# Chapter 2

# 68HC12 Assembly Programming

# Sample Program

Directive : Tells loader where to put program

```
                 ORG $4000

Label            Opcode   Operand      Comment
main:            LDAA     $800         ; A = m[$800]
                 ADDA     $801         ; A = A + m[$801]
                 ADDA     $802         ; A = A + m[$802]
                 STAA     $805         ; m[$805] = A

                 END
```

Directive : Tells assembler where program finished

# Assembler Directives

**-** Commands used by development tools when building program

**Example: END**

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

**Example: ORG**

- Assembler uses **location counter** to keep track of memory location where next machine code byte should be placed.
- Sets a new value for the location counter

## Data Declarations

Write assembly program to implement the following C code.

byte aa, bb, cc, dd;

dd = aa + bb – cc;

```
      org $800
aa rmb 1
bb rmb 1
cd rmb 1
dd rmb 1
      org $4000
ldaa  aa    ; ldaa $800
adda  bb    ; adda $801
suba  cc    ; suba $802
staa  dd    ; staa $803
end
```

Directive : Reserve Memory Byte

**Data Declaration Directives**

- Used to declare variables.
- Often preceded by **`org`** directive.
- Many variations: Choose to suit the job
  - Two sets for byte and word

# Directives to Simply Declare a Variable

- Simply allocate space.

| | Byte | | Word | |
|---|---|---|---|---|
| Define Storage | **ds** | num | **ds.w** | num |
| Reserve Memory | **rmb** | num | **rmw** | num |

**Examples :**

```
buffer ds 100   ; Reserves 100 bytes at current location
dbuf  ds.w  20  ; Reserve 40 bytes at current location.
```

## Directives to Declare <u>and Initialize</u> a Variable

| | Byte | | Word | |
|---|---|---|---|---|
| Define byte | **DB** | value | **DW** | value |
| Define Constant Byte | **DC.B** | value | **DC.W** | value |
| Form Constant Byte | **FCB** | value | **FDB** | value |

**Examples :**

```
      org $800
variable    dw    43
array       db    $11,$22,$33,$44
num         db    num-array
```

**fcc (form constant character)**

- Used to define a string of characters (a message).
- First (and last) character used as the delimiter.
- Last character must be the same as the first character.
- Delimiter must not appear in the string.
- 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:”
```

**fill (fill memory)**

- Allows the user to fill a certain number of memory locations with a given value.
- Syntax: **`var_name   fill value, count`**
- For example,

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

## equu (equate)

- Assigns a value to a label.
- Using this directive makes our program more readable.

-Examples:

```
NUM_ELEMENTS equ      100
DEFAULT      equ      0
array        fill     DEFAULT, NUM_ELEMENTS
```

## Multiple Precision Arithmetic

- HC12: limited to 16-bit operations.
- Problem may involve larger-size data.

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

```
      org $800
x     rmw 2
y     rmw 2
      org $810
z     rmb 4
```

x = \$ 88 33 9922
y = \$ 11 88 8855

---

z = \$ 99 BC 2177

**Multiple-Precision Solution:**

Addition starts from the LSB and proceeds toward MSB.

```
     org  $4000
ldd  x+2  ; add and save the two LSWs
addd y+2  ;   “
std  z+2  ;   “

ldaa x+1  ; add and save the second MSB
adca y+1  ;   “
staa z+1  ;   “

ldaa x    ; add and save the MSB
adca y    ;   “
staa z    ;   “
     end
```

# Assembling and running programs

(Demo)

# Multiplication and Division

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)<br>quotient → Y<br>remainder → D |
| EDIVS | signed 32 by 16 divide | (Y:D) ÷ (X)<br>quotient → Y<br>remainder → D |
| IDIV | unsigned 16 by 16 integer divide | (D) ÷ (X) → X<br>remainder → D |
| IDIVS | signed 16 by 16 integer divide | (D) ÷ (X) → X<br>remainder → D |

**Example 2.10** Write instruction sequence to multiply 16-bit numbers stored at \$800-\$801 and \$802-\$803. Store product at \$900-\$903.

**Solution:**

```
ldd     $800
ldy     $802
emul
sty     $900
std     $902
```

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

**Solution:**

```
ldd     $805
ldx     $820
idiv
stx     $900  ; store the quotient
std     $902  ; store remainder
```

**Example 2.13** Write a program to convert a 16-bit number stored at $800-$801 into its ASCII representation as a decimal number.

**Solution:**

- Binary number converted to decimal ASCII: repeated division by 10.
- Largest 16-bit binary number = 65535 (five decimal digits).
- First division by 10: least significant digit; second division by 10: second least significant digit, etc.

```
     org     $800
data dc.w    12345    ; data to be tested
     org     $900
result       ds.b 5   ; reserve bytes to store the result

     org     $1000
ldd  data
ldy  #result
ldx  #10
idiv
addb #$30    ; convert the digit into ASCII code
stab 4,Y     ; save the least significant digit
xgdx
ldx  #10
```

```
idiv
addb #$30
stab 3,Y      ; save second to least significant digit
xgdx
ldx  #10
idiv
addb #$30
stab 2,Y      ; save middle digit
xgdx
ldx  #10
idiv
addb #$30
stab 1,Y         ; save second most significant digit
xgdx
addb #$30
stab 0,Y         ; save most significant digit
end
```

# Program Loops - Branches

Four **types** of branch instructions:

-**Unary (unconditional) branch:** always execute

```
BRA   $1000
```

- **Simple branches:** branch taken when specific bit of CCR in a specific status

```
BCC   $1000
```

- **Unsigned branches:** branches taken when comparison or test of unsigned numbers results in specific combination of CCR bits

```
BHI   $1000
```

- **Signed branches:** branches taken when comparison or test of signed quantities results in a specific combination of CCR bits

```
BGT   $1000
```

**Example : Conditional Branches**

Suppose A contains $7F:

Unsigned Scenario

```
SUBA  #$80
BHI Bigger
```

Signed Scenario

```
SUBA #$80
BGT  Bigger
```

In each scenario, is the jump taken? Why?

Programmer MUST know how binary values are to be interpreted!
(e.g. value in AX above)

Branch Instructions use Relative Mode to encode the target

```
1:          =00001000       org $1000
2:     1000 B6 0800         ldaa $800
3:     1003 20 FB           bra  $1000
4:     1005                 END
```

8-bit relative offset

Let's execute-by-hand :

**Fetch**: **PC** = $1000 Instruction Register = ????????

**Execute**: **PC** = $1003 Instruction Register = B6 0800

**Fetch**: **PC** = $1003 Instruction Register = B6 0800

**Execute**: **PC** = $1005 Instruction be be executed = 20 FB

**Fetch**: **PC** = $1000 Instruction to be executed = 20 FB

## Relative Mode

- Used only by branch instructions.
- Used even when programmer use labels for the target.
    - Assembler calculates actual branch offset (distance) from the instruction that follows the branch instruction.

```
minus . . .
          bmi   minus
```

- Two **categories** of Branches

1. **Short Branches:** in the range of -128 ~ +127 bytes
    - 8-bit opcode and a signed 8-bit offset.
2. **Long Branches:** in the range of -32768 ~ +32767.
    - 8-bit opcode and a signed 16-bit offset. Range of long relative mode is -32768 ~ +32767.

# Compare and Test Instructions

- Condition flags need to be set up before simple and conditional branch instruction are to be executed.

Table 2.4 Summary of compare and test instructions

| Mnemonic | Function | Operation |
|---|---|---|
| | **Compare instructions** | |
| CBA | Compare A to B | (A) - (B) |
| CMPA | Compare A to memory | (A) - (M) |
| CMPB | Compare B to memory | (B) - (M) |
| CPD | Compare D to memory | (D) - (M:M+1) |
| CPS | Compare SP to memory | (SP) - (M:M+1) |
| CPX | Compare X to memory | (X) - (M:M+1) |
| CPY | Compare Y to memory | (Y) - (M:M+1) |
| | **Test instructions** | |
| Mnemonic | Function | Operation |
| TST | Test memory for zero or minus | (M) - \$00 |
| TSTA | Test A for zero or minus | (A) - \$00 |
| TSTB | Test B for zero or minus | (B) - \$00 |

## Loop Primitive Instructions

- 68HC12 provides a group of instructions that either decrement or increment a loop count to determine if the looping should be continued.

-These are SHORT branches. *Counter* = A, B, D, X, Y, S

**DBNE** counter, rel (or **DBEQ**)

- Decrement counter and branch if != 0

**IBNE** counter, rel (or **IBEQ**)

- Increment counter and branch if != 0

**TBEQ** counter, rel (or **TBNE**)

- Test counter and branch if = 0

- where counter = A,B,D,X,Y or SP

**Example** Write a program to compute the sum of the first n whole numbers (n<255), leaving the result in a word variable called "series".

**Solution**:

```
int series = 0;
for (byte i = N; i != 0;  i-- ) {
    series += i;
}
```

```
N     equ 20
      org $800
series   rmb   2

      org $4000
      ldab #N            ; b = i = 0
      ldx  #0            ; x = series = 0

next:
      abx                ; x (series) += i
      dbne b, next       ; i--; while b != 0
      stx  series        ; update series var.
      bra  *             ; end of program
```

**Example** Write a code fragment showing how to implement the following pseudocode

```
boolean  done = FALSE;
while ( ! done ) {
        . . .
        ; at some point
        done = TRUE;
}
```

**Solution:**

```
TRUE  equu 1
FALSE equ 0

     org $4000
     ldab #FALSE    ; b = done
notDone: TBNE b, end
     ; ...
     ; At some point,
     ldab #TRUE
     bra  notDone
end:     bra   *
```

# Special Conditional Branch Instructions

```
[<label>]   BRCLR  (opr) (msk) (rel)  [<comment>]
[<label>]   BRSET  (opr) (msk) (rel)  [<comment>]
```

where

**opr**: memory location to check. Specified using direct/indexed addressing mode.

**msk**: 8-bit mask. Specifies which bits to check (those whose bit positions are 1s in the mask).

**rel:** branch offset (relative mode).

Example :

```
loop        inc   count
            …
            brset  $66,$e0,loop
            …
```

Question: When will the branch be taken ?

**Example** Write a program to test whether a variable is divisible by 4, leaving the boolean result in accumulator A.

**Solution:** A number divisible by 4 would have the least significant two bits equal 0s.

```
FALSE equu   0
TRUE  equ    1
    org      $800
variable db  $19

    org      $4000
    brclr    variable,$03,yes ; check bits 1 and 0
    ldaa     #FALSE
    bra      continue
yes ldaa     #TRUE
continue
    . . .
    end
```

**Example** A robot has four motors, each of which can be off, on in forward direction, or on in reverse direction. Status of these motors are written by the robot into a status word, called "motors" in the following bitmask formation.

| 7 6 | 5 4 | 3 2 | 1 0 |
|---|---|---|---|
| Motor1 | Motor2 | Motor3 | Motor4 |

where the two bits for each motor are set according

| | |
|---|---|
| 01 | forward |
| 10 | reverse |
| 11 | off |

Write a code fragment that waits while motor1 is off before continuing on.

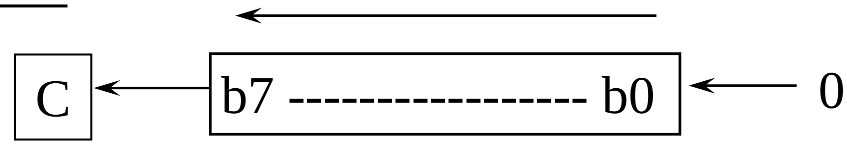
**Solution:**

```
org    $800
motors rmb     1
org    $4000
brset  motors,$C0,*   ; check bits 7 and 6
. . .
end
```

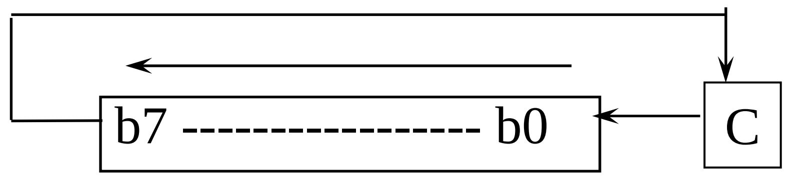
# Shift and Rotate Instructions

<u>Shift versus Rotate</u>

Shift



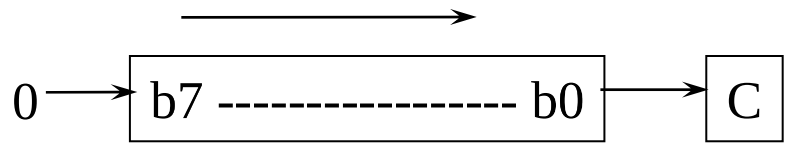

Rotate




<u>Arithmetic versus Logical Shift</u>

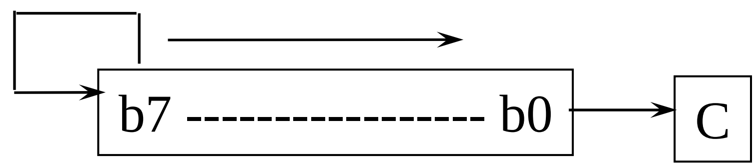
- Different only when shifting right.

Logical : Zero is injected




Arithmetic : MSB is injected.

Why ?



# Shift Instructions

- Apply to a memory location, accumulators A, B and D.
  - Memory operand specified using extended or indexed addressing modes.

8-bit arithmetic shift left instructions (**LOGICAL:** A -> L):

```
ASL   opr          ASR   opr
ASLA               ASRA
ASLB               ASRB
```

16-bit arithmetic shift left instruction:

```
ASLD               ASRD
```

“9-bit” rotate instructions.

```
ROL   opr          ROR   opr
ROLA               RORA
ROLB               RORB
```

**Example 2.23** Write a program to count number of 0s in the 16-bit number stored at $800-$801 and save the result in $805.

**Solution:**

* The 16-bit number is shifted to the right 16 times.
* If the bit shifted out is a 0 then increment the 0s count by 1.

```
        org  $800
db      $23,$55          ; test data
        org  $805
zero_cnt rmb 1
lp_cnt  rmb  1

        org  $4000
        clr  zero_cnt    ; initialize the 0s count to 0
        ldaa #16
        staa lp_cnt
        ldd  $800        ; place the number in D
loop    lsrd             ; shift the lsb of D to the C flag
        bcs  chkend      ; is the C flag a 0?
        inc  zero_cnt    ; increment 1s count if the lsb is a 1
chkend  dec  lp_cnt      ; check to see if D is already 0
        bne  loop
forever bra  forever     ; = bra *
        end
```

# Boolean Logic Instructions

- Changing a few bits are often done in I/O applications.
- Boolean logic operation can be used to change a few I/O port pins easily.

Table 2.8 Summary of Booleran logic instructions

| Mnemonic | Function | Operation |
|---|---|---|
| ANDA <opr> | AND A with memory | $A \leftarrow (A) \bullet (M)$ |
| ANDB <opr> | AND B with memory | $B \leftarrow (B) \bullet (M)$ |
| EORA <opr> | Exclusive OR A with memroy | $A \leftarrow (A) \oplus (M)$ |
| EORB <opr> | Exclusive OR B with memory | $B \leftarrow (B) \oplus (M)$ |
| ORAA <opr> | OR A with memory | $A \leftarrow (A) + (M)$ |
| ORAB <opr> | OR B with memory | $B \leftarrow (B) + (M)$ |
| COM <opr> | One's complement memory | M ← \$FF - (M) |
| COMA | One's complement A | A ← \$FF - (A) |
| COMB | One's complement B | B ← \$FF - (B) |
| NEG <opr> | Two's complement memory | M ← \$00 - (M) |
| NEGA | Two's complement A | A ← \$00 - (A) |
| NEGB | Two's complement B | B ← \$00 - (B) |

# Program Execution Time

- 68HC12 uses E clock as timing reference.
- Frequency of E clock is half of that of the crystal oscillator.
- Many applications require generation of time delays.

Creation of a time delay involves two steps:

1. Select a sequence of instructions that takes a certain amount of time to execute.
2. Repeat selected instruction sequence for an appropriate number of times.

Example: instruction sequence on following slide: 40 E cycles. Repeating this sequence certain number of times, other delays can be created.

Assume the 68HC12 runs under a crystal oscillator with a frequency of 16 MHz (E frequency: 8 MHz; clock period is 125 ns). The instruction sequence on next page will take 5 μs.

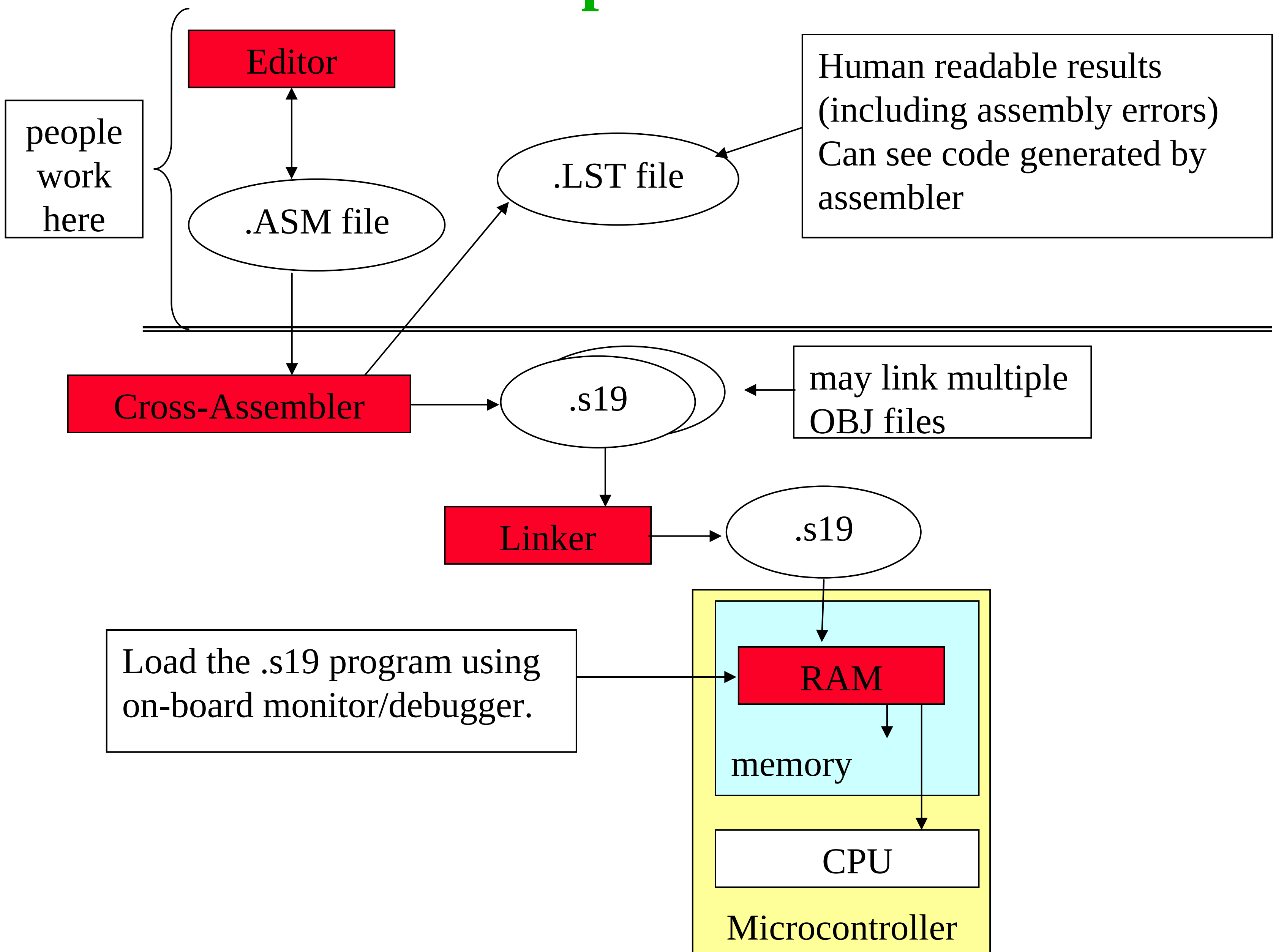
**Example 2.25** Write a program loop to create a delay of 100 ms.

**Solution:**

The following instruction sequence creates a delay of 100 ms.

```
        ldx     #20000
loop    psha                    ; 2 E cycles
        pula                    ; 3 E cycles
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        nop                     ; 1 E cycle
        nop                     ; 1 E cycle
        dbne    x,loop          ; 3 E cycles
```

# Development Process



# 2 Pass Assembly Process

Each pass: processes statements in .ASM file sequentially from start to finish

0th Pass: Macro Processor. Search/replace Macros and Constants (EQU)

1st Pass: for each statement:

1. check syntax
2. allocate any memory needed for image
   - memory declarations
   - instruction: opcode + operands
3. label definition: assign value to label; record (label, value) in Symbol Table
4. Generate instructions' opcodes/operands
5. Update .LST file

Any kind of errors: stop, else ....

2nd Pass:

- Associate addresses to not resolved symbols (forward jumps or variables; Symbol table)
- may require calculating offsets – may result in errors; e.g. trying to branch too far (target out of range) -
- write results to .LST file
- no errors? – write results to .s19 file

# Assembling: an example

**After 1st Pass:**

- **syntax OK**
- **bytes have been allocated to statements**
- **Symbol Table constructed:**

| Symbol | Value |
|---|---|
| start | 0000H |

**Label definition: in 1st pass – put value in Symbol table**

```
   $      binary_image
          =00001000               org    $1000
   1000 79 ????                   clr    zero_cnt   ; initialize the zero count to 0
   1003 86 10                     ldaa   #16
   1005 7A ????                   staa   lp_cnt     ; initialize loop count to 16
   1008 FC 0800                   ldd    $800       ; place the 16-bit number in D
   100B 49      again             lsrd
   100C 25 ??                     bcs    chk_end    ; have we tested all 16 bits yet?
   100E 72 ????                   inc    zero_cnt
   1011 73 ???? chk_end           dec    lp_cnt
   1014 26 F5                     bne    again
   1016 20 FE   forever           bra    forever

          =00000800               org    $800
   0800 23 55                     db     $23,$55
          =00000805               org    $805
   0805 +0001   zero_cnt          rmb    1
   0806 +0001   lp_cnt            rmb    1
                                  end
```

**Note: Do not confuse $ with PC!**

**$ = artifact of assembler**

**PC = artifact of processor**

# Assembling: an example

**After 2nd Pass:**

- **Missing symbols updated**

```
        $       binary_image
            =00001000                   org     $1000
        1000 79 0805                    clr     zero_cnt    ; initialize the zero count to 0
        1003 86 10                      ldaa    #16
        1005 7A 0806                    staa    lp_cnt      ; initialize loop count to 16
        1008 FC 0800                    ldd     $800        ; place the 16-bit number in D
        100B 49      again              lsrd
        100C 25 03                      bcs     chk_end     ; have we tested all 16 bits yet?
        100E 72 0805                    inc     zero_cnt
        1011 73 0806 chk_end            dec     lp_cnt
        1014 26 F5                      bne     again
        1016 20 FE   forever            bra     forever

             =00000800                  org     $800
        0800 23 55                      db      $23,$55
             =00000805                  org     $805
        0805 +0001   zero_cnt           rmb     1
        0806 +0001   lp_cnt             rmb     1
                                        end
```