


# ColdFire® CF4e Core User's Manual

---

V4ECFUM/D  
Rev. 0, 06/2001



ColdFire is a registered trademark and DigitalDNA is a trademark of Motorola, Inc.  
I<sup>2</sup>C is a registered trademark of Philips Semiconductors

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 and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

**How to reach us:**

**USA/EUROPE/Locations Not Listed:** Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1-303-675-2140 or 1-800-441-2447

**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

**Technical Information Center:** 1-800-521-6274

**HOME PAGE:** <http://www.motorola.com/semiconductors>

**Document Comments:** FAX (512) 895-2638, Attn: RISC Applications Engineering

**World Wide Web Addresses:** <http://www.motorola.com/PowerPC>  
<http://www.motorola.com/NetComm>  
<http://www.motorola.com/ColdFire>

Introduction	1
Registers	2
Instructions	3
FPU	4
EMAC	5
Execution Timing	6
Exceptions	7
Local Memory	8
MMU	10
Debug Module	11
Index	IND

1	Introduction
2	Registers
3	Instructions
4	FPU
5	EMAC
6	Execution Timing
7	Exceptions
8	Local Memory
10	MMU
11	Debug Module
IND	Index

# CONTENTS

Paragraph Number	Title	Page Number
---------------------	-------	----------------

## Chapter 1 Introduction

1.1	Core Overview .....	1-1
1.2	Features .....	1-1
1.3	CF4e Implementation Block Diagram .....	1-2
1.4	Architectural Summary .....	1-3
1.5	Programming Model .....	1-5
1.6	Address Map .....	1-7
1.7	Data Format Summary .....	1-9
1.7.1	Data Organization in Registers .....	1-9
1.7.1.1	Integer Data Format Organization in Registers .....	1-9
1.7.1.2	Integer Data Format Organization in Memory .....	1-10
1.7.2	EMAC Data Representation .....	1-11
1.7.2.1	Floating-Point Data Formats and Types .....	1-11
1.7.2.1.1	Signed-Integer Data Formats .....	1-12
1.7.2.1.2	Floating-Point Data Formats .....	1-12
1.8	Addressing Modes .....	1-12
1.9	Instruction Set Overview .....	1-13
1.9.1	Instruction Set Summary .....	1-16

## Chapter 2 Registers

2.1	Overview .....	2-1
2.2	User Programming Model .....	2-3
2.2.1	Data Registers (D7–D0) .....	2-4
2.2.2	Address Registers (A6–A0) .....	2-4
2.2.3	User Stack Pointer (A7) .....	2-5
2.2.4	Program Counter (PC) .....	2-5
2.2.5	Condition Code Register (CCR) .....	2-5
2.2.6	EMAC Programming Model .....	2-6
2.2.7	Floating-Point Programming Model .....	2-6
2.3	Supervisor Programming Model .....	2-7
2.3.1	Status Register (SR) .....	2-8

# CONTENTS

Paragraph Number	Title	Page Number
2.3.2	Vector Base Register (VBR).....	2-9
2.3.3	Supervisor/User Stack Pointers (A7 and OTHER_A7).....	2-9
2.3.4	Cache Control Register (CACR) .....	2-10
2.3.5	Access Control Registers (ACR0–ACR3).....	2-10
2.3.6	RAM Base Address Registers (RAMBAR0/RAMBAR1) .....	2-10
2.3.7	ROM Base Address Registers (ROMBAR0/ROMBAR1) .....	2-10
2.3.8	Module Base Address Register (MBAR) .....	2-10
2.4	Programming Model Table .....	2-12

## Chapter 3 Instructions

## Chapter 4 Floating-Point Unit (FPU)

4.1	FPU Overview .....	4-1
4.1.1	Notational Conventions .....	4-2
4.2	Operand Data Formats and Types.....	4-3
4.2.1	Signed-integer Data Formats .....	4-3
4.2.2	Floating-Point Data Formats.....	4-3
4.2.3	Floating-Point Data Types .....	4-4
4.2.3.1	Normalized Numbers.....	4-4
4.2.3.2	Zeros .....	4-4
4.2.3.3	Infinities .....	4-5
4.2.3.4	Not-A-Number.....	4-5
4.2.3.5	Denormalized Numbers .....	4-5
4.3	FPU Programmer's Model.....	4-7
4.3.1	Floating-Point Data Registers (FP0–FP7) .....	4-8
4.3.1.1	Floating-Point Control Register (FPCR) .....	4-8
4.3.2	Floating-Point Status Register (FPSR) .....	4-9
4.3.3	Floating-Point Instruction Address Register (FPIAR).....	4-11
4.3.4	Floating-Point Computational Accuracy .....	4-11
4.3.4.1	Intermediate Result .....	4-11
4.3.4.2	Rounding the Result .....	4-12
4.3.5	Floating-Point Post Processing .....	4-15
4.3.5.1	Underflow, Round, Overflow .....	4-16
4.3.5.2	Conditional Testing .....	4-16
4.3.6	Floating-Point Exceptions.....	4-19
4.3.7	Floating-Point Arithmetic Exceptions .....	4-20
4.3.8	Branch/Set on Unordered (BSUN) .....	4-21
4.3.9	Input Not-A-Number (INAN).....	4-22
4.3.10	Input Denormalized Number (IDE) .....	4-22

# CONTENTS

Paragraph Number	Title	Page Number
4.3.11	Operand Error (OPERR).....	4-23
4.3.12	Overflow (OVFL).....	4-23
4.3.13	Underflow (UNFL).....	4-24
4.3.14	Divide-by-Zero (DZ) .....	4-25
4.3.15	Inexact Result (INEX) .....	4-25
4.3.16	Floating-Point State Frames.....	4-26
4.4	Instructions.....	4-28
4.4.1	Floating-Point Instruction Overview .....	4-28
4.4.2	Floating-Point Instruction Execution Times.....	4-30
4.4.3	Key Differences between ColdFire and MC680x0 FPU Programming Models ..	4-31

## Chapter 5 Enhanced Multiply-Accumulate Unit (EMAC)

5.1	Multiply-Accumulate Unit.....	5-1
5.2	An Introduction to the MAC.....	5-2
5.3	General Operation .....	5-3
5.4	Memory Map/Register Set.....	5-6
5.4.1	MAC Status Register (MACSR).....	5-6
5.4.1.1	Fractional Operation Mode.....	5-9
5.4.1.1.1	Rounding .....	5-9
5.4.1.1.2	Saving and Restoring the EMAC Programming Model .....	5-10
5.4.1.1.3	MULS/MULU .....	5-11
5.4.1.1.4	Scale Factor in MAC or MSAC instructions.....	5-11
5.4.2	Mask Register (MASK).....	5-11
5.5	EMAC Instruction Set Summary .....	5-12
5.5.1	Data Representation.....	5-13
5.5.2	MAC Opcodes .....	5-13

## Chapter 6 Instruction Pipeline and Timing

6.1	Basic V4 Pipeline Strategy .....	6-1
6.2	Instruction Fetch Pipeline (IFP).....	6-4
6.3	Operand Execution Pipeline (OEP) .....	6-6
6.3.1	V4 OEP Conceptual Pipeline Model .....	6-6
6.3.2	Instruction Folding and the Limited Superscalar OEP .....	6-9
6.3.3	Sequence-Related OEP Stalls .....	6-11
6.3.4	EMAC-Specific OEP Sequence Stalls.....	6-13
6.3.5	FPU-Specific OEP Sequence Stalls.....	6-14
6.3.6	Operand Memory Sequence-Related Stalls .....	6-16

# CONTENTS

Paragraph Number	Title	Page Number
6.3.7	V4 OEP Summary .....	6-16
6.4	Instruction Execution Locations .....	6-18
6.5	Instruction Execution Times .....	6-21
6.5.1	MOVE Instruction Execution Times .....	6-23
6.5.2	Execution Timings—One-Operand Instructions .....	6-24
6.5.3	Execution Timings—Two-Operand Instructions.....	6-25
6.5.4	Miscellaneous Instruction Execution Times.....	6-27
6.5.5	Branch Instruction Execution Times .....	6-28
6.5.6	EMAC Instruction Execution Times .....	6-28
6.5.7	FPU Instruction Execution Times.....	6-30

## Chapter 7 Exception Processing

7.1	Overview.....	7-1
7.2	Supervisor/User Stack Pointers (A7 and OTHER_A7) 7-3	
7.3	Exception Stack Frame Definition.....	7-4
7.4	Processor Exceptions .....	7-5
7.5	Precise Faults .....	7-8

## Chapter 8 Local Memory

8.1	Local Memory Overview.....	8-1
8.2	Two-Stage Pipelined Local Bus (K-Bus) .....	8-5
8.3	Interactions between Local Memory Modules .....	8-7
8.4	Local Memory Connection Specification .....	8-8
8.4.1	K-Bus Memory Array Signal Connections.....	8-8
8.4.1.1	KRAM Information .....	8-8
8.4.1.2	KROM Controller Information.....	8-10
8.4.1.3	Instruction Cache Information.....	8-12
8.4.1.4	Data Cache Information.....	8-17
8.5	SRAM Overview .....	8-22
8.5.1	SRAM Operation .....	8-23
8.5.2	SRAM Programming Model.....	8-23
8.5.2.1	SRAM Base Address Registers (RAMBAR0/RAMBAR1).....	8-23
8.5.3	SRAM Initialization.....	8-25
8.5.4	SRAM Initialization Code .....	8-26
8.5.5	Programming RAMBARs for Power Management.....	8-27
8.6	ROM Overview.....	8-28
8.6.1	ROM Operation .....	8-28



# CONTENTS

Paragraph Number	Title	Page Number
8.6.2	ROM Programming Model .....	8-29
8.6.2.1	ROM Base Address Registers (ROMBAR0/ROMBAR1) .....	8-29
8.6.3	ROM Initialization .....	8-31
8.6.4	Programming ROMBARs for Power Management .....	8-32
8.7	Cache Overview .....	8-32
8.7.1	Optimizing Cache Recommendation .....	8-33
8.7.2	Cache Organization .....	8-33
8.7.2.1	Cache Line States: Invalid, Valid-Unmodified, and Valid-Modified .....	8-34
8.7.2.2	Cache at Start-Up .....	8-34
8.7.3	Cache Operation .....	8-35
8.7.4	Caching Modes .....	8-38
8.7.4.1	Cacheable Accesses .....	8-39
8.7.4.2	Write-Through Mode (Data Cache Only) .....	8-39
8.7.4.3	Copyback Mode (Data Cache Only) .....	8-39
8.7.5	Cache-Inhibited Accesses .....	8-39
8.7.6	Cache Protocol .....	8-40
8.7.6.1	Read Miss .....	8-41
8.7.6.2	Write Miss (Data Cache Only) .....	8-41
8.7.6.3	Read Hit .....	8-41
8.7.6.4	Write Hit (Data Cache Only) .....	8-42
8.7.7	Cache Coherency (Data Cache Only) .....	8-42
8.7.8	Memory Accesses for Cache Maintenance .....	8-42
8.7.8.1	Cache Filling .....	8-42
8.7.8.2	Cache Pushes .....	8-42
8.7.8.2.1	Push and Store Buffers .....	8-43
8.7.8.2.2	Push and Store Buffer Bus Operation .....	8-43
8.7.9	Cache Locking .....	8-44
8.7.10	Cache Registers .....	8-45
8.7.10.1	Cache Control Register (CACR) .....	8-46
8.7.10.2	Access Control Registers (ACR0–ACR3) .....	8-48
8.7.11	Cache Management .....	8-50
8.7.12	Cache Operation Summary .....	8-52
8.7.13	Instruction Cache State Transitions .....	8-52
8.7.13.1	Data Cache State Transitions .....	8-53

## Chapter 9 Core Interface

9.1	Core Interface Signals .....	9-1
9.2	CF4e Pin Characteristics .....	9-2
9.3	ColdFire Master Bus .....	9-6
9.3.1	M-Bus Signals .....	9-6

# CONTENTS

Paragraph Number	Title	Page Number
9.3.2	M-Bus Operation .....	9-8
9.3.2.1	Basic Bus Cycles .....	9-8
9.3.2.2	Pipelined Bus Cycles .....	9-9
9.3.2.3	Address and Data Phase Interactions.....	9-10
9.3.2.4	Data Size Operations .....	9-12
9.3.2.5	Line Transfers .....	9-13
9.3.2.6	Bus Arbitration .....	9-16
9.3.2.7	Interrupt Support.....	9-18
9.3.2.8	Reset Operation .....	9-18

## Chapter 10 Memory Management Unit (MMU)

10.1	Features .....	10-1
10.2	Virtual Memory Management Architecture.....	10-1
10.2.1	MMU Architecture Features .....	10-2
10.2.2	MMU Architectural Location .....	10-2
10.2.3	MMU Architecture Implementation .....	10-3
10.2.3.1	Precise Faults .....	10-4
10.2.3.2	MMU Access .....	10-4
10.2.3.3	Virtual Mode.....	10-4
10.2.3.4	Virtual Memory References .....	10-4
10.2.3.5	Instruction and Data Cache Addresses .....	10-4
10.2.3.6	Supervisor/User Stack Pointers .....	10-5
10.2.3.7	Access Error Stack Frame .....	10-5
10.2.3.8	Expanded Control Register Space .....	10-6
10.2.3.9	Changes to ACRs and CACR .....	10-6
10.2.3.10	ACR Address Improvements.....	10-6
10.2.3.11	Supervisor Protection.....	10-7
10.3	Debugging in a Virtual Environment.....	10-7
10.4	Virtual Memory Architecture Processor Support .....	10-7
10.4.1	Precise Faults .....	10-8
10.4.2	Supervisor/User Stack Pointers .....	10-8
10.4.3	Access Error Stack Frame Additions .....	10-8
10.5	MMU Definition .....	10-9
10.5.1	Effective Address Attribute Determination .....	10-9
10.5.2	MMU Functionality .....	10-10
10.5.3	MMU Organization.....	10-11
10.5.3.1	MMU Base Address Register (MMUBAR) .....	10-11
10.5.3.2	MMU Memory Map .....	10-11
10.5.3.3	MMU Control Register (MMUCR).....	10-12
10.5.3.4	MMU Operation Register (MMUOR).....	10-13

# CONTENTS

Paragraph Number	Title	Page Number
10.5.3.5	MMU Status Register (MMUSR).....	10-14
10.5.3.6	MMU Fault, Test, or TLB Address Register (MMUAR).....	10-15
10.5.3.7	MMU Read/Write Tag and Data Entry Registers (MMUTR and MMUDR) 10-15	
10.5.4	MMU TLB.....	10-17
10.5.5	MMU Operation .....	10-18
10.6	MMU Implementation .....	10-19
10.6.1	TLB Address Fields.....	10-20
10.6.2	TLB Replacement Algorithm .....	10-20
10.6.3	TLB Locked Entries.....	10-21
10.7	MMU Instructions.....	10-22

## Chapter 11 Debug Support

11.1	Overview.....	11-1
11.2	Signal Descriptions .....	11-3
11.2.1	Processor Status/Debug Data (PSTDDATA[7:0]) .....	11-4
11.3	Real-Time Trace Support.....	11-5
11.3.1	Begin Execution of Taken Branch (PST = 0x5).....	11-8
11.3.2	Processor Stopped or Breakpoint State Change (PST = 0xE) .....	11-9
11.3.3	Processor Halted (PST = 0xF) .....	11-9
11.4	Programming Model .....	11-10
11.4.1	Revision A Shared Debug Resources .....	11-13
11.4.2	Address Attribute Trigger Registers (AATR, AATR1).....	11-13
11.4.3	Address Breakpoint Registers (ABLR/ABLR1, ABHR/ABHR1) .....	11-15
11.4.4	BDM Address Attribute Register (BAAR).....	11-16
11.4.5	Configuration/Status Register (CSR).....	11-17
11.4.6	Data Breakpoint/Mask Registers (DBR/DBR1, DBMR/DBMR1) .....	11-19
11.4.7	Program Counter Breakpoint/Mask Registers (PBR, PBR1, PBR2, PBR3, PBMR) 11-20	
11.4.8	Trigger Definition Register (TDR).....	11-21
11.4.9	Extended Trigger Definition Register (XTDR).....	11-23
11.4.9.1	Resulting Set of Possible Trigger Combinations.....	11-25
11.4.10	PC Breakpoint ASID Control Register (PBAC).....	11-26
11.4.11	PC Breakpoint ASID Register (PBASID) .....	11-26
11.5	Background Debug Mode (BDM) .....	11-27
11.5.1	CPU Halt.....	11-28
11.5.2	BDM Serial Interface.....	11-29
11.5.2.1	Receive Packet Format .....	11-30
11.5.2.2	Transmit Packet Format.....	11-31
11.5.3	BDM Command Set.....	11-32

# CONTENTS

Paragraph Number	Title	Page Number
11.5.3.1	ColdFire BDM Command Format.....	11-33
11.5.3.1.1	Extension Words as Required.....	11-33
11.5.3.2	Command Sequence Diagrams.....	11-34
11.5.3.3	Command Set Descriptions .....	11-35
11.5.3.3.1	Read A/D Register (rareg/rdreg) .....	11-36
11.5.3.3.2	Write A/D Register (wareg/wdreg) .....	11-37
11.5.3.3.3	Read Memory Location (read).....	11-38
11.5.3.3.4	Write Memory Location (write) .....	11-39
11.5.3.3.5	Dump Memory Block (dump) .....	11-41
11.5.3.3.6	Fill Memory Block (fill) .....	11-43
11.5.3.3.7	Resume Execution (go) .....	11-45
11.5.3.3.8	No Operation (nop).....	11-46
11.5.3.3.9	Synchronize PC to the PSTDDATA Lines (sync_pc).....	11-47
11.5.3.3.10	Force Transfer Acknowledge (force_ta).....	11-47
11.5.3.3.11	Read Control Register (rcreg).....	11-49
11.5.3.3.12	Write Control Register (wcreg) .....	11-52
11.5.3.3.13	Read Debug Module Register (rdmreg) .....	11-53
11.5.3.3.14	Write Debug Module Register (wdmreg) .....	11-54
11.6	Real-Time Debug Support .....	11-55
11.6.1	Theory of Operation.....	11-55
11.6.1.1	Emulator Mode .....	11-57
11.6.2	Concurrent BDM and Processor Operation .....	11-58
11.7	Debug C Definition of PSTDDATA Outputs .....	11-58
11.7.1	User Instruction Set .....	11-59
11.7.2	Supervisor Instruction Set.....	11-64
11.8	ColdFire Debug History.....	11-65
11.8.1	ColdFire Debug Classic: The Original Definition.....	11-65
11.8.2	ColdFire Debug Revision B.....	11-66
11.8.3	ColdFire Debug Revision C.....	11-67
11.8.3.1	Debug Interrupts and Interrupt Requests (Emulator Mode).....	11-67
11.9	Motorola-Recommended BDM Pinout.....	11-68

## Chapter 12 Test

12.1	Scan Chains.....	12-2
12.1.1	Core Scan Chains.....	12-2
12.1.2	Wrapper Scan Chains.....	12-2
12.1.3	Scan Chains Block Diagram .....	12-3
12.2	Test Wrapper.....	12-3
12.2.1	Features .....	12-4
12.2.2	Wrapper Cells .....	12-5

# CONTENTS

Paragraph Number	Title	Page Number
12.2.3	Block Diagram .....	12-6
12.2.4	Timing .....	12-8
12.2.4.1	CF4eTW Testing of CF4e Core Inputs .....	12-8
12.2.4.2	CF4eTW Testing of CF4e Core Outputs .....	12-11
12.2.4.3	CF4eTW Testing of Noncore Inputs .....	12-13
12.2.4.4	CF4eTW Testing of Noncore Outputs .....	12-15
12.3	BIST .....	12-17
12.3.1	BIST Memory Controllers .....	12-18
12.3.2	BIST Core Ports .....	12-19
12.3.3	Power Analysis .....	12-20
12.3.4	Staging of Memories .....	12-20
12.3.5	Testing Algorithms .....	12-21
12.3.5.1	March C+ Algorithm .....	12-21
12.3.6	ROM BIST Algorithm .....	12-22
12.3.6.1	Modify BIST ROM Signature Script—Part 1 .....	12-23
12.3.6.2	Modify BIST ROM Signature Script—Part 2 .....	12-24
12.3.7	BIST Test Modes .....	12-25
12.3.8	Memory Data Retention .....	12-26
12.3.9	Timing .....	12-26
12.3.9.1	Memory Clock Determination .....	12-27
12.3.10	Timing Diagrams .....	12-28
12.4	Integration Connections .....	12-32
12.5	Test Controller .....	12-32
12.5.1	MTMOD[2:0] Encodings .....	12-32

## Appendix A Core Interface Timing Characteristics

# CONTENTS

**Paragraph  
Number**

**Title**

**Page  
Number**

# ILLUSTRATIONS

Figure Number	Title	Page Number
1-1	CF4e Core Block Diagram.....	1-3
1-2	V4 Core Block Diagram .....	1-4
1-3	ColdFire Programming Model.....	1-6
1-4	Organization of Integer Data Format in Data Registers .....	1-10
1-5	Organization of Integer Data Formats in Address Registers .....	1-10
1-7	Two's Complement, Signed Fractional Equation.....	1-11
1-6	Memory Operand Addressing.....	1-11
1-8	Floating-Point Data Formats.....	1-12
1-9	Mantissa .....	1-12
2-1	Programming Model .....	2-2
2-2	User Programming Model.....	2-4
2-3	Condition Code Register (CCR) .....	2-5
2-4	EMAC Register Set.....	2-6
2-5	Floating-Point Programmer's Model .....	2-7
2-6	Supervisor Programming Model.....	2-8
2-7	Status Register (SR).....	2-8
2-8	Vector Base Register (VBR).....	2-9
2-9	Module Base Address Register (MBAR) .....	2-11
4-1	Floating-Point Data Formats.....	4-3
4-2	Mantissa .....	4-4
4-3	Normalized Number Format .....	4-4
4-4	Zero Format .....	4-4
4-5	Infinity Format .....	4-5
4-6	Not-a-Number Format .....	4-5
4-7	Denormalized Number Format .....	4-5
4-8	Floating-Point Programmer's Model .....	4-7
4-9	Floating-Point Control Register (FPCR) .....	4-8
4-10	Floating-Point Status Register (FPSR) .....	4-9
4-11	Intermediate Result Format.....	4-12
4-12	Rounding Algorithm Flowchart.....	4-14
4-13	Floating-Point State Frame Contents .....	4-26
5-1	Multiply-Accumulate Functionality Diagram.....	5-2
5-2	Infinite Impulse Response (IIR) Filter .....	5-3
5-3	Four-Tap FIR Filter.....	5-3
5-4	Fractional Alignment .....	5-4
5-5	Signed and Unsigned Integer Alignment.....	5-5

# ILLUSTRATIONS

Figure Number	Title	Page Number
5-6	EMAC Register Set.....	5-6
5-7	MAC Status Register (MACSR).....	5-7
5-8	Two's Complement, Signed Fractional Equation.....	5-13
6-1	CF4e ColdFire Processor Complex Block Diagram.....	6-3
6-2	OAGComputeEngine Register Renaming Resources.....	6-8
6-3	Sequence-Related OEP Sequence Stall .....	6-11
6-4	EMAC-Specific OEP Sequence Stall .....	6-14
6-5	for_loop Example.....	6-16
6-6	CF4e Ex Execute Engines within the OEP .....	6-17
7-1	Exception Stack Frame .....	7-4
8-1	Generic CF4e Block Diagram.....	8-2
8-2	Local Memory Block Diagram Showing Cache, KRAM, and KROM Controllers .....	8-3
8-3	ColdFire Core Synchronous Memory Interface.....	8-4
8-4	Synchronous Memory Timing Diagram .....	8-4
8-5	Synchronous Memory Interface Block Diagram .....	8-5
8-6	Version 4 Cache Block Diagram .....	8-7
8-7	Cache Organization and Line Format (32-Kbyte Cache Size Shown) .....	8-14
8-8	Cache Organization and Line Format (32 Kbyte Cache Size shown) .....	8-19
8-9	SRAM Base Address Registers (RAMBARn) .....	8-24
8-10	ROM Base Address Registers (ROMBAR0/ROMBAR1) .....	8-29
8-11	Data Cache Organization and Line Format .....	8-34
8-12	Data Cache—A: at Reset, B: after Invalidation, C and D: Loading Pattern.....	8-35
8-13	Data Caching Operation.....	8-36
8-14	Write-Miss in Copyback Mode.....	8-41
8-15	Data Cache Locking.....	8-45
8-16	Cache Control Register (CACR) .....	8-46
8-17	Access Control Register Format (ACRn) .....	8-49
8-18	An Format (Data Cache).....	8-50
8-19	An Format (Instruction Cache) .....	8-50
8-20	Instruction Cache Line State Diagram.....	8-52
8-21	Data Cache Line State Diagram—Copyback Mode .....	8-54
8-22	Data Cache Line State Diagram—Write-Through Mode .....	8-54
9-1	Generic CF4e Block Diagram.....	9-1
9-2	Basic Read and Write Cycles.....	9-9
9-3	Pipelined Read and Write .....	9-10
9-4	Address Hold Followed By 1- and 0-Wait State Cycles.....	9-11
9-5	mapb and mahb Generated Mid-Data Phase.....	9-12
9-6	mahb Generation for 1X Clock Mode .....	9-12
9-7	Line Access Read with Zero Wait States .....	9-14
9-8	Line Access Read with One Wait State .....	9-15
9-9	Line Access Write with Zero Wait States.....	9-15
9-10	Line Access Write with One Wait State .....	9-16
9-11	Multiplexed M-Bus Structure .....	9-16



# ILLUSTRATIONS

Figure Number	Title	Page Number
9-12	Multiplexed M-Bus Operation .....	9-17
10-1	CF4e Processor Core Block with MMU .....	10-3
10-2	Exception Stack Frame .....	10-8
10-3	MMU Base Address Register .....	10-11
10-4	MMU Control Register (MMUCR) .....	10-12
10-5	MMU Operation Register (MMUOR) .....	10-13
10-6	MMU Status Register (MMUSR) .....	10-14
10-7	MMU Fault, Test, or TLB Register (MMUAR) .....	10-15
10-8	MMU Read/Write TLB Tag Register (MMUTR) .....	10-16
10-9	MMU Read/Write TLB Data Register .....	10-16
10-10	K-Bus Address and Attributes Generation .....	10-19
10-11	Version 4 ColdFire MMU Harvard TLB .....	10-22
11-1	Processor/Debug Module Interface .....	11-1
11-2	PSTCLK Timing .....	11-4
11-3	PSTDDATA: Single-Cycle Instruction Timing .....	11-5
11-4	Example JMP Instruction Output on PSTDDATA .....	11-8
11-5	Debug Programming Model .....	11-11
11-6	Address Attribute Trigger Registers (AATR, AATR1) .....	11-14
11-7	Address Breakpoint Registers (ABLR, ABHR, ABLR1, ABHR1) .....	11-15
11-8	BDM Address Attribute Register (BAAR) .....	11-16
11-9	Configuration/Status Register (CSR) .....	11-17
11-10	Data Breakpoint/Mask Registers (DBR/DBR1 and DBMR/DBMR1) .....	11-19
11-11	Program Counter Breakpoint Registers (PBR, PBR1, PBR2, PBR3) .....	11-21
11-12	Program Counter Breakpoint Mask Register (PBMR) .....	11-21
11-13	Trigger Definition Register (TDR) .....	11-22
11-14	Extended Trigger Definition Register (XTDR) .....	11-24
11-15	PC Breakpoint ASID Control Register (PBAC) .....	11-26
11-16	PC Breakpoint ASID Register (PBASID) .....	11-27
11-17	Maximum BDM Serial Interface Timing .....	11-30
11-18	Receive BDM Packet .....	11-30
11-19	Transmit BDM Packet .....	11-31
11-20	BDM Command Format .....	11-33
11-21	Command Sequence Diagram .....	11-34
11-23	rareg/rdreg Command Sequence .....	11-36
11-22	rareg/rdreg Command Format .....	11-36
11-25	wareg/wdreg Command Sequence .....	11-37
11-24	wareg/wdreg Command Format .....	11-37
11-27	read Command Sequence .....	11-38
11-26	read Command/Result Formats .....	11-38
11-28	write Command Format .....	11-39
11-29	write Command Sequence .....	11-40
11-30	dump Command/Result Formats .....	11-41
11-31	dump Command Sequence .....	11-42

# ILLUSTRATIONS

Figure Number	Title	Page Number
11-32	fill Command Format.....	11-43
11-33	fill Command Sequence.....	11-44
11-35	go Command Sequence.....	11-45
11-34	go Command Format .....	11-45
11-37	nop Command Sequence.....	11-46
11-36	nop Command Format .....	11-46
11-39	sync_pc Command Sequence .....	11-47
11-38	sync_pc Command Format .....	11-47
11-41	force_TA Command Sequence.....	11-48
11-40	force_ta Command.....	11-48
11-43	rcreg Command Sequence .....	11-49
11-42	rcreg Command/Result Formats .....	11-49
11-45	wcreg Command Sequence.....	11-52
11-44	wcreg Command/Result Formats.....	11-52
11-47	rdmreg Command Sequence.....	11-53
11-46	rdmreg bdm Command/Result Formats.....	11-53
11-49	wdmreg Command Sequence .....	11-54
11-48	wdmreg BDM Command Format .....	11-54
11-1	Recommended BDM Connector.....	11-69
12-1	CF4e Scan Chains Block Diagram .....	12-3
12-2	CF4e and Test Wrapper in SoC .....	12-4
12-3	CF4e Core Shared Wrapper Cells.....	12-5
12-4	CF4e Core Dedicated Input Wrapper Cell (P Cell) .....	12-6
12-5	Example of Registered CF4eTW Architecture .....	12-7
12-6	Scans and Flops.....	12-8
12-7	CF4eTW Input to CF4e Core Scan Stuck-At Vector Example .....	12-9
12-8	CF4eTW Input to CF4e Core Scan Delay Vector Example .....	12-10
12-9	CF4e Core to CF4eTW Output Scan Stuck-At Vector Example.....	12-12
12-10	CF4e Core to CF4eTW Output Scan Delay Vector Example.....	12-13
12-11	CF4eTW to Non-Core Input Scan Stuck-At Vector Example.....	12-14
12-12	CF4eTW to Non-Core Delay Scan Vector Example .....	12-15
12-13	Non-Core to CF4eTW Input Scan Stuck-At Vector Example.....	12-16
12-14	Non-Core to CF4eTW Input Scan Delay Vector Example.....	12-17
12-15	CF4e BIST Hierarchy .....	12-18
12-16	Flow of Characterization Method .....	12-25
12-17	March C+ Algorithm .....	12-26
12-18	512 x 32 RAM BIST Clock Cycles .....	12-27
12-19	512 x 32 ROM BIST Clock Cycles .....	12-27
12-20	PBIST Initialization .....	12-28
12-21	EBIST Timing Diagram for an 8-Kbyte Cache Tag Array.....	12-29
12-22	EBIST Timing Diagram For An 8-Kbyte Cache Data Array .....	12-30
12-23	EBIST Timing Diagram For A 2-Kbyte KRAM0 Array.....	12-31
12-24	EBIST Timing Diagram For 2-Kbyte KROM0 Array.....	12-31

# TABLES

Table Number	Title	Page Number
1-1	ColdFire CPU Space Assignments .....	1-7
1-2	Integer Data Formats.....	1-9
1-3	ColdFire Effective Addressing Modes.....	1-13
1-4	V4 New Instruction Summary .....	1-14
1-5	User-Mode Instruction Set Summary .....	1-16
1-6	Supervisor-Mode Instruction Set Summary.....	1-20
2-1	CCR Field Descriptions .....	2-5
2-2	Status Field Descriptions .....	2-8
2-3	MBAR Field Descriptions .....	2-11
2-4	ColdFire CPU Registers.....	2-12
4-1	Notational Conventions .....	4-2
4-2	Floating-Point Addressing Modes .....	4-3
4-3	Real Format Summary .....	4-6
4-4	FPCR Field Descriptions .....	4-8
4-5	FPSR Field Descriptions.....	4-9
4-6	Tie-Case Example.....	4-15
4-7	Round Mode Error Bounds.....	4-15
4-8	FPCC Encodings.....	4-16
4-9	Floating-Point Conditional Tests .....	4-18
4-10	Floating-Point Exception Vectors.....	4-19
4-11	Exception Priorities.....	4-20
4-12	BSUN Exception Enabled/Disabled Results .....	4-22
4-13	INAN Exception Enabled/Disabled Results .....	4-22
4-14	IDE Exception Enabled/Disabled Results .....	4-23
4-15	Possible Operand Errors .....	4-23
4-16	OPERR Exception Enabled/Disabled Results .....	4-23
4-17	OVFL Exception Enabled/Disabled Results.....	4-24
4-18	UNFL Exception Enabled/Disabled Results.....	4-25
4-19	DZ Exception Enabled/Disabled Results.....	4-25
4-20	Inexact Rounding Mode Values.....	4-25
4-21	INEX Exception Enabled/Disabled Results.....	4-26
4-22	Format Word Field Descriptions .....	4-27
4-23	Floating-Point Instruction Formats .....	4-28
4-24	Instruction Format Terminology.....	4-29
4-25	Floating-Point Instruction Execution Times, , .....	4-30
4-26	Key Programming Model Differences.....	4-31

# TABLES

Table Number	Title	Page Number
4-27	68K/ColdFire Operation Sequence 1 .....	4-32
4-28	68K/ColdFire Operation Sequence 2 .....	4-32
4-29	68K/ColdFire Operation Sequence 3 .....	4-32
5-1	MACSR Field Descriptions .....	5-7
5-2	Summary of S/U, F/I, and R/T Control Bits .....	5-9
5-3	EMAC Instruction Summary .....	5-12
6-1	CFxCore Processor Execution Latency .....	6-2
6-2	V4 RTS Execution Times .....	6-5
6-3	Instructions that Make Results Available to Subsequent Instructions .....	6-12
6-4	FPU Execution Example.....	6-15
6-5	V4 ColdFire Compute Engine Location .....	6-18
6-6	Misaligned Operand References .....	6-22
6-7	Move Byte and Word Execution Times.....	6-23
6-8	Move Long Execution Times.....	6-23
6-9	MAC and Miscellaneous Move Execution Times .....	6-24
6-10	One-Operand Instruction Execution Times .....	6-25
6-11	Two-Operand Instruction Execution Times.....	6-25
6-12	Miscellaneous Instruction Execution Times .....	6-27
6-13	General Branch Instruction Execution Times.....	6-28
6-14	Bcc Instruction Execution Times.....	6-28
6-15	EMAC Instruction Execution Times .....	6-29
6-16	FPU Instruction Execution Times, .....	6-30
7-1	Exception Vector Assignments.....	7-2
7-2	Format/Vector Word.....	7-5
7-3	Exceptions.....	7-6
7-4	OEP EX Cycle Operations.....	7-8
8-1	Synchronous Memory Truth Table (Sampled at Positive Edge of CLK) .....	8-5
8-2	KRAM Size.....	8-9
8-3	KRAM Memory Array Connections .....	8-9
8-4	KRAM0/KRAM1 Array Address Connection.....	8-10
8-5	KRAM0/KRAM1 Byte Write Enables .....	8-10
8-6	KRAM0/KRAM1 Size .....	8-11
8-7	KROM{0,1} Memory Array Connections.....	8-11
8-8	KROM Array Address Connection.....	8-12
8-9	Instruction Cache Sizes and Configurations .....	8-13
8-10	Instruction Cache Size .....	8-14
8-11	Instruction Cache Memory Array Connections .....	8-14
8-12	Instruction Cache Data Array Address Connection.....	8-16
8-13	Instruction Cache Tag Array Address Connection .....	8-16
8-14	Instruction Cache Tag Array Write Data Connection.....	8-16
8-15	Data Cache Sizes and Configurations.....	8-18
8-16	Data Cache Size .....	8-19
8-17	Data Cache Memory Array Connections.....	8-19

# TABLES

Table Number	Title	Page Number
8-18	Data Cache Data Array Address Connection.....	8-21
8-19	Data Cache Tag Array Address Connection.....	8-21
8-20	Data Cache Tag Array Write Data Connection .....	8-21
8-21	RAMBARn Field Description .....	8-24
8-22	KRAM Size Configuration .....	8-25
8-23	Examples of Typical RAMBAR Settings.....	8-27
8-24	ROMBAR Field Descriptions.....	8-30
8-25	KROM Size Configuration .....	8-30
8-26	Examples of Typical ROMBAR Settings.....	8-32
8-27	Valid and Modified Bit Settings .....	8-34
8-28	CACR Field Descriptions .....	8-46
8-29	ACRn Field Descriptions .....	8-49
8-30	Instruction Cache Line State Transitions.....	8-53
8-31	Data Cache Line State Transitions.....	8-54
8-32	Data Cache Line State Transitions (Previous State Invalid).....	8-56
8-33	Data Cache Line State Transitions (Previous State Valid).....	8-56
8-34	Data Cache Line State Transitions (Previous State Modified).....	8-57
9-1	CF4e Pin Characteristics.....	9-2
9-2	M-Bus Signals.....	9-6
9-3	Processor Operand Representation .....	9-12
9-4	mrdata Requirements for Read Transfers .....	9-13
9-5	mwdata Bus Requirements for Write Transfers.....	9-13
9-6	Allowable Line Access Patterns .....	9-14
10-1	New ACR and CACR Bits.....	10-6
10-2	Fault Status Encodings.....	10-8
10-3	MMU Base Address Register Field Descriptions.....	10-11
10-4	MMU Memory Map .....	10-12
10-5	MMUCR Field Descriptions.....	10-13
10-6	MMUOR Field Descriptions.....	10-13
10-7	MMUSR Field Descriptions .....	10-15
10-8	MMUAR Field Descriptions.....	10-15
10-9	MMUTR Field Descriptions .....	10-16
10-10	MMUDR Field Descriptions.....	10-17
10-11	Version 4 K-Bus Memory Pipelines .....	10-18
10-12	K-Bus Pipeline Cycles .....	10-18
10-13	PLRU State Bits .....	10-20
11-1	Debug Module Signals.....	11-3
11-2	PSTDDATA: Sequential Execution of Single-Cycle Instructions .....	11-4
11-3	PSTDDATA: Data Operand Captured.....	11-5
11-4	Processor Status Encoding.....	11-7
11-5	0xE Status Posting .....	11-9
11-6	BDM/Breakpoint Registers.....	11-11
11-7	Rev. A Shared BDM/Breakpoint Hardware .....	11-13

# TABLES

Table Number	Title	Page Number
11-8	AATR and AATR1 Field Descriptions.....	11-14
11-9	ABLR and ABLR1 Field Description.....	11-16
11-10	ABHR and ABHR1 Field Description.....	11-16
11-11	BAAR Field Descriptions .....	11-16
11-12	CSR Field Descriptions .....	11-17
11-13	DBRn Field Descriptions .....	11-20
11-14	DBMRn Field Descriptions .....	11-20
11-15	Access Size and Operand Data Location .....	11-20
11-16	PBR, PBR1, PBR2, PBR3 Field Descriptions .....	11-21
11-17	PBMR Field Descriptions .....	11-21
11-18	TDR Field Descriptions .....	11-22
11-19	XTDR Field Descriptions .....	11-24
11-20	PBAC Field Descriptions.....	11-26
11-21	PBASID Field Descriptions.....	11-27
11-22	Receive BDM Packet Field Description .....	11-31
11-23	Transmit BDM Packet Field Description .....	11-31
11-24	BDM Command Summary .....	11-32
11-25	BDM Field Descriptions .....	11-33
11-26	Definition of DRc Encoding—Read.....	11-53
11-27	PSTDDATA Nibble/CSR[BSTAT] Breakpoint Response.....	11-55
11-28	Exception Vector Assignments.....	11-56
11-29	PSTDDATA Specification for User-Mode Instructions.....	11-59
11-30	PSTDDATA Values for User-Mode Multiply-Accumulate Instructions .....	11-62
11-31	PSTDDATA Values for User-Mode Floating-Point Instructions.....	11-63
11-32	Data Markers and FPU Operand Format Specifiers .....	11-64
11-33	PSTDDATA Specification for Supervisor-Mode Instructions .....	11-64
12-1	CF4e Core Scan Chains .....	12-2
12-2	CF4e Wrapper Scan Chains .....	12-2
12-3	BIST Core Pins .....	12-19
12-4	BIST Cycles .....	12-27
12-5	EBIST Tag Output Data.....	12-29
12-6	CF4e Motorola Test Mode Encodings.....	12-32
A-1	Timing Budget Variables for Various Process and Frequency Targets.....	13-1

# About This Book

---

The primary objective of this user's manual is to define the functionality of CF4e ColdFire microprocessor core. The CF4e implementation of the Version 4 (V4) core includes the floating-point unit (FPU), enhanced multiply-accumulate unit (EMAC), and memory management unit (MMU) that are defined as optional in the V4 architecture.

The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

To locate any published errata or updates for this document, refer to the world-wide web at <http://www.motorola.com/coldfire>.

## Audience

This manual is intended for developers who want to use the CF4e processor core in their products. It is assumed that the reader understands operating systems, microprocessor system design, general principles of software and hardware, and basic details of the ColdFire architecture.

## Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, "Introduction," includes general descriptions of the CF4e implementation of the V4 architecture, focussing in particular on new features.
- Chapter 2, "Registers," describes the organization of CFe general-purpose and control registers in the user and supervisor programming models.
- Chapter 3, "Instructions," provides a pointer to the instruction descriptions in the *ColdFire Family Programmer's Reference Manual (PRM)*.
- Chapter 4, "Floating-Point Unit (FPU)," describes instructions implemented in the floating-point unit (FPU) designed for use with the ColdFire family of microprocessors. The FPU conforms to the American National Standards Institute (ANSI)/Institute of Electrical and Electronics Engineers (IEEE) *Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754).

- Chapter 5, “Enhanced Multiply-Accumulate Unit (EMAC),” describes the functionality, microarchitecture, and performance of the enhanced multiply-accumulate (EMAC) unit in the ColdFire family of processors.
- Chapter 6, “Instruction Pipeline and Timing,” describes performance features of the CF4e ColdFire processor pipeline structure. It is intended as a guide for developing compilers or optimizing assembly language application code. It describes the basic CF4e pipeline strategy, contrasting it with Version 2 and 3 designs. It also provides performance-related details of the instruction fetch and operand execution pipelines (IFP and OEP).
- Chapter 7, “Exception Processing,” describes CFe exception processing, focusing on differences from previous ColdFire versions. In particular, additional encodings have been added to the fault status (FS) field in the exception stack frame to indicate exceptions related to translation lookaside buffers (TLBs). This provides CF4e core designs with precise, recoverable faults for all K-Bus references to support demand-paged memory accesses.
- Chapter 8, “Local Memory,” describes the implementation of the V4 local memory specification, which implements a Harvard memory architecture, including separate caches, ROM, RAM, and the necessary buses and registers to support instruction and data memory. This chapter consists of the following major sections:
  - Section 8.5, “SRAM Overview,” describes the on-chip static RAM (SRAM) implementation. It covers general operations, configuration, and initialization. It also provides information and examples showing how to minimize power consumption when using the SRAM.
  - Section 8.6, “ROM Overview,” describes the on-chip ROM implementation. It covers general operations, configuration, and initialization. It also provides information and examples showing how to minimize power consumption when using the ROM.
  - Section 8.7, “Cache Overview,” describes the cache implementation, including organization, configuration, and coherency. It describes cache operations and how the caches interface with other memory structures.
- Chapter 9, “Core Interface,” describes the CF4e core interface and provides an overview of the functional operation of the master bus (M-Bus).
- Chapter 10, “Memory Management Unit (MMU),” describes the ColdFire virtual memory management unit (MMU), which provides virtual-to-physical address translation and memory access control.
- Chapter 11, “Debug Support,” describes the Revision D enhanced hardware debug support in the ColdFire Version 4. This revision of the ColdFire debug architecture encompasses earlier revisions. An expanded set of debug functionality is defined as Revision B (or Rev. B). The further enhanced debug architecture implemented in the Version 4 ColdFire is known as Revision C (or Rev. C).



- Chapter 12, “Test,” provides an overview of test features of CF4e. Some of the features, such as MBist hardware, are included in the CF4e design. The scan and wrapper methodology, described later in the chapter, are part of the CF4e design but are described here as a reference for properly designing CF4e for test.

This manual also includes an index.

## Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the ColdFire architecture.

### General Information

The following documentation provides useful information about the ColdFire architecture and computer architecture in general:

### ColdFire Documentation

The ColdFire documentation is available from the sources listed on the back cover of this manual. Document order numbers are included in parentheses for ease in ordering.

- *ColdFire Microprocessor Family Programmer's Reference Manual*, or *PRM* (COLDFIREPRM/AD, Rev 2)
- *Using Microprocessors and Microcomputers: The Motorola Family*, William C. Wray, Ross Bannatyne, Joseph D. Greenfield

Additional literature on ColdFire implementations is being released as new processors become available. For a current list of ColdFire documentation, refer to the World Wide Web at <http://www.motorola.com/ColdFire>.

## Conventions

This document uses the following notational conventions:

MNEMONICS	In text, instruction mnemonics are shown in uppercase.
mnemonics	In code and tables, instruction mnemonics are shown in lowercase.
COMMANDS	Command names are shown in small caps.
<i>italics</i>	Italics indicate variable command parameters. Book titles in text are set in italics.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
REG[FIELD]	Abbreviations for registers are shown in uppercase. Specific bits, fields, or ranges appear in brackets. For example, RAMBAR[BA] identifies the base address field in the RAM base address register.

## Acronyms and Abbreviations

nibble	A 4-bit data unit
byte	An 8-bit data unit
word	A 16-bit data unit
longword	A 32-bit data unit
x	In some contexts, such as signal encodings, x indicates a don't care.
<i>n</i>	Used to express an undefined numerical value
¬	NOT logical operator
&	AND logical operator
	OR logical operator

## Acronyms and Abbreviations

Table i lists acronyms and abbreviations used in this document.

**Table i. Acronyms and Abbreviated Terms**

Term	Meaning
ALU	Arithmetic logic unit
BDM	Background debug mode
BIST	Built-in self test
BSDL	Boundary-scan description language
CODEC	Code/decode
DMA	Direct memory access
DSP	Digital signal processing
EA	Effective address
EMAC	Enhanced multiply-accumulate unit
FIFO	First-in, first-out
IEEE	Institute for Electrical and Electronics Engineers
IFP	Instruction fetch pipeline
IPL	Interrupt priority level
JTAG	Joint Test Action Group
LIFO	Last-in, first-out
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
MAC	Multiple accumulate unit

**Table i. Acronyms and Abbreviated Terms (Continued)**

<b>Term</b>	<b>Meaning</b>
MBAR	Memory base address register
MSB	Most-significant byte
msb	Most-significant bit
Mux	Multiplex
NOP	No operation
OEP	Operand execution pipeline
PC	Program counter
PCLK	Processor clock
PLL	Phase-locked loop
PLRU	Pseudo least recently used
POR	Power-on reset
RISC	Reduced instruction set computing
Rx	Receive
SIM	System integration module
SOF	Start of frame
TAP	Test access port
Tx	Transmit



# Chapter 1

## Introduction

This section is an overview of the CF4e ColdFire microprocessor core. The CF4e implementation of the Version 4 (V4) core includes the floating-point unit (FPU), enhanced multiply-accumulate unit (EMAC), and memory management unit (MMU) that are defined as optional in the V4 architecture.

### 1.1 Core Overview

The V4 core includes a Harvard memory architecture, branch cache acceleration logic, and limited superscalar dual-instruction issue capabilities. The V4 core provides 1.54 Dhrystone 2.1 MIPS per MHz.

### 1.2 Features

The CF4e includes the following features defined as optional in the V4 core architecture:

- Floating-point unit (FPU)
- Virtual memory management unit (MMU)
- Enhanced multiply-accumulate unit (EMAC) for increased signal processing functionality plus backward code compatibility with the MAC unit of previous ColdFire processors

V4 architecture features are defined as follows:

- Variable-length RISC, clock-multiplied core
- Revision B of the ColdFire instruction set architecture (ISA\_B) providing new instructions to improve performance and code density
- Two independent, decoupled pipelines—four-stage instruction fetch pipeline (IFP) and five-stage operand execution pipeline (OEP) for increased performance.
- Ten-instruction, FIFO buffer decouples the IFP and OEP
- Limited superscalar design approaches dual-issue performance with the cost of a scalar execution pipeline
- Two-level branch acceleration mechanism with a branch cache plus a prediction table for increased performance of conditional Bcc instructions
- 32-bit address bus supporting 4 Gbytes of linear address space

## CF4e Implementation Block Diagram

- 32-bit data bus
- 16 user-accessible, 32-bit-wide, general-purpose registers
- Supervisor/user modes for system protection
- Two separate stack pointer (A7) registers—the supervisor stack pointer (SSP) and the user stack pointer (USP)—provide the required isolation between operating modes to support the MMU.
- Vector base register to relocate the exception-vector table
- Optimized for high-level language constructs

## 1.3 CF4e Implementation Block Diagram

Figure 1-1 shows the key elements of the CF4e core implementation. Individual pipeline stages are defined and described in detail in Chapter 6, “Instruction Pipeline and Timing.”

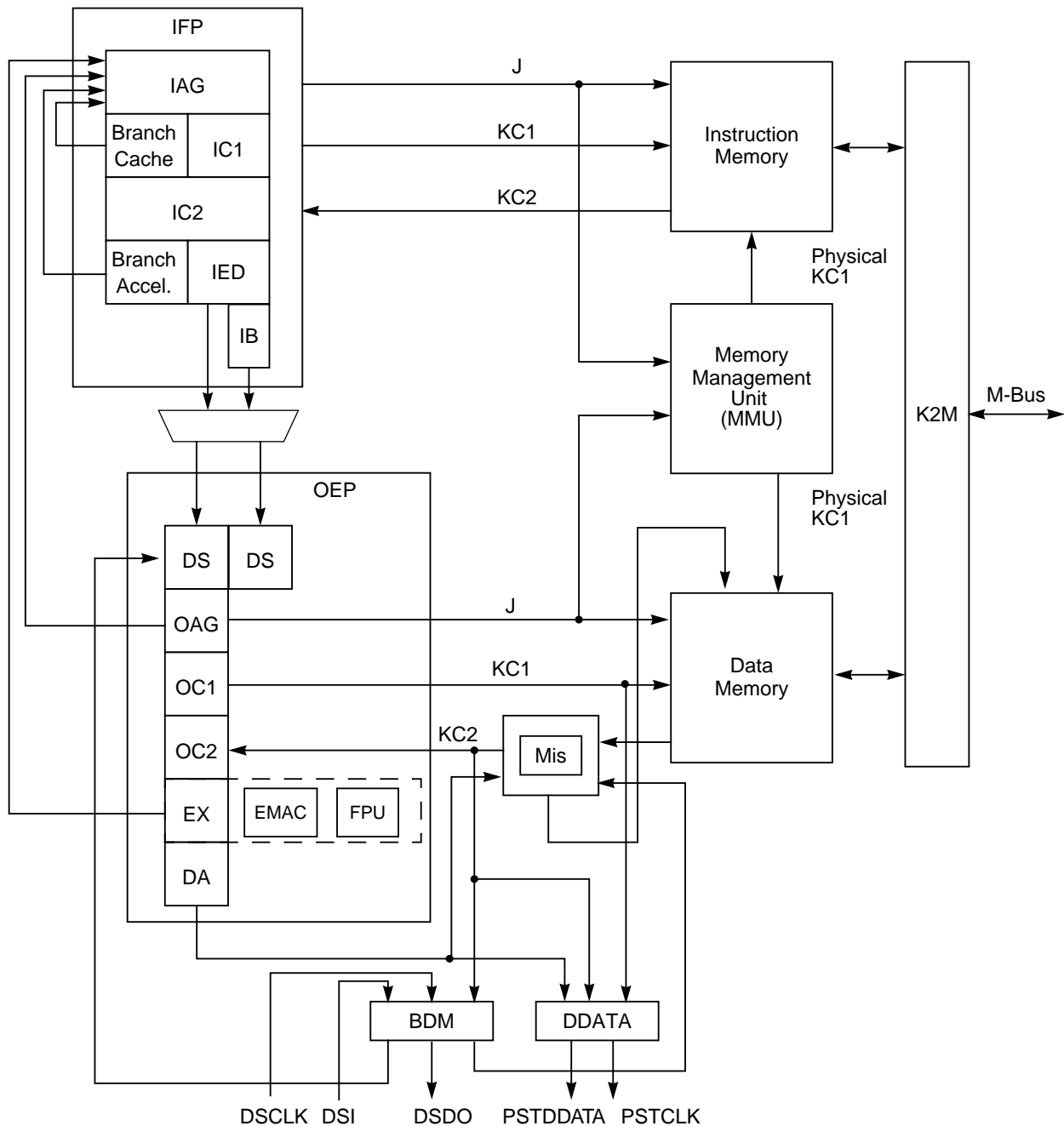


Figure 1-1. CF4e Core Block Diagram

## 1.4 Architectural Summary

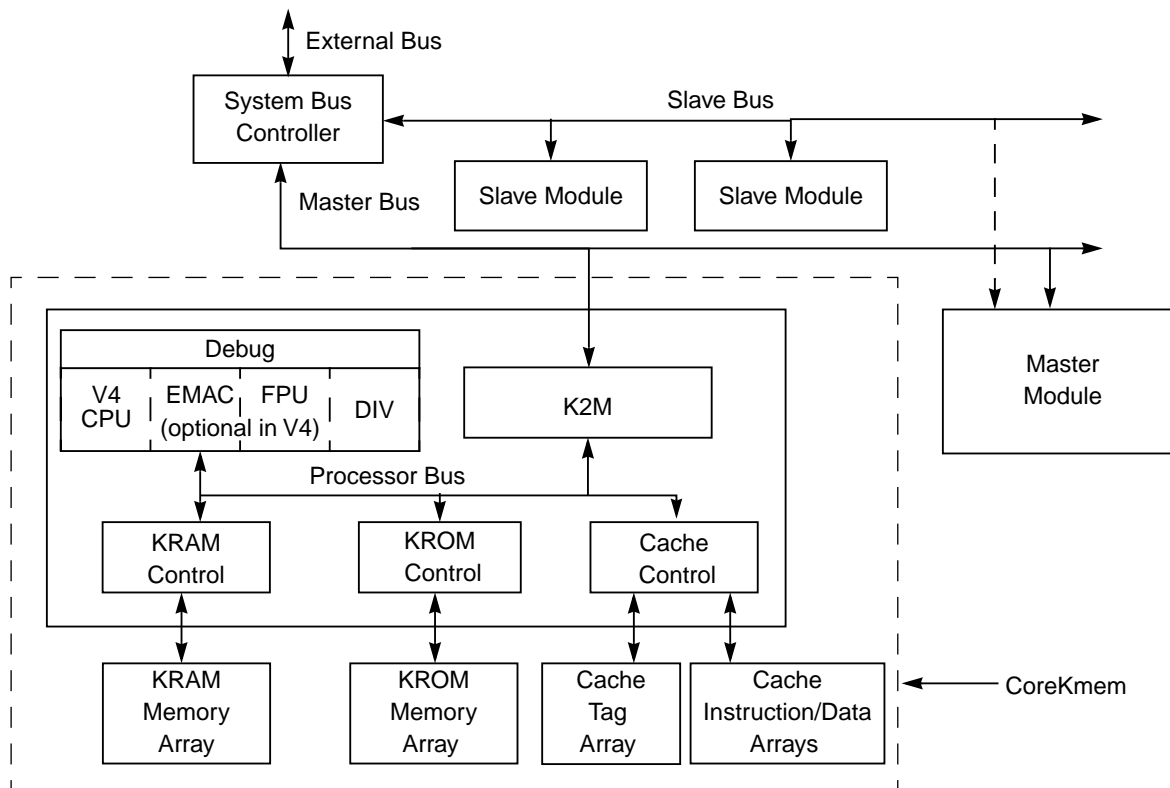
Figure 1-2 shows the standard CF4e microprocessor configuration. The hierarchical bus structure provides varying layers of data bandwidth and supports an efficient partitioning of optional on-chip modules. The bus hierarchy is as follows:

1. Processor-local bus (K-Bus)
2. Master bus (M-Bus)

## Architectural Summary

3. Slave bus (S-bus)
4. External bus (E-bus)

The CF4e reference design is defined by the V4 core hierarchy. The CoreKmem boundary includes the core design and processor-local memories required for a given design.



**Figure 1-2. V4 Core Block Diagram**

The processor connects to a number of memory controllers and a bus controller through a local, high-speed bus. Processor-local memories include caches, RAM, and ROM. V4 memory controllers support a range of sizes, allowing the ability to specify the optimum memory organization for a given application. The K2M bus controller controls transfers on the processor-local bus and initiates and controls all accesses onto the next-level system bus, the master bus (M-Bus). The processor-local bus is designed to maximize bandwidth from high-speed memories to support efficient instruction execution.

The M-Bus is the primary interface between the core and other system-level components. Devices that can initiate bus cycles are typically connected to the M-Bus. Example modules include direct-memory access devices (DMA) or another ColdFire processor complex. The M-Bus is typically connected to a system interface module (SIM), which provides two interfaces: one to a simple, on-chip slave bus (S-bus) and another to an application-specific external bus (E-bus). The S-bus generally is connected to any number of standard peripheral modules, such as timers, universal asynchronous receiver/transmitters (UARTs), other serial communication devices, and parallel ports. Use of a standard Motorola-defined bus protocol promotes reuse of these synthesizable modules. Specific implementation and



protocol details of the external bus can vary widely, depending on system requirements.

The V4 design allows a core to operate at any integer multiplier ( $n = 1, 2, 3, \dots$ ) faster than the rest of the design. For multiple clock domains, the boundary is the M-Bus; that is, the processor complex operates at the higher frequency, while the M-Bus and the rest of the microprocessor operate at the slower speed. This well-defined, easy-to-use clock boundary simplifies interface design and timing and eases production test complications.

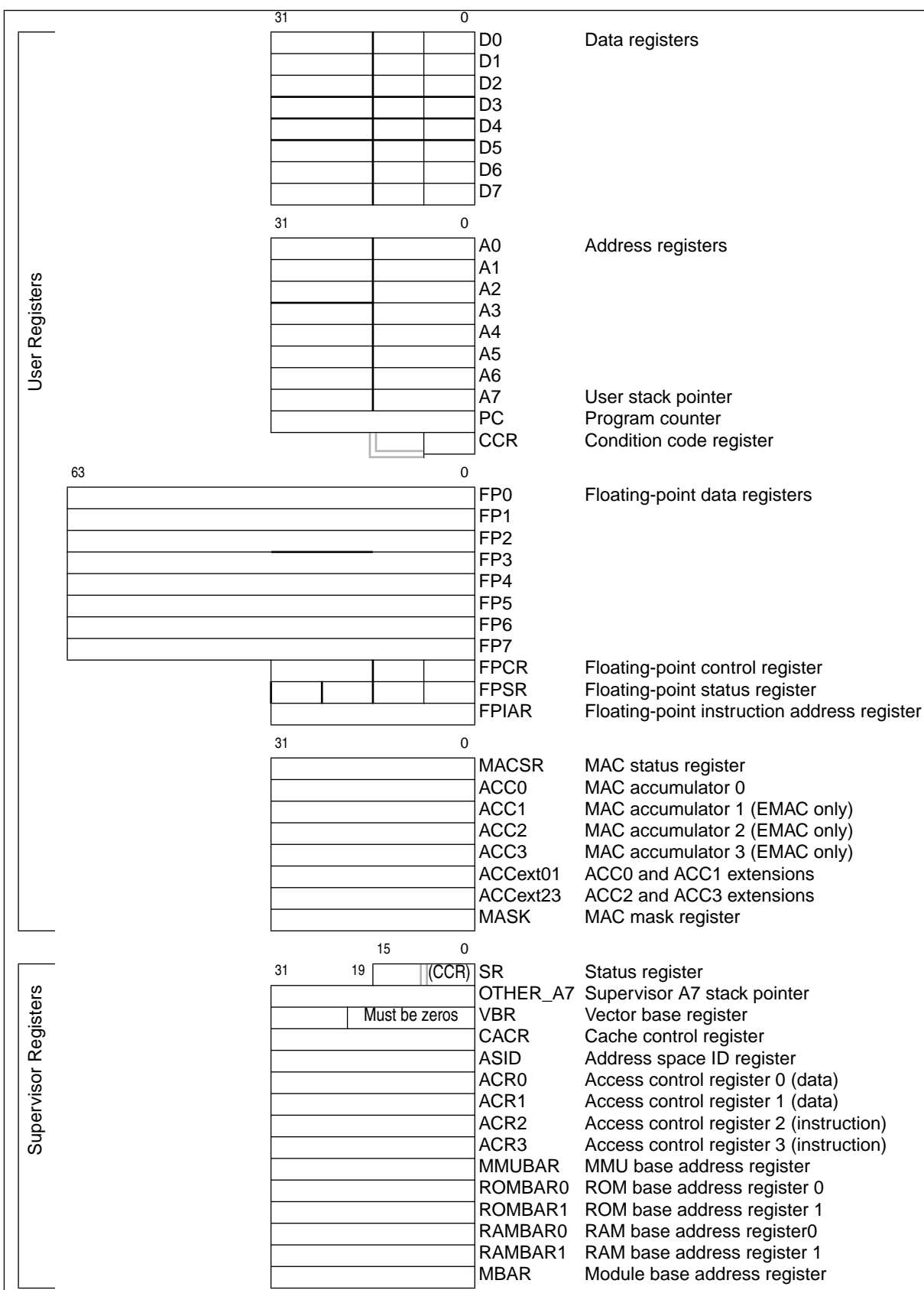
The overall ColdFire implementation strategy of 100% synthesizable designs and use of compiled memory arrays coupled with the modular system architecture allows easy migration to any process technology and provides cost-effective integration capabilities while targeting a variety of operating voltages and frequencies.

## 1.5 Programming Model

Figure 1-3 shows the V4 programming model, which is organized as follows:

- User mode. User-mode software is restricted to user-mode instructions and registers.
- Supervisor mode. Supervisor-mode software can reference all user- and supervisor-mode instructions and registers.

The status register supervisor bit (SR[S]) selects the mode. Note that this figure shows optional V4 registers implemented in the CF4e.



**Figure 1-3. ColdFire Programming Model**

## 1.6 Address Map

Table 1-1 shows the ColdFire CPU space assignment reserved for program-visible registers. In general, these registers can be read and written through this space for debug accesses, initiated by an external emulator through the serial BDM communication channel. All control and configuration registers can be written to using the privileged move control register (MOVEC) instruction.

### NOTE:

A core may not implement all registers or register fields defined by the architecture, and it may implement additional registers or fields.

Table 1-1 lists register names, the CPU space assignment, whether the register is written from the processor using the MOVEC instruction, and the complete register name.

**Table 1-1. ColdFire CPU Space Assignments**

Name	CPU Space Assignment	Written with MOVEC	Register Name
<b>Memory Management Control Registers</b>			
CACR	0x002	Yes	Cache control register
ASID	0x003	Yes	Address space identifier
ACR0–ACR3	0x004–0x007	Yes	Access control registers [0:3]
MMUBAR	0x008	Yes	MMU base address register
<b>Processor General-Purpose Registers</b>			
D0–D7	0x(0,1)80–0x(0,1)87	No	Data registers 0–7 (0 = load, 1 = store)
A0–A7	0x(0,1)88–0x(0,1)8F	No	Address registers 0–7 (0 = load, 1 = store) A7 is user stack pointer
<b>Processor Miscellaneous Registers</b>			
OTHER_A7	0x800	No	Other stack pointer
VBR	0x801	No	Vector base register
MACSR	0x804	No	MAC status register
MASK	0x805	No	MAC address mask register
ACC	0x806	No	MAC accumulator
ACC0–ACC3	0x806–0x80B	No	MAC accumulators 0–3
ACCEXT01	0x807	No	MAC accumulator 0, 1 extension bytes
ACCEXT23	0x808	No	MAC accumulator 2, 3 extension bytes
SR	0x80E	No	Status register
PC	0x80F	No	Program counter
<b>Processor Floating-Point Registers</b>			
FPU0	0x810	No	32 msbs of floating-point data register 0

Table 1-1. ColdFire CPU Space Assignments

Name	CPU Space Assignment	Written with MOVEC	Register Name
FPL0	0x811	No	32 lsbs of floating-point data register 0
FPU1	0x812	No	32 msbs of floating-point data register 1
FPL1	0x813	No	32 lsbs of floating-point data register 1
FPU2	0x814	No	32 msb of floating-point data register 2
FPL2	0x815	No	32 lsbs of floating-point data register 2
FPU3	0x816	No	32 msbs of floating-point data register 3
FPL3	0x817	No	32 lsbs of floating-point data register 3
FPU4	0x818	No	32 msbs of floating-point data register 4
FPL4	0x819	No	32 lsbs of floating-point data register 4
FPU5	0x81A	No	32 msbs of floating-point data register 5
FPL5	0x81B	No	32 lsbs of floating-point data register 5
FPU6	0x81C	No	32 msbs of floating-point data register 6
FPL6	0x81D	No	32 lsbs of floating-point data register 6
FPU7	0x81E	No	32 msbs of floating-point data register 7
FPL7	0x81F	No	32 lsbs of floating-point data register 7
FPIAR	0x821	No	Floating-point instruction address register
FPSR	0x822	No	Floating-point status register
FPCR	0x824	No	Floating-point control register
<b>Local Memory and Module Control Registers</b>			
ROMBAR0	0xC00	Yes	ROM base address register 0
ROMBAR1	0xC01	Yes	ROM base address register 1
RAMBAR0	0xC04	Yes	RAM base address register 0
RAMBAR1	0xC05	Yes	RAM base address register 1
MPCR	0xC0C	Yes	Multiprocessor control register <sup>1</sup>
EDRAMBAR	0xC0D	Yes	Embedded DRAM base address register <sup>1</sup>
SECMBAR	0xC0E	Yes	Secondary module base address register <sup>1</sup>
MBAR	0xC0F	Yes	Primary module base address register
<b>Local Memory Address Permutation Control Registers <sup>1</sup></b>			
PCR1U0	0xD02	Yes	32 msbs of RAM 0 permutation control register 1
PCR1L0	0xD03	Yes	32 lsbs of RAM 0 permutation control register 1
PCR2U0	0xD04	Yes	32 msbs of RAM 0 permutation control register 2
PCR2L0	0xD05	Yes	32 lsbs of RAM 0 permutation control register 2
PCR3U0	0xD06	Yes	32 msbs of RAM 0 permutation control register 3
PCR3L0	0xD07	Yes	32 lsbs of RAM 0 permutation control register 3
PCR1U1	0xD0A	Yes	32 msbs of RAM 1 permutation control register 1

**Table 1-1. ColdFire CPU Space Assignments**

Name	CPU Space Assignment	Written with MOVEC	Register Name
PCR1L1	0xD0B	Yes	32 lsbs of RAM 1 permutation control register 1
PCR2U1	0xD0C	Yes	32 msbs of RAM 1 permutation control register 2
PCR2L1	0xD0D	Yes	32 lsbs of RAM 1 permutation control register 2
PCR3U1	0xD0E	Yes	32 msbs of RAM 1 permutation control register 3
PCR3L1	0xD0F	Yes	32 lsbs of RAM 1 permutation control register 3

<sup>1</sup> Field definitions for these optional registers are implementation-specific

## 1.7 Data Format Summary

Table 1-2 lists the operand data formats. Integer operands can reside in registers, memory, or instructions. The operand size is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

**Table 1-2. Integer Data Formats**

Operand Data Format	Size
Bit	1 bit
Byte integer	8 bits
Word integer	16 bits
Longword integer	32 bits

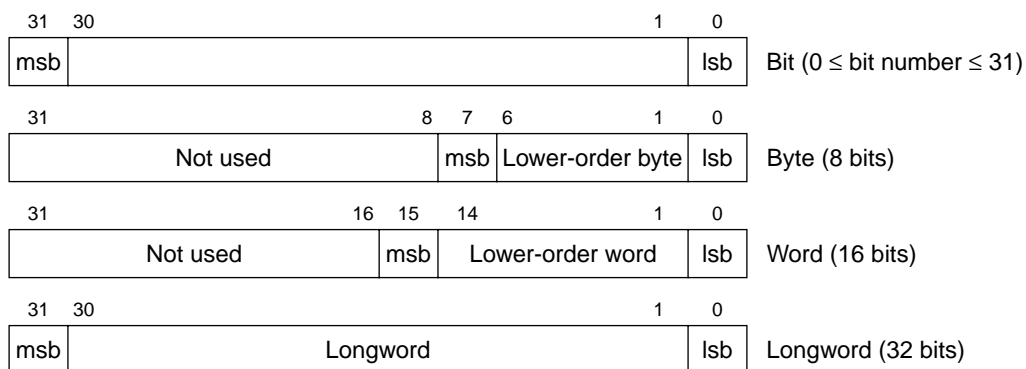
### 1.7.1 Data Organization in Registers

The following sections describe data organization in data, address, and control registers. Section 4.2.2, “Floating-Point Data Formats,” describes floating-point formatting.

#### 1.7.1.1 Integer Data Format Organization in Registers

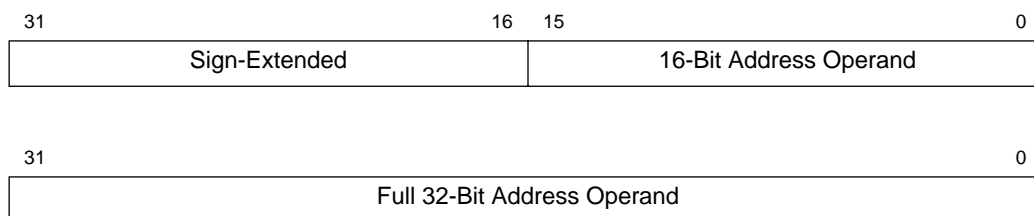
Figure 1-4 shows the integer format for data registers. Each integer data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Longword operands occupy the entire 32 bits of integer data registers. A data register that is either a source or destination operand only uses or changes the appropriate lower 8 or 16 bits in byte or word operations, respectively. The remaining high-order portion does not change. Note that the least-significant bit is bit 0 for all data types, whereas the msbs for longword integer is bit 31, the msb of a word integer is bit 15, and the msb of a byte integer is bit 7.

## Data Format Summary



**Figure 1-4. Organization of Integer Data Format in Data Registers**

Instruction encodings disallow use of address registers for byte operands. When an address register is a source operand, either the low-order word or the entire longword operand is used, depending on the operation size. Word-length source operands are sign-extended to 32 bits and then used in the operation with an address register destination. When an address register is a destination, the entire register is affected, regardless of the operation size. Figure 1-5 shows integer formats for address registers.



**Figure 1-5. Organization of Integer Data Formats in Address Registers**

The size of control registers varies according to function. Some have undefined bits reserved for future definition by Motorola. Those bits read as zeros and must be written as zeros for future compatibility. Operations to the SR and CCR are word-sized. The upper CCR byte is read as all zeros and is ignored when written, regardless of privilege mode.

### 1.7.1.2 Integer Data Format Organization in Memory

ColdFire processors use big-endian addressing. Byte-addressable memory organization allows lower addresses to correspond to higher-order bytes. The address N of a longword data item corresponds to the address of the high-order word. The lower-order word is at address N + 2. The address of a word data item corresponds to the address of the high-order byte. The lower-order byte is at address N + 1. This organization is shown in Figure 1-6.

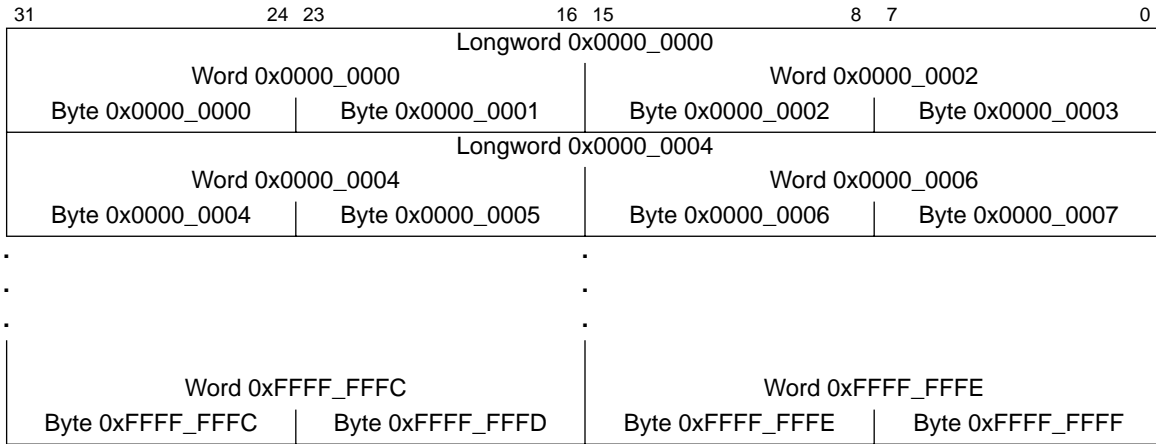


Figure 1-6. Memory Operand Addressing

## 1.7.2 EMAC Data Representation

The EMAC supports the following three modes, where each mode defines a unique operand type.

- Two's complement signed integer: In this format, an N-bit operand value lies in the range  $-2^{(N-1)} \leq \text{operand} \leq 2^{(N-1)} - 1$ . The binary point is right of the lsb.
- Unsigned integer: In this format, an N-bit operand value lies in the range  $0 \leq \text{operand} \leq 2^N - 1$ . The binary point is right of the lsb.
- Two's complement, signed fractional: In an N-bit number, the first bit is the sign bit. The remaining bits signify the first N-1 bits after the binary point. Given an N-bit number,  $a_{N-1}a_{N-2}a_{N-3}... a_2a_1a_0$ , its value is given by the equation in Figure 1-7.

$$\text{value} = -(1 \cdot a_{N-1}) + \sum_{i=0}^{N-2} 2^{(i+1-N)} \cdot a_i$$

Figure 1-7. Two's Complement, Signed Fractional Equation

This format can represent numbers in the range  $-1 \leq \text{operand} \leq 1 - 2^{(N-1)}$ .

For words and longwords, the largest negative number that can be represented is -1, whose internal representation is 0x8000 and 0x8000\_0000, respectively. The largest positive word is 0x7FFF or  $(1 - 2^{-15})$ ; the most positive longword is 0x7FFF\_FFFF or  $(1 - 2^{-31})$ .

For more information, see Chapter 5, "Enhanced Multiply-Accumulate Unit (EMAC)."

### 1.7.2.1 Floating-Point Data Formats and Types

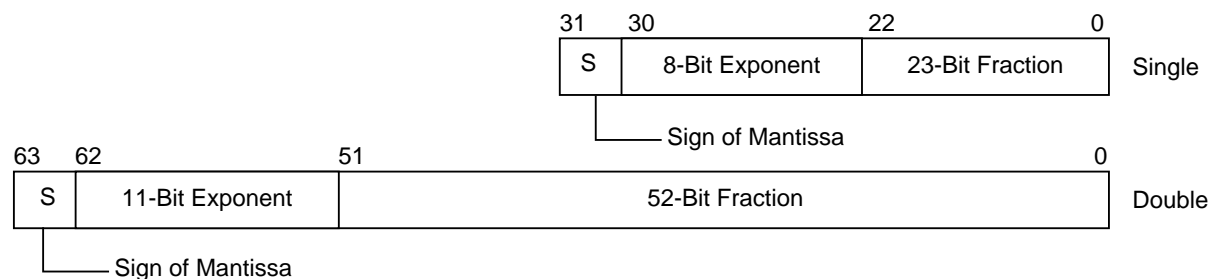
The FPU supports signed byte, word, and longword integer formats, which are identical to those supported by the integer unit. The FPU also supports single- and double-precision binary floating-point formats that fully comply with the IEEE-754 standard.

### 1.7.2.1.1 Signed-Integer Data Formats

The FPU supports 8-bit byte (B), 16-bit word (W), and 32-bit longword (L) integer data formats.

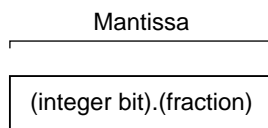
### 1.7.2.1.2 Floating-Point Data Formats

Figure 1-8 shows the two binary floating-point data formats.



**Figure 1-8. Floating-Point Data Formats**

Note that, throughout this chapter, a mantissa is defined as the concatenation of an integer bit, the binary point, and a fraction. A fraction is the term designating the bits to the right of the binary point in the mantissa.



**Figure 1-9. Mantissa**

The integer bit is implied to be set for normalized numbers and infinities, clear for zeros and denormalized numbers. For not-a-numbers (NaNs), the integer bit is ignored. The exponent in both floating-point formats is an unsigned binary integer with an implied bias added to it. Subtracting the bias from exponent yields a signed, two's complement power of two. This represents the magnitude of a normalized floating-point number when multiplied by the mantissa.

By definition, a normalized mantissa always takes values starting from 1.0 and going up to, but not including, 2.0; that is, [1.0...2.0).

## 1.8 Addressing Modes

Addressing modes are categorized by how they are used. Data addressing modes refer to data operands. Memory addressing modes refer to memory operands. Alterable addressing modes refer to alterable (writable) data operands. Control addressing modes refer to memory operands without an associated size.

These categories sometimes combine to form more restrictive categories. Two combined classifications are alterable memory (both alterable and memory) and data alterable (both



alterable and data). ColdFire microprocessors support 12 of the most commonly used M68000 Family effective addressing modes. Table 1-3 summarizes these modes.

**Table 1-3. ColdFire Effective Addressing Modes**

Addressing Modes	Syntax	Mode Field	Register Field	Category			
				Data	Memory	Control	Alterable
Absolute data addressing Short Long	(xxx).W (xxx).L	111 111	000 001	X X	X X	X X	— —
Address register indirect with scaled index 8-bit displacement	(d <sub>8</sub> , An, Xi*SF)	110	register no.	X	X	X	X
Immediate	#<xxx>	111	100	X	X	—	—
Program counter indirect with displacement	(d <sub>16</sub> , PC)	111	010	X	X	X	—
Program counter indirect with scaled index 8-bit displacement	(d <sub>8</sub> , PC, Xi*SF)	111	011	X	X	X	—
Register direct Data Address	Dn An	000 001	register no. register no.	X —	— —	— —	X X
Register indirect Address Address with Postincrement Address with Predecrement Address with Displacement	(An) (An)+ -(An) (d <sub>16</sub> , An)	010 011 100 101	register no. register no. register no. register no.	X X X X	X X X X	X — — X	X X X X

## 1.9 Instruction Set Overview

The original ColdFire ISA was derived from M68000 Family opcodes based on extensive analysis of embedded application code. After the first ColdFire compilers were created, developers identified ISA additions that would enhance both code density and overall performance. Additionally, as users implemented ColdFire-based designs into a wide range of embedded systems, they identified frequently used instruction sequences that could be improved by creating new instructions. This observation was especially prevalent in environments that used substantial amounts of assembly language code.

The original ISA minimized support for instructions referencing byte and word operands. MOVE.B and MOVE.W were fully supported; otherwise, only CLR (clear) and TST (test) supported these data types. Based on input from compiler writers and system users, a set of instruction enhancements was proposed to address the following:

- Enhanced support for byte and word-sized operands through new move operations
- Enhanced support for position-independent code

For descriptions of the ColdFire instruction set, see the latest version of the *ColdFire Programmer's Reference Manual*.

The following list summarizes new and enhanced instructions of ISA\_B:

- New instructions:
  - INTOUCH loads blocks of instructions to be locked in the instruction cache.
  - MOV3Q.L moves 3-bit immediate data to the destination location.
  - MOVE to/from USP loads and stores user stack pointer.
  - MVS.{B,W} sign-extends the source operand and moves it to the destination register.
  - MVZ.{B,W} zero-fills the source operand and moves it to the destination register.
  - SATS.L performs a saturation operation for signed arithmetic and updates the destination register depending on CCR[V] and bit 31 of the register.
  - TAS.B performs an indivisible read-modify-write cycle to test and set the addressed memory byte.
- Enhancements to existing Revision\_A instructions:
  - Longword support for branch instructions (Bcc, BRA, BSR)
  - Byte and word support for compare instructions (CMP, CMPI)
  - Word support for the compare address register instruction (CMPA)
  - Byte and longword support for MOVE.x, where the source is immediate data and the destination is specified by d16(Ax); that is, MOVE.{B,W} #<data>, d16(Ax)
- Floating-point instructions. See Chapter 4, “Floating-Point Unit (FPU).”
- EMAC instructions. See Chapter 5, “Enhanced Multiply-Accumulate Unit (EMAC).”

Table 1-4 shows the syntax for the new and enhanced instructions. As Table 1-4 shows, some ISA\_B opcodes were defined in the M68K family and others are new.

**Table 1-4. V4 New Instruction Summary**

Instruction	Mnemonic <sup>1</sup>	Source	Destination	68K
<b>ISA_B Extensions</b>				
Branch Always	bra.l		<label>	Yes
Branch Conditionally	bcc.l		<label>	Yes
Branch to Subroutine	bsr.l		<label>	Yes
Compare	cmp.{b,w,l}	<ea>y	Dx	Yes
Compare Address	cmpa.w	<ea>y	Ax	Yes
Compare Immediate	cmpi.{b,w}	#<data>	Dx	Yes
Instruction Fetch Touch	intouch	<Ay>		
Move 3-Bit Data Quick	mov3q.l	#<data>	<ea>x	
Move Data Source to Destination	move.{b,w}	#<data>	d16(Ax)	Yes
Move from USP	move.l	USP	Ax	Yes
Move to USP	move.l	Ay	USP	Yes

Table 1-4. V4 New Instruction Summary (Continued)

Instruction	Mnemonic <sup>1</sup>	Source	Destination	68K
Move with Sign Extend	mvs.{b,w}	<ea>y	Dx	
Move with Zero-Fill	mvz.{b,w}	<ea>y	Dx	
Signed Saturate	sats.l		Dx	
Test and Set an Operand	tas.b		<ea>x	Yes
<b>EMAC Extensions</b>				
Move from an Accumulator and Clear	movclr.l	ACCx	Rx	No
Copy an Accumulator	move.l	ACCy	ACCx	No
Move from Accumulator 0 and 1 Extensions	move.l	ACCext01	Rx	No
Move from Accumulator 2 and 3 Extensions	move.l	ACCext23	Rx	No
Move to Accumulator 0 and 1 Extensions	move.l	Ry	ACCext01	No
Move to Accumulator 2 and 2 Extensions	move.l	Ry	ACCext23	No
<b>FPU Instructions</b>				
Floating-Point Absolute Value	fabs.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Add	fadd.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Branch Conditionally	fbcc.{w,l}		<label>	Yes
Floating-Point Compare	fcmp.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Divide	fdiv.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Integer	fint.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Integer Round-to-Zero	fintrz.{b,w,l,s,d}	<ea>y	FPx	Yes
Move Floating-Point Data Register	fmove.{b,w,l,s,d}	<ea>y	FPx	Yes
Move from FPCR	fmove.l	FPCR	<ea>x	Yes
Move from FPIAR	fmove.l	FPIAR	<ea>x	Yes
Move from FPSR	fmove.l	FPSR	<ea>x	Yes
Move from FPCR	fmove.l	<ea>y	FPCR	Yes
Move from FPIAR	fmove.l	<ea>y	FPIAR	Yes
Move from FPSR	fmove.l	<ea>y	FPSR	Yes
Move Multiple Floating Point Data Registers	fmovem.d	#list <ea>y	<ea>x #list	Yes
Floating-Point Multiply	fmul.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Negate	fneg.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point No Operation	fnop			Yes
Restore Internal Floating Point State	frestore	<ea>y		Yes
Save Internal Floating Point State	fsave		<ea>x	Yes
Floating-Point Square Root	fsqrt.{b,w,l,s,d}	<ea>y	FPx	Yes
Floating-Point Subtract	fsub.{b,w,l,s,d}	<ea>y	FPx	Yes
Test Floating-Point Operand	ftst.{b,w,l,s,d}	<ea>y		Yes

<sup>1</sup> Operand sizes in this column reflect only newly supported operand sizes for existing instructions (Bcc, BRA, BSR, CMP, CMPA, CMPI, and MOVE)

## 1.9.1 Instruction Set Summary

Table 1-5 lists user-mode instructions by opcode.

**Table 1-5. User-Mode Instruction Set Summary**

Instruction	Operand Syntax	Operand Size	Operation
ADD	Dy,<ea>x	L	Source + Destination → Destination
ADDA	<ea>y,Dx	L	
	<ea>y,Ax	L	
ADDI	#<data>,Dx	L	Immediate Data + Destination → Destination
ADDQ	#<data>,<ea>x	L	
ADDX	Dy,Dx	L	Source + Destination + CCR[X] → Destination
AND	<ea>y,Dx	L	Source & Destination → Destination
	Dy,<ea>x	L	
ANDI	#<data>, Dx	L	Immediate Data & Destination → Destination
ASL	Dy,Dx	L	CCR[X,C] ← (Dx << Dy) ← 0
	#<data>,Dx	L	
ASR	Dy,Dx	L	msb → (Dx >> Dy) → CCR[X,C]
	#<data>,Dx	L	
Bcc	<label>	B, W, L	If Condition True, Then PC + d <sub>n</sub> → PC
BCHG	Dy,<ea>x	B, L	~ (<bit number> of Destination) → CCR[Z] →
	#<data>,<ea>x	B, L	
BCLR	Dy,<ea>x	B, L	~ (<bit number> of Destination) → CCR[Z];
	#<data>,<ea>x	B, L	
BRA	<label>	B, W, L	PC + d <sub>n</sub> → PC
BSET	Dy,<ea>x	B, L	~ (<bit number> of Destination) → CCR[Z];
	#<data>,<ea>x	B, L	
BSR	<label>	B, W, L	SP – 4 → SP; nextPC → (SP); PC + d <sub>n</sub> → PC
BTST	Dy,<ea>x	B, L	~ (<bit number> of Destination) → CCR[Z]
	#<data>,<ea>x	B, L	
CLR	<ea>x	B, W, L	0 → Destination
CMP	<ea>y,Dx	B, W, L	Destination – Source → CCR
CMPA	<ea>y,Ax	W, L	
CMPI	#<data>,Dx	B, W, L	Destination – Immediate Data → CCR
DIVS/DIVU	<ea>y,Dx	W, L	Destination / Source → Destination (Signed or Unsigned)
EOR	Dy,<ea>x	L	Source ^ Destination → Destination
EORI	#<data>,Dx	L	Immediate Data ^ Destination → Destination
EXT	Dx	B → W	Sign-Extended Destination → Destination
	Dx	W → L	
EXTB	Dx	B → L	

Table 1-5. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
FABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx Absolute Value of FPx → FPx
FADD	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source + FPx → FPx
FBcc	<label>	W, L	If Condition True, Then PC + d <sub>n</sub> → PC
FCMP	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source
FDABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx; round destination to double Absolute Value of FPx → FPx; round destination to double
FDADD	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source + FPx → FPx; round destination to double
FDDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx; round destination to double
FDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx
FDMOVE	FPy,FPx	D	Source → Destination; round destination to double
FDMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx; round destination to double
FDNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx; round destination to double - (FPx) → FPx; round destination to double
FDSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx; round destination to double Square Root of FPx → FPx; round destination to double
FDSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx; round destination to double
FINT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Integer Part of Source → FPx Integer Part of FPx → FPx
FINTRZ	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Integer Part of Source → FPx; round to zero Integer Part of FPx → FPx; round to zero
FMOVE	<ea>y,FPx FPy,<ea>x FPy,FPx FPcr,<ea>x <ea>y,FPcr	B,W,L,S,D B,W,L,S,D D L L	Source → Destination  FPcr can be any floating point control register: FPCR, FPIAR, FPSR
FMOVEM	#list,<ea>x <ea>y,#list	D	Listed registers → Destination Source → Listed registers
FMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx

Table 1-5. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
FNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx - (FPx) → FPx
FNOP	none	none	PC + 2 → PC (FPU Pipeline Synchronized)
FSABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx; round destination to single Absolute Value of FPx → FPx; round destination to single
FSADD	<ea>y,FPx FPy,FPx	B,W,L,S,D	Source + FPx → FPx; round destination to single
FSDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx; round destination to single
FSMOVE	<ea>y,FPx	B,W,L,S,D	Source → Destination; round destination to single
FSMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx; round destination to single
FSNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx; round destination to single - (FPx) → FPx; round destination to single
FSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx Square Root of FPx → FPx
FSSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx; round destination to single Square Root of FPx → FPx; round destination to single
FSSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx; round destination to single
FSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx
FTST	<ea>y	B, W, L, S, D	Source Operand Tested → FPCC
ILLEGAL	none	none	SP - 4 → SP; PC → (SP) → PC; SP - 2 → SP; SR → (SP); SP - 2 → SP; Vector Offset → (SP); (VBR + 0x10) → PC
JMP	<ea>y	none	Source Address → PC
JSR	<ea>y	none	SP - 4 → SP; nextPC → (SP); Source → PC
LEA	<ea>y,Ax	L	<ea>y → Ax
LINK	Ay,#<displacement>	W	SP - 4 → SP; Ay → (SP); SP → Ay, SP + d <sub>n</sub> → SP
LSL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0
LSR	Dy,Dx #<data>,Dx	L L	0 → (Dx >> Dy) → CCR[X,C] 0 → (Dx >> #<data>) → CCR[X,C]
MAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,A CCx	W, L W, L	ACCx + (Ry * Rx){<< >>}SF → ACCx ACCx + (Ry * Rx){<< >>}SF → ACCx; (<ea>y(&MASK)) → Rw

Table 1-5. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
MOV3Q	#<data>,<ea>x	L	Immediate Data → Destination
MOVCLR	ACCy,Rx	L	Accumulator → Destination, 0 → Accumulator
MOVE  MOVE from CCR MOVE to CCR	<ea>y,<ea>x MACcr,Dx <ea>y,MACcr CCR,Dx <ea>y,CCR	B,W,L L L W W	Source → Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK
MOVEA	<ea>y,Ax	W,L → L	Source → Destination
MOVEM	#list,<ea>x <ea>y,#list	L	Listed Registers → Destination Source → Listed Registers
MOVEQ	#<data>,Dx	B → L	Immediate Data → Destination
MSAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,ACCx	W, L W, L	ACCx - (Ry * Rx){<< >>}SF → ACCx ACCx - (Ry * Rx){<< >>}SF → ACCx; (<ea>y(&MASK)) → Rw
MULS/MULU	<ea>y,Dx	W * W → L L * L → L	Source * Destination → Destination (Signed or Unsigned)
MVS	<ea>y,Dx	B,W	Source with sign extension → Destination
MVZ	<ea>y,Dx	B,W	Source with zero fill → Destination
NEG	Dx	L	0 – Destination → Destination
NEGX	Dx	L	0 – Destination – CCR[X] → Destination
NOP	none	none	PC + 2 → PC (Integer Pipeline Synchronized)
NOT	Dx	L	~ Destination → Destination
OR	<ea>y,Dx Dy,<ea>x	L L	Source   Destination → Destination
ORI	#<data>,Dx	L	Immediate Data   Destination → Destination
PEA	<ea>y	L	SP – 4 → SP; <ea>y → (SP)
PULSE	none	none	Set PST = 0x4
REMS/REMU	<ea>y,Dw:Dx	L	Destination / Source → Remainder (Signed or Unsigned)
RTS	none	none	(SP) → PC; SP + 4 → SP
SATS	Dx	L	If CCR[V] == 1; then if Dx[31] == 0; then Dx[31:0] = 0x80000000; else Dx[31:0] = 0x7FFFFFFF; else Dx[31:0] is unchanged
Scc	Dx	B	If Condition True, Then 1s → Destination; Else 0s → Destination
SUB	<ea>y,Dx Dy,<ea>x	L L	Destination - Source → Destination
SUBA	<ea>y,Ax	L	

**Table 1-5. User-Mode Instruction Set Summary (Continued)**

Instruction	Operand Syntax	Operand Size	Operation
SUBI SUBQ	#<data>,Dx #<data>,<ea>x	L L	Destination – Immediate Data → Destination
SUBX	Dy,Dx	L	Destination – Source – CCR[X] → Destination
SWAP	Dx	W	MSW of Dx ↔ LSW of Dx
TAS	<ea>x	B	Destination Tested → CCR; 1 → bit 7 of Destination
TPF	none #<data> #<data>	none W L	PC + 2 → PC PC + 4 → PC PC + 6 → PC
TRAP	#<vector>	none	1 → S Bit of SR; SP – 4 → SP; nextPC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format/Offset → (SP) (VBR + 0x80 + 4*n) → PC, where n is the TRAP number
TST	<ea>y	B, W, L	Source Operand Tested → CCR
UNLK	Ax	none	Ax → SP; (SP) → Ax; SP + 4 → SP
WDDATA	<ea>y	B, W, L	Source → DDATA port

Table 1-6 describes supervisor-mode instructions.

**Table 1-6. Supervisor-Mode Instruction Set Summary**

Instruction	Operand Syntax	Operand Size	Operation
CPUSHL	ic,(Ax) dc,(Ax) bc,(Ax)	none	If data is valid and modified, push cache line; invalidate line if programmed in CACR (synchronizes pipeline)
FRESTORE	<ea>y	none	FPU State Frame → Internal FPU State
FSAVE	<ea>x	none	Internal FPU State → FPU State Frame
HALT	none	none	Halt processor core
INTOUCH	Ay	none	Instruction fetch touch at (Ay)
MOVE from SR	SR,Dx	W	SR → Destination
MOVE from USP	USP,Dx	L	USP → Destination
MOVE to SR	<ea>y,SR	W	Source → SR; Dy or #<data> source only
MOVE to USP	Ay,USP	L	Source → USP
MOVEC	Ry,Rc	L	Ry → Rc
RTE	none	none	2 (SP) → SR; 4 (SP) → PC; SP + 8 → SP Adjust stack according to format
STOP	#<data>	none	Immediate Data → SR; STOP
WDEBUG	<ea>y	L	Addressed Debug WDMREG Command Executed



# Chapter 2

## Registers

This chapter describes the organization of CF4e general-purpose and control registers in the user and supervisor programming models.

### 2.1 Overview

Figure 2-2 shows both the user and supervisor register sets, which are described in this chapter.

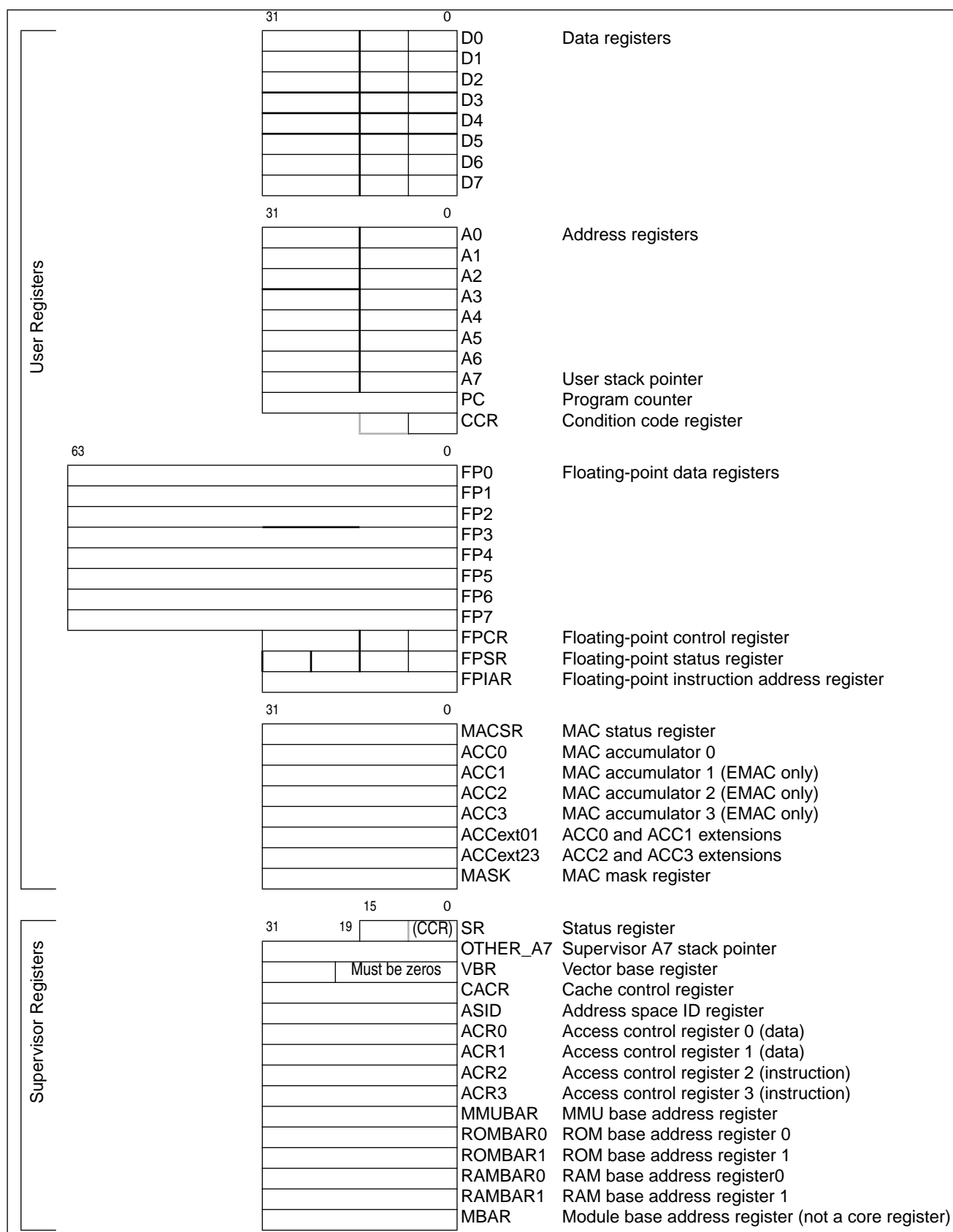


Figure 2-1. Programming Model

## 2.2 User Programming Model

The user programming model, shown in Figure 2-2, consists of the following registers:

- 16 general-purpose 32-bit registers (D7–D0 and A7–A0); A7 is user stack pointer
- 32-bit program counter
- 8-bit condition code register
- Registers to support the EMAC
- Register to support the floating-point unit (FPU)

Section 1.7, “Data Format Summary,” describes formats for integer and floating-point data.

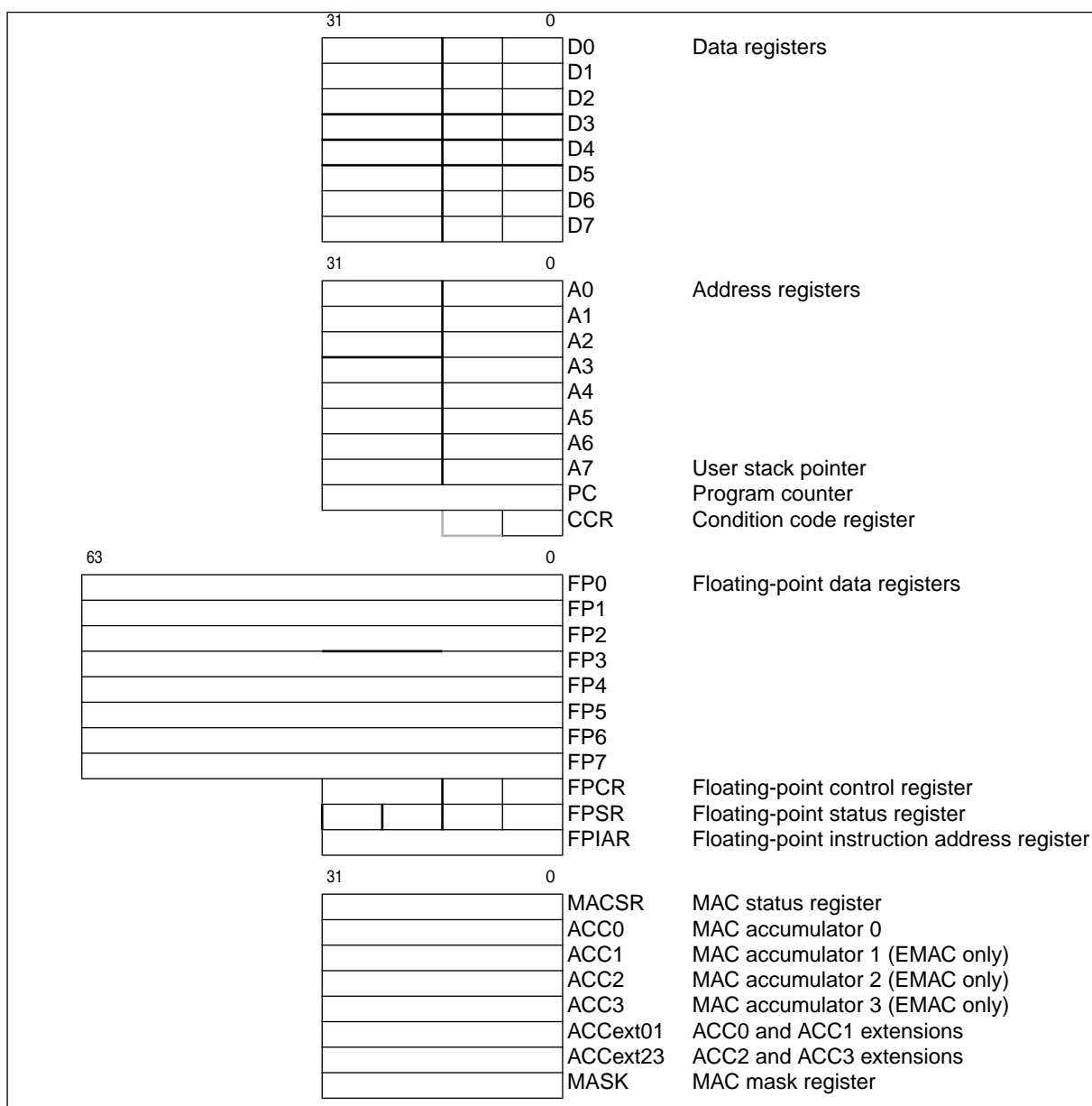


Figure 2-2. User Programming Model

### 2.2.1 Data Registers (D7–D0)

D7–D0 are used as data registers for bit, byte (8-bit), word (16-bit), and longword (32-bit) operations. They can also be used as index registers.

### 2.2.2 Address Registers (A6–A0)

A6–A0 can be used as software stack pointers, index registers, or base address registers and can be used for word and longword operations.

## 2.2.3 User Stack Pointer (A7)

The CF4e architecture supports two unique stack pointer (A7) registers—the supervisor stack pointer (SSP) and the user stack pointer (USP). This support provides the required isolation between operating modes as dictated by the virtual memory management scheme provided by the memory management unit (MMU). The SSP is described in Section 2.3.3, “Supervisor/User Stack Pointers (A7 and OTHER\_A7).”

## 2.2.4 Program Counter (PC)

The PC holds the address of the executing instruction. For sequential instructions, the processor automatically increments PC. When program flow changes, the PC is updated with the target instruction. For some instructions, the PC specifies the base address for PC-relative operand addressing modes. If two 16-bit instructions are dispatched together, the PC is advanced by 4 bytes, so that it points to the next instruction after this pair.

## 2.2.5 Condition Code Register (CCR)

The CCR occupies SR[7–0], as shown in Figure 2-3. CCR[4–0] are indicator flags based on results generated by arithmetic operations.

	7	5	4	3	2	1	0
Field	—		X	N	Z	V	C
Reset	000		Undefined				
R/W	R		R/W	R/W	R/W	R/W	R/W

**Figure 2-3. Condition Code Register (CCR)**

CCR fields are described in Table 2-1.

**Table 2-1. CCR Field Descriptions**

Bits	Name	Description
7–5	—	Reserved. These bits are read as 0; writes have no effect.
4	X	Extend condition code bit. Assigned the value of the carry bit for arithmetic operations; otherwise not affected or set to a specified result. Also used as an input operand for multiple-precision arithmetic.
3	N	Negative condition code bit. Set if the msb of the result is set; otherwise cleared.
2	Z	Zero condition code bit. Set if the result equals zero; otherwise cleared.
1	V	Overflow condition code bit. Set if an arithmetic overflow occurs, implying that the result cannot be represented in the operand size; otherwise cleared.
0	C	Carry condition code bit. Set if a carry-out of the data operand msb occurs for an addition or if a borrow occurs in a subtraction; otherwise cleared.

## 2.2.6 EMAC Programming Model

The registers in the EMAC portion of the user programming model, described in Section Chapter 5, “Enhanced Multiply-Accumulate Unit (EMAC),” include the following registers:

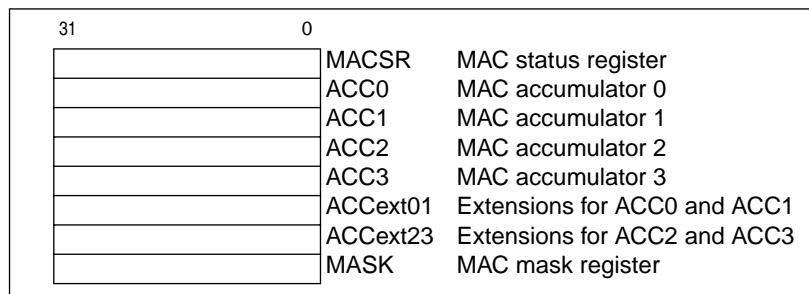
The EMAC provides the following program-visible registers:

- The EMAC programming model includes four 48-bit accumulator registers partitioned as follows:
  - Four 32-bit accumulators (ACC0–ACC3)
  - Eight 8-bit accumulator extension bytes (two per accumulator). These are grouped into two 32-bit values for load and store operations (ACCEXT01 and ACCEXT23).

Accumulators and extension bytes can be loaded, copied, and stored, and results from EMAC arithmetic operations generally affect the entire 48-bit destination.

- Eight 8-bit accumulator extensions (two per accumulator), packaged as two 32-bit values for load and store operations (ACCext01 and ACCext23)
- One 16-bit mask register (MASK)
- One 32-bit status register (MACSR) including four indicator bits signaling product or accumulation overflow (one for each accumulator: PAV0–PAV3)

These registers are shown in Figure 2-4.



**Figure 2-4. EMAC Register Set**

## 2.2.7 Floating-Point Programming Model

Figure 2-5 shows the FPU programming model.

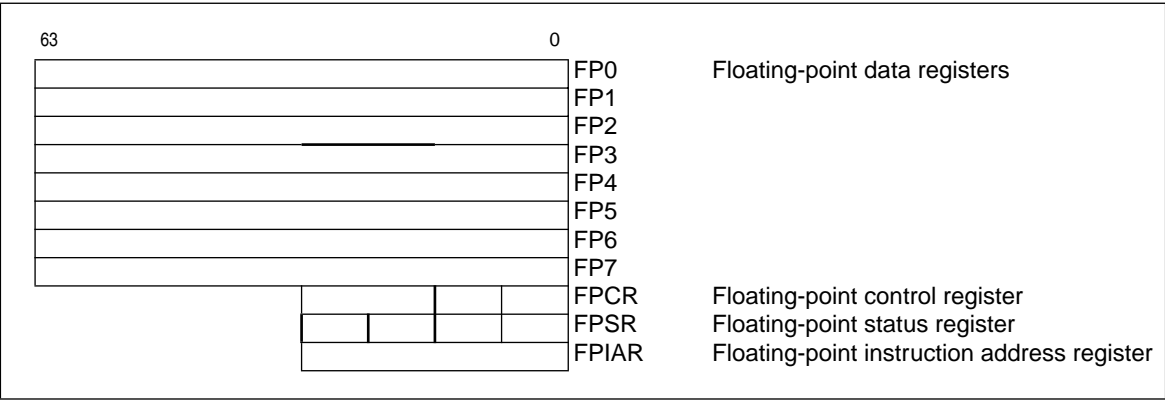


Figure 2-5. Floating-Point Programmer's Model

The programmer's model for the FPU consists of the following:

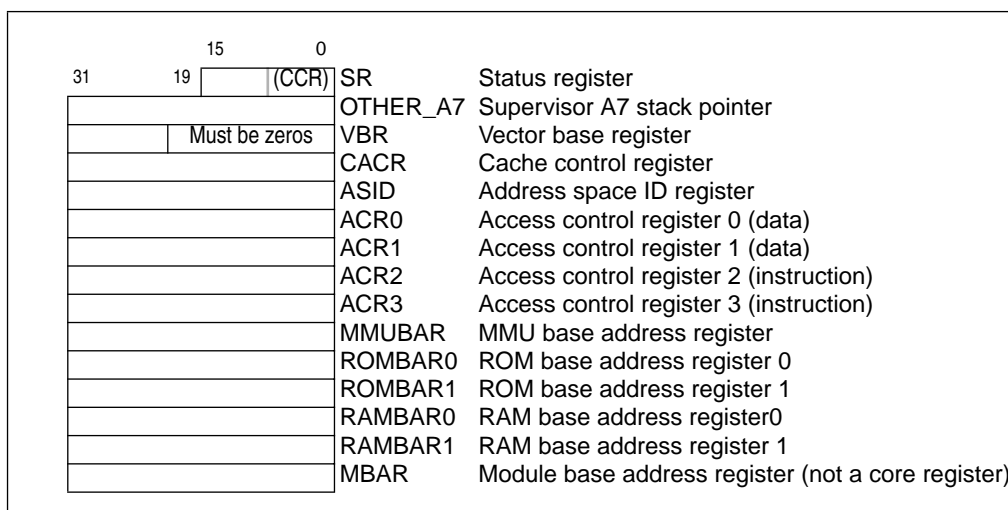
- Eight 64-bit floating-point data registers (FP0–FP7)
- One 32-bit floating-point control register (FPCR)
- One 32-bit floating-point status register (FPSR)
- One 32-bit floating-point instruction address register (FPIAR)

These registers are described in Section 4.3, “FPU Programmer's Model.”

## 2.3 Supervisor Programming Model

Typically, system programmers use the supervisor programming model to implement operating system functions and provide memory and I/O control. The CF4e supervisor programming model provides access to user registers and additional supervisor registers, which include the upper byte of the status register (SR), the supervisor stack pointer (SSP), the vector base register (VBR), and registers for configuring attributes of the address space connected to the processor core. Most supervisor-level registers are accessed by using the MOVEC instruction with the control register definitions in Table 2-4.

Figure 2-6 shows the supervisor programming model.



**Figure 2-6. Supervisor Programming Model**

### 2.3.1 Status Register (SR)

The SR stores the processor status, the interrupt priority mask, and other control bits. Supervisor software can read or write the entire SR; user software can read or write only SR[7–0], described in Section 2.2.5, “Condition Code Register (CCR).” Bits in the system byte indicate processor states—trace mode (T), supervisor or user mode (S), and master or interrupt state (M). SR is set to 0x27xx after reset.

	15	14	13	12	12	10	8	7	5	4	3	2	1	0
	System byte						Condition code register (CCR)							
Field	T	—	S	M	—	I	—	X	N	Z	V	C		
Reset	0	0	1	0	0	111	000	—	—	—	—	—		
R/W	R/W	R	R/W	R/W	R	R/W	R	R/W	R/W	R/W	R/W	R/W		

**Figure 2-7. Status Register (SR)**

Table 2-2 describes SR fields.

**Table 2-2. Status Field Descriptions**

Bits	Name	Description
15	T	Trace enable. When T is set, the processor performs a trace exception after every instruction.
13	S	Supervisor/user state. Indicates whether the processor is in supervisor or user mode 0 User mode 1 Supervisor mode
12	M	Master/interrupt state. Cleared by an interrupt exception. It can be set by software during execution of the RTE or move to SR instructions so the OS can emulate an interrupt stack pointer.



**Table 2-2. Status Field Descriptions (Continued)**

Bits	Name	Description
10–8	I	Interrupt priority mask. Defines the current interrupt priority. Interrupt requests are inhibited for all priority levels less than or equal to the current priority, except the edge-sensitive level-7 request, which cannot be masked.
7–0	CCR	Condition code register. See Table 2-3.

### 2.3.2 Vector Base Register (VBR)

The VBR holds the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table. VBR[19–0] are not implemented and are assumed to be zero, forcing the vector table to be aligned on a 0-modulo-1-Mbyte boundary.

	31	20	19	0
Field	Exception vector table base address			—
Reset	All zeros			
R/W	Written from a BDM serial command or from the CPU using the MOVEC instruction. VBR can be read from the debug module only. The upper 12 bits are returned, the low-order 20 bits are undefined.			
Rc	0x801			

**Figure 2-8. Vector Base Register (VBR)**

### 2.3.3 Supervisor/User Stack Pointers (A7 and OTHER\_A7)

The CF4e architecture supports two independent stack pointer (A7) registers—the supervisor stack pointer (SSP) and the user stack pointer (USP). This support provides the required isolation between operating modes as dictated by the virtual memory management scheme provided by the memory management unit (MMU).

The hardware implementation of these two programmable-visible 32-bit registers does not uniquely identify one as the SSP and the other as the USP. Rather, the hardware uses one 32-bit register as the currently-active A7 and the other as OTHER\_A7. Thus, the register contents are a function of the processor operating mode, as shown in the following:

```

if SR[S] = 1
    then
        A7 = Supervisor Stack Pointer
        other_A7 = User Stack Pointer
    else
        A7 = User Stack Pointer
        other_A7 = Supervisor Stack Pointer

```

The BDM programming model supports reads and writes to A7 and OTHER\_A7 directly. It is the responsibility of the external development system to determine the mapping of (A7 and OTHER\_A7) to the two program-visible definitions (SSP and USP), based on the setting of SR[S]. This functionality is enabled by setting by the dual stack pointer enable bit CACR[DSPE]. If this bit is cleared, only the stack pointer, A7, defined for previous ColdFire versions is available. DSPE is zero at reset.

If DSPE is set, the appropriate stack pointer register (SSP or USP) is accessed as a function of the processor's operating mode. To support dual stack pointers, the following two privileged MC680x0 instructions to load/store the USP are added to the ColdFire instruction set architecture:

```
move.l Ay,USP # move to USP
```

```
move.l USP,Ax # move from USP
```

These instructions are described in the *PRM*.

### 2.3.4 Cache Control Register (CACR)

The CACR controls operation of the instruction, data, and branch cache memories. It includes bits for enabling, freezing, and invalidating cache contents. It also includes bits for defining the default cache mode and write-protect fields. The CACR is described in Section 8.7.10.1, “Cache Control Register (CACR).”

### 2.3.5 Access Control Registers (ACR0–ACR3)

The access control registers, ACR0–ACR3 define attributes for four user-defined memory regions. ACR0 and ACR1 control data memory space and ACR2 and ACR3 control instruction memory space. Attributes include definition of cache mode, write protect and buffer write enables. The ACRs are described in Section 8.7.10.2, “Access Control Registers (ACR0–ACR3).”

### 2.3.6 RAM Base Address Registers (RAMBAR0/RAMBAR1)

RAMBAR registers are used to specify the base address of the internal RAM modules and indicate the types of references mapped to each. Each RAMBAR includes a base address, write-protect bit, address space mask bits, and an enable bit. RAM base address alignment is implementation specific. See Section 8.5.2.1, “SRAM Base Address Registers (RAMBAR0/RAMBAR1).”

### 2.3.7 ROM Base Address Registers (ROMBAR0/ROMBAR1)

ROMBAR registers determine the base address of the internal ROM modules and indicate the types of references mapped to each. Each ROMBAR includes a base address, write-protect bit, address space mask bits, and an enable bit. ROM base address alignment is implementation specific. See Section 8.6.2.1, “ROM Base Address Registers (ROMBAR0/ROMBAR1).”

### 2.3.8 Module Base Address Register (MBAR)

The supervisor-level MBAR, Figure 2-9, specifies the base address and allowable access types for all internal peripherals. Note that the MBAR is not implemented in the core, but

it is included here because it must be implemented by the system designer. It is written with a MOVEC instruction using the CPU address 0xC0F. (See the *ColdFire Family Programmer's Reference Manual*.) MBAR can be read or written through the debug module as a read/write register, as described in Chapter 11, “Debug Support.” Only the debug module can read MBAR.

The valid bit, MBAR[V], is cleared at system reset to prevent incorrect references before MBAR is written; other MBAR bits are uninitialized at reset. To access internal peripherals, write MBAR with the appropriate base address (BA) and set MBAR[V] after system reset.

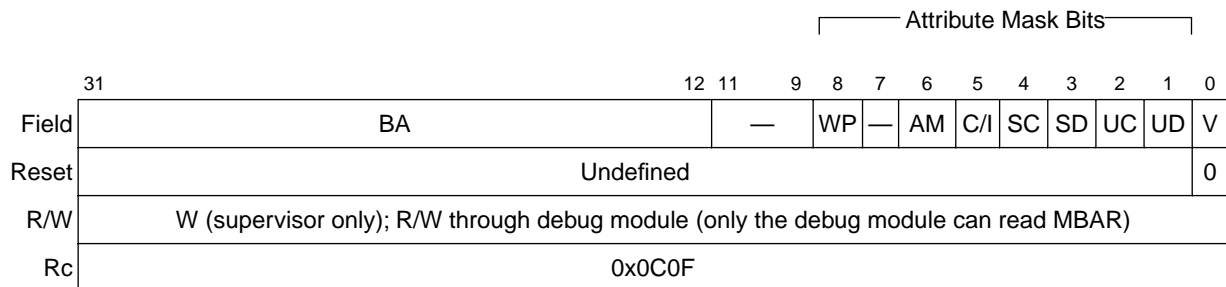
All internal peripheral registers occupy a single relocatable memory block along 4-Kbyte boundaries. If MBAR[V] is set, MBAR[BA] is compared to the upper 20 bits of the full 32-bit internal address to determine if an internal peripheral is being accessed. MBAR masks specific address spaces using the address space fields. Attempts to access a masked address space generate an external bus access.

Addresses hitting overlapping memory spaces take the following priority:

1. MBAR
2. RAM, ROM, and caches
3. Chip select

#### NOTE:

The MBAR region must be mapped to non-cacheable space.



**Figure 2-9. Module Base Address Register (MBAR)**

Table 2-3 describes MBAR fields.

**Table 2-3. MBAR Field Descriptions**

Bits	Field	Description
31–12	BA	Base address. Defines the base address for a 4-Kbyte address range.
11–9	—	Reserved, should be cleared.
8	WP	Write protect. Mask bit for write cycles in the MBAR-mapped register address range. 0 Module address range is read/write. 1 Module address range is read only.
7	—	Reserved, should be cleared.

Table 2-3. MBAR Field Descriptions (Continued)

Bits	Field	Description
6	AM	Alternate master mask. When AM = 0 and an alternate master (external master or DMA) accesses MBAR-mapped registers, MBAR[SC,SD,UC,UD] are ignored in address decoding. These fields mask address space, placing the MBAR-mapped register in a specific address space or spaces.
5	C/I	Mask CPU space and interrupt acknowledge cycles. Note that C/I must be set if BA = 0. 0 Activates the corresponding MBAR-mapped register 1 Regular external bus access
4	SC	Setting masks supervisor code space in MBAR address range
3	SD	Setting masks supervisor data space in MBAR address range
2	UC	Setting masks user code space in MBAR address range
1	UD	Setting masks user data space in MBAR address range
0	V	Valid. Determines whether MBAR settings are valid. 0 MBAR contents are invalid. 1 MBAR contents are valid.

The following example shows how to set the MBAR to location 0x1000\_0000 using the D0 register. Setting MBAR[V] validates the MBAR location. This example assumes all accesses are valid:

```
move.l #0x10000001,D0
movec D0,MBAR
```

## 2.4 Programming Model Table

Table 2-4 lists register names, the CPU space location, whether the register is written from the processor using the MOVEC instruction, and the complete register name.

Table 2-4. ColdFire CPU Registers

Name	CPU Space (Rc)	Written with MOVEC	Register Name
<b>Memory Management Control Registers</b>			
CACR	0x002	Yes	Cache control register
ASID	0x003	Yes	Address space identifier
ACR0–ACR3	0x004–0x007	Yes	Access control registers 0–3
MMUBAR	0x008	Yes	MMU base address register
<b>Processor General-Purpose Registers</b>			
D0–D7	0x(0,1)80–0x(0,1)87	No	Data registers 0–7 (0 = load, 1 = store)
A0–A7	0x(0,1)88–0x(0,1)8F	No	Address registers 0–7 (0 = load, 1 = store) A7 is user stack pointer

Table 2-4. ColdFire CPU Registers (Continued)

Name	CPU Space (Rc)	Written with MOVEC	Register Name
<b>Processor Miscellaneous Registers</b>			
OTHER_A7	0x800	No	Other stack pointer
VBR	0x801	Yes	Vector base register
MACSR	0x804	No	MAC status register
MASK	0x805	No	MAC address mask register
ACC0–ACC3	0x806–0x80B	No	MAC accumulators 0–3
ACCext01	0x807	No	MAC accumulator 0, 1 extension bytes
ACCext23	0x808	No	MAC accumulator 2, 3 extension bytes
SR	0x80E	No	Status register
PC	0x80F	Yes	Program counter
<b>Processor Floating-Point Registers</b>			
FPU0	0x810	No	32 msbs of floating-point data register 0
FPL0	0x811	No	32 lsbs of floating-point data register 0
FPU1	0x812	No	32 msbs of floating-point data register 1
FPL1	0x813	No	32 lsbs of floating-point data register 1
FPU2	0x814	No	32 msbs of floating-point data register 2
FPL2	0x815	No	32 lsbs of floating-point data register 2
FPU3	0x816	No	32 msbs of floating-point data register 3
FPL3	0x817	No	32 lsbs of floating-point data register 3
FPU4	0x818	No	32 msbs of floating-point data register 4
FPL4	0x819	No	32 lsbs of floating-point data register 4
FPU5	0x81A	No	32 msbs of floating-point data register 5
FPL5	0x81B	No	32 lsbs of floating-point data register 5
FPU6	0x81C	No	32 msbs of floating-point data register 6
FPL6	0x81D	No	32 lsbs of floating-point data register 6
FPU7	0x81E	No	32 msbs of floating-point data register 7
FPL7	0x81F	No	32 lsbs of floating-point data register 7
FPIAR	0x821	No	Floating-point instruction address register
FPSR	0x822	No	Floating-point status register
FPCR	0x824	No	Floating-point control register

**Table 2-4. ColdFire CPU Registers (Continued)**

Name	CPU Space (Rc)	Written with MOVEC	Register Name
<b>Local Memory and Module Control Registers</b>			
ROMBAR0	0xC00	Yes	ROM base address register 0
ROMBAR1	0xC01	Yes	ROM base address register 1
RAMBAR0	0xC04	Yes	RAM base address register 0
RAMBAR1	0xC05	Yes	RAM base address register 1
MBAR	0xC0F	Yes	Primary module base address register (not a core register)

# Chapter 3

## Instructions

The *ColdFire Family Programmer's Reference Manual*, or *PRM*, describes ColdFire instructions, addressing modes, and data formats for all ColdFire processors as well as optional modules such as the FPU and EMAC.





# Chapter 4

## Floating-Point Unit (FPU)

This chapter describes instructions implemented in the floating-point unit (FPU) designed for use with the ColdFire family of microprocessors. The FPU conforms to the American National Standards Institute (ANSI)/Institute of Electrical and Electronics Engineers (IEEE) *Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754).

The FPU does not support all IEEE-754 number types and operations in hardware; the hardware unit is optimized for real-time execution with exceptions disabled and default results provided for specific operations, operands, and number types. Exceptions can be enabled to support these cases in software.

### 4.1 FPU Overview

The FPU operates on 64-bit, double-precision floating-point data and supports single-precision and signed integer input operands. It can be used with ColdFire microarchitecture, Version 4 and higher. The FPU programming model is like that in the MC68060 microprocessor. The FPU is intended to accelerate the performance of certain classes of embedded applications, especially those requiring high-speed floating-point arithmetic computations. See Section 4.4.3, “Key Differences between ColdFire and MC680x0 FPU Programming Models.”

The FPU appears as another execute engine at the bottom stages of the operand execution pipeline (OEP), using operands from a dual-ported register file.

Setting bit 4 in the cache control register (CACR[DF]) disables the FPU. If CACR[DF] is cleared, all FPU instructions are issued and executed, otherwise the processor responds with a line F instruction exception (vector 11).

Operating systems often assume user applications are integer-only (to minimize the time required by save context) by setting CACR[DF] at process initiation. If the application includes floating-point instructions, the attempted execution of the first FP instruction generates the line F exception, which signals the kernel that the FPU registers must be included in the context for the application. The application then continues execution with CACR[DF] cleared to enable FPU execution.

## 4.1.1 Notational Conventions

Table 4-1 defines notational conventions used in this chapter. Table 4-2 describes addressing modes and syntax for floating-point instructions.

**Table 4-1. Notational Conventions**

Symbol	Description
<b>Single- and Double-Precision Operand Operations</b>	
+	Arithmetic addition or postincrement indicator
–	Arithmetic subtraction or predecrement indicator
×	Arithmetic multiplication
÷	Arithmetic division or conjunction symbol
~	Invert, operand is logically complemented. An overbar, $\bar{\phantom{x}}$ , is also used for this operation.
&	Logical AND
	Logical OR
→	Source operand is moved to destination operand
<op>	Any double-operand operation
<operand>tested	Operand is compared to zero and the condition codes are set appropriately
sign-extended	All bits of the upper portion are made equal to the high-order bit of the lower portion
<b>Other Operations</b>	
If <condition> then <operations> else <operations>	Test the condition. If true, the operations after then are performed. If the condition is false and the optional else clause is present, the operations after else are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.
<b>Register Specifications</b>	
An	Address register n (example: A3 is address register 3)
Ay, Ax	Source and destination address registers, respectively
Dn	Data register n (example: D3 is data register 3)
Dy,Dx	Source and destination data registers, respectively
FPCR	Floating-point control register
FPIAR	Floating-point instruction address register
FPn	Floating-point data register n (example: FP3 is FPU data register 3)
FPSR	Floating-point status register
FPy,FPx	Source and destination floating-point data registers, respectively
PC	Program counter
Rn	Address or data register
Rx	Destination register
Ry	Source register
Xi	Index register

Table 4-2 lists floating-point addressing modes.

**Table 4-2. Floating-Point Addressing Modes**

Addressing Modes	Syntax
Register direct Address register direct Address register direct	Dy Ay
Register indirect Address register indirect Address register indirect with postincrement Address register indirect with predecrement Address register indirect with displacement	(Ay) -(Ay) (d <sub>16</sub> ,Ay)
Program counter indirect with displacement	(d <sub>16</sub> ,PC)

## 4.2 Operand Data Formats and Types

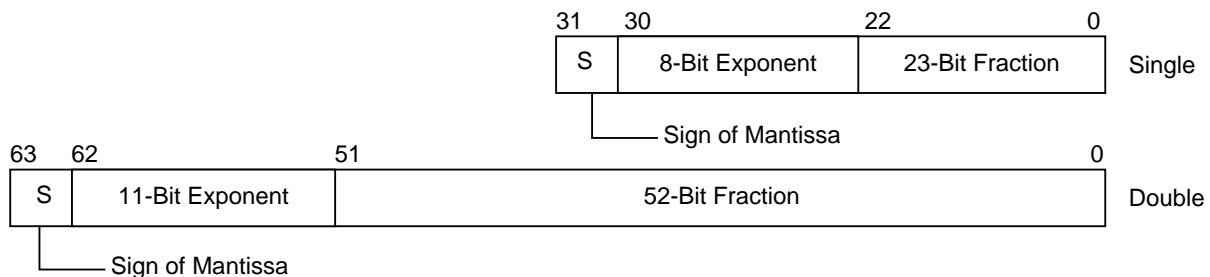
The FPU supports signed byte, word, and longword integer formats, which are identical to those supported by the integer unit. The FPU also supports single- and double-precision binary floating-point formats that fully comply with the IEEE-754 standard.

### 4.2.1 Signed-integer Data Formats

The FPU supports 8-bit byte (B), 16-bit word (W), and 32-bit longword (L) integer data formats.

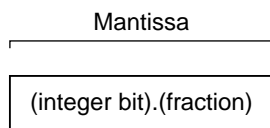
### 4.2.2 Floating-Point Data Formats

Figure 4-1 shows the two binary floating-point data formats.



**Figure 4-1. Floating-Point Data Formats**

Note that, throughout this chapter, a mantissa is defined as the concatenation of an integer bit, the binary point, and a fraction. A fraction is the term designating the bits to the right of the binary point in the mantissa.

**Figure 4-2. Mantissa**

The integer bit is implied to be set for normalized numbers and infinities, clear for zeros and denormalized numbers. For not-a-numbers (NaNs), the integer bit is ignored. The exponent in both floating-point formats is an unsigned binary integer with an implied bias added to it. Subtracting the bias from exponent yields a signed, two's complement power of two. This represents the magnitude of a normalized floating-point number when multiplied by the mantissa.

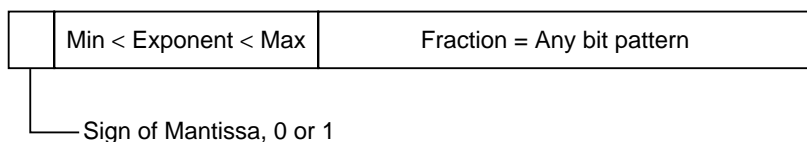
By definition, a normalized mantissa always takes values starting from 1.0 and going up to, but not including, 2.0; that is, [1.0...2.0).

### 4.2.3 Floating-Point Data Types

Each floating-point data format supports five unique data types: normalized numbers, zeros, infinities, NaNs, and denormalized numbers. The normalized data type, Figure 4-3, never uses the maximum or minimum exponent value for a given format.

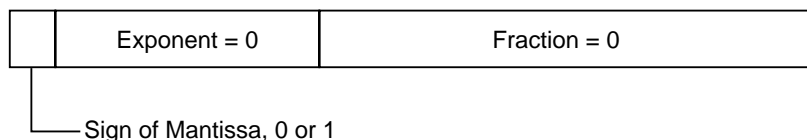
#### 4.2.3.1 Normalized Numbers

Normalized numbers include all positive or negative numbers with exponents between the maximum and minimum values. For single- and double-precision normalized numbers, the implied integer bit is one and the exponent can be zero.

**Figure 4-3. Normalized Number Format**

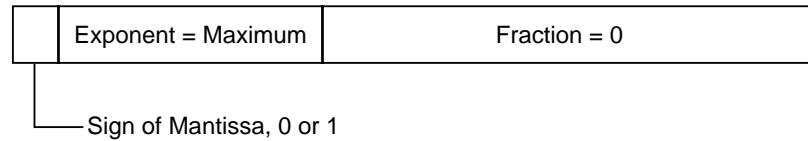
#### 4.2.3.2 Zeros

Zeros can be positive or negative and represent real values, + 0.0 and – 0.0. See Figure 4-4.

**Figure 4-4. Zero Format**

### 4.2.3.3 Infinities

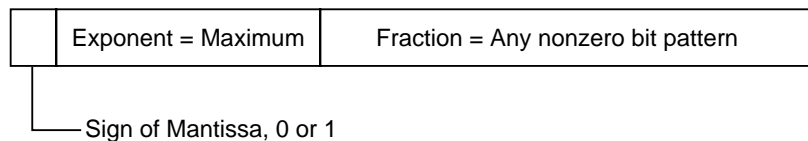
Infinities can be positive or negative and represent real values that exceed the overflow threshold. A result's exponent greater than or equal to the maximum exponent value indicates an overflow for a given data format and operation. This overflow description ignores the effects of rounding and the user-selectable rounding models. For single- and double-precision infinities, the fraction is a zero. See Figure 4-5.



**Figure 4-5. Infinity Format**

### 4.2.3.4 Not-A-Number

When created by the FPU, NaNs represent the results of operations having no mathematical interpretation, such as infinity divided by infinity. Operations using a NaN operand as an input return a NaN result. User-created NaNs can protect against uninitialized variables and arrays or can represent user-defined data types. See Figure 4-6.

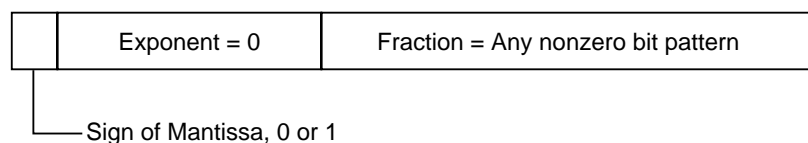


**Figure 4-6. Not-a-Number Format**

If an input operand to an operation is a NaN, the result is an FPU-created default NaN. When the FPU creates a NaN, the NaN always contains the same bit pattern in the mantissa: all mantissa bits are ones and the sign bit is zero. When the user creates a NaN, any nonzero bit pattern can be stored in the mantissa and the sign bit.

### 4.2.3.5 Denormalized Numbers

Denormalized numbers represent real values near the underflow threshold. Denormalized numbers can be positive or negative. For denormalized numbers in single- and double-precision, the implied integer bit is a zero. See Figure 4-7.



**Figure 4-7. Denormalized Number Format**

Traditionally, the detection of underflow causes floating-point number systems to perform a flush-to-zero. The IEEE-754 standard implements gradual underflow: the result mantissa is shifted right (denormalized) while the result exponent is incremented until reaching the

## Operand Data Formats and Types

minimum value. If all the mantissa bits of the result are shifted off to the right during this denormalization, the result becomes zero.

Denormalized numbers are not supported directly in the hardware of this implementation but can be handled in software if needed (software for the input denorm exception could be written to handle denormalized input operands, and software for the underflow exception could create denormalized numbers). If the input denorm exception is disabled, all denormalized numbers are treated as zeros.

Table 4-3 summarizes the data type specifications for byte, word, longword, single- and double-precision data formats.

**Table 4-3. Real Format Summary**

Parameter	Single-Precision	Double-Precision
Data Format	<div> <div>3130</div> <div>2322</div> <div>0</div> <div>s</div> <div>e</div> <div>f</div> </div>	<div> <div>6362</div> <div>5251</div> <div>0</div> <div>s</div> <div>e</div> <div>f</div> </div>
<b>Field Size in Bits</b>		
Sign (s)	1	1
Biased exponent (e)	8	11
Fraction (f)	23	52
Total	32	64
<b>Interpretation of Sign</b>		
Positive fraction	$s = 0$	$s = 0$
Negative fraction	$s = 1$	$s = 1$
<b>Normalized Numbers</b>		
Bias of biased exponent	+127 (0x7F)	+1023 (0x3FF)
Range of biased exponent	$0 < e < 255$ (0xFF)	$0 < e < 2047$ (0x7FF)
Range of fraction	Zero or Nonzero	Zero or Nonzero
Mantissa	1.f	1.f
Relation to representation of real numbers	$(-1)^s \times 2^{e-127} \times 1.f$	$(-1)^s \times 2^{e-1023} \times 1.f$
<b>Denormalized Numbers</b>		
Biased exponent format minimum	0 (0x00)	0 (0x000)
Bias of biased exponent	+126 (0x7E)	+1022 (0x3FE)
Range of fraction	Nonzero	Nonzero
Mantissa	0.f	0.f
Relation to representation of real numbers	$(-1)^s \times 2^{-126} \times 0.f$	$(-1)^s \times 2^{-1022} \times 0.f$
<b>Signed Zeros</b>		
Biased exponent format minimum	0 (0x00)	0 (0x00)
Mantissa	$0.f = 0.0$	$0.f = 0.0$

Table 4-3. Real Format Summary (Continued)

Parameter	Single-Precision	Double-Precision
<b>Signed Infinities</b>		
Biased exponent format maximum	255 (0xFF)	2047 (0x7FF)
Mantissa	0.f = 0.0	0.f = 0.0
<b>NANs</b>		
Sign	Don't Care	0 or 1
Biased exponent format maximum	255 (0xFF)	2047 (0x7FF)
Fraction	Nonzero	Nonzero
Representation of Fraction Nonzero Bit Pattern Created by User Fraction When Created by FPU	xxxxx...xxxx 11111...1111	xxxxx...xxxx 11111...1111
<b>Approximate Ranges</b>		
Maximum Positive Normalized	$3.4 \times 10^{38}$	$1.8 \times 10^{308}$
Minimum Positive Normalized	$1.2 \times 10^{-38}$	$2.2 \times 10^{-308}$
Minimum Positive Denormalized	$1.4 \times 10^{-45}$	$4.9 \times 10^{-324}$

## 4.3 FPU Programmer's Model

The programmer's model for the FPU consists of the following:

- Eight 64-bit floating-point data registers (FP0–FP7)
- One 32-bit floating-point control register (FPCR)
- One 32-bit floating-point status register (FPSR)
- One 32-bit floating-point instruction address register (FPIAR)

Figure 4-8 shows the FPU programming model.

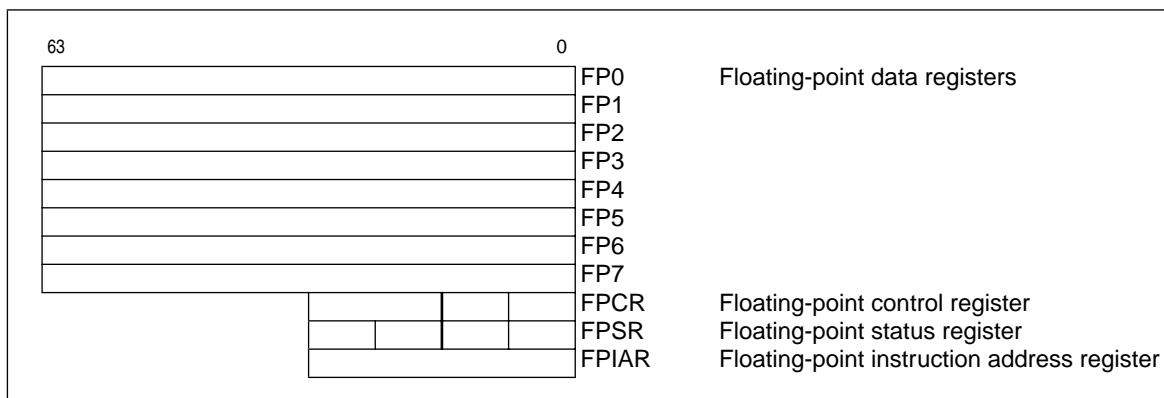


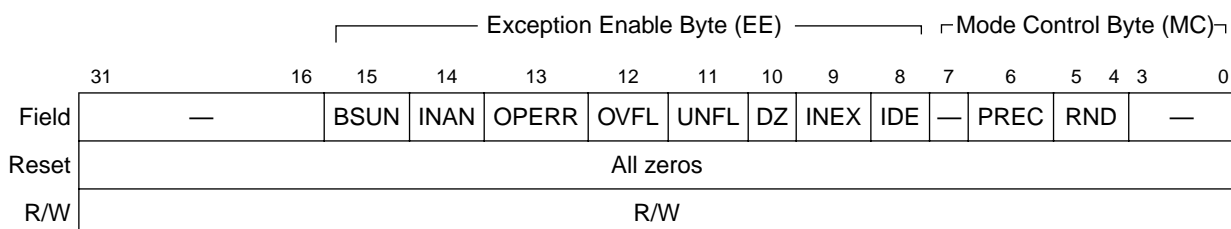
Figure 4-8. Floating-Point Programmer's Model

### 4.3.1 Floating-Point Data Registers (FP0–FP7)

Floating-point data registers are analogous to the integer data registers for the 68K/ColdFire family. They always contain numbers in double-precision format, even though the operand may be a single-precision value used in a single-precision calculation. All external operands, regardless of the source data format, are converted to double-precision format before being used in any calculation or being stored in a floating-point data register. A reset or a null-restore operation sets FP0–FP7 to positive, nonsignaling NaNs.

#### 4.3.1.1 Floating-Point Control Register (FPCR)

The FPCR, Figure 4-9, contains an exception enable byte (EE) and a mode control byte (MC). The user can read or write to FPCR using FMOVE or FRESTORE. A processor reset or a restore operation of the null state clears the FPCR. When this register is cleared, the FPU never generates exceptions.



**Figure 4-9. Floating-Point Control Register (FPCR)**

Table 4-4 describes FPCR fields.

**Table 4-4. FPCR Field Descriptions**

Bits	Field	Description
31–16	—	Reserved, should be cleared.
15–8	EE	Exception enable byte. Each EE bit corresponds to a floating-point exception class. The user can separately enable traps for each class of floating-point exceptions.
15	BSUN	Branch set on unordered
14	INAN	Input not-a-number
13	OPERR	Operand error
12	OVFL	Overflow
11	UNFL	Underflow
10	DZ	Divide by zero
9	INEX	Inexact operation
8	IDE	Input denormalized



**Table 4-4. FPCR Field Descriptions (Continued)**

Bits	Field	Description
7–0	MC	Mode control byte. Control FPU operating modes.
7		— Reserved, should be cleared.
6		PREC Rounding precision 0 Double (D) 1 Single (S)
5–4		RND Rounding mode 00 To nearest (RN) 01 To zero (RZ) 10 To minus infinity (RM) 11 To plus infinity (RP)
3–0		— Reserved, should be cleared.

### 4.3.2 Floating-Point Status Register (FPSR)

The FPSR, Figure 4-10, contains a floating-point condition code byte (FPCC), a floating-point exception status byte (EXC), and a floating-point accrued exception byte (AEXC). The user can read or write all FPSR bits. Execution of most floating-point instructions modifies FPSR. FPSR is loaded using FMOVE or FRESTORE. A processor reset or a restore operation of the null state clears the FPSR.

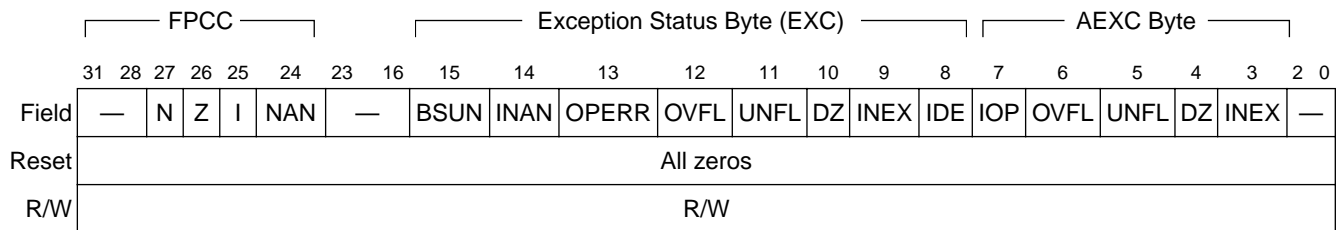
**Figure 4-10. Floating-Point Status Register (FPSR)**

Table 4-5 describes FPSR fields.

**Table 4-5. FPSR Field Descriptions**

Bits	Field	Description
31–24	FPCC	Floating-point condition code byte. Contains four condition code bits that are set after completion of all arithmetic instructions involving the floating-point data registers. The floating-point store operation, FMOVE, and move system control register instructions do not affect the FPCC.
31–28		Reserved, should be cleared.
27		N Negative
26		Z Zero
25		I Infinity
24		NAN Not-a-number
23–16	—	Reserved, should be cleared.

**Table 4-5. FPSR Field Descriptions (Continued)**

Bits	Field	Description	
15–8	EXC	Exception status byte. Contains a bit for each floating-point exception that might have occurred during the most recent arithmetic instruction or move operation. This byte is cleared at the start of all operations that generate floating-point exceptions (except FBcc only affects BSUN and that only for nonaware tests). Operations that do not generate floating-point exceptions do not clear this byte. An exception handler can use this byte to determine which floating-point exception or exceptions caused a trap. The equations below the table show the comparative relationship between the EXC byte and AEXC byte.	
15	EXC	BSUN	Branch/set on unordered
14		INAN	Input not-a-number
13		OPERR	Operand error
12		OVFL	Overflow
11		UNFL	Underflow
10		DZ	Divide by zero
9		INEX	Inexact result
8		IDE	Input is denormalized
7–0	AEXC	<p>Accrued exception byte. Contains 5 required bits for IEEE-754 exception-disabled operations. These exceptions are logical combinations of EXC bits. AEXC records all floating-point exceptions since AEXC was last cleared, either by writing to FPSR or as a result of reset or a restore operation of the null state.</p> <p>Many users disable traps for some or all floating-point exception classes. AEXC eliminates the need to poll EXC after each floating-point instruction. At the end of arithmetic operations, EXC bits are logically combined to form an AEXC value that is logically ORed into the existing AEXC byte (FBcc only updates IOP). This operation creates sticky floating-point exception bits in AEXC that the user can poll only at the end of a series of floating-point operations. A sticky bit is one that remains set until the user clears it.</p> <p>Setting or clearing AEXC bits neither causes nor prevents an exception. The equations below the table show relationships between EXC and AEXC. Comparing the current value of an AEXC bit with a combination of EXC bits derives a new value in the corresponding AEXC bit. These boolean equations apply to setting AEXC bits at the end of each operation affecting AEXC.</p>	
7	AEXC	IOP	Invalid operation
6		OVFL	Overflow
5		UNFL	Underflow
4		DZ	Divide by zero
3		INEX	Inexact result
2–0	—	Reserved, should be cleared.	

For AEXC[OVFL], AEXC[DZ], and AEXC[INEX], the next value is determined by ORing the current AEXC value with the EXC equivalent, as shown in the following:

- Next AEXC[OVFL] = Current AEXC[OVFL] | EXC[OVFL]
- Next AEXC[DZ] = Current AEXC[DZ] | EXC[DZ]
- Next AEXC[INEX] = Current AEXC[INEX] | EXC[INEX]

For AEXC[IOP] and AEXC[UNFL], the next value is calculated by ORing the current AEXC value with EXC bit combinations, as follows:

- Next AEXC[IOP] = Current AEXC[IOP] | EXC[BSUN | INAN | OPERR]
- Next AEXC[UNFL] = Current AEXC[UNFL] | EXC[UNFL & INEX]

### 4.3.3 Floating-Point Instruction Address Register (FPIAR)

The ColdFire OEP can execute integer and floating-point instructions simultaneously. As a result, the PC value stacked by the processor in response to a floating-point exception trap may not point to the instruction that caused the exception.

For FPU instructions that can generate exception traps, the 32-bit FPIAR is loaded with the instruction PC address before the FPU begins execution. In case of an FPU exception, the trap handler can use the FPIAR contents to determine the instruction that generated the exception. FMOVE to/from FPCR, FPSR, or FPIAR and FMOVEM instructions cannot generate floating-point exceptions and so do not modify FPIAR. A reset or a null-restore operation clears FPIAR.

### 4.3.4 Floating-Point Computational Accuracy

The FPU performs all floating-point internal operations in double-precision. It supports mixed-mode arithmetic by converting single-precision operands to double-precision values before performing the specified operation. The FPU converts all memory data formats to the double-precision data format and stores the value in a floating-point register or uses it as the source operand for an arithmetic operation. When moving a double-precision floating-point value from a floating-point data register, the FPU can convert the data depending on the destination, as follows:

- Valid data formats for memory destination: B, W, L, S, or D
- Valid data formats for integer data register destinations: B, W, L, or S

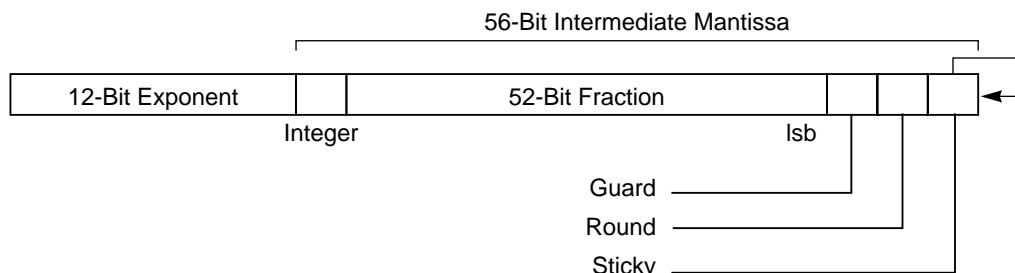
Normally if the input operand is a denormalized number, the number must be normalized before an FPU instruction can be executed. A denormalized input operand is converted to zero if the input denorm exception (IDE) is disabled. If IDE is enabled, the floating-point engine traps to allow software action to be taken by the handler.

#### 4.3.4.1 Intermediate Result

All FPU calculations use an intermediate result. When the FPU performs any operation, the calculation is carried out using double-precision inputs, and the intermediate result is calculated as if to produce infinite precision. After the calculation is complete, any necessary rounding of the intermediate result for the selected precision is performed and the result is stored in the destination.

Figure 4-11 shows the intermediate result format. The intermediate result's exponent for some dyadic operations (for example, multiply and divide) can easily overflow or underflow the 11-bit exponent of the designation floating-point register. To simplify overflow and underflow detection, intermediate results in the FPU maintain a 12-bit two's

complement, integer exponent. Detection of an intermediate result overflow or underflow always converts the 12-bit exponent into a 11-bit biased exponent before being stored in a floating-point data register. The FPU internally maintains a 56-bit mantissa for rounding purposes. The mantissa is always rounded to 53 bits (or fewer, depending on the selected rounding precision) before it is stored in a floating-point data register.



**Figure 4-11. Intermediate Result Format**

If the destination is a floating-point data register, the result is in double-precision format but may be rounded to single-precision, if required by the rounding precision, before being stored. If the single-precision mode is selected, the exponent value is in the correct range even if it is stored in double-precision format. If the destination is a memory location or an integer data register, rounding precision is ignored. In this case, a number in the double-precision format is taken from the source floating-point data register, rounded to the destination format precision, and then written to memory or the integer data register.

Depending on the selected rounding mode or destination data format, the location of the lsb of the mantissa and the locations of the guard, round, and sticky bits in the 56-bit intermediate result mantissa vary. Guard and round bits are calculated exactly. The sticky bit creates the illusion of an infinitely wide intermediate result. As the arrow in Figure 4-11 shows, the sticky bit is the logical OR of all bits to the right of the round bit in the infinitely precise result. During calculation, nonzero bits generated to the right of the round bit set the sticky bit. Because of the sticky bit, the rounded intermediate result for all required IEEE arithmetic operations in RN mode can err by no more than one half unit in the last place.

#### 4.3.4.2 Rounding the Result

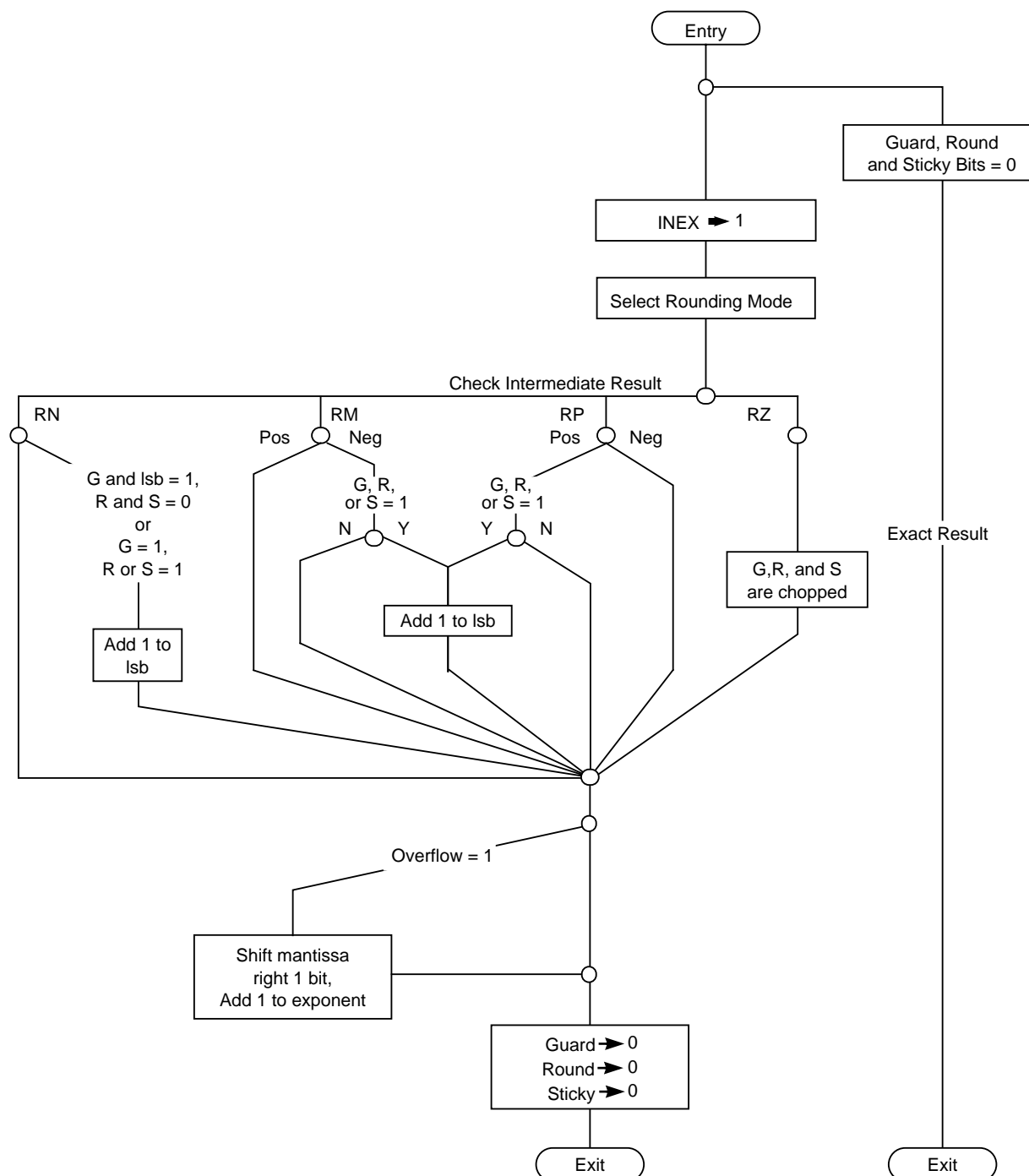
The FPU supports the four rounding modes specified by the IEEE-754 standard: round-to-nearest (RN), round-toward-zero (RZ), round-toward-plus-infinity (RP), and round-toward-minus-infinity (RM). The RM and RP modes are often referred to as directed-rounding-modes and are useful in interval arithmetic. Rounding is accomplished through the intermediate result. Single-precision results are rounded to a 24-bit mantissa boundary; double-precision results are rounded to a 53-bit mantissa boundary.

The current floating-point instruction can specify rounding precision, overriding the rounding precision specified in FPCR for the duration of the current instruction. For

example, the rounding precision for FADD is determined by FPCR, while the rounding precision for FSADD is single-precision, independent of FPCR.

Range control helps emulate devices that support only single-precision arithmetic by rounding the intermediate result's mantissa to the specified precision and checking that the intermediate exponent is in the representable range of the selected rounding precision. If the intermediate result's exponent exceeds the range, the appropriate underflow or overflow value is stored as the result in the double-precision format exponent. For example, if the data format and rounding mode is single-precision RM and the result of an arithmetic operation overflows the single-precision format, the maximum normalized single-precision value is stored as a double-precision number in the destination floating-point data register; that is, the unbiased 11-bit exponent is 0x0FF and the 52-bit fraction is 0xF\_FFFF\_E000\_0000. If an infinity is the appropriate result for an underflow or overflow, the infinity value for the destination data format is stored as the result; that is, the exponent has the maximum value and the mantissa is zero.

Figure 4-12 shows the algorithm for rounding an intermediate result to the selected rounding precision and destination data format. If the destination is a floating-point register, the rounding boundary is determined by either the selected rounding precision specified by FPCR[PREC] or by the instruction itself. For example, FSADD and FDADD specify single- and double-precision rounding regardless of FPCR[PREC]. If the destination is memory or an integer data register, the destination data format determines the rounding boundary. If the rounded result of an operation is inexact, INEX is set in FPSR[EXC].



**Figure 4-12. Rounding Algorithm Flowchart**

The three additional bits beyond the double-precision format, the difference between the intermediate result's 56-bit mantissa and the storing result's 53-bit mantissa, allow the FPU to perform all calculations as though it were performing calculations using a compute engine with infinite bit precision. The result is always correct for the specified destination's data format before rounding (unless an overflow or underflow error occurs). The specified rounding produces a number as close as possible to the infinitely precise

intermediate value and still representable in the selected precision. The tie case in Table 4-6 shows how the 56-bit mantissa allows the FPU to meet the error bound of the IEEE specification.

**Table 4-6. Tie-Case Example**

Result	Integer	52-Bit Fraction	Guard	Round	Sticky
Intermediate	x	xxx...x00	1	0	0
Rounded-to-Nearest	x	xxx...x00	0	0	0

The lsb of the rounded result does not increment even though the guard bit is set in the intermediate result. The IEEE-754 standard specifies this way of handling ties. If the destination data format is double-precision and there is a difference between the infinitely precise intermediate result and the round-to-nearest result, the relative difference is  $2^{-53}$  (the value of the guard bit). This error is equal to half of the lsb's value and is the worst case error that can be introduced with RN mode. Thus, the term one-half unit in the last place correctly identifies the error bound for this operation. This error specification is the relative error present in the result; the absolute error bound is equal to  $2^{\text{exponent}} \times 2^{-53}$ . Table 4-7 shows the error bound for other rounding modes.

**Table 4-7. Round Mode Error Bounds**

Result	Integer	52-Bit Fraction	Guard	Round	Sticky
Intermediate	x	xxx...x00	1	1	1
Rounded-to-Zero	x	xxx...x00	0	0	0

The difference between the infinitely precise result and the rounded result is  $2^{-53} + 2^{-54} + 2^{-55}$ , which is slightly less than  $2^{-52}$  (the value of the lsb). Thus, the error bound for this operation is not more than one unit in the last place. The FPU meets these error bounds for all arithmetic operations, providing accurate, repeatable results.

### 4.3.5 Floating-Point Post Processing

Most operations end with post-processing, for which the FPU provides two steps. First, FPSR[FPCC] bits are set or cleared at the end of each arithmetic or move operation to a single floating-point data register. FPCC bits are consistently set based on the result of the operation. Second, the FPU supports 32 conditional tests that allow floating-point conditional instructions to test floating-point conditions in the same way that integer conditional instructions test the integer condition code. The combination of consistently set FPCC bits and the simple programming of conditional instructions gives the processor a very flexible, efficient way to change program flow based on floating-point results. When the summary for each instruction is read, it should be assumed that an instruction performs post processing unless the summary specifically states otherwise. The following paragraphs describe post processing in detail.

### 4.3.5.1 Underflow, Round, Overflow

During calculation of an arithmetic result, the FPU has more precision and range than the 64-bit double-precision format. However, the final result is a double-precision value. In some cases, an intermediate result becomes either smaller or larger than can be represented in double-precision. Also, the operation can generate a larger exponent or more bits of precision than can be represented in the chosen rounding precision. For these reasons, every arithmetic instruction ends by checking for underflow, rounding the result and checking for overflow.

At the completion of an arithmetic operation, the intermediate result is checked to see if it is too small to be represented as a normalized number in the selected precision. If so, the underflow (UNFL) bit is set in FPSR[EXC]. If no underflow occurs, the intermediate result is rounded according to the user-selected rounding precision and mode. After rounding, the inexact bit (INEX) is set as described in Figure 4-12. Lastly, the magnitude of the result is checked to see if it exceeds the current rounding precision. If so, the overflow (OVFL) bit is set and a correctly signed infinity or correctly signed largest normalized number is returned, depending on the rounding mode.

**NOTE:**

INEX can also be set by OVFL, UNFL, and when denormalized numbers are encountered.

### 4.3.5.2 Conditional Testing

Unlike operation-dependent integer condition codes, an instruction either always sets FPCC bits in the same way or does not change them at all. Therefore, instruction descriptions do not include FPCC settings. This section describes how FPCC bits are set.

FPCC bits differ slightly from integer condition codes. An FPU operation's final result sets or clears FPCC bits accordingly, independent of the operation itself. Integer condition codes bits N and Z have this characteristic, but V and C are set differently for different instructions. Table 4-8 lists FPCC settings for each data type. Loading FPCC with another combination and executing a conditional instruction can produce an unexpected branch condition.

**Table 4-8. FPCC Encodings**

Data Type	N	Z	I	NAN
+ Normalized or Denormalized	0	0	0	0
– Normalized or Denormalized	1	0	0	0
+ 0	0	1	0	0
– 0	1	1	0	0
+ Infinity	0	0	1	0
– Infinity	1	0	1	0



**Table 4-8. FPCC Encodings (Continued)**

Data Type	N	Z	I	NAN
+ NAN	0	0	0	1
– NAN	1	0	0	1

The inclusion of the NAN data type in the IEEE floating-point number system requires each conditional test to include FPCC[NAN] in its boolean equation. Because it cannot be determined whether a NAN is bigger or smaller than an in-range number (that is, it is unordered), the compare instruction sets FPCC[NAN] when an unordered compare is attempted. All arithmetic instructions that result in a NAN also set the NAN bit. Conditional instructions interpret NAN being set as the unordered condition.

The IEEE-754 standard defines the following four conditions:

- Equal to (EQ)
- Greater than (GT)
- Less than (LT)
- Unordered (UN)

The standard requires only the generation of the condition codes as a result of a floating-point compare operation. The FPU can test for these conditions and 28 others at the end of any operation affecting condition codes. For floating-point conditional branch instructions, the processor logically combines the 4 bits of the FPCC condition codes to form 32 conditional tests, 16 of which cause an exception if an unordered condition is present when the conditional test is attempted (IEEE nonaware tests). The other 16 do not cause an exception (IEEE-aware tests). The set of IEEE nonaware tests is best used in one of the following cases:

- When porting a program from a system that does not support the IEEE standard to a conforming system
- When generating high-level language code that does not support IEEE floating-point concepts (that is, the unordered condition).

An unordered condition occurs when one or both of the operands in a floating-point compare operation is a NAN. The inclusion of the unordered condition in floating-point branches destroys the familiar trichotomy relationship (greater than, equal, less than) that exists for integers. For example, the opposite of floating-point branch greater than (FBGT) is not floating-point branch less than or equal (FBLE). Rather, the opposite condition is floating-point branch not greater than (FBNGT). If the result of the previous instruction was unordered, FBNGT is true; whereas, both FBGT and FBLE would be false because unordered fails both of these tests (and sets BSUN). Compiler code generators should be particularly careful of the lack of trichotomy in the floating-point branches because it is common for compilers to invert the sense of conditions.

When using the IEEE nonaware tests, the user receives a BSUN exception if a branch is attempted and FPCC[NAN] is set, unless the branch is an FBEQ or an FBNE. If the BSUN exception is enabled in FPCR, the exception takes a BSUN trap. Therefore, the IEEE nonaware program is interrupted if an unexpected condition occurs. Users knowledgeable of the IEEE-754 standard should use IEEE-aware tests in programs that contain ordered and unordered conditions. Because the ordered or unordered attribute is explicitly included in the conditional test, EXC[BSUN] is not set when the unordered condition occurs. Table 4-9 summarizes conditional mnemonics, definitions, equations, predicates, and whether EXC[BSUN] is set for the 32 floating-point conditional tests. The equation column lists FPCC bit combinations for each test in the form of an equation. Condition codes with an overbar indicate cleared bits; all other bits are set.

**Table 4-9. Floating-Point Conditional Tests**

Mnemonic	Definition	Equation	Predicate <sup>1</sup>	EXC[BSUN] Set
<b>IEEE Nonaware Tests</b>				
EQ	Equal	$Z$	000001	No
NE	Not equal	$\bar{Z}$	001110	No
GT	Greater than	$\overline{NAN}   Z   \bar{N}$	010010	Yes
NGT	Not greater than	$NAN   Z   N$	011101	Yes
GE	Greater than or equal	$Z   (\overline{NAN}   \bar{N})$	010011	Yes
NGE	Not greater than or equal	$NAN   (N \& \bar{Z})$	011100	Yes
LT	Less than	$N \& (\overline{NAN}   \bar{Z})$	010100	Yes
NLT	Not less than	$NAN   (Z   \bar{N})$	011011	Yes
LE	Less than or equal	$Z   (N \& \overline{NAN})$	010101	Yes
NLE	Not less than or equal	$NAN   (\bar{N}   \bar{Z})$	011010	Yes
GL	Greater or less than	$\overline{NAN}   \bar{Z}$	010110	Yes
NGL	Not greater or less than	$NAN   Z$	011001	Yes
GLE	Greater, less or equal	$\overline{NAN}$	010111	Yes
NGLE	Not greater, less or equal	$NAN$	011000	Yes
<b>IEEE-Aware Tests</b>				
EQ	Equal	$Z$	000001	No
NE	Not equal	$\bar{Z}$	001110	No
OGT	Ordered greater than	$\overline{NAN}   \bar{Z}   \bar{N}$	000010	No
ULE	Unordered or less or equal	$NAN   Z   N$	001101	No
OGE	Ordered greater than or equal	$Z   (\overline{NAN}   \bar{N})$	000011	No
ULT	Unordered or less than	$NAN   (N \& \bar{Z})$	001100	No
OLT	Ordered less than	$N \& (\overline{NAN}   \bar{Z})$	000100	No
UGE	Unordered or greater or equal	$NAN   (Z   \bar{N})$	001011	No
OLE	Ordered less than or equal	$Z   (N \& \overline{NAN})$	000101	No

**Table 4-9. Floating-Point Conditional Tests (Continued)**

Mnemonic	Definition	Equation	Predicate <sup>1</sup>	EXC[BSUN] Set
UGT	Unordered or greater than	$\text{NAN} \mid (\overline{\text{N}} \mid \overline{\text{Z}})$	001010	No
OGL	Ordered greater or less than	$\overline{\text{NAN}} \mid \overline{\text{Z}}$	000110	No
UEQ	Unordered or equal	$\text{NAN} \mid \text{Z}$	001001	No
OR	Ordered	$\overline{\text{NAN}}$	000111	No
UN	Unordered	NAN	001000	No
<b>Miscellaneous Tests</b>				
F	False	False	000000	No
T	True	True	001111	No
SF	Signaling false	False	010000	Yes
ST	Signaling true	True	011111	Yes
SEQ	Signaling equal	Z	010001	Yes
SNE	Signaling not equal	$\overline{\text{Z}}$	011110	Yes

<sup>1</sup> This column refers to the value in the instruction's conditional predicate field that specifies this test.

### 4.3.6 Floating-Point Exceptions

This section describes floating-point exceptions and how they are handled. Table 4-10 lists the vector numbers related to floating-point exceptions. If the exception is taken pre-instruction, the PC contains the address of the next floating-point instruction (nextFP). If the exception is taken post-instruction, the PC contains the address of the faulting instruction (fault).

**Table 4-10. Floating-Point Exception Vectors**

Vector Number	Vector Offset	Program Counter	Assignment
48	0x0C0	Fault	Floating-point branch/set on unordered condition
49	0x0C4	NextFP or Fault	Floating-point inexact result
50	0x0C8	NextFP	Floating-point divide-by-zero
51	0x0CC	NextFP or Fault	Floating-point underflow
52	0x0D0	NextFP or Fault	Floating-point operand error
53	0x0D4	NextFP or Fault	Floating-point overflow
54	0x0D8	NextFP or Fault	Floating-point input NAN
55	0x0DC	NextFP or Fault	Floating-point input denormalized number

In addition to these vectors, attempting to execute a FRESTORE instruction with an unsupported frame value generates a format error exception (vector 14). See the FRESTORE instruction in the *PRM*.

Attempting to execute an FPU instruction with an undefined or unsupported value in the 6-bit effective address, the 3-bit source/destination specifier, or the 7-bit opcode generates a line-F emulator exception, vector 11. See Table 4-23.

### 4.3.7 Floating-Point Arithmetic Exceptions

This section describes floating-point arithmetic exceptions; Table 4-11 lists these exceptions in order of priority:

**Table 4-11. Exception Priorities**

Priority	Exception
1	Branch/set on unordered (BSUN)
2	Input Not-a-Number (INAN)
3	Input denormalized number (IDE)
4	Operand error (OPERR)
5	Overflow (OVFL)
6	Underflow (UNFL)
7	Divide-by-zero (DZ)
8	Inexact (INEX)

Most floating-point exceptions are taken when the next floating-point arithmetic instruction is encountered (this is called a pre-instruction exception). Exceptions set during a floating-point store to memory or to an integer register are taken immediately (post-instruction exception).

Note that FMOVE is considered an arithmetic instruction because the result is rounded. Only FMOVE with any destination other than a floating-point register (sometimes called FMOVE OUT) can generate post-instruction exceptions. Post-instruction exceptions never write the destination. After a post-instruction exception, processing continues with the next instruction.

A floating-point arithmetic exception becomes pending when the result of a floating-point instruction sets an FPSR[EXC] bit and the corresponding FPCR[ENABLE] bit is set. A user write to the FPSR or FPCR that causes the setting of an exception bit in FPSR[EXC] along with its corresponding exception enabled in FPCR, leaves the FPU in an exception-pending state. The corresponding exception is taken at the start of the next arithmetic instruction as a pre-instruction exception.

Executing a single instruction can generate multiple exceptions. When multiple exceptions occur with exceptions enabled for more than one exception class, the highest priority exception is reported and taken. It is up to the exception handler to check for multiple exceptions. The following multiple exceptions are possible:

- Operand error (OPERR) and inexact result (INEX)
- Overflow (OVFL) and inexact result (INEX)
- Underflow (UNFL) and inexact result (INEX)
- Divide-by-zero (DZ) and inexact result (INEX)
- Input denormalized number (IDE) and inexact result (INEX)
- Input not-a-number (INAN) and input denormalized number (IDE)

In general, all exceptions behave similarly. If the exception is disabled when the exception condition exists, no exception is taken, a default result is written to the destination (except for BSUN exception, which has no destination), and execution proceeds normally.

If an enabled exception occurs, the same default result above is written for pre-instruction exceptions but no result is written for post-instruction exceptions.

An exception handler is expected to execute FSAVE as its first floating-point instruction. This also clears FPCR, which keeps exceptions from occurring during the handler. Because the destination is overwritten for floating-point register destinations, the original floating-point destination register value is available for the handler on the FSAVE state frame. The address of the instruction that caused the exception is available in the FPIAR. When the handler is done, it should clear the appropriate FPSR exception bit on the FSAVE state frame, then execute FRESTORE. If the exception status bit is not cleared on the state frame, the same exception occurs again.

Alternatively, instead of executing FSAVE, an exception handler could simply clear appropriate FPSR exception bits, optionally alter FPCR, and then return from the exception. Note that exceptions are never taken on FMOVE to or from the status and control registers and FMOVEM to or from the floating-point data registers.

At the completion of the exception handler, the RTE instruction must be executed to return to normal instruction flow.

### 4.3.8 Branch/Set on Unordered (BSUN)

A BSUN results from performing an IEEE nonaware conditional test associated with the FBcc instruction when an unordered condition is present. Any pending floating-point exception is first handled by a pre-instruction exception, after which the conditional instruction restarts. The conditional predicate is evaluated and checked for a BSUN exception before executing the conditional instruction. A BSUN exception occurs if the conditional predicate is an IEEE non-aware branch and FPCC[NAN] is set. When this condition is detected, FPSR[BSUN] is set.

**Table 4-12. BSUN Exception Enabled/Disabled Results**

Condition	BSUN	Description
Exception disabled	0	The floating-point condition is evaluated as if it were the equivalent IEEE-aware conditional predicate. No exceptions are taken.
Exception Enabled	1	<p>The processor takes a floating-point pre-instruction exception.</p> <p>The BSUN exception is unique in that the exception is taken before the conditional predicate is evaluated. If the user BSUN exception handler fails to update the PC to the instruction after the excepting instruction when returning, the exception executes again. Any of the following actions prevent taking the exception again:</p> <ul style="list-style-type: none"> <li>• Clearing FPSR[NAN]</li> <li>• Disabling FPCR[BSUN]</li> <li>• Incrementing the stored PC in the stack bypasses the conditional instruction. This applies to situations where fall-through is desired. Note that to accurately calculate the PC increment requires knowledge of the size of the bypassed conditional instruction.</li> </ul>

### 4.3.9 Input Not-A-Number (INAN)

The INAN exception is a mechanism for handling a user-defined, non-IEEE data type. If either input operand is a NAN, FPSR[INAN] is set. By enabling this exception, the user can override the default action taken for NAN operands. Because FMOVEM, FMOVE FPCR, and FSAVE instructions do not modify status bits, they cannot generate exceptions. Therefore, these instructions are useful for manipulating INANs. See Table 4-13.

**Table 4-13. INAN Exception Enabled/Disabled Results**

Condition	INAN	Description
Exception disabled	0	If the destination data format is single- or double-precision, a NAN is generated with a mantissa of all ones and a sign of zero transferred to the destination. If the destination data format is B, W, or L, a constant of all ones is written to the destination.
Exception enabled	1	The result written to the destination is the same as the exception disabled case unless the exception occurs on a FMOVE OUT, in which case the destination is unaffected.

### 4.3.10 Input Denormalized Number (IDE)

The input denorm bit, FPCR[IDE], provides software support for denormalized operands. When the IDE exception is disabled, the operand is treated as zero, FPSR[INEX] is set, and the operation proceeds. When the IDE exception is enabled and an operand is denormalized, an IDE exception is taken but FPSR[INEX] is not set to allow the handler to set it appropriately. See Table 4-14.

Note that the FPU never generates denormalized numbers. If necessary, software can create them in the underflow exception handler.

**Table 4-14. IDE Exception Enabled/Disabled Results**

Condition	IDE	Description
Exception disabled	0	Any denormalized operand is treated as zero, FPSR[INEX] is set, and the operation proceeds.
Exception enabled	1	The result written to the destination is the same as the exception disabled case unless the exception occurs on a FMOVE OUT, in which case the destination is unaffected. FPSR[INEX] is not set to allow the handler to set it appropriately.

### 4.3.11 Operand Error (OPERR)

The operand error exception encompasses problems arising in a variety of operations, including errors too infrequent or trivial to merit a specific exceptional condition. Basically, an operand error occurs when an operation has no mathematical interpretation for the given operands. Table 4-15 lists possible operand errors. When one occurs, FPSR[OPERR] is set.

**Table 4-15. Possible Operand Errors**

Instruction	Condition Causing Operand Error
FADD	$[(+\infty) + (-\infty)]$ or $[(-\infty) + (+\infty)]$
FDIV	$(0 \div 0)$ or $(\infty \div \infty)$
FMOVE OUT (to B, W, or L)	Integer overflow, source is NAN or $\pm\infty$
FMUL	One operand is 0 and the other is $\pm\infty$
FSQRT	Source is $< 0$ or $-\infty$
FSUB	$[(+\infty) - (+\infty)]$ or $[(-\infty) - (-\infty)]$

Table 4-16 describes results when the exception is enabled and disabled.

**Table 4-16. OPERR Exception Enabled/Disabled Results**

Condition	OPERR	Description
Exception disabled	0	When the destination is a floating-point data register, the result is a double-precision NAN, with its mantissa set to all ones and the sign set to zero (positive). For a FMOVE OUT instruction with the format S or D, an OPERR exception is impossible. With the format B, W, or L, an OPERR exception is possible only on a conversion to integer overflow, or if the source is either an infinity or a NAN. On integer overflow and infinity source cases, the largest positive or negative integer that can fit in the specified destination size (B, W, or L) is stored. In the NAN source case, a constant of all ones is written to the destination.
Exception enabled	1	The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, the user OPERR handler can overwrite the default result.

### 4.3.12 Overflow (OVFL)

An overflow exception is detected for arithmetic operations in which the destination is a floating-point data register or memory when the intermediate result's exponent is greater than or equal to the maximum exponent value of the selected rounding precision. Overflow

occurs only when the destination is S- or D-precision format; overflows for other formats are handled as operand errors. At the end of any operation that could potentially overflow, the intermediate result is checked for underflow, rounded, and then checked for overflow before it is stored to the destination. If overflow occurs, FPSR[OVFL,INEX] are set.

Even if the intermediate result is small enough to be represented as a double-precision number, an overflow can occur if the magnitude of the intermediate result exceeds the range of the selected rounding precision format. See Table 4-17.

**Table 4-17. OVFL Exception Enabled/Disabled Results**

Condition	OVFL	Description
Exception disabled	0	The values stored in the destination based on the rounding mode defined in FPCR[MODE]. RN Infinity, with the sign of the intermediate result. RZ Largest magnitude number, with the sign of the intermediate result. RM For positive overflow, largest positive normalized number For negative overflow, $-\infty$ . RP For positive overflow, $+\infty$ For negative overflow, largest negative normalized number.
Exception enabled	1	The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, the user OVFL handler can overwrite the default result.

### 4.3.13 Underflow (UNFL)

An underflow exception occurs when the intermediate result of an arithmetic instruction is too small to be represented as a normalized number in a floating-point register or memory using the selected rounding precision, that is, when the intermediate result exponent is less than or equal to the minimum exponent value of the selected rounding precision. Underflow can only occur when the destination format is single or double precision. When the destination is byte, word, or longword, the conversion underflows to zero without causing an underflow or an operand error. At the end of any operation that could underflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored in the destination. FPSR[UNFL] is set if underflow occurs. If the underflow exception is disabled, FPSR[INEX] is also set.

Even if the intermediate result is large enough to be represented as a double-precision number, an underflow can occur if the magnitude of the intermediate result is too small to be represented in the selected rounding precision. Table 4-18 shows results when the exception is enabled or disabled.



**Table 4-18. UNFL Exception Enabled/Disabled Results**

Condition	UNFL	Description
Exception disabled	0	The stored result is defined below. The UNFL exception also sets FPSR[INEX] if the UNFL exception is disabled. RN Zero, with the sign of the intermediate result. RZ Zero, with the sign of the intermediate result. RM For positive underflow, + 0 For negative underflow, smallest negative normalized number. RP For positive underflow, smallest positive normalized number For negative underflow, - 0
Exception enabled	1	The result written to the destination is the same as for the exception disabled case unless the exception occurs on a FMOVE OUT, in which case the destination is unaffected. If desired, the user UNFL handler can overwrite the default result. The UNFL exception does not set FPSR[INEX] if the UNFL exception is enabled so the exception handler can set FPSR[INEX] based on results it generates.

### 4.3.14 Divide-by-Zero (DZ)

Attempting to use a zero divisor for a divide instruction causes a divide-by-zero exception. When a divide-by-zero is detected, FPSR[DZ] is set. Table 4-18 shows results when the exception is enabled or disabled.

**Table 4-19. DZ Exception Enabled/Disabled Results**

Condition	DZ	Description
Exception disabled	0	The destination floating-point data register is written with infinity with the sign set to the exclusive OR of the signs of the input operands.
Exception enabled	1	The destination floating-point data register is written as in the exception is disabled case.

### 4.3.15 Inexact Result (INEX)

An INEX exception condition exists when the infinitely precise mantissa of a floating-point intermediate result has more significant bits than can be represented exactly in the selected rounding precision or in the destination format. If this condition occurs, FPSR[INEX] is set and the infinitely-precise result is rounded according to Table 4-20.

**Table 4-20. Inexact Rounding Mode Values**

Mode	Result
RN	The representable value nearest the infinitely-precise intermediate value is the result. If the two nearest representable values are equally near, the one whose lsb is 0 (even) is the result. This is sometimes called round-to-nearest-even.
RZ	The result is the value closest to and no greater in magnitude than the infinitely-precise intermediate result. This is sometimes called chop-mode, because the effect is to clear bits to the right of the rounding point.
RM	The result is the value closest to and no greater than the infinitely-precise intermediate result (possibly $-\infty$ ).
RP	The result is the value closest to and no less than the infinitely-precise intermediate result (possibly $+\infty$ ).

FPSR[INEX] is also set for any of the following conditions:

- If an input operand is a denormalized number and the IDE exception is disabled
- An overflowed result
- An underflowed result with the underflow exception disabled

Table 4-18 shows results when the exception is enabled or disabled.

**Table 4-21. INEX Exception Enabled/Disabled Results**

Condition	INEX	Description
Exception disabled	0	The result is rounded and then written to the destination.
Exception enabled	1	The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, the user INEX handler can overwrite the default result.

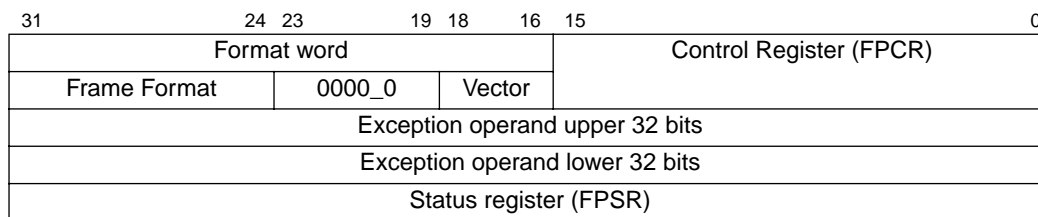
### 4.3.16 Floating-Point State Frames

Floating-point arithmetic exception handlers should have FSAVE as the first floating-point instruction; otherwise, encountering another floating-point arithmetic instruction will cause the exception to be reported again. After FSAVE executes, the handler should use FMOVEM to access floating-point data registers because it cannot generate further exceptions or change the FPSR.

Note that if no intervention is needed, instead of FSAVE, the handler can simply clear the appropriate FPCR and FPSR bits and then return from the exception.

Because the FPCR and FPSR are written in the FSAVE frame, a context switch needs only execute FSAVE and FMOVEM for data registers. The new process needs to load data registers by using a FMOVEM/FRESTORE sequence, then it can continue.

FSAVE operations always write a 4-longword floating-point state frame that holds a 64-bit exception operand. Figure 4-13 shows FSAVE frame contents.



**Figure 4-13. Floating-Point State Frame Contents**

Table 4-22 describes format word fields.

**Table 4-22. Format Word Field Descriptions**

Bits	Name	Description
31–24	Frame format	Defines the format of the frame. 0x00 Null Frame (NULL) 0x05 Idle Frame (IDLE) 0xE5 Exception Frame (EXCP)
23–19	—	Zeros
18–16	Vector	Exception vector 000 BSUN 001 INEX 010 DZ 011 UNFL 100 OPERR 101 OVFL 110 INAN 111 IDE

When FSAVE executes, the floating-point frame reflects the FPU state at the time of the FSAVE. Internally, the FPU can be in the NULL, IDLE, or EXCP states. Upon reset, the FPU is in NULL state, in which all floating-point registers contain NaNs and the FPCR, FPSR, and FPIAR contain zeros. The FPU remains in NULL state until execution of an implemented floating-point instruction (except FSAVE). At this point, the FPU transitions from NULL to an IDLE state. A FRESTORE of NULL returns the FPU to NULL state.

EXCP state is entered as a result of a floating-point exception or an unsupported data type exception. The vector field identifies exception types associated with the EXCP state. This field and the exception vector taken are determined directly from the exception control (FPCR) and status (FPSR) bits. An FSAVE instruction always clears FPCR after saving its state. Thus, after an FSAVE, a handler does not generate further floating-point exceptions unless the handler re-enables the exceptions. FRESTORE returns FPCR and FPSR to their previous state before entering the handler, as stored in the state frame. A handler could alter the state frame to restore the FPU (using FRESTORE) into a different state than that saved by using FSAVE.

Normally, an exception handler executes FSAVE, processes the exception, clears the exception bit in the FSAVE state frame status word, and executes FRESTORE. If appropriate exception bits set in the status word are not cleared, the same exception is taken again. If multiple exception bits are set in the status word, each should be processed, cleared, and restored by their respective handlers. In this way, all exceptions are processed in priority order.

If it is not necessary to handle multiple exceptions, the exception model can be simplified (after any processing) by the handler manually loading FPCR and FPSR and then discarding the state frame before executing an RTE. Given that state frames are 4 longwords, it may be quicker to discard the state frame by incrementing the address pointer (often the system stack pointer, A7) by 16.

The exception operand, contained in longwords 2 and 3 of the FSAVE frame, is always the value of the destination operand before the operation which caused the exception commenced. Thus, for dyadic register-to-register operations, the exception operand contains the value of the destination register before it was overwritten by the operation which caused the exception. This operand can be retrieved by an exception handler that needs both original operands in order to process the exception.

## 4.4 Instructions

This section includes an instruction set summary, execution times, and differences between ColdFire and 68K FPU programming models. For detailed instruction descriptions, see the *PRM*.

### 4.4.1 Floating-Point Instruction Overview

ColdFire instructions are 16, 32, or 48 bits long. The general definition of a floating-point operation and effective addressing mode require 32 bits; some addressing modes require another 16-bit extension word. Table 4-23 shows the minimum size instruction formats. The first word is the opword; the second is extension word 1.

**Table 4-23. Floating-Point Instruction Formats**

Mnemonic	Instruction Code																							
FABS	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FADD	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FBcc	1	1	1	1	0	0	1	0	1	SZ	COND PREDICATE		16b displacement or MS Word of 32b											
	LS Word of 32b Displacement																							
FCMP	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	0	1	1	1	0	0	0
FDIV	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FINT	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	0	0	0	0	0	0	1
FINTRZ	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	0	0	0	0	0	1	1
FMOVE	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	1	1	DEST FMT		SRC REG		0 0 0 0 0 0 0				
	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	1	0	dr	REG SEL		0 0 0		0 0 0 0 0 0 0 0				
FMOVEM	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	1	1	dr	1 0 0		0 0		REGISTER LIST				
FMUL	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FNEG	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FNOP	1	1	1	1	0	0	1	0	1	0	0 0 0		0 0 0		0 0 0		0 0 0		0 0 0 0 0 0 0 0					
FRESTORE	1	1	1	1	0	0	1	1	0	1	EA MODE	EA REG												
FSAVE	1	1	1	1	0	0	1	1	0	0	EA MODE	EA REG												
FSQRT	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						

**Table 4-23. Floating-Point Instruction Formats (Continued)**

Mnemonic	Instruction Code																							
FSUB	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	OPMODE						
FTST	1	1	1	1	0	0	1	0	0	0	EA MODE	EA REG	0	R/M	0	SRC SPEC	DEST REG	0	1	1	1	0	1	0

Table 4-24 defines the terminology used in Table 4-23.

**Table 4-24. Instruction Format Terminology**

Term	Definition
Instructions	Instructions appear in memory as sequential, 16-bit values, and are read in the above table left to right. An instruction can have from 1 to 3 16-bit words. A shaded block indicates this word is never used and is not present.
EA MODE EA REG	Defines the effective address for an operand located external to the FPU. For most FPU instructions, this field defines the location of an external source operand; for FP store operations, it specifies the destination location.
R/M	If R/M = 0, an FPU data register is one source operand, otherwise the source operand is specified by the EA {MODE, REG} fields.
SRC SPEC	Defines the format (byte, word, longword, single-, or double-precision) of an external operand.
DEST REG	Specifies the destination FPU data register.
COND PREDICATE	Defines the condition to be evaluated (EQ, NE, and so on) during the execution of the FPU conditional branch instruction.
OPMODE	Defines the exact operation to be performed by the FPU.
SZ	Defines the length of the PC-relative displacement for the FPU conditional branch instruction. If SZ = 0, the displacement is 16 bits, otherwise a 32-bit displacement is used.
dr	Specifies direction of the MOVE transfer. As a 0, it moves from memory to the FP; as 1, it moves from the FP to memory.
REGISTER LIST	Defines FPU data registers to be moved during the execution of the FMOVEM instruction.
REG SEL	Indicates the FPU control register to be moved during execution of an FMOVE control register instruction.

## 4.4.2 Floating-Point Instruction Execution Times

Table 4-25 shows the ColdFire execution times for the floating-point instructions in terms of processor core clock cycles. Each timing entry is presented as  $C(r/w)$ .

- $C$  = The number of processor clock cycles including all applicable operand reads and writes plus all internal core cycles required to complete instruction execution
- $r$  = The number of operand reads
- $w$  = The number of operand writes

### NOTE:

Timing assumptions are the same as those for the ColdFire ISA. See the *ColdFire Microprocessor Family Programmer's Reference Manual*.

**Table 4-25. Floating-Point Instruction Execution Times<sup>1, 2, 3</sup>**

Opcode	Format	Effective Address <ea>						
		FPn	Dn	(An)	(An)+	-(An)	(d <sub>16</sub> ,An)	(d <sub>16</sub> ,PC)
FABS	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
FADD	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FBcc	<label>	—	—	—	—	—	—	2(0/0) if correct, 9(0/0) if incorrect
FCMP	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FDIV	<ea>y,FPx	23(0/0)	23(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)
FINT	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FINTRZ	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FMOVE	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
	FPy,<ea>x	—	2(0/1)	2(0/1)	2(0/1)	2(0/1)	2(0/1)	—
	<ea>y,FP*R	—	6(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)
	FP*R,<ea>x	—	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	—
FMOVEM <sup>4</sup>	<ea>y,#list	—	—	2n(2n/0)	—	—	2n(2n/0)	2n(2n/0)
	#list,<ea>x	—	—	1+2n(0/2n)	—	—	1+2n(0/2n)	—
FMUL	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FNEG	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
FNOP		—	—	—	—	—	—	2(0/0)
FRESTORE	<ea>y	—	—	6(4/0)	—	—	6(4/0)	6(4/0)
FSAVE	<ea>x	—	—	7(0/4)	—	—	7(0/4)	—
FSQRT	<ea>y,FPx	56(0/0)	56(0/0)	56(1/0)	56(1/0)	56(1/0)	56(1/0)	56(1/0)
FSUB	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
FTST	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)

- <sup>1</sup> Add 1(1/0) for an external read operand of double-precision format for all instructions except FMOVEM, and 1(0/1) for FMOVE FPy,<ea>x when the destination is double-precision.
- <sup>2</sup> If the external operand is an integer format (byte, word, longword), there is a 4 cycle conversion time which must be added to the basic execution time.
- <sup>3</sup> If any exceptions are enabled, the execution time for FMOVE FPy,<ea>x increases by one cycle. If the BSUN exception is enabled, the execution time for FBcc increases by one cycle.
- <sup>4</sup> For FMOVEM, *n* refers to the number of registers being moved.

The ColdFire architecture supports concurrent execution of integer and floating-point instructions. The latencies in this table define the execution time needed by the FPU. After a multi-cycle FPU instruction is issued, subsequent integer instructions can execute concurrently with the FPU execution. For this sequence, the floating-point instruction occupies only one OEP cycle.

### 4.4.3 Key Differences between ColdFire and MC680x0 FPU Programming Models

This section is intended for compiler developers and developers porting assembly language routines from 68K to ColdFire. It highlights major differences between the ColdFire FPU instruction set architecture (ISA) and the equivalent 68K family ISA, using the MC68060 as the reference. The internal FPU datapath width is the most obvious difference. ColdFire uses 64-bit double-precision and the 68K Family uses 80-bit extended precision. Other differences pertain to supported addressing modes, both across all FPU instructions as well as specific opcodes. Table 4-26 lists key differences. Because all ColdFire implementations support instruction sizes of 48 bits or less, 68K operations requiring larger instruction lengths cannot be supported.

**Table 4-26. Key Programming Model Differences**

Feature	68K	ColdFire
Internal datapath width	80 bits	64 bits
Support for fpGEN d <sub>8</sub> (An,Xi),FPx	Yes	No
Support for fpGEN xxx.{w,l},FPx	Yes	No
Support for fpGEN d <sub>8</sub> (PC,Xi),FPx	Yes	No
Support for fpGEN #xxx,FPx	Yes	No
Support for fmovem (Ay)+,#list	Yes	No
Support for fmovem #list,-(Ax)	Yes	No
Support for fmovem FP Control Registers	Yes	No

Some differences affect function activation and return. 68K subroutines typically began with FMOVEM #list,-(a7) to save registers on the system stack, with each register occupying 3 longwords. In ColdFire, each register occupies 2 longwords and the stack pointer must be adjusted before the FMOVEM instruction. A similar sequence generally occurs at the end of the function, preparing to return control to the calling routine.

The examples in Table 4-27, Table 4-28, and Table 4-29 show a 68K operation and the equivalent ColdFire sequence.

**Table 4-27. 68K/ColdFire Operation Sequence 1<sup>1</sup>**

68K	ColdFire Equivalent
fmovem.x #list,-(a7)	lea -8*n(a7),a7;allocate stack space fmovem.d #list,(a7) ;save FPU registers
fmovem.x (a7)+,#list	fmovem.d (a7),#list ;restore FPU registers lea 8*n(a7),a7 ;deallocate stack space

<sup>1</sup> n is the number of FP registers to be saved/restored.

If the subroutine includes LINK and UNLK instructions, the stack space needed for FPU register storage can be factored into these operations and LEA instructions are not required.

The 68K FPU supports loads and stores of multiple control registers (FPCR, FPSR, and FPIAR) with one instruction. For ColdFire, only one can be moved at a time.

For instructions that require an unsupported addressing mode, the operand address can be formed with a LEA instruction immediately before the FPU operation. See Table 4-28.

**Table 4-28. 68K/ColdFire Operation Sequence 2**

68K	ColdFire Equivalent
fadd.s label,fp2	lea label,a0;form pointer to data fadd.s (a0),fp2
fmul.d (d8,a1,d7),fp5	lea (d8,a1,d7),a0;form pointer to data fmul.d (a0),fp5
fcmp.l (d8,pc,d2),fp3	lea (d8,pc,d2),a0;form pointer to data fcmp.l (a0),fp3

The 68K FPU allows floating-point instructions to directly specify immediate values; the ColdFire FPU does not support these types of immediate constants. It is recommended that floating-point immediate values be moved into a table of constants that can be referenced using PC-relative addressing or as an offset from another address pointer. See Table 4-29.

**Table 4-29. 68K/ColdFire Operation Sequence 3**

68K	ColdFire Equivalent
fadd.l #imm1,fp3	fadd.l (imm1_label,pc),fp3
fsub.s #imm2,fp4	fsub.s (imm2_label,pc),fp3



**Table 4-29. 68K/ColdFire Operation Sequence 3 (Continued)**

68K	ColdFire Equivalent
fdiv.d #imm3,fp5	<pre> fdiv.d (imm3_label,pc),fp3 align 4 imm1_label:     long imm1 ;integer longword imm2_label:     long imm2 ;single-precision imm3_label:     long imm3_upper,     imm3_lower ;double-precision </pre>

Finally, ColdFire and the 68K differ in how exceptions are made pending. In the ColdFire exception model, asserting both an FPSR exception indicator bit and the corresponding FPCR enable bit makes an exception pending. Thus, a pending exception state can be created by loading FPSR and/or FPCR. On the 68K, this type of pending exception is not possible.

Analysis of compiled floating-point applications indicates these differences account for most of the changes between 68K-compatible text and the equivalent ColdFire program.



# Chapter 5

## Enhanced Multiply-Accumulate Unit (EMAC)

This chapter describes the functionality, microarchitecture, and performance of the enhanced multiply-accumulate (EMAC) unit in the ColdFire family of processors.

### 5.1 Multiply-Accumulate Unit

This document details the functionality, microarchitecture, and performance of the hardware multiply-accumulate (MAC) unit in the ColdFire family of processors.

Motorola has incorporated a RISC-based processor design for peak performance and a simplified version of the M68000 Family variable-length instruction set for maximum code density. The result is a family of 32-bit microprocessors optimized for embedded applications requiring high performance in a small core size.

The ColdFire performance road map defines a series of microarchitecture versions that couple with improved process technology to offer increasing levels of performance.

The MAC design centers on the notion of providing a limited set of DSP operations currently used in embedded code, while supporting the integer multiply instructions of the baseline ColdFire architecture.

The MAC provides functionality in three related areas:

- Signed and unsigned integer multiplies
- Multiply-accumulate operations supporting signed and unsigned integer operands as well as signed, fixed-point, fractional operands
- Miscellaneous register operations

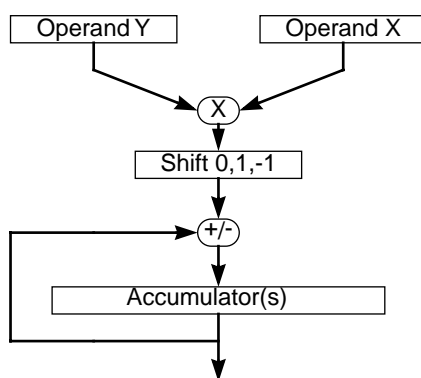
The ColdFire family supports two MAC implementations with different performance levels and capabilities for differing silicon costs. The original MAC is a three-stage execution pipeline, optimized for 16-bit operands and featuring a 16x16 multiply array with a single 32-bit accumulator. The enhanced MAC (EMAC) features a four-stage pipeline optimized for 32-bit operands, with a fully pipelined 32x32 multiply array and four 48-bit accumulators. Either can be attached to any version (V2, V3, or V4) ColdFire core as dictated by application requirements.

The first ColdFire MAC supported signed and unsigned integer operands and was optimized for 16x16 operations, based on a variety of target applications including servo control and image compression. As ColdFire-based systems proliferated, the desire for more precision on input operands increased. The result was an improved ColdFire MAC with user-programmable control to optionally enable use of fractional input operands.

EMAC improvements target three primary areas:

- Improved performance of 32x32 multiply operations.
- Addition of three more accumulators to minimize MAC pipeline stalls caused by exchanges between the accumulator and the pipeline's general-purpose registers.
- A 48-bit accumulation data path to allow the use of a 40-bit product plus the addition of 8 extension bits to increase the dynamic number range when implementing signal processing algorithms.

The three areas of functionality are addressed in detail in following sections. The logic required to support this functionality is contained in a MAC module, as shown in Figure 5-1.



**Figure 5-1. Multiply-Accumulate Functionality Diagram**

## 5.2 An Introduction to the MAC

The MAC is an extension of the basic multiplier found in most microprocessors. It is implemented either in hardware or in an iterative routine within an architecture and supports signal processing algorithms in an acceptable number of cycles, given application constraints. For example, small digital filters can tolerate some variance in an algorithm's execution time, but larger, more complicated algorithms such as orthogonal transforms may have more demanding speed requirements beyond the scope of any processor architecture and may require full DSP implementation.

The 68K/ColdFire architecture was not designed for high-speed signal processing, and a large DSP engine would be excessive in an embedded environment. In striking a balance between speed, size, and functionality, the ColdFire MAC is optimized for a small set of operations that involve multiplication and cumulative additions. Specifically, the multiplier

array is optimized for single-cycle pipelined operations with a possible accumulation after product generation. This functionality is common in many signal processing applications. The ColdFire core architecture also has been modified to allow an operand to be fetched in parallel with a multiply, increasing overall performance for certain DSP operations.

Consider a typical filtering operation where the filter is defined as in Figure 5-2.

$$y(i) = \sum_{k=1}^{N-1} a(k)y(i-k) + \sum_{k=0}^{N-1} b(k)x(i-k)$$

**Figure 5-2. Infinite Impulse Response (IIR) Filter**

Here, the output  $y(i)$  is determined by past output values or past input values. This is the general form of an infinite impulse response (IIR) filter. A finite impulse response (FIR) filter can be obtained by setting coefficients  $a(k)$  to zero. In either case, the operations involved in computing such a filter are multiplies and product summing. To show this point, reduce the above equation to a simple, four-tap FIR filter, shown in Figure 5-3, in which the accumulated sum is a sum of past data values and coefficients.

$$y(i) = \sum_{k=0}^3 b(k)x(i-k) = b(0)x(i) + b(1)x(i-1) + b(2)x(i-2) + b(3)x(i-3)$$

**Figure 5-3. Four-Tap FIR Filter**

## 5.3 General Operation

The MAC speeds execution of ColdFire integer multiply instructions (MULS and MULU) and provides additional functionality for multiply-accumulate operations. By executing MULS and MULU in the MAC, execution times are minimized and deterministic compared to the 2-bit/cycle algorithm with early termination that the OEP normally uses if no MAC hardware is present.

The added MAC instructions to the ColdFire ISA provide for the multiplication of two numbers, followed by the addition or subtraction of the product to/from the value in an accumulator. Optionally, the product may be shifted left or right by 1 bit before addition or subtraction. Hardware support for saturation arithmetic can be enabled to minimize software overhead when dealing with potential overflow conditions. Multiply-accumulate operations support 16- or 32-bit input operands of the following formats:

- Signed integers
- Unsigned integers
- Signed, fixed-point, fractional numbers

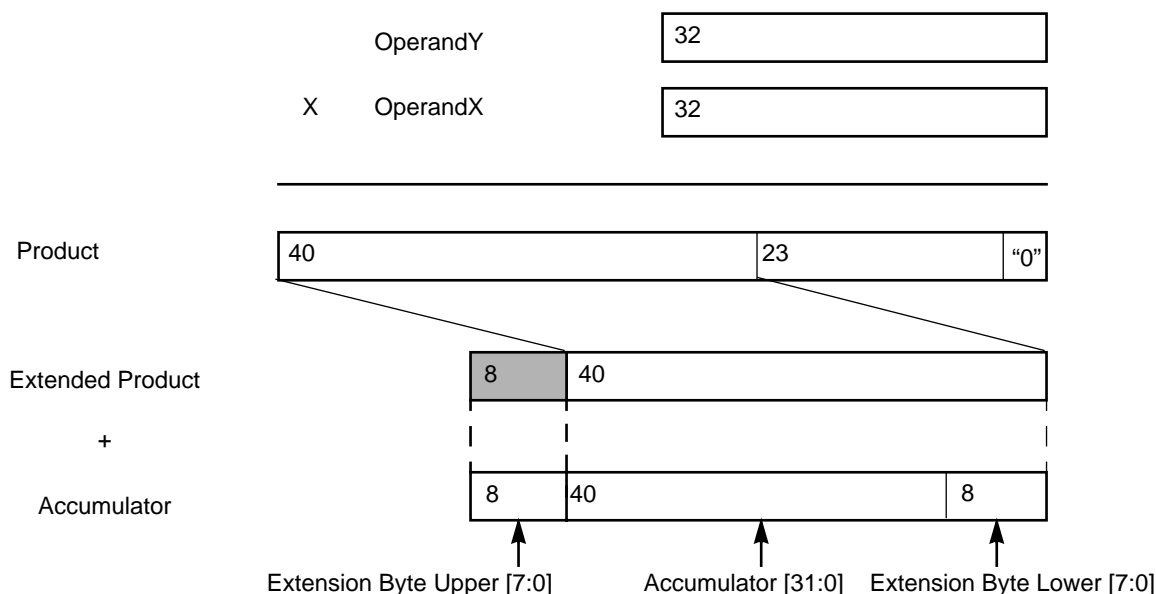
The EMAC is optimized for single-cycle, pipelined 32x32 multiplications. For word- and

## General Operation

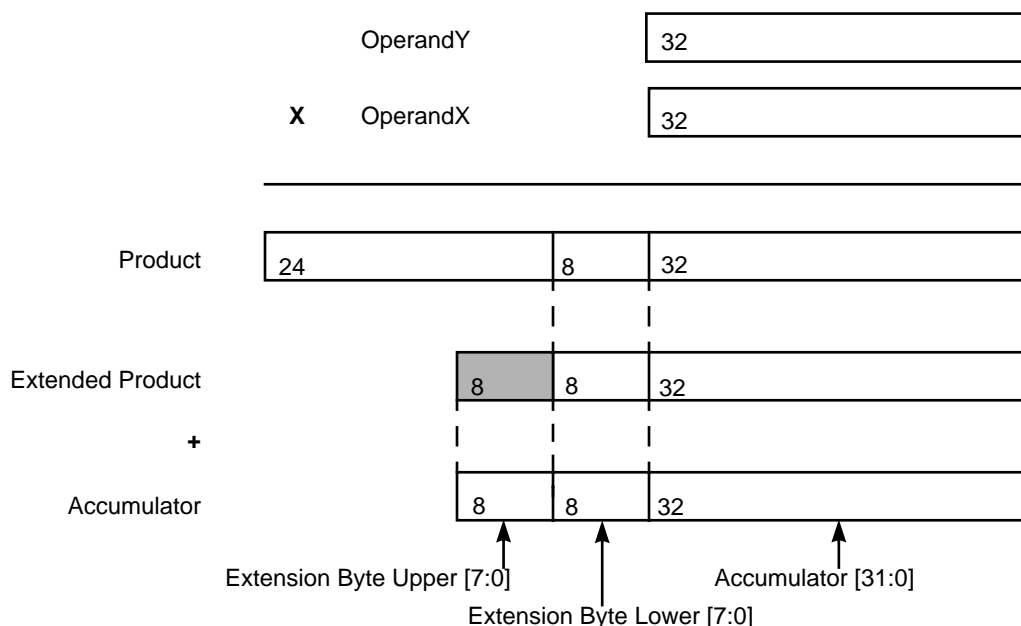
longword-sized integer input operands, the low-order 40 bits of the product are formed and used with the destination accumulator. For fractional operands, the entire 64-bit product is calculated and either truncated or rounded to the most-significant 40-bit result using the round-to-nearest (even) method before it is combined with the destination accumulator.

For all operations, the resulting 40-bit product is extended to a 48-bit value (using sign-extension for signed integer and fractional operands, zero-fill for unsigned integer operands) before being combined with the 48-bit destination accumulator.

Figure 5-4 and Figure 5-5 show relative alignment of input operands, the full 64-bit product, the resulting 40-bit product used for accumulation, and 48-bit accumulator formats.



**Figure 5-4. Fractional Alignment**



**Figure 5-5. Signed and Unsigned Integer Alignment**

Thus, the 48-bit accumulator definition is a function of the EMAC operating mode. Given that each 48-bit accumulator is the concatenation of 16-bit ACCextx contents and 32-bit ACCx contents, the specific definitions are as follows:

```
if MACSR[6:5] == 00/* signed integer mode */
    Complete Accumulator[47:0] = {ACCextx[15:0], ACCx[31:0]}
if MACSR[6:5] == -1/* signed fractional mode */
    Complete Accumulator [47:0] = {ACCextx[15:8], ACCx[31:0], ACCextx[7:0]}
if MACSR[6:5] == 10/* unsigned integer mode */
    Complete Accumulator[47:0] = {ACCextx[15:0], ACCx[31:0]}
```

The four accumulators are represented as an array, ACCx, where x selects the register.

Although the multiplier array is implemented in a four-stage pipeline, all arithmetic MAC instructions have an effective issue rate of 1 cycle, regardless of input operand size or type.

All arithmetic operations use register-based input operands, and summed values are stored internally in an accumulator. Thus, an additional move instruction is needed to store data in a general-purpose register. One feature new to MAC instructions is the ability to choose the upper or lower word of a register as a 16-bit input operand. This is useful in filtering operations if one data register is loaded with the input data and another is loaded with the coefficient. Two 16-bit multiply accumulates can be performed without fetching additional operands between instructions by alternating the word choice during the calculations.

The EMAC has four accumulator registers versus the MAC's single accumulator. The additional registers improve the performance of some algorithms by minimizing pipeline stalls needed to store an accumulator value back to general-purpose registers. Many algorithms require multiple calculations on a given data set. By applying different accumulators to these calculations, it is often possible to store one accumulator without any stalls while performing operations involving a different destination accumulator.

## Memory Map/Register Set

The need to move large amounts of data presents an obstacle to obtaining high throughput rates in DSP engines. New and existing ColdFire instructions can accommodate these requirements. A MOVEM instruction can move large blocks of data efficiently by generating line-sized burst references. The ability to simultaneously load an operand from memory into a register and execute a MAC instruction makes some DSP operations such as filtering and convolution more manageable.

The programming model includes a 16-bit mask register (MASK), which can optionally be used to generate an operand address during MAC + MOVE instructions. The application of this register with auto-increment addressing mode supports efficient implementation of circular data queues for memory operands.

The additional MACSR contains a 4-bit operational mode field and condition flags. Operational mode bits control the overflow/saturation mode, whether operands are signed or unsigned, whether operands are treated as integers or fractions, and how rounding is performed. Negative, zero, and multiple overflow condition flags are also provided.

## 5.4 Memory Map/Register Set

The EMAC provides the following program-visible registers:

- Four 32-bit accumulators (ACCx = ACC0, ACC1, ACC2, and ACC3)
- Eight 8-bit accumulator extensions (two per accumulator), packaged as two 32-bit values for load and store operations (ACCext01 and ACCext23)
- One 16-bit mask register (MASK)
- One 32-bit MAC status register (MACSR) including four indicator bits signaling product or accumulation overflow (one for each accumulator: PAV0–PAV3)

These registers are shown in Figure 5-6.

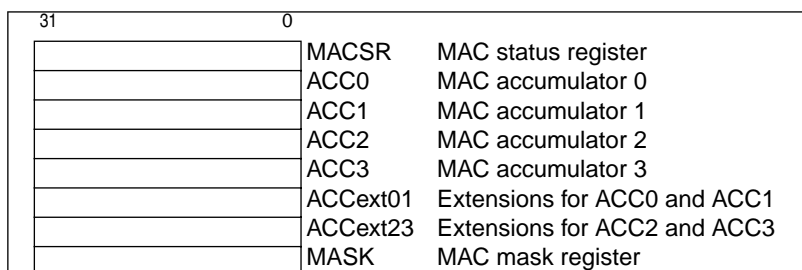


Figure 5-6. EMAC Register Set

### 5.4.1 MAC Status Register (MACSR)

MACSR functionality is organized as follows:

- MACSR[11–8] contains one product/accumulation overflow flag per accumulator.
- MACSR[7–4] defines the operating configuration of the MAC unit.



- MACSR[3–0] contains indicator flags from the last MAC instruction execution.

Bit	31	12	11	10	9	8	7	6	5	4	3	2	1	0		
					Prod/acc overflow flags				Operational Mode				PAV			
Field	—				PAV3	PAV2	PAV1	PAV0	OMC	S/U	F/I	R/T	N	Z	V	EV
Reset	All zeros															
R/W	R/W															

**Figure 5-7. MAC Status Register (MACSR)**

Table 5-1 describes MACSR fields.

**Table 5-1. MACSR Field Descriptions**

Bits	Name	Description
31–12	—	Reserved, should be cleared
11–8	PAVx	Product/accumulation overflow flags. Contains four flags, one per accumulator, indicating if past MAC or MSAC instructions generated an overflow during product calculation or from the 48-bit accumulation. When a MAC or MSAC instruction is executed, the PAVx flag associated with the destination accumulator is used to form the general overflow flag, MACSR[V]. Once set, each flag remains set until V is cleared by a MOV.L , MACSR instruction or the accumulator is loaded directly.

Table 5-1. MACSR Field Descriptions (Continued)

Bits	Name	Description
7–4		<b>Operational Mode Field</b>
7	OMC	Overflow/saturation mode. Used to enable or disable saturation mode on overflow. If set, the accumulator is set to the appropriate constant on any operation which overflows the accumulator. Once saturated, the accumulator remains unaffected by any other MAC or MSAC instructions until either the overflow bit is cleared or the accumulator is directly loaded.
6	S/U	Signed/unsigned operations. In integer mode: S/U determines whether operations performed are signed or unsigned. It also determines the accumulator value during saturation, if enabled. 0 Signed numbers. On overflow, if OMC is enabled, an accumulator saturates to the most positive (0x7FFF_FFFF) or the most negative (0x8000_0000) number, depending on both the instruction and the value of the product that overflowed. 1 Unsigned numbers. On overflow, if OMC is enabled, an accumulator saturates to the smallest value (0x0000_0000) or the largest value (0xFFFF_FFFF), depending on the instruction. In fractional mode: S/U controls rounding while storing an accumulator to a general-purpose register. 0 Move accumulator without rounding to a 16-bit value. Accumulator is moved to a general-purpose register as a 32-bit value. 1 The accumulator is rounded to a 16-bit value using the round-to-nearest (even) method when it is moved to a general-purpose register. See Section 5.4.1.1, “Fractional Operation Mode.” The resulting 16-bit value is stored in the lower word of the destination register. The upper word is zero-filled. The accumulator value is not affected by this rounding procedure.
5	F/I	Fraction/integer mode Determines whether input operands are treated as fractions or integers. 0 Integers can be represented in either signed or unsigned notation, depending on the value of S/U. 1 Fractions are represented in signed, fixed-point, two's complement notation. Values range from -1 to $1 - 2^{-15}$ for 16-bit fractions and -1 to $1 - 2^{-31}$ for 32-bit fractions. See Table 5-2.
4	R/T	Round/truncate mode. Controls the rounding procedure used with fractional MAC, MOV.L ACCx,Rx, or MSAC.L instructions. 0 Truncate. The product's lsbs are dropped before it is combined with the accumulator. Additionally, when a store accumulator instruction is executed (MOV.L ACCx,Rx), the 8 lsbs of the 48-bit accumulator logic are simply truncated. 1 Round-to-nearest (even). The 64-bit product of two 32-bit, fractional operands is rounded to the nearest 40-bit value. If the low-order 24 bits equal 0x80_0000, the upper 40 bits are rounded to the nearest even (lsb = 0) value. See Section 5.4.1.1, “Fractional Operation Mode.” Additionally, when a store accumulator instruction is executed (MOV.L ACCx,Rx), the lsbs of the 48-bit accumulator logic are used to round the resulting 16- or 32-bit value. If MACSR[S/U] = 0 and MACSR[R/T] = 1, the low-order 8 bits are used to round the resulting 32-bit fraction. If MACSR[S/U] = 1, the low-order 24 bits are used to round the resulting 16-bit fraction.

**Table 5-1. MACSR Field Descriptions (Continued)**

Bits	Name	Description	
3–0	—	<b>Flags</b>	
3		N	Negative. Set if the msb of the result is set, cleared otherwise. N is affected only by MAC, MSAC, and load operations and not by MULS and MULU instructions.
2		Z	Zero. Set if the result equals zero, cleared otherwise. This bit is affected only by MAC, MSAC, and load operations and not by MULS and MULU instructions.
1		V	Overflow. Set if an arithmetic overflow occurs on a MAC or MSAC instruction indicating that the result cannot be represented in the limited width of the EMAC. V is set only if a product overflow occurs or the accumulation overflows the 48-bit structure. V is evaluated on each MAC or MSAC operation and uses the appropriate PAVx flag in the next-state V evaluation.
0		EV	Extension overflow. Signals that the last MAC or MSAC instruction overflowed the 32 lsbs in integer mode or the 40 lsbs in fractional mode of the destination accumulator. However, the result is still accurately represented in the combined 48-bit accumulator structure. Although an overflow has occurred, the correct result, sign, and magnitude are contained in the 48-bit accumulator. Subsequent MAC or MSAC operations may return the accumulator to a valid 32/40-bit result.

Table 5-2 summarizes the interaction of the S/U, F/I, and R/T control bits (MACSR[6–4]).

**Table 5-2. Summary of S/U, F/I, and R/T Control Bits**

S/U	F/I	R/T	Operational Modes
0	0	x	Signed, integer
0	1	0	Signed, fractional Truncate on MAC.L and MSAC.L No round on accumulator stores
0	1	1	Signed, fractional, Round on MAC.L and MSAC.L Round-to-32-bits on accumulator stores
1	0	x	Unsigned, integer
1	1	0	Signed, fractional, Truncate on MAC.L and MSAC.L Round-to-16-bits on accumulator stores
1	1	1	Signed, fractional, Round on MAC.L and MSAC.L, Round-to-16-bits on accumulator stores

### 5.4.1.1 Fractional Operation Mode

This section describes behavior when fractional mode is used (MACSR[F/I] is set).

#### 5.4.1.1.1 Rounding

While the processor is in fractional mode, there are two operations during which rounding can occur.

- A store accumulator instruction is executed (MOV.L ACCx,Rx). The lsbs of the 48-bit accumulator logic are used to round the resulting 16- or 32-bit value. If MACSR[S/U] = 0, the low-order 8 bits are used to round the resulting 32-bit fraction. If MACSR[S/U] = 1, the low-order 24 bits are used to round the resulting 16-bit fraction.
- A MAC (or MSAC) instruction with 32-bit operands is executed. If MACSR[R/T] is zero, multiplying two 32-bit numbers creates a 64-bit product that is truncated to the upper 40-bits; otherwise, it is rounded using round-to-nearest (even) method.

To understand the round-to-nearest-even method, consider the following example involving the rounding of a 32-bit number, R0, to a 16-bit number. Using this method, the 32-bit number is rounded to the closest 16-bit number possible. Let the high-order 16 bits of R0 be named R0.U and the low-order 16 bits be R0.L.

- If R0.L is less than 0x8000, the result is truncated to the value of R0.U.
- If R0.L is greater than 0x8000, the upper word is incremented (rounded up).
- If R0.L is 0x8000, R0 is half-way between two 16-bit numbers. In this case, rounding is based on the lsb of R0.U, so the result is always even (lsb = 0).
  - If the lsb of R0.U = 1 and R0.L = 0x8000, the number is rounded up.
  - If the lsb of R0.U = 0 and R0.L = 0x8000, the number is rounded down.

This method minimizes rounding bias and creates as statistically correct an answer as possible.

The rounding algorithm is summarized in the following pseudocode:

```
if R0.L < 0x8000
    then Result = R0.U
    else if R0.L > 0x8000
        then Result = R0.U + 1
        else if LSB of R0.U = 0          /* R0.L = 0x8000 */
            then Result = R0.U
            else Result = R0.U + 1
```

The round-to-nearest-even technique is also known as convergent rounding.

### 5.4.1.1.2 Saving and Restoring the EMAC Programming Model

The presence of rounding logic in the output datapath of the EMAC requires that special care be taken during the save and restore of the EMAC programming model. In particular, any result rounding modes must be disabled during the save/restore process so the exact bit-wise contents of the EMAC registers are accessed. Consider the following memory structure containing the EMAC programming model:

```
struct macState {
    int acc0;
    int acc1;
    int acc2;
    int acc3;
    int accext01;
    int accext02;
    int mask;
```

```

    int macsr;
} macState;

```

The following assembly language routine shows the proper sequence for a correct EMAC state save. This code assumes all Dn and An registers are available for use and the memory location of the state save is defined by A7.

```

EMAC_state_save:
    move.l    macsr,d7      ; save the macsr
    clr.l     d0            ; zero the register to ...
    move.l    d0,macsr      ; disable rounding in the macsr
    move.l    acc0,d0       ; save the accumulators
    move.l    acc1,d1
    move.l    acc2,d2
    move.l    acc3,d3
    move.l    accext01,d4    ; save the accumulator extensions
    move.l    accext23,d5
    move.l    mask,d6        ; save the address mask
    movem.l   #0x00ff,(a7)  ; move the state to memory

```

The following code performs the EMAC state restore:

```

EMAC_state_restore:movem.l (a7),#0x00ff; restore the state from memory
    move.l    #0,macsr      ; disable rounding in the macsr
    move.l    d0,acc0       ; restore the accumulators
    move.l    d1,acc1
    move.l    d2,acc2
    move.l    d3,acc3
    move.l    d4,accext01    ; restore the accumulator extensions
    move.l    d5,accext23
    move.l    d6,mask        ; restore the address mask
    move.l    d7,macsr       ; restore the macsr

```

By executing this type of sequence, the exact state of the EMAC programming model can be correctly saved and restored.

#### 5.4.1.1.3 MULS/MULU

MULS and MULU are unaffected by fractional mode operation; operands are still assumed to be integers.

#### 5.4.1.1.4 Scale Factor in MAC or MSAC instructions

The scale factor is ignored while the MAC is in fractional mode.

### 5.4.2 Mask Register (MASK)

The 32-bit mask register (MASK) implements the low-order 16 bits, to minimize complications about loading and storing only 16 bits and the associated alignment requirements. When the MASK is loaded, the low-order 16 bits of the source operand are actually loaded into the register. When it is stored, the upper 16 bits are forced to all ones.

This register performs a simple AND with the address. That is, the processor calculates the normal operand address and, if enabled, that address is then ANDed with {0xFFFF, MASK[15:0]} to form the final address. So, MASK register bits are cleared, constraining the address to a certain region. This is used primarily to implement circular queues in conjunction with the (Ay) + addressing mode.

## EMAC Instruction Set Summary

This feature minimizes the addressing support required for filtering, convolution, or any routine that implements a data array as a circular queue. For MAC + MOVE operations, the MASK contents can optionally be included in all memory effective address calculations. The syntax is as follows:

MAC.sz Ry,RxSF,<ea>y&,Rw

The & operator enables the use of MASK and causes bit 5 of the extension word to be set. The exact algorithm for the use of MASK is as follows:

```
if extension word, bit [5] = 1, the MASK bit, then
    if <ea> = (An)
        oa = An & {0xFFFF, MASK}

    if <ea> = (An)+
        oa = An
        An = (An + 4) & {0xFFFF, MASK}

    if <ea> = -(An)
        oa = (An - 4) & {0xFFFF, MASK}
        An = (An - 4) & {0xFFFF, MASK}

    if <ea> = (d16,An)
        oa = (An + se_d16) & {0xFFFF0x, MASK}
```

Here, oa is the calculated operand address and se\_d16 is a sign-extended 16-bit displacement. For auto-addressing modes of post-increment and pre-decrement, the calculation of the updated An value is also shown.

Use of the post-increment addressing mode, (An)+ with MASK is suggested for circular queue implementations.

## 5.5 EMAC Instruction Set Summary

Figure 5-3 summarizes EMAC unit instructions.

**Table 5-3. EMAC Instruction Summary**

Command	Mnemonic	Description
Multiply Signed	MULS <ea>y,Dx	Multiplies two signed operands yielding a signed result
Multiply Unsigned	MULU <ea>y,Dx	Multiplies two unsigned operands yielding an unsigned result
Multiply Accumulate	MAC Ry,RxSF,ACCx MSAC Ry,RxSF,ACCx	Multiplies two operands and adds/subtracts the product to/from an accumulator
Multiply Accumulate with Load	MAC Ry,Rx,<ea>y,Rw,ACCx MSAC Ry,Rx,<ea>y,Rw,ACCx	Multiplies two operands and combines the product to an accumulator while loading a register with the memory operand
Load Accumulator	MOV.L {Ry,#imm},ACCx	Loads an accumulator with a 32-bit operand
Store Accumulator	MOV.L ACCx,Rx	Writes the contents of an accumulator to a CPU register
Copy Accumulator	MOV.L ACCy,ACCx	Copies a 48-bit accumulator
Load MACSR	MOV.L {Ry,#imm},MACSR	Writes a value to MACSR
Store MACSR	MOV.L MACSR,Rx	Write the contents of MACSR to a CPU register

**Table 5-3. EMAC Instruction Summary**

Command	Mnemonic	Description
Store MACSR to CCR	MOV.L MACSR,CCR	Write the contents of MACSR to the CCR
Load MAC Mask Reg	MOV.L {Ry,#imm},MASK	Writes a value to the MASK register
Store MAC Mask Reg	MOV.L MASK,Rx	Writes the contents of the MASK to a CPU register
Load AccExtensions01	MOV.L {Ry,#imm},ACCext01	Loads the accumulator 0,1 extension bytes with a 32-bit operand
Load AccExtensions23	MOV.L {Ry,#imm},ACCext23	Loads the accumulator 2,3 extension bytes with a 32-bit operand
Store AccExtensions01	MOV.L ACCext01,Rx	Writes the contents of accumulator 0,1 extension bytes into a CPU register
Store AccExtensions23	MOV.L ACCext23,Rx	Writes the contents of accumulator 2,3 extension bytes into a CPU register

### 5.5.1 Data Representation

MACSR[6–5] selects one of the following three modes, where each mode defines a unique operand type.

- Two's complement signed integer: In this format, an N-bit operand value lies in the range  $-2^{(N-1)} \leq \text{operand} \leq 2^{(N-1)} - 1$ . The binary point is right of the lsb.
- Unsigned integer: In this format, an N-bit operand value lies in the range  $0 \leq \text{operand} \leq 2^N - 1$ . The binary point is right of the lsb.
- Two's complement, signed fractional: In an N-bit number, the first bit is the sign bit. The remaining bits signify the first N-1 bits after the binary point. Given an N-bit number,  $a_{N-1}a_{N-2}a_{N-3} \dots a_2a_1a_0$ , its value is given by the equation in Figure 5-8.

$$\text{value} = -(1 \cdot a_{N-1}) + \sum_{i=0}^{N-2} 2^{(i+1-N)} \cdot a_i$$

**Figure 5-8. Two's Complement, Signed Fractional Equation**

This format can represent numbers in the range  $-1 \leq \text{operand} \leq 1 - 2^{(N-1)}$ .

For words and longwords, the largest negative number that can be represented is -1, whose internal representation is 0x8000 and 0x8000\_0000, respectively. The largest positive word is 0x7FFF or  $(1 - 2^{-15})$ ; the most positive longword is 0x7FFF\_FFFF or  $(1 - 2^{-31})$ .

### 5.5.2 MAC Opcodes

MAC opcodes are mapped into line A and are described in the *PRM* (see URL <http://www.motorola.com/ColdFire/>).

Note the following:

- Unless noted otherwise, the setting of MACSR[N,Z] is based on the result of the final operation involving the product and the accumulator.
- The overflow (V) flag is handled differently. It is set if the complete product cannot be represented as a 40-bit value (this applies to 32x32 integer operations only) or if the combination of the product with an accumulator cannot be represented in the given number of bits. The EMAC design includes an additional product/accumulation overflow bit for each accumulator that are treated as sticky indicators and are used to calculate the V bit on each MAC or MSAC instruction. See Section 5.4.1, “MAC Status Register (MACSR).”
- For the MAC design, the assembler syntax of the MAC (multiply and add to accumulator) and MSAC (multiply and subtract from accumulator) instructions does not include a reference to the single accumulator. For the EMAC, it is expected that assemblers support this syntax and that no explicit reference to an accumulator is interpreted as a reference to ACC0. These assemblers would also support syntaxes where the destination accumulator is explicitly defined.
- The optional 1-bit shift of the product is specified using the notation {<< | >>} SF, where <<1 indicates a left shift and >>1 indicates a right shift. The shift is performed before the product is added to or subtracted from the accumulator. Without this operator, the product is not shifted. If the EMAC is in fractional mode (MACSR[F/I] is set), SF is ignored and no shift is performed. Because a product can overflow, the following guidelines are followed:
  - For unsigned word and longword operations, a zero is shifted into the product on right shifts.
  - For signed, word operations, the sign bit is shifted into the product on right shifts unless the product is zero. For signed, longword operations, the sign bit is shifted into the product unless an overflow occurs or the product is zero, in which case a zero is shifted in.
  - For all left shifts, a zero is inserted into the lsb position.

The following pseudocode explains basic MAC or MSAC instruction functionality. This example is presented as a case statement covering the three basic operating modes with signed integers, unsigned integers, and signed fractionals. Throughout this example, a comma-separated list in curly brackets, {}, indicates a concatenation operation.

```
switch (MACSR[6:5])      /* MACSR[S/U, F/I] */
{
  case 0:                /* signed integers */
    if (MACSR.OMC == 0 || MACSR.PAVx == 0)
      then {
        MACSR.PAVx = 0
        /* select the input operands */
        if (sz == word)
          then {if (U/Ly == 1)
                  then operandY[31:0] = {sign-extended Ry[31], Ry[31:16]}
                  else operandY[31:0] = {sign-extended Ry[15], Ry[15:0]}
                if (U/Lx == 1)
```



```

        then operandX[31:0] = {sign-extended Rx[31], Rx[31:16]}
        else operandX[31:0] = {sign-extended Rx[15], Rx[15:0]}
    }
    else {operandY[31:0] = Ry[31:0]
        operandX[31:0] = Rx[31:0]
    }

    /* perform the multiply */
    product[63:0] = operandY[31:0] * operandX[31:0]

    /* check for product overflow */
    if ((product[63:39] != 0x0000_00_0) && (product[63:39] != 0xffff_ff_1))
    then { /* product overflow */
        MACSR.PAVx = 1
        MACSR.V = 1
        if (inst == MSAC && MACSR.OMC == 1)
        then if (product[63] == 1)
            then result[47:0] = 0x0000_7fff_ffff
            else result[47:0] = 0xffff_8000_0000
        else if (MACSR.OMC == 1)
            then /* overflowed MAC,
                saturationMode enabled */
                if (product[63] == 1)
                then result[47:0] = 0xffff_8000_0000
                else result[47:0] = 0x0000_7fff_ffff
            }

    /* sign-extend to 48 bits before performing any scaling */
    product[47:40] = {8{product[39]}} /* sign-extend */

    /* scale product before combining with accumulator */
    switch (SF) /* 2-bit scale factor */
    {
        case 0: /* no scaling specified */
            break;
        case 1: /* SF = "<< 1" */
            product[40:0] = {product[39:0], 0}
            break;
        case 2: /* reserved encoding */
            break;
        case 3: /* SF = ">> 1" */
            product[39:0] = {product[39], product[39:1]}
            break;
    }

    if (MACSR.PAVx == 0)
    then {if (inst == MSAC)
        then result[47:0] = ACCx[47:0] - product[47:0]
        else result[47:0] = ACCx[47:0] + product[47:0]
    }

    /* check for accumulation overflow */
    if (accumulationOverflow == 1)
    then {MACSR.PAVx = 1
        MACSR.V = 1
        if (MACSR.OMC == 1)
        then /* accumulation overflow,
            saturationMode enabled */
            if (result[47] == 1)
            then result[47:0] = 0x0000_7fff_ffff
            else result[47:0] = 0xffff_8000_0000
        }

    /* transfer the result to the accumulator */
    ACCx[47:0] = result[47:0]
}
MACSR.V = MACSR.PAVx
MACSR.N = ACCx[47]
if (ACCx[47:0] == 0x0000_0000_0000)
then MACSR.Z = 1
else MACSR.Z = 0
if ((ACCx[47:31] == 0x0000_0) || (ACCx[47:31] == 0xffff_1))
then MACSR.EV = 0

```

## EMAC Instruction Set Summary

```
        else MACSR.EV = 1
break;
case 1,3:          /* signed fractionals */
    if (MACSR.OMC == 0 || MACSR.PAVx == 0)
    then {
        MACSR.PAVx = 0
        if (sz == word)
        then {if (U/Ly == 1)
            then operandY[31:0] = {Ry[31:16], 0x0000}
            else operandY[31:0] = {Ry[15:0], 0x0000}
            if (U/Lx == 1)
            then operandX[31:0] = {Rx[31:16], 0x0000}
            else operandX[31:0] = {Rx[15:0], 0x0000}
        }
        else {operandY[31:0] = Ry[31:0]
            operandX[31:0] = Rx[31:0]
        }
        /* perform the multiply */
        product[63:0] = (operandY[31:0] * operandX[31:0]) << 1
        /* check for product rounding */
        if (MACSR.R/T == 1)
        then { /* perform convergent rounding */
            if (product[23:0] > 0x80_0000)
            then product[63:24] = product[63:24] + 1
            else if ((product[23:0] == 0x80_0000) && (product[24] == 1))
            then product[63:24] = product[63:24] + 1
        }
        /* sign-extend to 48 bits and combine with accumulator */
        /* check for the -1 * -1 overflow case */
        if ((operandY[31:0] == 0x8000_0000) && (operandX[31:0] == 0x8000_0000))
        then product[71:64] = 0x00 /* zero-fill */
        else product[71:64] = {8{product[63]}} /* sign-extend */
        if (inst == MSAC)
        then result[47:0] = ACCx[47:0] - product[71:24]
        else result[47:0] = ACCx[47:0] + product[71:24]
        /* check for accumulation overflow */
        if (accumulationOverflow == 1)
        then {MACSR.PAVx = 1
            MACSR.V = 1
            if (MACSR.OMC == 1)
            then /* accumulation overflow,
                saturationMode enabled */
                if (result[47] == 1)
                then result[47:0] = 0x007f_ffff_ff00
                else result[47:0] = 0xff80_0000_0000
            }
        /* transfer the result to the accumulator */
        ACCx[47:0] = result[47:0]
    }
    MACSR.V = MACSR.PAVx
    MACSR.N = ACCx[47]
    if (ACCx[47:0] == 0x0000_0000_0000)
    then MACSR.Z = 1
    else MACSR.Z = 0
    if ((ACCx[47:39] == 0x00_0) || (ACCx[47:39] == 0xff_1))
    then MACSR.EV = 0
    else MACSR.EV = 1
break;
case 2:          /* unsigned integers */
    if (MACSR.OMC == 0 || MACSR.PAVx == 0)
    then {
        MACSR.PAVx = 0
        /* select the input operands */
        if (sz == word)
        then {if (U/Ly == 1)
            then operandY[31:0] = {0x0000, Ry[31:16]}
            else operandY[31:0] = {0x0000, Ry[15:0]}
            if (U/Lx == 1)
            then operandX[31:0] = {0x0000, Rx[31:16]}
            else operandX[31:0] = {0x0000, Rx[15:0]}
        }
        else {operandY[31:0] = Ry[31:0]
```

```

        operandX[31:0] = Rx[31:0]
    }

    /* perform the multiply */
    product[63:0] = operandY[31:0] * operandX[31:0]

    /* check for product overflow */
    if (product[63:40] != 0x0000_00)
    then { /* product overflow */
        MACSR.PAVx = 1
        MACSR.V = 1
        if (inst == MSAC && MACSR.OMC == 1)
            then result[47:0] = 0x0000_0000_0000
        else if (MACSR.OMC == 1)
            then /* overflowed MAC,
                    saturationMode enabled */
                result[47:0] = 0xffff_ffff_ffff
    }

    /* zero-fill to 48 bits before performing any scaling */
    product[47:40] = 0 /* zero-fill upper byte */

    /* scale product before combining with accumulator */
    switch (SF) /* 2-bit scale factor */
    {
        case 0: /* no scaling specified */
            break;
        case 1: /* SF = "<< 1" */
            product[40:0] = {product[39:0], 0}
            break;
        case 2: /* reserved encoding */
            break;
        case 3: /* SF = ">> 1" */
            product[39:0] = {0, product[39:1]}
            break;
    }

    /* combine with accumulator */
    if (MACSR.PAVx == 0)
    then {if (inst == MSAC)
        then result[47:0] = ACCx[47:0] - product[47:0]
        else result[47:0] = ACCx[47:0] + product[47:0]
    }

    /* check for accumulation overflow */
    if (accumulationOverflow == 1)
    then {MACSR.PAVx = 1
        MACSR.V = 1
        if (inst == MSAC && MACSR.OMC == 1)
            then result[47:0] = 0x0000_0000_0000
        else if (MACSR.OMC == 1)
            then /* overflowed MAC,
                    saturationMode enabled */
                result[47:0] = 0xffff_ffff_ffff
    }

    /* transfer the result to the accumulator */
    ACCx[47:0] = result[47:0]
}
MACSR.V = MACSR.PAVx
MACSR.N = ACCx[47]
if (ACCx[47:0] == 0x0000_0000_0000)
    then MACSR.Z = 1
    else MACSR.Z = 0
if (ACCx[47:32] == 0x0000)
    then MACSR.EV = 0
    else MACSR.EV = 1
break;
}

```



# Chapter 6

## Instruction Pipeline and Timing

This chapter describes performance features of the V4 ColdFire processor pipeline structure. These features are common for all V4 standard products as well as custom products using the CF4e core. Relevant information on the CF4e EMAC, FPU, and MMU are also included. It is intended as a guide for developing compilers or optimizing assembly language application code.

This chapter describes the basic V4 pipeline strategy, contrasting it with Version 2 and 3 designs. It provides performance-related details of the instruction fetch and operand execution pipelines (IFP and OEP). Appendixes describe performance parameters for the complete ColdFire instruction set.

### 6.1 Basic V4 Pipeline Strategy

All ColdFire generations include two independent, decoupled pipelines. The instruction fetch pipeline (IFP) prefetches the instruction stream, examines it to predict changes of flow, partially decodes instructions, and packages fetched data into instructions for the OEP. The instruction stream is then gated into the operand execution pipeline (OEP), which decodes instructions, fetches required operands, and executes required functions. The IFP and OEP are decoupled by a FIFO instruction buffer. The IFP can prefetch instructions before the OEP needs them, minimizing the wait for instructions. This organization has proven very efficient implementation as dictated by the variable-length ColdFire instruction set.

Version 2 ColdFire processor design minimizes overall size while maintaining a reasonable performance level. As a result, the V2 OEP is a simple two-stage design, where instructions requiring operand memory read references are sent through both stages twice. With this and the significant K-Bus enhancements in the V3 design, V4 minimizes instruction latency by unrolling the OEP to support single-cycle latency for most opcodes.

Table 6-1 highlights these differences and presents a subset of execution times for the basic instructions across the ColdFire architecture versions.

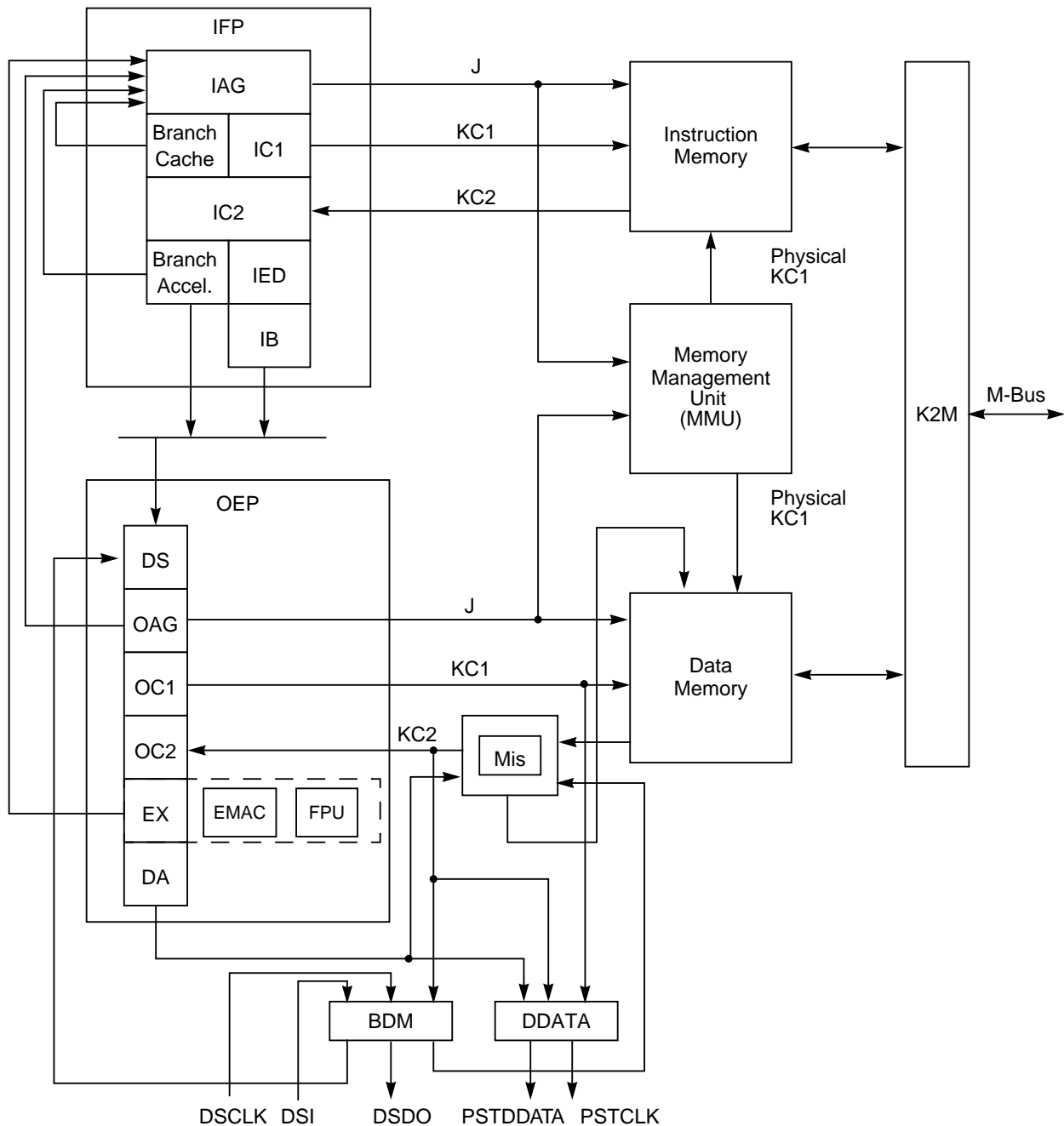
**Table 6-1. CFxCore Processor Execution Latency**

Instruction	Operation	Number of Processor Cycles		
		V2	V3	V4
<op> Ry,Rx	Register-to-register	1	1	1
mov.l <mem>y,Rx	32-bit load	2	3	1
mov.b <mem>y,Rx	8-bit load	3	4	1
mov.w <mem>y,Rx	16-bit load	3	4	1
mov.* Ry,<mem>x	store	1	1	1
mov.l <mem>y,<mem>x	Memory-to-memory	2	3	2
<op> <mem>y,Rx	Embedded load	3	4	1
<op> Ry,<mem>x	Read-modify-write	3	4	1
bsr, jsr label	Subroutine call	3	1	1
rts	Subroutine return	5	8	2
bra label	Branch always	2	1	1
bcc label (forward, not taken) (forward, taken) (backward, not taken) (backward, taken) (predicted correctly) (predicted incorrectly)	Conditional branch	1 3 3 2	    1 5	    0 8

Unrolling the OEP has two major ramifications on the processor complex:

- To support both pipeline structures, significantly more bandwidth is needed than earlier ColdFire versions. Both V2 and V3 instruction fetch and operand requests share the K-Bus. For V4, a unified structure does not provide enough bandwidth for instruction fetches, so a split bus, or Harvard architecture, is used so that separate instruction and data memories can be accessed concurrently. The basic two-stage pipeline V3 K-Bus structure is retained, with one K-Bus connecting the IFP to instruction memory and the other connecting the OEP to data memory.
- By increasing the number of OEP stages, coupled with a V3-style IFP, the combined pipeline depth requires architectural enhancements to handle branch instructions. Specifically, the V3 branch acceleration scheme alone does not achieve desired performance levels. V4 processors implement a two-level adaptive prediction scheme with a small, direct-mapped branch cache using V3-style branch acceleration to minimize mispredicted branches.

The resulting pipeline structure is shown in Figure 6-1.



**Figure 6-1. CF4e ColdFire Processor Complex Block Diagram**

The IFP and OEP consist of the following stages:

- Four-stage IFP
  - IAG (instruction address generation)
  - IC1 (instruction fetch cycle 1)
  - IC2 (instruction fetch cycle 2)
  - IED (instruction early decode)

- IB (instruction buffer)—(optional) Prefetched instructions can pass directly into the OEP instruction registers, hiding IB from software in most cases.
- Five-stage OEP with two optional write stages
  - DS (decode and select)
  - OAG (address generation)
  - OC1 (operand fetch cycle 1)
  - OC2 (operand fetch cycle 2)
  - EX (execute)
  - DA (write data available (operand write operations only))
  - ST (store data (operand write operations only))

In summary, the OEP implements a five-stage design with limited superscalar instruction issue capabilities to provide a cost-effective solution for the V4e core.

## 6.2 Instruction Fetch Pipeline (IFP)

The IFP generates instruction addresses and fetches. Because the fetch and execution pipelines are decoupled by the eight-instruction FIFO instruction buffer (IB), the IFP can prefetch instructions before the OEP needs them, minimizing stalls.

The IED stage provides early decoding, which effectively implements a hardware lookup table. First, the 32 bits of fetched instruction data are separated into 16-bit parcels, the minimum size for ColdFire instructions. Next, each parcel is used to index a hardware table that provides a vector to decode fields that provide information such as instruction length, data memory reference type, necessary register resources, and control information needed early in the OEP DS stage. Finally, the instruction and its early decode information are loaded into the instruction buffer or directly into the OEP. The early decode information becomes the extended opword as it enters the OEP.

The primary IFP/OEP interface includes 48 bits of instruction (16-bit opword and two optional 16-bit extension words) along with the extended opword containing the decode vector. The IFP also supplies another 16-bit opword and its extended opword for the next sequential instruction to the OEP to support the limited superscalar dispatch capabilities.

In addition to the prefetch function, the IFP improves the performance of change-of-flow operations through the following:

- 8-entry, direct-mapped branch cache unit (BCU). Associates branch instruction addresses with the target address for taken conditional branch instructions (Bcc). Each entry includes a 2-bit, four-state branch prediction value that predicts a Bcc instruction to be strongly or weakly taken or not taken. Branch folding of the branch cache entry allows zero-cycle Bcc execution times for correctly predicted taken branches. To maximize effectiveness of the small direct-mapped branch cache, a hashed address indexes into the cache. The hashed address is generated as follows:

```
hashedBcuAddress[2:0] = IfpAddr[7:5] XOR IfpAddr[4:2]
```



- 128-entry, direct-mapped prediction table unit (PTU).
- Predicts Bcc instructions that miss in the branch cache. If predicted as taken by the PTU, the Bcc is accelerated in the same manner as that used in the Version 3 processor. This acceleration is implemented in the IED stage of the prefetch pipeline and consists of the required hardware to calculate the target instruction address which is then fed back into the IFP's IAG stage. This mechanism is also used for certain unconditional change-of-flow instructions. Decoupling the IFP and OEP usually yields a 1-cycle execution time for correctly predicted accelerated branches. Again, a hashed address is generated to index into the prediction table. This hashed address is defined as follows:

```
hashedPtuAddress[6:0] = IfpAddr[15:9] XOR IfpAddr[8:2]
```

- 4-entry LIFO hardware return stack. Accelerates subroutine return instructions. Because an RTS can return control to any number of target addresses, RTS opcodes do not benefit from traditional branch cache structures, so the four-entry stack greatly improves performance of these instructions. This stack is invisible to application software. When a subroutine call is executed, the IFP pushes the return program counter (PC) onto the stack. When a subroutine return is encountered in the prefetch stream, the top of the LIFO stack is popped off (if valid) and used to establish a new prefetch stream. The OEP subsequently verifies that the predicted target address matches the return address on top of the memory-based system stack. If the address differs, the processor aborts processing and reestablishes control at the address defined by the memory stack. Table 6-2 lists RTS execution times.

**Table 6-2. V4 RTS Execution Times**

Execution Time	Condition
2 (1/0)	Predicted and correct
8 (1/0)	Not predicted
9 (1/0)	Predicted but incorrect

V4 core performance has been evaluated across a large suite of compiled (no assembly language optimizations) embedded benchmarks, from which the following IFP branch-related performance parameters have been measured:

- 64% of all Bcc instructions are folded so they execute in 0 cycles
- 87% of the predictions provided by the BCU + PTU on Bcc instructions are correct
- Conditional branches typically account for 11% of the dynamic pathlength
- 99% of all RTS opcodes are predicted correctly by the hardware return stack

The decoupled IFP and OEP and Harvard architecture of the V4 core efficiently handles the variable-length ColdFire instruction set. Performance measurements indicate that a Base CPI degradation factor of 0.06 cycles per instruction is caused by the OEP waiting for opwords or extension words to be supplied by the IFP. In some cases, this factor can be

reduced by forcing branch target instructions to be aligned on 0-modulo-4 addresses at the cost of increased code size. In all cases, the 16-bit TPF instruction (0x51FC) should be used for text fill, rather than a NOP instruction (0x4E71). NOP synchronizes the pipeline as it begins execution, producing a 6-cycle minimum latency versus the 1-cycle TPF opcode.

### 6.3 Operand Execution Pipeline (OEP)

The two instruction registers in the decode stage (DS) of the OEP are loaded from the FIFO instruction buffer or are bypassed directly from the instruction early decode (IED). The OEP consists of two, traditional two-stage RISC compute engines with a dual-ported register file access feeding an arithmetic logic unit (ALU).

The compute engine at the top of the OEP (the address ALU) is used typically for operand address calculations; the execution ALU at the bottom is used for instruction execution. The resulting structure provides almost 24 Mbytes/MHz of bandwidth to the two compute engines and supports single-cycle execution speeds for most instructions, including all load and store operations and most embedded-load operations. The V4 OEP supports the ColdFire Revision B instruction set, which adds a few new instructions to improve performance and code density.

The OEP also implements the following advanced performance features:

- Stalls are minimized by dynamically basing the choice between the address ALU or execution ALU for instruction execution on the pipeline state.
- The address ALU and register renaming resources together can execute heavily used opcodes and forward results to subsequent instructions with no pipeline stalls.
- Instruction folding involving MOVE instructions allows two instructions to be issued in one cycle. The resulting microarchitecture approaches full superscalar performance at a much lower silicon cost.

Unrolling the OEP into five stages improves V4 performance. The resulting structure is termed ‘limited superscalar’ because of certain, heavily used instruction constructs that support multiple-instruction dispatch. In particular, the notion of instruction folding where two consecutive operations are combined into a single issue effectively creates zero-cycle latency for some instructions.

#### 6.3.1 V4 OEP Conceptual Pipeline Model

The basic compute engine for the V4 ColdFire processor consists of a two-stage pipeline—a register file with dual read ports feeding an arithmetic/logic unit (ALU). This compute engine follows the traditional RISC model and is a three-terminal device: two input operands and a result. Because the ColdFire ISA is not a pure load/store model, the OEP consists of two of these compute engines, one for operand address generation in the DS/OAG stages and one for instruction located in the OC2/EX stages. The resulting port list defines the following set of resources associated with each compute engine:

```

module (OagComputeEngine)          module (ExComputeEngine)
  input  (baseRegister              input  (operand1,
    indexRegister)                  operand2)
  output (addressResult);           output (executeResult);

```

Thus, each instruction has these six resources (four input operands and two results) associated with its execution:

```

Instruction Resources = f(baseRegister, indexRegister, addressResult,
                        operand1,      operand2,      executeResult);

```

Although OagComputeEngine is most often used to calculate operand addresses for memory-referencing instructions, its use is not limited to address generation. The entire ColdFire ISA can be grouped into several broad categories:

- Memory-referencing instructions that use the OagComputeEngine to generate the operand address and execute in the ExComputeEngine
- Register-to-register instructions execute in the ExComputeEngine
- Register-to-register opcodes that execute in the OagComputeEngine

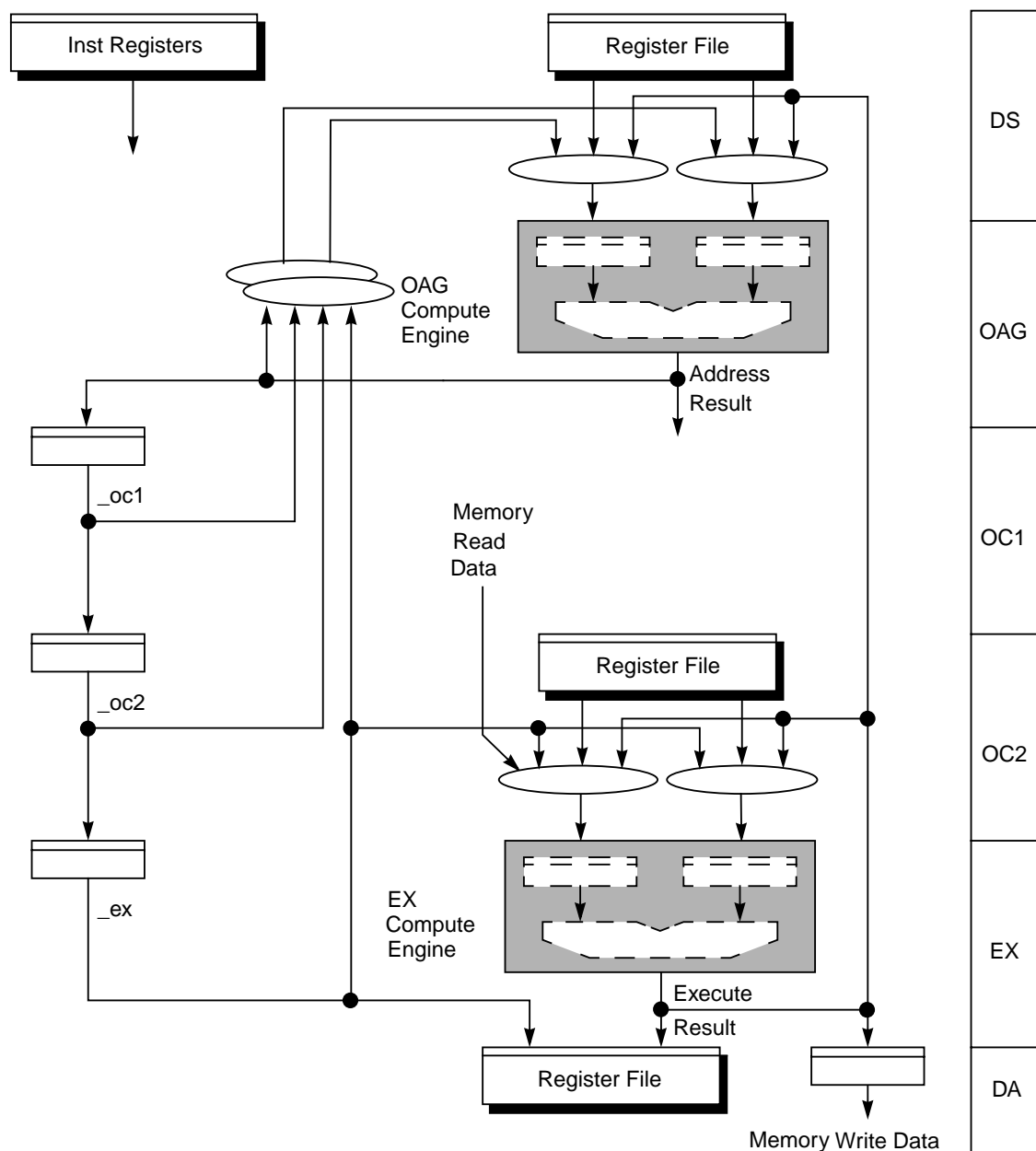
#### NOTE:

To support precise exceptions, results are not committed to program-visible registers until an instruction completes the EX stage even if it executes in the OagComputeEngine.

Register renaming makes executing instructions in OagComputeEngine advantageous because results are available to subsequent instructions immediately. Instructions executed in the OagComputeEngine include the following:

- lea<ea>y,Ax
- mov.l #<data>,Rx
- moveq #<data>,Dx

Register renaming resources are the cascaded pipeline registers with the \_oc1, \_oc2, and \_ex labels in Figure 6-2. Note that the contents of these resources can be dynamically forwarded to an operation in the DS stage with no pipeline stalls.



**Figure 6-2. OAGComputeEngine Register Renaming Resources**

V4 adds a fourth category ({register, immediate}-to-register instructions) to the three basic opcode classifications. These instructions can be executed in either compute engine. This capability, called dynamic execution relocation, improves performance by moving execution to the OagComputeEngine whenever possible, maximizing use of renaming hardware to eliminate pipeline stalls. This category includes `mov.l Ry,Rx` and most register-based arithmetic operations (for example, `op.l {Ry|#imm},Rx`). This pipeline technology, developed for the MC68060, minimizes pipeline register-busy stalls. As each instruction enters the OEP DS stage, the IFP's early-decode logic supplies the extended opword that specifies required instruction resources. Control logic checks whether required input register resources have writes pending from the ExComputeEngine in the OEP's

OAG, OC1, or OC2 stage. If no ExComputeEngine write from these stages is pending, the execution of the opcode may be relocated from the EX stage to the OAG stage.

The last four OEP stages provide a strict scheme for prioritizing pending register updates. A clear understanding of this scheme is crucial because there are often multiple updates for a single destination register in the pipeline state at any time. Prioritization is as follows:

1. ExComputeEngine update, OAG stage (highest priority)
2. OagComputeEngine update, OAG stage
3. ExComputeEngine update, OC1 stage
4. OagComputeEngine update, OC1 stage
5. ExComputeEngine update, OC2 stage
6. OagComputeEngine update, OC2 stage
7. ExComputeEngine update, EX stage
8. OagComputeEngine update, EX stage (lowest priority)

The OEP implements a 2- x 4-stage scoreboard for tracking pending register updates from the two compute engines. DS stage control logic uses this scoreboard to determine if dynamic execution relocation can be performed and to generate pipeline stalls on register-busy conditions, described in Section 6.3.3, “Sequence-Related OEP Stalls.”

V4 processor core performance measurements produce the following:

- 12% of the instructions always execute in the OagComputeEngine.
- 30% of the instructions can be executed in either compute engine, and 15% of the total instructions are relocated to the OagComputeEngine.
- 18% of the instructions include auto-addressing mode updates  $\{(An)+, -(An)\}$  performed by the OagComputeEngine. By summing these three classes, the OagComputeEngine is used on 45% of the dynamic instructions.

Section 6.4, “Instruction Execution Locations,” gives a complete specification of the OEP compute engine execution location for every ColdFire instruction.

### 6.3.2 Instruction Folding and the Limited Superscalar OEP

The V4 branch cache supports zero-cycle execution of correctly predicted taken conditional branch instructions. The V4 OEP also supports instruction folding on other heavily used constructs. In particular, MOVE is an ideal candidate for instruction folding for two reasons. First, two-operand ColdFire constructs mean that simple assignment operations using MOVE opcodes are used extensively. Second, because a MOVE simply involves an assignment but no other computation operation, it can be combined with other opcodes for dual-issue opportunities using both OEP compute engines. This instruction folding provides limited superscalar dispatch.

## Operand Execution Pipeline (OEP)

Using the nomenclature from the superscalar MC68060, two sequential instructions are loaded into the OEP when an opcode is requested from the IFP. The OEP is implemented as the primary OEP (pOEP) plus the DS stage for the secondary OEP (sOEP). The sOEP instruction dispatch criteria are evaluated in the DS pipeline stage; if successful, the secondary instruction is issued to the OAG stage. V4 sOEP instructions are restricted to 16 bits with no operand memory references and are executed in the ExComputeEngine.

V4 instruction pairs are grouped into the three following broad categories. By supporting superscalar dispatch for heavily-used instruction constructs, the design approaches the performance of a full, dual-pipeline OEP at a much lower silicon cost.

### Group 1: Zero-cycle loads

```
pOEP inst = mov.l {Ry | <mem>y}, Rx
sOEP inst = op.l {Rw | #qimm}, Rx
```

(#qimm represents a 3-bit quick immediate operand)

For this pair, a MOVE shares a destination register with a secondary instruction and the full capabilities of the ExComputeEngine's three-terminal structure can be exploited. The executeResult is a function of (operand1, operand2). The combined instructions are issued into the OAG stage as op.l {Ry | <mem>y}, {Rw | #qimm}, Rx to be executed by the OEP EX stage.

### Group 2: Zero-cycle stores

```
pOEP inst = store.* Ry, <mem>x
sOEP inst = mov.l   Rw, Rz           or,
               movq.l #imm, Rz
```

For this pair, a store operation (mov.{b,w,l} or clr.{b,w,l}) is combined with a simple register load. The Ex compute engine store unit executes the operand write. This function is tied directly to the operand1\_ex register, providing the required post-alignment multiplexing logic. The sOEP instruction is issued to the barrel shifter (BSU), that performs a passOperand2 operation. ExComputeEngine processes both operations concurrently.

### Group 3: Zero-cycle address results

```
pOEP inst =      lea      <ea>y, Rx           or,
                 mov.l    #imm, Rx           or,
                 movq.l    #imm, Rx           or,
                 clr.l     Rx                or,
                 mov3q.l   #qimm, Rx
sOEP inst =      op.l     {Rw | #qimm}, Rz    or,
                 mov.l     Rw, Rz            or,
                 movq.l     #imm, Rz         or,
                 cmp.l     Rw, Rz
```

This pair combines a pOEP instruction executed by the OagComputeEngine (the AddressResult) with a sOEP instruction executed in the ExComputeEngine.

Measurements show that folding instructions to create zero-cycle moves improves overall processor performance by 10% on compiled code. Combining this with improvements

provided by zero-cycle Bcc instructions, 33% of the dynamic instruction stream is executed as pairs in the OEP.

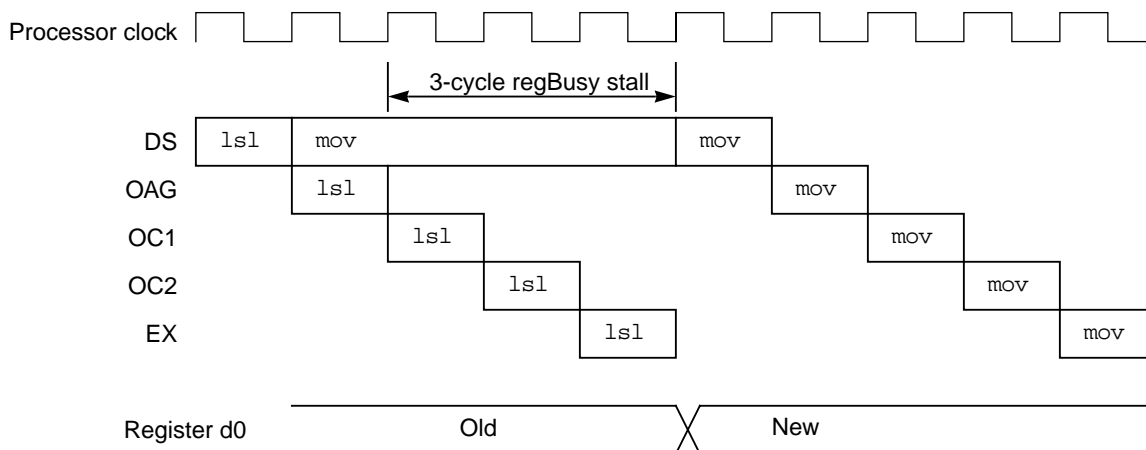
Superscalar dispatch can be disabled for performance analysis. Setting CACR[17] disables Bcc instruction folding. Setting CACR[0] disables OEP instruction folding on zero-cycle move pairs. Both are enabled at reset.

### 6.3.3 Sequence-Related OEP Stalls

The most common sequence-related OEP stall is the change/use (register-busy) register stall, which occurs when an instruction modifies a register in the ExComputeEngine that a subsequent instruction needs as an OagComputeEngine input. The subsequent instruction stalls in the DS stage until the register is updated. Consider the following:

```
lsl.l    d1,d0
mov.l    (d8,a0,d0.l*4),d1
```

In this sequence, shown in Figure 6-3, the 2-cycle mov.l instruction stalls waiting for lsl.l to update the d0 index register. The mov.l instruction requires a second pipeline cycle to calculate the three-component indexed operand address.



**Figure 6-3. Sequence-Related OEP Sequence Stall**

In Figure 6-3, mov.l cannot begin its second cycle of indexed operand address generation until the preceding instruction update d0, causing a worst-case, 3-cycle stall.

The worst-case change/use register-busy stalls are summarized as follows:

- If an instruction changes an ExComputeEngine base register (An) and the next instruction uses that register as an OagComputeEngine input, a 3-cycle stall occurs.
- If mov.l <mem>y,Ax loads a base register and the next instruction uses that register as an OagComputeEngine input, a 2-cycle stall occurs. This sequence, common in pointer manipulation, is optimized by adding a OC2-to-DS forwarding datapath.

## Operand Execution Pipeline (OEP)

- If an instruction changes an ExComputeEngine register that the next instruction uses as an index register as an OagComputeEngine input with a scale factor of 1 (Xi.l), a 2-cycle pipeline stall occurs.
- If an instruction changes a register from either OagComputeEngine or ExComputeEngine and the next instruction uses that register as an index register with a scale factor other than 1 (for example, Xi.l\*{2,4,8}) as an input to OagComputeEngine, a 3-cycle stall occurs. The case is shown in Figure 6-3.

The first three stalls are minimized by using ExComputeEngine-to-DS stage forwarding logic shown on the OEP block diagrams. For register-busy conditions involving index registers with scale factors not equal to one, the register file must be updated at the completion of the EX stage before the stalled instruction can continue.

Intervening instructions can reduce or eliminate the stall, as in the following sequence:

```
add.l    (d16,a7),a0
mov.l    (a0),d0
```

This sequence represents the first type of change/use pipeline stall (three cycles) on register A0. If instruction scheduling can be applied, the stall can be reduced or eliminated.

```
add.l    (d16,a7),a0
op1      Ry,Rx
op2      Rw,Rz
mov.l    (a0),d0
```

By inserting single-cycle instructions, op1 and op2, mov.l stalls only 1 cycle because the 3-cycle hazard is reduced by the 2 cycles during which op1 and op2 execute. A third single-cycle instruction would eliminate the stall.

The instructions in Table 6-3 prevent stalls by using register renaming logic to make destination register results available to subsequent instructions. These instructions are unconditionally executed in OagComputeEngine.

**Table 6-3. Instructions that Make Results Available to Subsequent Instructions**

Instruction
<op> (Ay)+,Rx
<op> -(Ay),Rx
<op> Ry,(Ax)+
<op> Ry,-(Ax)
clr.l dx
lea <ea>y,Ax
mov.l #imm,Rx
mov.w #imm,Ax
mov3q.l #qimm,Rx
moveq #imm,Dx



**NOTE:**

Ry indicates a Dy or Ay source register. Rx indicates a Dx or Ax destination register available either to the OagComputeEngine on subsequent instructions as a base register with no stall or as an index register with a scale factor of 1 and no stall. If the destination register is then used as an index with a different scale factor, the 3-cycle stall described above occurs

V4 CPU measurements indicate a 0.12-cycle-per-instruction degradation factor associated with change/use stalls across the embedded benchmark suite. Approximately 6% of the dynamic instruction stream encounter this type of stall. The average stall is about 2 cycles per register-busy.

### 6.3.4 EMAC-Specific OEP Sequence Stalls

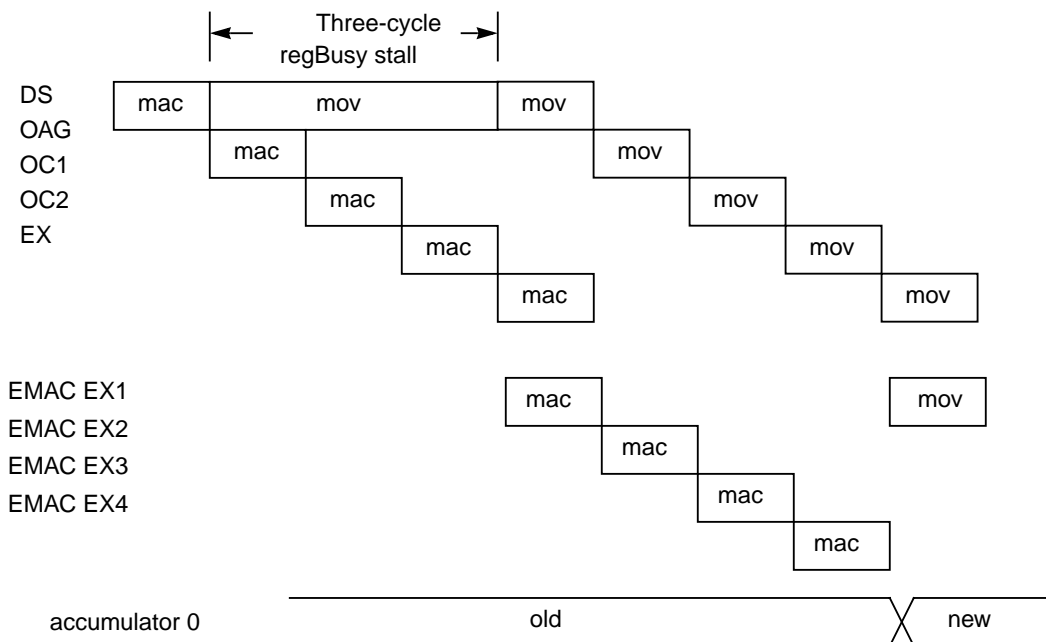
The ColdFire family supports two multiply-accumulate implementations that provide different levels of performance and capability for differing silicon costs. The EMAC features a four-stage execution pipeline, optimized for 32-bit operands with a fully-pipelined 32 x 32 multiply array and four 48-bit accumulators. A MAC or EMAC can be attached to any version ColdFire core as determined by application requirements.

The EMAC execution pipeline overlaps the EX stage of the OEP; that is, the first stage of the EMAC pipeline is the last stage of the basic OEP. EMAC units are designed for sustained, fully-pipelined operation on accumulator load, copy, and multiply-accumulate instructions. However, instructions that store contents of the multiply-accumulate programming model can generate OEP stalls that expose the EMAC execution pipeline depth, as in the following

```
mac.w    Ry,Rx,Acc0
mov.l    Acc0,Rz
```

The mov.l instruction that stores the accumulator to an integer register (Rz) stalls until the program-visible copy of the accumulator is available. Figure 6-4 shows EMAC timing.

## Operand Execution Pipeline (OEP)



**Figure 6-4. EMAC-Specific OEP Sequence Stall**

In Figure 6-4, the OEP stalls the store-accumulator instruction for 3 cycles: the depth of the EMAC pipeline minus 1. The minus 1 factor is needed because the OEP and EMAC pipelines overlap by a cycle, the EX stage. As the store-accumulator instruction reaches the EX stage where the operation is performed, the just-updated accumulator 0 value is available.

As with change/use stalls between accumulators and general-purpose registers, introducing intervening instructions that do not reference the busy register can reduce or eliminate sequence-related store-MAC instruction stalls. In fact, a major benefit of the EMAC is the addition of three accumulators to minimize stalls caused by exchanges between the accumulator(s) and the general-purpose registers.

### 6.3.5 FPU-Specific OEP Sequence Stalls

One FPU-related read-after-write register data hazard merits mention involving OEP register file operands sourced to the FPU (for example, `fpGEN Ry,FPx` instruction). If the source `Ry` register is modified by a previous instruction, the FPU operation stalls for 1 cycle, so an `executeResult-to-OC2` forwarding datapath is not required. This restriction allows the register operand sourced directly from the OEP register file to the FPU relatively early in the OC2 cycle. The stall occurs in OAG. Consider the following sequence:

```
sub.l <ea>y,d0
fadd.l d0,fp0
```

The source operand `d0` is modified immediately before the FPU uses it. The `fadd.l` detects the pending register update in its DS stage, forcing a 1-cycle stall in OAG. This stall delays the OC2 stage for `fadd.l` until the updated `d0` value is available from the OEP register file.

The following example shows the basic operation of the CF4e OEP with the FPU. Consider the following code example taken from a popular floating-point benchmark:

```
rInnerproduct (result, a, b, row, column)
float *result, a[rowsize+1][rowsize+1], b[rowsize+1][rowsize+1];
int    row, column;

/* computes the inner product of A[row,*] and B[*,column] */
{
    int i;
    *result = 0.0;
    for (i = 1; i <= rowsize; i++)
        *result = *result + a[row][i] * b[i][column];
}
```

The inner for loop generates the following compiled code:

```
for_loop:
    fmov.s   (a5),fp0           ; fp0 = b[i][column]
    fmul.s   (a1)+,fp0          ; fp0 = a[row][i] * b[i][column]
    fadd.s   (a4),fp0           ; fp0 = result + a[][] * b[][]
    subq.l   #1,d7              ; decrement loop counter
    lea      (const,a5),a5      ; adjust pointer for b[i][column]
    fmov.s   fp0,(a4)           ; store result
    bpl.b    for_loop           ; if done, exit, else continue loop
```

Due to concurrent OEP and FPU instruction execution, the visible execution time for this loop is less than the simple summation of individual execution times.

**Table 6-4. FPU Execution Example**

Instruction	Instruction Latency (CPU cycles)	Apparent Latency (CPU cycles)	Comments
fmov.s	1	1	FP load
fmul.s	4	4	FP multiply
fadd.s	4	4	FP add
subq.l	1	0	Hidden in OEP
lea	1	0	Hidden in OEP
fmov.s	2	2	FP store
bpl.b	1	0	Hidden via instruction folding
TOTAL	14	11	

Figure 6-5 shows the OEP/FPU pipeline diagrams for the example in Table 6-4.

## Operand Execution Pipeline (OEP)

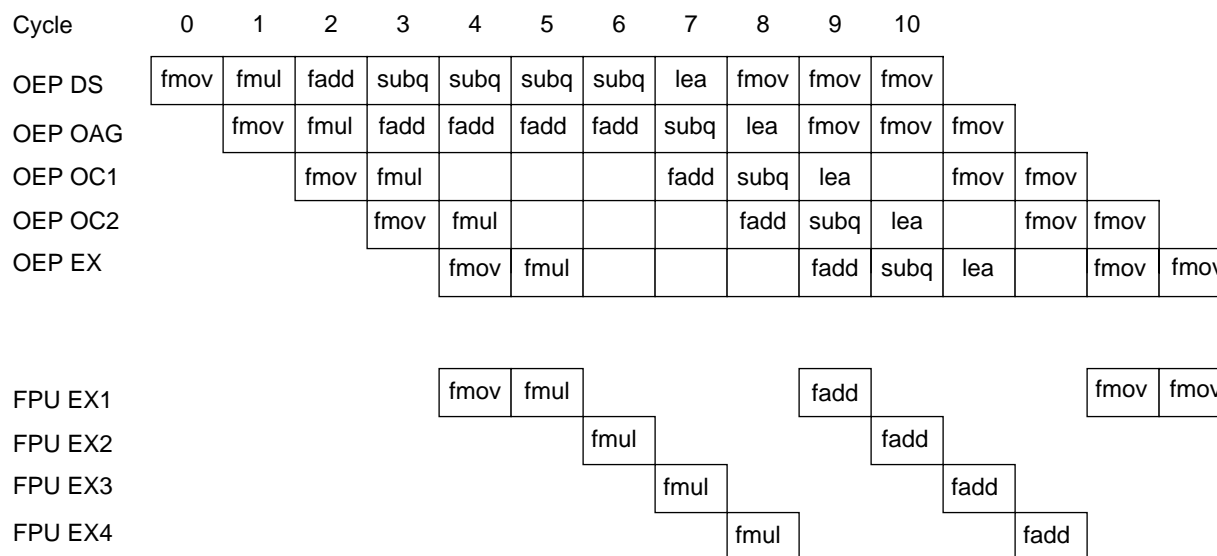


Figure 6-5. for\_loop Example

### 6.3.6 Operand Memory Sequence-Related Stalls

Operand reads and writes occur in different OEP stages; operand reads occur in OC1 and operand writes occur in ST. Accordingly, the read-after-write hazard in a given local memory can cause a 1-cycle stall. Specifically, if a memory read occupies OC1 and an operand write is in ST, the read stalls for a cycle to allow the write to complete because the memory array can only perform one operation per cycle. Performance measurements on this type of stall show a relatively minor degradation factor of 0.04 cycles per instruction.

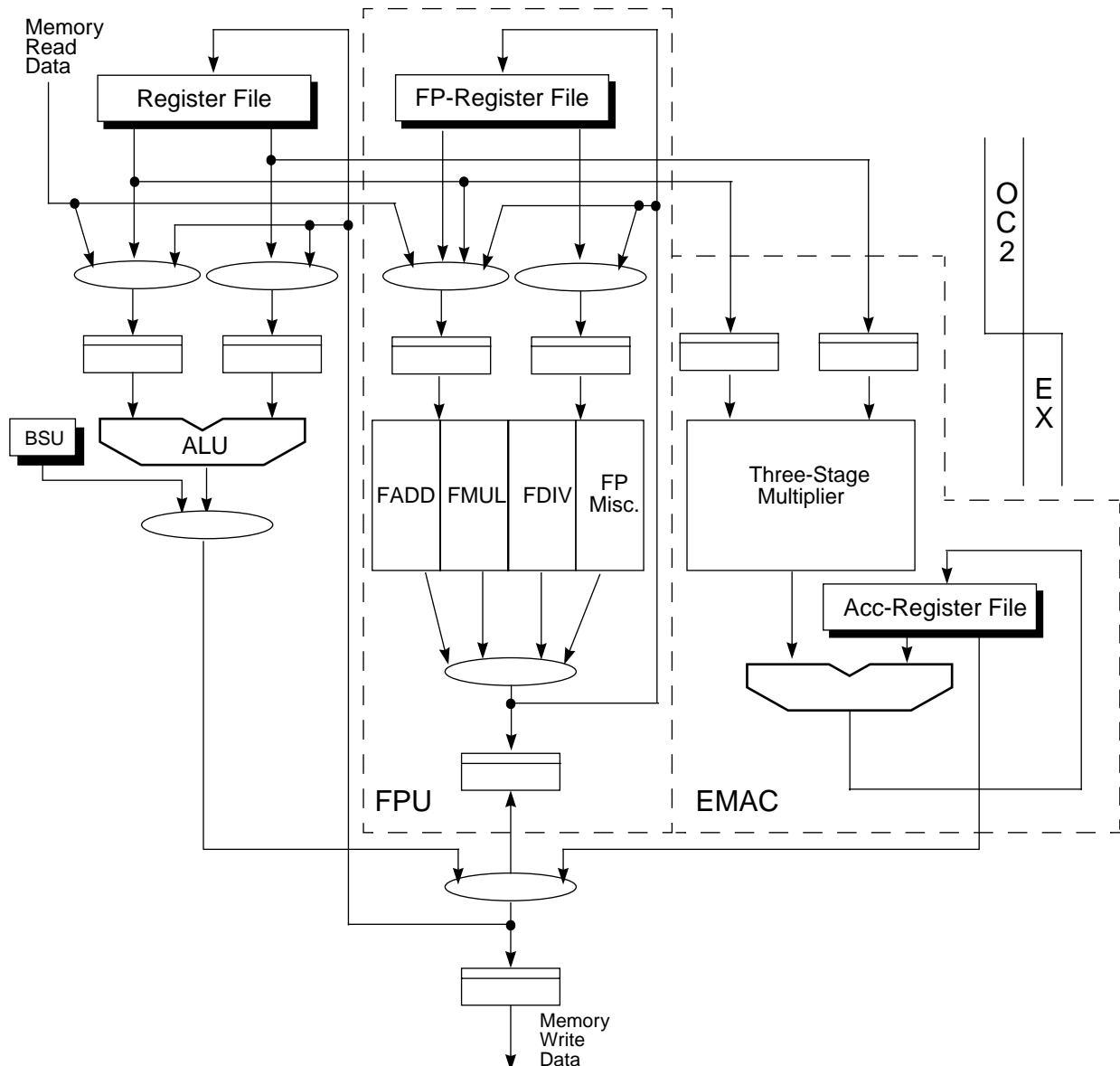
#### NOTE:

Read and write accesses must both be to the same local memory. For example, both access the data cache.

### 6.3.7 V4 OEP Summary

Measurements across the entire embedded suite into an ideal, infinitely large local memory show base CPI metrics ranging from 0.75 to 2.16 cycles per instruction, where the larger number includes benchmarks with significant use of multi-cycle arithmetic operations such as integer multiply and divide. The geometric mean across the entire suite is 1.34 cycles per instruction for the V4 core. Section 6.5, “Instruction Execution Times,” lists instruction execution times. Figure 6-6 shows a block diagram of the Ex execute engines of the V4 OEP.

Register elements in Figure 6-6 are shown as boxes with double lines showing boundaries between pipeline stages. Data multiplexers where various source operands are connected at the top are shown as ellipses; selected data is shown on the bottom.



**Figure 6-6. CF4e Ex Execute Engines within the OEP**

The last two OEP stages, operand fetch cycle 2 (OC2) and execute (EX), overlap the first stages of the FPU and EMAC pipelines. The OEP and FPU follow a traditional RISC execute engine model with a dual-ported register file feeding an ALU engine. In the CF4e pipeline, this function is partitioned across stages OC2 and EX.

The OEP has the two register-file read ports feeding into muxes that select source operands from data memory, the register file, or a fed-forward result from the preceding instruction. The two source operands are registered and sent to the execute engine, which consists of a basic ALU and BSU. The appropriate result is selected from all the potential sources (ALU, BSU, FPU, and EMAC) and routed back into the register file or sent to memory as write data for a store operation.

FPU hardware structure is similar to the OEP. Two source operands are selected from a variety of potential sources: memory read data, register operands from the OEP, and operands read from the FPU register file or the fed-forwarded results from the previous instruction. Recall the 68K/ColdFire floating-point instruction set architecture has eight unique floating-point data registers in the FPU's register file. Once selected, source operands are loaded into the two operand registers at the end of OC2 and then broadcast to the FPU's four internal engines: FADD, FMUL, FDIV, and miscellaneous unit. FADD executes all add, subtract, and compare instructions; FMUL performs all multiplication; FDIV performs division, square root, and move operations; and the miscellaneous module executes all other operations. If the destination is a FPU register, the appropriate output result bus is selected and fed back to the FPU register file; if the destination is an integer register or memory, results are registered and then sent to the OEP.

The EMAC's structure differs slightly because the basic instruction ( $Acc = Acc \pm Ry * Rx$ ) format differs from constructs executed by the OEP and FPU. For the EMAC, the two source operands ( $Ry, Rx$ ) are selected in the OEP and sent to the EMAC. As execution begins, the two operands enter a three-stage, 32x32 multiplier. The product is formed at the end of the third stage. It is then combined with the accumulator in stage four, when the accumulator is read from a register file and combined with the product; the result is then written back. Stores of any accumulator access a second read port and the appropriate value is sent to the OEP to be loaded into the destination location in the integer register file.

## 6.4 Instruction Execution Locations

Table 6-5 shows the OEP compute engine execution location for V4 instructions.

**Table 6-5. V4 ColdFire Compute Engine Location**

Instruction <sup>1</sup>	<ea> Oag	= Rn Ex	Either	<ea> Oag	= Mem Ex	Either	<ea> Oag	= Imm Ex	Either
add.l <ea>y,Rx			x		x				x
add.l Dy,<ea>x					x				
addi.l #imm,Dx									x
addq.l #imm,<ea>x			x		x				
addx.l Dy,Dx		x							
and.l <ea>y,Dx			x		x				x
and.l Dy,<ea>x					x				
andi.l #imm,Dx									x
asl.l <ea>y,Dx		x						x	
asr.l <ea>y,Dx		x						x	
bcc.{b,w,l}					x				
bchg {Dy #imm},<ea>x		x			x				
bclr {Dy #imm},<ea>x		x			x				

Table 6-5. V4 ColdFire Compute Engine Location (Continued)

Instruction <sup>1</sup>	<ea> Oag	= Rn Ex	Either	<ea> Oag	= Mem Ex	Either	<ea> Oag	= Imm Ex	Either
bra.{b,w,l}				x					
bset {Dy #imm},<ea>x		x			x				
bsr.{b,w,l}				x					
btst {Dy #imm},<ea>x		x			x				
clr.b <ea>x		x			x				
clr.w <ea>x		x			x				
clr.l <ea>x	x				x				
cmp.b <ea>y,Rx		x			x			x	
cmp.w <ea>y,Rx		x			x			x	
cmp.l <ea>y,Rx			x		x				x
cmpi.b #imm,Dx								x	
cmpi.w #imm,Dx								x	
cmpi.l #imm,Dx									x
cpushl <ea>y					x				
divs.w <ea>y,Dx		x			x			x	
divs.l <ea>y,Dx		x			x				
divu.w <ea>y,Dx		x			x			x	
divu.l <ea>y,Dx		x			x				
eor.l Dy,<ea>x			x		x				
eori.l #imm,Dx									x
ext.{w,l} Dx		x							
extb.l Dx		x							
halt									
illegal					x				
intouch									
jmp				x					
jsr				x					
lea <ea>y,Ax	x								
link.w Ay,#imm	x								
lsl.l <ea>y,Dx		x						x	
lsr.l <ea>y,Dx		x						x	
mac.w <ea>y,Rx,Accx		x			x				
mac.l <ea>y,Rx,Accx		x			x				
msac.w <ea>y,Rx,Accx		x			x				
msac.l <ea>y,Rx,Accx		x			x				

Table 6-5. V4 ColdFire Compute Engine Location (Continued)

Instruction <sup>1</sup>	<ea> Oag	= Rn Ex	Either	<ea> Oag	= Mem Ex	Either	<ea> Oag	= Imm Ex	Either
mov3q.l #imm,<ea>x	x				x				
movclr.l Accy,Rx		x							
move.b <ea>y,Dx		x			x			x	
move.b Dy,<ea>x					x				
move.b <mem>y,<mem>x					x				
move.w <ea>y,Dx		x			x			x	
move.w <ea>y,Ax		x			x		x		
move.w Ry,<ea>x					x				
move.w <ea>y,<ea>x					x				
move.l <ea>y,Rx			x		x		x		
move.l Ry,<ea>x					x				
move.l <ea>y,<ea>x					x				
move.l <ea>y,Accx		x						x	
move.l <ea>y,Mac.CR		x						x	
move.l Accy,Rx		x							
move.l Mac.CR,Rx		x							
move.w CCR,Dx		x							
move.w Dy,CCR		x							
move.w SR,Dx		x							
move.w <ea>y,SR		x						x	
movec Ry,Rc					x				
movem.l <ea>y,#list					x				
movem.l #list,<ea>x					x				
moveq #imm,Dx							x		
muls.w <ea>y,Dx		x			x			x	
muls.l <ea>y,Dx		x			x				
mulu.w <ea>y,Dx		x			x			x	
mulu.l <ea>y,Dx		x			x				
mvs.{b,w} <ea>y,Dx		x			x			x	
mvz.{b,w} <ea>y,Dx		x			x			x	
neg.l Dx			x						
negx.l Dx		x							
nop									
not.l Dx			x						
or.l <ea>y,Dx			x		x				x



Table 6-5. V4 ColdFire Compute Engine Location (Continued)

Instruction <sup>1</sup>	<ea> Oag	= Rn Ex	Either	<ea> Oag	= Mem Ex	Either	<ea> Oag	= Imm Ex	Either
or.l Dy,<ea>x					x				
ori.l #imm,Dx									x
pea <ea>y					x				
pulse					x				
rems.l <ea>y,Dx		x			x				
remu.l <ea>y,Dx		x			x				
rte					x				
rts					x				
sats.l Dx		x							
scc Dx		x							
stop #imm								x	
sub.l <ea>y,Rx			x		x				x
sub.l Dy,<ea>x					x				
subi.l #imm,Dx									x
subq.l #imm,<ea>x			x		x				
subx.l Dy,Dx		x							
swap Dx		x							
tas <ea>x					x				
tpf									
tpf.{w,l}									
trap #imm					x				
tst.{b,w,l} <ea>x		x			x			x	
unlk Ax	x								
wddata.{b,w,l} <ea>y					x				
wdebug <ea>y					x				

<sup>1</sup> All EMAC and FPU instructions are executed in the Ex pipeline stage.

## 6.5 Instruction Execution Times

The timing data in this section assumes the following:

- Execution times for individual instructions make no assumptions concerning the OEP's ability to dispatch multiple instructions in one machine cycle. For sequences where instruction pairs are issued, the execution time of the first instruction defines the execution time of pair; the second instruction effectively executes in zero cycles.

- The OEP is loaded with the opword and all required extension words at the beginning of each instruction execution. This implies that the OEP spends no time waiting for the IFP to supply opwords or extension words.
- The OEP experiences no sequence-related pipeline stalls. For the V4, the most common example of this type of stall occurs when a register is modified in the EX engine and a subsequent instruction generates an address that uses the previously modified register. The second instruction stalls in the OEP until the previous instruction updates the register. For example:

```

muls.l  #<data>,d0
move.l  (a0,d0.l*4),d1

```

move.l waits 3 cycles for the muls.l to update d0. If consecutive instructions update a register and use that register as a base of index value with a scale factor of 1 ( $Xi.l*1$ ) in an address calculation, a 2-cycle pipeline stall occurs. If the destination register is used as an index register with any other scale factor ( $Xi.l*2$ ,  $Xi.l*4$ ), a 3-cycle stall occurs. Table 6-3 lists instructions optimized to prevent such stalls.

### NOTE:

Address register results from postincrement and predecrement modes are available to subsequent instructions without stalls.

- The OEP can complete all memory accesses without memory causing any stalls. Thus, these timings assume an infinite, zero-wait state memory attached to the core.
- Operand accesses are assumed to be aligned as follows:
  - 16-bit operands are aligned on 0-modulo-2 addresses
  - 32-bit operands are aligned on 0-modulo-4 addresses

Operands that do not meet these guidelines are misaligned. Table 6-6 shows how the core decomposes a misaligned operand reference into a series of aligned accesses.

**Table 6-6. Misaligned Operand References**

A[1:0]	Size	Bus Operations	Additional C(R/W) <sup>1</sup>
x1	Word	Byte, Byte	2(1/0) if read 1(0/1) if write
x1	Long	Byte, Word, Byte	3(2/0) if read 2(0/2) if write
10	Long	Word, Word	2(1/0) if read 1(0/1) if write

<sup>1</sup> Each timing entry is presented as C(r/w), described as follows:

C is the number of processor clock cycles, including all applicable operand fetches and writes, as well as all internal core cycles required to complete the instruction execution.

r/w is the number of operand reads (r) and writes (w) required by the instruction. An operation performing a read-modify write function is denoted as (1/1).

## 6.5.1 MOVE Instruction Execution Times

The following tables show execution times for the MOVE.{B,W,L} instructions. Table 6-9 shows the timing for the other generic move operations.

### NOTE:

In these tables, times using PC-relative effective addressing modes are the same as using An-relative mode.

ET with {<ea> = (d16,PC)} equals ET with {<ea> = (d16,An)}

ET with {<ea> = (d8,PC,Xi\*SF)} equals ET with {<ea> = (d8,An,Xi\*SF)}

The (xxx).wl nomenclature refers to both forms of absolute addressing, (xxx).w and (xxx).l.

Table 6-7 lists execution times for MOVE.{B,W} instructions.

**Table 6-7. Move Byte and Word Execution Times**

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	(xxx).wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(Ay)+	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
-(Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(d16,Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,Ay,Xi*SF)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(xxx).w	1(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(xxx).l	1(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(d16,PC)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,PC,Xi*SF)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
#<xxx>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	—	—

Table 6-8 lists timings for MOVE.L.

**Table 6-8. Move Long Execution Times**

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	(xxx).wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(Ay)+	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)

**Table 6-8. Move Long Execution Times (Continued)**

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	(xxx).wl
-(Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(d16,Ay)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,Ay,Xi*SF)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(xxx).w	1(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(xxx).l	1(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(d16,PC)	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,PC,Xi*SF)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
#<xxx>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	—	—	—

Table 6-9 gives timings for MOVE.L instructions accessing program-visible MAC registers, along with other MOVE.L timings. Execution times for moving ACC or MACSR contents into a destination location represent the best-case scenario when the store instruction is executed and no load, MAC, or MSAC instructions are in the MAC execution pipeline. In general, these store operations take only 1 cycle to execute, but if preceded immediately by a load, MAC, or MSAC instruction, the MAC pipeline depth is exposed and execution time is 3 cycles.

Table 6-9 and Table 6-11 apply only to the MAC. Table 6-15 lists EMAC execution times.

**Table 6-9. MAC and Miscellaneous Move Execution Times**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
move.l	<ea>,ACC	1(0/0)	—	—	—	—	—	—	1(0/0)
move.l	<ea>,MACSR	6(0/0)	—	—	—	—	—	—	6(0/0)
move.l	<ea>,MASK	5(0/0)	—	—	—	—	—	—	5(0/0)
move.l	ACC,Rx	1(0/0)	—	—	—	—	—	—	—
move.l	MACSR,CCR	1(0/0)	—	—	—	—	—	—	—
move.l	MACSR,Rx	1(0/0)	—	—	—	—	—	—	—
move.l	MASK,Rx	1(0/0)	—	—	—	—	—	—	—
moveq	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
mov3q	#imm,<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	—
mvs	<ea>,Dx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
mvz	<ea>,Dx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)

## 6.5.2 Execution Timings—One-Operand Instructions

Table 6-10 shows standard timings for single-operand instructions.

**Table 6-10. One-Operand Instruction Execution Times**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#xxx
clr.b	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
clr.w	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
clr.l	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
ext.w	Dx	1(0/0)	—	—	—	—	—	—	—
ext.l	Dx	1(0/0)	—	—	—	—	—	—	—
extb.l	Dx	1(0/0)	—	—	—	—	—	—	—
neg.l	Dx	1(0/0)	—	—	—	—	—	—	—
negx.l	Dx	1(0/0)	—	—	—	—	—	—	—
not.l	Dx	1(0/0)	—	—	—	—	—	—	—
sats.l	Dx	1(0/0)	—	—	—	—	—	—	—
scc	Dx	1(0/0)	—	—	—	—	—	—	—
swap	Dx	1(0/0)	—	—	—	—	—	—	—
tas	<ea>	1(1/1)	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
tst.b	<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
tst.w	<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
tst.l	<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)

### 6.5.3 Execution Timings—Two-Operand Instructions

Table 6-11 shows standard timings for double operand instructions.

**Table 6-11. Two-Operand Instruction Execution Times**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
add.l	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
add.l	Dy,<ea>	—	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
addi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
addq.l	#imm,<ea>	1(0/0)	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
addx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—
and.l	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
and.l	Dy,<ea>	—	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
andi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
asl.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
asr.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
bchg	Dy,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)	—

**Table 6-11. Two-Operand Instruction Execution Times (Continued)**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
bchg	#imm,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—	—
bclr	Dy,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)	—
bclr	#imm,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—	—
bset	Dy,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)	—
bset	#imm,<ea>	2(0/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—	—
btst	Dy,<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	—
btst	#imm,<ea>	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	—	—	—
cmp.b	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
cmp.w	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
cmp.l	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
cmpi.b	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
cmpi.w	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
cmpi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
divs.w	<ea>,Dx	20(0/0)	20(1/0)	20(1/0)	20(1/0)	20(1/0)	21(1/0)	20(1/0)	20(0/0)
divu.w	<ea>,Dx	20(0/0)	20(1/0)	20(1/0)	20(1/0)	20(1/0)	21(1/0)	20(1/0)	20(0/0)
divs.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
divu.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
eor.l	Dy,<ea>	1(0/0)	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
eori.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
lea	<ea>,Ax	—	1(0/0)	—	—	1(0/0)	2(0/0)	1(0/0)	—
lsl.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
lsr.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
mac.w	Ry,Rx	1(0/0)	—	—	—	—	—	—	—
mac.l	Ry,Rx	3(0/0)	—	—	—	—	—	—	—
msac.w	Ry,Rx	1(0/0)	—	—	—	—	—	—	—
msac.l	Ry,Rx	3(0/0)	—	—	—	—	—	—	—
mac.w	Ry,Rx,ea,Rw	—	1(1/0)	1(1/0)	1(1/0)	1(1/0)	—	—	—
mac.l	Ry,Rx,ea,Rw	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
msac.w	Ry,Rx,ea,Rw	—	1(1/0)	1(1/0)	1(1/0)	1(1/0)	—	—	—
msac.l	Ry,Rx,ea,Rw	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
muls.w	<ea>,Dx	3(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	3(0/0)
mulu.w	<ea>,Dx	3(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	3(0/0)
muls.l	<ea>,Dx	5(0/0)	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
mulu.l	<ea>,Dx	5(0/0)	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
or.l	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)

**Table 6-11. Two-Operand Instruction Execution Times (Continued)**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
or.l	Dy,<ea>	—	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
or.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
rems.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
remu.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
sub.l	<ea>,Rx	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	1(0/0)
sub.l	Dy,<ea>	—	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
subi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
subq.l	#imm,<ea>	1(0/0)	1(1/1)	1(1/1)	1(1/1)	1(1/1)	2(1/1)	1(1/1)	—
subx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—

## 6.5.4 Miscellaneous Instruction Execution Times

Table 6-12 lists timings for miscellaneous instructions.

**Table 6-12. Miscellaneous Instruction Execution Times**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
cpushl	(Ax)	—	9(0/1)	—	—	—	—	—	—
intouch	(Ay)	—	19(1/0)	—	—	—	—	—	—
link.w	Ay,#imm	2(0/1)	—	—	—	—	—	—	—
move.w	CCR,Dx	1(0/0)	—	—	—	—	—	—	—
move.w	<ea>,CCR	1(0/0)	—	—	—	—	—	—	1(0/0)
move.w	SR,Dx	1(0/0)	—	—	—	—	—	—	—
move.w	<ea>,SR	4(0/0)	—	—	—	—	—	—	4(0/0)
movec	Ry,Rc	20(0/1)	—	—	—	—	—	—	—
movem.l <sup>1</sup>	<ea>,&list	—	n(n/0)	—	—	n(n/0)	—	—	—
movem.l	&list,<ea>	—	n(0/n)	—	—	n(0/n)	—	—	—
nop		6(0/0)	—	—	—	—	—	—	—
pea	<ea>	—	1(0/1)	—	—	1(0/1) <sup>2</sup>	2(0/1) <sup>3</sup>	1(0/1)	—
pulse		1(0/0)	—	—	—	—	—	—	—
stop	#imm	—	—	—	—	—	—	—	6(0/0) <sup>4</sup>
trap	#imm	—	—	—	—	—	—	—	18(1/2)
tpf		1(0/0)	—	—	—	—	—	—	—
tpf.w		1(0/0)	—	—	—	—	—	—	—
tpf.l		1(0/0)	—	—	—	—	—	—	—

**Table 6-12. Miscellaneous Instruction Execution Times (Continued)**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
unlk	Ax	1(1/0)	—	—	—	—	—	—	—
wddata.l	<ea>	—	1(1/0)	1(1/0)	1(1/0)	1(1/0)	2(1/0)	1(1/0)	—
wdebug.l	<ea>	—	3(2/0)	—	—	3(2/0)	—	—	—

<sup>1</sup> *n* is the number of registers moved by the MOVEM opcode.

<sup>2</sup> PEA execution times are the same for (d16,PC).

<sup>3</sup> PEA execution times are the same for (d8,PC,Xi\*SF).

<sup>4</sup> The execution time for STOP is the time required until the processor begins sampling continuously for interrupts.

## 6.5.5 Branch Instruction Execution Times

Table 6-13 shows general branch instruction timing.

**Table 6-13. General Branch Instruction Execution Times**

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
bra		—	—	—	—	1(0/1) <sup>1</sup>	—	—	—
bsr		—	—	—	—	1(0/1) <sup>1</sup>	—	—	—
jmp	<ea>	—	5(0/0)	—	—	5(0/0) <sup>1</sup>	6(0/0)	1(0/0) <sup>1</sup>	—
jsr	<ea>	—	5(0/1)	—	—	5(0/1)	6(0/1)	1(0/1) <sup>1</sup>	—
rte		—	—	15(2/0)	—	—	—	—	—
rts		—	—	2(1/0) <sup>2</sup> 9(1/0) <sup>3</sup> 8(1/0) <sup>4</sup>	—	—	—	—	—

<sup>1</sup> Assumes branch acceleration. Depending on the pipeline status, execution times may vary from 1 to 3 cycles.

<sup>2</sup> If predicted correctly by the hardware return stack.

<sup>3</sup> If mispredicted by the hardware return stack.

<sup>4</sup> If not predicted by the hardware return stack.

Table 6-14 shows timing for Bcc instructions.

**Table 6-14. Bcc Instruction Execution Times**

Opcode	Branch Cache Correctly Predicts Taken	Prediction Table Correctly Predicts Taken	Predicted Correctly as Not Taken	Predicted Incorrectly
bcc	0(0/0)	1(0/0)	1(0/0)	8(0/0)

## 6.5.6 EMAC Instruction Execution Times

Table 6-15 specifies instruction execution times associated with the enhanced multiply-accumulate (EMAC) execute engine.



Table 6-15. EMAC Instruction Execution Times

Opcode	<ea>y	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xi*SF) (d8,PC,Xi*SF)	xxx.wl	#xxx
mac.l	Ry,Rx,ACCx	1(0/0)	—	—	—	—	—	—	—
mac.l	Ry,Rx,<ea>,Rw,ACCx	—	1(1/0)	1(1/0)	1(1/0)	1(1/0) <sup>1</sup>	—	—	—
mac.w	Ry,Rx,ACCx	1(0/0)	—	—	—	—	—	—	—
mac.w	Ry,Rx,<ea>,Rw,ACCx	—	1(1/0)	1(1/0)	1(1/0)	1(1/0) <sup>1</sup>	—	—	—
mov.l	<ea>y,ACCx	1(0/0)	—	—	—	—	—	—	1(0/0)
mov.l	ACCy,ACCx	1(0/0)	—	—	—	—	—	—	—
mov.l	<ea>y,MACSR	8(0/0)	—	—	—	—	—	—	8(0/0)
mov.l	<ea>y,MASK	7(0/0)	—	—	—	—	—	—	7(0/0)
mov.l	<ea>y,ACCext01	1(0/0)	—	—	—	—	—	—	1(0/0)
mov.l	<ea>y,ACCext23	1(0/0)	—	—	—	—	—	—	1(0/0)
mov.l	ACCx,<ea>x	1(0/0) <sup>2</sup>	—	—	—	—	—	—	—
mov.l	MACSR,<ea>x	1(0/0)	—	—	—	—	—	—	—
mov.l	MASK,<ea>x	1(0/0)	—	—	—	—	—	—	—
mov.l	ACCext01,<ea>x	1(0/0)	—	—	—	—	—	—	—
mov.l	ACCext23,<ea>x	1(0/0)	—	—	—	—	—	—	—
msac.l	Ry,Rx,ACCx	1(0/0)	—	—	—	—	—	—	—
msac.l	Ry,Rx,<ea>,Rw,ACCx	—	1(1/0)	1(1/0)	1(1/0)	1(1/0) <sup>1</sup>	—	—	—
msac.w	Ry,Rx,ACCx	1(0/0)	—	—	—	—	—	—	—
msac.w	Ry,Rx,<ea>,Rw,ACCx	—	1(1/0)	1(1/0)	1(1/0)	1(1/0) <sup>1</sup>	—	—	—
muls.l	<ea>y,Dx	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	—	—	—
muls.w	<ea>y,Dx	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	4(0/0)
mulu.l	<ea>y,Dx	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	—	—	—
mulu.w	<ea>y,Dx	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	4(0/0)

<sup>1</sup> Effective address of (d16,PC) not supported.

<sup>2</sup> Storing the accumulator requires 1 additional clock cycle when saturation is enabled, or fractional rounding is performed (MACSR[7:4] = 1---, -11-, --11).

Execution times for moving the contents of the ACC, ACCext[01,23], MACSR, or MASK into a destination location <ea>x in this table represent the best-case scenario when the store is executed and no load, copy, MAC, or MSAC instructions are in the EMAC execution pipeline. In general, these store operations require only a single cycle for execution, but if preceded immediately by a load, copy, MAC, or MSAC instruction, the depth of the EMAC pipeline is exposed and the execution time is 4 cycles.

## 6.5.7 FPU Instruction Execution Times

Table 6-16 specifies the instruction execution times associated with the FPU execute engine.

**Table 6-16. FPU Instruction Execution Times<sup>1, 2</sup>**

Opcode	Format	Effective Address <ea>						
		FPn	Dn	(An)	(An)+	-(An)	(d <sub>16</sub> ,An)	(d <sub>16</sub> ,PC)
fabs	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
fadd	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
fbcc	<label>	—	—	—	—	—	—	2(0/0) if correct, 9(0/0) if incorrect
fcmp	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
fdiv	<ea>y,FPx	23(0/0)	23(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)
fint	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
fintrz	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
fmove	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
fmove	FPy,<ea>x	—	2(0/1)	2(0/1)	2(0/1)	2(0/1)	2(0/1)	—
fmove	<ea>y,FP*R	—	6(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)
fmove	FP*R,<ea>x	—	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	—
fmovem <sup>3</sup>	<ea>y,#list	—	—	2n(2n/0)	—	—	2n(2n/0)	2n(2n/0)
fmovem <sup>3, 4</sup>	#list,<ea>x	—	—	1+2n(0/2n)	—	—	1+2n(0/2n)	—
fmul	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
fneg	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)
fnop		—	—	—	—	—	—	2(0/0)
frestore	<ea>y	—	—	6(4/0)	—	—	6(4/0)	6(4/0)
fsave	<ea>x	—	—	7(0/3)	—	—	7(0/3)	—
fsqrt	<ea>y,FPx	56(0/0)	56(0/0)	56(1/0)	56(1/0)	56(1/0)	56(1/0)	56(1/0)
fsub	<ea>y,FPx	4(0/0)	4(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)
ftst	<ea>y,FPx	1(0/0)	1(0/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)	1(1/0)

<sup>1</sup> Add 1(1/0) for an external read operand of double-precision format for all instructions except FMOVEM, and 1(0/1) for FMOVE FPy,<ea>x when the destination is double-precision.

<sup>2</sup> If the external operand is an integer format (byte, word, or longword), there is a 4-cycle conversion time that must be added to the basic execution time.

<sup>3</sup> For FMOVEM, *n* refers to the number of registers being moved.

<sup>4</sup> If any exceptions are enabled, the execution time for FMOVE FPy,<ea>x increases by 1 cycle. If the BSUN exception is enabled, the execution time for FBcc increases by one cycle.

# Chapter 7

## Exception Processing

This chapter describes CF4e exception processing, focusing on differences from previous ColdFire versions. In particular, additional encodings have been added to the fault status (FS) field in the exception stack frame to indicate exceptions related to translation lookaside buffers (TLBs). This provides CF4e core designs with precise, recoverable faults for all K-Bus references to support demand-paged memory accesses.

### 7.1 Overview

Exception processing for ColdFire processors is streamlined for performance. Differences from previous ColdFire Family processors include the following:

- An instruction restart model for translation (TLB miss) and access faults. This new functionality extends the existing ColdFire access error fault vector and exception stack frames.
- Use of separate system stack pointers for user and supervisor modes.

Previous ColdFire processors use an instruction restart exception model but require additional software support to recover from certain access errors.

Exception processing can be defined as the time from the detection of the fault condition until the fetch of the first handler instruction has been initiated. It consists of the following four major steps:

1. The processor makes an internal copy of the status register (SR) and then enters supervisor mode by setting SR[S] and disabling trace mode by clearing SR[T]. The occurrence of an interrupt exception also clears SR[M] and sets the interrupt priority mask, SR[I] to the level of the current interrupt request.
2. The processor determines the exception vector number. For all faults except interrupts, the processor bases this calculation on exception type. For interrupts, the processor performs an interrupt acknowledge (IACK) bus cycle to obtain the vector number from peripheral. The IACK cycle is mapped to a special acknowledge address space with the interrupt level encoded in the address.
3. The processor saves the current context by creating an exception stack frame on the system stack. As a result, the exception stack frame is created at a 0-modulo-4 address on top of the system stack pointed to by the supervisor stack pointer (SSP).

As shown in Figure 7-1, the CF4e processor uses the same fixed-length stack frame as previous ColdFire Versions with additional fault status (FS) encodings to support the MMU. In some exception types, the program counter (PC) in the exception stack frame contains the address of the faulting instruction (fault); in others the PC contains the next instruction to be executed (next). (Note that previous ColdFire processors support a single stack pointer in the A7 address register.)

If the exception is caused by an FPU instruction, the PC contains the address of either the next floating-point instruction (nextFP) if the exception is pre-instruction, or the faulting instruction (fault) if the exception is post-instruction.

4. The processor acquires the address of the first instruction of the exception handler. The instruction address is obtained by fetching a value from the exception table at the address in the vector base register. The index into the table is calculated as  $4 \times \text{vector\_number}$ . When the index value is generated, the vector table contents determine the address of the first instruction of the desired handler. After the fetch of the first opcode of the handler is initiated, exception processing terminates and normal instruction processing continues in the handler.

The vector base register described in the *ColdFire PRM*, holds the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table. VBR[19–0] are not implemented and are assumed to be zero, forcing the vector table to be aligned on a 0-modulo-1-Mbyte boundary.

ColdFire processors support a 1,024-byte vector table aligned on any 0-modulo-1 Mbyte address boundary; see Table 7-1. The table contains 256 exception vectors, the first 64 of which are defined by Motorola. The rest are user-defined interrupt vectors.

**Table 7-1. Exception Vector Assignments**

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter <sup>1</sup>	Assignment
0	000	—	Initial supervisor stack pointer
1	004	—	Initial program counter
2	008	Fault	Access error
3	00C	Fault	Address error
4	010	Fault	Illegal instruction
5	014	Fault	Divide by zero
6–7	018–01C	—	Reserved
8	020	Fault	Privilege violation
9	024	Next	Trace
10	028	Fault	Unimplemented line-a opcode
11	02C	Fault	Unimplemented line-f opcode
12	030	Next	Non-PC breakpoint debug interrupt

**Table 7-1. Exception Vector Assignments (Continued)**

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter <sup>1</sup>	Assignment
13	034	Next	PC breakpoint debug interrupt
14	038	Fault	Format error
15	03C	Next	Uninitialized interrupt
16–23	040–05C	—	Reserved
24	060	Next	Spurious interrupt
25–31	064–07C	Next	Level 1–7 autovectored interrupts
32–47	080–0BC	Next	Trap #0–15 instructions
48	0C0	Fault	Floating-point branch on unordered condition
49	0C4	NextFP or Fault	Floating-point inexact result
50	0C8	NextFP	Floating-point divide-by-zero
51	0CC	NextFP or Fault	Floating-point underflow
52	0D0	NextFP or Fault	Floating-point operand error
53	0D4	NextFP or Fault	Floating-point overflow
54	0D8	NextFP or Fault	Floating-point input not-a-number (NaN)
55	0DC	NextFP or Fault	Floating-point input denormalized number
56–60	0E0–0F0	—	Reserved
61	0F4	Fault	Unsupported instruction
62–63	0F8–0FC	—	Reserved
64–255	100–3FC	Next	User-defined interrupts

<sup>1</sup> 'Fault' refers to the PC of the faulting instruction. 'Next' refers to the PC of the instruction immediately after the faulting instruction. NextFP' refers to the PC of the next floating-point instruction.

ColdFire processors inhibit sampling for interrupts during the first instruction of all exception handlers. This allows any handler to effectively disable interrupts, if necessary, by raising the interrupt mask level in the SR.

## 7.2 Supervisor/User Stack Pointers (A7 and OTHER\_A7)

The CF4e architecture supports two unique stack pointer (A7) registers—the supervisor stack pointer (SSP) and the user stack pointer (USP). This support provides the required isolation between operating modes as dictated by the virtual memory management scheme provided by the memory management unit (MMU). Note that only the SSP is used during creation of the exception stack frame.

The hardware implementation of these two programmable-visible 32-bit registers does not uniquely identify one as the SSP and the other as the USP. Rather, the hardware uses one

## Exception Stack Frame Definition

32-bit register as the currently-active A7 and the other as OTHER\_A7. Thus, the register contents are a function of the processor operating mode:

```
if SR[S] = 1
    then
        A7 = Supervisor Stack Pointer
        other_A7 = User Stack Pointer
    else
        A7 = User Stack Pointer
        other_A7 = Supervisor Stack Pointer
```

The BDM programming model supports reads and writes to A7 and OTHER\_A7 directly. It is the responsibility of the external development system to determine the mapping of (A7 and OTHER\_A7) to the two program-visible definitions (SSP and USP), based on the setting of SR[S]. This functionality is enabled by setting by the dual stack pointer enable bit CACR[DSPE]. If this bit is cleared, only the stack pointer, A7 (defined for previous ColdFire versions), is available. DSPE is zero at reset.

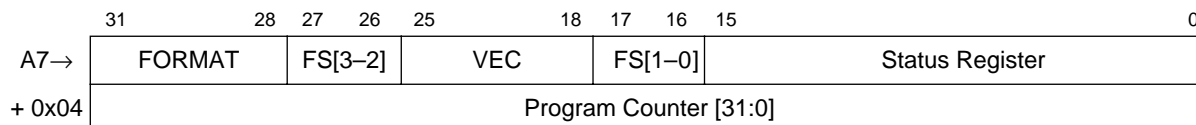
If DSPE is set, the appropriate stack pointer register (SSP or USP) is accessed as a function of the processor's operating mode. To support dual stack pointers, the following two privileged MC680x0 instructions to load/store the USP are added to the ColdFire instruction set architecture:

```
mov.l Ay,USP # move to USP: opcode = 0x4E6(0xxx)
mov.l USP,Ax # move from USP: opcode = 0x4E6(1xxx)
```

The address register number is encoded in the low-order 3 bits of the opcode.

## 7.3 Exception Stack Frame Definition

The first longword of the exception stack frame, Figure 7-1, holds the 16-bit format/vector word (F/V) and 16-bit status register. The second holds the 32-bit program counter address.



**Figure 7-1. Exception Stack Frame**

Table 7-2 describes F/V fields. FS encodings added to support the CF4e MMU are noted.

Table 7-2. Format/Vector Word

Bits	Field	Description		
31–28	FORMAT	Format field. Written with a value of {4,5,6,7} by the processor indicating a 2-longword frame format. FORMAT records any longword stack pointer misalignment when the exception occurred.		
		<b>A7 at Time of Exception, Bits[1:0]</b>	<b>A7 at First Instruction of Handler</b>	<b>FORMAT</b>
		00	Original A7—8	0100
		01	Original A7—9	0101
		10	Original A7—10	0110
		11	Original A7—11	0111
27–26	FS[3–2]	Fault status. Defined for access and address errors and for interrupted debug service routines. 0000 Not an access or address error nor an interrupted debug service routine 0001 Reserved 0010 Interrupt during a debug service routine for faults other than access errors. <sup>1</sup> 0011 Reserved 0100 Error (for example, protection fault) on instruction fetch 0101 TLB miss on opword of instruction fetch (New in CF4e) 0110 TLB miss on extension word of instruction fetch (New in CF4e) 0111 IFP access error while executing in emulator mode (New in CF4e) 1000 Error on data write 1001 Error on attempted write to write-protected space 1010 TLB miss on data write (New in CF4e) 1011 Reserved 1100 Error on data read 1101 Attempted read, read-modify-write of protected space (New in CF4e) 1110 TLB miss on data read, or read-modify-write (New in CF4e) 1111 OEP access error while executing in emulator mode (New in CF4e)		
25–18	VEC	Vector number. Defines the exception type. It is calculated by the processor for internal faults and is supplied by the peripheral for interrupts. See Table 7-1.		
17–16	FS[1–0]	See bits 27–26.		

<sup>1</sup> This generally refers to taking an I/O interrupt during a debug service routine but also applies to other fault types. If an access error occurs during a debug service routine, FS is set to 0111 if it is due to an instruction fetch or to 1111 for a data access. This applies only to access errors with the MMU present. If an access error occurs without an MMU, FS is set to 0010.

## 7.4 Processor Exceptions

Table 7-3 describes CF4e exceptions. Note that if a ColdFire processor encounters any fault while processing another fault, it immediately halts execution with a catastrophic fault-on-fault condition. A reset is required to force the processor to exit this halted state.

Table 7-3. Exceptions

Type	Description
Access error	<p>If the MMU is disabled, access errors are reported only in conjunction with an attempted store to write-protected memory. Thus, access errors associated with instruction fetch or operand read accesses are not possible. The Version 4 processor, unlike the Version 2 and 3 processors, updates the condition code register if a write-protect error occurs during a CLR or MOV3Q operation to memory.</p> <p>K-Bus accesses that fault (that is, terminated with a K-Bus transfer error acknowledge) generate an access error exception. MMU TLB misses and access violations use the same fault. If the MMU is enabled, all TLB misses and protection violations generate an access error exception. To determine if a fault is due to a TLB miss or another type of access error, new FS encodings (described in Table 7-2) signal TLB misses on the following:</p> <ul style="list-style-type: none"> <li>• Instruction fetch</li> <li>• Instruction extension fetch</li> <li>• Data read</li> <li>• Data write</li> </ul>
Address error	<p>An address error is caused by an attempted execution transferring control to an odd instruction address (that is, if bit 0 of the target address is set), an attempted use of a word-sized index register (Xi.w) or by an attempted execution of an instruction with a full-format indexed addressing mode. If an address error occurs on a JSR instruction, the Version 4 processor first pushes the return address onto the stack and then calculates the target address.</p> <p>On Version 2 and 3 processors, the target address is calculated then the return address is pushed on stack. If an address error occurs on an RTS instruction, the Version 4 processor preserves the original return PC and writes the exception stack frame above this value. On Version 2 and 3 processors, the faulting return PC is overwritten by the address error stack frame.</p>
Illegal instruction	<p>The scope of illegal instruction detection is implementation-specific across the generations of ColdFire cores. For the CF4e core, the complete 16-bit opcode is decoded and this exception is generated if execution of an unsupported instruction is attempted. Additionally, attempting to execute an illegal line A or line F opcode generates unique exception types: vectors 10 and 11, respectively. ColdFire processors do not provide illegal instruction detection on extension words of any instruction, including MOVEC. Attempting to execute an instruction with an illegal extension word causes undefined results.</p>
Divide-by-zero	<p>Attempting to divide by zero causes an exception (vector 5, offset = 0x014).</p>
Privilege violation	<p>Caused by attempted execution of a supervisor mode instruction while in user mode. The <i>ColdFire Programmer's Reference Manual</i> lists supervisor- and user-mode instructions.</p>
Trace exception	<p>Trace mode, which allows instruction-by-instruction tracing, is enabled by setting SR[T]. If SR[T] is set, instruction completion (for all but the STOP instruction) signals a trace exception. The STOP instruction has the following effects:</p> <ol style="list-style-type: none"> <li>1 The instruction before the STOP executes and then generates a trace exception. In the exception stack frame, the PC points to the STOP opcode.</li> <li>2 When the trace handler is exited, the STOP instruction is executed, loading the SR with the immediate operand from the instruction.</li> <li>3 The processor then generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the value loaded in the previous step.</li> </ol> <p>If the processor is not in trace mode and executes a STOP instruction where the immediate operand sets SR[T], hardware loads the SR and generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the value loaded in step 2. Note that because ColdFire processors do not support hardware stacking of multiple exceptions, it is the responsibility of the operating system to check for trace mode after processing other exception types. For example, when a TRAP instruction executes in trace mode, the processor initiates the TRAP exception and passes control to the corresponding handler. If the system requires a trace exception, the TRAP exception handler must check for this condition (SR[15] in the exception stack frame set) and pass control to the trace handler before returning from the original exception.</p>



Table 7-3. Exceptions (Continued)

Type	Description
Unimplemented line-a opcode	A line-a opcode results when bits 15–12 of the opword are 1010. This exception is generated by the attempted execution of an undefined line-a opcode.
Unimplemented line-f opcode	A line-f opcode results when bits 15–12 of the opword are 1111. This exception is generated under the following conditions: <ul style="list-style-type: none"> <li>• When attempting to execute an undefined line-f opcode.</li> <li>• When attempting to execute an FPU instruction when the FPU has been disabled in the CACR.</li> </ul>
Debug interrupt	The debug interrupt exception is caused by a hardware breakpoint register trigger. Rather than generating an IACK cycle, the processor internally calculates the vector number (12 or 13, depending on the type of breakpoint trigger). Additionally, SR[M,I] are unaffected by the interrupt. Separate exception vectors are provided for PC breakpoints and for address/data breakpoints. In the case of a two-level trigger, the last breakpoint determines the vector. The two unique entries occur when a PC breakpoint generates the 0x034 vector. In case of a two-level trigger, the last breakpoint event determines the vector. See Chapter 11, “Debug Support.”
Format error	When an RTE instruction executes, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire processor, attempted execution of an RTE where the format is not equal to {4, 5, 6, 7} generates a format error. The exception stack frame for the format error is created without disturbing the original exception frame and the stacked PC points to RTE. The selection of the format value provides limited debug support for porting code from M68000 applications. On M68000 Family processors, the SR was at the top of the stack. Bit 30 of the longword addressed by the system stack pointer is typically zero. Attempting an RTE using this old format generates a format error on a ColdFire processor. If the format field defines a valid type, the processor does the following: <ol style="list-style-type: none"> <li>1 Reloads the SR operand.</li> <li>2 Fetches the second longword operand.</li> <li>3 Adjusts the stack pointer by adding the format value to the auto-incremented address after the first longword fetch.</li> <li>4 Transfers control to the instruction address defined by the second longword operand in the stack frame.</li> </ol> When the processor executes a FRESTORE instruction, if the restored FPU state frame contains a non-supported value, execution is aborted and a format error exception is generated.
Trap	Executing a TRAP instruction always forces an exception and is useful for implementing system calls. The trap instruction may be used to change from user to supervisor mode.
Interrupt exception	Interrupt exception processing, with interrupt recognition and vector fetching, includes uninitialized and spurious interrupts as well as those where the requesting device supplies the 8-bit interrupt vector. Autovectoring can optionally be configured through the system interface module (SIM).
Reset exception	Asserting the reset input signal ( $\overline{\text{RSTI}}$ ) causes a reset exception, which has the highest exception priority and provides for system initialization and recovery from catastrophic failure. When assertion of $\overline{\text{RSTI}}$ is recognized, current processing is aborted and cannot be recovered. The reset exception places the processor in supervisor mode by setting SR[S] and disables tracing by clearing SR[T]. It clears SR[M] and sets SR[I] to the highest level (0b111, priority level 7). Next, VBR is cleared. Configuration registers controlling operation of all processor-local memories are invalidated, disabling the memories. Note: Implementation-specific supervisor registers are also affected at reset. After $\overline{\text{RSTI}}$ is negated, the processor waits 16 cycles before beginning the reset exception process. During this time, certain events are sampled, including the assertion of the debug breakpoint signal. If the processor is not halted, it initiates the reset exception by performing two longword read bus cycles. The longword at address 0 is loaded into the stack pointer and the longword at address 4 is loaded into the PC. After the initial instruction is fetched from memory, program execution begins at the address in the PC. If an access error or address error occurs before the first instruction executes, the processor enters a fault-on-fault halted state.
Unsupported instruction exception	If the CF4e attempts to execute a valid instruction but the required optional hardware module is not present in the OEP, a non-supported instruction exception is generated (vector 0x61). Control is then passed to an exception handler that can then process the opcode as required by the system.

## 7.5 Precise Faults

To support a demand-paged virtual memory environment, all memory references require precise, recoverable faults. The ColdFire instruction restart mechanism ensures that a faulted instruction restarts from the beginning of execution; that is, no internal state information is saved when an exception occurs and none is restored when the handler ends. Given the PC address defined in the exception stack frame, the processor reestablishes program execution by transferring control to the given location as part of the RTE (return from exception) instruction.

The instruction restart recovery model requires program-visible register changes made during execution to be undone if that instruction subsequently faults.

The Version 4 (and later) OEP structure naturally supports this concept for most instructions; program-visible registers are updated only in the final OEP stage when fault collection is complete. If any type of exception occurs, pending register updates are discarded.

For V4 cores and later, most single-cycle instructions already support precise faults and instruction restart. Some complex instructions do not. Consider the following memory-to-memory move:

```
mov.l    (Ay)+, (Ax)+    # copy 4 bytes from source to destination
```

On a Version 4 processor, this instruction takes one cycle to read the source operand (Ay) and one to write the data into Ax. Both the source and destination address pointers are updated as part of execution. Table 7-4 lists the operations performed in execute stage (EX).

**Table 7-4. OEP EX Cycle Operations**

EX Cycle	Operations
1	Read source operand from memory @ (Ay), update Ay, new Ay = old Ay + 4
2	Write operand into destination memory @ (Ax), update Ax, new Ax = old Ax + 4, update CCR

A fault detected with the destination memory write is reported during the second cycle. At this point, operations performed in the first cycle are complete, so if the destination write takes any type of access error, Ay is updated. After the access error handler executes and the faulting instruction restarts, the processor's operation is incorrect because the source address register has an incorrect (post-incremented) value.

To recover the original state of the programming model for all instructions, the CF4eCpu adds the needed hardware to support full register recovery. This hardware allows program-visible registers to be restored to their original state for multi-cycle instructions so that the instruction restart mechanism is supported. Memory-to-memory moves and move multiple loads are representative of the complex instructions needing the special recovery support.

The other major pipeline change affects the IFP. The IFP and OEP are decoupled by a FIFO

instruction buffer. In the V4 IFP, each buffer entry includes 48 bits of instruction data fetched from memory and 64 bits of early decode and branch prediction information. This datapath is expanded slightly to include IFP fault status information. Thus, every IFP access can be tagged in case an instruction fetch terminates with an error acknowledge.

**NOTE:**

For access errors signaled on instruction prefetches, an access error exception is generated only if instruction execution is attempted. If an instruction fetch access error exception is generated and the FS field indicates the fault occurred on an extension word, it may be necessary for the exception PC to be rounded-up to the next page address to determine the faulting instruction fetch address.



# Chapter 8

## Local Memory

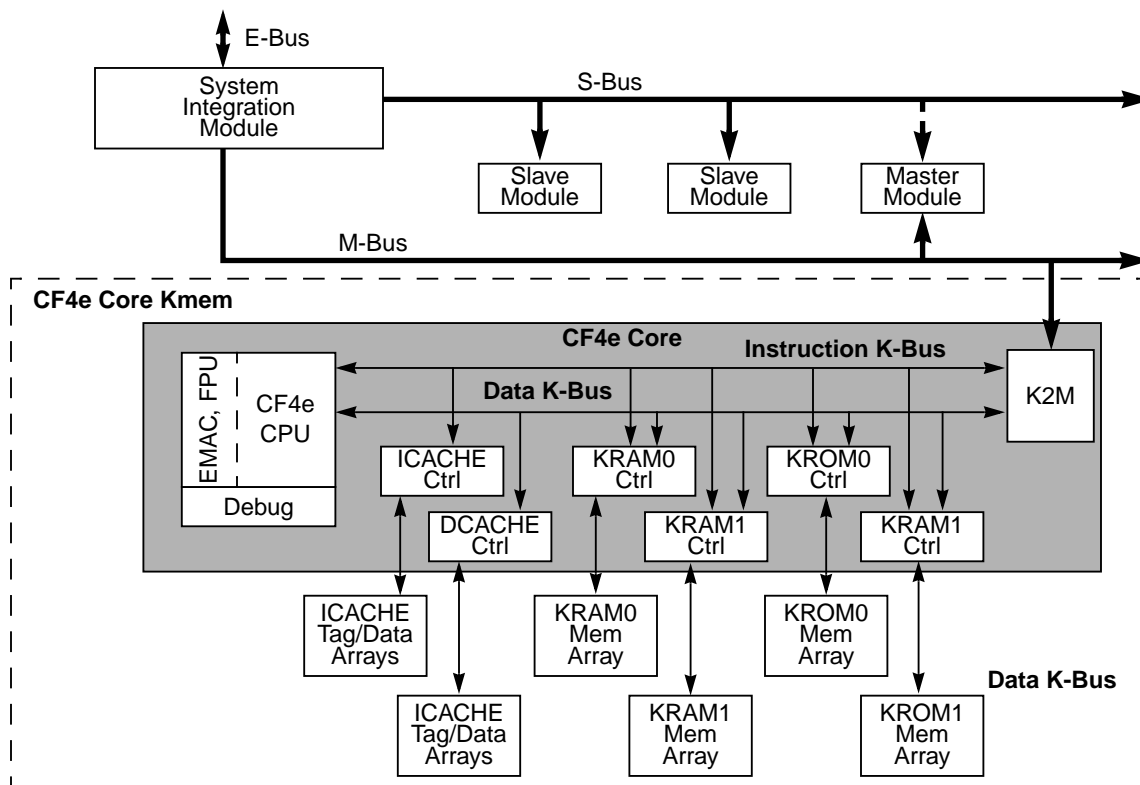
This chapter describes the implementation of the ColdFire CF4e local memory specification. The V4 local memory specification implements a Harvard memory architecture, including separate caches, ROM, RAM, and the necessary buses and registers to support instruction and data memory. This chapter consists of the following major sections.

- Section 8.5, “SRAM Overview,” describes the on-chip static RAM (SRAM) implementation. It covers general operations, configuration, and initialization. It also provides information and examples showing how to minimize power consumption when using the SRAM.
- Section 8.6, “ROM Overview,” describes the on-chip ROM implementation. It covers general operations, configuration, and initialization. It also provides information and examples showing how to minimize power consumption when using the ROM.
- Section 8.7, “Cache Overview,” describes the cache implementation, including organization, configuration, and coherency. It describes cache operations and how the caches interface with other memory structures.

### 8.1 Local Memory Overview

Figure 8-1 is a generic block diagram of a CF4e core interface.

## Local Memory Overview



**Figure 8-1. Generic CF4e Block Diagram**

The system buses have the following hierarchy:

- K-Bus—Instruction and operand; processor core and dedicated on-chip memories
- M-Bus—Internal multi-master with centralized arbitration
- S-Bus—Slave module bus controlled by the system integration module (SIM)
- E-Bus—External interface bus

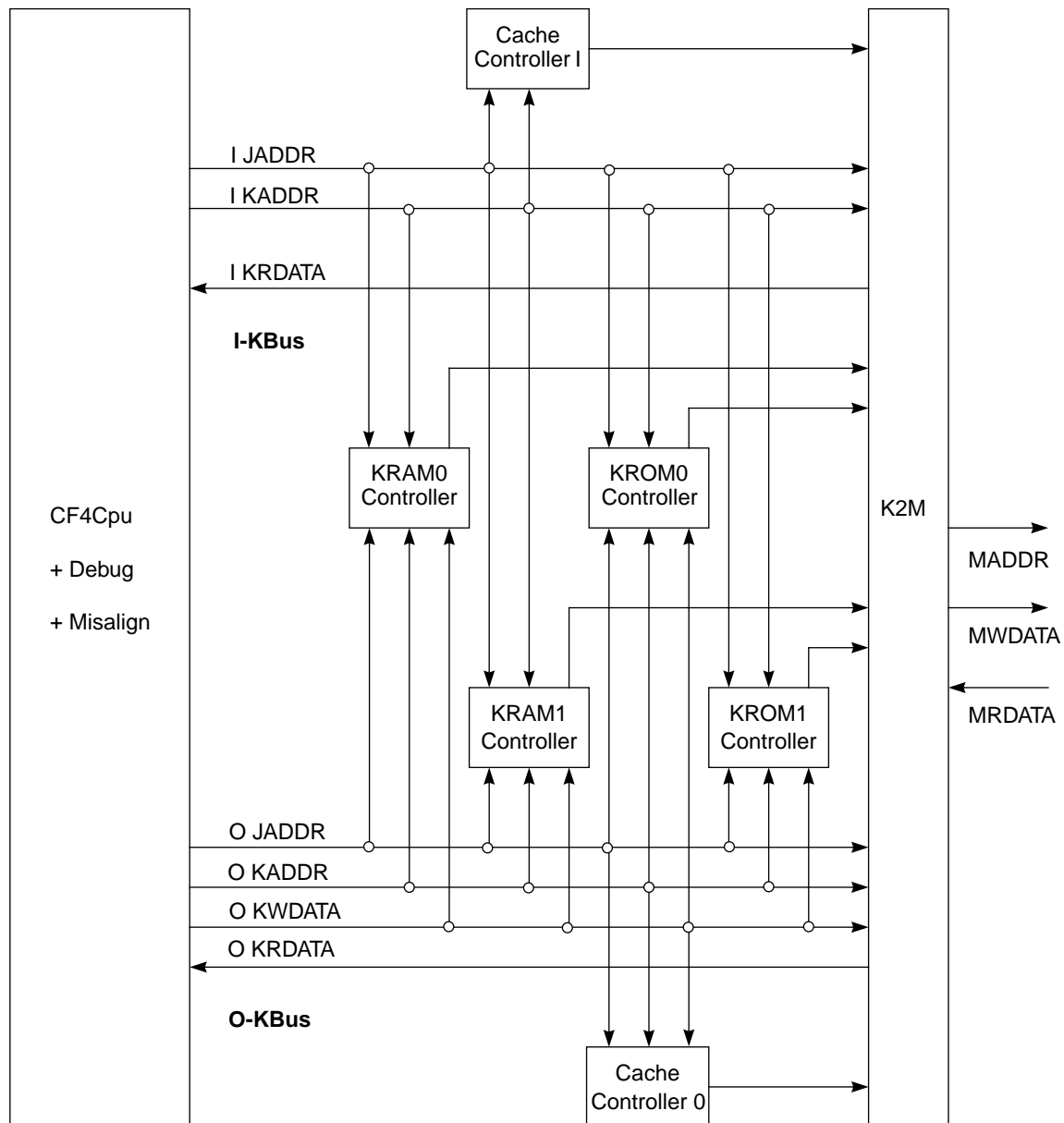
To maximize processor performance, RAM, ROM, and cache controllers reside on the high-speed local bus. These controllers support a range of memory sizes, such that when coupled with the use of compiled memory arrays, provide system designers with the ability to configure the implementation with the optimum amount of local memory for a given application.

The KROM controllers are each associated with a ROMBAR control register (KROM0 with ROMBAR0 and KROM1 with ROMBAR1). A CF4e design may have 0–2 KROM controllers. The controllers independently support array sizes of 512 bytes, and 1, 2, 4, 8, 16 or 32 Kbytes. These arrays can be configured by a bit in the appropriate ROMBAR control register to be on either the instruction K-Bus or the data K-Bus. This allows the KROM controllers and their associated arrays to be moved from one K-Bus to the other.

Input configuration signals provide the ability to automatically load the ROMBAR control registers at reset so the read-only memories can be used as boot devices.

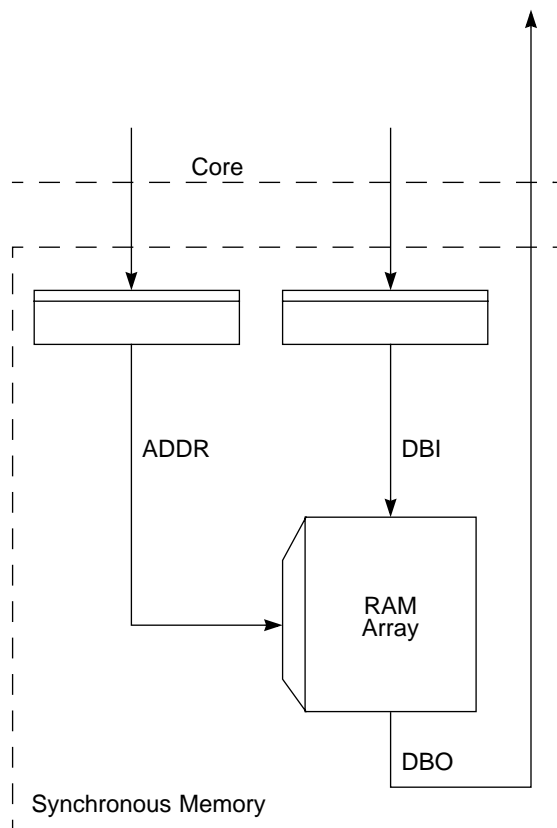
The KROM controller has an integrated BIST controller to test the KROM array.

Figure 8-2 shows the cross-bar connections involving the KRAM and KROM memories.



**Figure 8-2. Local Memory Block Diagram Showing Cache, KRAM, and KROM Controllers**

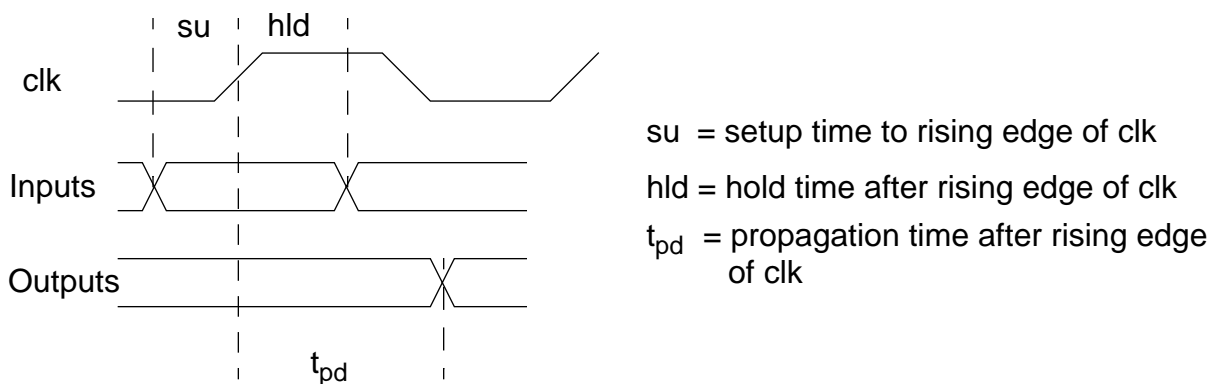
In each controller, the interface to the memory arrays is defined as a synchronous one. As shown in the following figure, the input registers for capturing the reference address and write data are specified to be internal to the memory module:



**Figure 8-3. ColdFire Core Synchronous Memory Interface**

The rectangular boxes with the double-bar at the top represent rising-edge, register storage elements, and the following signals are defined: ADDR is the reference address, DBI is the data bus input, and DBO is the data bus output.

As shown in Figure 8-4, all input signals have a setup and hold time with respect to the rising edge of the clock. All outputs transition after a propagation delay from the rising edge of the clock (clk).

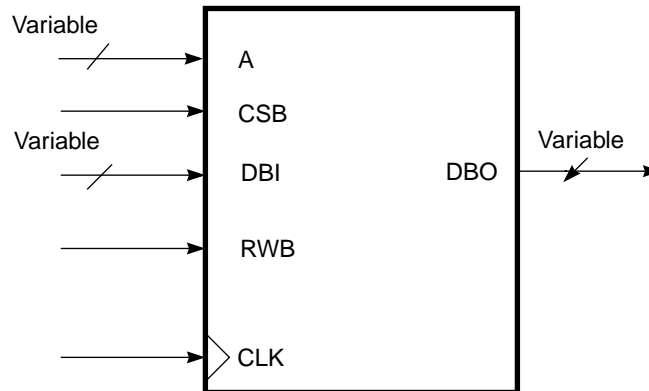


**Figure 8-4. Synchronous Memory Timing Diagram**



Memory outputs are held valid until the next rising edge of the clock.

A generic port list and synchronous memory functionality are shown in Figure 8-5.



**Figure 8-5. Synchronous Memory Interface Block Diagram**

In Figure 8-5, the memory address width is a function of the capacity of the local memory, and the data bus widths (DBI and DBO) are a function of the type of synchronous memory (cache, RAM, or ROM). The port names for the memory block are defined as follows: A is the reference address, CSB is an active-low chip select, DBI is the data bus input, RWB is the read/write control (read = 1, write = 0), CLK is the processor's clock, and DBO is the data bus output.

The corresponding functional truth table is shown in Table 8-1.

**Table 8-1. Synchronous Memory Truth Table (Sampled at Positive Edge of CLK)**

CSB	RWB	Operation
1	x	Idle (minimum power)
0	1	Read memory, DBO = Memory[A]
0	0	Write memory, Memory[A] = DBI

## 8.2 Two-Stage Pipelined Local Bus (K-Bus)

In the pipelined K-Bus design, consider a read operation. The first stage (KC1) is dedicated to the actual memory access, while the second stage (KC2) supports data transmission back to the processor. This structure provides an optimum time balance of the basic functions associated with a K-Bus reference, because it effectively provides an entire machine cycle for the memory array access.

The pipelined operation actually begins with a J cycle, where part of the reference address and certain control signals are sent from the processor to the K-Bus memory controllers in the cycle immediately preceding the KC1 stage. This transmission is necessary to allow controllers/arrays to have a local registered copy of the time-critical portion of the reference address.

## Two-Stage Pipelined Local Bus (K-Bus)

The KC1 access begins with the reference address contained in a register within the memory arrays. The memory controller performs the actual access and registers the data output for a read operation in a local data register in the controller. Thus, the entire operation is contained within the controller and the compiled memory array. During the KC2 stage, the read operand is selected from the appropriate source (cache, RAM, ROM, or the K-to-M-Bus controller) and routed back onto the K-Bus where it eventually is registered by the processor or debug module.

For operand write references, the data is sourced onto the K-Bus during the KC1 cycle, but the actual memory array update is delayed until the KC2 cycle, so that the appropriate memory unit can be identified.

To summarize, the basic pipelined K-Bus operations are shown below:

- Read
  - J: Send the low-order portion of the reference address plus certain control signals to the memories
  - KC1: Broadcast to all memories which may contain data, perform read access
  - KC2: K2M selects appropriate memory as source, and routes data back to CPU
- Write
  - J: Send the low-order portion of the reference address plus certain control signals to the memories
  - KC1: K2M signals the appropriate memory as destination, so it can capture data
  - KC2: Destination memory performs the actual write access

Given that the write strategy performs the operation during KC2, there are cases where consecutive write/read accesses may incur a 1-cycle K-Bus pipeline stall to handle the read-after-write hazard.

For cache misses or accesses that are not mapped into a K-Bus memory, the access proceeds to the KC2 stage where it is stalled as an M-Bus transfer is initiated. As the M-Bus access completes, the KC2 stall is negated and K-Bus operation is terminated.

The following block diagram presents the cache functions within the two-stage pipelined K-Bus structure:

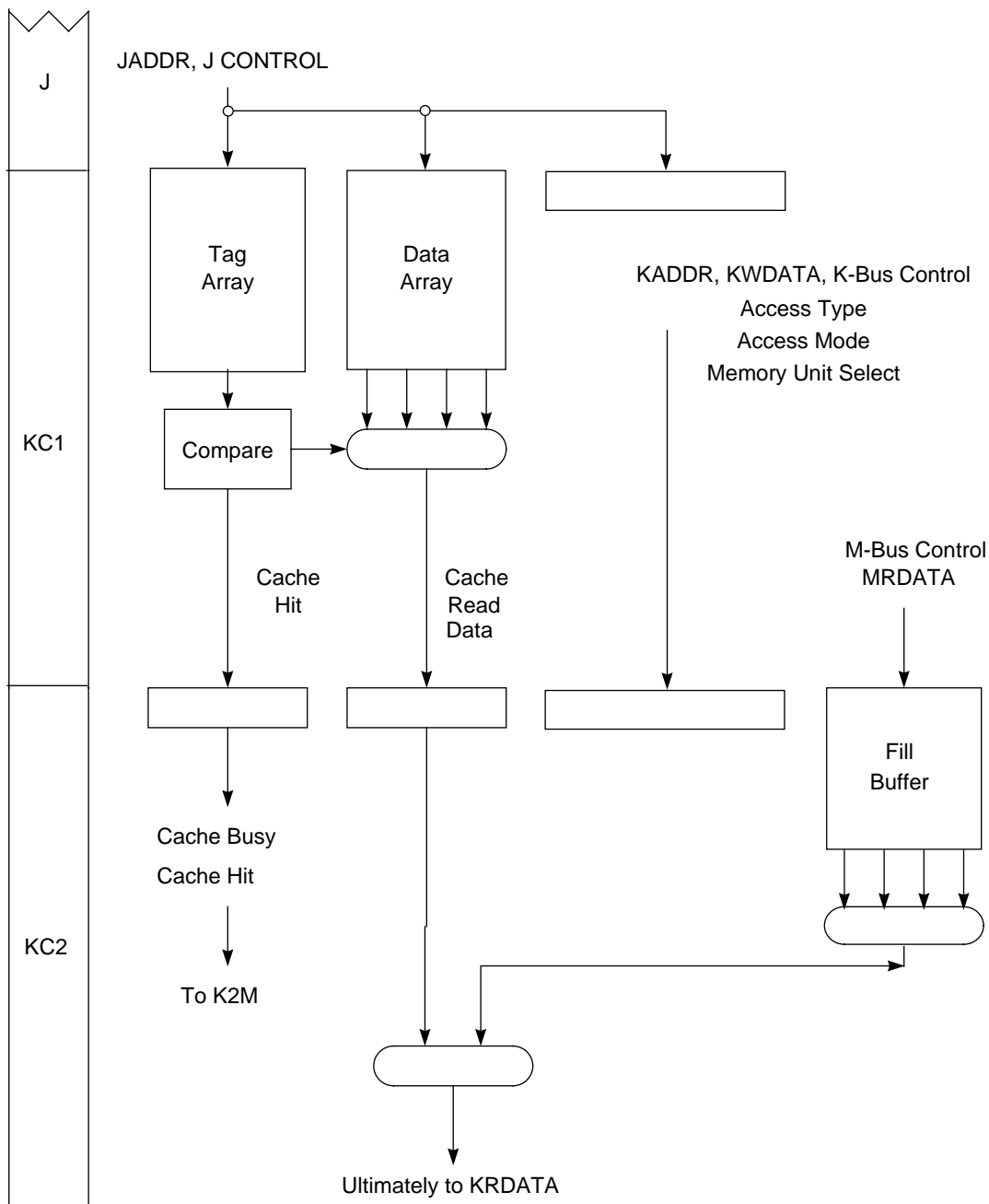


Figure 8-6. Version 4 Cache Block Diagram

### 8.3 Interactions between Local Memory Modules

Depending on configuration information, instruction fetches and data read accesses may be sent simultaneously to the SRAM, ROM, and cache controllers. This approach is required because the controllers are memory-mapped devices and the hit/miss determination is made concurrently with the read data access. Power dissipation can be minimized by configuring the ROM and SRAM base address registers (ROMBARs and RAMBARs) to mask unused address spaces whenever possible.

If the access address is mapped into the region defined by the SRAM (and this region is not masked), it provides the data back to the processor and any cache or ROM data is discarded. If the access address does not hit the SRAM, but is mapped into the region defined by the ROM (and this region is not masked), the ROM provides the data back to the processor and any cache data is discarded. Accesses from the SRAM and ROM modules are never cached. The complete definition of the processor's local bus priority scheme for read references is as follows:

The internal processor memory hierarchy uses the following priority:

1. SRAM (highest)
2. ROM
3. Cache (if space is defined as cacheable)
4. External access (lowest)

## 8.4 Local Memory Connection Specification

This section describes the how memory devices are connected and how memory sizes are configured.

### 8.4.1 K-Bus Memory Array Signal Connections

The processor supports the following six K-Bus processor local memory controllers:

- The KRAM0 and KRAM1 (SRAM) memory controllers
- the KROM0 and KROM1 (ROM) memory controllers
- Instruction cache controller
- Data cache controller.

Each controller supports a range of memory arrays, which must be synchronous SRAM or ROM structures that use the same clock as the core. The following information details the range of memory arrays supported and the necessary array connections.

#### 8.4.1.1 KRAM Information

KRAM controllers use synchronous SRAM memory arrays external to the core. These synchronous SRAMs must use the same clock as the core.

The KRAM controllers support a 32-bit array width (with byte write control) and array sizes of 512 bytes and 1, 2, 4, 8, 16, 32, and 64 Kbytes. The controller uses the `kram{0,1}size[3:0]` inputs to determine the connected array size, as shown in Table 8-2.

**Table 8-2. KRAM Size**

kram{0,1}size[3:0]	Total Size	Configuration
0000	0 bytes	KRAM{0,1} disabled
0001	512 bytes	128 x 4 bytes
0010	1 Kbytes	256 X 4 bytes
0011	2 Kbytes	512 X 4 bytes
0100	4 Kbytes	1024 X 4 bytes
0101	8 Kbytes	2048 X 4 bytes
0110	16 Kbytes	4096 X 4 bytes
0111	32 Kbytes	8192 X 4 bytes
1000	64 Kbytes	16384 X 4 bytes
1001–1111	RFU	RFU

The signals in Table 8-3 connect a KRAM controller to its SRAM array.

**Table 8-3. KRAM Memory Array Connections**

Direction/Size	Signal Name	Definition
output[15:2]	kram0addr kram1addr	KRAM0 14-bit address KRAM1 14-bit address
output[31:0]	kram0di kram1di	KRAM0 32-bit data in KRAM1 32-bit data in
output[3:0]	kram0web kram1web	KRAM0 byte write enables (active-low) KRAM1 byte write enables (active-low)
output	kram0csb kram1csb	KRAM0 chip select (active-low) KRAM1 chip select (active-low)
input[31:0]	kram0do kram1do	KRAM0 32-bit data out KRAM1 32-bit data out

KRAM memories are 32 bits wide. The kram0di[31:0] and kram1di[31:0] signals are the array input data and kram0do[31:0] and kram1do[31:0] are the array output data for all KRAM sizes.

The kram0addr[15:2] and kram1addr[15:2] signals are the array addresses. The KRAM controller provides enough address bits for the largest-supported array sizes.

The 2 low-order address bits, which are not sourced from the KRAM controller to the KRAM arrays directly, select the bytes in the 32-bit KRAM data array interface. For read operations, the KRAM controller always fetches 32 bits and the controller sends this information to the 32-bit K-Bus. The K-Bus data requester is responsible for using only the bytes selected. The KRAM controller uses byte write enables to select bytes for write operations.

Table 8-4 shows how the array address is connected for all supported KRAM array sizes.

**Table 8-4. KRAM0/KRAM1 Array Address Connection**

kram{0,1}size[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 bytes	KRAM0 disabled KRAM1 disabled	—	kram0addr[15:2] kram1addr[15:2]
0001	512 bytes	128 X 4 bytes 128 X 4 bytes	kram0addr[8:2] kram1addr[8:2]	kram0addr[15:9] kram1addr[15:9]
0010	1 Kbytes	256 X 4 bytes 256 X 4 bytes	kram0addr[9:2] kram1addr[9:2]	kram0addr[15:10] kram1addr[15:10]
0011	2 Kbytes	512 X 4 bytes 512 X 4 bytes	kram0addr[10:2] kram1addr[10:2]	kram0addr[15:11] kram1addr[15:11]
0100	4 Kbytes	1024 X 4 bytes 1024 X 4 bytes	kram0addr[11:2] kram1addr[11:2]	kram0addr[15:12] kram1addr[15:12]
0101	8 Kbytes	2048 X 4 bytes 2048 X 4 bytes	kram0addr[12:2] kram1addr[12:2]	kram0addr[15:13] kram1addr[15:13]
0110	16 Kbytes	4096 X 4 bytes 4096 X 4 bytes	kram0addr[13:2] kram1addr[13:2]	kram0addr[15:14] kram1addr[15:14]
0111	32 Kbytes	8192 X 4 bytes 8192 X 4 bytes	kram0addr[14:2] kram1addr[14:2]	kram0addr[15] kram1addr[15]
1000	64 Kbytes	16384 X 4 bytes 16384 X 4 bytes	kram0addr[15:2] kram1addr[15:2]	— —
1001–1111	RFU	RFU	RFU	RFU

The kram0csb and kram1csb signals are the chip selects for KRAM0 and KRAM1. If a chip-select signal is negated, all other KRAM signals are don't cares. If multiple array instances are used to implement the KRAM array configuration, the signal must be used as the chip enable for all instances.

The kram0web[3:0] and kram1web[3:0] signals are the byte write enables. The byte write enables correspond to the 4 data bytes in the 32-bit KRAM, as shown in Table 8-5.

**Table 8-5. KRAM0/KRAM1 Byte Write Enables**

Byte Write Enable	Array Data in Bits Controlled
kram0web[0] kram1web[0]	kram0di[31:24] kram1di[31:24]
kram0web[1] kram1web[1]	kram0di[23:16] kram1di[23:16]
kram0web[2] kram1web[2]	kram0di[15:8] kram1di[15:8]
kram0web[3] kram1web[3]	kram0di[7:0] kram1di[7:0]

#### 8.4.1.2 KROM Controller Information

The KROM controllers uses synchronous ROM memory arrays external to the core. These arrays must use the same clock as the core.

The KROM controllers support an array width of 32 bits and array sizes of 512 bytes and 1, 2, 4, 8, 16, 32, and 64 Kbytes. The controller uses the `krom{0,1}size[3:0]` inputs to determine the connected array size as shown in Table 8-6.

**Table 8-6. KRAM0/KRAM1 Size**

<code>krom{0,1}size[3:0]</code>	Total Size	Configuration
0000	0 bytes	KROM{0,1} disabled
0001	512 bytes	128 x 4 bytes
0010	1 Kbytes	256 X 4 bytes
0011	2 Kbytes	512 X 4 bytes
0100	4 Kbytes	1024 X 4 bytes
0101	8 Kbytes	2048 X 4 bytes
0110	16 Kbytes	4096 X 4 bytes
0111	32 Kbytes	8192 X 4 bytes
1000	64 Kbytes	16384 X 4 bytes
1001–1111	RFU	RFU

The signals in Table 8-7 connect a KRAM controller to its ROM array.

**Table 8-7. KROM{0,1} Memory Array Connections**

Direction/Size	Signal Name	Definition
output[15:2]	<code>krom0addr</code> <code>krom1addr</code>	KROM0 15 bit address KROM1 15 bit address
output	<code>krom0csb</code> <code>krom1csb</code>	KROM0 chip select (active-low) KROM1 chip select (active-low)
input[31:0]	<code>krom0do</code> <code>krom1do</code>	KROM0 32 bit data out KROM1 32 bit data out

KROM memories are 32 bits wide. The `krom0do[31:0]` and `krom1do[31:0]` signals are the array output data for all sizes of KRAM0 and KRAM1.

The `krom0addr[15:2]` and `krom1addr[15:2]` signals are the array addresses. Each KROM controller provides enough address bits for the largest supported array sizes.

The 2 low-order address bits, which are not sourced from the KROM controller to the KROM arrays directly, select the bytes within the 32-bit KROM data array interface. For read operations, the KROM controller always fetches 32 bits and sends this information to the 32-bit K-Bus. The K-Bus data requester is responsible for using only the bytes selected.

The array address is connected as shown in Table 8-8 for all supported KROM array sizes.

**Table 8-8. KROM Array Address Connection**

krom{0,1}size[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 bytes	KROM0 disabled KROM1 disabled	—	krom0addr[15:2] krom1addr[15:2]
0001	512 bytes	128 X 4 bytes 128 X 4 bytes	krom0addr[8:2] krom1addr[8:2]	krom0addr[15:9] krom1addr[15:9]
0010	1 Kbytes	256 X 4 bytes 256 X 4 bytes	krom0addr[9:2] krom1addr[9:2]	krom0addr[15:10] krom1addr[15:10]
0011	2 Kbytes	512 X 4 bytes 512 X 4 bytes	krom0addr[10:2] krom1addr[10:2]	krom0addr[15:11] krom1addr[15:11]
0100	4 Kbytes	1024 X 4 bytes 1024 X 4 bytes	krom0addr[11:2] krom1addr[11:2]	krom0addr[15:12] krom1addr[15:12]
0101	8 Kbytes	2048 X 4 bytes 2048 X 4 bytes	krom0addr[12:2] krom1addr[12:2]	krom0addr[15:13] krom1addr[15:13]
0110	16 Kbytes	4096 X 4 bytes 4096 X 4 bytes	krom0addr[13:2] krom1addr[13:2]	krom0addr[15:14] krom1addr[15:14]
0111	32 Kbytes	8192 X 4 bytes 8192 X 4 bytes	krom0addr[14:2] krom1addr[14:2]	krom0addr[15] krom1addr[15]
1000	64 Kbytes	16384 X 4 bytes 16384 X 4 bytes	krom0addr[15:2] krom1addr[15:2]	— —
1001–1111	RFU	RFU	RFU	RFU

The krom0csb and krom1csb signals are chip-selects for the KROMs. When a signal is inactive, all other corresponding KROM signals are don't cares. If multiple array instances are used to implement the KROM array configuration, the signal must be used as the chip enable for all instances.

### 8.4.1.3 Instruction Cache Information

The instruction cache controller uses synchronous SRAM memory arrays external to the core for its memory array needs. These arrays must use the same clock as the core.

The instruction cache design contains a non-blocking, 4-way set-associative instruction cache with a 16-byte line. Cache size is configurable with 2, 4, 8, 16, or 32 Kbyte capacities available. The cache improves system performance by providing low-latency data to the core instruction fetch pipeline, decoupling processor performance from system memory response speeds and providing increased bus availability for alternate bus masters.

The non-blocking cache services read hits from the processor while a fill (caused by a cache allocation) is in progress.

The instruction cache is virtual address indexed and physical address tagged (see Chapter 10, “Memory Management Unit (MMU),” for detailed information). If the address matches one of the cache entries, the access hits in the cache. For a read operation, the cache supplies the data to the processor. If the access does not match one of the cache entries



(misses in the cache), the K2M (K-Bus to M-Bus) controller performs a transfer on the M-Bus.

The four-way set-associative cache is organized as four levels (ways) of 32, 64, 128, 256, or 512 sets (for 2-, 4-, 8-, 16-, or 32-Kbyte cache sizes, respectively), with each line containing 16 bytes (four longwords) of storage.

Table 8-9 shows the various cache set counts, line counts, address bits, tag bits, etc., for each available cache size. For all caches sizes, a 16-byte line is used (i.e., column G, line size, is always 16 bytes and the in-line address is always A3–A0) and the level of associativity is always 4 (that is, column F, number of levels, is always 4). The number of sets (column E) is related to the number of bits in the set index by the expression number of sets equals  $2^n$  where  $n$  is the number of bits in the set index. Any address bits A31–A0 not used in the set index or the in-line address are used for the tag address (column B). Finally, the cache size can be calculated as: cache size = number of sets x number of levels x line size.

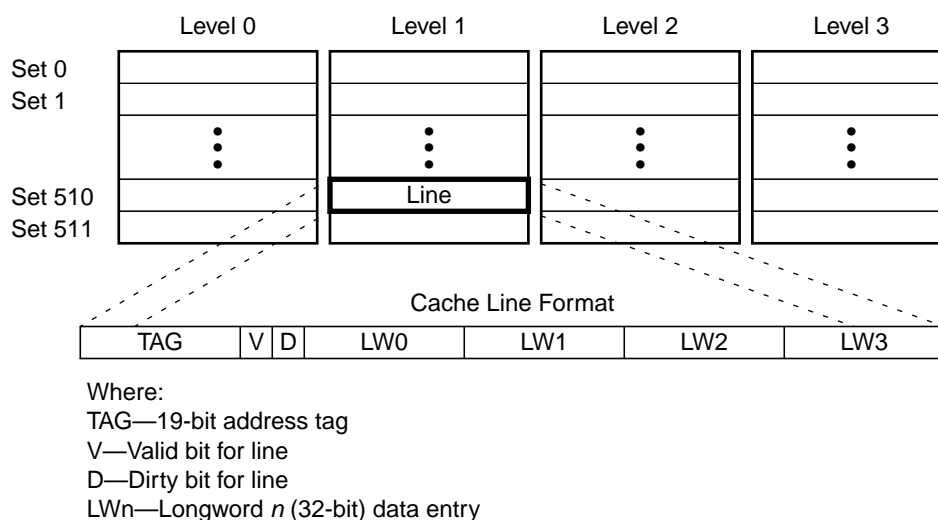
**Table 8-9. Instruction Cache Sizes and Configurations**

A	B	C	D	E	F	G
Cache Size	Tag Address	Set Index	In-line Address	Number of Sets	Number of Levels	Line Size
2 Kbytes	A31–A09	A08–A04	A03–A00	32	4	16 bytes
4 Kbytes	A31–A10	A09–A04	A03–A00	64	4	16 bytes
8 Kbytes	A31–A11	A10–A04	A03–A00	128	4	16 bytes
16 Kbytes	A31–A12	A11–A04	A03–A00	256	4	16 bytes
32 Kbytes	A31–A13	A12–A04	A03–A00	512	4	16 bytes

Address bits A[12:4] (as needed for the selected cache size) provide an index to select a set. Levels are selected according to the rules of set association.

Each line consists of an address tag (the upper 19–23 bits of the addresses needed for the selected cache size), two status bits, and four longwords of data. The two status bits consist of a valid bit (V) and a dirty bit (D) for the line. The dirty bit indicates the line was been written or modified. The instruction cache never sets this bit during normal operation but this bit must be implemented for correct array test operation. Address bits A3 and A2 select the longword within the line.

## Local Memory Connection Specification



**Figure 8-7. Cache Organization and Line Format (32-Kbyte Cache Size Shown)**

The controller uses the `icsize[3:0]` input to determine the connected array sizes, as shown in Table 8-10.

**Table 8-10. Instruction Cache Size**

icsize[3:0]	Total Size Data Array	Configuration Data Array	Total Depth Tag Array	Configuration Tag Array
0000	0 bytes	Instruction cache disabled	0 rows	Instruction cache disabled
0001	0 bytes	Instruction cache disabled	0 rows	Instruction cache disabled
0010	0 bytes	Instruction cache disabled	0 rows	Instruction cache disabled
0011	2 Kbytes	4 x 128 X 4 bytes	32 rows	32 X 25 bits
0100	4 Kbytes	4 x 256 X 4 bytes	64 rows	64 X 24 bits
0101	8 Kbytes	4 x 512 X 4 bytes	128 rows	128 X 23 bits
0110	16 Kbytes	4 x 1024 X 4 bytes	256 rows	256 X 22 bits
0111	32 Kbytes	4 x 2048 X 4 bytes	512 rows	2512 X 21 bits
1000–1111	RFU	RFU	RFU	RFU

The signals in Table 8-11 connect the instruction cache controller to its SRAM array.

**Table 8-11. Instruction Cache Memory Array Connections**

Direction/Size	Signal Name	Bus Width	Definition
Output	nsientb		Next-state instruction cache tag enable
Output	nsiwrddb		Next-state instruction cache tag write
Output	nsiwlvt	[3:0]	Next-state instruction cache tag write level
Output	nsirowst	[9:0]	Next-state instruction cache tag address
Output	nsiaddrt	[31:9]	Next-state instruction cache tag data
Output	nsisw		Next-state instruction cache tag written bit

**Table 8-11. Instruction Cache Memory Array Connections (Continued)**

Direction/Size	Signal Name	Bus Width	Definition
Output	nsisv		Next-state instruction cache tag valid bit
Output	nsiendb		Next-state instruction cache data enable
Output	nsiwrtdb	[3:0]	Next-state instruction cache data write level
Output	nsiwtbyted	[3:0]	Next-state instruction cache data byte write
Output	nsirowd	[11:0]	Next-state instruction cache data address
Output	nsicwrdata	[31:0]	Next-state instruction cache write data
Input	ictag3do	[31:9]	Instruction cache level 3 tag data output
Input	icw3do		Instruction cache level 3 written bit output
Input	icv3do		Instruction cache level 3 valid bit output
Input	ictag2do	[31:9]	Instruction cache level 2 tag data output
Input	icw2do		Instruction cache level 2 written bit output
Input	icv2do		Instruction cache level 2 valid bit output
Input	ictag1do	[31:9]	Instruction cache level 1 tag data output
Input	icw1do		Instruction cache level 1 written bit output
Input	icv1do		Instruction cache level 1 valid bit output
Input	ictag0do	[31:9]	Instruction cache level 0 tag data output
Input	icw0do		Instruction cache level 0 written bit output
Input	icv0do		Instruction cache level 0 valid bit output
Input	iclv3do	[31:0]	Instruction cache level 3 data output
Input	iclv2do	[31:0]	Instruction cache level 2 data output
Input	iclv1do	[31:0]	Instruction cache level 1 data output
Input	iclv0do	[31:0]	Instruction cache level 0 data output

All instruction cache data arrays are 32 bits wide. The nsicwrdata[31:0] signals are the data array input data and iclv0do[31:0], iclv1do[31:0], iclv2do[31:0], and iclv3do[31:0] are the data array output data for all sizes and levels of instruction cache.

All instruction cache tag arrays are 24 bits wide. The nsiaddrt[31:9] signals are the tag array input data and ictag3do[31:9], ictag2do[31:9], ictag1do[31:9], and ictag0do[31:9] are the tag array output data for all sizes of instruction cache.

The nsirowst[9:0] signals the array address for the tag arrays. The nsirowd[11:0] signals are the array address for the data arrays. The instruction cache controller provides enough address bits for the largest supported cache sizes.

The array address is connected as shown in Table 8-12 for all supported instruction cache data array sizes:

**Table 8-12. Instruction Cache Data Array Address Connection**

icsize[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 bytes	Instruction cache disabled	—	nsirowd[11:0]
0001	0 bytes	Instruction cache disabled	—	nsirowd[11:0]
0010	0 bytes	Instruction cache disabled	—	nsirowd[11:0]
0011	2 Kbytes	4 x 128 X 4 bytes	nsirowd[6:0]	nsirowd[11:7]
0100	4 Kbytes	4 x 256 X 4 bytes	nsirowd[7:0]	nsirowd[11:8]
0101	8 Kbytes	4 x 512 X 4 bytes	nsirowd[8:0]	nsirowd[11:9]
0110	16 Kbytes	4 x 512 X 4 bytes	nsirowd[9:0]	nsirowd[11:10]
0111	32 Kbytes	4 x 1024 X 4 bytes	nsirowd[10:0]	nsirowd[11]
1000–1111	RFU	RFU	RFU	RFU

The tag array has one entry for every four entries in the data array. Therefore, the tag array does not use ichadd[3:2]. The array address is connected as shown in Table 8-13 for all supported instruction cache tag array sizes.

**Table 8-13. Instruction Cache Tag Array Address Connection**

icsize[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 rows	Instruction cache disabled	—	nsirowst[9:0]
0001	0 rows	Instruction cache disabled	—	nsirowst[9:0]
0010	0 rows	Instruction cache disabled	—	nsirowst[9:0]
0011	32 rows	32 X 25 bits	nsirowst[4:0]	nsirowst[9:5]
0100	256 rows	256 X 24 bits	nsirowst[5:0]	nsirowst[9:6]
0101	512 rows	512 X 23 bits	nsirowst[6:0]	nsirowst[9:7]
0110	1024 rows	1024 X 22 bits	nsirowst[7:0]	nsirowst[9:8]
0111	2048 rows	2048 X 21 bits	nsirowst[8:0]	nsirowst[9]
1000–1111	RFU	RFU	RFU	RFU

The instruction cache controller provides enough tag array write data bits for the smallest supported cache size. Each larger cache size needs one fewer (low order) tag bit. Unnecessary tag bits may be implemented in the tag array (the cache controller always writes and reads them as 0) or they may be tied to 0 at the core boundary. The tag array write data is connected as shown in Table 8-14 for all supported cache tag array sizes.

**Table 8-14. Instruction Cache Tag Array Write Data Connection**

icsize[3:0]	Total Size	Configuration	Array Write Data	Unused Write Data (Must Be Tied to 0)
0000	0 rows	Instruction cache disabled	—	nsiaddrt[31:9]: nsisw:nsisv
0001	0 rows	Instruction cache disabled	—	nsiaddrt[31:9] nsisw:nsisv

**Table 8-14. Instruction Cache Tag Array Write Data Connection (Continued)**

icsize[3:0]	Total Size	Configuration	Array Write Data	Unused Write Data (Must Be Tied to 0)
0010	0 rows	Instruction cache disabled	—	nsiaddrt[31:9] nsisw:nsisv
0011	32 rows	32 X 25 bits	nsiaddrt[31:9]: nsisw:nsisv	----
0100	256 rows	256 X 24 bits	nsiaddrt[31:10]: nsisw:nsisv	nsiaddrt[9]
0101	512 rows	512 X 23 bits	nsiaddrt[31:11]: nsisw:nsisv	nsiaddrt[10:9]
0110	1024 rows	1024 X 22 bits	nsiaddrt[31:12]: nsisw:nsisv	nsiaddrt[11:9]
0111	2048 rows	2048 X 21 bits	nsirowst[31:13]: nsisw:nsisv	nsirowst[12:9]
1000–1111	RFU	RFU	RFU	RFU

The nsoendb signal is the chip select for the data array of the instruction cache. If nsoendb is negated, all other instruction cache data array signals are don't cares. If multiple array instances are used to implement the instruction cache data array configuration, these signals must be used as the chip enable for all instances.

The nsoentb signal is the chip select for the instruction cache tag array. If this signal is negated, all other instruction cache tag array signals are don't cares. If multiple array instances are used to implement the instruction cache tag array configuration, this signal must be used as the chip enable for all instances.

The nsowtbyted[3:0] signals are the write enables for the data array; nsowrttb is the write enable for the tag array.

#### 8.4.1.4 Data Cache Information

The data cache controller uses synchronous SRAM memory arrays external to the core for its memory array needs. These synchronous SRAM memories must use the same clock as the core.

The data cache design contains a non-blocking, 4-way set-associative data cache with a 16-byte line size. Cache size is configurable with 2-, 4-, 8-, 16-, or 32-Kbyte capacities available. The cache improves system performance by providing low-latency data to the operand fetch pipeline, decoupling processor performance from system memory response speeds, and providing increased bus availability for alternate bus masters.

The nonblocking cache services read hits from the processor while a fill (caused by a cache allocation) is in progress.

The data cache is virtual address indexed and physical address tagged (see Chapter 10, “Memory Management Unit (MMU),” for detailed information). If the address matches one

of the cache entries, the access hits in the cache. For a read operation, the cache supplies the data to the processor. If the access does not match one of the cache entries (misses in the cache) the K2M (K-Bus to M-Bus) controller performs a transfer on the M-Bus.

The four-way set-associative cache is organized as four levels (ways) of 32, 64, 128, 256, or 512 sets (for 2-, 4-, 8-, 16-, or 32-Kbyte cache sizes, respectively), with each line containing 16 bytes (four longwords) of storage.

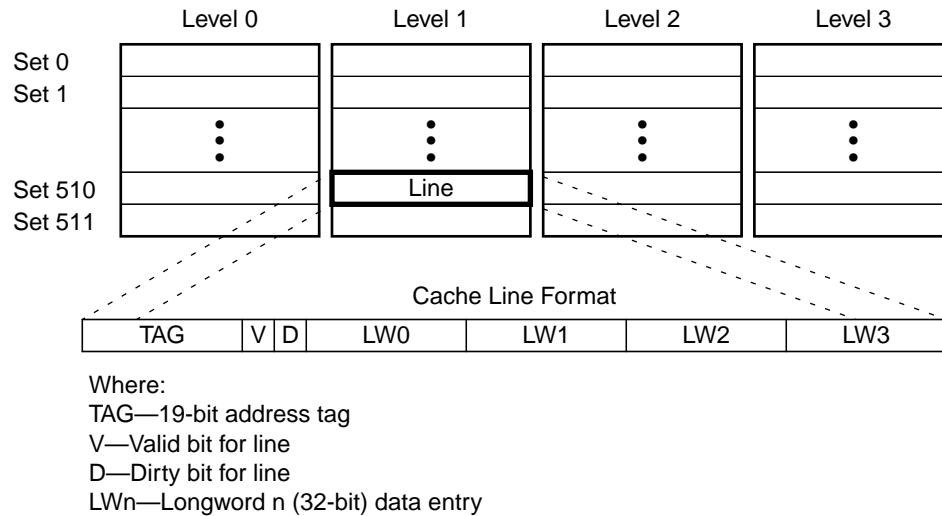
Table 8-15 shows the various cache set counts, line counts, address bits, tag bits, etc. for each available cache size. For all caches sizes, a 16-byte line size is used (i.e., column G, line size, is always 16 bytes and the in-line address is always A3–A0) and the level of associativity is always 4 (i.e., column F, number of levels, is always 4). The number of sets (column E) is related to the number of bits in the set index by the expression number of sets equals  $2^n$  where n is the number of bits in the set index. Any address bits A31–A0 not used in the set index or the in-line address are used for the tag address (column B). Finally, the cache size can be calculated as: cache size = number of sets x number of levels x line size.

**Table 8-15. Data Cache Sizes and Configurations**

A	B	C	D	E	F	G
Cache Size	Tag Address	Set Index	In-line Address	Number of Sets	Number of Levels	Line Size
2 Kbytes	A31–A09	A08–A04	A03–A00	32	4	16 bytes
4 Kbytes	A31–A10	A09–A04	A03–A00	64	4	16 bytes
8 Kbytes	A31–A11	A10–A04	A03–A00	128	4	16 bytes
16 Kbytes	A31–A12	A11–A04	A03–A00	256	4	16 bytes
32 Kbytes	A31–A13	A12–A04	A03–A00	512	4	16 bytes

Address bits A[12:4] (as needed for the selected cache size) provide an index to select a set. Levels are selected according to the rules of set association.

Each line consists of an address tag (the upper 19–23 bits of the addresses needed for the selected cache size), two status bits, and four longwords of data. The two status bits consist of a valid bit (V) and a dirty bit (D) for the line. The dirty bit indicates the line was been written or modified by an operand reference. Address bits A3 and A2 select the longword within the line.

**Figure 8-8. Cache Organization and Line Format (32 Kbyte Cache Size shown)**

The controller uses `ocsize[2:0]` to determine the connected array sizes as shown in Table 8-16.

**Table 8-16. Data Cache Size**

ocsize[3:0]	Total Size Data Array	Configuration Data Array	Total Depth Tag Array	Configuration Tag Array
0000	0 bytes	Data cache disabled	0 rows	Data cache disabled
0001	0 bytes	Data cache disabled	0 rows	Data cache disabled
0010	0 bytes	Data cache disabled	0 rows	Data cache disabled
0011	2 Kbytes	4 x 128 X 4 bytes	32 rows	32 X 24 bits
0100	4 Kbytes	4 x 256 X 4 bytes	64 rows	64 X 24 bits
0101	8 Kbytes	4 x 512 X 4 bytes	128 rows	128 X 24 bits
0110	16 Kbytes	4 x 1024 X 4 bytes	256 rows	256 X 24 bits
0111	32 Kbytes	4 x 2048 X 4 bytes	512 rows	2512 X 24 bits
1000–1111	RFU	RFU	RFU	RFU

The signals in Table 8-17 connect the data cache controller to its SRAM array:

**Table 8-17. Data Cache Memory Array Connections**

Direction/Size	Signal Name	Bus Width	Definition
Output	nsoentb		Next-state O-Cache tag enable
Output	nsowrttb		Next-state O-Cache tag write
Output	nsowlvt	[3:0]	Next-state O-Cache tag write level
Output	nsorowst	[9:0]	Next-state O-Cache tag address
Output	nsoaddrt	[31:9]	Next-state O-Cache tag data
Output	nsosw		Next-state O-Cache tag written bit

**Table 8-17. Data Cache Memory Array Connections (Continued)**

Direction/Size	Signal Name	Bus Width	Definition
Output	nsosv		Next-state O-Cache tag valid bit
Output	nsoendb		Next-state O-Cache data enable
Output	nsowrtdb	[3:0]	Next-state O-Cache data write level
Output	nsowtbyted	[3:0]	Next-state O-Cache data byte write
Output	nsorowsd	[11:0]	Next-state O-Cache data address
Output	nsocwrdata	[31:0]	Next-state O-Cache write data
Input	octag3do	[31:9]	O-Cache level 3 tag data output
Input	ocw3do		O-Cache level 3 written bit output
Input	ocv3do		O-Cache level 3 valid bit output
Input	octag2do	[31:9]	O-Cache level 2 tag data output
Input	ocw2do		O-Cache level 2 written bit output
Input	ocv2do		O-Cache level 2 valid bit output
Input	octag1do	[31:9]	O-Cache level 1 tag data output
Input	ocw1do		O-Cache level 1 written bit output
Input	ocv1do		O-Cache level 1 valid bit output
Input	octag0do	[31:9]	O-Cache level 0 tag data output
Input	ocw0do		O-Cache level 0 written bit output
Input	ocv0do		O-Cache level 0 valid bit output
Input	oclv3do	[31:0]	O-Cache level 3 data output
Input	oclv2do	[31:0]	O-Cache level 2 data output
Input	oclv1do	[31:0]	O-Cache level 1 data output
Input	oclv0do	[31:0]	O-Cache level 0 data output

All data cache data arrays are 32 bits wide. The nsocwrdata[31:0] signals are the data array input data and oclv0do[31:0], oclv1do[31:0], oclv2do[31:0], and oclv3do[31:0] are the data array output data for all sizes and levels of data cache.

All data cache tag arrays are 24 bits wide. The nsoaddr[31:9] signals are the tag array input data and octag3do[31:9], octag2do[31:9], octag1do[31:9], and octag0do[31:9] are the tag array output data for all sizes of data cache.

The nsorowst[9:0] signals are the array address for the tag arrays. The nsorowsd[11:0] signals are the array address for the data arrays. The data cache controller provides enough address bits for the largest supported cache sizes.

The array address is connected as shown in Table 8-18 for all supported data cache data array sizes.



**Table 8-18. Data Cache Data Array Address Connection**

ocsize[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 bytes	Data cache disabled	—	nsorowsd[11:0]
0001	0 bytes	Data cache disabled	—	nsorowsd[11:0]
0010	0 bytes	Data cache disabled	—	nsorowsd[11:0]
0011	2 Kbytes	4 x 128 X 4 bytes	nsorowsd[6:0]	nsorowsd[11:7]
0100	4 Kbytes	4 x 256 X 4 bytes	nsorowsd[7:0]	nsorowsd[11:8]
0101	8 Kbytes	4 x 512 X 4 bytes	nsorowsd[8:0]	nsorowsd[11:9]
0110	16 Kbytes	4 x 512 X 4 bytes	nsorowsd[9:0]	nsorowsd[11:10]
0111	32 Kbytes	4 x 1024 X 4 bytes	nsorowsd[10:0]	nsorowsd[11]
1000–1111	RFU	RFU	RFU	RFU

The tag array has one entry for every four data array entries; therefore, the tag array does not use ochadd[3:2]. The array address is connected as shown in Table 8-19 for all supported data cache tag array sizes.

**Table 8-19. Data Cache Tag Array Address Connection**

ocsize[3:0]	Total Size	Configuration	Array Address	Unused Address
0000	0 rows	Data cache disabled	—	nsorowst[9:0]
0001	0 rows	Data cache disabled	—	nsorowst[9:0]
0010	0 rows	Data cache disabled	—	nsorowst[9:0]
0011	32 rows	32 X 24 bits	nsorowst[4:0]	nsorowst[9:5]
0100	256 rows	256 X 24 bits	nsorowst[5:0]	nsorowst[9:6]
0101	512 rows	512 X 24 bits	nsorowst[6:0]	nsorowst[9:7]
0110	1024 rows	1024 X 24 bits	nsorowst[7:0]	nsorowst[9:8]
0111	2048 rows	2048 X 24 bits	nsorowst[8:0]	nsorowst[9]
1000–1111	RFU	RFU	RFU	RFU

The data cache controller provides enough tag array write data bits for the smallest supported cache size. Each larger cache size needs one fewer (low order) tag bit. Unnecessary tag bits may be implemented in the tag array (the cache controller always writes and reads them as 0) or they may be tied to 0 at the core boundary. The tag array write data is connected as shown in Table 8-20 for supported data cache tag array sizes.

**Table 8-20. Data Cache Tag Array Write Data Connection**

ocsize[3:0]	Total Size	Configuration	Array Write Data	Unused Write Data (Must Be Tied to 0)
0000	0 rows	Data cache disabled	—	nsoaddr[31:9]: nsosw:nsosv
0001	0 rows	Data cache disabled	—	nsoaddr[31:9] nsosw:nsosv

**Table 8-20. Data Cache Tag Array Write Data Connection**

ocsize[3:0]	Total Size	Configuration	Array Write Data	Unused Write Data (Must Be Tied to 0)
0010	0 rows	Data cache disabled	—	nsoaddrt[31:9] nsosw:nsosv
0011	32 rows	32 X 25 bits	nsoaddrt[31:9]: nsosw:nsosv	----
0100	256 rows	256 X 24 bits	nsoaddrt[31:10]: nsosw:nsosv	nsoaddrt[9]
0101	512 rows	512 X 23 bits	nsoaddrt[31:11]: nsosw:nsosv	nsoaddrt[10:9]
0110	1024 rows	1024 X 22 bits	nsoaddrt[31:12]: nsosw:nsosv	nsoaddrt[11:9]
0111	2048 rows	2048 X 21 bits	nsorowst[31:13]: nsosw:nsosv	nsorowst[12:9]
1000–1111	RFU	RFU	RFU	RFU

The nsoendb signal is the chip select for the data array of the data cache. If this signal is negated, all other data cache data array signals are don't cares. If multiple array instances are used to implement the data cache data array configuration, this signal must be used as the chip enable for all instances.

The nsoentb signal is the chip select for the instruction cache tag array. If this signal is negated, all other data cache tag array signals are don't cares. If multiple array instances are used to implement the data cache tag array configuration, this signal must be used as the chip enable for all instances.

The nsowtbyted[3:0] signals are write enables for the data array; nsowrttb is the write enable for the tag array.

## 8.5 SRAM Overview

On-chip SRAM modules connect to the instruction and data buses, as shown in Figure 8-1 and Figure 8-2 internal bus, and they provide pipelined, single-cycle access to devices memory-mapped to them. Memory can be independently mapped to any properly aligned location in the 4-Gbyte address space and configured to respond to either instruction or data accesses.

Time-critical functions can be mapped into instruction memory. The system stack or other heavily referenced data can be mapped into data memory.

The following summarizes features of the CF4e SRAM implementation:

- Single-cycle throughput; when the pipeline is full, one access can occur per clock cycle.
- Physical location on the processor's high-speed local buses with a user-programmed connection to the internal instruction or data bus

- Memory location programmable on any aligned boundary; typically boundaries are 0-modulo-size aligned.
- Byte, word, longword, and line-sized access capabilities
- The RAM base address registers (RAMBAR0 and RAMBAR1) define the logical base address, attributes, and access types for the two SRAM modules.

## 8.5.1 SRAM Operation

An SRAM module provides a general-purpose memory block that the ColdFire processor can access with single-cycle throughput.

The memory block's location can be specified to any word-aligned address in the 4-Gbyte address space by  $\text{RAMBAR}_n[\text{BA}]$ , described in Section 8.5.2.1, "SRAM Base Address Registers (RAMBAR0/RAMBAR1)." Such memory is ideal for storing critical code or data structures or for use as the system stack. When mapped as an instruction memory, the SRAM can service instruction fetches generated by the processor core. When mapped as a data memory, the SRAM can service operand accesses from the processor and memory-referencing debug module commands.

The Version 4 ColdFire processor core implements a Harvard memory architecture. Each SRAM module may be logically connected to either the processor's internal instruction or data bus. This logical connection is controlled by a configuration bit in the RAM base address registers (RAMBAR0 and RAMBAR1).

If an instruction fetch is mapped into the region defined by the SRAM, the SRAM sources the data to the processor and any data fetched from the ROM or cache is discarded. If it misses in the SRAM and hits in the ROM, the ROM data is used and the data fetched from the ROM or cache is discarded. If it misses in the SRAM and hits in the ROM, the ROM data is used and the data fetched from the cache is discarded."

Likewise, if a data access is mapped into the region defined by the SRAM, the SRAM services the access and the cache is not affected. Accesses from SRAM and ROM modules are never cached, and debug-initiated references are treated as data accesses.

Note also that on-chip DMAs cannot access SRAMs. The on-chip system configuration allows concurrent core and DMA execution, where the core can reference code or data from the internal SRAMs or caches while performing a DMA transfer.

## 8.5.2 SRAM Programming Model

The SRAM programming model consists of RAMBAR0 and RAMBAR1.

### 8.5.2.1 SRAM Base Address Registers (RAMBAR0/RAMBAR1)

The SRAM modules are configured through the RAMBARs, shown in Figure 8-9.

- Each RAMBAR holds the base address of the SRAM.

- The MOVEC instruction provides write-only access to this register from the processor.
- Each RAMBAR can be read or written from the debug module in a similar manner.
- All undefined RAMBAR bits are reserved. These bits are ignored during writes to the RAMBAR and return zeros when read from the debug module.
- The valid bits, RAMBAR $n$ [V], are cleared at reset, disabling the SRAM modules. All other bits are unaffected.

**Figure 8-9. SRAM Base Address Registers (RAMBAR<sub>n</sub>)**

### Table 8-21. RAMBAR<sub>n</sub> Field Description

 **MOTOROLA**

The KRAM controller uses the base address bits it needs and ignores lower-order bits to support the configured KRAM size, as shown in Table 8-22.

**Table 8-22. KRAM Size Configuration**

KRAM Size Input Vector <sup>1</sup>	KRAM Size	Active Base Address Bits	Ignored Base Address Bits
0000	0 Bytes	None	31:9
0001	512 Bytes	31:9	None
0010	1 Kbytes	31:10	9
0011	2 Kbytes	31:11	10:9
0100	4 Kbytes	31:12	11:9
0101	8 Kbytes	31:13	12:9
0110	16 Kbytes	31:14	13:9
0111	32 Kbytes	31:15	14:9
1000	64 Kbytes	31:16	15:9
1001–1111	Reserved		

<sup>1</sup> The KRAM size input vector is determined by the core input signals `kram0size[3:0]` and `kram1size[3:0]`. Therefore, the KRAMs sizes are independently selected.

The mapping of a given access into the RAM uses the following algorithm to determine if the access hits in the memory:

```

if (RAMBAR[0] = 1)
if (((access = instructionFetch) & (RAMBAR[7] = 1)) |
    ((access = dataReference) & (RAMBAR[7] = 0)))
    if (requested address[31:n] = RAMBAR[31:n]
        if (ASn of the requested type = 0)
            Access is mapped to the RAM module
            if (access = read)
                Read the RAM and return the data
            if (access = write)
                if (RAMBAR[8] = 0)
                    Write the data into the RAM
                else Signal a write-protect access error

```

ASn refers to the five address space mask bits: C/I, SC, SD, UC, and UD.

### 8.5.3 SRAM Initialization

After a hardware reset, the contents of each SRAM module are undefined. The valid bits, `RAMBARn[V]`, are cleared, disabling the SRAM modules. If the SRAM requires initialization with instructions or data, the following steps should be performed:

1. Load `RAMBARn` with bit 7 = 0, mapping the SRAM module to the desired location. Clearing `RAMBARn[7]` logically connects the SRAM module to the processor's data bus.

2. Read the source data and write it to the SRAM. Various instructions support this function, including memory-to-memory move instructions and the move multiple instruction (MOVEM). MOVEM is optimized to generate line-sized burst fetches on line-aligned addresses, so it generally provides maximum performance.
3. After the data is loaded into the SRAM, it may be appropriate to revise the RAMBAR attribute bits, including the write-protect and address space mask fields. If the SRAM contains instructions, RAMBAR[D/I] must be set to logically connect the memory to the processor's internal instruction bus.

Remember that the SRAM cannot be accessed by on-chip DMAs. The on-chip system configuration allows concurrent core and DMA execution where the core can execute code out of internal SRAM or cache during DMA access, where the core can access instructions and operands from the internal RAM, ROM, or cache while the DMA is operational.

The ColdFire processor or an external emulator using the debug module can perform these initialization functions.

### 8.5.4 SRAM Initialization Code

The code segment below initializes the SRAM using RAMBAR0. The code sets the base address of the SRAM at 0x2000\_0000 and then initializes the 2-Kbyte block to zeros.

```
RAMBASE      EQU      0x20000000      ;set this variable to 0x20000000
RAMVALID     EQU      0x00000035
move.l       #RAMBASE+RAMVALID,D0    ;load RAMBASE + valid bit into D0
movec.l      D0, RAMBAR0              ;load RAMBAR0 and enable SRAM
```

The following loop initializes the entire SRAM to zero:

```
lea.l        RAMBASE,A0              ;load pointer to SRAM
move.l       #512,D0                 ;load loop counter into D0

SRAM_INIT_LOOP:
clr.l        (A0)+                   ;clear 4 bytes of SRAM
subq.l       #1,D0                   ;decrement loop counter
bne.b        SRAM_INIT_LOOP          ;exit if done; else continue looping
```

The following function copies the number of bytesToMove from the source (\*src) to the processor's local RAM at an offset relative to the SRAM base address defined by destinationOffset. The bytesToMove must be a multiple of 16. For best performance, source and destination SRAM addresses should be line-aligned (0-modulo-16).

```
; copyToCpuRam (*src, destinationOffset, bytesToMove)
```

```
RAMBASE      EQU      0x20000000      ;SRAM base address
RAMVALID     EQU      0x00000035      ;RAMBAR valid + mask bits

lea.l        -12(a7),a7              ;allocate temporary space
movem.l      #0x1c,(a7)              ;store D2/D3/D4 registers
```

```

; stack arguments and locations
; +0    saved d2
; +4    saved d3
; +8    saved d4
; +12   returnPc
; +16   pointer to source operand
; +20   destinationOffset
; +24   bytesToMove

    move.l    RAMBASE+RAMVALID,a0    ;define RAMBAR0 contents
    movec.l   a0,rambar0             ;load it

    move.l    16(a7),a0              ;load argument defining *src

    lea.l     RAMBASE,a1             ;memory pointer to RAM base
    add.l     20(a7),a1              ;include destination nOffset

    move.l    24(a7),d4              ;load byte count
    asr.l     #4,d4                  ;divide by 16 to convert to loop count

loop:  .align    4                    ;force loop on 0-mod-4 address
    movem.l   (a0),#0xf              ;read 16 bytes from source
    movem.l   #0xf,(a1)              ;store into RAM destination
    lea.l     16(a0),a0              ;increment source pointer
    lea.l     16(a1),a1              ;increment destination pointer
    subq.l    #1,d4                  ;decrement loop counter
    bne.b     loop                  ;if done, then exit, else continue

    movem.l   (a7),#0x1c             ;restore d2/d3/d4 registers
    lea.l     12(a7),a7              ;deallocate temporary space
    rts

```

## 8.5.5 Programming RAMBARs for Power Management

Because processor memory references may be simultaneously sent to an SRAM module and cache, power can be minimized by configuring RAMBAR address space masks as precisely as possible. For example, if an SRAM is mapped to the internal instruction bus and contains supervisor instruction data, setting the ASn mask bits associated with all operand references and user-mode instruction fetches can decrease power dissipation. Similarly, if the SRAM contains only supervisor data, setting the ASn bits associated with instruction fetches and user-mode data accesses minimizes power.

Table 8-23 shows typical RAMBAR configurations.

**Table 8-23. Examples of Typical RAMBAR Settings**

RAMBAR[7–0]	Data Contained in SRAM
0x21	Both code and data
0x2B	Code only
0x35	Data only
0x35	Supervisor and user data
0x37	Supervisor-only data
0x3C	User-only data
0xAB	Supervisor and user code

**Table 8-23. Examples of Typical RAMBAR Settings (Continued)**

RAMBAR[7-0]	Data Contained in SRAM
0xAF	Supervisor-only code
0xBB	User-only code

## 8.6 ROM Overview

On-chip ROM modules connect to the instruction and data buses, as shown in Figure 8-1 and Figure 8-2 internal bus, and they provide pipelined, single-cycle access to devices memory-mapped to them. Memory can be independently mapped to any properly aligned location in the 4-Gbyte address space and configured to respond to either instruction or data accesses.

Time-critical functions can be mapped into instruction memory. The system stack or other heavily referenced data can be mapped into data memory.

The following summarizes features of the CF4e ROM implementation:

- Single-cycle throughput. When the pipeline is full, one access can occur per clock cycle.
- Physical location on the processor's high-speed local buses with a user-programmed connection to the internal instruction or data bus
- Memory location programmable on any aligned boundary; typically boundaries are 0-modulo-size aligned.
- Byte, word, longword, and line-sized access capabilities
- The ROM base address registers (ROMBAR0 and ROMBAR1) define the logical base address, attributes, and access types for the two ROM modules.

### 8.6.1 ROM Operation

A ROM module provides a general-purpose block of read-only memory that the ColdFire processor can access with single-cycle throughput.

The memory block's location can be specified to any word-aligned address in the 4-Gbyte address space by ROMBAR $_n$ [BA], described in Section 8.5.2.1, "SRAM Base Address Registers (RAMBAR0/RAMBAR1)." Such memory is ideal for storing critical code or data structures or for use as the system stack. When mapped as an instruction memory, the ROM can service instruction fetches generated by the processor core. When mapped as a data memory, the ROM can service operand accesses from the processor and memory-referencing debug module commands.

The Version 4 ColdFire processor core implements a Harvard memory architecture. Each ROM module may be logically connected to either the processor's internal instruction or data bus. This logical connection is controlled by a configuration bit in the ROM base



address registers (ROMBAR0 and ROMBAR1).

Note also that on-chip DMAs cannot access ROMs. The on-chip system configuration allows concurrent core and DMA execution, where the core can reference code or data from the internal SRAMs or caches while performing a DMA transfer.

## 8.6.2 ROM Programming Model

The ROM programming model consists of ROMBAR0 and ROMBAR1.

### 8.6.2.1 ROM Base Address Registers (ROMBAR0/ROMBAR1)

The ROM modules are configured through the ROMBARs, shown in Figure 8-10.

- Each ROMBAR holds the base address of the ROM.
- The MOVEC instruction provides write-only access to this register from the processor.
- Each ROMBAR can be read or written from the debug module in a similar manner.
- All undefined ROMBAR bits are reserved. These bits are ignored during writes to the ROMBAR and return zeros when read from the debug module.
- The initial state of the valid bit (V) is controlled by the value of a core input pin. If the `kromvldrst` input is asserted at reset, the contents of the ROMBAR is forced to `0x0000_0121`. This defines a valid ROM memory, based at address 0, write-protected with the CPU space/interrupt acknowledge accesses masked. If `kromvldrst` is negated, `ROMBARn[V]` is cleared by reset, disabling the ROM module.

	31					9	8		6	5	4	3	2	1	0	
Field	BA							WP	D/I	—	C/I	SC	SD	UC	UD	V
Reset	—								00		—	—	—	—	—	x <sup>1</sup>
R/W	W for CPU; R/W for debug															
Address	CPU space + 0xC00 (ROMBAR0), 0xC01 (ROMBAR1)															

<sup>1</sup> If `kromvldrst` is asserted at reset, the contents of the ROMBAR is forced to `0x0000_0121`. This defines a valid ROM memory, based at address 0, write-protected with the CPU space/interrupt acknowledge accesses masked. If `kromvldrst` is negated, the valid bit is cleared by reset, disabling the ROM module.

**Figure 8-10. ROM Base Address Registers (ROMBAR0/ROMBAR1)**

ROMBAR<sub>n</sub> fields are described in detail in Table 8-24.

Table 8-24. ROMBAR Field Descriptions

Bits	Name	Description
31–9	BA	Base address. Defines the ROM module base address. ROM alignment is implementation specific. See Table 8-25.
8	WP	Write protect. Controls read/write properties of the ROM. 0 Allows read and write accesses to the SROM module 1 Allows only read accesses to the SROM module. Any attempted write reference generates an access error exception to the ColdFire processor core.
7	D/I	Data/instruction bus. Indicates whether ROM is connected to the internal data or instruction bus. 0 Data bus 1 Instruction bus
6	—	Reserved, should be cleared.
5–1	C/I, SC, SD, UC, UD	Address space masks (AS <sub>n</sub> ). Allows specific address spaces to be enabled or disabled, placing internal modules in a specific address space. If an address space is disabled, an access to the register in that address space becomes an external bus access, and the module resource is not accessed. These bits are useful for power management as described in Section 8.6.4, “Programming ROMBARs for Power Management.” In particular, C/I is typically set. The address space mask bits are follows: C/I = CPU space/interrupt acknowledge cycle mask. <b>Note:</b> C/I must be set if BA = 0. SC = Supervisor code address space mask SD = Supervisor data address space mask UC = User code address space mask UD = User data address space mask For each AS <sub>n</sub> bit: 0 An access to the ROM module can occur for this address space 1 Disable this address space from the ROM module. References to this address space cannot access the ROM module and are processed like other non-ROM references.
0	V	Valid. Indicates whether ROMBAR contents are valid. The BA value is not used and the ROM module is not accessible until V is set. 0 Contents of ROMBAR are not valid. The ROM module is disabled. 1 Contents of ROMBAR are valid. The ROM module is enabled. If kromvldrst is asserted at reset, the contents of the ROMBAR are forced to 0x0000_0121. This defines a valid ROM memory, based at address 0, write-protected with the CPU space/interrupt acknowledge accesses masked. If kromvldrst is negated, the valid bit is cleared by reset, disabling the ROM module.

The KROM controller uses the base address bits it needs and ignores lower-order bits to support the configured KROM size, as shown in Table 8-25.

Table 8-25. KROM Size Configuration

KROM Size Input Vector <sup>1</sup>	KROM Size	Active Base Address Bits	Ignored Base Address Bits
0000	0 Bytes	None	31:9
0001	512 Bytes	31:9	None
0010	1 Kbytes	31:10	9
0011	2 Kbytes	31:11	10:9
0100	4 Kbytes	31:12	11:9
0101	8 Kbytes	31:13	12:9
0110	16 Kbytes	31:14	13:9

**Table 8-25. KROM Size Configuration (Continued)**

KROM Size Input Vector <sup>1</sup>	KROM Size	Active Base Address Bits	Ignored Base Address Bits
0111	32 Kbytes	31:15	14:9
1000	64 Kbytes	31:16	15:9
1001–1111	Reserved		

<sup>1</sup> The KROM size input vector is determined by the core input signals `kram0size[3:0]` and `kram1size[3:0]`. Therefore, the KROMs sizes are independently selected.

The mapping of a given access into the ROM uses the following algorithm to determine if the access hits in the memory:

```

if (ROMBAR[0] = 1)
if (((access = instructionFetch) & (ROMBAR[7] = 1)) |
    ((access = dataReference) & (ROMBAR[7] = 0)))
    if (requested address[31:n] = ROMBAR[31:n]
        if (ASn of the requested type = 0)
            Access is mapped to the ROM module
            if (access = read)
                Read the ROM and return the data
            if (access = write)
                if (ROMBAR[8] = 0)
                    Write the data into the ROM
                else Signal a write-protect access error

```

ASn refers to the five address space mask bits: C/I, SC, SD, UC, and UD.

### 8.6.3 ROM Initialization

After a hardware reset, the contents of each ROM module are undefined. If the ROM requires initialization with instructions or data, the following steps should be performed:

1. Load ROMBAR<sub>n</sub> with bit 7 = 0, mapping the ROM module to the desired location. Clearing ROMBAR<sub>n</sub>[7] logically connects the ROM module to the processor's data bus.
2. Read the source data and write it to the ROM. Various instructions support this function, including memory-to-memory move instructions and the move multiple instruction (MOVEM). MOVEM is optimized to generate line-sized burst fetches on line-aligned addresses, so it generally provides maximum performance.
3. After the data is loaded into the ROM, it may be appropriate to revise the ROMBAR attribute bits, including the write-protect and address space mask fields. If the ROM contains instructions, ROMBAR[D/I] must be set to logically connect the memory to the processor's internal instruction bus.

Remember that the ROM cannot be accessed by on-chip DMAs. The on-chip system configuration allows concurrent core and DMA execution where the core can execute code out of internal ROM or cache during DMA access.

The ColdFire processor or an external emulator using the debug module can perform these initialization functions.

### 8.6.4 Programming ROMBARs for Power Management

Depending on the ROMBAR configuration, memory accesses can be sent to a ROM module and cache simultaneously. If an access hits both, the ROM module sources read data and the instruction cache access is discarded. Because the ROM contains only data, setting ROMBAR[SC,UC] lowers power dissipation by disabling the ROM during instruction fetches.

Table 8-26 shows typical ROMBAR settings:

**Table 8-26. Examples of Typical ROMBAR Settings**

ROMBAR[7–0]	Data Contained in ROM
0x21	Both code and data
0x2B	Code only
0x35	Data only
0x35	Supervisor and user data
0x37	Supervisor-only data
0x3C	User-only data
0xAB	Supervisor and user code
0xAF	Supervisor-only code
0xBB	User-only code

RAMBARs are configured similarly, as described in Section 8.5.5, “Programming RAMBARs for Power Management.”

## 8.7 Cache Overview

This section describes the Harvard cache implementation, including organization, configuration, and coherency. It describes cache operations and how the instruction and data caches interact with other memory structures.

The CF4e implements a special branch instruction cache for accelerating branches, enabled by a bit in the cache access control register (CACR[BEC]).

Caches improve system performance by providing single-cycle access to the instruction and data pipelines. This decouples processor performance from system memory performance, increasing bus availability for on-chip DMA or external devices. Figure 8-1 and Figure 8-2 show the integration of the instruction and data caches in the local memory model.

Both the instruction and data cache implement line-fill buffers to optimize line-sized burst accesses. The data cache supports operation of copyback, write-through, or cache-inhibited modes. A four-entry, 32-bit buffer supports cache line-push operations, and can be configured to defer write buffering in write-through or cache-inhibited modes. The cache lock feature can be used to guarantee deterministic response for critical code or data areas.

A nonblocking cache services read hits or write hits from the processor while a fill (caused by a cache allocation) is in progress.

The data and instruction caches are physically addressed. A cache hit occurs when an address matches a cache entry. For a read, the cache supplies data to the processor. For a write, which is permitted only to the data cache, the processor updates the cache. If an access does not match a cache entry (misses the cache) or if a write access must be written through to memory, the core system bus controller performs the necessary system bus transactions.

Cache modules do not implement bus snooping; cache coherency with other possible bus masters must be maintained in software.

### 8.7.1 Optimizing Cache Recommendation

The following is recommended for optimal cache performance.

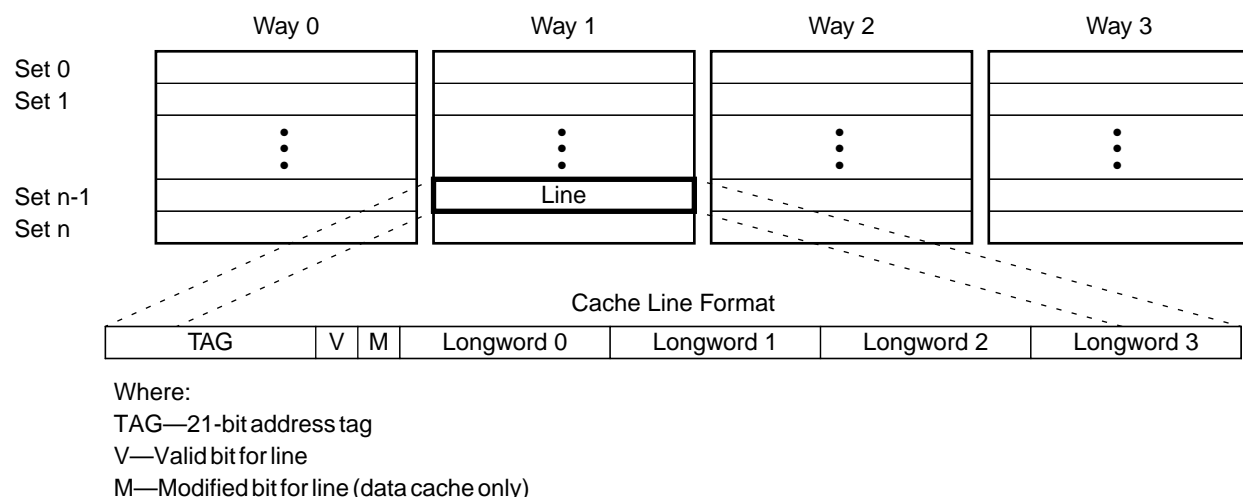
- Cache data and instruction space to improve system performance.
- Do not cache the following:
  - SIM space
  - Memory-mapped I/O space
  - DMA space

### 8.7.2 Cache Organization

A four-way set associative cache is organized as four ways (levels). Each line contains 16 bytes (4 longwords). Entire cache lines are loaded from memory by burst-mode accesses that cache 4 longwords of data or instructions. All 4 longwords must be loaded for the cache line to be valid.

Figure 8-11 shows the data cache organization, which differs from the instruction cache by the inclusion of the modified bit, which indicates that the data cache block has been written to without changing the corresponding location in system memory.

## Cache Overview



**Figure 8-11. Data Cache Organization and Line Format**

A set is a group of four lines (one from each level, or way), corresponding to the same index into the cache array.

### 8.7.2.1 Cache Line States: Invalid, Valid-Unmodified, and Valid-Modified

As shown in Table 8-27, a data cache line can be invalid, valid-unmodified (often called exclusive), or valid-modified. An instruction cache line can be valid or invalid.

**Table 8-27. Valid and Modified Bit Settings**

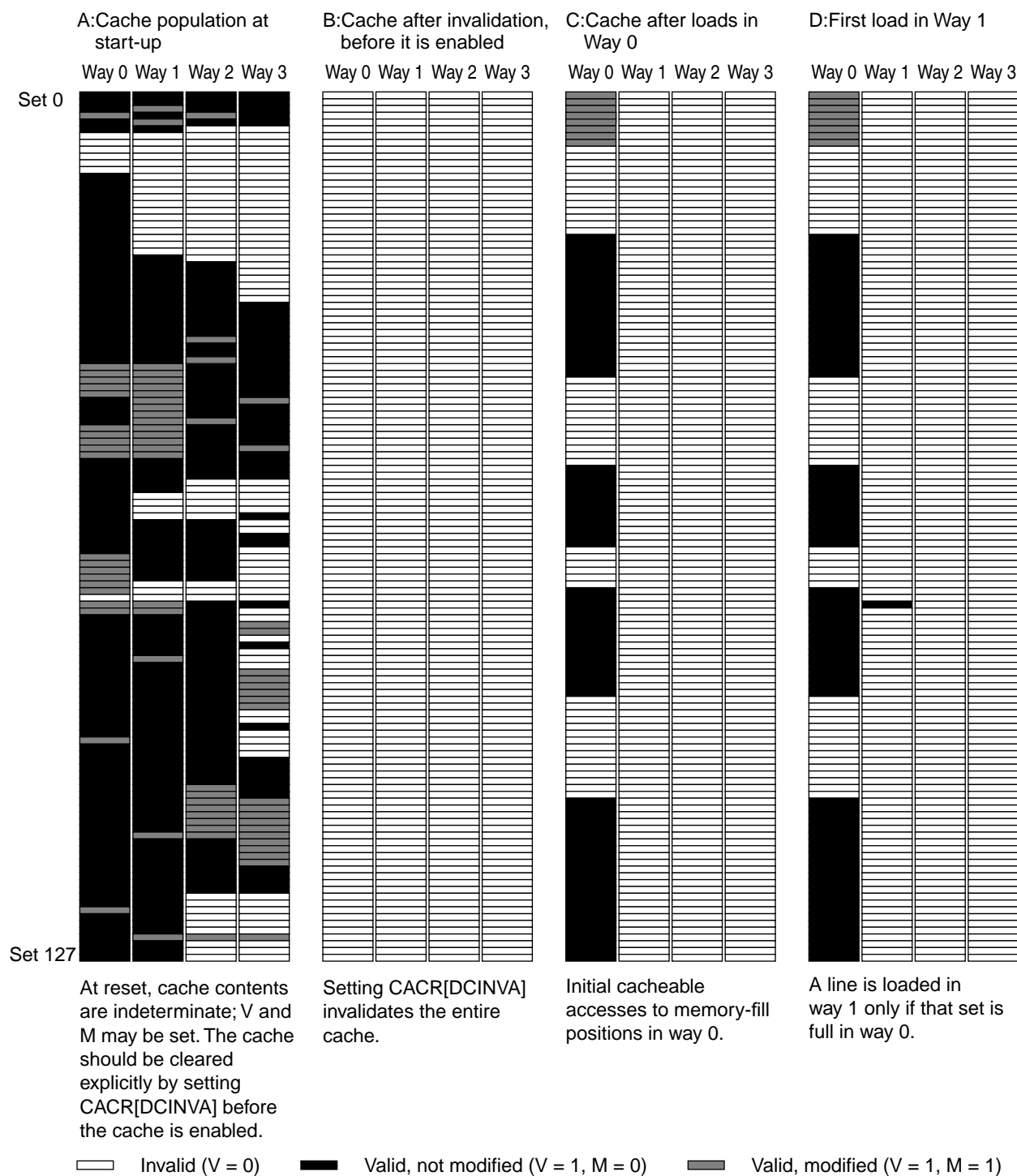
V	M	Description
0	x	Invalid. Invalid lines are ignored during lookups.
1	0	Valid, unmodified. Cache line has valid data that matches system memory.
1	1	Valid, modified. Cache line contains most recent data, data at system memory location is stale.

A valid line can be explicitly invalidated by executing a CPUSHL instruction.

### 8.7.2.2 Cache at Start-Up

As Figure 8-12 (A) shows, after power-up, cache contents are undefined; V and M may be set on some lines even though the cache may not contain the appropriate data for start-up. Because reset and power-up do not invalidate cache lines automatically, the cache must be cleared explicitly by setting CACR[DCINVA, ICINVA] before the cache is enabled (B).

After the entire cache is automatically flushed, cacheable entries are loaded first in way 0. If way 0 is occupied, the cacheable entry is loaded into the same set in way 1, as shown in Figure 8-12 (D). This process is described in detail in Section 8.7.3, “Cache Operation.”



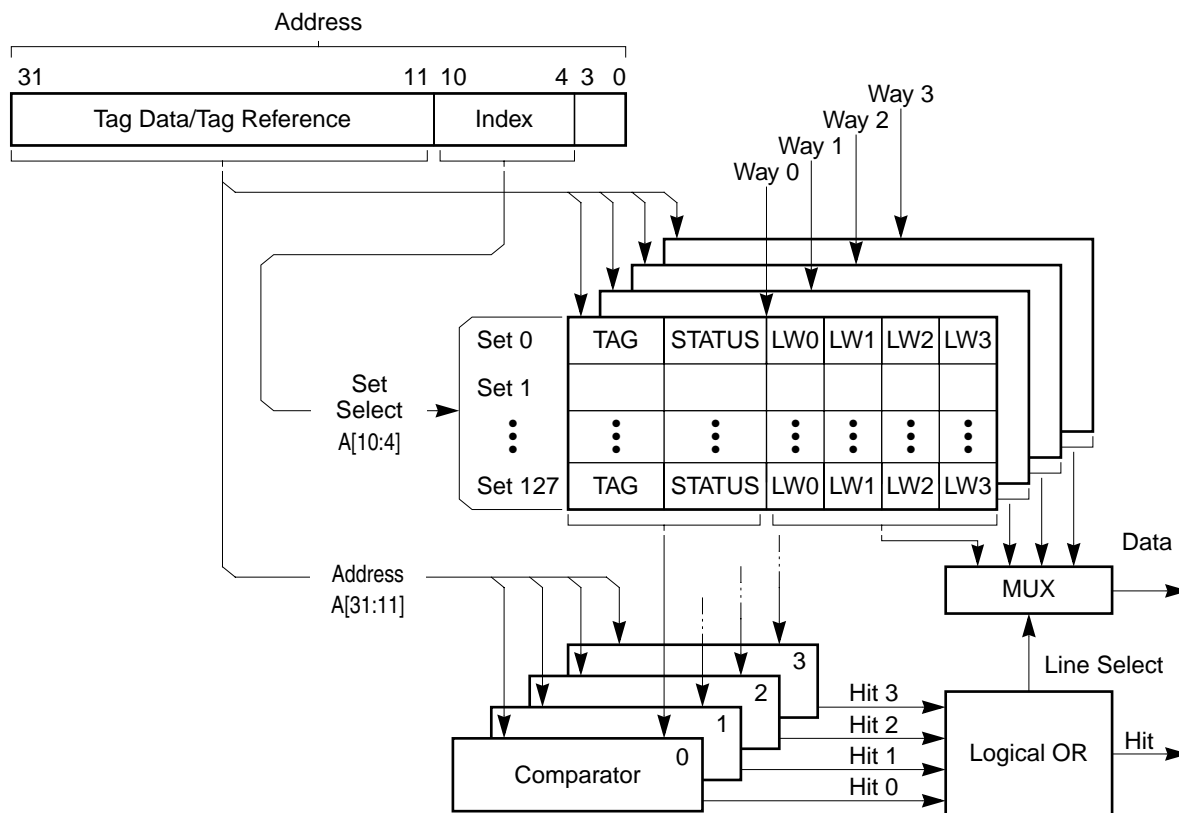
**Figure 8-12. Data Cache—A: at Reset, B: after Invalidation, C and D: Loading Pattern**

### 8.7.3 Cache Operation

Figure 8-13 shows the general flow of a caching operation using the 8-Kbyte data cache as an example. The discussion in this chapter assumes a data cache. Instruction cache

## Cache Overview

operations are similar except that there is no support for writing to the cache directly; therefore such notions as modified cache lines and write allocation do not apply.



**Figure 8-13. Data Caching Operation**

The following steps determine if a data cache line is allocated for a given address:

1. The cache set index,  $A[10:4]$ , selects one cache set.
2.  $A[31:11]$  and the cache set index are used as a tag reference or are used to update the cache line tag field.
3. Note that  $A[31:11]$  can specify  $2^{21}$  possible addresses that can be mapped to one of four ways.
4. The four tags from the selected cache set are compared with the tag reference. A cache hit occurs if a tag matches the tag reference and the V bit is set, indicating that the cache line contains valid data. If a cacheable write access hits in a valid cache line, the write can occur to the cache line without having to load it from memory.

If the memory space is copyback, the updated cache line is marked modified ( $M = 1$ ), because the new data has made the data in memory out of date. If the memory location is write-through, the write is passed on to system memory and the M bit is never used. Note that the tag does not have TT or TM bits.

To allocate a cache entry, the cache set index selects one of the cache's 128 sets. The cache control logic looks for an invalid cache line to use for the new entry. If none is available,



the cache controller uses a pseudo-round-robin replacement algorithm to choose the line to be deallocated and replaced. First the cache controller looks for an invalid line, with way 0 the highest priority. If all lines have valid data, a 2-bit replacement counter is used to choose the way. After a line is allocated, the pointer increments to point to the next way.

Cache lines from ways 0 and 1 can be protected from deallocation by enabling half-cache locking. If  $CACR[DHLCK, IHLCK] = 1$ , the replacement pointer is restricted to way 2 or 3.

As part of deallocation, a valid, unmodified cache line is invalidated. It is consistent with system memory, so memory does not need to be updated. To deallocate a modified cache line, data is placed in a push buffer (for an external cache line push) before being invalidated. After invalidation, the new entry can replace it. The old cache line may be written after the new line is read.

When a cache line is selected to host a new cache entry, the following three things happen:

1. The new address tag bits  $A[31:11]$  are written to the tag.
2. The cache line is updated with the new memory data.
3. The cache line status changes to a valid state ( $V = 1$ ).

Read cycles that miss in the cache allocate normally as previously described.

Write cycles that miss in the cache do not allocate on a cacheable write-through region, but do allocate for addresses in a cacheable copyback region.

A copyback byte, word, longword, or line write miss causes the following:

1. The cache initiates a line fill or flush.
2. Space is allocated for a new line.
3.  $V$  and  $M$  are both set to indicate valid and modified.
4. Data is written in the allocated space. No write to memory occurs.

Note the following:

- Read hits cannot change the status bits and no deallocation or replacement occurs; the data or instructions are read from the cache.
- If the cache hits on a write access, data is written to the appropriate portion of the accessed cache line. Write hits in cacheable, write-through regions generate an external write cycle and the cache line is marked valid, but is never marked modified. Write hits in cacheable copyback regions do not perform an external write cycle; the cache line is marked valid and modified ( $V = 1$  and  $M = 1$ ).
- Misaligned accesses are broken into at least two cache accesses.
- Validity is provided only on a line basis. Unless a whole line is loaded on a cache miss, the cache controller does not validate data in the cache line.

Write accesses designated as cache-inhibited by the  $CACR$  or  $ACR$  bypass the cache and perform a corresponding external write.

Normally, cache-inhibited reads bypass the cache and are performed on the external bus. The exception to this normal operation occurs when all of the following conditions are true during a cache-inhibited read:

- The cache-inhibited fill buffer bit, CACR[DNFB], is set.
- The access is an instruction read.
- The access is normal (that is, transfer type (TT) equals 0).

In this case, an entire line is fetched and stored in the fill buffer, where it remains valid. The cache can service additional read accesses from this buffer until either another fill or a cache-invalidate-all operation occurs.

Valid cache entries that match during cache-inhibited address accesses are neither pushed nor invalidated. Such a scenario suggests that the associated cache mode for this address space was changed. To avoid this, it is generally recommended to use the CPUSHL instruction to push or invalidate the cache entry or set CACR[DCINVA] to invalidate the data cache before switching cache modes.

### 8.7.4 Caching Modes

For every memory reference generated by the processor or debug module, a set of effective attributes is determined based on the address and the ACRs. Caching modes determine how the cache handles an access. A data access can be cacheable in either write-through or copyback mode; it can be cache-inhibited in precise or imprecise modes. For normal accesses, the ACR<sub>n</sub>[CM] bit corresponding to the address of the access specifies the caching modes. If an address does not match an ACR, the default caching mode is defined by CACR[DDCM,IDCM]. The specific algorithm for the data cache prioritization (which uses ACR0 and ACR1) is as follows:

```
if (address == ACR0-address including mask)
    effective attributes = ACR0 attributes
else if (address == ACR1-address including mask)
    effective attributes = ACR1 attributes
else effective attributes = CACR default attributes
```

Addresses matching an ACR can also be write-protected using ACR[W]. Addresses that do not match either ACR can be write-protected using CACR[DW].

Reset disables the cache and clears all CACR bits. As shown in Figure 8-12, reset does not automatically invalidate cache entries; they must be invalidated through software.

The ACRs allow the defaults selected in the CACR to be overridden. In addition, some instructions (for example, CPUSHL) and processor core operations perform accesses that have an implicit caching mode associated with them. The following sections discuss the different caching accesses and their associated cache modes.

### 8.7.4.1 Cacheable Accesses

If  $ACR_n[CM]$  or the default field of the CACR indicates write-through or copyback, the access is cacheable. A read access to a write-through or copyback region is read from the cache if matching data is found. Otherwise, the data is read from memory and the cache is updated. When a line is being read from memory for either a write-through or copyback read miss, the longword within the line that contains the core-requested data is loaded first and the requested data is given immediately to the processor, without waiting for the three remaining longwords to reach the cache.

The following sections describe write-through and copyback modes in detail. Note that some of this information applies to data caches only.

### 8.7.4.2 Write-Through Mode (Data Cache Only)

Write accesses to regions specified as write-through are always passed on to the external bus, although the cycle can be buffered, depending on the state of  $CACR[DESB]$ . Writes in write-through mode are handled with a no-write-allocate policy—that is, writes that miss in the cache are written to the external bus but do not cause the corresponding line in memory to be loaded into the cache. Write accesses that hit always write through to memory and update matching cache lines. The cache supplies data to data-read accesses that hit in the cache; read misses cause a new cache line to be loaded into the cache.

### 8.7.4.3 Copyback Mode (Data Cache Only)

Copyback regions are typically used for local data structures or stacks to minimize external bus use and reduce write-access latency. Write accesses to regions specified as copyback that hit in the cache update the cache line and set the corresponding M bit without an external bus access.

Be sure to flush the cache using the CPUSHL instruction before invalidating the cache in copyback mode. Modified cache data is written to memory only if the line is replaced because of a miss or a CPUSHL instruction pushes the line. If a byte, word, longword, or line write access misses in the cache, the required cache line is read from memory, thereby updating the cache. When a miss selects a modified cache line for replacement, the modified cache data moves to the push buffer. The replacement line is read into the cache and the push buffer contents are then written to memory.

## 8.7.5 Cache-Inhibited Accesses

Memory regions can be designated as cache-inhibited, which is useful for memory containing targets such as I/O devices and shared data structures in multiprocessing systems. It is also important to not cache memory-mapped registers. If the corresponding  $ACR_n[CM]$  or  $CACR[DDCM]$  indicates cache-inhibited, precise or imprecise, the access is cache-inhibited. The caching operation is identical for both cache-inhibited modes, which differ only regarding recovery from an external bus error.

In determining whether a memory location is cacheable or cache-inhibited, the CPU checks memory-control registers in the following order:

1. RAMBARs
2. ROMBARs
3. ACR0 and ACR2
4. ACR1 and ACR3
5. If an access does not hit in the RAMBARs, ROMBARs, or the ACRs, the default is provided for all accesses in CACR.

Cache-inhibited write accesses bypass the cache and a corresponding external write is performed. Cache-inhibited reads bypass the cache and are performed on the external bus, except when all of the following conditions are true:

- The cache-inhibited fill-buffer bit, CACR[DNFB], is set.
- The access is an instruction read.
- The access is normal (that is, TT = 0).

In this case, a fetched line is stored in the fill buffer and remains valid there; the cache can service additional read accesses from this buffer until another fill occurs or a cache-invalidate-all operation occurs.

If  $ACR_n[CM]$  indicates cache-inhibited mode, precise or imprecise, the controller bypasses the cache and performs an external transfer. If a line in the cache matches the address and the mode is cache-inhibited, the cache does not automatically push the line if it is modified, nor does it invalidate the line if it is valid. Before switching cache mode, execute a CPUSHL instruction or set CACR[DCINVA,ICINVA] to invalidate the entire cache.

If  $ACR_n[CM]$  indicates precise mode, the sequence of read and write accesses to the region is guaranteed to match the instruction sequence. In imprecise mode, the processor core allows read accesses that hit in the cache to occur before completion of a pending write from a previous instruction. Writes are not deferred past data-read accesses that miss the cache (that is, that must be read from the bus).

Precise operation forces data-read accesses for an instruction to occur only once by preventing the instruction from being interrupted after data is fetched. Otherwise, if the processor is not in precise mode, an exception aborts the instruction and the data may be accessed again when the instruction is restarted. These guarantees apply only when  $ACR_n[CM]$  indicates precise mode and aligned accesses.

CPU space-register accesses, such as MOVEC, are treated as cache-inhibited and precise.

## 8.7.6 Cache Protocol

The following sections describe the cache protocol for processor accesses and assumes that the data is cacheable (that is, write-through or copyback). Note that the discussion of write operations applies to the data cache only.

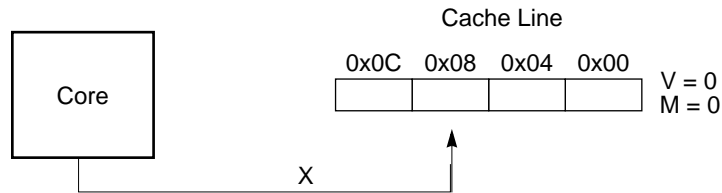
### 8.7.6.1 Read Miss

A processor read that misses in the cache requests the cache controller to generate a bus transaction. This bus transaction reads the needed line from memory and supplies the required data to the processor core. The line is placed in the cache in the valid state.

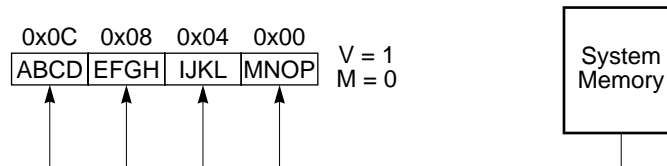
### 8.7.6.2 Write Miss (Data Cache Only)

The cache controller handles processor writes that miss in the data cache differently for write-through and copyback regions. Write misses to copyback regions cause the cache line to be read from system memory, as shown in Figure 8-14.

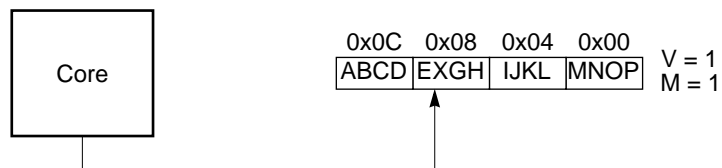
1. Writing character X to 0x0A generates a write miss. Data cannot be written to an invalid line.



2. The cache line (characters A–P) is updated from system memory, and the line is marked valid.



3. After the cache line is filled, the write that initiated the write miss (the character X) completes.



**Figure 8-14. Write-Miss in Copyback Mode**

The new cache line is then updated with write data and the M bit is set for the line, leaving it in modified state. Write misses to write-through regions write directly to memory without loading the corresponding cache line into the cache.

### 8.7.6.3 Read Hit

On a read hit, the cache provides the data to the processor core and the cache line state remains unchanged. If the cache mode changes for a specific region of address space, lines in the cache corresponding to that region that contain modified data are not pushed out to memory when a read hit occurs within that line. First execute a CPUSHL instruction or set CACR[DCINVA, ICINVA] before switching the cache mode.

#### 8.7.6.4 Write Hit (Data Cache Only)

The cache controller handles processor writes that hit in the data cache differently for write-through and copyback regions. For write hits to a write-through region, portions of cache lines corresponding to the size of the access are updated with the data. The data is also written to external memory. The cache line state is unchanged. For copyback accesses, the cache controller updates the cache line and sets the M bit for the line. An external write is not performed and the cache line state changes to (or remains in) the modified state.

#### 8.7.7 Cache Coherency (Data Cache Only)

The CF4e provides limited cache coherency support in multiple-master environments. Both write-through and copyback memory update techniques are supported to maintain coherency between the cache and memory.

The cache does not support snooping (that is, cache coherency is not supported while external or DMA masters are using the bus). Therefore, any on-chip DMAs access local memory and do not maintain coherency with the data cache. Therefore, cache coherency is left to the user.

#### 8.7.8 Memory Accesses for Cache Maintenance

The cache controller performs all maintenance activities that supply data from the cache to the core, including requests to the SIM for reading new cache lines and writing modified lines to memory. The following sections describe memory accesses resulting from cache fill and push operations.

##### 8.7.8.1 Cache Filling

When a new cache line is required, the core system bus controller performs a burst-read transfer on the system bus.

SIM line accesses implicitly request burst-mode operations from memory.

The first cycle of a cache-line read loads the longword entry corresponding to the requested address. Subsequent transfers load the remaining longword entries.

A burst operation is aborted by a write-protection fault, which is the only possible access error. Exception processing proceeds immediately. Note that unlike Version 2 and Version 3 access errors, in this version, the program counter stored on the exception stack frame points to the faulting instruction. See Chapter 7, “Exception Processing.”

##### 8.7.8.2 Cache Pushes

Cache pushes occur for line replacement and as required for the execution of the CPUSHL instruction. To reduce the requested data's latency in the new line, the modified line being replaced is temporarily placed in the push buffer while the new line is fetched from

memory. After the bus transfer for the new line completes, the modified cache line is written back to memory and the push buffer is invalidated.

### 8.7.8.2.1 Push and Store Buffers

The 16-byte push buffer reduces latency for requested new data on a cache miss by holding a displaced modified data cache line while the new data is read from memory.

If a cache miss displaces a modified line, a miss read reference is immediately generated. While waiting for the response, the current contents of the cache location load into the push buffer. When the burst-read bus transaction completes, the cache controller can generate the appropriate line-write bus transaction to write the push buffer contents into memory. Note that this is not programmable and is always on if the cache is used.

In imprecise mode, the FIFO store buffer can defer pending writes to maximize performance. The store buffer can support as many as four entries (16 bytes maximum) for this purpose.

Data writes destined for the store buffer cannot stall the core. The store buffer effectively provides a measure of decoupling between the pipeline's ability to generate writes (one per cycle maximum) and the external bus's ability to retire those writes. In imprecise mode, writes stall the core only if the store buffer is full and a write operation is on the internal bus. The internal write cycle is held, stalling the data execution pipeline.

If the store buffer is not used (that is, store buffer disabled or cache-inhibited precise mode), external bus cycles are generated directly for each pipeline write operation. The instruction is held in the pipeline until external bus transfer termination is received.

The data store buffer enable bit, CACR[DESB], controls the enabling of the data store buffer. This bit can be set and cleared by the MOVEC instruction. DESB is zero at reset and all writes are performed in order (precise mode). ACR<sub>n</sub>[CM] or CACR[DDCM] generates the mode used when DESB is set. Cacheable write-through and cache-inhibited imprecise modes use the store buffer.

The store buffer can queue data as much as 4 bytes wide per entry. Each entry matches the corresponding bus cycle it generates; therefore, a misaligned longword write to a write-through region creates two entries if the address is to an odd-word boundary. It creates three entries if it is to an odd-byte boundary—one per bus cycle.

### 8.7.8.2.2 Push and Store Buffer Bus Operation

As soon as the push or store buffer has valid data, the internal bus controller uses the next available external bus cycle to generate the appropriate write cycles. In the event that another cache fill is required (for example, cache miss to process) during the continued instruction execution by the processor pipeline, the pipeline stalls until the push and store buffers are empty, then generates the required external bus transaction.

Exception processing and the cpushl, intouch, halt, move-to-SR, movec, nop, rte, stop, wdebug instructions synchronize the processor core and guarantee the push and store

buffers are empty before proceeding. Note that the NOP instruction should be used only to synchronize the pipeline. The preferred no-operation function is the TPF instruction.

### 8.7.9 Cache Locking

Ways 0 and 1 of the data cache can be locked by setting CACR[DHLCK]; likewise, ways 0 and 1 of the instruction cache can be locked by setting CACR[IHLCK]. If a cache is locked, cache lines in ways 0 and 1 are not subject to being deallocated by normal cache operations.

As Figure 8-15 (B and C) shows, the algorithm for updating the cache and for identifying cache lines to be deallocated is otherwise unchanged. If ways 2 and 3 are entirely invalid, cacheable accesses are first allocated in way 2. Way 3 is not used until the location in way 2 is occupied.

Ways 0 and 1 are still updated on write hits (D in Figure 8-15) and may be pushed or cleared explicitly by using specific cache push/invalidate instructions. However, new cache lines cannot be allocated in ways 0 and 1.

Figure 8-15 also shows the benefits of using the INTOUCH instruction to systematically fill ways 0 and 1.



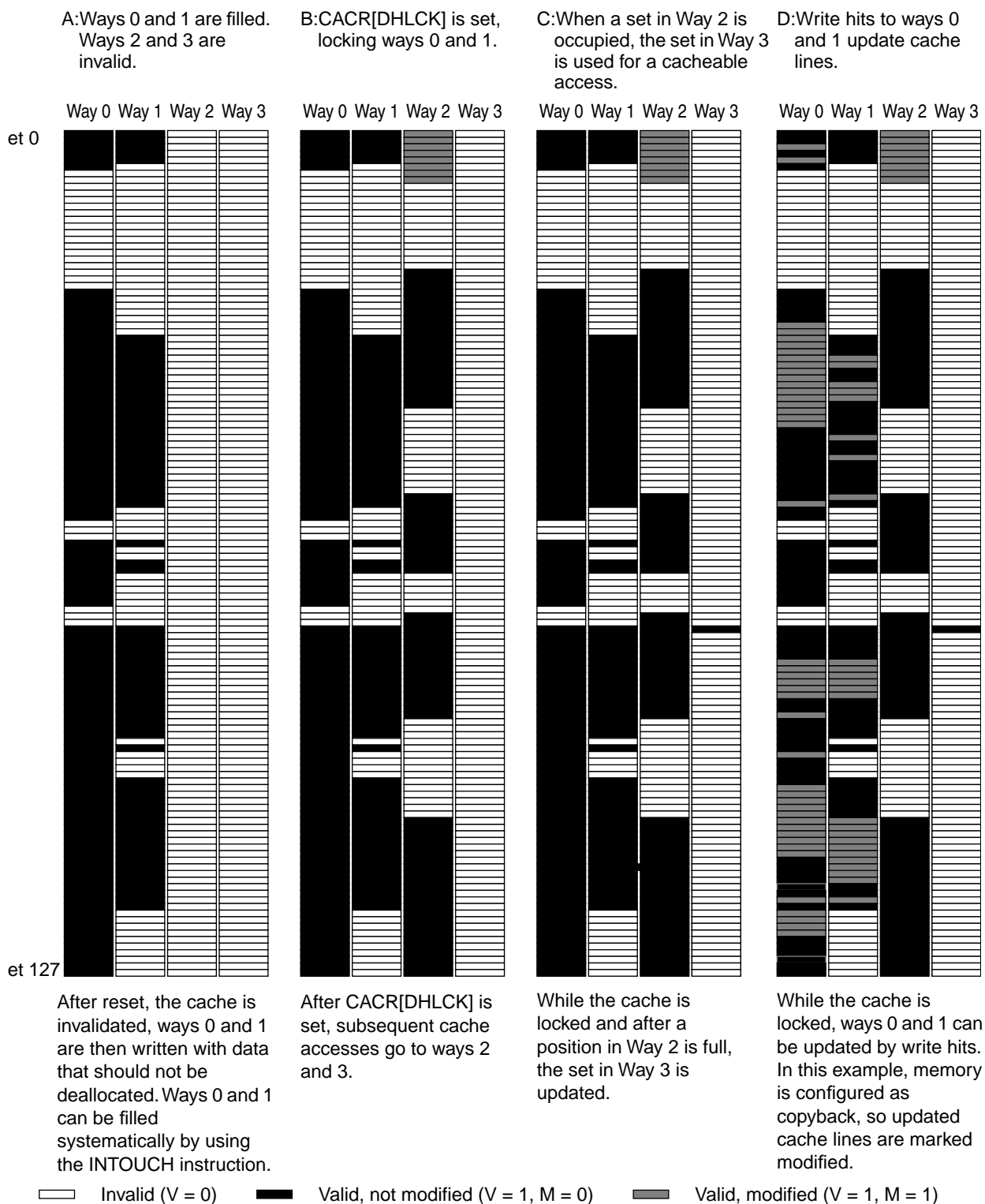


Figure 8-15. Data Cache Locking

### 8.7.10 Cache Registers

This section describes the CF4e cache registers.

### 8.7.10.1 Cache Control Register (CACR)

The CACR in Figure 8-16 contains bits for configuring the cache. It can be written by the MOVEC register instruction and can be read or written from the debug facility. A hardware reset clears CACR, which disables the cache; however, reset does not affect the tags, state information, or data in the cache.

	31	30	29	28	27	26	25	24	23	22		20	19	18	17	16
Field	DEC	DW	DESB	DDPI	DHLCK	DDCM	DCINVA	DDSP		—		BEC	BCINVA		—	
Reset	All zeros															
R/W	Write (R/W by debug module)															
	15	14	13	12	11	10	9	8	7	6	5	4	3	3		0
Field	IEC	—	DNFB	IDPI	IHLCK	IDCM	—	ICINVA	IDSP	—	EUSP	DF		—		
Reset	All zeros															
R/W	Write (R/W by debug module)															
Rc	0x002															

**Figure 8-16. Cache Control Register (CACR)**

Table 8-28 describes CACR fields. Note that some implementations may include fields not defined here; consult the part-specific documentation.

**Table 8-28. CACR Field Descriptions**

Bits	Name	Description
31	DEC	Enable data cache. 0 Cache disabled. The data cache is not operational, but data and tags are preserved. 1 Cache enabled.
30	DW	Data default write-protect. For normal operations that do not hit in the RAMBARs or ACRs, this field defines write-protection. See Section 8.7.4, “Caching Modes.” 0 Not write protected. 1 Write protected. Write operations cause an access error exception.
29	DESB	Enable data store buffer. Affects the precision of transfers. 0 Imprecise-mode, write-through or cache-inhibited writes bypass the store buffer and generate bus cycles directly. Section 8.7.8.2.1, “Push and Store Buffers,” describes the associated performance penalty. 1 The four-entry FIFO store buffer is enabled; to maximize performance, this buffer defers pending imprecise-mode, write-through or cache-inhibited writes. Precise-mode, cache-inhibited accesses always bypass the store buffer. Precise and imprecise modes are described in Section 8.7.5, “Cache-Inhibited Accesses.”
28	DDPI	Disable CPUSHL invalidation. 0 Normal operation. A CPUSHL instruction causes the selected line to be pushed if modified, then invalidated. 1 No clear operation. A CPUSHL instruction causes the selected line to be pushed if modified, then left valid.

Table 8-28. CACR Field Descriptions (Continued)

Bits	Name	Description
27	DHLCK	Half-data cache lock mode 0 Normal operation. The cache allocates the lowest invalid way. If all ways are valid, the cache allocates the way pointed at by the counter and then increments this counter. 1 Half-cache operation. The cache allocates to the lower invalid way of levels 2 and 3; if both are valid, the cache allocates to Way 2 if the high-order bit of the round-robin counter is zero; otherwise, it allocates Way 3 and increments the round-robin counter. This locks the contents of ways 0 and 1. Ways 0 and 1 are still updated on write hits and may be pushed or cleared by specific cache push/invalidate instructions.
26–25	DDCM	Default data cache mode. For normal operations that do not hit in the RAMBARs, ROMBARs, or ACRs, this field defines the effective cache mode. 00 Cacheable write-through imprecise 01 Cacheable copyback 10 Cache-inhibited precise 11 Cache-inhibited imprecise Precise and imprecise accesses are described in Section 8.7.5, “Cache-Inhibited Accesses.”
24	DCINVA	Data cache invalidate all. Writing a 1 to this bit initiates entire cache invalidation. Once invalidation is complete, this bit automatically returns to 0; it is not necessary to clear it explicitly. Note the caches are not cleared on power-up or normal reset, as shown in Figure 8-12. 0 No invalidation is performed. 1 Initiate invalidation of the entire data cache. The cache controller sequentially clears V and M bits in all sets. Subsequent data accesses stall until the invalidation is finished, at which point, this bit is automatically cleared. In copyback mode, the cache should be flushed using a CPUSHL instruction before setting this bit.
23	DDSP	Data default supervisor-protect. For normal operations that do not hit in the RAMBAR, ROMBAR, or ACRs, this field defines supervisor-protection 0 Not supervisor protected 1 Supervisor protected. User operations cause a fault
22–20	—	Reserved, should be cleared.
19	BEC	Enable branch cache. 0 Branch cache disabled. This may be useful if code is unlikely to be reused. 1 Branch cache enabled.
18	BCINVA	Branch cache invalidate. Invalidation occurs when this bit is written as a 1. Note that branch caches are not cleared on power-up or normal reset. 0 No invalidation is performed. 1 Initiate an invalidation of the entire branch cache.
17–16	—	Reserved, should be cleared.
15	IEC	Enable instruction cache 0 Instruction cache disabled. All instructions and tags in the cache are preserved. 1 Instruction cache enabled.
14	—	Reserved, should be cleared.
13	DNFB	Default cache-inhibited fill buffer 0 Fill buffer does not store cache-inhibited instruction accesses (16 or 32 bits). 1 Fill buffer can store cache-inhibited accesses. The buffer is used only for normal (TT = 0) instruction reads of a cache-inhibited region. Instructions are loaded into the buffer by a burst access (line fill). They stay in the buffer until they are displaced; subsequent accesses may not appear on the external bus. Setting DNFB can cause a coherency problem for self-modifying code. If a cache-inhibited access uses the buffer while DNFB = 1, instructions remain valid in the buffer until a cache-invalidate-all instruction, another cache-inhibited burst, or a miss that initiates a fill. A write to the line in the fill goes to the external bus without updating or invalidating the buffer. Subsequent reads of that written data are serviced by the fill buffer and receive stale information. <b>Note:</b> Motorola discourages the use of self-modifying code.

**Table 8-28. CACR Field Descriptions (Continued)**

Bits	Name	Description
12	IDPI	Instruction CPUSHL invalidate disable. 0 Normal operation. A CPUSHL instruction causes the selected line to be invalidated. 1 No clear operation. A CPUSHL instruction causes the selected line to be left valid.
11	IHLCK	Instruction cache half-lock. 0 Normal operation. The cache allocates to the lowest invalid way; if all ways are valid, the cache allocates to the way pointed at by the round-robin counter and then increments this counter. 1 Half cache operation. The cache allocates to the lowest invalid way of ways 2 and 3; if both of these ways are valid, the cache allocates to way 2 if the high-order bit of the round-robin counter is zero; otherwise, it allocates way 3 and then increments the round-robin counter. This locks the contents of ways 0 and 1. Ways 0 and 1 are still updated on write hits and may be pushed or cleared by specific cache push/invalidate instructions.
10	IDCM	Instruction default cache mode. For normal operations that do not hit in the RAMBARs or ACRs, this field defines the effective cache mode. 0 Cacheable 1 Cache-inhibited
9	—	Reserved, should be cleared.
8	ICINVA	Instruction cache invalidate. Invalidation occurs when this bit is written as a 1. Note the caches are not cleared on power-up or normal reset. 0 No invalidation is performed. 1 Initiate invalidation of instruction cache. The cache controller sequentially clears all V bits. Subsequent local memory bus accesses stall until invalidation completes, at which point ICINVA is cleared automatically without software intervention. For copyback mode, use CPUSHL before setting ICINVA.
7	IDSP	Default instruction supervisor protection bit. For normal operations that do not hit in the RAMBAR, ROMBAR, or ACRs, this field defines supervisor-protection. 0 Not supervisor protected 1 Supervisor protected. User operations cause a fault
6	—	Reserved, should be cleared.
5	EUSP	Enable USP. Enables the use of the user stack pointer. 0 USP disabled. Core uses a single stack pointer. 1 USP enabled. Core uses separate supervisor and user stack pointers.
4	DF	Disable FPU. Determines whether the FPU is enabled. See Section 4.1, “FPU Overview.” 0 FPU enabled. 1 FPU disabled
3–0	—	Reserved, should be cleared.

### 8.7.10.2 Access Control Registers (ACR0–ACR3)

The ACRs, Figure 8-17, assign control attributes, such as cache mode and write protection, to specified memory regions. ACR0 and ACR1 control data attributes; ACR2 and ACR3 control instruction attributes. Registers are accessed with the MOVEC instruction with the Rc encodings in Figure 8-17.

For overlapping data regions, ACR0 takes priority; ACR2 takes priority for overlapping instruction regions. Data transfers to and from these registers are longword transfers.

#### NOTE:

The SIM MBAR region should be mapped as cache-inhibited through an ACR or the CACR.

	31	24	23	16	15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
Field	Address Base				Address Mask				E	S	—	AMM	—	CM	SP		W <sup>1</sup>	—	
Reset	Uninitialized								0	Uninitialized									
R/W	Write (R/W by debug module)																		
Rc	ACR0: 0x004; ACR1: 0x005; ACR2: 0x006; ACR3: 0x007																		

<sup>1</sup> Reserved in ACR2 and ACR3.

**Figure 8-17. Access Control Register Format (ACR<sub>n</sub>)**

Table 8-29 describes ACR<sub>n</sub> fields.

**Table 8-29. ACR<sub>n</sub> Field Descriptions**

Bits	Name	Description
31–24	Address base	Address base. Compared with address bits A[31:24]. Eligible addresses that match are assigned the access control attributes of this register.
23–16	Address mask	Address mask. Setting a mask bit causes the corresponding address base bit to be ignored. The low-order mask bits can be set to define contiguous regions larger than 16 Mbytes. The mask can define multiple noncontiguous regions of memory.
15	E	Enable. Enables or disables the other ACR <sub>n</sub> bits. 0 Access control attributes disabled 1 Access control attributes enabled
14–13	S	Supervisor mode. Specifies whether only user or supervisor accesses are allowed in this address range or if the type of access is a don't care. 00 Match addresses only in user mode 01 Match addresses only in supervisor mode 1x Execute cache matching on all accesses
12–11	—	Reserved, should be cleared.
10	AMM	Address mask mode. 0 The ACR hit function allows control of a 16 Mbytes or greater memory region. 1 The upper 8 bits of the address and ACR are compared without a mask function. Address bits [23:20] of the address and ACR are compared using ACR[19:16] as a mask, allowing control of a 1–16 Mbyte memory region.
9–7	—	Reserved; should be cleared.
6–5	CM	Cache mode. Selects the cache mode and access precision. Precise and imprecise modes are described in Section 8.7.5, “Cache-Inhibited Accesses.” 00 Cacheable, write-through 01 Cacheable, copyback 10 Cache-inhibited, precise 11 Cache-inhibited, imprecise
4	—	Reserved, should be cleared.
3	SP	Supervisor protect. 0 Indicates supervisor and user mode access allowed, reset value is 0 1 Indicates only supervisor access is allowed to this address space and attempted user mode accesses generate an access error exception

**Table 8-29. ACR<sub>n</sub> Field Descriptions (Continued)**

Bits	Name	Description
2	W	ACR0/ACR1 only. Write protect. Selects the write privilege of the memory region. ACR2[W] and ACR3[W] are reserved. 0 Read and write accesses permitted 1 Write accesses not permitted
1–0	—	Reserved, should be cleared.

## 8.7.11 Cache Management

The cache can be enabled and configured by using a MOVEC instruction to access CACR. A hardware reset clears CACR, disabling the cache and removing all configuration information; however, reset does not affect the tags, state information, or data in the cache.

Set CACR[DCINVA,ICINVA] to invalidate the caches before enabling them.

The privileged CPUSHL instruction supports cache management by selectively pushing and invalidating cache lines. The address register used with CPUSHL directly addresses the cache's directory array. The CPUSHL instruction flushes a cache line.

The value of CACR[DDPI,IDPI] determines whether CPUSHL invalidates a cache line after it is pushed. To push the entire cache, implement a software loop to index through all sets and through each of the four ways in each set. The state of CACR[DEC,IEC] does not affect the operation of CPUSHL or CACR[DCINVA,ICINVA]. Disabling a cache by setting CACR[IEC] or CACR[DEC] makes the cache nonoperational without affecting tags, state information, or contents.

The contents of *An* used with CPUSHL specify cache row and line indexes. Figure 8-18 shows the *An* format for the data cache.

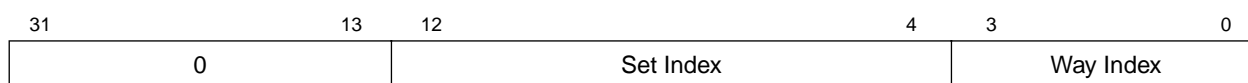
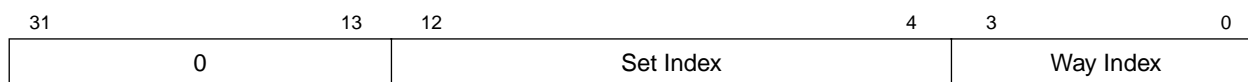
**Figure 8-18. *An* Format (Data Cache)**

Figure 8-19 shows the *An* format for the instruction cache.

**Figure 8-19. *An* Format (Instruction Cache)**

The following code example flushes the entire data cache:

```

_cache_disable:
    nop
    move.w    #0x2700,SR      ;mask off IRQs
    jsr      _cache_flush    ;flush the cache completely
    clr.l    d0
    movec    d0,ACR0         ;ACR0 off

```

```

        movec      d0,ACR1          ;ACR1 off
        move.l     #0x01000000,d0   ;Invalidate and disable cache
        movec      d0,CACR
        rts

_cache_flush:
        nop                      ;synchronize-flush store buffer
        moveq.l    #0,d0           ;initialize way counter
        moveq.l    #0,d1           ;initialize set counter
        move.l     d0,a0           ;initialize cpushl pointer

setloop:
        cpushl     dc,(a0)         ;push cache line a0
        add.l      #0x0010,a0      ;increment set index by 1
        addq.l     #1,d1           ;increment set counter
        cmpi.l     #128,d1         ;are sets for this way done?
        bne        setloop

        moveq.l    #0,d1           ;set counter to zero again
        addq.l     #1,d0           ;increment to next way
        move.l     d0,a0           ;set = 0, way = d0
        cmpi.l     #4,d0           ;flushed all the ways?
        bne        setloop
        rts

```

The following CACR loads assume the instruction cache has been invalidated, the default instruction cache mode is cacheable, and the default data cache mode is copyback.

dataCacheLoadAndLock:

```

        move.l     #0xA3080800,d0   ;enable and invalidate data cache ...
        movec      d0,cacr          ;... in the CACR

```

The following code preloads half of an 8-Kbyte data cache. It assumes a contiguous block of data is to be mapped into the cache, starting at a 0-modulo-8K address.

```

        move.l     #256,d0           ;256 16-byte lines in 4K space
        lea        data_,a0         ;load pointer defining data area
dataCacheLoop:
        tst.b      (a0)             ;touch location + load into data cache
        lea        16(a0),a0        ;increment address to next line
        subq.l     #1,d0            ;decrement loop counter
        bne.b      dataCacheLoop    ;if done, then exit, else continue

;A 4-Kbyte region was loaded into levels 0 and 1 of the 8-Kbyte cache. Lock it!

        move.l     #0xAA088000,d0   ;set the data cache lock bit ...
        movec      d0,cacr          ;... in the CACR
        rts

        align      16

```

The following CACR loads assume the data cache has been invalidated, the default instruction cache mode is cacheable, and the default operand cache mode is copyback.

Note that this function must be mapped into a cache inhibited or SRAM space or these text lines will be prefetched into the instruction cache, which may displace some of the 16-Kbyte space being explicitly fetched.

## Cache Overview

instructionCacheLoadAndLock:

```
    move.l    #0xa2088100,d0    ;enable and invalidate the instruction
    movec     d0,cacr            ;cache in the CACR
```

The following code segments preload half of a 16-Kbyte instruction cache. It assumes a contiguous block of data is to be mapped, starting at a 0-modulo-8K address.

```
    move.l    #512,d0            ;512 16-byte lines in 8K space
    lea       code_,a0          ;load pointer defining code area
instCacheLoop:
;    intouch (a0)                ;touch location + load into instruction cache

;Note in the assembler we use, there is no INTOUCH opcode. The following
;is used to produce the required binary representation

    cpushl    #nc,(a0)          ;touch location + load into
                                ;instruction cache
    lea       16(a0),a0         ;increment address to next line
    subq.l    #1,d0             ;decrement loop counter
    bne.b     instCacheLoop     ;if done, then exit, else continue
;A 8K region was loaded into levels 0 and 1 of the 16-Kbyte instruction cache.
;lock it!

    move.l    #0xa2088800,d0    ;set the instruction cache lock bit
    movec     d0,cacr            ;in the CACR
    rts
```

## 8.7.12 Cache Operation Summary

This section gives operational details for the cache and presents instruction and data cache-line state diagrams.

## 8.7.13 Instruction Cache State Transitions

Because the instruction cache does not support writes, it supports fewer operations than the data cache. As Figure 8-20 shows, an instruction cache line can be in one of two states, valid or invalid. Modified state is not supported. Transitions are labeled with a capital letter indicating the previous state and with a number indicating the specific case listed in Table 8-30. These numbers correspond to the equivalent operations on data caches, described in Section 8.7.13.1, “Data Cache State Transitions.”

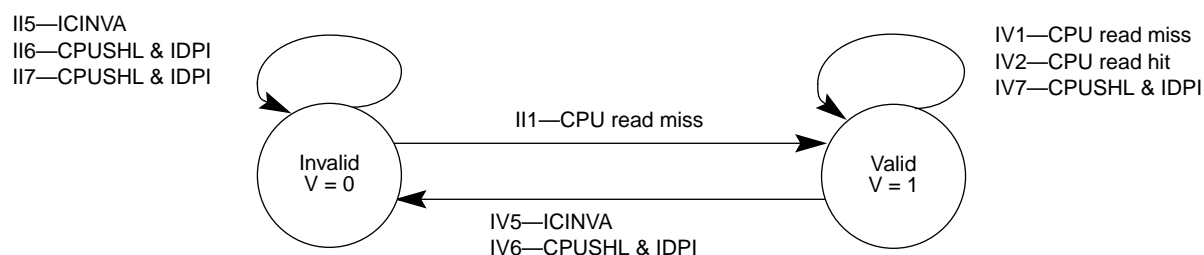


Figure 8-20. Instruction Cache Line State Diagram



Table 8-30 describes the instruction cache state transitions shown in Figure 8-20.

**Table 8-30. Instruction Cache Line State Transitions**

Access	Previous State			
	Invalid (V = 0)		Valid (V = 1)	
Read miss	II1	Read line from memory and update cache; supply data to processor; go to valid state.	IV1	Read new line from memory and update cache; supply data to processor; stay in valid state.
Read hit	II2	Not possible	IV2	Supply data to processor; stay in valid state.
Write miss	II3	Not possible	IV3	Not possible
Write hit	II4	Not possible	IV4	Not possible
Cache invalidate	II5	No action; stay in invalid state.	IV5	No action; go to invalid state.
Cache push	II6, II7	No action; stay in invalid state.	IV6	No action; go to invalid state.
			IV7	No action; stay in valid state.

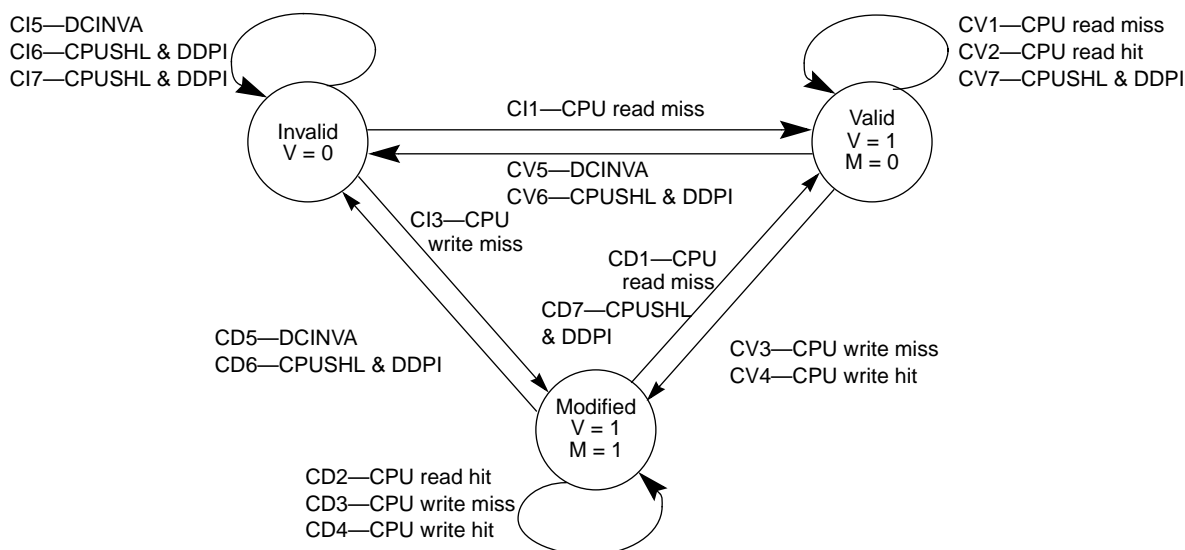
### 8.7.13.1 Data Cache State Transitions

Using the V and M bits, the data cache supports a line-based protocol allowing individual cache lines to be invalid, valid, or modified. To maintain memory coherency, the data cache supports both write-through and copyback modes, specified by the corresponding ACR[CM], or CACR[DDCM] if no ACR matches.

Read or write misses to copyback regions cause the cache controller to read a cache line from memory into the cache. If available, tag and data from memory update an invalid line in the selected set. The line state then changes from invalid to valid by setting the V bit. If all lines in the row are already valid or modified, the pseudo-round-robin replacement algorithm selects one of the four lines and replaces the tag and data. Before replacement, modified lines are temporarily buffered and later copied back to memory after the new line has been read from memory.

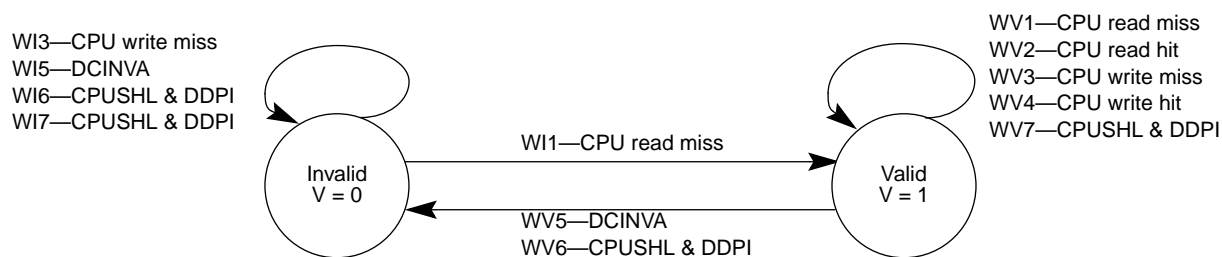
Figure 8-21 shows the three possible data cache line states and possible processor-initiated transitions for memory configured as copyback. Transitions are labeled with a capital letter indicating the previous state and a number indicating the specific case listed in Table 8-21.

## Cache Overview



**Figure 8-21. Data Cache Line State Diagram—Copyback Mode**

Figure 8-22 shows the two possible states for a cache line in write-through mode.



**Figure 8-22. Data Cache Line State Diagram—Write-Through Mode**

Table 8-31 describes data cache line transitions and the accesses that cause them.

**Table 8-31. Data Cache Line State Transitions**

Access	Previous State					
	Invalid (V = 0)		Valid (V = 1, M = 0)		Modified (V = 1, M = 1)	
Read miss	CI1, WI1	Read line from memory and update cache; supply data to processor; go to valid state.	CV1, WV1	Read new line from memory and update cache; supply data to processor; stay in valid state.	CD1	Push modified line to buffer; read new line from memory and update cache; supply data to processor; write push buffer contents to memory; go to valid state.
Read hit	CI2, WI2	Not possible.	CV2, WV2	Supply data to processor; stay in valid state.	CD2	Supply data to processor; stay in modified state.

Table 8-31. Data Cache Line State Transitions (Continued)

Access	Previous State					
	Invalid (V = 0)		Valid (V = 1, M = 0)		Modified (V = 1, M = 1)	
Write miss (copy-back)	CI3	Read line from memory and update cache; write data to cache; go to modified state.	CV3	Read new line from memory and update cache; write data to cache; go to modified state.	CD3	Push modified line to buffer; read new line from memory and update cache; write push buffer contents to memory; stay in modified state.
Write miss (write-through)	WI3	Write data to memory; stay in invalid state.	WV3	Write data to memory; stay in valid state.	WD3	Write data to memory; stay in modified state. Cache mode changed for the region corresponding to this line. To avoid this state, execute a CPUSHL instruction or set CACR[DCINVA,ICINVA] before switching modes.
Write hit (copy-back)	CI4	Not possible.	CV4	Write data to cache; go to modified state.	CD4	Write data to cache; stay in modified state.
Write hit (write-through)	WI4	Not possible.	WV4	Write data to memory and to cache; stay in valid state.	WD4	Write data to memory and to cache; go to valid state. Cache mode changed for the region corresponding to this line. To avoid this state, execute a CPUSHL instruction or set CACR[DCINVA,ICINVA] before switching modes.
Cache invalidate	CI5, WI5	No action; stay in invalid state.	CV5, WV5	No action; go to invalid state.	CD5	No action (modified data lost); go to invalid state.
Cache push	CI6, CI7, WI6, WI7	No action; stay in invalid state.	CV6, WV6	No action; (DDPI = 0) go to invalid state.	CD6	Push modified line to memory; (DDPI = 0) go to invalid state.
			CV7, WV7	No action; (DDPI = 1) stay in valid state.	CD7	Push modified line to memory; (DDPI = 1) go to valid state.

The following tables present the same information as Table 8-31, organized by the previous state of the cache line. In Table 8-32 the previous state is invalid.

**Table 8-32. Data Cache Line State Transitions (Previous State Invalid)**

Access	Response	
Read miss	CI1, WI1	Read line from memory and update cache; supply data to processor; go to valid state.
Read hit	CI2, WI2	Not possible
Write miss (copyback)	CI3	Read line from memory and update cache; write data to cache; go to modified state.
Write miss (write-through)	WI3	Write data to memory; stay in invalid state.
Write hit (copyback)	CI4	Not possible
Write hit (write-through)	WI4	Not possible
Cache invalidate	CI5, WI5	No action; stay in invalid state.
Cache push	CI6, WI6	No action; stay in invalid state.
Cache push	CI7, WI7	No action; stay in invalid state.

Table 8-33 shows transitions when the previous state is valid.

**Table 8-33. Data Cache Line State Transitions (Previous State Valid)**

Access	Response	
Read miss	CV1, WV1	Read new line from memory and update cache; supply data to processor; stay in valid state.
Read hit	CV2, WV2	Supply data to processor; stay in valid state.
Write miss (copyback)	CV3	Read new line from memory and update cache; write data to cache; go to modified state.
Write miss (write-through)	WV3	Write data to memory; stay in valid state.
Write hit (copyback)	CV4	Write data to cache; go to modified state.
Write hit (write-through)	WV4	Write data to memory and to cache; stay in valid state.
Cache invalidate	CV5, WV5	No action; go to invalid state.
Cache push	CV6, WV6	No action; go to invalid state.
Cache push	CV7, WV7	No action; stay in valid state.

Table 8-34 shows transitions when the previous state is modified.

**Table 8-34. Data Cache Line State Transitions (Previous State Modified)**

Access	Response	
Read miss	CD1	Push modified line to buffer; read new line from memory and update cache; supply data to processor; write push buffer contents to memory; go to valid state.
Read hit	CD2	Supply data to processor; stay in modified state.
Write miss (copyback)	CD3	Push modified line to buffer; read new line from memory and update cache; write push buffer contents to memory; stay in modified state.
Write miss (write-through)	WD3	Write data to memory; stay in modified state. Cache mode changed for the region corresponding to this line. To avoid this state, execute a CPUSHL instruction or set CACR[DCINVA,ICINVA] before switching modes.
Write hit (copyback)	CD4	Write data to cache; stay in modified state.
Write hit (write-through)	WD4	Write data to memory and to cache; go to valid state. Cache mode changed for the region corresponding to this line. To avoid this state, execute a CPUSHL instruction or set CACR[DCINVA,ICINVA] before switching modes.
Cache invalidate	CD5	No action (modified data lost); go to invalid state.
Cache push	CD6	Push modified line to memory; go to invalid state.
Cache push	CD7	Push modified line to memory; go to valid state.



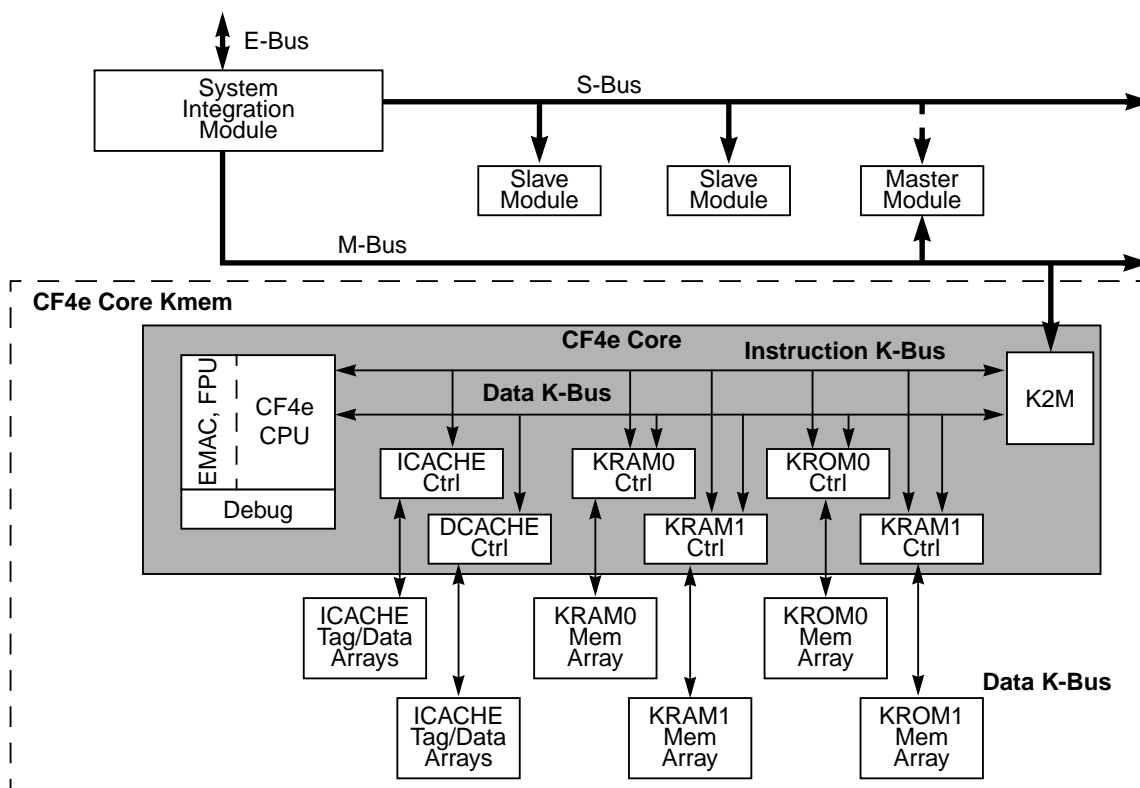
# Chapter 9

## Core Interface

This chapter describes the CF4e core interface and provides an overview of the functional operation of the master bus (M-Bus).

### 9.1 Core Interface Signals

Figure 9-1 is a generic block diagram of a CF4e core interface.



**Figure 9-1. Generic CF4e Block Diagram**

The system buses have the following hierarchy:

- K-Bus—Instruction and operand; processor core and dedicated on-chip memories
- M-Bus—Internal multi-master with centralized arbitration
- S-Bus—Slave module bus controlled by the SIM
- E-Bus—External interface bus

## 9.2 CF4e Pin Characteristics

Table 9-1 provides CF4e core pin characteristics. Most M-Bus and debug input signals are driven directly into input capture registers in the CF4e core. K-Bus memories are specified as synchronous; CF4e core outputs are next-state values registered in the memory device.

### NOTE:

The letter ‘b’ at the end of a name indicates an active-low signal.

**Table 9-1. CF4e Pin Characteristics**

No.	Type	Name	Bus Width <sup>1</sup>	Description
<b>M-Bus Outputs</b>				
1	Output	maddr	[31:0]	M-Bus address
2	Output	mtt	[1:0]	M-Bus transfer type
3	Output	mtm	[2:0]	M-Bus transfer modifier
4	Output	mrwb	—	M-Bus read/write
5	Output	msiz	[1:0]	M-Bus transfer size
6	Output	mwdata	[31:0]	M-Bus write data
7	Output	mwdataoe	—	M-Bus output enable
8	Output	mapb	—	M-Bus address phase
9	Output	mdpb	—	M-Bus data phase
10	Output	mlockb	—	M-Bus locked access
<b>Debug Outputs</b>				
11	Output	bdmforceackb	—	BDM force M-Bus acknowledge
12	Output	cpustopb	—	Processor is stopped
13	Output	cpuhaltb	—	Processor is halted
14	Output	pstclk	—	PST/DDATA clock
<b>Test Outputs</b>				
15	Output	so	[42:0]	Core parallel scan outputs
16	Output	tbso	[7:0]	Test boundary scan outputs
17	Output	bistdone	—	BIST done indicator
18	Output	bistdata	3:0]	BIST bitmap data
19	Output	bistfail	—	BIST memory failure indicator
20	Output	bisthold	—	BIST holding indicator
<b>Outputs to K-Bus Memories</b>				
21	Output	nsientb	—	Next-state I-cache tag enable
22	Output	nsiwrttb	—	Next-state I-cache tag write
23	Output	nsiwlvt	[3:0]	Next-state I-cache tag write level



Table 9-1. CF4e Pin Characteristics (Continued)

No.	Type	Name	Bus Width <sup>1</sup>	Description
24	Output	nsirowst	[9:0]	Next-state I-cache tag address
25	Output	nsiaddrt	[31:9]	Next-state I-cache tag data
26	Output	nsisw	—	Next-state I-cache tag written bit
27	Output	nsisv	—	Next-state I-cache tag valid bit
28	Output	nsiendb	—	Next-state I-cache data enable
29	Output	nsiwrtdb	[3:0]	Next-state I-cache data write level
30	Output	nsiwtbyted	[3:0]	Next-state I-cache data byte write
31	Output	nsirowsd	[11:0]	Next-state I-cache data address
32	Output	nsicwrdata	[31:0]	Next-state I-cache write data
33	Output	nsoentb	—	Next-state D-cache tag enable
34	Output	nsowrttb	—	Next-state D-cache tag write
35	Output	nsowlvt	[3:0]	Next-state D-cache tag write level
36	Output	nsorowst	[9:0]	Next-state D-cache tag address
37	Output	nsoaddrt	[31:9]	Next-state D-cache tag data
38	Output	nsosw	—	Next-state D-cache tag written bit
39	Output	nsosv	—	Next-state D-cache tag valid bit
40	Output	nsoendb	—	Next-state D-cache data enable
41	Output	nsowrtdb	[3:0]	Next-state D-cache data write level
42	Output	nsowtbyted	[3:0]	Next-state D-cache data byte write
43	Output	nsorowsd	[11:0]	Next-state D-cache data address
44	Output	nsocwrdata	[31:0]	Next-state D-cache write data
45	Output	kram0addr	[15:2]	Next-state KRAM0 address
46	Output	kram0di	[31:0]	Next-state KRAM0 write data
47	Output	kram0web	[3:0]	Next-state KRAM0 write enable
48	Output	kram0csb	—	Next-state KRAM0 chip select
49	Output	kram1addr	[15:2]	Next-state KRAM1 address
50	Output	kram1di	[31:0]	Next-state KRAM1 write data
51	Output	kram1web	[3:0]	Next-state KRAM1 write enable
52	Output	kram1csb	—	Next-state KRAM1 chip select
<b>Clock Inputs</b>				
53	Input	mclken	—	Clock phase relationship definer
54	Input	clkfast	—	Processor core clock
<b>More Debug Inputs</b>				
55	Output	krom0csb	—	Next-state KROM 0 chip select
56	Output	krom0addr	[15:2]	Next-state KROM 0 address

Table 9-1. CF4e Pin Characteristics (Continued)

No.	Type	Name	Bus Width <sup>1</sup>	Description
57	Output	krom1csb	—	Next-state KROM 1 chip select
58	Output	krom1addr	[15:2]	Next-state KROM 1 address
59	Output	pstddata	[7:0]	Processor status and debug data
60	Output	dsdo	—	Development system data output
<b>M-Bus Inputs</b>				
61	Input	mrdata <sup>2</sup>	[31:0]	M-Bus read data
62	Input	mtab <sup>3</sup>	—	M-Bus transfer acknowledge
63	Input	mahb <sup>3</sup>	—	M-Bus address hold
64	Input	mip1b <sup>4</sup>	[2:0]	M-Bus interrupt request priority level
<b>Miscellaneous Control and Debug Inputs</b>				
65	Input	mrstib <sup>4</sup>	—	M-Bus reset
66	Input	dsclk <sup>5</sup>	—	Development system clock
67	Input	dsdi <sup>5</sup>	—	Development system data input
68	Input	mbkptb <sup>5</sup>	—	Development system breakpoint
<b>Test and Cache Configuration Inputs</b>				
69	Input	mtmod	[2:0]	Test mode indicators
70	Input	bistreleas	—	BIST release data retention
71	Input	bistmemory	[2:0]	BIST memory select
72	Input	si	[31:0]	Core parallel scan inputs
73	Input	se	—	Core parallel scan enable
74	Input	tbsi	[3:0]	Test boundary scan inputs
75	Input	tbsei	—	Test boundary scan enable—inputs
76	Input	tbseo	—	Test boundary scan enable—outputs
77	Input	tbte	—	Test boundary pcell test enable
78	Input	icsize	[3:0]	Encoded I-cache size
79	Input	ocsize	[3:0]	Encoded D-cache size
<b>Inputs from K-Bus Memories + Memory Configuration Definitions</b>				
80	Input	ictag3do	[31:9]	I-cache level 3 tag data output
81	Input	icw3do	—	I-cache level 3 written bit output
82	Input	icv3do	—	I-cache level 3 valid bit output
83	Input	ictag2do	[31:9]	I-cache level 2 tag data output
84	Input	icw2do	—	I-cache level 2 written bit output
85	Input	icv2do	—	I-cache level 2 valid bit output
86	Input	ictag1do	[31:9]	I-cache level 1 tag data output
87	Input	icw1do	—	I-cache level 1 written bit output

Table 9-1. CF4e Pin Characteristics (Continued)

No.	Type	Name	Bus Width <sup>1</sup>	Description
88	Input	icv1do	—	I-cache level 1 valid bit output
89	Input	ictag0do	[31:9]	I-cache level 0 tag data output
90	Input	icw0do	—	I-cache level 0 written bit output
91	Input	icv0do	—	I-cache level 0 valid bit output
92	Input	iclv13do	[31:0]	I-cache level 3 data output
93	Input	iclv12do	[31:0]	I-cache level 2 data output
94	Input	iclv11do	[31:0]	I-cache level 1 data output
95	Input	iclv10do	[31:0]	I-cache level 0 data output
96	Input	octag3do	[31:9]	D-cache level 3 tag data output
97	Input	ocw3do	—	D-cache level 3 written bit output
98	Input	ocv3do	—	D-cache level 3 valid bit output
99	Input	octag2do	[31:9]	D-cache level 2 tag data output
100	Input	ocw2do	—	D-cache level 2 written bit output
101	Input	ocv2do	—	D-cache level 2 valid bit output
102	Input	octag1do	[31:9]	D-cache level 1 tag data output
103	Input	ocw1do	—	D-cache level 1 written bit output
104	Input	ocv1do	—	D-cache level 1 valid bit output
105	Input	octag0do	[31:9]	D-cache level 0 tag data output
106	Input	ocw0do	—	D-cache level 0 written bit output
107	Input	ocv0do	—	D-cache level 0 valid bit output
108	Input	oclv13do	[31:0]	D-cache level 3 data output
109	Input	oclv12do	[31:0]	D-cache level 2 data output
110	Input	oclv11do	[31:0]	D-cache level 1 data output
111	Input	oclv10do	[31:0]	D-cache level 0 data output
112	Input	enspecialkram	—	Enable special KRAM1 mapping
113	Input	kram0size	[3:0]	Encoded KRAM0 size
114	Input	kram0do	[31:0]	KRAM0 data output
115	Input	kram1size	[3:0]	Encoded KRAM1 size
116	Input	kram1do	[31:0]	KRAM1 data output
117	Input	krom0size	[3:0]	Encoded KROM0 size
118	Input	krom0do	[31:0]	KROM0 data output
119	Input	krom0vldrst	—	KROM0 valid at reset
120	Input	krom1size	[3:0]	Encoded KROM1 size

**Table 9-1. CF4e Pin Characteristics (Continued)**

No.	Type	Name	Bus Width <sup>1</sup>	Description
121	Input	krom1do	[31:0]	KROM1 data output
122	Input	krom1vldrst	—	KROM1 valid at reset

<sup>1</sup> Bus widths are specified using vector notation. A dash (—) in this column indicates a scalar (1-bit) signal.

<sup>2</sup> The MRDATA[31:0] input capture register is only loaded by the termination of an M-Bus data phase.

<sup>3</sup> Because mahb and mtab are driven into combinational logic before being registered, they have a greater setup timing requirement.

<sup>4</sup> miplb[2:0] and mrstib are routed into free-running input capture registers.

<sup>5</sup> dsclk, dsdi, and mbkptb are routed into two levels of free-running registers that serve as synchronizers.

## 9.3 ColdFire Master Bus

The ColdFire architecture implements a hierarchy of buses to provide interconnection and necessary bandwidth among system components such as processors and peripherals. The M-Bus is the system interconnect between multiple masters (including processors) and the system integration module (SIM). The SIM provides additional connectivity to an optional internal S-Bus that contains on-chip peripheral modules and to the external system through the E-Bus. The M-, S-, and E-Buses use a Motorola-defined bus protocol. Providing this bus protocol support allows integration of devices at any level in the system.

The ColdFire architecture supports multiple clock frequency domains. A ColdFire processor can operate at any integer multiple of the M-Bus clock frequency. The M-Bus interface is the boundary from the processor's clock domain to the M-Bus clock domain. The following sections describe specific M-Bus protocols needed to support the multiple clock domains and give system clocking guidelines.

### 9.3.1 M-Bus Signals

Table 9-2 defines required M-Bus signals. These signals are described as viewed by the bus master. Although signal timings are referenced to the system clock, the system clock is not considered a bus signal. Clock routing is expected to meet application requirements.

In this chapter, bus cycle refers to a request to transfer data between the bus master and a slave device.

**Table 9-2. M-Bus Signals**

Name	Direction	Description
maddr[31:0]	Out	Address bus. Address of the first item of a bus transfer for a normal bus cycle.
mahb	In	Address hold. Asserted to indicate that the address and attributes should be held. mahb indicates that the SIM is not ready to accept the address phase of the bus cycle. mahb is also used in bus arbitration to halt the master when it is not granted the M-Bus.
mapb	Out	Address phase. Indicates that the address and attributes are being driven and that the address phase of the bus cycle is active.

**Table 9-2. M-Bus Signals (Continued)**

<b>Name</b>	<b>Direction</b>	<b>Description</b>
mdpb	Out	Data phase. Indicates the data phase of the cycle is active. This means that the bus master drives data during the cycle if the access is a write. During a read, data may be driven back to the bus master. The bus cycle is always terminated during the data phase.
mipb[2:0]	In	Interrupt priority level. Indicates the priority level of a pending interrupt request. 111 No interrupt pending 110 Level 1 101 Level 2 100 Level 3 011 Level 4 010 Level 5 001 Level 6 000 Level 7
mlockb	Out	Locked access. Indicates the current M-Bus cycle is part of a locked, or indivisible, read-modify-write.
mrdata[31:0]	In	Read data bus. Provides a read data path between the SIM and internal masters. The 32-bit read data bus can transfer 8, 16, or 32 bits of data per bus transfer. During a line transfer, data lines are time-multiplexed across multiple cycles to carry 128 bits.
mrstib	In	M-Bus reset. Directs all M-Bus modules (including the core) to enter reset mode.
mrwb	Out	Read/write. Indicates the data transfer direction for the current bus cycle. A high level indicates a read cycle and a low level indicates a write cycle.
msiz[1:0]	Out	Transfer size. Indicate the bus transfer data size. 00 Longword (4 bytes) 01 Byte (1 byte) 10 Word (2 bytes) 11 Line (16 bytes)
mtab	In	Transfer acknowledge. Asserted to indicate successful completion of a requested bus transfer. The CF4e core also generates a debug output signal, bdmforceackb, to help break a hung external bus condition. During debug, an incorrect reference to a memory address may effectively hang the external bus because no slave device responds. In such cases, a new serial BDM command can be sent into the debug module. After decoding this command, the core asserts bdmforceackb for an entire M-Bus clock period. This output may be factored into the external or M-Bus termination logic to unconditionally force a transfer acknowledge so that debug can continue without a system reset.

**Table 9-2. M-Bus Signals (Continued)**

Name	Direction	Description
mtm[2:0]	Out	<p>Transfer modifier. Give supplemental information for each transfer type. For interrupt acknowledge transfers, mtm[2:0] carry the interrupt level being acknowledged. For CPU space transfers, mtm[2:0] are low.</p> <p><u>When mtt[1:0] = 0x—Normal transfers</u></p> <p>000 Reserved  001 User data access  010 User code access  011–100 Reserved  101 Supervisor data access  110 Supervisor code access  111 Reserved</p> <p><u>When mtt[1:0] = 10—Processor emulator mode access</u></p> <p>000–100, 111 Reserved  101 Emulator mode data access  110 Emulator mode code access</p> <p><u>When mtt[1:0] = 11—Acknowledge or CPU space access</u></p> <p>000 CPU space  001 Interrupt level 1 acknowledge  010 Interrupt level 2 acknowledge  011 Interrupt level 3 acknowledge  100 Interrupt level 4 acknowledge  101 Interrupt level 5 acknowledge  110 Interrupt level 6 acknowledge  111 Interrupt level 7 acknowledge</p>
mtt[1:0]	Out	<p>Transfer type. Indicates the type of access of the current bus cycle. The alternate master access is used to indicate a non-core master is requesting the transfer.</p> <p>00 Processor access  01 Alternate master access  10 Processor emulator mode access  11 Acknowledge or CPU space access</p>
mwdata[31:0]	Out	<p>Write data bus. Provides the write data path between an internal master and the SIM. The write data bus is 32 bits wide and can transfer 8, 16, or 32 bits per bus transfer. During a line transfer, the data lines are time-multiplexed across multiple cycles to carry 128 bits.</p>
mwdataoe	Out	<p>Write data bus output enable. ColdFire cores implement unidirectional read and write data buses. If the system designer chooses to implement a bidirectional data bus, mwdataoe can be used to control the three-state enable during write operations."</p>

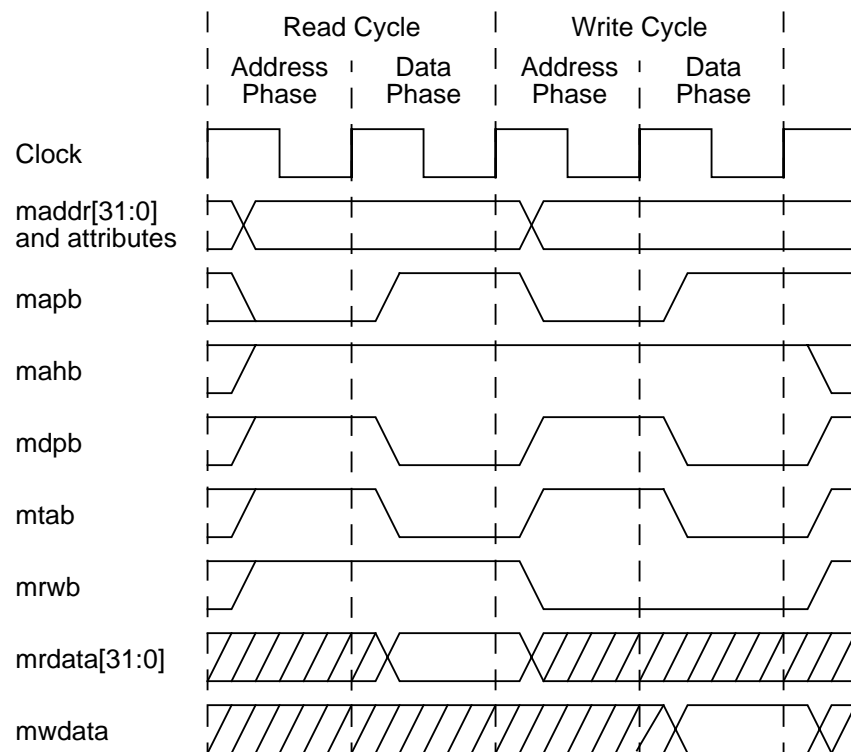
## 9.3.2 M-Bus Operation

The two-stage, synchronous pipelined M-Bus has an effective bandwidth rate of up to one transfer per clock.

### 9.3.2.1 Basic Bus Cycles

The bus transaction is split into two phases. During the address phase, the address (maddr[31:0]) and attribute signals (msiz[1:0], mrwb, mtt[1:0], and mtm[2:0]) are driven. The address phase signal (mapb) is asserted to show that the bus is in the address phase. During the data phase, the data phase (mdpb) signal is asserted until the bus cycle terminates with a transfer acknowledge (mtab). On a write cycle, the write data bus (mwdata) is driven for the entire data phase. On a read cycle, the bus master samples the read data bus (mrdata[31:0]) concurrently with mtab at the rising clock edge. Figure 9-2

shows basic read and write operations.



**Figure 9-2. Basic Read and Write Cycles**

### 9.3.2.2 Pipelined Bus Cycles

Because the bus is pipelined, the address phase of the next bus cycle can become valid while the data phase of the current bus cycle is still valid. Address and data phases cannot be concurrently valid for the same bus cycle. Figure 9-3 shows two basic pipelined bus cycles. For illustration purposes, a read and write cycle are used in Figure 9-3. There are no restrictions on cycles being either reads or writes for them to be pipelined.

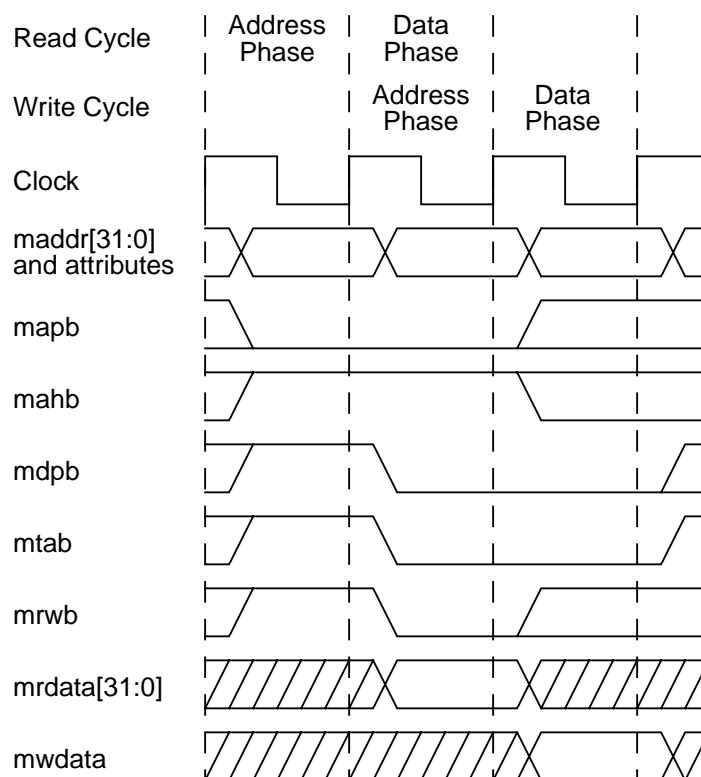


Figure 9-3. Pipelined Read and Write

### 9.3.2.3 Address and Data Phase Interactions

Bus timing, performance, and arbitration are controlled by handling the address and data phases of the bus cycle. The general rules for controlling the phases include the following:

1. The address phase is allowed to begin when there is no active address phase.
2. The address phase is allowed to end and the data phase to begin when the address hold (mahb) signal is not asserted and there is either no active data phase or the active data phase is being terminated.
3. The data phase is allowed to end when the cycle is terminated with mtab.
4. This rule, which is a restriction on the rule 2, applies only to ColdFire processor masters running at the same frequency as the M-Bus, that is, processor and M-Bus clock domains have the same frequency—1X clock mode.

In 1X clock mode, the processor's address phase is allowed to end and the data phase is allowed to begin when the following conditions are met:

- Address hold (mahb) is not asserted
- There is either no active data phase or the active data phase is not from this processor and is being terminated.

That is, for a ColdFire processor operating in 1X clock mode, there must be one M-Bus cycle where that processor's data phase is inactive before its active address phase can progress to a data phase.



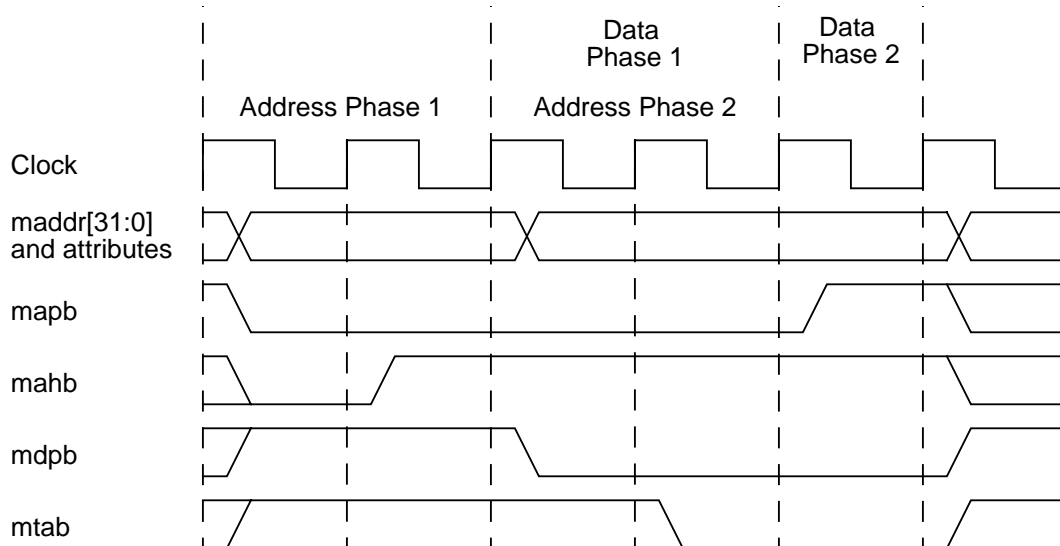
The implications of the general bus rules are as follows:

- The bus master is held off (usually for bus arbitration) by asserting mahb to ensure that the address and attributes remain valid and that the data phase is not entered.
- Pipelining is accomplished by allowing the next address phase to begin during the data phase as soon as the next address is available.
- Wait states are introduced by withholding the termination signal mtab.

The implications of the special 1X clock mode rule are as follows:

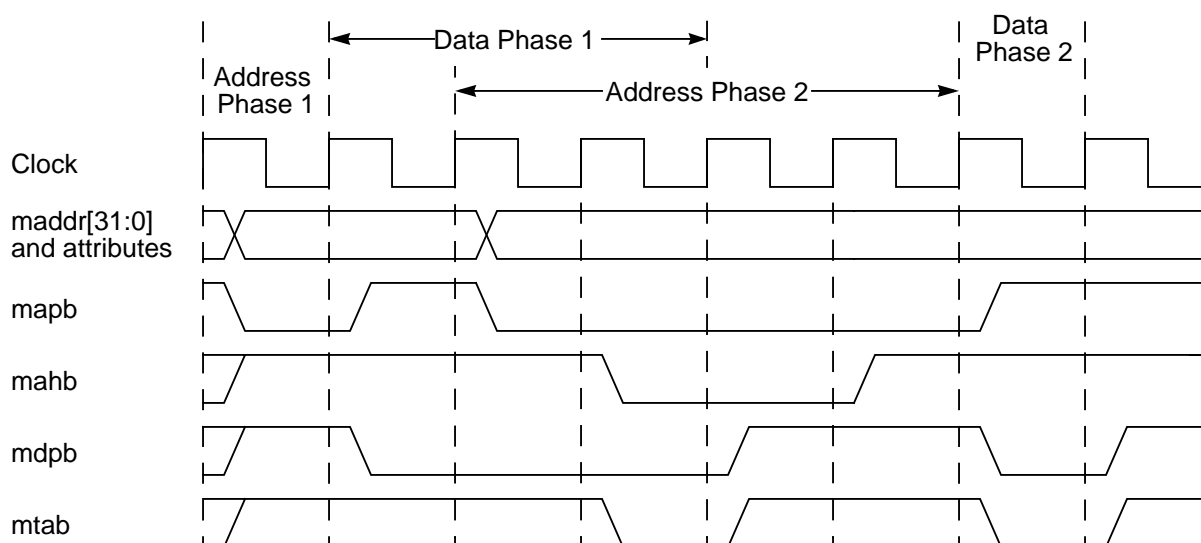
- If a ColdFire processor operating in 1X clock mode has both an active address phase and an active data phase, the M-Bus control module must assert mahb on the last M-Bus transfer acknowledge. This forces the ColdFire processor to hold its address phase until its data phase is idle for at least one cycle.
- A simple implementation of this 1X clock mode rule is to connect mtab from the SIM to both the mtab and mahb inputs ports of the CF4e core design.

Figure 9-4 shows mapb and mahb asserted during the same clock. The address phase is held until mahb is negated, when mdpb is asserted to show the start of data phase 1. Because the address for the next bus cycle is available, mapb stays asserted indicating the start of the address phase 2. A wait state is inserted by delaying mtab until the next clock. In this case, mapb is negated after termination because no other address is available from the bus master. mdpb is not negated because at termination data phase 2 begins. Because the termination signal remains asserted, data phase 2 is only one clock long.



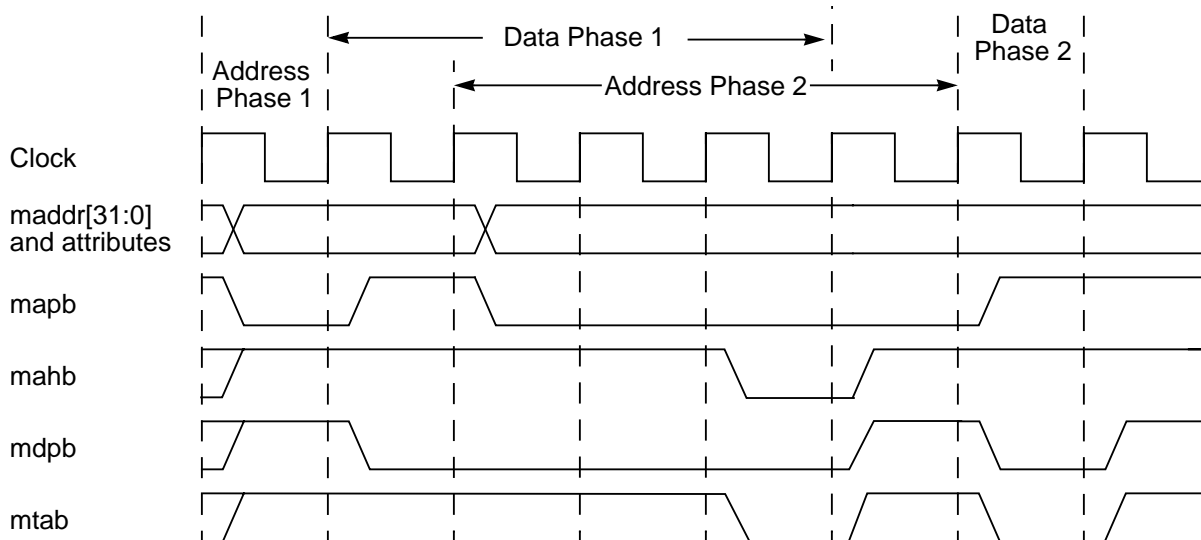
**Figure 9-4. Address Hold Followed By 1- and 0-Wait State Cycles**

Figure 9-5 shows that mapb can be generated in the center of the data phase. It also shows that mahb may be generated while a data phase is active. In this case, the current data phase is completed, but the next cycle is not allowed to transition to the data phase.



**Figure 9-5. mapb and mahb Generated Mid-Data Phase**

Figure 9-6 shows the special rule for 1X clock mode: mahb holds a 1X clock-mode processor in its address phase (address phase 2) on the last mtab of data phase 1.



**Figure 9-6. mahb Generation for 1X Clock Mode**

### 9.3.2.4 Data Size Operations

Table 9-3 shows operand designations for transfers on a byte-boundary system.

**Table 9-3. Processor Operand Representation**

Format	Bits[31:24]	Bits[23:16]	Bits[15:8]	Bits[7:0]
Longword Operand	OP0	OP1	OP2	OP3
Word Operand	—	—	OP2	OP3
Byte Operand	—	—	—	OP3

A bus cycle is a request to transfer data between the bus master and a slave device. To ensure that master and slave devices can handle misaligned operands, the bus architecture must guarantee that each data byte is aligned to the proper lane. For line transfers, data alignment is treated as 4 longword transfers. The next section discusses protocols to handle these transfers. M-Bus transfers assume 32-bit M-Bus devices. The SIM generally handles dynamic sizing to byte or word ports; to support this, M-Bus masters must perform some data replication functions during write cycles. For all transfers, `maddr[31:2]` is the longword base address of the first byte of the reference item. `maddr[1:0]` indicates the byte offset from this address. `msiz[1:0]` and the 2 low-order address bits determine data bus usage. Table 9-4 shows `mrdata` requirements for read transfers.

**Table 9-4. `mrdata` Requirements for Read Transfers**

Size	<code>msiz[1:0]</code>	<code>maddr[1:0]</code>	<code>mrdata[31:24]</code>	<code>mrdata[23:16]</code>	<code>mrdata[15:8]</code>	<code>mrdata[7:0]</code>
Byte	01	00	OP3	Ignored	Ignored	Ignored
	01	01	Ignored	OP3	Ignored	Ignored
	01	10	Ignored	Ignored	OP3	Ignored
	01	11	Ignored	Ignored	Ignored	OP3
Word	10	00	OP2	OP3	Ignored	Ignored
	10	10	Ignored	Ignored	OP2	OP3
Long	00	00	OP0	OP1	OP2	OP3
Line	11	00	OP0	OP1	OP2	OP3

Table 9-5 shows `mwdata` requirements for write transfers.

**Table 9-5. `mwdata` Bus Requirements for Write Transfers**

Size	<code>msiz[1:0]</code>	<code>maddr[1:0]</code>	<code>mwdata[31:24]</code>	<code>mwdata[23:16]</code>	<code>mwdata[15:8]</code>	<code>mwdata[7:0]</code>
Byte	01	00	OP3	Ignored	Ignored	Ignored
	01	01	OP3	OP3	Ignored	Ignored
	01	10	OP3	Ignored	OP3	Ignored
	01	11	OP3	Ignored	Ignored	OP3
Word	10	00	OP2	OP3	Ignored	Ignored
	10	10	OP2	OP3	OP2	OP3
Long	00	00	OP0	OP1	OP2	OP3
Line	11	00	OP0	OP1	OP2	OP3

Table 9-4 and Table 9-5 define all allowable `msiz[1:0]` and `maddr[1:0]` combinations.

### 9.3.2.5 Line Transfers

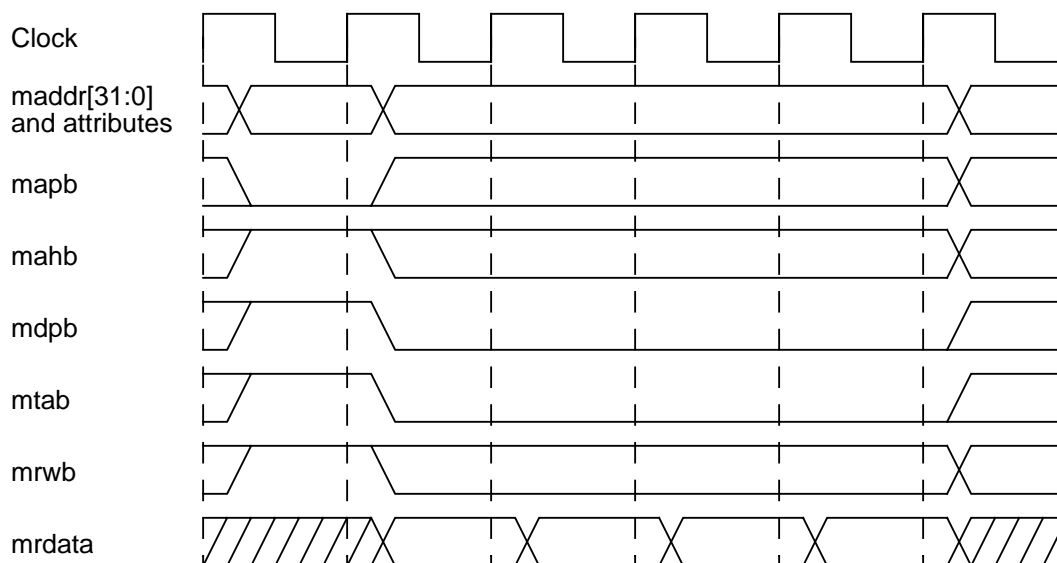
A line is defined as being 16 bytes wide, aligned in memory on 0-modulo-16 address boundary. On the M-Bus, this is seen as an address phase followed by a data phase during which 4 longwords of data are transferred a longword per transfer. Although the line is

aligned on 16-byte boundaries, a line access does not necessarily begin on a 0-modulo-16 address. It can begin at any aligned longword address with  $\text{maddr}[1:0] = 00$ . Therefore, a slave system (combination of the SIM, modules, and external devices) must be able to cycle through the longword addresses. Table 9-6 shows allowable patterns during line accesses.

**Table 9-6. Allowable Line Access Patterns**

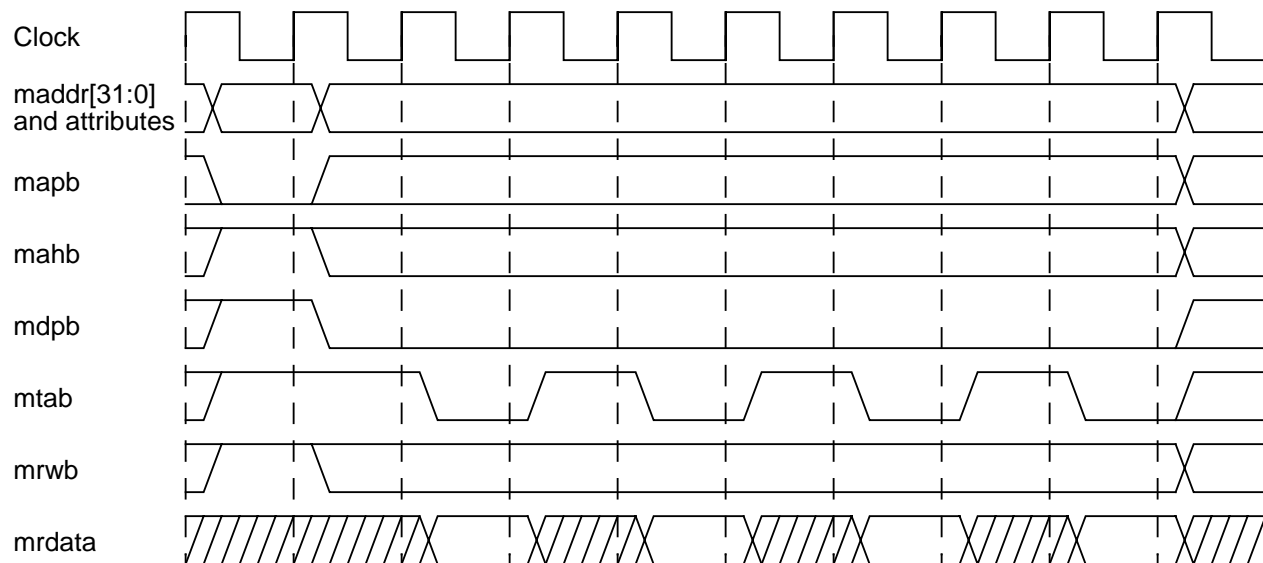
<b>maddr[3:2]</b>	<b>Longword Accesses</b>
00	0x0–0x4–0x8–0xC
01	0x4–0x8–0xC–0x0
10	0x8–0xC–0x0–0x4
11	0xC–0x0–0x4–0x8

Figure 9-7 and Figure 9-8 show line access reads. Note that an address phase for the next bus cycle can be initiated any time during the data phase. Also note that address hold can be asserted during this time without affecting the data phase of a line access. Line accesses complete and the address is held before the next data phase is allowed.



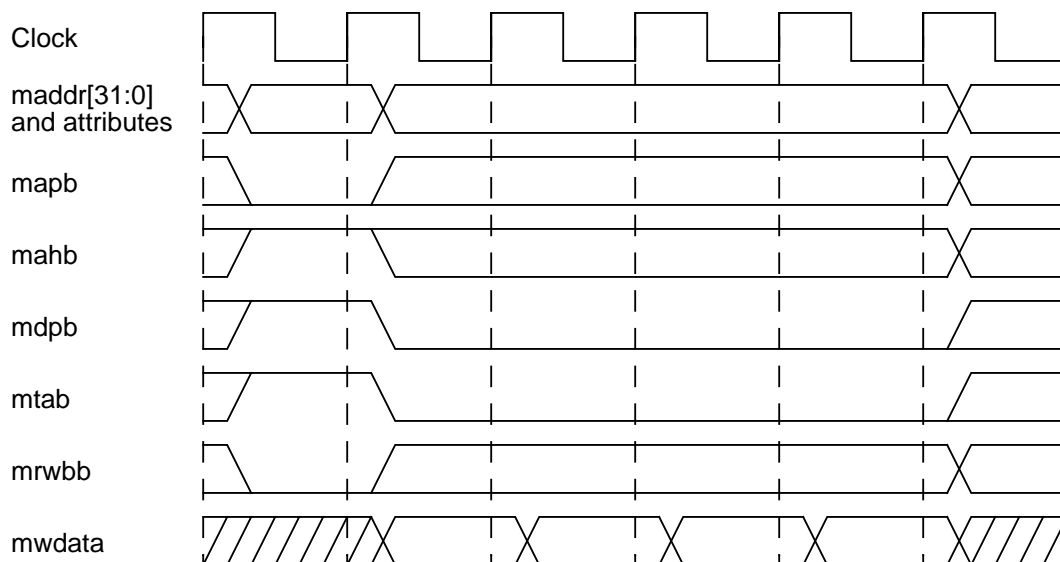
**Figure 9-7. Line Access Read with Zero Wait States**

Figure 9-8 shows a line access read with one wait state.



**Figure 9-8. Line Access Read with One Wait State**

Figure 9-9 and Figure 9-10 show line write accesses. Note that the next longword of data is available on the clock immediately after termination. There may be cases where data may be pipelined to the external bus by terminating the access and registering the data in the SIM during the first clock of the data phase. This allows the next longword of data to be available at the next rising clock edge.



**Figure 9-9. Line Access Write with Zero Wait States**

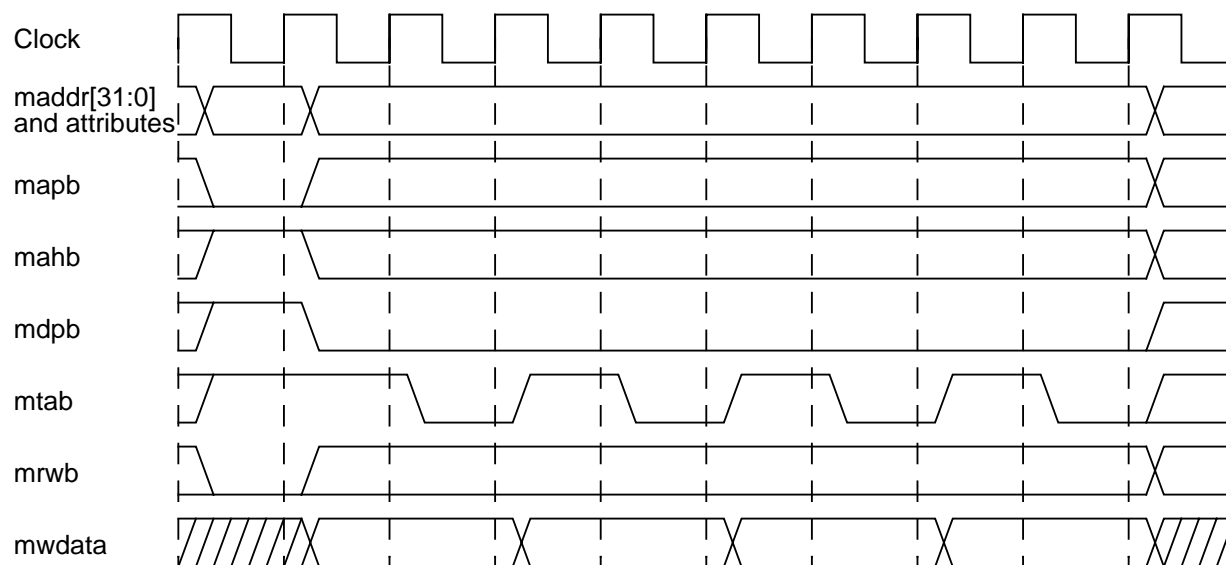


Figure 9-10. Line Access Write with One Wait State

### 9.3.2.6 Bus Arbitration

The arbitration block provides a multiplexed bus scheme to handle multiple M-Bus masters. Multiple masters cannot be on the same physical bus. Figure 9-11 shows the top level architecture of a two-master, multiplexed M-Bus system. The address, attributes, write data, mapb, and mdpb are multiplexed to the SIM. The current bus master's signals are muxed onto the common bus. The termination and address hold signals are demultiplexed and routed to the appropriate bus master. Reset signals and read data need not be multiplexed. Arbitration logic generates address hold to stall a device that is not the current bus master.

The multiplexing scheme was adopted to accommodate a standard cell methodology. There are no three-state or bidirectional signals on the bus, so adding bus masters complicates multiplexing and may affect timing. Designs should limit the number of M-Bus masters. For instance, a three-channel DMA is preferable to three DMA modules on the M-Bus.

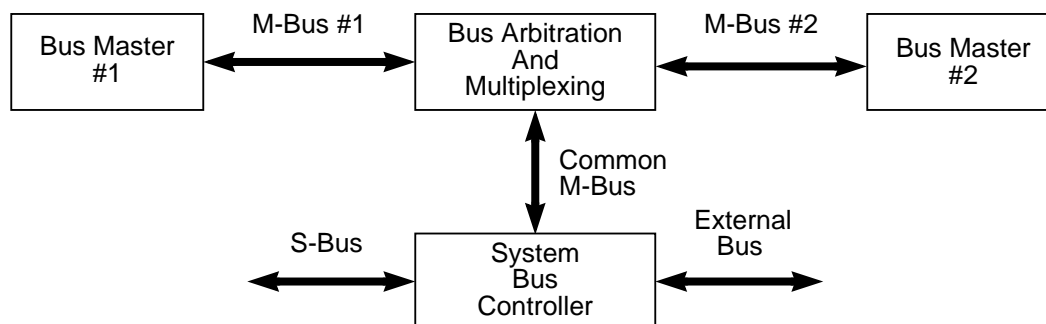
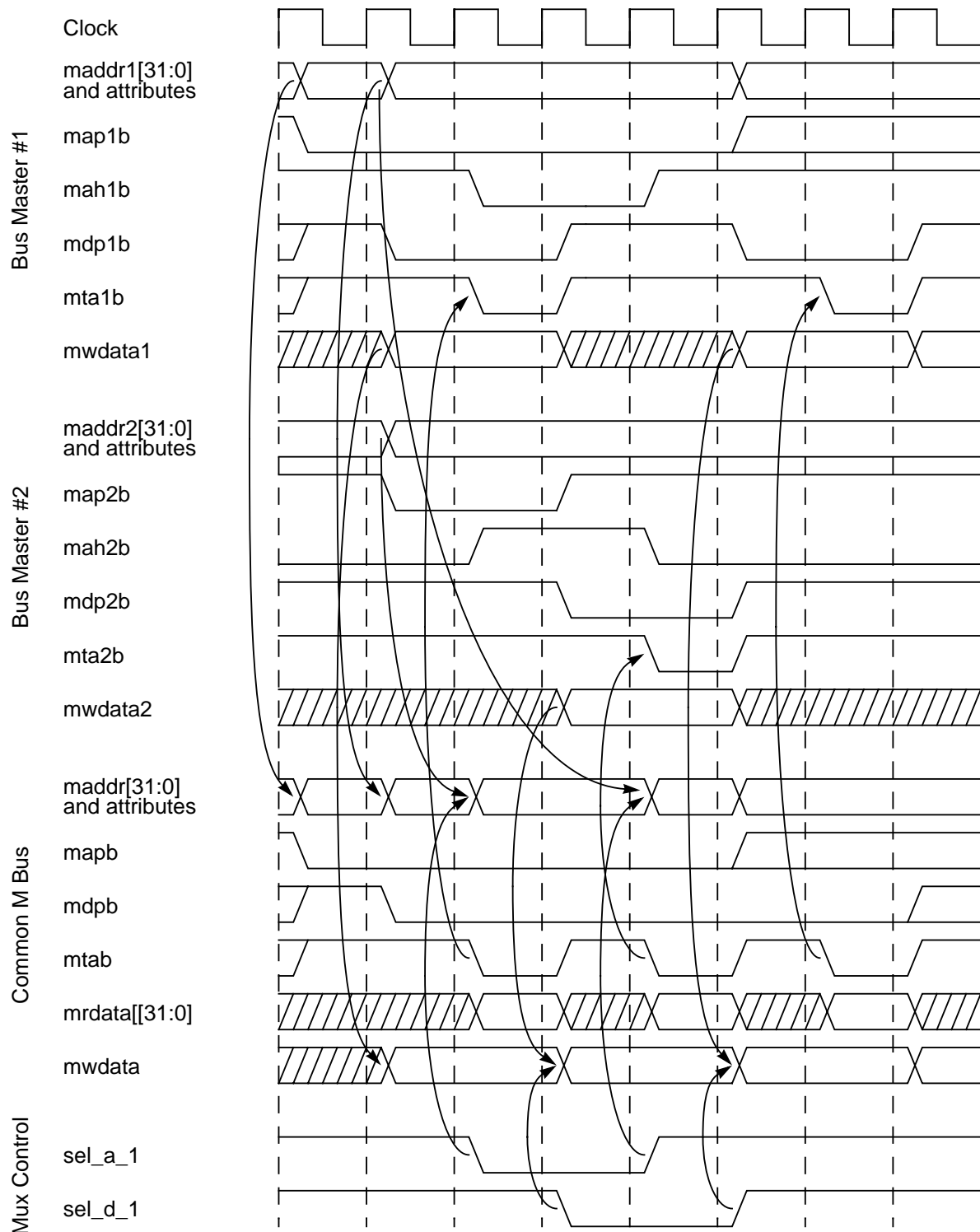


Figure 9-11. Multiplexed M-Bus Structure

Figure 9-11 shows waveforms with two bus masters multiplexed onto a common M-Bus. The exact arbitration scheme and relative priority of bus masters is determined by the

arbitration logic implemented.



**Figure 9-12. Multiplexed M-Bus Operation**

Here, bus master #1 is the default master, such as a core processor. Its mahb is normally high giving it bus access as needed. Bus master #2, a DMA controller for example, requests

the bus by asserting its mapb. The arbiter responds by asserting mah1b to hold off bus master #1. It also transitions sel\_a\_1 (the mux control signal for address, attributes) and mapb. Because an active data phase is on the bus, the data portion of the bus cannot be muxed until that cycle terminates. When sel\_d\_1, the mux control for mwdata, mdpb, and mtab are toggled. Bus master #2 runs its cycle on the common bus, then control returns to bus master #1.

### NOTE:

There is no need to multiplex mrdata. Because the bus master samples data when the data phase is terminated, control of the termination signal is sufficient.

### 9.3.2.7 Interrupt Support

Interrupts are supported on the M-Bus by miplb[2:0] and interrupt acknowledge cycles. When an interrupt is pending, the SIM is responsible for driving miplb[2:0] to the processor to request interrupt processing. The interrupted processor runs an acknowledge cycle to request the interrupt vector to begin exception processing. The interrupt acknowledge cycle looks like a standard byte read cycle. For this cycle, the mtt[1:0] signals indicate an acknowledge cycle (mtt[1:0] = 11) and the interrupt level of the interrupt being processed is specified in the mtm[2:0] signals. Additionally, the address lines maddr[31:5] are all driven high, the interrupt level is reflected on maddr[4:2], and the lower two address bits, maddr[1:0], are zero. The 8-bit interrupt vector is returned on mrdata[31:24].

### 9.3.2.8 Reset Operation

When a master is reset (mrstib is driven low), its M-Bus control signals are driven inactive. This means that mapb, mdpb, mrwb, and mtab are all driven high. However, whether mahb is driven high or low depends on the implementation.



# Chapter 10

## Memory Management Unit (MMU)

This chapter describes the ColdFire virtual memory management unit (MMU), which provides virtual-to-physical address translation and memory access control. The MMU consists of memory-mapped control, status, and fault registers that provide access to translation-lookaside buffers (TLBs). Software can control address translation and access attributes of a virtual address by configuring MMU control registers and loading TLBs. With software support, the MMU provides demand-paged, virtual addressing.

### 10.1 Features

The MMU has the following features:

- MMU memory-mapped control, status, and fault registers
  - Support a flexible, software-defined virtual environment
  - Provide control and maintenance of TLBs
  - Provide fault status and recovery information functions
- Separate, 32-entry, fully associative instruction and data TLBs (Harvard TLBs)
  - Resides in the K-Bus controller
  - Operates in parallel with the K-Bus memories
  - Suffers no performance penalty on TLB hits
  - Supports 1-, 4-, and 8-Kbyte and 1-Mbyte page sizes concurrently
  - Contains register-based TLB entries
- Core extensions:
  - User stack pointer
  - All K-Bus access error exceptions are precise and recoverable
- Harvard TLB provides 97% of baseline performance on large embedded applications using equivalent V4 without MMU support as a baseline.

### 10.2 Virtual Memory Management Architecture

The ColdFire memory management architecture provides a demand-paged, virtual-address environment with hardware address translation acceleration. It supports supervisor/user,

read, write, and execute permission checking on a per-memory request basis. The optional MMU is placed with a software-controlled TLB and associated logic in the core at the K-Bus level of the ColdFire memory/bus hierarchy. Other changes better isolate supervisor and user modes and make core access error exceptions precise.

The architecture defines the MMU TLB, associated control logic, TLB hit/miss logic, address translation based on the TLB contents, and access faults due to TLB misses and access violations. It intentionally leaves some virtual environment details undefined to maximize the software-defined flexibility. These include the exact structure of the memory-resident pointer descriptor/page descriptor tables, the base registers for these tables, the exact information stored in the tables, the methodology (if any) for maintenance of access, and written information on a per-page basis.

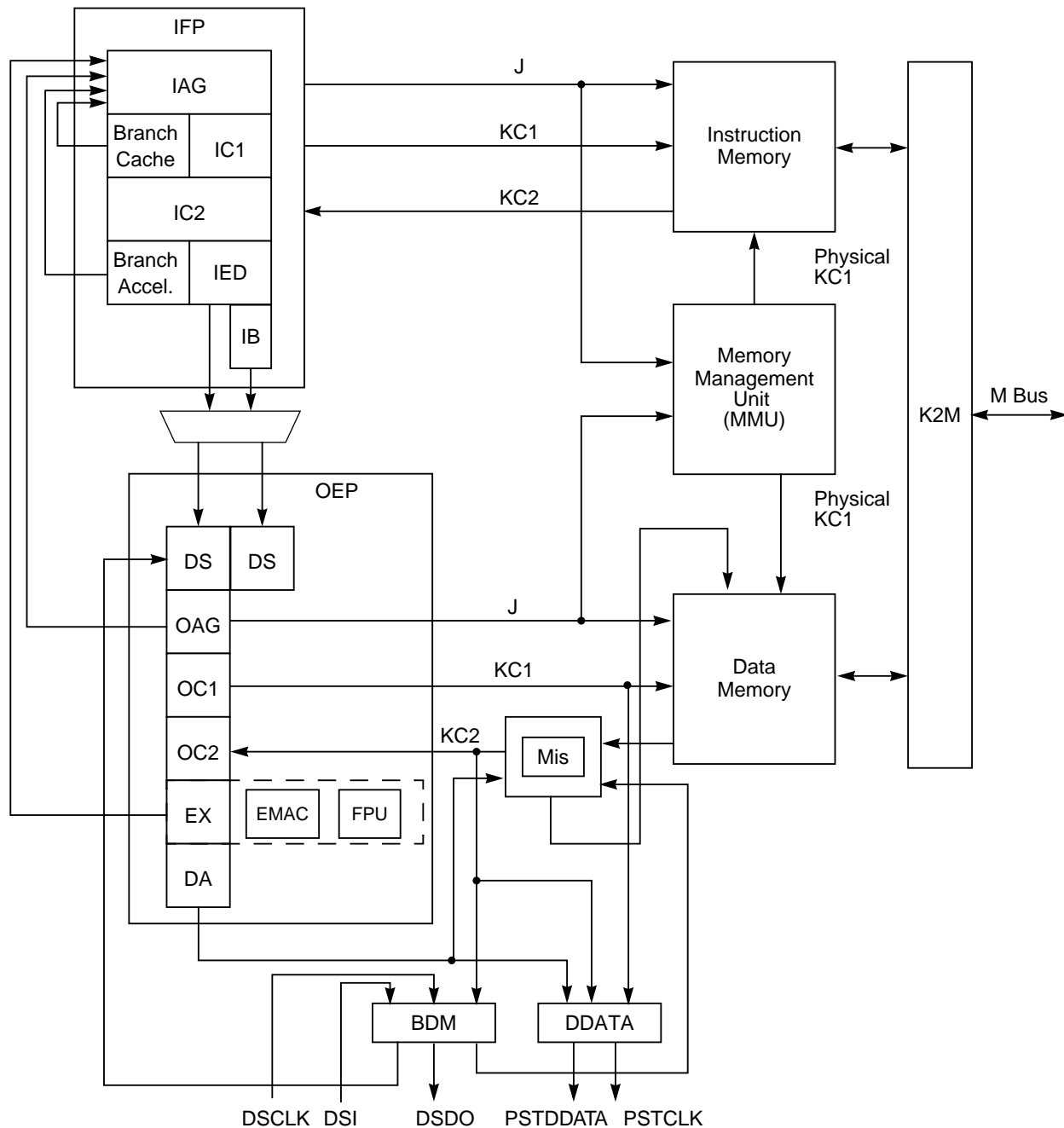
### 10.2.1 MMU Architecture Features

To add optional virtual addressing support, demand-page support, permission checking, and hardware address translation acceleration to the ColdFire architecture, the MMU architecture features the following:

- Addresses from the core to the K-Bus are treated as physical or virtual addresses.
- The address access control logic, address attribute logic, K-Bus memories, and K-Bus to M-Bus controller function as in previous ColdFire versions with the addition of the MMU. The MMU, its TLB, and associated control reside in the K-Bus logic.
- The MMU appears as a memory-mapped device in the K-Bus space. Information for access error fault processing is stored in the MMU.
- A precise K-Bus fault (transfer error acknowledge) signals the core on translation (TLB miss) and access faults. The core supports an instruction restart model for this fault class. Note that this structure uses the existing ColdFire access error fault vector and needs no new ColdFire exception stack frames.
- The following additions are made to the K-Bus memory access control to better support the fault processing and memory maintenance necessary for this virtual addressing environment. These additions improve K-Bus memory performance and functionality for physical and virtual address environments:
  - New supervisor-protect bits to the access control registers (ACRs) and the cache control register (CACR)
  - Improved addressing of the ACRs

### 10.2.2 MMU Architectural Location

Figure 10-1 shows the placement of the MMU/TLB hardware. It follows a traditional model in which it is closely coupled to the processor local-memory controllers.



**Figure 10-1. CF4e Processor Core Block with MMU**

### 10.2.3 MMU Architecture Implementation

This section describes ColdFire design additions and changes for the MMU architecture. It includes precise faults, MMU access, virtual mode, virtual memory references, instruction and data cache addresses, supervisor/user stack pointers, access error stack frame additions, expanded control register space, ACR address improvements, supervisor protection, and debugging in a virtual environment.

### 10.2.3.1 Precise Faults

The MMU architecture performs virtual-to-physical address translation and permission checking in the core on the K-Bus interface. To support demand-paging, the core design provides a precise, recoverable fault for all K-Bus references.

### 10.2.3.2 MMU Access

The MMU TLB control registers are memory-mapped. The TLB entries are read and written indirectly through the MMU control registers. The memory space for these resources is defined by a new supervisor program model register, the MMU base address register (MMUBAR). This register defines a supervisor-mode, data-only space. It has the highest priority for the data K-Bus address mode determination.

### 10.2.3.3 Virtual Mode

Every K-Bus instruction and data reference is either a virtual or physical address mode access. All addresses for special mode (that is, interrupt acknowledges, emulator mode operations, and so on) accesses are physical. All addresses are physical if the optional MMU is not present or not enabled. If the MMU is present and enabled, the address mode for normal accesses is determined by the MMUBAR, RAMBARs, ROMBARs, and ACRs in normal priority order. Addresses that hit in the MMUBAR, RAMBARs, ROMBARs, and ACRs are treated as physical references. These addresses are not translated and their address attributes are sourced from the highest priority mapping register they hit. If an address hits none of these mapping registers, it is a virtual address and is sent to the MMU. If the MMU enabled, the default CACR information is not used.

### 10.2.3.4 Virtual Memory References

The ColdFire MMU architecture references the MMU for all virtual mode accesses to the K-Bus. MMU, KRAM, KROM, and ACR memory spaces are treated as physical address spaces and all permissions that apply to these spaces are contained in the respective mapping register. The virtual mode access either hits or misses in the TLB of the MMU. A TLB miss generates an access fault in the processor, allowing software to either load the appropriate translation into the TLB and restart the faulting instruction or abort the process. Each TLB hit checks permissions based on the access control information in the referenced TLB entry.

### 10.2.3.5 Instruction and Data Cache Addresses

For a given page size, virtual address bits that reference within a page are called the in-page address. All bits above this are the virtual page number. Likewise, the physical address has a physical page number and in-page address bits. Virtual and physical in-page address bits are the same; the MMU translates the virtual page number to the physical page number.

Instruction and data caches are accessed with the untranslated K-Bus address. The translated address is used for cache allocation. That is, caches are virtual-address accessed

and physical-address tagged. If instruction and data cache addresses are not larger than the in-page address for the smallest active MMU page, the cache is considered physically accessed, but if they are larger, the cache can have aliasing problems between virtual and cache addresses. Software handles these problems by forcing the virtual address to be equal to the physical address for those bits addressing the cache but above the in-page address of the smallest active page size. The number of these bits depends on cache and page sizes.

Caches are addressed with the virtual address because the cache uses synchronous memory elements, and an access starts at the rising-clock edge of the first K-Bus pipeline stage. The MMU provides a physical address midway through this cycle.

If the cache set address has fewer bits than the in-page address, the cache is considered physically addressed because these bits are the same in the virtual and physical addresses. If the cache set address has more bits than the in-page address, one or more of the low-order virtual page number bits are used to address the cache. The MMU translates these bits; the resulting low-order physical page number bits are used to determine cache hits.

Address aliasing problems occur when two virtual addresses access one physical page. This is generally allowed and, if the page is cacheable, one coherent copy of the page image is mapped in the cache at any time.

If multiple virtual addresses pointing to the same physical address differ only in the low-order virtual page number bits, conflicting copies can be allocated. For an 8-Kbyte, 4-way set-associative cache with a 16-byte line size, the cache set address uses address bits 10–4. If virtual addresses 0x0\_1000 and 0x0\_1400 are mapped to physical address 0x0\_1000, using virtual address 0x0\_1000 loads cache set 0x00, while using virtual address 0x0\_1400 loads cache set 0x40. This puts two copies of the same physical address in the cache making this memory space not coherent. To avoid this problem, software must force low-order virtual page number bits to be equal to low-order physical address bits for all bits used to address the cache set.

### 10.2.3.6 Supervisor/User Stack Pointers

To isolate supervisor and user modes, CF4e implements two A7 register stack pointers, one for supervisor mode and one for user mode. Two former M680x0 privileged instructions to load and store the user stack pointer are restored in the instruction set architecture.

### 10.2.3.7 Access Error Stack Frame

K-Bus accesses that fault (that is, terminate with a K-Bus transfer error acknowledge) generate an access error exception. MMU TLB misses and access violations use the same fault. To quickly determine if a fault was due to a TLB miss or another type of access error, new fault status field (FS) encodings signal TLB misses on the following:

- Instruction fetch
- Instruction extension fetch
- Data read

- Data write

See Section 10.4.3, “Access Error Stack Frame Additions,” for more information.

### 10.2.3.8 Expanded Control Register Space

MMUBAR is added for ColdFire virtual mode. Like other control registers, it can be accessed from the debug module or written using the privileged MOVEC instruction. See Chapter 2, “Registers.”

### 10.2.3.9 Changes to ACRs and CACR

New ACR and CACR bits, Table 10-1, improve address granularity and supervisor mode protection. These improvements are not necessary to implement the ColdFire MMU but they improve K-Bus memory functionality for physical and virtual address environments.

**Table 10-1. New ACR and CACR Bits**

Bits	Name	Description
ACR $n$ [10]	AMM	Address mask mode. Determines access to the associated address space. 0 The ACR hit function is the same as previous versions, allowing control of a 16-Mbyte or greater memory region. 1 The upper 8 bits of the address and ACR are compared without a mask function; bits 23–20 of the address and ACR are compared masked by ACR[19–16], allowing control of a 1- to 16-Mbyte region. Reset value is 0.
ACR $n$ [3]	SP	Supervisor protect. Determines access to the associated address space. 0 Supervisor and user access allowed. 1 Only supervisor access allowed. Attempted user access causes an access error exception. Reset value is 0.
CACR[23]	DDSP	Default data supervisor protect. Determines access to the associated data space. 0 Supervisor and user access allowed. 1 Only supervisor access allowed. Attempted user access causes an access error exception. Reset value is 0.
CACR[7]	DISP	Default instruction supervisor protect. Determines access to the associated instruction space. 0 Supervisor and user access allowed. 1 Only supervisor access allowed. Attempted user access causes access error exception Reset value is 0.

### 10.2.3.10 ACR Address Improvements

ACRs provide a 16-Mbyte address window. For a given request address, if the ACR is valid and the request mode matches the mode specified in the supervisor mode field, ACR $n$ [S], hit determination is specified as follows:

```
ACRx_Hit = 0;
if ((address[31:24] & ~ACR $n$ [23:16]) == (ACR $n$ [31:24] & ~ACR $n$ [23:16]))
    ACRx_Hit = 1;
```

With this hit function, ACRs can assign address attributes for user or supervisor requests to memory spaces of at least 16 MBytes (through the address mask). With the MMU definition, the ACR hit function is improved by the address mask mode bit (ACR $n$ [AMM]), which supports finer address granularity. See Table 10-1.

The revised hit determination becomes the following:

```
ACRx_Hit = 0;
if (ACRn[10] == 1)
    if ((address[31-24] == ACRn[31-24])) &&
        ((address[23-20] & ~ACRn[19-16]) == (ACRn[23-20] & ~ACRn[19-16]))
        ACRx_Hit = 1;
elseif (address[31-24] & ~ACRn[23-16]) == (ACRn[31-24] & ~ACRn[23-16]))
    ACRx_Hit = 1;
```

### 10.2.3.11 Supervisor Protection

Each K-Bus instruction or data reference is either a supervisor or user access. The CPU's status register supervisor bit (SR[S]) determines the operating mode. New ACR and CACR bits protect supervisor space. See Table 10-1.

## 10.3 Debugging in a Virtual Environment

To support debugging in a virtual environment, numerous enhancements are implemented in the ColdFire debug architecture. These enhancements are collectively called Debug revision D and primarily relate to the addition of an 8-bit address space identifier (ASID) to yield a 40-bit virtual address. This expansion affects two major debug functions:

- The ASID is optionally included in the hardware breakpoint registers specification. For example, the four PC breakpoint registers are expanded by 8 bits each, so that a specific ASID value can be part of the breakpoint instruction address. Likewise, data address/data breakpoint registers are expanded to include an ASID value. The new control registers define whether and how the ASID is included in the breakpoint comparison trigger logic.
- The debug module implements the concept of ownership trace in which an ASID value can be optionally displayed as part of real-time trace. When enabled, real-time trace displays instruction addresses on any change-of-flow instruction that is not absolute or PC-relative. For Debug revision D architecture, the address display is expanded to optionally include ASID contents, thus providing the complete instruction virtual address on these instructions. Additionally, when a Sync\_PC serial BDM command is loaded from the external development system, the processor displays the complete virtual instruction address, including the 8-bit ASID value.

The MMU control registers are accessible through serial BDM commands. See Chapter 11, "Debug Support."

## 10.4 Virtual Memory Architecture Processor Support

To support the MMU, enhancements have been made to the exception model, the stack pointers, and the access error stack frame.

## 10.4.1 Precise Faults

To support demand-paging, all memory references require precise, recoverable faults. The ColdFire instruction restart mechanism ensures that a faulted instruction restarts from the beginning of execution; that is, no internal state information is saved when an exception occurs and none is restored when the handler ends. Given the PC address defined in the exception stack frame, the processor reestablishes program execution by transferring control to the given location as part of the RTE (return from exception) instruction.

For a detailed description, see Section 7.5, “Precise Faults.”

## 10.4.2 Supervisor/User Stack Pointers

To provide the required isolation between these operating modes as dictated by a virtual memory management scheme, a user stack pointer (A7-USP) is added. The appropriate stack pointer register (SSP, USP) is accessed as a function of the processor’s operating mode. In addition, the following two privileged MC680x0 instructions to load/store the USP are added to the ColdFire instruction set architecture:

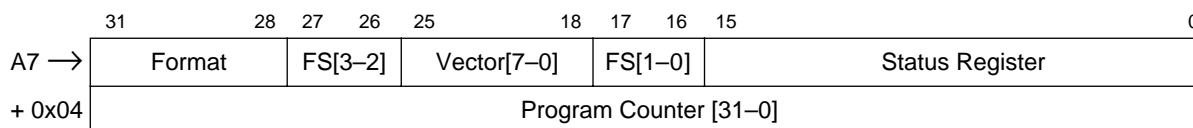
```
mov.l  Ay,USP  # move to   USP: opcode = 0x4E6{0-7}
mov.l  USP,Ax   # move from USP: opcode = 0x4E6{8-F}
```

The address register number is encoded in the low-order three bits of the opcode.

These instructions are described in detail in Section 10.7, “MMU Instructions.”

## 10.4.3 Access Error Stack Frame Additions

ColdFire exceptions generate a standard 2-longword stack frame, signaling the contents of the SR and PC at the time of the exception, the exception type, and a 4-bit fault status field. FS. The first longword contains the 16-bit format/vector word (F/V) and the 16-bit status register. The second contains the 32-bit PC address of the faulted instruction.



**Figure 10-2. Exception Stack Frame**

The FS field is used for access and address errors. To optimize TLB miss exception handling, new FS encodings (Table 10-2) allow quick error classification.

**Table 10-2. Fault Status Encodings**

FS	Definition
0000	Not an access or address error
0001, 001x	Reserved



**Table 10-2. Fault Status Encodings (Continued)**

FS	Definition
0100	Error (for example, protection fault) on instruction fetch
0101	TLB miss on opword of instruction fetch (New in CF4e)
0110	TLB miss on extension word of instruction fetch (New in CF4e)
0111	IFP access error while executing in emulator mode (New in CF4e)
1000	Error on data write
1001	Attempted write of protected space
1010	TLB miss on data write (New in CF4e)
1011	Reserved
1100	Error on data read
1101	Attempted read, read-modify-write of protected space (New in CF4e)
1110	TLB miss on data read, or read-modify-write (New in CF4e)
1111	OEP access error while executing in emulator mode (New in CF4e)

## 10.5 MMU Definition

The ColdFire MMU provides a virtual address, demand-paged memory architecture. The MMU supports hardware address translation acceleration using software-managed TLBs. It enforces permission checking on a per-memory request basis, and has control, status, and fault registers for MMU operation.

### 10.5.1 Effective Address Attribute Determination

The ColdFire core generates an effective memory address for all instruction fetches and data read and write memory accesses. The previous ColdFire memory access control model is based strictly on physical addresses. Every memory request address is a physical address that is analyzed by this logic and assigned address attributes, which include the following:

- Cache mode
- K-Bus RAM (KRAM) enable information
- K-Bus ROM (KROM) enable information
- Write protect information
- Write mode information

These attributes control processing of the memory request. The address itself is not affected by memory access control logic.

Instruction and data references base effective address attributes and access mode on the instruction type and the effective address. K-Bus accesses are of the following two types:

- Special mode accesses, including interrupt acknowledges, reads/writes to program-visible control registers (such as CACR, ROMBARs, RAMBARs, and ACRs), cache control commands (CPUSHL and INTOUCH), and emulator mode operations. These accesses have specific attributes such as the following:
  - Non-cacheable
  - Precise
  - No write protectionUnless the CPU space/IACK mask bit is set, interrupt acknowledge cycles and emulator mode operations are allowed to hit in RAMBARs and ROMBARs. All other operations are normal mode accesses.
- Normal mode accesses. For these accesses, an effective cache mode, precision and write-protection are calculated for each request.

For the data K-Bus, a normal mode access address is compared with the following priority, from highest to lowest: RAMBAR0, RAMBAR1, ROMBAR0, ROMBAR1, ACR0, and ACR1. If no match is found, default attributes in the CACR are used. The priority for instruction K-Bus accesses is RAMBAR0, RAMBAR1, ROMBAR0, ROMBAR1, ACR2, and ACR3. Again, if no match is found, default CACR attributes are used.

Only the test-and-set (TAS) instruction can generate a normal mode access with implied cache mode and precision. TAS is a special, byte-sized, read-modify-write instruction used in synchronization routines. A TAS data access that does not hit in the RAMBARs is non-cacheable and precise. TAS uses the normal effective write protection.

The ColdFire MMU is an optional enhancement to the memory access control. If the MMU is present and enabled, it adds two factors for calculating effective address attributes:

- MMUBAR defines a memory-mapped, privileged data-only space with the highest priority in effective address attribute calculation for the data K-Bus (that is, the MMUBAR has priority over RAMBAR0).
- If virtual mode is enabled, any normal mode access that does not hit in the MMUBAR, RAMBARs, ROMBARs, or ACRs is considered a normal mode virtual address request and generates its access attributes from the MMU. For this case, the default CACR address attributes are not used.

The MMU also uses TLB contents to perform virtual-to-physical address translation.

## 10.5.2 MMU Functionality

The MMU provides virtual-to-physical address translation and memory access control. The MMU consists of memory-mapped, control, status, and fault registers and a TLB that can be accessed through MMU registers. Supervisor software can access these resources through MMUBAR. Software can control address translation and access attributes of a virtual address by configuring MMU control registers and loading the MMU's TLB, which functions as a cache, associating virtual addresses to corresponding physical addresses and

providing access attributes. Each TLB entry maps a virtual page. Several page sizes are supported. Features such as clear all and probe for hit help maintain TLBs.

Fault-free, virtual address accesses that hit in the TLB incur no pipeline delay. Accesses that miss the TLB or hit the TLB but violate an access attribute generate an access error exception. On an access error, software can reference address and information registers in the MMU to retrieve data. Depending on the fault source, software can obtain and load a new TLB entry, modify the attributes of an existing entry, or abort the faulting process.

## 10.5.3 MMU Organization

Access to the MMU memory-mapped region is controlled by MMUBAR, a 32-bit supervisor control register at 0x008 that is accessed using MOVEC or the serial BDM debug port. The *PRM* describes the MOVEC instruction.

### 10.5.3.1 MMU Base Address Register (MMUBAR)

Figure 10-3 shows MMUBAR. The default reset state is an invalid MMUBAR, so that the MMU is disabled and the memory-mapped space is not visible.

	31	16	15	1	0
Field	BA			—	V
Reset	—				0
R/W	R/W				
Rc	0x008				

**Figure 10-3. MMU Base Address Register**

Table 10-3 describes MMU base address register fields.

**Table 10-3. MMU Base Address Register Field Descriptions**

Bits	Name	Description
31–16	BA	Base address. Defines the base address for the 64-Kbyte address space mapped to the MMU.
15–1	—	Reserved, should be cleared. Writes are ignored and reads return zeros.
0	V	Valid. Indicates when MMUBAR contents are valid. BA is not used unless V is set. 0 MMUBAR contents are not valid. 1 MMUBAR contents are valid.

### 10.5.3.2 MMU Memory Map

MMUBAR holds the base address for the 64-Kbyte MMU memory map, shown in Table 10-4. The MMU memory map area is not visible unless the MMUBAR is valid and must be referenced aligned. A large portion of the map is reserved for future use.

**Table 10-4. MMU Memory Map**

Offset from MMUBAR	Name
+ 0x0000	MMU control register (MMUCR)
+ 0x0004	MMU operation register (MMUOR)
+ 0x0008	MMU status register (MMUSR)
+ 0x000C	Reserved
+ 0x0010	MMU fault, test, or TLB address register (MMUAR)
+ 0x0014	MMU read/write TLB tag register (MMUTR)
+ 0x0018	MMU read/write TLB data register (MMUDR)
+ 0x001C–0xFFFC	Reserved <sup>1</sup>

<sup>1</sup>May be used for implementation-specific information/control registers.

The address space ID (ASID) is located in a CPU space control register. The 8-bit ASID value located in the low order byte of a 32-bit supervisor control register, mapped into CPU space at address 0x003 and accessed using a MOVEC instruction. The *ColdFire Family Programmer's Reference Manual* describes MOVEC.

This 8-bit field is the current user ASID. The ASID is an extension to the virtual address. Address space 0x00 may be reserved for supervisor mode. See address space mode functionality in Section 10.5.3.3, “MMU Control Register (MMUCR).” The other 255 address spaces are used to tag user processes. The TLB entry ASID values are compared to this value for user mode unless the TLB entry is marked shared (MMUTR[SG] is set). The TLB entry ASID value may be compared to 0x00 for supervisor accesses.

### 10.5.3.3 MMU Control Register (MMUCR)

MMUCR, Figure 10-4, has the address space mode and virtual mode enable bits. The user must force pipeline synchronization after writing to this register. Therefore, all writes to this register must be immediately followed by a NOP instruction.

	31		2	1	0
Field	—			ASM	EN
Reset	—				0
R/W	R/W				
Rc	0x000				

**Figure 10-4. MMU Control Register (MMUCR)**

Table 10-5 describes MMUCR fields.

**Table 10-5. MMUCR Field Descriptions**

Bits	Name	Description
31–2	—	Reserved, should be cleared. Writes are ignored and reads return zeros.
1	ASM	Address space mode. Controls how the address space ID is used for TLB hits. 0 TLB entry ASID values are compared to the address space ID register value for user or supervisor mode unless the TLB entry is marked shared (MMUTR[SG] = 1). The address space ID register value is the effective address space for all requests, supervisor and user. 1 Address space 0x00 is reserved for supervisor mode and the effective address space is forced to 0x00 for all supervisor accesses. The other 255 address spaces are used to tag user processes. The TLB entry ASID values are compared to the address space ID register for user mode unless the TLB entry is marked shared (SG = 1). The TLB entry ASID value is always compared to 0x00 for supervisor accesses. This allows two levels of sharing. All users but not the supervisor share an entry if SG = 1 and ASID ≠ 0. All users and the supervisor share an entry if SG = 1 and ASID = 0
0	EN	Virtual mode enabled. Indicates when virtual mode is enabled. 0 Virtual mode is disabled. 1 Virtual mode is enabled.

### 10.5.3.4 MMU Operation Register (MMUOR)

Figure 10-5 shows the MMU operation register.

	31	16	15	9	8	7	6	5	4	3	2	1	0			
Field	AA					—		STLB	CA	CNL	CAS	ITLB	ADR	R/W	ACC	UAA
Reset	—															
R/W	Read Only					R/W										
Rc	0x0004															

**Figure 10-5. MMU Operation Register (MMUOR)**

Table 10-6 describes MMUOR fields.

**Table 10-6. MMUOR Field Descriptions**

Bits	Name	Description
31–16	AA	TLB allocation address. This read-only field is maintained by MMU hardware. Its range and format depend on the TLB implementation (specific TLB size in entries, associativity, and organization). The access TLB function can use AA to read or write the addressed TLB entry. The MMU loads AA on the following three events: <ul style="list-style-type: none"> <li>On DTLB access errors, it loads the address of the TLB entry that caused the error.</li> <li>If UAA is set, it loads the address of the TLB entry chosen by the MMU for replacement.</li> <li>If STLB is set, it uses the data in MMUAR to search the TLB and if the TLB hits, loads the address of the TLB entry that hits, or if the TLB misses, loads the TLB entry chosen by the MMU for replacement.</li> </ul> The MMU never picks a locked entry for replacement, and TLB hits of locked entries do not update hardware replacement algorithm information. This is so access error handlers mapped with locked TLB entries do not influence the replacement algorithm. Further, TLB search operations do not update the hardware replacement algorithm information while TLB writes (loads) do update the hardware replacement algorithm information. The algorithm used to choose the allocation address depends on the TLB implementation (such as LRU, round-robin, pseudo-random).
15–9	—	Reserved, should be cleared. Writes are ignored and reads return zeros.

**Table 10-6. MMUOR Field Descriptions (Continued)**

Bits	Name	Description
8	STLB	Search TLB. STLB always reads as zero. 0 No operation 1 The MMU searches the TLB using data in MMUAR. This operation updates the probe TLB hit bit in the status register plus loads the AA field as described above.
7	CA	Clear all TLB entries. CA always reads as zero. 0 No operation 1 Clear all TLB entries and all hardware TLB replacement algorithm information.
6	CNL	Clear all non-locked TLB entries. Setting CNL clears all TLB entries that do not have their locked bit set. CNL always reads as zero. 0 No operation 1 Clear all non-locked TLB entries.
5	CAS	Clear all non-locked TLB entries that match ASID. CAS is always reads as a zero. 0 No operation 1 Clear all non-locked TLB entries that match ASID register.
4	ITLB	ITLB operation. Used by TLB search and access operations that use the TLB allocation address. 0 The MMU uses the DTLB to search or update the allocation address. 1 The MMU uses the ITLB for searches and updates of the allocation address.
3	ADR	TLB address select. Indicates which address to use when accessing the TLB. 0 Use the TLB allocation address for the TLB address. 1 Use MMUAR for the TLB address.
2	R/W	TLB access read/write select. Indicates whether to do a read or a write when accessing the TLB. 0 Write 1 Read
1	ACC	MMU TLB access. This bit always reads as a zero. STLB is used for search operations. 0 No operation. ACC should be a zero to search the TLB. 1 The MMU reads or writes the TLB depending on R/W. For TLB reads, TLB tag and data results are loaded into MMUTR and MMUDR. For TLB writes, the contents of these registers are written to the TLB. The TLB is accessed using the TLB allocation address if ADR is zero or using MMUAR if ADR is set.
0	UAA	Update allocation address. UAA always reads as a zero. 0 No operation 1 MMU updates the allocation address field with the MMU's choice for the allocation address in the ITLB or DTLB depending on the ITLB instruction operation bit.

### 10.5.3.5 MMU Status Register (MMUSR)

MMUSR, Figure 10-6, is updated on all data access faults and search TLB operations.

	31						6	5	4	3	2	1	0
Field	—							SPF	RF	WF	—	HIT	—
Reset	—												
R/W	R/W												
Rc	0x0008												

**Figure 10-6. MMU Status Register (MMUSR)**

Table 10-7 describes MMUSR fields.

**Table 10-7. MMUSR Field Descriptions**

Bits	Name	Description
31–6	—	Reserved, should be cleared. Writes are ignored and reads return zeros.
5	SPF	Supervisor protect fault. Indicates if the last data fault was a user mode access that hit in a TLB entry that had its supervisor protect bit set. 0 Last data access fault did not have a supervisor protect fault. 1 Last data access fault had a supervisor protect fault.
4	RF	Read access fault. Indicates if the last data fault was an data read access that hit in a TLB entry that did not have its read bit set. 0 Last data access fault did not have a read protect fault. 1 Last data access fault had a read protect fault.
3	WF	Write access fault. Indicates if the last data fault was an data write access that hit in a TLB entry that did not have its write bit set. 0 Last data access fault did not have a write protect fault. 1 Last data access fault had a write protect fault.
2	—	Reserved, should be cleared. Writes are ignored and reads return zeros.
1	HIT	Search TLB hit. Indicates if the last data fault or the last search TLB operation hit in the TLB. 0 Last data access fault or search TLB operation did not hit in the TLB. 1 Last data access fault or search TLB operation hit in the TLB.
0	—	Reserved, should be cleared. Writes are ignored and reads return zeros.

### 10.5.3.6 MMU Fault, Test, or TLB Address Register (MMUAR)

The MMUAR format, Figure 10-7, depends on how the register is used.

	31	0
Field	FA	
Reset	—	
R/W	R/W	
Rc	0x0010	

**Figure 10-7. MMU Fault, Test, or TLB Register (MMUAR)**

Table 10-8 describes MMUAR fields.

**Table 10-8. MMUAR Field Descriptions**

Bits	Name	Description
31–0	FA	Form address. Written by the MMU with the virtual address on DTLB misses and access faults. For this case, all 32 bits are address bits. This register may be written with a virtual address and address attribute information for searching the TLB (MMUCR[STLB]). For this case, FA[31–1] are the virtual page number and FA[0] is the supervisor bit. The current ASID is used for the TLB search. MMUAR can also be written with a TLB address for use with the access TLB function (using MMUCR[ACC]).

### 10.5.3.7 MMU Read/Write Tag and Data Entry Registers (MMUTR and MMUDR)

Each TLB entry consists of a 32-bit TLB tag entry and a 32-bit TLB data entry. TLB entries are referenced through MMUTR and MMUDR. For read TLB accesses, the contents of the TLB tag and data entries referenced by the allocation address or MMUAR are loaded in

## MMU Definition

MMUTR and MMUDR. TLB write accesses place MMUTR and MMUDR contents into the TLB tag and data entries defined by the allocation address or MMUAR.

MMUTR, Figure 10-8, contains the virtual address tag, the address space ID (ASID), a shared page indicator, and the valid bit.

	31		10	9		2	1	0
Field	VA				ID		SG	V
Reset	—							
R/W	R/W							
Rc	0x0014							

**Figure 10-8. MMU Read/Write TLB Tag Register (MMUTR)**

Table 10-9 describes MMUTR fields.

**Table 10-9. MMUTR Field Descriptions**

Bits	Name	Description
31–10	VA	Virtual address. Defines the virtual address mapped by this entry. The number of bits used in the TLB hit determination depends on the page size field in the corresponding TLB data entry.
9–2	ID	Address space ID (ASID). This extension to the virtual address marks this entry as part of 1 of 256 possible address spaces. Address space 0x00 can be reserved for supervisor mode. The other 255 address spaces are used to tag user processes. TLB entry ASID values are compared to the ASID register value for user mode unless the TLB entry is marked shared (SG = 1). The TLB entry ASID value may be compared to 0x00 for supervisor accesses or to the ASID. The description of MMUCR[ASM] in Table 10-5 gives details on supervisor mode and ASID compares.
1	SG	Shared global. Indicates when the entry is shared among user address spaces. If an entry is shared, its ASID is not part of the TLB hit determination for user accesses. 0 This entry is not shared globally. 1 This entry is shared globally. Note that the ASID can be used to determine supervisor mode hits to allow two sharing levels. If SG and MMUCR[ASM] are set and the ASID is not zero, all users (but not the supervisor) share an entry. If SG and MMUCR[ASM] are set and the ASID is zero, all users and the supervisor share an entry. The description of ASM in Table 10-5 details supervisor mode and ASID compares.
0	V	Valid. Indicates when the entry is valid. Only valid entries generate a TLB hit. 0 Entry is not valid. 1 Entry is valid.

MMUDR, Figure 10-9, contains the physical address, cache mode field, page size, supervisor-protect bit, read, write, execute permission bits, and lock-entry bit.

	31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																								
--	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

**Figure 10-9. MMU Read/Write TLB Data Register**

Table 10-9 describes MMUDR fields.



Table 10-10. MMUDR Field Descriptions

Bits	Name	Descriptions															
31–10	PA	Physical address. Defines the physical address which is mapped by this entry. The number of bits used to build the effective physical address if this TLB entry hits depends on the page size field.															
9–8	SZ	<p>Page size. Page size for this entry.</p> <table> <tr> <th>SZ</th><th>Page Size</th><th>Function</th></tr> <tr> <td>00</td><td>1 Mbyte</td><td>VA[31–20] used for TLB hit</td></tr> <tr> <td>01</td><td>4 Kbyte</td><td>VA[31–12] used for TLB hit</td></tr> <tr> <td>10</td><td>8 Kbyte</td><td>VA[31–13] used for TLB hit</td></tr> <tr> <td>11</td><td>1 Kbyte</td><td>VA[31–10] used for TLB hit</td></tr> </table>	SZ	Page Size	Function	00	1 Mbyte	VA[31–20] used for TLB hit	01	4 Kbyte	VA[31–12] used for TLB hit	10	8 Kbyte	VA[31–13] used for TLB hit	11	1 Kbyte	VA[31–10] used for TLB hit
SZ	Page Size	Function															
00	1 Mbyte	VA[31–20] used for TLB hit															
01	4 Kbyte	VA[31–12] used for TLB hit															
10	8 Kbyte	VA[31–13] used for TLB hit															
11	1 Kbyte	VA[31–10] used for TLB hit															
7–6	CM	<p>Cache mode. If a Harvard TLB implementation is used, CM0 is a don't care for the ITLB. CM is ignored on writes and always reads as zero for the ITLB.</p> <p>Instruction cache modes:</p> <p>1x Page is non-cacheable.</p> <p>0x Page is cacheable.</p> <p>Data cache modes</p> <p>00 Page is cacheable writethrough.</p> <p>01 Page is cacheable copyback.</p> <p>10 Page is non-cacheable precise.</p> <p>11 Page is non-cacheable imprecise.</p>															
5	SP	<p>Supervisor protect. Controls user mode access to the page mapped by this entry.</p> <p>0 Entry is not supervisor protected.</p> <p>1 Entry is supervisor protected. An attempted user mode access that matches this entry generates an access error exception.</p>															
4	R	<p>Read access enable. Indicates if data read accesses to this entry are allowed. If a Harvard TLB implementation is used, this bit is a don't care for the ITLB. This bit is ignored on writes and always reads as zero for the ITLB.</p> <p>0 Do not allow data read accesses. Attempted data read accesses that match this entry generate an access error exception.</p> <p>1 Allow data read accesses.</p>															
3	W	<p>Write access enable. Indicates if data write accesses are allowed to this entry. If separate ITLB and DTLBs) are used, W is a don't care for the ITLB. W is ignored on writes and reads as zero for the ITLB.</p> <p>0 Do not allow data write accesses. Attempted data write accesses that match this entry generate an access error exception.</p> <p>1 Allow data write accesses.</p>															
2	X	<p>Execute access enable. Indicates if instruction fetches to this entry are allowed. If separate ITLB and DTLBs are is used, X is a don't care for the DTLB. X is ignored on writes and reads as zero for the DTLB.</p> <p>0 Do not allow instruction fetches. Attempted instruction fetches that match this entry cause an access error exception.</p> <p>1 Allow instruction fetch accesses.</p>															
1	LK	<p>Lock entry bit. Indicates if this entry is included in the replacement algorithm. TLB hits of locked entries do not update replacement algorithm information.</p> <p>0 Include this entry when determining the best entry for a TLB allocation.</p> <p>1 Do not allow this entry to be selected by the replacement algorithm.</p>															
0	—	Reserved, should be cleared. Writes are ignored and reads return zeros.															

## 10.5.4 MMU TLB

Each TLB entry consists of two 32-bit fields. The first is the TLB tag entry; the second is

the TLB data entry. TLB size and organization are implementation dependent. TLB entries can be read and written through MMU registers. TLB contents are unaffected by reset.

## 10.5.5 MMU Operation

The processor sends instruction fetch requests and data read/write requests to the K-Bus in the instruction and operand address generation cycles (IAG and OAG). The K-Bus controller and memories occupy the next two pipeline stages, instruction fetch cycles 1 and 2 (IC1 and IC2) and operand fetch cycles 1 and 2 (OC1 and OC2). For late writes, optional data pipeline stages are added to the K-Bus controller as well as any writable memories.

Table 10-11 shows the association between K-Bus memory pipeline stages and the processor's pipeline structures, shown in Figure 10-1.

**Table 10-11. Version 4 K-Bus Memory Pipelines**

K-Bus Memory Pipeline Stage	Instruction Fetch Pipeline	Operand Execution Pipeline
J stage	IAG	OAG
KC1 stage	IC1	OC1
KC2 stage	IC2	OC2
Operand execute stage	n/a	EX
Late-write stage	n/a	DA

Version 4 K-Buses use the same 2-cycle read pipeline developed for Version 3. Each K-Bus has 32-bit address and 32-bit read data paths. Version 4 uses synchronous memory elements for all memory control units. To support this, certain control information and all address bits are sent on the K-Buses at the end of the cycle before the initial bus access cycle (J cycle). The data K-Bus has an additional 32-bit write data path. For processor store operations, Version 4 ColdFire uses a late-write strategy, which can require 2 additional data K-Bus cycles. This yields the K-Bus pipeline behavior described in Table 10-12.

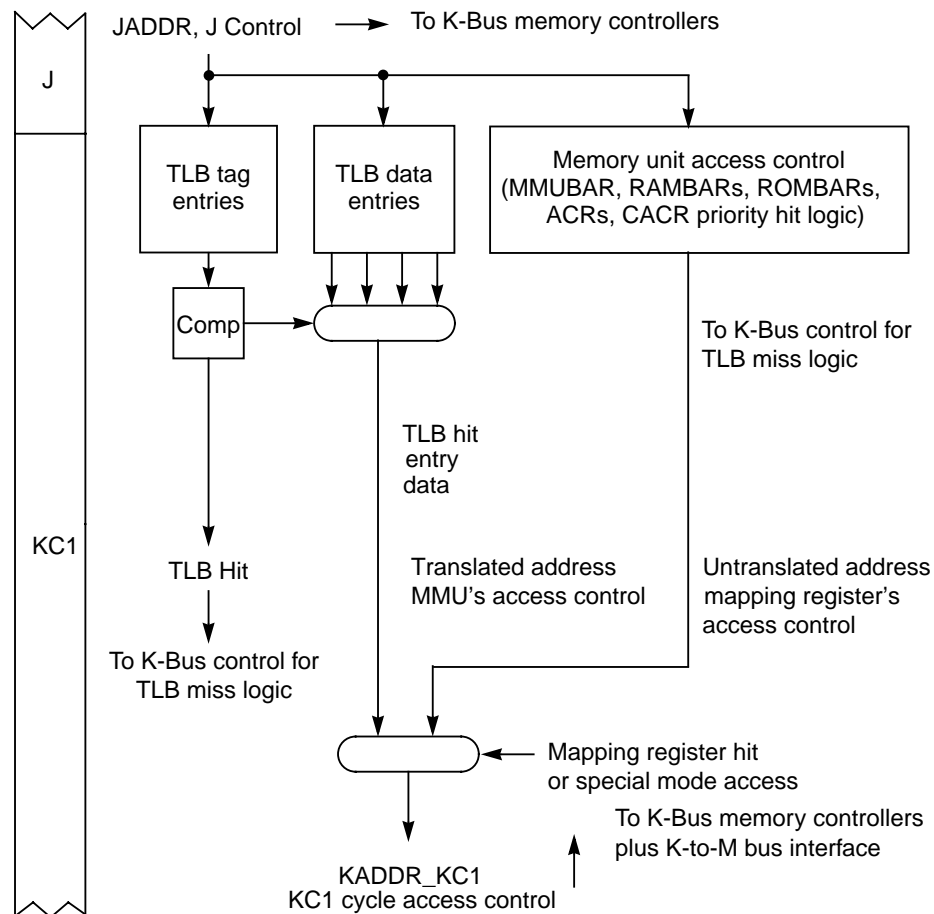
**Table 10-12. K-Bus Pipeline Cycles**

Cycle	Description
J	Control and partial address broadcast (to start synchronous memories)
KC1	Complete address and control broadcast plus MMU information. It is during this cycle that all memory element read operations are performed; that is, memory arrays are accessed.
KC2	Select appropriate memory as source, return data to processor, handle cache misses or hold K-Bus pipeline as needed.
EX	Optional write stage, pipeline address and control for store operations.
DA	Data available for stores from processor; memory element update occurs in the next cycle.

The K2M module contains two independent memory unit access controllers and two independent K-Bus controllers (I-Kc1 and O-Kc1). Each instruction and data K-Bus request is analyzed to see which, if any, K-Bus memory controller is referenced. This information, along with cache mode, store precision, and fault information, is sourced during KC1.

The optional MMU is referenced concurrently with the memory unit access controllers. It has two independent control sections to simultaneously process an instruction and data K-Bus request. Figure 10-1 shows how the MMU and memory unit access controllers fit in the K-Bus pipeline. As the diagram shows, core address and attributes are used to access the mapping registers and the MMU. By the middle of the KC1 cycle, the K-Bus physical memory address (KADDR\_KC1) is available along with its corresponding access control.

Figure 10-10 shows more details of the MMU structure. The TLB is accessed at the beginning of the KC1 pipeline stage so the resulting physical address can be sourced to the cache controllers to factor into the cache hit/miss determination. This is required because caches are virtually indexed but physically mapped.



**Figure 10-10. K-Bus Address and Attributes Generation**

## 10.6 MMU Implementation

The MMU implements a 64-entry full-associative Harvard TLB architecture with 32-entry ITLB and DTLB. This section provides more details of this specific TLB implementation. This section details the operation and looks at the size, frequency, miss rate, and miss recovery time of this specific TLB implementation.

## 10.6.1 TLB Address Fields

Because the TLB has a total of 64 entries (32 each for the ITLB and DTLB), a 6-bit address field is necessary. TLB addresses 0–31 reference the ITLB and TLB addresses 32–63 reference the DTLB.

In the MMUOR, bits 0 through 5 of the TLB allocation address (AA[5–0]), have this address format for CF4e. The remaining TLB allocation address bits (AA[15–6]) are ignored on updates and always read as zero.

When MMUAR is used for a TLB address, bits FA[5–0] also have this address format for CF4e. The remaining form address bits (FA[31–6]) are don't cares when this register is being used for a TLB address.

## 10.6.2 TLB Replacement Algorithm

The instruction and data TLBs provide low-latency access to recently used instruction and operand translation information. CF4e ITLBs and DTLBs are 32-entry fully associative caches. The 32 ITLB entries are searched on each instruction K-Bus reference; the 32 DTLB entries are searched on each operand K-Bus reference.

CF4e TLBs are software controlled. The TLB clear-all function clears valid bits on every TLB entry and resets the replacement logic. A new valid entry is loaded in the TLBs may be designated as locked and unavailable for allocation. TLB hits to locked entries do not update replacement algorithm information.

When a new TLB entry needs to be allocated, the user can specify the exact TLB entry to be updated (through MMUOR[ADR] and MMUAR) or let TLB hardware pick the entry to update based on the replacement algorithm. A pseudo-least-recently used (PLRU) algorithm picks the entry to be replaced on a TLB miss. The algorithm works as follows:

- If any element is empty (non-valid), use the lowest empty element as the allocate entry (that is, entry 0 before 1, 2, 3, and so on).
- If all entries are valid, use the entry indicated by the PLRU as the allocate entry.

The PLRU algorithm uses 31 most-recently used state bits per TLB to track the TLB hit history. Table 10-13 lists these state bits.

**Table 10-13. PLRU State Bits**

State Bits	Meaning
rdRecent31To16	A one indicates 31To16 is more recent than 15To00
rdRecent31To24	A one indicates 31To24 is more recent than 23To16
rdRecent15To08	A one indicates 15To08 is more recent than 07To00
rdRecent31To28	A one indicates 31To28 is more recent than 27To24
rdRecent23To20	A one indicates 23To20 is more recent than 19To16
rdRecent15To12	A one indicates 15To12 is more recent than 11To08

**Table 10-13. PLRU State Bits (Continued)**

State Bits	Meaning
rdRecent07To04	A one indicates 07To04 is more recent than 03To00
rdRecent31To30	A one indicates 31To30 is more recent than 29To28
rdRecent27To26	A one indicates 27To26 is more recent than 25To24
rdRecent23To22	A one indicates 23To22 is more recent than 21To20
rdRecent19To18	A one indicates 19To18 is more recent than 17To16
rdRecent15To14	A one indicates 15To14 is more recent than 13To12
rdRecent11To10	A one indicates 11To10 is more recent than 09To08
rdRecent07To06	A one indicates 07To06 is more recent than 05To04
rdRecent03To02	A one indicates 03To02 is more recent than 01To00
rdRecent31	A one indicates 31 is more recent than 30
rdRecent29	A one indicates 29 is more recent than 28
rdRecent27	A one indicates 27 is more recent than 26
rdRecent25	A one indicates 25 is more recent than 24
rdRecent23	A one indicates 23 is more recent than 22
rdRecent21	A one indicates 21 is more recent than 20
rdRecent19	A one indicates 19 is more recent than 18
rdRecent17	A one indicates 17 is more recent than 16
rdRecent15	A one indicates 15 is more recent than 14
rdRecent13	A one indicates 13 is more recent than 12
rdRecent11	A one indicates 11 is more recent than 10
rdRecent09	A one indicates 09 is more recent than 08
rdRecent07	A one indicates 07 is more recent than 06
rdRecent05	A one indicates 05 is more recent than 04
rdRecent03	A one indicates 03 is more recent than 02
rdRecent01	A one indicates 01 is more recent than 00

Binary state bits are updated on all TLB write (load) operations as well as normal ITLB and DTLB hits of non-locked entries. Also, if all entries in a binary state are locked, then that state is always set. That is, if entries 15, 14, 13, and 12 were locked, LRU state bit rdRecent15To14 is forced to one.

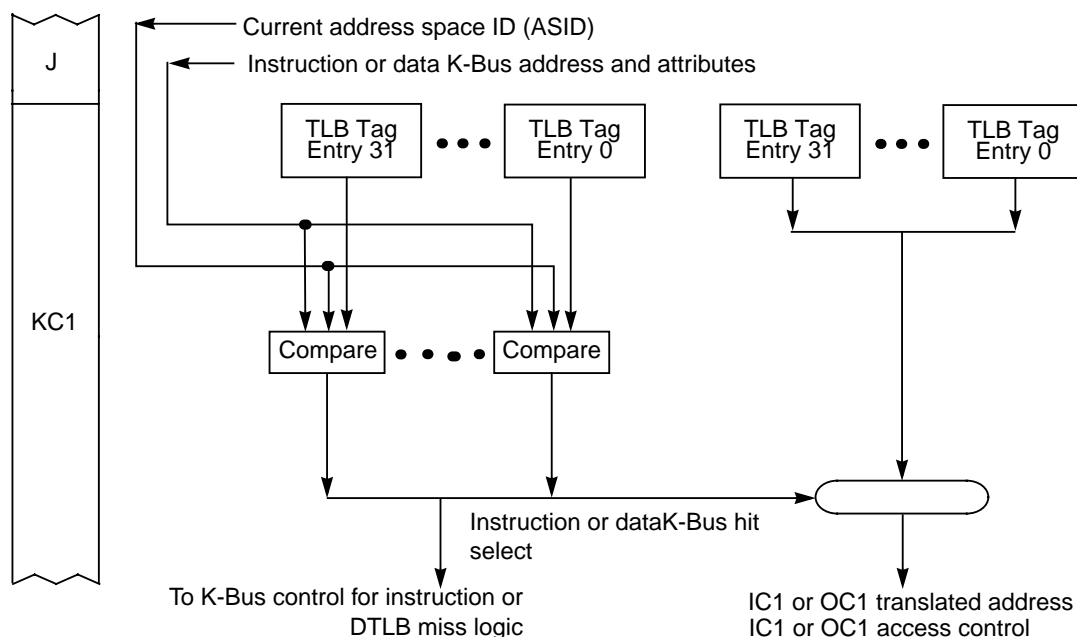
For a completely valid TLB, binary state information determines the LRU entry. The CF4e replacement algorithm is deterministic and, for the case of a full TLB, with no locked entries, and always touching new pages, the replacement entry repeats every 32 TLB loads.

### 10.6.3 TLB Locked Entries

Figure 10-11 is a ColdFire MMU Harvard TLB block diagram.

For TLB miss faults, the instruction restart model completely reexecutes an instruction on returning from the exception handler. An instruction can touch two instruction pages (a 32- or 48-bit instruction can straddle two pages) or four data pages (a memory-to-memory word or longword move where misaligned source and destination operands straddle two pages). Therefore, one instruction may take two ITLB misses and allocate two ITLB pages before completion. Likewise, one instruction may require four DTLB misses and allocate four DTLB pages. Because of this, a pool of unlocked TLB entries must be available if virtual memory is used.

The above examples show the fewest entries needed to guarantee an instruction can complete execution. For good MMU performance, more unlocked TLB entries should be available.



**Figure 10-11. Version 4 ColdFire MMU Harvard TLB**

## 10.7 MMU Instructions

The MOVE to USP and MOVE from USP instructions have been added for accessing the USP. Refer to the *PRM* for more information.

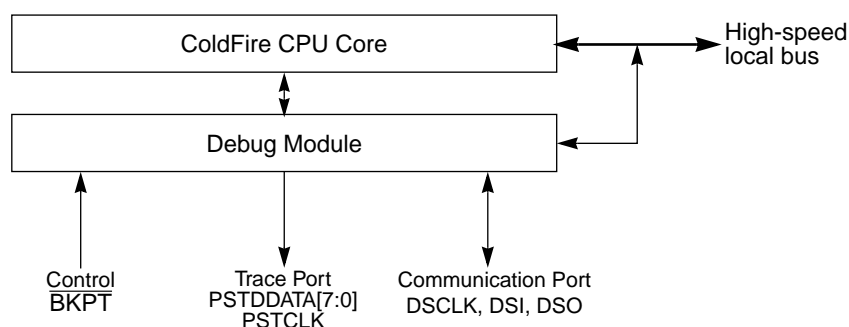
# Chapter 11

## Debug Support

This chapter describes the Revision D enhanced hardware debug support in the ColdFire Version 4. This revision of the ColdFire debug architecture encompasses earlier revisions. An expanded set of debug functionality is defined as Revision B (or Rev. B). The further enhanced debug architecture implemented in the Version 4 ColdFire is known as Revision C (or Rev. C).

### 11.1 Overview

The debug module interface is shown in Figure 11-1.



**Figure 11-1. Processor/Debug Module Interface**

Debug support is divided into three areas:

- Real-time trace support: The ability to determine the dynamic execution path through an application is fundamental for debugging. The ColdFire solution implements an 8-bit parallel output bus that reports processor execution status and data to an external BDM emulator system. See Section 11.3, “Real-Time Trace Support.”
- Background debug mode (BDM): Provides low-level debugging in the ColdFire processor complex. In BDM, the processor complex is halted and a variety of commands can be sent to the processor to access memory and registers. The external BDM emulator uses a three-pin, serial, full-duplex channel. See Section 11.5, “Background Debug Mode (BDM),” and Section 11.4, “Programming Model.”

- Real-time debug support: BDM requires the processor to be halted, which many real-time embedded applications cannot do. Debug interrupts let real-time systems execute a unique service routine that can quickly save key register and variable contents and return the system to normal operation without halting. External development systems can access saved data because the hardware supports concurrent operation of the processor and BDM-initiated commands. In addition, the option is provided to allow interrupts to occur. See Section 11.6, “Real-Time Debug Support.”

The Version 2 ColdFire core implemented the original debug architecture, now called Revision A. Based on feedback from customers and third-party developers, enhancements have been added to succeeding generations of ColdFire cores. For Revision A, CSR[HRL] is 0. See Section 11.4.5, “Configuration/Status Register (CSR).”

The Version 3 core implements Revision B of the debug architecture, offering more flexibility for configuring the hardware breakpoint trigger registers and removing the restrictions involving concurrent BDM processing while hardware breakpoint registers are active. For Revision B, CSR[HRL] is 1.

Revision C of the debug architecture more than doubles the on-chip breakpoint registers and provides an ability to interrupt debug service routines. For Revision C, CSR[HRL] is 2.

Differences between Revision B and C are summarized as follows:

- Debug Revision B has separate PST[3:0] and DDATA[3:0] signals.
- Debug Revision C adds breakpoint registers and supports normal interrupt request service during debug. It combines debug signals into PSTDDATA[7:0]

The addition of the memory management unit (MMU) to the baseline architecture requires corresponding enhancements to the ColdFire debug functionality, resulting in Revision D. For Revision D, the revision level bit, CSR[HRL], is 3.

With software support, the MMU can provide a demand-paged, virtual address environment. To support debugging in this virtual environment, the debug enhancements are primarily related to the expansion of the virtual address to include the 8-bit address space identifier (ASID). Conceptually, the virtual address is expanded to a 40-bit value: the 8-bit ASID plus the 32-bit address.

The expansion of the virtual address affects two major debug functions:

- The ASID is optionally included in the specification of the hardware breakpoint registers. As an example, the four PC breakpoint registers are each expanded by 8 bits, so that a specific ASID value may be programmed as part of the breakpoint instruction address. Likewise, each operand address/data breakpoint register is expanded to include an ASID value. Finally, new control registers define if and how the ASID is to be included in the breakpoint comparison trigger logic.



- The debug module implements the concept of ownership trace in which the ASID value may be optionally displayed as part of the real-time trace functionality. When enabled, real-time trace displays instruction addresses on every change-of-flow instruction that is not absolute or PC-relative. For Rev. D, this instruction address display optionally includes the contents of the ASID, thus providing the complete instruction virtual address on these instructions.

Additionally when a Sync\_PC serial BDM command is loaded from the external development system, the processor optionally displays the complete virtual instruction address, including the 8-bit ASID value.

In addition to these ASID-related changes, the new MMU control registers are accessible by using serial BDM commands. The same BDM access capabilities are also provided for the EMAC and FPU programming models.

Finally, a new serial BDM command is implemented to assist debugging when a software error generates an incorrect memory address that hangs the external bus. The new BDM command attempts to break this condition by forcing a bus termination.

## 11.2 Signal Descriptions

Table 11-1 describes debug module signals. All ColdFire debug signals are unidirectional and related to a rising edge of the processor core's clock signal. The standard 26-pin debug connector is shown in Section 11.9, "Motorola-Recommended BDM Pinout."

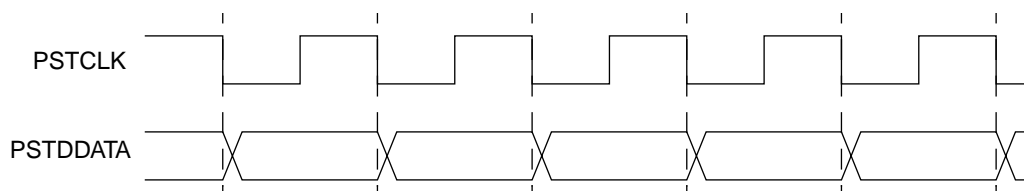
**Table 11-1. Debug Module Signals**

Signal	Description
Development Serial Clock (DSCLK)	Internally synchronized input. (The logic level on DSCLK is validated if it has the same value on two consecutive rising bus clock edges.) Clocks the serial communication port to the debug module during packet transfers. Maximum frequency is PSTCLK/5. At the synchronized rising edge of DSCLK, the data input on DSI is sampled and DSO changes state.
BDM Force Transfer Acknowledge (BDMFORCEACKB)	Helps break a hung external bus condition. An incorrect reference to a memory address that effectively hangs the external bus because no slave device responds. For such situations, the new serial BDM command can be sent into the debug module of the CF4e core. After decoding this command, the CF4e core asserts BDMFORCEACKB for an entire M-Bus clock period. This output can be factored into the external or M-Bus termination logic to unconditionally force a transfer acknowledge so that debug can continue without requiring a system reset. See Section 11.5.3.3.10, "Force Transfer Acknowledge (force_ta)."
Development Serial Input (DSI)	Internally synchronized input that provides data input for the serial communication port to the debug module, once the DSCLK has been seen as high (logic 1).
Development Serial Output (DSO)	Provides serial output communication for debug module responses. DSO is registered internally. The output is delayed from the validation of DSCLK high.
Breakpoint ( $\overline{\text{BKPT}}$ )	Input used to request a manual breakpoint. Assertion of $\overline{\text{BKPT}}$ puts the processor into a halted state after the current instruction completes. Halt status is reflected on processor status/debug data signals (PSTDDATA[7:0]) as the value 0xF. If CSR[BKD] is set (disabling normal $\overline{\text{BKPT}}$ functionality), asserting $\overline{\text{BKPT}}$ generates a debug interrupt exception in the processor.

**Table 11-1. Debug Module Signals (Continued)**

Signal	Description
Processor Status Clock (PSTCLK)	Half-speed version of the processor clock. Its rising edge appears in the center of the two-processor-cycle window of valid PSTDDATA output. See Figure 11-2. PSTCLK indicates when the development system should sample PSTDDATA values. If real-time trace is not used, setting CSR[PCD] keeps PSTCLK and PSTDDATA outputs from toggling without disabling triggers. Non-quiescent operation can be reenabled by clearing CSR[PCD], although the external development systems must resynchronize with the PSTDDATA output. PSTCLK starts clocking only when the first non-zero PST value (0xC, 0xD, or 0xF) occurs during system reset exception processing. Table 11-4 describes PST values.
Processor Status/Debug Data (PSTDDATA[7:0])	These outputs, which change on the negative edge of PSTCLK, indicate both processor status and captured address and data values and are discussed more thoroughly in Section 11.2.1, "Processor Status/Debug Data (PSTDDATA[7:0])."

Figure 11-2 shows PSTCLK timing with respect to PSTDDATA.

**Figure 11-2. PSTCLK Timing**

### 11.2.1 Processor Status/Debug Data (PSTDDATA[7:0])

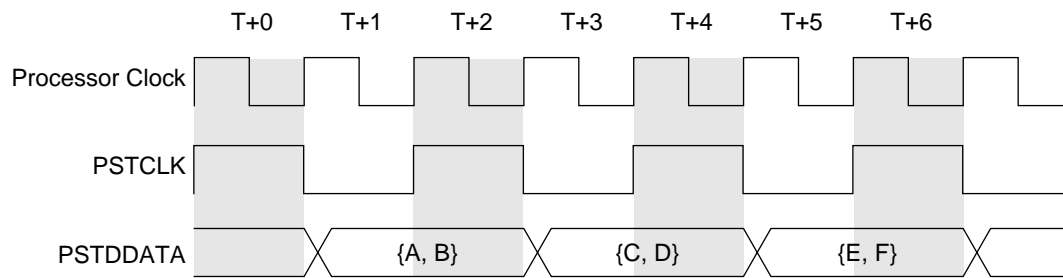
Processor status data outputs are used to indicate both processor status and captured address and data values. They operate at half the processor's frequency. Given that real-time trace information appears as a sequence of 4-bit data values, there are no alignment restrictions; that is, the processor status (PST) values and operands may appear on either nibble of PSTDDATA[7:0]. The upper nibble (PSTDDATA[7:4]) is the more significant and yields values first.

CSR controls capturing of data values to be presented on PSTDDATA. Executing the WDDATA instruction captures data that is displayed on PSTDDATA too. These signals are updated each processor cycle and display two values at a time for two processor clock cycles. Table 11-2 shows the PSTDDATA output for the processor's sequential execution of single-cycle instructions (A, B, C, D...). Cycle counts are shown relative to processor frequency. These outputs indicate the current processor pipeline status and are not related to the current bus transfer.

**Table 11-2. PSTDDATA: Sequential Execution of Single-Cycle Instructions**

Cycles	PSTDDATA[7:0]
T+0, T+1	{PST for A, PST for B}
T+2, T+3	{PST for C, PST for D}
T+4, T+5	{PST for E, PST for F}

The signal timing for the example in Table 11-2 is shown in Figure 11-3.



**Figure 11-3. PSTDDATA: Single-Cycle Instruction Timing**

Table 11-3 shows the case where a PSTDDATA module captures a memory operand on a simple load instruction: `mov.l <mem>, Rx`.

**Table 11-3. PSTDDATA: Data Operand Captured**

Cycle	PSTDDATA[7:0]
T	{PST for mov.l, PST marker for captured operand} = {0x1, 0xB}
T+1	{0x1, 0xB}
T+2	{Operand[3:0], Operand[7:4]}
T+3	{Operand[3:0], Operand[7:4]}
T+4	{Operand[11:8], Operand[15:12]}
T+5	{Operand[11:8], Operand[15:12]}
T+6	{Operand[19:16], Operand[23:20]}
T+7	{Operand[19:16], Operand[23:20]}
T+8	{Operand[27:24], Operand[31:28]}
T+9	{Operand[27:24], Operand[31:28]}
T+10	(PST for next instruction)
T+11	(PST for next instruction,...)

#### NOTE:

A PST marker and its data display are sent contiguously. Except for this transmission, the IDLE status (0x0) can appear anytime. Again, given that real-time trace information appears as a sequence of 4-bit values, there are no alignment restrictions. That is, PST values and operands may appear on either nibble of PSTDDATA.

## 11.3 Real-Time Trace Support

Real-time trace, which defines the dynamic execution path, is a fundamental debug function. The ColdFire solution is to include a parallel output port providing encoded processor status and data to an external development system. This 8-bit port is partitioned

into two consecutive 4-bit nibbles. Each nibble can either transmit information concerning the processor's execution status (PST) or debug data (DDATA). The processor status may not be related to the current bus transfer, due to the decoupling FIFOs.

External development systems can use PSTDDATA outputs with an external image of the program to completely track the dynamic execution path. This tracking is complicated by any change in flow, especially when branch target address calculation is based on the contents of a program-visible register (variant addressing). PSTDDATA outputs can be configured to display the target address of such instructions in sequential nibble increments across multiple processor clock cycles, as described in Section 11.3.1, "Begin Execution of Taken Branch (PST = 0x5)." Four 32-bit storage elements form a FIFO buffer connecting the processor's high-speed local bus to the external development system through PSTDDATA[7:0]. The buffer captures branch target addresses and certain data values for eventual display on the PSTDDATA port, two nibbles at a time starting with the least significant bit (lsb).

Execution speed is affected only when three storage elements contain valid data to be dumped to the PSTDDATA port. This occurs only when two values are captured simultaneously in a read-modify-write operation. The core stalls until two FIFO entries are available.

Table 11-4 shows the encoding of these signals.

**Table 11-4. Processor Status Encoding**

PST[3:0]		Definition
Hex	Binary	
0x0	0000	Continue execution. Many instructions execute in one processor cycle. If an instruction requires more clock cycles, subsequent clock cycles are indicated by driving PSTDDATA outputs with this encoding.
0x1	0001	Begin execution of one instruction. For most instructions, this encoding signals the first clock cycle of an instruction's execution. Certain change-of-flow opcodes, plus the PULSE and WDDATA instructions, generate different encodings.
0x2	0010	Begin execution of two instructions. For superscalar instruction dispatches, this encoding signals the first clock cycle of the simultaneous instructions' execution.
0x3	0011	Entry into user-mode. Signaled after execution of the instruction that caused the ColdFire processor to enter user mode. If the display of the ASID is enabled (CSR[3] = 1), the following occurs: <ul style="list-style-type: none"> <li>• The 8-bit ASID follows the instruction address; that is, the PSTDDATA sequence is {0x3, 0x5, marker, instruction address, 0x8, ASID}, where 0x8 is the ASID data marker.</li> <li>• Whenever the current ASID is loaded by the privileged MOVEC instruction, the ASID is displayed on PSTDDATA. The resulting PSTDDATA sequence for the MOVEC instruction is then {0x1, 0x8, ASID}, where the 0x8 is the data marker for the ASID.</li> </ul>
0x4	0100	Begin execution of PULSE and WDDATA instructions. PULSE defines logic analyzer triggers for debug or performance analysis. WDDATA lets the core write any operand (byte, word, or longword) directly to the PSTDDATA port, independent of debug module configuration. When WDDATA is executed, a value of 0x4 is signaled, followed by the appropriate marker, and then the data transfer on the PSTDDATA port. Transfer length depends on the WDDATA operand size.
0x5	0101	Begin execution of taken branch or SYNC_PC command. For some opcodes, a branch target address may be displayed on PSTDDATA depending on the CSR settings. CSR also controls the number of address bytes displayed, indicated by the PST marker value preceding the DDATA nibble that begins the data output. See Section 11.3.1, "Begin Execution of Taken Branch (PST = 0x5)." Also indicates that the SYNC_PC command has been issued.
0x6	0110	Begin execution of instruction plus a taken branch. The processor completes execution of a taken conditional branch instruction and simultaneously starts executing the target instruction. This is achieved through branch folding.
0x7	0111	Begin execution of return from exception (RTE) instruction.
0x8–0xB	1000–1011	Indicates the number of bytes to be displayed on the DDATA port on subsequent clock cycles. The value is driven onto the PSTDDATA port one cycle before the data is displayed. <ul style="list-style-type: none"> <li>0x8 Begin 1-byte transfer on PSTDDATA.</li> <li>0x9 Begin 2-byte transfer on PSTDDATA.</li> <li>0xA Begin 3-byte transfer on PSTDDATA.</li> <li>0xB Begin 4-byte transfer on PSTDDATA.</li> </ul>
0xC	1100	Normal exception processing. Exceptions that enter emulation mode (debug interrupt or optionally trace) generate a different encoding, as described below. Because the 0xC encoding defines a multiple-cycle mode, PSTDDATA outputs are driven with 0xC until exception processing completes.
0xD	1101	Emulator mode exception processing. Displayed during emulation mode (debug interrupt or optionally trace). Because this encoding defines a multiple-cycle mode, PSTDDATA outputs are driven with 0xD until exception processing completes.
0xE	1110	A breakpoint state change causes this encoding to assert for one cycle only followed by the trigger status value. If the processor stops waiting for an interrupt, the encoding is asserted for multiple cycles. See Section 11.3.2, "Processor Stopped or Breakpoint State Change (PST = 0xE)."
0xF	1111	Processor is halted. Because this encoding defines a multiple-cycle mode, the PSTDDATA outputs display 0xF until the processor is restarted or reset. (see Section 11.5.1, "CPU Halt")

### 11.3.1 Begin Execution of Taken Branch (PST = 0x5)

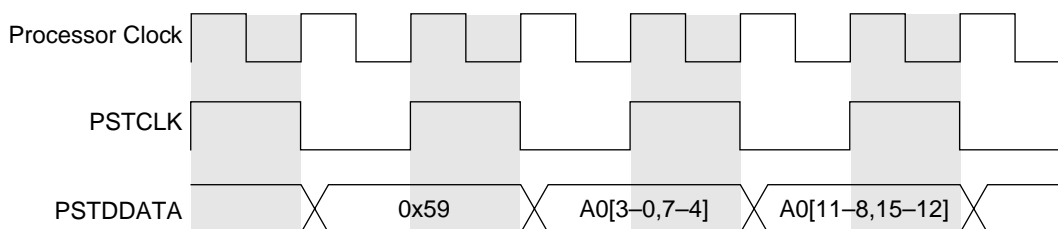
PST is 0x5 when a taken branch is executed. For some opcodes, a branch target address may be displayed on PSTDDATA depending on the CSR settings. CSR also controls the number of address bytes displayed, which is indicated by the PST marker value immediately preceding the PSTDDATA nibble that begins the data output.

Multiple byte DDATA values are displayed in least-to-most-significant order. The processor captures only those target addresses associated with taken branches which use a variant addressing mode, that is, RTE and RTS instructions, JMP and JSR instructions using address register indirect or indexed addressing modes, and all exception vectors.

The simplest example of a branch instruction using a variant address is the compiled code for a C language case statement. Typically, the evaluation of this statement uses the variable of an expression as an index into a table of offsets, where each offset points to a unique case within the structure. For such change-of-flow operations, the V4 microarchitecture uses the debug pins to output the following sequence of information on two successive processor clock cycles:

1. Use PSTDDATA (0x5) to identify that a taken branch is executed.
2. Optionally signal the target address to be displayed sequentially on the PSTDDATA pins. Encodings 0x9–0xB identify the number of bytes displayed.
3. The new target address is optionally available on subsequent cycles using the PSTDDATA port. The number of bytes of the target address displayed on this port is configurable (2, 3, or 4 bytes, where the encoding is 0x9, 0xA, and 0xB, respectively).

Another example of a variant branch instruction would be a JMP (A0) instruction. Figure 11-4 shows when the PSTDDATA outputs that indicate when a JMP (A0) executed, assuming the CSR was programmed to display the lower 2 bytes of an address.



**Figure 11-4. Example JMP Instruction Output on PSTDDATA**

PSTDDATA is driven two nibbles at a time with a 0x59; 0x5 indicates a taken branch and the marker value 0x9 indicates a 2-byte address. Thus, the subsequent 4 nibbles display the lower 2 bytes of address register A0 in least-to-most-significant nibble order. The PSTDDATA output after the JMP instruction continues with the next instruction.

### 11.3.2 Processor Stopped or Breakpoint State Change (PST = 0xE)

The 0xE encoding is generated either as a one- or multiple-cycle issue as follows:

- When the core is stopped by a STOP instruction, this encoding appears in multiple-cycle format. The ColdFire processor remains stopped until an interrupt occurs; thus, PSTDDATA outputs display 0xE until stopped mode is exited.
- When a breakpoint status change is to be output on PSTDDATA, 0xE is displayed for one cycle, followed immediately with the 4-bit value of the current trigger status, where the trigger status is left justified rather than in the CSR[BSTAT] description. Section 11.4.5, “Configuration/Status Register (CSR),” shows that status is right justified. That is, the displayed trigger status on PSTDDATA after a single 0xE is as follows:
  - 0x0 = no breakpoints enabled
  - 0x2 = waiting for level-1 breakpoint
  - 0x4 = level-1 breakpoint triggered
  - 0xA = waiting for level-2 breakpoint
  - 0xC = level-2 breakpoint triggered

Thus, 0xE can indicate multiple events, based on the next value, as Table 11-5 shows.

**Table 11-5. 0xE Status Posting**

PSTDDATA Stream Includes	Result
{0xE, 0x2}	Breakpoint state changed to waiting for level-1 trigger
{0xE, 0x4}	Breakpoint state changed to level-1 breakpoint triggered
{0xE, 0xA}	Breakpoint state changed to waiting for level-2 trigger
{0xE, 0xC}	Breakpoint state changed to level-2 breakpoint triggered
{0xE, 0xE}	Stopped mode.

### 11.3.3 Processor Halted (PST = 0xF)

PST is 0xF when the processor is halted (see Section 11.5.1, “CPU Halt”). Because this encoding defines a multiple-cycle mode, the PSTDDATA outputs display 0xF until the processor is restarted or reset. Therefore, PSTDDATA[7:0] continuously are 0xFF.

**NOTE:**

HALT can be distinguished from a data output 0xFF by counting 0xFF occurrences on PSTDDATA. Because data always follows a marker (0x8, 0x9, 0xA, or 0xB), the longest occurrence in PSTDDATA of 0xFF in a data output is four.

Two scenarios exist for data 0xFFFF\_FFFF:

## Programming Model

- The B marker occurs on the most-significant nibble of PSTDDATA with the data of 0xFF following:  
PSTDDATA[7:0]  
0xBF  
0xFF  
0xFF  
0xFF  
0xFF (X indicates that the next PST value is guaranteed to not be 0xF.)
- The B marker occurs on the least-significant nibble of PSTDDATA with the data of 0xFF following:  
PSTDDATA[7:0]  
0xYB  
0xFF  
0xFF  
0xFF  
0xFF  
0xXY (X indicates the PST value is guaranteed not to be 0xF, and Y signifies a PSTDDATA value that doesn't affect the 0xFF count.)

### NOTE:

As the result of the above, a count of at least nine or more sequential single 0xF values or five or more sequential 0xFF values indicates the HALT condition.

## 11.4 Programming Model

In addition to the existing BDM commands that provide access to the processor's registers and the memory subsystem, the debug module contains 19 registers to support the required functionality. These registers are also accessible from the processor's supervisor programming model by executing the WDEBBUG instruction (write only). Thus, the breakpoint hardware in the debug module can be read or written by the external development system using the debug serial interface or written by the operating system running on the processor core. Software is responsible for guaranteeing that accesses to these resources are serialized and logically consistent. Hardware provides a locking mechanism in the CSR to allow the external development system to disable any attempted writes by the processor to the breakpoint registers (setting CSR[IPW]). BDM commands must not be issued if the WDEBBUG instruction is used to access debug module registers or the resulting behavior is undefined.

These registers, shown in Figure 11-5, are treated as 32-bit quantities, regardless of the number of implemented bits.



31	15	7	0		AATR	Address attribute trigger register
31	15		0		ABLR	Address low breakpoint register
					ABHR	Address high breakpoint register
31	15	7	0		AATR1	Address 1 attribute trigger register
31	15		0		ABLR1	Address low breakpoint 1 register
					ABHR1	Address high breakpoint 1 register
31	15	7	0		BAAR	BDM address attributes register
31	15		0		CSR	Configuration/status register
31	15		0		DBR	Data breakpoint register
					DBMR	Data breakpoint mask register
31	15		0		DBR1	Data breakpoint 1 register
					DBMR1	Data breakpoint mask 1 register
31	15		0		PBR	PC breakpoint register
					PBR1	PC breakpoint 1 register
					PBR2	PC breakpoint 2 register
					PBR3	PC breakpoint 3 register
					PBMR	PC breakpoint mask register
31	15		0		TDR	Trigger definition register
31	15		0		XTDR	Extended trigger definition register

Note: Each debug register is accessed as a 32-bit register; shaded fields above are not used (don't care). All debug control registers are writable from the external development system or the CPU via the WDEBUD instruction. CSR is write-only from the programming model. It can be read from and written to through the BDM port. CSR is accessible in supervisor mode as debug control register 0x00 using the WDEBUD instruction and through the BDM port using the RDMREG and WDMREG commands.

**Figure 11-5. Debug Programming Model**

The registers in Table 11-7 are accessed through the BDM port by BDM commands, WDMREG and RDMREG, described in Section 11.5.3.3, “Command Set Descriptions.” These commands contain a 5-bit field, DRc, that specifies the register, as shown in Table 11-6.

**Table 11-6. BDM/Breakpoint Registers**

DRc[4–0]	Register Name	Abbreviation	Initial State	Page
0x00	Configuration/status register <sup>1</sup>	CSR	0x0020_0000	p. 11-17
0x01–0x05	Reserved	—	—	—
0x04	PC breakpoint ASID control	PBAC	—	p. 11-26
0x05	BDM address attribute register	BAAR	0x0000_0005	p. 11-16

**Table 11-6. BDM/Breakpoint Registers (Continued)**

DRc[4–0]	Register Name	Abbreviation	Initial State	Page
0x06	Address attribute trigger register	AATR	0x0000_0005	p. 11-13
0x07	Trigger definition register	TDR	0x0000_0000	p. 11-21
0x08	Program counter breakpoint register	PBR	—	p. 11-20
0x09	Program counter breakpoint mask register	PBMR	—	p. 11-20
0x0A–0x0B	Reserved	—	—	—
0x0C	Address breakpoint high register	ABHR	—	p. 11-15
0x0D	Address breakpoint low register	ABLR	—	p. 11-15
0x0E	Data breakpoint register	DBR	—	p. 11-19
0x0F	Data breakpoint mask register	DBMR	—	p. 11-19
0x10–0x13	Reserved	—	—	—
0x14	PC breakpoint ASID register	PBASID	—	p. 11-26
0x15	Reserved	—	—	—
0x16	Address attribute trigger register 1	AATR1	0x0000_0005	p. 11-13
0x17	Extended trigger definition register	XTDR	0x0000_0000	p. 11-24
0x18	Program counter breakpoint 1 register	PBR1	0x0000_0000	p. 11-20
0x19	Reserved	—	—	—
0x1A	Program counter breakpoint register 2	PBR2	0x0000_0000	p. 11-20
0x1B	Program counter breakpoint register 3	PBR3	0x0000_0000	p. 11-20
0x1C	Address high breakpoint register 1	ABHR1	—	p. 11-15
0x1D	Address low breakpoint register 1	ABLR1	—	p. 11-15
0x1E	Data breakpoint register 1	DBR1	—	p. 11-19
0x1F	Data breakpoint mask register 1	DBMR1	—	p. 11-19

<sup>1</sup> CSR is write-only from the programming model. It can be read or written through the BDM port using the RDMREG and WDMREG commands.

These registers are also accessible from the processor's supervisor programming model through the execution of the WDEBBUG instruction. Thus, the external development system and the operating system running on the processor core can access the breakpoint hardware. It is the responsibility of the software to guarantee that all accesses to these resources are serialized and logically consistent. The hardware provides a locking mechanism in the CSR to allow the external development system to disable any attempted writes by the processor to the breakpoint registers (setting IPW = 1). BDM commands must not be issued if the ColdFire processor is accessing debug module registers with the WDEBBUG instruction or the resulting behavior is undefined.

The ColdFire debug architecture supports a number of hardware breakpoint registers, that can be configured into single- or double-level triggers based on the PC or operand address ranges with an optional inclusion of specific data values. With the addition of the MMU

capabilities, the breakpoint specifications must be expanded to optionally include the address space identifier (ASID) in these user-programmable virtual address triggers.

The core includes four PC breakpoint triggers and two sets of operand address breakpoint triggers, each with two independent address registers (to allow specification of a range) and a data breakpoint with masking capabilities. Core breakpoint triggers are accessible through the serial BDM interface or written through the supervisor programming model using the WDEBUB instruction.

Two ASID-related registers (PBAC and PBASID) are added for the PC breakpoint qualification, and two existing registers (AATR and AATR1) are expanded for the address breakpoint qualification.

### 11.4.1 Revision A Shared Debug Resources

In the Revision A implementation of the debug module, certain hardware structures are shared between BDM and breakpoint functionality as shown in Table 11-7.

**Table 11-7. Rev. A Shared BDM/Breakpoint Hardware**

Register	BDM Function	Breakpoint Function
AATR	Bus attributes for all memory commands	Attributes for address breakpoint
ABHR	Address for all memory commands	Address for address breakpoint
DBR	Data for all BDM write commands	Data for data breakpoint

Thus, loading a register to perform a specific function that shares hardware resources is destructive to the shared function. For example, a BDM command to access memory overwrites an address breakpoint in ABHR. A BDM write command overwrites the data breakpoint in DBR.

Revision B added hardware registers to eliminate these shared functions. The BAAR is used to specify bus attributes for BDM memory commands and has the same format as the LSB of the AATR. Note that the registers containing the BDM memory address and the BDM data are not program visible.

### 11.4.2 Address Attribute Trigger Registers (AATR, AATR1)

The address attribute trigger registers (AATR and AATR1, Figure 11-6), define address attributes and a mask to be matched in the trigger. The register value is compared with address attribute signals from the processor's local high-speed bus, as defined by the setting of the trigger definition register (TDR) for AATR and the extended trigger definition register (XTDR) for AATR1.

This register is expanded to include an optional ASID specification and a control bit that enables the use of the ASID field.

	31					25		24				23				16
Field	—						ASIDCTRL		ATTRASID							
Reset	0000_0000_0000_0000															
R/W	Write only. AATR and AATR1 are accessible in supervisor mode as debug control register 0x06 and 0x16 respectively using the WDEBBUG instruction and through the BDM port using the WDMREG command.															

	15	14	13	12	11	10		8	7	6	5	4	3	2		0
Field	RM	SZM		TTM		TMM			R	SZ		TT		TM		
Reset	0000_0000_0000_0101															
R/W	Write only. AATR and AATR1 are accessible in supervisor mode as debug control register 0x06 and 0x16 respectively using the WDEBBUG instruction and through the BDM port using the WDMREG command.															
DRc[4–0]	0x06 (AATR); 0x16 (AATR1)															

**Figure 11-6. Address Attribute Trigger Registers (AATR, AATR1)**

Table 11-8 describes AATR and AATR1 fields.

**Table 11-8. AATR and AATR1 Field Descriptions**

Bits	Name	Description
31–25	—	Reserved, should be cleared.
24	ASIDCTRL	ABLR/ABHR/AATR address breakpoint ASID enable. Corresponds to the ASID control enable for the address breakpoint defined in ABLR, ABHR, and AATR. 0 Disable ASID qualifier (reset default) 1 Enable ASID qualifier
23–16	ATTRASID	ABLR/ABHR/AATR ASID. Corresponds to the ASID to be included in the address breakpoint specified by ABLR, ABHR, and AATR.
15	RM	Read/write mask. Setting RM masks R in address comparisons.
14–13	SZM	Size mask. Setting an SZM bit masks the corresponding SZ bit in address comparisons.
12–11	TTM	Transfer type mask. Setting a TTM bit masks the corresponding TT bit in address comparisons.
10–8	TMM	Transfer modifier mask. Setting a TMM bit masks the corresponding TM bit in address comparisons.
7	R	Read/write. R is compared with the $R/\overline{W}$ signal of the processor's local bus.
6–5	SZ	Size. Compared to the processor's local bus size signals. 00 Longword 01 Byte 10 Word 11 Reserved
4–3	TT	Transfer type. Compared with the local bus transfer type signals. 00 Normal processor access 01 Reserved 10 Emulator mode access 11 Acknowledge/CPU space access These bits also define the TT encoding for BDM memory commands. In this case, the 01 encoding indicates an external or DMA access (for backward compatibility). These bits affect the TM bits.

**Table 11-8. AATR and AATR1 Field Descriptions (Continued)**

Bits	Name	Description
2–0	TM	<p>Transfer modifier. Compared with the local bus transfer modifier signals, which give supplemental information for each transfer type.</p> <p><u>TT = 00 (normal mode):</u></p> <p>000 Data and instruction cache line push</p> <p>001 User data access</p> <p>010 User code access</p> <p>011 Instruction cache invalidate</p> <p>100 Data cache push/Instruction cache invalidate</p> <p>101 Supervisor data access</p> <p>110 Supervisor code access</p> <p>111 INTOUCH instruction access</p> <p><u>TT = 10 (emulator mode):</u></p> <p>0xx–100 Reserved</p> <p>101 Emulator mode data access</p> <p>110 Emulator mode code access</p> <p>111 Reserved</p> <p><u>TT = 11 (acknowledge/CPU space transfers):</u></p> <p>000 CPU space access</p> <p>001–111 Interrupt acknowledge levels 1–7</p> <p>These bits also define the TM encoding for BDM memory commands (for backward compatibility).</p>

### 11.4.3 Address Breakpoint Registers (ABLR/ABLR1, ABHR/ABHR1)

The address breakpoint low and high registers (ABLR, ABLR1, ABHR, and ABHR1, Figure 11-7), define regions in the processor's data address space that can be used as part of the trigger. These register values are compared with the address for each transfer on the processor's high-speed local bus. The trigger definition register (TDR) identifies the trigger as one of three cases:

- Identically the value in ABLR
- Inside the range bound by ABLR and ABHR inclusive
- Outside that same range

XTDR determines the same for ABLR1 and ABHR1.

	31	0
Field	Address	
Reset	—	
R/W	Write only. ABHR and ABHR1 are accessible in supervisor mode as debug control registers 0x0C and 0x1C, using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands. ABLR and ABLR1 are accessible in supervisor mode as debug control register 0x0D and 0x1D, using the WDEBUG instruction and via the BDM port using the WDMREG command.	
DRc[4–0]	0x0D (ABLR); 0x1D (ABLR1); 0x0C (ABHR); 0x1C (ABHR1)	

**Figure 11-7. Address Breakpoint Registers (ABLR, ABHR, ABLR1, ABHR1)**

Table 11-9 describes ABLR and ABLR1 fields.

**Table 11-9. ABLR and ABLR1 Field Description**

Bits	Name	Description
31–0	Address	Low address. Holds the 32-bit address marking the lower bound of the address breakpoint range. Breakpoints for specific addresses are programmed into ABLR or ABLR1.

Table 11-10 describes ABHR and ABHR1 fields.

**Table 11-10. ABHR and ABHR1 Field Description**

Bits	Name	Description
31–0	Address	High address. Holds the 32-bit address marking the upper bound of the address breakpoint range.

### 11.4.4 BDM Address Attribute Register (BAAR)

The BAAR defines the address space for memory-referencing BDM commands. To maintain compatibility with Revision A, BAAR is loaded with any data written to the LSB of AATR. See Figure 11-8. The reset value of 0x5 sets supervisor data as the default address space.

	7	6	5	4	3	2	0
Field	R	SZ	TT	TM			
Reset	0000_0101						
R/W	Write only. BAAR[R,SZ] are loaded directly from the BDM command; BAAR[TT,TM] can be programmed as debug control register 0x05 from the external development system. For compatibility with Rev. A, BAAR is loaded each time AATR is written.						
DRc[4–0]	0x05						

**Figure 11-8. BDM Address Attribute Register (BAAR)**

Table 11-11 describes BAAR fields.

**Table 11-11. BAAR Field Descriptions**

Bits	Name	Description
7	R	Read/write 0 Write 1 Read
6–5	SZ	Size 00 Longword 01 Byte 10 Word 11 Reserved
4–3	TT	Transfer type. See the TT definition in Table 11-8.
2–0	TM	Transfer modifier. See the TM definition in Table 11-8.

## 11.4.5 Configuration/Status Register (CSR)

The configuration/status register (CSR) defines the debug configuration for the processor and memory subsystem and contains status information from the breakpoint logic. CSR is write-only from the programming model. CSR is accessible in supervisor mode as debug control register 0x00 using the WDEBUI instruction and through the BDM port using the RDMREG and WDMREG commands. It can be read from and written to through the BDM port.

	31	28	27	26	25	24	23	20	19	17	16				
Field	BSTAT			FOF	TRG	HALT	BKPT	HRL		—	BKD	PCD	IPW		
Reset	0000			0	0	0	0	0010		0	0	0	0		
R/W <sup>1</sup>	R			R	R	R	R	R		—	R/W	R/W	R/W		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
Field	MAP	TRC	EMU	DDC		UHE	BTB		—	NPL	—	SSM	OTE	—	
Reset	0	0	0	00		0	00		0	0	0	0	0	—	
R/W	R/W	R/W	R/W	R/W		R/W	R/W		R	R/W	—	R/W	—		
DRc[4–0]	0x00														

**Figure 11-9. Configuration/Status Register (CSR)**

Table 11-12 describes CSR fields.

**Table 11-12. CSR Field Descriptions**

Bits	Name	Description
31–28	BSTAT	Breakpoint status. Provides read-only status information concerning hardware breakpoints. Also output on PSTDDATA when it is not displaying PST or other processor data. BSTAT is cleared by a TDR or XTDR write or by a CSR read when either a level-2 breakpoint is triggered or a level-1 breakpoint is triggered and the level-2 breakpoint is disabled. 0000 No breakpoints enabled 0001 Waiting for level-1 breakpoint 0010 Level-1 breakpoint triggered 0101 Waiting for level-2 breakpoint 0110 Level-2 breakpoint triggered
27	FOF	Fault-on-fault. If FOF is set, a catastrophic halt occurred and forced entry into BDM.
26	TRG	Hardware breakpoint trigger. If TRG is set, a hardware breakpoint halted the processor core and forced entry into BDM. Reset and the debug GO command clear TRG.
25	HALT	Processor halt. If HALT is set, the processor executed a HALT and forced entry into BDM. Reset and the debug GO command clear HALT.
24	BKPT	Breakpoint assert. If BKPT is set, $\overline{\text{BKPT}}$ is asserted, forcing the processor into BDM. Reset and the debug GO command clear BKPT.
23–20	HRL	Hardware revision level. Indicates the level of debug module functionality. An emulator could use this information to identify the level of functionality supported. 0000 Initial debug functionality (Revision A) 0001 Revision B 0010 Revision C 0011 Revision D

Table 11-12. CSR Field Descriptions (Continued)

Bits	Name	Description
19	—	Reserved, should be cleared.
18	BKD	Breakpoint disable. Used to disable the normal $\overline{\text{BKPT}}$ input functionality and to allow the assertion of $\overline{\text{BKPT}}$ to generate a debug interrupt. 0 Normal operation 1 $\overline{\text{BKPT}}$ is edge-sensitive: a high-to-low edge on $\overline{\text{BKPT}}$ signals a debug interrupt to the processor. The processor makes this interrupt request pending until the next sample point, when the exception is initiated. In the ColdFire architecture, the interrupt sample point occurs once per instruction. There is no support for nesting debug interrupts.
17	PCD	PSTCLK disable. Setting PCD disables generation of PSTCLK and PSTDDATA outputs and forces them to remain quiescent.
16	IPW	Inhibit processor writes. Setting IPW inhibits processor-initiated writes to the debug module's programming model registers. IPW can be modified only by commands from the external development system.
15	MAP	Force processor references in emulator mode. 0 All emulator-mode references are mapped into supervisor code and data spaces. 1 The processor maps all references while in emulator mode to a special address space, TT = 10, TM = 101 or 110. The internal SRAM and caches are disabled.
14	TRC	Force emulation mode on trace exception. If TRC = 1, the processor enters emulator mode when a trace exception occurs. If TRC=0, the processor enters supervisor mode.
13	EMU	Force emulation mode. If EMU = 1, the processor begins executing in emulator mode. See Section 11.6.1.1, "Emulator Mode."
12–11	DDC	Debug data control. Controls operand data capture for PSTDDATA, which displays the number of bytes defined by the operand reference size before the actual data; byte displays 8 bits, word displays 16 bits, and long displays 32 bits (one nibble at a time across multiple clock cycles). See Table 11-4. 00 No operand data is displayed. 01 Capture all M-Bus write data. 10 Capture all M-Bus read data. 11 Capture all M-Bus read and write data.
10	UHE	User halt enable. Selects the CPU privilege level required to execute the HALT instruction. 0 HALT is a supervisor-only instruction. 1 HALT is a supervisor/user instruction.
9–8	BTB	Branch target bytes. Defines the number of bytes of branch target address PSTDDATA displays. 00 0 bytes 01 Lower 2 bytes of the target address 10 Lower 3 bytes of the target address 11 Entire 4-byte target address See Section 11.3.1, "Begin Execution of Taken Branch (PST = 0x5)."
7	—	Reserved, should be cleared.



**Table 11-12. CSR Field Descriptions (Continued)**

Bits	Name	Description
6	NPL	Non-pipelined mode. Determines whether the core operates in pipelined or mode. 0 Pipelined mode 1 Non-pipelined mode. The processor effectively executes one instruction at a time with no overlap. This adds at least 5 cycles to the execution time of each instruction. Superscalar instruction dispatch is disabled when operating in this mode. Given an average execution latency of 1.6, throughput in non-pipeline mode would be 6.6, approximately 25% or less of pipelined performance. Regardless of the NPL state, a triggered PC breakpoint is always reported before the triggering instruction executes. In normal pipeline operation, the occurrence of an address or data breakpoint trigger is imprecise. In non-pipeline mode, triggers are always reported before the next instruction begins execution and trigger reporting can be considered precise. An address or data breakpoint should always occur before the next instruction begins execution. Therefore, the occurrence of the address/data breakpoints should be guaranteed.
5	—	Reserved, should be cleared.
4	SSM	Single-step mode. Setting SSM puts the processor in single-step mode. 0 Normal mode. 1 Single-step mode. The processor halts after execution of each instruction. While halted, any BDM command can be executed. On receipt of the GO command, the processor executes the next instruction and halts again. This process continues until SSM is cleared.
3	OTE	Ownership-trace enable. 1 The display of the ASID on the PSTDDATA outputs by entering in user mode, by loading the ASID by a MOVEC, or by executing a BDM SYNC_PC command.
3–0	—	Reserved, should be cleared.

### 11.4.6 Data Breakpoint/Mask Registers (DBR/DBR1, DBMR/DBMR1)

The data breakpoint registers (DBR/DBR1, Figure 11-10), specify data patterns used as part of the trigger into debug mode. DBR $n$  bits are masked by setting corresponding DBMR bits, as defined in TDR.

	31	0
Field	Data (DBR/DBR1); Mask (DBMR/DBMR1)	
Reset	Uninitialized	
R/W	DBR and DBR1 are accessible in supervisor mode as debug control register 0x0E and 0x1E, using the WDEBBUG instruction and through the BDM port using the RDMREG and WDMREG commands. DBMR and DBMR1 are accessible in supervisor mode as debug control register 0x0F and 0x1F, using the WDEBBUG instruction and via the BDM port using the WDMREG command.	
DRc[4–0]	0x0E (DBR), 0x1E (DBR1); 0x0F (DBMR), 0x1F (DBMR1)	

**Figure 11-10. Data Breakpoint/Mask Registers (DBR/DBR1 and DBMR/DBMR1)**

Table 11-13 describes DBR $n$  fields.

**Table 11-13. DBR $n$  Field Descriptions**

Bits	Name	Description
31–0	Data	Data breakpoint value. Contains the value to be compared with the data value from the processor's local bus as a breakpoint trigger.

Table 11-14 describes DBMR $n$  fields.

**Table 11-14. DBMR $n$  Field Descriptions**

Bits	Name	Description
31–0	Mask	Data breakpoint mask. The 32-bit mask for the data breakpoint trigger. Clearing a DBR $n$ bit allows the corresponding DBR $n$ bit to be compared to the appropriate bit of the processor's local data bus. Setting a DBMR $n$ bit causes that bit to be ignored.

DBRs support both aligned and misaligned references. Table 11-15 shows relationships between processor address, access size, and location within the 32-bit data bus.

**Table 11-15. Access Size and Operand Data Location**

A[1:0]	Access Size	Operand Location
00	Byte	D[31:24]
01	Byte	D[23:16]
10	Byte	D[15:8]
11	Byte	D[7:0]
0x	Word	D[31:16]
1x	Word	D[15:0]
xx	Longword	D[31:0]

### 11.4.7 Program Counter Breakpoint/Mask Registers (PBR, PBR1, PBR2, PBR3, PBMR)

Each PC breakpoint register (PBR, PBR1, PBR2, PBR3) defines an instruction address for use as part of the trigger. These registers' contents are compared with the processor's program counter register when the appropriate valid bit is set and TDR or XTDR are configured appropriately. PBR bits are masked by setting corresponding PBMR bits. Results are compared with the processor's program counter register, as defined in TDR or XTDR. PBR1–PBR3 are not masked. Figure 11-11 shows the PC breakpoint register.

	31	1	0
Field	Program Counter		V <sup>1</sup>
Reset	—		0
R/W	Write. PC breakpoint registers are accessible in supervisor mode using the WDEBBUG instruction and through the BDM port using the RDMREG and WDMREG commands using values shown in Section 11.5.3.3, “Command Set Descriptions.”		
DRc[4–0]	0x08 (PBR); 0x18 (PBR1); 0x1A (PBR2); 0x1B (PBR3)		

<sup>1</sup> PBR does not have a valid bit. PBR[0] is read as 0 and should be cleared.

**Figure 11-11. Program Counter Breakpoint Registers (PBR, PBR1, PBR2, PBR3)**

Table 11-16 describes PBR, PBR1, PBR2, and PBR3 fields.

**Table 11-16. PBR, PBR1, PBR2, PBR3 Field Descriptions**

Bits	Name	Description
31–1	Address	PC breakpoint address. The 31-bit address to be compared with the PC as a breakpoint trigger. PBR does not have a valid bit.
0	V	Valid. Breakpoint registers are compared with the processor’s program counter register when the appropriate valid bit is set and TDR or XTDR are configured appropriately. This bit is not implemented on PBR.

Figure 11-11 shows PBMR.

	31	0
Field	Mask	
Reset	—	
R/W	Write. PBMR is accessible in supervisor mode as debug control register 0x09 using the WDEBBUG instruction and via the BDM port using the WDMREG command.	
DRc[4–0]	0x09	

**Figure 11-12. Program Counter Breakpoint Mask Register (PBMR)**

Table 11-17 describes PBMR fields.

**Table 11-17. PBMR Field Descriptions**

Bits	Name	Description
31–0	Mask	PC breakpoint mask. A zero in a bit position causes the corresponding PBR bit to be compared to the appropriate PC bit. Set PBMR bits cause PBR bits to be ignored.

## 11.4.8 Trigger Definition Register (TDR)

The TDR, shown in Table 11-13, configures the operation of the hardware breakpoint logic that corresponds with the ABHR/ABLR/AATR, PBR/PBR1/PBR2/PBR3/PBMR, and DBR/DBMR registers within the debug module. In conjunction with the XTDR and its associated debug registers, TDR controls the actions taken under the defined

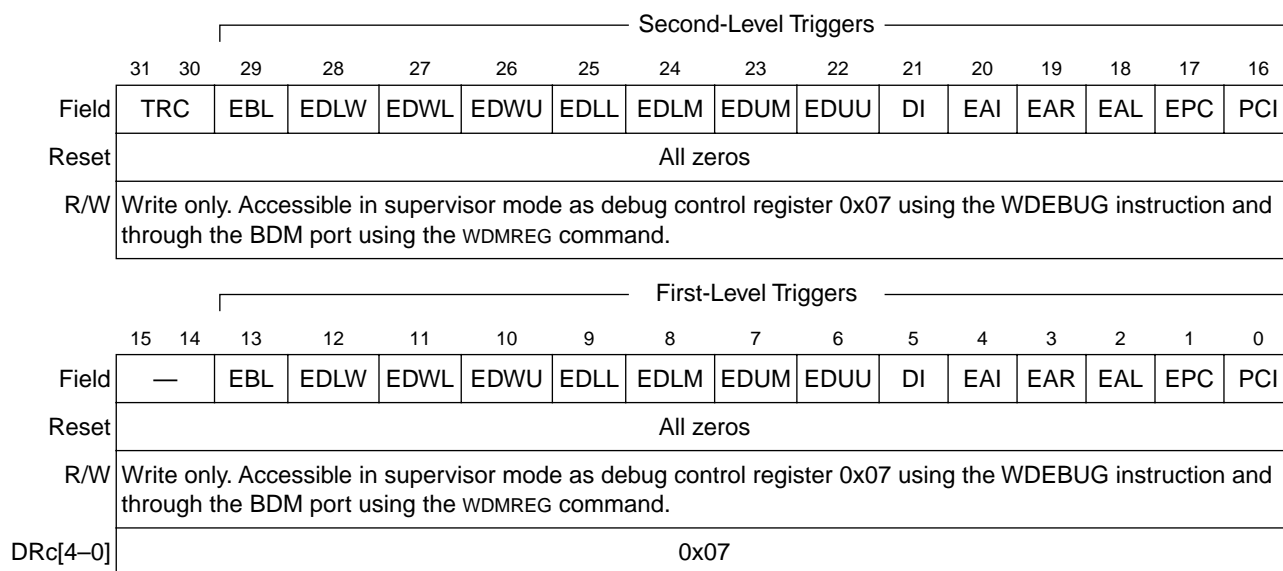
## Programming Model

conditions. Breakpoint logic may be configured as one- or two-level triggers. TDR[31–16] or XTDR[31–16] define second-level triggers and bits 15–0 define first-level triggers.

### NOTE:

The debug module has no hardware interlocks, so to prevent spurious breakpoint triggers while the breakpoint registers are being loaded, disable TDR and XTDR (by clearing TDR[29,13] and XTDR[29,13]) before defining triggers.

A write to TDR clears the CSR trigger status bits, CSR[BSTAT].



**Figure 11-13. Trigger Definition Register (TDR)**

Table 11-18 describes TDR fields.

**Table 11-18. TDR Field Descriptions**

Bits	Name	Description
31–30	TRC	Trigger response control. Determines how the processor responds to a completed trigger condition. The trigger response is always displayed on PSTDDATA. 00 Display on PSTDDATA only 01 Processor halt 10 Debug interrupt 11 Reserved
15–14	—	Reserved, should be cleared.
29/13	EBL	Enable breakpoint. Global enable for the breakpoint trigger. Setting TDR[EBL] or XTDR[EBL] enables a breakpoint trigger. If both TDL[EBL] and XTDL[EBL] are cleared, all breakpoints are disabled.

**Table 11-18. TDR Field Descriptions (Continued)**

Bits	Name	Description	
28–22 12–6	EDx	Enable data. Setting an EDx bit enables the corresponding data breakpoint condition based on the size and placement on the processor's local data bus. Clearing all EDx bits disables data breakpoints.	
28/12		EDLW	Data longword. Entire processor's local data bus.
27/11		EDWL	Lower data word.
26/10		EDWU	Upper data word.
25/9		EDLL	Lower lower data byte. Low-order byte of the low-order word.
24/8		EDLM	Lower middle data byte. High-order byte of the low-order word.
23/7		EDUM	Upper middle data byte. Low-order byte of the high-order word.
22/6		EDUU	Upper upper data byte. High-order byte of the high-order word.
21/5	DI	Data breakpoint invert. Provides a way to invert the logical sense of all the data breakpoint comparators. This can develop a trigger based on the occurrence of a data value other than the DBR contents.	
20–18/ 4–2	EAx	Enable address bits. Setting an EA bit enables the corresponding address breakpoint. Clearing all three bits disables the breakpoint.	
20/4		EAI	Enable address breakpoint inverted. Breakpoint is based outside the range between ABLR and ABHR. Trigger if address > ABHR or if address < ABLR.
19/3		EAR	Enable address breakpoint range. The breakpoint is based on the inclusive range defined by ABLR and ABHR. Trigger if address ≥ ABHR or if address ≤ ABLR.
18/2		EAL	Enable address breakpoint low. The breakpoint is based on the address in the ABLR. Trigger address = ABLR
17/1	EPC	Enable PC breakpoint. If set, this bit enables the PC breakpoint.	
16/0	PCI	Breakpoint invert. If set, this bit allows execution outside a given region as defined by PBR/PBR1/PBR2/PBR3 and PBMR to enable a trigger. If cleared, the PC breakpoint is defined within the region defined by PBR/PBR1/PBR2/PBR3 and PBMR.	

### 11.4.9 Extended Trigger Definition Register (XTDR)

The XTDR configures the operation of the hardware breakpoint logic that corresponds with the ABHR1/ABLR1/AATR1 and DBR1/DBMR1 registers within the debug module and, in conjunction with the TDR and its associated debug registers, controls the actions taken under the defined conditions. The breakpoint logic may be configured as a one- or two-level trigger, where TDR[31–16] or XTDR[31–16] define the second-level trigger and bits 15–0 define the first-level trigger. The XTDR is accessible in supervisor mode as debug control register 0x17 using the WDEBUG instruction and via the BDM port using the WDMREG command.

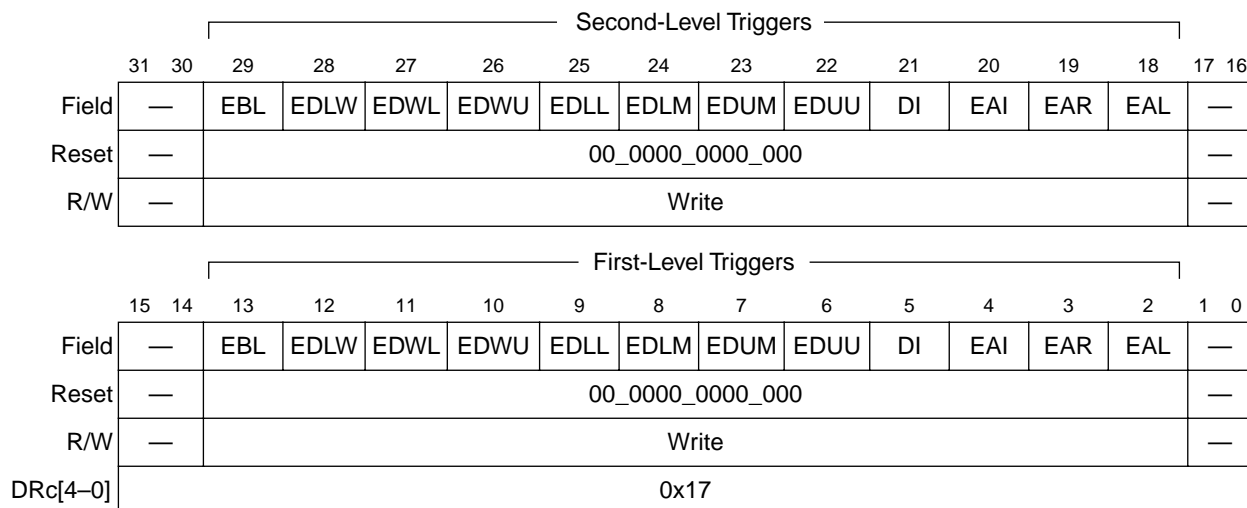
#### NOTE:

The debug module has no hardware interlocks, so to prevent spurious breakpoint triggers while the breakpoint registers are being loaded, disable TDR and XTDR (by clearing TDR[29,13] and XTDR[29,13]) before defining triggers.

## Programming Model

A write to the XTDR clears the trigger status bits, CSR[BSTAT].

Section 11.4.9.1, “Resulting Set of Possible Trigger Combinations,” describes how to handle multiple breakpoint conditions.



**Figure 11-14. Extended Trigger Definition Register (XTDR)**

Table 11-19 describes XTDR fields.

**Table 11-19. XTDR Field Descriptions**

Bits	Name	Description
29/13	EBL	Enable breakpoint level. If set, EBL is the global enable for the breakpoint trigger; that is, if TDR[EBL] or XTDR[EBL] is set, a breakpoint trigger is enabled. Clearing both disables all breakpoints.
28–22 12–6	EDx	Setting an EDx bit enables the corresponding data breakpoint condition based on the size and placement on the processor's local data bus. Clearing all EDx bits disables data breakpoints.
28/12		EDLW Data longword. Entire processor's local data bus.
27/11		EDWL Lower data word.
26/10		EDWU Upper data word.
25/9		EDLL Lower lower data byte. Low-order byte of the low-order word.
24/8		EDLM Lower middle data byte. High-order byte of the low-order word.
23/7		EDUM Upper middle data byte. Low-order byte of the high-order word.
22/6		EDUU Upper upper data byte. High-order byte of the high-order word.
21/5	DI	Data breakpoint invert. Provides a way to invert the logical sense of all the data breakpoint comparators. This can develop a trigger based on the occurrence of a data value other than the DBR1 contents.

**Table 11-19. XTDR Field Descriptions (Continued)**

Bits	Name	Description
20–18/ 4–2	EAx	Enable address bits. Setting an EAx bit enables the corresponding address breakpoint. If all three bits are cleared, this breakpoint is disabled.
20/4		EAI    Enable address breakpoint inverted. Breakpoint is based outside the range between ABLR1 and ABHR1. Trigger if address > ABHR or if address < ABLR.
19/3		EAR    Enable address breakpoint range. Breakpoint is based on the range defined between ABLR1 and ABHR1. Trigger if address ≥ ABHR or if address ≤ ABLR.
18/2		EAL    Enable address breakpoint low. The breakpoint is based on the address in ABLR1. Trigger if address = ABLR.
17–16, 1–0	—	Reserved, should be cleared.

### 11.4.9.1 Resulting Set of Possible Trigger Combinations

The resulting set of possible breakpoint trigger combinations consist of the following options where || denotes logical OR, && denotes logical AND, and { } denotes an optional additional trigger term:

One-level triggers of the form:

```

if      (PC_breakpoint)
if      (PC_breakpoint | Address_breakpoint{&& Data_breakpoint})
if      (PC_breakpoint | Address_breakpoint{&& Data_breakpoint}
        || Address1_breakpoint{&& Data1_breakpoint})

if      (Address_breakpoint {&& Data_breakpoint})
if      ((Address_breakpoint {&& Data_breakpoint}
        || Address1_breakpoint{&& Data1_breakpoint}))

if      (Address1_breakpoint {&& Data1_breakpoint})

```

Two-level triggers of the form:

```

if      (PC_breakpoint)
  then if      (Address_breakpoint{&& Data_breakpoint})

if      (PC_breakpoint)
  then if      (Address_breakpoint{&& Data_breakpoint}
        || Address1_breakpoint{&& Data1_breakpoint})

if      (PC_breakpoint)
  then if      (Address1_breakpoint{&& Data1_breakpoint})

if      (Address_breakpoint {&& Data_breakpoint})
  then if      (Address1_breakpoint{&& Data1_breakpoint})

if      (Address1_breakpoint {&& Data1_breakpoint})
  then if      (Address_breakpoint{&& Data_breakpoint})

if      (Address_breakpoint {&& Data_breakpoint})
  then if      (PC_breakpoint)

if      (Address1_breakpoint {&& Data1_breakpoint})
  then if      (PC_breakpoint)

if      (Address_breakpoint {&& Data_breakpoint})
  then if      (PC_breakpoint)

```

## Programming Model

```
        ||      Address1_breakpoint{&& Data1_breakpoint})  
if      (Address1_breakpoint {&& Data1_breakpoint})  
  then if (PC_breakpoint  
        ||      Address_breakpoint{&& Data_breakpoint})
```

In this example, PC\_breakpoint is the logical summation of the PBR/PBMR, PBR1, PBR2, and PBR3 breakpoint registers; Address\_breakpoint is a function of ABHR, ABLR, and AATR; Data\_breakpoint is a function of DBR and DBMR; Address1\_breakpoint is a function of ABHR1, ABLR1, and AATR1; and Data1\_breakpoint is a function of DBR1 and DBMR1. In all cases, the data breakpoints can be included with an address breakpoint to further qualify a trigger event as an option.

### 11.4.10 PC Breakpoint ASID Control Register (PBAC)

The PBAC configures the breakpoint qualification for each PC breakpoint register (PBR, PBR1, PBR2, and PBR3). Four bits are dedicated for each breakpoint register and specify how the ASID is used in PC breakpoint qualification.

	15	12	11	8	7	4	3	0	
Field	PBR3AC				PBR2AC		PBR1AC		PBRAC
Reset	All zeros								
R/W	W								
DRc[4–0]	0x0A								

**Figure 11-15. PC Breakpoint ASID Control Register (PBAC)**

PBR3AC, PBR2AC, PBR1AC, and PBRAC apply to PBR3, PBR2, PBR1, and PBR, respectively, and are functionally identical. They enable or disable ASID, supervisor mode, and user mode breakpoint qualification. Reset clears these fields, disabling qualifications and defaulting to the Revision C debug module functionality.

**Table 11-20. PBAC Field Descriptions**

Bits	Name	Description
15–12	PBR3AC	PBR <sub>n</sub> ASID control. Corresponds to the ASID control associated with PBR <sub>n</sub> . Determines whether the ASID is included in the PC breakpoint comparison and whether the operating mode (supervisor or user) is included in the comparison logic. x00x No ASID qualification; no mode qualification x010 No ASID qualification; user mode qualification enabled x011 No ASID qualification; supervisor mode qualification enabled x10x ASID qualification enabled; no mode qualification x110 ASID qualification enabled; user mode qualification enabled x111 ASID qualification enabled; supervisor mode qualification enabled
11–8	PBR2AC	
7–4	PBR1AC	
3–0	PBRAC	

### 11.4.11 PC Breakpoint ASID Register (PBASID)

Each PC breakpoint register (PBR, PBR1, PBR2, or PBR3) specifies an instruction address that can be used to trigger a breakpoint. To support debugging in a virtual



environment, an ASID can optionally be associated with the instruction address in the PC breakpoint registers. The optional specification of an ASID value is made using PBASID, and its exact inclusion within the breakpoint specification defined by the PBAC.

	31	24	23	16	15	8	7	0
Field	PBR3ASID			PBR2ASID		PBR1ASID		PBRASID
Reset	—							
R/W	W							
Address	0x14							

**Figure 11-16. PC Breakpoint ASID Register (PBASID)**

PBASID contains one 8-bit ASID values for each PC breakpoint register, as described in Table 11-21, which allows each PC breakpoint register to be associated with a unique virtual address and process.

**Table 11-21. PBASID Field Descriptions**

Bits	Name	Description
31–24	PBA3SID	PBR3ASID. Corresponds to the ASID associated with PBR3.
23–16	PBA2SID	PBR2ASID. Corresponds to the ASID associated with PBR2.
15–8	PBA1SID	PBR1ASID. Corresponds to the ASID associated with PBR1.
7–0	PBASID	PBRASID. Corresponds to the ASID associated with PBR.

## 11.5 Background Debug Mode (BDM)

The ColdFire Family implements a low-level system debugger in the microprocessor hardware. Communication with the development system is handled through a dedicated, high-speed serial command interface. The ColdFire architecture implements the BDM controller in a dedicated hardware module. Although some BDM operations, such as CPU register accesses, require the CPU to be halted, all other BDM commands, such as memory accesses, can be executed while the processor is running.

BDM is useful for the following reasons:

- In-circuit emulation is not needed, so physical and electrical characteristics of the system are not affected.
- BDM is always available for debugging the system and provides a communication link for upgrading firmware in existing systems.
- Provides high-speed cache downloading (500 Kbytes/sec), especially useful for flash programming
- Provides absolute control of the processor, and thus the system. This feature allows quick hardware debugging with the same tool set used for firmware development.

## 11.5.1 CPU Halt

Although most BDM operations can occur in parallel with CPU operations, unrestricted BDM operation requires the CPU to be halted. The sources that can cause the CPU to halt are listed below in order of priority:

1. A catastrophic fault-on-fault condition automatically halts the processor.
2. A hardware breakpoint can be configured to generate a pending halt condition similar to the assertion of  $\overline{\text{BKPT}}$ . This type of halt is always first made pending in the processor. Next, the processor samples for pending halt and interrupt conditions once per instruction. When a pending condition is asserted, the processor halts execution at the next sample point. See Section 11.6.1, “Theory of Operation.”
3. The execution of a HALT instruction immediately suspends execution. Attempting to execute HALT in user mode while  $\text{CSR}[\text{UHE}] = 0$  generates a privilege violation exception. If  $\text{CSR}[\text{UHE}] = 1$ , HALT can be executed in user mode. After HALT executes, the processor can be restarted by serial shifting a GO command into the debug module. Execution continues at the instruction after HALT.
4. The assertion of the  $\overline{\text{BKPT}}$  input is treated as a pseudo-interrupt; that is, asserting  $\overline{\text{BKPT}}$  creates a pending halt, which is postponed until the processor core samples for halts/interrupts. The processor samples for these conditions once during the execution of each instruction; if a pending halt is detected then, the processor suspends execution and enters the halted state.

The assertion of  $\overline{\text{BKPT}}$  should be considered in the following two special cases:

- After the system reset signal is negated, the processor waits for 16 processor clock cycles before beginning reset exception processing. If the  $\overline{\text{BKPT}}$  input is asserted within eight cycles after  $\overline{\text{RSTI}}$  is negated, the processor enters the halt state, signaling halt status (0xF) on the PSTDDATA outputs. While the processor is in this state, all resources accessible through the debug module can be referenced. This is the only chance to force the processor into emulation mode through  $\text{CSR}[\text{EMU}]$ .

After system initialization, the processor’s response to the GO command depends on the set of BDM commands performed while it is halted for a breakpoint. Specifically, if the PC register was loaded, the GO command causes the processor to exit halted state and pass control to the instruction address in the PC, bypassing normal reset exception processing. If the PC was not loaded, the GO command causes the processor to exit halted state and continue reset exception processing.

- The ColdFire architecture also handles a special case of  $\overline{\text{BKPT}}$  being asserted while the processor is stopped by execution of the STOP instruction. For this case, the processor exits the stopped mode and enters the halted state, at which point, all BDM commands may be exercised. When restarted, the processor continues by executing the next sequential instruction, that is, the instruction following the STOP opcode.

CSR[27–24] indicates the halt source, showing the highest priority source for multiple halt conditions. Debug module Revisions A and B clear CSR[27–24] upon a read of the CSR, but Revision C and D (in V4) do not. The debug GO command clears CSR[26–24].

HALT can be recognized by counting 0xFF occurrences on PSTDDATA. The count is necessary to determine between a possible data output value of 0xFF and the HALT condition. Because data always follows a marker (0x8, 0x9, 0xA, or 0xB), PSTDDATA can display no more than four data 0xFFs. Two such scenarios exist:

- A B marker occurs on the left nibble of PSTDDATA with the data of 0xFF following:  
PSTDDATA[7:0]  
0xBF  
0xFF  
0xFF  
0xFF  
0xFF (X indicates that the next PST value is guaranteed to not be 0xF)
- A B marker occurs on the right nibble of PSTDDATA with the data of 0xFF following:  
PSTDDATA[7:0]  
0xYB  
0xFF  
0xFF  
0xFF  
0xFF  
0xXY (X indicates that the PST value is guaranteed to not be 0xF; and Y indicates a PSTDDATA value that doesn't affect the 0xFF count).

Thus, a count of either nine or more sequential single 0xF values or five or more sequential 0xFF values signifies the HALT condition.

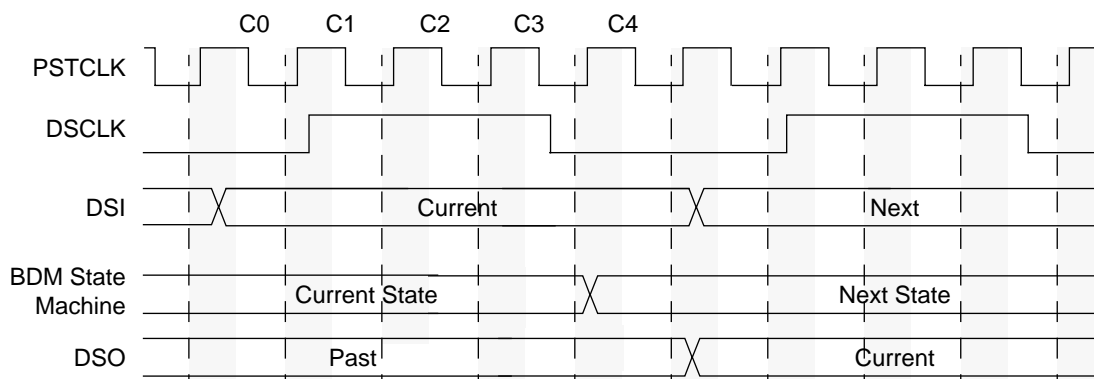
## 11.5.2 BDM Serial Interface

When the CPU is halted and PSTDDATA reflects the halt status, the development system can send unrestricted commands to the debug module. The debug module implements a synchronous protocol using two inputs (DSCLK and DSI) and one output (DSO), where DSO is specified as a delay relative to the rising edge of the processor clock. See Table 11-1. The development system serves as the serial communication channel master and must generate DSCLK.

The serial channel operates at a frequency from DC to 1/5 of the PSTCLK frequency. The channel uses full-duplex mode, where data is sent and received simultaneously by both master and slave devices. The transmission consists of 17-bit packets composed of a status/control bit and a 16-bit data word. As shown in Figure 11-17, all state transitions are

## Background Debug Mode (BDM)

enabled on a rising edge of the PSTCLK clock when DSCLK is high; that is, DSI is sampled and DSO is driven.



**Figure 11-17. Maximum BDM Serial Interface Timing**

DSCLK and DSI are synchronized inputs. DSCLK acts as a pseudo clock enable and is sampled, along with DSI, on the rising edge of PSTCLK. DSO is delayed from the DSCLK-enabled PSTCLK rising edge (registered after a BDM state machine state change). All events in the debug module's serial state machine are based on the PSTCLK rising edge. DSCLK must also be sampled low (on a positive edge of PSTCLK) between each bit exchange. The msb is sent first. Because DSO changes state based on an internally recognized rising edge of DSCLK, DSO cannot be used to indicate the start of a serial transfer. The development system must count clock cycles in a given transfer. C0–C4 are described as follows:

- C0: Set the state of the DSI bit.
- C1: First synchronization cycle for DSI (DSCLK is high).
- C2: Second synchronization cycle for DSI (DSCLK is high).
- C3: BDM state machine changes state depending upon DSI and whether the entire input data transfer has been transmitted.
- C4: DSO changes to next value.

### NOTE:

A not-ready response can be ignored except during a memory-referencing cycle. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods.

### 11.5.2.1 Receive Packet Format

The basic receive packet, Figure 11-18, consists of 16 data bits and 1 status bit.



**Figure 11-18. Receive BDM Packet**

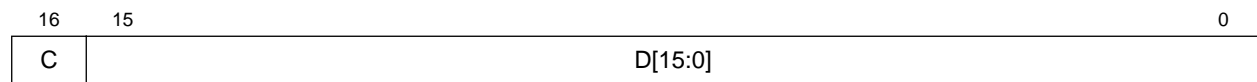
Table 11-22 describes receive BDM packet fields.

**Table 11-22. Receive BDM Packet Field Description**

Bits	Name	Description
16	S	Status. Indicates the status of CPU-generated messages listed below. The not-ready response can be ignored unless a memory-referencing cycle is in progress. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods. <div> <div>S</div> <div>Data</div> <div>Message</div> </div> <div> <div>0</div> <div>xxxx</div> <div>Valid data transfer</div> </div> <div> <div>0</div> <div>0xFFFF</div> <div>Status OK</div> </div> <div> <div>1</div> <div>0x0000</div> <div>Not ready with response; come again</div> </div> <div> <div>1</div> <div>0x0001</div> <div>Error: Terminated bus cycle; data invalid</div> </div> <div> <div>1</div> <div>0xFFFF</div> <div>Illegal command</div> </div>
15–0	Data	Data. Contains the message to be sent from the debug module to the development system. The response message is always a single word, with the data field encoded as shown above.

### 11.5.2.2 Transmit Packet Format

The basic transmit packet, Figure 11-19, consists of 16 data bits and 1 control bit.



**Figure 11-19. Transmit BDM Packet**

Table 11-23 describes transmit BDM packet fields.

**Table 11-23. Transmit BDM Packet Field Description**

Bits	Name	Description
16	C	Control. This bit is reserved. Command and data transfers initiated by the development system should clear C.
15–0	Data	Contains the data to be sent from the development system to the debug module.

## 11.5.3 BDM Command Set

Table 11-24 summarizes the BDM command set. Subsequent paragraphs contain detailed descriptions of each command. Issuing a BDM command when the processor is accessing debug module registers using the WDEBUI instruction causes undefined behavior.

**Table 11-24. BDM Command Summary**

Command	Mnemonic	Description	CPU State <sup>1</sup>	Section	Command (Hex)
Read A/D register	rareg/ rdreg	Read the selected address or data register and return the results through the serial interface.	Halted	11.5.3.3.1	0x218 {A/D, Reg[2:0]}
Write A/D register	wareg/ wdreg	Write the data operand to the specified address or data register.	Halted	11.5.3.3.2	0x208 {A/D, Reg[2:0]}
Read memory location	read	Read the data at the memory location specified by the longword address.	Steal	11.5.3.3.3	0x1900—byte 0x1940—word 0x1980—lword
Write memory location	write	Write the operand data to the memory location specified by the longword address.	Steal	11.5.3.3.4	0x1800—byte 0x1840—word 0x1880—lword
Dump memory block	dump	Used with READ to dump large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. A DUMP command retrieves subsequent operands.	Steal	11.5.3.3.5	0x1D00—byte 0x1D40—word 0x1D80—lword
Fill memory block	fill	Used with WRITE to fill large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. A FILL command writes subsequent operands.	Steal	11.5.3.3.6	0x1C00—byte 0x1C40—word 0x1C80—lword
Resume execution	go	The pipeline is flushed and refilled before resuming instruction execution at the current PC.	Halted	11.5.3.3.7	0x0C00
No operation	nop	Perform no operation; may be used as a null command.	Parallel	11.5.3.3.8	0x0000
Output the current PC	sync_pc	Capture the current PC and display it on the PSTDDATA output pins.	Parallel	11.5.3.3.9	0x0001
Read control register	rcreg	Read the system control register.	Halted	11.5.3.3.1 1	0x2980
Write control register	wcreg	Write the operand data to the system control register.	Halted	11.5.3.3.1 2	0x2880
Read debug module register	rdmreg	Read the debug module register.	Parallel	11.5.3.3.1 3	0x2D {0x4 <sup>2</sup> DRc[4:0]}
Write debug module register	wdmreg	Write the operand data to the debug module register.	Parallel	11.5.3.3.1 4	0x2C {0x4 <sup>2</sup> DRc[4:0]}

<sup>1</sup> General command effect and/or requirements on CPU operation:

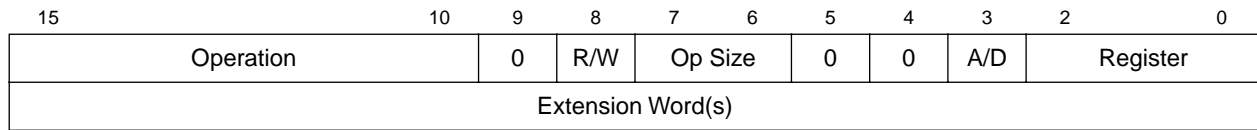
- Halted. The CPU must be halted to perform this command.
- Steal. Command generates bus cycles that can be interleaved with bus accesses.
- Parallel. Command is executed in parallel with CPU activity.

<sup>2</sup> 0x4 is a three-bit field.

Unassigned command opcodes are reserved by Motorola. All unused command formats within any revision level perform a NOP and return the illegal command response.

### 11.5.3.1 ColdFire BDM Command Format

All ColdFire Family BDM commands include a 16-bit operation word followed by an optional set of one or more extension words, as shown in Figure 11-20.



**Figure 11-20. BDM Command Format**

Table 11-25 describes BDM fields.

**Table 11-25. BDM Field Descriptions**

Bit	Name	Description										
15–10	Operation	Specifies the command. These values are listed in Table 11-24.										
9	0	Reserved										
8	R/W	Direction of operand transfer. 0 Data is written to the CPU or to memory from the development system. 1 The transfer is from the CPU to the development system.										
7–6	Operand Size	Operand data size for sized operations. Addresses are expressed as 32-bit absolute values. Note that a command performing a byte-sized memory read leaves the upper 8 bits of the response data undefined. Referenced data is returned in the lower 8 bits of the response. <table><tr><td>Operand Size</td><td>Bit Values</td></tr><tr><td>00 Byte</td><td>8 bits</td></tr><tr><td>01 Word</td><td>16 bits</td></tr><tr><td>10 Longword</td><td>32 bits</td></tr><tr><td>11 Reserved</td><td>—</td></tr></table>	Operand Size	Bit Values	00 Byte	8 bits	01 Word	16 bits	10 Longword	32 bits	11 Reserved	—
Operand Size	Bit Values											
00 Byte	8 bits											
01 Word	16 bits											
10 Longword	32 bits											
11 Reserved	—											
5–4	00	Reserved										
3	A/D	Address/data. Determines whether the register field specifies a data or address register. 0 Indicates a data register. 1 Indicates an address register.										
2–0	Register	Contains the register number in commands that operate on processor registers.										

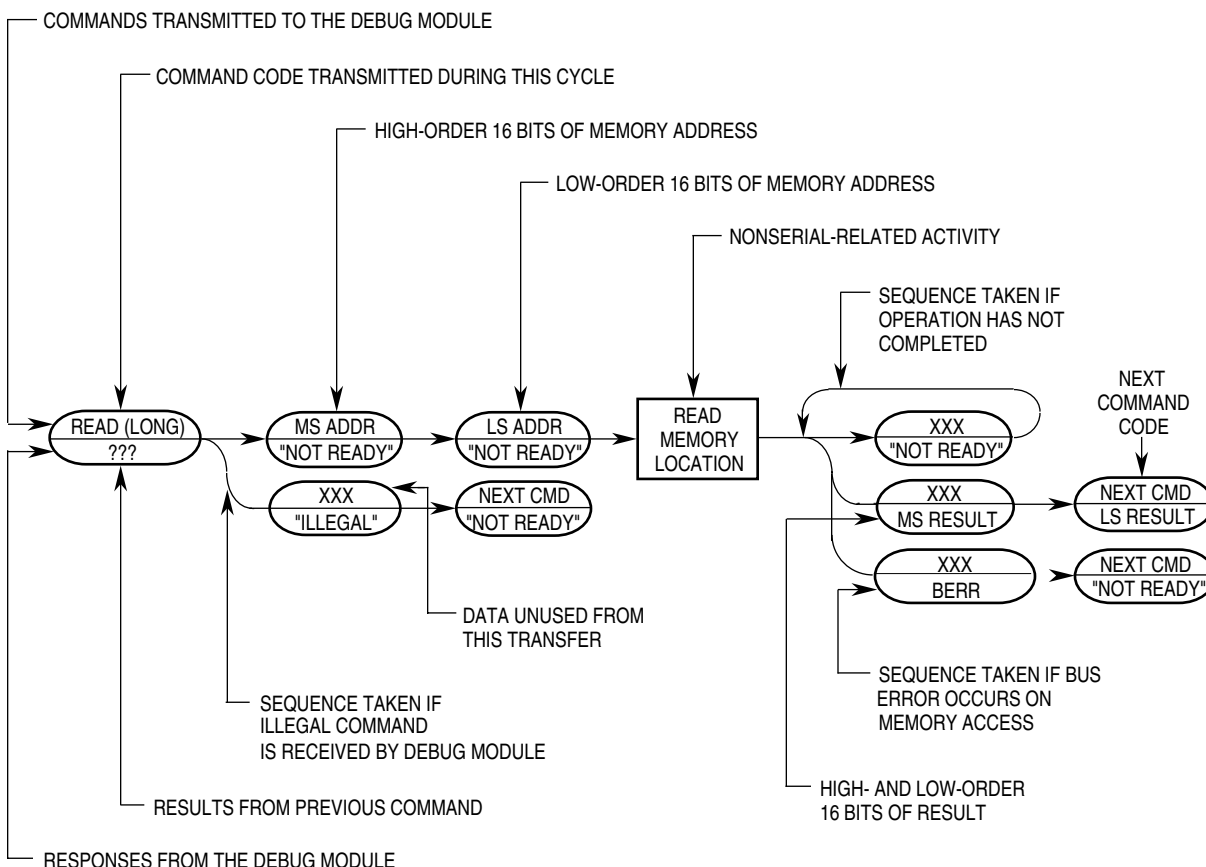
#### 11.5.3.1.1 Extension Words as Required

Some commands require extension words for addresses or immediate data. Addresses require two extension words because only absolute long addressing is permitted. Longword accesses are forcibly longword-aligned and word accesses are forcibly word-aligned. Immediate data can be 1 or 2 words long. Byte and word data each requires a single extension word and longword data requires two extension words.

Operands and addresses are transferred most-significant word first. In the following descriptions of the BDM command set, the optional set of extension words is defined as address, data, or operand data.

### 11.5.3.2 Command Sequence Diagrams

The command sequence diagram in Figure 11-21 shows serial bus traffic for commands. Each bubble represents a 17-bit bus transfer. The top half of each bubble indicates the data the development system sends to the debug module; the bottom half indicates the debug module's response to the previous development system commands. Command and result transactions overlap to minimize latency.



**Figure 11-21. Command Sequence Diagram**

The sequence is as follows:

- In cycle 1, the development system command is issued (READ in this example). The debug module responds with either the low-order results of the previous command or a command complete status of the previous command, if no results are required.
- In cycle 2, the development system supplies the high-order 16 address bits. The debug module returns a not-ready response unless the received command is decoded as unimplemented, which is indicated by the illegal command encoding. If this occurs, the development system should retransmit the command.



**NOTE:**

A not-ready response can be ignored except during a memory-referencing cycle. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods.

- In cycle 3, the development system supplies the low-order 16 address bits. The debug module always returns a not-ready response.
- At the completion of cycle 3, the debug module initiates a memory read operation. Any serial transfers that begin during a memory access return a not-ready response.
- Results are returned in the two serial transfer cycles after the memory access completes. For any command performing a byte-sized memory read operation, the upper 8 bits of the response data are undefined and the referenced data is returned in the lower 8 bits. The next command's opcode is sent to the debug module during the final transfer. If a memory or register access is terminated with a bus error, the error status ( $S = 1$ ,  $DATA = 0x0001$ ) is returned instead of result data.

### 11.5.3.3 Command Set Descriptions

The following sections describe the commands summarized in Table 11-24.

**NOTE:**

The BDM status bit ( $S$ ) is 0 for normally completed commands;  $S = 1$  for illegal commands, not-ready responses, and transfers with bus-errors. Section 11.5.2, “BDM Serial Interface,” describes the receive packet format.

Motorola reserves unassigned command opcodes for future expansion. Unused command formats in any revision level perform a NOP and return an illegal command response.

### 11.5.3.3.1 Read A/D Register (RAREG/RDREG)

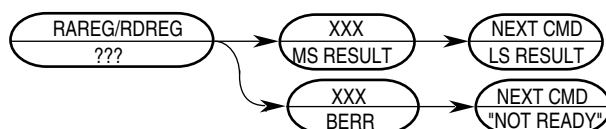
Read the selected address or data register and return the 32-bit result. A bus error response is returned if the CPU core is not halted.

Command/Result Formats:

	15	12	11	8	7	4	3	2	0
Command	0x2		0x1		0x8		A/D	Register	
Result	D[31:16]								
	D[15:0]								

**Figure 11-22. RAREG/RDREG Command Format**

Command Sequence:



**Figure 11-23. RAREG/RDREG Command Sequence**

Operand Data: None

Result Data: The contents of the selected register are returned as a longword value, most-significant word first.

11.5.3.3.2 Write A/D Register (WAREG/WDREG)

The operand longword data is written to the specified address or data register. A write alters all 32 register bits. A bus error response is returned if the CPU core is not halted.

Command Format:

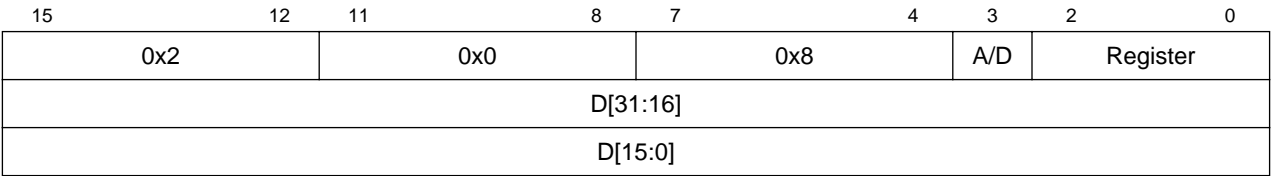


Figure 11-24. WAREG/WDREG Command Format

Command Sequence

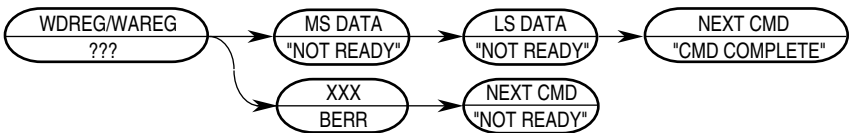


Figure 11-25. WAREG/WDREG Command Sequence

Operand Data	Longword data is written into the specified address or data register. The data is supplied most-significant word first.
Result Data	Command complete status is indicated by returning 0xFFFF (with S cleared) when the register write is complete.

### 11.5.3.3.3 Read Memory Location (READ)

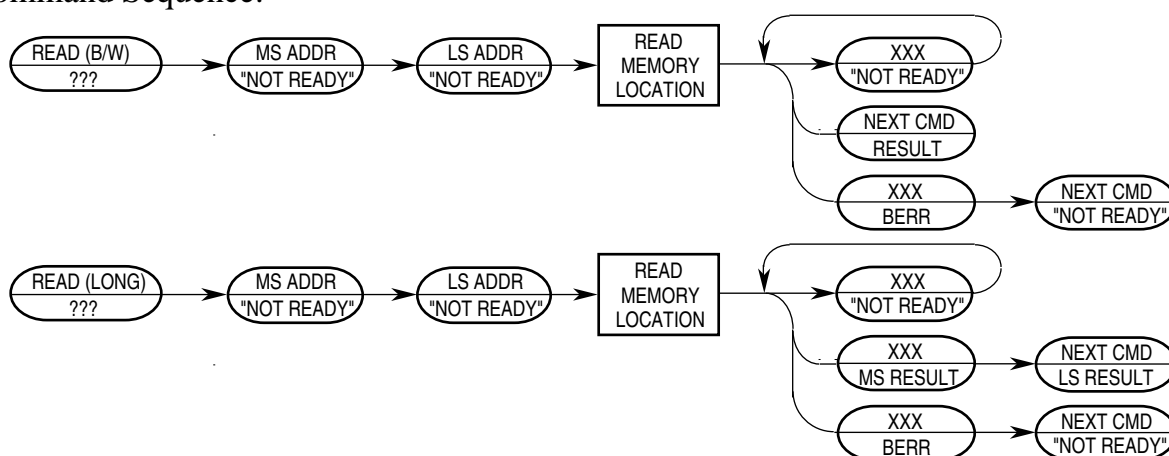
Read data at the longword address. Address space is defined by BAAR[TT,TM]. Hardware forces low-order address bits to zeros for word and longword accesses to ensure that word addresses are word-aligned and longword addresses are longword-aligned.

Command/Result Formats:

		15				12		11		8		7		4		3		0	
Byte	Command	0x1				0x9				0x0				0x0					
		A[31:16]																	
	A[15:0]																		
	Result	X	X	X	X	X	X	X	X	D[7:0]									
Word	Command	0x1				0x9				0x4				0x0					
		A[31:16]																	
		A[15:0]																	
	Result	D[15:0]																	
Longword	Command	0x1				0x9				0x8				0x0					
		A[31:16]																	
		A[15:0]																	
	Result	D[31:16]																	
		D[15:0]																	

**Figure 11-26. READ Command/Result Formats**

Command Sequence:



**Figure 11-27. READ Command Sequence**

Operand Data

The only operand is the longword address of the requested location.

Result Data

Word results return 16 bits of data; longword results return 32. Bytes are returned in the LSB of a word result, the upper byte is undefined. 0x0001 (S = 1) is returned if a bus error occurs.

### 11.5.3.3.4 Write Memory Location (WRITE)

Write data to the memory location specified by the longword address. The address space is defined by BAAR[TT,TM]. Hardware forces low-order address bits to zeros for word and longword accesses to ensure that word addresses are word-aligned and longword addresses are longword-aligned.

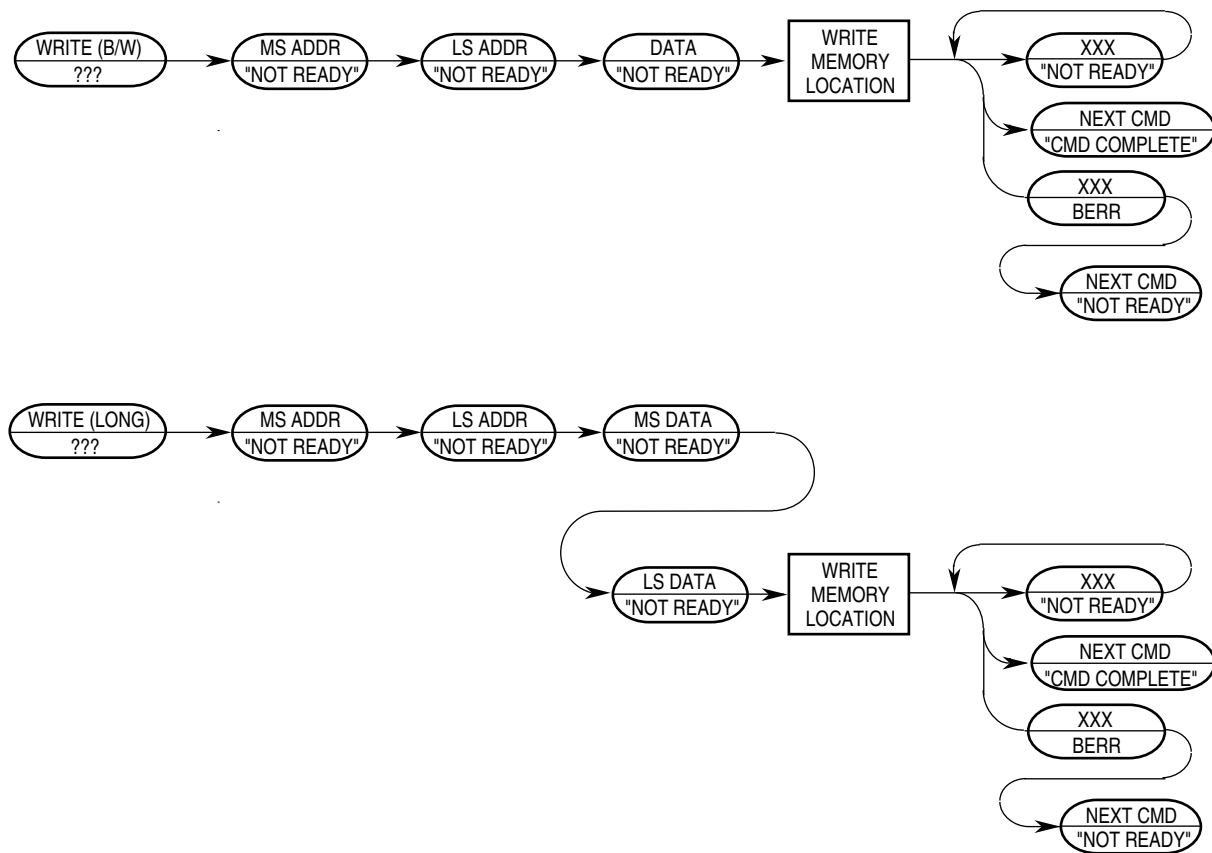
Command Formats:

	15	12	11	8	7	4	3	1	
Byte	0x1				0x8		0x0		0x0
	A[31:16]								
	A[15:0]								
	X	X	X	X	X	X	X	X	D[7:0]
Word	0x1				0x8		0x4		0x0
	A[31:16]								
	A[15:0]								
	D[15:0]								
Longword	0x1				0x8		0x8		0x0
	A[31:16]								
	A[15:0]								
	D[31:16]								
	D[15:0]								

**Figure 11-28. WRITE Command Format**

## Background Debug Mode (BDM)

### Command Sequence:



**Figure 11-29. WRITE Command Sequence**

#### Operand Data

This two-operand instruction requires a longword absolute address that specifies a location to which the data operand is to be written. Byte data is sent as a 16-bit word, justified in the LSB; 16- and 32-bit operands are sent as 16 and 32 bits, respectively

#### Result Data

Command complete status is indicated by returning 0xFFFF (with S cleared) when the register write is complete. A value of 0x0001 (with S set) is returned if a bus error occurs.

### 11.5.3.3.5 Dump Memory Block (DUMP)

DUMP is used with the READ command to access large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. If an initial READ is not executed before the first DUMP, an illegal command response is returned. The DUMP command retrieves subsequent operands. The initial address is incremented by the operand size (1, 2, or 4) and saved in a temporary register. Subsequent DUMP commands use this address, perform the memory read, increment it by the current operand size, and store the updated address in the temporary register.

#### NOTE:

DUMP does not check for a valid address; it is a valid command only when preceded by NOP, READ, or another DUMP command. Otherwise, an illegal command response is returned. NOP can be used for intercommand padding without corrupting the address pointer.

The size field is examined each time a DUMP command is processed, allowing the operand size to be dynamically altered.

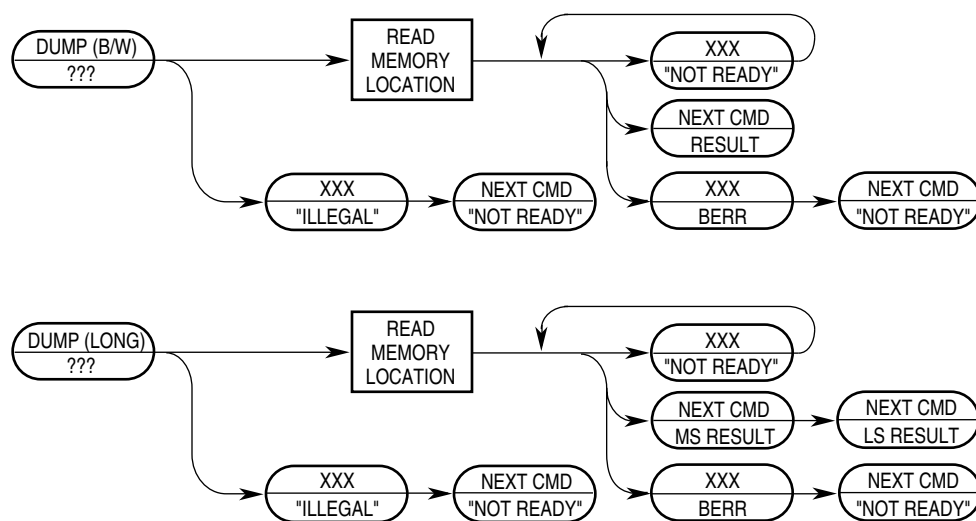
Command/Result Formats:

		15			12			11			8			7			4			3			0
Byte	Command	0x1				0xD				0x0				0x0									
	Result	X	X	X	X	X	X	X	X	D[7:0]													
Word	Command	0x1				0xD				0x4				0x0									
	Result	D[15:0]																					
Longword	Command	0x1				0xD				0x8				0x0									
	Result	D[31:16]																					
		D[15:0]																					

**Figure 11-30. DUMP Command/Result Formats**

## Background Debug Mode (BDM)

Command Sequence:



**Figure 11-31. DUMP Command Sequence**

Operand Data:

None

Result Data:

Requested data is returned as either a word or longword. Byte data is returned in the least-significant byte of a word result. Word results return 16 bits of significant data; longword results return 32 bits. A value of 0x0001 (with S set) is returned if a bus error occurs.



### 11.5.3.3.6 Fill Memory Block (FILL)

A FILL command is used with the WRITE command to access large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. The FILL command writes subsequent operands. The initial address is incremented by the operand size (1, 2, or 4) and saved in a temporary register after the memory write. Subsequent FILL commands use this address, perform the write, increment it by the current operand size, and store the updated address in the temporary register.

If an initial WRITE is not executed preceding the first FILL command, the illegal command response is returned.

#### NOTE:

The FILL command does not check for a valid address: FILL is a valid command only when preceded by another FILL, a NOP, or a WRITE command. Otherwise, an illegal command response is returned. The NOP command can be used for intercommand padding without corrupting the address pointer.

The size field is examined each time a FILL command is processed, allowing the operand size to be altered dynamically.

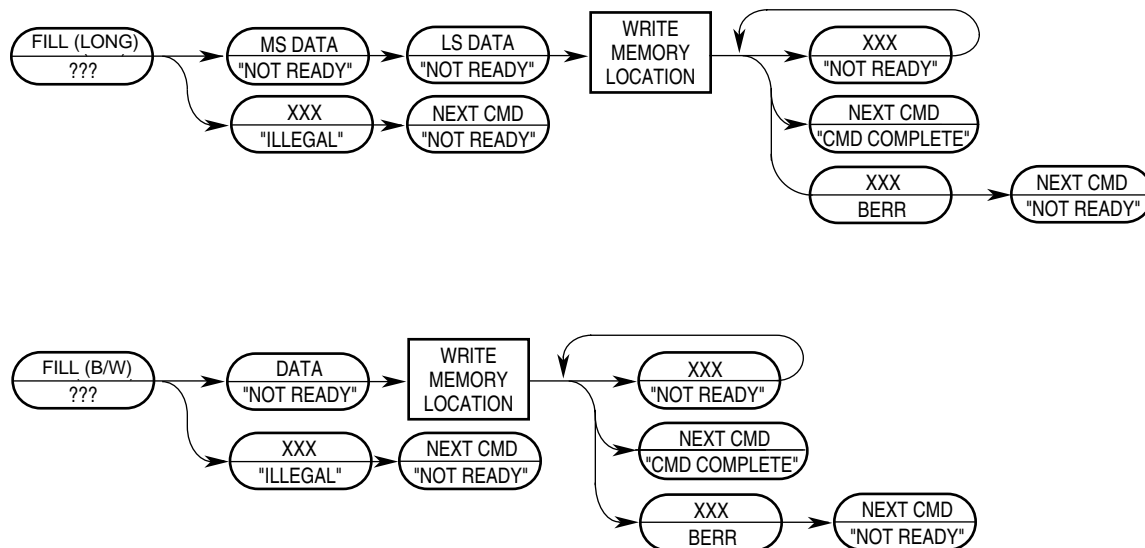
Command Formats:

	15				12				11				8				7				4				3				0			
Byte	0x1								0xC								0x0								0x0							
	X	X	X	X	X	X	X	X	X	X	X	X	X	D[7:0]																		
Word	0x1								0xC								0x4								0x0							
	D[15:0]																															
Longword	0x1								0xC								0x8								0x0							
	D[31:16]																															
	D[15:0]																															

**Figure 11-32. FILL Command Format**

## Background Debug Mode (BDM)

### Command Sequence:



**Figure 11-33. FILL Command Sequence**

**Operand Data:** A single operand is data to be written to the memory location. Byte data is sent as a 16-bit word, justified in the least-significant byte; 16- and 32-bit operands are sent as 16 and 32 bits, respectively.

**Result Data:** Command complete status (0xFFFF) is returned when the register write is complete. A value of 0x0001 (with S set) is returned if a bus error occurs.

11.5.3.3.7 Resume Execution (Go)

The pipeline is flushed and refilled before normal instruction execution resumes. Prefetching begins at the current address in the PC and at the current privilege level. If any register (such as the PC or SR) is altered by a BDM command while the processor is halted, the updated value is used when prefetching resumes. If a GO command is issued and the CPU is not halted, the command is ignored.

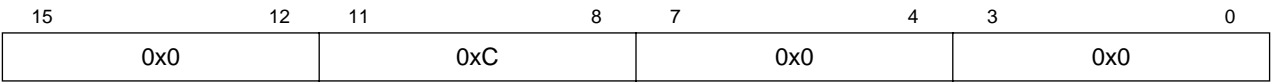


Figure 11-34. go Command Format

Command Sequence:

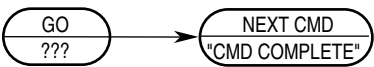


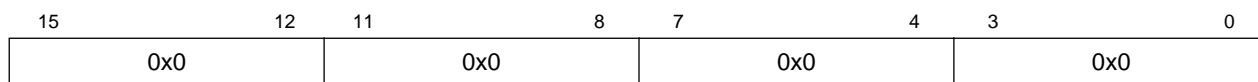
Figure 11-35. go Command Sequence

- Operand Data: None
- Result Data: The command-complete response (0xFFFF) is returned during the next shift operation.

### 11.5.3.3.8 No Operation (NOP)

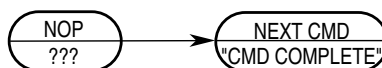
NOP performs no operation and may be used as a null command where required.

Command Formats:



**Figure 11-36. NOP Command Format**

Command Sequence:



**Figure 11-37. NOP Command Sequence**

Operand Data:

None

Result Data:

The command-complete response, 0xFFFF (with S cleared), is returned during the next shift operation.

### 11.5.3.3.9 Synchronize PC to the PSTDDATA Lines (SYNC\_PC)

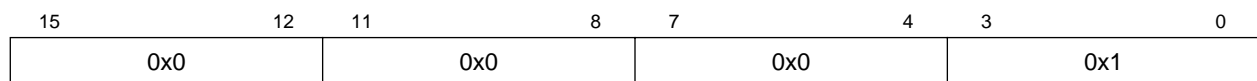
The SYNC\_PC command captures the current PC and displays it on the PSTDDATA outputs. After the debug module receives the command, it sends a signal to the ColdFire processor that the current PC must be displayed. The processor then forces an instruction fetch at the next PC with the address being captured in the DDATA logic under control of CSR[BTB]. The specific sequence of PSTDDATA values is as follows:

1. Debug signals a SYNC\_PC command is pending.
2. CPU completes the current instruction.
3. CPU forces an instruction fetch to the next PC, generates a PST = 0x5 value indicating a taken branch and signals the capture of DDATA.
4. The instruction address corresponding to the PC is captured.
5. The PST marker (0x9–0xB) is generated and displayed as defined by CSR[BTB] followed by the captured PC address.

If the option to display ASID is enabled (CSR[3] = 1), the 8-bit ASID follows the address. That is, the PSTDDATA sequence is {0x5, Marker, Instruction Address, 0x8, ASID}, where the 0x8 is the marker for the ASID.

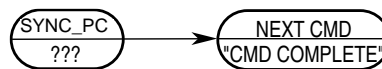
The SYNC\_PC command can be used to dynamically access the PC for performance monitoring. The execution of this command is considerably less obtrusive to the real-time operation of an application than a HALT-CPU/READ-PC/RESUME command sequence.

Command Formats:



**Figure 11-38. SYNC\_PC Command Format**

Command Sequence:



**Figure 11-39. SYNC\_PC Command Sequence**

Operand Data: None

Result Data: Command complete status (0xFFFF) is returned when the register write is complete.

### 11.5.3.3.10 Force Transfer Acknowledge (FORCE\_TA)

DEBUG\_D logic implements the new FORCE\_TA serial BDM command to resolve a hung bus condition. In some system designs, references to certain unmapped memory addresses may cause the external bus to hang with no transfer acknowledge generated by any bus responders. The FORCE\_TA forces generation of a transfer acknowledge signal, which can

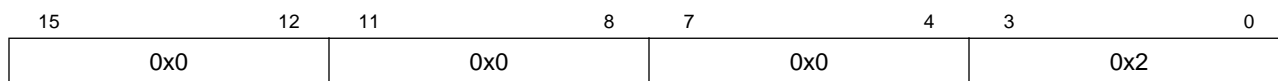
## Background Debug Mode (BDM)

be logically summed into the normal acknowledge logic located in the system integration module (SIM) outside of the ColdFire core.

There are two scenarios of interest, one caused by a processor access and the other caused by a BDM access. The following sequences identify the operations needed to break the hung bus condition:

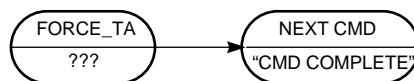
- Bus hang caused by processor or external or internal alternate master:
  - Assert the breakpoint input to force a processor core halt.
  - If the bus hang was caused by a processor access, send in FORCE\_TA commands until the processor is halted, as signaled by PST = 0xF. Due to pipeline and store buffer depths, many memory accesses may be queued up behind the access causing the bus hang. Repeated FORCE\_TA commands eventually allow processing of all these pending accesses. As soon as the processor is halted, the system reaches a quiescent, controllable state.
  - If the hang was caused by another master, such as a DMA channel, the processor can halt immediately. In this case as well, multiple assertions of the FORCE\_TA command may be required to terminate the alternate master's errant access.
- Bus hang caused by BDM access:
  - It is assumed the processor is already halted at the time of the errant BDM access. To resolve the hung bus, it is necessary to process four or more FORCE\_TA commands because the BDM command may have initiated a cache line access, which fetches 4 longwords, each needing a unique transfer acknowledge.

Formats:



**Figure 11-40. FORCE\_TA Command**

Command Sequence:



**Figure 11-41. FORCE\_TA Command Sequence**

Operand Data: None

Result Data: The command complete response, 0xFFFF (with the status bit cleared), is returned during the next shift operation. This response indicates the FORCE\_TA command was processed correctly and does not necessarily reflect the status of any internal bus.

### 11.5.3.3.11 Read Control Register (RCREG)

Read the selected control register and return the 32-bit result. Accesses to the processor/memory control registers are always 32 bits wide, regardless of register width. The second and third words of the command form a 32-bit address, which the debug module uses to generate a special bus cycle to access the specified control register. The 12-bit Rc field is the same as that used by the MOVEC instruction.

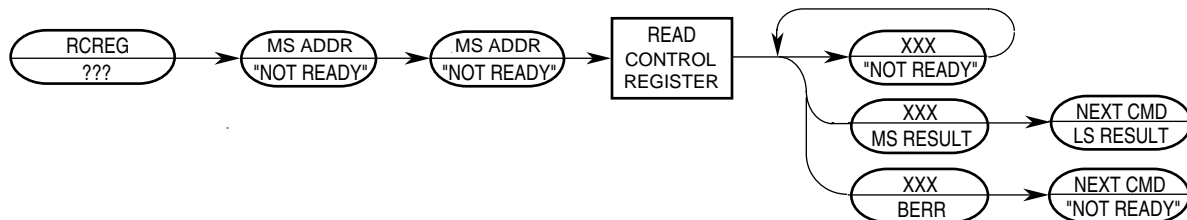
Command/Result Formats:

	15	12	11	8	7	4	3	0
Command	0x2		0x9		0x8		0x0	
	0x0		0x0		0x0		0x0	
	0x0		Rc					
Result	D[31:16]							
	D[15:0]							

**Figure 11-42. RCREG Command/Result Formats**

Rc encoding: See Table 2-4.

Command Sequence:



**Figure 11-43. RCREG Command Sequence**

**Operand Data:** The only operand is the 32-bit Rc control register select field.

**Result Data:** Control register contents are returned as a longword, most-significant word first. The implemented portion of registers smaller than 32 bits is guaranteed correct; other bits are undefined.

### BDM Accesses of the Stack Pointer Registers (A7: SSP and USP)

The Version 4 ColdFire core supports two unique stack pointer (A7) registers: the supervisor stack pointer (SSP) and the user stack pointer (USP). The hardware implementation of these two programmable-visible 32-bit registers does not uniquely identify one as the SSP and the other as the USP. Rather, the hardware uses one 32-bit register as the currently-active A7; the other is named simply the OTHER\_A7. Thus, the contents of the two hardware registers is a function of the operating mode of the processor:

```

if SR[S] = 1
    then
        A7 = Supervisor Stack Pointer
        OTHER_A7 = User Stack Pointer

```

## Background Debug Mode (BDM)

```
else                A7 = User Stack Pointer
OTHER_A7 = Supervisor Stack Pointer
```

The BDM programming model supports reads and writes to A7 and OTHER\_A7 directly. It is the responsibility of the external development system to determine the mapping of A7 and OTHER\_A7 to the two program-visible definitions (supervisor and user stack pointers), based on the SR[S].

### BDM Accesses of the EMAC Registers

The presence of rounding logic in the output datapath of the EMAC requires special care for BDM-initiated reads and writes of its programming model. In particular, any result rounding modes must be disabled during the read/write process so the exact bit-wise EMAC register contents are accessed.

For example, a BDM read of an accumulator (ACCx) requires the following sequence:

```
BdmReadACCx (
    rcreg    macsr;                // read current macsr contents & save
    wcreg    #0,macsr;            // disable all rounding modes
    rcreg    ACCx;                // read the desired accumulator
    wcreg    #saved_data,macsr;    // restore the original macsr
)
```

Likewise to write an accumulator register, the following BDM sequence is needed:

```
BdmWriteACCx (
    rcreg    macsr;                // read current macsr contents & save
    wcreg    #0,macsr;            // disable all rounding modes
    wcreg    #data,ACCx;          // write the desired accumulator
    wcreg    #saved_data,macsr;    // restore the original macsr
)
```

Additionally, writes to the accumulator extension registers must be performed after the corresponding accumulators are updated because a write to any accumulator alters the corresponding extension register contents.

For more information on saving and restoring the complete EMAC programming model, see the appropriate section of the EMAC chapter.

### BDM Accesses of Floating-Point Data Registers (FPn)

The ColdFire debug architecture allows BDM accesses of the entire programming model (including all FPU-related registers) of the processor core using RCREG and WCREG. However, certain hardware restrictions require the accesses related to the 64-bit FPn data registers be performed in a certain manner to guarantee correct operation.

The serial BDM command structure supports 8-, 16- and 32-bit accesses, but there is no direct mechanism for accessing 64-bit data values. Rather than changing this well-established protocol and command set, BDM accesses of 64-bit data values are treated as two independent 32-bit references. In particular, 64-bit FPn data registers are treated as two separate values from the BDM perspective. Each FPn is partitioned into upper and lower longwords, FPU<sub>n</sub> and FPL<sub>n</sub>.



Either longword can be read first. The processor treats the BDM read command as a pseudo-FMOVEM. Accordingly, all rounding modes and exception enables are ignored and the 32-bit contents of  $FPU_n$  or  $FPL_n$  are sent to the debug module for transmission over the serial communication channel. The FPU programming model is unchanged.

To write to an FPU data register,  $FPU_n$  must be written first and followed by a write to  $FPL_n$ . The processor operates as follows: The BDM write to  $FPU_n$  is performed loads the upper 32 bits of an internal double-precision operand register. The BDM write to  $FPL_n$  loads the supplied operand into the lower 32 bits of the same internal register, after which the entire 64-bit value is loaded into the selected  $FP_n$ . Failure to execute this sequence of commands produces an undefined value in the  $FPU_n$ .

Note that any BDM write of an FPU register changes the internal state from NULL to IDLE.

### 11.5.3.3.12 Write Control Register (WCREG)

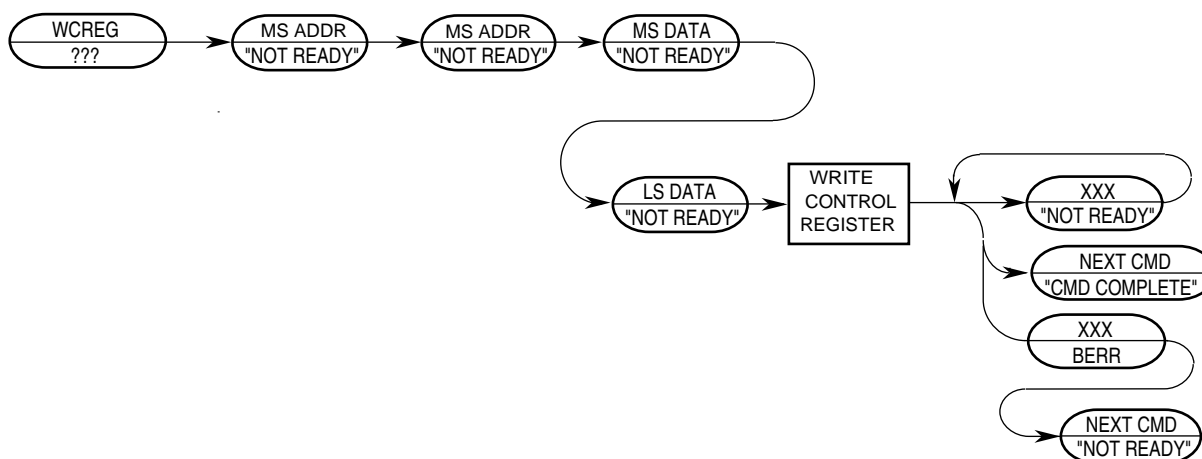
The operand (longword) data is written to the specified control register. The write alters all 32 register bits. See the RCREG instruction description for the Rc encoding and for additional notes on writes to the A7 stack pointers and the EMAC and FPU programming models.

Command/Result Formats:

	15	12	11	8	7	4	3	0
Command	0x2		0x8		0x8		0x0	
	0x0		0x0		0x0		0x0	
	0x0		Rc					
Result	D[31:16]							
	D[15:0]							

**Figure 11-44. WCREG Command/Result Formats**

Command Sequence:



**Figure 11-45. WCREG Command Sequence**

**Operand Data:** This instruction requires two longword operands. The first selects the register to which the operand data is to be written; the second contains the data.

**Result Data:** Successful write operations return 0xFFFF. Bus errors on the write cycle are indicated by the setting of bit 16 in the status message and by a data pattern of 0x0001.

### 11.5.3.3.13 Read Debug Module Register (RDMREG)

Read the selected debug module register and return the 32-bit result. The only valid register selection for the RDMREG command is CSR (DRc = 0x00). Note that this read of the CSR clears the trigger status bits (CSR[BSTAT]) if either a level-2 breakpoint has been triggered or a level-1 breakpoint has been triggered and no level-2 breakpoint has been enabled.

Command/Result Formats:

	15	12	11	8	7	5	4	0
Command	0x2		0xD		100		DRc	
Result	D[31:16]							
	D[15:0]							

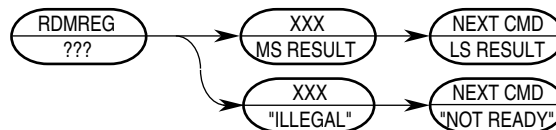
**Figure 11-46. RDMREG BDM Command/Result Formats**

Table 11-26 shows the definition of DRc encoding.

**Table 11-26. Definition of DRc Encoding—Read**

DRc[4:0]	Debug Register Definition	Mnemonic	Initial State	Page
0x00	Configuration/Status	CSR	0x0	p. 11-17
0x01–0x1F	Reserved	—	—	—

Command Sequence:



**Figure 11-47. RDMREG Command Sequence**

Operand Data: None

Result Data: The contents of the selected debug register are returned as a longword value. The data is returned most-significant word first.

11.5.3.3.14 Write Debug Module Register (WDMREG)

The operand (longword) data is written to the specified debug module register. All 32 bits of the register are altered by the write. DSCLK must be inactive while the debug module register writes from the CPU accesses are performed using the WDEBUG instruction.

Command Format:

Figure 11-48. WDMREG BDM Command Format

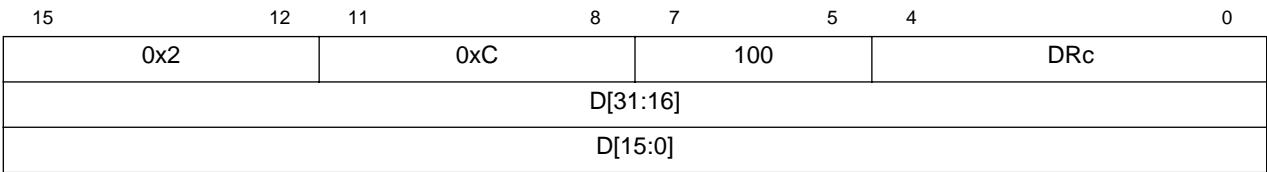


Table 11-6 shows the definition of the DRc write encoding.

Command Sequence:

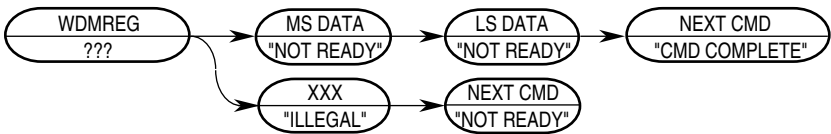


Figure 11-49. WDMREG Command Sequence

- Operand Data:
- Longword data is written into the specified debug register. The data is supplied most-significant word first.
- Result Data:
- Command complete status (0xFFFF) is returned when register write is complete.

## 11.6 Real-Time Debug Support

The ColdFire Family provides support debugging real-time applications. For these types of embedded systems, the processor must continue to operate during debug. The foundation of this area of debug support is that while the processor cannot be halted to allow debugging, the system can generally tolerate the small intrusions of the BDM inserting instructions into the pipeline with minimal effect on real-time operation.

The debug module provides three types of breakpoints: PC with mask, operand address range, and data with mask. These breakpoints can be configured into one- or two-level triggers with the exact trigger response also programmable. The debug module programming model can be written from either the external development system using the debug serial interface or from the processor's supervisor programming model using the WDEBUG instruction. Only CSR is readable using the external development system.

### 11.6.1 Theory of Operation

Breakpoint hardware can be configured through TDR[TCR] to respond to triggers by displaying PSTDDATA, initiating a processor halt, or generating a debug interrupt. As shown in Table 11-27, when a breakpoint is triggered, an indication (CSR[BSTAT]) is provided on the PSTDDATA output port of the DDATA information when it is not displaying captured processor status, operands, or branch addresses. See Section 11.3.2, "Processor Stopped or Breakpoint State Change (PST = 0xE)."

**Table 11-27. PSTDDATA Nibble/CSR[BSTAT] Breakpoint Response**

PSTDDATA Nibble/CSR[BSTAT] <sup>1</sup>	Breakpoint Status
0000/0000	No breakpoints enabled
0010/0001	Waiting for level-1 breakpoint
0100/0010	Level-1 breakpoint triggered
1010/0101	Waiting for level-2 breakpoint
1100/0110	Level-2 breakpoint triggered

<sup>1</sup> Encodings not shown are reserved for future use.

The breakpoint status is also posted in CSR. Note that CSR[BSTAT] is cleared by a CSR read when either a level-2 breakpoint is triggered or a level-1 breakpoint is triggered and a level-2 breakpoint is not enabled. Status is also cleared by writing to either TDR or XTDR to disable trigger options.

BDM instructions use the appropriate registers to load and configure breakpoints. As the system operates, a breakpoint trigger generates the response defined in TDR.

PC breakpoints are treated in a precise manner: exception recognition and processing are initiated before the excepting instruction is executed. All other breakpoint events are recognized on the processor's local bus, but are made pending to the processor and

sampled like other interrupt conditions. As a result, these interrupts are said to be imprecise.

In systems that tolerate the processor being halted, a BDM-entry can be used. With  $TDR[TRC] = 01$ , a breakpoint trigger causes the core to halt ( $PST = 0xF$ ).

If the processor core cannot be halted, the debug interrupt can be used. With this configuration,  $TDR[TRC] = 10$ , the breakpoint trigger becomes a debug interrupt to the processor, which is treated higher than the nonmaskable level-7 interrupt request. As with all interrupts, it is made pending until the processor reaches a sample point, which occurs once per instruction. Again, the hardware forces the PC breakpoint to occur before the targeted instruction executes and is precise. This is possible because the PC breakpoint is enabled when interrupt sampling occurs. For address and data breakpoints, reporting is considered imprecise because several instructions may execute after the triggering address or data is detected.

As soon as the debug interrupt is recognized, the processor aborts execution and initiates exception processing. This event is signaled externally by the assertion of a unique PST value ( $PST = 0xD$ ) for multiple cycles. The core enters emulator mode when exception processing begins. After the standard 8-byte exception stack is created, the processor fetches a unique exception vector from the vector table. Table 11-28 describes the two unique entries that distinguish PC breakpoints from other trigger events.

**Table 11-28. Exception Vector Assignments**

Vector Number	Vector Offset (Hex)	Stacked Program Counter	Assignment
12	0x030	Next	Non-PC-breakpoint debug interrupt
13	0x034	Next	PC-breakpoint debug interrupt

(Refer to the *ColdFire Programmer's Reference Manual*.)

In the case of a two-level trigger, the last breakpoint event determines the exception vector; however, if the second-level trigger is PC || Address {&& Data} (as shown in the last condition in the code example in Section 11.4.9.1, “Resulting Set of Possible Trigger Combinations”), the vector taken is determined by the first condition that occurs after the first-level trigger: vector 13 if PC occurs first or vector 12 if Address {&& Data} occurs first. If both occur simultaneously, the non-PC-breakpoint debug interrupt is taken (vector number 12).

Execution continues at the instruction address in the vector corresponding to the breakpoint triggered. The debug interrupt handler can use supervisor instructions to save the necessary context such as the state of all program-visible registers into a reserved memory area.

During a debug interrupt service routine, all normal interrupt requests are evaluated and sampled once per instruction. If any exception occurs, the processor responds as follows:

1. It saves a copy of the current value of the emulator mode state bit and then exits emulator mode by clearing the actual state.
2. Bit 1 of the fault status field (FS1) in the next exception stack frame is set to indicate the processor was in emulator mode when the interrupt occurred. This corresponds to bit 17 of the longword at the top of the system stack. See Section 7.3, “Exception Stack Frame Definition.”
3. It passed control to the appropriate exception handler.
4. It executes an RTE instruction when the exception handler finishes. During the processing of the RTE, FS1 is reloaded from the system stack. If this bit is set, the processor sets the emulator mode state and resumes execution of the original debug interrupt service routine. This is signaled externally by the generation of the PST value that originally identified the debug interrupt exception, that is,  $PST = 0xD$ .

Fault status encodings are listed in Table 10-2. Implementation of this debug interrupt handling fully supports the servicing of a number of normal interrupt requests during a debug interrupt service routine.

The emulator mode state bit is essentially changed to be a program-visible value, stored into memory during exception stack frame creation, and loaded from memory by the RTE instruction.

When debug interrupt operations complete, the RTE instruction executes and the processor exits emulator mode. After the debug interrupt handler completes execution, the external development system can use BDM commands to read the reserved memory locations.

In Revision A, if a hardware breakpoint such as a PC trigger is left unmodified by the debug interrupt service routine, another debug interrupt is generated after the completion of the RTE instruction. In Revisions B and C, the generation of another debug interrupt during the first instruction after the RTE exits emulator mode is inhibited. This behavior is consistent with the existing logic involving trace mode where the first instruction executes before another trace exception is generated. Thus, all hardware breakpoints are disabled until the first instruction after the RTE completes execution, regardless of the programmed trigger response.

### 11.6.1.1 Emulator Mode

Emulator mode is used to facilitate nonintrusive emulator functionality. This mode can be entered in three different ways:

- Setting  $CSR[EMU]$  forces the processor into emulator mode. EMU is examined only if  $\overline{RSTI}$  is negated and the processor begins reset exception processing. It can be set while the processor is halted before reset exception processing begins. See Section 11.5.1, “CPU Halt.”
- A debug interrupt always puts the processor in emulation mode when debug interrupt exception processing begins.

## Debug C Definition of PSTDDATA Outputs

- Setting CSR[TRC] forces the processor into emulation mode when trace exception processing begins.

While operating in emulation mode, the processor exhibits the following properties:

- Unmasked interrupt requests are serviced. The resulting interrupt exception stack frame has FS[1] set to indicate the interrupt occurred while in emulator mode.
- If CSR[MAP] = 1, all caching of memory and the SRAM module are disabled. All memory accesses are forced into a specially mapped address space signaled by TT = 0x2, TM = 0x5 or 0x6. This includes stack frame writes and the vector fetch for the exception that forced entry into this mode.

The RTE instruction exits emulation mode. The processor status output port provides a unique encoding for emulator mode entry (0xD) and exit (0x7).

### 11.6.2 Concurrent BDM and Processor Operation

The debug module supports concurrent operation of both the processor and most BDM commands. BDM commands may be executed while the processor is running, except the following:

- Read/write address and data registers
- Read/write control registers

For BDM commands that access memory, the debug module requests the processor's local bus. The processor responds by stalling the instruction fetch pipeline and waiting for current bus activity to complete before freeing the local bus for the debug module to perform its access. After the debug module bus cycle, the processor reclaims the bus.

#### NOTE:

Breakpoint registers must be carefully configured in a development system if the processor is executing. The debug module contains no hardware interlocks, so TDR and XTDR should be disabled while breakpoint registers are loaded, after which TDR and XTDR can be written to define the exact trigger. This prevents spurious breakpoint triggers.

Because there are no hardware interlocks in the debug unit, no BDM operations are allowed while the CPU is writing the debug's registers (DSCLK must be inactive).

## 11.7 Debug C Definition of PSTDDATA Outputs

This section specifies the ColdFire processor and debug module's generation of the PSTDDATA output on an instruction basis. In general, the PSTDDATA output for an instruction is defined as follows:

$$\text{PSTDDATA} = 0x1, \{[0x89B], \text{operand}\}$$



where the {...} definition is optional operand information defined by the setting of the CSR.

The CSR provides capabilities to display operands based on reference type (read, write, or both). A PST value {0x8, 0x9, or 0xB} identifies the size and presence of valid data to follow on the PSTDDATA output {1, 2, or 4 bytes}. Additionally, for certain change-of-flow branch instructions, CSR[BTB] provides the capability to display the target instruction address on the PSTDDATA output {2, 3, or 4 bytes} using a PST value of {0x9, 0xA, or 0xB}.

## 11.7.1 User Instruction Set

Table 11-29 shows the PSTDDATA specification for user-mode instructions. Rn represents any {Dn, An} register. In this definition, the ‘y’ suffix generally denotes the source and ‘x’ denotes the destination operand. For a given instruction, the optional operand data is displayed only for those effective addresses referencing memory.

**Table 11-29. PSTDDATA Specification for User-Mode Instructions**

Instruction	Operand Syntax	PSTDDATA
add.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
add.l	Dy,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
adda.l	<ea>y,Ax	PSTDDATA = 0x1,{0xB, source operand}
addi.l	#<data>,Dx	PSTDDATA = 0x1
addq.l	#<data>,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
addx.l	Dy,Dx	PSTDDATA = 0x1
and.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
and.l	Dy,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
andi.l	#<data>,Dx	PSTDDATA = 0x1
asl.l	{Dy,#<data>},Dx	PSTDDATA = 0x1
asr.l	{Dy,#<data>},Dx	PSTDDATA = 0x1
bcc.{b,w,l}		if taken, then PSTDDATA = 0x5, else PSTDDATA = 0x1
bchg.{b,l}	#<data>,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bchg.{b,l}	Dy,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bclr.{b,l}	#<data>,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bclr.{b,l}	Dy,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bra.{b,w,l}		PSTDDATA = 0x5
bset.{b,l}	#<data>,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bset.{b,l}	Dy,<ea>x	PSTDDATA = 0x1,{0x8, source},{0x8, destination}
bsr.{b,w,l}		PSTDDATA = 0x5,{0xB, destination operand}
btst.{b,l}	#<data>,<ea>x	PSTDDATA = 0x1,{0x8, source operand}
btst.{b,l}	Dy,<ea>x	PSTDDATA = 0x1,{0x8, source operand}

**Table 11-29. PSTDDATA Specification for User-Mode Instructions (Continued)**

Instruction	Operand Syntax	PSTDDATA
clr.b	<ea>x	PSTDDATA = 0x1, {0x8, destination operand}
clr.l	<ea>x	PSTDDATA = 0x1, {0xB, destination operand}
clr.w	<ea>x	PSTDDATA = 0x1, {0x9, destination operand}
cmp.b	<ea>y, Dx	PSTDDATA = 0x1, {0x8, source operand}
cmp.l	<ea>y, Dx	PSTDDATA = 0x1, {0xB, source operand}
cmp.w	<ea>y, Dx	PSTDDATA = 0x1, {0x9, source operand}
cmpa.l	<ea>y, Ax	PSTDDATA = 0x1, {0xB, source operand}
cmpa.w	<ea>y, Ax	PSTDDATA = 0x1, {0x9, source operand}
cmpi.b	#<data>, Dx	PSTDDATA = 0x1
cmpi.l	#<data>, Dx	PSTDDATA = 0x1
cmpi.w	#<data>, Dx	PSTDDATA = 0x1
divs.l	<ea>y, Dx	PSTDDATA = 0x1, {0xB, source operand}
divs.w	<ea>y, Dx	PSTDDATA = 0x1, {0x9, source operand}
divu.l	<ea>y, Dx	PSTDDATA = 0x1, {0xB, source operand}
divu.w	<ea>y, Dx	PSTDDATA = 0x1, {0x9, source operand}
eor.l	Dy, <ea>x	PSTDDATA = 0x1, {0xB, source}, {0xB, destination}
eori.l	#<data>, Dx	PSTDDATA = 0x1
ext.l	Dx	PSTDDATA = 0x1
ext.w	Dx	PSTDDATA = 0x1
extb.l	Dx	PSTDDATA = 0x1
illegal		PSTDDATA = 0x1 <sup>1</sup>
jmp	<ea>y	PSTDDATA = 0x5, {[0x9AB], target address} <sup>2</sup>
jsr	<ea>y	PSTDDATA = 0x5, {[0x9AB], target address}, {0xB, destination operand} <sup>2</sup>
lea.l	<ea>y, Ax	PSTDDATA = 0x1
link.w	Ay, #<displacement>	PSTDDATA = 0x1, {0xB, destination operand}
lsl.l	{Dy, #<data>}, Dx	PSTDDATA = 0x1
lsr.l	{Dy, #<data>}, Dx	PSTDDATA = 0x1
mov3q.l	#<data>, <ea>x	PSTDDATA = 0x1, {0xB, destination operand}
move.b	<ea>y, <ea>x	PSTDDATA = 0x1, {0x8, source}, {0x8, destination}
move.l	<ea>y, <ea>x	PSTDDATA = 0x1, {0xB, source}, {0xB, destination}
move.w	<ea>y, <ea>x	PSTDDATA = 0x1, {0x9, source}, {0x9, destination}
move.w	CCR, Dx	PSTDDATA = 0x1
move.w	{Dy, #<data>}, CCR	PSTDDATA = 0x1
movea.l	<ea>y, Ax	PSTDDATA = 0x1, {0xB, source}
movea.w	<ea>y, Ax	PSTDDATA = 0x1, {0x9, source}
movem.l	#list, <ea>x	PSTDDATA = 0x1, {0xB, destination}, ... <sup>3</sup>

**Table 11-29. PSTDDATA Specification for User-Mode Instructions (Continued)**

Instruction	Operand Syntax	PSTDDATA
movem.l	<ea>y,#list	PSTDDATA = 0x1,{0xB, source},... <sup>3</sup>
moveq.l	#<data>,Dx	PSTDDATA = 0x1
muls.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
muls.w	<ea>y,Dx	PSTDDATA = 0x1,{0x9, source operand}
mulu.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
mulu.w	<ea>y,Dx	PSTDDATA = 0x1,{0x9, source operand}
mvs.b	<ea>y,Dx	PSTDDATA = 0x1, {0x8, source operand}
mvs.w	<ea>y,Dx	PSTDDATA = 0x1, {0x9, source operand}
mvz.b	<ea>y,Dx	PSTDDATA = 0x1, {0x8, source operand}
mvz.w	<ea>y,Dx	PSTDDATA = 0x1, {0x9, source operand}
neg.l	Dx	PSTDDATA = 0x1
negx.l	Dx	PSTDDATA = 0x1
nop		PSTDDATA = 0x1
not.l	Dx	PSTDDATA = 0x1
or.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
or.l	Dy,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
ori.l	#<data>,Dx	PSTDDATA = 0x1
pea.l	<ea>y	PSTDDATA = 0x1,{0xB, destination operand}
pulse		PSTDDATA = 0x4
rems.l	<ea>y,Dw:Dx	PSTDDATA = 0x1,{0xB, source operand}
remu.l	<ea>y,Dw:Dx	PSTDDATA = 0x1,{0xB, source operand}
rts		PSTDDATA = 0x1, PSTDDATA = 0x5, {[0x9AB], target address}
sats.l	Dx	PSTDDATA = 0x1
scc.b	Dx	PSTDDATA = 0x1
sub.l	<ea>y,Dx	PSTDDATA = 0x1,{0xB, source operand}
sub.l	Dy,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
suba.l	<ea>y,Ax	PSTDDATA = 0x1,{0xB, source operand}
subi.l	#<data>,Dx	PSTDDATA = 0x1
subq.l	#<data>,<ea>x	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
subx.l	Dy,Dx	PSTDDATA = 0x1
swap.w	Dx	PSTDDATA = 0x1
tas.b	<ea>x	PSTDDATA = 0x1, {0x8, source}, {0x8, destination}
tpf		PST = 0x1
tpf.l	#<data>	PST = 0x1
tpf.w	#<data>	PST = 0x1
trap	#<data>	PSTDDATA = 0x1 <sup>1</sup>

**Table 11-29. PSTDDATA Specification for User-Mode Instructions (Continued)**

Instruction	Operand Syntax	PSTDDATA
tst.b	<ea>x	PSTDDATA = 0x1,{0x8, source operand}
tst.l	<ea>y	PSTDDATA = 0x1,{0xB, source operand}
tst.w	<ea>y	PSTDDATA = 0x1,{0x9, source operand}
unlk	Ax	PSTDDATA = 0x1,{0xB, destination operand}
wddata.b	<ea>y	PSTDDATA = 0x4, {0x8, source operand}
wddata.l	<ea>y	PSTDDATA = 0x4, {0xB, source operand}
wddata.w	<ea>y	PSTDDATA = 0x4, {0x9, source operand}

<sup>1</sup> During normal exception processing, the PSTDDATA output is driven to a 0xC indicating the exception processing state. The exception stack write operands, as well as the vector read and target address of the exception handler may also be displayed.

```
Exception Processing  PSTDDATA = 0xC,    {0xB,destination}, // stack frame
                                     {0xB,destination}, // stack frame
                                     {0xB,source},      // vector read
                                     PSTDDATA = 0x5,    {[0x9AB],target} // handler PC
```

The PSTDDATA specification for the reset exception is shown below:

```
Exception Processing  PSTDDATA = 0xC,
                     PSTDDATA = 0x5,    {[0x9AB],target} // handler PC
```

The initial references at address 0 and 4 are never captured nor displayed since these accesses are treated as instruction fetches.

For all types of exception processing, the PSTDDATA = 0xC value is driven at all times, unless the PSTDDATA output is needed for one of the optional marker values or for the taken branch indicator (0x5).

- <sup>2</sup> For JMP and JSR instructions, the optional target instruction address is displayed only for those effective address fields defining variant addressing modes. This includes the following <ea>x values: (An), (d16,An), (d8,An,Xi), (d8,PC,Xi).
- <sup>3</sup> For Move Multiple instructions (MOVEM), the processor automatically generates line-sized transfers if the operand address reaches a 0-modulo-16 boundary and there are four or more registers to be transferred. For these line-sized transfers, the operand data is never captured nor displayed, regardless of the CSR value.
- The automatic line-sized burst transfers are provided to maximize performance during these sequential memory access operations.

Table 11-30 shows the PSTDDATA specification for multiply-accumulate instructions.

**Table 11-30. PSTDDATA Values for User-Mode Multiply-Accumulate Instructions**

Instruction	Operand Syntax	PSTDDATA
mac.l	Ry,Rx	PSTDDATA = 0x1
mac.l	Ry,Rx,<ea>y,Rw,ACCx	PSTDDATA = 0x1,{0xB, source operand}
mac.l	Ry,Rx,ACCx	PSTDDATA = 0x1
mac.l	Ry,Rx,ea,Rw	PSTDDATA = 0x1,{0xB, source operand}
mac.w	Ry,Rx	PSTDDATA = 0x1
mac.w	Ry,Rx,<ea>y,Rw,ACCx	PSTDDATA = 0x1,{0xB, source operand}
mac.w	Ry,Rx,ACCx	PSTDDATA = 0x1
mac.w	Ry,Rx,ea,Rw	PSTDDATA = 0x1,{0xB, source operand}

**Table 11-30. PSTDDATA Values for User-Mode Multiply-Accumulate Instructions**

Instruction	Operand Syntax	PSTDDATA
move.l	{Ry,#<data>},ACCext01	PSTDDATA = 0x1
move.l	{Ry,#<data>},ACCext23	PSTDDATA = 0x1
move.l	{Ry,#<data>},ACCx	PSTDDATA = 0x1
move.l	{Ry,#<data>},MACSR	PSTDDATA = 0x1
move.l	{Ry,#<data>},MASK	PSTDDATA = 0x1
move.l	ACCext01,Rx	PSTDDATA = 0x1
move.l	ACCext23,Rx	PSTDDATA = 0x1
move.l	ACCy,ACCx	PSTDDATA = 0x1
move.l	ACCy,Rx	PSTDDATA = 0x1
move.l	MACSR,CCR	PSTDDATA = 0x1
move.l	MACSR,Rx	PSTDDATA = 0x1
move.l	MASK,Rx	PSTDDATA = 0x1
msac.l	Ry,Rx	PSTDDATA = 0x1
msac.l	Ry,Rx,<ea>y,Rw,ACCx	PSTDDATA = 0x1,{0xB, source operand}
msac.l	Ry,Rx,ACCx	PSTDDATA = 0x1
msac.l	Ry,Rx,<ea>y,Rw	PSTDDATA = 0x1,{0xB, source},{0xB, destination}
msac.w	Ry,Rx	PSTDDATA = 0x1
msac.w	Ry,Rx,<ea>y,Rw,ACCx	PSTDDATA = 0x1,{0xB, source operand}
msac.w	Ry,Rx,ACCx	PSTDDATA = 0x1
msac.w	Ry,Rx,<ea>y,Rw	PSTDDATA = 0x1,{0xB, source},{0xB, destination}

Table 11-31 shows the PSTDDATA specification for floating-point instructions; note that <ea>y includes FPy, Dy, Ay, and <mem>y addressing modes. The optional operand capture and display applies only to the <mem>y addressing modes. Note also that the PSTDDATA values are the same for a given instruction, regardless of explicit rounding precision.

**Table 11-31. PSTDDATA Values for User-Mode Floating-Point Instructions**

Instruction <sup>1</sup>	Operand Syntax	PSTDDATA
fabs.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fadd.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fbcc.{w,l}	<label>	if taken, then PSTDDATA = 5, else PSTDDATA = 0x1
fcmp.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fdiv.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fint.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fintr.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}

**Table 11-31. PSTDDATA Values for User-Mode Floating-Point Instructions**

Instruction <sup>1</sup>	Operand Syntax	PSTDDATA
fmove.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fmove.sz	FPy,<ea>x	PSTDDATA = 0x1, [89B], destination}
fmove.l	<ea>y,FP*R	PSTDDATA = 0x1, B, source}
fmove.l	FP*R,<ea>x	PSTDDATA = 0x1, B, destination}
fmovem	<ea>y,#list	PSTDDATA = 0x1
fmovem	#list,<ea>x	PSTDDATA = 0x1
fmul.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fneg.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fnop		PSTDDATA = 0x1
fsqrt.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
fsub.sz	<ea>y,FPx	PSTDDATA = 0x1, [89B], source}
ftst.sz	<ea>y	PSTDDATA = 0x1, [89B], source}

<sup>1</sup> The FP\*R notation refers to the floating-point control registers: FPCR, FPSR, and FPIAR.

Depending on the size of any external memory operand specified by the f<op>.fmt field, the data marker is defined as shown in Table 11-32.

**Table 11-32. Data Markers and FPU Operand Format Specifiers**

Format Specifier	Data Marker
.b	8
.w	9
.l	B
.s	B
.d	Never captured

## 11.7.2 Supervisor Instruction Set

The supervisor instruction set has complete access to the user mode instructions plus the opcodes shown below. The PSTDDATA specification for these opcodes is shown in Table 11-33.

**Table 11-33. PSTDDATA Specification for Supervisor-Mode Instructions**

Instruction	Operand Syntax	PSTDDATA
cpushl	dc,(Ax) ic,(Ax) bc,(Ax)	PSTDDATA = 0x1
frestore	<ea>y	PSTDDATA = 0x1
fsave	<ea>x	PSTDDATA = 0x1

**Table 11-33. PSTDDATA Specification for Supervisor-Mode Instructions**

Instruction	Operand Syntax	PSTDDATA
halt		PSTDDATA = 0x1, PSTDDATA = 0xF
intouch	(Ay)	PSTDDATA = 0x1
move.l	Ay,USP	PSTDDATA = 0x1
move.l	USP,Ax	PSTDDATA = 0x1
move.w	SR,Dx	PSTDDATA = 0x1
move.w	{Dy,#<data>},SR	PSTDDATA = 0x1, {0x3}
movec.l	Ry,Rc	PSTDDATA = 0x1, {8, ASID}
rte		PSTDDATA = 0x7, {0xB, source operand}, {3},{0xB, source operand}, {DD}, PSTDDATA = 0x5, {[0x9AB], target address}
stop	#<data>	PSTDDATA = 0x1, PSTDDATA = 0xE
wdebug.l	<ea>y	PSTDDATA = 0x1, {0xB, source, 0xB, source}

The move-to-SR and RTE instructions include an optional PSTDDATA = 0x3 value, indicating an entry into user mode. Additionally, if the execution of a RTE instruction returns the processor to emulator mode, a multiple-cycle status of 0xD is signaled.

Similar to the exception processing mode, the stopped state (PSTDDATA = 0xE) and the halted state (PSTDDATA = 0xF) display this status throughout the entire time the ColdFire processor is in the given mode.

## 11.8 ColdFire Debug History

This section describes the origins of the ColdFire debug systems.

### 11.8.1 ColdFire Debug Classic: The Original Definition

The original design, Revision A, provided debug support in three separate areas:

- Real-time trace.
- Background debug mode (BDM)
- Real-time debug

The real-time debug features may be accessed from the external BDM emulator or from the supervisor programming model of the processor. The hardware breakpoint registers include: a PC breakpoint + mask, two address registers for defining a specific address or a range of addresses, and a data breakpoint + mask. The original design supported breakpoints of the form:

```
if PC_breakpoint is triggered
    then respond using user-defined configuration
```

## ColdFire Debug History

```
if Address_breakpoint {&& Data_breakpoint} is triggered
    then respond using user-defined configuration
```

Two-level triggers of the form:

```
if PC_breakpoint is triggered
    then if Address_breakpoint {&& Data_breakpoint} is triggered
        then respond using user-defined configuration

if Address_breakpoint {&& Data_breakpoint} is triggered
    then if PC_breakpoint is triggered
        then respond using user-defined configuration
```

The data\_breakpoint can be included as an optional part of an address breakpoint.

The ColdFire debug architecture was created to provide this set of functionality *without* requiring the traditional connection to the external system bus. Rather, the functionality is provided using only a connection to a Motorola-defined 26-pin debug connector. By providing the required debug signals in customer-specific designs, standard third-party emulators can be used for debug of these designs.

### NOTE:

The baseline debug functionality is described in any of the *ColdFire MCF52xx User's Manuals*, which are available as PDF files at: <http://www.motorola.com/ColdFire/>. As an example, see the debug section of the *MCF5272 User's Manual* located under MCF5272 Product Information.

Implementation of the original debug module produced design requiring approximately 34,000 transistors, 22,500 for the BDM/real-time debug function, 7,500 for the DDATA module, and 4,000 for the PC breakpoint logic in the processor.

## 11.8.2 ColdFire Debug Revision B

During development of the Version 3 ColdFire design, there were a number of enhancements to the original debug functionality requested by customers and third-party developers. These requests resulted in an expanded set of debug functionality named Revision B.

The Rev. B enhancements are as follows:

- Addition of a BDM SYNC\_PC command to display the processor's current PC
- Creation of more flexible hardware breakpoint triggers, i.e., support for "OR" combinations
- Removal of the restrictions involving concurrent hardware breakpoint use and BDM command activity
- Redefinition of the processor status values for the RTS instruction



- An external mechanism to generate a debug interrupt
- A mechanism to inhibit debug interrupts after the RTE exit
- A mechanism to identify the revision level of the debug module

Rev. B enhancements provide backward compatibility with the original design.

### 11.8.3 ColdFire Debug Revision C

Continuing discussions with customers and the developer community led to Revision C design enhancements primarily related to improvements in the real-time debug capabilities of the ColdFire architecture. The remainder of this section details these enhancements.

#### 11.8.3.1 Debug Interrupts and Interrupt Requests (Emulator Mode)

In Rev. A and Rev. B ColdFire debug implementations, the response to a user-defined breakpoint trigger can be configured to be one of three possibilities:

- The breakpoint trigger can merely be displayed on the DDATA bus, with no internal reaction to the trigger. The trigger state information is displayed on DDATA in all situations.
- The breakpoint trigger can force the processor to halt and allow BDM activities.
- The breakpoint trigger can generate a special debug interrupt to allow real-time systems to quickly process the interrupt and return to normal system executing as rapidly as possible.

The occurrence of the debug interrupt exception is treated as a special type of interrupt. It is considered to be higher in priority than all normal interrupt requests and has special processor status values to provide an external indication that this interrupt has occurred.

Additionally, the execution of the debug interrupt service routine is forced to be interrupt-inhibited by the processor hardware with an optional capability to map all instruction and operand references while in this service routine into a separate address space so that an emulator could define the routine dynamically. The current processor implementations actually include a program-invisible state bit which defines this emulator mode of operation. Also note, the interrupt mask level is not modified during the processing of a debug interrupt.

Customers with real-time embedded systems have specifically asked for the ability to service normal interrupt requests while processing the debug interrupt service routine. In many systems of this type, motion-based servo interrupts must be considered as the highest priority interrupt request.

To provide this functionality and be able to service any number of normal interrupt requests (including the possibility of nested interrupts), the processor state signaling emulator mode must be included as part of the exception stack frame.

As part of the Rev. C functionality, the operation of the debug interrupt is modified in the following manner:

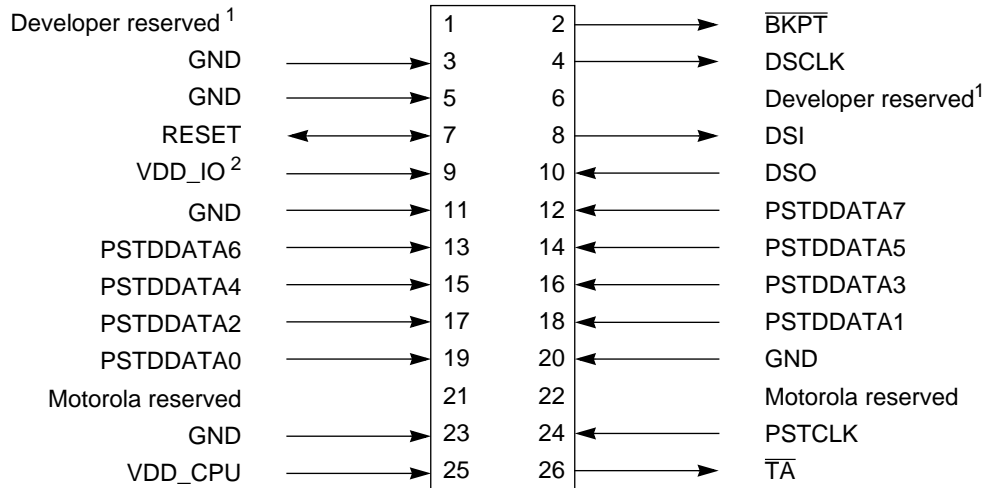
1. The occurrence of the breakpoint trigger, configured to generate a debug interrupt, is treated exactly as before. The debug interrupt is treated as a higher priority exception relative to the normal interrupt requests encoded on the interrupt priority input signals.
2. At the appropriate sample point, the ColdFire processor initiates debug interrupt exception processing. This event is signaled externally by the generation of a unique PST value ( $PST = 0xD$ ) asserted for multiple cycles. The processor sets the emulator mode state bit as part of this exception processing.
3. While the processor in the debug interrupt service routine, all normal interrupt requests are evaluated and sampled once per instruction. While in this routine, if any type of exception occurs, the processor responds in the following manner:
  - a) In response to the new exception, the processor saves a copy of the current value of the emulator mode state bit and then exits emulator mode by clearing the actual state.
  - b) The new exception stack frame sets bit 1 of the fault status field, using the saved emulator mode bit, indicating execution while in emulator mode has been interrupted. This corresponds to bit 17 of the longword at the top of the system stack.
  - c) Control is passed to the appropriate exception handler.
  - d) When the exception handler is complete, a Return From Exception (RTE) instruction is executed. During the processing of the RTE,  $FS[1]$  is reloaded from the system stack. If this bit is asserted, the processor sets the emulator mode state and resumes execution of the original debug interrupt service routine. This is signaled externally by the generation of the PST value that originally identified the occurrence of a debug interrupt exception, that is,  $PST = 0xD$ .

Implementation of this revised debug interrupt handling fully supports the servicing of any number of normal interrupt requests while in a debug interrupt service routine. The emulator mode state bit is essentially changed to be a program-visible value, stored into memory during exception stack frame creation and loaded from memory by the RTE instruction.

## 11.9 Motorola-Recommended BDM Pinout

The ColdFire BDM connector, Figure 11-1, is a 26-pin Berg connector arranged 2 x 13.

## Motorola-Recommended BDM Pinout



<sup>1</sup> Pins reserved for BDM developer use.

<sup>2</sup> Supplied by target.

**Figure 11-1. Recommended BDM Connector**



# Chapter 12

## Test

This chapter provides an overview of test features of CF4e. Some of the features, such as MBist hardware, are included in the CF4e design. The scan and wrapper methodology, described later in the chapter, are part of the CF4e design but are described here as a reference for properly designing CF4e for test.

Because it is less accessible, an embedded core device is more difficult to test. The solution to applying the test to an embedded core should not be less efficient, should not lessen the quality, and should not significantly increase integration costs by the need for extra signals, routes, and logic. High quality levels and efficient test vectors can result from using the proper mix of test techniques to address the embedded market.

Testing the structure of general combinational and sequential logic in the core is typically done by application of vectors measured against the stuck-at fault model. The test vectors applied may be functional or scan based; however, full-scan testing is the most efficient test architecture for generating and applying stuck-at vectors. A further optimization to a full-scan test architecture that allows the reduction of the shift cost (number of clock cycles required to load in a state) associated with the scan architecture is the support of a parallel-pin architecture with multiple, simultaneously operational scan chains.

Testing memory arrays in an embedded core is most efficiently done with a memory built-in-self test (MBIST) architecture. One major goal is to reduce the overall test time by testing each embedded memory simultaneously with at-speed data transfers, reads, and writes. Another goal is to reduce the signal interface involved in testing multiple embedded memories by requiring only the invoke, done, and fail indicators (as a minimum).

All logic internal to the boundary of an embedded core can be tested efficiently with support from test selection, scan, and MBIST architectures. These architectures, in conjunction with tester pauses and current measurement techniques, allow all test and DFT goals to be met. In addition, the optional test wrapper scan architecture at the embedded core hierarchy boundary permits testing beyond the embedded core.

The embedded CF4e is designed to be tested independently of the rest of the chip in all test modes. It also allows noncore logic to be tested up to the core interface. No reliance on internal core logic or specific core test modes should be required to test non-core logic. Considerations are taken to ensure that the CF4e can be put in the functional (nontest) mode with no residual effect or interference from the test logic.

## Scan Chains

We recommend using full-scan, multiplexed, D flip-flop methodology on the CF4e implementation.

The CF4e core has one clock domain. The core has an optional test boundary (or test wrapper) comprised of wrapper cells used to access all functional input and output ports.

The core has a BIST controller to BIST-test memory arrays attached to the core. The processor-local memories are external to the CF4e design to allow the system designer to configure and size those memories for a given application.

## 12.1 Scan Chains

This section describes the core and wrapper scan chains.

### 12.1.1 Core Scan Chains

Customers can choose the number of scan chains in the CF4e. Table 12-1 describes the test ports of these scan chains.

**Table 12-1. CF4e Core Scan Chains**

Scan Inputs	Scan Outputs	Scan Enable	Clock
si[N-1:0] <sup>1</sup>	so[N-1:0] <sup>1</sup>	se	clkfast

<sup>1</sup> N represents the number of core scan chains

### 12.1.2 Wrapper Scan Chains

Because scan is part of the implementation phase, the soft CF4e does not contain the test wrapper in its RTL code when it is delivered to customers. However, a test wrapper can be optionally created using synthesis scripts. Table 12.2 gives a more detailed description of the test wrapper. If the test wrapper is used, customers can choose the number of wrapper scan chains. An input wrapper chain has cells connected only to functional input ports of the core. An output wrapper chain has cells connected only to functional output ports of the core. The CF4e has no I/O or three-state ports. There are no wrapper cells on the clock (clkfast), memory, scan input, or scan output ports of the cores. Table 12-2 describes the test ports of the wrapper scan chains.

**Table 12-2. CF4e Wrapper Scan Chains**

Signal Types	Port Names
Wrapper scan data inputs	tbsi[K-1:0] <sup>1</sup>
Wrapper scan data outputs	tbso[K-1:0] <sup>1</sup>
Input wrapper scan enable	tbsei
Output wrapper scan enable	tbseo
Clock for wrapper scan chains	clkfast

<sup>1</sup> K: Number of wrapper scan chains

### 12.1.3 Scan Chains Block Diagram

For a multiplexed D flip-flop scan chain DFT methodology, the core chains and the wrapper chains as shown in Figure 12-1.

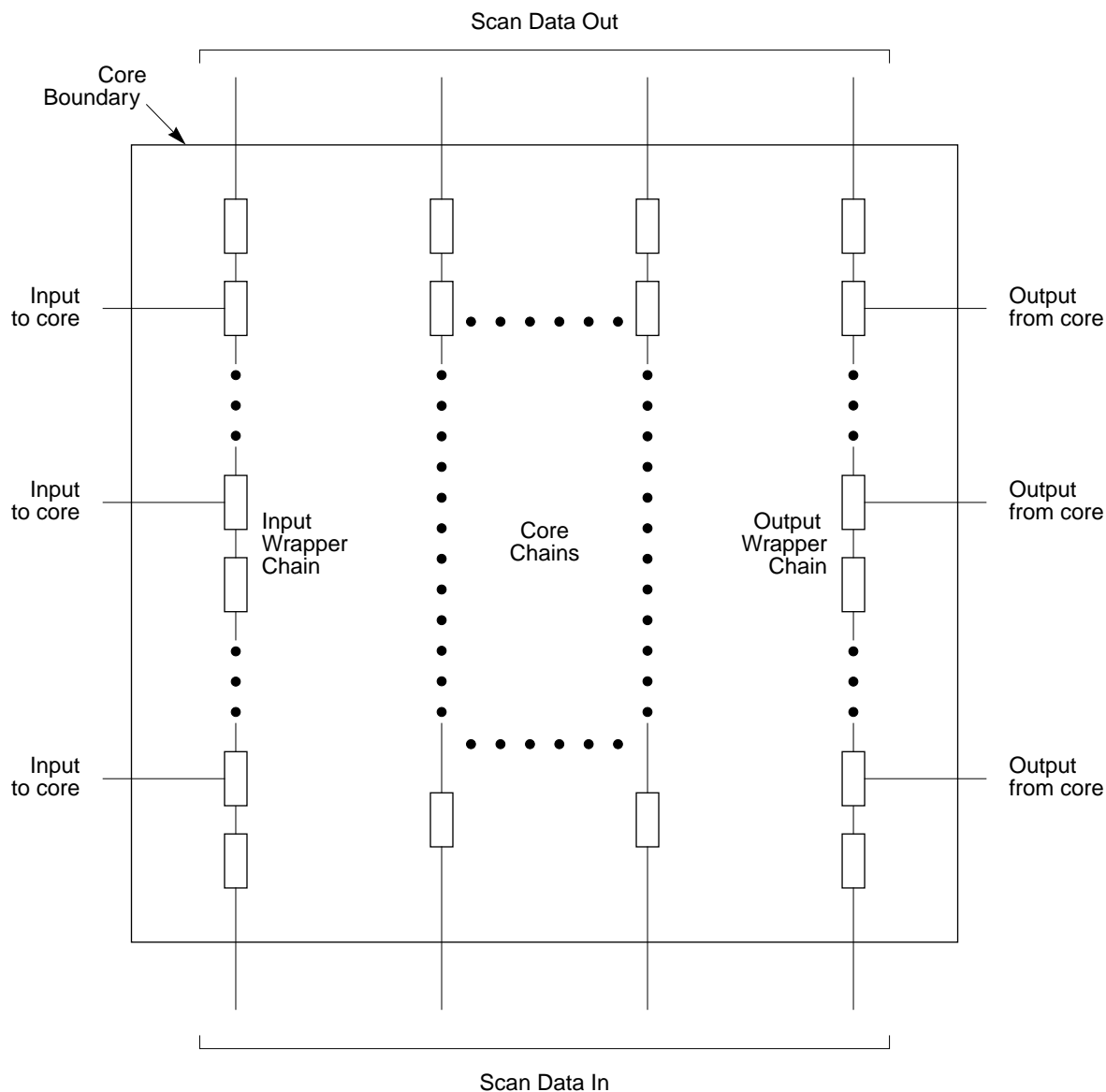


Figure 12-1. CF4e Scan Chains Block Diagram

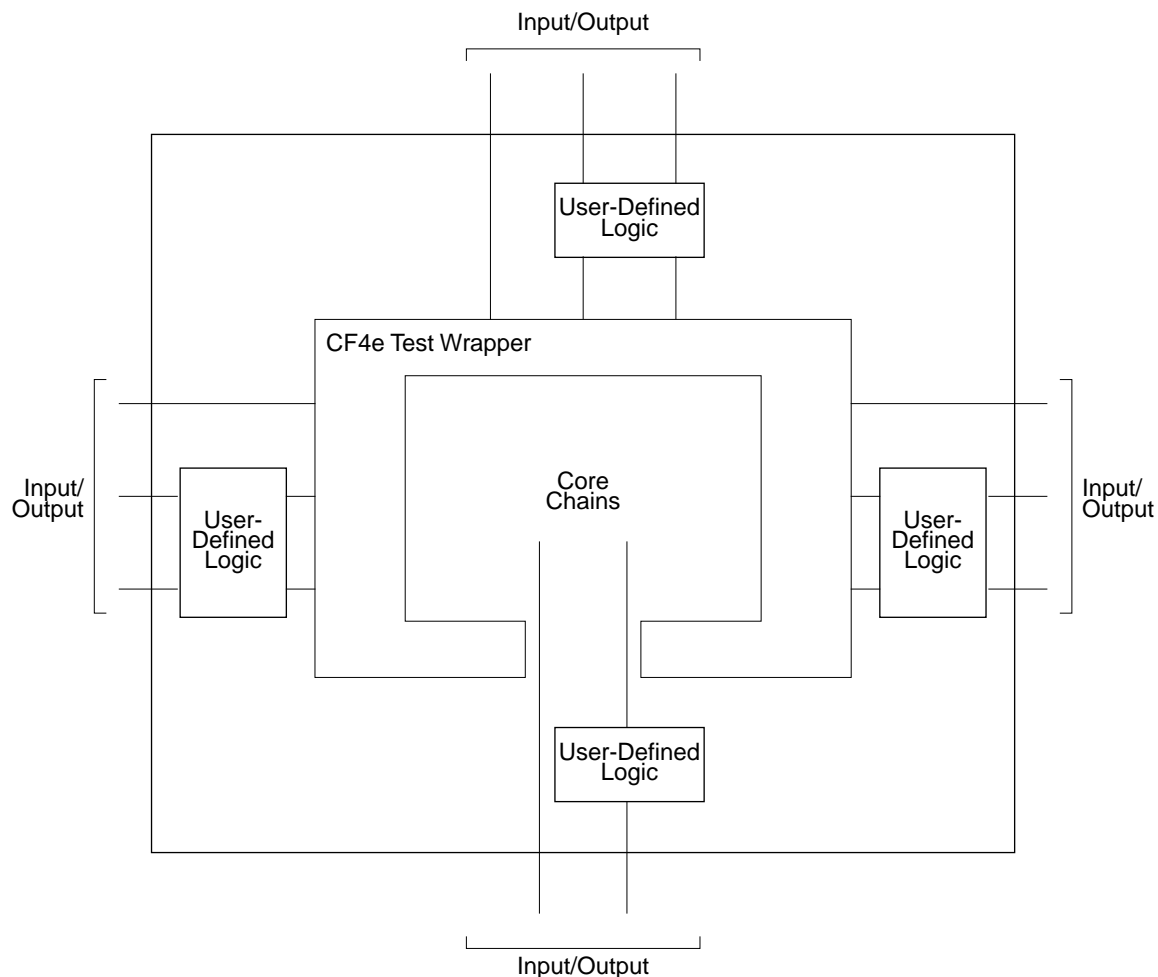
## 12.2 Test Wrapper

As described in Section 12.1.2, “Wrapper Scan Chains,” because scan is part of the implementation phase, the soft CF4e does not contain the test wrapper in its RTL code when it is delivered to customers. However, a test wrapper can be optionally created in later

process during synthesis using synthesis scripts. This section provides information on the test wrapper (abbreviated as CF4eTW), which is implemented using multiplexed, D flip-flop DFT methodology and the shared wrapper type.

### 12.2.1 Features

The test wrapper, shown in Figure 12-2, allows the CF4e to be tested independently of the rest of the system-on-a-chip (SoC) and allows noncore logic to be tested up to the CF4e interface. An SoC includes standalone virtual components, integration logic, and application platforms. SoC can go to tape out and run and become final product. The CF4e is used as an embedded core in an SoC environment.



**Figure 12-2. CF4e and Test Wrapper in SoC**

The test wrapper provides the following:

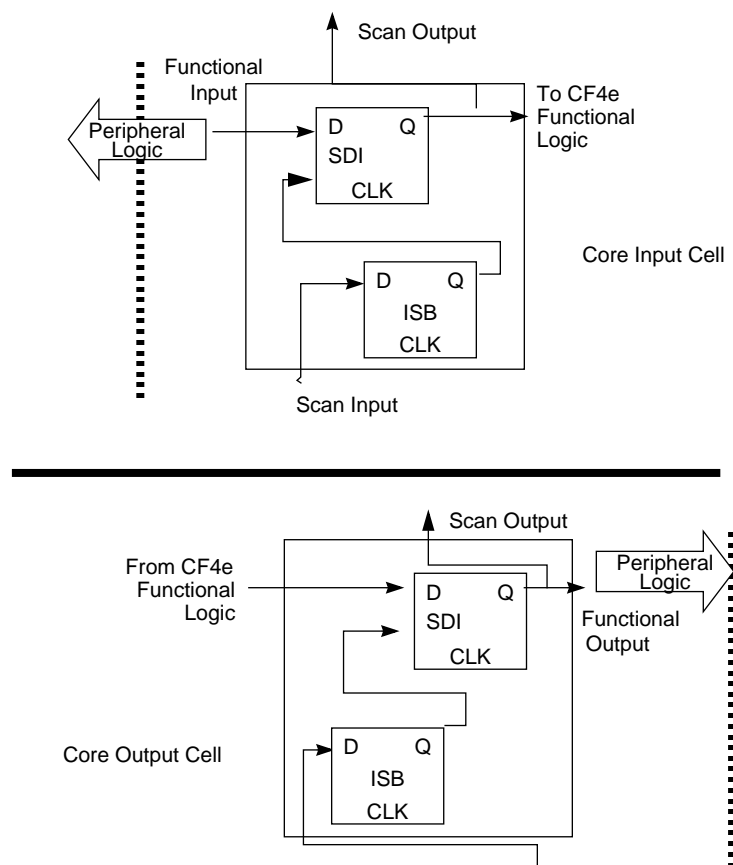
- Support for reapplying original test vectors when the target CF4e is embedded into an SoC
- Ability to control and observe the CF4e port interface boundary



- Pattern generation for the interface between the CF4e within an SoC without knowledge of the block functionality
- Support for AC testing of the interface to and from the SoC interface logic (at-speed transitions to be launched or captured by the test wrapper)
- Safe state to the non-core logic while the CF4e is under test, and vice versa
- Transparency when the test wrapper is not used. The test wrapper is logically removed during functional mode.

## 12.2.2 Wrapper Cells

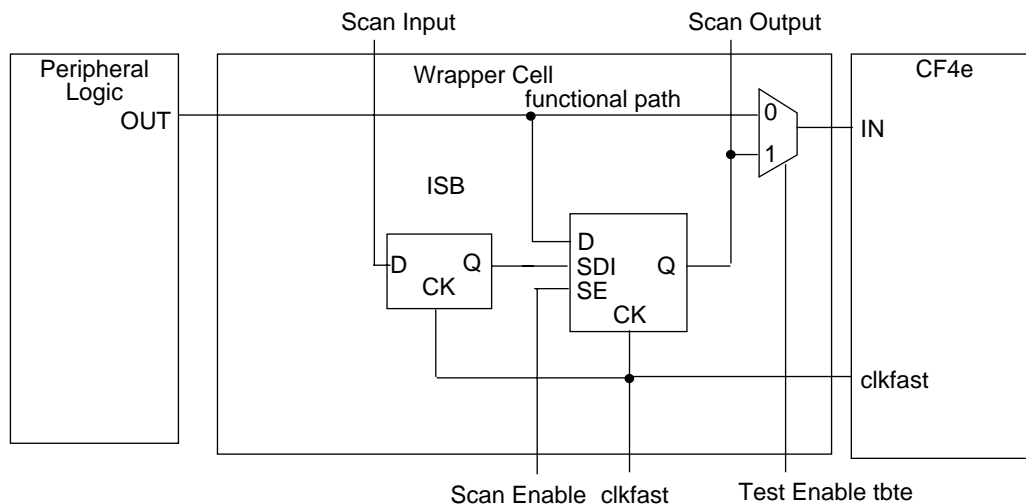
If a wrapper is created using multiplexed, D flip-flop DFT methodology, its scan chains are mainly composed of shared wrapper cells (see Figure 12-3). A shared wrapper cell contains a functional register and a D flip-flop called an independent shift bit (ISB) that allows a second bit of data to be shifted into the core sequentially to perform path delay testing on the core input and output paths. A shared wrapper cell can be used only on a registered input or output. Nonregistered inputs or outputs must use partition cell (p-cell).



**Figure 12-3. CF4e Core Shared Wrapper Cells**

There are two nonregistered inputs on the CF4e core. These inputs require dedicated wrapper partition cells (P cells) to control the two (see Figure 12-4). These P cells are

included in the input wrapper scan chain of the CF4e.



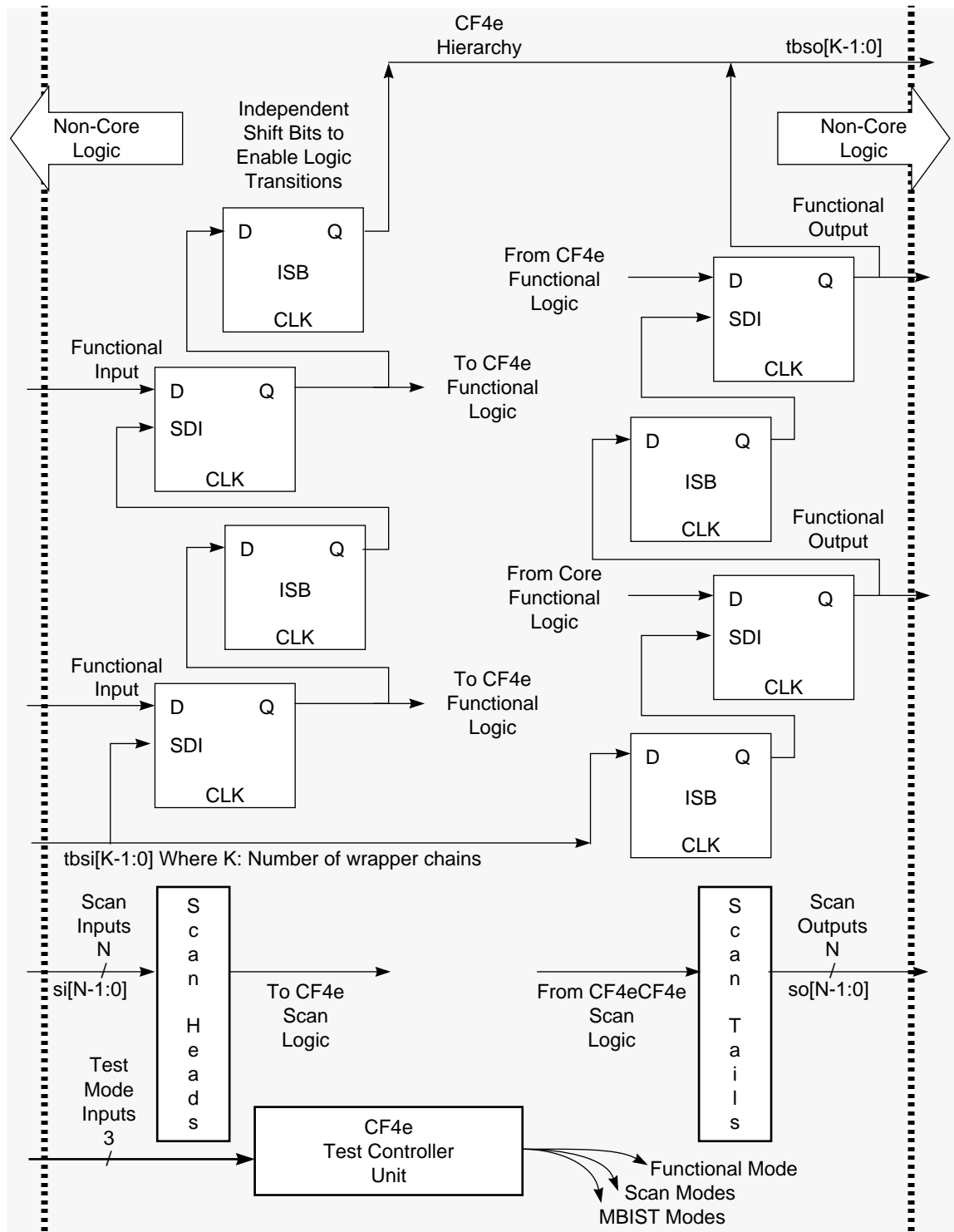
**Figure 12-4. CF4e Core Dedicated Input Wrapper Cell (P Cell)**

The two P cells in the CF4e core wrapper have a functional path through the multiplexer. The logic on either side of this wrapper cell is testable while the test enable is asserted. However, to test the wire and possibly a critical timing path, the core and peripheral scan chains must be loaded with the test enable (tbte) set to transparent mode (active low).

A synthesis script adds wrapper logic during synthesis. Chains must be evaluated for correctness after script use.

### 12.2.3 Block Diagram

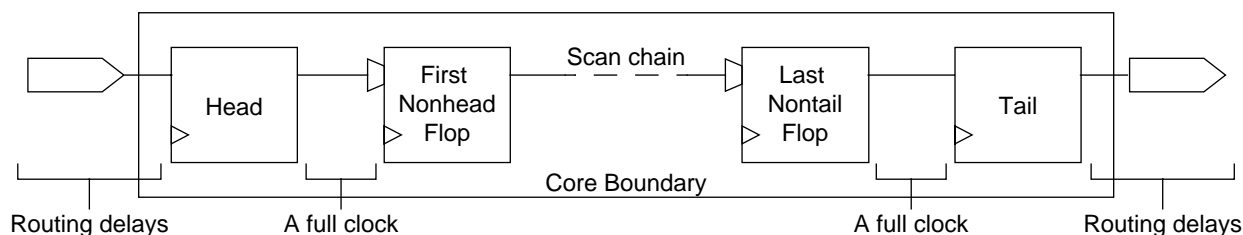
Figure 12-5 shows a shared wrapper architecture. This design has K wrapper scan chains and N general core scan chains.



**Figure 12-5. Example of Registered CF4eTW Architecture**

The first and last registers in a scan chain (called the head and tail registers) are used in each scan chain only for scan shift operations and have no functional use. They ensure a full clock cycle to propagate data into the first nonhead flop and from the last nontail flop of the scan chains. They also eliminate the need to consider routing delays from the CF4e

boundary to the first scan-in flop and from the last scan-out flop.



**Figure 12-6. Scans and Flops**

The ISB is described in Section 12.2.2, “Wrapper Cells.”

## 12.2.4 Timing

When it is embedded and integrated within a chip, the CF4eTW scan architecture tests the following:

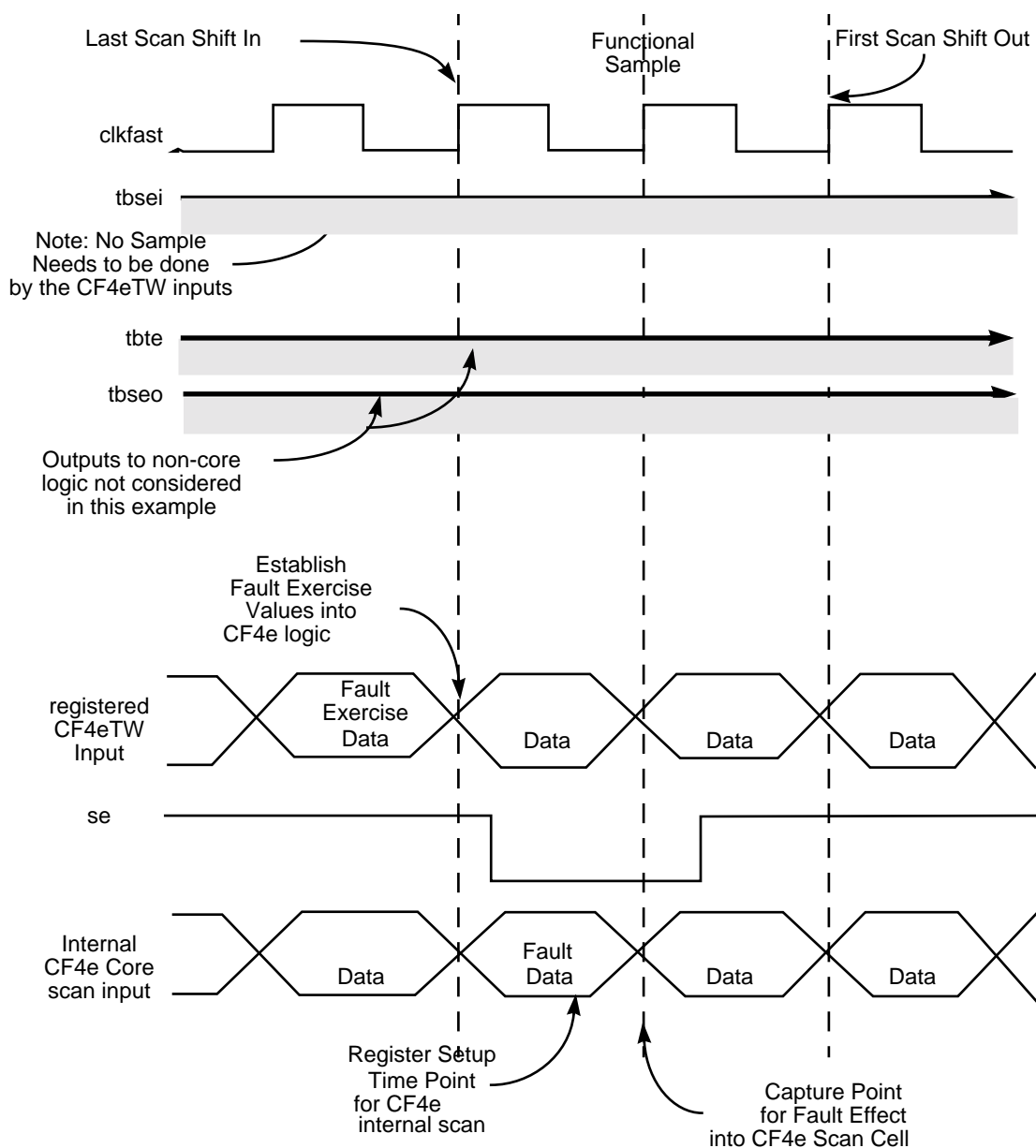
- CF4e inputs for structure and timing
- CF4e outputs for structure and timing
- The interface between the CF4e output signals and the non-core logic input signals for structure and timing
- The interface between the non-core logic output signals and the CF4e input signals for structure (and possibly timing)

These operations are described in the following sections.

### 12.2.4.1 CF4eTW Testing of CF4e Core Inputs

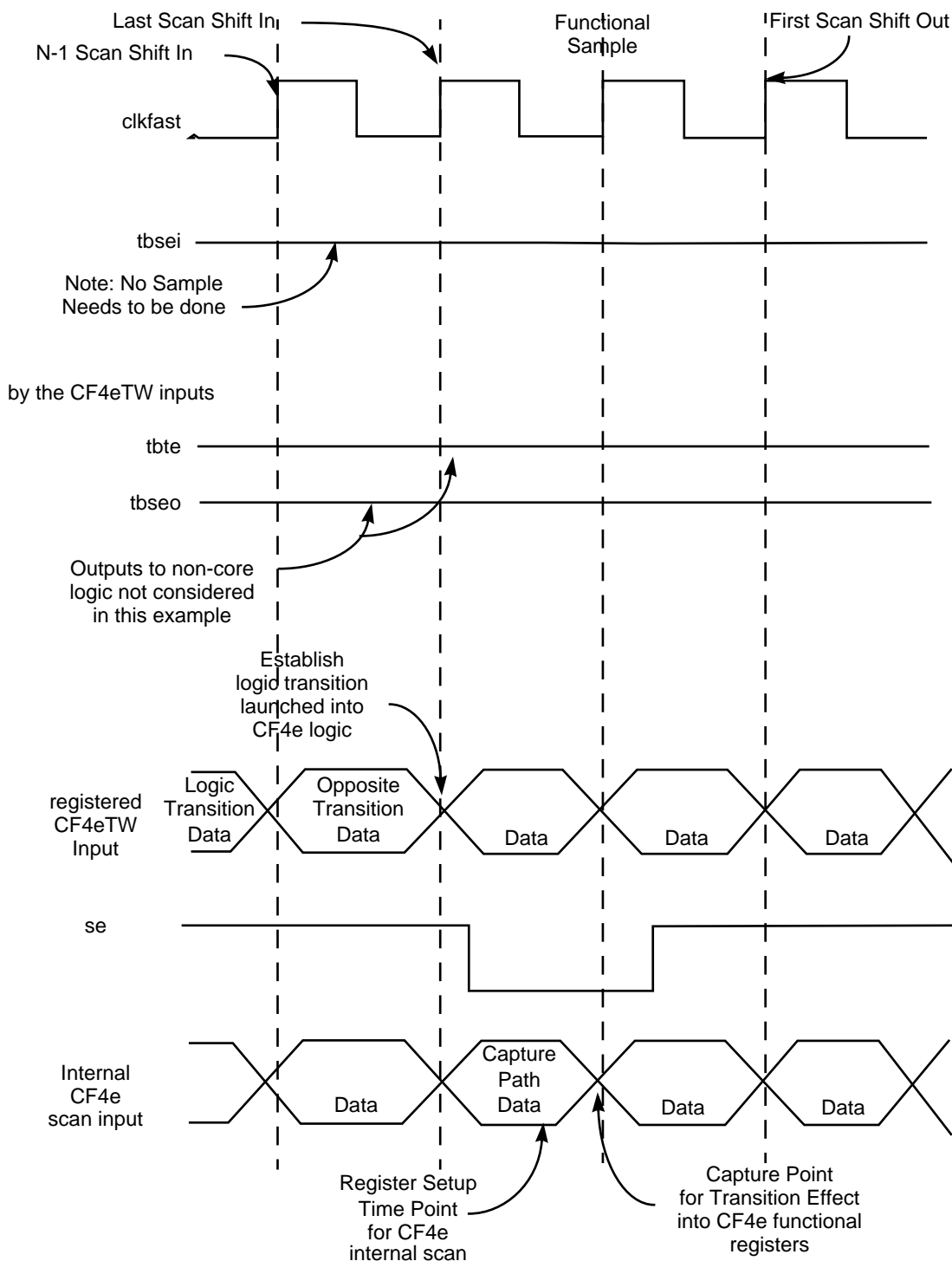
CF4eTW testing of CF4e inputs is a manufacturing test operation. Verification and testing of CF4e inputs for structure and timing is done by applying vectors through the CF4eTW scan architecture. Testing is accomplished by launching logic values into the CF4e from the functional input registers in the CF4eTW scan chain.

The CF4e internal parallel scan chains must capture these launched values. This confirms that the functional register operates and that connections from the functional register into the CF4e are correct. If logic values launched into the core are configured as a vector pair with logic transitions, the logic paths from the interface register into the CF4e are also verified for timing with reference to the clock cycle. Figure 12-7 describes timing diagram for an input wrapper to CF4e core scan stuck-at vector.



**Figure 12-7. CF4eTW Input to CF4e Core Scan Stuck-At Vector Example**

Figure 12-8 describes timing diagram for an input wrapper to CF4e core scan delay vector.



**Figure 12-8. CF4eTW Input to CF4e Core Scan Delay Vector Example**

Delay testing first identifies a target path (usually with static timing analysis). It then creates a vector that shifts a logic value into the functional register in the n-1 shift (next to last shift) and then shifts in the opposite logic value into the functional register as the last shift. This launches a transition into the CF4e core. Because the CF4e core and test wrapper share the same system clock, the next system clock rising-edge after the last shift is the sample cycle,

so the internal CF4e core scan enable signal must be negated to allow the CF4e core internal scan architecture to capture the effect of the launched transition.

Note that the CF4eTW input side scan enable does not need to negate because this test is to verify from the inputs into the core. Sampling done by input registers of the CF4e test wrapper would verify outputs of the noncore logic. Similarly, for this example, the output scan enable, *tbseo*, does not need to be negated (in reality, however, stuck-at testing of the CF4e core uses the CF4eTW input registers and the CF4eTW output registers simultaneously: path delay may be done individually).

When a hard-layout core is delivered, understanding this operation is unnecessary because vectors exist to accomplish these operations (for example, the CF4e core manufacturing test program).

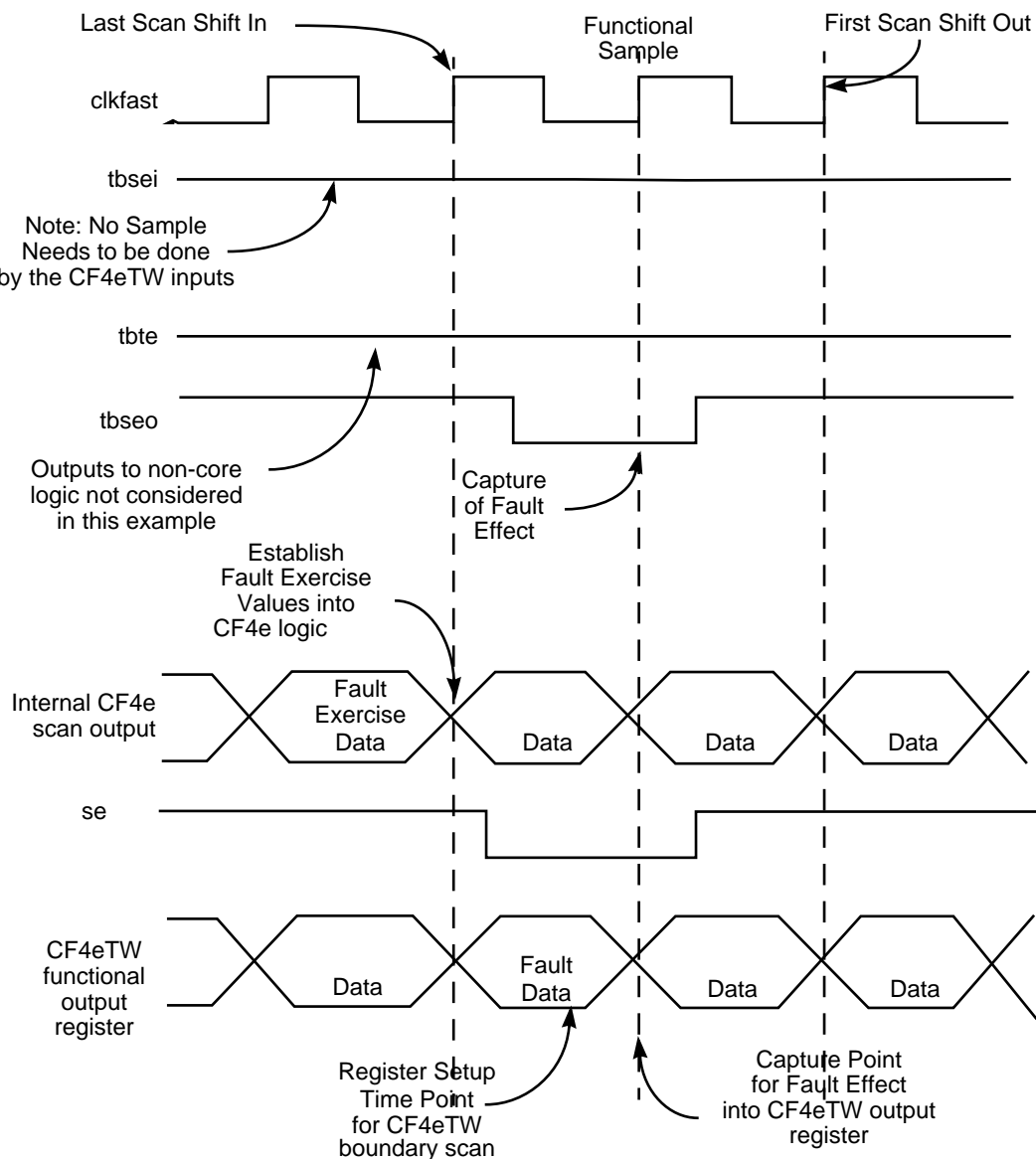
### 12.2.4.2 CF4eTW Testing of CF4e Core Outputs

CF4eTW testing of CF4e core outputs is also considered a manufacturing test operation of the CF4e core. Verification and testing of CF4e core outputs for structure and timing is the application of vectors through the CF4e core internal parallel scan architecture to be captured by the CF4eTW scan architecture. Testing is done by launching logic values from the CF4e core internal parallel scan registers to be captured by the functional output registers included in the CF4eTW scan chain.

Logic values launched by the CF4e core scan chains must be captured by the CF4eTW scan chains. This verifies that the CF4eTW functional output register operates and that connections from the CF4e core internal functional registers are correct. If logic values launched from the CF4e core are configured as a vector pair with logic transitions, logic paths from the CF4e core internals to the CF4eTW interface register are also verified for timing with reference to the clock cycle.

Delay testing is done by first identifying a target path (usually with static timing analysis) and then creating a vector that shifts a logic value into the internal functional register as the last shift. The CF4e core scan enable is then negated, and the next system clock conducts a functional state transition, which launches a logic transition into the CF4e core along the identified path.

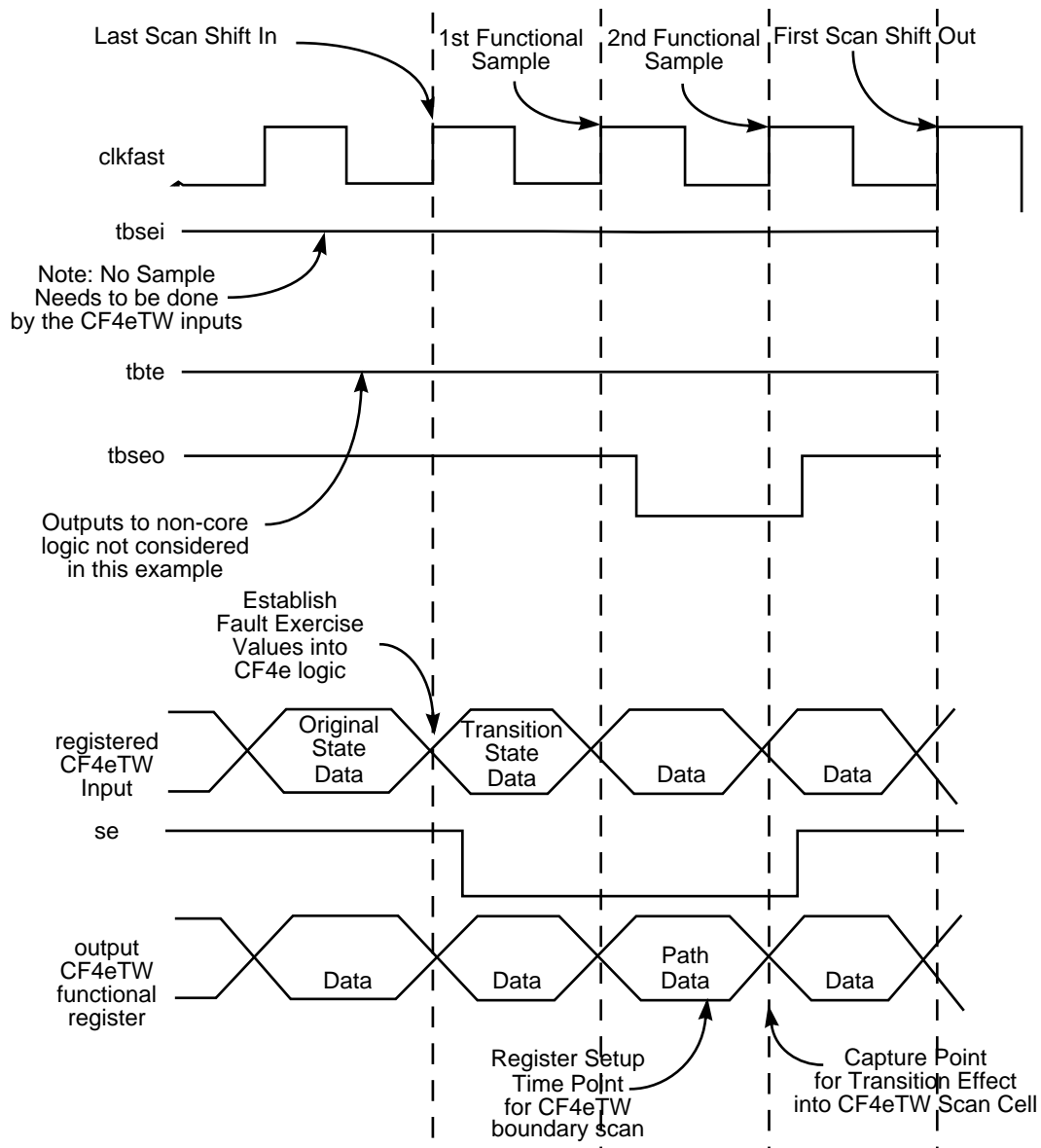
Because the CF4e core and the CF4eTW share the same system clock, the next system clock rising-edge after the first sample is the path delay effect sample cycle. This requires that the CF4eTW scan enable signal must be negated to allow the CF4eTW scan architecture to capture the effect of the launched transition. Note that the CF4eTW input side scan enable does not need to negate because this test is to verify from the internals to functional outputs. Any sampling done by the input registers of the CF4eTW would be verifying the outputs of the noncore logic. Figure 12-9 shows timing for a CF4e core to output wrapper scan stuck-at vector.



**Figure 12-9. CF4e Core to CF4eTW Output Scan Stuck-At Vector Example**

Figure 12-10 shows timing for a CF4e core to output wrapper scan delay vector.





**Figure 12-10. CF4e Core to CF4eTW Output Scan Delay Vector Example**

When a hard-layout-core is delivered, understanding this operation is unnecessary because vectors exist for these operations (for example, the CF4e core manufacturing test program).

### 12.2.4.3 CF4eTW Testing of Noncore Inputs

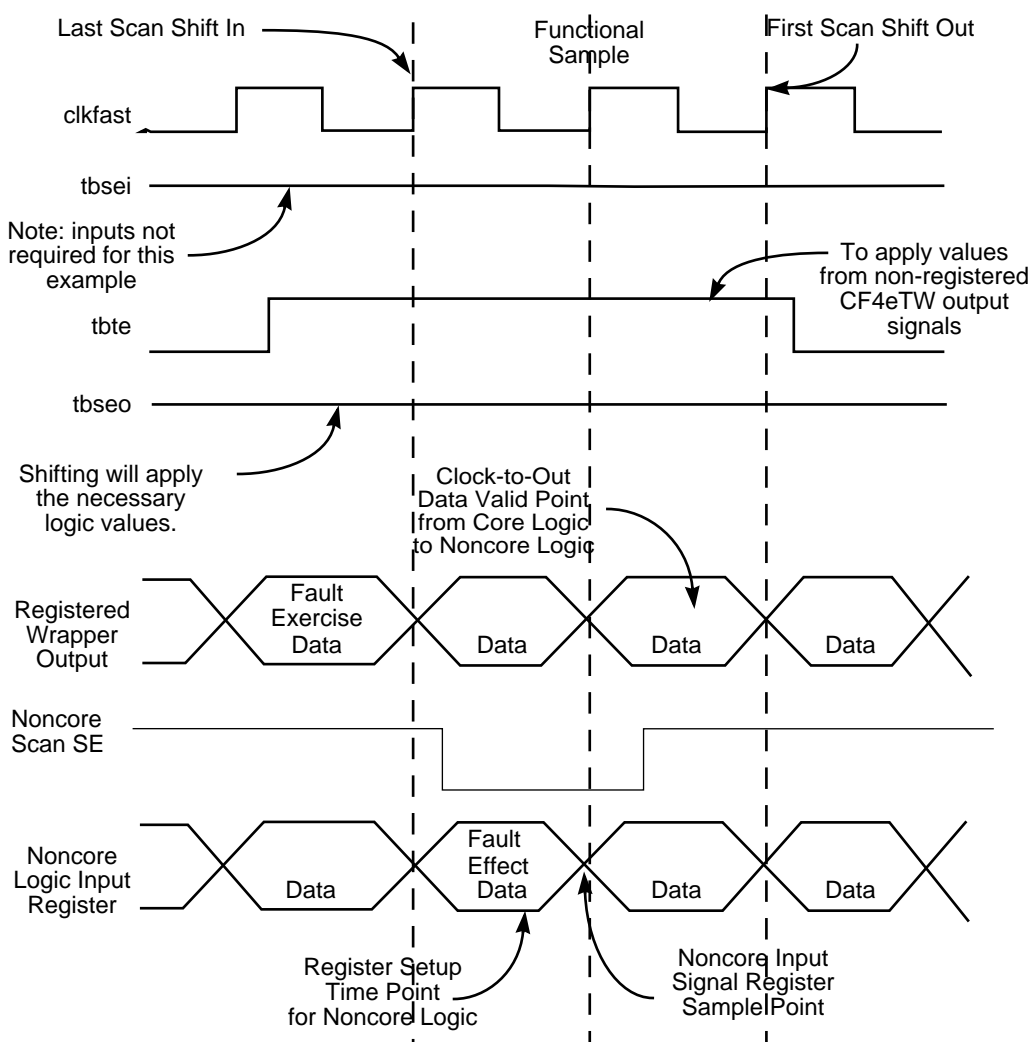
The CF4eTW is designed as a standalone device; it does not need CF4e core scan architecture for all testing. A gate-level netlist of the CF4eTW can be included as part of the noncore logic netlist when vector generation is to be done.

Wrapper scan launch mode uses the CF4eTW's ability to launch single logic values or vector pair logic transition values into the noncore logic. This testing uses **tbseo**, which enables the CF4eTW scan architecture to either shift data through the CF4eTW output side scan chains (launching data into the noncore logic) or to capture data from the CF4e core

logic. The CF4eTW can be operated coincidentally with noncore logic test structures and can be used to enable structural testing or timing delay testing.

The tbseo signal, together with the non-core logic scan or functional test mode control signals, allows launching of a single logic value to conduct structural stuck-at testing, or allows the launching of two consecutive differing logic values (vector pairs) on targeted input signals, while holding other signals stable for 2 cycles (applying the same value). The 2-cycle transition type of sequence that holds off-path values stable results in what is known as a robust delay test.

Figure 12-11 shows timing for a wrapper to non-CF4e logic scan stuck-at vector.



**Figure 12-11. CF4eTW to Non-Core Input Scan Stuck-At Vector Example**

Figure 12-12 shows timing for a wrapper to non-CF4e logic scan delay vector.

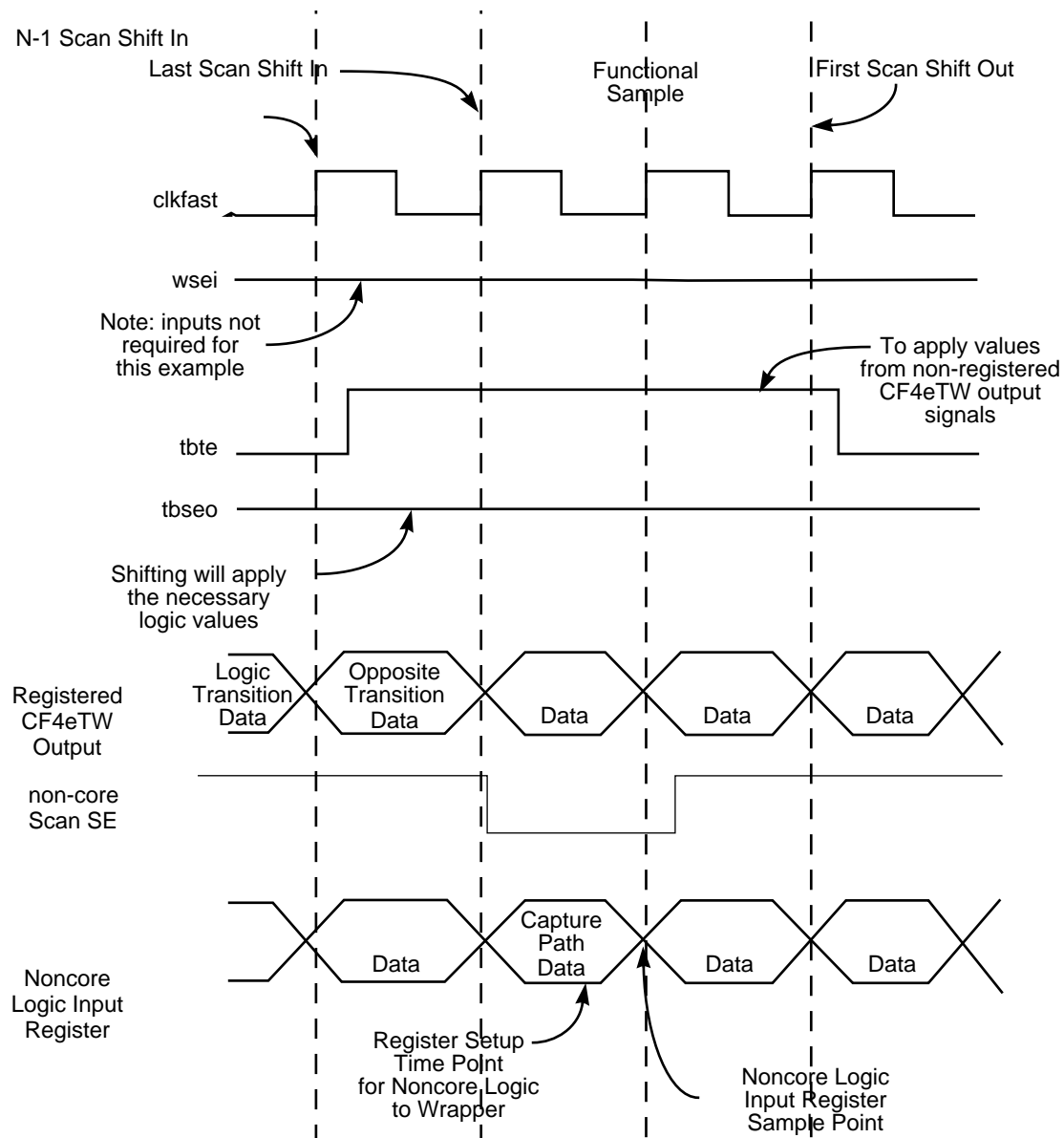


Figure 12-12. CF4eTW to Non-Core Delay Scan Vector Example

#### 12.2.4.4 CF4eTW Testing of Noncore Outputs

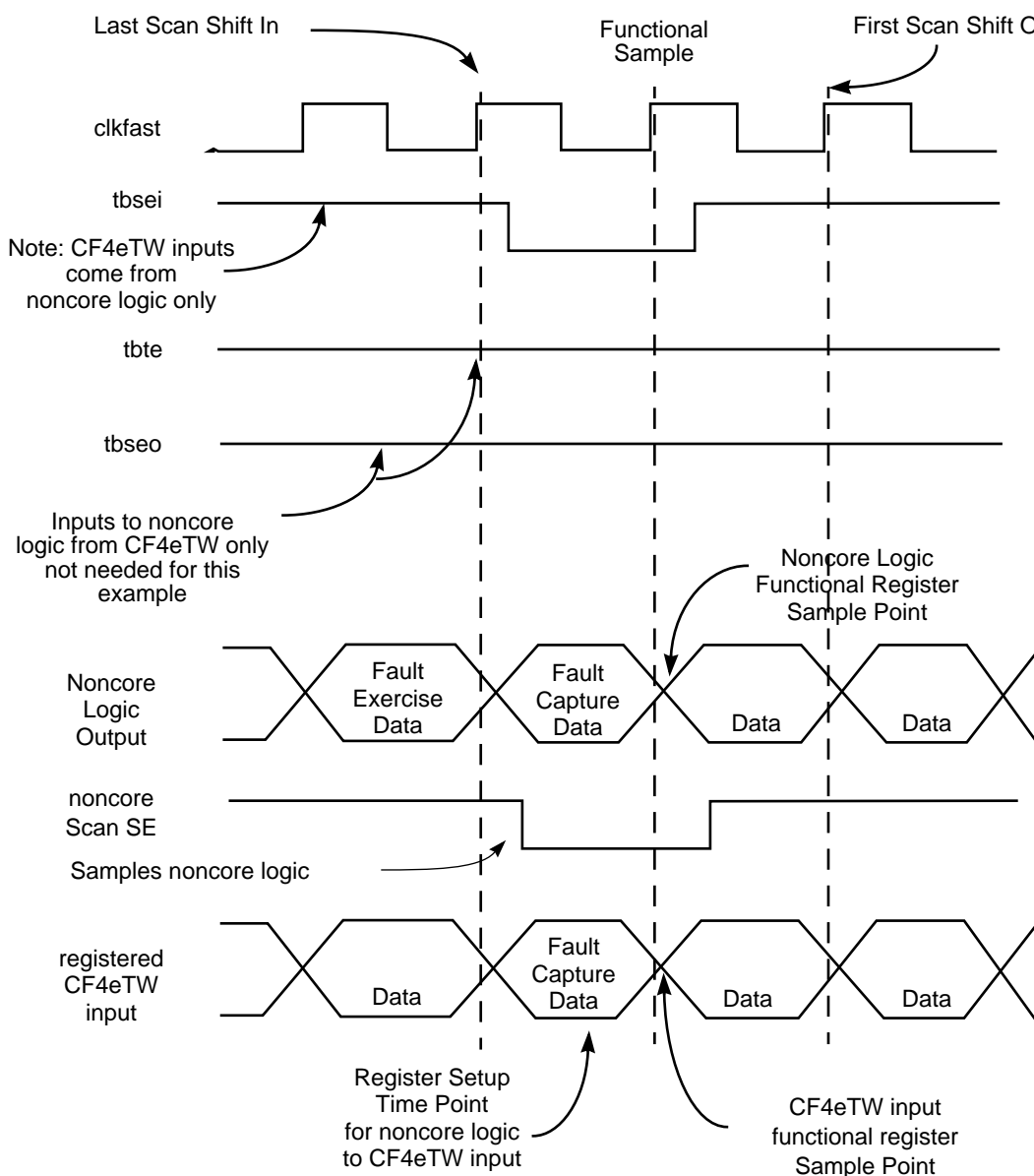
The CF4eTW wrapper scan capture mode operates independently of the CF4e core scan architecture. The CF4eTW can be appended to the noncore logic to become the capture part of its test structure. Using the CF4eTW to launch or capture logic values associated with the noncore logic requires the use of a gate-level netlist of the CF4eTW to be included as part of the noncore logic netlist when vector generation is to be accomplished.

Wrapper scan capture mode uses the CF4eTW's ability to capture logic values or logic transition values launched from the noncore logic. The CF4eTW can be operated coincidentally with the noncore logic test structures and can be used to enable structural testing or timing delay testing.

Wrapper scan capture mode testing uses the tbsei signal in conjunction with the noncore logic scan or functional test mode control signals to allow the capture of single logic values, or two consecutive, differing logic values on targeted input signals.

The tbsei signal enables the CF4eTW scan architecture to either shift data through the CF4eTW input side scan chains (launching logic values into the CF4e core) or to capture data from the noncore logic. If noncore logic can launch transitions (vector pairs), the wrapper can be used to capture one or both cycles of the transitioning test. It must be noted, however, that having the ability to capture vector pairs launched from the noncore logic requires that the noncore logic supports the logic test structures to launch the vector pairs.

Figure 12-13 shows timing for a non-CF4e logic to input wrapper scan stuck-at vector.



**Figure 12-13. Non-Core to CF4eTW Input Scan Stuck-At Vector Example**

Figure 12-14 shows timing for a non-CF4e logic to input wrapper scan delay vector.

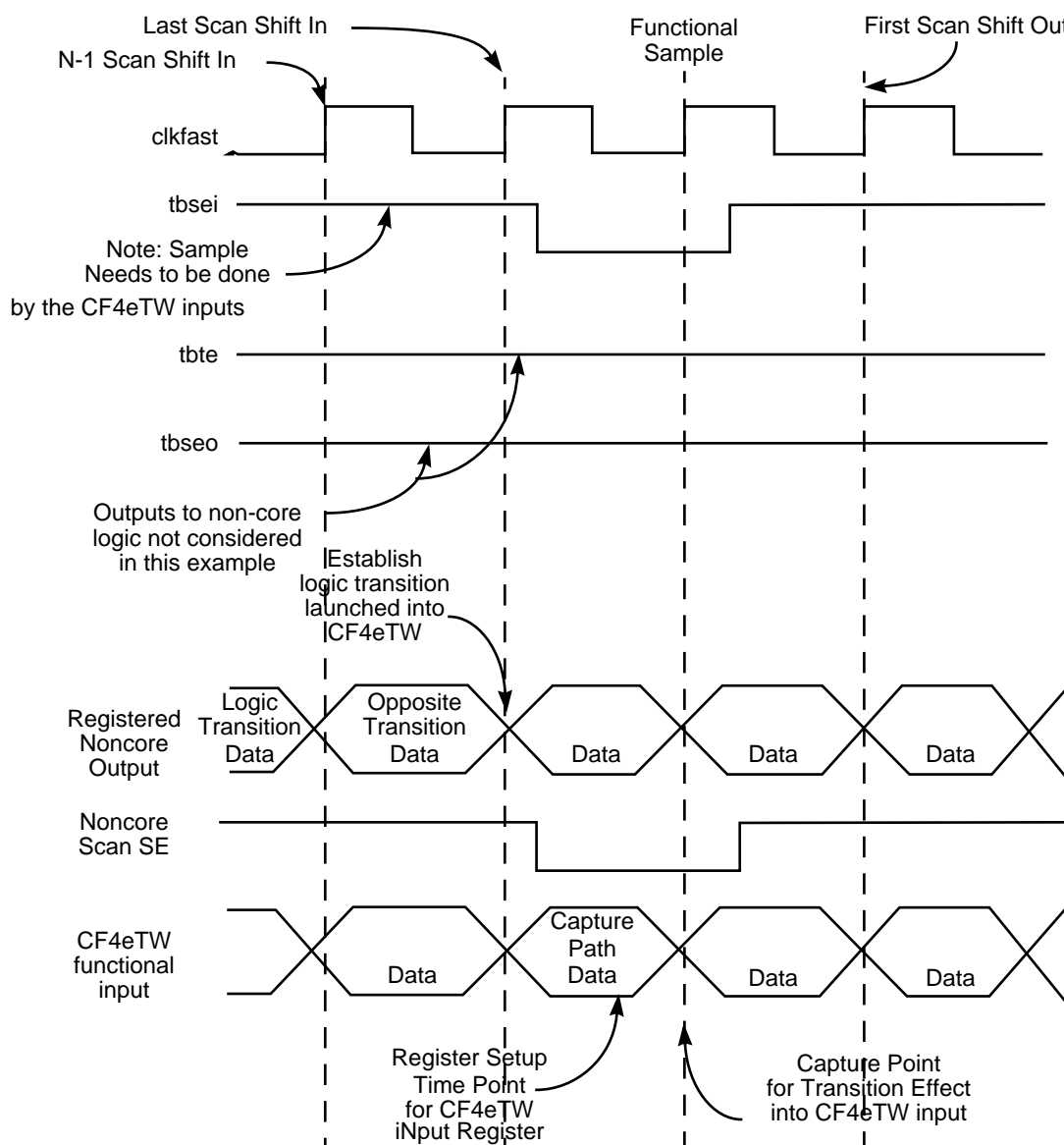


Figure 12-14. Non-Core to CF4eTW Input Scan Delay Vector Example

## 12.3 BIST

The following sections describe the Version 4 BIST structure, which is intended to assist customers, production engineers, and test engineers. Topics include the following:

- BIST memory controllers
- BIST core ports
- Power analysis
- Testing algorithms
- BIST test modes

## BIST

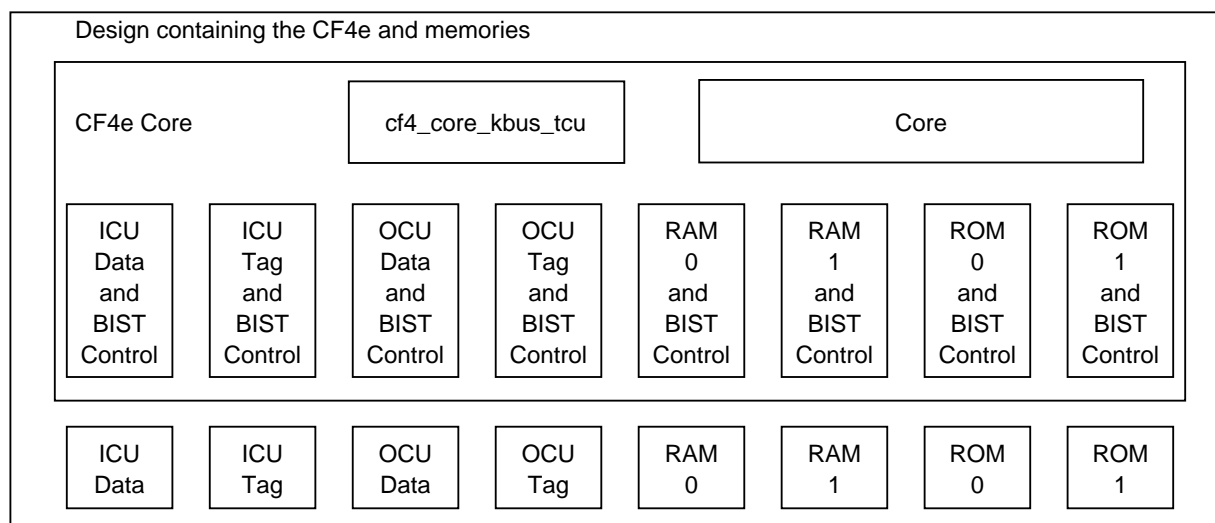
- Memory data retention
- Timing

The Version 4 test methodology for testing memories differs from the Version 3 approach in the following respects:

- In Version 4, the BIST control logic is integrated into the memory controllers rather than being a separate, added function. This solution supports all V4 memories sizes; KRAM and KROM sizes from 512 bytes to 64 Kbytes and cache sizes from 2 to 32 Kbytes. This enhancement from the Version 3 BIST is developed around the memories after memory size is selected for a design. In the Version 3 scheme, modification of any memories requires rework of BIST logic.
- Data retention methodology adds automatic hold logic to the BIST controllers, which greatly simplifies control logic for data retention and reduces the knowledge and intervention needed by the user/tester.
- Simplifying data retention logic eliminates the need for staggering hold signals for different size memory arrays. In Version 3, MTMOD signals controlled assertion and release of the BIST hold signals for memory devices. Due to BIST hold encodings, MTMOD signals had processor clock timing associated with them. In Version 4, MTMOD signals have system clock timing associated with them because hold logic is removed from the MTMOD decode. In addition, significantly fewer modes are needed with MTMOD for BIST.

### 12.3.1 BIST Memory Controllers

Control logic includes BIST memory controllers for each memory, reducing critical timing paths by collapsing sequential multiplexing of address lines. This reduces wiring and simplifies register sharing. Figure 12-15 shows how this change embeds BIST logic in the core.



**Figure 12-15. CF4e BIST Hierarchy**

The ColdFire Version 4 reference design handles up to 64-Kbyte memory sizes, using the memory size indicator to create a specific test for each memory. The `cf4_core_kbus_tcu` provides global control of each BIST controller associated with each memory.

BIST memory controllers now have automatic hold logic. Two holds are performed during the first background. The `bisthold` signal indicates when the memory is in a hold state; the `bistrelease` input removes memories from the hold state.

### 12.3.2 BIST Core Ports

During memory array testing, BIST ports must be set as shown in Table 12-3. Memory BIST logic has two modes chosen through `mtmod[2:0]`. Production BIST (PBIST) indicates failing or passing devices. Engineering BIST (EBIST) characterizes memory. The `resetB` signal is driven with an OR of `MTMOD` equal to the modes (see Table 12-3). The remaining core signals are set to an inactive state. All signals are registered at the CF4e boundary.

The `bistdata` output operates at half the processor clock frequency; all other BIST signals operate at system clock speed. The system/processor clock ratio does not affect BIST operation. If this were a pad-level rather than a core-level boundary, `MTMOD` would be expanded from 2:0 to 3:0 where the assertion of `MTMOD3` indicates PLL bypass mode, and a `hizb` signal would be asserted during data retention to three-state the outputs.

**Table 12-3. BIST Core Pins**

Port Name	I/O	BIST Mode	Description
<code>bistdone</code>	Output	PBIST	Indicates the test is complete. The largest memory size determines cycle run-time for production test. When asserted, <code>bistdone</code> remains asserted and the PBIST stops. During BIST initialization, this signal toggles (0-1-0) for a stuck-at fault test on this individual signal. Because <code>bistdone</code> is registered at the core boundary, a few cycles are added to BIST run time for signal output generation.
<code>bistfail</code>	Output	PBIST	Indicates a memory array failed. Once asserted, <code>bistfail</code> remains asserted and PBIST stops. During BIST initialization, <code>bistfail</code> toggles from 0-1-0 for a stuck-at-fault test.
<code>bisthold</code>	Output	PBIST/ EBIST	Indicates when BIST memory controllers are in a hold state, allowing the user to begin timing data retention. During BIST initialization, <code>bisthold</code> toggles from 0-1-0 for a stuck-at-fault test.
<code>bistdata[3:0]</code>	Output	EBIST	Outputs bit-map array data. Operates at 1/2 of the processor clock frequency. Section 12.3.8, "Memory Data Retention," gives cycle-by-cycle details. For pad-level boundaries, <code>bistdata</code> can be muxed with <code>pstddata[3:0]</code> .
<code>mtmod[2:0]</code>	Input	PBIST/ EBIST	During BIST, <code>mtmod</code> is applied one cycle before use. 101 PBIST mode. Asserted during the entire PBIST memory test. 110 EBIST mode. Asserted during the entire EBIST test, used to characterize the array selected by <code>bistmemory[2:0]</code> .

**Table 12-3. BIST Core Pins (Continued)**

Port Name	I/O	BIST Mode	Description
bistrelease	Input	PBIST/ EBIST	Used to release memories from hold state during data retention. If data retention testing is not desired, assert this signal for the entire test.
bistmemory[2:0]	Input	EBIST	Selects which memory is characterized. 000 Data cache tag 001 Data cache data 010 Instruction cache tag 011 Instruction cache data 100 RAM 0 101 RAM 1 110 ROM 0 111 ROM 1

### 12.3.3 Power Analysis

To maximize testing capabilities, reduce design time, and prevent potential brown-out conditions that could occur if too many memories are switching simultaneously, it is critical to analyze power considerations when memories are tested in parallel. Packaging can also affect power considerations. The reference design has a potential of eight memories:

- Operand cache unit (OCU) tag
- OCU data
- Instruction cache unit (ICU) tag
- ICU data
- RAM 0
- RAM 1
- ROM 0
- ROM 1 array

Caches can be from 2 to 32 Kbytes. RAMs and ROMs can be 512 bytes to 64 Kbytes. Based on BIST controller operation, memory size is independent. If an array is 8 or 32 Kbytes, only one 2-Kbyte 512 x 32 SRAM block is active at a time; 2-Kbyte basic memory blocks are assumed.

Pins and internal core logic not used for BIST (that is, all signals not in Table 12-3) are set to a quiescent state during initialization, so BIST power consumption is negligible compared to functional operation.

### 12.3.4 Staging of Memories

If the number of memories to be tested warrants parallel-sequential testing, memories are split into two groups and one is tested first. After all memories in the first group are tested (logical AND of all controller bistdone signals) with no failures, the second group is tested. A similar approach is used to test data retention, using an output signal for each array



controller to indicate when memories are in a hold state.

As soon as all of the memories are held in the first group, the second group is initialized. This staging is controlled internally and does not affect the user. Staging adds 200 ms to the BIST memory testing and an additional 200 ms to data retention testing if it is not included in memory testing. Staging increases estimated test time from 0.9 to 1.3 seconds.

## 12.3.5 Testing Algorithms

Data and instruction cache arrays are tested along with the RAM arrays using the March C+ algorithm. ROM arrays are tested using a prime polynomial of order 32.

### 12.3.5.1 March C+ Algorithm

The March C+ algorithm has six parts with an example background of 5s and As.

#### NOTE:

Automatic holds, or self pauses, occur only during background 5-A at the end of the first two parts.

1. Part 1
  - a) Begin at first address location (address 0)
  - b) Initialize (write) all locations with the initial data background
  - c) Conduct self-pause (only for first data background pass 5-A)
2. Part 2
  - a) Release self-pause (only for first data background pass 5-A)
  - b) Begin at first address location (address 0)
  - c) Read (initial)—write (complement)—read (complement)
  - d) Increment address and repeat for entire address space
  - e) Conduct self-pause (only for first data background pass 5-A)
3. Part 3
  - a) Release self-pause (only for first data background pass 5-A)
  - b) Begin at first address location (address 0)
  - c) Read (data)—write (complement)—read (complement)
  - d) Increment address and repeat for entire address space
4. Part 4
  - a) Begin at last address location (address max)
  - b) Read (data)—write (complement)—read (complement)
  - c) Decrement address and repeat for entire address space

5. Part 5
  - a) Begin at last address location (address max)
  - b) Read (data)—write (complement)—read (complement)
  - c) Decrement address and repeat for entire address space
6. Part 6
  - a) Begin at last address location (address max)
  - b) Read (data)
  - c) Decrement address and repeat for entire address space

The March C+ algorithm is a 14N per data background with 14 operations per data word. A data background is a bit stream pair, such as 5-A, 3-C, and 0-F. Motorola prefers using three backgrounds to provide coverage of the memory array independent of physical organization. The March C+ algorithm provides a fault coverage of more than 99.9% of the memory defect classes.

### 12.3.6 ROM BIST Algorithm

A memory BIST for a read-only memory (ROM) has two purposes: to verify data in the ROM and to ensure no defects affect that data when a read operation is conducted (a destructive read). ROM verification uses a compression scheme in which the compressed data (or signature) is compared against a golden value created when the ROM data is programmed. The scheme is based on cyclic-redundancy code (CRC) methodology, which uses a linear feedback shift register (LFSR). An LFSR that has inputs as the output data bus from memory is generally called a multiple input signature-analysis register (MISR).

MISR-LFSR is a shift register in which the last (right-most) bit is brought back to various bits along the length of the shift register through XOR gates. The memory data bus is also brought into the LFSR through XOR gates. The bits that receive the feedback are chosen from polynomial tables. The initial LFSR state before it captures any data is called the seed. The MISR-LFSR operation is similar to division by a prime number, as follows:

- The input data stream = the dividend
- The polynomial = the prime divisor
- The state of the MISR-LFSR after each capture cycle = the remainder

The ROM BIST has two read passes on the memory, as shown in the following steps:

1. Begin at first address location (address 0)
2. Read (data)—compress (data)
3. Increment address and repeat for entire address space
4. Conduct self-pause
5. Release self-pause
6. Begin at first address location (address 0)

7. Read (data)—compress (data)
8. Increment address and repeat for entire address space
9. Conduct self-pause
10. Release self-pause
11. Compare signature to stored signature value
12. Assert done and assert fail if necessary

The first pass through the ROM should begin at address 0. Each address location is read as the test increments through the address space and captures the data in the MISR. The first pass verifies the program data and the address decode.

The second read also starts at address 0 (step 6) and increments through the address. The second read tests for any defects resulting from addressing or reading destructive reads. The ROM is designed so that the golden comparison signature sets the MISR to a known value (all zeros) at the end of the second read pass. The ROM signature is placed in the last address (address max), so this value can be read during testing because it returns the signature analyzer to its pretest state. The Version 4 ROM BIST logic assumes ROM array dimensions of 32 bits by length L. At the conclusion of the reads, a fail flag is set if the calculated signature does not match the golden signature.

#### NOTE:

A retention test is not required, but is in the Version 4 ROM solution. HOLD mode pauses the ROM simultaneously with other memory arrays. This allows memories to gracefully remain in a quiescent state during data retention.

The scripts written in Section 12.3.6.1, “Modify BIST ROM Signature Script—Part 1,” and Section 12.3.6.2, “Modify BIST ROM Signature Script—Part 2,” together create a self-checking ROM signature value. The pseudo signature is the last line in the array. Hence, the last line of the ROM cannot be used for functional purposes. When the BIST MISR is applied to the ROM array, the final signature results in a value of all zeros if the array has no errors. The script calculates the signature using the same algorithm, prime polynomial of order 32, that the ROM BIST module uses. The prime polynomial used in an N-bit polynomial is a N-bit binary expansion of a prime number that falls between  $2^{(N-1)}$  and  $2^N$ . Because of the math theorem that states that there is at least one prime number between any numbers X and 2X, at least one prime number lies between  $2^{(N-1)}$  and  $2^N$ . Motorola chose one of these prime numbers for the algorithm.

#### 12.3.6.1 Modify BIST ROM Signature Script—Part 1

```
#####
# FILE: modify_signature.sh
# DESCRIPTION:
# shell to modify N x 32 ROM last word (signature) for V4 ROM BIST. This shell
# takes as input a file name of the original ROM code, and creates a new file
# called by the original filename with ".modified" appended. This shell also
# diffs the two files.
```

```
# The only difference is the modified last line. The original file's last line
# does not change. The new file's last line contains the correct ROM BIST
# signature of the proceeding N-1 lines.
#####
# INPUT: The input file is an array of N lines each of which contains 32 ascii
# ones
# (1)s and zeroes (0)s. It should NOT contain any blank lines, NOR any spaces.
#####
# OPERATION: call as such: modify_signature.sh original_filename
# This script invokes a awk script called modify_signature.awk.
#####
sed 's/0/ 0/g' $1 | sed 's/1/ 1/g' | sed 's/^ //' | awk -f modify_signature.awk
|
sed 's/ //'g' > $1.modified
diff $1 $1.modified
```

### 12.3.6.2 Modify BIST ROM Signature Script—Part 2

```
#####
# FILE: modify_signature.awk
# DESCRIPTION: Use awk to emulate V4 ROM BIST algorithm.
# N is the length of the array.
#####
# initialize the signature register to all zeroes
#####
BEGIN{for(initial_count=0;initial_count<32;initial_count++)
signature[initial_count]=0}
#####
# calculate next_signature, the only thing special is bit 31
#####
{for(next_signature_count=0;next_signature_count<31;next_signature_count++)
next_signature[(30-next_signature_count)]=xor(signature[(31-
next_signature_count)],$(next_signature_count+2));
next_signature[31]
xor(xor(xor(signature[0],signature[1]),xor(signature[2],signature[22])),,$1);
#####
# move next_signature to current signature.
#####
for(signature_count=0;signature_count<32;signature_count++)
signature[signature_count] = next_signature[signature_count];
#####
# The last line of the file is the signature.
# If NR is equal to the last line, modify it. Else reprint the line.
#####
if (NR == N)
print xor($1,signature[31]) xor($2,signature[30]) xor($3,signature[29])
xor($4,signature[28]),
xor($5,signature[27]), xor($6,signature[26]) xor($7,signature[25])
xor($8,signature[24]),
xor($9,signature[23]), xor($10,signature[22]) xor($11,signature[21])
xor($12,signature[20]),
xor($13,signature[19]) xor($14,signature[18]) xor($15,signature[17])
xor($16,signature[16]),
xor($17,signature[15]) xor($18,signature[14]) xor($19,signature[13])
xor($20,signature[12]),
xor($21,signature[11]) xor($22,signature[10]) xor($23,signature[9])
xor($24,signature[8]),
xor($25,signature[7]), xor($26,signature[6]) xor($27,signature[5])
xor($28,signature[4]),
xor($29,signature[3]) xor($30,signature[2]), xor($31,signature[1])
xor($32,signature[0])
else
print $1 $2 $3 $4,$5 $6 $7 $8,$9 $10 $11 $12,$13 $14 $15 $16,$17 $18 $19 $20,$21
22 $23 $24,$25 $26 $27 $28,$29 $30 $31 $32}
#####
```

```
# xor function
#####
function xor(first,second)
{if (first == second)
return 0
else
return 1
```

### 12.3.7 BIST Test Modes

Production BIST (PBIST) indicates failing or passing devices. Engineering BIST (EBIST) characterizes memory. These modes are chosen through `mtmod[2:0]` (see Table 12-3). PBIST tests all memories in parallel with individual BIST controllers. If one fails, a failflag is asserted. When memory testing completes, `bistdone` is asserted.

EBIST uses the same memory BIST algorithm for bit-mapping memories. Due to VLSI tester limitations, only one memory can be bit-mapped at a time and is chosen through `bistmemory[2:0]` which selects one of the potential eight memories listed in Section 12.3.3, “Power Analysis.” The data of the memory that is characterized is output onto `bistdata[3:0]`.

On Version 4, `bistdata` operates at half the processor clock frequency. BIST data is captured at the frequency at which the memories are operating. To output the data, a complete BIST test is executed outputting the first data (`data[3:0]`). Next, the test is rerun on the entire memory selecting bit `data[7:4]`. This sequence repeats until all memory is bit-mapped. No data is lost between capturing the memory data at the processor clock speed and multiplexing the data onto `bistdata`. BIST algorithms take multiple processor clocks per address so data is not changing on a processor cycle-by-cycle basis. This flow process is exited when the MTMOD state changes. Figure 12-16 illustrates this process.

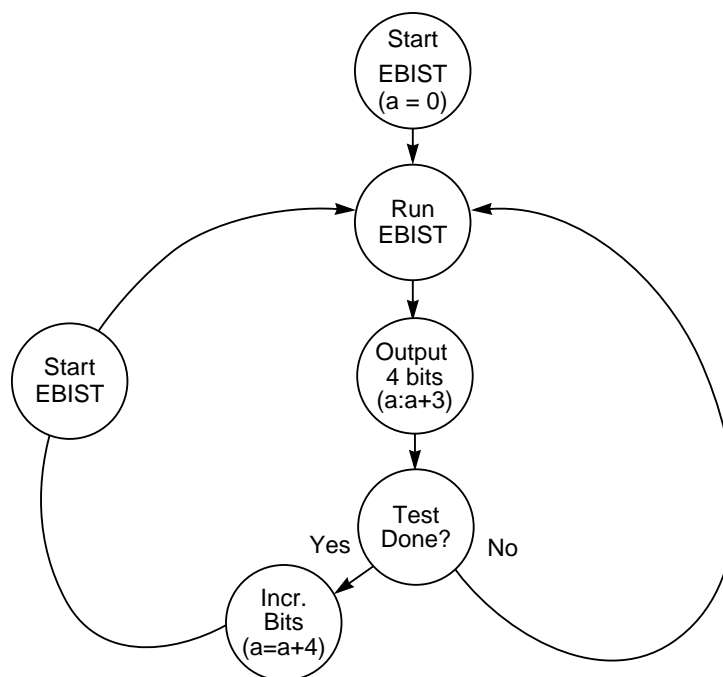
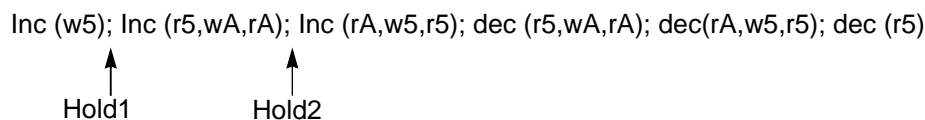


Figure 12-16. Flow of Characterization Method

## 12.3.8 Memory Data Retention

For maximum efficiency, the data retention scheme initializes all memories and places each in a hold state at the appropriate time. Two data retention holds occur automatically during background 5-A. As Figure 12-17 shows, the first hold occurs when memories under test are initialized with 5s; a second occurs when they are initialized with As.



**Figure 12-17. March C+ Algorithm**

The bisthold output indicates that all memories under test are in hold state. At that time, the user begins counting the hold time on the memories. When bistrelease is asserted, the hold is released and testing continues. bistrelease should be negated before the end of the next march part. Steps for data retention are as follows:

1. Assert PBIST (MTMOD = 101) or EBIST (MTMOD = 110)
2. When bisthold is asserted, the user can start a data retention counter. (Assume bistrelease is negated.)
3. Assert bistrelease to release hold on memories and continue testing.
4. To perform data retention on inverse data, negate bistrelease and repeat steps 2 and 3.

The bistfail signal asserts if a memory failure occurs during the next read of the array. The internal controller initializes all memories in each group if memory staging is required; that is, power consumption becomes a criterion because of the number of memories under test. The internal controller releases the first group so one group can be tested at a time. This complication is not apparent to the user.

BIST ROM controllers include data retention logic for consistency when testing other memories with data retention, not for ROM data retention testing itself. For example, when all memories on a chip are tested in PBIST mode with data retention, the clock may not always be asserted during the actual data retention. Therefore, ROM BIST logic should contain hold logic to prevent the ROM BIST controller from becoming lost. For ROM BIST, the hold occurs after a read has completed.

If data retention is not desired, assert bistrelease throughout the test. At least two processor clock delays occur for each hold.

## 12.3.9 Timing

The following sections describe how to determine the clock cycle count and provide BIST timing examples.

### 12.3.9.1 Memory Clock Determination

For tests using the March C+ algorithm, note that the algorithm has six parts, each with a READ-WRITE protocol. Each protocol requires six clock cycles to conduct an rwr, r, or w sequence, three data backgrounds, and the number of address locations. The example in Figure 12-18 determines the minimum cycle count for a 512 x 32 RAM with one clock cycle read and write protocols.

**NOTE:**

This example does not factor in initialization time.

$$(512 \text{ words} \times 6 \text{ cycles/word}) \times (6 \text{ parts/March C+ algorithm}) \times (3 \text{ backgrounds}) = 55296 \text{ cycles}$$

**Figure 12-18. 512 x 32 RAM BIST Clock Cycles**

ROM array cycles have three parts to each READ sequence (three clocks per address). The read through the memory occurs twice. The example in Figure 12-19 is for a 512 x 32 ROM with 1 clock cycle read protocols. This example does not factor in initialization time.

$$(512 \text{ words} \times 3 \text{ cycles/word}) \times (2 \text{ read parts}) = 3072 \text{ cycles}$$

**Figure 12-19. 512 x 32 ROM BIST Clock Cycles**

The clock cycle determination for PBIST and each EBIST is calculated in Figure 12-4 with examples of an 8-Kbyte cache and a 4-Kbyte RAM. As discussed above, BIST test requires 6 clock cycles per address, the March C+ test has six parts for a complete test, there are three backgrounds applied per BIST test, and there are eight tests run for a complete EBIST test, given a data array of width 32.

A PBIST test is completed when the largest array has finished or when a failure occurs on an array. The variable X accounts for extra clock cycles, including core initialization, BIST reset, and holds associated with data retention. During the beginning of a BIST test, there are 16 processor clocks for core initialization and 12 processor clocks for BIST reset initialization. Each restart of the BIST test during EBIST mode requires another BIST reset initialization. For an PBIST test without data retention, X is 28 processor clocks. A few cycles should be padded at the end of a PBIST mode test to wait for bisdone to assert.

**Table 12-4. BIST Cycles**

Parameter	Data Array (8K)	RAM Array (4K)	Tag Array (8K)
Max address space (MAS)	2048	1024	512
Clks/March C+ algorithm part (part offset: P)	$6 \times 2048 = 12288$	$6 \times 1024 = 6144$	$6 \times 512 = 3072$
Clocks/March C+ algorithm (background offset: B)	$6 \times 12288 = 73728$	$6 \times 6144 = 36864$	$6 \times 3072 = 18432$
Clocks/March C+ algorithm with three backgrounds (Test Offset: T)	$3 \times 73728 = 221184$	$3 \times 36864 = 110592$	$3 \times 18432 = 55296$

Table 12-4. BIST Cycles

Parameter	Data Array (8K)	RAM Array (4K)	Tag Array (8K)
Clocks/complete PBIST test	221184 + X	110592 + X	55296 + X
Clocks/complete EBIST test	$4 \times 221184 = 884736 + X$	$4 \times 110592 = 442368 + X$	$4 \times 55296 = 221184 + X$

### 12.3.10 Timing Diagrams

Initialization is similar for PBIST and EBIST. In the PBIST mode example in Figure 12-20, Only the first 27 cycles are shown; the remaining cycles should remain at the state shown in cycle 27 unless there is a failure with the exception of *bisthold* and *bistrelease*. The *mtmod[2:0]* signals are always assumed to be in non-BIST state before going into BIST mode in cycle 3. In PBIST mode, core initialization takes 16 processor clocks, during which the internal BIST control resets the internal control logic and toggles the BIST outputs *bistdone*, *bistfail*, and *bisthold* for stuck-at-fault coverage.

The testing algorithm begins at cycle 30. For PBIST tests, *bistfail* and *bistdone* are monitored. For EBIST tests, reinitialization between BIST runs takes 12 cycles and would be represented by cycles 18–30.

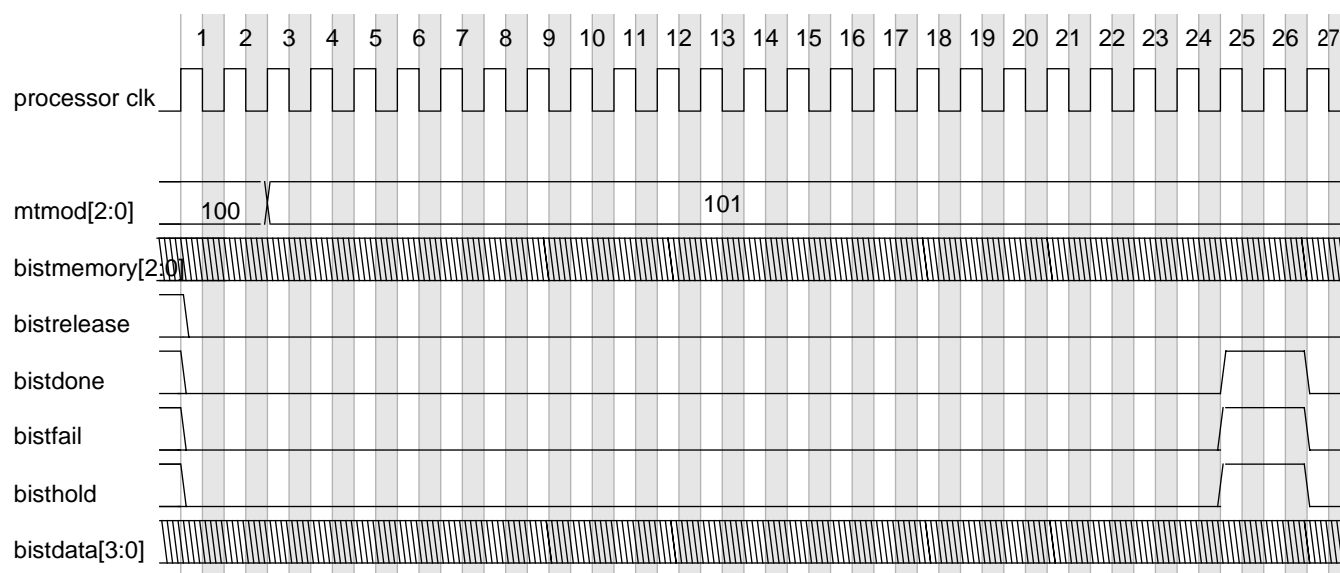


Figure 12-20. PBIST Initialization

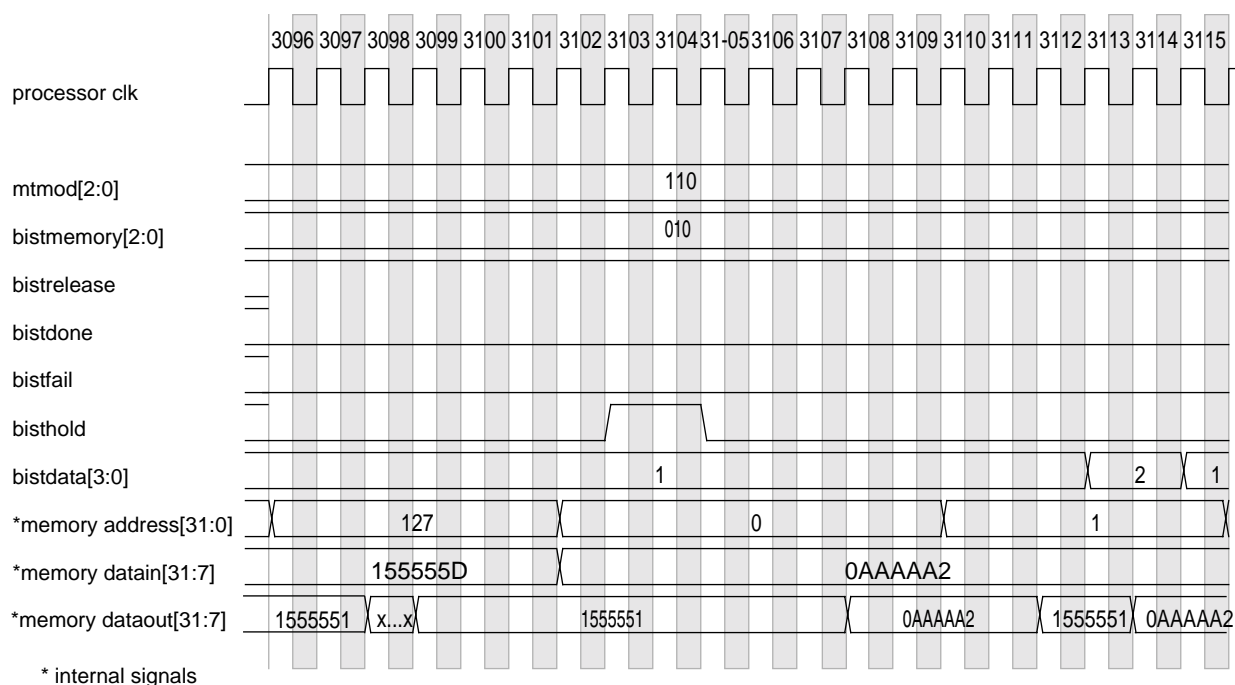
In Figure 12-21, an 8-Kbyte ICU tag array is tested with EBIST mode. The cycles shown represent a transition from the [W(5)] March C+ part to the [R(5),W(A),R(A)] part. This example shows *bistdata* output during address 0, [R(5)] March C+ part output data, all other March C+ part output data, and a minimum data retention hold.

The first March Part [W(5)] *bistdata* is always data, or 5 in this case. Consider the following cases:



- During address 0 when there is a minimum automatic hold, bistdata is [data, data, data, data, data, data, data, data].
- During address 0 when there is not an automatic hold, bistdata is [data, data, data, data, data, data, data].
- At all other addresses, bistdata is [data,data,data,DATA,DATA,data], in this case [1,1,1,2,2,1].

The bistdata signals always operate at half the processor clock frequency. All other BIST input and output pins are clocked by the system clock. In this example, the system/processor clock ratio is 2:1, only the processor clock is displayed. This ratio can be 2:1, 3:1, or 4:1. This example is specific to EBIST tag arrays.



**Figure 12-21. EBIST Timing Diagram for an 8-Kbyte Cache Tag Array**

Figure 12-21 shows EBIST test 0 with background 0. The bistdata output for part of the array is data 1, data 2 due to the fact that the tag arrays widths are maximum of 25 bits. Figure 12-5 shows variations on the BIST data output for the difference size memory arrays given the test number and background. The information in the table is data, data, and old data, where old data is the data from the last background.

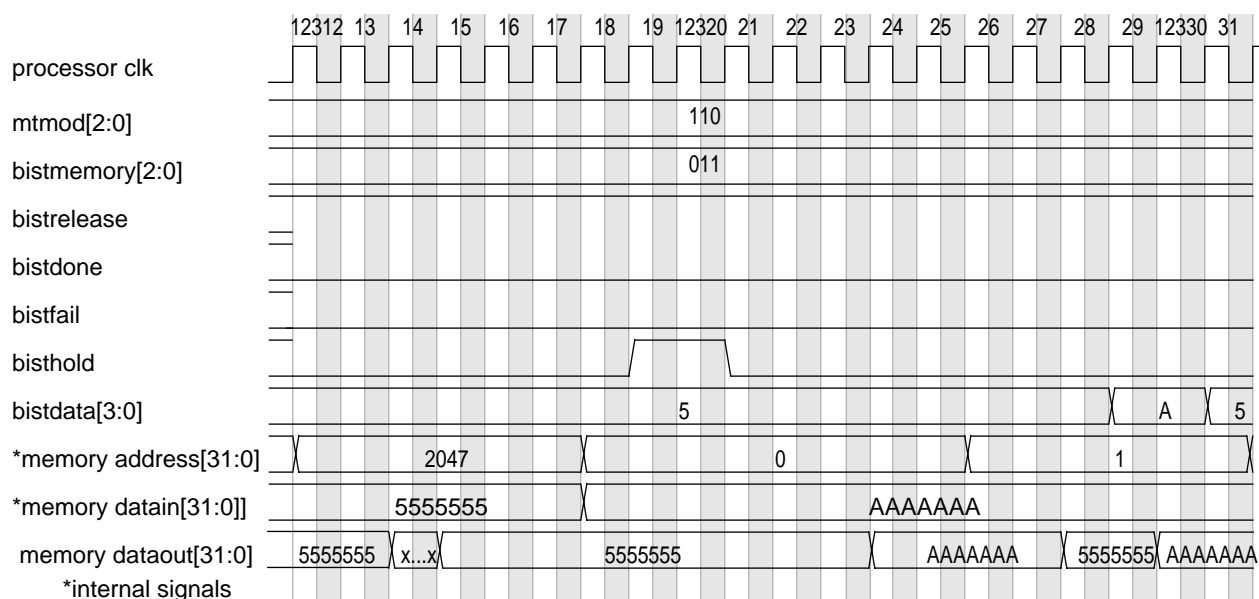
**Table 12-5. EBIST Tag Output Data**

Tag Size	Test 0			Test 1			Test 2+		
	Bckgnd 0	Bckgnd 1	Bckgnd 2	Bckgnd 0	Bckgnd 1	Bckgnd 2	Bckgnd 0	Bckgnd 1	Bckgnd 2
2K	5, A, X	3, C, 5	0, F, 3	5, A, 0	3, C, 5	0, F, 3	5, A, 0	3, C, 5	0, F, 3
4K	1, A, X	3, 8, 1	0, D, 3	5, A, 0	3, C, 5	0, F, 3	5, A, 0	3, C, 5	0, F, 3
8K	1, 2, X	3, 0, 1	0, 3, 3	5, A, 0	3, C, 5	0, F, 3	5, A, 0	3, C, 5	0, F, 3

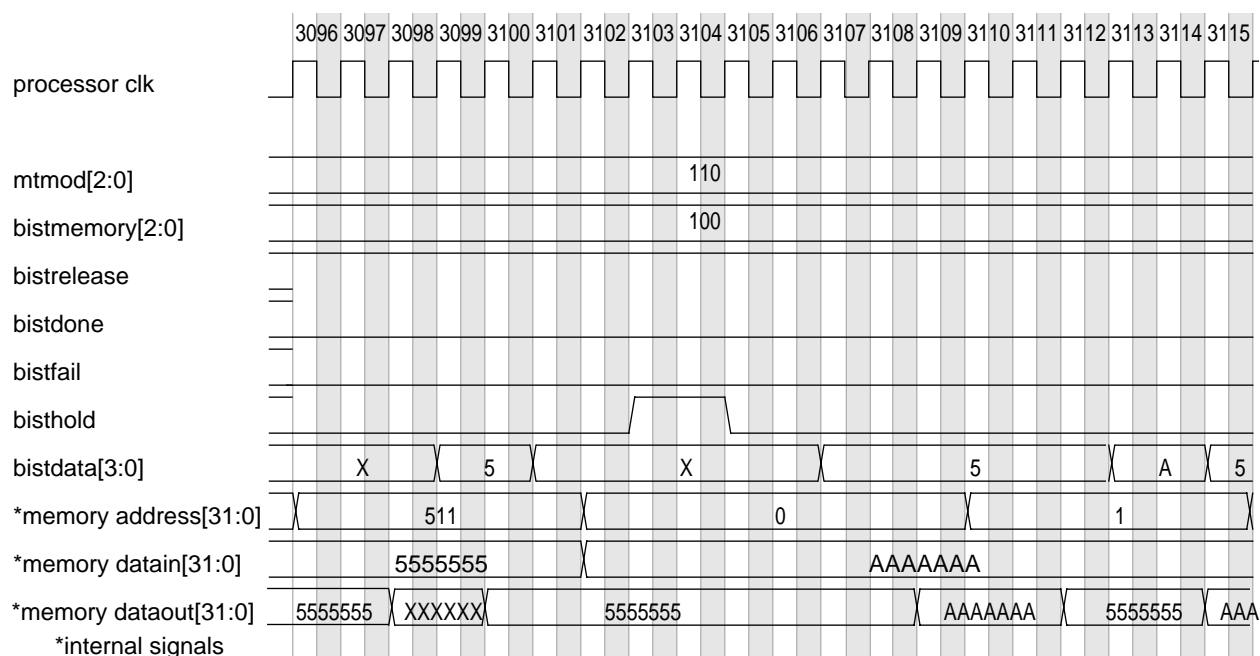
**Table 12-5. EBIST Tag Output Data (Continued)**

Tag Size	Test 0			Test 1			Test 2+		
	Bckgnd 0	Bckgnd 1	Bckgnd 2	Bckgnd 0	Bckgnd 1	Bckgnd 2	Bckgnd 0	Bckgnd 1	Bckgnd 2
16K	1, 2, X	3, 0, 1	0, 3, 3	4, A, 0	2, C, 5	0, E, 2	5, A, 0	3, C, 5	0, F, 3
32K	1, 2, X	3, 0, 1	0, 3, 3	4, 8, 0	0, C, 4	0, C, 0	5, A, 0	3, C, 5	0, F, 3

In Figure 12-22, an 8-Kbyte ICU data array is tested with EBIST mode. The cycles shown represent a transition from the [W(5)] March C+ part to the [R(5),W(A),R(A)] part. During address 0, bistdata is [data, data, data, data, data, data, data, data] for the first two part transitions of the first background; minimum of two processor clock halt for the automatic hold signal. For the remaining address 0, bistdata[3:0] output is [data, data, data, data, data, data, data]. For all other addresses, then it is [data,data,data,DATA,DATA,data], in this case [5,5,5,A,A,5]. In this example, the system/processor clock ratio is 2:1, but only the processor clock is displayed.

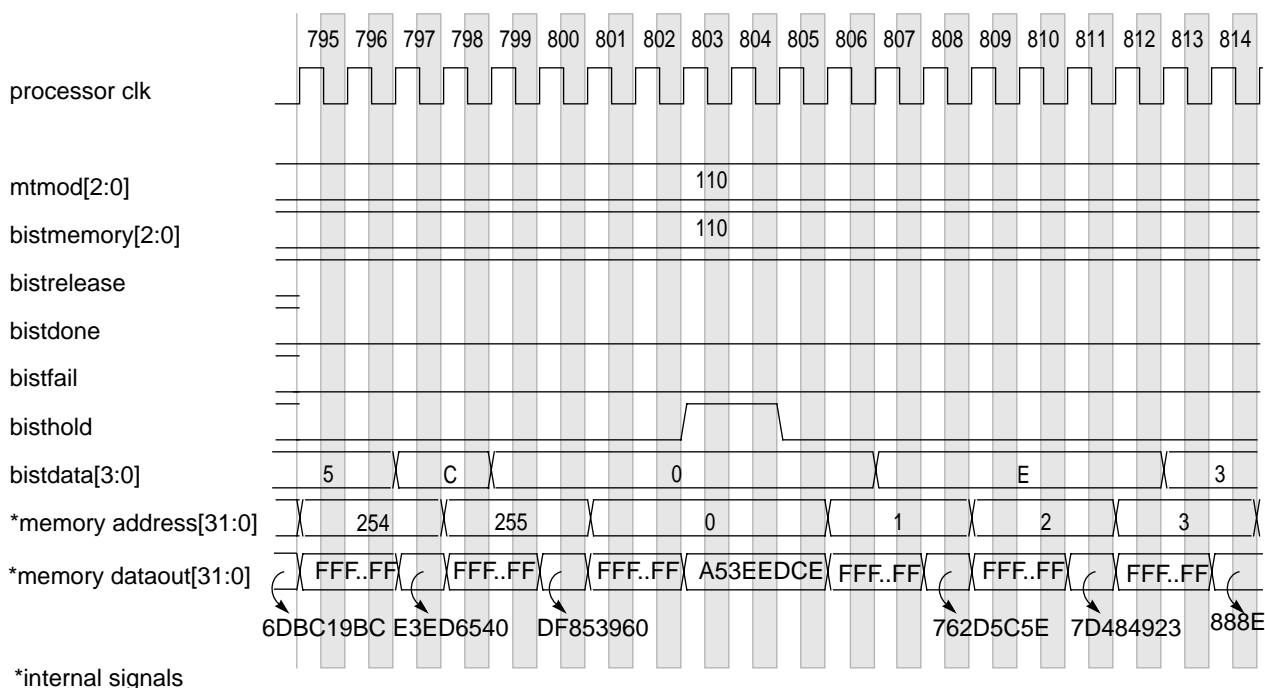
**Figure 12-22. EBIST Timing Diagram For An 8-Kbyte Cache Data Array**

In Figure 12-23, a 2-Kbyte KRAM0 array is tested in EBIST mode. The cycles shown represent a transition from the [W(5)] March C+ part to the [R(5),W(A),R(A)] part. During address 0, bistdata[3:0] is [olddata, olddata, olddata, olddata, olddata, data, data, data] for the first two parts of the first background; minimum of two processor clock halt for the automatic hold signal. Old data is the data from the previous March Part. For the remaining address 0, bistdata is [olddata, olddata, olddata, data, data, data]. For all other addresses, the bistdata output is [data,data,data,DATA,DATA,data], in this case [5,5,5,A,A,5].



**Figure 12-23. EBIST Timing Diagram For A 2-Kbyte KRAM0 Array**

Recall that the ROM EBIST algorithm is different from the other arrays. In the 2-Kbyte ROM in EBIST mode example (Figure 12-24), there are 3 processor clocks per address. The bistdata output is sampled on cycles 1 and 3. Notice that this array output does not have the special output data cases that the other arrays have. The minimum data retention delay of two processor clocks occur at address 0.



**Figure 12-24. EBIST Timing Diagram For 2-Kbyte KROM0 Array**

## 12.4 Integration Connections

The three mtmod test input signals must be routed from the package pins to the cf4\_core\_kbus\_tcu. This can be a direct connection from the package pins to the CF4eCore or another form of dedicated chip-level test selection that creates the three mtmod signals in a chip-level test controller.

Rules for connecting the CF4eCore scan interface are straightforward. When the mtmod encoding distributed to the CF4eCore is either 3 (burn-in scan mode) or 4 (production scan mode), the si and tbsi scan inputs must be connected to package pins as inputs; so and tbso scan outputs must be connected to package pins as outputs; and se, tbte, tbsei, and tbseo scan shift control signals must be connected to package pins as inputs.

The CF4e parallel scan connections can be gated off whenever they are not being used in a scan mode (when encodings 3 and 4 are not selected). At that time, parallel scan inputs must be driven to a logic 0 and parallel scan outputs must be ignored.

On the other hand, the CF4e test wrapper scan connections can be used across several test modes. Because the test wrapper is shared by CF4e and non-CF4e logic, access to the CF4e test wrapper connections must be provided in both core and non-core test modes. This should include any CF4e core scan test mode, any non-core logic scan test mode, and any non-core logic test mode that wishes to drive or view the CF4e interface.

## 12.5 Test Controller

The CF4e test controller unit (TCU) contains the head and tail registers of all the scan chains in the design. A head or tail register is the first or last register in a scan chain, respectively. Head and tail registers are used only for scan shift operations and have no functional use. The TCU decodes the three Motorola test mode signals (mtmod[2:0].) The encodings are described in the following section.

### 12.5.1 MTMOD[2:0] Encodings

The CF4e has a set of three test mode pins (mtmod[2:0]) that control all test modes of the core. Table 12-6 describes the encoding for these signals.

**Table 12-6. CF4e Motorola Test Mode Encodings**

Hex	MTMOD[2:0]	Mode or Action
0	000	Functional mode
1	001	Functional mode with JTAG mode active
2	010	PLL test mode
3	011	Burn-in scan mode
4	100	Production scan mode
5	101	Production BIST (PBIST)

**Table 12-6. CF4e Motorola Test Mode Encodings (Continued)**

Hex	MTMOD[2:0]	Mode or Action
6	110	Engineering BIST (EBIST)
7	111	Safe mode

Note the following:

- The 000 mode would be chosen for standard functional operation. None of the test signals are asserted and all test inputs default to the quiescent 0-driven state.
- The CF4e does not contain a PLL, but it is set up such that when mode 010 is chosen the core goes into an idle mode after reset.
- During burn-in scan more, both internal and wrapper scan chains are used. The memory arrays are active for activities in which randomly applied scan vectors will be written into and exercise the arrays.
- During production scan mode, both internal and wrapper scan chains are used. Memories external to the core should be write inhibited. This memory lock is to reduce noise, power, and to create a safe environment for the memory arrays since scan will randomly toggle the data, address, read-write, and output enable control signals.
- The two BIST encodings put the core into idle mode as well.
- Safe mode puts the core into reset, but still allows use of the wrapper scan chains for testing the peripheral logic. This helps with power issues and to keep the core safe while other portions of the device are being tested.



# Appendix A

## Core Interface Timing Characteristics

This appendix provides a Synopsys-compatible timing budget constraint file, which details the relative input arrival times and output delays for every interface signal in the CF4e core design. The relative timings are expressed as a fraction of the processor's cycle time to provide a relatively technology-independent timing budget.

### NOTE:

This appendix is provided as a reference. Actual pin timing is a function of synthesis methodology, process technology, place-and-route details, and external signal loading.

In this budget, maximum clock period is the period of the processor's fast clk; VCLK is simply a virtual clock reference with the same period as the maximum clock period. The virtual clock is used as a way to reference input and output timings. The variable REGSETUP defines the register setup time budget; REGDELAY defines the register output time budget. These budgets include the actual register setup and clock-to-out times, some small amount of logic, and the clock skew. Note that these variables must be linked to the target technology. The variable clk\_logic\_period is maximum clock period minus both REGSETUP and REGDELAY.

The relative timings given are designed to provide a good timing budget across a wide range of process and frequency targets. Table A-1 gives some suggestions for REGSETUP, REGDELAY, and clock logic period for various process and frequency targets.

**Table A-1. Timing Budget Variables for Various Process and Frequency Targets**

Process and Frequency Target	Maximum Clock Period	REGSETUP	REGDELAY	Clock Logic Period
0.18 $\mu$ to 0.25 $\mu$ 200 MHz	5.00 ns	1.00 ns	1.00 ns	3.00 ns
0.13 $\mu$ to 0.25 $\mu$ 250 MHz	4.00 ns	0.80 ns	0.80 ns	2.60 ns
0.13 $\mu$ to 0.18 $\mu$ 300 MHz	3.33 ns	0.67 ns	0.67 ns	2.00 ns
$\leq$ 0.13 $\mu$ 350MHz	2.86 ns	0.80 ns <sup>1</sup>	0.80 ns <sup>1</sup>	1.66 ns
$\leq$ 0.13 $\mu$ 400MHz	2.50 ns	0.80 ns <sup>1</sup>	0.80 ns <sup>1</sup>	1.30 ns

<sup>1</sup> REGSETUP and REGDELAY are padded to account for possible interconnect delay between a transmitted register and a receiving register .

```

/*****
/*****
//
// Version 4 ColdFire Reference Design INPUT/OUTPUT SIGNALS
//
/*****
/*****

/* Outputs */
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"bistdone")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"bistdata[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"bistfail")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"bisthold")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"maddr[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mtt[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mtm[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mrw")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"msiz[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mwdata[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mwdataoe")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mapb")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mdp")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"mlockb")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"bdmforceackb")
set_output_delay { REGSETUP + ( clk_logic_period * ( 1.00 - 0.10 ) ) } -clock
"VCLK" find(port,"so[*]")
set_output_delay { REGSETUP + ( clk_logic_period * ( 1.00 - 0.10 ) ) } -clock
"VCLK" find(port,"tbso[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"cpustopb")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"cpuhalte")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"pstclk")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsientb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiwrttb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiwlvt[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsirowst[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiaddr[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsisw")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsisv")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiendb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiwrtdb[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsiwtbyted[*]")

```



```

set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsirowsd[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsicwrdata[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsoentb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsowrttb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsowlvt[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsorowst[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsoaddr[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsosw")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsosv")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsoendb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsowrtdb[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsowtbyted[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsorowsd[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"nsocwrdata[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kram0addr[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kram0di[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kram0web[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kram0csb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kramladdr[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kramldi[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kramlweb[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kramlcsb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"krom0csb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"krom0addr[*]")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kromlcsb")
set_output_delay { REGSETUP - 0.20 } -clock "VCLK" find(port,"kromladdr[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"pstddata[*]")
set_output_delay { REGSETUP + 0.30 + ( clk_logic_period * ( 1.00 - 0.00 ) ) } -
clock "VCLK" find(port,"dsdo")

/* Inputs */
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"mclken")
set_input_delay { 0.00 } -clock "VCLK" find(port,"mtmod[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"bistrelease")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"bistmemory[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"si[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.00 ) } -clock "VCLK"
find(port,"se")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"tbsi[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.20 ) } -clock "VCLK"
find(port,"tbsei")
set_input_delay { REGDELAY + ( clk_logic_period * 0.20 ) } -clock "VCLK"
find(port,"tbseo")
set_input_delay { REGDELAY + ( clk_logic_period * 0.00 ) } -clock "VCLK"
find(port,"tbte")
set_input_delay { REGDELAY + ( clk_logic_period * 0.10 ) } -clock "VCLK"
find(port,"mrdata[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.15 ) } -clock "VCLK"
find(port,"mtab")
set_input_delay { REGDELAY + ( clk_logic_period * 0.15 ) } -clock "VCLK"
find(port,"mahb")
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"mip1b[*]")

```

```

set_input_delay { 0.00 } -clock "VCLK" find(port,"icsize[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"ocsize[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ictag3do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icw3do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icv3do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ictag2do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icw2do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icv2do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ictag1do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icw1do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icv1do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ictag0do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icw0do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"icv0do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"iclvl3do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"iclvl2do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"iclvl1do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"iclvl0do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"octag3do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocw3do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocv3do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"octag2do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocw2do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocv2do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"octag1do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocw1do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocv1do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"octag0do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocw0do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.50 ) } -clock "VCLK"
find(port,"ocv0do")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"

```

```

find(port,"oclv13do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"oclv12do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"oclv11do[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"oclv10do[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"enspecialkram")
set_input_delay { 0.00 } -clock "VCLK" find(port,"kram0size[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"kram0do[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"kram1size[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"kram1do[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"krom0size[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"krom0do[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"krom0vldrst")
set_input_delay { 0.00 } -clock "VCLK" find(port,"krom1size[*]")
set_input_delay { REGDELAY + ( clk_logic_period * 0.66 ) } -clock "VCLK"
find(port,"krom1do[*]")
set_input_delay { 0.00 } -clock "VCLK" find(port,"krom1vldrst")
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"mrstib")
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"dsclk")
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"dsdi")
set_input_delay { REGDELAY + ( clk_logic_period * 0.75 ) } -clock "VCLK"
find(port,"mbkptb")

set_load -pin_load 0.5 all_outputs()

/* Set max_fanout so timing can be characterized for toolkit */
set_max_fanout 1 all_inputs() - find(clock, "")

/* Ensure all outputs are buffered */
set_max_fanout default_max_fanout current_design
set_fanout_load default_max_fanout all_outputs()

/* to improve timing on illegalInst_ied, flatten CF4CpuIfpEarlyDecode: */
set_flatten true -design find(design,"CF4CpuIfpEarlyDecode") -effort medium

```



# INDEX

## A

- AATR, 11-26
- ABLR/ABHR, 11-26
- Address map
  - CPU, 1-7
- Addressing
  - modes, 1-12
  - variant, 11-8
- Architectural summary, 1-3
- Architecture
  - virtual memory management, 10-1

## B

- BDM
  - address attribute register, 11-16
  - command
    - format, 11-33
    - sequence diagrams, 11-34
    - set descriptions, 11-35
  - command set summary, 11-32
  - extension words as required, 11-33
  - Motorola-recommended pinout, 11-68
  - receive packet format, 11-30
  - serial interface, 11-29
  - transmit packet format, 11-31
- BIST
  - core ports, 12-19
  - general, 12-17
  - memory
    - clock determination, 12-27
    - controllers, 12-18
    - data retention, 12-26
  - modify ROM signature script, 12-23
  - power analysis, 12-20
  - ROM algorithm, 12-22
  - staging of memories, 12-20
  - test modes, 12-25
  - testing algorithms, 12-21
  - timing
    - cycles, 12-26
    - diagrams, 12-28
- Branch instruction execution timing, 6-28
- Buffers
  - cache push and store, 8-43
- Bus
  - arbitration, 9-16
  - basic cycles, 9-8

- controller system, 8-18
- pipelined cycles, 9-9

## C

- Cache
  - accesses, 8-39
  - buffer bus operation, 8-43
  - caching modes, 8-38
  - coherency, 8-42
  - control register, 8-46
  - copyback mode, 8-39
  - data state transitions, 8-53
  - filling, 8-42
  - inhibited accesses, 8-39
  - instruction state transitions, 8-52
  - locking, 8-44
  - management, 8-50
  - memory accesses for maintenance, 8-42
  - operation
    - general, 8-35
    - summary, 8-52
  - optimizing recommendation, 8-33
  - organization, 8-33
  - overview, 8-32
  - protocol, 8-40
  - push and store buffers, 8-43
  - pushes, 8-42
  - read hit, 8-41
  - read miss, 8-41
  - registers, 8-45
  - registers, access control, 8-48
  - start-up, 8-34
  - write hit, 8-42
  - write miss, 8-41
  - write-through mode, 8-39
- CCR, 2-5
- CF4eTW architecture example, 12-7
- ColdFire core status register, 2-8
- ColdFire debug
  - Revision B, 11-66
  - Revision C, 11-67
- Core
  - features, 1-1
  - implementation block diagram, 1-2
  - overview, 1-1
  - supervisor status register, 5-7

# INDEX

Core interface  
  address and data phase interactions, 9-10  
  basic bus cycles, 9-8  
  bus arbitration, 9-16  
  CF4e pin characteristics, 9-2  
  ColdFire master bus, 9-6  
  data size operations, 9-12  
  line transfers, 9-13  
  M-Bus operation, 9-8  
  M-Bus signals, 9-6  
  pipelined bus cycles, 9-9  
  signals, 9-1  
  timing characteristics, 13-1  
CPU address map, 1-7

## D

Data cache state transitions, 8-53  
Data organization  
  registers, 1-9  
Data representation  
  EMAC, 1-11  
Debug  
  attribute trigger register, 11-13  
  background mode (BDM), 11-27  
  breakpoint operation theory, 11-55  
  C definition of PSTDDATA outputs, 11-58  
  ColdFire  
    history, 11-65  
    Revision C, 11-67  
  Coldfire  
    Revision B, 11-66  
  concurrent BDM and processor operation, 11-58  
  configuration/status register, 11-17  
  CPU halt, 11-28  
  data breakpoint/mask registers, 11-19  
  dump memory block, 11-41  
  emulator mode, 11-57  
  fill memory block, 11-43  
  force transfer acknowledge, 11-47  
  interrupts and requests (emulator mode), 11-67  
  no operation, 11-46  
  overview, 11-1  
  PC breakpoint ASID register, 11-26  
  program counter breakpoint/mask registers, 11-20  
  programming model  
    address attribute trigger register (AATR), 11-26  
    address breakpoint registers (ABLR, ABHR), 11-26  
    general, 11-10  
    trigger definition register (TDR), 11-23  
  read  
    A/D register, 11-36  
    control register, 11-49  
    memory location, 11-38

    register, 11-53  
  real-time trace support, 11-55  
    general, 11-5  
    processor halted, 11-9  
    processor stopped, 11-9  
  registers  
    address attribute (AATR), 11-26  
    address breakpoint (ABLR, ABHR), 11-26  
    trigger definition (TDR), 11-24  
  resume execution, 11-45  
  Revision A shared resources, 11-13  
  signal descriptions  
    general, 11-3  
    processor status/debug data, 11-4  
  supervisor instruction set, 11-64  
  taken branch, 11-8  
  trigger definition register, 11-21  
  user instruction set, 11-59  
  write  
    control register, 11-52  
    memory location, 11-39  
    register, 11-54  
    writeA/D register, 11-37  
Debugging in a virtual environment, 10-7  
DSCLK, 11-3

## E

EMAC  
  data representation, 1-11, 5-13  
  instruction  
    execution times, 6-28  
    set summary, 5-12  
  memory map/register set, 5-6  
  multiply-accumulate unit, 5-1  
  OEP sequence stalls, 6-13  
  programming model, 2-6  
Exception processing  
  overview, 7-1  
  precise faults, 7-8  
  processor exceptions, 7-5  
  sack frame definition, 7-4  
  supervisor/user stack pointers, 7-3  
Exceptions, processor, 7-5  
Execution locations, instruction, 6-18  
Execution times  
  branch instruction, 6-28  
  EMAC instruction, 6-28  
  FPU instruction, 6-30  
  instruction, 6-21  
  miscellaneous, 6-27  
  miscellaneous instruction, 6-27  
  MOVE instruction, 6-23  
  one-operand, 6-24  
  two-operand, 6-25

# INDEX

## F

Fault-on-fault halt, 11-28

### FPU

- computational accuracy, 4-11
  - conditional testing, 4-16
  - control register, 4-8
  - data registers, 4-8
  - data types, 4-4
    - denormalized numbers, 4-5
    - infinities, 4-5
    - normalized numbers, 4-4
    - not-a-number, 4-5
    - zeros, 4-4
  - exceptions
    - arithmetic, 4-20
    - branch/set on unordered (BSUN), 4-21
    - divide-by-zero (DZ), 4-25
    - general, 4-19
    - inexact result (INEX), 4-25
    - input
      - denormalized number (IDE), 4-22
      - not-a-number (INAN), 4-22
    - operand error (OPERR), 4-23
    - overflow (OVFL), 4-23
    - state frames, 4-26
    - underflow (UNFL), 4-24
  - floating-point data formats, 4-3
  - instruction
    - address register, 4-11
    - execution times, 4-30, 6-30
  - instructions
    - overview, 4-28
  - notational conventions, 4-2
  - operand data formats and types, 4-3
  - overview, 4-1
  - post processing, 4-15
  - programmer's model, 4-7
  - programming model, 2-6
    - differences, 4-31
  - results
    - intermediate, 4-11
    - rounding, 4-12
  - signed-integer data formats, 4-3
  - status register, 4-9
  - underflow, round, overflow, 4-16
- FPU-specific OEP stalls, 6-14

## H

Halt, fault-on-fault, 11-28

## I

### Instruction

- execution locations, 6-18

execution times, 6-21, 6-27

Instruction cache state transitions, 8-52

### Instruction set

- fetch pipeline, 6-4
- overview, 1-13
- summary, 1-16

Integer data formats, memory, 1-10

## K

K-Bus signal connections, 8-8

## L

Limited superscalar OEP, 6-9

### Local memory

- connection specification, 8-8
- interactions between modules, 8-7
- K-Bus array signal connections, 8-8
- overview, 8-1
- SRAM overview, 8-22
- two-stage pipelined local bus (K-Bus), 8-5

## M

### MAC

- fractional operation mode, 5-9
- general operation, 5-3
- introduction, 5-2
- mask register, 5-11
- opcodes, 5-13
- overview, 5-1
- status register, 5-6

MBAR, 2-10

### M-Bus

- interrupt support, 9-18
- reset operation, 9-18

MC680x0 differences, 4-31

### Memory

- accesses for cache maintenance, 8-42
- integer data formats, 1-10

### MMU

- architecture
  - features, 10-2
  - location, 10-2
- architecture implementation
  - access, 10-4
  - access error stack frame, 10-5
  - ACR address improvements, 10-6
  - changes to ACRs and CACR, 10-6
  - expanded control register space, 10-6
  - general, 10-3
  - instruction and data cache addresses, 10-4
  - precise faults, 10-4
  - supervisor protection, 10-7

# INDEX

- supervisor/user stack pointers, 10-5
- virtual memory references, 10-4
- virtual mode, 10-4
- features, 10-1
- instructions, 10-22
- MMU definition
  - base address register, 10-11
  - control register, 10-12
  - effective address attribute determination, 10-9
  - fault, test, or TLB address register, 10-15
  - functionality, 10-10
  - general, 10-9
  - memory map, 10-11
  - operation, 10-18
  - operation register, 10-13
  - organization, 10-11
  - read/write tag and data entry registers, 10-15
  - status register, 10-14
  - TLB, 10-17
- MMU implementation
  - general, 10-19
  - TLB address fields, 10-20
  - TLB locked entries, 10-21
  - TLB replacement algorithm, 10-20
- MOVE
  - instruction
    - timing, 6-23
- MOVEC instruction, 11-49
- O**
- OEP
  - EMAC-specific sequence stalls, 6-13
  - general, 6-6
  - instruction folding, 6-9
  - sequence-related stalls, 6-11, 6-14
  - V4 conceptual model, 6-6
- Operand memory sequence-related stalls, 6-16
- P**
- Pinout, Motorola-recommended BDM, 11-68
- Pipelines
  - instruction fetch, 6-4
  - operand execution, 6-6
- PULSE instruction, 11-7
- R**
- Registers
  - AATR, 11-13, 11-26
  - ABLR/ABHR, 11-15
  - access control (ACR0–ACR3), 2-10
  - address (A6–A0), 2-4
  - BAAR, 11-16
  - BDM address attribute, 11-16
  - cache, 8-45
  - cache access control, 8-48
  - cache control (CACR), 2-10, 8-46
  - condition code (CCR), 2-5
  - CSR, 11-17
  - data (D7–D0), 2-4
  - data breakpoint/mask, 11-19
  - data organization, 1-9
  - debug
    - ABLR/ABHR, 11-26
    - attribute trigger, 11-13
    - configuration/status, 11-17
    - PC breakpoint AISD, 11-26
    - TDR module, 11-24
  - F-P
    - control, 4-8
    - instruction address, 4-11
    - status, 4-9
  - MAC
    - mask, 5-11
    - status, 5-6
  - MASK, 2-5
  - MBAR, 2-10
  - MMU
    - base address, 10-11
    - control, 10-12
    - fault, test, or TLB address, 10-15
    - operation, 10-13
    - read/write tag and data entry, 10-15
  - PBR, 11-20
  - program counter, 2-5
  - programming model table, 2-12
  - RAM base address (RAMBAR0/RAMBAR1), 2-10
  - RAREG/RDREG, 11-36
  - RCREG, 11-49
  - RDMREG, 11-53
  - read
    - A/D, 11-36
    - control, 11-49
  - read debug module, 11-53
  - ROM base address, 8-29
  - ROM base address (ROMBAR0/ROMBAR1), 2-10
  - SIM, base address, 2-10
  - SR, 2-8
  - status, 2-8, 2-8
  - TDR, 11-21
  - trigger definition, 11-21
  - user programming model, 2-3
  - user stack pointer, 2-5
  - VB, 2-9
  - vector base, 2-9, 2-9, 7-2
  - WAREG/WDREG, 11-37
  - WCREG, 11-52



# INDEX

WDMREG, 11-54  
write control, 11-52  
write debug module, 11-54  
XTDR, 11-23  
ROM  
base address registers, 8-29  
initialization, 8-31  
operation, 8-28  
overview, 8-28  
programming model, 8-29  
ROMBAR power management programming, 8-32

## S

SRAM  
initialization, 8-25  
initialization code, 8-26  
operation, 8-23  
overview, 8-22  
programming model, 8-23  
Stalls  
EMAC-specifics, 6-13  
OEP, 6-11, 6-16  
Status register, 2-8  
STOP instruction, 11-9, 11-28  
Supervisor programming model, 2-7  
Supervisor/user stack pointers, 2-9  
System bus controller, 8-18

## T

Test controller  
MTMOD encodings, 12-32  
Test features  
CF4e core  
inputs, 12-8  
outputs, 12-11  
noncore  
inputs, 12-13  
outputs, 12-15  
scan chains  
block diagram, 12-3  
core, 12-2  
general, 12-2  
wrapper, 12-2  
timing, 12-8  
wrapper  
block diagram, 12-7  
cells, 12-5  
general, 12-3  
Timing  
branch instruction execution, 6-28  
core interface characteristics, 13-1  
MOVE instructions, 6-23  
one-operand, 6-24

two-operand, 6-25  
Transfers, line, 9-13

## V

V4  
basic pipeline strategy, 6-1  
OEP  
conceptual model, 6-6  
summary, 6-16  
programming model, 1-5  
Variant address, 11-8  
Vector base register, 2-9, 2-9, 7-2  
Virtual memory  
access error stack frame additions, 10-8  
architecture processor support, 10-7  
management architecture, 10-1  
precise faults, 10-8  
supervisor/user stack pointers, 10-8

## W

WDDATA execution, 11-7

# INDEX

Introduction	1
Registers	2
Instructions	3
FPU	4
EMAC	5
Execution Timing	6
Exceptions	7
Local Memory	8
Core Interface	9
MMU	10
Debug Module	11
Test	12
Timing Constraints	A
Index	IND

1	Introduction
2	Registers
3	Instructions
4	FPU
5	EMAC
6	Execution Timing
7	Exceptions
8	Local Memory
9	Core Interface
10	MMU
11	Debug Module
12	Test
A	Timing Constraints
IND	Index