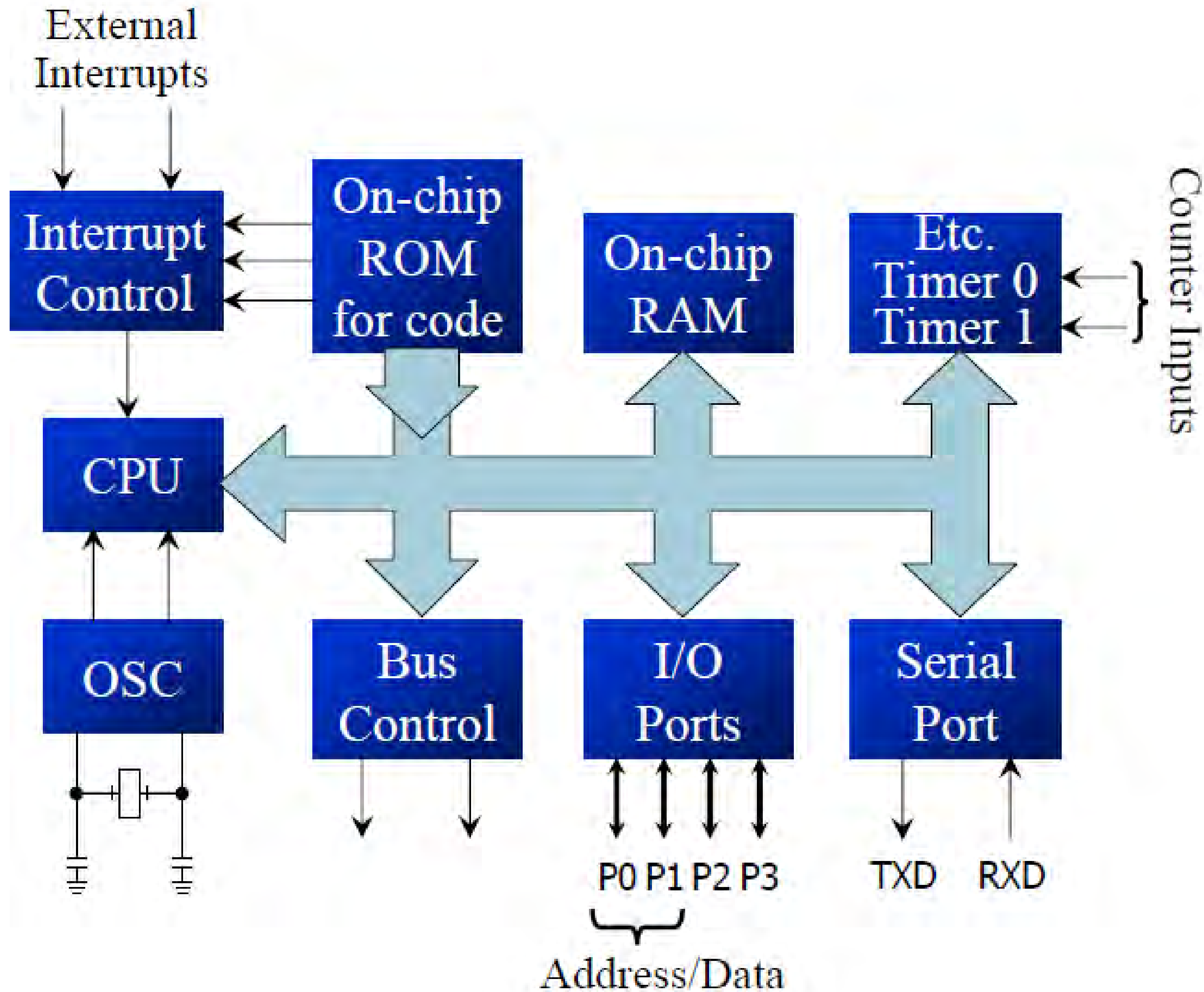
SELECTION OF MICROCONTROLLER

- Meeting the computing needs of the task at hand efficiently and cost effectively
  - Speed
  - Packaging
  - Power consumption
  - The amount of RAM and ROM on chip
  - The number of I/O pins and the timer on chip
  - How easy to upgrade to higher-performance or lower power-consumption versions

MICROCONTROLLER 8051

- Intel introduced 8051, referred as MCS-51, in 1981
  - The 8051 is an 8-bit processor
    - The CPU can work on only 8 bits of data at a time
  - The 8051 had
    - 128 bytes of RAM
    - 4K bytes of on-chip ROM
    - Two timers
    - One serial port
    - Four I/O ports, each 8 bits wide
    - 6 interrupt sources
- The 8051 became widely popular after allowing other manufactures to make and market any flavor of the 8051, but remaining code-compatible

OVERVIEW OF 8051 FAMILY



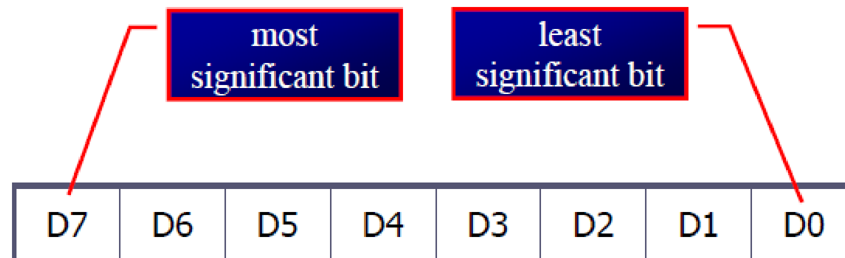
DIFFERENT MICROCONTROLLERS

- The 8051 is a subset of the 8052
- The 8031 is a ROM-less 8051
  - Add external ROM to it
  - You lose two ports, and leave only 2 ports for I/O operations

Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

INSIDE THE 8051

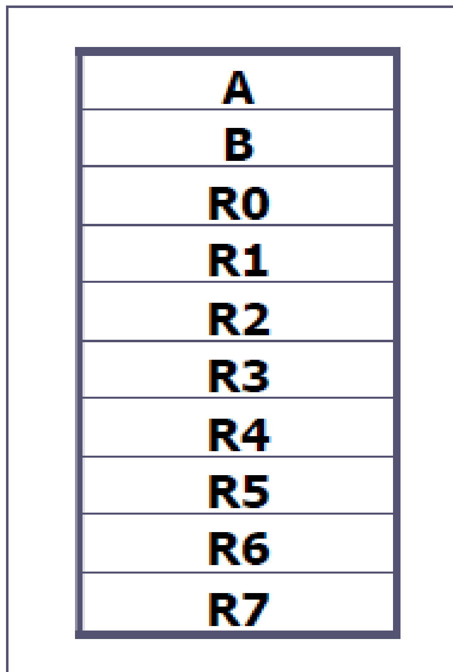
- Register are used to store information temporarily, while the information could be
  - a byte of data to be processed, or
  - an address pointing to the data to be fetched
- The vast majority of 8051 register are 8-bit registers
  - There is only one data type, 8 bits
- The 8 bits of a register are shown from MSB D7 to the LSB D0
  - With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed

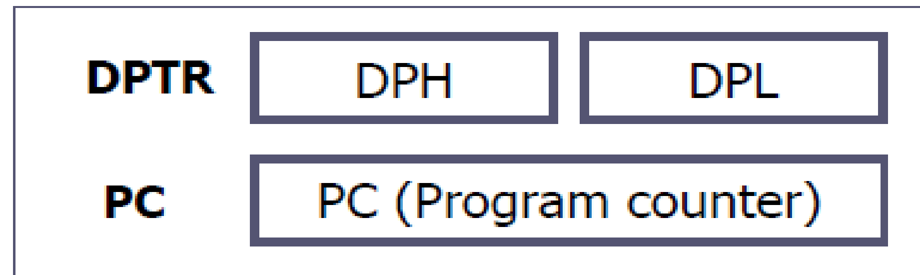


8 bit Registers

REGISTERS

- The most widely used registers
  - A (Accumulator)
    - For all arithmetic and logic instructions
  - B, R0, R1, R2, R3, R4, R5, R6, R7
  - DPTR (data pointer), and PC (program counter)





ASSEMBLY LANGUAGE - MOV INSTRUCTION

`MOV destination, source` ;copy source to dest.

- The instruction tells the CPU to move (in reality, COPY) the source operand to the destination operand

“#” signifies that it is a value

```
MOV   A,#55H     ;load value 55H into reg. A
MOV   R0,A       ;copy contents of A into R0
                 ;(now A=R0=55H)
MOV   R1,A       ;copy contents of A into R1
                 ;(now A=R0=R1=55H)
MOV   R2,A       ;copy contents of A into R2
                 ;(now A=R0=R1=R2=55H)
MOV   R3,#95H    ;load value 95H into R3
                 ;(now R3=95H)
MOV   A,R3       ;copy contents of R3 into A
                 ;now A=R3=95H
```

MOV INSTRUCTION

- Value (proceeded with #) can be loaded directly to registers A, B, or R0 – R7
  - `MOV A, #23H`
  - `MOV R5, #0F9H`

Add a 0 to indicate that F is a hex number and not a letter

If it's not preceded with #, it means to load from a memory location

- If values 0 to F moved into an 8-bit register, the rest of the bits are assumed all zeros
  - "`MOV A, #5`", the result will be A=05; i.e., A = 00000101 in binary
- Moving a value that is too large into a register will cause an error
  - `MOV A, #7F2H` ; ILLEGAL: 7F2H>8 bits (FFH)

ADD INSTRUCTION

```
ADD A, source      ;ADD the source operand
                   ;to the  accumulator
```

- The ADD instruction tells the CPU to add the source byte to register A and put the result in register A
- Source operand can be either a register or immediate data, but the destination must always be register A
  - "ADD R4, A" and "ADD R2, #12H" are invalid since A must be the destination of any arithmetic operation

```
MOV A, #25H       ;load 25H into A
MOV R2, #34H      ;load 34H into R2
ADD A, R2 ;add R2 to Accumulator
                  ;(A = A + R2)
```

```
MOV A, #25H       ;load one operand
          ;into A (A=25H)
ADD A, #34H       ;add the second
                  ;operand 34H to A
```

STRUCTURE OF ASSEMBLY LANGUAGE

- Assembly language instruction includes
  - a mnemonic (abbreviation easy to remember)
    - the commands to the CPU, telling it what those to do with those items
  - optionally followed by one or two operands
    - the data items being manipulated
- A given Assembly language program is a series of statements, or lines
  - Assembly language instructions
    - Tell the CPU what to do
  - Directives (or pseudo-instructions)
    - Give directions to the assembler

ASSEMBLY LANGUAGE

- An Assembly language instruction consists of four fields:

```
[label:] Mnemonic [operands] [;comment]
```

```
      ORG   0H              ;start(origin) at location 0
      MOV   R5, #25H        ;load 25H into R5
      MOV   R7, #34H        ;load 34H in
      MOV   A, #0           ;load 0 into
      ADD   A, R5           ;add content
                            ;now A = A +
      ADD   A, R7           ;add contents of R7 to A
                            ;now A = A + R7
      ADD   A, #12H         ;add to A value 12H
                            ;now A = A + 12H
HERE: SJMP  HERE            ;stay in this loop
      END                   ;en
```

Directives do not generate any machine code and are used only by the assembler

Mnemonics produce opcodes

Comments may be at the end of a line or on a line by themselves The assembler ignores comments

The label field allows the program to refer to a line of code by name

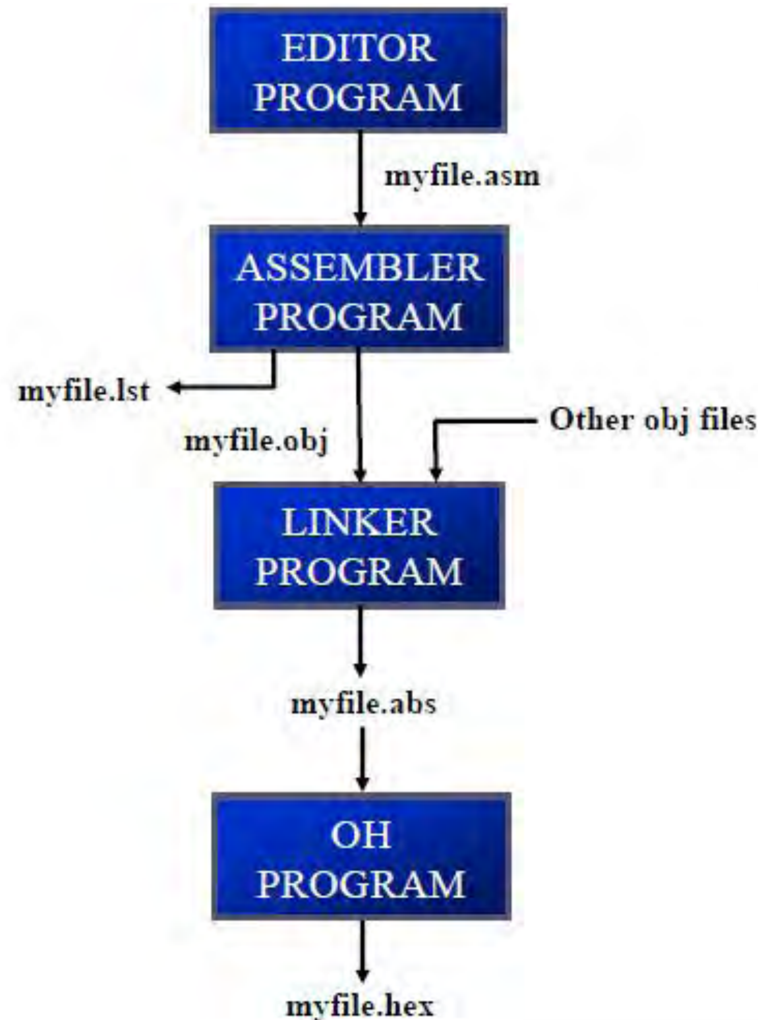
ASSEMBLE AND RUN

- The step of Assembly language program are outlines as follows:
  1) First we use an editor to type a program, many excellent editors or word processors are available that can be used to create and/or edit the program
     - Notice that the editor must be able to produce an ASCII file
     - For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension “asm“ or “src”, depending on which assembly you are using

ASSEMBLE AND RUN

2) The “asm” source file containing the program code created in step 1 is fed to an 8051 assembler
   - The assembler converts the instructions into machine code
   - The assembler will produce an object file and a list file
   - The extension for the object file is “obj” while the extension for the list file is “lst”
3) Assembler require a third step called *linking*
   - The linker program takes one or more object code files and produce an absolute object file with the extension “abs”
   - This abs file is used by 8051 trainers that have a monitor program

4) Next the "abs" file is fed into a program called "OH" (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM
   - This program comes with all 8051 assemblers
   - Recent Windows-based assemblers combine step 2 through 4 into one step



LIST FILE

- The lst (list) file, which is optional, is very useful to the programmer
  - It lists all the opcodes and addresses as well as errors that the assembler detected
  - The programmer uses the lst file to find the syntax errors or debug

```
1 0000            ORG 0H          ;start (origin) at 0
2 0000    7D25    MOV R5,#25H     ;load 25H into R5
3 0002    7F34    MOV R7,#34H     ;load 34H into R7
4 0004    7400    MOV A,#0        ;load 0 into A
5 0006    2D      ADD A,R5        ;add contents of R5 to A
                                  ;now A = A + R5
6 0007    2F      ADD A,R7        ;add contents of R7 to A
                                  ;now A = A + R7
7 0008    2412    ADD A,#12H      ;add to A value 12H
                                  ;now A = A + 12H
8 000A    80EF HERE: SJMP HERE;stay in this loop
9 000C            END             ;end of asm source file
```

PROGRAM COUNTER

- The program counter points to the address of the next instruction to be executed
  - As the CPU fetches the opcode from the program ROM, the program counter is increasing to point to the next instruction
- The program counter is 16 bits wide
  - This means that it can access program addresses 0000 to FFFFH, a total of 64K bytes of code
- All 8051 members start at memory address 0000 when they're powered up
  - Program Counter has the value of 0000
  - The first opcode is burned into ROM address 0000H, since this is where the 8051 looks for the first instruction when it is booted
  - We achieve this by the `ORG` statement in the source program

CODING TO ROM

- Examine the list file and how the code is placed in ROM

```
1 0000              ORG 0H            ;start (origin) at 0
2 0000    7D25      MOV R5,#25H       ;load 25H into R5
3 0002    7F34      MOV R7,#34H       ;load 34H into R7
4 0004    7400      MOV A,#0          ;load 0 into A
5 0006    2D        ADD A,R5          ;add contents of R5 to A
                                      ;now A = A + R5
6 0007    2F        ADD A,R7          ;add contents of R7 to A
                                      ;now A = A + R7
7 0008    2412      ADD A,#12H        ;add to A value 12H
                                      ;now A = A + 12H
8 000A    80EF      HERE: SJMP HERE   ;stay in this loop
9 000C              END               ;end of asm source file
```

ROM Address	Machine Language	Assembly Language
0000	7D25	MOV R5, #25H
0002	7F34	MOV R7, #34H
0004	7400	MOV A, #0
0006	2D	ADD A, R5
0007	2F	ADD A, R7
0008	2412	ADD A, #12H
000A	80EF	HERE: SJMP HERE

- After the program is burned into ROM, the opcode and operand are placed in ROM memory location starting at 0000

ROM contents

Address	Code
0000	7D
0001	25
0002	7F
0003	34
0004	74
0005	00
0006	2D
0007	2F
0008	24
0009	12
000A	80
000B	FE

PROGRAM EXECUTION

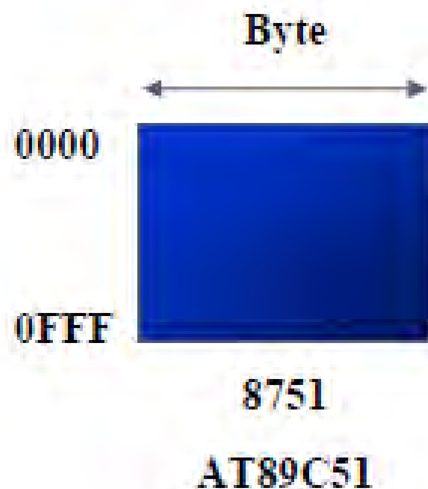
- A step-by-step description of the action of the 8051 upon applying power on it
  1. When 8051 is powered up, the PC has 0000 and starts to fetch the first opcode from location 0000 of program ROM
     - Upon executing the opcode 7D, the CPU fetches the value 25 and places it in R5
     - Now one instruction is finished, and then the PC is incremented to point to 0002, containing opcode 7F
  2. Upon executing the opcode 7F, the value 34H is moved into R7
     - The PC is incremented to 0004

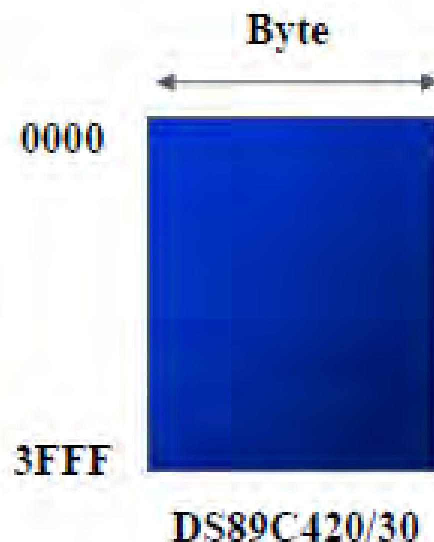
PROGRAM EXECUTION

- (cont')
  3. The instruction at location 0004 is executed and now PC = 0006
  4. After the execution of the 1-byte instruction at location 0006, PC = 0007
  5. Upon execution of this 1-byte instruction at 0007, PC is incremented to 0008
     - This process goes on until all the instructions are fetched and executed
     - The fact that program counter points at the next instruction to be executed explains some microprocessors call it the *instruction pointer*

ROM MEMORY MAP

- No member of 8051 family can access more than 64K bytes of opcode
  - The program counter is a 16-bit register





DATA TYPE

- 8051 microcontroller has only one data type - 8 bits
  - The size of each register is also 8 bits
  - It is the job of the programmer to break down data larger than 8 bits (00 to FFH, or 0 to 255 in decimal)
  - The data types can be positive or negative

DIRECTIVES

- The DB directive is the most widely used data directive in the assembler
  - It is used to define the 8-bit data
  - When DB is used to define data, the numbers can be in decimal, binary, hex, ASCII formats

```
        ORG    500H
DATA1:  DB     28              ;DECIMAL (1C in Hex)
DATA2:  DB     00110101B       ;BINARY (35 in Hex)
DATA3:  DB     39H             ;HEX
        ORG    510H
DATA4:  DB     "2591"
        ORG    518H
DATA6:  DB     "My name is Joe"
                               ;ASCII CHARACTERS
```

The "D" after the decimal number is optional, but using "B" (binary) and "H" (hexadecimal) for the others is required

The Assembler will convert the numbers into hex

Place ASCII in quotation marks The Assembler will assign ASCII code for the numbers or characters

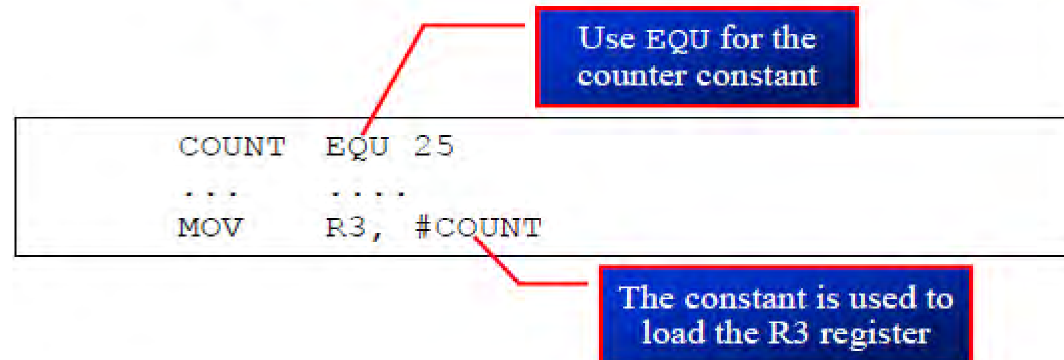
Define ASCII strings larger than two characters

DIRECTIVES

- ORG (origin)
  - The ORG directive is used to indicate the beginning of the address
  - The number that comes after ORG can be either in hex and decimal
    - If the number is not followed by H, it is decimal and the assembler will convert it to hex
- END
  - This indicates to the assembler the end of the source (asm) file
  - The END directive is the last line of an 8051 program
    - Mean that in the code anything after the END directive is ignored by the assembler

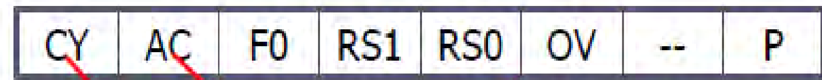
DIRECTIVES

- EQU (equate)
  - This is used to define a constant without occupying a memory location
  - The EQU directive does not set aside storage for a data item but associates a constant value with a data label
    - When the label appears in the program, its constant value will be substituted for the label
  - Assume that there is a constant used in many different places in the program, and the programmer wants to change its value throughout
    - By the use of EQU, one can change it once and the assembler will change all of its occurrences



PROGRAM STATUS WORD

- The program status word (PSW) register, also referred to as the *flag register*, is an 8 bit register
  - Only 6 bits are used
    - These four are CY (*carry*), AC (*auxiliary carry*), P (*parity*), and OV (*overflow*)
      - They are called *conditional flags*, meaning that they indicate some conditions that resulted after an instruction was executed
    - The PSW3 and PSW4 are designed as RS0 and RS1, and are used to change the bank
  - The two unused bits are user-definable



A carry from D3 to D4

Carry out from the d7 bit

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
--	PSW.5	Available to the user for general purpose
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
--	PSW.1	User definable bit.
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of 1 bits in the accumulator.

Reflect the number of 1s in register A

The result of signed number operation is too large, causing the high-order bit to overflow into the sign bit

RS1	RS0	Register Bank	Address
0	0	0	00H – 07H
0	1	1	08H – 0FH
1	0	2	10H – 17H
1	1	3	18H – 1FH

Instructions that affect flag bits

Instruction	CY	OV	AC
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RPC	X		
PLC	X		
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, bit	X		
ANL C, /bit	X		
ORL C, bit	X		
ORL C, /bit	X		
MOV C, bit	X		
CJNE	X		

PRACTICE PROBLEMS

Show the status of the CY, AC and P flag after the addition of 38H and 2FH in the following instructions.

```
MOV A, #38H
ADD A, #2FH ;after the addition A=67H, CY=0
```

Solution:

38	00111000
+ 2F	00101111
67	01100111

CY = 0 since there is no carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bi

P = 1 since the accumulator has an odd number of 1s (it has five 1s)

PRACTICE PROBLEMS

Show the status of the CY, AC and P flag after the addition of 9CH and 64H in the following instructions.

```
MOV A, #9CH

ADD A, #64H  ;after the addition A=00H, CY=1
```

Solution:

9C	10011100
+ 64	01100100
100	00000000

CY = 1 since there is a carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bi

P = 0 since the accumulator has an even number of 1s (it has zero 1s)

PRACTICE PROBLEMS

Show the status of the CY, AC and P flag after the addition of 88H and 93H in the following instructions.

```
MOV A, #88H

ADD A, #93H  ;after the addition A=1BH, CY=1
```

Solution:

	Hex	Binary
	88	10001000
+	93	10010011
	11B	00011011

CY = 1 since there is a carry beyond the D7 bit

AC = 0 since there is no carry from the D3 to the D4 bi

P = 0 since the accumulator has an even number of 1s (it has four 1s)

NEXT - MEMORY MAPPING AND ADDRESSING MODES