8051 ASSEMBLY LANGUAGE PROGRAMMING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

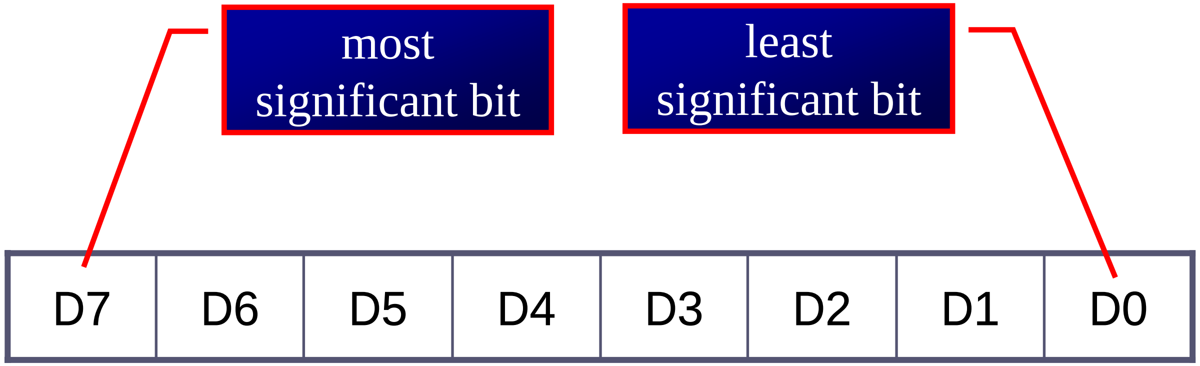
Registers

- 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

INSIDE THE 8051

Registers (cont')

- 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



most significant bit

least significant bit

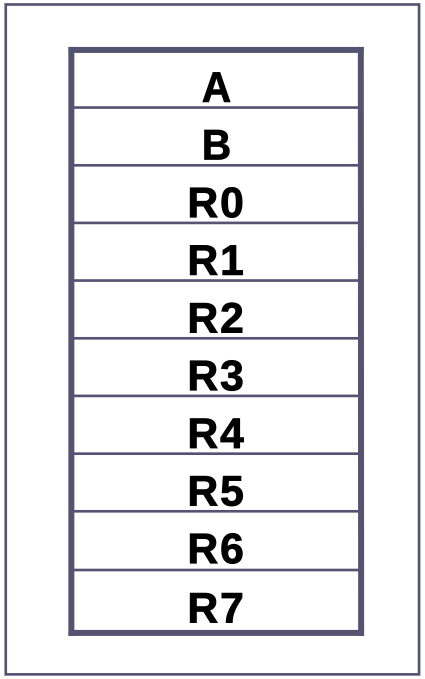
D7	D6	D5	D4	D3	D2	D1	D0

8 bit Registers

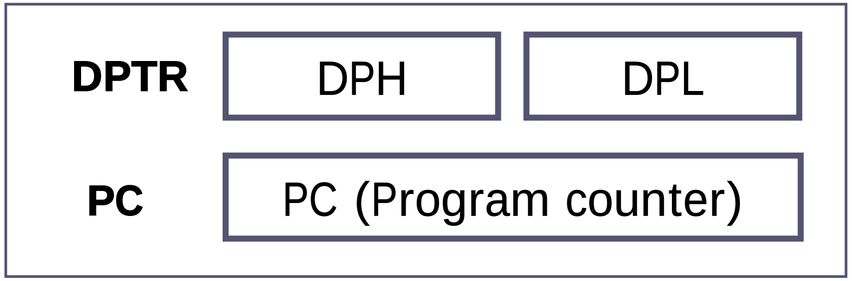
INSIDE THE 8051

Registers (cont')

- 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)



A
B
R0
R1
R2
R3
R4
R5
R6
R7



DPTR DPH DPL
PC PC (Program counter)

INSIDE THE 8051

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
```

INSIDE THE 8051

MOV Instruction (cont')

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)`

INSIDE THE 8051

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)
```

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

```
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

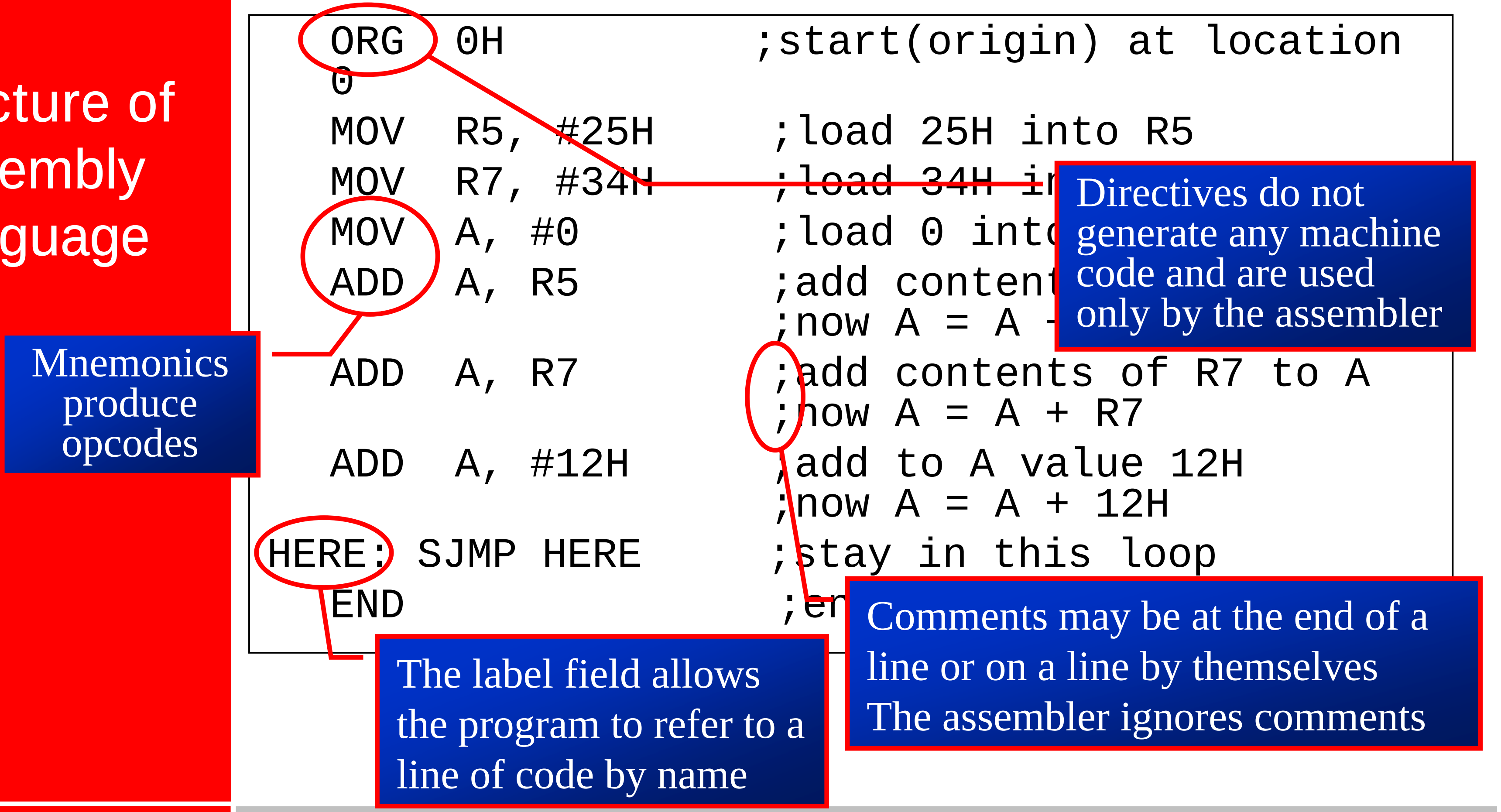
Structure of Assembly Language

- 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

Structure of 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 ir
   MOV  A, #0          ;load 0 intc
   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

ASSEMBLING AND RUNNING AN 8051 PROGRAM

- 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

ASSEMBLING AND RUNNING AN 8051 PROGRAM (cont')

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"

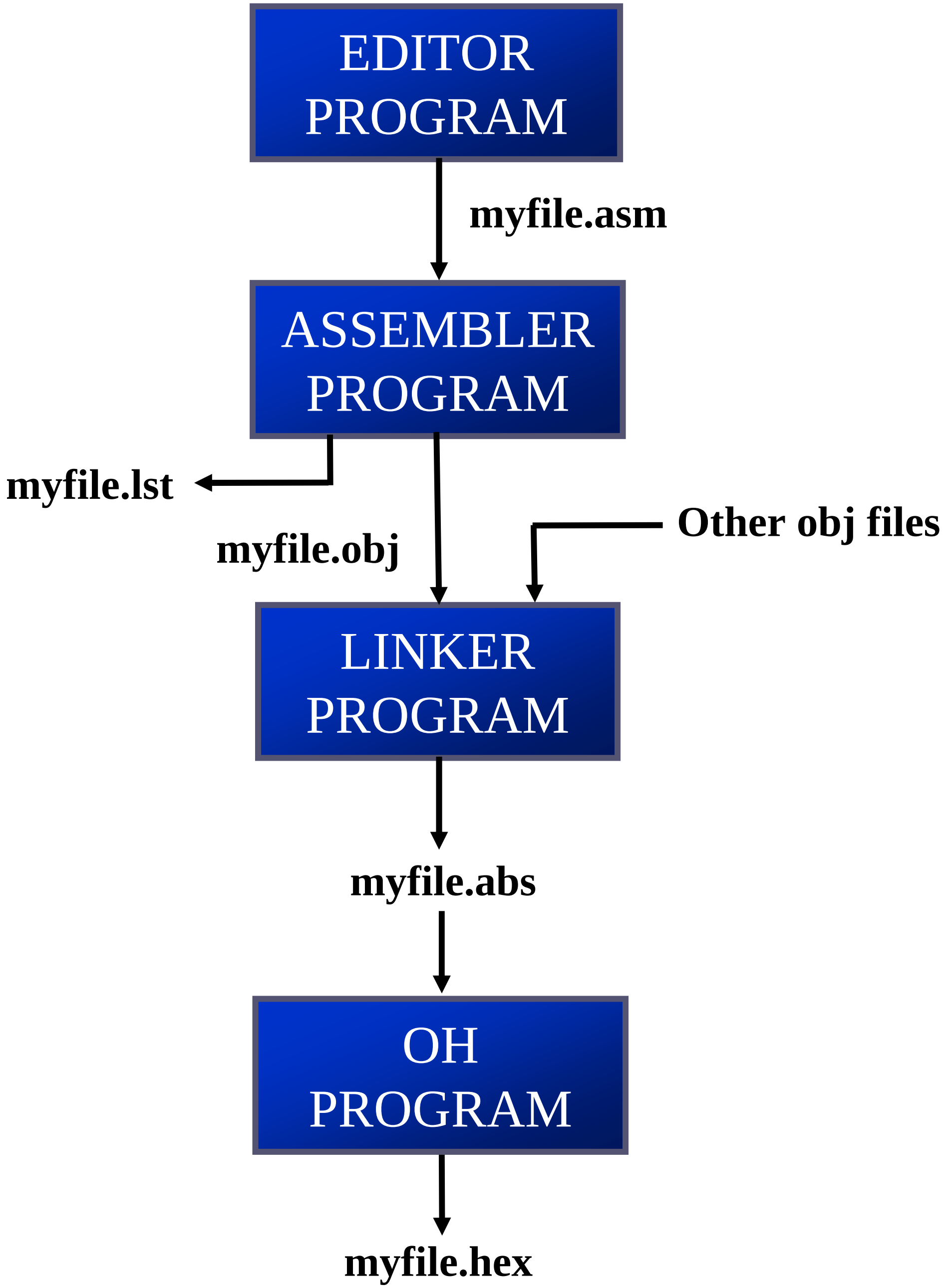
3) Assembler require a third step called *linking*
   - The linker program takes one or more object code files and produce an absolute object file with the extension "abs"
   - This abs file is used by 8051 trainers that have a monitor program

ASSEMBLING AND RUNNING AN 8051 PROGRAM (cont')

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

ASSEMBLING AND RUNNING AN 8051 PROGRAM

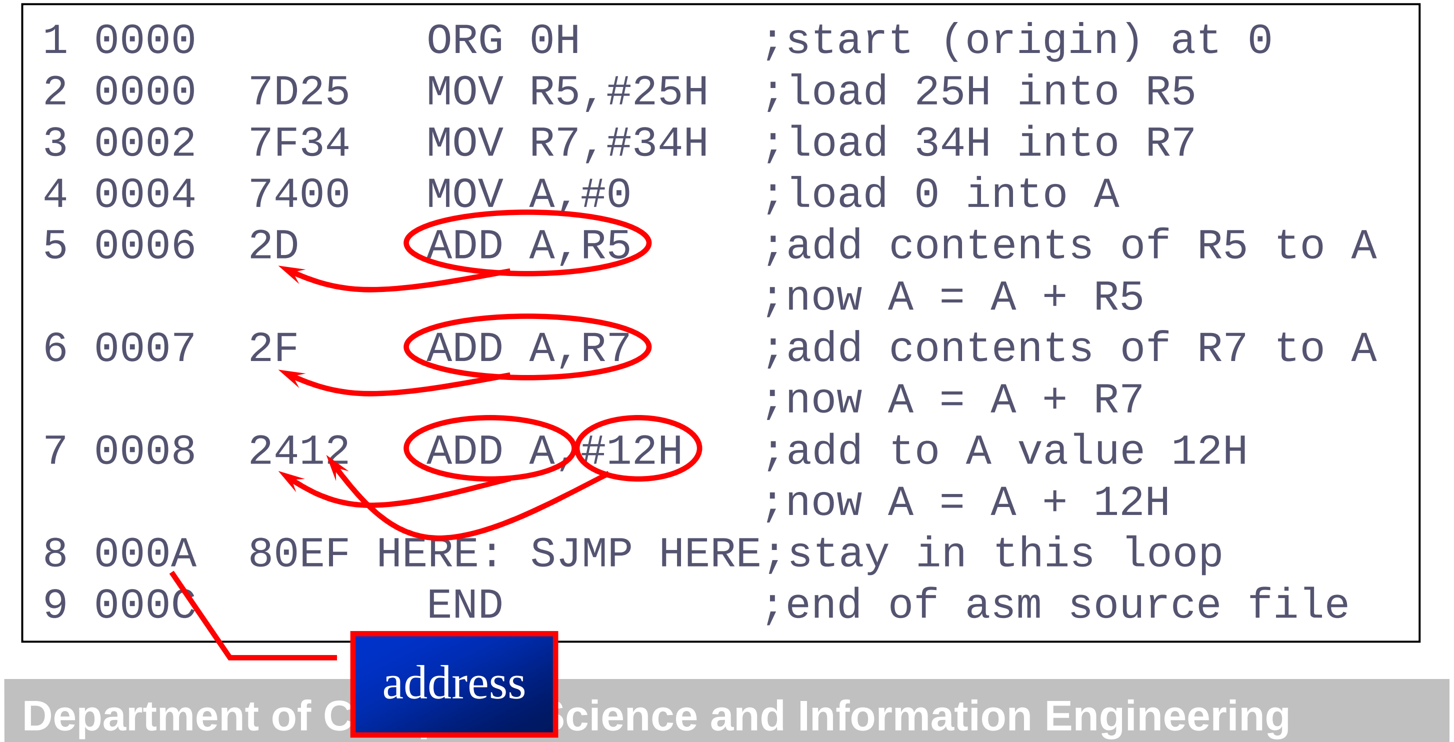
Steps to Create a Program



ASSEMBLING AND RUNNING AN 8051 PROGRAM

lst 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 AND ROM SPACE

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

PROGRAM COUNTER AND ROM SPACE

Power up

- 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

PROGRAM COUNTER AND ROM SPACE

Placing Code in ROM

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

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

PROGRAM COUNTER AND ROM SPACE

Placing Code in ROM
(cont')

- 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 COUNTER AND ROM SPACE

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')

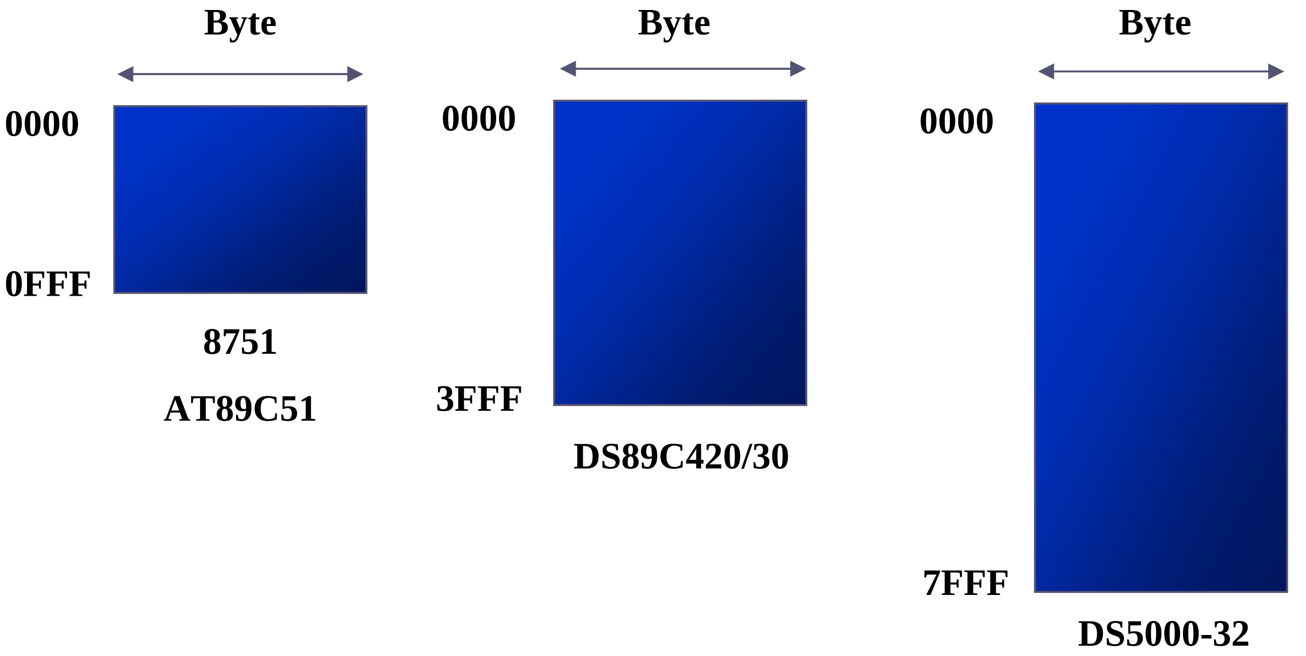
- (cont')

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*

PROGRAM COUNTER AND ROM SPACE

ROM Memory Map in 8051 Family

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



8051 DATA TYPES AND DIRECTIVES

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

- 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

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

```
        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 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

- 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

8051 DATA TYPES AND DIRECTIVES

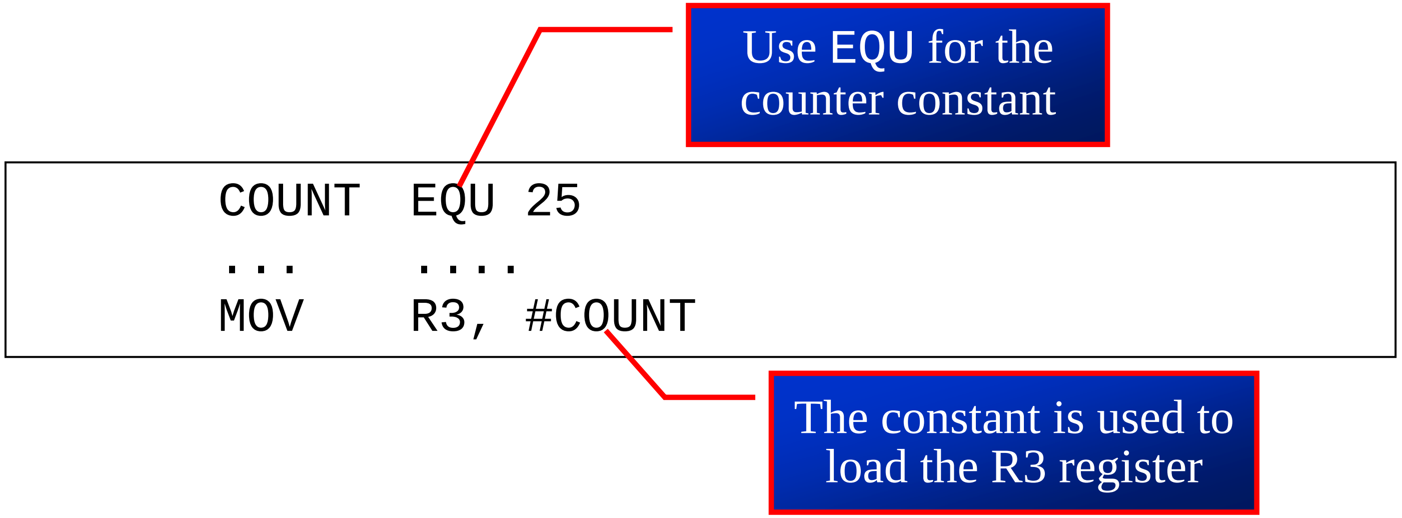
Assembler directives (cont')

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

❑ EQU (equate) (cont')

- 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



Use EQU for the counter constant

```
COUNT  EQU 25
...    ....
MOV    R3, #COUNT
```

The constant is used to load the R3 register

FLAG BITS AND PSW REGISTER

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

FLAG BITS AND PSW REGISTER

Program Status Word (cont')

CY	AC	F0	RS1	RS0	OV	--	P

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

FLAG BITS AND PSW REGISTER

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		

- 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 AND PSW REGISTER

ADD Instruction And PSW

(cont')

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)

FLAG BITS AND PSW REGISTER

ADD Instruction And PSW (cont')

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)

REGISTER BANKS AND STACK

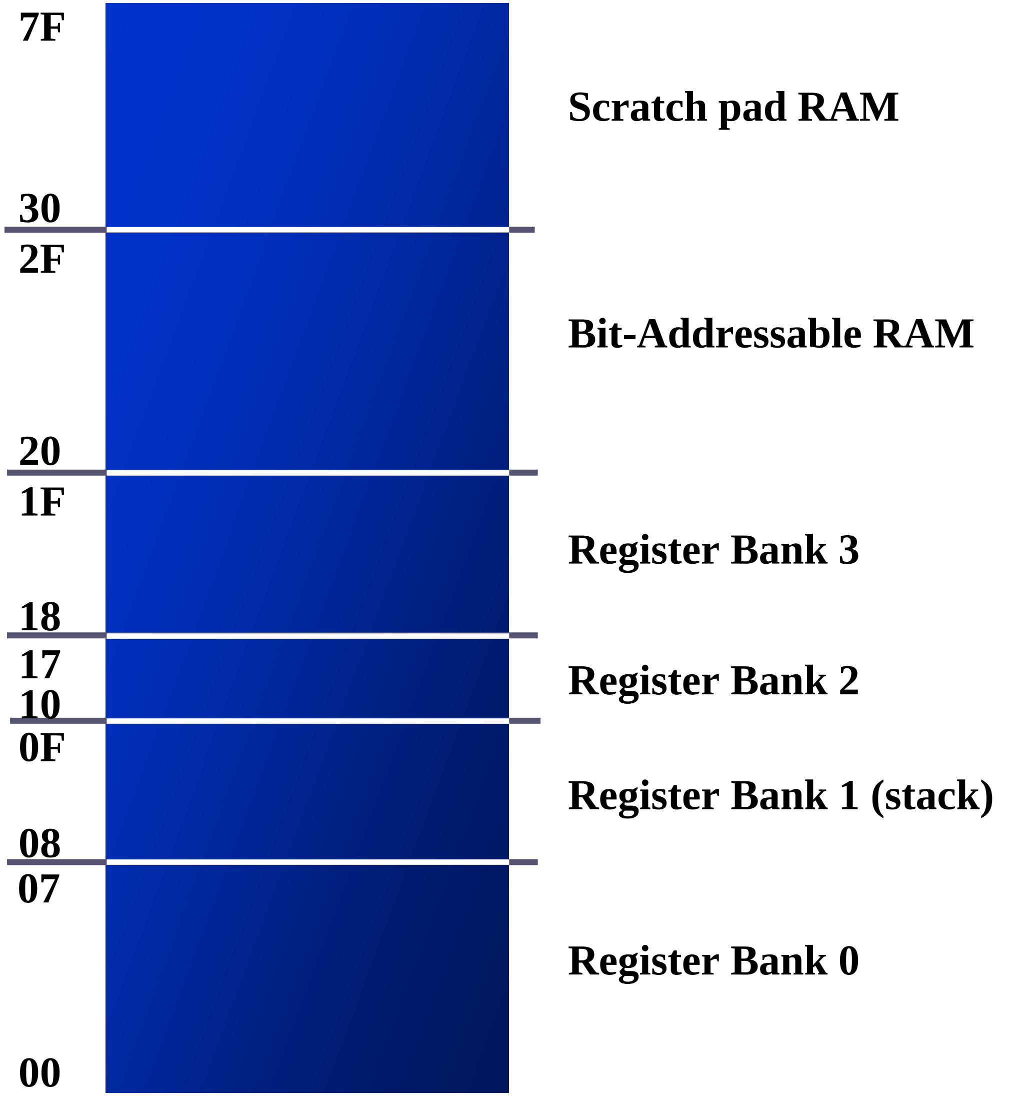
RAM Memory Space Allocation

- 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*

8051 REGISTER BANKS AND STACK

RAM Memory Space Allocation (cont')

RAM Allocation in 8051



- 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 Banks (cont')

Register banks and their RAM address

	Bank 0		Bank 1		Bank 2		Bank 3
7	R7	F	R7	17	R7	1F	R7
6	R6	E	R6	16	R6	1E	R6
5	R5	D	R5	15	R5	1D	R5
4	R4	C	R4	14	R4	1C	R4
3	R3	B	R3	13	R3	1B	R3
2	R2	A	R2	12	R2	1A	R2
1	R1	9	R1	11	R1	19	R1
0	R0	8	R0	10	R0	18	R0

Register Banks (cont')

- 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

8051 REGISTER BANKS AND STACK

Register Banks (cont')

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
```

8051 REGISTER BANKS AND STACK

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

8051 REGISTER BANKS AND STACK

Stack (cont')

- 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

8051 REGISTER BANKS AND STACK

Pushing onto Stack

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:

		After PUSH 6		After PUSH 1		After PUSH 4	
0B		0B		0B		0B	
0A		0A		0A		0A	F3
09		09		09	12	09	12
08		08	25	08	25	08	25
Start SP = 07		SP = 08		SP = 09		SP = 0A	

8051 REGISTER BANKS AND STACK

Popping From Stack

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:

0B	54
0A	F9
09	76
08	6C

Start SP = 0B

After POP 3

0B	
0A	F9
09	76
08	6C

SP = 0A

After POP 5

0B	
0A	
09	76
08	6C

SP = 09

After POP 2

0B	
0A	
09	
08	6C

SP = 08

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

8051 REGISTER BANKS AND STACK

Incrementing Stack Pointer

- 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

8051 REGISTER BANKS AND STACK

Stack and Bank 1 Conflict

- 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

8051 REGISTER BANKS AND STACK

Stack And Bank 1 Conflict (cont')

Example 2-10

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

```
MOV SP, #5FH    ;make RAM location 60H
                ;first stack location
MOV R2, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH  2
PUSH  1
PUSH  4
```

Solution:

		After PUSH 2		After PUSH 1		After PUSH 4	
63		63		63		63	
62		62		62		62	F3
61		61		61	12	61	12
60		60	25	60	25	60	25
Start SP = 5F		SP = 60		SP = 61		SP = 62	