PREFACE

This manual defines the functionality of the 32-bit M•CORE microRISC Engine.

Audience

This manual is intended for system software developers and applications programmers who want to develop products for M•CORE-based microcontroller systems. It is assumed that the reader understands operating systems, microprocessor and microcontroller system design, and the basic principles of RISC processing.

Additional Reading

In addition to this manual, the user should consult the user’s manual for the particular M•CORE-based microcontroller of interest. The user’s manual contains information on system interfacing, development support, the hardware accelerator unit, and on-chip peripherals for the microcontroller.

For a list of available documentation, refer to the following web site:

http://www.motorola.com/mcore

Conventions

This document uses the following notational conventions:

- **mnemonics** Instruction mnemonics are shown in lowercase bold.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0F</td>
<td>Hexadecimal numbers</td>
</tr>
<tr>
<td>0b0011</td>
<td>Binary numbers</td>
</tr>
<tr>
<td>x</td>
<td>In certain contexts, this indicates a don’t care. For example, if a field is binary encoded 0bx01, the state of the first bit is a don’t care.</td>
</tr>
</tbody>
</table>

Nomenclature

- **Logic level one** is the voltage that corresponds to Boolean true (1) state.

- **Logic level zero** is the voltage that corresponds to Boolean false (0) state.

  To **set** a bit or bits means to establish logic level one on the bit or bits.

  To **clear** a bit or bits means to establish logic level zero on the bit or bits.

- **LSB** means least significant bit or bits. **MSB** means most significant bit or bits. References to low and high bytes are spelled out.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Paragraph</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PREFACE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Audience</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Additional Reading</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Conventions</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Nomenclature</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>LIST OF TABLES</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Features</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2</td>
<td>Microarchitecture Summary</td>
<td>1-2</td>
</tr>
<tr>
<td>1.3</td>
<td>Programming Model</td>
<td>1-3</td>
</tr>
<tr>
<td>1.4</td>
<td>Data Format Summary</td>
<td>1-4</td>
</tr>
<tr>
<td>1.5</td>
<td>Operand Addressing Capabilities</td>
<td>1-5</td>
</tr>
<tr>
<td>1.6</td>
<td>Instruction Set Overview</td>
<td>1-6</td>
</tr>
<tr>
<td>2.1</td>
<td>User Programming Model</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.1</td>
<td>General-Purpose Registers</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Program Counter</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Condition Code/Carry Bit</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2</td>
<td>Supervisor Programming Model</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Alternate Register File</td>
<td>2-3</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Processor Status Register</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2.2.1</td>
<td>Updates to the PSR</td>
<td>2-6</td>
</tr>
<tr>
<td>2.2.2.2</td>
<td>Exception Recognition and Processing Updates</td>
<td>2-6</td>
</tr>
<tr>
<td>2.2.2.3</td>
<td>RTE and RFI Instruction Updates</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.2.4</td>
<td>MTCR Instruction Updates</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Vector Base Register</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Supervisor Storage Registers</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Exception Shadow Registers</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Global Control Register</td>
<td>2-8</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Global Status Register</td>
<td>2-8</td>
</tr>
<tr>
<td>3.1</td>
<td>Instruction Types and Addressing Modes</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Register-to-Register Instructions</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1.1</td>
<td>Monadic Register Addressing Mode</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1.2</td>
<td>Dyadic Register Addressing Mode</td>
<td>3-1</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Paragraph</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1.3</td>
<td>Register with 5-Bit Immediate Mode</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.1.4</td>
<td>Register with 5-Bit Offset Immediate Mode</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.1.5</td>
<td>Register with 7-Bit Immediate Mode</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.1.6</td>
<td>Control Register Addressing Mode</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Data Memory Access Instructions</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Scaled 4-Bit Immediate Addressing Mode</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.2.2</td>
<td>Load/Store Register Quadrant Mode</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.2.3</td>
<td>Load/Store Multiple Register Mode</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.2.4</td>
<td>Load Relative Word Mode</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Flow Control Instructions</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.3.1</td>
<td>Scaled 11-Bit Displacement Mode</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.3.2</td>
<td>Register Addressing Mode</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.3.3</td>
<td>Indirect Mode</td>
<td>3-5</td>
</tr>
<tr>
<td>3.1.3.4</td>
<td>Register with 4-Bit Negative Displacement Mode</td>
<td>3-5</td>
</tr>
<tr>
<td>3.2</td>
<td>Opcode Map</td>
<td>3-6</td>
</tr>
<tr>
<td>3.3</td>
<td>Instruction Set</td>
<td>3-9</td>
</tr>
</tbody>
</table>

### SECTION 4 EXCEPTION PROCESSING

<table>
<thead>
<tr>
<th>Paragraph</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Exception Processing Overview</td>
<td>4-1</td>
</tr>
<tr>
<td>4.2</td>
<td>Stages of Exception Processing</td>
<td>4-2</td>
</tr>
<tr>
<td>4.3</td>
<td>Exception Vectors</td>
<td>4-3</td>
</tr>
<tr>
<td>4.4</td>
<td>Exception Types</td>
<td>4-4</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Reset Exception (Vector Offset 0x0)</td>
<td>4-4</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Misaligned Access Exception (Vector Offset 0x4)</td>
<td>4-4</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Access Error Exception (Vector Offset 0x8)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Divide-by-Zero Exception (Vector Offset 0x0C)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Illegal Instruction Exception (Vector Offset 0x10)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Privilege Violation Exception (Vector Offset 0x14)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Trace Exception (Vector Offset 0x18)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.4.8</td>
<td>Breakpoint Exception (Vector Offset 0x1C)</td>
<td>4-6</td>
</tr>
<tr>
<td>4.4.9</td>
<td>Unrecoverable Error Exception (Vector Offset 0x20)</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4.10</td>
<td>Soft Reset Exception (Vector Offset 0x24)</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4.11</td>
<td>Interrupt Exceptions</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4.11.1</td>
<td>Normal Interrupt (INT)</td>
<td>4-8</td>
</tr>
<tr>
<td>4.4.11.2</td>
<td>Fast Interrupt (FINT)</td>
<td>4-8</td>
</tr>
<tr>
<td>4.4.12</td>
<td>Hardware Accelerator Exception (Vector Offset 0x30)</td>
<td>4-8</td>
</tr>
<tr>
<td>4.4.13</td>
<td>Instruction Trap Exception (Vector Offset 0x40-0x5C)</td>
<td>4-9</td>
</tr>
<tr>
<td>4.5</td>
<td>Exception Priorities</td>
<td>4-9</td>
</tr>
<tr>
<td>4.6</td>
<td>Returning from Exception Handlers</td>
<td>4-11</td>
</tr>
</tbody>
</table>

### INDEX

MOTOROLA M•CORE REFERENCE MANUAL
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Programming Model</td>
<td>1-4</td>
</tr>
<tr>
<td>1-2</td>
<td>Data Organization in Memory</td>
<td>1-5</td>
</tr>
<tr>
<td>1-3</td>
<td>Data Organization in Registers</td>
<td>1-5</td>
</tr>
<tr>
<td>2-1</td>
<td>User Programming Model</td>
<td>2-1</td>
</tr>
<tr>
<td>2-2</td>
<td>Supervisor Additional Resources</td>
<td>2-3</td>
</tr>
<tr>
<td>2-3</td>
<td>Processor Status Register</td>
<td>2-4</td>
</tr>
<tr>
<td>3-1</td>
<td>Monadic Format</td>
<td>3-1</td>
</tr>
<tr>
<td>3-2</td>
<td>Dyadic Format</td>
<td>3-2</td>
</tr>
<tr>
<td>3-3</td>
<td>Register with 5-Bit Immediate Format</td>
<td>3-2</td>
</tr>
<tr>
<td>3-4</td>
<td>Register with 5-Bit Offset Immediate Format</td>
<td>3-2</td>
</tr>
<tr>
<td>3-5</td>
<td>Register with 7-Bit Immediate Format</td>
<td>3-2</td>
</tr>
<tr>
<td>3-6</td>
<td>Control Register Addressing Format</td>
<td>3-3</td>
</tr>
<tr>
<td>3-7</td>
<td>Scaled 4-Bit Immediate Format</td>
<td>3-3</td>
</tr>
<tr>
<td>3-8</td>
<td>Load/Store Register Quadrant Format</td>
<td>3-3</td>
</tr>
<tr>
<td>3-9</td>
<td>Load/Store Multiple Registers Format</td>
<td>3-4</td>
</tr>
<tr>
<td>3-10</td>
<td>Load Relative Word Format</td>
<td>3-4</td>
</tr>
<tr>
<td>3-11</td>
<td>Scaled 11-Bit Displacement Format</td>
<td>3-4</td>
</tr>
<tr>
<td>3-12</td>
<td>Register Addressing Format</td>
<td>3-4</td>
</tr>
<tr>
<td>3-13</td>
<td>Indirect Format</td>
<td>3-5</td>
</tr>
<tr>
<td>3-14</td>
<td>Register with 4-Bit Displacement Addressing Format</td>
<td>3-5</td>
</tr>
<tr>
<td>4-1</td>
<td>Interrupt Interface Signals</td>
<td>4-8</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>M•CORE Instruction Set</td>
<td>1-6</td>
</tr>
<tr>
<td>3-1</td>
<td>Opcode Map</td>
<td>3-6</td>
</tr>
<tr>
<td>4-1</td>
<td>Exception Vector Assignments</td>
<td>4-3</td>
</tr>
<tr>
<td>4-2</td>
<td>Exception Priority Groups</td>
<td>4-9</td>
</tr>
<tr>
<td>4-3</td>
<td>Exceptions, Tracing and BRKREQ Results</td>
<td>4-10</td>
</tr>
</tbody>
</table>

M•CORE MOTOROLA
REFERENCE MANUAL
SECTION 1 INTRODUCTION

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

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

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

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

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

1.1 Features

The main features of the M•CORE are as follows:

- 32-Bit Load/Store RISC Architecture
- Fixed 16-Bit Instruction Length
- 16 Entry 32-Bit General-Purpose Register File
- Efficient 4-Stage Execution Pipeline, Hidden from Application Software
- Single-Cycle Instruction Execution for many Instructions
• Two Cycles for Branches and Memory Access Instructions
• Support for Byte, Halfword, and Word Memory Accesses
• Fast Interrupt Support with 16-Entry Dedicated Alternate Register File
• Vectored and Autovectored Interrupt Support

1.2 Microarchitecture Summary

The M•CORE instruction execution pipeline consists of the following stages:

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

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

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

The execution unit consists of a 32-bit arithmetic/logic unit (ALU), a 32-bit barrel shifter, find-first-one unit (FFO), result feed-forward hardware, and miscellaneous support hardware for multiplication and multiple register loads and stores. Arithmetic and logical operations are executed in a single cycle with the exception of the multiply, signed divide, and unsigned divide instructions. The multiply instruction is implemented with a 2-bit per clock, overlapped-scan, modified Booth algorithm with early-out capability to reduce execution time for operations with small multiplier values. The signed divide and unsigned divide instructions also have data-dependent timing. A find-first-one unit operates in a single clock cycle.

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

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

A single condition code/carry (C) bit is provided for condition testing and for use in implementing arithmetic and logical operations greater than 32 bits. Typically, the C bit is set only by explicit test/comparison operations, not as a side-effect of normal instruction operation. Exceptions to this rule occur for specialized operations for which it is desirable to combine condition setting with actual computation.
A 16-entry alternate register file is provided to support low overhead interrupt exception processing. The CPU supports both vectored and auto vectored interrupts.

1.3 Programming Model

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

User programs can only access registers specific to the user mode; system software executing in the supervisor mode can access all registers, using the control registers to perform supervisory functions. User programs are thus restricted from accessing privileged information. The operating system performs management and service tasks for the user programs by coordinating their activities.

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

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

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

The registers depicted in the programming model (see Figure 1-1) provide operand storage and control. The registers are partitioned into two levels of privilege: user and supervisor. The user programming model consists of sixteen general-purpose 32-bit registers, the 32-bit program counter (PC) and the condition/carry (C) bit. The C bit is implemented as bit 0 of the PSR. This is the only portion of the PSR accessible by the user. The supervisor programming model consists of sixteen additional 32-bit general-purpose registers (the alternate file), as well as a set of status/control registers and scratch registers. By convention, register R15 serves as the link register for subroutine calls, and register R0 is typically used as the current stack pointer.

The alternate file is selected for use via a control bit in the PSR. The status, control, and scratch registers are accessed via the Move from Control Register (mfcr) and Move to Control Register (mtcr) instructions. When the alternate file is selected via the AF bit in the PSR, general-purpose operands are accessed from it. When the AF bit is cleared, operands are accessed from the normal file. This alternate file is provided to allow very low overhead context switching capability for real-time event handling.
The supervisor programming model includes the PSR, which contains operation control and status information. In addition, a set of exception shadow registers are provided to save the state of the PSR and the program counter at the time an exception occurs. A separate set of shadow registers is provided for fast interrupt support to minimize context saving overhead.

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

### 1.4 Data Format Summary

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

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

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

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

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

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

Table 1-1 provides an alphabetized listing of the M•CORE instruction set. Refer to SECTION 3 INSTRUCTIONS for more details on instruction operation.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute Value</td>
</tr>
<tr>
<td>ADDC</td>
<td>Add with C bit</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>ADDU</td>
<td>Add Unsigned</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>ANDI</td>
<td>Logical AND Immediate</td>
</tr>
<tr>
<td>ANDN</td>
<td>AND NOT</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td>ASRC</td>
<td>Arithmetic Shift Right, Update C Bit</td>
</tr>
<tr>
<td>BCLRI</td>
<td>Clear Bit</td>
</tr>
<tr>
<td>BF</td>
<td>Branch on Condition False</td>
</tr>
<tr>
<td>BGENI</td>
<td>Bit Generate Immediate</td>
</tr>
<tr>
<td>BGENR</td>
<td>Bit Generate Register</td>
</tr>
<tr>
<td>BKPT</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>BMASKI</td>
<td>Bit Mask Immediate</td>
</tr>
<tr>
<td>BR</td>
<td>Branch</td>
</tr>
<tr>
<td>BREV</td>
<td>Bit Reverse</td>
</tr>
<tr>
<td>BSETI</td>
<td>Bit Set Immediate</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch to Subroutine</td>
</tr>
<tr>
<td>BT</td>
<td>Branch on Condition True</td>
</tr>
<tr>
<td>BTSTI</td>
<td>Bit Test Immediate</td>
</tr>
<tr>
<td>CLRF</td>
<td>Clear Register on Condition False</td>
</tr>
<tr>
<td>CLRT</td>
<td>Clear Register on Condition True</td>
</tr>
<tr>
<td>CMPHS</td>
<td>Compare Higher or Same</td>
</tr>
<tr>
<td>CMPLT</td>
<td>Compare Less-Than</td>
</tr>
<tr>
<td>CMPLTI</td>
<td>Compare Less-Than Immediate</td>
</tr>
<tr>
<td>CMPNE</td>
<td>Compare Not Equal</td>
</tr>
<tr>
<td>CMPNEI</td>
<td>Compare Not Equal Immediate</td>
</tr>
<tr>
<td>DECF</td>
<td>Decrement on Condition False</td>
</tr>
<tr>
<td>DECGT</td>
<td>Decrement Register and Set Condition if Result Greater-Than Zero</td>
</tr>
<tr>
<td>DEGLT</td>
<td>Decrement Register and Set Condition if Result Less-Than Zero</td>
</tr>
<tr>
<td>DEGNE</td>
<td>Decrement Register and Set Condition if Result Not Equal to Zero</td>
</tr>
<tr>
<td>DECT</td>
<td>Decrement on Condition True</td>
</tr>
<tr>
<td>DIVS</td>
<td>Divide (Signed)</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide (Unsigned)</td>
</tr>
<tr>
<td>DOZE</td>
<td>Doze</td>
</tr>
<tr>
<td>FF1</td>
<td>Find First One</td>
</tr>
<tr>
<td>INCF</td>
<td>Increment on Condition False</td>
</tr>
<tr>
<td>INCT</td>
<td>Increment on Condition True</td>
</tr>
<tr>
<td>IXH</td>
<td>Index Halfword</td>
</tr>
<tr>
<td>IXW</td>
<td>Index Word</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JMPI</td>
<td>Jump Indirect</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to Subroutine</td>
</tr>
<tr>
<td>JSRI</td>
<td>Jump to Subroutine Indirect</td>
</tr>
<tr>
<td>LD,[BHW]</td>
<td>Load</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple Registers</td>
</tr>
<tr>
<td>LDQ</td>
<td>Load Register Quadrant</td>
</tr>
<tr>
<td>LOPT</td>
<td>Decrement with C-Bit Update and Branch if Condition True</td>
</tr>
<tr>
<td>LRW</td>
<td>Load Relative Word</td>
</tr>
<tr>
<td>LSL, LSR</td>
<td>Logical Shift Left and Right</td>
</tr>
<tr>
<td>LSLC, LSRC</td>
<td>Logical Shift Left and Right, Update C bit</td>
</tr>
<tr>
<td>LSLI, LSRI</td>
<td>Logical Shift Left and Right by Immediate</td>
</tr>
<tr>
<td>MFCR</td>
<td>Move from Control Register</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>MOVI</td>
<td>Move Immediate</td>
</tr>
<tr>
<td>MOVF</td>
<td>Move on Condition False</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move on Condition True</td>
</tr>
<tr>
<td>MTCR</td>
<td>Move to Control Register</td>
</tr>
<tr>
<td>MULT</td>
<td>Multiply</td>
</tr>
<tr>
<td>MVC</td>
<td>Move C Bit to Register</td>
</tr>
<tr>
<td>MVCV</td>
<td>Move Inverted C Bit to Register</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical Complement</td>
</tr>
<tr>
<td>OR</td>
<td>Logical Inclusive-OR</td>
</tr>
<tr>
<td>ROTLI</td>
<td>Rotate Left by Immediate</td>
</tr>
<tr>
<td>RSUB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>RSUBI</td>
<td>Reverse Subtract Immediate</td>
</tr>
<tr>
<td>RTE</td>
<td>Return from Exception</td>
</tr>
<tr>
<td>RFI</td>
<td>Return from Interrupt</td>
</tr>
<tr>
<td>SEXTB</td>
<td>Sign-Extend Byte</td>
</tr>
<tr>
<td>SEXTH</td>
<td>Sign-Extend Halfword</td>
</tr>
<tr>
<td>ST,[BHW]</td>
<td>Store</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple Registers</td>
</tr>
<tr>
<td>STQ</td>
<td>Store Register Quadrant</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop</td>
</tr>
<tr>
<td>SUBC</td>
<td>Subtract with C Bit</td>
</tr>
<tr>
<td>SUBU</td>
<td>Subtract</td>
</tr>
<tr>
<td>SUBI</td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>SYNC</td>
<td>Synchronize</td>
</tr>
<tr>
<td>TRAP</td>
<td>Trap</td>
</tr>
<tr>
<td>TST</td>
<td>Test Operands</td>
</tr>
<tr>
<td>TSTNBZ</td>
<td>Test for No Byte Equal Zero</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>XSR</td>
<td>Extended Shift Right</td>
</tr>
<tr>
<td>XTRB0</td>
<td>Extract Byte 0</td>
</tr>
<tr>
<td>XTRB1</td>
<td>Extract Byte 1</td>
</tr>
<tr>
<td>XTRB2</td>
<td>Extract Byte 2</td>
</tr>
<tr>
<td>XTRB3</td>
<td>Extract Byte 3</td>
</tr>
<tr>
<td>ZEXTB</td>
<td>Zero-Extend Byte</td>
</tr>
<tr>
<td>ZEXTH</td>
<td>Zero-Extend Halfword</td>
</tr>
</tbody>
</table>
SECTION 2 REGISTERS

This section describes the organization of the M·CORE general-purpose registers (GPRs) and control registers in the user and supervisor programming models. Refer to SECTION 4 EXCEPTION PROCESSING for details on the exception model.

2.1 User Programming Model

The user programming model, shown in Figure 2-1, consists of the following registers:

- Sixteen general-purpose 32-bit registers (R0-R15)
- 32-bit program counter (PC)
- Condition code/carry flag (C bit)

![Figure 2-1 User Programming Model](image)

R0  Stack pointer
R1  Volatile
R2  Volatile, 1st arg
R3  Volatile, 2nd arg
R4  Volatile, 3rd arg
R5  Volatile, 4th arg
R6  Volatile, 5th arg
R7  Volatile, 6th arg
R8  Non-volatile
R9  Non-volatile
R10 Non-volatile
R11 Non-volatile
R12 Non-volatile
R13 Non-volatile
R14 Non-volatile
R15 Link register
PC  Program counter
C

Figure 2-1  User Programming Model

2.1.1 General-Purpose Registers

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

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

2.1.3 Condition Code/Carry Bit

The C bit represents a condition generated by a processor operation. The C bit can be set explicitly by comparison operations or implicitly as a result of executing extended precision arithmetic and logical operations. In addition, specialized instructions (such as the Decrement, Loop, and Extract Byte instructions) update the C bit as a result of normal execution.

2.2 Supervisor Programming Model

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

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

- 16-entry, 32-bit alternate register file
- Processor status register (PSR)
- Vector base register (VBR)
- Exception saved PSR (EPSR)
- Fast interrupt saved PSR (FPSR)
- Exception saved program counter (EPC)
- Fast interrupt saved program counter (FPC)
- Five 32-bit supervisor scratch registers (SS0-SS4)
- 32-bit global control register (GCR)
- 32-bit global status register (GSR)
The following paragraphs describe the supervisor programming model registers. Additional information can be found in **SECTION 4 EXCEPTION PROCESSING**.

### 2.2.1 Alternate Register File

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

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

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

The processor status register (PSR) stores the processor status (including the C bit) and control data. The control bits indicate the following states for the processor: trace mode (TM bits), supervisor or user mode (S bit), and normal or alternate file state (AF). They also indicate whether exception shadow registers are available for context saving, and whether interrupts are enabled. The PSR can be accessed in supervisor mode only.

**PSR — Processor Status Register**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>SP</td>
<td>U3</td>
<td>U2</td>
<td>U1</td>
<td>U0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEC</td>
</tr>
</tbody>
</table>

**RESET:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>SP</td>
<td>U3</td>
<td>U2</td>
<td>U1</td>
<td>U0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEC</td>
</tr>
</tbody>
</table>

**Notes:**

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

**Figure 2-3 Processor Status Register**

S — Supervisor Mode

0 = Processor is operating in user mode.
1 = Processor is operating in supervisor mode.

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

SP — Spare

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

U3–U0 — Hardware Accelerator Control

The U3-U0 bits control execution of the hardware accelerator instructions. Each bit corresponds to one encoded value of the UU field of a hardware accelerator opcode. If the appropriate bit is cleared, an attempt to execute a corresponding hardware accelerator instruction is aborted, and a disabled hardware accelerator exception is taken. These bits are cleared by hard reset and are unaffected by other exceptions. Refer to the appropriate microcontroller user’s manual for details of the hardware accelerator instructions. In addition, **4.4.12 Hardware Accelerator Exception (Vector Offset 0x30)** describes the disabled hardware accelerator exception.
VEC — Vector Number
This seven-bit field is written with the vector number used to fetch an exception vector when an exception occurs. This field is cleared by reset.

TM — Trace Mode
When this field is non-zero, the M•CORE is placed in trace mode. A trace exception may be taken after the execution of each instruction or only after potential change of flow instructions. This field is cleared by reset and also by hardware when exception processing is initiated. The field is defined as follows:
- 00 = Normal execution
- 01 = Instruction trace mode
- 10 = Reserved
- 11 = Change of flow trace mode
Refer to 4.4.7 Trace Exception (Vector Offset 0x18) for more details on trace operation.

TP — Trace Pending
This bit is set in the appropriate PSR shadow register as part of exception recognition when the M•CORE is in instruction trace mode, a trace exception is pending, and another exception takes priority over the trace exception at an instruction boundary. Setting this bit in the PSR has no effect, as the bit is hardwired to zero. A trace exception is taken after the execution of rte or rfi if this bit is set in the appropriate shadow PSR.

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

SC — Spare Control
This bit is a spare control bit. This bit is cleared when an exception occurs and by reset.

MM — Misalignment Exception Mask
- 0 = Loading and storing instructions to a misaligned address causes a misalignment exception to occur instead of a memory access.
- 1 = Alignment restrictions are ignored, and the lower address bit(s) are ignored and assumed to be zero.
This bit does not affect exceptions for the jmpi or jsri instructions. This bit is cleared by reset and is unaffected by other exceptions.

EE — Exception Enable
- 0 = Shadowing of the PSR and PC by the EPSR and EPC registers on an exception is assumed to result in an unrecoverable error.
- 1 = The EPSR and EPC shadow registers are available to save the exception state. This bit is cleared by reset. Hardware clears this bit on any exception (including a fast interrupt exception) to indicate that processor context for the exception cannot be overwritten in a recoverable manner.
IC — Interrupt Control

0 = Interrupts are only recognized on instruction boundaries.
1 = A valid pending interrupt (INT or FINT) is allowed to cause a long latency, multi-
cycle instruction (divs, divu, ldm, ldq, mult, stm, or stq) to be interrupted be-
fore completion. The instruction will be restarted on return from the interrupt
handler. For the ldm, ldq, stm, and stq instructions, an access in progress will
complete prior to interruption.

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

IE — Interrupt Enable

0 = The INT interrupt input is disabled.
1 = The INT interrupt is sampled.

This bit is cleared by soft reset. It is also cleared when any exception occurs to disable
interrupts signalled by the INT input.

FE — Fast Interrupt Enable

0 = The FPSR and FPC shadow registers are frozen and the FINT interrupt input
is disabled.
1 = The FPSR and FPC registers are unfrozen, shadowing by these registers is en-
abled, and the FINT interrupt can be sampled.

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

AF — Alternate File Enable

0 = The general file is enabled.
1 = The alternate file is enabled.

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

C — Condition Code/Carry bit

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

Bits 30, 28, 23, 11, 10, 5, 3, 2 — Reserved for future use and must always be written to
zero.

2.2.2.1 Updates to the PSR

The content of the PSR can be modified by exception recognition and processing, the
rte and rfi instructions, and the mtcpr instruction. Each affects the PSR in different
ways.

2.2.2.2 Exception Recognition and Processing Updates

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

The `rte` and `rfi` instructions update the PSR. These updates may change the state of all bits in the PSR. Changes to the S, DB, TM, TP, TC, IE, FE and EE bits are effective prior to the fetch of the instruction at the return PC location. Changes to the U3-U0, VEC, MM, IC, AF, and C bits are effective prior to the execution of this instruction.

2.2.2.4 MTCR Instruction Updates

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

2.2.3 Vector Base Register

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

\[ \text{VBR — Vector Base Register} \]

<table>
<thead>
<tr>
<th>31</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR BASE</td>
<td>RESERVED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

2.2.4 Supervisor Storage Registers

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

2.2.5 Exception Shadow Registers

The EPSR, EPC, FPSR, and FPC registers are used during exceptions to store execution context of the processor. Refer to \textbf{SECTION 4 EXCEPTION PROCESSING} for details.
2.2.6 Global Control Register

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

2.2.7 Global Status Register

The 32-bit global status register (GSR) is used for global status reporting by devices and events external to the core. Thirty-two parallel inputs are provided at the core interface for implementation-defined purposes. Device status and other events can be easily monitored using the GSR. It is beyond the scope of this document to specify the exact meaning of the bits in this register. Refer to the appropriate microcontroller user’s manual for details. This register is read only. Writes to this register while in supervisor mode are ignored. (Writes to this register in user mode result in a privilege violation exception.)
SECTION 3 INSTRUCTIONS

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

3.1 Instruction Types and Addressing Modes

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

M•CORE implements three types of instructions: flow control, data memory access, and register-to-register operations. Flow control instructions alter the sequential flow of instruction execution. Data memory access instructions load or store operands to or from the general-purpose registers. Register-to-register instructions perform operations on general-purpose registers, or are used to access control registers. Instruction formats are shown in the following paragraphs.

3.1.1 Register-to-Register Instructions

M•CORE supports five addressing modes for register-to-register instructions as described in the next paragraphs.

3.1.1.1 Monadic Register Addressing Mode

Monadic register addressing uses a single 4-bit register field to specify the source/destination for an operation. Instructions with this format include \texttt{abs}, \texttt{asrc}, \texttt{brev}, \texttt{clrf}, \texttt{clrt}, \texttt{decf}, \texttt{decg}, \texttt{dect}, \texttt{decne}, \texttt{dct}, \texttt{ff1}, \texttt{inbf}, \texttt{inct}, \texttt{lsic}, \texttt{lsrc}, \texttt{mvc}, \texttt{mvcm}, \texttt{not}, \texttt{sexb}, \texttt{sext}, \texttt{stnbz}, \texttt{xsr}, \texttt{zexth}, and \texttt{zexth}.

![Figure 3-1 Monadic Format](image)

3.1.1.2 Dyadic Register Addressing Mode

Dyadic register addressing uses two 4-bit register fields encoded in the instruction to specify a source register and a source/destination register. For some instructions, only a single source value is used; the second register specifier is used as a destination specifier only. Instructions with this format include \texttt{addc}, \texttt{addd}, \texttt{and}, \texttt{andn}, \texttt{asr}, \texttt{bgenr}, \texttt{cmp[hltne]i}, \texttt{ixh}, \texttt{ixw}, \texttt{isli}, \texttt{lsr}, \texttt{mov}, \texttt{movf}, \texttt{movt}, \texttt{mult}, \texttt{or}, \texttt{rsub}, \texttt{subc}, \texttt{subt}, \texttt{st}, and \texttt{xor}.
3.1.1.3 Register with 5-Bit Immediate Mode

Register with 5-bit immediate addressing uses a 4-bit register field encoded in the instruction to specify a source/destination register and a 5-bit field to specify an unsigned immediate value as the second source operand. Instructions with this format include andi, asri, bclri, bgeni, bmaski, bseti, btsti, cmpnei, Isli, Isri, rotli, and rsbi.

3.1.1.4 Register with 5-Bit Offset Immediate Mode

Register with 5-bit offset immediate addressing uses a 4-bit register field encoded in the instruction to specify a source/destination register, and a 5-bit field to specify an unsigned immediate value as the second source operand. The binary encoding for the immediate value is offset by one from the actual immediate value, thus, offset immediate values fall in the range from 1 to 32, corresponding to binary encodings of 0 to 31. Instructions with this format include addi, subi and cmplti.

3.1.1.5 Register with 7-Bit Immediate Mode

Register with 7-bit immediate addressing uses a 4-bit register field encoded in the instruction to specify a destination register and a 7-bit field to specify an unsigned immediate value as the source operand. Only the movi instruction uses this format.
3.1.1.6 Control Register Addressing Mode
Control register addressing uses a 4-bit register field encoded in the instruction to specify a general-purpose source/destination register and a 5-bit field to specify a control register. Only the \texttt{mfcr} and \texttt{mtcr} instructions use this format.

![Control Register Addressing Format](image)

3.1.2 Data Memory Access Instructions
M•CORE supports four addressing modes for accessing memory-based operands.

3.1.2.1 Scaled 4-Bit Immediate Addressing Mode
The \texttt{ld} and \texttt{st} instructions use this addressing mode for effective address calculations. The contents of the general-purpose register specified by the RX instruction field are added to the unsigned 4-bit immediate field which has been scaled (shifted left) according to the size of the memory access to form the effective address for the access. Register RZ serves as the destination register for loads and as the source of store data for stores.

![Scaled 4-Bit Immediate Format](image)

3.1.2.2 Load/Store Register Quadrant Mode
The \texttt{ldq} and \texttt{stq} instructions use this mode to transfer a contiguous set of registers to or from the memory location pointed to by the contents of general-purpose register RX. Registers R4 through R7 are transferred in ascending order to or from memory.

![Load/Store Register Quadrant Format](image)

3.1.2.3 Load/Store Multiple Register Mode
The \texttt{ldm} and \texttt{stm} instructions use this mode to transfer a contiguous set of registers to or from the memory location pointed to by the contents of general-purpose register R0. The RF instruction field specifies the first register in the list to be transferred. Registers RF through R15 are transferred in ascending order to or from memory.
3.1.2.4 Load Relative Word Mode

The \texttt{lrw} instruction uses this format to address a 32-bit word located relative to the program counter (PC). The effective address is obtained by adding the zero-extended value of the 8-bit displacement field, scaled by four, to the value of $\text{PC} + 2$. The lower two bits of this value are truncated to 00, and a word is fetched from this location into the general-purpose register specified by the RZ instruction field.

\begin{figure}
\centering
\begin{tabular}{|c|c|c|}
\hline
15 & 12 & 11 \ \\
\hline
8 & 7 & 0 \ \\
\hline
\end{tabular}
\caption{Load Relative Word Format}
\end{figure}

3.1.3 Flow Control Instructions

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

3.1.3.1 Scaled 11-Bit Displacement Mode

The \texttt{br}, \texttt{bf}, \texttt{bt}, and \texttt{bsr} instructions use this addressing mode for branch target address calculations. The content of the program counter plus two ($\text{PC} + 2$) are added to the sign-extended 11-bit displacement field which has been scaled by two (shifted left by one bit).

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
15 & 11 \ \\
\hline
10 & 0 \ \\
\hline
\end{tabular}
\caption{Scaled 11-Bit Displacement Format}
\end{figure}

3.1.3.2 Register Addressing Mode

The \texttt{jmp} and \texttt{jsr} instructions use this addressing mode for effective address calculations. The target address is contained in the general-purpose register specified by the RX instruction field.

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
15 & 4 \ \\
\hline
3 & 0 \ \\
\hline
\end{tabular}
\caption{Register Addressing Format}
\end{figure}
3.1.3.3 Indirect Mode

The `jmp`i` and `jsr`i instructions use this format to address a 32-bit word located relative to the program counter (PC). The effective address is obtained by adding the zero-extended value of the 8-bit displacement field, scaled by four, to the value of PC + 2. The lower two bits of this value are truncated to 00, and a word is fetched from this location and loaded into the program counter (PC). If the value of the fetched word is even, instruction execution proceeds from the new program counter value; otherwise a misaligned access exception is taken.

![Figure 3-13 Indirect Format](image)

3.1.3.4 Register with 4-Bit Negative Displacement Mode

The `loopt` instruction uses this addressing mode for effective address calculations. The target address is formed by extending the DISP_4 instruction field with ones, shifting this negative number left by one to scale by two, and adding the resultant displacement to PC + 2. A count value is held in the general-purpose register specified by the RX instruction field.

![Figure 3-14 Register with 4-Bit Displacement Addressing Format](image)
### 3.2 Opcode Map

Table 3-1 shows the opcode map for M•CORE.

#### Table 3-1 Opcode Map

<table>
<thead>
<tr>
<th>Legend:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rrrr - RX field</td>
<td></td>
</tr>
<tr>
<td>ssss - RY field</td>
<td></td>
</tr>
<tr>
<td>zzzz - RZ field</td>
<td></td>
</tr>
<tr>
<td>ffff - Rfirst field</td>
<td></td>
</tr>
<tr>
<td>ccccc - control register specifier</td>
<td></td>
</tr>
<tr>
<td>iii..i - one of several immediate fields</td>
<td></td>
</tr>
<tr>
<td>ddddddddddd - branch displacement</td>
<td></td>
</tr>
<tr>
<td>bbbb - loopt displacement</td>
<td></td>
</tr>
<tr>
<td>uu- accelerator unit</td>
<td></td>
</tr>
<tr>
<td>ee..e - execution code</td>
<td></td>
</tr>
<tr>
<td>nnn - register count</td>
<td></td>
</tr>
<tr>
<td>p - update option</td>
<td></td>
</tr>
<tr>
<td>xx..x - undefined fields</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>bkpt</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>sync</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>rte</td>
</tr>
<tr>
<td>0000000000000011</td>
<td>rfi</td>
</tr>
<tr>
<td>00000000000000100</td>
<td>stop</td>
</tr>
<tr>
<td>00000000000000101</td>
<td>wait</td>
</tr>
<tr>
<td>00000000000000110</td>
<td>doze</td>
</tr>
<tr>
<td>00000000000000111</td>
<td></td>
</tr>
<tr>
<td>0000000000000101ii</td>
<td>trap #i</td>
</tr>
<tr>
<td>000000000000011xx</td>
<td></td>
</tr>
<tr>
<td>000000000000001rrrr</td>
<td>mvc</td>
</tr>
<tr>
<td>000000000001rrrr</td>
<td>mvcv</td>
</tr>
<tr>
<td>0000000000100rrrr</td>
<td>ldq</td>
</tr>
<tr>
<td>0000000000101rrrr</td>
<td>stq</td>
</tr>
<tr>
<td>0000000000110rrrr</td>
<td>ldm</td>
</tr>
<tr>
<td>0000000000111rrrr</td>
<td>stm</td>
</tr>
<tr>
<td>0000000001000rrrr</td>
<td>dect</td>
</tr>
<tr>
<td>0000000001001rrrr</td>
<td>decf</td>
</tr>
<tr>
<td>0000000001010rrrr</td>
<td>inct</td>
</tr>
<tr>
<td>0000000001011rrrr</td>
<td>incf</td>
</tr>
<tr>
<td>0000000001100rrrr</td>
<td>jmp</td>
</tr>
<tr>
<td>0000000001101rrrr</td>
<td>jsr</td>
</tr>
<tr>
<td>0000000001110rrrr</td>
<td>ff1</td>
</tr>
<tr>
<td>0000000001111rrrr</td>
<td>brev</td>
</tr>
<tr>
<td>00000000010000rrrr</td>
<td>xtrb3</td>
</tr>
<tr>
<td>00000000010001rrrr</td>
<td>xtrb2</td>
</tr>
<tr>
<td>00000000010010rrrr</td>
<td>xtrb1</td>
</tr>
<tr>
<td>00000000010011rrrr</td>
<td>xtrb0</td>
</tr>
<tr>
<td>00000000010100rrrr</td>
<td>zextb</td>
</tr>
<tr>
<td>00000000010101rrrr</td>
<td>sextb</td>
</tr>
<tr>
<td>00000000010110rrrr</td>
<td>zexth</td>
</tr>
</tbody>
</table>
Table 3-1 Opcode Map (Continued)

<table>
<thead>
<tr>
<th>Opcode Pattern</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000010111rrrr</td>
<td>sexth</td>
</tr>
<tr>
<td>000000011000rrrr</td>
<td>declt</td>
</tr>
<tr>
<td>000000011010rrrr</td>
<td>tstnbz</td>
</tr>
<tr>
<td>000000011010rrrr</td>
<td>decgt</td>
</tr>
<tr>
<td>000000011101rrrr</td>
<td>decne</td>
</tr>
<tr>
<td>000000011100rrrr</td>
<td>clrt</td>
</tr>
<tr>
<td>000000011101rrrr</td>
<td>clrf</td>
</tr>
<tr>
<td>000000011110rrrr</td>
<td>abs</td>
</tr>
<tr>
<td>000000011111rrrr</td>
<td>not</td>
</tr>
<tr>
<td>00000010ssssrrrr</td>
<td>movt</td>
</tr>
<tr>
<td>00000011ssssrrrr</td>
<td>mult</td>
</tr>
<tr>
<td>00000100ssssbbbb</td>
<td>loopt</td>
</tr>
<tr>
<td>00000101ssssrrrr</td>
<td>subu</td>
</tr>
<tr>
<td>00000110ssssrrrr</td>
<td>addc</td>
</tr>
<tr>
<td>00000111ssssrrrr</td>
<td>subc</td>
</tr>
<tr>
<td>00001000ssssrrrr</td>
<td></td>
</tr>
<tr>
<td>00001001ssssrrrr</td>
<td></td>
</tr>
<tr>
<td>00001010ssssrrrr</td>
<td>movf</td>
</tr>
<tr>
<td>00010010ssssrrrr</td>
<td>lsr</td>
</tr>
<tr>
<td>00001011ssssrrrr</td>
<td>cmpfs</td>
</tr>
<tr>
<td>00010101ssssrrrr</td>
<td>cmplt</td>
</tr>
<tr>
<td>00001110ssssrrrr</td>
<td>tst</td>
</tr>
<tr>
<td>00001111ssssrrrr</td>
<td>cmpne</td>
</tr>
<tr>
<td>00101000cccccccc</td>
<td>mfcr</td>
</tr>
<tr>
<td>00101010ssssrrrr</td>
<td>mov</td>
</tr>
<tr>
<td>00101011ssssrrrr</td>
<td>bgenr</td>
</tr>
<tr>
<td>00101010ssssrrrr</td>
<td>rsub</td>
</tr>
<tr>
<td>00101011ssssrrrr</td>
<td>ixw</td>
</tr>
<tr>
<td>00101110ssssrrrr</td>
<td>and</td>
</tr>
<tr>
<td>00101111ssssrrrr</td>
<td>xor</td>
</tr>
<tr>
<td>00101100cccccccc</td>
<td>mtcrc</td>
</tr>
<tr>
<td>00110101ssssrrrr</td>
<td>asr</td>
</tr>
<tr>
<td>00110111ssssrrrr</td>
<td>lsl</td>
</tr>
<tr>
<td>00111001ssssrrrr</td>
<td>addu</td>
</tr>
<tr>
<td>00111101ssssrrrr</td>
<td>ixh</td>
</tr>
<tr>
<td>00111110ssssrrrr</td>
<td>or</td>
</tr>
<tr>
<td>00111111ssssrrrr</td>
<td>andn</td>
</tr>
<tr>
<td>00100100iiiiiiii</td>
<td>addi</td>
</tr>
<tr>
<td>00100101iiiiiiii</td>
<td>cmplti</td>
</tr>
<tr>
<td>00101001iiiiiiii</td>
<td>subi</td>
</tr>
<tr>
<td>00100111iiiiiiii</td>
<td></td>
</tr>
<tr>
<td>00101001iiiiiiii</td>
<td>rsubi</td>
</tr>
<tr>
<td>00101011iiiiiiii</td>
<td>cmpnei</td>
</tr>
<tr>
<td>00101110100000rrrr</td>
<td>bmaski #32 (set)</td>
</tr>
<tr>
<td>0010111000001rrrr</td>
<td>divu</td>
</tr>
<tr>
<td>0010111001xxrrrr</td>
<td></td>
</tr>
<tr>
<td>0010111001xxrrrr</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-1 Opcode Map (Continued)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>001011001111rrrr</td>
<td>bmaski</td>
</tr>
<tr>
<td>001011001111rrrr</td>
<td>bmaski</td>
</tr>
<tr>
<td>001011111111rrrr</td>
<td>andi</td>
</tr>
<tr>
<td>001100011111rrrr</td>
<td>bclri</td>
</tr>
<tr>
<td>001100100000rrrr</td>
<td></td>
</tr>
<tr>
<td>001100100011rrrr</td>
<td>divs</td>
</tr>
<tr>
<td>001101000010rrrr</td>
<td></td>
</tr>
<tr>
<td>001101000110rrrr</td>
<td></td>
</tr>
<tr>
<td>001100100111rrrr</td>
<td>bgeni</td>
</tr>
<tr>
<td>001101001111rrrr</td>
<td>bgeni</td>
</tr>
<tr>
<td>001101111111rrrr</td>
<td>bseti</td>
</tr>
<tr>
<td>001111101111rrrr</td>
<td>btsti</td>
</tr>
<tr>
<td>001111000000rrrr</td>
<td>xsr</td>
</tr>
<tr>
<td>001111001111rrrr</td>
<td>rotli</td>
</tr>
<tr>
<td>001111010000rrrr</td>
<td>asrc</td>
</tr>
<tr>
<td>001111011111rrrr</td>
<td>asri</td>
</tr>
<tr>
<td>001111110000rrrr</td>
<td>lslc</td>
</tr>
<tr>
<td>001111101111rrrr</td>
<td>lsli</td>
</tr>
<tr>
<td>001111110000rrrr</td>
<td>lsic</td>
</tr>
<tr>
<td>001111111111rrrr</td>
<td>lsri</td>
</tr>
<tr>
<td>0100uu00eeeeeee</td>
<td>h_exec</td>
</tr>
<tr>
<td>0100uu01nnneeee</td>
<td>h_ret</td>
</tr>
<tr>
<td>0100uu01nnneeee</td>
<td>h_call</td>
</tr>
<tr>
<td>0100uu10111111rrrr</td>
<td>h_ld</td>
</tr>
<tr>
<td>0100uu10111111rrrr</td>
<td>h_st</td>
</tr>
<tr>
<td>0100uu10111111rrrr</td>
<td>h ld.h</td>
</tr>
<tr>
<td>0100uu10111111rrrr</td>
<td>h st.h</td>
</tr>
<tr>
<td>0101xxxxxxxxxxxx</td>
<td></td>
</tr>
<tr>
<td>011001111111rrrr</td>
<td>movi</td>
</tr>
<tr>
<td>01101xxxxxxxxxxx</td>
<td></td>
</tr>
<tr>
<td>0111zzzzddddd</td>
<td>lrw</td>
</tr>
<tr>
<td>01110000ddddddd</td>
<td>jmpi</td>
</tr>
<tr>
<td>01111111ddddddd</td>
<td>jsri</td>
</tr>
<tr>
<td>1000zzzziiiiirrr</td>
<td>ld</td>
</tr>
<tr>
<td>1001zzzziiiiirrr</td>
<td>st</td>
</tr>
<tr>
<td>1010zzzziiiiirrr</td>
<td>ld.b</td>
</tr>
<tr>
<td>1011zzzziiiiirrr</td>
<td>st.b</td>
</tr>
<tr>
<td>1100zzzziiiiirrr</td>
<td>ld.h</td>
</tr>
<tr>
<td>1101zzzziiiiirrr</td>
<td>st.h</td>
</tr>
<tr>
<td>11100ddddddddddd</td>
<td>bt</td>
</tr>
<tr>
<td>11101ddddddddddd</td>
<td>bf</td>
</tr>
<tr>
<td>11110ddddddddddd</td>
<td>br</td>
</tr>
<tr>
<td>11111ddddddddddd</td>
<td>bsr</td>
</tr>
</tbody>
</table>
3.3 Instruction Set
The following paragraphs provide detailed description of each M•CORE instruction. The descriptions are arranged in alphabetical order according to instruction mnemonic.
ABS

Absolute Value

Operation: \( RX \leftarrow |RX| \)

Assembler Syntax: abs rx

Description: Calculate the absolute value of register \( X \), and store the result in register \( X \). Note that an input operand of \( 0x80000000 \) yields a result of \( 0x80000000 \). No special indication of this is provided.

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

Register X field — Specifies register RX.
- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15
ADDCC  Unsigned Add with C Bit, Update C Bit

Operation: RX ← RX + RY + C, C ← carryout

Assembler
Syntax: addc rx,ry

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

Condition Code: C ← carryout

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Register Y field — Specifies register RY.
0000 – Register R0
0001 – Register R1
..
1111 – Register R15
ADDI

Unsigned Add with Immediate

Operation: \[ RX \leftarrow RX + \text{unsigned OIMM5} \]

Assembler Syntax: `addi rx,oimm5`

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OIMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15

OIMM5 field — Specifies immediate value to be added to RX. Note the binary encoding is offset by one from the actual value to be added.

- 00000 – add 1
- 00001 – add 2
- ..
- 11111 – add 32
ADDU
Unsigned Add

Operation: RX ← RX + RY

Assembler Syntax: addu rx,ry

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

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

Register Y field — Specifies register RY.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
AND

Logical AND

Operation: \( RX \leftarrow RX \land RY \)

Assembler Syntax: `and rx,ry`

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

Condition Code: Unaffected

Instruction Format:

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

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 – register Y
1 – register X

Instruction Fields:

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

Register Y field — Specifies source register RY.
- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15
ANDI Logical AND with Immediate

Operation: RX ← RX ∧ (unsigned IMM5)

Assembler Syntax: andi rx,imm5

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

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

IMM5 field — Specifies the unsigned 5-bit immediate value.
- 00000 – 0
- 00001 – 1
- ..
- 11111 – 31
ANDN
Logical AND NOT

Operation: RX ← RX \land (RY!)

Assembler Syntax: andn rx,ry

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Y field — Specifies source register RY.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15

Register X field — Specifies destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
ASR

Arithmetic Shift Right (Dynamic)

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

Assembler Syntax: `asr rx,ry`

Description: \( RX \leftarrow \text{asr}(RX) \) by \( RY[5:0] \) bits. IF \( RY[5:0] > 30 \), \( RX \leftarrow 0 \) or \(-1\). Arithmetically shift right the value in register \( X \) by the value of \( RY[5:0] \), and store the result in register \( X \). If the value of register \( Y[5:0] \) is greater than 30, \( RX \) will be 0 or \(-1\) depending on the original sign of \( RX \).

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Y field — Specifies source register \( RY \).
- 0000 – Register R0
- 0001 – Register R1
  ..
- 1111 – Register R15

Register X field — Specifies source/destination register \( RX \).
- 0000 – Register R0
- 0001 – Register R1
  ..
- 1111 – Register R15
ASRC

Arithmetic Shift Right by 1 Bit, Update C Bit

Operation: C ← RX[0], RX ← asr(RX) by 1.

Assembler Syntax: asrc rx

Description: RX ← asr(RX) by one bit into the C bit.

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

Instruction Format:

```
<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Instruction Fields:

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

Arithmetic Shift Right Immediate (Static)

Operation: RX ← asr(RX) by IMM5 bits.

Assembler Syntax: asri rx,imm5

Description: RX ← asr(RX) by IMM5 bits (1–31); arithmetically shift right the value in register X by the value of the IMM5 field.

Condition Code: Unaffected

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Instruction Fields:

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

IMM5 field — Specifies shift value. Must be in the range 1 to 31
   00001 – 1
   ..
   11111 – 31
BCLRI

Bit Clear Immediate

Operation: Clear bit [IMM5] of RX

Assembler Syntax: bclri rx,imm5

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

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

- IMM5 field — Specifies bit of RX to be cleared.
  - 00000 – Bit 0
  - 00001 – Bit 1
  - ..
  - 11111 – Bit 31
BF  Conditional Branch if False

**Operation:**  Conditional Branch if False:

- If (C==0),
  - \( PC \leftarrow PC + 2 + \text{(signed-extended 11-bit displacement } \ll 1) \)
- else
  - \( PC \leftarrow PC + 2 \)

**Assembler Syntax:**  bf label

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

**Condition Code:**  Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- Branch Displacement Field — Specifies the branch displacement.
**BGENI**

**Bit Generate Immediate (Static)**

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

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

**Description:** Set the bit of register X specified by the immediate field and clear all other bits of register X. Note that immediate values of zero to six are implemented using the `movi` instruction.

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- **Register X field** — Specifies destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **IMM5 field** — Specifies the single bit to be set in RX.

  Note that values for the immediate field of 00000 to 00110 are not allowed, as these opcodes are used for other instructions. Assemblers will map the mnemonics for `bgeni rx,#0-6` to the `movi` instruction. Disassembly will indicate the `movi` instruction instead of the original `bgeni` mnemonic.

  - 00111 – Bit 7
  - 01000 – Bit 8
  - 01001 – Bit 9
  - ..
  - 11111 – Bit 31
BGENR  Bit Generate Register (Dynamic)

Operation:  if $RY[5] = 0$, $RX \leftarrow 2^{RY[4...0]}$  else  $RX \leftarrow 0$

Assembler
Syntax:   bgenr rx,ry

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

Condition Code:  Unaffected

Instruction Format:

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

Instruction Fields:

Register X field — Specifies destination register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15

Register Y field — Specifies source register RY.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
BKPT  

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

**Assembler**  
**Syntax:**  bkpt  

**Description:**  Breakpoint.  

**Instruction Format:**

```
  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```
BMASKI
Bit Mask Generate Immediate

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

Assembler Syntax: `bmaski rx,imm5`

Description: Set the low-order IMM5 bits of register RX to one and clear the remaining upper bits. From one to 32 bits may be set. An IMM5 value of 32 is encoded in the instruction as an IMM5 field of 00000 by the assembler. Note that immediate values of one to seven are implemented using the `movi` instruction.

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>IMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15

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

Note that values for the immediate field of 00001 to 00111 are not allowed, as these opcodes are used for other instructions. Assemblers will map the mnemonics for `bmaski rx, #1-7` to the `movi` instruction. Disassembly will indicate the `movi` instruction instead of the original `bmaski` mnemonic.

- 00000 – Set bits 0 to 31
- 00001 to 00111 – Invalid
- 01000 – Set bits 0 to 7
- 01001 – Set bits 0 to 8
- ..
- 11111 – Set bits 0 to 30
BR

Unconditional Branch

**Operation:** Unconditional Branch; PC ← PC + 2 + (signed-extended 11-bit displacement <<1)

**Assembler Syntax:** 
br label

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Branch displacement field</td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Branch Displacement Field — Specifies the branch displacement.
Bit Reverse

Operation: Reverse the bits in RX

Assembler Syntax: `brev rx`

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

Condition Code: Unaffected

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

register X
```

Instruction Fields:

Register X field — Specifies source/destination register RX.

- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15
BSETI

Bit Set Immediate

Operation: Set bit [IMM5] of RX

Assembler Syntax: bseti rx,imm5

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

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

IMM5 field — Specifies the bit of RX to be set.
- 00000 – Bit 0
- 00001 – Bit 1
- ..
- 11111 – Bit 31
Branch to Subroutine

Operation: Branch to Subroutine:

\[
\begin{align*}
R15 & \leftarrow \text{PC} + 2 \\
\text{PC} & \leftarrow \text{PC} + 2 + (\text{signed-extended 11 bit displacement} \ll 1)
\end{align*}
\]

Assembler Syntax: `bsr label`

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch displacement field

Instruction Fields:

Branch Displacement Field — Specifies the branch displacement.
**BT**

Conditional Branch if True

**Operation:**
Conditional Branch if True:

\[
\text{If } (C=1) \\
\quad \text{PC} \leftarrow \text{PC} + 2 + (\text{signed-extended 11 bit displacement} \ll 1) \\
\text{else} \\
\quad \text{PC} \leftarrow \text{PC} + 2
\]

**Assembler Syntax:**
bt label

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch displacement field

**Instruction Fields:**
Branch Displacement Field — Specifies the branch displacement.
**BTSTI**  
Bit Test Immediate; Update C bit

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

**Assembler Syntax:**  
btsti rx,imm5

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

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

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>IMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- **Register X field** — Specifies source register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ...
  - 1111 – Register R15

- **IMM5 field** — Specifies which bit of RX is to be tested.
  - 00000 – Bit 0
  - 00001 – Bit 1
  - ...
  - 11111 – Bit 31
CLRF

Clear if Condition False

Operation: Conditionally clear RX to 0; if (C==0), RX ← 0.

Assembler Syntax: clrf rx

Description: If (C==0), RX ← 0. Move the value zero to RX when C bit is cleared.

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:
- Register X field — Specifies destination register RX.
  
 0000 – Register R0
  
 0001 – Register R1
  
  ..
  
  1111 – Register R15
CLRT  Clear if Condition True

Operation:  Conditionally clear RX to 0; if (C==1), RX ← 0

Assembler Syntax:  clrt rx

Description:  If (C==1), RX ← 0. Move the value zero to RX if the C bit is set.

Condition Code:  Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>register X</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies destination register RX.

0000 – Register R0
0001 – Register R1
.. 
1111 – Register R15
CMPHS

Compare for Higher or Same

Operation: Compare register X to register Y. If the value in register X is higher than or the same as the value in register Y,

\[ \text{Cbit} \leftarrow 1 \]

Else

\[ \text{Cbit} \leftarrow 0 \]

Assembler

Syntax: cmphs rx,ry

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

Condition Code: Set as a result of the comparison operation

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>register Y</td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies register RX.
- 0000 – Register R0
- 0001 – Register R1
  ..
- 1111 – Register R15

Register Y field — Specifies register RY.
- 0000 – Register R0
- 0001 – Register R1
  ..
- 1111 – Register R15
CMPLT  Compare for Less Than

Operation: Compare register X to register Y. If specified condition is true,
            Cbit ← 1
            else
            Cbit ← 0

Assembler Syntax: cmplt rx,ry

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

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

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

register Y  register X

Instruction Fields:

Register X field — Specifies register RX.

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

Register Y field — Specifies register RY.

- 0000 – Register R0
- 0001 – Register R1
  ..
- 1111 – Register R15
CMPLTI  Compare with Immediate for Less Than

Operation:  Compare register X with immediate value.  
            If specified condition is true,  
            Cbit ← 1  
            else  
            Cbit ← 0

Assembler Syntax:  cmplti rx, oimm5

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

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

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OIMM5       register X
```

Instruction Fields:

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

OIMM5 field — Specifies immediate value to be compared to RX.  Note the binary encoding is offset by one from the actual immediate value to be compared.  
  00000  – Compare with 1  
  00001  – Compare with 2  
  ..  
  11111  – Compare with 32
CMPNE

Compare for Not Equal

Operation: Compare register X to register Y.
If specified condition is true,
    Cbit ← 1
else
    Cbit ← 0

Assembler Syntax: cmpne rx,ry

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

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

Instruction Format:

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

Instruction Fields:

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

Register Y field — Specifies register RY.
    0000 – Register R0
    0001 – Register R1
    ..
    1111 – Register R15
CMPNEI  Compare with Immediate for Not Equal

Operation:  Compare register X with immediate value.
If specified condition is true,
   Cbit ← 1
else
   Cbit ← 0

Assembler Syntax:  cmpnei rx, imm5

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

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

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>IMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>register X</td>
</tr>
</tbody>
</table>

Instruction Fields:

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

  IMM5 field — Specifies immediate value to be compared with RX.
    00000 – Compare with 0
    00001 – Compare with 1
    ..
    11111 – Compare with 31
DECF
Decrement Conditionally on False

Operation: if C == 0, then
          RX ← RX – 1
else
          RX ← RX

Assembler Syntax: decf rx

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies the source/destination register RX.
0000 – Register R0
0001 – Register R1
...
1111 – Register R15
Decrement, Set C Bit on Greater Than

**Operation:** RX ← RX – 1
Update C bit

**Assembler Syntax:** decgt rx

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

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

**Instruction Format:**

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

**Instruction Fields:**

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
DECLT  Decrement, Set C Bit on Less Than

Operation:  RX ← RX – 1
            Update C bit

Assembler Syntax:  declt rx

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

Condition Code:  C bit is set to one if the result in RX is less than zero; otherwise the
                 C bit is cleared.

Instruction Format:

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

```
  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0
```

Instruction Fields:

  Register X field — Specifies source/destination register RX.
  0000 – Register R0
  0001 – Register R1
    ...
  1111 – Register R15
**DECNE**

Decrement, Set C Bit on Not Equal

**Operation:** RX ← RX − 1, update C bit

**Assembler Syntax:**

```plaintext
decne rx
```

**Description:** Decrement the value in register RX and set the C bit if the result is not equal to zero.

**Condition Code:** C bit is set to one if the result in RX is not equal to zero; otherwise the C bit is cleared.

**Instruction Format:**

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

**Instruction Fields:**

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
DECT
Decrement Conditionally on True

Operation: if C == 1 then,
          RX ← RX – 1
else
          RX ← RX

Assembler Syntax: dect rx

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>register X</td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source/destination register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
## DIVS

**Signed Divide RX by R1**

**Operation:**

\[
RX \leftarrow \frac{RX}{R1}
\]

**Assembler Syntax:**

divs rx,r1

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

**Condition Code:** Undefined

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Register X field — Specifies register RX.

- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
DIVU

Unsigned Divide RX by R1

Operation: \[ RX \leftarrow \frac{RX}{R1} \]

Assembler Syntax: `divu rx,r1`

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

Condition Code: Undefined

Instruction Format:

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

Instruction Fields:

- **Register X field** — Specifies register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
**DOZE**

Enter Low-Power Doze Mode

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

**Assembler Syntax:** doze

**Attributes:** Privileged

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

**Condition Code:** Unaffected

**Instruction Format:**

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
```
**FF1**  
Find First One in RX

**Operation:**  
RX ← ff1(RX)

**Assembler Syntax:**  
ff1 rx

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

**Condition Code:**  
Unaffected

**Instruction Format:**

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

**Instruction Fields:**

Register X field — Specifies source/destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
INCF
Increment RX Conditionally on False

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

Assembler Syntax:  incf rx

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

Condition Code:  Unaffected

Instruction Format:

```
0 0 0 0 0 0 0 1 0 1 1
```

Instruction Fields:
Register X field — Specifies source/destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
INCT  Increment RX Conditionally on True

Operation: if C == 1, then
            RX ← RX + 1
else
            RX ← RX

Assembler Syntax: inct rx

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

   Register X field — Specifies source/destination register RX.
      0000 – Register R0
      0001 – Register R1
           ..
      1111 – Register R15
IXH  Index Halfword

Operation: RX ← RX + [RY « 1]

Assembler Syntax: ixh rx,ry

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

Condition Code: Unaffected

Instruction Format:

Instruction Fields:

Register X field — Specifies source/destination register RX.

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

Register Y field — Specifies source register RY.

- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15
IXW  
Index Word

Operation:  
RX ← RX + [RY « 2]

Assembler Syntax:  
ixw rx,ry

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

Condition Code:  
Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Register Y field — Specifies source register RY.
- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15
**JMP**  
Unconditional Jump

**Operation:** Unconditional jump:

\[ \text{PC} \leftarrow (RX) \]

**Assembler Syntax:**
\[ \text{jmp \ rx} \]

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Register X field — Specifies source register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
JMPI  Unconditional Jump Indirect

Operation: Unconditional jump indirect:

PC ← MEM[(PC + 2 + (unsigned disp_8 << 2)) & 0xffffffff]

Assembler Syntax: jmpi [label]

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DISP_8</td>
</tr>
</tbody>
</table>

Instruction Fields:

DISP_8 Field — Unsigned 8-bit displacement
JSR  
Unconditional Jump to Subroutine

Operation:  Unconditional jump to subroutine:

\[
\begin{align*}
R15 & \leftarrow PC + 2, \\
PC & \leftarrow (RX)
\end{align*}
\]

Assembler Syntax:  
\text{jsr } rx

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

\textbf{CAUTION}  
RX must \textit{not} specify R15; this condition is undefined and undetected.

Condition Code:  Unaffected

Instruction Format:

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \text{register } X
\end{array}
\]

Instruction Fields:

Register X field — Specifies source register RX.  
- 0000 – Register R0  
- 0001 – Register R1  
..  
- 1110 – Register R14

1111 – Illegal specifier; do not use
JSRI  Unconditional Jump Subroutine Indirect

Operation:  
Unconditional jump subroutine indirect:

\[ R15 \leftarrow PC + 2, \]
\[ PC \leftarrow \text{MEM}[(PC + 2 + (\text{unsigned} \ disp_8 \ll 2)) \& 0xffffffff] \]

Assembler Syntax:  
\text{jsri [label]} 

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

Condition Code:  
Unaffected

Instruction Format:

\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Instruction Fields:

\text{DISP}_8 \text{ Field} — \text{Unsigned} 8\text{-bit displacement}
LD.[BHW]  Load Register from Memory

**Operation:** Source/destination register \( \leftarrow \) Memory location:

\[ RZ \leftarrow \text{MEM}[RX + \text{unsigned IMM4} \ll\{0,1,2\}] \]

**Assembler Syntax:**

\[ \text{ld.[bhw]} \ rz, (rx, disp) \]
\[ \text{[ld, ldb, ldh, ldw]} \ rz, (rx, disp) \]

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Size</td>
<td>0</td>
<td>register Z</td>
<td>IMM4</td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- **Size** — Specifies load size.
  - 00 – Word
  - 01 – Byte
  - 10 – Halfword

- **Register Z** — Specifies the destination register for the load.

- **IMM4 Field** — Specifies a 4-bit scaled immediate value.

- **Register X** — Specifies the base address to be added to the scaled immediate field.
LDM  Load Multiple Registers from Memory

Operation:  Destination Registers ← Memory

Assembler
Syntax:  ldm rf–r15,(r0)

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

Condition Code:  Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register First field — Specifies the first register to be transferred. Only R1–R14 should be specified.
**LDQ**

Load Register Quadrant from Memory

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

**Assembler Syntax:** ldq r4–r7,(rx)

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

**Condition Code:** Unaffected

**Instruction Format:**

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

**Instruction Fields:**

Register X — Specifies the base address for the transfers. Register X should not specify R4, R5, R6, or R7.
LOOPT

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

Operation:  if C == 1, then
             PC ← PC + 2 + (one’s extended 4-bit displacement << 1)
             RY ← RY –1
             Update C bit
         else
             RY ← RY –1
             Update C bit

Assembler
Syntax:     loopt ry, label

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

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

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Y field — Specifies source/destination register RY.
0000 – Register R0
0001 – Register R1
..  
1111 – Register R15

Disp_4 field — Loop displacement from PC + 2.
0000 – Displacement of –32
0001 – Displacement of –30
..  
1111 – Displacement of –2
LRW

Load PC-Relative Word

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

**Assembler Syntax:** lrw rz,[label]

**Description:** The DISP_8 displacement field is zero extended, scaled by four (i.e., left-shifted by two), and added to PC + 2. The low-order two bits of this address are forced to zero, and a word is loaded from this address into register RZ. RZ may not be R0 (the stack pointer by convention) or R15 (the link register). Note that only forward offsets from the PC are available for referencing the load data.

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>Register Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- **Register Z** — Specifies the destination register for the load. May not be R0 (0000) or R15 (1111).
- **DISP_8 Field** — Specifies an unsigned 8-bit displacement value.
LSL
Logical Shift Left (Dynamic)

Operation: \( RX \leftarrow \text{lsl}(RX) \) by \( \text{RY}[5:0] \) bits.

Assembler Syntax: \( \text{lsl rx,ry} \)

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

Condition Code: Unaffected

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Instruction Fields:

Register X field — Specifies source/destination register \( RX \).
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15

Register Y field — Specifies source register \( RY \).
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
LSLC

Logical Shift Left by 1, Update C Bit

Operation: \( C \leftarrow RX[31], RX \leftarrow \text{lsl}(RX) \) by 1.

Assembler Syntax: \text{lslc} \hspace{1em} \text{rx}

Description: \( RX \leftarrow \text{lsl}(RX) \) by 1 bit into the C bit

Condition Code: Copy \( RX[31] \) into the C bit before shifting occurs.

Instruction Format:

\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \text{register X} & \end{array}
\]

Instruction Fields:

Register X field — Specifies source/destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- \ldots
- 1111 – Register R15
**LSLI**

Logical Shift Left Immediate (Static)

**Operation:** RX ← lsli(RX) by IMM5 bits (1..31).

**Assembler Syntax:** lsli rx,imm5

**Description:** RX ← lsli(RX) by IMM5 bits (1..31); logically shift left the value in register X by the value of the IMM5 field.

**Condition Code:** Unaffected

**Instruction Format:**

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>IMM5</td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Instruction Fields:**

- **Register X field** — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **IMM5 field** — Specifies shift value. Must be in the range 1..31
  - 00001 – 1
  - ..
  - 11111 – 31
LSR
Logical Shift Right (Dynamic)

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

Assembler
Syntax: lsr rx,ry

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

Condition Code: Unaffected

Instruction Format:

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & \text{register Y} & \text{register X}
\end{array}
\]

Instruction Fields:

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

Register Y field — Specifies source register RY.
- 0000 – Register R0
- 0001 – Register R1
  ...
- 1111 – Register R15
LSRC

Logical Shift Right by 1 Bit, Update C Bit

Operation: C ← RX[0], RX ← lsr(RX) by 1.

Assembler Syntax: lsrc rx

Description: RX ← lsr(RX) by one bit into the C bit.

Condition Code: Copy RX0 into the C bit before shifting occurs.

Instruction Format:

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

Instruction Fields:

Register X field — Specifies source/destination register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
**LSRI**

Logical Shift Right Immediate (Static)

**Operation:** RX ← lsr(RX) by IMM5 bits (1..31).

**Assembler Syntax:** lsr rx,imm5

**Description:** RX ← lsr(RX) by IMM5 bits. Perform logical shift right of the value in register X by the value of the IMM5 field.

**Condition Code:** Unaffected.

**Instruction Format:**

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

**Instruction Fields:**

- **Register X field** — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **IMM5 field** — Specifies shift value. Must be in the range 1 to 31.
  - 00001 – 1
  - ..
  - 11111 – 31
MFCR  Move from Control Register

Operation: move from control register: RX ← CRy

Assemble Syntax: mfcr rx,cry

Attributes: Privileged

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

Condition Code: Unaffected

Instruction Format:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Instruction Fields:

Register X field — Specifies destination register RX.
  0000 – Register R0
  0001 – Register R1
 ..
  1111 – Register R15

CRegister Y field — Specifies source control register CRy.
  00000 – Control register CR0
  00001 – Control register CR1
 ..
  11111 – Control register CR31
MOV

Logical Move

Operation: RX ← RY

Assembler Syntax: mov rx,ry

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15

Register Y field — Specifies source register RY.
- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
**MOVF**

**Move RY to RX if Condition False**

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

**Assembler Syntax:**
`movf rx,ry`

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

**Condition Code:**
Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>register Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>register X</td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- **Register X field** — Specifies destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  
  ..
  - 1111 – Register R15

- **Register Y field** — Specifies source register RY.
  - 0000 – Register R0
  - 0001 – Register R1
  
  ..
  - 1111 – Register R15
**MOVI**

Logical Move Immediate

**Operation:**  
RX ← unsigned IMM7

**Assembler Syntax:**  
movi rx,imm7

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

**Condition Code:**  
Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>IMM7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Register X field — Specifies destination register RX.
- 0000 – Register R0
- 0001 – Register R1
- ...
- 1111 – Register R15

IMM7 field — Specifies immediate value to be moved to RX.
- 00000000 – 0
- 00000001 – 1
- ...
- 11111111 – 127
Move RY to RX if Condition True

Operation: Conditionally move RY to RX; if (C==1), RX ← RY

Assembler Syntax: movt rx,ry

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies destination register RX.
0000 – Register R0
0001 – Register R1
...
1111 – Register R15

Register Y field — Specifies source register RY.
0000 – Register R0
0001 – Register R1
...
1111 – Register R15
MTCR  Move to Control Register

Operation:    Move to control register: CRy ← RX

Assembler Syntax:  mtcr rx,cry

Attributes:    Privileged

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

Condition Code:  Unaffected unless CR0 (PSR) specified

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CRegY</td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

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

CRegister Y field — Specifies destination control register CRy.
  00000 – Control register CR0
  00001 – Control register CR1
  ..
  11111 – Control register CR31
**MULT**

Multiply

**Operation:** RX ← RX x RY

**Assembler Syntax:** mult rx,ry

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

**Condition Code:** Unaffected

**Instruction Format:**

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

**Instruction Fields:**

- **Register X field** — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **Register Y field** — Specifies source register RY.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
**MVC**  
Move C Bit to Register

**Operation:**  
RX ← C bit

**Assembler Syntax:**  
mvc rx

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

**Condition Code:**  
Unaffected

**Instruction Format:**

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

**Instruction Fields:**

- Register X field — Specifies destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
**MVCV**  
Move Inverted C Bit to Register

**Operation:**  
RX ← (C bit)!

**Assembler Syntax:**  
mvcv rx

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

**Condition Code:**  
Unaffected

**Instruction Format:**

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

**Instruction Fields:**  
Register X field — Specifies destination register RX.  
0000 – Register R0  
0001 – Register R1  
..  
1111 – Register R15
NOT

Logical NOT

Operation: RX ← (RX)!

Assembler Syntax: not rx

Description: Logically invert the value of register X.

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>register X</td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
Logical OR

**Operation:** RX ← RX ∨ RY

**Assembler Syntax:** or rx,ry

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

**Condition Code:** Unaffected

**Instruction Format:**

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

**Instruction Fields:**

- **Register X field** — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **Register Y field** — Specifies source register RY.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
RFI

Return from Fast Interrupt

Operation: \( \text{PC} \leftarrow \text{FPC (CR5)}; \)
\( \text{PSR} \leftarrow \text{FPSR (CR3)} \)

Assembler Syntax: \text{rfi}

Attributes: Privileged

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

Instruction Format:

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]
ROTLI

Operation: RX ← rotl(RX) by IMM5 bits (1..31)

Assembler Syntax: rotli rx,imm5

Description: RX ← rotl(RX) by IMM5 bits (1..31). Rotate the value in register X left by the value of the IMM5 field.

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>IMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source/destination register RX.

0000 – Register R0
0001 – Register R1
.. 1111 – Register R15

IMM5 field — Specifies rotate value. Must be in the range 1 to 31.

00001 – 1
.. 11111 – 31
RSUB
Reverse Subtract

Operation: RX ← RY – RX

Assembler Syntax: rsub rx,ry

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

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>registerY</td>
<td>registerX</td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source/destination register RX.

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

Register Y field — Specifies source register RY.

- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
RSUBI  Reverse Subtract with Immediate

Operation:  RX ← [unsigned IMM5] – RX

Assembler Syntax:  rsubi rx,imm5

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

Condition Code:  Unaffected

Instruction Format:

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

Instruction Fields:

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

- IMM5 field — Specifies immediate value to be used.
  - 00000 – 0
  - 00001 – 1
  - ..
  - 11111 – 31
RTE  

Return from Exception

Operation:  PC ← EPC (CR4); PSR ← EPSR (CR2)

Assembler Syntax:  rte

Attributes:  Privileged

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

Instruction Format:

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

**Sign Extend Byte**

**Operation:** RX ← RX[7:0] sign-extended to 32 bits

**Assembler Syntax:** sextb rx

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

**Condition Code:** Unaffected

**Instruction Format:**

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

**Instruction Fields:**

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
SEXTH  

Sign Extend Halfword

Operation:  RX ← RX[15:0] sign-extended to 32 bits

Assembler Syntax:  sexth rx

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

Condition Code:  Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source/destination register RX.
0000 – Register R0
0001 – Register R1
.. 
1111 – Register R15
ST.[B,H,W]  Store Register to Memory

Operation:  Memory ← Source Register:

\[ \text{MEM}[RX + \text{unsigned IMM4 } \ll{0,1,2}] \leftarrow \text{RZ} \]

Assembler Syntax:

\[ \text{st.[b,h,w] rz,(rx,disp)} \]

\[ [\text{st, stw sth stb}] \text{rz,(rx,disp)} \]

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

Condition Code:  Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Size</td>
<td>1</td>
<td>register Z</td>
<td></td>
<td>IMM4</td>
<td></td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Size — Specifies store size.
00 – Word
01 – Byte
10 – Halfword

Register Z — Specifies the source register for store data.

IMM4 Field — Specifies a 4-bit scaled immediate value.

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

**Store Multiple Registers to Memory**

**Operation:** Memory ← Source Registers

**Assembler Syntax:** `stm rf–r15,(r0)`

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>register first</td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Register First field — Specifies the first register to be transferred. Only R1–R14 should be specified.
STOP
Enter Low-Power Stop Mode

Operation: Enter stop mode

Assembler Syntax: stop

Attributes: Privileged

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

Condition Code: Unaffected

Instruction Format:

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

Store Register Quadrant to Memory

**Operation:** Memory ← Source Registers

**Assembler Syntax:**

```
stq r4–r7,(rx)
```

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

**Condition Code:** Unaffected

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>register X</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

Register X — Specifies the base address for the transfers. Register X should not specify R4, R5, R6, or R7.
SUBC  Unsigned Subtract with C Bit; Update C Bit

Operation:  RX ← RX – RY – (C!)
            C ← carryout

Assembler Syntax:  subc rx,ry

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

Condition Code:  C ← carryout

Instruction Format:

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

Instruction Fields:

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

    Register Y field — Specifies register RY.
        0000 – Register R0
        0001 – Register R1
    ... 1111 – Register R15
SUBI

Unsigned Subtract with Immediate

Operation: RX ← RX – [unsigned OIMM5]

Assembler Syntax: subi rx,oimm5

Description: Subtract the immediate value from the contents of register X. The immediate value must be in the range of 1 ... 32.

Condition Code: Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>OIMM5</td>
<td>register X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies register RX.

0000 – Register R0
0001 – Register R1
...
1111 – Register R15

OIMM5 field — Specifies immediate value to be subtracted from RX. Note the encoding is offset by one from the actual value to be subtracted:

00000 – Subtract 1
00001 – Subtract 2
...
11111 – subtract 32
SUBU  Unsigned Subtract

Operation:  RX ← RX – RY

Assembler Syntax:

subu rx,ry
sub rx,ry

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

Condition Code:  Unaffected

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies register RX.

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

Register Y field — Specifies register RY.

- 0000 – Register R0
- 0001 – Register R1
- ..
- 1111 – Register R15
SYNC

Synchronize CPU

**Operation:**  Cause the CPU to synchronize

**Assembler Syntax:**  sync

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

**Condition Code:**  Unaffected

**Instruction Format:**

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

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
```
TRAP

Unconditional Trap to OS

Operation: Cause a trap exception to occur

Assembler Syntax: trap #trap_number

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

Vec2 — 2-bit immediate field to describe trap number.
TST

Test with Zero

Operation: if (RX & RY) ≠ 0, then
            Cbit ← 1
else
            Cbit ← 0

Assembler Syntax: tst rx,ry

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

Condition Code: Set if (RX & RY) ≠ 0; cleared otherwise.

Instruction Format:

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

Instruction Fields:

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

Register Y field — Specifies register RY.
  0000 – Register R0
  0001 – Register R1
  ...
  1111 – Register R15
TSTNBZ  Test Register for No Byte Equal to Zero

Operation: If no byte of register X is zero, then
            Cbit ← 1
else
            Cbit ← 0

Assembler
Syntax:   tstnbz rx

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

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

Instruction Format:

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

Instruction Fields:

Register X field—Specifies register RX.
   0000 – Register R0
   0001 – Register R1
   ...
   1111 – Register R15
Wait

Stop Execution and Wait for Interrupt

Operation: Enter low-power wait mode

Assembler Syntax: wait

Attributes: Privileged

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

Instruction Format:

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

Logical Exclusive OR

**Operation:** \( RX \leftarrow RX \oplus RY \)

**Assembler Syntax:** `xor rx,ry`

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

**Condition Code:** Unaffected

**Instruction Format:**

```
\begin{array}{ccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & & & & & & & \text{register Y} & \text{register X}
\end{array}
```

**Instruction Fields:**

- **Register X field** — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15

- **Register Y field** — Specifies source register RY.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
**XSR**

**Extended Shift Right**

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

**Assembler Syntax:** `xsr rx`

**Description:** Shift RX right by one bit through the C bit, i.e.,

```
Ctmp ← C  
C ← RX[0]  
lsr(RX,1)  
RX[31] ← Ctmp
```

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

**Instruction Format:**

```
0 0 1 1 1 0 0 0 0 0 0 0 register X
```

**Instruction Fields:**

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
XTRB0  Extract High-Order Byte into R1 and Zero-Extend

Operation:  R1 ← byte 0 of RX (bits 31:24) zero extended to 32 bits

Assembler Syntax:  xtrb0 r1,rx

Description:  Extract high order byte of RX into R1 and zero-extend;
             R1[7:0] ← RX[31:24], R1[31:8] ← 0, C ← (result is ≠0?)

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

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>register X</td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
XTRB1  

Extract Byte 1 into R1 and Zero-Extend

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

Assembler Syntax:    xtrb1 r1,rx

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

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

Instruction Format:

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

Instruction Fields:

Register X field — Specifies source register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
XTRB2
Extract Byte 2 into R1 and Zero-Extend

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

Assembler Syntax: \text{xtrb2 r1,rx}

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

Condition Code: The C bit is set to one if the result is \( \neq 0 \) and cleared otherwise.

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>register X</td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source register RX.
- 0000 – Register R0
- 0001 – Register R1
  ...
- 1111 – Register R15
XTRB3  Extract Low-order Byte into R1 and Zero-Extend

Operation:  \( R1 \leftarrow \text{byte 3 of } RX \text{ zero extended to 32 bits} \)

Assembler
Syntax:  \text{xtrb3 r1,rx}

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

Condition Code:  The C bit is set to one if the result is \( \neq 0 \) and cleared otherwise.

Instruction Format:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>register X</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fields:

Register X field — Specifies source register RX.
  0000 – Register R0
  0001 – Register R1
  ..
  1111 – Register R15
**ZEXTB**  
Zero Extend Byte

**Operation:** RX ← low-order byte of RX zero-extended to 32 bits

**Assembler Syntax:**
```
zextb rx
```

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

**Condition Code:** Unaffected

**Instruction Format:**

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

**Instruction Fields:**

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ..
  - 1111 – Register R15
ZEXTH

Zero Extend Halfword

Operation: RX ← low-order half of RX zero-extended to 32 bits

Assembler Syntax: zexth rx

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

Condition Code: Unaffected

Instruction Format:

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

Instruction Fields:

- Register X field — Specifies source/destination register RX.
  - 0000 – Register R0
  - 0001 – Register R1
  - ... 1111 – Register R15
SECTION 4 EXCEPTION PROCESSING

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

4.1 Exception Processing Overview

Exception processing is the transition from the normal processing of a program to the processing required for any special internal or external condition that pre-empts normal processing. External conditions that cause exceptions are interrupts from external devices, hardware breakpoint requests, access errors, and resets. Internal conditions that cause exceptions are instructions, misalignment errors, privilege violations, and tracing. The trap and bkpt instructions generate exceptions as part of their normal execution. In addition, the following cause exceptions: illegal instructions; misaligned addresses for ld, ldm, ldq, st, stm, and stq instructions; odd valued destination addresses for the jmpi and jsri instructions; and privilege violations.

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

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

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

Exception processing occurs in the following steps:

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

On an unrecoverable error exception, the EPSR and ECP are still updated. If the unrecoverable error is due to an exception on an exception vector table fetch as part of exception processing (described below), the values saved in the EPSR and EPC are undefined.

Next, the processor changes to the supervisor mode by setting the PSR(S) bit and inhibits tracing of the exception handler by clearing the TM field in the PSR. The interrupt enable flag (IE) is also cleared to inhibit normal interrupt recognition. Fast interrupt exceptions and resets clear the fast interrupt enable (FE) bit; it is unaffected by other exceptions.

The translation control (TC) bit in the PSR is cleared to disable address translation by an optional external memory management unit for the remainder of exception processing, allowing the following accesses to be performed untranslated. The exception handler may re-enable translation as appropriate.

2. During this step, the processor determines the vector number for the exception. For vectored interrupts, the processor latches the vector number directly from the interrupt controller interface to the core. For all other exceptions, internal logic provides the vector number. This vector number is used in the last step to calculate the address of the exception vector by multiplying it by four to convert it to a vector offset. The vector number associated with the exception is loaded into the VEC field of the PSR to assist shared exception handlers.

3. During this step, the processor determines the address of the first instruction of the exception handler and then passes control to the handler. The processor combines the vector offset with the value contained in the vector base register to obtain the memory address of the exception vector. Next, the processor fetches a word from the vector table entry, loads the new program counter (PC) value from the exception vector table entry with the address of the first instruction of the exception handler, and loads the PSR(AF) control bit from the low order bit of the vector table entry to determine which register file to use when the exception handler is entered. The processor then resumes execution at the new PC location.
4.3 Exception Vectors

All exception vectors are located in the supervisor address space and are accessed using program relative references. Only the reset and soft reset vectors are fixed in the processor’s memory map. Once initialization is complete, the base address of the exception vector table can be relocated after reset by programming the VBR.

The M•CORE supports a 512-byte vector table containing 128 exception vectors (see Table 4-1). The first 32 vectors are used for internally recognized exceptions. External devices such as an interrupt controller are provided with an additional 96 vectors. These vectors are used by supplying a seven-bit vector number along with an interrupt request. The M•CORE latches the interrupt vector number when the interrupt request is accepted, and vectors to the appropriate location. For external devices that cannot provide a vector, an autovector capability is provided. Separate autovectors are provided for normal interrupts (INT) and fast interrupts (FINT).

Table 4-1 Exception Vector Assignments

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

The following paragraphs describe the external interrupt exceptions and the different types of exceptions generated internally by the M•CORE. The following exceptions are discussed:

• Reset
• Misaligned access
• Access error
• Divide by zero
• Illegal instruction
• Privilege violation
• Trace
• Breakpoint
• Unrecoverable error
• Soft reset
• Interrupt
• Fast interrupt
• Hardware accelerator
• Trap instructions

4.4.1 Reset Exception (Vector Offset 0x0)

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

The reset exception places the processor in the supervisor mode by setting the S bit and disables tracing by clearing the TM field in the PSR. This exception also clears the PSR interrupt enable bits IE and FE, the debug mode bit (DB), and the hardware accelerator enable bits U3-U0. The vector base register (VBR) is cleared to place the base of the exception table at address zero (0x00000000). The CPU fetches the reset vector from offset $0 in the exception vector table, which is then loaded into the PC. Reset exception handling proceeds with the transfer of control to the memory location pointed to by the PC.

4.4.2 Misaligned Access Exception (Vector Offset 0x4)

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

In addition to data-related misaligned access exceptions, the jmpi and jsri instructions cause misaligned access exceptions to occur if the destination address of these change-of-flow instructions is odd. In this case, the EPC contains the value fetched, not the address of the jmpi or jsri instruction. This is the only condition in which the EPC value is odd when an exception handler is entered.
4.4.3 Access Error Exception (Vector Offset 0x8)
An access error exception occurs under certain conditions when the transfer error acknowledge (TEA) signal is asserted externally to terminate a bus cycle.

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

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

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

4.4.4 Divide-by-Zero Exception (Vector Offset 0x0C)
Exception processing for divide instructions with a divisor of zero is similar to that for instruction traps. When the processor detects a divisor of zero for a divide instruction, it initiates exception processing instead of attempting to execute the instruction. On exception handler entry, the EPC points to the divide instruction.

4.4.5 Illegal Instruction Exception (Vector Offset 0x10)
Exception processing for illegal instructions is similar to that for instruction traps. When the processor decodes an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. On exception handler entry, the EPC points to the illegal instruction.

4.4.6 Privilege Violation Exception (Vector Offset 0x14)
To provide system security, certain instructions are privileged. An attempt to execute one of the following privileged instructions while in the user mode causes a privilege violation exception: mfcr, mtcr, rfi, rte, stop, wait, and doze.

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

4.4.7 Trace Exception (Vector Offset 0x18)
To aid in program development, the M•CORE includes an instruction-by-instruction and instruction change of flow tracing capability. In the instruction trace mode, each instruction generates a trace exception after the instruction completes execution, allowing a debugging program to monitor execution of a program. In change of flow trace mode, a trace exception is taken after each instruction which could cause a change of flow (branch, jmp, etc.) The exception is taken regardless of the outcome of a conditional branch or loop instruction.
The following instructions cause trace exceptions to be generated in the change-of-flow trace mode: jmp, jsr, jmpi, jsri, br, bt, bf, bsr, loopt. The trace exception is enabled regardless of the outcome of a conditional change-of-flow instruction.

If an instruction to be traced does not complete due to an instruction-related exception, trace exception processing is deferred, and no trace exception is taken or marked pending. If an interrupt is pending at the completion of an instruction to be traced, trace exception processing is deferred and is marked as pending in the EPSR as part of exception recognition.

The TM field in the PSR controls tracing. The state of the TM field when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. See 2.2.2 Processor Status Register for the definition of the TM field.

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

Certain control related instructions (rte, rfi, trap, stop, wait, doze and bkpt) are never traced, although a trace exception may be taken as part of the normal execution of an rte or rfi instruction if the EPSR(TP) bit is set. This occurs independent of the setting of the TM field in the PSR or EPSR.

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

4.4.8 Breakpoint Exception (Vector Offset 0x1C)

The Breakpoint instruction bkpt and the hardware breakpoint request input (BRKREQ) are assigned a unique exception vector. Refer to the appropriate microcontroller user’s manual for operation of the BRKREQ input. To minimize the chance that hardware breakpoint exceptions are lost, this exception has higher priority than interrupts when generated as a result of the BRKREQ input signal. If the BRKREQ input is asserted for an instruction prefetch, that instruction will not be executed, and a breakpoint exception will be taken if the instruction normally would begin execution (i.e., is not discarded as the result of a change of instruction flow). If the BRKREQ input is asserted for a data fetch, a breakpoint exception is taken after the instruction completes. For the bkpt instruction, or for instruction accesses on which the BRKREQ input is asserted, the EPC points to the instruction on exception handler entry. For data accesses that are marked with a BRKREQ request, the EPC points to the next instruction.
4.4.9 Unrecoverable Error Exception (Vector Offset 0x20)

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

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

4.4.10 Soft Reset Exception (Vector Offset 0x24)

A soft reset exception is recognized when the SRST input signal is asserted. This exception is non-maskable. The soft reset exception has the highest priority of any exception below a hard reset; it provides for system recovery from catastrophic failure. A soft reset also aborts any processing in progress when SRST is recognized; processing cannot be recovered. Soft reset exception processing begins once the SRST input is negated.

The soft reset exception places the processor in the supervisor mode by setting the S bit and disables tracing by clearing the TM field in the PSR. This exception also clears the processor’s interrupt enable bits IE and FE in the PSR and the exception shadowing enable bit EE. The hardware accelerator enable bits U3-U0 and the misalignment mask bit MM are undefined. The vector base register (VBR) is cleared to place the base of the exception table at address zero (0x00000000). The CPU fetches the soft reset vector from offset 0x24 in the exception vector table, which is then loaded into the PC. Reset exception handling proceeds with the transfer of control to the memory location pointed to by the PC.

4.4.11 Interrupt Exceptions

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

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

Two signals are provided for requesting interrupts, and both autovectoring and explicit vectoring capability is provided.
Figure 4-1 shows the interrupt related interface signals to the CPU core. For vectorable interrupts (FINT and INT), a seven-bit interrupt vector number can be supplied at the time a request is generated, or the autovector input AVEC can be asserted to indicate that the appropriate predefined vector should be used. If AVEC is asserted, the VEC# inputs are ignored.

4.4.11.1 Normal Interrupt (INT)

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

4.4.11.2 Fast Interrupt (FINT)

FINT is the fast interrupt request input. If enabled (PSR(FE) set), it has higher priority than the normal interrupt input INT. Fast interrupts use the FPSR and FPC exception shadow registers and consequently are not masked when PSR(EE) is cleared. Fast interrupts are masked when PSR(FE) is cleared. When FINT is asserted, either the AVEC input can be asserted to cause autovectoring to occur, or a 7-bit vector number can be provided to select one of vectors 32–127 to be used. (No explicit attempt is made by the processor to preclude use of vectors 0–31.) If a fast interrupt is autovectored, the vector at offset 0x2C from the vector table base is used.

4.4.12 Hardware Accelerator Exception (Vector Offset 0x30)

The M•CORE provides a set of enable bits in the PSR (U3-U0) for controlling execution of hardware accelerator instructions. When an attempt to execute an hai opcode associated with a disabled accelerator block occurs, the instruction is aborted, and this exception is taken. This exception vector is also used for exceptions reported by a hardware unit as part of hai instruction execution. Refer to the appropriate microcontroller user’s manual for details of the operation of the hai instruction and the hardware accelerator interface.
4.4.13 Instruction Trap Exception (Vector Offset 0x40-0x5C)

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

4.5 Exception Priorities

Exceptions can be divided into the five groups identified in Table 4-2. These groups are defined by specific characteristics and the order in which they are handled. Table 4-2 represents the priority used for simultaneous faults, as viewed by the M•CORE hardware. In Table 4-2, 1.0 represents the highest priority and 6 represents the lowest. Note that there are shared priorities for exceptions within group 4 and 5, since these types are mutually exclusive.

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

Within the M•CORE, more than one exception can occur at the same time. The reset exception is unique; resets override all other exceptions which may have occurred at the same time. All other exceptions are handled according to the priorities defined in Table 4-2.

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

When multiple exceptions are pending, the exception with the highest priority is processed first, and the remaining exceptions may be regenerated when the original faulting instruction is restarted. Table 4-3 shows the relationships between certain exceptions.
Table 4-3 Exceptions, Tracing and BRKRQ Results

1. Exception recognized on decode boundary for instruction N

<table>
<thead>
<tr>
<th>Ill/Priv/trap/bkpt</th>
<th>dacc</th>
<th>dbkpt</th>
<th>iacc (tea on N)</th>
<th>ibkpt (brkrq on N)</th>
<th>Interrupt Pending</th>
<th>Misalign</th>
<th>Trace of Previous inst is Pending</th>
<th>Saved TP in EPSR</th>
<th>Saved EPC</th>
<th>Exception Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>No exception</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>ill / priv / trap / bkpt</td>
</tr>
<tr>
<td>x</td>
<td>—</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Interrupt</td>
</tr>
<tr>
<td>x</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>x</td>
<td>—</td>
<td></td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>x</td>
<td>—</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>N</td>
<td>Trace (for prev inst)</td>
</tr>
<tr>
<td>x</td>
<td>—</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>Interrupt</td>
</tr>
<tr>
<td>x</td>
<td>—</td>
<td>—</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>Breakpoint</td>
</tr>
</tbody>
</table>

2. Exception recognized during execution of mult/div instruction M

<table>
<thead>
<tr>
<th>Interrupted (IC bit)</th>
<th>dacc (tea on acc)</th>
<th>dbkpt (brkrq on acc)</th>
<th>iacc (tea on M)</th>
<th>ibkpt (brkrq on M)</th>
<th>Interrupt Pending</th>
<th>Misalign</th>
<th>Trace of this inst Enabled</th>
<th>Saved TP in EPSR</th>
<th>Saved EPC</th>
<th>Exception Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Divide by zero</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Interrupt</td>
<td></td>
</tr>
</tbody>
</table>

3. Exception recognized during execution of ld/st class instruction M

<table>
<thead>
<tr>
<th>Interrupted (IC bit)</th>
<th>dacc (tea on acc)</th>
<th>dbkpt (brkrq on acc)</th>
<th>iacc (tea on M)</th>
<th>ibkpt (brkrq on M)</th>
<th>Interrupt Pending</th>
<th>Misalign</th>
<th>Trace of this inst Enabled</th>
<th>Saved TP in EPSR</th>
<th>Saved EPC</th>
<th>Exception Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>M</td>
<td>Misaligned</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Interrupt</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>M+2</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>M+2</td>
<td>Breakpoint</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>M+2</td>
<td>Breakpoint</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>M+2</td>
<td>Breakpoint</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>1</td>
<td>M+2</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Access error</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>x</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Breakpoint</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>—</td>
<td>x</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>x</td>
<td>0</td>
<td>M</td>
<td>Breakpoint</td>
<td></td>
</tr>
</tbody>
</table>
4.6 Returning from Exception Handlers

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

–A–
ABS instruction 3-10
Absolute value 3-10
Access error exception 4-5
ADDC instruction 3-11
ADDI instruction 3-12
Addressing modes 3-1
ADDU instruction 3-13
AF bit 1-3, 2-3, 2-6
Alternate file 1-3, 2-3, 2-6
AND instruction 3-14
ANDI instruction 3-15
ANDN instruction 3-16
Arithmetic Shift Right
by 1 Bit 3-18
Dynamic 3-17
Immediate (Static) 3-19
ASR instruction 3-17
ASRC instruction 3-18
ASRI 3-19
AVEC 4-8

–B–
BCLRI instruction 3-20
BF instruction 3-21
BGENI instruction 3-22
BGENR instruction 3-23
Big-endian byte ordering 1-4
Bit
Clear Immediate 3-20
Generate
Immediate (Static) 3-22
Register (Dynamic) 3-23
Mask Generate Immediate 3-25
Reverse 3-27
Set Immediate 3-28
Test Immediate 3-31
BKPT instruction 3-24, 4-6
BMASKI instruction 3-25
BR instruction 3-26
Branch
if True 3-30
on True, Decrementing Count 3-59
to Subroutine 3-29
Breakpoint
exception 4-6
instruction 3-24

–C–
C bit 1-2, 1-3, 2-2, 2-6
Clear
if Condition False 3-32
if Condition True 3-33
CLRIF instruction 3-32
CLRT instruction 3-33
CMPSH instruction 3-34
CMPLT instruction 3-35
CMPLTI instruction 3-36
CMPNE instruction 3-37
CMPNEI instruction 3-38
Compare
for Higher or Same 3-34
for Less Than 3-35
with Immediate
for Less Than 3-36
for Not Equal 3-38
Condition code/carry bit 1-2, 1-3, 2-2, 2-6
Conditional Branch
if False 3-21
if True 3-30
Control registers 1-3
addressing 3-3

–D–
Data memory access instructions 3-3
Data organization
in memory 1-5
in registers 1-5
DECIF instruction 3-39
DECGT instruction 3-40
DECLT instruction 3-41
DECNNE instruction 3-42
Decrement
Conditionally on False 3-39
Conditionally on True 3-43
set C bit on Greater Than 3-40
set C bit on Less Than 3-41
Set C bit on Not Equal 3-42
DECT instruction 3-43
Divide-by-zero exception 4-5
DIVS instruction 3-44
DIVU instruction 3-45
DOZE instruction 3-46
Dyadic register addressing 3-1

EE bit 2-5
EPC 4-2
EPSR 1-3, 4-2, 4-6
Exceptions 4-1
  enable bit 2-5
  priorities 4-9
  processing 1-3
  stages 4-2
  returning from 4-11
  shadow registers 1-4, 2-7
  types 4-4
  vectors 4-3
Extended Shift Right 3-98
Extract
  Byte 1 into R1 and Zero-Extend 3-100
  Byte 2 into R1 and Zero-Extend 3-101
  High-order Byte into R1 and Zero-Extend 3-99
  Low-order Byte into R1 and Zero-Extend 3-102

Fast interrupts 4-8
  enable bit 2-6
FE bit 2-6
FF1 instruction 3-47
Find First One in RX 3-47
FINT 4-8
Flow control instructions 3-4
FPC 4-2
FPSR 1-3, 4-2

GCR 2-8
General-purpose registers 1-3, 2-1
Global
  control register 2-8
  status register 2-8
GPRs 2-1
GSR 2-8

Hardware accelerator
  control bits 2-4
  exception 4-8

IC bit 2-6
IE bit 2-6
Illegal instruction exception 4-5
INCF instruction 3-48
Increment RX Conditionally
  on False 3-48
  on True 3-49
INCT instruction 3-49
Index
  Halfword 3-50
  Word 3-51
Indirect mode 3-5
Instructions 3-1
  register-to-register 3-1
  timing 1-2
  trap exception 4-9
INT 4-8
Interrupts
  control bit 2-6
  enable bit 2-6
  exception 4-7
  Fast 4-8
  normal 4-8
IXH instruction 3-50
IXW instruction 3-51

JMP instruction 3-52
JMPI instruction 3-53
JSR instruction 3-54
JSRI instruction 3-55

LD instruction 3-56
LDM instruction 3-57
LDQ instruction 3-58
Link register 1-3
Load
  Multiple Registers from Memory 3-57
  PC-Relative Word 3-60
  Register from Memory 3-56
  Register Quadrant from Memory 3-58
  relative word mode 3-4
  store multiple register mode 3-3
  store register quadrant mode 3-3
Load and store 1-2
Logical
  AND 3-14
  AND NOT 3-16
  AND with Immediate 3-15
  Exclusive OR 3-97
  Move 3-68
  Move Immediate 3-70
  NOT 3-76
  OR 3-77
Shift Left
by 1, Update C Bit 3-62
Dynamic 3-61
Immediate (Static) 3-63
Shift Right
by 1 bit, Update C Bit 3-65
Dynamic 3-64
Immediate (Static) 3-66
LOOPT instruction 3-59
Low-Power
Doze Mode, Enter 3-46
Stop Mode, Enter 3-87
LRW 3-60
LSL instruction 3-61
LSLC instruction 3-62
LSLI instruction 3-63
LSR instruction 3-64
LSRC instruction 3-65
LSRI instruction 3-66

Memory
load and store 1-2
organization 1-4
MFCR instruction 1-3, 3-67
Misaligned access exception 4-4
Misalignment exception mask 2-5
MM bit 2-5
Monadic register addressing 3-1
MOV instruction 3-68
Move C Bit to Register 3-74
Move from Control Register 1-3, 3-67
Move Inverted C Bit to Register 3-75
Move RY to RX if Condition False 3-69
Move RY to RX if Condition True 3-71
Move to Control Register 1-3, 3-72
MOVF instruction 3-69
MOVI instruction 3-70
MOV instruction 3-71
MTCR instruction 1-3, 2-7, 3-72
MULT instruction 3-73
Multiply 1-2, 3-73
MVC instruction 3-74
MVCV instruction 3-75

Normal interrupts 4-8
NOT instruction 3-76

Opcode map 3-6
OR instruction 3-77

PC 1-2, 1-3, 2-2
Pipeline, execution 1-2
Privilege modes 1-3
supervisor 1-3
user 1-3
Privilege violation exception 4-5
Processor status register 2-4
updating 2-6
Program counter 1-2, 1-3, 2-2
Programming model 1-3
supervisor 2-2
user 2-1
PSR 2-4
updating 2-6

R0 1-3, 1-5, 2-3
R15 1-3
Register
addressing mode 3-4
plus-four-bit scaled displacement 1-5
to-register instructions 3-1
with 4-bit negative displacement mode 3-5
with 5-bit immediate addressing 3-2
with 5-bit offset immediate addressing 3-2
with 7-bit immediate addressing 3-2
Registers 2-1
Reset exception 4-4
soft 4-7
Return
from Exception 1-3, 3-82
from Fast Interrupt 1-3, 3-78
Reverse Subtract 3-80
with Immediate 3-81
RFI instruction 1-3, 2-7, 3-78
RISC 1-1
Rotate Left Immediate (Static) 3-79
ROT instruction 3-79
RSUB instruction 3-80
RSUBI instruction 3-81
RTE instruction 1-3, 2-7, 3-82

S bit 1-3, 2-4
SC bit 2-5
Scaled
11-bit displacement mode 3-4
4-bit immediate addressing 3-3
Scratch registers 1-3, 1-4
SEXTB instruction 3-83
SEXT instruction 3-84
Shadow registers 1-3, 1-4, 2-7, 4-2
Sign Extend
Byte 3-83
Halfword 3-84
Signed Divide RX by R1 3-44
Soft reset exception 4-7
SP bit 2-4
Spare bits 2-4
Spare control 2-5
ST instruction 3-85
Stack pointer 1-3, 1-5, 2-3
    supervisor 2-7
Status registers 1-3
STM instruction 3-86
STOP instruction 3-87
Store
    Multiple Registers to Memory 3-86
    Register Quadrant to Memory 3-88
    Register to Memory 3-85
STQ instruction 3-88
SUBC 3-89
SUBI instruction 3-90
Subroutine calls 1-3
SUBU instruction 3-91
Supervisor
    bit 2-4
    mode 1-3
    programming model 2-2
    stack pointer 2-7
    storage registers 2-7
SYNC instruction 3-92

–T–

TC bit 2-5
TEA signal 4-5
Test
    Register for No Byte Equal to Zero 3-95
    with Zero 3-94
TM bit 4-6
TM field 2-5
TP bit 2-5, 4-6
Trace
    exception 4-5
    mode 2-5, 4-6
    pending 2-5, 4-6
Transfer error acknowledge 4-5
Translation control 2-5
TRAP instruction 3-93
TST instruction 3-94
TSTNBZ instruction 3-95

–U–

U bits 2-4
Unconditional
    Branch 3-26
    Jump 3-52
    Indirect 3-53
    Subroutine Indirect 3-55
    to Subroutine 3-54
    Trap to OS 3-93
Unrecoverable error exception 4-7

Unsigned
    Add 3-13
        with C Bit 3-11
        with Immediate 3-12
    Divide RX by R1 3-45
    Subtract 3-91
        with C Bit 3-89
        with Immediate 3-90
User
    mode 1-3
    programming model 2-1

–V–

VBR 2-7
VEC field 2-5
VEC# 4-8
Vectors, exception 4-3
    base register 2-7
    number 2-5

–W–

WAIT instruction 3-96

–X–

XOR instruction 3-97
XSR instruction 3-98
XTRBO 3-99
XTRB1 instruction 3-100
XTRB2 instruction 3-101
XTRB3 instruction 3-102

–Z–

Zero Extend
    Byte (ZEXTB) instruction 3-103
    Halfword (ZEXTH) instruction 3-104