

XM23p Assembler User's Guide

Larry Hughes, PhD

9 May 2024

Revision 2.2

Table of Contents

1	The Assembler	1
2	The assembly language file	1
2.1	Record format	2
2.2	Labels	2
2.3	Instructions	2
2.4	Directives	2
2.5	Operands.....	4
2.6	Comments.....	5
2.7	Notes	5
3	Built-in symbols	5
4	Examples	5
4.1	First pass errors - .LIS file	5
4.2	Second pass - .LIS file	9
4.3	Second pass - .XME file	12
5	Internals of the assembler	13
5.1	First pass	13
5.2	Second pass.....	14
5.3	Notes	14
5.4	Errors in the assembler	14
6	XM23 Instruction Set	15

1 The Assembler

The XM23p assembler is a one-pass assembler with an optional second pass which is used if forward references are encountered.

The first pass builds the symbol table (from records in an XM23p assembly-language source module) and generates an executable load module (a file) for the XM23p machine. If a forward reference is found, the assembler continues building the symbol table until the end of file is reached and starts the second pass.

The optional second pass re-reads the file and using the symbol table, generates an XM23p executable load module.

In either case, a list file is produced containing the records from the assembly-language file at the end of the first pass (if the executable could be created or errors were detected during the first pass) or the end of the second pass (whether errors are detected during the second pass).

This document describes how to use the XM23p assembler and the internals of the assembler.

2 The assembly language file

An assembly language module is any text file (for example, created through Notepad or Word-pad) consisting of *records*. Each record has a specific format, described below.

2.1 Record format

An assembly-language file consists of one or more *records* containing instructions and data for the assembler to translate into machine-readable records. The record is *free format*, meaning that it has no fixed fields. All records have the same format, defined as follows (**bold** indicates terminal symbols):¹

Record = (*Label*) + ([*Instruction* | *Directive*]) + (*Operand*) + (; *Comment*)

Label = *Alphabetic* + 0 {*Alphanumeric*} 32

Instruction = * *An instruction mnemonic, see section 2.3* *

Directive = * *An assembler directive, described in section 2.4* *

Operand = * *The operand(s) associated with the Instruction or Directive, see section 2.5* *

Comment = * *Text associated with the record – ignored by the assembler, see section 2.6* *

Alphabetic = [**A..Z** | **a..z** | **_**]

Alphanumeric = [**A..Z** | **a..z** | **0..9** | **_**]

Instructions and *Directives* are treated as case insensitive (that is, the instruction or directive can be upper case, lower case, or some combination thereof). However, a *Label*, if it exists, is case sensitive, meaning that, for example, the label **Alpha** is not the same as the label **ALPHA**.

2.2 Labels

A *Label* is a text-string of up to 32 characters. Each label must begin with an alphabetic character and be followed by zero or more alphabetic or numeric characters (i.e., alphanumeric). *Labels* are case sensitive.

Valid labels are stored in the symbol table with either the value of the location counter (i.e., where the instruction will be placed in memory, if there is an instruction on the record or if the remainder of the record is blank or contains a comment) or the value associated with the equate directive (see 2.4, *Directives*).

2.3 Instructions

The assembler supports XM23p's Instruction Set Architecture (ISA), found in *XM23p Instruction Set Architecture*. A summary of the ISA can be found in section 6.

Instructions are case insensitive. The assembler produces the same executable code for the following instructions (**LD** – load register):

LD R2,R3 ; R3 <- R2

Ld R2,R3 ; R3 <- R2

lD R2,R3 ; R3 <- R2

ld R2,R3 ; R3 <- R2

Regardless of case, *Instructions* are **reserved words** and cannot be used as *Labels* or *Operands*.

2.4 Directives

A directive (or *pseudo-instruction*) is a command in a record that is processed by the assembler (i.e., it directs the assembler to do something). It has no corresponding machine instruction, although it can

¹ The example is shown in a Data Dictionary format that can be used to define data structures: '(' ')' – optional; '[' ']' – choice; 'LB{' '}'UB' – sequence from LB (default 1) to UB (default ∞); and '+' – AND or concatenate.

produce machine code. The directives currently supported by XM23p are (*Operand* is described in section 2.5, Operands):

ALIGN

Increments the location counter to the next even-byte address if the location counter is odd. If the address is odd, the executable file will contain a zero-value in the .XME file. **ALIGN** does not take an operand.

AORG *Operand*

The **AORG** (absolute origin) directive functions as the **ORG** directive does, changing the value of the location counter to the address specified in the *Operand*; these addresses are referred to as absolute addresses and are typically used when referring to fixed addresses in reserved memory (usually device and interrupt vector addresses). **AORG** requires the linker to resolve the addresses.

ASCII *Operand*

The **ASCII** directive's *Operand* is a character string enclosed in double quotes. The characters in the string are converted into their binary equivalent and stored in contiguous locations. The directive only accepts one string. Escaped characters, such as Tab and NUL, are supported.

BSS *Operand*

The **BSS** (Block Started by Symbol) directive reserves a block of memory of *Operand* bytes is reserved. If there is a *Label* associated with the **BSS**, it is stored in the symbol table with the value of the location counter. The location counter is increased by the specified number of bytes. The label can be omitted.

BYTE *Operand*

One byte is stored in the memory location associated with the location counter. An *Operand* larger than 8-bits in length is an error. If there is a *Label*, it is stored in the symbol table along with the location counter. The location counter is increased by one.

CODE

Output from the assembler is written to S1 records. At present, the **ASCII**, **BSS**, **BYTE**, and **WORD** directives ignore the **CODE** directive. The **AORG** and **ORG** change the location counter to refer to code memory.

DATA

Output from the assembler is written to S2 records. At present, all instructions change the assembler's output back to S1 records. The **AORG** and **ORG** change the location counter to refer to data memory.

END (*Operand*)

Denotes the end of the program. Any records that follow the **END** record are ignored. If an *Operand* is supplied, it must refer to a *Label* in the symbol table or an actual address; this is the starting address used by the loader.

EQU *Operand*

The **EQU** (equate) record's *Label* is equated with (i.e., assigned to) the *Operand*. The *Label* and the value of the *Operand* are stored in the symbol table. A *Label* is required. The location counter is not incremented.

ORG *Operand*

The **ORG** (origin) directive changes current location counter value to the address specified in *Operand*.

WORD *Operand*

Two bytes are stored in consecutive memory locations, starting at the location specified by the current value of the location counter. If there is a *Label*, it is stored in the symbol table along with the location counter. The location counter is increased by two bytes. Note that 16-bit quantities should fall on even-byte boundaries. The **ALIGN** directive can ensure proper alignment.

Directives are case insensitive. The assembler makes no distinction between the following directives:

```
BYTE      #FF
Byte      #FF
byte      #FF
```

Regardless of case, *Directives* are reserved words and cannot be used as *Labels* or *Operands*.

2.5 Operands

An *Operand* contains up to three *Values* (separated by commas without leading or following spaces) required by the *Instruction* or *Directive*. It is defined as:

Operand = *Value* + 0{“,” + *Operand*}3

A *Value* is either a numeric value or a label (section 2.2):

Value = [*Numeric* | *Label* | *String*]

Numeric and *Label* values are distinguished using a prefix, with ‘\$’, ‘r’, and ‘#’ denoting a numeric *Value* (*Alphanumeric* is defined above and terminal symbols or values are in **bold**):

Numeric = [“\$” + [*Unsigned* | *Signed*] | “r” + *Char* | “#” + *Hex*]

Unsigned = [0 .. 65535]

Signed = [-32768 .. +0 .. +65535]

Char = [*Alphanumeric* | *Escaped*] + “r”

Hex = 1{0 .. 9 | A .. F | a .. f} * Hex values range from #0 to #FFFF *

Escaped = “\” + *Alphanumeric*

String = 1{*Char*}128

The *Escaped Alphanumeric* value is limited to the ASCII-escape sequences shown in Table 1.

Table 1: Accepted escape sequences

Character	Converted valued	Meaning
'\b'	#08	BS - Backspace
'\t'	#09	TAB
'\n'	#0a	Linefeed, Newline
'\r'	#0d	Carriage return
'\0'	#00	NUL
'\\'	#5c	Backslash
'\"'	#27	Single quote
'\"'	#22	Double quote
'\Unknown'	#3f ('?')	Invalid/unacceptable character

2.6 Comments

A comment is any text after a semicolon (“;”) to the end-of-record (delimited by a NUL character). Comments are ignored by the assembler.

2.7 Notes

- The location counter is incremented by the number of bytes associated with the **ALIGN**, **BSS**, **BYTE**, **ORG**, or **WORD** directive.
- Directives and instructions are reserved words and cannot be used as labels.
- Duplicate labels are not permitted.
- Characters begin and end with the single quote character “’”.
- Decimal values are prefixed with “\$”.
- Hexadecimal numbers are prefixed with “#”.
- Hexadecimal numbers cannot be signed.
- Unsigned values can be associated with the “+” sign (that is, -32768 to +65535)
- Any **WORD** value that exceeds its range is truncated to the least significant two bytes (four nibbles).
- Any **BYTE** value that exceeds its range is truncated to the least significant byte (two nibbles).

In addition, the assembler does not support:

- External references (i.e., references to *Labels* in other files)
- Include files (i.e., external files to be “copied into” the program for assembly)
- In-line arithmetic expressions (i.e., operands containing arithmetic operators)

3 Built-in symbols

There are eight built-in symbols, representing XM23p’s eight CPU registers (R0, R1, R2, R3, R4, R5, R6, and R7). Other symbols can be equated as registers, for example:

```
LR    equ    R5           ; LR is equated with R5
r6    equ    R6           ; r6 is equated with R6
SP    equ    r6           ; SP is equated with r6 (equated with R6)
SP    EQU    R6           ; Same as above, but will result in an error
pc    equ    R7
```

...

```
    mov    LR,pc          ; Subroutine return
    mov    R5,R7           ; Equivalent to the previous record
```

4 Examples

The following example shows how a problem (assigning the ASCII characters ‘A’ through ‘Z’ to a block of memory starting at location #800) could be solved using the XM23p assembler.

4.1 First pass errors - .LIS file

The first attempt at solving the problem is as follows:

```

;
; Sample X-Makina program
; Initialize a block of memory to 'A' through 'Z'
;
; L. Hughes
; 3 May 24: Update to XM23p
; 15 May 18: Comment correct R1 = 'Z'
; 1 May 18: Initial release
;
SIZE equ    $26
CAP_A equ   'A'
CAP_Z equ   'Z'
;
; Start of data area
;
    Data
    org #800
Base bss    SIZE          ; Reserve SIZE bytes
; Start of code area
    Code
    org #1000
;
Main
    movlz   'A',R0        ; R0 = 'A'
    movlz   CAP_Z,R1      ; R1 = 'Z'
    movl    Base,R2       ; R2 = Base (Address for chars)
    movh    Base,R2
;
Loop
    st.b    R0,R2+        ; [R2] = R0; R2 = R2 + 1
    cmp.b   R0,R1         ; R0 = R1 ('Z')
    bep     Done          ; If equal, goto Done
    add.b   $1,R0         ; No: R0 = R0 + 1 (next ASCII char)
    swap    R0,R0         ; NOP to test count
    bra     Loop          ; Repeat loop
; End of program
Done movlz  '*',R1
    cmp.b   R1,R1
;
    end Main              ; End of program - Code begins at "Main"

```

The program is dragged-and-dropped onto the assembler, which stops at the end of the first pass, indicating that first-pass errors occurred. The corresponding .LIS file contains the following:

X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17

Input file name: ArrayInit.asm

Time of assembly: Thu 2 May 2024 09:33:36

```
1          ;
2          ; Sample X-Makina program
3          ; Initialize a block of memory to 'A' through 'Z'
4          ;
5          ; L. Hughes
6          ; 3 May 24: Update to XM23p
7          ; 15 May 18: Comment correct R1 = 'Z'
8          ; 1 May 18: Initial release
9          ;
10         SIZE equ    $26
11         CAP_A equ    'A'
12         CAP_Z equ    'Z'
13         ;
14         ; Start of data area
15         ;
16         Data
17         org    #800
18 0800 0000 Base bss    SIZE    ; Reserve SIZE bytes
19         ; Start of code area
20         Code
21         org    #1000
22         ;
23 1000 6A08 Main movl    'A',R0    ; R0 = 'A'
24 1002 6AD1      movl    CAP_Z,R1  ; R1 = 'Z'
25 1004 6002      movl    Base,R2   ; R2 = Base (Address for chars)
26 1006 7842      movh    Base,R2
27         ;
28         Loop
29 1008 5CC2      st.b     R0,R2+    ; [R2] = R0; R2 = R2 + 1
30 100A 4541      cmp.b   R0,R1     ; R0 = R1 ('Z')
31         bep     Done           ; If equal, goto Done
32         **** Expecting INST or DIR
32 100C 40C8      add.b    $1,R0    ; No: R0 = R0 + 1 (next ASCII char)
33 100E 4C80      swap    R0,R0     ; NOP to test count
34 1010 3FFB      bra     Loop     ; Repeat loop
```



```

35          ; End of program
36 1012 6951 Done movlz '*',R1
37 1014 4549      cmp.b R1,R1
38          ;
39          end Main      ; End of program - Code begins at "Main"

```

First pass errors - assembly terminated

** Symbol table **

Constants (Equates)

Name	Type	Value	Decimal	
CAP_Z	CON	005A	90	PRI
CAP_A	CON	0041	65	PRI
SIZE	CON	001A	26	PRI

Labels (Code)

Name	Type	Value	Decimal	
Done	REL	1012	4114	PRI
bep	REL	100C	4108	PRI
Loop	REL	1008	4104	PRI
Main	REL	1000	4096	PRI

Labels (Data)

Name	Type	Value	Decimal	
Base	REL	0800	2048	PRI

Registers

Name	Type	Value	Decimal	
R7	REG	0007	7	PRI
R6	REG	0006	6	PRI
R5	REG	0005	5	PRI
R4	REG	0004	4	PRI
R3	REG	0003	3	PRI
R2	REG	0002	2	PRI
R1	REG	0001	1	PRI
R0	REG	0000	0	PRI

The assembler indicates that an instruction or directive was expected on record 30 of the .ASM file ("**** Expecting INST or DIR"). The instruction "bep" is taken as a label by the assembler since it is not a valid instruction (the *fault*), meaning that the operand "Done" is taken as an operator (the *error*).

The invalid instruction can be corrected by changing "bep" to "beq". A good example of minding your P's and Q's!

4.2 Second pass - .LIS file

By running the corrected assembler file through the assembler a second time. The assembler indicates that the assembly was successful:

```
Successful completion of assembly - 2P
```

The "2P" indicates it took two passes. Why was this necessary?

Two new files are in the directory, one the list file (.LIS) and the other, the executable (.XME). The .LIS file contains the name of the input (.ASM) file (without its full path), a listing of the assembled program (from left-to-right: the record number, the machine address, the instruction or data to be stored in that location, and the original .ASM record), the symbol table (from left-to-right: the name or label, the type [LBL – label or REG – register], and its value), and the name of the .XME file (with its full path):

X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17

Input file name: ArrayInit.asm

Time of assembly: Thu 2 May 2024 09:39:45

```
1          ;
2          ; Sample X-Makina program
3          ; Initialize a block of memory to 'A' through 'Z'
4          ;
5          ; L. Hughes
6          ; 3 May 24: Update to XM23p
7          ; 15 May 18: Comment correct R1 = 'Z'
8          ; 1 May 18: Initial release
9          ;
10         SIZE equ    $26
11         CAP_A equ    'A'
12         CAP_Z equ    'Z'
13         ;
14         ; Start of data area
15         ;
16         Data
17         org    #800
18 0800 0000 Base bss    SIZE          ; Reserve SIZE bytes
19         ; Start of code area
20         Code
21         org    #1000
22         ;
23 Main
24 1000 6A08      movl    'A',R0          ; R0 = 'A'
25 1002 6AD1      movl    CAP_Z,R1       ; R1 = 'Z'
26 1004 6002      movl    Base,R2        ; R2 = Base (Address for chars)
27 1006 7842      movh    Base,R2
28         ;
29 Loop
30 1008 5CC2      st.b    R0,R2+         ; [R2] = R0; R2 = R2 + 1
31 100A 4541      cmp.b   R0,R1          ; R0 = R1 ('Z')
32 100C 2003      beq     Done           ; If equal, goto Done
33 100E 40C8      add.b   $1,R0          ; No: R0 = R0 + 1 (next ASCII char)
34 1010 4C80      swap    R0,R0         ; NOP to test count
35 1012 3FFA      bra     Loop          ; Repeat loop
36         ; End of program
37 1014 6951 Done movl    '*',R1
38 1016 4549      cmp.b   R1,R1
```

```

39          ;
40          end    Main          ; End of program - Code begins at "Main"

```

Successful completion of assembly - 2P

** Symbol table **

Constants (Equates)

Name	Type	Value	Decimal
CAP_Z	CON	005A	90 PRI
CAP_A	CON	0041	65 PRI
SIZE	CON	001A	26 PRI

Labels (Code)

Name	Type	Value	Decimal
Done	REL	1014	4116 PRI
Loop	REL	1008	4104 PRI
Main	REL	1000	4096 PRI

Labels (Data)

Name	Type	Value	Decimal
Base	REL	0800	2048 PRI

Registers

Name	Type	Value	Decimal
R7	REG	0007	7 PRI
R6	REG	0006	6 PRI
R5	REG	0005	5 PRI
R4	REG	0004	4 PRI
R3	REG	0003	3 PRI
R2	REG	0002	2 PRI
R1	REG	0001	1 PRI
R0	REG	0000	0 PRI

.XME file: C:\Users\larry\OneDrive\Courses\ECED 3403 - Computer Architecture\XXM-23\XXM - Assembler\x64\Release\ArrayInit.xme

4.3 Second pass - .XME file

The .XME file is the executable file produced by the assembler from the .ASM file at the end of the successful second pass. The file consists of [S-records](#):

```
S01000004172726179496E69742E61736DED
S204080000F3
S11B1000086AD16A02604278C25C41450320C840804CFA3F51694945EF
S9031000EC
```

The assembler supports three types of S-record:

S0: The header record containing the name of the .ASM file from which the executable was obtained. The file name is the name of the file found in the directory; the full path is omitted. In this example, the S0-record fields are as follows:

S0: S0 record indicator
10: Length of record (address total bytes, 2; source module name total bytes, 13; and checksum total bytes, 1; for a total of 16 bytes).
0000: Address field, ignored.
4172726179496E69742E61736D: The name of the source module, encoded in ASCII.²
98: The checksum

S2: Data are stored in S2-records.³ In this example, there is one S2 record; its fields are:

S2: S2 record indicator
04: Length of the record (address bytes, data/instruction bytes, and checksum: 4 bytes)
0800: Address field – indicates location of first byte, as specified by the origin record (address #0800).
00: First data/instruction byte. Since this came from a BSS, the assembler produces only the first byte.
F3: The checksum of the length byte, address bytes, and data/instruction bytes.

S1: The second record is an S1 record, indicating it contains instructions. Its fields are:

S1: S1 record indicator
1B: Length of the record (address bytes, data/instruction bytes, and checksum: 27 bytes)
1000: Address field – indicates location of first byte, as specified by the origin record (address #1000)
086AD16A02604278C25C41450320C840804CFA3F51694945EF: The data/instruction bytes produced by the assembler from the original .ASM file. In this example, the first byte, #08, is stored in location #1000, the second, #6A, in location #1001, the third, #D1, in location #1002, and so on. The byte ordering used is *little-endian* (least-significant byte is stored in the first byte address and the most-significant byte is stored in the second byte address). In the case of #086A, #08 is stored first, then #6A. This pattern can be seen by comparing the output of the .LIS file with that of the .XME file.

² Decoded is `ArrayInit.asm`.

³ In ECED 3403, we are using the S2 records to hold data records with a 16-bit address field.

EF: The checksum of the length byte, address bytes, and data/instruction bytes. Note it is included in the data stream. When the bytes are being summed for the checksum, the final byte is the one's complement of the preceding byte.

S9: The starting address of the program is used when the program is loaded into XM23p's memory. If an address is specified as part of the **END** record in the .ASM file, it is used here. If the **END** record is omitted or there no starting address specified, the assembler defaults to zero. The data/instruction field is omitted. In this example, the program included a starting address, #1000:

S9: S9 record indicator

03: Length of the record (3 bytes)

1000: Starting address (#1000)

EC: The checksum of the length byte and address bytes.

5 Internals of the assembler

The XM23p assembler is two-pass: The first pass checks each non-commented record for validity, stores the label in the symbol table (if present), and increments the location counter, while the second pass rereads the file, generating the corresponding machine code for each non-comment record.

The assembler has a *location counter* that indicates where the next instruction or data value is to be loaded (i.e., resides) in memory. The location counter is incremented by 2 for all instructions and the number of bytes associated with the directive (if **BSS**, **BYTE**, or **WORD**). The location counter is incremented by 0 or 1 (**ALIGN**, depending on the current value of the location counter), it is incremented by the size of a reserved block of memory (**BSS**), and is assigned an entirely new value (if **ORG**).

5.1 First pass

The first pass repeats the following until end-of-file or an **END** directive is found:

1. Read the next record.
2. Comment records are ignored and the location counter is not to be incremented. *Labels* are stored in the symbol table along with the value of the location counter (i.e., the *Label's* address). A duplicate label is an error.
3. The next field must be an *Instruction*, *Directive*, *Comment*, or nothing. Anything else is an error.
4. If there is an *Instruction* or *Directive* and it requires an *Operand* field, the operand must exist and be correct. An invalid or unexpected *Operand* will cause an error diagnostic to be issued. If the *Instruction* is valid, its opcode is found, and the *Operand(s)* are determined from supplied *Value* or *Label*. The opcode and any values are combined according to the rules associated with the *Instruction's* format to create the machine instruction. The machine instruction is then emitted, along with the current value of the location counter. The location counter is incremented by 2. It is possible that the *Operand* is undefined until the second pass (if it is a *Label* that is a forward reference); in this case, the instruction is not emitted. This does not affect the location counter; it is simply incremented by the number of bytes required by the instruction or directive.
5. *Comments*, if they exist, are ignored.

If an error is found in a record, the subsequent records are processed until the end-of-file or **END**. The file and any errors are written to the .LIS file.

If there are no errors or forward references, the

5.2 Second pass

The second pass is performed if one or more errors are detected on the first pass or there was a forward reference. If not errors are found, the .LIS file is rewound for output of the listing file and the .XME file is opened.

The second pass repeats the following until end-of-file or an **END** directive is found:

1. Read the next record.
2. If the record is a comment, it should be ignored and step 1 repeated.
3. Ignore the *Label* if there is one (it was handled during the first pass).
4. A record containing an *Instruction* has the instruction and operand(s) extracted. The *Instruction's* corresponding opcode is found and the *Operand(s)* are determined from supplied *Value* or *Label*. The opcode and any values are combined according to the rules associated with the *Instruction's* format to create the machine instruction. The machine instruction is then emitted, along with the current value of the location counter. The location counter is incremented by 2.
5. If the record contains a *Directive*, directive-specific steps are performed; for example, **ORG** changes the location counter and **BYTE** or **WORD** writes the value to the .XME file.

If the Directive is **END**, the assembler stops reading the file. If errors were detected, the file is removed.

5.3 Notes

- Error messages are generated for missing *Operands* (the number depends upon the *Instruction* or *Directive*) or a supplied *Operand* (if *Operands* are not required by the *Instruction* or *Directive*).
- The MOVH instruction takes the most significant 8-bits of the 16-bit value. Using an 8-bit value will result in the assembler using a value of #00:

```
MOVH      #FFEE,R0    ; R0 = #FF
MOVH      #FF,R0      ; R0 = #00
```

You must use 16-bit values (e.g., *Labels*) to initialize the high-byte of a register using MOVH.

5.4 Errors in the assembler

If errors are found, please contact Dr. Hughes, supplying the .ASM program and any other supporting documentation to allow for the correction of the error.

6 XM23 Instruction Set

Table 2: Bit value definitions for XM-23p Instruction Set (see Table 5)

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

Table 3: Register and Constant values for R/C and SRC bits (see Table 5)

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

Table 4: Conditional execution codes and descriptions for CEX instruction (see Table 5)

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

Table 5: XM-23p Instruction Set

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