Chapter 2
ColdFire Core

This chapter provides an overview of the microprocessor core of the MCF5307. The chapter begins with a description of enhancements from the Version 2 (V2) ColdFire core, and then fully describes the V3 programming model as it is implemented on the MCF5307. It also includes a full description of exception handling, data formats, an instruction set summary, and a table of instruction timings.

2.1 Features and Enhancements

The MCF5307 is the first standard product to contain a Version 3 ColdFire microprocessor core. To reach higher levels of frequency and performance, numerous enhancements were made to the V2 architecture. Most notable are a deeper instruction pipeline, branch acceleration, and a unified cache, which together provide 75 (Dhrystone 2.1) MIPS at 90 MHz.

The MCF5307 core design emphasizes performance, and backward compatibility represents the next step on the ColdFire performance roadmap.

The following list summarizes MCF5307 features:

- Variable-length RISC, clock-multiplied Version 3 microprocessor core
- Two independent, decoupled pipelines—four-stage instruction fetch pipeline (IFP) and two-stage operand execution pipeline (OEP)
- Eight-instruction FIFO buffer provides decoupling between the pipelines
- Branch prediction mechanisms for accelerating program execution
- 32-bit internal address bus supporting 4 Gbytes of linear address space
- 32-bit data bus
- 16 user-accessible, 32-bit-wide, general-purpose registers
- Supervisor/user modes for system protection
- Vector base register to relocate exception-vector table
- Optimized for high-level language constructs

2.1.1 Clock-Multiplied Microprocessor Core

The MCF5307 incorporates a clock-multiplying phase-locked loop (PLL). Increasing the internal speed of the core also allows higher performance while providing the system designer with an easy-to-use lower speed system interface.

The frequency of the processor complex can be 2x, 3x, or 4x the external bus speed.

The processor, cache, integrated SRAM, and misalignment module operate at the higher speed clock (PCLK); other system integrated modules operate at the speed of the bus clock (BCLKO). When combined with the enhanced pipeline structure of the Version 3 ColdFire core, the processor and its local memories provide a high level of performance for today’s demanding embedded applications.

PCLK can be disabled to minimize dissipation when a low-power mode is entered. This is described in Section 7.2.3, “Reduced-Power Mode.”

2.1.2 Enhanced Pipelines

The IFP prefetches instructions. The OEP decodes instructions, fetches required operands, then executes the specified function. The two independent, decoupled pipeline structures maximize performance while minimizing core size. Pipeline stages are shown in Figure 2-1 and are summarized as follows:

- Four-stage IFP (plus optional instruction buffer stage)
  - Instruction address generation (IAG) calculates the next prefetch address.
  - Instruction fetch cycle 1 (IC1) initiates prefetch on the processor’s local instruction bus.
  - Instruction fetch cycle 2 (IC2) completes prefetch on the processor’s instruction local bus.
  - Instruction early decode (IED) generates time-critical decode signals needed for the OEP.
  - Instruction buffer (IB) optional stage uses FIFO queue to minimize effects of fetch latency.
- Two-stage OEP
  - Decode, select/operand fetch (DSOC) decodes the instruction and selects the required components for the effective address calculation, or the operand fetch cycle.
  - Address generation/execute (AGEX) Calculates the oeprand address, or performs the execution of the instruction.

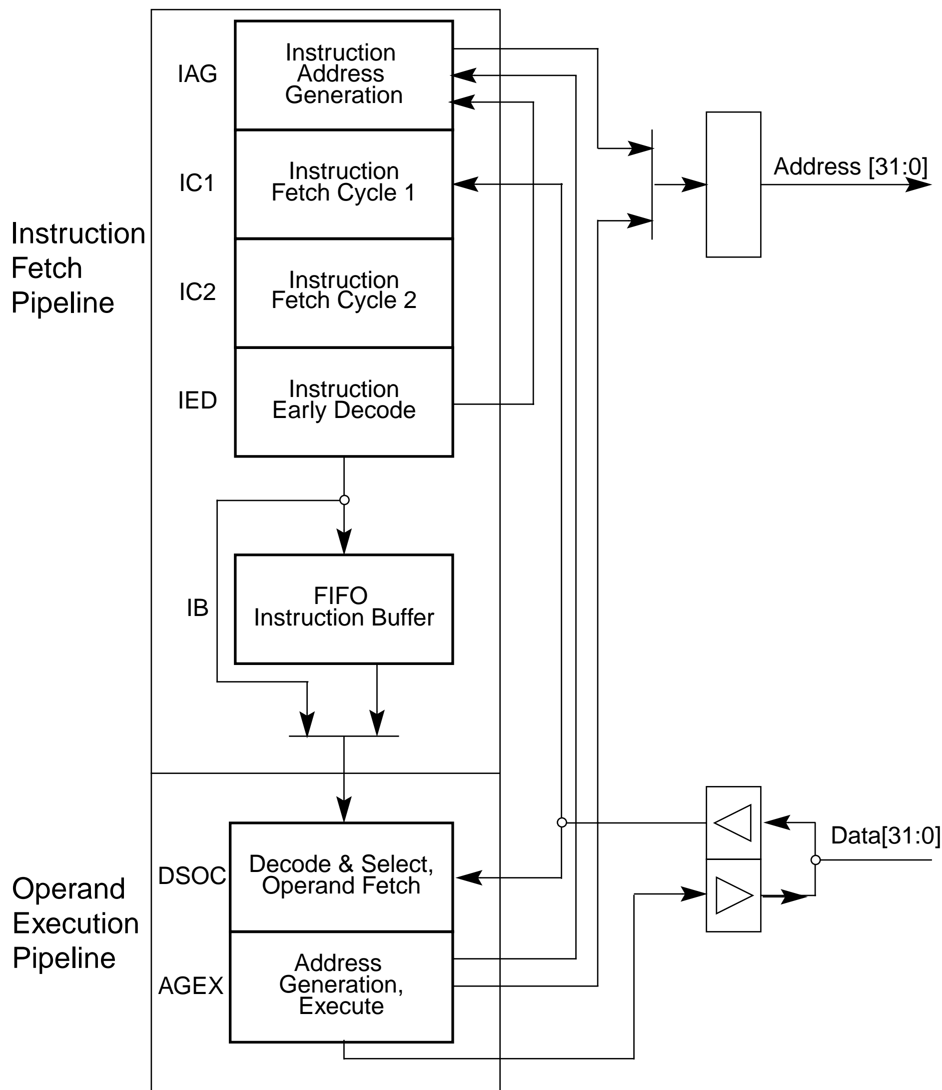

Figure 2-1. ColdFire Enhanced Pipeline

2.1.2.1 Instruction Fetch Pipeline (IFP)

Because the fetch and execution pipelines are decoupled by an eight-instruction FIFO buffer, the IFP can prefetch instructions before the OEP needs them, minimizing stalls.

2.1.2.1.1 Branch Acceleration

Because the IFP and the OEP are decoupled by the instruction buffer, the increased depth of the IFP is generally hidden from the OEP's instruction execution. The one exception is change-of-flow instructions such as unconditional branches or jumps, subroutine calls, and taken conditional branches. To minimize the effects of the increased depth of the IFP, the prefetched instruction stream is monitored for change-of-flow opcodes. When certain types of change-of-flow instructions are detected, the target instruction address is calculated, and fetching immediately begins in the target stream.

For example, if an unconditional BRA instruction is detected, the IED calculates the target of the BRA instruction, and the IAG immediately begins fetching at the target address. Because of the decoupled nature of the two pipelines, the target instruction is available to the OEP immediately after the BRA instruction, giving it a single-cycle execution time.

The acceleration logic uses a static prediction algorithm when processing conditional branch (Bcc) instructions. The default scheme is forward Bcc instructions are predicted as not-taken, while backward Bcc instructions are predicted as taken. A user-mode control bit, CCR[7], allows users to dynamically alter the prediction algorithm for forward Bcc instructions. See Section 2.2.1.5, “Condition Code Register (CCR).

2.1.2.2 Operand Execution Pipeline (OEP)

The OEP is a two-stage pipeline featuring a traditional RISC datapath with a register file feeding an arithmetic/logic unit. For simple register-to-register instructions, the first stage of the OEP performs the instruction decode and fetching of the required register operands (OC), while the actual instruction execution is performed in the second stage (EX).

For memory-to-register instructions, the instruction is effectively staged through the OEP twice in the following way:

- The instruction is decoded and the components of the operand address are selected (DS).
- The operand address is generated using the “execute engine” (AG).
- The memory operand is fetched while any register operand is simultaneously fetched (OC).
- The instruction is executed (EX).

For register-to-memory operations, the stage functions (DS/OC, AG/EX) are effectively performed simultaneously allowing single-cycle execution. For read-modify-write instructions, the pipeline effectively combines a memory-to-register operation with a store operation.

2.1.2.2.1 Illegal Opcode Handling

To aid in conversion from M68000 code, every 16-bit operation word is decoded to ensure that each instruction is valid. If the processor attempts execution of an illegal or unsupported instruction, an illegal instruction exception (vector 4) is taken.

2.1.2.2.2 Hardware Multiply/Accumulate (MAC) Unit

The MAC is an optional unit in Version 3 that provides hardware support for a limited set of digital signal processing (DSP) operations used in embedded code, while supporting the integer multiply instructions in the ColdFire microprocessor family. The MAC features a three-stage execution pipeline, optimized for 16 x 16 multiplies. It is tightly coupled to the OEP, which can issue a 16 x 16 multiply with a 32-bit accumulation plus fetch a 32-bit operand in a single cycle. A 32 x 32 multiply with a 32-bit accumulation requires three cycles before the next instruction can be issued.

Figure 2-2 shows basic functionality of the MAC. A full set of instructions are provided for signed and unsigned integers plus signed, fixed-point fractional input operands.

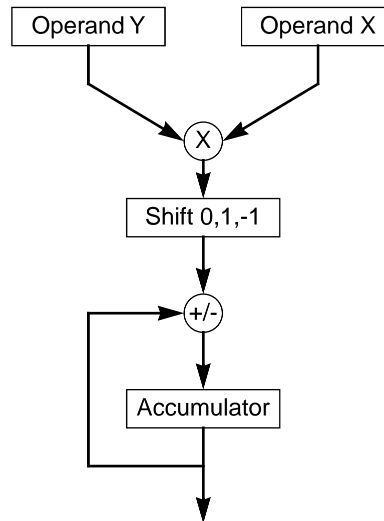

Figure 2-2. ColdFire Multiply-Accumulate Functionality Diagram

The MAC provides functionality in the following three related areas, which are described in detail in Chapter 3, “Hardware Multiply/Accumulate (MAC) Unit.”

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

2.1.2.2.3 Hardware Divide Unit

The hardware divide unit performs the following integer division operations:

- 32-bit operand/16-bit operand producing a 16-bit quotient and a 16-bit remainder
- 32-bit operand/32-bit operand producing a 32-bit quotient
- 32-bit operand/32-bit operand producing a 32-bit remainder

2.1.3 Debug Module Enhancements

The ColdFire processor core debug interface supports system integration in conjunction with low-cost development tools. Real-time trace and debug information can be accessed through a standard interface, which allows the processor and system to be debugged at full speed without costly in-circuit emulators. The MCF5307 debug unit is a compatible upgrade to the MCF52xx debug module with enhancements that include:

- A new command to obtain the value of the program counter (PC)
- Allowing ORing of terms in creating breakpoints
- Increased flexibility of the breakpoint registers

On-chip breakpoint resources include the following:

- Configuration/status register (CSR)
- Background debug mode (BDM) address attributes register (BAAR)
- Bus attributes and mask register (AATR)
- Breakpoint registers. These can be used to define triggers combining address, data, and PC conditions in single- or dual-level definitions. They include the following:
  - PC breakpoint register (PBR)
  - PC breakpoint mask register (PBMR)
  - Data operand address breakpoint registers (ABHR/ABLR)
  - Data breakpoint register (DBR)
- Data breakpoint mask register (DBMR)
- Trigger definition register (TDR) can be programmed to generate a processor halt or initiate a debug interrupt exception.

These registers can be accessed through the dedicated debug serial communication channel, or from the processor's supervisor programming model, using the WDEBUG instruction.

The enhancements of the Revision B debug specification are fully backward-compatible with the A revision. For more information, see Chapter 5, "Debug Support."

2.2 Programming Model

The MCF5307 programming model consists of three instruction and register groups—user, MAC (also user-mode), and supervisor, shown in Figure 2-2. User mode programs are restricted to user and MAC instructions and programming models. Supervisor-mode system software can reference all user-mode and MAC instructions and registers and additional supervisor instructions and control registers. The user or supervisor programming model is selected based on SR[S]. The following sections describe the registers in the user, MAC, and supervisor programming models.

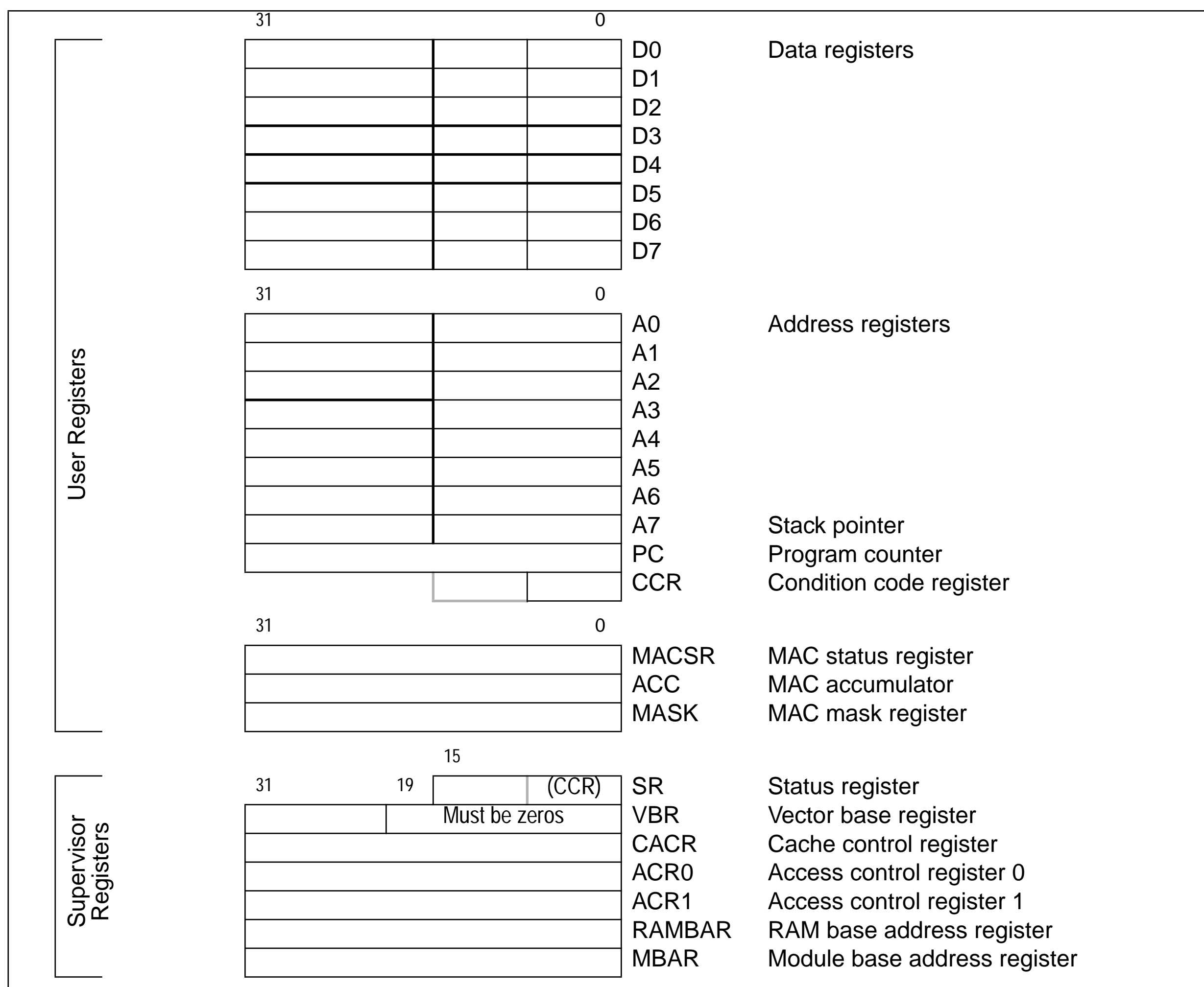

Figure 2-3. ColdFire Programming Model

2.2.1 User Programming Model

As Figure 2-3 shows, the user programming model consists of the following registers:

- 16 general-purpose 32-bit registers, D0–D7 and A0–A7
- 32-bit program counter
- 8-bit condition code register

2.2.1.1 Data Registers (D0–D7)

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

2.2.1.2 Address Registers (A0–A6)

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

2.2.1.3 Stack Pointer (A7, SP)

The processor core supports a single hardware stack pointer (A7) used during stacking for subroutine calls, returns, and exception handling. The stack pointer is implicitly referenced by certain operations and can be explicitly referenced by any instruction specifying an address register. The initial value of A7 is loaded from the reset exception vector, address 0x0000. The same register is used for user and supervisor modes, and may be used for word and longword operations.

A subroutine call saves the program counter (PC) on the stack and the return restores the PC from the stack. The PC and the status register (SR) are saved on the stack during exception and interrupt processing. The return from exception instruction restores SR and PC values from the stack.

2.2.1.4 Program Counter (PC)

The PC holds the address of the executing instruction. For sequential instructions, the processor automatically increments PC. When program flow changes, the PC is updated with the target instruction. For some instructions, the PC specifies the base address for PC-relative operand addressing modes.

2.2.1.5 Condition Code Register (CCR)

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

Table 2-1 describes the CCR fieldsMAC Programming ModelFigure 2-3 shows the registers in the MAC portion of the user programming model. These registers are described as follows:Accumulator (ACC)—This 32-bit, read/write, general-purpose register is used to accumulate the results of MAC operations.

	7	6	5	4	3	2	1	0
Field	P	—		X	N	Z	V	C
Reset	0	00		Undefined				
R/W	R/W	R		R/W	R/W	R/W	R/W	R/W

Table 2-1. CCR Field Descriptions

Bits	Name	Description
7	P	Branch prediction bit. Alters the static prediction algorithm used by the branch acceleration logic in the IFP on forward conditional branches. 0 Predicted as not-taken. 1 Predicted as taken.
6–5	—	Reserved, should be cleared.
4	X	Extend condition code bit. Assigned the value of the carry bit for arithmetic operations; otherwise not affected or set to a specified result. Also used as an input operand for multiple-precision arithmetic.
3	N	Negative condition code bit. Set if the msb of the result is set; otherwise cleared.
2	Z	Zero condition code bit. Set if the result equals zero; otherwise cleared.

Table 2-1. CCR Field Descriptions (Continued)

Bits	Name	Description
1	V	Overflow condition code bit. Set if an arithmetic overflow occurs, implying that the result cannot be represented in the operand size; otherwise cleared.
0	C	Carry condition code bit. Set if a carry-out of the data operand msb occurs for an addition or if a borrow occurs in a subtraction; otherwise cleared.

- Mask register (MASK)—This 16-bit general-purpose register provides an optional address mask for MAC instructions that fetch operands from memory. It is useful in the implementation of circular queues in operand memory.
- MAC status register (MACSR)—This 8-bit register defines configuration of the MAC unit and contains indicator flags affected by MAC instructions. Unless noted otherwise, MACSR indicator flag settings are based on the final result, that is, the result of the final operation involving the product and accumulator.

2.2.2 Supervisor Programming Model

The MCF5307 supervisor programming model is shown in Figure 2-3. Typically, system programmers use the supervisor programming model to implement operating system functions and provide memory and I/O control. The supervisor programming model provides access to the user registers and additional supervisor registers, which include the upper byte of the status register (SR), the vector base register (VBR), and registers for configuring attributes of the address space connected to the Version 3 processor core. Most supervisor-mode registers are accessed by using the MOVEC instruction with the control register definitions in Table 2-2.

Table 2-2. MOVEC Register Map

Rc[11–0]	Register Definition
0x002	Cache control register (CACR)
0x004	Access control register 0 (ACR0)
0x005	Access control register 1 (ACR1)
0x801	Vector base register (VBR)
0xC04	RAM base address register (RAMBAR)
0xC0F	Module base address register (MBAR)

2.2.2.1 Status Register (SR)

The SR stores the processor status, the interrupt priority mask, and other control bits. Supervisor software can read or write the entire SR; user software can read or write only SR[7–0], described in Section 2.2.1.5, “Condition Code Register (CCR).” The control bits indicate processor states—trace mode (T), supervisor or user mode (S), and master or interrupt state (M). SR is set to 0x27*xx* after reset.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	System byte								Condition code register (CCR)							
Field	T	—	S	M	—	I			P	—		X	N	Z	V	C
Reset	0	0	1	0	0	111			0	00		—	—	—	—	—
R/W	R/W	R	R/W	R/W	R	R/W			R/W	R		R/W	R/W	R/W	R/W	R/W

Figure 2-5. Status Register (SR)

Table 2-3 describes SR fields.

Table 2-3. Status Field Descriptions

Bits	Name	Description
15	T	Trace enable. When T is set, the processor performs a trace exception after every instruction.
13	S	Supervisor/user state. Indicates whether the processor is in supervisor or user mode 0 User mode 1 Supervisor mode
12	M	Master/interrupt state. Cleared by an interrupt exception. It can be set by software during execution of the RTE or move to SR instructions so the OS can emulate an interrupt stack pointer.
10–8	I	Interrupt priority mask. Defines the current interrupt priority. Interrupt requests are inhibited for all priority levels less than or equal to the current priority, except the edge-sensitive level-7 request, which cannot be masked.
7–0	CCR	Condition code register. See Table 2-1.

2.2.2.2 Vector Base Register (VBR)

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

	31–20	19–0
Field	Exception vector table base address	—
Reset	0000_0000_0000_0000_0000_0000_0000_0000	
R/W	Written from a BDM serial command or from the CPU using the MOVEC instruction. VBR can be read from the debug module only. The upper 12 bits are returned, the low-order 20 bits are undefined.	
Rc[11–0]	0x801	

Figure 2-6. Vector Base Register (VBR)

2.2.2.3 Cache Control Register (CACR)

The CACR controls operation of both the instruction and data cache memory. It includes bits for enabling, freezing, and invalidating cache contents. It also includes bits for defining the default cache mode and write-protect fields. See Section 4.10.1, “Cache Control Register (CACR).”

2.2.2.4 Access Control Registers (ACR0–ACR1)

The access control registers (ACR0–ACR1) define attributes for two user-defined memory regions. Attributes include definition of cache mode, write protect and buffer write enables. See Section 4.10.2, “Access Control Registers (ACR0–ACR1).”

2.2.2.5 RAM Base Address Register (RAMBAR)

The RAMBAR register determines the base address location of the internal SRAM module and indicates the types of references mapped to it. The RAMBAR includes a base address, write-protect bit, address space mask bits, and an enable. The RAM base address must be aligned on a 0-modulo-32-Kbyte boundary. See Section 4.4.1, “SRAM Base Address Register (RAMBAR).”

2.2.2.6 Module Base Address Register (MBAR)

The module base address register (MBAR) defines the logical base address for the memory-mapped space containing the control registers for the on-chip peripherals. See Section 6.2.2, “Module Base Address Register (MBAR).”

2.3 Integer Data Formats

Table 2-4 lists the integer operand data formats. Integer operands can reside in registers, memory, or instructions. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

Table 2-4. Integer Data Formats

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

2.4 Organization of Data in Registers

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

2.4.1 Organization of Integer Data Formats in Registers

Figure 2-7 shows the integer format for data registers. Each integer data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Longword operands occupy the entire 32 bits of integer data registers. A data register that is either a source or destination operand only uses or changes the appropriate lower 8 or 16 bits in byte or word operations, respectively. The remaining

high-order portion does not change. The least significant bit (lsb) of all integer sizes is zero, the most-significant bit (msb) of a longword integer is 31, the msb of a word integer is 15, and the msb of a byte integer is 7.

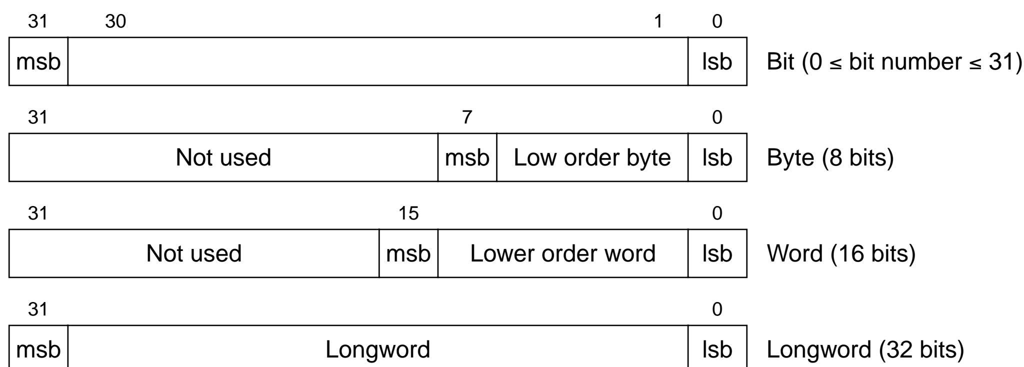

Figure 2-7. Organization of Integer Data Formats in Data Registers

The instruction set encodings do not allow the use of address registers for byte-sized operands. When an address register is a source operand, either the low-order word or the entire longword operand is used, depending on the operation size. Word-length source operands are sign-extended to 32 bits and then used in the operation with anaddress register destination. When an address register is a destination, the entire register is affected, regardless of the operation size. Figure 2-8 shows integer formats for address registers.

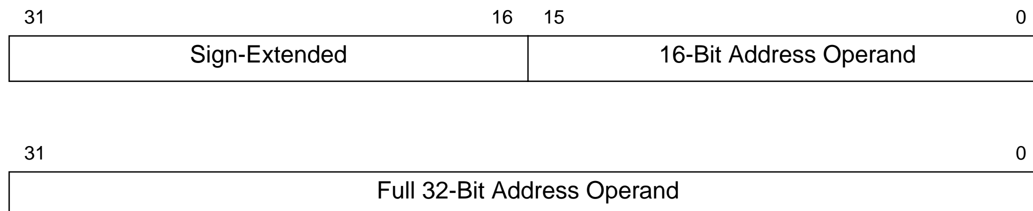

Figure 2-8. Organization of Integer Data Formats in Address Registers

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

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

2.4.2 Organization of Integer Data Formats in Memory

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

organization is shown in Figure 2-9.

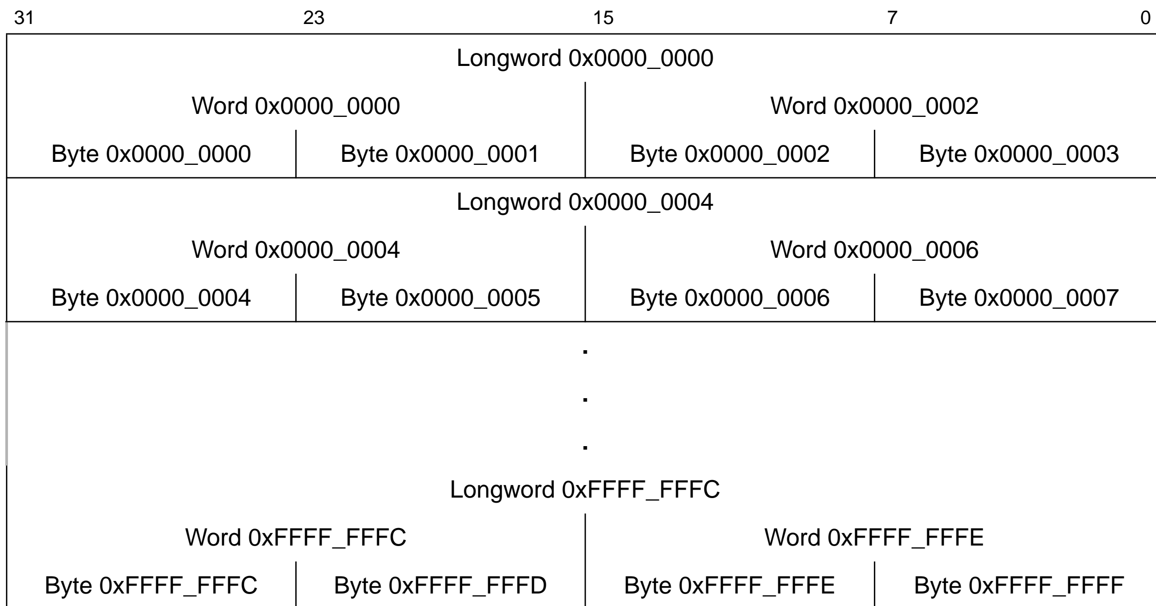

Figure 2-9. Memory Operand Addressing

2.5 Addressing Mode Summary

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

These categories sometimes combine to form more restrictive categories. Two combined classifications are alterable memory (both alterable and memory) and data alterable (both alterable and data). Twelve of the most commonly used effective addressing modes from the M68000 Family are available on ColdFire microprocessors. Table 2-5 summarizes these modes and their categories;

Table 2-5. ColdFire Effective Addressing Modes

Addressing Modes	Syntax	Mode Field	Reg. Field	Category			
				Data	Memory	Control	Alterable
Register direct Data Address	 Dn An	 000 001	 reg. no. reg. no.	 X —	 — —	 — —	 X X
Register indirect Address Address with Postincrement Address with Predecrement Address with Displacement	 (An) (An)+ –(An) (d_{16}, An)	 010 011 100 101	 reg. no. reg. no. reg. no. reg. no.	 X X X X	 X X X X	 X — — X	 X X X X
Address register indirect with index 8-bit displacement	(d_8, An, Xi)	110	reg. no.	X	X	X	X
Program counter indirect with displacement	(d_{16}, PC)	111	010	X	X	X	—
Program counter indirect with index 8-bit displacement	(d_8, PC, Xi)	111	011	X	X	X	—
Absolute data addressing Short Long	 (xxx).W (xxx).L	 111 111	 000 001	 X X	 X X	 X X	 — —
Immediate	#<xxx>	111	100	X	X	—	—

2.6 Instruction Set Summary

The ColdFire instruction set is a simplified version of the M68000 instruction set. The removed instructions include BCD, bit field, logical rotate, decrement and branch, and integer multiply with a 64-bit result. Nine new MAC instructions have been added.

Table 2-6 lists notational conventions used throughout this manual.

Table 2-6. Notational Conventions

Instruction	Operand Syntax
Opcode Wildcard	
cc	Logical condition (example: NE for not equal)

Table 2-6. Notational Conventions (Continued)

Instruction	Operand Syntax
Register Specifications	
An	Any address register n (example: A3 is address register 3)
Ay,Ax	Source and destination address registers, respectively
Dn	Any data register n (example: D5 is data register 5)
Dy,Dx	Source and destination data registers, respectively
Rc	Any control register (example VBR is the vector base register)
Rm	MAC registers (ACC, MAC, MASK)
Rn	Any address or data register
Rw	Destination register w (used for MAC instructions only)
Ry,Rx	Any source and destination registers, respectively
Xi	index register i (can be an address or data register: Ai, Di)
Register Names	
ACC	MAC accumulator register
CCR	Condition code register (lower byte of SR)
MACSR	MAC status register
MASK	MAC mask register
PC	Program counter
SR	Status register
Port Name	
DDATA	Debug data port
PST	Processor status port
Miscellaneous Operands	
#<data>	Immediate data following the 16-bit operation word of the instruction
<ea>	Effective address
<ea>y,<ea>x	Source and destination effective addresses, respectively
<label>	Assembly language program label
<list>	List of registers for MOVEM instruction (example: D3–D0)
<shift>	Shift operation: shift left (<<), shift right (>>)
<size>	Operand data size: byte (B), word (W), longword (L)
uc	Unified cache
# <vector>	Identifies the 4-bit vector number for trap instructions
<>	identifies an indirect data address referencing memory
<xxx>	identifies an absolute address referencing memory
dn	Signal displacement value, n bits wide (example: d16 is a 16-bit displacement)
SF	Scale factor (x1, x2, x4 for indexed addressing mode, <<1n>> for MAC operations)

Table 2-6. Notational Conventions (Continued)

Instruction	Operand Syntax
Operations	
+	Arithmetic addition or postincrement indicator
–	Arithmetic subtraction or predecrement indicator
x	Arithmetic multiplication
/	Arithmetic division
~	Invert; operand is logically complemented
&	Logical AND
|	Logical OR
^	Logical exclusive OR
<<	Shift left (example: D0 << 3 is shift D0 left 3 bits)
>>	Shift right (example: D0 >> 3 is shift D0 right 3 bits)
→	Source operand is moved to destination operand
←→	Two operands are exchanged
sign-extended	All bits of the upper portion are made equal to the high-order bit of the lower portion
If <condition> then <operations> else <operations>	Test the condition. If the condition is true, the operations in the then clause are performed. If the condition is false and the optional else clause is present, the operations in the else claue are performed. If the condition is false and the else clause is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.
Subfields and Qualifiers	
{}	Optional operation
()	Identifies an indirect address
d_n	Displacement value, n-bits wide (example: d_{16} is a 16-bit displacement)
Address	Calculated effective address (pointer)
Bit	Bit selection (example: Bit 3 of D0)
lsb	Least significant bit (example: lsb of D0)
LSB	Least significant byte
LSW	Least significant word
msb	Most significant bit
MSB	Most significant byte
MSW	Most significant word
Condition Code Register Bit Names	

Table 2-6. Notational Conventions (Continued)

Instruction	Operand Syntax
P	Branch prediction
C	Carry
N	Negative
V	Overflow
X	Extend
Z	Zero

2.6.1 Instruction Set Summary

Table 2-7 lists implemented user-mode instructions by opcode.

Table 2-7. User-Mode Instruction Set Summary

Instruction	Operand Syntax	Operand Size	Operation
ADD	Dy,<ea>x <ea>y,Dx	.L .L	Source + destination → destination
ADDA	<ea>y,Ax	.L	Source + destination → destination
ADDI	#<data>,Dx	.L	Immediate data + destination → destination
ADDQ	#<data>,<ea>x	.L	Immediate data + destination → destination
ADDX	Dy,Dx	.L	Source + destination + X → destination
AND	Dy,<ea>x <ea>y,Dx	.L .L	Source & destination → destination
ANDI	#<data>,Dx	.L	Immediate data & destination → destination
ASL	Dy,Dx #<data>,Dx	.L .L	X/C ← (Dx << Dy) ← 0 X/C ← (Dx << #<data>) ← 0
ASR	Dy,Dx #<data>,Dx	.L .L	MSB → (Dx >> Dy) → X/C MSB → (Dx >> #<data>) → X/C
Bcc	<label>	.B,.W	If condition true, then PC + 2 + d_n → PC
BCHG	Dy,<ea>x #<data>,<ea-1>x	.B,.L .B,.L	~(<bit number> of destination) → Z, Bit of destination
BCLR	Dy,<ea>x #<data>,<ea-1>x	.B,.L .B,.L	~(<bit number> of destination) → Z; 0 → bit of destination
BRA	<label>	.B,.W	PC + 2 + d_n → PC
BSET	Dy,<ea>x #<data>,<ea-1>x	.B,.L .B,.L	~(<bit number> of destination) → Z; 1→ bit of destination
BSR	<label>	.B,.W	SP – 4 → SP; next sequential PC→ (SP); PC + 2 + d_n → PC
BTST	Dy,<ea>x #<data>,<ea-1>x	.B,.L .B,.L	~(<bit number> of destination) → Z
CLR	<ea>y,Dx	.B,.W,.L	0 → destination
CMP	<ea>y,Ax	.L	Destination – source
CMPA	<ea>y,Dx	.L	Destination – source

Table 2-7. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
CMPI	<ea>y,Dx	.L	Destination – immediate data
DIVS	<ea-1>y,Dx <ea>y,Dx	.W .L	Dx /<ea>y → Dx {16-bit remainder; 16-bit quotient} Dx /<ea>y → Dx {32-bit quotient} Signed operation
DIVU	<ea-1>y,Dx Dy,<ea>x	.W .L	Dx /<ea>y → Dx {16-bit remainder; 16-bit quotient} Dx /<ea>y → Dx {32-bit quotient} Unsigned operation
EOR	Dy,<ea>x	.L	Source ^ destination → destination
EORI	#<data>,Dx	.L	Immediate data ^ destination → destination
EXT	#<data>,Dx	.B →.W .W →.L	Sign-extended destination → destination
EXTB	Dx	.B →.L	Sign-extended destination → destination
HALT[1]	None	Unsized	Enter halted state
JMP	<ea-3>y	Unsized	Address of <ea> → PC
JSR	<ea-3>y	Unsized	SP – 4 → SP; next sequential PC → (SP); <ea> → PC
LEA	<ea-3>y,Ax	.L	<ea> → Ax
LINK	Ax,#<d16>	.W	SP – 4 → SP; Ax → (SP); SP → Ax; SP + d16 → SP
LSL	Dy,Dx #<data>,Dx	.L .L	X/C ← (Dx << Dy) ← 0 X/C ← (Dx << #<data>) ← 0
LSR	Dy,Dx #<data>,Dx	.L .L	0 → (Dx >> Dy) → X/C 0 → (Dx >> #<data>) → X/C
MAC	Ry,RxSF	.L + (.W × .W) → .L .L + (.L × .L) → .L	ACC + (Ry × Rx){<< 1 | >> 1} → ACC ACC + (Ry × Rx){<< 1 | >> 1} → ACC; (<ea>y{&MASK}) → Rw
MACL	Ry,RxSF,<ea-1>y,Rw	.L + (.W × .W) → .L, .L .L + (.L × .L) → .L, .L	ACC + (Ry × Rx){<< 1 | >> 1} → ACC ACC + (Ry × Rx){<< 1 | >> 1} → ACC; (<ea-1>y{&MASK}) → Rw
MOVE	<ea>y,<ea>x	.B,.W,.L	<ea>y → <ea>x
MOVE from MAC	MASK,Rx ACC,Rx MACSR,Rx	.L	Rm → Rx
	MACSR,CCR	.L	MACSR → CCR
MOVE to MAC	Ry,ACC Ry,MACSR Ry,MASK	.L	Ry → Rm
	#<data>,ACC #<data>,MACSR #<data>,MASK	.L	#<data> → Rm
MOVE from CCR	CCR,Dx	.W	CCR → Dx
MOVE to CCR	Dy,CCR #<data>,CCR	.B	Dy → CCR #<data> → CCR
MOVEA	<ea>y,Ax	.W,.L → .L	Source → destination

Table 2-7. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
MOVEM	#<list>,<ea-2>x <ea-2>y,#<list>	.L .L	Listed registers → destination Source → listed registers
MOVEQ	#<data>,Dx	.B → .L	Sign-extended immediate data → destination
MSAC	Ry,RxSF	.L - (.W × .W) → .L .L - (.L × .L) → .L	ACC – (Ry × Rx){<< 1 | >> 1} → ACC
MSACL	Ry,RxSF,<ea-1>y,Rw	.L - (.W × .W) → .L, .L .L - (.L × .L) → .L, .L	ACC – (Ry × Rx){<< 1 | >> 1} → ACC; (<ea-1>y{&MASK}) → Rw
MULS	<ea>y,Dx	.W X .W → .L .L X .L → .L	Source × destination → destination Signed operation
MULU	<ea>y,Dx	.W X .W → .L .L X .L → .L	Source × destination → destination Unsigned operation
NEG	Dx	.L	0 – destination → destination
NEGX	Dx	.L	0 – destination – X → destination
NOP	none	Unsized	Synchronize pipelines; PC + 2 → PC
NOT	Dx	.L	~ Destination → destination
OR	<ea>y,Dx Dy,<ea>x	.L	Source | destination → destination
ORI	#<data>,Dx	.L	Immediate data | destination → destination
PEA	<ea-3>y	.L	SP – 4 → SP; Address of <ea> → (SP)
PULSE	none	Unsized	Set PST= 0x4
REMS	<ea-1>,Dx	.L	Dx/<ea>y → Dw {32-bit remainder} Signed operation
REMU	<ea-1>,Dx	.L	Dx/<ea>y → Dw {32-bit remainder} Unsigned operation
RTS	none	Unsized	(SP) → PC; SP + 4 → SP
Scc	Dx	.B	If condition true, then 1s — destination; Else 0s → destination
SUB	<ea>y,Dx Dy,<ea>x	.L .L	Destination – source → destination
SUBA	<ea>y,Ax	.L	Destination – source → destination
SUBI	#<data>,Dx	.L	Destination – immediate data → destination
SUBQ	#<data>,<ea>x	.L	Destination – immediate data → destination
SUBX	Dy,Dx	.L	Destination – source – X → destination
SWAP	Dx	.W	MSW of Dx ←→ LSW of Dx
TRAP	#<vector>	Unsized	SP – 4 → SP;PC → (SP); SP – 2 → SP;SR → (SP); SP – 2 → SP; format → (SP); Vector address → PC
TRAPF	None #<data>	Unsized .W .L	PC + 2 → PC PC + 4 → PC PC + 6 → PC

Table 2-7. User-Mode Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation
TST	<ea>y	.B,.W,.L	Set condition codes
UNLK	Ax	Unsized	Ax →SP; (SP) → Ax; SP + 4 → SP
WDDATA	<ea>y	.B,.W,.L	<ea>y →DDATA port

[1] By default the HALT instruction is a supervisor-mode instruction; however, it can be configured to allow user-mode execution by setting CSR[UHE].

Table 2-8 describes supervisor-mode instructions.

Table 2-8. Supervisor-Mode Instruction Set Summary

Instruction	Operand Syntax	Operand Size	Operation
CPUSHL	(An)	Unsized	Invalidate instruction cache line Push and invalidate data cache line Push data cache line and invalidate (I,D)-cache lines
HALT[1]	none	Unsized	Enter halted state
MOVE from SR	SR, Dx	.W	SR → Dx
MOVE to SR	Dy,SR #<data>,SR	.W	Source → SR
MOVEC	Ry,Rc	.L	Ry → Rc **Rc Register Definition** 0x002 Cache control register (CACR) 0x004 Access control register 0 (ACR0) 0x005 Access control register 1 (ACR1) 0x006 Access control register 2 (ACR2) 0x007 Access control register 3 (ACR3) 0x801 Vector base register (VBR) 0xC04 RAM base address register 0 (RAMBAR0) 0xC05 RAM base address register 1 (RAMBAR1)
RTE	None	Unsized	(SP+2) → SR; SP+4 → SP; (SP) → PC; SP + formatfield — SP
STOP	#<data>	.W	Immediate data → SR; enter stopped state
WDEBUG	<ea-2>y	.L	<ea-2>y → debug module

[1] The HALT instruction can be configured to allow user-mode execution by setting CSR[UHE].

2.7 Instruction Timing

The timing data presented in this section assumes the following:

- The OEP is loaded with the opword and all required extension words at the beginning of each instruction execution. This implies that the OEP spends no time waiting for the IFP to supply opwords and/or extension words.
- The OEP experiences no sequence-related pipeline stalls. For the MCF5307, the most common example of this type of stall involves consecutive store operations, excluding the MOVEM instruction. For all store operations (except MOVEM),

certain hardware resources within the processor are marked as "busy" for two clock cycles after the final DSOC cycle of the store instruction. If a subsequent store instruction is encountered within this two-cycle window, it is stalled until the resource again becomes available. Thus, the maximum pipeline stall involving consecutive store operations is two cycles.

- The OEP can complete all memory accesses without memory causing any stall conditions. Thus, timing details in this section assume an infinite zero-wait state memory attached to the core.
- All operand data accesses are assumed to be aligned on the same byte boundary as the operand size:
  - 16-bit operands aligned on 0-modulo-2 addresses
  - 32-bit operands aligned on 0-modulo-4 addresses

  Operands that do not meet these guidelines are misaligned. Table 2-9 shows how the core decomposes a misaligned operand reference into a series of aligned accesses.

Table 2-9. Misaligned Operand References

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

[1] Each timing entry is presented as C(r/w), described as follows:
C is the number of processor clock cycles, including all applicable operand fetches and writes, as well as all internal core cycles required to complete the instruction execution.
r/w is the number of operand reads (r) and writes (w) required by the instruction. An operation performing a read-modify write function is denoted as (1/1).

2.7.1 MOVE Instruction Execution Times

The execution times for the MOVE.{B,W,L} instructions are shown in the next tables. Table 2-12 shows the timing for the other generic move operations.

NOTE:

For all tables in this chapter, the execution time of any instruction using the PC-relative effective addressing modes is equivalent to the time using 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 2-10 lists execution times for MOVE.{B,W} instructions.

Table 2-10. Move Byte and Word Execution Times

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	(xxx).wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
(Ay)+	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
-(Ay)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)
(d16,Ay)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—
(d8,Ay,Xi*SF)	5(1/0)	5(1/1)	5(1/1)	5(1/1)	—	—	—
(xxx).w	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
(xxx).l	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
(d16,PC)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—
(d8,PC,Xi*SF)	5(1/0)	5(1/1)	5(1/1)	5(1/1)	—	—	—
#<xxx>	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

Table 2-11 lists timings for MOVE.L.

Table 2-11. Move Long Execution Times

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	(xxx).wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(Ay)+	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
-(Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(d16,Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,Ay,Xi*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
(xxx).w	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(xxx).l	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(d16,PC)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,PC,Xi*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
#<xxx>	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

Table 2-12 gives execution times for MOVE.L instructions accessing program-visible registers of the MAC unit, along with other MOVE.L timings. Execution times for moving contents of the ACC or MACSR into a destination location represent the best-case scenario when the store instruction is executed and there are no load or MAC or MSAC instruction

in the MAC execution pipeline.

Table 2-12. MAC Move Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
move.l	<ea>,ACC	1(0/0)	—	—	—	—	—	—	1(0/0)
move.l	<ea>,MACSR	2(0/0)	—	—	—	—	—	—	2(0/0)
move.l	<ea>,MASK	1(0/0)	—	—	—	—	—	—	1(0/0)
move.l	ACC,Rx	3(0/0)	—	—	—	—	—	—	—
move.l	MACSR,CCR	3(0/0)	—	—	—	—	—	—	—
move.l	MACSR,Rx	3(0/0)	—	—	—	—	—	—	—
move.l	MASK,Rx	3(0/0)	—	—	—	—	—	—	—

2.7.2 Execution Timings—One-Operand Instructions

Table 2-13 shows standard timings for single-operand instructions.

Table 2-13. One-Operand Instruction Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#xxx
clr.b	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
clr.w	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
clr.l	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
ext.w	Dx	1(0/0)	—	—	—	—	—	—	—
ext.l	Dx	1(0/0)	—	—	—	—	—	—	—
extb.l	Dx	1(0/0)	—	—	—	—	—	—	—
neg.l	Dx	1(0/0)	—	—	—	—	—	—	—
negx.l	Dx	1(0/0)	—	—	—	—	—	—	—
not.l	Dx	1(0/0)	—	—	—	—	—	—	—
scc	Dx	1(0/0)	—	—	—	—	—	—	—
swap	Dx	1(0/0)	—	—	—	—	—	—	—
tst.b	<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
tst.w	<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
tst.l	<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)

2.7.3 Execution Timings—Two-Operand Instructions

Table 2-14 shows standard timings for two-operand instructions.

Table 2-14. Two-Operand Instruction Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
add.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
add.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
addi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
addq.l	#imm,<ea>	1(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
addx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—
and.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
and.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
andi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
asl.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
asr.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
bchg	Dy,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bchg	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
bclr	Dy,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bclr	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
bset	Dy,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	6(1/1)	5(1/1)	—
bset	#imm,<ea>	2(0/0)	5(1/1)	5(1/1)	5(1/1)	5(1/1)	—	—	—
btst	Dy,<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	—
btst	#imm,<ea>	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	—	—	—
cmp.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
cmpi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
divs.w	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(1/0)	20(0/0)
divu.w	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(1/0)	20(0/0)
divs.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
divu.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
eor.l	Dy,<ea>	1(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
eori.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
lea	<ea>,Ax	—	1(0/0)	—	—	1(0/0)	2(0/0)	1(0/0)	—
lsl.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
lsr.l	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
mac.w	Ry,Rx	1(0/0)	—	—	—	—	—	—	—
mac.l	Ry,Rx	3(0/0)	—	—	—	—	—	—	—
msac.w	Ry,Rx	1(0/0)	—	—	—	—	—	—	—
msac.l	Ry,Rx	3(0/0)	—	—	—	—	—	—	—
mac.w	Ry,Rx,ea,Rw	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—

Table 2-14. Two-Operand Instruction Execution Times (Continued)

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
mac.l	Ry,Rx,ea,Rw	—	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
moveq	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
msac.w	Ry,Rx,ea,Rw	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
msac.l	Ry,Rx,ea,Rw	—	5(1/0)	5(1/0)	5(1/0)	5(1/0)	—	—	—
muls.w	<ea>,Dx	3(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	3(0/0)
mulu.w	<ea>,Dx	3(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	3(0/0)
muls.l	<ea>,Dx	5(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	—	—	—
mulu.l	<ea>,Dx	5(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	—	—	—
or.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
or.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
or.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
rems.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
remu.l	<ea>,Dx	35(0/0)	35(1/0)	35(1/0)	35(1/0)	35(1/0)	—	—	—
sub.l	<ea>,Rx	1(0/0)	4(1/0)	4(1/0)	4(1/0)	4(1/0)	5(1/0)	4(1/0)	1(0/0)
sub.l	Dy,<ea>	—	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
subi.l	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
subq.l	#imm,<ea>	1(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
subx.l	Dy,Dx	1(0/0)	—	—	—	—	—	—	—

2.7.4 Miscellaneous Instruction Execution Times

Table 2-15 lists timings for miscellaneous instructions.

Table 2-15. Miscellaneous Instruction Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
cpushl	(Ax)	—	11(0/1)	—	—	—	—	—	—
link.w	Ay,#imm	2(0/1)	—	—	—	—	—	—	—
move.w	CCR,Dx	1(0/0)	—	—	—	—	—	—	—
move.w	<ea>,CCR	1(0/0)	—	—	—	—	—	—	1(0/0)
move.w	SR,Dx	1(0/0)	—	—	—	—	—	—	—
move.w	<ea>,SR	9(0/0)	—	—	—	—	—	—	9(0/0)[1]
movec	Ry,Rc	11(0/1)	—	—	—	—	—	—	—
movem.l [2]	<ea>,&list	—	2+n(n/0)	—	—	2+n(n/0)	—	—	—
movem.l	&list,<ea>	—	2+n(0/n)	—	—	2+n(0/n)	—	—	—

Table 2-15. Miscellaneous Instruction Execution Times (Continued)

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
nop		3(0/0)	—	—	—	—	—	—	—
pea	<ea>	—	2(0/1)	—	—	2(0/1)[3]	3(0/1)[4]	2(0/1)	—
pulse		1(0/0)	—	—	—	—	—	—	—
stop	#imm	—	—	—	—	—	—	—	3(0/0)[5]
trap	#imm	—	—	—	—	—	—	—	18(1/2)
trapf		1(0/0)	—	—	—	—	—	—	—
trapf.w		1(0/0)	—	—	—	—	—	—	—
trapf.l		1(0/0)	—	—	—	—	—	—	—
unlk	Ax	3(1/0)	—	—	—	—	—	—	—
wddata.l	<ea>	—	7(1/0)	7(1/0)	7(1/0)	7(1/0)	8(1/0)	7(1/0)	—
wdebug.l	<ea>	—	10(2/0)	—	—	10(2/0)	—	—	—

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

2.7.5 Branch Instruction Execution Times

Table 2-16 shows general branch instruction timing.

Table 2-16. General Branch Instruction Execution Times

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	(xxx).wl	#<xxx>
bra		—	—	—	—	1(0/1)[1]	—	—	—
bsr		—	—	—	—	1(0/1)[1]	—	—	—
jmp	<ea>	—	5(0/0)	—	—	5(0/0)[1]	6(0/0)	1(0/0)[1]	—
jsr	<ea>	—	5(0/1)	—	—	5(0/1)	6(0/1)	1(0/1)[1]	—
rte		—	—	14(2/0)	—	—	—	—	—
rts		—	—	8(1/0)	—	—	—	—	—

[1] Assumes branch acceleration. Depending on the pipeline status, execution times may vary from 1 to 3 cycles.

For the conditional branch opcodes (bcc), a static algorithm is used to determine the prediction state of the branch. This algorithm is:

```
if bcc is a forward branch && CCR[7] == 0
        then the bcc is predicted as not-taken
```

```
if bcc is a forward branch && CCR[7] == 1
        then the bcc is predicted as taken

        else if bcc is a backward branch
                then the bcc is predicted as taken
```

Table 2-17 shows timing for Bcc instructions.

Table 2-17. Bcc Instruction Execution Times

Opcode	Predicted Correctly as Taken	Predicted Correctly as Not Taken	Predicted Incorrectly
bcc	1(0/0)	1(0/0)	5(0/0)

2.8 Exception Processing Overview

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

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

ColdFire processors use an instruction restart exception model but require more software support to recover from certain access errors. See Table 2-18 for details.

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

1. The processor makes an internal copy of the SR and then enters supervisor mode by setting SR[S] and disabling trace mode by clearing SR[T]. The occurrence of an interrupt exception also forces SR[M] 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 a peripheral device. The IACK cycle is mapped to a special acknowledge address space with the interrupt level encoded in the address.
3. The processor saves the current context by creating an exception stack frame on the system stack. ColdFire processors support a single stack pointer in the A7 address register; therefore, there is no notion of separate supervisor and user stack pointers. As a result, the exception stack frame is created at a 0-modulo-4 address on the top of the current system stack. Additionally, the processor uses a simplified

fixed-length stack frame for all exceptions. The exception type determines whether the program counter in the exception stack frame defines the address of the faulting instruction (fault) or of the next instruction to be executed (next).

4. The processor acquires the address of the first instruction of the exception handler. The exception vector table is aligned on a 1-Mbyte boundary. This instruction address is obtained by fetching a value from the table at the address defined in the vector base register. The index into the exception table is calculated as 4 x vector_number. When the index value is generated, the vector table contents determine the address of the first instruction of the desired handler. After the fetch of the first opcode of the handler is initiated, exception processing terminates and normal instruction processing continues in the handler.

ColdFire processors support a 1024-byte vector table aligned on any 1-Mbyte address boundary; see Table 2-18. The table contains 256 exception vectors where the first 64 are defined by Motorola; the remaining 192 are user-defined interrupt vectors.

Table 2-18. Exception Vector Assignments

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter [1]	Assignment
0	000	—	Initial stack pointer
1	004	—	Initial program counter
2	008	Fault	Access error
3	00C	Fault	Address error
4	010	Fault	Illegal instruction
5	014	Fault	Divide by zero
6–7	018–01C	—	Reserved
8	020	Fault	Privilege violation
9	024	Next	Trace
10	028	Fault	Unimplemented line-a opcode
11	02C	Fault	Unimplemented line-f opcode
12	030	Next	Debug interrupt
13	034	—	Reserved
14	038	Fault	Format error
15	03C	Next	Uninitialized interrupt
16–23	040–05C	—	Reserved
24	060	Next	Spurious interrupt
25–31	064–07C	Next	Level 1–7 autovectored interrupts
32–47	080–0BC	Next	Trap #0–15 instructions
48–60	0C0–0F0	—	Reserved
61	0F4	Fault	Unsupported instruction

Table 2-18. Exception Vector Assignments (Continued)

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter [1]	Assignment
62–63	0F8–0FC	—	Reserved
64–255	100–3FC	Next	User-defined interrupts

[1] The term ‘fault’ refers to the PC of the instruction that caused the exception. The term ‘next’ refers to the PC of the instruction that immediately follows the instruction that caused the fault.

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

2.8.1 Exception Stack Frame Definition

The exception stack frame is shown in Figure 2-10. The first longword of the exception stack frame contains the 16-bit format/vector word (F/V) and the 16-bit status register. The second longword contains the 32-bit program counter address.

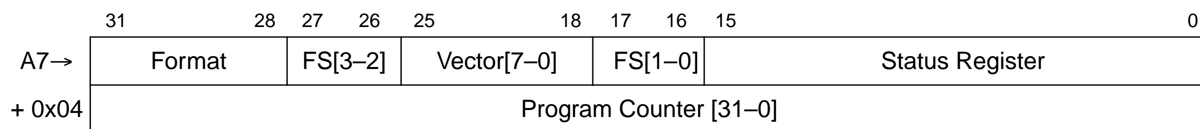

Figure 2-10. Exception Stack Frame Form

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

- Format field—This 4-bit field at the top of the system stack is always written with a value of {4,5,6,7} by the processor indicating a 2-longword frame format. See Table 2-19. This field records any longword misalignment of the stack pointer that may have existed when the exception occurred.

Table 2-19. Format Field Encoding

Original A7 at Time of Exception, Bits 1–0	A7 at First Instruction of Handler	Format Field Bits 31–28
00	Original A[7–8]	0100
01	Original A[7–9]	0101
10	Original A[7–10]	0110
11	Original A[7–11]	0111

- Fault status field—The 4-bit field, FS[3–0], at the top of the system stack is defined for access and address errors along with interrupted debug service routines. See Table 2-20.

Table 2-20. Fault Status Encodings

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

- Vector number—This 8-bit field, vector[7–0], defines the exception type. It is calculated by the processor for internal faults and is supplied by the peripheral for interrupts. See Table 2-18.

2.8.2 Processor Exceptions

Table 2-21 describes MCF5307 exceptions.

Table 2-21. MCF5307 Exceptions

Exception	Description
Access Error	Access errors are reported only in conjunction with an attempted store to write-protected memory. Thus, access errors associated with instruction fetch or operand read accesses are not possible.
Address Error	Caused by an attempted execution transferring control to an odd instruction address (that is, if bit 0 of the target address is set), an attempted use of a word-sized index register (Xi.w) or a scale factor of 8 on an indexed effective addressing mode, or attempted execution of an instruction with a full-format indexed addressing mode.
Illegal Instruction	On Version 2 ColdFire implementations, only some illegal opcodes were decoded and generated an illegal instruction exception. The Version 3 processor decodes the full 16-bit opcode and generates this exception if execution of an unsupported instruction is attempted. Additionally, attempting to execute an illegal line A or line F opcode generates unique exception types: vectors 10 and 11, respectively. ColdFire processors do not provide illegal instruction detection on extension words of any instruction, including MOVEC. Attempting to execute an instruction with an illegal extension word causes undefined results.
Divide by Zero	Attempted division by zero causes an exception (vector 5, offset = 0x014) except when the PC points to the faulting instruction (DIVU, DIVS, REMU, REMS).
Privilege Violation	Caused by attempted execution of a supervisor mode instruction while in user mode. The ColdFire Programmer’s Reference Manual lists supervisor- and user-mode instructions.

Table 2-21. MCF5307 Exceptions (Continued)

Exception	Description
Trace Exception	ColdFire processors provide instruction-by-instruction tracing. While the processor is in trace mode (SR[T] = 1), instruction completion signals a trace exception. This allows a debugger to monitor program execution. The only exception to this definition is the STOP instruction. If the processor is in trace mode, the instruction before the STOP executes and then generates a trace exception. In the exception stack frame, the PC points to the STOP opcode. When the trace handler is exited, the STOP instruction is executed, loading the SR with the immediate operand from the instruction. The processor then generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the just-loaded value. If the processor is not in trace mode and executes a STOP instruction where the immediate operand sets the trace bit in the SR, hardware loads the SR and generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the just-loaded value. Because ColdFire processors do not support hardware stacking of multiple exceptions, it is the responsibility of the operating system to check for trace mode after processing other exception types. As an example, consider a TRAP instruction executing in trace mode. The processor initiates the TRAP exception and passes control to the corresponding handler. If the system requires that a trace exception be processed, the TRAP exception handler must check for this condition (SR[15] in the exception stack frame asserted) and pass control to the trace handler before returning from the original exception.
Debug Interrupt	Caused by a hardware breakpoint register trigger. Rather than generating an IACK cycle, the processor internally calculates the vector number (12). Additionally, the M bit and the interrupt priority mask fields of the SR are unaffected by the interrupt. See Section 2.2.2.1, “Status Register (SR).”
RTE and Format Error Exceptions	When an RTE instruction executes, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire processor, any attempted execution of an RTE where the format is not equal to {4,5,6,7} generates a format error. The exception stack frame for the format error is created without disturbing the original exception frame and the stacked PC points to RTE.The selection of the format value provides limited debug support for porting code from M68000 applications. On M68000 Family processors, the SR was at the top of the stack. Bit 30 of the longword addressed by the system stack pointer is typically zero; so, attempting an RTE using this old format generates a format error on a ColdFire processor. If the format field defines a valid type, the processor does the following: 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 first longword fetch. 4 Transfers control to the instruction address defined by the second longword operand in the stack frame.
TRAP	Executing TRAP always forces an exception and is useful for implementing system calls. The trap instruction may be used to change from user to supervisor mode.
Interrupt Exception	Interrupt exception processing, with interrupt recognition and vector fetching, includes uninitialized and spurious interrupts as well as those where the requesting device supplies the 8-bit interrupt vector. Autovectoring may optionally be configured through the system interface module (SIM). See Section 9.2.2, “Autovector Register (AVR).”

Table 2-21. MCF5307 Exceptions (Continued)

Exception	Description
Reset Exception	Asserting the reset input signal ($\overline{RSTI}$) causes a reset exception. Reset has the highest exception priority; it provides for system initialization and recovery from catastrophic failure. When assertion of $\overline{RSTI}$ is recognized, current processing is aborted and cannot be recovered. The reset exception places the processor in supervisor mode by setting SR[S] and disables tracing by clearing SR[T]. This exception also clears SR[M] and sets the processor's interrupt priority mask in the SR to the highest level (level 7). Next, the VBR is initialized to 0x0000_0000. Configuration registers controlling the operation of all processor-local memories (cache and RAM modules on the MCF5307) are invalidated, disabling the memories. Note: Other implementation-specific supervisor registers are also affected. Refer to each of the modules in this manual for details on these registers. After $\overline{RSTI}$ is negated, the processor waits 80 cycles before beginning the actual reset exception process. During this time, certain events are sampled, including the assertion of the debug breakpoint signal. If the processor is not halted, it initiates the reset exception by performing two longword read bus cycles. The longword at address 0 is loaded into the stack pointer and the longword at address 4 is loaded into the PC. After the initial instruction is fetched from memory, program execution begins at the address in the PC. If an access error or address error occurs before the first instruction executes, the processor enters the fault-on-fault halted state.
Unsupported Instruction Exception	If the MCF5307 attempts to execute a valid instruction but the required optional hardware module is not present in the OEP, a non-supported instruction exception is generated (vector 0x61). Control is then passed to an exception handler that can then process the opcode as required by the system.

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.