# CpE 213
# Digital Systems Design
## 8051 Assembly Programming

Lecture 13

Wednesday 9/21/2005

UMR UNIVERSITY OF MISSOURI-ROLLA

The Name. The Degree. The Difference.

## Overview

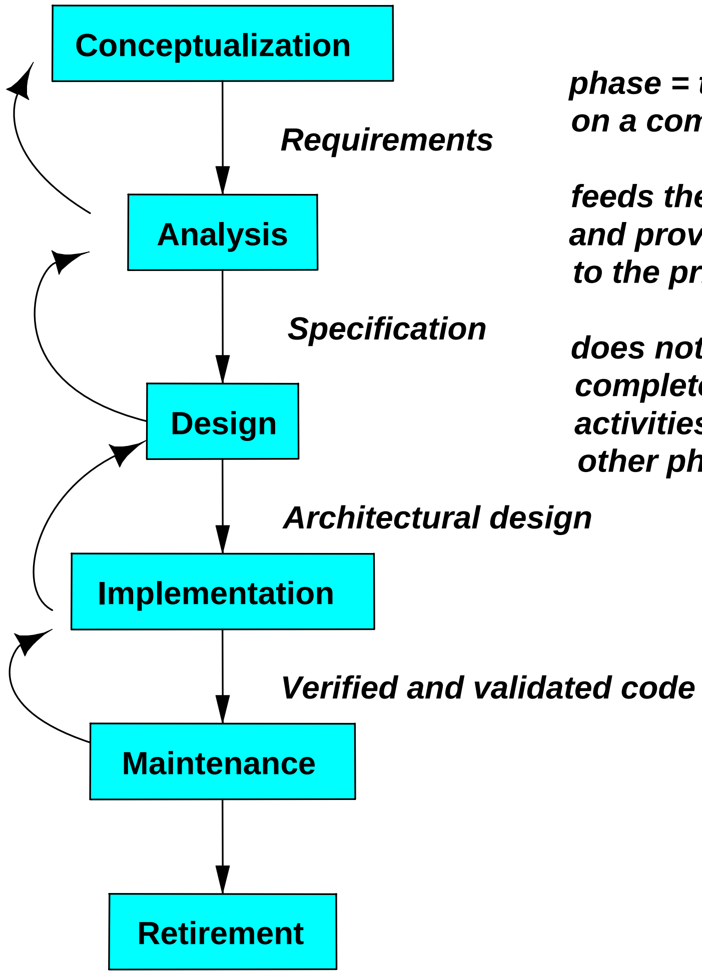
- 8051 assembly programming
  - Steps necessary for program development.
  - Motivation for studying assembly language programming.
  - Describe the components of a software development system.
  - Explain the functions of: Editor, Assembler, Linker, Monitor and Simulator.
  - Basics of the 8051 assembly language.
  - Use of assembler directives.
- Keil demo

# 8051 Assembly Programming

Some slides adapted from Mr. K. T. Ng and Dr. H. J. Pottinger

# Generic Software Development Cycle

*Applies to any software development:*
- *large-scale applications*
- *assemblies of components*
- *fine-grain components*
- *other software parts*



*phase = tasks focused on a common theme.*

*feeds the next phase and provides feedback to the prior phase.*

*does not have to be complete before activities start in other phases*

# Machine Language vs. Assembly

```
7D 25 7F 34 74 00 2D        MOV   R5,#25H
                            MOV   R7,#34H
                            MOV   A,#0
                            ADD   A,R5

Machine Language            Assembly Language
```

- Machine language is a binary sequence interpreted by the computer as a sequence of instructions.
- Machine instructions show basic capabilities of 8051.
- Assembly language (ASM) is built around these capabilities.
- Assembly language is a form of machine language that uses mnemonics (memory aids) for the instruction 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).

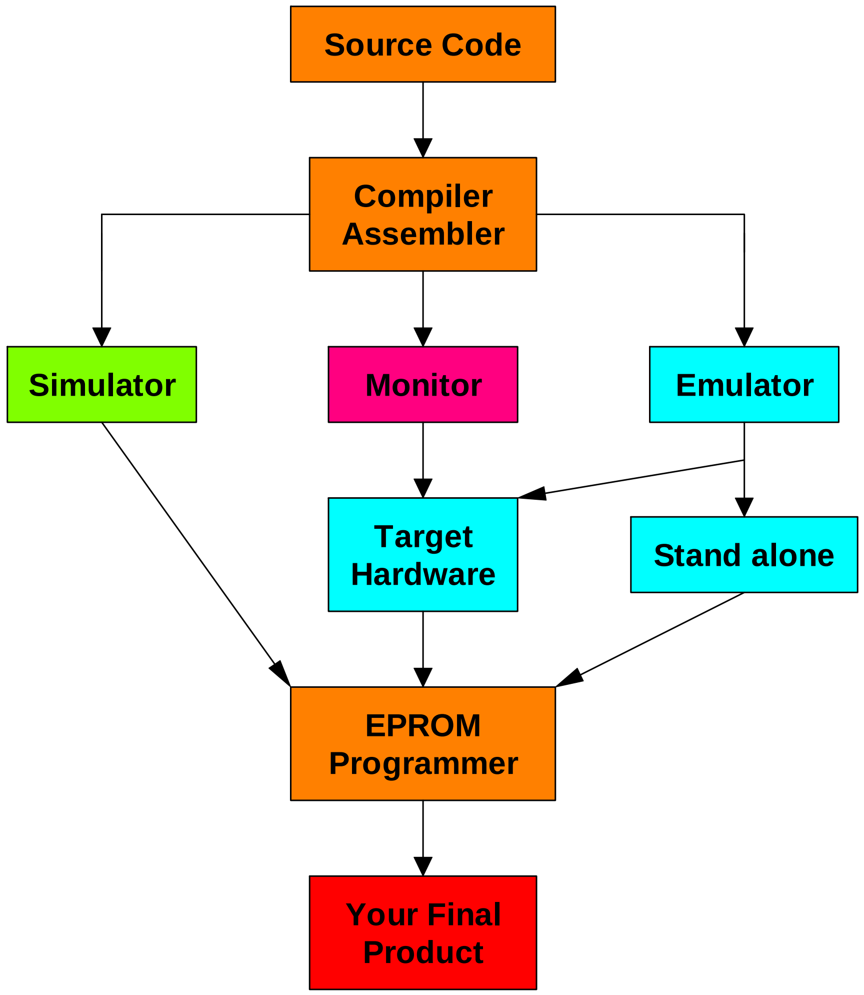
## Disadvantages of Assembly Language

- 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 as compared to a high-level language.
- Effective assembly programming typically requires a lot of programming experience.

## Use of Assembly Language

- The use of assembly 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

- Program development utilities enable the user to write, assemble, and test assembly language programs. They include:
  - Editor
  - Assembler
  - Linking Loader
  - Monitor
  - Simulator

## Types of Assemblers

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

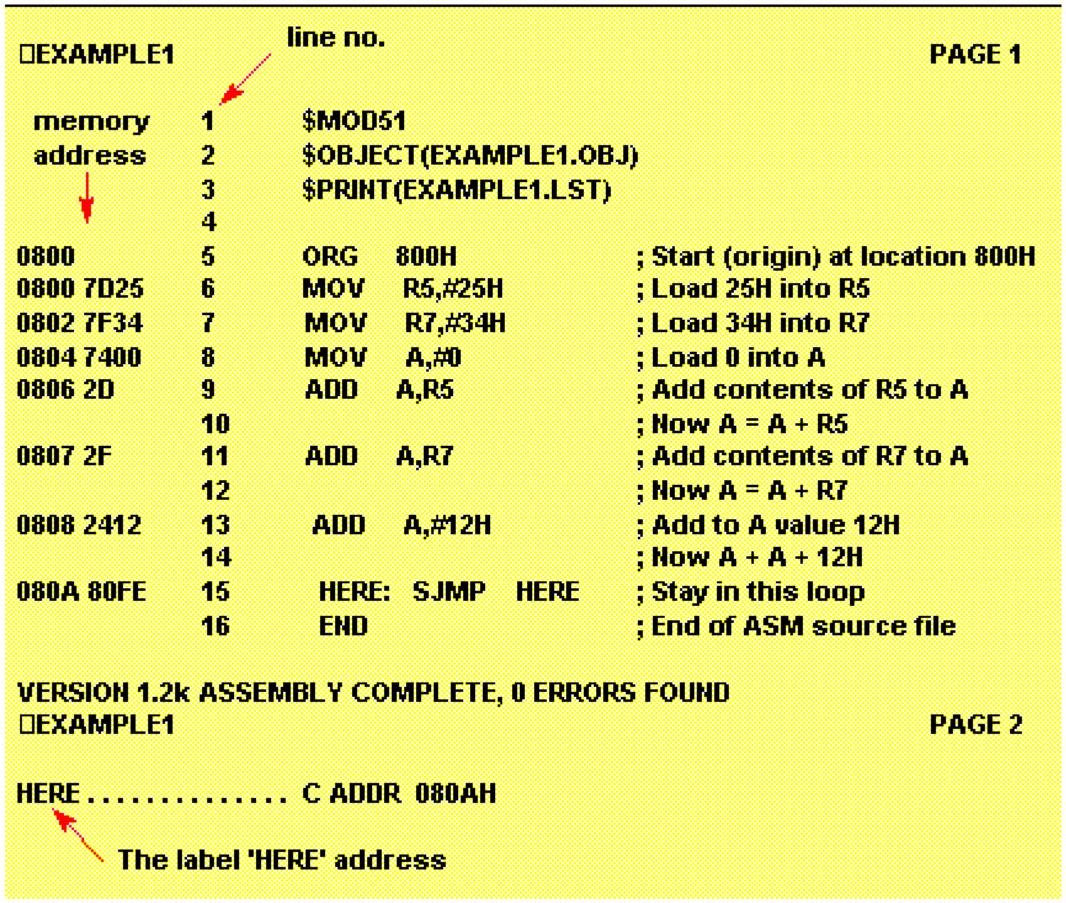
# Assembler Source File

```
$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 )

# Assembler Listing File

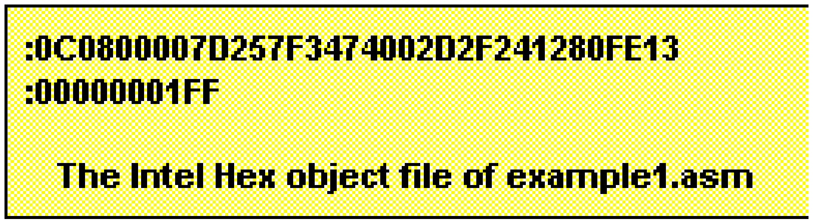


```
EXAMPLE1                                                      PAGE 1

memory     1       $MOD51
address    2       $OBJECT(EXAMPLE1.OBJ)
           3       $PRINT(EXAMPLE1.LST)
           4
0800       5       ORG    800H          ; Start (origin) at location 800H
0800 7D25  6       MOV    R5,#25H       ; Load 25H into R5
0802 7F34  7       MOV    R7,#34H       ; Load 34H into R7
0804 7400  8       MOV    A,#0          ; Load 0 into A
0806 2D    9       ADD    A,R5          ; Add contents of R5 to A
           10                           ; Now A = A + R5
0807 2F    11      ADD    A,R7          ; Add contents of R7 to A
           12                           ; Now A = A + R7
0808 2412  13      ADD    A,#12H        ; Add to A value 12H
           14                           ; Now A + A + 12H
080A 80FE  15      HERE:  SJMP  HERE    ; Stay in this loop
           16      END                  ; End of ASM source file

VERSION 1.2k ASSEMBLY COMPLETE, 0 ERRORS FOUND
EXAMPLE1                                                      PAGE 2

HERE . . . . . . . . . . . . . . C ADDR  080AH
```

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

(produced by Metalink 8051 cross assembler )

## Assembler Object File



```
:0C0800007D257F3474002D2F241280FE13
:00000001FF
```

The Intel Hex object file of example1.asm

- One standard for storing machine language programs in a displayable or printable format is known as "Intel hexadecimal format".

## Intel HEX Format

- The "Intel-Standard" HEX file is one of the most popular and commonly used formats in the 8051 world.
- The standard is used to burn the 8051 program into an EPROM, PROM, etc. For example, an 8052 assembler will usually generate an Intel Standard HEX file which can then be loaded into an EPROM programmer and burned into the chip.

## Intel HEX File

- An Intel Standard HEX file is an ASCII file with one "record" per line. Each line has the following format:

| Position | Description |
|---|---|
| **1** | **Record Marker**: The first character of the line is always a colon (ASCII 0x3A) to identify the line as an Intel HEX file |
| **2 – 3** | **Record Length**: This field contains the number of data bytes in the register represented as a 2-digit hex number. This is the total number of *data* bytes, not including the checksum byte nor the first 9 characters of the line. |
| **4 - 7** | **Address**: This field contains the address (in hex) where the data should be loaded into the chip |
| **8 - 9** | **Record Type**: This field indicates the type of record for this line. The possible values are: **00**=Register contains normal data. **01**=End of File. **02**=Extended address. |

(continued)

## Intel HEX File (continued)

| Position | Description |
|---|---|
| **10 - ?** | **Data Bytes**: The following bytes are the actual data that will be burned into the EPROM. The data is represented as 2-digit hex values. |
| **Last 2 characters** | **Checksum**: The last two characters of the line are a checksum for the line. The checksum value is calculated by taking the two's complement of the sum of all the preceding data bytes, excluding the checksum byte itself and the colon at the beginning of the line. |

# Intel Hex format

```
:0B000300E4FF0FBF14FC7F14DFFE22 9F
:030000000002000E ED
:0C000E00787FE4F6D8FD758107020003 3E
:00000001 FF
```

- Byte count + start address + record type + data + checksum
- Checksum ED = -(03+02+0E)
- Question: What does this do for the first 100 microsec or so?

# Disassembly example

```
0000 00 02000E ED
000E 00 787F E4 F6 D8FD 758107 020003
```

```
0000 02 000E LJMP 000E
000E 78 7F   MOV R0,#7f
0010 E4      CLR A
0011 F6      MOV @R0,A
0012 D8 FD   DJNZ R0,-3 ;jmp 0011
0014 75 8107 MOV SP, #7 ;sp=sfr 81
0017 02 0003 LJMP 3
```

## 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.
  - We will use the Keil development environment, which includes an editor, a monitor, a simulator, and other tools.
- 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 | `ADD A,#7H` |
| Pseudo-operations | `DS 32` |
| Assembler directives | `$MOD51` |

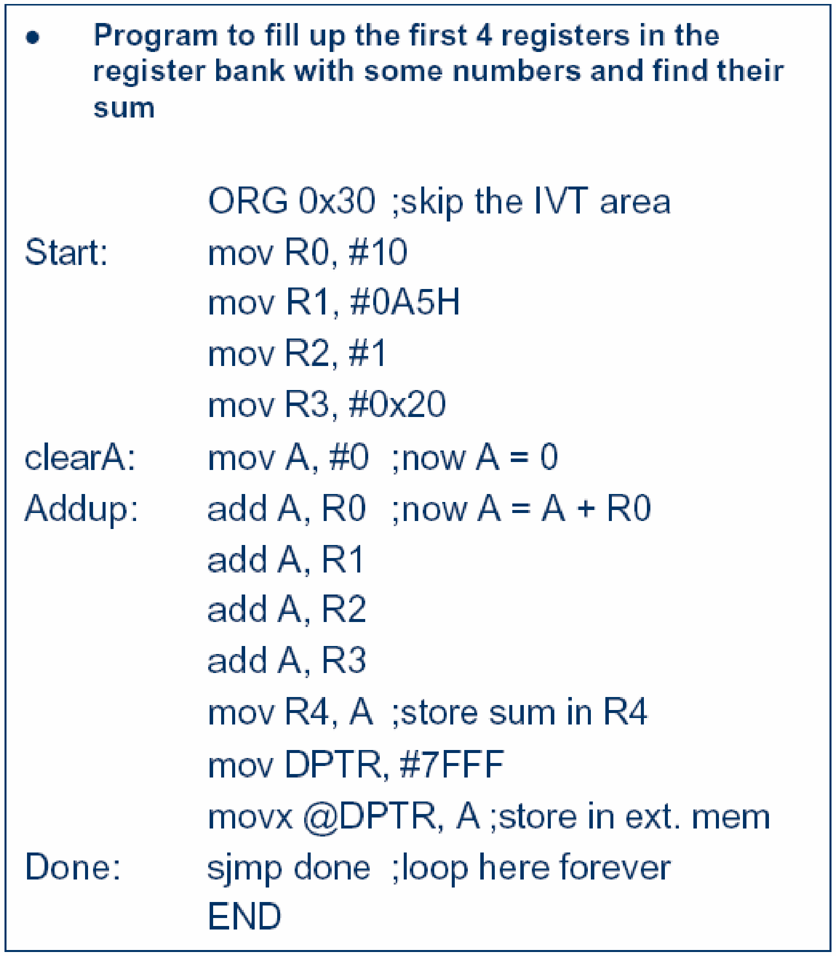
- Assembly instructions are translated into machine language instructions to be stored in program memory and executed by the CPU.
- Pseudo-operations allocate space for variables or constants.
- Assembler directives are commands to the assembler program.

# 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, which will facilitate maintenance.
  - “Always code as if the guy that ends up maintaining and/or testing your code is a violent psychopath who knows exactly where you live.”

## 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) |
| less: | (valid) |
| goback: | (valid) |
| delay1: | (valid) |
| 1ST_VARIABLE: | (invalid) |
| alpha#: | (invalid) |
| MOV | (invalid) |
| LOW | (invalid: assembly operator) |
| DATA | (invalid: assembly directive) |

## 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.

  -
    ```
    MOV    R1,#23H     ;R1,#23H is the operand field
    HERE:  SJMP HERE   ;HERE is the operand field
    EQU    10          ;10 is the operand field of the
                       ;assembler directive EQU
    ```

## 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

- An ASM directive (or pseudo-opcode) has the same format as assembler instruction (opcode), but it is NOT assembled into machine code.
- Directives 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.
- List of assembler directives for Keil is posted on Blackboard.

# 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
  - Example: `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
  - Example: `RSEG segment_name`
    - segment_name needs to be previously defined with the SEGMENT directive
  - Example:

```
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

- name 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)

- Variables are declared within segments.
- Ex: `mybyte:      DS      1`

  name      define storage      # of bytes

- In program: `MOV mybyte,#42H`
- Assembler codes as: `MOV 2AH,#42H`
- 2AH 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
   ↑      ↑      ↑
 name    type  location
```

## Constants in Code Space

- Example:

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

name define storage values

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

## General Program Layout

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

# 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:
    ```
    USING 1
    MOV A,R0
    ```
  - Will still be switching banks with PSW.
- EQU: create “assembler” constant
  - Ex:
    ```
    answer EQU  #42H
    MOV A,answer
    ```
- END: last statement in source file

## Immediate Data

Examples:

- `MOV A,#2AH              ;hex`
- `MOV A,#42D              ;decimal`
- `MOV A,#0010 1010B       ;binary`
- `MOV A,#42               ;decimal is default`
- `MOV A,#41+1`

## SFRs

- Examples:
  - `SETB D7H`
  - `SETB PSW.7`
- Examples:
  - `SETB D0H`
  - `SETB CY`
- SFRs can generally be referenced by name.

## Keil Software

- Used to create and simulate programs for the 8051.
- Can be downloaded from:

  [http://www.keil.com/demo/eval/c51.htm](http://www.keil.com/demo/eval/c51.htm)
- Will be briefly introduced in class.
- Read handouts posted on Blackboard and chapters 4 and 5 of the uVision2 Getting Started (from Tool’s Users Guide in application).

## For Friday

- Review today’s lecture notes and Chapter 2 of your textbook.
- Download handouts on Keil.
- Practice working with Keil.
- Finish assignment 4.