Tutorial One

Submitted By:

Pradip Dhungana
201751
BESE 4th Day

1) Explain the levels of programming language.

=> levels of programming language are:
(i) High level programming language
(ii) Assembly level language
(iii) Machine level language

(i) High level programming language:
These languageshide the details of the computer and operating system on which they will run from the programmer. They are said to be platform-independent. The same code can be converted and run on computers with different hp and operating systems without modification. languages such as C++, Java, and Fortran are high-level language.

(ii) Assembly level language:
Assembly language are at a much lower level of abstraction. Each hp has its own assembly language. A program written in the assembly language of one microprocessor

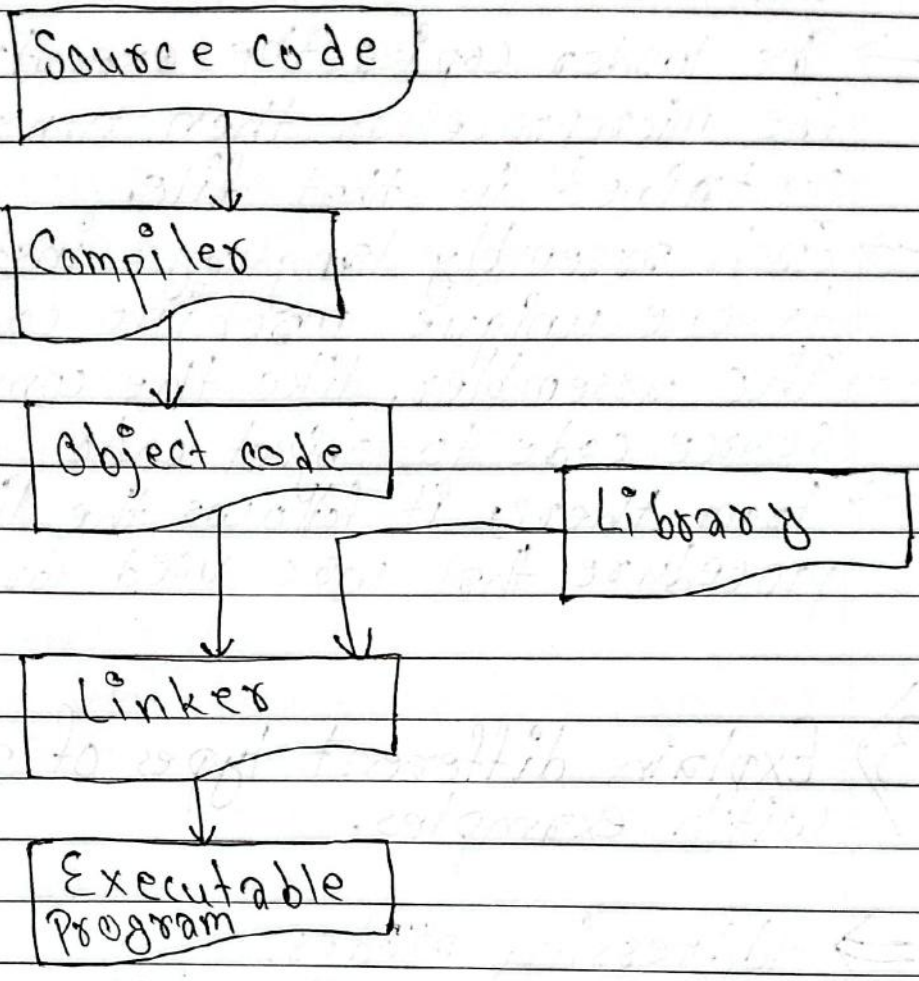
cannot be run on a computer that has a different microprocessor - with one very important exception. It is platform dependent and Assembly level language makes use of mnemonics instead of numeric generation codes.

(iii) Machine level language:
The lowest level of programming languages are machine languages. These languages contain the binary values that cause the microprocessor to perform certain operations. When a microprocessor reads and executes an instruction, it is a machine language instruction. Programmers do not write programs in machine language. It has faster execution time.

2) Explain assembly language & compiling programming languages process with suitable examples.

=> Assembly level language:
↳ Makes use of mnemonics instead of numeric generation codes.
↳ platform dependent
↳ Assembler converts into machine language.
↳ Slower than machine language.

→ Compiling & assembling program



→ A program written in high level language is input to a compiler. The compiler makes sure that every statement in program is valid.
→ When a program has no errors, the compiling of the program will be finished i.e. source code generates object code.
→ At this point, the program has been compiled successfully, but it is not yet ready to be executed.
→ Some programs use the object code of other program in addition to their own.
→ Linker combines your object code with any other required object code.
→ This combined code is stored as an executable

file. It is actually the code that the computer run.

→ The loader copies the executable file into memory the microprocessor then runs the machine code contained in that file.

→ Each assembly language instruction corresponds to one unique machine code instruction.

→ The assembler, like the compiler converts its source code to object code.

→ From there, it follows the linking & loading procedure that was used for compiled code.

3) Explain different types of addressing modes with examples.

⇒ Addressing modes:
↳ The way in which operands are specified in an instruction.

(i) Direct mode:
↳ Contains memory addresses that CPU accesses
Eg: LDAC 5, read data from memory location 5 and stores in accumulator.

(ii) Indirect mode:
↳ The address in the instruction is not the address of operand, rather it contain the address of memory location where operand resides.
E.g: LDAC @5 or LDAC (5), first retrieve the content of memory location 5.

say 10. The CPU then retrieves the content of memory location 10. that actually contains the operand.

(iii) Register direct & register indirect modes:
⇒ Works same as direct & indirect modes, except they do not specify memory addres, instead they specify a register.
Eg: LDAC (R) or LDAC @R
LDAC R

(iv) Immediate mode:
↳ Operand specified is not an address, it is actual data to be used.
Eg: LDAC #5 moves the data value 5 into accumulator.

(v) Implicit mode:
↳ doesn't explicitly specify an operand, instruction always refers to accumulator
Eg: CLAC, which clears the accumulator.

(vi) Relative mode:
↳ Operand specified is an offset, not the actual address which is added to the content of program counter register to generate required address.
Eg: LDAC $5
⇒ If next instruction is at location 12 i.e. PC has value 12, the instruction reads data from 12+5=17 location & stores in accumulator.

(viii) Index mode & base address mode:

↳ Index mode works like relative mode, except the address supplied by instruction is added to the content of index register.

Eg: LOAC 5(X), reads from 10+5=15
(index register contains value 10)

↳ Base address mode is same as index mode except index register is replaced by base register.

4) Write about ISA. What are the factors to be considered while designing ISA? Explain.

⇒ ISA (Instruction set Architecture):

↳ Very important in the design of the microprocessor

↳ Poorly designed ISA, even if implemented well leads to bad microprocessor design

↳ Well designed ISA leads to powerful microprocessor.

↳ No magical formula for designing ISA: same requirement can have different ISA design, each of which can be valid.

↳ Must evaluate trade-off's between performance, size and cost.

↳ Before beginning the design of ISA, it is to be decided what should ISA and its processor be able to do.

↳ If processor is to be used for general purpose computing, requires relatively large set of instruction to perform

variety of tasks; for specialized processor, the task of microprocessor is very little and well known in advance.

↳ Orthogonality:

Instructions are orthogonal if they do not overlap or perform the same function. Good ISA design should minimize the overlapping.

→ Factors that should be considered while designing ISA:

(A) ⇒ Optimization of register set:

↳ If a processor is to be used for general purpose computing, such as in a personal computer, it will probably require quite a rich instruction set architecture.
Registers have a large effect on the performance of a CPU. The CPU can store data in its internal registers instead of memory. For many program segments, such as program loops, having data in registers significantly speeds up program execution. Having too few registers causes a program to make more references to memory, thus reducing performance.

(B) Type and size of data that microprocessor uses:

↳ If a µp will have to process floating-point data, for example, it is important to include instructions in the instruction set architecture that works on floating-point data. It is also necessary to incorporate registers to store

floating point values.

(c) Are interrupts needed?
↳ Very few tasks absolutely require that a µp have interrupts but many tasks can be performed more efficiently if interrupts are incorporated into the microprocessor. If interrupts are included, the instruction set architecture must include the instructions & registers needed to process the interrupts.

(d) Are conditional instruction needed?
↳ A jump or branch instruction may be conditional; that is, some condition outside of the instruction determines whether or not the jump is taken.

5) What do you mean by Instruction set in the processor? How are they different from micro instructions? Explain the machine cycles associated with the instruction cycle with the example.

⇒ An instruction set is a collection of commands or operations that a processor can understand and execute. These commands define the tasks the processor can perform, such as arithmetic operations, data movement and control flow. Programmers use instructions from the instruction set to write software and perform computations on a computer system

b) Micro instructions represent the lower-level control signals and operations that the processor's hardware uses to execute those instructions efficiently. While the Instruction set represents the higher level commands that software developers use to write programs.

b) The instruction cycle is the procedure a microprocessor goes through to process an instruction. First the up fetches, or reads the instruction from memory. Then it decodes the instruction, determining which instruction it has fetched. Finally, it performs the operations necessary to execute the instruction.

6) Explain the basic computer organization with a block diagram.

⇒

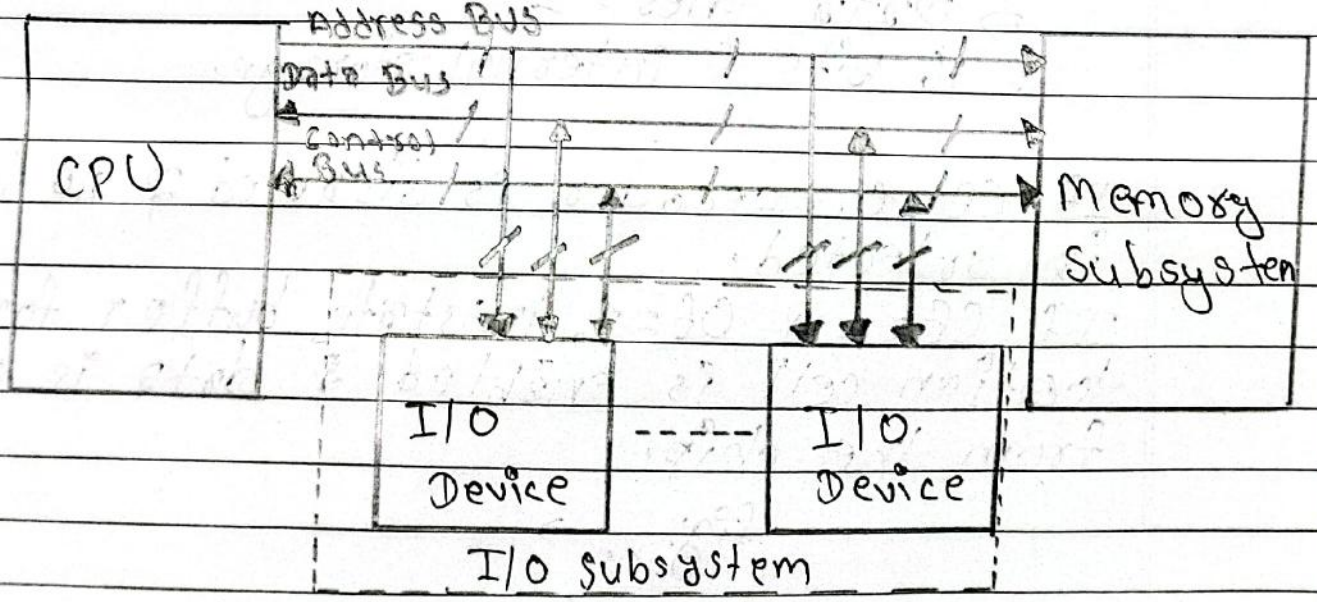


fig: Basic Computer Organization

→ This organization has 3 sub systems: CPU, memory sub-system and I/O sub-system.
→ CPU performs various operations and controls the computer.
→ Memory sub-system is used to store programs being executed by the CPU along with the necessary data.
→ The I/O sub-system allows the CPU to interact with input and output devices such as keyboards, monitors, etc.

7) Explain the linear chip organization with an example.

⇒ Here,
Consider an 8×2 ROM chip:
⇒ Address line = 3
⇒ Data lines = 2
⇒ 16 bits of internal storage.

If CE = 0, decoder is disabled & no location is selected.
If CE = 1 & OE = 1, tri-state buffer for that location cell is enabled & data is output from the chip.

fig. →

→ As the number of locations increases, the size of address decoder needed, becomes extremely large. To remedy this problem, the memory chip can be designed using multiple dimension of decoding.

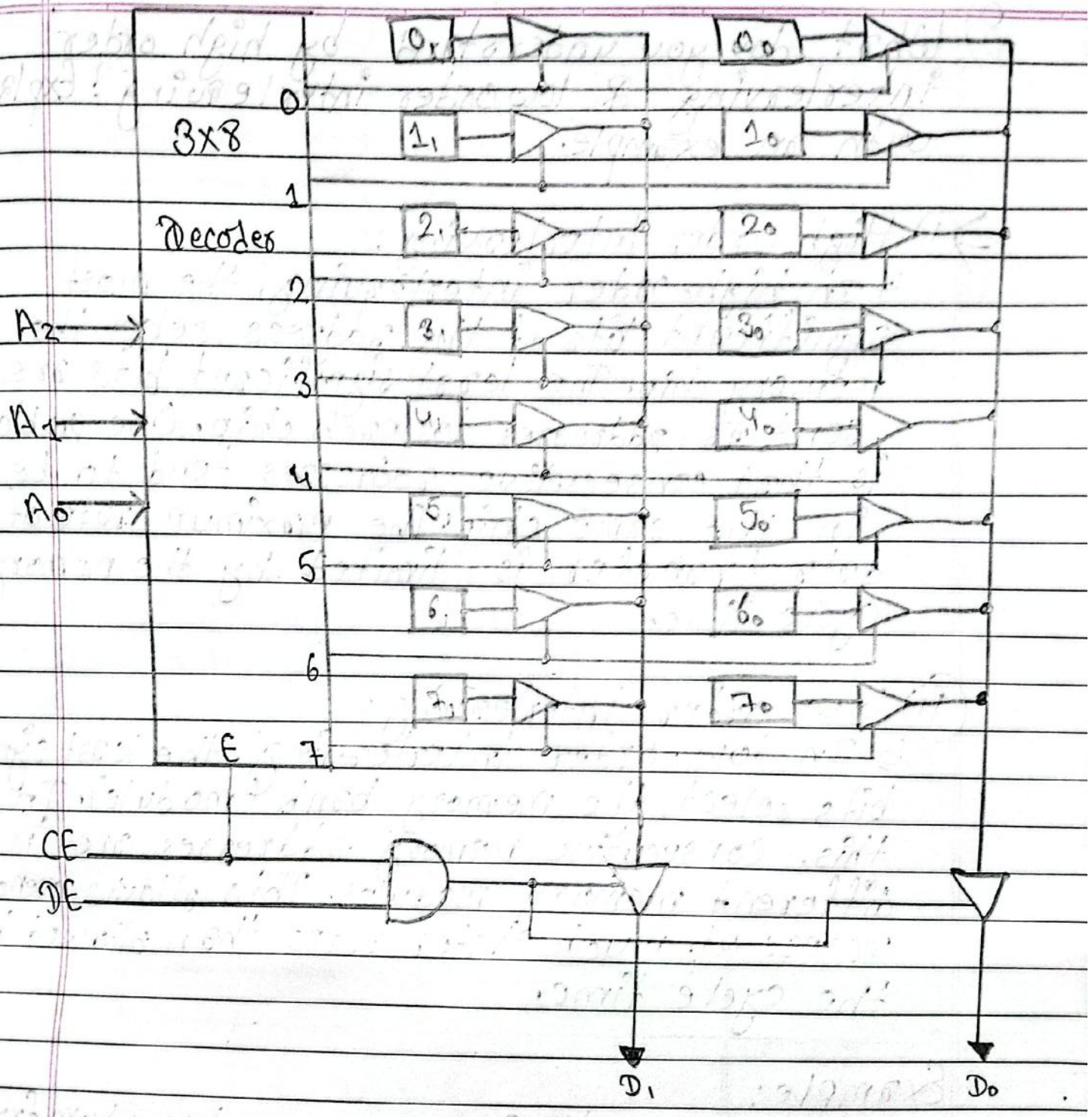


fig: Internal linear organization of an 8x2 ROM chip.

8) What do you understand by high order interleaving & low order interleaving? Explain with an example.

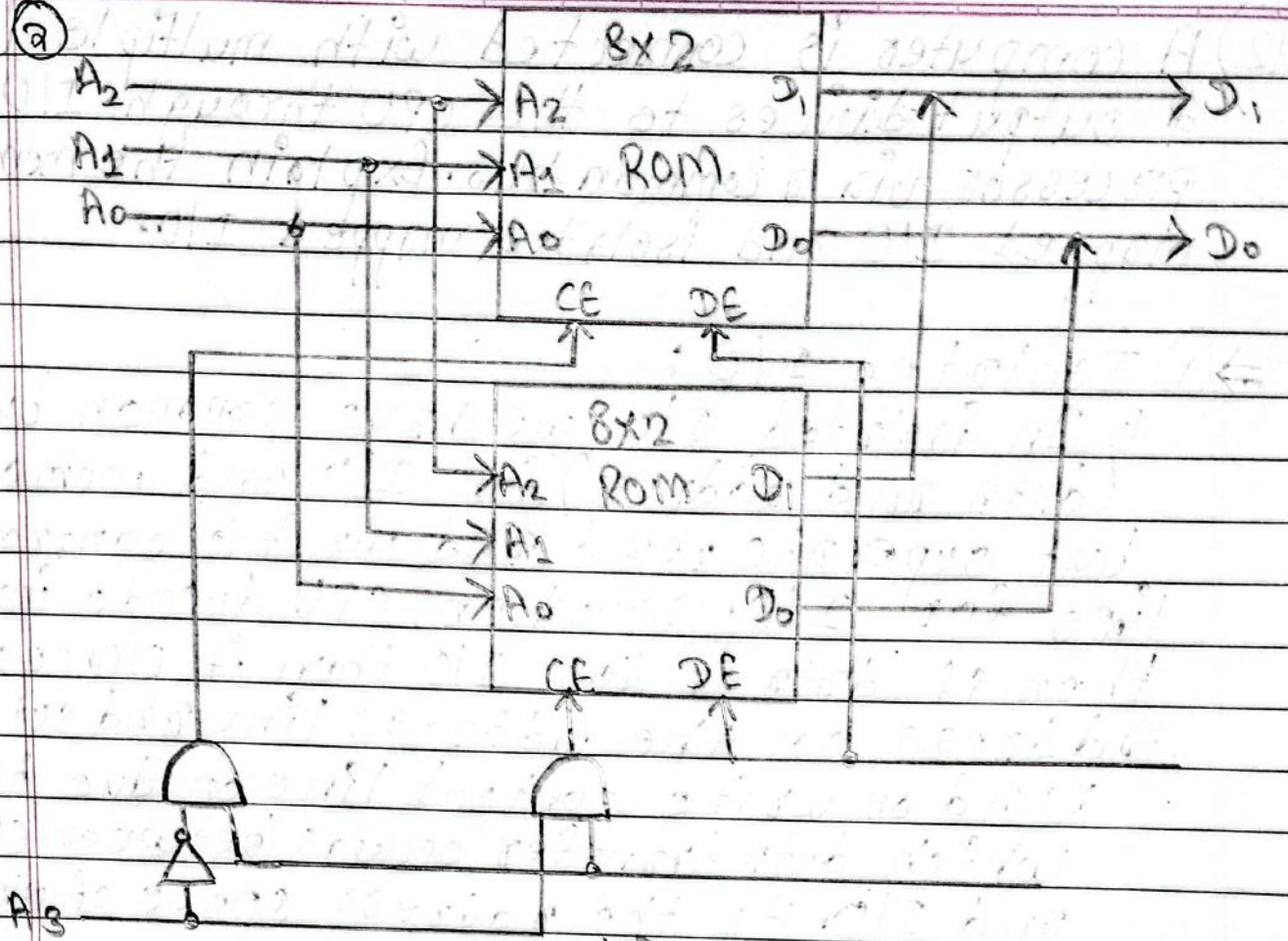
⇒ (I) High Order Interleaving:
↳ In high-order interleaving, the most significant bits of the address select the memory chip. The least significant bits are sent as addresses to each chip. One problem is that consecutive addresses tend to be in the same chip. The maximum rate of data transfer is limited by the memory cycle time.

(II) low order Interleaving:
↳ In low-order interleaving, the least significant bits select the memory bank (module). In this, consecutive memory addresses are in different memory modules. This allows memory access at much faster rates than allowed by the cycle time.

Example:
Constructing a 16x2 memory subsystem from two 8x2 ROM chips with:
a) high-order interleaving
b) low-order interleaving

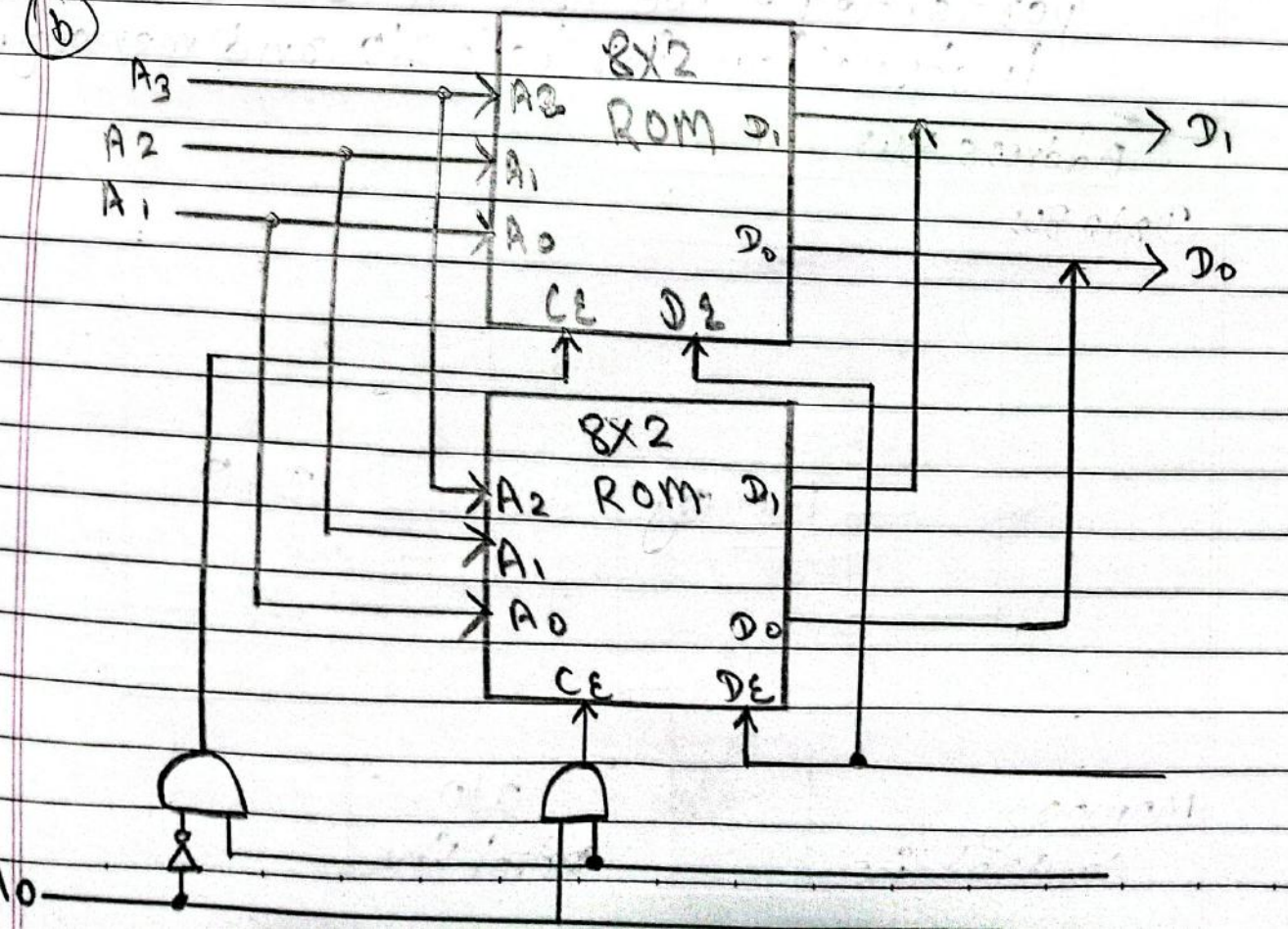
→

(a)



(a) (high-order interleaving)

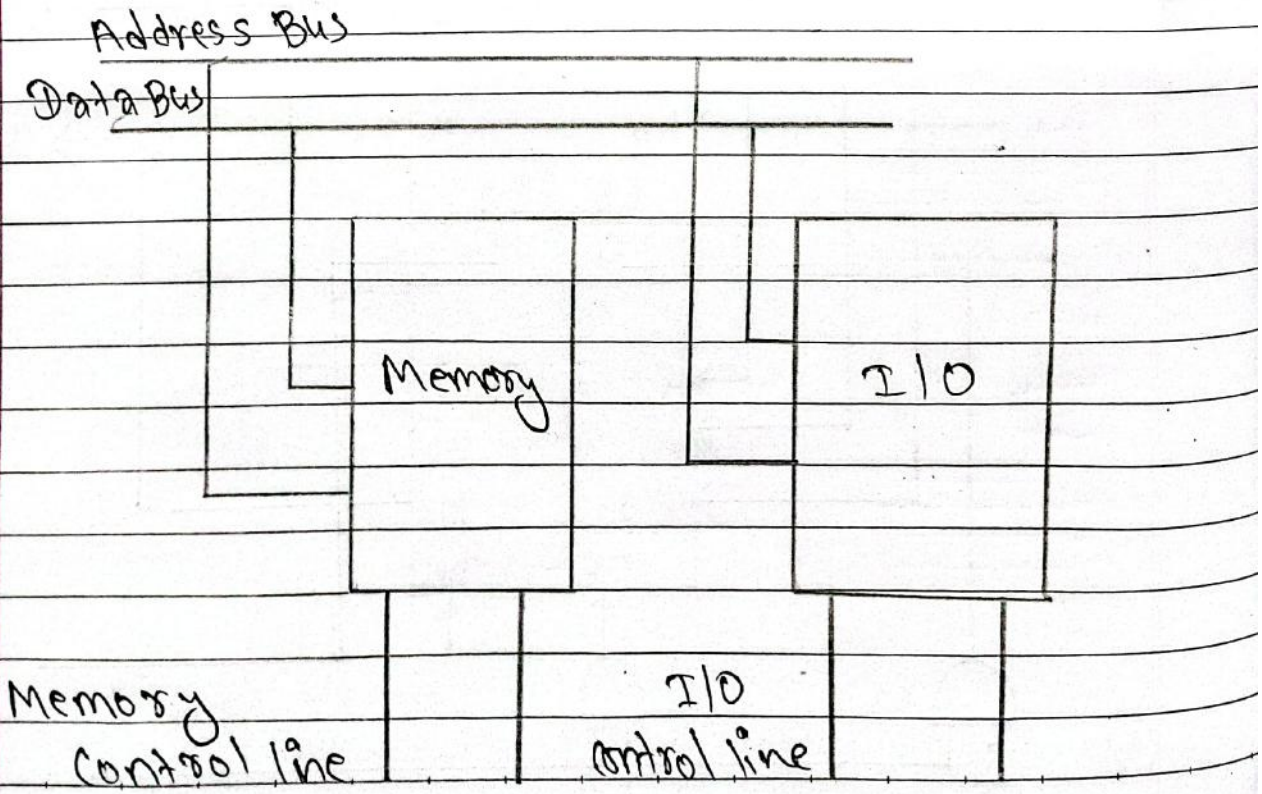
(b)



12) A computer is connected with multiple input & output devices to the CPU through I/O processor via a common bus. Explain the memory mapped I/O and isolated mapped I/O.

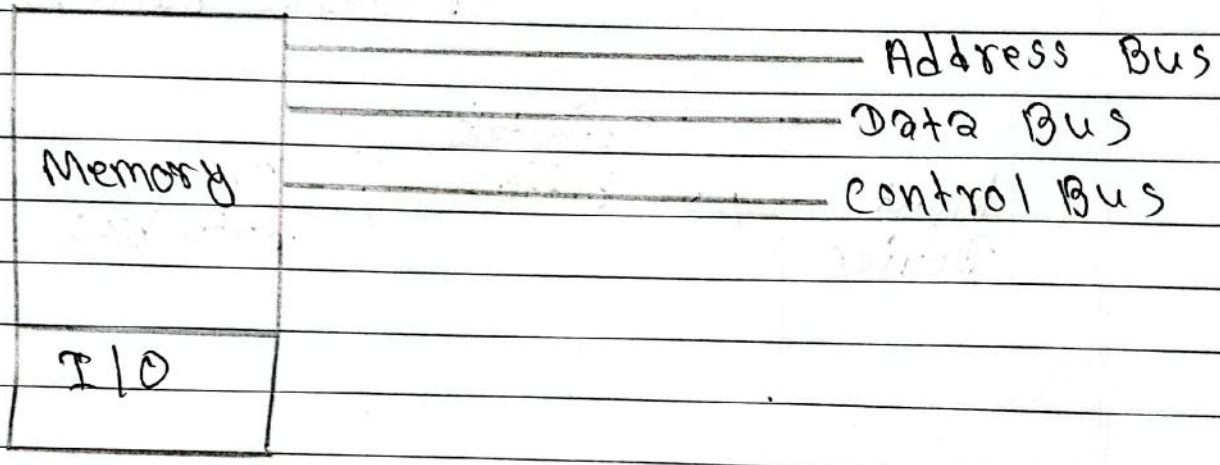
⇒ (1) Isolated I/O:

↳ In isolated I/O we have common bus (data and address) for I/O and memory but seperate read and write control lines for I/O. so when CPU decode instructi then if data is for I/O then it places the address on the address line and set I/O read or write control line on due to which data transfer occurs between CPU and I/O. As the address space of memory and I/O is isolated and the name is so. The address for I/O here is called ports. Here we have different read-write instruction for both I/O and memory.



(11) Memory mapped I/O:

↳ In this case every bus in common due to which the same set of instructions work for memory and I/O. Hence we manipulate I/O same as memory & both have same address space, due to which addressing capability of memory become less because some part is occupied by the I/O.



13) Write about IO system organization and Interfacing for input device and output device with necessary diagram.

⇒ I/O subsystem organization & Interfacing:

① for input device:

↳ The data from the input device goes to the tri-state buffer.

↳ When the values on the address bus and control bus are correct, the buffers are enabled and data passes onto the data bus.

↳ The CPU, then can read these data.

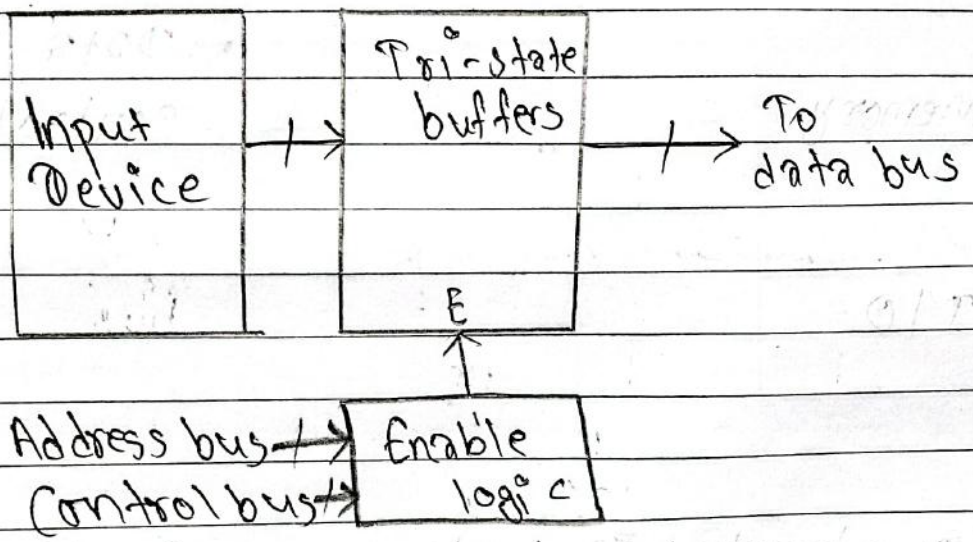
↳ The key to this design is the enable logic.

⇒ For the input device, the read signal (RD) should

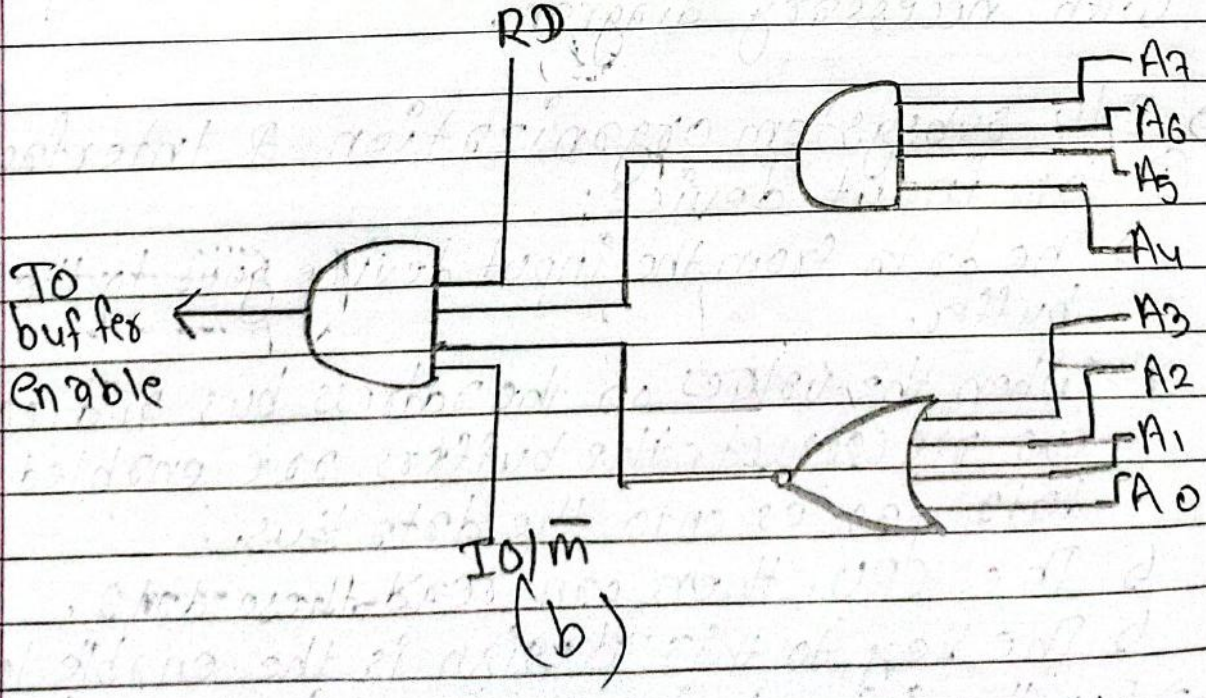
be asserted.

⇒ Figure (b) shows the enable logic for an input device at address 11110000 with 8-b address and control signals RD and IO/Memo (IO/$\overline{M}$)

→ An input device; (a) with its interface and (b) the enable logic for the tri-state buffers



(a)



(b)

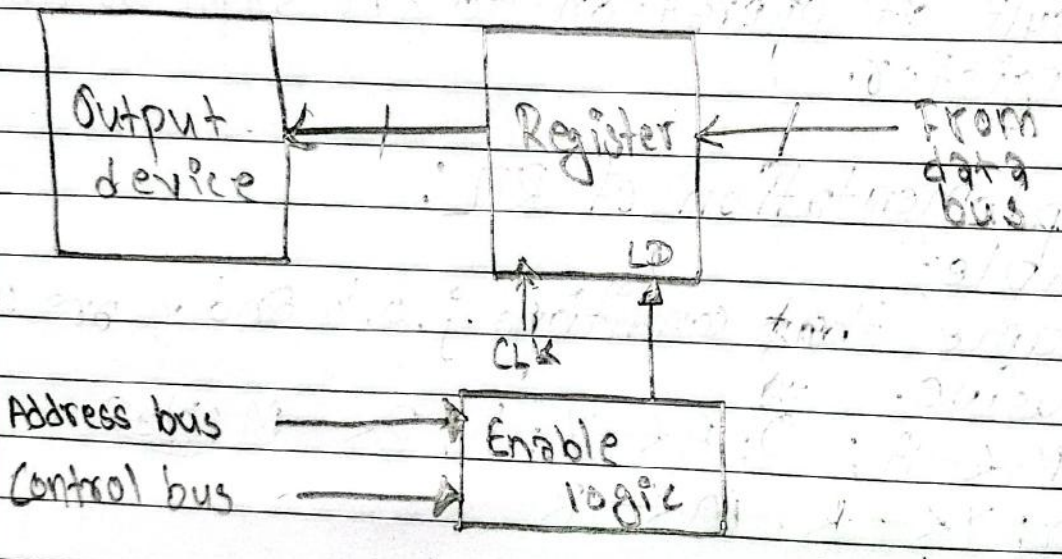
(ii) For output device

↳ The tri-state buffer is replaced by a register.

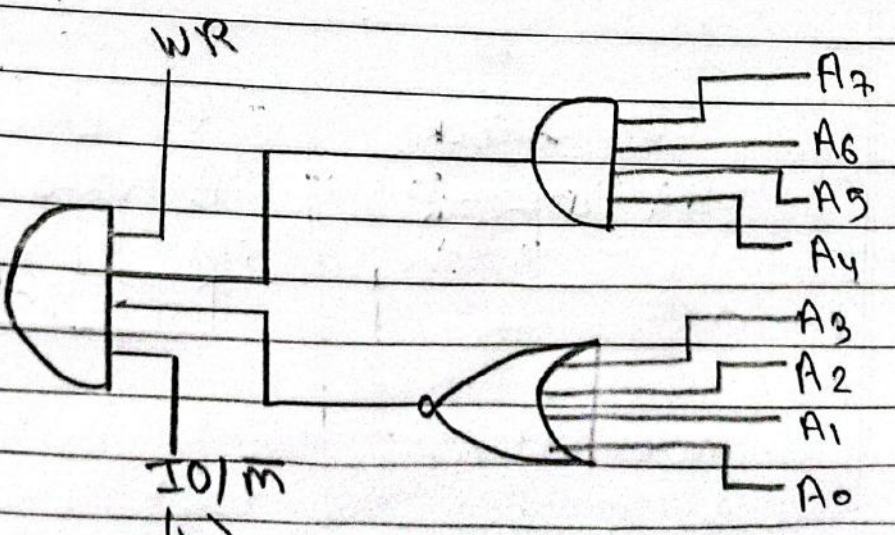
↳ The tri-state buffers are used in input device interface to make sure that no more than one device writes data to the bus at any time.

↳ Since, the o/p devices read data from the bus, they do not need buffers so that data can be made available to all output devices.

→ An output device: (a) with its interface and (b) the load logic for the register.



(a)



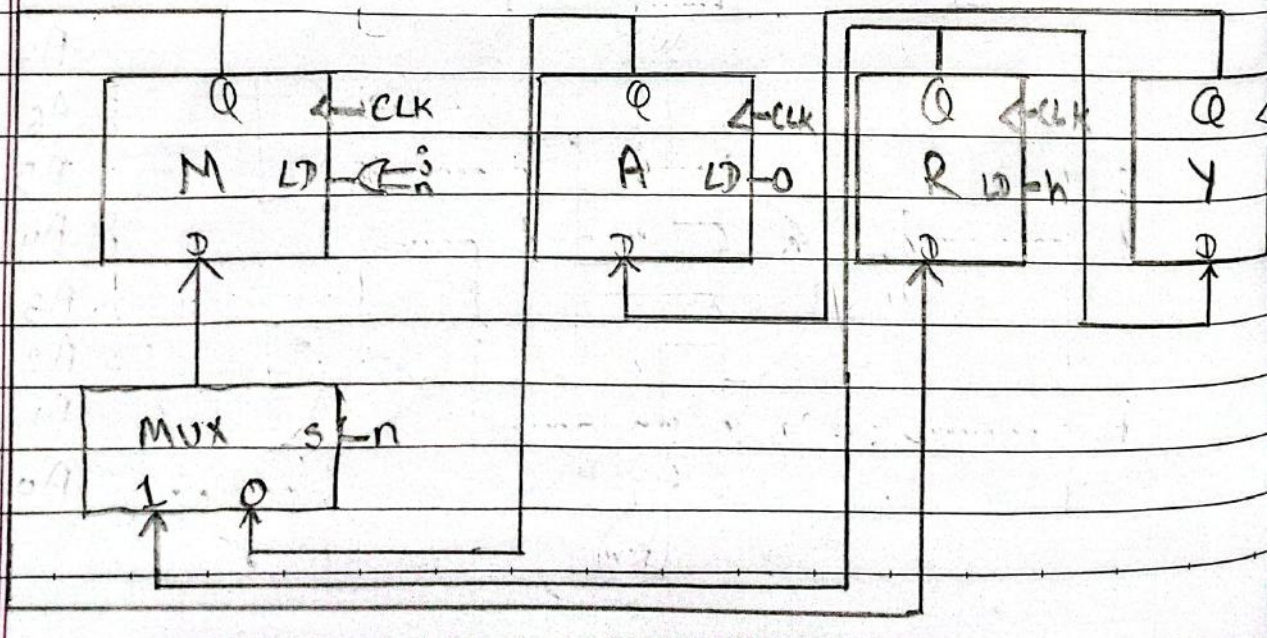
(b)

25) What do you mean by RTL? In how many ways RTL are implemented explain with an example.

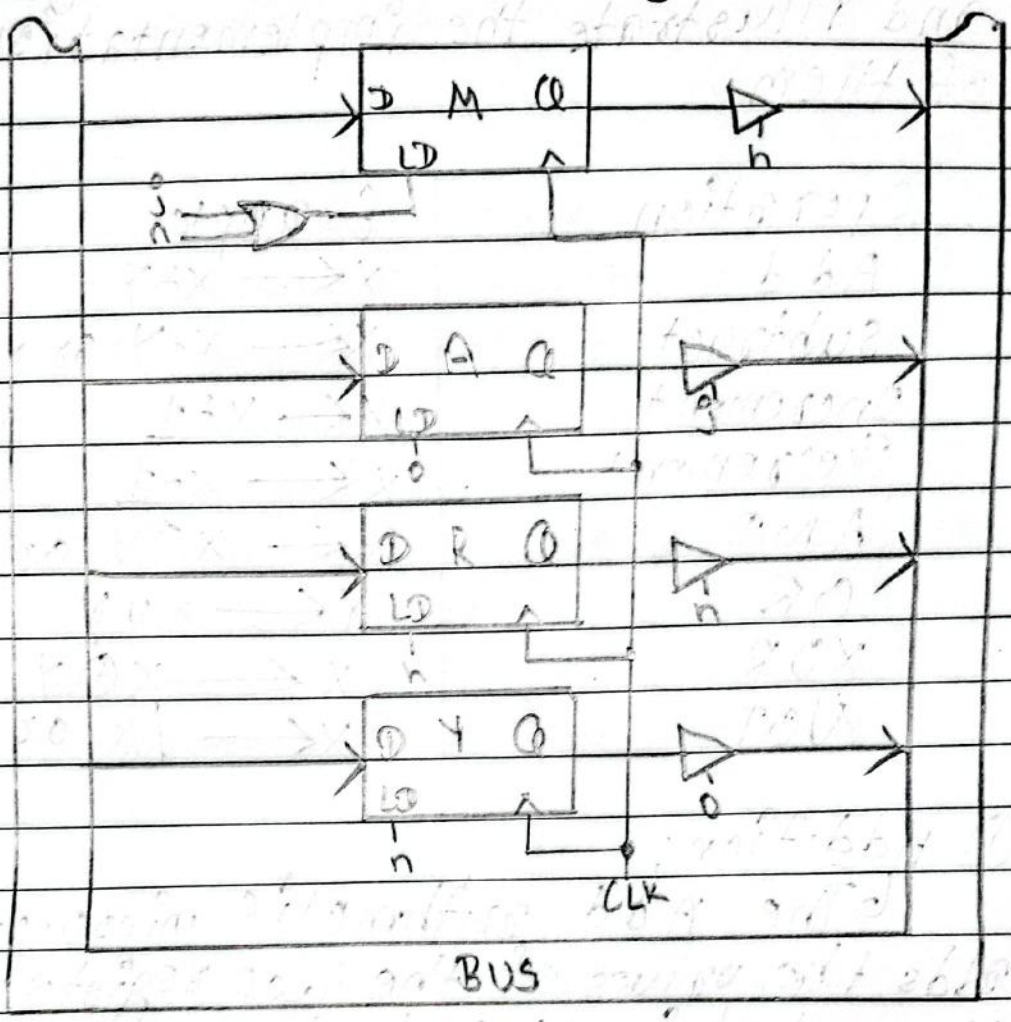
⇒ RTL (Register transfer language) is used to describe the micro-operations transfer among registers. It is a kind of intermediate representation (IR) that is very close to assembly language. Such as that which is used in a compiler. The term "Register Transfer" can perform micro operations and transfer the result of operation to the same or other register.

→ Implementation of RTL:
Example;
Assume that conditions j, o, h and n are mutually exclusive.
j: M ← A; o: A ← Y; h: R ← M;
n: Y ← R, M ← R

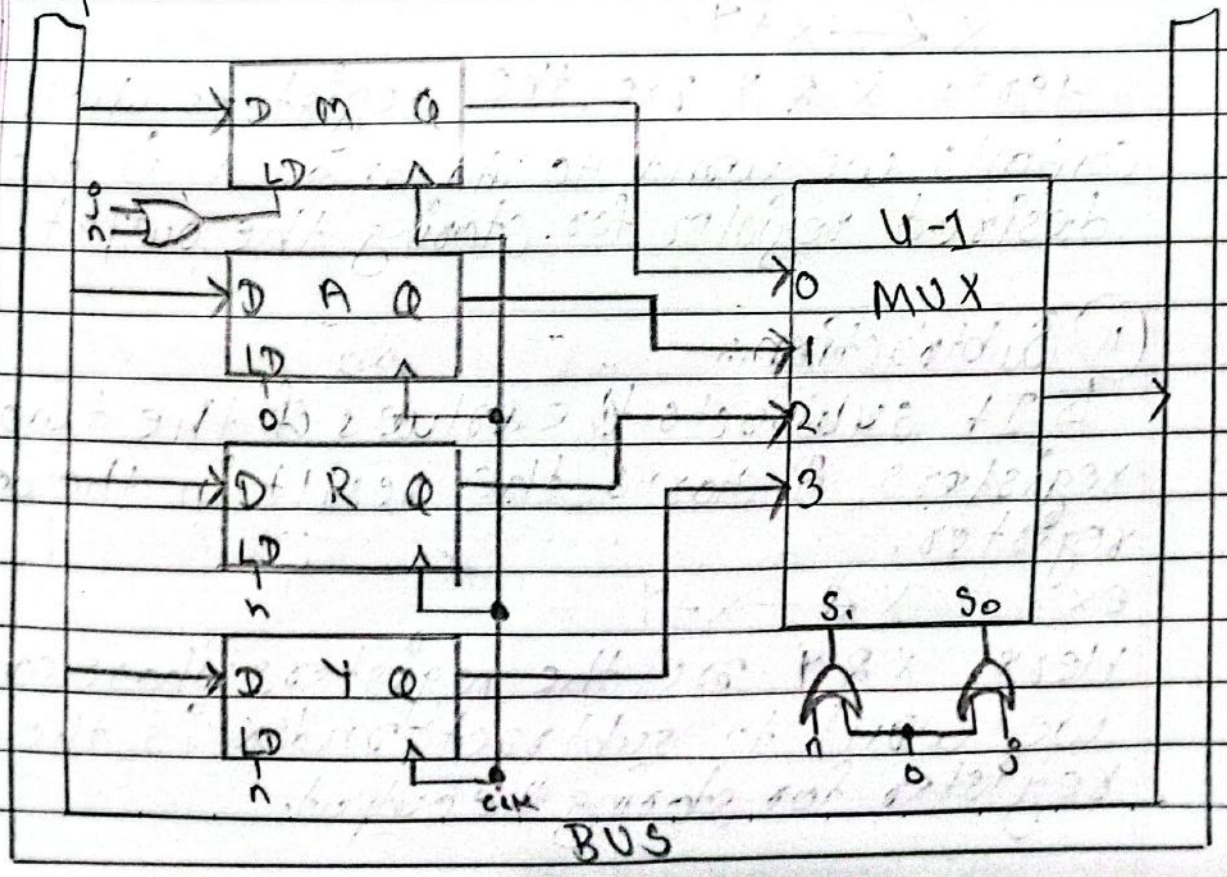
(a) Implement RTL code using direct connections



(b) Implement the RTL code using a bus & tri-state buffer.



(c) Implement the RTL code using a bus & MUX.



26) List out the arithmetic & logical micro-operations and illustrate the implementation of each of them.

⇒

Operation	Example
Add	$X \leftarrow X+Y$
Subtract	$X \leftarrow X-Y$ or $X \leftarrow X+Y'+1$
Increment	$X \leftarrow X+1$
Decrement	$X \leftarrow X-1$
AND	$X \leftarrow X \wedge Y$ or $X \leftarrow XY$
OR	$X \leftarrow X \vee Y$
XOR	$X \leftarrow X \oplus Y$
NOT	$X \leftarrow X'$ or $X \leftarrow X'$

① Addition:

↳ The Add arithmetic micro-operation adds the values of the two registers and stores the output in the desired register.

Ex:-

$X \leftarrow X+Y$

Here, X & Y are the registers whose contents we want to add and, X is the desired register for storing the output.

(11) Subtraction:

↳ It subtracts the values of the two registers & stores the result in the desired register.

ex:- $X \leftarrow X-Y$

Here, X & Y are the registers whose contents we want to subtract and X is the desired register for storing the output.

(3) Increment:
↳ It increments the value of a register by 1.

Ex:-
X ← X+1

Here, we incremented the value of X register and stored in X register.

(4) Decrement:
↳ It decrements the value of a register by 1.

Ex:-
X ← X-1

Here, we decrement the value of X register & store in X.

(5) AND:
↳ It multiplies the contents of given registers (logically And).

Ex:-
X ← X^Y

Here, the contents of X & Y registers are logically Anded and stored in X.

(6) OR
↳ It adds (logically OR) the contents of given registers.

Ex:-
X ← X v Y

(7) XOR:
↳ Exclusive OR

Ex:- X ← X ⊕ Y

29) Explain different types of shift microoperations with example.

⇒

Operation	Notation
Linear shift left	shl(x)
Linear shift right	shr(x)
Circular shift left	cil(x)
circular shift right	cir(x)
Arithmetic shift left	ashl(x)
Arithmetic shift right	ashr(x)
Decimal shift left	dshl(x)
Decimal shift right	dshr(x)

⇒ 1) Linear shift:

It transfers the 0 zero through the serial input. We use the symbols ~~[illegible]~~ shl for the ~~logical~~ linear left shift & shr(x) for the logical right shift.

a) linear shift left:

This left shift operator is denoted by the ~~double le~~ shl(x).

Ex:-

MSB LSB

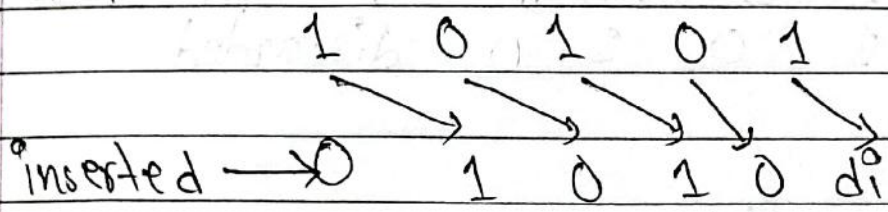
1 0 1 0 1

↙ ↙ ↙ ↙ ↙

discarded 0 1 0 1 0 ← inserted

b) linear shift right
↳ The right shift operator is denoted by shr(x).

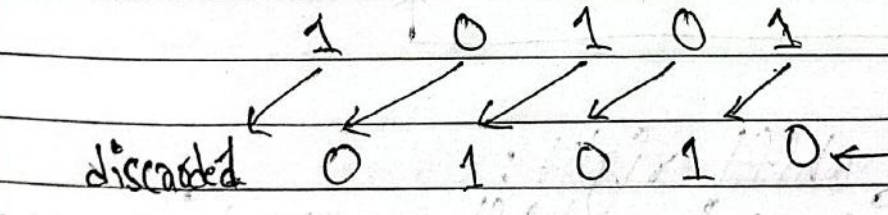
Ex:-



2) Arithmetic shift:-
The arithmetic shift micro-operation moves the signed binary number either to the left or to the right position.

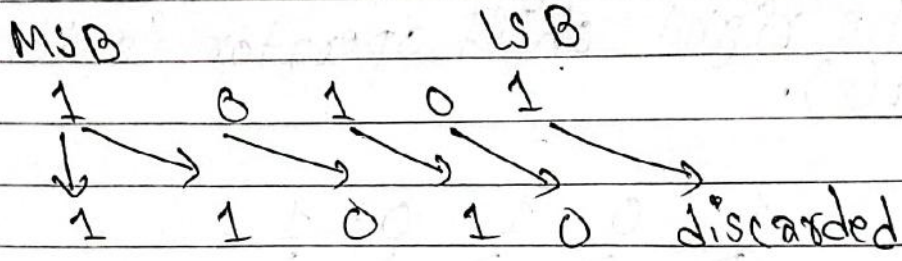
a) Arithmetic left shift:-
↳ In this shift, each bit is moved to the left one by one. The empty least significant bit (LSB) is filled with zero and the most significant bit (MSB) is rejected. Same as the left logical shift.

Ex:- MSB LSB



b) Arithmetic right shift:-
↳ In this shift, each bit is moved to the right one by one and the least significant (LSB) bit is rejected and the empty most significant bit (MSB) is filled with the value of the previous MSB.

Ex:



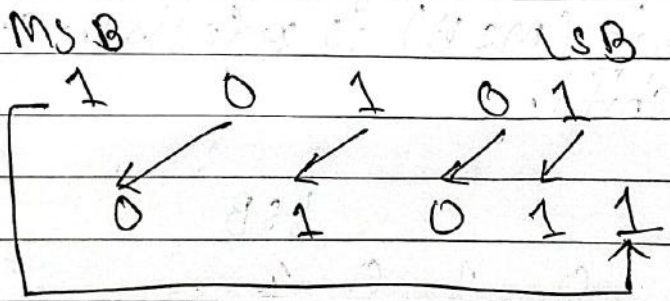
2) Circular shift:

↳ The circular shift circulates the bits in the sequence of the register around both ends without any loss of information.

a) Circular shift left:

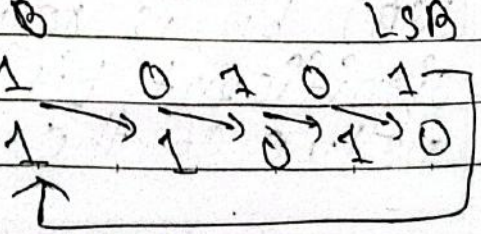
↳ In this micro shift operation each bit in the register is shifted to the left one by one. After shifting, the LSB becomes empty, so the value of the MSB is filled in there.

Ex:



b) Circular shift right:

↳ In this micro shift operation each bit in the register is shifted to the right one by one. After shifting, the MSB becomes empty, so the value of the LSB is filled in there. Ex: MSB LSB



30) Perform eight different shift operations on X = 1101010101110.

⇒ (i) linear shift left:

1 1 0 1 0 1 0 1 0 1 1 1 0

1 0 1 0 1 0 1 0 1 1 1 0 0

(ii) linear shift right:

= 0 1 1 0 1 0 1 0 1 0 1 1

(iii) Arithmetic left shift:

= 1 0 1 0 1 0 1 0 1 1 0 0

(iv) Arithmetic Right shift:

= 1 1 1 0 1 0 1 0 1 0 1 1

(v) Circular left shift:

= 1 0 1 0 1 0 1 0 1 1 0 1

(vi) Circular right shift:

= 0 1 1 0 1 0 1 0 1 0 1 1

(vii) Decimal shift left:

= 1 0 1 0 1 0 1 0 1 1 0 0

(viii) Decimal shift right:

= 1 1 0 1 0 1 0 1 0 1 1

31) Repeat the above on 10101100.

⇒ (i) linear shift left:

= 0 1 0 1 1 0 0 0

(ii) linear right shift:

= 0 1 0 1 0 1 1 0

(iii) Arithmetic left shift:

= 0 1 0 1 1 0 0 0

(iv) Arithmetic Right shift:
= 11010110

(v) Circular left shift:
= 01011001

(vi) Circular right shift:
= 00101110

(vii) Decimal left shift:
= 01011000 0

(viii) Decimal Right shift:
= 101011 0

32) Write about VHDL code and explain different sections of VHDL with its advantages and disadvantages.

⇒ VHDL (Very high speed integrated circuit (VHSIC) Hardware Description language)

↳ VHDL was developed to provide a standard for designing digital systems.
↳ VHDL specifies a formal syntax. The designer creates a design file using that syntax just as a programmer writes a C-program.
↳ The designer then synthesizes the design using only design package that can accept VHDL files. This is equivalent to the programmer compiling the C-Program. It checks for errors in syntax and declaration, but not in logic.
↳ Finally, the designer debugs the design using simulation tools.

→ Advantages of VHDL:

(a) PORTABILITY:

Just as a valid C-program can be compiled by any compliant C compiler, a VHDL design can be synthesized by any design system that supports VHDL.

(b) Device Independent:

The VHDL file is device Independent. The same file can be used to implement the design on a custom ICs, on ASICs or any PLD that is capable of containing the design.

(c) SIMULATION:

VHDL designs can be simulated by the design system, allowing the designer to verify the design performance before committing it to hardware.

(d) SYSTEM SPECIFICATION:

The designer can design the system using a high level of abstraction, such as a finite state machine, down to a low level digital logic implementation.

→ Disadvantages of VHDL:

↳ VHDL source code often becomes long and difficult to follow, especially at a low level of abstraction.

↳ Different design tools may produce different valid design for the same system, espically for high level of abstraction, providing no details about the implementation.

33) Write down VHDL code for following:

⇒ a) AND gate:

```
-- Header file declaration
library IEEE;
use IEEE.std_logic_1164.all;

-- Entity declaration

entity andGate is
    port( A : in std_logic; --A input
          B: in std_logic; -- B input
          Y: out std_logic; -- Output
end andGate;

architecture andLogic of andGate is

begin
        Y <= A AND B;

end andLogic;
```

⇒ "--" is used to comment.

b) OR gate:

→
```
library IEEE;
use IEEE.std_logic_1164.all

entity OrGate is

    port(A: in std_logic;
         B: in std_logic;
         Y: out std_logic);
end OrGate;

architecture orLogic of OrGate is
 begin
        Y <= A OR B;
 end orLogic;
```

c) NOT gate:

→
```
library IEEE;
use IEEE.std_logic_1164.all;
entity not_gate is
     port (A: in std_logic;
           Y: out std_logic);
     end not_gate;
architecture notLogic of not_gate is
begin
        Y <= not(A);
        end notlogic;
```

d) NOR Gate:

```
library Ieee;
use Ieee.std_logic_1164.all;
entity nor_gate is
port(A: in std_logic;
     B: in std_logic;
     Y: out std_logic);
end nor_gate;
architecture norlogic of no_gate is
begin
     Y <= not(A OR B)
end norlogic;
```

e) NAND Gate:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity nand_gate is
  port (A: in std_logic;
        B: in std_logic;
        Y: out std_logic;
end nand_gate;
architecture nandlogic of nand_gate is
begin
     Y <= not (A and B);
end nandlogic;
```

f) XOR Gate:

→
```
library IEEE;
use IEEE.std_logic_1164.all;
entity XOR_gate is
port(A: in std_logic;
     B: in std_logic;
     Y: out std_logic);
end XOR_gate;
architecture XORlogic of XOR_gate is
begin
     Y<=A XOR B;
end XORlogic;
```

g) XNOR Gate:

→
```
library IEEE;
use IEEE.std_logic_1164.all;
entity xnor_gate is
 port(A: in std_logic;
      B: in std_logic;
      Y: out std_logic);
end xnor_gate;
architecture xnorlogic of xnor_gate is
begin
     Y<= not (A xor B);
     end xnorlogic;
```

h) Half adder:

→
```
library IEEE;
use IEEE.std_logic_1164.all;
entity H_adder is
 port (A: in std_logic;
       B: in std_logic;
       Sum: out std_logic;
       Carryout: out std_logic);
end H_adder;
architecture flow of H_adder is
begin
    sum <= A xor B;
    carryout <= A and B;
end flow;
```

i) full adder:

→
```
library IEEE;
use IEE.std_logic_1164.all;
entity full_adder is
port (A, B, carry_in: in std_logic;
      Sum, carr_out: out std_logic );
end full_adder;
architecture flow of full_adder is
component half_adder is
 port(A, B: in std_logic;
      Sum, carryout: out std_logic);
 end component;
component or_gate is
port(A,B: in std_logic;
Y: out std_logic);
```

```
end component;
signal T1, T2, T3: std_logic;
begin
half_adder1: half_adder port map
    (A => A, B => B, sum => T1, carryout => T2);
half_adder2: half_adder port map
    (A => T1, B => carry_in, sum => sum,
     carryout => T3);
or_gate1: or_gate port map (A => T3, B => T2,
          Y => carry_out);
end flow;
```

j) Half subtractor:

```
→ library IEEE;
use IEEE.std_logic_1164.all;
entity half_sub is
  port (A,B: in std_logic;
        diff, borrow: out std_logic);
end half_sub;
architecture flow of half_sub is
begin
    diff <= A xor B;
    borrow <= not(A) and B;
    end flow;
```

k) Full Subtractor:

→
```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_sub is
  port(A, B, borrow_in: in std_logic;
   diff, borrow_out: out std_logic);

end full_sub;
architecture flow of full_sub is
component half_sub is
port(A, B: in std_logic;
      diff, borrow: out std_logic);
end component;

Component or_gate is
port(A, B: in std_logic;
      Y: out std_logic);
end component;
signal T1, T2, T3: std_logic;
begin
half_sub1: half_sub port map
          (A => A, B => B, diff => T1, borrow => T2);
half_sub2: half_sub port map
          (A => T1, B => borrow_in, diff => diff,
          borrow => T3);
or_gate1: or_gate port map(A => T3, B => T2,
          Y => borrow_out);
end flow;
```

1) 4×1 multiplexer:

→
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;
entity mux_tb
use ieee.std_logic_unsigned.all;

entity MUX_SOURCE is
    Port (S: in std_logic_vector (1 downto 0);
          I: in std_logic_vector (3 downto 0);
          Y: out std_logic);
end MUX_SOURCE

architecture Behavioral of MUX_SOURCE is
begin
      process (S, I)
begin
   if (S <= "00") then
       Y <= I(0);
  elsif (S <= "01") then
  Y <= I(1);
  elsif (S <= "10") then
   Y <= I(2);
   else
    Y <= I(3);

end if;
end process;
end Behavioral;
```

m) 1x4 demultiplexer:

→
```
library IEEE;
use IEEE.std_logic_1164.all;

entity Demultiplexer1x4 is
port(
    A: in std_logic;
    S: in std_logic_vector(1 downto 0);
    F, G, H, I: out std_logic);
end entity Demultiplexer1x4;

architecture Behavioral of Demultiplexer1x4 is
begin
    with S select
        F <= A when "00",
             '0' when others;
    with S select
        G <= A when "01",
             '0' when others;
    with S select
        H <= A when "10",
             '0' when others;
    with S select
        H <= A when "10",
             '0' when others;
    with S select
        I <= A when "11",
             '0' when others;

end architecture Behavioral;
```

1) 8x3 encoder:

→
```
library ieee;
use ieee.std_logic_1164.all;

entity Encoder8x3 is
  port (
        A,B,C,D,E,F,G,H: in std_logic;
        Y: out std_logic_vector(2 downto 0));
end entity Encoder8x3;

architecture Behavioral of Encoder8x3 is
begin
   Y<="000" when A = '1' else
      "001" when B = '1' else
      "010" when C = '1' else
      "011" when D = '1' else
      "100" when E = '1' else
      "101" when F = '1' else
      "110" when G = '1' else
      "111" when H = '1'
      "000";
end architecture Behavioral;
```

0) 3x8 decoder:

→
```
library ieee;
use ieee.std_logic_1164.all;

entity Decoder3x8 is
    port(
        A, B, C : in std_logic;
        Y : out std_logic_vector(7 downto 0));
end entity Decoder3x8;

architecture Behavioral of Decoder3x8 is
begin
```

~~Y <= "00000001"~~	Y <= "00000001" when (A='0' and
~~"00000010"~~	B='0' and C='0') else
~~"00000011"~~	
~~"00000100"~~	"00000010" when(A='0'
~~"00000101"~~	and B='0' and C='1') else
~~"00000110"~~	
~~"00000~~	"00000100" when(A='0'
~~"~~	and B='1' and C='0') else

```
"00001000" when(A='0' and B='1' and C='1') else
"00010000" when(A='1' and B='0' and C='0') else
"00100000" when(A='1' and B='0' and C='1') else
"01000000" when(A='1' and B='1' and C='0') else
"10000000" when(A='1' and B='1' and C='1') else
"00000000";
end architecture Behavioral;
```

p) SR flipflop:

→
```
library ieee;
use ieee.std_logic_1164.all;

entity SR_Flipflop is
  port (
        S, R: in std_logic;
        Q, Qbar: out std_logic);
end entity SR_Flipflop;

architecture Behavioral of SR_flipflop is
signal Q_int, Qbar_int: std_logic;
begin
    process (S, R)
    begin
        if (R = '1') then
            Q_int <= '0';
            Qbar_int <= '1';
        elsif (S = '1') then
            Q_int <= '1';
            Qbar_int <= '0';
        endif;
    end process;

    Q <= Q_int;
    Qbar <= Qbar_int;

end architecture Behavioral;
```

q) JK Flipflop:

→
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity JK_FF is
PORT (J, K, CLOCK: in std_logic;
Q, QB: out std_logic);
end JK_FF;

architecture behavioral of JK_FF is
begin
PROCESS (CLOCK)
variable TMP: std_logic;
begin
  if (CLOCK='1' and CLOCK'EVENT) then
   if (J='0' and K='0') then
    TMP:=TMP;
  elsif (J='1' and K='1') then
    TMP:= not TMP;
  elsif (J='0' and K='1') then
    TMP:='0';
  else
    TMP:='1';
  endif;
  end if;
   Q<=TMP;
   Q<= not TMP;
  end process;
  end behavioral;
```

8) T flipflop:

→
```
library ieee;
use ieee.std_logic_1164.all;

entity T_FF is
port(T: in std_logic;
clock: in std_logic;
Q: out std_logic);
end T_FF;

architecture behavioral of T_FF is
signal tmp: std_logic;
begin
process(clock)
begin
if Clock'event and clock='1' then
if T='0' then
tmp <= tmp
elsif T='1' then
tmp <= not(tmp);
end if;
end if;
end process;
Q<tmp;
end Behavioral;
```

5) D FlipFlop:

→
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity D_FF is
PORT (D, CLOCK: in std_logic;
Q: out std_logic);
end D_FF;

architecture behavioral of D_FF is
begin
process(CLOCK)
begin
if(CLOCK='1' and CLOCK'EVENT) then
Q<=D;
end if;
end process;
end behavioral;
```

34) Write down the VHDL code for MODULO 6 counter using a low level of abstraction.

⇒
```
library ieee;
use ieee.std_logic_1164.all;

entity mod6 is
    port (
        v, clk: in std_logic;
        q2, q1, q0: buffer std_logic;
        v2, v1, v0, c: out std_logic);
end mod 6;

architecture amod6 of mod6 is
begin
    cct_mod6: process (q2, q1, q0, v, clk)
    begin
        if rising_edge(clk) then
q2 <= (q2 and (not q1) and (not q0)) or
      (q2 and (not q1) and (not v)) or
      ((not q2) and q1 and q0 and v);

q1 <= ((not q2) and q1 and (not q0)) or
      ((not q2) and q1 and (not v)) or
      ((not q2) and (not q1) and q0 and v);

q0 <= ((not q2) and (not q0) and v) or
      ((not q1) and (not q0) and v) or
      ((not q2) and q0 and (not v)) or
      ((not q1) and q0 and (not v));
end if;
v2 <= q2;
```

```
    v1 <= q1;
    v0 <= (q2 and q1) or q0;
    c <= not (q2 or q1 or q0);
end process cct_mod6;
end amod6;
```

35) Write a VHDL code for the following combinational circuits.

a) $F = AB + A'B'$

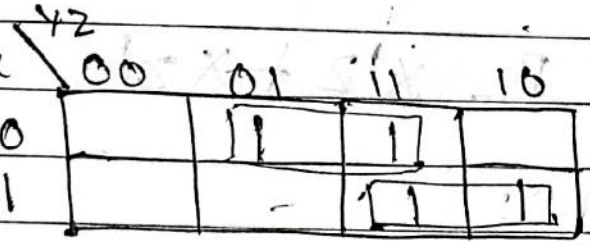
→
```
library ieee;
use ieee.std_logic_1164.all;

entity CombinationalCircuit1 is
    port (
        A,B: in std_logic;
        F : out std_logic);
end entity CombinationalCircuit1;

architecture Behavioral of CombinationalCircuit
is
begin
    F <= A and B or (not A and not B);
end architecture Behavioral;
```

(b) $F(x,y,z) = \Sigma(1,3,6,7)$ → sum of product
(c) $F(x,y,z) = \Sigma(1,3,4,6)$ → sum of product
(d) $F(x,y,z) = \Pi(0,2,5,7)$ → product of sum

Solution:

(b)

→

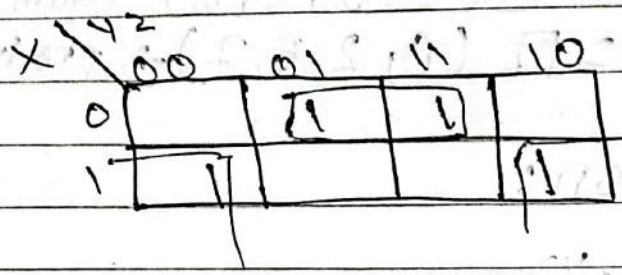


$F = x'z + xy$

VHDL code:

→
```
library ieee;
use ieee.std_logic_1164.all;

entity CombinationalCircuit2 is
port (
    A, B : in std_logic;
    F : out std_logic);
end entity CombinationalCircuit2;

architecture Behavioral of CombinationalCircuit2
is
begin
    F <= (not A and B) or (A and B);
end architecture Behavioral;
```

(c) Soln



$F = x'z + xz'$

VHDL code:

```
library ieee;
use ieee.std_logic_1164.all;

entity Circuit3 is
    port
        ( A,B : in std_logic;
          F : out std_logic);
end entity Circuit3;

architecture Behavioral of Circuit3 is
begin
    F <= (not A and B) or (A and not B);
end architecture Behavioral;
```

(d) Soln

x \ y2y1	00	01	11	10
0	0	1	1	0
1	1	0	0	1

$(x+z)(x'+z')$

VHDL code:

```
→ library ieee;
  use ieee.std_logic_1164.all;

entity circuit4 is
  port
    (A, B: in std_logic;
     F: out std_logic);
  end entity circuit4;

architecture Behavioral of circuit4 is
begin
  F <= (A or B) and (not A or not B);
end architecture Behavioral;
```

27) Draw the state diagram of the modulo 6 counter and implement it using registers.

⇒ State table for the modulo 6 counter:

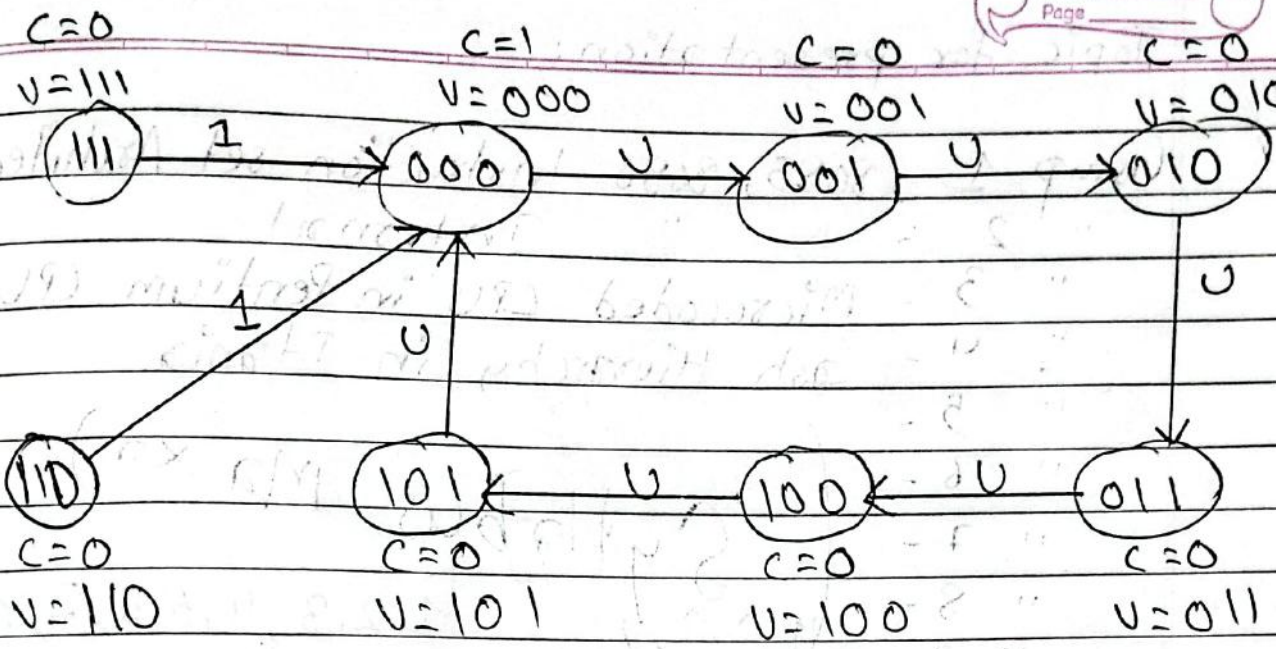
Present State	U	Next state	C	$U_2U_1U_0$
S_0	0	S_0	1	000
S_0	1	S_1	0	001
S_1	0	S_1	0	001
S_1	1	S_2	0	010
S_2	0	S_2	0	010
S_2	1	S_3	0	011
S_3	0	S_3	0	011
S_3	1	S_4	0	100
S_4	0	S_4	0	100
S_4	1	S_5	0	101
S_5	0	S_5	0	101
S_5	1	S_0	1	000
S_6	X	S_0	1	000
S_7	X	S_0	1	000

→ The behaviour of the modulo 6 counter can be expressed by:

$(S_0 + S_1 + S_2 + S_3 + S_4)U$: $V \leftarrow V+1$, $C \leftarrow 0$

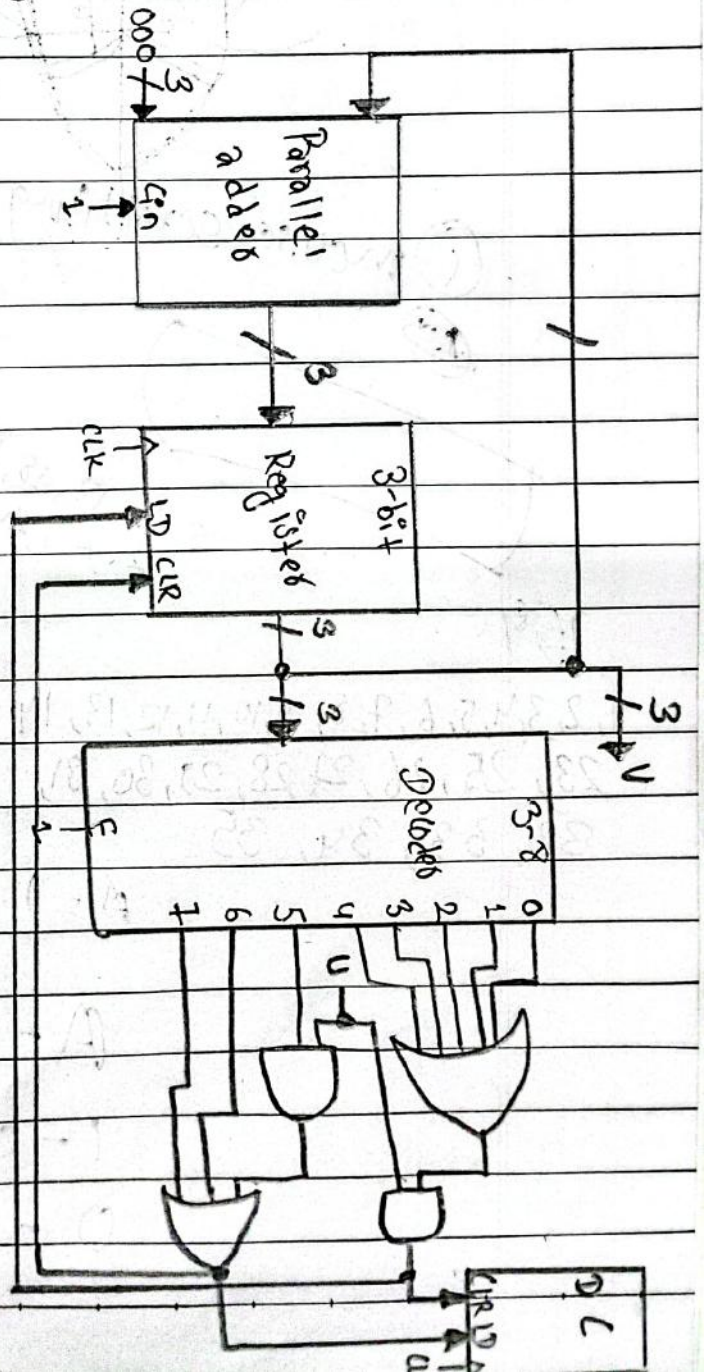
$S_5U + S_6 + S_7$: $V \leftarrow 0$, $C \leftarrow 1$

→ State diagram for the modulo 6 counter:



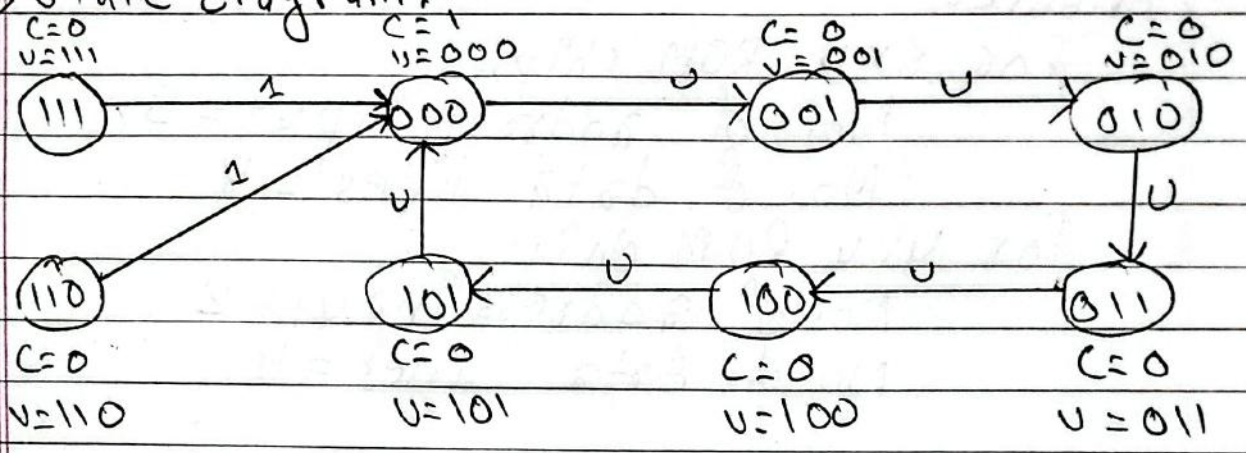
→ Implementation of the RTL code for the modulo 6 counter

(a) Using register:

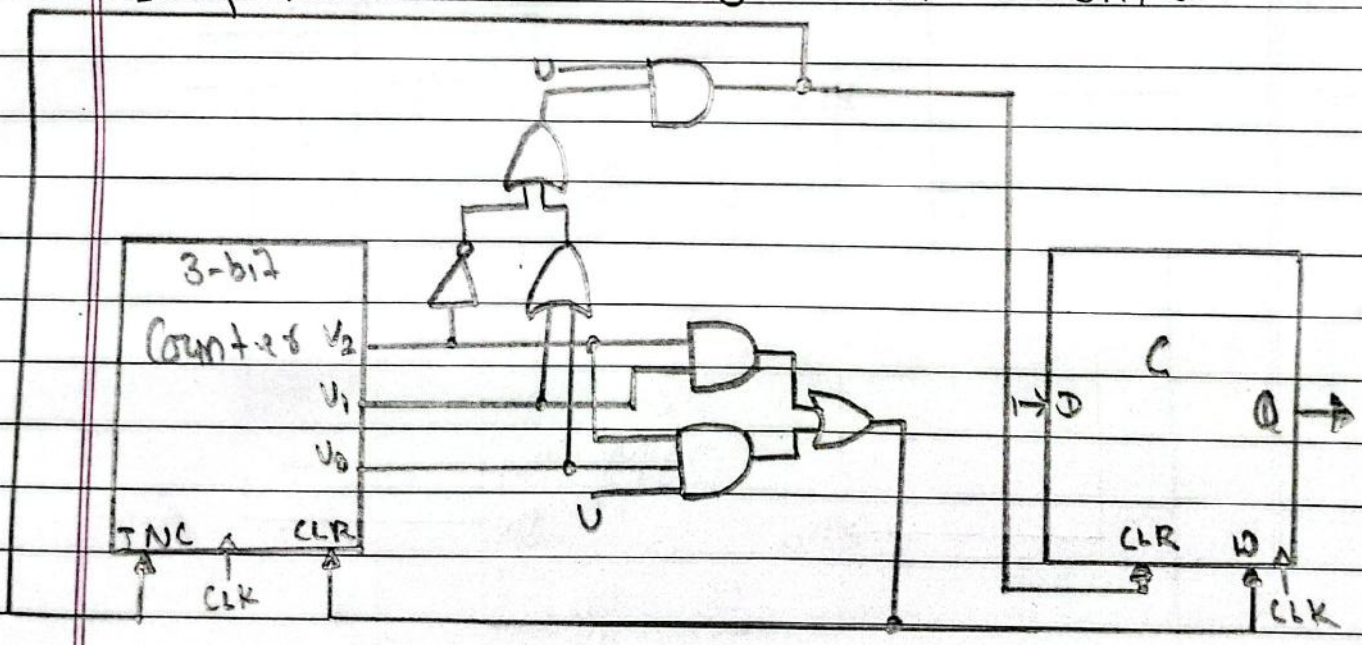


28) Draw the state diagram of the modulo 6 counter and implement it using a 3 bit counter.

⇒ State diagram:



→ Implementation using 3-bit counter:



23) Design an 8x4 ROM chip using 4x4 ROM chips. Illustrate using low level interleaving.

⇒ Answer:

For 8x4 ROM chip;

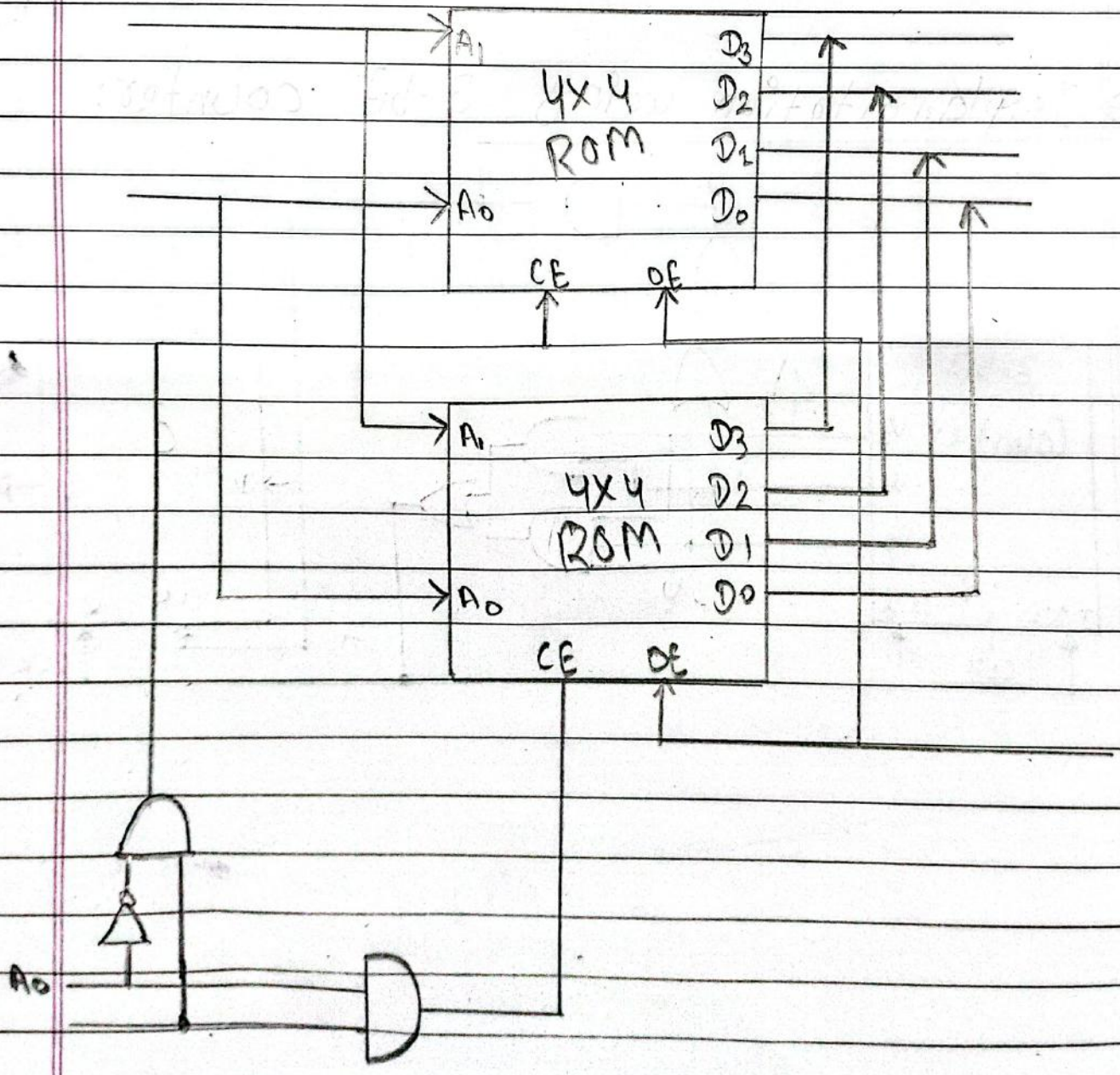
No. of address input = 3

No. of data lines = 4

For 4x4 ROM chip

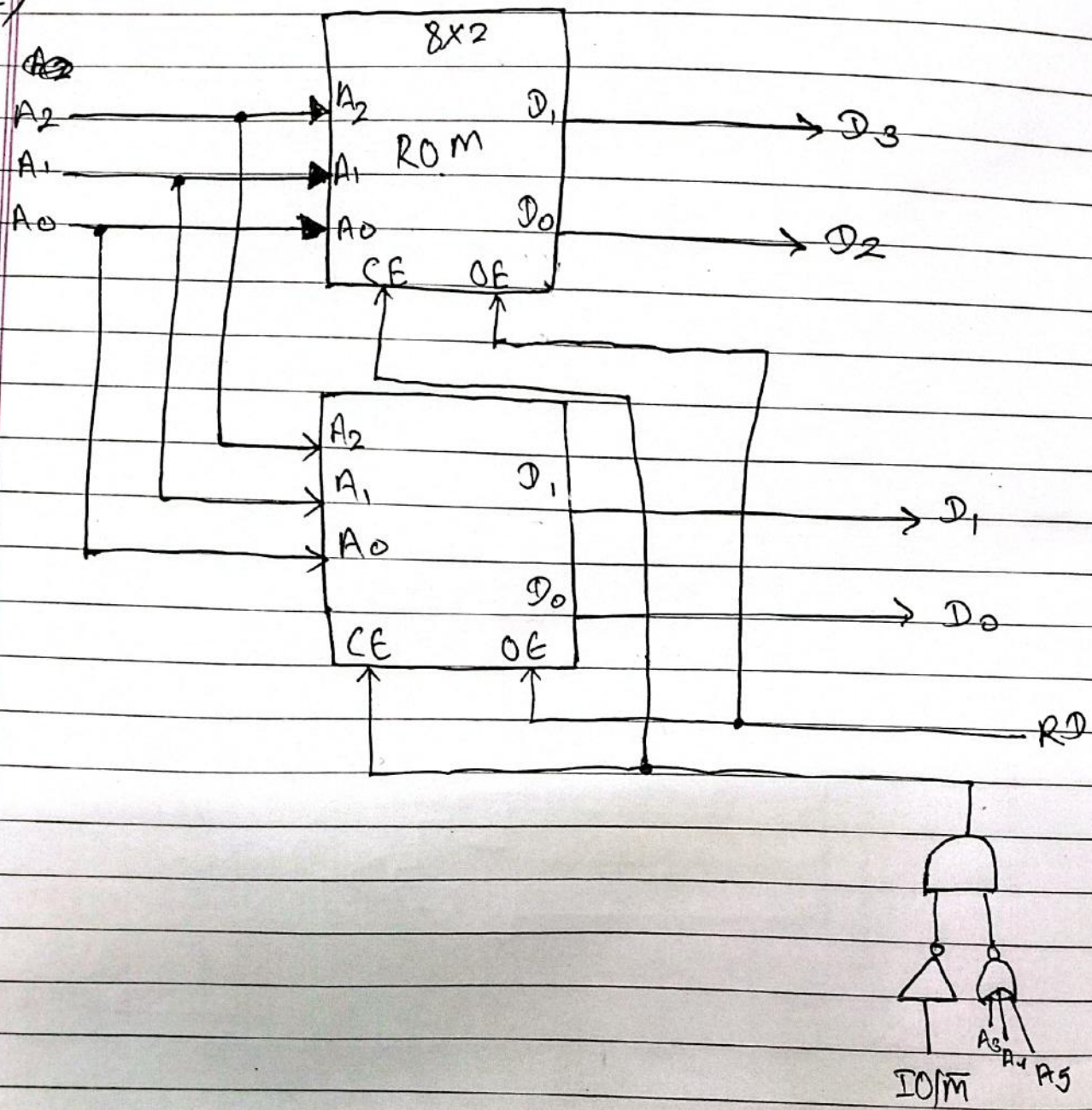
No. of address input = 2

No. of data lines = 4



10) Construct 8X4 memory subsystems using 8X2 ROM chips with required control signals.

⇒



9) Design two dimensional chip organization of 16 x 2 ROM chips.

⇒ Answer:

For 16x2 ROM chips,

No of address inputs = 4 (say $A_3 A_2 A_1 A_0$)

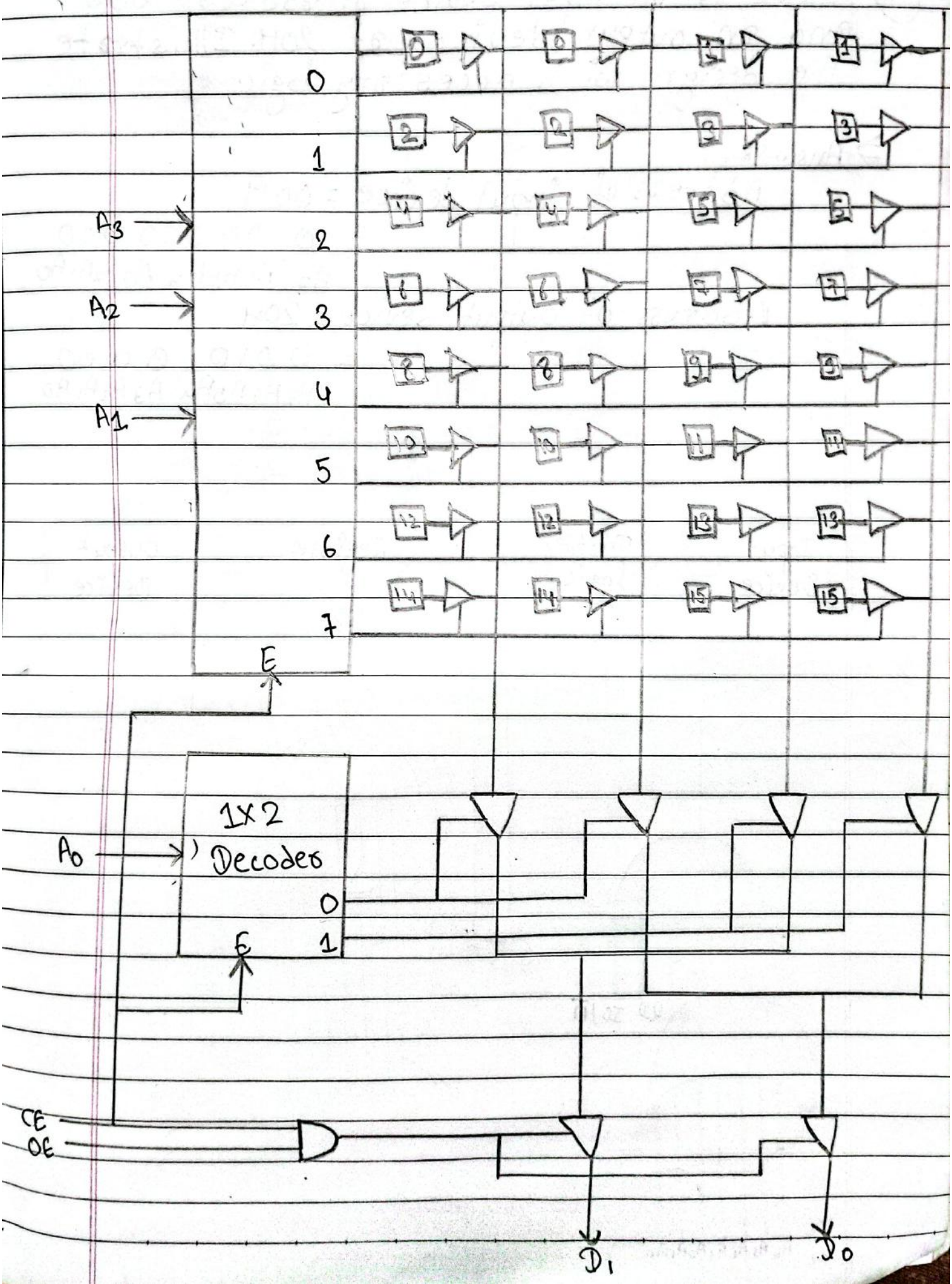
No of data lines = 2 (say D_0, D_1)

For 16x2 memory chip organization 2-D arrangement is given as:

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

↳ In the above arrangement, for 3 higher order bits the chip value is same and low order is 0,1 alternatively. Then,

9) contd--- (figure)



(11) Design 8X4 memory subsystem at address 80H constructed using 8x2 ROM.

→ Answer:

For 8X4 memory subsystem,
No. of Address input = 3
No. of data lines = 4

For 8X2 ROM,
No. of address input = 3
No. of data lines = 2

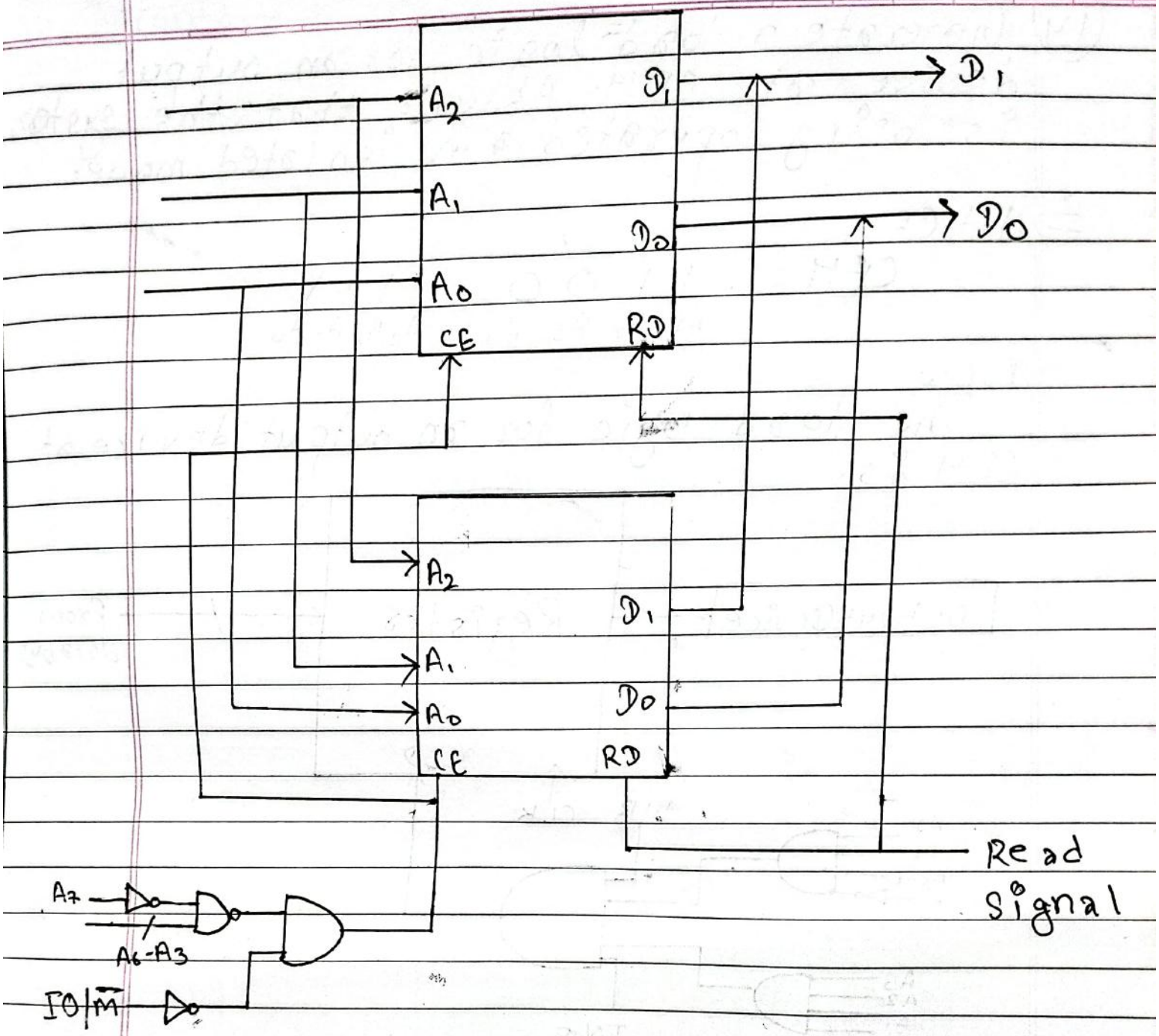
Then,
Starting Address given = 80H

so,

1 0 0 0 0 | 0 0 0

1 0 0 0 0 — used for CE logic

0 0 0 — used for the memory selection


A2
A1
A0
CE
D1
D0
RD
D1
D0
A2
A1
A0
CE
D1
D0
RD
Read signal
A7
A6-A3
IO/M

(14) Generate a load logic for an output device at CFH. Assume that the system is being operated in isolated mode.

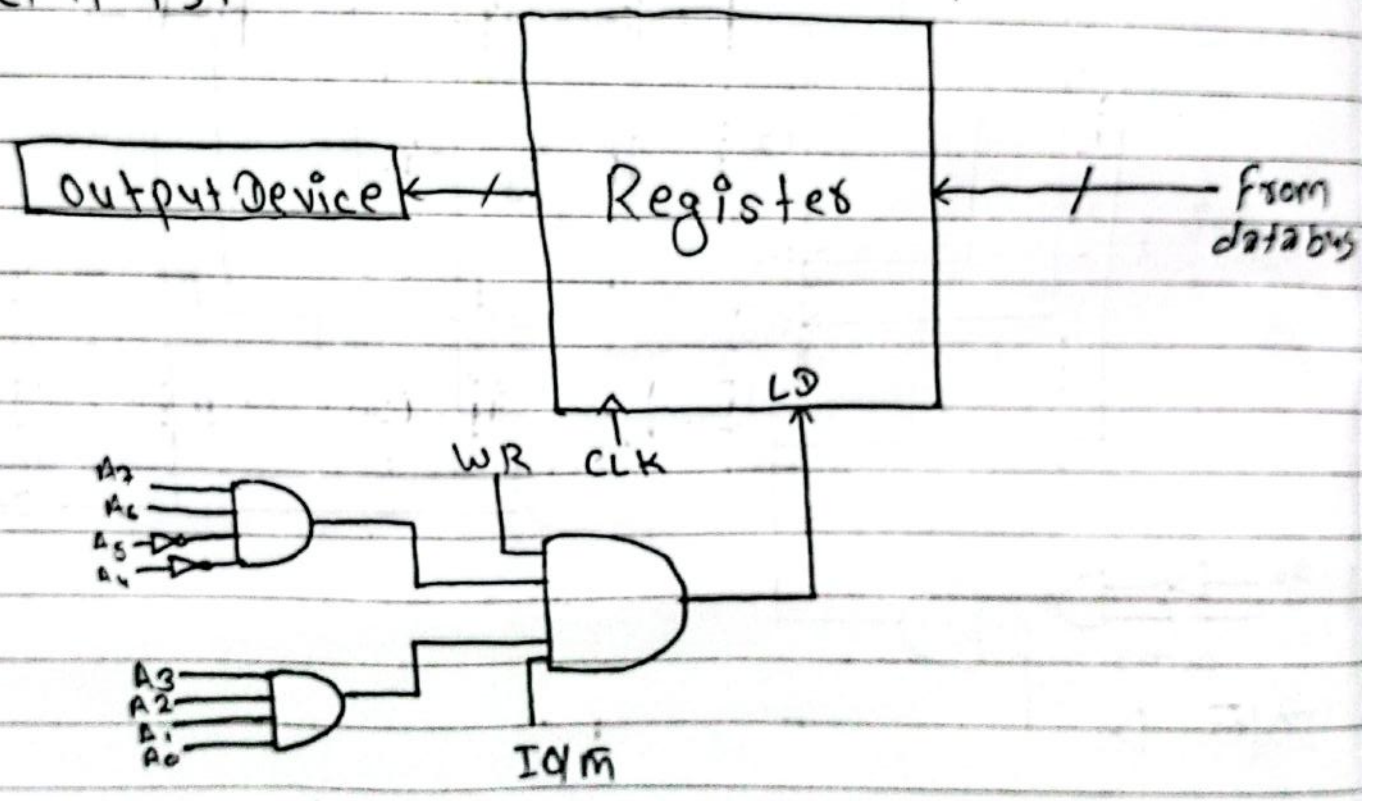
⇒ At CFH,

CFH : 1 1 0 0 1 1 1 1

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Now,

The load logic for an output device at CFH is:



(15) There is an input device at address 00H and an output device at 20H. Illustrate the design with necessary logic.

⇒ Answer:

Address of input device = 00H

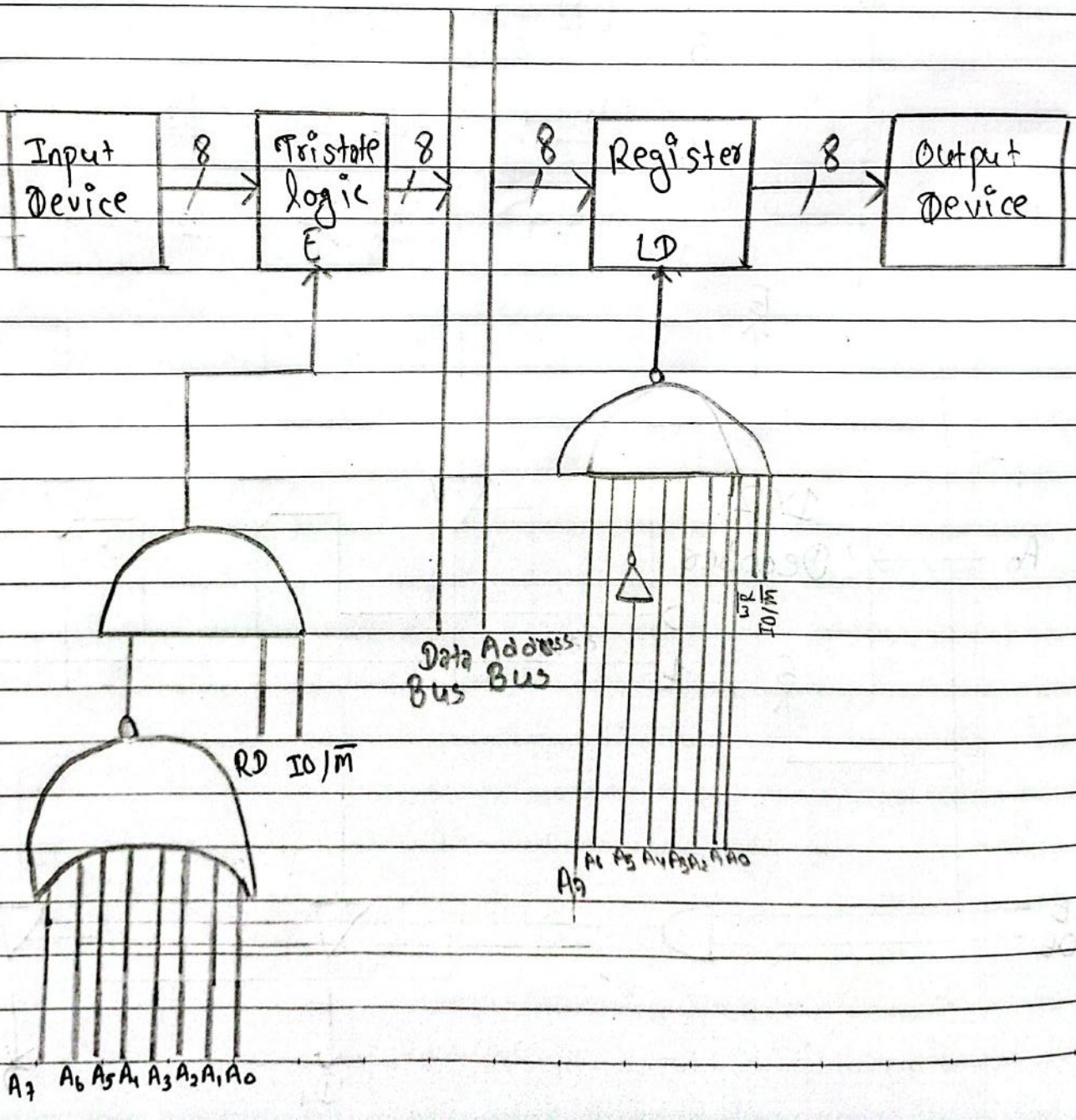
= 0 000 0 000

A_7 $A_6A_5A_4$ $A_3A_2A_1A_0$

Address of output device = 20H

= 0 010 0 000

A_7 $A_6A_5A_4$ $A_3A_2A_1A_0$



(16) A computer has a CPU with an 8 bit address bus and 16 bit data bus. The computer uses isolated I/O. It has 64x16 ROM at 00H constructed using two 32X16 ROM chips. It also has 32x16 of RAM at C0H. The system has an input device at 15H and the output device at 76H. Show the design for the required system including all necessary logic.

→ Answer:

↳ For 64x16 of ROM, (2^6 X16)

No. of address input = 6

No. of data lines = 16

00H = 0 0 0 0 0 0 0 0

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

→ For 32X16 ROM and RAM

No. of address input = 5

No. of data lines = 16

For RAM,

C0H = 1 1 0 0 0 0 0 0

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

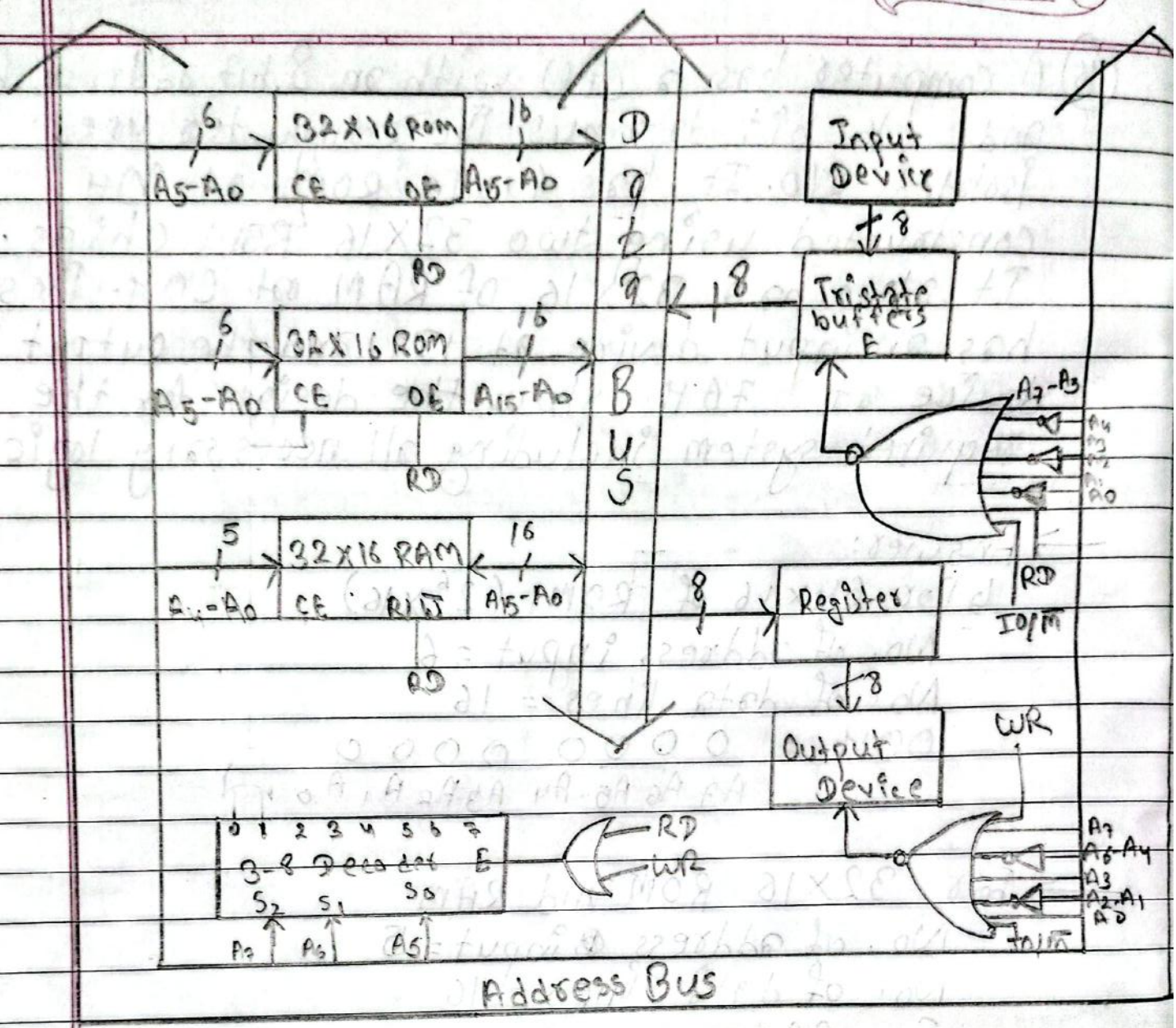
for an Input and Output Device,

For input device = 15H = [0 0 0 1 0 1 0 1]

For output device = 76H = [0 1 1 1 0 1 1 0]

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

(8)



Address Bus