X-Makina Assembler – User's Guide

Larry Hughes, PhD

24 May 2018 (Revised)

1 The Assembler

The X-Makina assembler is a two-pass assembler for the X-Makina machine: the first pass builds the symbol table (from records in an X-Makina assembly-language file), while the second generates an executable module (a file) for the X-Makina 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).

2 The assembly language file

An assembly language file 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 to be *free-format*, meaning that it has no fixed fields. All records have the same format, defined as follows (**bold** indicates terminal symbols):

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 X-Makina's Instruction Set Architecture (ISA), found in *Introduction to the X-Makina Instruction Set Architecture* (http://lh.ece.dal.ca/eced3403/X-Makina_ISA.pdf). 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, although it can produce machine code. The directives currently supported by X-Makina 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) required by the *Instruction* or *Directive*. It is defined as:

Operand = *Value* + ("**,**" + *Operand*)

A *Value* is either a numeric value or a label:

Value = [*Numeric* | *Label*]

Numeric and *Label* values are distinguished by the use of the '**#**' symbol, with '**#**' 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 .. +32767**]

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:

Character	Converted valued	Meaning
'\b'	0x08	BS - Backspace
'\t'	0x09	TAB
'\n'	0x0a	Linefeed, Newline
'\r'	0x0d	Carriage return
'\0'	0x00	NUL
'\\'	0x5c	Backslash
'\''	0x27	Single quote
'\"'	0x3f	Double quote
‘\Unknown’	'?'	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 an end with the single quote character “**’**”.
- Hexadecimal numbers are prefixed with “**$**”.
- Hexadecimal numbers cannot be signed.
- Unsigned values can be associated with the “**+**” sign.
- 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 X-Makina’s eight CPU registers (R0, R1, R2, R3, R4, R5, R6, and R7). Other symbols can be equated as registers; for example:

```
LR    equ   R4          ; LR is equated with R4
r5    equ   R5          ; r5 is equated with R5
```

```
SP    equ   r5          ; SP is equated with r5 (which is equated with R5)
PSW   EQU   R6
pc    equ   R7

…

      mov   LR,pc       ; Subroutine return
      mov   R4,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 $0000) could be solved using the X-Makina 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
; 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
      org   #0
Base  bss   SIZE        ; Reserve SIZE bytes
; Start of code area
      org   #$100
Start movlz CAP_A,R0    ; R0 = 'A'
      movlz CAP_Z,R1    ; R1 = 'Z'
      movlz Base,R2     ; R2 = Base (Base address for characters
;
Loop
      st.b  R0,R2+      ; [R2] = R0; R2 = R2 + 1
      cmp.b R0,R1       ; R0 = R1 ('Z')
      beg   Done        ; Yes: Goto Done
      add.b #1,R0       ; No: R0 = R0 + 1 (next ASCII char)
      bal   Loop        ; Repeat loop
; End of program
Done  bal   Done        ; Infinite loop to "stop" the program
;
      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:

```
X-Makina Assembler - Version 1.1 (19 May 2018)
.ASM file: C:\Users\Larry\OneDrive\Courses\ECED 3403\2018\Sample code\Test1\Debug\18 05 15 - ArrayInit.asm

  1               ;
  2               ; Sample X-Makina program
  3               ; Initialize a block of memory to 'A' through 'Z'
  4               ; L. Hughes
  5               ; 15 May 18: Comment correct R1 = 'Z'
  6               ;  1 May 18: Initial release
  7               ;
  8               SIZE  equ   #26
  9               CAP_A equ   #'A'
 10               CAP_Z equ   #'Z'
 11               ; Start of data area
 12                     org   #0
 13               Base  bss   SIZE        ; Reserve SIZE bytes
 14               ; Start of code area
 15                     org   #$100
 16               Start movlz CAP_A,R0    ; R0 = 'A'
 17                     movlz CAP_Z,R1    ; R1 = 'Z'
 18                     movlz Base,R2     ; R2 = Base (Base address for characters
 19               ;
 20               Loop
 21                     st.b  R0,R2+            ; [R2] = R0; R2 = R2 + 1
 22                     cmp.b R0,R1       ; R0 = R1 ('Z')
 23                     beg   Done        ; Yes: Goto Done
            **** Expecting INST or DIR
 24                     add.b #1,R0       ; No: R0 = R0 + 1 (next ASCII char)
 25                     bal   Loop        ; Repeat loop
 26               ; End of program
 27               Done  bal   Done        ; Infinite loop to "stop" the program
 28               ;
 29                     end   Start       ; End of program - first executable address is "Start"

First pass errors - assembly terminated

** Symbol table **
Name        Type  Value Decimal
Done        LBL   010A  266
beg         LBL   010A  266
Loop        LBL   0106  262
```

```
Start       LBL   0100  256
Base        LBL   0000  0
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 22 of the .ASM file ("`**** Expecting INST or DIR`"). The instruction "beg" is actually taken as a label since it is not a valid instruction, meaning that the operand "Done" is actually causing the error.

4.2 Second pass - .LIS file

The invalid instruction can be corrected (changing "beg" to "beq" or branch-if-equal). 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):

```
X-Makina Assembler - Version 1.1 (19 May 2018)
.ASM file: 18 05 15 - ArrayInit.asm

  1                 ;
  2                 ; Sample X-Makina program
  3                 ; Initialize a block of memory to 'A' through 'Z'
  4                 ; L. Hughes
  5                 ; 15 May 18: Comment correct R1 = 'Z'
  6                 ;  1 May 18: Initial release
  7                 ;
  8                 SIZE  equ   #26
  9                 CAP_A equ   #'A'
 10                 CAP_Z equ   #'Z'
 11                 ; Start of data area
 12                       org   #0
 13    0000  0000   Base  bss   SIZE        ; Reserve SIZE bytes
 14                 ; Start of code area
 15                       org   #$100
 16    0100  9A08   Start movlz CAP_A,R0    ; R0 = 'A'
 17    0102  9AD1         movlz CAP_Z,R1          ; R1 = 'Z'
 18    0104  9802         movlz Base,R2           ; R2 = Base (Base address for characters
 19                 ;
 20                 Loop
 21    0106  8942         st.b  R0,R2+            ; [R2] = R0; R2 = R2 + 1
 22    0108  6841         cmp.b R0,R1       ; R0 = R1 ('Z')
 23    010A  2402         beq   Done        ; Yes: Goto Done
 24    010C  60C8         add.b #1,R0       ; No: R0 = R0 + 1 (next ASCII char)
 25    010E  3FFB         bal   Loop        ; Repeat loop
 26                 ; End of program
 27    0110  3FFF   Done  bal   Done        ; Infinite loop to "stop" the program
 28                 ;
 29                       end   Start       ; End of program - first executable address is "Start"

Successful completion of assembly

** Symbol table **
Name        Type  Value Decimal
Done        LBL   0110  272
Loop        LBL   0106  262
Start       LBL   0100  256
Base        LBL   0000  0
```

```
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\2018\Sample code\Test1\Debug\18 05 15 - 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*:[1]

```
S01B000018 05 15 - ArrayInit.asm01
S104000000FB
S1150100089AD19A0298428941680224C860FB3FFF3F08
S9030100FB
```

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
`1B`: Length of record (address bytes, filename bytes, and checksum: 27 bytes)
`0000`: Address field
`18 05 15 - ArrayInit.asm`: The name of the file
`01`: 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)
`0000`: Address field – indicates location of first byte, as specified by the origin record (address `#0`).
`00`: First data/instruction byte. Since this came from a BSS, the assembler produces only the first byte.
`FB`: The checksum of the length byte, address bytes, and data/instruction bytes.

The second record's fields are:

`S1`: S1 record indicator
`15`: Length of the record (address bytes, data/instruction bytes, and checksum: 21 bytes)
`0100`: Address field – indicates location of first byte, as specified by the origin record (address `#$100`)
`089AD19A0298428941680224C860FB3FFF3F`: The data/instruction bytes produced by the assembler from the original .ASM file. In this example, the first byte, `$08`, is stored in location `$0100`, the second, `$9A`, in location `$0101`, the third, `$D1`, in location `$0102`, 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 `$9A08`, `$08` is stored first, then `$9A`. This pattern can be seen by comparing the output of the .LIS file with that of the .XME file.

[1] For a description of S-Records, see http://www.amelek.gda.pl/avr/uisp/srecord.htm.

`08`: 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 X-Makina'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, `$0100`:

`S9`: S9 record indicator
`03`: Length of the record (3 bytes)
`0100`: Starting address ($0100)
`FB`: The checksum of the length byte and address bytes.

5 Internals of the assembler

The X-Makina 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 limitations

The assembler has the following known limitation as of 10 May 2018:

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
```

2. It should be remembered that the assembler treats numeric values and labels in slightly different ways:

   a) As a value to be assigned to a register:

```
; Assigning the number 32 to register R3 using numeric values and labels
; As a numeric value:
       MOVLZ #32,R3            ; The value must be prefixed with a '#'
; As a label:
LIMIT  EQU         #32         ; The label
       MOVLZ LIMIT,R3    ; The label must not be prefixed with a '#'
```

   b) As a branch target address, only labels are permitted and the '#' cannot be not used:

```
Loop    …
```

```
        BAL   Loop
; This is not permitted:
        BAL   #$100
; However, an equated label can be used:
Loop2   EQU   #$1000
        …
        BAL   Loop2
```

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 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. Word = 0 and Byte = 1.

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 1**). The constants (0, 1, 2, 4, 8, $00FF, $FF00 and -1) are encoded as 000, 001, 010, 011, 100, 101, 110, and 111, respectively.

Table 1: Register and Constant values from R/C and SRC bits

R/C value		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	*$00FF*	5
R6/PSW	*$FF00*	6
R7/PC	-1	7

Table 2: 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