Computer Organization & Architecture

Architecture: It refers to those attributes which are visible to the programmer & have a direct impact on logical execution of a program.

- 'what' does the system do.
- It defines the instructions of the system which are available for the user, i.e., defines the abstract manner of the System.

eg. types of instrⁿ

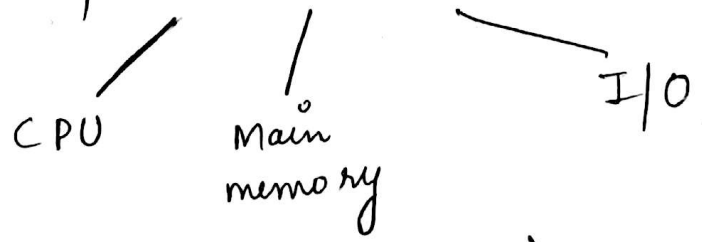
Organization: It refers to the operational units & the interconnections b/w them that achieve the architectural specifications.

- 'How' to implement
- transparent from the programmer
- Realization of abstract model

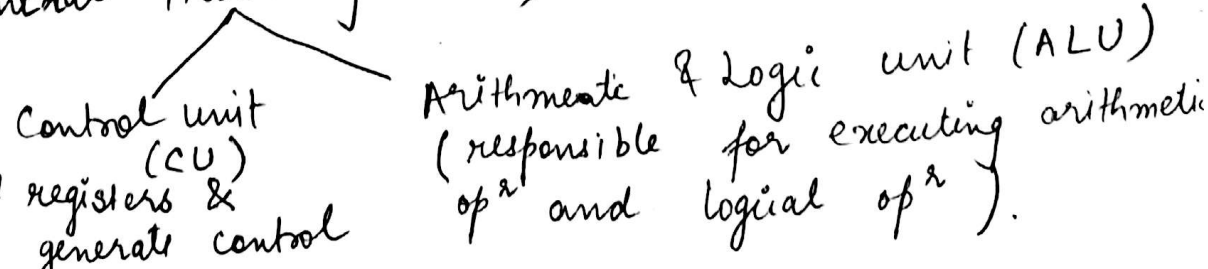
eg. Physical components (adders / subtractors).

(Real-Life eg: Visiting a house.)

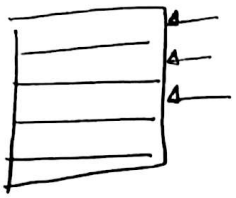
Basic components of a computer



① CPU (Central Processing Unit)



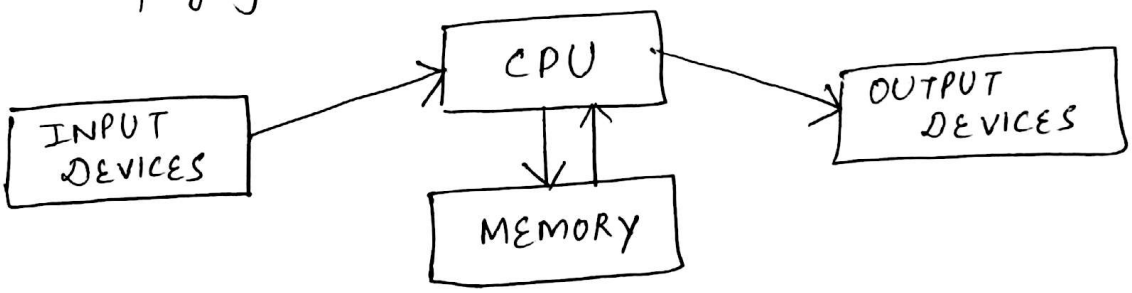
(2) Main Memory / Memory

- where inst° & data are stored temporarily.
-  Same memory for both data & inst°.

(3) I/O Components
- Input module
- Output module

a) I/P module : consists of some basic components that accepts data & instr^n in some form and convert them into an internal form of signals that is usable by the system.

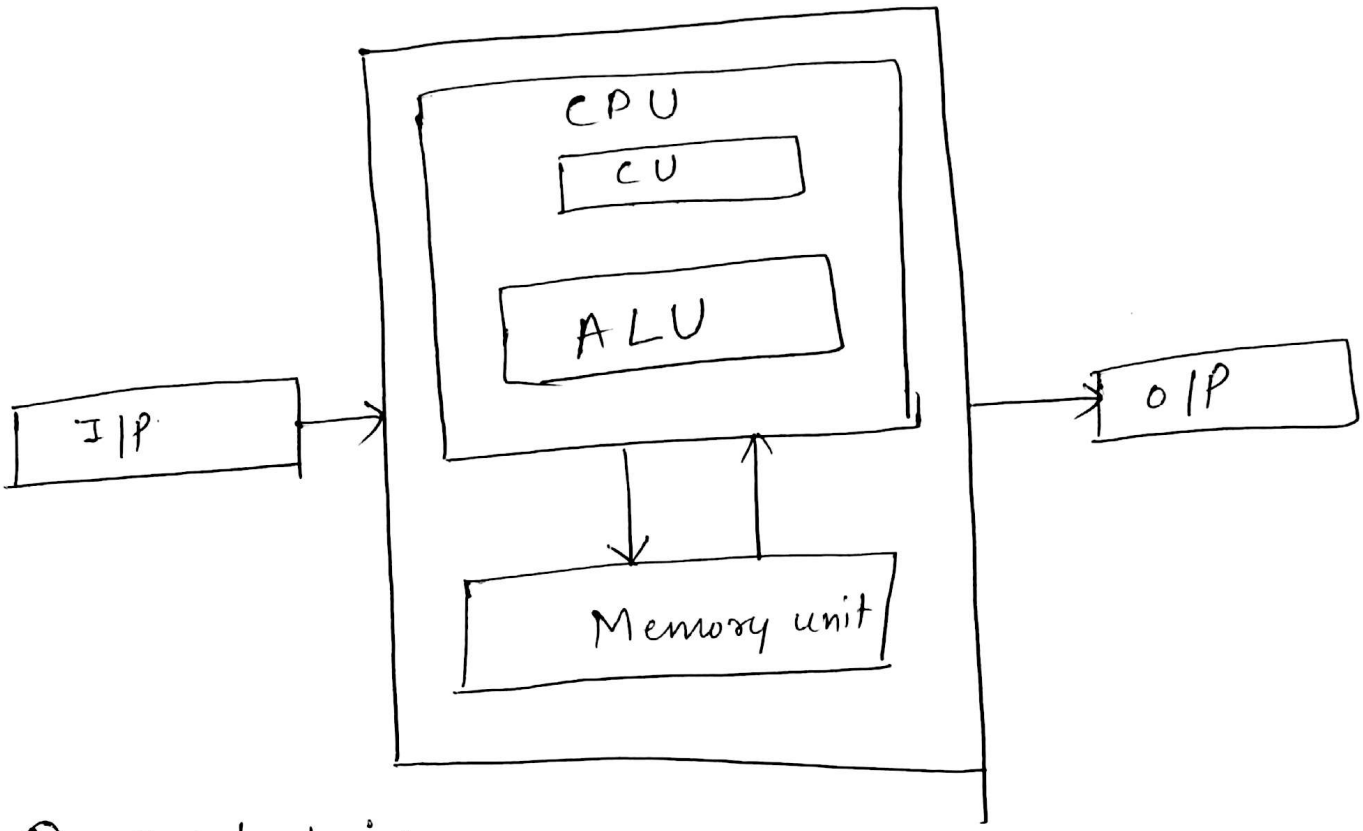
b) O/P module : It serves as a medium for displaying the results.



VON-NEUMANN ARCHITECTURE

- Basic fu^n is execution of program.
- It is based on stored-program computer concept, where inst^n data & program data are stored in the same memory.
- It consists of:
  - .) I/p unit
  - .) o/p unit
  - .) ALU
  - .) CU
  - .) Memory unit

Block Diagram



(1) Input devices
- mouse, joysticks, CD, floppy drive

(2) output devices
- displays result to the outside world
- monitor, CD drive.

(3) ALU
- +, −, *

(4) CU
- operations are co-ordinated & controlled by control unit.

(5) Memory unit
- each memory location has unique address.
- The speed of transmission / retrieval depends on the BUS.

Key - features of Von- neumann

1. It uses a stored program concept.
2. Program & data are stored in same memory
3. Each location of memory can be addressed independently.
4. Sequential fashion.

HARVARD ARCHITECTURE

(*) The limitation of von- neumann is that it has only one bus which is used for both data transfer & instr, wasting a lot of time. Also, no checks for interrupts.

1. Harvard architecture provides separate storage for data & instr.
2. It splits the memory into 2 parts, one for data & another for instr / programs.

Performance measures of CPU

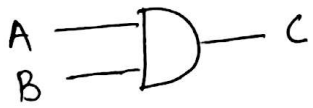
The performance of a computer is directly related to throughput & hence it is reciprocal of execution time.

$$\text{Performance} = \frac{1}{\text{execution time}}$$

Overview of Logic gates

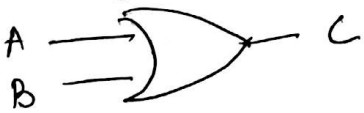
- logic gates are the basic building blocks of any digital system.

① AND Gate



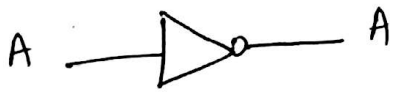
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

② OR gate



A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

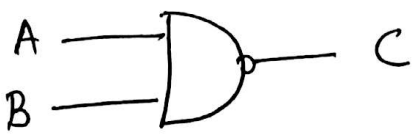
③ NOT gate



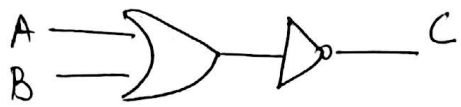
A	$\bar{A}$
0	1
1	0

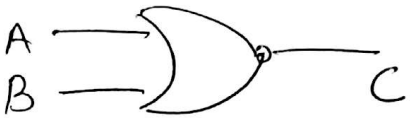
④ NAND gate

A	B	$C = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0



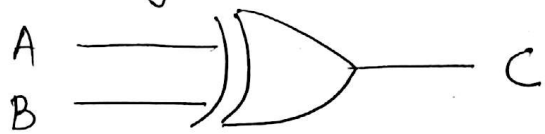
(5) NOR gate





A	B	$C = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

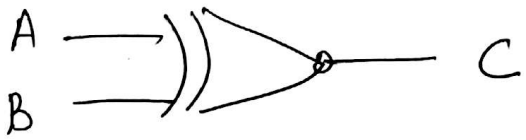
(6) XOR gate



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$C = A\bar{B} + \bar{A}B$

(7) XNOR gate



$C = \overline{A \oplus B}$

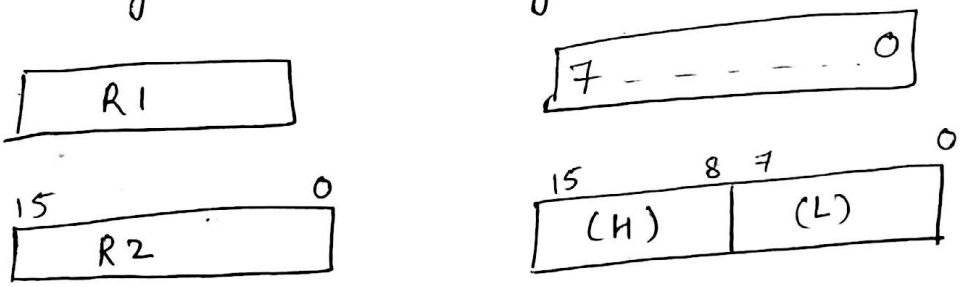
A	B	$A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

Chapter - Register Transfer Language

- Digital modules are best defined by the registers they contain & the operations that are performed.
- The op^n executed on data stored in registers are called microoperations.
- The symbolic notation used to specify the sequence of μop^n is called register transfer language.

Registers

- denoted by capital letters
- reg. that holds memory address is MAR (mem. add. reg.)
- Transfer is made by R2 ← R1

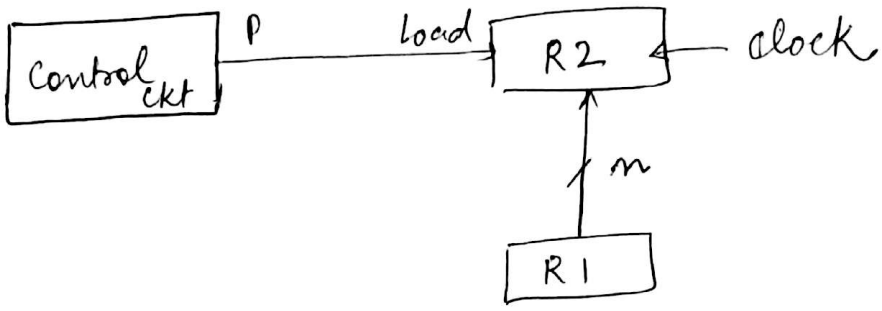


eg. if (P=1) then (R2 ← R1) ⇒ P: R2 ← R1

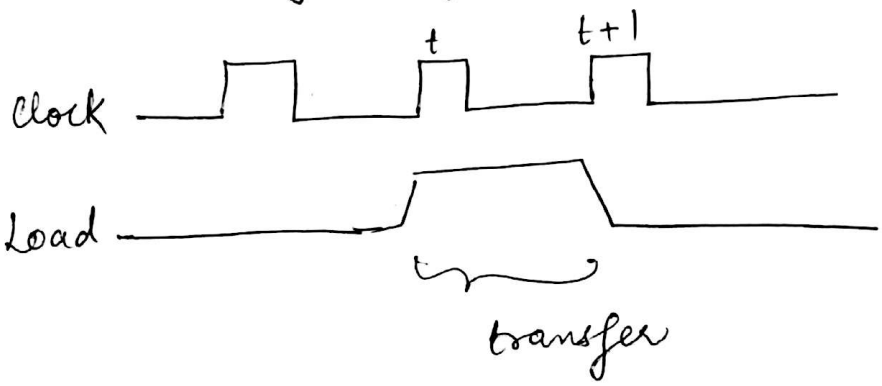
Basic Symbols

1) Letters → denotes a reg → MAR, PC, IR, R1
2) Paranthesis → " a part of reg → R2(0-7), R2(L)
3) Arrow ← = denotes transfer of info
4) Comma = separates two μop^n : R2 ← R1, R1 ← R2

Transfer from R1 to R2 when p = 1.



Timing diag:



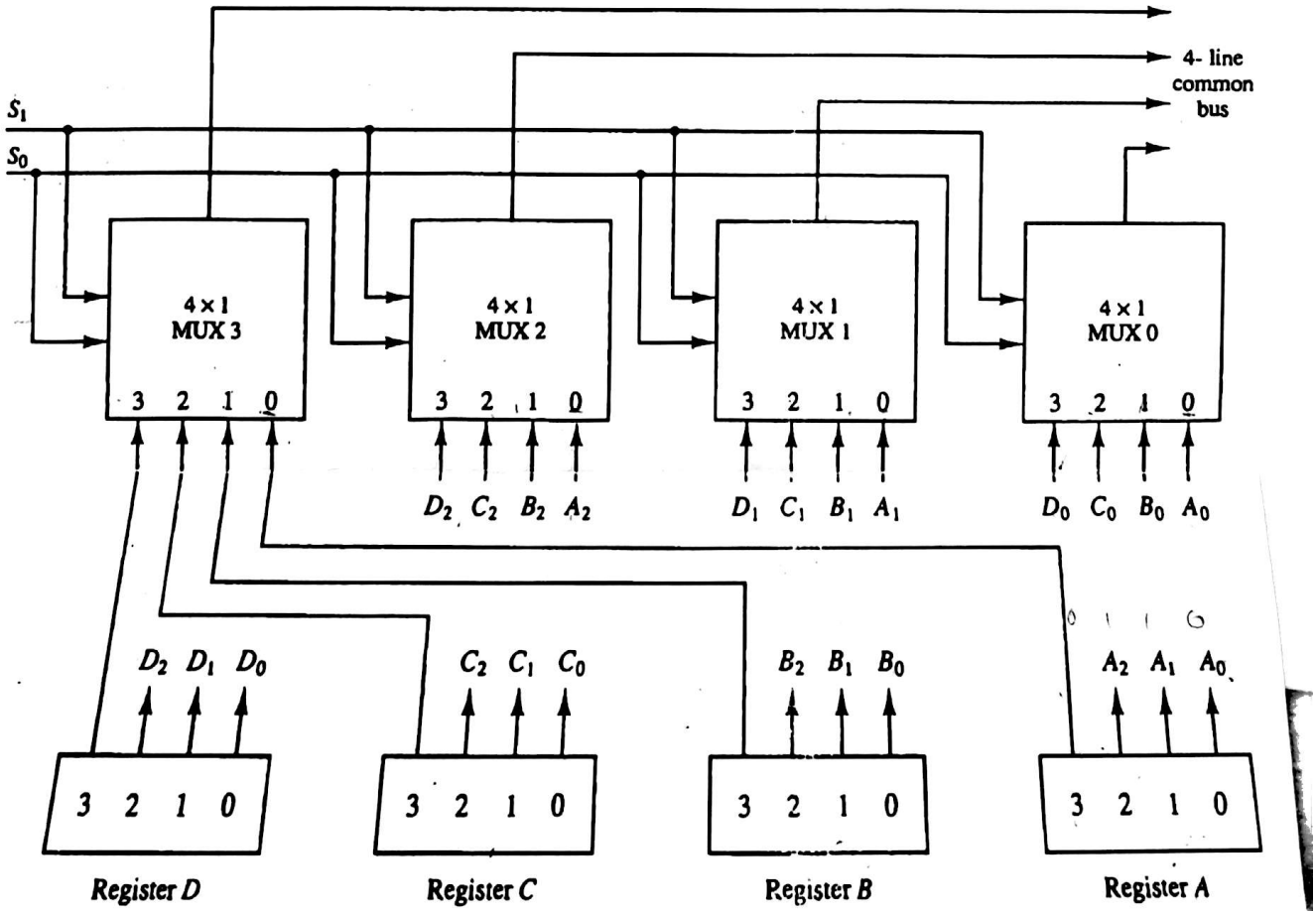
Bus and Memory Transfers

- A computer has many registers for storing data & instructions.
- The data need to be transferred from one reg to another register.
- One way, is to connect each reg with every other register, but the no. of lines / wires will be excessive.
  - Thus, we use a common bus system
  - Control signals determine which register is selected by the bus during each particular register transfer.

Common bus system using Multiplexers

- In this structure, a no. of registers are connected to the multiplexers.
- The register to be selected is decided by the select lines of mux.

Figure 4-3 Bus system for four registers.



Truth table:

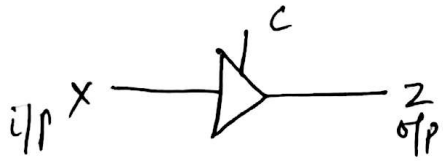
S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

Note: A bus system will multiplex 'k' registers of n bits each to produce an n-line common bus.

- The no. of mux = n
- Size of mux = $k \times 1$

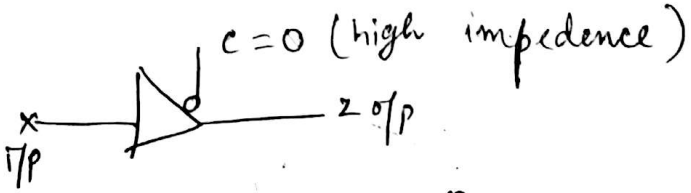
Using three-state gates

- The bus system can be constructed using three-state gates instead of multiplexers.
- It has three states
  - Logic 0
  - Logic 1
  - high-impedance state (behaves like an open-ckt)
- It is a useful device that allows us to control the system when current passes through a device & when it doesn't.



- i/p = o/p

i.e., when C = 1, behaves like a normal buffer.



when C = 0

o/p is Z, i.e, no electrical current flows.

Imp. Multiple devices are accessing the bus, hence we require some control for reading / writing. Thus we use a three-state buffer over a multiplexer to resolve this issue.

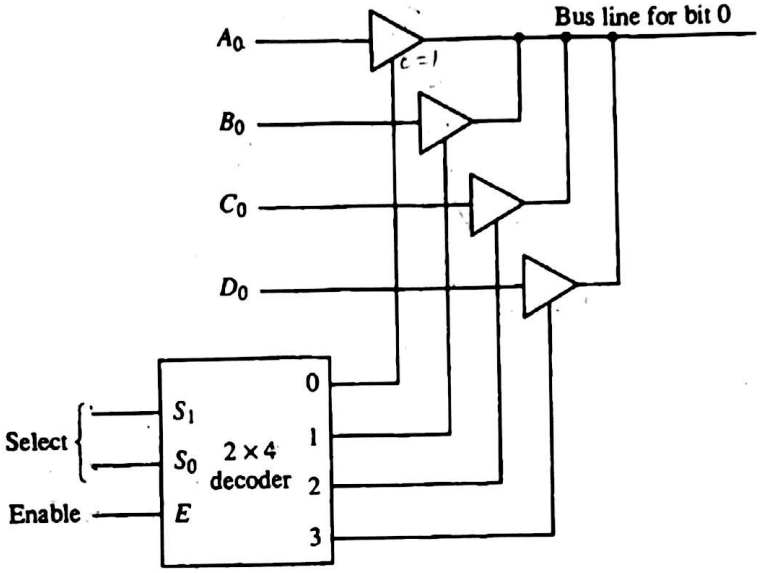


Figure 4-5 Bus line with three state-buffers.

E = 0

- We need to ensure that no more than one control input is active at any given time, thus we use decoder.
- When enabled, one of three-state buffers will be active, depending on $S_0 S_1$.

Memory Transfer

→ memory to user : Read
→ new info into memory : Write

eg. Read : $DR \leftarrow M[AR]$

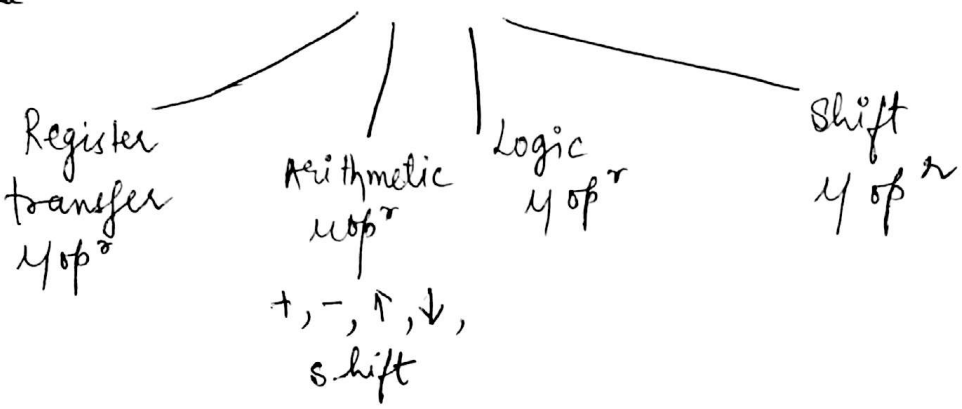
- Memory word is symbolized by M.
- Specify the address of memory using [ ]
- DR = data reg
  AR = address reg

eg. Write : $M[AR] \leftarrow R1$

- transfer of info from R1 into memory word M selected by the address in AR.

Arithmetic Microoperations

- A μop^{n} is an op^{n} performed with the data stored in registers.
- It is of 4 categories:

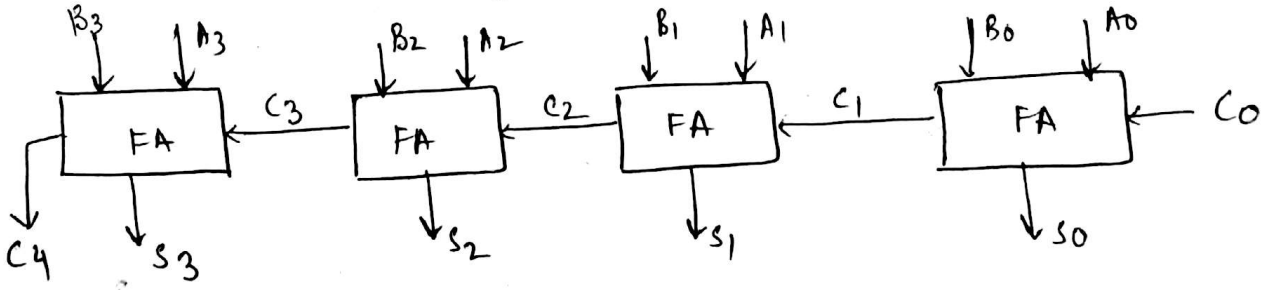


② eg. $R3 \leftarrow R1 + R2$ Specifies add microoperation.

$R3 \leftarrow R1 + \overline{R2} + 1$ or $R_1 - R_2$ (subtraction)

Binary adder

- implemented using a full adder
- The ckt that generates the arithmetic sum of 2 binary nos. is called a binary adder.



- 'n' bit binary adder requires 'n' full adders.

Binary Adder - Subtractor

- (+, −) can be combined in one common ckt by including an ex-or gate with the FA.

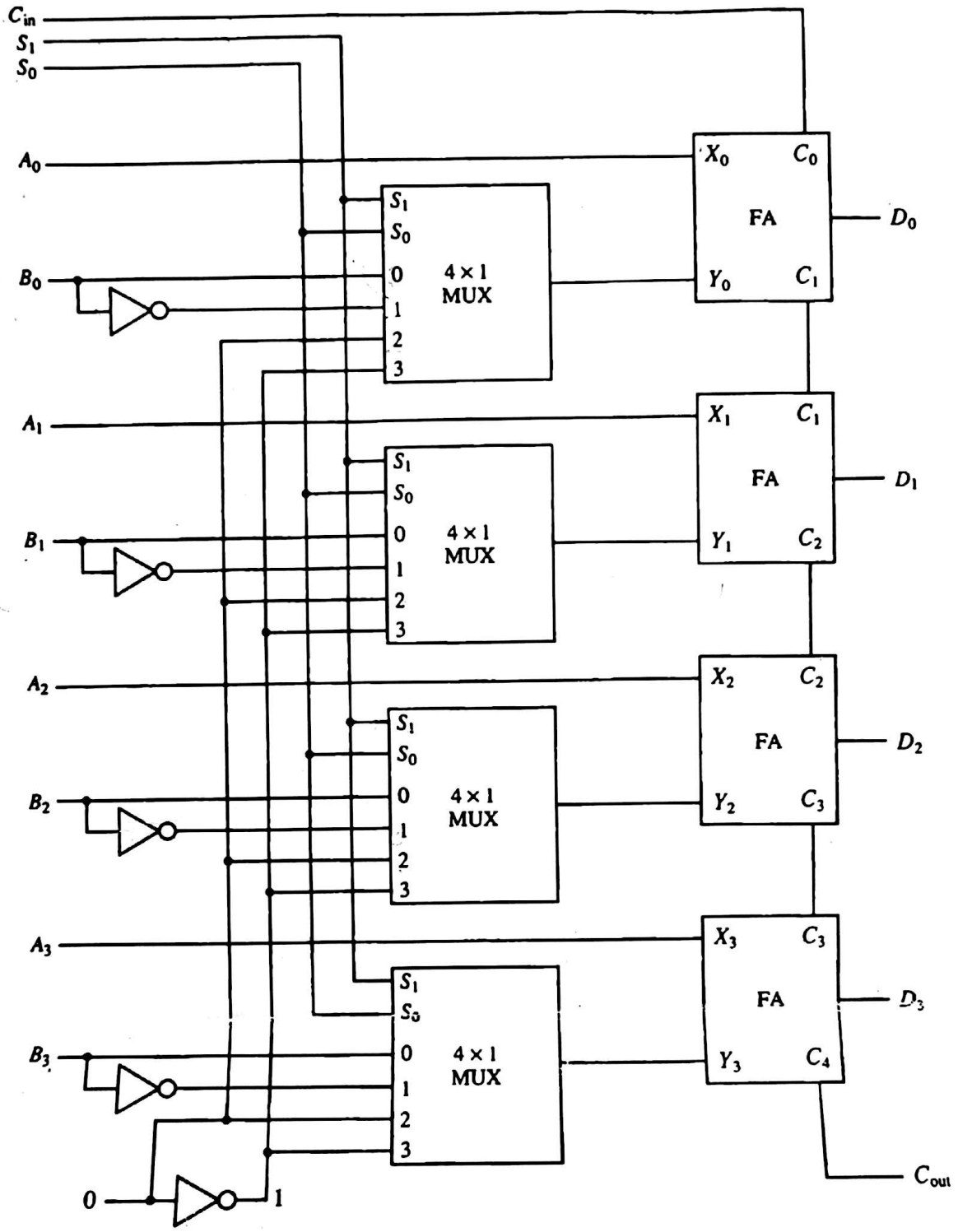


Figure 4-9 4-bit arithmetic circuit.

I/P :

O/P :

In mux, inputs are connected to B, B, 0 & 1.

O/p of binary adder is calculated using →

$$D = A + Y + C_{in}$$

(to the x input) (input carry)

(to the Y-inputs)

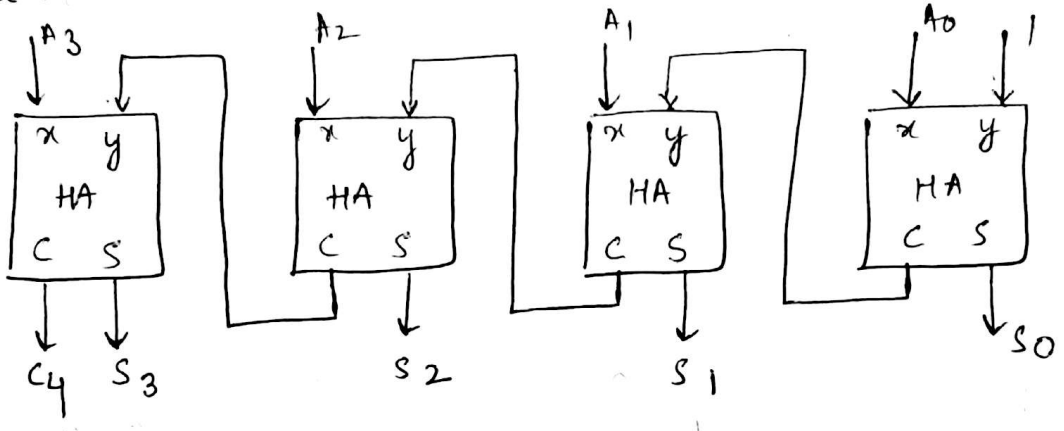
Binary Incrementer

eg. $R_1 = 0110$ (6)

$+ \quad 1$

0111 (7)

- implemented with binary counter with the help of half adders.
- at every clock pulse, the contents of reg is incremented by one.



Arithmetic circuit

- The basic component of an arithmetic ckt is the parallel adder.
- By controlling data i/p to adder, we can obtain diff. types of arithmetic op^n.
- Different options can be obtained using a multiplexer.

Case - 1 : $S_1 S_0 = 00$

$C_{in} = 0$; $D = A + B$ (Add)

$C_{in} = 1$; $D = A + B + 1$ (Add with carry)

Case - 2 : $S_1 S_0 = 01$

$C_{in} = 0$, $D = A + \bar{B}$ (Subtract with borrow)

$C_{in} = 1$; $D = A + \bar{B} + 1$ (Subtract)

Case - 3 : $S_1 S_0 = 10$

$C_{in} = 0$; $D = A$ (Transfer A)

$C_{in} = 1$; $D = A + 1$ (Increment A)

Case 4 : $S_1 S_0 = 11$

$C_{in} = 0$; $D = A - 1$ (Decrement A)

$C_{in} = 1$; $D = A$ (Transfer A)

∵ bcoz a no. with all ones is equal to the 2's complement of 1 (2's of 0001 = 1111).

Thus, A + 2's of 1 = A - 1

∴ when $C_{in} = 1$, $D = A - 1 + 1 = \underline{\underline{A}}$.

Logic Microoperations

- specify binary operations for string of bits stored in registers.

eg. P: R1 ← R1 ⊕ R2

Let R1 = 1010

R2 = 1100

Then R1, after P = 1 → 0110

Special Symbols

OR = ∨

AND = ∧

Compliment : $A = \bar{A}$

Applications

- Logic μop^s are useful for manipulating individual bits or a portion of word stored in a register.

① Selective - set (logic OR)

00	0
01	1
10	1
11	1

② selective complement

- complements bits in A where there are corresponding 1's in B

00	0
01	1
10	1
11	0

(Ex - OR)

```
1010
1100
----
0110
```

③ selective clear

- clears to 0 in A only where there are corresponding 1's in B.

```
A : 1010
B : 1100
-------
  0010
```

$(A = A\bar{B})$

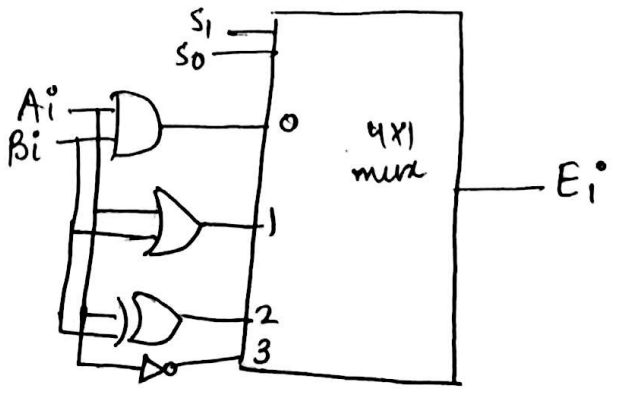
④ Mask

- clears A where there is 0's in B.

```
1010
1100
----
1000
```

(AND)

One stage of Logic ckt



SHIFT MICROOPERATIONS

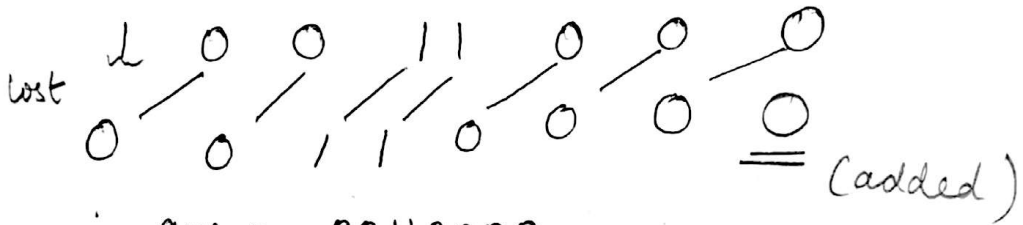
→ used for serial transfer of data.

→ Types — logical
circular
arithmetic

(1) * Logical shift (left) = shl

- transfers 0 through the serial input.

eg. 10011000



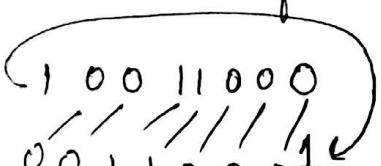
∴ ans = 00110000.

* logical Right shift (shr)

eg. 10011000

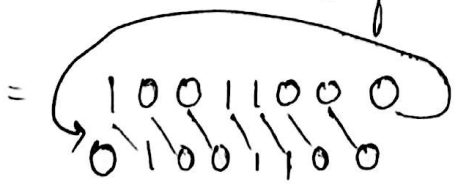
= 01001100 ← ans.

(2) Circular shift left (cil)



ans = 00110001

circular shift right (& cir)



ans = 01001100

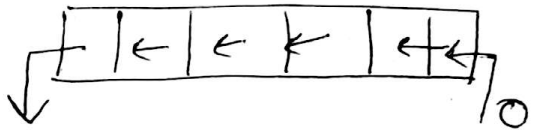
(3) Arithmetic shift

- it shifts a signed binary no. to left or right.
- in left, it multiplies the no. by 2
- " right, it divides the no. by 2
- The signed bit remains unchanged.

* left = ashl

```
1 1 0 1 1 0 0 1
 \ \ \ \ \ \ \
 1 0 1 1 0 0 1 [0]
```

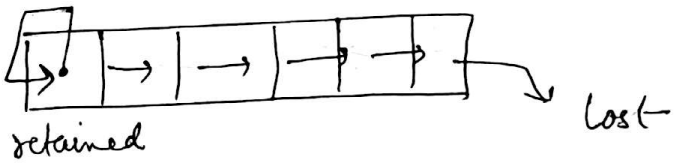
= 10110010



* right = ashr

```
  1 0 0 1 1 0 0 1
  ↓ \ \ \ \ \ \ \
= [1] 1 0 0 1 1 0 0
```

= 11001100

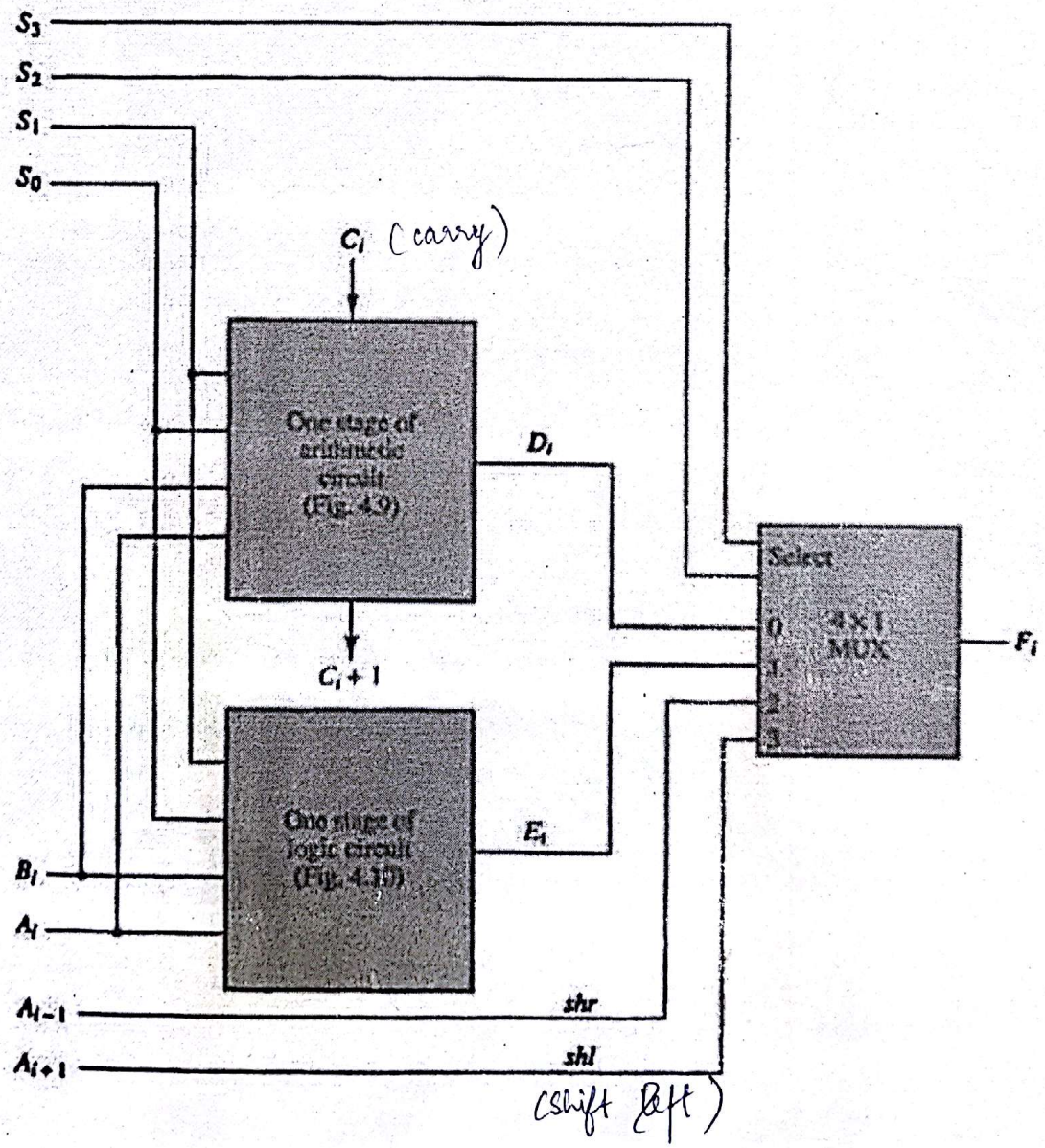


ALU

To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4-13. The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic

Figure 4-13 One stage of arithmetic logic shift unit.



units. A particular microoperation is selected with inputs S_1 and S_0. A 4×1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i. The data in the multiplexer are selected with inputs S_3 and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig. 4-13 must be repeated n times for an n-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in}, which provides a selection variable for the arithmetic operations.

The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

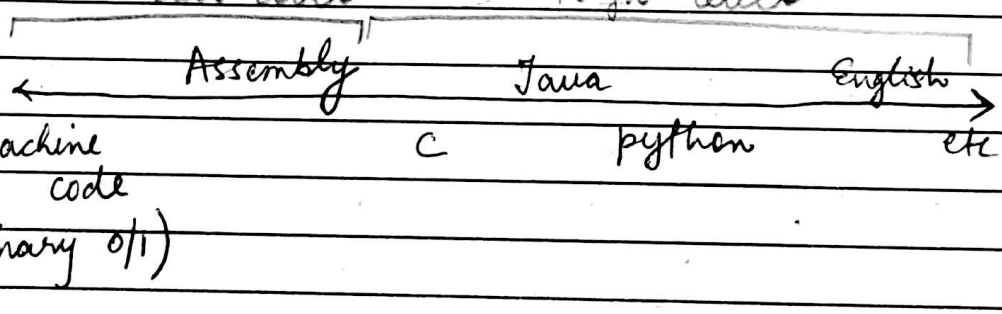
Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations (see Table 4-4) and are selected with $S_3S_2 = 00$. The next four are logic operations (see Fig. 4-10) and are selected with $S_3S_2 = 01$. The input carry has no effect during the logic operations and is marked with don't-care $\times$'s. The last two operations are shift operations and are selected with $S_3S_2 = 10$ and 11. The other three selection inputs have no effect on the shift.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	$\times$	$F = A \wedge B$	AND
0	1	0	1	$\times$	$F = A \vee B$	OR
0	1	1	0	$\times$	$F = A \oplus B$	XOR
0	1	1	1	$\times$	$F = \overline{A}$	Complement A
1	0	$\times$	$\times$	$\times$	$F = \text{shr } A$	Shift right A into F
1	1	$\times$	$\times$	$\times$	$F = \text{shl } A$	Shift left A into F

Levels of Programming Language.

Programming Lang - These are formal constructed languages designed to communicate inst^{r} to a machine.



(1) Low Level

- Machine code has the least abstraction; processors execute it directly.
- Assembly & high-level code must be translated to machine code.
- ADD → ADD
  LOAD → LDE
  Multipl^{n} → MUL
- Assembly lang are more abstracted.
- They use simple mnemonics

eg. ADD R2, 10 ⟶ converts into hex & executes.

(2) High Level Language

- most like ordinary languages, easier to use. eg. C++, python, Ruby
- They are portable, thus they can be executed on many computer, i.e., no h/w specific code (like low level) is required.
- More abstract lang. leave more work to be done at runtime, so are much slower to execute.

Compilation process:

HLL ⟶ executable form