XM3 Assembler User's Guide

Larry Hughes, PhD

1 May 2020

Table of Contents

1 The Assembler

The X-Makina assembler is a two-pass assembler for the XM3 machine: the first pass builds the symbol table (from records in an XM3 assembly-language source module), while the second generates an executable load module (a file) for the XM3 machine. A list file is produced containing the records from the assembly-language file at the end of the first pass (if errors are detected during the first pass) or the end of the second pass (whether or not errors are detected during the second pass).

This document describes how to use the XM3 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 *record*s. 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*} 30

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). *Label*s 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 XM3's Instruction Set Architecture (ISA), found in *Introduction to the XM3 Instruction Set Architecture*. A summary of the ISA can be found in section 7.

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, *Instruction*s are **reserved words** and cannot be used as *Label*s or *Operand*s.

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,

[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.

although it can produce machine code. The directives currently supported by XM3 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.

`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.

`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.

*Directive*s are case insensitive. The assembler makes no distinction between the following directives:

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

Regardless of case, *Directive*s are reserved words and cannot be used as *Label*s or *Operand*s.

2.5 Operands

An *Operand* contains up to three *Value*s (separated by commas without 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*]

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

Numeric = ["**$**" + [*Unsigned* | *Signed*] | "**'** " + *Char* | "**#**" + *Hex*]

Unsigned = [**0 .. 65535**]

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

Char = [*Alphanumeric* | *Escaped*] + "**'** "

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

Escaped = "\" + *Alphanumeric*

The *Escaped Alphanumeric* value is limited to the following C-escape sequences ("Unknown" refers to an *Alphanumeric* character not in the list of supported characters):

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 ('?')	Unknown 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 *Label*s 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 XM3'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
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 XM3 assembler.

4.1 First pass errors - .LIS file

The first attempt at solving the problem is as follows:

```
;
; Sample XM3 program
; Initialize a block of memory to 'A' through 'Z'
; ECED 3403
; 1 May 2020
;
SIZE  equ   $26
CAP_A equ   'A'
CAP_Z equ   'Z'
;
; Start of data area
;
  org #800
Base  bss   SIZE        ; Reserve SIZE bytes
;
; Start of code area
;
  org #1000
;
Start
  movlz     CAP_A,R0    ; R0 = 'A'
  movlz     CAP_Z,R1    ; R1 = 'Z'
; R2 = Base (Base address to store characters)
  movl      Base,R2     ; R2 = LSByte(Base) or #00
  movh      Base,R2     ; R2 = MSByte(Base) or #08
;
Loop
  st.b      R0,R2+      ; [R2] = R0; R2 = R2 + 1
  cmp.b     R0,R1       ; R0 = R1 ('Z')
  cez       NE,$3,$0    ; If not equal, do next 3 instruction, otherwise skip
  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
  bra Done
;
  end Start       ; End of program - first executable address is "Start"
```

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:

```
XM3 Assembler - Version 3.0 (10 April 2020)
Input file name: ArrayInit.asm
Time of assembly: Fri 1 May 2020 14:02:22
  1               ;
  2               ; Sample XM3 program
  3               ; Initialize a block of memory to 'A' through 'Z'
  4               ; ECED 3403
  5               ; 1 May 2020
  6               ;
  7               SIZE  equ   $26
  8               CAP_A equ   'A'
  9               CAP_Z equ   'Z'
 10               ;
 11               ; Start of data area
 12               ;
 13                     org   #800
 14               Base  bss   SIZE        ; Reserve SIZE bytes
 15               ;
 16               ; Start of code area
 17               ;
 18                     org   #1000
 19               ;
 20               Start movlz CAP_A,R0    ; R0 = 'A'
 21                     movlz CAP_Z,R1    ; R1 = 'Z'
 22               ; R2 = Base (Base address to store characters)
 23                     movl  Base,R2     ; R2 = LSByte(Base) or #00
 24                     movh  Base,R2     ; R2 = MSByte(Base) or #08
 25               ;
 26               Loop
 27                     st.b  R0,R2+      ; [R2] = R0; R2 = R2 + 1
 28                     cmp.b R0,R1       ; R0 = R1 ('Z')
 29                     cez   NE,$3,$0    ; If not equal, do next 3 instruction, otherwise skip
            **** Expecting INST or DIR
 30                     add.b $1,R0       ; No: R0 = R0 + 1 (next ASCII char)
 31                     swap  R0,R0       ; NOP to test count
 32                     bra   Loop        ; Repeat loop
 33               ; End of program
 34               Done  movlz '*',R1
 35                     bra   Done
 36               ;
 37                     end   Start       ; End of program - first executable address is "Start"
```

```
First pass errors - assembly terminated

** Symbol table **
Name                                 Type  Value Decimal
Done                                 LBL   100C  4108
cez                                  LBL   100C  4108
Loop                                 LBL   1008  4104
Start                                LBL   1000  4096
Base                                 LBL   0800  2048
CAP_Z                                LBL   005A  90
CAP_A                                LBL   0041  65
SIZE                                 LBL   001A  26
R7                                   REG   0007  7
R6                                   REG   0006  6
R5                                   REG   0005  5
R4                                   REG   0004  4
R3                                   REG   0003  3
R2                                   REG   0002  2
R1                                   REG   0001  1
R0                                   REG   0000  0
```

The assembler indicates that an instruction or directive was expected on record 29 of the .ASM file (“`**** Expecting INST or DIR`”). The instruction “`cex`” is taken as a label by the assembler since it is not a valid instruction, meaning that the operand “`Done`” is causing the error.

4.2 Second pass - .LIS file

The invalid instruction can be corrected (changing “`cez`” to “`cex`”). By running the corrected assembler file through the assembler a second time, the assembler indicates that the assembly was successful.

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):

```
XM3 Assembler - Version 3.0 (10 April 2020)
Input file name: ArrayInit.asm
Time of assembly: Fri 1 May 2020 14:19:55
  1                   ;
  2                   ; Sample XM3 program
  3                   ; Initialize a block of memory to 'A' through 'Z'
  4                   ; ECED 3403
  5                   ; 1 May 2020
  6                   ;
  7                   SIZE  equ   $26
  8                   CAP_A equ   'A'
  9                   CAP_Z equ   'Z'
 10                   ;
 11                   ; Start of data area
 12                   ;
 13                         org   #800
 14   0800  0000  Base  bss   SIZE        ; Reserve SIZE bytes
 15                   ;
 16                   ; Start of code area
 17                   ;
 18                         org   #1000
 19                   ;
 20   1000  6A08  Start movlz CAP_A,R0    ; R0 = 'A'
 21   1002  6AD1        movlz CAP_Z,R1    ; R1 = 'Z'
 22                   ; R2 = Base (Base address to store characters)
 23   1004  6002        movl  Base,R2           ; R2 = LSByte(Base) or #00
 24   1006  7842        movh  Base,R2           ; R2 = MSByte(Base) or #08
 25                   ;
 26                   Loop
 27   1008  5CC2        st.b  R0,R2+            ; [R2] = R0; R2 = R2 + 1
 28   100A  4541        cmp.b R0,R1       ; R0 = R1 ('Z')
 29   100C  2458        cex   NE,$3,$0    ; If not equal, do next 3 instruction, otherwise skip
 30   100E  40C8        add.b $1,R0       ; No: R0 = R0 + 1 (next ASCII char)
 31   1010  4C80        swap  R0,R0       ; NOP to test count
 32   1012  23FA        bra   Loop        ; Repeat loop
 33                   ; End of program
 34   1014  6951  Done  movlz '*',R1
 35   1016  23FE        bra   Done
 36                   ;
 37                         end   Start       ; End of program - first executable address is "Start"
```

```
Successful completion of assembly

** Symbol table **
Name                                    Type  Value Decimal
Done                                    LBL   1014  4116
Loop                                    LBL   1008  4104
Start                                   LBL   1000  4096
Base                                    LBL   0800  2048
CAP_Z                                   LBL   005A  90
CAP_A                                   LBL   0041  65
SIZE                                    LBL   001A  26
R7                                      REG   0007  7
R6                                      REG   0006  6
R5                                      REG   0005  5
R4                                      REG   0004  4
R3                                      REG   0003  3
R2                                      REG   0002  2
R1                                      REG   0001  1
R0                                      REG   0000  0

.XME file: C:\Users\larry\OneDrive\Courses\ECED 3403 - 2020\XM3\XM3 - Test files\Completed XM3
tests\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*:

```
S0100000ArrayInit.asmED
S104080000F3
S11B1000086AD16A02604278C25C41455824C840804CFA235169FE231F
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 bytes [2], filename bytes [13], and checksum [1]: 16 bytes)
`0000`: Address field
`ArrayInit.asm`: The name of the file
`ED`: The checksum

S1: Data and instructions are stored in S1-records. In this example, there are two S1 records.

The first record's fields are:

`S1`: S1 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.

The second record's 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`)
`086AD16A02604278C25C41455824C840804CFA235169FE23`: 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.
`1F`: The checksum of the length byte, address bytes, and data/instruction bytes.

S9: The starting address of the program, used when the program is loaded into XM3'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 XM3 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. *Label*s 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. It is possible that the *Operand* is undefined until the second pass (if it is a *Label* that is a forward reference). This does not affect the location counter; it is simply incremented by the number of bytes required by the instruction or directive.
5. *Comment*s, 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.

5.2 Second pass

The second pass is performed if one or more errors are detected on the first pass. 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 *Operand*s (the number depends upon the *Instruction* or *Directive*) or a supplied *Operand* (if *Operand*s are not required by the *Instruction* or *Directive*).

6 Known limitation

The assembler has the following known limitation as of 1 May 2020:

1. 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
```

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.

7 XM3 Instruction Set

Bit value definitions for XM3 Instruction Set (last page)

0	1	Instruction opcode bit values
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 below)
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).

Register and Constant values for R/C and SRC bits

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

Conditional execution codes and descriptions for CEX instruction

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

XM3 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	BRA	Unconditional branch (branch always)
0	0	1	0	0	1	C	C	C	C	T	T	T	F	F	F	CEX	Conditional execution
0	0	1	0	1	0	0	0	0	0	0	0	0	PR	PR	PR	SETPRI	Set current priority
0	0	1	0	1	0	0	0	0	0	0	1	SA	SA	SA	SA	SVC	Control passes to address specified in vector [SA]
0	0	1	0	1	0	0	0	0	0	1	V	SLP	N	Z	C	SETCC	Set PSW bits (1 = set)
0	0	1	0	1	0	0	0	0	1	0	V	SLP	N	Z	C	CLRCC	Clear PSW bits (1 = clear)
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 with Carry: 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 with Carry: 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 + 1)
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	BIT	Bit test: DST & SCR/CON
0	1	0	0	1	0	0	1	R/C	W/B	S/C	S/C	S/C	D	D	D	BIC	Bit clear: DST = DST & ~SRC/CON
0	1	0	0	1	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	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	1	W/B	0	0	0	D	D	D	RRC	Rotate DDD right (1 bit) through carry
0	1	0	0	1	1	1	0	0	0	0	0	0	D	D	D	SWPB	Swap bytes in DDD
0	1	0	0	1	1	1	0	0	0	0	0	1	D	D	D	SXT	Sign-extend byte to word in DDD
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