Digital Computers:-

A computer performs a task by processing a sequential list of instructions. A list of instructions constitute a program. A set of programs to achieve a desired objective is known as software. The physical components of a computer constitute the hardware.

A computer consists of 5 basic hardware blocks, They are,

1. Central Processing unit (CPU), 2. Main Memory unit (MMU),
3. Secondary Storage unit (SSU), 4. Input Unit (IU), 5. Output Unit (OU).

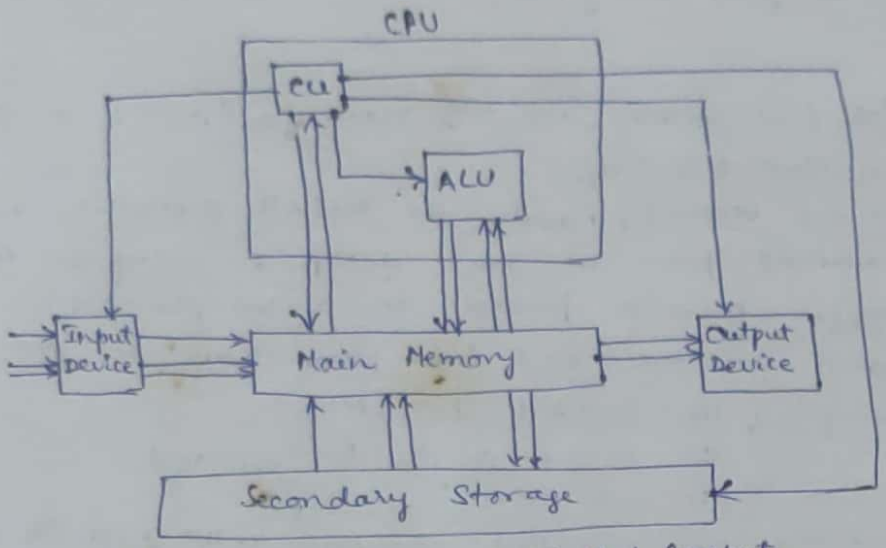


Fig. Block diagram of Digital computer

CPU = CU + ALU
where, CU = control unit
ALU = Arithmetic and logic unit.

The microprocessor functions as the CPU of the computer system.

The unit of the processing capacity of CPU is defined interms of Million Instructions Per Second (MIPS) or Floating Point Operations Per Second (FLOPS).

Microprocessors :-

"Microprocessor is a programmable electronic device that controls interpretation and execution of the instructions. It is called as CPU of the computer."

The word "micro" in the microprocessor refers to its small size and the "processor" refers to the device that performs computational and control operations.

The world's first microprocessor is Intel's 4004, introduced in 1971.

The present single chip, 16-bit microprocessors market has 3 major designers and manufacturers.

* The 8086 : designed by Intel
* The Z8000 : designed by Zilog
* The MC 68000 : designed by Motorola.

The Intel 8086 is the oldest and the simplest. The 8086 has 95 basic instructions of which a substantial no. of them are only 8-bits long and support 24 addressing modes.

Microprocessor

1. Microprocessor contains ALU, GPRs, stack pointer, program counter, clock timing ckt and interrupt circuit.
2. It has many instructions to move data between memory and CPU.
3. It has one or two bit handling instructions.
4. Access times for memory and I/O devices are more.
5. Microprocessor based systems requires more hardware.
6. Microprocessor based system is more flexible in design point of view.
7. It has single memory map for data and code.
8. Less no. of pins are multifunctioned.

Microcontroller

1. Microcontroller contains the circuitry of microprocessor and in addition it has built-in ROM, RAM, I/O devices, timers and counters.
2. It has one or two instructions to move data between memory and CPU.
3. It has many bit handling instructions.
4. Less access times for built-in memory and I/O devices.
5. Microcontroller based system requires less hardware, reducing PCB size and increasing the reliability.
6. Less flexible in design point of view.
7. It has seperate memory map for data and code.
8. More no. of pins are multifunctioned.

Microprocessor

① A silicon chip representing a CPU, which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions.

② It is a dependent unit. It requires the combination of other chips like timers, program and data memory chips, interrupt controllers, etc. for functioning.

③ Most of the time general purpose in design and operation.

④ Doesnot contain a builtin I/O port. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255.

⑤ Targeted for high end market where performance is important.

⑥ Limited power saving options compared to microcontrollers.

Microcontroller

① A microcontroller is a highly integrated chip that contains a CPU, scratch pad RAM, special and general purpose register arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports.

② It is a self-contained unit and doesnot require external interrupt controller, timer, UART, etc. for its functioning.

③ Mostly application-oriented or domain-specific.

④ Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32-bit port or as individual port pins.

⑤ Targeted for embedded market where performance is not so critical.

⑥ Includes lot of power saving features.

RISC	CISC
① Lesser number of instructions.	① Greater number of instructions
② Instruction pipelining and increased execution speed.	② Generally no instruction pipelining feature.
③ Orthogonal instruction set (Allows each instruction to operate on any register and use any addressing mode).	③ Non-orthogonal instruction set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction-specific).
④ Operations are performed on registers only, the only memory operations are load and store.	④ Operations are performed on registers or memory depending on the instruction.
⑤ A large no. of registers are available.	⑤ Limited no. of general purpose registers.
⑥ Programmer needs to write code to execute a task since the instructions are simpler ones.	⑥ Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC.
⑦ Single, fixed length instructions	⑦ Variable length instructions
⑧ Less silicon usage and pin count	⑧ More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
⑨ With Harvard Architecture	⑨ Can be Harvard or Von-Neumann architecture.

Harvard vs. Von-Neumann Processor/ controller Architecture :-

Microprocessors / controllers based on the Von-Neumann architecture shares a single common bus for fetching both instructions and data. Program instructions and data are stored in a common main memory. Von-Neumann architecture based processors / controllers first fetch an instruction and then fetch the data to support the instruction from code memory. The two seperate fetches slows down the controller's operation. Von-Neumann architecture is also referred as Princeton architecture, since it was developed by the Princeton University.

Microprocessors / controllers based on the Harvard architecture will have seperate data bus and instruction bus. This allows the data transfer and program fetching to occur simultaneously on both buses. With Harvard architecture, the data memory can be read and written while the program memory is being accessed. These seperated data memory and code memory buses allow one instruction to execute while the next instruction is fetched (pre-fetching). The pre-fetch allows much faster execution than Von-Neumann architecture. Since some additional hardware logic is required for the generation

of control signals for this type of operation, it adds silicon complexity to the system.

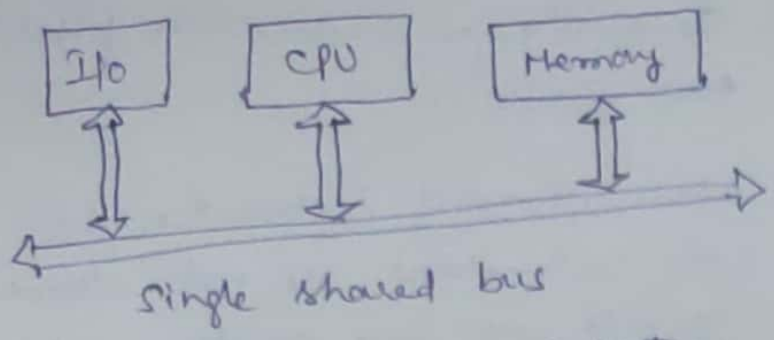


fig. Von-Neumann architecture

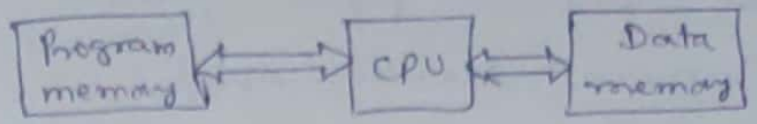


fig. Harvard architecture

Harvard Architecture

① seperate buses for instruction and data fetching.

② Easier to pipeline, so high performance can be achieved

③ Comparatively cost high

④ No memory alignment problems

⑤ Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory.

Von-Neumann Architecture

① Single shared bus for instruction and data fetching.

② Low performance compared to Harvard architecture

③ cheaper

④ Allows self modifying codes (self modifying code is a code/instruction which modifies itself while execution).

⑤ Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory.

Intel 8086

Introduction :-

The 8086 is Intel's first 16-bit microprocessor. The 8086 is designed using the HMOS technology and contains approximately 29,000 transistors. The 8086 is packaged in a 40-pin DIP (Dual In-line Package) and requires a single 5V power supply. The 8086 can be operated at 3 different clock speeds. The standard 8086 runs at 5MHz internal clock frequency, whereas the 8086-2 and 8086-4 run at internal clock frequencies of 8 and 4 MHz, respectively. An external clock generator/driver chip such as the Intel 8284 is needed to generate the 8086 clock input signal.

The 8086 has a 20-bit address and hence it can directly address upto one megabyte (2^{20}) of memory. The 8086 uses a segmented memory.

Register organisation of 8086 :-

The 8086 processor provides on-chip (within the processor) storage elements known as registers. The registers are 16-bit registers.

The fourteen, 16-bit hardware registers are divided into 5 groups as,

- General Purpose Registers — AX, BX, CX, and DX → General Purpose registers
- Pointer and Index Registers — SP, BP, SI, and DI → Special purpose registers
- Segment Registers — CS, DS, ES, and SS.
- Instruction Pointer Register — IP
- Flag Register — F

The general purpose registers are either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes, etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.

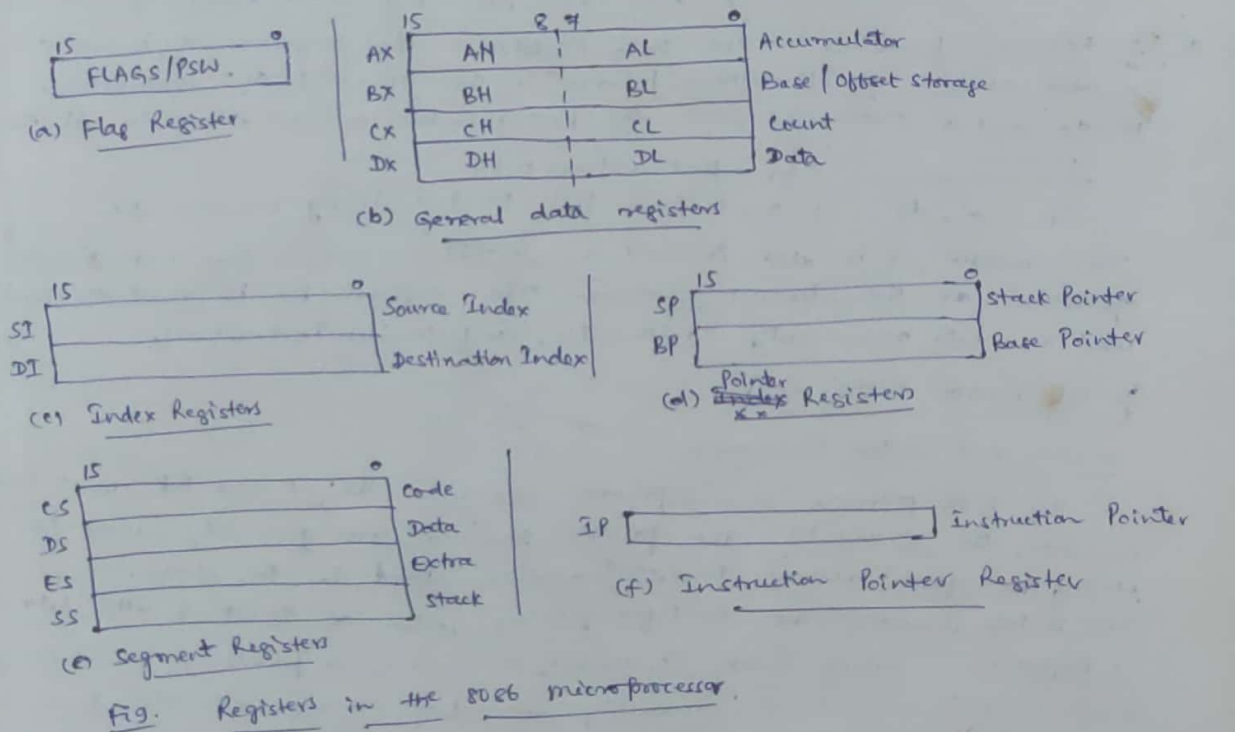


Fig. Registers in the 8086 microprocessor.

I. General Purpose Registers (GPRs):-

The 4 GPRs of 8086 which are 16-bits, can be used as operands and also used as the source and destination register during the transfer of data and computation, as pointers to memory and as counters.

① AX Register:- function as the Accumulator. The register AX is always involved in multiplication and division as the default operand. The usage of AX register is the most efficient in data movement, arithmetic and logical operations. The register AX is divided into 2 parts. The lower 8-bits of the AX register is AL register and the higher 8-bit of the AX register is AH register. The splitting of the AX register into AH and AL registers is convenient for performing the byte-data operations.

② BX Register:- functions as the Base Register. It is used as a pointer to a memory location. It is suitable for accessing the elements of an array from the memory.

Ex: The code for reading a word from memory location 40 in DS into register AX is as follows:

```
..
mov   bx, 40      ; BX = address of the memory location
mov   ax, [bx]    ; read memory pointed to by BX register.
```

The register BX is loaded with the offset of the memory location in DS. The register BX can be treated as two 8-bit registers BH and BL.

③ CX Register:- functions as the Count Register. Loop and Repeat instructions of 80x86 use CX register to hold repeat count value.

Ex:- To repeat a block of code 100 times, the outline of the program is as:

```
..
mov   cx, 100          ; CX = loop count
BeginOfLoop:
..
                       ; Body of the loop
loop  BeginOfLoop      ; CX ← CX-1, if CX ≠0 then BeginOfLoop.
..
```

④ DX Register:- functions as the Data Register. The register DX is the only register used as an I/O address pointer in the IN and OUT instructions.

Ex: To read a character from the port number 100 and stores in AL register.

```
..
mov   dx, 100     ; DX = port address = 100
in    al, dx      ; read from port pointed to by register DX.
```

The register DX is also involved in 16-bit multiplication and division operations as the default operand. The register DX is most efficiently used in data movement, arithmetic, and logical instructions.

II. Special Purpose Registers:-

1. Pointer and Index Registers:-

The 8086 processor has 2 pointer registers, the SP and BP and 2 index registers, the SI and DI. The pointer registers are generally used for storing the offset address of data elements stored in the stack. The index registers are used for storing index or offset of the data elements

① SI Register:- Source Index, SI, register is used as a pointer to a memory location. It is suitable for accessing the data array (elements of an array) from the memory.

Ex: Code for reading a word from a memory location 40 in DS into AX register is,

```
..
mov   si, 40      ; SI = address of the memory location
mov   ax, [si]    ; read memory pointed to by DS:SI register.
```

The register SI is loaded with the offset of the memory location in DS. The register SI is useful in accessing contiguous memory locations, such as a text string. In repeat string instructions, SI is used as a pointer to a source string element.

② DI Register :- Destination index Register, DI, register is used as a pointer to a memory location. ④

Ex: code for reading a word from memory location 40 in DS into AX register:

```
..
mov  di, 40   ; DI = address of the memory location
mov  ax, [di] ; read memory pointed to by DS : DI register.
```

The Register DI is loaded with the offset of the memory location in DS. Useful in accessing contiguous memory locations, such as text string. In repeat string instructions, DI is used as a pointer to a destination string element.

③ BP Register :- Base Pointer, BP, register is used as a pointer to a memory location is similar to the use of BX, SI, and DI registers.

Ex: The outline of the code for reading the first parameter from the stack by the C language procedure call convention is as follows:

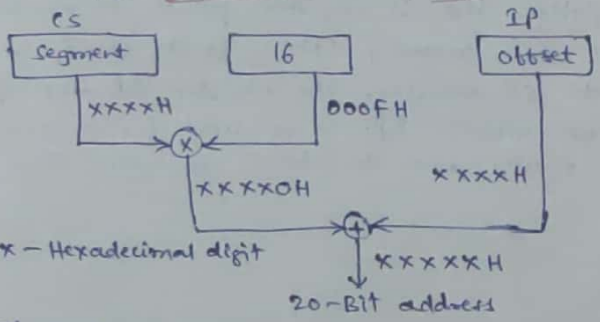
```
..
push  bp        ; save BP onto the stack
mov   bp, sp    ; assign SP to BP
mov   ax, [bp+4] ; peek for stack and put the contents into AX register
..
```

BP is designed to provide support for passing of parameters between the procedures, local variables, and other stack based allocation and operations. The BP register is usually used to access any location directly in the stack.

④ SP Register :- Stack Pointer, SP, register is dedicated for maintaining an area of memory used as a stack. The register SP always points to the top of the stack relative to SS. The process of storing data onto the stack is known as push operation. It is performed by PUSH instruction; first SP is decremented by 2 and then the data is placed into a new location pointed to by SP. The SP is decremented by 2, because the stack in 8086 system grows from higher to lower memory locations. The process of retrieving (reading) the data from the stack is known as pop operation. It is performed by POP instruction. First, the contents of the memory location pointed to by SP is read and stored in the destination and then SP is incremented by 2 so that SP points to the next element in the stack after the pop operation. The push and pop operations are performed on 16-bit operands.

2. IP Register :-

The Instruction Pointer, IP, register always points to the memory (offset) address of the next instruction to be executed. However, some instructions, such as jump and procedure calls, can cause the IP to be loaded with a new value, thereby branching to the target code (instruction). IP does not fully specify the address of the next instruction to be executed. The effective address is computed as,



Physical Address = Segment * 16 + Offset.

Fig. Computation of Physical address.

Effective address of the instruction = CS * 16 + IP = CS * 10H + IP.

where, CS is the code segments register.

The effective address generated is a 20-bit address which is a pointer to the next instruction to be executed.

3. Segment Registers :-

In 80x86 processor based systems, the memory is organized into segments of 64 Kb size. The 8086 processor is capable of addressing 1MB (2^{20}) of memory. However, 8086 processor has 16-bit pointers to the memory, but a 20-bit address is required to address the main memory of 1MB size. In 8086 processor, the address of memory has 2 parts, the segment and the offset. Each 16-bit memory pointer or a memory offset, is combined with the contents of the 16-bit segment register to form a 20-bit memory address.

① CS Register :- is the code segment register. The CS register points to the start of the 64 Kb memory block which contains the next instruction to be executed. The next instruction to be executed in the code segment is pointed to by the offset in the register IP; i.e., the segment:offset combination is indicated by CS:IP. The 8086 processor never fetches instruction from the segment other than that is defined by CS and IP. The register CS can be changed by different instructions, including certain jumps, and calls and returns (far jump, far call, or far return). The register CS cannot be loaded directly.

The register IP operates only relative to the register CS and no other register operate relative to it.

② DS Register :- is the Data Segment Register. The register DS is used to store the data set. The DS register points to the start of the data segment, a memory block of size 64 Kb, where most of the operands are stored. Normally, the memory offset is stored in the registers BX, SI and DI operate relative to DS, however, the register DI operates relative to ES in the case of string instructions

③ ES Register :- is the Extra Segment register. The register ES is not dedicated for any definite purpose. The extra segment is generally used to make an additional block of memory of size 64 Kb available for data storage. The memory access in ES is less efficient than the memory access in DS. The ES register points to the start of a memory block of size 64 Kb.

The ES is extensively used in string instructions. It is used as the destination segment, addressed by the combination of register ES:DI. It is useful in block copy, string comparision, memory scanning, and clearing block of memory locations. The register ES can point to the data segment if an extra block of memory (64 Kb) is not required.

④ SS Register :- is the stack segment register. The SS register points to the start of the memory block (64 Kb) known as the stack memory. The offset stored in SP is capable of addressing only through the SS register. The register BP also operates relative to SS. The register SS allows the register BP to be used for accessing the parameters (parameters passed through stack) and the local variables that are created on the stack.

Flag Register :-

The 16-bit flag register of the 8086 processor stores information about the status of the processor and the status of the instruction executed most recently. The 8086 flag register is divided into two parts, viz. (a) condition code or status flags and (b) machine control flags.

The flags are, (i) Overflow flag – O, (ii) Direction flag – D, (iii) Interrupt flag – I, (iv) Trap flag – T, (v) Sign flag – S, (vi) Zero flag – Z, (vii) Auxiliary flag – Ac, (viii) Parity flag – P, (ix) Carry flag – Cy, etc.

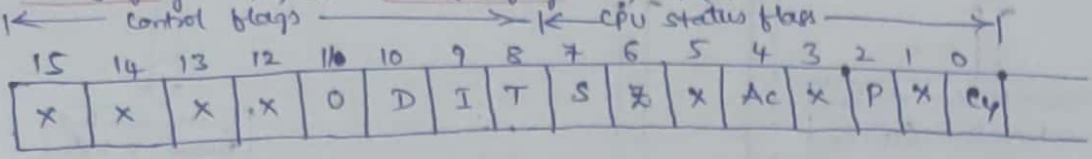


Fig. Flag Register of 8086

The description of each flag bit is as follows :

S – Sign flag :- This flag is set, when the result of any computation is negative. for signed computations, the signed flag equals the MSB of the result.

Z – Zero flag :- This flag is set, if the result of the computation or comparision performed by the previous instruction/instructions is zero.

P – Parity flag :- This flag is set to 1, if the lower byte of the result contains even no. of bits.

C – Carry flag :- This flag is set, when there is a carry out MSB in case of addition or a borrow in case of subtraction.

T – Trap flag :- If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

I – Interrupt flag :- If this flag is set, the maskable interrupts are recognised by the CPU, otherwise, they are ignored.

D – Direction flag :- This used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., autoincrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e. autodecrementing mode.

AC – Auxiliary Carry flag :- This is set, if there is a carry from the lowest nibble, i.e, bit three, during addition or borrow for the lowest nibble, i.e, bit three, during subtraction. The conditional jump instructions do not use this flag.

O – overflow flag :- This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to be accommodated in the destination register. for ex, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e, the result is of more than 7-bits in size of 8-bit signed operations and more than 15-bits in size in case of 16-bit signed operations, then the overflow flag will be set.

Pin diagram of 8086 Microprocessor

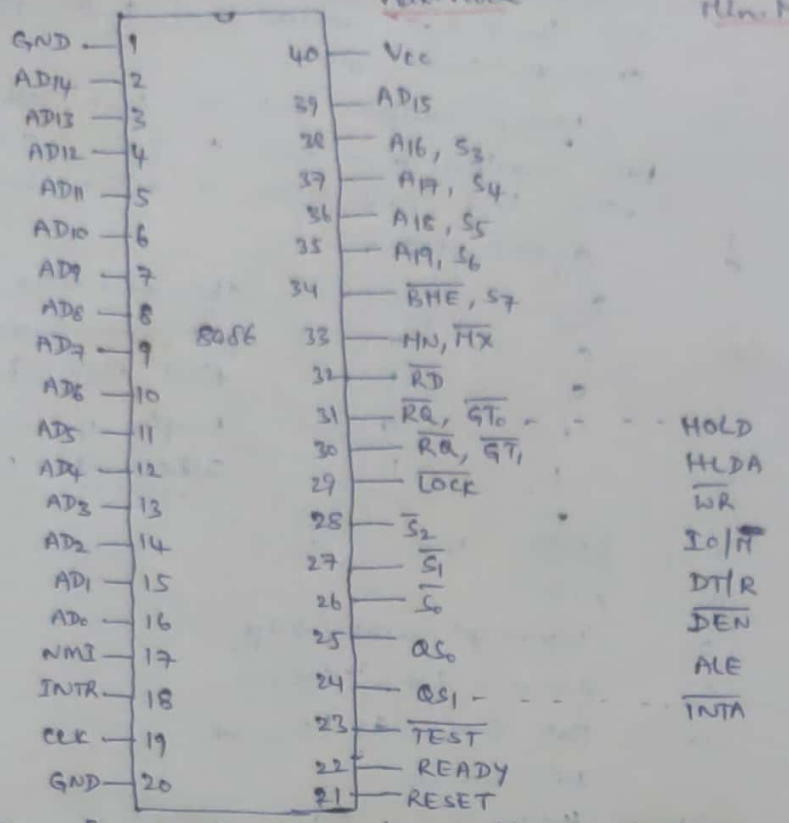


Fig. Pin configuration for 8086 Microprocessor

Symbol	Pin No.	Type	Name and function
$AD_{15}-A_{D0}$	2-16, 39	I/O	Address/Data Bus: These lines constitute the time multiplexed memory/I/O address (T_1) and data (T_2, T_3, T_W, T_4) bus. A_0 is analogous to BHE for lower byte of data bus.
A19/S6, A18/S5, A17/S4, A16/S3	35-38	O	Address/status
BHE/S7	34	O	Bus High Enable/Status
RD	32	O	Read
READY	22	I	Ready
INTR	18	I	Interrupt Request
TEST	23	I	Test
NMI	17	I	Non-Maskable Interrupt
RESET	21	I	Reset
CLK	19	I	clock → provides basic timing for processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
Vcc	40		Vcc → 5V power supply
GND	1, 20		Ground
MN/MX	33	I	Minimum/Maximum
S_2, S_1, S_0	26-28	O	Status → These signals float to 3-state off in "hold acknowledge". These status lines are encoded as,

S_2	S_1	S_0	characteristic
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port

0	1	1	Halt
1 (High)	0	0	Code Access (Instruction fetch)
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

$\overline{RQ}/GT_0$, $\overline{RQ}/GT_1$	30, 31	I/O	Request / Grant → pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.
LOCK	29	O	Lock → output indicates that other system bus masters are not to gain control of the system bus while LOCK is active LOW.
QS_1, QS_0	24, 25	O	QUEUE status.

QS_1	QS_0	Characteristics
0	0	No operation
0	1	First Byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent Byte from Queue

$\overline{M}/IO$	28	O	Status line
$\overline{WR}$	29	O	Write
$\overline{INTA}$	24	O	Interrupt acknowledge
ALE	25	O	Address latch Enable
$DT/\overline{R}$	27	O	Data Transmit / Receive
$\overline{DEN}$	26	O	Data Enable
HOLD	31, 30	I/O	Hold.

~~Basic 8086 System Bus operation :-~~

The 8086 and 8088 has been designed to work in 2 operating modes:
① Minimum Mode, ② Maximum mode.

The minimum mode is used for a small systems with a single processor and maximum mode is used for medium to large size systems, which often include two or more processors.
The 8086 is a 16-bit microprocessor with a 16-bit data bus.
The 8088 is a 16-bit " " 8-bit data bus.
The 8086 signals can be categorised in 3 groups.
* Signals having common functions in both max. and min. modes.
* Signals having special functions for min. mode.
* Signals having special functions for max. mode.

Bus structure and operation :-

Intel 8086 Architecture :-

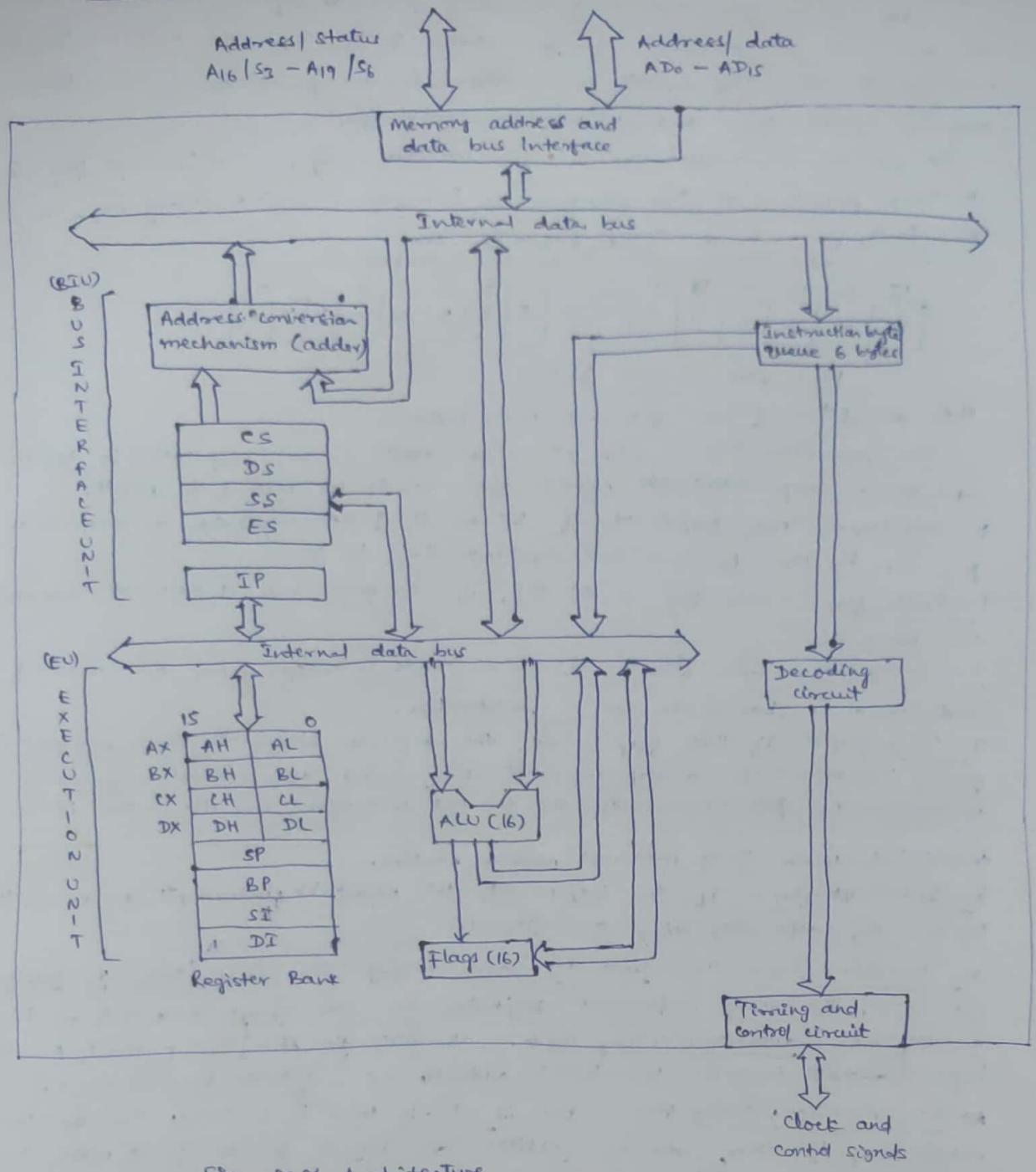


Fig. 8086 Architecture

The 8086 central processing unit is divided into 2 independent functional units. They are, 1. Bus Interface Unit (BIU), 2. Execution unit (EU).

1. Bus Interface unit (BIU) :-

The BIU is responsible for transfer of instructions, addresses, and data on the system bus to the execution unit. It handles the transfer of data between the processor, memory, and I/O devices. It includes instruction fetch, data fetch, address transfer, and computation of effective address of the memory.

The functional parts of the Bus Interface Unit are,

(i) Instruction Queue (IQ), (ii) Segment Registers, (iii) Instruction Pointer (IP).

(i) Instruction Queue (IQ) :- is of 6 bytes in length and is used to speed up the execution of programs, by prefetching 6 instructions bytes in advance from the memory. The prefetched instructions are stored in a group of high speed registers known as the instruction queue. The BIU works in parallel with the EU. The BIU fetches the instruction bytes while the EU is executing an instruction. The simultaneous operations of BIU and EU are possible only when the EU doesn't require the system bus.

The process of fetching the next instruction in advance while EU is executing the current instruction, is known as "pipelining".

(ii) Segment Registers:-

The BIU contains 4 16-bit segment registers. They are,
(a) Code Segment (CS) register, (b) Data segment (DS) register,
(c) Stack Segment (SS) register, (d) Extra segment (ES) register.

These registers are used to store the 16-bit starting address of the 4 memory segments. The BIU generates a 20-bit address using the segment and the offset components of an address. BIU can address the memory locations starting from 00000H to FFFFFH (0 to 1Mb).

Ex: Segment address → 1005 H
Offset address → 5555H
Segment address → 1005H → 0001 0000 0000 0101
shifted by 4 bit positions → 0001 0000 0000 0101 0000
Offset address → 0101 0101 0101 0101
Physical address → 0001 0101 0101 1010 0101 — 155A5 H
1 5 5 A 5

(iii) Instruction Pointer (IP):-

The register IP always holds the address of memory location (offset) of the next instruction to be executed. As the instruction is executed, the IP is advanced to point to the next instruction in the memory. Instruction pointer is also called as the program counter in other microprocessors.

2. Execution Unit (EU):-

The EU works in parallel with the BIU. It informs the BIU the location at which the next instruction/data is to be fetched. The phases of execution of the instruction are fetch, decode, Execute, and Write.

The fetch phase performs fetching of the instruction from the instruction queue.
The decode phase performs the decoding of the instruction.
The execute phase performs real (actual) operations on the datas.

The write phase performs the operation of storing the computed result at the destination.

The functional parts of the EU are,
(i) control system and Instruction decode, (ii) Arithmetic and Logic unit (ALU),
(iii) Flag Register, (iv) General Purpose Registers, (v) Stack Pointer Register,
(vi) Pointer and Index Registers.

(i) Control circuitry and Instruction decoder:- The control circuit of the EU directs all the internal operations of the processor. The instruction in the EU translates the instruction fetched from the memory into a series of actions carried out by the EU.

(ii) ALU:- Performs 8-bit or 16-bit mathematical operations such as addition, subtraction, multiplication, division, data conversion and logical operations like logical NOR, OR, or AND. It also performs register increment, decrement, and shift operations.

(iii) Flag Register:- The 8086 processor has nine, 1-bit flags to reflect the status of the CPU. The 16-bit flag register of 8086 stores the information about the status of the processor and the status of the instruction executed most recently.

(iv) General Purpose Registers:- The 8086 processor has four, 16-bit GPRs. They are AX, BX, CX and DX. Each of these 16-bit registers can be considered as two 8-bit registers distinguished as high and low order bytes, ~~of the respective 16~~ and referenced as AH, AL, BH, BL, CH, CL, and DH, DL.

(v) Stack Pointer Register:- SP (register) points to the current top of the stack.
The register BP is used as the stack pointer.
The SI and DI registers are the source and destination index registers respectively used primarily for manipulating strings.

Working Principle of 8086 :-

The 8086 CPU consists of 2 seperate processing units, the EU and the BIU, connected by a 16-bit ALU data- bus and an 8-bit queue bus. The EU obtains instructions from the instruction prefetch queue (IQ) maintained by the BIU and executes the instruction using a 16-bit ALU. Execution of the instruction involves maintainance of the CPU status, the control logic, and the manipulations of segment and offset address within the 16-bit limits. The EU accesses the memory and peripheral devices through requests to the BIU, which is the second processing unit, performing the bus operations for the EU on a demand basis. It involves generating physical address from the segment registers, the offset values, and writing out the bus results. The BIU is also responsible for prefetching instructions from the IQ wherever possible, to keep the EU busy with prefetched instructions under normal condition and for resetting the IQ when EU transfers control to another location (say JUMP). The BIU and EU operate independently enabling the 8086 to overlap the instruction fetch phase and the execution phase so as to speed up the execution of instructions.

When a microprocessor is interrupted, it stops executing its current program and calls a special routine which "services" the interrupt. The event that causes the interruption is called interrupt and the special routine executed to service the interrupt is called interrupt service routine/procedure (ISR).

Normal program can be interrupted by 3 ways:-

① By external signal, ② By a special instruction in the program,

③ By occurance of some condition.

An interrupt caused by an external signal is referred as a hardware interrupt.

conditional interrupts or interrupts caused by special instructions are called software interrupts.

8086 Interrupts :-

An 8086 interrupt can be come from any one of the 3 sources:

(i) External signal, (ii) Special instruction in the program

(iii) condition produced by instruction.

(i) External Signal (Hardware Interrupt) :-

An 8086 can get interrupt from an external signal applied to the non-maskable interrupt (NMI) input pin, or the interrupt (INTR) input pin.

(ii) Special instruction :-

8086 supports a special instruction, INT to execute special program. At the end of the ISR, execution is usually returned to the interrupted program.

(iii) Condition Produced by Instruction :-

An 8086 is interrupted by some condition produced in the 8086 by the execution of an instruction. for ex., divide by zero: Program execution will automatically be interrupted if you attempt to divide an operand by zero.

At the end of each instruction cycle, 8086 checks to see if there is any interrupt request. If so, 8086 responds to the interrupt by performing series of actions:

(i) It decrements SP by 2 and pushes the flag register on the stack.

(ii) It disables the INTR interrupt input by clearing the interrupt flag in the flag register

(iii) It resets the trap flag in the flag register

(iv) It decrements SP by 2 and pushes the current CS contents on the stack

(v) It decrements SP by 2 and pushes the current IP contents on the stack

(vi) It does an indirect far jump at the start of the procedure by loading the CS and IP values for the start of the ISR.

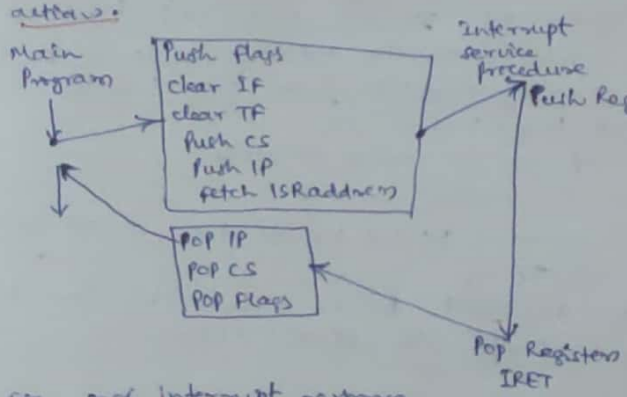


Fig. 8086 interrupt response

An IRET instruction at the end of the ISR returns execution to main program.

The 8086 gets the new values of CS and IP register from 4 memory addresses. When it responds to an interrupt, the 8086 goes to memory locations to get the CS and IP values for the start of the ISR. In an 8086 system, the first 1Kbyte of memory from 00000H to 003FFH is reserved for storing the starting addresses of ISR. This block of memory is often called the interrupt vector table or the interrupt pointer table. Since 4 bytes are required to store the CS and IP values for each ISR, the table can hold the starting addresses for 256 ISRs.

Each interrupt type is given a number between 0 to 255 and the address of each interrupt is found by multiplying the type by 4, Ex: for type 11, interrupt address is $11 \times 4 = 44 = (44)_{10} = 002CH$.

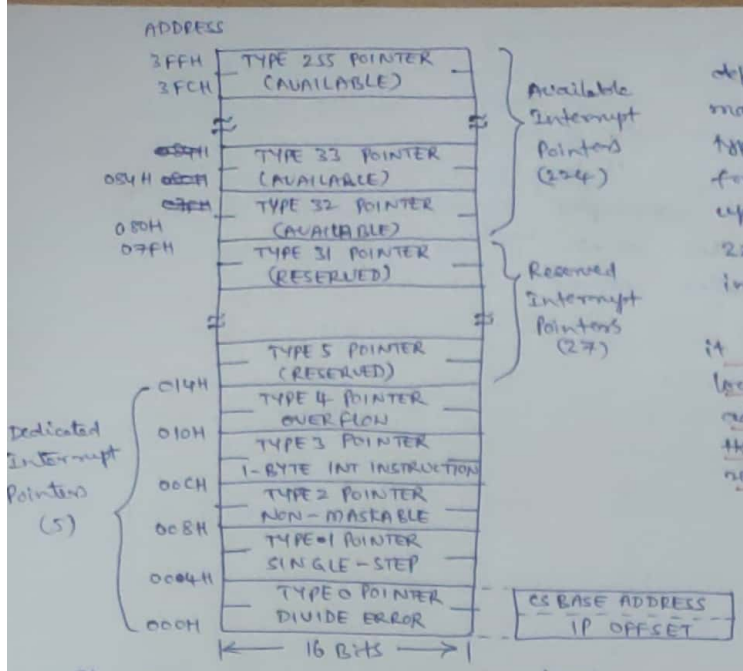


Fig: 8086 interrupt vector table (IVT)

Only first five types have explicit definitions such as divide by zero and non-maskable interrupt. The next 27 interrupt types, from 5 to 31, are reserved by INTEL for use in future microprocessors. The upper 24 interrupt types, from 32 to 255, are available for use for h/w or s/w interrupts.

When 8086 responds to an interrupt, it automatically goes to the specified location in the IVT to get the starting address of ISR. So user has to load these starting addresses for different routines at the start of the program.

8086 Interrupt types:-

① Divide by zero interrupt (Type 0), ② Single step interrupt (Type 1),
③ Non-Maskable interrupt (Type 2), ④ Break point interrupt (Type 3)
⑤ Overflow interrupt (Type 4), ⑥ Software interrupts (Type 0-255).

① Divide by Zero Interrupt (Type 0):-

When the quotient from either a DIV or IDIV instruction is too large to fit in the result register, 8086 will automatically execute type 0 interrupt.

② Single Step Interrupt (Type 1):-

This type 1 interrupt is the single step trap. In the single step mode, system will execute one instruction and wait for further direction from user. Then user can examine the contents of registers and memory locations and if they are correct, user can tell the system to execute the next instruction. This feature is useful for debugging assembly language program.

An 8086 system is used in the single step mode by setting the trap flag. If the trap is set, the 8086 will automatically execute a type 1 interrupt after execution of each instruction. But the 8086 has no such instruction to directly set or reset the trap flag. These operations can be performed by taking the flag register contents into memory, changing the memory contents so to set or reset trap flag and move the memory contents into flag register.

Assembly language program to set trap flag is:-

```
PUSHF       ; save the contents of trap flag in stack memory
MOV BP, SP  ; copy SP to BP for use as index
OR [BP+0], 0100H ; set the bit 8 in memory pointed by BP, i.e, set TF bit
POPF        ; Restore the flag register with TF=1.
```

To reset the trap flag, we have to reset bit 8. This can be done by using AND [BP+0], 0FEFFH instruction instead of OR [BP+0], 0100H.

③ Non-Maskable Interrupt (Type 2):-

This interrupt cannot be disabled by an s/w instruction. This interrupt is activated by low to high transition on 8086 NMI input pin. In response, 8086 will do a type 2 interrupt.

④ Break Point Interrupt (Type 3):-

The type 3 interrupt is used to implement break point function in the system. The type 3 interrupt is produced by execution of the INT 3 instruction. Break point function is often used as a debugging aid in cases where single stepping provides more detail than wanted. When you insert a breakpoint, the system executes the instruction upto the breakpoint, and then goes to the breakpoint procedure. In the breakpoint procedure, you can can write program to display register contents, memory & contents and other information that is required to debug you program. You can insert as many breakpoints as you want in your program.

(5) Overflow interrupt (Type 4) :-

The type 4 interrupt is used to check overflow condition after any signed arithmetic operation in the system. The 8086 overflow flag, OF, will be represented in the destination register or memory location.

For ex., if you add the 8-bit signed number 0111 1000 (+120 decimal) and the 8-bit signed number 0110 1010 (+106 decimal), result is 1110 0010 (-98 decimal). In signed numbers, MSB is reserved for sign and other bits represents magnitude of the number. In this ex., after addition of two 8-bit signed numbers, the result is negative, since it is too large to fit in 7 bits. To detect this condition in the program, you can put interrupt on overflow instruction, INTO, immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will simply function as an NOP (No operation). However, if the overflow flag is set, indicating an overflow error, the 8086 will execute a type 4 interrupt after executing the INTO instruction.

Another way to detect and respond to an overflow error in a program is to put the jump if overflow instruction (JO) immediately after the arithmetic instruction. If the overflow flag is set as a result of arithmetic operation, execution will jump to the address specified in the JO instruction. At this address, you can put an error routine which responds in the way you want to the overflow.

(6) Software interrupts (Type 0 - 255) :-

The 8086 INT instruction can be used to cause the 8086 to do one of the 256 possible interrupt types. The interrupt type is specified by the number as a part of the instruction. You can use an INT 2 instruction to send execution to an NMI interrupt service routine. This allows you to test the NMI routine without needing to apply an external signal to the NMI input of the 8086.

With the S/W interrupts, you can call the desired routines from many different programs in a system, Ex., BIOS in IBM PC. The IBM PC has an its ROM collection of routines, each performing some specific function such as reading character from keyboard, writing character to CRT. This collection of routines referred to as Basic Input/Output System (BIOS).

The BIOS routines are called with INT instructions.

We will summarize interrupt response and how it is serviced by going through following steps :-

① 8086 pushes the flag register on the stack.
② It disables the single step and the INTR input by clearing the trap flag and interrupt flag in the flag register.
③ It saves the current CS and IP register contents by pushing them on the stack.
④ It does an indirect far jump to the start of the routine by loading the new values of CS and IP register from the memory whose address calculated by multiplying 4 to the interrupt type; Ex., If interrupt type is 4, then memory address is $4 \times 4 = 16_{10} = 10H$. So, 8086 will read new value of IP from 00010H and CS from 00012H.
⑤ Once these values are loaded in the CS and IP, 8086 will fetch the instructions from the new address which is the starting address of ISR.
⑥ An IRET instruction at the end of the ISR gets the previous values of CS and IP by popping the CS and IP from the stack.
⑦ At the end, the flag register contents are copied back into flag register by popping the flag register from stack.

Maskable Interrupt (INTR) :-

The 8086 INTR input can be used to interrupt a program execution. The 8086 is provided with a maskable handshake interrupt. This interrupt is implemented by using two pins – INTR and $\overline{INTA}$. This interrupt is enable or disabled by STI (IF=1) or CLI (IF=0), respectively. When the 8086 is reset, the interrupt flag is automatically cleared (IF=0). So after reset, INTR is disabled. User has to execute STI instruction to enable INTR interrupt.

The 8086 responds to an INTR interrupt as follows:

① The 8086 first does 2 interrupt acknowledge machine cycles as shown in the fig. to get the interrupt type from the external device. In the first interrupt acknowledge machine cycle, the 8086 floats the data bus lines $AD_0 - AD_{15}$ and sends out an $\overline{INTA}$ pulse on its $\overline{INTA}$ output pin. This indicates an interrupt acknowledge cycle in ~~program~~ progress and the system is ready to accept the interrupt type from the external device. During the second interrupt acknowledge machine cycle, the 8086 sends out another pulse on its $\overline{INTA}$ output pin. In response to this second $\overline{INTA}$ pulse, the external device puts the interrupt type on lower 8 bits of the data bus.

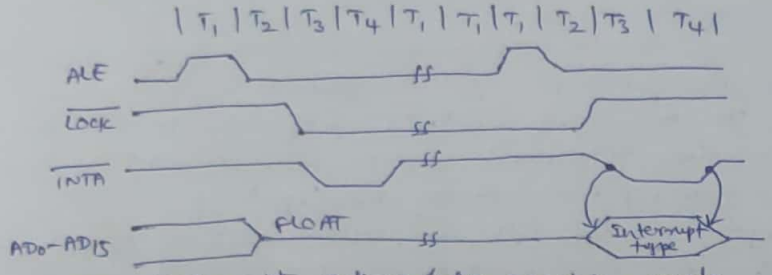


fig. Interrupt acknowledge machine cycle

② Once the 8086 receives the interrupt type, it pushes the flag register on the stack, clears TF and IF, and pushes the CS and IP values of the next instruction on the stack.

③ The 8086 then gets the new value of IP from the memory address equal to 4 times the interrupt type (number), and CS value from memory address equal to 4 times the interrupt number +2.

Interrupt Priorities :-

Software interrupts ~~have~~ have the highest priority.

Interrupt	Priority
Divide Error, Int n, Into	Highest
NMI	↓
INTR	↓
Single-step	Lowest

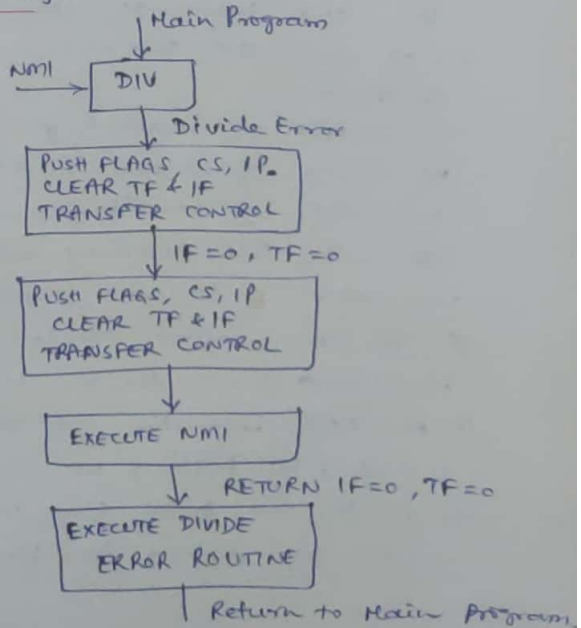


fig. Flowchart for divide error routine

31/1

UNIT-2

8086 Programming

1. Program development steps and rules.
2. Instruction set
3. Addressing modes
4. Assembler Directives
5. Writing Arithemitic and Assembler sorting Problems.

1. Program Development Rules & Steps :-

→ Major steps in Developing An Assembly Language Program.

1. Defining the Program (or) Problem.
2. Representing Program Operations
3. Finding the right instructions
4. Writing A Program

1. Defining the Problem :- Find out the problem

Ex :- Multiplication, Division, Sorting, Addition

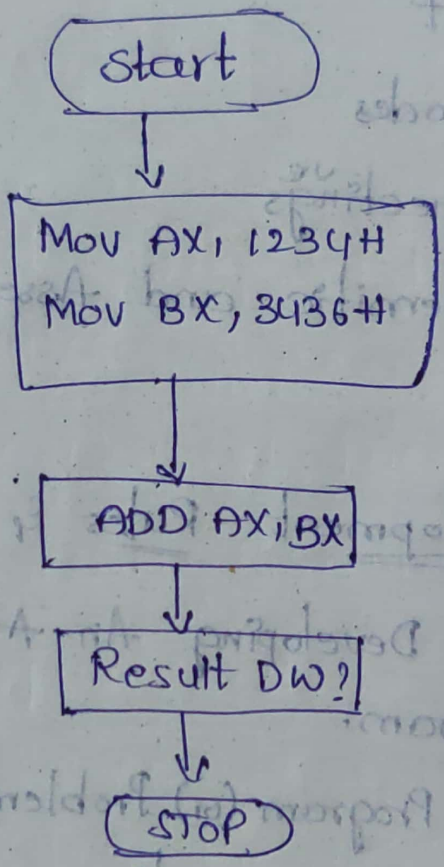
2. Representing Program Operations :-

Formula (or) Sequence of operations used to solve a programming problem is called Algorit

There are two ways of representing Algorithm:-

1. Flow chart
2. Structed Programming & SUDO CODE.

Ex:- for Flow chart



3. Finding the Right Instructions

Instructions in 8086 are mainly divided into following categories:-

1. Data Transfer Instructions (MOVE, push, POP Load)
2. Arithmetic Instructions [ADD, MUL, DIV, SUB]
3. Bit manipulation instruction (DD, DW, DB)
4. string instruction [shift, rotate, right, left, N1, N2, 13
5. Program Execution Transfer Instruction

6. Processor control system [JNZ, ZF] [JNZ - JUMP NOT ZERO] [JZ = JUMP IF ZERO]

1. Writing A Program :-

i, Initialization and Instructions : It is used to Initialize various parts of the programs like segment registers, FLAGS & programmable fork devices.

ii, Standed Program Format :- It is a tabular program format containing address, data (or) code lables, mnemonics, operands and commands.

iii, Documentation :- You should document the program.

* Program Development Tools :-

1. Editor
2. Assembler
3. linker
4. locator
5. Debugger
6. Emulator

1. Editor :- An editor is a program which allows you to create a file containing Assembly language statements for your program.

If you make a typing error the editor will let you backup and correct it.

```
C:TASM\\EDIT Filename.ASM
```

2. Assembler :- It is a device which converts Assembly Language to Machine Language (or) Binary Language.

→ It generates two files on floppy (or) Hard Disk.

→ The file generated by the assembler is called as Assembler List file.

```
C:TASM\\TASM Filename.ASM
```

3. Linker :- A program used to join link several object files into one large ~~several~~ object file.

→ while writing large programs it is usually more efficient to define large programs into small modules.

```
C:TASM\\TLINK Filename.ASM
```

4. locator :- A program used to assign specific address of where the segment of object code or to be loaded into memory.

5. Debugger : A debugger is a program which allows to load your object program into system Memory.

→ We have to run and debug it.

C:TASM\TD filename.exe

6. Emulator :- It is similar to a debugger

→ It is used to test Hardware & Software of External systems.

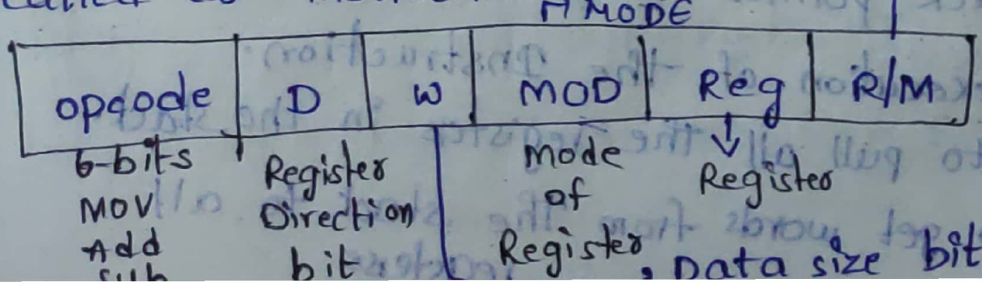
2/2/23

Instruction set :-

It is a command to the processor to perform the specific Task.

Need of a Instruction :- A Microprocessor is a multipurpose programmable device. It can perform the required operation buy giving commands to instruction without giving any command to the instruction is not a useful device. It is a machine, so the user will give command with the help of understandable language (ALP)

Instruction Format :- The representation and size of an instruction is called as instruction format.



Ex :- MOV AX, DS

MOV AX, 1234H

INSTRUCTION SET OF 8086 :-

The 8086 instructions are categorised into the following main types:-

1. Data copy/Transfer Instructions
2. Arithematic and Logical Instructions
3. Branch Instructions
4. Loop Instructions
5. Machine control Instructions
6. Flag manipulation Instructions
7. string & shift and Rotate Instructions

i, DATA COPY/TRANSFER INSTRUCTIONS:-

These types of instructions are used to transf data from source operand to destination operand

MOV :- This instruction copies a word or byte of data from source to destination

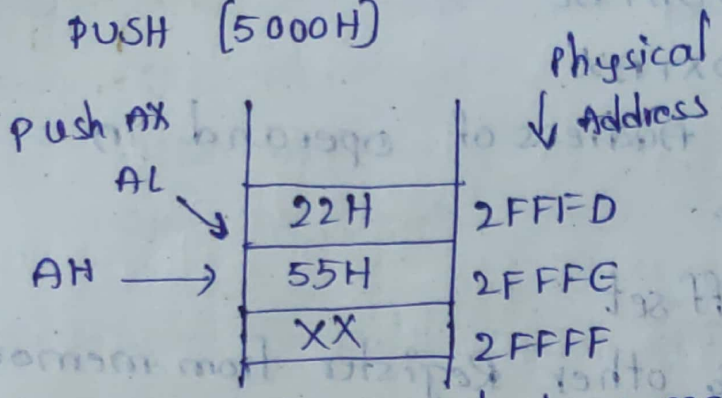
Ex:- MOV AX, 5000H
MOV AX, [SI]
MOV DS, CX
MOV CL, [357AH]

PUSH :- This instruction pushes the contents of the specified Reg/Mem location on to the stack The stack pointer is decremented by '2' after each execution of the Instruction.

PUSHA :- used to pull all the registers in the stack.

POPA :- used to get words from the stack to all registers

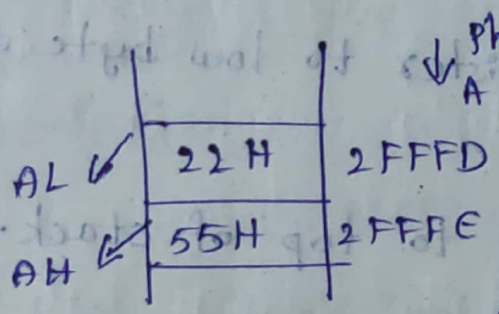
Ex:- PUSH AX
PUSH DS
PUSH [5000H]



Push data to stack memory

Pop:- This Instruction when executed, loads the specified Reg/Mem location with the contents of the memory location.

* stack pointer is incremented by '2'.



Ex:- POP AX
POP DS
POP [5000H]

Poping register content from stack Memory

XCHG:- Exchange byte (or) WORD:-

This Instruction changes the contents of the specified source and destination operands.

Ex:- XCHG [5000H], AX
XCHG BX, AX

Simple Input and o/p Port T/F's Instructions:-

IN:- copy a byte from specified port to accumulator

Ex:- IN AL, 03H

OUT :- copy a byte from Accumulator to port

Ex:- OUT D3H, AL

OUT DX, AX

LEA :- Load Effective Address of operand in specific Register.

Ex:- LEA reg, off-set

LDS :- Load DS Reg & other Register from memory

Ex:- LDS reg, mem

FLAG TRANSFER INSTRUCTIONS:-

LAHF :- Load AH with low byte of Flag Register

Ex:- LAHF

SAHF :- store AH Register to low byte of Flag Re

Ex:- SAHF

PUSHF :- copy Flag reg to top of stack.

ex:- PUSHF

[SP] ← [FLAGS]

POPF :- copy content of top of stack to Flag Register. Ex:- POPF

[Flags] ← [SP]

ARITHMETIC INSTRUCTIONS:-

The 8086 provides many arithmetic operation Addition, subtraction, multiplication and comparing two values.

Instructions to perform Addition:-

* ADD:- used to add the provide byte.

Ex:- ADD AX, 0100H
ADD AX, BX
ADD AX, [SI]

2. ADC:- used to add carry (ADC Add with carry)

Ex:- ADC AX, BX
ADC AX, [SI]
ADC [AX, 5000]

3. INC:- used to increment the provided byte/word by 1.

Ex:- INC AX
INC [BX]

4. AAA: used to adjust ASCII after addition.

5. DAA: used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction:-

* SUB:- used to subtract the byte from byte/word from word. Ex:- SUB AX, 0100H
SUB AX, BX

2. SBB: used to perform subtraction with borrow

Ex:- SBB AX, 0100H
SBB AX, BX

3. DEC: used to decrement the provide byte/word by 1.

Ex:- DEC AX
DEC [5000H]

4. NPG : used to negate each bit of the provi byte/word and add 1/2's complement.

5. INC : This can be increases the contents of the specified Register. Ex:- INC AX
INC [BX]

6. CMP : used to compare 2 provided byte/word

Ex:- CMP BX, 0100H
CMP AX, 0100H
CMP BX, CX

7. AAS : used to adjust ASCII codes after subtrac

8. DAS : used to adjust decimal after subtraction

Instructions to perform multiplication:-

1. MUL :- used to multiply unsigned byte/word by word.

2. IMUL :- used to multiply signed byte by byte/ word by word.

3. AAM : used to adjust ASCII codes after multiplication.

Instructions to perform division:-

1. DIV :- used to divide the unsigned word by byte (or) unsigned double word by wor

2. IDIV :- used to divide the signed word by byte (or) signed double word by word.

AAB: used to adjust ASCII codes after division.

CBW:- used to fill the upper byte of the word with the copies of sign bit of the lower byte

CWD: used to fill the upper word of the double word with the sign bit of the lower word.

DAS: Decimal Adjust After Subtraction.

Logical Instructions:

The instructions of this group perform logical AND, OR, XOR, NOT and TEST operations.

AND: This instruction bit by bit ADS the source operand that may be an immediate Register/memory.

EX:- AND AX, 0008H
AND AX, BX

OR:- It performs bit by bit logical OR operation of two operands and places the result in the specified destination.

XOR:- It performs bit by bit XOR operation of two operands and places the result in the specified destination.

NOT :- Take one's complement of the content of a specified register (or) memory locations.

TEST :- It perform logical AND operation of a specified operand with another specified operand.

Branch Instructions:-

1. These instructions transfer control of execution to the specified address. All the call, Jump, interrupt and return instructions belong to this category.
2. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be Transferred.
3. Branch instructions transfer the flow of execution of the program to a new address specified in the instruction directly (or) indirectly.
4. The Branch instructions are classified into 2 types
   1. unconditional Branch Instructions
   2. conditional Branch Instructions

1. unconditional Branch Instructions:

In unconditional control Transfer instruction the execution control is transferred to the specified location independent of any status (or) condition. The CS and IP are unconditionally modified to the new CS and IP

CALL: The instruction is used to call a subroutine from a main program. Address of procedure may be specified directly or indirectly.

RET [Return from the procedure]: Returns program execution from a procedure to the next instruction

IRET [Return from ISR]: Returns program execution from an interrupt service procedure to main program

INT N [Interrupt Type N]: when INT N instructions is executed the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in [0000] segment.

INTO [Interrupt on overflow]: This instruction is executed, when the overflow flag OF is set. This is equivalent to a type 4 interrupt instruction.

Conditional Branch Instructions:-

JZ: Transfer execution control to address label, if ZF=1

JNZ: Transfer execution control to address label, if ZF=0

DJNZ: Do not jump if not zero.

Loop Instructions:-

* The loop, LOOPNZ and loopz instructions belong to this category. These are useful to implement different loop structures.

*Loop :- Jump to defined label until CX = 0

*LOOPNZ : decrement CX register and JUMP if CX ≠ 0 and ZF = 0

*LOOPZ / LOOPE :- decrement CX Register and Jump if CX ≠ 0 and ZF = 1.

Here CX = Register

ZF = Zero Flag

Machine control instructions :-

*These instruction control the machine status.

NOP, HLT, WAIT and LOCK.

WAIT - wait for test input pin to go low

HLT - halt the process

NOP - No operation

LOCK - Bus lock instruction prefix

ESC - Escape to external device like NDP

Flag manipulation instructions :-

All instructions which directly effect the flag register belong to this category. This instructions

CLD, STD, CLI, STI etc.

CLC - clear carry Flag

CMC - complement carry Flag

STC - set carry Flag

CLD - clear direction Flag

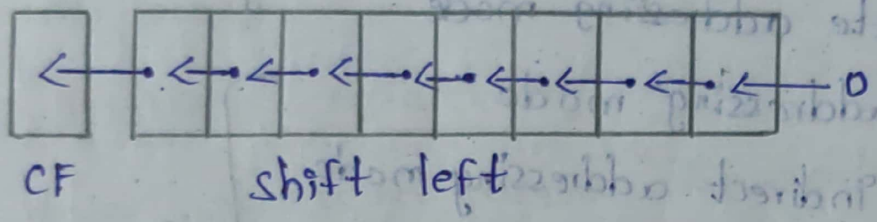
STD - set direction Flag

CLI - clear Interrupt Flag

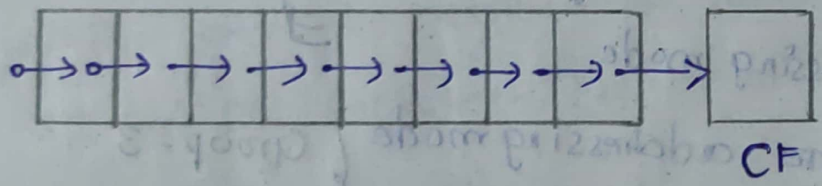
STI - set interrupt Flag.

Shift and Rotate Instructions :-

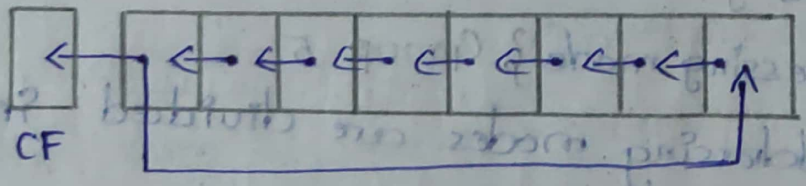
SAL/SHL :- This instruction shifts each bit in the specified destination to the left and '0' is stored at LSB position. The MSB is shifted into the carry Flag.



SAR/SHR :- shift Right

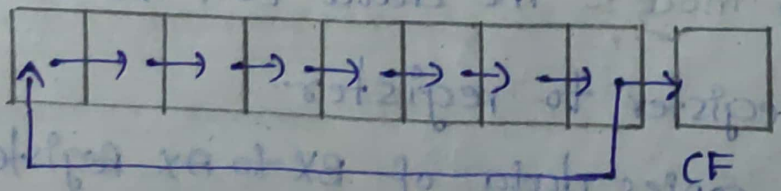


ROL : Rotate left



MSB bit is either new LSB (or) NEW CF

ROR :- Rotate Right



Addressing Mode :-

→ Every instruction of a program has to be operate on data. The method of specifing data to be operated by an instruction is called as Addressing mode. There are "12" Adressing mode are following can be classified into "5" groups.

1. Register addiessing mode } Group-1
2. Immediate addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Base addressing mode } Group-2
6. Indexed addressing mode
7. Base indexed addressing mode
8. string addressing mode
9. direct Io core addressing mode } Group-3
10. Indirect addressing mode
11. Relative addressing mode } Group 4
12. Implied addressing mode } Group 5

Here the 12 addressing modes are divided into 5 groups are :-

Register addressing mode :- The data can be transfer (or) copies from register to register.

Ex:- MOV AX, BX [It copies data of BX to AX Register]

MOV BX, CX [It copies data of CX to BX Register]

Immediate Addressing mode:- In this addressing mode, immediate data is a part of instruction and appears in the form of successive byte (or) byte

Ex:- MOV AX, 0050H [0050H is a immediate data and it is moved to register AX]

Direct Addressing mode:- In the direct addressing mode, a 16-bit address is directly specified in the instruction as a part of it.

Ex:- MOV AX, [1000H].

Register Indirect Addressing mode:-

In this addressing mode, the address of the memory location which contains data (or) operand is determined in an indirect way using offset registers. The offset address of data is the either BX (or) SI (or) DI register. The default segment register is either DS (or) ES:

Ex:- MOV AX, [BX]

Base addressing mode:- In this effective address is the sum of base register and displacement.

Ex:- MOV AL, [BP+0100]

Indexed Addressing mode:- In this type of addressing mode the effective address is sum of index register and displacement. Ex:-

MOV AX, [SI+2000]
MOV AL, [DI+3000]

Base indexed addressing mode :-

In this the address is sum of base and index register.

Base register : BX, BP

Index register : SI, DI

Ex :- MOV AL, [BP+SI]

MOV AX, [BX+BI]

string addressing mode :-

This addressing mode is related to string instru

In this value of SI and DI are auto incremente and decremented depending upon the value of directional flag.

Ex :- MOVSB

MOVSW

Direct IO core addressing mode : This addressin mode is related with the input & output operati

Ex :- INA, 45

OUTA, 50

Indirect Addressing mode :- In this addressing mo port is available in register. DX before executing I/O - Input Output operation.

Ex :- IN AX, [DX]

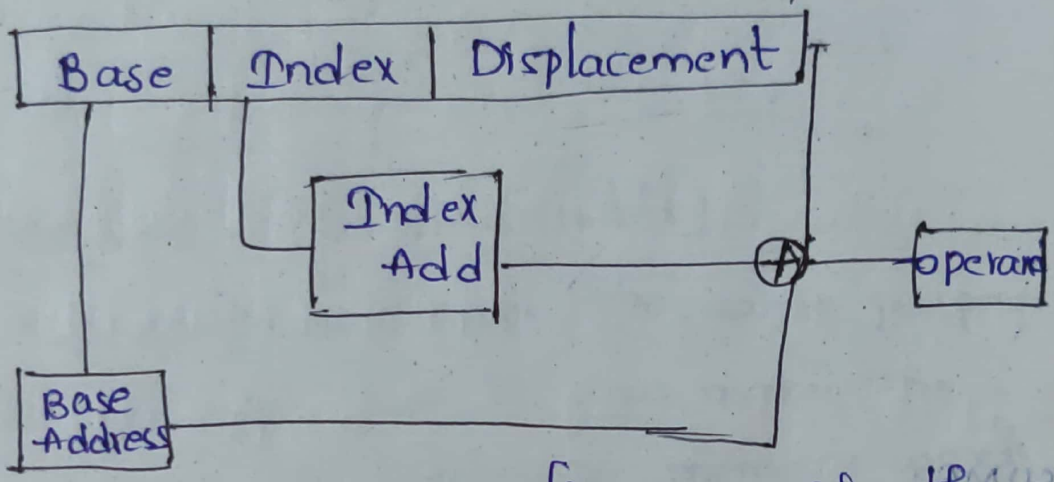
Out, [DX], AX

Relative Addressing mode: The data is available at an effective address formed by adding 8 bit (or) 16 bit displacement with content of any register BX, BP, SI & DI.

→ offset address

MOV [CX (BX + DI + 04)]

(BX + DT + 08)



Implied Addressing mode: [itself specify the operand]

In this addressing mode the instruction is predefined. In this mode the register are used for specifying operands but these registers predefined

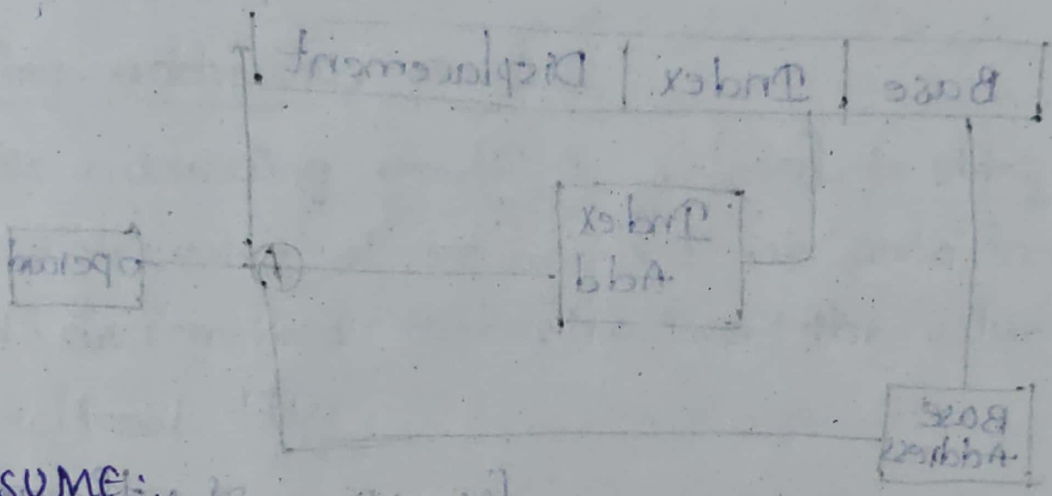
Ex: MUL BX

DIV CX

DAA

AAA.

Assembler directives



ASSUME:-

1. shows the segment name to the assembler.
2. It provides information to the assembler regarding the name of the program (or) data segment for that particular segment.
3. The directive specifies that the instruction of the source program is stored in logical segme

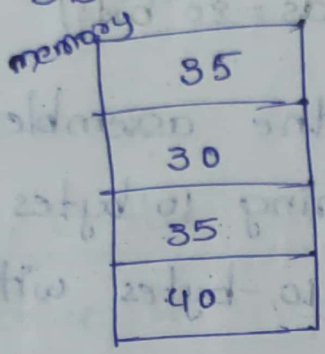
Ex:- ASSUME CS :- DONE

ASSUME CS : CODE, DS:DATA, SS:STACK

DB [Define byte] :- [8 bits]

→ The DB directive is used to reserve byte (or) byte of memory locations in the available memory.

Ex:- MARKS DB 35H, 30H, 35H, 40H

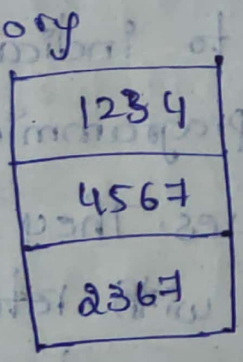


DW [Define word] : [2 byts - 16 bit]

→ The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes.

Syntax : variable name DW initialization values

Ex: WORDS DW 1234H, 4567H, 2367H



DQ [Define Quad word] : (data stored in memory)

→ This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the

specified variable and may initialize it with specified values. [4 words - 8 bytes - 64 bits]

Syntax: Name of variable DQ initialize values

Ex: Data1 DQ 123456789ABCDEF2H.

DT [Define Ten bytes]: [5 words = 80 bits]

The DT directive directs the assembler to define the specified variable requiring 10 bytes for its storage and initialize the 10-bytes with specific

Syntax: Name of variable DT initialize values.

Ex: Data1 DT 123456789ABCDEF34567H.

END [END of Program]: ALP [Assembly language Program]

→ The END directive marks the end of an ALP
→ The statement after the directive END will be ignored by the assembler. The END should "END"

ENDP [END of procedure]:

* The ENDP directive is used to indicate the end of procedure in the AL programming the subroutines are called procedures. They may be independent program modules with returns particular values to calling programs.

Ex: procedure start

:

start ENDP

SKIP: JNC DX

:

SKIP END

ENDS [End of segment]:

The ENDS directive is used to indicate the end of logical segment / It appears will prefix to mark the end of segment.

Ex:- DATA SEGMENT code segment

:

DATA ENDS code ends

DD [Define Double word] : (4 bytes, 32 bits)

This directive is used to declare a variable of type double word (or) restore memory locations which can be accessed as type double word.

EQU (Equate):

This directive EQU is used to assign a label with a value (or) a symbol. The use of this directive is just to reduce the recurrence of the numerical values (or) constants in a program code. Ex: Marks EQU 40H, 60H, 70H, 80H.

Extern (External):

→ It is used to tell the assembler that the name (or) label following the directive are some other assembly module.

Ex:- Module 1 statement

external Factorial FAR

Module 1 ENDS.

Public: The directive is used to instruct the assemble that a specified name (or) label will be accessed from other modules.

Ex: Module 1 statement

Public Factorial FAR

Module 1 ENDS

ORG (origin):

→ The ORG statement changes the starting offset address of the data.

→ It allows to set the location counter to a desired value at any point in the program.

Ex: code segment

ORG 5000H

Note: i.e., The code segment starts from 5000

PTR (Pointer): The pointer operator, used to declare the type of label, variable (or) memory operand. Ex:- MOV AL, BYTE PTR [SI]

MOV AL, WORDPTR [2000.H]

Far PTR (far Pointer): This directive indicates the Assembler that the label "far PTR" is not available within the same segment.

Ex:- JMP FAR PTR LABEL

UNIT-III

8051 Micro controller

→ Architecture intel 8051 Microcontroller

→ Pin Diagram

→ Input and output codes and circuits, memory organization, counters (or) Timers.

→ Serial data input (or) output

→ Interrupts.

Micro Controller:- [in built components]

→ A single chip (or) A CPU with all pheripherals like RAM, ROM, I/O ports, timers, ADC, etc, on same chip

Analog Digital circuit

Ex:- Intel 8051
Motoral 6811
zilog's Z8
PIC 16X etc.

Evaluation of Microcontroller:-

→ First MC name is TMS1000 was introduced by Texas instrument in 1974

→ In 1976, Motorola designed a MP chip called 6801.

1974 → Texa's - TMS 1000

1976 → Motoral - 6801

later → Intel - 8048

Specification with CPU, 1 kB ROM, 64 bytes of RAM, 27710 pins

1980 → Intel 8051 [4KB of ROM, 128B of RAM, Two 16-bit Timer, 32-I/O pins]

1982 Intel 8096 MC [16 bit]

Intel 80196 [16 bit MC]

1985 PickMC16C64 [8 bit MC]

Motoral MPC 505 [32 bit]

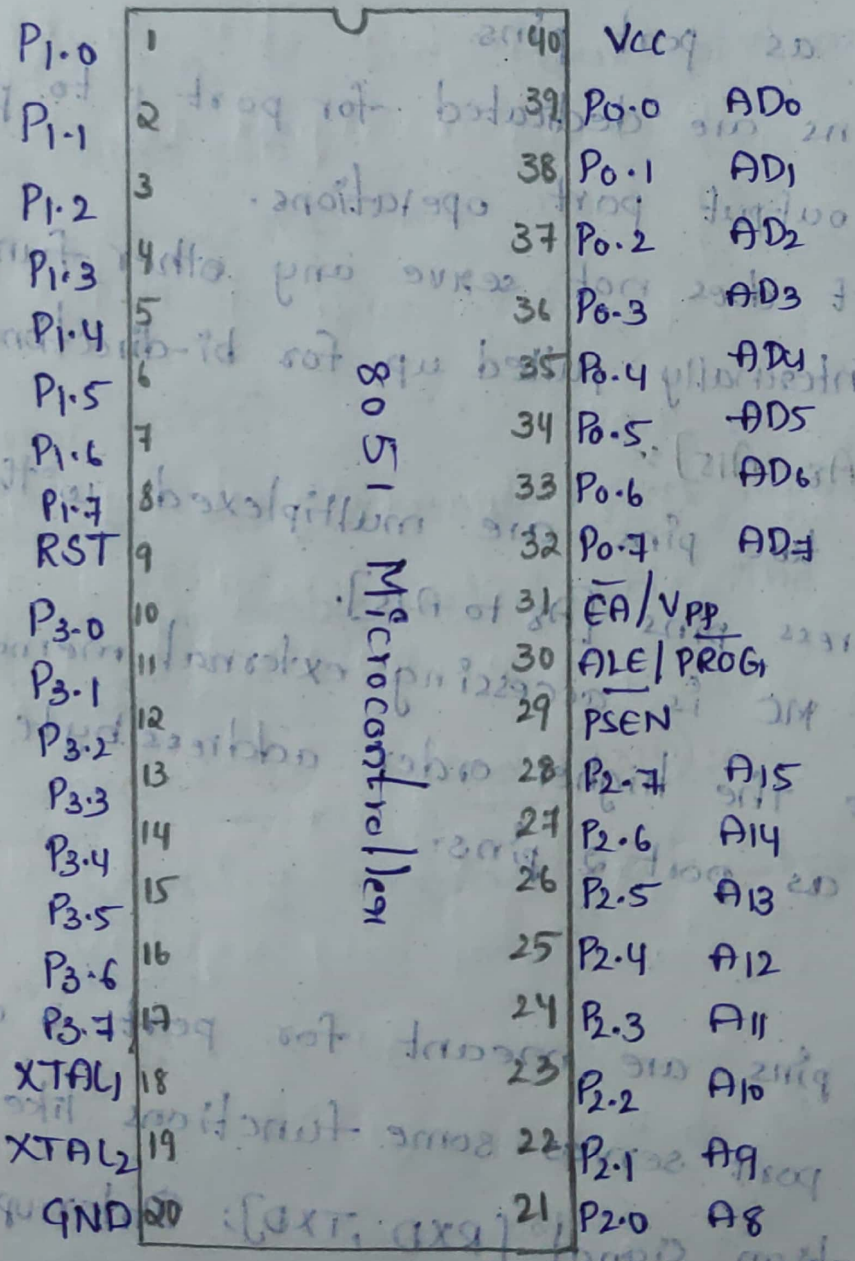
IBM company 403GA [32 bit] → 1983

In recent times

Pin Diagram of 8051 Microcontroller :-

→ The 8051 MC is available as a 40 pin dip CHIP [dip - dual in package] and it works at 5vdc

→ Among 40 pins a total of 32 pins are alloted for the 4 parallel ports P_0, P_1, P_2, P_3 i.e, each occupies 8 pins remaining are Vcc, GND, RESET, XTAL1, XTAL2, EA, ALE, -etc.,



Signal Description:-

Port 0.0 - 0.7 [AD_0 - AD_7]:-

→ The port zero pins multiplexed with address/data Pin

→ If Micro controller is accessing external memory these pins will acts as address/data pins. otherwise they are used for port zero pins.

$P_{1.0}$ - 1.7 :-

→ It acts as port pins

→ These pins are dedicated for port 1 to perform input or output port operations.

→ These port does not serve any other functions.

→ It is internally pulled up for bi-directional I/O port

$P_{2.0}$ - 2.7 (A_8 - A_{15}) :-

→ The port two pins are multiplexed with higher order address pins [A8 to A15].

→ when the MC is accessing external memory these pins provide the higher order address byte otherwise they act as port 2 pins.

$P_{3.0}$ - 3.7 :-

→ This 8 pins are meant for port 3 operations and this port serves some functions like serial communication signal [RXD, TXD]; Interrupts [$\overline{INT_0}$, $\overline{INT_1}$]; Timers [T_0, T_1]; control signals [$\overline{WR}$, $\overline{RD}$].

XTAL1 and XTAL2 :-

→ These pins are used for interfacing an external crystal oscillator to get the system clock.

GND [GROUND]:-

→ It grounded the internal circuit.

Vcc :- This pin is used to provide the power supply to the circuit.

RESET :-

→ The RESET pin is an input pin and it is an Active high pin.

→ If RST=1 the MC will reset and Terminates all activities.

EA [External Access]:-

→ which stands for External Access input.

→ It is used to enable (or) disable the external memory interfacing.

EA is connected to Ground (GND):-

→ when MC accessing program code stored in external memory.

EA is connected to Vcc :-

→ when MC accessing program code stored in chip Memory.

ALE [Address latch Enable]:-

→ ALE stands for Address latch Enable.

→ It is used to demultiplex the address & da signals of port.

→ If ALE = 1 the bus will acts as Address bus.

→ If ALE = 0 the bus will act as Data bus.

PSEN [Program store Enable]:-

→ PSEN stands for program store Enable.

→ It is used to read a signal for the extern program memory.

→ This is an output pin and it active low sig

Architecture of 8051:-

→ 8051 is an 8 bit MC designed by an inted in 1981.

→ It has 4KB of ROM and 128 bit of RAM.

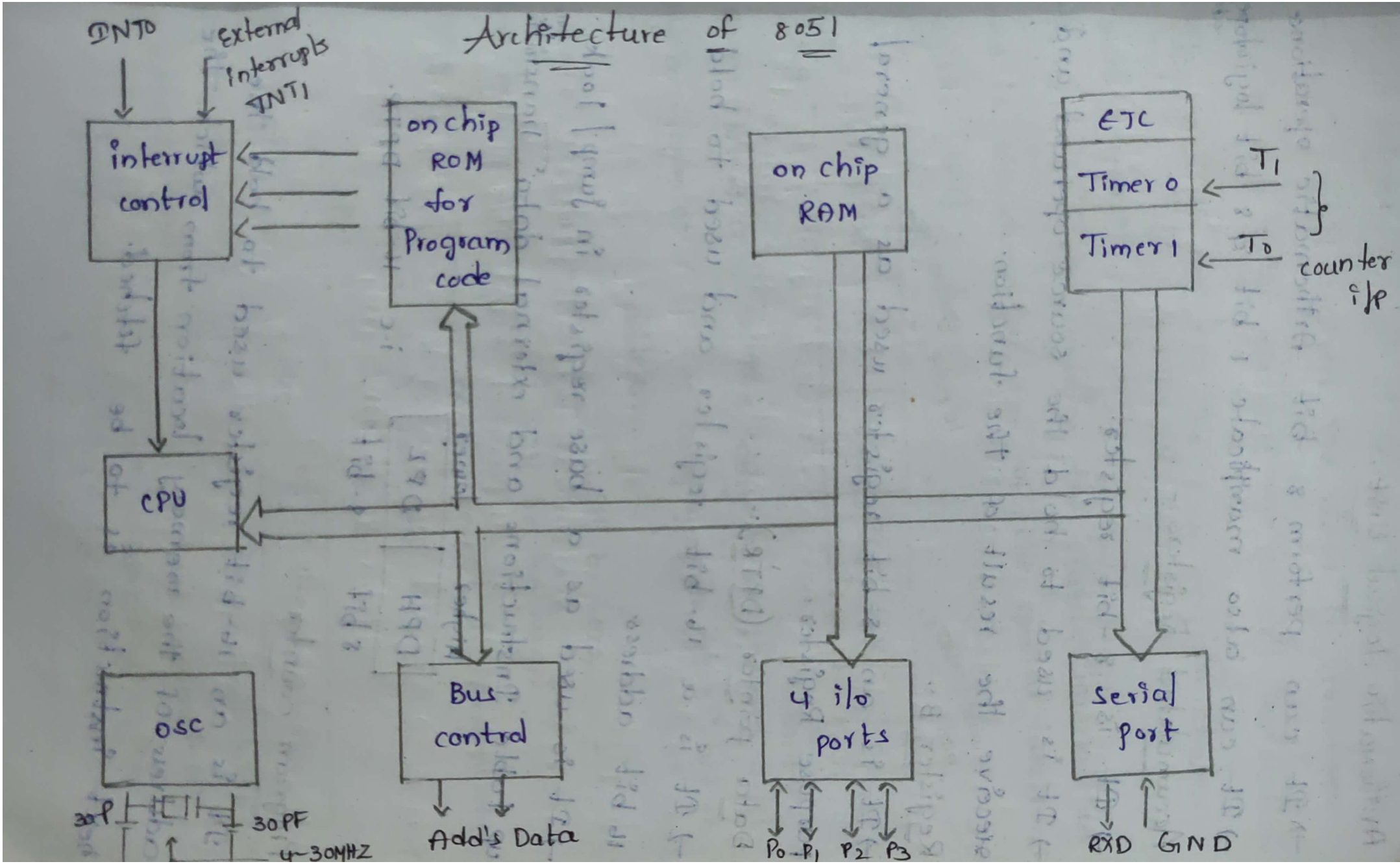
CPU:-

→ Arithematic Logic Unit

→ Registers A, B, PSW, temporary registers.

→ 16 bit program counter TC.

→ Data pointer (DPTR) and SP.



Arithematic logical Unit:-

→ It can perform 8 bit Arithematic operation

→ It can also manipicate 1 bit & 8 bit logical

Accumulated Registers:-

→ It is 8-bit register.

→ It is used to hold the source operand and receive the result of the function.

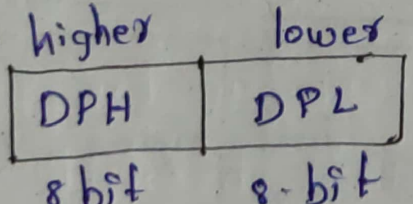
Register B:-

→ It is an 8-bit register used as a general purpose Register.

Data pointer (DPTR):-

→ It is a 16-bit register and used to hold 16 bit address.

→ It is used as a base register in jump / look up table Instructions and external data Transmit



i.e 16-bit DPTR.

Program counter:-

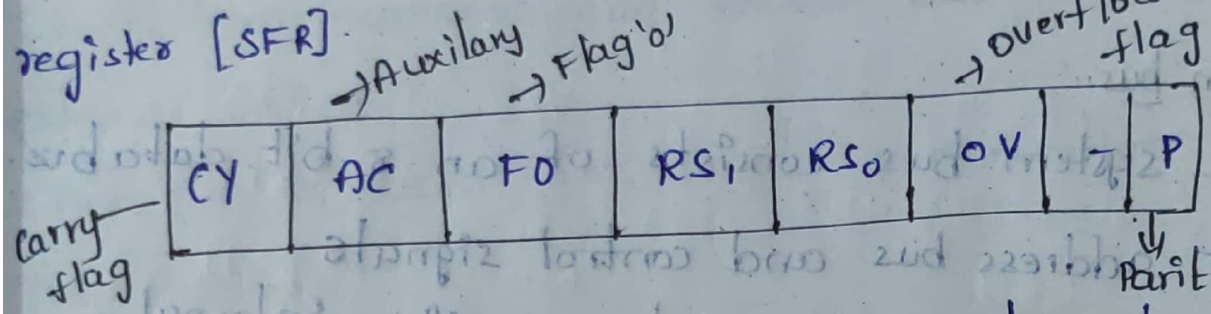
It is an 16-bit register used to hold the address of the memory location from which the next instruction is to be fetched.

stack pointer:-

This is 8 bit register the data is stored on to the stack using pull(or) push (or) all instructions

Flag (or) PSW register:- The psw program status word register.

The set of flags contains the status information and is considered as one of the special function register [SFR].



carry flag => CF=1 if the carry occurs beyond 7-bit

Auxilary flag => AF=1 if the carry generate at 5th bit of addition.

Flag "0" => F0=1 when result will be zero.

Overflow flag => OV=1 result will be too large.

Parity Flag => P= means no.of present in Accumulator

P=0 no.of '1' is even

P=1 no.of '1' is odd

RS0, RS1 { register Bank select signals }

RS1	RS0	Bank selection
0	0	Bank 0 => 00H to 07H
0	1	Bank 1 => 08H to 0FH
1	0	Bank 2 => 10H to 17H
1	1	Bank 3 => 18H to 1FH

Special Function Registers:

→ This is a set of special function register using their respective address using respective a

→ In 8051 all access to the input output ports, C registers, Timer counters, logic control registers can be done through the SFR's.

→ SFR's located from 80H to FFH.

System bus:

→ The system bus consists of an 8 bit datab 16-bit address bus and control signals.

→ It is used to connect all the internal devices to the CPU.

I/o ports:

→ 8051 has 32-bidirectional, I/o pins, whic are organized as 4-parallel 8-bit I/o ports.

Port 0 → It act as dual role as I/o ports & multiplexed to $AD_0 - AD_7$.

Port 1 → only for I/o.

Port 2 → I/o ports and higher order Address pins [A8-A15]

Port 3 → I/o ports & multifunctional.

Interrupts:-

→ 8051 provides "5" interrupts source.

→ External Hardware interrupts = INT0, INT1

→ External Timer interrupts => TF0, TF1

→ serial port interrupt => RI/TI
[Reciever INT/Transmit].

Serial port :- The serial port of 8051 is full-duplex i.e, it can transmit and receive data to an external device.

→ And it is used for serial communication.

Time Register:-

There are two [16-bit] registers can be accessed as lower byte and upper byte.

TL0 → Timer "0" lower byte accessed.

TH0 → Timer "0" Higher

TL1 → Timer "1" lower

TH1 → Timer "1" Higher

→ All these registers can be accessed using the address alloted to them which is in SFR and the range is in between 80H-FFH.

Oscillator:- The circuit generates the basic Timing clock signal for the operation of circuit using crystal oscillator.

Memory Timing and control:- This is necessary Timing and control signals required for the internal operation of the circuit.

SFR Registers and their Addresses

S.No	Register Name	Bit Address	Address
1	Accumulator (A)	Yes	0E0H
2	B	Yes	0F0H
3	PSW	Yes	0D0H
4	stack Pointer	No	81H
5	DPH	No	82H
6	DP_2	No	83H
7	P_0	Yes	80H
8	P_1	Yes	90H
9	P_2	Yes	0A0H
10	P_3	Yes	0B0H
11	IP	Yes	0B8H
12	IE	Yes	0A8H
13	TMOD	No	89H
14	TCON	Yes	88H
15	TH0	No	8CH
16	TL0	No	8AH
17	TH_1	No	8DH

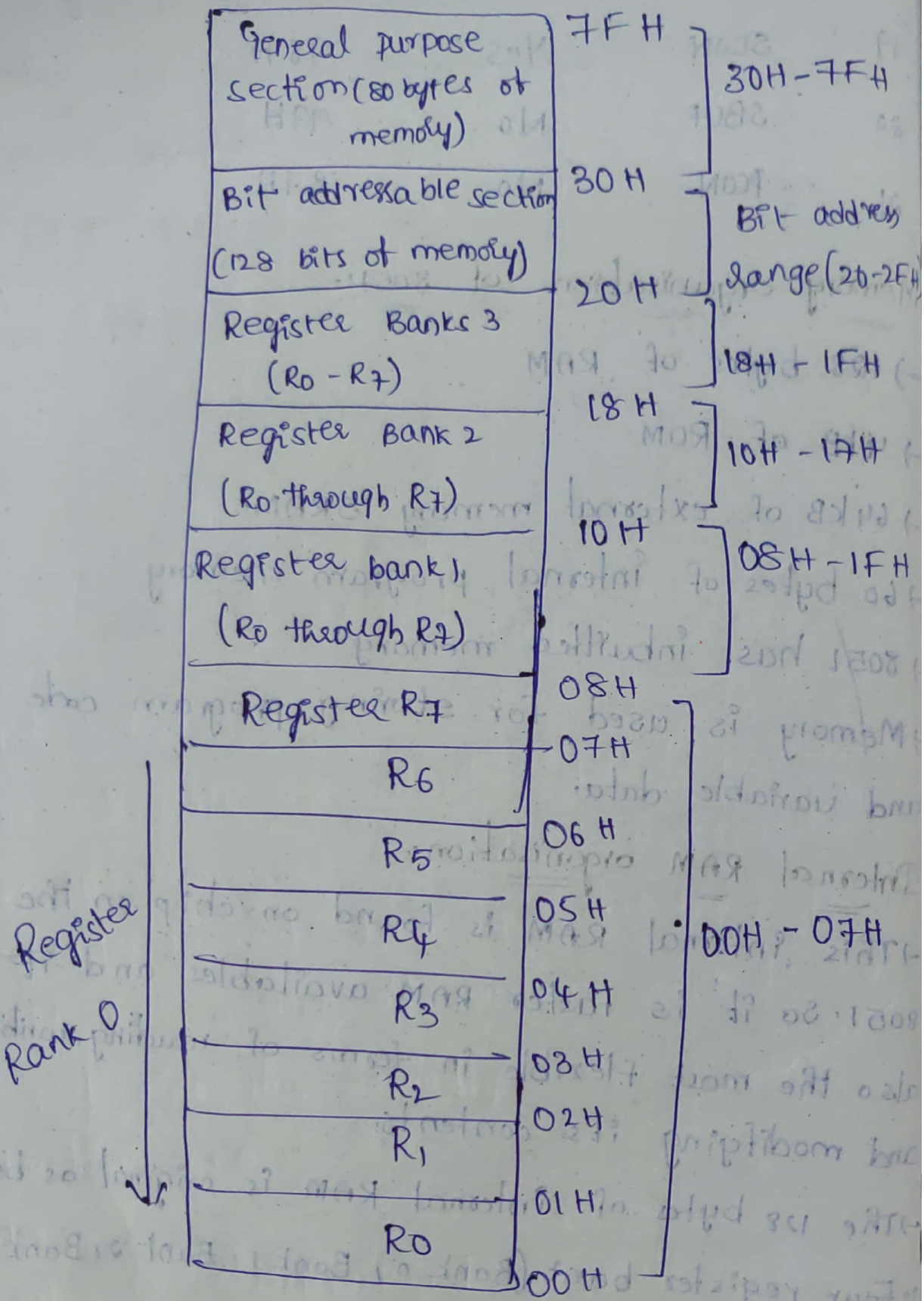
18	TL_1	No	8BH
19	SCON	Yes	98H
20	SBUF	No	99H
21	PCON	No	87H

Memory organisation of 8051:-

→ 128 bytes of RAM.
→ 4kB of ROM.
→ 64kB of external memory SRAM.
→ 60 bytes of internal program memory
→ 8051 has inbuilted memory.
→ Memory is used for storing program code and variable data.

Internal RAM organisation:-

→ This internal RAM is found on-chip on the 8051. So it is faster RAM availables and it also the most flexible in terms of reading, writing and modifying its contents.
→ The 128 bytes of internal RAM is original as below
1. Four register bank (Bank 0, Bank 1, Bank 2, Bank 3) each of 8-bits (total 32 bytes). The de-fault bank register is bank 0. The remaining banks are selected with the help of RS_0 & RS_1 bits of PSW register.



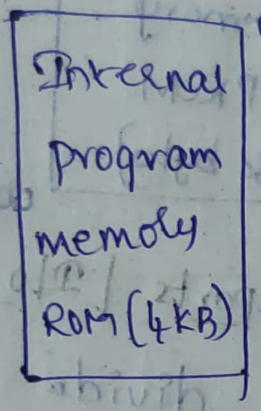
2. 16 bytes of bit addressable area.

3. 80 bytes of general purpose area (scratch pad memory). This area is also utilised by the

Micro controller as a storage area for the operating stack. 32-byte of RAM from address 00H to 1FH are used as working registers. The registers are named as R_0-R_7

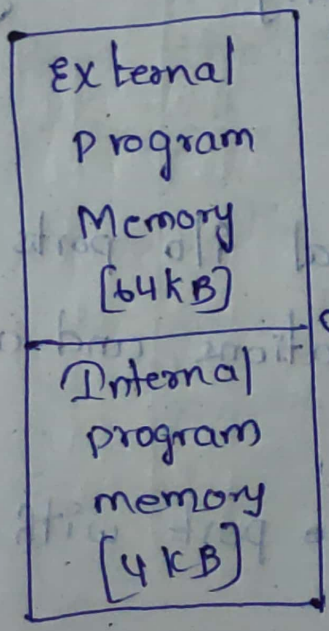
* Each register can be addressed by it name (or) in RAM address

4kB of ROM



* EA (External access) pin of 8051 is connected to the logic high.

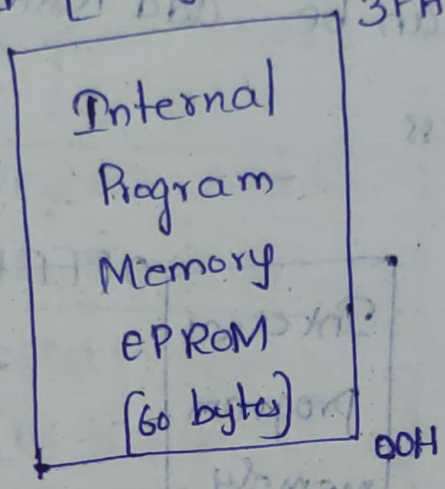
64 kB of external memory SRAM



[External Access] pin of 8051 is the connected to the logic high.

60-bytes of Internal program Memory :-

→ [External Access] pin of 8051 is connected to the ground [zero].



Input and Output ports [I/o ports] and circuit :-

→ The I/o ports are divided into four ports in 8051 Microcontroller i.e. : port 0, port 1, port 2, port

→ In order to make them input, all the ports must be set i.e.. a high bit must be sent to all the port pins. This normally done by the instruction "SET B".

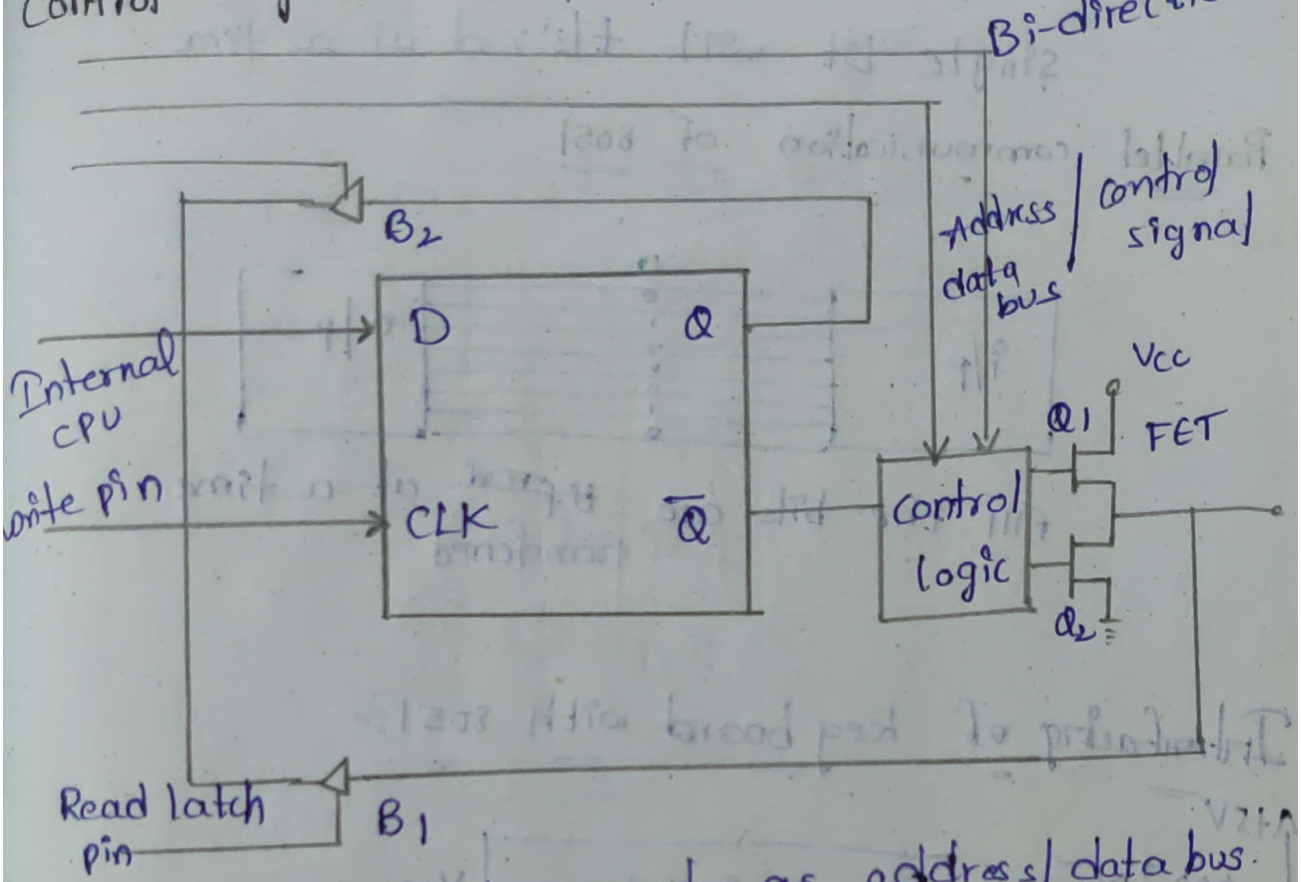
Port 0 :-

→ port 0 is a bidirectional i/o ports used for input and output operations and also holds address and data lines.

→ port 0 is an 8-bit I/o port with dual purpo

→ If external memory is used, these ports pins are used for the lower address byte addres data [AD0-AD7]

Control signal access data:

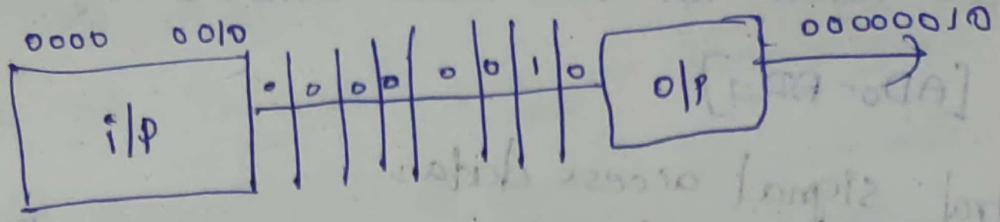


→ port 0 can also be used as address/data bus. [AD0-AD7], allowing it to be used for both address and data bus.

→ when connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes the address and data through port 0 to save the pins.

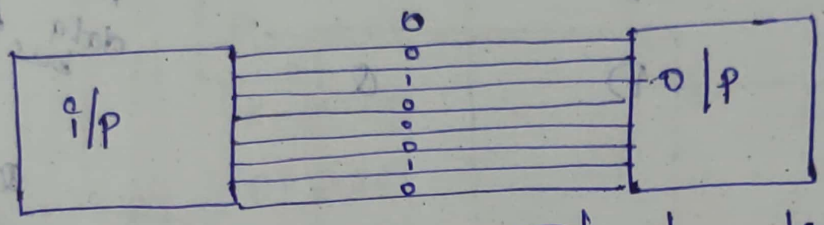
→ when ALE=0, it provides data AD0-AD7, and when ALE=1 it provides address and data with the help of latch.

Serial communication of 8051



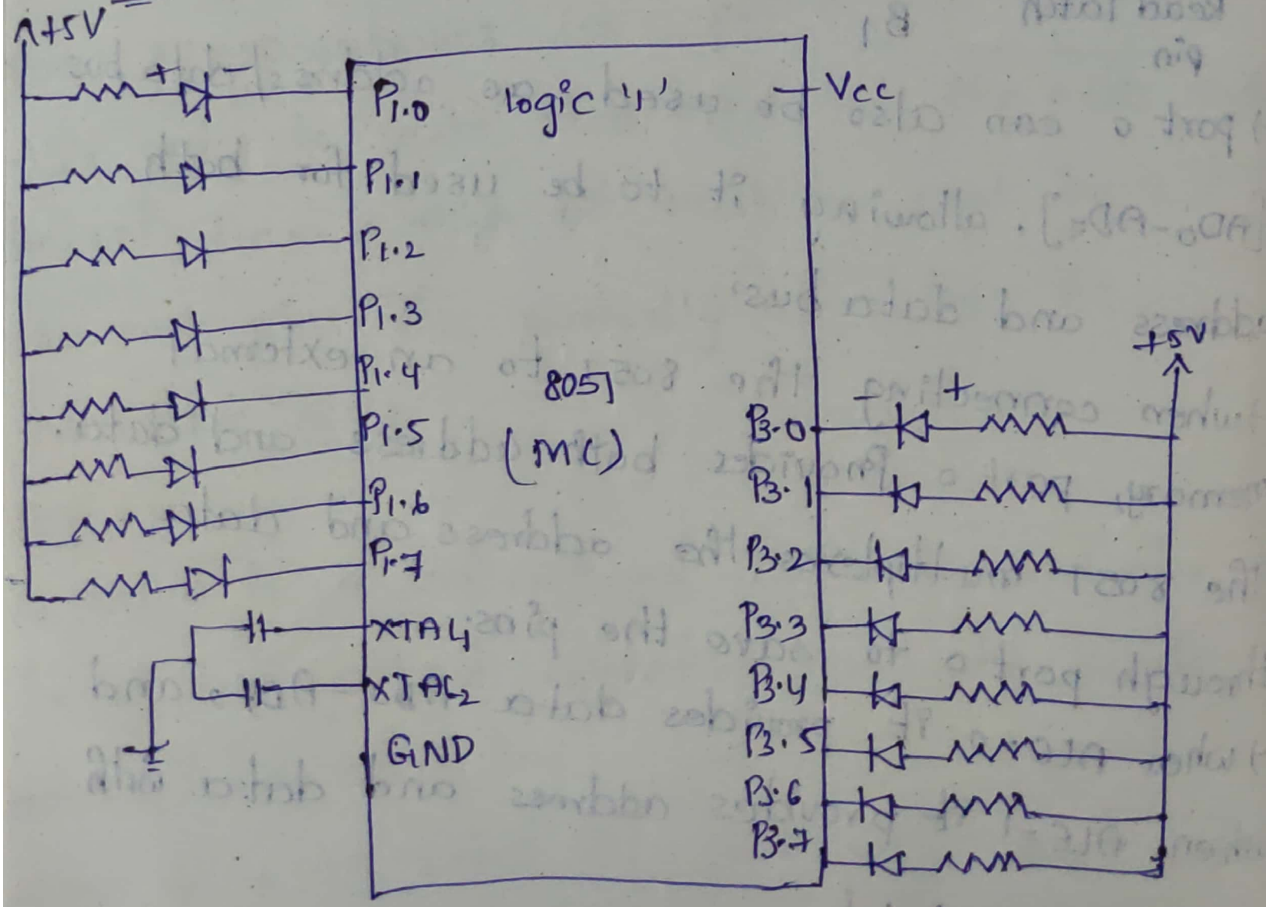
single bit will t/s'rd at a time

Parallel communication of 8051

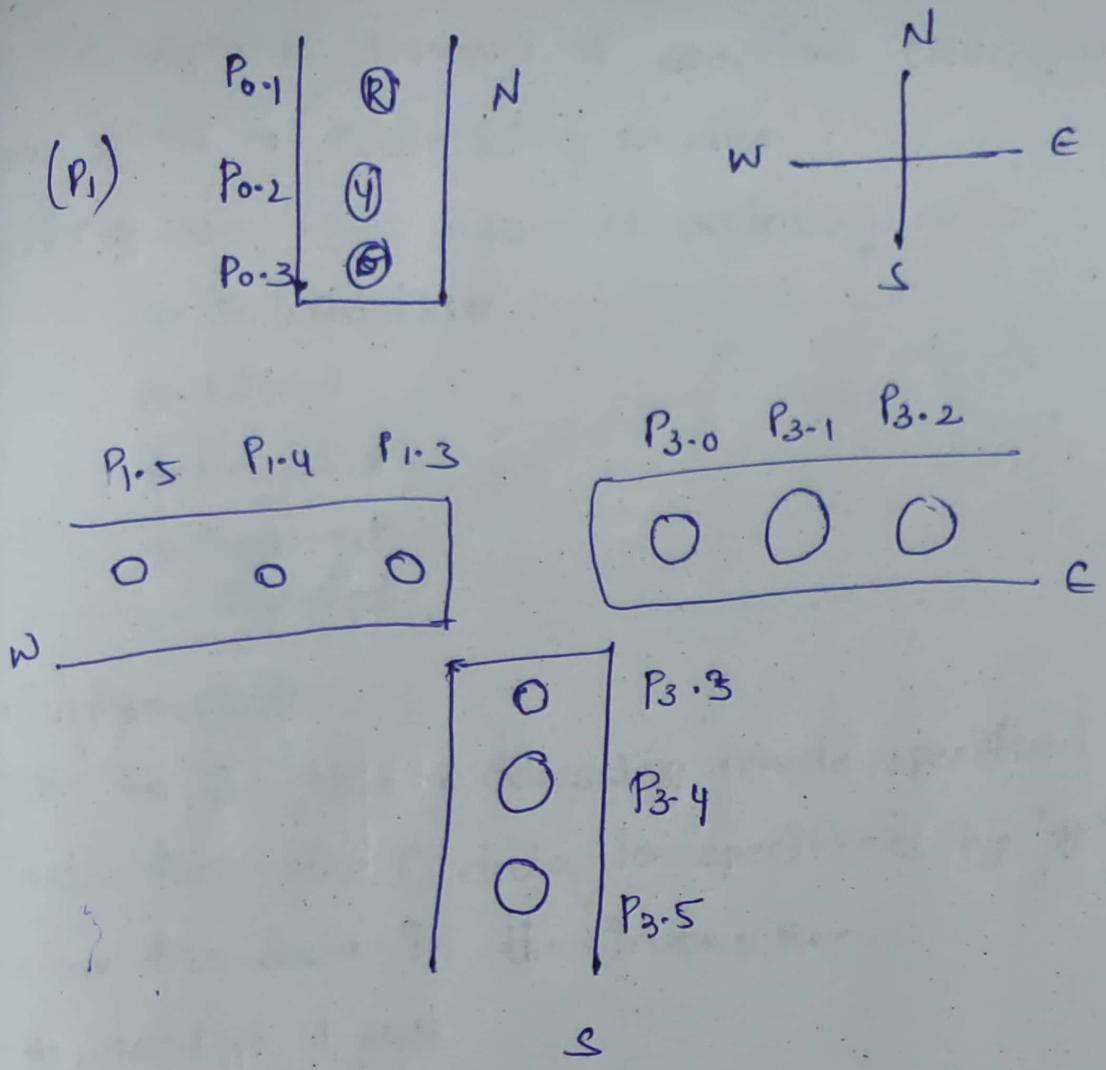


All the bits are ~~tfred~~ transfered at a time

Interfacing of key board with 8051:-



Traffic light



Addressing modes of 8051 :-

→ The various formates of specifing the operands are called a Addressing modes.

→ There are Five types of addressing modes

1. Intermediate
2. Direct
3. Register
4. Indirect
5. Indexed

1. Intermediate :-

In this addressing mode specified in instruction itself. data is specified by '#' symbol before the data in the instruction.

Ex :- MOV A, #34H
mov DPTR, #1234H.

Direct :-

In this addressing of operand is specified instruction only internal RAM and SFR Address allowed

Ex :- MOV A, 35H (using internal RAM)
MOV A, 90H (using internal SFR)

UNIT - 5 ARM PROCESSOR

ARM PROCESSOR :

An Arm processor is one of a family of central processing units (CPUs) bas ed on the reduced instruction set computer (RISC) architecture for computer pr ocessors. Arm Limited, the company behind the Arm processor, designs the cor e CPU components and licenses the intellectual property to partner organizatio ns, which then build Arm-based chips according to their own requirements. Ar m Limited does not manufacture or sell any chips directly.

The ARM microcontroller stands for Advance RISC Machine; it is one of the extensive and most licensed processor cores in the world. The first ARM proce ssor was developed in the year 1978 by Cambridge University, and the first AR M RISC processor was produced by the Acorn Group of Computers in the year 1985. These processors are specifically used in portable devices like digital ca meras, mobile phones, home networking modules and wireless communicatio n technologies and other embedded systems due to the benefits, such as low power consumption, reasonable performance, etc. This article gives an overvie w of ARM architecture with each module's principle of working.

An Introduction to ARM Architecture w ith Each Module's Working Principle

ARM Architecture :

The ARM architecture processor is an advanced reduced instruction set co mputing [RISC] machine and it's a 32bit reduced instruction set computer (RIS C) microcontroller. It was introduced by the Acron computer organization in 19 87. This ARM is a family of microcontroller
developed by makers like ST Microelectronics,Motorola, and so on. The ARM a rchitecture comes with totally different versions like ARMv1, ARMv2, etc., and, each one has its own advantage and ARM Architecture comes with totally diff erent versions like ARMv1, ARMv2, etc., and, each one has its own advantage a nd disadvantages.

ARM ARCHITECTURE

The ARM cortex is a complicated microcontroller within the ARM family that h as ARMv7 design. There are 3 subfamilies within the ARM cortex family :

- ARM Cortex Ax-series
- ARM-Cortex Rx-series
- ARM-Cortex Mx-series

The ARM Architecture

- Arithmetic Logic Unit
- Booth multiplier
- Barrel shifter
- Control unit

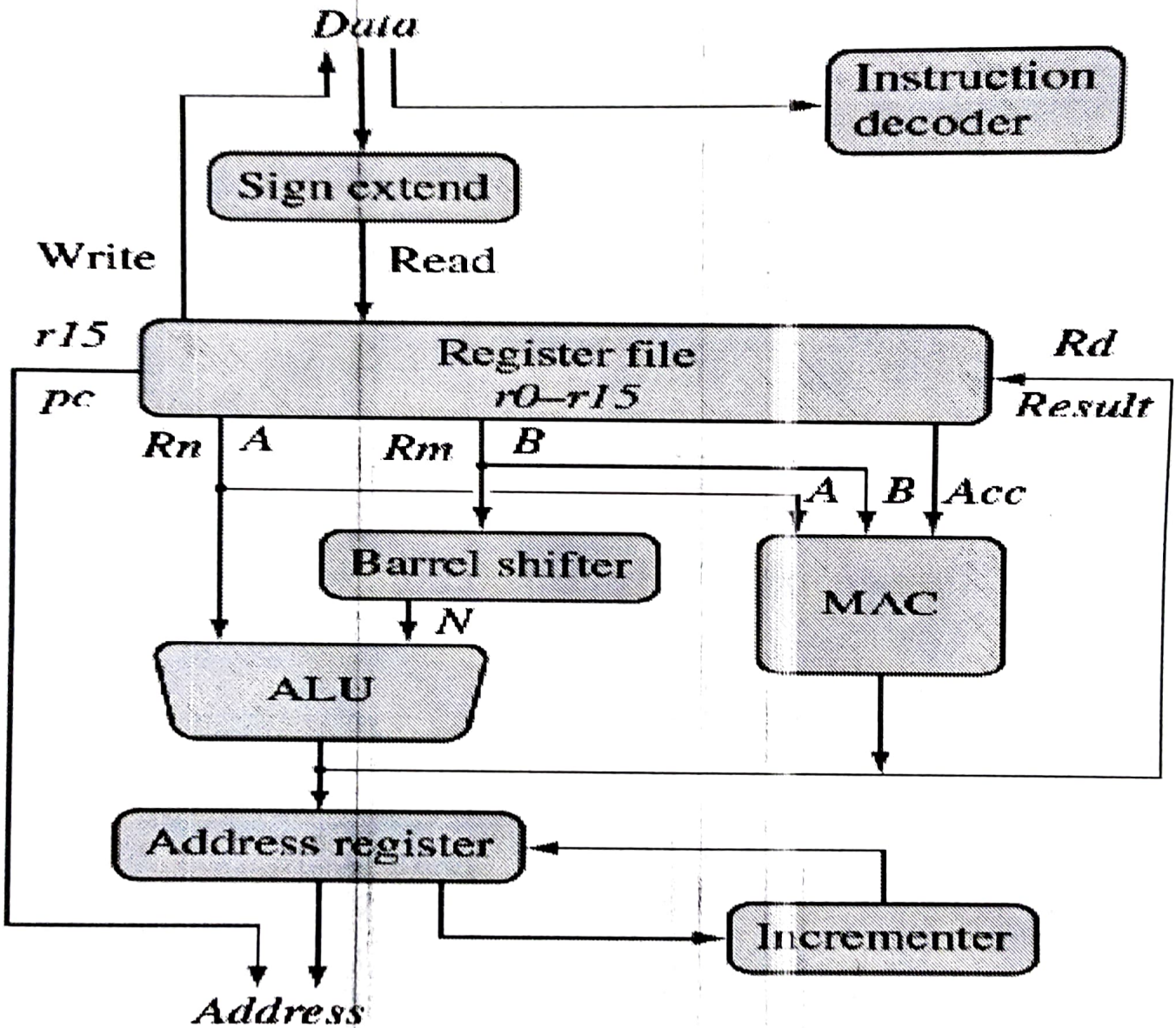


Fig:- ARM ARCHITECTURE

Arithmetic Logic Unit (ALU)

The ALU has two 32-bits inputs. The primary comes from the register file, wher eas the other comes from the shifter. Status registers flags modified by the AL U outputs. The V-bit output goes to the V flag as well as the Count goes to the C flag. Whereas the foremost significant bit really represents the S flag, the ALU output operation is done by NORed to get the Z flag. The ALU has a 4-bit function bus that permits up to 16 opcode to be implemented.

Booth Multiplier Factor

The multiplier factor has 3 32-bit inputs and the inputs return from the register file. The multiplier output is barely 32-Least Significant Bits of the merchandi se. The entity representation of the multiplier factor is shown in the above bloc k diagram. The multiplication starts whenever the beginning 04 input goes a ctive. Fin of the output goes high when finishing.

Booth Algorithm

Booth algorithm is a noteworthy multiplication algorithmic rule for 2's comple ment numbers. This treats positive and negative numbers uniformly. Moreover, the runs of 0's or 1's within the multiplier factor are skipped over without any a ddition or subtraction being performed, thereby creating possible quicker multi plication. The figure shows the simulation results for the multiplier test be nch. It's clear that the multiplication finishes only in16 clock cycle.

Barrel Shifter

The barrel shifter features a 32-bit input to be shifted. This input is coming ba ck from the register file or it might be immediate data. The shifter has differen t control inputs coming back from the instruction register. The Shift field within the instruction controls the operation of the barrel shifter. This field indicates the kind of shift to be performed (logical left or right, arithmetic right or ro tate right). The quantity by which the register ought to be shifted is contained i n an immediate field within the instruction or it might be the lower 6 bits of a r egister within the register file.

The shift_val input bus is 6-bits, permitting up to 32 bit shift. The shifttype indi cates the needed shift sort of 00, 01, 10, 11 are corresponding to shift left, shif t right, an arithmetic shift right and rotate right, respectively. The barrel shifter i s especially created with multiplexers.

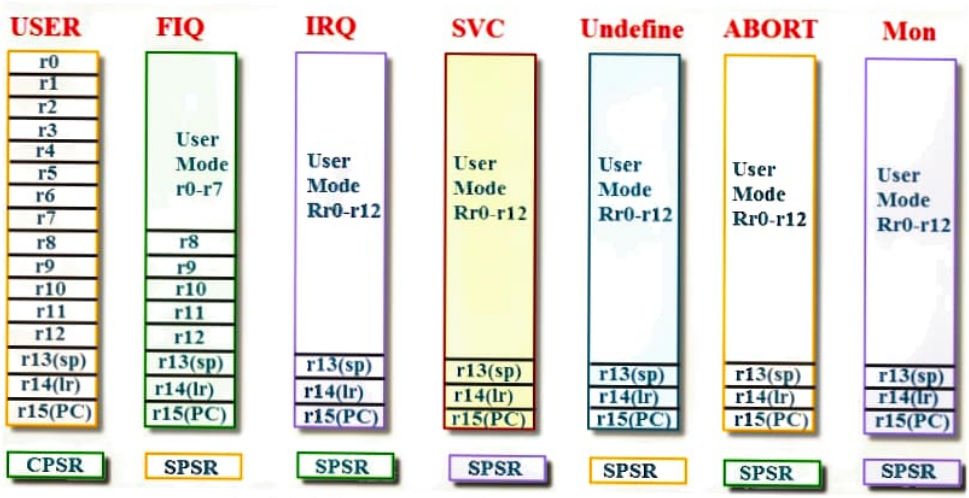
Control Unit

For any microprocessor, control unit is the heart of the whole process and it is r esponsible for the system operation,so the control unit design is the most imp ortant part within the whole design. The control unit is sometimes a pure comb inational circuit design. Here, the control unit is implemented by easy state ma chine. The processor timing is additionally included within the control unit. Sig nals from the control unit are connected to each component within the process or to supervise its operation.

ARM Microcontroller Register Modes

An ARM micrcontroller is a load store reducing instruction set computer archite cture means the core cannot directly operate with the memory. The data opera ions must be done by the registers and the information is stored in the memory by an address. The ARM cortex-M3 consists of 37 register sets wherein 31 are general purpose registers and 6 are status registers. The ARM uses seven proc essing modes to run the user task.

- USER Mode
- FIQ Mode
- IRQ Mode
- SVC Mode
- UNDEFINED Mode
- ABORT Mode
- Monitor Mode



ARM Microcontroller Register Modes

USER Mode: The user mode is a normal mode, which has the least number of registers. It doesn't have SPSR and has limited access to the CPSR.

FIQ and IRQ: The FIQ and IRQ are the two interrupt caused modes of the CPU. The FIQ is processing interrupt and IRQ is standard interrupt. The FIQ mode ha s additional five banked registers to provide more flexibility and high performa nce when critical interrupts are handled.

SVC Mode: The Supervisor mode is the software interrupt mode of the process

or to start up or reset.

Undefined Mode: The Undefined mode traps when illegal instructions are exe
uted. The ARM core consists of 32-bit data bus and faster data flow.

THUMB Mode: In THUMB mode 32-bit data is divided into 16-bits and increas
s the processing speed.

THUMB-2 Mode: In THUMB-2 mode the instructions can be either 16-bit or 32
bit and it increases the performance of the ARM cortex –M3 microcontroller. T
he ARM cortex-m3 microcontroller uses only THUMB-2 instructions.

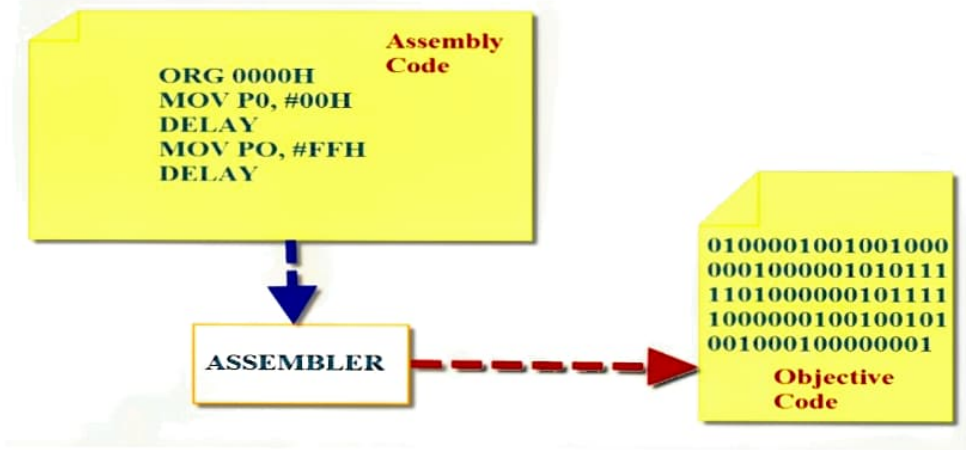
Some of the registers are reserved in each mode for the specific use of the co
e. The reserved registers are

- Stack Pointer (SP).
- Link Register (LR).
- Program Counter (PC).
- Current Program Status Register (CPSR).
- Saved Program Status Register (SPSR).

The reserved registers are used for specific functions. The SPSR and CPSR co
ntain the status control bits which are used to store the temporary data. The S
PSR and CPSR register have some properties that are defined operating mode
s, Interrupt enable or disable flags and ALU status flag. The ARM core operates
in two states 32-bit state or THUMBS state.

ARM-Cortex Microcontroller Programming

In the present days, the microcontroller vendors are offering 32-bit microcontro
llers based on ARM cortex-m3 architecture. Many embedded system developer
s are starting to use these 32-bit microcontrollers for their projects. The ARM
microcontrollers supports for both low-level and high level programming langu
ages. Some of the traditional microcontroller architectures are made with man
y limitations therefore, difficult to use the high level programming language.



ARM-Cortex Microcontroller Programming

For example the memory size is limited and performance might not be suffici ent. The ARM microcontrollers runs at 100Mhz frequency and higher performa nce, therefore it supports the higher level languages. The ARM microcontroller is programmed with different IDES such as keiluvision3, keiluvision4, coocox a nd so on. A 8-bit microcontroller use 8-bit instructions and the ARM cortex-M u ses a 32-instructions.

Additional Uses of the Cortex Processor

It is a reduced instruction set computing Controller

- 32-bit high performance central processing unit
- 3-stage pipeline and compact one

It has THUMB-2 technology

- Merges optimally with 16/32 bit instructions
- High performance

It supports tools and RTOS and its core Sight debug and trace

- JTAG or 2-pin serial wire debugs connection
- Support for multiple processors

Low power Modes

- It supports sleep modes
- Control the software package

ARM Cortex M3- Processor Architecture:

The Cortex-M3 processor is specifically developed for high-performance, low-cost platforms.

The Cortex-M3 is 32-bit MP. It has 32-bit data path, 32-bit Register bank, and 32-bit memory interface.

NVIC (Nested Vectored Interrupt controller):

* The highly configurable NVIC is an integral part of the cortex M3 processor and provides the processor's outstanding Interrupt handling abilities.
* It supplies a Non-Maskable Interrupt (NMI) and 32 general purpose physical Interrupts
* The cortex M3- processor uses a Re-locatable vector table that contains the address of the functions to be executed.
* It supports nesting (stacking) of Interrupts.

Memory Protection unit (MPU):

* It can improve the reliability of an embedded system by protecting the critical data used by operating system from i) user application.
  ii)
* Seperating processing task by disallowing access ~~data~~ to each other's data
* Disabling access to memory regions.
* Allowing memory regions to be defined an ROM
* Detecting unexpected memory accesses.

* The MPU seperates the memory into distinct Regions and implements protection by preventing disallowed accesses. The MPU supports upto 8 regions each of which can be divided into 8 sub regions.

Debug and Trace:

The debug access into a cortex-M3 processor based system is through the Debug Access port (DAP).

(i) DAP can be implemented a serial wire Debug port (SW-DP) for a two-pin (clock and data) interface

ii) Serial wire JTAG Debug port (SWJ-DP) that enables either JTAG or SW protocol to be used.

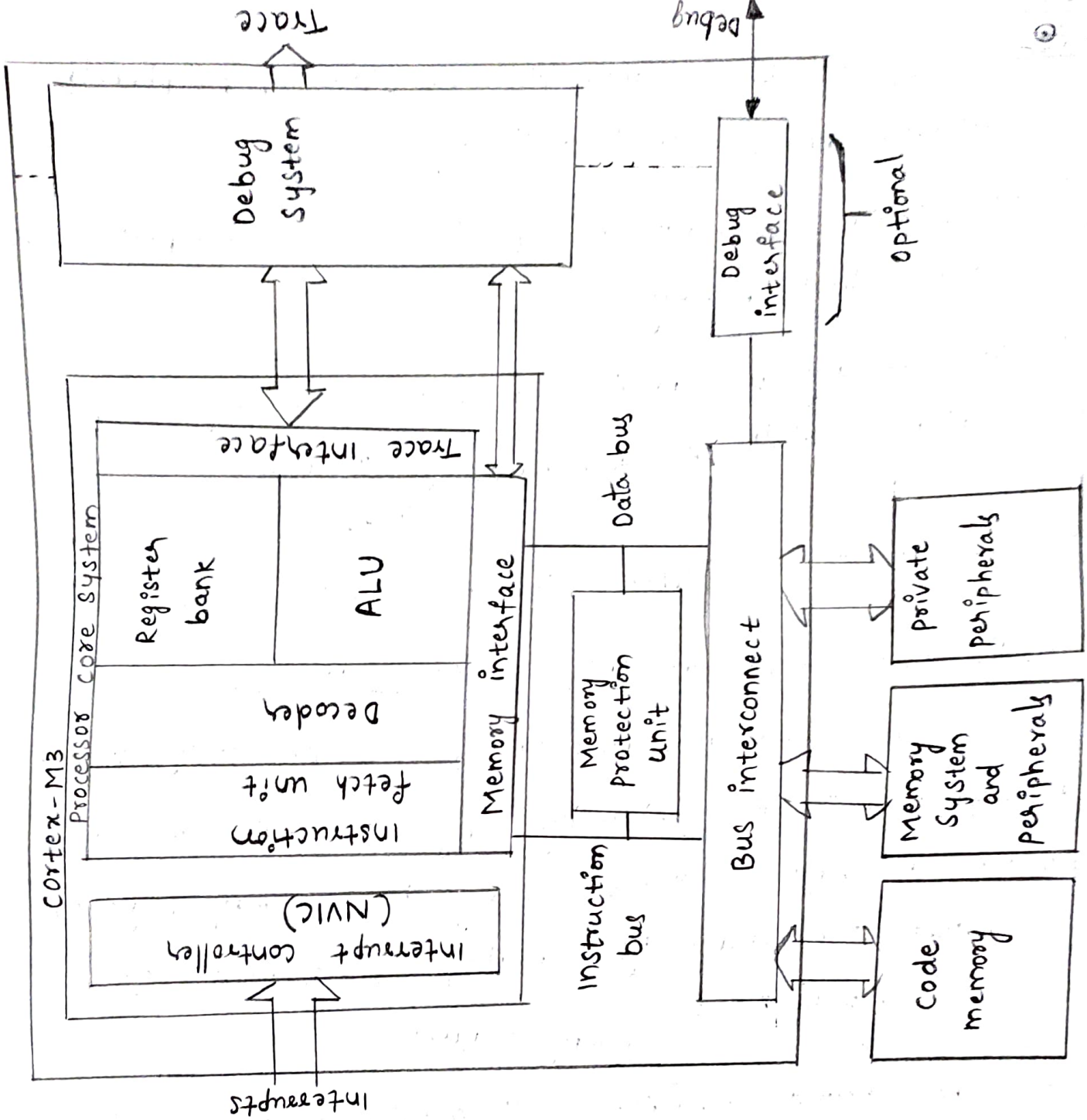
* When a debug event takes place, the cortex-M3 processor can either be halt mode or, the debug monitor mode.

Decoder, ALU, Register Bank:

* The cortex-M3 core contains a decoder for traditional thumb and new thumb-2 Instructions.

* An Advanced ALU with support H/W multiply and divide, control logic, and interfaces to the other components of the processor.

* The cortex M3- processor contains 13 general purpose Registers, two stack pointers,
A link Register,
A program counter,
A NO. of special Registers including Program status Registers.



BUS Interconnect:

* The Bus Interconnect connect the processor and debug interface to the external buses.
* This allows cortex-M3 to carry Instruction fetches
* The main bus Interfaces are as follows
  (i) system bus
  ii) code memory bus
  iii) private peripheral bus

* The system bus is used to access (SRAM), peripherals, external RAM, external devices.

* The private peripheral bus provides access to a part of the system-level memory.

* Code memory bus provides access to code memory, which consists of two buses < I-code / D-code

* The cortex M3 supports thumb-2 Instruction set.

SOFTWARE DELAY IN ARM CORTEX-M3 PROCESSOR:

In cortex M3 processor, It requires 12 cycles of the Processor clock for executing single Instruction cycle.

⇒ For an ARM cortex-M3 clocked by a 12MHz crystal, the time taken by executing one instruction cycle

$$1 \text{ instruction} = \frac{12}{12\text{MHz}} = 1\mu\text{sec}.$$

⇒ The shortest Instruction will execute in 1μsec and other instructions will take 2 or more μseconds depending up on the size of the instruction.

* Thus a time delay of any magnitude can be generated by looping suitable instructions a, required no. of time.

* Software delay is not very accurate because we cannot exactly predict how much time it takes for executing single Instruction.

* It is better to use timer for generating delay in time critical applications.
* How ever software delay routines are very easy to develop and well enough for less critical and simple applications

generate a delay using looping:

```
Delay - one_ms:
MOV r0, #3
wait : sub R0, #1
nops   bne wait
       nop
       nop
       bx lr
```

⇒ You can use SYSTICK inside ARM processor for delay. Just program it to count each 1µs, or less if you have enough clock speed, and do-while loop till your delay value expires.

```
void wait us (int us)
{ unsigned int cnt;
  while (us----->0)
  {
    cnt = STK_VAL;        (systick counter)
    while (STK_VAL - cnt) < 2);   (Repeat till 2 ticks)
  }
}
```

SUBROUTINE:

A subroutine is a block of code that performs a task based on some arguments and optionally returns a result.

* Registers R_0 to R_3 are used to pass arguments to subroutines
* And R_0 is used to pass a result back to the callers.
* only one copy of the instructions that constitute the subroutine is placed in memory and can be accessed repeatedly.

Components of Subroutines:

An Assembly language routine has

1. An entry point
   The location of the first instruction in the routine.
2. Parameters
   The list of Registers or memory locations that contain the Parameters for the routine.
3. Return values
   The list of Registers or, memory locations to save the results.
4. Working Storage
   The Reg or, mem locations required by the routine to Perform its task.

Calling subroutine:

1. You should be able to call the subroutine from anywhere
2. once the subroutine is complete, it should return back

* In ARM, the Branch and link instruction (BL) is used to Branch to subroutine.

```
BL  my_subroutine ;  it points 1st line to subroutine
```

* This instruction saves the current address of program counter (PC) in the link register (LR) before placing the starting address of the subroutine in (PC).
* When subroutine has completed the task, the processor must be able to branch back to the instruction immediately
* To return from subroutine should use the following.

Ex:

```
BX  LR ;        Return back
MOV PC, LR ;
```

Program: This subroutine takes a number as input accumulates the sum and stores the result in memory.

⇒

```
PRESERVES thumb
AREA   myData, DATA, READWRITE
SUMP   DCD 0      ; intialized to zero
AREA   CODE  READONLY  ALIGN=2
ENTRY
EXPORT__main
MAIN:  LDR R1, N
       MOV R0, #0
       BL  SUMP
       LDR R3, =sump
       STR R0, [R3]
       B STOP
```

```
SUMUP   PROC
        ADD R0, R0, R1
        SUBS R1, R1, #1
        BGT  SUMUP
        BX   LR
        ENDP
N       DCD 5
        ALIGN
STOP
END
```

PARAMETER PASSING:

When calling a subroutine, a calling program needs a mechanism to provide to the subroutine the i/p parameters, the operands that will be used in computation in the sub routines or their addresses.

The exchange of information between a calling program and a subroutine is called as parameter passing

* The parameter passing may be accomplished in three different ways.

1. Place the parameters in the registers.
2. Place the parameters in block of memory
3. Transfer the parameters and results on the H/W stack.

Passing Parameters in Registers:

This is the simplest method to pass parameters via the registers. Memory address, counters and other data can be passed to the subroutines through registers.

Ex: A subroutine takes two strings of equal length as i/p parameters.

(A) The length of the string can be passed through the Registers R0 and starting address of both the strings passed through Registers R1 & R2

Pgm:
```
MOV   R0 , # str length
mov   R1 , = string1
mov   R2 , = string2
BL    my_subroutine
```

Types of Parameters:

1. Pass-by-value: You are making copy in memory of the actual parameter's value that is passed in.
   * Changing the value in subroutine will not change the actual value of the parameter
2. Pass-by-Reference: In this method address of Parameters placed in the parameter's list.
   * Changing the parameter in the subroutine will also change the actual value.
3. Pass-by-name: Instead of Passing either value or, the reference of the parameter a string containing the name of the parameter is passed.

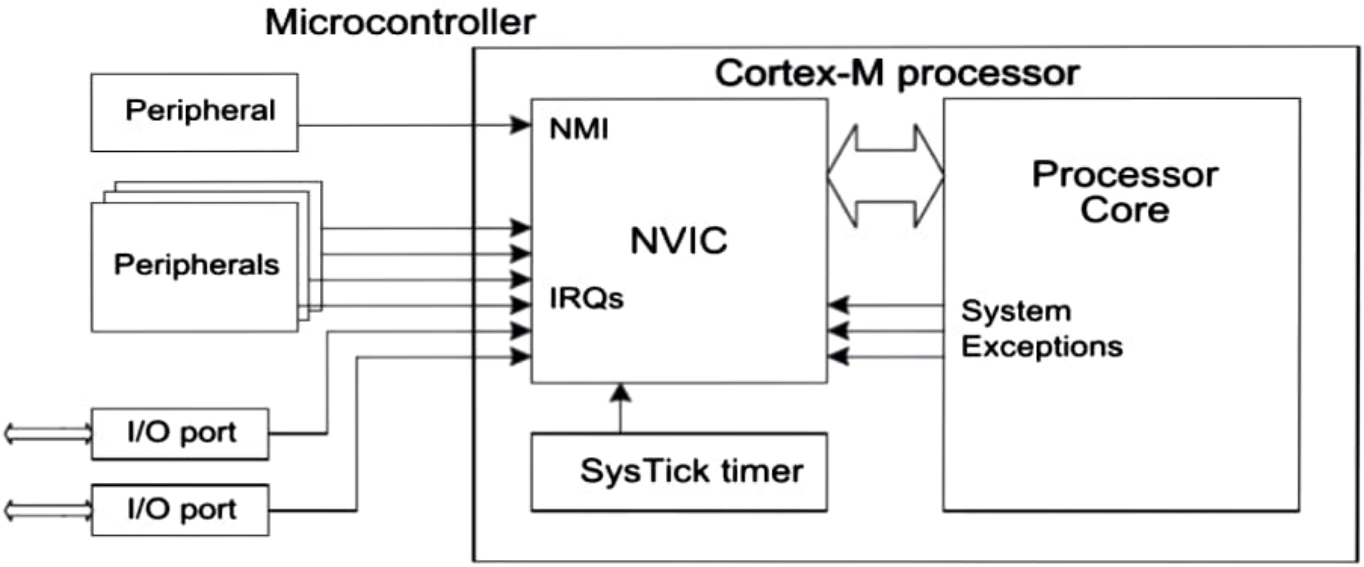
* This method is flexible, but time Consuming to verify lookup table all the time.

⇒ copy the subroutine program below.

NESTED VECTORED INTERRUPT CONTROLLER(NVIC) :

NVIC is an on-chip controller that provides fast and low latency response to int errupt-driven events in ARM Cortex-M MCUs. In this tutorial, We will explain the role of the nested vectored interrupt controller (NVIC) in interrupt handling requ ests of ARM Cortex-M microcontrollers. At the start, we will explain the excepti on and interrupt concepts that are related to Cortex-M architecture. After that,

we will also see how interrupts are handled by the Nested Vectored Interrupt Co ntroller (NVIC) of ARM MCU. In the last section, we will discuss what is the nee d of prioritizing an interrupt or exception.
In Cortex-M microcontrollers, a nested vectored interrupt controller usually kno wn as NVIC is used to handle all the interrupts and exceptions that Cortex-M s upports



The nested vectored interrupt controller is basically an integrated part of Corte x-M because of its tight integration with the cortex-M core. We can also config ure the interrupt controller according to our needs using specific registers. The mode of operation of most of the interrupt registers is privileged i.e. they can o nly be accessed in privileged mode, but if the interrupt is a software interrupt t han these registers can be accessed in user mode also. Followings are the mai n responsibilities of NVIC:

- Interrupts handling
- Programmable interrupt feature
- Interrup tail chaining
- Low interrupt latency management