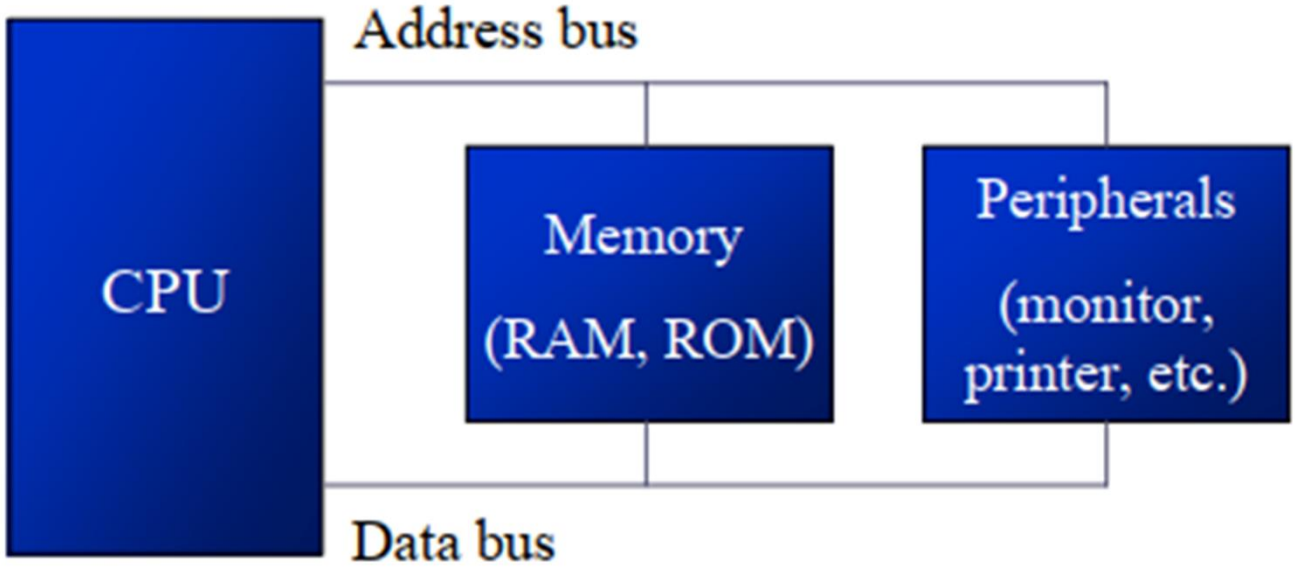
8051-Microcontroller
Assembly Language and Embedded C

Dr. V. Ramesh Kumar
Asst. Prof. Dept of ECE
IIIT Sri City

Outline

- Inside the Computer
- Microcontroller
- Difference Between Microcontroller & Microprocessor
- Assembly Language
- Embedded C

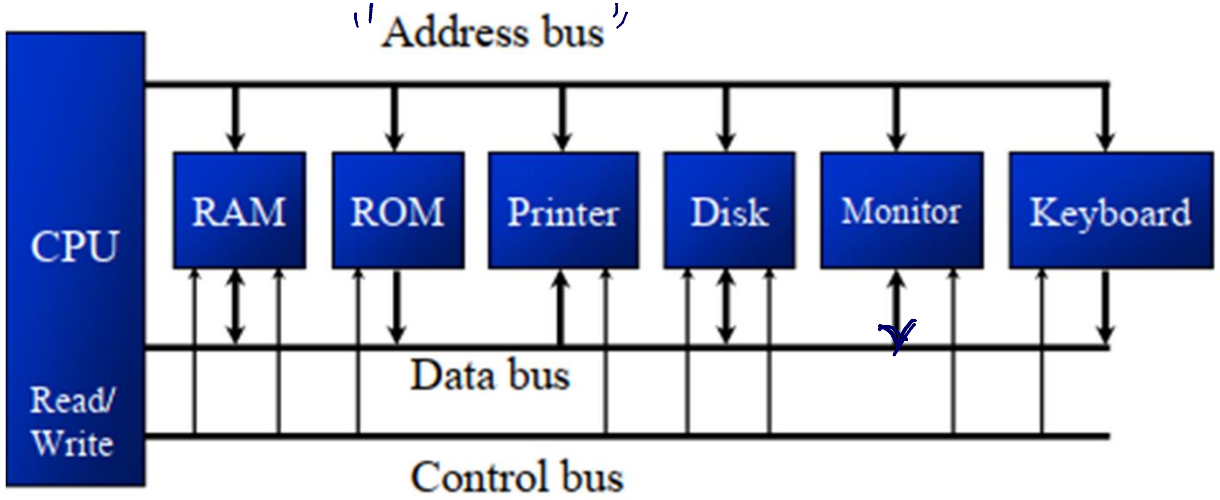
Inside the Computer



Internal Organization of Computer

- CPU (Central Processing Unit)
  - Execute information stored in memory
- I/O (Input/output) devices
  - Provide a means of communicating with CPU
- Memory
  - RAM (Random Access Memory) – temporary storage of programs that computer is running
    - The data is lost when computer is off
  - ROM (Read Only Memory) – contains programs and information essential to operation of the computer
    - The information cannot be changed by use, and is not lost when power is off
      - It is called *nonvolatile memory*

- The CPU is connected to memory and I/O through strips of wire called a *bus*
  - Carries information from place to place
    - Address bus
    - Data bus
    - Control bus



Address bus

For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address

The address assigned to a given device must be unique

The CPU puts the address on the address bus, and the decoding circuitry finds the device

Data bus

The CPU either gets data from the device or sends data to it

Control bus

Provides read or write signals to the device to indicate if the CPU is asking for information or sending it information

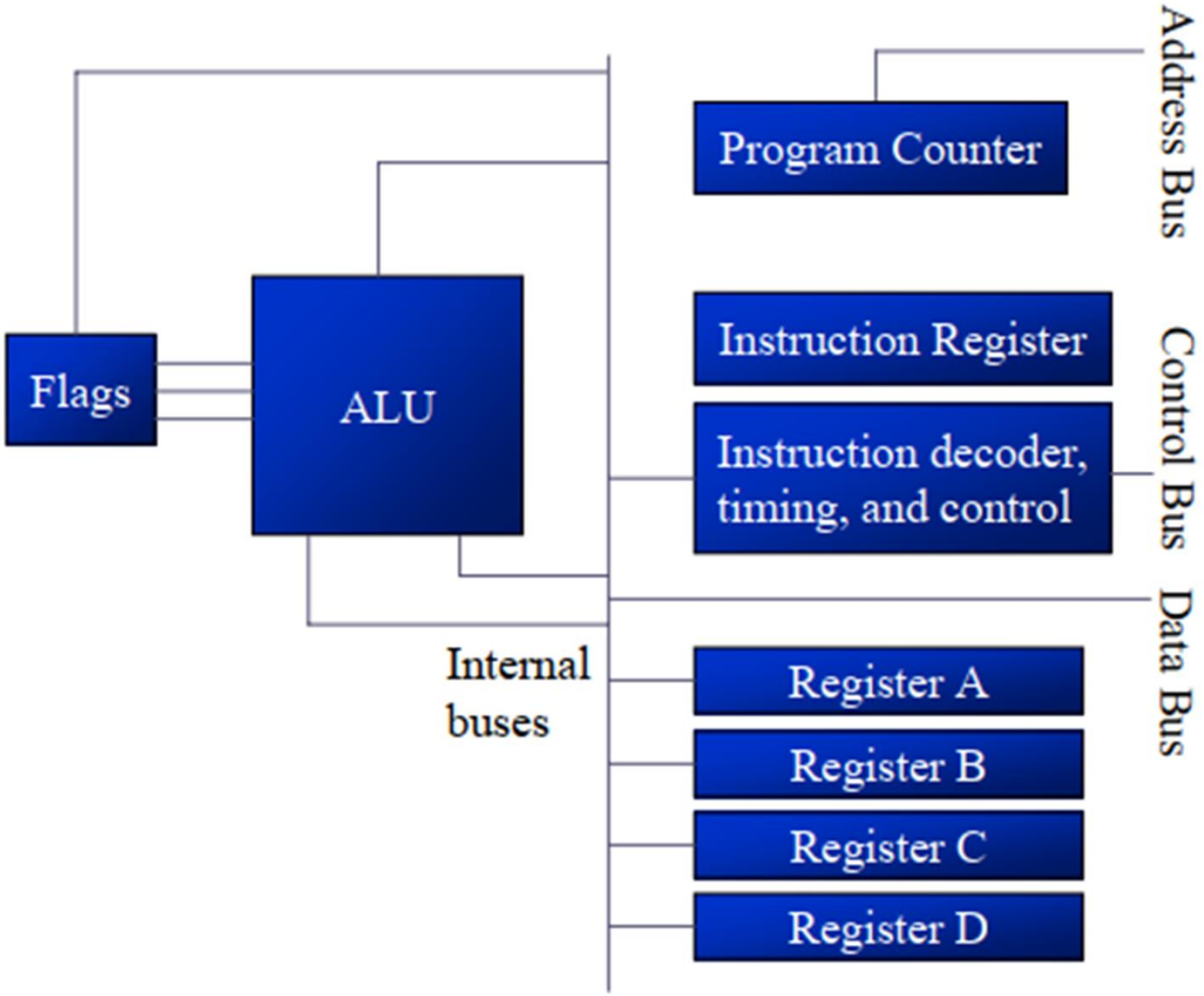
Data Bus

- The more data buses available, the better the CPU
  - Think of data buses as highway lanes
- More data buses mean a more expensive CPU and computer
  - The average size of data buses in CPUs varies between 8 and 64
- Data buses are bidirectional
  - To receive or send data
- The processing power of a computer is related to the size of its buses

- For the CPU to process information, the data must be stored in RAM or ROM, which are referred to as *primary memory*
- ROM provides information that is fixed and permanent
  - Tables or initialization program
- RAM stores information that is not permanent and can change with time
  - Various versions of OS and application packages
  - CPU gets information to be processed
    - first form RAM (or ROM)
    - if it is not there, then seeks it from a mass storage device, called *secondary memory*, and transfers the information to RAM

- Registers
  - The CPU uses registers to store information temporarily
    - Values to be processed
    - Address of value to be fetched from memory
  - In general, the more and bigger the registers, the better the CPU
    - Registers can be 8-, 16-, 32-, or 64-bit
    - The disadvantage of more and bigger registers is the increased cost of such a CPU

Inside CPU


Address Bus
Program Counter
Instruction Register
Control Bus
Flags
ALU
Instruction decoder, timing, and control
Data Bus
Internal buses
Register A
Register B
Register C
Register D

- ALU (arithmetic/logic unit)
  - Performs arithmetic functions such as add, subtract, multiply, and divide, and logic functions such as AND, OR, and NOT
- Program counter
  - Points to the address of the next instruction to be executed
    - As each instruction is executed, the program counter is incremented to point to the address of the next instruction to be executed
- Instruction decoder
  - Interprets the instruction fetched into the CPU
    - A CPU capable of understanding more instructions requires more transistors to design

Ex. A CPU has registers A, B, C, and D and it has an 8-bit data bus and a 16-bit address bus. The CPU can access memory from addresses 0000 to FFFFH

Assume that the code for the CPU to move a value to register A is B0H and the code for adding a value to register A is 04H

The action to be performed by the CPU is to put 21H into register A, and then add to register A values 42H and 12H

...

Ex. (cont')

Action	*Code*	*Data*
Move value 21H into reg. A	B0H	21H
Add value 42H to reg. A	04H	42H
Add value 12H to reg. A	04H	12H

Mem. addr.	*Contents of memory address*
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

...

How these instructions are working?

Ex. (cont')

The actions performed by CPU are as follows:

1. The program counter is set to the value 1400H, indicating the address of the first instruction code to be executed
2.
   - The CPU puts 1400H on address bus and sends it out
     - The memory circuitry finds the location
   - The CPU activates the READ signal, indicating to memory that it wants the byte at location 1400H
     - This causes the contents of memory location 1400H, which is B0, to be put on the data bus and brought into the CPU

Ex. (cont')

3.
   - The CPU decodes the instruction B0
   - The CPU commands its controller circuitry to bring into register A of the CPU the byte in the next memory location
     - The value 21H goes into register A
   - The program counter points to the address of the next instruction to be executed, which is 1402H
     - Address 1402 is sent out on the address bus to fetch the next instruction

Ex. (cont')

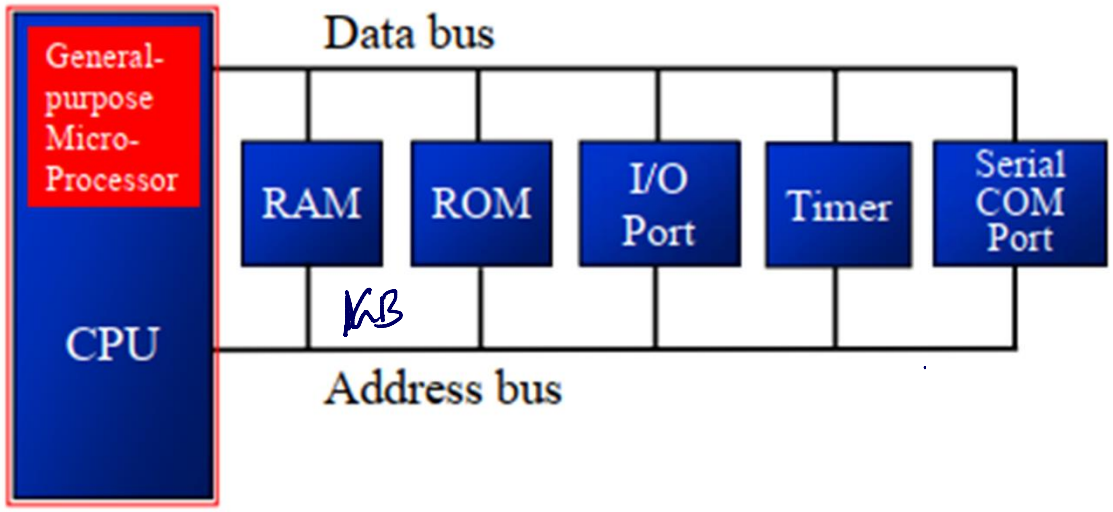
4.
   - From memory location 1402H it fetches code 04H
   - After decoding, the CPU knows that it must add to the contents of register A the byte sitting at the next address (1403)
   - After the CPU brings the value (42H), it provides the contents of register A along with this value to the ALU to perform the addition
     - It then takes the result of the addition from the ALU's output and puts it in register A
     - The program counter becomes 1404, the address of the next instruction

Ex. (cont')

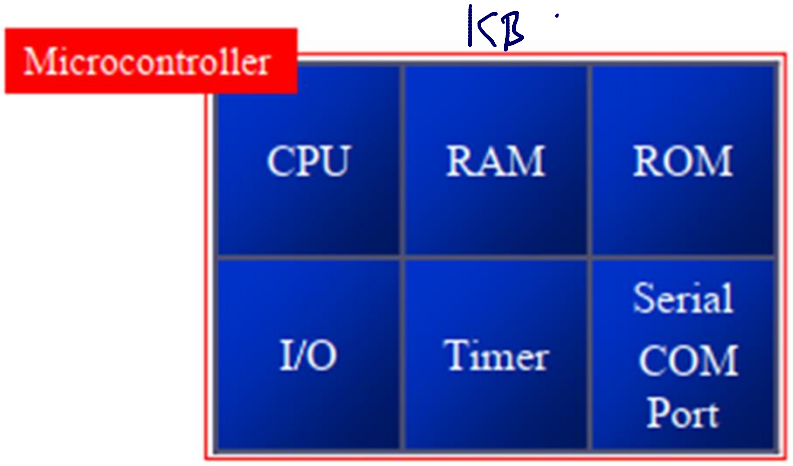
5.
   - Address 1404H is put on the address bus and the code is fetched into the CPU, decoded, and executed
     - This code is again adding a value to register A
     - The program counter is updated to 1406H
6.
   - The contents of address 1406 are fetched in and executed
   - This HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction

8051
MICROCONTROLLERS

- General-purpose microprocessors contains
  - No RAM
  - No ROM
  - No I/O ports
- Microcontroller has
  - CPU (microprocessor)
  - RAM
  - ROM
  - I/O ports
  - Timer
  - ADC and other peripherals


General-purpose Micro-Processor
CPU
Data bus
RAM
ROM
I/O Port
Timer
Serial COM Port
Address bus

μp


Microcontroller
CPU
RAM
ROM
I/O
Timer
Serial COM Port

μc

- General-purpose microprocessors
  - Must add RAM, ROM, I/O ports, and timers externally to make them functional
  - Make the system bulkier and much more expensive
  - Have the advantage of versatility on the amount of RAM, ROM, and I/O ports
- Microcontroller
  - The fixed amount of on-chip ROM, RAM, and number of I/O ports makes them ideal for many applications in which cost and space are critical
  - In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power

- An embedded product uses a microprocessor (or microcontroller) to do one task and one task only
  - There is only one application software that is typically burned into ROM
- A PC, in contrast with the embedded system, can be used for any number of applications
  - It has RAM memory and an operating system that loads a variety of applications into RAM and lets the CPU run them
  - A PC contains or is connected to various embedded products
    - Each one peripheral has a microcontroller inside it that performs only one task

Embedded Systems

Microcontrollers for embedded systems

In the literature discussing microprocessors, we often see the term *embedded system*. Microprocessors and microcontrollers are widely used in embedded system products. An embedded product uses a microprocessor (or microcontroller) to do one task and one task only. A printer is an example of embedded system since the processor inside it performs only one task ; namely, getting the data and printing it. Contrast this with a Pentium-based PC (or any x86 IBM-compatible PC). A PC can be used for any number of applications such as word processor, print server, bank teller terminal, video game player, network server, or internet terded system, there is only one application software that is typically burned into ROM. An x86 PC contains or is connected to various embedded products such as the keyboard, printer, modem, disk controller, sound card, CD-ROM driver, mouse, and so on. Each one of these peripherals has a microcontroller inside it that performs only one task. For example, inside every mouse there is a microcontroller that performs the task of finding the mouse position and sending it to the PC. Table 1-1 lists some embedded products.

Table 1.1 Some Embedded Products using Microcontroller

- **Home**
  - Appliances, intercom, telephones, security systems, garage door openers, answering machines, fax machines, home computers, TVs, cable TV tuner, VCR, camcorder, remote controls, video games, cellular phones, musical instruments, sewing machines, lighting control, paging, camera, pinball machines, toys, exercise equipment
- **Office**
  - Telephones, computers, security systems, fax machines, microwave, copier, laser printer, color printer, paging
- **Auto**
  - Trip computer, engine control, air bag, ABS, instrumentation, security system, transmission control, entertainment, climate control, cellular phone, keyless entry

- 8-bit microcontrollers
  - Motorola's 6811
  - Intel's 8051
  - Zilog's Z8
  - Microchip's PIC
- There are also 16-bit and 32-bit microcontrollers made by various chip makers

Table 1-3: Features of the 8051

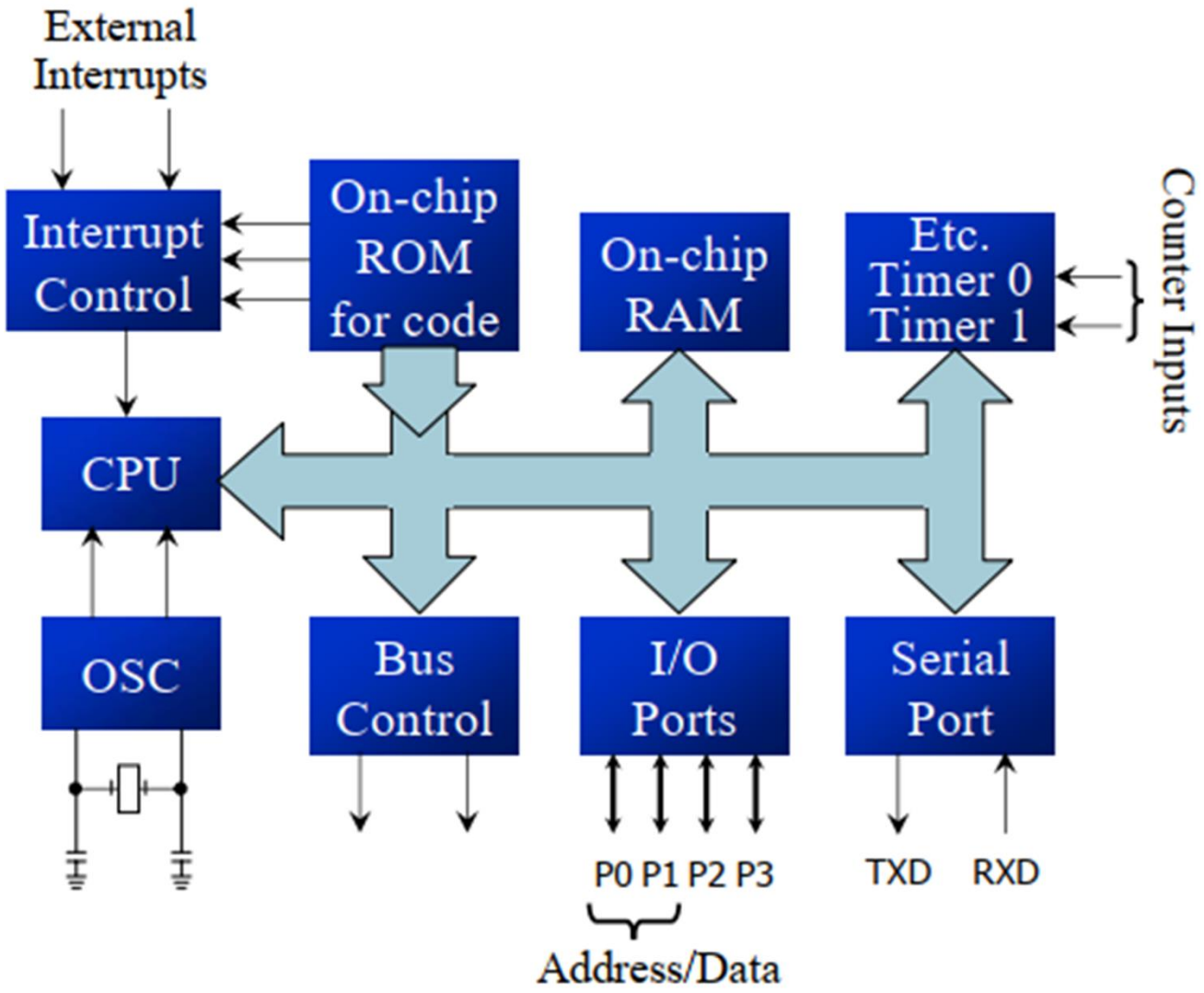
Feature	Quantity
ROM	4K bytes
RAM	128 bytes
Timer	2
I/O pins	32
Serial port	1
Interrupt sources	6

Note: ROM amount indicates on-chip program space.

- 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

8051 Internal Block Diagram

Important.



8051 Family

- 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

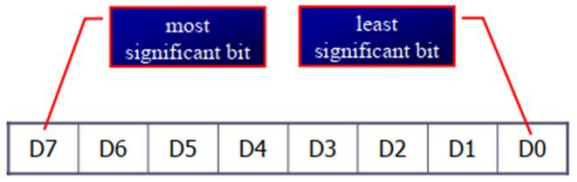
Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

"KEIL"
"8051"

8051 ASSEMBLY LANGUAGE PROGRAMMING

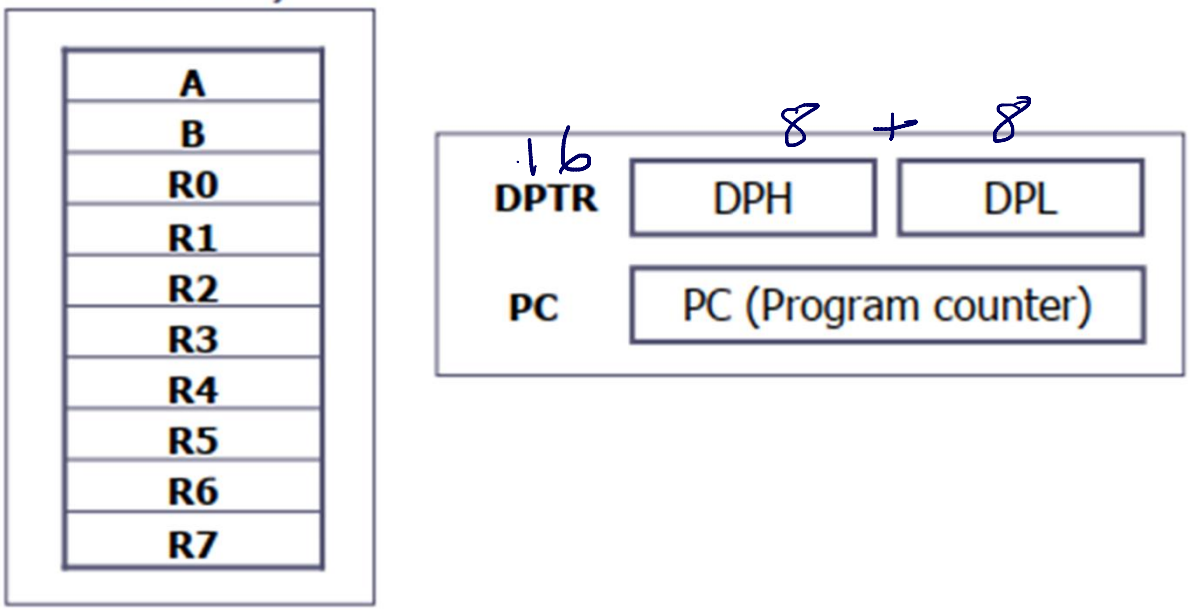
8051: Inside Registers

- 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

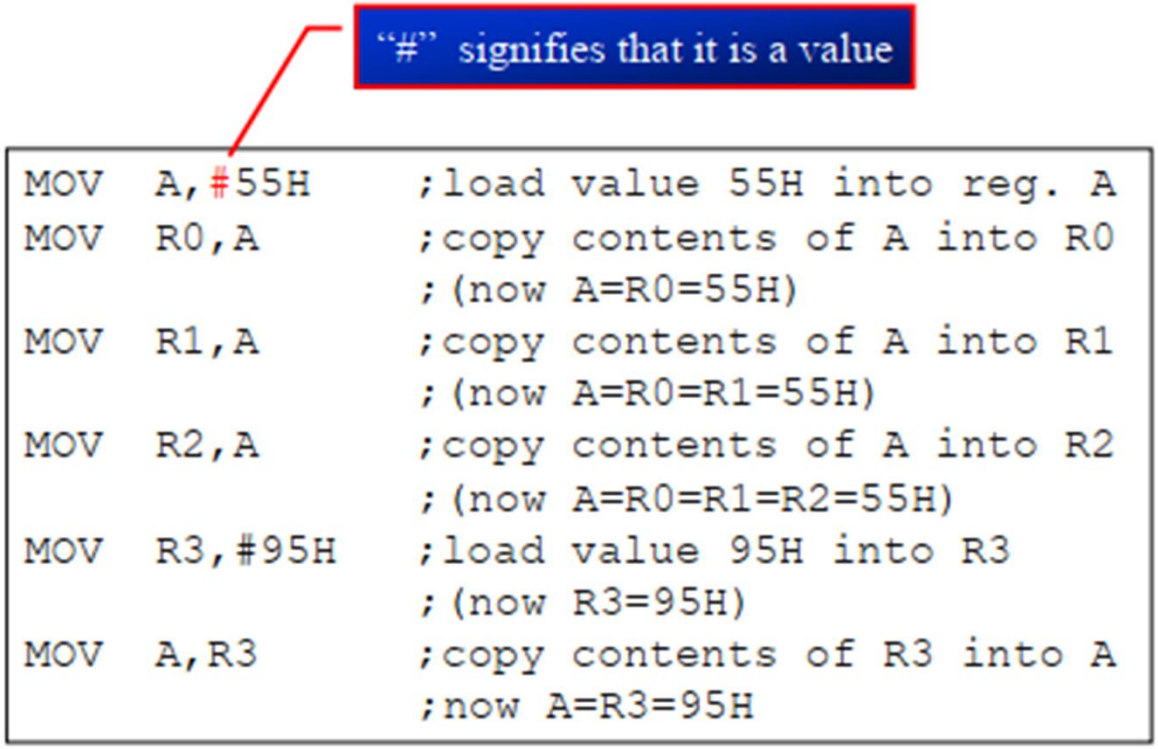
- The most widely used registers
  - A (Accumulator)
    - For all arithmetic and logic instructions
  - B, R0, R1, R2, R3, R4, R5, R6, R7
  - DPTR (data pointer), and PC (program counter)



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



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

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

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

  - 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

- In the early days of the computer, programmers coded in *machine language*, consisting of 0s and 1s
  - Tedious, slow and prone to error
- *Assembly languages*, which provided mnemonics for the machine code instructions, plus other features, were developed
  - An Assembly language program consist of a series of lines of Assembly language instructions
- Assembly language is referred to as a *low-level language*
  - It deals directly with the internal structure of the CPU

- 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

- An Assembly language instruction consists of four fields:

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

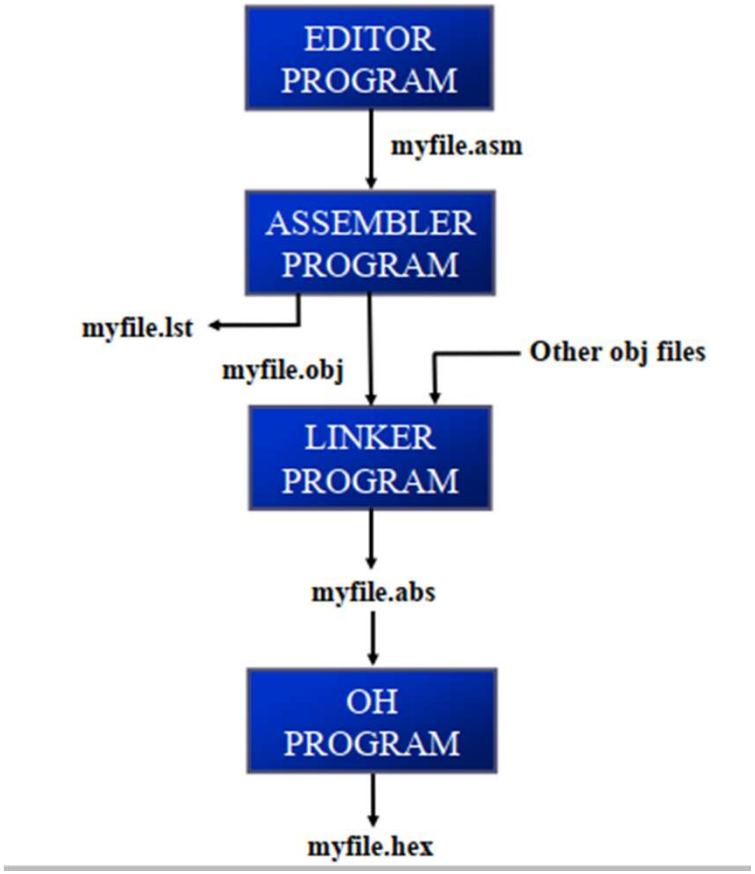
```
   ORG   0H              ;start(origin) at location
   0
   MOV   R5, #25H        ;load 25H into R5
   MOV   R7, #34H        ;load 34H in
   MOV   A, #0           ;load 0 into
   ADD   A, R5           ;add content
                         ;now A = A +
   ADD   A, R7           ;add contents of R7 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

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

Steps to create a program



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

- 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

- 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
```

address

- 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

- 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 valu 34H is moved into R7
     - The PC is incremented to 0004

- (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*

Assembler 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

- 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

CY	AC	F0	RS1	RS0	OV	--	P

A carry from D3 to D4

Carry out from the d7 bit

Reflect the number of 1s in register A

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.

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

FLAG Bits ADD Instruction

- The flag bits affected by the ADD instruction are CY, P, AC, and OV

Example 2-2

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)

FLAG Bits ADD Instruction

Example 2-3

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)

ADDRESSING MODES

- The CPU can access data in various ways, which are called *addressing modes*
  - Immediate
  - Register
  - Direct
  - Register indirect
  - Indexed

Direct, Register indirect, Indexed → Accessing memories

Immediate Addressing Mode

- The source operand is a constant
  - The immediate data must be preceded by the pound sign, "#"
  - Can load information into any registers, including 16-bit DPTR register
    - DPTR can also be accessed as two 8-bit registers, the high byte DPH and low byte DPL

```
MOV A,#25H       ;load 25H into A
MOV R4,#62       ;load 62 into R4
MOV B,#40H       ;load 40H into B
MOV DPTR,#4521H ;DPTR=4512H
MOV DPL,#21H     ;This is the same
MOV DPH,#45H     ;as above

;illegal!! Value > 65535 (FFFFH)
MOV DPTR,#68975
```

Register Addressing Mode

- Use registers to hold the data to be manipulated

```
MOV A,R0     ;copy contents of R0 into A
MOV R2,A     ;copy contents of A into R2
ADD A,R5     ;add contents of R5 to A
ADD A,R7     ;add contents of R7 to A
MOV R6,A     ;save accumulator in R6
```

- The source and destination registers must match in size
  - `MOV DPTR,A` will give an error

```
MOV DPTR,#25F5H
MOV R7,DPL
MOV R6,DPH
```

- The movement of data between Rn registers is not allowed
  - `MOV R4,R7` is invalid

Direct Addressing Mode

- It is most often used the direct addressing mode to access RAM locations 30 – 7FH
  - The entire 128 bytes of RAM can be accessed
  - The register bank locations are accessed by the register names

Direct addressing mode

```
MOV A,4     ;is same as
MOV A,R4    ;which means copy R4 into A
```

- Contrast this with immediate addressing mode
  - There is no "#" sign in the operand

Register addressing mode

```
MOV R0,40H  ;save content of 40H in R0
MOV 56H,A   ;save content of A in 56H
```

Register Indirect Addressing Mode

- A register is used as a pointer to the data
  - Only register R0 and R1 are used for this purpose
  - R2 – R7 cannot be used to hold the address of an operand located in RAM
- When R0 and R1 hold the addresses of RAM locations, they must be preceded by the "@" sign

```
MOV A,@R0   ;move contents of RAM whose
            ;address is held by R0 into A
MOV @R1,B   ;move contents of B into RAM
            ;whose address is held by R1
```

Indexed Addressing Mode

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM
- The instruction used for this purpose is
  `MOVC A,@A+DPTR`
  - Use instruction `MOVC`, "C" means code
  - The contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data

MOV A, #10H //moves the value 10H in the accumulator
MOV DPTR, #0180H //moves the value 0180H in DPTR
MOVC A, @A+DPTR //It adds the offset(10H) value in accumulator to the base (0180H) value in DPTR and then places the values located at address 0190H (0180H+10H) into the accumulator.

Embedded C

Why program the 8051 in C?

Compilers produce hex files that we download into the ROM of the microcontroller. The size of the hex file produced by the compiler is one of the main concerns of microcontroller programmers, for two reasons:

1. Microcontrollers have limited on-chip ROM.
2. The code space for the 8051 is limited to 64K bytes.

How does the choice of programming language affect the compiled program size? While Assembly language produces a hex file that is much smaller than C, programming in Assembly language is tedious and time consuming. C programming, on the other hand, is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language. The following are some of the major reasons for writing programs in C instead of Assembly:

1. It is easier and less time consuming to write in C than Assembly.
2. C is easier to modify and update.
3. You can use code available in function libraries.
4. C code is portable to other microcontrollers with little or no modification.

- Compilers produce hex files that is downloaded to ROM of microcontroller
  - The size of hex file is the main concern
    - Microcontrollers have limited on-chip ROM
    - Code space for 8051 is limited to 64K bytes
- C programming is less time consuming, but has larger hex file size
- The reasons for writing programs in C
  - It is easier and less time consuming to write in C than Assembly
  - C is easier to modify and update
  - You can use code available in function libraries
  - C code is portable to other microcontroller with little of no modification

- A good understanding of C data types for 8051 can help programmers to create smaller hex files
  - Unsigned char
  - Signed char
  - Unsigned int
  - Signed int

- The character data type is the most natural choice
  - 8051 is an 8-bit microcontroller
- Unsigned char is an 8-bit data type in the range of 0 – 255 (00 – FFH)
  - One of the most widely used data types for the 8051
    - Counter value
    - ASCII characters
- C compilers use the signed char as the default if we do not put the keyword *unsigned*

Write an 8051 C program to send values 00 – FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char z;
    for (z=0;z<=255;z++)
       P1=z;
  }
```

1. Pay careful attention to the size of the data
2. Try to use unsigned *char* instead of *int* if possible

Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mynum[]="012345ABCD";
    unsigned char z;
    for (z=0;z<=10;z++)
       P1=mynum[z];
  }
```

- The signed char is an 8-bit data type
  - Use the MSB D7 to represent – or +
  - Give us values from –128 to +127
- We should stick with the unsigned char unless the data needs to be represented as signed numbers
  - temperature

- The unsigned int is a 16-bit data type
  - Takes a value in the range of 0 to 65535 (0000 – FFFFH)
  - Define 16-bit variables such as memory addresses
  - Set counter values of more than 256
  - Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file
- Signed int is a 16-bit data type
  - Use the MSB D15 to represent – or +
  - We have 15 bits for the magnitude of the number from –32768 to +32767

Addition Operation Using Embedded C

```
#include <reg51.h>
void main(void)
{
unsigned char x,y,z;
x=0x34;
y=0x04;
P1=0x00;
        z=x+y;

        P1=z;

}
```

Sample Questions

1) Draw the Internal block diagram of 8051
2) Difference between microprocessor and microcontroller
3) What are different addressing modes in 8051
4) Write assembly language program to add/mul/sub/division of two values 51H and 45H stored in the registers R1 and R2, respectively. Store the result in R3.
5) What is an Embedded system? Write a program to add the values in Embedded C language.