



Computer Architecture

2024 – 2025 Academic Year

CEIT – 51023 Computer Architecture I

(Fifth Year – First Semester)

Dr. Theingi Myint

Professor

Department of Computer Engineering and Information Technology

Mandalay Technological University

Contents

Chapter (1): Digital Logic Circuits

Chapter (2): Digital Components

Chapter (4): Register Transfer and Microoperations

Chapter (6): Programming the Basic Computer

Chapter (8): Central Processing Unit

Chapter Two: Digital Components

- Explain the logic operation of the most common standard digital components such as decoders, multiplexers, registers, counters, and memories.
- Apply these digital components as building blocks for the design of larger units

Integrated Circuits

- Digital circuits are constructed with integrated circuits.
- An *integrated circuit* (IC) is a small silicon semiconductor crystal, called, a chip, containing the electronic components for the digital gates.
- The various gates are interconnected inside the chip to form the required circuit.
- The chip is mounted in a *ceramic or plastic container*, and connections are welded by thin gold wires to external pins to form the integrated circuit.
- The number of pins may range from 14 in a small IC package to 100 or more in a larger package.
- Each IC has a numeric designation printed on the surface of the package for identification.

Integrated Circuits

- The differentiation between chips that have a few internal gates and those having hundreds or thousands of gates is made by a customary reference to a package as being either a **small-, medium-, or large-scale** integration device.

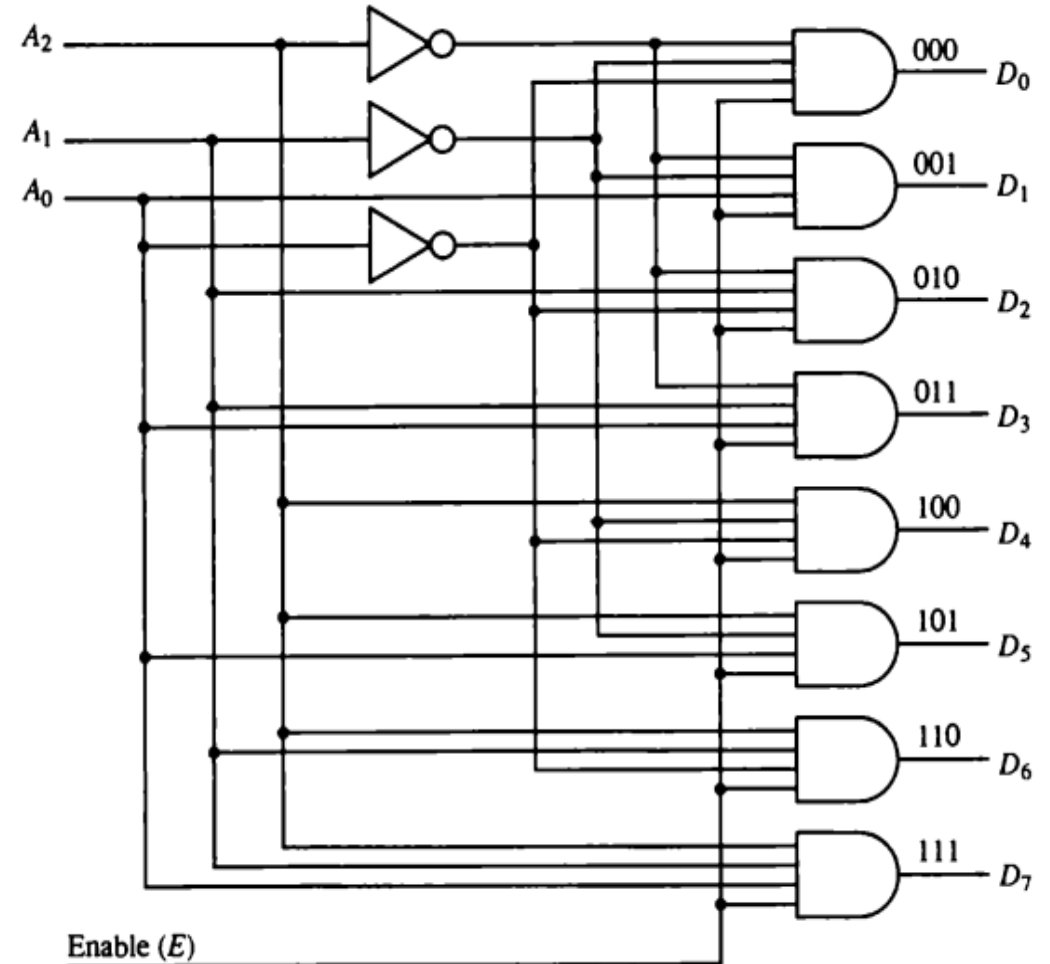
SSI (Small-scale integration)	MSI (Medium-scale integration)	LSI (Large-scale integration)	VLSI (Very-large-scale integration)
<ul style="list-style-type: none">- contain several independent gates in a single package- Inputs and outputs of the gates are connected directly to the pins in the package- number of gates (less than 10)	<ul style="list-style-type: none">- a complexity of approximately 10 to 200 gates in a single package- Perform specific elementary digital functions such as decoders, adders, and registers	<ul style="list-style-type: none">- Between 200 and a few thousand gates in a single package- Digital systems, such as processors, memory chips, and programmable modules	<ul style="list-style-type: none">- thousands of gates within a single package- large memory arrays and complex microcomputer chips- Small size and low cost, VLSI devices

Decoders

- A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs.
- If the n -bit coded information has unused bit combinations, the decoder may have less than 2^n outputs.
- The n -to- m -line decoders, where $m \leq 2^n$, is to generate the 2^n (or fewer) binary combinations of the n input variables.
- A decoder has n inputs and m outputs and is also referred to as an $n \times m$ decoder.

Decoders

- A particular application of this decoder is a **binary-to-octal** conversion.
- The **input** variables represent a **binary** number and the **outputs** represent the **eight digits** of the **octal number** system.
- A **3-to-8-line decoder** can be used for decoding any 3-bit code to provide eight outputs, one for each combination of the binary code.
- The decoder is **enabled** when E is equal to **1** and **disabled** when E is equal to **0**.



Decoders

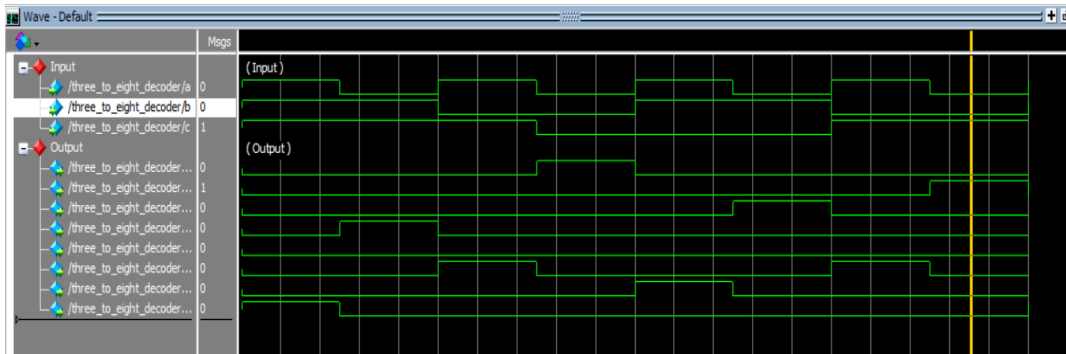
Enable	Inputs			Outputs							
<i>E</i>	<i>A</i> ₂	<i>A</i> ₁	<i>A</i> ₀	<i>D</i> ₇	<i>D</i> ₆	<i>D</i> ₅	<i>D</i> ₄	<i>D</i> ₃	<i>D</i> ₂	<i>D</i> ₁	<i>D</i> ₀
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Decoders

Verilog for 3-to-8 Decoder

```
module decoder(a,b,c,d0,d1,d2,d3,d4,d5,d6,d7);
    input a,b,c;
    output d0,d1,d2,d3,d4,d5,d6,d7;
    assign d0=(~a&~b&~c),
           d1=(~a&~b&c),
           d2=(~a&b&~c),
           d3=(~a&b&c),
           d4=(a&~b&~c),
           d5=(a&~b&c),
           d6=(a&b&~c),
           d7=(a&b&c);
endmodule
```

Simulation Results:



Testbench code for 3-to-8 Decoder

```
module testmodule;
    // Inputs
    reg a;
    reg b;
    reg c;
    // Outputs
    wire d0;
    wire d1;
    wire d2;
    wire d3;
    wire d4;
    wire d5;
    wire d6;
    wire d7;
    // Instantiate the Unit Under Test (UUT)
    decoder uut(.a(a),.b(b),.c(c),.d0(d0),.d1(d1),.d2(d2),.d3(d3),.d4(d4),.d5(d5),.d6(d6),.d7(d7));
    initial begin
        // Initialize Inputs
        a = 0;
        b = 0;
        c = 0;
        // Wait 100ns for global reset to finish
        #100;
        a = 1;
        b = 0;
        c = 1;
        // Wait 100ns for global reset to finish
        #100;
    end
endmodule
```

Encoders

- An encoder is a digital circuit that performs the inverse operation of a decoder.
- An encoder has 2^n (or less) input lines and n output lines.
- The output lines generate the binary code corresponding to the input value.
- The octal-to-binary encoder has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number.
- The encoder can be implemented with OR gates whose inputs are determined directly from the truth table.

Encoders

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

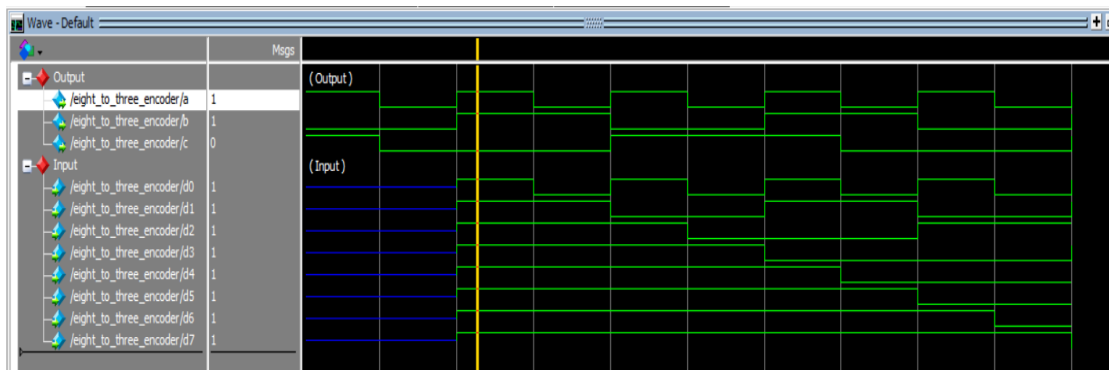
The encoder can be implemented with **three OR gates**.

Encoders

Verilog for 8-to-3 Encoder

```
module
encoder(d0,d1,d2,d3,d4,d5,d6,d7,a,b,c);
    input d0,d1,d2,d3,d4,d5,d6,d7;
    output a,b,c;
    or(a,d4,d5,d6,d7);
    or(b,d2,d3,d6,d7);
    or(c,d1,d3,d5,d7);
endmodule
```

Simulation Results:



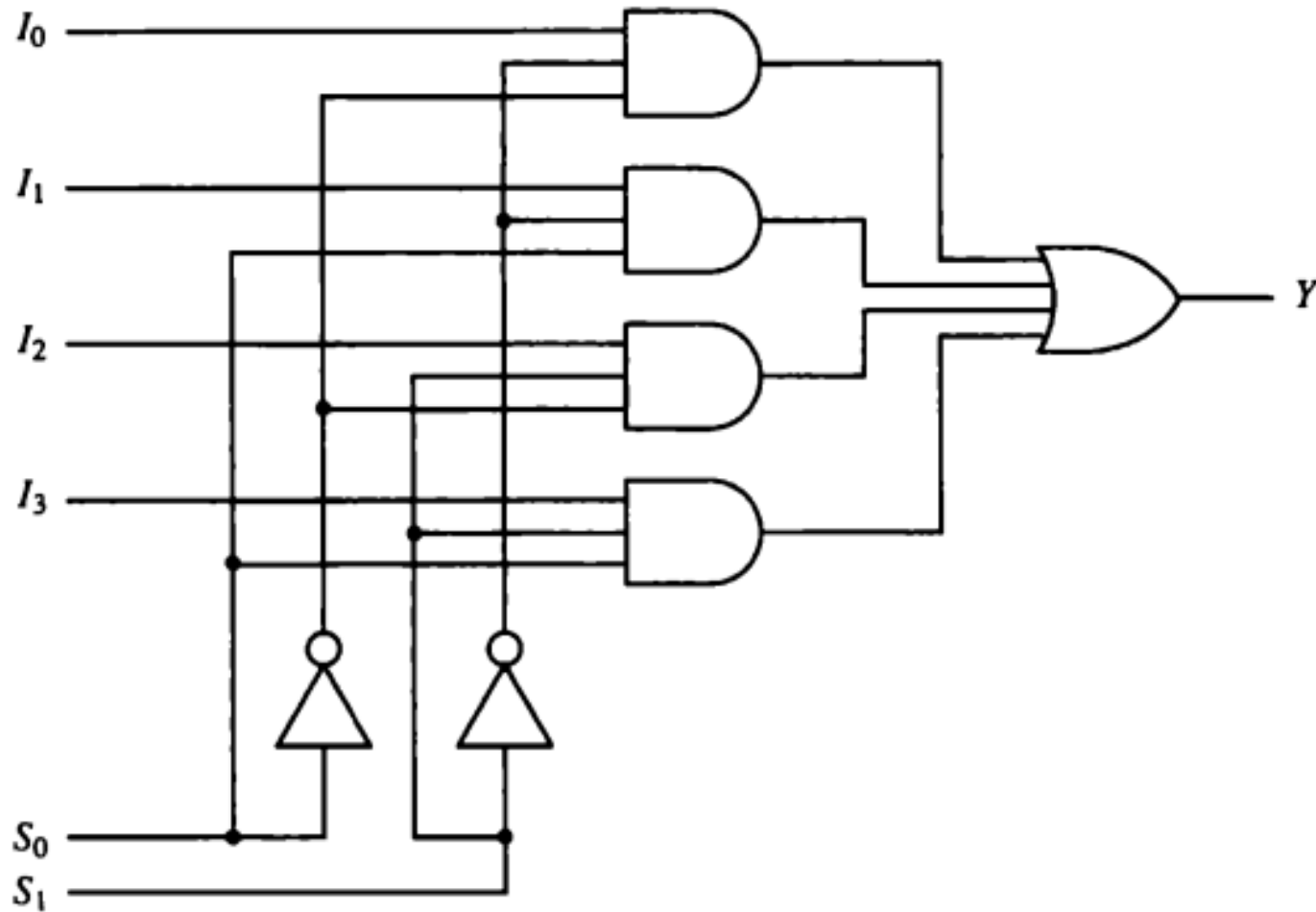
Testbench code for 8-to-3 Encoder

```
module testmodule;
    // Inputs
    reg d0; reg d1; reg d2; reg d3; reg d4; reg d5; reg d6; reg d7;
    // Outputs
    wire a; wire b; wire c;
    // Instantiate the Unit Under Test (UUT)
    encoder uut(.d0(d0),.d1(d1),.d2(d2),.d3(d3),.d4(d4),.d5(d5),.d6(d6),.d7(d7),.a(a),.b(b),.c(c));
    initial begin
        // Initialize Inputs
        d0 = 0;
        d1 = 0;
        d2 = 0;
        d3 = 0;
        d4 = 0;
        d5 = 0;
        d6 = 0;
        d7 = 0;
        // Wait 100ns for global reset to finish
        #100;
        d0 = 0;
        d1 = 0;
        d2 = 0;
        d3 = 1;
        d4 = 0;
        d5 = 0;
        d6 = 0;
        d7 = 0;
        // Wait 100ns for global reset to finish
        #100;
        // Add stimulus here
    end
endmodule
```


Multiplexers

- A multiplexer is a combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line.
- The selection of a particular input data line for the output is determined by a set of selection inputs.
- A 2^n -to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.
- In 4-to-1-line multiplexer, each of the four data inputs I_0 through I_3 is applied to one input of an AND gate.
- The two selection inputs S_1 and S_0 are decoded to select a particular AND gate.
- The outputs of the AND gates are applied to a single OR gate to provide the single output.

Multiplexers



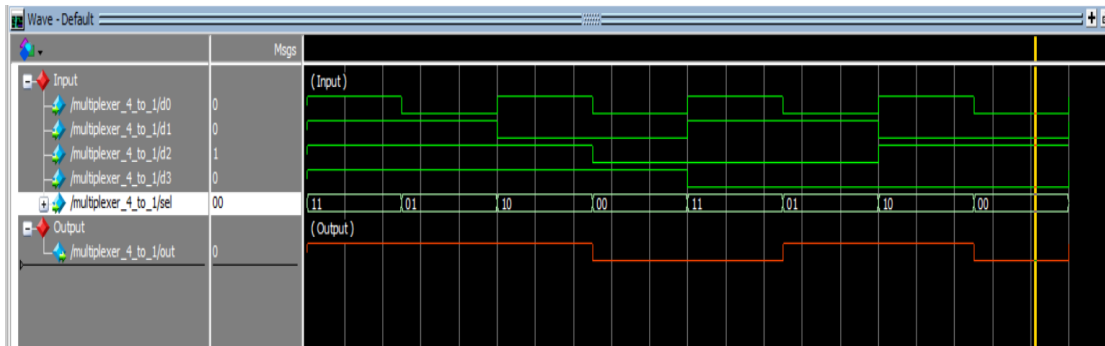
Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Multiplexers

Verilog for 4-to-1-line Multiplexer

```
module Multiplexer(d0,d1,d2,d3,sel,out);
    input d0,d1,d2,d3;
    input [1:0] sel;
    output reg out;
    always@(sel)
    begin
        case(sel)
            2'b00 : out <= d0;
            2'b01 : out <= d1;
            2'b10 : out <= d2;
            2'b11 : out <= d3;
        endcase
    end
endmodule
```

Simulation Results:



Testbench code for 4-to-1-Line Multiplexer

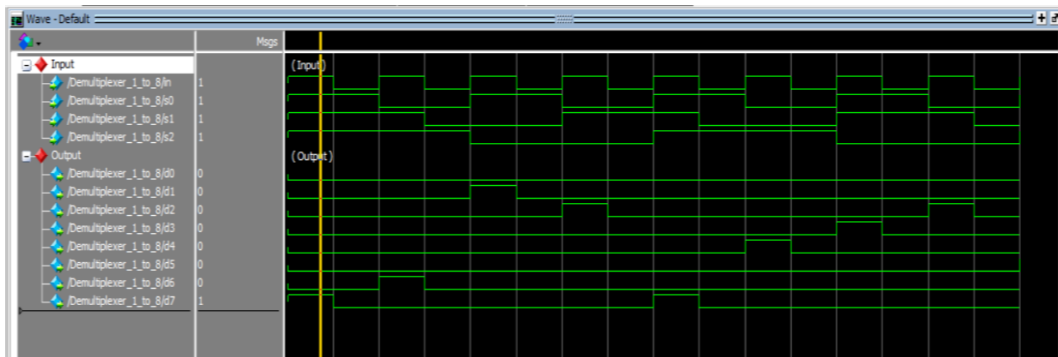
```
module TestModule;
    // Inputs
    reg d0; reg d1; reg d2; reg d3; reg [1:0] sel;
    // Outputs
    wire out;
    // Instantiate the Unit Under Test (UUT)
    Multiplexer uut(.d0(d0),.d1(d1),.d2(d2),.d3(d3),.sel(sel),.out(out));
    initial begin
        // Initialize Inputs
        d0 = 0;
        d1 = 0;
        d2 = 0;
        d3 = 0;
        sel = 0;
        // Wait 100ns for global reset to finish
        #100;
        d0 = 0;
        d1 = 0;
        d2 = 0;
        d3 = 1;
        sel = 1;
        // Wait 100ns for global reset to finish
        #100;
        // Add stimulus here
    end
endmodule
```

Demultiplexers

Verilog for 1-to-8-line Demultiplexer

```
module
Demultiplexer(in,s0,s1,s2,d0,d1,d2,d3,d4,d5,d6,d7);
    input in,s0,s1,s2;
    output d0,d1,d2,d3,d4,d5,d6,d7;
    assign d0=(in& ~s2 & ~s1 & ~s0),
           d1=(in & ~s2 & ~s1 & s0),
           d2=(in & ~s2 & s1 & ~s0),
           d3=(in & ~s2 & s1 & s0),
           d4=(in & s2 & ~s1 & ~s0),
           d5=(in & s2 & ~s1 & s0),
           d6=(in & s2& s1 & ~s0),
           d7=(in & s2 & s1 & s0);
endmodule
```

Simulation Results:



Testbench code for 1-to-8-Line Demultiplexer

```
module TestModule;
    // Inputs
    reg in; reg s0; reg s1; reg s2;
    // Outputs
    wire d0; wire d1; wire d2; wire d3; wire d4; wire d5; wire d6; wire d7;
    // Instantiate the Unit Under Test (UUT)
    Demultiplexer
    uut(.in(in),.s0(s0),.s1(s1),.s2(s2),.d0(d0),.d1(d1),.d2(d2),.d3(d3),.d4(d4),.d5(d5),.d6(d6),
    .d7(d7));
    initial begin
        // Initialize Inputs
        in = 0;
        s0 = 0;
        s1 = 0;
        s2 = 0;
        // Wait 100ns for global reset to finish
        #100;
        in = 1;
        s0 = 0;
        s1 = 1;
        s2 = 0;
        // Wait 100ns for global reset to finish
        #100;
        // Add stimulus here
        end
endmodule
```


To do the Lab Report Assignment (IV)

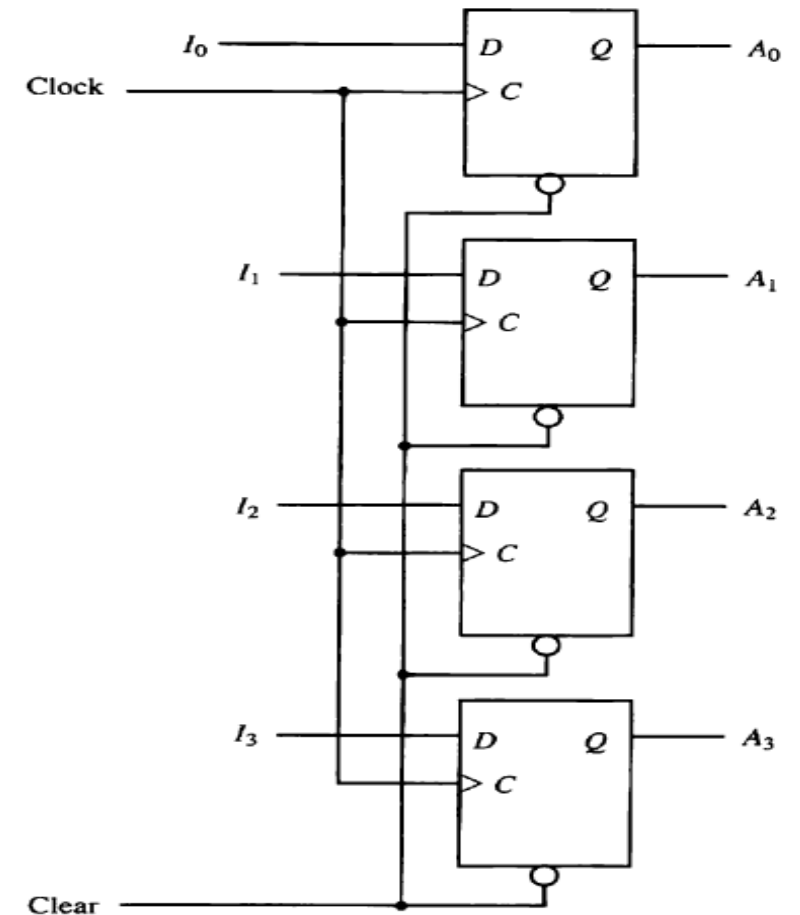
- 8-to-1-line multiplexer
- 1-to-4-line demultiplexer

Registers

- A register is a group of flip-flops with each flip-flop capable of storing one bit of information.
- An *n*-bit register has a group of *n* flip-flops and is capable of storing any binary information of *n* bits.
- A register may have combinational gates that perform certain data-processing tasks.
- The flip-flops hold the binary information and the gates control when and how new information is transferred into the register.
- The transfer of new information into a register is referred to as loading the register.

Registers

- In the 4-bit register, the four outputs can be sampled at any time to obtain the binary information stored in the register.
- The **clear input** goes to a special terminal in each flip-flop.
- When this input goes to **0**, all flip-flops are **reset** asynchronously.
- The clear input is useful for clearing the register to all 0's prior to its clocked operation.
- The **clear input** must be maintained at logic **1** during normal clocked operation.
- The clock signal enables the **D** input but that the clear input is independent of the clock.

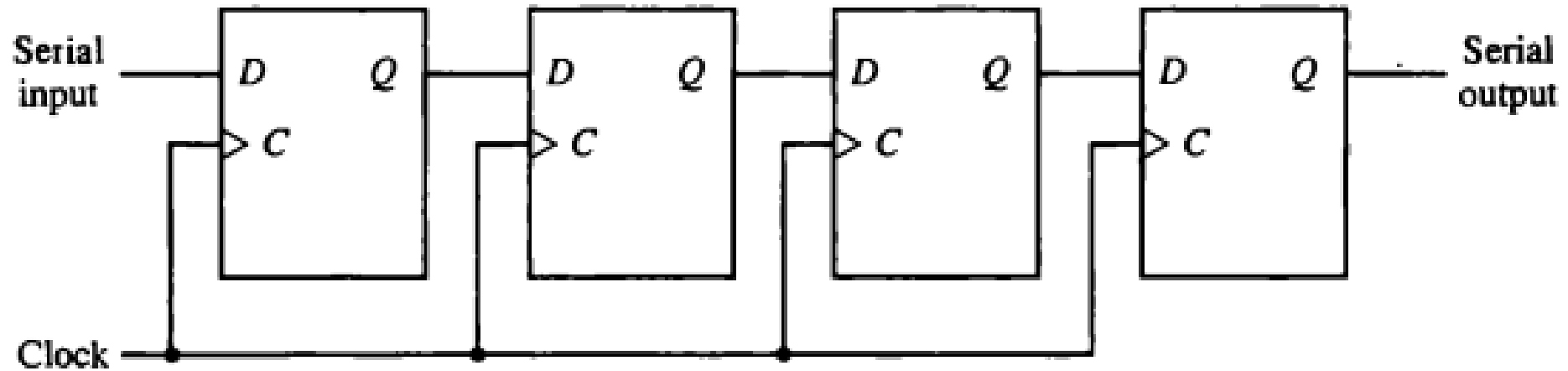


Shift Registers

- A register capable of shifting its binary information in one or both directions is called **a shift register**.
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop.
- All flip-flops receive common clock pulses that initiate the shift from stage to the next.
- A register capable of shifting in **one direction only** is called a **unidirectional** shift register.
- A register that can shift in **both directions** is called a **bidirectional** shift register.

Shift Registers

- The output of a given flip-flop is connected to the D input of the flip-flop at its right.
- The clock is common to all flip-flops.
- The **serial input** determines what goes into the **leftmost position** during the shift.
- The **serial output** is taken from the output of the **rightmost** flip-flop.



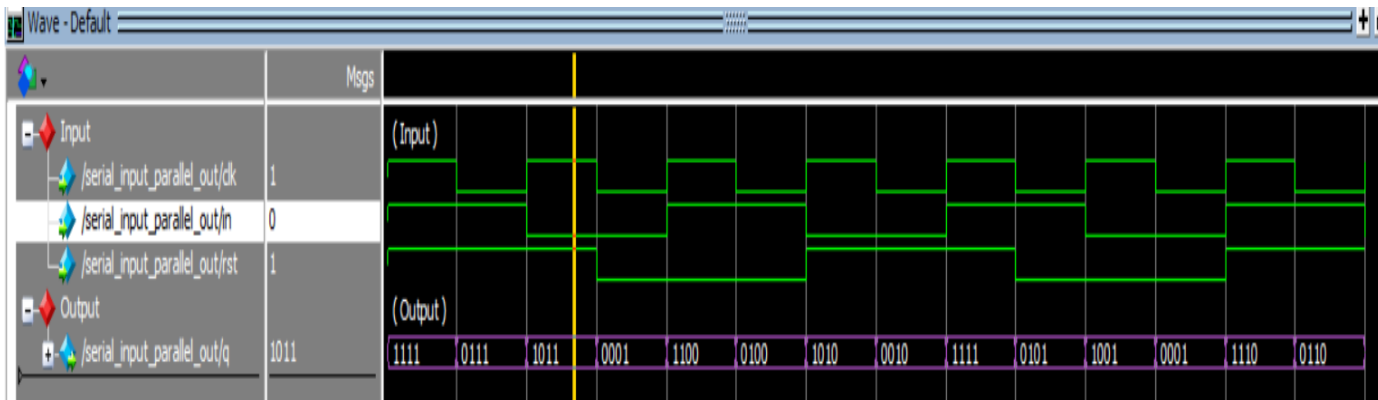
Shift Registers (Serial Input Parallel Output)

Verilog for Shift Register (SIPO)

```
module sipo(clk, in, rst, q);
    input clk, rst;
    input in;
    output reg [3:0]q;

    always @ (posedge clk, posedge rst)
    begin
        if(rst)
            q <= 0;
        else
            q <= {in,q[3:1]};
        end
    end
endmodule
```

Simulation Results for serial input parallel output (SIPO)



Testbench code for Shift Register (SIPO)

```
module sipotb();
    reg clk,in,rst;
    wire [3:0]q;
    sipo dut(clk,in,rst,q);

    initial
    begin
        clk = 1;
        forever #5 clk = ~clk;
    end

    initial
    begin
        rst=1;
        #10 rst=0;in=1;
        #10 in=0;
        #10 in=1;
        #10 in=0;
    end

    initial
    begin
        $dumpfile("sipotb.vcd");
        $dumpvars(0,sipotb);
        #100 $finish;
    end
endmodule
```

Shift Registers (Serial Input Serial Output)

Verilog for Shift Register (SISO)

```
module siso(clk,in,rst,q);
    input clk,rst;
    input in;
    output q;
    reg [3:0]qq;
    assign q=qq[0];
    always @ (posedge clk, posedge rst)
    begin
        if(rst)
            qq <=0;
        else
            qq <= {in,qq[3:1]};
    end
endmodule
```

Testbench code for Shift Register (SISO)

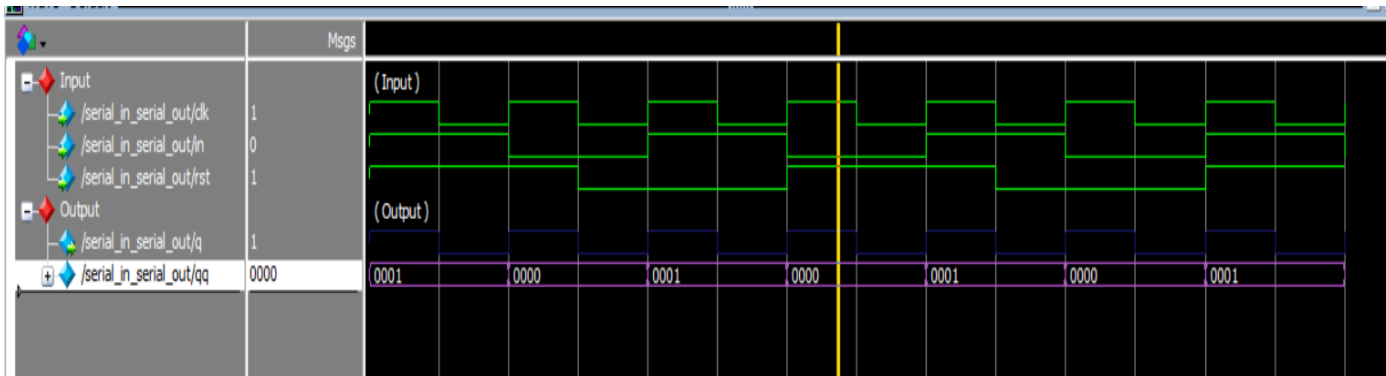
```
module sisotb();
    reg clk,in,rst;
    wire q;
    siso dut(clk,in,rst,q);

    initial
    begin
        clk = 1;
        forever #5 clk = ~clk;
    end

    initial
    begin
        rst=1;
        #10 rst=0;in=0;
        #10 in=1;
        #10 in=1;
        #10 in=0;
    end

    initial
    begin
        $dumpfile("sisotb.vcd");
        $dumpvars(0,sisotb);
        #100 $finish;
    end
endmodule
```

Simulation Results for serial input serial output (SISO)



Shift Registers (Parallel Input Serial Output)

Verilog for Shift Register (PISO)

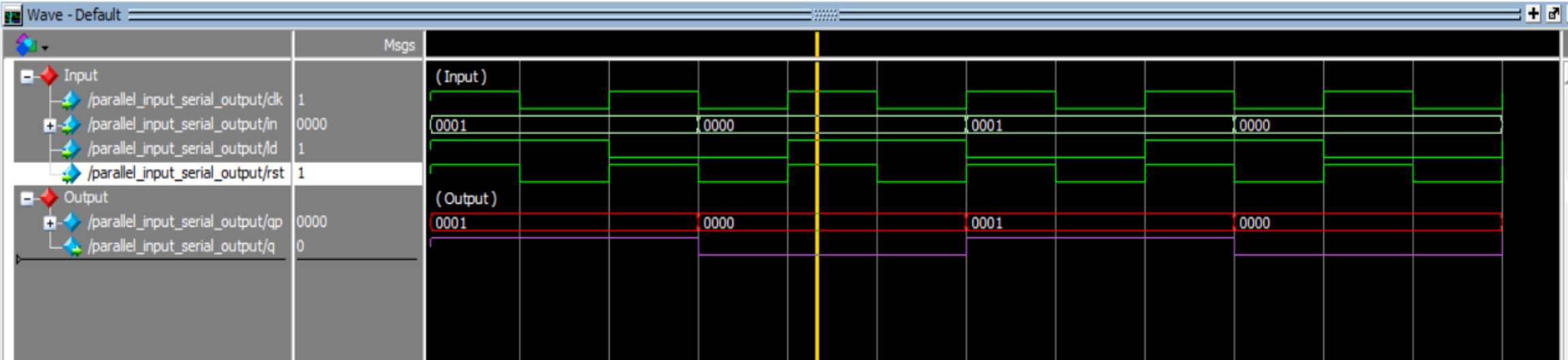
```
module piso(clk,rst,ld,in,q);
    input clk,rst,ld;
    input [3:0]in;
    output reg q;
    reg [3:0]qp;
    always @(posedge clk)
        begin
            if (rst)
                begin
                    q=0;
                    qp=0;
                end
            else if(ld)
                begin
                    qp=in[3:0];
                    q=qp[0];
                end
            else
                begin
                    qp={1'b0,qp[3:1]};
                    q=qp[0];
                end
        end
    end
endmodule
```

Testbench code for Shift Register (PISO)

```
module pisotb();
    reg clk,rst,ld;
    reg in;
    wire q;
    piso m(clk,rst,ld,in,q);
    initial
        begin
            $dumpfile("pisotb.vcd");
            $dumpvars(0,pisotb);
        end
    initial
        begin
            clk=1;
            forever #5 clk=~clk;
        end
    initial
        begin
            #10 rst=1;
            #10 rst=0;ld=1;
            #10 in=1;
            #10 in=0;
            #10 in=0;
            #10 in=1;
            #200 $finish;
        end
    end
endmodule
```


Shift Registers (Parallel Input Serial Output)

Simulation Results for parallel input serial output (PISO)



Shift Registers (Parallel Input Parallel Output)

Verilog for Shift Register (PIPO)

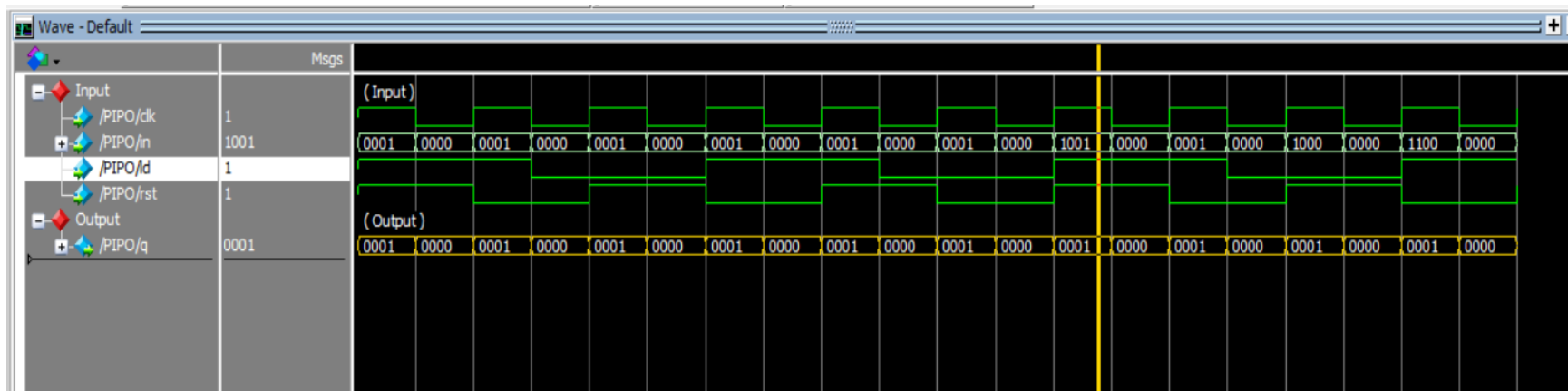
```
module pipo(clk,rst,ld,in,q);
    input clk,rst,ld;
    input [3:0]in;
    output reg [3:0] q;
    always @(posedge clk)
        begin
            if (rst)
                q <= 0;
            else if(ld)
                q=in[3:0];
            end
        end
endmodule
```

Testbench code for Shift Register (PIPO)

```
module pipotb();
    reg clk,rst,ld;
    reg [3:0]in;
    wire [3:0]q;
    pipo m(clk,rst,ld,in,q);
    initial
        begin
            $dumpfile("pipotb.vcd");
            $dumpvars(0,pipotb);
        end
    initial
        begin
            clk=1;
            forever #5 clk=~clk;
        end
    initial
        begin
            #10 rst=1;
            #10 rst=0;ld=1;
            #10 in=4'b1001;

            #200 $finish;
        end
endmodule
```

Simulation Results for parallel input parallel output (PIPO)



Binary Counters

- A register that goes through a predetermined sequence of states upon the application of input pulses is called **a counter**.
- Counters are found in almost all equipment containing digital logic.
- They are used for counting the number of occurrences of an event and are useful for generating timing signals to control the sequence of operations in digital computers.
- A counter that follows the binary number sequence is called **a binary counter**.
- An **n -bit binary counter** is a register of **n flip-flops** and associated gates that follows a sequence of states according to the binary count of n bits, **from 0 to $2^n - 1$** .

Binary Counters

Verilog for 4-bit Binary Counter

```
module counter(clk, reset, up_down, load, data, count);  
    input clk, reset, load, up_down;  
    input [3:0] data;  
    output reg [3:0] count;  
  
    always@(posedge clk)  
    begin  
        if(reset)           // Set Counter to Zero  
            count <= 0;  
        else if(load)       // Load the counter with data value  
            count <= data;  
        else if (up_down)   // count up  
            count <= count + 1;  
        else                // count down  
            count <= count -1 ;  
    end  
endmodule
```

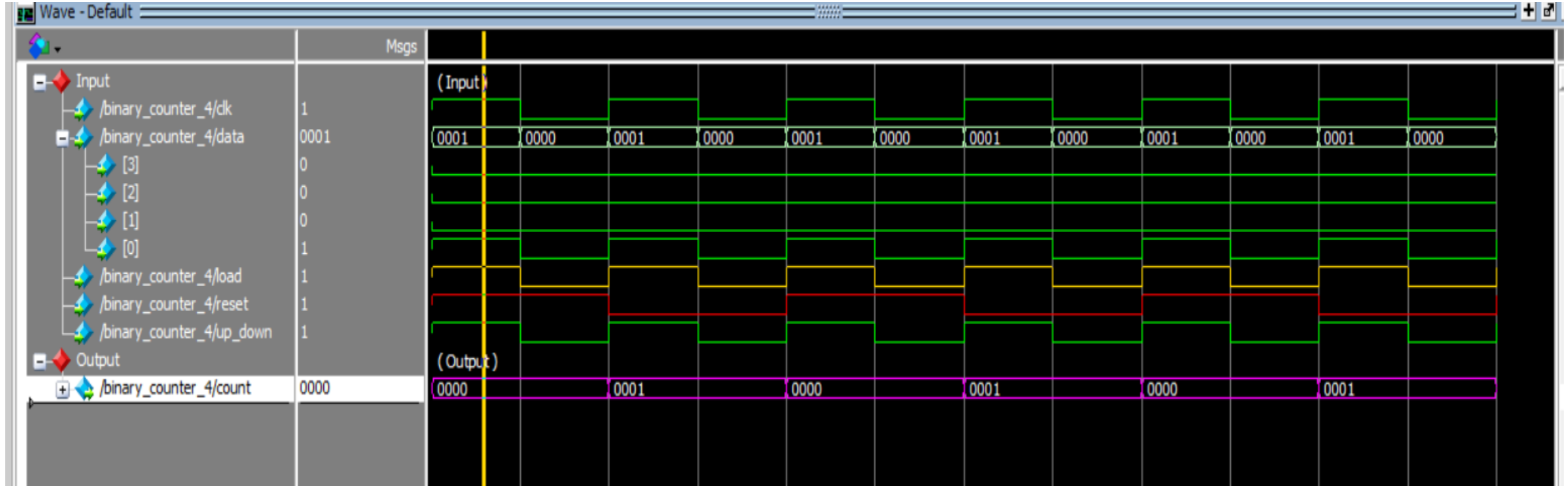
Binary Counters

Testbench for 4-bit Binary Counter

```
module counter_tb;
    reg clk, reset, load, ud;
    reg [3:0] data;
    wire [3:0] count;
    counter uut(.clk(clk),.reset(reset),.load(load),.data(data),.ud(up_down),.count(count));
    initial begin clk = 1'b0; repeat(30) # 3 clk = ~clk; end //clock generator
    initial begin reset = 1'b1; #7 reset = 1'b0; #35 reset= 1'b1; end // insert all input signal
    initial begin #12 load= 1'b1; #5 load = 1'b0; end
    initial begin #5 ud= 1'b1; #24 ud= 1'b0; end
    initial begin data = 4'b1000; #14 data=4'b1101; #2 data = 4'b1111; end
    initial begin $monitor("time=%0d,reset=%b,load=%b,ud=%b,data=%d,count=%d",$time,reset,
load,ud,data,count); end // monitor all input and output ports at times when any inputs changes its
state
endmodule
```


Binary Counters

Expected Output for 4-bit Binary Counter



Memory Unit

- A **memory unit** is a collection of storage cells together with associated circuits needed to transfer information in and out of storage.
- The memory stores binary information in groups of bits called **words**.
- A word in memory is an entity of bits that move in and out of storage as a unit.
- A memory word is **a group of 1's and 0's** and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary-coded information.
- A group of 8 bits is called **a byte**.
- Most computer memories use words whose number of bits is a multiple of 8.
- The **capacity of memories** in commercial computers is stated as the **total number of bytes** that can be stored.

Memory Unit

- Each word in memory is assigned an identification number, called **an address**, starting from 0 and continuing with 1, 2, 3, up to $2^k - 1$, where k is the number of address lines.
- Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits.
- K is equal to 2^{10} , M is equal to 2^{20} , G is equal to 2^{30} .
- $64K = 2^{16}$, $2M = 2^{21}$, $4G = 2^{32}$.
- Two major types of memories are used in computer systems:
 - **random-access memory (RAM)**
 - **read-only memory (ROM)**

Memory Unit

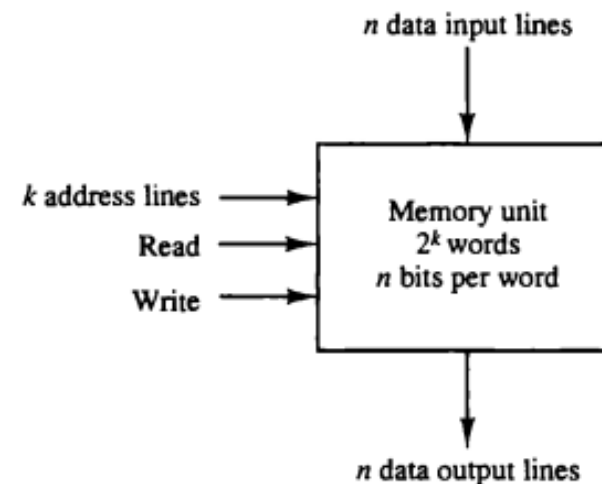
Random-Access Memory (RAM)

- In RAM, the memory cells can be accessed for information transfer from any desired random location.
- The process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory: thus the name “random access”.

Memory Unit

Random-Access Memory (RAM)

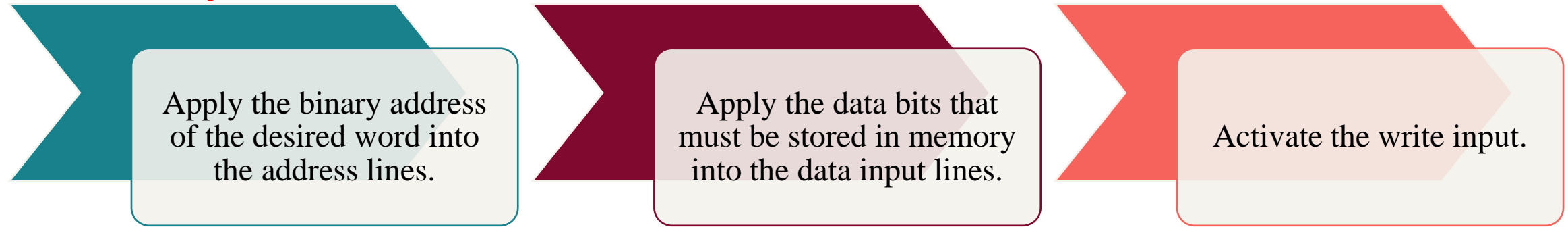
- The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory.
- The k address lines provide a binary number of k bits that specify a particular word chosen among the 2^k available inside the memory.
- The control inputs specify the direction of transfer desired.
- The **write** signal specifies a **transfer-in** operation.
- The **read** signal specifies a **transfer-out** operation.



Memory Unit

Random-Access Memory (RAM)

- The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:



- The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:



Memory Unit

Random-Access Memory (RAM)

Single-Port Random-Access Memory (RAM)

- A single port RAM is a that RAM in which **only one address** can be accessed at a particular time.
- The address can be accessed for **read operation or write operation** at a particular interval of time.
- Single port RAM allows **only one memory cell** to be read or write during each clock cycle .
- It has one enable(en) input and one write(we) input.
- If **both are logic '1'**, data is **written** into RAM and **enable(en)** is logic '1' and write (we) is logic '0' , then the data is **read** from the RAM.

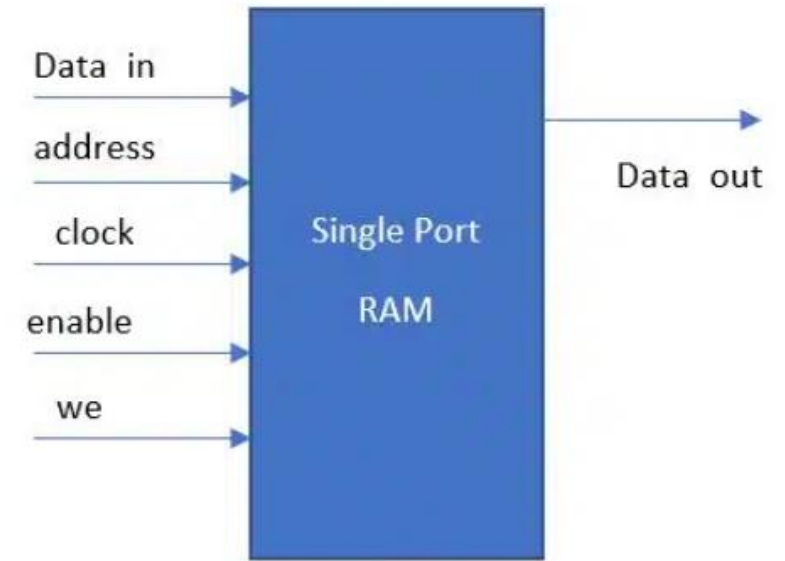


Fig. Block Diagram of a Single Port RAM

Memory Unit

Random-Access Memory (RAM)

Verilog for Single-Port Random-Access Memory (RAM)

```
module single_port_ram(data_in, ram_address, write_enable, clk, data_out);
    input [7:0]data_in;
    input [5:0] ram_address;
    input write_enable;
    input clk;
    output [7:0]data_out;
    reg [7:0] ram_memory[31:0]; // a 32 byte (32*8 bit) RAM
    reg [5:0] address_register;

    always@(posedge clk)
    begin
        if (write_enable) //write operation
            ram_memory[ram_address] <= data_in;
        else
            address_register <= ram_address;
        end
        assign data_out = ram_memory[address_register];
    endmodule
```

Memory Unit

Random-Access Memory (RAM)

Testbench for Single-Port Random-Access Memory (RAM)

```
module single_port_ram_testbench;
    reg [7:0]data_in;
    reg [5:0] ram_address;
    reg write_enable;
    reg clk;
    wire [7:0]data_out;

    single_port_ram ram1(data_in, ram_address, write_enable, clk, data_out);

    initial begin // clock initialization
        clk = 1'b1;
        forever #10 clk =~clk;
    end

    initial begin // write data into the memory
        write_enable = 1'b1;
        #20 ram_address = 5'd0;
        data_in = 8'h10;
```

Memory Unit

Random-Access Memory (RAM)

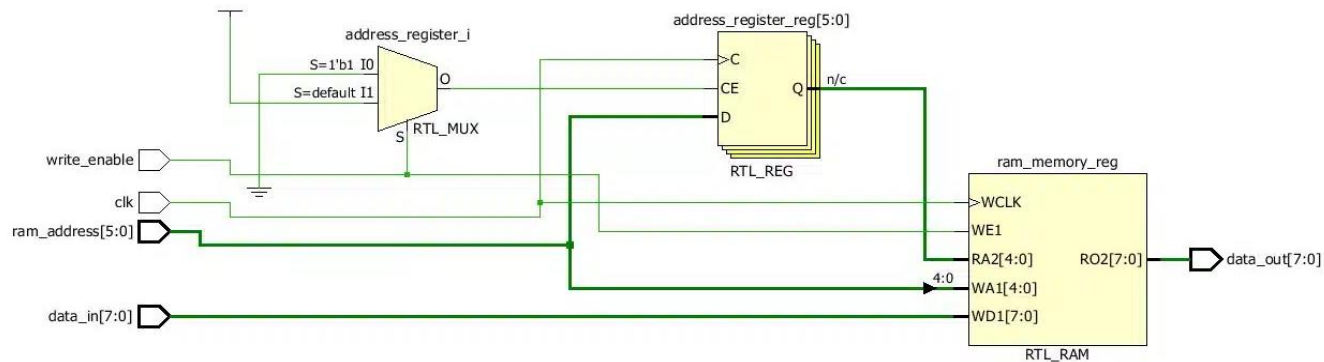
Testbench for Single-Port Random-Access Memory (RAM)

```
#20 ram_address = 5'd2;  
data_in = 8'h11;  
#20 ram_address = 5'd7;  
data_in = 8'haf;  
#20  
// reading data from the memory  
write_enable = 1'b0;  
ram_address = 5'd0;  
#20  
ram_address = 5'd2;  
#20  
ram_address = 5'd7;  
#20  
  
$finish;  
end  
endmodule
```

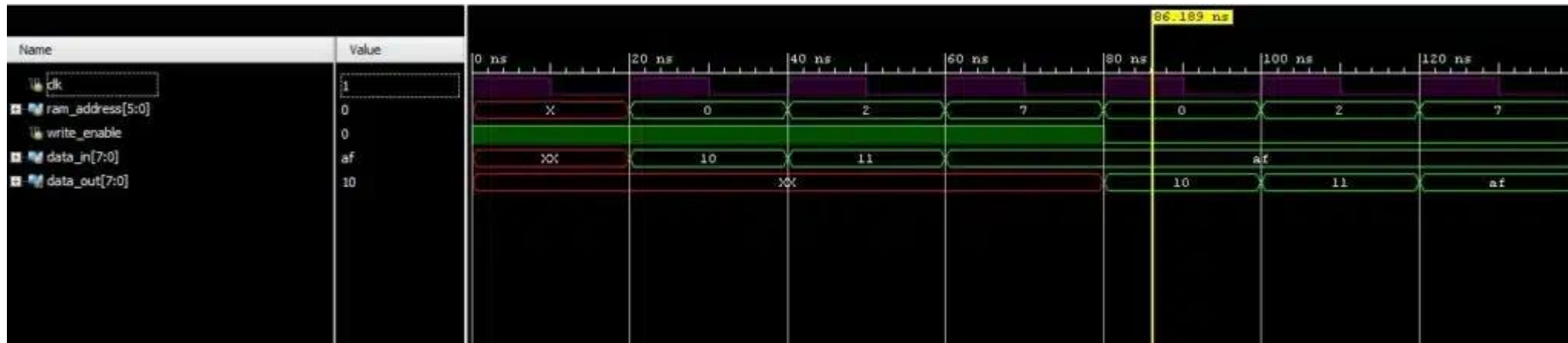
Memory Unit

Random-Access Memory (RAM)

RTL Schematic Diagram for Single-Port Random-Access Memory (RAM)



Simulation Results for Single-Port Random-Access Memory (RAM)



Memory Unit

Random-Access Memory (RAM)

Dual-Port Random-Access Memory (RAM)

- Dual-port RAM is a type of computer memory that allows **two separate devices to read and write** data simultaneously.
- This means that two processors or devices can **access the same memory at the same time**, without causing data conflicts or delays.
- Two operations can be done on the dual port ram at single interval of time.
- It allows **read and write of the data at the same time** (two different addresses can be accessed at a particular interval of time for read and write operations.)
- Dual-port Ram is commonly used in **multi-tasking systems** where multiple processors or devices need to access the same data quickly and efficiently, such as in **networking devices, image processing systems, and real-time data acquisition systems.**

Memory Unit

Random-Access Memory (RAM)

Verilog for Dual-Port Random-Access Memory (RAM)

```
module dual_port_ram(  
    input clock,  
    input write_enable_A,  
    input write_enable_B,  
    input [7:0] data_in_A,  
    input [7:0] data_in_B,  
    input [7:0] address_A,  
    input [7:0] address_B,  
    output [7:0] data_out_A,  
    output [7:0] data_out_B);  
  
    reg [7:0] mem [15:0];  
  
    always@(posedge clock) begin  
        if (write_enable_A) begin  
            mem[address_A] <= data_in_A;  
        end  
        if (write_enable_B) begin  
            mem[address_B] <= data_in_B;  
        end  
    end  
end
```

assign data_out_A = mem[address_A];
assign data_out_B = mem[address_B];

endmodule

Memory Unit

Random-Access Memory (RAM)

Testbench for Dual-Port Random-Access Memory (RAM)

```
module dual_port_ram_tb;
    reg clock;
    reg write_enable_A;
    reg write_enable_B;
    reg [7:0] data_in_A;
    reg [7:0] data_in_B;
    reg [7:0] address_A;
    reg [7:0] address_B;
    wire [7:0] data_out_A;
    wire [7:0] data_out_B;

    dual_port_ram dut (
        .clock(clock),
        .write_enable_A(write_enable_A),
        .write_enable_B(write_enable_B),
        .data_in_A(data_in_A),
        .data_in_B(data_in_B),
        .address_A(address_A),
        .address_B(address_B),
        .data_out_A(data_out_A),
        .data_out_B(data_out_B));

    initial begin
        clock = 0;
        write_enable_A = 1;
        write_enable_B = 1;
        data_in_A = 4'hF;
        data_in_B = 4'h0;
        address_A = 4'h0;
        address_B = 4'hF;

        #10 write_enable_A = 0;
        write_enable_B = 0;

        #10 address_A = 4'hF;
        address_B = 4'h0;

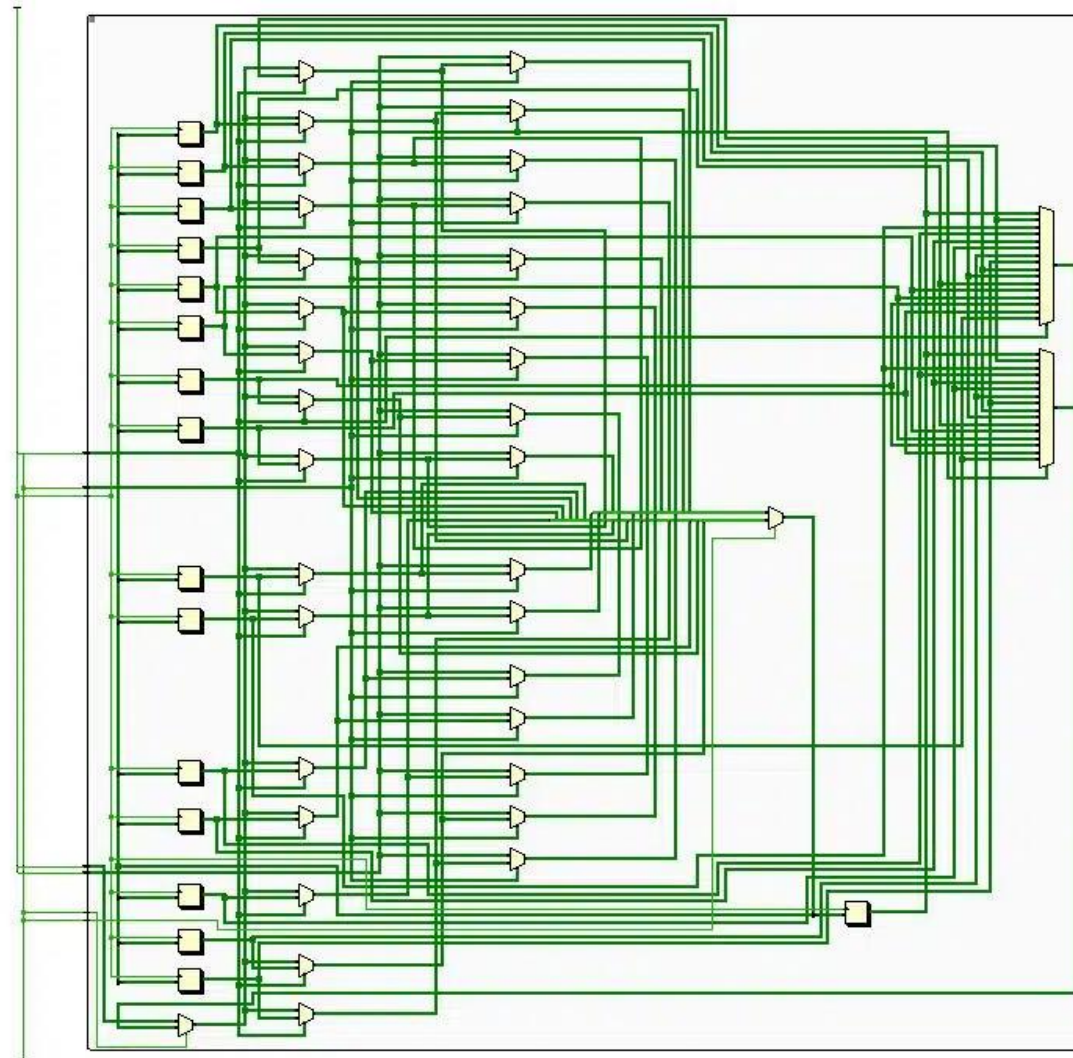
        #10
        $finish;
    end
    always begin
        clock = 0; #5; clock = 1; #5;
    end
endmodule

#10 write_enable_A = 1;
write_enable_B = 1;
address_A = 4'hF;
address_B = 4'h0;
```


Memory Unit

Random-Access Memory (RAM)

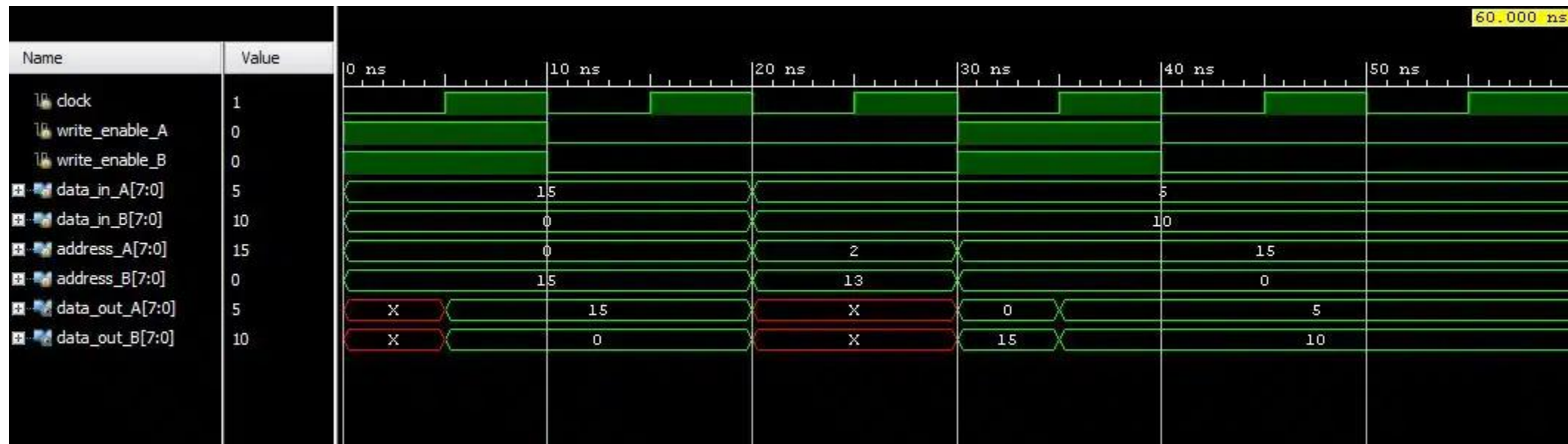
RTL Schematic Diagram for Dual-Port Random-Access Memory (RAM)



Memory Unit

Random-Access Memory (RAM)

Simulation Results for Dual-Port Random-Access Memory (RAM)



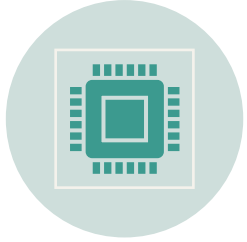
Memory Unit

Read Only Memory (ROM)

- A ROM is a memory unit that performs the **read operation only**; it does **not have a write capability**.
- A ROM is restricted to reading words that are permanently stored within the unit.
- ROMs come with special internal electronic fuses that can be “programmed” for a specific configuration.
- It stays within the unit even when power is turned off and on again.

Memory Unit

Read Only Memory (ROM)



An $m \times n$ ROM is an array of binary cells organized into m words of n bits each.



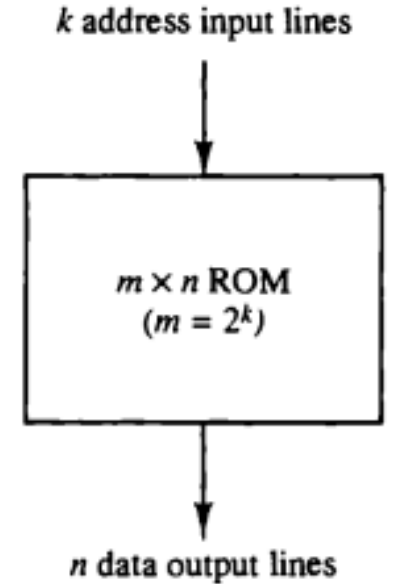
An integrated circuit ROM may have one or more enable inputs for expanding a number of packages into a ROM with larger capacity.



A ROM has k address input lines to select one of $2^k = m$ words of memory, and n output lines, one for each bit of the word.



ROM is employed in the design of control units for digital computer.



Memory Unit

Read Only Memory (ROM)

Verilog for Read-Only Memory (ROM)

```
// ROM module design
module rom (
    input clk, //clk
    input en, //enable
    input [3:0] addr, //address
    output reg [3:0] data //output data
);

    reg [3:0] mem [15:0]; //4-bit data and 16 locations

    always @ (posedge clk)
        begin
            if (en)
                data <= mem[addr];
            else
                data <= 4'bxxxx;
        end
end
```

```
initial
begin
    mem[0] = 4'b0010;
    mem[1] = 4'b0010;
    mem[2] = 4'b1110;
    mem[3] = 4'b0010;
    mem[4] = 4'b0100;
    mem[5] = 4'b1010;
    mem[6] = 4'b1100;
    mem[7] = 4'b0000;
    mem[8] = 4'b1010;
    mem[9] = 4'b0010;
    mem[10] = 4'b1110;
    mem[11] = 4'b0010;
    mem[12] = 4'b0100;
    mem[13] = 4'b1010;
    mem[14] = 4'b1100;
    mem[15] = 4'b0000;
end
endmodule
```

Memory Unit

Read Only Memory (ROM)

Testbench for Read-Only Memory (ROM)

```
// ROM testbench
```

```
module rom_tb;
  reg clk; //clk
  reg en; //enable
  reg [3:0] addr; //address
  wire [3:0] data; //output data

  rom r1(.clk(clk),.en(en),.addr(addr),.data(data));

  initial
  begin
    $dumpfile("dump.vcd");
    $dumpvars(1, rom_tb);

    clk=1'b1;
    forever #5 clk = ~clk;
  end
  initial
  begin
    en = 1'b0;
    #10;
```

```
en = 1'b1;
addr = 4'b1010;
#10;
```

```
addr = 4'b0110;
#10;
```

```
addr = 4'b0011;
#10;
```

```
en = 1'b0;
addr = 4'b1111;
#10;
```

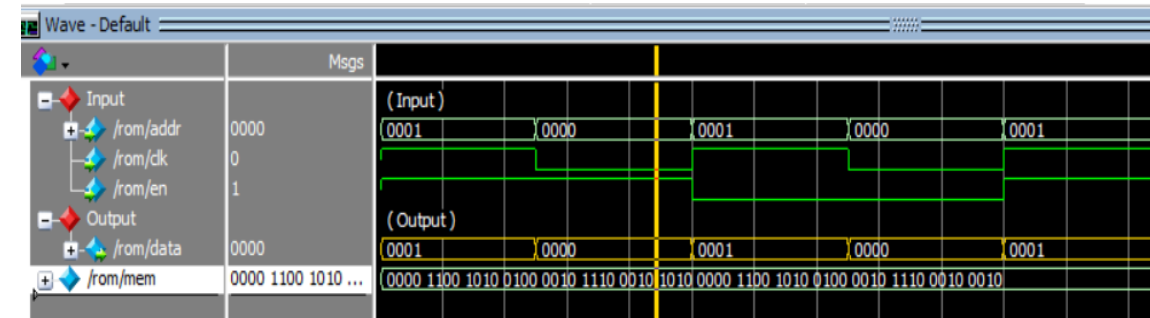
```
en = 1'b1;
addr = 4'b1000;
#10;
```


```
addr = 4'b0000;
#10;
```

```
addr = 4'bxxxx;
#10;
end
```

```
initial
begin
  #80 $stop;
end
endmodule
```

Simulation Results for Read-Only Memory (ROM)



A vibrant image of the Aurora Borealis (Northern Lights) in shades of green and blue, set against a dark, starry night sky. The aurora's glowing bands curve across the frame, creating a dynamic and ethereal background.

Next Lecture:

Chapter (4): Register Transfer and Microoperations