XM-23
Instruction Set Architecture

Larry Hughes, PhD

30 April 2023

Contents

1 Introduction

XM-23 is the latest generation of the X-Makina 16-bit Instruction Set Architecture family of processors. XM-23 includes some of the features found in the XMX processor and the earlier versions of the XM series. The most significant change is the expansion of the transfer of control (branching) instruction set to include eight new instructions.

In XM-23, the 16-bit Instruction Set Architecture (ISA) with 64-KiB of random-access memory remains unchanged. Instructions and data are fetched from, and data is written to, memory by the central processing unit (CPU) over the 16-bit system bus. XM-23 can communicate with the outside world using any of its eight devices. Device status changes can be detected by the CPU polling or the device signaling the Interrupt Controller to cause a CPU interrupt.

The basic architecture of XM-23 is shown in Figure 1.

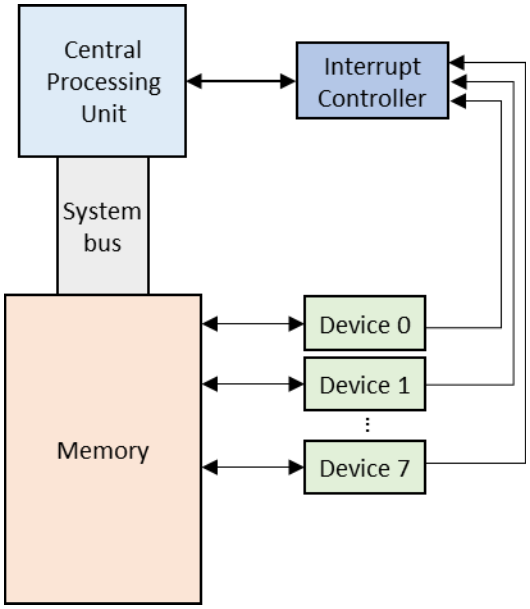

Figure 1: The XM-23 architecture

XM-23 is load-store machine that supports the following features:

- A total of 40 instructions supporting memory access, initialization, arithmetic, logic, and transfer of control. Each instruction is 16-bits wide. Many instructions can operate on a byte, a word, or both. Most of the mnemonics for the arithmetic and logic instructions are based on those found in a variety of machines, while register initialization, memory access instruction mnemonics, and branching are like ones found in the ARM Cortex.

- A total of eight 16-bit registers: three special purpose (program counter, stack pointer, and link register) and five general purpose (for data, addressing, or both). The link register can also be used as a general-purpose register by storing it on the stack, giving the programmer access to six general-purpose registers.

- Some instructions allow the programmer to use one of eight pre-defined constants in place of a source register. This can reduce the number of registers used in an arithmetic operation. A similar approach is used by the TI MSP-430.
- An additional 26 instructions that can be emulated by using existing instructions, allowing operations such as subroutine call and return, interrupt return, and stack push and pull. Many of these instructions are based on the TI MSP-430's emulated instructions.
- Instructions are classified into one of: arithmetic and logical, data manipulation, (consisting of an opcode and one or two operands), transfer of control (a signed PC-relative value), memory access, and special purpose. Each instruction is associated with one or more of XM-23's five addressing modes: register, direct, indexed (pre- and post- auto-increment and auto-decrement), relative, and immediate.
- Eight-levels of priority, up to eight devices for input or output, and 16 interrupt vectors for exceptions (i.e., device interrupts, faults, and traps). The interrupt structure is based on that of the ARM Cortex.
- XM-23 has a RISC architecture and can be implemented as either a traditional von Neumann or pipelined architecture.

1.1 Terminology

The following terminology is used in this text:

- A *bit* is a binary value, either zero, '0', or one, '1'.
- A contiguous grouping of 8-bits is a *byte* and a contiguous grouping of 16-bits (two-bytes) is a *word*.
- A byte or a word is referred to collectively as a *unit*.
- Bits in a unit are numbered from right-to-left, starting at zero.
- The right-most bit (bit '0') is the *least-significant bit*. If the value is zero, the unit is even, if non-zero (i.e., '1'), the unit is odd. The least-significant bit is numbered '0', regardless of the type of unit.
- The left-most bit is the *most-significant bit*. If the unit is treated as a signed quantity, a zero-value of the most-significant bit means it is positive, whereas a non-zero value means it is negative. The most-significant bit is numbered 'N-1' where 'N' is the number of bits in the unit; for a byte, the most-significant bit is bit '7', while for a word, it is bit '15'.
- The symbol '#' (hash) denotes a hexadecimal number, while '$' (dollar) denotes a signed or unsigned integer value.
- To distinguish between assignments and checks for equality, we use '←' for assignment and '=' for equality.

2 Central Processing Unit

XM-23's Central Processing Unit (or CPU) is responsible for fetching, decoding, and executing instructions from memory via the system bus. It consists of the following components (see Figure 2):

- Three memory access registers: a bi-directional, 16-bit Memory Data Register (MDR)[1] for reading instructions and data from, and writing data to, memory; a unidirectional 16-bit Memory Address Register (MAR) that carries an address specifying a memory location to be read or written; and a Control register indicating whether the action is a read or a write. All three registers use the system bus connecting the CPU to memory.
- The 16-bit CPU Data bus carrying instructions from the MDR to the Instruction Register, data from the MDR to a register in the register file or to the MDR from a register in the register file, or the output of the ALU to a register in the register file.
- The 16-bit CPU Address bus carrying an address from a register in the register file or output from the ALU to the MAR.
- The Instruction Register (IR) which takes a 16-bit value, assumed to be an instruction, fetched from memory, for decoding by the Instruction decoder. The IR is inaccessible to the programmer.
- The Instruction Decoder responsible for taking the value stored in the IR and decoding it into its operand and any operands associated with it. If the CPU is in the conditional-execution state, instructions can be fetched but not decoded, allowing XM-23 to operate as a pipelined processor supporting conditional branching. If the value is not a valid instruction, an illegal-instruction fault occurs (see section 11.4).
- The Register File containing XM-23's eight program-accessible registers: R0-R4 (General Purpose), R5 (Link Register or LR), R6 (Stack Pointer or SP), and R7 (Program Counter or PC). Each register is 16-bits wide and can be accessed as a 16-bit word or as an 8-bit high- or low-byte (see Chapter 4). The register file also contains a temporary register for interim values and a numeric constant for use in register arithmetic or logical operations (see section 1.1). The registers and constant-values are connected to the Arithmetic and Logic Unit by a pair of 16-bit busses, as either a source value (S-bus) or destination value (D-bus). A register can be assigned a value from the CPU Data bus (either from memory or the result of an ALU operation, both via the MDR). A register value can be used as an address when copied to the CPU Address bus or a data value to be stored in memory using the CPU Data bus. All of XM-23's general-purpose registers can be used for data or addressing, or both.
- An Arithmetic and Logic Unit (ALU) which performs arithmetic and logic operations using the contents of one or more registers. The register values are supplied from the register file over the S- and D-bus. The operation performed by the ALU is specified by the Control Unit as an ALU function code, determined from the opcode extracted from the instruction by the Instruction decoder. The result is made available to either the CPU Data bus or the CPU Address bus, depending on the actions specified by the opcode. The result can also change one or more of the CPU's status bits (see Chapter 5).

[1] In some machine, the Memory Data Register is referred to as the Memory Buffer Register or MBR.

- The Control Unit orchestrates the entire operation of the CPU, signaling each component when it is to perform its designated task as part of each instruction cycle, consisting of three phases:

  Fetch: In the fetch phase, the Control Unit signals the Register File to put the program counter (PC) onto the CPU Address bus for latching in the MAR. The Control Unit then sets the Control register to indicate a memory read is to be performed and signals the MAR to write the address to the system bus. While the memory read is taking place, the Control Unit increments the PC by 2 (using the ALU).

  Memory copies the value in the specified memory location onto the system bus; it is latched into the MDR. The Control Unit then signals the MDR to put the value onto the CPU Data bus and signals the IR to copy the value from the Data bus.

  Decode: In the decode phase, the Control Unit signals that Instruction decoder to read the IR and then decode the instruction by extracting the opcode (or operation code), any instruction modifiers, and the operand(s). After a set time interval, the decoder makes the extracted values available to the Control Unit.

  Execution: In the execution phase, the Control Unit performs the steps to complete each instruction. The number of steps required varies by instruction. Since all arithmetic and logic operations involve registers or built-in constants, they take place within the CPU and are therefore the fastest. Load and store operations access memory are the slowest as they require the CPU to wait for the memory access (either read or write) to complete. Each of XM-23's instructions are described in Chapter 6 .

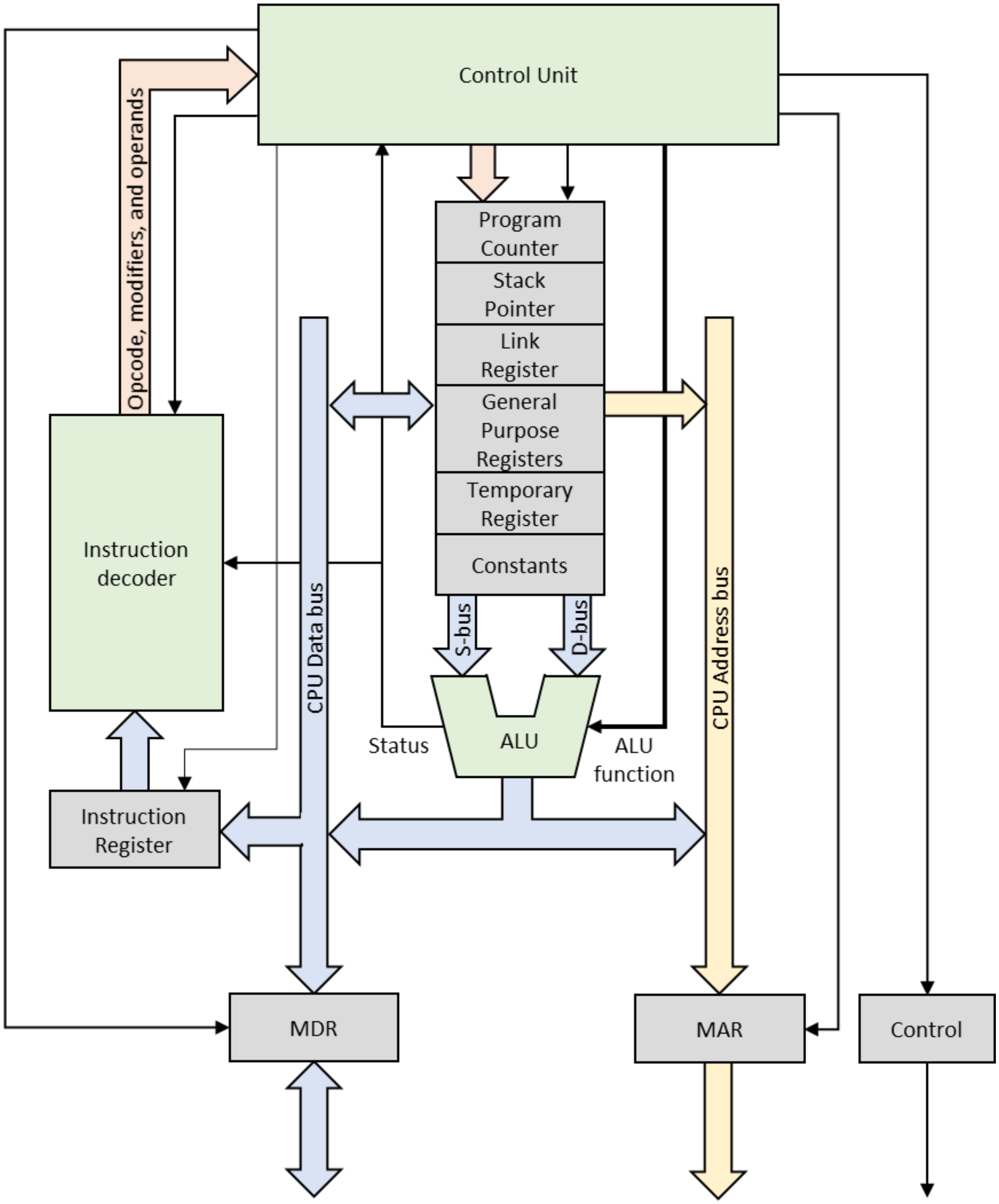

Figure 2: XM-23's Central Processing Unit (CPU)

3 Memory

Executable applications (i.e., code and data), device registers, and exception vectors are all stored in XM-23's memory.

Memory can be thought of as an array of bytes or words (two-byte). In the description of XM-23, the lowest address (i.e., 0) is shown at the top of the page and the highest address (i.e., n-1, where 'n' is the size of memory in bytes or words) at the bottom. There are two reasons for this format:

- When discussing programs, the program flow is from the top to the bottom of the page.
- A stack "grows" upwards from high memory to low memory, with each push putting a value onto the stack, moving it upwards (to a lower address), and each pull (or pop) removing a value from the stack, moving it downwards (to a higher address).

XM-23 supports 65,536 (2^{16}) bytes or 32,768 (2^{15}) words of memory.

3.1 Byte organization

A byte is 8-bits long (a **signed** or **unsigned char**). When accessing data as bytes, the address range is #0000 through #FFFF; bytes can fall on odd or even addresses (see Figure 3).

	7	6	5	4	3	2	1	0
#0000	Byte							
#0001	Byte							
#0002	Byte							
…	…							
#FFFD	Byte							
#FFFE	Byte							
#FFFF	Byte							

Figure 3: Byte memory-organization

3.2 Word organization

A word is a 16-bit quantity in XM-23 (equivalent to a **signed** or **unsigned short**), spanning two bytes. Words must fall (that is, start) on even-byte boundaries. A word with address #0001 refers to bytes #0002 and #0003, while a word with address #7FFE refers to bytes #FFFC and #FFFD. The starting byte of any word is simply the word address shifted left by 1. Instructions and 16-bit integers are stored as words, as shown in Figure 4.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
#0000	Word															
#0001	Word															
…	…															
#7FFE	Word															
#7FFF	Word															

Figure 4: Word memory-organization

Larger, multiple-word and compound structures are not supported by the ISA but can be implemented in software.

3.3 Byte-ordering

A word is divided into two bytes, a high-order, or most-significant, byte (bits 15 through 8) and a low-order, or least-significant, byte (bits 7 through 0). When a word is stored in memory, its high-order (Most Significant Byte or MSB) and low-order (Least Significant Byte or LSB) bytes are stored in consecutive bytes; however, the ISA can store the bytes as either LSB-then-MSB (little-endian) or MSB-then-LSB (big-endian). An example of how the 16-bit quantity #1234 (LSB: 34 and MSB: 12) would be stored as little-endian (least-significant byte first) or big-endian (most-significant byte first) byte-ordering is shown in Figure 5.

nnnn	#12	MSB
nnnn+1	#34	LSB
	Big-endian	

nnnn	#34	LSB
nnnn+1	#12	MSB
	Little-endian	

Figure 5: Storing #1234 in two different endian structures

XM-23 is a little-endian ISA, with the least significant byte in a word in the low-order byte and the most significant byte in the high-order byte (see Figure 6).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
High-order byte (Most significant)								Low-order byte (Least significant)							

Figure 6: Little-endian byte-ordering

Figure 7 shows how #1234 would appear in a little-endian 16-bit word.

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

Figure 7: Little-endian representation of #1234 in a 16-bit word

Since the XM-23 is a little-endian ISA, the least-significant byte of a word is stored in location nnnn (an even address), while the most-significant byte is stored in location nnnn+1 (an odd address). Figure 8 shows memory organized in terms of bytes.

Byte	7	6	5	4	3	2	1	0
#0000	Byte							
#0001	Byte							
#0002	Byte							
…	…							
#FFFD	Byte							
#FFFE	Byte							
#FFFF	Byte							

Figure 8: Byte organization of memory (Address in parenthesis)

Figure 9 shows XM-23's memory organized as words. Words are 16-bits long, consisting of a low-byte (an even-byte address) and a high-byte (odd-byte address). Referring to word #0001 accesses the contiguous bytes #0002 and #0003.

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
#0000	High byte (#0001)								Low byte (#0000)							
#0001	High byte (#0003)								Low byte (#0002)							
…	…															
#7FFE	High byte (#FFFD)								Low byte (#FFFC)							
#7FFF	High byte (#FFFF)								Low byte (#FFFE)							

Figure 9: Word organization of memory (Address in parenthesis)

Unless otherwise indicated, all memory references will be expressed in terms of bytes rather than words.

3.4 XM-23's reserved memory

In addition to memory for instructions and data, XM-23 has two reserved memory areas: device-mapped memory and interrupt vectors. Care should be taken when writing to either area as it could cause unexpected events that are difficult to track down.

3.4.1 Device-register memory

XM-23 supports up to eight devices. Each device is associated with a one-byte control/status register (low-byte) and a one-byte data register (high byte) in the first eight words of memory (see Figure 10). Device registers occupy one word each and their registers are stored contiguously, starting in location #0000.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
#0000	Device 0 Data								Device 0 Control/Status							
#0002	Device 1 Data								Device 1 Control/Status							
…	…															
#000C	Device 6 Data								Device 6 Control/Status							
#000E	Device 7 Data								Device 7 Control/Status							

Figure 10: XM-23's device-register memory (#0000-#000F)

Details on XM-23's devices are given in Chapter 10 .

3.4.2 Interrupt vectors

Memory locations #FFC0 through #FFFA are XM-23's 15 interrupt vectors. The vectors hold the address of exception handlers responsible for dealing with exceptions: device interrupts, system-faults, and application traps. The vectors occupy two words each, the low-order word is the program status word (PSW) to be used when the handler is invoked and the high-order word is the entry-point (i.e., address) of the handler.

The first eight vectors for interrupt-enabled devices while the remaining seven for system-faults and traps. The vectors and their use are described in Chapter 11 . The final two words of high memory (#FFFC and #FFFE) hold the system's restart Program Status Word and the address of the restart code, respectively; see Chapter 12 . This memory is CPU-resident to permit fast access to the vectors and XM-23's PSW by the CPU. The map of reserved memory is shown in Figure 11.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
#FFC0	Vector 0 - Program Status Word															
#FFC2	Vector 0 - Entry point for handler 0															
#FFC4	Vector 1 - Program Status Word															
#FFC6	Vector 1 - Entry point for handler 1															
…	…															
#FFF8	Vector 14 - Program Status Word															
#FFFA	Vector 14 – Entry point for handler 14															
#FFFC	Vector 15 – Reset Program Status Word															
#FFFE	Entry point for system restart															

Figure 11: XM-23's interrupt-vector memory (#FFC0-#FFFE)

4 CPU Registers

XM-23 has eight 16-bit programmer-accessible registers, their names and functions are shown in Table 1. The registers are stored in the register file.

Table 1: Register names and functions (alternate register names in parenthesis)

Name	Function
R0, R1, R2, R3	General Purpose registers
R4 (BP)	Base Pointer or General-Purpose register
R5 (LR)	Link Register or General-Purpose register
R6 (SP)	Stack Pointer
R7 (PC)	Program Counter

4.1 General Purpose registers (R0-R3)

Five 16-bit registers that can be used for addressing or arithmetic and logic instructions. A register can hold signed or unsigned quantities. The sign-bit is context driven, in 8-bit arithmetic it is bit 7, whereas in 16-bit arithmetic it is bit 15.

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

4.2 Base Pointer (R4 or BP)

Register 4 is a general-purpose register that can, by convention, be used as a subroutine's Base Pointer (BP).[2] The Base Pointer is intended to hold the address of the current stack frame during a subroutine call. Relative addressing is used to access the calling subroutine's parameters and the running subroutine's automatic variables (see section 9.2). That said, XM-23 allows any general-purpose register to be used as the BP.

4.3 Link register (R5 or LR)

The Link register (LR) can hold one of:

- Subroutine calls are made with the BL instruction (Branch with Link). The return address is stored in the Link register. The least-significant bit (bit 0) is always clear (i.e., 0); if not, an invalid address fault will occur (see section 11.4).

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

- When an interrupt occurs, the CPU stores an invalid address (#FFFF) in the Link register to indicate that an exception (i.e., an interrupt, fault, or trap) handler is active. This is used by the CPU when returning from the interrupt (see Chapter 11).

[2] The Base Pointer is sometimes referred to as the "Back Pointer".

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

If necessary, Link register can also be used as a general-purpose register (R5). When doing this, it is strongly advised to push its value onto the stack before using it as a general-purpose register and then pulling when its previous value is needed (e.g., prior to leaving a subroutine). Similarly, if inside a subroutine and another subroutine is to be called, Link register should be pushed onto the stack. On return, the value of Link register should be pulled to restore the original value.

4.4 Stack Pointer (R6 or SP)

16-bit register, points to the value on the current top-of-stack. A pull reads this value and increments the Stack Pointer (SP), while a push decrements the Stack Pointer and then writes a value, making it the new top-of-stack. The Stack Pointer should always refer to an even-address instruction (i.e., the least-significant bit should be clear).

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

Using the Stack Pointer as a general-purpose register or setting the least-significant bit can lead to unpredictable results.

4.5 Program counter (R7 or PC)

The program counter, PC, contains the address of the next instruction to be executed. Instructions must fall on even-byte boundaries. As with the stack pointer, an odd-byte address will result in a fault.

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

Moving a value to the Program Counter is equivalent to a JUMP instruction,[3] control will pass to the specified address.

[3] XM-23 does not have a JUMP instruction; however, it can be emulated using MOV or SWAP with a register and the PC; see Chapter 7 . Branching, or transfer of control, is explained in section 6.4.

5 Machine state (Program Status Word)

Each time XM-23 executes an instruction, its state changes; for example, arithmetic instructions can change a register's value, loading and storing can change a register's value or a memory location, and all instructions change the program counter.

The results of arithmetic and logical instructions, such as a negative or zero result, can also change XM-23's state. The Program Status Word (PSW) is a two-byte internal CPU register (i.e., it cannot be directly accessed by software) containing the status bits (Carry, Zero, Negative, and oVerflow), indicating the status changes of the last instruction executed. It also holds the current and previous priority level (0 through 7) of the CPU (see Chapter 11) and an indication if the CPU is in a low-power sleep-state. Its layout is shown in Figure 12.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Previous priority			-	-	-	-	FLT	Current priority			V	SLP	N	Z	C

Figure 12: Program Status Word layout

The PSW status bits are defined as follows:

C: Carry. From experience we know that adding two bits will produce a carry, either 0 or 1. The carry bit is clear if either or both bits is clear and set if both bits are set. In the CPU, the Carry bit represents the value of the carry from bit N-1 of a low order 16-bit word or an 8-bit byte structure into bit 0 of a high order 16-bit word or an 8-bit byte structure. The Carry bit indicates a carry has occurred in either addition, or subtraction or compare; see sections 1.1 and 8.3. Carry has no meaning in signed arithmetic (see 'V', oVerflow, below).

Z: Zero. Indicates whether the last operation resulted in a zero value: #0000 (word) or #00 (byte). If the result is zero the bit is set (Z=1), if non-zero, the bit is cleared (Z=0).

N: Negative. The sign bit is the most significant bit in a word (bit 15) or byte (bit 7) is used to indicate whether the structure is considered as positive (0) or negative (1) in signed arithmetic operations (add, subtract, or compare). The negative bit is set if the most significant bit of the result of the arithmetic operation is set (bit 15 if word operation or bit 7 if byte operation). However, if any instruction sets the most significant bit, the N-bit can be changed (i.e., set or cleared).

V: oVerflow. Overflow can occur in a signed addition, subtraction, or compare using either bytes (8-bit) or words (16-bit). The overflow condition, V, indicates that the result of an addition of two positive or two negative numbers produces a change in sign (i.e., positive + positive gives a negative result or negative + negative gives a positive result).[4] Overflow is functionally equivalent to a carry (i.e., the result exceeds the available bits to represent the result in the structure).

[4] Overflow can occur in subtraction or comparison since these instructions are implemented in the CPU using two's complement addition (see section 1.1 and Chapter 8).

If the condition is not met, the PSW bit in question is cleared (given a zero value). For additional information on the status bits, see sections 6.5 and 8.1.

The remaining other bits are:

SLP: Sleep-state. Indicates whether the CPU is in the sleep (SLP=1) or execution state (SLP=0).

Current priority: The priority of the currently executing application.

FLT: Fault state. Indicates whether the CPU has experienced a system fault (FLT=1) or not (FLT=0).

Previous priority: The priority of the previously executing application (i.e., the one in which an exception occurred). The previous priority value is always less than the current priority value.

To avoid potential deadlock, XM-23 enforces the following with respect to the sleep-state bit (**SLP**):

- When entering or leaving an interrupt service routine, PSW.SLP is cleared.
- When the CPU priority is 7, PSW.SLP cannot be set.

When an exception occurs, the PSW is pushed onto the stack by the CPU (along with the PC, LR, and CEX state). Before the current priority is updated with the exception priority, the value of the PSW's current priority is copied to the previous priority. At the completion of the exception, the CPU determines which application to resume using the value of the previous priority. See Chapter 11 (Exceptions) for details.

6 Instructions

All machines have instructions identified by unique opcodes which the CPU decodes and extracts the operands and modifiers required to execute the instruction.

In this section, XM-23's instructions are described:

- Register initialization instructions
- Memory accessing instructions
- Arithmetic and logic instructions
- Register exchange instructions
- Single-register instructions
- Transfer of control instructions
- PSW and systems instructions

1.1 Register initialization instructions

In addition to the memory-accessing instructions, registers can be assigned initial 8-bit values using four different move instructions: move-low and leave the high-byte unchanged (MOVL), move-low and zero all bits in the high-byte (MOVLZ), move-low and set all bits in the high-byte (MOVLS), and move-high and leave the low-byte unchanged (MOVH). All instructions have the same format, shown in Figure 13.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">Opcode</td><td colspan="8">8-bit value</td><td colspan="3">DST</td></tr>
</table>

Figure 13: Register initialization format

The byte to be stored in the destination register (DST) is stored in bits 3 through 10 of the instruction. The operation, description, and opcode of each instruction is listed in Table 2. There is no source register in these instructions.

Table 2: Register initialization instructions
(LSB – least-significant byte; MSB – most-significant byte)[5]

Instruction	Operation	Description
MOVL Value,DST	*DST.LSB← Value* *DST.MSB unchanged*	Value is assigned to the low-byte of the destination register. The high-byte is unchanged.
MOVLZ Value,DST	*DST.LSB ← Value* *DST.MSB ←#00*	Value is assigned to the low-byte (LSB) of the destination register. The high-byte (MSB) is zeroed.
MOVLS Value,DST	*DST.LSB ← Value* *DST.MSB ← #FF*	Value is assigned to the low-byte (LSB) of the destination register. The high-byte (MSB) is assigned #FF.
MOVH Value,DST	*DST.LSB unchanged* *DST.MSB ← Value*	Value is assigned to the high-byte of the destination register. The low-byte is unchanged.

The destination register can be any register (R0 through R7) or an equated value, such as:

```
BP   equ  R4   ; ’BP’ can be used in place of R4 (base pointer)
LR   equ  R5   ; ’LR’ can be used in place of R5 (link register)
SP   equ  R6   ; ’SP’ can be used in place of R6 (stack pointer)
PC   equ  R7   ; ’PC’ can be used in place of R7 (program counter)
```

The value can be any eight-bit quantity recognized by the assembler, notably:

Decimal values: Signed values from $-128 through $0 to $127 and unsigned values from $0 to $255.

Hexadecimal values: Any hexadecimal value between #0 (or #00) to #FF.

Characters: Any ASCII character enclosed in single quotations, such as ’A’, ’\t’ (tab), ’\’’ (single quote), and ’1’ (character, not number, ’1’).

For example, they can be named constants such as:

```
CapA     equ ’A’  ; Equated character constant ’A’
Limit    equ $255; Equated decimal constant $255
NUL      equ #0   ; Equated hex constant #0
;
    org     #800
; To assign R0 the value #2010:
    movl    #10,R0
    movh    #20,R0
;
; To assign CapA to the lower byte of R3:
    movl    CapA,R3
```

A register can be zeroed (equivalent to a clear) using a single MOVLZ instruction. For example, to zero register R3:

```
    MOVLZ   $0,R3
```

[5] Value can be a numeric value or a label. For rules regarding numeric values and labels, see the XM-23 Assembler User’s Guide.

Upon completion of the instruction, R3 has the value #0000. The previous value is lost.

To assign the value -1 (#FFFF) to a register can be done in one instruction:

```
    MOVLS    #FF,R4
```

The least-significant byte of R4 is explicitly assigned the value #FF, while the instruction sets the bits in the most-significant byte.

Initializing a register to a specific address requires two steps, assigning the low-byte of the address using MOVL and the high-byte of the address, using MOVH. For example, to assign the 16-bit address of Array to R0:

```
    ORG  #1000
; Data
Array    bss  $16      ; Reserve 16 bytes
; Code
    ORG      #2000
    MOVL     Array,R0 ; Least-significant byte of Array assigned to R0
    MOVH     Array,R0 ; Most-significant byte of Array assigned to R0
```

The XM-23 assembler extracts the lower 8-bits of Value for MOVL, MOVLZ, and MOVLS and the higher 8-bits for MOVH.

There is a trick that can be used requiring only one move instruction when initializing a register. If the data is stored in the first 256 bytes of memory (#0000 through #00FF) only require one instruction, MOVLZ, to initialize a register to its address, since the high byte of the register is explicitly cleared:

```
    ORG  #80
;
Array    BSS  $10      ; 10 bytes reserved; Address is #0080
;
; R0.Low-byte = Low-byte of Address (#80)
; R0.High-byte = #00
;
     MOVLZ   Array,R0 ; R0 = #0080
```

6.1 Memory access

Memory access instructions allow a program to access data memory for loading the contents of a memory location into a register (i.e., “reading” the memory location) or storing the contents of a register into a memory location (i.e., “writing” to a memory location). The address of the location being accessed is referred to as the *effective-address* (EA).

In some machines, most instructions can access a memory location. In other machines a limited number of memory-accessing instructions are used to access memory, these are referred to as *load-store computers*. XM-23 is a load-store computer.

XM-23 has four memory accessing instructions, two for loading data from a memory location into a register and two for storing the contents of a register in a memory location. The instructions allow word (16-bit) or byte (8-bit) access.

In these instructions, two registers are used. One of the registers and, in some cases, a modifier, determine the effective address. The contents of the memory location specified by the effective address can be loaded into the second register) or the contents of the second register can be stored in the specified memory location.

The load-store instructions allow direct addressing, indexed addressing, and relative addressing.

6.1.1 Direct addressing

In direct addressing, the register used to determine the effective address is not modified by the instruction, it specifies the memory location to be loaded or stored. Unless the address is modified by another instruction, the same location is accessed each time the instruction is executed.

The format of the LD (load) and ST (store) instructions for direct addressing is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode						0	0	0	W/B	SRC			DST		

The instruction fields are interpreted as follows:

W/B: Indicates whether a word (0) or a byte (1) is to be accessed. Word-access always falls on an even memory address and accesses 16-bits, whereas byte-access can refer to an odd or even (byte) address. When loading a register with a byte or storing a byte from a register, only the low-order byte of the register is used; the high-order byte is unchanged.

SRC: The source register, SRC, indicates where the data is coming from, either a memory location (in load) or a register (in store).

When loading, the source register value is the effective address (i.e., it refers to the memory location from which the data is being loaded from). When storing, the source register contains the value to be written to the memory location.

DST: The destination register, DST, indicates where the data is going to, either a memory location (in store) or a register (in load).

When loading, the destination register is assigned the value loaded from memory; however, when storing, the destination register value is the effective address of the memory location where the data is being stored.

Table 3 lists the direct memory addressing instructions, their operation, and a description.

Table 3: Direct memory addressing

Instruction	Operation	Description
LD(.B or .W) SRC,DST	*EA ← SRC* *DST ← memory [EA]*	Load a register (DST) from memory location specified by the effective address (EA), the value in the SRC register. Reading a byte stores the value in the low-byte of the DST register; the high-byte is unchanged.
ST(.B or .W) SRC,DST	*EA ← DST* *memory [EA] ← SRC*	Store a register (SRC) in memory location specified by the effective address. Writing a byte to the low-byte of a word does not change the word's high-byte.

A single word or byte memory location can be accessing using the LD or ST instructions. For example, to copy Data1 to Data2, we could write:

```
     org  #1000
;
```

```
Data1    word #1234     ; Data1 occupies a word with the value
Data2    bss  $2   ; Data2 occupies two bytes (word) and is uninitialized
;
    org       #2000
Start
;
; Get address of Data1 into Register 0
; This requires two steps:
; - First put the least significant byte of the address in R0
; - Then put most-significant byte of the address in R0
;
    movl      Data1,R0  ; Low-byte
    movh      Data1,R0  ; High-byte
; Get address of Data2 into Register 1
    movl      Data2,R1  ; Low-byte
    movh      Data2,R1  ; High-byte
; Load Data1 (address R0) into temporary register (R2)
    ld        R0,R2
; Store R2 into Data2 (address R1)
    st        R2,R1
;
    end       Start
```

When execution finishes, Data1 should be unchanged and Data2 should have the value of Data1 or #1234.

6.1.2 Indexed addressing

Direct addressing is limited in that unless the register used to determine the effective address is modified, the instruction will access the same location each time it is executed. A variation on direct addressing is to modify the register within the load or store instruction itself, allowing access to, for example, array structures. This is referred to as *indexed addressing* with the effective address being obtained from a register using as an *index register*.

XM-23, like most other machines that use indexed addressing allows the programmer to increment or decrement the index register before or after accessing a memory location. This is referred to as *pre-increment*, *pre-decrement*, *post-increment*, and *post-decrement*.

The format of the LD (load) and ST (store) instructions using indexed addressing is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode						PRPO	DEC	INC	W/B	SRC			DST		

The register used for memory-access, the index register, can be modified using combinations of the following bits:

PRPO: Pre- or post- increment or decrement of the register specifying the memory location to access. A clear value (i.e., zero) can indicate either a post-increment or post-decrement or no action, while a set value (i.e., one) indicates a pre-increment or pre-decrement action.

DEC: Decrement the register (before or after the instruction is executed, see PRPO). A clear value indicates no decrementing, while a set value indicates decrementing.

INC: Increment the register (before or after the instruction is executed, PRPO). A clear value indicates no incrementing, while a set value indicates incrementing.

SRC: The source-register (R0 through R7). The SRC register is where the data comes “from”.

The SRC register in the load instruction (LD) refers to the memory location from which a data value is read and loaded-into the DST register (see below). The SRC register can be modified (pre/post and decrement/increment) to move through memory a byte or a word at a time.

The SRC register in the store instruction (ST) contains the value of the data to be written to memory (the address of which is specified by the DST register, see below).

DST: The destination-register (R0 through R7). The DST register is where the data goes "to".

The DST register in the load instruction (LD) is the register to which the data value from memory is to be written (the address of which is specified by the SRC register, see above).

The DST register in the store instruction (ST) refers to the memory location to which a data value is written-to from the SRC register. The DST register can be modified (pre/post and decrement/increment) to move through memory a byte or a word at a time.

The possible address modifier combinations are listed in Table 4. A modified register used to indicate an address in a byte load or store will increment or decrement by 1, while a register referring to a word will increment or decrement by 2.

Table 4: Valid PRPO, DEC, and INC combinations and their meanings
(PRPO, DEC, and INC combinations 011, 100, and 111 are undefined)

Register format	Definition	Effective address (EA) and register value	PRPO	DEC	INC
Rn	Unmodified register (This is direct addressing.)	*EA = Rn* *Access memory [EA]*	0	0	0
+Rn	Pre-increment register	*EA = Rn + 1 (byte) or + 2 (word)* *Access memory [EA]* *Rn = EA*	1	0	1
Rn+	Post-increment register	*EA = Rn* *Access memory [EA]* *Rn = Rn + 1 (byte) or + 2 (word)*	0	0	1
-Rn	Pre-decrement the register	*EA = Rn - 1 (byte) or - 2 (word)* *Access memory [EA]* *Rn = EA*	1	1	0
Rn-	Post-decrement the register	*EA = Rn* *Access memory [EA]* *Rn = Rn - 1 (byte) or - 2 (word)*	0	1	0

The uses of the source (SRC) and destination (DST) registers depend on the instruction (LD or ST) and are explained in Table 5.

Table 5: Load and store register-direct and register-direct with pre- or post-auto-increment or auto-decrement

Instruction	Operation	Description
LD(.B or .W) +SRC,DST LD(.B or .W) -SRC,DST LD(.B or .W) SRC+,DST LD(.B or .W) SRC-,DST	*if Pre-Incr or Pre-Decr then* *SRC = SRC + address modifiers* *EA = SRC* *DST ← memory[EA]* *if Post-Incr or Post-Decr then* *SRC = SRC + address modifiers*	Load a register (DST) from memory location specified by the effective address (EA), the value in the SRC register and the address-modifier bits. Loading a byte stores the value in the low-byte of the DST register; the high-byte is unchanged.
ST(.B or .W) SRC,+DST ST(.B or .W) SRC,-DST ST(.B or .W) SRC,DST+ ST(.B or .W) SRC,DST-	*if Pre-Incr or Pre-Decr then* *DST = DST + address modifiers* *EA = DST* *memory [EA] ← SRC* *if Post-Incr or Post-Decr then* *DST = DST + address modifiers*	Store a register (SRC) in memory location specified by the effective address. The effective address is obtained from the DST value and the address modifier bits. Writing a byte to the low-byte of a word does not change the word's high-byte.

To illustrate how Indexed addressing words, the program in the following example stores two bytes, #F0 and #0F, into locations #1000 and #1001, respectively. The value of location #1000 is #0FF0.

```
    org     #1000
Data1       bss  #2      ; Reserve two bytes
;
    org     #2000
;
; R0 <- address of Data1. This is the byte address of Data1.
    movl    Data1,R0
    movh    Data1,R0
;
; Store #F0 in memory[R0] (1000) and increment R0 by 1 (byte)
    movlz   #F0,R1
    st.b    R1,R0+
;
; Store #0F in memory[R0] (1001) using direct addressing
    movlz   #0F,R1
    st.b    R1,R0
;
    ld.w    R0,R2   ; R2 <- memory[R0] as a word
```

When the last instruction is executed, register 2, which loads R2 with the word referenced by the contents of R0 (#1001). When loading a word, the register value, which refers to the byte address of the data, is shifted left by 1 to give the word address (#0800 in this case). Since word #0800 maps into bytes #1000 and #1001, the value loaded into register 2 is, correctly, #0FF0.

The LD and ST instructions also permit array accessing (e.g., 8-bit and 16-bit arrays) using register increment and decrement (if required; for example, to reverse a string).

For example, to copy 10 words from Array1 to Array2:

```
AR1  BSS     $20     ; 10 words (20 bytes) of storage
AR2  BSS     $20
```

```
;
    MOVL     AR1,R2   ; Ptr1 = &AR1 (low and high byte)
    MOVH     AR1,R2   ; R2 has address of AR1
;
    MOVL     AR2,R3   ; Ptr2 = &AR2 (low and high byte)
    MOVH     AR2,R3   ; R3 has address of AR22
;
    MOVLZ    $10,R0   ; Counter (R0) = 10
Loop
    LD       R2+,R1   ; Data (R1) = *Ptr1++ (mem[R2]; R2=R2+2)
    ST       R1,R3+   ; *Ptr2++ = Data
    SUB      $1,R0        ; Counter (R0) = Counter (R0) - 1
;
; If Counter != 0 Then repeat from Loop
;
    BNZ      Loop     ; Execute next instruction if R0 not zero
```

R2 and R3 are incremented by 2 since LD and ST are operating on words.

The number of bytes in a NUL-terminated string can be counted using LD and CMP:[6]

```
NUL EQU      #0
Str BSS      $80      ; Space for 80 character
;
    MOVL     Str,R1   ; char *ptr = String
    MOVH     Str,R1   ; R1 has address of String
    MOVLZ    $0,R0    ; count = 0
Loop
    LD.B     R1+,R2   ; data = *ptr++
    CMP.B    NUL,R2   ; if data = NUL, leave loop
;
; Continue until R1 is NUL    (i.e., equals zero)
;
    BZ       Done     ; Execute next 2 instruction if not zero
    ADD      $1,R0    ; count++
    BRA      Loop     ; Repeat for next character
;
; R2 is NUL, R0 (count) has length of string
;
Done
```

In the above example, R1 is incremented by 1 because LD.B operates on bytes.

The load and store instructions with pre- and post- increment and decrement can also be used to create stacks and stack operators. For example, LD and ST can be used to implement stack instructions PUSH and PULL (POP).

[6] Note: An array of characters can be considered a string.

```
SP       EQU  R6       ; SP equated stack pointer
STKTOP   EQU  #FFC0    ; Start of stack if FFC0
;    ...
; Stack is pointed to by SP
;
    MOVL     STKTOP,SP
    MOVH     STKTOP,SP
;
; Push R2 onto stack, allowing R2 to be used
;
    ST       R2,-SP   ; Push R2 (save R2)
    MOVLZ    $0,R2    ; Clear R2
;
; Other instructions involving R2
;

    LD       SP+,R2   ; Pull R2 (restore R2)
```

The stack pointer always point to the top-of-stack which holds the most recently pushed value. In the above example, the stack pointer is decremented before the PUSH to point to the next available word in memory (using ST and –SP), a pre-decrement. To PULL, memory referred to by the address held in the stack point is read and then the stack pointer is incremented (using LD and SP+), a post-increment.

6.1.3 Relative addressing

In *relative addressing*, the effective address is obtained by adding a base address to an offset; that is, the location being accessed is *relative* to a base address. Relative addressing is useful when accessing C-like structs and stack frames.

XM-23 supports relative addressing using two load and store instructions, LDR and STR. In relative addressing, the address accessed is relative to a base address specified by a register. In this case, the effective address is determined from the register value (the base address) plus the value of a signed 7-bit offset encoded in the instruction:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode		7-bit offset							W/B	SRC			DST		

The 7-bit signed offset (bits 7 through 13) values are extracted and stored in bits 0 through 6 of an internal 16-bit register with the value of bit 6 being duplicated in bits 7 through 15 (i.e., the sign-bit is extended, this is referred to as *sign extension*).[7]

The signed value in the internal register is added to the SRC or DST register (SRC for load or DST for store) to become the effective address. Since the sign extended value is in two's complement format, adding a negative value allows access to locations with addresses less-than the base address. The offset range is the value of the SRC-64 for the LDR register or DST-64 for the STR register through the value of the SRC+63 for the LDR register or DST+63 for the STR register.

Up to 128 bytes or 64 words can be addressed. A word or a byte can be accessed by specifying 0 (word) or 1 (byte) in the W/B bit (see Table 6):

[7] The register used to determine the sign extension is not accessible by software. It is internal to the CPU. None of the PSW bits are affected.

Word offsets: Word offsets refer to words (on even byte addresses) with a word address range relative to the base address of -32 through 0 to +31. This is equivalent to the base address + the offset ANDed with 0xFFFE.

Byte offsets: Byte offsets refer to bytes on odd or even byte-boundaries and have a byte address range of -64 through 0 to +63 relative to the base address.

Table 6: Relative addressing using word or byte offsets
Range of addressable memory: Base address - 64 through Base address + 63

Offset values (bits 13..7)	Sign-extended value (original 7-bits in red)	Offset	Effective address	
			Byte	Word
100.0000	1111.1111.1100.0000	-64	Base_addr - 64	(Base_addr - 64) & 0xFFFE
100.0001	1111.1111.1100.0001	-63	Base_addr - 63	(Base_addr - 63) & 0xFFFE
		. . .		
111.1110	1111.1111.1111.1110	-2	Base_addr - 2	(Base_addr - 2) & 0xFFFE
111.1111	1111.1111.1111.1111	-1	Base_addr - 1	(Base_addr - 1) & 0xFFFE
000.0000	0000.0000.0000.0000	+0	Base_addr + 0	(Base_addr + 0) & 0xFFFE
000.0001	0000.0000.0000.0001	+1	Base_addr + 1	(Base_addr + 1) & 0xFFFE
000.0000	0000.0000.0000.0010	+2	Base_addr + 2	(Base_addr + 2) & 0xFFFE
		. . .		
011.1110	0000.0000.0011.1110	+62	Base_addr + 62	(Base_addr + 62) & 0xFFFE
011.1111	0000.0000.0011.1111	+63	Base_addr + 63	(Base_addr + 63) & 0xFFFE

The byte and word effective addresses are determined by the CPU. The programmer's job is to specify the memory locations being accessed.

The two instructions are defined in Table 7, where SRC and DST refer to the source and destination registers and OFF is the offset with a value between -64 and +63. An offset of zero is equivalent to direct addressing.

Table 7: Load and store register-relative instructions[8]

Instruction	Operation	Description
LDR(.B or .W) SRC,OFF,DST	*EA = SRC + sign-extended offset* *DST ← memory [EA]*	Load a register (DST) from memory location specified by the effective address (EA).
STR(.B or .W) SRC,DST,OFF	*EA = DST + sign-extended offset* *memory [EA] ← SRC*	Store a register (SRC) in memory location specified by the effective address.

The format both instructions is from somewhere to somewhere. In LDR, it is from SRC+OFF to DST, and in STR, it is from SRC to DST+OFF.

The following example shows how a two-byte structure (Data1) can be initialized using relative stores and then accessed using a relative load:

```
    org  #1000
```

[8] OFF (offset) can be a numeric value or a label. For rules regarding numeric values and labels, see the XM-23 Assembler User's Guide.

```
Data1    bss  #2
;
    org  #2000
Start
;
; R0 <- address of Data1
;
    movl     Data1,R0 ; Low-byte of R0 <- #00
    movh     Data1,R0 ; High-byte of R0 <- #10
;
; Write byte #F0 to Data1[0]
;
    movlz    #F0,R1   ; R1 <- #F0
    str.b    R1,R0,#0 ; mem[R0 + #0] <- R1
;
; Write byte #0F to Data1[1]
;
    movlz    #0F,R1   ; R1 <- $0F
    str.b    R1,R0,#1 ; mem[R0 + #1] <- R1
;
; Retrieve word from Data1[0]
;
    ldr.w    R0,#0,R2 ; R2 <- mem[R0 + #0]
```

When the above finishes execution, R2 has the value #0FF0.

1.1 Two-operand (register-register and constant-register) instructions

The register-register and constant-register instructions perform operations on any pair of registers (SRC and DST) or a constant (CON) and a register (DST). With two exceptions, all instructions store the result of the operation in the destination register.

The format of these instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode								R/C	W/B	SRC/CON			DST		

Where Opcode indicates the instruction, W/B whether the operation is on a word (W/B=0) or a byte (W/B=1), and DST, the destination register. The R/C field indicates whether the SRC field is a register (R0 through R7) or an encoded constant. These are XM-23's arithmetic and logic instructions.

In these two-operand instructions, the SRC can be either a register or one of the eight encoded constants listed in Table 8. If R/C (bit 7) has a value of 0, the SRC/CON value (bits 3 through 5) is treated as register (R0 through R7). On the other hand, if R/C has a value of 1, XM-23 uses the value of SRC/CON to indicate one of eight immediate-value constants (0, 1, 2, 4, 8, 16, 32, or -1). For example, if R/C is 1 and SRC/CON is 111 (7), XM-23 uses the constant -1, while an R/C value 0 and a SRC/CON value of 100 (4) causes XM-23 to use the contents of R5 (or LR).

Table 8: Interpretation of R/C bit and SRC/CON bits

R/C		SRC/CON
0	**1**	**Value (bits 3-5)**
Register	**Constant value**	
R0	0	000
R1	1	001
R2	2	010
R3	4	011
R4	8	100
R5/LR	16	101
R6/SP	32	110
R7/PC	-1	111

The constants 0 and -1 are used as they often indicate the end of a quantity (such as a NULL pointer or a NUL-terminated string), while the constants 1, 2, 4, and 8 can be used as an increment through a byte (8-bits, i.e., character), short (2-bytes), integer (4-bytes), and double (8-bytes).

The constants can also be used when accessing bits 0 through 5 in a structure. The value 32 also represents ASCII space and is also the difference between lower- and upper-case ASCII characters.

XM-23's two-operand instructions are listed in Table 9. These instructions can be modified using the R/C bit (to use either a register or a constant) or the W/B bit (to operate on w 16-bit word or the least-significant byte).

Table 9: Two-operand instructions
(SRC can be a register or a constant [see Table 8])

Instruction	Operation	Description
ADD(.B or .W) SRC,DST	*DST ← DST + SRC*	Add SRC to DST
ADDC(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Add SRC and carry to DST
SUB(.B or .W) SRC,DST	*DST ← DST + ~SRC + 1*	Subtract SRC from DST
SUBC(.B or .W) SRC,DST	*DST ← DST + ~SRC + C*	Subtract SRC from DST plus carry
DADD(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Decimal-add SRC and carry to DST
CMP(.B or .W) SRC,DST	*DST + ~SRC + 1*	Compare DST with SRC (subtraction)
XOR(.B or .W) SRC,DST	*DST ← DST ⊕ SRC*	XOR SRC with DST
AND(.B or .W) SRC,DST	*DST ← DST* & *SRC*	AND SRC with DST
OR(.B or .W) SRC,DST	*DST ← DST | SRC*	OR SRC with DST
BIT(.B or .W) SRC,DST	*DST* & (1 << *SRC)*	Test if bit set in SRC is set in DST
BIC(.B or .W) SRC,DST	*DST ← DST* & ~(1 << *SRC)*	Clear bit in DST specified by SRC
BIS(.B or .W) SRC,DST	*DST ← DST* | (1 << *SRC)*	Set bit in DST specified by SRC

Except for CMP and BIT, all instructions have the same underlying format:

DST ← DST <operation> [*SRC* | *CON*]

The instructions can also change the PSW bits V, N, Z, and C.

There are four arithmetic operators, ADD, ADDC, SUB, and SUBC, allowing pairs of 8- or 16-bit numbers to be added or subtracted; for example, to add two numbers:

```
; Add alpha to beta and store in alpha
    org  #100
alpha    word    #10
beta     word    #20
;
    org  #200
; R0 <- address of alpha
    movl     alpha,R0
    movh     alpha,R0
; R1 <- alpha
    ld       R0,R1
; R2 <- beta, the word following alpha, use relative addressing
    ldr      R0,#1,R2
; Add
    add      R1,R2   ; R2 <- R2 + R1
; Store result in alpha
    st       R1,R0
```

Byte operations affect the least-significant byte only; the most-significant byte is untouched. For example, performing byte addition with #FFFF in register R0 and #0002 in register R1:

```
; Using a register:
    MOVLS    $-1,R0    ; R0 = #FFFF
    MOVLZ    $2,R1     ; R1 = #0002
;
    ADD.B    R1,R0     ; #FF + #02 equals #01 with a carry
```

After the ADD.B, R0 has a value of #FF01, and the PSW bit values: C=1, N=0, Z=0, and V=0.

Constants can reduce the number of registers used in an expression. In the following example, R2 is to be incremented by 2:

```
; Using a temporary register:
    MOVLZ    $2,R3   ; R3 is used as a temp register with value of 2
    ADD      R3,R2   ; R2 = R2 + R3 (i.e., incremented by 2)
;
; Using a constant:
    ADD  $2,R2   ; R2 = R2 + 2
```

When using a constant, one less instruction is required, and it is not necessary to use a temporary register (R3 in the above example). However, using a register offers a wider range of values. Constants can be used with bytes or words.

An unsigned 16-bit number has values from #0 ($0) to #FFFF ($65535), while a signed 16-bit number uses the same 16-bits to represent the sign of the quantity (the most significant bit, bit 15) and the remaining bits constitute the number. Consequentially, signed 16-bit values range from #8000 ($-32768) through #FFFF ($-1), #0 ($0), #1 ($1) to #7FFF ($32767).

By itself, the CPU makes no distinction between positive and negative numbers; this is determined by the programmer and the program. However, if an operation results in the most significant bit in the operation (bit-7 in byte arithmetic or bit-15 in word arithmetic) being set, the CPU sets the N (negative) bit in the PSW, otherwise the N-bit is cleared.

When adding, the result can exceed the specified size (byte or word); for example, adding two 16-bit numbers can result in a 17-bit result. When this occurs, the carry bit is set. If multiple-word arithmetic is taking place, the carry-bit can be added to next most-significant byte or word using ADDC (add with carry). Using the carry bit allows multiple-word arithmetic. For example, a 32-bit word can be created from two 16-bit words:

```
    org #80
;
LOword   word     #FFFF    ; Low-order word (lower 16-bits)
HOword   word     #0       ; High-order word (upper 16-bits)
;
    org      #1000
    movlz    LOword,R0     ; R0 <- address of LOword
    ld       R0,R1         ; R1 <- LOword
    ldr      R0,#2,R2      ; R2 <- HOword
;
; Add #2 to LOword (in R1)
;
    add      #2,R1    ; LOword <- LOword + 2
;
; R1 contains #0001 and Carry is set
; Carry must be added to HOword
;
    addc     #0,R2    ; HOword <- HOword + 0 + C (Include carry)
;
; R2 contains #0001 and Carry is clear
; New 32-bit value is HOword: #0001 and LOword: #0001
```

In multiple-word signed arithmetic, the sign bit is in the highest-order word and is ignored in the lower-order words.

The subtract and compare (see below) instructions are performed using two's complement addition to determine the result of DST – SRC: the DST is added to the one's complement of the SRC (obtained by inverting the SRC's bits, denoted as ~SRC) and *then* 1 is added, giving the result:[9]

Result = DST + ~SRC + 1

Because this is two's complement, the result can be treated as either unsigned or signed. If unsigned, the carry bit indicates the result of the addition of the most-significant bits in a low-order structure (word or byte) that is to be added to the complement of the least-significant sit in the high-order structure (word or byte). Carry is used with the SUBC instruction and is described in Chapter 8 .

When subtracting a multiple structure (typically words, although byte subtraction is supported); for example, using four 16-bit words to create a 64-bit quadword, the least-significant word in the subtraction uses SUB, while the remaining words use the SUBC (subtract with carry) instruction. Both SUB and SUBC obtain their result from the equation *DST + ~SRC + X*:

- In SUB, the value of X is 1 to obtain the 2's complement of the low-order structure, which is understandable since this is the first structure in the subtraction (a word in this case).

[9] Although *~SRC + 1* is the two's complement of SRC, by making this a two-step process, it is possible to detected overflow by examining the sign bits of DST and ~SRC. The one's complement of SRC (~SRC) is used rather than the two's complement (~SRC + 1) to handle the most negative number (-2^{N-1}) in the range. For example, the one's complement of #80 is #7F, whereas the two's complement of #80 is (#7F + 1 or #80).

- In SUBC, X is the value Carry bit. If clear, X has a value of 0; in this case, the result of word N is *DST_N + ~SRC_N + 0*. However, if set, X has a value of 1 and the result of word N is *DST_N + ~SRC_N + 1*.
- The sign-bit only has meaning in the most-significant word of the structure. For example, in a signed quadword, bit 15 of the most-significant word (bit 63 of the quadword) is the sign bit and the remaining 63 bits (the 15 bits in the most-significant word and the bits in the three least-significant words) determine its range (-2^{63} through 0 to +2^{63}-1). There is no sign bit in the least-significant three words of the structure.

In signed arithmetic, the Negative bit is the value of the most significant bit of the result: a set value indicates a negative result (N=1), if clear, a non-negative result (N=0). The overflow bit (V) is used in signed arithmetic to indicate whether the result of the subtraction has overwritten the sign bit; the V-bit is set when the DST and ~SRC have the same sign, but the result has the opposite sign (otherwise it is cleared):

DST	**~SRC**	**Result**
Positive	Positive	Negative
Negative	Negative	Positive

Since the CPU has no indication whether an arithmetic instruction is being used for signed or unsigned arithmetic, it changes the V, N, and C bits in the PSW. It is up to the application to determine whether the result should be considered signed or unsigned.

The following are examples of the SUB instruction:

```
    org      #1000
;
; Example 1: R0 <- #1 – 0
; Result:
; R0=#0001
; C=1
; N=0 (non-negative result)
;
    movlz    #1,R0
    sub      #0,R0    ; R0 <- R0 - 0
;
; Example 2: R0 <- #1 - #2
; Result:
; R0=#FFFF ($-1)
; C=0
; N=1 (negative result)
;
    movlz    #1,R0
    sub      #2,R0    ; R0 <- R0 - 2
;
; Example 3: R0 <- #1 – ($-1)
; Result:
; R0=#0002 ($2)
; C=0
; N=1 (non-negative result)
;
    movlz    #1,R0
    sub      $-1,R0   ; R0 <- R0 - (-1)
;
; Example 4: R0 <- #1 - #1
; Result:
; R0=#0000 ($0)
; C=1
; N=0 (negative result)
; Z=1 (zero result)
;
    movlz    #1,R0
    sub      #1,R0    ; R0 <- R0 - 1
```

The CPU supports Binary-Coded Decimal (BCD) addition. BCD can be useful when used with seven-segment displays. A BCD number is stored in a byte with a decimal digit value between #0 ($0) and #9 ($9); bit patterns #A through #F are invalid. BCD numbers stored in 16-bits therefore have values between 00 and 99. Adding BCD numbers is handled by the CPU; for example:

```
; Assume R2 refers to a BCD counter that is incremented periodically
    DADD     #1,R2
```

If R2 has the value #12, adding #1 gives #13. At #19, adding #1 gives #20; the low-order byte carry is handled by the instruction. Adding #1 to #99 results in #00 and the carry bit being set. In the situation, it is necessary to increase the next BCD-digit pair (assuming they exist). Using DADD to add two non-BCD numbers gives unpredictable results; it is up to the programmer to ensure that the numbers are valid BCD values.

Registers can be XORed, ANDed, and ORed (using BIS). For example, to clear the low-order nibble of R1 (bits 0 through 3):

```
; A mask register is required
    MOVLZ    #0F,R2   ; R2 <- #000F
    AND      R2,R1    ; R1 = R1 & #000F
```

The CMP (compare) and BIT (bit test) instructions are used for explicit comparisons and can change the PSW bits. Both instructions are non-destructive in that the DST register is not changed.

In CMP, the DST register subtracts the SRC register or constant value and sets the V, N, Z, and C bits as required; the subtraction is performed using two's complement addition (see above). The PSW bits are set or cleared according to the rules found in Chapter 5 . For example:

```
    org  #1000
;
; R0 <- #0000
    movlz    #0,R0
;
; Example 1: R0 - #0000
; Result is #0000
; Z=1 (zero result)
; C=1
; Values are equal
;
    cmp  #0,R0        ; V=0 N=0 Z=1 C=1
;
; Example 2: R0 - #0001
; Result is #FFFF ($-1)
; Z=0 (non-zero result)
; N=1 (negative result)
; C=0
; SRC (R0) is less than DST (#0001)
;
   cmp  #1,R0        ; V=0 N=1 Z=0 C=0
;
; Example 3: R0 - #FFFF ($-1, signed; $65535, unsigned)
; Result is #0001 ($1)
; Z=0 (non-zero result)
; N=0 (negative result)
; C=0
; SRC (R0) is less than DST (#0001) - unsigned
;
   cmp  $-1,R0   ; V=0 N=0 Z=0 C=0
```

The bit instructions (BIT, bit test; BIC, bit clear; and BIS, bit set) require the source value (register or constant) to have a value that specifies the bit being accessed (0 to 15). The constants 0, 1, 2, 4, and 8 access to bits 0, 1, 2, 4, and 8, respectively. For example, to clear bit 2 in register R0, the source value has the value 2:

```
   BIC  $2,R0
```

BIC (bit clear) and BIS (bit set) clear and set bits in the DST register, respectively. BIC sets the PSW zero (Z) bit if the result is zero, while BIS clears the zero-status bit.

The BIT instruction, unlike CMP, allows specific bits to be compared by ANDing the DST with the SRC/CON. The value in SRC/CON is the *bit mask*. The result of the test can change the zero (Z) and negative (N) bits in the PSW. If the Z bit is set, it means that none of the bits in the SRC or DST registers were congruent; if it is zero, there was at least one bit in common between the two registers. The N-bit is set if the result indicates that both the most significant bit of the SRC /CON and DST is set (bit 7 in a byte compare and bit 15 in a word compare).

BIT is not the same as AND in that the result of the ANDing operation does not change the DST register. Testing for an odd number can be done with BIT, leaving the contents of the register unchanged after the instruction is executed. CMP would require the register to be ANDed with #1 before the comparison, losing the contents of the register.

BIT can be useful when, for example, testing a device's status bits. In the following code fragment, XM-23's control and status register (CLKCSR) is loaded into R0 and the DBA bit (indicating that a specific clock interval has passed) is checked using BIT. If the bit is clear, the Z is set, the DBA bit is not set; however, if the Z bit is clear, the DBA bit is set. It sometimes takes a moment to remember that the results are reversed of what one would logically expect:

```
   org  #0
;
; Clock control and status register in XM-23's first byte
CLKCSR   bss  1
;
   org  #1000
;
; To check the status register, need its address
; R0 <- #0000
```

```
;
    movlz    CLKCSR,R0    ; R0 - address of CLKSCR (base addr)
;
; Read status register, check bit 4
    ld  R0,R1        ; Check for Clock DBA (clear until clock tick)
    bit $4,R1        ; Z=1 (not set), Z=0 (set)
```

Using BIT rather than AND means that the register's bit can be inspected by subsequent instructions, possibly checking for other conditions being set. This capability is lost when using AND because the DST register is overwritten.

6.2 Register-exchange instructions

XM-23 supports two register-exchange instructions, SWAP (exchanges SRC and DST registers) and MOV (moves SRC register to DST register), shown in Table 10. These instructions do not change the PSW status bits.

Table 10: Register-exchange instructions

Instruction	Operation	Description
MOV(.B or .W) SRC,DST	*DST ← SRC*	Move SRC to DST. SRC can be a register or a constant.
SWAP SRC,DST	*TMP ← DST* *DST ← SRC* *SRC ← TMP*	Swap or exchange SRC and DST. TMP is an internal register that cannot be accessed by the programmer. SRC and DST are registers. R/C and W/B are ignored since SWAP exchanges register.

As an example, swapping the contents of registers R2 and R3 is written in a single instruction (rather than using, for example, the stack as a temporary location):

```
    swap     R2,R3    ; Contents of R2 and R3 are exchanged
```

A word or a byte can be copied between registers using MOV. For example, moving the LSB of R3 to the LSB of R0 requires the byte (.B) modifier:

```
    MOV.B    R3,R2    ; R3's LSB copied to R2's LSB, R2's MSB unchanged
```

The MSB cannot be copied, other than, for example, swapping the bytes in a register and then copying:

```
    SWPB     R3       ; MSB and LSB of R3 are exchanged
    MOV.B    R3,R2    ; MSB of R3 now copied into LSB of R2
    SWPB     R3       ; Restore R3 (if required)
```

Although MOV can move one of the predefined constants to a register, apart from -1, using one or more of MOVL, MOVLZ, MOVLS, or MOVH might be just as effective.

6.3 Single-register instructions without an operand

XM-23 has four single-operand instructions (sometimes called one-address instructions) that operate on a single register (DST). These instructions modify the contents of the register: moving bits, swapping bytes, or extending the sign.

Bit movement instructions fall into two categories:

Shift: The data is shifted either left or right, equivalent to multiplying or dividing by 2. Adding a number to itself is equivalent to a left shift. With left shift, a zero or the Carry-bit can be fed-into the least-significant byte and the most-significant byte can be discarded or assigned to the Carry-bit. Left shifting can be emulated using the ADD or ADDC instruction (see chapter 7).

Right shifting usually requires a special instruction, although the same effect can be achieved with an integer division. The least-significant bit can be discarded or copied into the Carry bit. The most-significant bit can be fed a zero-bit; this will mean the loss of the sign-bit. To maintain the sign-bit, right-shift instructions simply duplicate the value of the most-significant bit. Maintaining the sign-bit is often referred to as *arithmetic shifting*.

Rotate: In a rotation, the bits in the register can be moved simultaneously left or right. In a right-shift, the most-significant bit is copied into the least-significant bit, while in a left-shift, the least-significant bit is copied into the most-significant bit. The Carry-bit can be used as an intermediary. If the Carry-bit is used, rotating N+1 times (where 'N' is the size of the structure being rotated) returns the structure to its original value.

The instructions share a generic format with a nine-bit opcode and can operate on the word or byte of the instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode									W/B	0	0	0	DST		

The single-operand instructions are listed in Table 11.

Table 11: Single-operand bit-movement instructions

Instruction	Operation	Description
SRA(.B or .W) DST	*DST.MSB →…→ DST.LSB → C*	Arithmetic shift DST right one bit through Carry with sign extension. Shift can operate on a word or a byte. The Most Significant Bit (MSB) remains unchanged
RRC(.B or .W) DST	*C→ DST.MSB → … → DST.LSB → C*	Rotate DST right one bit through Carry. Rotate can operate on a word or a byte.

The arithmetic shift right (SRA) instruction maintains the value of the sign bit, thereby supporting signed division by 2 with the SRA instruction (shift-right arithmetic, that is, with sign extension); for example, dividing -16 (#FFF0), stored in R0, by 2:

```
    org #1000
Start
    movlz    $16,R0   ; R0 <- 16
    comp     R0       ; One's complement of R0
    add      #1,R0    ; Two's complement of R0
;
    sra  R0   ; R0/2 (-8 or #FFF8)
    sra  R0   ; R0/2 (-4 or #FFFC)
    sra  R0   ; R0/2 (-2 or #FFFE)
    sra  R0   ; R0/2 (-1 or #FFFF)
```

The rotate-right instruction (RRC) stores the Carry-bit into the most-significant bit while shifting the register's bits to the right by 1 and stores the least-significant bit into the Carry-bit. The following example counts the number of bits that are set in R0:

```
    clrcc    c   ; Clear carry bit - could already be set
;
    rrc      R0  ; Bits shifted right with C <- R0.lsb and R0.msb <- C
    addc     #0,R1    ; R1 < R1 + C (increment by 0 or 1)
```

A shift is different from a rotate in that a rotation performed N+1 times results in the original value, whereas repeating a shift sixteen times results in -1 or 0, depending on the original sign-bit. Rotate-left and shift-left are handled with emulated instructions (Chapter 7).

The byte-exchange and sign extension instructions have the same format:

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

The two instructions are list in Table 12.

Table 12: Single-operand byte-exchange and sign extension instructions

Instruction	Operation	Description
SWPB DST	*TMP ← DST.MSB* *DST.MSB ← DST.LSB* *DST.LSB ← TMP*	Swap bytes in DST (word only). W/B bit is ignored.
SXT DST	*bit 7→ bit 8 → … → bit 15*	Sign-extend LSB byte to word in DST (word only). W/B bit is ignored.

These instructions are byte specific. Swap byte (SWPB) exchanges the two bytes in a word, while sign extension (SXT) duplicates the most-significant bit in the first byte in the second byte.

The swap-byte instruction (SWPB) swaps the bytes in a register, whereas SWAP swaps the contents of two registers. To extract the most significant byte for subsequent operations, one could write:

```
    org      #80
Data
    byte     'A'  ; Low-order byte
    byte     'B'  ; High-order byte
;
    org      #1000
    movlz    Data,R0  ; R0 <- Address of Data
    ld       R0,R0    ; R0 <- ”BA” (#4241)
;
    swpb     R0       ; R0 <- ”AB” (#4142)
```

The most significant-bit of the least-significant byte can be extended using the sign extension instruction (SXT), allowing a signed 8-bit number to be extended to a signed 16-bit number. In the first example, R1 contains #FF0F (sign-bit of the byte is zero), when SXT is executed, the sign bit is extended and R1 becomes #000F. In the second example, R1 contain #0080 (the sign-bit is set), after SXT is executed, it contains #FF80.

```
    org      #1000
```

```
     movls    #0F,R1   ; R1 <- #FF0F
     sxt      R1       ; R1 <- #000F
;
     movlz    #80,R1   ; R1 <- #0080
     sxt      R1       ; R1 <- #FF80
```

6.4 Transfer of control

When a program executes, the effective address of the next instruction to be executed is stored in the program counter. As part of the instruction cycle, the program counter is incremented by 2 for the address of the next instruction to be fetched, decoded, and executed. If looping structures, such as C's while and do-while, are to be supported, it is necessary to have instructions that can explicitly change the program counter so that previously executed code can be executed again based on some condition.

Similarly, calling a subroutine will change the program counter. When the subroutine has finished execution, the program counter must be assigned a value that returns it to the instruction following the subroutine call.

Branching and subroutine calls are collectively referred to as *transfer-of-control* instructions.

XM-23's transfer-of-control instructions create an effective address which is *relative* to the address of the instruction *plus* 2. This is because the program counter is incremented immediately after the instruction is fetched and the execution of the transfer-of-control instruction using the current value of the (incremented) PC (i.e., the address of the instruction plus 2) for the next value of the program counter. For example, a branching instruction at location #1000 would use a PC value of #1002 in its address calculation.

Table 13 lists XM-23's thirteen transfer of control instructions, BL, for subroutine calls, and BRA, for to pass control to another instruction.

Table 13: Transfer of control instructions

Instruction	Operation (*label* is the left-shifted, sign-extended value of the offset)	Description	Type
BL label	*LR ← PC* *PC ← PC + label*	Branch with link to subroutine; store return address in LR. A return can be realized by using any instruction that copies LR into the PC, such as MOV or SWAP. See Chapter 11 on exceptions for additional information on LR.	-
BEQ label	*PC ← PSW.Z = 1 ? PC + label : PC*	Branch to label if equal (Z = 1)	-
BZ label		Branch to label if zero flag is set	
BNE label	*PC ← PSW.Z = 0 ? PC + label : PC*	Branch to label if not equal (Z = 0)	-
BNZ label		Branch to label if zero flag is cleared	
BC label	*PC ← PSW.C = 1 ? PC + label : PC*	Branch to label if carry set (C = 1)	Unsigned
BHS label		Branch to label if higher or same	
BNC label	*PC ← PSW.C = 0 ? PC + label : PC*	Branch to label if carry clear (C = 0)	Unsigned
BLO label		Branch to label if lower	
BN label	*PC ← PSW.N = 1 ? PC + label : PC*	Branch to label if negative (N = 1)	-
BGE label	*PC ← PSW.N ⊕ PSW.V = 0 ? PC + label : PC*	Branch to label if greater or equal	Signed
BLT label	*PC ← PSW.N ⊕ PSW.V = 1 ? PC + label : PC*	Branch to label if less than	Signed
BRA label	*PC ← PC + label*	Branch always (unconditional) to label	-

Since XM-23’s instructions must fit into 16-bits, both the opcode and the offset used to calculate the relative address must be stored together in this structure. The ISA allows 13-bit offsets for the BL instruction (see Figure 14) and 10-bit offsets for the BRA instruction (see Figure 15).

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

Figure 14: BL has a 13-bit offset

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

Figure 15: All other branching instruction have a 10-bit offset

6.4.1 Calculating the effective address of the new program counter

When the transfer-of-control instruction has finished executing the program counter has the address of the first instruction following the label specified in the BL or BRA instruction. The effective address is *relative* to the address stored in the program counter, which is the address of the BL or BRA instruction plus 2.

The design of the offsets was based on two observations:

- The relative address can be less or greater than the program counter making it is necessary to include the sign bit in the offset. The CPU will need to extend the sign bit when the relative address is being calculated.

- Although the relative address needs to be even (i.e., the least-significant bit is zero), it does not mean that the stored offset should be even. If the offset is right-shifted when the assembler generates the instruction (the offset is stored as an odd or even value), the CPU can left-shift it when calculating the relative address.

Regardless of the instruction, the effective address is calculated using the same algorithm:

- Extract the offset from the instruction (13 or 10-bits).

  The range of values for the 13-bit offset in the BL instruction is #0000 through #1FFF, with bit 12 as the sign bit.

  The range of values for the 10-bit offset in the BRA instruction is #000 through #3FF, with bit 9 as the sign bit.

1. Extend the sign-bit (bit 12 or 9) of the offset. If set, the most-significant bits are set, otherwise they are cleared.
2. Left-shift the offset; the least-significant bit will be clear (zero).

   The 13-bit offset used with BL will have a positive value in the range of +0 (#0000) through +8190 (#1FFE) or a negative value in the range -2 (#FFFE) through -8192 (#E000). See Figure 16.

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

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

Figure 16: 13-bit offset extended to 16-bits (original 13-bit offset shaded in green)
Top: Positive; Bottom: Negative

The 10-bit offset used with instruction other than BL has a positive value in the range 0 (#0000) through 1022 (#03FE) and a negative offset has a value in the range -2 (#FFFE) through -1024 (#FC00). See Figure 17.

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	0/1	0/1	0/1	0/1	0/1	0/1	0

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

Figure 17: 10-bit offset extended to 16-bits (original 10-bit offset shaded in green)
Top: Positive; Bottom: Negative

3. Add the offset to the program counter to obtain the effective address. The range of possible effective addresses relative to the address of the branching instruction is obtained from the equation *PC + 2 + sign-extended, left-shifted offset*:

   - For BL, the range is *PC - 8190* to *PC + 8192*,
   - For BRA, the range is *PC - 1022* to *PC + 1024*.

   The effective address is then assigned to the program counter, overwriting its current value.

   If the offset is -2, an infinite loop will occur; that is, the program counter refers to the transfer-of-control instruction.

6.4.2 Subroutine calls

The subroutine call BL passes control from the instruction calling the subroutine named in the operand to the subroutine

```
     BL   Subr     ; Call to Subr
```

The instruction stores the return address (the value of the program counter which has the address of the instruction following the BL) in the link register (LR). The link register does not require the use of the stack, making the instruction considerably faster the conventional subroutine calls since it does not need to push the return address onto the stack.

The ISA does not support a return instruction because a subroutine return can be achieved in a variety of ways, for example:

- If the subroutine does not change the link register, the return is simply a matter of moving the link register to the program counter:

```
Subr
; Instructions that do not change the link register (LR)
; ...
; Return by moving the LR to the PC
  MOV    LR,PC
```

- If the subroutine makes a call to another subroutine (using the link register), it is necessary to store the link register, typically on the stack:

```
Subr
; Save the link register on the stack
     ST   LR,-SP
;
     BL   Subr2         ; Call to a second subroutine
;
; Return by popping the stack into the PC
;
     LD   SP+,PC
```

- Similarly, inside a subroutine, if a sixth general-purpose register is needed, LR (R5) can be pushed onto the stack as used accordingly. On return, LR can be pulled from the stack and stored directly in the PC, causing control to return to the caller:

```
Subroutine
     ST   LR,-SP   ; Store LR on stack
     ...           ; Other instructions, LR (R5) can be used
     LD   SP+,PC   ; Pull LR from stack and store in PC
```

6.4.3 The branching instruction

XM-23's branching instruction is unconditional; when it is executed, the program counter is assigned the effective address determined from the label. This is equivalent to high-level language statements such as `go to`, `break`, and `continue`.

For example, an infinite loop can be created by writing a program which loops back on itself, sometimes written in C as `while(1)` or `for(;;)`:

```
While1Loop
```

```
;
; Statements inside while(1)
;
    bra  While1Loop
```

When control passes to this loop, the program remains in the loop until an external event can cause it to stop.

6.5 Conditional Execution

By itself, XM-23's unconditional branching instruction means that conditional statements such as IF-THEN-ELSE, WHILE, DO-WHILE, and FOR cannot be supported.

Conditional statements have a condition which determines the flow-of-control, changing the value of the program counter depending on the condition, for example:

```
IF Data = 8 THEN
    Alpha ← 1;
ELSE
    Alpha ← 2;
ENDIF
Beta ← 1;
```

If the condition, `Data = 8`, is true, `Alpha` is assigned the value 1 and then control passes to the statement following the `ENDIF`, `Beta ← 1`. However, if `Data` is not equal to 8, `Alpha` is assigned the value 2 and control continues to the statement `Beta ← 1`. This requires changing the value of the program counter depending on the condition.

XM-23's ISA has two instructions, CMP and BIT, that can be used to explicitly test for a condition; for example, to determine if Data (R0 in this case) has a value of 8, we could write:

```
    CMP  $8,R0
```

In addition to an unconditional branching instruction, many machines have a suite of conditional branching instructions, such as BEQ (branch if equal), BNE (branch if not equal), and BGT (branch if greater than). The above code could be written as:

```
; Assume Data, Alpha, and Beta are R0, R1, and R2, respectively
; IF Data = 8 THEN
    cmp     $8,R0
    bne     ElsePart     ; Branch if compare indicates Not Equal (Z=0)
; Alpha ← 1;
    movlz   $1,R1
    bra     EndIf
; ELSE
ElsePart
; Alpha ← 2;
    movlz   $2,R1
; ENDIF
EndIF
; Beta ← 1;
    movlz   $1,R2
```

6.5.1 The CEX Instruction

In other machines, notably pipeline RISC processors such as the ARM Cortex, efforts are made by the compiler-writer in conjunction with the systems designer to avoid branches whenever possible. In such ISAs, fetching instructions and not executing them is seen as less expensive than the overhead of branching and flushing the pipeline of unwanted instructions.

Since XM-23 is a RISC ISA and can be designed as a pipeline processor, it includes a conditional execution instruction, CEX (Figure 18). The instruction tells the CPU which PSW bits are to be tested (the code, C, bits 6 through 9). (The different codes, the suffix used by the assembler to determine the code, the description of the code, and the PSW bits to be tested are shown in Table 14.) Bits 3 through 5 indicate the number of instructions to be executed if the true-state condition is met (T, true-count) and bits 0 through 2 is the count of instructions to be executed if the condition is not met (F, false-count). The t-count and f-count values need not be equal, and the order of the conditional execution code bits is unrelated to the order of the PSW bits.

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

Figure 18: The Conditional Execution (CEX) instruction

Table 14: Conditional execution codes and their meanings

Assembler code	Description	PSW bit values inspected	Instruction code
EQ	Equal / equals zero	Z = 1	0000
NE	Not equal	Z = 0	0001
CS / HS	Carry set / unsigned higher or same	C = 1	0010
CC / LO	Carry clear / unsigned lower	C = 0	0011
MI	Minus / negative	N = 1	0100
PL	Plus / positive or zero	N = 0	0101
VS	Overflow	V = 1	0110
VC	No overflow	V = 0	0111
HI	Unsigned higher	C = 1 and Z = 0	1000
LS	Unsigned lower or same	C = 0 or Z = 1	1001
GE	Signed greater than or equal	N == V	1010
LT	Signed less than	N != V	1011
GT	Signed greater than	Z = 0 and (N == V)	1100
LE	Signed less than or equal	Z = 1 or (N != V)	1101
TR	True part is always executed	Ignored	1110
FL	False part is always executed	Ignored	1111

The conditions are based on combinations of the PSW conditional code bits, V, N, Z, and C after a subtraction or compare; the bit fields in the condition bear no relation to those in the PSW. The first eight (EQ through VC) are based on the single condition code bits, while the remainder are combinations of two or more condition code bits; the following assumes the result of the CMP instruction, DST - SRC|CON. For unsigned comparisons there are two instructions:[10]

HI: In unsigned arithmetic, the sign bit (indicated by the value of Negative bit in the PSW) is ignored. If DST is greater SRC, the Carry bit is set and the difference is non-zero (Z=1), the SRC and DST are not equal. If the PSW bits have values C=1 and Z=0, the result is considered unsigned higher.

LS: Unsigned lower or same is the complement of HI, that is, C=0 or Z=1. It can be obtained using de Morgan's laws.

Signed comparisons treat SRC and DST as signed (i.e., their most-significant bits are the sign bits) and require the use of the overflow, V bit, in combination with other PSW bits:

GE: Greater than or equal is determined from both the N-bit (Negative) and V-bit (oVerflow). If V is clear, the result of the subtraction has not overwritten the sign bit; the sign bit will not be overwritten if DST and ~SRC have opposite signs. In this case if N=0, DST ≥ SRC and if N=1, DST < SRC.

However, if V is set, the result of the subtraction has "overflowed", and the sign bit has been overwritten. This occurs when the sign of DST and ~SRC are the same and the sign of the result is different (for example, positive + positive gives a negative result or negative + negative gives a positive result). Note that values with the same sign do not necessarily cause overflow. When V is set, if N=0, DST < SRC, and if N=1, DST ≥ SRC.

For example, comparing $127 (SRC) and $-128 (DST) as bytes:

DST ($-128)	#80	*#80*	Value of DST
SRC ($127)	#7F		Value of SRC
~SRC	#80	*#80*	One's complement of SRC
DST + ~SRC		*#00*	Sum of DST and ~SRC
DST + ~SRC + 1		*#01*	Add 1 giving Result ($1)

At this point, overflow has occurred because DST and ~SRC are both negative and the Result is positive so V=1; since the result is positive (not negative), N=0. From this we can conclude that DST ($-128) is less than SRC ($127).

Comparing $-8 (SRC) and $120 (DST) as bytes:

DST ($120)	#78	*#78*	Value of DST in hex
SRC ($-8)	#F8		Value of SRC
~SRC	#07	*#07*	One's complement of SRC
DST + ~SRC		*#7F*	Sum of DST and ~SRC
DST + ~SRC + 1		*#80*	Add 1 giving Result ($-128)

In this case, overflow has occurred because DST and ~SRC are both positive and their sum is negative (V=1) and the result is negative, so N=1. Since both V and N are set, we can conclude that DST ($120) is greater than or equal to SRC ($-8).

Table 15 shows how the results of a comparison or subtraction instruction can be interpreted. The results are functionally equivalent to GE ← N XOR V.

[10] Subtraction is implemented as the addition of the DST to the complement of the SRC: DST + ~SRC + 1. (See section 8.3). In signed arithmetic (codes GE, LT, GT, and LE), the overflow condition, V, is determined from the sign of the DST and the sign of the ~SRC, not the SRC value.

Table 15: Result of CMP SRC,DST or SUB/SUBC SRC,DST (from DST+~SRC + Carry)

		N	
		0	**1**
V	**0**	GE	LT
	1	LT	GE

LT: From de Morgan's laws, LT (signed less than) is the complement of GE, or N != V.

GT: Greater than checks for GE's conditions (N == V) and ignores any equal cases (Z = 0).

LE: From de Morgan's laws, LE (signed less than or equal) is the complement of GT, Z =1 or N != V. Alternatively, it is a check for equal (Z = 1) combined with LT (signed less than).

When the CEX instruction is fetched, the condition is evaluated to either TRUE or FALSE. If the result is TRUE, the next *t-count* instructions are fetched, decoded, and executed, and the next *f-count* instructions are fetched and ignored. However, if the result is FALSE, the next *t-count* instructions are fetched and ignored, while the remaining *f-count* instructions are fetched, decoded, and executed. The instructions executed and skipped after a CEX instruction is executed are shown in Table 16.

Table 16: Instructions executed and skipped after a CEX instruction

Location after CEX instruction	**Result of Condition**	
	TRUE	**FALSE**
nnnn	Executed	Skipped
…	…	…
nnnn + (t_count – 2)	Executed	Skipped
nnnn + t_count	Skipped	Executed
…	…	…
nnnn + t_count + (f_count – 2)	Skipped	Executed
nnnn + t_count + f_count	Instruction not affected by CEX	

6.5.1.1 Examples

An IF-THEN-ELSE instruction requires two branches, the first to branch over the THEN-instructions to the start of the ELSE-instructions and the second, at the end of the THEN-instructions to branch over the ELSE-instructions.

The above example, of assigning 1 or 2 to Alpha depending on the value of Data could be written as follows using CEX to branch over the true-part instructions:

```
; Assume Data, Alpha, and Beta are R0, R1, and R2, respectively
; IF Data = 8 THEN
    cmp $8,R0
;
; Test the PSW bit for the NE condition (Z=0)
; If Z=0, then do the instruction following the CEX, otherwise skip
; There is no false-count, so execution would being immediately
; following the BRA:
;
    cex ne,$1,$0
    bra ElsePart ; Executed if NE condition is met
```

```
;
; Control passes here if NE condition not met (i.e., Data = 8)
;
    movlz    $1,R1
    bra      EndIf
;
; ELSE
; Control passes here is NE condition is met (i.e., Data != 8)
;
ElsePart
    movlz $2,R1
;
; ENDIF
;
EndIF
    movlz $1,R2
```

The above code can be optimized, removing all the branches and labels:

```
; Assume Data, Alpha, and Beta are R0, R1, and R2, respectively
; IF Data = 8 THEN
    cmp  $8,R0
;
; Test the PSW bit for the EQ condition (Z=1)
; If Z=1, then do the instruction following the CEX, otherwise skip
;
    cex      eq,$1,$1
;
    movlz    $1,R1    ; Executed if EQ, skipped if NE
;
    movlz    $2,R1    ; Skipped if EQ, executed if NE
;
    movlz    $1,R2
```

Using multiple branches can become expensive with certain software constructs, such as C’s conditional (ternary) operator (“?:”):

```
     Alpha = (Alpha >= 32) ? Gamma + Beta : Gamma – Beta;
```

The implementation with CEX does not require branching instructions (assume most-significant byte of Alpha, Beta, and Gamma is #00):

```
     movlz    Beta,R0
     ld       R0,R1
     movlz    Gamma,R0
     ld       R0,R2
     movlz    Alpha,R0
     ld       R0,R3
; (Alpha > 32)
     cmp      $32,R3   ; R3 (Alpha) - $32
;
; Result is greater if R3>32
; If GE, execute 1 and then skip 1
; If !GE, skip 1 and then execute 1
;
     cex      GE,$1,$1
     add      R1,R2        ; True-part: R2 = Gamma + Beta
     sub      R1,R2        ; False-part: R2 = Gamma - Beta
     st       R2,R0
```

While the savings may appear small, they can greatly improve the performance of pipelined processors.

Like IF-THEN statements, looping instructions that repeat a block of instructions, such as WHILE and DO-WHILE, also require the use of conditions, for example:

```
WHILE <Condition> DO
    /* Instructions to make the <Condition> FALSE */
END WHILE
```

The basic structure of a pre-test loop such as the above can be implemented using CEX as:

```
WhileStart
     ;
     ; Code to determine the value of <Condition>
     ;
     CEX <Condition>, Number of Instruction to execute in loop, $0
     ;
     ; Instructions to make <Condition> FALSE
     ;
BRA  WhileStart
```

If there are more than seven instructions to execute in the loop, it is necessary to obtain the complement of the condition and then branch to the first statement following the end of the loop:

```
WhileStart
     ;
     ; Code to determine the value of <Condition>
     ;
     CEX NOT <Condition>, $1, $0
     BRA EndWhile ; Executed if <Condition> is FALSE
     ;
     ; Instructions to make <Condition> FALSE
     ;
BRA  WhileStart
EndWhile
```

A common method of returning a Boolean indication from a subroutine is to for the subroutine to set or clear the Carry bit before returning to indicate whether the result is true or false (using SETCC or CLRCC, see section 6.6). The calling sequence can inspect the carry bit using CEX and CS (Carry bit set) or CC (Carry bit clear):

```
BL   Subr     ; Call to Subr
CEX CS,$CSCount,$CCCount
;
; Up to CSCount instructions to execute if carry is set
;
; Up to CCCount instructions to execute if carry is clear
;
; First instruction after Carry Set or Cleared instructions
```

An odd number has its least-significant bit set. This can be tested using BIT and CEX using EQ or NE as the condition; in this example, it is assumed that the even-number handling code requires more than seven instructions so a branch over the code is required:

```
     BIT $1,R0         ; And R0 with 1 – if odd Z=0
     CEX NE,$1,$0
     BRA OddNumber     ; Branch if not equal
     ;
     ; Code to handle even number
     ;
     BRA OddEvenCheckDone
OddNumber
     ;
     ; Code to handle odd number
     ;
OddEvenCheckDone
```

6.6 Program Status Word and Supervisory Call instructions

XM-23 has three instructions for accessing the program status word and an instruction for making system supervisory calls (or traps). All instructions share the same format, an 11-bit opcode and five instruction-specific (IS) bits:

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

The instructions are listed in Table 17.

Table 17: Program Status Word and

Instruction	Operation	Description

SETPRI	*IF NewPri < PSW.Current Priority THEN*
	PSW.Current Priority ← NewPri
	ELSE
	A CPU priority fault occurs
		Change the applications to a new priority that is lower than its current priority

SVC	*IF NewPri > PSW.Current Priority THEN* *Stack ← PC, LR, PSW, and CEX state* *PSW ← mem [VectBase + SA]* *PC ← mem [VectBase + SA + 2]* *LR ← #FFFF* *Clear CEX STATE information* *ELSE* *A priority fault occurs*	Control passes to supervisory control routine specified in one of XM-23's interrupt vectors. The requested priority must be greater than the current priority, otherwise a priority fault occurs.
SETCC	*IF PSW.Current Priority = 7 THEN* *Slp ← 0* *Set the specified PSW bits*	Set one or more of the PSW condition code bits or the SLP bit, or a combination of all five. Sleep cannot be set at priority 7.
CLRCC	*Clear the specified PSW bits*	Clear one or more of the PSW condition code bits or the SLP bit, or a combination of all five.

The set current priority instruction (SETPRI) changes the current priority of the running process; the new priority (0 through 7) is specified in the least-significant three bits of the instruction (0, 1, and 2 of the IS field). Bits 3 and 4 are cleared. The requested priority must be less than the existing priority. For example, to set the application's priority to 2, one could write:

```
SETPRI   #2
```

If the application's current priority is 2 or less, a priority fault will occur.

The supervisory call instruction (SVC) allows an executing application to generate an exception (a trap) to an exception handler, the address of which is stored in one of the 16 exception vectors (see Chapter 11). For example, to trap to the software associated with vector 4, one could write:

```
SVC  #4
```

As with all exceptions, the current priority must be less than the priority specified in the vector's PSW. The SVC must be to a higher priority application than the currently running application; if not, a priority fault will occur.

The handler can be the operating system; SVC allows an uninterruptible call to the system. In such a case, parameters are often passed through registers. SVC can pass control through any vector, include device interrupt vectors or system faults. This can be used to test exception handling software.

The SETCC and CLRCC instructions let the application set or clear any of four PSW condition code bits (V, N, Z, and C) and the SLP bit in the PSW.

The bits being set or clear should be listed together. For example, to clear V, Z, and SLP, we could write:

```
CLRCC    VZS ; 'S' denotes sleep
```

The order the bits are written is resolved into V, S, N, S, C by the assembler, so the following two instructions with their operands reversed are functionally identical:

```
     SETCC    VSNZC
     SETCC    CZNSV
```

These instructions can also be used to signal the success of a result in a subroutine call, for example:

```
     CMP      #3,R0
     BNZ      R0_NE_3  ; Not equal, clear C bit
     SETCC    C        ; R3=0, set C bit
     BRA      ChkDone  ; Done
R0_NE_3
     CLRCC    C        ; R3!=0, Clear C
ChkDone
```

Alternatively, a shorter version of the solution could be written using CEX:

```
     CMP      #3,R0
     CEX      EQ,#1,#1
     SETCC    C        ; If R0=3, set C bit (true)
     CLRCC    C        ; If R0!=3, clear C bit (false)
     MOV      LR,PC    ; return
```

7 Emulated instructions

The XM-23 ISA has a limited number of instructions when compared to other processors; for example, it does not have instructions for returning from a subroutine, incrementing registers, or accessing the stack. Table 18 lists an additional 26 instructions can be emulated from existing instructions using registers and, in many cases, constants. This can effectively reduce the size and complexity of the CPU design.

Table 18: Emulated instructions
(`.x` indicates a byte, `.B`, or a word, `.W`, can be specified)

Instruction	Emulation	Description
`ADC.`*x Rx*	`ADDC.`*x* `#0,`*Rx*	Add carry to Rx
`CALL` *subr*	`BL` *subr*	Call subr; Return address put in LR
`CLC`	`CLRCC C`	Clear PSW Carry bit
`CLN`	`CLRCC N`	Clear PSW Negative bit
`CLS`	`CLRCC S`	Clear PSW Sleep bit
`CLV`	`CLRCC V`	Clear PSW oVerflow bit
`CLZ`	`CLRCC Z`	Clear PSW Zero bit
`CLR` *Rx*	`MOVLZ #0,`*Rx*	Clear Rx
`DADC.`*x Rx*	`DADD.`*x* `#0,`*Rx*	Decimal add carry to Rx
`DEC.`*x Rx*	`SUB.`*x* `#1,`*Rx*	Decrement Rx
`DECD.`*x Rx*	`SUB.`*x* `#2,`*Rx*	Double Rx
`INC.`*x Rx*	`ADD.`*x* `#1,`*Rx*	Increment Rx
`INCD.`*x Rx*	`ADD.`*x* `#2,`*Rx*	Double increment Rx
`JUMP` *Rx*	`MOV` *Rx*`,PC`	Jump to destination (in Rx)
`NOP`	`MOV R0,R0`	No operation
`PULL` *Rx*	`LD SP+,`*Rx*	Stack Pull (POP) Rx
`PUSH` *Rx*	`ST` *Rx*`,-SP`	Stack Push Rx
`RET`	`MOV LR,PC`	Return from subroutine or interrupt Service Routine
`RLC.`*x Rx*	`ADDC.`*x Rx*`,`*Rx*	Rotate left Rx through carry
`SBC.`*x Rx*	`SUBC.`*x* `#0,`*Rx*	Subtract carry from Rx
`SEC`	`SETCC C`	Set PSW Carry bit
`SEN`	`SETCC N`	Set PSW Negative bit
`SEV`	`SETCC V`	Set PSW oVerflow bit
`SEZ`	`SETCC Z`	Set PSW Zero bit
`SLA.`*x Rx*	`ADD.`*x Rx*`,`*Rx*	Shift left arithmetic (shift left 1 bit) Rx; Multiply by 2
`TST.`*x Rx*	`CMP.`*x* `#0,`*Rx*	Test Rx for zero

Emulated instructions can be assembled to machine code directly by the assembler; however, this can add to the complexity of the assembler. A more straightforward approach is to have a *preprocessor* that takes an emulated instruction and translates it into its XM-23 equivalent.

The instructions produced by a preprocessor must follow the formatting rules of the target assembler. If the instruction is not in the translation table, it is left unchanged. Examples of the translation are listed in Table 19.

Table 19: Examples of input and output of preprocessor

Emulated	Translated	Description
`Call Sine`	`bl        Sine`	Call to Sine
`jump r3`	`mov       r3,PC`	PC ← r3
`RET`	`Mov       LR,PC`	PC ← LR
`clr.b R2`	`movlz     #0,R2`	R2 ← #0 (low and high byte)
`SLA.b R0`	`add.b     R0,R0`	R0 ← R0 + R0 (left shift)
`inc.b R3`	`add.b     #1,R3`	R3.low byte ← R3.low byte + 1
`add #1,R1`	`add       #1,R1`	Not emulated, therefore not translated
`CMP #2,R0`	`CMP       #2,R0`	Not emulated, therefore not translated

8 Arithmetic

8.1 Sign, carry, and overflow

Most arithmetic or shift/rotate operations can change the value of an operand's most-significant bit (the sign bit) or the carry-bit, or both.

The sign-bit (N-bit in the PSW) indicates the sign of the unit (either positive, '0', or negative, '1'), whereas the carry-bit (C-bit in the PSW) indicates that the operation required an extra bit to hold the result; for example, adding two numbers with the most-significant bits set will result in the carry-bit being set. The sign-bit is typically used in signed arithmetic, while the carry-bit can be used in both signed and unsigned arithmetic.

The overflow bit (V-bit in the PSW) is used in signed-arithmetic to indicate that the operation has exceeded the signed-variable range (i.e., the sign-bit and the result are invalid). Overflow can be set when adding, subtracting, or comparing; the rules are:

Addition rule: In addition, the initial destination and source signs are the same, but the resulting destination sign different from the initial:

Destination sign	Opr	Source sign	Destination sign (result)
Positive	+	Positive	Negative
Negative	+	Negative	Positive

Subtraction and comparison rule: In subtraction and compare, the initial destination (DST) and source (SRC) signs are different and the sign of the result is different from that of the destination (note that the compare instructions do not overwrite the destination):

Destination sign	Opr	Source sign	Destination sign (result)
Positive	–	Negative	Negative
Negative	–	Positive	Positive

Since XM-23's subtraction and compare instructions use addition to determine the result, this rule is redundant. See section 8.3.

The sign of the result (the most-significant bit) is part of the unit and its value immediately after the operation is stored in the PSW. However, neither the carry nor the overflow bits are stored with the unit; they are kept in the PSW until the next arithmetic operation changes them.

8.1.1 Examples

The following examples highlight the differences between the sign, carry, and overflow bits (8-bit examples are used):[11]

[11] Remember, XM-23's arithmetic operations are interpreted as *result = DST <operator> SRC*. For example, ADD R1,R2 means R2=R2+R1, while SUB R0,R2 means R2=R2-R0.

5 + (-2): In this example, 5 (#05) is added to -2 (#FE). The result is '+3'. The most-significant bit (bit 7) is '0', therefore PSW.N=0. However, since the result of the addition required an extra bit, PSW.C=1. The sign of the source ('+5') is the same as the result ('+3'), indicating that overflow has not occurred (PSW.V=0).

C	7/S	6	5	4	3	2	1	0	Value	Register
-	0	0	0	0	0	1	0	1	+5	DST
-	1	1	1	1	1	1	1	0	-2	SRC
1	0	0	0	0	0	0	1	1	+3	DST

127 + 2: Here, the largest possible 8-bit signed number (+127 or #7F) is added to 2 (#02), producing the result 129. While the result itself is correct (as an unsigned number), it exceeds the number of bits available to represent it (i.e., seven data bits and one sign bit), therefore an arithmetic overflow has occurred (the initial and resulting sign bits are different, so PSW.V=1). The addition did not result in a carry (the carry-bit is clear, or '0').

C	7/S	6	5	4	3	2	1	0	Value	Register
-	0	1	1	1	1	1	1	1	+127	DST
-	0	0	0	0	0	0	1	0	+2	SRC
0	1	0	0	0	0	0	0	1	+129	DST

-1 + (-8): In this case, two negative numbers are added, '-1' (#FF) and '-8' (#F8), giving the result #F7 or '-9'. The sign of the source and destination are the same, meaning that arithmetic overflow has not occurred (PSW.V=0). The sign and carry bits are set (PSW.N=1 and PSW.C=1).

C	7/S	6	5	4	3	2	1	0	Value	Register
-	1	1	1	1	1	1	1	1	-1	DST
-	1	1	1	1	1	0	0	0	-8	SRC
1	1	1	1	1	0	1	1	1	-9	DST

5 - 6: If subtraction is achieved using addition (that is, by adding DST to the two's complement of the SRC; see section 8.3), the addition rule regarding overflow can be used to determine if overflow has occurred. In this example, +6 (the SRC) is subtracted from the DST (+5). The result is obtained by obtaining the two's complement of 6 (-6 or #FA) and adding it to #05. The result is -1 and there is no carry, meaning that PSW.N=1 and PSW.C=0. Since the sign of the initial value of DST is different from the resulting value, there is no overflow; PSW.V=0.

C	7/S	6	5	4	3	2	1	0	Value	Register
-	0	0	0	0	0	1	0	1	5	DST
-	1	1	1	1	1	0	1	0	-6	SRC
0	1	1	1	1	1	1	1	1	-1	DST

-128 - 1: The DST (-128) is added to the two's complement of the SRC (-1 or #FE). The result is +127 (#7F) and PSW.N=0 and PSW.C=1. Applying the overflow addition rules to the operation and result, overflow has occurred because the sign-bit of the resulting DST value (0) is different from its original value, therefore, PSW.V=1. In unsigned arithmetic, overflow is ignored:

C	7/S	6	5	4	3	2	1	0	Value	Register
-	1	0	0	0	0	0	0	0	-128	DST
-	1	1	1	1	1	1	1	1	-1	SRC
1	0	1	1	1	1	1	1	1	+127	DST

The sign, carry, and overflow bits (in the PSW register) indicate the state of the most recent arithmetic operation. It is up to the software designer to decide whether the result is valid. For example, the flow of control in the following code depends on whether the arithmetic is signed or unsigned:

```
NUM1 EQU  $-1  ; #FFFF
NUM2 EQU  $-8  ; #FFF8
;
; The following addition adds the bytes -1 (#FF) (NUM1 in R0) to
; -8(#F8) (NUM2 in R1), and stores the result in -9 (#F7) (Num2 in
; R0). The status bits change: 'N' (set), 'V' (clear), and 'C' (set)
;
    MOVLZ    NUM1,R0
    MOVLZ    NUM2,R1
    ADD.B    R1,R0
;
; The result can be checked:
; Check for negative:
    BN       SignSet      ; Bit 7 is set
; Check for carry:
    BC       CarrySet     ; Carry is set (unsigned result > 8 bits)
;
; Value not negative and carry is not set
;
```

Arithmetic overflow can be checked using the CEX instruction and a branch, for example:

```
; Arithmetic operations that might change oVerflow
    CEX      VS,$1,$0
    BRA      OverflowSet
```

8.2 Addition

XM-23 supports both 8-bit and 16-bit addition using the ADD.B, ADD.W, and ADD instructions. Multiple-unit addition requires the use of the carry-bit (set or clear by the different ADD instructions) from the least-significant unit and then the add-with-carry instruction (ADDC.B, ADDC.W, or ADDC) for the subsequent units. For example, the following adds two 32-bit numbers (double-word or long on XM-23):

```
; DW0 = DW0 + DW1
;
DW0  DS   $4   ; 4 bytes of storage
DW1  DS   $4   ; 4 bytes of storage
; ...
    MOVL     DW0,R0   ; Low address byte of DW0 to R0
    MOVH     DW0,R0   ; High address byte of DW0 to R0
    MOVL     DW1,R1   ; Low address byte of DW1 to R1
    MOVH     DW1,R1   ; High address byte of DW1 to R1
    BL       DWAdd,LR
; ...
;
; DWAdd performs a 32-bit addition on two 32-bit structures pointed to
; by R0 and R1.  The result is stored in location pointed to by R0
; (the destination).  Return address is in LR.
;
DWAdd
    LD       R0,R2
    LD       R1+,R3
    ADD.W    R3,R2    ; R2 = R2 + R3 (DW0 and DW1 low words)
    ST       R2,R0+   ; Low word of DW0 = DW0 + DW1
                      ; C (carry) is clear or set
                      ; R0 incremented to address of DW high-word
; Repeat steps for high-word using
    LD       R0,R2
    LD       R1,R3
    ADDC.W   R3,R2    ; R2 = R2 + R3 + C (DW0 and DW1 high words)
    ST       R2,R0    ; High word of DW0 = DW0 + DW1
;
    MOV      LR,PC    ; Return
```

The sign bit has meaning only on the most-significant unit.

8.3 Subtraction

XM-23 performs subtraction using the method of complements by taking the ones-complement of the subtrahend, adding it to the minuend, and adding '1' to the result to give the difference. If the quantities being subtracted are represented in multiple units of data, the carry-bit is used to indicate the result of the addition plus the Carry bit in the lower-order structure (word or byte).

As with addition, there are two subtraction instructions. The first SUB (or SUB.W or SUB.B) is applied to the least significant (or only) unit of data. If the subtraction involves multiple units of data, a second instruction, SUBC (or SUBC.W or SUBC.B) is applied to each unit to include the value of the Carry bit.

The subtract instruction (SUB, SUB.W, and SUB.B) is implemented as follows:

- The operation of the subtract instruction is defined as:[12]

  dst + ~ src + 1 → dst

[12] The compare instruction (CMP, CMP.W, and CMP.B) all perform a non-destructive subtraction. That is, compare does a subtraction (as described in this section) without writing the result to the *dst*.

The *dst* (the minuend) is added to the one’s complement of the *src* (the subtrahend) plus ‘1’, the result is stored in the *dst* (the difference).

- The carry-bit is used for multiple-unit (i.e., bytes or words) subtraction. Its use is described below (section 8.3.1).

With these two requirements in mind, consider the following examples of subtracting bytes (i.e., using the SUB.B instruction):

3 – 2 (#03 – #02): In this example, #03 is the *dst* and #02 is the *src*. The first row indicates the Action (always addition), ‘C’ is the value of carry (its value is ignored, ‘-‘, until the subtraction has completed, ‘7’ through ‘0’ are the bit positions, and Comments describe what the bit pattern represents (i.e., the steps associated with each of the requirements listed above). The difference is #01 and there was no Carry. Since this is single-byte subtraction, Carry is ignored.

Action	C	7/S	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of #03
+	-	1	1	1	1	1	1	0	1	~*src*: 1s complement of #02
+	-	0	0	0	0	0	0	0	1	Add 1
	1	0	0	0	0	0	0	0	1	Answer: #01. ‘C’ set, ignored

3 – 3 (#03 – #03): In this example, the first #03 is the *dst* and the second #03 is the *src*. The difference is #00 and there was no Carry.

Action	C	7/S	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of #03
+	-	1	1	1	1	1	1	0	0	~*src*: 1s complement of #03
+	-	0	0	0	0	0	0	0	1	Add 1
	1	0	0	0	0	0	0	0	0	Answer: #00. ‘C’ set, ignored

3 – 4 (#03 – #04): In this example, #03 is the *dst* and #04 is the *src*. The difference is #FF or ‘-1’ and ‘C’ is clear. Carry can be ignored in this case as this is single-byte arithmetic.

Action	C	7/S	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of #03
+	-	1	1	1	1	1	0	1	1	*.NOT. src*: 1s complement of #04
+	-	0	0	0	0	0	0	0	1	Add 1
	0	1	1	1	1	1	1	1	1	Answer: #FF. ‘C’ clear, ignored

– 3 – 4 (#F5 – #04): In this example, #F5 (‘-3’) is the *dst* and #04 is the *src*. The difference is #F1 or ‘-7’.

Action	C	7/S	6	5	4	3	2	1	0	Comments
	-	1	1	1	1	1	1	0	1	*dst*: Value of #F5 (‘-3’)
+	-	1	1	1	1	1	0	1	1	*.NOT. src*: 1s complement of #04
+	-	0	0	0	0	0	0	0	1	Add 1
	1	1	1	1	1	1	0	0	1	Answer: #F1. ‘C’ set, ignored

3 – (-3) (#03 – #F5): In this example, #03 is the *dst* and #F5 is the *src*.

Action	C	7/S	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of #03
+	-	0	0	0	0	0	0	1	0	*.NOT. src*: 1s complement of #F5
+	-	0	0	0	0	0	0	0	1	Add 1
	0	0	0	0	0	0	1	1	0	Answer: #06. 'C' clear, ignored

The next section shows how the carry bit is used in multiple-byte subtraction.

8.3.1 Multiple-byte and multiple-word subtraction

If the subtraction involves N-bytes or N-words, it will require N-subtractions. Each byte or word in the subtraction, except the first, must use a different subtract instruction, SUBC (or SUBC.W or SUBC.B), to apply the Carry bit to the subtraction.

Using the last example from the previous section (3 – (-3) or #03 – #FD), but representing the two numbers as pairs of bytes rather than a single byte, that is #00.03 – #FF.FD (note that the entire subtrahend is complemented), one would find:

Most-significant byte (MSB)								Least-significant byte (LSB)								
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	#00.03
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	#FF.FD

Multi-byte arithmetic uses SUB.B on the least-significant minuend and subtrahend bytes and SUBC.B on the remaining pairs of minuend and subtrahend bytes. The subtraction (i.e., SUB.B) on the least-significant byte is identical to that shown above with the result indicating the state of the Carry bit (in this case it is clear):

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of LSB of #03
+	-	0	0	0	0	0	0	1	0	*.NOT. src*: 1s complement of LSB of #FD
+	-	0	0	0	0	0	0	0	1	Add 1
	0	0	0	0	0	0	1	1	0	Answer: #06. 'C' clear, used by high-order byte

It is necessary to use SUBC.B on the remaining bytes to ensure that the resulting Carry is included in the subtraction of the higher-order bytes. The subtract-with-carry instruction (both byte and word) is slightly different from the subtract instruction, as the instruction's requirements show:

- The operation of the SUBC instruction is defined as:

  $dst + {\sim} src + C \rightarrow dst$

  The minuend (*dst*) is added to the ones-complement of the subtrahend (NOT *src*). However, unlike the SUB instruction, SUBC adds the carry-bit (the result of the SUB operation).

- The value of the previous carry is added to the complement of the source and together, they are added to the destination.

With these two requirements, the operation of SUBC.B on the most-significant bytes of #0003 and #FFF5 (i.e., *dst* of #00 and *src* of #FF) is as follows:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	0	0	*dst*: Value of MSB of #00
+	-	0	0	0	0	0	0	0	0	*.NOT. src*: 1s complement of MSB of #FF
+	-								0	Add carry from LSB operation
	0	0	0	0	0	0	0	0	0	Answer: #00. 'C' clear, ignored by high-order byte

Combining the two bytes gives a value of #0006 ('+6'):

Most-significant byte (MSB)								Least-significant byte (LSB)								
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	1	0	#0006

SUBC.B works with non-zero most-significant bytes as well; for example, 259 – 4 or (#0103 – #0004) (using byte arithmetic):

Most-significant byte (MSB)								Least-significant byte (LSB)								
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	0	0	0	0	0	1	1	#0103
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	#0004

First, using SUB.B on the least-significant bytes (the ones-complement of #0004 is #FFFB):

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of LSB of #03
+	-	1	1	1	1	1	0	1	1	*~src*: 1s complement of MSB of #04
+	-	0	0	0	0	0	0	0	1	Add 1
	0	1	1	1	1	1	1	1	1	Answer: #FF. 'C' clear, used by high-order byte

Next, SUBC.B is used on the most-significant bytes:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	0	1	*dst*: Value of MSB of #01
+	-	1	1	1	1	1	1	1	1	*~src*: 1s complement of MSB of #00
+									0	Add carry from LSB operation
	1	0	0	0	0	0	0	0	0	Answer: #00. 'C' set, ignored by high-order byte

Since this is the last byte, carry can be ignored. The value of the resulting byte-pair is #00FF or 255, which is correct:

Most-significant byte (MSB)								Least-significant byte (LSB)								
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	1	1	1	1	1	#00FF

Two final examples of multiple-byte subtraction:

0 – 0 (#0000 – #0000): In this example, both the least and most significant bytes are zero:

First, using SUB.B on the least-significant bytes:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	0	0	*dst*: Value of LSB of #00
+	-	1	1	1	1	1	1	1	1	~ *src*: 1s complement of LSB of #00
+	-	0	0	0	0	0	0	0	1	Add 1
	1	0	0	0	0	0	0	0	0	Answer: #00. 'C' set, used by high-order byte

Next, SUBC.B is used on the most-significant bytes:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	0	0	*dst*: Value of MSB of #00
+	-	1	1	1	1	1	1	1	1	~*src*: 1s complement of MSB of #00
+									1	Add carry from LSB operation
	1	0	0	0	0	0	0	0	0	Answer: #00. 'C' set, ignored by high-order byte

Combining the LSB (#00) and the MSB (#00) gives a result of #0000.

259 – 260 (or #0103 – #0104):

First, using SUB.B on the least-significant bytes. This results in the Carry bit being clear, because #03 is less than #04:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	1	1	*dst*: Value of LSB of #03
+	-	1	1	1	1	1	0	1	1	~*src*: 1s complement of MSB of #04
+									1	Add 1
	0	1	1	1	1	1	1	1	1	Answer: #FF. 'C' clear, used by high-order byte

Next, SUBC.B is used on the most-significant bytes:

Action	C	7	6	5	4	3	2	1	0	Comments
	-	0	0	0	0	0	0	0	1	*dst*: Value of MSB of #01
+	-	1	1	1	1	1	1	1	0	~*src*: 1s complement of MSB of #01
+									0	Add carry from LSB operation
	0	1	1	1	1	1	1	1	1	Answer: #FF. 'C' clear, ignored by high-order byte

The result of 259 – 260 is the combination of the LSB and MSB (#FF and #FF) or #FFFF, a signed value of -1:

Most-significant byte (MSB)								Least-significant byte (LSB)								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	#FFFF

Due to page-width restrictions, all examples in this section are of 8-bit rather than 16-bit subtraction. However, the concepts are the same for 16-bit subtraction, the difference being that the carry bit is set or cleared as the result of an 8-bit subtraction (SUB.B and SUBC.B) or 16-bit subtraction (SUB, SUBC, SUB.W, and SUBC.W). Since this is a 16-bit machine, there is little call for performing 8-bit arithmetic unless the multiple-byte object is greater than 16-bits. For example, subtracting one 32-bit number (Num1) from a second 32-bit number (Num0) could be performed as follows:

```
Num0 DS      $4      ; Four bytes of storage
Num1 DS      $4      ; Four bytes of storage
;
     MOVL    Num0,R0 ; R0 = address of Num0
     MOVH    Num0,R0
     MOVL    Num1,R1 ; R1 = address of Num1
     MOVH    Num1,R1
;
     LD      R0,R2   ; R2 = first word of Num0
     LD      R1+,R3  ; R3 = first word of Num1
                     ; R1 now refers to second word of Num1
     SUB     R3,R2   ; R2 = R2 - R3
     ST      R2,R0+  ; First word of Num0 = Num0 - Num1
                     ; R0 now refers to second word of Num2
     LD      R0,R2   ; R2 = second word of Num0
     LD      R1,R3   ; R3 = second word of Num1
     SUBC    R2,R3   ; R2 = R2 - R3 (Carry included)
     ST      R2,R0
```

Registers R0 and R1 point to Num0 and Num1, respectively. The SUB.W instruction subtracts the least-significant words, storing the result in Num0. Both R0 and R1 are incremented by 2 to point to their respective most-significant words using ST and LD auto-increment, respectively. The SUBC instructions subtracts the most-significant words and includes the Carry from the previous SUB or SUBC instruction, storing the result in the most-significant word of Num0 (i.e., the location specified by the address of Num1 + 2, referred to by the address in R0).

9 Subroutines and parameters

XM-23 does not support functions and parameters directly, they are higher-level language constructs build from XM-23's subroutine call instruction, registers, and a software structure referred to as a *stack frame*.

9.1 Subroutine calls

A subroutine is called using the BL (Branch with Link) instruction:

1. LR ← PC
2. PC ← PC + 2 + sign-extended offset from BL instruction

The LR contains the return address. If an embedded subroutine call is to take place using BL or the LR is to be changed in the subroutine, LR should be saved (e.g., onto the stack).

Returning to the calling sequence uses either SWAP LR,PC or MOV LR,PC.

For example:

```
; Call to subrx
    BL   subrx    ; LR <- PC
; ...
Subrx
; ...
; Return by moving LR into the PC
    MOV  LR,PC    ; PC <- TOS (return address)
```

Alternatively, LR can be pushed onto the stack and used as the general-purpose register R5:

```
Subrx
    ST   LR,-SP   ; Push LR
;
; LR can now be used as a general-purpose register; for example:
;
    MOVLZ    R5   ; R5 and LR are the same register, LR is R5's alias
; ...
; Return by pulling LR and storing in PC
    ST   SP+,PC   ; Pull top-of-stack (pushed LR) and store in PC
```

9.2 Subroutine parameters

Any of XM-23's general-purpose registers, R0 to R4 can be used to pass parameters to a subroutine, therefore in some applications requiring five or fewer parameters may not to use the stack since the registers can be used. However, in those applications where it is either required or necessary, the stack is the most obvious choice of data structure to hold the parameters.

Each subroutine is associated with a *stack frame*, that part of the stack containing its parameters, return address, and automatic (or local) variables. Each stack frame also has a *frame pointer* (or *base pointer*), typically a register, which allows the subroutine to access the parameters (within the subroutine, referred to as parameters).

C parameters are pushed onto the stack from right-to-left, ensuring the first (leftmost) parameter is on the top of the stack. For example, the following code:

```
a = 10;
b = ’x’;
c = 4;
result = subr(a, b, c);
```

Can be implemented as follows, with parameters a, b, and c pushed onto the stack and then the call to subr is made (assuming a, b, and c are stored in registers R0, R1, and R2, respectively):

```
      st  R2,-SP
      st  R1,-SP
      st  R0,-SP
      bl  subr
```

The stack contains the following (remember, the stack grows from high memory to low memory):

1002	10	a	⇐ SP (Top of stack and first parameter)
1004	’x’	b	Second parameter
1006	4	c	First parameter
1008		Last word in caller’s stack frame	

The subroutine is written as follows (note that p, q, and r are referred to as *parameters*):

```
char subr(int p, char q, int r)
{
int i, j; /* Automatics */
i = p – r;
j = p + r;
return q;
}
```

In this example, it is assumed that R4 or BP is the stack-frame pointer.

The entry point to the called subroutine must preserve the caller’s stack frame by storing its BP on the stack. It then creates its own stack frame pointer (BP frame, the same register) by assigning the current top-of-stack (SP) to R4 (or BP).

Since the stack pointer points to the top-of-stack, by not decrementing it, the called subroutine’s BP points to the caller’s BP value on the stack. Finally, the SP is decremented by the number of bytes reserved for the automatics.

The entry-point assembler code for any subroutine is as follows:

```
      st      R3,-SP   ; Save previous BP on top-of-stack
      mov.w   SP,R3    ; New BP points to subr’s stack frame
      sub     $4,SP    ; Reserved for automatics i and j (two words)
```

The stack now contains:

0FFC	Reserved for j	⇐ SP	BP – 4 (automatic ‘j’)
0FFE	Reserved for i		BP – 2 (automatic ‘i’)
1000	Previous BP	⇐ BP	BP (caller’s BP)
1002	10	**p**	BP + 2 (parameter)
1004	’x’	**q**	BP + 4 (parameter)
1006	4	**r**	BP + 6 (parameter)
1008		Last word in caller’s stack frame	

BP can be used to access any value from the stack frame: positive offsets refer to parameters (`p`, `q`, and `r`) while a negative offset refers to the automatics. For example, BP + 4 refers to parameter `q`, while BP – 2 refers to the automatic variable `i`.

Since the stack frame is static and the locations of the parameters and automatics are known, they can be readily accessed using register-relative addressing and BP. The assembler code for the two statements can be written as follows:

```
ldr     BP,$2,R0      ; R0 = [BP+2] value of p
ldr     BP,$6,R1      ; R1 = value of r
mov     R0,R2         ; R2 = R0 (value of p)
sub     R1,R0         ; R0 (i) = R0 (p) – R1 (r)
str     R0,BP,$-2     ; [BP-2] (location for i) = R0 (value of i)
add     R1,R2         ; R2 (value of j) = R2 (p) + R1 (r)
str     R2,BP,$-4     ; BP-4 (j) = R2 (value of j)
```

If a subroutine is implemented as a function, it can return a value using a shared register. In this case, R0 is used (again, a purely arbitrary choice):

```
ldr     BP,$4,R0 ; R0 = [BP+4] (value of q)
```

Returning from the subroutine requires freeing the space reserved for the automatics and restoring the calling subroutine’s BP. These two operations can be implemented as follows:

```
mov     BP,SP    ; SP = BP (skip automatics)
ld      SP+,BP   ; Restore caller’s BP
```

The stack now contains:

1002	10	p	⇐ SP (Current top of stack and third parameter on stack)
1004	’x’	**q**	Second parameter on stack
1006	4	**r**	First parameter on stack
1008		Last word in calling stack frame	

The last instruction in the called subroutine is a return, which writes the value of the link register (LR) to the PC:

```
mov     LR,PC
```

The calling subroutine must remove its parameters from the stack as well as save the return value. Removing the parameters involves increasing the SP by (3 words); to do this means adding the constant 2 and then the constant 4 to SP (alternatively, a register could be used, the value $6 moved to it and then the register added to SP):

```
;
; Caller must discard parameters (3 words or 6 bytes in total) by
; adjusting the stack (SP ← SP + 6). This can be done by adding the
; constants $2 and $4 to the SP:
;
     add     $2,SP    ; SP ← SP + 2
     add     $4,SP    ; SP ← SP + 4
;
; R0 contains the return value from the subroutine.  It will need to
; be stored or ignored.
;
```

10 Devices

XM-23 is supplied with three devices that support: a programmable timer, keyboard input, and console (screen) output. The devices are memory-mapped, which means the program accesses the devices through XM-23's memory.

10.1 Device registers

Each device, regardless of its operation, is associated with a generic one-word structure consisting of a control/status register (CSR) and data register (DR):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data (I/O)								Reserved			ENA	OF	DBA	I/O	IE

The low byte (bits 0 through 7) is the control/status register, allowing access to the following status and control information:

IE: Interrupt enable (bit 0). Indicates whether this device is enabled for interrupts. It is enabled by setting the bit. If the bit is not set, the device can still be active; however, to determine whether a status change has occurred, it is necessary to poll the device. This is a control bit.

I/O: Input/output enable (bit 1). Indicates whether the device is for input (set bit) or output (clear bit). This is a control bit.

DBA: Data or buffer available (bit 2). If the device is programmed to be an input device and this bit is set, it means that data is available for reading. If the device is programmed to be an output device and this bit is clear, it means that the buffer is available for writing. The bit is cleared by the device when an input byte is read and set when the device writes a byte to the buffer. The bit is cleared by the device when an output byte is written and set by the device when the byte is transmitted. This is a status bit.

OF: Overflow (bit 3). This bit is set if the device is enabled for input and a byte is overwritten by a subsequent byte (i.e., it was not read quickly enough) or the device is enabled and the CPU writes bytes to the buffer before the device has time to transmit the byte. This is a status bit.

ENA: Device enabled (bit 4). Before a device indicate its status or transmit or receive data, it must be enabled. If the ENA bit is set, the device is enabled. If the bit is cleared, the device is disabled. This is a control bit.

Reserved: These bits are reserved for future use or device-specific encodings (bits 5 through 7).

The data register is one-byte in length:

Data (I/O): These bits are for typically for input or output, depending on how the device has been defined and subsequently enabled (bits 8 through 15). See sections 10.2 through 10.4.

The device ports are found in addresses 0x0000 through 0x000E. Device ports can be accessed as byte-pairs, the even-numbered byte holding the control/status register and the odd-numbered byte for the data register.

Each device is associated with a device number, which can be used to access either the device's port or its interrupt vector. For example, device 0 has port 0x0000 and vector 0xFFC0.

The status of a device can be determined by polling the device (i.e., repeatedly checking its status bits to determine if a change has occurred) or through interrupts. An interrupt occurs *after* an instruction has completed execution. Interrupts are discussed at length in Chapter 11 .

10.2 Timer

The timer is a clock that can be programmed to indicate the passage of a time-interval, from 3.92 milliseconds to 1000 milliseconds (1 second).

The timer is device number 0. Its CSR is byte 0 and its DR is byte 1.

The CSR bits are used as follows:

IE: If this bit is set and the timer is enabled, when the timer reaches zero, a timer interrupt is triggered.

I/O: The timer is an input device. This bit must be set.

DBA: Set if the timer has expired, clear otherwise. The bit is cleared when the timer's CSR is read.

OF: Set if the previous timer-expired indication (DBA set) had not been read before the next timer expiration.

ENA: If this bit is set, timer status changes will be detected. If it is cleared, timer-status changes will be ignored.

The time is multi-shot in that it is reset to the specified maximum after each status change, resuming its countdown.

The timer is programmed through its DR by indicating the number of status changes per second to be signalled. The longest interval (1 second) requires a DR value of 1 and the shortest (about 3.92 milliseconds) requires a DR value of 255. A zero takes the default value of 1. The default start-up value is 1. The interval is 1000 / (Value in DR) milliseconds.

10.3 Keyboard

Keyboard input (data-bytes) can be received by XM-23 through the programmable keyboard-input device. The keyboard is device number 1 and its CSR is byte 2 and DR is byte 3.

The keyboard CSR bits are used as follows:

IE: If this bit is set and the keyboard is enabled, the arrival of a keyboard input will result in a keyboard interrupt.

I/O: The keyboard is an input device. This bit must be set.

DBA: Set if a data byte is available (i.e., DBA), clear otherwise. The bit is cleared when the DR is read.

OF: Set if the previous data-byte had not been read before the next data-byte is received. The first data-byte is lost.

ENA: If this bit is set, keyboard status changes will be detected. If it is cleared, keyboard-status changes will be ignored.

The DR holds the most recently received character. The character does not change if the DR read. The DR is read-only (data written to it is ignored). The next data-byte received overwrite the current contents of the DR.

10.4 Screen

A program can write to the XM-23 console using the programmable screen-output device. The screen has device number 2 and its CSR and DR bytes are 4 and 5, respectively.

The screen CSR bits are as follows:

IE: If this bit is set and the screen is enabled, when a character has been written to the console, a screen interrupt occurs.

I/O: The screen is an output device. This bit must be cleared.

DBA: This bit is set when a data-byte has been sent (that is, the data-buffer is available). It is cleared when a data-byte is written to the DR.

OF: The overflow bit is not used by the screen subsystem.

ENA: If this bit is set, changes to the screen's status will be detected. If it is cleared, system activities are undefined.

10.5 Example

```
; Timer and console output
; Outputs a character to the screen whenever the timer reaches zero.
; The character is incremented after each clock interrupt.  This is
; polled device access.  Uses the default timer interval of 1 second.
;
; 13 April 2020
;
DEVREGS  equ $0  ; Base address of device registers
;
; Device offsets from device base address:
;
CLKCSR   equ $0  ; Clock Control and Status Register
SCRCSR   equ $4  ; Screen Control and Status Register
SCRDR    equ $5  ; Screen Data Register
;
ASC_A    equ 'A' ; Fist character to output
;
    org  #1000
Start
    movlz    DEVREGS,R0   ; R0 - base address for devices
    movlz    ASC_A,R3     ; R3 - output char, increment each second
;
; Wait for clock tick (1 sec)
;
InLoop
    ldr.b    R0,CLKCSR,R1 ; R1 <- mem[DEVREGS + CLKCSR]
    bit      $4,R1        ; Test CLK DBA bit (clear until clock tick)
    beq      InLoop       ; Repeat if zero, Z=1
;
; Reach here, DBA bit set indicating 1 second has passed.
; Clock tick indication is cleared by reading CSR (Clk DR)
;
    ldr      R0,$-1,R1    ; [KBCSR - 1]
```

```
;
; Clock tick - write ch (R3) to Scr
;
    str.b    R3,R0,SCRDR  ; mem[DEVREGS+SRCDR] <- R3 (ch)
;
; Wait until output completed
; Poll output DBA (data buffer available)
;
OutLoop
    ldr.b    R0,SCRCSR,R1 ; R1 <- mem[DEVREGS + SCRCSR]
    bit      #4,R1        ; Is DBA set?  Yes: Z=0
    beq      OutLoop      ; If Z=1 repeat SCR DBA poll
;
; Output done
;
    add.b    #1,R3        ; R3++ (ch++)
;
; Display first 128 ASCII characters only
;
    movlz    #7F,R4       ; R4 is mask register
    and.b    R4,R3        ; R3 <- R3 & R4
;
    bra      InLoop       ; Wait for next tick
;
    end      Start
```

11 Exceptions

This section explains how XM-23 handles *exceptions*, which collectively refers to interrupts, faults, and traps, where:

- An *interrupt* is a mechanism that allows an external device to signal the CPU that it has undergone a state change. Interrupts occur after an instruction has finished execution.
- A *fault* occurs when the CPU is executing an instruction that causes the CPU to enter a state that precludes it from functioning normally.
- A *trap* is a way for a running program to pass control to some form of control program, such as an operating system, a monitor, or a system service. A trap is an instruction that is executed by the CPU.

11.1 Overview of exceptions

When an exception occurs:

- The CPU is expected to suspend the currently running application, service the exception, and, depending on its cause, resume the suspended application.
- The CPU has state, notably the program counter and the program status word. Depending on the application being executed, various registers may have values as well. If the application is to be resumed after the interrupt has been serviced, the current state of the application must be saved when the interrupt occurs.

Servicing an exception requires a set of instructions dependent on the type of exception:

- Instructions handling an interrupt must query the device to determine its status and, depending on the device and the cause of the interrupt, give data to, or remove data from, it.
- Fault-handling instructions attempt to recover from the fault to allow the faulting instruction to complete its execution; otherwise, the application must be terminated.
- The instructions for the trap attempt to handle the service request from the application. When completed, control typically returns to the instruction following the trap.

The instructions serving the exception are often referred given the generic title, *interrupt service routine* or *ISR*. They can also be referred to as *exception handlers*.

An exception has characteristics like those of a subroutine in that the currently executing application stops while the subroutine or exception-handler performs its task. Similarly, the state of the caller or application associated with the exception should be saved so that it can resume correctly when the routine has completed.

A subroutine has an entry point which is available to the calling routine at *load-time*. Moreover, the calling routine calls the subroutine at specific points during its execution. The same cannot be said for an interrupted or faulting application because these exceptions do not necessarily occur at the same time and the application may not be aware of the device and its ISR or the fault handler.

This requires XM-23 to maintain a list of exception-handler *entry-points*; this list is usually referred to as *interrupt vector table* and is stored in the machine's memory. Each vector is associated with a specific exception (i.e., interrupt, fault, or trap). When the exception occurs, the CPU knows which handler is to be invoked:

Interrupts: The vector number of the interrupting device is supplied by the Programmable Interrupt Controller (or PIC) to the CPU when an interrupt occurs. The instruction cycle finishes before the CPU is informed of the interrupt.

Faults: Faults occur in specific parts of the CPU; for example, an invalid instruction fault is generated by the instruction decoder, while an invalid address fault occurs during the fetch phase of the instruction cycle. Faults typically occur during the instruction cycle. A fault sets the FLT bit in the PSW.

Traps: A trap is generated by an application using the SVC instruction to specify the vector number. Traps occur after the SVC has completed.

Although it is exception dependent, an exception causes the CPU to save some of the application's state (at a minimum, most machines save at least the program counter and the current program status word) on the running stack. Other state information may or may not be saved by the CPU. The program counter is then assigned the value of the handler's entry point from the interrupt vector table and the new PSW. On completion of the handler, instructions usually exist to restore the state of the application to allow its execution to resume.

In XM-23, the program counter, link register, program status word, and conditional execution state are saved on the stack when an exception occurs.

XM-23 has a 16-entry interrupt vector table stored in high memory from #FFC0 to #FFFF (see Figure 11). Each vector is associated with two words: a PSW (holding the new priority, in the low-word) and the address of the application handling the exception, the entry point (the new PC, in the high-word):

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
nnnn	Previous			-	-	-	-	FLT	Current			V	SLP	N	Z	C	PSW
nnnn+2																	PC

Vector 0 occupies words #FFC0 (PSW) and #FFC2 (vector 0's entry point), while vector 15 occupies words #FFFC (Active PSW) and #FFFE (Machine's reset address).

The SLP (sleep-state) bit of the new PSW must be clear (i.e., 0), otherwise the CPU will immediately enter the sleep state when the exception handler begins execution. To avoid this, PSW.SLP is cleared by the CPU before the handler begins execution.

11.2 Actions taking place during an exception

When an exception occurs, the CPU determines the address of the appropriate handler. If the priority of the handler in question (indicated in the PSW part of the handler's interrupt vector) is greater than the current priority (in the running PSW), the current state (PC, LR, PSW, and CEX State) is saved, the handler's PSW becomes the current PSW, and control passes to the handler (Previous-priority is an internal CPU register):

1. Push PC
2. Push LR
3. Push PSW
4. Push CEX State

5. Previous-priority ← PSW.CURRENT
6. PSW ← PSW of handler (from memory [Vector Address])
7. PSW.SLP ← 0
8. PSW.PREVIOUS = Previous-priority
9. PC ← Address of handler (from memory [Vector Address + 2])
10. LR ← #FFFF
11. Clear CEX State

The link register (LR) is saved and then assigned the value #FFFF. Since XM-23 does not support an interrupt-return instruction (other than, for example, MOV LR,PC), the value of #FFFF indicates to XM-23 that an exception has occurred. (This is used by the CPU after an exception has completed to indicate that, unlike a normal return, the CPU must pull the CEX state, PSW, LR, and PC from the stack.)

At this point, the CPU resumes the instruction cycle with a fetch from the address specified by the PC. Since this is the address of the first instruction in the handler (obtained from the interrupt vector), the handler begins execution. The handler is responsible for determining the cause of the exception and then responding to it. For example, an input device handler could check if data is available, and if it is, read the data register, while a trap handler could process the application's service request.

When the handler has completed its task, the LR value is copied into the PC. XM-23 detects this as an invalid address, indicating that an exception has completed. Rather than directly returning to the application (pulling the application's PSW, LR, and PC from the stack), it first checks if there is a pending interrupt that has a higher priority than PSW.PREVIOUS (i.e., the original application). If there is, the pending interrupt must be executed before control returns to any other lower-priority exceptions (possibly other interrupts or the original application).

These actions are performed during the fetch phase of the return from the exception handler (the end-of-exception subsystem):

```
1. IF PC = #FFFF THEN
   a) Check for pending interrupts
   b) IF no pending interrupts or pending device has lower priority than PSW.PREVIOUS
      1. Pull CEX State
      2. Pull PSW
      3. PSW.SLP  ← 0
      4. Pull LR
      5. Pull PC
   c) ELSE /* Pending device has a higher priority that PSW.PREVIOUS */
      1. Previous-priority ← PSW.CURRENT
      2. PSW ← PSW of handler (from memory [Device Vector Address])
      3. PSW.SLP  ← 0
      4. PSW.PREVIOUS ← Previous-priority
      5. PC ← Address of handler (from memory [Device Vector Address + 2])
   d) ENDIF
2. ENDIF
```

If there are no pending interrupts, the PSW is pulled from the stack, the program status and CPU priority are restored to the state prior to the interrupt. The SLP bit in the PSW is cleared by the CPU after the PSW is restored, thereby allowing the sleeping application to resume.

When returning from an exception, there cannot be a pending fault or trap as these would have occurred inside the exception handler before it ended.

By checking for pending interrupts rather than returning to the application and letting a pending device interrupt with a higher priority pre-empt the application, the overheads associated with the pulling and pushing of state information twice is avoided. This is referred to as *tail-chaining* and is used in the ARM Cortex.

11.3 Interrupts

A device can interrupt the CPU if it has a status change and its priority is greater than the current priority maintained in the PSW. During the interrupt, the handler must service the device. This is device dependent as discussed in Chapter 10 . On completion, the CPU's fetch phase determines whether to return to the application or start another interrupt handler (see section 11.2).

11.3.1 The Programmable Interrupt Controller

When a device is configured to interrupt, it does not signal the CPU directly, instead it signals the interrupt controller; each device has a separate interrupt channel to the Programmable Interrupt Controller (or PIC). In turn, the interrupt controller checks the priority of the device and if it is either (see Figure 19):

- The only device signaling an interrupt, or
- The highest priority device of several devices with pending interrupts.

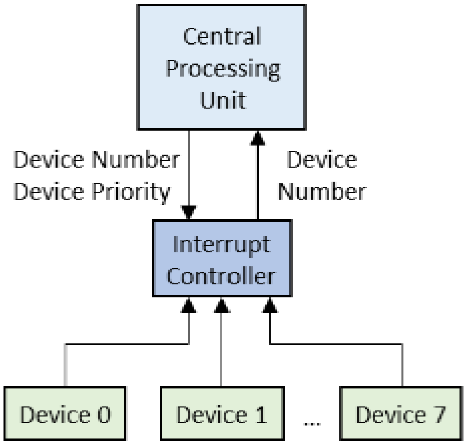

Figure 19: The relationship between the CPU, the PIC, and the devices

The PIC then signals the CPU that a specific device has generated an interrupt. If the device's priority is higher than the CPU's current priority, the CPU will service the device's interrupt (see section 11.2). The device's interrupt is cleared by the CPU accessing the device's registers.

The interrupt controller is programmable in that when the CPU updates the priority of a device by writing to the PSW word of its interrupt vector, the device's priority written to the PSW is also supplied to the interrupt controller. This means the device priority is the same for the interrupt controller and the CPU.

By having the PIC, the design of the CPU is simplified as there only needs to be one interrupt channel (from the PIC to the CPU) rather than separate channels to the CPU from each device.

11.3.2 Interrupt scenarios

The following section describes different device interrupt scenarios and how they are handled.

The first scenario is when there are no pending interrupts and is shown in Figure 20. When Device 1 interrupts, the application's PC, LR, PSW, and CEX state are saved on the stack, these registers are updated (the LR is assigned #FFFF to indicate that an interrupt return is required) and execution begins in Device 1's ISR. When the ISR has finished, the PC is assigned the value of the LC; on the next fetch, the CPU recognizes that an interrupt has finished and proceeds to restore the application's state. The SLP (sleep) bit is cleared because if the CPU was sleeping (waiting for an interrupt), the interrupt has occurred, so there is no need to sleep.

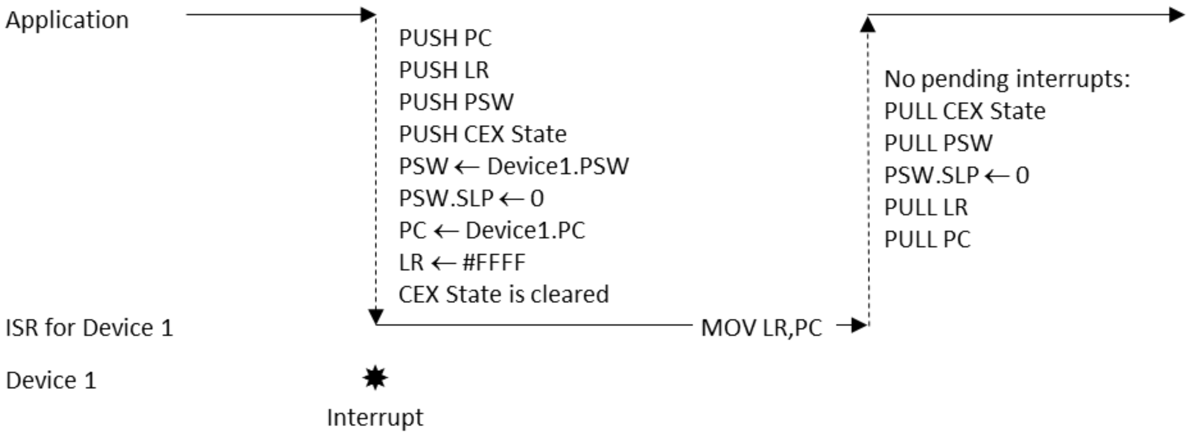

Figure 20: CPU actions during an interrupt from one device

A higher priority device can interrupt an exception, causing its state to be saved, giving the higher-priority device control of the machine. Figure 21 is an example of a higher-priority device interrupting a lower-priority ISR.[13]

The lower-priority state is saved on the stack (its LR is #FFFF) and the higher priority ISR executes (also with an LR of #FFFF). When Device 2's ISR has finished, the stack is pulled because its LR is equal to #FFFF, causing Device 1's ISR to resume execution since it is a pending interrupt. When Device 1's ISR has finished, its LR also indicates end-of-exception, causing the CPU to check for pending interrupts. Since there are none, the CEX state, PSW, LR, and PC are pulled from the stack. In this case, the application is resumed.

[13] This scenario is applicable to a higher-priority device interrupting a fault or a trap.

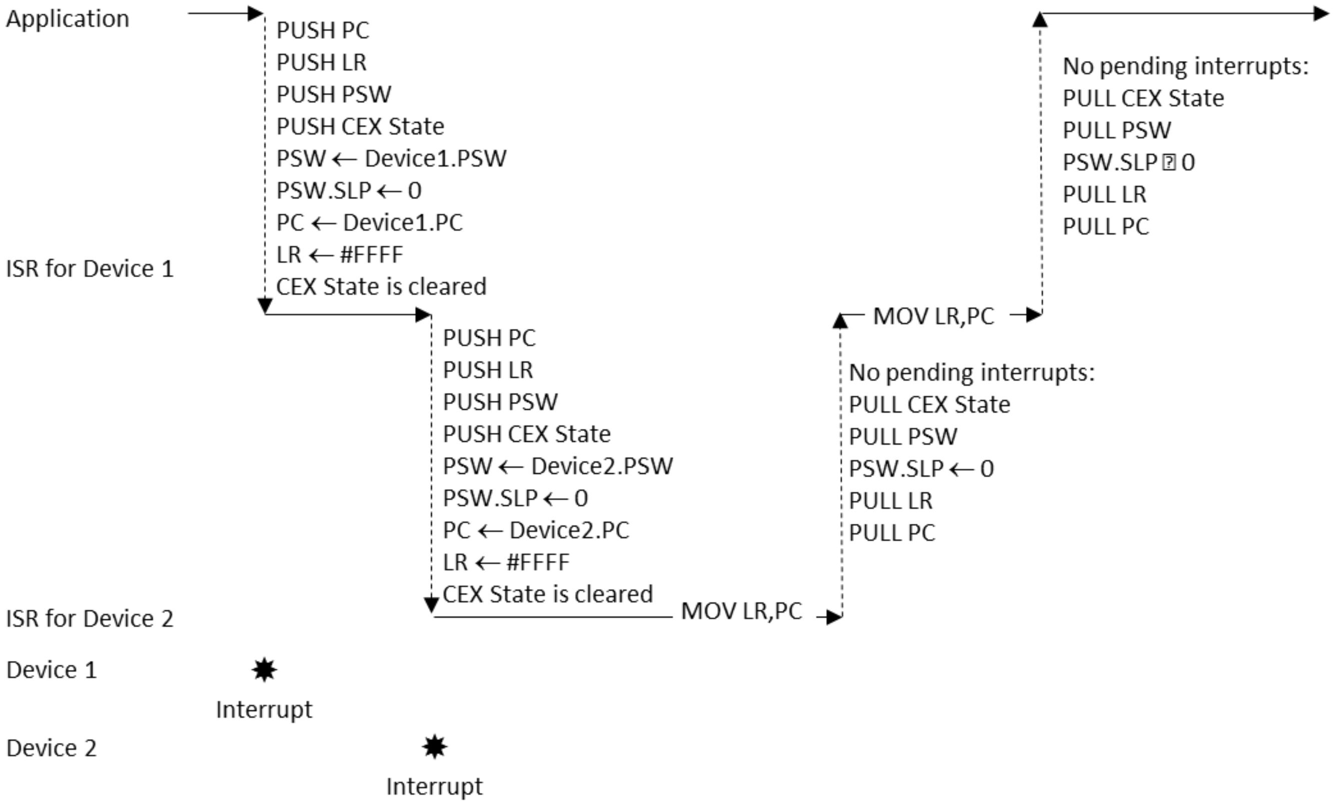

Figure 21: Actions taken when an exception handler (a device interrupt handler in this case) is interrupted by a higher-priority device

An example of *tail-chaining* is shown in Figure 22, where a high priority exception (a device ISR in this example, although it could be a fault or trap handler) is running when a lower-priority device undergoes a status change. When Device 2's ISR is completed, control passes through the end-of-exception subsystem which detects that Device 1 has a pending interrupt (signalled by the PIC). Rather than unstacking the application's registers, allowing the interrupt to occur, and restacking them, control passes to Device 1's ISR directly. When Device 1's ISR has completed, the end-of-exception subsystem detects there are no pending interrupts and pulls the application's CEX State, PSW, LR, and PC from the stack.

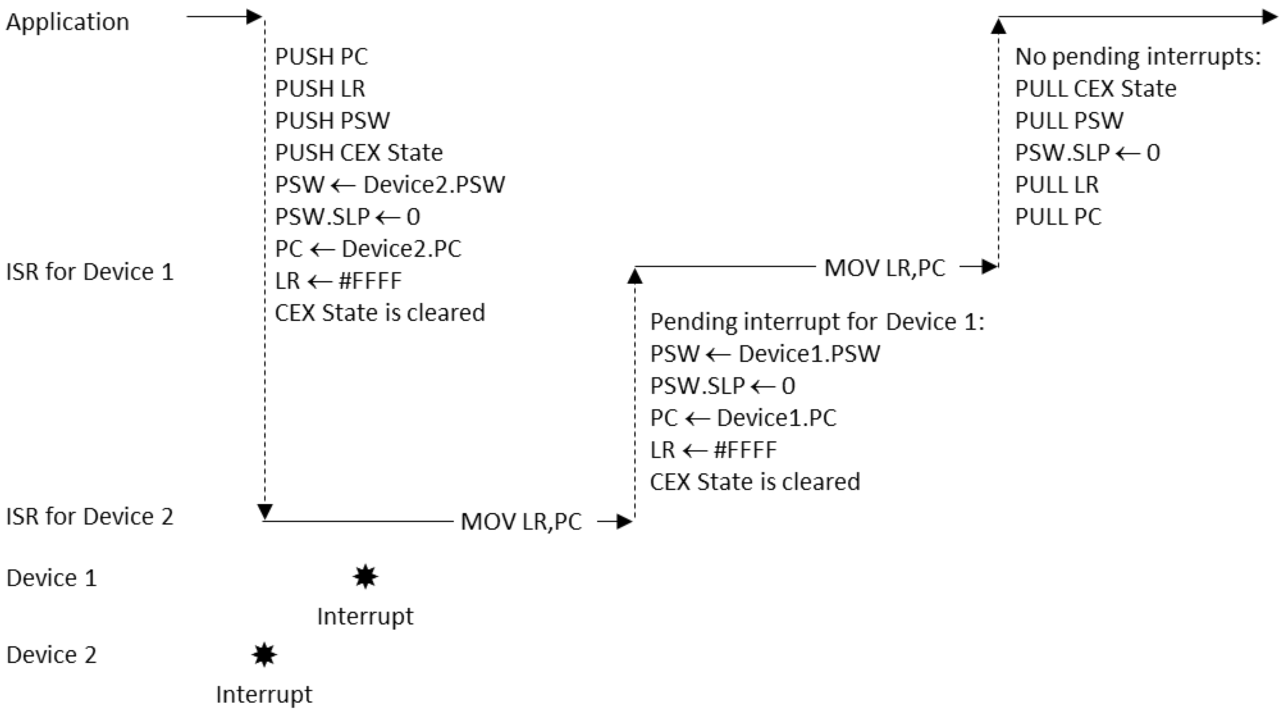

Figure 22: Tail chaining: Actions taken by the CPU when an interrupt is pending

11.4 Faults

A fault occurs during the execution of an instruction, putting the CPU into an undefined or unsafe state. Fault-handlers are intended to service these faults, typically one of stopping the execution of the application, changing the system state to allow the application to continue execution, or stopping the machine entirely.

XM-23 currently supports four faults (the vector number associated with the fault is listed in parenthesis):

Illegal instruction fault ($8): The CPU attempts to decode an unknown instruction.

Invalid address fault ($9): If the CPU attempts to fetch an invalid address (i.e., an odd numbered address in the program counter), an invalid address fault will occur.

Priority fault ($10): A priority fault occurs when an application at priority X executes an SVC that attempts to trap to an application that is the same or lower priority than the application current priority. The fault can also occur if the application changes the current priority to a higher priority using a SETPRI.

Double fault ($11): The application is in a fault state and another fault occurs, leading to a double fault. The CPU should stop execution at this point to avoid an infinite cycle of double faults.

An example of a fault is given in Figure 23. An application executes an illegal instruction, causing XM-23's fault-handling subsystem to start the fault handler specified at vector 8. The fault handler is then invoked and attempts to handle the fault. In this example, the fault handler starts a new application. Normally, a diagnostic would be displayed to inform the user of the fault.

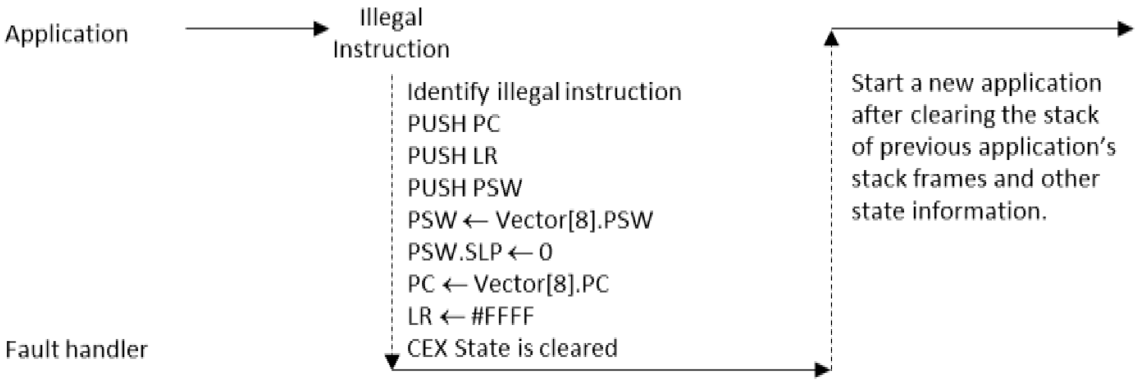
Application
Illegal Instruction
Identify illegal instruction
PUSH PC
PUSH LR
PUSH PSW
PSW ← Vector[8].PSW
PSW.SLP ← 0
PC ← Vector[8].PC
LR ← #FFFF
CEX State is cleared
Fault handler
Start a new application after clearing the stack of previous application's stack frames and other state information.

Figure 23: Fault example: Illegal instruction

11.5 Traps

A trap occurs when an application executes the SVC (supervisory call) instruction (unlike a fault, the SVC instruction has completed and the PC refers to the next instruction following the SVC). SVC has one operand, the vector number of the exception which is to be executed. The operand's value can take any value between 0 and 15, meaning that a trap can invoke any interrupt handler, fault handler, or trap handler. It should therefore be used judiciously.

A typical trap takes the path shown in Figure 24. The application's state is saved and the handler is executed. If a higher priority exception occurs, the handler will be pre-empted until completion of the higher-priority handler. On completion of the trap, the CPU checks for pending exceptions.

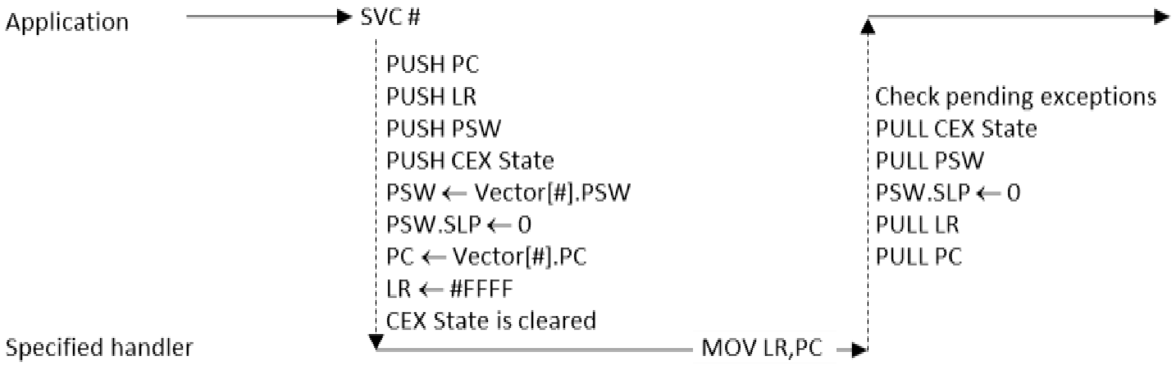

Figure 24: Actions taken during a trap

A trap cannot pass control to a lower-priority handler. If this attempted, a priority fault will occur (see section 11.4).

11.6 Example

The following example shows how clock and screen interrupts can be implemented on XM-23. The clock interrupts 16 times a second (every 62.5 milliseconds) and the screen interrupts when an output has completed.

```
;
; Clock tick and Screen output interrupts example
; Clock interrupts 16 times/sec.  Clock ISR increases the value of the
; character (loc #80) while the screen ISR writes the character.
; To get things started, the character must be written to cause the first
; interrupt.
;
; ECED 3403
; 19 April 2019
;
CLKCSR   equ  #0
CLKDATA  equ  #1
SCRCSR   equ  #4
SCRDATA  equ  #5
;
INTR_ENA equ  $1
;
LR   equ  R5   ; Alias for LR
SP   equ  R6   ; Alias for SP
PC   equ  R7
ASC_A    equ  'A'
;
VECTORS  equ  #FFC0
XMVECT   equ  #FFFC    ; Machine vectors (PSW and Reset)
;
CLK_INT  equ  $16  ; 16 CLK interrupts/sec
;
PSW3 equ  #0060    ; Priority 3
PSW4 equ  #0080    ; Priority 4
PSW5 equ  #00A0    ; Priority 5
PSW6 equ  #00C0    ; Priority 6
PSW7 equ  #00E0    ; Priority 7
;
APP_PRI  equ  #3   ; Application priority is 3
;
; Data
;
     org  #0080
Ch   byte ASC_A
;
; Code
;
     org  #800
;
; Something's wrong.  Stop execution
;
Reset    bra  Reset
;
     org  #1000
;
; Application
;
Start
;
; CPU priority is default 7 at start
; Initialize for clock and screen interrupts
;
```

```
    movls     VECTORS,SP    ; SP = FFC0 (TOS)
    movlz     CLKCSR,R0     ; R0 - address of CLKSCR (base addr)
;
; Enable CLK interrupts
;
    ld.b R0,R1          ; R1 = [CLKSCR]
    bis  INTR_ENA,R1    ; IE = 1
    st.b R1,R0          ; [CLKSCR] = R1
;
; CLK interrupt interval
;
    movlz     CLK_INT,R1
    str.b     R1,R0,CLKDATA ; [CLKSCR+1 or CLKDATA] = CLK_INT
;
; Enable SCR interrupts
;
    ldr.b     R0,SCRCSR,R1  ; R1 = [SCRCSR]
    bis       INTR_ENA,R1   ; IE = 1
    str.b     R1,R0,SCRCSR  ; [SCRCSR] = R1
;
; Write first character ('A' from Ch)
;
    movlz     Ch,R1         ; R1 = address of Ch
    ld.b      R1,R1         ; R1 = [R1]
    str.b     R1,R0,SCRDATA ; [R0] = R1
;
;
; Lower CPU priority to allow interrupts
;
    setpri    APP_PRI
;
WaitLoop
;
    bl   DoSomething
    bra  WaitLoop       ; Wait for next interrupt
;
; Subroutines
;
DoSomething
; Do something (not much actually) while in the background while the
; interrupts work in the foreground. Set sleep (slp) bit in PSW
;
    setcc     s         ; Set SLP
;
; Sleep until interrupt
; Wakeup - return
;
    mov  LR,PC          ; return
;
; ************************************************************
;
    org  #2000
;
; Clock ISR - called when clock interrupt detected
;
ClkISR
    st   R0,-SP         ; push R0
```

```
     st   R1,-SP         ; push R1
;
; Clear interrupt
; Clear clock signal by reading ClkCSR
;
     movlz     CLKCSR,R0
     ld.b      R0,R1     ; Should check OV for missed tick
;
; Change character
; Should be a protected operation?
;
     movlz     Ch,R0     ; R0 = addr of Ch
     ld.b      R0,R1
     add.b     #1,R1     ; R3++ (ch++)
     st        R1,R0     ; Update R1 to next character
;
; Done
;
     ld   SP+,R1    ; pull r1
     ld   SP+,R0    ; pull r0
     mov  LR,PC     ; return
;
; Screen ISR - called when output completed
; Load and write next character
;
ScrISR
     st   R0,-SP    ; push R0
     st   R1,-SP    ; push R1
;
; Display Ch
;
     movlz     Ch,R0          ; R0 = addr of Ch
     ld.b      R0,R1          ; R1 (Ch) has character
     movlz     CLKCSR,R0      ; R0 = addr of CLKCSR
     str.b     R1,R0,SCRDATA ; Write Ch (R1) to screen
;
; Done
;
     ld   SP+,R1    ; pull r1
     ld   SP+,R0    ; pull r0
     mov  LR,PC     ; return
;
; Interrupt vectors
;
     org  VECTORS
CLKPSW    word PSW5      ; Clock vector
CLKISR    word ClkISR
KBPSW     word PSW7      ; KB not used, use as spacer
KBISR     word Reset     ; Control to Reset if KB interrupt occurs
SCRPSW    word PSW4
SCRISR    word ScrISR
;
     org  XMVECT
word PSW7      ; Starting PSW
word Reset     ; Reset address
;
     end  Start
```

12 Initial CPU state

When the CPU is first powered-up, vector 15, the Reset vector, is accessed for the initial PSW (word #FFFC) and the address of the reset or start-up subroutine (i.e., word #FFFE). No other registers are initialized; this must be done in the subroutine.

The default values for the Reset vector are:

- The program counter (from #FFFE) is #0800. The first executable instruction is assumed to be stored in location #0800.
- The stack pointer is #0800. This does not conflict with the program counter value since the stack moves “down”; the first register pushed to the stack is stored in memory location #07FE. The value of the stack point is not stored in an interrupt vector.
- The program status word (from #FFFC) is #60E0: all status bits (C, N, Z, and V), SLP, and Fault are clear, the current priority is 7, and the previous priority is 3.

If the program counter and program status word values are supplied in the loaded program, the reset or start-up subroutine should operate with the highest priority (7) to ensure that it is not interrupted. This is indicated in the reset PSW.

The reset vector is not used when running XM-23’s debugger.

13 Structures

Programming languages can be discussed in terms of code structures and data structures. This section gives some examples of how XM-23 could support C code and data structures.

13.1 Code structures

13.1.1 Sequential operations

A sequential operation (such as an assignment statement) consists of a set of instructions that does not alter the program's flow of control. In other words, the program counter is simply incremented through the instructions. The following code fragment contains three sequential statements:

```
a1 = a1 + 1;
a2 = a1 + 3;
a3 = a1 + a2 * 2;
```

Since XM-23 offers a (relatively) large number of registers, tools such as compilers and assembly-coders can use registers to reduce the number of memory accesses, thereby increasing the speed of the program, for example:

```
;
a1   word     #11      ; a1 initial value #11 ($17)
a2   bss      $2       ; Reserve two byte (1 word) for a2
a3   bss      $2       ; Reserve two byte (1 word) for a3
;
;Access a1’s initial value from memory
;
     MOVL     a1,R0    ; Low-byte of R0 <- low-byte of a1’ address
     MOVH     a1,R0    ; High-byte of R0 <- High-byte of a1’ address
;
; a1 = a1 + 1
;
     LD       R0,R1    ; R1 <- mem[R0] or value of a1
     ADD      #1,R1    ; a1 <- a1 + 1
     ST       R1,R0    ; Write a1 to its memory location
;
; Reuse R1
     MOV      R1,R2        ; Use R2 for a2
                       ; Initialize a2 with a1
     ADD      $2,R2    ; To get 3, add 2 and then add 1
     ADD      $1,R2    ; a2 = a1 + 3
;
     MOVL     a2,R0    ; Update a2’s value in memory
     MOVH     a2,R0    ; R0 is being reused
     ST       R2,R0
;
; Assuming a2 is never referenced again, it can be used for a3.
; That is, the value is a2 is used in the equation as the initial
; value of a3
```

```
;
      ADD     R2,R2         ; a3 (R2) = a2 (i.e., R2) * 2
      ADD     R1,R2         ; a3 = a3 + a1
;
      MOVL    AddrA3,R0     ; Update a3's value in memory
      MOVH    AddrA3,R0
      ST      R2,R0
```

By performing arithmetic operations in the CPU's registers, the number of memory accesses can be reduced. This, and other of ISA constructs, can improve program performance. Since memory is inexpensive, the additional instructions required may be considered a small price to pay.

13.1.2 Conditional statements

A conditional statement changes the program's flow based upon some condition being met. This is usually described in terms of one of three constructs: IF-THEN-ENDIF, IF-THEN-ELSE-ENDIF, and SWITCH-CASE.

13.1.2.1 IF condition THEN true-part ENDIF

An IF-statement with a true-part must test the condition to determine whether the condition is met (i.e., is true) in order to allow the true-part code to be executed. In a C-condition such as:

```
if (a==b)
{
     /* ... */
}
```

The condition begins with using the compare instruction:

```
     CMP  R0,R1    ; Assume R0 (a) and R1 (b)
```

The test, to determine whether 'a' equals 'b', could be:

```
     CMP  R0,R1    ; R0 – R1, if zero, PSW.Z=1

BEQ  TruePart ; Go to TruePArt if PSW.Z=1
```

There are two problems here. First, deciding where TruePart is located, and second, determining what should take place after the BRA TruePart instruction. A possible solution is to write the code as follows:

```
     CMP  R0,R1    ; R0 – R1, if zero, PSW.Z=1
     BEQ  TruePart ; Go to TruePart if PSW.Z=1
     BRA  EndIF    ; Go to EndIF if PSW.Z != 1 (which it must be)
TruePart
     ; TruePart instructions
EndIF
```

The BRA EndIf is unnecessary. By applying de Morgan's rules, the EQ can be replaced with an NE to the EndIF instruction and the second BRA can be removed. The instructions following the BRA EndIf are only executed if the condition is true:

```
      CMP  R0,R1         ; R0 - R1, if zero, PSW.Z=1
      BNE  EndIF         ; Go to EndIF if PSW.Z != 1
      ; TruePart instructions
      ; ...
EndIF
```

In some situations in C, the condition is simply a variable and the compiler is to generate code to determine if the condition is zero or non-zero. For example:

```
if (a)
     /* Instructions to execute if a is not equal to zero */
endif
```

This can be implemented with the test (TST) extended instruction (test for equal to zero) and an EQ:

```
      CMP $0,R0
      BNE  EndIF
      ; Instructions to execute if a is not equal to zero
```

In this case, if R0 is equal to zero, control passes to the end-if (or, if used, the else-part) of the IF.

C also allows conditions to short-circuit, meaning that a Boolean expression using '&&' and '||' need not be fully evaluated before control passes to the true or false part.

An example of a short-circuit using '&&' (Boolean 'and'):

```
if (a==b && c!=d)
```

For the true part to be executed, 'a' must equal 'b' *and* 'c' must not equal 'd'. This can be implemented as (assume 'a' is R0, 'b' is R1, 'c' is R2, and 'd' is R3):

```
      CMP  R0,R1
      BNE  FalsePart    ; If a != b, no need to check c and d
; At this point, a=b
      CMP  R2,R3
      BEQ  FalsePart    ; If c == d, true part should not be executed
;
; At this point, a==b and c!=d
; ...
      BRA  EndIF
FalsePart
; ...
EndIF
```

An example of short-circuiting using '||' (Boolean 'or'):

```
if (a==b || c!=d)
```

For the true part to be executed, either 'a' is equal to 'b' *or* 'c' is not equal to 'd'. This can be implemented as (assume 'a' is R0, 'b' is R1, 'c' is R2, and 'd' is R3):

```
    CMP  R0,R1
    BEQ  TruePart ; If a == b, no need to check c and d
;
; At this point, a != b
;
    CMP  R2,R3
    BEQ  FalsePart    ; If c == d, true part should not be executed
TruePart
;
; At this point, a == b or c != d
; ...
    BRA  EndIF
FalsePart
; ...
EndIF
```

13.1.2.2 IF condition THEN true-part ELSE false-part ENDIF

This is simply a variation on an IF-statement with the following changes:

- The branch instruction following the condition passes control to the first instruction in the false part.
- The last instruction of the true part must be an unconditional branch to EndIF.

As an example:

```
if (a>=b)
    a=b;
else
    a=0;
```

This can be implemented as (assume R0 is 'a' and R1 is 'b'):

```
    CMP  R0,R1
    BLT  ElsePart
    MOV  R1,R0
    BRA  EndIF
ElsePart
    CLR  R0
EndIF
```

13.1.2.3 Switch-Case

Switch-case is discussed in section 13.2.2 (Switch-Case implementation using arrays), below.

13.1.3 Looping statements

There are three types of loop structure: pretest, post-test, and deterministic.

13.1.3.1 Pretest

In a pretest loop, the condition for remaining in the loop is tested before entering the loop. The C pretest loop structure is the while statement. The loop is executed zero or more times. For example:

```
i = 0;
while (i < 16)
{
    /* ... */
    i = i + 1;
}
```

Can be implemented as:

```
; i = 0
    MOV  $0,R0        ; Use R0 as ’i’
; while (i < 16)
WhileStart
; {
    CMP  $16,R0   ; tmp = R0 – 16
    BGE  WhileEnd ; remain in loop until R0>=10
; ...
; i = i + 1
    ADD  $1,R0
    BRA  WhileStart
; }
WhileEnd
```

The different conditions discussion in section 13.1.2, above are applicable in both pre- and post-test loops.

13.1.3.2 Post-test

In a post-test loop, the condition for remaining in the loop is tested after the loop has been executed at least once. The C post-test loop structure is the do-while statement. For example:

```
i = 10;
do
{
    /* ... */
    i--;
}
while (i > 0);
```

Can be implemented as:

```
; i = 10;
    MOVLZ    $10,R0
DoStart
; ...
; i--;
    SUB      $1,R0    ; N and Z bits can change
    BNE      DoStart  ; If not equal to zero, repeat
```

13.1.3.3 Deterministic loops

A deterministic loop, such as C’s ‘for’ statement can be implemented as a pre-test loop with a known starting value, ending value, and increment. For example

```
for(ch = ’A’; ch != ’Z’; ch++)
{
     /* ... */
}
```

For example:

```
; ch = ’A’
    MOVLZ    ’A’,R0       ; Use R0 as ’ch’
    MOVLZ    ’Z’,R1
ForLoop
; ch != ’Z’
     CMP.B   R1,R0
     BEQ     ForEnd       ; Branch if equal to zero
; ...
; ch++
     ADD.B   $1,R0        ; Next character
     BRA     ForLoop
ForEnd
```

13.2 Data structures

In addition to shorts and chars, C supports data structures such as arrays and structures.

13.2.1 Arrays

An array is a contiguous block of memory reserved at compile-time or assembly time. In a high-level language, the array is usually associated with a type, whereas within the ISA, type is not specified and must be enforced by instructions. For example, an array of short integers and an array of characters could be declared as:

```
short iarray[5];
char carray[6];
```

These can be implemented as uninitialized blocks:

```
iarray   bss  $10  ; 10 bytes (5 shorts)
carray   bss  $5   ; 5 bytes (5 characters))
```

Accessing the array can be done using an index or subscript; for example:

```
iarray[3] = 0;
carray[0] = ’x’;
carray[1] = ‘y’;
```

Can be implemented using indexed addressing:

```
    movl     iarray,R0
    movh     iarray,R0
    movlz    $0,R1
; mem[R0+6] = 0
    str      R1,R0,$6 ; 6th byte or 3rd word
```

If the character array (byte-addressed) is packed, bytes are stored contiguously, in even and odd bytes. This means the register should increment by 1 to move to the next byte in the array:

```
    movl     carray,R3    ; R3 = address of carray
    movh     carray,R3
    movlz    'x',R2
    st.b     R2,R3        ; mem[R3 + 0] = R2 (’x’)
    add      $1,R3        ; R3++
    movlz    'y',R2
    st.b     R2,R3        ; mem[R3 + 1] = ’y’
```

When accessing a byte element, it is necessary to append the ‘.b’ to the assembly instruction for the assembler to generate the correct machine instruction. There are no bound checks when using indexed addressing; ensuring that the subscripts are within the limits of the array is the responsibility of the software designer.

13.2.2 Switch-Case implementation using arrays

An array can also hold the address of an instruction, allowing, for example, the choice of case-labels at run-time in a switch statement.

The follow switch statement and its associated cases can be implemented in XM-23 assembler using an array:

```
switch(x)
{
case 10: /* x = 10 */
    result = ’A’;
break;
case 11:
case 15: /* x = 11 or 15 */
    result = ’B’;
break;
case 12: /* x = 12 */
    result = ’C’;
break;
default: /* All other values of x */
    result = ’D’;
}
```

A generic switch-handling subroutine can be designed to take the switch expression (‘x’ in this example), the list of case values, and the address of block of statements associated with each case label.

The switch expression falls into a range, from a lower-bound (the lowest case value) to an upper bound (the highest case value). There should be case-labels associated with case value between the upper- and lower-bound; however, if there aren’t, and for those values outside the lower-upper bound range, there is a default case value. By storing this information in a table, a function can be written to determine the intended case-address. The case-table structure is show in Table 20.

Table 20: Case-table entries

Field	Offset
Lower bound	+0
Upper bound	+2
Default address	+4
First case address	+6
…	
Last case address	

In the example, the switch-statement is implemented as follows:

```
;
    org      #1100
Case10
    ; Case 10 instructions
    movlz    'A',R2
    bra      CaseExit ; break
;
    org      #1200
Case11
Case15
    ; Case 11 and 15 instructions
    movlz    'B',R2
    bra      CaseExit ; break
;
    org  #1300
Case12
    ; Case 12 instructions
    movlz    'C',R2
    bra      CaseExit ; break
;
    org  #1400
Default
    ; Default instructions
    movlz    'D',R2
    ; End of switch, next instruction follows '}'
;
CaseExit
```

The case-table can be built from the above; it is specific to this switch-case statement, other switch-case statements would have their own tables:

```
CaseTable
    word     $10      ; Lower bound         (+0)
    word     $16      ; Upper bound+1       (+2)
    word     Default  ; Default address     (+4)
; List of case label addresses
    word     Case10   ; 10                  (+6)
    word     Case11   ; 11
    word     Case12   ; 12
    word     Default  ; 13
```

```
     word     Default  ; 14
     word     Case15   ; 15
     word     Default  ; Catch-all
```

Since cases 13 and 14 are not supplied, they are treated as default.

The generic subroutine for determining the target case address, CaseOffset, is listed below. It is the type of code that a compiler would generate when required to handle a switch-case statement.

CaseOffset two parameters, the expressions (R0) and the address of the case-table (R1). R1 is returned as the address of the case- or default-block to be executed. The subroutine first checks whether R0 is within the lower- and upper-bound range. If it isn't, the default address is returned. If it is within range, R0 is normalized (0 through upper-bound minus lower-bound), turned into a word address (doubling its value), and added to the address of the case-label addresses (offset 6). This entry can then be read and returned in R1 as the address of the case-block to execute.

```
;
; Equated values for access to case table - note these are all
; offsets into CaseTable (i.e., relative addresses)
;
LB   equ  $0   ; Offset for lower bound
UB   equ  $2   ; Offset for upper bound
DEF  equ  $4   ; Offset for default (last in list)
LIST equ  $6   ; First label in list
;
; Entry point to generic Case offset calculator
CaseOffset
;
; R0 is expression
; R1 is address of case table structure
; R2 is scratch register (undefined after call)
; Return R1 as address of case label
;
; 1. Compare with lower and upper limits
; Check lower bound
;
    ldr  R1,LB,R2 ; R2 = mem[R1+LB] (lower bound)
    cmp  R2,R0        ; tmp = R0 - R2 *** Note: DST (R0) - SRC (R2)
    ; R0 < LowerBound - out of range
    blt  CaseOffOOR
    ldr  R1,UB,R2     ; R2 = mem[R1+UB] (upper bound)
    cmp  R2,R0        ; tmp = R0 - R2 *** Note: DST (R0) - SRC (R2)
    blt  CaseOffIR    ; R0 < UpperBound - in range; else out-of range
;
; 2. Case offset is out of range use default
;
CaseOffOOR
    ldr  R1,DEF,R1    ; R1 = mem[R1+DEF]
    mov  LR,PC        ; return R1 as Default case address
;
CaseOffIR
;
; 3. Case offset is in range: R0 >= LB and R0 < UB (10..15)
```

```
;
; 'x' (R0) is between LB (10) and UB (15)
; Normalize R0 to 0 by subtracting it from the LB (10), giving 0..5
;
    ldr R1,LB,R2 ; R2 = mem[R1+LB] (lower bound)
    sub R2,R0         ; R0 = R0 - R2
;
; R0's value is 0..5
; Multiply R0 by 2 (shift left by 1) to get word offset
; (0, 2, 4, 6, 8, or 10). Adding R0 to itself is functionally
; equivalent to multiplying by 2
;
    add.w    R0,R0
;
; Pass control to specified address:
; a) Get base address of Case labels (LIST) from the CaseTable (R1)
;
    movlz    LIST,R2  ; R2 = LIST (#6)
    add      R2,R1    ; R1 = R1 (address of CaseTable) + R2 (LIST)
;
; b) Add R0 (x * 2) to the base address (R1) to get address of
; target Case
;
    add      R0,R1    ; R1 = R1 + R0
;
; c) R1 is the address of the location holding the Case label address
;
    ld       R1,R1    ; R1 = [R1]
;
; d) Return target Case label address in R1
;
    mov      LR,PC    ; Return
;
```

In the example, CaseOffset is called as follows:

```
; R0 contains 'x', the value of the switch expression
    movl     CaseTable,R1 ; R1 points to CaseTable
    movh     CaseTable,R1
    bl       CaseOffset   ; call CaseOffset()
```

When control return from CaseOffset, R1 has the address of the case-block to be executed. Control passes to the block by moving R1 into the PC:

```
    mov R1,PC         ; PC = R1 -- Control passed to address in R1
```

The case-block is now executed.

13.2.3 The C structure (struct)

A structure, such as a C struct, is used to aggregate a number of characteristics associated with an entity. The struct can hold any fundamental type (such as a char or int), arrays, and other structs. Consider the following example:

```
struct example
```

```
{
short a;
short b;
char c[5];
};
```

This structure is now a new data type, it does not occupy storage. It has three fields, two short integers, ‘a’ and ‘b’, and an array of five characters, ‘str’. A variable of type struct example can be declared:

```
struct example ex;
```

‘ex’ now occupies memory; the fields, their locations, and sizes are show in Table 21.

Table 21: Field locations of structure example

Offset	Field	Type	Size (bytes)
+0	a	short	2
+1			
+2	b	short	2
+3			
+4	c[0]	char	1
+5	c[1]	char	1
+6	c[2]	char	1
+7	c[3]	char	1
+8	c[4]	char	1

The structure, `ex`, of type `example`, can be initialized; for example:

```
  ex.a = 10;
  ex.b = 13;
  ex.c[0] = ’a’;
  ex.c[1] = ’b’;
  ex.c[2] = NUL;
```

For the structure, field ‘a’ is offset 0, field ‘b’ is offset 2, and field ‘c’ has offset 4. Using R0 to access the structure ‘ex’:

```
  ; R0 contains address of ex
    MOVL     ex,R0
    MOVH     ex,R0
  ; ex.a = 10
    MOVLZ    $10,R1
    ST.W     R1,R0+   ; [R0] = 10; R0 is address of ex, R0=R0+2
  ; ex.b = 13
    MOVLZ    $13,R1
    ST.W     R1,RO+   ; [R0] = 13; R0 is address of ex+offset ’b’
                      ; R0 = R0 + 2
  ;
  ; R0 now points to ex + 4
  ; ex.c[0] = ’a’
  ;
    MOVLZ    ’a’,R1
    ST.B     R1,R0+   ; R0 = R0 + 1
  ;
```

```
; ex.c[1] = ’b’
;
  ADD.B   $1,R1         ; Incr ’a’ to ’b’
  ST.B    R1,R0+   ; R0 = R0 + 1
;
; ex.c[2] = NUL
;
  MOVLZ   NUL,R1
  ST.B    R1,R0
```

Alternatively, STR can be used, specifying an offset into the structure, thereby avoiding the need to increment the base address of the structure:

```
; R0 contains address of ex
  MOVL    ex,R0
  MOVH    ex,R0
; ex.a = 10
  MOVLZ   $10,R1
  STR.W   R1,R0,$0 ; [R0 + 0] = 10, or field ’a’
; ex.b = 13
  MOVLZ   $13,R1
  STR.W   R1,RO,$2 ; [R0 + 2] = 13, where R0+2 is field ’b’
; ex.c[0] = ’a’
  MOVLZ   ’a’,R1
  STR.B   R1,R0,$4 ; [R0 + 4] = ’a’, where R0+4 is c[0]
; ex.c[1] = ’b’
  MOVL    ’b’,R1
  STR.B   R1,R0,$5 ; [R0 + 5] = ’b’, where R0+5 is c[1]
; ex.c[2] = NUL
  MOVLZ   NUL,R1
  STR.B   R1,R0,$6 ; [R0 + 6] = NUL, where R0+6 is c[2]b
```

A structure can be passed to a subroutine as an parameter. If it is passed by-value, the data in the original structure is copied into a corresponding structure in the subroutine. This can be done field-by-field or simply as a byte-copy. Alternatively, the structure can be passed by-reference, as the address of the original structure.

Structures can be organized as arrays. For example, an array of five example structures (above) could be declared as follows:

```
struct example ex_array[5];
```

However, there is a problem with this array as becomes evident when looking at the first two array elements, ex_array[0] and ex_array[1]:

Offset	Contents	Type
+0	ex_array[0].a	short
+1		
+2	ex_array[0].b	short
+3		
+4	ex_array[0].c[0]	char
+5	ex_array[0].c[1]	char
+6	ex_array[0].c[2]	char
+7	ex_array[0].c[3]	char
+8	ex_array[0].c[4]	char
+9	ex_array[1].a	short
+10		
+11	ex_array[1].b	short
+12		
+13	ex_array[1].c[0]	char
+14	ex_array[1].c[1]	char
+15	ex_array[1].c[2]	char
+16	ex_array[1].c[3]	char
+17	ex_array[1].c[4]	char

The second element of the array (ex_array[1]) begins on an odd-byte boundary (offset+9). This means that it will not be possible to access ex_array[1].a (a short) since it must fall on an even-byte boundary. Solving this problem is straight-forward, a padding byte is required in `struct example` to ensure that the next array element begins on an even-byte boundary (this increases the size of the structure from 9 to 10 bytes):

```
struct example
{
short a;
short b;
char c[5];
char pad;
};
```

Alternatively, a compiler could be designed to recognize that the structure was only 9 bytes long, so it could insert its own (hidden) padding byte. If the structure was written in assembler language, the programmer could use a directive such as ALIGN to signal the assembler to start the structure on an even-byte boundary.

The padding byte does not need to be initialized as all it is doing is increasing the number of bytes in the structure:

Offset	Contents	Type
+0	ex_array[0].a	short
+1		
+2	ex_array[0].b	short
+3		
+4	ex_array[0].c[0]	char
+5	ex_array[0].c[1]	char
+6	ex_array[0].c[2]	char
+7	ex_array[0].c[3]	char
+8	ex_array[0].c[4]	char
+9	**ex_array[0].pad**	**char**
+10	ex_array[1].a	short
+11		
+12	ex_array[1].b	short
+13		
+14	ex_array[1].c[0]	char
+15	ex_array[1].c[1]	char
+16	ex_array[1].c[2]	char
+17	ex_array[1].c[3]	char
+18	ex_array[1].c[4]	char
+19	**ex_array[1].pad**	**char**
+20	ex_array[2].a	short
+21	…	

Padding bytes are typically added by the compiler; however, when programming in assembler, it may be necessary for the programmer to be aware of the limitation. If the programmer is writing assembly code to be linked with a high-level language using padding bytes, the assembly code would probably be required to follow the standard.

Traversing an array of structures requires knowledge of the size of the structure and then incrementing the index by that amount, for example:

```
  MOVL    ex_array,R3
  MOVH    ex_array,R3
Loop
; ...
; Increment R3 by 10 to get next array element
; Add 2 and then add 8 to get 10:
;
  ADD     $2,R3
  ADD     $8,R3
  BRA     Loop
```

It is slightly more complex if the index is supplied as, for example, an parameter to a subroutine, and from this, the correct element is to be found. In this case, the index must be multiplied by 10 (the size of `struct example`) and then added to the base address. If the index is supplied in R3, it is first necessary to multiply R3 by 10, this can be done by a series of three left shifts (each shift multiplies the index by 2), and then adding the resulting offset to the base address of the array:

```
; R3 is index
    ADD      R3,R3    ; R3 * 2
    MOV      R3,R4    ; R4 = index * 2
    ADD      R3,R3    ; R3 * 4
    ADD      R3,R3    ; R3 * 8
    ADD      R3,R4    ; R4 = index * 2 + index * 8
; Address of element is base address + offset:
    MOVL     ex_array,R0
    MOVH     ex_array,R0
;
    ADD      R0,R4    ; R4 = address of ex_array + offset into array
```

13.2.4 Pointers

A pointer holds the address of a data structure (that is, it "points" to the data). The address of the data structure can be statically allocated at compile-time or assembly-time or dynamically allocated at run-time. Although a pointer is an address, inside the XM-23 CPU it is stored in a register, allowing it to be manipulated like data.

The address of a data structure can be obtained at assembly-time using immediate-mode addressing, for example:

```
; Static data are associated with a fixed address
; In this case, "alpha" is at location #1000
;
    ORG  #1000
alpha    bss $2       ; Reserve two bytes for alpha
;
    ORG  #2800
;
; R0 = address of alpha (#1000)
;
    MOVL     alpha,R0
    MOVH     alpha,R0
```

Register R0 now contains the address of the structure `alpha`. It can be accessing using register-direct addressing, for example:

```
     MOVLS   $-1,R1   ; R1 = -1 (#FFFF)
     ST      R1,R0    ; R1 (-1) stored in alpha
     LD      R0,R2    ; R2 is assigned the contents of alpha
```

The above example used a static address. A pointer can be assigned a dynamic value in several ways, including:

- Copying a by-reference address from the stack in a subroutine call, since the address can vary each time the subroutine is called:

```
; R0 = address of alpha (#1000)
    MOVL alpha,R0
    MOVH alpha,R0
;
; Push R0 onto top-of-stack
; Top-of-stack contains the value of R0 (address of alpha)
; SP points to top-of-stack
```

```
;
     ST      R0,-SP   ; Push R3 onto stack
     BL      Subr     ; Call Subr
     ADD     $2,SP    ; Restore top of stack
; ...
Subr
     LD      SP,R2    ; Top of stack moved to R2
```

- Assigning the address of a structure allocated at run-time from the heap; for example, after a call to malloc():

  ```
  ptr = malloc(10)
  *ptr++ = 0 */ Assume *ptr points to a 16-bit short */
  *ptr = 1;
  ```

  Implemented as:

```
     MOVLZ   $10,R0
     BL      malloc   ; Assume R0 indicates number of bytes to allocate
                      ; R0 returns with the address supplied by
malloc()
; ...
     MOVLZ   $0,R1    ; R1 = 0
     ST      R1,R0    ; *ptr = 0
     ADD     $2,R0    ; ptr++; /* assume short – 2 byte increment */
     MOVLZ   $1,R1    ; R1 = 1
     ST      R1,R0    ; *ptr = 1
```

Functions such as strlen() can be implemented using a pointer:

```
strlen
    st       R1,-SP   ; Assume R1 points to NUL-terminated string
    movlz    $0,R0    ; Return length in R0 (initially zero)
;
strlen1
    ld.b     R1+,R2   ; Copy byte [R1] to R2
                      ; R1 incremented by 1
    cmp.b    NUL,R2   ; Compare byte (in R2) with NUL
    beq      strlen2  ; If equal, go to strlen2
    add      $1,R0    ; R0 (length) incremented by 1
    bra      strlen1  ; Check next byte
;
strlen2
    ld       SP+,R1   ; Restore R1
    mov      LR,PC    ; Return to caller.  R0 has length
```

13.2.5 Linked lists

Linked data structures can be supported with XM-23's registers. For example, consider a structure and link-list traversing code such as:

```
struct link_ex
{
short data1;      /* Offset +0 */
short data2;      /* Offset +2 */
struct link_ex *next; /* Offset +4 */
};
...
void print_entries(struct link_ex *ptr)
{
while (ptr != NULL)
{
  print_data(ptr -> data1, ptr -> data2);
  ptr = ptr -> next;
}
}
```

The function print_entries() can be implemented as follows:

```
print_entries
;
; Prints data1 and data2 from each entry in the linked list
; Register R0 is the pointer to the list
;
; Save volatile registers on stack:
;
    st  R0,-SP
    st  R1,-SP  ; For data1
    st  R2,-SP  ; For data2
    st  LR,-SP  ; Preserve LR for return
;
; while (ptr != NULL)
;
While
    cmp $0,R0
    beq EndWhile
;
    ldr R0,$0,R1     ; R1 = mem[R0 + 0] data1
    ldr R0,$2,R2     ; R2 = mem[R0 + 2] data2
    bl  print_data
;
    ldr R0,$4,R0     ; R0 = mem[R0 + 4] *next
;
    bra While
;
EndWhile
;
; Restore volatile registers
;
    ld  SP+,LR
    ld  SP+,R2
    ld  SP+,R1
    ld  SP+,R0
```

```
;
; Return
;
    mov  LR,PC
```

14 XM-23 Instruction Set

Table 22: Bit value definitions for XM-23 Instruction Set (Table 25)

0	1	Instruction opcode bit values (0 or 1).
PRPO		Pre- or post-increment or pre- or post-decrement (Load and Store).
DEC		Decrement the register (before or after the instruction is executed).
INC		Increment the register (before or after the instruction is executed).
W/B		Word (16-bits) or byte (8-bits) addressing or register size.
R/C		Register (0) or Constant (1).
S		Source register bit (one of 3).
D		Destination register bit (one of 3).
B		Bit (one of 8) in MOVL, MOVLZ, MOVLS, and MOVH instructions.
OFF		A bit used in an offset (in LDR, STR, and branching instructions).
S/C		Source register or constant value (see Table 23)
SA		SVC (Service Call) vector address (#0 through #F).
C		Conditional execution code (#0 to #E)
T		THEN (True) count (#0 to #7)
F		ELSE (False) count (#0 to #7)
V, SLP, N, Z, C		Condition code values (oVerflow, Sleep, Negative, Zero, and Carry).

Table 23: Register and Constant values for R/C and SRC bits

R/C		SRC
0	1	Encoding
Register	Constant	(bits 3-5)
R0	0	000
R1	1	001
R2	2	010
R3	4	011
R4	8	100
R5/LR	16	101
R6/SP	32	110
R7/PC	-1	111

Table 24: Conditional execution codes and descriptions for CEX instruction

Code	Description	True state	Code	Description	True state
0000	Equal / equals zero	Z	1000	Unsigned higher	C and !Z
0001	Not equal	!Z	1001	Unsigned lower or same	!C or Z
0010	Carry set / unsigned higher or same	C	1010	Signed greater than or equal	N == V
0011	Carry clear / unsigned lower	!C	1011	Signed less than	N != V
0100	Minus / negative	N	1100	Signed greater than	!Z and (N == V)
0101	Plus / positive or zero	!N	1101	Signed less than or equal	Z or (N != V)
0110	Overflow	V	1110	True	any
0111	No overflow	!V	1111	False	any

Table 25: XM-23 Instruction Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Instruction
0	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BL	Branch with Link
0	0	1	0	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BEQ/BZ	Branch if equal or zero
0	0	1	0	0	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BNE/BNZ	Branch if not equal or not zero
0	0	1	0	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BC/BHS	Branch if carry/higher or same (unsigned)
0	0	1	0	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BNC/BLO	Branch if no carry/lower (unsigned)
0	0	1	1	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BN	Branch if negative
0	0	1	1	0	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BGE	Branch if greater or equal (signed)
0	0	1	1	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BLT	Branch if less (signed)
0	0	1	1	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BRA	Branch Always
0	1	0	0	0	0	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	ADD	Add: DST ← DST + SRC/CON
0	1	0	0	0	0	0	1	R/C	W/B	S/C	S/C	S/C	D	D	D	ADDC	Add: DST ← DST + (SRC/CON + Carry)
0	1	0	0	0	0	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	SUB	Subtract: DST ← DST + (¬SRC/CON + 1)
0	1	0	0	0	0	1	1	R/C	W/B	S/C	S/C	S/C	D	D	D	SUBC	Subtract: DST ← DST + (¬SRC/CON + Carry)
0	1	0	0	0	1	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	DADD	Decimal add: DST ← DST + (SRC/CON + Carry)
0	1	0	0	0	1	0	1	R/C	W/B	S/C	S/C	S/C	D	D	D	CMP	Compare: DST – SRC/CON
0	1	0	0	0	1	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	XOR	Exclusive OR: DST ← DST ⊕ SRC/CON
0	1	0	0	0	1	1	1	R/C	W/B	S/C	S/C	S/C	D	D	D	AND	AND: DST ← DST & SRC/CON
0	1	0	0	1	0	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	OR	OR: DST ← DST | SRC/CON
0	1	0	0	1	0	0	1	R/C	W/B	S/C	S/C	S/C	D	D	D	BIT	Bit test: DST & (1 << SCR/CON)
0	1	0	0	1	0	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	BIC	Bit clear: DST ← DST & ~(1 << SRC/CON)
0	1	0	0	1	0	1	1	R/C	W/B	S/C	S/C	S/C	D	D	D	BIS	Bit set: DST ← DST | (1 << SRC/CON)
0	1	0	0	1	1	0	0	0	W/B	S	S	S	D	D	D	MOV	DST ← SRC
0	1	0	0	1	1	0	0	1	0	S	S	S	D	D	D	SWAP	Swap SRC and DST
0	1	0	0	1	1	0	1	0	W/B	0	0	0	D	D	D	SRA	Shift DDD right (1 bit) arithmetic
0	1	0	0	1	1	0	1	0	W/B	0	0	1	D	D	D	RRC	Rotate DDD right (1 bit) through carry
0	1	0	0	1	1	0	1	0	W/B	0	1	0	D	D	D	COMP	Ones' complement of DDD
0	1	0	0	1	1	0	1	0	0	0	1	1	D	D	D	SWPB	Swap bytes in DDD
0	1	0	0	1	1	0	1	0	0	1	0	0	D	D	D	SXT	Sign-extend byte to word in DDD
0	1	0	0	1	1	0	1	1	0	0	0	0	PR	PR	PR	SETPRI	Set current priority
0	1	0	0	1	1	0	1	1	0	1	0	SA	SA	SA	SA	SVC	Control passes to address specified in vector[SA]
0	1	0	0	1	1	0	1	1	1	0	V	SLP	N	Z	C	SETCC	Set PSW bits (1 = set)
0	1	0	0	1	1	0	1	1	1	1	V	SLP	N	Z	C	CLRCC	Clear PSW bits (1 = clear)
0	1	0	1	0	0	C	C	C	C	T	T	T	F	F	F	CEX	Conditional execution
0	1	0	1	1	0	PRPO	DEC	INC	W/B	S	S	S	D	D	D	LD	DST ← mem[SRC plus addressing]
0	1	0	1	1	1	PRPO	DEC	INC	W/B	S	S	S	D	D	D	ST	mem[DST plus addressing] = SRC
0	1	1	0	0	B	B	B	B	B	B	B	B	D	D	D	MOVL	DST.Low byte ← BBBBBBBB; DST.High byte unchanged
0	1	1	0	1	B	B	B	B	B	B	B	B	D	D	D	MOVLZ	DST.Low byte ← BBBBBBBB; DST.High byte ← 00000000
0	1	1	1	0	B	B	B	B	B	B	B	B	D	D	D	MOVLS	DST.Low byte ← BBBBBBBB; DST.High byte ← 11111111
0	1	1	1	1	B	B	B	B	B	B	B	B	D	D	D	MOVH	DST.Low byte unchanged; DST.High byte ← BBBBBBBB
1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	W/B	S	S	S	D	D	D	LDR	DST ← mem[SRC + sign-extended 7-bit offset]
1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	W/B	S	S	S	D	D	D	STR	mem[DST + sign-extended 7-bit offset] ← SRC

15 Index