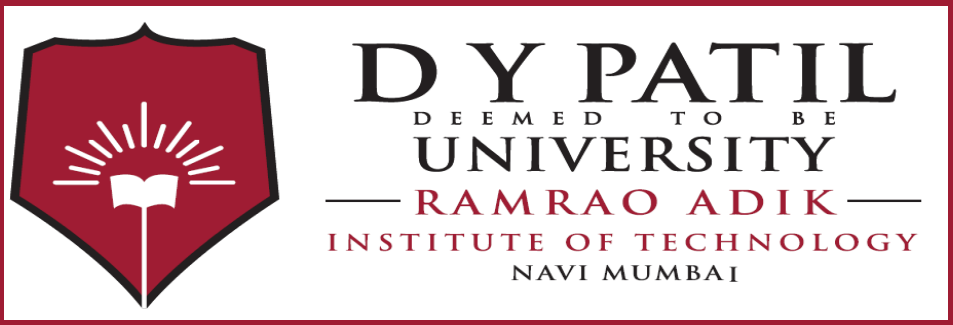


Embedded System
(Multidisciplinary Minor)
Module 3: MCS-51 Microcontroller

Index

8051 Microcontroller

Intel introduced 8051, referred as MCS- 51, in 1981

The 8051 is an 8-bit processor (system on chip)

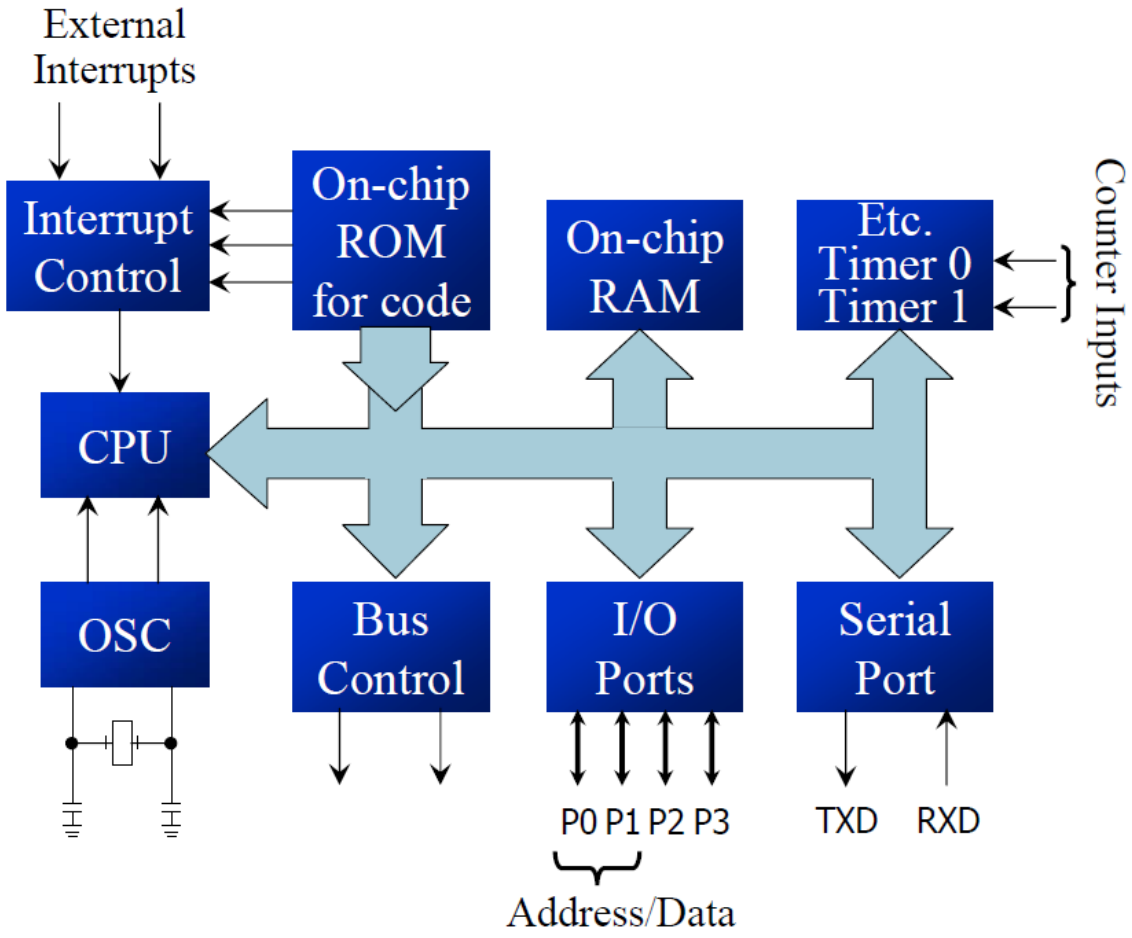
- The CPU can work on only 8 bits of data at a time

The 8051 had

- 128 bytes of RAM
- 4K bytes of on-chip ROM
- Two timers/Counters
- One serial port
- Four I/O ports, each 8 bits wide (32 Input/Output Pins)
- 6 interrupt sources (INT0, TFO, INT1, TF1, RI/TI)
- only 1 On chip oscillator (external crystal)(11.0592 MHz or 12 MHz. Each machine cycle in the 8051 is composed of 12 clock cycles)

The 8051 became widely popular after allowing other manufactures to make and market any flavor of the 8051, but remaining code-compatible.

8051 Microcontroller



Inside 8051 Microcontroller

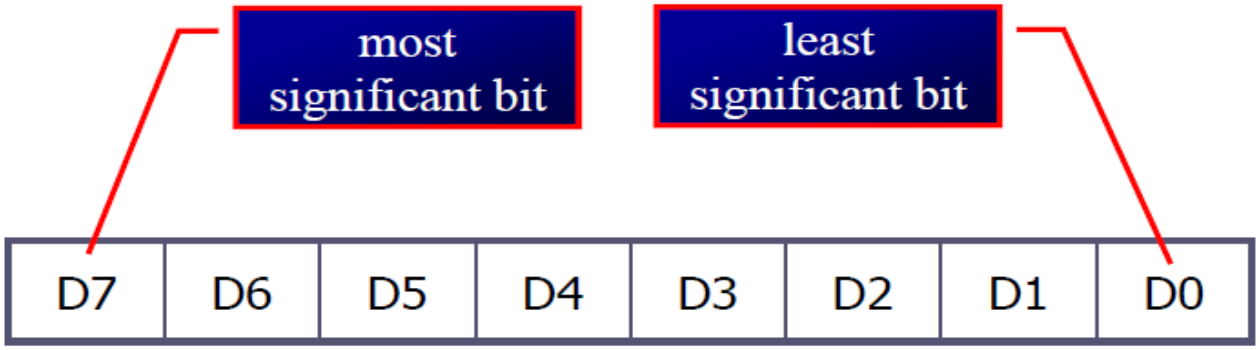
- Register are used to store information temporarily, while the information could be
  - a byte of data to be processed, or
  - an address pointing to the data to be fetched
- The vast majority of 8051 register are 8-bit registers
  - There is only one data type, 8 bits

Program Counter(16 Bit Register)

- 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

Registers

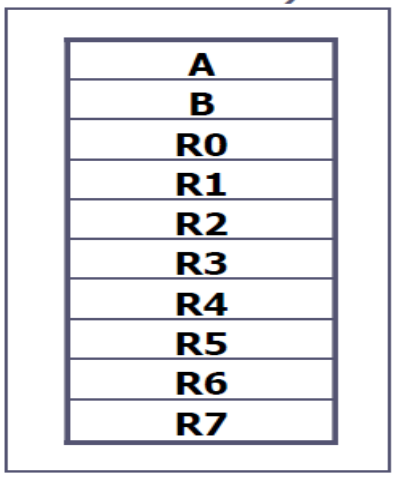
- The 8 bits of a register are shown from MSB D7 to the LSB D0
  - With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed



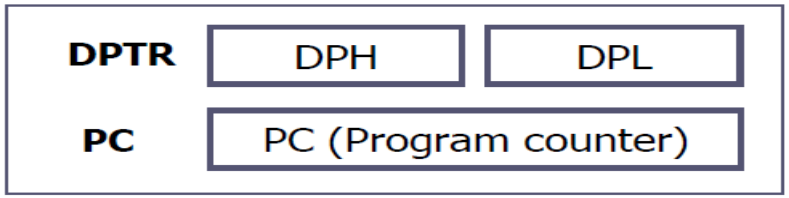
8 bit Registers

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)



8 Bit Registers



16 Bit Registers

Assembly Programming: 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
```

D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

- Notes on programming
  - 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 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

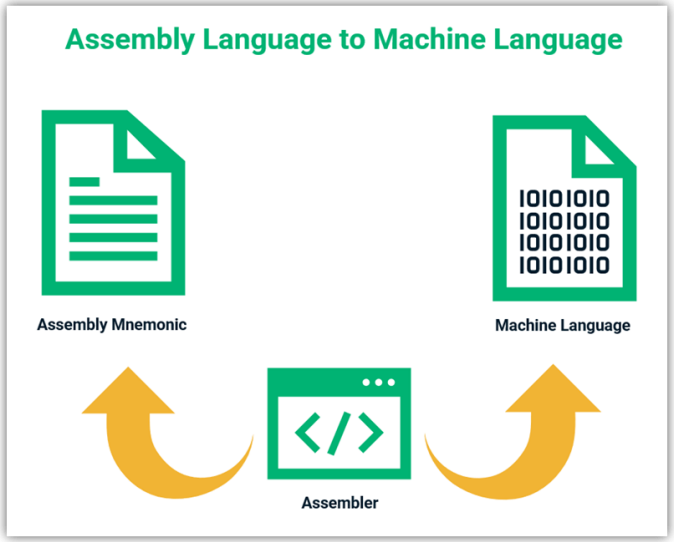
```
MOV A, #25H       ;load 25H into A
MOV R2, #34H      ;load 34H into R2
ADD A, R2 ;add R2 to Accumulator
                  ;(A = A + R2)
```

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

There are always many ways to write the same program, depending on the registers used

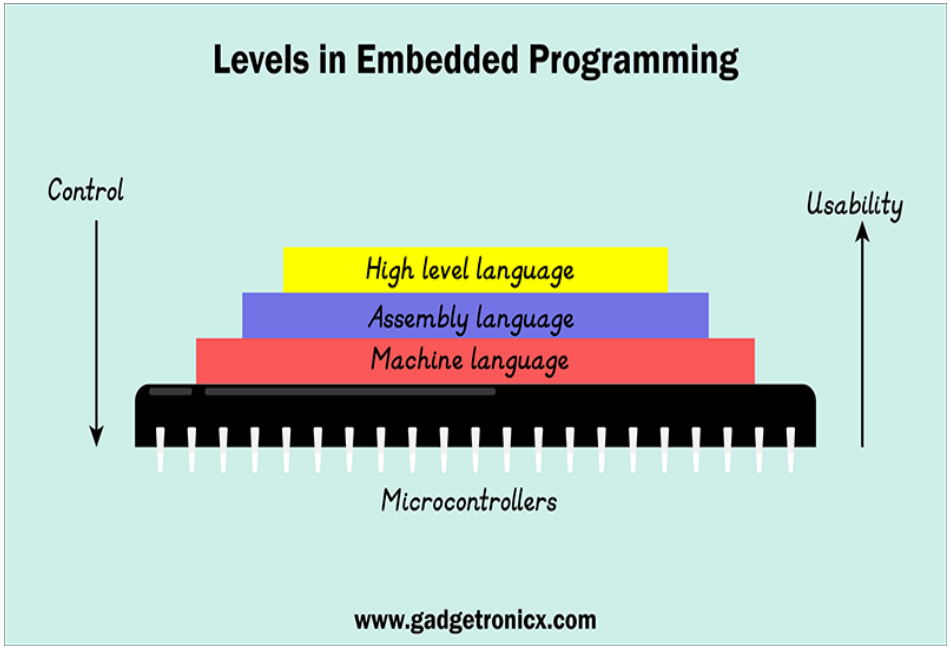
Assembly Language Programming

- 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 Programming

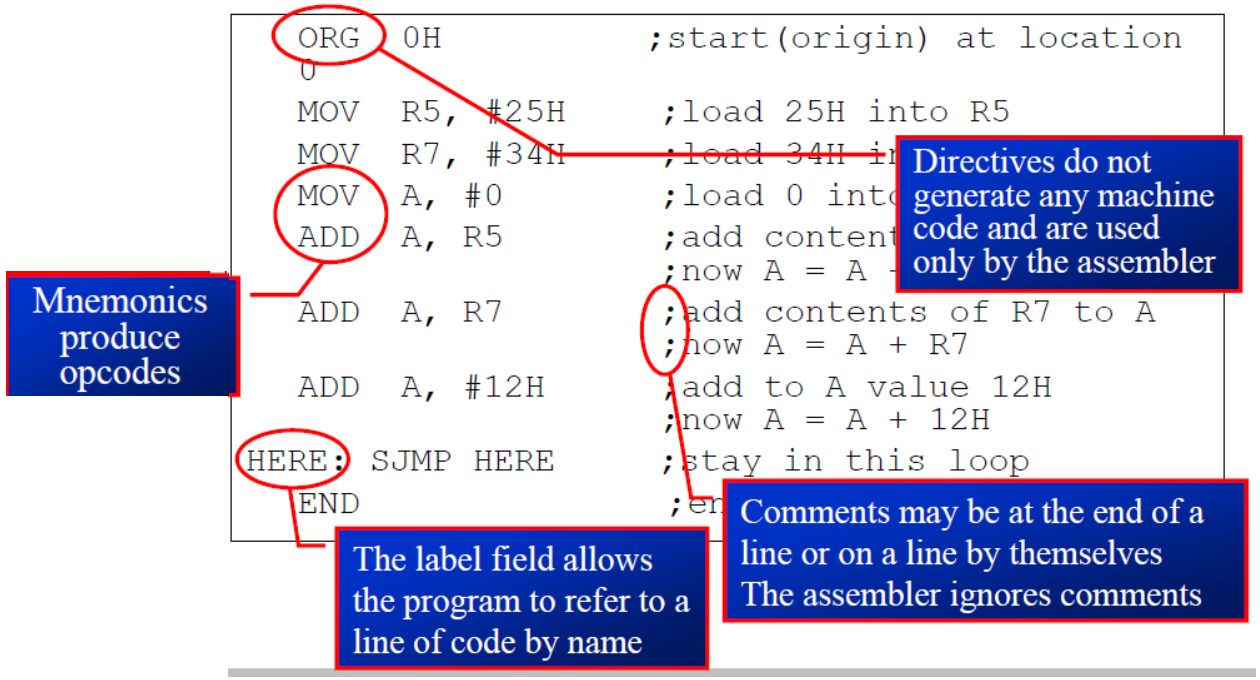
- Assembly language instruction includes
  - a mnemonic (abbreviation easy to remember)
    - the commands to the CPU, telling it what those to do with those items
  - optionally followed by one or two operands
    - the data items being manipulated
- A given Assembly language program is a series of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



Structure of Assembly Language

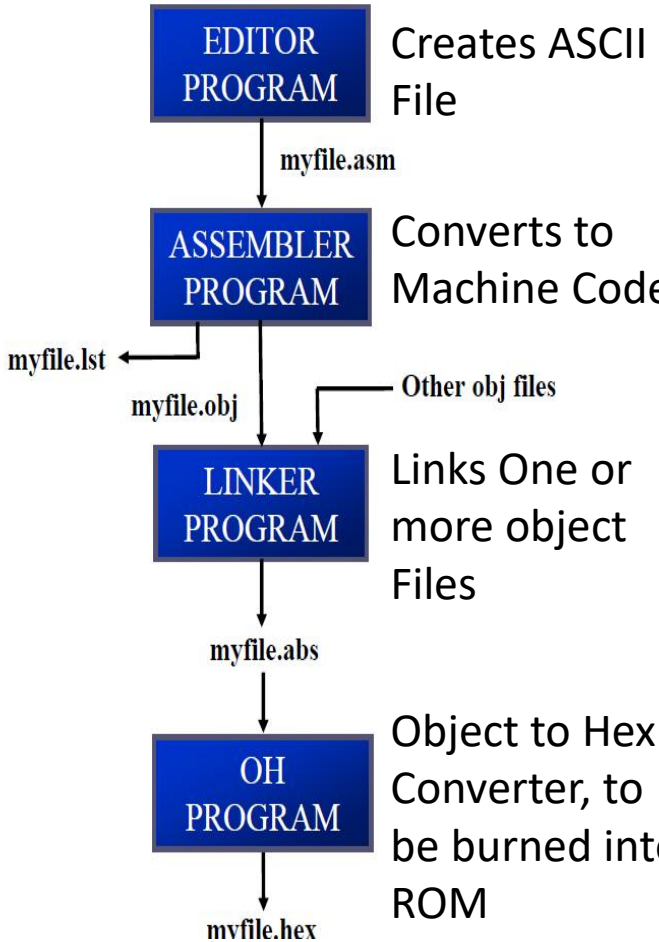
- An Assembly language instruction consists of four fields:

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



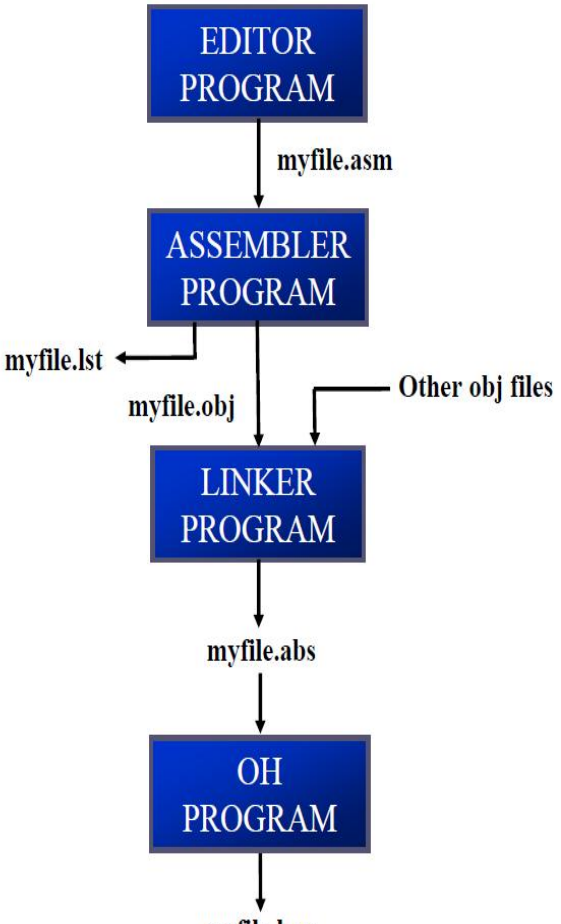
Assembly Language Program Outlines

- 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



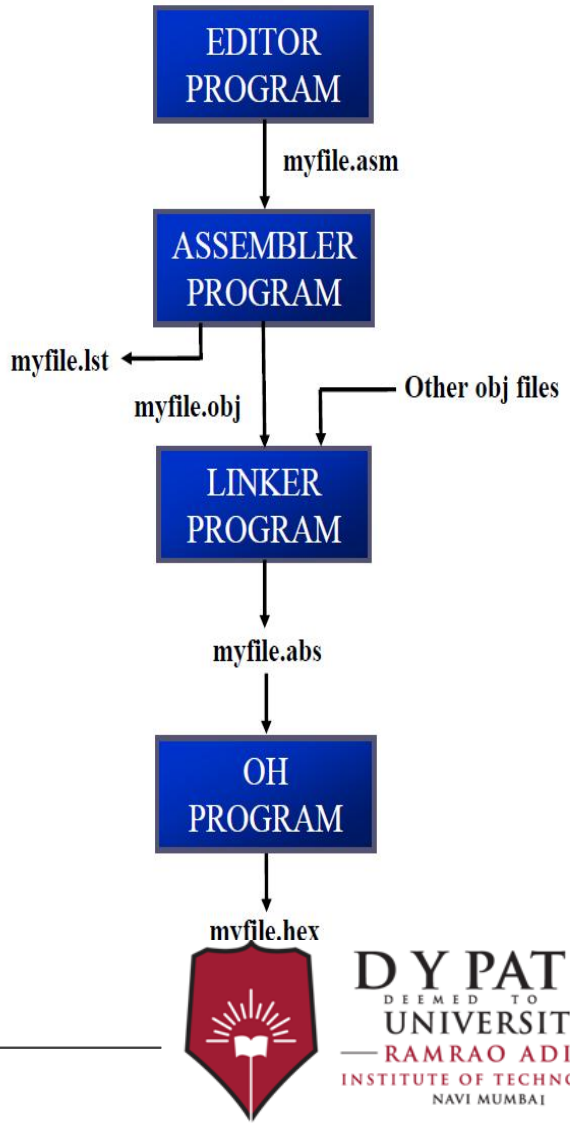
Assembly Language Program Outlines

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



Assembly Language Program Outlines

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



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

Placing Codes in 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

Placing Codes in ROM

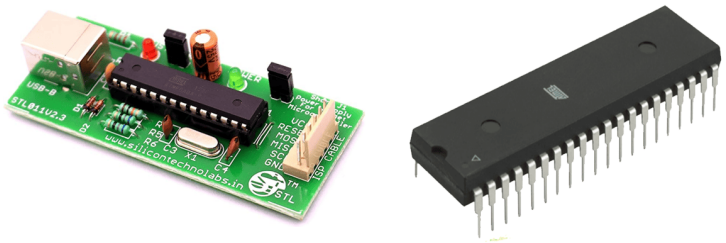
- After the program is burned into ROM, the opcode and operand are placed in ROM memory location starting at 0000

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

Campus

8051 Microcontroller



Getting Started with 8051 Microcontroller

Executing Program

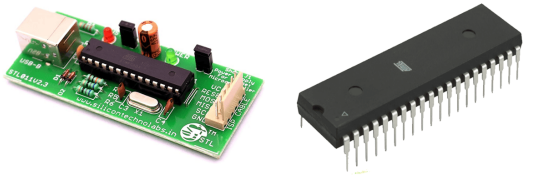
- A step-by-step description of the action of the 8051 upon applying power on it
  1. When 8051 is powered up, the PC has 0000 and starts to fetch the first opcode from location 0000 of program ROM
     - Upon executing the opcode 7D, the CPU fetches the value 25 and places it in R5
     - Now one instruction 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

Executing Program

- (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*

Campus

8051 Microcontroller



Getting Started with 8051 Microcontroller

- 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

Data Directive (DB: Define Byte (Use to Define 8 Bit Data))

- 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

Assembler Directives

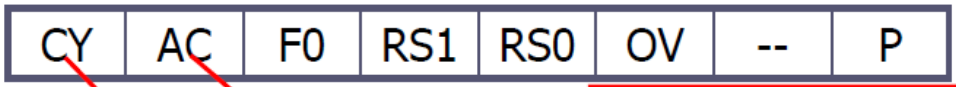
- ORG (origin)
  - The ORG directive is used to indicate the beginning of the address
  - The number that comes after ORG can be either in hex and decimal
    - If the number is not followed by H, it is decimal and the assembler will convert it to hex
- END
  - This indicates to the assembler the end 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

PSW Register

- 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

If the result has an **odd number of 1's**, then **P = 1**. If the result has an **even number of 1's**, then **P = 0**.

PSW Register



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

ADD Instruction and PSW

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		

Flag Bits and PSW Register

- 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)

Example

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)

D Y PATIL DEEMED TO BE UNIVERSITY
RAMRAO ADIK INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Example

Example 2-4

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:

	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)

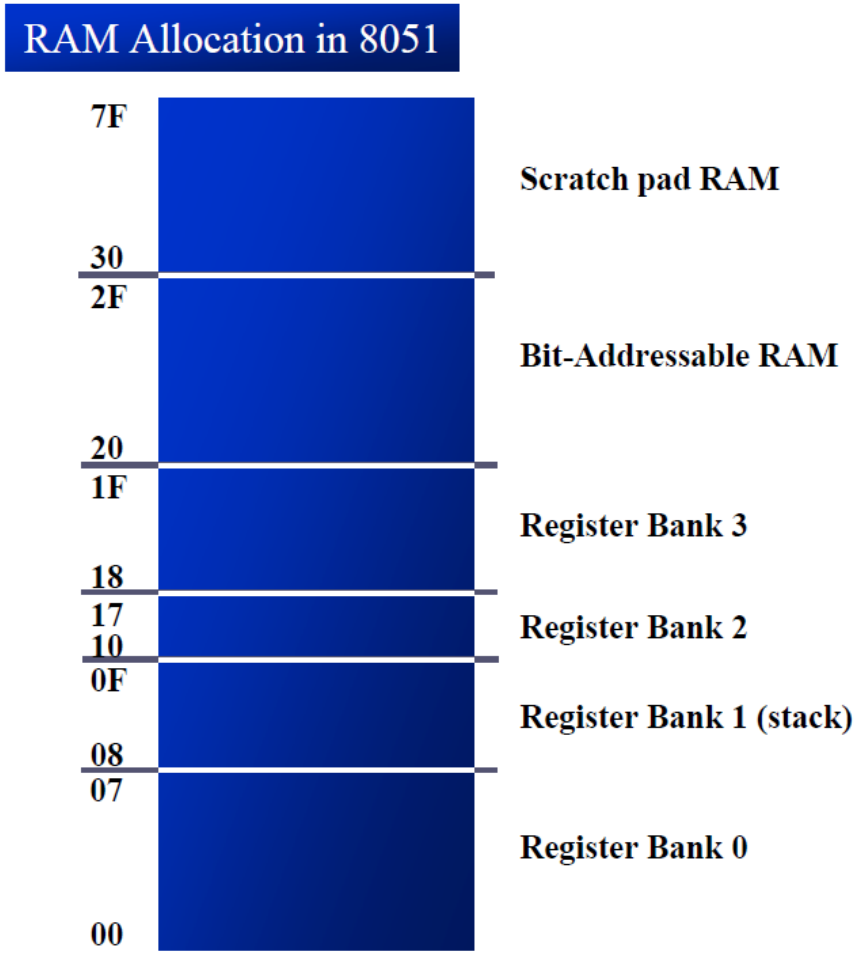
D Y PATIL DEEMED TO BE UNIVERSITY RAMRAO ADIK INSTITUTE OF TECHNOLOGY NAVI MUMBAI

RAM Memory Space Allocation

- There are 128 bytes of RAM in the 8051
  - Assigned addresses 00 to 7FH
- The 128 bytes are divided into three different groups as follows:
  1) A total of 32 bytes from locations 00 to 1F hex are set aside for register banks and the stack
  2) A total of 16 bytes from locations 20H to 2FH are set aside for bit-addressable read/write memory
  3) A total of 80 bytes from locations 30H to 7FH are used for read and write storage, called *scratch pad*

O TO 127:
Total 128

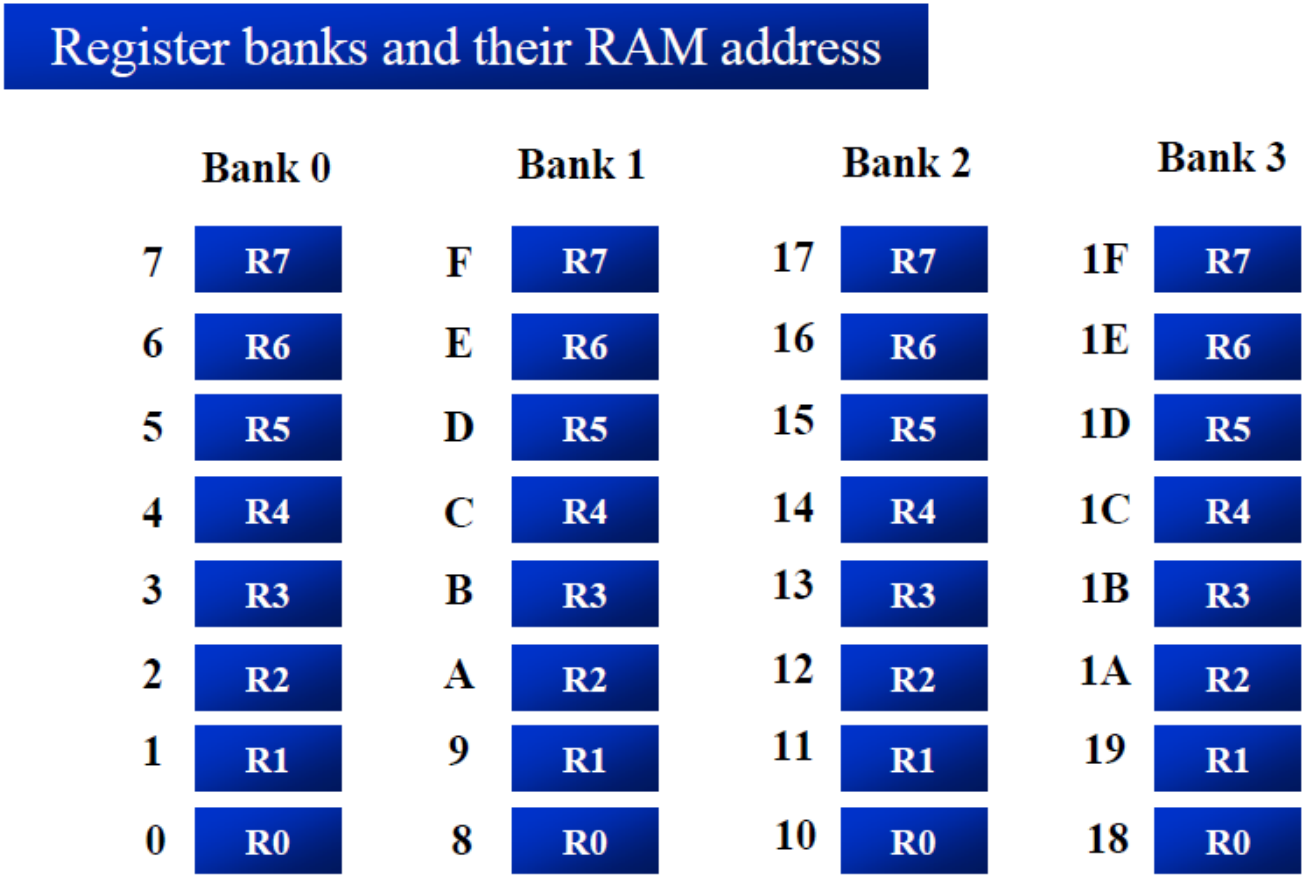
RAM Allocation



RAM Space Allocation

- These 32 bytes are divided into 4 banks of registers in which each bank has 8 registers, R0-R7
  - RAM location from 0 to 7 are set aside for bank 0 of R0-R7 where R0 is RAM location 0, R1 is RAM location 1, R2 is RAM location 2, and so on, until memory location 7 which belongs to R7 of bank 0
  - It is much easier to refer to these RAM locations with names such as R0, R1, and so on, than by their memory locations
- Register bank 0 is the default when 8051 is powered up

Register Bank and their RAM address



Selecting Register Baks

- We can switch to other banks by use of the PSW register
  - Bits D4 and D3 of the PSW are used to select the desired register bank
  - Use the bit-addressable instructions SETB and CLR to access PSW.4 and PSW.3

PSW bank selection

	RS1(PSW.4)	RS0(PSW.3)
Bank 0	0	0
Bank 1	0	1
Bank 2	1	0
Bank 3	1	1

Example

Example 2-5

```
MOV R0, #99H      ;load R0 with 99H
MOV R1, #85H      ;load R1 with 85H
```

Example 2-6

```
MOV 00, #99H      ;RAM location 00H has 99H
MOV 01, #85H      ;RAM location 01H has 85H
```

Example 2-7

```
SETB PSW.4        ;select bank 2
MOV R0, #99H      ;RAM location 10H has 99H
MOV R1, #85H      ;RAM location 11H has 85H
```

Stack

- The stack is a section of RAM used by the CPU to store information temporarily
  - This information could be data or an address
- The register used to access the stack is called the SP (stack pointer) register
  - The stack pointer in the 8051 is only 8 bit wide, which means that it can take value of 00 to FFH
  - When the 8051 is powered up, the SP register contains value 07
    - RAM location 08 is the first location begin used for the stack by the 8051

Stack

- The storing of a CPU register in the stack is called a `PUSH`
  - SP is pointing to the last used location of the stack
  - As we push data onto the stack, the SP is incremented by one
    - This is different from many microprocessors
- Loading the contents of the stack back into a CPU register is called a `POP`
  - With every pop, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented once

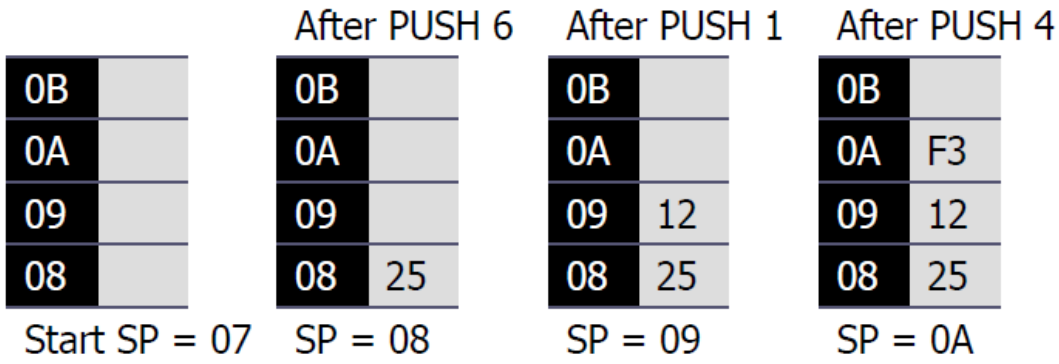
Example

Example 2-8

Show the stack and stack pointer from the following. Assume the default stack area.

```
MOV R6, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH  6
PUSH  1
PUSH  4
```

Solution:



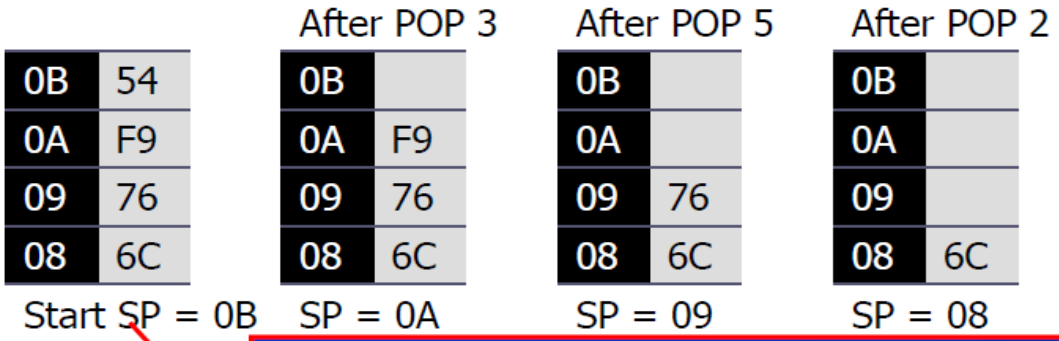
Example

Example 2-9

Examining the stack, show the contents of the register and SP after execution of the following instructions. All value are in hex.

```
POP     3          ; POP stack into R3
POP     5          ; POP stack into R5
POP     2          ; POP stack into R2
```

Solution:



Because locations 20-2FH of RAM are reserved for bit-addressable memory, so we can change the SP to other RAM location by using the instruction "MOV SP, #XX"

- The CPU also uses the stack to save the address of the instruction just below the `CALL` instruction
  - This is how the CPU knows where to resume when it returns from the called subroutine

Stack

- The reason of incrementing SP after push is
  - Make sure that the stack is growing toward RAM location 7FH, from lower to upper addresses
  - Ensure that the stack will not reach the bottom of RAM and consequently run out of stack space
  - If the stack pointer were decremented after push
    - We would be using RAM locations 7, 6, 5, etc. which belong to R7 to R0 of bank 0, the default register bank

Stack

- When 8051 is powered up, register bank 1 and the stack are using the same memory space
  - We can reallocate another section of RAM to the stack

Loop and Jump Instructions

LOOP AND JUMP INSTRUCTIONS

Looping

A loop can be repeated a maximum of 255 times, if R2 is FFH

- Repeating a sequence of instructions a certain number of times is called a *loop*
  - Loop action is performed by

    `DJNZ reg, Label`
    - The register is decremented
    - If it is not zero, it jumps to the target address referred to by the label
    - Prior to the start of loop the register is loaded with the counter for the number of repetitions
    - Counter can be R0 – R7 or RAM location

```
;This program adds value 3 to the ACC ten times
       MOV  A,#0      ;A=0, clear ACC
       MOV  R2,#10    ;load counter R2=10
AGAIN: ADD  A,#03     ;add 03 to ACC
       DJNZ R2,AGAIN  ;repeat until R2=0,10 times
       MOV  R5,A      ;save A in R5
```

D Y PATIL DEEMED TO BE UNIVERSITY — RAMRAO ADIK — INSTITUTE OF TECHNOLOGY NAVI MUMBAI

Nested Loops

LOOP AND JUMP INSTRUCTIONS

Nested Loop

- If we want to repeat an action more times than 256, we use a loop inside a loop, which is called *nested loop*
  - We use multiple registers to hold the count

Write a program to (a) load the accumulator with the value 55H, and (b) complement the ACC 700 times

```
        MOV  A,#55H   ;A=55H
        MOV  R3,#10   ;R3=10, outer loop count
NEXT:   MOV  R2,#70   ;R2=70, inner loop count
AGAIN:  CPL  A        ;complement A register
        DJNZ R2,AGAIN ;repeat it 70 times
        DJNZ R3,NEXT
```

Jump Instructions

- Jump only if a certain condition is met

JZ label ;jump if A=0

```
        MOV   A,R0      ;A=R0
        JZ    OVER      ;jump if A = 0
        MOV   A,R1      ;A=R1
        JZ    OVER      ;jump if A = 0
        ...
OVER:
```

Can be used only for register A, not any other register

Determine if R5 contains the value 0. If so, put 55H in it.

```
        MOV   A,R5      ;copy R5 to A
        JNZ   NEXT      ;jump if A is not zero
        MOV   R5,#55H
NEXT:   ...
```

Conditional Jumps

- (cont')

```
JNC label    ;jump if no carry, CY=0
```

  - If CY = 0, the CPU starts to fetch and execute instruction from the address of the label
  - If CY = 1, it will not jump but will execute the nex instruction below JNC

Conditional Jumps

8051 conditional jump instructions

Instructions	Actions
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and Jump if A ≠ 0
CJNE A,byte	Jump if A ≠ byte
CJNE reg,#data	Jump if byte ≠ #data
JC	Jump if CY = 1
JNC	Jump if CY = 0
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

- All conditional jumps are short jumps
  - The address of the target must within -128 to +127 bytes of the contents of PC

Unconditional Jumps

- The unconditional jump is a jump in which control is transferred unconditionally to the target location

LJMP (long jump)

- 3-byte instruction
  - First byte is the opcode
  - Second and third bytes represent the 16-bit target address
    - Any memory location from 0000 to FFFFH

SJMP (short jump)

- 2-byte instruction
  - First byte is the opcode
  - Second byte is the relative target address
    - 00 to FFH (forward +127 and backward -128 bytes from the current PC)

Jump Instructions

- To calculate the target address of a short jump (`SJMP`, `JNC`, `JZ`, `DJNZ`, etc.)
  - The second byte is added to the PC of the instruction immediately below the jump
- If the target address is more than -128 to +127 bytes from the address below the short jump instruction
  - The assembler will generate an error stating the jump is out of range

Call Instructions

- Call instruction is used to call subroutine
  - Subroutines are often used to perform tasks that need to be performed frequently
  - This makes a program more structured in addition to saving memory space

LCALL (long call)

- 3-byte instruction
  - First byte is the opcode
  - Second and third bytes are used for address of target subroutine
    - Subroutine is located anywhere within 64K byte address space

ACALL (absolute call)

- 2-byte instruction
  - 11 bits are used for address within 2K-byte range

Call Instructions

- When a subroutine is called, control is transferred to that subroutine, the processor
  - Saves on the stack the the address of the instruction immediately below the LCALL
  - Begins to fetch instructions form the new location
- After finishing execution of the subroutine
  - The instruction RET transfers control back to the caller
    - Every subroutine needs RET as the last instruction

Call Instructions

CALL INSTRUCTIONS

LCALL (cont')

```
        ORG    0
BACK:   MOV    A,#55H     ;load A with 55H
        MOV    P1,A       ;send 55H to port 1
        LCALL  DELAY      ;time delay
        MOV    A,#0AAH    ;load A with AA (in hex)
        MOV    P1,A       ;send AAH to port 1
        LCALL  DELAY
        SJMP   BACK       ;keep doing this indefinitely

;---------- this is delay subroutine ------------
        ORG    300H       ;put DELAY at address 300H
DELAY:  MOV    R5,#0FFH   ;R5=255 (FF in hex), counter
AGAIN:  DJNZ   R5,AGAIN   ;stay here until R5 become 0
        RET               ;return to caller (when R5 =0)
        END
```

Upon executing "LCALL DELAY", the address of instruction below it, "MOV A,#0AAH" is pushed onto stack, and the 8051 starts to execute at 300H.

The counter R5 is set to FFH; so loop is repeated 255 times.

When R5 becomes 0, control falls to the RET which pops the address from the stack into the PC and resumes executing the instructions after the CALL.

The amount of time delay depends on the frequency of the 8051

Call Instructions (Difference between LCALL and ACALL)

- The only difference between ACALL and LCALL is
  - The target address for LCALL can be anywhere within the 64K byte address
  - The target address of ACALL must be within a 2K-byte range
- The use of ACALL instead of LCALL can save a number of bytes of program ROM space

LCALL and ACALL Instructions

```
        ORG    0
BACK:   MOV    A,#55H     ;load A with 55H
        MOV    P1,A       ;send 55H to port 1
        LCALL  DELAY      ;time delay
        MOV    A,#0AAH    ;load A with AA (in hex)
        MOV    P1,A       ;send AAH to port 1
        LCALL  DELAY
        SJMP   BACK       ;keep doing this indefinitely
         ...
        END               ;end of asm file
```

A rewritten program which is more efficiently

```
        ORG    0
        MOV    A,#55H     ;load A with 55H
BACK:   MOV    P1,A       ;send 55H to port 1
        ACALL  DELAY      ;time delay
        CPL    A          ;complement reg A
        SJMP   BACK       ;keep doing this indefinitely
         ...
        END               ;end of asm file
```

Thank You