Practical-1

Aim :- To study Verilog HDL to understand use of Hardware Description Language.

Hardware description language (HDL) is a specialized computer language used to program electronic and digital logic circuits. The structure, operation and design of the circuits are programmable using HDL. HDL includes a textual description consisting of operators, expressions, statements, inputs and outputs. Instead of generating a computer executable file, the HDL compilers provide a gate map. The gate map obtained is then downloaded to the programming device to check the operations of the desired circuit. The language helps to describe any digital circuit in the form of structural, behavioral and gate level and it is found to be an excellent programming language for FPGAs and CPLDs.

Three common HDL are−

- Verilog
- VHDL
- System-C

The three common HDLs are Verilog, VHDL, and SystemC. Of these, SystemC is the newest. The HDLs will allow fast design and better verification. In most of the industries, Verilog and VHDL are common. Verilog, one of the main Hardware Description Language standardized as IEEE 1364 is used for designing all types of circuits. It consists of modules and the language allows Behavioral, Dataflow and Structural Description. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is standardized by IEEE1164. The design is composed of entities consisting of multiple architectures. SystemC is a language that consist a set of C++classes and macros. It allows electronic system level and transaction modeling.

VERILOG

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog supports a design at many levels of abstraction. The major three are –

- Behavioural level Modelling
- Structural level Modelling
- Gate level Modelling

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Benefits of HDL

The major benefit of the language is fast design and better verification. The Top-down design and hierarchical design method allows the design time; design cost and design errors to be reduced. Another major advantage is related to complex designs, which can be managed and verified easily. HDL provides the timing information and allows the design to be described in gate level and register transfer level. Reusability of resources is one of the other advantage.

1) Behavioural level Modeling :

This level describes a system by concurrent algorithms (Behavioural). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and blocks are the main elements. There is no regard to the structural realization of the design.

Ex. AND gate

```
module andgate(a,b);

input [1:0] a;
output reg b;
always@(a)
begin

case(a)
```

```
2'b11:b=1'b1;
default:b=1'b0;
endcase

end

endmodule
```

2) Register−Transfer or Structural Level Modeling

Designs using the Register−Transfer Level specify the characteristics of a circuit using operations and the transfer of data between the registers. Modern definition of an RTL code is "Any code that is synthesizable is called RTL code".

Ex. AND gate

```
module andstructurel(a,b,c);
input a;
input b;
output c;

andgate a1(a,b,c);
endmodule
```

3) Gate Level Modelling

Within the logical level, the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (basic gates). Gate level modelling may not be a right idea for logic design. Gate level code is generated using tools like synthesis tools and his netlist is used for gate level simulation and for backend.

Ex. AND gate

```
module andgate(a,b,y);
input a,b;
output y;
assign y=(a&b);

endmodule
```

Practical-2

Aim :- Write the working of 8085 simulator and study basic architecture of 8085 along with 8085 instruction set.

8085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8-bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration −

- 8-bit data bus
- 16-bit address bus, which can address upto 64KB
- A 16-bit program counter
- A 16-bit stack pointer
- Six 8-bit registers arranged in pairs: BC, DE, HL
- Requires +5V supply to operate at 3.2 MHZ single phase clock

It is used in washing machines, microwave ovens, mobile phones, etc.

8085 Microprocessor – Functional Units :

8085 consists of the following functional units −

1) **Accumulator**

It is an 8-bit register used to perform arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU.

2) **Arithmetic and logic unit**

As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

3) **General purpose register**

There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H & L. Each register can hold 8-bit data.

4) **Program counter**

It is a 16-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

5) **Stack pointer**

It is also a 16-bit register works like stack, which is always incremented/decremented by 2 during push & pop operations.

6) **Temporary register**

It is an 8-bit register, which holds the temporary data of arithmetic and logical operations.

7) **Flag register**

It is an 8-bit register having five 1-bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.

8) **Instruction register and decoder**

It is an 8-bit register. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder decodes the information present in the Instruction register.

9) **Timing and control unit**

It provides timing and control signal to the microprocessor to perform operations. the timing and control signals, which control external and internal circuits.

10) **Interrupt control**

As the name suggests it controls the interrupts during a process. When a microprocessor is executing a main program and whenever an interrupt occurs, the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control goes back to the main program.

11) **Serial Input/output control**

It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).

12) **Address buffer and address-data buffer**

The content stored in the stack pointer and program counter is loaded into the address buffer and address-data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

13) **Address bus and data bus**

Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

8085 Architecture :

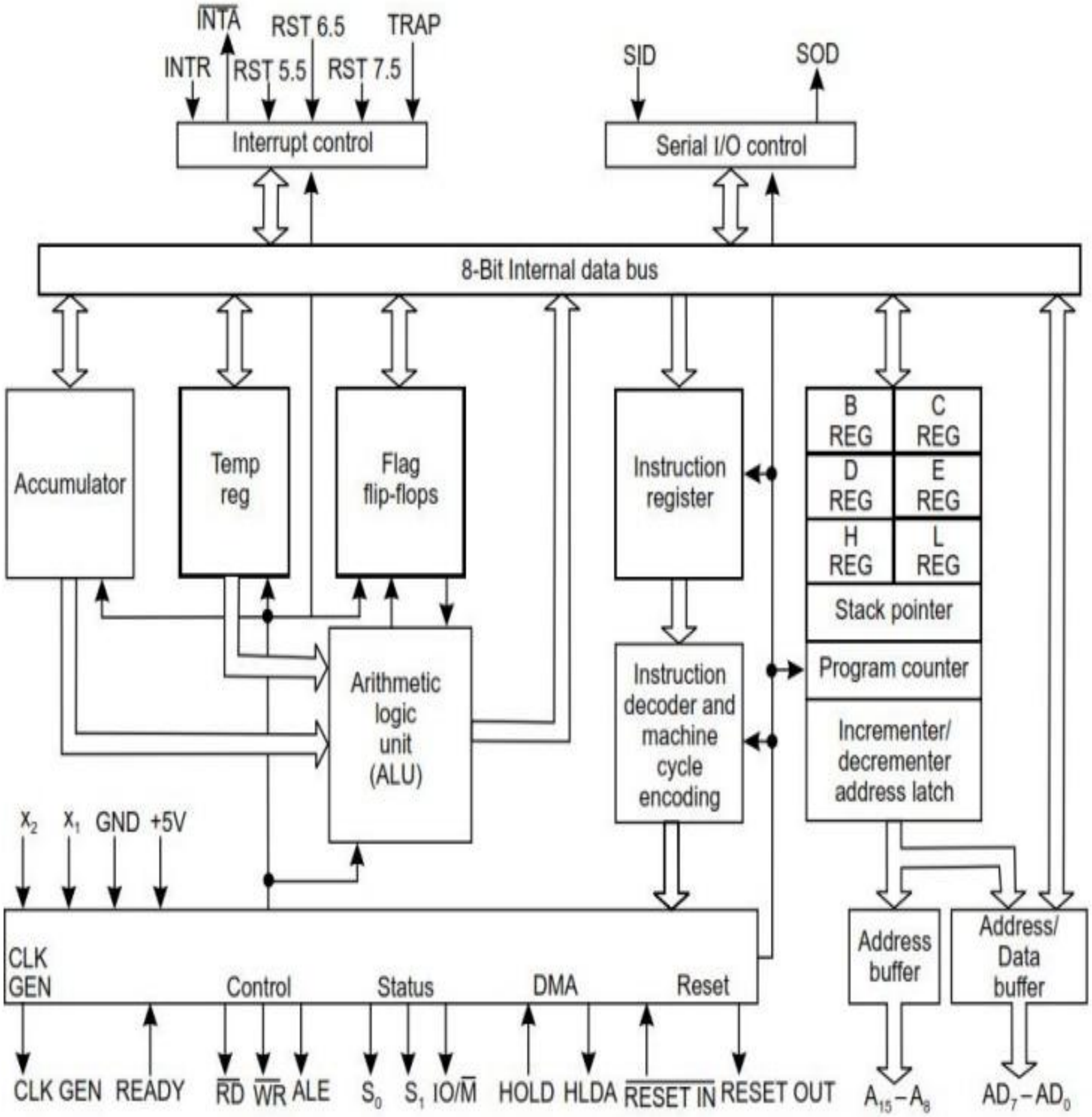
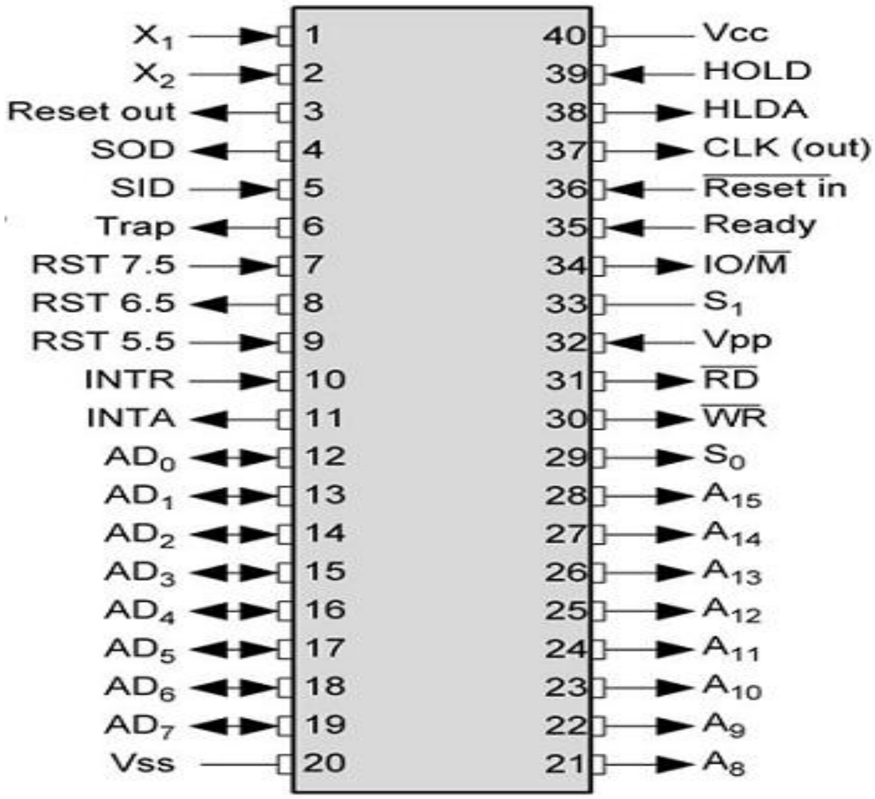


Fig 1.1 Hardware Architecture of 8085

8085 Pin Diagram:



The pins of a 8085 microprocessor can be classified into seven groups −

1) **Address bus**

A15-A8, it carries the most significant 8-bits of memory/IO address.

2) **Data bus**

AD7-AD0, it carries the least significant 8-bit address and data bus.

3) **Control and status signals**

These signals are used to identify the nature of operation. There are 3 control signal and 3 status signals.

Three control signals are RD, WR & ALE.

- RD − This signal indicates that the selected IO or memory device is to be read and is ready for accepting data available on the data bus.
- WR − This signal indicates that the data on the data bus is to be written into a selected memory or IO location.
- ALE − It is a positive going pulse generated when a new operation is started by the microprocessor. When the pulse goes high, it indicates address. When the pulse goes down it indicates data.

Three status signals are IO/M, S0 & S1.

- IO/M - This signal is used to differentiate between IO and Memory operations, i.e. when it is high indicates IO operation and when it is low then it indicates memory operation.
- S1 & S0 - These signals are used to identify the type of current operation.

4) Power supply

There are 2 power supply signals − VCC & VSS. VCC indicates +5v power supply and VSS indicates ground signal.

5) Clock signals

There are 3 clock signals, i.e. X1, X2, CLK OUT.

- X1, X2 − A crystal (RC, LC N/W) is connected at these two pins and is used to set frequency of the internal clock generator. This frequency is internally divided by 2.
- CLK OUT − This signal is used as the system clock for devices connected with the microprocessor.

6) Interrupts & externally initiated signals

Interrupts are the signals generated by external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. We will discuss interrupts in detail in interrupts section.

- INTA − It is an interrupt acknowledgment signal.
- RESET IN − This signal is used to reset the microprocessor by setting the program counter to zero.
- RESET OUT − This signal is used to reset all the connected devices when the microprocessor is reset.
- READY − This signal indicates that the device is ready to send or receive data. If READY is low, then the CPU has to wait for READY to go high.
- HOLD − This signal indicates that another master is requesting the use of the address and data buses.
- HLDA (HOLD Acknowledge) − It indicates that the CPU has received the HOLD request and it will relinquish the bus in the next clock cycle. HLDA is set to low after the HOLD signal is removed.

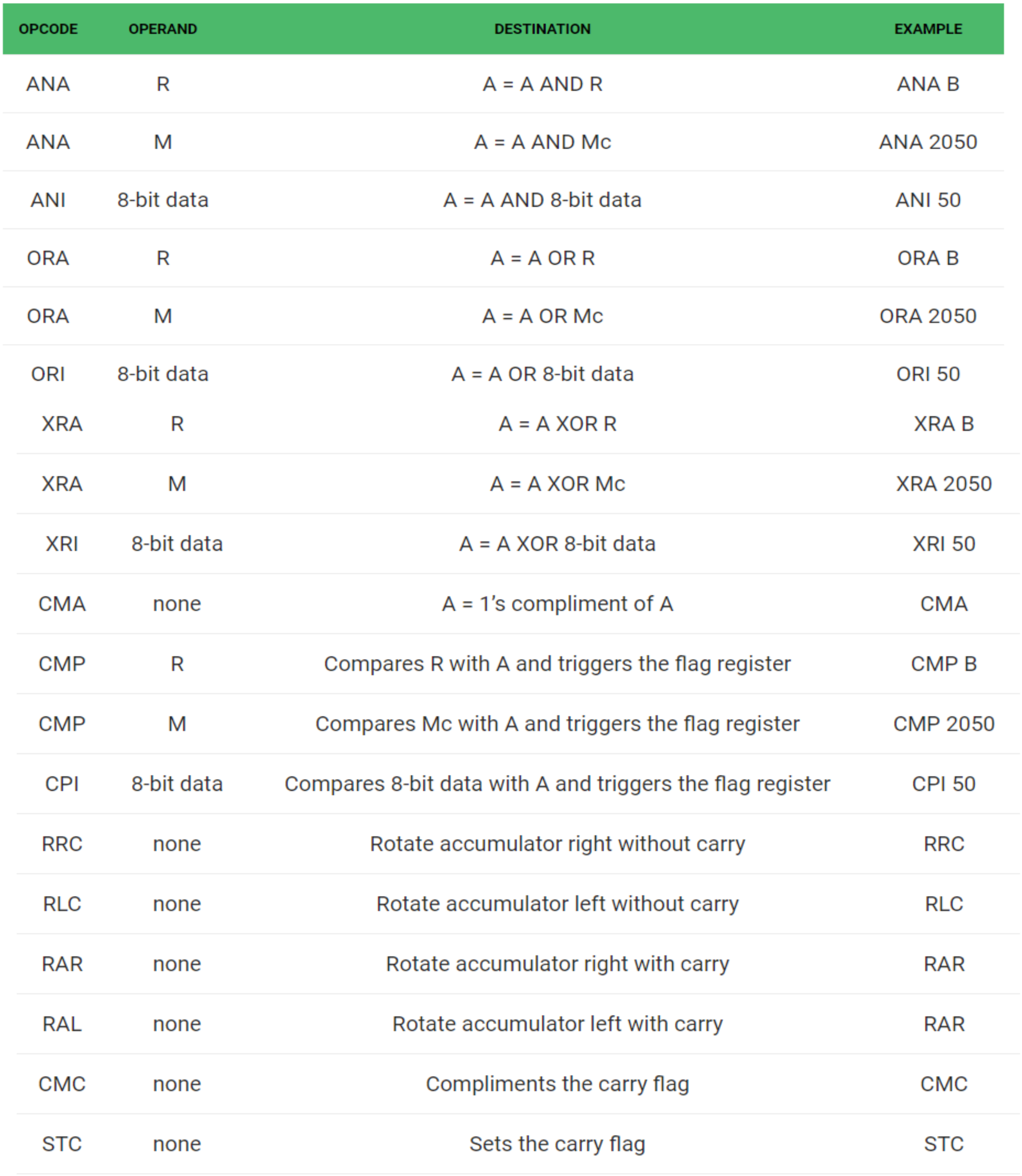
7) Serial I/O signals

There are 2 serial signals, i.e. SID and SOD and these signals are used for serial communication.

- SOD (Serial output data line) − The output SOD is set/reset as specified by the SIM instruction.
- SID (Serial input data line) − The data on this line is loaded into accumulator whenever a RIM instruction is executed.

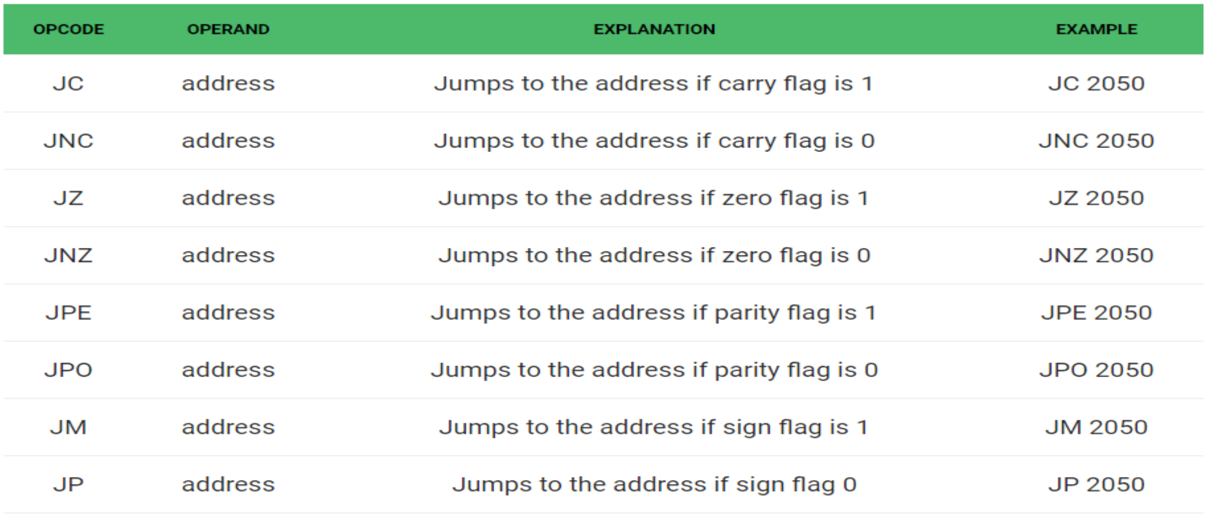
<u>8085 INSTRUCTIONS SET :</u>

1) <u>Logical Instructions :</u>

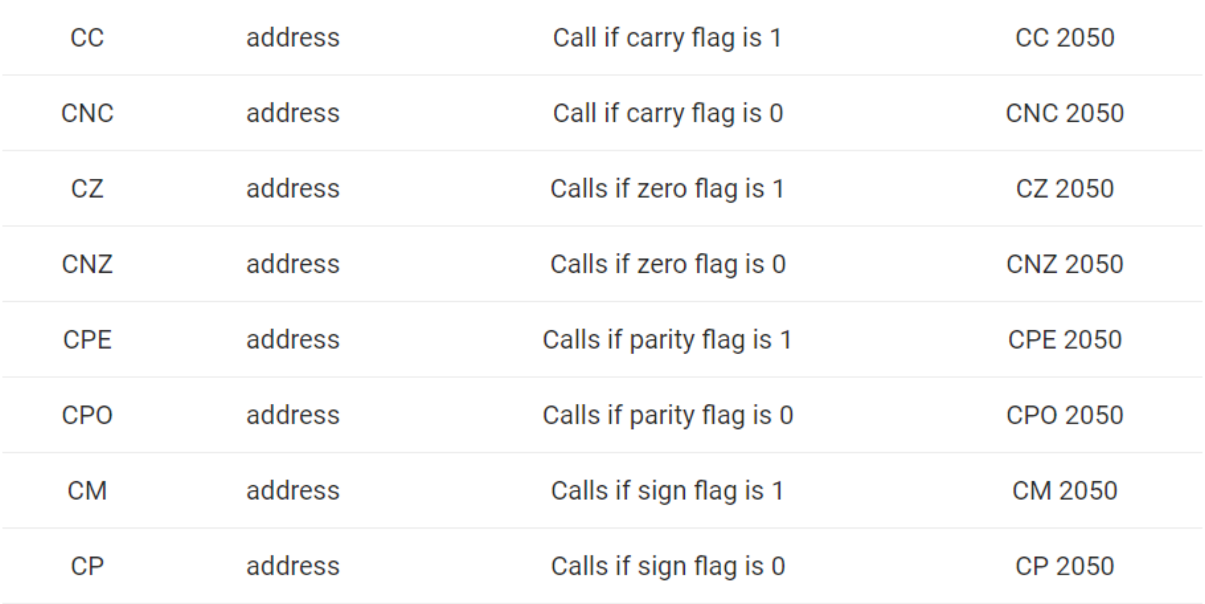


OPCODE	OPERAND	DESTINATION	EXAMPLE
ANA	R	A = A AND R	ANA B
ANA	M	A = A AND Mc	ANA 2050
ANI	8-bit data	A = A AND 8-bit data	ANI 50
ORA	R	A = A OR R	ORA B
ORA	M	A = A OR Mc	ORA 2050
ORI	8-bit data	A = A OR 8-bit data	ORI 50
XRA	R	A = A XOR R	XRA B
XRA	M	A = A XOR Mc	XRA 2050
XRI	8-bit data	A = A XOR 8-bit data	XRI 50
CMA	none	A = 1's compliment of A	CMA
CMP	R	Compares R with A and triggers the flag register	CMP B
CMP	M	Compares Mc with A and triggers the flag register	CMP 2050
CPI	8-bit data	Compares 8-bit data with A and triggers the flag register	CPI 50
RRC	none	Rotate accumulator right without carry	RRC
RLC	none	Rotate accumulator left without carry	RLC
RAR	none	Rotate accumulator right with carry	RAR
RAL	none	Rotate accumulator left with carry	RAR
CMC	none	Compliments the carry flag	CMC
STC	none	Sets the carry flag	STC

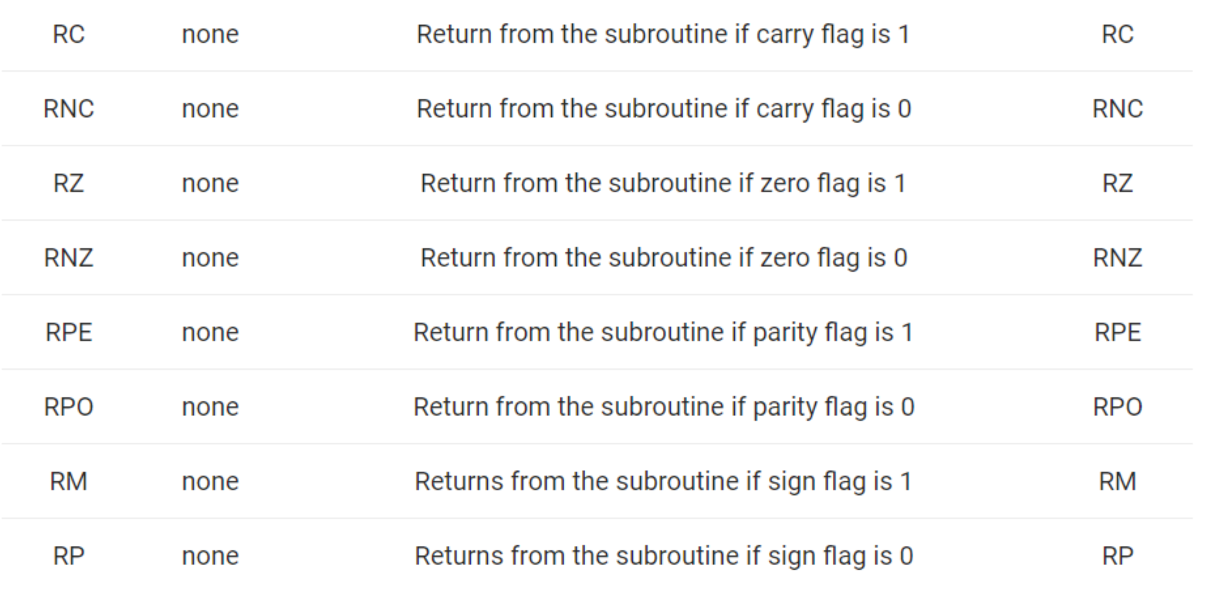
2) <u>Branching Instructions :</u>



OPCODE	OPERAND	EXPLANATION	EXAMPLE
JC	address	Jumps to the address if carry flag is 1	JC 2050
JNC	address	Jumps to the address if carry flag is 0	JNC 2050
JZ	address	Jumps to the address if zero flag is 1	JZ 2050
JNZ	address	Jumps to the address if zero flag is 0	JNZ 2050
JPE	address	Jumps to the address if parity flag is 1	JPE 2050
JPO	address	Jumps to the address if parity flag is 0	JPO 2050
JM	address	Jumps to the address if sign flag is 1	JM 2050
JP	address	Jumps to the address if sign flag 0	JP 2050

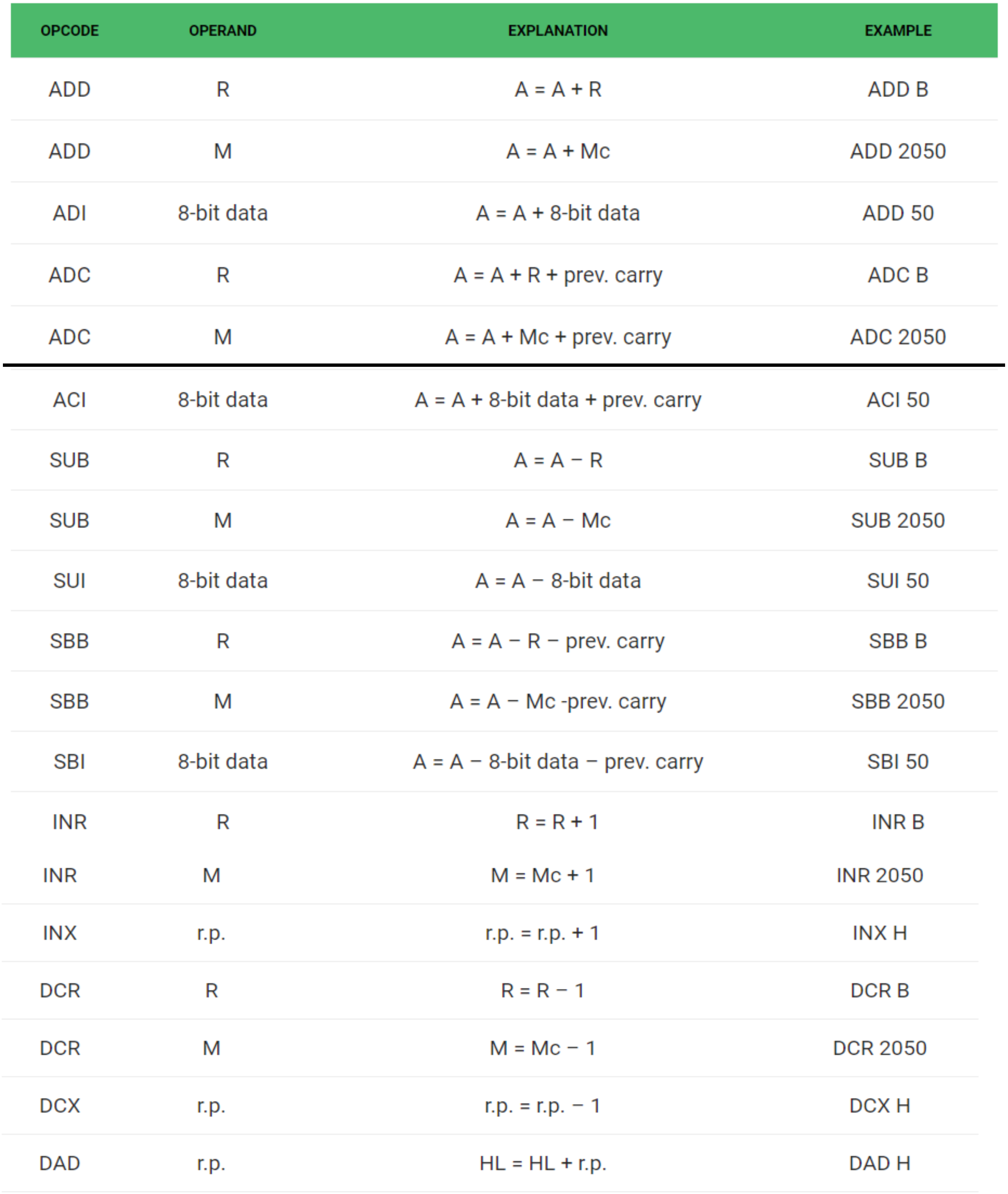


CC	address	Call if carry flag is 1	CC 2050
CNC	address	Call if carry flag is 0	CNC 2050
CZ	address	Calls if zero flag is 1	CZ 2050
CNZ	address	Calls if zero flag is 0	CNZ 2050
CPE	address	Calls if parity flag is 1	CPE 2050
CPO	address	Calls if parity flag is 0	CPO 2050
CM	address	Calls if sign flag is 1	CM 2050
CP	address	Calls if sign flag is 0	CP 2050



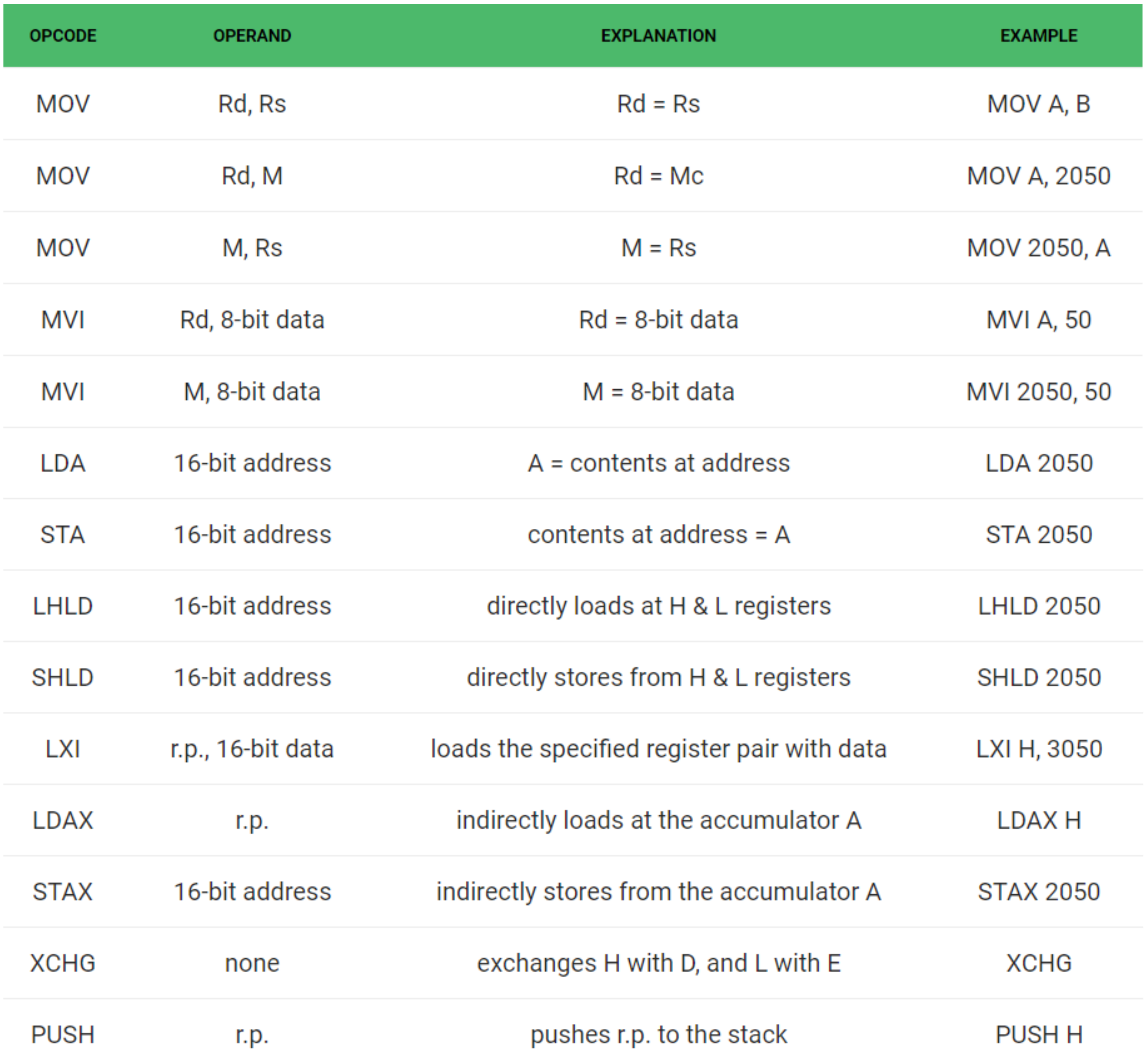
RC	none	Return from the subroutine if carry flag is 1	RC
RNC	none	Return from the subroutine if carry flag is 0	RNC
RZ	none	Return from the subroutine if zero flag is 1	RZ
RNZ	none	Return from the subroutine if zero flag is 0	RNZ
RPE	none	Return from the subroutine if parity flag is 1	RPE
RPO	none	Return from the subroutine if parity flag is 0	RPO
RM	none	Returns from the subroutine if sign flag is 1	RM
RP	none	Returns from the subroutine if sign flag is 0	RP

3) Arithmetic Instructions :

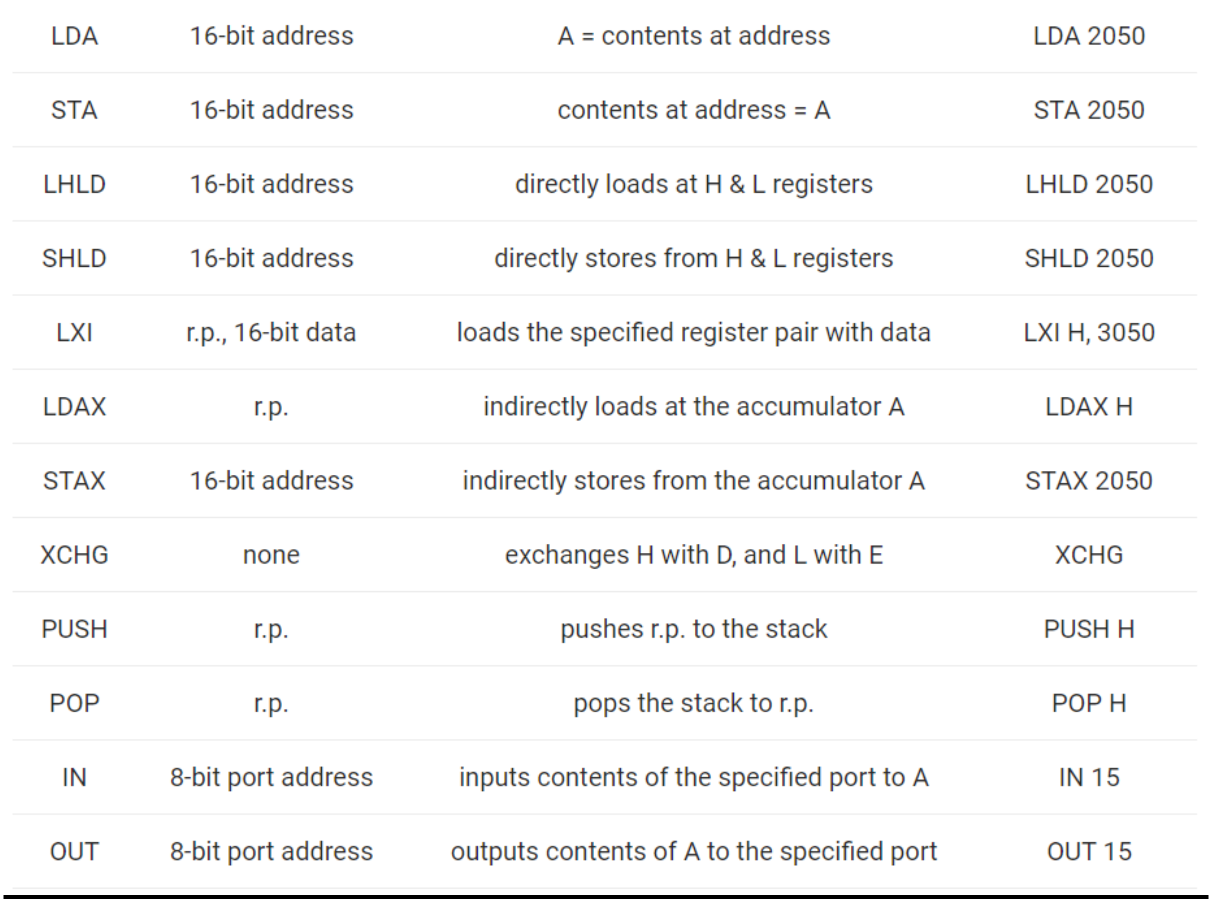


OPCODE	OPERAND	EXPLANATION	EXAMPLE
ADD	R	A = A + R	ADD B
ADD	M	A = A + Mc	ADD 2050
ADI	8-bit data	A = A + 8-bit data	ADD 50
ADC	R	A = A + R + prev. carry	ADC B
ADC	M	A = A + Mc + prev. carry	ADC 2050
ACI	8-bit data	A = A + 8-bit data + prev. carry	ACI 50
SUB	R	A = A – R	SUB B
SUB	M	A = A – Mc	SUB 2050
SUI	8-bit data	A = A – 8-bit data	SUI 50
SBB	R	A = A – R – prev. carry	SBB B
SBB	M	A = A – Mc -prev. carry	SBB 2050
SBI	8-bit data	A = A – 8-bit data – prev. carry	SBI 50
INR	R	R = R + 1	INR B
INR	M	M = Mc + 1	INR 2050
INX	r.p.	r.p. = r.p. + 1	INX H
DCR	R	R = R – 1	DCR B
DCR	M	M = Mc – 1	DCR 2050
DCX	r.p.	r.p. = r.p. – 1	DCX H
DAD	r.p.	HL = HL + r.p.	DAD H

4) Data Transfer Instructions :

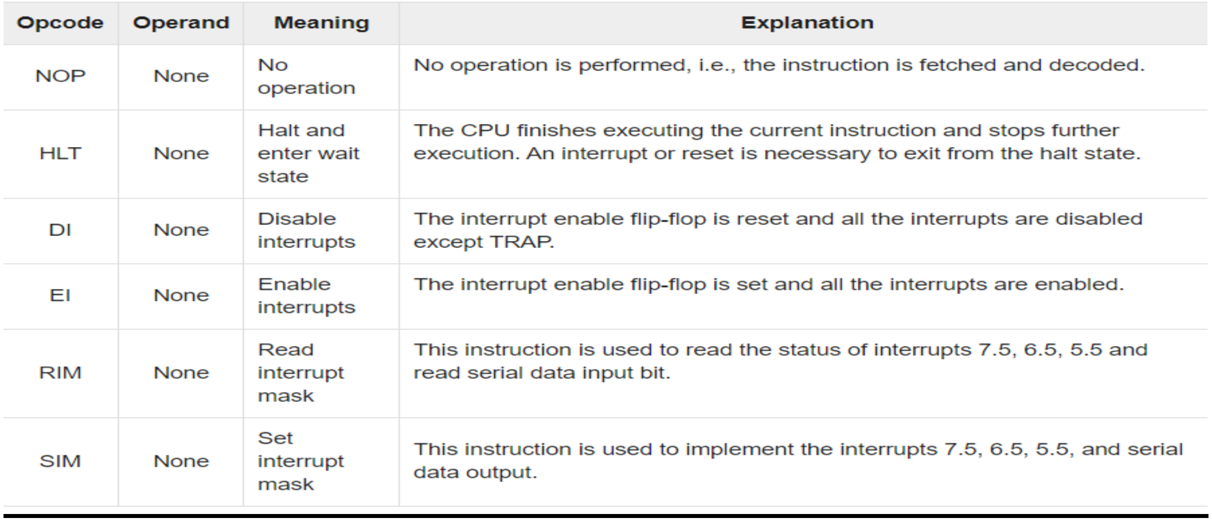


OPCODE	OPERAND	EXPLANATION	EXAMPLE
MOV	Rd, Rs	Rd = Rs	MOV A, B
MOV	Rd, M	Rd = Mc	MOV A, 2050
MOV	M, Rs	M = Rs	MOV 2050, A
MVI	Rd, 8-bit data	Rd = 8-bit data	MVI A, 50
MVI	M, 8-bit data	M = 8-bit data	MVI 2050, 50
LDA	16-bit address	A = contents at address	LDA 2050
STA	16-bit address	contents at address = A	STA 2050
LHLD	16-bit address	directly loads at H & L registers	LHLD 2050
SHLD	16-bit address	directly stores from H & L registers	SHLD 2050
LXI	r.p., 16-bit data	loads the specified register pair with data	LXI H, 3050
LDAX	r.p.	indirectly loads at the accumulator A	LDAX H
STAX	16-bit address	indirectly stores from the accumulator A	STAX 2050
XCHG	none	exchanges H with D, and L with E	XCHG
PUSH	r.p.	pushes r.p. to the stack	PUSH H



LDA	16-bit address	A = contents at address	LDA 2050
STA	16-bit address	contents at address = A	STA 2050
LHLD	16-bit address	directly loads at H & L registers	LHLD 2050
SHLD	16-bit address	directly stores from H & L registers	SHLD 2050
LXI	r.p., 16-bit data	loads the specified register pair with data	LXI H, 3050
LDAX	r.p.	indirectly loads at the accumulator A	LDAX H
STAX	16-bit address	indirectly stores from the accumulator A	STAX 2050
XCHG	none	exchanges H with D, and L with E	XCHG
PUSH	r.p.	pushes r.p. to the stack	PUSH H
POP	r.p.	pops the stack to r.p.	POP H
IN	8-bit port address	inputs contents of the specified port to A	IN 15
OUT	8-bit port address	outputs contents of A to the specified port	OUT 15

5) Control Instructions :



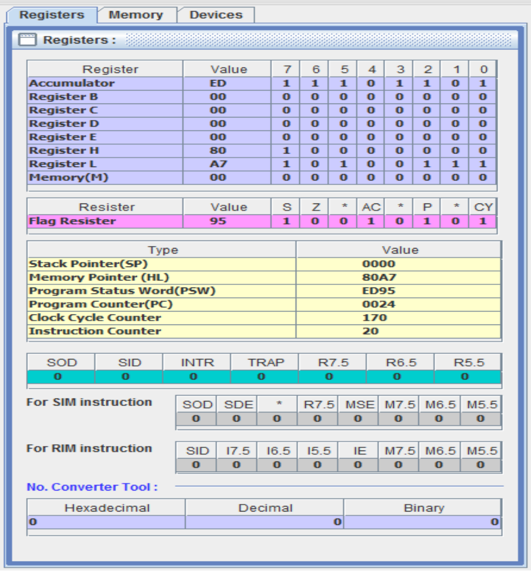
Opcode	Operand	Meaning	Explanation
NOP	None	No operation	No operation is performed, i.e., the instruction is fetched and decoded.
HLT	None	Halt and enter wait state	The CPU finishes executing the current instruction and stops further execution. An interrupt or reset is necessary to exit from the halt state.
DI	None	Disable interrupts	The interrupt enable flip-flop is reset and all the interrupts are disabled except TRAP.
EI	None	Enable interrupts	The interrupt enable flip-flop is set and all the interrupts are enabled.
RIM	None	Read interrupt mask	This instruction is used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit.
SIM	None	Set interrupt mask	This instruction is used to implement the interrupts 7.5, 6.5, 5.5, and serial data output.

PRACTICAL 3

Aim : Write an assembly language code in 8085 simulator to implement arithmetic, logical and shift instructions.

1) ARITHMETIC INSTRUCTION:

```
LXI H , C050
MOV A , M
INX H
ADD M
STA C052
INX H
ACI 10
STA C053
INX H
ADI 10
STA C054
DAD H
INX H
SUI 33
STA C055
INR A
STA C056
DCR A
STA C057
HLT
```



Register	Value	7	6	5	4	3	2	1	0
Accumulator	ED	1	1	1	0	1	1	0	1
Register B	00	0	0	0	0	0	0	0	0
Register C	00	0	0	0	0	0	0	0	0
Register D	00	0	0	0	0	0	0	0	0
Register E	00	0	0	0	0	0	0	0	0
Register H	80	1	0	0	0	0	0	0	0
Register L	A7	1	0	1	0	0	1	1	1
Memory(M)	00	0	0	0	0	0	0	0	0

Resister	Value	S	Z	*	AC	*	P	*	CY
Flag Resister	95	1	0	0	1	0	1	0	1

Type	Value
Stack Pointer(SP)	0000
Memory Pointer (HL)	80A7
Program Status Word(PSW)	ED95
Program Counter(PC)	0024
Clock Cycle Counter	170
Instruction Counter	20

SOD	SID	INTR	TRAP	R7.5	R6.5	R5.5
0	0	0	0	0	0	0

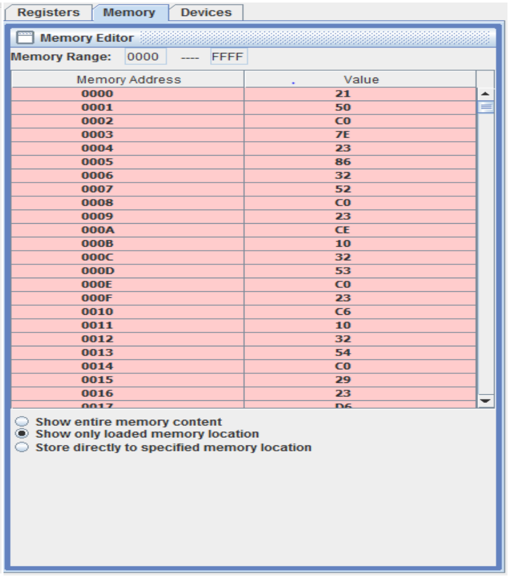
For SIM instruction

SOD	SDE	*	R7.5	MSE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

For RIM instruction

SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

Hexadecimal	Decimal	Binary
0	0	0



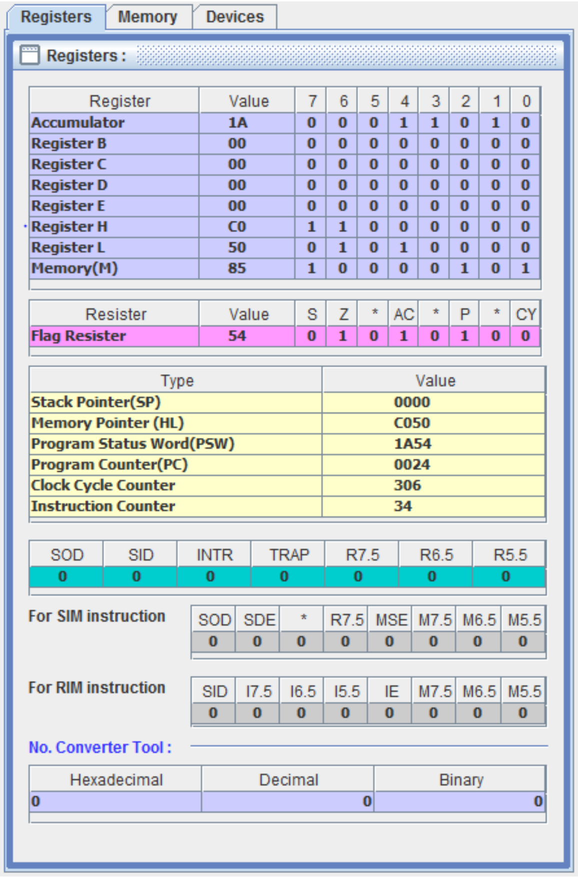
Memory Range: 0000 ---- FFFF

Memory Address	Value
0000	21
0001	50
0002	C0
0003	7E
0004	23
0005	86
0006	32
0007	52
0008	C0
0009	23
000A	CE
000B	10
000C	32
000D	53
000E	C0
000F	23
0010	C6
0011	10
0012	32
0013	54
0014	C0
0015	29
0016	23
0017	D6

○ Show entire memory content
● Show only loaded memory location
○ Store directly to specified memory location

2) LOGICAL INSTRUCTION:

```
    LXI H, C050
    MOV A, M
    CMA          // COMPLEMENT ACC
    STA C051
    CMC// COMPLEMENT CARRY
    STA C052
    STC  // SET CARRY TO 1
    STA C053
    ANI 86       // LOGICAL AND WITH ACC
    STA C054
    ORI 08       // LOGICAL OR WITH ACC
    STA C055
    XRI 10       // LOGICAL XOR WITH ACC
    STA C056
    CPI 1A       // COMPARE ACC WITH 7AH
    STA C057
    HLT
  # ORG C050
  # DB 85H
```



Registers :

Register	Value	7	6	5	4	3	2	1	0
Accumulator	1A	0	0	0	1	1	0	1	0
Register B	00	0	0	0	0	0	0	0	0
Register C	00	0	0	0	0	0	0	0	0
Register D	00	0	0	0	0	0	0	0	0
Register E	00	0	0	0	0	0	0	0	0
Register H	C0	1	1	0	0	0	0	0	0
Register L	50	0	1	0	1	0	0	0	0
Memory(M)	85	1	0	0	0	0	1	0	1

Resister	Value	S	Z	*	AC	*	P	*	CY
Flag Resister	54	0	1	0	1	0	1	0	0

Type	Value
Stack Pointer(SP)	0000
Memory Pointer (HL)	C050
Program Status Word(PSW)	1A54
Program Counter(PC)	0024
Clock Cycle Counter	306
Instruction Counter	34

SOD	SID	INTR	TRAP	R7.5	R6.5	R5.5
0	0	0	0	0	0	0

For SIM instruction

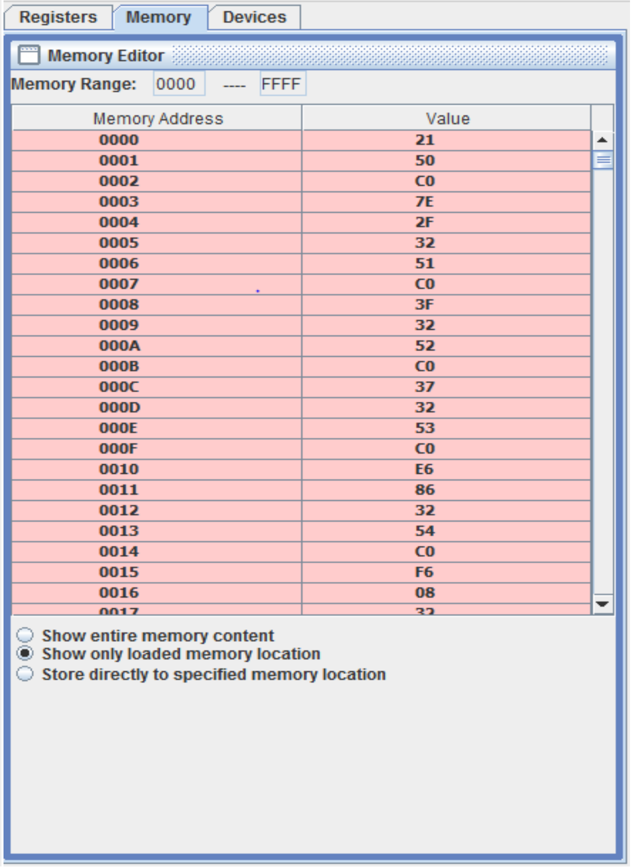
SOD	SDE	*	R7.5	MSE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

For RIM instruction

SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

No. Converter Tool :

Hexadecimal	Decimal	Binary
0	0	0



Memory Editor

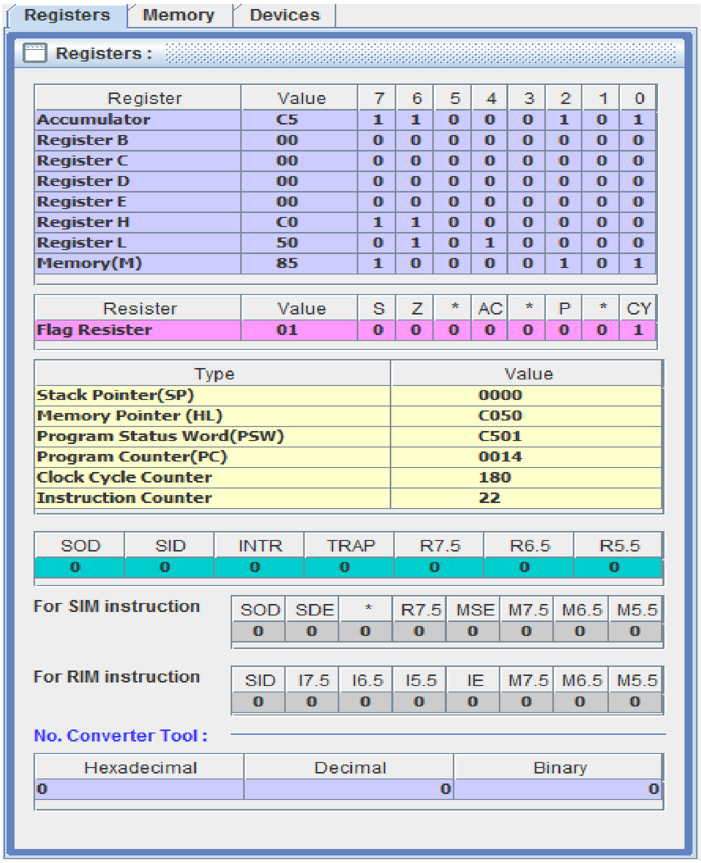
Memory Range: 0000 ---- FFFF

Memory Address	Value
0000	21
0001	50
0002	C0
0003	7E
0004	2F
0005	32
0006	51
0007	C0
0008	3F
0009	32
000A	52
000B	C0
000C	37
000D	32
000E	53
000F	C0
0010	E6
0011	86
0012	32
0013	54
0014	C0
0015	F6
0016	08
0017	32

○ Show entire memory content
● Show only loaded memory location
○ Store directly to specified memory location

3)SHIFT INSTRUCTIONS:

```
    LXI H,C050
    MOV A,M
    RLC // LEFT ROTATE ACC
    STA C051
    RAL // LEFT ROTATE ACC WITH CARRY
    STA C052
    RRC // RIGHT ROTATE ACC
    STA C053
    RAR // RIGHT ROTATE ACC WITH CARRY
    STA C054
    HLT
  # ORG C050
  # DB 85H
```



Registers :

Register	Value	7	6	5	4	3	2	1	0
Accumulator	C5	1	1	0	0	0	1	0	1
Register B	00	0	0	0	0	0	0	0	0
Register C	00	0	0	0	0	0	0	0	0
Register D	00	0	0	0	0	0	0	0	0
Register E	00	0	0	0	0	0	0	0	0
Register H	C0	1	1	0	0	0	0	0	0
Register L	50	0	1	0	1	0	0	0	0
Memory(M)	85	1	0	0	0	0	1	0	1

Resister	Value	S	Z	*	AC	*	P	*	CY
Flag Resister	01	0	0	0	0	0	0	0	1

Type	Value
Stack Pointer(SP)	0000
Memory Pointer (HL)	C050
Program Status Word(PSW)	C501
Program Counter(PC)	0014
Clock Cycle Counter	180
Instruction Counter	22

SOD	SID	INTR	TRAP	R7.5	R6.5	R5.5
0	0	0	0	0	0	0

For SIM instruction

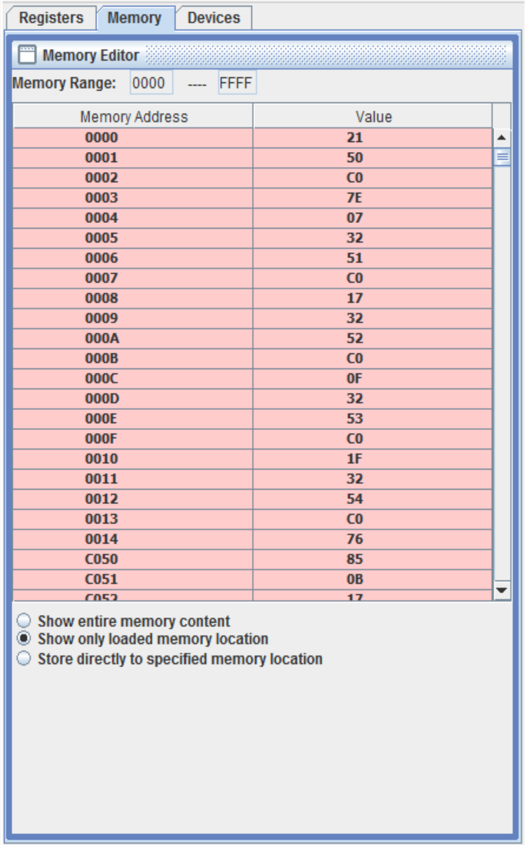
SOD	SDE	*	R7.5	MSE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

For RIM instruction

SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

No. Converter Tool :

Hexadecimal	Decimal	Binary
0	0	0



Memory Editor

Memory Range: 0000 ---- FFFF

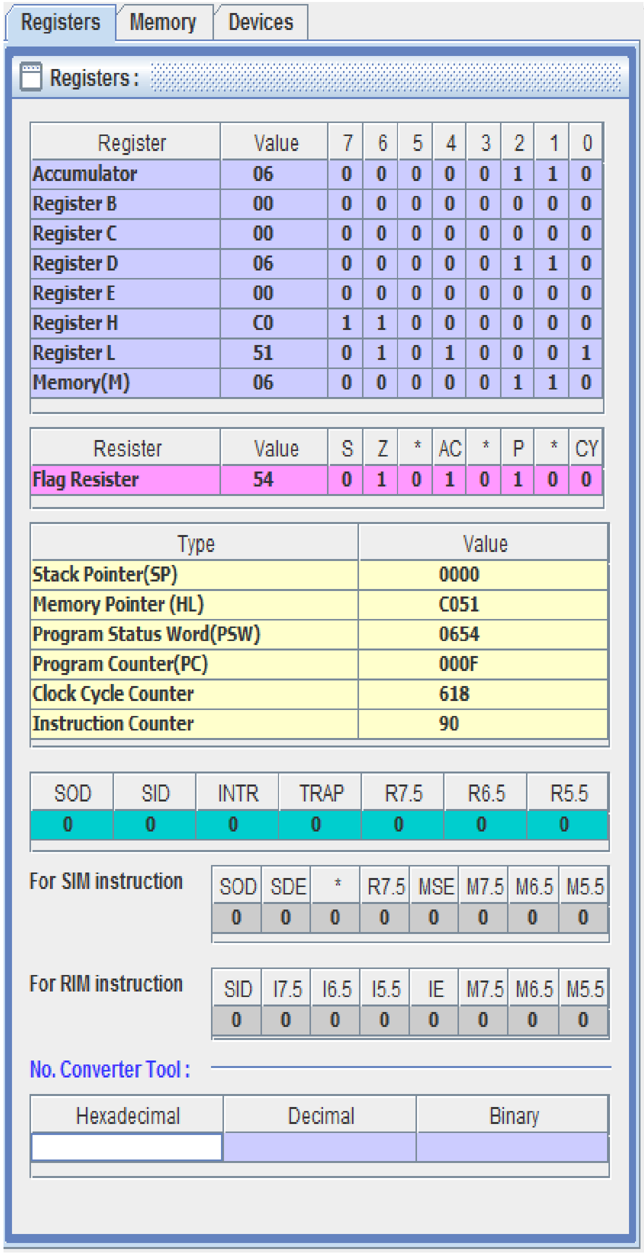
Memory Address	Value
0000	21
0001	50
0002	C0
0003	7E
0004	07
0005	32
0006	51
0007	C0
0008	17
0009	32
000A	52
000B	C0
000C	0F
000D	32
000E	53
000F	C0
0010	1F
0011	32
0012	54
0013	C0
0014	76
C050	85
C051	0B
C052	17

○ Show entire memory content
● Show only loaded memory location
○ Store directly to specified memory location

Practical-4

Aim :- Write an Assembly language code in 8085 simulator to find the Factorial of a number.

```
LXI H,C050
MOV B,M
MVI D,01H

FACT:           CALL MULTIPLY
                 DCR B
                 JNZ FACT
                 INX H
                 MOV M,D
                 HLT


MULTIPLY:        MOV E,B
                MVI A,00H


MULTIPLYLOOP:         ADD D
                      DCR E
                      JNZ MULTIPLYLOOP
                      MOV D,A
                      RET

# ORG C050
# DB 03H
```



Registers :

Register	Value	7	6	5	4	3	2	1	0
Accumulator	06	0	0	0	0	0	1	1	0
Register B	00	0	0	0	0	0	0	0	0
Register C	00	0	0	0	0	0	0	0	0
Register D	06	0	0	0	0	0	1	1	0
Register E	00	0	0	0	0	0	0	0	0
Register H	C0	1	1	0	0	0	0	0	0
Register L	51	0	1	0	1	0	0	0	1
Memory(M)	06	0	0	0	0	0	1	1	0

Resister	Value	S	Z	*	AC	*	P	*	CY
Flag Resister	54	0	1	0	1	0	1	0	0

Type	Value
Stack Pointer(SP)	0000
Memory Pointer (HL)	C051
Program Status Word(PSW)	0654
Program Counter(PC)	000F
Clock Cycle Counter	618
Instruction Counter	90

SOD	SID	INTR	TRAP	R7.5	R6.5	R5.5
0	0	0	0	0	0	0

For SIM instruction

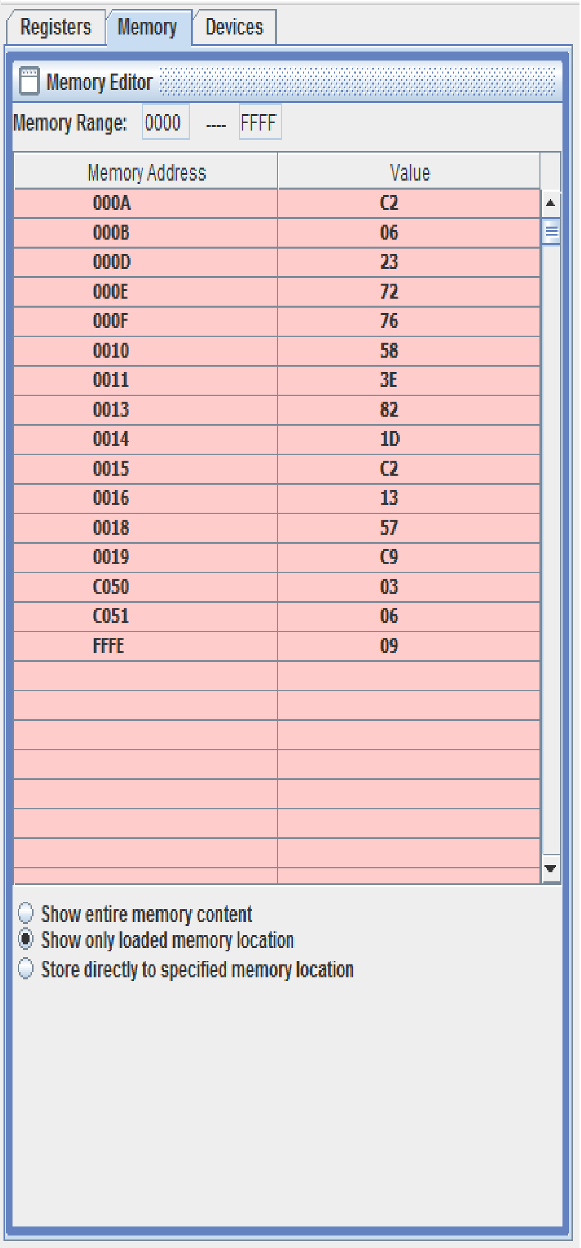
SOD	SDE	*	R7.5	MSE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

For RIM instruction

SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5
0	0	0	0	0	0	0	0

No. Converter Tool :

Hexadecimal	Decimal	Binary



Memory Editor

Memory Range: 0000 ---- FFFF

Memory Address	Value
000A	C2
000B	06
000D	23
000E	72
000F	76
0010	58
0011	3E
0013	82
0014	1D
0015	C2
0016	13
0018	57
0019	C9
C050	03
C051	06
FFFE	09

○ Show entire memory content
● Show only loaded memory location
○ Store directly to specified memory location

Practical-5

Aim: Write an assembly language code to arrange an array of data in ascending order and find the largest number in an array.

STEP 1: Storing Data into Memory

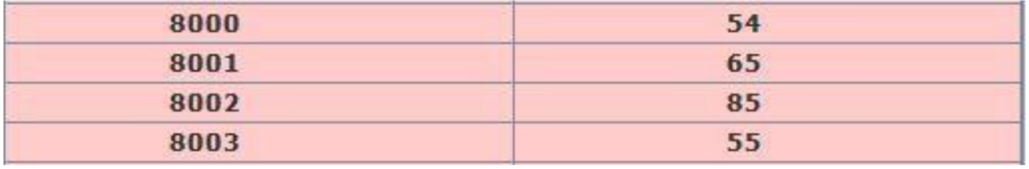
```
MVI A,54
MVI B,10
MVI C,20
MVI D,30
STA 8000 // 54 @ 8000
INR A
ADD B
STA 8001 // 65 @ 8001
ADD C
STA 8002 // 85 @ 8002
SUB D
STA 8003 // 55 @ 8003
```

STEP-2: Arranging an array of data in ascending order and finding the largest number in an array

```
        MVI B,03 // B=03
START:  LXI H,8000 //H=80, L=00
        MVI C,03 // C=03
BACK:   MOV A,M // Move data (located at memory address that HL refers) to
                   Accumulator).
        INX H
        CMP M // A-M
        JC SKIP // if carry is generated, jump to SKIP
        JZ SKIP // if z=0, jump to SKIP
        MOV D,M
        MOV M,A
        DCX H
        MOV M,D
        INX H
SKIP:   DCR C
        JNZ BACK // if z!=0, jump to BACK
        DCR B
        JNZ START // if z!=0 jump to START
        JZ MAX // if z=0 jump to MAX
MAX:    MVI C,03 // C=03
```

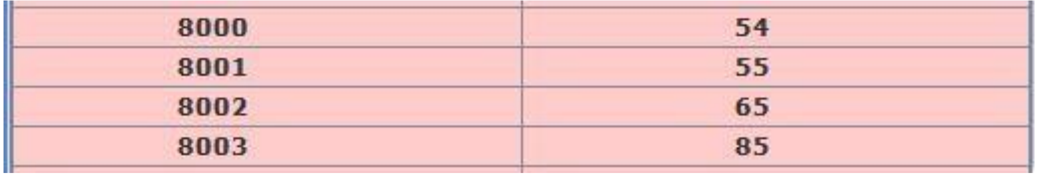
```
            LXI H,8000 // H=80,L=00
            MOV A,M

AGAIN: INX H
            CMP M //A-M
            JNC GO // if carry is not generated, jump to GO
            MOV A,M
GO:         DCR C
            JNZ AGAIN
            STA 8050 // Store Result at 8050 memory address from Accumulator
            HLT //Terminate task
```

Before Arranging an array :



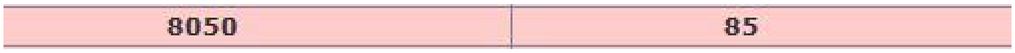
8000	54
8001	65
8002	85
8003	55

After Arranging an array:



8000	54
8001	55
8002	65
8003	85

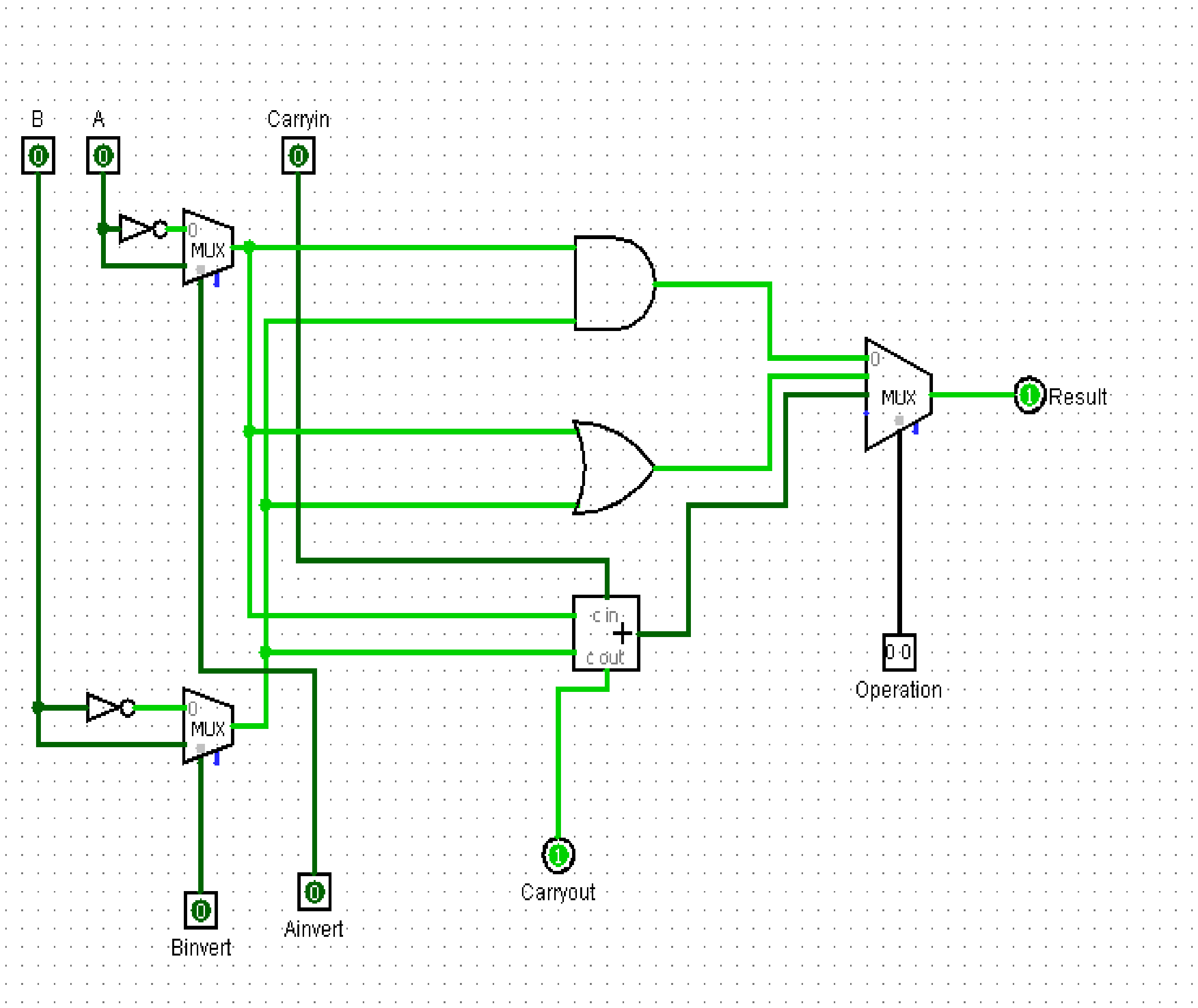
Max Number:



8050	85

Practical-6

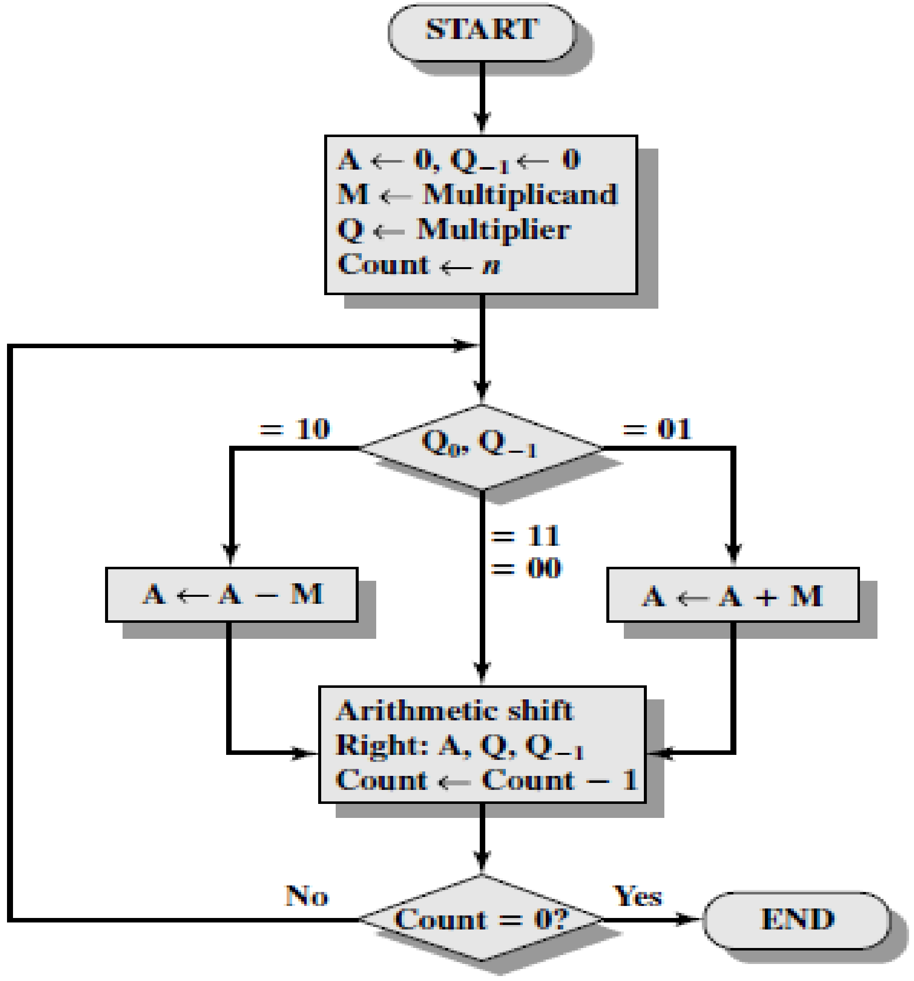
Aim: Design ALU using LOGISIM



Practical-7

Aim: Implement Booth’s Algorithm

FLOWCHART



ALGORITHM

STEP 1: Load A=0,Q_{-1}=0

M=Multiplicand

Q=Multiplier

Count=n

STEP 2: Check the status of Q_0Q_{-1}

if Q_0Q_{-1} =10 perform A ← A-M

if Q_0Q_{-1} =01 perform A ← A+M

STEP 3: Arithmetic shift right: A, Q, Q-1

STEP 4: Decrement count if not zero, repeat step 2 through 4
STEP 5: Stop

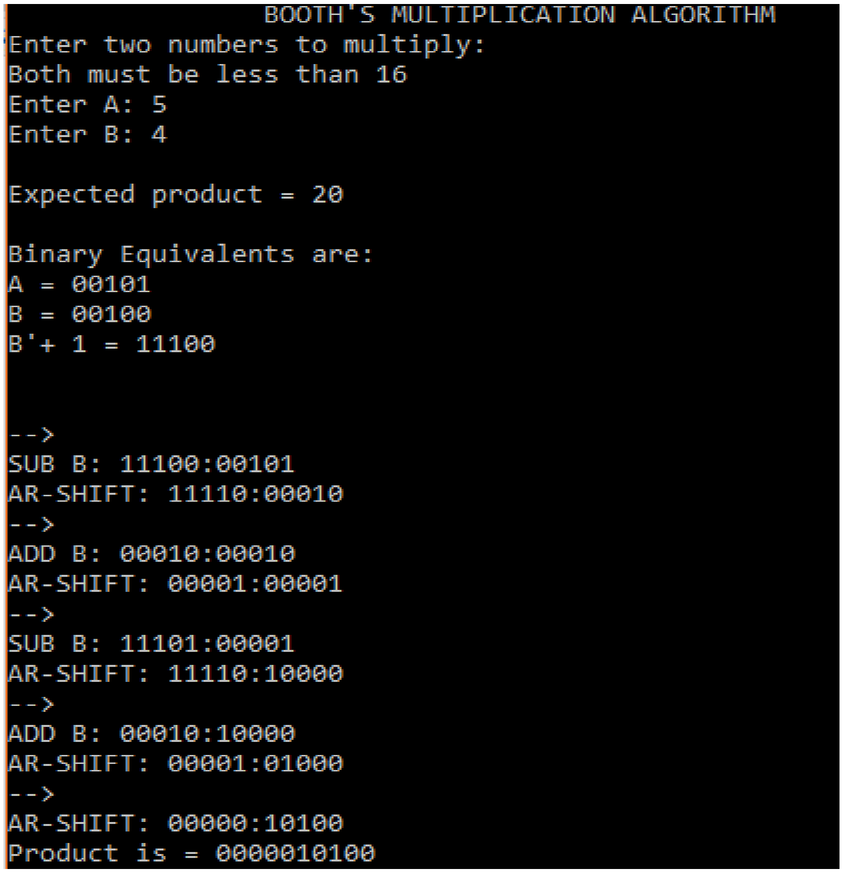
C PROGRAM OF BOOTH’S MULTIPLICATION

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

int a=0,b=0,c=0,a1=0,b1=0,com[5]={1,0,0,0,0};
int anum[5]={0},anumcp[5] ={0},bnum[5]={0};
int acomp[5]={0},bcomp[5]={0},pro[5]={0},res[5]={0};

void binary(){
    a1 = fabs(a);
    b1 = fabs(b);
    int r, r2, i, temp;
    for(i = 0; i < 5; i++){
        r = a1 % 2;
        a1 = a1 / 2;
        r2 = b1 % 2;
        b1 = b1 / 2;
        anum[i] = r;
        anumcp[i] = r;
        bnum[i] = r2;
        if(r2 == 0){
            bcomp[i] = 1;
        }
        if(r == 0){
            acomp[i] =1;
        }
    }
  //part for two's complementing
  c = 0;
  for( i = 0; i < 5; i++){
        res[i] = com[i]+ bcomp[i] + c;
        if(res[i]>=2){
            c = 1;
        }
```

```
         else
             c = 0;
         res[i] = res[i]%2;
    }
  for(i = 4; i>= 0; i--){
    bcomp[i] = res[i];
  }
  //in case of negative inputs
  if(a<0){
     c = 0;
    for(i = 4; i>= 0; i--){
         res[i] =0;
    }
    for( i = 0; i < 5; i++){
         res[i] = com[i]+ acomp[i] + c;
         if(res[i]>=2){
             c = 1;
         }
         else
             c = 0;
         res[i] = res[i]%2;
    }
    for(i = 4; i>= 0; i--){
         anum[i] = res[i];
         anumcp[i] = res[i];
    }

  }
  if(b<0){
    for(i=0;i<5;i++){
         temp = bnum[i];
         bnum[i] = bcomp[i];
         bcomp[i] = temp;
    }
  }
}
void add(int num[]){
    int i;
  c = 0;
```

```
    for( i = 0; i < 5; i++){
        res[i] = pro[i]+ num[i] + c;
        if(res[i]>=2){
            c = 1;
        }
        else
            c = 0;
        res[i] = res[i]%2;
    }
    for(i = 4; i>= 0; i--){
    pro[i] = res[i];
        printf("%d",pro[i]);
    }
  printf(":");
  for(i = 4; i>= 0; i--){
        printf("%d",anumcp[i]);
    }
}
void arshift(){//for arithmetic shift right
    int temp = pro[4], temp2 = pro[0],i;
    for(i = 1; i <5  ; i++){//shift the MSB of product
        pro[i-1] = pro[i];
    }
    pro[4] = temp;
  for(i = 1; i < 5  ; i++){//shift the LSB of product
        anumcp[i-1] = anumcp[i];
    }
  anumcp[4] = temp2;
    printf("\nAR-SHIFT: ");//display together
  for(i = 4; i>= 0; i--){
        printf("%d",pro[i]);
    }
  printf(":");
  for(i = 4; i>= 0; i--){
        printf("%d",anumcp[i]);
    }
}

void main(){
    int i, q=0;
    printf("\t\tBOOTH'S MULTIPLICATION ALGORITHM");
    printf("\nEnter two numbers to multiply: ");
    printf("\nBoth must be less than 16");
  //simulating for two numbers each below 16
    do{
        printf("\nEnter A: ");
        scanf("%d",&a);
        printf("Enter B: ");
        scanf("%d",&b);
    }while(a>=16 || b>=16);

    printf("\nExpected product = %d", a*b);



    binary();
    printf("\n\nBinary Equivalents are: ");
    printf("\nA = ");
    for(i = 4; i>= 0; i--){
        printf("%d",anum[i]);
    }
    printf("\nB = ");
    for(i = 4; i>= 0; i--){
        printf("%d",bnum[i]);
    }
  printf("\nB'+ 1 = ");
  for(i = 4; i>= 0; i--){
        printf("%d",bcomp[i]);
    }
    printf("\n\n");
    for(i=0;i<5;i++){
        if(anum[i] == q){//just shift for 00 or 11
        printf("\n-->");
            arshift();
     q = anum[i];
        }
        else if(anum[i] == 1 && q == 0){//subtract and shift for 10
        printf("\n-->");
     printf("\nSUB B: ");
```

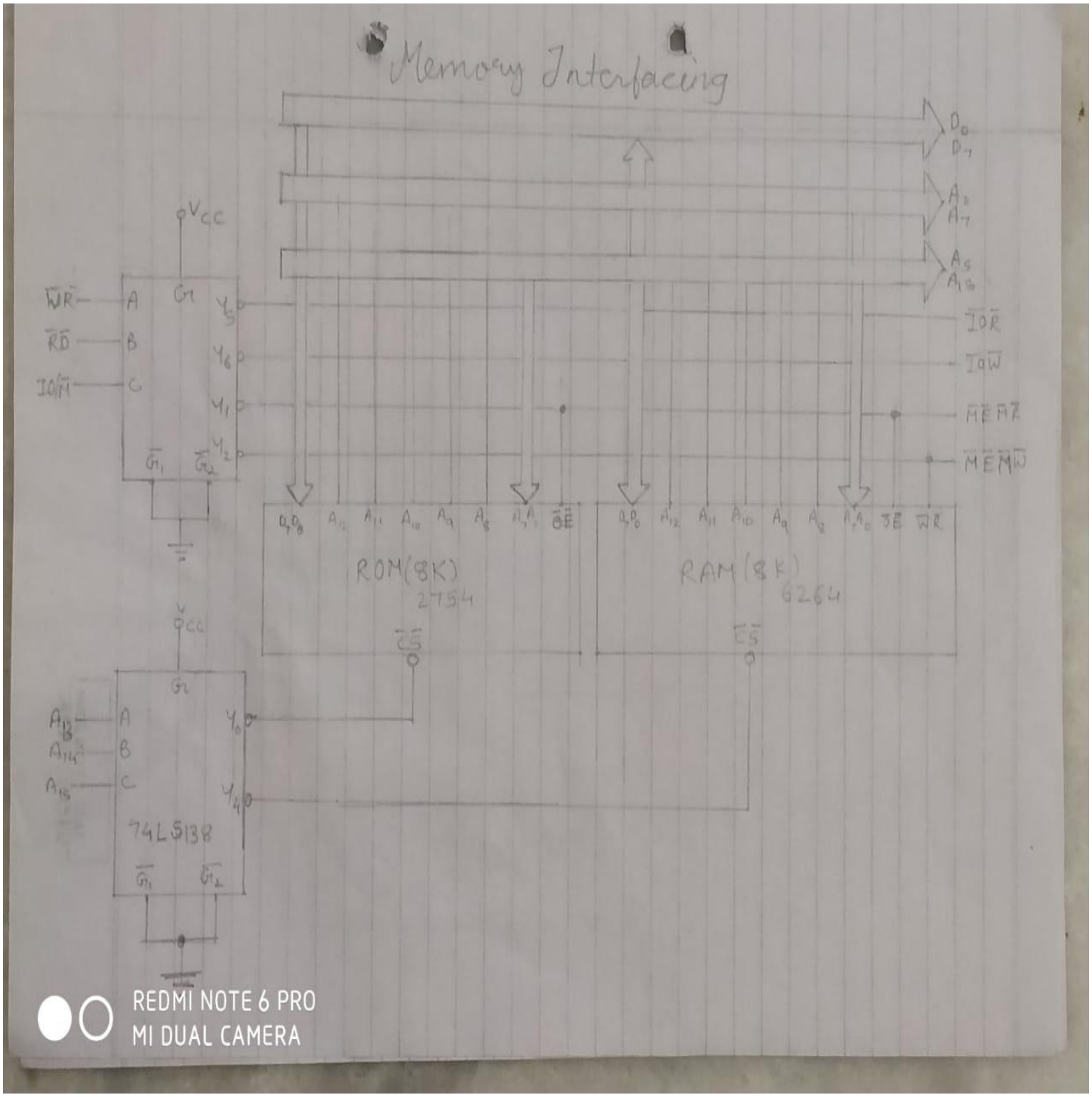
```
                add(bcomp);//add two's complement to implement subtraction
                arshift();
        q = anum[i];
          }
          else{//add ans shift for 01
          printf("\n-->");
        printf("\nADD B: ");
                add(bnum);
                arshift();
        q = anum[i];
          }
    }

    printf("\nProduct is = ");
    for(i = 4; i>= 0; i--){
          printf("%d",pro[i]);
    }
  for(i = 4; i>= 0; i--){
          printf("%d",anumcp[i]);
    }
getch();
}
```



```
                BOOTH'S MULTIPLICATION ALGORITHM
Enter two numbers to multiply:
Both must be less than 16
Enter A: 5
Enter B: 4

Expected product = 20

Binary Equivalents are:
A = 00101
B = 00100
B'+ 1 = 11100


-->
SUB B: 11100:00101
AR-SHIFT: 11110:00010
-->
ADD B: 00010:00010
AR-SHIFT: 00001:00001
-->
SUB B: 11101:00001
AR-SHIFT: 11110:10000
-->
ADD B: 00010:10000
AR-SHIFT: 00001:01000
-->
AR-SHIFT: 00000:10100
Product is = 0000010100
```

Practical-8

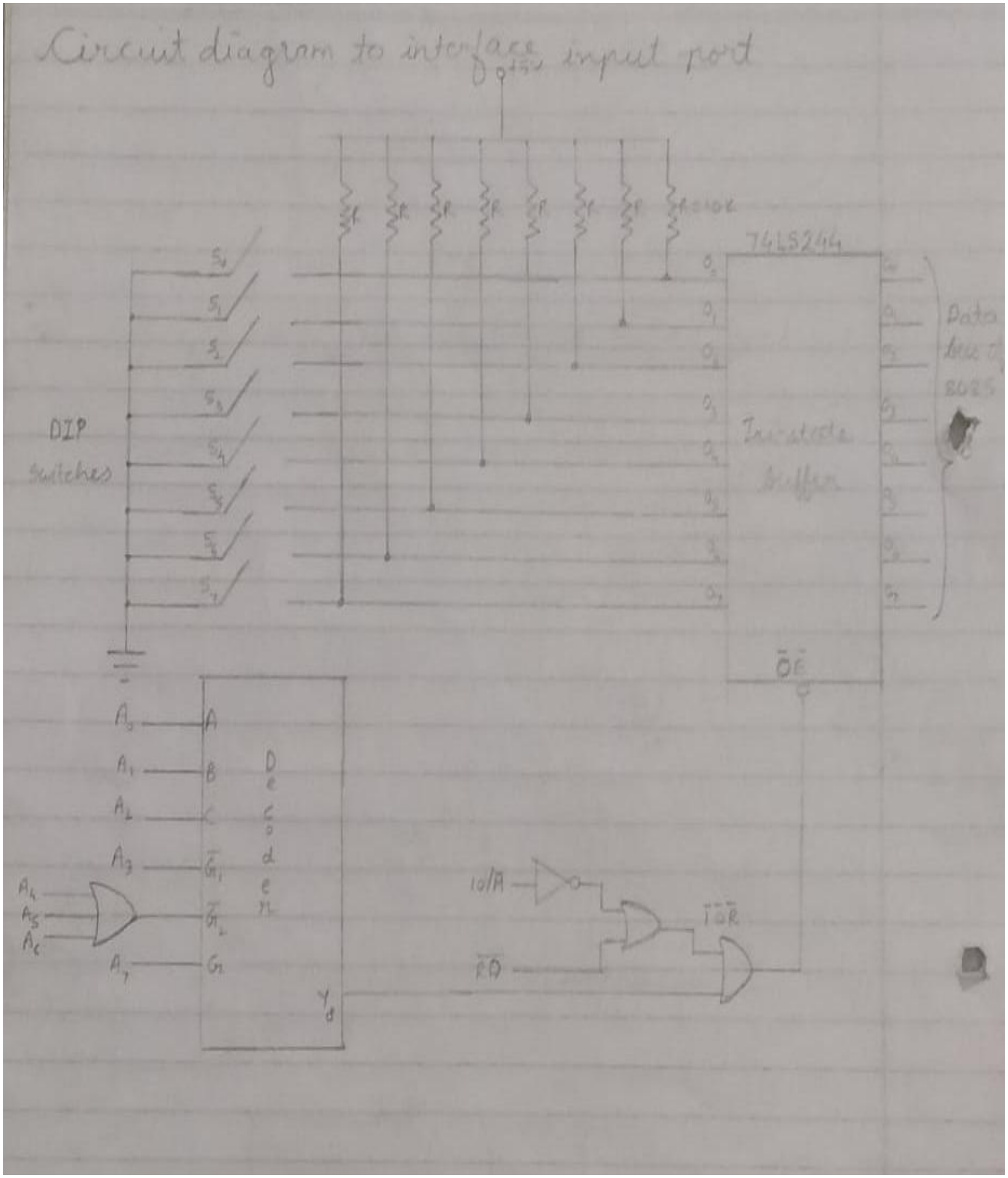
Aim: Design circuit for interfacing memory with processor.



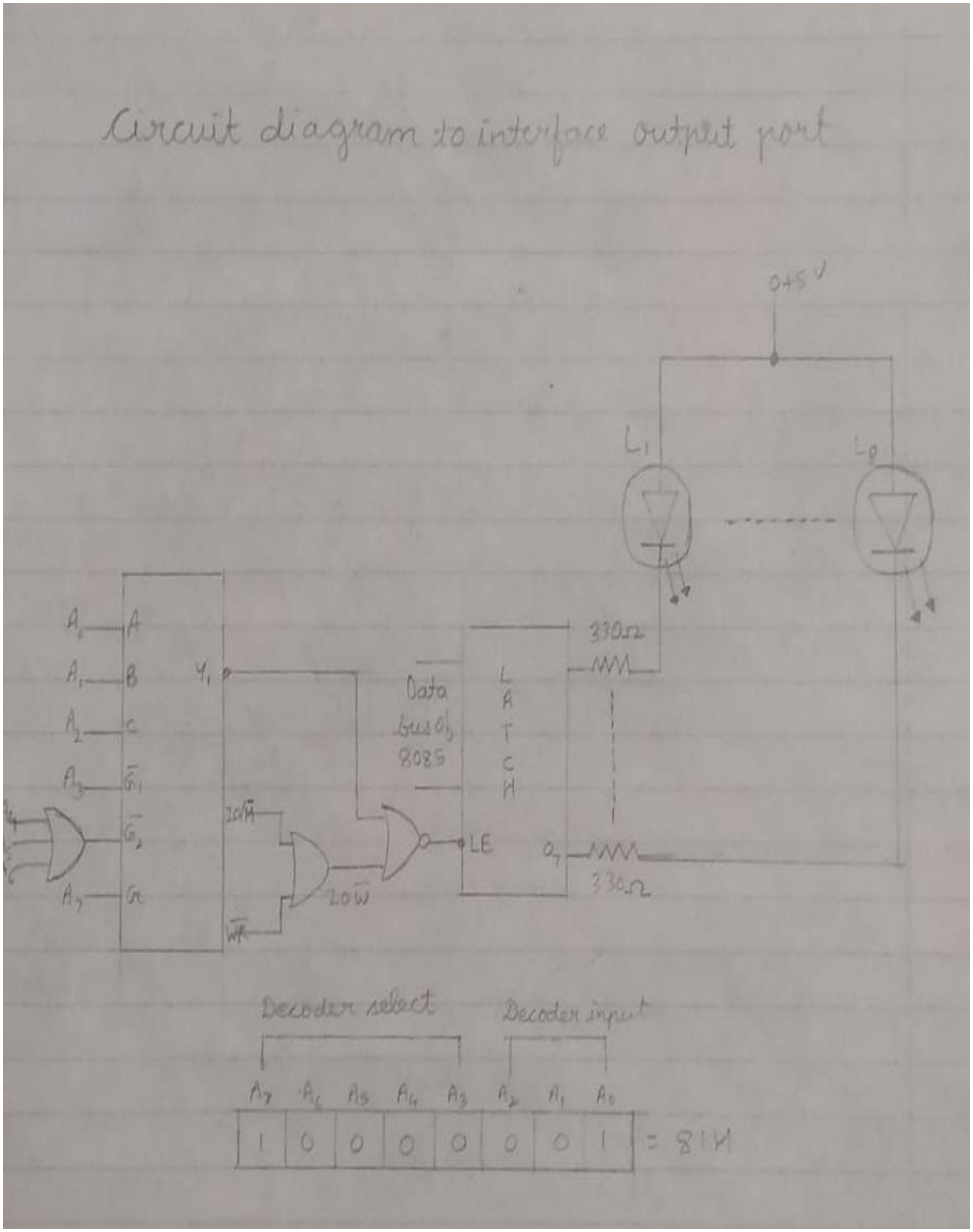
Practical-9

Design circuit for

1) Interfacing Input Port with processor



2) Interfacing Output Port with processor



Circuit diagram to interface output port

A7	A6	A5	A4	A3	A2	A1	A0	
1	0	0	0	0	0	0	1	= 81H

Practical-10

1) Differentiate between RISC and CISC.

No	Characteristics	CISC	RISC
1	Instruction size	Varies	Fixed
2	Instruction length	1,2,3 or 4 bytes	4 bytes
3	No of Instruction	More	Less
4	Instruction decoding	Serial to decode	Easy to decode
5	Instruction semanctics	Varies from simple to complex	Almost always one simple operation
6	Addressing mode	Supports complex Addressing modes	Complex addressing modes are systhesized in software
7	Instruction execution speed	Slow	Medium
8	Registers	Few	Many
9	Hardware	Complicated	Simple
10	Memory access	Frequently	Rarely

2) Case study of different RISC and CISC processors.

2.1. ARM Cortex-M

The ARM Cortex®-M processor family is an upwards compatible range of energy-efficient , It delivers more features at a lower cost, increasing connectivity, better code reuse and improved energy efficiency.
The Cortex-M family is optimized for cost and power sensitive MCU and mixed-signal devices for end applications such as smart metering, human interface devices, automotive and industrial control systems, white goods, consumer products and medical instrumentation.
More information on ARM embedded products and resources is available in the Embedded Group on ARM Connected Community.

Industry standard

ARM Cortex-M processors is a global microcontroller standard, having been licensed to over 40 ARM partners including leading vendors such as Freescale, NXP Semiconductors, STMicroelectronics, Texas Instruments, and Toshiba. Using a standard processor allows ARM partners to create devices with a consistent architecture while enabling them to focus on creating superior device implementations.

Energy efficiency

Lower energy costs, longer battery life

- Run at lower MHz or with shorter activity periods
- Architected support for sleep modes

- Work smarter, sleep longer than 8/16-bit

Smaller code

Lower silicon costs

- High density instruction set
- Achieve more per byte than 8/16-bit devices
- Smaller RAM, ROM or Flash requirement

Ease of use

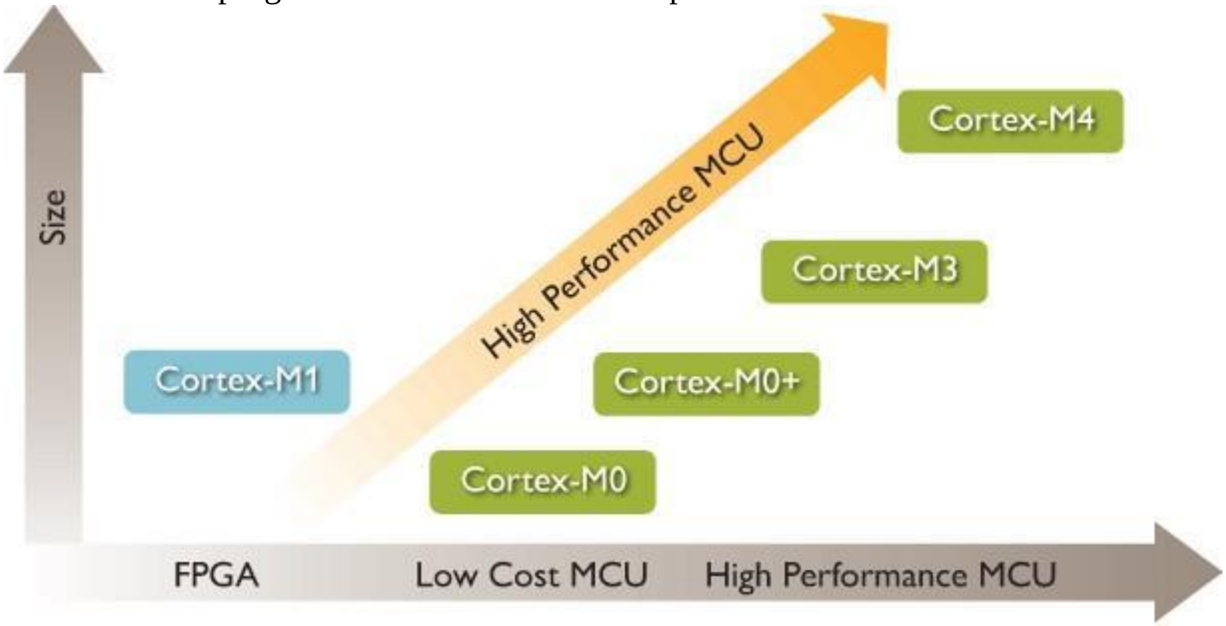
Faster software development and reuse

- Global standard across multiple vendors
- Code compatibility
- Unified tools and OS support

Comparing Cortex-M processors

ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	ARM Cortex-M4
"8/16-bit" applications Low cost and simplicity	"8/16-bit" applications Low cost, best energyefficiency	"16/32-bit" applications Performance, general purpose	"32-bit/DSC" applications Efficient digital signal control

Cortex-M family processors are all binary upwards compatible, enabling software reuse and a seamless progression from one Cortex-M processor to another.



2.2. Intel-i7 Processor

1. INTRODUCTION

The Intel core i7-900 desktop processor extreme edition series and Intel core i7-900 desktop processor series are intended for high performance high –end desktop, uni-processor server and workstation systems. Core i7 is the processor using Nehalem microarchitecture. With faster, intelligent multi-core technology that applies processing power where it is needed the most the processor implements key new technologies: A. Integrated memory controller B. Point to point link interface based on C. Intel quick path interconnection.

Nehalem is the code name for the Intel processor microarchitecture, successor to core microarchitecture, the first processor released with the Nehalem architecture is the desktop core i7.

The processor is optimized for performance with the power efficiencies of a low-power micro architecture to enable smaller, quieter systems.

The Intel core i7 desktop processor extreme edition series are multi-core processors based on 45nm process technology. The processors supports all the existing streaming SIMD extensions 2(SSE2), streaming SIMD extensions3 (SSE3) and streaming SIMD extensions 4(SSE4). The processor supports several advanced technologies: execute disable bit, Intel 64 technology, enhanced Intel speed step technology, Intel virtualization technology, Intel turbo boost technology and hyper- threading technology.

2. FEATURES

1) Modular design: The core i7 processors have been designed to help Intel create different versions this means 8-core processors, 6-channel memory and larger cache processors.

2) Hyper threading returns: hyper threading is a technology that allows a single core to emulate two cores by using unused core hardware to run a separate thread.

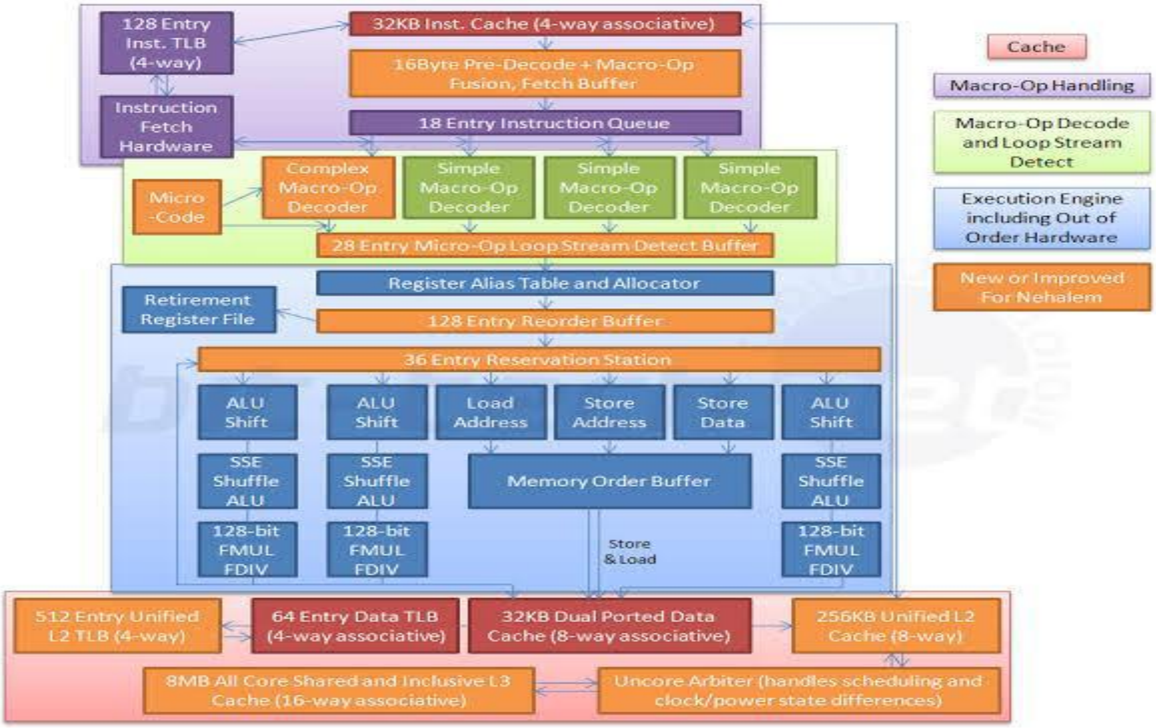
3) Integrated memory controller: Intel has placed all memory controlling hardware directly into the processor this means more bandwidth and lower latencies by passing the FSB. More memory is supported due to the triple channel.

4) Quick path interconnect: the QPI is connection interface between processor and rest of the system. It runs independent of other modules in the processor it also transmits and receives per clocks so hence this module is rated in transfers per second instead of frequency.

5) Turbo modes: The processor is able to self- overclock by changing the multiplier by two speed bins. It will self-overclock if the processor senses that there is enough thermal and power headroom to overclock without straining itself. Usually this happens when there are cores in sleep state due to lack of multiple threads. The PCU (power control unit) is advanced enough to know all this and will be very self-aware.

6) Overclocking: Overclocking is the process of forcing the computer component to run at higher clock rate it was designed in order to increase the performance of the computers.
7) Support for SSE4.2 and SSE4.1 instruction sets: SSE4 is an instruction set used in intel core microarchitecture it consists of 54 instructions referred to as SSE4.1 additionally SSE4.2 a subset consisting of 7 remaining instructions will first be available in core i7.

Architecture of intel-i7 Nehlam


128 Entry Inst. TLB (4-way)
32KB Inst. Cache (4-way associative)
16Byte Pre-Decode + Macro-Op Fusion, Fetch Buffer
Instruction Fetch Hardware
18 Entry Instruction Queue
Micro -Code
Complex Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
28 Entry Micro-Op Loop Stream Detect Buffer
Register Alias Table and Allocator
Retirement Register File
128 Entry Reorder Buffer
36 Entry Reservation Station
ALU Shift
ALU Shift
Load Address
Store Address
Store Data
ALU Shift
SSE Shuffle ALU
SSE Shuffle ALU
Memory Order Buffer
SSE Shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & Load
128-bit FMUL FDIV
512 Entry Unified L2 TLB (4-way)
64 Entry Data TLB (4-way associative)
32KB Dual Ported Data Cache (8-way associative)
256KB Unified L2 Cache (8-way)
8MB All Core Shared and Inclusive L3 Cache (16-way associative)
Uncore Arbiter (handles scheduling and clock/power state differences)
Cache
Macro-Op Handling
Macro-Op Decode and Loop Stream Detect
Execution Engine including Out of Order Hardware
New or Improved For Nehalem

3) Describe various physical forms of Interconnection structures.

Interconnection Structures
- Time-shared common bus
- Multiport Memory
- Crossbar Switch
- Multistage Switching Network
- Hypercube system Time-shared common bus

3.1 Time-shared common bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. 1. Only one processor can communicate with the memory or another processor at any given time.

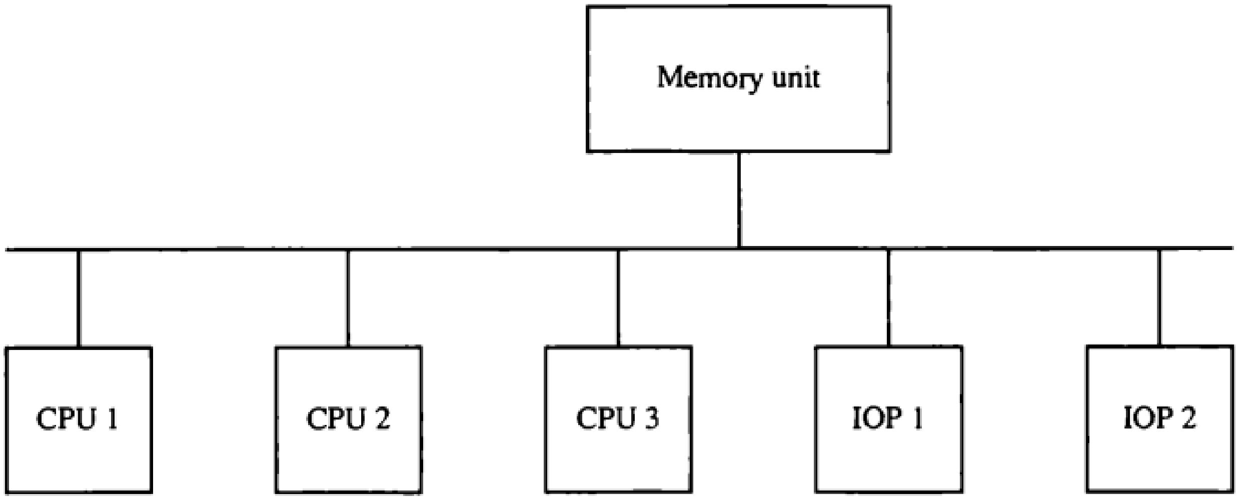


Figure 1 Time-shared common bus organization.

Transfer operations are conducted by the processor that is in control of the bus at the time.

Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer.

A command is issued to inform the destination unit what operation is to be performed.

The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated.

The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

A single common-bus system is restricted to one transfer at a time.

This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus.

As a consequence, the total overall transfer rate within the system is limited by the speed of the single path.

The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. 2.

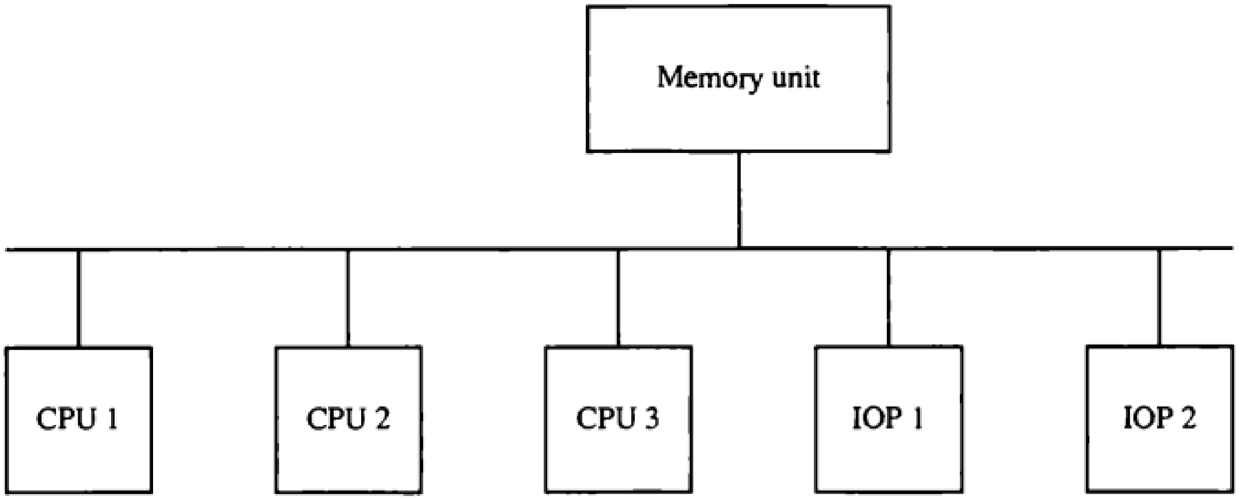


Figure 1 Time-shared common bus organization.

3.2 Multiport Memory

A multiport memory system employs separate buses between each memory module and each CPU.
This is shown in Fig . 3 for four CPUs and four memory modules (MMs).
Each processor busis Connected to each memory module.
A processor bus consists of the address, data, and control lines required to communicate with memory.
The memory module is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port.
The priority for memory acoess associated with each processor may be established by the physical port position that its bus occupies in each module.
Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.
The advantage of the multi port memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory.
The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this interconnection structure is usually appropriate for systems with a small number of processors.

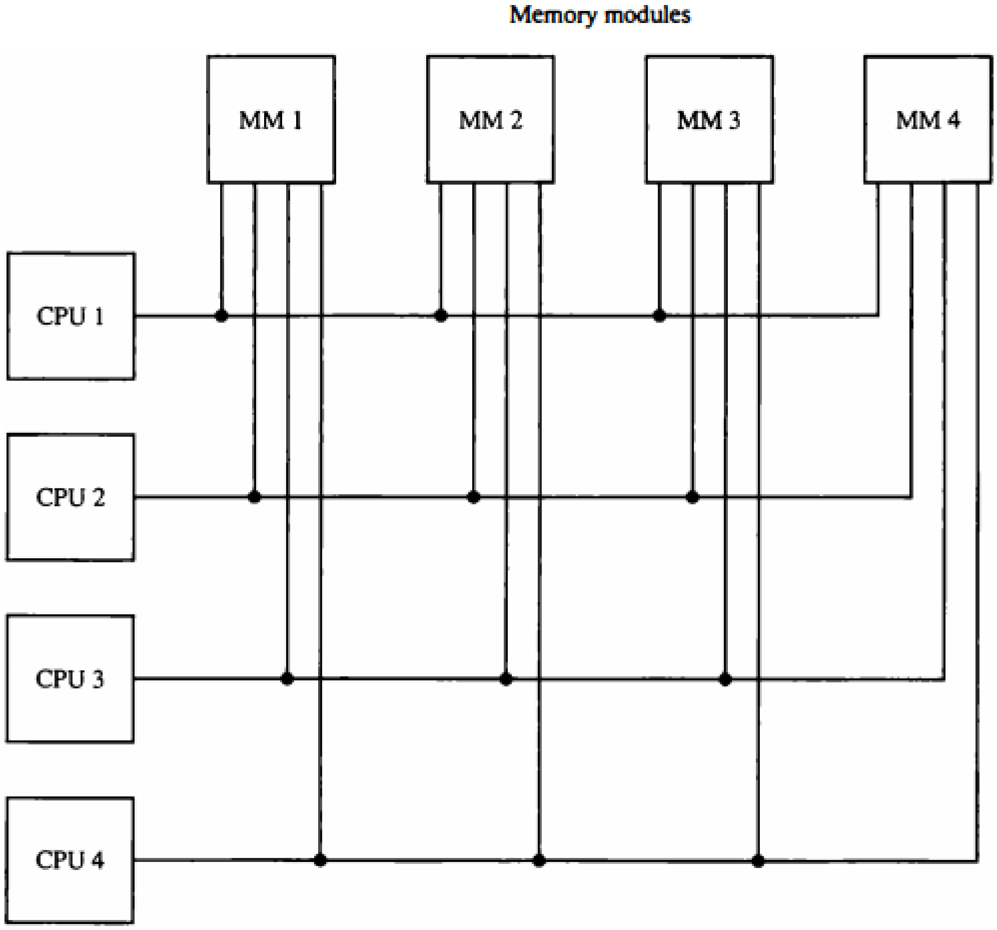


Figure 3 Multiport memory organization.

3.3 Crossbar Switches

The Crossbar exchanges were developed during 1940s. They achieve full access and non-blocking capabilities with the Crossbar switches and common control equipment, used in the Crossbar exchanges. The active elements called Crosspoints are placed between the input and the output lines. In the common control switching systems, the separation between switching and control operations allows the usage of switching networks by a group of common control switches to establish many calls at the same time on a shared basis.

The Features of Crossbar Switches

The features are described in brief below −

While processing a call, the common control system helps in the sharing of resources.

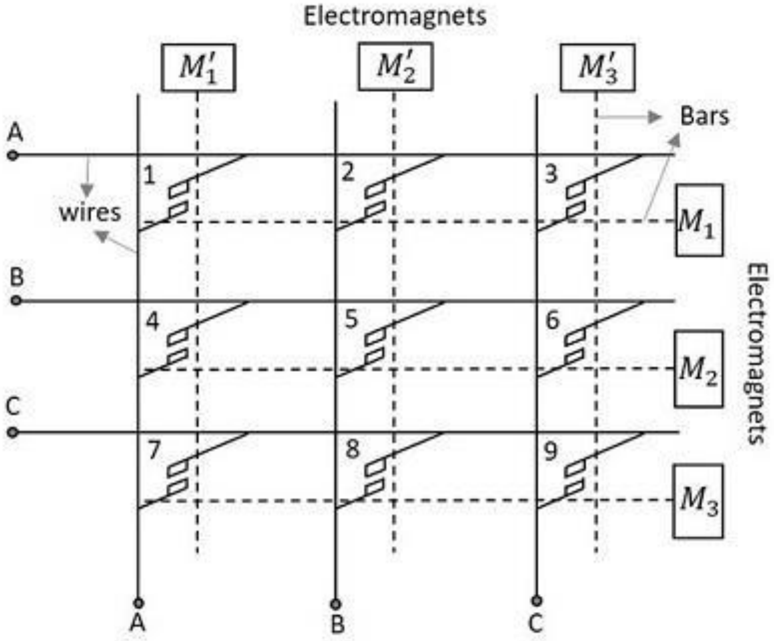
The specific route functions of call processing are hardwired because of the Wire logic computers.

The flexible system design helps in the appropriate ratio selection is allowed for a specific switch.

Fewer moving parts ease the maintenance of Crossbar switching systems.
The Crossbar switching system uses the common control networks which enable the switching network to perform event monitoring, call processing, charging, operation and maintenance as discussed previously. The common control also provides uniform numbering of subscribers in a multi-exchange area like big cities and routing of calls from one exchange to another using the same intermediate exchanges. This method helps to avoid the disadvantages associated with the step-by-step switching method through its unique process of receiving and storing the complete number to establish a call connection.

Crossbar Switching Matrix

The Crossbar arrangement is a matrix which is formed by the M X N sets of contacts arranged as vertical and horizontal bars with contact points where they meet. They need nearly M + N number of activators to select one of the contacts. The Crossbar matrix arrangement is shown in the following figure.



3 X 3 Crossbar Switching

3.4 Multistage Switching Network

The general concept of the multi-stage interconnection network, together with its routing properties, have been used in the preceding chapter to describe the operation of various designs of fast packet switch. In this chapter those networks that bear particular relevance to applications within the eld of fast packet switching will be described in some detail within the context of interconnection networks in general.

3.5 Hypercube system Time-shared common bus

Hypercube networks are a type of network topology used to connect multiple processors with memory modules and accurately route data. Hypercube networks consist of 2m nodes. These nodes form the vertices of squares to create an internetwork connection. A hypercube is basically a multidimensional mesh network with two nodes in each dimension. Due to similarity, such topologies are usually grouped into a k-ary d-dimensional mesh topology family where d represents the number of dimensions and k represents the number of nodes in each dimension.

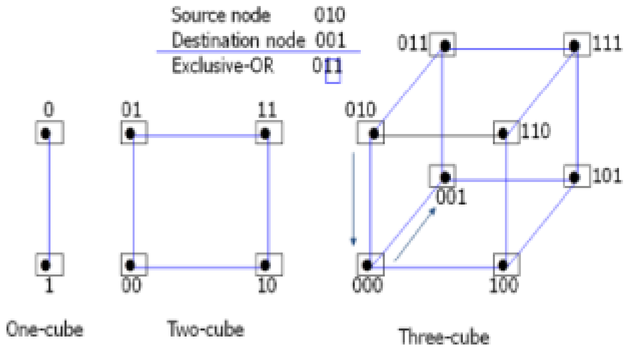


Fig: Hypercube structures for n=1,2,3