# INSTRUCTION SET OF8086

The instructions of 8086 are classified into data transfer, arithmetic, logical, flag manipulation, control transfer, shift/rotate, string and machine control instructions.

**Data transfer instructions**:

The data transfer instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LSS, LAHF and SAHF

**MOV:** MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register or a memory location or an immediate number. The general format of MOV instruction is

MOV Destination, Source Examples:

MOV BL, 50H; Move immediate data 50H to BL

**PUSH:** PUSH instruction is used to store the word in a register or a memory location into stack as explained in stack addressing modes. SP is decremented by 2 after execution of PUSH.

Examples:

PUSH CX; PUSH CX content in stack

**POP:** POP instruction copies the top word from the stack to a destination specified in the instruction. The destination can be a general purpose register, a segment register or a memory location. After the word is copied to the specified destination, the SP is incremented by 2 to point to the next word in the stack.

Examples:

POP BX; Pop BX content from the stack

**XCHG:** The XCHG instruction exchanges the contents of a register with the contents of a memory location. It cannot exchange directly the contents of two memory locations. The source and destination must both be words or they must both be bytes. The segment registers cannot be used in this instruction.

Examples:

XCHG AL, BL; Exchanges content of AL and BL

XCHG AX, [BX]; Exchanges content of AX with content of memory at [BX]

**XLAT:** The XLAT instruction is used to translate a byte in AL from one code to another code. The instruction replaces a byte in the AL register with a byte in memory at [BX], which is data in a lookup table present in memory.

Before XLAT is executed, the lookup table containing the desired codes must be put in data segment and the offset address of the starting location of the lookup table is stored in BX. The code byte to be translated is put in AL. When XLAT is now executed, it adds the content of the AL with BX to find the offset address of the data in the above lookup table and the byte in that offset address is copied to AL.

**IN:** The IN instruction copies data from a port to AL or AX register. If an 8-bit port is read, the data is stored in AL and if a 16 bit port is read, the data is stored in AX. The IN instruction has two formats namely **fixed port and variable port.**

For the fixed port type IN instruction, the 8-bit address of a port is specified directly in the instruction. With this form, anyone of 256 possible ports can be addressed.

Examples:

IN AL, 80H; Input a byte from the port with address 80H to AL IN AX, 40H; Input a word from port with address 40H to AX

For the variable port type IN instruction, the port address is loaded into DX register before the IN instruction. Since DX is a 16 bit register, the port address can be any number between 0000H and FFFFH. Hence up to 65536 ports are addressable in this mode. The following example shows a part of the program having IN instruction and the operations done when the instructions are executed are given in the corresponding comment field.

Examples:

MOV DX, 0FE50H; Initialize DX with port address of FE50H

IN AL, DX ; Input a byte from 8-bit port with port address FE50H into AL IN AX, DX ; Input a word from 16-bit port with port address FE50H into AX

**OUT:** The OUT instruction transfers a byte from AL or a word from AX to the specified port. Similar to IN instruction, OUT instruction has two forms namely fixed port and variable port.

**LEA:** Load Effective Address

The general format of LEA instruction is LEA register, source

This instruction determines the offset address of the variable or memory location named as the source and puts this offset address in the indicated 16 bit register.

Examples:

LEA BX, COST; Load BX with offset address of COST in data segment where COST is the name assigned to a memory location in data segment.

LEA CX, [BX][SI]; Load CX with the value equal to (BX)+(SI) where (BX) and (SI) represents content of BX and SI respectively.

**LDS:** Load register and DS with words from memory the general form of this instruction is

LDS register, memory address of first word

The LDS instruction copies a word from the memory location specified in the instruction into the register and then copies a word from the next memory location into the DS register.

LDS is useful for initializing SI and DS registers at the start of a string before using one of the String instructions.

Example:

LDS SI,[2000H]; Copy content of memory at offset address 2000H in data segment to lower byte of SI, content of 2001H to higher byte of SI. Copy content at offset address 2002H in data segment to lower byte of DS and 2003H to higher byte of DS.

**LES, LSS:** LES and LSS instructions are similar to LDS instruction except that instead of DS register, ES and SS registers are loaded respectively along with the register specified in the instruction.

**LAHF:** This instruction copies the low byte of flag register into AH.

**SAHF:** Store content of AH in the low byte of flag register.

Except SAHF and POPF instructions, all other data transfer instructions do not affect flag register.

**Arithmetic and Logical instructions**

**ADD:** The general format of ADD instruction is ADD destination, source

The data from the source and destination are added and the result is placed in the destination. The source may be an immediate number, a register or a memory location. The destination can be a register or memory location. But the source and destination cannot both be memory locations. The data from the source and destination must be of the same type either bytes or words.

Examples:

ADD BL,80H; Add immediate data 80H to BL

ADD AX,CX; Add content of AX and CX and store result in AX

ADD AL,[BX]; Add content of AL and the byte from memory at [BX] and store result in AL. The flags AF, CF, OF, PF, SF and ZF flags are affected by the execution of ADD instruction

**ADC:** This instruction adds the data in source and destination along with the content of carry flag and stores the result in the destination. The general format of this instruction is

ADC destination, source

**SUB:** The general form of subtract (SUB) instruction is SUB destination, source

It subtracts the number in the source from the number in the destination and stores the result in destination. The source may be an immediate number, a register or a memory location. The destination can be a register or memory location. But the source and destination cannot both be memory locations. The data from the source and destination must be of the same type either bytes or words.

For subtraction, the carry flag (CF) functions as borrow flag. If the result is negative after subtraction, CF is set, otherwise it is reset. The rules for source and destination are same as that of ADD instruction. The flags AF, CF, OF, PF, SF and ZF are affected by SUB instruction.

**SBB:** Subtract with Borrow

The general form of this instruction is SBB destination, source

SBB instruction subtracts the content of source and content of carry flag from the content of destination and stores the result in destination. The rules for the source and destination are same as that of SUB instruction. AF, CF, OF, PF, SF and ZF are affected by this instruction.

**INC:** The increment (INC) instruction adds 1 to a specified register or to a memory location. The data incremented may be a byte or word. Carry flag is not affected by this instruction. AF, OF, PF, SF and ZF flags are affected.

Examples:

INC CL; Increment content of CL by 1

INC AX; Increment content of AX by 1

INC BYTE PTR [BX]; Increment byte in memory at [BX] by 1 INC WORD PTR [SI]; Increment word in memory at [SI] by 1

In the above examples, the term BYTE PTR and WORD PTR are assembler directives which are used to specify the type of data (byte or word respectively) to be incremented in memory.

**DEC:** The decrement (DEC) instruction subtracts 1 from the specified register or memory location. The data decremented may be a byte or word. CF is not affected and AF,OF, PF, SF and ZF flags are affected by this instruction.

**NEG:** The negate (NEG) instruction replaces the byte or word in the specified register or memory location by its 2's complement (i.e. changing the sign of the data). CF,AF, SF, PF, ZF and OF flags are affected by this instruction.

Examples:

NEG AL; Take 2's complement of the data in AL and store it in AL

NEG BYTE PTR [BX]; Take 2's complement of the byte in memory at [BX] and store result in the same place.

**CMP:** The general form of compare (CMP) instruction is given below:

CMP destination, source

This instruction compares a byte or word in the source with a byte or word in the destination and affects only the flags according to the result. The content of source and destination are not affected by the execution of this instruction. The comparison is done by subtracting the content of source from destination.

AF, OF, SF, ZF, PF and CF flags are affected by the instruction. The rules for source and destination are same as that of SUB instruction.

Example:

After the instruction CMP AX, DX is executed, the status of CF, ZF and SF will be as follows:

| | CF | ZF | SF |
|---|---|---|---|
| If AX=DX | 0 | 1 | 0 |
| If AX>DX | 0 | 0 | 0 |
| If AX<DX | 1 | 0 | 1 |

**MUL:** The multiply (MUL) instruction is used for multiplying two unsigned bytes or words. The general form of MUL instruction is

MUL Source

The source can be byte or word from a register or memory location which is considered as the multiplier. The multiplicand is taken by default from AL or AX for byte or word type data respectively. The result of multiplication is stored in AX or DX-AX (i.e. Most significant word of result in DX and least significant word of result in AX) for byte or word type data respectively. (Note: Multiplying two 8 bit data gives 16 bit result and multiplying two 16 bit data gives 32 bit result).

Examples:

MUL CH; Multiply AL and CH and store result in AX MUL BX; Multiply AX and BX and store result in DX-AX

MUL BYTE PTR [BX]; multiply AL with the byte in memory at [BX] and store result in DX-AX

If the most significant byte of the 16 bit result is 00H or the most significant word of a 32 bit result is 0000h, both CF and OF will both be 0s. Checking these flags allows us to decide whether the leading 0s in the result have to be discarded or not. AF, PF, SF and ZF flags are undefined (i.e. some random number will be stored in these bits) after the execution of MUL instruction

**IMUL:** The IMUL instruction is used for multiplying signed byte or word in a register or memory location with AL or AX respectively and stores the result in AX or DX-AX respectively. If the magnitude of the result does not require all the bits of the destination, the unused bits are filled with copies of the sign bit. If the upper byte of a 16 bit result or upper word of a 32 bit result contains only copies of the sign bit (all 0s or all 1s) then CF and OF will both be 0 otherwise both will be 1. AF, PF, SF and ZF are undefined after IMUL. To multiply a signed byte by a signed word, the byte is moved into a word location and the upper byte of the word is filled with the copies of the sign bit. If the byte is moved into AL, by using the CBW (Convert Byte to Word) instruction, the sign bit in AL is extended into all the bits of AH. Thus AX contains the 16 bit sign extended word.

Examples:

IMUL BL; multiply AL with BL and store result in AX IMUL AX; multiply AX and AX and store result in DX-AX

IMUL BYTE PTR [BX]; multiply AL with byte from memory at [BX] and store result in AX IMUL WORD PTR [SI]; Multiply AX with word from memory at [SI] and store result in DX-AX

**DIV:** The divide (DIV) instruction is used for dividing unsigned data. The general form of DIV instruction is

DIV source

Where source is the divisor and it can be a byte or word in a register or memory location. The dividend is taken by default from AX and DX-AX for byte or word type data division respectively.

Examples

DIV DL; Divide word in AX by byte in DL. Quotient is stored in AL and remainder is stored in AH

DIV CX; Divide double word (32 bits) in DX-AX by word in CX. Quotient is stored in AX and remainder is stored in DX

DIV BYTE PTR [BX]; Divide word in AX by byte from memory at [BX]. Quotient is stored in AL and remainder is stored in AH.

**IDIV:** The IDIV instruction is used for dividing signed data. The general form and the rules for IDIV instruction are same as DIV instruction. The quotient will be a signed number and the sign of the remainder is same as the sign of the dividend.

To divide a signed byte by a signed byte, the dividend byte is put in AL and using **CBW** (Convert Byte to Word) instruction, the sign bit of the data in AL is extended to AH and thereby the byte in AL is converted to signed word in AX. To divide a signed word by a signed word, the dividend byte is put in AX and using **CWD** (Convert Word to Double word) instruction, the sign bit of the data in AX is extended to DX and thereby the word in AX is converted to signed double word in DX-AX.

If an attempt is made to divide by 0 or if the quotient is too large or two low to fit in AL or AX for 8 or 16 bit division respectively (i.e. either when the result is greater than +127 decimal in 8 bit division or +32767 decimal in 16 bit division or if the result is less than -128 decimal in 8 bit division or - 32767 decimal in 16 bit division), the 8086 automatically generate a type 0 interrupt. All flags are undefined after a DIV instruction.

**DAA:** Decimal Adjust AL after BCD addition

This instruction is used to get the result of adding two packed BCD numbers (two decimal digits are represented in 8 bits) to be a BCD number. The result of addition must be in AL for DAA to work correctly. If the lower nibble (4 bits) in AL is greater than 9 after addition or AF flag is set by the addition then the DAA will add 6 to the lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or the carry flag is set by the addition, then the DAA will add 60H to AL.

Examples:

Let        AL=0101 1000=58 BCD CL=0011 0101=35 BCD

Consider the execution of the following instructions: ADD AL, CL; AL=10001101=8DH and AF=0 after execution

DAA - Add 0110 (decimal 6) to AL since lower nibble in AL is greater than 9 AL=10010011= 93 BCD and CF=0

Therefore the result of addition is 93 BCD.

**DAS:** Decimal Adjust after BCD subtraction

DAS is used to get the result is in correct packed BCD form after subtracting two packed BCD numbers. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by subtraction then the DAS will subtract 6 from the lower nibble of AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS will subtract 60H from AL.

Examples:

Let AL=86 BCD=1000 0110 CH=57 BCD=0101 0111

Consider the execution of the following instructions:

SUB AL, CH; AL=0010 1111=2FH and CF=0 after execution

DAS; Lower nibble of result is 1111, so DAS subtracts 06H from AL to make AL=0010 1001=29 BCD and CF=0 to indicate there is no borrow.

The result is 29 BCD. AAA:

**AAA** (ASCII Adjust after Addition) instruction must always follow the addition of two unpacked BCD operands in AL. When AAA is executed, the content of AL is changed to a valid unpacked BCD number and clears the top 4 bits of AL. The CF is set and AH is incremented if a decimal carry out from AL is generated.

Example:

Let AL=05 decimal=0000 0101 BH=06 decimal=0000 0100 AH=00H

Consider the execution of the following instructions:

ADD AL, BH ; AL=0BH=11 decimal and CF=0 AAA ; AL=01 and AH=01 and CF=1

Since 05+06=11(decimal)=0101 H stored in AX in unpacked BCD form. When this result is to be sent to the printer, the ASCII code of each decimal digit is easily formed by adding 30H to each byte.

**AAS:** ASCII Adjust after Subtraction

This instruction always follows the subtraction of one unpacked BCD operand from another unpacked BCD operand in AL. It changes the content of AL to a valid unpacked BCD number and clears the top 4 bits of AL. The CF is set and AH is decremented if a decimal carry occurred.

Example:

Let AL=09 BCD=0000 1001 CL=05 BCD =0000 0101 AH=00H

Consider the execution of the following instructions:

SUB AL, CL; AL=04 BCD

AAS ; AL=04 BCD and CF=0

; AH=00H

AAA and AAS affect AF and CF flags and OF, PF, SF and ZF are left undefined. Another salient feature of the above two instructions are that the input data used in the addition or subtraction can be even in ASCII form of the unpacked decimal number and still we get the result in ordinary unpacked decimal number form and by adding 30H to the result , again we get ASCII form of the result.

**AAD:** The ASCII adjust AX before Division instruction modifies the dividend in AH and AL, to prepare for the division of two valid unpacked BCD operands. After the execution of AAD, AH will be cleared and AL will contain the binary equivalent of the original unpacked two digit numbers. Initially AH contains the most significant unpacked digit and AL contains the least significant unpacked digit.

Example: To perform the operation 32 decimal / 08 decimal Let AH=03H; upper decimal digit in the dividend

AL=02H; lower decimal digit in the dividend CL=08H; divisor

Consider the execution of the following instructions:

AAD; AX=0020H (binary equivalent of 32 decimal in 16 bit form)

DIV CL; Divide AX by CL; AL will contain the quotient and AH will contain the remainder. AAD affects PF, SF and ZF flags. AF, CF and OF are undefined after execution of AAD.

**AAM:** The ASCII Adjust AX after Multiplication instruction corrects the value of a multiplication of two valid unpacked decimal numbers. The higher order digit is placed in AH and the low order digit in AL.

Example:

Let AL=05 decimal

CL=09 decimal

Consider the execution of the following instructions:

MUL CH; AX=002DH=45 decimal

AAM; AH=04 and AL=05 (unpacked BCD form decimal number of 45)

OR AX, 3030H; To get ASCII code of the result in AH and AL (Note: this instruction is used only when it is needed). AAM affects flags same as that of AAD.

**AND:** The AND instruction perform logical AND operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of AND instruction is

AND destination, Source

The rules for destination and source for AND instruction are same as that of ADD instruction. CF and OF are both 0 after AND. PF, SF and ZF are updated after execution of AND instruction. AF is undefined. PF has meaning only for ANDing 8-bit operand.

**OR:** The OR instruction perform logical OR operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of OR instruction is

OR destination, Source

The rules for the source and destination and the way flags are affected for OR instruction are same as that of AND instruction.

**XOR:** The XOR instruction performs logical XOR operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of XOR instruction is

XOR destination, source

The rules for the same source and destination and the way flags are affected for XOR instruction are same as that of AND instruction.

**NOT:** The Not instruction inverts each bit (forms the 1’s complement) of the byte or word at the specified destination. The destination can be a register or a memory location. No flags are affected by the NOT instruction.

Example:

NOT AL; Take 1’s complement of AL NOT BX; Take 1’s complement of BX

NOT [SI]; Take 1’s complement of data in memory at [SI]

**TEST:** This instruction ANDs the content of a source byte or word with the content of the specified destination byte or word respectively. Flags are updated, but neither operand is changed. The TEST instruction is often used to set flags before a conditional jump instruction. The general form of TEST instruction is

TEST destination, source

The rules for the source and destination are same as that of AND instruction and the way flag are affected is also same as that of AND instruction.

Example:

Let AL=0111 1111 =7FH

TEST AL, 80H; AL=7FH (unchanged)

ZF=1 since (AL) AND (80H)=00H; SF=0; PF=1

**Flag manipulations instructions**

| • **Mnemonics** | • **Function** |
|---|---|
| • LAHF | • Load low byte of flag register into AH |
| • SAHF | • Store AH into the low byte of flag register |
| • PUSHF | • Push flag register’s content into stack |
| • POPF | • Pop top word of stack into flag register |
| • CMC | • Complement carry flag (CF = complement of CF) |
| • CLC | • Clear carry flag (CF= 0) |

| | |
|---|---|
| • STC | • Set carry flag (CF= 1) |
| • CLD | • Clear direction flag (DF= 0) |
| • STD | • Set direction flag (DF= 1) |
| • CLI | • Clear interrupt flag (IF= 0) |
| • STI | • Set interrupt flag (IF=1) |

**Control transfer instructions:** (Unconditional and Conditional jump instructions)

| **Mnemonics** | **Descripton** |
|---|---|
| JMP addr | Jump unconditionally to addr |
| CALL addr | Call procedure or subroutine starting at addr |
| RET | Return from procedure or subroutine |
| JA addr | Jump if above to addr (jump if CF = ZF =0) |
| JAE addr | Jump if above or equal to addr (jump if CF=0) |
| JB addr | Jump if below to addr (jump if CF=1) |
| JBE addr | Jump if below or equal to addr (Jump if CF =1 or ZF = 1) |
| JC addr | jump if carry to addr (jump if CF = 1) |
| JCXZ addr | Jump if CX = 0 |
| JE addr | Jump if equal (jump if ZF = 1) |

| **JL addr** | **Jump if not less (Jump if SF=OF)** |
|---|---|
| JNLE addr | Jump if not less or equal (Jump if ZF = 0 and SF = OF) |
| JNO addr | Jump if not overflow (Jump if OF = 0) |
| JNP addr | Jump if not parity (Jump if PF = 0) |
| JNS addr | Jump if not sign (jump if SF=0) |
| JNZ addr | Jump if not zero (jump if ZF=0) |

| | |
|---|---|
| JO addr | Jump if overflow (jump if OF=1) |
| JP addr | Jump if parity (jump if PF=1) |
| JPE addr | Jump if parity even (jump if PF=1) |
| JPO addr | Jump if parity odd (jump if PF=0) |
| JS addr | Jump if sign (jump if SF=1) |
| JZ addr | Jump if zero (jump if ZF=1) |

**Shift and Rotate instructions:** The Shift instructions perform logical left shift and right shift, and arithmetic left shift and right shift operation. The arithmetic left shift (SAL) and logical left shift (SHL) have the same function.

**SAL/SHL:** The general format of SAL/SHL instruction is SAL/SHL Destination, Count

The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB position, a 0 is put in the LSB position. The MSB will be shifted into carry flag (CF).

If the number of shifts to be done is 1 then it can be directly specified in the instruction with count equal to 1. For shifts of more than one bit position, the desired number of shifts is loaded into the CL register and CL is put in the count position of the instruction. CF, SF and ZF are affected according to the result. PF has meaning only when AL is used as destination. SAL instruction can be used to multiply an unsigned number by a power of 2. Doing one bit or two bits left shift of a number multiplies the number by 2 or 4 respectively and so on.

Examples:

SAL AX,1; Shift left the content of AX by 1 bit SAL BL,1; Shift left the content of BL by 1 bit

SAL BYTE PTR [SI],1; Shift left the byte content of memory at [SI] by 1 bit

**SAR:** The general format of SAR instruction is

SAR Destination, Count

The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position (i.e. the sign bit is copied into the MSB). The LSB will be shifted into carry flag (CF).

The rules for the Count in the instruction are same as that of the SAL instruction. CF, SF and ZF are affected according to the result. PF has meaning only when AL is used as destination.

**SHR:**

The general format of SHR instruction is SHR Destination, Count

The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in the MSB position. The LSB will be shifted into carry flag (CF).

The rules for the Count in the instruction are same as that of the SHL instruction. CF, SF and ZF are affected according to the result. PF has meaning only when 8 bit destination is used.

**ROR:** This instruction rotates all the bits of the specified byte or word by some number of bit positions to the right. The operation done when ROR is executed is shown below.

The general format of ROR instruction is ROR Destination, Count

The data bit moved out of the LSB is also copied into CF. ROR affects only CF and OF. For single bit rotate, OF will be 1 after ROR, if the MSB is changed by the rotate. ROR is used to swap nibbles in a byte or to swap two bytes within a word. It can also be used to rotate a bit in a byte or word into CF, where it can be checked and acted upon by the JC and JNC instruction. CF will contain the bit most recently rotated out of LSB in the case of multiple bit rotate.

The rules for count are same as that of the shift instruction which is discussed above. Examples:

ROR CH, 1 : rotate right byte in CH by one bit position.

ROR BX, CL : Rotate right word in BX by number of bit positions given by CL.

**ROL:** ROL rotates all the bits in a byte or word in the destination to the left either by 1 bit position and more than 1 bit positions using CL as shown below:

**RCR:** Rotate the byte or word in the destination right through carry flag (CF) either by one bit position or the number of bit positions given by the CL as shown below.The flag are affected by similar to ROR.

**RCL:** Rotate the byte or word in the destination left through carry flag (CF) either by one bit position or the number of bit positions given by the CL as shown below.The flags are affected similar to ROL.

**String Instructions**

The string instructions operate on element of strings of bytes or word. Registers SI and DI contain the offset address within a segment of an element (Byte or Word) in the source string and the destination string respectively. The source string is in the data segment at the offset address given by SI and destination string is in the extra segment at the offset address given by DI.After each string operation, SI and/or DI are automatically incremented or decremented by 1 or 2 (for byte or word operation) according to the D flag in the flag register. If D=0, SI and/or DI are automatically incremented and if D=1, SI and/or DI are automatically decremented.

| MNEMONICS | FUNCTION |
|---|---|
| MOVSB | Move string byte from DS:[SI] to ES:[DI] |
| MOVSW | Move string word from DS:[SI] to ES:[DI] |
| CMPSB | Compare string byte (Done by subtracting byte at ES:[DI] from the byte at DS:[SI affected and the content of bytes compared is unaffected. |
| CMPSW | Compare string word (Done by subtracting word at ES:[DI] from the word at DS:[SI affected and the content of words compared is unaffected. |
| LODSB | Load string byte at DS:[SI] into AL |
| LODSW | Load string word at DS:[SI] into AX |
| STOSB | Store string byte in AL at ES:[DI] |
| STOSW | Store string word in AX at ES:[DI] |
| SCASB | Compare string byte (Done by subtracting byte at ES:[DI] from the byte at ). Only flags the content of bytes compared is unaffected. |
| SCASW | Compare string word (Done by subtracting word at ES:[DI] from the byte at AX). Only and the content of words compared is unaffected. |
| REP | Decrement CX and Repeat the following string operation if CX ≠ 0. |
| REPE or REPZ | Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=1. |
| REPNE or REPNZ | Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=0. |

**Machine or processor control instructions**

**HLT:** The halt instruction stops the execution of all instructions and places the processor in a halt state. An interrupt or a reset signal will cause the processor to resume execution from the halt state.

**LOCK:** The lock instruction asserts for the processor an exclusive hold on the use of the system bus. It activates an external locking signal ( ) of the processor and it is placed as a prefix in front of the instruction for which a lock is to be asserted. The lock will function only with the XCHG, ADD, OR, ADC, SBB, AND, SUB, XOR, NOT, NEG, INC and DEC instructions, when they involve a memory operand. An undefined opcode trap interrupt will be generated if a LOCK prefix is used with any instruction not listed above.

**NOP:** No operation. This instruction is used to insert delay in software delay programs.

**ESC:** This instruction is used to pass instructions to a coprocessor such as the 8087, which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6- bit code embedded in the escape instruction.

As the 8086 fetches instruction bytes from memory the coprocessor also catches these bytes from the data bus and puts them in a queue. However the coprocessor treats all the normal 8086 instructions as NOP instruction. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6- bit code specified in the instruction.

**WAIT:** When this instruction is executed, the 8086 checks the status of its input pin and if the input is high, it enters an idle condition in which it does not do any processing. The 8086 will remain in this state until the 8086’s input pin is made low or an interrupt signal is received on the INTR or NMI pins. If a valid interrupt occurs while the 8086 is in this idle state, it will return to the idle state after the interrupt service routine is executed. WAIT instruction does not affect flags. It is used to synchronize 8086 with external hardware such as 8087 coprocessor.

## ASSEMBLERDIRECTIVES

An *assembler directive* is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process. Or a statement in an assembly-language program that gives instructions to the assembler. They are describedbelow.

---

**ASSUME**

**DB** - Defined Byte.

**DD** - Defined Double Word

**DQ** - Defined Quad Word

**DT** - Define Ten Bytes

**DW** - Define Word

-----------------------------------------------------------------------------------------------------------------------------------

**ASSUME Directive**:

The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment. The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

**Example:**

ASUMECS:CODE ; this tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

ASUMEDS:DATA ; this tells the assembler that for any instruction which refers to a data in the data segment, data will found in the logical segmentDATA.

**DB:** DB directive is used to declare a byte-type variable or to store a byte in memory location.

**Example:**

1. PRICE DB 49h,98h,29h ; Declare an array of 3 bytes, named as PRICE andinitialize.
2. NAME DB'ABCDEF' ; Declare an array of 6 bytes and initialize with ASCII code forletters
3. TEMP DB100 DUP(?) ;Set100bytesofstorageinmemoryandgiveitthenameasTEMP,

   ;but leave the 100 bytes uninitialized. Program instructions will load

   ;values into these locations.

**DW**: The DW directive is used to define a variable of type word or to reserve storage location of type word in memory.

**Example:**

MULTIPLIERDW437Ah ; this declares a variable of type word and named it asMULTIPLIER.

;This variable is initialized with the value 437Ah when it is loaded into memory to run.

EXP1 DW 1234h,3456h,5678h; this declares an array of 3 words and initialized withspecified values.

STOR1 DW100DUP(0) ; Reserve an array of 100 words of memory and initialize all words with 0000.Array is named asSTOR1.

-----------------------------------------------------------------------------------------------------------------------------------

**END** - End Program

**ENDP** - End Procedure

**ENDS** - End Segment

**EQU** - Equate

**EVEN** - Align on Even Memory Address

**EXTRN**

---------------------------------------------------------------------------------------------------------------------------------------

**END**: END directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an END directive. Carriage return is required after the END directive.

**ENDP**: ENDP directive is used along with the name of the procedure to indicate the end of a procedure to the assembler

**Example:**

```
SQUARE_NUMPROCE          ; It start the procedure Some steps to find the square root of a number
SQUARE_NUMENDP            ; Hear it is the End for theprocedure
```

**ENDS** - This ENDS directive is used with name of the segment to indicate the end of that logic segment.

**Example:**

```
CODESEGMENT               ; Hear it Start the logic segment containingcode
                          ; Some instructions statements to perform the logical operation
CODEENDS                  ; End of segment named asCODE
```

**EQU**: This EQU directive is used to give a name to some value or to a symbol. Each time the assembler finds the name in the program, it will replace the name with the value or symbol you given to that name.

**Example:**

```
FACTOREQU03H         ; you has to write this statement at the starting of your program and later in the
                     program you can use this asfollows
ADDAL, FACTOR        ; When it codes this instruction the assembler will code it as ADDAL,03H
                     ; The advantage of using EQU in this manner is, if FACTOR is used many no of
                     times in a program and you want to change the value, all you had to do is change
                     the EQU statement at beginning, it will changes the rest of all.
```

**EVEN**: This **EVEN** directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word is at even address 8086 can read a memory in 1 buscycle.

If the word starts at an odd address, the 8086 will take 2 bus cycles to get the data. A series of words can be read much more quickly if they are at even address. When EVEN is used the location counter will simply incremented to next address and NOP instruction is inserted in that incremented location.

**Example**:

DATA1 SEGMENT; Location counter will point to 0009 after assembler reads nextstatement SALES DB 9DUP (?)  ; declare an array of 9bytes

EVEN  ; increment location counter to000AH

RECORD DW 100DUP (0)  ; Array of 100 words will start from an even address for quicker read DATA1ENDS

--------------------------------------------------------------------------------------------------------------------------------------------

**GROUP -** Group Related Segments

**LABEL**

**NAME**

**OFFSET**

**ORG** - Originate

--------------------------------------------------------------------------------------------------------------------------------------------

**GROUP** - The GROUP directive is used to group the logical segments named after the directive into one logical group segment.

**LABEL**: LABEL is used to assign a name to the current memory location. If the memory location is within the same segment then NEAR label is used and if the memory location is available in other than the current segment then FAR label is used.

**Ex**: REPEAT LABEL NEAR

CALCULATE LABEL FAR

**NAME:** It is used to assign NAME to the assembly language program.

**OFFSET:** Offset directive is used to determine the offset address of the label.

**Ex:** Mov BX, OFFSET UP –offset address of label UP is moved to BX.

**ORG:** The Origin directive indicates the start of the memory locations for segments (CODE,DATA,STACK)

**Ex:** ORG 1000H

**INCLUDE** - This INCLUDE directive is used to insert a block of source code from the named file into the current source module.

--------------------------------------------------------------------------------------------------------------------------------------------

**PROC** - Procedure

**PTR** - Pointer

**PUBLC**

**SEGMENT**

**SHORT**

**TYPE**

------------------------------------------------------------------------------------------------------------------------------------------

**PROC**: The PROC directive is used to identify the start of a procedure. The term near or far is used to specify the type of theprocedure.

**Example:**

SMART PROCFAR ; This identifies that the start of a procedure named as SMART and instructs the assembler that the procedure isfar.

SMART ENDP

This PROC is used with ENDP to indicate the break of the procedure.

**PTR**: This PTR operator is used to assign a specific type of a variable or to a label.

**Example**:

INC [BX] **;** this instruction will not know whether to increment the byte pointed to by BX or a word pointed to byBX.

INC BYTEPTR[BX] ; increment the byte ; pointed to byBX

This PTR operator can also be used to override the declared type of variable. If we want to access the a byte in anarray

WORDS DW 437Ah, 0B97h,
MOV AL, BYTE PTR WORDS

**PUBLIC/EXTRN** - The PUBLIC directive is used to instruct the assembler that a specified name or label will be accessed from other modules.

**Example:**

PUBLICDIVISOR,DIVIDEND ; these two variables are public so these are available to all modules. If an instruction in a module refers to a variable in another assembly module, we can access that module by declaring as EXTRNdirective.

**TYPE** - TYPE operator instructs the assembler to determine the type of a variable and determines the number of bytes specified to that variable.

**Example:**

Byte type variable – assembler will give a value 1
Word type variable – assembler will give a value 2

Double word type variable – assembler will give a value 4

ADD BX, TYPE WORD_ ARRAY; hear we want to increment BX to point to next word in an array of words.

**SHORT:** It is used to represent the BYTE displacement in branch instructions.

## DOS Function Calls

|  |  |
|---|---|
| **AH 00H** | : Terminate a Program |
| **AH 01H** | : Read theKeyboard |
| **AH 02H** | : Write to a Standard Output Device |
| **AH 08H** | : Read a Standard Input without Echo |
| **AH 09H** | : Display a CharacterString |
| **AH 0AH** | : Buffered keyboardInput |
| **INT21H** | : Call DOSFunction. |

## PROCEDURES AND MACROS:

### Procedures

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC

      ; here goes thecode

      ; of the procedure...

RET

name ENDP
```

**name** - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures. The RET instruction is used to return from procedure

### Macros

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it.

Macro definition:

```
name    MACRO [parameters,...]


<instructions>


ENDM
```

**Differences between Procedures and Macros**

| **PROCEDURES** | **MACROS** |
|---|---|
| Accessed by CALL and RET instructions during program execution. | Accessed during assembly when name given to macro is written as an instruction in the assembly program. |
| Machine code for instructions is put only once in the memory. | Machine code is generated for instructions each time a macro is called. |
| This as all machine code is defined only once so less memory is required. | This due to repeated generation of machine code requires more memory. |
| Parameters can be passed in register memory location or stack. | Parameters are passed as a part of the statement in which macro is called. |

## INTRODUCTION TO PROGRAMMING THE8086

**Programming Languages:** To run a program, a microcomputer must have the program stored in binary form in successive memory locations. There are three language levels that can be used to write a program for a microcomputer.

1. MachineLanguage
2. AssemblyLanguage
3. High-levelLanguages

## ASSEMBLY LANGUAGE PROGRAM DEVELOPMENTTOOLS

For all but the very simplest assembly language programs, you will probably want to use some type of microcomputer development system and *program development tools* to make your work easier. Most of the program development tools are programs which you run to perform some function on the program you are writing.

Program development tools are:

1. Editor
2. Assembler
3. Linker
4. Locator
5. Debugger
6. Emulator

**Editor:** An editor is a program which allows you to create a file containing the assembly language statements for your program. When you have typed in your entire program, you then save the file on a hard disk. This file is called source file. The next step is to process the source file with an assembler. If you are going to use the TASM or MASM assembler, you should give your source file name the extension .ASM.

**Assembler:** An assembler is programming tool which is used to translate the assembly language mnemonics for instructions to the corresponding binary codes. The assembler generates two files. The first file, called the *object file*, is given the extension .OBJ. The object file contains the binary codes for the instructions and information about the addresses of the instructions. After further processing the contents of this file will be loaded into memory and run. The second file generated by the assembler is called the *assembler list file* and is given the extension .LST.

**Linker:** The linker is program used to join several object files into one large object file. The linkers which come with the TASM or MASM assemblers produce link files with the .EXEextension.

**Locator:** A locator is a program used to assign the specific addresses of where the segments of object code are to be loaded into memory.

**Debugger:** If your program requires no external hardware or requires only hardware accessible directly from your microcomputer, then you can use debugger to run and debug your program. A debugger is a program which allows you to load your object code program into system memory, execute the program, and troubleshoot or' debug' it.

**Emulator:** Another way to run your program is with an emulator. An emulator is a mixture of hardware and software. It is usually used to test and debug the hardware and software of an external system.

## ASSEMBLY LANGUAGEPROGRAMS

### Simple programs

1. Write an ALP in 8086 to perform series addition of N 16-bitnumbers.

```
            ASSUME CS: CODE
            ORG 2000H

    CODE    SEGMENT

    START:  MOV SI,3000H

            MOV CL, [SI]

            INC SI

            MOV AX, [SI]

             DEC CL

    UP:     INC SI

            INC SI

            MOV BX, [SI]

             ADC AX, BX

            DEC CL

            JNZ UP

            INT 03H

    CODE    ENDS

            END
```

2. Write an ALP in 8086 to exchange a block of N bytes between source location and destination.

```
            ASSUME CS:CODE

            ORG 4000H
    CODE    SEGMENT

            MOV   SI,   2000H
            MOV   DI,   3000H
            MOV CL, [SI]

    UP:     INCSI

            MOV AL, [SI]

            MOV   BL,   [DI]
            XCHG   AL,   BL
            MOV   [SI],   AL
            MOV [DI], BL

            INC DI

            LOOP UP
            INT 03H
```

```
CODE    ENDS

        END
```

3. Write an ALP in 8086 to count no. of even and odd numbers from the given array.

```
        ASSUME CS:CODE

        ORG 2000H
CODE    SEGMENT

        MOV SI, 3000H

        MOV CL, [SI]

        MOV BX, 0000H

        MOV DX, 0000H

        MOV AX, 0000H

UP:     INCSI

        MOV   AL,   [SI]
        ROR AL, 01H

        JC ODD

        INC BX

        JMP DOWN
ODD:    INC DX

        DOWN: LOOP UP

        INT 03H

        CODE ENDS

        END
```

4. Write an ALP in 8086 to check whether the given string is palindrome ornot.

```
        ASSUME CS: CODE
        ORG 5000H

CODE    SEGMENT

        MOV CX, 0000H
        MOV SI, 3000H
        MOV CL, [SI]

        MOV DI, SI

        ADD DI, CX
        MOV AL, CL
        MOV BL, 02H
        DIV BL
        MOV CL, AL
        INC SI
UP:     CMPSB
        JNE EXIT
         INC SI
        DEC SI
        LOOPUP
        MOV AX, FFFFH
        INT 03H
EXIT:   MOV AX, 0000H
        INT 03H
        CODENDS
        END
```

5. Write an ALP in 8086 to convert ASCII to Hexadecimal number.

**<u>ASCIITOHEX</u>**

```
DATA SEGMENT
A DB 41H
 R DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START:          MOVAX,DATA
                MOV DS,AX
                MOV AL,A
                SUB AL,30H
                CMP AL,39H
                JBE L1
```

```
                        SUB AL,7H

                        L1: MOV R,AL

                        INT 3H

          CODE ENDS

          END START

          END
```

6. Write an ALP in 8086 to convert Hexadecimal number to ASCII.

**HEX TO ASCII**

```
     DATA SEGMENT

     A DB 08H

      C DB ?

     DATA ENDS

     CODE SEGMENT

     ASSUME CS: CODE, DS: DATA

     START: MOV AX,DATA

            MOV DS,AX

            MOV AL,A

            ADD AL,30H

            CMP AL,39H

            JBE L1

            ADD AL,7H

            L1:    MOVC,AL

            INT 3H

            CODE ENDS
     END START

            END
```

7. Write an ALP in 8086 to find the factorial of a given number.

**FACTORIAL**

```
       DATA SEGMENT
```

```
 ORG 2000H

 FIRST DW 3H

 SEC DW 1H

DATA ENDS

CODE SEGMENT

ASSUME CS:CODE,DS:DATA

START: MOV AX,DATA

           MOV DS,AX

           MOV AX,SEC

           MOV CX,FIRST

     L1: MUL CX

           DEC CX

           JCXZ L2

           JMP L1

      L2: INT 3H

      CODE ENDS

      END START

      END
```

8. Write an ALP in 8086 to find the fibonocci series.

**<u>FIBONOCCI</u>**

```
DATA SEGMENT

ORG 2000H

FIRST DW 0H

SEC DW 01H

THIRD DW 50H

RESULT DW ?

DATA ENDS

CODE SEGMENT
```

```
ASSUME CS: CODE, DS:DATA
START:      MOVAX,DATA
             MOV DS,AX
             MOV SI,OFFSET RESULT
             MOV AX,FIRST
             MOV BX,SEC
             MOV CX,THIRD
             MOV [SI],AX
                  L1: INCSI
                      INC SI
             MOV [SI],BX
             ADD AX,BX
             XCHG AX,BX
             CMP BX,CX
              INT3H
  CODE ENDS
  END START
  END
```

9. Write an ALP in 8086 to find the greater number from the given numbers.

**GREATER**

```
DATA SEGMENT
ORG 2000H
FIRST DW 5H,2H,3H,1H,4H
COUNT EQU (($-FIRST)/2)-1
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START: MOV AX,DATA
```

```
                MOV DS,AX

                MOV CX,COUNT

                MOV SI,OFFSET FIRST

                 MOV AX,[SI]

          L2: INC SI

                    INC SI

                    MOV BX,[SI]

                    CMP AX,BX

                    JGEL1

                    XCHG AX,BX

                    JMPL1

          L1: DEC CX

          JCXZ L4

          JMP L2

           L4: INT 3H

           CODEENDS

ENDSTART

END
```

## INTRODUCTION

**What is a Microcontroller?**

A Microcontroller is a programmable digital processor with necessary peripherals. Both microcontrollers and microprocessors are complex sequential digital circuits meant to carry out job according to the program / instructions. Sometimes analog input/output interface makes a part of microcontroller circuit of mixed mode (both analog and digital nature). A microcontroller can be compared to a Swiss knife with multiple functions incorporated in the same IC.

**Fig. A Microcontroller compared with a Swiss knife**

**Microcontrollers Vs Microprocessors**

1. A microprocessor requires an external memory for program/data storage. Instruction execution requires movement of data from the external memory to the microprocessor or vice versa. Usually, microprocessors have good computing power and they have higher clock speed to facilitate faster computation.
2. A microcontroller has required on-chip memory with associated peripherals. A microcontroller can be thought of a microprocessor with inbuilt peripherals.
3. A microcontroller does not require much additional interfacing ICs for operation and it functions as a standalone system. The operation of a microcontroller is multipurpose, just like a Swiss knife.
4. Microcontrollers are also called embedded controllers. A microcontroller clock speed is limited only to a few tens of MHz. Microcontrollers are numerous and many of them are application specific.

**Development/Classification of microcontrollers**

Microcontrollers have gone through a silent evolution (invisible). The evolution can be rightly termed as silent as the impact or application of a microcontroller is not well known to a common user, although microcontroller technology has undergone significant change since early 1970's. Development of some popular microcontrollers is given as follows.

| | | |
|---|---|---|
| Intel 4004 | 4 bit (2300 PMOS trans, 108 kHz) | 1971 |
| Intel 8048 | 8 bit | 1976 |
| Intel 8031 | 8 bit (ROM-less) | . |
| Intel 8051 | 8 bit (Mask ROM) | 1980 |
| Microchip PIC16C64 | 8 bit | 1985 |

| | | |
|---|---|---|
| Motorola 68HC11 | 8 bit (on chip ADC) | . |
| Intel 80C196 | 16 bit | 1982 |
| Atmel AT89C51 | 8 bit (Flash memory) | . |
| Microchip PIC 16F877 | 8 bit (Flash memory + ADC) | . |

**Development of microprocessors**

Microprocessors have undergone significant evolution over the past four decades. This development is clearly perceptible to a common user, especially, in terms of phenomenal growth in capabilities of personal computers. Development of some of the microprocessors can be given as follows.

| | | |
|---|---|---|
| Intel 4004 | 4 bit (2300 PMOS transistors) | 1971 |
| Intel8080<br>8085 | 8-bit(NMOS)<br>8 bit | 1974 |
| Intel8088<br>8086 | 16-bit<br>16 bit | 1978 |
| Intel80186<br>80286 | 16-bit<br>16 bit | 1982 |
| Intel 80386 | 32 bit (275000 transistors) | 1985 |
| Intel80486SX<br>DX | 32-bit<br>32 bit (built in floating point unit) | 1989 |
| Intel 80586 I<br>MMX<br>Celeron II<br>III<br>IV | 64 bit | 1993<br>1997<br>1999<br>2000 |
| Z-80 (Zilog) | 8 bit | 1976 |
| Motorola Power PC 601<br>602<br>603 | 32-bit | 1993<br><br>1995 |

We use more number of microcontrollers compared to microprocessors. Microprocessors are primarily used for computational purpose, whereas microcontrollers find wide application in devices needing real time processing/control. Applications of microcontrollers are numerous. Starting from domestic applications such as in washing machines, TVs, air conditioners, microcontrollers are used in automobiles, process control industries, cell phones, electrical drives, and robotics and in space applications.

**Microcontroller Chips**

Broad Classification of different microcontroller chips could be as follows:

- Embedded (Self -Contained) 8 - bit Microcontroller
- 16 to 32 Microcontrollers
- Digital Signal Processors

**Features of Modern Microcontrollers**

- Built-in Monitor Program
- Built-in Program Memory
- Interrupts
- Analog I/O
- Serial I/O

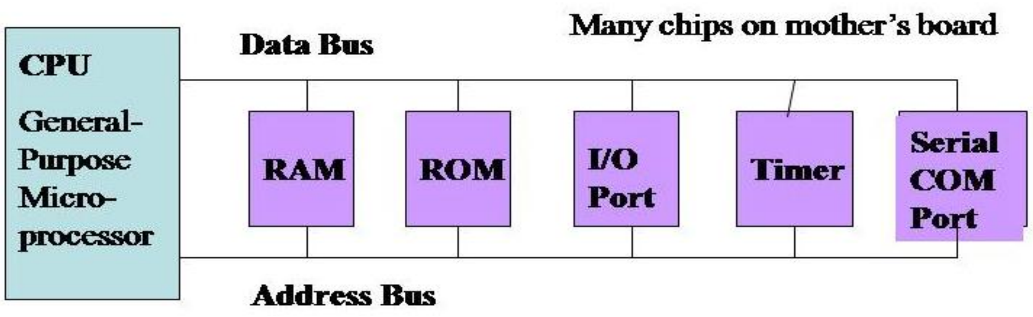
- Facility to Interface External Memory
- Timers

## MICROPROCESSOR VS. MICROCONTROLLER

**Microprocessor:**

General-purpose microprocessor

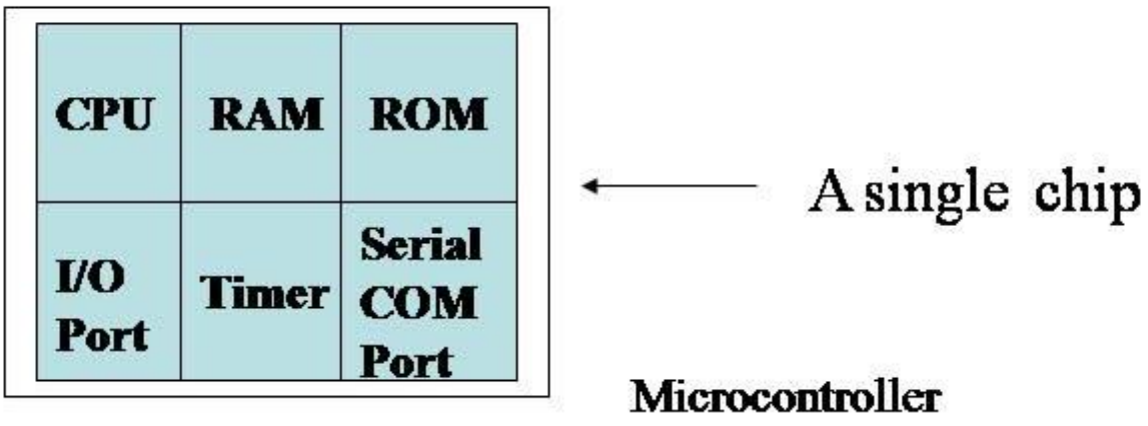
- CPU is stand-alone, RAM, ROM, I/O, timer are separate
- Designer can decide on the amount of ROM, RAM and I/O ports.
- expansive
- versatility
- general-purpose
- Example： Intel’s x86, Motorola’s 680x0



General-Purpose Microprocessor System

**Microcontroller :**

- CPU, RAM, ROM, I/O and timer are all on a single chip
- fix amount of on-chip ROM, RAM, I/O ports
- for applications in which cost, power and space are critical
- single-purpose
- Example: Motorola’s 6811, Intel’s 8051, Zilog’s Z8 and PIC 16X



**Advantages over microprocessor:**

- Cost is lower
- Standalone mp never used – memory, I/O, clock necessary
- For microprocessor- large size PCB
- Large PCB- more effort and cost
- Big physical size
- More difficult to trouble shoot mp based
- A microcontroller is a microprocessor with integrated peripherals.

**Advantages of microcontroller:**

- Low cost
- Small size of product
- Easy to troubleshoot and maintain
- More reliable
- Additional memory, I/o can also be added
- Software security feature
- All features available with 40 pins.
- Useful for small dedicated applications and not for larger system designs which may require many more I/O ports.
- Mostly used to implement small control functions.

## INTERNAL STRUCTURE OF A MICROCONTROLLER

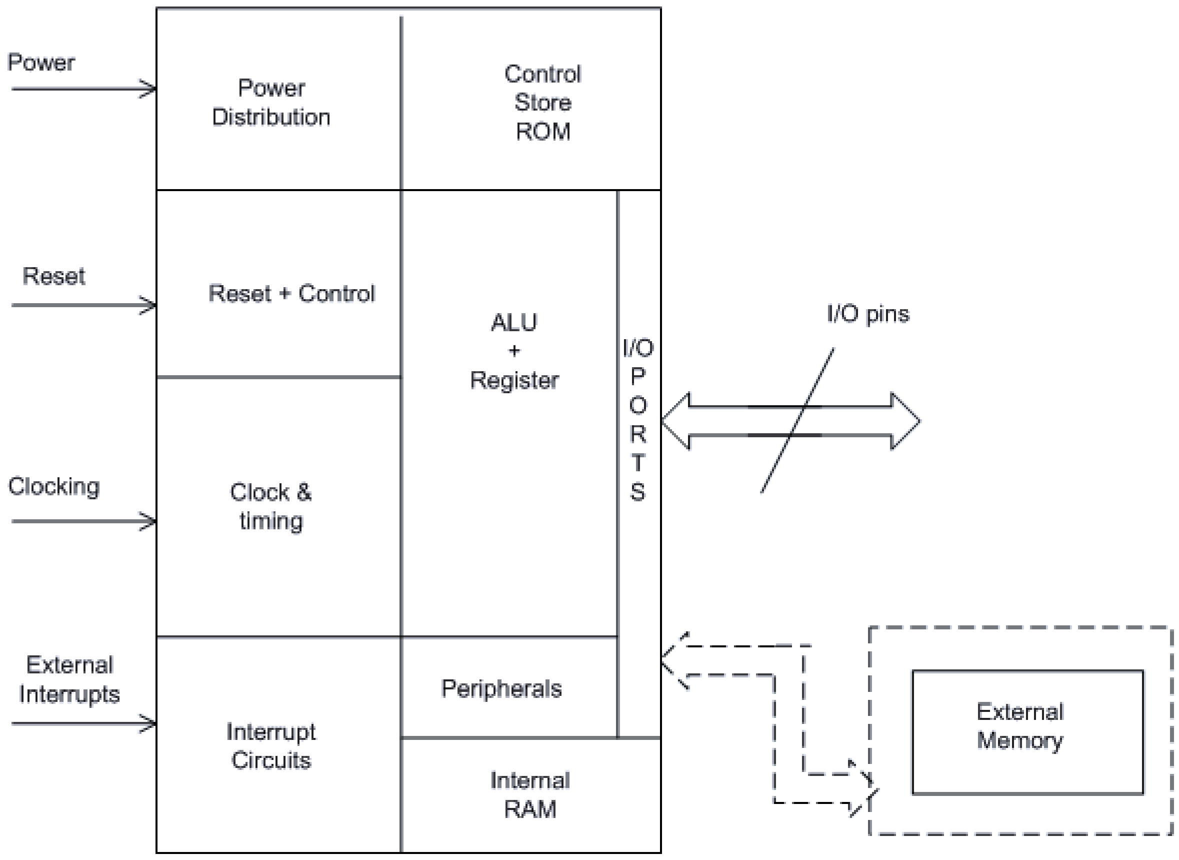


**Fig. Internal Structure of a Microcontroller**

At times, a microcontroller can have external memory also (if there is no internal memory or extra memory interface is required). Early microcontrollers were manufactured using bipolar or NMOS technologies. Most modern microcontrollers are manufactured with CMOS technology, which leads to reduction in size and power loss. Current drawn by the IC is also reduced considerably from 10mA to a few micro Amperes in sleep mode (for a microcontroller running typically at a clock speed of 20MHz).

### Harvard vs. Princeton Architecture

Many years ago, in the late 1940's, the US Government asked Harvard and Princeton universities to come up with a computer architecture to be used in computing distances of Naval artillery shell for defense applications. Princeton suggested computer architecture with a single memory interface. It is also known as Von Neumann architecture after the name of the chief scientist of the project in Princeton University John Von Neumann (1903 - 1957 Born in Budapest, Hungary).

Harvard suggested a computer with two different memory interfaces, one for the data / variables and the other for program / instructions. Although Princeton architecture was accepted for simplicity and ease of implementation, Harvard architecture became popular later, due to the parallelism of instruction execution.

**Princeton Architecture  (Single memory interface)**

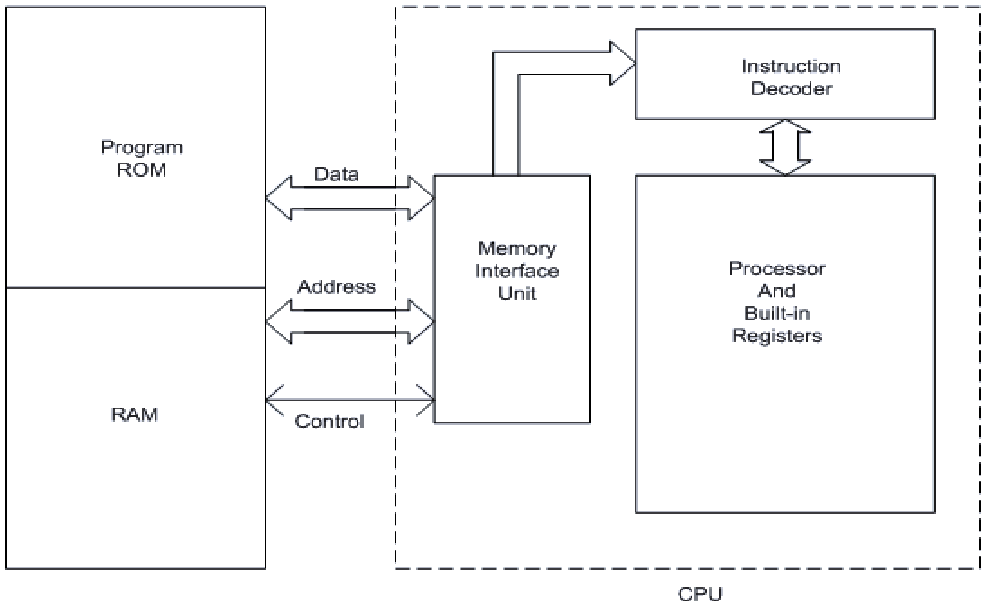


**Fig.  Princeton Architecture**

Example : An instruction "Read a data byte from memory and store it in the accumulator" is executed as follows: -

Cycle 1 - Read Instruction
Cycle 2 - Read Data out of RAM and put into Accumulator

**Harvard Architecture   (Separate Program and Data Memory interfaces)**

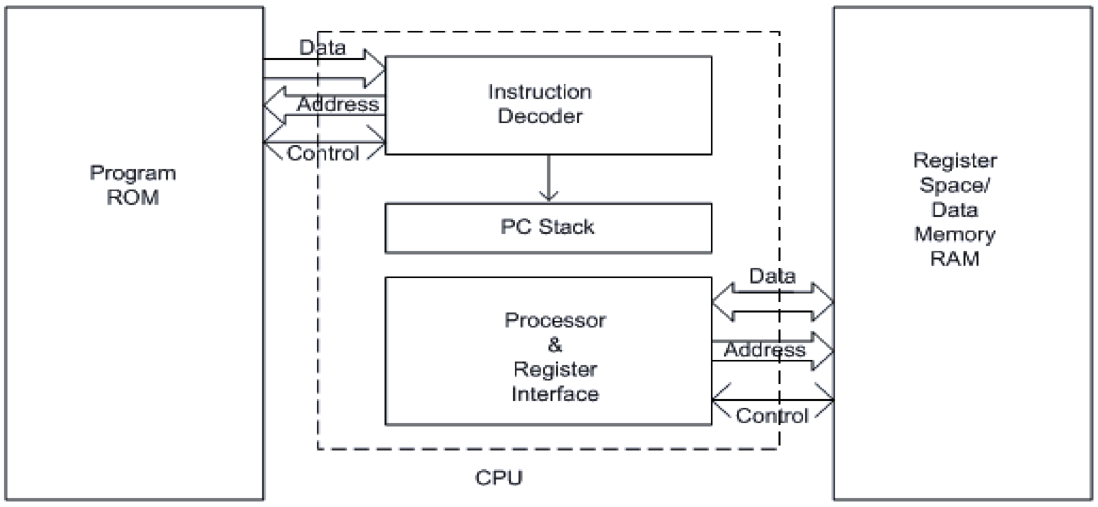


**Fig. 2.3     Harvard Architecture**

The same instruction (as shown under Princeton Architecture) would be executed as follows:

Cycle 1
- Complete previous instruction
- Read the "Move Data to Accumulator" instruction

Cycle 2
- Execute "Move Data to Accumulator" instruction
- Read next instruction

Hence each instruction is effectively executed in one instruction cycle, except for the ones that modify the content of the program counter. For example, the "jump" (or call) instructions takes 2 cycles. Thus, due to parallelism, Harvard architecture executes more instructions in a given time compared to Princeton Architecture.

**Some of the microcontrollers of 8051 family are given as follows:**

| **DEVICE** | **ON-CHIP DATA MEMORY (bytes)** | **ON-CHIP PROGRAM MEMORY (bytes)** | **16-BIT TIMER/COUNTER** | **NO. OF VECTORED INTERUPTS** | **FULL DUPLEX I/O** |
|---|---|---|---|---|---|
| 8031 | 128 | None | 2 | 5 | 1 |
| 8032 | 256 | none | 2 | 6 | 1 |
| 8051 | 128 | 4k ROM | 2 | 5 | 1 |
| 8052 | 256 | 8k ROM | 3 | 6 | 1 |
| 8751 | 128 | 4k EPROM | 2 | 5 | 1 |
| 8752 | 256 | 8k EPROM | 3 | 6 | 1 |
| AT89C51 | 128 | 4k Flash Memory | 2 | 5 | 1 |
| AT89C52 | 256 | 8k Flash memory | 3 | 6 | 1 |

## INTEL 8051 MICROCONTROLLER

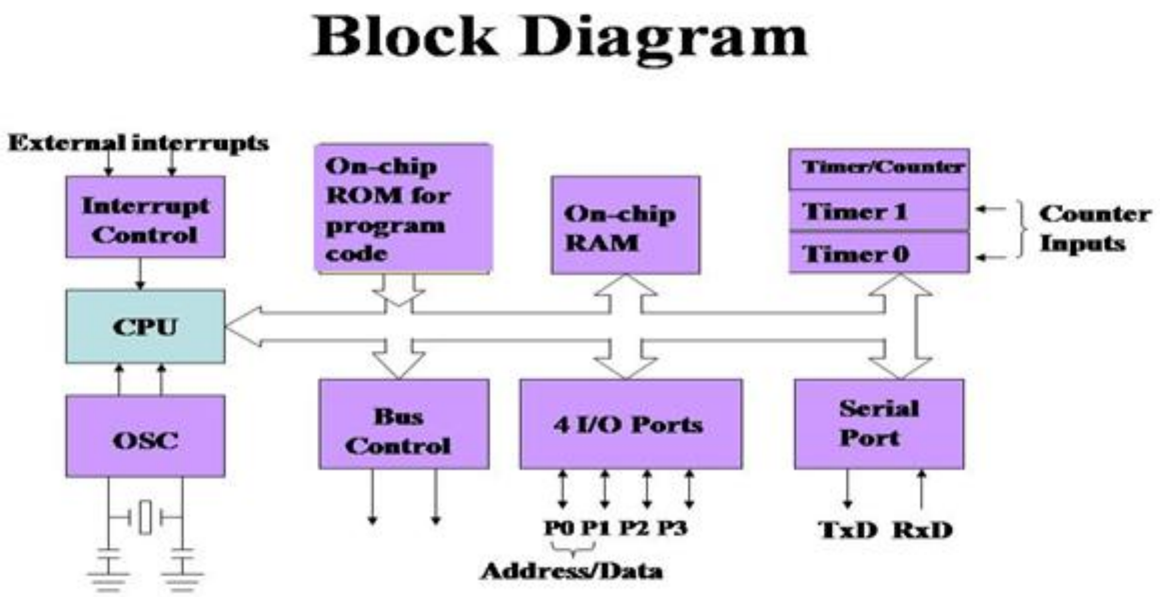
**Introduction:**
8051 employs Harvard architecture. It has some peripherals such as 32 bit digital I/O, Timers and Serial I/O. The basic architecture of 8051 is given in fig below

**Features:**

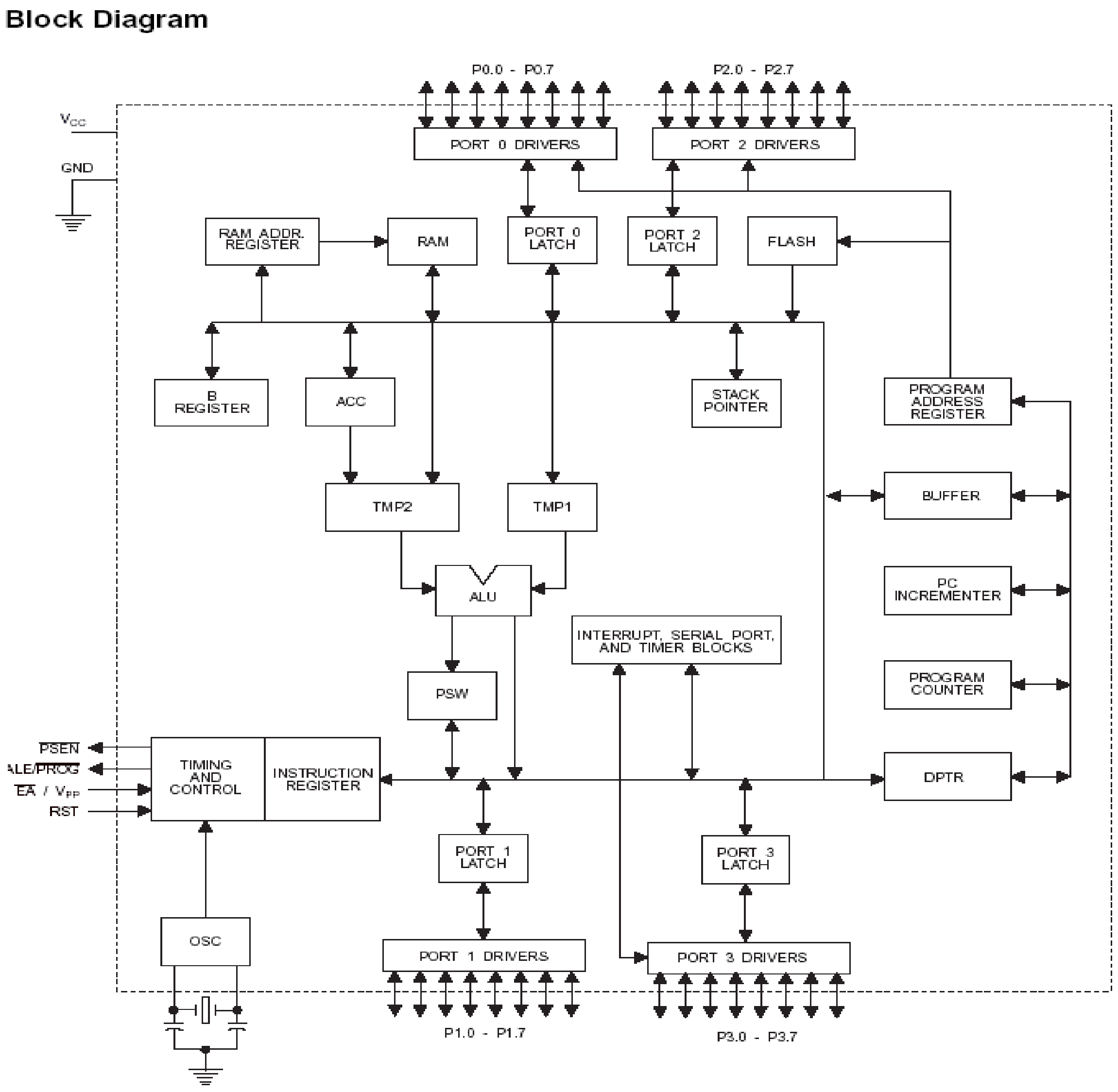
Various features of 8051 microcontroller are given as follows.

- 8-bit CPU
- 16-bit Program Counter
- 8-bit Processor Status Word (PSW)
- 8-bit Stack Pointer
- Internal RAM of 128bytes
- On chip ROM is 4KB
- Special Function Registers (SFRs) of 128 bytes
- 32 I/O pins arranged as four 8-bit ports (P0 - P3)
- Two 16-bit timer/counters : T0 and T1
- Two external and three internal vectored interrupts
- One full duplex serial I/O

**Basic Block Diagram:**



## Internal Architecture of 8051 Microcontroller



**Block Diagram Description:**

***Accumulator (Acc):***

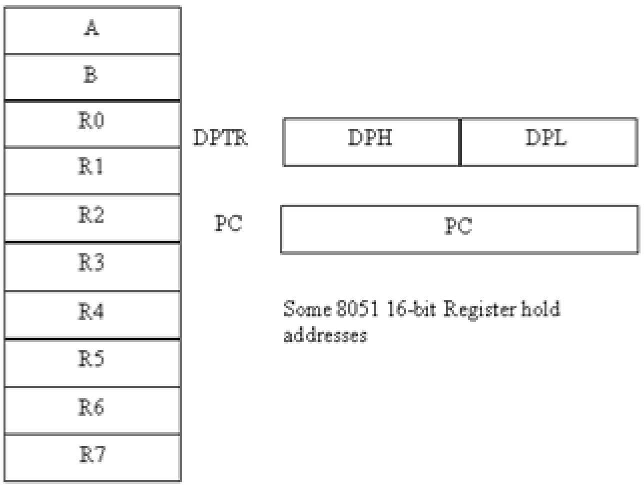
- Operand register
- Implicit or specified in the instruction
- Has an address in on chip SFR bank

***B Register:*** *Used* to store one of the operands for multiplication and division, otherwise, scratch pad considered as a SFR.
***Program Status Word (PSW):*** Set of flags contains status information.

***Stack Pointer (SP):*** 8 bit wide register. Incremented before data is stored on to the stack using PUSH or CALL instructions. Stack defined anywhere on the 128 byte RAM
***Data Pointer (DPTR):*** *1*6 bit register contains DPH and DPL Pointer to external RAM address. DPH and DPL allotted separate addresses in SFR bank
***Port 0 To 3 Latches & Drivers:*** Each I/O port allotted a latch and a driver Latches allotted address in SFR. User can communicate via these ports P0, P1, P2, and P3.
***Serial Data Buffer:*** *I*nternally had TWO independent registers, TRANSMIT buffer (parallel in serial out – PISO) and RECEIVE buffer (serial in parallel out –SIPO) identified by SBUF and allotted an address in SFR.

***Timer Registers:*** for Timer0 (16 bit register – TL0 & TH0) and for Timer1 (16 bit register – TL1 & TH1) four addresses allotted in SFR
***Control Registers:*** Control registers are IP, IE, TMOD, TCON, SCON, and PCON. These registers contain control and status information for interrupts, timers/counters and serial port. Allotted separate address in SFR.
***Timing and Control Unit:*** This unit derives necessary timing and control signals for internal circuit and external system bus
***Oscillator:*** generates basic timing clock signal using crystal oscillator.
***Instruction Register:*** decodes the opcode and gives information to timing and control unit.
***EPROM & program address Register***: provide on chip EPROM and mechanism to address it. All versions don't have EPROM.
***Ram & Ram Address Register:*** provide internal 128 bytes RAM and a mechanism to address internally
***ALU:*** Performs 8 bit arithmetic and logical operations over the operands held by TEMP1 and TEMP 2.User cannot access temporary registers.
***SFR Register Bank:*** set of special function registers address range: 80 H to FF H. Interrupt, serial port and timer units control and perform specific functions under the control of timing and control unit

## REGISTER SET OF 8051



Some 8051 16-bit Register hold addresses

Some 8-bitt Registers of the 8051

### Accumulator

ACC is the Accumulator register. The mnemonics for accumulator-specific instructions, howeve refer to the accumulator simply as A.

### B Register

The B register is used during multiply and divide operations. For other instructions it can b treated as another scratch pad register.

### Program Status Word

The PSW register contains program status information as detailed in Table below

**Table .**PSW: Program Status Word Register

| CY | AC | F0 | RS1 | RS0 | OV | – | P |
|---|---|---|---|---|---|---|---|

| | | |
|---|---|---|
| CY | PSW.7 | Carry flag. |
| AC | PSW.6 | Auxiliary carry flag. |
| F0 | 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 instuction cycle to indicate an odd/even number of 1 bits in the accumulator. |

| RS1 | RS0 | Register Bank | Address |
|---|---|---|---|
| 0 | 0 | 0 | 00H - 07H |
| 0 | 1 | 1 | 08H - 0FH |
| 1 | 0 | 2 | 10H - 17H |
| 1 | 1 | 3 | 18H - 1FH |

### Stack Pointer

The Stack Pointer register is 8 bits wide. It is incremented before data is stored during PUSH and CALL executions. While the stack may reside anywhere in on-chip RAM, the Stack Pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H.

**Data Pointer**
The Data Pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its intended function is to hold a 16-bit address. It may be manipulated as a 16-bit register or as two independent 8-bit registers.

**Ports 0 to 3**
P0, P1, P2 and P3 are the SFR latches of Ports 0, 1, 2 and 3, respectively.
**Serial Data Buffer**
The Serial Data Buffer is actually two separate registers, a transmit buffer and a receive buffer register. When data is moved to SBUF, it goes to the transmit buffer where it is held for serial transmission. (Moving a byte to SBUF is what initiates the transmission.) When data is moved from SBUF, it comes from the receive buffer.

**Timer Registers**
Register pairs (TH0, TL0), (TH1, TL1), and (TH2, TL2) are the 16-bit counting registers
for Timer/Counters 0, 1, and 2, respectively.
**Capture Registers**
The register pair (RCAP2H, RCAP2L) are the capture register for the Timer 2 'capture mode'. In this mode, in response to a transition at the 80C52's T2EX pin, TH2 and TL2 are copied into RCAP2H and RCAP2L. Timer 2 also has a 16-bit auto-reload mode, and RCAP2H and RCAP2L hold the reload value for this mode.
**Control Registers:** Special Function Registers IP, IE, TMOD, TCON, T2CON, SCON, and PCON contain control and status bits for the interrupt system, the timer/counters, and the serial port.

## Special Function Registers (SFR)

The set of Special Function Registers (SFRs) contains important registers such as Accumulator, Register B, I/O Port latch registers, Stack pointer, Data Pointer, Processor Status Word (PSW) and various control registers. Some of these registers are bit addressable. Addresses from 80H to FFH of all Special Function Registers

- PSW, P0-P3, IP, IE, TCON,SCON
  - Bit addressable, 8bit each, 11 in number
- SP, DPH,DPL,TMOD,TH0,TL0,TH1,TL1,SBUF,PCON
  - Byte addressable, 8bit each.
  - DPTR – data pointer, accesses external memory. DPH + DPL = DPTR
- Starting 32 bytes of RAM – general purpose registers, divided into 4 register banks of 8 registers each. Only one of these banks accessible at one time. RS1 and RS0 of PSW used to select bank.

- TH0-TL0 and TH1-TL1
  - 16 bit timer registers
- P0-P3 – port latches
- SP, PSW, IP – Interrupt Priority, IE – enable
- TCON – timer/counter control register to turn on/off the timers, interrupt control flags for external interrupts like $INT_1$ and $INT_0$
- TMOD – modes of operation of timer/counter
- SCON – serial port mode control register
- SBUF – serial data buffer for transmit and receive
- PCON – Power control register – power down bit, idle bit

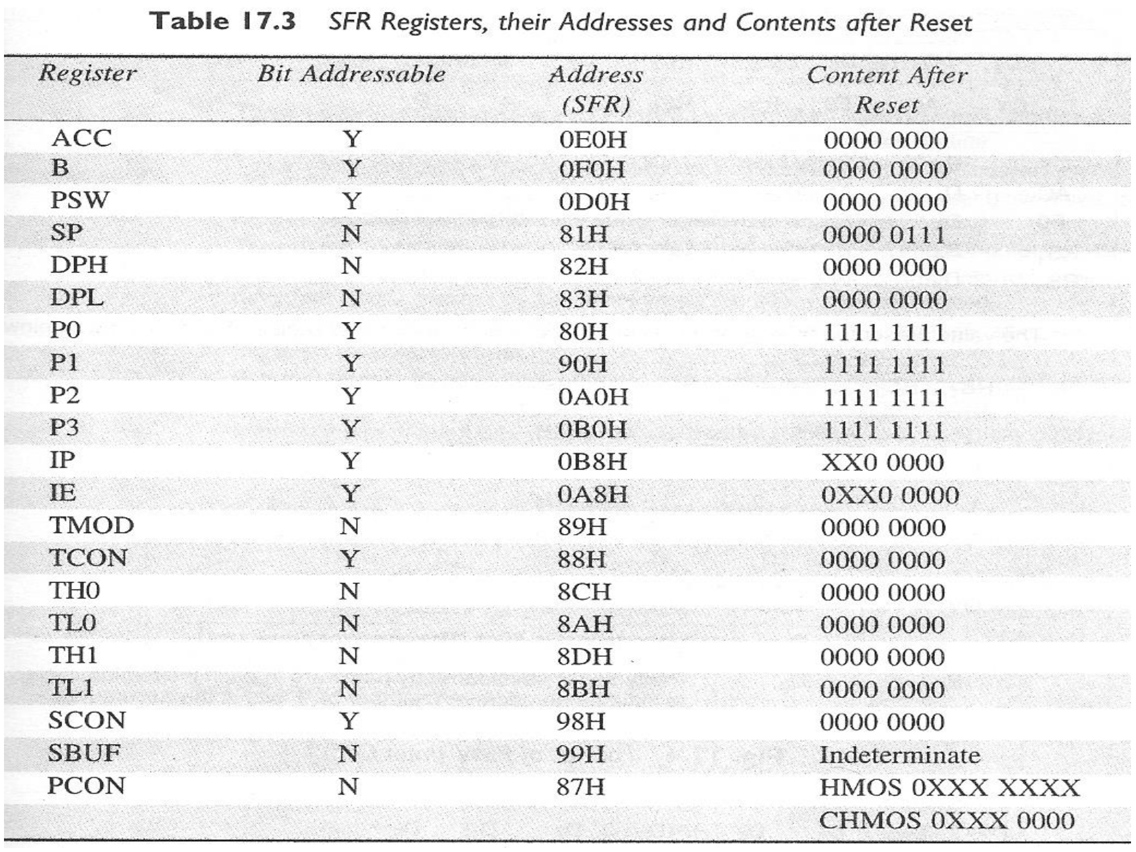


**Table 17.3** *SFR Registers, their Addresses and Contents after Reset*

| *Register* | *Bit Addressable* | *Address (SFR)* | *Content After Reset* |
|---|---|---|---|
| ACC | Y | 0E0H | 0000 0000 |
| B | Y | 0F0H | 0000 0000 |
| PSW | Y | 0D0H | 0000 0000 |
| SP | N | 81H | 0000 0111 |
| DPH | N | 82H | 0000 0000 |
| DPL | N | 83H | 0000 0000 |
| P0 | Y | 80H | 1111 1111 |
| P1 | Y | 90H | 1111 1111 |
| P2 | Y | 0A0H | 1111 1111 |
| P3 | Y | 0B0H | 1111 1111 |
| IP | Y | 0B8H | XX0 0000 |
| IE | Y | 0A8H | 0XX0 0000 |
| TMOD | N | 89H | 0000 0000 |
| TCON | Y | 88H | 0000 0000 |
| TH0 | N | 8CH | 0000 0000 |
| TL0 | N | 8AH | 0000 0000 |
| TH1 | N | 8DH | 0000 0000 |
| TL1 | N | 8BH | 0000 0000 |
| SCON | Y | 98H | 0000 0000 |
| SBUF | N | 99H | Indeterminate |
| PCON | N | 87H | HMOS 0XXX XXXX |
| | | | CHMOS 0XXX 0000 |

**8051 Clock and Instruction Cycle:**

In 8051, one instruction cycle consists of twelve (12) clock cycles. Instruction cycle is sometimes called as Machine cycle by some authors.

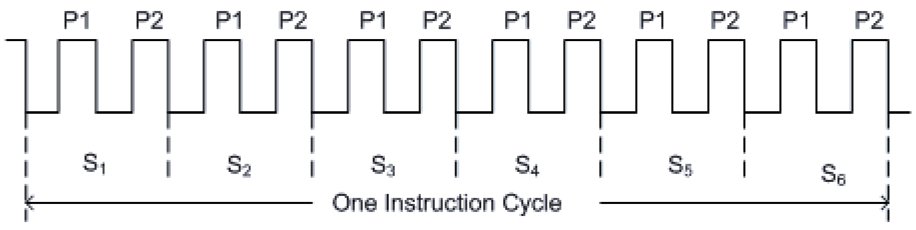


**Fig 5.2 : Instruction cycle of 8051**

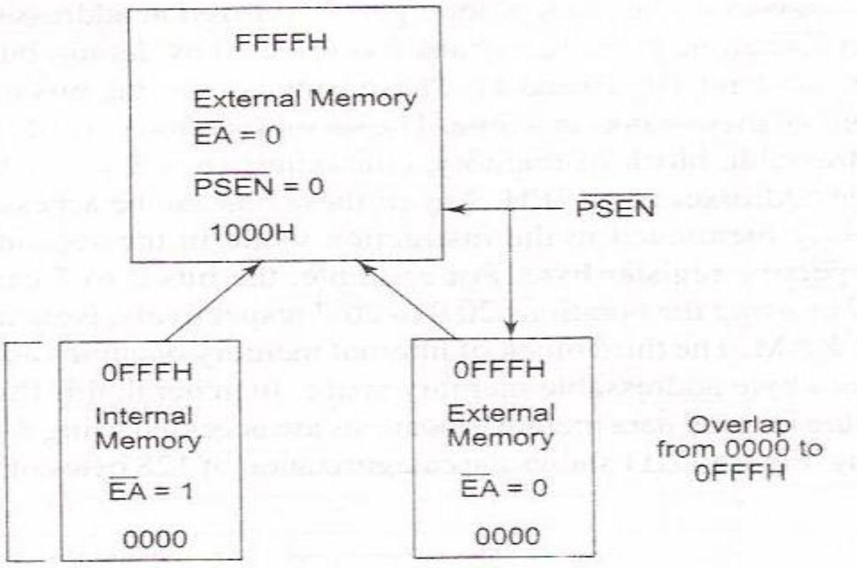
In 8051, each instruction cycle has six states ($S_1$ - $S_6$). Each state has two pulses (P1 and P2)

## **MEMORY ADDRESSING (OR) 8051 MEMORY ORGANISATION:-**

- Program memory - EPROM
  - Intermediate results, variables, const
  - 4KB internal from 0000 – 0FFFH
  - 64KB external with PSEN, till FFFFH
  - Internal –external difference PSEN
- Data Memory – RAM
  - 64KB of external with DPTR signal

- Internal memory two parts - 128 bytes Internal RAM and secondly set of addresses from 80-FFH for SFR's
- 128 bytes from 00 – 7FH direct or indirect

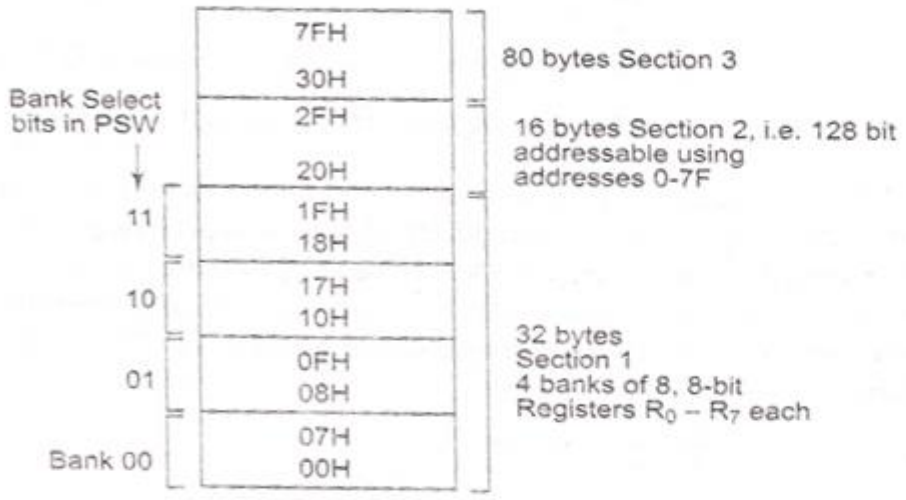
SFR addresses – only direct addressing mode



* On chip EPROM may be 8 k/16 k in some versions of 8051

**Fig. 17.9** *Program Memory Map of an 8051 System*

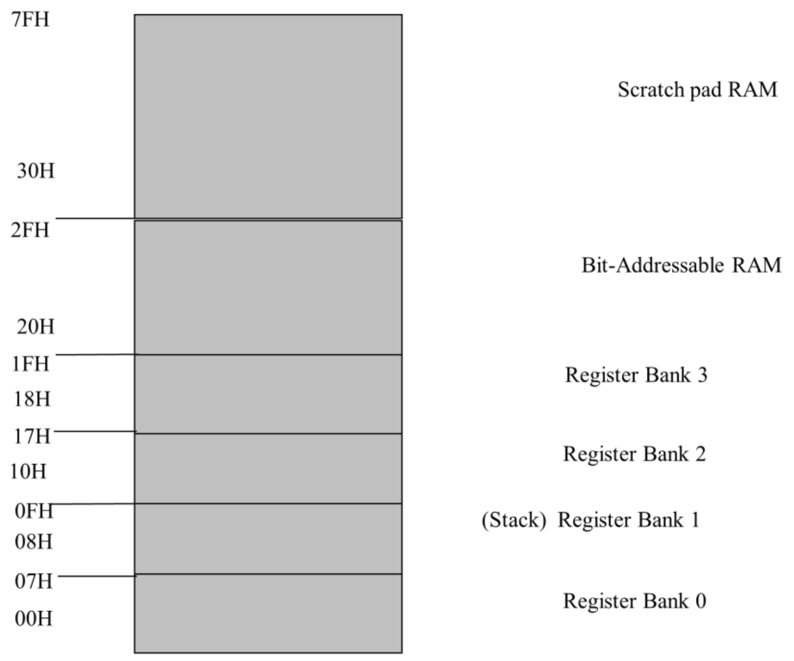
- Lower 128 bytes in three sections
  - 00-1F – 32 bytes 4 banks 00,01,10,11 each containing 8 registers of 8 bits each. Only one accessible at a time with PSW bits.
  - 20-2FH – 16bytes is bit addressable with addresses 0F to 7FH, 20.7 or 20.0, or 0-7
  - 30-7F – 80 bytes of general purpose data memory. It is byte addressable, used for stack



Bank Select
bits in PSW
7FH
30H
2FH
20H
1FH
18H
17H
10H
0FH
08H
07H
00H
11
10
01
Bank 00
80 bytes Section 3
16 bytes Section 2, i.e. 128 bit addressable using addresses 0-7F
32 bytes Section 1 4 banks of 8, 8-bit Registers $R_0 - R_7$ each

**Fig. 17.11** *Functional Description of Internal Lower 128 Bytes of RAM*

- **RAM memory space allocation in the 8051**



**128 bytes of Internal RAM Structure (lower address space)**

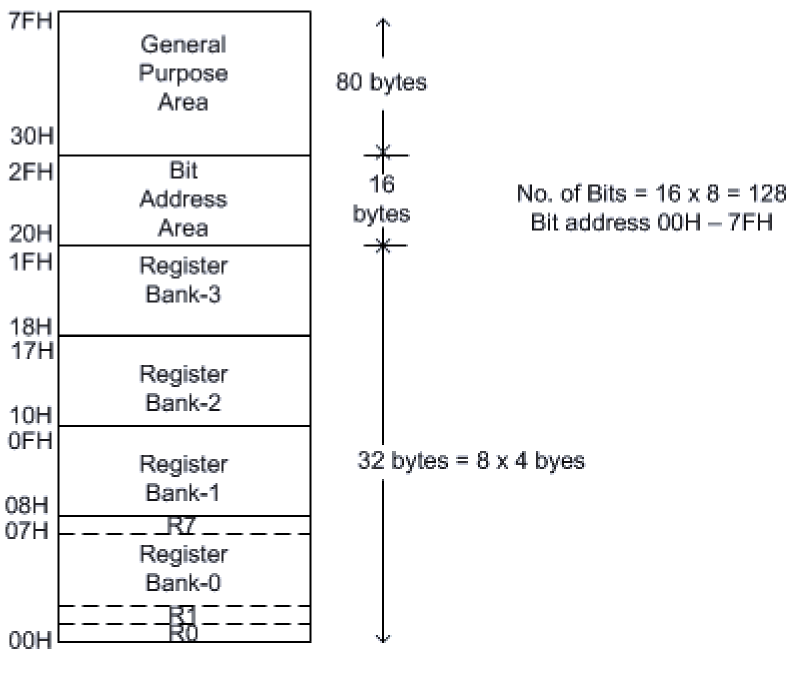


**Fig: Internal RAM Structure**

The lower 32 bytes are divided into 4 separate banks. Each register bank has 8 registers of one byte each. A register bank is selected depending upon two bank select bits in the PSW register. Next 16bytes are bit addressable. In total, 128bits (16X8) are available in bitaddressable area. Each bit can be accessed and modified by suitable instructions. The bit addresses are from 00H (LSB of the first byte in 20H) to 7FH (MSB of the last byte in 2FH). Remaining 80bytes of RAM are available for general purpose.

**Internal Data Memory and Special Function Register (SFR) Map**

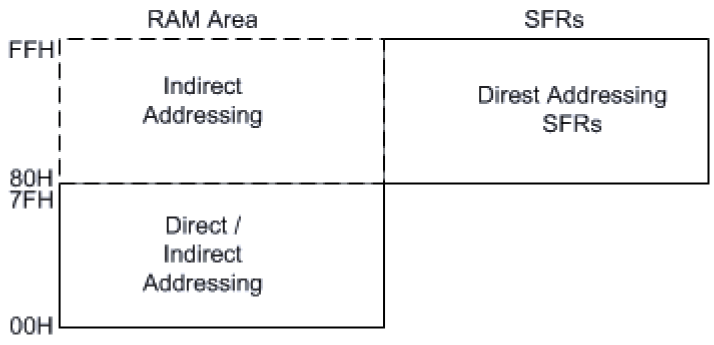


**Fig: Internal Data Memory Map**

The special function registers (SFRs) are mapped in the upper 128 bytes of internal data memory address. Hence there is an address overlap between the upper 128 bytes of data RAM and SFRs. Please note that the upper 128 bytes of data RAM are present only in the 8052 family. The lower128 bytes of RAM (00H - 7FH) can be accessed both by direct or indirect addressing while the upper 128 bytes of RAM (80H - FFH) are accessed by indirect addressing. The SFRs (80H - FFH) are accessed by direct addressing only. This feature distinguishes the upper 128 bytes of memory from the SFRs, as shown in figure above.

## <u>TIMERS / COUNTERS</u>

8051 has two 16-bit programmable UP timers/counters. They can be configured to operate either as timers or as event counters. The names of the two counters are T0 and T1 respectively.  The timer content is available in four 8-bit special function registers, viz, TL0, TH0, TL1 and TH1 respectively.

In the "timer" function mode, the counter is incremented in every machine cycle. Thus, one can think of it as counting machine cycles. Hence the clock rate is 1/12 th of the oscillator frequency.

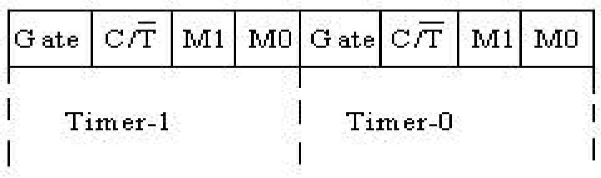
In the "counter" function mode, the register is incremented in response to a 1 to 0 transition at its corresponding external input pin (T0 or T1). It requires 2 machine cycles to detect a high to low transition. Hence maximum count rate is 1/24 th of oscillator frequency.

The operation of the timers/counters is controlled by two special function registers, TMOD and TCON respectively.

**Timer Mode control (TMOD) Special Function Register:**

TMOD register is not bit addressable.

TMOD Address: 89 H



Various bits  of TMOD are described as follows -

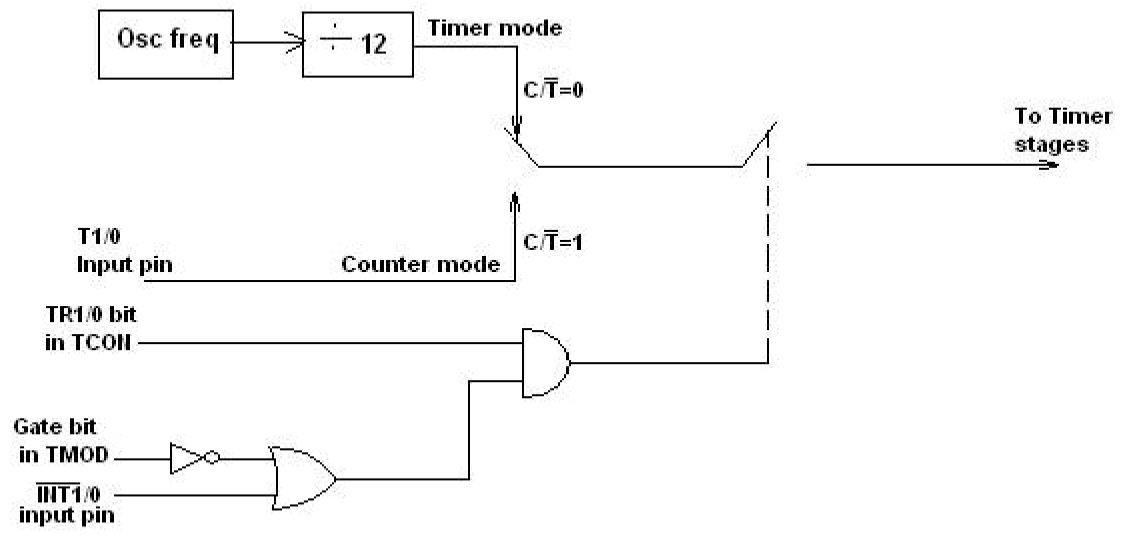
**Gate:** This is an OR Gate enabled bit which controls the effect of $\overline{INT1}/0$on START/STOP of Timer. It is set to one ('1') by the program to enable the interrupt to start/stop the timer. If TR1/0 in TCON is set and signal on $\overline{INT1}/0$pin is high then the timer starts counting using either internal clock (timer mode) or external pulses (counter mode).

**C/$\overline{T}$:** It is used for the selection of Counter/Timer mode.
Mode Select Bits:

| M1 | M0 | Mode |
|---|---|---|
| 0 | 0 | Mode 0 |
| 0 | 1 | Mode 1 |
| 1 | 0 | Mode 2 |
| 1 | 1 | Mode 3 |

M1 and M0 are mode select bits.

**Timer/ Counter control logic:**



**Timer control (TCON) Special function register:**

TCON is bit addressable. The address of TCON is 88H. It is partly related to Timer and partly to interrupt.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |
| Timer | | | | Interrupt | | | |

The various bits of TCON are as follows.

**TF1:** Timer1 overflow flag. It is set when timer rolls from all 1s to 0s. It is cleared when processor vectors to execute ISR located at address 001BH.
**TR1:** Timer1 run control bit. Set to 1 to start the timer / counter.
**TF0:** Timer0 overflow flag. (Similar to TF1)
TR0: Timer0 run control bit.
**IE1:** Interrupt1 edge flag. Set by hardware when an external interrupt edge is detected. It is cleared when interrupt is processed.
**IE0:** Interrupt0 edge flag. (Similar to IE1)
**IT1**: Interrupt1 type control bit. Set/ cleared by software to specify falling edge / low level triggered external interrupt.
**IT0:** Interrupt0 type control bit. (Similar to IT1)
As mentioned earlier, Timers can operate in four different modes. They are as follows

## **INTERRUPTS**

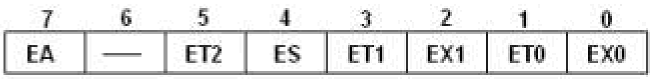
8051 provides 5 vectored interrupts. They are -

1. $\overline{\text{INT0}}$
2. TF0
3. $\overline{\text{INT1}}$
4. TF1
5. RI/TI

Out of these, $\overline{\text{INT0}}$and $\overline{\text{INT1}}$ are external interrupts whereas Timer and Serial port interrupts are generated internally. The external interrupts could be negative edge triggered or low level triggered. All these interrupt, when activated, set the corresponding interrupt flags. Except for serial interrupt, the interrupt flags are cleared when the processor branches to the Interrupt Service Routine (ISR). The external interrupt flags are cleared on branching to Interrupt Service Routine (ISR), provided the interrupt is negative edge triggered. For low level triggered external interrupt as well as for serial interrupt, the corresponding flags have to be cleared by software by the programmer.

**Interrupt Enable register (IE):**

Address: A8H



| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| EA | — | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

EX0 $\longrightarrow$ $\overline{\text{INT0}}$ interrupt (External) enable bit

ET0 $\longrightarrow$ Timer-0 interrupt enable bit

EX1 $\longrightarrow$ $\overline{\text{INT1}}$ interrupt (External) enable bit

ET1 $\longrightarrow$ Timer-1 interrupt enable bit

ES $\longrightarrow$ Serial port interrupt enable bit

ET2 $\longrightarrow$ Timer-2 interrupt enable bit

EA $\longrightarrow$ Enable/Disable all

Setting '1' $\longrightarrow$ Enable the corresponding interrupt

Setting '0' $\longrightarrow$ Disable the corresponding interrupt

**Priority level structure:**

Each interrupt source can be programmed to have one of the two priority levels by setting (high priority) or clearing (low priority) a bit in the IP (Interrupt Priority) Register . A low priority interrupt can itself be interrupted by a high priority interrupt, but not by another low priority interrupt. If two interrupts of different priority levels are received simultaneously, the request of higher priority level is served. If the requests of the same priority level are received simultaneously, an internal polling sequence determines which request is to be serviced. Thus, within each priority level, there is a second priority level determined by the polling sequence, as follows.

| Source | Priority level |
|---|---|
| IE0 | Highest |
| TF0 | |
| IE1 | $\downarrow$ |
| TF1 | |
| RI+TI | Lowest |

**Interrupt Priority register (IP)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| —— | —— | PT2 | PS | PT1 | PX1 | PT0 | PX0 |

'0' $\longrightarrow$ low priority

'1' $\longrightarrow$ high priority

## SERIAL INTERFACE

The serial port of 8051 is full duplex, i.e., it can transmit and receive simultaneously.

The register SBUF is used to hold the data. The special function register SBUF is physically two registers. One is, write-only and is used to hold data to be transmitted out of the 8051 via TXD. The other is, read-only and holds the received data from external sources via RXD. Both mutually exclusive registers have the same address 099H.

**Serial Port Control Register (SCON):**

Register SCON controls serial data communication.
Address: 098H (Bit addressable)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

Mode select bits

| SM0 | SM1 | Mode |
|---|---|---|
| 0 | 0 | Mode 0 |
| 0 | 1 | Mode 1 |
| 1 | 0 | Mode 2 |
| 1 | 1 | Mode 3 |

SM2: multi processor communication bit
REN: Receive enable bit
TB8: Transmitted bit 8 (Normally we have 0-7 bits transmitted/received)
RB8: Received bit 8
TI: Transmit interrupt flag
RI: Receive interrupt flag

**Power Mode control Register(PCON):**

Register PCON controls processor power down, sleep modes and serial data baud rate. Only one bit of PCON is used with respect to serial communication. The seventh bit (b7) (SMOD) is used to generate the baud rate of serial communication.

Address: 87H

| b7 | | | | | | | b0 |
|---|---|---|---|---|---|---|---|
| SMOD | — | — | — | GF1 | GF0 | PD | IDL |

SMOD: Serial baud rate modify bit
GF1: General purpose user flag bit 1
GF0: General purpose user flag bit 0
PD: Power down bit
IDL: Idle mode bit

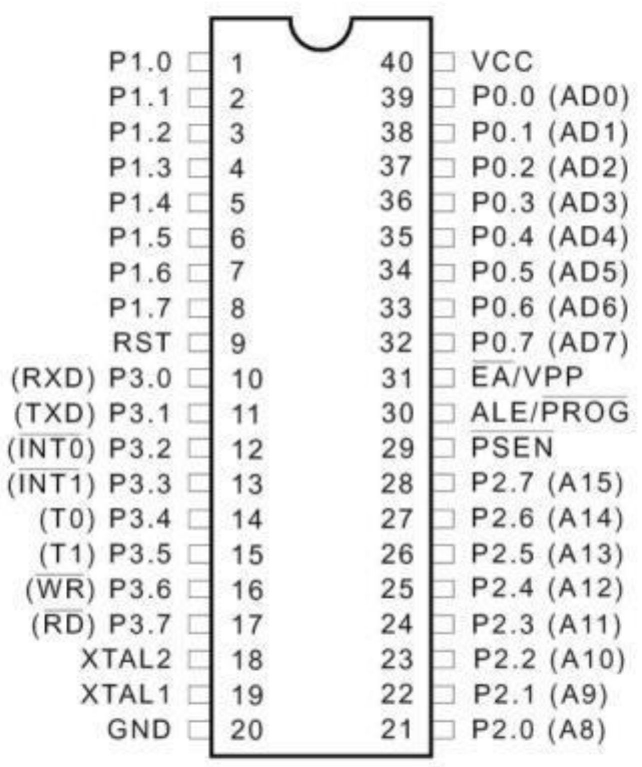
**Data Transmission**

Transmission of serial data begins at any time when data is written to SBUF. Pin P3.1 (Alternate function bit TXD) is used to transmit data to the serial data network. TI is set to 1 when data has been transmitted. This signifies that SBUF is empty so that another byte can be sent.

**Data Reception**

Reception of serial data begins if the receive enable bit is set to 1 for all modes. Pin P3.0 (Alternate function bit RXD) is used to receive data from the serial data network. Receive interrupt flag, RI, is set after the data has been received in all modes. The data gets stored in SBUF register from where it can be read.

## PIN DIAGRAM OF 8051



**Pin out Description:**

**Pins 1-8:** Port 1 Each of these pins can be configured as an input or an output.

**Pin 9**: RS A logic one on this pin disables the microcontroller and clears the contents of most registers. In other words, the positive voltage on this pin resets the microcontroller. By applying logic zero to this pin, the program starts execution from the beginning.

**Pins10-17:** Port 3 Similar to port 1, each of these pins can serve as general input or output. Besides, all of them have alternative functions:

**Pin 10:** RXD Serial asynchronous communication input or Serial synchronous communication output.

**Pin 11:** TXD Serial asynchronous communication output or Serial synchronous communication clock output.

**Pin 12:** INT0 Interrupt 0 input.

**Pin 13:** INT1 Interrupt 1 input.

**Pin 14:** T0 Counter 0 clock input.

**Pin 15:** T1 Counter 1 clock input.

**Pin 16:** WR Write to external (additional) RAM.

**Pin 17:** RD Read from external RAM.

**Pin 18, 19:** X2, X1 Internal oscillator input and output. A quartz crystal which specifies operating frequency is usually connected to these pins. Instead of it, miniature ceramics resonators can also be used for frequency stability. Later versions of microcontrollers operate at a frequency of 0 Hz up to over 50 Hz.

**Pin 20:** GND Ground.

**Pin 21-28:** Port 2 If there is no intention to use external memory then these port pins are configured as general inputs/outputs. In case external memory is used, the higher address byte, i.e. addresses A8-A15 will appear on this port. Even though memory with capacity of 64Kb is not used, which means that not all eight port bits are used for its addressing, the rest of them are not available as inputs/outputs.

**Pin 29:** PSEN If external ROM is used for storing program then a logic zero (0) appears on it every time the microcontroller reads a byte from memory.

**Pin 30:** ALE Prior to reading from external memory, the microcontroller puts the lower address byte (A0-A7) on P0 and activates the ALE output. After receiving signal from the ALE pin, the external register (usually 74HCT373 or 74HCT375 add-on chip) memorizes the state of P0 and uses it as a memory chip address. Immediately after that, the ALU pin is returned its previous logic state and P0 is now used as a Data Bus. As seen, port data multiplexing is performed by means of only one additional (and cheap) integrated circuit. In other words, this port is used for both data and address transmission.

**Pin 31:** EA By applying logic zero to this pin, P2 and P3 are used for data and address transmission with no regard to whether there is internal memory or not. It means that even there is a program written to the microcontroller, it will not be executed. Instead, the program written to external ROM will be executed. By applying logic one to the EA pin, the microcontroller will use both memories, first internal then external (if exists).

**Pin 32-39:** Port 0 Similar to P2, if external memory is not used, these pins can be used as general inputs/outputs. Otherwise, P0 is configured as address output (A0-A7) when the ALE pin is driven high (1) or as data output (Data Bus) when the ALE pin is driven low (0).

**Pin 40:** VCC +5V power supply.

## 8051 ADDRESSING MODES

8051 has four addressing modes.

- Immediate Addressing
- Bank Addressing or Register Addressing
- Direct Addressing
- Register Indirect Addressing

**Immediate Addressing:** Data is immediately available in the instruction.
For example -
ADD A, #77; Adds 77 (decimal) to A and stores in A
ADD A, #4DH; Adds 4D (hexadecimal) to A and stores in A
MOV DPTR, #1000H; Moves 1000 (hexadecimal) to data pointer

**Bank Addressing or Register Addressing:** This way of addressing accesses the bytes in the current register bank. Data is available in the register specified in the instruction. The register bank is decided by 2 bits of Processor Status Word (PSW).
For example-
ADD A, R0; Adds content of R0 to A and stores in A

**Direct Addressing:** The address of the data is available in the instruction.
For example -
MOV A, 088H; Moves content of SFR TCON (address 088H)to A

**Register Indirect Addressing:** The address of data is available in the R0 or R1 registers as specified in the instruction.
For example -
MOV A, @R0 moves content of address pointed by R0 to A

**External Data Addressing:** Pointer used for external data addressing can be either R0/R1 (256 byte access) or DPTR (64kbyte access).
For example -
MOVX A, @R0; Moves content of 8-bit address pointed by R0 to A
MOVX A, @DPTR; Moves content of 16-bit address pointed by DPTR to A
**External Code Addressing:** Sometimes we may want to store non-volatile data into the ROM e.g. look-up tables. Such data may require reading the code memory.
This may be done as follows -
MOVC A, @A+DPTR; Moves content of address pointed by A+DPTR to A
MOVC A, @A+PC; Moves content of address pointed by A+PC to A

## 8051 INSTRUCTIONS

8051 has about 111 instructions. These can be grouped into the following categories

1. Arithmetic Instructions
2. Logical Instructions
3. Data Transfer instructions
4. Boolean Variable Instructions
5. Program Branching Instructions

The following nomenclatures for register, data, address and variables are used while write instructions.
A: Accumulator

B: "B" register

C: Carry bit

Rn: Register R0 - R7 of the currently selected register bank

Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of RAM (00 - 7FH) or it could be in the special function register (80 - FFH).

@Ri: 8-bit external or internal RAM address available in register R0 or R1. This is used for indirect addressing mode.

#data8: Immediate 8-bit data available in the instruction.

#data16: Immediate 16-bit data available in the instruction.

Addr11: 11-bit destination address for short absolute jump. Used by instructions AJMP & ACALL. Jump range is 2 kbyte (one page).

Addr16: 16-bit destination address for long call or long jump.

Rel: 2's complement 8-bit offset (one - byte) used for short jump (SJMP) and all conditional jumps.

bit: Directly addressed bit in internal RAM or SFR

**Arithmetic Instructions:**

| Mnemonics | Description | Bytes | Instruction Cycles |
|---|---|---|---|
| ADD A, Rn | A $\longleftarrow$ A + Rn | 1 | 1 |
| ADD A, direct | A $\longleftarrow$ A + (direct) | 2 | 1 |
| ADD A, @Ri | A $\longleftarrow$ A + @Ri | 1 | 1 |
| ADD A, #data | A $\longleftarrow$ A + data | 2 | 1 |
| ADDC A, Rn | A $\longleftarrow$ A + Rn + C | 1 | 1 |
| ADDC A, direct | A $\longleftarrow$ A + (direct) + C | 2 | 1 |
| ADDC A, @Ri | A $\longleftarrow$ A + @Ri + C | 1 | 1 |
| ADDC A, #data | A $\longleftarrow$ A + data + C | 2 | 1 |
| DA A | Decimal adjust accumulator | 1 | 1 |
| DIV AB | Divide A by B | 1 | 4 |

| | | | |
|---|---|---|---|
| | A ⟵quotient<br>B ⟵remainder | | |
| DEC A | A ⟵A -1 | 1 | 1 |
| DEC Rn | Rn ⟵Rn - 1 | 1 | 1 |
| DEC direct | (direct) ⟵(direct) - 1 | 2 | 1 |
| DEC @Ri | @Ri ⟵@Ri - 1 | 1 | 1 |
| INC A | A ⟵A+1 | 1 | 1 |
| INC Rn | Rn ⟵Rn + 1 | 1 | 1 |
| INC direct | (direct) ⟵(direct) + 1 | 2 | 1 |
| INC @Ri | @Ri ⟵@Ri +1 | 1 | 1 |
| INC DPTR | DPTR ⟵DPTR +1 | 1 | 2 |
| MUL AB | Multiply A by B<br>A ⟵low byte (A*B)<br>B ⟵high byte (A* B) | 1 | 4 |
| SUBB A, Rn | A ⟵A - Rn - C | 1 | 1 |
| SUBB A, direct | A ⟵A - (direct) - C | 2 | 1 |
| SUBB A, @Ri | A ⟵A - @Ri - C | 1 | 1 |
| SUBB A, #data | A ⟵A - data - C | 2 | 1 |

**Logical Instructions:**

| Mnemonics | Description | Bytes | Instruction Cycles |
|---|---|---|---|
| ANL A, Rn | A ⟵A AND Rn | 1 | 1 |
| ANL A, direct | A ⟵A AND (direct) | 2 | 1 |
| ANL A, @Ri | A ⟵A AND @Ri | 1 | 1 |
| ANL A, #data | A ⟵A AND data | 2 | 1 |
| ANL direct, A | (direct) ⟵(direct) AND A | 2 | 1 |
| ANL direct, #data | (direct) ⟵(direct) AND data | 3 | 2 |
| CLR A | A⟵ 00H | 1 | 1 |
| CPL A | A⟵A | 1 | 1 |
| ORL A, Rn | A ⟵A OR Rn | 1 | 1 |
| ORL A, direct | A ⟵A OR (direct) | 1 | 1 |
| ORL A, @Ri | A ⟵A OR @Ri | 2 | 1 |
| ORL A, #data | A ⟵A OR data | 1 | 1 |
| ORL direct, A | (direct) ⟵(direct) OR A | 2 | 1 |
| ORL direct, #data | (direct) ⟵(direct) OR data | 3 | 2 |
| RL A | Rotate accumulator left | 1 | 1 |
| RLC A | Rotate accumulator left through carry | 1 | 1 |
| RR A | Rotate accumulator right | 1 | 1 |
| RRC A | Rotate accumulator right through carry | 1 | 1 |
| SWAP A | Swap nibbles within Acumulator | 1 | 1 |
| XRL A, Rn | A ⟵A EXOR Rn | 1 | 1 |
| XRL A, direct | A ⟵A EXOR (direct) | 1 | 1 |
| XRL A, @Ri | A ⟵A EXOR @Ri | 2 | 1 |
| XRL A, #data | A ⟵A EXOR data | 1 | 1 |
| XRL direct, A | (direct) ⟵(direct) EXOR A | 2 | 1 |

|  |  |  |  |
|---|---|---|---|
| XRL direct, #data | (direct) ⟵(direct) EXOR data | 3 | 2 |

**Data Transfer Instructions:**

| Mnemonics | Description | Bytes | Instruction Cycles |
|---|---|---|---|
| MOV A, Rn | A ⟵Rn | 1 | 1 |
| MOV A, direct | A ⟵(direct) | 2 | 1 |
| MOV A, @Ri | A ⟵@Ri | 1 | 1 |
| MOV A, #data | A ⟵data | 2 | 1 |
| MOV Rn, A | Rn ⟵A | 1 | 1 |
| MOV Rn, direct | Rn ⟵(direct) | 2 | 2 |
| MOV Rn, #data | Rn ⟵data | 2 | 1 |
| MOV direct, A | (direct) ⟵A | 2 | 1 |
| MOV direct, Rn | (direct) ⟵Rn | 2 | 2 |
| MOV direct1, direct2 | (direct1) ⟵(direct2) | 3 | 2 |
| MOV direct, @Ri | (direct)⟵ @Ri | 2 | 2 |
| MOV direct, #data | (direct) ⟵#data | 3 | 2 |
| MOV @Ri, A | @Ri ⟵A | 1 | 1 |
| MOV @Ri, direct | @Ri ⟵(direct) | 2 | 2 |
| MOV @Ri, #data | @Ri ⟵data | 2 | 1 |
| MOV DPTR, #data16 | DPTR ⟵data16 | 3 | 2 |
| MOVC A, @A+DPTR | A ⟵Code byte pointed by A + DPTR | 1 | 2 |
| MOVC A, @A+PC | A ⟵Code byte pointed by A + PC | 1 | 2 |
| MOVC A, @Ri | A ⟵Code byte pointed by Ri 8-bit address) | 1 | 2 |
| MOVX A, @DPTR | A ⟵External data pointed by DPTR | 1 | 2 |
| MOVX @Ri, A | @Ri ⟵A (External data - 8bit address) | 1 | 2 |
| MOVX @DPTR, A | @DPTR ⟵A(External data - 16bit address) | 1 | 2 |
| PUSH direct | (SP) ⟵(direct) | 2 | 2 |
| POP direct | (direct) ⟵(SP) | 2 | 2 |
| XCH Rn | Exchange A with Rn | 1 | 1 |
| XCH direct | Exchange A with direct byte | 2 | 1 |
| XCH @Ri | Exchange A with indirect RAM | 1 | 1 |
| XCHD A, @Ri | Exchange least significant nibble of A with that of indirect RAM | 1 | 1 |

**Boolean Variable Instructions:**

| Mnemonics | Description | Bytes | Instruction Cycles |
|---|---|---|---|
| CLR C | C-bit ⟵0 | 1 | 1 |
| CLR bit | bit ⟵0 | 2 | 1 |
| SET C | C ⟵1 | 1 | 1 |
| SET bit | bit ⟵1 | 2 | 1 |
| CPL C | C⟵ $\overline{\text{C-bit}}$ | 1 | 1 |

| | | | |
|---|---|---|---|
| CPL bit | bit $\longleftarrow \overline{\text{bit}}$ | 2 | 1 |
| ANL C, /bit | C $\longleftarrow$C . $\overline{\text{bit}}$ | 2 | 1 |
| ANL C, bit | C $\longleftarrow$C. bit | 2 | 1 |
| ORL C, /bit | C$\longleftarrow$ C + $\overline{\text{bit}}$ | 2 | 1 |
| ORL C, bit | C $\longleftarrow$C + bit | 2 | 1 |
| MOV C, bit | C$\longleftarrow$ bit | 2 | 1 |
| MOV bit, C | bit $\longleftarrow$C | 2 | 2 |

**Program Branching Instructions:**

| Mnemonics | Description | Bytes | Instruction Cycles |
|---|---|---|---|
| ACALL addr11 | PC + 2 $\longrightarrow$(SP) ; addr 11$\longrightarrow$ PC | 2 | 2 |
| AJMP addr11 | Addr11 $\longrightarrow$PC | 2 | 2 |
| CJNE A, direct, rel | Compare with A, jump (PC + rel) if not equal | 3 | 2 |
| CJNE A, #data, rel | Compare with A, jump (PC + rel) if not equal | 3 | 2 |
| CJNE Rn, #data, rel | Compare with Rn, jump (PC + rel) if not equal | 3 | 2 |
| CJNE @Ri, #data, rel | Compare with @Ri A, jump (PC + rel) if not equal | 3 | 2 |
| DJNZ Rn, rel | Decrement Rn, jump if not zero | 2 | 2 |
| DJNZ direct, rel | Decrement (direct), jump if not zero | 3 | 2 |
| JC rel | Jump (PC + rel) if C bit = 1 | 2 | 2 |
| JNC rel | Jump (PC + rel) if C bit = 0 | 2 | 2 |
| JB bit, rel | Jump (PC + rel) if bit = 1 | 3 | 2 |
| JNB bit, rel | Jump (PC + rel) if bit = 0 | 3 | 2 |
| JBC bit, rel | Jump (PC + rel) if bit = 1 | 3 | 2 |
| JMP @A+DPTR | A+DPTR $\longrightarrow$PC | 1 | 2 |
| JZ rel | If A=0, jump to PC + rel | 2 | 2 |
| JNZ rel | If A ≠ 0 , jump to PC + rel | 2 | 2 |
| LCALL addr16 | PC + 3 $\longrightarrow$(SP), addr16 $\longrightarrow$PC | 3 | 2 |
| LJMP addr 16 | Addr16 $\longrightarrow$PC | 3 | 2 |
| NOP | No operation | 1 | 1 |
| RET | (SP) $\longrightarrow$PC | 1 | 2 |
| RETI | (SP) $\longrightarrow$PC, Enable Interrupt | 1 | 2 |
| SJMP rel | PC + 2 + rel $\longrightarrow$PC | 2 | 2 |
| JMP  @A+DPTR | A+DPTR $\longrightarrow$PC | 1 | 2 |
| JZ  rel | If A = 0. jump PC+ rel | 2 | 2 |
| JNZ  rel | If A ≠ 0, jump PC + rel | 2 | 2 |
| NOP | No operation | 1 | 1 |