EEL 4744C: Microprocessor Applications

Lecture 3

Part 1

Assembly Language Programming and Debugging
Reading Assignment

•Microcontrollers and Microcomputers: Chapter 5

Or

•Software and Hardware Engineering (new version): Chapter 2.4, Chapter 5.3, Chapter 6

Plus

•CPU 12 Reference Manual: Chapters 1, 2, and 3
Assembly Language

• Gives you a better appreciation of how the hardware works than high-level languages

• Resources are given by the programmer’s model

• These include registers, hardware resources in CPU, and memory used for data storage

• Look through the instructions in a category, find the mnemonic for the required operation, then find the correct addressing mode
The Assembler

• Converts (assembles) a source file into binary codes (machine codes) executed by the computer
  – 1-to-1 correspondence between assembly language statements and the machine code in memory

• Assembly source code fields: 4 of these
  – Label Field
  – Operation Code Field
  – Operand Field
  – Comment Field
Assembly Source Code Fields

• Label field
  – Optional and provides a symbolic memory reference
  – Represents the address of the first byte of an instruction or a data element
  – Also used to define constants
  – Usually starts with an alphabetic character, may contain digits and other characters

• Operation code field
  – Contains either a mnemonic for the operation or an assembler directive or pseudo-operation
  – Mnemonics are assembled into code to be placed into memory
  – Assembler directives or pseudo-operations are instructions to direct the assembler how to do its job
• Operation code field (continued)
  – In the following code example, the ORG (for origination) directive specifies where the code is located in memory
  – When assembler sees the ORG directive, it sets the value of current location counter to $1000

```
ORG $1000
op_code operand
op_code operand
JMP TARGET
op_code operand
op_code operand
... op_code operand
```

Use label to specify target address (relative address)

ORG directive helps determine actual address
Assembly Source Code Fields

• **Operand field**
  – Can be: *names of registers, numeric/symbolic constants, labels, algebraic expressions* to be evaluated by assembler
  – The following 2 instructions load the A, B registers from memory location DATA1 and DATA1+1
    
    \[
    \text{mov} \quad \text{a, DATA1} \\
    \text{mov} \quad \text{b, DATA1+1}
    \]

• **Comment field**
  – Ignored during assembly
  – May have source code lines that are only comments or blank
  – For some assemblers, beginning of comment fields are denoted by “;” or “*” characters
• Assembler **directives or pseudo-operations field**
  – We have seen the ORG example
  – There are others to define symbols, provide data in memory locations, reserve memory locations for data storage, define macros, etc.
Macro Assembler

• What is it?
  – An assembler in which frequently used assembly instructions can be collected into a single statement
  – It makes the assembler more like a high-level language

```cpp
DEF_MACRO
mov   a, c   ; (A) <= (C)
add   a, b   ; (A) <= (A)+(B)
mov   c, a   ; (C) <= (A)+(B)
END_MACRO
```

• 3 stages of using a macro: macro definition, macro invocation, macro expansion
Macros and Subroutines

- Each time a macro is invoked, the assembler expands the macro “in line”. A subroutine code is included only once. So the macro’s make your program larger.

- A subroutine requires a call or JMP, a macro does not. So a macro is faster than a subroutine to run.

- Both allows reuse of code segments. Both make the program easier to read, and allow changes to be made in one place. Both hide details of the program so you don’t need to know how it is doing something but just what it is doing.

Dr. Tao Li
Two-Pass Assemblers

• Allow symbols to be used before they are defined. This is called a forward reference

• The assembler evaluates these symbols by making 2 passes through the source code

• On the first pass, any symbol definitions it finds are recorded in a symbol table (containing values defined for symbols using the EQU directive, and the memory locations for labels, e.g. TARGET

• On the second pass, the assembler uses the symbol table to substitute the values for the symbols
Cross and Native Assemblers

• **Cross assembler**: for microcontroller applications, different computers are often used to edit and assemble the programs. (i.e. cross platforms)

• **Native assembler**: one that runs on the target processor

• **Assembler output**:  
  – Executable file or object file  
  – Assembler listing (source code + assembled code)  
  – Symbol table, cross-reference listing…
Code Location Problem

- Tied to the hardware’s specific memory map
- In a dedicated application system, code is located in ROM (read-only, non-volatile); data is in RAM
• **Absolute assemblers**
  – The code on slide #5, where the ORG directive specifies the code location, can be assembled with an absolute assembler

• **Major disadvantage**
  – The source file MUST contain ALL the source code to be in the program, i.e. all code must be assembled with any change
Code Location Problem

• Relocatable assemblers
  – Source code file need not have the complete program, nor does it need to have location information, or ORG statements
  – The program can be split into multiple source files and assembled separately, producing multiple object files.
  – These object files are then combined using a linker program
  – Code and data location information are supplied to the linker program to produce the final executable file

Dr. Tao Li
Code Location Problem

• The linker program
  – Takes modules that have been assembled by a relocatable assembler, links them together, and locates all addresses
  – May also receive object files from a library. Libraries are collections of object files that have been preassembled and bound together by a librarian program
  – Linker pulls from the library only those object files needed
Code Location Problem

- Source File 1: (MODULE1.ASM)
  - Relocatable Assembler
  - Object File 1: (MODULE1.OBJ)
- Source File 2: (MODULE2.ASM)
  - Relocatable Assembler
  - Object File 2: (MODULE2.OBJ)

- Librarian Program
- Library File: (FILE.LIB)
Code Location Problem

![Diagram showing the code location problem](image_url)
• **Loader program**
  - Puts an executable file into the memory of a computer
  - Can take on many forms, e.g. OS as MS-DOS programs, a modem/downloader program (cross platforms), a program or system that burns the programmable ROM

• **Assembly time**
  - Refers to data values that are known at the time of assembly
  - Includes constants (e.g. loop counters, ASCII character codes), operand expressions, absolute addresses and relative addresses (with relocatable assemblers)

• **Linking time**
  - Evaluate address/_constants left by the relocatable assembler
Code Location Problem

• Load time
  – When the executable file produced at link time (relocatable assembler) or at assembly time (absolute assembler) is loaded into the memory of the computer
  – Using assembler directives, it is possible to initialize the variables in RAM at load time

• Run time
  – Refers to the time when the program is running
  – Data elements are initialized and manipulated at run time
  – Addresses can also be initialized and manipulated at run time
Program Debugging

• **Debugging**: the process of finding the clues and interpreting these clues to find the problem

• **1st approach: synthesis**
  – Try to fix the problem by changing the code somewhere
  – Wrong!

• **2nd approach: analysis**
  – Try to find out what your program is doing before any fixes
  – Right!

• **Every program has 2 parts: data, logic**
  – Most program bugs occur in the logic
Program Debugging

What we think will happen.

Input Data Set

Stepwise Program Model

Stepwise Expected Outputs

Expected Next Program Step

What actually happens.

Input Data Set

Stepwise Actual Program

Stepwise Actual Outputs

Debugging the Data

Debugging the Program Flow

Actual Next Program Step

Dr. Tao Li
Program Debugging

• Choose an input data set

• Predict what the program will do with the input data set at each step of the program, and what the program will do next (the program model)

• Now run the program, and look for data values and program steps that differ from the prediction

• Once we find out where the program deviates from the prediction, we are on track to finding the bugs and fixes for the bugs
• **Program trace**: stepping through the program one statement at a time. Values of registers are shown during each step. Data elements in memory have to be checked manually.

• **Breakpoints**: much like a high-level language debugger, e.g. Gdb. These cause program flow to be interrupted, and control transferred to the debugger (the user).

• Breakpoints can be set at statements, as well as when certain conditions are met, e.g. data elements become some specific values.
Debugging Data Elements

• **Registers**: state of the registers must be known at each step. Debugger usually will display contents of all the registers, including the CCR. Watch for any deviating register content values

• **Memory**: usually are displayed in hexadecimal

• You will most likely need a source code listing generated by the assembler, not the original source code file. And you will need an up-to-date one!

• Debugging plan: no universal solutions, but in general make your code “modularly”
EEL 4744C: Microprocessor Applications

Lecture 3

Part 2

68HC12 Assembly Program
Reading Assignment

• Software and Hardware Engineering (old version): Chapter 3

Or

• Software and Hardware Engineering (new version): Chapter 5.1, 5.4, 5.5, 5.6,

Plus

• M68HC12B Family Data Sheet: Chapter 1
Hello World! (Assembler Listing)

1 ; Example program to print "Hello World"
2 ; Source File:  hello.asm
3 ; Author: F. M. Cady
4 ; Created: 4/97
5 ; Constant equates
6 EOS:  EQU    0       ; End of string
7 ; Debug-12 Monitor equates
8 printf: EQU    $FE06   ; Print a string
9 ; Memory map equates
10 PROG: EQU    $0800    ; RAM in the EVB
11 DATA: EQU    $0900    ; Middle of RAM
12 STACK: EQU    $0a00    ; Stack pointer
13 ORG     PROG      ; Locate the program
14 ; Initialize stack pointer
15     lds    #STACK
16 ; Print Hello World! string
17     ldd    #HELLO ; Pass the adr of the string
18     ldx    printf ; The adr of the printf routine
19     jsr    0,x
20 ; Return to the monitor
21     swi
22 ; Define the string to print
23 HELLO: Dc.B    'Hello World!',EOS

Dr. Tao Li
• **Label:** see textbook section 5.3 for how to define labels
  – A whitespace character is needed for a source code line without label

• ** Opcode field:** mnemonic, assembler directive or pseudooperation, macro name

• **Operand field:** symbols, constants, expressions (see textbook section 5.3 for detail)
  – Special symbols: * and $ represent the current value of program counter
  – Constants: Decimal (default), Hexadecimal /w ($) prefix, Binary /w (%) prefix, ASCII /w ‘ ’ and “ ”
  – Expressions: evaluated by assembler, used for specifying constant only

• **Comment field:** ; or * (in column 1)
68HC12 Assembler Program

- **Assembler pseudo-ops:**
  - **ORG:** set assembler’s location counter (e.g. `ORG $1000`)
  - **EQU:** equate symbol to value (e.g. `COUNT EQU $10`)
  - **DS (define storage):** reserve storage (e.g. `ARRAY DS 10`)
  - **DC.B (define byte constant), DC.W (define word constant):** define constants in memory storage (e.g. `DATA1 DC.B $10,$20,$30`)

Pseudo-ops help the assembler generate code for the program

Dr. Tao Li
Example: ORG

0000  1  ROM:   EQU   $F000  ; Location of ROM
0000  2  RAM:   EQU   $0800  ; Location of RAM
0000  3  STACK: EQU   $0a00  ; Location of stack
4  ;
F000  5  ORG   ROM  ; Set program counter to ROM
6                            ; for the program
7  ; The following code is located at memory address ROM
F000  CF0A00  8   lds   #STACK  ; Initialize SP
F003 B60800  9   ldaxa DATA_1 ; Load from memory address RAM
10  ;   ---
0800 11  ORG   RAM  ; Set program counter to RAM
12                            ; for the data
0800 20 13 DATA_1: DC.B   $20  ; Set aside $20 bytes
Example: EQU

```
0000 1 CRLF:  EQU  $0D0A ; For each occurrence of
       ; CRLF, the assembler will
0000 2       ; substitute the value $0D0A
0000 3 COUNT: EQU  5  ; Loop counters often need to
       ; be initialized
0000 4 COUNT1: EQU COUNT*5 ; The assembler can evaluate
       ; an expression to provide
       ; a value of 25 for COUNT1
0000 5 LSMASK: EQU $0F  ; A mask that picks off the
       ; least significant nibble
       ; in a byte
0000 6 ls_mask: EQU %0001111; A binary mask equate is
       ; more readable and
       ; informative than one
       ; given in hexadecimal
```

Any constant value can be defined for the assembler using the EQU

Dr. Tao Li
Example: DS Directive

Show how to use the DS directive to reserve 10 bytes for data. Initialize each byte to zero in a small program segment.

```
  0000 1  NUMBER:  EQU   10 ; Number of bytes allocated
  0000 2  PROG:  EQU   $0800 ; Program location
  0000 3  RAM:  EQU   $0900 ; Location of RAM
  0800 4  ORG   PROG
       5 ; ---
  0800  C60A  6  ldab   #NUMBER ; Initialize B with a loop
               ; counter
  0802  CE0900  8  ldx     #BUF ; X points to the start of the
               ; buffer
  0805  6930  10  loop: clr     1,x+ ; Clear each location and
               ; point to the next location
  0807  0431FB 12  dbne   b,loop ; Decrement the loop counter
               ; and branch if the loop
               ; counter is not zero
       15 ; ---
  0900 16  ORG   RAM ; Locate the data area
  0900 17  BUF:  DS   NUMBER ; Allocate the data area
```
Assembler directives: (see textbook Table 5-5 for complete list)

- Exact syntax varies by assembler program used; most interesting options for **conditional assembly** and **macros**

- Directives are invoked by placing a /, # or $ as the first character of a line followed by the directive and any parameters

- Conditional assembly directives permit different sections of source code to be assembled depending on setting of condition;

Assembler directives control
How the assembler creates its output files

Dr. Tao Li
Example: Conditional Assembly

1 ; Set the version number for this software
2  PARAM1:  EQU   $76 ; Parameter to use for vers 1
3  PARAM2:  EQU   $77 ; Parameter to use for vers 2
4 $SET      Ver1
5 ; . . .
6 ; The conditional assembly follows.
7 IF Ver1
8 ; This code is assembled if Ver1 has been SET.
9  ldaa #PARAM1
10 ELSEIF
11 ; This code will be assembled if Ver1 has been SETNOT.
12  ldaa #PARAM2
13 $ENDIF
68HC12 Assembler Program

• Assembler directives: Macros
  – Assembler directives MACRO and MACROEND encapsulate the code
  – Macro parameters: specified by %n (n is the n_th parameters) in macro definition
  – Labels in Marcos: A label may not occur more than once in a program. The assembler expands macro labels to make each label unique
Example: MACRO Parameters

1 ; Macro definition for a variable arithmetic
2 ; shift left of the A register
3 $MACRO alsa_n num
4 ; Shift the A register left num bits
5 ; where n is a parameter in the macro call.
6 ; Save B to set up a loop counter
7 pshb ; Save B on the stack
8 ldab #1
9 loop: asla ; Shift the A register
10 dbne b,loop ; Decr and branch if not zero
11 pulb ; restore the B register

$MACROEND

; The macro call is with a parameter
macro alsa_n 3
PSHB
LDAB #1
DBNE B,LOOP
PULB
Assembler Output Files

- Assembler listing (includes a symbol table) and executable (S-Record file)
  - Assembler listing
    Loc [Cycles] Obj Code Line Source Line
  - Symbol table: lists all symbols used in a program and their values
  - Cross-reference table: shows where the symbols are defined and referenced
Example: Symbol Table

... ...

5 ; Constant equates
6 EOS: EQU 0 ; End of string
7 ; Debug-12 Monitor equates
8 printf: EQU $FE06 ; Print a string
9 ; Memory map equates
10 PROG: EQU $0800 ; RAM in the EVB
11 DATA: EQU $0900 ; Middle of RAM
12 STACK: EQU $0a00 ; Stack pointer
13 ORG PROG ; Locate the program
14 ; Initialize stack pointer
15 lds #STACK
16 ; Print Hello World! string
17 ldd #HELLO ; Pass the adr of the string
18 ldx printf ; The adr of the printf routine
19 jsr 0,x
20 ; Return to the monitor
21 swi
22 ; Define the string to print
23 HELLO: DB 'Hello World!',EOS

Symbol Table

DATA 0900
EOS 0000
HELLO 080C
PRINTF FE06
PROG 0800
STACK 0A00

Dr. Tao Li