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
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 registers 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.

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
</table>

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)**
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.

```
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
```

"#" signifies that it is a value.
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.

- 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.

```assembly
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.

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

```assembly
ORG 0H ; start (origin) at location 0
MOV R5, #25H ; load 25H into R5
MOV R7, #34H ; load 34H into R7
MOV A, #0 ; load 0 into A
ADD A, R5 ; add contents of R5 to A
; now A = A + R5
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 ; end of asm source file
```

Mnemonics produce opcodes

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

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

Comments may be at the end of a line or on a line by themselves. The assembler ignores comments
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
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
ASSEMBLING AND RUNNING AN 8051 PROGRAM

Steps to Create a Program

EDITOR PROGRAM

ASSEMBLER PROGRAM

LINKER PROGRAM

OH PROGRAM

myfile.asm

myfile.lst

myfile.obj

Other obj files

myfile.abs

myfile.hex

myfile.obj

Other obj files

myfile.lst
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>ORG 0H</td>
<td>start (origin) at 0</td>
</tr>
<tr>
<td>0000</td>
<td>7D25 MOV R5,#25H</td>
<td>load 25H into R5</td>
</tr>
<tr>
<td>0002</td>
<td>7F34 MOV R7,#34H</td>
<td>load 34H into R7</td>
</tr>
<tr>
<td>0004</td>
<td>7400 MOV A,#0</td>
<td>load 0 into A</td>
</tr>
<tr>
<td>0006</td>
<td>2D ADD A,R5</td>
<td>add contents of R5 to A; now A = A + R5</td>
</tr>
<tr>
<td>0007</td>
<td>2F ADD A,R7</td>
<td>add contents of R7 to A; now A = A + R7</td>
</tr>
<tr>
<td>0008</td>
<td>2412 ADD A,#12H</td>
<td>add to A value 12H; now A = A + 12H</td>
</tr>
<tr>
<td>000A</td>
<td>80EF HERE: SJMP HERE</td>
<td>stay in this loop</td>
</tr>
<tr>
<td>000C</td>
<td>END</td>
<td>end of asm source file</td>
</tr>
</tbody>
</table>
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
Examine the list file and how the code is placed in ROM

<table>
<thead>
<tr>
<th>ROM Address</th>
<th>Machine Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>ORG 0H</td>
<td>start (origin) at 0</td>
</tr>
<tr>
<td>0000</td>
<td>7D25 MOV R5,#25H</td>
<td>load 25H into R5</td>
</tr>
<tr>
<td>0002</td>
<td>7F34 MOV R7,#34H</td>
<td>load 34H into R7</td>
</tr>
<tr>
<td>0004</td>
<td>7400 MOV A,#0</td>
<td>load 0 into A</td>
</tr>
<tr>
<td>0006</td>
<td>2D ADD A,R5</td>
<td>add contents of R5 to A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>now A = A + R5</td>
</tr>
<tr>
<td>0007</td>
<td>2F ADD A,R7</td>
<td>add contents of R7 to A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>now A = A + R7</td>
</tr>
<tr>
<td>0008</td>
<td>2412 ADD A,#12H</td>
<td>add to A value 12H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>now A = A + 12H</td>
</tr>
<tr>
<td>000A</td>
<td>80EF HERE: SJMP HERE</td>
<td>stay in this loop</td>
</tr>
<tr>
<td>000C</td>
<td>END</td>
<td>end of asm source file</td>
</tr>
</tbody>
</table>
After the program is burned into ROM, the opcode and operand are placed in ROM memory location starting at 0000

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>7D</td>
</tr>
<tr>
<td>0001</td>
<td>25</td>
</tr>
<tr>
<td>0002</td>
<td>7F</td>
</tr>
<tr>
<td>0003</td>
<td>34</td>
</tr>
<tr>
<td>0004</td>
<td>74</td>
</tr>
<tr>
<td>0005</td>
<td>00</td>
</tr>
<tr>
<td>0006</td>
<td>2D</td>
</tr>
<tr>
<td>0007</td>
<td>2F</td>
</tr>
<tr>
<td>0008</td>
<td>24</td>
</tr>
<tr>
<td>0009</td>
<td>12</td>
</tr>
<tr>
<td>000A</td>
<td>80</td>
</tr>
<tr>
<td>000B</td>
<td>FE</td>
</tr>
</tbody>
</table>
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
(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*
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

```
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" ; ASCII NUMBERS
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.

Define ASCII strings larger than two characters

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

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

  Use **EQU** for the counter constant

  The constant is used to load the R3 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
FLAG BITS AND PSW REGISTER

Program Status Word (cont’)

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

<table>
<thead>
<tr>
<th>CY</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>--</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>PSW.7</td>
<td>Carry flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td>PSW.6</td>
<td>Auxiliary carry flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>PSW.5</td>
<td>Available to the user for general purpose</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS1</td>
<td>PSW.4</td>
<td>Register Bank selector bit 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS0</td>
<td>PSW.3</td>
<td>Register Bank selector bit 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OV</td>
<td>PSW.2</td>
<td>Overflow flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>PSW.1</td>
<td>User definable bit.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>PSW.0</td>
<td>Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of 1 bits in the accumulator.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A carry from D3 to D4

Carry out from the d7 bit

Reflect the number of 1s in register A

<table>
<thead>
<tr>
<th>RS1</th>
<th>RS0</th>
<th>Register Bank</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00H – 07H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>08H – 0FH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10H – 17H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>18H – 1FH</td>
</tr>
</tbody>
</table>
## Instructions that affect flag bits

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CY</th>
<th>OV</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ADDC</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUBB</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUL</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB C</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR C</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPL C</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL C, /bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORL C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORL C, /bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CJ NE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 bit
P = 1 since the accumulator has an odd number of 1s (it has five 1s)
Example 2-3
Show the status of the CY, AC and P flag after the addition of 9CH and 64H in the following instructions.

\[
\text{MOV } A, \ #9CH \\
\text{ADD } A, \ #64H \ ; \text{after the addition } A=00H, \ CY=1
\]

Solution:

\[
\begin{array}{c}
9C & 10011100 \\
+ & 01100100 \\
\hline
100 & 00000000 \\
\end{array}
\]

CY = 1 since there is a carry beyond the D7 bit
AC = 1 since there is a carry from the D3 to the D4 bit
P = 0 since the accumulator has an even number of 1s (it has zero 1s)
Example 2-4

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

\[ \text{MOV A, #88H} \]
\[ \text{ADD A, #93H} \quad \text{;after the addition A=1BH, CY=1} \]

**Solution:**

\[
\begin{array}{c|c}
88 & 10001000 \\
+ 93 & 10010011 \\
\hline
11B & 00011011 \\
\end{array}
\]

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

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

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

- Scratch pad RAM
- Bit-Addressable RAM
- Register Bank 3
- Register Bank 2
- Register Bank 1 (stack)
- Register Bank 0
Register Banks

- 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
8051 REGISTER BANKS AND STACK

Register Banks (cont’)

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R7</td>
<td>F</td>
<td>R7</td>
</tr>
<tr>
<td>6</td>
<td>R6</td>
<td>E</td>
<td>R6</td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td>D</td>
<td>R5</td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td>C</td>
<td>R4</td>
</tr>
<tr>
<td>3</td>
<td>R3</td>
<td>B</td>
<td>R3</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>A</td>
<td>R2</td>
</tr>
<tr>
<td>1</td>
<td>R1</td>
<td>9</td>
<td>R1</td>
</tr>
<tr>
<td>0</td>
<td>R0</td>
<td>8</td>
<td>R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
<td>R0</td>
</tr>
</tbody>
</table>

Register banks and their RAM address

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

<table>
<thead>
<tr>
<th>Bank</th>
<th>RS1(PSW.4)</th>
<th>RS0(PSW.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bank 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Bank 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bank 3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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
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
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 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:

<table>
<thead>
<tr>
<th>Reg</th>
<th>After PUSH 6</th>
<th>After PUSH 1</th>
<th>After PUSH 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>0B</td>
<td>0B</td>
<td>0B</td>
</tr>
<tr>
<td>0A</td>
<td>0A</td>
<td>0A</td>
<td>0A F3</td>
</tr>
<tr>
<td>09</td>
<td>09</td>
<td>09 12</td>
<td>09 12</td>
</tr>
<tr>
<td>08</td>
<td>08 25</td>
<td>08 25</td>
<td>08 25</td>
</tr>
</tbody>
</table>

Start SP = 07  SP = 08  SP = 09  SP = 0A
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.
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
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
Example 2-10

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

```assembly
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:

<table>
<thead>
<tr>
<th></th>
<th>After PUSH 2</th>
<th>After PUSH 1</th>
<th>After PUSH 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>62</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>61</td>
<td>61</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td>60</td>
<td>60 25</td>
<td>60 25</td>
<td>60 25</td>
</tr>
</tbody>
</table>

Start SP = 5F  SP = 60  SP = 61  SP = 62