CpE 390: Microprocessor Systems

Lecture Note 2

68HC12 Assembly Programming (1)

Three Sections of a 68HC12 Assembly Program

1. Assembler Directives

- Define data and symbol
- Reserve and initialize memory locations
- Set assembler and linking condition
- Specify output format
- Specifies the end of a program.

2. Assembly Language Instructions

- 68HC12 instructions

3. Comments

- Explain the function of a single or a group of instructions

Program Structure

Example Code:

```
                ORG   $800
results         RMB   4                ; reserve 4 bytes for results
incre_data      EQU   $30                ; use symbol to represent data
data            DW  $1122                ; define a word

                ORG  $900                 ; program begins
                LDD data                 ; load $1122 into D
                SUBD #10
                STD results
                ADDA  #incre_data
                STAA  results+2
                END                      ; program ends
```

Fields of a 68HC12 Instruction

1. **Label field**

- Is optional
- Starts with a letter and followed by letters, digits, or special symbols (_ or .)
- Can start from any column if ended with ":" (not true for Motorola freeware as11)
- Must start from column 1 if not ended with ":"

2. **Operation field**

- Contains the mnemonic of a machine instruction or an assembler directive
- Is separated from the label by at least one space

3. **Operand field**

- Follows the operation field and is separated from the operation field by at least one space
- Contains operands for instructions or arguments for assembler directives

4. **Comment field**

- Any line starts with a ***** or **;** is a comment
- Is separated from the operand and operation field for at least one space
- Is optional

Identify the Four Fields of an Instruction

Example

loop ADDA #$40 ; add 40 to accumulator A

(1) “loop” is a label
(2) “ADDA” is an instruction mnemonic
(3) “#$40” is the operand
(4) “add #$40 to accumulator A” is a comment

Assembler Directives

1. END

- Ends a program to be processed by an assembler
- Any statement following the END directive is ignored

2. ORG

- The assembler uses a location counter to keep track of the memory location where the next machine code byte should be placed.
- This directive sets a new value for the location counter of the assembler

The sequence

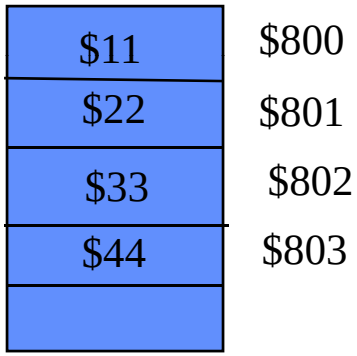
```
ORG   $1000
LDAB  #$FF
```

will put the opcode byte for the instruction LDAB #$FF at location $1000.

db (define byte)
dc.b (define constant byte)
fcb (form constant byte)

- These three directives define the value of a byte or bytes that will be placed at a given location.
- These directives are often preceded by the **org** directive.
- For example,

```
            org  $800
array       db  $11,$22,$33,$44
```



dw (define word)
dc.w (define constant word)
fdb (form double bytes)

- Define the value of a word or words that will be placed at a given location.
- The value can be specified by an expression.
- For example,

```
vec_tab     dw          $1234, abc-20
```

fcc (form constant character)

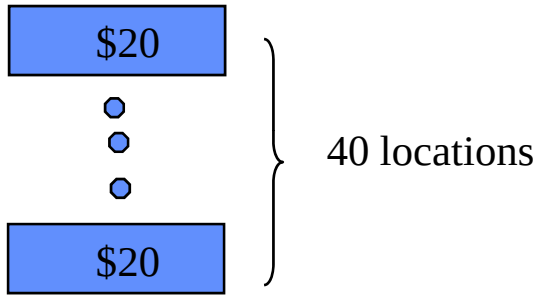
- Used to define a string of characters (a message).
- The first character (and the last character) is used as the delimiter.
- The last character must be the same as the first character.
- The delimiter must not appear in the string.
- The space character cannot be used as the delimiter.
- Each character is represented by its ASCII code.
- For example,

```
msg     fcc   "Please enter 1, 2 or 3:"
msg     fcc   (Please enter 1, 2 or 3:(
msg     fcc   “Please enter 1, 2 or 3:”
```

fill (fill memory)

- This directive allows the user to fill a certain number of memory locations with a given value.
- The syntax is **fill value,count**
- For example,

```
space_line  fill   $20,40
```



ds (define storage)

rmb (reserve memory byte)

ds.b (define storage bytes)

- Each of these directives reserves a number of bytes given as the arguments to the directive.
- For example,

```
buffer      ds    100     ; reserve 100 bytes
```

ds.w (define storage word)

rmw (reserve memory word)

- Each of these directives increments the location counter by the value indicated in the number-of-words argument multiplied by two.
- For example,

```
dbuf        ds.w  20
```

reserves 40 bytes starting from the current location counter.

equ (equate)

- This directive assigns a value to a label.
- Using this directive makes our program more readable.
- Examples:

```
arr_cnt     equ   100
oc_cnt      equ   50
```

loc

- This directive increments and produces an internal counter used in conjunction with the backward tick mark (`). By using the **loc** directive and the ` mark you can write program segments like the following example, without thinking up new labels:

```
            loc
            ldaa  #2                                    ldaa  #2
loop`       deca              same as   loop001         deca
            bne   loop`                                 bne  loop001
            loc                                         loc
loop`       brclr  0,x,$55,loop`          loop002       brclr  0,x,$55,loop002
```

Software Development Process

- **Problem definition**: Identify what should be done.
- **Develop the algorithm**. Algorithm is the overall plan for solving the problem at hand.
- An algorithm is often expressed in the following format:

  Step 1

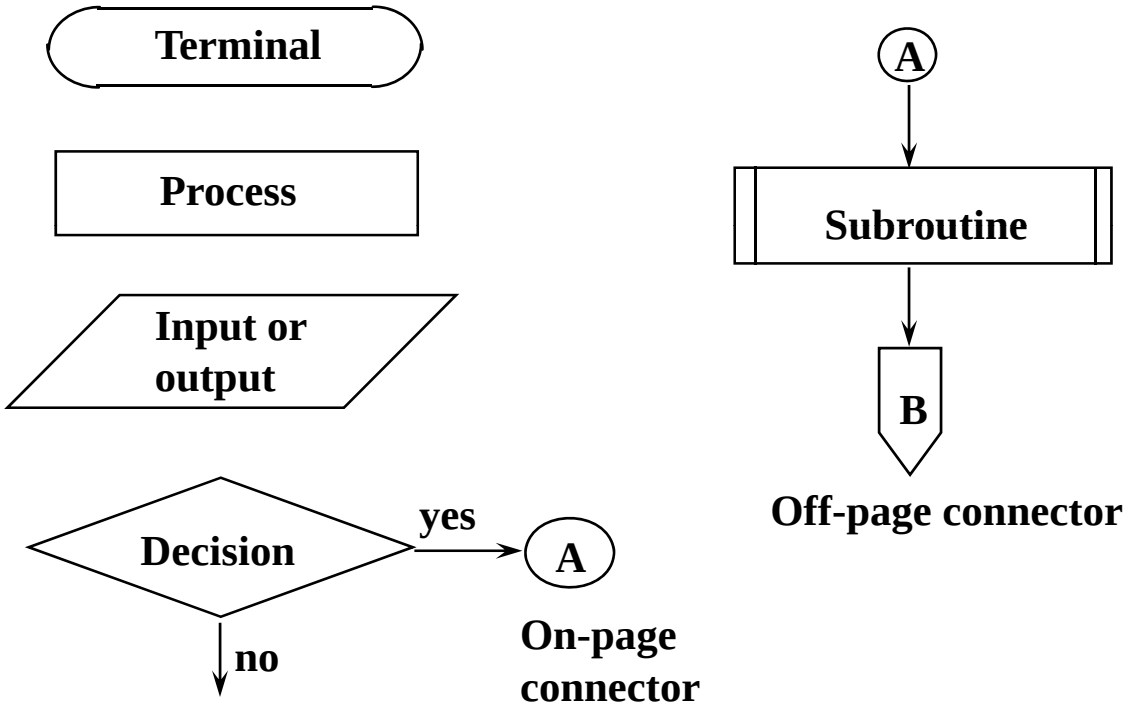
  …

  Step 2

  …

- Another way to express overall plan is to use **flowchart**.
- **Programming.** Convert the algorithm or flowchart into **programs**.
- **Program Testing**
- **Program maintenance**

Symbols of Flowchart



Programs to do simple arithmetic

Example 2.3 Write a program to add the values of memory locations at $800, $801, and $802, and save the result at $900.

Solution:

Step 1

A ⇐ m[$800]

Step 2

A ⇐ A + m[$801]

Step 3

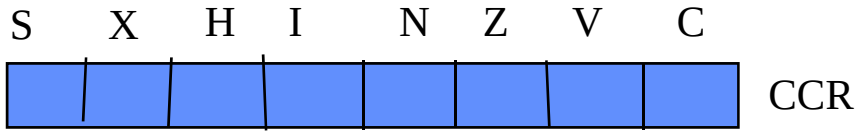
A ⇐ A + m[$802]

Step 4

$900 ⇐ A

```
ORG     $1000
LDAA    $800
ADDA    $801
ADDA    $802
STAA    $900
END
```

CCR flags affected by arithmetic computation



Single precision: operands < 16 bits

Using 8-bit register

Ex 1:

```
  $20
+$30
-------
  $50
```

```
LDAA #$20
ADAA #$30
```

Ex 2:

```
  $93
+$8B
-------
$1, 1E
```

```
LDAA #$93
ADAA #$8B
```

Set C (carry) bit in CCR to 1
Set V (overflow) bit in CCR to 1

Using 16-bit register

Ex 3:

```
  $A200
+$7000
----------
$1 1200
```

```
LDD  #$A200
ADDD #$7000
```

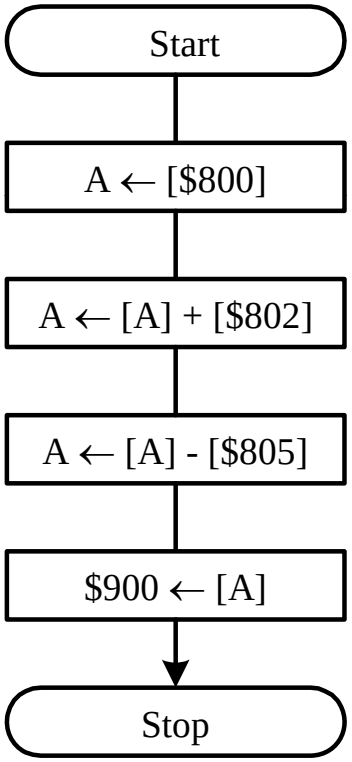
Set C bit in CCR to 1
Set V bit in CCR to 1

Multiprecision: operands > 16 bits

```
  $24A4E8
+$D5B456
--------------
  $FA593E
```

Example 2.4 Write a program to subtract the contents of the memory location at $805 from the sum of the memory locations at $800 and $802, and store the difference at $900.

Solution:




```
org     $1000
ldaa    $800
adda    $802
suba    $805
staa    $900
end
```

Figure 2.2 Logic flow of program 2.4

Example 2.6 Write a program to add two 16-bit numbers that are stored at $800-$801 and $802-$803 and store the sum at $900-$901.

Solution:

```
        org     $1000
        ldd     $800
        addd    $802
        std     $900
        end
```

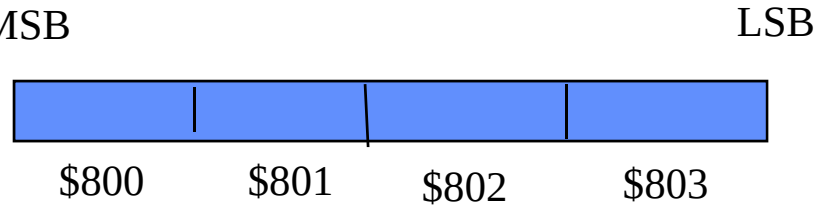
Example 2.7 Write a program to add two 4-byte numbers that are stored at $800-$803 and $804-$807, and store the sum at $810-$813.

Solution:
Addition starts from the LSB and proceeds toward MSB.

```
org     $1000
ldd     $802    ; add and save the least significant two bytes
addd    $806    ;       “
std     $812    ;       “

ldaa    $801    ; add and save the second most significant bytes
adca    $805    ;       “
staa    $811    ;       “

ldaa    $800    ; add and save the most significant bytes
adca    $804    ;       “
staa    $810    ;       “
end
```

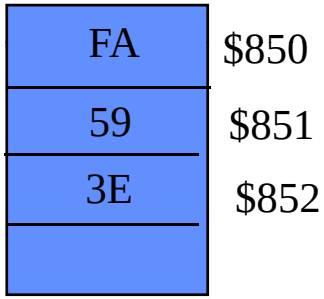


Multiprecision

Operands > 16 bits

```
    $24A4E8
+   $D5B456
------------------
    $FA593E
```

```
ORG  $800
LDD   #$A4E8
ADDD  #$B456
STD   $851
LDAA  #$24
ADCA  #$D5
STAA   $850
END
```



Subtraction

- Similar to ADD
- Multiprecision SUB also use C flag, but to hold the “borrow” bit
  - i.e., set C to 1, if the operand borrows a 1 during subtraction
- Has subtract with carry command
  - SBCA opr
  - SBCB opr

BCD (Binary-Coded-Decimal) Numbers and Addition

- Each digit is encoded by 4 bits (digit number from 0 to 9)
  - ex: 0010 0100 0011 1001 -> 2439
- Two digits are packed into one byte

```
  $12
 +$34
------
  $46
```

Compare to decimal ops

```
  12
 +34
-----
  46
```

No adjustment needed

```
  $12
 +$08
------
  $1A
```

```
  12          $1A
 +08          +$06
-----   ->   -------
  20          $20
```

Problem: A is not between 0-9
Solution: add $6 to every sum Digit that is greater than 9

```
  $29 with half carry
 +$49
------
  $72
```

```
  29          $72
 +49          +$06
-----   ->   ------
  78          $78
```

Problem: final result does not match due to half carry
Solution: add $6 to every sum digit that has a carry of one to the next higher digit

BCD Numbers

- 68HC12 provides DAA (Decimal Adjust Accumulator A) instruction
- It can be applied after ADDA, ADCA, ABA.
- Advantage
  - Simplify I/O conversion
- Disadvantage
  - Complexity of arithmetic processing

Example:

```
LDAA  $800
ADDA $801
DAA
STAA  $02
```

Multiplication and Division: Single Precision

Table 2.1 Summary of 68HC12 multiply and divide instructions

Mnemonic	Function	Operation
EMUL	unsigned 16 by 16 multiply	(D) × (Y) → Y:D
EMULS	signed 16 by 16 multiply	(D) × (Y) → Y:D
MUL	unsigned 8 by 8 multiply	(A) × (B) → A:B
EDIV	unsigned 32 by 16 divide	(Y:D) ÷ (X) quotient → Y remainder → D
EDIVS	signed 32 by 16 divide	(Y:D) ÷ (X) quotient → Y remainder → D
FDIV	16 by 16 fractional divide	(D) ÷ (X) → X remainder → D
IDIV	unsigned 16 by 16 integer divide	(D) ÷ (X) → X remainder → D
IDIVS	signed 16 by 16 integer divide	(D) ÷ (X) → X remainder → D

Multiplication and Division: Single Precision

- Unsigned 8 x 8: MUL D <- [A] x [B]
- Unsigned 16 x 16: EMUL Y:D <- [D] x [Y]
- Signed 16 x 16: EMULS Y:D <- [D] x [Y]

Y: higher word D: lower word

Division is similar to multiplication, see Table 2.1.

Example:

```
LDAA   #$45
LDAB   #$56
MUL
STD    $860
END
```

```
LDD #$5824
LDX #$1200
IDIV
END
```

Example 2.10 Write an instruction sequence to multiply the 16-bit numbers stored at $800-$801 and $802-$803 and store the product at $900-$903.

Solution:

```
ldd     $800
ldy     $802
emul            ; (D) x (Y) -> Y:D
sty     $900
std     $902
```

Example 2.11 Write an instruction sequence to divide the 16-bit number stored at $820-$821 into the 16-bit number stored at $805-$806 and store the quotient and remainder at $900 and $902, respectively.

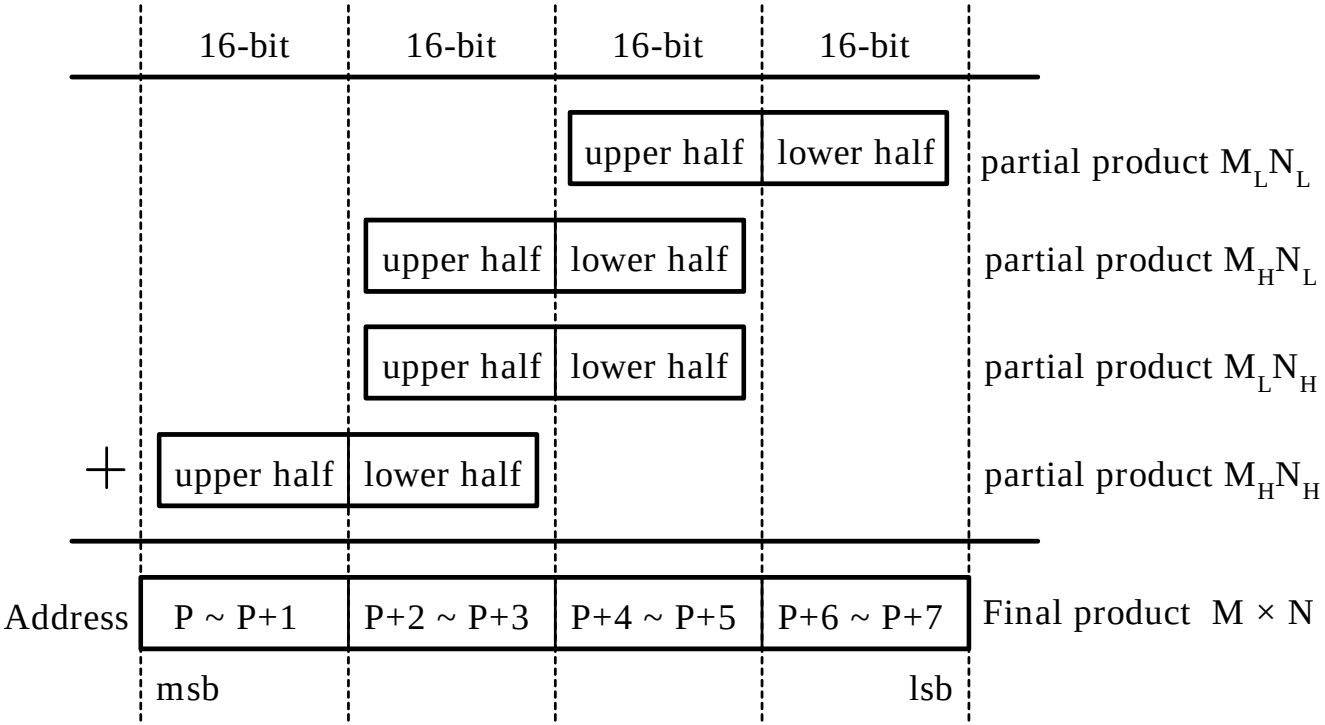
Solution:

```
ldd     $805
ldx     $820
idiv            ; (D) / (X) -> X, remainder ->D
stx     $900    ; store the quotient
std     $902    ; store the remainder
```

Illustration of 32-bit by 32-bit Multiplication: Multi-precision

- Two 32-bit numbers M and N are divided into two 16-bit halves

$M = M_H M_L \quad N = N_H N_L$



65 x 48

```
   65
X  48
---------
   40
  48
  20
 24
---------
 3120
```

Note: msb stands for most significant byte and lsb for least significant byte

Figure 2.3 Unsigned 32-bit by 32-bit multiplication

Example 2.12 Write a program to multiply two unsigned 32-bit numbers stored at M~M+3 and N~N+3, respectively and store the product at P~P+7.

Solution:

```
        org     $800
M       ds.b    4           ; reserve to hold the multiplicand
N       ds.b    4           ; reserve to hold the multiplier
P       ds.b    8           ; reserve to hold the product

        org     $1000
        ldd     M+2
        ldy     N+2
        emul                ; compute M_L N_L
        sty     P+4         ; save the upper 16 bits
        std     P+6         ; save the lower 16 bits
        ldd     M
        ldy     N
        emul                    ; compute M_H N_H
        sty     P
        std     P+2
        ldd     M
        ldy     N+2
        emul                    ; compute M_H N_L
```

```
; add M_H N_L to memory locations P+2~P+5
        addd    P+4
        std     P+4
        tfr     Y,D         ; transfer Y to D
        adcb    P+3
        stab    P+3
        adca    P+2
        staa    P+2
; propagate carry to the most significant byte
        ldaa    P+1
        adca    #0              ; add carry to the location at P+1
        staa    P+1             ;           “
        ldaa    P               ; add carry to the location at P
        adca    #0              ;           “
        staa    P               ;           “
; compute M_L N_H
        ldd     M+2
        ldy     N
        emul
```

```
; add M_L N_H to memory locations P+2 ~ P+5
        addd    P+4
        std     P+4
        tfr     Y,D
        adcb    P+3
        stab    P+3
        adca    P+2
        staa    P+2
; propagate carry to the most significant byte
        clra
        adca    P+1
        staa    P+1
        ldaa    P
        adca    #0
        staa    P
        end
```

Homework #2

- See course website: http://390.revan.us
  - click homework tab
- Please submit a hard copy