CpE 213
Digital Systems Design

Lecture 14

Tuesday 10/14/2003

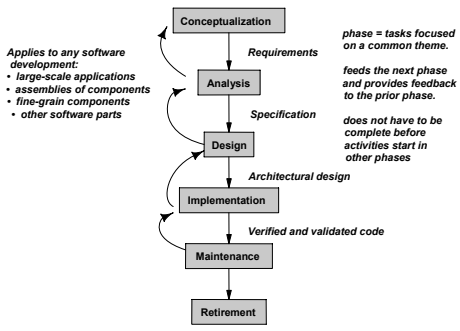
Announcements

- Exam 1 has been graded.
- Exams can be picked up during office hours.
- Send me one email per group with names of all group members. Indicate which members are taking CpE 214.
- Assignment #5 has been posted. Work on it as soon as possible after this lecture.

Assembly Language Programming for the 8051
ISM Chapter 7
Sections 7.1 – 7.5

Some slides adapted from Microprocessor Systems Notes by K.T. Ng

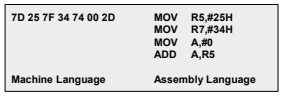
Generic Software Development Cycle



8051 Assembly Programming

- Machine instructions show basic capabilities of 8051.
- Assembly language (ASM) is built around these capabilities.
- Our version of ASM is slightly different than that of the textbook.

Machine Language vs. Assembly



- Machine language is a binary sequence interpreted by the computer as a sequence of instructions.
- Assembly language is a form of machine language that uses mnemonics (memory aid) for the instruction type, and operands and is easier for humans to read.
- Generally, one-to-one correspondence between assembly instructions and machine instructions.

Why Study Assembly Language ?

Some notable advantages:

- The object code is usually very efficient and may execute many times faster than object code generated by a high level language.
- Memory can be much more efficiently used in assembly language.
- It provides direct contact with the hardware.
- It provides a good understanding of computer architecture (e.g. data representation, instruction execution, etc.)
- Ideal for embedded controllers where cost needs to minimized (CPU speed & memory size).

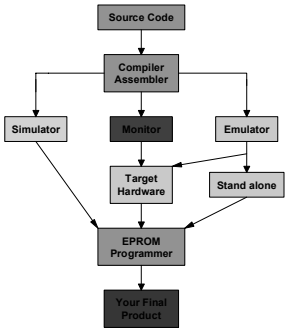
Disadvantages of Assembly Language

- The source code is often very lengthy as compared to high level languages.
- It is machine-dependent, and hence not very portable.
- It takes considerably more time to develop a program in assembly language than with a high-level language.
- It requires a lot of programming experience to be used effectively.

Use of Assembly Language

- The use of assembly language programming is normally restricted to relatively small programs or sections of programs in applications where:
  - Speed of execution is important, e.g., real-time systems.
  - Memory cost or size is a factor.
  - More I/O and control is involved than computation.

Embedded Development Cycle



Tools for Developing Assembly Language Programs

- The program development utilities enable the user to write, assemble, and test assembly language programs; they include:
  - Editor
  - Assembler
  - Linking Loader
  - Monitor and
  - Simulator

Types of Assemblers

- Translation from assembly language to machine instructions done by a program called the assembler.
- There are several types of assemblers, the most common of which are
  - one-pass assembler: The one-pass assembler was the first type to be developed and is therefore the most primitive. Because of the forward reference problem, it is seldom in use nowadays.
  - two-pass assembler: In a two-pass assembler the source code is processed twice. It is the most popular type of assembler currently in use.

Two-Pass Assemblers

- First pass
  - records any known address information
  - records all labels and symbols
  - Assembles opcode and operands
  - Determines how much space each section of code will take
- Second pass
  - Fills in missing addressing information for labels and symbols
  - Produces final object code

From ASM to Executable Code

- Create the assembly source file test.a51
- Assemble the ASM file
  - Assembler produces error and code list in test.lst
  - If no errors, assembler produces .obj file
- Link the .obj files to produce an .abs file
  - May also produce a listing file
- Create hex file from the .abs file
  - Most assemblers directly produce the .hex file
- Download the .hex file onto the board or burn it into an EPROM.

Why the Link Step ?

- The linker is a program that combines relocatable object programs (modules) and produces an absolute object program that is executable by a computer.
- A linker is sometimes called a “linker/locator” to reflect its separate functions of combining relocatable modules (linking) and setting the address for execution (locating).
- Relocatable means that addresses in the object code can be changed (relocated) so that the program can be loaded and executed anywhere in memory.

The Assembler Source File

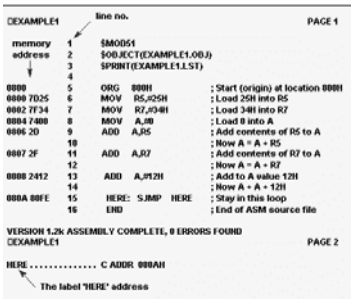
```
$MOD51
$OBJECT(EXAMPLE1.OBJ)<--   generate the Intel hex object file
$PRINT(EXAMPLE1.LST)  <--   generate an listing file

        ORG   800H      ; Start (origin) at location 800H
        MOV   R5,#25H   ; Load 25H into R5
        MOV   R7,#34H   ; Load 34H into R7
        MOV   A,#0      ; Load 0 into A
        ADD   A,R5      ; Add contents of R5 to A
                        ; Now A = R5
        ADD   A,R7      ; Add contents of R7 to A
                        ; Now A = R5 + R7
        ADD   A,#12H    ; Add to A value 12H
                        ; Now A = R5 + R7 + 12H
HERE:   SJMP  HERE      ; Stay in this loop
        END             ; End of ASM source file
```

The source code of EXAMPLE1.ASM

(written for Metalink 8051 cross assembler)

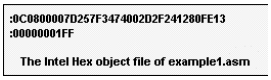
The Assembler Listing File



The listing file (EXAMPLE.LST) of EXAMPLE1.ASM

(produced by Metalink 8051 cross assembler)

The Assembler Object File



- One standard for storing machine language programs in a displayable or printable format is known as “Intel hexadecimal format”.
- This format was discussed in lecture 13.

The Monitor Program

- The monitor is a small program with commands that provide a primitive level of system operation and user interaction.
- Typical monitor commands allow
  - Memory examine/change
  - Register dump
  - Loading programs
  - Executing programs
  - Single stepping and setting breakpoints.

Simulator vs. Emulator

- A simulator is a program that runs on a host development system and is used to simulate the operation and features (memory, registers, status flags, I/O ports etc.) of a target microprocessor.
- In-circuit emulators are tools that allow us to develop code with debugging capabilities while running it on the actual target hardware. Unlike a simulator that requires us to generate any I/O signals in software, the emulator can use the actual target hardware.

Assembly Programs

- An assembly language file is a text file including:
  - Instructions
  - Pseudo-operations
  - Assembler directive
- Instructions are translated into machine language instructions to be stored in program memory and executed by the CPU.
- Pseudo-operations allocate space for variables or constants.
- Assembler directives are commands to the assembler program.

Structure of a Line of Assembly Language Code

- An assembly language instruction has 4 fields (optional ones are in brackets).

  [label:] mnemonic [operands] [;comment]
- The label gives us a way to refer to an memory address so that it will be easier to find and remember.
- The mnemonic is an 8051 instruction or an assembler directive.
- Operands are the arguments of the instruction or directive.
- Comments make the program much easier to read and provide documentation.

Example

- Target 8051 dev system
  - Std 8051 device
  - 2K on-chip ROM running a monitor program
  - 32K external RAM at address 0x0000 to 0x7FFF
  - This RAM is both code and data
  - First 0x30 locations in external RAM are dedicated to the InterruptVector Table (IVT)

- **Program to fill up the first 4 registers in the register bank with some numbers and find their sum**

```
        ORG 0x30 ;skip the IVT area
Start:  mov R0, #10
        mov R1, #0A5H
        mov R2, #1
        mov R3, #0x20
clearA: mov A, #0  ;now A = 0
Addup:  add A, R0  ;now A = A + R0
        add A, R1
        add A, R2
        add A, R3
        mov R4, A  ;store sum in R4
        mov DPTR, #7FFF
        movx @DPTR, A ;store in ext. mem
Done:   sjmp done  ;loop here forever
        END
```

Slide adapted from UT Dallas

The Label Field

- A label is used to associate a specific line of code, memory location, or constant definition with a text string. Rules are:
  - Labels must begin in the first column with an alphabetic character and end with a colon.
  - It must have a unique character pattern within the first 32 characters.
  - The first character MUST be either a letter or special character (? or _).
  - It must not be a reserved string (e.g. opcode and directives etc.).
  - Labels must not be redefined (reused).

Some Valid and Invalid Labels

- Serial_Port_Buffer: (valid) (invalid)
- less: (valid) (invalid)
- goback: (valid) (invalid)
- delay1: (valid) (invalid)
- 1ST_VARIABLE: (valid) (invalid)
- alpha#: (valid) (invalid)
- MOV (valid) (invalid)
- LOW (valid) (invalid)
- DATA (valid) (invalid)

The Mnemonic Field

- The mnemonic field contains the mnemonic names for machine instructions (e.g. MOV, SUB etc.) and assembler directives (e.g. EQU, ORG etc.).
- If a label is present, the opcode or directive must be separated from the label field by at least one space.
- If there is no label, the operation field must be at least one space from the left margin.

The Operand Field

- Not always present.
- If an operand field is present, it follows the mnemonic field and is separated from the mnemonic field by at least one space.
- The operand field may contain operands for instructions or arguments for assembler directives.

The Comment Field

- Comments begin after opcode/operand with a semi-colon (;), or after a semi-colon in the first column.
- Assembler ignores EVERYTHING that follows a semi-colon until a carriage return/line feed (CR/LF).
- Comments are critical to the source code documentation of an assembly file.

ASM Directive/Pseudo-opcode

- Directive (or known as pseudo-opcode) has the same format as assembler instruction (opcode), but it is **NOT** assembled into machine code.
- They function as commands to the assembler to define properties of the code. These properties include items such as:
  - Addresses to start code or data segments
  - Allocation of memory space
  - Constant or symbol definitions
- Beware ! Assembler directives vary from one software vendor to another.

Typical Assembler Directives

Mnemonic	Function
ORG	This command sets the starting address for the code or data to follow. It let you put code and data anywhere in program memory you wish.
EQU	This command equates the value to a label. The label can then be used as a constant throughout the rest of the code.
DS/DB/DW	DS(define storage) simply allocates memory whereas DB (define single byte) and DW (define word: 2 bytes) to allocate memory space and store an initial value
END	Tells the assembler to stop assembling. Anything after END will be ignored.

Some Examples of Directives

```
            ORG   800H          ;set PC to address 800H
            TEN   EQU   10      ;Symbol TEN equated to 10
IO_BUFFER:  DS    8             ;Reserve a buffer (8 bytes)
                                ;for the I/O
   STRING:  DB    'Hello'       ;ASCII literal
   RADIX:   DW    'H',1000H     ;1st byte contains 0
                                ;2nd byte contains 48H
                                ;3rd byte contains 10H
                                ;4th byte contains 0
                                ;"Always high byte first !"
ALSO_TEN    EQU   TEN           ;Symbol equated to a
                                ;previously defined symbol.
```

Segment Selection Directives

- Two Types of segments: absolute and relocatable
- Absolute: located at specific location
  - Ex: CSEG at 0000H
  - Types are:
    - CSEG code space
    - DSEG directly addressable internal data space
    - ISEG indirectly addressable internal data space
    - BSEG bit-addressable space
    - XSEG external data space

Relocatable Segments

- Assembler decides location of segment.
  - generally best choice
  - Ex: RSEG segment_name
  - segment_name needs to be previously defined with the SEGMENT directive
  - Ex: mydata SEGMENT data
    RSEG mydata
    - first directive only defines mydata to be a data segment
    - don't actually use the segment until RSEG directive is issued

Syntax of SEGMENT directive

- symbol SEGMENT segment_type
- segment type can be one of:
  - CODE code
  - DATA directly addressable internal data
  - IDATA indirectly addressable internal data
  - BIT bit
  - XDATA external data
  - CONST constant, typically stored in code space

Example – hardware ports

- C

```
code char x;
xdata PORTA _at_ 0x4000;
```

- Assembly

```
myrom   SEGMENT CODE
        RSEG myrom      ;relocatable cseg
x:      DS 1            ;define one byte
XSEG    AT 4000h        ;absolute segment
PORTA:  DS 1            ;1 byte port at 4000
```

Usage of Segments

<name segment and declare type>

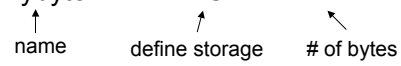
<declare beginning of segment>

<segment contents (code, vars, consts)>

<next segment>

Labels (includes variables)

- Variable are declared in segment.
- Ex: mybyte: DS 1



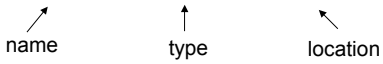
- In program: _____
- Assembler codes as: _____
- ____ is the location chosen by assembler.

Storage Definition Directives

- DS define storage byte
- DSB define storage byte
- DSW define storage word
- DSD define storage double
- DBIT define storage bit

Variables declared outside of segments

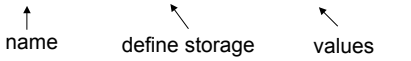
- Can be of types: DATA, IDATA, CODE, XDATA and BIT
- Ex: another DATA 7FH



Constants in Code Space

- Example:

```
CSEG      AT      0100H
lookup:   DB      1,2,3,4,5,42
```



- Storage definition could also be of type DW or DD.

More Assembler Directives

- EXTERN: declare variables from other modules (ASM files)
- PUBLIC: declare variable to be used in other module
- USING: tell compiler current register bank
  - Ex:
- EQU: create “assembler” constant
  - Ex:
- END: last statement in source file

Immediate Data

Example:

- MOV A, #2AH ; hex
- MOV A, ____ ; decimal
- MOV A, _____ ; binary
- MOV A, _____ ; _____ is default
- MOV A, _____

SFRs

- Example:
  - SETB D7H
  - SETB ____
- Example:
  - SETB D0H
  - SETB ____
- SFRs can generally be referenced by name.

General Program Layout

- Data Segments
  - declare segments
  - declare variables
- Code segments
  - declare segments
  - declare constants