 reset related signals. The three reset inputs on the M210 are summarized in [Table D-3](#).

**Table D-3. M210 Reset and Clock Domains**

Signal	Description	Clock Domain	Assertion Time Requirement
$\overline{\text{RST}}$	Processor (system) reset	CLK	10 $\mu\text{s}$ (CDR1)
$\overline{\text{TRST}}$	JTAG reset	TCLK	10 ns
POR	Power-on reset	CLK and TCLK	10 ns

[Figure D-24](#) depicts the two functional reset and clock domains of M210. The clock synchronization logic between the two clock domains is also shown.



**Figure D-24. M210 Clocks and Reset Domains**

The reset and clock domains have been partitioned such that the  $\overline{\text{RST}}$  signal does not affect JTAG/OnCE logic and  $\overline{\text{TRST}}$  does not affect processor logic. It is possible and desirable to access OnCE registers while the processor is running or in reset. Alternatively, it is also possible and desirable to assert  $\overline{\text{TRST}}$  and clear the JTAG/OnCE logic without affecting the processor.

**NOTE:** Since the clock synchronization logic and the JTAG-based OnCE state machine must be cleared at power-up, either  $\overline{\text{TRST}}$  or POR must be

*asserted during processor power-up reset ( $\overline{RST}$ ) for proper operation. After power-up, the two clock domains may be reset independently.*

**Table D-4** describes the functions of the reset related signals.

**Table D-4. Reset Signals**

Signal	Description
$\overline{RST}$	The $\overline{RST}$ input is the M210 processor's active low reset. $\overline{RST}$ is considered an asynchronous input and is sampled by the clock control logic in the M210 debug module in order to exit from reset gracefully. The hold time specified in <a href="#">Table D-3</a> reflects the time required to flush all M210 scan chains to zero.
$\overline{TRST}$	The $\overline{TRST}$ signal is the JTAG reset, commonly referred to in the IEEE1149.1 specification. This is an asynchronous clear with a very short assertion time requirement. It is ORed with POR and the resulting signal clears the JTAG tap controller and associated registers as well as the OnCE state machine. $\overline{TRST}$ (along with the other JTAG signals) is part of a defined JTAG/OnCE port interface connector which has been developed for Motorola and third party debug controllers.
POR	The POR input is an asynchronous clear with a very short assertion time requirement. It is ORed with $\overline{TRST}$ and the resulting signal clears the JTAG tap controller and associated registers as well as the OnCE state machine. POR also initializes the clock logic.

### D.13.1 System Issues

JTAG compliance requires that the pin associated with  $\overline{TRST}$  include a pullup resistor. Since it is desirable to connect a device containing an M210 processor to a debug controller via the JTAG/OnCE port interface connector, most designs leave  $\overline{TRST}$  floating. The pullup resistor causes a logic 1 on the pin and  $\overline{TRST}$  is negated.

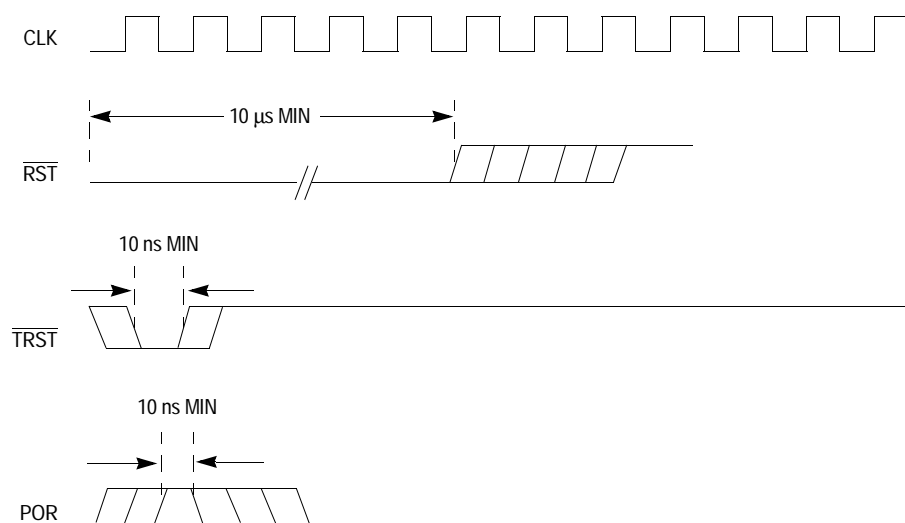
As mentioned earlier, the synchronization logic between the processor and debug module requires an assertion of either  $\overline{TRST}$  or POR during processor power-up reset ( $\overline{RST}$ ) in order to ensure proper operation. If the pin associated with the  $\overline{TRST}$  input is designed with a pullup resistor and left floating, then assertion of POR is required during processor reset. Similarly, for those systems which do not have a power-on reset circuit and choose to tie POR low, it is required to assert  $\overline{TRST}$  during

processor power-up reset. Again, once a power-up reset has been achieved, the two clock domain resets can be asserted independently.

It is strongly suggested that customers implement a power-on reset circuit to drive POR. In those cases where  $\overline{RST}$  may be unknown during the initial power-on sequence, POR is used to prevent internal bus contention and therefore may increase the long term reliability of the part.

### D.13.2 Timing

The timing requirements for the resets are shown in [Figure D-25](#).



**Figure D-25. Reset Timing Requirements**

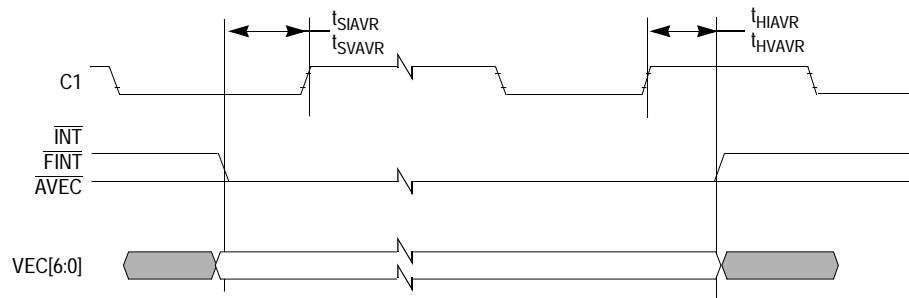
## D.14 Interrupt Interface Operation

The M210 core provides a flexible interrupt interface to an external interrupt control module.

The  $\overline{\text{FINT}}$  and  $\overline{\text{INT}}$  inputs are used to request a particular type of interrupt, and the  $\overline{\text{AVEC}}$  and  $\text{VEC}[6:0]$  inputs are used to control the interrupt vectoring process. When  $\overline{\text{FINT}}$  or  $\overline{\text{INT}}$  is asserted, either the  $\overline{\text{AVEC}}$  or  $\text{VEC}[6:0]$  inputs must be driven to a valid value as well, in order to properly generate the interrupt exception vector. If  $\overline{\text{INT}}$  has been asserted and  $\overline{\text{FINT}}$  becomes asserted, the vector number provided must track the interrupt which will be recognized, otherwise the wrong vector number may be used. On each rising clock edge, the vector number and/or  $\overline{\text{AVEC}}$  must be driven appropriately for the particular interrupt request lines which are also asserted.

Interrupt inputs to the M210 core are all level sensitive, not edge-triggered, thus the interrupt request as well as the  $\text{VEC}[6:0]$  or  $\overline{\text{AVEC}}$  inputs must remain asserted until the interrupt is serviced to guarantee that the M210 core recognizes the request. On the other hand, once a request is generated, there is no guarantee the M210 core will not recognize the interrupt request even if the request is later removed.

**Figure D-26** shows the functional timing of these signals. All of these signals must meet setup time requirements to the rising edge of the clock for proper CPU operation. Once asserted, the CPU recognizes and begins to process the interrupt exception on the next instruction decode boundary where the interrupt is the highest priority exception.

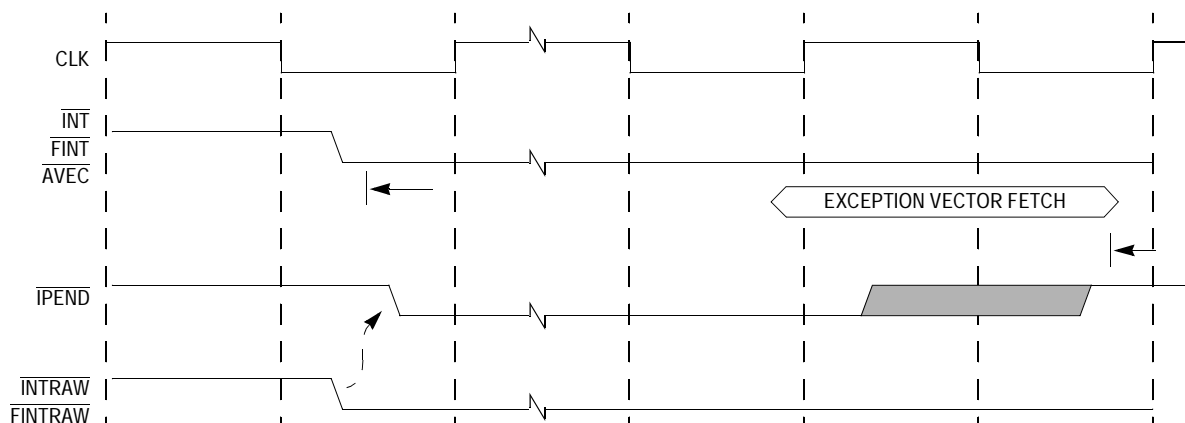


**Figure D-26. Interrupt Interface Signals**

The  $\overline{IPEND}$  output can be used to control power management operation as well as assist in bus arbitration priority schemes. The  $\overline{IPEND}$  output is a function of  $\overline{INTRAW}$  and  $\overline{FINTRAW}$ , as well as interrupt enable bits in the PSR. These interrupt input signals have no setup and hold requirements with respect to the rising edge of the clock, as  $\overline{IPEND}$  is generated from a combination of these inputs, thus is not referenced to a clock edge. This allows it to be used as a wakeup signal to an external power management/clock generation module when CLK has been disabled in the high state.  $\overline{IPEND}$  remains asserted until the processor begins exception processing and updates the PSR to mask further interrupts. At this point, the  $\overline{IPEND}$  setup to the rising edge of the CLK is negated unless a normal interrupt is pending and acknowledged and a fast interrupt becomes pending prior to the normal negation of  $\overline{IPEND}$ . Refer to [Figure D-27](#).

The  $\overline{IPEND}$  output is not guaranteed to be negated if another interrupt which is not masked by recognition of the first ( $\overline{FINT}$  following  $\overline{INT}$ ) is presented to the core before the first instruction of the handler for the original interrupt is fetched and decoded, and exception processing begins again for the higher priority interrupt.

**NOTE:**  $\overline{INTRAW}$  and  $\overline{FINTRAW}$  do not directly request interrupts, they only participate in  $\overline{IPEND}$  and  $\overline{WAKEUP}$  signal generation.  $\overline{INT}$  and  $\overline{FINT}$  are used to actually control interrupt recognition.



**Figure D-27. Interrupt Signals**

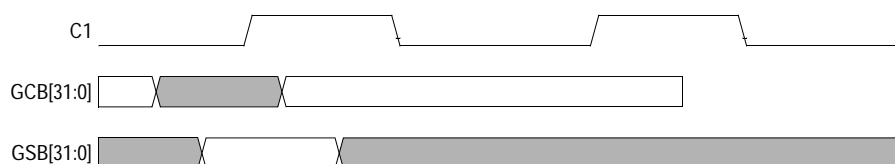
## D.15 Global Status and Control Interface Operation

The M210 core provides two control registers as part of the supervisor programming model to monitor global status in the integrated system as well as to provide global control outputs to the system. Refer to [Section 2. Registers](#) for register information.

The GCB[31:0] outputs change state when the GCR register is updated by the MTCR instruction.

The GSB[31:0] inputs are sampled by the core and the corresponding values appear in the GSR for transfer to a general register when an MFCR instruction referencing the GSR is executed.

**Figure D-28** shows the functional timing of these signals. The GSB inputs are sampled with the falling edge of the C1, and the GCB outputs transition following the rising edge of the C1.



**Figure D-28. Global Status and Control Signals**

## D.16 Power Management Interface Operation

The M210 core provides three instructions, DOZE, STOP, and WAIT, to implement low-power operating modes. The functionality of these modes is not dictated by the M210 core, but is determined by the design of external power management circuitry. The M210 core provides output signals associated with the execution of each of these instructions that can be monitored by external logic to control M210 core and system operation.

When a DOZE, STOP, and WAIT instruction is executed, the appropriate mode is indicated on the  $\overline{\text{LPMD}}[1:0]$  outputs. External logic can decode the signals, then place the M210 core in a low-power state

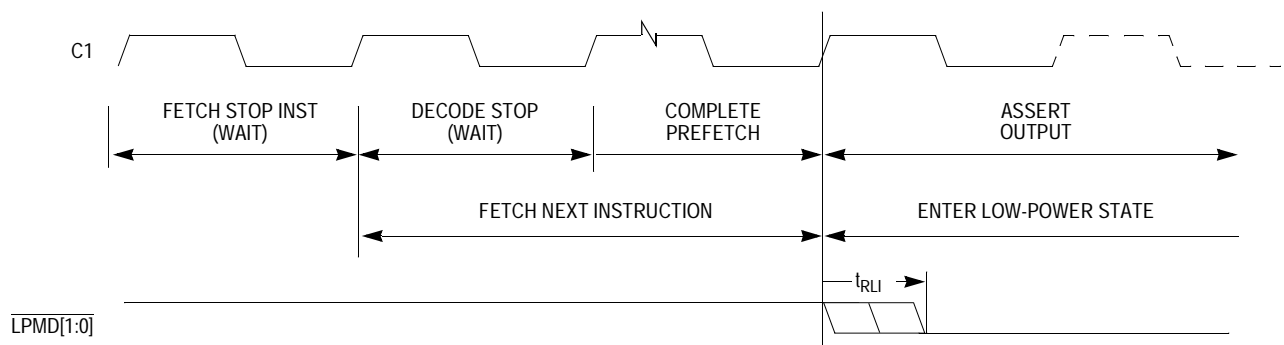
## M210/M210S Interface Operation

by forcing CLK high. The M210 core can be re-enabled by providing CLK as system events dictate.

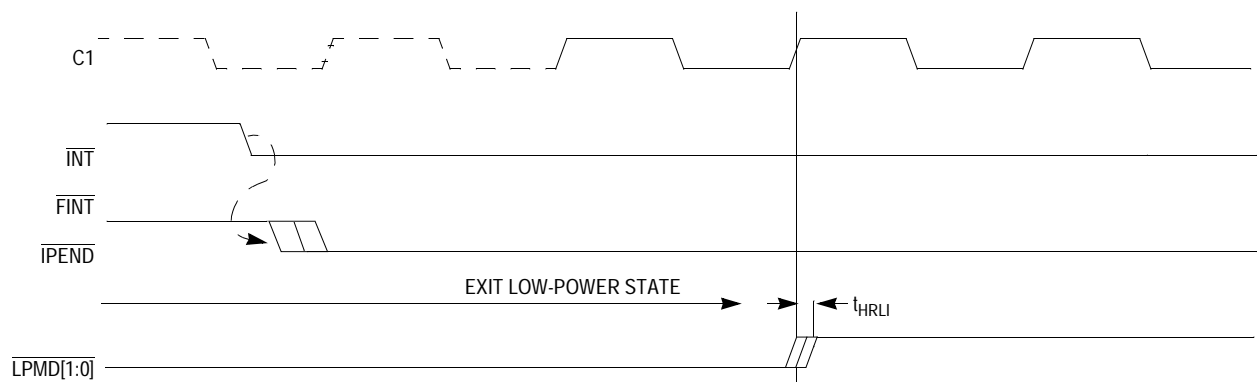
Terminating a low-power mode requires recognition of a valid interrupt service request, and the assertion of  $\overline{IPEND}$  or the assertion of a debug request. The M210 core remains in a stopped or waiting state until a valid interrupt is pending and CLK has been re-enabled.

Execution of the DOZE, STOP, and WAIT instruction is delayed until any outstanding prefetch has completed.

**Figure D-29** and **Figure D-30** show the functional timing of these signals. The  $\overline{LPMD}[1:0]$  outputs transition following the rising edge of CLK after all outstanding fetches have been completed.



**Figure D-29. Power Management Signals Assertion**



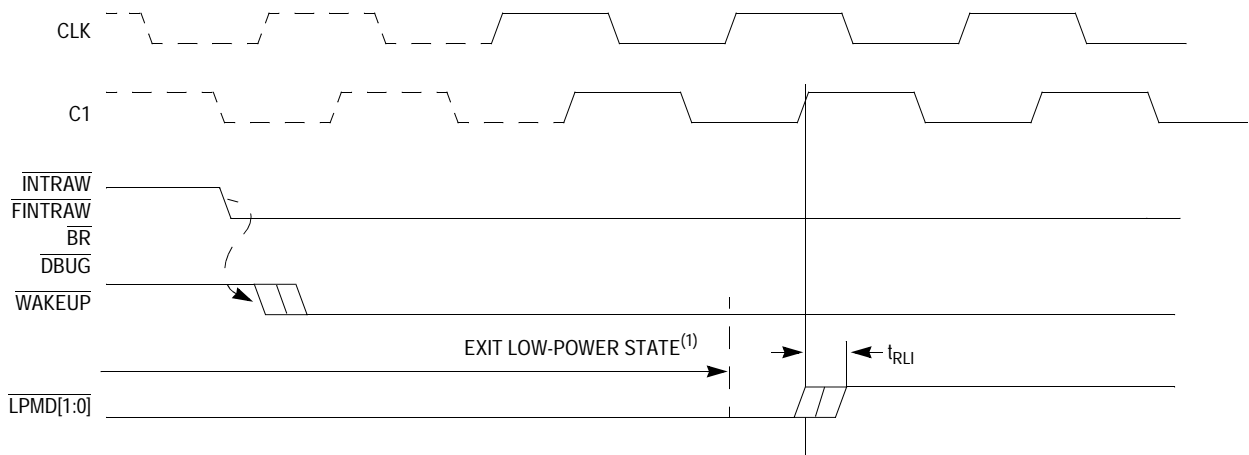
**Figure D-30. Power Management Signals Negation**



Refer to [D.14 Interrupt Interface Operation](#) for more information on interrupt recognition while in a stopped or waiting state.

**Figure D-31** shows the  $\overline{\text{WAKEUP}}$  signal. The  $\overline{\text{WAKEUP}}$  output is asserted to cause the system level clock generation logic to reapply the CPU clock in the case of the assertion of the  $\overline{\text{INTRAW}}$ ,  $\overline{\text{FINTRAW}}$ ,  $\overline{\text{BR}}$ , or  $\overline{\text{DEBUG}}$  inputs. In the case of  $\overline{\text{INTRAW}}$ ,  $\overline{\text{FINTRAW}}$ , or  $\overline{\text{DEBUG}}$ , the processor exits low-power modes, and  $\overline{\text{LPMD}}[1:0]$  are negated. In the case where  $\overline{\text{BR}}$  only is asserted, the  $\overline{\text{WAKEUP}}$  is asserted, but the CPU does not exit a low-power state. Arbitration occurs once CLK has been re-enabled by the system clock controller. Once the alternate master has relinquished the bus and the arbitration sequence has completed,  $\overline{\text{WAKEUP}}$  is negated (unless another cause to wakeup exists) and the CLK may be disabled.

$\overline{\text{WAKEUP}}$  is generated asynchronously from these inputs, so it should be synchronized if required. However, certain problems with clock freezing and bus arbitration may occur. Refer to [D.12.3 Bus Arbitration and Entry into Low-Power States](#) for more information.



Note 1. Exit low-power state does not occur for  $\overline{\text{BR}}$ -only assertion.

**Figure D-31. Wakeup Control Signal ( $\overline{\text{WAKEUP}}$ )**

### D.17 Emulation/Debug Interface Operation

When asserted, the  $\overline{\text{DBUG}}$  input sends a debug request to the M210 processor and subsequently forces the processor to enter debug mode. This signal is part of a defined JTAG/OnCE port interface connector which has been developed for Motorola and Third Party debug controllers. For more information refer to [Section 8. JTAG Test Access Port and OnCE](#).

## Index

### A

$\overline{\text{ABORT}}$ .....	185, 200, 209
Aborted bus cycles .....	209
ABS instruction .....	68
Absolute value .....	68
Accelerator block .....	220, 221
Access error exception .....	170
ADDC instruction .....	69
ADDI instruction .....	70
Address	
bus .....	182
Addressing	
control register .....	59
dyadic register .....	56
indirect mode .....	61
modes .....	54
monadic register .....	55
register with 4-bit negative displacement .....	62
register with 5-bit immediate .....	57
register with 5-bit offset immediate .....	58
register with 7-bit immediate .....	58
scaled 11-bit displacement mode .....	61
scaled 4-bit immediate .....	59
ADDU instruction .....	71
AF bit .....	32, 43, 47
Alternate file .....	30, 31, 43
bit .....	32, 43, 47

AND instruction .....	72
ANDI instruction.....	73
ANDN instruction.....	74
Arithmetic shift right	
by 1 bit .....	76
dynamic .....	75
immediate (static).....	77
ASR instruction .....	75
ASRC instruction.....	76
ASRI instruction.....	77
Autovector signal.....	190
$\overline{\text{AVEC}}$ .....	175, 190, 213

## B

BCLRI instruction.....	78
BF instruction.....	79
BGENI instruction .....	80
BGENR instruction .....	81
Big-endian byte ordering .....	33
Bit	
clear immediate .....	78
generate	
immediate (static) .....	80
register (dynamic).....	81
mask generate immediate .....	83
reverse .....	85
set immediate.....	86
test immediate .....	89
$\overline{\text{BKPT}}$ instruction .....	82, 173
BMASKI instruction .....	83
BR instruction .....	84

Branch	
if true	88
on true, decrementing count	117
to subroutine	87
unconditional	84
Breakpoint	
bus cycles	208
exception	173
instruction	82
signal	188
BREV instruction	85
$\overline{\text{BRKRQ}}$	173, 188
BSETI instruction	86
BSR instruction	87
BT instruction	88
BTSTI instruction	89
Bus	
aborted cycles	209
address	182, 196
alignment	196, 197, 199, 200, 201, 202, 203, 204, 206, 207, 208, 209, 210, 211
breakpoint cycles	208
changes of flow	210
characteristics	196
control signals	196, 197, 199, 200, 201, 202, 203, 204, 206, 207, 208, 209, 210, 211
data	182, 185, 196, 206
data multiplexer	197, 199, 200, 201, 202, 203, 204, 206, 207, 208
data transfer	196
error	170, 207, 208
exception cycles	207, 208
reset cycles	211
transfers	185, 197, 199, 200, 201, 202, 203, 204, 206, 207, 208, 209, 210, 211
wait states	202, 204

## C

C bit .....	30, 41, 48
Clear	
if condition false .....	90
if condition true .....	91
CLK .....	192
Clock signal .....	191, 192
CLRf instruction .....	90
CLRT instruction .....	91
CMPHS instruction .....	92
CMPLT instruction .....	93
CMPLTI instruction .....	94
CMPNE instruction .....	95
CMPNEI instruction .....	96
Compare	
for higher or same .....	92
for less than .....	93
for not equal .....	95
with immediate	
for less than .....	94
for not equal .....	96
Condition code/carry bit .....	30, 41, 48
Conditional branch	
if false .....	79
if true .....	88
Connections	
power .....	193
Control	
interface signals .....	192
register addressing mode .....	59
registers .....	31
addressing .....	59
Cycle	
bus transfer .....	185

**D**

D2A .....	187, 210
Data	
bus .....	182, 185
bus hand-off .....	206
memory access instructions .....	59
multiplexer .....	196
organization	
in memory .....	33
in registers .....	34
read cycles .....	200, 201, 202
to address signal .....	187
transfer mechanism .....	196
Data organization	
in memory .....	331
$\overline{\text{DBGRQ}}$ .....	193, 217
$\overline{\text{DEBUG}}$ .....	193, 217
Debug	
acknowledge signal .....	193
interface .....	217
request signal .....	193
signals .....	193
DECF instruction .....	97
DECGT instruction .....	98
DECLT instruction .....	99
DECNE instruction .....	100
Decrement	
conditionally on false .....	97
conditionally on true .....	101
set C bit on greater than .....	98
set C bit on less than .....	99
set C bit on not equal .....	100
DECT instruction .....	101
Divide-by-zero exception .....	170
DIVS instruction .....	102

DIVU instruction . . . . .	103
DOZE instruction . . . . .	104
Dyadic register addressing . . . . .	56
Dynamic bus sizing . . . . .	196

## E

EE bit . . . . .	46
Emulation interface . . . . .	217
Emulation support . . . . .	193
EPC . . . . .	50, 165
EPSR . . . . .	31, 50, 165
Exception	
control cycles . . . . .	207, 208
processing . . . . .	164
Exceptions	
access error . . . . .	170
breakpoint . . . . .	173
bus error . . . . .	170
divide-by-zero . . . . .	170
enable bit . . . . .	46
hardware accelerator . . . . .	176
illegal instruction . . . . .	170
interrupt . . . . .	175
misaligned access . . . . .	169
priorities . . . . .	176
privilege violation . . . . .	171
processing . . . . .	31, 165
reset . . . . .	169, 173
returning from . . . . .	178
shadow registers . . . . .	33, 50
trace . . . . .	171
trap . . . . .	176
types . . . . .	168
unrecoverable error . . . . .	173
vectors . . . . .	167
Execution status . . . . .	191



Extended shift right .....	156
External accelerator block .....	220, 221
Extract	
byte 1 into R1 and zero-extend .....	158
byte 2 into R1 and zero-extend .....	159
high-order byte into R1 and zero-extend .....	157
low-order byte into R1 and zero-extend .....	160

## F

Fast interrupt request signal .....	189
Fast interrupts .....	175
enable bit .....	47
FE bit .....	47
FF1 instruction .....	105
Find first one in RX .....	105
FINT .....	175, 189, 196, 212
Flow control instructions .....	61
FPC .....	50, 165
FPSR .....	31, 50, 165

## G

GCR .....	51
General-purpose registers .....	31, 41
Global	
control register .....	51
status register .....	51
GPRs .....	39, 41
GSR .....	51

## H

$\overline{H\_BUSY}$ .....	224
H_CALL .....	230, 238
$\overline{H\_DA}$ .....	230, 237

$\overline{H\_DEC}$ .....	223
$\overline{H\_DERR}$ .....	230
$\overline{H\_DS}$ .....	230
$\overline{H\_ERR}$ .....	237
$\overline{H\_EXCP}$ .....	228, 237
$\overline{H\_EXEC}$ .....	224, 237, 241
$H\_LD$ .....	239
$H\_OP$ .....	222
$H\_RET$ .....	230, 239
$H\_ST$ .....	240
$H\_SUP$ .....	222
$HAI$ .....	222
Hand-off	
data bus .....	206
Hard reset .....	211
Hardware accelerator .....	220
back-to-back execution .....	225
control bits .....	45
control handshake .....	222, 224, 225, 227, 229, 230, 233, 234, 237
data transfer .....	229, 230
exception .....	176
instruction primitives .....	221, 238, 239, 240, 241, 242, 243, 244, 245, 246
instruction transfer .....	222, 224, 225, 227
memory transfer .....	233, 234
register snooping .....	221, 222, 224, 227, 229, 230
register transfer .....	230
signals .....	192, 221, 222, 224, 225, 227, 229, 230, 233, 234, 237
transfer error .....	237
HDP .....	230

## I

IC bit .....	47
IE bit .....	47
Illegal instruction exception .....	170
INCF instruction .....	106
Increment RX conditionally	
on false .....	106
on true .....	107
INCT instruction .....	107
Index	
halfword .....	108
word .....	109
Indirect mode .....	61
Instruction	
primitives .....	238
H_CALL .....	238, 242
H_EXEC .....	241, 244
H_LD .....	239, 245
H_RET .....	239, 243
H_ST .....	240, 246
read cycles .....	200, 201, 202
Instructions .....	54
ABS .....	68
ADDC .....	69
ADDI .....	70
ADDU .....	71
AND .....	72
ANDI .....	73
ANDN .....	74
ASR .....	75
ASRC .....	76
ASRI .....	77
BCLRI .....	78
BF .....	79
BGENI .....	80
BGENR .....	81
BKPT .....	82, 173

BMASKI .....	83
BR .....	84
BREV .....	85
BRKRQ .....	173
BSETI .....	86
BSR .....	87
BT .....	88
BTSTI .....	89
CLRF .....	90
CLRT .....	91
CMPHS .....	92
CMPLT .....	93
CMPLTI .....	94
CMPNE .....	95
CMPNEI .....	96
DECF .....	97
DECGT .....	98
DECLT .....	99
DECNE .....	100
DECT .....	101
DIVS .....	102
DIVU .....	103
DOZE .....	104
FF .....	105
flow control .....	61
INCF .....	106
INCT .....	107
IXH .....	108
IXW .....	109
JMP .....	110
JMPI .....	111
JSR .....	112
JSRI .....	113
LD .....	114
LDM .....	115
LDQ .....	116
LOOP .....	117
LRW .....	118
LSL .....	119
LSLC .....	120
LSLI .....	121

LSR	122
LSRC	123
LSRI	124
MFCR	125
MOV	126
move from control register	32
move to control register	32
MOVF	127
MOVI	128
MOVT	129
MTCR	130
MULT	131
MVC	132
MVCV	133
NOT	134
OR	135
register-to-register	54
RFI	136
ROTLI	137
RSUB	138
RSUBI	139
RTE	140
SEXTB	141
SEXTH	142
ST	143
STM	144
STOP	145
STQ	146
SUBC	147
SUBI	148
SUBU	149
SYNC	150
timing	29
TRAP	151
trap exception	176
TST	152
TSTNBZ	153
WAIT	154
XOR	155
XSR	156
XTRB0	157

XTRB1 .....	158
XTRB2 .....	159
XTRB3 .....	160
ZEXTB .....	161
ZEXTH .....	162
INT .....	175, 189, 196, 212
Interrupt	
control signals .....	189, 212
pending status signals .....	189
request signal .....	189
vector number signal .....	189
Interrupts	
autovector .....	190
control bit .....	47
cycles .....	212
enable bit .....	47
fast .....	175, 189, 212
normal .....	175, 189, 212
status .....	189
vector number .....	189
IPEND .....	189, 213
IXH instruction .....	108
IXW instruction .....	109

## J

JMP instruction .....	110
JMPI instruction .....	111
JSR instruction .....	112
JSRI instruction .....	113

## L

LD instruction .....	114
LDM instruction .....	115
LDQ instruction .....	59, 116
Link register .....	32

Load	
multiple registers from memory	115
PC-relative word	118
register from memory	114
register quadrant from memory	116
relative word mode	60
store	
multiple register mode	60
register quadrant mode	59
Load and store	30
Logical	
AND	72
AND NOT	74
AND with immediate	73
exclusive OR	155
move	126
move immediate	128
NOT	134
OR	135
shift left	
by 1, update C bit	120
dynamic	119
immediate (static)	121
shift right	
by 1 bit, update C bit	123
dynamic	122
immediate (static)	124
LOOP instruction	117
Low-power	
doze mode	
enter	104
mode signals	190
stop mode	
enter	145
wait mode	
enter	154
LRW instruction	118

LSL instruction . . . . .	119
LSLC instruction . . . . .	120
LSLI instruction . . . . .	121
LSR instruction . . . . .	122
LSRC instruction . . . . .	123
LSRI instruction . . . . .	124

## M

Memory	
load and store . . . . .	30
organization . . . . .	33
Memory management	
signals . . . . .	212
MFCR instruction . . . . .	32, 125, 171
Misaligned access exception . . . . .	169
Misaligned transfers . . . . .	196
Misalignment exception mask . . . . .	46
MM bit . . . . .	46
Monadic register addressing . . . . .	55
MOV instruction . . . . .	126
Move	
C bit to register . . . . .	132
from control register . . . . .	32, 125
inverted C bit to register . . . . .	133
to control register . . . . .	32, 130
Move RY to RX if condition false . . . . .	127
Move RY to RX if condition true . . . . .	129
MOVF instruction . . . . .	127
MOVI instruction . . . . .	128
MOVT instruction . . . . .	129
MTCR instruction . . . . .	32, 49, 130
MULT instruction . . . . .	131
Multiply . . . . .	29, 131



MVC instruction .....	132
MVCV instruction .....	133

## N

Normal interrupts .....	175
NOT instruction .....	134

## O

Opcode map .....	63
OR instruction .....	135

## P

PC .....	30, 31, 41
Pipeline	
instruction execution .....	29
Power connections .....	193
Power management	
control signals .....	190
instructions .....	215
low-power mode .....	190
restart timing .....	215
signals .....	213, 215
Power-on reset .....	211
Primitives	
H_CALL .....	238, 242
H_EXEC .....	241, 244
H_LD .....	239, 245
H_RET .....	239, 243
H_ST .....	240, 246
Privilege mode .....	30
supervisor .....	30
user .....	30
Privilege violation exception .....	171
Processor clock .....	192

Processor status register .....	43
updating .....	48
Processor status signals .....	191
Program counter .....	30, 31, 41
Programming mode .....	30
supervisor .....	41
user .....	40
PSR .....	43
updating .....	48
PSTAT .....	191

## R

$R/\overline{W}$ .....	186, 203
R0 .....	32, 34, 43
R15 .....	32
Read cycles .....	200, 201, 202
Read/write signal .....	186
Register	
addressing mode .....	61
plus-four-bit scaled displacement .....	34
snooping .....	220, 221, 222, 224, 225, 230, 233, 234, 237
vector base .....	49
with 4-bit negative displacement mode .....	62
with 5-bit immediate addressing .....	57
with 5-bit offset immediate addressing .....	58
with 7-bit immediate addressing .....	58
Registers	
control .....	31
EPC .....	165
EPSR .....	165
FPC .....	165
FPSR .....	165
general-purpose .....	31, 41
global control .....	51
global status .....	51
link .....	32

processor status	43, 48
scratch	31, 33
shadow	33, 50, 165
status	31
supervisor storage	50
vector base	49
$\overline{\text{REGWR}}$	221
Reset	
exception	169
hard	211
soft	173, 188, 211
Return	
from exception	31, 140
from fast interrupt	31, 136
Reverse	
subtract	138
with immediate	139
RFI instruction	31, 48, 136
RISC	28
Rotate	
left immediate (static)	137
ROTLI instruction	137
RSUB instruction	138
RSUBI instruction	139
RTE instruction	31, 48, 140

## S

S bit	31, 44
SC bit	46
Scaled	
11-bit displacement mode	61
4-bit immediate addressing	59
Scratch registers	31, 33
$\overline{\text{SEQ}}$	187

Sequential access signal .....	187
SEXTB instruction .....	141
SEXTH instruction .....	142
Shadow registers .....	31, 33, 50, 165
Sign extend	
byte .....	141
halfword .....	142
Signals	
autovector .....	190
breakpoint .....	188
bus control .....	197, 200, 201, 202, 203, 204, 206, 207, 208, 209, 210, 211
clock .....	191
control interface .....	192
data to address .....	187
debug .....	193
debug acknowledge .....	193
debug interface .....	217
debug request .....	193
electrical state .....	193
fast interrupt request .....	189
hardware accelerator .....	192, 220, 221, 222, 224, 225, 227, 229, 230, 233, 234, 237
interrupt control .....	189
interrupt pending status .....	189
interrupt request .....	189
interrupt vector number .....	189
low-power mode .....	190
memory management .....	188, 212
power management .....	190, 215
processor clock .....	192
processor status .....	191
read/write .....	186
sequential access .....	187
soft reset .....	188
status .....	191, 192, 214
test .....	193
transfer abort .....	185
transfer acknowledge .....	187

transfer attribute .....	185
transfer busy .....	185
transfer code .....	186
transfer control .....	187
transfer error acknowledg .....	188
transfer request .....	185
transfer size .....	186
translate control .....	188
Signed divide RX by R1 .....	102
Soft	
reset .....	211
reset exception .....	173
reset signal .....	188
SP bit .....	44
Spare bits .....	44
Spare control .....	46
SRST .....	188, 196, 211
ST instruction .....	143
Stack pointer .....	32, 34, 43
supervisor .....	50
Status	
and control interface .....	214
registers .....	31
signals .....	191, 192
STM instruction .....	144
STOP instruction .....	145
Store	
multiple registers to memory .....	144
register quadrant to memory .....	146
register to memory .....	143
STQ instruction .....	146
SUBC instruction .....	147
SUBI instruction .....	148
Subroutine calls .....	32
SUBU instruction .....	149

Supervisor	
bit	44
mode	30
programming model	41, 192
stack pointer	50
storage registers	50
SYNC instruction	150
Synchronize CPU	150
Synchronous clock signal	192

## T

$\overline{TA}$	187
$\overline{TBUSY}$	185, 199
TC	186
TC bit	46
$\overline{TE}$	188
$\overline{TEA}$	188, 208
$\overline{TEA}$ signal	170
Test	
register for no byte equal to zero	153
with zero	152
Test signals	193
TM	43
TM field	45
TP bit	45
Trace	
exception	171
mode	45
pending	45
Transfer	
abort signal	185
acknowledge signal	187
attribute signals	185
busy signal	185
code signals	186

control signals .....	187
error acknowledge .....	170
error acknowledge signal .....	188
request signal .....	185
size signals .....	186
Translate control signal .....	188
Translation control .....	46
TRAP instruction .....	151
$\overline{TREQ}$ .....	185, 199, 204
TSIZ .....	186
TST instruction .....	152
TSTNBZ instruction .....	153

## U

U bits .....	45
Unconditional	
branch .....	84
jump .....	110
indirect .....	111
subroutine indirect .....	113
to subroutine .....	112
trap to OS .....	151
Unrecoverable error exception .....	173
Unsigned	
add .....	71
with C bit .....	69
with immediate .....	70
divide RX by R1 .....	103
subtract .....	149
with C bit .....	147
with immediate .....	148
User	
mode .....	30
programming model .....	40

**V**

VBR .....	49
$\overline{\text{VEC}}$ .....	189
VEC field .....	45
VEC# .....	175, 212
Vectors	
exception .....	167
base register .....	49
number .....	45

**W**

WAIT instruction .....	154
Wait states .....	202
Write cycles .....	203, 204

**X**

XOR instruction .....	155
XSR instruction .....	156
XTRB0 instruction .....	157
XTRB1 instruction .....	158
XTRB2 instruction .....	159
XTRB3 instruction .....	160

**Z**

Zero extend	
byte instruction .....	161
halfword instruction .....	162
ZEXTB instruction .....	161
ZEXTH instruction .....	162





## How to Reach Us:

### **USA/EUROPE/LOCATIONS NOT LISTED:**

Motorola Literature Distribution  
P.O. Box 5405  
Denver, Colorado 80217  
1-303-675-2140  
1-800-441-2447

### **TECHNICAL INFORMATION CENTER:**

1-800-521-6274

### **JAPAN:**

Motorola Japan Ltd.  
SPS, Technical Information Center  
3-20-1, Minami-Azabu, Minato-ku  
Tokyo 106-8573 Japan  
81-3-3440-3569

### **ASIA/PACIFIC:**

Motorola Semiconductors H.K. Ltd.  
Silicon Harbour Centre  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T., Hong Kong  
852-26668334

### **HOME PAGE:**

<http://www.motorola.com/semiconductors/>



**MCORERM/D**  
**REV 1**