Freescale Semiconductor
White Paper

Document Number: V1CFWP
Rev. 0, 07/2006

Version 1 ColdFire® White Paper

1 Introduction

The microcontroller landscape is changing...

The bit boundaries that have traditionally defined MCU-based solutions are blurring as application performance in the 8-bit world is pushing migration upward at the same time reduced cost in the 32-bit segment is enabling increased access for these more-capable devices. Consider the following introductory statement from a recent *Microprocessor Report* article:

"Although Microprocessor Report focuses on higher-end CPUs whose microarchitectures continue to evolve, there's an interesting battle raging among microcontroller vendors. The semiconductor companies that make 32-bit MCUs—such as Atmel, Freescale, Oki, Philips, STMicroelectronics, and others—are fighting over the growing number of customers whose 8- and 16-bit CPUs are hitting a performance limit.

Contents

1 Introduction 1
1.1 Overview 4
1.2 Features 9
1.3 Modes of Operation 10
2 Memory Map and Register Definition 11
2.1 Memory Map 11
2.2 Register Descriptions 12
3 Instruction Set Architecture (ISA_C) 18
4 Exception Processing Overview 19
4.1 Exception Stack Frame Definition 22
4.2 S08 and ColdFire Exception Processing Comparison 23
4.3 Processor Exceptions 24
5 Debug Overview 32
5.1 Debug Register Descriptions 32
5.2 Background Debug Mode (BDM) 34
5.3 Processor Status (PST)/Debug Data (DDATA) and Trace Support 42
6 V1 ColdFire Core Hardware Details 52
6.1 Core Microarchitecture 52
6.2 Power Dissipation 65
6.3 Code Density and Performance 66

© Freescale Semiconductor, Inc., 2006. All rights reserved.

freescale™ semiconductor

Often overlooked by the microprocessor cognoscenti, the MCU market is enormous: more than $12 billion in 2005. Of this, more than $10 billion consisted of 8- and 16-bit devices. However, the relative size and power consumption of 32-bit MCUs have dropped dramatically in recent years, so 32-bit vendors are vigorously attempting to lure system designers away from the smaller devices." ["Freely Scaling from 8 to 32 Bits," J. Scott Gardner, *Microprocessor Report*, May 1, 2006]

The Freescale Controller Continuum outlines a roadmap that will provide pin-for-pin compatible 8-bit and 32-bit devices that share peripherals and a common set of tools.

Demand for increased performance and functionality continues to push designs up the microcontroller food chain. For 8-bit veterans, Freescale plans expansion at the high-end of the 8-bit portfolio with enhanced peripherals and expanded memory options.

Freescale is continuing to make the transition to 32-bit easier through entry-level ColdFire microcontrollers with connectivity and security features for cost-sensitive applications. Freescale plans to roll out pin-for-pin compatible devices that will allow 8-bit designs to easily upgrade to 32-bit performance while maintaining the same peripheral interfaces during 2007.

Additionally, Freescale plans continued enhancements to its award-winning CodeWarrior™ Development Studio with automatic code generation, which enables first-time users to create working projects in as few as seven clicks. Freescale is also defining a unified hardware development platform that will provide common board and cable interfaces across architectures. See Figure 1.

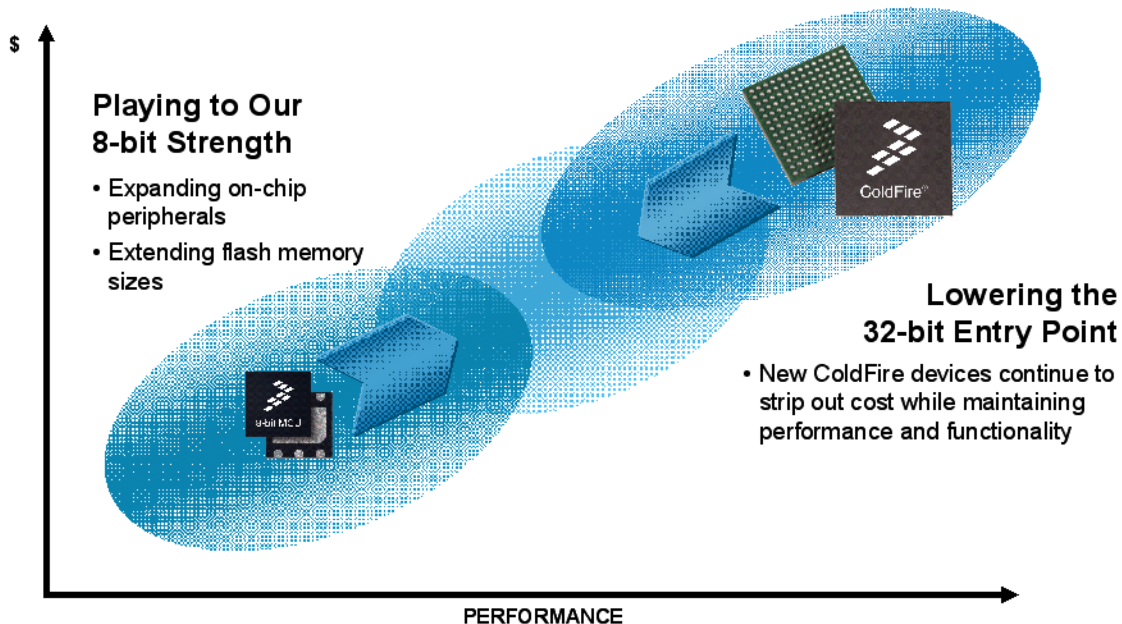


Figure 1. Freescale's Controller Continuum

Critical to the controller continuum and the creation of the industry's first 8/32-bit compatible architectures is the Version 1 ColdFire core. As shown in Figure 2, the V1 ColdFire core can be viewed as the missing link bridging the continuum, providing 32-bit performance with 8-bit ease of use with devices that are peripheral set and pin compatible with their 8-bit "little brothers".

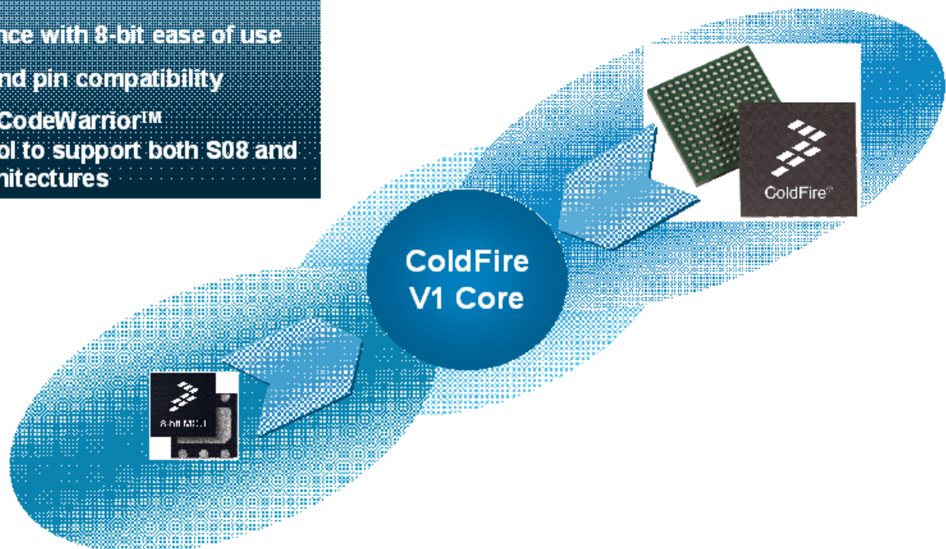


Figure 2. V1 ColdFire: Controller Continuum Missing Link

With the V1 ColdFire core providing the link between the 8- and 32-bit worlds, migration to 32-bits is greatly simplified as shown in Figure 3.

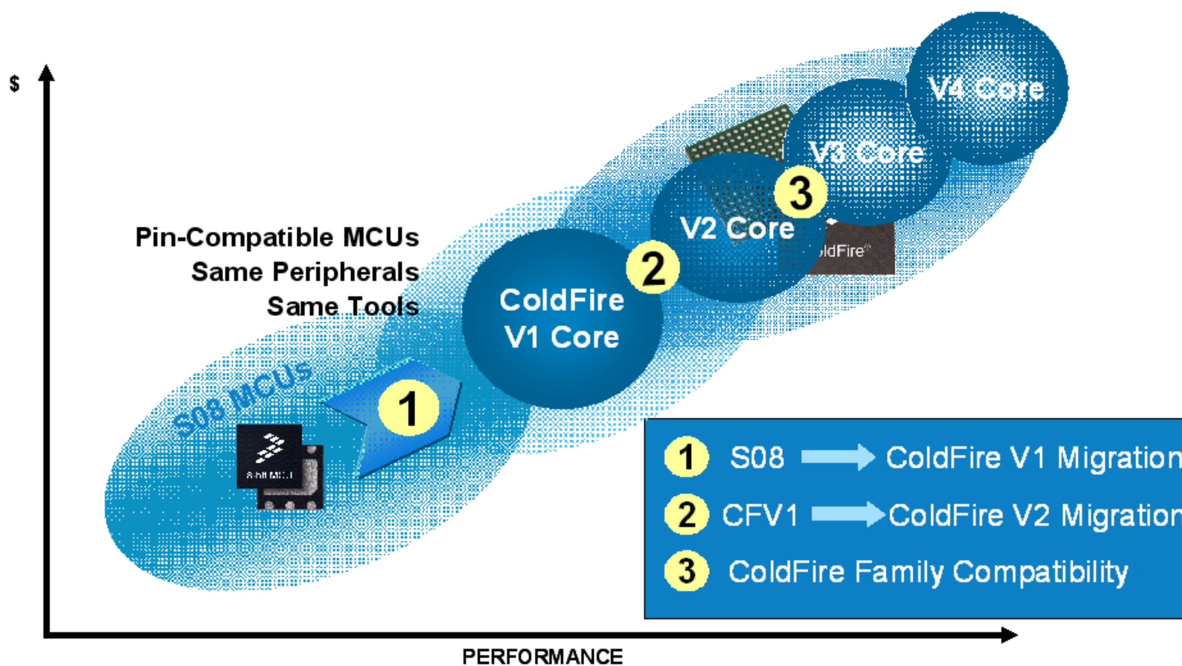


Figure 3. Migration to 32-bit: As Easy As 1-2-3

The remainder of this document provides technical details on the V1 ColdFire core and platform, covering its instruction set architecture and debug architecture as well as information on the processor and platform microarchitecture.

1.1 Overview

In the creation of the V1 ColdFire core as the missing link in Freescale's Controller Continuum, the following principles guided the development:

- The design followed strict priorities when making core architecture and implementation trade-offs
  - Minimal size
  - Minimal power dissipation
  - Maximum performance
- The peripheral set and pin compatibility with S08 devices had significant impacts on both the core and debug architectural definitions
- A simplified realization of the V2 ColdFire core pipeline was viewed as the appropriate microarchitecture foundation upon which to build the V1 core

1.1.1 Instruction Set Architecture

To begin, consider the peripheral set compatibility with S08 devices. Since most of the slave devices are 8-bit peripherals, support for Revision C of the ColdFire instruction set architecture (ISA_C) was specified. ISA_C is the most current ColdFire definition and, in particular, includes a number of instructions added as part of the incremental changes to ISA_B and ISA_C that improve the handling of 8- and 16-bit data values.

The original definition of the ColdFire Instruction Set Architecture (ISA_A) was derived from M68K Family opcodes based on extensive analysis of embedded application code. The ISA was optimized for code compiled from high-level languages where the dominant operand size was the 32-bit `int` type. This approach minimized processor complexity and cost, while providing excellent performance for compiled applications.

After the initial ColdFire compilers were created, developers noted there were certain ISA additions that would enhance both code density as well as overall performance. Additionally, as users implemented ColdFire-based designs into a wide range of embedded systems, they noted certain frequently-used instruction sequences that could be improved by the creation of new instructions.

The original ISA definition minimized the support for instructions referencing byte- and word-sized operands compared to the M68K family. Full support for the move byte (MOVE.B) and move word (MOVE.W) instructions was provided, but the only other opcodes supporting these data types were clear (CLR) and test (TST). Based on the input from compiler writers and users, the ColdFire ISA has been improved with the ISA_B and ISA_C extensions. These expanded ISA definitions have improved performance and code density in three areas:

1. Enhanced support for byte and word-sized operands through added move and compare operations
2. Enhanced support for position-independent code
3. Improved support for certain types of bit manipulation operators

Accordingly, the ISA specification for the V1 ColdFire core is defined as ISA_C since code size and performance associated with byte- and word-sized references to the S08 peripheral set is such an important

factor. For the specifics of the ISA_C definition, see Chapter 3 of the *ColdFire Programmer's Reference Manual*, especially the instruction set cross-reference presented in Table 3-16.

It should also be noted that careful consideration and study was given to the possibility of supporting a reduced user programming model for the V1 core. In particular, a proposal to reduce the number of general-purpose registers (R*n*) from 16 to 8 was given serious consideration. In this proposal, the number of address (A*n*) and data (D*n*) registers was halved, so that the V1 core would only support the {D0, D1, D6, D7} and {A0, A1, A6, A7} registers. This proposal was driven by the fact that the register file is the largest single structure in the core and a sizable reduction in this function could have an interesting impact on the overall core size. However, in the final analysis it was decided that code compatibility across the entire ColdFire family and the ability to reuse the existing development tools (compilers, debuggers, etc.) was more important than the gate savings achievable through this program-visible redefinition of the register set.

1.1.2 Pin Compatibility with HCS08 Devices

Next, consider the impact of pin compatibility with HCS08 devices. This requirement has significant ramifications on the ColdFire debug architecture, primarily in two distinct areas.

The "classic" Version 2 ColdFire debug architecture specifies a 12-pin interface: 3 pins for the serial background debug mode (BDM) communication channel (clock, data-in, data-out), 1 pin for a breakpoint function, and 8 pins associated with the processor status (PST) and debug data (DDATA) outputs that provide real-time instruction (and optional data) trace capabilities. In contrast, the HCS08 family of devices support all their debug functionality via a single-pin BDM interface. As expected, this reduced interface significantly affects the V1 debug definition. In particular, a single-pin BDM interface for the V1 core means:

- The traditional 3-pin full-duplex BDM serial communication protocol is replaced with the HCS08 single-pin protocol. The classic ColdFire BDM mechanism is based on 17-bit full-duplex packet protocol. For V1, the HCS08 protocol is fully adopted where all BDM communications utilize a basic 8-bit data packet over a single package pin (BKGD).
- The reduced debug package pin count also requires that the classic ColdFire program and data trace capabilities be completely redefined for the V1 core. The resulting implementation retains the basic notions of encoded processor status and captured debug data but introduces the notion of PST compression and recording the resulting stream of PST/DDATA values in an on-chip trace buffer. Once recorded in the debug module's trace buffer, the contents can be accessed as BDM-visible registers so the data can be post-processed to provide the needed trace information. Support for the WDDATA instruction, which allows the application code to write directly to the DDATA port, is retained as this provides a simple hardware "printf" capability.

Overall, this approach is significant since it enables a common development environment where the same tools (including the USB cables from the development system to the microcontroller) can be used for both HCS08 or V1 ColdFire devices.

1.1.3 Core Pipeline Organization

As previously noted, the pipeline organization and structure from the V2 core is significantly leveraged for the V1 implementation. The resulting V1 ColdFire processor core pipeline features two independent, decoupled pipeline structures with a unified bus interface to maximize performance while minimizing core size to strip out cost.

The instruction fetch pipeline (IFP) is a 2-stage pipeline for pre-fetching instructions. The pre-fetched instruction stream is then gated into the 2-stage operand execution pipeline (OEP), which decodes the instruction, fetches the required operands and then executes the required function. Since the IFP and OEP pipelines are decoupled by an instruction buffer which serves as a FIFO queue, the IFP is able to pre-fetch instructions in advance of their actual use by the OEP, thereby minimizing time stalled waiting for instructions.

The instruction fetch pipeline consists of two stages with an optional instruction buffer stage:

–Instruction Address Generation	IAG Cycle
–Instruction Fetch Cycle	IC Cycle
–Instruction Buffer	IB Cycle

When the instruction buffer is empty, opcodes are loaded directly from the IC cycle into the operand execution pipeline. If the buffer is not empty, the IFP stores the contents of the fetch cycle in the FIFO queue until it is required by the OEP. In the Version 1 implementation, the instruction buffer contains three 32-bit longwords of storage.

The operand execution pipeline is implemented in a two-stage pipeline featuring a traditional RISC datapath with a dual-read-ported register file (RGF) feeding an arithmetic/logic unit. In this design, the pipeline stages have multiple functions:

–Decode & Select/Operand Cycle	DSOC Cycle
–Address Generation/Execute Cycle	AGEX Cycle

In an effort to minimize gate cost, the address path is reduced from 32 to 24 bits, sufficient for a 16-Mbyte memory map for the V1 devices. Additionally, the V1 core includes design parameters to easily include/exclude various “execute engines” associated with specific instructions. These optional execute engines include an integer divider (DIV), {E}MAC multiply-accumulate engines, and a cryptographic acceleration unit (CAU).

The V1 OEP extends a pipeline optimization associated with 32-bit move (load) instructions to also apply to 8- and 16-bit moves, including the move-with-zero-fill and move-with-sign-extension instructions. The result is improved 2-cycle performance on this very important class of load instructions:

```
move.b   <mem>y,Dx         // byte load
mvs.b    <mem>y,Dx         // byte load with sign-extension
mvz.b    <mem>y,Dx         // byte load with zero-fill
move.w   <mem>y,Rx         // word load
mvs.w    <mem>y,Dx         // word load with sign-extension
mvz.w    <mem>y,Dx         // word load with zero-fill
move.l   <mem>y,Rx         // long word
```

where the source operand is memory, defined by the effective address <mem>y, and the destination is one of the general-purpose registers (Dx is a data register, Rx is either a data or address register). The `TST.sz <mem>y` instructions also have this same 2-cycle execution time.

The resulting high-level block diagram of the V1 ColdFire core and the low-cost platform is shown in Figure 4. In this diagram, the following platform modules are included:

- V1 ColdFire CPU with optional execute engines plus the background debug controller (BDC) and debug module
- Local bus controller
- Platform flash memory controller
- Platform RAM memory controller
- Platform-to-peripheral bus controller
- Rapid GPIO (RGPIO) controller

The two platform memory controllers connect to off-platform memory arrays and each supports a wide range of storage capacities. The RGPIO controller connects to chip-level I/O pads and provides high-speed general purpose input/output functionality, since this module is connected to the core's high-speed platform bus. The peripheral bus bridge provides two functions. First, it converts the platform's system bus protocol into the simpler protocol for the slave peripheral modules. Second, it serves as the clock domain boundary, separating the core platform's high-speed domain from the slower peripheral domain. The peripheral bridge supports any *n*:1 clock ratio, where *n* = 1,2,3, ... ; for most device implementations, the core platform-to-peripheral speed ratio is fixed at 2:1.

Finally, as shown in Figure 4, the simplest CF1 low-cost platform includes a single bus master, the V1 core. It should also be noted that the low-cost core platform has been constructed to support a second bus mastering device for future SoC definitions. In particular, future V1 devices with other bus master devices with built-in DMA functionality are planned.

In addition to these platform modules, there is an interrupt controller (INTC) which is an off-platform slave module. Because its functionality is important to the core's operation, the features of the INTC are also detailed in this document.

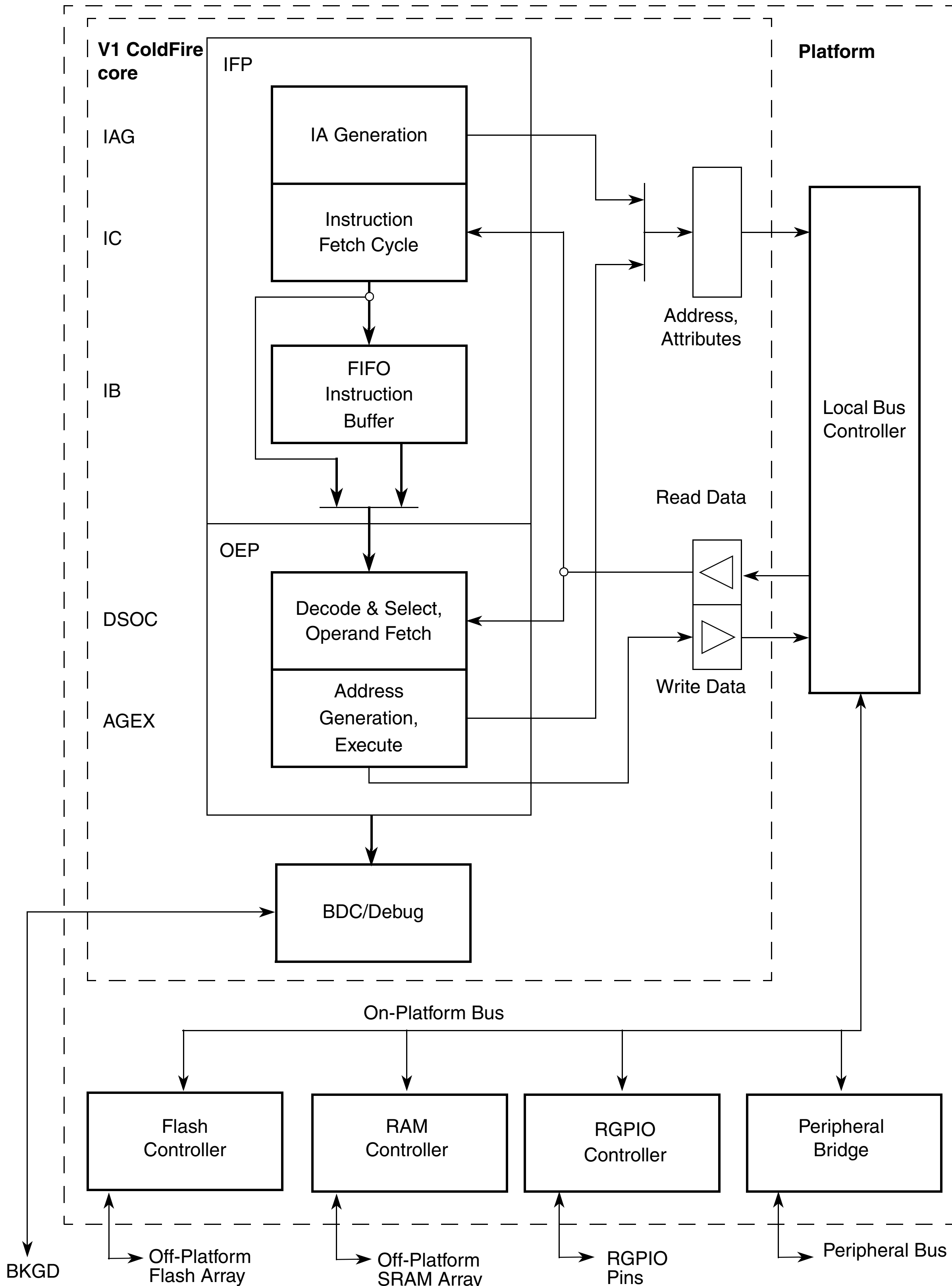


Figure 4. CF1 Low-Cost Platform (cf1_lcp) Block Diagram

1.2 Features

The Version 1 ColdFire core and platform include the following features:

- Version 1 ColdFire processor core
  - Variable-length RISC architecture and implementation
  - Supports Revision C of the ColdFire Instruction Set Architecture
  - Supports the standard ColdFire user programming model with 16 general-purpose, 32-bit data and address registers
  - Simplified supervisor programming model supporting a supervisor stack pointer, vector base register, and CPU configuration register
  - Support for optional execute engines including integer divider, multiply-accumulate units (MAC, EMAC) and cryptography acceleration unit (CAU)
  - Programmable response upon detection of certain illegal opcodes and illegal addresses (processor exception or system reset)
  - Implementation contains 24-bit address and 32-bit data paths
  - Up to 50 MHz core frequency in a low-voltage, low-power 0.25-micron process technology
  - 0.85 Dhrystone 2.1 MIPS per MHz performance when executing from flash, 1.05 DMIPS when executing from RAM
  - Aggressive clock gating for reducing power dissipation
- Version 1 ColdFire debug support
  - Classic ColdFire Debug B+ functionality mapped into the single-pin BDM interface
  - Capture of compressed processor status and debug data into on-chip trace buffer provides program (and optional slave bus data) trace capabilities
  - On-chip trace buffer provides programmable start/stop recording conditions plus support for continuous or PC-profiling modes
  - Real time debug support, with 6 hardware breakpoints (four PC, one address, and one data) that can be configured into a 1- or 2-level trigger with a programmable response (processor halt or interrupt)
  - Debug resources are accessible via single-pin BDM interface or the privileged WDEBUG instruction from the core
- Platform memories
  - Low-power 2-cycle flash memory for text and read-only data storage
    - Flash controller implements address speculation features to reduce effective access time
  - Single-cycle RAM on platform’s internal high-speed bus
    - Write buffer supports zero wait-state response, but does require a dummy write to push contents into RAM before entering any low-power mode
- High-speed GPIO
  - Up to 16 bits of rapid GPIO connected to platform’s bus for high-speed bit-banging
  - Single-cycle accesses with ColdFire core
  - Set, clear, toggle registers allow manipulation using simple store instructions

- Peripheral Bridge
  - — Converts from platform's system bus to external bus for off-platform slave devices
    - – Provides a standardized connection scheme for off-platform peripherals
    - – Support for byte- and word-sized accesses to peripheral space
  - — Programmable control for buffered writes for improved system performance
  - — Provides clock domain boundary from high-speed core platform to slower speed slave domain
    - – Support for *n*:1 platform:peripheral speed ratio, where *n* = 1,2,3, ... (typically fixed at 2:1 speed ratio)
- Interrupt Controller
  - — Peripheral-mapped off-platform slave module
    - – 64 byte space located at top end of memory: 0xFF_FFC0–0xFF_FFFF
    - – Programming model accessed via peripheral space
    - – Encoded interrupt level and vector sent directly to processor core
  - — Initial support of 30 peripheral I/O interrupt requests plus 7 software interrupt requests
  - — Fixed association between interrupt request source and level + priority
    - – 30 I/O requests assigned across seven available levels and nine priorities per level
    - – Exactly matches S08 interrupt request priorities
    - – Up to two requests can be re-mapped to the highest non-maskable level + priority
  - — Unique vector number for each interrupt source
    - – Default defined as: `CF_Vector_# = S08_Vector_# + 62`
  - — Support for service routine interrupt acknowledge (IACK) read cycles for improved system performance
  - — Combinatorial path provides wake-up signal from low-power sleep modes

1.3 Modes of Operation

The ColdFire Family programming model consists of two register groups, corresponding to the processor privilege level: user and supervisor. Applications executing in the user mode use only the programming model registers in the user group. System software executing in the supervisor mode can access all registers and use the control registers in the supervisor group to perform supervisory functions.

In addition, the V1 core supports the standard ColdFire definition of operating modes, where the processor can be running, stopped, or halted. The processor operating mode status is visible to the external world via the single-pin BDM interface by accessing one of the debug module's configuration/status registers. In particular, ColdFire cores source two bits which define the operating mode, both as output signals as well as program-visible status. These bits are *cpu_is_stopped* and *cpu_is_halted* and the operating mode is defined as:

Table 1. ColdFire Core Operating Mode Definition

Core Mode	cpu_is_stopped	cpu_is_halted
Running	0	0
Stopped	1	0
Halted	0	1

2 Memory Map and Register Definition

This section defines the standardized V1 system memory map and provides an overview of the processor and debug module programming models.

2.1 Memory Map

The definition of the system memory map is an important consideration as the platform's local bus controller must be able to quickly route requests to the appropriate slave destination. As the V1 core (or other masters) generate platform bus requests, the local bus controller must be able to very quickly identify the targeted slave so that the request can be properly evaluated and routed to the appropriate device. The net result is the address steering mechanism must be very simple and very fast, since any incremental delay in this function adds directly to the critical timing paths of the core platform design.

Each bus mastering module includes a connection to the input side of the local bus controller for initiating platform bus transactions. As shown in Figure 4, the core platform includes a number of local controllers as well as a bridge controller to the standardized peripheral bus. The slave interface is typically used to access the program-visible registers of the off-platform peripherals.

The platform bus is a 2-stage pipelined protocol, where the bus master drives the address and attribute signals during the first stage (the address phase) and the bus slave responds with data and termination signals in the second stage (the data phase). A single data-phase termination signal is negated to insert wait-states into the slave response, and effectively stall both stages of the bus pipeline. Conversely, the peripheral bus is a simple non-pipelined protocol, where address, attributes, data and termination signals are all valid within a machine cycle.

Recall the memory space defined by the V1 core uses a 24-bit address, providing support for a 16-MByte definition. The resulting Version 1 ColdFire core system memory map is shown in Table 2.

Table 2. V1 ColdFire System Memory Map

Address Range	Destination Slave	Slave Region Size
0x00_0000 – 0x7F_FFFF	Platform flash	8 Mbytes
0x80_0000 – 0xBF_FFFF	Platform RAM	4 Mbytes
0xC0_0000 – 0xDF_FFFF	Platform modules, e.g., RGPIO	2 Mbytes
0xE0_0000 – 0xFF_7FFF	Reserved	2 Mbytes – 32 Kbytes
0xFF_8000 – 0xFF_FFFF	Off-platform Peripheral Modules	32 Kbytes

Since the physical memory space is often considerably smaller than the allocated space, the actual memory space begins at the lower (starting) address, and creates memory holes in the 16 Mbyte definition. For example, a 128 Kbyte flash resides at address range 0x00_0000–0x01_FFFF and an 8 Kbyte RAM resides at 0x80_0000–0x80_1FFF. In general, attempted accesses to memory holes generate error terminations and the bus cycle is aborted.

The allowable access types to the different regions in the system memory map also vary. See Table 3.

Table 3. Supported Access Types by Region

Base Address	Region	Read			Write		
		Byte	Word	Long	Byte	Word	Long
0x00_0000	Flash	x	x	x	—	—	x
0x80_0000	RAM	x	x	x	x	x	x
0xC0_0000	RGPIO	x	x	x	x	x	x
0xFF_8000	Peripherals	x	x	—	x	x	—

Bus cycles which attempt to access a region with an unsupported reference size are terminated with an error.

The use of certain operand addressing modes by the V1 core was also considered during the definition of the system memory map. Specifically, it is suggested that the short-A5-relative addressing mode {addressing register indirect with a 16-bit signed displacement: d16(A5)} be used for data accesses to the RAM. This can often be enabled with a small data area compiler option. It is also suggested that the use of the 16-bit absolute short addressing mode (XXX.W) be used for peripheral accesses. With both suggestions utilized, the resulting code image produces best code density and highest performance.

2.2 Register Descriptions

The following sections describe the V1 ColdFire processor core registers in the user and supervisor programming models. The appropriate programming model is selected based on the privilege level (user mode or supervisor mode) of the processor as defined by the S bit of the status register (SR).

The user programming model is the same as the M68000 family microprocessors, consisting of the following registers:

- 16 general-purpose 32-bit registers (D0–D7, A0–A7)
- 32-bit program counter (PC)
- 8-bit condition code register (CCR)

The supervisor programming model is intended to be used only by system control software to implement restricted operating system functions, I/O control, and memory management. All accesses that affect the control features of ColdFire processors are in the supervisor programming model, which consists of registers available in user mode as well as the following control registers:

- 16-bit status register (SR)
- 32-bit supervisor stack pointer (SSP)
- 32-bit vector base register (VBR)

- 32-bit CPU configuration register (CPUCR)

Table 4 provides a summary of the V1 CPU programming model, including the BDM command to access each register.

Table 4. Version 1 ColdFire CPU Programming Model

BDM Command[1]	Register	Width (bits)	Access	Reset Value	Written with MOVEC[2]	Section/Page
Supervisor/User Registers						
Read: 0x60 Write: 0x40	Data Register 0 (D0)	32	R/W	0xCF1*_**29	No	2.2.1/-13
Read: 0x61 Write: 0x41	Data Register 1 (D1)	32	R/W	0x010*0_10*0	No	2.2.1/-13
Read: 0x6{2–7} Write: 0x4{2–7}	Data Register 2–7 (D2–D7)	32	R/W	Undefined	No	2.2.1/-13
Read: 0x6{8–E} Write: 0x4{8–E}	Address Register 0–6 (A0–A6)	32	R/W	Undefined	No	2.2.2/-14
Read: 0x6F Write: 0x4F	Supervisor/User A7 Stack Pointer (A7)	32	R/W	Undefined	No	2.2.3/-14
Read: 0xEF Write: 0xCF	Program Counter (PC)	32	R/W	Contents of memory @ 0x00_0004	No	2.2.4/-15
Read: 0xEE Write: 0xCE	Condition Code Register (CCR)	8	R/W	Undefined	No	2.2.5/-15
Supervisor Registers						
Read: 0xEE Write: 0xCE	Status Register (SR)	16	R/W	0x27--	No	2.2.6/-16
Read: 0xE0 Write: 0xC0	User/Supervisor A7 Stack Pointer (OTHER_A7)	32	R/W	Contents of memory @ 0x00_0000	No	2.2.3/-14
Read: 0xE1 Write: 0xC1	Vector Base Register (VBR)	32	W	0x0000_0000	Yes; Rc = 0x801	2.2.7/-17
Read: 0xE2 Write: 0xC2	CPU Configuration Register (CPUCR)	32	W	0x0000_0000	Yes; Rc = 0x802	2.2.8/-17

[1] The values listed in this column represent the 8-bit BDM command code used when accessing core registers via the1-pin port. For more information see Section 5.2, “Background Debug Mode (BDM).”

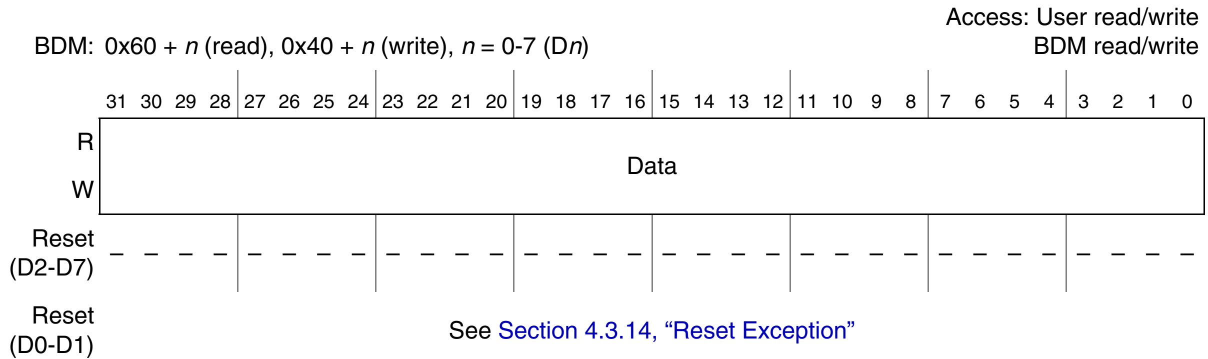
[2] If the given register is written using the MOVEC instruction, the 12-bit control register address (Rc) is also specified.

2.2.1 Data Registers (D0–D7)

Registers D0–D7 are used as data registers for bit (1-bit), byte (8-bit), word (16-bit) and longword (32-bit) operations; they can also be used as index registers.

NOTE

Registers D0 and D1 contain hardware configuration details after reset. See Section 4.3.14, “Reset Exception,” for more details.



2.2.2 Address Registers (A0–A6)

These registers can be used as software stack pointers, index registers, or base address registers; they can also be used for word and longword operations.

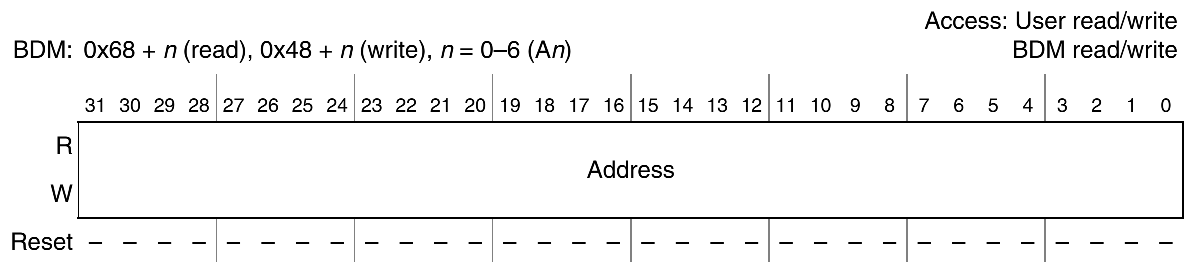


Figure 5. Address Registers (A0–A6)

2.2.3 Supervisor/User Stack Pointers (A7 and OTHER_A7)

The ColdFire architecture supports two independent stack pointer (A7) registers—the supervisor stack pointer (SSP) and the user stack pointer (USP). The hardware implementation of these two programmable-visible 32-bit registers does not identify one as the SSP and the other as the USP. Instead, the hardware uses one 32-bit register as the active A7 and the other as OTHER_A7. Thus, the register contents are a function of the processor operation mode, as shown in the following:

```
if SR[S] = 1
         then     A7       = Supervisor Stack Pointer
                  OTHER_A7 = User Stack Pointer
         else     A7       = User Stack Pointer
                  OTHER_A7 = Supervisor Stack Pointer
```

The BDM programming model supports direct reads and writes to A7 and OTHER_A7. It is the responsibility of the external development system to determine, based on the setting of SR[S], the mapping of A7 and OTHER_A7 to the two program-visible definitions (SSP and USP).

To support the dual stack pointers, the following two supervisor instructions are included in the ColdFire instruction set architecture to load/store the USP:

```
move.l Ay, USP; move to USP
move.l USP, Ax; move from USP
```

These instructions are described in the *ColdFire Family Programmer's Reference Manual*.

NOTE

The USP must be initialized using the `move.l Ay,USP` instruction before any entry into user mode. The SSP is loaded during reset exception processing with the contents of location 0x00_0000.

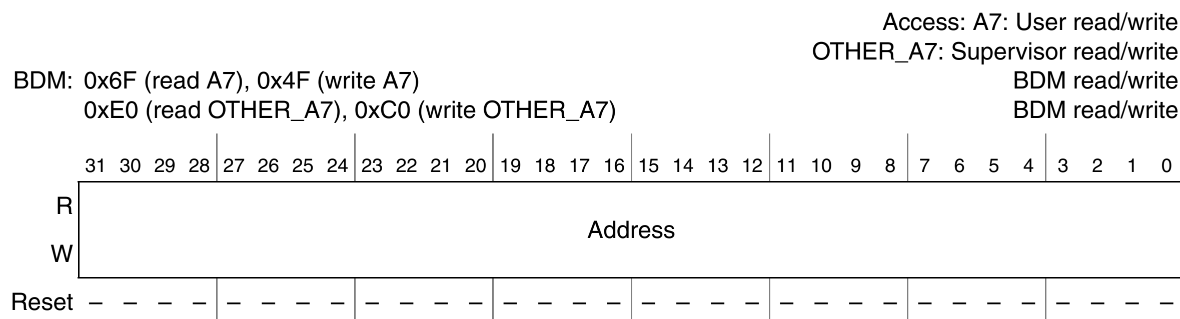


Figure 6. Stack Pointer Registers (A7 and OTHER_A7)

2.2.4 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. The PC is used as a base address for PC-relative operand addressing.

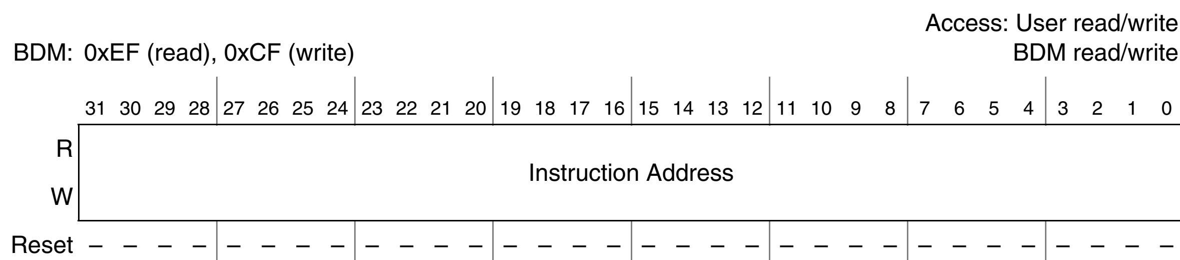


Figure 7. Program Counter Register (PC)

The PC is initially loaded during reset exception processing with the contents of location 0x00_0004.

2.2.5 Condition Code Register (CCR)

The CCR is the LSB of the processor status register (SR). Bits 4–0 act as indicator flags for results generated by processor operations. The extend bit (X) is also used as an input operand during multiprecision arithmetic computations. The CCR register must be explicitly loaded after reset and before any compare (CMP), Bcc, or Scc instructions are executed.

BDM: 0xEE (SR read), 0xCE (SR write)

Access: User read/write
BDM read/write

	7	6	5	4	3	2	1	0
R	0	0	0	X	N	Z	V	C
W								
Reset:	0	0	0	—	—	—	—	—

Table 5. CCR Field Descriptions

Field	Description
7–5	Reserved, should be cleared.
4 X	Extend condition code bit. Set to the value of the C-bit for arithmetic operations; otherwise not affected or set to a specified result.
3 N	Negative condition code bit. Set if the most significant bit of the result is set; otherwise cleared.
2 Z	Zero condition code bit. Set if the result equals zero; otherwise cleared.
1 V	Overflow condition code bit. Set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise cleared.
0 C	Carry condition code bit. Set if a carry out of the operand msb occurs for an addition, or if a borrow occurs in a subtraction; otherwise cleared

2.2.6 Status Register (SR)

The SR stores the processor status and includes the CCR, the interrupt priority mask, and other control bits. In supervisor mode, software can access the entire SR. In user mode, only the lower 8 bits are accessible (CCR). The control bits indicate these states for the processor: trace mode (T bit), supervisor or user mode (S bit), and master or interrupt state (M bit). All defined bits in the SR have read/write access when in supervisor mode. The SR register (actually the CCR) must be explicitly loaded after reset and before any compare (CMP), Bcc, or Scc instructions are executed.

BDM: 0xee (read), 0xce (write)

Access: Supervisor read/write
BDM read/write

	System Byte								Condition Code Register (CCR)							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	T	0	S	M	0	I			0	0	0	X	N	Z	V	C
W																
Reset	0	0	1	0	0	1	1	1	0	0	0	—	—	—	—	—

Table 6. SR Field Descriptions

Field	Description
15 T	Trace enable. When set, the processor performs a trace exception after every instruction.
14	Reserved, should be cleared.
13 S	Supervisor/user state. 0 User mode 1 Supervisor mode
12 M	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.
11	Reserved, should be cleared.
10–8 I	Interrupt level mask. Defines the current interrupt level. Interrupt requests are inhibited for all priority levels less than or equal to the current level, except the edge-sensitive level 7 request, which cannot be masked.
7–0 CCR	Refer to Section 2.2.5, “Condition Code Register (CCR).”

2.2.7 Vector Base Register (VBR)

The VBR contains the base address of the exception vector table in memory. To access the vector table, the displacement of an exception vector is added to the value in VBR. The lower 20 bits of the VBR are not implemented by any ColdFire processors; they are assumed to be zero, forcing the table to be aligned on a 1-MByte boundary. In addition, since the V1 ColdFire core supports a 16 Mbyte address space, the upper byte of the VBR is also forced to zero. The VBR can be used to relocate the exception vector table from its default position in the flash memory (address 0x00_0000) to the base of the RAM (address 0x80_0000) if needed.

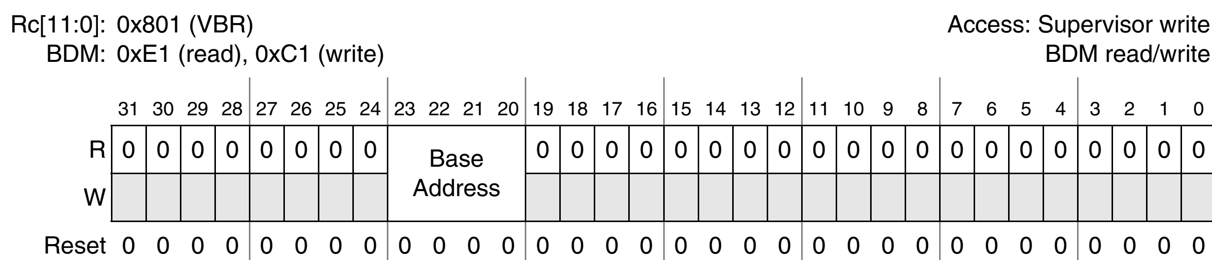


Figure 8. Vector Base Register (VBR)

2.2.8 CPU Configuration Register (CPUCR)

The CPUCR provides supervisor mode configurability of specific core functionality. Certain hardware features can be enabled/disabled individually based on the state of the CPUCR.

Rc[11:0]: 0x802 (CPUCR)
BDM: 0xE2 (read), 0xC2 (write)

Access: Supervisor write
BDM read/write

	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
R	ARD	IRD	IAE	0	BWD	0	FSD	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9. CPU Configuration Register (CPUCR)

Table 7. CPUCR Field Descriptions

Field	Description
31 ARD	Address-related reset disable. Used to disable the generation of a reset event in response to a processor exception caused by an address error, a bus error, an RTE format error or a fault-on-fault halt condition. 0 The detection of these types of exception conditions or the fault-on-fault halt condition generate a reset event. 1 No reset is generated in response to these exception conditions.
30 IRD	Instruction-related reset disable. Used to disable the generation of a reset event in response to a processor exception caused by the attempted execution of an illegal instruction (except for the ILLEGAL opcode), illegal line A, illegal line F instructions, or a privilege violation. 0 The detection of these types of exception conditions generate a reset event. 1 No reset is generated in response to these exception conditions.
29 IAE	Interrupt acknowledge (IACK) enable. Forces the processor to generate an IACK read cycle from the interrupt controller during exception processing to retrieve the vector number of the interrupt request being acknowledged. The processor's execution time for an interrupt exception is slightly improved when this bit is cleared. 0 The processor uses the vector number provided by the interrupt controller at the time the request is signaled. 1 IACK read cycle from the interrupt controller is generated.
28	Reserved, should be cleared.
27 BWD	Buffered peripheral bus write disable. 0 Peripheral bus writes are buffered and the platform bus cycle is terminated immediately and does not wait for the peripheral bus termination status. 1 Disables the buffering of peripheral bus writes and does not terminate the platform bus cycle until a registered version of the peripheral bus termination is received. **Note:** If buffered writes are enabled (BWD = 0), any peripheral bus error status is lost.
26	Reserved, should be cleared.
25 FSD	Flash speculation disabled. Disables certain performance-enhancing features related to address speculation in the platform flash memory controller. 0 The platform flash controller tries to speculate on read accesses to improve processor performance by minimizing the exposed flash memory access time. Recall the basic flash access time is two processor cycles. 1 Certain flash address speculation is disabled.
24–0	Reserved, should be cleared.

3 Instruction Set Architecture (ISA_C)

The original ColdFire instruction set architecture (ISA) was derived from the M68000-family opcodes based on extensive analysis of embedded application code. After the initial ColdFire compilers were created, developers identified ISA additions that would enhance both code density and overall performance. Additionally, as users implemented ColdFire-based designs into a wide range of embedded

systems, they identified frequently used instruction sequences that could be improved by the creation of new instructions. This observation was especially prevalent in development environments that made use of substantial amounts of assembly language code.

Table 8 summarizes the instructions added to revision ISA_A to form revision ISA_C. For more details see the *ColdFire Family Programmer's Reference Manual*.

Table 8. Instruction Enhancements over Revision ISA_A

Instruction	Description
BITREV	The contents of the destination data register are bit-reversed; that is, new Dn[31] = old Dn[0], new Dn[30] = old Dn[1], ..., new Dn[0] = old Dn[31].
BYTEREV	The contents of the destination data register are byte-reversed; that is, new Dn[31:24] = old Dn[7:0], ..., new Dn[7:0] = old Dn[31:24].
FF1	The data register, Dn, is scanned, beginning from the most-significant bit (Dn[31]) and ending with the least-significant bit (Dn[0]), searching for the first set bit. The data register is then loaded with the offset count from bit 31 where the first set bit appears.
INTOUCH	Loads blocks of instructions to be locked in the instruction cache.
MOV3Q.L	Moves 3-bit immediate data to the destination location.
Move from USP Move to USP	Loads and stores user stack pointer
MVS.{B,W}	Sign-extends the source operand and moves it to the destination register.
MVZ.{B,W}	Zero-fills the source operand and moves it to the destination register.
SATS.L	Performs a saturation operation for signed arithmetic and updates the destination register depending on CCR[V] and bit 31 of the register.
TAS.B	Performs an indivisible read-modify-write cycle to test and set the addressed memory byte.
Bcc.L	Branch conditionally, longword
BSR.L	Branch to sub-routine, longword
CMP.{B,W}	Compare, byte and word
CMPA.W	Compare address, longword
CMPI.{B,W}	Compare immediate, byte and word
MOVEI	Move immediate, byte and word to Ax with displacement

4 Exception Processing Overview

Exception processing for ColdFire processors is streamlined for performance. The ColdFire processors differ from the M68K family in that they include:

- A simplified exception vector table
- Reduced relocation capabilities using the vector base register
- A single exception stack frame format
- Use of separate system stack pointers for user and supervisor modes

All ColdFire processors use an instruction restart exception model. Exception processing includes all actions from the detection of the fault condition to the initiation of fetch for the first handler instruction. Exception processing is comprised of four major steps.

1. 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 interrupt priority mask to be set to the level of the current interrupt request.
2. 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 the interrupt controller if CPUCR[IAE] = 1. The IACK cycle is mapped to special locations within the interrupt controller's address space with the interrupt level encoded in the address. If CPUCR[IAE] = 0, then the processor uses the vector number supplied by the interrupt controller at the time the request was signaled for improved performance.
3. The processor saves the current context by creating an exception stack frame on the system stack. As a result, the exception stack frame is created at a 0-modulo-4 address on top of the system stack pointed to by the supervisor stack pointer (SSP). As shown in Figure 10, 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).
4. The processor calculates 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 generated by fetching an exception vector from the table located at the address defined in the vector base register. The index into the exception table is calculated as (4 × vector number). Once the exception vector has been fetched, the contents of the vector 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.

All ColdFire processors support a 1024-byte vector table aligned on any 1 Mbyte address boundary (see Table 9). For the CF1 core, the only practical locations for the vector table are based at 0x00_0000 in the flash or 0x80_0000 in the RAM. The table contains 256 exception vectors; the first 64 are defined by Freescale and the remaining 192 are user-defined interrupt vectors. For the V1 ColdFire core, the table is partially populated with the first 64 reserved for internal processor exceptions, while vectors 64-102 are reserved for the peripheral I/O requests and the 7 software interrupts. The IRQ assignments are device-specific as they depend on the exact set of peripherals for any given device.

Table 9. V1 ColdFire Exception Vector Assignments

Vector Number(s)	Vector Offset (Hex)	Stacked Program Counter	Assignment
0	0x000	—	Initial supervisor stack pointer
1	0x004	—	Initial program counter
2	0x008	Fault	Access error
3	0x00C	Fault	Address error

Table 9. V1 ColdFire Exception Vector Assignments (continued)

Vector Number(s)	Vector Offset (Hex)	Stacked Program Counter	Assignment
4	0x010	Fault	Illegal instruction
5	0x014	Fault	Divide by zero
6–7	0x018–0x01C	—	Reserved
8	0x020	Fault	Privilege violation
9	0x024	Next	Trace
10	0x028	Fault	Unimplemented line-a opcode
11	0x02C	Fault	Unimplemented line-f opcode
12	0x030	Next	Debug interrupt
13	0x034	—	Reserved
14	0x038	Fault	Format error
15–23	0x03C–0x05C	—	Reserved
24	0x060	Next	Spurious interrupt
25–31	0x064–0x07C	—	Reserved
32–47	0x080–0x0BC	Next	Trap # 0-15 instructions
48–63	0x0C0–0x0FC	—	Reserved
64–95	0x100–0x17c	Next	Reserved for Peripheral IRQs
96	0x180	Next	Level 7 Software Interrupt
97	0x184	Next	Level 6 Software Interrupt
98	0x188	Next	Level 5 Software Interrupt
99	0x18c	Next	Level 4Software Interrupt
100	0x190	Next	Level 3 Software Interrupt
101	0x194	Next	Level 2 Software Interrupt
102	0x198	Next	Level 1Software Interrupt
103–255	0x19c–0x3FC	—	Reserved; unused for V1

[1] “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.

All ColdFire processors inhibit interrupt sampling 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. In addition, the V1 instruction set architecture (ISA_C) includes an instruction (STLDSR) that stores the current interrupt mask level and loads a value into the SR. This instruction is specifically intended for use as the first instruction of an interrupt service routine that services multiple interrupt requests with different interrupt levels. For more details, see the *ColdFire Family Programmer’s Reference Manual*.

4.1 Exception Stack Frame Definition

The exception stack frame is shown in Figure 10. The first longword of the exception stack frame contains the 16-bit format/vector word (F/V) and the 16-bit status register, and the second longword contains the 32-bit program counter address.

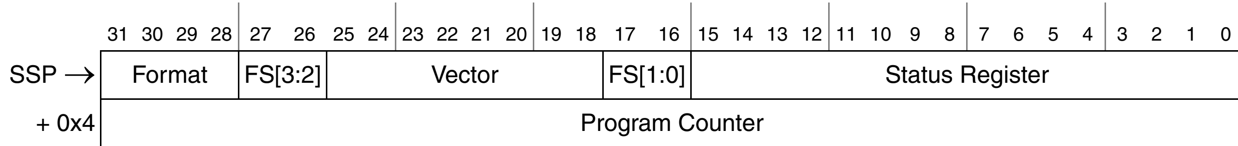


Figure 10. Exception Stack Frame Form

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, or 7 by the processor indicating a two-longword frame format. See Table 10.

Table 10. Format Field Encodings

Original SSP @ Time of Exception, Bits 1:0	SSP @ 1st Instruction of Handler	Format Field
00	Original SSP - 8	0100
01	Original SSP - 9	0101
10	Original SSP - 10	0110
11	Original SSP - 11	0111

- There is 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 11.

Table 11. Fault Status Encodings

FS[3:0]	Definition
00*xx*	Reserved
0100	Error on instruction fetch
0101	Reserved
011-	Reserved
1000	Error on operand write
1001	Attempted write to write-protected space
101-	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 interrupt controller in the case of an interrupt. Refer to Table 9.

4.2 S08 and ColdFire Exception Processing Comparison

This section presents a brief summary comparing the exception processing differences between the S08 and ColdFire processor families.

Table 12. Exception Processing Comparison

Attribute	S08	V1 ColdFire
Exception Vector Table	32, 2-byte entries, fixed location at upper end of memory	103, 4-byte entries, located at lower end of memory at reset, relocatable with the VBR
More on Vectors	2 for CPU + 30 for IRQs, reset at upper address	64 for CPU + 39 for IRQs, reset at lowest address
Exception Stack Frame	5-byte frame: CCR, A, X, PC	8-byte frame: F/V, SR, PC; General-purpose registers (An, Dn) must be saved/restored by the ISR
Interrupt Levels	1 = *f*(CCR[I])	7= *f*(SR[I]) with automatic hardware support for nesting
Non-Maskable IRQ Support	No	Yes with level 7 interrupts
Core-enforced IRQ Sensitivity	No	Level 7 is edge sensitive, else level sensitive
INTC Vectoring	Fixed priorities and vector assignments	Fixed priorities and vector assignments, plus any 2 IRQs can be remapped as the highest priority level 6 requests
Software IACK	No	Yes
Exit Instruction from ISR	RTI	RTE

The notion of a software IACK refers to the ability to query the interrupt controller near the end of an interrupt service routine (after the current interrupt request has been cleared) to determine if there are any pending (but currently masked) interrupt requests. If the response to the software IACK's byte operand read is non-zero, then the service routine uses the value as the vector number of the highest pending interrupt request and passes control to the appropriate new handler. This process avoids the overhead of a context restore and RTE instruction execution followed immediately by another interrupt exception and context save. In system environments with high rates of interrupt activity, this mechanism can improve overall performance noticeably.

Emulation of the S08's 1-level IRQ processing can easily be handled by software convention within the ColdFire interrupt service routines. For this type of operation, only two of the seven interrupt levels are used:

- SR[I] = 0 indicates interrupts are enabled
- SR[I] = 7 indicates interrupts are disabled

Recall that ColdFire treats true level 7 interrupts as edge-sensitive, non-maskable requests. Typically, only the IRQ input pin and a low-voltage detect are assigned as level 7 requests. All the remaining interrupt requests (levels 1-6) are masked when SR[I] = 7. In any case, all ColdFire processors guarantee that the first instruction of any exception handler is executed before interrupt sampling resumes. By making the first instruction of the ISR either a store/load status register (`STLDSR #0x2700`) or a move-to-SR (`MOVE.W #2700,SR`) instruction, interrupts can be safely disabled until the service routine is exited with an RTE instruction that lowers the SR[I] back to level 0.

4.3 Processor Exceptions

4.3.1 Access Error Exception

The default operation of the V1 ColdFire processor is the generation of an illegal address reset event if an access error (also known as a bus error) is detected. If CPUCR[ARD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

The exact processor response to an access error depends on the type of memory reference being performed. For an instruction fetch, the processor postpones the error reporting until the faulted reference is needed by an instruction for execution. Therefore, faults that occur during instruction prefetches that are then followed by a change of instruction flow do not generate an exception. When the processor attempts to execute an instruction with a faulted opword and/or extension words, the access error is signaled and the instruction aborted. For this type of exception, the programming model has not been altered by the instruction generating the access error.

If the access error occurs on an operand read, the processor immediately aborts the current instruction's execution and initiates exception processing. In this situation, any address register updates attributable to the auto-addressing modes, {for example, (An)+,-(An)}, have already been performed, so the programming model contains the updated An value. In addition, if an access error occurs during the execution of a MOVEM instruction loading from memory, any registers already updated before the fault occurs contain the operands from memory.

The V1 ColdFire processor uses an imprecise reporting mechanism for access errors on operand writes. Because 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.

4.3.2 Address Error Exception

The default operation of the V1 ColdFire processor is the generation of an illegal address reset event if an address error is detected. If CPUCR[ARD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

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

Any attempted use of a word-sized index register (Xn.w) or a scale factor of eight on an indexed effective addressing mode generates an address error, as does an attempted execution of a full-format indexed addressing mode which is defined by bit 8 of extension word 1 being set.

If an address error occurs on an RTS instruction, the Version 1 ColdFire processor overwrites the faulting return PC with the address error stack frame.

4.3.3 Illegal Instruction Exception

The default operation of the V1 ColdFire processor is the generation of an illegal opcode reset event if an illegal instruction is detected. If CPUCR[IRD] = 1, then the reset is disabled and a processor exception is generated as detailed below. There is one special case involving the ILLEGAL opcode (0x4afc); attempted execution of this instruction always generates an illegal instruction exception, regardless of the state of the CPUCR[IRD] bit.

Recall the ColdFire variable-length instruction set architecture supports three instruction sizes: 16, 32, or 48 bits. The first instruction word is known as the operation word (or opword), while the optional words are known as extension word 1 and extension word 2. The opword is further subdivided into three sections: the upper four bits segment the entire ISA into 16 instruction lines, the next 6 bits define the operation mode (opmode), and the low-order 6 bits define the effective address. See Figure 11. The opword line definition is shown in Table 13.

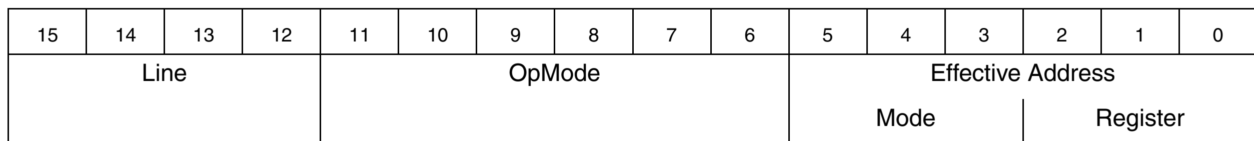


Figure 11. ColdFire Instruction Operation Word (Opword) Format

Table 13. ColdFire Opword Line Definition

Opword[15:12], Line	Instruction Class
0x0	Bit manipulation, Arithmetic and Logical Immediate
0x1	Move Byte
0x2	Move Long
0x3	Move Word
0x4	Miscellaneous
0x5	Add (ADDQ) and Subtract Quick (SUBQ), Set according to Condition Codes (Scc)
0x6	PC-relative change-of-flow instructions Conditional (Bcc) and unconditional (BRA) branches, subroutine calls (BSR)
0x7	Move Quick (MOVEQ), Move with sign extension (MVS) and zero fill (MVZ)
0x8	Logical OR (OR)
0x9	Subtract (SUB), Subtract Extended (SUBX)
0xA	{E}MAC, Move 3-bit Quick (MOV3Q)
0xB	Compare (CMP), Exclusive-OR (EOR)
0xC	Logical AND (AND), Multiply Word (MUL)
0xD	Add (ADD), Add Extended (ADDX)
0xE	Arithmetic and logical shifts (ASL, ASR, LSL, LSR)
0xF	Floating-point Instructions, Cache Push (CPUSHL), Write DDATA (WDDATA), Write Debug (WDEBUG)

In the original M68K ISA definition, lines A and F were effectively reserved for user-defined operations (line A) and co-processor instructions (line F). Accordingly, there are two unique exception vectors associated with illegal opwords in these two lines.

Any attempted execution of an illegal 16-bit opcode (except for line-A and line-F opcodes) generates an illegal instruction exception (vector 4). Additionally, any attempted execution of any illegal line-A or line-F opcode generates their unique exception types, vector numbers 10 and 11, respectively. ColdFire cores do not provide illegal instruction detection on the extension words on any instruction, including MOVEC.

The V1 ColdFire processor also detects two special cases involving illegal instruction conditions:

1. If execution of the STOP instruction is attempted and neither low-power stop nor wait modes are enabled, then the processor signals an illegal instruction.
2. If execution of the HALT instruction is attempted and BDM is not enabled (XCSR[ENBDM] = 0), then the processor signals an illegal instruction.

In both cases, the processor response is then dependent on the state of CPUCR[IRD]—either a reset event or a processor exception.

4.3.4 Divide-By-Zero

If the optional integer divide execute engine is present, an attempt to divide by zero causes an exception (vector 5, offset = 0x014).

4.3.5 Privilege Violation

The default operation of the V1 ColdFire processor is the generation of an illegal opcode reset event if a privilege violation is detected. If CPUCR[IRD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

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

4.3.6 Trace Exception

To aid in program development, all ColdFire processors provide an instruction-by-instruction tracing capability. While in trace mode, indicated by setting of the SR[T] bit, the completion of an instruction execution (for all but the STOP instruction) signals a trace exception. This functionality allows a debugger to monitor program execution.

The STOP instruction has the following effects:

1. The instruction before the STOP executes and then generates a trace exception. In the exception stack frame, the PC points to the STOP opcode.
2. When the trace handler is exited, the STOP instruction is executed, loading the SR with the immediate operand from the instruction.

3. The processor then generates a trace exception. The PC in the exception stack frame points to the instruction after the STOP, and the SR reflects the value loaded in the previous step.

If the processor is not in trace mode and executes a STOP instruction where the immediate operand sets SR[T], hardware loads the SR and generates a trace exception. The PC in the exception stack frame points to the instruction after the STOP, and the SR reflects the value loaded in step 2.

Because 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 will initiate the TRAP exception and then pass 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[T] in the exception stack frame set) and pass control to the trace handler before returning from the original exception.

4.3.7 Unimplemented Line-A Opcode

The default operation of the V1 ColdFire processor is the generation of an illegal opcode reset event if an unimplemented line-A opcode is detected. If CPUCR[IRD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

A line-A opcode is defined when bits 15-12 of the opword are 0b1010. This exception is generated by the attempted execution of an undefined line-A opcode.

4.3.8 Unimplemented Line-F Opcode

The default operation of the V1 ColdFire processor is the generation of an illegal opcode reset event if an unimplemented line-F opcode is detected. If CPUCR[IRD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

A line-F opcode is defined when bits 15-12 of the opword are 0b1111. This exception is generated when attempting to execute an undefined line-F opcode.

4.3.9 Debug Interrupt

This special type of program interrupt 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, SR[M,I] are unaffected by this interrupt.

4.3.10 RTE and Format Error Exception

The default operation of the V1 ColdFire processor is the generation of an “illegal address” reset event if an RTE format error is detected. If CPUCR[ARD] = 1, then the reset is disabled and a processor exception is generated as detailed below.

When an RTE instruction is executed, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire core, any attempted RTE execution 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 RTE frame and the stacked PC pointing to the RTE instruction.

The selection of the format value provides some limited debug support for porting code from M68K applications. On M68K 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.

4.3.11 TRAP Instruction Exception

The TRAP #n instructions always force an exception as part of their execution and are useful for implementing system calls. The trap instructions may be used to change from user to supervisor mode.

This set of 16 instructions provides a similar but expanded functionality compared to the S08’s SWI (software interrupt) instruction. These instructions and their functionality should not be confused with the software-scheduled interrupt requests, which are handled like normal I/O interrupt requests by the interrupt controller. The processing of the software-scheduled IRQs can be masked, based on the interrupt priority level defined by the SR[I] field.

4.3.12 Interrupt Exception

Interrupt exception processing includes interrupt recognition and the calculation of the appropriate vector, retrieved from the interrupt controller either using an IACK cycle or using the previously-supplied vector number, under control of CPUCR[IAE]. The SR[M] bit is cleared and SR[I] is set to the interrupt level being processed. See Section 1.2, “Features” for an overview of the interrupt controller. All ColdFire processors sample for interrupts once during each instruction’s execution, during the first cycle of execution in the OEP unless specifically noted otherwise.

The details on IRQ and vector assignments are device-specific.

4.3.13 Fault-on-Fault Halt

The default operation of the V1 ColdFire processor is the generation of an illegal address reset event if a fault-on-fault halt condition is detected. If CPUCR[ARD] = 1, then the reset is disabled and the processor is halted as detailed below.

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.

4.3.14 Reset Exception

There are multiple events that can generate a reset signal to the processor. Asserting a reset input signal (RESET), setting of a BDM "force_reset" control bit, detecting of certain illegal address or opcode conditions, all force the processor to initiate 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 SR[S] bit and disables tracing by clearing the SR[T] bit. This exception also clears the SR[M] bit and sets the processor's SR[I] field to the highest level (level 7, 0b111). Next, the VBR is initialized to zero (0x0000_0000). The control registers specifying the CPU configuration are also cleared.

The processor initiates the reset exception processing by performing two longword read bus cycles. The first longword at address 0 is loaded into the supervisor 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.

ColdFire processors load hardware configuration information into the D0 and D1 general-purpose registers after system reset. The hardware configuration information is loaded immediately after the reset-in signal is negated. This allows an emulator to read out the contents of these registers via BDM to determine the hardware configuration before instruction execution begins, if desired.

Information loaded into D0 defines the processor hardware configuration as shown in Figure 12.

BDM: 0x60 (D0 read)

Access: User read-only
BDM read-only

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	PF								VER				REV			
W																
Reset	1	1	0	0	1	1	1	1	0	0	0	1	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	MAC	DIV	EMAC	FPU	MMU	CAU	0	0	ISA				DEBUG			
W																
Reset	*	*	*	0	0	*	0	0	0	0	1	0	1	0	0	1

Figure 12. D0 Hardware Configuration Info

Table 14. D0 Hardware Configuration Info Field Description

Field	Description
31–24 PF	Processor family. This field is fixed to a hex value of 0xCF indicating a ColdFire core is present.
23–20 VER	ColdFire core version number. Defines the hardware microarchitecture version of the ColdFire core. 0001 V1 ColdFire core (This is the value used for this device) 0010 V2 ColdFire core 0011 V3 ColdFire core 0100 V4 ColdFire core 0101 V5 ColdFire core Else Reserved for future use.
19–16 REV	Processor revision number. The default is 0b0000.
15 MAC	MAC present.This bit signals if the optional multiply-accumulate (MAC) execution engine is present in the processor core. 0 MAC execute engine not present in core. 1 MAC execute engine is present in core. The reset state depends on CPU hardware configuration.
14 DIV	Divide present. This bit signals if the hardware divider (DIV) is present in the processor core. Certain CF1 core implementations do not include hardware support for integer divide operations. 0 Divide execute engine not present in core. 1 Divide execute engine is present in core. The reset state depends on the CPU hardware configuration.
13 EMAC	EMAC present. This bit signals if the optional enhanced multiply-accumulate (EMAC) execution engine is present in the processor core. 0 EMAC execute engine not present in core. 1 EMAC execute engine is present in core. The reset state depends on the CPU hardware configuration.
12 FPU	FPU present. This bit signals if the optional floating-point (FPU) execution engine is present in the processor core. 0 FPU execute engine not present in core. (This is the value used for this device) 1 FPU execute engine is present in core.
11 MMU	MMU present. This bit signals if the optional virtual memory management unit (MMU) is present in the processor core. 0 MMU execute engine not present in core. (This is the value used for this device) 1 MMU execute engine is present in core.

Table 14. D0 Hardware Configuration Info Field Description (continued)

Field	Description
10 CAU	Cryptographic acceleration unit present. This bit signals if the optional cryptographic acceleration unit (CAU) is present in the processor core. 0 CAU coprocessor engine not present in core. 1 CAU coprocessor engine is present in core. The reset state depends on the CPU hardware configuration.
9–8	Reserved.
7–4 ISA	ISA revision. This 4-bit field defines the instruction set architecture (ISA) revision level implemented in the ColdFire processor core. 0000 ISA_A 0001 ISA_B 0010 ISA_C (This is the value used for this device) 1000 ISA_A+ Else Reserved
3–0 DEBUG	Debug module revision number. This 4-bit field defines the revision level of the debug module implemented in the ColdFire processor core. 0000 DEBUG_A 0001 DEBUG_B 0010 DEBUG_C 0011 DEBUG_D 0100 DEBUG_E 1001 DEBUG_B+ (This is the value used for this device) 1011 DEBUG_D+ Else Reserved

Information loaded into D1 defines the platform memory hardware configuration as shown in the Table 15 below.

BDM 0x61 (D1 read)

Access: User read-only
BDM read-only

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	0	0	1	0	0	0	0	FLASHSZ				0	0	0	0
W																
Reset	0	0	0	1	0	0	0	0	*	*	*	*	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	0	1	0	0	0	0	RAMSZ				0	0	0	0
W																
Reset	0	0	0	1	0	0	0	0	*	*	*	*	0	0	0	0

Figure 13. D1 Hardware Configuration Info

Table 15. D1 Hardware Configuration Information Field Description

Field	Description
31–24	Reserved
23–20 FLASHSZ	Flash size. 0000-0101 No flash 0110 16 Kbytes 0111 32 Kbytes 1000 64 Kbytes 1001 128 Kbytes 1010 256 Kbytes 1011 512 Kbytes Else Reserved for future use. The reset state depends on the size of the attached memory.
19–8	Reserved
7–4 RAMSZ	RAM size. 0000 No SRAM 0001 512 bytes 0010 1 Kbytes 0011 2 Kbytes 0100 4 Kbytes 0101 8 Kbytes 0110 16 Kbytes 0111 32 Kbytes 1000 64 Kbytes 1001 128 Kbytes Else Reserved for future use The reset state depends on the size of the attached memory.
3-0	Reserved

5 Debug Overview

As previously discussed, the V1 core implements Revision B+ of the ColdFire debug architecture using a single-pin BDM interface with enhanced capabilities related to program (and optional data) trace. The baseline DEBUG_B+ functionality is detailed in any number of V2 ColdFire device reference manuals; see Chapter 31 of the MCF52235RM as an example. This document is available at:

http://www.freescale.com/files/32bit/doc/ref_manual/MCF52235RM.pdf

5.1 Debug Register Descriptions

In addition to the BDM commands that provide access to the processor's registers and memory, the debug module contains 27 registers to support the required functionality. Most of these registers (all except the PST/DDATA trace buffer) are also accessible from the processor's supervisor programming model by executing the WDEBUG instruction (write only). Thus, the breakpoint hardware in the debug module can be written by the external development system using the debug serial interface or by the operating system running on the processor core. Software is responsible for guaranteeing that accesses to these resources are serialized and logically consistent. Hardware provides a locking mechanism in the CSR to allow the external development system to disable any attempted writes by the processor to the breakpoint registers

(CSR[IPW] = 1). BDM commands must not be issued if the ColdFire processor is using the WDEBUG instruction to access debug module registers, or the resulting behavior is undefined.

These registers, shown in Table 16, are treated as 32-bit quantities, regardless of the number of implemented bits. These registers are also accessed through the BDM port by the commands, READ_DREG and WRITE_DREG, described in Section 5.2.4, "BDM Command Set." These commands contain a DRc field that specifies the register, as shown in Table 16.

Table 16. Debug Control Register Memory Map

DRc	Register	Width (bits)	Access	Reset Value	Section/Page
0x00	Configuration/Status Register (CSR)	32	Note[1]	0x0090_0000	See MCF52235RM, Chapter 31
0x01	Extended Configuration/Status Register (XCSR)[2]	32	Note[1]	0x0000_0000	To be supplied
0x02	Hyper-extended Configuration/Status Register (XXCSR)[2]	32	Note[1]	0x0000_0000	To be supplied
0x05	BDM Address Attribute Register (BAAR)	32[3]	Note[1]	0x05	See MCF52235RM, Chapter 31
0x06	Address Attribute Trigger Register (AATR)	32[3]	Note[1]	0x0005	See MCF52235RM, Chapter 31
0x07	Trigger Definition Register (TDR)	32	Note[1]	0x0000_0000	See MCF52235RM, Chapter 31
0x08	PC Breakpoint Register 0 (PBR0)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x09	PC Breakpoint Mask Register (PBMR)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x0C	Address High Breakpoint Register (ABHR)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x0D	Address Low Breakpoint Register (ABLR)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x0E	Data Breakpoint Register (DBR)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x0F	Data Breakpoint Mask Register (DBMR)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x18	PC Breakpoint Register 1 (PBR1)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31

Table 16. Debug Control Register Memory Map (continued)

DRc	Register	Width (bits)	Access	Reset Value	Section/Page
0x1A	PC Breakpoint Register 2 (PBR2)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x1B	PC Breakpoint Register 3 (PBR3)	32	Note[1]	Undefined	See MCF52235RM, Chapter 31
0x20	PST Trace Buffer 0 (PSTB0)	32	BDM	Undefined	To be supplied
0x21	PST Trace Buffer 1 (PSTB1)	32	BDM	Undefined	To be supplied
0x22	PST Trace Buffer 2 (PSTB2)	32	BDM	Undefined	To be supplied
0x23	PST Trace Buffer 3 (PSTB3)	32	BDM	Undefined	To be supplied
0x24	PST Trace Buffer 4 (PSTB4)	32	BDM	Undefined	To be supplied
0x25	PST Trace Buffer 5 (PSTB5)	32	BDM	Undefined	To be supplied
0x26	PST Trace Buffer 6 (PSTB6)	32	BDM	Undefined	To be supplied
0x27	PST Trace Buffer 7 (PSTB7)	32	BDM	Undefined	To be supplied
0x28	PST Trace Buffer 8 (PSTB8)	32	BDM	Undefined	To be supplied
0x29	PST Trace Buffer 9 (PSTB9)	32	BDM	Undefined	To be supplied
0x2A	PST Trace Buffer 10 (PSTB10)	32	BDM	Undefined	To be supplied
0x2B	PST Trace Buffer 11 (PSTB11)	32	BDM	Undefined	To be supplied

NOTES

Debug control registers can be written by the BDM or the CPU through the WDEBUG instruction. These control registers are write-only from the programming model and they can be written through the BDM port using the WRITE_DREG command. In addition, the three configuration/status registers (CSR, XCSR, XXCSR) can be read through the BDM port using the READ_DREG command.

The most significant bytes of the XCSR and XXCSR registers support special control functions and are only writeable via BDM using the WRITE_XCSR_BYTE and WRITE_XXCSR_BYTE serial commands. These two debug control registers can be read from BDM using the READ_DREG commands.

Each debug register is accessed as a 32-bit register; reserved fields are not used (don't care).

5.2 Background Debug Mode (BDM)

The V1 ColdFire core supports the classic BDM functionality using the S08's single-pin interface. The traditional 3-pin full-duplex ColdFire BDM serial communication protocol based on 17-bit data packets is replaced with the HCS08 protocol where all communications are based on an 8-bit data packet using a

single package pin (BKGD). Subsequent paragraphs in this section provide details on the BKGD pin, the background debug serial interface controller (BDC), a standard 6-pin BDM connector, and the BDM command set.

5.2.1 BKGD Pin Description

BKGD is the single-wire background debug interface pin. The primary function of this pin is for bidirectional serial communication of active background (halt) mode commands and data. During reset, this pin is used to select between starting in active background mode or starting the user's application program. This pin is also used to request a timed sync response pulse to allow a host development tool to determine the correct clock frequency for background debug serial communications.

BDC serial communications use a custom serial protocol first introduced on the M68HC12 Family of microcontrollers and later used in the M68HCS08 family. This protocol assumes the external controller (the host) knows the communication clock rate that is determined by the target BDC clock rate. All communication is initiated and controlled by the host that drives a high-to-low edge to signal the beginning of each bit time. Commands and data are sent most significant bit (msb) first. For a detailed description of the communications protocol, refer to Section 5.2.3, "BDM Communication Details."

If a host is attempting to communicate with a target MCU that has an unknown BDC clock rate, a SYNC command may be sent to the target MCU to request a timed synchronization response signal from which the host can determine the correct communication speed.

BKGD is a pseudo-open-drain pin and there is an on-chip pullup so no external pullup resistor is required. Unlike typical open-drain pins, the external RC time constant on this pin, which is influenced by external capacitance, plays almost no role in signal rise time. The custom protocol provides for brief, actively driven speed-up pulses to force rapid rise times on this pin without risking harmful drive level conflicts. Refer to Section 5.2.3, "BDM Communication Details," for more detail.

When no debugger pod is connected to the standard 6-pin BDM interface connector (Section 5.2.2, "Standard BDM Connector"), the internal pullup on BKGD chooses normal operating mode. When a development system is connected, it can pull both BKGD and $\overline{\text{RESET}}$ low, release $\overline{\text{RESET}}$ to select active background mode rather than normal operating mode, then release BKGD. It is not necessary to reset the target MCU to communicate with it through the background debug interface. There is also a mechanism to generate a reset event in response to the programming of a special BDM control bit.

5.2.2 Standard BDM Connector

Typically, a relatively simple interface pod is used to translate commands from a host computer into commands for the custom serial interface to the single-wire background debug system. Depending on the development tool vendor, this interface pod may use a standard RS-232 serial port, a parallel printer port, or some other type of communications such as a universal serial bus (USB) to communicate between the host PC and the pod. The pod typically connects to the target system with ground, the BKGD pin, $\overline{\text{RESET}}$, and sometimes V_{DD}. An open-drain connection to reset allows the host to force a target system reset, which is useful to regain control of a lost target system or to control startup of a target system before the on-chip nonvolatile memory has been programmed. Sometimes V_{DD} can be used to allow the pod to use power from the target system to avoid the need for a separate power supply. However, if the pod is powered

separately, it can be connected to a running target system without forcing a target system reset or otherwise disturbing the running application program.

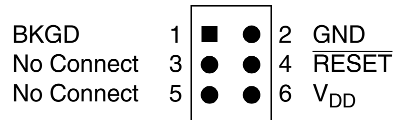


Figure 14. BDM Tool Connector

5.2.3 BDM Communication Details

The BDC serial interface requires the external host controller to generate a falling edge on the BKGD pin to indicate the start of each bit time. The external controller provides this falling edge whether data is transmitted or received.

BKGD is a pseudo-open-drain pin that can be driven either by an external controller or by the MCU. Data is transferred msb first at 16 BDC clock cycles per bit (nominal speed). The interface times out if 512 BDC clock cycles occur between falling edges from the host. Any BDC command that was in progress when this time-out occurs is aborted without affecting the memory or operating mode of the target MCU system.

The custom serial protocol requires the debug pod to know the target BDC communication clock speed. The clock switch (CLKSW) control bit in the XCSR[31:24] control register allows the user to select the BDC clock source. The BDC clock source can either be the peripheral bus or the alternate BDC clock source.

The BKGD pin can receive a high or low level or transmit a high or low level. The following diagrams show timing for each of these cases. Interface timing is synchronous to clocks in the target BDC, but asynchronous to the external host. The internal BDC clock signal is shown for reference in counting cycles.

Figure 15 shows an external host transmitting a logic 1 or 0 to the BKGD pin of a target CF1 MCU. The host is asynchronous to the target so there is a 0-to-1 cycle delay from the host-generated falling edge to where the target perceives the beginning of the bit time. Ten target BDC clock cycles later, the target senses the bit level on the BKGD pin. Typically, the host actively drives the pseudo-open-drain BKGD pin during host-to-target transmissions to speed up rising edges. Because the target does not drive the BKGD pin during the host-to-target transmission period, there is no need to treat the line as an open-drain signal during this period.

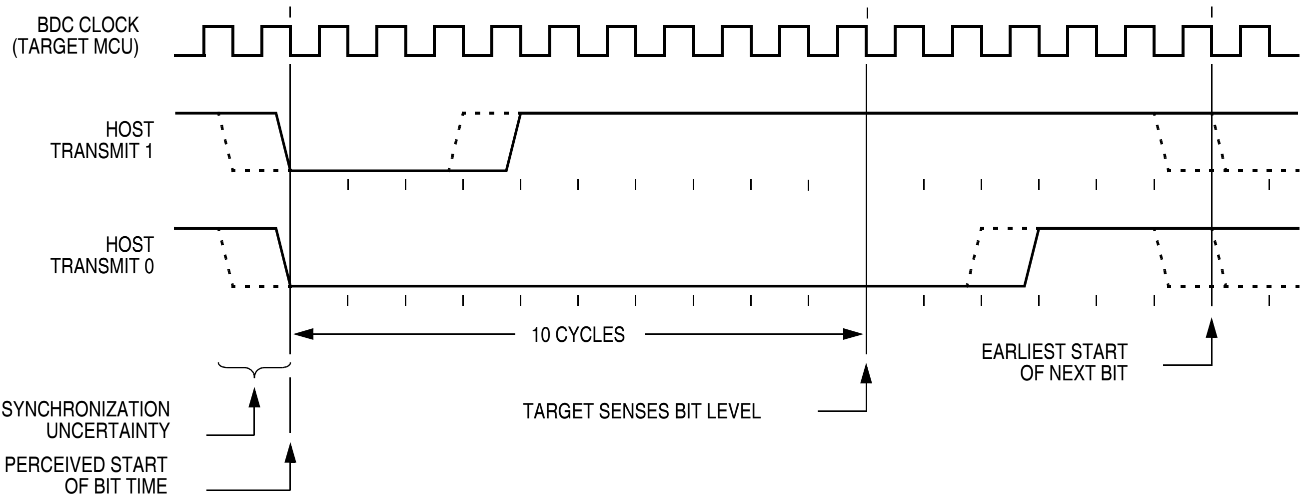


Figure 15. BDC Host-to-Target Serial Bit Timing

Figure 16 shows the host receiving a logic 1 from the target CF1 MCU. Because the host is asynchronous to the target MCU, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the perceived start of the bit time in the target MCU. The host holds the BKGD pin low long enough for the target to recognize it (at least two target BDC cycles). The host must release the low drive before the target MCU drives a brief active-high speedup pulse seven cycles after the perceived start of the bit time. The host should sample the bit level about 10 cycles after it started the bit time.

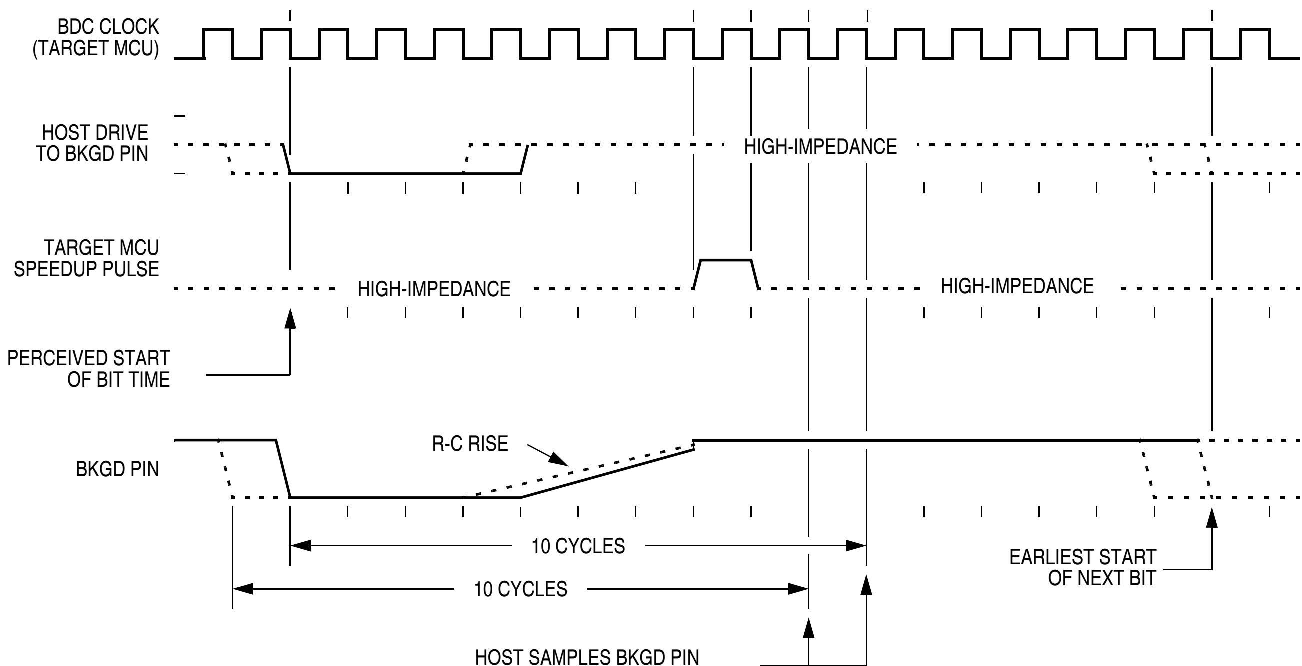


Figure 16. BDC Target-to-Host Serial Bit Timing (Logic 1)

Figure 17 shows the host receiving a logic 0 from the target CF1 MCU. Because the host is asynchronous to the target MCU, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the start of the bit time as perceived by the target MCU. The host initiates the bit time, but the target CF1 finishes

it. Because the target wants the host to receive a logic 0, it drives the BKGD pin low for 13 BDC clock cycles, then briefly drives it high to speed up the rising edge. The host samples the bit level about 10 cycles after starting the bit time.

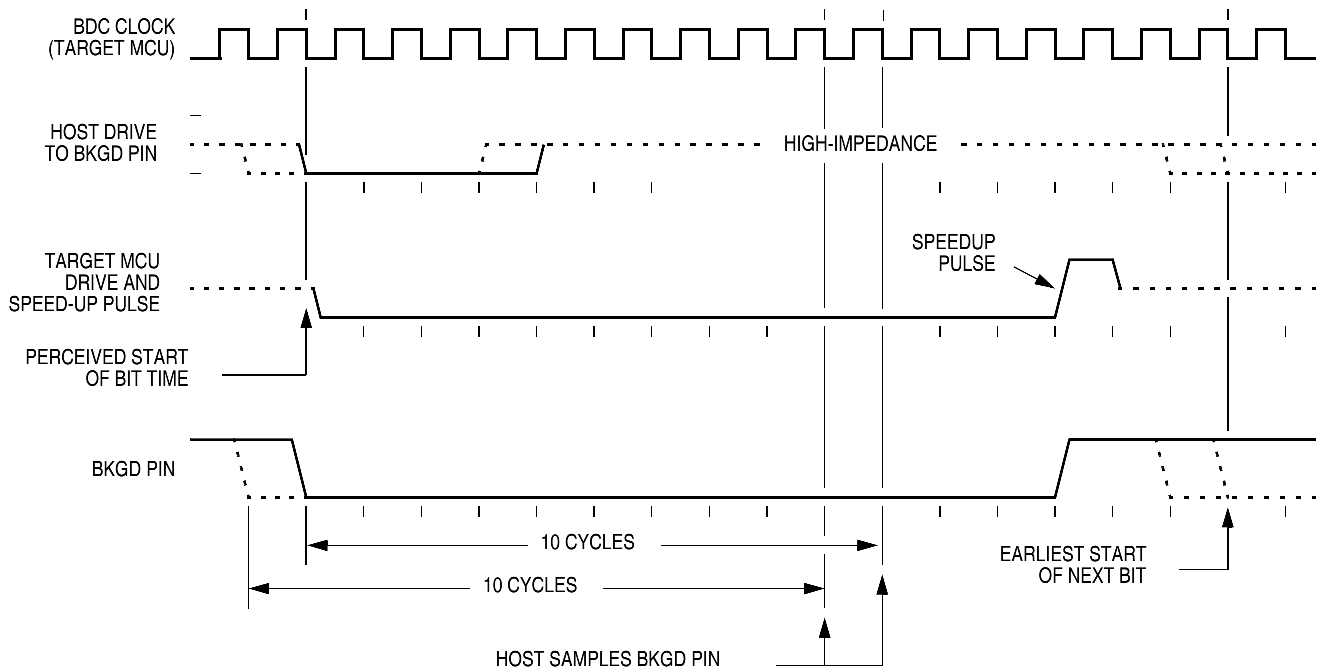


Figure 17. BDM Target-to-Host Serial Bit Timing (Logic 0)

5.2.4 BDM Command Set

The V1 BDM command set is based on transmission of one or more 8-bit data packets for each operation. Each operation begins with a host-to-target transmission of an 8-bit command code packet. The command code definition broadly classifies the operations into three categories as shown in Figure 18.

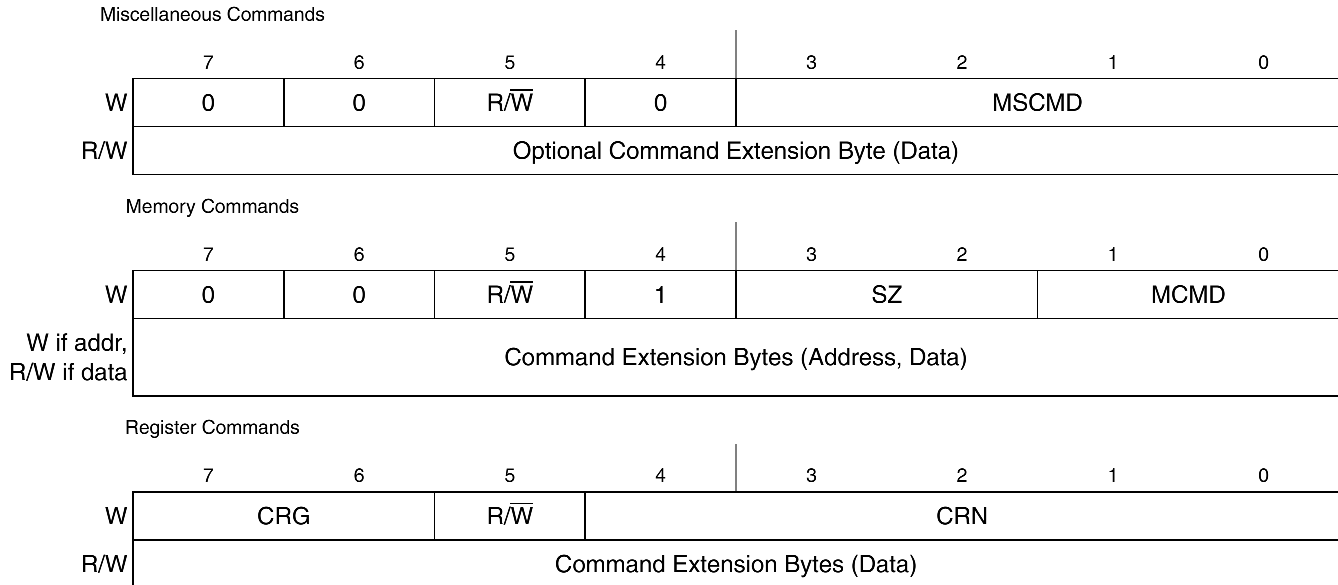


Figure 18. BDM Command Code Encodings

Table 17. BDM Command Code Field Descriptions

Field	Description
5 R/$\overline{W}$	Read/$\overline{Write}$. 0 The command is performing a write operation. 1 The command is performing a read operation.
3–0 MSCMD	Miscellaneous command. Defines the miscellaneous command to be performed. 0000 No operation 0001 Display the CPU's program counter (PC) and optionally capture it in the PST trace buffer 0010 Enable the BDM acknowledge communication mode 0011 Disable the BDM acknowledge communication mode 0100 Force a CPU halt (background) 1000 Resume CPU execution (go) 1101 Read/write of the debug XCSR most significant byte 1110 Read/write of the debug XXCSR most significant byte

Table 17. BDM Command Code Field Descriptions (continued)

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>3–2
SZ</td><td>Memory operand size. Defines the size of the memory reference.
00 8-bit byte
01 16-bit word
10 32-bit long</td></tr>
<tr><td>1–0
MCMD</td><td>Memory command. Defines the type of the memory reference to be performed.
00 Simple write if R/$\overline{W}$ = 0; simple read if R/$\overline{W}$ = 1
01 Write + status if R/$\overline{W}$ = 0; read + status if R/$\overline{W}$ = 1
10 Fill if R/$\overline{W}$ = 0; dump if R/$\overline{W}$ = 1
11 Fill + status if R/$\overline{W}$ = 0; dump + status if R/$\overline{W}$ = 1</td></tr>
<tr><td>7–6
CRG</td><td>Core register group. Defines the core register group to be referenced.
01 CPU’s general-purpose registers (An, Dn)
10 DBG’s control registers
11 CPU’s control registers (PC, SR, VBR, CPUCR, ...)</td></tr>
<tr><td>4–0
CRN</td><td>Core register number. Defines the specific core register (its number) to be referenced.
<table>
<tr><th>CRG</th><th>CRN</th><th>Register</th></tr>
<tr><td rowspan="2">01</td><td>0x00–0x07</td><td>D0–7</td></tr>
<tr><td>0x08–0x0F</td><td>A0–7</td></tr>
<tr><td>10</td><td colspan="2">DRc[4:0] as described in Table 16</td></tr>
<tr><td rowspan="5">11</td><td>0x00</td><td>OTHER_A7</td></tr>
<tr><td>0x01</td><td>VBR</td></tr>
<tr><td>0x02</td><td>CPUCR</td></tr>
<tr><td>0x0E</td><td>SR</td></tr>
<tr><td>0x0F</td><td>PC</td></tr>
</table>
where all other CRN values are reserved</td></tr>
</table>

Table 18 presents a BDM command summary. The commands are presented in alphabetical order, based on their mnemonics, except for the special SYNC command which is presented first.

Command Structure Nomenclature

This nomenclature is used in Table 18 to describe the structure of the BDM commands.

Commands begin with an 8-bit hexadecimal command code in the host-to-target direction (most significant bit first)

```
/        =     separates parts of the command
d        =     delay 16 target BDC clock cycles
ad24     =     24-bit memory address in the host-to-target direction
rd8      =     8 bits of read data in the target-to-host direction
rd16     =     16 bits of read data in the target-to-host direction
rd32     =     32 bits of read data in the target-to-host direction
rd.sz    =     read data, size defined by sz, in the target-to-host direction
wd8      =     8 bits of write data in the host-to-target direction
wd16     =     16 bits of write data in the host-to-target direction
wd32     =     32 bits of write data in the host-to-target direction
wd.sz    =     write data, size defined by sz, in the host-to-target direction
```

```
ss      =       the contents of XCSR[31:24] in the target-to-host direction (STATUS)
sz      =       memory operand size (0b00 = byte, 0b01 = word, 0b10 = long)
crn     =       core register number
```

Table 18. BDM Command Summary

Command Mnemonic	CPU State[1]	Command Structure	Description
SYNC	parallel	n/a[22]	Request a timed reference pulse to determine the target BDC communication speed
ACK_DISABLE	parallel	0x03/d	Disable the communication handshake. This command does not issue an ACK pulse.
ACK_ENABLE	parallel	0x02/d	Enable the communication handshake. Issues an ACK pulse after the command is executed.
BACKGROUND	parallel	0x04/d	Halt the CPU if ENBDM = 1, else ignore.
DUMP_MEM.sz	steal	(0x32+4 x sz)/d/rd.sz	Dump (read) memory based on operand size (sz). Used with READ_MEM to dump large blocks of memory. An initial READ_MEM is executed to set up the starting address of the block and to retrieve the first result. A DUMP_MEM command retrieves subsequent operands.
DUMP_MEM.sz_WS	steal	(0x33+4 x sz)/d/ss/rd.sz	Dump (read) memory based on operand size (sz) and report status. Used with READ_MEM{_WS} to dump large blocks of memory. An initial READ_MEM{_WS} is executed to set up the starting address of the block and to retrieve the first result. A DUMP_MEM{_WS} command retrieves subsequent operands.
FILL_MEM.sz	steal	(0x12+4 x sz)/d/wd.sz	Fill (write) memory based on operand size (sz). Used with WRITE_MEM to fill large blocks of memory. An initial WRITE_MEM is executed to set up the starting address of the block and to write the first operand. A FILL_MEM command writes subsequent operands.
FILL_MEM.sz_WS	steal	(0x13+4 x sz)/d/wd.sz/ss	Fill (write) memory based on operand size (sz) and report status. Used with WRITE_MEM{_WS} to fill large blocks of memory. An initial WRITE_MEM{_WS} is executed to set up the starting address of the block and to write the first operand. A FILL_MEM{_WS} command writes subsequent operands.
GO	halted	0x08/d	Resume the CPU's execution
NOP	parallel	0x00/d	No operation
READ_CREG	halted, stopped	(0xe0+crn)/d/rd32	Read one of the CPU's control registers
READ_DREG	halted, stopped	(0xa0+crn)/d/rd32	Read one of the debug module's control registers
READ_MEM.sz	steal	(0x30+4 x sz)/ad24/d/rd.sz	Read the appropriately-sized (sz) memory value from the location specified by the 24-bit address

Table 18. BDM Command Summary (continued)

Command Mnemonic	CPU State[1]	Command Structure	Description
READ_MEM.sz_WS	steal	(0x31+4 x sz)/ad24/d/ss/rd.sz	Read the appropriately-sized (sz) memory value from the location specified by the 24-bit address and report status
READ_PSTB	parallel	(0x50+crn)/d/rd32	Read the requested longword location from the PST trace buffer
READ_Rn	halted, stopped	(0x60+crn)/d/rd32	Read the requested general-purpose register (An, Dn) from the CPU
READ_XCSR_BYTE	parallel	0x2d/rd8	Read the most significant byte of the debug module's XCSR
READ_XXCSR_BYTE	parallel	0x2e/rd8	Read the most significant byte of the debug module's XXCSR
SYNC_PC	steal	0x01/d	Display the CPU's current PC and optionally capture it in the PST trace buffer
WRITE_CREG	halted, stopped	(0xc0+crn)/wd32/d	Write one of the CPU's control registers
WRITE_DREG	halted, stopped	(0x80+crn)/wd32/d	Write one of the debug module's control registers
WRITE_MEM.sz	steal	(0x10+4 x sz)/ad24/wd.sz/d	Write the appropriately-sized (sz) memory value to the location specified by the 24-bit address
WRITE_MEM.sz_WS	steal	(0x11+4 x sz)/ad24/wd.sz/d/ss	Write the appropriately-sized (sz) memory value to the location specified by the 24-bit address and report status
WRITE_Rn	halted, stopped	(0x40+crn)/wd32/d	Write the requested general-purpose register (An, Dn) of the CPU
WRITE_XCSR_BYTE	parallel	0x0d/wd8	Write the most significant byte of the debug module's XCSR
WRITE_XXCSR_BYTE	parallel	0x0e/wd8	Write the most significant byte of the debug module's XXCSR

[1] General command effect and/or requirements on CPU operation:
- Halted, Stopped: The CPU must be halted or stopped to perform this command.
- Steal: Command generates bus cycles that can be interleaved with CPU activity.
- Parallel: Command is executed in parallel with CPU activity.

[2] The SYNC command is a special operation which does not have a command code.

Unassigned command opcodes are reserved by Freescale. All unused command formats within any revision level perform no operation and return the illegal command response.

More detailed descriptions of the individual BDM commands will be provided in future documentation.

5.3 Processor Status (PST)/Debug Data (DDATA) and Trace Support

The classic ColdFire debug architecture supports real-time trace via the PST/DDATA output signals. For this functionality, the following apply:

- One (or more) PST value is generated for each executed instruction
- Branch target instruction address information is displayed on all non-PC-relative change-of-flow instructions, where the user selects a programmable number of bytes of target address
  - Displayed information includes PST marker plus target instruction address as DDATA
  - Captured address creates the appropriate number of DDATA entries, each with 4 bits of address
- Optional data trace capabilities are provided for accesses mapped to the slave peripheral bus
  - Displayed information includes PST marker plus captured operand value as DDATA
  - Captured operand creates the appropriate number of DDATA entries, each with 4 bits of data

The resulting PST/DDATA output stream, in conjunction with the application program memory image, provides an instruction-by-instruction dynamic trace of the execution path.

For the V1 ColdFire core, the availability of only a single pin for debug requires that support for trace functionality be completely redefined. The V1 solution provides an on-chip PST/DDATA trace buffer (known simply as the PSTB) to record the stream of PST and DDATA values.

Even with the application of an PST trace buffer, there remain problems associated with the PST bandwidth and associated fill rate of the buffer. Given that there is >1 PST entry per instruction, the PSTB would fill very rapidly without some type of data compression. Luckily, the PST compression technology was previously developed and included as part of the V5 ColdFire core.

Consider the following example to illustrate the PST compression algorithm. Most sequential instructions generate a single PST = 1 value. Without compression, the execution of ten sequential instructions generates a stream of ten PST = 1 values. With PST compression, the reporting of any PST = 1 value is delayed so that consecutive PST = 1 values can be accumulated. When a PST ≠ 1 value is reported, or the maximum accumulation count reached, or a debug data value captured, then a single accumulated PST value is generated. Returning to the example with compression enabled, the execution of ten sequential instructions generates a single PST value indicating ten sequential instructions have been executed.

This technique has proven to be very effective at significantly reducing the average PST entries per instruction and PST entries per machine cycle. The application of this compression technique makes the application of an useful PST trace buffer for the V1 ColdFire core realizable. The resulting 5-bit PST definitions are shown in Table 19.

Table 19. V1 Processor Status Encodings

PST[4:0]	Definition
0x00	Continue execution. Many instructions execute in one processor cycle. If an instruction requires more processor clock cycles, subsequent clock cycles are indicated by driving PST with this encoding.
0x01	Begin execution of one instruction. For most instructions, this encoding signals the first processor clock cycle of an instruction's execution. Certain change-of-flow opcodes, plus the PULSE and WDDATA instructions, generate different encodings.
0x02	Reserved
0x03	Entry into user-mode. Signaled after execution of the instruction that caused the ColdFire processor to enter user mode.

Table 19. V1 Processor Status Encodings (continued)

PST[4:0]	Definition
0x04	Begin execution of PULSE and WDDATA instructions. PULSE defines triggers or markers for debug and/or performance analysis. WDDATA lets the core write any operand (byte, word, or longword) directly to the DDATA port, independent of debug module configuration. When WDDATA is executed, a value of 0x04 is signaled on the PST port, followed by the appropriate marker, and then the data transfer on the DDATA port. The number of captured data bytes depends on the WDDATA operand size.
0x05	Begin execution of taken branch or SYNC_PC BDM command. For some opcodes, a branch target address may be displayed on DDATA depending on the CSR settings. CSR also controls the number of address bytes displayed, indicated by the PST marker value preceding the DDATA nibble that begins the data output. This encoding also indicates that the SYNC_PC command has been processed.
0x06	Reserved
0x07	Begin execution of return from exception (RTE) instruction.
0x08–0x0B	Indicates the number of data bytes to be displayed as DDATA on subsequent processor clock cycles. This marker value is driven as the PST one processor clock cycle before the data is displayed on DDATA. The capturing of peripheral bus data references is controlled by CSR[DDC]. 0x08 Begin 1-byte data transfer on DDATA 0x09 Begin 2-byte data transfer on DDATA 0x0A Reserved 0x0B Begin 4-byte data transfer on DDATA
0x0C–0x0F	Indicates the number of address bytes to be displayed as DDATA on subsequent processor clock cycles. This marker value is driven as the PST one processor clock cycle before the address is displayed on DDATA. The capturing of branch target addresses is controlled by CSR[BTB]. 0x0C Reserved 0x0D Begin 2-byte address transfer on DDATA (Displayed address is shifted right 1: ADDR[17:1]) 0x0E Begin 3-byte address transfer on DDATA (Displayed address is shifted right 1: ADDR[24:1]) 0x0F Reserved
0x10–0x11	Reserved
0x12	Completed execution of 2 sequential instructions
0x13	Completed execution of 3 sequential instructions
0x14	Completed execution of 4 sequential instructions
0x15	Completed execution of 5 sequential instructions
0x16	Completed execution of 6 sequential instructions
0x17	Completed execution of 7 sequential instructions
0x18	Completed execution of 8 sequential instructions
0x19	Completed execution of 9 sequential instructions
0x1A	Completed execution of 10 sequential instructions
0x1B	This value signals there has been a change in the breakpoint trigger state machine. It appears as a single marker for each state change.

Table 19. V1 Processor Status Encodings (continued)

PST[4:0]	Definition
0x1C	Exception processing. This value signals the processor has encountered an exception condition. Although this is a multi-cycle mode, there are only two PST = 0x1C values recorded before the mode value is suppressed.
0x1D	Reserved
0x1E	Processor is stopped. This value signals the processor has executed a STOP instruction. Although this is a multi-cycle mode since the ColdFire processor remains stopped until an interrupt or reset occurs, there are only two PST = 0x1E values recorded before the mode value is suppressed.
0x1F	Processor is halted. This value signals the processor has been halted. Although this is a multi-cycle mode since the ColdFire processor remains halted until a BDM go command is received or reset occurs, there are only two PST = 0x1F values recorded before the mode value is suppressed.

As PST and DDATA values are captured and loaded in the trace buffer, each entry is 6 bits in size so that the type of the entry can easily be determined when post-processing the PSTB. See Figure 19.

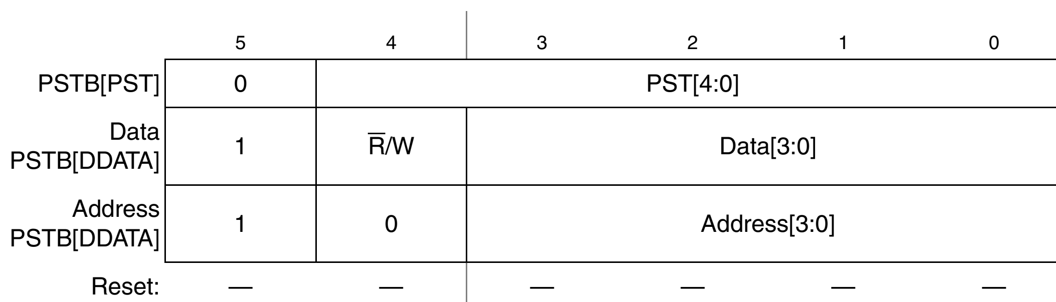


Figure 19. V1 PST/DDATA Trace Buffer Entry Format

5.3.1 PST/DDATA Example and the PST Trace Buffer (PSTB)

In this section, an example showing the behavior of the PST/DDATA functionality is detailed. Consider the following interrupt service routine that counts the interrupt, then negates the IRQ, performs a software IACK and then exits. This example is presented here since it exercises a considerable set of the PST/DDATA capabilities.

```
                      _isr:
01074: 46fc 2700           mov.w   &0x2700,%sr          # disable interrupts
01078: 2f08                mov.l   %a0,-(%sp)           # save a0
0107a: 2f00                mov.l   %d0,-(%sp)           # save d0
0107c: 302f 0008           mov.w   (8,%sp),%d0          # load format/vector word
01080: e488                lsr.l   &2,%d0               # align vector number
01082: 0280 0000 00ff      andi.l  &0xff,%d0            # isolate vector number
01088: 207c 0080 1400      mov.l   &int_count,%a0       # base of interrupt counters

                         _isr_entry1:
0108e: 52b0 0c00           addq.l  &1,(0,%a0,%d0.l*4)  # count the interrupt
01092: 11c0 a021           mov.b   %d0,IGCR0+1.w        # negate the irq
01096: 1038 a020           mov.b   IGCR0.w,%d0          # force the write to complete
0109a: 4e71                nop                          # synchronize the pipelines
0109c: 71b8 ffe0           mvz.b   SWIACK.w,%d0         # software iack: pending irq?
010a0: 0c80 0000 0041      cmpi.l  %d0,&0x41            # level 7 or none pending?
```

```
010a6: 6f08             ble.b   _isr_exit          # yes, then exit
010a8: 52b9 0080 145c   addq.l  &1,swiack_count    # increment the swiack count
010ae: 60de             bra.b   _isr_entry1        # continue at entry1

                      _isr_exit:
010b0: 201f             mov.l   (%sp)+,%d0         # restore d0
010b2: 205f             mov.l   (%sp)+,%a0         # restore a0
010b4: 4e73             rte                        # exit
```

This ISR executes mostly as straight-line code: there is a single conditional branch @ PC = 0x10A6, which is taken in this example. The following description includes the PST and DDATA values generated as this code snippet executes. In this example, the CSR setting enables the display of 2-byte branch addresses and both peripheral bus read and write operands are being traced. The sequence begins with an interrupt exception:

```
interrupt exception occurs @ pc = 5432 while in user mode
                                                   # pst   = 1c, 1c, 05, 0d
                                                   # ddata = 2a, 23, 28, 20
                                                   #         trg_addr = 083a << 1
                                                   #         trg_addr = 1074
                      _isr:
01074: 46fc 2700        mov.w   &0x2700,%sr        # pst   = 01
01078: 2f08             mov.l   %a0,-(%sp)         # pst   = 01
0107a: 2f00             mov.l   %d0,-(%sp)         # pst   = 01
0107c: 302f 0008        mov.w   (8,%sp),%d0        # pst   = 01
01080: e488             lsr.l   &2,%d0             # pst   = 01
01082: 0280 0000 00ff   andi.l  &0xff,%d0          # pst   = 01
01088: 207c 0080 1400   mov.l   &int_count,%a0     # pst   = 01
0108e: 52b0 0c00        addq.l  &1,(0,%a0,%d0.l*4) # pst   = 01
01092: 11c0 a021        mov.b   %d0,IGCR0+1.w      # pst   = 01, 08
                                                   # ddata = 30, 30
                                                   #         wdata.b = 0x00
01096: 1038 a020        mov.b   IGCR0.w,%d0        # pst   = 01, 08
                                                   # ddata = 28, 21
                                                   #         rdata.b = 0x18
0109a: 4e71             nop                        # pst   = 01
0109c: 71b8 ffe0        mvz.b   SWIACK.w,%d0       # pst   = 01, 08
                                                   # ddata = 20, 20
                                                   #         rdata.b = 0x00
010a0: 0c80 0000 0041   cmpi.l  %d0,&0x41          # pst   = 01
010a6: 6f08             ble.b   _isr_exit          # pst   = 05 (taken branch)
010b0: 201f             mov.l   (%sp)+,%d0         # pst   = 01
010b2: 205f             mov.l   (%sp)+,%a0         # pst   = 01
010b4: 4e73             rte                        # pst   = 07, 03, 05, 0d
                                                   # ddata = 29, 21, 2a, 22
                                                   #trg_addr = 2a19 << 1
                                                   #trg_addr = 5432
```

As the PSTs are compressed, the resulting stream of 6-bit hexadecimal entries is loaded into consecutive locations in the PST trace buffer:

```
PSTB[*] =1c, 1c, 05, 0d,  // interrupt exception
         2a, 23, 28, 20,  // branch target addr = 1074
         19, 08, 30, 30,  // 9 sequential insts, write byte
         01, 08, 28, 21,  // 1 sequential inst,  read byte
         12, 08, 20, 20,  // 2 sequential insts, read byte
         01, 05, 12,      // 1 + taken_branch + 2 sequential
         07, 03, 05, 0d,  // rte, entry into user mode
```

```
29, 21, 2a, 22    // branch target addr = 5432
```

Architectural studies on the compression algorithm were used to determine an appropriate size for the PST trace buffer. Using a suite of ten MCU benchmarks, a 64-entry PSTB was found to capture an average ‘window of time’ of 520 processor cycles with program trace using 2-byte addresses enabled.

5.3.2 PST/DDATA Definition for ISA_C

This section specifies the ColdFire processor and debug module’s generation of the processor status (PST) and debug data (DDATA) output on an instruction basis. In general, the PST/DDATA output for an instruction is defined as follows:

PST = 0x01, {PST = [0x0{89B}], DDATA= operand}

where the {...} definition is optional operand information defined by the setting of the CSR.

The CSR provides capabilities to display peripheral bus operands based on reference type (read, write, or both). A PST value (0x08, 0x09, or 0x0B) identifies the size and presence of valid data to follow on the DDATA output (one, two, or four bytes). Additionally, for certain change-of-flow branch instructions, the CSR[BTB] bit provides the capability to display the target instruction address on the DDATA output (two or three bytes) using a PST value of (0x0D or 0x0E).

5.3.2.1 User Instruction Set

Table 20 shows the PST/DDATA specification for user-mode instructions. Rn represents any {Dn, An} register. In this definition, the ‘y’ suffix generally denotes the source, and ‘x’ denotes the destination operand. For a given instruction, the optional operand data is displayed only for those effective addresses referencing memory. The ‘DD’ nomenclature refers to the DDATA outputs.

Table 20. PST/DDATA Specification for User-Mode Instructions

Instruction	Operand Syntax	PST/DDATA
add.l	<ea>y,Rx	PST = 0x01, {PST = 0x0B, DD = source operand}
add.l	Dy,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
addi.l	#imm,Dx	PST = 0x01
addq.l	#imm,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
addx.l	Dy,Dx	PST = 0x01
and.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
and.l	Dy,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
andi.l	#imm,Dx	PST = 0x01
asl.l	{Dy,#imm},Dx	PST = 0x01
asr.l	{Dy,#imm},Dx	PST = 0x01
bcc.{b,w,l}		if taken, then PST = 0x05, else PST = 0x01
bchg	#imm,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
bchg	Dy,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
bclr	#imm,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}

Table 20. PST/DDATA Specification for User-Mode Instructions (continued)

Instruction	Operand Syntax	PST/DDATA
bclr	Dy,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
bitrev.l	Dx	PST = 0x01
bra.{b,w,l}		PST = 0x05
bset	#imm,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
bset	Dy,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
bsr.{b,w,l}		PST = 0x05, {PST = 0x0B, DD = destination operand}
btst	#imm,<ea>x	PST = 0x01, {PST = 0x08, DD = source operand}
btst	Dy,<ea>x	PST = 0x01, {PST = 0x08, DD = source operand}
byterev.l	Dx	PST = 0x01
clr.b	<ea>x	PST = 0x01, {PST = 0x08, DD = destination operand}
clr.l	<ea>x	PST = 0x01, {PST = 0x0B, DD = destination operand}
clr.w	<ea>x	PST = 0x01, {PST = 0x09, DD = destination operand}
cmp.b	<ea>y,Rx	PST = 0x01, {PST = 0x08, DD = source operand}
cmp.l	<ea>y,Rx	PST = 0x01, {PST = 0x0B, DD = source operand}
cmp.w	<ea>y,Rx	PST = 0x01, {PST = 0x09, DD = source operand}
cmpi.b	#imm,Dx	PST = 0x01
cmpi.l	#imm,Dx	PST = 0x01
cmpi.w	#imm,Dx	PST = 0x01
divs.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
divs.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
divu.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
divu.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
eor.l	Dy,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
eori.l	#imm,Dx	PST = 0x01
ext.l	Dx	PST = 0x01
ext.w	Dx	PST = 0x01
extb.l	Dx	PST = 0x01
ff1.l	Dx	PST = 0x01
illegal		PST = 0x01[3]
jmp	<ea>x	PST = 0x05, {PST = [0x0{DE}], DD = target address} [1]
jsr	<ea>x	PST = 0x05, {PST = [0x0{DE}], DD = target address}, {PST = 0xB , DD = destination operand}[1]
lea	<ea>y,Ax	PST = 0x01
link.w	Ay,#imm	PST = 0x01, {PST = 0x0B, DD = destination operand}
lsl.l	{Dy,#imm},Dx	PST = 0x01

Table 20. PST/DDATA Specification for User-Mode Instructions (continued)

Instruction	Operand Syntax	PST/DDATA
lsr.l	{Dy,#imm},Dx	PST = 0x01
mov3q.l	<ea>y,Rx	PST = 0x01, {PST = 0x0B, DD = source operand}
move.b	<ea>y,<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
move.l	<ea>y,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
move.w	<ea>y,<ea>x	PST = 0x01, {PST = 0x09, DD = source}, {PST = 0x09, DD = destination}
move.w	CCR,Dx	PST = 0x01
move.w	{Dy,#imm},CCR	PST = 0x01
movem.l	#list,<ea>x	PST = 0x01, {PST = 0x0B, DD = destination},... [2]
movem.l	<ea>y,#list	PST = 0x01, {PST = 0x0B, DD = source},... [2]
moveq	#imm,Dx	PST = 0x01
muls.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
muls.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
mulu.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
mulu.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
mvs.b	<ea>y,Dx	PST = 0x01, {PST = 0x08, DD = source operand}
mvs.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
mvz.b	<ea>y,Dx	PST = 0x01, {PST = 0x08, DD = source operand}
mvz.w	<ea>y,Dx	PST = 0x01, {PST = 0x09, DD = source operand}
neg.l	Dx	PST = 0x01
negx.l	Dx	PST = 0x01
nop		PST = 0x01
not.l	Dx	PST = 0x01
or.l	<ea>y,Dx	PST = 0x01, {PST = 0x0B, DD = source operand}
or.l	Dy,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
ori.l	#imm,Dx	PST = 0x01
pea	<ea>y	PST = 0x01, {PST = 0x0B, DD = destination operand}
pulse		PST = 0x04
rems.l	<ea>y,Dx:Dw	PST = 0x01, {PST = 0x0B, DD = source operand}
remu.l	<ea>y,Dx:Dw	PST = 0x01, {PST = 0x0B, DD = source operand}
rts		PST = 0x01, {PST = 0x0B, DD = source operand}, PST = 0x05, {PST = [0x0{DE}], DD = target address}
sats	Dx	PST = 0x01
scc	Dx	PST = 0x01
sub.l	<ea>y,Rx	PST = 0x01, {PST = 0x0B, DD = source operand}
sub.l	Dy,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}

Table 20. PST/DDATA Specification for User-Mode Instructions (continued)

Instruction	Operand Syntax	PST/DDATA
subi.l	#imm,Dx	PST = 0x01
subq.l	#imm,<ea>x	PST = 0x01, {PST = 0x0B, DD = source}, {PST = 0x0B, DD = destination}
subx.l	Dy,Dx	PST = 0x01
swap	Dx	PST = 0x01
tas	<ea>x	PST = 0x01, {PST = 0x08, DD = source}, {PST = 0x08, DD = destination}
trap	#imm	PST = 0x01[3]
trapf		PST = 0x01
tst.b	<ea>x	PST = 0x01, {PST = 0x08, DD = source operand}
tst.l	<ea>x	PST = 0x01, {PST = 0x0B, DD = source operand}
tst.w	<ea>x	PST = 0x01, {PST = 0x09, DD = source operand}
unlk	Ax	PST = 0x01, {PST = 0x0B, DD = destination operand}
wddata.b	<ea>y	PST = 0x04, {PST = 0x08, DD = source operand
wddata.l	<ea>y	PST = 0x04, {PST = 0x0B, DD = source operand
wddata.w	<ea>y	PST = 0x04, {PST = 0x09, DD = source operand

1 For JMP and JSR instructions, the optional target instruction address is displayed only for those effective address fields defining variant addressing modes. This includes the following <ea>x values: (An), (d16,An), (d8,An,Xi), (d8,PC,Xi).

2 For move multiple instructions (MOVEM), the processor automatically generates line-sized transfers if the operand address reaches a 0-modulo-16 boundary and there are four or more registers to be transferred. For these line-sized transfers, the operand data is never captured nor displayed, regardless of the CSR value.
The automatic line-sized burst transfers are provided to maximize performance during these sequential memory access operations.

3 During normal exception processing, the PST output is driven to a 0x1C 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 = 0x1C, 0x1C,
                           {PST = 0x0B, DD = destination},             // stack frame
                           {PST = 0x0B, DD = destination},             // stack frame
                           {PST = 0x0B, DD = source},                  // vector read
                            PST = 0x05,{PST = [0x0{DE}], DD = target}  // handler PC
```

The PST/DDATA specification for the reset exception is shown below:

```
Exception Processing        PST = 0x1C, 0x1C,
                            PST = 0x05,{PST = [0x0{DE}], DD = target}  // handler PC
```

The initial references at address 0 and 4 are never captured nor displayed, because these accesses are treated as instruction fetches. For all types of exception processing, the PST = 0x1C value is driven for two trace buffer entries.

Table 21 shows the PST/DDATA specification for multiply-accumulate instructions.

Table 21. PST/DDATA Specification for {E}MAC Instructions

Instruction	Operand Syntax	PST/DDATA
mac.l	Ry,Rx,Accx	PST = 0x01
mac.l	RyRx,<ea>,Rw,Accx	PST = 0x01, {PST = 0x0B, DD = source operand}
mac.w	Ry,Rx,Accx	PST = 0x01
mac.w	Ry,Rx,<ea>,Rw,Accx	PST = 0x01, {PST = 0x0B, DD = source operand}
move.l	<ea>y,Accx	PST = 0x01
move.l	Accy,Accx	PST = 0x01
move.l	<ea>y,MACR	PST = 0x01
move.l	<ea>y,MASK	PST = 0x01
move.l	<ea>y,Accext01	PST = 0x01
move.l	<ea>y,Accext23	PST = 0x01
move.l	Accy,Rx	PST = 0x01
move.l	MACSR,CCR	PST = 0x01
move.l	MACSR,Rx	PST = 0x01
move.l	MASK,Rx	PST = 0x01
move.l	Accext01,Rx	PST = 0x01
move.l	Accext23,Rx	PST = 0x01
msac.l	Ry,Rx,Accx	PST = 0x01
msac.l	Ry,Rx,<ea>,Rw,Accx	PST = 0x01, {PST = 0x0B, DD = source operand}
msac.w	Ry,Rx,Accx	PST = 0x01
msac.w	Ry,Rx,<ea>,Rw,Accx	PST = 0x01, {PST = 0x0B, DD = source operand}

5.3.2.2 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 shown in Table 22.

Table 22. PST/DDATA Specification for Supervisor-Mode Instructions

Instruction	Operand Syntax	PST/DDATA
halt		PST = 0x01, PST = 0x1F
move.w	SR,Dx	PST = 0x01
move.l	USP,Ax	PST = 0x01
move.w	{Dy,#imm},SR	PST = 0x01, {PST = 0x3}
move.w	Ay,USP	PST = 0x01
movec	Ry,Rc	PST = 0x01

Table 22. PST/DDATA Specification for Supervisor-Mode Instructions (continued)

Instruction	Operand Syntax	PST/DDATA
rte		PST = 0x07, {PST = 0x0B, DD = source operand}, {PST = 3}, PST =0x0B, DD =source operand}, PST = 0x05, {[PST = 0x0{DE}], DD = target address}
stldsr.w	#imm	PST = 0x01, {PST = 0xA, DD = destination operand, PST = 0x3}
stop	#imm	PST = 0x01, PST = 0x1E
wdebug	<ea>y	PST = 0x01, {PST = 0x0B, DD = source, PST = 0x0B, DD = source}

The move-to-SR, STLDSR, and RTE instructions include an optional PST = 0x3 value, indicating an entry into user mode.

Similar to the exception processing mode, the stopped state (PST = 0x1E) and the halted state (PST = 0x1F) display this status for two cycles while the ColdFire processor is in the given mode.

6 V1 ColdFire Core Hardware Details

This section includes details on the V1 ColdFire core's microarchitecture including pipeline organization and instruction execution times. Additionally, information of measured code density and performance is presented.

6.1 Core Microarchitecture

Recall the V1 core's pipeline structure was first discussed in Section 1.1.3, "Core Pipeline Organization" including the V1 platform block diagram shown in Figure 4. This section provides additional details on the core's hardware organization.

6.1.1 Core Pipeline Structure

A spatial pipeline diagram of the CF1 CPU is shown in Figure 20. The hardware structure of the operand execution pipeline is a two-stage pipeline featuring a traditional RISC datapath with a dual-read-ported register file (RGF) feeding an arithmetic/logic unit. In the spatial processor block diagram, the ALU/BSM structure represents the ALU (arithmetic/logic unit) and the BSM which provides the barrel shifter and a 'miscellaneous' execute engine. Recall in the V1 design, the two OEP pipeline stages have multiple functions:

- Decode & Select/Operand Cycle DSOC Cycle
- Address Generation/Execute Cycle AGEX Cycle

The pipeline stages for the IFP and OEP are shown at the far right in the diagram.

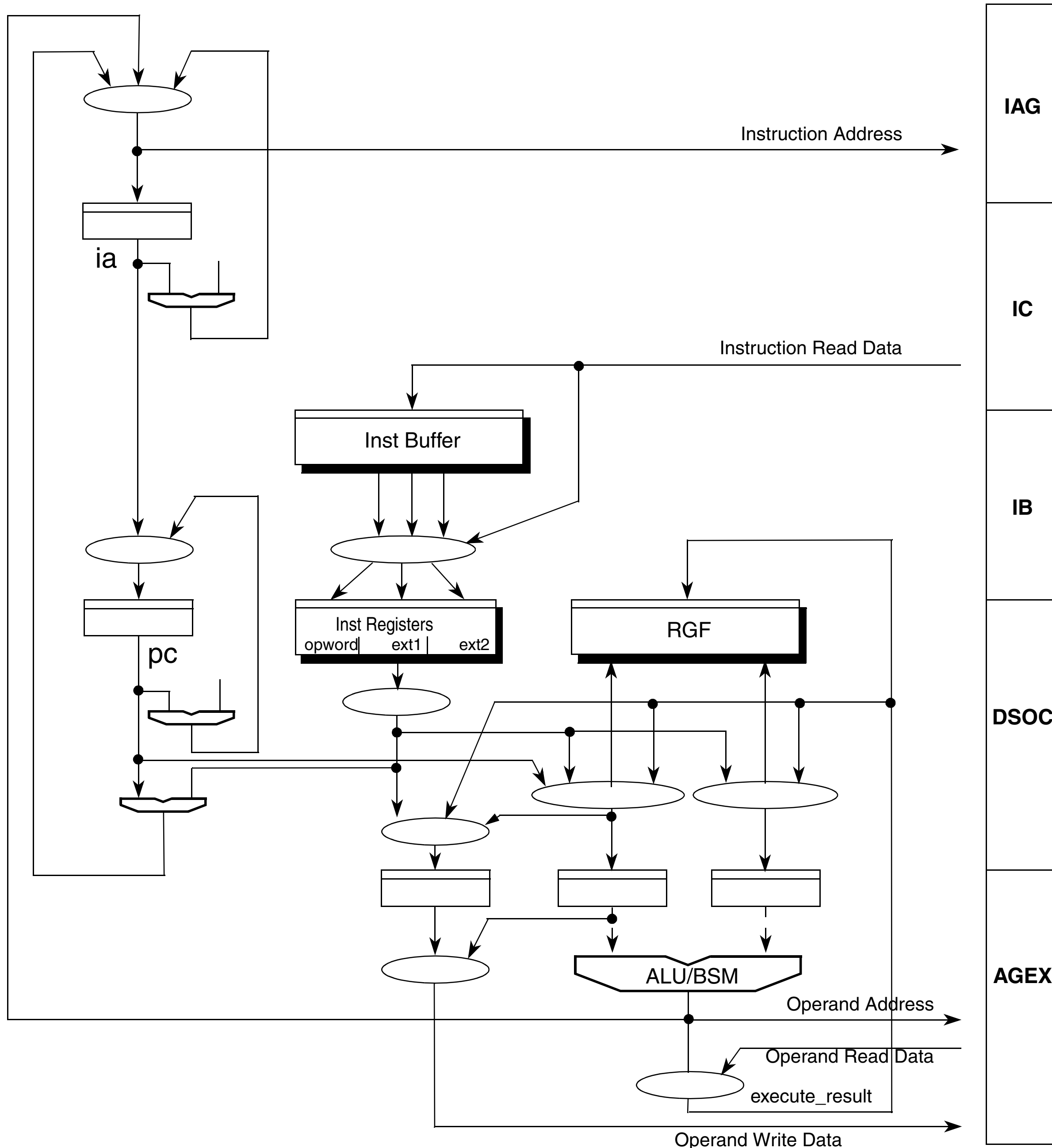


Figure 20. CF1 CPU Spatial Pipeline Diagram

As shown in Figure 4 and Figure 20, the CF1 CPU's operand execution pipeline (OEP) stages map directly onto the 2-stage pipelined platform bus. This structure allows the processor access to the platform memory modules for single-cycle pipelined reads and writes with a zero wait-state (as viewed in the system bus data phase stage) response. Consider the following mapping of pipeline stages of the processor's OEP and the platform's system bus as shown in Table 23 and the timing diagram of Figure 21.

Table 23. CF1 CPU Pipeline Stage Mapping

Reference Type	CF1 CPU Pipeline Stage	System Bus Pipeline Stage
inst fetch	1st Stage = IAG	1st Stage = platform address phase
	2nd Stage = IC	2nd Stage = platform data phase
read	1st Stage = AGEX	1st Stage = platform address phase
	2nd Stage = DSOC	2nd Stage = platform data phase
write	1st Stage = AGEX	1st Stage = platform address phase
	2nd Stage = DA	2nd Stage = platform data phase

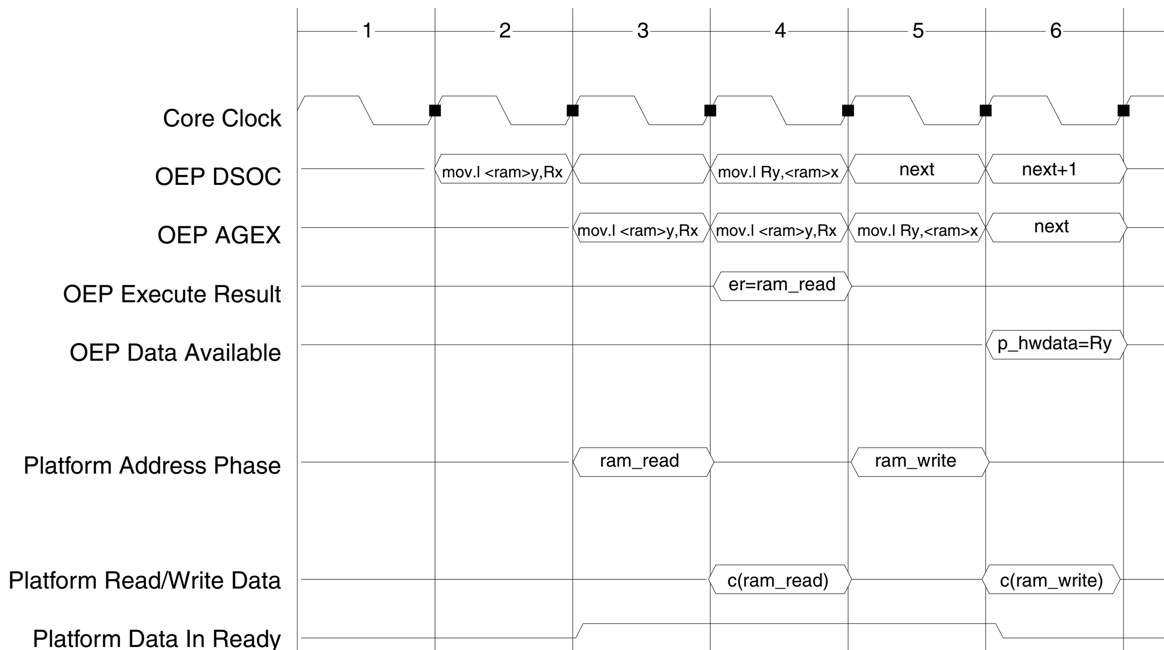


Figure 21. CF1 CPU & Platform System Bus Timing Diagram

The timing diagram shows a 2-instruction sequence accessing the platform RAM as it executes in the CPU’s OEP: a 32-bit load operation (mov.l <ram>y,Rx) followed by a 32-bit store operation (mov.l Ry,<ram>x). The load instruction reads from the platform RAM and the store instruction writes to this memory. For this sequence, the 32-bit load operation is a 2-cycle instruction and the store operation is a single-cycle instruction. The timing diagram shows the instructions progressing through the two stages of the operand execution pipeline (DSOC and AGEX stages) as well as the processor’s internal ‘execute_result’ bus and a virtual ‘data_available’ pipeline stage for store operations. Additionally, a subset of the platform bus signals are shown. Recall the platform bus implements a two-stage pipelined protocol with an address phase followed by a data phase. The following table associates the two instructions where their relative location in the processor’s pipeline and the platform bus in the timing diagram.

Table 24. CF1Core & Platform Bus Stage Association

Cycle	CPU_OEP DSOC Stage	CPU_OEP AGEX Stage	CPU_OEP Virtual Data Available	Platform Address Phase	Platform Data Phase
2	1st cycle: mov.l <ram>y,Rx				
3	2nd cycle: mov.l <ram>y,Rx	1st cycle: mov.l <ram>y,Rx		RAM read	
4	1st cycle: mov.l Ry,<mem>x	2nd cycle: mov.l <ram>y,Rx			RAM read
5	next inst	1st cycle: mov.l Ry,<ram>x		RAM write	
6	next+1 inst	next inst	1st cycle: mov.l Ry,<ram>x		RAM write

The instruction fetch pipeline prefetches instructions from platform memories using a 2-stage structure. For sequential prefetches, the next instruction address is generated by adding 4 to the last prefetch address. This function is performed during the IAG stage, and the resulting prefetch address gated onto the platform's bus (if there are no pending operand memory accesses which are assigned a higher priority). Once the prefetch address is driven onto the platform bus, the instruction fetch cycle accesses the appropriate memory and returns the instruction read data back to the IFP during the next cycle (IC stage). If the accessed data is not present in a platform memory or not immediately available, or an external access cycle is required, the IFP is stalled in the IC stage until the referenced data is available. As the prefetch data arrives in the IFP, it can be loaded into the FIFO instruction buffer, or gated directly into the operand execution pipeline.

The V1 design uses a simple static conditional branch prediction algorithm (forward assumed as not-taken, backward assumed as taken), and all change-of-flow operations are calculated by the OEP and the target instruction addresses fed back to the IFP. Two separate target address calculations are performed: one in the DSOC stage for conditional branches and another, the main ALU for operand generation in the AGEX stage, for all other change-of-flow instructions.

The IFP and OEP are decoupled by the FIFO instruction buffer, allowing instruction prefetching to occur with the available platform bus bandwidth not used for operand memory accesses. For the V1 design, the instruction buffer contains three 32-bit locations.

The following table presents the V1 execution times for the most-heavily used instruction types. For a complete listing of the instruction execution times, see Section 6.1.2, "Core Instruction Execution Times."

Table 25. V1 Instruction Execution Times

Instruction Mnemonic	Operation	V1 Execution Time
<op> Ry,Rx	register-to-register	1
mov.{b,w,l} <mem>y,Rx	8,16,32-bit load	2
mvs.{b,w} <mem>y,Rx	8,16-bit load with sign extension	2
mvz.{b,w} <mem>y,Rx	8,16-bit load with zero fill	2

Table 25. V1 Instruction Execution Times (continued)

Instruction Mnemonic	Operation	V1 Execution Time
mov.* Ry,<mem>x	store	1
mov.l <mem>y,<mem>x	memory-to-memory	2
<op> <mem>y,Rx	embedded-load	3
<op> Ry,<mem>x	read-modify-write	3
bsr, jsr <label>	subroutine call	3
rts	subroutine return	5
bra <label>	branch always	2
bcc <label> (forward, not taken) (forward, taken) (backward, not taken) (backward, taken)	conditional branch	 1 3 3 2

Consider the operation of the OEP for three basic classes of non-branch instructions:

```
Register-to-Register         op        Ry,Rx
Embedded-Load                op        <mem>y,Rx
Register-to-Memory (Store) move      Ry,<mem>x
```

For this discussion, a simplified OEP block diagram is utilized.

For simple register-to-register instructions, the first stage of the OEP performs the instruction decode and fetching of the required register operands (OC) from the dual-ported register file, while the actual instruction execution is performed in the second stage (EX) in one of the “execute engines” (e.g., ALU, barrel shifter, divider, {E}MAC}. There are no operand memory accesses associated with this class of instructions, and the execution time typically is a single machine cycle. See Figure 22.

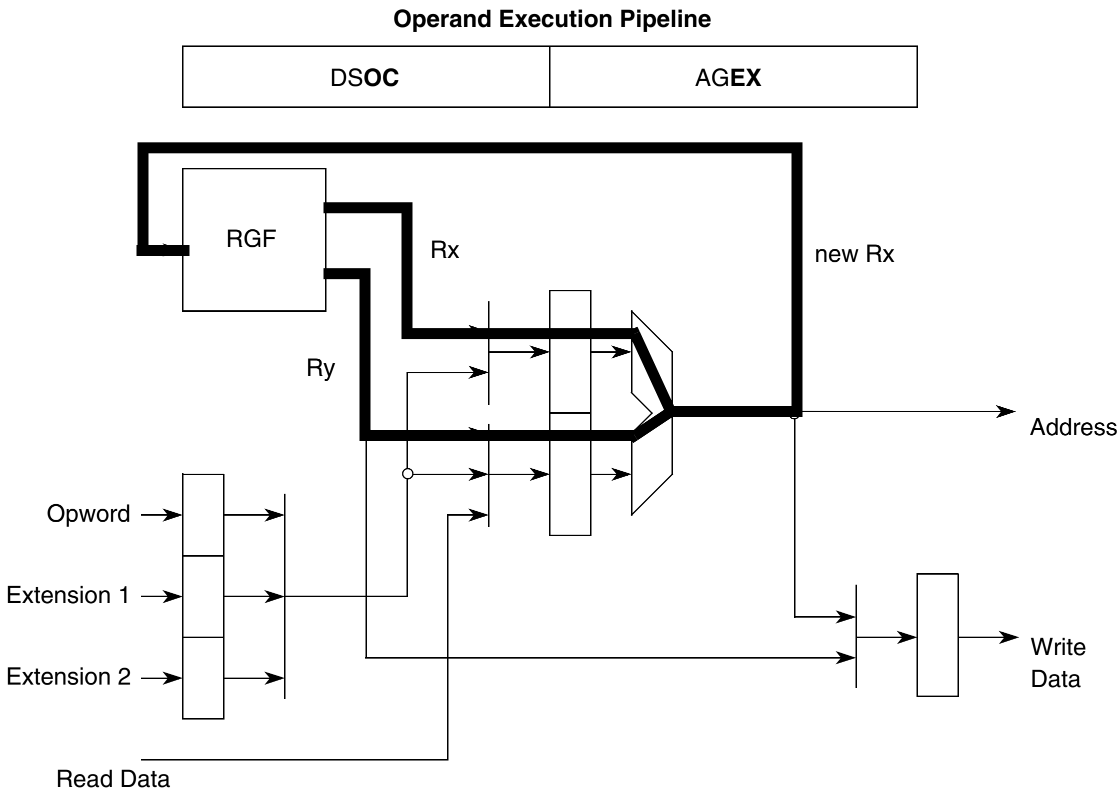


Figure 22. V1 OEP Register-to-Register Instructions

For memory-to-register (embedded-load) instructions, the instruction is effectively staged through the OEP twice with a basic execution time of three cycles. First, the instruction is decoded and the components of the operand address (base register from the RGF and displacement) are selected (DS). Second, the operand effective address is generated using the ALU execute engine (AG). Third, the memory read operand is fetched from the platform bus, while any required register operand is simultaneously fetched (OC) from the RGF. Finally, in the fourth cycle, the instruction is executed (EX). The following example in Figure 24 shows an effective address of the form <ea>y = (d16,Ay), i.e., a 16-bit signed displacement added to a base register Ay. Recall the heavily-used load and load with sign extension or zero fill instructions are optimized to support a 2-cycle execution time.

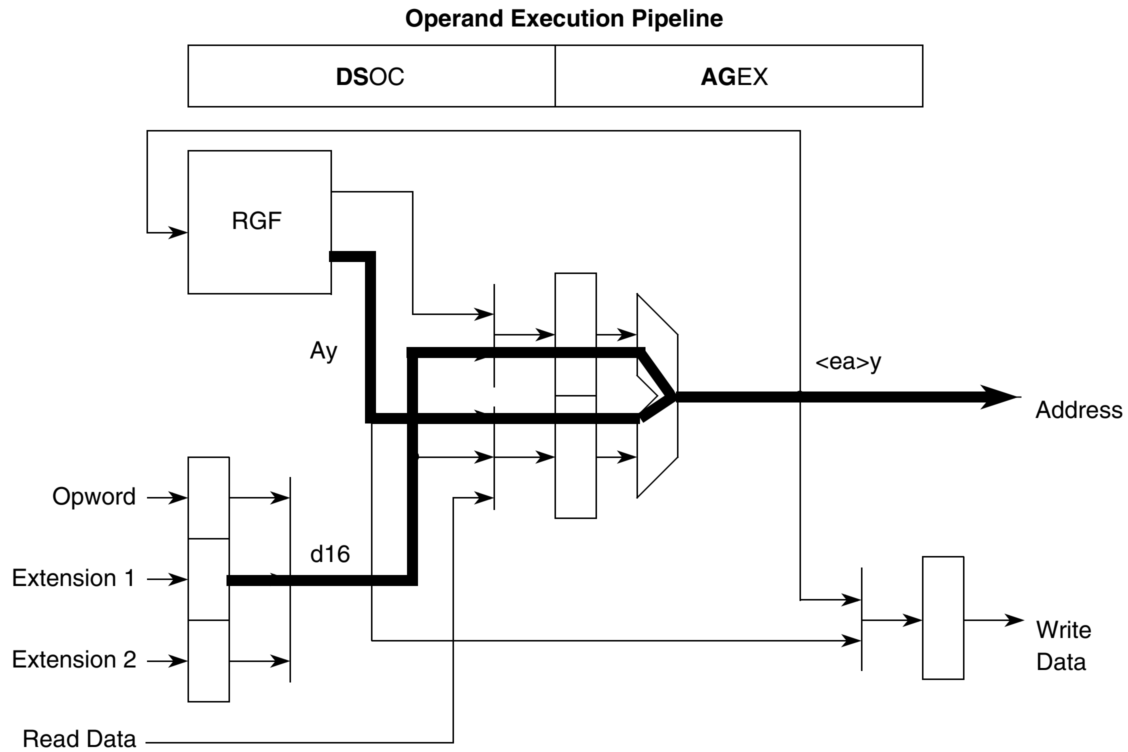


Figure 23. V1 OEP Register-to-Register Instructions (Part 1)

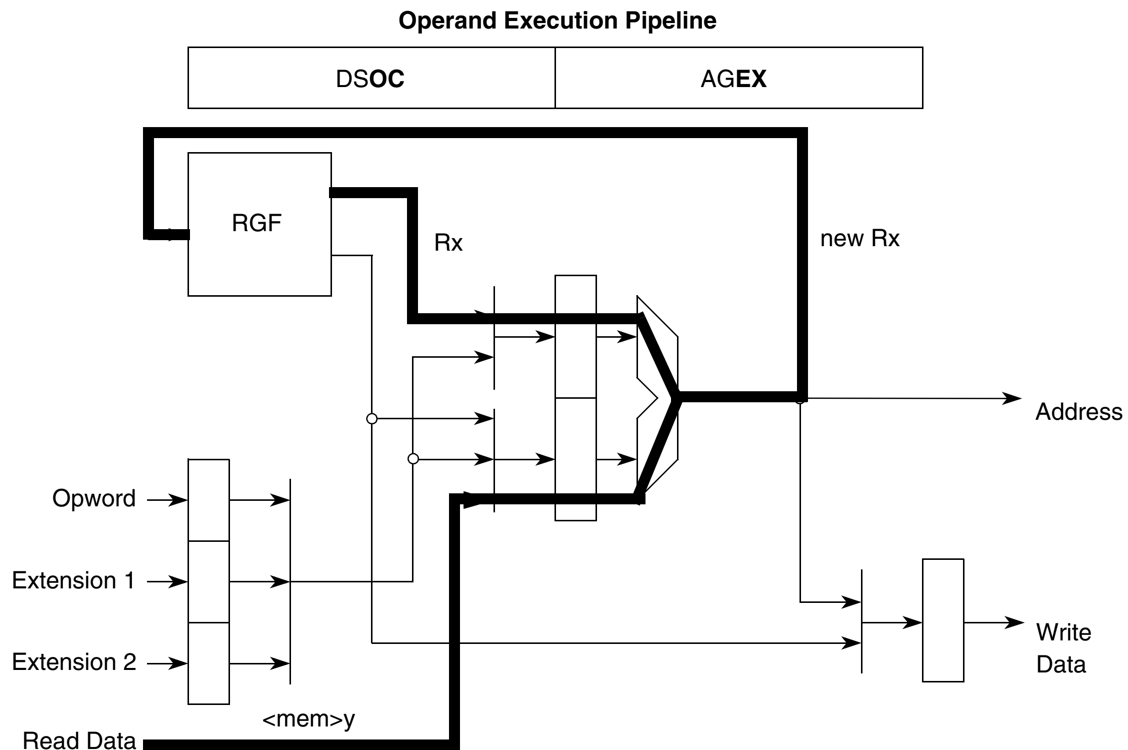


Figure 24. V1 OEP Embedded-Load Instructions (Part 2)

For register-to-memory (store) operations, the stage functions (DS/OC, AG/EX) are effectively performed simultaneously allowing single-cycle execution. See Figure 25 where again the effective address is of the form <ea>x = (d16,Ax), i.e., a 16-bit signed displacement added to a base register Ax.

For read-modify-write instructions, the pipeline effectively combines an embedded-load with a store operation for a three-cycle execution time.

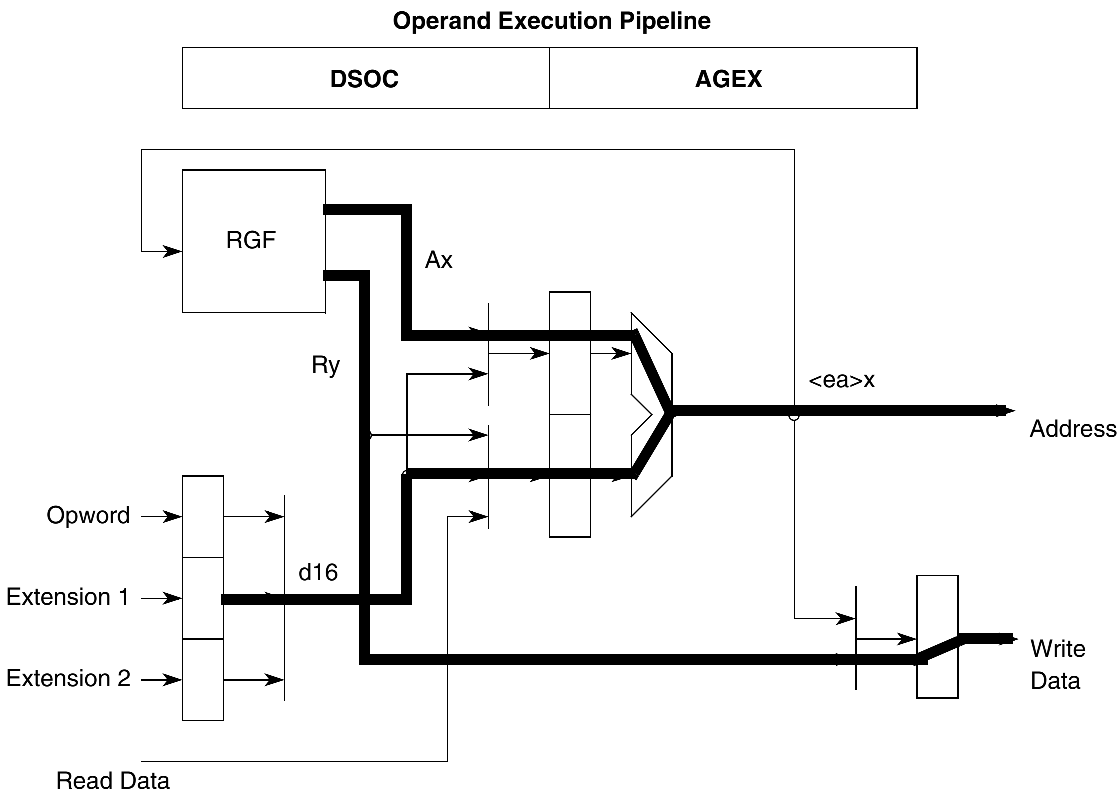


Figure 25. V2 OEP Register-to-Memory Instructions

6.1.2 Core Instruction Execution Times

This section presents processor instruction execution times in terms of processor core clock cycles. The number of operand references for each instruction is enclosed in parentheses following the number of processor clock cycles. Each timing entry is presented as C(R/W) where:

- C is the number of processor clock cycles, including all applicable operand fetches and writes, and all internal core cycles required to complete the instruction execution.
- R/W is the number of operand reads (R) and writes (W) required by the instruction. An operation performing a read-modify-write function is denoted as (1/1).

This section includes the assumptions concerning the timing values and the execution time details.

6.1.2.1 Timing Assumptions

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

1. The 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 IFP to supply opwords and/or extension words.

   The OEP does not experience any sequence-related pipeline stalls. 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 decode and select/operand fetch cycle (DSOC) 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.
2. 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.
3. 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.

   The processor core decomposes misaligned operand references into a series of aligned accesses as shown in Table 26.

Table 26. Misaligned Operand References

address[1:0]	Size	Bus Operations	Additional C(R/W)
01 or 11	Word	Byte, Byte	2(1/0) if read 1(0/1) if write
01 or 11	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

6.1.2.2 MOVE Instruction Execution Times

Table 27 lists execution times for MOVE.{B,W} instructions; Table 28 lists timings for MOVE.L; Table 29 lists the timings for miscellaneous move instructions.

NOTE

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.

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

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

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

Table 27. 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)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1))	3(1/1)
(Ay)+	2(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1))	3(1/1)
-(Ay)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1))	3(1/1)
(d16,Ay)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,Ay,Xi*SF)	3(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
xxx.w	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
xxx.l	2(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(d16,PC)	2(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,PC,Xi*SF)	3(1/0)	4(1/1)	4(1/1)	4(1/1))	—	—	—
#xxx	1(0/0)	3(0/1)	3(0/1)	3(0/1)	1(0/1)	—	—

Table 28. 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)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(Ay)+	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
-(Ay)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(d16,Ay)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,Ay,Xi*SF)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
xxx.w	2(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
xxx.l	2(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(d16,PC)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,PC,Xi*SF)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
#xxx	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

Table 29. Miscellaneous MOVE Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	xxx.wl	#<xxx>
moveq.l	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
mov3q	#imm,<ea>x	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
mvs	<ea>y,Dx	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)	1(0/0)
mvz	<ea>y,Dx	1(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)	1(0/0)

6.1.2.3 Standard One Operand Instruction Execution Times

Table 30. One Operand Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
bitrev	Dx	1(0/0)	—	—	—	—	—	—	—
byterev	Dx	1(0/0)	—	—	—	—	—	—	—
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)	—	—	—	—	—	—	—
ff1	Dx	1(0/0)	—	—	—	—	—	—	—
neg.l	Dx	1(0/0)	—	—	—	—	—	—	—
negx.l	Dx	1(0/0)	—	—	—	—	—	—	—
not.l	Dx	1(0/0)	—	—	—	—	—	—	—
sats.l	Dx	1(0/0)	—	—	—	—	—	—	—
scc	Dx	1(0/0)	—	—	—	—	—	—	—
swap	Dx	1(0/0)	—	—	—	—	—	—	—
tas.b	<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
tst.b	<ea>	1(0/0)	2(1/0)	2(1/0)	2(1/0)	2(1/0)	3(1/0)	2(1/0)	1(0/0)
tst.w	<ea>	1(0/0)	2(1/0)	2(1/0)	2(1/0)	2(1/0)	3(1/0)	2(1/0)	1(0/0)
tst.l	<ea>	1(0/0)	2(1/0)	2(1/0)	2(1/0)	2(1/0)	3(1/0)	2(1/0)	1(0/0)

6.1.2.4 Standard Two Operand Instruction Execution Times

Table 31. Two Operand Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xn*SF) (d8,PC,Xn*SF)	xxx.wl	#xxx
add.l	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
add.l	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
addi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
addq.l	#imm,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
addx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—
and.l	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
and.l	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(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)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
bchg	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
bclr	Dy,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
bclr	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
bset	Dy,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
bset	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
btst	Dy,<ea>	2(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	—
btst	#imm,<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
cmp.b	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
cmp.w	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
cmp.l	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
cmpi.b	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
cmpi.w	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
cmpi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
divs.w	<ea>,Dx	20(0/0)	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)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
divu.l	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
eor.l	Dy,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
eori.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
lea	<ea>,Ax	—	1(0/0)	—	—	1(0/0)	2(0/0)	1(0/0)	—

Table 31. Two Operand Instruction Execution Times (continued)

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xn*SF) (d8,PC,Xn*SF)	xxx.wl	#xxx
lsl.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
lsr.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
or.l	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
or.l	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
ori.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
rems.l	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
remu.l	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
sub.l	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
sub.l	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
subi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
subq.l	#imm,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
subx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—

6.1.2.5 Miscellaneous Instruction Execution Times

Table 32. Miscellaneous Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
link.w	Ay,#imm	2(0/1)	—	—	—	—	—	—	—
move.l	Ay,USP	3(0/0)	—	—	—	—	—	—	—
move.l	USP,Ax	3(0/0)	—	—	—	—	—	—	—
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	7(0/0)	—	—	—	—	—	—	7(0/0) [2]
movec	Ry,Rc	9(0/1)	—	—	—	—	—	—	—
movem.l	<ea>,&list	—	1+n(n/0)	—	—	1+n(n/0)	—	—	—
movem.l	&list,<ea>	—	1+n(0/n)	—	—	1+n(0/n)	—	—	—
nop		3(0/0)	—	—	—	—	—	—	—
pea	<ea>	—	2(0/1)	—	—	2(0/1) [4]	3(0/1) [5]	2(0/1)	—
pulse		1(0/0)	—	—	—	—	—	—	—
stldsr	#imm	—	—	—	—	—	—	—	5(0/1)
stop	#imm	—	—	—	—	—	—	—	3(0/0) [3]

Table 32. Miscellaneous Instruction Execution Times (continued)

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
trap	#imm	—	—	—	—	—	—	—	15(1/2)
tpf		1(0/0)	—	—	—	—	—	—	—
tpf.w		1(0/0)	—	—	—	—	—	—	—
tpf.l		1(0/0)	—	—	—	—	—	—	—
unlk	Ax	2(1/0)	—	—	—	—	—	—	—
wddata	<ea>	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	—
wdebug	<ea>	—	5(2/0)	—	—	5(2/0)	—	—	—

[1]n is the number of registers moved by the MOVEM opcode.
[2]If a MOVE.W #imm,SR instruction is executed and imm[13] = 1, the execution time is 1(0/0).
[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,Xn*SF).

6.1.2.6 Branch Instruction Execution Times

Table 33. 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
bra		—	—	—	—	2(0/1)	—	—	—
bsr		—	—	—	—	3(0/1)	—	—	—
jmp	<ea>	—	3(0/0)	—	—	3(0/0)	4(0/0)	3(0/0)	—
jsr	<ea>	—	3(0/1)	—	—	3(0/1)	4(0/1)	3(0/1)	—
rte		—	—	10(2/0)	—	—	—	—	—

Table 34. Bcc Instruction Execution Times

Opcode	Forward Taken	Forward Not Taken	Backward Taken	Backward Not Taken
bcc	3(0/0)	1(0/0)	2(0/0)	3(0/0)

6.2 Power Dissipation

Power analysis work for the V1 core and platform is currently underway and results will be supplied at a later date. It should be noted that there are multiple dimensions to address power dissipation. These include:

- Capabilities of the underlying process technology, like the use of a low-voltage, low-power technology

- Implementation of multiple low-power stop and wait modes at the chip level, typically including clock and power management techniques
- Aggressive clock gating throughout the design

6.3 Code Density and Performance

6.3.1 Code Density

In this section, preliminary results from a code density study is presented. For this initial work, seven ‘toy’ benchmarks were compiled for both S08 and ColdFire targets. The benchmarks were compiled with three different structures:

1. In one form, the variables were typed simply as `int`. For the S08, this translates into a 16-bit variable, while it is a 32-bit value for the ColdFire compilers. This is labelled as ‘int vs. int’ in the data tables.
2. In the second form, the variables were specifically typed as `short`, which are 16-bit variables for both architectures. This is labelled as ‘short vs. short’ in the tables.
3. In the third form, the variables were specifically types as `char`, which are 8-bit variables for both architectures. This is labelled as ‘char vs. char’ in the tables.

Multiple compilers were studied. For the S08, CodeWarrior V5.1 was used exclusively, while three ColdFire compilers were studied. These are simply labelled as compilers “CFx, CFy, CFz”. In addition, the ColdFire compilers targeted both ISA_A (the original instruction set architecture) and ISA_C, the ISA implemented in the V1 core. Note the ColdFire compiler “x” only supported the ISA_A target.

A summary of the relative code sizes is presented in Table 35 and Table 36. For this table, the smaller the number, the smaller the code size and the better the code density. The S08 ‘int vs. int’ data is used as the reference throughout this analysis.

Table 35. S08 vs. ColdFire (ISA_C) Code Size

Benchmark	int vs. int				short vs. short				char vs. char			
	S08	CFx	CFy	CFz	S08	CFx	CFy	CFz	S08	CFx	CFy	CFz
ISA Target	S08	ISA_A	ISA_C	ISA_C	S08	ISA_A	ISA_C	ISA_C	S08	ISA_A	ISA_C	ISA_C
bit	1.00	1.11	0.89	0.92	1.00	1.21	0.97	0.95	0.67	1.37	0.92	0.95
crc	1.00	0.33	0.35	0.36	1.00	0.43	0.50	0.42	0.38	0.57	0.45	0.47
init	1.00	0.77	0.58	0.73	1.00	0.85	0.54	0.92	0.84	0.71	0.51	0.63
max	1.00	0.56	0.48	0.56	1.00	0.78	0.48	0.63	1.12	0.77	0.54	0.85
rotate	1.00	0.90	0.59	0.80	1.00	0.98	0.59	0.82	0.86	0.91	0.64	0.79
search	1.00	0.73	0.64	0.61	1.00	0.97	0.64	0.79	0.97	0.58	0.55	0.68
sort	1.00	0.65	0.51	0.63	1.00	0.71	0.55	0.68	0.44	1.02	0.59	0.82
GMEAN	1.00	0.68	0.56	0.63	1.00	0.81	0.59	0.72	0.71	0.81	0.58	0.72

The final row is the geometric mean across all seven benchmarks.

Table 36. S08 vs. ColdFire (ISA_A) Code Size

Benchmark	int vs. int				short vs. short				char vs. char			
	S08	CFx	CFy	CFz	S08	CFx	CFy	CFz	S08	CFx	CFy	CFz
ISA Target	S08	ISA_A	ISA_A	ISA_A	S08	ISA_A	ISA_A	ISA_A	S08	ISA_A	ISA_A	ISA_A
bit	1.00	1.11	0.92	0.97	1.00	1.21	0.97	1.11	0.67	1.37	0.95	0.92
crc	1.00	0.33	0.37	0.39	1.00	0.43	0.48	0.57	0.38	0.57	0.49	0.54
init	1.00	0.77	0.58	0.73	1.00	0.85	0.58	1.08	0.84	0.71	0.76	0.89
max	1.00	0.56	0.51	0.56	1.00	0.78	0.63	0.81	1.12	0.77	0.69	0.92
rotate	1.00	0.90	0.59	0.84	1.00	0.98	0.71	0.82	0.86	0.91	0.91	0.88
search	1.00	0.73	0.64	0.61	1.00	0.97	0.85	0.91	0.97	0.58	0.76	0.88
sort	1.00	0.65	0.51	0.63	1.00	0.71	0.63	0.80	0.44	1.02	0.67	0.88
GMEAN	1.00	0.68	0.57	0.65	1.00	0.81	0.68	0.85	0.71	0.81	0.73	0.83

There are a number of observations shown in this data:

- The S08 compiler produces its smallest code image when the variables are specifically typed as `char`. This is likely a function of both the compiler and the underlying instruction set architecture.
- The ColdFire compilers produce their best code when the variables are typed as `int`, as expected. Additionally, the differences between ISA_A and ISA_C are minimal for the `int` code base.
- For ColdFire, the `short` and `char` code bases have varying degrees of optimization compared to the `int` code. Compiler y produces the densest code image for the `short` and `char` code, followed by compiler z and finally compiler x. For the ISA_C target, the relative differences between the three code bases for compiler y are relatively small.
- As expected, the ISA_C target for compilers y and z shows more improvement for the `short` and `char` code bases, based on the instructions added in this revision.
- In general for this benchmark set, the ColdFire compilers with the ISA_C target produce code considerably more dense than the S08, especially for the `int` base and to a lesser extent, the `short` and `char` code bases.

The code density studies are continuing with additional benchmarks and application code. Results will be reported in subsequent document revisions.

6.3.2 Performance

In this section, initial performance results for the V1 ColdFire core are presented.

See Appendix A, "ColdFire Performance Analysis Overview," for an overview of the ColdFire performance methodology.

For the initial analysis, the Dhrystone 2.1 benchmark was used. Four software targets were studied, along with two V1 core memory configurations.

Table 37. Dhrystone 2.1 Configurations

Configuration Name	Description
Software	
isa_a	Compiled with ISA_A target
isa_a_no_div	Compiled with ISA_A target but divide instructions are emulated via function calls
isa_c	Compiled with ISA_C target
isa_c_no_div	Compiled with ISA_C target but divide instructions are emulated via function calls
Hardware	
text = pflash	Executable object file has text in flash, data in RAM
text = pram	Executable object file has text in RAM, data in RAM

The measured V1 ColdFire Dhrystone 2.1 metrics are presented in Table 38. The benchmark is configured to execute its inner loop ten times.

Table 38. V1 ColdFire Core Dhrystone 2.1 Performance Metrics

Configuration	Text Size [bytes]	Dynamic Insts [executed insts]	Eff CPI [cycle per inst]	DMIPS 2.1 per MHz
text = pflash				
isa_a	1252	3316	2.53	0.83
isa_a_no_div	1530	3846	2.47	0.73
isa_c	1202	3105	2.65	0.85
isa_c_no_div	1482	3635	2.57	0.74
text = pram				
isa_a	1252	3316	2.10	1.01
isa_a_no_div	1530	3846	2.02	0.90
isa_c	1202	3105	2.17	1.05
isa_c_no_div	1482	3635	2.07	0.93

The reported HCS08 Dhrystone 2.1 performance is 0.0876 DMIPS 2.1 per MHz. Therefore, the measured V1 performance levels (isa_c_no_div, isa_c) represent a 8.5–12.0x performance improvement for a constant frequency relative to the S08.

For the text = pflash configuration, the effect of disabling the flash speculation was also measured. This configuration is defined by setting CPUCR[FSD] = 1. For this benchmark, disabling the flash speculation decreases Dhrystone 2.1 performance by 12–13%.

Appendix A ColdFire Performance Analysis Overview

A.1 Average Instruction Time Methodology

Since this analysis is based on a quantitative approach, this section presents a brief overview of the processor core performance prediction methodology.

The performance of a processor can be predicted using an Average Instruction Time Methodology. In its simplest form, this CPI (cycles per instruction) metric (or its reciprocal IPC - instructions per cycle) represents the number of machine cycles per instruction and is calculated for a single-issue architecture as:

CPI [cycles/inst] = Summation {F(i) x ET(i)} *Eqn. 1*

IPC [inst/cycle] = 1/CPI *Eqn. 2*

where CPI is the average instruction time expressed in cycles/inst, IPC is the average performance expressed in inst/cycle, F(i) represents the fractional dynamic frequency of occurrence per instruction and ET(i) is the execution time for a given instruction i. By summing the product of relative frequency and execution time for each instruction type, the average instruction time for a processor executing any given instruction mix can be calculated. For this methodology, the cycles are always expressed in processor clocks. The resulting instruction per cycle metric is proportional to performance, so the smaller the CPI metric, the better the performance.

Consider the definition of a Base Average Instruction Time (BaseCPI). Let the BaseCPI represent the maximum performance of the processor strictly as a function of the instruction mix. Stated differently, this metric represents the processor's performance assuming the rest of the system (i.e., caches, memory modules, etc.) is ideal.

A more accurate equation for BaseCPI can be written as:

```
BaseCPI [cycle/inst] = (Summation {F(i) x ET(i)} + Sequence-Related Pipeline Stalls)
```

where the summation product was previously defined and the Sequence-Related Pipeline Stalls includes all pipeline breaks caused by the instruction sequence itself. These stalls may include:

- Wait cycles inserted in the pipeline when required instruction(s) are not available.
- Pipeline breaks, or stalls, generated by hardware interlocks which are asserted to handle some type of resource conflict or limitation.

By summing the product of the relative frequency of occurrence and execution time for each instruction type plus the sequence-related holds, the base average instruction time can be calculated.

The BaseCPI provides a parameter to quantify the performance of a given processor microarchitecture. To convert this value into a more realistic measure of predicted system performance, a series of degradation factors must be considered. Let the Effective Average Instruction Time (Effective CPI, or EffCPI) represent this more realistic measure of performance.

Recall the definition of BaseCPI assumed all system elements, except the processor, were ideal. By quantifying the degradation factors associated with these other system components, the Effective CPI can be calculated:

```
EffCPI [inst/cycle] = (BaseCpi
                  + Summation of Memory Subsystem Factors
                  + Summation of System Factors)
```

For microcontroller configurations, the Memory Subsystem Factors typically include wait states associated with flash and/or SRAM access times, while the System Factors may include system bus arbitration delays, etc.

Given the Average Instruction Time methodology, there are a number of additional tools that have been developed to assist in this type of performance analysis.

Specifically, there are a number of architectural models that can be utilized to analyze various factors within the CPI performance equations. These tools are typically high-level C-language models of certain functions within the design, driven with information output from an instruction set simulator (ISS), which is typically a C-language program defining the expected results of the execution of the ISA which is used in all the design verification work. By inputting an executable memory image file, this ISA simulator then “executes” the program on an instruction by instruction basis, updating all program-visible machine registers and memory as required. Most ISA simulators have been instrumented to optionally output information on instruction fetch, operand addresses, and program counter values.

By executing the target application on the ISA simulator with the appropriate outputs enabled, a stream of data from the executing application can be input to one of the architectural models. This input data then provides the required stimulus to the architectural models.

There are programs which gather detailed statistics about dynamic opcode usage. Recalling the BaseCPI equation, this program provides the F(i) factors associated with the various opcodes across different suites of embedded applications, and is useful in examining the runtime differences across applications for key performance factors, e.g., the relative frequency of branch instructions, etc.

In addition, there are other models used for Effective CPI analysis and the degradation factors associated with memory accesses.

First, there is a cache model that is used to quantify numerous performance parameters for various cache sizes, associativity and organizations. This model uses the stream of reference addresses generated by the ISA simulator from a hardware trace as input, and models the behavior of caches of varying size and organizations. Second, there are often models to quantify the performance of the platform memories, for example, the flash memory with its page buffer implementation.

Finally, a third model provides information for memory address profiling. Using the stream of reference addresses as input, this model merely profiles the memory access patterns to identify critical functions and/or heavily-referenced operand locations. For certain types of microcontroller configurations, this profiling is essential to understanding the required amount of RAM and which variables to map into this space to maximize performance.

Figure A-1 illustrates the methodology for predicting the Effective CPI.

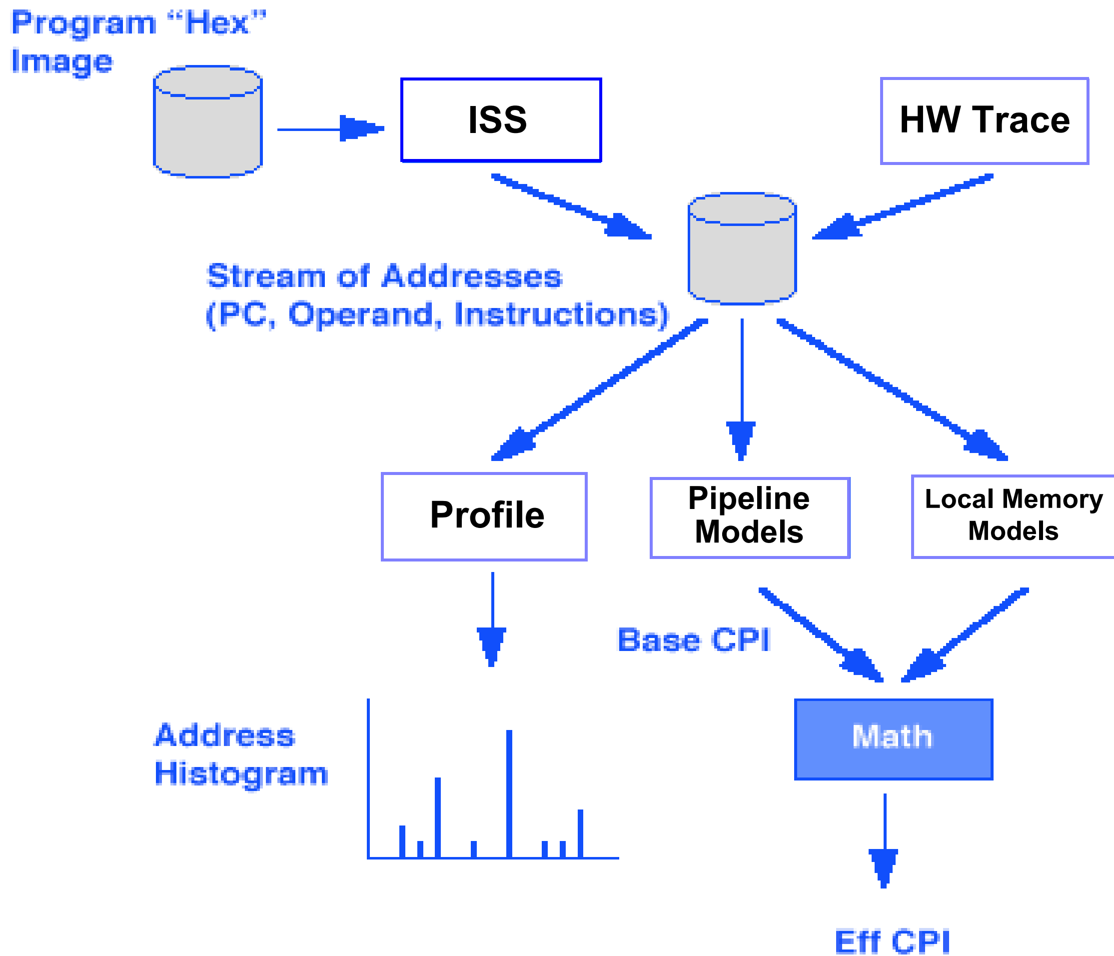


Figure A-1. Effective CPI/IPC Prediction Methodology

In certain applications, there may be an RTL model (or approximation) of the design that can be utilized to directly measure performance of the system under development. For the V1 ColdFire family, this is the performance estimation process used here. Given the Effective IPC definition, the resulting performance of a device, as measured in [1/seconds], can be calculated knowing three factors: IPC [inst/cycle], cycle time [sec/cycle] and the number of instructions executed (the dynamic path length):

$$\mathrm{Perf}[1/\mathrm{sec}] = \frac{\mathrm{Effective\ IPC}[\mathrm{inst/cycle}]}{\mathrm{Cycle\ Time}[\mathrm{sec/cycle}] \times \mathrm{Executed\ Insts}[\mathrm{inst}]} \quad \textbf{\textit{Eqn. 3}}$$

Finally, the relative performance between two systems X and Y can be calculated using the above performance equation. For devices executing the same code image at the same operating frequency, the Executed Insts and the CycleTime factors can be removed (since they are both 1.0) and the relative performance simply defined as:

$$\frac{\mathrm{Performance\ X}}{\mathrm{Performance\ Y}} = \frac{\mathrm{Effective\ IPC\ X\ [inst/cycle]}}{\mathrm{Effective\ IPC\ Y[inst/cycle]}} \quad \textbf{\textit{Eqn. 4}}$$

This equation is used extensively in any relative performance study.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: V1CFWP
Rev. 0
07/2006

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

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

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2006. All rights reserved.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.