68HC11 PROGRAMMING IN ASSEMBLY LANGUAGE

- Assembler Process
- Assembler Directives
- Assembler Operation

Assembly Language

❖ Assembler language ease the correct writing program, readable and efficient

❖ A statement of assembly language equals to a machine operation

❖ Assembler language created by editor with the extension of .ASM

❖ Assembler is a program that translate the assembly language to object code.

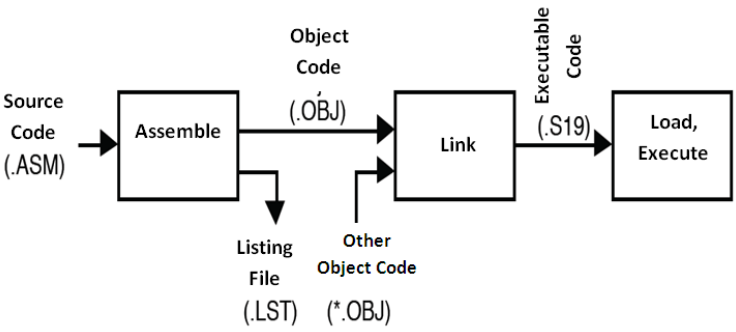
❖ Object code is the added info of machine language . It enable the file object linked and becoming a larger program.

❖ The usual object code used the extension of .OBJ

Assembler

- Assembler also generate listing file that suitable to read compare to the source code or object code
- Listing file usually have the extension of .LST
- Linker is a program that combine some of the object module to generate an execute file with extension of .S19
- The standard format of this file is Motorola S-record
- Still not a machine code but it enable the loader program to load the program and data into memory.

Assembler Process



Two routes assembler

*First route

> Record any information of known address

> Record all labels and symbols

> Assemble opcode and operand

> Determine the space needed for each instructions

*Second route

> Load the address information for the label and symbol that not completed yet

> Provide the object code

Assembler Input

```
* THIS LINE IS A COMMENT. YOU COULD PUT THE TITLE HERE.
*
        LDAA    $26     ; A <- MEMORY LOCATION HEX 26
        DECA
        STAA    $C401   ; A -> MEMORY LOCATION HEX C401
        LDAB    #23     ; B <- DATA BYTE DECIMAL 23
        LDX     $21     ; X <- MEMORY LOCATION HEX 21
        STAB    17,X    ; B -> (X) + DECIMAL 17
        BRA     -15     ; LOOP BACK (DECIMAL) 15 BYTES
        SWI             ; PSEUDO-"STOP" FOR EVB
```

Assembler Output

```
  Assembling prog.asm
0001                              * This line is a comment. You could put the title here.
0002                              *
0003 0000 96 26             [  3 ]          ldaa $26 ; A <- memory location hex 26
0004 0002 4a                [  2 ]          deca
0005 0003 b7 c4 01          [  4 ]          staa $c401 ; A -> memory location hex c401
0006 0006 c6 17             [  2 ]          ldab #23 ; B <- data byte decimal 23
0007 0008 de 21             [  4 ]          ldx $21 ; X <- memory location hex 21
0008 000a e7 11             [  4 ]          stab 17,x ; B -> memory location (X) plus decimal 17
0009 000c 20 e3             [  3 ]          bra -15 ; loop back (decimal) 15 bytes
0010 000e 3f                [ 14 ]          swi ; pseudo-"stop" for EVB
  Cycles Counted:  36


Number of errors 0
Number of warnings 0
```

If no ORG then the starting address will be assumed as $0000.

Label Field

- Label refer to the memory address or a constant
- Label must start from column 1 with a letter
- Other than label, only comment can start at column 1
- Must not exceed 16 character
- Uppercase and lowercase is assume as different
- Special string (like opcode) is prohibited
- Cannot use the register name
- Label must be unique (use once)
- Isolated from following field with a space

Instruction and Operand Field

*Instruction field:

Cannot be in column 1

must be an opcode (processor instruction) or pseudo-op (Assembler instruction)

*Operand field:

Not necessary to have:

- immediate – common mistake

% - binary value

$ - hexadecimal value

Comment Field

✓ Comment start after opcode/operand with : ; or * at column 1

✓ Assembler ignore all the character after * or ; till the end of line

✓ Critical for documentation

Object Code

- ASCII file text consist of machine code for µC use.
- Normally is the format of Motorola S-record. Other like Intel hex format is available too.
- S-record contain 2 types of record: data record and ending record.
- Data record have following:
  - > "S1" string
  - > two character count data (1 hex byte) – total byte from address to checksum = # byte data + 3
  - > Address input data (2 byte)
  - > data byte
  - > checksum (1 byte) – sum from address to checksum % 256 must be $FF

Example of data record

S112000096264AB7C401C617DE21E71120E33F55

• Have 18 byte from 00 to the check byte, so the data length (nearest byte after "S1) is $12.

• Add all the digit from data length until last byte:

$12 + $00 +$00 + $96 + …+ $3F => $6AA

• Check byte is the reversed total of lowest byte:

~$AA => ~%10101010 => %01010101 => $55

• Check byte is valid with adding each byte after "S1". The last byte total suppose to be $FF:

$12 + $00 + $00 + $96 + … + $3F + $55 => $6FF

End record

End record contain below info:

"S9"

byte data counter = "03

4 character of start address

2 character check byte

CR

LF

Example for program start at $F000:

S903F0000C

General format of listing file

>Line of listing file have below format:

AAAA [CC] VVVVVV LLLL Source code

* **AAAA** – start address for load generated machine code
* **[CC]** – If the counter cycle been choose, CC represents the total machine cycle to execute this line machine code
* **VVVVVV** – machine code generated for assembly code
* **LLLL**– line number source code
* Source code – copy source code including comment

> Listing file may end with symbol table – list of label that used in program followed by symbol address.

Assembler Directive

- Assembler directive known as pseudo-opcode
- Have the same format as the assembly instruction (opcode) but NOT producing the machine code
- Control code the generated including:
  - > start address of code or data
  - > provisions of memory space
  - > definitions of constant or label

RMB – Reserve Memory Byte

*Syntax:

[label] RMB <exp> [comment]

*Reserve or skip <exp> byte memory without load any value
*Label is given value equal to first byte address, if have it.
*Example: reserve 5 byte memory

FCB – Form Constant Byte

- Syntax:

[label] FCB<exp1>[,<exp2>,…] [comment]

- Set the content byte with value <exp1> and next
- Example: load $45, $56 and $FF in sequence

```
FCB   $45, $56, $FF
```


FCC – Form Constant Character

- Syntax :

[label] FCC <del><string><del>

- sequence will besieged by character, as long as they are similar
- additional FCB 0 instruction is needed when storing the sequence that end with NULL character.

- Example: 2 message "Hello!" and "How are you?"

```
MESSAGE FCC     /Hello!/
CSTRING FCC     $How are you?$
        FCB     0
```

FDB – Form Double Byte

✓ Syntax:

[label] FDB <exp1>[,<exp2>, …]
[comment]

✓ Set the word content (double byte) memory with value <exp1> and next
✓ **Example:** load $0045, $1056 and $00FF in sequence

FCB $45, $1056, $FF

EQU -- Equals

❖ Syntax:

<label> EQU <exp>

❖ Give the value <exp> to <label> enable the usage <label> as constant
❖ Example: Set the START equals to $B600

```
        START   EQU     $B600
    So, sequence
        START   EQU     $B600
                LDAA    START
        equals to
                LDAA    $B600
```

ORG -- Origin

- Syntax:

[label] ORG <exp> [comment]

- Gives the value to the assembly location counter '*'
- Set the code start address or data that followed
- Example: Set the data $45 into the address $100, $56 at $101.

```
ORG  $100
FCB  $45, $56
```


The good assembly program

- Normally, code divided to some parts,
  1. constant definition with EQU command
  2. variable definition with RMB, FCB and FDB
  3. true program

- Is good to announce all the constant as symbol (with EQU)
- FCB and FDB reserve the space and set the early value. Data is not return to the former program if it changed unless the program is being load again
- The true program contain a lot of comment!

A bad program!

- Terrible documentation – title is too short, some comments are long, EQU definition needs comment, data and program is all over the place, and part of the comments have wrong information
- Poor label – the usage of name and label have no meaning or useless, opcode and register name is used as label, wrong label
- Bad number – usage of decimal address
- Poor statement – Statement for 8 bit size and 16 bit is mixing
- Wrong instruction – the usage of number as address, confusing from the wrong label
- Excessive statement – the location and left statement from the previous version program

Equipment Development

Software	Equipment
Translator *Assemblers *Compilers	In-circuit Emulators *Substitutes for the CPU chip *Allows seeing "inside" µC
Linkers	Logic Analyzer *View timing and Bus Cycles
Debug Monitor *SDK Monitor ROM	Logic Probe
Performance Analyzers *Find Execution Bottlenecks	Oscilloscope
	Lights and Beepers

Software – Assembler-AS11, PCBUG11, BUFFALO
Hardware – Development Board, EVB, ICE
Combination – EVB board and software

How the Assembler Works

Assembler uses the following data structures:

* Input file: text assembly file

* Location counter: holds assigned address of instruction being assembled. Can be referred as "*" in the program.

* Symbol table: holds labels, their values and other info (such as local vs global type etc)

* Output file: binary program output

* List file: text file annotated with symbol table and human-readable binary program output

How the Assembler Works: Two Passes

❖ The assembler works in two passes so it can handle "forward references" like this

```
        BNE   LABEL
        ...
LABEL   LDAA#5
```

- Pass 1:
  Build symbol table
- Pass 2:
  Generate machine code

How the Assembler Works: Passes 1

✓ Identify all symbolic references to memory locations & identity starting address of all instructions

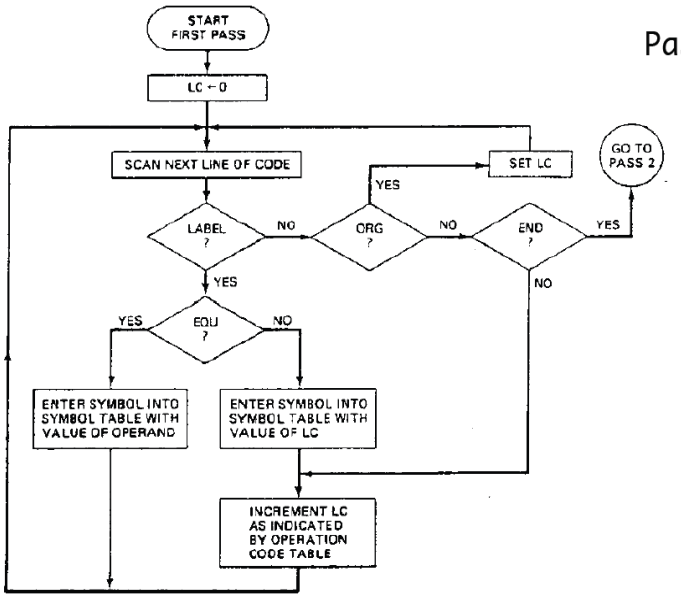
✓ Build symbol table
* Contains name and value for each symbol
* Symbols may be labels or created using EQU

✓ How?
* Read assembly file
* Increment location counter for each instruction or RMB/FCC
> E.g. if it finds "RMB 14" then location counter is incremented 14 bytes
* ORG directives are easy
> "ORG $300" means "location counter = $300"
*When it encounters a label, it records two things n the symbol table

Example Pass 1

```
        ORG     $100    Set LC=$100
BUFFER1 RMB     10      Records (BUFFER1,$100) in symbol table,
                          Increments LC by 20, now LC=$114
BUFFER2 FCB     2,5     Records (BUFFER2,$114) in symbol table,
                          LC = LC + 2
        ORG     $200    Set LC=$200
BACK    CLRA            This instruction takes 1 byte so LC=$201
                          Also records (BACK,$200) in sym.table
        LDAB    #5      This instruction takes 2 bytes so LC=$203
        BRA     *       Branch to same place
```

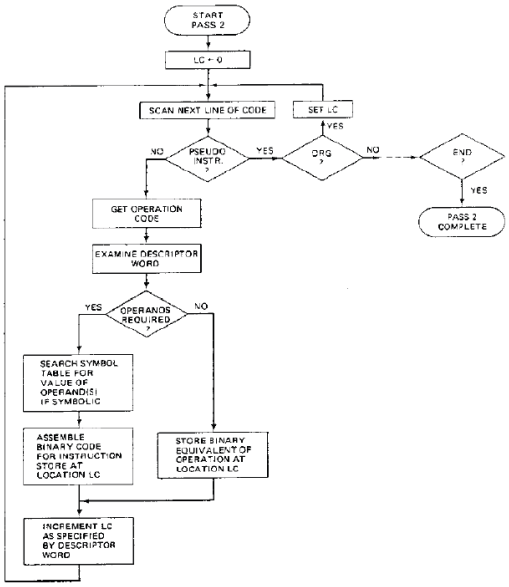
Pass 1



How the Assembler Works: Pass 2

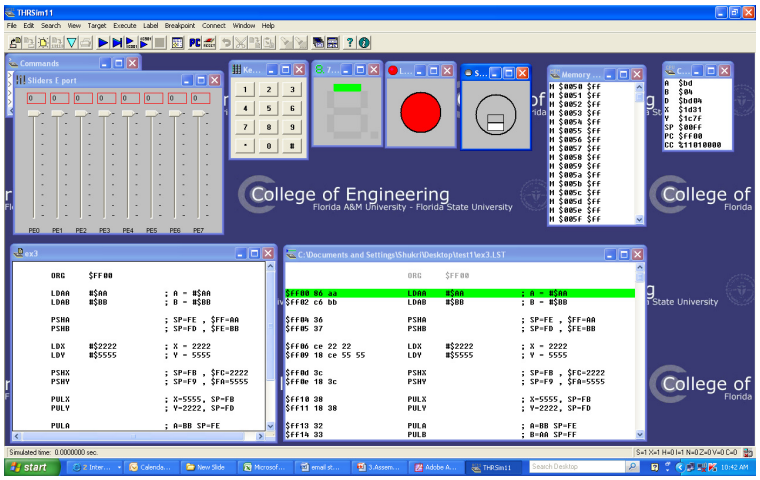
- Pass 2 is when the output file gets written
  - > Assembler restarts at top of input file
  - > Every instruction is converted to binary
  - > Values from FCB/FCC are converted to binary
  - > Each times it encounters a label, it inserts the address recorded in the symbol table
- Converts all symbolic references to absolute memory references and produces the final object code
- Uses the symbol table information

Pass 2



THRSIM11 Simulator And Compiler

THRSIM11 Simulator And Compiler



HC LOAD Hex Loader