TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 5
Processor-Local Memories

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 7
Test

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

SECTION 1
INTRODUCTION

This manual summarizes the operation and use of the Version 3 ColdFire processor complex reference design. The processor complex design includes the processor core, the debug module, high-speed processor local bus and associated memory controllers plus interface bus controller. Collectively, this reference design is known as CFxRef, where x defines the appropriate version of the microarchitecture. This document details the microarchitecture, functionality, core interface and test strategy for the Version 3 ColdFire reference design. Specific deployments of the CF3Ref design are named by a notation which identifies the presence of optional functional blocks. As examples, the CF3 design includes the basic CF3Ref design without the optional Multiply-Accumulate Unit (MAC), while the CF3M implementation includes the MAC unit.

The ColdFire microprocessor architecture provides new levels of price and performance to the emerging cost-sensitive, high-volume embedded markets, especially in the area of consumer products. Based on the concept of a variable-length RISC technology, ColdFire combines the architectural simplicity of conventional 32-bit RISC with a memory-saving, variable-length instruction set. In defining the ColdFire architecture for embedded processing applications, Motorola has incorporated a RISC-based processor design for peak performance and a simplified version of the variable-length instruction set found in the M68000 Family for maximum code density. The result is a family of 32-bit microprocessors ideally suited for those embedded applications requiring high performance in a small core size.

The ColdFire performance roadmap, announced in 3Q96, defines a series of microarchitecture versions, which when coupled with improved process technology provides increasing levels of performance, up to 300 Dhrystone 2.1 MIPS by the year 2001. The Version 3 processor represents the early-midpoint of the roadmap providing a performance of approximately 70 Dhrystone 2.1 MIPS in a 90 MHz implementation using 0.35 micron semiconductor process technology. This performance metric can also be expressed as 0.78 Dhrystone 2.1 MIPS per MHz for the Version 3 ColdFire core, assuming a cache size of 4 KBytes or larger.

1.1 WHY COLDFIRE!

The ColdFire family of 32-bit microprocessors provides balanced system solutions to a variety of embedded markets. The following list details a number of the basic philosophies applicable to all ColdFire designs:

- The Instruction Set Architecture (ISA) and resulting code density directly translate in lower memory costs, both for internal and external memory subsystems

- Small, fully-synthesizable processor complexes
  - Developments are on track with performance roadmap reaching 300 MIPS by 2001
  - 100% synthesizable design and use of compiled memory arrays plus function-level parameterization allow system designers to easily define CPU configurations
  - Can easily move to any process technology targeting different operating voltages and frequencies
  - Supports cost-effective integration capabilities
- Modular system architecture
  - A hierarchy of system buses provides layers of bandwidth and supports an efficient partitioning of the optional, on-chip modules
  - CFxRef designs support configurable processor-local memories, e.g., cache, RAM, ROM, with sizes from [0 - 32 KBytes]
  - Standard Motorola peripheral bus promotes reuse of synthesizable modules
- Full-featured debug module
  - Common debug architecture does not require traditional connection to external bus, and yet provides background debug mode (BDM) capabilities plus real-time trace and debug functionality
  - Standard interface used in Motorola parts and completely embedded, customer-specific designs using 3rd-party developer tools
- Bridge from the 19-year M68000 Family legacy
  - Reuse of 68K assembly language simplified through conversion tool
  - Leverages system designer and programmer knowledge base
  - Leverages mature 3rd-party developer tools

SECTION 2
ARCHITECTURAL OVERVIEW

The following block diagram depicts the standard ColdFire microprocessor configuration. The hierarchical bus structure (Processor-Local, Master, Slave and External Buses) provides varying layers of data bandwidth and supports an efficient partitioning of the optional, on-chip modules. This hierarchy of buses are also known by their abbreviated names: the processor-local bus is the K-Bus, the Master Bus is the M-Bus, the Slave Bus is the S-Bus and finally, the External Bus is the E-Bus. The modular system architecture is readily apparent. The ColdFire processor complex reference design is defined by the CFxCore level of hierarchy. The CFxCoreKmem boundary includes the core design plus the required processor-local memories for a given design.

Within the CFxCore, the processor is connected via a local, high-speed bus to a number of memory controllers and a bus controller. The processor-local memories include cache storage, as well as blocks of RAM and ROM. The memory controllers contained within the CFxCore design all support a range of sizes, allowing the system designer the ability to specify the optimum memory organization for a given application. Transfers on the processor-local bus are controlled by the K2M bus controller, which is also responsible for initiation and control of all accesses onto the next-level system bus, the Master Bus. The processor-local bus is designed to provide a maximum amount of bandwidth from these high-speed memories to support the processor's efficient execution of instructions.

The CFxCore plus all other bus masters are connected at the microprocessor level via the Master Bus, which provides the primary interface between the ColdFire core and the other system-level components. Any device which can initiate bus cycles is typically connected to the Master Bus. Example modules include Direct-Memory Access devices (DMA), or another ColdFire processor complex. The Master Bus is typically connected to a System Bus Controller (SBC) which provides two interfaces: one to a simple, on-chip Slave Bus, and another to an application-specific External Bus. The Slave Bus generally is connected to any number of standard peripheral modules, including functions like timers, UARTs and other serial communication devices, parallel ports, etc. The use of a standard Motorola-defined bus protocol promotes the reuse of these synthesizable modules. The specific implementation and protocol details of the External Bus generally vary widely, depending on system requirements.

In many implementations, the process technology may allow the processor complex to operate at a higher frequency compared to the rest of the microprocessor. The CFxCore design supports this notion of multiple *clock domains*, and features a standard implementation which allows the core to be operated at any integer multiplier (n = 1,2,3,...) faster than the rest of the design. For multiple clock domains, the boundary is the Master

Bus, i.e., the processor complex operates at the higher frequency, while the Master Bus and the remainder of the microprocessor operate at the slower speed. This design approach provides a well-defined and easy-to-use clock boundary, which simplifies interface design and timing and eases production test complications. This topic is covered in more detail in **Section 3: ColdFire Core.**

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

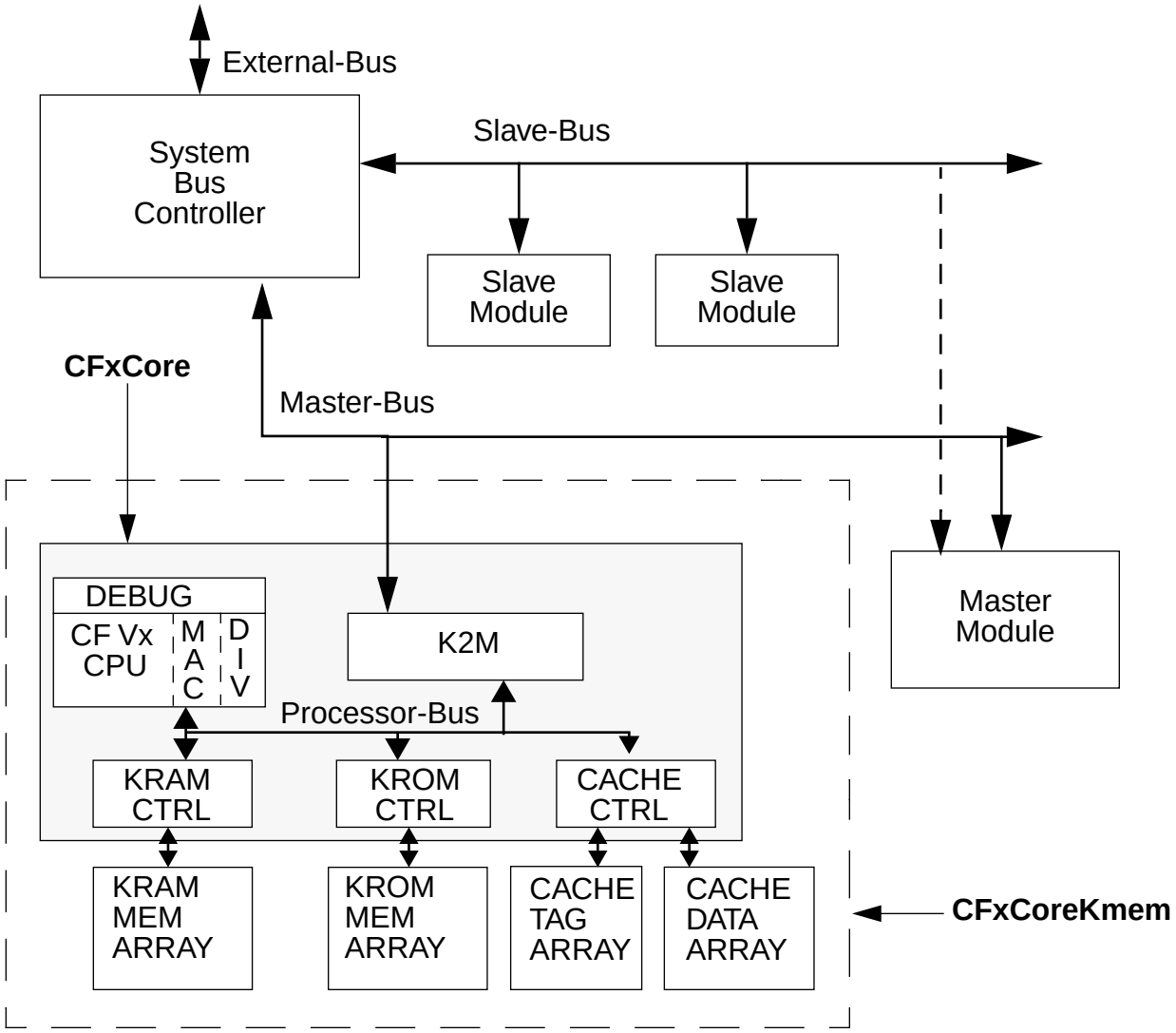


Figure 2-1. Generic ColdFire System Block Diagram

All ColdFire processor cores consist of two independent, decoupled pipeline structures to maximize performance while minimizing core size. The Instruction Fetch Pipeline (IFP) prefetches instructions, while the Operand Execution Pipeline (OEP) decodes the instructions, fetches the required operands and then executes the specified functions. Since the IFP and OEP are decoupled by an instruction buffer that serves as a FIFO queue, the IFP can prefetch instructions in advance of their actual use by the OEP, thereby minimizing time stalled waiting for the variable-length instructions. Consider the following Version 3 ColdFire processor block diagram:

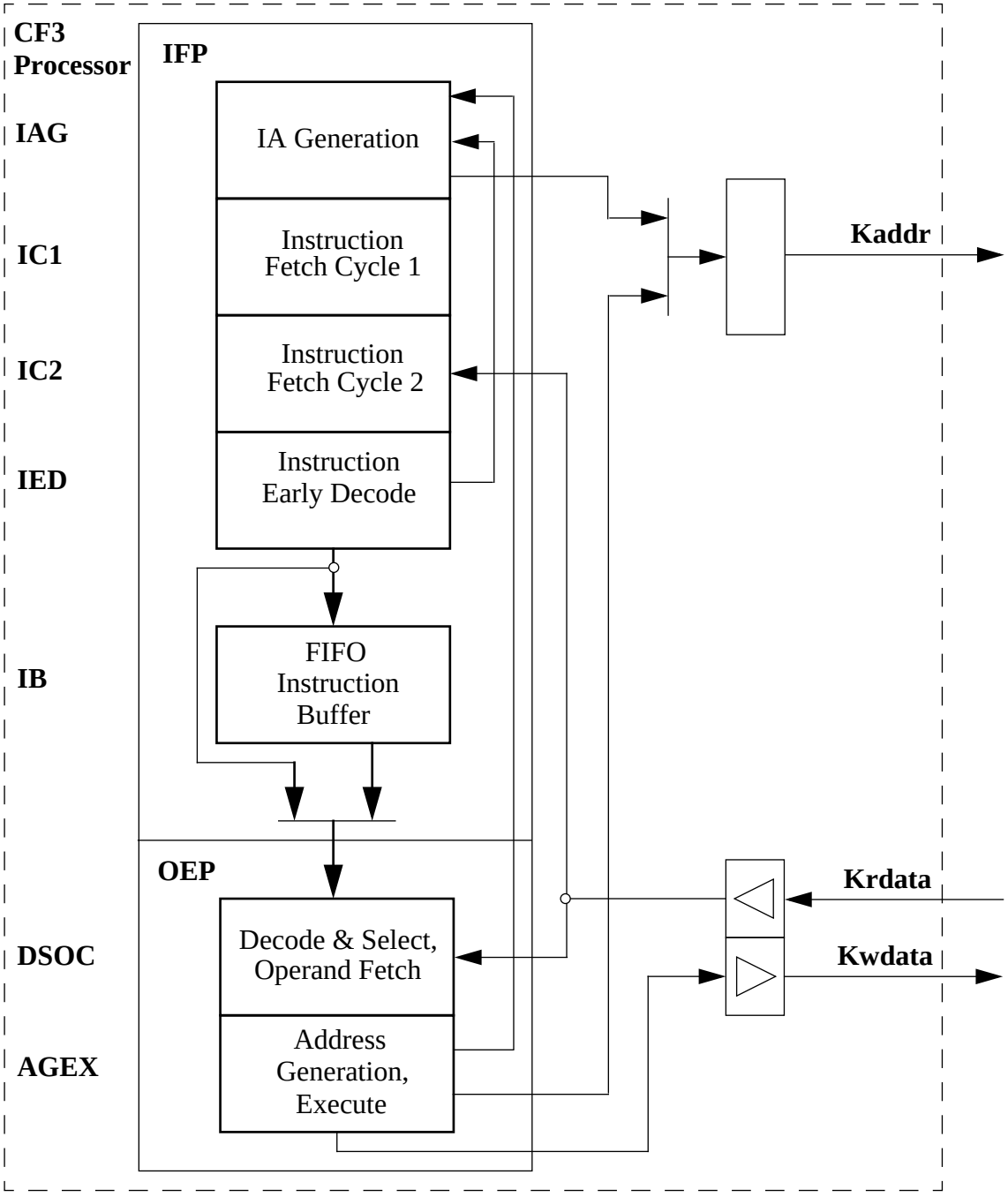


Figure 2-2. Version 3 ColdFire Processor Block Diagram

Here the processor's non-Harvard architecture is readily apparent. The processor's connection to the local bus (K-Bus) is defined by the reference address (**Kaddr**), and two unidirectional data buses, **Krdata** (read data) and **Kwdata** (write data), which transfer instructions and operands into the processor core, or to the destination memories. This structure minimizes the core size *without* compromising performance to a large degree.

The IFP is a four-stage pipeline for prefetching instructions and partially decoding them, while the OEP is implemented in a two-stage pipeline featuring a traditional RISC datapath with a dual-read-ported register file feeding an arithmetic/logic unit.

Subsequent sections provide further details on the microarchitecture of the Version 3 ColdFire processor complex, along with a description of the Master Bus interface.

SECTION 3
VERSION 3 CORE

3.1 INTRODUCTION

This section details the CF3Core interface and provides an overview of the functional operation of the Master Bus (M-Bus).

Note that the CF3Core pin naming definition uses all *lower case* signal names, due to various tool limitations. However, most of the documentation presented in this manual, except for **Section 3.2: CF3Core Signals**, follows a convention with *upper case* names. It is important to note that these conventions are meant to be equivalent, i.e., port signal "**xyz**" is the same as signal "**XYZ**". Additionally, the use of a "b" suffix in the pin naming definition indicates an active-low signal, while the rest of the documentation uses an overbar, i.e., signal "**xyzb**" (**xyz** bar) is the same as "$\overline{\textbf{XYZ}}$".

3.2 CF3CORE SIGNALS

This section details the pin name definition and pin order for the Version 3 ColdFire reference design, specifically, the CF3Core design. This core is typically deployed in a configuration where the processor-local memories are **not** included in the design to provide the system designer with the ability to configure and size those memories for a given application.

A generic block diagram of a Version 3 ColdFire design is shown below, where the CF3Core is represented by the shaded area.

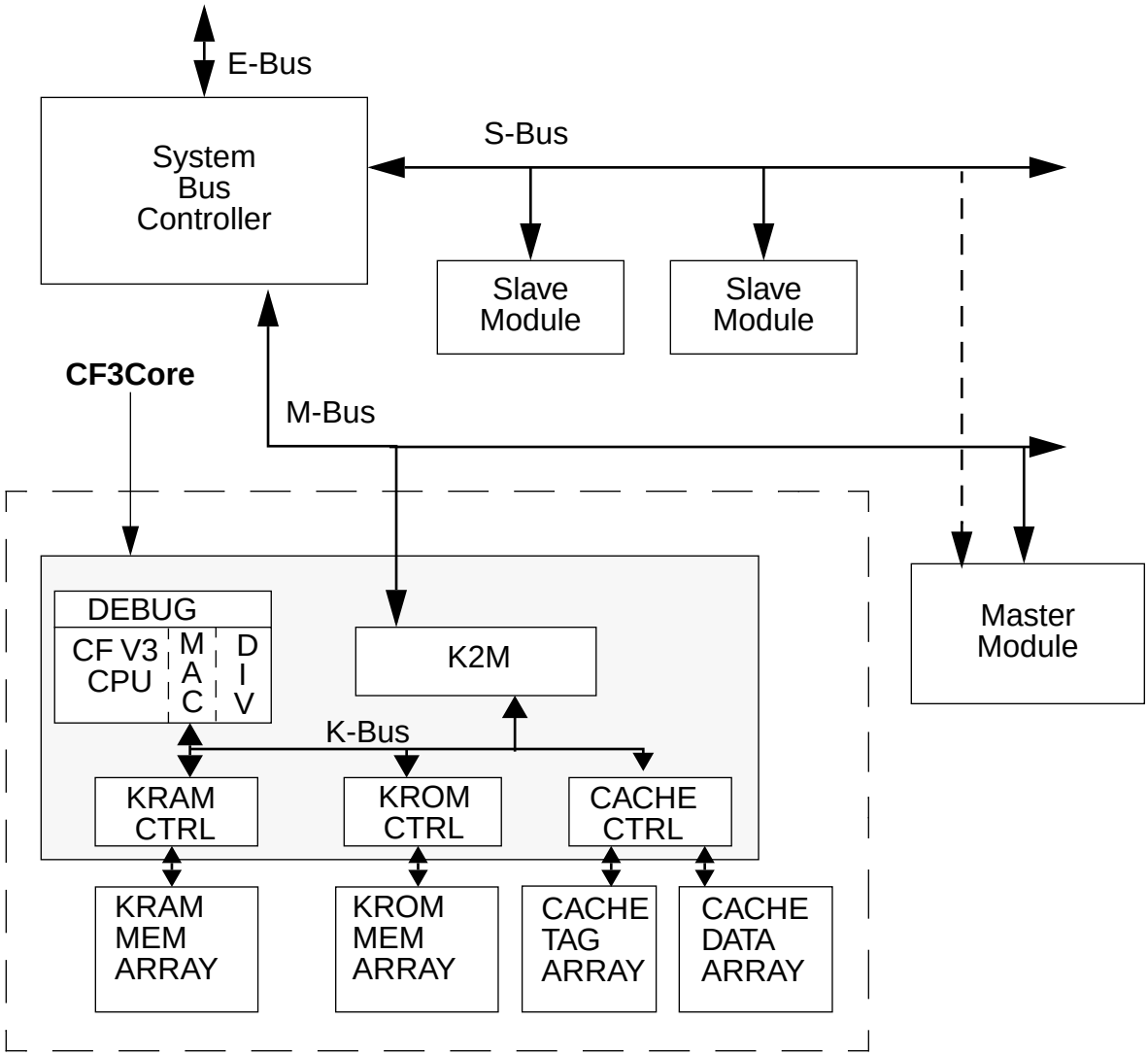


Figure 3- 1. Generic Version 3 ColdFire Block Diagram

This pin name and ordering definition is used in the following “views” of the CF3Core design:

1. Behavioral RTL model used as input to synthesis and other implementation tools
2. C model used in encrypted form by the ISD toolkit
3. Gate-level netlist
4. Bus functional model
5. Interface structure for any cycle-based models

It should be noted that the CF3Core/KBus configurable memory interface is not actually modeled in the bus functional or cycle-based models. For these models, the behavior of this interface is described at a different abstraction level, but the pin list remains consistent across all views.

The CF3Core pin list can be broadly classified into the following groups:

Pins {1-41}	**CF3Core outputs**
Pins {1-9}	M-Bus outputs
Pins {10-15}	Debug outputs
Pins {16-21}	Test outputs
Pins {22-41}	Outputs to K-Bus memories

Pins {42-88}	**CF3Core inputs**
Pins {42-46}	M-Bus inputs
Pins {47-50}	Debug and configuration inputs
Pins {51-57}	Test inputs
Pins {58-86}	Inputs from K-Bus memories + memory configuration definitions
Pins {87-88}	Clock inputs

All K-Bus memories are specified to be synchronous devices, where the CF3Core outputs are *next-state* values which are registered within the memory device.

The pin specification and ordering for the CF3Core is detailed in the following table. The use of a "*b*" suffix in the name indicates an active-low signal. Bus widths are specified using a vector notation, while no entry in this column indicates a scalar (1-bit) signal.

The following notes are applicable to certain CF3Core signals:

1. In general, most of the M-Bus and debug input signals are driven directly into input capture registers within the CF3Core design. The **mahb** and **mtab** signals are driven into combinational logic before being registered, so these inputs have a greater setup timing requirement.

The **mrdata[31:0]** input capture register is only loaded by the termination of an M-Bus data phase. The **miplb[2:0]** and **mrstib** input signals are routed into free-running input capture registers, while the **dsclk**, **dsdi** and **mbkptb** input signals are routed into two levels of free-running registers which effectively serve as synchronizers.

All M-Bus and debug output signals are driven directly from registers within the CF3Core design.

2. For CF3Core designs, it is necessary to output a clock signal from the microprocessor to the standard ColdFire debug connector so that external emulators can correctly sample the **pst[3:0]** and **ddata[3:0]** output signals. This output clock, typically named **pstclk**, is a derivative of the processor's clock signal and *must be formed external to the CF3Core design* using the following boolean equation:

$$\mathbf{pstclk} = \mathbf{clkfast} \ \& \ \mathbf{enpstclk}$$

where **pstclk** is the output signal, **clkfast** is the processor's clock signal and **enpstclk** is a logical enable, defined by the user-programmed configuration of the debug

module within the CF3Core design.

Table 3-1. CF3Core Pin Specification

No.	Type	Name	Bus Width	Description
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	**mrw**		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	**cpustopb**		Processor is stopped
11	Output	**cpuhaltb**		Processor is halted
12	Output	**enpstclk**		PST/DDATA clock enable
13	Output	**pst**	[3:0]	Processor status
14	Output	**ddata**	[3:0]	Debug data
15	Output	**dsdo**		Development system data output
16	Output	**so**	[31:0]	Core parallel scan outputs
17	Output	**tbso**	[3:0]	Test boundary scan outputs
18	Output	**ucpaddr**	[31:4]	U-Cache push tag address for BIST
19	Output	**ucpdata**	[31:0]	U-Cache push data for BIST
20	Output	**rcpshdrtyk2**		U-Cache push written bit for BIST
21	Output	**rcpshvldk2**		U-Cache push valid bit for BIST
22	Output	**nsentb**		Next-state U-Cache tag enable
23	Output	**nswrttb**		Next-state U-Cache tag write
24	Output	**nswlvt**	[3:0]	Next-state U-Cache tag write level
25	Output	**nsinvat**		Next-state U-Cache tag invalidate all
26	Output	**nsrowst**	[8:0]	Next-state U-Cache tag address

Table 3-1. CF3Core Pin Specification

No.	Type	Name	Bus Width	Description
27	Output	**nsaddrt**	[31:9]	Next-state U-Cache tag data
28	Output	**nssw**		Next-state U-Cache tag written bit
29	Output	**nssv**		Next-state U-Cache tag valid bit
30	Output	**nsendb**		Next-state U-Cache data enable
31	Output	**nswrtdb**	[3:0]	Next-state U-Cache data write level
32	Output	**nswtbyted**	[3:0]	Next-state U-Cache data byte write
33	Output	**nsrowsd**	[10:0]	Next-state U-Cache data address
34	Output	**nscwrdata**	[31:0]	Next-state U-Cache write data
35	Output	**kramaddr**	[14:2]	Next-state KRAM address
36	Output	**kramdi**	[31:0]	Next-state KRAM write data
37	Output	**kramweb**	[3:0]	Next-state KRAM write enable
38	Output	**kramcsb**		Next-state KRAM chip select
39	Output	**kramdata**	[31:0]	Next-state KRAM data for BIST
40	Output	**kromaddr**	[14:2]	Next-state KROM address
41	Output	**kromcsb**		Next-state KROM chip select
42	Input	**mrdata**	[31:0]	M-Bus read data
43	Input	**mtab**		M-Bus transfer acknowledge
44	Input	**mahb**		M-Bus address hold
45	Input	**miplb**	[2:0]	M-Bus interrupt request priority level
46	Input	**mrstib**		M-Bus reset
47	Input	**dsclk**		Development system clock
48	Input	**dsdi**		Development system data input
49	Input	**mbkptb**		Development system breakpoint
50	Input	**en000iack**		Enable 68000-style IACK cycles
51	Input	**bistplltest**		BIST or PLL test mode
52	Input	**si**	[31:0]	Core parallel scan inputs
53	Input	**se**		Core parallel scan enable
54	Input	**tbsi**	[3:0]	Test boundary scan inputs

Table 3-1. CF3Core Pin Specification

No.	Type	Name	Bus Width	Description
55	Input	**tbsei**		Test boundary scan enable - inputs
56	Input	**tbseo**		Test boundary scan enable - outputs
57	Input	**tbte**		Test boundary pcell test enable
58	Input	**ucsz**	[2:0]	Encoded U-Cache size
59	Input	**ucnoif**		Block instructions from U-Cache
60	Input	**ucnoop**		Block operands from U-Cache
61	Input	**bistmode**		BIST test mode
62	Input	**bisttaglvl**	[1:0]	BIST tag level select
63	Input	**bistdatalvl**	[1:0]	BIST data level select
64	Input	**uctag3do**	[31:9]	U-Cache level 3 tag data output
65	Input	**ucw3do**		U-Cache level 3 written bit output
66	Input	**ucv3do**		U-Cache level 3 valid bit output
67	Input	**uctag2do**	[31:9]	U-Cache level 2 tag data output
68	Input	**ucw2do**		U-Cache level 2 written bit output
69	Input	**ucv2do**		U-Cache level 2 valid bit output
70	Input	**uctag1do**	[31:9]	U-Cache level 1 tag data output
71	Input	**ucw1do**		U-Cache level 1 written bit output
72	Input	**ucv1do**		U-Cache level 1 valid bit output
73	Input	**uctag0do**	[31:9]	U-Cache level 0 tag data output
74	Input	**ucw0do**		U-Cache level 0 written bit output
75	Input	**ucv0do**		U-Cache level 0 valid bit output
76	Input	**uclvl3do**	[31:0]	U-Cache level 3 data output
77	Input	**uclvl2do**	[31:0]	U-Cache level 2 data output
78	Input	**uclvl1do**	[31:0]	U-Cache level 1 data output
79	Input	**uclvl0do**	[31:0]	U-Cache level 0 data output
80	Input	**kramsz**	[2:0]	Encoded KRAM size
81	Input	**encf5307kram**		Enable CF5307-style KRAM
82	Input	**enraptorkram**		Enable Raptor-style KRAM

Table 3-1. CF3Core Pin Specification

No.	Type	Name	Bus Width	Description
83	Input	**kramdo**	[31:0]	KRAM data output
84	Input	**kromsz**	[2:0]	Encoded KROM size
85	Input	**kromvldrst**		KROM valid at reset
86	Input	**kromdo**	[31:0]	KROM data output
87	Input	**mclken**		Clock phase relationship definer
88	Input	**clkfast**		Processor core clock

See **Appendix A: CF3Core Interface Timing Constraints** for detailed information on the synthesis timing budgets for the CF3Core interface signals.

3.3 COLDFIRE MASTER BUS

3.3.1 Introduction

The ColdFire architecture implements a hierarchy of buses to provide the necessary interconnection and bandwidth among the various components (processors, peripherals, etc.) in a system.The Master Bus (M-Bus) is the system interconnect between multiple masters (including processors) and the System Bus Controller (SBC). The System Bus Controller provides additional connectivity to an optional internal Slave Bus (S-Bus) containing on-chip peripheral modules, as well as the external system via the External Bus (E-Bus). The M-, S-, and E-Buses operate with a Motorola-defined bus protocol. Providing this bus protocol support allows integration of devices at any level in the system.

The ColdFire architecture is designed to allow multiple clock frequency domains. The ColdFire processor can be operated at any integer multiple (n:1, where n = 1, 2,...) of the M-Bus clock frequency. The ColdFire processor's M-Bus interface is the boundary from the processor's clock domain to the M-Bus clock domain.

This section presents the M-Bus and its operation. It details specific M-Bus protocols needed to support the multiple clock domains and gives system clocking guidelines.

3.3.2 M-Bus Signals

This section defines the signals required by the M-Bus. Although the timing of all of these signals is referenced to the system clock, the system clock is not considered a bus signal. It is expected that the clock is routed as needed to meet application requirements.

This section describes M-Bus signals as viewed by the Bus Master. Table 3-2 gives a summary of the signals. A brief description of the signal's functionality follows.

Note that an overbar indicates an active-low signal.

Table 3-2. M-Bus Signal Summary

SIGNAL NAME	DIRECTION	DESCRIPTION
MRDATA[31:0]	In	Read Data Bus
$\overline{\text{MAH}}$	In	Address Hold
$\overline{\text{MTA}}$	In	Transfer Acknowledge
$\overline{\text{MRSTI}}$	In	M-Bus Reset
$\overline{\text{MIPL}}$[2:0]	In	Interrupt Priority Level
MADDR[31:0]	Out	Address Bus
$\overline{\text{MAP}}$	Out	Address Phase
$\overline{\text{MDP}}$	Out	Data Phase
MSIZ[1:0]	Out	Transfer Size
MR$\overline{\text{W}}$	Out	Read/Write
MTT[1:0]	Out	Transfer Type
MTM[2:0]	Out	Transfer Modifier
MWDATA	Out	Write Data Bus

The preceding section provided the actual pin names and order for the Version 3 ColdFire core reference design, while this section details the M-Bus operation from a functional perspective.

3.3.2.1 M-BUS READ DATA (MRDATA[31:0]). These unidirectional input signals provide the read data path between the system bus controller and internal masters. The read data bus is 32 bits wide and can transfer 8, 16 or 32 bits of data per bus transfer. During a line transfer, the data lines are time-multiplexed across multiple cycles to carry 128 bits.

3.3.2.2 M-BUS ADDRESS HOLD ($\overline{\text{MAH}}$). This input signal is asserted to indicate that the address and attributes should be held. This signal indicates that the SBC is not ready to accept the address phase of the bus cycle. This signal is also used in bus arbitration situations to halt the master when it does not have the M-Bus.

3.3.2.3 M-BUS TRANSFER ACKNOWLEDGE ($\overline{\text{MTA}}$). This input signal is asserted to indicate the successful completion of a requested bus transfer.

3.3.2.4 M-BUS RESET ($\overline{\text{MRSTI}}$). This input signal directs all M-Bus modules (including the core) to enter reset mode.

3.3.2.5 M-BUS INTERRUPT PRIORITY LEVEL (MIPL[2:0]). These three input signals indicate to the processor that there is a pending interrupt request. Table 3-3 shows the encoding for the **MIPL** signals.

Table 3-3. M-Bus Interrupt Priority Level Encodings

MIPL[2:0]	Interrupt Level
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

3.3.2.6 M-BUS ADDRESS (MADDR[31:0]). During a normal bus cycle, these output signals provide the address of the first item of a bus transfer. **MADDR** is 32 bits wide with all signals being unidirectional.

3.3.2.7 M-BUS ADDRESS PHASE (MAP). This output signal indicates that the address and attributes are being driven and that the address phase of the bus cycle is active.

3.3.2.8 M-BUS DATA PHASE (MDP). This output signal indicates that the data phase of the cycle is active. This means that data is driven by the bus master 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.

3.3.2.9 M-BUS TRANSFER SIZE (MSIZ[1:0]). These output signals indicate the data size for the bus transfer. Refer to Table 3-4 for the bus size encodings.

Table 3-4. M-Bus Transfer Size Encodings - 32-bit Data Bus

MSIZ[1:0]	Transfer Size
00	Longword (4 bytes)
01	Byte (1 byte)
10	Word (2 bytes)
11	Line (16 bytes)

3.3.2.10 M-BUS READ/WRITE (MR$\overline{W}$). This output signal 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.

3.3.2.11 M-BUS TRANSFER TYPE (MTT[1:0]). These output signals indicate the type of access of the current bus cycle. Table 3-5 shows the definition of the transfer type encodings. The alternate master access is used to indicate a non-core master is requesting the transfer.

Table 3-5. M-Bus Transfer Type Encodings

MTT[1:0]	Transfer Type
00	Processor Access
01	Alternate Master Access
10	Processor Emulator Mode Access
11	Acknowledge or CPU Space Access

3.3.2.12 M-BUS TRANSFER MODIFIER (MTM[2:0]). These output signals provide supplemental information for each transfer type. Refer to Table 3-6 for normal transfer encodings and Table 3-7 for processor emulator mode transfer encodings. Table 3-8 shows the encoding for acknowledge or CPU Space accesses. For interrupt acknowledge transfers, the **MTM** signals carry the interrupt level being acknowledged. For CPU space transfers, the **MTM** signals are low.

Table 3-6. M-Bus Transfer Modifier Encodings for MTT = 0-

MTM[2:0]	Transfer Modifier
000	Reserved
001	User Data Access
010	User Code Access
011 - 100	Reserved
101	Supervisor Data Access
110	Supervisor Code Access
111	Reserved

Table 3-7. M-Bus Transfer Modifier Encodings for MTT = 10

MTM[2:0]	Transfer Modifier
000 - 100, 111	Reserved

Table 3-7. M-Bus Transfer Modifier Encodings for MTT = 10

MTM[2:0]	Transfer Modifier
101	Emulator Mode Data Access
110	Emulator Mode Code Access

Table 3-8. M-Bus Transfer Modifier Encodings for MTT = 11

MTM[2:0]	Transfer Modifier
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

3.3.2.13 M-BUS WRITE DATA (MWDATA[31:0]). These unidirectional output signals provide the write data path between an internal master and the system bus controller. The write data bus is 32 bits wide and can transfer 8, 16 or 32 bits of data per bus transfer. During a line transfer, the data lines are time-multiplexed across multiple cycles to carry 128 bits.

3.3.3 M-Bus Operation

The M-Bus is a two-stage, synchronous pipelined bus. This gives it an effective bandwidth rate of up to one transfer per clock.

3.3.3.1 BASIC BUS CYCLES. The bus transaction is split into two phases. The first phase is the address phase. During this phase, the address (**MADDR**) and attribute signals (**MSIZ**, **MRW**, **MTT**, and **MTM**) are driven. The Address Phase signal (**MAP**) is asserted to show that the bus is in the address phase.

The second part of the transaction is the data phase. The Data Phase (**MDP**) signal is asserted to show that the bus is in the data phase and that data transfer may now take place. The data phase stays active until the bus cycle is terminated with a Transfer Acknowledge (**MTA**). On a write cycle, the Write Data Bus (**MWDATA**) is driven for the duration of the data phase. On a read cycle, the Read Data Bus (**MRDATA**) is sampled by the bus master concurrently with **MTA** at the rising clock edge. Figure 3- 4 shows the basic read and write operations.

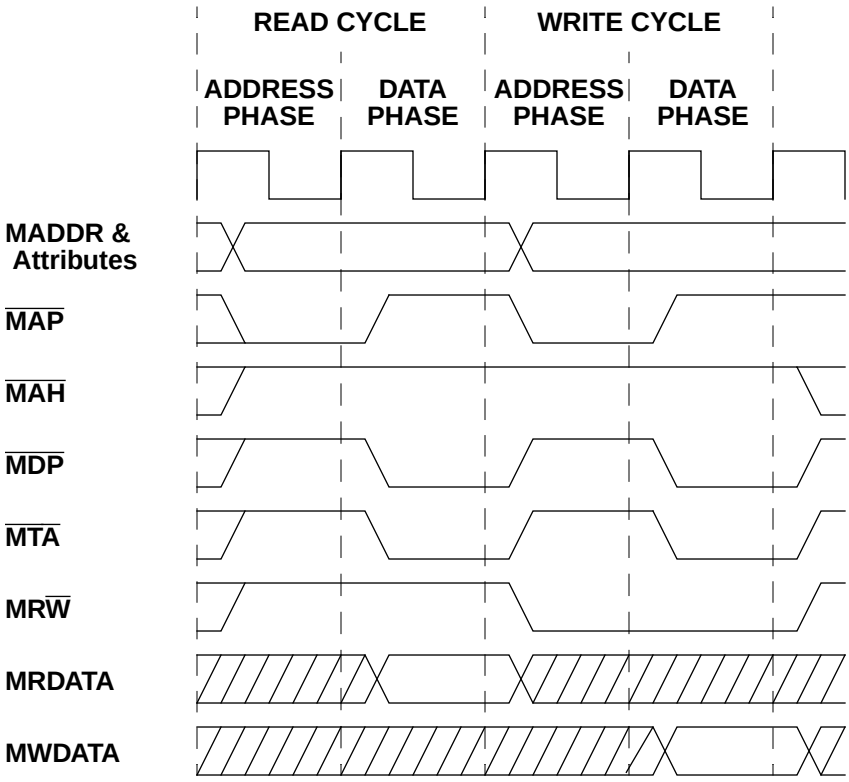


Figure 3- 4. Basic Read and Write Cycles

3.3.3.2 PIPELINED BUS CYCLES. Since the bus is pipelined, it is possible for the address phase of the next bus cycle to become valid while the data phase of the current bus cycle is still valid. It is not possible for the address and data phases of the same bus cycle to be concurrently valid. Figure 3- 5 shows two basic bus cycles that have been pipelined. For illustration purposes, a read and write cycle are used in Figure 3-4. There are no restrictions on cycles being either reads or writes in order for them to be pipelined.

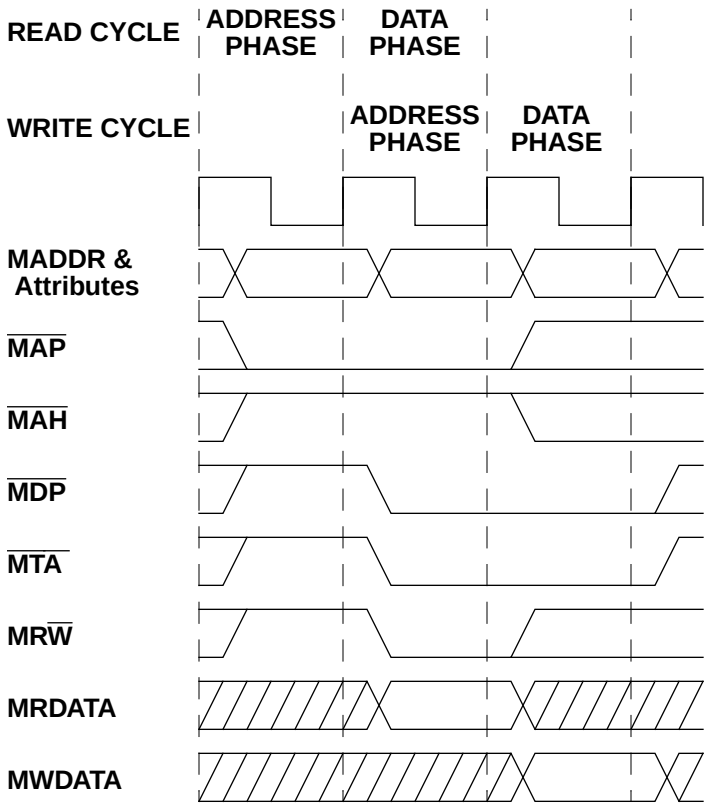


Figure 3- 5. Pipelined Read and Write

3.3.3.3 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 are:

- The address phase is allowed to begin when there is no active address phase.
- The address phase is allowed to end and the data phase to begin when the address hold (**MAH**) signal is not asserted and there is either no active data phase or the active data phase is being terminated.
- The data phase is allowed to end when the cycle is terminated with **MTA**.

There is one special rule that applies only to M-Bus masters that are ColdFire processors operating at the same clock frequency as the M-Bus (i.e., the processor ‘s clock domain and the M-Bus clock domain have the same frequency, the so-called *1X clock mode)*. This special rule is a restriction on the second general rule above:

- For a processor operating in 1X clock mode, the processor’s address phase is allowed to end and the data phase to begin when the address hold (**MAH**) signal is not asserted and 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:

- The bus master is held off (usually for bus arbitration) by asserting the **$\overline{\text{MAH}}$** signal. This assures 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 **$\overline{\text{MTA}}$**.

The implications of the special 1X clock mode rule are:

- 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 the **$\overline{\text{MAH}}$** signal on the last M-Bus transfer acknowledge. This forces the ColdFire processor to hold in its address phase until its data phase has been idle for at least one cycle.
- A simple implementation of this 1X clock mode rule is to connect the **$\overline{\text{MTA}}$** signal from the System Bus Controller to both the **$\overline{\text{MTA}}$** and **$\overline{\text{MAH}}$** inputs ports of the CF3Core design.

Figure 3- 6 shows the **$\overline{\text{MAP}}$** signal asserted during the same clock that **$\overline{\text{MAH}}$** is asserted. The address phase is held until **$\overline{\text{MAH}}$** is negated. At this point, **$\overline{\text{MDP}}$** is asserted to show that the data phase of the first cycle has begun. Since the address for the next bus cycle is available, **$\overline{\text{MAP}}$** remains asserted to indicate that the address phase of the second cycle has begun. One wait state is inserted into the bus cycle by delaying **$\overline{\text{MTA}}$** until the next clock. In this case, **$\overline{\text{MAP}}$** is negated after termination because there is not another address available from the bus master. **$\overline{\text{MDP}}$** is not negated because at termination the address phase of the second cycle transitions to the data phase. Since the termination signal remains asserted, the data phase of the second cycle is only one clock long.

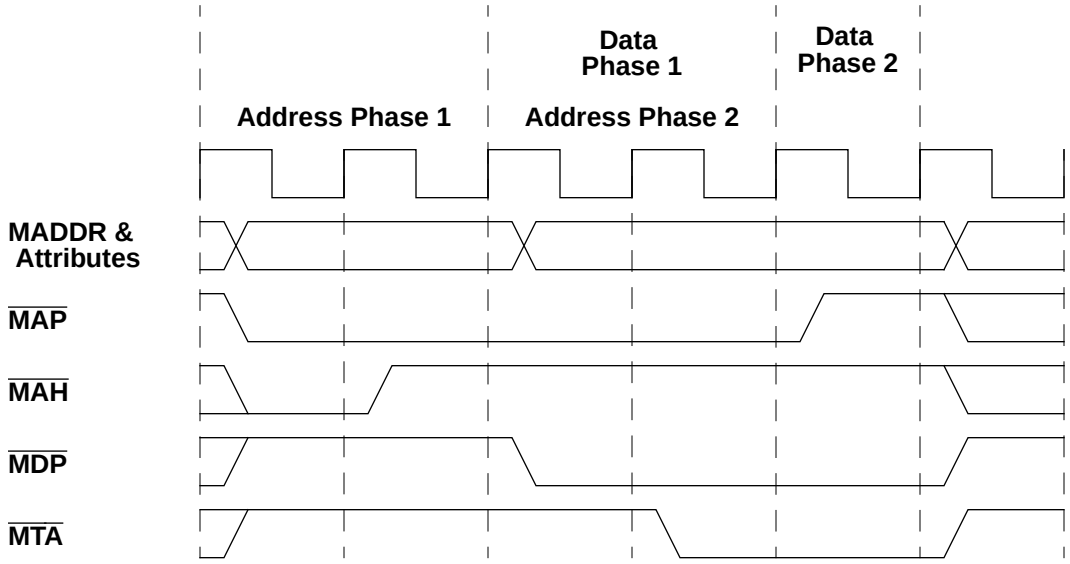


Figure 3- 6. Address Hold Followed by 1- and 0-Wait State Cycles

Figure 3- 7 demonstrates that **MAP** may be generated in the center of the data phase. It also shows that **MAH** 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.

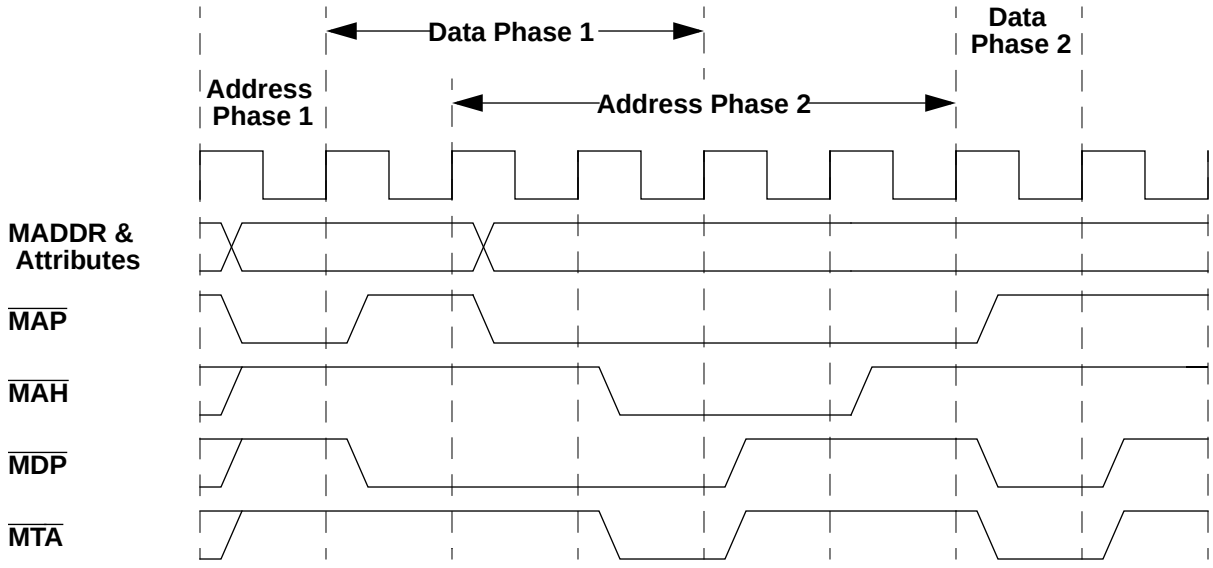


Figure 3- 7. MAP and MAH Generated Mid-Data Phase

Figure 3-6 demonstrates the special rule for 1X clock mode. It shows a 1X clock mode processor in its address phase (Address Phase 2) being held on the last **MTA** of its current data phase (Data Phase 1) by **MAH**.

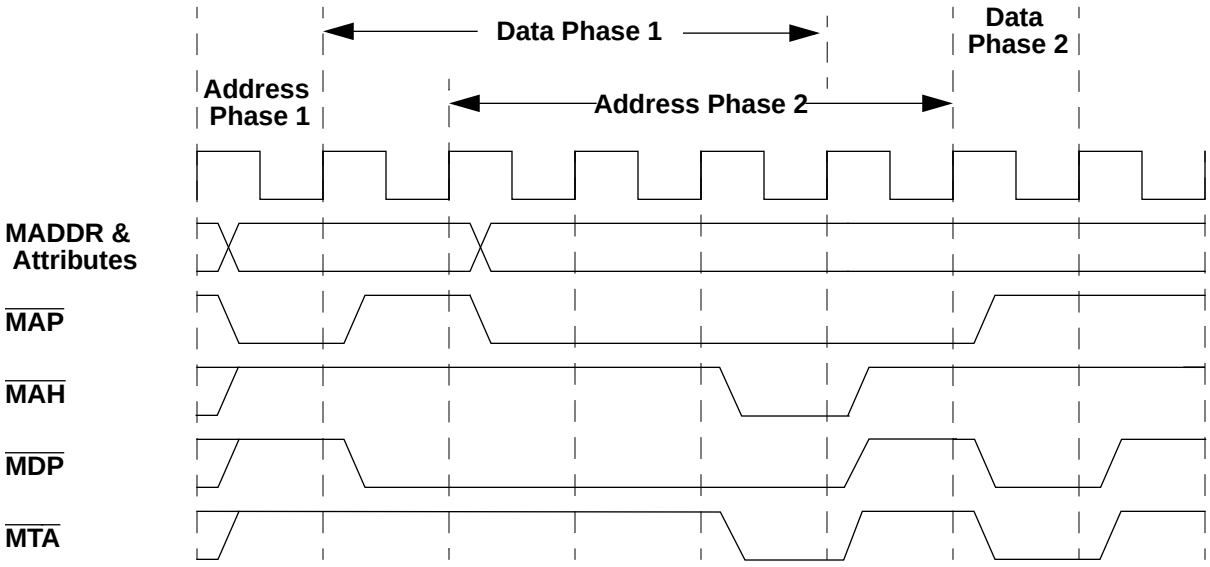


Figure 3- 8. $\overline{\text{MAH}}$ Generation for 1X Clock Mode

3.3.3.4 DATA SIZE OPERATIONS. The processor designates all operands for transfers on a byte-boundary system using the nomenclature shown in Table 3-9. These designations shown are used in the subsequent descriptions.

Table 3-9. Processor Operand Representation

BITS[31:24]	BITS[23:16]	BITS[15:8]	BITS[7:0]	FORMAT
OP0	OP1	OP2	OP3	Longword Operand
		OP2	OP3	Word Operand
			OP3	Byte Operand

A bus cycle is a request to transfer data from the bus master to some slave device. Since the ColdFire Architectures supports byte, word, and longword operand types, on misaligned boundaries, there are certain requirements on the bus architecture to support these data types. The main support is to guarantee that each byte of data is aligned to the proper “lane” to assure it is handled properly by both master and slave. Note also, that for line transfers, the data alignment is treated as 4 longword transfers. Specific protocols to handle these transfers are discussed in the next section.

All transfers on M-Bus assume that the devices on M-Bus are 32 bits wide. If dynamic sizing is supported in a system, to word or byte ports, it is handled by the System Bus Controller. To support this bus sizing feature, there are certain *data replication functions which must be performed by all M-Bus masters during write cycles*. For all data transfers, **MADDR[31:2]** indicates the longword base address of the first byte of the reference item. **MADDR[1:0]**

indicates the byte offset from this base address. The **MSIZ[1:0]** field along with the low-order 2 address bits are used to determine how the data buses are used.

The following tables, Table 3-10 and Table 3-11, indicate **MRDATA** requirements for read transfers and **MWDATA** requirements for write transfers. A “-” indicates a “don’t care”, i.e., the value is ignored. These tables define the complete set of allowable combinations of **MSIZ[1:0]** and **MADDR[1:0]**.

Table 3-10. MRDATA Requirements for Read Transfers

Transfer Size	MSIZ [1:0]	MADDR [1:0]	MRDATA [31:24]	MRDATA [23:16]	MRDATA [15:8]	MRDATA [7:0]
Byte	01	00	OP3	-	-	-
	01	01	-	OP3	-	-
	01	10	-	-	OP3	-
	01	11	-	-	-	OP3
Word	10	00	OP2	OP3	-	-
	10	10	-	-	OP2	OP3
Long	00	00	OP0	OP1	OP2	OP3
Line	11	00	OP0	OP1	OP2	OP3

Table 3-11. MWDATA Bus Requirements for Write Transfers

Transfer Size	MSIZ [1:0]	MADDR [1:0]	MWDATA [31:24]	MWDATA [23:16]	MWDATA [15:8]	MWDATA [7:0]
Byte	01	00	OP3	-	-	-
	01	01	OP3	OP3	-	-
	01	10	OP3	-	OP3	-
	01	11	OP3	OP3	-	OP3
Word	10	00	OP2	OP3	-	-
	10	10	OP2	OP3	OP2	OP3
Long	00	00	OP0	OP1	OP2	OP3
Line	11	00	OP0	OP1	OP2	OP3

3.3.3.5 LINE TRANSFERS. A line is defined as a 16-byte value, 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. Transfers on each of these data phases are longword in size. Although the line itself is aligned on 16-byte boundaries, the line access does not necessarily begin on a 0-modulo-16 address. They can begin at

any aligned long word address with **MADDR[1:0]** = 00. Therefore, the slave system (combination of the SBC, modules, and external devices) must be able to cycle through the longword addresses. The allowable patterns during a line accesses are shown in Table 3-12 below:

Table 3-12. Allowable Line Access Patterns

MADDR[3:2]	Longword Accesses
00	$0 - $4 - $8 - $C
01	$4 - $8 - $C - $0
10	$8 - $C - $0 - $4
11	$C - $0 - $4 - $8

Figure 3-7 shows a line access read with zero wait states. Note that another address phase may be initiated at any time during the data phase. This address phase corresponds to the next bus cycle. Also note that the address hold may be asserted during this time. Address hold has no effect on the data phase of a line access. The line access completes and the address is held before the next data phase is allowed.

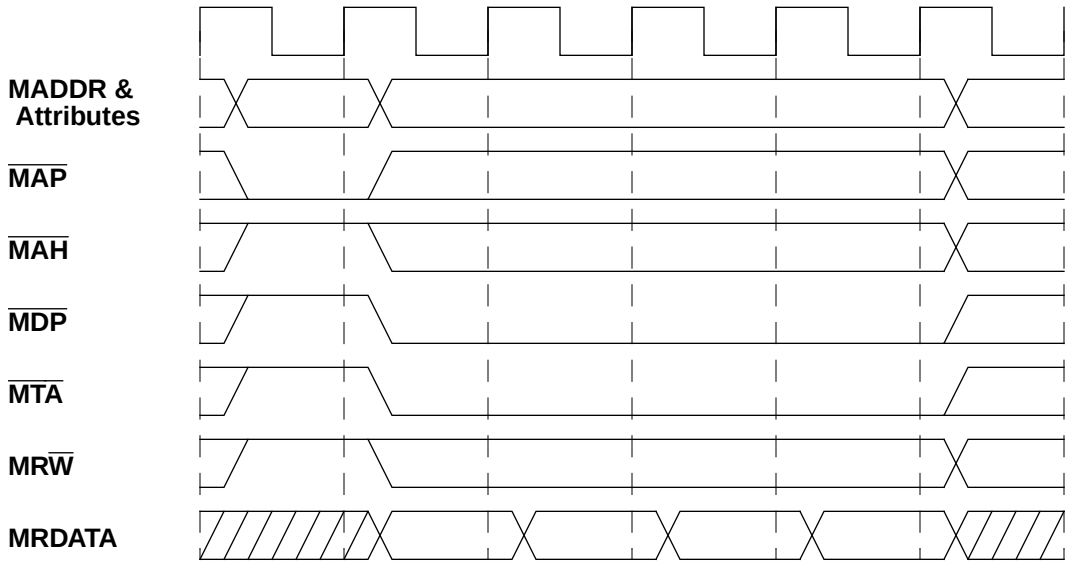


Figure 3- 9. Line Access Read with Zero Wait States

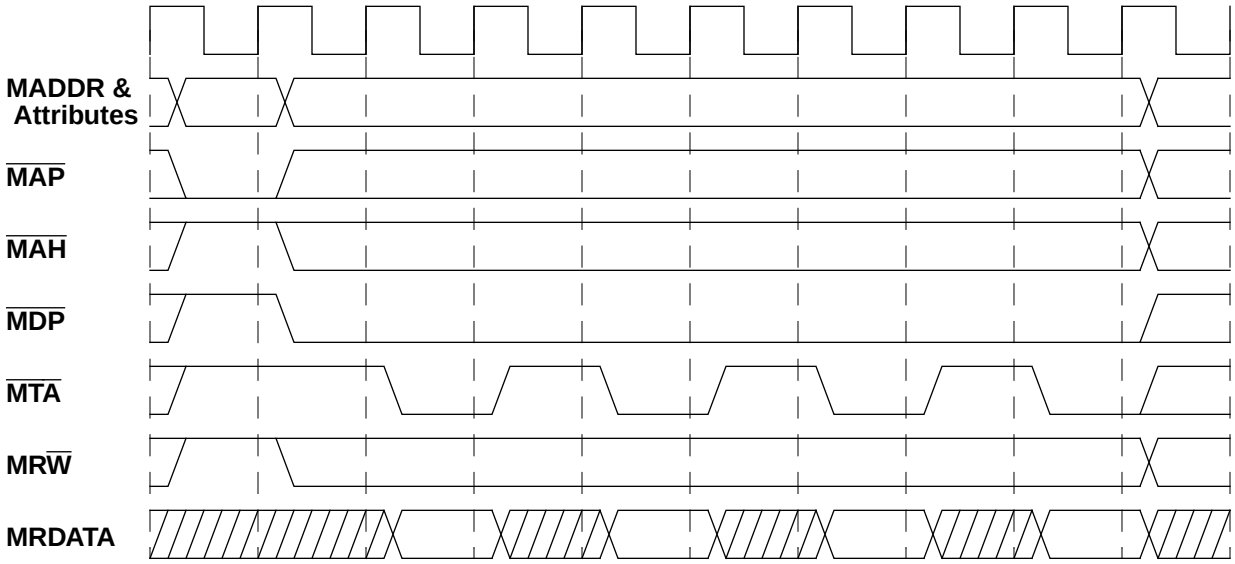


Figure 3- 10. Line Access Read with 1 Wait State

Figure 3-9 and Figure 3-10 show line write accesses. Note that the next long word of data is available on the clock immediately following the termination. There may be cases where data may be pipelined to the external bus by terminating the access and registering the data in the System Bus Controller during the first clock of the data phase. This allows the next longword of data to be available at the next rising clock edge.

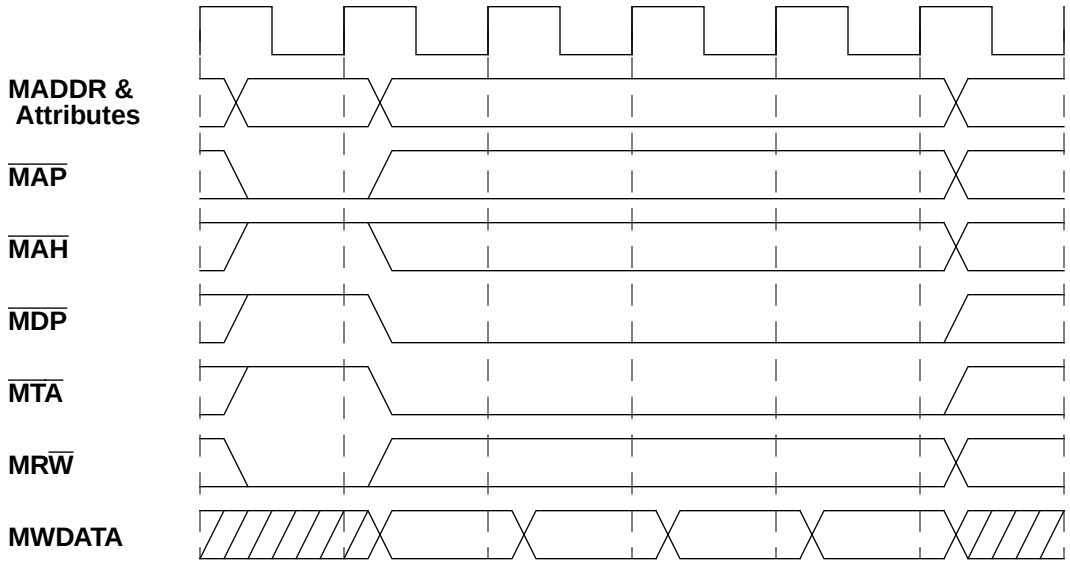


Figure 3- 11. Line Access Write with Zero Wait States

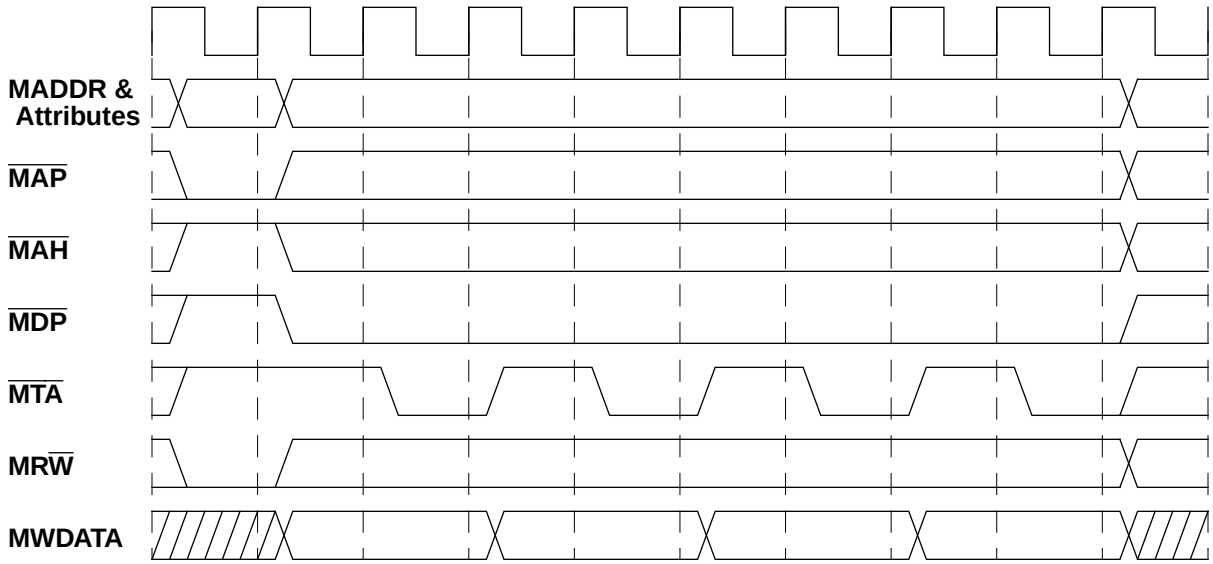


Figure 3- 12. Line Access Write with One Wait State

3.3.3.6 BUS ARBITRATION. Multiple bus masters are handled on the M-Bus through a multiplexed bus scheme. There cannot be multiple masters on the same physical bus. Figure 3-11 shows the top level architecture of a two-master, multiplexed M-Bus system. Mux control is provided by the arbitration block. The address, attributes, write data, **MAP** and **MDP** are multiplexed to the System Bus Controller. 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 do not need to be multiplexed. Address hold is generated by the arbitration logic to stall the master that does not currently have the bus.

The multiplexed scheme was adopted to more easily accommodate a standard cell methodology. There are no three-state or bidirectional signals on the bus. One implication of this architecture is that the addition of more bus masters causes the multiplexing to become more complex and possibly creates timing problems. Designs should seek to limit the number of M-Bus masters. For instance, instead of putting three DMA modules on the M-Bus, a single 3-channel DMA should be investigated.

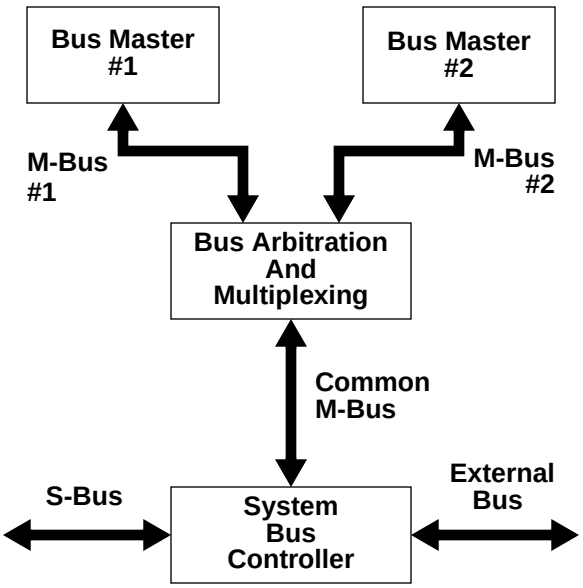


Figure 3- 13. Multiplexed M-Bus Structure

Figure 3-12 shows waveforms with two bus masters multiplexed onto a common M-Bus. The exact arbitration scheme and relative priority of bus masters is not defined in this document. That is determined by the implementation of the arbitration logic. In this example, bus master #1 represents the default bus master, such as a core processor. The **MAH** signal for this master is normally high allowing the master to utilize the bus as needed. When bus master #2, which serves as the alternate master, such as a DMA controller, needs the bus, it asserts its **MAP** signal which serves as the bus request. The arbiter begins the bus transition by asserting **MAH1** to hold off the first bus master. It also transitions **SEL_A_1** which is the mux control signal for address, attributes, and **MAP**. Since there is an active data phase on the bus, the data portion of the bus is not allowed to be muxed until termination of that bus cycle. At that point, **SEL_D_1** the mux control for **MWDATA**, **MDP**, and **MTA** is toggled. The second bus master runs its bus cycle, on the common bus, and the bus is then returned to the first bus master. Note that there is no need to multiplex **MRDATA**. Since data is sampled by the bus master when the data phase is terminated, control of the termination signal is sufficient.

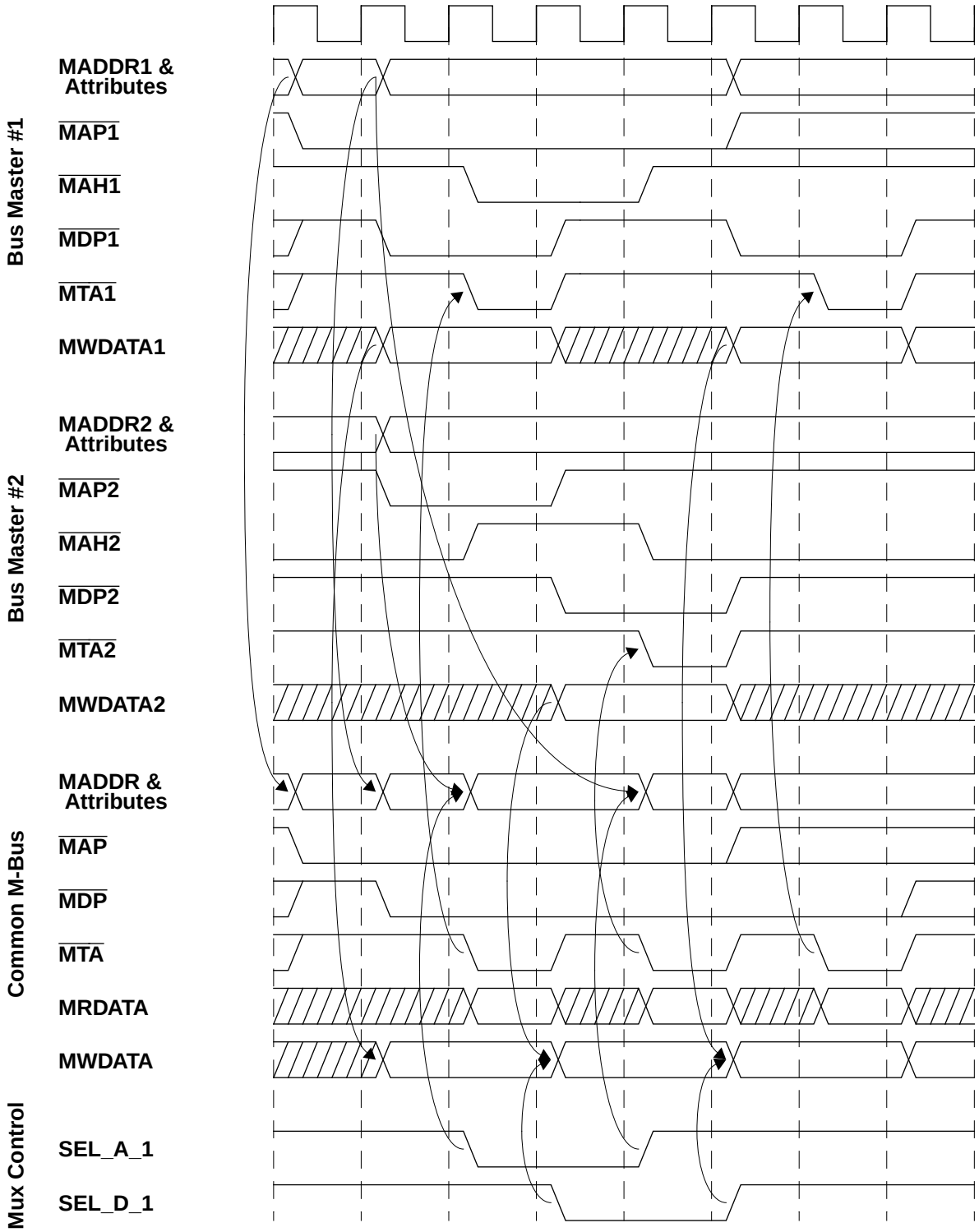


Figure 3- 14. Multiplexed M-Bus Operation

3.3.3.7 INTERRUPT SUPPORT. Interrupts are supported on the M-Bus by the **MIPL** signals and Interrupt Acknowledge Cycles. When an interrupt is pending, the SBC is

responsible for driving the **MIPL** signals 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** signals indicate an acknowledge cycle (**MTT[1:0]** = 11) and the interrupt level of the interrupt being processed is specified in the **MTM** 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 **MDATA[31:24]**.

3.3.3.8 RESET OPERATION. When a master is reset, that is when **MRSTI** is driven low, the bus control signals of that M-Bus master are driven to their inactive state. This means that **MAP**, **MDP**, **MRW** and **MTA** are all driven high. **MAH** is an exception to this case and may be driven high or low depending on the specific system implementation.

Date: 08-21-98
Revision No: 0.1
Pages affected: 25, 26 (see change bars)

SECTION 4
V3 CPU

4.1 INTRODUCTION

The design focus of the Version 3 (V3) ColdFire processor was the development of a higher performance core while maintaining backward code compatibility with the previous generation, the Version 2 core. The V3 core represents another step on the ColdFire roadmap, and with its enhanced pipeline structure and local memories, provides a high level of performance needed by today's demanding embedded applications.

4.2 VERSION 3 PROCESSOR MICROARCHITECTURE

4.2.1 Version 3 Processor Pipeline Overview

All ColdFire processor cores consist of two independent, decoupled pipeline structures to maximize performance. The Instruction Fetch Pipeline (IFP) prefetches instructions, while the Operand Execution Pipeline (OEP) decodes the instruction, fetches the required operands and then executes the specified function. While one of the goals of the original ColdFire microarchitecture was to minimize overall size, the driving factor in the Version 3 design was to better balance the logic delays associated with each pipeline stage to allow the operating frequency to be raised significantly. For some functions, this required new pipeline stages to be added to support the higher frequency goals. In particular, accesses on the processor's local, high-speed bus were reimplemented to use a 2-stage pipelined bus to the cache, RAM and ROM memories. Additionally, the time-critical instruction decode functions within the Operand Execution Pipeline were relocated into a new stage in the IFP, named the Instruction Early Decode stage. The implementation of the early decode pipeline stage was first used in the development of the superscalar MC68060 microprocessor, and is a proven technology addressing the decode issues normally associated with variable-length instructions.

The net effect is the Version 3 pipeline structure is considerably different than the Version 2 design. The V3 Instruction Fetch Pipeline is a 4-stage design with an optional instruction buffer stage, while the Operand Execution Pipeline retains its 2-stage structure. In the OEP design, each pipeline stage has multiple functions.

The V3 processor pipeline stages are:

Instruction Fetch Pipeline

- Instruction Address Generation (IAG) — Calculation of the next prefetch address

- Instruction Fetch Cycle 1 (IC1) — Initiation of prefetch access on the processor's local bus
- Instruction Fetch Cycle 2 (IC2) — Completion of prefetch access on the processor's local bus
- Instruction Early Decode (IED) — Generation of time-critical decode signals needed for the OEP
- Instruction Buffer (IB) — Optional buffer stage using FIFO queue

Operand Execution Pipeline

- Decode, Select/Operand Fetch Cycle (DSOC) — Decode the instruction and select the required components for the effective address calculation, or the operand fetch cycle
- Address Generation/Execute Cycle (AGEX) — Calculate the operand address, or perform the execution of the instruction

4.2.2 Version 3 Instruction Fetch Pipeline

The resulting four-stage IFP implementation calculates the next prefetch address, fetches the instruction data with two stages mapped onto the 2-stage pipeline local-memory bus structure, followed by the early decode stage. When the instruction buffer is empty, prefetched instruction data is loaded directly from the IED stage into the Operand Execution Pipeline. If the buffer is not empty, the IFP stores the contents of the prefetch in the FIFO queue until it is required by the OEP.

It should be noted that the organization of the Version 3 instruction buffer is fundamentally different than the V2 approach. One of the time-critical decode fields provided by the early decode stage of the IFP is the instruction length. By knowing the length of the prefetched instructions, the IED field is able to package the fetched data into machine instructions and load them into the FIFO instruction buffer in that form.The Version 3 design implements an 8-entry instruction buffer, where each entry contains one machine instruction in the form of the operation word, the early decode information (also known as the extended operation word), and the optional extension words 1 and 2. This approach greatly simplifies and accelerates the OEP's read logic. As one instruction is completed in the OEP, the next instruction, regardless of instruction length, is read from the next sequential buffer location and loaded into the instruction registers.

4.2.2.1 CHANGE OF FLOW ACCELERATION. Since the Version 3 Instruction Fetch and Operand Execution Pipelines are decoupled by the instruction buffer, the increased depth of the IFP is generally hidden from the OEP's instruction execution. The one exception is change-of-flow instructions, e.g., unconditional branches or jumps, subroutine calls, taken conditional branches, etc. For these instructions, the increased depth of the IFP pipeline is fully exposed. To minimize the effects of this increased depth, a logic module dedicated to

change-of-flow acceleration was developed for the IED stage of the Instruction Fetch Pipeline.

Given that the instruction boundaries are known in the IED stage, a logical extension was the creation of branch acceleration module which could “monitor” the prefetched stream, looking for change-of-flow opcodes.The basic premise of the Version 3 branch acceleration is to detect certain types of change-of-flow instructions, calculate their target instruction address, and immediately begin fetching down the target stream. By allowing the switching of the prefetch stream to be handled completely within the IFP without any Operand Execution Pipeline intervention, the typical execution time is greatly improved.

As an example, consider a PC-relative unconditional branch using the BRA instruction. The branch acceleration logic searches the prefetch stream for this type of opcode. Once encountered, the acceleration logic calculates the target address by summing the current instruction prefetch address with a displacement contained in the instruction. This detection and calculation of the target address occurs in the IED stage of the BRA prefetch. The target address is then immediately fed back into the IAG stage, causing the current prefetch stream to be discarded and a new stream at the target address established. Given that the two pipelines are decoupled, in many cases, the target instruction is available to the OEP immediately after the BRA instruction, making its execution time appear as a single cycle.

The acceleration logic uses a static prediction algorithm when processing conditional branch (Bcc) instructions. The default prediction scheme is forward Bcc instructions are predicted as not-taken, while backward Bcc opcodes are predicted as taken. A user-mode control bit (bit 7 of the CCR) is provided to allow users to dynamically alter the prediction algorithm for forward Bcc instructions. See **Section 4.4.5: Condition Code Register** for details.

Depending on the runtime characteristics of an application, processor performance may be increased significantly by the assertion or negation of this configuration bit. See Appendix B on Branch Instruction Execution Times for details on individual instruction performance.

4.2.3 Version 3 Operand Execution Pipeline

The OEP is implemented in a two-stage pipeline featuring a traditional RISC datapath with a dual-read-ported register file feeding an arithmetic/logic unit. For simple register-to-register instructions, the first stage of the OEP performs the instruction decode and fetching of the required register operands (OC), while the actual instruction execution is performed in the second stage (EX). For memory-to-register (embedded-load) instructions, the instruction is effectively staged through the OEP twice. First, the instruction is decoded and the components of the operand address are selected (DS). Second, the operand address is generated using the “execute engine” (AG). Third, the memory operand is fetched from the processor local bus, while any register operand is simultaneously fetched (OC). Finally, in the last cycle, the instruction is executed (EX). For register-to-memory operations, the stage functions (DS/OC, AG/EX) are effectively performed simultaneously allowing single-cycle execution. For read-modify-write instructions, the pipeline effectively combines an embedded-load with a store operation.

4.2.3.1 ILLEGAL OPCODE HANDLING. As an aid in conversion from M68000 Family code, the complete space defined by the 16-bit opcode is decoded. If the processor attempts execution of an illegal or non-supported instruction, an illegal instruction exception is taken.

4.2.3.2 HARDWARE MULTIPLY-ACCUMULATE (MAC) AND DIVIDE EXECUTION UNITS. The optional MAC unit is designed to provide hardware support for a limited set of signal processing operations that are currently being used in embedded code today, while supporting the integer multiply instructions in the ColdFire microprocessor family.

The MAC unit provides functionality in three related areas:

- Signed and unsigned integer multiplies
- Multiply-accumulate operations supporting signed and unsigned operands
- Miscellaneous register operations

The ColdFire MAC has been optimized for 16x16 multiplies to minimize silicon costs. The

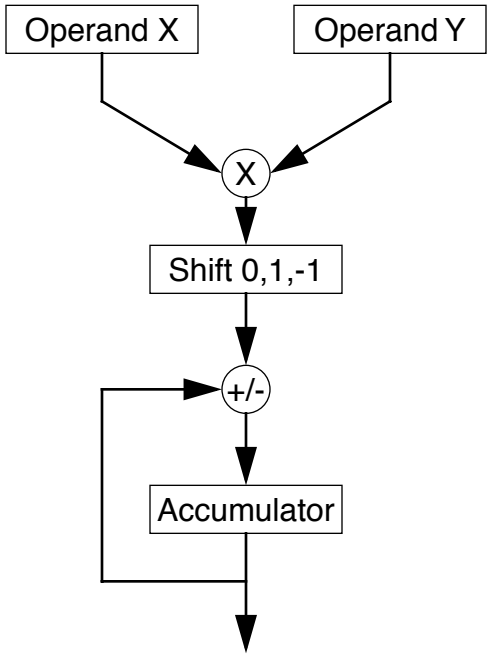


Figure 4-1. ColdFire Multiply-Accumulate Functionality Diagram

MAC unit is tightly coupled to the processor's Operand Execution Pipeline and features a 3-stage execution pipeline. The OEP can issue a 16 x 16 multiply with a 32-bit accumulate operation in a single cycle, while a 32 x 32 multiply with a 32-bit accumulation requires three cycles before the next instruction can be issued. Figure 4-1 shows the basic functionality of the ColdFire MAC.

The Operand Execution Pipeline also includes a hardware execute engine which performs all integer divide operations. The supported divide functions include: 32/16 producing a 16-

bit quotient and a 16-bit remainder, 32/32 producing a 32-bit quotient, and 32/32 producing a 32-bit remainder.

If execution of a MAC or divide opcode is attempted and the corresponding hardware unit is not present, then a non-supported instruction exception is generated.

For detailed instruction descriptions on the MAC and divide opcodes, see the ColdFire Microprocessor Family Programmer's Reference Manual (MCF5200PRM/AD).

4.2.4 Version 3 Processor Pipeline Block Diagrams and Summary

The following diagrams present a more detailed view of the internal pipeline structures for the Version 3 processor. Compared to the two-stage Version 2 design, note the increased length of the IFP with the early decode (ED) table lookup and the branch acceleration target address adders in the IED stage with the feedback to the prefetch address logic in the IAG stage. The OEP is essentially unchanged from the Version 2 design with the exception of the extended opword provided from the IFP as part of the instruction interface:

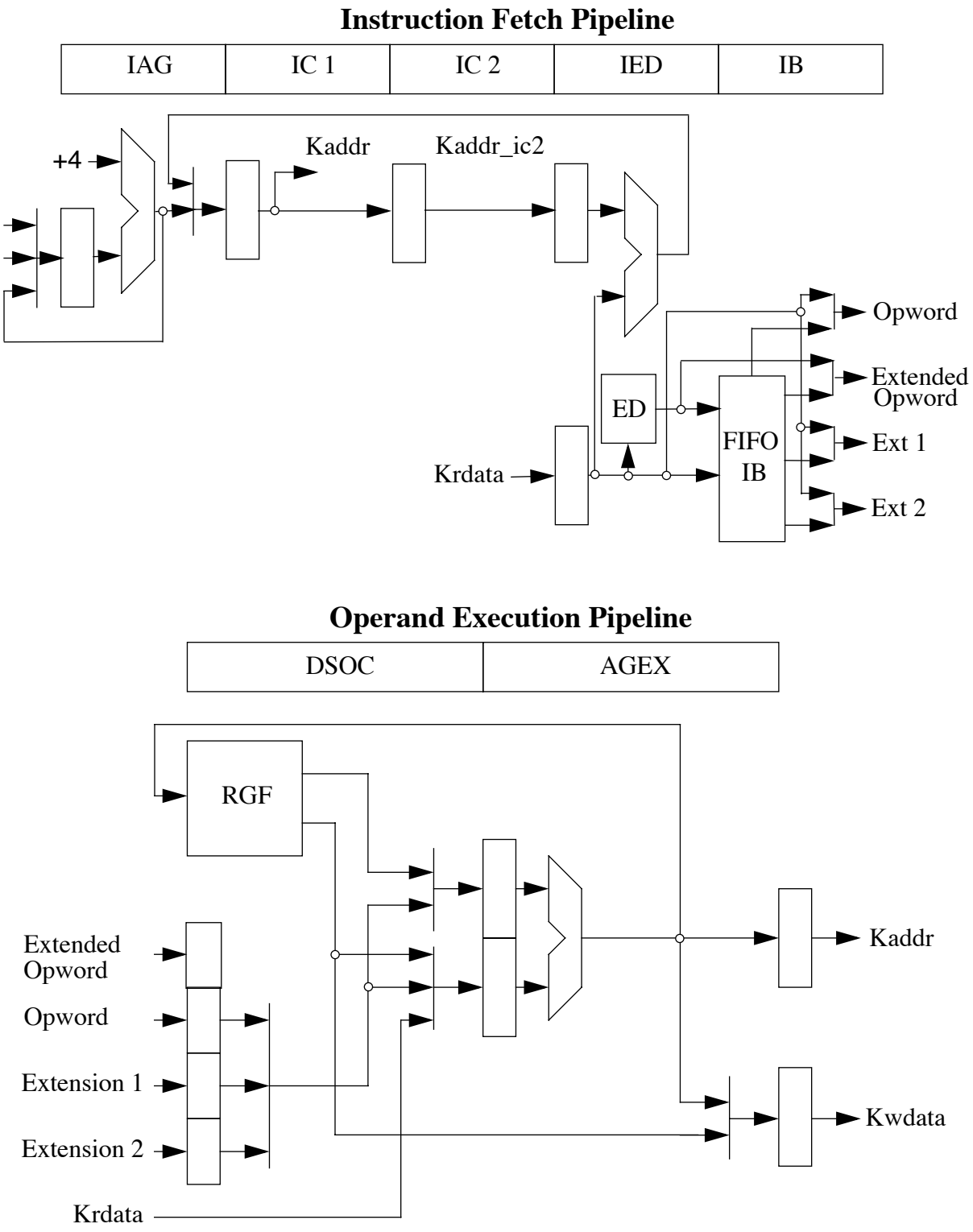


Figure 4-2. Version 3 ColdFire Pipeline Diagram

As a result of the increased IFP pipelining and an exposed cycle of latency on most operand read references, the cycles per instruction performance of the V3 core is usually slightly lower than V2 at a given frequency. However, the entire design focus of the V3 development was to maximize the operating frequency, and comparison of speeds in a given process technology indicate this goal was achieved. Using a common 0.35 micron process technology, the Version 3 core synthesizes into a ~200,000 transistor implementation with a size of 3 mm^2 (no MAC or DIV units) and 3.8 mm^2 with the MAC and DIV. Operating frequency is increased to 1.5x relative to V2 and reaches 90-100 MHz. Finally, the Version 3 microarchitecture provides a 0.78 Dhrystone 2.1 MIPS per MHz performance with an 8 KByte unified cache.

A comprehensive analysis using a standard set of embedded benchmarks has measured the following relative performance based on initial implementation for each core generation:

90/45 MHz V3 = 2.5 x 33.3 MHz V2

with the following configurations:

- A 90 MHz V3 processor complex with an 8 KByte unified cache with a 1/2x speed external bus with a 4-2-2-2 memory response speed
- A 33.3 MHz V2 processor complex with a 2 KByte unified cache with a 3-1-1-1 memory response speed
- Copyback cache mode for both processors, no KRAM memory, and no MAC or DIV instructions

4.3 COLDFIRE PROCESSOR PROGRAMMING MODEL

Refer to the ColdFire Microprocessor Family Programmer's Reference Manual (MCF5200PRM/AD) for detailed information on the operation of the instruction set and addressing modes.

The core programming model consists of three instruction and register groups: user, user-mode MAC, and supervisor. Programs executing in user mode are restricted to the basic user and MAC programming models. System software executing in supervisor mode can reference all user-mode and MAC instructions and registers, plus an additional set of privileged instructions and control registers. The appropriate programming model is selected based on the privilege level (user or supervisor) of the processor as defined by the S-bit of the status register. The following paragraphs describe the registers in the user, MAC and supervisor programming models.

4.4 User Programming Model

Figure 4-3 illustrates the user programming model. It consists of the following registers:

- 16 general-purpose 32-bit registers
- 32-bit program counter
- 8-bit condition code register

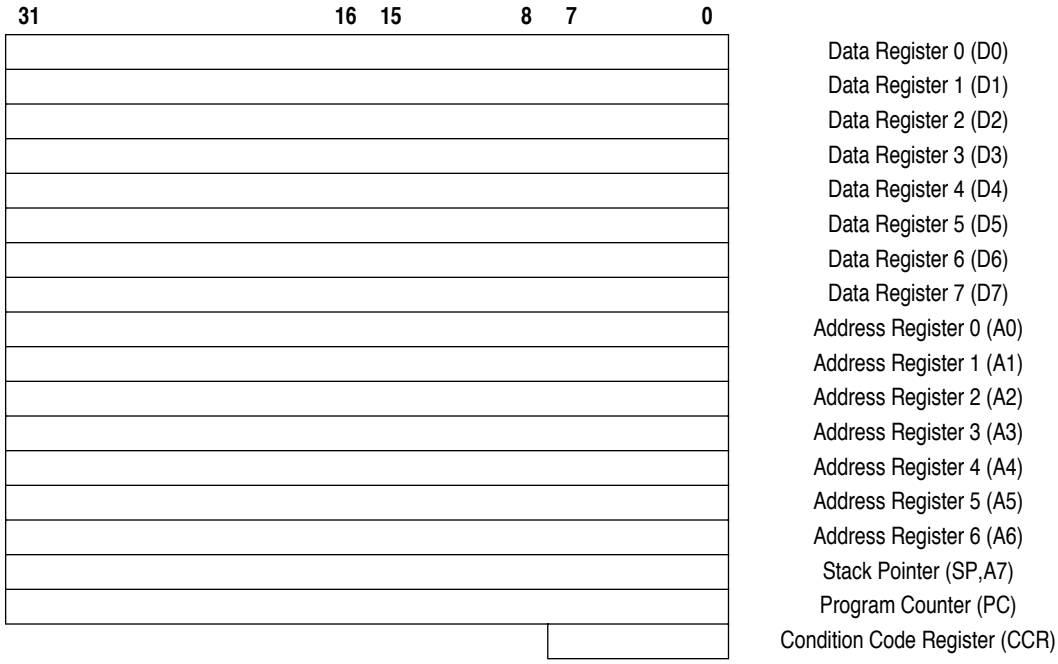


Figure 4-3. User Programming Model

4.4.1 DATA REGISTERS (D0 – D7) . Registers D0–D7 are used as data registers for bit (1 bit), byte (8 bits), word (16 bits), and longword (32 bits) operations and may also be used as index registers.

4.4.2 ADDRESS REGISTERS (A0 – A6) . These registers can be used as software stack pointers, index registers, or base address registers and may be used for word and longword operations.

4.4.3 STACK POINTER (A7, SP) . The processor core supports a single hardware stack pointer (A7) used during stacking for subroutine calls, returns, and exception handling. The initial value of A7 is loaded from the reset exception vector, address $0. The same register is used for user and supervisor modes, and may be used for word and longword operations.

A subroutine call saves the PC on the stack, and the return restores the PC from the stack. Both the PC and the status register (SR) are saved on the stack during the processing of exceptions and interrupts. The return from exception instruction restores the SR and PC values from the stack.

4.4.4 PROGRAM COUNTER (PC). The PC contains the address of the currently executing instruction. During instruction execution and exception processing, the processor automatically increments the contents of the PC or places a new value in the PC, as appropriate. For some addressing modes, the PC can be used as a pointer for PC-relative operand addressing.

4.4.5 CONDITION CODE REGISTER (CCR). The CCR is the least significant byte of the processor status register (SR), as shown later.

Bit 7, the branch prediction bit, provides a mechanism to alter the static prediction algorithm used by the branch acceleration logic in the Instruction Fetch Pipeline. The prediction algorithm is defined as:

```
if Bcc instruction is a forward branch && (CCR.P == 0)
   then the Bcc is predicted as not-taken
if Bcc instruction is a forward branch && (CCR.P == 1)
   then the Bcc is predicted as taken
```

All backwards Bcc instructions are predicted as taken. The forward/backward classification is defined by the sign of the address displacement: if the address displacement is positive, the Bcc is forward, while a negative displacement produces a backward branch.

Depending on the dynamic characteristics of a given application, the processor performance may be increased by the assertion or negation of this control bit.

Bits 4–0 represent indicator flags based on results generated by processor operations. Bit 4, the extend bit (X bit), is also used as an input operand during multiprecision arithmetic computations.

BITS	7	6	5	4	3	2	1	0
FIELD	P	-	-	X	N	Z	V	C
RESET	0	0	0	-	-	-	-	-
R/W	R/W	R	R	R/W	R/W	R/W	R/W	R/W

Condition Code Register (CCR)

Field Definitions:

P[7]—Branch Prediction Bit

Setting this bit causes forward conditional branches to be predicted as taken. Clearing this bit causes forward conditional branch instructions to be predicted as not-taken.

X[4]—Extend Condition Code

Assigned the value of the carry bit for arithmetic operations; otherwise not affected.

N[3]—Negative Condition Code

Set if the most significant bit of the result is set; otherwise cleared.

Z[2]—Zero Condition Code

Set if the result equals zero; otherwise cleared.

V[1]—Overflow Condition Code

Set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise cleared.

C[0]—Carry Condition Code

Set if a carry out of the most significant bit of the operand occurs for an addition, or if a borrow occurs in a subtraction; otherwise cleared.

See the ColdFire Microprocessor Family Programmer's Reference Manual (MCF5200PRM/AD) for more information on how specific instructions affect the condition code register bits.

4.5 MAC Programming Model

Figure 4-4 illustrates the MAC portion of the user programming model available on the processor core. It consists of the following registers:

- 32-bit accumulator (ACC)
- 16-bit mask register (MASK)
- 8-bit MAC status register (MACSR)

The instructions which reference the MAC registers always transfer 32 bits of data, regardless of the implemented size of the register.

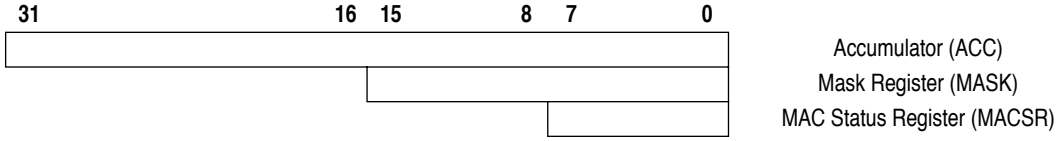


Figure 4-4. MAC Unit User Programming Model

4.5.1 ACCUMULATOR (ACC). This is a 32-bit general-purpose register used to accumulate the results of MAC operations.

4.5.2 MASK REGISTER (MASK). This is a 16-bit general-purpose register for use as an optional address mask during MAC instructions which fetch operands from memory. It is useful in the implementation of circular queues in operand memory.

4.5.3 MAC STATUS REGISTER (MACSR). This is an 8-bit special-purpose register which defines the operating configuration of the MAC unit, and contains indicator flags from the results of MAC instructions.

4.6 Supervisor Programming Model

System programmers use the supervisor programming model to implement sensitive operating systems, I/O control, memory configuration and management. The following paragraphs briefly describe the registers in the supervisor programming model. All accesses that affect the control features of the processor are in the supervisor programming model, which consists of the instructions and registers accessible in the user and MAC models, as well as the registers listed in Figure 4-5.

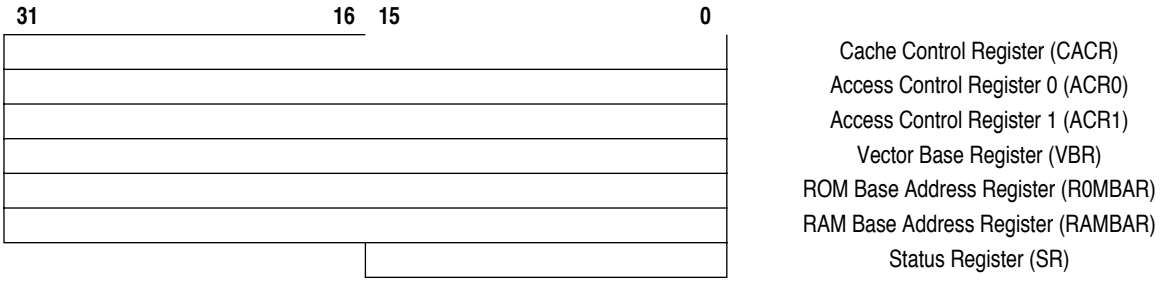


Figure 4-5. Supervisor Programming Model

Most of the control registers are accessed via the MOVEC instruction using the control register definitions shown in Table 4-1.

Table 4-1. MOVEC Register Map

RC[11:0]	REGISTER DEFINITION
$002	Cache Control Register (CACR)
$004	Access Control Register 0 (ACR0)
$005	Access Control Register 1 (ACR1)
$801	Vector Base Register(VBR)
$C00	ROM Base Address Register (R0MBAR)
$C04	RAM Base Address Register (RAMBAR)

4.6.1 CACHE CONTROL REGISTER (CACR). The CACR controls the operation of the unified cache memory. This register includes enable, freeze and invalidate controls, plus line fill buffer configuration control as well as the default cache mode and write protect fields. See Section 4.3 for a complete description of the CACR.

4.6.2 ACCESS CONTROL REGISTERS (ACR0, ACR1). The ACR registers allow specification of certain attributes for two user-defined regions of memory. These attributes include definition of cache mode, write protect and buffer write enables. See **Section 5.3.3** for a complete description of the ACR registers.

4.6.3 VECTOR BASE REGISTER (VBR). The VBR contains 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. The lower 20 bits of the VBR are not implemented by ColdFire processors; they are assumed to be zero, forcing the table to be aligned on a 0-modulo-1 MByte boundary.

4.6.4 RAM BASE ADDRESS REGISTER (RAMBAR). This register determines the base address location of the processor local RAM module, plus provides definition of the types of references that are mapped into it. The register includes a base address, write protect bit, address space mask bits and enable. See **Section 5.4.2** for a complete description of the RAMBAR register.

4.6.5 ROM BASE ADDRESS REGISTER (R0MBAR). This register determines the base address location of the processor local ROM module, plus provides definition of the types of references that are mapped into it. The register includes a base address, write protect bit,

address space mask bits and enable. See **Section 5.5.2** for a complete description of the R0MBAR register.

4.6.6 STATUS REGISTER (SR). The following illustrates the SR, which stores the processor status, the interrupt priority mask, and other control bits. In supervisor mode, software can access the entire SR, but in user mode, only the lower 8 bits are accessible as the CCR. The control bits indicate the following states for the processor: trace mode (T-bit), supervisor mode (S-bit) and master mode (M-bit).

BITS	15	14	13	12	11	10		8	7	6	5	4	3	2	1	0
FIELD	T	-	S	M	-	I			P	-		X	N	Z	V	C
RESET	0	0	1	0	0	7			0	00		-	-	-	-	-
R/W	R/W	R	R/W	R/W	R	R/W			R/W	R		R/W	R/W	R/W	R/W	R/W

Status Register (SR)

Field Definitions:

T[15]—Trace Enable

When set, the processor performs a trace exception after every instruction; otherwise no trace exception is performed.

S[13]—Supervisor / User State

Denotes the processor privilege mode: supervisor mode (S =1) or user mode (S = 0).

M[12]—Master / Interrupt State

This bit is cleared by an interrupt exception, and can be set by software during execution of the RTE or move to SR instructions.

I[10:8]—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 level seven request, which cannot be masked.

4.7 EXCEPTION PROCESSING OVERVIEW

Exception processing for ColdFire processors is streamlined for performance. Differences from previous M68000 Family processors include:

- A simplified exception vector table
- Reduced relocation capabilities using the vector base register
- A single exception stack frame format
- Use of a single, self-aligning system stack pointer

ColdFire processors use an instruction restart exception model but do require software support to recover from certain access errors. See **Section 4.7.1 Access Error Exception** for details.

Exception processing is comprised of four major steps and can be defined as the time from the detection of the fault condition until the fetch of the first handler instruction has been initiated.

First, the processor makes an internal copy of the SR and then enters supervisor mode by setting the S-bit and disabling trace mode by clearing the T-bit. The occurrence of an interrupt exception also forces the M-bit to be cleared and the debug

priority mask to be set to the level of the current interrupt request.

Second, the processor determines the exception vector number. For all faults except interrupts, the processor performs this calculation based on the exception type. For interrupts, the processor performs an interrupt-acknowledge (IACK) bus cycle to obtain the vector number from a peripheral device. The IACK cycle is mapped to a special acknowledge address space with the interrupt level encoded in the address.

Third, the processor saves the current context by creating an exception stack frame on the system stack. ColdFire processors support a single stack pointer in the A7 address register; therefore, there is no notion of separate supervisor or user stack pointers. As a result, the exception stack frame is created at a 0-modulo-4 address on the top of the current system stack. Additionally, the processor uses a simplified fixed-length stack frame for all exceptions. The exception type determines whether the program counter placed in the exception stack frame defines the location of the faulting instruction (fault) or the address of the next instruction to be executed (next).

Fourth, the processor acquires the address of the first instruction of the exception handler. By definition, the exception vector table is aligned on a 1 MByte boundary. This instruction address is obtained by fetching a value from the table located at the address defined in the vector base register. The index into the exception table is calculated as (4 x vector_number). Once the index value has been generated, the contents of the vector table determine the address of the first instruction of the desired handler. After the instruction fetch for the first opcode of the handler has been initiated, exception processing terminates and normal instruction processing continues in the handler.

ColdFire processors support a 1024-byte vector table aligned on any 1 MByte address boundary (see Table 4-2). The table contains 256 exception vectors where the first 64 are defined by Motorola and the remaining 192 are user-defined interrupt vectors.

Table 4-2. Exception Vector Assignments

VECTOR NUMBER(S)	VECTOR OFFSET (HEX)	STACKED PROGRAM COUNTER	ASSIGNMENT
0	$000	-	Initial stack pointer
1	$004	-	Initial program counter
2	$008	Fault	Access error
3	$00C	Fault	Address error
4	$010	Fault	Illegal instruction
5-7	$014-$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	Debug interrupt
13	$034	-	Reserved
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
48-60	$0C0-$0F0	-	Reserved
61	$0F4	Fault	Non-supported Instruction
62-63	$0F8-$0FC	-	Reserved
64-255	$100-$3FC	Next	User-defined interrupts

"Fault" refers to the PC of the instruction that caused the exception
"Next" refers to the PC of the next instruction that follows the instruction that caused the fault.

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 contained in the status register.

4.6 EXCEPTION STACK FRAME DEFINITION

The exception stack frame is shown in Figure 4-6. The first longword of the exception stack frame contains the 16-bit format/vector word (F/V) and the 16-bit status register. The second longword contains the 32-bit program counter address.

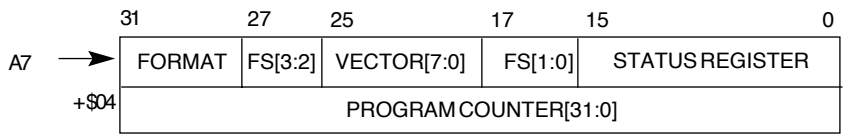


Figure 4-6. Exception Stack Frame Form

The 16-bit format/vector word contains 3 unique fields:

- A 4-bit format field at the top of the system stack is always written with a value of {4,5,6,7} by the processor indicating a two-longword frame format. See Table 4-3. This field records any longword misalignment of the stack pointer which might have existed at the time the exception occurred.

Table 4-3. Format Field Encoding

ORIGINAL A7 @ TIME OF EXCEPTION, BITS 1:0	A7 @ 1ST INSTRUCTION OF HANDLER	FORMAT FIELD BITS 31:28
00	Original A7 - 8	0100
01	Original A7 - 9	0101
10	Original A7 - 10	0110
11	Original A7 - 11	0111

- A 4-bit fault status field, FS[3:0], at the top of the system stack. This field is defined for access and address errors only and written as zeros for all other types of exceptions. See Table 4-4.

Table 4-4. Fault Status Encodings

FS[3:0]	DEFINITION
0000	Not an access or address error
0001	Reserved
001x	Reserved
0100	Error on instruction fetch
0101	Reserved
011x	Reserved
1000	Error on operand write
1001	Attempted write to write-protected space
101x	Reserved
1100	Error on operand read
1101	Reserved
111x	Reserved

- The 8-bit vector number, vector[7:0], defines the exception type and is calculated by the processor for all internal faults and represents the value supplied by the peripheral in the case of an interrupt. Refer to Table 4-2.

4.7 PROCESSOR EXCEPTIONS

4.7.1 Access Error Exception

For the Version 3 ColdFire core, access errors are only reported in conjunction with an attempted store to a write-protected memory space. Thus, access errors associated with instruction fetch or operand read accesses are not possible.

The ColdFire processor uses an imprecise reporting mechanism for access errors on operand writes. Since the actual write cycle may be decoupled from the processor's issuing of the operation, the signaling of an access error appears to be decoupled from the instruction that generated the write. Accordingly, the PC contained in the exception stack frame merely represents the location in the program when the access error was signaled. All programming model updates associated with the write instruction are completed. The NOP instruction can collect access errors for writes. This instruction delays its execution until all previous operations, including all pending write operations, are complete. If any previous write terminates with an access error, it is guaranteed to be reported on the NOP instruction.

4.7.2 Address Error Exception

Any attempted execution transferring control to an odd-byte instruction address (i.e., if bit 0 of the target address is set) results in an address error exception.

Any attempted use of a word-sized index register (Xi.w) or a scale factor of 8 on an indexed effective addressing mode generates an address error as does an attempted execution of an instruction with a full-format indexed addressing mode.

4.7.3 Illegal Instruction Exception

On the Version 2 ColdFire microprocessor implementation, only certain illegal opcodes were decoded and generated an illegal instruction exception. However, the Version 3 processor decodes the full 16-bit opcode and generates an illegal instruction exception if the execution of any non-supported instruction is attempted. Additionally, if execution of any illegal line A or line F opcode is attempted, unique exception types are generated: vector numbers 10 and 11, respectively.

ColdFire processors do not provide illegal instruction detection on the extension words on any instruction, including MOVEC. If execution of any instruction with an illegal extension word is attempted, the resulting operation is undefined.

4.7.4 Privilege Violation

The attempted execution of a supervisor mode instruction while in user mode generates a privilege violation exception. See the ColdFire Microprocessor Family Programmer's Reference Manual for lists of supervisor- and user-mode instructions.

4.7.5 Trace Exception

To aid in program development, the ColdFire processors provide an instruction-by-instruction tracing capability. While in trace mode, indicated by the assertion of the T-bit in the status register (SR[15] = 1), the completion of an instruction execution signals a trace exception. This functionality allows a software debugger to monitor program execution.

The single exception to this definition is the STOP instruction. If the processor is executing in trace mode, the instruction preceding the STOP executes and then generates a trace exception. In the exception stack frame, the PC is pointing to the STOP opcode. Once the trace handler is exited, control returns to the STOP instruction, which is then executed,

loading the SR with the immediate operand from the instruction. The processor then immediately generates a trace exception. The PC in the exception stack frame points to the instruction following the STOP, and the SR reflects the just-loaded value.

If the processor is not operating in trace mode, but executes a STOP instruction where the immediate operand sets the trace bit in the SR, the hardware loads the SR, and then immediately generates a trace exception. The PC in the exception stack frame points to the instruction following the STOP, and the SR reflects the just-loaded value.

Since ColdFire processors do not support any hardware stacking of multiple exceptions, it is the responsibility of the operating system to check for trace mode after processing other exception types. As an example, consider the execution of a TRAP instruction while in trace mode. The processor initiates the TRAP exception and then passes control to the corresponding handler. If the system requires that a trace exception be processed, it is the responsibility of the TRAP exception handler to check for this condition (SR[15] in the exception stack frame asserted) and pass control to the trace handler before returning from the original exception.

4.7.6 Debug Interrupt

This special type of program interrupt is discussed in detail in **Section 6: Debug Support**. This exception is generated in response to a hardware breakpoint register trigger. The processor does not generate an IACK cycle but rather calculates the vector number internally (vector number 12). Additionally, the M-bit and the interrupt priority mask fields of the status register are unaffected by the occurrence of a debug interrupt.

4.7.7 RTE and Format Error Exceptions

When an RTE instruction is executed, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire processor, any 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 the RTE instruction.

The selection of the format value provides some limited debug support for porting code from 68000 applications. On M68000 Family processors, the SR was located at the top of the stack. On those processors, bit[30] of the longword addressed by the system stack pointer is typically zero. Thus, if an RTE is attempted using this “old” format, it generates a format error on a ColdFire processor.

If the format field defines a valid type, the processor: (1) reloads the SR operand, (2) fetches the second longword operand, (3) adjusts the stack pointer by adding the format value to the auto-incremented address after the fetch of the first longword, and then (4) transfers control to the instruction address defined by the second longword operand within the stack frame.

4.7.8 TRAP Instruction Exceptions

The TRAP #n instruction always forces an exception as part of its execution and is useful for implementing system calls. The trap instruction may be used to change from the user to supervisor mode.

4.7.9 Non-Supported Instruction Exceptions

If a ColdFire processor attempts to execute a valid instruction, but the required optional hardware module is not physically present in the Operand Execution Pipeline, a non-supported instruction exception is generated (vector number 61). Control is then passed to an exception handler which can then process the opcode as required by the system.

4.7.10 Interrupt Exception

The 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 may optionally be supported through the System Bus Controller.

4.7.11 Fault-on-Fault Halt

If a ColdFire processor encounters any type of fault during the exception processing of another fault, the processor immediately halts execution with the catastrophic “fault-on-fault” condition. A reset is required to force the processor to exit this halted state.

4.7.12 Reset Exception

Asserting the reset input signal to the processor causes a reset exception. The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. Reset also aborts any processing in progress when the reset input is recognized. Processing cannot be recovered.

The reset exception places the processor in the supervisor mode by setting the S-bit and disables tracing by clearing the T-bit in the SR. This exception clears the M-bit and sets the processor’s interrupt priority mask in the SR to the highest level (level 7). The branch prediction bit in the CCR is also cleared. Next, the VBR is initialized to zero ($00000000). The control registers specifying the operation of any memories (e.g., cache, RAM and ROM modules) connected directly to the processor are disabled. Refer to the specific sections covering those modules for more information.

After the reset signal is negated, the processor waits for sixteen cycles before beginning the actual reset exception process. During this window of time, certain events are sampled, including the assertion of the debug breakpoint signal. If the processor is not halted, it then initiates the reset exception by performing two longword read bus cycles. The first longword at address 0 is loaded into the stack pointer and the second longword at address 4 is loaded into the program counter. 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 is executed, the processor enters the fault-on-fault halted state.

4.8 INTEGER DATA FORMATS

Table 4-5 lists the integer operand data formats. Integer operands can reside in registers, memory, or instructions. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

Table 4-5. Integer Data Formats

OPERAND DATA FORMAT	SIZE
Bit	1 Bit
Byte Integer	8 Bits
Word Integer	16 Bits
Longword Integer	32 Bits

4.9 ORGANIZATION OF DATA IN REGISTERS

The following paragraphs describe data organization within the data, address, and control registers.

4.9.1 Organization of Integer Data Formats in Registers

Figure 4-7 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. The least significant bit (LSB) of all integer sizes is bit zero, the most significant bit (MSB) of a longword integer is bit 31, the MSB of a word integer is bit 15, and the MSB of a byte integer is bit 7.

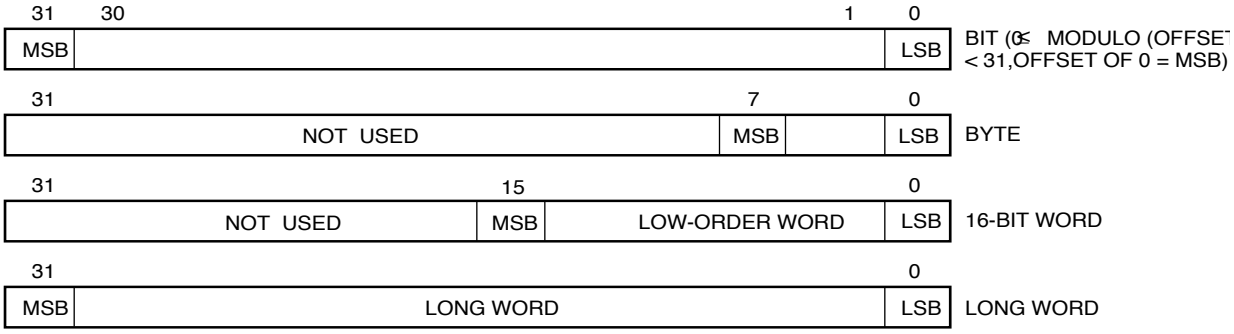


Figure 4-7. Organization of Integer Data Formats in Data Registers

Because address registers and stack pointers are 32-bits wide, address registers cannot be used for byte-size 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. When an address register is used, the entire register is affected, regardless of the operation size. If the source operand is a word size, it is sign-extended to 32 bits and then used in the

operation to an address register destination. Address registers are primarily for addresses and address computation support.

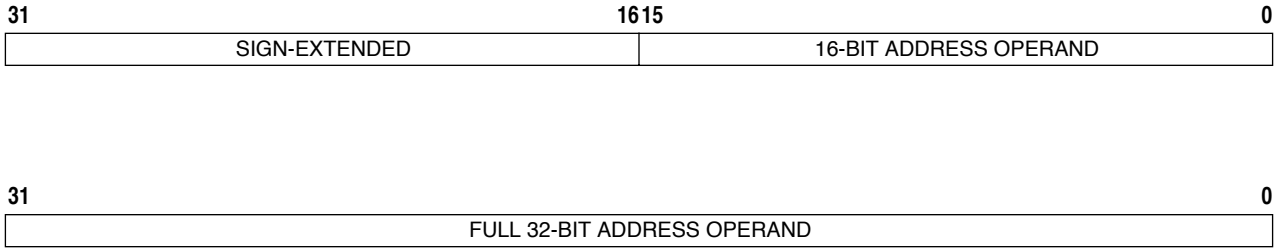


Figure 4-8. Organization of Integer Data Formats in Address Registers

Control registers vary in size according to function. Some control registers have undefined bits reserved for future definition by Motorola. Those particular bits read as zeros and must be written as zeros for future compatibility.

All operations to the SR and CCR are word-size operations. For all CCR operations, the upper byte is read as all zeros and is ignored when written, regardless of privilege mode.

4.8.2 Organization of Integer Data Formats in Memory

All ColdFire processors use a big-endian addressing scheme. The byte-addressable organization of memory 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 located at address N + 2. The address N of a word data item corresponds to the address of the high order byte. The lower order byte is located at address N + 1. This organization is shown in Figure 4-8.

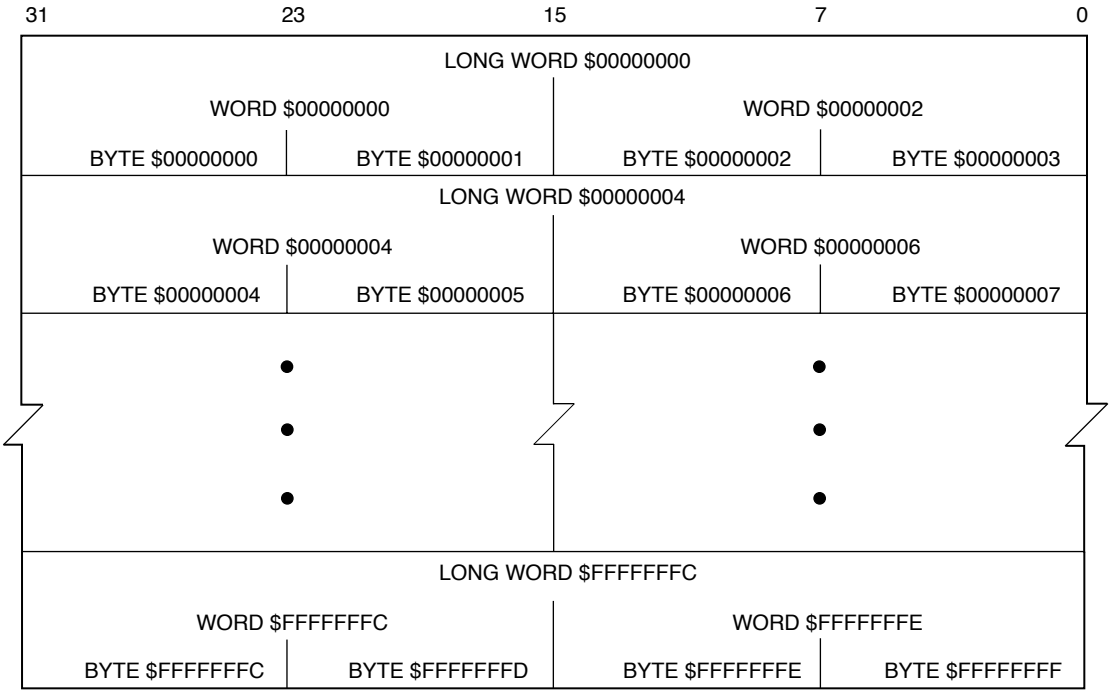


Figure 4-9. Memory Operand Addressing

4.10 ADDRESSING MODE SUMMARY

The addressing modes are grouped into categories according to the mode of use. Data addressing modes refer to data operands. Memory addressing modes refer to memory operands. Alterable addressing modes refer to alterable (writable) operands. Control addressing modes refer to memory operands without an associated size.

These categories sometimes combine to form new categories that are more restrictive. Two combined classifications are alterable memory (both alterable and memory) and data alterable (both alterable and data). Table 4-6 lists a summary of effective addressing modes

and their categories. Twelve of the most commonly used addressing modes from the M68000 Family are available on ColdFire microprocessors.

Table 4-6. Effective Addressing Modes and Categories

ADDRESSING MODES	SYNTAX	MODE FIELD	REG. FIELD	CATEGORY			
				DATA	MEMORY	CONTROL	ALTERABLE
Register Direct Data Address	 Dn An	 000 001	 reg. no. reg. no.	 X –	 – –	 – –	 X X
Register Indirect Address Address with Postincrement Address with Predecrement Address with Displacement	 (An) (An)+ –(An) (d_{16}, An)	 010 011 100 101	 reg. no. reg. no. reg. no. reg. no.	 X X X X	 X X X X	 X – – X	 X X X X
Address Register Indirect with Index 8-Bit Displacement	(d_8, An, Xi)	110	reg. no.	X	X	X	X
Program Counter Indirect with Displacement	(d_{16}, PC)	111	010	X	X	X	–
Program Counter Indirect with Index 8-Bit Displacement	(d_8, PC, Xi)	111	011	X	X	X	–
Absolute Data Addressing Short Long	 (xxx).W (xxx).L	 111 111	 000 001	 X X	 X X	 X X	 – –
Immediate	#<xxx>	111	100	X	X	–	–

4.11 INSTRUCTION SET SUMMARY

Table 4-7 lists the notational conventions used throughout this manual unless otherwise specified. Table 4-8 lists the ColdFire instruction set by opcode. This instruction set is a simplified version of the M68000 instruction set. The removed instructions include BCD, bit field, logical rotate, decrement and branch, and integer multiply with a 64-bit result. In addition, nine new MAC instructions have been added.

See Appendix B for detailed information on the instruction execution times for the Version 3 ColdFire processor core.

Table 4-7. Notational Conventions

OPCODE WILDCARDS	
cc	Logical Condition (example: NE for not equal)
REGISTER OPERANDS	
An	Any Address Register n (example: A3 is address register 3)
Ay,Ax	Source and destination address registers, respectively
Dn	Any Data Register n (example: D5 is data register 5)
Dy,Dx	Source and destination data registers, respectively
Rn	Any Address or Data Register
Ry,Rx	Any source and destination registers, respectively
Rw	Any second destination register
Rc	Any Control Register (example: VBR is the vector base register)

Table 4-7. Notational Conventions (Continued)

REGISTER/PORT NAMES	
ACC	MAC Accumulator
DDATA	Debug Data Port
CCR	Condition Code Register (lower byte of status register)
MACSR	MAC Status Register
MASK	Mask Register
PC	Program Counter
PST	Processor Status Port
SR	Status Register
MISCELLANEOUS OPERANDS	
#<data>	Immediate data following the instruction word(s)
<ea>	Effective Address
<ea>y,<ea>x	Source and Destination Effective Addresses, respectively
<label>	Assembly Program Label
<list>	List of registers (example: D3–D0)
<size>	Operand data size: Byte (B), Word (W), Longword (L)
OPERATIONS	
+	Arithmetic addition or postincrement indicator
–	Arithmetic subtraction or predecrement indicator
x	Arithmetic multiplication
/	Arithmetic division
~	Invert; operand is logically complemented
&	Logical AND
|	Logical OR
^	Logical exclusive OR
<<	Shift left (example: D0 << 3 is shift D0 left 3 bits)
>>	Shift right (example: D0 >> 3 is shift D0 right 3 bits)
→	Source operand is moved to destination operand
←→	Two operands are exchanged
sign-extended	All bits of the upper portion are made equal to the high-order bit of the lower portion
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.
SUBFIELDS AND QUALIFIERS	
{}	Optional Operation
()	Identifies an indirect address
d_n	Displacement Value, n-Bits Wide (example: d_{16} is a 16-bit displacement)
Address	Calculated Effective Address (pointer)
Bit	Bit Selection (example: Bit 3 of D0)
LSB	Least Significant Bit (example: LSB of D0)
LSW	Least Significant Word
MSB	Most Significant Bit
MSW	Most Significant Word

Table 4-7. Notational Conventions (Continued)

CONDITION CODE REGISTER BIT NAMES	
P	Branch Prediction Bit in CCR
C	Carry Bit in CCR
N	Negative Bit in CCR
V	Overflow Bit in CCR
X	Extend Bit in CCR
Z	Zero Bit in CCR

Table 4-8. Instruction Set Summary

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
ADD	Dy,<ea>x <ea>y,Dx	32 32	Source + Destination → Destination
ADDA	<ea>y,Ax	32	Source + Destination → Destination
ADDI	#<data>,Dx	32	Immediate Data + Destination → Destination
ADDQ	#<data>,<ea>x	32	Immediate Data + Destination → Destination
ADDX	Dy,Dx	32	Source + Destination + X → Destination
AND	Dy,<ea>x <ea>y,Dx	32 32	Source & Destination → Destination
ANDI	#<data>,Dx	32	Immediate Data & Destination → Destination
ASL	Dy,Dx #<data>,Dx	32 32	X/C ← (Dx << Dy) ← 0 X/C ← (Dx << #<data>) ← 0
ASR	Dy,Dx <data>,Dx	32 32	MSB → (Dx >> Dy) → X/C MSB → (Dx >> #<data>) → X/C
Bcc	<label>	8,16	If Condition True, Then PC + 2 + d_n → PC
BCHG	Dy,<ea>x #<data>,<ea>x	8,32 8,32	~(<Bit Number> of Destination) → Z, Bit of Destination
BCLR	Dy,<ea>x #<data>,<ea>x	8,32 8,32	~(<Bit Number> of Destination) → Z; 0 → Bit of Destination
BRA	<label>	8,16	PC + 2 + d_n → PC
BSET	Dy,<ea>x #<data>,<ea>x	8,32 8,32	~(<Bit Number> of Destination) → Z; 1→ Bit of Destination
BSR	<label>	8,16	SP – 4 → SP; next sequential PC→ (SP); PC + 2 + d_n → PC
BTST	Dy,<ea>x #<data>,<ea>x	8,32 8,32	~(<Bit Number> of Destination) → Z
CLR	<ea>x	8,16,32	0 → Destination
CMPI	#<data>,Dx	32	Destination – Immediate Data
CMP	<ea>y,Dx	32	Destination – Source
CMPA	<ea>y,Ax	32	Destination – Source
CPUSHL	(Ax)	none	Push and Invalidate Cache Line
DIVS	<ea>y,Dx	16 32	Dx /<ea>y → Dx {16-bit Remainder; 16-bit Quotient} Dx /<ea>y → Dx {32-bit Quotient} Signed operation
DIVU	<ea>y,Dx	16	Dx /<ea>y → Dx {16-bit Remainder; 16-bit Quotient} Dx /<ea>y → Dx {32-bit Quotient} Unsigned operation
EOR	Dy,<ea>x	32	Source ^ Destination → Destination
EORI	#<data>,Dx	32	Immediate Data ^ Destination → Destination
EXT	Dx Dx	8 → 16 16 → 32	Sign-Extended Destination → Destination
EXTB	Dx	8 → 32	Sign-Extended Destination → Destination
HALT	none	none	Enter Halted State

JMP	<ea>	none	Address of <ea> → PC
JSR	<ea>	32	SP – 4 → SP; next sequential PC → (SP); <ea> → PC
LEA	<ea>y,Ax	32	<ea> → Ax
LINK	Ax,#<data>	16	SP – 4 → SP; Ax → (SP); SP → Ax; SP + d16 → SP
LSL	Dy,Dx #<data>,Dx	32 32	X/C ← (Dx << Dy) ← 0 X/C ← (Dx << #<data>) ← 0
LSR	Dy,Dx #<data>,Dx	32 32	0 → (Dx >> Dy) → X/C 0 → (Dx >> #<data>) → X/C
MAC	Ry,Rx <shift> Ry,Rx<shift>,<ea>y,Rw	16 × 16 + 32 → 32 32 → 32	ACC + (Ry × Rx){<< 1 | >> 1} → ACC ACC + (Ry × Rx){<< 1 | >> 1} → ACC; (<ea>y{&MASK}) → Rw
MACL	Ry,Rx<shift> Ry,Rx<shift>,<ea>y,Rw	32 × 32 + 32 → 32 32 → 32	ACC + (Ry × Rx){<< 1 | >> 1} → ACC ACC + (Ry × Rx){<< 1 | >> 1} → ACC; (<ea>y{&MASK}) → Rw
MOVE	<ea>y,<ea>x	8,16,32	<ea>y → <ea>x
MOVE from ACC	ACC,Rx	32	ACC

SECTION 5
PROCESSOR-LOCAL MEMORIES

5.1 LOCAL MEMORY OVERVIEW

To maximize processor performance, there are three generic memory controllers residing on the high-speed, local bus: a unified cache controller plus RAM and ROM controllers. These controllers are designed to 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 CF3Core implementation with the optimum amount of local memory for a given application.

For all three controllers, 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:

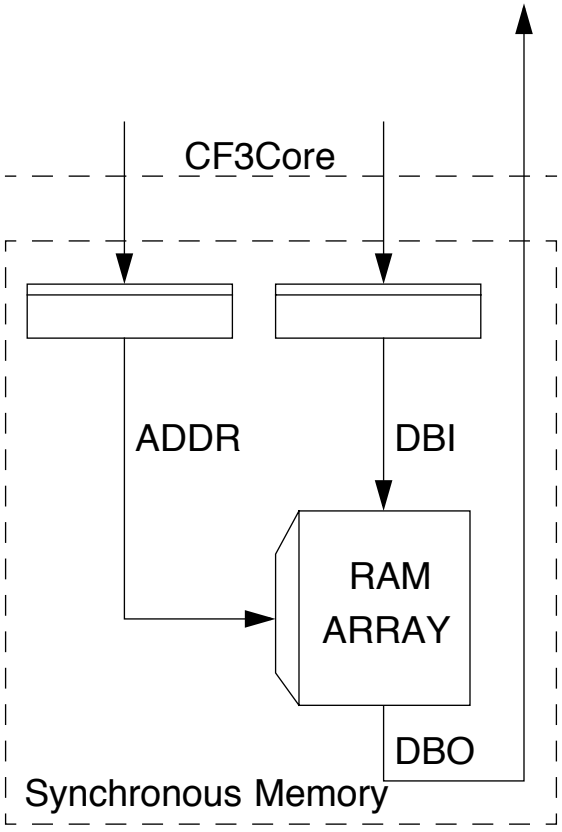


Figure 5-1. ColdFire Core Synchronous Memory Interface

where 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 the following figure, 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).

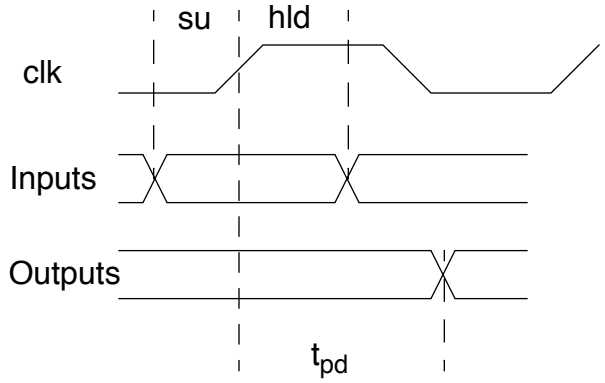


Figure 5-2. Synchronous Memory Timing Diagram

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

Consider the generic port list and functionality for a synchronous memory. See the following figure for the “black-box” diagram of the synchronous memory:

.

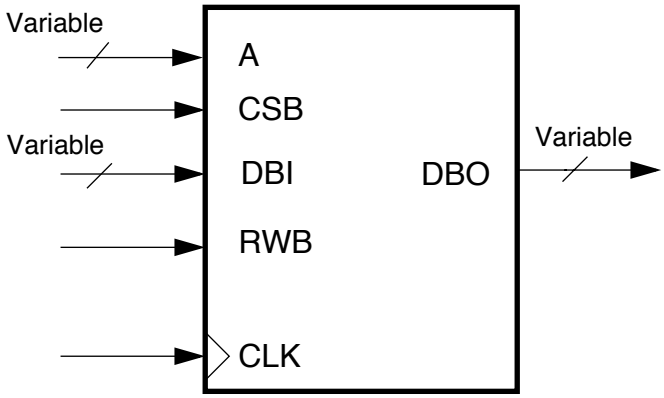


Figure 5-3. Synchronous Memory Interface Block Diagram

where 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 (Unified Cache, RAM, ROM). The port names for the memory block are defined as: 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 5-1.

Table 5-1. Synchronous Memory Truth Table (Sampled @ 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

See Appendix D for information detailing the exact CF3Core to memory connections for the unified cache, RAM and ROM compiled arrays.

5.2 THE TWO-STAGE PIPELINED LOCAL BUS (K-BUS)

Compared to the single-cycle V2 bus structure, the redesign of the processor's local KBus into a 2-stage pipelined bus represented the single largest design activity of the V3 development since the revised bus protocol affected all system elements connected to the K-Bus. This included the processor and Debug Module, as well as all the K-Bus memory controllers including the unified cache, the ROM and RAM, and the K-to-M-Bus controller.

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, since 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 the controllers/arrays to have a local registered copy of the time-critical portion of the reference address.

The KC1 access begins with the reference address contained in a register within the memory array(s). 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 {K2M} 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 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 one 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 unified cache functions within the two-stage pipelined K-Bus structure:

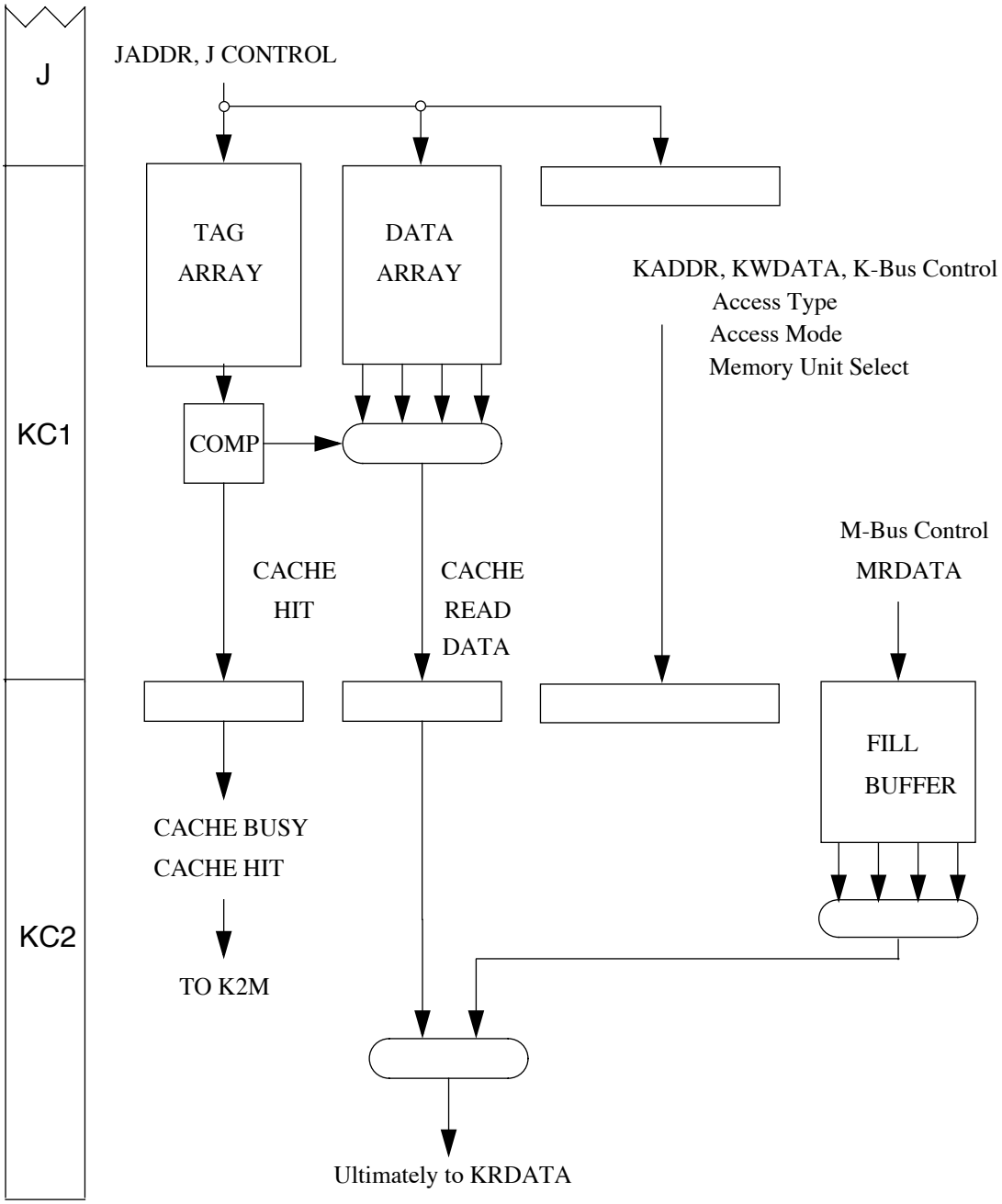


Figure 5-4. Version 3 Unified Cache Block Diagram

5.3 UNIFIED CACHE

The CF3Core design contains a non-blocking, 2 KByte,4-way set-associative, unified (instruction and 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 CF3Core instruction fetch and operand execution pipelines,

decoupling processor performance from system memory response speeds, which results in increased bus availability for alternate bus masters.

The CF3Core non-blocking cache services read hits or write hits from the processor while a fill (caused by a cache allocation) is in progress.

As shown in Figure 5-5, both instruction and operand accesses are performed using a single unified bus connected to the cache. All addresses from the processor to the cache are physical addresses. 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, and for a write operation, the data from the processor updates the cache. If the access does not match one of the cache entries (misses in the cache) or a write access must be written through to memory, the K2M (K-Bus to M-Bus) controller performs a transfer on the M-Bus and correspondingly on the external bus by way of the system bus controller (SBC). Throughout this section, all cache accesses on the internal M-Bus are assumed to have a corresponding access on the external bus performed by the System Bus Controller.

The CF3Core does not implement any type of bus snooping. Accordingly, it is the responsibility of the system software to maintain cache coherency with other possible bus masters in shared memory spaces.

5.3.1 Cache Organization

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. Figure 5-6 illustrates the cache organization (as well as the terminology used) along with the cache line format.

Table 5-2 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.

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 (discussed under **Section 5.3.2 Cache Operation**).

Each line consists of an address tag (upper 19 through 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 (the V-bit) and a dirty bit (the D-bit) 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.

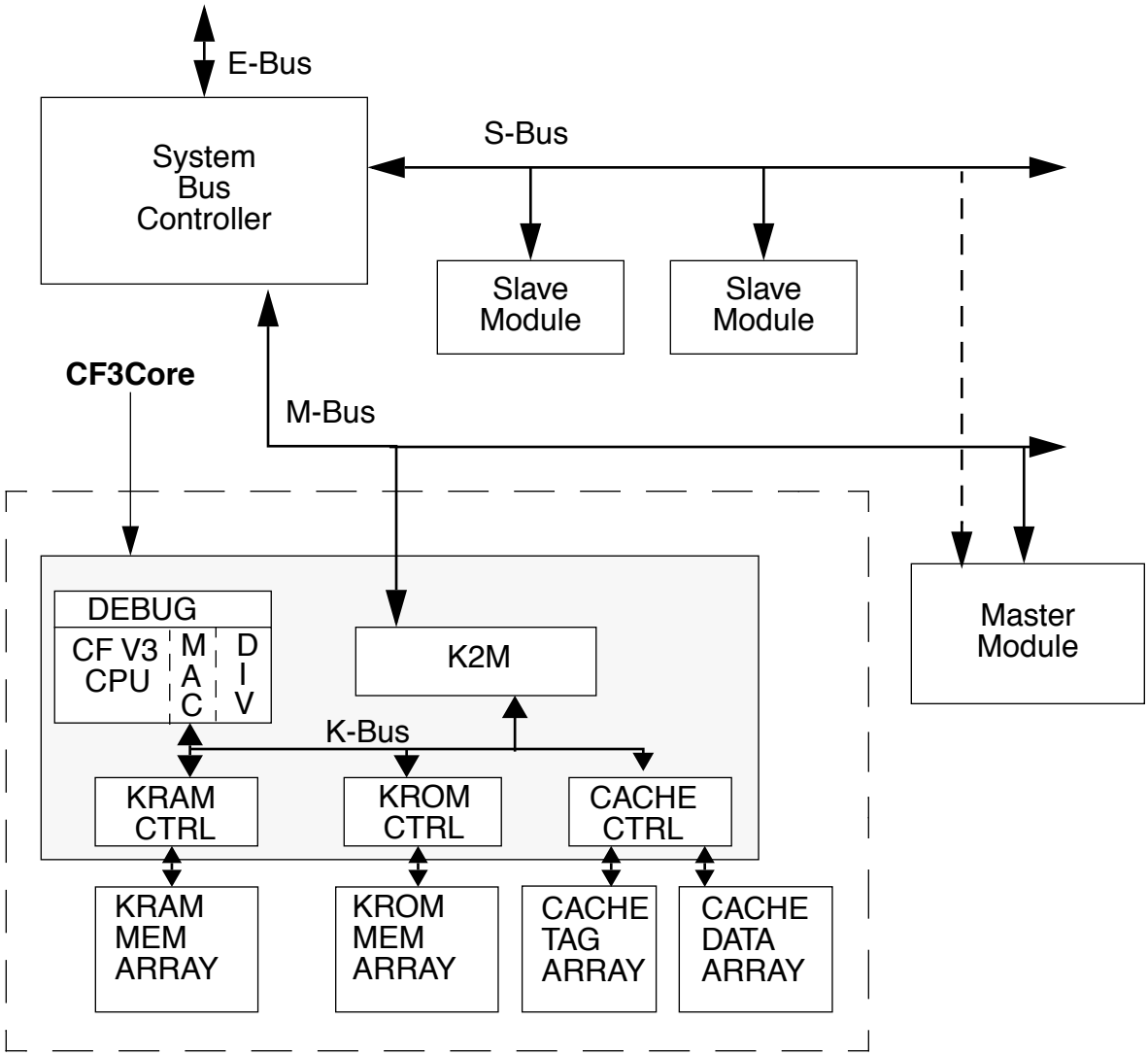


Figure 5-5. CF3Core Generic Block Diagram

Table 5-2. CF3Core Unified Cache Sizes and Configurations

A	B	C	D	E	F	G
cache size	**tag address**	**set index**	**in-line address**	**# of sets**	**# 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

5.3.2 Cache Operation

This four-way set-associative cache has a variable number of sets (based on size) of four 16 Byte lines. Each line consists of an address tag (upper 23 bits of the address), two status

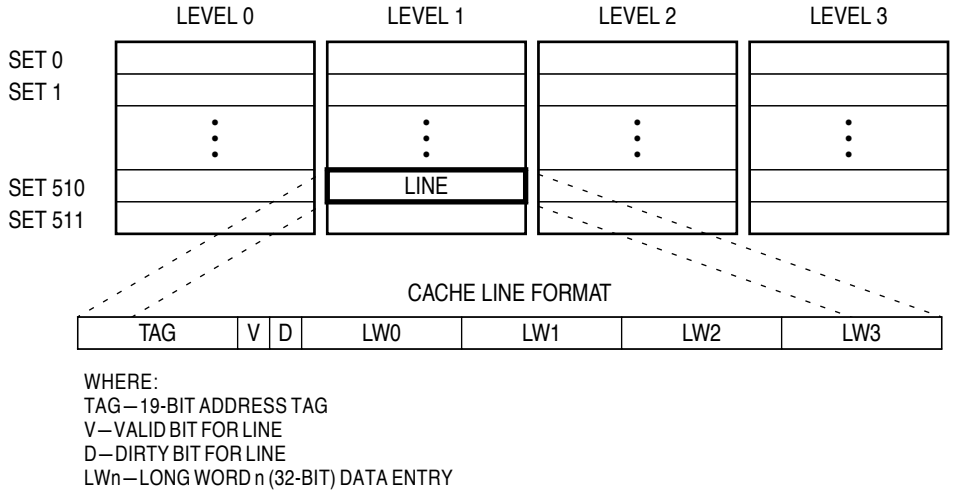


Figure 5-6. Cache Organization and Line Format (32 KByte cache size shown)

bits and four long words of data. The two status bits consist of a valid bit and a dirty bit for the line. Figure 5-7 illustrates the cache line format.

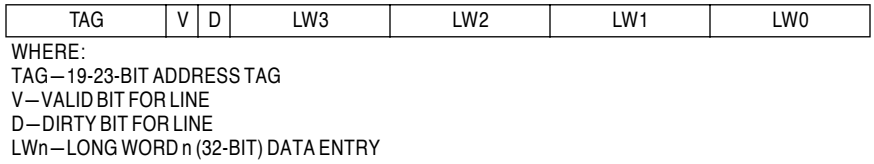


Figure 5-7. Cache Line Format

The cache stores an entire line, thereby providing validity on a line-by-line basis. For burst-mode accesses, only those that successfully read four longwords are cached.

A cache line is always in one of three states: invalid, valid, or dirty. For invalid lines, the V-bit is clear, causing the cache line to be ignored during lookups. Valid lines have their V-bit set and D-bit cleared, indicating the line contains valid data consistent with memory. Dirty cache lines have the V- and D-bits set, indicating that the line has valid entries that have not been written to memory.

A cache line changes states from valid or dirty to invalid if the execution of the CPUSHL instruction explicitly invalidates the cache line**.** The cache must be explicitly cleared by setting the CINVA bit of the CACR after a hardware reset because reset does not invalidate the cache lines. Following initial power-up, the cache contents are undefined. The V- and D-bits may be set on some lines, necessitating the clearing of the cache before it is enabled.

In the following example, a unified cache size of 32 KBytes is assumed.

Figure 5-7 illustrates the general flow of a caching operation.

To determine if the address is already allocated in the cache, (1) the cache set index (A12-A04) is used to select one cache setsets of cache lines. A set is defined as the grouping of four lines (one from each level), corresponding to the same index into the cache array. (2) The address bits of higher order (A31-A13) than the cache set index (A12-A04) are used as a tag reference or used to update the cache line tag field. (3)The four tags from the selected cache setset are compared with the tag reference. If any one of the four tags matches the tag reference and the tag status is either valid or dirty, a cache hit has occurred. (4a) A cache hit indicates that the data entries (LW0–LW3) in that cache line contain valid data (for a read access), or (4b) can be written with new data (for a write access).

To allocate an entry into the cache, the set index (A12-A04) is used to select one of the

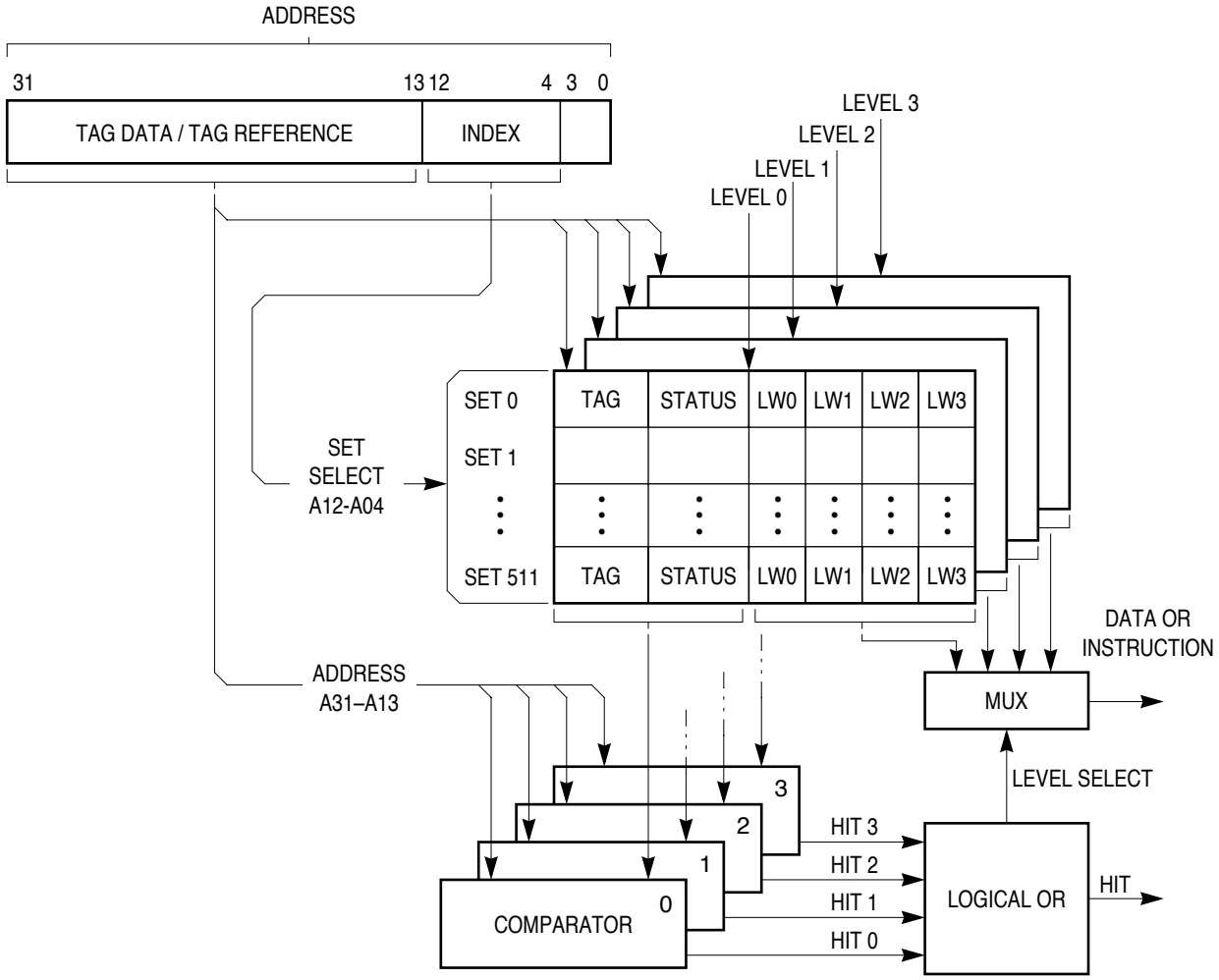


Figure 5-8. Caching Operation (32KByte cache size shown)

cache's 32512 sets of cache lines. The status of each of the four cache lines for the selected set is examined. The cache control logic first looks for an invalid cache line to use for the new entry. If there are multiple invalid entries within a set, the cache uses a fixed priority scheme to select the level to be filled: level 0 is used first, then level 1, then level 2 and

finally, level 3. If no invalid cache lines are available, a line from one of the four levels must be deallocated to host the new entry. The cache controller uses a pseudo-round-robin replacement algorithm to determine which cache line will be deallocated and replaced. After a cache line is allocated, the replacement pointer increments to point to the next level. During half-cache lock operation (HLCK equal to 1), the replacement pointer is forced to either level 2 or level 3.

In the process of deallocation, a cache line that is valid and not dirty is invalidated. A dirty cache line is placed in a push buffer (to do an external cache line push) before being invalidated. Once a cache line is invalidated, a new entry can replace it.

When a cache line is selected to host a new entry, three events happen:

1. The new address tag bits A[31:13] are written to the tag;
2. The data bits LW0–LW3 are updated with the new memory data;
3. The cache line status changes to a valid state.

Read cycles that miss in the cache allocate normally as previously described. Write cycles that miss in the cache do not allocate on a cachable writethrough region, but do allocate for addresses in a cachable copyback region**.** A copyback byte, word, or longword write miss will cause the cache to initiate a line fill, allocate space for the new line, set the status bits to indicate valid and dirty, and write the data into the allocated space. No M-Bus write to memory occurs. A copyback line write miss will not initiate a line fill, but will allocate space for the new line, set status bits to indicate valid and dirty, and write the data into the allocated space. No M-Bus write to memory occurs and no M-Bus line fill occurs. A copyback byte, word, longword, or line write miss will:

1. Cause the cache to initiate a line fill;
2. Allocate space for a new line;
3. Set the status bits to indicate valid and dirty;
4. Write the data in the allocated space. No write to memory occurs.

Read hits do not change the status of the cache line and no deallocation or replacement occurs. Write hits in cachable writethrough regions perform an external write cycle; write hits in cachable copyback regions do not perform an external write cycle. In both cases, the modified data is written into the appropriate cache entry.

If the cache hits on a read access, data is driven back to the processor core. If the cache hits on a write access, the data is written to the appropriate portion of the accessed cache line. If the data access is misaligned, the misalignment module breaks up the access into a sequence of smaller, aligned accesses. Any misaligned operand reference generates at least two accesses. Since entry validity is provided only on a line basis, the entire line must be loaded from system memory on a miss for the cache to contain any valid information for that line address.

Noncachable write accesses (i.e., those designated as cache-inhibited by the Cache Control Register (CACR) or Access Control Registers (ACR)) bypass the cache and perform

a corresponding external write. Normally, noncachable read accesses bypass the cache and the read access is performed on the external bus. The exception to this normal operation occurs when all of the following conditions are true during a noncachable read:

- Noncachable fill buffer bit (DNFB) is set in the Cache Control Register (CACR);
- Access is an instruction read;

Access is normal (i.e., transfer type (TT) equals 0.

- The appropriate noncacheable fill buffer bit (DNFB or NFB) is set
- Access is an instruction read
- Access is normal (i.e., transfer type (TT) equals 0)
- Access longword address is 0, 4, or 8 (i.e., the access is not referencing any of the last four bytes of a line)

In this case, an entire line is fetched and stored in the fill buffer. It remains valid there and the cache can service additional read accesses from this buffer until another fill occurs or a MOVEC cache invalidate all occurs.

Valid cache entries that match during noncachable address accesses are neither pushed nor invalidated. Such a scenario suggests that the associated cache mode for this address space was changed. System software must use the CPUSHL instruction to push and/or invalidate the cache entry, or set the CINVA bit of the CACR to invalidate the entire cache before switching cache modes.

5.3.3 Cache Control Register (CACR)

The CACR is a 32-bit register that contains cache control information. The CACR can be written via the MOVEC instruction (register control field of the MOVEC instruction = $002). A hardware reset clears the CACR, which disables the cache, but does not affect the tags, state information, and data within the cache. The CACR format is illustrated below. Note that all bits shown as “0” are reserved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EC	0	ESB	DPI	HLCK	0	0	CINVA	0	0	0	0	0	0	0	0	0	0	0	0	0	DNFB	DCM		0	0	DW	0	0	0	0	0

EC—Enable Cache

0 = cache disabled
1 = cache enabled

Bit 30—Reserved

ESB — Enable Store Buffer

0 = all writes to writethrough or noncachable imprecise space bypass the store buffer and generate M-Bus cycles directly.
1 = the 4-entry, first-in-first-out (FIFO) store buffer is enabled; this buffer defers pending writes to writethrough or cache-inhibited imprecise regions to maximize performance

Accesses to cache-inhibited precise space always bypass the store buffer.

DPI—Disable CPUSHL Invalidation

0 = each cache line is invalidated as it is pushed
1 = CPUSHL'd lines remain valid in the cache

HLCK—1/2 Cache Lock Mode

0 = cache operates in normal full cache mode
1 = cache operates in one-half cache lock mode

When this mode is enabled, levels 0 and 1 of the cache within a set are locked such that their lines are never be displaced or allocated. Invalid entries in levels 0 and 1 can still be allocated. This implementation allows maximum use of the available cache memory and also provides the flexibility of asserting the HLCK bit before, during, or after the needed allocations occur.

Bits 26–25—Reserved

CINVA—Cache Invalidate All

0 = no invalidation is performed
1 = initiate an invalidation of the entire cache

Setting this bit initiates invalidation of the entire cache. The cache controller sequences through all sets, clearing the valid and dirty control bits. Any subsequent K-Bus accesses are stalled until the invalidation process is finished. Once invalidation is complete, this bit is automatically returned to 0 (i.e., it does not have to be cleared by software). This bit is always read as a 0.

Bits 23–11—Reserved

DNFB—Default Noncachable Fill Buffer

0 = fill buffer is not used to store noncachable accesses
1 = fill buffer is used to store noncachable accesses

—fill buffer used only for normal (TT = 00) instruction fetches of a noncachable region from longword addresses of 0, 4, or 8
—the instructions are loaded into the fill buffer via a burst access (same as a line fill)

Note

It is possible that this feature can cause a coherency problem for self-modifying code. If enabled and a noncachable access occurs that uses the fill buffer, the instructions remain valid in the fill buffer until a MOVEC cache-invalidate-all instruction is issued, another noncachable burst, or any miss that initiates a fill occurs. If a write occurs to the address in the fill buffer, the write goes to the M-Bus without updating or invalidating the fill buffer. Any subsequent instruction reads of the given address are serviced by the fill buffer and receive the original (stale) data.

DCM—Default Cache Mode

This field selects the default cache mode and access precision as follows:

00 = cachable, writethrough
01 = cachable, copyback
10 = cache-inhibited, precise exception model
11 = cache-inhibited, imprecise exception model

Bits 7,6—Reserved

DW—Default Write Protect

This bit indicates the default write privilege.

0 = read and write accesses permitted
1 = write accesses not permitted

Bits 4–0—Reserved

5.3.4 Access Control Registers

The 32-bit Access Control Registers (ACR0 and ACR1) assign access control attributes to specific regions of the “normal” address space. The ACRs are only examined for accesses where the transfer type is zero (**TT** = 00). These registers can be written via the MOVEC instruction. (ACR0 has register control field of the MOVEC instruction = $004; ACR1 has register control field of the MOVEC instruction = $005). For overlapping regions, ACR0 takes priority. The control attributes include cache mode specification and write protection. The register below illustrates the ACR format. The following paragraphs describe the fields within the ACRs. Bits 12–7, 4, 3, 1, and 0 always read as zero. At reset, the enable bit is forced to zero, disabling the ACR functionality.

31 24	23 16	15	14 13	12	11	10	9	8	7	6 5	4	3	2	1	0
ADDRESS BASE	ADDRESS MASK	E	S-FIELD	0	0	0	0	0	0	CM	0	0	W	0	0

Bits 31–24— Address Base

This 8-bit field is compared with address bits A31–A24. Addresses that match in this comparison (and are otherwise eligible) are assigned the access control attributes of this register.

Bits 23–16— Address Mask

Since this 8-bit field contains a mask for the address base field, setting a bit in this field causes the corresponding bit in the address base field to be ignored. Regions of memory larger than 16 MBytes can be assigned the access control attributes of this register by setting some of the address mask bits. The low-order bits of this field can be set to define contiguous regions larger than 16 MBytes. The mask can define multiple noncontiguous regions of memory.

E—Enable

This bit enables or disables the access control attributes of the region defined by this register:

0 = access control attributes disabled
1 = access control attributes enabled

S-Field —Supervisor Mode

This field specifies the way the most significant bit of the transfer modifier field (**TM[2]**) is used in the address matching:

00 = match only if **TM[2]** = 0 (user mode access)
01 = match only if **TM[2]** = 1 (supervisor mode access)
1X = ignore **TM[2]** when matching

Bits 12–7—(Reserved by Motorola)

—Noncacheable Fill Buffer (NFB)

0 = fill buffer is not used to store noncacheable accesses
1 = fill buffer is used to store noncacheable accesses

—fill buffer used only for normal (TT = 0) instruction reads of a noncacheable region from longword addresses of 0, 4, or 8
—the instructions are loaded into the fill buffer via a burst access (same as a line fill)

Note

It is possible this feature can cause a coherency problem for self-modifying code. If enabled and a noncacheable access occurs that uses the fill buffer, the instructions remain valid in the fill buffer until a MOVEC, another noncacheable burst, or any miss that initiates a fill occurs. If a write occurs to the line in the fill buffer, the write will go to the M-Bus without

updating or invalidating the fill buffer. Any subsequent reads of that written data will be serviced by the fill buffer and receive stale information.

CM—Cache Mode

This field selects the cache mode and access precision as follows:

00 = cachable, writethrough
01 = cachable, copyback
10 = cache-inhibited, precise exception model
11 = cache-inhibited, imprecise exception model

W—Write Protect

This bit indicates the write privilege of the ACR region.

0 = read and write accesses permitted
1 = write accesses not permitted

Bits 4,3,1,0—Reserved by Motorola

5.3.5 Cache Management

By using the MOVEC instruction to access the CACR, system software can enable and configure the cache. A hardware reset clears the CACR, disabling the cache, and removing all configuration information, but does not affect the tags, state information, and data within the cache. The system start-up code must set the CINVA bit in the CACR to invalidate the cache before enabling it.

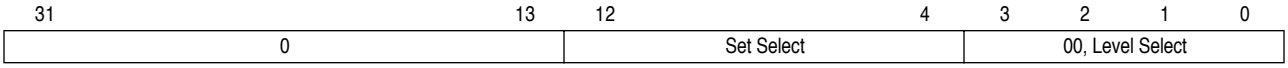
The CINVA bit of the CACR allows invalidation of the entire cache only. The privileged CPUSHL instruction supports cache management by selectively pushing and invalidating an individual cache line. The address register used with the CPUSHL instruction directly addresses the cache's directory array. The CPUSHL instruction either pushes and invalidates a line, or pushes and leaves the line valid, depending on the state of the DPI bit of the CACR. To push the entire cache, a software loop must be implemented which indexes through all 32 sets and each of the four levels within each set (for a total of 128 lines). The state of the cache enable bit in the CACR does not affect the operation of CPUSHL instruction nor the CINVA bit of the CACR.

The CPUSHL instruction pushes a modified cache line to memory and optionally invalidates the referenced entry. The format for this privileged instruction is shown below, where An is an address register.

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

The contents of the An used with the CPUSHL instruction directly specify the cache set and level indexes. This differs from the MC680x0 implementations where An specified a physical address.

The format for the An is shown below where bits A12 - A04 specify the cache set select, and bits A01-A00 define the level select. Address bits A03-A02 are not used for the instruction.



The following code example flushes the entire CF3Core unified cache (the size variable equals the number of sets for the given cache capacity):

```
_cache_flush:
       nop                     ; synchronize - flush store buffer
       clr.l       d0          ; disable cache...
       movec       d0, cacr    ; ... by clearing the cacr

       moveq.l     #4, d0      ; initialize levelCounter
setup:
       sub.l       a0, a0      ; clear address register
       move.l      #size, d1   ; initialize setCounter
       lea         -1(a0,d0),a0; include (levelCounter - 1) in address

setloop:
       cpushl      bc,(a0)     ; push the cache line identified by a0
       lea         0x10(a0),a0 ; increment setSelect by 1
       subq.l      #1, d1      ; decrement setCounter
       bne         setloop     ; are all sets for current level are done?

       subq.l      #1, d0      ; decrement levelCounter
       bne         setup       ; are all levels done?

       rts
```

5.3.6 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 Access Control Registers. An access can be cachable in either the writethrough or copyback modes, or it can be cache-inhibited in precise or imprecise modes. For normal accesses, the CM field (from the ACR) corresponding to the address of the access specifies one of these caching modes. When the access address does not match either of the ACRs, the default caching mode defined by the DCM field of the CACR is used. The specific algorithm is:

```
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 the W bit of that ACR. Addresses that do not match either of the ACRs can be write-protected using the DW bit of the CACR.

A hardware reset disables the cache and clears the CACR and ACR bits. Consequently, after reset, the defaults are writethrough cache mode and no addresses are write-protected. Note that system start-up code—and not reset—must invalidate cache entries.

The ACRs allow the defaults to be overridden. In addition, some instructions (e.g., CPUSHL) and processor core operations perform accesses that have an implicit caching mode associated with them. The following paragraphs discuss the different caching accesses and their related cache modes.

5.3.6.1 CACHABLE ACCESSES. If the CM field of an ACR or the default field of the CACR indicates writethrough or copyback, the access is cachable. A read access to a writethrough or copyback region is read from the cache if matching data is found. Otherwise, the data is read from memory and updates the cache. When a line is being read from memory for both a writethrough read miss and a copyback read miss, the longword within the line that contains the core-requested data is fetched first, and the requested data is given immediately to the processor. This operation releases the processor while the remaining three longwords of the line are read from memory and stored in the cache.

The following paragraphs describe the writethrough and copyback modes in detail.

5.3.6.1.1 Writethrough Mode. Write accesses to regions specified as writethrough are always passed on to the external bus, although the cycle can be buffered (depending on the state of the ESB bit in the CACR). Writes in writethrough mode are handled with a no-write-allocate policy, i.e., 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 instruction or data-read accesses that hit in the cache; read misses cause a new cache line to be loaded into the cache.

5.3.6.1.2 Copyback Mode. 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 D-bit without an external bus access. The dirty cache data is written to memory only if the line is replaced because of a miss or if a CPUSHL instruction pushes the line. If a byte, word,or longword, or line write access misses in the cache, the required cache line is read from memory, thereby updating the cache. If a line write access misses in the cache, the cache line will be completely sourced by the core and thus a cache line read from memory is avoided. When a miss selects a dirty cache line for replacement, the current 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.

5.3.6.2 CACHE-INHIBITED ACCESSES. System software can designate as cache-inhibited those address space regions containing targets such as I/O devices and shared data structures in multiprocessing systems. If the corresponding CM field (of the ACR) or DCM field (of the CACR) indicates precise or imprecise, then the access is cache-inhibited. The caching operation is identical for both cache-inhibited modes. The difference between these inhibited cache modes has to do with performance issues related to operand writes.

Noncachable write accesses bypass the cache and a corresponding M-Bus external write is performed. Normally, noncachable read accesses bypass the cache and the read access is performed on the external bus. The exception to this normal operation occurs when all of the following conditions are true during a noncachable read:

- The noncachable fill buffer bit (DNFB) is set in the Cache Control Register (CACR)
- Access is an instruction read
- Access is normal (i.e., transfer type (TT) equals 0)
- The appropriate noncacheable fill buffer bit (DNFB or NFB) is set
- Access is an instruction read
- Access is normal (i.e., transfer type (TT) equals 0)
- Access longword address is 0, 4, or 8 (i.e., the access is not referencing any of the last four bytes of a line).

In this case, an entire line is fetched and stored in the fill buffer. It remains valid there and the cache can service additional read accesses from this buffer until another fill occurs or a MOVEC “cache invalidate all” occurs.

If the CM field indicates either noncachable precise or noncachable imprecise modes, the cache controller bypasses the cache and performs an external transfer. If a cache line matching the current address is already resident in the cache and the cache mode for that region is cache-inhibited, the cache does not automatically push the line if it is dirty, nor does it invalidate the line if it is valid. System software must first execute a CPUSHL instruction or set the CINVA bit of the CACR (to invalidate the entire cache) prior to switching the cache mode.

If the CM field indicates precise mode, the sequence of read and write accesses to the region is guaranteed to match the sequence of the instruction order. In imprecise mode, the processor core allows read accesses that hit in processor-local memories to occur before completion of a pending write from a previous instruction. Writes are not deferred past operand read accesses that miss in the cache (i.e., that must be read from the bus). Precise operation forces operand read accesses for an instruction to occur only once by preventing the instruction from being interrupted after the operand-fetch stage. Otherwise, if not in precise mode and an exception occurs, the instruction is aborted and the operand may be accessed again when the instruction is restarted. These guarantees apply only when the CM field indicates the precise mode and the accesses are aligned.

All CPU-space register accesses (e.g., MOVEC) are always treated as noncachable and precise.

5.3.7 Cache Protocol

The following paragraphs describe the cache protocol for processor accesses and assumes that the data is cachable (i.e., writethrough or copyback).

5.3.7.1 READ MISS. A processor read that misses in the cache causes an M-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.

5.3.7.2 WRITE MISS. The cache controller handles processor writes that miss in the cache differently for writethrough and copyback regions. Byte, word, or longword, or line write misses to copyback regions load the cache line from an M-Bus line read. Line write misses to copyback regions do not cause an M-Bus line read to load the cache line. The line is completely sourced by the core, thereby avoiding the line read from memory. The new cache line is then updated with write data and the D-bit for the line is set, leaving the cache line in the dirty state. Write misses to writethrough regions write directly to memory without loading the corresponding cache line into the cache.

5.3.7.3 READ HIT. On a read hit, the cache provides the data to the processor core. No M-Bus transaction is performed 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 dirty data are not be pushed out to memory when a read hit occurs within that line. System software must first execute a CPUSHL instruction or set the CINVA bit of the CACR (to invalidate the entire cache) before switching the cache mode.

5.3.7.4 WRITE HIT. The cache controller handles processor writes that hit in the cache differently for writethrough and copyback regions. For write hits to a writethrough region, the portions of the cache line(s) corresponding to the size of the access are updated with the data. The data is also written to the external memory. The cache line state remains unchanged. If the access is copyback, the cache controller updates the cache line and sets the D-bit for the line. An external write is not performed and the cache line state changes to (or remains in) the dirty state.

5.3.8 Cache Coherency

The CF3Core provides limited support for maintaining cache coherency in multi-master environments. Both writethrough and copyback memory update techniques are supported to maintain coherency between the cache and memory.

The cache does not support snooping (i.e., cache coherency is not supported while alternate masters are using the bus).

5.3.9 Memory Accesses for Cache Maintenance

The cache controller performs all maintenance activities that supply data from the cache to the processor core. These activities include requesting accesses to the System Bus Controller for reading new cache lines and writing dirty cache lines to memory. The following paragraphs describe the memory accesses resulting from cache-fill and push operations. Refer to **Section 3.3.3 M- Bus Operation** for detailed information about the required M-Bus cycles.

5.3.10 Cache Filling

When a new cache line is required, the K2M internal bus controller requests a line read from the M-Bus. The M-Bus requests a burst-read transfer by indicating a line access with the size signals (**MSIZ[1:0]**).

The responding device supplies four longwords of data in sequence. For all cases of line-sized transfers, the critical longword defined by bits[3:2] of the miss address is accessed first, followed by the remaining three longwords, which are accessed by incrementing the address in a modulo-16 fashion as shown below:

```
if address[3:2] = 00,fetch sequence = {$0, $4, $8, $C}
if address[3:2] = 01,fetch sequence = {$4, $8, $C, $0}
if address[3:2] = 10,fetch sequence = {$8, $C, $0, $4}
if address[3:2] = 11,fetch sequence = {$C, $0, $4, $8}
```

5.3.11 Cache Pushes

When the cache controller selects a dirty cache line for replacement, memory must be updated with the dirty data before the line is replaced. 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 dirty 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 dirty cache line is written back to memory and the push buffer is invalidated.

5.3.12 Push and Store Buffers

The push buffer reduces latency for requested new data on a cache miss by temporarily holding displaced dirty data while the new data is fetched from memory. The push buffer contains16 Bytes of storage (one displaced cache line).

If a cache miss displaces a dirty line, the miss read reference is immediately placed on the M-Bus. While waiting for the response, the current contents of the cache location load into the push buffer. Once the bus transaction (burst read) completes, the cache controller can generate the appropriate line-write bus transaction to write the contents of the push buffer to memory.

The store buffer implements a FIFO buffer that can defer pending writes to imprecise regions in order to maximize performance. The store buffer can support as many as four entries (16 Bytes maximum) for this purpose.

For operand writes destined for the store buffer, the processor core incurs no stalls. The store buffer effectively provides a measure of decoupling between the pipeline's ability to generate writes (one write per cycle maximum) and the ability of the M-Bus to retire those writes. When writing to imprecise regions, a stall occurs only if the store buffer is full and a write operation is present on the K-Bus. In this case, the K-Bus write cycle is held, stalling the processor's operand execution pipeline.

If the store buffer is not used (i.e., store buffer disabled or cache-inhibited precise mode), M-Bus cycles are generated directly for each pipeline write operation. The next instruction is held in the AGEX cycle of the operand execution pipeline (OEP) until external bus transfer termination is received. This means each write operation is stalled for five cycles, making the minimum write time equal to six cycles when the store buffer is not used.

The store buffer enable bit (ESB bit of the CACR) controls the enabling of the store buffer. This bit can be set and cleared via the MOVEC instruction. At reset, this bit is cleared and all writes are precise. The ACR CM field or CACR DCM field generates the mode used when this bit is set. The cachable writethrough and the cache-inhibited imprecise modes use the store buffer.

The store buffer can queue as much as four bytes of data per entry. Each entry matches the corresponding bus cycle it will generate. Therefore, a misaligned longword write to a writethrough region creates two entries if the address is to an odd-word boundary, three entries if to an odd-byte boundary—one per bus cycle.

5.3.12.1 PUSH AND STORE BUFFER BUS OPERATION. Once the push or store buffer has valid data, the K2M 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 (e.g., cache miss to process) during the continued instruction execution by the processor pipeline, the pipeline stalls until the push and store buffers are empty before generating the required M-Bus transaction.

Certain instructions and exception processing that synchronize the processor core guarantee the push and store buffers are empty before proceeding.

5.3.13 Cache Operation Summary

The following paragraphs summarize the operational details for the cache and present state diagrams depicting the cache line state transitions.

The cache supports a line-based protocol allowing individual cache lines to be in one of three states: invalid, valid, or dirty. To maintain coherency with memory, the cache supports both writethrough and copyback modes, specified by the CM field for the matched ACR or the DCM field of the CACR if no ACR matches.

Read misses and write misses to copyback regions cause the cache controller to read a new cache line from memory into the cache. If available, tag and data from memory update an invalid line in the selected setset. The line state then changes from invalid to valid by setting the V-bit for the line. If all lines in the set are already valid or dirty, the pseudo-round-robin replacement algorithm selects one of the four lines and replaces the tag and data contents of the line with the new line information. Before replacement, dirty lines are temporarily stored in the push buffer and later copied back to memory after the new line has been read from memory. Figure 5-9 illustrates the three possible states for a cache line, with the possible processor-initiated transitions. Transitions are labeled with a capital letter, indicating the previous state, followed by a number indicating the specific case listed in Table 5-3.

Table 5-3. Cache Line State Transitions

CACHE OPERATION	CURRENT STATE					
	INVALID CASES		VALID CASES		DIRTY CASES	
READ MISS	(C,W)I1	Read line from memory and update cache; Supply data to processor; Go to valid state.	(C,W)V1	Read new line from memory and update cache; supply data to processor; Remain in current state.	CD1	Push dirty cache line to push 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	(C,W)I2	Not possible.	(C,W)V2	Supply data to processor; Remain in current state.	CD2	Supply data to processor; Remain in current state.
WRITE MISS (COPY-BACK MODE)	CI3	Read line from memory and update cache; Write data to cache; Go to dirty state.	CV3	Read new line from memory and update cache; Write data to cache; Go to dirty state.	CD3	Push dirty cache line to push buffer; Read new line from memory and update cache; Write push buffer contents to memory; Remain in current state.
WRITE MISS (WRITE-THROUGH MODE)	WI3	Write data to memory; Remain in current state.	WV3	Write data to memory; Remain in current state.	WD3	Write data to memory; Remain in current state.
WRITE HIT (COPY-BACK MODE)	CI4	Not possible.	CV4	Write data to cache; Go to dirty state.	CD4	Write data to cache; Remain in current state.
WRITE HIT (WRITE-THROUGH MODE)	WI4	Not possible.	WV4	Write data to memory and to cache; Remain in current state.	WD4	Write data to memory and to cache; Go to valid state.
CACHE INVALIDATE	(C,W)I5	No action; Remain in current state.	(C,W)V5	No action; Go to invalid state.	CD5	No action (dirty data lost); Go to invalid state.
CACHE PUSH	(C,W)I6	No action; Remain in current state.	(C,W)V6	No action; Go to invalid state.	CD6	Push dirty cache line to memory; Go to invalid state.
CACHE PUSH	(C,W)I7	No action; Remain in current state.	(C,W)V7	No action; Remain in current state.	CD7	Push dirty cache line to memory; Go to valid state

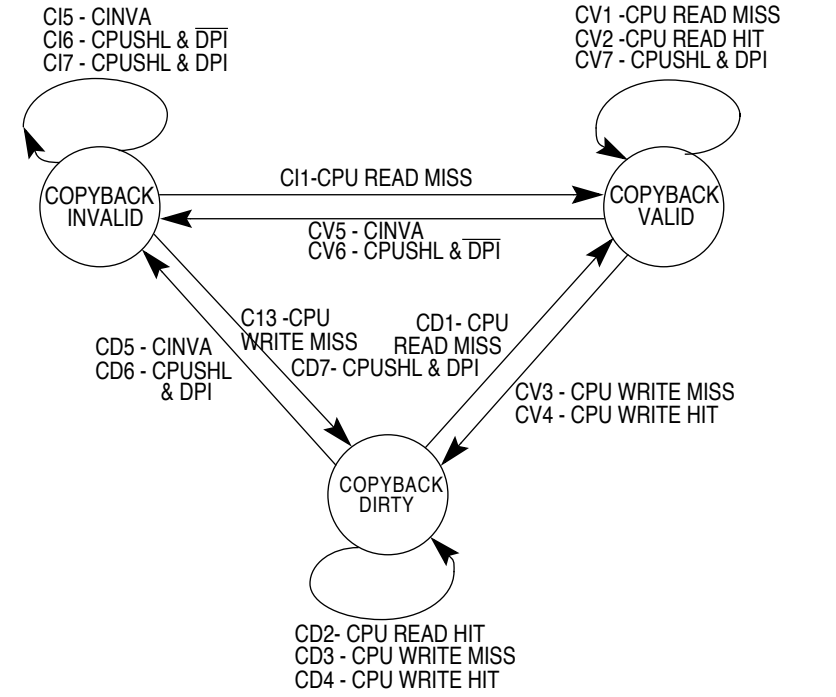
Note

The shaded areas indicate that the cache mode has changed for the region corresponding to this cache line. In writethrough mode, a cache line should never be dirty.

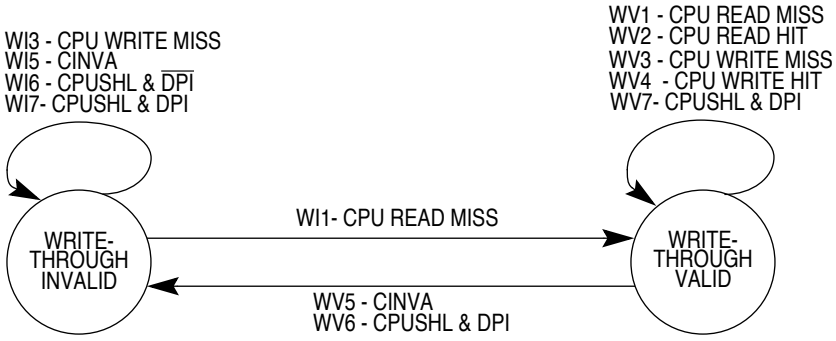
To avoid these states:

1. Execute a CPUSHL instruction, or

2. Set the CINVA bit of the CACR (to invalidate the entire cache) before switching the cache mode.



COPYBACK CACHING MODE



WRITETHROUGH CACHING MODE

Figure 5-9. Cache Line State Diagrams

5.4 PROCESSOR-LOCAL RANDOM ACCESS MEMORY (RAM)

The Version 3 ColdFire processor core supports a local random-access memory with the following features:

- 0 - 32 KByte capacity, organized as (size/4) x 32 Bits
- RAM size defined by static core input signals, **KRAM_SZ[2:0]**
- Pipelined Single-Cycle Access
- Physically Located on Processor's High-Speed Local Bus
- Memory Location Programmable on any (0-Modulo-size) Address
- Programmable Memory Address Space Mappings
- Byte, Word, Longword Addressable

5.4.1 RAM Operation

The RAM module provides a general-purpose memory region that the ColdFire processor can access in a single cycle. The location of the memory can be specified to any 0-modulo-size address within the four gigabyte address space. The memory is ideal for storing critical code or data structures or for use as the system stack. Since the RAM module is physically connected to the processor's high-speed local bus, it can service processor-initiated accesses or memory-referencing commands from the Debug Module.

Depending on configuration information, instruction fetches and operand references may be sent to all local memory controllers simultaneously. The memory controllers implement a fixed priority scheme which determines the appropriate responding device. The RAM is treated as the highest priority memory, followed by the ROM, and then the unified cache. If the read reference is mapped into the region defined by the RAM, it provides the data back to the processor, and any ROM or cache data is discarded. Accesses from the RAM module are not cached.

5.4.2 RAM Programming Model

The configuration information in the RAM Base Address Register (RAMBAR) controls the operation of the RAM module.

- The RAMBAR is the register that holds the base address of the RAM. The RAMBAR is accessed as control register $C04 using the privileged MOVEC instruction. The MOVEC instruction provides write-only access to this register.
- The RAMBAR register can be accessed from the Debug Module in a similar manner. From the Debug Module, the register can be read or written.
- All undefined bits in the register are reserved. These bits are ignored during writes to the RAMBAR, and return zeroes when read from the Debug Module.
- The RAMBAR valid bit is cleared by reset, disabling the RAM module. All other bits are unaffected.

The RAMBAR register contains four control fields. These fields are detailed in the following subsections. The next illustration defines the format of the RAM Base Address Register (RAMBAR).

RAM Base Address Register (RAMBAR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BA31	BA30	BA29	BA28	BA27	BA26	BA25	BA24	BA23	BA22	BA21	BA20	BA19	BA18	BA17	BA16
RESET:															
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BA15	BA14	BA13	BA12	BA11	BA10	BA09	WP	-	-	C/I	SC	SD	UC	UD	V
RESET:															
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0

RAM Base Address Register (RAMBAR)

BA[31:9] - Base Address

This field defines the 0-modulo-size base address of the RAM module. The RAM memory occupies a size-space defined by the contents of the Base Address field. By programming this field, the RAM may be located on any 0-modulo-size boundary within the processor's four gigabyte address space. The number of bits in this field is a function of the RAM size:

Table 5-4. RAM Base Address Bits

RAM Size	BA Field Bits
512	BA[31:09]
1K	BA[31:10]
2K	BA[31:11]
4K	BA[31:12]
8K	BA[31:13]
16K	BA[31:14]
32K	BA[31:15]

WP - Write Protect

This field allows only read accesses to the RAM. When this bit is set, any attempted write access generates an access error exception to the ColdFire processor core.

0 = Allow read and write accesses to the RAM module
1 = Allow only read accesses to the RAM module

C/I, SC, SD, UC, UD - Address Space Masks

This five-bit field allows certain types of accesses to be “masked,” or inhibited from accessing the RAM module. The address space mask bits are:

- C/I = CPU Space/Interrupt acknowledge cycle mask
- 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 address space bit:

0 = An access to the RAM module can occur for this address space
1 = Disable this address space from the RAM module. If a reference using this address space is made, it is inhibited from accessing the RAM module, and is processed like any other non-RAM reference.

In particular, the C/I mask bit is normally set. These bits are useful for power management as detailed in **Section 5.4.5: RAM Power Management.**

V - Valid

The valid bit (V-bit) is specified by RAMBAR[0]. A hardware reset clears this bit. When set, this bit enables the RAM module; otherwise, the module is disabled.

0 = Contents of RAMBAR are not valid
1 = Contents of RAMBAR are valid

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 (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
```

where ASn refers to the five address space masks (C/I, SC, SD, UC, UD).

5.4.3 RAM Initialization

After a hardware reset, the contents of the RAM module are undefined. The valid bit of the RAMBAR is cleared, disabling the module. If the RAM needs to be initialized with instructions or data, perform the following steps:

1. Load the RAMBAR mapping the RAM module to the desired location within the address space.
2. Read the source data and write it to the RAM. There are various instructions to support this function, including memory-to-memory move instructions, or the MOVEM opcode. The MOVEM instruction is optimized to generate line-sized burst fetches on 0-modulo-16 addresses, so this opcode generally provides maximum performance.
3. After the data has been loaded into the RAM, it may be appropriate to load a revised value into the RAMBAR with a new set of “attributes.” These attributes consist of the write-protect and address space mask fields.

The ColdFire processor or an external emulator using the Debug Module can perform these initialization functions.

5.4.4 RAM Initialization Code

The code segment below describes how to initialize a 4 KByte RAM. The code sets the base address of the RAM at $20000000 and then initializes it to zeros.

```
RAMBAR_BAEQU$20000000     ;define the RAMBAR Base Address +
                          ;set this variable to $20000000
RAMBAR_VEQU$1             ;define the RAMBAR valid bit

move.l #RAMBAR_BA+RAMBAR_V,D0 ;load RAMBAR base address + valid into D0
movec.lD0, RAMBAR         ;load RAMBAR and enable RAM

; the following loop initializes the entire RAM to zero

lea.l  RAMBAR_BA,A0       ;load base address pointer to RAM
move.l #1024,D0           ;load loop counter into D0

RAM_INIT_LOOP:
clr.l  (A0)+              ;clear 4 bytes of RAM
subq.l #1,D0              ;decrement loop counter
bne    RAM_INIT_LOOP      ;if done, then exit; else continue looping
```

5.4.5 RAM Power Management

As noted previously, depending on the configuration defined by the RAMBAR, instruction fetch and operand read accesses may be sent to the RAM and unified cache simultaneously. If the access is mapped to the RAM module, it sources the read data, and the unified cache access is discarded. If the RAM is used only for data operands, asserting the ASn bits associated with instruction fetches can decrease power dissipation. Additionally, if the RAM contains only instructions, masking operand accesses can reduce power dissipation.

Consider the examples in Table 5-5 of typical RAMBAR settings:

Table 5-5. Examples of Typical RAMBAR Settings

DATA CONTAINED IN RAM	RAMBAR[7:0]
Code Only	$2B
Data Only	$35
Both Code And Data	$21

5.5 PROCESSOR-LOCAL READ-ONLY MEMORY (ROM)

The Version 3 ColdFire processor core supports a local read-only memory with the following features:

- 0 - 32 KByte capacity, organized as (size/4) x 32 Bits
- ROM size defined by static core input signals, **KROM_SZ[2:0]**
- Pipelined Single-Cycle Access
- Physically Located on Processor's High-Speed Local Bus
- Memory Location Programmable on any (0-Modulo-size) Address
- Programmable Memory Address Space Mappings
- Byte, Word, Longword Addressable
- Configurable at reset to serve as boot memory

5.5.1 ROM Operation

The ROM module provides a general-purpose memory region that the ColdFire processor can access in a single cycle. The location of the memory can be specified to any 0-modulo-size address within the four gigabyte address space. The memory is ideal for storing critical code or read-only data structures. By asserting a CF3Core input signal, the ROM can be configured to act as the boot memory device, i.e., the ROM can be based at address 0 and made valid at reset. Since the ROM module is physically connected to the processor's high-speed local bus, it can service processor-initiated accesses or memory-referencing commands from the Debug Module.

Depending on configuration information, instruction fetches and operand references may be sent to all local memory controllers simultaneously. The memory controllers implement a fixed priority scheme which determines the appropriate responding device. The RAM is treated as the highest priority memory, followed by the ROM, and then the unified cache. If the read reference is mapped into the region defined by the RAM, it provides the data back to the processor, and any ROM or cache data is discarded. Accesses from the ROM module are not cached.

5.5.2 ROM Programming Model

The configuration information in the ROM Base Address Register (ROMBAR) controls the operation of the ROM module.

- The ROMBAR is the register that holds the base address of the ROM. The ROMBAR is accessed as control register $C00 using the privileged MOVEC instruction. The MOVEC instruction provides write-only access to this register.
- The ROMBAR register can be accessed from the Debug Module in a similar manner. From the Debug Module, the register can be read or written.
- All undefined bits in the register are reserved. These bits are ignored during writes to the ROMBAR, and return zeroes when read from the Debug Module.
- The initial value of the ROMBAR is controlled by the state of a CF3Core input pin. If the input signal **KROMVLDRST** is set at reset time, the contents of the ROMBAR is forced to $0000_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 ROMBAR valid bit is cleared by reset, disabling the ROM module. All other bits are unaffected.

The ROMBAR register contains four control fields. These fields are detailed in the following subsections.

ROM Base Address Register (ROMBAR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BA31	BA30	BA29	BA28	BA27	BA26	BA25	BA24	BA23	BA22	BA21	BA20	BA19	BA18	BA17	BA16
RESET: -	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BA15	BA14	BA13	BA12	BA11	BA10	BA09	WP	-	-	C/I	SC	SD	UC	UD	V
RESET: -	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0

ROM Base Address Register (ROMBAR)

BA[31:9] - Base Address

This field defines the 0-modulo-size base address of the ROM module. The ROM memory occupies a size-space defined by the contents of the Base Address field. By programming this field, the ROM may be located on any 0-modulo-size boundary within the processor's four gigabyte address space. The number of bits in this field is a function of the ROM size:

Table 5-6. ROM Base Address Bits

ROM Size	BA Field Bits
512	BA[31:09]
1K	BA[31:10]
2K	BA[31:11]

Table 5-6. ROM Base Address Bits

ROM Size	BA Field Bits
4K	BA[31:12]
8K	BA[31:13]
16K	BA[31:14]
32K	BA[31:15]

WP - Write Protect

This field is reserved for future use. It can be used for debugging purposes, since if this bit is set, any attempted write access generates an access error exception to the ColdFire processor core.

0 = Allow read and write accesses to the ROM module
1 = Allow only read accesses to the ROM module

C/I, SC, SD, UC, UD - Address Space Masks

This five bit field allows certain types of accesses to be “masked,” or inhibited from accessing the ROM module. The address space mask bits are:

- C/I = CPU Space/Interrupt acknowledge cycle mask
- 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 address space bit:

0 = An access to the ROM module can occur for this address space
1 = Disable this address space from the ROM module. If a reference using this address space is made, it is inhibited from accessing the ROM module, and is processed like any other non-ROM reference.

In particular, the C/I mask bit is normally set. These bits are useful for power management as detailed in **Section 5.5.3: ROM Power Management.**

V - Valid

The initial state of the ROMBAR valid bit is controlled by the value of a CF3Core input pin. If the input signal **KROMVLDRST** is set at reset time, the contents of the ROMBAR is forced to $0000_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 ROMBAR valid bit is cleared by reset, disabling the ROM module.

When set, this bit enables the ROM module; otherwise, the module is disabled.

0 = Contents of ROMBAR are not valid
1 = Contents of ROMBAR are valid

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 (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] = 1)
                     Signal a write-protect access error
```

where ASn refers to the five address space masks (C/I, SC, SD, UC, UD).

5.5.3 ROM Power Management

As noted previously, depending on the configuration defined by the ROMBAR, instruction fetch and operand read accesses may be sent to the local memory controllers simultaneously. If the access is mapped to the ROM module, it sources the read data, and the unified cache access is discarded. If the ROM is used only for data operands, asserting the ASn bits associated with instruction fetches can decrease power dissipation. Additionally, if the ROM contains only instructions, masking operand accesses can reduce power dissipation.

Consider the examples in Table 5-7 of typical ROMBAR settings:

Table 5-7. Examples of Typical ROMBAR Settings

DATA CONTAINED IN ROM	ROMBAR[7:0]
Code Only	$2B
Read-Only Data Only	$35
Both Code And Data	$21

5.6 INTERACTIONS BETWEEN THE KBUS MEMORIES

Depending on configuration information, instruction fetches and operand read accesses may be sent to all three of the K-Bus memory controllers, i.e., the RAM, the ROM and the unified cache simultaneously. This approach is required since all three controllers are memory-mapped devices and the hit/miss determination is made concurrently with the read data access. As previously discussed, power dissipation can be minimized by configuring the RAMBAR and ROMBAR control registers to mask unused address spaces whenever possible.

If the access address is mapped into the region defined by the RAM (and this region is not masked), the RAM provides the data back to the processor, and the ROM and unified cache data is discarded. Accesses from the RAM module are never encached. The ROM behaves

similarly, although its priority is below that of the RAM. The complete definition of the processor's local bus priority scheme for read references is:

```
if (RAM "hits")
     RAM supplies data to the processor
else if (ROM "hits")
          ROM supplies data to the processor
     else if (unified cache "hits")
               unified cache supplies data to the processor
          else M-Bus reference to access data
```

For operand write references, the memory-mapping into the local memories is resolved before the appropriate destination memory is accessed. Accordingly, only the targeted local memory is accessed for operand write transfers.

SECTION 6
DEBUG SUPPORT

This section details the hardware debug support functions within the ColdFire family of processors. The Version 3 ColdFire implements an enhanced debug architecture compared to the original specification. The original design plus these enhancements is known as Revision B (or Rev. B), while the initial definition is Revision A (or Rev. A). The enhanced functionality is clearly identified in this section. The Rev. B enhancements are backward compatible with the original ColdFire debug definition.

The general topic of debug support is divided into three separate areas:

- Real-Time Trace Support
- Background Debug Mode (BDM)
- Real-Time Debug Support

Each of the three areas is addressed in detail in the following subsections.

Version 3 ColdFire processors implement the enhanced Revision B debug module definition. Enhancements include the following:

- Serial BDM command to display current program counter without halting the CPU
- Added capability to logically OR hardware breakpoint triggers
- Added registers to support concurrent BDM commands and active breakpoints
- An external mechanism to generate a debug interrupt
- A program-visible register field to identify the debug module revision

The logic required to support these three areas is contained in a Debug Module, which is shown in the system block diagram in Figure 6-1.

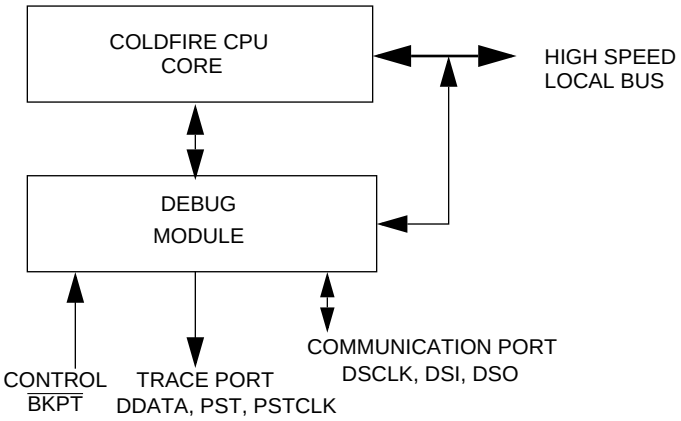


Figure 6-1. Processor/Debug Module Interface

6.1 SIGNAL DESCRIPTION

This section describes the ColdFire signals associated with the Debug Module. All ColdFire debug signals are unidirectional and related to the rising-edge of the processor core's clock signal.

6.1.1 Breakpoint ($\overline{\text{BKPT}}$)

6.1.1.1 REV A FUNCTIONALITY. This active-low, input signal is used to request a manual breakpoint. It's assertion causes the processor to enter a halted state after the completion of the current instruction. The halt status is reflected on the processor status (PST) pins as the value $F.

6.1.1.2 REV. B ENHANCEMENT. In addition to the baseline functionality, if the BKD bit of the Configuration/Status Register is set (CSR[18]), the assertion of the $\overline{\text{BKPT}}$ signal generates a debug interrupt exception in the processor.

6.1.2 Debug Data (DDATA[3:0])

These output signals display the hardware register breakpoint status as a default, or optionally, captured address and operand values. The capturing of data values is controlled by the setting of the CSR. Additionally, execution of the WDDATA instruction by the processor captures operands which are displayed on DDATA. These signals are updated each processor cycle.

6.1.3 Development Serial Clock (DSCLK)

This input signal is synchronized internally and provides the clock for the serial communication port to the Debug Module. The maximum frequency is 1/5 the speed of the processor's clock (CLK). At the synchronized rising edge of DSCLK, the data input on DSI is sampled, and the DSO output changes state.

6.1.4 Development Serial Input (DSI)

The input signal is synchronized internally and provides the data input for the serial communication port to the Debug Module.

6.1.5 Development Serial Output (DSO)

This signal provides serial output communication for the Debug Module responses.

6.1.6 Processor Status (PST[3:0])

These output signals report the processor status. Table 6-1 shows the encoding of these signals. These outputs indicate the current status of the processor pipeline and, as a result, are not related to the current bus transfer. The PST value is updated each processor cycle.

6.1.7 Processor Status Clock (PSTCLK)

Since the debug trace port signals transition each processor cycle and are not related to the external bus frequency, an additional signal is output from the ColdFire microprocessor. The PSTCLK signal is a delayed version of the processor's high-speed clock and its rising-edge is used by the development system to sample the values on the PST and DDATA output buses. The PSTCLK signal is intended for use in the standard 26-pin debug connector.

If the real-time trace functionality is not being used, the PCD bit of the CSR may be set (CSR[17] = 1) to force the PSTCLK, PST and DDATA outputs to be quiescent and not toggle at the processor's clock speed.

Table 6-1. Processor Status Encoding

PST[3:0]		DEFINITION
(HEX)	(BINARY)	
$0	0000	Continue execution
$1	0001	Begin execution of an instruction
$2	0010	Reserved
$3	0011	Entry into user-mode
$4	0100	Begin execution of **PULSE** and **WDDATA** instructions
$5	0101	Begin execution of taken branch or Sync_PC[1]
$6	0110	Reserved
$7	0111	Begin execution of **RTE** instruction
$8	1000	Begin 1-byte transfer on DDATA
$9	1001	Begin 2-byte transfer on DDATA
$A	1010	Begin 3-byte transfer on DDATA
$B	1011	Begin 4-byte transfer on DDATA
$C	1100	Exception processing†
$D	1101	Emulator-mode entry exception processing†
$E	1110	Processor is stopped, waiting for interrupt†
$F	1111	Processor is halted †

NOTE: †These encodings are asserted for multiple cycles.

1 Rev. B ehnancement.

6.2 REAL-TIME TRACE SUPPORT

In the area of debug functions, one fundamental requirement is support for real-time trace functionality, i.e., definition of the dynamic execution path. The ColdFire solution is to include a parallel output port providing encoded processor status and data to an external development system. This port is partitioned into two nibbles (4 bits): one nibble allows the processor to transmit information concerning the execution status of the core (processor status, PST), while the other nibble allows operand data to be displayed (debug data, DDATA). The processor status (PST) timing is synchronous with the processor clock (CLK) and may not be related to the current bus transfer. Table 6-1 shows the encoding of these signals.

The processor status (PST) outputs can be used with an external image of the program to completely track the dynamic execution path of the machine when used with external development systems. The tracking of this dynamic path is complicated by any change-of-flow operation. This is especially evident when the branch target address is calculated based on the contents of a program-visible register (variant addressing.) For this reason, the debug data (DDATA) outputs can be configured to display the target address of these types of change-of-flow instructions. Because the DDATA bus is only 4 bits wide, the address is displayed a nibble at a time across multiple clock cycles.

The Debug Module includes two 32-bit storage elements for capturing the internal ColdFire bus information. These two elements effectively form a FIFO buffer connecting the processor's high-speed local bus to the external development system through the DDATA signals. The FIFO buffer captures branch target addresses along with certain operand data values for eventual display on the DDATA output port on nibble at a time starting with the least-significant bit. The execution speed of the ColdFire processor is affected only when both storage elements contain valid data waiting to be dumped onto the DDATA port. In this case, the processor core is stalled until one FIFO entry is available. In all other cases, data output on the DDATA port does not impact execution speed.

6.2.1 Processor Status Signal Encoding

The processor status (PST) signals are encoded to reflect the state of the Operand Execution Pipeline, and are generally not related to the current external bus transfer.

6.2.1.1 CONTINUE EXECUTION (PST = $0). Many instructions complete in a single processor cycle. If an instruction requires more clock cycles, the subsequent clock cycles are indicated by driving the PST outputs with this encoding.

6.2.1.2 BEGIN EXECUTION OF AN INSTRUCTION (PST = $1). 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.

6.2.1.3 ENTRY INTO USER MODE (PST = $3). This encoding indicates the ColdFire processor has entered user mode. This encoding is signaled after the instruction which caused the user mode entry has executed (signaled with the appropriate encoding.)

6.2.1.4 BEGIN EXECUTION OF PULSE OR WDDATA INSTRUCTIONS (PST = $4). The ColdFire instruction set architecture includes a PULSE opcode. This opcode generates a

unique PST encoding, $4, when executed. This instruction can define logic analyzer triggers for debug and/or performance analysis. Additionally, a WDDATA instruction is supported that allows the processor core to write any operand (byte, word, longword) directly to the DDATA port, independent of any Debug Module configuration. This opcode also generates the special PST encoding ($4) when executed, followed by the appropriate marker and then the data transfer on the DDATA outputs. The length of the data transfer is dependent on the operand size of the WDDATA instruction.

6.2.1.5 BEGIN EXECUTION OF TAKEN BRANCH (PST = $5). This valueis generated whenever a taken branch is executed. For certain opcodes, the branch target address may be optionally displayed on DDATA depending on the control parameters contained in the configuration/status register (CSR). The number of bytes of the address to be displayed is also controlled in the CSR and indicated by the PST marker value immediately preceding the DDATA outputs.

The bytes are always displayed in a least-significant to most-significant order. The processor captures only those target addresses associated with taken branches using a variant addressing mode, i.e., all JMP and JSR instructions using address register indirect or indexed addressing modes, all RTE and RTS instructions as well as 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 these types of change-of-flow operations, the ColdFire processor uses the debug pins to output a sequence of information on successive processor clock cycles

1. Identify a taken branch has been executed using the PST pins ($5).
2. Using the PST pins, optionally signal the target address is to be displayed on the DDATA pins. The encoding ($9, $A, $B) identifies the number of bytes that are displayed.
3. The new target address is optionally available on subsequent cycles using the nibble-wide DDATA port. The number of bytes of the target address displayed on this port is a configurable parameter (2, 3, or 4 bytes).

Another example of a variant branch instruction would be a JMP (A0) instruction. If the CSR was programmed to display the lower two bytes of an address, the outputs of the PST and DDATA signals when this instruction executed are shown in Figure 6-2.

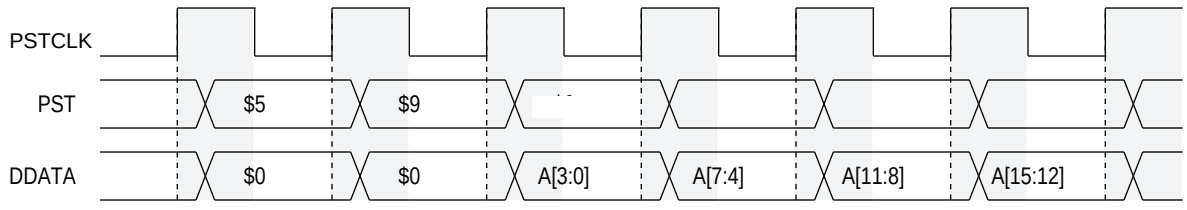


Figure 6-2. Example PST/DDATA Diagram

PST is driven with a $5 indicating a taken branch. In the second cycle, PST is driven with a marker value of $9 indicating a two-byte address that is displayed four bits at a time on the DDATA signals over the next four clock cycles. The remaining four clock cycles display the lower two-bytes of the address (A0), least significant nibble to most significant nibble. The output of the PST signals after the JMP instruction completes is dependent on the target instruction. The PST can continue with the next instruction before the address has completely displayed on the DDATA because of the DDATA FIFO. If the FIFO is full and the next instruction needs to display captured values on DDATA, the pipeline stalls (PST = $0) until space is available in the FIFO.

6.2.1.6 BEGIN EXECUTION OF RTE INSTRUCTION (PST = $7). The unique encoding is generated whenever the return-from-exception (RTE) instruction is executed.

6.2.1.7 BEGIN DATA TRANSFER (PST = $8 - $B). These encodings serve as markers to indicate the number of bytes to be displayed on the DDATA port on subsequent clock cycles. This encoding is driven onto the PST port one processor cycle before the actual data is displayed on DDATA. When PST outputs a $8/$9/$A/$B marker value, the DDATA port outputs 1/2/3/4 bytes of captured data respectively on consecutive processor cycles.

6.2.1.8 EXCEPTION PROCESSING (PST = $C). This encoding is displayed during normal exception processing. Exceptions which enter emulation mode (debug interrupt, or optionally trace) generate a different encoding. Because this encoding defines a multicycle mode, the PST outputs are driven with this value until exception processing is completed.

6.2.1.9 EMULATOR MODE EXCEPTION PROCESSING (PST = $D). This encoding is displayed during emulation mode (debug interrupt, or optionally trace). Because this encoding defines a multicycle mode, the PST outputs are driven with this value until exception processing is completed.

6.2.1.10 PROCESSOR STOPPED (PST = $E). This encoding is generated as a result of the STOP instruction. The ColdFire processor remains in the stopped state until an interrupt occurs. Because this encoding defines a multicycle mode, the PST outputs are driven with this value until the stopped mode is exited.

6.2.1.11 PROCESSOR HALTED (PST = $F). This encoding is generated when the ColdFire processor is halted (see **Section 6.3.1 CPU Halt**.) Because this encoding defines a multicycle mode, the PST outputs are driven with this value until the processor is restarted, or reset.

6.3 BACKGROUND-DEBUG MODE (BDM)

The ColdFire Family supports a modified version of the background debug mode (BDM) functionality found on Motorola's CPU32 family of parts. BDM implements a low-level system debugger in the microprocessor hardware. Communication with the development system is handled via a dedicated, high-speed serial command interface.

Unless noted otherwise, the BDM functionality provided by ColdFire is a proper subset of the CPU32 functionality. The main differences include the following:

- ColdFire implements the BDM controller in a dedicated hardware module. Although some BDM operations do require the CPU to be halted (e.g. CPU register accesses), other BDM commands such as memory accesses can be executed while the processor is running.
- On CPU32 parts, the DSO signal can inform hardware that a serial transfer can start. ColdFire clocking schemes restrict the use of this bit. Because DSO changes only when DSCLK is high, DSO cannot be used to indicate the start of a serial transfer. The development system count the number of clocks in any given transfer.
- The read/write system register commands, RSREG and WSREG, have been replaced by read/write control register commands, RCREG and WCREG. These commands use the register coding scheme from the MOVEC instruction.
- The read/write Debug Module register commands, RDMREG and WDMREG, have been added to support Debug Module register accesses.
- CALL and RST commands are not supported and generates an illegal command response.
- Illegal command responses can be returned using the FILL and DUMP commands, if not immediately preceded by certain, specific BDM commands.
- For any command performing a byte-sized memory read operation, the upper 8 bits of the response data are undefined. The referenced data is returned in the lower 8 bits of the response.
- The Debug Module forces alignment for memory-referencing operations: long accesses are forced to a 0-modulo-4 address; word accesses are forced to a 0-modulo-2 address. An address error response is never returned.

6.3.1 CPU Halt

Although many BDM operations can occur in parallel with CPU operation, unrestricted BDM operation requires the CPU to be halted. A number of sources can cause the CPU to halt, including the following as shown in order of priority:

1. The occurrence of the catastrophic fault-on-fault condition automatically halts the processor.
2. The occurrence of a hardware breakpoint can be configured to generate a pending halt condition in a manner similar to the assertion of the BKPT signal. In all cases, the assertion of this type of halt is first made pending in the processor. Next, the processor samples for pending halt and interrupt conditions once per instruction. Once the pending condition is asserted, the processor halts execution at the next sample point. See **Section 6.4.1 Theory of Operation** for more detail.
3. The execution of the HALT instruction**,** also known as BGND on the 683xx devices, immediately suspends execution. By default this is a supervisor instruction and attempted execution while in user mode generates a privilege violation exception. A User Halt Enable (UHE) control bit is provided in the Configuration/Status Register (CSR) to allow execution of HALT in user mode. The processor may be restarted after the execution of the HALT instruction by serial shifting a “GO” command into the debug module. Execution continues at the instruction following the HALT opcode.

4. The assertion of the BKPT input pin is treated as a pseudo-interrupt, i.e., the halt condition is made pending until the processor core samples for halts/interrupts. The processor samples for these conditions once during the execution of each instruction. If there is a pending halt condition at the sample time, the processor suspends execution and enters the halted state.

There are two special cases involving the assertion of the BKPT pin to be considered.

After the system reset signal is negated, the processor waits for 16 clock cycles before beginning reset exception processing. If the BKPT input pin is asserted within the first eight cycles after $\overline{\text{RSTI}}$ is negated, the processor enters the halt state, signaling that halt status,($F), on the PST outputs. While in this state, all resources accessible via the Debug Module can be referenced. This is the only opportunity to force the ColdFire processor into emulation mode via the EMU bit in the configuration/status register (CSR). Once the system initialization is complete, the processor response to a BDM GO command is dependent on the set of BDM commands performed while breakpointed. Specifically, if the processor's PC register was loaded, then the GO command simply causes the processor to exit the halted state and pass control to the instruction address contained in the PC. Note in this case, the normal reset exception processing is bypassed. Conversely, if the PC register was not loaded, then the GO command causes the processor to exit the halted state and continue with reset exception processing.

ColdFire also handles a special case of the assertion of BKPT 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. Once halted, all BDM commands may be exercised. When the processor is restarted, it continues with the execution of the next sequential instruction, i.e., the instruction following the STOP opcode.

The halt source is indicated in CSR[27:24]. For simultaneous halt conditions, the highest priority source is indicated.

6.3.2 BDM Serial Interface

Once the CPU is halted and the halt status reflected on the PST outputs, the development system may send unrestricted commands to the Debug Module. The Debug Module implements a synchronous protocol using a three-pin interface: development serial clock (DSCLK), development serial input (DSI), and development serial output (DSO). The development system serves as the serial communication channel master and is responsible for generation of the clock (DSCLK). The operating range of the serial channel is DC to 1/5 of the processor frequency. The channel uses a full duplex mode, where data is transmitted 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 seen in Figure 6-3, all state transitions are enabled on a rising edge of the processor clock when DSCLK is high, i.e., DSI is sampled and DSIis driven. The DSCLK signal must also be sampled low (on a positive edge of CLK) between each bit exchange. The MSB is transferred first.

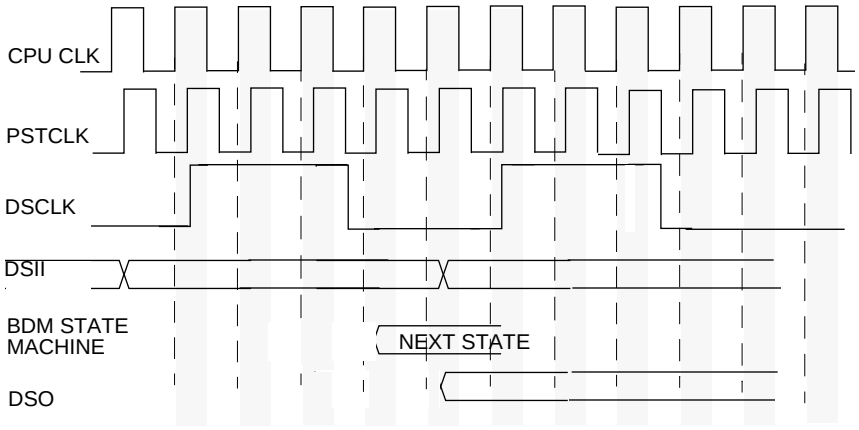


Figure 6-3. BDM Serial Transfer

Both DSCLK and DSI are synchronized inputs.The DSCLK signal essentially acts as a pseudo “clock enable” and is sampled on the rising edge of CLK as well as the DSI. The DSO output is delayed from the DSCLK-enabled CLK rising edge. All events in the Debug Module’s serial state machine are based on the rising edge of the microprocessor clock (see Figure 6-4 below).

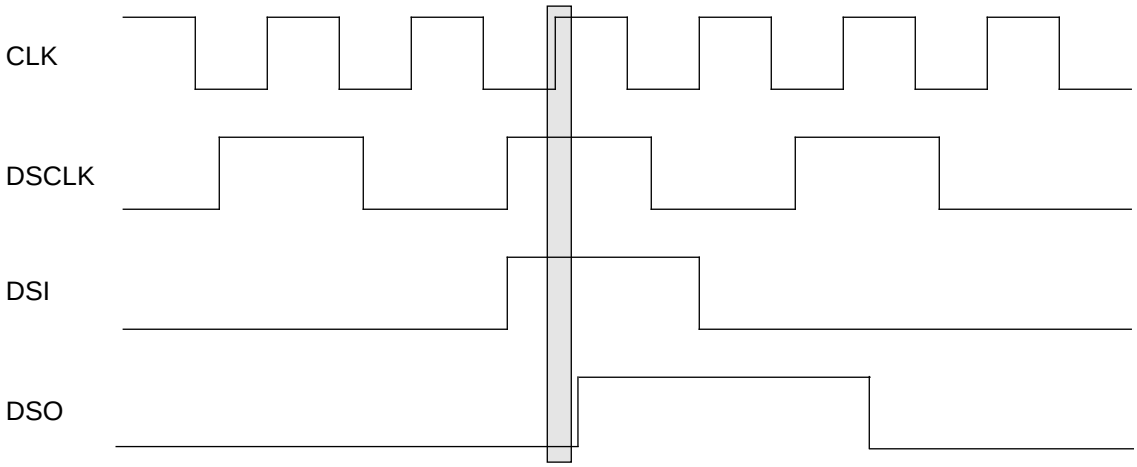


Figure 6-4. BDM Signal Sampling

6.3.2.1 RECEIVE PACKET FORMAT. The basic receive packet of information is 17 bits long,16 data bits plus a status bit, as shown below in Figure 6-5.

16	15	0
S	DATA FIELD [15:0]	

Figure 6-5. Receive BDM Packet

Status[16]

The status bit indicates the status of CPU-generated messages as listed in Table 6-2.

Table 6-2. CPU-Generated Message Encoding

S BIT	DATA	MESSAGE TYPE
0	xxxx	Valid Data Transfer
0	$FFFF	Status Ok
1	$0000	Not Ready with Response; Come Again
1	$0001	Error - Terminated Bus Cycle; Data Invalid
1	$FFFF	Illegal Command

Data Field[15:0]

The data field contains the message data to be communicated from the Debug Module to the development system. The response message is always a single word, with the data field encoded as shown in Table 6-2.

6.3.2.2 TRANSMIT PACKET FORMAT. The basic transmit packet of information is 17 bits long,16 data bits plus a control bit, as shown below in Figure 6-6.

16	15	0
C	DATA FIELD [15:0]	

Figure 6-6. Transmit BDM Packet

Control[16]

The control bit is not used but is reserved by Motorola for future use. Command and data transfers initiated by the development system should clear bit 16.

Data Field[15:0]

The data field contains the message data to be communicated from the development system to the Debug Module.

6.3.3 BDM Command Set

ColdFire supports a subset of BDM instructions from the MC683xx parts, as well as extensions to provide access to new hardware features. The BDM commands must not be issued whenever the ColdFire processor is accessing the Debug Module registers using the WDEBUG instruction, or the resulting behaviour is undefined.

6.3.3.1 BDM COMMAND SET SUMMARY. The BDM command set is summarized in Table 6-3. Subsequent paragraphs contain detailed descriptions of each command.

Table 6-3. BDM Command Summary

COMMAND	MNEMONIC	DESCRIPTION	CPU IMPACT[1]	PAGE
READ A/D REGISTER	RAREG/RDREG	Read the selected address or data register and return the results via the serial interface.	HALTED	6-14
WRITE A/D REGISTER	WAREG/WDREG	The data operand is written to the specified address or data register.	HALTED	6-15
READ MEMORY LOCATION	READ	Read the data at the memory location specified by the longword address.	STEAL	6-16
WRITE MEMORY LOCATION	WRITE	Write the operand data to the memory location specified by the longword address.	STEAL	6-18
DUMP MEMORY BLOCK	DUMP	Used in conjunction with the READ command 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. Subsequent operands are retrieved with the DUMP command.	STEAL	6-20
FILL MEMORY BLOCK	FILL	Used in conjunction with the WRITE command 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. Subsequent operands are written with the FILL command.	STEAL	6-22
RESUME EXECUTION	GO	The pipeline is flushed and refilled before resuming instruction execution at the current PC.	HALTED	6-24
NO OPERATION	NOP	NOP performs no operation and may be used as a null command.	PARALLEL	6-25
OUTPUTS THE CURRENT PC	SYNC_PC	Captures the current PC and displays it on the PST/DDATA output pins.	PARALLEL	6-26
READ CONTROL REGISTER	RCREG	Read the system control register.	HALTED	6-26
WRITE CONTROL REGISTER	WCREG	Write the operand data to the system control register.	HALTED	6-28
READ DEBUG MODULE REGISTER	RDMREG	Read the Debug Module register.	PARALLEL	6-29
WRITE DEBUG MODULE REGISTER	WDMREG	Write the operand data to the Debug Module register.	PARALLEL	6-29

NOTE: 1. General command effect and/or requirements on CPU operation:

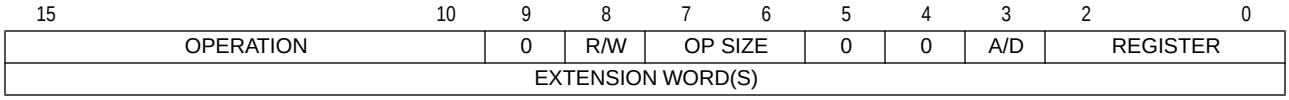
Halted - The CPU must be halted to perform this command

Steal - Command generates bus cycles which can be interleaved with CPU accesses

Parallel - Command is executed in parallel with CPU activity

Refer to command summaries for detailed operation descriptions.

6.3.3.2 COLDFIRE BDM COMMANDS. All ColdFire Family BDM commands include a 16-bit operation word followed by an optional set of one or more extension words.



BDM Command Format

Operation Field

The operation field specifies the command.

R/W Field

The R/W field specifies the direction of operand transfer. When the bit is set, the transfer is from the CPU to the development system. When the bit is cleared, data is written to the CPU or to memory from the development system.

Operand Size

For sized operations, this field specifies the operand data size. All addresses are expressed as 32-bit absolute values. The size field is encoded as listed in Table 6-4.

Table 6-4. BDM Size Field Encoding

ENCODING	OPERAND SIZE	BIT VALUES
00	Byte	8 bits
01	Word	16 bits
10	Longword	32 bits
11	Reserved	

Address / Data (A/D) Field

The A/D field is used in commands that operate on address and data registers in the processor. It determines whether the register field specifies a data or address register. A one indicates an address register; zero, a data register.

Register Field

In commands that operate on processor registers, this field specifies which register is selected. The field value contains the register number.

Extension Word(s) (as required):

Certain commands require extension words for addresses and/or immediate data. Addresses require two extension words because only absolute long addressing is permitted. Immediate data can be either one or two words in length—byte and word data each require a single extension word; longword data requires two words. Both 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”, “Dta” or “Operand Data.”

6.3.3.3 COMMAND SEQUENCE DIAGRAM. A command sequence diagram (see Figure 6-7) illustrates the serial bus traffic for each command. Each bubble in the diagram represents a single 17-bit transfer across the bus. The top half in each bubble corresponds to the data transmitted by the development system to the Debug Module; the bottom half corresponds to the data returned by the Debug Module in response to the previous development system commands. Command and result transactions are overlapped to minimize latency.

The cycle in which the command is issued contains the development system command mnemonic (in this example, “read memory location”). During the same cycle, the Debug Module responds with either the low-order results of the previous command or a command complete status (if no results were required).

During the second cycle, the development system supplies the high-order 16 bits of the memory address. The Debug Module returns a “not ready” response unless the received command was decoded as unimplemented, in which case the response data is the illegal command encoding. If an illegal command response occurs, the development system should retransmit the command.

NOTE

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.

In the third cycle, the development system supplies the low-order 16 bits of a memory address. The Debug Module always returns the “not ready” response in this cycle. At the completion of the third cycle, the Debug Module initiates a memory read operation. Any serial transfers that begin while the memory access is in progress return the “not ready” response.

Results are returned in the two serial transfer cycles following the completion of the memory access. The data transmitted to the Debug Module during the final transfer is the opcode for the following command. If a memory or register access is terminated with a bus error, the error status (S=1, DATA=$0001) is returned in place of the result data.

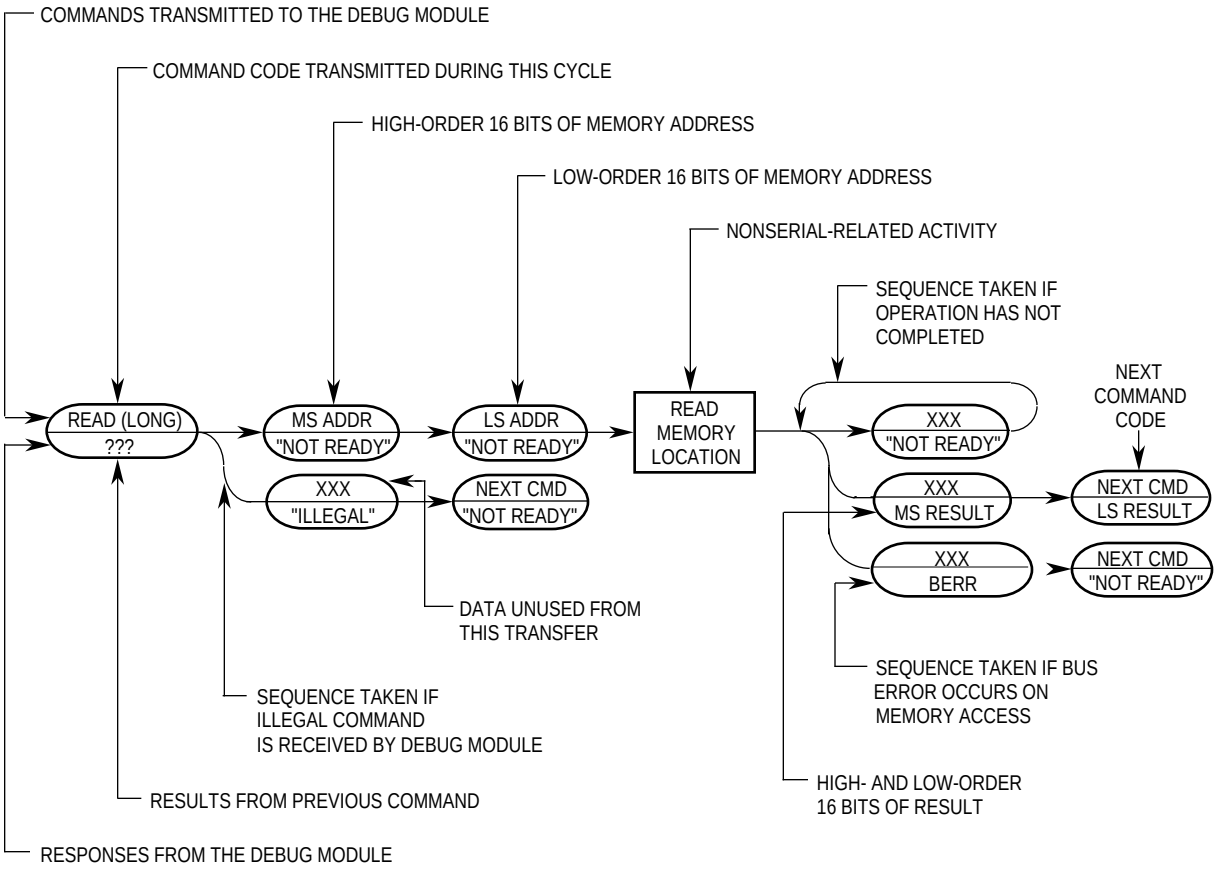


Figure 6-7. Command Sequence Diagram

6.3.3.4 COMMAND SET DESCRIPTIONS. The BDM command set is summarized in Table 6-3. Subsequent paragraphs contain detailed descriptions of each command.

Note

The BDM status bit (S) is zero for normally-completed commands, while illegal commands, “not ready” responses and bus-errored transfers return a logic one in the S bit. Refer to **Section 6.3.2 BDM Serial Interface** for information on the serial packet receive packet format

Unassigned command opcodes are reserved by Motorola for future expansion. All unused command formats within any revision level performs a NOP and return the ILLEGAL command response.

6.3.3.4.1 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.

Formats:

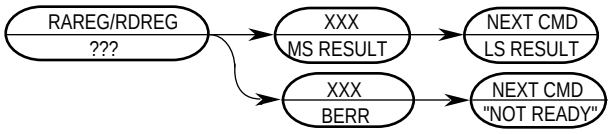
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$2				$1				$8				A/D	REGISTER		

RAREG/RDREG Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA [31:16]															
DATA [15:0]															

RAREG/RDREG Result

Command Sequence:



Operand Data:
None

Result Data:
The contents of the selected register are returned as a longword value. The data is returned most significant word first.

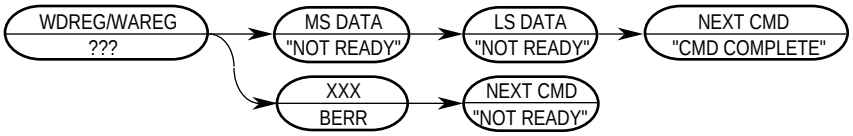
6.3.3.4.2 Write A/D Register (WAREG/WDREG). The operand longword data is written to the specified address or data register. All 32 register bits are altered by the write. A bus error response is returned if the CPU core is not halted.

Command Formats:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$2				$0				$8				A/D	REGISTER		
DATA [31:16]															
DATA [15:0]															

WAREG/WDREG Command

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 the data $FFFF (with the status bit cleared) when the register write is complete.

6.3.3.4.3 Read Memory Location (READ). Read the operand data from the memory location specified by the longword address. The address space is defined by the contents of the low-order 5 bits {TT, TM} of the BDM Address Attribute Register (BAAR). The hardware forces the low-order bits of the address to zeros for word and longword accesses to ensure that operands are always accessed on natural boundaries: words on 0-modulo-2 addresses, longwords on 0-modulo-4 addresses.

Formats:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$9				$0				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															

Byte READ Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	DATA [7:0]							

Byte READ Result

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$9				$4				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															

Word READ Command

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

Word READ Result

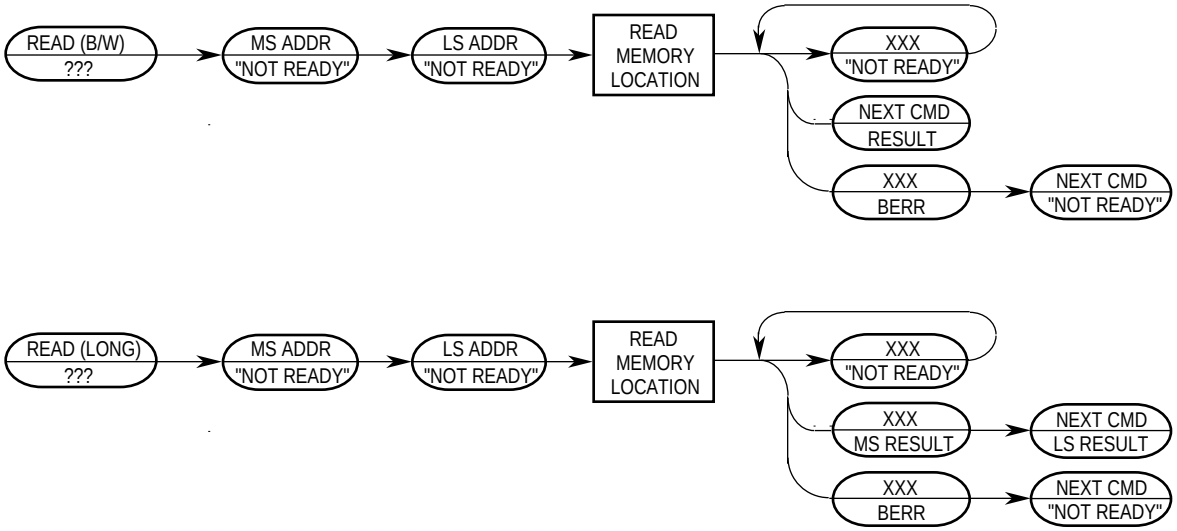
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$9				$8				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															

Long READ Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA [31:16]															
DATA [15:0]															

Long READ Result

Command Sequence:



Operand Data:

The single operand is the longword address of the requested memory location.

Result Data:

The requested data is returned as either a word or longword. Byte data is returned in the least significant byte of a word result, with the upper byte undefined. Word results return 16 bits of significant data; longword results return 32 bits. A value of $0001 (with the status bit set) is returned if a bus error occurs.

6.3.3.4.4 Write Memory Location (WRITE). Write the operand data to the memory location specified by the longword address. The address space is defined by the contents of the low-order 5 bits {TT, TM} of the BDM Address Attribute Register (BAAR). The hardware forces the low-order bits of the address to zeros for word and longword accesses to ensure that operands are always accessed on natural boundaries: words on 0-modulo-2 addresses, longwords on 0-modulo-4 addresses.

Formats:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$8				$0				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															
X	X	X	X	X	X	X	X	DATA [7:0]							

Byte WRITE Command

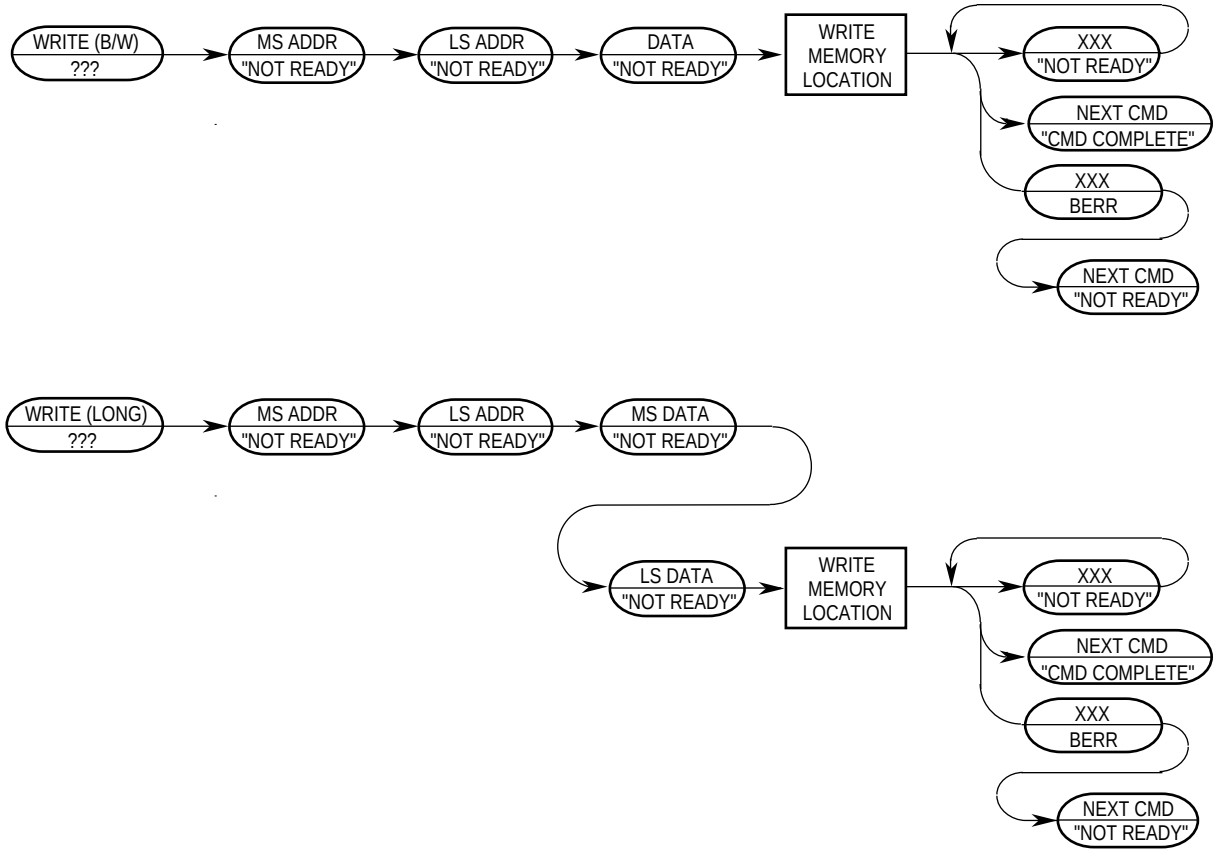
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$8				$4				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															
DATA [15:0]															

Word WRITE Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$8				$8				$0			
ADDRESS [31:16]															
ADDRESS [15:0]															
DATA [31:16]															
DATA [15:0]															

Long WRITE Command

Command Sequence:



Operand Data:

Two operands are required for this instruction. The first operand is a longword absolute address that specifies a location to which the operand data is to be written. The second operand is the data. Byte data is transmitted as a 16-bit word, justified in the least significant byte; 16- and 32-bit operands are transmitted as 16 and 32 bits, respectively.

Result Data:

Command complete status is indicated by returning the data $FFFF (with the status bit cleared) when the register write is complete. A value of $0001 (with the status bit set) is returned if a bus error occurs.

6.3.3.4.5 Dump Memory Block (DUMP). DUMP is used in conjunction 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. 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

The DUMP command does not check for a valid address — DUMP is a valid command only when preceded by another DUMP, NOP or by a READ 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 DUMP command is processed, allowing the operand size to be dynamically altered.

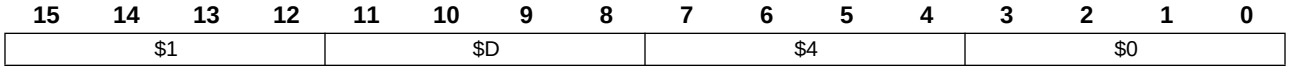
Command Formats:

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

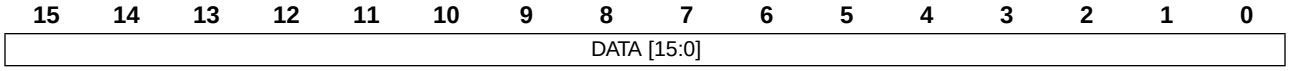
Byte DUMP Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	DATA [7:0]							

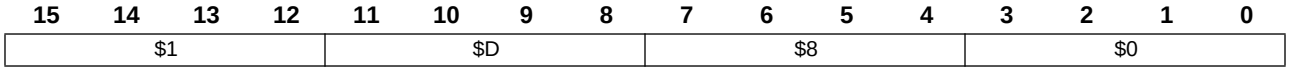
Byte DUMP Result



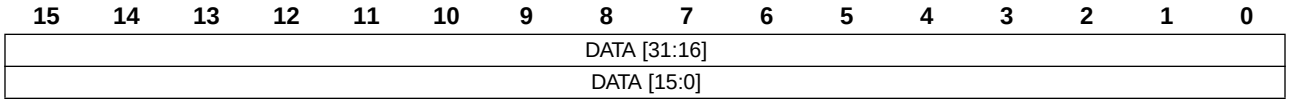
Word DUMP Command



Word DUMP Result

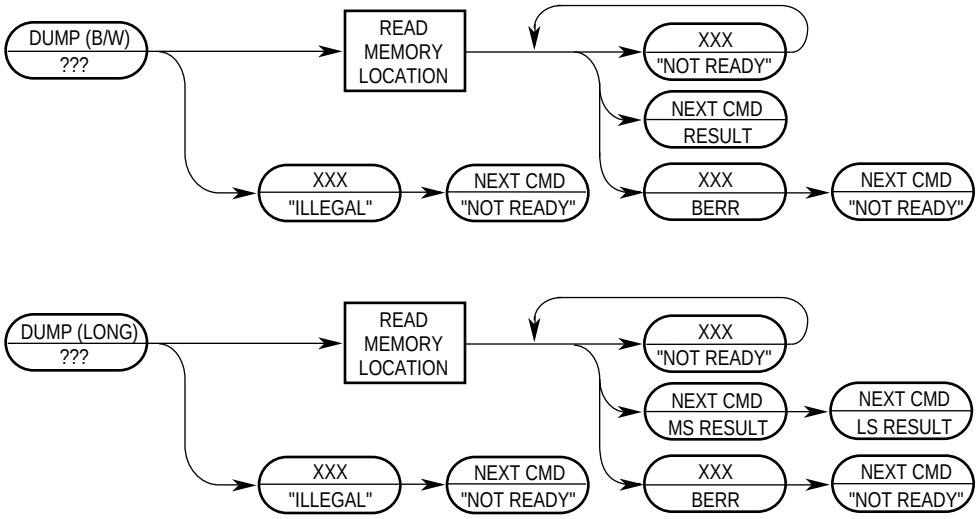


Long DUMP Command



Long DUMP Result

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 $0001 (with the status bit set) is returned if a bus error occurs.

6.3.3.4.6 Fill Memory Block (FILL). FILL is used in conjunction 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.

NOTE

The FILL command does not check for a valid address —FILL is a valid command only when preceded by another FILL, NOP or by 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.

Formats:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$C				$0				$0			
X	X	X	X	X	X	X	X	DATA [7:0]							

Byte FILL Command

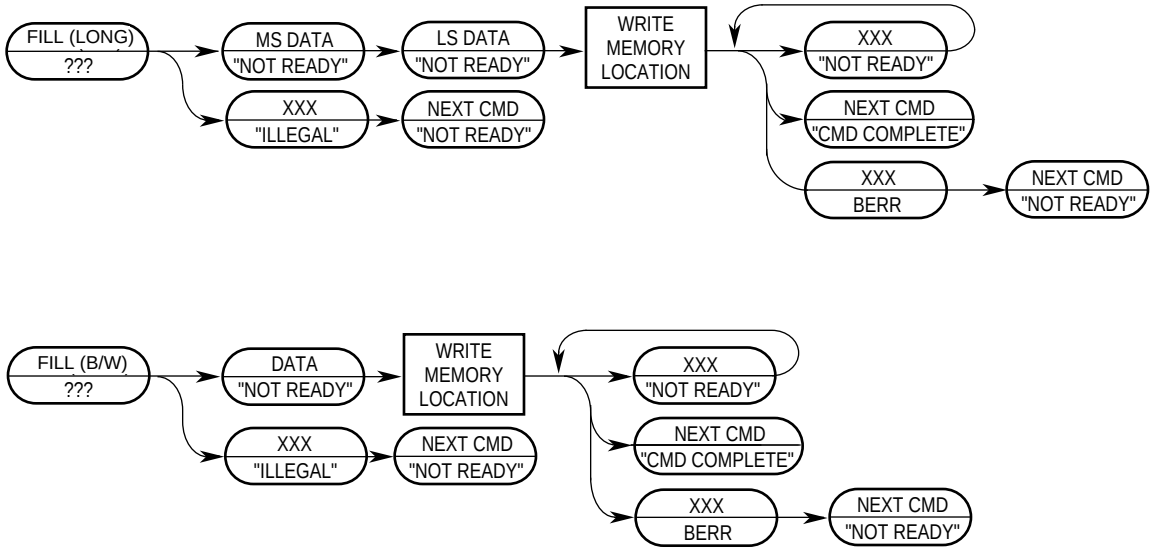
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$C				$4				$0			
DATA [15:0]															

Word FILL Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$1				$C				$8				$0			
DATA [31:16]															
DATA [15:0]															

Long FILL Command

Command Sequence:



Operand Data:

A single operand is data to be written to the memory location. Byte data is transmitted as a 16-bit word, justified in the least significant byte; 16- and 32-bit operands are transmitted as 16 and 32 bits, respectively.

Result Data:

Command complete status is indicated by returning the data $FFFF (with the status bit cleared) when the register write is complete. A value of $0001 (with the status bit set) is returned if a bus error occurs.

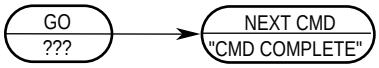
6.3.3.4.7 Resume Execution (GO). The pipeline is flushed and refilled before resuming normal instruction execution. Prefetching begins at the current PC and current privilege level. If any register (e.g., the PC or SR) was altered by a BDM command while halted, the updated value is used as the prefetching resumes.

Formats:

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

GO Command

Command Sequence:



Operand Data:
None

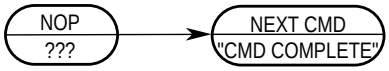
Result Data:
The “command complete” response ($0FFFF) is returned during the next shift operation.

6.3.3.4.8 No Operation (NOP). NOP performs no operation and may be used as a null command where required.

Formats:

15	12	11	8	7	4	3	0
$0		$0		$0		$0	

Command Sequence:



Operand Data:
None

Result Data:
The “command complete” response, $FFFF (with the status bit cleared), is returned during the next shift operation.

6.3.3.4.9 Synchronize PC to the PST/DDATA Lines(SYNC_PC). Capture the current PC and display it on the PST/DDATA 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 the BTB bits of the CSR (CSR [9:8]). The specific sequence of PST and DDATA values is defined below :

Debug signals a SYNC_PC command is pending. CPU completes the current instruction. CPU forces an instruction fetch to the next PC, generates a PST = $5 value indicating a “taken branch” and signals DDATA. DDATA captures the instruction address corresponding to the PC. DDATA generates a PST marker ($9 - $B) as defined by CSR.BTB and displays the captured PC address.

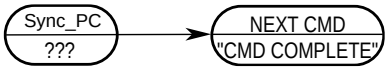
This command can beused to dynamically access the PC for performance monitoring. The execution of this command is considerably less obtrusive to the real-time operation of an application than a “halt-CPU/read-PC/resume” command sequence.

Format:

15 12	11 8	7 4	3 0
$0	$0	$0	$1

SYNC_PC REG Command

Command Sequence:



Operand Data:

None

Result Data:

The "command complete" response, $FFFF (with the status bit cleared), is returned during the next shift operation.

6.3.3.4.10 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 in size, regardless of the implemented register width. The second and third words of the command effectively form a 32-bit address used by the Debug Module 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.

Formats:

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

RCREG Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA [31:16]															
DATA [15:0]															

RCREG Result

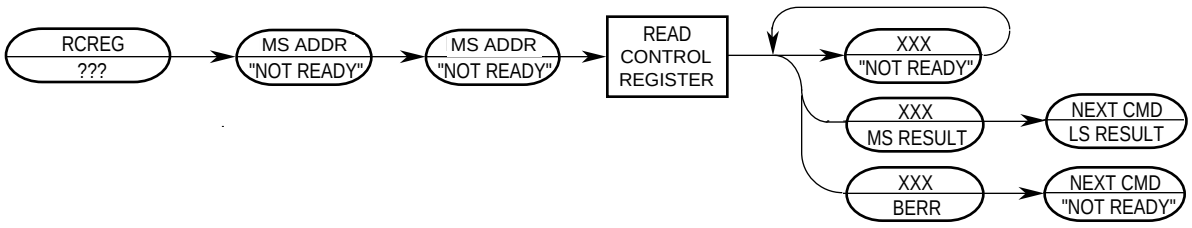
Rc encoding:

Table 6-5. Control Register Map

Rc	REGISTER DEFINITION
$002	Cache Control Register (CACR)
$004	Access Control Register 0 (ACR0)
$005	Access Control Register 1 (ACR1)
$801	Vector Base Register (VBR)
$804	MAC Status Register (MACSR)†
$805	MAC Mask Register (MASK)†
$806	MAC Accumulator (ACC)†
$80E	Status Register (SR)
$80F	Program Register (PC)
$C00	ROM Base Address Register (ROMBAR)
$C04	RAM Base Address Register (RAMBAR)

NOTE: †Available if the optional MAC unit is present.

Command Sequence:



Operand Data:

The single operand is the 32-bit Rc control register select field.

Result Data:

The contents of the selected control register are returned as a longword value. The data is returned most significant word first. For those control registers with widths less than 32 bits, only the implemented portion of the register is guaranteed to be correct. The remaining bits of the longword are undefined.

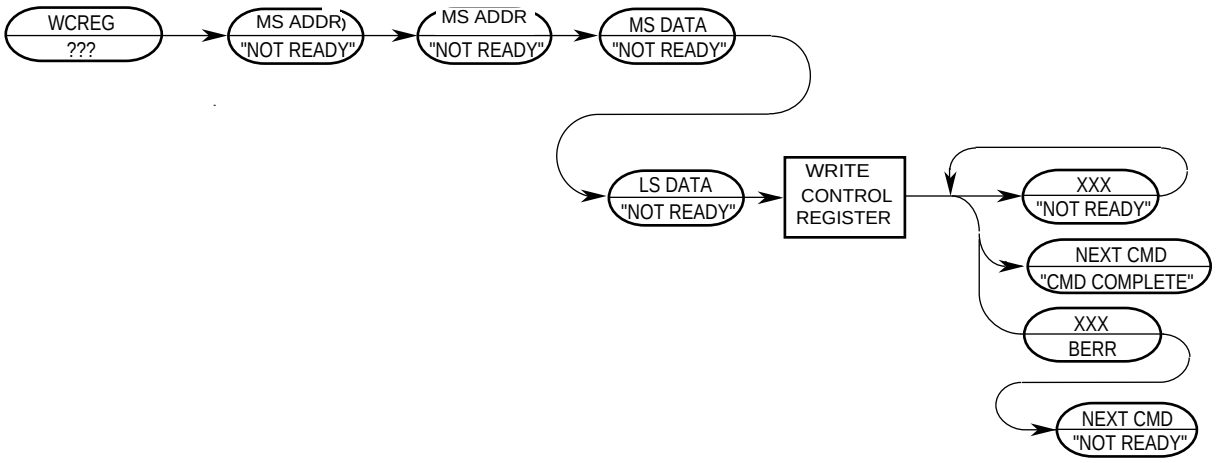
6.3.3.4.11 Write Control Register (WCREG). The operand (longword) data is written to the specified control register. The write alters all 32 register bits.

Formats:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$2				$8				$8				$0			
$0				$0				$0				$0			
$0				Rc											
DATA [31:16]															
DATA [15:0]															

WCREG Command

Command Sequence:



Operand Data:

Two operands are required for this instruction. The first long operand selects the register to which the operand data is to be written. The second operand is the data.

Result Data:

Successful write operations return a $FFFF. Bus errors on the write cycle are indicated by the assertion of bit 16 in the status message and by a data pattern of $0001.

6.3.3.4.12 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 the CSR (DRc = $0).

Command Formats:

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

RDMREG BDM Command

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA [31:16]															
DATA [15:0]															

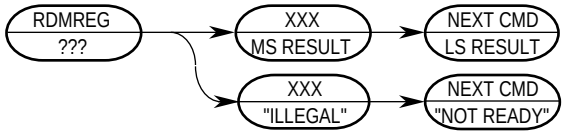
RDMREG BDM Result

DRc encoding:

Table 6-6. Definition of DRc Encoding - Read

DRc[3:0]	DEBUG REGISTER DEFINITION	MNEMONIC	INITIAL STATE
$0	Configuration/Status	CSR	$0
$1-$F	Reserved	-	–

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.

6.3.3.4.13 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. The DSCLK signal must be inactive while debug module register writes from the CPU accesses are performed using the WDEBUG instruction.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$2				$C				$8				DRc			
DATA [31:16]															
DATA [15:0]															

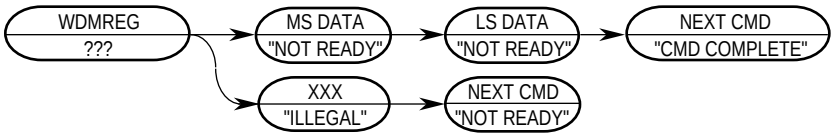
WDMREG BDM Command

DRc encoding:

Table 6-7. Definition of DRc Encoding - Write

DRc[3:0]	DEBUG REGISTER DEFINITION	MNEMONIC	INITIAL STATE
$0	Configuration/Status	CSR	$0
$1-$4	Reserved	-	–
$5	BDM Address Attribute	BAAR	$5
$6	Bus Attributes and Mask	AATR	$5
$7	Trigger Definition	TDR	$0
$8	PC Breakpoint	PBR	–
$9	PC Breakpoint Mask	PBMR	–
$A-$B	Reserved	–	–
$C	Operand Address High Breakpoint	ABHR	–
$D	Operand Address Low Breakpoint	ABLR	–
$E	Data Breakpoint	DBR	–
$F	Data Breakpoint Mask	DBMR	–

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 ($0FFFF) is returned when register write is complete.

6.3.3.4.14 Unassigned Opcodes. Unassigned command opcodes are reserved by Motorola. All unused command formats within any revision level performs a NOP and return the ILLEGAL command response**.**

6.4 REAL-TIME DEBUG SUPPORT

The ColdFire Family provides support for the debug of real-time applications. For these types of embedded systems, the processor cannot be halted during debug, but must continue to operate. The foundation of this area of debug support is that while the processor cannot be halted to allow debugging, the system can generally tolerate small intrusions into the real-time operation.

The Debug Module provides a number of hardware resources to support various hardware breakpoint functions. Specifically, three types of breakpoints are supported: PC with mask, operand address range, and data with mask. These three basic breakpoints can be configured into one- or two-level triggers with the exact trigger response also programmable. The Debug Module programming model is accessible from either the external development system using the serial interface or from the processor's supervisor programming model using the WDEBUG instruction.

6.4.1 Theory of Operation

The breakpoint hardware can be configured to respond to triggers in several ways. The desired response is programmed into the Trigger Definition Register. In all situations where a breakpoint triggers, an indication is provided on the DDATA output port, when not displaying captured operands or branch addresses, as shown in Table 6-8.

Table 6-8. DDATA[3:0], CSR[31:28] Breakpoint Response

DDATA[3:0], CSR[31:28]	BREAKPOINT STATUS
$000x	No Breakpoints Enabled
$001x	Waiting for Level 1 Breakpoint
$010x	Level 1 Breakpoint Triggered
$101x	Waiting for Level 2 Breakpoint
$110x	Level 2 Breakpoint Triggered
All other encodings are reserved for future use.	

The breakpoint status is also posted in the CSR.

The BDM instructions load and configure the desired breakpoints using the appropriate registers. As the system operates, a breakpoint trigger generates a response as defined in the TDR. If the system can tolerate the processor being halted, a BDM-entry can be used. With the TRC bits of the TDR equal to $1, the breakpoint trigger causes the core to halt as reflected in the PST = $F status. For PC breakpoints, the halt occurs before the targeted instruction is executed. For address and data breakpoints, the processor may have executed several additional instructions. As a result, trigger reporting is considered imprecise.

If the processor core cannot be halted, the special debug interrupt can be used. With this configuration, TRC bits of the TDR equal to $2, the breakpoint trigger is converted into a debug interrupt to the processor. This interrupt 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 immediately (before the execution of the targeted instruction). This is possible because the PC breakpoint comparison is enabled at the same time the interrupt sampling occurs. For the address and data breakpoints, the reporting is considered imprecise because several additional instructions may be executed after the triggering address or data is seen.

Once the debug interrupt is recognized, the processor aborts execution and initiates exception processing. At the initiation of the exception processing, the core enters emulator mode. After the standard 8-byte exception stack is created, the processor fetches a unique exception vector, 12, from the vector table (Refer to the *ColdFire Programmer's Reference Manual Rev 1.0* MCF5200PRM/AD).

Execution continues at the instruction address contained in this exception vector. All interrupts are ignored while in emulator mode. You can program the debug-interrupt handler to perform the necessary context saves using the supervisor instruction set. As an example, this handler may save the state of all the program-visible registers as well as the current context into a reserved memory area.

Once the required operations are completed, the return-from-exception (RTE) instruction is executed and the processor exits emulator mode. Once the debug interrupt handler has completed its execution, the external development system can then access the reserved memory locations using the BDM commands to read memory.

In the Rev. A implementation, if a hardware breakpoint (e.g., a PC trigger) is left unmodified by the debug interrupt service routine, another debug interrupt is generated after the RTE instruction completes execution. In the Rev. B design, the hardware has been modified to inhibit the generation of another debug interrupt during the first instruction after the RTE exits emulator mode. This behaviour is consistent with the existing logic involving trace mode, where the execution of the first instruction occurs before another trace exception is generated. This Rev. B enhancement disables all hardware breakpoints until the first instruction after the RTE has completed execution, regardless of the programmed trigger response.

6.4.1.1 EMULATOR MODE. Emulator mode is used to facilitate non-intrusive emulator functionality. This mode can be entered in three different ways:

- The EMU bit in the CSR may be programmed to force the ColdFire processor to begin execution in emulator mode. This bit is only examined when $\overline{\text{RSTI}}$ is negated and the processor begins reset exception processing. It may be set while the processor is halted before the reset exception processing begins. Refer to **Section 6.3.1 CPU Halt**.
- A debug interrupt always enters emulation mode when the debug interrupt exception processing begins.
- The TCR bit in the CSR may be programmed to force the processor into emulation mode when trace exception processing begins.

During emulation mode, the ColdFire processor exhibits the following properties:

- All interrupts are ignored, including level seven.

- If the MAP bit of the CSR is set, all memory accesses are forced into a specially mapped address space signalled by TT = $2, TM = $5 or $6. This includes the stack frame writes and the vector fetch for the exception which forced entry into this mode.
- If the MAP bit in the CSR is set, all caching of memory accesses is disabled. Additionally, the SRAM module is disabled while in this mode.

The return-from-exception (RTE) instruction exits emulation mode. The processor status output port provides a unique encoding for emulator mode entry ($D) and exit ($7).

6.4.1.2 DEBUG MODULE HARDWARE.

6.4.1.2.1 Reuse of Debug Module Hardware (Rev. A). The Debug Module implementation provides a common hardware structure for both BDM and breakpoint functionality. Several structures are used for both BDM and breakpoint purposes. Table 6-9 identifies the shared hardware structures.

Table 6-9. 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

The shared use of these hardware structures means the loading of the register to perform any specified function is destructive to the shared function. For example, if an operand address breakpoint is loaded into the Debug Module, a BDM command to access memory overwrites the breakpoint. If a data breakpoint is configured, a BDM write command overwrites the breakpoint contents.

6.4.1.2.2 The New Debug Module Hardware (Rev. B). The new Debug Module implementation has added hardware registers so that there are no restrictions concerning the interaction between BDM commands and the use of the hardware breakpoint logic. In some cases, the additional hardware is not program-visible, while in other cases, there have been extensions to the Debug Module programming model. As example, consider the following two registers:

The hardware register containing the BDM memory address is not a program-visible resource. Rather, it is a hardware register loaded automatically during the execution of a BDM commands. In the Rev B design, the execution of a BDM command does not affect the hardware breakpoint logic unless those registers are specifically accessed.

The other register added to the Debug Module programming model is the BDM Address Attribute Register (BAAR). It is mapped to an DRc[3:0] address of $5. This 8-bit register is equivalent in the format of the low-order byte of the AATR register (Refer to section15.4.2.7). This register specifies the memory space attributes associated with all BDM memory-referencing commands.

6.4.2 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 nine registers to support the required functionality. All of these registers are treated as 32-bit quantities, regardless of the actual number of bits in the implementation. The registers, known as the debug control registers, are accessed through the BDM port using two new BDM commands: WDMREG and RDMREG. These commands contain a 4-bit field, DRc, which specifies the particular register being accessed.

These registers are also accessible from the processor's supervisor programming model through the execution of the WDEBUG instruction. Thus, the breakpoint hardware within the Debug Module may be accessed by the external development system using the serial interface, or by the operating system running on the processor core. 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). The BDM commands must not be issued if the ColdFire processor is accessing the Debug Module registers using the WDEBUG instruction.

Figure 6-8 illustrates the Debug Module programming model.

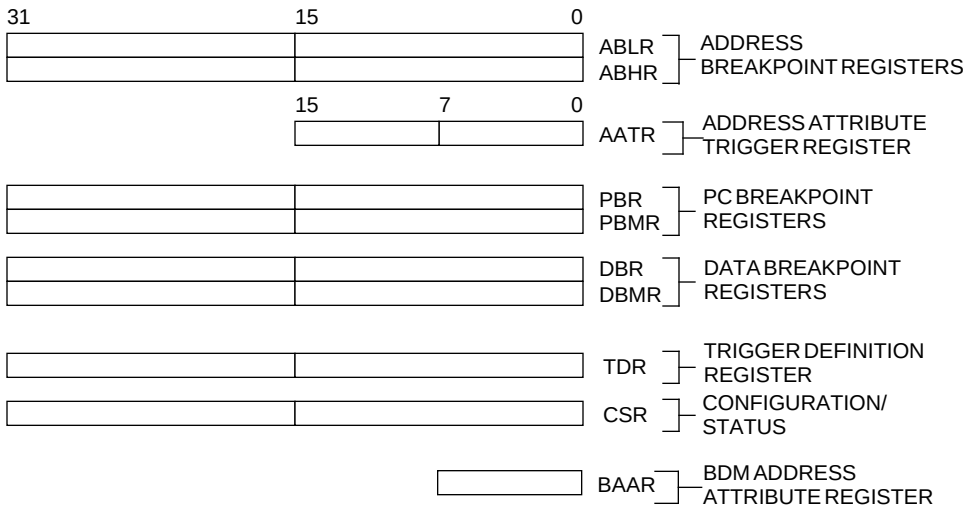


Figure 6-8. Debug Programming Model

6.4.2.1 ADDRESS BREAKPOINT REGISTERS (ABLR, ABHR). The address breakpoint registers define a region in the operand address space of the processor that can be used as part of the trigger. The full 32-bits of the ABLR and ABHR values are compared with the address for all transfers on the processor's high-speed local bus. The trigger definition register (TDR) determines if the trigger is the inclusive range bound by ABLR and ABHR, all addresses outside this range, or the address in ABLR only. The ABHR is accessible in supervisor mode as debug control register $C using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands. The ABLR is accessible in

supervisor mode as debug control register $D using the WDEBUG instruction and via the BDM port using the WDMREG commands. The ABHR is overwritten by the BDM hardware when accessing memory as described in **Section 6.4.1.2 Debug Module Hardware**.

BITS	31															0
FIELD	ADDRESS															
RESET	-															
R/W	W															

Address Breakpoint Low Register (ABLR)

Field Definition:

ADDRESS[31:0]–Low Address
This field contains the 32-bit address which marks the lower bound of the address breakpoint range. Additionally, if a breakpoint on a specific adrdess is required, the value is programmed into the ABLR.

BITS	31															0
FIELD	ADDRESS															
RESET	-															
R/W	W															

Address Breakpoint High Register (ABHR)

Field Definition:

ADDRESS[31:0]–High Address
This field contains the 32-bit address which marks the upper bound of the address breakpoint range.

6.4.2.2 ADDRESS ATTRIBUTE TRIGGER REGISTER (AATR). The AATR defines the address attributes and a mask to be matched in the trigger. The AATR value is compared with the address attribute signals from the processor's local high-speed bus, as defined by the setting of the TDR. The AATR is accessible in supervisor mode as debug control register $6 using the WDEBUG instruction and via the BDM port using the WDMREG command. The lower five bits of the AATR are also used for BDM command definition to define the address space for memory references as described in **Section 6.4.1.2 Debug Module Hardware**.

BITS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	RM	SZM		TTM		TMM			R	SZ		TT		TM		
RESET	0	0		0		0			0	0		0		101		
R/W	W	W		W		W			W	W		W		W		

Address Attribute Trigger Register (AATR)

Field Definitions:

RM[15]–Read/Write Mask

This field corresponds to the R-field. Setting this bit causes R to be ignored in address comparisons.

SZM[14:13]–Size Mask

This field corresponds to the SZ field. Setting a bit in this field causes the corresponding bit in SZ to be ignored in address comparisons.

TTM[12:11]–Transfer Type Mask

This field corresponds to the TT field. Setting a bit in this field causes the corresponding bit in TT to be ignored in address comparisons.

TMM[10:8]–Transfer Modifier Mask

This field corresponds to the TM field. Setting a bit in this field causes the corresponding bit in TM to be ignored in address comparisons.

R[7]–Read/Write

This field is compared with the R/W signal of the processor’s local bus.

SZ[6:5]—Size

This field is compared to the size signals of the processor’s local bus. These signals indicate the data size for the bus transfer.

00 = Longword
01 = Byte
10 = Word
11 = Reserved

TT[4:3]—Transfer Type

This field is compared with the transfer type signals of the processor's local bus. These signals indicate the transfer type for the bus transfer. These signals are always encoded as if the ColdFire is in the ColdFire IACK mode.

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 generates an alternate master access (For backward compatibility).

TM[2:0]—Transfer Modifier

This field is compared with the transfer modifier signals of the processor's local bus. These signals provide supplemental information for each transfer type. These signals are always encoded as if the processor is operating in the ColdFire IACK mode. The encoding for normal processor transfers (TT = 0) is:

000 = Explicit Cache Line Push
001 = User Data Access
010 = User Code Access
011 = Reserved
100 = Reserved
101 = Supervisor Data Access
110 = Supervisor Code Access
111 = Reserved

The encoding for emulator mode transfers (TT = 10) is:

0xx = Reserved
100 = Reserved
101 = Emulator Mode Data Access
110 = Emulator Mode Code Access
111 = Reserved

The encoding for acknowledge/CPU space transfers (TT = 11) is:

000 = CPU Space Access
001 = Interrupt Acknowledge Level 1
010 = Interrupt Acknowledge Level 2
011 = Interrupt Acknowledge Level 3
100 = Interrupt Acknowledge Level 4
101 = Interrupt Acknowledge Level 5
110 = Interrupt Acknowledge Level 6
111 = Interrupt Acknowledge Level 7

These bits also define the TM encoding for BDM memory commands (For backward compatibility).

6.4.2.3 PROGRAM COUNTER BREAKPOINT REGISTER (PBR, PBMR). The PC breakpoint registers define a region in the code address space of the processor that can be used as part of the trigger. The PBR value is masked by the PBMR value, allowing only those bits in PBR that have a corresponding zero in PBMR to be compared with the processor's program counter register, as defined in the TDR. The PBR is accessible in supervisor mode as debug control register $8 using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands. The PBMR is accessible in supervisor mode as debug control register $9 using the WDEBUG instruction and via the BDM port using the WDMREG command.

BITS	31															0
FIELD	ADDRESS															
RESET	-															
R/W	W															

Program Counter Breakpoint Register (PBR)

Field Definition:

ADDRESS[31:0]–PC Breakpoint Address

This field contains the 32-bit address to be compared with the PC as a breakpoint trigger.

BITS	31															0
FIELD	MASK															
RESET	-															
R/W	W															

Program Counter Breakpoint Mask Register (PBMR)

Field Definition:

MASK[31:0]–PC Breakpoint Mask

This field contains the 32-bit mask for the PC breakpoint trigger. A zero in a bit position causes the corresponding bit in the PBR to be compared to the appropriate bit of the PC. A one causes that bit to be ignored.

6.4.2.4 DATA BREAKPOINT REGISTER (DBR, DBMR). The data breakpoint registers define a specific data pattern that can be used as part of the trigger into debug mode.The DBR value is masked by the DBMR value, allowing only those bits in DBR that have a corresponding zero in DBMR to be compared with the data value from the processor's local

bus, as defined in the TDR. The DBR is accessible in supervisor mode as debug control register $E using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands. The DBMR is accessible in supervisor mode as debug control register $F using the WDEBUG instruction and via the BDM port using the WDMREG command. The DBR is overwritten by the BDM hardware when accessing memory as described in **Section 6.4.1.2 Debug Module Hardware**.

BITS	31															0
FIELD	ADDRESS															
RESET																
R/W	W															

Data Breakpoint Register (DBR)

Field Definition:

DATA[31:0]–Data Breakpoint Value

This field contains the 32-bit value to be compared with the data value from the processor's local bus as a breakpoint trigger.

BITS	31															0
FIELD	MASK															
RESET	-															
R/W	W															

Data Breakpoint Mask Register (DBMR)

Field Definition:

MASK[31:0]–Data Breakpoint Mask

This field contains the 32-bit mask for the data breakpoint trigger. A zero in a bit position causes the corresponding bit in the DBR to be compared to the appropriate bit of the internal data bus. A one causes that bit to be ignored.

The data breakpoint register supports both aligned and misaligned references. The relationship between the processor address, the access size, and the corresponding location within the 32-bit data bus is shown in Table 6-10.

Table 6-10. Access Size and Operand Data Location

ADDRESS[1:0]	ACCESS SIZE	OPERAND LOCATION
00	Byte	Data[31:24]
01	Byte	Data[23:16]
10	Byte	Data[15:8]
11	Byte	Data[7:0]
0x	Word	Data[31:16]
1x	Word	Data[15:0]
xx	Long	Data[31:0]

6.4.2.5 TRIGGER DEFINITION REGISTER (TDR). The TDR configures the operation of the hardware breakpoint logic within the Debug Module and controls the actions taken under the defined conditions. The breakpoint logic may be configured as a one- or two-level trigger, where bits [31:16] of the TDR define the 2nd level trigger and bits [15:0] define the first level trigger. The TDR is accessible in supervisor mode as debug control register $7 using the WDEBUG instruction and via the BDM port using the WDMREG command.

BITS	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	TRC		EBL	EDLW	EDWL	EDWU	EDLL	EDLM	EDUM	EDUU	DI	EAI	EAR	EAL	EPC	PCI
RESET	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	W		W	W	W	W	W	W	W	W	W	W	W	W	W	W
BITS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	LXT		EBL	EDLW	EDWL	EDWU	EDLL	EDLM	EDUM	EDUU	DI	EAI	EAR	EAL	EPC	PCI
RESET	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	W		W	W	W	W	W	W	W	W	W	W	W	W	W	W

Trigger Definition Register (TDR)

Field Definitions:

TRC–Trigger Response Control

The trigger response control determines how the processor is to respond to a completed trigger condition. The trigger response is always displayed on the DDATA pins.

00 = display on DDATA only
01 = processor halt
10 = debug interrupt
11 = reserved

LxT–Level-x Trigger

This is a Rev. B function. The Level-x Trigger bit determines the logic operation for the trigger between the PC_condition and the (Address_range & Data_condition) where the

inclusion of a Data condition is optional. The ColdFire debug architecture supports the creation of single or double-level triggers.

TDR[15]
- 0 Level-2 trigger = PC_condition & Address_range & Data_condition
- 1 Level-2 trigger = PC_condition | (Address_range & Data_condition)

TDR[14]
- 0 Level-1 trigger = PC_condition & Address_range & Data_condition
- 1 Level-1 trigger = PC_condition | (Address_range & Data_condition)

EBL–Enable Breakpoint Level

If set, this bit serves as the global enable for the breakpoint trigger. If cleared, all breakpoints are disabled.

EDLW–Enable Data Breakpoint for the Data Longword

If set, this bit enables the data breakpoint based on the entire processor's local data bus. The assertion of any of the ED bits enables the data breakpoint. If all bits are cleared, the data breakpoint is disabled.

EDWL–Enable Data Breakpoint for the Lower Data Word

If set, this bit enables the data breakpoint based on the low-order word of the processor's local data bus.

EDWU–Enable Data Breakpoint for the Upper Data Word

If set, this bit enables the data breakpoint trigger based on the high-order word of the processor's local data bus.

EDLL–Enable Data Breakpoint for the Lower Lower Data Byte

If set, this bit enables the data breakpoint trigger based on the low-order byte of the low-order word of the processor's local data bus.

EDLM–Enable Data Breakpoint for the Lower Middle Data Byte

If set, this bit enables the data breakpoint trigger based on the high-order byte of the low-order word of the processor's local data bus.

EDUM–Enable Data Breakpoint for the Upper Middle Data Byte

If set, this bit enables the data breakpoint trigger on the low-order byte of the high-order word of the processor's local data bus.

EDUU–Enable Data Breakpoint for the Upper Upper Data Byte

If set, this bit enables the data breakpoint trigger on the high-order byte of the high-order word of the processor's local data bus.

DI–Data Breakpoint Invert

This bit provides a mechanism to invert the logical sense of all the data breakpoint comparators. This can develop a trigger based on the occurrence of a data value not equal to the one programmed into the DBR.

EAI–Enable Address Breakpoint Inverted

If set, this bit enables the address breakpoint based outside the range defined by ABLR and ABHR. The assertion of any of the EA bits enables the address breakpoint. If all three bits are cleared, this breakpoint is disabled.

EAR–Enable Address Breakpoint Range

If set, this bit enables the address breakpoint based on the inclusive range defined by ABLR and ABHR.

EAL–Enable Address Breakpoint Low

If set, this bit enables the address breakpoint based on the address contained in the ABLR.

EPC–Enable PC Breakpoint

If set, this bit enables the PC breakpoint.

PCI–PC Breakpoint Invert

If set, this bit allows execution outside a given region as defined by PBR and PBMR to enable a trigger.If cleared, the PC breakpoint is defined within the region defined by PBR and PBMR.

6.4.2.6 CONFIGURATION/STATUS REGISTER (CSR). The CSR defines the debug configuration for the processor and memory subsystem. In addition to defining the microprocessor configuration, this register also contains status information from the breakpoint logic. The CSR is cleared during system reset. The CSR can be read and written by the external development system and written by the supervisor programming model. The CSR is accessible in supervisor mode as debug control register $0 using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands.

BITS	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	STATUS				FOF	TRG	HALT	BKPT	HRL				-	BKD	-	IPW
RST	0				0	0	0	0	-				-	-	-	0
R/W†	R				R	R	R	R	R				-	-	-	R/W
BITS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	MAP	TRC	EMU	DDC		UHE	BTB		-	NPL	IPI	SSM	-			
RESET	0	0	0	0		0	0		0	0	0	0	-			
R/W†	R/W	R/W	R/W	R/W		R/W	R/W		R	R/W	R/W	R/W	-			

NOTE: †The CSR is a write only register from the programming model. It can be read from and written to via the BDM port.

Configuration/Status Register (CSR)

Field Definitions:

STATUS[31:28]–Breakpoint Status

This 4-bit field provides read-only status information concerning the hardware breakpoints. This field is defined as follows:

000x = no breakpoints enabled
001x = waiting for level 1 breakpoint
010x = level 1 breakpoint triggered
101x = waiting for level 2 breakpoint
110x = level 2 breakpoint triggered

The breakpoint status is also output on the DDATA port when it is not busy displaying other processor data. A write to the TDR resets this field.

FOF[27]–Fault-on-Fault

If this read-only status bit is set, a catastrophic halt has occurred and forced entry into BDM. This bit is cleared on a read from the CSR.

TRG[26]–Hardware Breakpoint Trigger

If this read-only status bit is set, a hardware breakpoint has halted the processor core and forced entry into BDM. This bit is cleared by reading CSR.

HALT[25]–Processor Halt

If this read-only status bit is set, the processor has executed the HALT instruction and forced entry into BDM. This bit is cleared by reading the CSR.

$\overline{\text{BKPT}}$[24]–Breakpoint Assert

If this read-only status bit is set, the BKPT signal was asserted, forcing the processor into BDM. This bit is cleared on a read from the CSR.

HRL[23:20]-Hardware Revision Level

This hardware revision level indicates the level of functionality implemented in the Debug Module. This information could be used by an emulator to identify the level of functionality supported. A zero value would indicate the initial debug functionality. For example, a value of 1 would represent Revision B while a value of 0 would represent the earlier release of Revision A.

BKD[18]-Disable the Normal BKPT Input Signal Functionality

This bit is used to disable the normal BKPT input signal functionality, and allow the assertion of this pin to generate a debug interrupt. If set, the assertion of the $\overline{\text{BKPT}}$ pin is treated as an edge-sensitive event. Specifically, a high-to-low edge on the $\overline{\text{BKPT}}$ pin generates a signal to the processor indicating a debug interrupt. The processor makes this interrupt request pending until the next sample point occurs. At that time, the debug interrupt exception is initiated. In the ColdFire architecture, the interrupt sample point occurs once per instruction. There is no support for any type of “nesting” of debug interrupts.

PCD[17]–PSTCLK Disable

If set, this bit disables the generation of the PSTCLK output signal, and forces this signal to remain quiescent.

IPW[16]–Inhibit Processor Writes to Debug Registers

If set, this bit inhibits any processor-initiated writes to the Debug Module's programming model registers. This bit can only be modified by commands from the external development system.

MAP[15]–Force Processor References in Emulator Mode

If set, this bit forces the processor to map all references while in emulator mode to a special address space, TT = $2, TM = $5 or $6. If cleared, all emulator-mode references are mapped into supervisor code and data spaces.

TRC[14]–Force Emulation Mode on Trace Exception

If set, this bit forces the processor to enter emulator mode when a trace exception occurs.

EMU[13]–Force Emulation Mode

If set, this bit forces the processor to begin execution in emulator mode. Refer to **Section 6.4.1.1 Emulator Mode**.

DDC[12:11]–Debug Data Control

This 2-bit field provides configuration control for capturing operand data for display on the DDATA port. The encoding is:

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

In all cases, the DDATA port displays the number of bytes defined by the operand reference size, i.e., byte displays 8 bits, word displays 16 bits, and long displays 32 bits (one nibble at a time across multiple clock cycles.) Refer to **Section 6.2.1.7 Begin Data Transfer (PST = $8 - $B)**.

UHE[10]-User Halt Enable

This bit selects the CPU privilege level required to execute the HALT instruction.

0 = HALT is a privileged, supervisor-only instruction
1 = HALT is a non-privileged, supervisor/user instruction

BTB[9:8]–Branch Target Bytes

This 2-bit field defines the number of bytes of branch target address to be displayed on the DDATA outputs. The encoding is

00 = 0 bytes
01 = lower two bytes of the target address
10 = lower three bytes of the target address
11 = entire four-byte target address

Refer to **Section 6.2.1.5 Begin Execution of Taken Branch (PST = $5)**.

NPL[6]–Non-Pipelined Mode

If set, this bit forces the processor core to operate in a nonpipeline mode of operation. In this mode, the processor effectively executes a single instruction at a time with no overlap.

When operating in non-pipilined mode, performance is severely degraded. For the V3 design, operation in this mode essentially adds 6 cycles to the execution time of each instruction. Given that the measured Effective Cycles per Instruction for V3 is ~2 cycles/ instruction, meaning performance in non-pipeline mode would be ~8 cycles/instruction, or approximately 25% compared to the pipelined performance.

Regardless of the state of CSR[6], if a PC breakpoint is triggered, it is always reported before the instruction with the breakpoint is executed. The occurrence of an address and/or data breakpoint trigger is imprecise in normal pipeline operation. When operating in non-pipeline mode, these triggers are always reported before the next instruction begins execution. In this mode, the trigger reporting can be considered to be precise.

As previously discussed, the occurrence of an address and/or data breakpoint should always happen before the next instruction begins execution. Therefore the occurrence of the address/data breakpoints should be guaranteed.

IPI[5]–Ignore Pending Interrupts

If set, this bit forces the processor core to ignore any pending interrupt requests signalled while executing in single-instruction-step mode.

SSM[4]–Single-Step Mode

If set, this bit forces the processor core to operate in a single-instruction-step mode. While in this mode, the processor executes a single instruction and then halts. While halted, any of the BDM commands may be executed. On receipt of the GO command, the processor executes the next instruction and then halts again. This process continues until the single-instruction-step mode is disabled.

6.4.2.7 BDM ADDRESS ATTRIBUTE (BAAR). TThe BAAR register defines the address space for memory-referencing BDM commands. Bits [7:5] are loaded directly from the BDM command, while the low-order 5 bits can be programmed from the external development system. To maintain compatibility with the Rev. A implementation, this register is loaded any time the AATR is written. The BAR is initialized to a value of $5, setting “supervisor data” as the default address space.

BITS	7	6	5	4	3	2	1	0
FIELD	R	SZ		TT		TM		
RESET	0	0		0		1	0	1
R/W	W	W		W		W		

BDM ADDRESS ATTRIBUTE REGISTER (BAAR)

Field Definitions:

R[7]–Read/Write

0 = Write

1 = Read

SZ[6:5]—Size

00 = Longword
01 = Byte
10 = Word
11 = Reserved

TT[4:3]—Transfer Type

See the TT definition in the AATR description, **Section 6.4.2.2.**

TM[2:0]—Transfer Modifier

See the TM definition in the AATR description, **Section 6.4.2.2**.

6.4.3 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 for the operations that access processor/memory registers:

- 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 then waiting until all current bus activity is complete. At that time, the processor relinquishes the local bus to allow the Debug Module to perform the required operation. After the conclusion of the Debug Module bus cycle, the processor reclaims ownership of the bus.

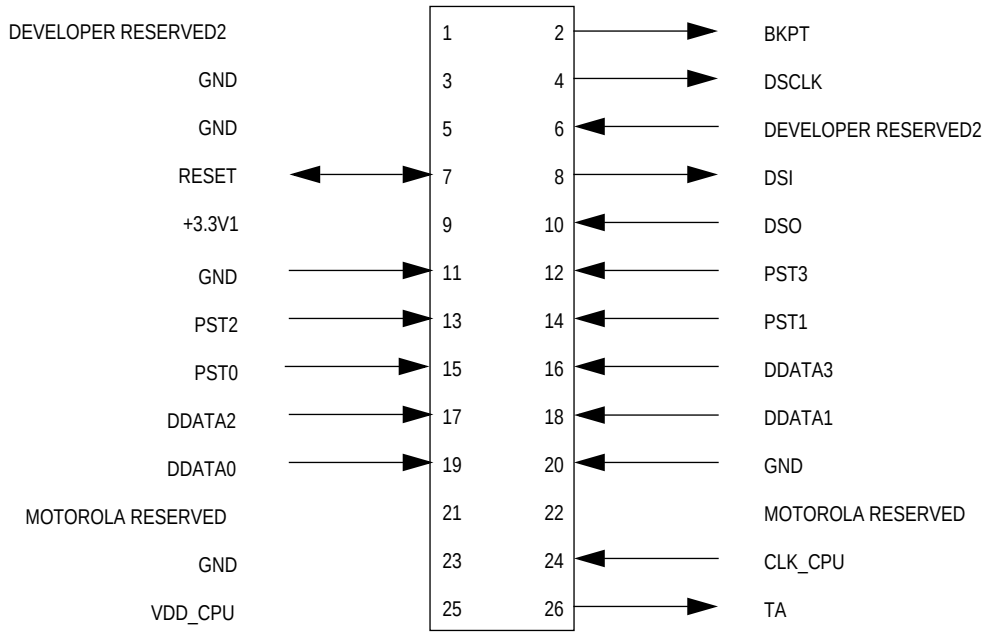
The development system must use caution in configuring the breakpoint registers if the processor is executing. The Debug Module does not contain any hardware interlocks, so Motorola recommends that the TDR be disabled while the breakpoint registers are being

loaded. At the conclusion of this process, the TDR can be written to define the exact trigger. This approach guarantees that no spurious breakpoint triggers occur.

Because there are no hardware interlocks in the debug unit, no BDM operations are allowed while the CPU is writing the debug's registers (BKPTDSCLK must be inactive).

6.4.4 Motorola-Recommended BDM Pinout

The ColdFire BDM connector is a 26-pin Berg Connector arranged 2x13, shown in Figure 6-9.



NOTES: 1. Supplied by target

2. Pins reserved for BDM developer use. Contact developer.

Figure 6-9. Recommended BDM Connector

SECTION 7
TEST

7.1 INTRODUCTION

This section describes how to use and integrate the test features of the Version 3 ColdFire embedded core (hereafter referred to as CF3Core). It is separated into several sub-sections that address the following topics:

- How to use the CF3Core test features.
- Understanding the CF3Core test wrapper (CF3TW).
- Interfacing and integrating the CF3Core test features within a chip.
- Reusing the CF3Core scan vectors and integrating them in the chip test program.

7.2 CF3CORE DESIGN-FOR-TEST

7.2.1 CF3Core Test Goals

The CF3Core embeddable core product has been designed to be tested in “isolation” with a self-contained “virtual test socket” known as a test wrapper. This allows the CF3Core to be delivered and integrated so that an existing vector set can be reused regardless of the ultimate chip configuration.

The purpose of the CF3Core test architecture and vector set is to verify that the manufacturing process did not introduce any faults, and that the “data sheet” specifications of the functional operation (structure), the frequency (clock speed) of operation of internal logic, and the signal timing involved with the CF3Core interface, are met.

The CF3Core has also been designed to support the ability to measure the quality level of the core in line with the manufacturing test requirements stipulated by Motorola’s Imaging Systems Division (ISD) as a standard requirement for all parts made and manufactured by Motorola. In addition to general logic and memory verification, these goals may include, logic retention, memory retention, power measurement (I_{dd} current), and current leakage (I_{ddq}).

7.2.2 CF3Core Test Features

To meet the defined test goals for the CF3Core, the core design includes test architectures for At-Speed Multiple-Chain Internal Full-Scan and At-Speed Test Wrapper Scan. The core design is also configured to allow for static operation and current measurement testing. Each of the test architectures is described more fully in the following sections.

7.2.2.1 FUNCTIONAL MODE WITH DEBUG. This mode represents the use of the core in normal operational mode with no test features active.

During functional mode, at the core/test wrapper boundary, the scan architecture is quiescent as long as the core and test wrapper scan enable signals are deasserted (placed to logic 0). Note that toggling of the scan enable signals of the core logic, or of the wrapper, can interfere with the functional operation of the CF3Core, so a quiescent default state must be guaranteed during chip design.

7.2.2.2 THE SCAN MODES. The scan modes enable the 32 CF3Core scan chains which are used to verify the general sequential and combinational logic for structure. Since the scan modes are designed to operate at the rated frequency, the scan chains are also used to verify the internal timing and the interface specifications. When the scan mode is coupled with a static verification (i.e., a tester pause - chip clock stop function) and/or a current measurement, then general logic retention and I_{ddq} testing are accomplished.

The production scan mode is designed to allow the scan architecture to operate at the full rated frequency with some memory protection logic enabled, whereas the burn-in scan mode is designed to allow the scan architecture to operate at a lower frequency with the memory arrays contributing to the high activity level.

7.2.2.3 THE CPU LOCK MODE. The CPU lock mode is provided to keep the processor in a quiescent state after the negation of the reset signal. This is required to minimize any core toggling while a chip-level non-core module is being tested, or executing some type of self-test. This encoding has been supplied because the test operation may apply clocks to the CF3Core logic. Note that the CPU is similarly locked when in any MBIST mode.

7.2.3 Alternate, Non-Covered Fault Models, Specialty Logic Test Support

Testing of alternate fault models and non-covered logic of the CF3Core is done in a similar manner to the static operation test, but using a current measurement technique during the pause (I_{ddq}). In scan mode, no extra step aside from stopping the clock is required.

7.3 CF3TW TEST ARCHITECTURE AND TEST INTERFACE

Since the ColdFire CF3Core was designed to be fully embedded and tested in isolation, a scannable Test Wrapper (CF3TW) interface was developed to provide dedicated test access in an optimized manner. The CF3TW is a dedicated test interface that is separate from the functional interface, but resides at the same hierarchical boundary. The goals of the CF3TW are:

- To allow the test wrapper to be a “virtual test socket” so the CF3Core can be tested independently of the rest of the chip in all test modes
- To allow all CF3Core functional signals to transparently pass through the test wrapper boundary during functional mode
- To allow a dedicated and minimized test interface that consists of a set of test mode signals, scan data and scan control signals
- To allow the non-core chip logic to be tested up to the CF3Core interface with no reliance on internal core logic or specific core test modes

- To use internal core and test wrapper-based at-speed scan chains to meet cost-of-test and fault coverage requirements of all logic within the core boundary
- To use a registered scan interface to allow a measure of independence from “external route timing”
- To allow the existing CF3Core vectors to be reused

The CF3TW can be viewed as the self-contained test access port for the internal CF3Core scan test architecture, the CF3Core/non-core interface boundary scan test architecture, and the internal memory BIST architecture. The test port signals are described in the table below.

Table 7-1. CF3TW Test Features and Signals

Test Signal Name	Test Use	Connection
si[31:0]	CF3Core Parallel Scan Inputs	Input to CF3Core
so[31:0]	CF3Core Parallel Scan Outputs	Output from CF3Core
se	CF3Core Parallel Scan Enable	Input to CF3Core
tbsi[3:0]	CF3TW Interface Scan Inputs	Input to CF3TW
tbso[3:0]	CF3TW Interface Scan Outputs	Output from CF3TW
tbsei	CF3TW Interface Input Scan Enable	Input to CF3TW
tbseo	CF3TW Interface Output Scan Enable	Output from CF3TW
tbte	CF3TW Interface Test Enable	Input to CF3TW

7.3.1 Access to the CF3Core Internal Scan Architecture

The CF3TW provides access to the CF3Core scan architecture. The CF3Core scan architecture consists of 32 scan chains with a target shift bit depth of 151 scan cells each. Access to the scan architecture is through the **si**, **so**, and **se** signals that pass through the CF3TW boundary. To use the scan architecture, the **mtmod** signals must be placed in either the $9 or $B encodings to configure the internal core features for scan operation. The internal scan architecture must be operated in conjunction with the test wrapper scan chains which are accessed through the **tbsi, tbso, tbte, tbsei**, and **tbseo**.

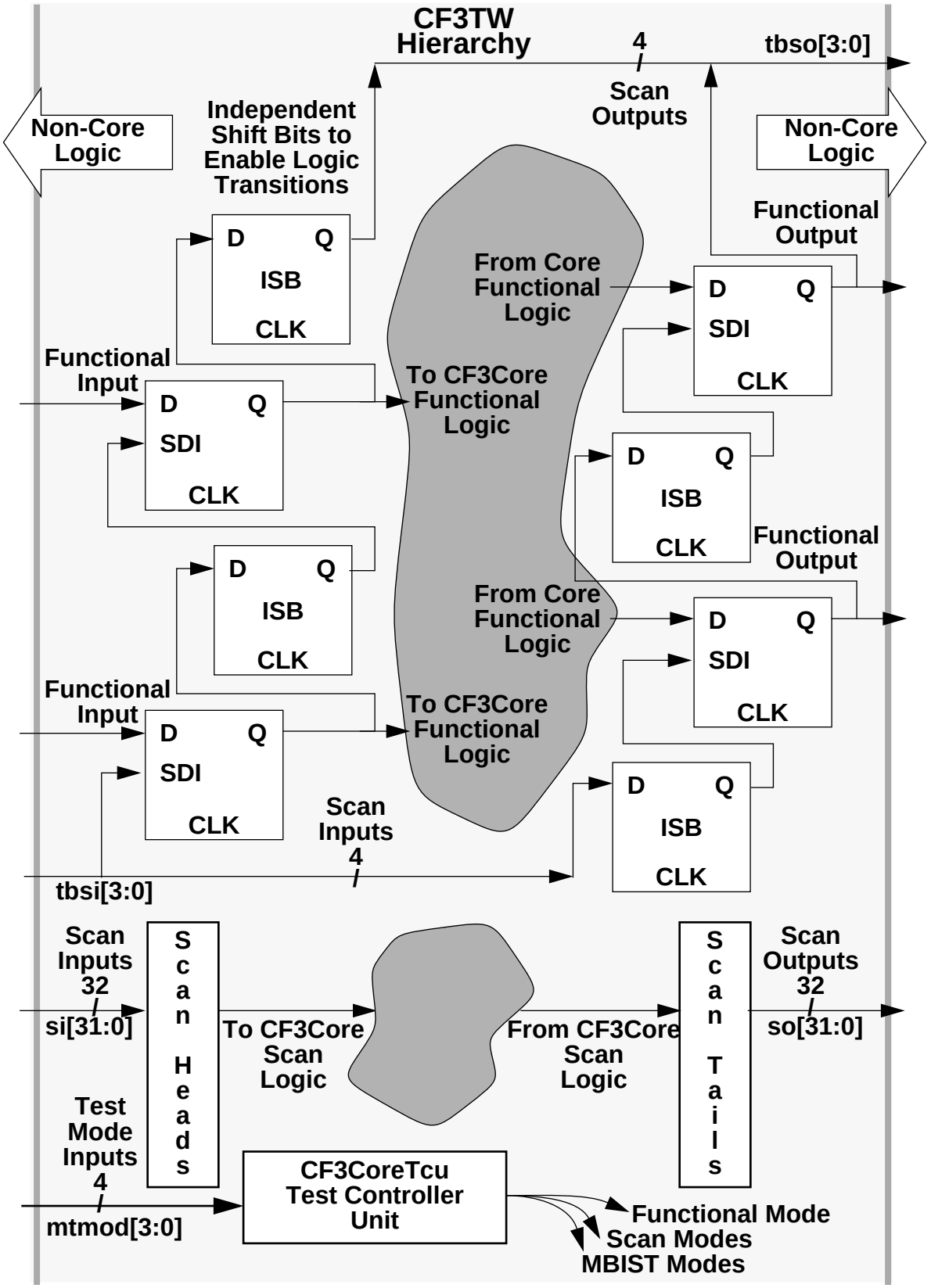


Figure 7-1. Example Registered CF3TW Architecture

7.3.2 The CF3TW Boundary Scan Architecture

The CF3TW scan architecture is provided as a separate scan architecture so it may be used in conjunction with CF3Core testing and non-core logic testing. In cases where intellectual property concerns do not allow a gate-level netlist to be delivered to the integrator, a structural interface model is needed for vector generation of the non-core logic. The CF3TW was developed for this purpose.

The CF3TW is a scan architecture made from existing CF3Core registers since this particular test wrapper relies on the fact that almost all of the input and output functional signals of the CF3Core are naturally registered. The at-speed CF3TW test wrapper scan chain is the collection of these functional registers into scan chains.

The placing of the functional input and output registers into the test wrapper scan chains allows the control and observation of interface logic values without requiring the connection of the functional interface signals to the chip-level package pins. It must also be noted that any at-speed data transfers to and from the CF3TW verify the functional interface timing since the included registers within the wrapper are the functional registers.

The main purpose of the CF3TW is to provide a boundary scan architecture for the CF3Core to allow the core to be tested in isolation. The basic CF3TW operation allows the functional input signals to be applied and the output signals to be observed for both AC and DC structural verification without bringing the entire functional interface to chip-level package pins. A secondary purpose of the CF3TW is to provide a similar function for the non-core logic. Here, the core's output registers can be loaded and driven into the non-core logic, and the results from the non-core logic observed on the CF3TW's input registers.

The CF3TW scan test architecture consists of four scan chains that begin on the **tbsi[3:0]** signals (wrapper scan inputs), and end on the **tbso[3:0]** signals (wrapper scan outputs). One scan chain includes all of the functional input signal registers, and the remaining three scan chains include all of the functional output registers. Each scan chain contains a number of shift bits as described in the table below.

Table 7-2. CF3TW Scan Architecture Signals

Wrapper Scan Ports	Bit Length/ Fanout	Comment
tbsi[0]->tbso[0]	TBD (42)	Functional Input Signals
tbsi[1]->tbso[1]	TBD (31)	Functional Output Signals
tbsi[2]->tbso[2]	TBD (31)	Functional Output Signals
tbsi[3]->tbso[3]	TBD (31)	Functional Output Signals
tbsei	TBD (83)	Scan Enable Input Side
tbseo	TBD (180)	Scan Enable Output Side
tbte	TBD (2)	Non-Register Test Enable

7.3.2.1 CF3TW TESTING OF NON-CORE INPUTS. The CF3TW has been designed to be used as a stand-alone device that does not need the CF3Core scan architecture to conduct all testing. The CF3TW can be appended to the Non-Core logic to become part of it's test structure. This requires the use of a gate-level netlist of the CF3TW to be included as part of the Non-Core logic netlist when vector generation is to be accomplished.

One of the ways that the CF3TW has been designed to operate is in the "Wrapper Scan Launch Mode". This is a test mode that uses the ability of the CF3TW to launch single logic values or vector-pair logic transition values into the Non-Core logic. This type of testing is accomplished by utilizing the **tbseo** signal which enables the CF3TW scan architecture to either shift data through the CF3TW output side scan chains (launching data into the Non-Core logic), or to capture data from the CF3Core logic. The CF3TW can be operated coincidentally with the Non-Core logic test structures, and can be used to enable structural testing or timing delay testing.

The **tbseo** signal in conjunction with the Non-Core logic scan or functional test mode control signals will allow the launching of a single logic value to conduct structural stuck-at testing, or will allow the launching of 2 consecutive differing logic values (vector-pairs) on targeted input signals, while holding other signals stable for 2 cycles (applying the same value). The two-cycle transition type of sequence that holds "off-path" values stable results in what is known as a Robust Delay Test.

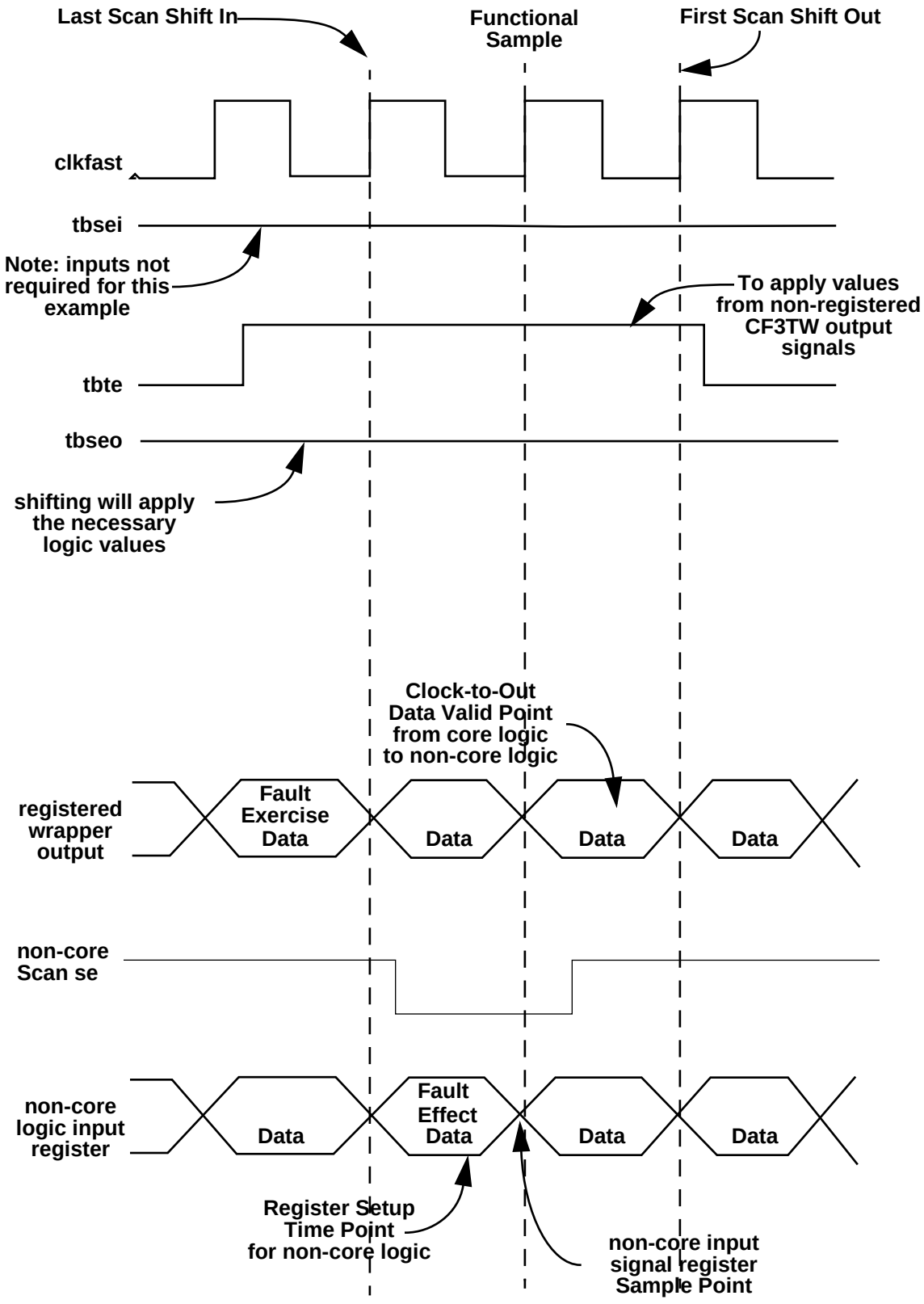


Figure 7-2. CF3TW to Non-Core Input Scan Stuck-At Vector

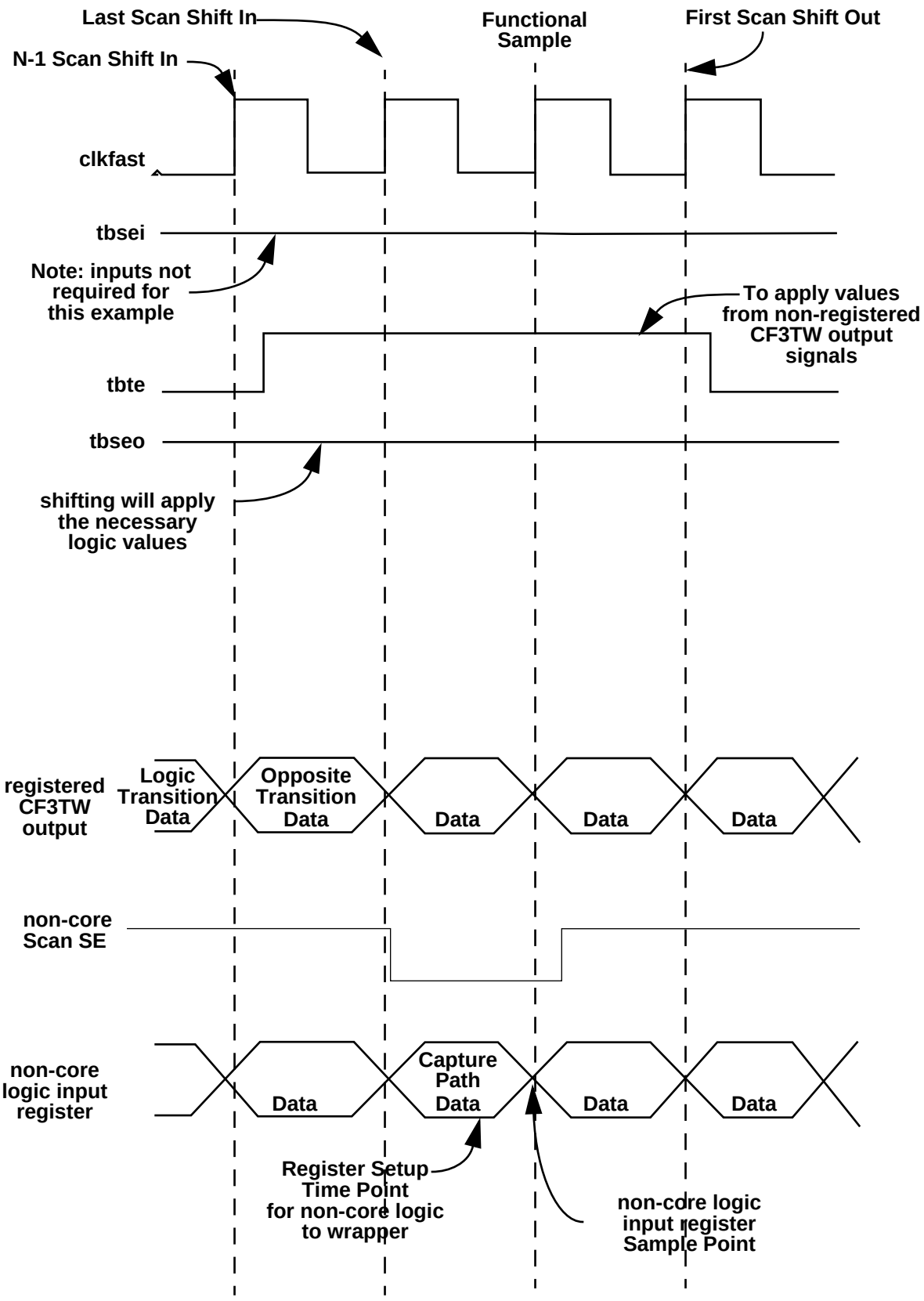


Figure 7-3. CF3TW to Non-Core Delay Scan Vector Example

7.3.2.2 CF3TW TESTING OF NON-CORE OUTPUTS. Another of the CF3TW stand-alone modes (operation independent of the CF3Core scan architecture) is the Wrapper Scan Capture Mode. The CF3TW can be appended to the Non-Core logic to become the capture part of it's test structure. Using the CF3TW to launch or capture logic values associated with the Non-Core logic requires the use of a gate-level netlist of the CF3TW to be included as part of the Non-Core logic netlist when vector generation is to be accomplished.

One of the ways that the CF3TW has been designed to operate is in the Wrapper Scan Capture Mode. This is a test mode that uses the ability of the CF3TW to capture logic values or logic transition values launched from the Non-Core logic. The CF3TW can be operated coincidentally with the Non-Core logic test structures, and can be used to enable structural testing or timing delay testing.

The Wrapper Scan Capture Mode testing is accomplished by utilizing the **tbsei** signal in conjunction with the Non-Core logic scan or functional test mode control signals to allow the capture of single logic values, or 2 consecutive differing logic values on targeted input signals.

The **tbsei** signal will enable the CF3TW scan architecture to either shift data through the CF3TW input side scan chains (launching logic values into the CF3Core), or to capture data from the non-core logic. If the Non-Core logic has the ability to launch transitions (vector pairs), then 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 Non-Core logic requires that the Non-Core logic supports the logic test structures to launch the vector-pairs.

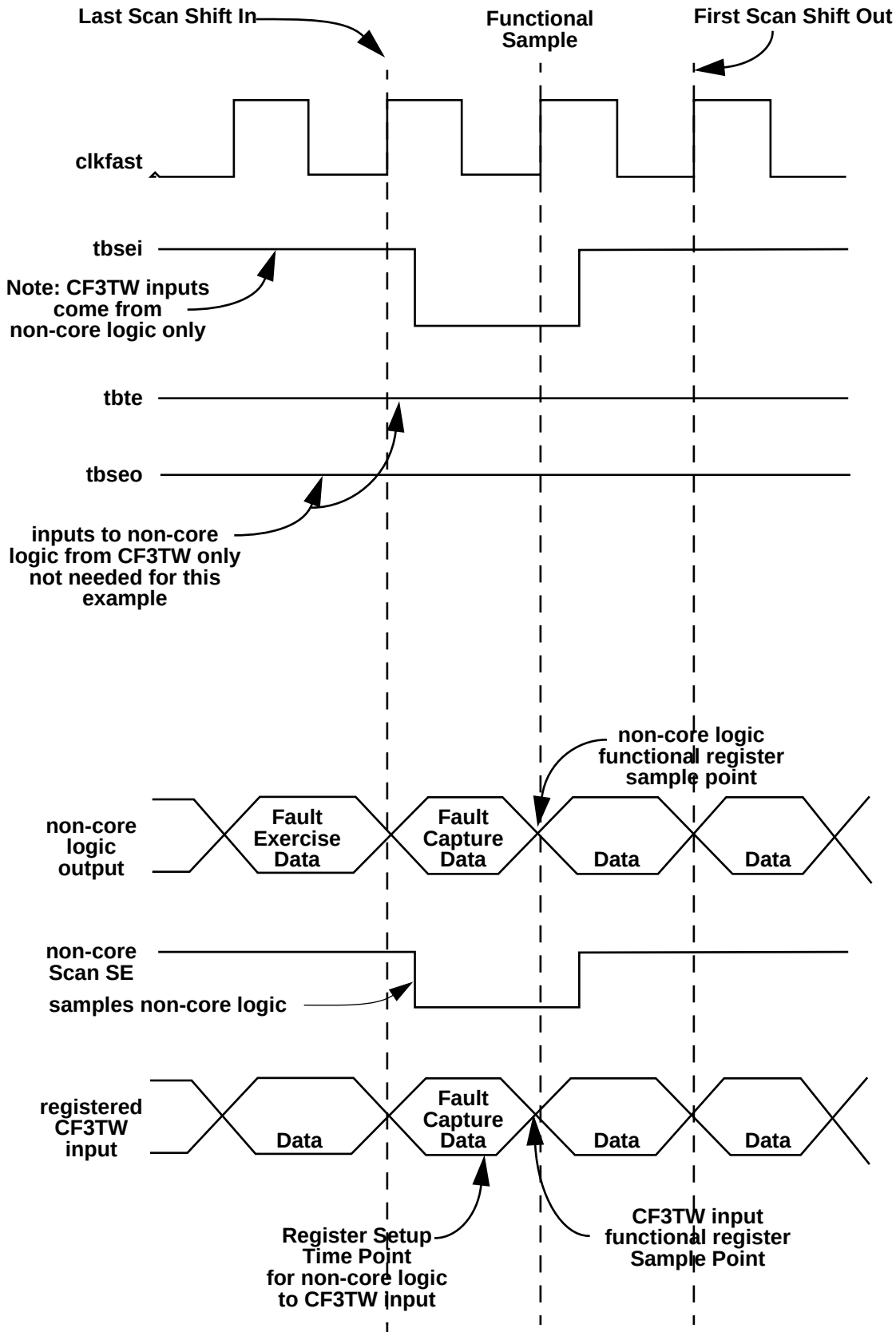


Figure 7-4. Non-Core to CF3TW Input Scan Stuck-At Vector Example

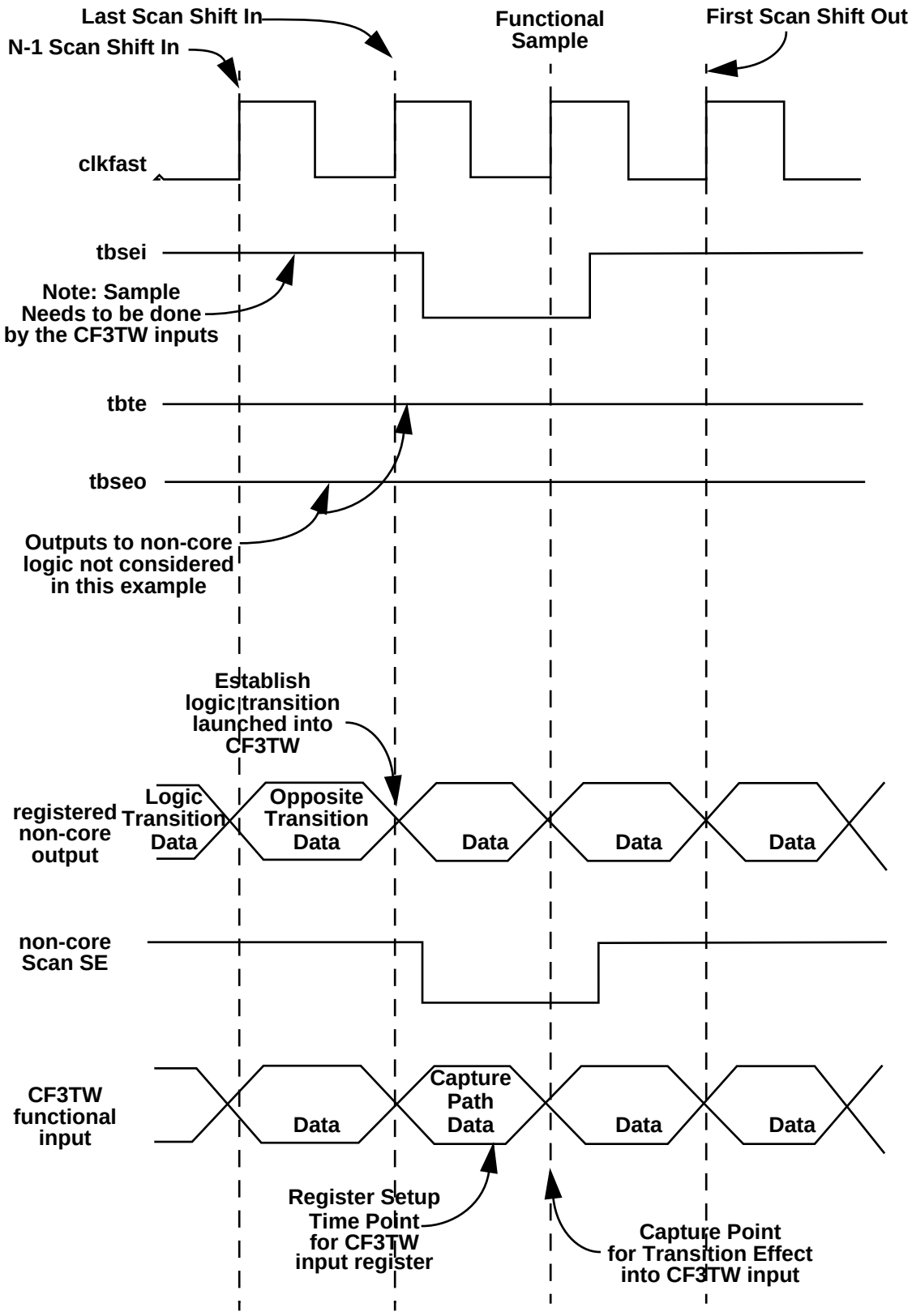


Figure 7-5. Non-Core to CF3TW Input Scan Delay Vector Example

7.4 CHIP-LEVEL INTEGRATION & TEST ISSUES

At the chip level, the CF3Core may be just one of several design units to be integrated into the overall device. It is the task of the chip integrator to understand and to connect the CF3Core test features to meet the overall test goals of the final chip, and to allow reuse of the existing vectors. In order to understand the trade-offs for the several possible ways to connect the CF3Core test architecture, and to design the interplay between the CF3Core test features and the test features of other Non-CF3Core design units, the chip-level test goals need to be understood and described.

The basic test goal of any chip is to ensure that the design includes the test architecture features that provide a high level of quality measurement at an economical cost. Since the CF3Core has been designed with “testing in isolation” as a goal, this section describes the goals and issues of this type of methodology.

When multiple embedded cores are included within a chip, and they are to be tested in isolation, there are chip-level issues and architectures that must be addressed. Also, at the chip-level, there may be test issues independent of the embedded cores (such as the inclusion of IEEE 1149.1 - JTAG on the chip). The first step in addressing these issues is specification of the chip-level test goals. For example, some chip-level test goals may be:

- Providing a chip-level test architecture for board-level chip integration (e.g., JTAG, De-bug/Real-Time Trace)
- Conducting “whole chip” I_{ddq} testing
- Conducting “whole chip” burn-in testing
- Meeting a “whole chip” test time, or test cost budget
- Meeting a “whole chip” test program size limitation
- Meeting a “whole chip” structural (stuck-at) standard
- Meeting an individual chip component’s structural (stuck-at) standard (e.g., Core A must meet M%, Core B must meet N%, etc.)
- Meeting a “whole chip” frequency, timing and delay fault standard
- Meeting an individual chip component frequency, timing and delay fault standard
- Applying pre-existing vectors for chip components (e.g., Core A, Core B, etc.) on silicon in an economical fashion
- Providing access to individual chip component test architectures without significantly compromising “whole chip” design goals such as silicon area/die size, frequency, architectural performance, power consumption, etc.
- Testing individual chip components simultaneously if power consumption permits (i.e., test time reduction)

7.4.1 Chip-Level Test Program Goals

Motorola understands the core test program is just one portion of the overall test program, and has designed the test features to produce a cost-optimized vestor set. The chip test program may include DC parametrics, scan vectors, memory test vectors, retention vectors, and any specialty logic vectors (e.g., PLL, A/D, etc.). A chip containing cores that are meant

to be tested in isolation, however, may have a complete test program for each core. This means that the overall chip-level test program has DC parametrics, then the whole test program for Core A, the whole test program for Core B, and then some Core A/Core B interaction vectors (where each test program may have I_{ddq} and retention vectors included).

Integration options can impact test time and cost. If the core test program can be operated simultaneously with other chip test program components, then test time may be optimized.

During the chip-level integration of individual cores, the goals involving test program optimization need to be addressed. One of the most significant time impacts is retention testing. It is advisable to reduce the tester pause operations to a minimal number.

The vectors for the core are generated for the specific deployed configuration, and can be adjusted to the chip environment through the ISD test kit. The test kit can translate the core test ports to their ultimate chip-level pins, and can add any chip-level vector preamble to put the chip in core test mode.

7.4.2 CF3Core Integration Connections

To reuse the vectors delivered with the CF3Core, the core test features must be integrated in a particular manner. This section will discuss the proper connection requirements.

The four **mtmod** test input signals must be routed from the package pins to the CF3CoreTcu. This can be a direct connection from the package pins to the CF3Core, or can be some other form of dedicated chip-level test selection that creates the four **mtmod** signals in a chip-level test controller.

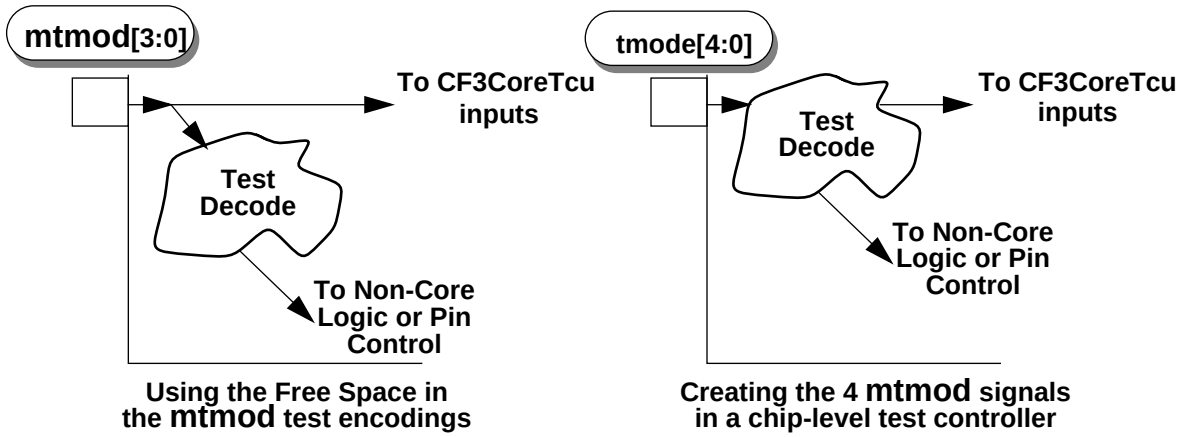


Figure 7-6. Two Allowed Methods of mtmod distribution

The method of connection of the **mtmod** signals is fundamental to the integration options of the CF3Core. The different possible connections can limit the ability to share test resources between the CF3Core and non-core logic. If the integration decision is to map the chip test modes into the unused encoding space of the four **mtmod**s, then the non-core test modes are mutually exclusive from the CF3Core test modes. In some cases, it would be advisable for the non-core test logic and the CF3Core test logic to operate simultaneously to reduce test time.

Also, the predetermined or predesigned integration of mutual exclusivity or shared modes

can limit the ability to meet some test goals. For example, mutual exclusivity would not allow “whole chip” I_{ddq} or burn-in.

Another method that can be used to distribute the **mtmod** to the CF3Core is to create them, or pass them through, a chip-level test controller. For example, five package pins can be used to create up to 32 possible test modes. The lower four signals may be passed on to the CF3Core only when the fifth signal (MSB) is a logic 0, and the four signals can be forced to place the CF3Core in the production scan mode when the fifth signal is set to a logic 1 (where these represent non-core logic or chip-level test modes).

Using a chip-level test controller allows the ability to encompass both the mutually exclusive and the shared modes. The production scan test mode for the CF3Core, for example, may be a mutually exclusive mode when the fifth **mtmod** package pin is a logic 0 and the four lower **mtmod** signals are set to the $B encoding; the non-core logic may be in a mutually exclusive scan test mode when the fifth **mtmod** is at a logic 0 and the lower four **mtmod** signals are in one of the unused encodings such as $8; but when the fifth package pin is set to a logic 1 with the lower four **mtmod** encoding of $B, then both scan architectures may be active.

Another example would be having one mode for CF3Core scan burn-in mode, one mode for non-core logic burn-in mode, and then one mode which has all of the scan inputs (CF3Core and non-core logic) being fed from the same pins, but having outputs on different pins. This allows an external pseudo-random pattern generator (PRPG) to place random patterns into both functions at the same time. However, the response evaluation must still come from the endpoints of each separate scan chain.

In summary:

- The four **mtmod** signals can be passed directly to the CF3Core from the package pins
- The remaining four **mtmod** encodings can be used to make mutually exclusive non-core or chip-level test modes
- The twelve **mtmod** encodings used for the CF3Core can be shared if hardware resources or power consumption permits
- The four **mtmod** signals can be created within a chip-level TCU and distributed to the CF3Core
- Both mutually exclusive modes and shared modes can exist if the chip-level TCU mediates more than 16 test modes
- The chip-level TCU can use any method to create the test modes (e.g., the test register, state machine, combinational encoding).

7.4.3 CF3Core Scan Connections

The rules for the connection of the CF3Core scan interface are straightforward. When the **mtmod** encoding distributed to the CF3Core is either $B or $9, then the **si**, and **tbsi** scan inputs must be connected to package pins as inputs; the **so** and **tbso** scan outputs must be connected to package pins as outputs; and the **se, tbte, tbsei**, and **tbseo** scan shift control signals must be connected to package pins as inputs.

When a scan mode is not selected, then the parallel scan inputs must be driven to a logic 0, and the parallel scan outputs must be ignored. Since the test wrapper scan architecture is

also used by the non-core logic, then these connections should not be mode-gated to be limited to use during CF3Core scan modes.

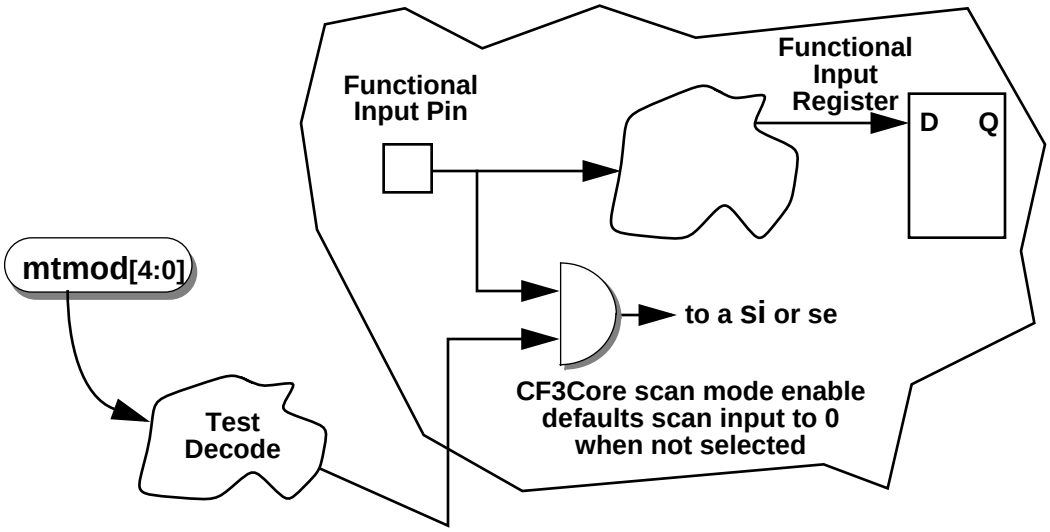


Figure 7-7. Chip-Level CF3Core Parallel Scan Input Connection

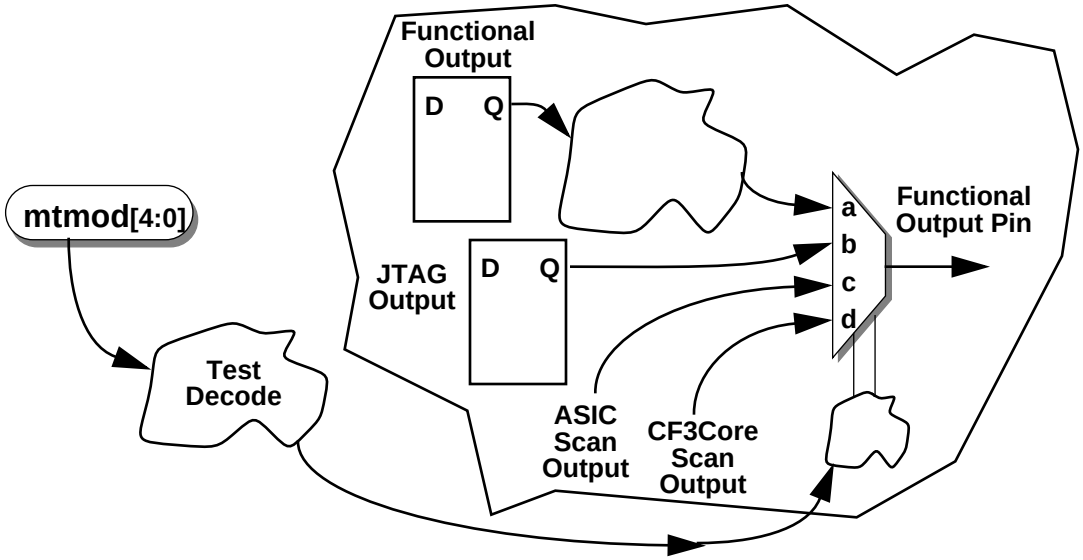


Figure 7-8. Chip-Level CF3Core Parallel Scan Output Connection

The CF3TW is a shared test resource for the CF3Core and the non-core logic alike. This means that access to the CF3TW connections must be provided in both test modes.

The CF3Core parallel scan connections can be gated off whenever they are not being used in a scan mode (when encodings $B and $9 are not selected), but the CF3TW scan connections may be used across several test modes. The package pin connections used for the CF3Core scan can be used, but the select signal must allow more test modes to access the scan architecture. This should include any CF3Core scan test mode, any non-core logic

APPENDIX A
CF3CORE INTERFACE TIMING CONSTRAINTS

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 CF3Core design. The relative timings are expressed as a fraction of the processor's cycle time which essentially provides a technology-independent budget. Note this timing budget file is provided as reference, and the actual timing specification on each interconnection pin is a function of the process technology, synthesis methodology, place-and-route details and external signal loading.

In this timing constraint budget, `clk_max_period` defines the period of the processor's fast clk, and VCLK is simply a virtual clock reference with the same period as the `clk_max_period`. The virtual clock is used as a method to reference input and output timings.

```
/*****************************************************************************/
/*****************************************************************************/
//
// Version 3 ColdFire Reference Design INPUT/OUTPUT SIGNALS
//
/*****************************************************************************/
/*****************************************************************************/

/* OUTPUTS */

set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, maddr*)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mtt*)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mtm[*])
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mrw)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, msiz*)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mwdata[*])
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mwdataoe)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mapb)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, mdpb)
```

```
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, cpustopb)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, cpuhaltb)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, enpstclk)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, pst[*])
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, ddata*)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, dsdo)

/* core scan and test boundary output signals */
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, so)
set_output_delay { clk_max_period - (0.20   * clk_max_period)} -clock VCLK
 find(port, tbso)

/* outputs to the Unified Cache tag and data arrays */
/* outputs to tag */
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsentb)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nswrttb)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nswlvt*)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsinvat)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsrowst*)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsaddrt*)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nssw)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nssv)

/* outputs to array */
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsendb)
set_output_delay { clk_max_period - (0.80  * clk_max_period)} -clock VCLK
 find(port, nswrtdb*)
set_output_delay { clk_max_period - (0.80  * clk_max_period)} -clock VCLK
 find(port, nswtbyted*)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
 find(port, nsrowsd*)
set_output_delay { clk_max_period - (0.82  * clk_max_period)} -clock VCLK
```

```
 find(port, nscwrdata*)

/* outputs to the KRAM data array */
set_output_delay { clk_max_period - (0.85   * clk_max_period)} -clock VCLK
 find(port, kramaddr*)
set_output_delay { clk_max_period - (0.85   * clk_max_period)} -clock VCLK
 find(port, kramdi*)
set_output_delay { clk_max_period - (0.75   * clk_max_period)} -clock VCLK
 find(port, kramweb*)
set_output_delay { clk_max_period - (0.85   * clk_max_period)} -clock VCLK
 find(port, kramcsb)

/* output to the KROM data array */
set_output_delay { clk_max_period - (0.85   * clk_max_period)} -clock VCLK
 find(port, kromaddr*)
set_output_delay { clk_max_period - (0.85   * clk_max_period)} -clock VCLK
 find(port, kromcsb)

/*****************************************************************************/

/* INPUTS */

set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, mrdata*)
set_input_delay { 0.25  * clk_max_period } -clock VCLK find(port, mtab)
set_input_delay { 0.25  * clk_max_period } -clock VCLK find(port, mahb)

set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, miplb*)
set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, mrstib)
set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, dsclk)
set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, dsdi)
set_input_delay { 0.75  * clk_max_period } -clock VCLK find(port, mbkptb)

/* core scan and test boundary input signals */
/* test mode select */
set_input_delay { 0.10   * clk_max_period } -clock VCLK find(port, bistplltest)

/* parallel core scan input signals */
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, si*)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, se)

/* test boundary input signals */
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, tbsi*)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, tbsei)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, tbseo)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, tbte)
```

```
/* inputs from the U-Cache memory arrays + configuration */
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, ucsz)
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, ucnoif)
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, ucnoop)

/* inputs from the Unified Cache tag and data arrays */
/* inputs - from tag */
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uctag3do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucw3do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucv3do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uctag2do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucw2do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucv2do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uctag1do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucw1do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucv1do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uctag0do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucw0do)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, ucv0do)

/* inputs - from array */
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uclvl3do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uclvl2do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uclvl1do*)
set_input_delay { 0.45  * clk_max_period } -clock VCLK find(port, uclvl0do*)

/* inputs from the KRAM memory + configuration */
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, kramsz)
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, encf5307kram)
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, enraptorkram)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, kramdo*)

/* inputs from the KROM memory + configuration */
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, kromsz)
set_input_delay { 0.0   * clk_max_period } -clock VCLK find(port, kromvldrst)
set_input_delay { 0.50  * clk_max_period } -clock VCLK find(port, kromdo*)

/* processor clock enables */
set_input_delay { 0.70  * clk_max_period } -clock VCLK find(port, mclken)


set_load -pin_load  0.5 all_outputs()
```

APPENDIX B
INSTRUCTION EXECUTION TIMES

This appendix provides detailed instruction execution timings for the CF3Core processor complex.

B.1 TIMING ASSUMPTIONS

For the timing data presented in this section, the following assumptions apply:

1. The operand execution pipeline (OEP) is loaded with the opword and all required extension words at the beginning of each instruction execution. This implies that the OEP does not wait for the instruction fetch pipeline (IFP) to supply opwords and/or extension words.
2. The OEP does not experience any sequence-related pipeline stalls. For Version 2 and Version 3 ColdFire processors, the most common example of this type of stall involves consecutive store operations, excluding the MOVEM instruction. For all STORE operations (except MOVEM), certain hardware resources within the processor are marked as “busy” for two clock cycles after the final DSOC cycle of the store instruction. If a subsequent STORE instruction is encountered within this 2-cycle window, it is stalled until the resource again becomes available. Thus, the maximum pipeline stall involving consecutive STORE operations is 2 cycles. The MOVEM instruction uses a different set of resources and this stall does not apply.
3. The OEP completes all memory accesses without any stall conditions caused by the memory itself. Thus, the timing details provided in this section assume that an infinite zero-wait state memory is attached to the processor core.
4. All operand data accesses are aligned on the same byte boundary as the operand size, i.e., 16-bit operands aligned on 0-modulo-2 addresses, 32-bit operands aligned on 0-modulo-4 addresses.

If the operand alignment fails these guidelines, it is misaligned. The processor core decomposes the misaligned operand reference into a series of aligned accesses as shown in Table B-1..

Table B-1. Misaligned Operand References

ADDRESS[1:0]	SIZE	BUS OPERATIONS	ADDITIONAL C(R/W)
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

B.2 MOVE INSTRUCTION EXECUTION TIMES

The execution times for the MOVE.{B,W} instructions are shown in Table 2, while Table 3 provides the timing for MOVE.L.

For all tables in this section, the execution time of any instruction using the PC-relative effective addressing modes is the same for the comparable An-relative mode.

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

Table B-2. 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)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
(Ay)+	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
-(Ay)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
(d16,Ay)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—
(d8,Ay,Xi*SF)	5(1/0)	5(1/1)	5(1/1)	5(1/1)	—	—	—
xxx.w	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
xxx.l	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
(d16,PC)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—
(d8,PC,Xi*SF)	5(1/0)	5(1/1)	5(1/1)	5(1/1)	—	—	—
#xxx	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

Table B-3. 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)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(Ay)+	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
-(Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(d16,Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,Ay,Xi*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
xxx.w	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
xxx.l	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—

Table B-3. Move Long Execution Times (Continued)

SOURCE	DESTINATION						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	xxx.wl
(d16,PC)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,PC,Xi*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
#xxx	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

The following table specifies the execution times for the Move Long instructions accessing the program-visible registers of the MAC unit.

Table B-4. MAC Move Long Instruction 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	1(0/0)	—	—	—	—	—	—	1(0/0)
move.l	<ea>,MASK	1(0/0)	—	—	—	—	—	—	1(0/0)
move.l	ACC,Rx	3(0/0)	—	—	—	—	—	—	—
move.l	MACSR,CCR	3(0/0)	—	—	—	—	—	—	—
move.l	MACSR,Rx	3(0/0)	—	—	—	—	—	—	—
move.l	MASK,Rx	3(0/0)	—	—	—	—	—	—	—

B.3 STANDARD ONE OPERAND INSTRUCTION EXECUTION TIMES

Table B-5. 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)	—	—	—	—	—	—	—
scc	Dx	1(0/0)	—	—	—	—	—	—	—
swap	Dx	1(0/0)	—	—	—	—	—	—	—
tst.b	<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
tst.w	<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
tst.l	<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)

B.4 STANDARD TWO OPERAND INSTRUCTION EXECUTION TIMES

Table B-6. Two Operand Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xi*SF) (d8,PC,Xi*SF)	xxx.wl	#xxx
add.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
add.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
addi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
addq.l	#imm,<ea>	1(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
addx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—
and.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
and.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(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)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bchg	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
bclr	Dy,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bclr	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
bset	Dy,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bset	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
btst	Dy,<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	—
btst	#imm,<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	—	—	—
cmp.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
cmpi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
divs.w	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(1/0)	20(0/0)
divu.w	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(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)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(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	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
mac.l	Ry,Rx,ea,Rw	—	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
msac.w	Ry,Rx,ea,Rw	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
msac.l	Ry,Rx,ea,Rw	—	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
moveq	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
muls.w	<ea>,Dx	3(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	3(0/0)
mulu.w	<ea>,Dx	3(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	3(0/0)
muls.l	<ea>,Dx	5(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	—	—	—
mulu.l	<ea>,Dx	5(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	—	—	—
or.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
or.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—

Table B-6. Two Operand Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xi*SF) (d8,PC,Xi*SF)	xxx.wl	#xxx
ori.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
rems.l	<ea>,Dx:Dw	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
remu.l	<ea>,Dx:Dw	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
sub.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
sub.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
subi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
subq.l	#imm,<ea>	1(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
subx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—

B.5 MISCELLANEOUS INSTRUCTION EXECUTION TIMES

Table B-7. Miscellaneous Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	xxx.wl	#xxx
cpushl	(Ax)	—	11(0/1)	—	—	—	—	—	—
halt		6(0/0)	—	—	—	—	—	—	—
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	9(0/0)	—	—	—	—	—	—	9(0/0)[1]
movec	Ry,Rc	11(0/1)	—	—	—	—	—	—	—
movem.l	<ea>,&list	—	2+n(n/0)[2]	—	—	2+n(n/0)[2]	—	—	—
movem.l	&list,<ea>	—	2+n(0/n)[2]	—	—	2+n(0/n)[2]	—	—	—
nop		3(0/0)	—	—	—	—	—	—	—
pea	<ea>	—	2(0/1)	—	—	2(0/1)[4]	3(0/1)[5]	2(0/1)	—
pulse		1(0/0)	—	—	—	—	—	—	—
stop	#imm	—	—	—	—	—	—	—	3(0/0)[3]
trap	#imm	—	—	—	—	—	—	—	18(1/2)
trapf		1(0/0)	—	—	—	—	—	—	—
trapf.w		1(0/0)	—	—	—	—	—	—	—
trapf.l		1(0/0)	—	—	—	—	—	—	—
unlk	Ax	3(1/0)	—	—	—	—	—	—	—
wddata	<ea>	—	7(1/0)	7(1/0)	7(1/0)	7(1/0)	8(1/0)	7(1/0)	—
wdebug	<ea>	—	10(2/0)	—	—	10(2/0)	—	—	—

[1] If a MOVE.W #imm,SR instruction is executed and #imm[13] = 1, the execution time is 1(0/0).
[2] n is the number of registers transferred by the MOVEM opcode.
[3] The execution time for STOP is the time required until the processor begins sampling continuously for interrupts.
[4] PEA execution times are the same for (d16,PC).
[5] PEA execution times are the same for (d8,PC,Xi*SF).

B.6 BRANCH INSTRUCTION EXECUTION TIMES

Table B-8. General Branch Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xi*SF) (d8,PC,Xi*SF)	xxx.wl	#xxx
bsr		—	—	—	—	1(0/1)[2]	—	—	—
jmp	<ea>	—	5(0/0)	—	—	5(0/0)[1]	6(0/0)	1(0/0)[1]	—
jsr	<ea>	—	5(0/1)	—	—	5(0/1)	6(0/1)	1(0/1)[2]	—
rte		—	—	14(2/0)	—	—	—	—	—
rts		—	—	8(1/0)	—	—	—	—	—

Table B-9. BRA, Bcc Instruction Execution Times

OPCODE	FORWARD TAKEN	FORWARD NOT TAKEN	BACKWARD TAKEN	BACKWARD NOT TAKEN
bra	1(0/0)[1]	—	1(0/0)[1]	—
bcc	5(0/0)	1(0/0)	1(0/0)[3]	5(0/0)

The following notes apply to the branch execution times:

1. For the jmp <ea> instructions, where <ea> is (d16,PC) or xxx.wl, the branch acceleration logic of the Instruction Fetch Pipeline calculates the target address and begins prefetching the new path. Since the Instruction Fetch and Operand Execution Pipelines are decoupled by the FIFO instruction buffer, the execution time can vary from 1 to 3 cycles, depending on the amount of decoupling.

This same mechanism is used for the bra opcode.

For all other <ea> values of the jmp instruction, the branch acceleration logic is not used, and the execution times are fixed.

2. For the jsr xxx.wl opcodes, the same branch acceleration mechanism is used to initiate the fetch of the target instruction. Depending on the amount of decoupling between the IFP and OEP, the resulting execution times can vary from 1 to 3 cycles.

The same acceleration techniques are applied to the bsr opcode.

For the remaining <ea> values for the jsr instruction, the branch acceleration logic is not used, and the execution times are fixed.

3. For conditional branch opcodes (bcc), there is a static algorithm used to determine the prediction state of the branch. This algorithm is:

```
if bcc is a forward branch && CCR[7] == 0
    then the bcc is predicted as not-taken

if bcc is a forward branch && CCR[7] == 1
    then the bcc is predicted as taken

    else if bcc is a backward branch
         then the bcc is predicted as taken
```

The execution times in the BRA, Bcc Table assume that CCR[7] is negated. Another representation of the Bcc execution times is shown below:

Table B-10. Another Table of Bcc Instruction Execution Times

OPCODE	PREDICTED CORRECTLY AS TAKEN	PREDICTED CORRECTLY AS NOT-TAKEN	MISPREDICTED
bcc	1(0/0)	1(0/0)	5(0/0)

The execution time for the “predicted correctly as taken” column can vary between 1 to 3 cycles depending on the amount of decoupling between the Instruction Fetch and Operand Execution Pipelines as previously discussed.

APPENDIX C
PROCESSOR STATUS, DDATA DEFINITION

This section specifies the Version 3 ColdFire processor's generation of the Processor STatus (**PST**) and Debug DATA (**DDATA**) output pins for each instruction.

The instruction set of the ColdFire processor can be separated into 2 groups: user and supervisor opcodes. *In general, the **PST/DDATA** outputs for an instruction are defined as:*

PST = 1, {PST = [89B], DDATA= operand}

where the {...} definition is *optional* operand information defined by the setting of the Debug Module's Configuration/Status Register (CSR). The CSR provides capabilities to display M-Bus operands based on reference type (read, write, or both). Additionally, for certain change-of-flow branch instructions, another CSR field provides the capability to display {2,3,4} bytes of the target instruction address. For both situations, an optional PST value {8,9,B} provides the *marker* identifying the size and presence of valid data on the DDATA outputs.

C.1 USER INSTRUCTION SET

The **PST/DDATA** specification for the user-mode instructions is defined by the following list. Throughout the document, the "DD" nomenclature refers to the **DDATA** outputs:

add.l	<ea>y,Rx	PST = 1, {PST = B, DD = source operand}
add.l	Dy,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
addi.l	#imm,Dx	PST = 1
addq.l	#imm,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
addx.l	Dy,Dx	PST = 1
and.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
and.l	Dy,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
andi.l	#imm,Dx	PST = 1
asl.l	{Dy,#imm},Dx	PST = 1
asr.l	{Dy,#imm},Dx	PST = 1
bcc.{b,w}		if taken, then PST = 5, else PST = 1
bchg	Dy,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
bchg	#imm,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
bclr	Dy,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
bclr	#imm,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
bra.{b,w}		PST = 5
bset	Dy,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
bset	#imm,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}

bsr.{b,w}		PST = 5, {PST = B, DD = destination operand}
btst	Dy,<ea>x	PST = 1, {PST = 8, DD = source operand}
btst	#imm,<ea>x	PST = 1, {PST = 8, DD = source operand}
clr.b	<ea>x	PST = 1, {PST = 8, DD = destination operand}
clr.w	<ea>x	PST = 1, {PST = 9, DD = destination operand}
clr.l	<ea>x	PST = 1, {PST = B, DD = destination operand}
cmp.l	<ea>y,Rx	PST = 1, {PST = B, DD = source operand}
cmpi.l	#imm,Dx	PST = 1
divs.w	<ea>y,Dx	PST = 1, {PST = 9, DD = source operand}
divu.w	<ea>y,Dx	PST = 1, {PST = 9, DD = source operand}
divs.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
divu.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
eor.l	Dy,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
eori.l	#imm,Dx	PST = 1
ext.w	Dx	PST = 1
ext.l	Dx	PST = 1
extb.l	Dx	PST = 1
jmp	<ea>x	PST = 5, {PST = [9AB], DD = target address} (See Notes)
jsr	<ea>x	PST = 5, {PST = [9AB], DD = target address}, {PST = B, DD= destination operand} (See Notes)
lea	<ea>y,Ax	PST = 1
link.w	Ay,#imm	PST = 1, {PST = B, DD = destination operand}
lsl.l	{Dy,#imm},Dx	PST = 1
lsr.l	{Dy,#imm},Dx	PST = 1
mac.w	Ry,Rx	PST = 1
mac.l	Ry,Rx	PST = 1
mac.w	Ry,Rx,ea,Rw	PST = 1, {PST = B, DD = source operand}
mac.l	Ry,Rx,ea,Rw	PST = 1, {PST = B, DD = source operand}
move.b	<ea>y,<ea>x	PST = 1, {PST = 8, DD = source}, {PST = 8, DD = destination}
move.w	<ea>y,<ea>x	PST = 1, {PST = 9, DD = source}, {PST = 9, DD = destination}
move.l	<ea>y,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
move.l	<ea>y,ACC	PST = 1
move.l	<ea>y,MACSR	PST = 1
move.l	<ea>y,MASK	PST = 1
move.l	ACC,Rx	PST = 1
move.l	MACSR,CCR	PST = 1
move.l	MACSR,Rx	PST = 1
move.l	MASK,Rx	PST = 1
move.w	CCR,Dx	PST = 1
move.w	{Dy,#imm},CCR	PST = 1
movem.l	<ea>y,#list	PST = 1, {PST = B, DD = source},... (See Notes)
movem.l	#list,<ea>x	PST = 1, {PST = B, DD = destination},... (See Notes)

moveq	#imm,Dx	PST = 1
msac.w	Ry,Rx	PST = 1
msac.l	Ry,Rx	PST = 1
msac.w	Ry,Rx,ea,Rw	PST = 1, {PST = B, DD = source operand}
msac.l	Ry,Rx,ea,Rw	PST = 1, {PST = B, DD = source operand}
muls.w	<ea>y,Dx	PST = 1, {PST = 9, DD = source operand}
mulu.w	<ea>y,Dx	PST = 1, {PST = 9, DD = source operand}
muls.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
mulu.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
neg.l	Dx	PST = 1
negx.l	Dx	PST = 1
nop		PST = 1
not.l	Dx	PST = 1
or.l	<ea>y,Dx	PST = 1, {PST = B, DD = source operand}
or.l	Dy,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
ori.l	#imm,Dx	PST = 1
pea	<ea>y	PST = 1, {PST = B, DD = destination operand}
pulse		PST = 4
rems.l	<ea>y,Dx:Dw	PST = 1, {PST = B, DD = source operand}
remu.l	<ea>y,Dx:Dw	PST = 1, {PST = B, DD = source operand}
rts		PST = 1, {PST = B, DD = source operand},
		PST = 5, {PST = [9AB], DD = target address}
scc	Dx	PST = 1
sub.l	<ea>y,Rx	PST = 1, {PST = B, DD = source operand}
sub.l	Dy,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
subi.l	#imm,Dx	PST = 1
subq.l	#imm,<ea>x	PST = 1, {PST = B, DD = source}, {PST = B, DD = destination}
subx.l	Dy,Dx	PST = 1
swap	Dx	PST = 1
trap	#imm	PST = 1 (See Notes)
trapf		PST = 1
trapf.w		PST = 1
trapf.l		PST = 1
tst.b	<ea>x	PST = 1, {PST = 8, DD = source operand}
tst.w	<ea>x	PST = 1, {PST = 9, DD = source operand}
tst.l	<ea>x	PST = 1, {PST = B, DD = source operand}
unlk	Ax	PST = 1, {PST = B, DD = destination operand}
wddata.b	<ea>y	PST = 4, PST = 8, DD = source operand
wddata.w	<ea>y	PST = 4, PST = 9, DD = source operand
wddata.l	<ea>y	PST = 4, PST = B, DD = source operand

where Rn represents any {Dn, An} register. In this definition, the “y” suffix is generally used to denote the source operand and the “x” suffix is used for the destination operand. For any given instruction, the optional operand data is displayed only for those effective addresses referencing memory.

NOTES

1. For the JMP and JSR instructions, the optional target instruction address is only displayed 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).

2. For the 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.

3. During normal exception processing, the **PST** output is driven to a $C 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    PST = C, {PST = B, DD = destination},   // stack frame write
                                 {PST = B, DD = destination},   // stack frame write
                                 {PST = B, DD = source},        // vector read
                        PST = 5, {PST = [9AB], DD = target}     // PC of handler
```

The **PST/DDATA** specification for the *reset exception* is shown below:

```
Exception Processing    PST = C,
                        PST = 5, {PST = [9AB], DD = target}     // initial 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 **PST** = $C value is driven at all times, unless the **PST** output is needed for one of the optional marker values or the taken branch indicator ($5).

C.2 SUPERVISOR INSTRUCTION SET

The supervisor instruction set has complete access to the user mode instructions plus the opcodes shown below. The **PST/DDATA** specification for these opcodes is:

```
cpushl      (Ax)            PST = 1
halt                        PST = 1,  PST = F
move.w      SR,Dx           PST = 1
move.w      {Dy,#imm},SR    PST = 1, {PST = 3}
movec       Ry,Rc           PST = 1
rte                         PST = 7, {PST = B, DD = source operand}, {PST = 3},
```

		{PST = B, DD = source operand},
		PST = 5, {PST = [9AB], DD = target address}
stop	#imm	PST = 1, PST = E
wdebug	<ea>y	PST = 1, {PST = B, DD = source, PST = B, DD = source}

The move-to-SR and RTE instructions include an optional **PST** = $3 value, indicating an entry into user mode.

Similar to the exception processing mode, the stopped state (**PST** = $E) and the halted state (**PST** = $F) display the PST status throughout the entire time the ColdFire processor is in the given mode.

APPENDIX D
LOCAL MEMORY CONNECTIONS

This appendix provides an example Verilog file showing the instantiation of the processor-local memories (Unified Cache, RAM and ROM) and the appropriate connections with the CF3Core.

This example is defined with maximum-sized memories, i.e., a 32 KByte Unified Cache, a 32 KByte RAM and a 32 KByte ROM.

The memory array instantiations appear in the following order: Unified Cache tag storage, Unified Cache data storage, RAM storage, and finally, the ROM storage.

```
//************************************************************
//************************************************************
//
//  KBUS UNIFIED CACHE TAG & DATA ARRAY MEMORIES
//
//************************************************************
//************************************************************

//************************************************************
// Cache Tag SRAM arrays - 4 x sram512x25 = tags for a 32 KByte cache
//************************************************************

// LEVEL 3 TAG
sram512x25  ucTag0SramLevel3(
        .dbo            ({uctag3do[31:9], ucw3do, ucv3do}),

        .a              ( nsrowst[8:0]),
        .dbi            ({nsaddrt[31:9], nssw, nssv}),
        .csB            ( nsentb),
        .rwB            (~nswlvt[3]),
        .clk            ( clkfast) );

// LEVEL 2 TAG
sram512x25  ucTag0SramLevel2(
        .dbo            ({uctag2do[31:9], ucw2do, ucv2do}),
```

```
        .a              ( nsrowst[8:0]),
        .dbi            ({nsaddrt[31:9], nssw, nssv}),
        .csB            ( nsentb),
        .rwB            (~nswlvt[2]),
        .clk            ( clkfast) );

// LEVEL 1 TAG
sram512x25  ucTag0SramLevel1(
        .dbo            ({uctag1do[31:9], ucw1do, ucv1do}),

        .a              ( nsrowst[8:0]),
        .dbi            ({nsaddrt[31:9], nssw, nssv}),
        .csB            ( nsentb),
        .rwB            (~nswlvt[1]),
        .clk            ( clkfast) );

// LEVEL 0 TAG
sram512x25  ucTag0SramLevel0(
        .dbo            ({uctag0do[31:9], ucw0do, ucv0do}),

        .a              ( nsrowst[8:0]),
        .dbi            ({nsaddrt[31:9], nssw, nssv}),
        .csB            ( nsentb),
        .rwB            (~nswlvt[0]),
        .clk            ( clkfast) );

//*************************************************************
// Cache Data SRAM arrays - 4 x [4 x sram2048x8] = data for a 32
//                                                  KByte cache
//*************************************************************

// LEVEL 3 DATA
sram2048x8 ucData3SramByte0 (
        .dbo            ( uclvl3do[31:24]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[31:24]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[3] | ~nswtbyted[0]),
        .clk            ( clkfast) );
```

```
sram2048x8 ucData3SramByte1 (
        .dbo            ( uclvl3do[23:16]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[23:16]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[3] | ~nswtbyted[1]),
        .clk            ( clkfast) );

sram2048x8 ucData3SramByte2 (
        .dbo            ( uclvl3do[15:8]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[15:8]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[3] | ~nswtbyted[2]),
        .clk            ( clkfast) );

sram2048x8 ucData3SramByte3 (
        .dbo            ( uclvl3do[7:0]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[7:0]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[3] | ~nswtbyted[3]),
        .clk            ( clkfast) );

// LEVEL 2 DATA
sram2048x8 ucData2SramByte0 (
        .dbo            ( uclvl2do[31:24]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[31:24]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[2] | ~nswtbyted[0]),
        .clk            ( clkfast) );

sram2048x8 ucData2SramByte1 (
        .dbo            ( uclvl2do[23:16]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[23:16]),
```

```
        .csB            ( nsendb),
        .rwB            ( nswrtdb[2] | ~nswtbyted[1]),
        .clk            ( clkfast) );

sram2048x8 ucData2SramByte2 (
        .dbo            ( uclvl2do[15:8]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[15:8]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[2] | ~nswtbyted[2]),
        .clk            ( clkfast) );

sram2048x8 ucData2SramByte3 (
        .dbo            ( uclvl2do[7:0]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[7:0]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[2] | ~nswtbyted[3]),
        .clk            ( clkfast) );

// LEVEL 1 DATA
sram2048x8 ucData1SramByte0 (
        .dbo            ( uclvl1do[31:24]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[31:24]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[1] | ~nswtbyted[0]),
        .clk            ( clkfast) );

sram2048x8 ucData1SramByte1 (
        .dbo            ( uclvl1do[23:16]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[23:16]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[1] | ~nswtbyted[1]),
        .clk            ( clkfast) );

sram2048x8 ucData1SramByte2 (
```

```
        .dbo            ( uclvl1do[15:8]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[15:8]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[1] | ~nswtbyted[2]),
        .clk            ( clkfast) );

sram2048x8 ucData1SramByte3 (
        .dbo            ( uclvl1do[7:0]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[7:0]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[1] | ~nswtbyted[3]),
        .clk            ( clkfast) );

// LEVEL 0 DATA
sram2048x8 ucData0SramByte0 (
        .dbo            ( uclvl0do[31:24]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[31:24]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[0] | ~nswtbyted[0]),
        .clk            ( clkfast) );

sram2048x8 ucData0SramByte1 (
        .dbo            ( uclvl0do[23:16]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[23:16]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[0] | ~nswtbyted[1]),
        .clk            ( clkfast) );

sram2048x8 ucData0SramByte2 (
        .dbo            ( uclvl0do[15:8]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[15:8]),
        .csB            ( nsendb),
```

```
        .rwB            ( nswrtdb[0] | ~nswtbyted[2]),
        .clk            ( clkfast) );

sram2048x8 ucData0SramByte3 (
        .dbo            ( uclvl0do[7:0]),

        .a              ( nsrowsd[10:0]),
        .dbi            ( nscwrdata[7:0]),
        .csB            ( nsendb),
        .rwB            ( nswrtdb[0] | ~nswtbyted[3]),
        .clk            ( clkfast) );

//*****************************************************************
//*****************************************************************
//
// KBUS RANDOM ACCESS MEMORY
//
//*****************************************************************
//*****************************************************************

sram8192x8  KramByte0(
        .dbo            ( kramdo[31:24]),

        .a              ( kramaddr[14:2]),
        .dbi            ( kramdi[31:24]),
        .csB            ( kramcsb),
        .rwB            ( kramweb[0]),
        .clk            ( clkfast) );

sram8192x8  KramByte1 (
        .dbo            ( kramdo[23:16]),

        .a              ( kramaddr[14:2]),
        .dbi            ( kramdi[23:16]),
        .csB            ( kramcsb),
        .rwB            ( kramweb[1]),
        .clk            ( clkfast) );

sram8192x8  KramByte2 (
        .dbo            ( kramdo[15:8]),

        .a              ( kramaddr[14:2]),
```

```
        .dbi            ( kramdi[15:8]),
        .csB            ( kramcsb),
        .rwB            ( kramweb[2]),
        .clk            ( clkfast) );

sram8192x8  KramByte3 (
        .dbo            ( kramdo[7:0]),

        .a              ( kramaddr[14:2]),
        .dbi            ( kramdi[7:0]),
        .csB            ( kramcsb),
        .rwB            ( kramweb[3]),
        .clk            ( clkfast) );

//*****************************************************************
//*****************************************************************
//
// KBUS READ-ONLY MEMORY
//
//*****************************************************************
//*****************************************************************

rom8192x16  KromWord0 (
        .dbo            ( kromdo[31:16]),

        .a              ( kromaddr[14:2]),
        .csB            ( kromcsb),
        .clk            ( clkfast) );

rom8192x16  KromWord2 (
        .dbo            ( kromdo[15:0]),

        .a              ( kromaddr[14:2]),
        .csB            ( kromcsb),
        .clk            ( clkfast) );
```

INDEX

E

F

G

H

I

M

N

P

R

S

T

V

W