UNIT-III

Microcontroller

Freescale's	6811
Intel's	8051
Zilog's	Z8
PIC	16x

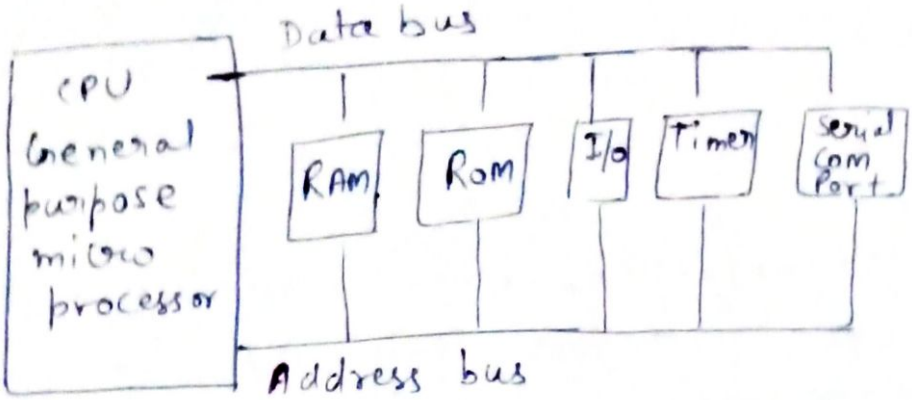
each of these microcontrollers has a unique instruction set and register set, therefore, they are not compatible with each other. Program written for one will not run on the others.

8051 microcontroller →

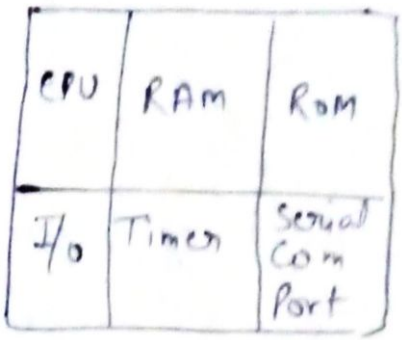
→ in 1981, intel corporation introduced an 8 bit microcontroller called the 8051. This microcontroller had 128 bytes of RAM, 4K bytes of on-chip Rom two timers, one serial port, and four ports (each 8-bits wide) all on a single chip. At the time it was also referred to as a "system on a chip" The 8051 is an 8-bit processor, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU.

The 8051 has a total of four I/O ports each 8 bit wide.

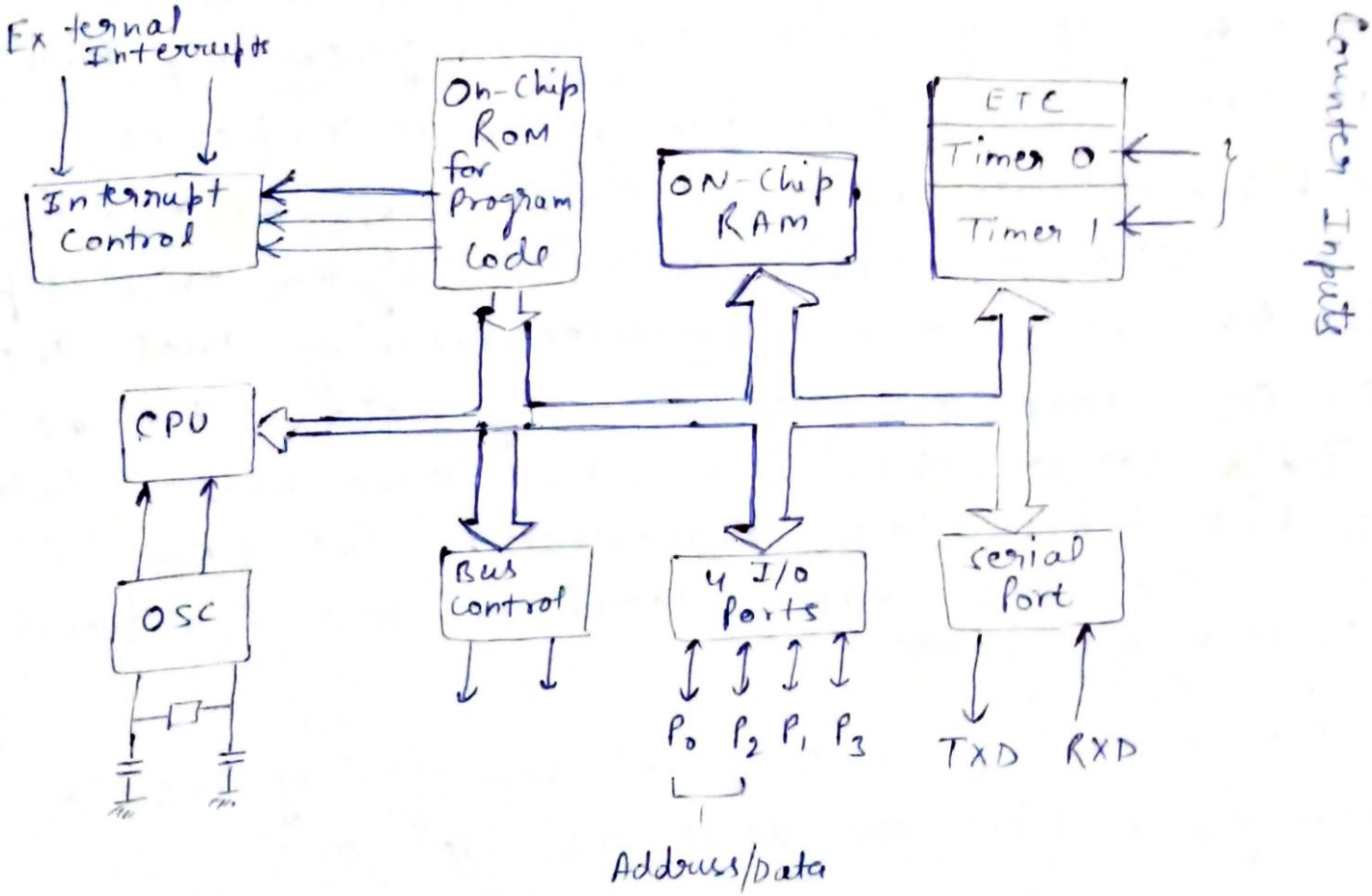
→ The 8051 is the original member of the 8051 family. intel refers to it as MCS-51



General-purpose microprocessor system



microcontroller



Inside the 8051 microcontroller
Block Diagram

- The 8052 is another member of the 8051 family. The 8052 has all the standard features of the 8051 as well as an extra 128 bytes of RAM and an extra timer.

In other words, the 8052 has 256 bytes of RAM and 3 timers. it also has 8K bytes of on-chip program ROM instead of 4K bytes.

feature	8051	8052	8031
ROM (bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timer	2	3	2
I/O Pins	32	32	32
Serial port	1	1	1
Interrupt Sources	6	8	6

Comparison of 8051 family members

As can be seen the 8051 is a subset of the 8052 therefore, all programs written for the 8051 will run on the 8052, but the reverse is Not true

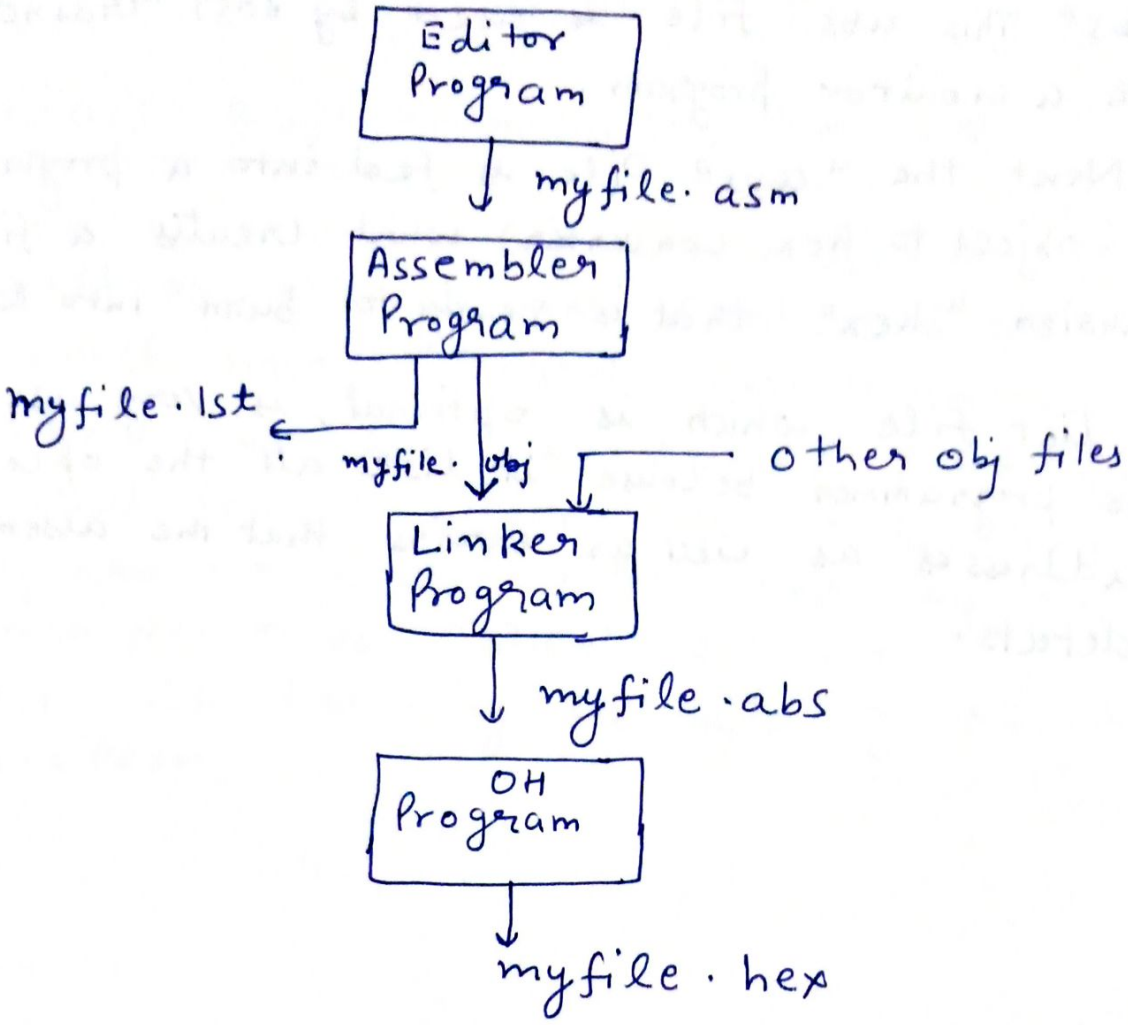
- Another member of the 8051 family is the 8031 chip. This chip is often referred to as a ROM-less 8051 since it has 0K bytes of on-chip ROM. To use this chip you must add external ROM to it. This External ROM must contain the program that the 8031 will fetch and execute. Contrast that to the 8051 in which the on-chip ROM

contains the Program to be fetched and Executed but is limited to only 4K bytes of code
- The ROM containing the Program attached to the 8031 can be as large as 64K bytes.

Various 8051 microcontrollers →

— 8051 is available in different memory types, such as UV-EPROM, flash, and NV-RAM all of which have different Part Number.
— ATMel flash 8051 is called AT89C51
— Dallas Semiconductor calls DS89C4X0
— NV-RAM version of the 8051 made by Dallas Semiconductor is called DS5000.
— 8751 chip has only 4K bytes of on-chip UV-EPROM

Assembling and running An 8051 Program →



→ first we use an editor to type in a program A widely used editor is the MS-DOS EDIT Program or Notepad in window, which comes with all microsoft OS. The source file has the extension "asm" or "src" depending on which assembler you are using

→ The "asm" Source file containing the program code created in step 1 is fed to an 8051 assembler. The assembler converts the instructions into machine code. The assembler will produce an object file and a list file. The extension for the object file is "obj" while the extension for the list file is "lst"

→ Assemblers require a third step called linking. The Link program takes one or more object files and produces an absolute object file with the extension "abs" This abs file is used by BOSI trainers that have a monitor program.

→ Next the "abs" file is fed into a program called OH (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM

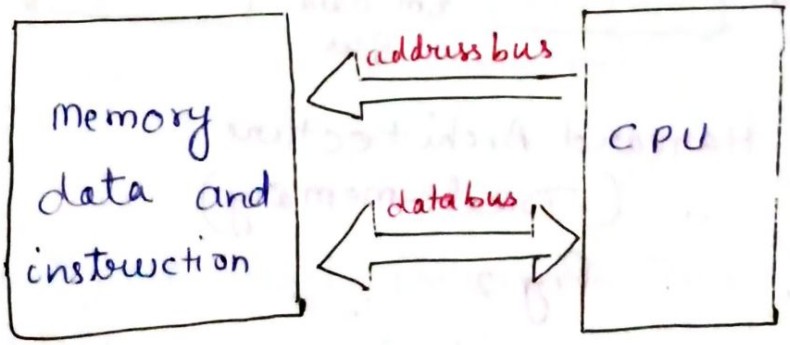
— List file which is optional, is very useful to the programmer because it lists all the opcodes and addresses as well as errors that the assembler detects.

UNIT-(I)III

Architecture of Digital Signal Processors :→

The biggest Advantages of DSP algorithms is transferring Information to and from Memory.
This Includes data, such as samples from the Input signal and the filter coefficients as well as program instructions.
The binary Codes that go into the program Sequencer.
for Example, suppose we need to multiply two Numbers that Reside Somewhere in Memory To do this, we must fetch three binary values from Memory, the No to be Multiplied, plus the Program instruction describing what to do

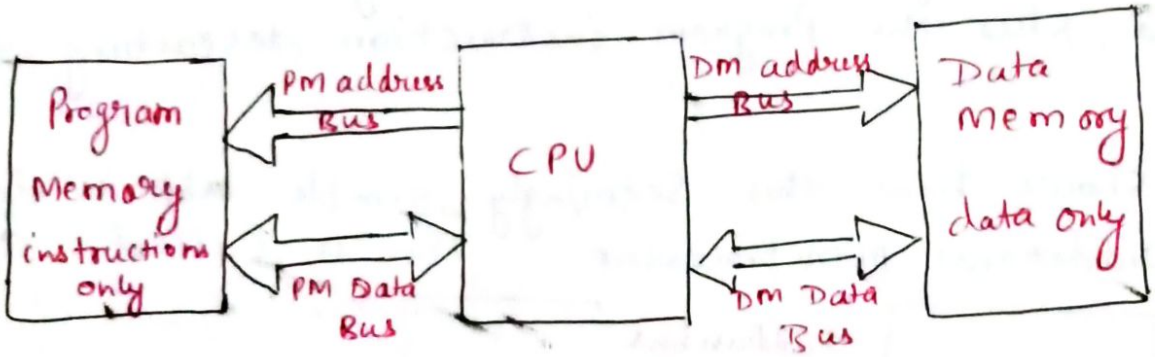
fig-1 shows how this Seemingly simple task is done in a traditional Microprocessor. This is called a



Von Neumann Architecture
(Single memory)
(fig-1)

Von Neumann architecture, after that brilliant American mathematician John Von Neumann
As shown in fig-1 a Von neumann Architecture Contains a Single memory and a single bus for transferring

data into and out of the Central processing Unit (CPU).
Multiplying two No. Requires at least three clockcycles one to transfer each of the three Numbers over the bus from the memory to the CPU. The Von-Neumann design is quite satisfactory, when you are content to execute all of the Required tasks in serial in fact most Computers today are of the Von-Neumann design.
We only Need other Architecture when very fast Processing is Required and we are willing to Pay the Price of Increased Complexity
fig 2 shows the Harvard Architecture. This is named for the work done at Harvard University in the 1940s

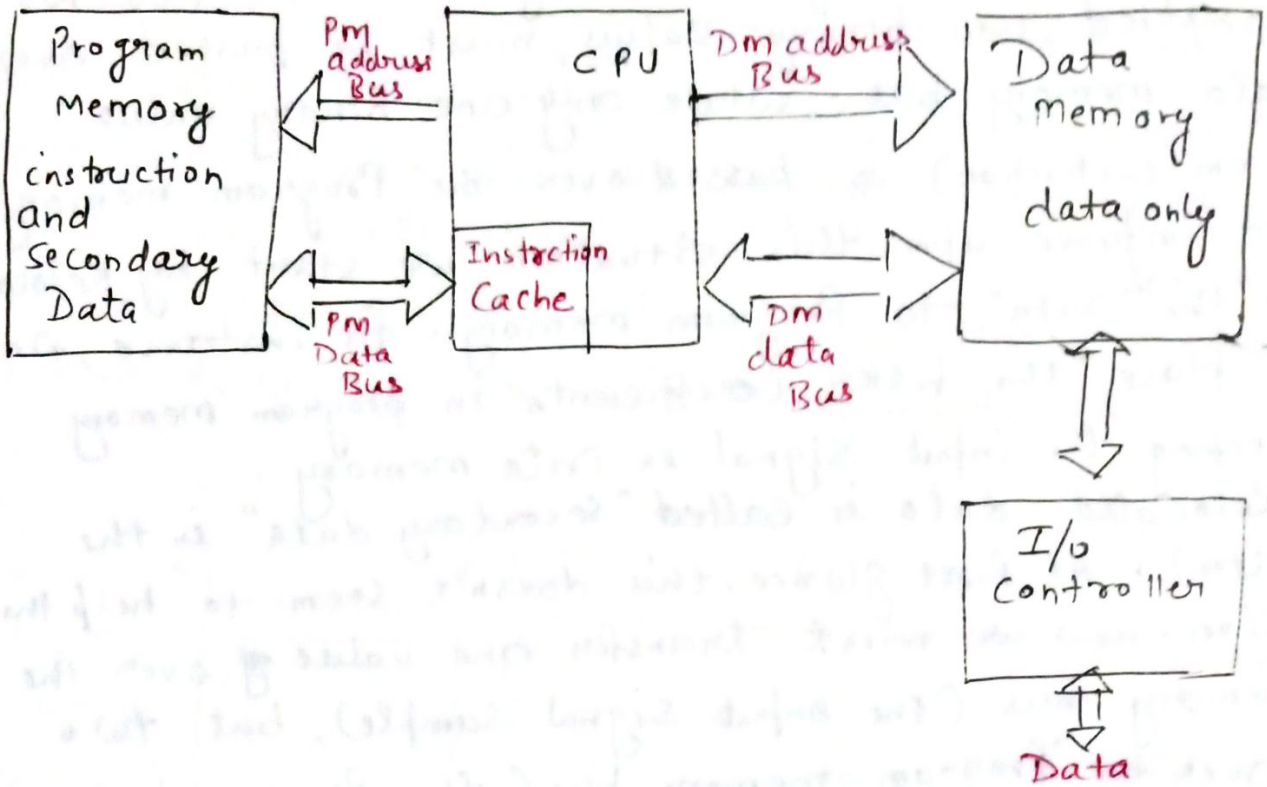


Harvard Architecture
(Dual memory)
fig 2

Under the leadership of Howard Aiken (1900-1973) Aiken insisted on separate memories for data and Program instructions with separates buses for each. Since the buses operate independently, Program instructions and data can be fetched at the same time, improving the speed over the single bus design. most Present day DSPs use the dual

bus architecture.

fig. 3 shows the Super harvard Architecture. This term was coined by Analog Devices to describe the internal Operation of their ADSP-2106X and New ADSP-21xx



Super Harvard Architecture
(Dual memory, instruction Cache
I/o Controller)

fig-3

families of Digital Signal Processors. These are called SHARC @ DSP's a Contraction of the longer term Super Harvard ARChitecture. The idea is to build upon the Harvard architecture by adding features to improve the throughput, while the SHARC DSP,s are optimized in Dozens of ways, two areas are important enough to be included in fig 3, an instruction cache and an I/o Controller.

first let's look at How the instruction Cache improves the performance of the Harvard architecture. A handicap of the basic Harvard design is that the data memory bus is busier than the Program memory bus. When two No. are multiplied, two binary Values, must be passed over the data memory bus, while only one binary value (program instruction) is passed over the Program memory bus. To improve upon this situation, we start by Relocating part of the "Data" to Program memory. for instance, we might place the filter coefficients in program memory while keeping the input signal in Data memory. (This Relocated data is called "secondary data" in the illustration). At first glance, this doesn't seem to help the situation. now we must transfer one value over the data memory bus (the input signal sample), but two values over the Program memory bus (the Program instruction and the coefficient). in fact, if we were executing random instruction, this situation would be no better at all.

However, DSP algorithms generally spend most of their execution time in loops, such as instructions, this means that the same set of program instructions will continually pass from program memory to the CPU. The super Harvard Architecture takes advantage of this situation by including an instruction Cache in the CPU. This is a small memory that contains about 32 of the most Recent programa instructions The first time through a loop the program instruction must be passed over the programm Memory bus. This Results in Slower operation because of the conflicts with the coefficients that must

must also be fetched along this path. However on additional execution of the loop, the Program instruction can be pulled from the instruction Cache. This means that all of the Memory to CPU information transfers can be accomplished in a single cycle. the Sample from the input signal comes over the data memory bus. the Coefficients comes over the Program memory bus, and the Program instruction comes from the instruction Cache. In the jargon of the field, this efficient transfer of data is called a high memory-access bandwidth.

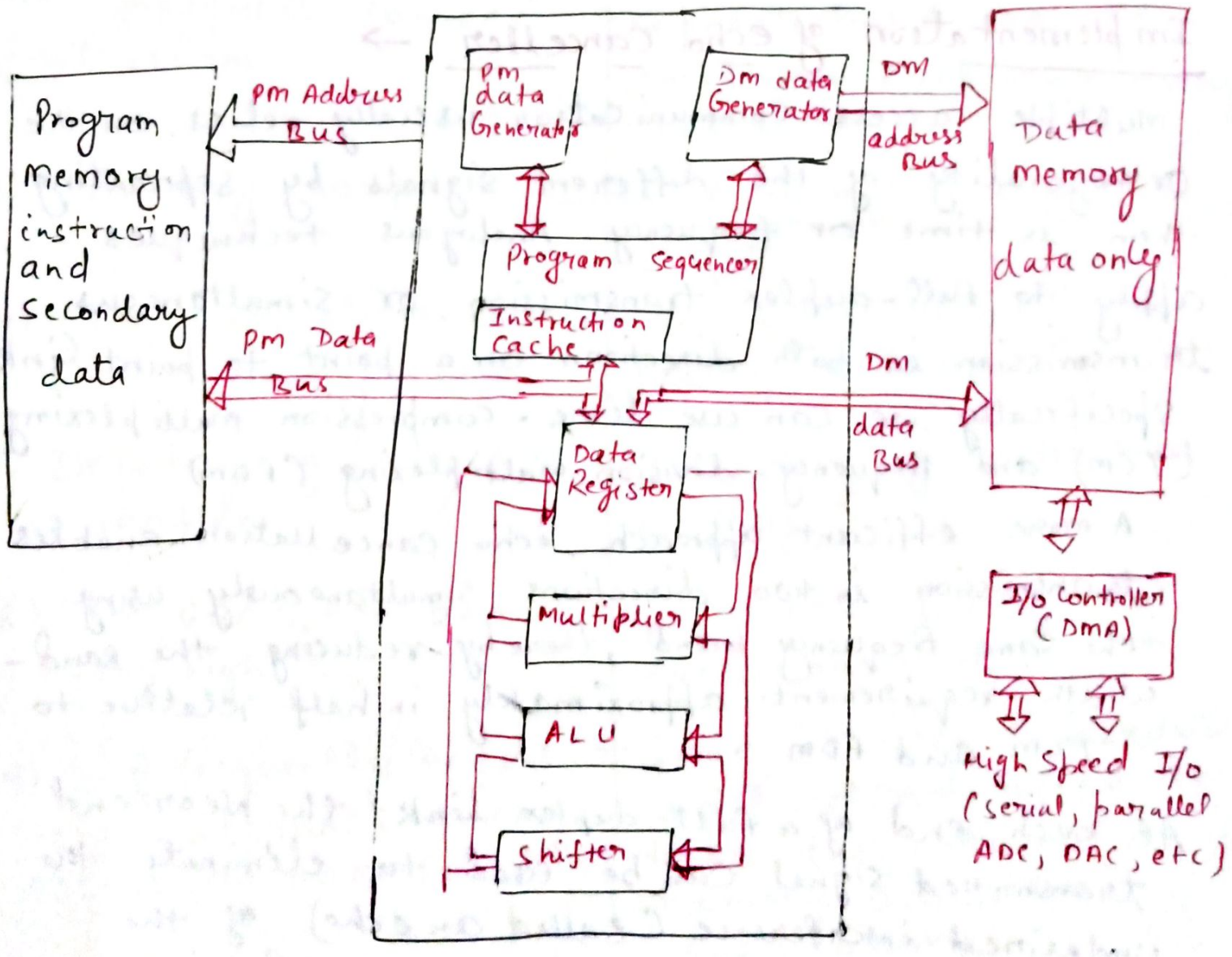


fig 3

fig 3 Presents a more detailed view of the SHARC architecture, showing the I/O Controller Connected to data memory, this is how the signals enter and exit the System. for Instance, the SHARC DSPs Provides both serial and parallel Communication Ports. Thuse are extremely high speed connections, for example, at a 40 MHz clock speed, there are two serial ports that operate at 40 Mbits/second each, while six parallel ports each provides a 40 Mbytes/second data transfer. when all six parallel ports are used together, the data transfer rate is an incredible 240 Mbytes/second.

CISC	RISC
(1) complex instruction Set computer (CISC) Processors has a bigger instruction set with many Addressing Modes	Reduced instruction set computer (RISC) Processors has a smaller instruction set with few addressing Modes
(2) It has to use a seperate micro-programming unit with a control memory to implement complex instructions	it has a hard-wired Programmed unit without a control memory and seperate hardware to implement each and every instruction
(3) An easy compiler design	(3) A complex compiler design
(4) The calculations are slower and precise	The calculations are faster and precise
(5) Decoding of instruction is complex	Decoding of instruction is easy
(6) The execution time is very high	it takes a very less execution time
(7) it frequently needs the external memory access to make Calculations	Since it uses a hardwired model, it not often to take the external memory access for Calculations

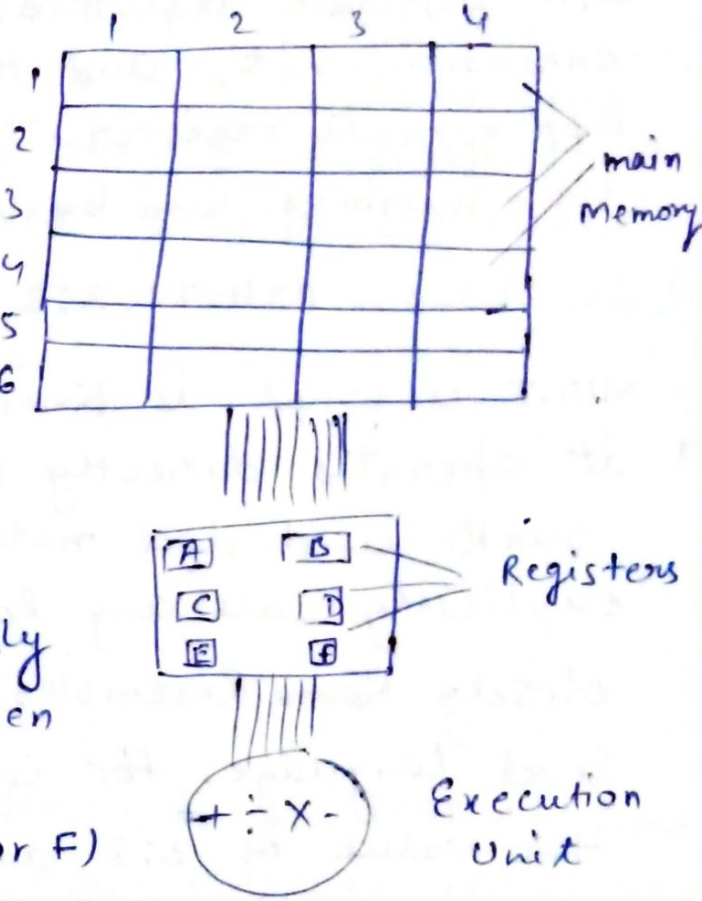
CISC	RISC
(8) Pipelining does not function Correctly here because of complexity in instructions	piplining is Not a major problem and this option speed up the processors
(9) These processors often Stall because of pipelining Problem	Since the instructions are not complex, stalling is Mostly reduced.
(10) Code Expansion is Not a Problem in CISC processors	Code expansion Can be a problem in some Cases in RISC Processors
(11) Disc Space is wasted	Disc space is saved
(12) used in low end applications such as Security Systems, Home Automation	

CISC and RISC approach →

The simplest way to examine the advantages and disadvantages of RISC architecture is by contrasting it with it's predecessor: CISC (complex Instruction Set computers)

multiplying two No in Memory →

On the Right is a diagram representing the storage scheme for a generic computer. The main memory is divided into Locations numbered from (row) 1: (Column) 1 to (row) 6: (column) 4.



The execution unit is responsible for carrying out all computations. However the execution unit can only operate on data that has been loaded into one of the six Registers (A, B, C, D, E or F)

Let's say we want to find the product of two No. — one stored in location 2:3 and another stored in location 5:2 and then store the product back in the location 2:3

The CISC approach →

The primary goal of CISC architecture is to complete a task in as few lines of assembly

as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. for this particular task, a CISC Processor would come prepared with a specific instruction (we'll call it "MULT")

when executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the Product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction

MULT 2:3, 5:2

MULT is what is Know as a complex instruction it operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. it closely Resembles a command in a higher level language. for instance, if we let 'a' represent the value of 2:3 and 'b' represent the value of 5:2 then this command is identical to the c statement 'a = a * b'

One of the primary advantage of this System is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively Short; very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC approach →

RISC Processors only used simple instructions that can be executed within one clock cycle. Thus the "MULT" command described above could be divided into three separate commands

"Load" which moves data from the memory bank to a Register "PROD" which finds the Product of two operands located within the Registers and "STORE" which moves data from a Register to the memory bank. In ~~other~~ order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly.

```
Load A, 2:3
Load B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. the compiler must also perform more work to convert a high-level language Statement into code of this form.

However, the RISC strategy also brings some very important advantages because each instruction requires only one clock cycle to execute

the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instruction" Require less transistors of Hardware space than the complex instructions, leaving more Room for General purpose Registers. Because all of the instruction execute in a uniform amount of time (i.e one clock), pipelining is possible.

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the compiler must perform. After a CISC-style "MULT" command is executed the Processor automatically erases the Registers if one of the operands needs ~~be~~ to be used for another computation, the Processor must re-load the data from the memory bank in to a Register. In RISC, the operand will remain in the Register until another value is loaded in its place

The performance equation →

The following equation is commonly used for expressing a computer's performance ability

$$\frac{\text{time}}{\text{Program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{Cycles}}{\text{instruction}} \times \frac{\text{Instruction}}{\text{Program}}$$

The CISC approach attempts to minimize the number of instructions per program, ~~Sacrificing~~ Sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the Number of instruction per program.

RISC Advantage →

Today, the intel x86 is arguable the only chip which retains CISC Architecture. this is primarily due to advancement in other areas of computer technology. The price of RAM has decreased dramatically. in 1977, 1MB of DRAM cost about $5,000 By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated So that the RISC use of RAM and emphasis on Software - has become ideal.

Arithmetic, Logic instructions →

Unsigned Numbers are defined as data in which all the bits are used to represent data, and no bits are set aside for the Positive or negative sign this means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data

Addition of Unsigned Numbers →

In the 8051, in order to add Numbers together, the accumulator register (A) must be involved. The form of the ADD instruction is

ADD A, source ; A = A + source

Que show how the flag register is affected by the following instruction

```
MOV A, #0F5H      ; A = F5 hex
ADD A, #0BH       ; A = F5 + 0B = 00
```

Sol -

```
  F5H
  0BH
------
 100H
```

```
 1111   0101
 0000   1011
-----------
10000   0000
```

After the addition, register A (destination) Contains 00 and the flags are as follows

CY=1 since there is a carry out from D_7
P=0 because the No of 1s is zero (an even no)
AC=1 since there is a carry from D_3 to D_4

ADDC and addition of 16-bit numbers →

ADDC → add with Carry

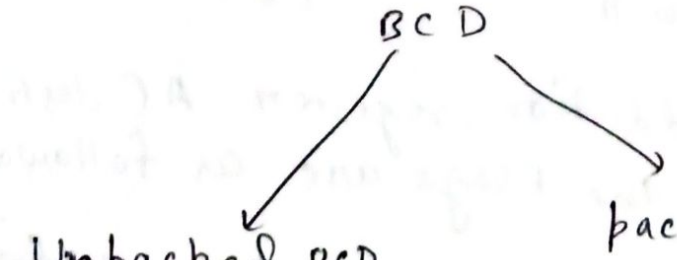
Eg write a program to add two 16-bit numbers. The numbers are 3CE7H and 3B8DH place the sum in R_7 and R_6 ; R_6 should have the lower byte

Sol

```
CLR C          ; make CY=0
MOV A, #0E7H   ; load the low byte Now A= E7H
ADD A, #8DH    ; add the low byte now
                 A= 74H and CY=1
MOV R6, A      ; Save the low byte of the Sum
                 in R6
MOV A, #3CH    ; load the high byte
ADDCA, #3BH    ; add with the carry
                 3B+3C+1= 78 (All in Hex)
MOV R7, A      ; Save the high byte of the Sum
```

BCD (binary coded decimal)

binary representation of 0 to 9 is called BCD



In Unpacked BCD, the lower 4 bits of the number represent the BCD Number, and the rest of the bits are 0. for example

0000 1001 and 0000 0101 are Unpacked BCD, for 9 and 5

Unpacked BCD requires 1 byte of memory or an 8-bit register to contain it

In Packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits, and one in the upper 4 bits. for example "0101 1001 is packed BCD for 59H

Packed BCD is twice as efficient in storing data.

```
MOV A, #17H
ADD A, #28H
```

Adding these two numbers gives 0011 1111B (3FH) which is not BCD. A BCD number can only have digits from 0000 to 1001 (or 0 to 9). In other words adding two BCD Number must give a BCD result the Result above should have been 17+28 = 45 (0100 0101) To correct this Problem, the Programmer must add 6 (0110) to the low digit 3F + 06 = 45H. The same problem could have happened in the upper digit (for example, in 52H+67H = D9H)

DA instruction →

The DA (decimal adjust for addition)

```
MOV  A, #47H   ; A = 47H first BCD operand
MOV  B, #25H   ; B = 25  second BCD  "
ADD  A, B      ; hex (binary) addition
                 A = 6CH
```

```
DA   A        ; adjust for BCD addition
                (A = 72 H)
```

After the Program is executed, register A will contain 72H (47 + 25 = 72)

The DA instruction works only on A. In other words, while the source can be an operand of any addressing mode. the destination must be in register A in order for DA to work

Qu Assume that 5 BCD data items are stored in RAM locations starting at 40H as shown below. write a program to find the sum of all the numbers. The result must be in BCD

40 = (71)
41 = (11)
42 = (65)
43 = (59)
44 = (37)

Sol

```
        MOV   R0, #40H    ; load pointer
        MOV   R2, #5      ; load counter
        CLR   A           ; A=0
        MOV   R7, A       ; clear R7
AGAIN:  ADD   A, @R0      ; add the byte pointer to A by R0
        DA    A           ; adjust for BCD
        JNC   NEXT        ; if CY=0 don't accumulate Carry
        INC   R7          ; Keep track of Carriers
```

NEXT : INC R_0 ; increment pointer
DJNZ R_2, AGAIN ; repeat until R_2 is zero.

Subtraction of Unsigned Number →

SUBB A, Source ; A = A - Source - Cy

Eg show the steps involved in the following

```
CLR C            ; make cy=0
MOV A, #3FH      ; load 3FH into A (A = 3FH)
MOV R3, #23H     ; load 23H into R3 (R3 = 23H)
SUBB A, R3       ; subtract A - R3 ; Place Result in A
```

A = 3f 0011 1111
R3 = 23 0010 0011
1C

```
  0011 1111
+ 1101 1101
-------------
1 0001 1100
```

I

Invert the carry

2's comp

The flag would be set as follows

CY = 0, AC = 0. And the programmer must look at the carry flag to determine if the Result is positive or negative

Que Analyze the following program

```
        CLR C
        MOV A, #4CH       ; Load A with value
                          ; 4CH (A=4CH)
        SUBB A, #6EH      ; subtract 6E from A
        JNC NEXT          ; if CY=0 jump to
                          ; NEXT target
        CPL A             ; if CY=1 then take
                          ; 1's complement
        INC A             ; and increment to
                          ; get 2's complement
NEXT:   MOV R1, A         ; Save A in R1
```

4C	0100 1100		0100 1100
−6E	0100 1110	2's com	1001 0011
−22			0 1101 1110

CY=1, the result is −ve, in 2's complement

Que Analyze the following program

```
CLR C             ; CY=0
MOV A, #62H       ; A=62H
SUBB A, #96H      ; 62H-96H
                  ; =CCH with CY=1
MOV R7, A         ; Save the result
MOV A, #27H       ; A=27H
SUBB A, #12H      ; 27H-12H-1 = 14H
MOV R6, A
```

Sol:

After the SUBB, A = 62H − 96H = CCH and the Carry flag is set high indicating there is a borrow. Since CY=1, when SUBB is executed the second time A = 27H − 12H − 1 = 14H. Therefore, we have

2762H − 1296H = 14CCH

Unsigned multiplication and division →

```
MUL AB ;  A x B place 16 bit Result in B and A
```

Ex

```
mov A, #25H ; load 25H to reg. A
mov B, #65H ; load 65H in Reg B
MUL AB      ; 25H * 65H = E99 where
            ; B = 0EH and A = 99H
```

Division

```
DIV AB ; divide A by B
```

Ex

```
mov A, #95  ; load 95 into A
mov B, #10  ; load 10 into B
DIV AB      ; Now A = 09 (Quotient)
              and B = 05 (Remainder)
```

Que Write a prog

(a) to make P_1 an Input port

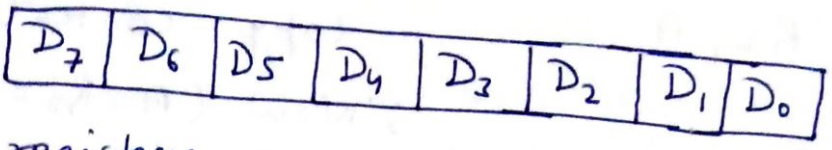
(b) to get a byte of hex data in the range of 00 - FFH from P_1 and convert it to decimal. Save the digits in R_7 R_6 and R_5, where the least significant digit is in R_7

Sol

```
mov   A, #0FFH
mov   P1, A      ; make P1 an input port
mov   A, P1      ; (read data from P1)
mov   B, #10     ; B = 0A hex (10 dec)
DIV   AB         ; divide by 10
mov   R7, B      ; Save lower digit
mov   B, #10     ;
DIV   AB         ; divide by 10 once more
mov   R6, B      ; Save the next digit
mov   R5, A      ; Save the last digit
```

Registers →

In the CPU registers are used to store informati temporarily.



8051 registers are 8-bit registers.

D7 — MSB
D0 — LSB

The most widely used registers of the 8051 are A (accumulator), B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR (data pointer), and PC (Program Counter). All of the above Registers are 8-bits except DPTR and the Program Counter

The Accumulator, register A is used for all arithmetic and logic instruction.

Mov instruction →

Mov instruction copies data from one location to another location

→ Mov destination, Source ;

Ex → Mov A, R0 it copies the contents of Register R0 to Register A

Ex →
```
Mov A, #55H ; Load value 55H into register A
Mov R0, A   ; Copy contents of A
```

into R_0
; Now ($A = R_0 = 55H$)

```
Mov   R1, A        ; copy contents of A into R1
                   ; Now (A = R0 = R1 = 55H)
mov   R2, A        ; copy contents of A into R2
                   ; Now (A = R0 = R1 = R2 = 55H)
Mov   R3, #95H     ; load value 95H into R3
                   ; Now (R3 = 95H)
mov   A, R3        ; copy contents of R3 into A
                   ; Now (A = R3 = 95H)
```

signifies that it is a value.

→ `Mov R5, #0F9H` that a 0 is used between the # and f to indicate that f is a hex Number and not a letter. In other words `Mov R5, #F9H` will cause an error.

→ `Mov A, #5` the result will be A = 05 that is A = 00000101 in binary.

→ A value to be loaded into a register must be proceeded with a pound sign (#) otherwise it means to load from a memory location

Ex- `Mov A, 17H` means to move into A the value held in memory location 17H which could have any value.

`Mov A, #17H` → Load the value 17H

into the Accumulator

ADD Instruction →

The ADD instruction has the following format

```
ADD A, Source      ; ADD the Source
                   ; operand to the accumul
                   ; ator
```

→ The ADD instruction tells the CPU to add the Source byte to register A and Put the result in Register A

→ To add two Numbers such as 25H and 34H, each can be moved to a Register and then added together

```
MOV   A, #25H      ; load 25H into A
MOV   R2, #34H     ; Load 34H into R2
ADD   A, R2        ; add R2 to
                   ; accumulator
                   ; (A = A + R2)
```

Executing the Program above results in A = 59H (25H + 34H) = 59H and R_2 = 34H

Notice that the content of R_2 does not change The program above can be written in many ways, depending on the Registers used Another way might be

```
MOV   R5, #25H     ; Load 25H into R5
MOV   R7, #34H     ; Load 34H into R7
```

```
MOV  A, #0      ; Load 0 into A.
                ; (A=0, clear A)
ADD  A, R5      ; add to A Content of
                ; R5 where A = A + R5
ADD  A, R7      ; add to A content of R7
                ; where A = A + R7
```

Another variation of Above program

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

Immediate operand.

→ ADD R2, #12H is Invalid Since Register A (accumulator) must be involved in any arithmetic operation

→ ADD R4, A is also invalid for the reason that A must be the destination of any arithmetic operation.

Question → write the instructions to move value 34H into Register A and value 3FH into Register B, then add them together.

Que → Write the instructions to add the values 16H and CDH. Place the Result in Register R2.

Addressing modes in 8051

→ The CPU can access data in various ways. The data could be in a register or in memory, or be provided as an immediate value. These various ways of accessing data are called addressing modes.

The 8051 provides a total of five distinct addressing modes. they are

1. immediate
2. register
3. direct
4. register indirect
5. indexed

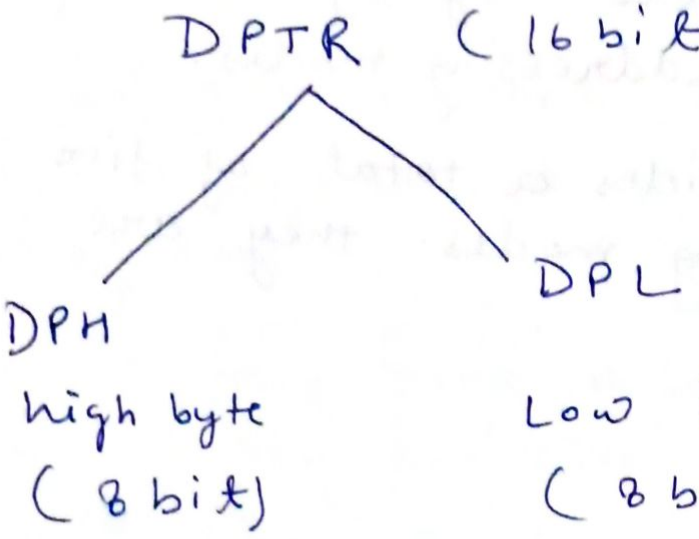
(1) Immediate and Register addressing modes →

In this addressing mode, the source operand is constant. In immediate addressing mode, as the name implies. when the instruction is assembled the operand comes immediately after the opcode.

Notice that the immediate data must be preceded by the Pound sign #

EX →

```
mov A, #25H        ; Load 25H into A
mov R4, #62        ; Load the decimal value
                     62 into R4
mov B, #40H        ; Load 40H into B
Move DPTR, #4521H  ; DPTR = 4512H
```




```
MOV  DPTR, #2550H
     is the same as
MOV  DPL, #50H
MOV  DPH, #25H
```

(2) Register addressing mode →

Register addressing mode involves the use of register to hold the data to be manipulated

Ex →

```
mov A, R0    ; copy the contents of
               R0 into A
mov R2, A    ; copy the contents of A
               into R2
```

ADD A, R_5 ; add the contents of R_5 to contents of A

→ It should be noted that the source and destination registers must match in size

In other words, coding "MOV DPTR, A" will give an error, since the source is an 8-bit register and the destination is a 16-bit Register.

```
MOV  DPTR, #25F5H
MOV  R7, DPL
MOV  R6, DPH
```

③ direct addressing mode → (~~Register~~ ~~Indirect Addressing~~)

128 bytes of RAM in the 8051 → RAM has been assigned addresses 00 to 7FH. The following is a summary of the allocation of these 128 bytes

(1) RAM locations 00 — 1FH are assigned to the Register banks and stack.

(2) RAM Locations 20 — 2FH are set a side as bit-addressable space to save single bit data

(3) RAM locations 30 - 7FH are available as a place to save byte-Size data

→ In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction.

Contrast this with immediate addressing mode, in which the operand itself is provided with the instruction. The # sign distinguishes between the two modes

Ex
```
MOV  R0, 40H   ; save content of
                 RAM location 40H in R0
MOV 56H, A     ; save content of A
                 in RAM location 56H
MOV R4, 7FH    ; move contents of
                 RAM location 7FH
                 to R4
```

RAM locations 0 to 7 are allocated to 0registers R_0-R_7. These Registers can be accessed in two ways

Ex
```
MOV A, 4   ; is same as
MOV A, R4  ; which means copy R4
             into A.
```

SFR Registers and their addresses →

→ R_0-R_7 are part of the 128 bytes of RAM memory

→ Register A, B, PSW and DPTR are part of the group of registers commonly referred to as SFR(Special function Registers)

→ The SFR can be accessed by their names or by their addresses.

Symbol	address
PSW	0D0H
SP	81H
Acc	0E0H
B	0F0H
DPTR	
↳ DPL	82H
DPH	83H

Ex mov 0E0H, #55H ; is the same as
mov A, #55H ; which means load 55H into A

(3) stack and direct addressing mode →

Another major use of direct addressing mode is the stack. In the 8051 family, only direct direct Addressing mode is Allowed for Pushing onto the stack. therefore an instruction such as "PUSH A" is invalid. Pushing the Accumulator onto the stack must be coded as "PUSH 0E0H" where 0E0H is the address of Register A.

similarly, Pushing R_3 of bank 0 is coded as "PUSH 03"

Direct addressing mode must be used for the POP instruction as well

Ex "POP 04" will pop the top of the stack into R_4 of bank 0

(4) Register indirect addressing mode →

Instruction Set

In the Register indirect addressing mode, a register is used as a pointer to the data. If the data is inside the CPU, only registers R_0 and R_1 are used for this purpose.

In other words, $R_2 - R_7$ cannot be used to hold the address of an operand located in RAM when using this addressing mode.

When R_0 and R_1 are used as Pointers that is when they hold the address of RAM locations, they must be preceded by the "@" sign

```
MOV A, @R0   ; move contents of RAM location
             ; whose address is held by
               R0 into A
```

Qu4 Write a Program to copy the value of 55H into RAM memory locations 40H to 45H using

(a) direct addressing mode
(b) register indirect addressing mode without a loop
(c) " " " " with a loop

Sol (a)

```
MOV A, #55H     ; load A with value 55H
MOV 40H, A      ; Copy A to RAM location 40H
MOV 41H, A      ;  "  "  "  "  "  "  41H
MOV 42H, A      ;  "  "  "  "  "  "  42H
MOV 43H, A      ;  "  "  "  "  "  "  43H
MOV 44H, A      ;  "  "  "  "  "  "  44H
```

Instruction set

(4)

(b)

```
MOV   A, #55H      ; load A with value 55H
MOV   R0, #40H     ; load the pointer. R0 = 40H
MOV   @R0, A       ; copy A to RAM location R0
                     Points to
INC   R0           ; increment pointer. Now R0 = 41H
MOV   @R0, A       ; copy A to RAM Location R0
                     Points to
INC   R0           ; increment pointer Now R0 = 42H
MOV   @R0, A
INC   R0
MOV   @R0, A
INC   R0           ; increment pointer. Now R0 = 44H
MOV   @R0, A
```

(c)

```
MOV   A, #55       ;  A = 55H
MOV   R0, #40H     ; load pointer. R0 = 40H, RAM address
MOV   R2, #05      ; Load counter, R2 = 5
AGAIN: MOV  @R0, A ; copy 55H to RAM location R0
                     points to
       INC  R0     ; increment R0 pointer
       DJNZ R2, AGAIN ; loop until counter = zero
```

Qu write a program to clear 16 RAM locations starting at RAM address 60H

Sol

```
CLR   A            ; A = 0
```

Instruction Set

```
        MOV     R1, #60H    ; load pointer . R1 = 60H
        MOV     R7, #16     ; load counter , R7 = 16
                              ( 10 in Hex)
AGAIN:  MOV     @R1, A      ; clear RAM location R1
                              Points to
        INC     R1          ; increment R1 Pointer
        DJNZ    R7, AGAIN   ; loop until Counter = zero
```

Ex: write a program to copy a block of 10 bytes of data from RAM locations starting at 35H to RAM locations starting at 60H

Sol

```
        MOV     R0, #35H    ; source pointer
        MOV     R1, #60H    ; destination pointer
        MOV     R3, #10     ; counter
BACK:   MOV     A, @R0      ; get a byte from source
        MOV     @R1, A      ; wfy it to destination
        INC     R0          ; increment source pointer
        INC     R1          ; increment destination
                              pointer
        DJNZ    R3, BACK    ; keep doing it for all ten
                              bytes
```

Limitation of Register indirect addressing Mode in the 8051 →

→ Looping is Not possible in direct addressing Mode but possible in Register indirect " "

→ R0, and R1 are the only Registers that can be used for pointers in Register indirect

Instruction Set

addressing mode.

Since R0 and R1 are 8 bits wide their use is limited to accessing any information in the internal RAM. However, there are times when we need to access data stored in external RAM or in the code space of on-chip Rom. whether accessing externally connected RAM or on-chip Rom, we need a 16-bit pointer. In such cases the DPTR register is used

Indirect addressing mode and ON-Chip Rom access →

Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the Program Rom space of the 8051

The instruction used for this purpose is "MOVC A, @A+DPTR". The 16-bit Register DPTR and register A are used to form the address of the data element stored in ON-chip Rom.

Qu4 write a Prg, assume that the word USA is burned into ROM locations starting at 200H, and that the Program is burned into Rom locations starting at 0. Analyze how the Program works and state where USA is stored after this Program is Run

Sol

```
        ORG   0000H          ; burn into Rom
                             starting at 0
        MOV DPTR, #200H      ; DPTR=200H look-up
                             table address
        CLR A                ; clear A(A=0)
        MOVC A, @A+DPTR      ; get the char
                             from code space
        MOV  R0, A           ; save it in R0
        INC  DPTR            ; DPTR=201 pointing
                               to next char
        CLR A                ; clear A(A=0)
        MOVC A, @A+DPTR      ; get the next char
        MOV  R1, A           ; save it in R1
        INC  DPTR            ; DPTR=202 pointing
                               to Next char
        CLR A                ; clear A(A=0)
        MOVC A, @A+DPTR      ; get the next char
        MOV  R2, A           ; save it in R2
HERE:   SJMP HERE            ; stay here.
;Data is burned into Code Space starting
 at 200H
        ORG  200H
MYDATA: DB "USA"
        END                  ; end of program
```

Qu4 Write a Program to

(a) clear ACC, then

(b) add 3 to the accumulator ten times

Sol ; This Program adds value 3 to the Acc ten times

```
        MOV  A, #0      ; A=0 clear ACC
        MOV  R2, #10    ; load counter R2=10
AGAIN:  ADD  A, #03     ; add 03 to ACC
        DJNZ R2, AGAIN  ; Repeat until
                        R2=0 (10 times)
        MOV  R5, A      ; Save A in R5
```

Instruction Set

(1) JUMP, Loop and CALL Instructions →

Looping in the 8051 → Repeating a sequence of instructions a certain Number of times is called a loop. The loop is one of most widely used actions that any microprocessor performs.

In the 8051, the loop action is performed by the instruction "DJNZ reg, label". In this instruction, the Register is decremented; if it is not zero it jumps to the target address referred to by the label. Perior to the start of the loop the register is loaded with the counter for the Number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.

Que write a program to

(a) load the accumulator with the value 55H

(b) complement the Acc 7co times

Sol since 7co is larger than 255 (the maximum Capacity of any register), we use two Registers to hold the count. The following code show how to use R_2 and R_3 for the count.

```
        MOV   A, #55H          ; A=55H
        MOV   R3, #10          ; R3=10, the outer loop
                               count
Next:   MOV   R2, #70          ; R2=70, the inner loop
                               count
AGAIN:  CPL   A                ; complem A Register
        DJNZ  R2, AGAIN        ; Repeat it 70 times
        DJNZ  R3, NEXT
```

Other Conditional jumps →

Instruction	Action
JZ	Jump if A=0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and jump if Register ≠ 0
CJNE A, data	Jump if A ≠ data
CJNE reg, #data	Jump if byte ≠ #data
JC	Jump if Cy=1
JNC	Jump if Cy=0
JB	Jump if bit=1
JNB	Jump if bit=0
JBC	Jump if bit=1 and clear bit

8 write a program to determine if R_5 contains the value 0. if so, put 55H in it

Sol

```
MOV  A, R5      ; copy R5 to A
JNZ  NEXT       ; Jump if A is not zero
MOV  R5, #55H
Next: ---
```

JZ (Jump if A=0)

In this instruction the content of Register A is checked. if it is zero, it jumps to the target address

Ex

```
MOV  A, R0      ; A=R0
JZ   OVER       ; jump if A=0
MOV  A, R1      ; A=R1
JZ   OVER       ; jump if A=0
--
OVER:
```

In this program, if either R_0 or R_1 is zero it jumps to the label OVER. Notice that the JZ instruction can be used only for register A. it can only check to see whether the accumulator is zero. and it does not apply to any other Register.

we don't have to perform an airthmetic instruction such as decrement to use the JNZ instruction.

JNC (Jump if no Carry, jumps if Cy=0)

In this instruction, the Carry flag bit in the flag (PSW) register is used to make the decision whether to jump. In executing "JNC label" the processor looks at the Carry flag to see if it is raised (Cy=1). if it is not, the CPU starts to fetch and execute instructions from the address of the label. if Cy=1, it will not jump but will execute the next instruction below JNC

Note that there is also a "JC label" instruction. in the JC instruction, if Cy=1 it jumps to the target address.

Qu find the sum of the values 79H, F5H and E2H. Put the sum in registers R0 (low byte) and R5 (high byte)

Sol

```
        MOV A, #0          ; clear A (A=0)
        MOV R5, A          ; clear R5
        ADD A, #79H        ; A=0+79H=79H
        JNC step1          ; if No Carry, add Next No
        INC R5             ; if Cy=1, increment R5
step1:  ADD A, #0F5H       ; A=79+F5=6E and cy=1
        JNC step2          ; Jump if cy=0
        INC R5             ; if cy=1 then increment (R5=1)
step2:  ADD A, #0E2H       ; A=6E+E2=50 and cy=1
        JNC OVER
        INC R5
OVER:   MOV R0, A          ; Now R0=50H, and R5=02
```

→ All conditional jumps are short jumps

It must be Noted all conditional jumps are short jumps, meaning that the address of the target must be within -128 to +127 bytes of the contents of the Program Counter (PC)

Unconditional jump instructions →

The Unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051 there are two Unconditional jumps LJMP (long jump) and SJMP (short jump)

LJMP → it is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH

SJMP (short jump) instruction, which is a 2-byte instruction as opposed to the 3-byte LJMP instruction. This can save some bytes of memory in many applications where memory space is in short supply

SJMP →

In this 2-byte instruction, the first byte is the opcode and the second byte is relative address of the target location. The relative address range of 00-FFH is

divided into forward and backward jumps that is within -128 to +127 bytes of memory relative to the address of the current PC. If the Jump is forward, the target address can be within a space of 127 bytes from the current PC.

Calculating the short jump address →

In addition to the SJMP instruction, all Conditional jumps such as JNC, JZ, and DJNZ are also short jumps due to the fact that they are all two-byte instruction. In these instruction the first byte is the opcode and the second byte is the relative address.

The target address is relative to the value of the Program Counter.

TO calculate the target address, the second byte is added to the PC of the instruction immediately below the jump.

Pry Using the following list File, verify the jump forward address Calculation

PC	opcode	Mnemonic	operand
		ORG	0000
0000			
0000	7800	MOV	R0, #0
0002	7455	MOV	A, #55H
0004	6003	JZ	NEXT
0006	08	INC	R0

PC	Opcode		
0007	04	AGAIN:	INC A
000B	04		INC A
0009	2477	NEXT:	ADD A, #77H
000B	5005		JNC OVER
000D	E4		CLR A
000E	F8		MOV R0, A
000F	f9		MOV R1, A
0010	FA		MOV R2, A
0011	FB		MOV R3, A
0012	2B	OVER:	ADD A, R3
0013	50F2		JNC AGAIN
0015	80FE	HERE:	SJMP HERE
0017			END

→ JN and JNC instruction both jump forward. The target address for the forward jump is calculated by adding the PC of the following instruction to the second byte of the short jump instruction. which is called the relative address

In line 4 the instruction J2 NEXT has opcode of 60 and operand of 03 at the address of 0004 and 0005

CALL Instructions →

is used to call a Subroutine. Subroutine are often used to perform tasks that need to be performed frequently. This makes a program more structured in adding to saving memory space. In the 8051 there are two instructions for call: LCALL (long call) and ACALL (absolute call)

LCALL (long call)

In this 3-byte instruction, the first byte is the opcode and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call Subroutines located anywhere within the 64K-byte address space of the 8051

Q4 Write a program to toggle all the bits of port 1 by sending to it the values 55H and AAH continuously. Put a time delay in between each issuing of data to port 1

~~This program will be used to test the port of the~~

~~8051~~

Sol

```
        ORG   0
BACR:   MOV   A, #55H      ; load A with 55H
        MOV   P1, A        ; send 55H to port 1
        LCALL DELAY        ; time delay
        MOV   A, #0AAH     ; load A with AA (in hex)
        MOV   P1, A        ; send AAH to port 1
        LCALL DELAY.
        SJMP  BACR         ; keep doing this
                           ; definitely this is
                           the delay subroutine

        ORG   300H         ; put time delay at address 300H
DELAY:  MOV   R5, #0FFH    ; R5 =255 (FF in hex), the
                           counter
AGAIN:  DJNZ  R5, AGAIN    ; stay here until R5
                           becouse 0
        RET                ; return to caller (when R5=0)
        END                ; end of asm file.
```

ACALL — ACALL is a 2-byte instruction in contrast to LCALL, which is 3 bytes since ACALL is a 2-byte instruction, the target address of the subroutine must be within 2K bytes because only 11 bits of the 2 byte are used for the address there is no difference between ACALL and LCALL in terms of saving the Program counter on the stack or the function of the RET instruction.

The only difference is that the target address for LCALL

In many variations of the 8051 marketed by different companies, on chip Rom is as low as 1K byte. In such cases, the use of ACALL instead of LCALL can save a No of bytes of Program Rom space.

R Reg-

```
        ORG 0
        MOV  A, #55H      ; load A with 55H
BACK:   MOV  P1, A        ; issue value in reg A to port1
        ACALL DELAY       ; time delay
        CPL  A            ; complement reg A
        SJMP BACK         ;

DELAY:
        MOV  R5, #0FFH    ; R5 = 255 (FF in hex)
AGAIN:  DJN2 R5, AGAIN    ; stay Here until R5
                          becomes 0
        RET               ; Return to caller
        END               ; end of ASM file
```

Roll-over timer flag and interrupt →

If the timer interrupt in the IE Register is enabled, whatever the timer rolls over, TF is raised and the micro controller is interrupted in whatever it is doing. and Jump to the interrupt vector table to service the ISR.

Eg write a Prg that continuosly gets 8-bit data from P_0 and sends it to P_1 while simultaneously creating a square wave of 200 μs period on Pin2 P2.1 use Timer 0 to create the square wave Assume that XTAL = 11.0592 MHz

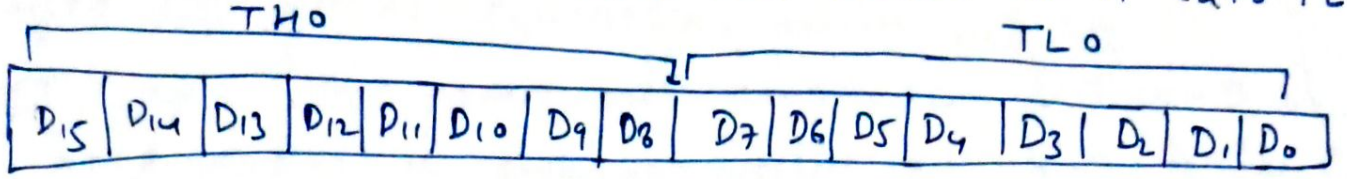
Sol

→ Both Timer 0 and Timer 1 are 16 bits wide Since the 8051 has an 8-bit architecture each 16 bit timer is accessed as two separate Registers of low byte and high byte,

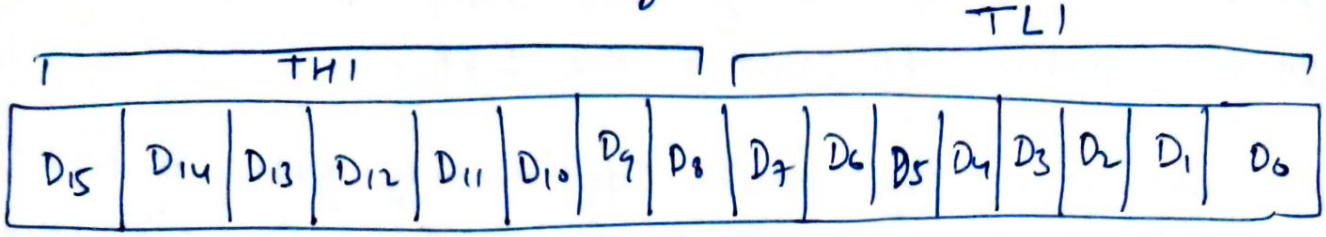
Timer 0 Registers

The 16 bit Register of Timer 0 is ~~asss~~ accessed as low byte and high byte. The low byte Register is called TL0 (Timer 0 Low byte) And the high byte register is reffered to as TH0 (Timer 0 high byte) These Register Can be accessed like any other Register

Such as A, B, R0, R1, R2 etc for Ex. the instruction MOV TL0, #4FH moves the value 4FH into TL0



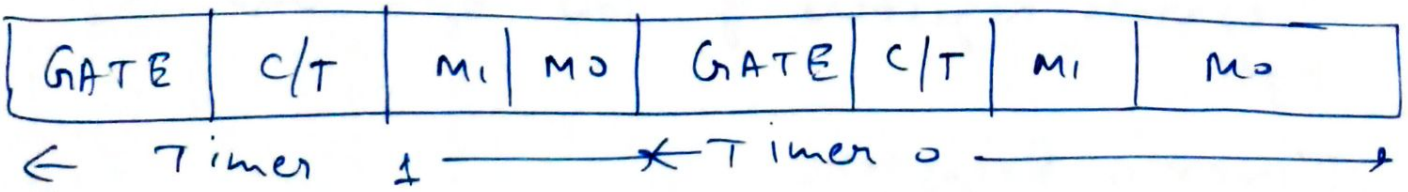
Timer 0 Register



Timer 1 Register

TMOD (timer mode) Register –

Both timers 0 and 1 use the same Register Called TMOD, to set the various timer operation modes. TMOD is an 8 bit register in which the lower 4 bits are set aside for timer 0 and the upper 4 bits for timer 1. In each case the lower 2 bits are set the timer mode and the upper 2 bits to specify the operation



GATE → Grating control when set. The timer/counter is enabled only while the INTx Pin is High and TRx Contol Pin is set when cleared, the timer is enabled whenever the TRx control bit is set

C/T – Timer or counter selected cleared for timer operation (Input from internal system clock) set the counter operation (input from Tx Pin)

M_1 — mode bit 1

M_0 — mode bit 0

M_1	M_2	mode	operating mode
0	0	0	13-bit mode 8 bit timer/counter
0	1	1	16 bit timer mode
1	0	2	8 bit auto Reload
1	1	3	split timer mode

Q Indicate which mode and which timer is selected

(1) MOV TMOD, #01H

TMOD = 00000001 — Mode 1 of timer 0

(2) MOV TMOD, #20H

TMOD = 00100000 Mode 2 of timer 1

(3) MOV TMOD, #12H

TMOD = 00010010 — Mode 2 timer 0 and mode 1 of timer 1

C/T (Clock/timer) – This bit in the TMOD register is used to decide whether the timer is used as a delay generator or an event counter. if C/T = 0, it is used as a timer for time delay generation. The crystal freq. attached to the 8051 is the source of the clock for the timer. This means that the size of the crystal freq. attached to the 8051 also decides the speed at which the 8051 timer ticks. The freq. for the timer is Always 1/12 the freq. of the crystal attached to the 8051

Que Find the timer's clock frequency and its period for various 8051-based systems with the following crystal frequency

(a) 12 MHz

$1/12 \times 12\ MHz = 1\ MHz$ and $T = 1/1\ MHz = 1\ \mu s$

(b) 16 MHz

$1/12 \times 16\ MHz = 1.333\ MHz$ and $T = \frac{1}{1.333\ MHz}$

$= .75\ \mu s$

(c) 11.0592 MHz

$1/12 \times 11.0592\ MHz$

$= 921.6\ KHz$

$T = 1/921.6\ KHz = 1.085\ \mu s$

→ Although various 8051-based systems have an XTAL frequency of 10 MHz to 40 MHz

we will concentrate on the XTAL freq of 11.0592 MHz The reason behind such an odd No. has to do with the baud rate for serial communication of the 8051 XTAL = 11.0592 MHz allows the 8051 system to communicate with the IBM PC with no error

GATE → what is the purpose of using GATE? every timer has a means of starting and stopping. Some timer do this by Software, some by Hardware and some have both Software and hardware Controls.

The timer in the 8051 have both. The start and STOP of the timer are controlled by way of Software by the TR (timer start) bit TR0 and TR1. This is achieved by the instructions, "SETB TR1 and CLR TR1 for timer 1 and "SETB TR0" and CLR TR0 for timer 0. The SETB instruction starts it. and it is stopped by the CLR instruction

These instruction start and stop the timer as long as GATE = 0 in the TMOD register.

The hardware way of starting and stopping the timer by an external source is achieved by

Making GATE = 1 in the TMOD register

Que Find the value for TMOD if we want to Program Timer 0 in mode 2. use 8051 XTAL for the clock source, and use instructions to start and stop the timer.

Sol TMOD = 0000 0010 Timer 0, mode 2

C/T = 0 to use XTAL clock source and
gate = 0 to use Internal (Software) start and stop method

Mode 1 programming →

(1) it is a 16 bit timer, therefore it allows values of 0000 to FFFFH to be loaded into the timer's registers TL and TH

(2) After TH and TL are loaded with a 16 bit initial value, the timer must be started. This is done by "SETB, TR0" for timer 0 and SETB TR1 for timer 1

(3) After the timer is started it starts to count up it counts up until it reaches its limit of FFFFH when it Rolls over from FFFFH to 0000H it sets high a flag bit Called Tf (timer flag) This timer flag can be Monitored. when the timer flag is raised one option would be to stop the timer with the instructions CLR TR0 or CLR TR1 for Timer 0 and Timer 1. Again it must be noted that each timer has its own timer flag. TF0 for timer 0 and TF1 for timer 1

(4) After the timer reaches its limit and rolls over in order to repeat the process the registers, TH and TL must be reloaded with the original value and Tf must be reset to 0

Steps to program in Mode 1

To generate a time delay using the timer mode 1 the following steps are taken

(1) Load the TMOD value Register indicating which timer (Timer 0 or timer 1) is to be used and which timer mode (0 or 1) is selected.

(2) Load Register TL and TH with initial count value

(3) start the timer

(4) Keep monitoring the timer flag (TF) with the "JNB TFx, target" instruction to see if it is raised get out the loop when TF becomes High

(5) stop the timer

(6) clear the TF flag for the next round

(7) go back to step 2 to Load TH and TL again

Qu In the following Program, we are creating a square wave of 50% duty cycle (with equal portions high and Low) on the P1.5 bit. Timer 0 is used to generate the time delay. Analyze the Program

```
HERE:  MOV  TMOD, #01   ;Timer 0 mode 1
       MOV  TL0, #0F2H  ; TL0=F2H the Low byte
       MOV  TH0, #0FFH  ; TH0=FFH the High byte
       CPL  P1.5        ; toggle P1.5
       ACALL DELAY
       SJMP HERE        ; Load TH, TL again

DELAY
       SETB TR0         ; start Timer 0

AGAIN  JNB  TF0, AGAIN  ; monitor Timer 0 flag until it
       CLR  TR0 ; stop timer 0   Rolls over
       CLR  TF0 ; clear timer 0 flag.
       RET
```

Sol (1) TMOD is loaded

(2) FFF2H is Loaded into TH0 – TL0

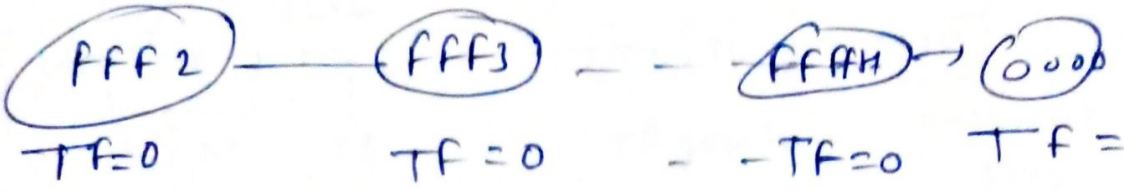
(3) P1.5 is toggled for the high and Low portions of the Pulse

(4) The DELAY subroutine using the timer is called

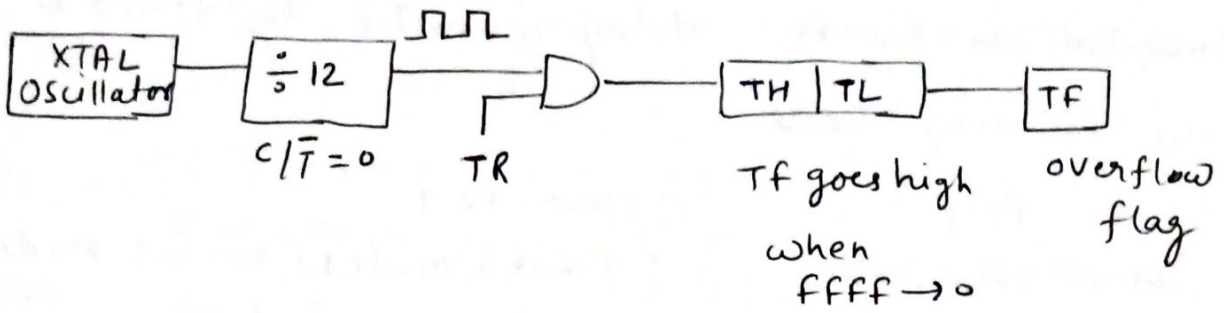
(5) In the DELAY subroutine, Timer 0 is started by the SETB TR0 instruction

(6) Timer 0 counts up with the passing of each clock, which is provided by the crystal oscillator As the timer counts up, it goes through the states of FFF3, FFF4, FFF5, FFF6 - - - - FFFB and so on until it Reaches FFFFH One more clock rolls it to 0, raising the timer flag (TF0 = 1) at that point the JNB instruction fall through

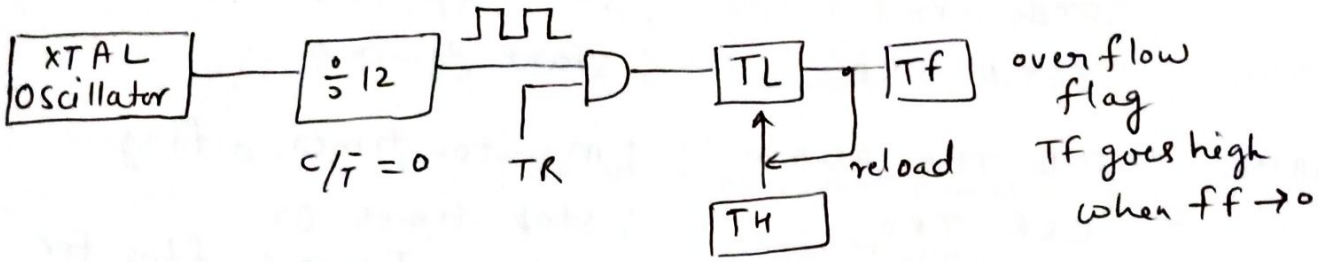
(7) Timer 0 is stopped by instruction CLR TR0 The DELAY subRoutine End.



Mode 1 (16 bit)



Mode 2 (8 bit - auto-Reload)



Ex In previous Example calculate the amount of Time delay in the delay Subroutine generated by the timer.

Sol Timer works with a clock freq. of 1/12 of XTAL freq. we have 11.0592 MHz/12 = 921.6 KHz as the timer frequency As a result each clock has a period of

$T = 1/921.6\ KHz = 1.085\ \mu s$

Timer 0 counts up each 1.085 μs resulting in delay = No of Counts × 1.085 μs

The No of Counts for the rollover is ffff H - fff2H = 0 DH (13 decimal) However we add one to 13 because of extra clock needed when it rolls over from ffff H to 0 and Raises the TF flag

This gives 14 × 1.085 μs = 15.19 μs for half the pulse

for the Entire period T = 2 × 15.19 μs = 30.38 μs

gives us the time delay generated by the timer

Ex In the example (Previous) Calculate the ~~freq of Square wave generated an fin ~~ft~~.s~~ delay generated by timer 0 in the following code

```
        CLR   P2.3          ; clear P2.3
        MOV   TMOD, #01     ; Timer 0, mode 1 (16-bit mode)
HERE:   MOV   TL0, #3EH     ; TL0 = 3EH, Low byte
        MOV   TH0, #0BBH    ; TH0 = BBH, High Byte
        SETB  P2.3          ; SET High P2.3
        SETB  TR0           ; start timer 0
AGAIN:  JNB   TF0, AGAIN    ; Monitor timer 0 flag
        CLR   TR0           ; stop timer 0
        CLR   TF0           ; clear Timer 0 flag for
                              next round
        CLR   P2.3
```

Sol(a) ffff - BB3E +1 = 47C2H = 18370 in decimal and

18370 X 1.085 MS = 19.93145 ms

Delay Calculating formula

(a) In Hex

(ffff - YYXX +1) X 1.085 MS

where YYXX are TH, TL initial values respectively

Notice that values YYXX are in hex

In decimal

Convert YYXX values of the TH, TL register to decimal to get a NNNNN decimal ND then (65536 - NNNNN) X1.085 MS.

Q2 Modify TL and TH in Previous example to get the largest time delay Possible. find the delay in ms

In your Calculation, exclude the overhead due to the

instruction in the loop

Sol To get the largest delay we make TL and TH both 0 this will count up from 0000 to FFFF H and then roll over to zero

```
        CLR   P2.3          ; clear P2.3
        MOV   TMOD,#01      ; Timer 0, mode 1 (16-bit mode)
HERE:   MOV   TL0,#0        ; TL0=0, Low byte
        MOV   TH0,#0        ; TH0=0, High byte
        SETB  P2.3          ; SET P2.3 high
        SETB  TR0           ; start timer 0
AGAIN:  JNB   TF0,AGAIN     ; Monitor Timer 0 flag
        CLR   TR0           ; stop Timer 0
        CLR   TF0           ; clear Timer 0 flag
        CLR   P2.3
```

Making TH and TL both zero means that the timer will count from 0000 to FFFFH and then roll over to raise the TF flag. As a Result, it goes through a total of 65536 states therefore, we have delay

$$(65536-0)\times 1.085\ \mu s = 71.1065\ ms$$

Ex The following program generates a square wave on Pin P1.5 Continuously using Timer 1 for a time delay. find the frequency of square wave if XTAL = 11.0592 MHz in your Calculation ~~do not include the~~

```
        MOV   TMOD, #10H    ; Timer 1, mode 1 (16 bit)
AGAIN:  MOV   TL1, #34H     ; TL1 =34H, Low byte
        MOV   TH1, #76H     ; TH1=76H, High byte
```

```
        SETB   TR1              ; Start Timer 1
BACK    JNB    TF1, BACK        ; Stay until timer rolls over
        CLR    TR1              ; stop Timer 1
        CPL    P1.5             ; comp P1.5 to get hi, Lo
        CLR    TF1              ; clear Timer 1 flag
        SJMP   AGAIN            ; reload timer
                                   Since Mode 1 is not
                                   Auto-reload
```

Sol In the above program Notice the target of SJMP in Mode 1, the Program must reload the TH, TL Register every time if we want to have a continuous wave. Now the Calculation Since FFFFH - 7634H = 89CBH + 1 = 89CCH and B9CCH = 35276 clock count

35276 X 1.085 μs = 38.274 ms for half of the Square wave. The entire square wave length is 38.274 x 2 = 76.548 ms and has a freq = 13.064 Hz

Finding values to be loaded into the timer —

Steps

(1) Divide the desired time delay by 1.085 μs

(2) Perform 65536 - n where n is the decimal value we got in step 1

(3) Convert the result of step 2 to Hex., where yyxx is the initial hex value to be loaded into the timer's registers

(4) set TL = Xx and TH = Yy

Pg Assume that XTAL = 11.0592 MHz. what value do we need to load into the timer's register if we want to have a time delay of 5 ms ? show the Program for Timer 0 to create a Pulse width of 5 ms on P2.3

Sol XTAL = 11.0592 MHz, the counter counts up every 1.085 μs. This means that out of many 1.085 μs intervals we must make a 5 ms Pulse.

To get that, we divide one by other we need

5 ms / 1.085 μs = 4608 clocks. TO achieve that we need to load into TL and TH the value 65536 − 4608 = 60928 = EE00H. Therefore we have TH = EE and TL = 00

```
        CLR   P2.3
        MOV   TMOD, #01          Timer 0  Mode 1
HERE    MOV   TL0, #0
        MOV   TH0, #0EEH
        SETB  P2.3
        SETB  TR0
AGAIN:  JNB   TF0, AGAIN
        CLR   P2.3
        CLR   TR0
        CLR   TF0
```

Pg Assuming that XTAL = 11.0592 MHz, write a Program to generate a square wave of 2 KHz freq. on Pin P1.5

Sol

(a) $T = 1/f = 1/2\,kHz = 500\,\mu s$ the period of the square wave

(b) $1/2$ of it for the high and Low portion of the Pulse is $250\,\mu s$

(c) $250\,\mu s = 1/1.085\,\mu s = 230$ and $65536 - 230 = 65306$ which in Hex is $FF1AH$

(d) $TL = 1AH$ and $TH = FFH$, all in Hex, The program is as follows

Sol

```
        MOV  TMOD, #10H
AGAIN:  MOV  TL1, #1AH
        MOV  TH1, #0FFH
        SETB TR1
BACK    JNB  TF1, BACK
        CLR  TR1
        CPL  P1.5
        CLR  TF1
        SJMP AGAIN
```

Pg Examine the following program and find the time delay in seconds. ~~Exectude the~~

Sol

```
        MOV  TMOD, #10H     ; Timer 1, Mode 1
        MOV  R3, #200       ; Counter for multiple delay
AGAIN:  MOV  TL1, #06H      ; TH1=06, Low cycle byte
        MOV  TH1, #01H      ; TH1=01, High byte
        SETB TR1            ; Start timer 1
BACK:   JNB  TF1, BACK      ; stay until timer Rolls over
        CLR  TR1            ; stop timer 1
        CLR  TF1            ; clear timer 1 flag
        DJNZ R3, AGAIN      ; if R3 not zero then
                            reload timer
```

Sol TH-TL = 0106H = 264 in decimal and 65536 − 264 = 65272
Now 65272 X 1.085 MS = 70.820 ms and for 200 of them we have 200 x 70.820 ms = 14.164024 second

Mode 2 Programming

Pg Assuming that XTAL = 11.0592 MHz find (a) the freq. of the square wave generated on Pin P1.0 in the following Program and (b) the Smallest frequency achievable in this program, and the TH value to do that

Sol

```
        MOV  TMOD, #20H     ; T1/mode/2/8-bit/auto-reload
        MOV  TH1, #5        ; TH1=5
        SETB TR1            ; start Timer 1
BACK:   JNB  TF1, BACK      ; stay until timer rolls over
        CPL  P1.0           ; Comp. P1.0 to get hi, lo
        CLR  TF1            ; clear Timer 1 flag
        SJMP BACK           ; Mode 2 is auto-reload
```

Sol (a) First Notice the target address of SJMP. in mode 2 we do not need to reload TH since it is auto reload. Now $(256-05) \times 1.085\ \mu s = 251 \times 1.085\ \mu s$ $= 272.33\ \mu s$ is the high portion of the Pulse. Since it is a 50% duty cycle Square wave, the Period T is twice that

as a Result $T = 2 \times 273.33\ \mu s = 544.67\ \mu s$ and the frequency $= 1.83597$ KHz

(b) TO get the Smallest freq., we need the largest T and that is achieved when TH=00 in that case we have $T = 2 \times 256 \times 1.085\ \mu s = 555.52\ \mu s$ and the freq. $= 1.8$ KHz

Prg find the frequency of a square wave generated on Pin P1.0

Sol

```
        MOV   TMOD ,#2H     ; Timer 0, mode 2
        MOV   TH0, #0       ; TH0=0
AGAIN:  MOV   R5, #250      ; count for multiple delay
        ACALL DELAY         ;
        CPL   P1.0          ; toggle P1.0
        SJMP  AGAIN         ; repeat
DELAY:  SETB  TR0           ; start timer 0
BACK:   JNB   TF0, BACK     ; stay until timer rolls over
        CLR   TR0           ; stop timer 0
        CLR   TF0           ; clear TF for next round
        DJNZ  R5, DELAY
        RET
```

Sol $T = 2(250 \times 256 \times 1.085\ \mu s) = 138.88$ ms and freq $= 7.2$ Hz

8051 Interrupts →

→ A single microcontroller can serve several devices There are two ways to do that:

1. interrupts
2. polling

→ In the interrupt method, when any device needs its service, the device notifies the microcontroller by sending it an interrupt signal, upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the devices. The program associate with the interrupt is called the interrupt service Routine (ISR) or Interrupt handler

→ The polling, the microcontroller continuously monitors the status of a given device, when the status condition is met, it performs the service. after that it moves on to monitor the Nent device until each one is serviced.

Six interrupts in the 8051 →

1. Reset – when the reset pin is activated, the 8051 jumps to address location 0000,
2. Two interrupts are set aside for the timers One for Timer 0 and one for timer 1. memory locations 000BH and 001BH in the interrupt vector table belong to timers 0 and timer 1
3. Two interrupts are set aside for hardware external hardware interrupts.

Pin No 12 (P3.2) and 13 (P3.3) in Port 3 are for the external hardware interrupts INT0 and INT1

→ These external interrupts are also referred to as EX1 and EX2. memory locations 0003H and 0013H in the interrupt vector table are assigned to INT0 and INT1

④ Serial communication ~~are~~ ~~assigned~~ has a single interrupt that belongs to both receive and transmit The interrupt vector table location 0023H belongs to this interrupt.

Interrupt	Rom location (HEX)	Pin	flag clearing
Reset	0000	9	Auto
External hardware interrupt 0 (INT0)	0003	P3.2 (12)	Auto
Timer 0 interrupt (TF0)	000B		Auto
External hardware interrupt 1 (INT 1)	0013	P3.3(13)	Auto
Timer 1 interrupt (TF1)	001B		Auto
Serial Com interrupt (R_I and T_I)	0023		Programmer clears it

Enabling an interrupt → (Interrupt Enable Register)

D7							D0
EA	-	ET2	ES	ET1	EX1	ET0	EX0

EA. IE.7 Disables all interrupts. if EA=0, no interrupt is Acknowledged if EA=1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

ET2 IE.5 Enables or disables Timer 2 overflow or Capture interrupt (8052 only)

ES IE.4 Enables or disables the serial port interrupt

ET1 IE.3 Enables or disables Timer 1 overflow interrupt.

EX1 IE.2 Enables or disables external interrupt 1

ET0 IE.1 Enables or disables Timer 0 overflow interrupt.

EX0 IE.0 Enables or disables external interrupt 0

Ex show the instructions to

(a) enable the serial interrupt, Timer 0 interrupt and external hardware interrupt 1 (EX1)

(b) disable (mask) the timer 0 interrupt,

(c) show how to disable all the interrupts with a single instruction

Sol

(a) MOV IE, #10010110B ; enable serial Timer 0, EX1 Since IE is a bit

addressable register, we can use the following instructions to access individual bits of the register

```
(b)   CLR  IE.1     ; mask (disable) Timer 0
                      interrupt only
(c)   CLR  IE.7     ; disable all interrupts
```

Another way to perform the "MOV IE, #10010110B" instruction is by using single bit instructions as shown

```
SETB   IE.7     ; EA=1, global Enable
SETB   IE.4     ; enable serial Interrupt
SETB   IE.1     ; enable Timer 0 interrupt
SETB   IE.2     ; enable EX1
```

Prg Assume that the INT1 Pin is connected to a switch that is Normally high. whenever it goes low, it should turn on an LED. The LED is connected to P1.3 and is normally off. when it is turned on it should stay on for a fraction of a second. As long as the switch is pressed low, the LED should stay on.

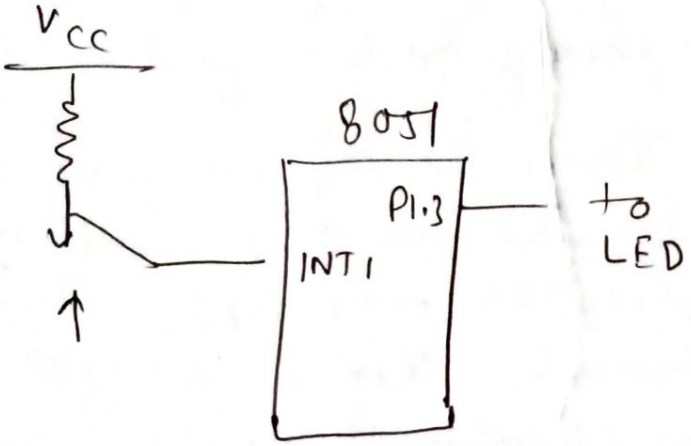
Sol

```
        ORG   0000H
        LJMP  MAIN          ; bypass interrupt vector table
--- ISR for hardware interrupt INT1 to turn on the
    LED
        ORG   0013H         ; INT1 ISR
        SETB  P1.3          ; turn on LED
        MOV   R3, #255      ; load counter
```

```
BACK:   DJNZ   R3, BACK   ; Keep LED on for a
                            while
        CLR    P1.3       ; turn off the LED
RET:                       ; Return from ISR
```

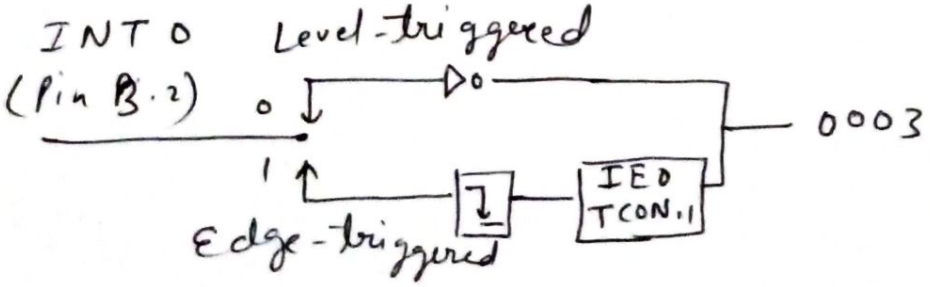
Main Program for initialization

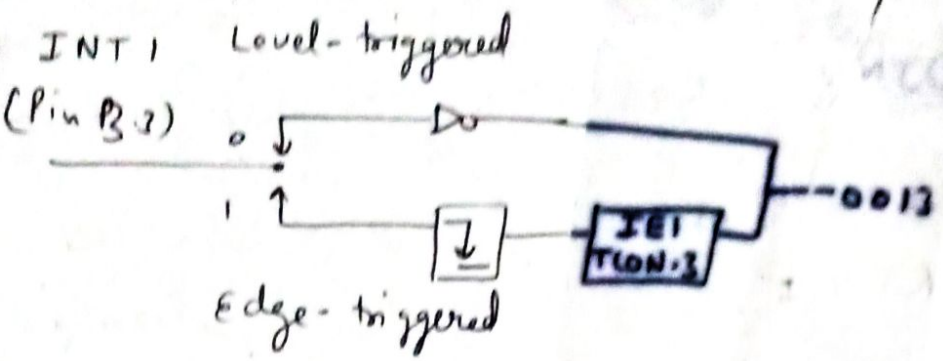
```
        ORG 300H
main:   MOV    IE, #10000100B  ; enable External
                                 INT1
HERE:   SJMP   HERE       ; stay here until interrupted
        END
```



Programming External Hardware Interrupts →

The 8051 has two external hardware interrupts Pin 12 (P3.2) and Pin 13 (P3.3) of the 8051, designated as INT0 and INT1 are used as external hardware interrupts,





External interrupts INT0 and INT1 -

There are two types of activation for the external hardware interrupts

(1) Level triggered
(2) edge triggered

(1) Level-triggered interrupt -

INT0 } — Normaly High and if Low
INT1 }

Signal is applied to them, it triggers the interrupt. Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt. This is called a level-triggered or level-activated Interrupt and is the default mode upon reset of the 8051

→ The low level signal at the INT Pin must be removed before the execution of the Last instruction of the interrupt service routine RETI otherwise another interrupt will be generated