Lab No: 7

DECODERS AND ENCODERS

Objectives:

In this lab, student will be able to

1. Learn the concept of decoders, encoders and priority encoders.
2. Write Verilog code for decoders, decoder trees and encoders.

Decoders

- Decoder circuits are used to decode the encoded information.
- A binary decoder, depicted in Fig. 8.1, is a logic circuit with n inputs and 2^n outputs.
- Each output corresponds to one valuation of the inputs, and only one output is asserted at a time.
- The decoder also has an enable input En, that is used to disable the outputs; if En = 0, then none of the decoder outputs is asserted.
- If En = 1, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted.
- Larger decoders can be built using smaller decoders referred to as decoder tree.

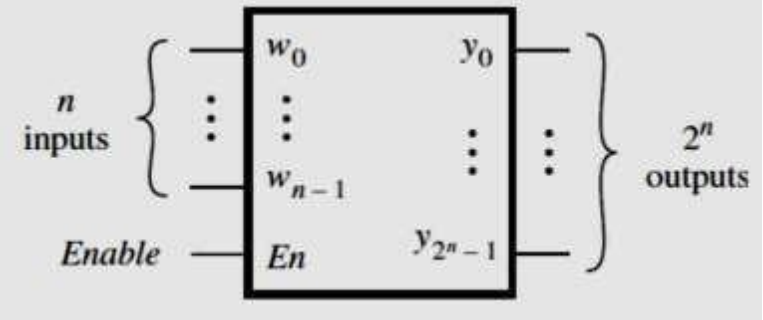


Figure 7.1 Decoder

Binary Encoders

- A binary encoder encodes information from 2^n inputs into an n-bit code, as indicated in Fig. 8.2.
- Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1.

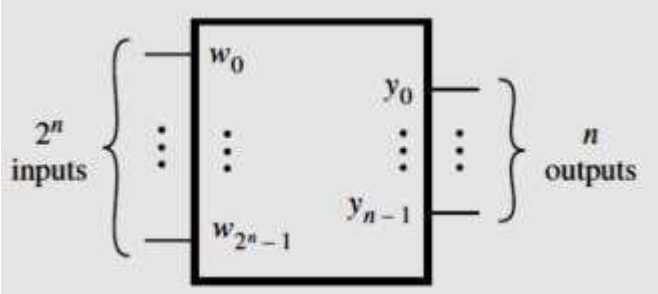


Figure 7.2 Encoder

Priority Encoder

- In a priority encoder, each input has a priority level associated with it.
- When an input with a high priority is asserted, the other inputs with lower priority are ignored. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition.
- The truth table for a 4-to-2 priority encoder is shown in Fig. 8.3.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 7.3 Truth table for a 4 to 2 priority encoder

Casex Statement:

- Verilog provides variants of the **case** statement that treat the *z* and *x* values in a different way.
- The **casez** statement treats all *z* values as don't cares.
- The **casex** statement treats all *z* and *x* values as don't cares.

Solved exercise

Write behavioral Verilog code for 2 to 4 binary decoder using **for** loop.

Verilog code:

```
module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output [0:3] Y;
    reg [0:3] Y;
    integer k;
    always @(W or En)
    for (k = 0; k <= 3; k = k+1)
    if ((W == k) && (En == 1))
    Y[k] = 1;
    else
    Y[k] = 0;
endmodule
```

Lab Exercises

1. Write behavioral Verilog code for a 2 to 4 decoder with active-high enable input and active high output using the **if-else** statement. Using the 2 to 4 decoders above, design a 3 to 8 decoder and write the Verilog code for the same.
2. Write behavioral Verilog code for a 3 to 8 decoder with active-high enable input and active high output using **for** loop. Using the 3 to 8 decoders above, design a 4 to 16 decoder and write the Verilog code for the same.
3. Write behavioral Verilog code for a 4 to 2 priority encoder using **casex** statement.

Lab No: 8

COMPARATORS AND CODE CONVERTERS

Objectives:

In this lab, student will be able to

1. Learn the concept of comparators and code converters.
2. Write Verilog code to simulate comparators and code converters.

Arithmetic Comparison Circuits

- Compare the relative magnitude of two binary numbers.

Code Converters

- Convert from one type of input representation to a different output representation.

Parameters

- A parameter associates an identifier name with a constant.
- Using the following declaration, the identifier n can be used in place of the number 4.

  parameter n = 4;

Verilog Procedural Statements

- Rather than using gates or logic expressions, circuits can be specified in terms of their behavior.
- Also called sequential statements.
- Procedural statements are evaluated in the order in which they appear in the code whereas concurrent statements are executed in parallel.
- Verilog syntax requires that procedural statements should be inside an **always** block.

Always Block

- An **always** block is a construct that contains one or more procedural statements.
- It has the form

```
always @(sensitivity_list)
[begin]
    [procedural assignment statements]
    [if-else statements]
    [case statements]
    [while, repeat, and for loops]
    [task and function calls]
[end]
```

- The ***sensitivity_list*** is a list of signals that directly affect the output results generated by the **always** block.
- If the value of a signal in the sensitivity list changes, then the statements inside the **always** block are evaluated in the order presented.

Variables:

- A variable can be assigned a value and this value is retained until it is overwritten in a subsequent assignment statement.
- There are two types of variables, **reg,** and an **integer**.
  - The keyword **reg** does not denote a storage element or register. In Verilog code, **reg** variables can be used to model either combinational or sequential parts of a circuit.
  - **Integer** variables are useful for describing the behavior of a module, but they do not directly correspond to nodes in a circuit.

The if-else Statement

- The general form of the **if-else** statement is given below.

```
if (expression1)
begin
    statement;
end
else if (expression2)
begin
    statement;
end
else
begin
    statement;
end
```

- If expression1 is true, then the first statement is evaluated.
- When multiple statements are involved, they have to be included inside a **begin-end** block.
- The **else if** and **else** clauses are optional.
- Verilog syntax specifies that when **else if** or **else** are included, they are paired with the most recent unfinished **if** or **else if**.

for Loop

- the general form of **for** loop

```
for (initial_index; terminal_index; increment)
  begin
      statement;
  end
```

- The ***initial_index*** is evaluated once, before the first loop iteration and typically performs the initialization of the **integer** loop control variable.
- In each loop iteration, the begin-end block is performed, and then the increment statement is evaluated.
- Finally, the ***terminal_index*** condition is checked, and if it is True (1), then another loop iteration is done.

Solved Exercise

Write Verilog code to simulate a 1-bit equality comparator.

Verilog code:

```
module compare(A, B, AeqB);
    input A, B;
    output reg AeqB;
    always @(A,B)
    begin
        AeqB=0;
        if (A==B)
            AeqB=1;
    end
endmodule
```

Lab exercises

1. Design a 5-bit comparator using only logic gates. Write behavioral Verilog code to simulate the design.
2. Write behavioral Verilog code to simulate a code converter that converts a decimal digit from 8, 4,-2,-1 code to BCD.

Lab No: 9

FLIP FLOPS AND REGISTERS

Objectives:

In this lab. student will be able to

1. Learn different types of Flip Flops.
2. Understand the concept of triggering flip flops and different types of reset.
3. Understand the concept of registers and shift registers.
4. Write Verilog code for Flip Flops and registers.

I. Flip Flops and Registers

Flip Flops:

- A flip flop circuit can maintain a binary state until directed by an input signal to switch the state.
- Major differences among various types of flip flops are in the number of inputs they process and in the manner in which the inputs affect the binary state.

Triggering of Flip-Flops:

- The state of a flip flop is switched by a momentary change in the input signal which is called triggering the flip flop.

Positive Edge and Negative Edge:

- A positive clock source remains at 0 during the interval between pulses and goes to 1 during the occurrence of a pulse.
- The pulse transition from 0 to 1 is called a **positive edge** and return from 1 to 0 is called a **negative edge.**

Registers:

- A register is a group of binary cells suitable for holding binary information.

Shift Registers:

- A register capable of shifting its binary information either to the right or to the left is called a shift register.

II. Verilog Constructs for Storage Elements

- The Verilog keywords **posedge** and **negedge** are used to implement edge-triggered circuits.
- The keyword **posedge** specifies that a change may occur only on the positive edge of Clock.

- The keyword **negedge** specifies that a change may occur only on the negative edge of Clock.

Blocking and Non-Blocking Assignments

Blocking

- A Verilog compiler evaluates the statements in an **always** block in the order in which they are written.
- If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.
- Denoted by the '=' symbol
- Example

```
Q1 = D;
Q2 = Q1;
```

- The above statement results in Q2=Q1=D

Non-Blocking

- Verilog also provides a non-blocking assignment, denoted with '<='.
- All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered.
- Thus, a given variable has the same value for all statements in the block.
- The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.
- Example

```
Q1 <= D;
Q2 <= Q1;
```

- The variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block.

Flip-Flops with Clear Capability

- By using a particular sensitivity list and a specific style of an **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

Solved Exercise:

Write behavioral Verilog code for positive edge-triggered D FF with synchronous reset.

Verilog Code:

```
module flipflop (D, Clock, Resetn, Q);
        input D, Clock, Resetn;
        output Q;
        reg Q;
        always @(posedge Clock)
        if (!Resetn)
        Q <= 0;
        else
        Q <= D;
endmodule
```

Shift	Load	Register Operation
0	0	Shift left
0	1	Load parallel data
1	X	No change

Lab Exercises

1. Write behavioral Verilog code for a positive edge-triggered D FF with asynchronous active high reset.
2. Write behavioral Verilog code for a negative edge triggered T FF with asynchronous active low reset.
3. Write behavioral Verilog code for a positive edge-triggered JK FF with synchronous active high reset.

Lab No: 10

COUNTERS

Objectives:

In this lab, student will be able to

1. Learn the concept of synchronous/asynchronous up/down counters.
2. Learn the concept of ring and Johnson counters.
3. Write Verilog code for different types of counters.

Counters:

- A counter is essentially a register that goes through a predetermined sequence of states upon the application of input pulses.
- A binary counter with a reverse count is called a binary down counter.

Ripple Counters

- Also called as asynchronous counters.
- The CP inputs of all flip flops (except the first) are triggered not by the incoming pulses, but by the transition that occurs in other flip flops.
- Four-bit binary ripple counter is shown in Fig. 11.1

Synchronous Counters

- The input pulses are applied to the CP input of all the flip flops.
- The common pulse triggers all flip flops simultaneously.
- The change of state of a particular flip flop is dependent on the present state of other flip flops.
- For synchronous sequential circuits the design procedure is as follows:
    3. From the given information (word description/state diagram/timing diagram/other pertinent information) about the circuit, obtain the state table.
    4. Determine the number of flip flops needed.
    5. From the state table, derive the circuit excitation and output tables.
    6. Derive the circuit output functions and the flip flop input functions by simplification.
    7. Draw the logic diagram.

Ring Counter

- Circular shift register with only one flip flop being set at any particular time, all others are cleared.
- N bit ring counter will have N states and requires N flip flops.
- Fig. 11.2 shows a 4-bit ring counter using decoder and counter.

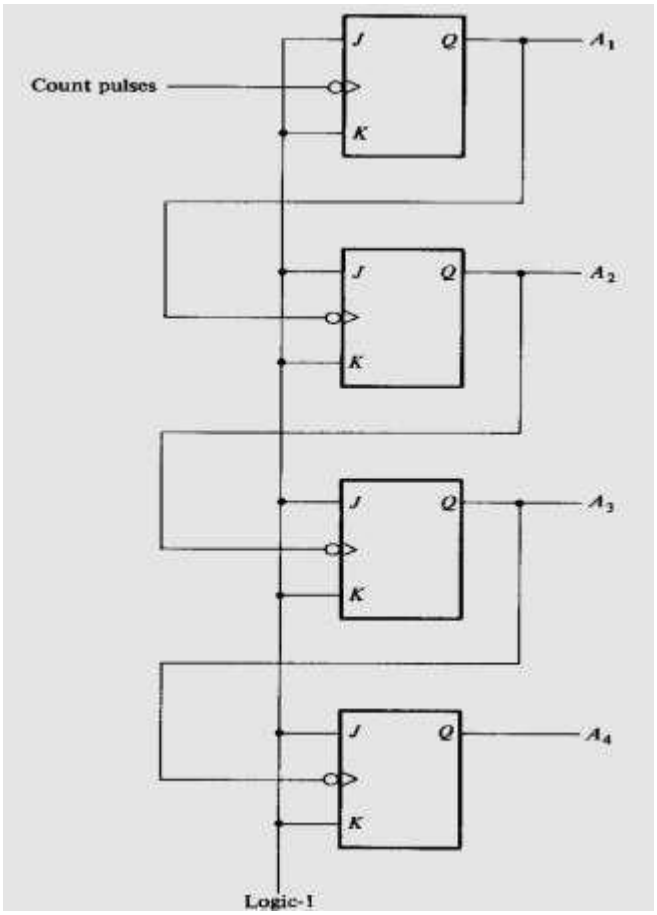


Figure 10.1 4-bit binary ripple counter

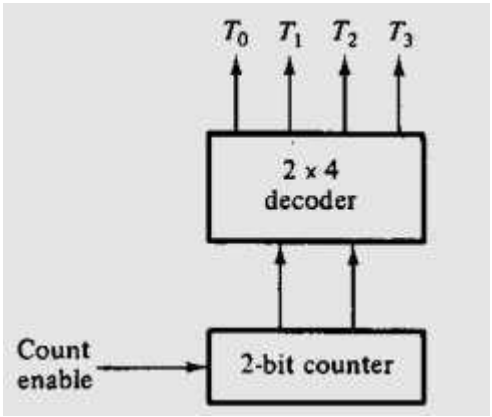


Figure 10.2 4-bit ring counter

Johnson counter or Switch tail ring counter

- Circular shift register with the complement output of the last flip flop connected to the input of the first flip flop.
- k-bit switch tail ring counter with 2k decoding gates provide outputs for 2k timing signals.

- k- bit Johnson counter requires k flip flops.
- Fig. 11.3 shows a 4-bit Johnson counter

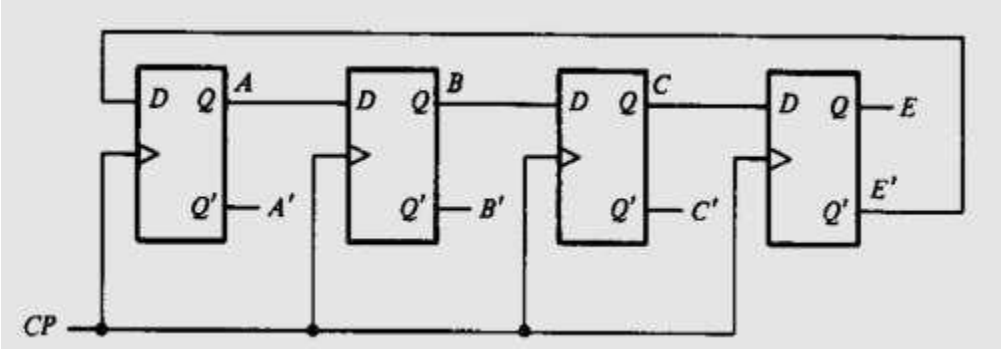


Figure 10.3 4-bit Johnson counter

Solved Exercise

Write Verilog code for a 2-bit asynchronous up counter.

```
module tff1(T, Clock, Q);
        input T, Clock;
        output Q;
        reg Q;
        always @(negedge Clock)
        if (!T)
          Q <= Q;
        else
          Q <= ~Q;
endmodule

module twobit (clock, Q);
        input clock;
        output [1:0]Q;
        tff1 f1(1,clock,Q[0]);
        tff1 f2(1,Q[0],Q[1]);
endmodule
```

Lab Exercises

1. Design and simulate the following counters
    a) 4-bit ring counter.
    b) 5 bit Johnson counter.
    c) 4 bit asynchronous up counter
    d) 4 bit synchronous up counter

Lab No: 11

REALIZE 4 BIT BINARY TO GRAY CONVERTER

THEORY: Gray code is a non-weighted code. In these codes while traversing from one step to another step only one bit in the code group changes. In case of Gray Code two adjacent code numbers differ from each other by only one bit.

The problem with natural binary codes is that, with physical, mechanical switches, it is very unlikely that switches will change states exactly in synchrony. For example, in the transition between 011 to 100, all three switches change state. In the brief period while all are changing, the switches will read some spurious position. Even without key bounce, the transition might look like 011 — 001 — 101 — 100. When the switches appear to be in position 001, the observer cannot tell if that is the "real" position 001, or a transitional state between two other positions. If the output feeds into a sequential system, possibly via combinational logic, then the sequential system may store a false value.The reflected binary code or Gray code solves this problem by changing only one switch at a time.

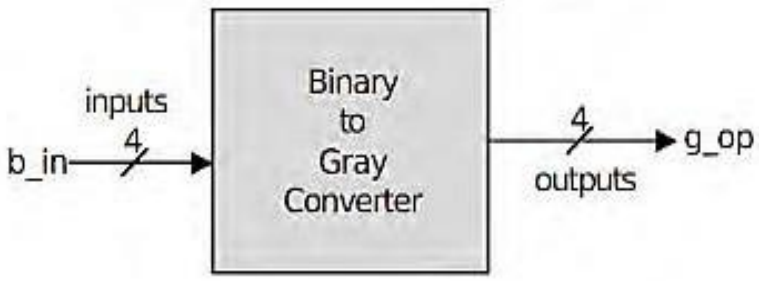
Conversion of Binary to Gray code:

1. The MSB of the gray code will be equal to the MSB of the given binary code.
2. The second bit of the gray code will be exclusive-or of the first and second bit of the given binary number, i.e. if both the bits are same the result will be 0 and if they are different the result will be 1.
3. The third bit of gray code will be equal to the exclusive-or of the second and third bit of the given binary number. This process continues until we get the last bit.

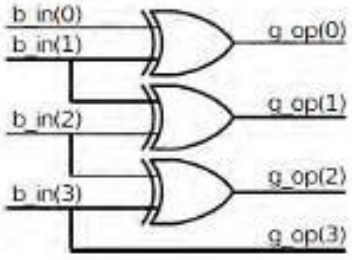
Conversion of Gray to Binary code:

1. The MSB of the binary code will be equal to the MSB of the given gray code.
2. Take the next bit down in the Gray code and the most significant bit of the binary and do a logical XOR and this becomes the next bit in the binary code.
3. Take the next bit down in the Gray code and the next bit down in the binary code and do a logical XOR and this becomes the next bit in the binary code.
4. Continue this process until we get the last bit.

Block Diagram:



Logic Diagram and expressions:



Boolean Expressions

$g\_op(3) = b\_in(3)$

$g\_op(2) = b\_in(3) \oplus b\_in(2)$

$g\_op(1) = b\_in(2) \oplus b\_in(1)$

$g\_op(0) = b\_in(1) \oplus b\_in(0)$

Truth Table:

Decimal	Inputs				Outputs			
	Binary				Gray			
	b_in(3)	b_in(2)	b_in(1)	b_in(0)	g_op(3)	g_op(2)	g_op(1)	g_op(0)
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

VERILOG CODE of 4 bit binary to gray converter:

```
module binary_gray(b_in, g_op);
input [3:0] b_in;
output [3:0] g_op;

assign g_op[3] = b_in[3];
assign g_op[2] = b_in[3] ^ b_in[2];
assign g_op[1] = b_in[2] ^ b_in[1];
assign g_op[0] = b_in[1] ^ b_in[0];
endmodule
```

Lab No: 12

SIMPLE PROCESSOR DESIGN

Objectives:

In this lab student will be able to

1. Learn the concept of bus structure.
2. Understand the concept of a simple processor and a bit counting circuit.
3. Write Verilog code to implement bus structure, simple processor and bit counting circuit.

Bus Structure:

- When a digital system contains a number of n-bit registers, to transfer data from any register to any other register, a simple way of providing the desired interconnectivity is to connect each register to a common set of n wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a bus.
- If common paths are used to transfer data from multiple sources to multiple destinations, it is necessary to ensure that only one register acts as a source at any given time and other registers do not interfere.

There are two arrangements for implementing the bus structure.

- Using Tri-State Drivers
- Using Multiplexers

Using Tri-State Drivers to Implement a Bus:

- Consider a system that contains k n-bit registers, R1 to Rk. Figure 12.1 shows how these registers can be connected using tri-state drivers to implement the bus structure. The data outputs of each register are connected to tri-state drivers. When selected by their enable signals, the drivers place the contents of the corresponding register onto the bus wires. The enable signals are generated by a control circuit.
- In addition to registers, in a real system, other types of circuit blocks would be connected to the bus. The figure shows how n bits of data from an external source can be placed on the bus, using the control input Extern.
- It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. The control circuit must ensure that only one of the tri-state driver enables signals, R1out, . . . , Rkout, is asserted at a given time. The control circuit also produces the signals R1in, . . . , Rkin, which determines when data is loaded into each register.
- In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and controlling the processing of data in

various functional units of the system. Figure 12.1 shows an input signal named Function that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

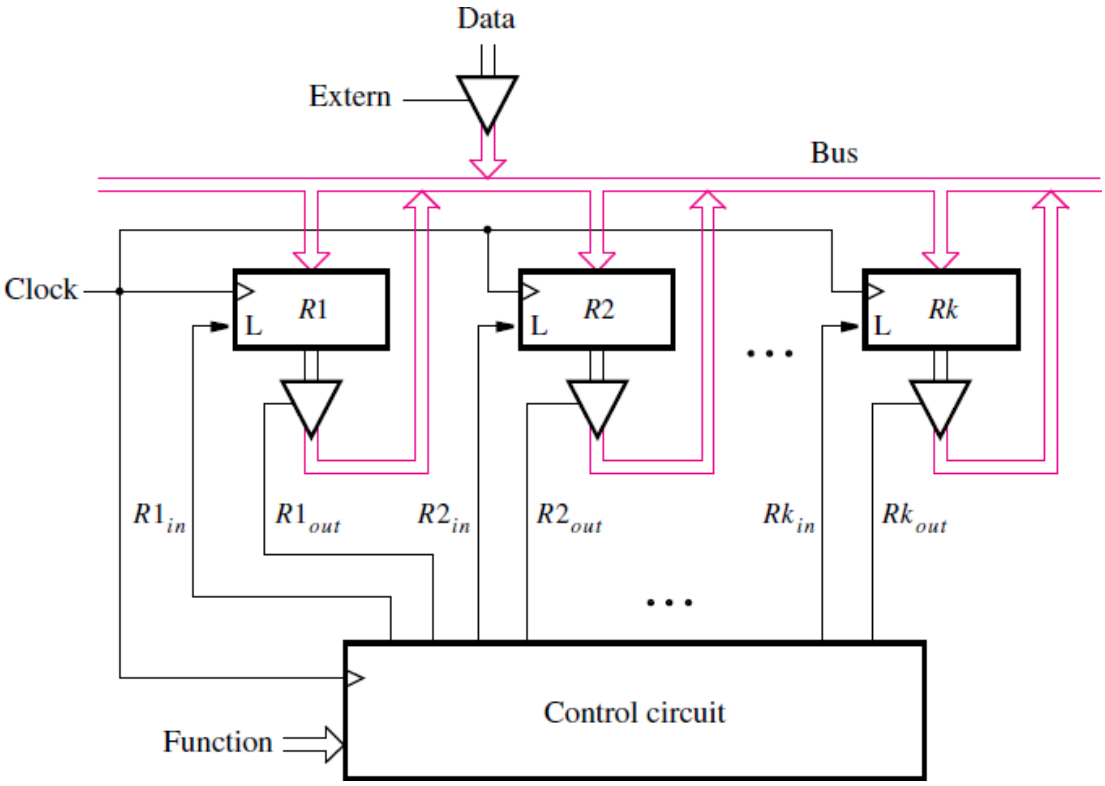


Fig 12.1 A digital system with k registers.

Solved Exercise

Consider a system that has three registers, R1, R2, and R3. The control circuit performs a single function—it swaps the contents of registers R1 and R2, using R3 for temporary storage. The required swapping is done in three steps, each needing one clock cycle. In the first step, the contents of R2 are transferred into R3. Then the contents of R1 are transferred into R2. Finally, the contents of R3, which are the original contents of R2, are transferred into R1. To transfer the contents of one register into another bus is used. The control circuit for this task can be explained in the form of a finite state machine as shown in Figure 12.2. Its state table is shown in Table 12.1

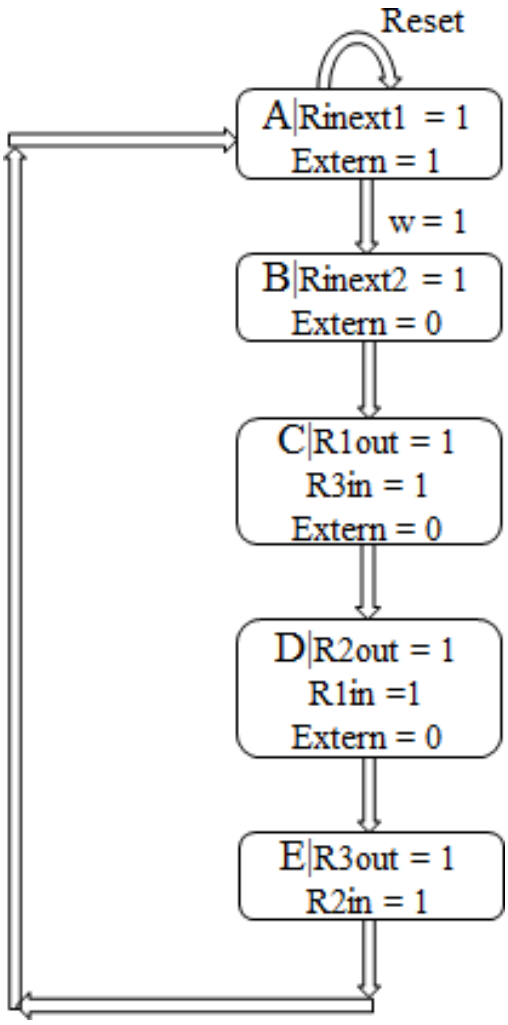


Fig. 12.2

Initially, the bus is loaded with the data when Extern=1. The initial state is A corresponding to load the data from the bus to register R1. To initiate the state transition from state A to state B, an input signal w is made equal to 1. In state A and state B, Extern = 1 to load two different data to the bus and then from the bus to the registers R1 and R2. The states B, C and D change their states if Extern = 0. From B, it goes to C state where R1 is copied to R3. From C it goes to state D where R2 is copied to R1. From D, the next state is E to copy R3 to R2 and an output Done = 1 to indicate that swap is completed. Form E the next state is the initial state A. All transfers are taking place through the bus.

Table 12.1

PS	NS		Outputs								
	Extern=1	Extern=0	Rinext1	Rinext2	R1in	R1out	R2in	R2out	R3in	R3out	done
A	B	A	1	0	0	0	0	0	0	0	0
B	B	C	0	1	0	0	0	0	0	0	0
C	C	D	0	0	0	1	0	0	1	0	0
D	D	E	0	0	1	0	0	1	0	0	0
E	A	A	0	0	0	0	1	0	0	1	1

Verilog code

```
module regn (R, L, Clock, Q);
parameter n = 8;
input [n-1:0] R;
input L, Clock;
output [n-1:0] Q;
reg [n-1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
endmodule // Code for 8-bit register

module swap1 (Resetn, Clock, w, Data, Extern, R1, R2, R3, BusWires, Done);
parameter n = 8;
input Resetn, Clock, w, Extern;
input [n-1:0] Data;
output [n-1:0] BusWires ,R1, R2, R3 ;
reg [n-1:0] BusWires, R1, R2, R3;
output Done;
wire R1in, R1out, R2in, R2out, R3in, R3out, RinExt1, RinExt2;
reg [2:0] y, Y;
parameter [2:0] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100;
// Define the next state combinational circuit for FSM
always @(w or y)
begin
case (y)
A: if (w) Y = B;
```

```
else Y = A;
B: Y = C;
C: Y = D;
D: Y = E;
E: Y = A;
endcase
end
// Define the sequential block for FSM
always @(negedge Resetn or posedge Clock)
begin
if (Resetn == 0) y <= A;
else y <= Y;
end
// Define outputs of FSM
assign RinExt1 = (y == A);
assign RinExt2 = (y == B);
assign R3in = (y == C);
assign R1out = (y == C);
assign R2out = (y == D);
assign R1in = (y == D);
assign R3out = (y == E);
assign R2in = (y == E);
assign Done = (y == E);

always @(Extern or R1out or R2out or R3out)
if (Extern) BusWires = Data;
else if (R1out) BusWires = R1;
```

else if (R2out) BusWires = R2;

else if (R3out) BusWires = R3;

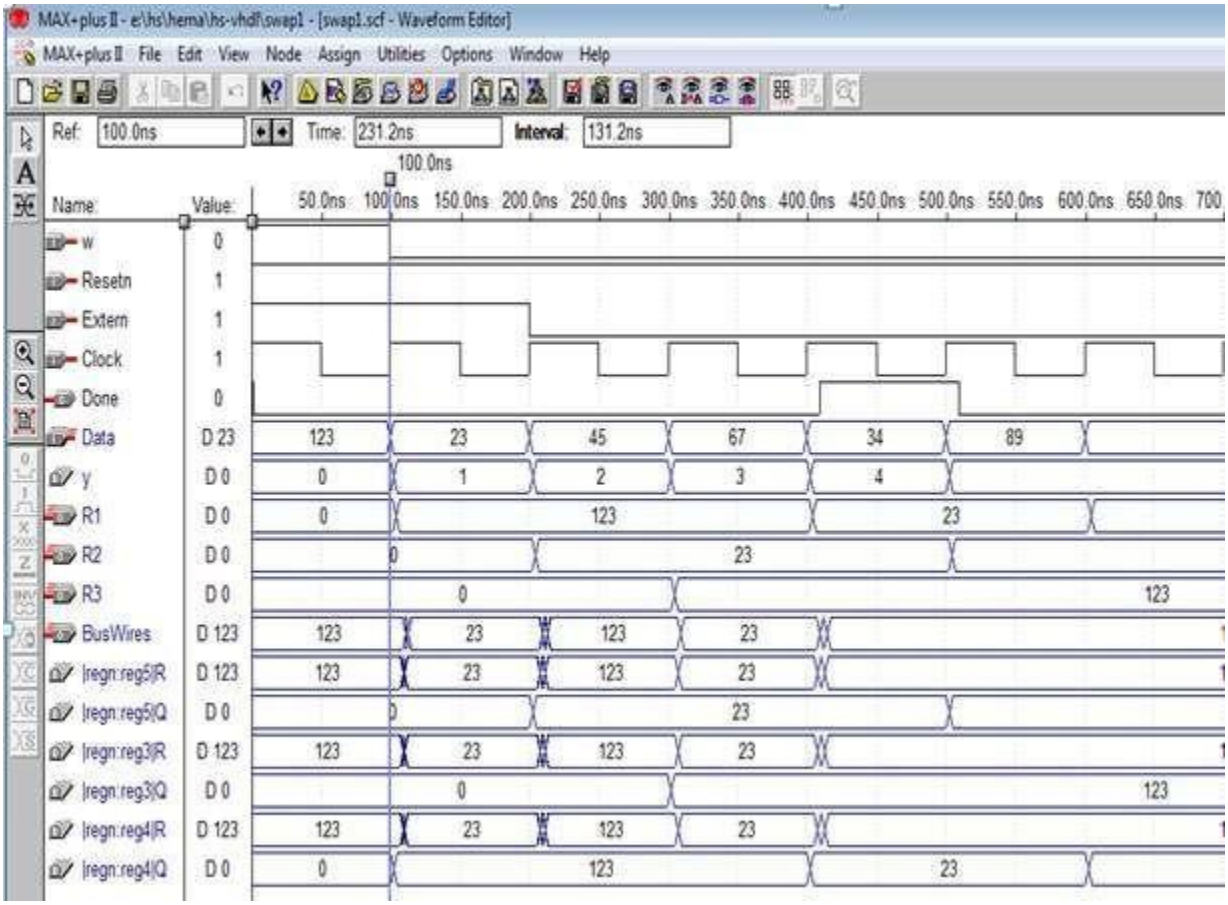
regn reg3 (BusWires, R3in, Clock, R3);

regn reg4 (BusWires, RinExt1 | R1in, Clock, R1);

regn reg5 (BusWires, RinExt2 | R2in, Clock, R2);

endmodule

Output:



Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

References:

1. Stephen Brown and Zvonko Vranesic, “Fundamentals of digital logic with Verilog design”, Tata MGH publishing Co.Ltd., 3rd edition, 2014.
2. M.Morris Mano, “Digital design”, PHI Pvt. Ltd., 2nd edition, 2000