Registers are used in the CPU to store information on temporarily basis which could be data to be processed, or an address pointing to the data which is to be fetched. In 8051, there is one data type is of 8-bits, from the MSB (most significant bit) D7 to the LSB (least significant bit) D0. With 8-bit data type, any data type larger than 8-bits must be broken into 8-bit chunks before it is processed.

The most widely used registers of the 8051 are A (accumulator), B, R0-R7, DPTR (data pointer), and PC (program counter). All these registers are of 8-bits, except DPTR and PC.

Storage Registers in 8051

We will discuss the following types of storage registers here −

- Accumulator
- R register
- B register
- Data Pointer (DPTR)
- Program Counter (PC)
- Stack Pointer (SP)

Accumulator

The accumulator, register A, is used for all arithmetic and logic operations. If the accumulator is not present, then every result of each calculation (addition, multiplication, shift, etc.) is to be stored into the main memory. Access to main memory is slower than access to a register like the accumulator because the technology used for the large main memory is slower (but cheaper) than that used for a register.

The "R" Registers

The "R" registers are a set of eight registers, namely, R0, R1 to R7. These registers function as auxiliary or temporary storage registers in many operations. Consider an example of the sum of 10 and 20. Store a variable 10 in an accumulator and another variable 20 in, say, register R4. To process the addition operation, execute the following command −

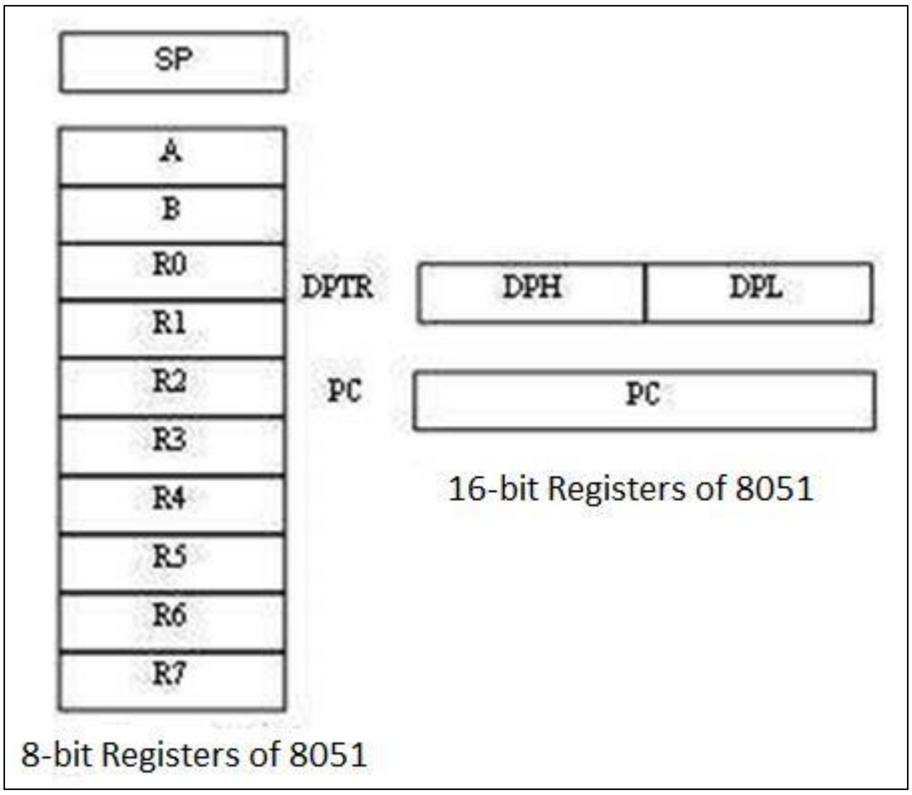
```
ADD A, R4
```

After executing this instruction, the accumulator will contain the value 30. Thus "R" registers are very important auxiliary or **helper registers**. The Accumulator alone would not be very useful if it were not for these "R" registers. The "R" registers are meant for temporarily storage of values.

Let us take another example. We will add the values in R1 and R2 together and then subtract the values of R3 and R4 from the result.

```
MOV A, R3     ; Move the value of R3 into the accumulator
ADD A, R4     ; add the value of R4
MOV R5, A     ; Store the resulting value temporarily in R5
MOV A, R1     ; Move the value of R1 into the accumulator
ADD A, R2     ; add the value of R2
SUBB A, R5; Subtract the value of R5 (which now contains R3 + R4)
```

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this is not the most efficient way to calculate (R1 + R2) – (R3 + R4), but it does illustrate the use of the "R" registers as a way to store values temporarily.



The "B" Register

The "B" register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value. The "B" register is used only by two 8051 instructions: **MUL AB** and **DIV AB**. To quickly and easily multiply or divide A by another number, you may store the other number in "B" and make use of these two instructions. Apart from using MUL and DIV instructions, the "B" register is often used as yet another temporary storage register, much like a ninth R register.

The Data Pointer

The Data Pointer (DPTR) is the 8051's only user-accessible 16-bit (2-byte) register. The Accumulator, R0–R7 registers and B register are 1-byte value registers. DPTR is meant for pointing to data. It is used by the 8051 to access external memory using the address indicated by DPTR. DPTR is the only 16-bit register available and is often used to store 2-byte values.

The Program Counter

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute can be found in the memory. PC starts at 0000h when the 8051 initializes and is incremented every time after an instruction is executed. PC is not always incremented by 1. Some instructions may require 2 or 3 bytes; in such cases, the PC will be incremented by 2 or 3.

Branch, jump, and **interrupt** operations load the Program Counter with an address other than the next sequential location. Activating a power-on reset will cause all values in the register to be lost. It means the value of the PC is 0 upon reset, forcing the CPU to fetch the first opcode from the ROM location 0000. It means we must place the first byte of upcode in ROM location 0000 because that is where the CPU expects to find the first instruction.

The Stack Pointer (SP)

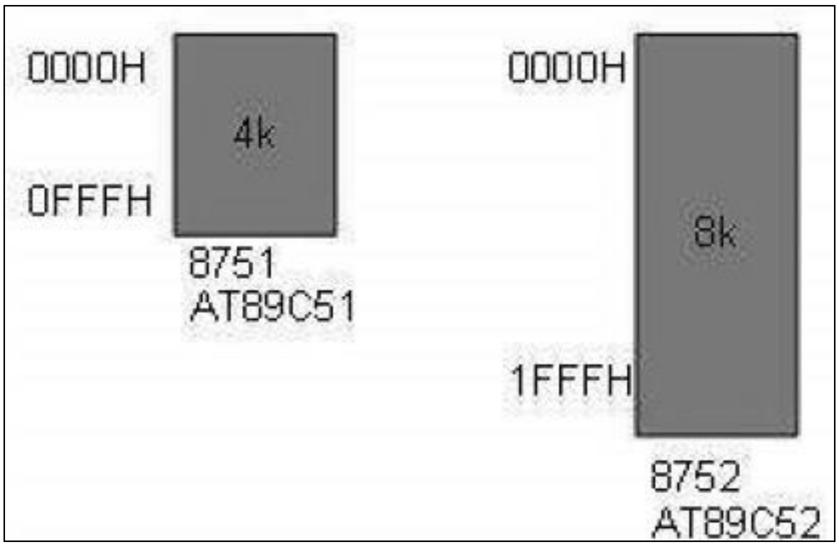
The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer tells the location from where the next value is to be removed from the stack. When a value is pushed onto the stack, the value of SP is incremented and then the value is stored at the resulting memory location. When a value is popped off the stack, the value is returned from the memory location indicated by SP, and then the value of SP is decremented.

This order of operation is important. SP will be initialized to 07h when the 8051 is initialized. If a value is pushed onto the stack at the same time, the value will be stored in the internal RAM address 08h because the 8051 will first increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h). SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI.

ROM Space in 8051

Some family members of 8051 have only 4K bytes of on-chip ROM (e.g. 8751, AT8951); some have 8K ROM like AT89C52, and there are some family members with 32K bytes and 64K bytes of on-chip ROM such as Dallas Semiconductor. The point to remember is that no member of the 8051 family can access more than 64K bytes of opcode since the program counter in 8051 is a 16-bit register (0000 to FFFF address).

The first location of the program ROM inside the 8051 has the address of 0000H, whereas the last location can be different depending on the size of the ROM on the chip. Among the 8051 family members, AT8951 has $k bytes of on-chip ROM having a memory address of 0000 (first location) to 0FFFH (last location).



8051 Flag Bits and PSW Register

The program status word (PSW) register is an 8-bit register, also known as **flag register**. It is of 8-bit wide but only 6-bit of it is used. The two unused bits are **user-defined flags**. Four of the flags are called **conditional flags**, which mean that they indicate a condition which results after an instruction is executed. These four are **CY** (Carry), **AC** (auxiliary carry), **P** (parity), and **OV** (overflow). The bits RS0 and RS1 are used to change the bank registers. The following figure shows the program status word register.

The PSW Register contains that status bits that reflect the current status of the CPU.

CY	CA	F0	RS1	RS0	OV	-	P

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag
F0	PSW.5	Flag 0 available to 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 FLAG
P	PSW.0	Parity FLAG. Set/ cleared by hardware during instruction cycle to indicate even/odd number of 1 bit in accumulator.

We can select the corresponding Register Bank bit using RS0 and RS1 bits.

RS1	RS2	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

- **CY, the carry flag −** This carry flag is set (1) whenever there is a carry out from the D7 bit. It is affected after an 8-bit addition or subtraction operation. It can also be reset to 1 or 0 directly by an instruction such as "SETB C" and "CLR C" where "SETB" stands for set bit carry and "CLR" stands for clear carry.
- **AC, auxiliary carry flag −** If there is a carry from D3 and D4 during an ADD or SUB operation, the AC bit is set; otherwise, it is cleared. It is used for the instruction to perform binary coded decimal arithmetic.
- **P, the parity flag −** the parity flag represents the number of 1's in the accumulator register only. If the A register contains odd number of 1's, then P = 1; and for even number of 1's, P = 0.
- **OV, the overflow flag −** this flag is set whenever the result of a signed number operation is too large causing the high-order bit to overflow into the sign bit. It is used only to detect errors in signed arithmetic operations.

Example

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

MOV A, #9CH

ADD A, # 64H

```
Solution:   9C     10011100
           +64     01100100
           100     00000000
```

CY = 1 since there is a carry beyond D7 bit

AC = 0 since there is a carry from D3 to D4

P = 0 because the accumulator has even number of 1's

To register banks and stack in an 8051 microcontroller, let's first understand the basic concepts.

8051 Microcontroller Registers: The 8051 microcontroller has various registers used for different purposes. These registers can be classified into four banks: Bank 0, Bank 1, Bank 2, and Bank 3. Each bank consists of eight registers, numbered R0 to R7.

The registers R0 to R7 are general-purpose registers that can be used for storing data during program execution. The banked registers are used for specific purposes such as the stack pointer, data pointer, and program status word (PSW).

Stack: The stack is a special area in the microcontroller's memory used for storing temporary data during program execution. It follows the Last-In-First-Out (LIFO) principle, meaning the last data pushed onto the stack is the first one to be popped off.

The stack is essential for storing return addresses when executing subroutines or interrupts. It also helps in saving and restoring the context of the program during interrupt service routines.

Register Banks and Stack in 8051: The 8051 microcontroller uses two register banks at a time: Bank 0 and Bank 1. By default, Bank 0 is selected after a reset, and its registers R0 to R7 are accessible. However, to access registers R0 to R7 of Bank 1, you need to set the RS1 and RS0 bits of the PSW (program status word) register appropriately.

To switch between register banks, you can modify the RS1 and RS0 bits of the PSW register. For example, to select Bank 1, you set RS1=1 and RS0=0. To select Bank 0, you set RS1=0 and RS0=0.

Here's an example code snippet to demonstrate register bank switching:

```
ORG 0x0000            ; Start of program memory

; Main program

MAIN:

    MOV A, #0xFF      ; Move value FFH to accumulator A

    MOV R0, A         ; Move accumulator A to register R0
```

```
SETB RS1          ; Set RS1=1 to select Bank 1
CLR RS0           ; Set RS0=0 to select Bank 1
MOV A, R0         ; Move register R0 to accumulator A (from Bank 1)
                   ; Perform operations with Bank 1 registers
... CLR RS1         ; Set RS1=0 to select Bank 0
CLR RS0            ; Set RS0=0 to select Bank 0
MOV R0, A          ; Move accumulator A to register R0 (in Bank 0)

; Perform operations with Bank 0 registers
...
SJMP MAIN         ; Jump back to the main program
END               ; End of program
```

In the above example, the SETB and CLR instructions are used to set or clear the RS1 and RS0 bits, respectively. This allows you to switch between register banks and access the desired registers for performing operations.

Note that the stack in the 8051 microcontroller is located in the internal RAM of the microcontroller. The stack pointer (SP) is used to keep track of the top of the stack. When pushing data onto the stack, the SP is decremented, and when popping data off the stack, the SP is incremented.

You can use the PUSH and POP instructions to push data onto the stack or pop data off the stack, respectively.

I/O Programming

I/O port programming is an essential aspect of working with the 8051 microcontroller. The 8051 microcontroller has four I/O ports: Port 0 (P0), Port 1 (P1), Port 2 (P2), and Port 3 (P3). These ports can be used for both input and output operations, allowing you to interface with external devices such as sensors, displays, and other peripherals.

Here are some common operations and techniques for I/O port programming in the 8051 microcontroller:

1. Configuring I/O Ports: Before using an I/O port, you need to configure it as either input or output. Each port pin can be individually configured as input (logic high impedance) or output (logic low impedance) using the corresponding bits in the port's associated register.

For example, to configure Port 1 as output and Port 2 as input, you can use the following code:

```
MOV P1, #0xFF     ; Configure all pins of Port 1 as output
MOV P2, #0x00     ; Configure all pins of Port 2 as input
```

2. Reading from Input Ports: To read the status of the input pins of a port, you can use the port's associated register. The values read from the register indicate the logic level of the pins. A logic high (1) indicates that the pin is receiving a high voltage, while a logic low (0) indicates a low voltage.

For example, to read the status of the pins in Port 3 and store the result in the accumulator (A), you can use the following code:

```
MOV A, P3     ; Read the status of Port 3 and store it in accumulator A
```

3. Writing to Output Ports: To set the output values of the pins in an output port, you can use the port's associated register. Writing a logic high (1) to a pin sets it to a high voltage level, while writing a logic low (0) sets it to a low voltage level.

For example, to set the pins of Port 0 to high and the pins of Port 1 to low, you can use the following code:

```
MOV P0, #0xFF     ; Set all pins of Port 0 to high
MOV P1, #0x00     ; Set all pins of Port 1 to low
```

4. Manipulating Individual Pins: Sometimes, you may need to manipulate individual pins of a port while keeping the others unchanged. You can achieve this using bitwise operations such as AND, OR, XOR, and bit shifting.

For example, to toggle the state of pin P2.3 while keeping the other pins of Port 2 unchanged, you can use the following code:

```
CPL P2.3      ; Toggle the state of pin P2.3
```

5. External Pull-Up/Down Resistors: The 8051 microcontroller doesn't have internal pull-up or pull-down resistors. If you want to use external pull-up or pull-down resistors for the input pins, you need to connect them externally to the corresponding pins.

These are some of the basic techniques for I/O port programming in the 8051 microcontroller. You can use these concepts to interface with various external devices and create complex applications. Remember to consult the datasheet of your specific microcontroller variant for precise details on the available ports and their registers.

Addressing modes of 8051

In this section, we will see different addressing modes of the 8051 microcontrollers. In 8051 there is 1-byte, 2-byte instructions and very few 3-byte instructions are present. The opcodes are 8-bit long. As the opcodes are 8-bit data, there are 256 possibilities. Among 256, 255 opcodes are implemented.

The clock frequency is12MHz, so 64 instruction types are executed in just 1 µs, and rest are just 2 µs. The Multiplication and Division operations take 4 µsto to execute.

In 8051 there are six types of addressing modes.

1. Immediate Addressing Mode
2. Register Addressing Mode
3. Direct Addressing Mode
4. Register Indirect Addressing Mode
5. Indexed Addressing Mode
6. Implied Addressing Mode

1. Immediate addressing mode

In this Immediate Addressing Mode, the data is provided in the instruction itself. The data is provided immediately after the opcode. These are some examples of Immediate Addressing Mode.

MOVA, #0AFH;

MOVR3, #45H;

MOVDPTR, #FE00H;

In these instructions, the # symbol is used for immediate data. In the last instruction, there is DPTR. The DPTR stands for Data Pointer. Using this, it points the external data memory location. In the first instruction, the immediate data is AFH, but one 0 is added at the beginning. So when the data is starting with A to F, the data should be preceded by 0.

2. Register addressing mode

In the register addressing mode the source or destination data should be present in a register (R0 to R7). These are some examples of Register Addressing Mode.

MOVA, R5;

MOVR2, #45H;

MOVR0, A;

In 8051, there is no instruction like **MOVR5, R7**. But we can get the same result by using this instruction **MOV R5, 07H**, or by using **MOV 05H, R7**. But this two instruction will work when the selected register bank is **RB0**. To use another register bank and to get the same effect, we have to add the starting address of that register bank with the register number. For an example, if the RB2 is selected, and we want to access R5, then the address will be (10H + 05H = 15H), so the instruction will look like this **MOV 15H, R7**. Here 10H is the starting address of Register Bank 2.

3. Direct Addressing Mode

In the Direct Addressing Mode, the source or destination address is specified by using 8-bit data in the instruction. Only the internal data memory can be used in this mode. Here some of the examples of direct Addressing Mode.

MOV80H, R6;

MOVR2, 45H;

MOVR0, 05H;

The first instruction will send the content of registerR6 to port P0 (Address of Port 0 is 80H). The second one is forgetting content from 45H to R2. The third one is used to get data from Register R5 (When register bank RB0 is selected) to register R5.

4. Register indirect addressing Mode

In this mode, the source or destination address is given in the register. By using register indirect addressing mode, the internal or external addresses can be accessed. The R0 and R1 are used for 8-bit addresses, and DPTR is used for 16-bit addresses, no other registers can be used for addressing purposes. Let us see some examples of this mode.

```
MOV0E5H, @R0;
MOV@R1, 80H
```

In the instructions, the @ symbol is used for register indirect addressing. In the first instruction, it is showing that theR0 register is used. If the content of R0 is 40H, then that instruction will take the data which is located at location 40H of the internal RAM. In the second one, if the content of R1 is 30H, then it indicates that the content of port P0 will be stored at location 30H in the internal RAM.

```
MOVXA, @R1;
MOV@DPTR, A;
```

In these two instructions, the X in MOVX indicates the external data memory. The external data memory can only be accessed in register indirect mode. In the first instruction if the R0 is holding 40H, then A will get the content of external RAM location40H. And in the second one, the content of A is overwritten in the location pointed by DPTR.

5. Indexed addressing mode

In the indexed addressing mode, the source memory can only be accessed from program memory only. The destination operand is always the register A. These are some examples of Indexed addressing mode.

```
MOVCA, @A+PC;
MOVCA, @A+DPTR;
```

The C in MOVC instruction refers to code byte. For the first instruction, let us consider A holds 30H. And the PC value is1125H. The contents of program memory location 1155H (30H + 1125H) are moved to register A.

6. Implied Addressing Mode

In the implied addressing mode, there will be a single operand. These types of instruction can work on specific registers only. These types of instructions are also known as register specific instruction. Here are some examples of Implied Addressing Mode.

```
RLA;

SWAPA;
```

These are 1- byte instruction. The first one is used to rotate the A register content to the Left. The second one is used to swap the nibbles in A.