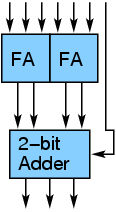
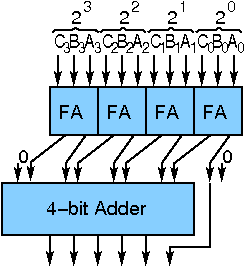
LAB\_6

A) Design of Wallace Tree Adders :

There are many cases where it is desired to add more than two numbers together. The straightforward way of adding together m numbers (all n bits wide) is to add the first two, then add that sum to the next using cascading full adders. This requires a total of m − 1 additions, for a total gate delay of O(m lg n) (assuming lookahead carry adders). Instead, a tree of adders can be formed, taking only O(lg m · lg n) gate delays.A Wallace tree adder adds together n bits to produce a sum of log2n bits.The design of a Wallace tree adder to add seven bits (W7) is illustrated below:



An adder tree to add three 4-bit numbers is shown below:



B) Write an assembly language program for calculating the factorial of a number using 8086 microprocessor

Algorithm –

1. Input the Number whose factorial is to be find and Store that Number in CX Register (Condition for LOOP Instruction)
2. Insert 0001 in AX(Condition for MUL Instruction) and 0000 in DX
3. Multiply CX with AX until CX become Zero(0) using LOOP Instruction
4. Copy the content of AX to memory location 0600
5. Copy the content of DX to memory location 0601
6. Stop Execution

| ADDRESS | MNEMONICS | COMMENTS |
| --- | --- | --- |
| 0400 | MOV CX, [0500] | CX <- [0500] |
| 0404 | MOV AX, 0001 | AX <- 0001 |
| 0407 | MOV DX, 0000 | DX <- 0000 |
| 040A | MUL CX | DX:AX <- AX \* CX |
| 040C | LOOP 040A | Go To [040A] till CX->00 |
| 0410 | MOV [0600], AX | [0600]<-AX |
| 0414 | MOV [0601], DX | [0601]<-DX |
| 0418 | HLT | Stop Execution |

Explanation –

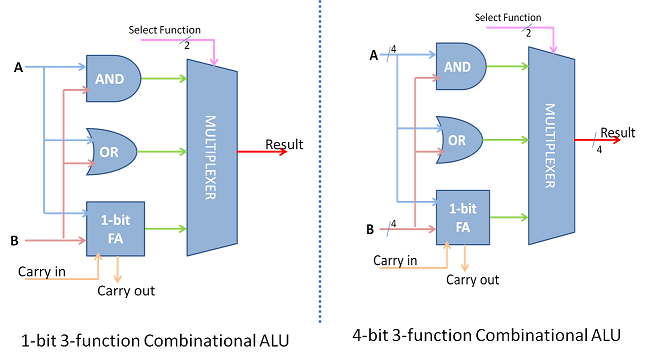
1. MOV CX, [0500] loads 0500 Memory location content to CX Register
2. MOV AX, 0001 loads AX register with 0001
3. MOV DX, 0000 loads DX register with 0000
4. MUL CX multiply AX with CX and store result in DX:AX pair
5. LOOP 040A runs loop till CX not equal to Zero
6. MOV [0600], AX store AX register content to memory location 0600
7. MOV [0601], DX store DX register content to memory location 0601
8. HLT stops the execution of program

LAB-7

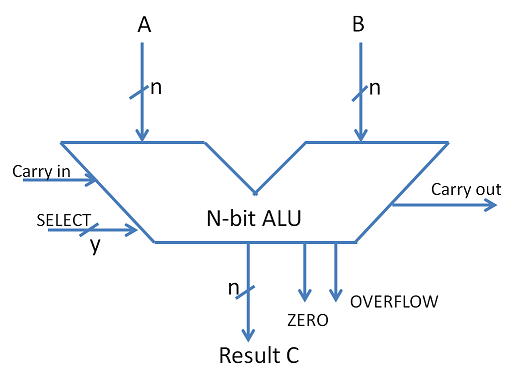
A) Arithmetic Logic Unit

An arithmetic-logic unit is the part of a central processing unit that carries out arithmetic and logic operations on the operands in computer instruction words. In some processors, the ALU is divided into two units: an arithmetic unit (AU) and a logic unit (LU).

The design specifications of ALU are derived from the Instruction Set Architecture. The ALU must have the capability to execute the instructions of ISA. An instruction execution in a CPU is achieved by the movement of data/datum associated with the instruction. This movement of data is facilitated by the Datapath. For example, a LOAD instruction brings data from memory location and writes onto a GPR. The navigation of data over datapath enables the execution of LOAD instruction. We discuss Datapath more in details in the next chapter on Control Unit Design. The trade-off in ALU design is necessitated by the factors like Speed of execution, hardware cost, the width of the ALU.



The simplest ALU has more functions that are essential to support the ISA of the CPU. Therefore the ALU combines the functions of 2's complement, Adder, Subtractor, as part of the arithmetic unit. The logical unit would generate logical functions of the form f(x,y) like AND, OR, NOT, XOR etc. Such a combination supplements most of a CPU's fixed point data processing instructions.



B) Assembly language program to find LCM.

.MODEL SMALL

.DATA

Num1 DW 250

Num2 DW 100

Ans DW ?

.CODE

MOV AX,@DATA

MOV DS, AX

MOV AX, Num1

MOV BX, Num2

MOV DX, 0000h

NEXT: PUSH AX

PUSH DX

DIV BX

CMP DX, 0000h

JZ LAST

POP DX

POP AX

ADD AX, Num1

JNC NEXT

INC DX

JMP NEXT

LAST: POP Ans+2

POP Ans

MOV AH, 4Ch

INT 21h

END

LAB 8

A)Verification of Registers and Counters:-

Design of Registers and Counterss :

In a sequential circuit the present output is determined by both the present input and the past output. In order to receive the past output some kind of memory element can be used. The memory element commonly used in the sequential circuits are time-delay devices. The block diagram of the sequential circuit-



A circuit with flip-flops is considered a sequential circuit even in the absence of combinational logic. Circuits that include flip-flops are usually classified by the function they perform. Two such circuits are registers and counters:

Register is a group of flip-flops. Its basic function is to hold information within a digital system so as to make it available to the logic units during the computing process.

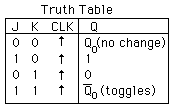
Counter is essentially a register that goes through a predetermined sequence of states.

There are various different kind of Flip-Flops. Some of the common flip-flops are: R-S Flip-Flop, D Flip-Flop, J-K Flip-Flop, T Flip-Flop. The block diagram of different flip-flops are shown here –



 RS flipflop If R is high then reset state occurs and when S=1 set state.the both cannot be high simultaneouly. this input combination is avoided.

 JK flipflop If J and K are both low then no change occurs. If J and K are both high at the clock edge then the output will toggle from one state to the other.



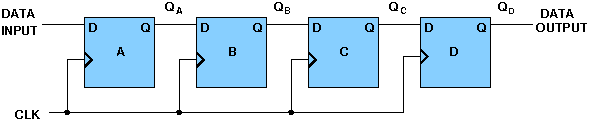
 D flipflop The D flip-flop tracks the input, making transitions with match those of the input D. It is used as data store.

 Tflipflop The T or "toggle" flip-flop changes its output on each clock edge,

Types of Registers:

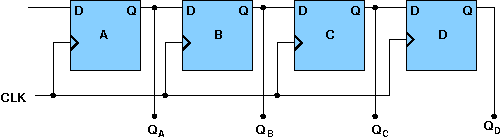
4-bit Serial-in Serial-out

4 bit serial-in serial-out register accepts digital data serially that is one bit at the time on one line. It produces the stored information on its output also in serial form. This is a shift register, as The binary number is "Shifted" one bit at time from one flip flop to the next. The block diagram is-



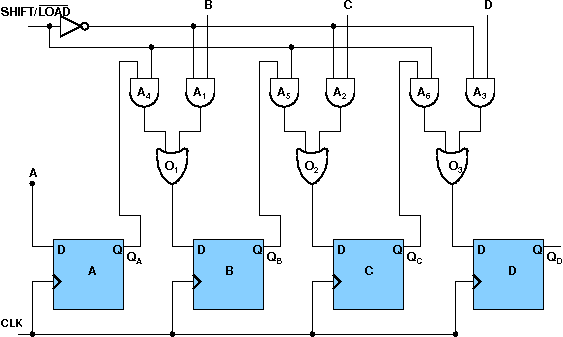
4-bit Serial-in Parallel-out

In serial-in parallel-out register the data are loaded serially and read out in parallel. The block diagram is



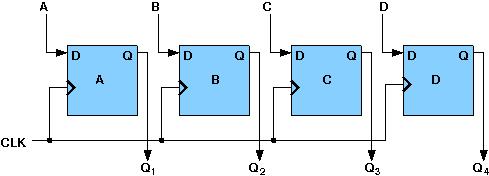
4-bit Parallel-in Serial-out

In parallel-in serial out register the bits are entered simultaneously into their respective stages on parallel-lines, rather than on a bit-by-bit basis on one line as with serial data inputs and output is read out out parallaly. The block diagram is-



4-bit Parallel-in Parallel-out

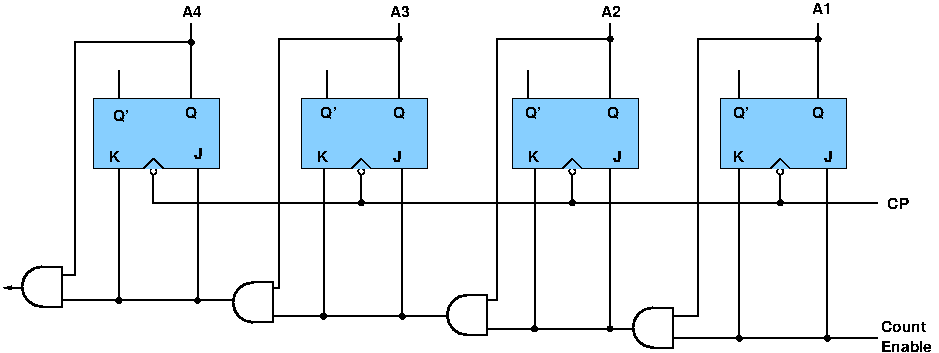
In parallel-in parallel out register the data is loaded in parallel and shifted out serially. The block diagram is-



Types of Counters:

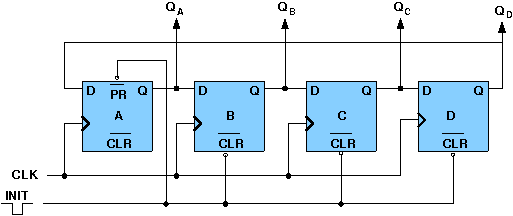
4-bit Synchronous Binary Counter

A counter is a sequential circuit that moves through a predefined sequence of states upon applying of clock pulses. The sequence of states may follow the binary number sequence or an arbitrary manner (no sequence). The simplest example of a counter is the binary counter which follows the binary number sequence. An n-bit binary counter contains n flip-flops and can count binary numbers from 0 to (2n -1)(up counter which is incremental, if it counts decrementally it is then down counter). logic diagram of 4 bit synchronous counter-



4-bit Synchronous Ring Counter

If the output of a shift register is fed back to the input. a ring counter results. The data pattern contained within the shift register will recirculate as long as clock pulses are applied. logic diagram of synchronous ring counter-



B) **Assembly language program to find GCD**

**Explanation:**

**Step I :** Initialize the data segment.

**Step II          :**Load AX and BX registers with the operands.

**Step III        :**     Check if the two numbers are equal. If yes goto step X, else goto step IV.

**Step IV         :**Is number 1 > number 2 ? If yes goto step VI else goto step V.

**Step V          :**     Exchange the contents of AX and BX register, such that AX contains the bigger number.

**Step VI         :**     Initialize DX register with 00H.

**Step VII       :**Perform the division operation (contents of AX / contents of BX).

**Step VIII     :**Check if there is remainder. If yes goto step IX, else goto step X.

**Step IX        :**     Move the remainder into AX register and goto step IV.

**Step X          :**     Save the contents of BX as GCD.

**Step XI        :**     Display the result.

**Step XII       :**Stop.

.model small

.stack 100

.data

no1 dw 0120

no2 dw 0090

gcd dw 0h

.code

mov ax,@data ; initialize DS

mov ds, ax

mov ax, no1 ; get the first number

mov bx, no2 ; get the second number

again: cmp ax, bx ; check if nos are equal

je endd ; if equal, save the GCD

jb exchg ; if no,

; is AX ; if yes interchange

l2: mov dx, 0

div bx ; check if ax is

; divisible by bx

cmp dx, 0 ;

je endd

mov ax, dx ; mov the remainder

; as no1 data

jmp again

exchg : xchg ax, bx jmp l2

endd : mov gcd, bx

mov ch, 04h ; Count of digits to be

; displayed

mov cl, 04h ; Count to roll by 4 bits

l12: rol bx, cl ; roll bl so that msb

; comes to lsb

mov dl, bl ; load dl with data

; to be displayed

and dl, 0fH ; get only lsb

cmp dl, 09 ; check if digit is 0-9

; or letter A-F

jbe l4

add dl, 07 ; if letter add 37H else

; only add 30H

l4: add dl, 30H

mov ah, 02 ; INT 21H

; (Display character)

int 21H

dec ch ; Decrement Count

jnz l12

mov ah, 4ch

int 21h

end

LAB-10:- A) Design of Direct Mapped cache:

Cache memory is a small (in size) and very fast (zero wait state) memory which sits between the CPU and main memory. The notion of cache memory actually rely on the correlation properties observed in sequences of address references generated by CPU while executing a programm(principle of locality).When a memory request is generated, the request is first presented to the cache memory, and if the cache cannot respond, the request is then presented to main memory.

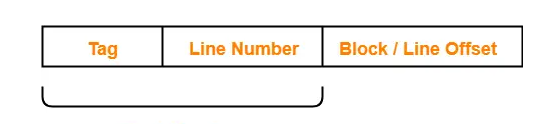
Hit: a cache access finds data resident in the cache memory

Miss: a cache access does not find data resident, so it forces to access the main memory.

**Direct Mapping-**

In direct mapping,

* A particular block of main memory can map to only one particular line of the cache.
* The line number of cache to which a particular block can map is given by-



B) Assembly language program to search an element using binary search

**Assumption –**   
Assume data to compare it with is stored in 3000H, list of numbers is from 3010H to 3019H and results are stored as follows: number of iterations in 3003H, success/failure (1/2) in 3001H and index in 3002H

**Algorithm –**

1. Move 0 to Accumulator and store it in 3003H, to indicate number of iterations so far.
2. Move 0 and 9 to L and H registers, respectively.
3. Load the data to search for in Accumulator from 3000H and shift it to B register.
4. Retrieve the number of iterations from 3003H, increase it by one and store back in 3003H.
5. Move value of H register to Accumulator and compare with L register.
6. If carry is generated, binary search is over so JUMP to step 20.
7. Add value of L register to Accumulator and right rotate it.
8. Store value of Accumulator in register C and force reset carry flag, if set.
9. Load the start address of the array in D-E register pair.
10. Add the value of accumulator to Register E and store the result in E.
11. Move 0 to Accumulator and use the ADC command to add any possible carry generated due to previous addition and store it back in Register D.
12. Load the value pointed to by D-E pair and compare with Register B. If carry is generated, JUMP to step 15 and if Zero flag is set, JUMP to step 17.
13. Move value of Register C to Accumulator and decrement Accumulator.
14. Move value of Accumulator to H and JUMP back to step 4.
15. Move value of Register C to Accumulator and increment Accumulator.
16. Move value of Accumulator to L and JUMP back to step 4.
17. Move 1 to Accumulator ad store in 3001H to indicate success.
18. Move value of Register C to Accumulator and store it in 3002H to save the index.
19. JUMP to statement 21.
20. Move 2 to Accumulator and store it in 3001H to indicate failure.
21. End the program.

**Program –**

| **Address** | **Label** | **Instruction** | **Comment** |
| --- | --- | --- | --- |
| 2000H |  | LDA 3000H | Load value to search for |
| 2003H |  | MOV B, A | Save it in register B |
| 2004H |  | MVI A, 0 |  |
| 2006H |  | STA 3003H | Store iteration number |
| 2009H |  | M0V L, A |  |
| 200AH |  | MVI A, 9 |  |
| 200CH |  | MOV H, A | Storing high and low indices in H-L pair done |
| 200DH | start\_loop: | LDA 3003H | Load iteration number |
| 2010H |  | INR A | Increment iteration number |
| 2011H |  | STA 3003H | Store back in 3003H |
| 2014H |  | MOV A, H | Store high index in Accumulator |
| 2015H |  | CMP L | Compare with lower index |
| 2016H |  | JC loop\_end | If carry generated, this means high is less than low so binary search over |
| 2019H |  | ADD L | Add high to low |
| 202AH |  | RAR | Right rotate to divide by two and generate mid |
| 202BH |  | MOV C, A | Save mid in register C |
| 202CH |  | JNC reset | If carry flag unset, go directly to reset. |
| 202FH |  | CMC | Force unset carry flag |
| 2030H | reset | NOP |  |
| 2031H |  | LXI D, 3010H | Load start address in D-E pair |
| 2034H |  | ADD E | Add mid to E to get the offset |
| 2035H |  | MOV E, A | Get the changed address back in E so it becomes a pointer to arr[mid] |
| 2036H |  | MVI A, 0 | Handle possible overflow |
| 2038H |  | ADC D |  |
| 2039H |  | MOV D, A | Memory index handled |
| 203AH |  | LDAX D | Load the array element in accumulator |
| 203BH |  | CMP B | Compare with value to search |
| 203CH |  | JC else\_block | Implies value is greater than value at mid, so we need low=mid+1 |
| 203FH |  | JZ print | If zero flag set, match found. Jump to print block |
| 2042H |  | MOV A, C | Neither executed so value<mid and we need high=mid-1 |
| 2043H |  | DCR A | mid=mid-1 |
| 2044H |  | MOV H, A | h=mid |
| 2045H |  | JMP start\_loop | Jump back |
| 2046H | else\_block | MOV A, C | We need low=mid+1 |
| 2047H |  | INR A | mid=mid+1 |
| 2048H |  | MOV A, L | l=mid |
| 2049H |  | JMP start\_loop |  |
| 204CH | print | MVI A, 1 | Move 1 to Accumulator |
| 204EH |  | STA 3001H | Store it in 3001H to indicate success |
| 2051H |  | MOV A, C | Move index, that is mid, back to Accumulator |
| 2052H |  | STA 3002H | Store it in 3002H |
| 2055H |  | JMP true\_end | Jump to end of the code |
| 2058H | loop\_end | MVI A, 2 |  |
| 205AH |  | STA 3001H | Store 2 in 3001H to indicate failure |
| 205DH | true\_end | HLT | Terminate |

**Explanation –**

1. We move value of higher and lower index (9 and 0 in this case) to H and L registers respectively in step 2
2. Higher and lower indices are compared in step 5. On getting a carry, which indicates low>high, we jump to end of loop else go to step 6.
3. In steps 7 and 8 we add value of H and L registers and right rotate it, which is equivalent to (high+low)/2 in order to find the index in say C language
4. In step 10, we add the value of mid to start address of array so that it acts as an offset, similar to how \*(arr+x) and arr[x] is identical in C.
5. Step 11 ensures no overflow occurs.
6. In step 12, we compare the value at mid index with the value to be searched. If it’s equal, we jump out of the loop and set the values appropriately.
7. If they are not equal, step 12 branches appropriately to let us increment/decrement mid by 1 and move that value to L/H register, as necessary (just like high=mid-1 or low=mid+1 is done in C) and go back to start of loop, that is step 2.

LAB-11

1. Associative cache Design

Design of Associative Cache:

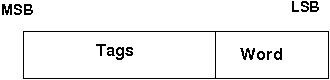
Cache memory is a small (in size) and very fast (zero wait state) memory which sits between the CPU and main memory. The notion of cache memory actually rely on the correlation properties observed in sequences of address references generated by CPU while executing a programm(principle of locality).When a memory request is generated, the request is first presented to the cache memory, and if the cache cannot respond, the request is then presented to main memory.

Hit: a cache access finds data resident in the cache memory

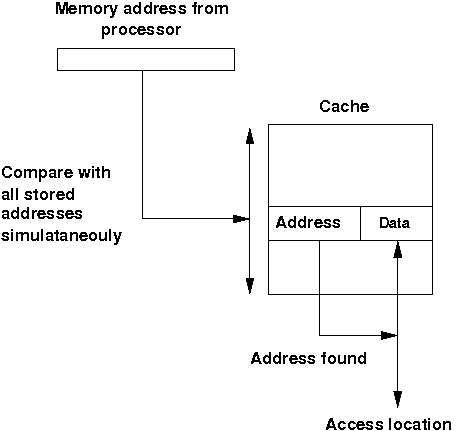
Miss: a cache access does not find data resident, so it forces to access the main memory.

Associative Cache

Any main memory block can mapped into any cache line. main memory address is divided into two groups which are tags and word bits. Words are low-order bits and identifies the location of a word within a block and tags are high-order bits which identifies the block.



Block diagram of a associated cache :



If a miss occur CPU bring the block from the main memory to the cache, if there is no free block in the corresponding set it replaces a block and put the new one. CPU uses different replacement policies to decide which block is to replace. The disadvantage of the associative cache is its high cost for implementing parallel tag comparison, but suffer the most from thrashing due to the 'conflict misses' giving more miss penalty.

1. **Assembly language program to sort numbers using bubble sort**

**Assumption –** Size of list is stored at 2040H and list of numbers from 2041H onwards.

**Algorithm –** 

1. Load size of list in C register and set D register to be 0
2. Decrement C as for n elements n-1 comparisons occur
3. Load the starting element of the list in Accumulator
4. Compare Accumulator and next element
5. If accumulator is less than or equal to the next element jump to step 8
6. Swap the two elements
7. Set D register to 1
8. Decrement C
9. If C>0 take next element in Accumulator and go to point 4
10. If D=0, this means in the iteration, no exchange takes place consequently we know that it won’t take place in further iterations so the loop in exited and program is stopped
11. Jump to step 1 for further iterations

**Program –**

| **Address** | **Label** | **Instruction** | **Comment** |
| --- | --- | --- | --- |
| 2000H | START | LXI H, 2040H | Load size of array |
| 2003H |  | MVI D, 00H | Clear D register to set up a flag |
| 2005H |  | MOV C, M | Set C register with number of elements in list |
| 2006H |  | DCR C | Decrement C |
| 2007H |  | INX H | Increment memory to access list |
| 2008H | CHECK | MOV A, M | Retrieve list element in Accumulator |
| 2009H |  | INX H | Increment memory to access next element |
| 200AH |  | CMP M | Compare Accumulator with next element |
| 200BH |  | JC NEXTBYTE | If accumulator is less then jump to NEXTBYTE |
| 200EH |  | JZ NEXTBYTE | If accumulator is equal then jump to NEXTBYTE |
| 2011H |  | MOV B, M | Swap the two elements |
| 2012H |  | MOV M, A |  |
| 2013H |  | DCX H |  |
| 2014H |  | MOV M, B |  |
| 2015H |  | INX H |  |
| 2016H |  | MVI D, 01H | If exchange occurs save 01 in D register |
| 2018H | NEXTBYTE | DCR C | Decrement C for next iteration |
| 2019H |  | JNZ CHECK | Jump to CHECK if C>0 |
| 201CH |  | MOV A, D | Transfer contents of D to Accumulator |
| 201DH |  | CPI 01H | Compare accumulator contents with 01H |
| 201FH |  | JZ START | Jump to START if D=01H |
| 2022H |  | HLT | HALT |

**Explanation-**

* Retrieve an element in accumulator.
* Compare it with next element, if it is greater then swap otherwise move to next index.
* If in one entire loop there has been no exchange, halt otherwise start the whole iteration again.
* The following approach has two loops, one nested inside other so-

Worst and Average Case Time Complexity: O(n\*n). Worst case occurs when array is reverse sorted.   
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***