Assembly Language Programming

by Alex Milenkovich, milenkovic@computer.org

1. Assembly Language Programming: An Introduction

An assembler is a program that converts an assembly language application program to a binary machine language program (e.g., opcode and operands bytes in the MPS430's memory). Similarly, a compiler is a program that converts an application program written in C or C++ into an intermediate file called an object file.

Modern software engineering encompasses many steps, such as requirement analysis, user interface design, software design, software implementation (programming), software testing, tuning, and optimization. Here we will focus on assembly language software development illustrated in Figure 1.

The first step in the process is to develop assembly program text. Assemblers typically provide assembly text editors to help program development. At *assemble time*, the input text is passed to the assembler, which parses the text, emitting a program listing and possibly error reports. If there are no errors, the assembler produces a binary machine language module. The module must contain information about where the program or module is to be loaded in memory, and if it contains the starting address of the program, this start symbol must be made known. If the module has to be linked with other modules, then it must include this additional linkage information. This means that all labels in the module that have to be visible to other modules must be specified as public symbols. Similarly, all labels that are defined in other modules must be specified as *external symbols*.

At *link time*, the linker combines separately assembled modules into a single load module. The linker will also add any initialization or finalization code to allow the operating system to start the program or return control to the OS, once the program has completed.

At *load time*, the program loader copies the program into computer's main memory.

At *run-time*, the program execution begins.

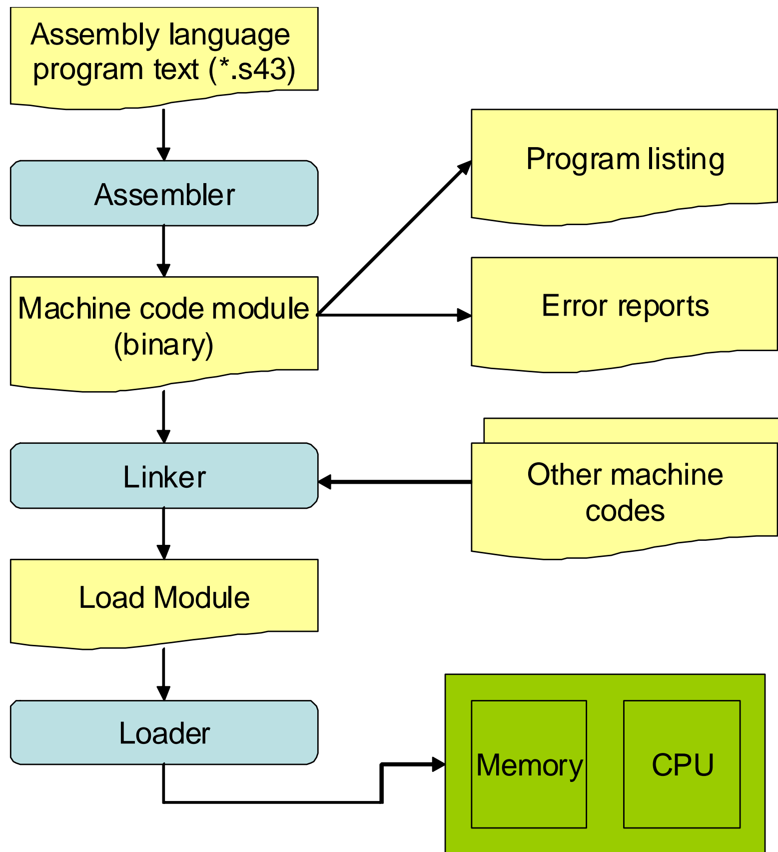

Figure 1. Program development flow using assembly language programming.

2. What do assemblers do?

Assemblers typically provide the following capabilities.

1. Allow programmers to access and use all ISA components (instructions, registers, memory)
2. A means for specifying run-time location of program and data in memory
3. Provide symbolic labels for the representation of constants and addresses
4. Perform compile-time arithmetic
5. Provide the use of any synthetic instructions
6. Expand programmer-defined macro routines
7. Emit machine code in a form that can be loaded and executed
8. Report syntax errors and generate program listings
9. Provide an interface to the module linkers and program loader.

3. Assembly Programs

Assembly Language Syntax

Each assembler has its own unique syntactical structure (use of uppercase or lowercase, label definitions, token separators, etc). In spite of this, all assemblers share some common features.

The assembly text is usually divided into fields, separated by space or tabs. A typical assembly code line is shown below:

```
[Label]   MOV  Operand1, Operand2       ; Comment
```

The first field, which is optional, is the label field, used to specify symbolic labels and constants. Some assemblers require labels to end with a colon character. The next field is the opcode field, and the third and the following fields are operand fields (usually comma-separated). The comment field starts with a delimiter such as the semicolon and continues to the end of the line.

Assembly language directives: An MSP430 Example

Figure 2 illustrates typical directives used in the IAR software development suite to define constants and allocate space in memory.

```
        ORG 0xF000
b1:     DB   5       ; allocates a byte in memory and initialize it with constant 5;
                     ; equivalent to DC8 5
b2:     DB   -122    ; allocates a byte with constant -122
b3:     DB   10110111b ; binary value of a constant
b4:     DB   0xA0    ; hexadecimal value of a constant
b5:     DB   123q    ; octal value of a constant
        EVEN         ; move a location pointer to the first even address

tf      EQU 25

w1:     DW   32330    ; allocates a a word size constant in memory; equivalent to DC16 32330
w2:     DW  -32000
dw1:    DL  100000    ; allocates a long word size constant in memory; equivalent to DC32 100000
dw2:    DL  -10000
dw3:    DL 0xFFFFFFFF
dw4:    DL tf
s1:     DB 'ABCD'     ; allocates 4 bytes in memory with string ABCD
s2:     DB "ABCD"     ; allocates 5 bytes in memory with string ABCD and \0 character at the end


        ORG 0x0200
v1b     DS  1        ; allocates a byte in memory; equivalent to DS8
v2b     DS  1        ; allocates a byte in memory;
v3w     DS  2        ; allocates a word of 2 bytes in memory; equivalent to DS8 2 or DS16
v4b     DS32 4       ; allocates a buffer of 4 long words; 4x4=16 bytes in memory
```

Figure 2. Illustration of MSP430 assembly language directives.

Assembly programs: An MSP430 Example

Figure 3 shows an example MSP430 assembly program used in the Lab2. Please read the Lab2 materials. The program counts the number of characters ‘E’ in a string.

```
/*-----------------------------------------------------------------
* Program    : Counts the number of characters E in a string
* Input      : The input string is the myStr
* Output     : The port one displays the number of E's in the string
* Written by : A. Milenkovic
* Date       : August 14, 2008
* Description: MSP430 IAR EW; Demonstration of the MSP430 assembler
*-------------------------------------------------------------------*/

#include "msp430.h"                     ; #define controlled include file

        ORG 0FF00h
myStr:   DB "HELLO WORLD, I AM THE MSP430!" ; the string is placed on the stack
; the null character is automatically added after the '!'

        NAME    main                    ; module name

        PUBLIC  main                    ; make the main label visible
                                        ; outside this module
        ORG     0FFFEh
        DC16    init                    ; set reset vector to 'init' label

        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        BIS.B   #0FFh,&P1DIR            ; configure P1.x output
        MOV.W   #myStr, R4                ; load the starting address of the string into the
register R4
        CLR.B   R5                        ; register R5 will serve as a counter
gnext:  MOV.B   @R4+, R6                  ; get a new character
        CMP     #0,R6
        JEQ     lend                        ; go to the end
        CMP.B   #'E',R6
        JNE     gnext
        INC     R5                          ; increment counter
        JMP     gnext

lend:   MOV.B   R5,&P1OUT                ; Set all P1 pins
        BIS.W   #LPM4,SR                 ; LPM4
        NOP                              ; Required only for debugger

        END
```

Figure 3. MSP430 Assembly Code for Count Character Program.

Here is a short description of the assembly code.

1. The comments in a single line start with a column character (;). Multi-line comments can use C-style /* comment */ notation.
2. #include <msp430.h>; This is a C-style pre-processor directive that specifies a header file to be included in the source. The header file includes all macro definitions, for example, special function register addresses (WDTCTL), and control bits (WDTPW+WDTHOLD).
3. Use ORG assembly directive to set the program location counter of the current segment to the value of an expression that follows. Here `ORG 0FF00h` sets the location counter at the absolute address 0FF00h. This means that location counter is moved to this address.
4. Next, we allocate the string myStr that will start at the location 0FF00h using DB directive: `myStr DB "HELLO WORLD, I AM THE MSP430!"`. As explained, this directive will allocate 30 bytes in memory starting at the address 0FF00h and initialize it with the string content.

The content in memory will be as follows: 48 45 4c 4c 4f 20 57 4f 52 4c 44 2c 20 49 20 41 4d 20 54 48 45 20 4d 53 50 34 33 30 21 00.

5. How does my program execute on an MSP430? Upon powering-up the MSP430 control logic always generates a RESET interrupt request (it is the highest priority interrupt request). The value stored at the address 0xFFFE (the last word in the 64KB address space) is reserved to keep the starting address of the reset handler (interrupt service routine), and the first thing that the microcontroller does is to fetch the content from this address and put it in the program counter (PC, register R0). Thus, the starting address of our program should be stored at location 0xFFFE. Here, we move location counter to 0xFFFE and allocate 2 bytes (DC16 allocates 16 bits or two bytes) that will be initialized with the starting address of the main program. The starting address of the main program is marked by the label `init`.
6. RSEG is a segment control assembler directive that controls how code and data are located in memory. RSEG is used to mark the beginning of a relocatable code or data segment. CODE and DATA are recognized segment types that are resolved by the linker. The IAR XLINK linker can recognize any other type of segment (e.g., CSTACK for program stack).
7. First instruction initializes the stack pointer register (`MOV #SFE(CSTACK), SP`). Our program does not use the stack, so we could have omitted RSEG CSTACK and this instruction.
8. The instruction `MOV.W #WDTPW+WDTHOLD,&WDTCTL` sets certain control bits of the watchdog timer control register (WDTCTL) to disable it. The watchdog timer by default is active upon reset, generating interrupt requests periodically. As this functionality is not needed in our program, we simply need to disable it.
9. Parallel ports in MSP430 microcontroller can be configured as input or output. A control register PxDIR determines whether the port x is input or output (we can configure each individual port pin). Our program drives all pins of the port P1, so it should be configured as the output (P1DIR=0xFF). Register R4 is loaded to point to the first character in the string. Register R5, the counter, is cleared before starting the main program loop.
10. The main loop starts at the `gnext` label. We use autoincrement addressing mode to read a new character (one byte) from the string (`MOV.B @R4+, R6`). The current character is kept in register R6. We compare the current character with the NULL character (`CMP #0, R6`). If it is the NULL character it is the end of the string and we exit the loop (`JEQ lend`). Pay attention that we used `JEQ` instruction? Why?
    If it is not the end of the string, we compare the current character with ‘E’. If there is no match we go back on the first instruction in the loop. Otherwise, we increase the value of the counter (register R5).

Finally, we move the lower byte from R5 to the parallel port 1, P1OUT=R5[7:0].

Note: Absolute vs. Relocatable Assemblers

Absolute assemblers. When writing assembly programs we know from our hardware design where the code is to be located in memory. A special directive ORG provides this information to the assembler. All code is located absolutely at a specific memory address. The simplest form of assembler is so called absolute assembler. It takes the source assembly file and produces an executable file that is downloaded to the target system.

Disadvantages of absolute assemblers are as follows: all code must be in one module which is inconvenient for larger group projects; program becomes less portable across different platforms.

Relocatable assemblers. A relocatable assembler accepts an assembler program as a source file that does not need to contain location directives. The assembler produces an output file, called an object file, which contains the binary codes for the operations and un-resolved symbols. For example, when a branch address cannot be evaluated the assembler adds this symbol to the object file, so a linker program can provide the final address. Program can also be split into multiple source files and assembled at different times.

4. How do assemblers work?

During the process of assembling a program, the assembler must pass over the text, identify and resolve all symbols to their numeric values, and covert the assembly language text to binary machine code.

As the assembler passes over the text, it will encounter programmer-defined symbols, e.g., labels of defined constants (allocated in memory or synonyms), program labels, or other entries. The assembler resolves these symbols by means of a symbol table. The symbol table is maintained by the assembler, and each entry has the following fields: symbol name, symbol type (e.g, integer, float, label, external, public, unknown), symbol value, and the symbol status (e.g., defined, undefined).

Assemblers typically make two passes (two-pass assemblers) through assembly language text. In the first pass the assembler reads the assembly language text, identify instructions, operands, and user-defined symbols. Assembly language mnemonics are kept in an instruction table with corresponding binary equivalents. When the assembler encounters an assembly instruction it searches the instruction table to get the binary opcode. Similarly, operand fields are parsed. User-defined symbols are also resolved immediately if possible (e.g., EQU directive associates a symbol with its value). If all information is available the assembler literary assembles the binary equivalent of the instruction under consideration. To keep track of instruction and data addresses, the assembler maintains a so-called *location counter*. This counter is assembly-time equivalent of the program counter. The assembler initializes the location counter to 0 before it begins the assembly process. When it encounters instruction or data primitives the location counters is incremented appropriately. The ORG directive forces the assembler to move the location counter to the value specified by the ORG directive.

Some symbols cannot be resolved immediately (e.g., labels for forward branches). All unresolved symbols are flagged as such in the symbol table. Later in the first pass these symbols will be resolved. In the second pass, the assembler replaces symbolic references with their binary values.

The assembler completes the process by inserting linkage information such as the values of public and external symbols, and the value of the program start symbol.