The X-Makina Instruction Set Architecture

Larry Hughes, PhD

24 May 2018 (rev. 1.03)[1]

[1] The author would like to thank Gary Hubley and Mrs. H. for their assistance with the original text and the ECED 3403 students who have made suggestions to correct and improve the text.

Table of Contents

1 Introduction

X-Makina is a 16-bit Instruction Set Architecture (ISA) with the following features:[2]

- 33 instructions (memory access, arithmetic, logic, and control), all 16-bits in width. Many instructions can operate on a byte, a word, or both.
- An additional 32 instructions that can be emulated by using existing instructions, allowing operations such as subroutine return, interrupt return, and stack push and pull.
- A common instruction format, containing an opcode and one or two operand addresses (for arithmetic and logical instructions) or a signed 10-bit offset (for branching instructions).
- Eight 16-bit registers: four special purpose (program counter, stack pointer, program status word, and link register) and four general purpose (for data, addressing, or both).
- Four addressing modes (register, register with pre- and post- auto-increment and auto-decrement, register-relative, and immediate).
- A 16-bit data/address bus, allowing up to 64 KiB of random-access memory.

The ISA exhibits many features found in existing commercial load-store or RISC (reduced instruction set computer) machines.

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 considered signed, 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 '$' denotes a hexadecimal number.

2 Central Processing Unit

The Central Processing Unit (or CPU) is responsible for fetching, decoding, and executing instructions from memory. The program counter (or PC) contains the address of either the next

[2] If you're *really* interested, X-Makina is a tip-of-the hat to the film *Ex Machina* (Alex Garland, 2014). Since "X-Machine" is too similar to other projects, I opted for "makina". A google-search revealed that "makina" was (is?) Spanish rave music from the 1990s. It was, by chance, part of the rave-scene in Newcastle upon Tyne during this decade, where coincidentally, I did my PhD in the Computing Laboratory at the University of Newcastle upon Tyne (a bit before the 1990s). Hence, X-Makina.

instruction to be fetched or a value associated with the current instruction (an immediate data value or an address). Each instruction is held in the instruction register (inaccessible by software) and decoded by the instruction decoder. The decoder interprets the instruction and determines the steps required by the CPU to perform it.

The Arithmetic and Logic Unit performs arithmetic and logic operations on the contents or one or more registers. Register values are copied from the register file into the ALU. The destination of the result (for example, back to the register file) is determined by the instruction and its addressing mode. The CPU's state is changed by the ALU and recorded in the status register (PSW). The instruction decoder controls access to the memory bus, specifying which address is to be accessed and indicating whether the contents are to be read (instructions or data) or written (data).

X-Makina's ALU, register file, and internal buses are shown in Figure 1.

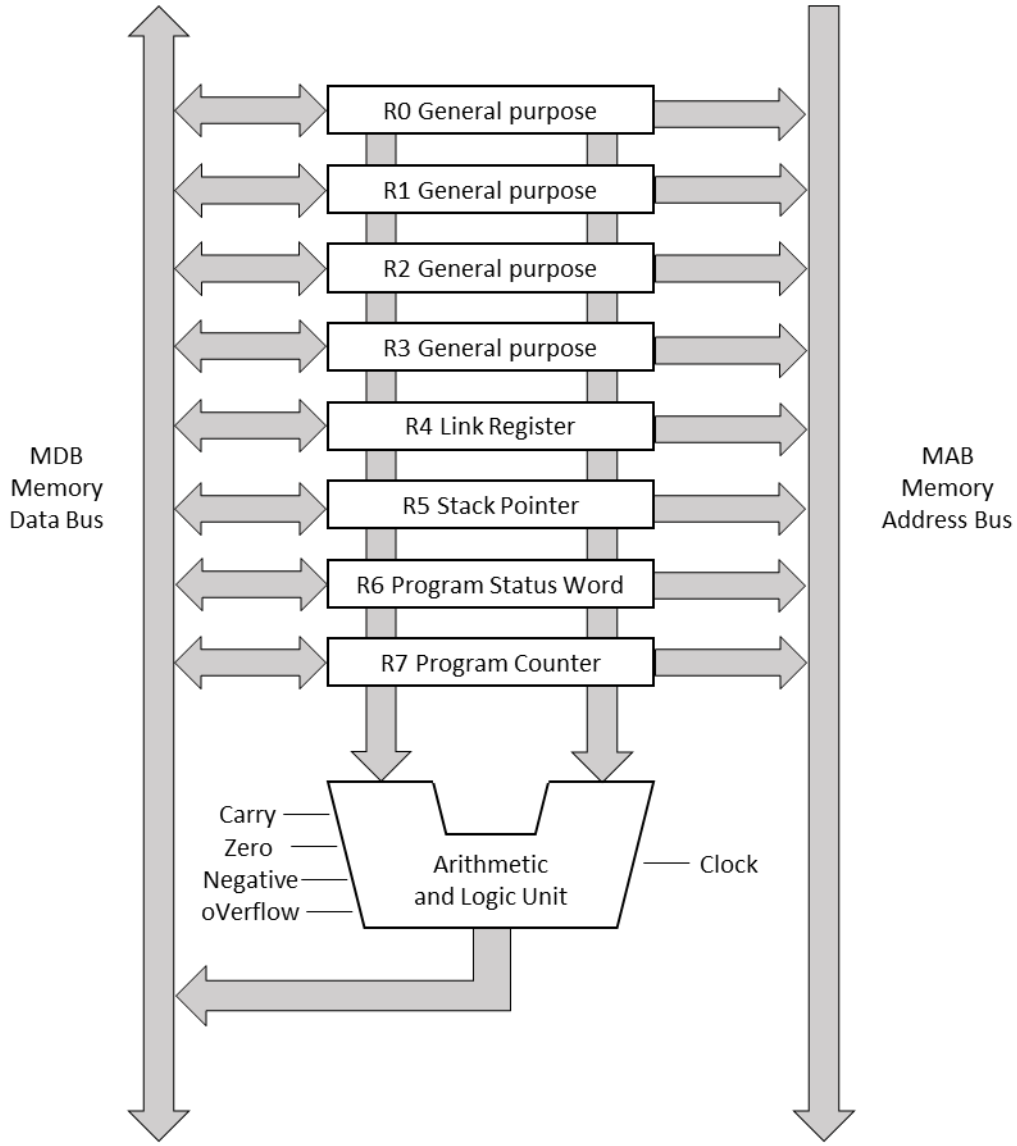


Figure 1: X-Makina's register file, ALU, and internal busses

3 Memory

Memory can be thought of as an array of bytes or words, with the lowest address (i.e., 0) is shown at the top of the page, while the highest (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 upward (towards $0000), and each pull (or pop) removing a value from the stack, moving it downward (towards $FFFF).

X-Makina supports the addressing of 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 2).

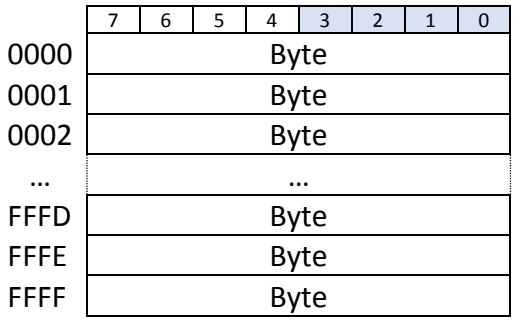


Figure 2: Byte memory-organization

3.2 Word organization

A word is a 16-bit quantity on X-Makina (a **signed** or **unsigned short**). Words must fall (that is, start) on even-byte boundaries; in other words, their addresses are always even (i.e., the least-significant bit of the address is always zero). Instructions and 16-bit integers are stored as words, as shown in Figure 3.

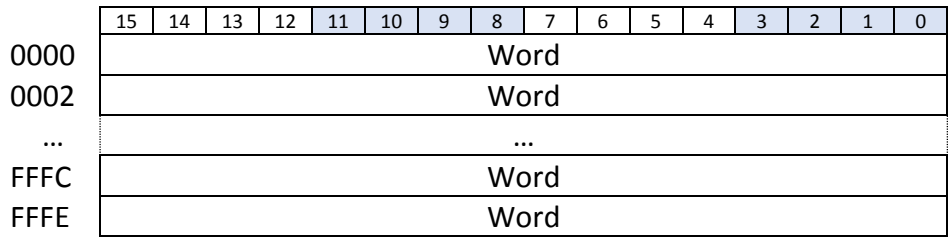


Figure 3: Word memory-organization

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, or most-significant, byte (bits 15 through 8) and a low, or least-significant, byte (bits 7 through 0). When a word is stored in memory, its high (MSB) and low (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 or big-endian byte-ordering is shown in Figure 4.

Figure 4: Storing $1234 in two different endian structures

X-Makina is a little-endian ISA, with the least significant byte in a word in the low-byte and the most significant byte in the high-byte (see Figure 5).

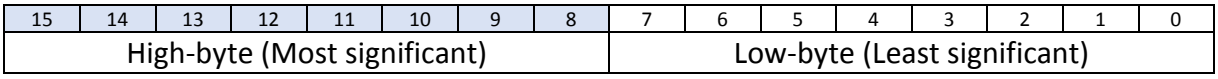


Figure 5: Little-endian byte-ordering

Figure 6 shows how $1234 would appear in a 16-bit word.

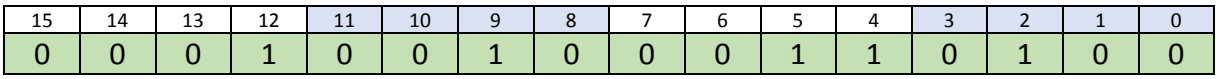


Figure 6: Little-endian representation of $1234 in a 16-bit word

Since the X-Makina 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 7 shows memory organized in terms of bytes, while Figure 8 shows memory organized as words.

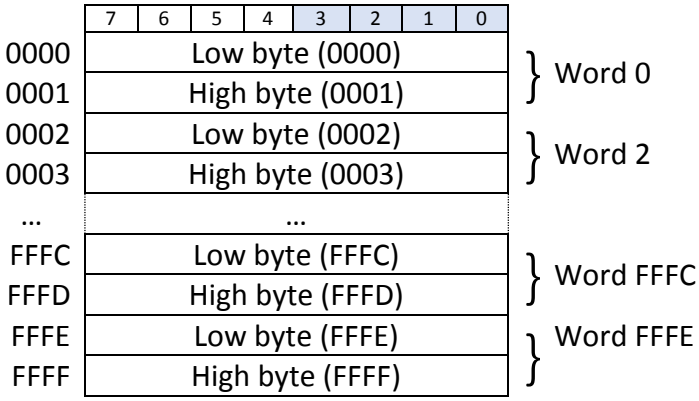


Figure 7: Byte organization of memory (Address in parenthesis)

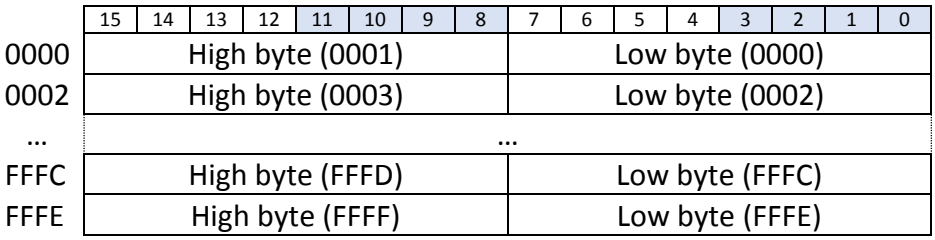


Figure 8: Word organization of memory (Address in parenthesis)

3.4 X-Makina's reserved memory

X-Makina has two reserved memory areas: device-mapped memory and interrupt vectors.

X-Makina 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). The registers occupy one word each and are stored contiguously, starting in location $0000.

There are 16 interrupt vectors (eight for interrupt-enabled devices and eight for faults and traps). The vectors occupy two words each, the low-word is the PSW of the handler associated with the vector and the high-word is the entry-point (address) of the handler. The first reserved location for the vectors is $FFE0.

The reserved memory map is shown in Figure 9.

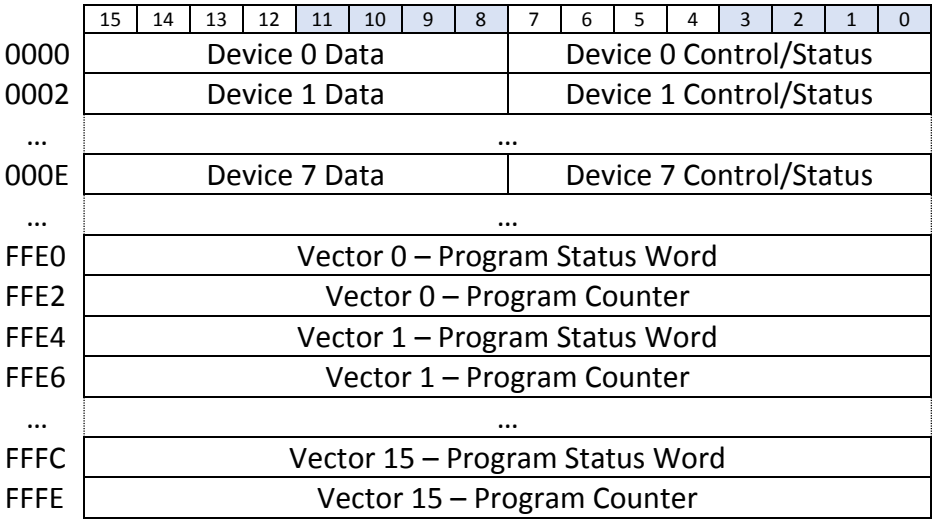


Figure 9: X-Makina's memory map showing reserved memory

4 Registers

X-Makina has eight 16-bit registers, their names and functions are shown in Table 1.

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

Name	Function
R0, R1, R2, R3	General Purpose
R4 (LR)	Link Register or General Purpose
R5 (SP)	Stack Pointer
R6 (PSW)	Program Status Word
R7 (PC)	Program Counter

4.1 General Purpose registers (R0-R3)

Four 16-bit registers that can be used for addressing or arithmetic and logic instructions. A register can hold signed or unsigned quantities.

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 Link register (R4 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 LR. The least-significant bit (bit 0) is always 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

- When an interrupt occurs, the CPU stores an invalid address ($FFFF) in the LR to indicate that an interrupt, fault, or trap handler is active. This is used by the CPU when returning from the interrupt (see section 9, Interrupts, faults, and traps).

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, LR can also be used as a general purpose register (R4). 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, LR should be pushed onto the stack. On return, the value of LR should be pulled to restore the original value.

4.3 Stack pointer (R5 or SP)

16-bit register, points to the value on the current top-of-stack. A pull reads this value and increments the SP, while a push decrements the SP and then writes a value, making it the new top-of-stack. The stack pointer always refers to an even-address instruction, therefore the least-significant bit (bit 0) is always cleared by the hardware.

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 can lead to unpredictable results.

4.4 Program Status Word (R6 or PSW)

The PSW contains the Carry, Zero, Negative, and oVerflow status bits, indicating the CPU status changes associated with the last instruction executed. It also holds the priority level (0 through 7) of the currently executing instruction (see section 9, Interrupts, faults, and traps).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	Priority			V	-	N	Z	C

The status bits are defined as follows:

C: Carry. Last operation, treated as unsigned, produced a value that exceeded 16-bits (word operation) or 8-bits (byte-operation). This is a carry (in addition) or a borrow (in subtraction)

Z: Zero. The last operation resulted in a zero value: $0000 (if word operation) or $00 (if byte operation).

N: Negative. The most significant bit is set (bit 15 if word operation or bit 7 if byte operation).

V: oVerflow. An arithmetic overflow occurred on a signed operation (8-bit or 16-bit only).

For addition information on the status bits, see section 7.1 (Sign, carry, and overflow bits). This register should not be used as a general purpose register.

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, therefore the hardware always clears the least-significant bit.

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 PC is equivalent to a JUMP instruction,[3] control will pass to the specified address. Note that there will be a one or two instruction delay if X-Makina is implemented as a pipeline processor.

5 Instructions

X-Makina's 33 instructions are described in this section.

5.1 Memory access

Memory access instructions allow a program to access data memory. Two registers are used, one register specifying the effective-address (EA) of a memory location to be read-from (in this case, the contents of the memory location are copied to the second register) or written-to (in this case, the contents of the second register are copied to memory location).

[3] X-Makina does not have a JUMP instruction; however, it can be emulated; see section 6. Branching, or transfer of control, is explained in section 5.3.

There are four memory access instructions.

5.1.1 Register direct and register direct with pre or post auto-increment or auto-decrement

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

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

The register used for memory-access can be modified using:

PRPO: Pre- or post- increment or decrement of the memory register specifying the 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”.

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

The possible address modifier combinations are listed in Table 2. 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 2: 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	*EA = Rn* *memory[EA]*	0	0	0
+Rn	Pre-increment register	*EA = Rn + 1 (byte) or* *EA = Rn + 2 (word)* *memory[EA]* *Rn = EA*	1	0	1
Rn+	Post-increment register	*EA = Rn* *memory[EA]* *Rn = Rn + 1 (byte) or* *Rn = Rn + 2 (word)*	0	0	1
-Rn	Pre-decrement the register	*EA = Rn - 1 (byte) or* *EA = Rn - 2 (word)* *memory[EA]* *Rn = EA*	1	1	0
Rn-	Post-decrement the register	*EA = Rn* *memory[EA]* *Rn = Rn - 1 (byte) or* *Rn = Rn - 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 3.

Table 3: Load and store register-direct and register-direct with pre- or post-auto-increment or auto-decrement

Instruction	Operation	Description	Opcode
LD(.B or .W) SRC,DST **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* *EA = SRC + address modifiers* *DST ← memory[EA]* *if Post-Incr or Post-Decr then* *EA = SRC + address modifiers*	Load a register (DST) from memory location specified by the effective address (EA). Reading a byte stores the value in the low-byte of the DST register; the high-byte is unchanged.	1.0000
ST(.B or .W) SRC,DST **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* *EA = DST + address modifiers* *memory [EA] ← SRC* *if Post-Incr or Post-Decr then* *EA = SRC + address modifiers*	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.	1.0001

The LD and ST instructions permit array accessing (e.g., 8-bit and 16-bit arrays)[4] as well as the creation of stacks and stack operators. For example, to copy 10 words from Array1 to Array2:

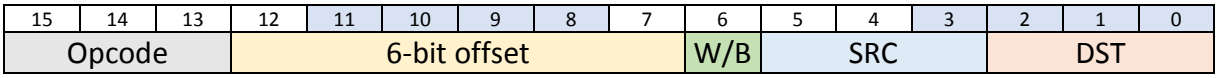
```
MOVL        Array1,R2   ; Ptr1 = &Array1
MOVH        Array1,R2
```

[4] Note: An 8-bit array of characters is a string.

```
        MOVL        Array2,R3   ; Ptr2 = &Array2
        MOVH        Array2,R3
        MOVLZ       R0,#10      ; Counter = 10
Loop    LD          R2+,R1      ; Data = *Ptr1++
        ST          R1,R3+      ; *Ptr2++ = Data
        SUB         #1,R0       ; Counter = Counter - 1
        BNZ         Loop        ; If Counter ≠ 0 Then goto Loop
```

5.1.2 Register relative

X-Makina also supports two load and store instructions, LDR and STR, which use register-relative addressing where the effective address is determined from the register value plus the value of a 6-bit signed offset stored in the instruction:



The 6-bit signed offset (bits 6 through 11) values are extracted and stored in bits 0 through 5 of an internal 16-bit register with the value of bit 5 being duplicated in bits 6 through 15 (i.e., the sign-bit is extended).[5] The signed, shifted 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. The offset range is the value of the SRC or DST register-32 through the value of the SRC or DST register+31.

Up to 64 bytes or 32 words can be addressed, with bytes starting on odd or even byte-boundaries and words on even-byte boundaries only (see Table 4).

Table 4: Relative addressing
Range of addressable memory: reg-32 through reg+31

Effective address	Byte offset	Word offset
reg-32	-32	-32
reg-31	-31	
...		
reg-2	-2	-2
reg-1	-1	
reg+0	+0	+0
reg+1	+1	
...		
reg+30	+30	+30
reg+31	+31	

The offset bits and their 16-bit signed equivalent are shown in Table 5. A word or a byte can be accessed by specifying 0 (word) or 1 (byte) in the W/B bit.

[5] The internal register is not accessible by software. It is internal to the CPU.

Table 5: Offset values

Offset values (bits 11..6)	Shifted (original 6-bits in red)	Signed decimal value
10.0000	1111.1111.1110.0000	-32
10.0001	1111.1111.1110.0001	-31
. . .		
11.1110	1111.1111.1111.1110	-2
11.1111	1111.1111.1111.1111	-1
00.0000	0000.0000.0000.0000	+0
00.0001	0000.0000.0000.0001	+1
00.0010	0000.0000.0000.0010	+2
. . .		
01.1110	0000.0000.0001.1110	+30
01.1111	0000.0000.0001.1111	+31

The two instructions are defined in Table 6. Using an offset of zero is equivalent to register direct without register modification.

Table 6: Load and store register-relative instructions[6]

Instruction	Operation	Description	Opcode
LDR(.B or .W) SRC,OFF,DST	*EA = SRC + signed offset* *DST ← memory[EA]*	Load a register (DST) from memory location specified by the effective address (EA).	0100
STR(.B or .W) SRC,DST,OFF	*EA = DST + signed offset* *memory [EA] ← SRC*	Store a register (SRC) in memory location specified by the effective address.	0101

Register-relative allows accessing of subroutine arguments and fields in a structure. For example, to copy the third argument of a subroutine call to the subroutine's second automatic variable (assuming R3 is the Base or Frame Pointer, see section 8, Subroutines and arguments, for more details on accessing a stack during a subroutine call):

```
LDR   R3,#6,R0    ; R0 = mem[R3 + 6]
STR   R0,R3,#-4   ; mem[R3 - 4] = R0
```

5.2 Register initialization instructions

In addition to the load instructions, registers can be assigned initial 8-bit values using three different instructions: move-low (MOVL), move-low and zero high-byte (MOVLZ), and move-high (MOVH). All instructions have the same format, shown in Figure 10.

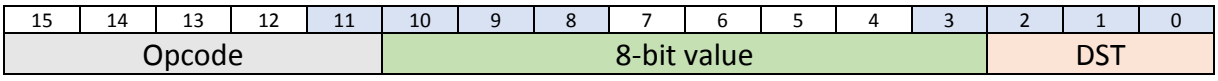


Figure 10: Register initialization format

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 7.

[6] OFF (offset) can be a numeric value or a label. For rules regarding numerics and labels, see the X-Makina Assembler User's Guide.

Table 7: Register initialization instructions
(LSB – least-significant byte; MSB – most-significant byte)[7]

Instruction	Operation	Description	Opcode
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.	1.0010
MOVLZ Value,DST	*DST.LSB ← Value* *DST.MSB ← 0*	Value is assigned to the low-byte (LSB) of the destination register. The high-byte (MSB) is zeroed.	1.0011
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.	1.0100

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

```
MOVLZ       #0,R3
```

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:

```
MOVL   Array,R0   ; Least-significant byte of Array assigned to R0
MOVH   Array,R0   ; Most-significant byte of Array assigned to R0
```

The above example assumes that the assembler can extract the lower 8-bits of Value for MOVL and MOVLZ and the higher 8-bits for MOVH.

5.3 Transfer of control (branching)

A branch instruction can change the value of the program counter relative to the address of the instruction to change the flow of control. X-Makina supports nine branching instructions, seven of which are conditional. The target address is obtained by adding the value of the PC to the instruction's signed-extended offset: 13-bits ($0000 to $1FFF; bit 12 is the sign bit) for branch with link and 10-bits ($000 through $3FF; bit 9 is the sign bit) for all other branching instructions. There are two instruction formats, one for subroutine calls (Figure 11, top) and one for conditional and unconditional branching (Figure 11, bottom).

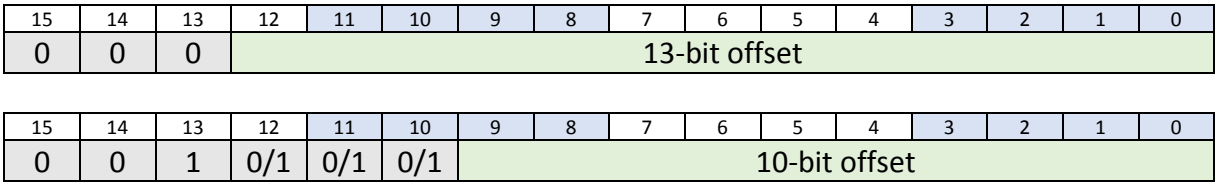


Figure 11: Transfer of control instruction formats
Top: 13-bit offset for BL instruction
Bottom: 10-bit offset for branching instructions other than BL

[7] Value can be a numeric value or a label. For rules regarding numerics and labels, see the X-Makina Assembler User's Guide.

The branching instructions are list in Table 8.

Table 8: Branching instructions
(*label* refers to the sign-extended value)[8]

Instruction	Operation	Description	Opcode	
			15..13	12..10
BL label	*LR ← PC+2* *PC ← PC+ (label × 2)*	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 section 9 (Interrupts, faults, and traps) for additional information on LR.	000	Offset
BEQ label **BZ label**	*PC ← PSW.Z = 1 ? PC+ (label × 2) : PC+2*	Branch to label if zero flag is set	001	000
BNE label **BNZ label**	*PC ← PSW.Z = 0 ? PC+ (label × 2) : PC+2*	Branch to label if zero flag is cleared	001	001
BC label	*PC ← PSW.C = 1 ? PC+ (label × 2) : PC+2*	Branch to label if carry flag is set	001	010
BNC label	*PC ← PSW.C = 0 ? PC+ (label × 2) : PC+2*	Branch to label if carry flag is cleared	001	011
BN label	*PC ← PSW.N = 1 ? PC+ (label × 2) : PC+2*	Branch to label if negative flag is set	001	100
BGE label	*PC ← PSW.N ⊕ PSW.V = 0 ? PC+ (label × 2) : PC+2*	Branch to label if greater than or equal	001	101
BLT label	*PC ← PSW.N ⊕ PSW.V = 1 ? PC+ (label × 2) : PC+2*	Branch to label if less than	001	110
BAL label	*PC ← PC+ (label × 2)*	Branch always (unconditional) to label	001	111

The new address of the PC is determined as follows:

1. The signed-offset (13-bits or 10-bits) is extracted from the instruction (see Figure 11). The opcode field is ignored (see Figure 12).

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

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

Figure 12: Extracted offset bits
Top: 13-bits offset (BL instruction); Bottom: 10-bit offset (All other branching instructions)

2. The offset is shifted right by one, feeding a zero into the least-significant bit, thereby ensuring the value is even (see Figure 13).

[8] See the X-Makina Assembler User's Guide for rules regarding branching labels.

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

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

Figure 13: Converting to even address
Top: 13-bit offset shifted to 14 bits (BL instruction);
Bottom: 10-bit offset shifter to 11 bits (All other branching instructions)

3. The value in the sign-bit (bit 13 or bit 10) is now signed-extended, resulting in a positive or negative 16-bit word.

   a) The 13-bit offset used with BL has a positive value in the range of +0 ($0000) through +8190 ($1FFE) (see Figure 14, top) and a negative value in the range -2 ($FFFE) through -8192 ($E000) (see Figure 14, bottom).

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 14: 13-bit offset extended to 16-bits (original 13-bit offset shaded in green)
Top: Positive; Bottom: Negative

   b) The 10-bit offset used with instruction other than BL has a positive value in the range 0 ($0000) through 1022 ($03FE) (see Figure 15, top) and a negative offset has a value in the range -2 ($FFFE) through -1024 ($FC00) (see Figure 15, bottom).

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 15: 10-bit offset extended to 16-bits (original 10-bit offset shaded in green)
Top: Positive; Bottom: Negative

4. The value of the 16-bit offset (positive or negative) is then added to the PC. Since fetching an instruction also increments the PC (i.e., the current value of the PC is the address of the instruction plus 2), the possible addresses relative to the address of the branching *instruction* are PC + 2 + offset:

   - For BL, the range is PC – 8190 to PC + 8192,
   - For all other branching instructions, the range is PC – 1022 to PC + 1024.

   Note that it is possible for a program to branch to itself with an offset of -2.

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. The calling sequence can inspect the carry bit (using BC or BNC):

```
      BL    Subr
;
      BNC   FalsePart   ; Go to FalsePart of Carry is clear (i.e., no Carry)
; Get here if Carry is set
```

An odd number has its least-significant bit set. This can be tested using BIT and BNE:

```
      BIT   #1,R0 ; And R0 with 1 – if odd, the Zero status bit is clear
      BNE   OddNumber
```

Branching to a subroutine uses the LR register (R4). If it is in use before the call is made, it should be saved on the stack and retrieved on return:

```
      ST    LR,-SP      ; Save LR on stack
      BL    subroutine
      LD    SP+,LR      ; Restore LR from the stack
```

Similarly, inside a subroutine, if a fifth general-purpose register is needed, LR 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:

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

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

The register-register and constant-register instructions perform operations on any pair of registers or a constant and a 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			DST		

Where Opcode indicates the instruction, W/B whether the operation is on a word or a byte, and DST, the destination register. The R/C field indicates whether the SRC field is a register or an encoded constant. These instructions are grouped into arithmetic and logic instructions and register-exchange instructions.

5.4 Arithmetic and logic instructions

In these instructions, the SRC can be either a register or one of eight encoded constants, as shown in Table 9. If R/C (bit 7) has a value of 0, SRC (bits 3 through 5) is treated as register (R0 through R7). On the other hand, if R/C has a value of 1, X-Makina uses the value of SRC to indicate one of eight immediate-value constants (0, 1, 2, 4, 8, $00FF, $FF00, or -1). For example, if R/C is 1 and SRC is 7, X-Makina uses the constant -1, while an R/C value 0 and a SRC value of 4 causes X-Makina to use the contents of R4 (LR).

Table 9: Interpretation of R/C value and SRC values

R/C		SRC
0	1	Value (bits 3-5)
Register	Constant value	
R0	0	0
R1	1	1
R2	2	2
R3	4	3
R4/LR	8	4
R5/SP	$00FF	5
R6/PSW	$FF00	6
R7/PC	-1	7

The two-operand instructions are listed in Table 10. These instructions can be modified using the R/C bit or the W/B bit. They also affect the PSW bits.

Table 10: Two-operand instructions

Instruction	Operation	Description	Opcode
ADD(.B or .W) SRC,DST	*DST ← DST + SRC*	Add SRC to DST	0110.0000
ADDC(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Add SRC and carry to DST	0110.0010
SUB(.B or .W) SRC,DST	*DST ← DST + (¬SRC + 1)*	Subtract SRC (2s comp) from DST	0110.0100
SUBC(.B or .W) SRC,DST	*DST ← DST + ¬SRC + C*	Subtract SRC (1s comp) plus carry from DST	0110.0110
DADC(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Decimal add SRC and carry to DST	0100.0000
CMP(.B or .W) SRC,DST	*DST – SRC*	Compare DST with SRC	0110.1000
XOR(.B or .W) SRC,DST	*DST ← DST ⊕ SRC*	XOR SRC with DST	0110.1010
AND(.B or .W) SRC,DST	*DST ← DST & SRC*	AND SRC with DST	0110.1100
BIT(.B or .W)	*DST & SRC*	Test if bits set in SRC are set in DST	0110.1110
BIC(.B or .W)	*DST ← DST & ~SRC*	Clear bits in DST specified by SRC	0111.0000
BIS(.B or .W)	*DST ← DST | SRC*	Set bits in DST specified by SRC	0111.0010

The BIT instruction is non-destructive in that it simply ANDs the DST with the SRC, the result is used to change the PSW status bits zero and negative.

The difference between SRC as a register and SRC as a constant is illustrated in the following example, in which R2 is to be incremented by 2:

```
; Using a register:
      MOVLZ       #2,R3 ; R3 is used as a temp register holding the value 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. A register can be used as an 8-bit counter.

The bit-operation instructions (BIT, BIC, and BIS) require the use of a bit-mask when accessing one or more bits in a destination register. A bit that is set in the mask is affected, whereas cleared bits are ignored. For example, to clear bit 2 in register R0, a mask of $04 is used, not the bit number:

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

That is:

```
BIC     #4,R0
```

5.5 Register-exchange instructions

X-Makina supports two register-exchange instructions, SWAP and MOV, shown in Table 11. These instructions do not change the PSW status bits.

Table 11: Register-exchange instructions

Instruction	Operation	Description	Opcode
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.	0111.0100
MOV(.B or .W) SRC,DST	*DST ← SRC*	Move SRC to DST. SRC can be a constant (see above) or a register.	0111.0110

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 .B modifier:

```
MOV.B       R3,R2 ; LSB of R3 copied to LSB R2, MSB of R2 is not changed
```

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 needed)
```

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

5.6 Single-register instructions without an operand

X-Makina has four single-operand instructions (sometimes called one-address instructions) that only operate on a register, they all share a common format:

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

Where:

Opcode holds the eight bits reserved for the opcode.

B/W indicates whether the instruction is to operate on a word (B/W = 0) or a byte (B/W = 1). Note that SWPB and SXT are byte-specific instructions (that is, they operate on the bytes in a word).

DST specifies the register number (0 through 7).

The one-operand instructions are listed in Table 12.

Table 12: One-operand instructions

Instruction	Operation	Description	Opcode
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	0111.1000
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.	0111.1010
SWPB(.W) DST	*TMP ← DST.MSB* *DST.MSB ← DST.LSB* *DST.LSB ← TMP*	Swap bytes in DST (word only). W/B bit is ignored.	0111.1100
SXT(.W) DST	*bit 7→ bit 8 →…→ bit 15*	Sign extend LSB byte to word in DST (word only). W/B bit is ignored.	0111.1110

The arithmetic shift instruction does not change the sign bit, thereby supporting signed division by 2 with the SRA instruction (shift-right arithmetic, that is, with sign-extension); for example, dividing -6 ($FFFA), stored in R3, by 2:

```
SRA.W       R3
```

When shifted right, the value of R3 becomes $FFFD (note sign-extension) or -3.

The rotate-right instruction copies the carry-bit into the most-significant bit while shifting the register's bits to the right by 1. The least-significant bit is copied into the carry-bit. The following example counts the number of bits that are set in R0:

```
      movlz R1    ; R1 is the count of set bits in R0 (initially zero)
Loop
      ; Bit zero of PSW (Carry) is cleared (to clear MSB of R0 in next rotate)
      bic   #1,PSW
      cmp   #0,R0 ; Is R0 zero?
      bz    Done  ; Yes: done; no, continue
      rrc   R0    ; LSB of R0 copied to Carry (zero or non-zero)
      addc  #0,R1 ; R1 = R1 + #0 + Carry
      bal   Loop  ; Repeat
Done
```

A shift is different from a rotate in that a rotation performed sixteen 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 (section 6).

SWPB swaps the bytes in a register, whereas SWAP swaps the contents of two registers.

6 Emulated instructions

The X-Makina ISA has a limited number of instructions when compared to other processors; for example, it does not have instructions for return-from-subroutine, register increment, or stack access. Table 13 lists an additional 24 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.

Table 13: Emulated instructions

Instruction	Emulation	Description
ADC.*x Rx*	ADDC.*x* #0,*Rx*	Add carry to Rx
CALL subr	BL subr	Call subr; Return address in LR
CLR.*x Rx*	MOV.*x* #0,*Rx*	Clear Rx
CLRC	BIC #1,PSW	Clear carry bit
CLRN	BIC #4,PSW	Clear negative bit
CLRZ	BIC #2,PSW	Clear zero bit
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
INV.*x Rx*	XOR.*x* #−1,*Rx*	Invert Rx
JUMP *Rx*	MOV *Rx*,PC	Branch to destination (in Rx)
NOP	MOV PSW,PSW	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 ISR
RLC.*x Rx*	ADDC.*x Rx*,*Rx*	Rotate left Rx through carry
SBC.*x Rx*	SUBC.*x* #0,*Rx*	Subtract borrow (1−carry) from Rx
SETC	BIS #1,PSW	Set carry bit
SETN	BIS #4,PSW	Set negative bit
SETZ	BIS #2,PSW	Set 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

Since X-Makina can run at eight different interrupt levels (rather than a single level of interrupt), the single-level of interrupt instructions for enabling and disabling interrupt can be replaced with a Set Priority Level (SPL) instruction. Table 14 shows the emulated SPL instructions. Note that all status bits are cleared.

Table 14: Set Priority Level instructions

Emulated	Actual	Description
SPL0	MOVLZ #$0,PSW	PSW.Priority = 0
SPL1	MOVLZ #$20,PSW	PSW.Priority = 1
SPL2	MOVLZ #$40,PSW	PSW.Priority = 2
SPL3	MOVLZ #$60,PSW	PSW.Priority = 3
SPL4	MOVLZ #$80,PSW	PSW.Priority = 4
SPL5	MOVLZ #$A0,PSW	PSW.Priority = 5
SPL6	MOVLZ #$C0,PSW	PSW.Priority = 6
SPL7	MOVLZ #$E0,PSW	PSW.Priority = 7

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 *pre-assembler* that takes an emulated instruction and translates it into its X-Makina equivalent. For example, a record containing the **RET** instruction would be translated into **MOV LR,PC**, while **INC.B r3** would become **ADD.B #1,r3**.

By using a constant it is often unnecessary to use a second register. For example, R0 can be incremented by 2 using a constant rather than a second register:

```
; Using a register value:
      movlz       #2,R1 ; R1 = 2
      …
      add   R1,R0 ; R0 = R0 + R1 (2)
;
; Using a constant:
      add   #2,R0 ; R0 = R0 + 2
```

Of course, this is only true when the constants -1, 0, 1, 2, 4, 8, $FF, or $FF00 are used. However, since these values are widely-used constants, the need for second register has been greatly reduced.

7 Arithmetic

7.1 Sign, carry, and overflow bits

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 by the add, subtract, and compare instructions when:

- In addition, the source and destination signs are different:

Source sign	Opr	Destination sign	Destination sign
Positive	+	Positive	Negative
Negative	+	Negative	Positive

- In subtraction and compare, the initial sign of the destination is different from the result (note that the compare instructions do not overwrite the destination):

Destination value	Opr	Source sign	Destination value
Positive	–	Negative	Negative
Negative	–	Positive	Positive

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.

7.1.1 Examples

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

5 + (-2): In this example, 5 ($05) is added to -2 ($FE). The result is '+3'. The most-significant bit (bit 7) is '0', meaning the sign-bit (S) is clear ('0'). However, since the result of the addition required an extra bit, the carry-bit is set. The sign of the source ('+5') is the same as the result ('+3'), indicating that an overflow has not occurred.

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

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 source and destination signs are different). The addition did not result in a carry (the carry-bit is clear, or '0'). The V-bit in the PSW is set.

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

-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. The sign-bit indicates a negative value ('-1'). The carry-bit is set.

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

Finally, the sign, carry, and overflow bits 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    ; $FF
NUM2  EQU   -8    ; $F8
;
; The following addition adds the bytes -1/$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:
;
      BN    SignSet     ; Bit 7 is set (true in either signed or unsigned)
      BC    CarrySet    ; In unsigned arithmetic, result exceeds 8 bits
```

Unlike some ISAs, X-Makina does not have an explicit test for arithmetic overflow. However, the BGE (branch if greater or equal) and BL (branch if less) use the overflow bit as part of the conditional test.

7.2 Addition

X-Makina 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 X-Makina):

```
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 (low words of DW0 and DW1)
      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
      LD          R0,R2
      LD          R1,R3
      ADDC.W      R3,R2       ; R2 = R2 + R3 + C (high words of DW0 and DW1)
      ST          R2,R0       ; High word of DW0 = DW0 + DW1
;
      MOV         LR,PC       ; Return
```

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

7.3 Subtraction

X-Makina 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 that it is necessary to borrow.

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 any borrow.

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

1. The operation of the subtract instruction is defined as:[9]

   dst + .NOT. src + 1 → dst

   The dst (the minuend) is added to the ones-complement of the src (the subtrahend) plus '1', the result is stored in the dst (the difference).

2. The rules for "borrow" (represented by the carry-bit) are defined as:

   C: Set if there is a carry from the MSB of the result, reset otherwise.
   Set to 1 if no borrow, reset if borrow.

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

The carry-bit is used for multiple-unit (i.e., bytes or words) subtraction. If it is set, it indicates that unit 'N' does not need to borrow from unit 'N+1', whereas if it is clear, it indicates that unit 'N' must borrow from unit 'N+1'. Its use is described below (section 7.3.1).

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 borrow. Since this is single-byte subtraction, borrow 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	*.NOT. 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, no borrow

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 borrow.

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	*.NOT. 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, no borrow

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. Although the carry being clear indicates that borrowing is required, it 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, borrow indicated

– 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, no borrow

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, borrow indicated

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

7.3.1 Multiple-byte and multiple-word subtraction

If the subtraction involves multiple-units (i.e., multiple bytes or words) will require the result of the unit 'N' subtraction (that is, whether borrow is required) to be conveyed to unit 'N+1'. This requires the use of a new instruction, SUBC (or SUBC.W or SUBC.B), to apply borrow 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 ($0003 – $FFFD; 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	$0003
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	$FFFD

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 final result indicating whether borrowing is necessary (in this case it is):

Action	C	7	6	5	4	3	2	1	0	Comments
	0	0	0	0	0	0	0	1	1	*dst*: Value of LSB of $03
+	0	0	0	0	0	0	0	1	0	*.NOT. src*: 1s complement of LSB of $FD
+	0	0	0	0	0	0	0	0	1	Add 1
	0	0	0	0	0	0	1	1	0	Answer: $06. 'C' clear, borrow indicated

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

1. The operation of the SUBC instruction is defined as:

   $dst + .NOT.\ src + C \rightarrow dst$

   The minuend (*dst*) is added to the ones-complement of the subtrahend (NOT *src*). However, unlike the SUB instruction, SUBC adds the carry-bit ('1' if borrow is not required, '0' otherwise).

2. The rules for "borrow" (represented as Carry) are the same as for the subtract instruction:

C: Set if there is a carry from the MSB of the result, reset otherwise.
Set to 1 if no borrow, reset if borrow.

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	0	*dst*: Value of MSB of $00
+	0	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, borrow indicated

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	*.NOT. 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, borrow indicated

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	*.NOT. 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, no borrow indicated

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	*.NOT. 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, no borrow

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	*.NOT. 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, no borrow

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 a borrow (Carry = 0) 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	*.NOT. src*: 1s complement of MSB of $04
+	-								1	Add 1
	0	1	1	1	1	1	1	1	1	Answer: $FF. 'C' clear, borrow indicated

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	*.NOT. 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, borrow indicated

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 (with borrow)
       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 any borrow, 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).

8 Subroutines and arguments

8.1 Subroutine calls

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

1. LR ← PC
2. PC ← PC + 2 + sign-extended offset

The LR has 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
       ...
Subrx
       ...
       MOV    LR,PC
```

Alternatively, LR can be pushed onto the stack and used as another general purpose register (R4):

```
Subrx
      ST    LR,-SP
; LR can be used as a general purpose register
      MOVLZ R4          ; R4 and LR are the same register, LR is an alias of R4
      ...
; Return by pulling LR and storing in PC
      ST    SP+,PC
```

8.2 Subroutine arguments

X-Makina has four general purpose registers (R0 through R3) that can be used to pass arguments to a subroutine, therefore some applications may not be a need for passing arguments by the stack. However, in those applications where it is either required or necessary, the stack is the most obvious choice of data structure to hold the arguments.

Each subroutine is associated with a **stack frame**, that part of the stack containing its arguments, 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 arguments (within the subroutine, referred to as parameters).

C arguments are pushed onto the stack from right-to-left, ensuring the first (leftmost) argument 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 arguments 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:

1002	10	a ← SP
1004	'x'	b
1006	4	c
1008		Last word in caller's stack frame

The subroutine is written as follows:

```
char subr(int p, char q, int r)
{
int i, j;
i = p - r;
j = p + r;
return q;
}
```

In this example, it is assumed that R3 is the stack-frame pointer (this is a purely arbitrary choice). The entry point to the called subroutine must preserve the previous stack frame by storing its

stack frame point (R3) on the stack. It then creates its own stack frame pointer (R3, the same register) by assigning the current top-of-stack (SP) to R3. Since the stack pointer points to the top-of-stack, by not decrementing it, the new stack frame pointer points to the previous stack frame pointer value on the stack. Finally, the SP is decremented by the number of bytes reserved for the automatics. The entry-point assembler code is as follows:

```
st        R3,-SP  ; Save previous stack frame (R3) on top-of-stack
mov.w     SP,R3   ; New stack frame points to subr's stack frame
add       #-4,SP  ; Reserve space for automatics i and j (two 16-bit words)
```

The stack now contains:

0FFC	Reserved for j	←SP	R3 – 4 (automatic 'j')
0FFE	Reserved for i		R3 – 2 (automatic 'i')
1000	Previous R3	←R3	R3 (stack frame pointer)
1002	10	**p**	R3 + 2
1004	'x'	**q**	R3 + 4
1006	4	**r**	R3 + 6
1008		Last word in caller's stack frame	

R3 can be used to access any value from the stack frame: positive offsets refer to parameters (`p`, `q`, and `r`) while negative offset refer to the automatics. For example, R3 + 4 refers to parameter `q`, while R3 – 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 R3. The assembler code for the two statements can be written as follows:

```
ldr       r3,#2,r0    ; r0 = [r3+2] value of p
ldr       r3,#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,r3,#-2   ; [r3-2] (location for i) = r0 (value of i)
add       r1,r2       ; r2 (value of j) = r2 (p) + r1 (r)
str       r2,r3,#-4   ; r3-4 (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       r3,#4,r0    ; R0 = [r3+4] (value of q)
```

Returning from the subroutine requires freeing the space reserved for the automatics and restoring the calling subroutine's stack-frame pointer. These two operations can be implemented as follows:

```
mov       R3,SP       ; SP = R3 (skip automatics)
ld        sp+,r3      ; Restore caller's stack frame pointer (R3)
```

The stack now contains:

1002	10	**a** ←SP
1004	'x'	**b**
1006	4	**c**
1008		Last word in calling stack frame

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

```
mov       lr,pc
```

The calling subroutine must remove its arguments from the stack as well as save the return value. Removing the arguments involves increasing the SP by (3 words); to do this means adding the constant 2 and then the constant 4 to SP:

```
add       #2,SP       ; Discard arguments (3 words or 6 bytes in total)
add       #4,SP       ;
str       r0,r3,#-10  ; Assume result is an automatic on the stack [r3-10]
```

9 Interrupts, faults, and traps

This section explains how X-Makina handles **exceptions**, commonly referred to as 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 a monitor or operating system. A trap is an instruction that is executed by the CPU.

When one of these exceptions 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 upon 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 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, the device.
- 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**.

An exception has characteristics similar to that of a subroutine in that the currently executing application stops while the subroutine or ISR performs its task. Similarly, the state of the caller or application associated with the exception should be saved in order to allow it to resume correctly when the routine has completed.

A subroutine has an entry point which the calling routine knows. 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 X-Makina's ISA to support 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 device or other exceptions (i.e., faults and traps). When the exception occurs, the CPU knows which ISR is to be invoked.

Although it is ISA dependent, an interrupt causes the CPU to save some of the application's state (at a minimum, the program counter and the program status word). Other state information may or may not be saved by the CPU. The program counter is then assigned the value of the ISR's entry point from the interrupt vector table. On completion of the ISR, instructions usually exist to restore the state of the application.

X-Makina has a 16-entry interrupt vector table stored in high memory (see Figure 9). Each location is associated with a PSW (holding the new priority) and the entry point of the corresponding exception handler; the lowest address ($FFE0) has the lowest priority (0), while the highest address ($FFFC) has the highest priority (15).

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, PSW, and LR) is saved, the handler's PSW becomes the current PSW, and control passes to the handler:

1. Push PC
2. Push PSW
3. Push LR
4. PSW ← PSW of handler (memory[Vector Address])
5. PC ← Address of handler (memory[Vector Address + 2])
6. LR ← $FFFF

When the handler has completed, control returns to the previously executing instruction by copying the current LR to the PC, either SWAP LR,PC or MOV LR,PC (as with a subroutine return). Since the LR contains an invalid address ($FFFF), the CPU performs the following:

1. Pull LR
2. Pull PSW
3. Pull PC

Pending interrupts, faults, and traps are queued, in order of priority; for those at the same priority, they are queued in order of arrival (oldest to youngest).

10 Initial CPU state

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

The reset or start-up subroutine should operate with the highest priority (7) to ensure that it is not interrupted.

11 Structures

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

11.1 Code structures

11.1.1 Sequential statements

A sequential 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 X-Makina offers a (relatively) large number of registers, compilers and assembly-coders can use registers to reduce the number of memory accesses, thereby increasing the speed of the program; for example:

```
  MOVL      AddrA1,R0   ; Access a1's initial value from memory
  MOVH      AddrA1,R0
;
  LD        R0,R1       ; R1 is a1
  ADD       #1,R1       ; a1 = a1 + 1
  ST        R1,R0       ; Write a1 to its memory location
;
  MOV       R1,R2       ; R2 is a2
                        ; Initialize a2 with a1
  ADD       #3,R2       ; a2 = a1 + 3
;
  MOVL      AddrA2,R0   ; Update a2's value in memory
  MOVH      AddrA2,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
;
  SLA 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.

11.1.2 Conditional statements

A conditional statement changes the program's flow based upon some condition being met. Broadly speaking, there are three such constructs.

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 BEQ 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
    BNE   EndIF       ; Go to EndIF if PSW.Z != 1 (which it must be)
TruePart
    ; TruePart instructions
EndIF
```

In this case, the BNE could be replaced since if the comparison wasn't true (i.e., equal), then it must have been false, so BAL could be used in place of BNE:

```
    CMP   R0,R1       ; R0 – R1, if zero, PSW.Z=1
    BEQ   TruePart    ; Go to TruePart if PSW.Z=1
    BAL   EndIF       ; Always go to EndIF
TruePart
    ; TruePart instructions
EndIF
```

The BAL is unnecessary. By applying de Morgan's rules, the BEQ can be replaced with a BNE to the EndIF instruction and the BAL can be removed. The instructions following BNE 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) instruction (test for equal to zero) and a BEQ:

```
    TST   R0    ; Implemented as a CMP #0,R0
    BEQ   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 is not equal to b, no need to check c and d
; At this point, a=b
    CMP   R2,R3
    BEQ   FalsePart   ; c is equal to d, true-part should not be executed
; At this point, a=b and c!=d
    ...
    BAL   EndIF
FalsePart
    ...
EndIF
```

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

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

For the true-part to be executed, either 'a' equals 'b' or 'c' is not equal to 'd'. This can be implemented as:

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

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:

```
    CMP   R0,R1
    BLT   ElsePart
    MOV   B,A
    BAL   EndIF
ElsePart
    CLR   R0
EndIF
```

Switch-Case

Switch-case is discussed in section 11.2.2 (Switch-Case implementation using arrays), below.

11.1.3 Looping statements

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

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 < 10)
{
     ...
     i = i + 1;
}
```

Can be implemented as:

```
; i = 0
    MOV   #0,R0       ; Use R0 as 'i'
; while (i < 10)
WhileStart            ; {
    CMP   #10,R0
    BGE   WhileEnd    ; 10-RO; remain in loop until R0>=10
    ...
    ; i = i + 1
    ADD   #1,R0
    BAL   WhileStart
WhileEnd              ; }
```

The different conditions discussion in section 11.1.2, above are applicable in both pre- and post-test loops.

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
    TST   R0
    BNE   DoStart
```

Deterministic

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++)
{
     ...
}
```

Can be implemented as:

```
; ch = 'A'
    MOVLZ #'A',R0     ; Use R0 as 'ch'
    MOVLZ #'Z',R1
;
ForLoop
; ch != 'Z'
    CMP.B R1,R0
    BEQ   ForEnd
    ...
; ch++
    ADD.B #1,R0       ; Next character
    BAL   ForLoop
ForEnd
```

11.2 Data structures

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

11.2.1 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
clr       R1
;
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
st.b      #'x',R3     ; [R3 + 0] = 'x'
add       #1,R3       ; R3++
st.b      #'y',R3     ; [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.

11.2.2 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:

```
switch(x)
{
case 0: ...
case 1: ...
case 2: ...
case 3: ...
}
```

Can be implemented as:

```
; Case label table holding addresses
CaseList  word  Case0
          word  Case1
          word  Case2
          word  Case3
          ...
```

```
; Calculate case value (x) 0, 1, 2, or 3 and store in R3
; Multiply R3 by 2 (shift left by 1) to get word offset (0, 2, 4, or 6)
          sla.w R3
; Pass control to specified address:
; Add R3 (x * 2) to CaseList (base address)
; PC = [R3]
          movl  CaseList,R2
          movh  CaseList,R2
          add   R2,R3
          mov   R3,PC
```

11.2.3 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 15**.

Table 15: 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

'ex' 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, where R0 is address of ex, R0=R0+2
; ex.b = 13
    MOVLZ       #13,R1
    ST.W        R1,RO+      ; [R0] = 13, where 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+
; ex.c[1] = 'b'
    ADD.B       #1,R1       ; Incr 'a' to 'b'
    ST.B        R1,R0+
; 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]
```

A structure can be passed to a subroutine as an argument. 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.

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

```
    MOV   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
    BAL   Loop
```

It is slightly more complex if the index is supplied as, for example, an argument 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
```

```
    SLA   R3                ; R3 * 2
    MOV   R3,R4             ; R4 = index * 2
    SLA   R3                ; R3 * 4
    SLA   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
```

11.2.4 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 CPU it is stored in a general purpose 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:

```
    MOVL  alpha,R0
    MOVH  alpha,R0
```

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

```
    MOVLZ #0,R1
    ST    R1,R0       ; Clears alpha (R1=0)
    LD    R0,R2       ; R2 is assigned the contents of alpha
```

(This also shows a shortcoming of the current design of X-Makina, since it can only be used with register operands, meaning an instruction such as ST #0,R0 is invalid.)

The above example used a static address. A pointer can be assigned a dynamic value in a number of 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:

```
    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            ; Access first 16-bit locations (assuming short *ptr
*ptr = 1;
```

  Implemented as:

```
    MOVLZ #10,R0
    BL    malloc      ; Assume R0 indicates number of bytes to allocate
                      ; And 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
            bal   strlen1     ; Check next byte
;
strlen2     ld    sp+,R1      ; Restore R1
            mov   lr,pc       ; Return to caller.  R0 has length
```

12 X-Makina Instruction Set

Bit meanings:

PRPO – pre- or post- increment or 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 or byte address/action

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)

OFF – one bit of an 11-bit offset

S/C – Source register or Constant encoding (see Table 16).

Table 16: Register and Constant values from R/C and SRC bits

R/C		SRC
0	**1**	**Encoding (bits 3-5)**
Register	**Constant**	
R0	0	0
R1	1	1
R2	2	2
R3	4	3
R4/LR	8	4
R5/SP	0x00FF	5
R6/PSW	0xFF00	6
R7/PC	-1	7

Table 17: X-Makina Instruction Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Instruction
1	0	0	0	0	PRPO	DEC	INC	0	W/B	S	S	S	D	D	D	**LD**	Load DST from mem[SRC plus addressing]
1	0	0	0	1	PRPO	DEC	INC	0	W/B	S	S	S	D	D	D	**ST**	Store SRC in mem[DST plus addressing]
1	1	0	OFF	OFF	OFF	OFF	OFF	OFF	W/B	S	S	S	D	D	D	**LDR**	Load DST from mem[SRC + offset]
1	1	1	OFF	OFF	OFF	OFF	OFF	OFF	W/B	S	S	S	D	D	D	**STR**	Store SRC in mem[DST + sign-extended offset]
1	0	0	1	0	B	B	B	B	B	B	B	B	D	D	D	**MOVL**	DST.Low byte ← BBBBBBBB; High byte unchanged
1	0	0	1	1	B	B	B	B	B	B	B	B	D	D	D	**MOVLZ**	DST.Low byte ← BBBBBBBB; Zero high byte
1	0	1	0	0	B	B	B	B	B	B	B	B	D	D	D	**MOVH**	DST.High byte ← BBBBBBBB ; Low byte unchanged
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	**BNE/BNZ**	Branch if not equal or not zero
0	0	1	0	0	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	**BEQ/BZ**	Branch if equal or zero
0	0	1	0	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	**BNC/BLO**	Branch if no carry/lower
0	0	1	0	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	**BC/BHS**	Branch if carry/higher or same
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
0	0	1	1	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	**BLT**	Branch if less
0	0	1	1	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	**BAL**	Branch Always (unconditionally)
0	1	1	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	1	0	0	0	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**ADDC**	Add: DST ← DST + Carry + SRC/CON
0	1	1	0	0	1	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**SUB**	Subtract: DST ← DST – SRC/CON
0	1	1	0	0	1	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**SUBC**	Subtract: DST ← DST – SRC/CON
0	1	0	0	0	0	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**DADC**	Decimal add: DST ← DST + Carry + SRC/CON
0	1	1	0	1	0	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**CMP**	Compare: DST – SRC/CON
0	1	1	0	1	0	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**XOR**	Exclusive or: DST ← DST ⊕ SRC/CON
0	1	1	0	1	1	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**AND**	Logical AND: DST ← DST & SRC/CON
0	1	1	0	1	1	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**BIT**	Bit test: DST & SCR/CON
0	1	1	1	0	0	0	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**BIC**	Bit clear: DST ← DST & ~SRC/CON
0	1	1	1	0	0	1	0	R/C	W/B	S/C	S/C	S/C	D	D	D	**BIS**	Bit set: DST ← DST | SRC/CON
0	1	1	1	0	1	0	0	0	0	S	S	S	D	D	D	**SWAP**	Swap SRC and DST
0	1	1	1	0	1	1	0	0	0	S	S	S	D	D	D	**MOV**	DST ← SRC/CON
0	1	1	1	1	0	0	0	0	W/B	0	0	0	D	D	D	**SRA**	Shift DDD right (1 bit) arithmetic
0	1	1	1	1	0	1	0	0	W/B	0	0	0	D	D	D	**RRC**	Rotate DDD right (1 bit) through carry
0	1	1	1	1	1	0	0	0	0	0	0	0	D	D	D	**SWPB**	Swap bytes in DDD
0	1	1	1	1	1	1	0	0	0	0	0	0	D	D	D	**SXT**	Sign extend byte to word in DDD

13 Revision history

Revision	Date	Page	Correction
1.0	18 05 08		Original version
1.01	18 05 10	9	#Array1 and #Array2 replaced with Array1 and Array2, respectively
		10	#OFF replaced with OFF Addition of footnote 5 regarding offsets and how numerics and labels are treated differently
		11	Table 7, #Value replaced with Value #Array replaced with Array Footnote 6 added.
		21	#NUM1, #NUM2 replaced with NUM1, NUM2, respectively #DW0 replaced with DW0
		22	#DW1 replaced with DW1
		27	#NUM0 replaced with NUM0 #NUM1 replaced with NUM1
		32	#AddrA1 replaced with AddrA1 #AddrA2 replaced with AddrA2
		37	#iarray, #carray replaced with iarray and carray, respectively
		38	#CaseList replaced with CaseList
		39	#ex, #NUL replaced with ex and NUL, respectively
		41, 42	#ex_array replaced with ex_array
		42	#alpha replaced with alpha
		43	#NUL replaced with NUL cmp.b R2,#NUL replaced with cmp.b NUL,R2
1.02	18 05 15	17	RRA.W replaced with SRA.W
		25	Subtraction example fixed MSB was identical to LSB $0103 - $0004 now gives the correct explanation for the result
		19	SPL1 repeated in Table 14 replaced with correct values (SPL1 through SPL5)
1.03	18 05 24	7	Footnote regarding JUMP
		Various	Replaced “jump” and “jumping” with “branch” and “branching”
		i, ii	Added Table of Contents