VERILOG

<CODE/>

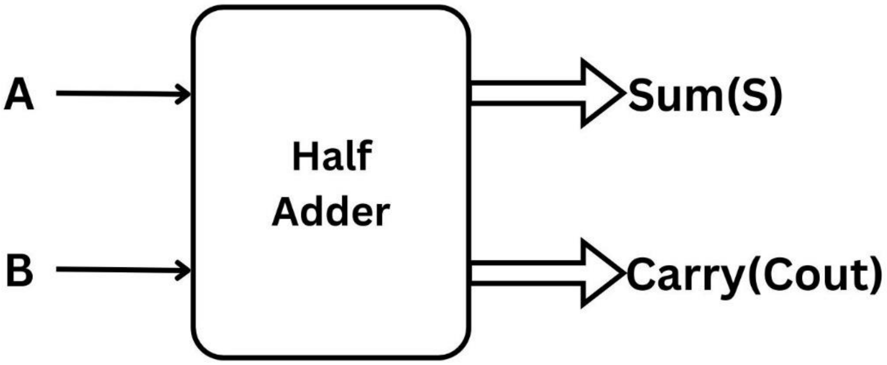
EXAMPLES

JAIRAJ MIRASHI
DESIGN VERIFICATION
ENGINEER

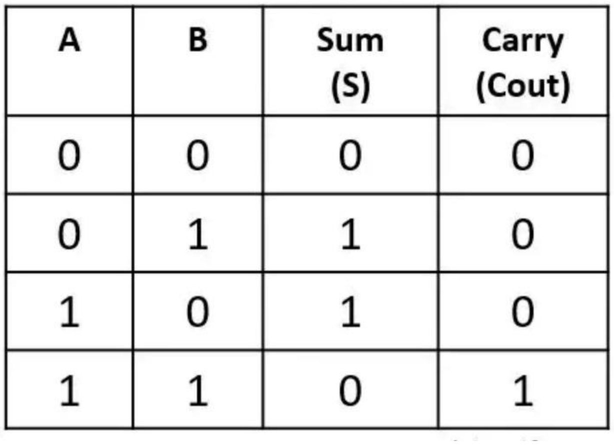
Half Adder

Half Adder is a basic combinational design that can add two single bits and results to a sum and carry bit as an output.

Block Diagram



Truth Table



A	B	Sum (S)	Carry (Cout)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Output:

S = A ^ B

Cout = A · B

Half Adder Verilog Code

```
module half_adder(input a, b, output s, Cout);
  assign S = a ^ b;
  assign Cout = a & b;
endmodule
```

Testbench Code

```
module tb_top;
  reg a, b;
  wire s, c_out;

  half_adder ha(a, b, s, c_out);

  initial begin
    $monitor("At time %0t: a=%b b=%b, sum=%b, carry=%b",$time, a,b,s,c_out);
    a = 0; b = 0;
    #1;
    a = 0; b = 1;
    #1;
    a = 1; b = 0;
    #1;
    a = 1; b = 1;
  end
endmodule
```

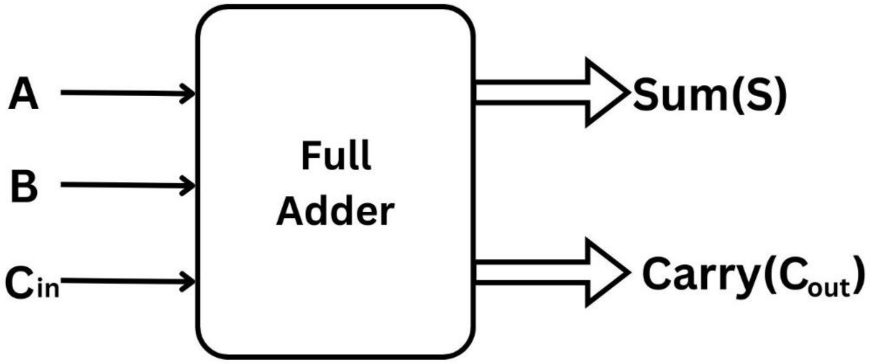
Output:

```
At time 0: a=0 b=0, sum=z, carry=0
At time 1: a=0 b=1, sum=z, carry=0
At time 2: a=1 b=0, sum=z, carry=0
At time 3: a=1 b=1, sum=z, carry=1
```

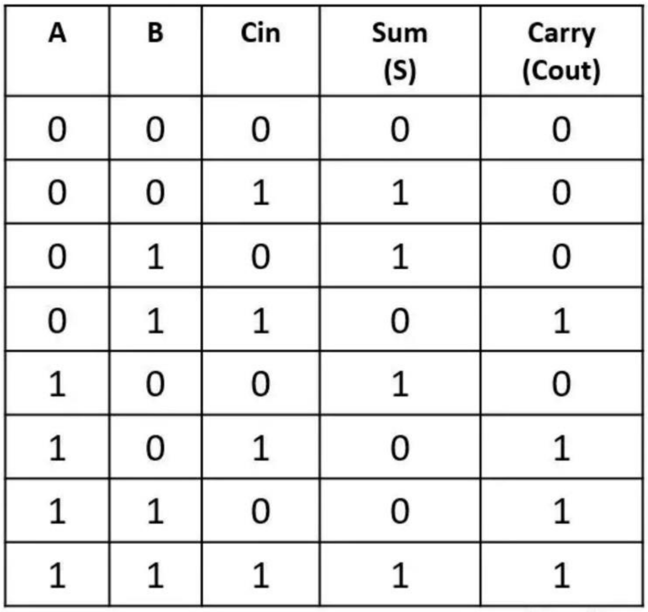
Full Adder

The full adder adds three single-bit input and produce two single-bit output. Thus, it is useful when an extra carry bit is available from the previously generated result.

Block Diagram



Truth Table



A	B	Cin	Sum (S)	Carry (Cout)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Output:

S = A ^ B ^ Cin

Cout = A·B + B·Cin + A·Cin

Full Adder Verilog Code

```
module full_adder(input a, b, cin, output S, Cout);
  assign S = a ^ b ^ cin;
  assign Cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

Testbench Code

```
module tb_top;
  reg a, b, c;
  wire s, c_out;

  full_adder fa(a, b, c, s, c_out);

  initial begin
    $monitor("At  time  %0t:  a=%b  b=%b,  cin=%b,  sum=%b,  carry=%b",$time,
a,b,c,s,c_out);
    a = 0; b = 0; c = 0; #1;
    a = 0; b = 0; c = 1; #1;
    a = 0; b = 1; c = 0; #1;
    a = 0; b = 1; c = 1; #1;
    a = 1; b = 0; c = 0; #1;
    a = 1; b = 0; c = 1; #1;
    a = 1; b = 1; c = 0; #1;
    a = 1; b = 1; c = 1;
  end
endmodule
```

Output:

```
At time 0: a=0 b=0, cin=0, sum=0, carry=0
At time 1: a=0 b=0, cin=1, sum=1, carry=0
At time 2: a=0 b=1, cin=0, sum=1, carry=0
At time 3: a=0 b=1, cin=1, sum=0, carry=1
At time 4: a=1 b=0, cin=0, sum=1, carry=0
At time 5: a=1 b=0, cin=1, sum=0, carry=1
At time 6: a=1 b=1, cin=0, sum=0, carry=1
At time 7: a=1 b=1, cin=1, sum=1, carry=1
```

Full Adder using Half Adder Verilog Code

```
module half_addr(input a, b, output s, c);
  assign s = a^b;
  assign c = a & b;
endmodule

module full_adder(input a, b, cin, output s_out, c_out);
  wire s, c0, c1;
  half_addr HA1 (a, b, s, c0);
  half_addr HA2 (s, cin, s_out, c1);

  assign c_out = c0 | c1;
endmodule
```

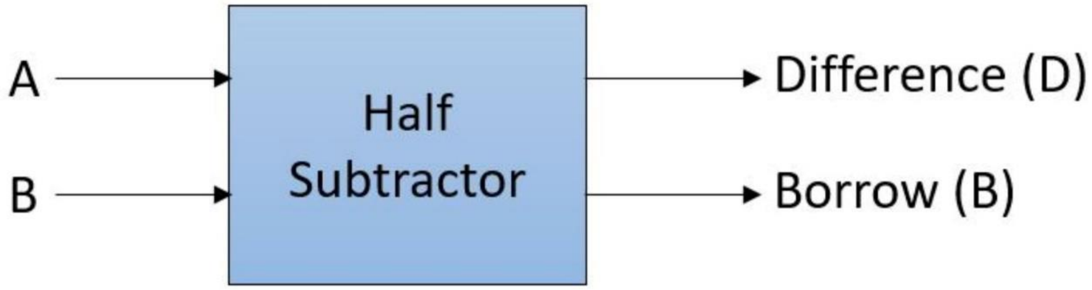
Output:

```
At time 0: a=0 b=0, cin=0, sum=0, carry=0
At time 1: a=0 b=0, cin=1, sum=1, carry=0
At time 2: a=0 b=1, cin=0, sum=1, carry=0
At time 3: a=0 b=1, cin=1, sum=0, carry=1
At time 4: a=1 b=0, cin=0, sum=1, carry=0
At time 5: a=1 b=0, cin=1, sum=0, carry=1
At time 6: a=1 b=1, cin=0, sum=0, carry=1
At time 7: a=1 b=1, cin=1, sum=1, carry=1
```

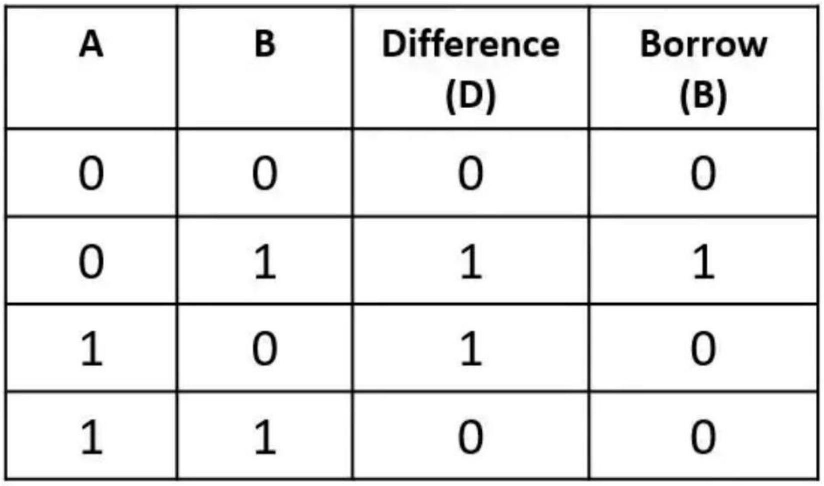
Half Subtractor

The half subtractor works opposite to the half adder as it substracts two single bits and results in a difference bit and borrow bit as an output.

Block Diagram



Truth Table



A	B	Difference (D)	Borrow (B)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Output:

D = A ^ B

B = A’ · B

Half Subtractor Verilog Code

```
module half_subtractor(input a, b, output D, B);
  assign D = a ^ b;
  assign B = ~a & b;
endmodule
```

Testbench Code

```
module tb_top;
  reg a, b;
  wire D, B;

  half_subtractor hs(a, b, D, B);

  initial begin
    $monitor("At time %0t: a=%b b=%b, difference=%b, borrow=%b",$time,
a,b,D,B);
    a = 0; b = 0;
    #1;
    a = 0; b = 1;
    #1;
    a = 1; b = 0;
    #1;
    a = 1; b = 1;
  end
endmodule
```

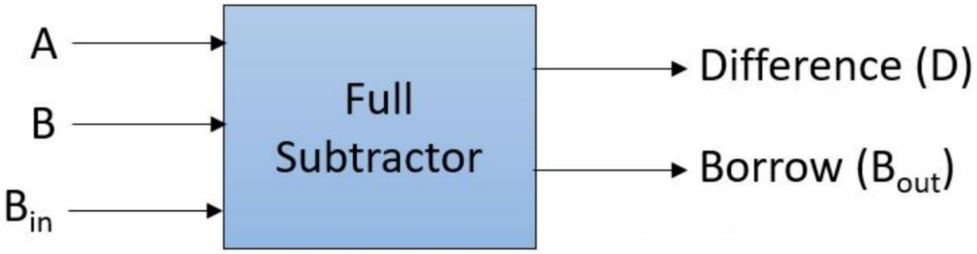
Output:

```
At time 0: a=0 b=0, difference=0, borrow=0
At time 1: a=0 b=1, difference=1, borrow=1
At time 2: a=1 b=0, difference=1, borrow=0
At time 3: a=1 b=1, difference=0, borrow=0
```

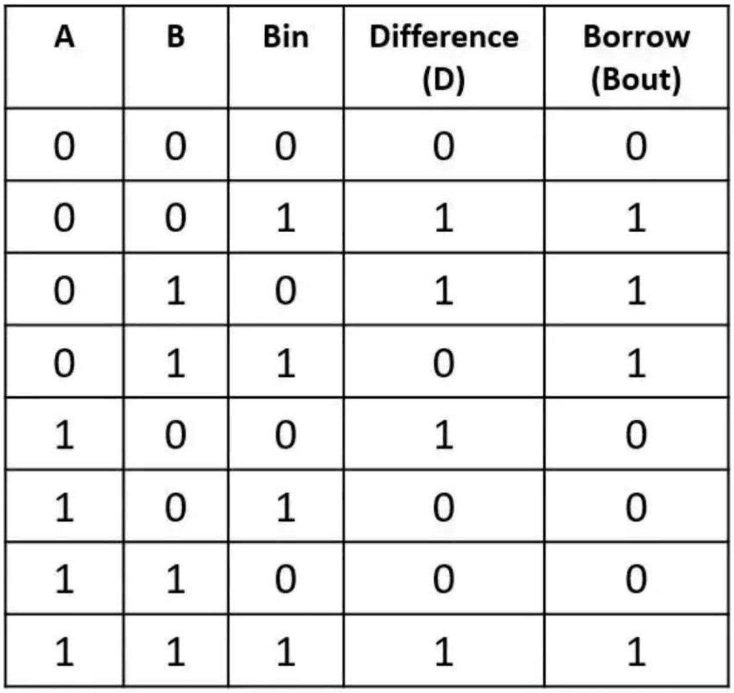
Full Subtractor

A full subtractor is designed to accommodate the extra borrow bit from the previous stage. Thus it has three single-bit inputs and produces two single-bit outputs.

Block Diagram



Truth Table



A	B	Bin	Difference (D)	Borrow (Bout)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Output

D = A ^ B ^ Bin

Bout = A’ · B + (A ^ B)’ · Bin

Full Subtractor Verilog Code

```
module full_subtractor(input a, b, Bin, output D, Bout);
  assign D = a ^ b ^ Bin;
  assign Bout = (~a & b) | (~(a ^ b) & Bin);
endmodule
```

Testbench Code

```
module tb_top;
  reg a, b, Bin;
  wire D, Bout;

  full_subtractor fs(a, b, Bin, D, Bout);

initial begin
  $monitor("At time %0t: a=%b b=%b, Bin=%b, difference=%b, borrow=%b",$time,
a,b,Bin,D,Bout);
    a = 0; b = 0; Bin = 0; #1;
    a = 0; b = 0; Bin = 1; #1;
    a = 0; b = 1; Bin = 0; #1;
    a = 0; b = 1; Bin = 1; #1;
    a = 1; b = 0; Bin = 0; #1;
    a = 1; b = 0; Bin = 1; #1;
    a = 1; b = 1; Bin = 0; #1;
    a = 1; b = 1; Bin = 1;
  end
endmodule
```

Output:

```
At time 0: a=0 b=0, Bin=0, difference=0, borrow=0
At time 1: a=0 b=0, Bin=1, difference=1, borrow=1
At time 2: a=0 b=1, Bin=0, difference=1, borrow=1
At time 3: a=0 b=1, Bin=1, difference=0, borrow=1
At time 4: a=1 b=0, Bin=0, difference=1, borrow=0
At time 5: a=1 b=0, Bin=1, difference=0, borrow=0
At time 6: a=1 b=1, Bin=0, difference=0, borrow=0
At time 7: a=1 b=1, Bin=1, difference=1, borrow=1
```

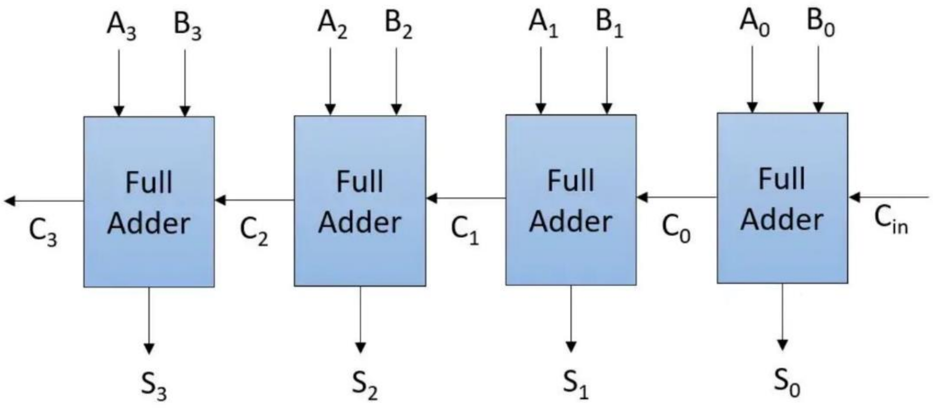
Ripple Carry Adder

Full adder can add single bit two inputs and extra carry bit generated from its previous stage. To add multiple ‘n’ bits binary sequence, multiples cascaded full adders can be used which can generate a carry bit and be applied to the next stage full adder as an input till the last stage of full adder. This appears as carry-bit ripples to the next stage, hence it is known as “Ripple carry adder”.

Ripple carry adder delay computation

Worst case delay = [(n-1) full adder * carry propagation delay of each adder] + sum propagation delay of each full adder

Block Diagram



4-Bit Ripple Carry Adder

Ripple Carry Adder Verilog Code

```
module full_adder(
  input a, b, cin,
  output sum, cout
);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module ripple_carry_adder #(parameter SIZE = 4) (
  input [SIZE-1:0] A, B,
  input Cin,
  output [SIZE-1:0] S, Cout);

  genvar g;

  full_adder fa0(A[0], B[0], Cin, S[0], Cout[0]);
  generate  // This will instantiate full_adder SIZE-1 times
    for(g = 1; g<SIZE; g++) begin
      full_adder fa(A[g], B[g], Cout[g-1], S[g], Cout[g]);
    end
  endgenerate
endmodule
```

Testbench Code

```
module RCA_TB;
  wire [3:0] S, Cout;
  reg [3:0] A, B;
  reg Cin;
  wire[4:0] add;

  ripple_carry_adder rca(A, B, Cin, S, Cout);
  assign add = {Cout[3], S};

  initial begin
    $monitor("A = %b: B = %b, Cin = %b --> S = %b, Cout[3] = %b, Addition =
%0d", A, B, Cin, S, Cout[3], add);
    A = 1; B = 0; Cin = 0; #3;
    A = 2; B = 4; Cin = 1; #3;
    A = 4'hb; B = 4'h6; Cin = 0; #3;
    A = 5; B = 3; Cin = 1; #3;
    $finish;
  end

  initial begin
    $dumpfile("waves.vcd");
    $dumpvars;
  end
endmodule
```

Output:

```
A = 0001: B = 0000, Cin = 0 --> S = 0001, Cout[3] = 0, Addition = 1
A = 0010: B = 0100, Cin = 1 --> S = 0111, Cout[3] = 0, Addition = 7
A = 1011: B = 0110, Cin = 0 --> S = 0001, Cout[3] = 1, Addition = 17
A = 0101: B = 0011, Cin = 1 --> S = 1001, Cout[3] = 0, Addition = 9
```

4-bit Adder Subtractor

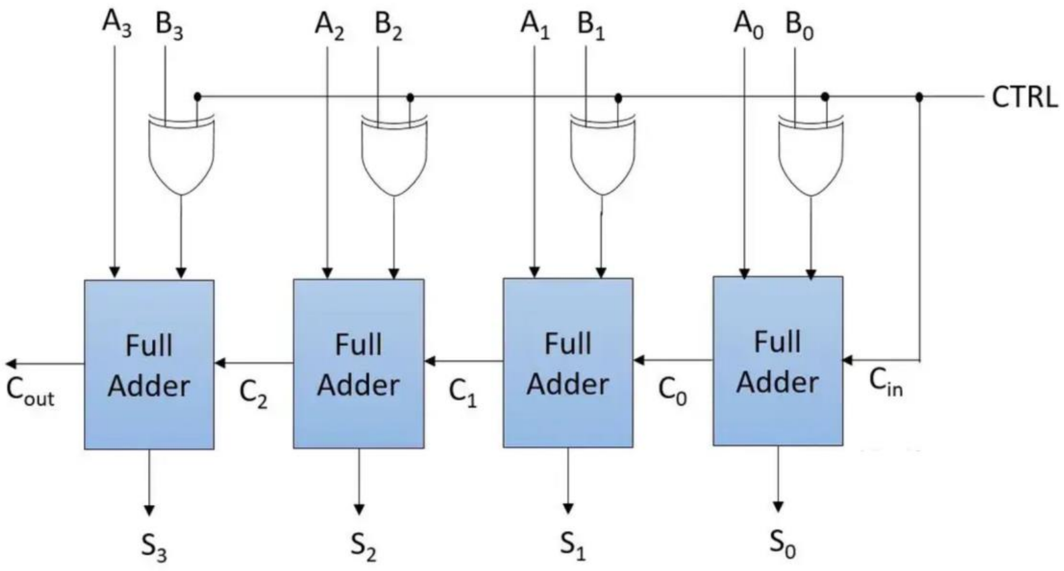
When control bit CTRL decides whether to do addition or subtraction for two 4 bit integers A and B. The signal CTRL is attached to one of the inputs of the XOR gate and another input is connected to B.

CTRL = 0: Addition is performed (A + B)

CTRL = 1: Subtraction is performed. XOR gate output becomes a complement of B. So, addition is performed as A + (-B)

If Cout = 1, ignore it and sum S is the answer and if it is 0, Sum S represents 2's complement of the answer and the number is negative.

Block Diagram



4-Bit Adder Subtractor

4 bit Adder Subtractor Verilog Code

```
module full_adder(
  input a, b, cin,
  output sum, cout
);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module ripple_carry_adder_subtractor #(parameter SIZE = 4) (
  input [SIZE-1:0] A, B,
  input CTRL,
  output [SIZE-1:0] S, Cout);
  bit [SIZE-1:0] Bc;
  genvar g;

  assign Bc[0] = B[0] ^ CTRL;

  full_adder fa0(A[0], Bc[0], CTRL, S[0], Cout[0]);
  generate  // This will instantial full_adder SIZE-1 times
    for(g = 1; g<SIZE; g++) begin
      assign Bc[g] = B[g] ^ CTRL;
      full_adder fa(A[g], Bc[g], Cout[g-1], S[g], Cout[g]);
    end
  endgenerate
endmodule
```

Testbench Code

```
module RCAS_TB;
  wire [3:0] S, Cout;
  reg [3:0] A, B;
  reg ctrl;

  ripple_carry_adder_subtractor rcas(A, B, ctrl, S, Cout);

  initial begin
    $monitor("CTRL=%b: A = %b, B = %b --> S = %b, Cout[3] = %b", ctrl, A, B,
S, Cout[3]);
    ctrl = 0;
    A = 1; B = 0;
    #3 A = 2; B = 4;
    #3 A = 4'hb; B = 4'h6;
    #3 A = 5; B = 3;
    ctrl = 1;
    A = 1; B = 0;
    #3 A = 2; B = 4;
    #3 A = 4'hb; B = 4'h6;
    #3 A = 5; B = 3;
    #3 $finish;
  end

  initial begin
```

```
    $dumpfile("waves.vcd");
    $dumpvars;
  end
endmodule
```

Output:

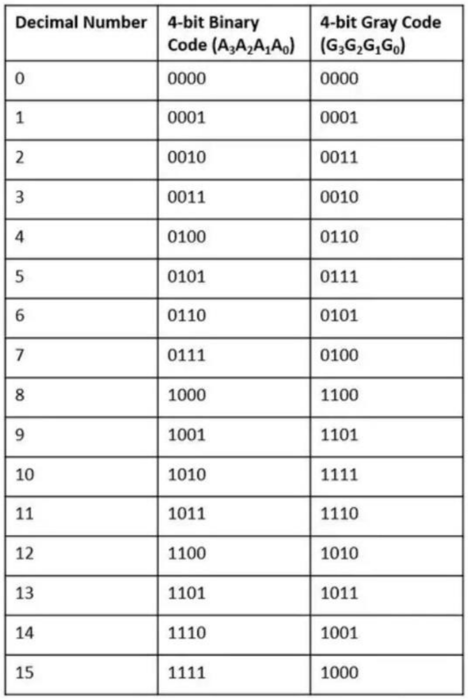
```
CTRL=0: A = 0001, B = 0000 --> S = 0001, Cout[3] = 0
CTRL=0: A = 0010, B = 0100 --> S = 0110, Cout[3] = 0
CTRL=0: A = 1011, B = 0110 --> S = 0001, Cout[3] = 1
CTRL=1: A = 0001, B = 0000 --> S = 0001, Cout[3] = 1
CTRL=1: A = 0010, B = 0100 --> S = 1110, Cout[3] = 0
CTRL=1: A = 1011, B = 0110 --> S = 0101, Cout[3] = 1
CTRL=1: A = 0101, B = 0011 --> S = 0010, Cout[3] = 1
```

Binary to Gray Code Converter

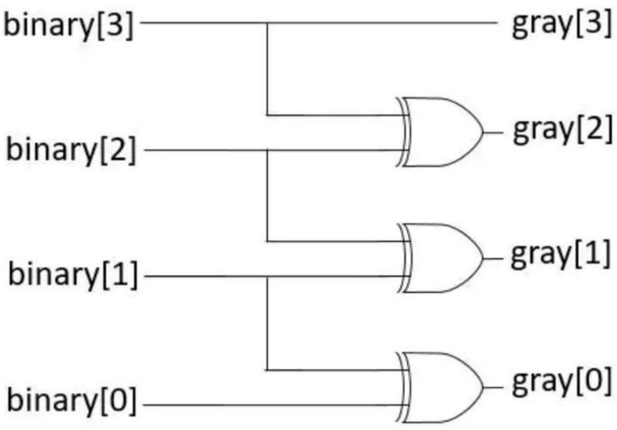
Gray Code is similar to binary code except its successive number differs by only a single bit. Hence, it has importance in communication systems as it minimizes error occurrence. They are also useful in rotary, optical encoders, data acquisition systems, etc.

Let’s see binary numbers and their equivalent gray code in the below table.

Truth Table



Decimal Number	4-bit Binary Code ($A_3A_2A_1A_0$)	4-bit Gray Code ($G_3G_2G_1G_0$)
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000



Binary to Gray Code Converter

Binary to Gray Code Converter Verilog Code

```
module b2g_converter #(parameter WIDTH=4) (input [WIDTH-1:0] binary, output
[WIDTH-1:0] gray);
  genvar i;
  generate
    for(i=0;i<WIDTH-1;i++) begin
      assign gray[i] = binary[i] ^ binary[i+1];
    end
  endgenerate

  assign gray[WIDTH-1] = binary[WIDTH-1];
endmodule
```

Testbench Code

```
module TB;
  reg [3:0] binary, gray;
  b2g_converter b2g(binary, gray);

  initial begin
    $monitor("Binary = %b --> Gray = %b", binary, gray);
    binary = 4'b1011; #1;
    binary = 4'b0111; #1;
    binary = 4'b0101; #1;
    binary = 4'b1100; #1;
    binary = 4'b1111;
  end
endmodule
```

Output:

```
Binary = 1011 --> Gray = 1110
Binary = 0111 --> Gray = 0100
Binary = 0101 --> Gray = 0111
Binary = 1100 --> Gray = 1010
Binary = 1111 --> Gray = 1000
```

Compact implementation is mentioned below:

Alternative Verilog implementation

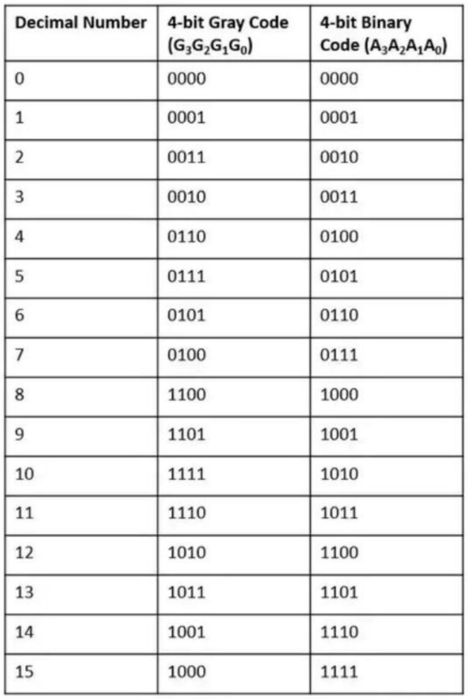
```
module b2g_converter #(parameter WIDTH=4) (input [WIDTH-1:0] binary, output
[WIDTH-1:0] gray);
  assign gray = binary ^ (binary >> 1);
endmodule
```

Gray to Binary Code Converter

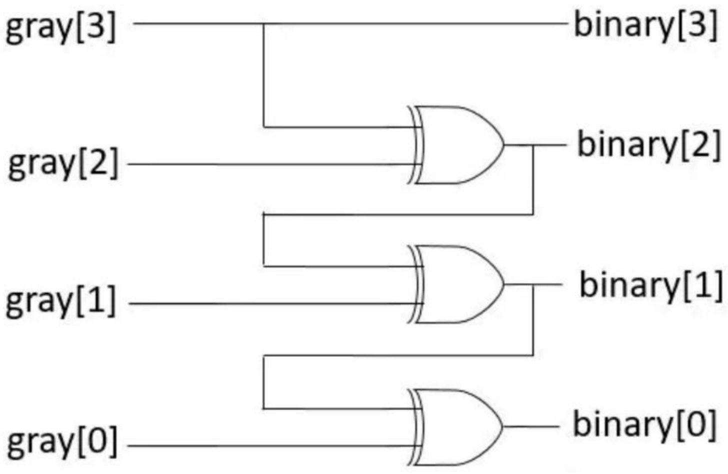
Gray code has its own applications and we have seen how binary code is converted to the Gray code in the previous post Binary to Gray Code Converter.

Let’s Gray code to Binary code implementation in Verilog.

Truth Table



Decimal Number	4-bit Gray Code ($G_3G_2G_1G_0$)	4-bit Binary Code ($A_3A_2A_1A_0$)
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111



Gray to Binary Code Converter

Gray to Binary Code Converter Verilog code

```
module g2b_converter #(parameter WIDTH=4) (input [WIDTH-1:0] gray, output
[WIDTH-1:0] binary);
  /*
  assign binary[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0];
  assign binary[1] = gray[3] ^ gray[2] ^ gray[1];
  assign binary[2] = gray[3] ^ gray[2];
  assign binary[3] = gray[3];
  */
  // OR
  genvar i;
  generate
    for(i=0;i<WIDTH;i++) begin
      assign binary[i] = ^(gray >> i);
    end
  endgenerate
endmodule
```

Testbench Code

```
module TB;
  reg [3:0] binary, gray;
  g2b_converter g2b(gray, binary);

  initial begin
    $monitor("Gray = %b --> Binary = %b", gray, binary);
    gray = 4'b1110; #1;
    gray = 4'b0100; #1;
    gray = 4'b0111; #1;
    gray = 4'b1010; #1;
    gray = 4'b1000;
  end
endmodule
```

Output:

```
Gray = 1110 --> Binary = 1011
Gray = 0100 --> Binary = 0111
Gray = 0111 --> Binary = 0101
Gray = 1010 --> Binary = 1100
Gray = 1000 --> Binary = 1111
```

Multiplexer

A multiplexer (MUX) is a combinational circuit that connects any one input line (out of multiple N lines) to the single output line based on its control input signal (or selection lines)

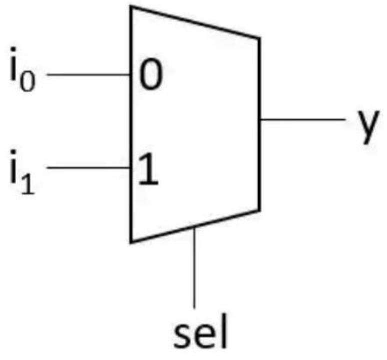
Usually, for ‘n’ selection lines, there are N = 2^n input lines.

Nomenclature: N:1 denotes it has ‘N’ input lines and one output line.

2:1 Multiplexer

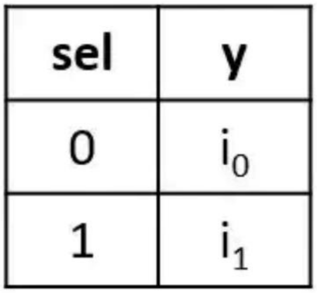
2:1 MUX has 2 input lines and one select line.

Block Diagram



2:1 MUX

Truth Table



sel	y
0	i_0
1	i_1

2:1 MUX Verilog Code

```
module mux_2_1(
  input sel,
  input i0, i1,
  output y);

  assign y = sel ? i1 : i0;
endmodule
```

Testbench Code

```
module mux_tb;
  reg i0, i1, sel;
  wire y;

  mux_2_1 mux(sel, i0, i1, y);
  initial begin
    $monitor("sel = %h: i0 = %h, i1 = %h --> y = %h", sel, i0, i1, y);
    i0 = 0; i1 = 1;
    sel = 0;
    #1;
    sel = 1;
  end
endmodule
```

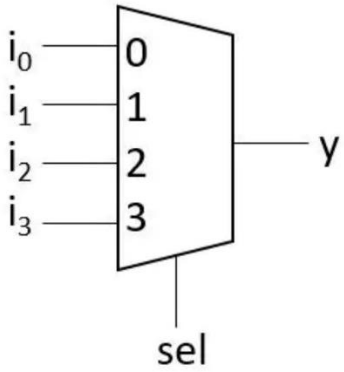
Output:

```
sel = 0: i0 = 0, i1 = 1 --> y = 0
sel = 1: i0 = 0, i1 = 1 --> y = 1
```

4:1 Multiplexer

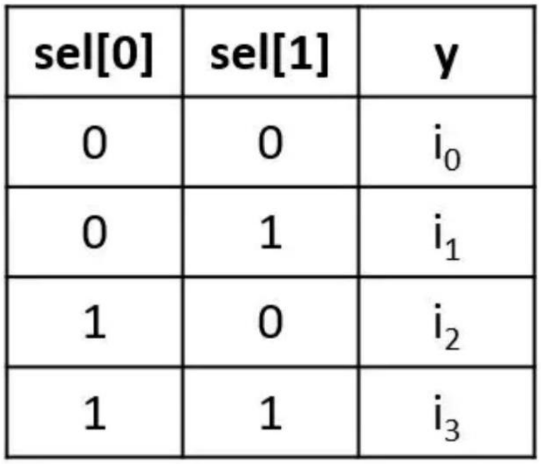
4:1 has 4 input lines and two select lines.

Block Diagram



4:1 MUX

Truth Table



sel[0]	sel[1]	y
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

4:1 MUX Verilog Code

```
module mux_example(
  input [1:0] sel,
  input  i0,i1,i2,i3,
  output reg y);

  always @(*) begin
    case(sel)
      2'h0: y = i0;
      2'h1: y = i1;
      2'h2: y = i2;
      2'h3: y = i3;
      default: $display("Invalid sel input");
    endcase
  end
endmodule
```

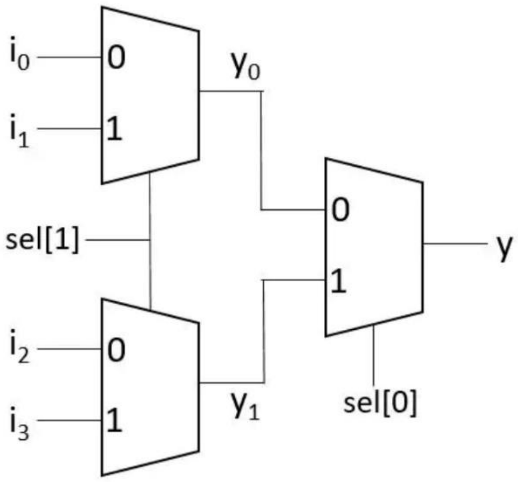
Testbench Code

```
module tb;
  reg [1:0] sel;
  reg i0,i1,i2,i3;
  wire y;

  mux_example mux(sel, i0, i1, i2, i3, y);

  initial begin
    $monitor("sel = %b -> i3 = %0b, i2 = %0b ,i1 = %0b, i0 = %0b -> y = %0b",
sel,i3,i2,i1,i0, y);
    {i3,i2,i1,i0} = 4'h5;
    repeat(6) begin
      sel = $random;
      #5;
    end
  end
endmodule
```

Output:

```
sel = 00 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1
sel = 01 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel = 11 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel = 01 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
```

4:1 MUX using 2:1 MUXes
Block Diagram



4:1 MUX using 2:1 MUXes

4:1 MUX using 2:1 MUXes Verilog Code

```
module mux_2_1(
  input sel,
  input i0, i1,
  output y);

  assign y = sel ? i1 : i0;
endmodule

module mux_4_1(
  input sel0, sel1,
  input  i0,i1,i2,i3,
  output reg y);

  wire y0, y1;

  mux_2_1 m1(sel1, i2, i3, y1);
  mux_2_1 m2(sel1, i0, i1, y0);
  mux_2_1 m3(sel0, y0, y1, y);
endmodule
```

Testbench Code

```
module tb;
  reg sel0, sel1;
  reg i0,i1,i2,i3;
  wire y;

  mux_4_1 mux(sel0, sel1, i0, i1, i2, i3, y);
  initial begin
    $monitor("sel0=%b, sel1=%b -> i3 = %0b, i2 = %0b ,i1 = %0b, i0 = %0b -> y
= %0b", sel0,sel1,i3,i2,i1,i0, y);
    {i3,i2,i1,i0} = 4'h5;

    repeat(6) begin
      {sel0, sel1} = $random;
      #5;
    end
  end
endmodule
```

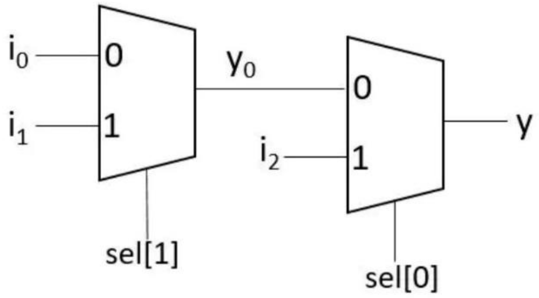
Output:

```
sel0=0, sel1=0 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1
sel0=0, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel0=1, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel0=0, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
```

3:1 Multiplexer

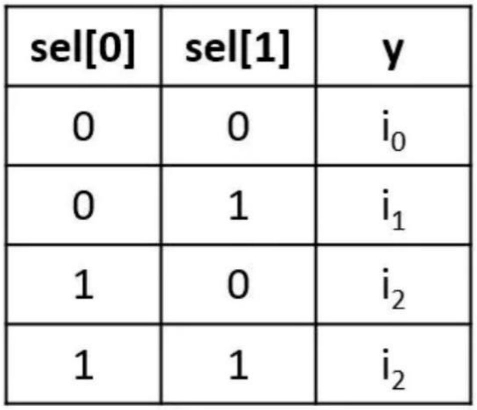
In 3:1 MUX, for two select inputs (sel = 2’b01 and sel = 2’b11), same input (i2) will be driven to the output y.

Block Diagram



3:1 MUX using 2:1 MUXes

Truth Table



sel[0]	sel[1]	y
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_2

3:1 MUX Verilog Code

```
module mux_2_1(
  input sel,
  input i0, i1,
  output y);

  assign y = sel ? i1 : i0;
endmodule

module mux_3_1(
  input sel0, sel1,
  input  i0,i1,i2,i3,
  output reg y);

  wire y0, y1;

  mux_2_1 m1(sel1, i0, i1, y0);
  mux_2_1 m2(sel0, y0, i2, y);
endmodule
```

Testbench Code

```
module tb;
  reg sel0, sel1;
  reg i0,i1,i2,i3;
  wire y;

  mux_3_1 mux(sel0, sel1, i0, i1, i2, i3, y);

  initial begin
    $monitor("sel0=%b, sel1=%b -> i3 = %0b, i2 = %0b ,i1 = %0b, i0 = %0b -> y
= %0b", sel0,sel1,i3,i2,i1,i0, y);
    {i3,i2,i1,i0} = 4'h5;

    repeat(8) begin
      {sel0, sel1} = $random;
      #5;
    end
  end
endmodule
```

Output:

```
sel0=0, sel1=0 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1
sel0=0, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel0=1, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1
sel0=0, sel1=1 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0
sel0=1, sel1=0 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1
```

Demultiplexer

A demultiplexer (DEMUX) is a combinational circuit that works exactly opposite to a multiplexer. A DEMUX has a single input line that connects to any one of the output lines based on its control input signal (or selection lines)

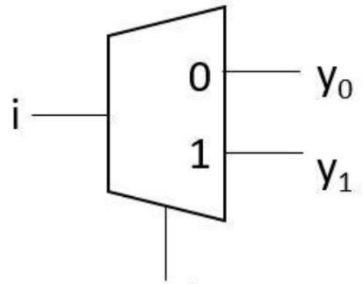
Usually, for ‘n’ selection lines, there are N = 2^n output lines.

Nomenclature: 1:N denotes one input line and ‘N’ output lines.

1:2 Demultiplexer

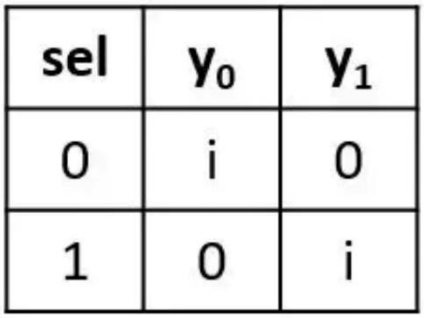
1:2 DEMUX has one select line and 2 output lines.

Block Diagram



1:2 DEMUX

Truth Table



sel	y_0	y_1
0	i	0
1	0	i

1:2 Demux Verilog Code

```
module demux_2_1(
  input sel,
  input i,
  output y0, y1);

  assign {y0,y1} = sel?{1'b0,i}: {i,1'b0};
endmodule
```

Testbench Code

```
module demux_tb;
  reg sel, i;
  wire y0, y1;

  demux_2_1 demux(sel, i, y0, y1);
  initial begin
    $monitor("sel = %h: i = %h --> y0 = %h, y1 = %h", sel, i, y0, y1);
    sel=0; i=0; #1;
    sel=0; i=1; #1;
    sel=1; i=0; #1;
    sel=1; i=1; #1;
  end
endmodule
```

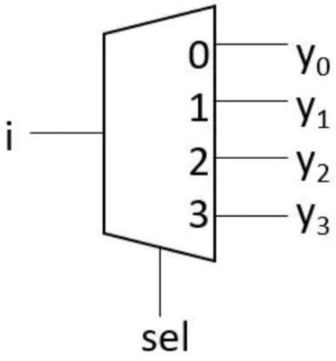
Output:

```
sel = 0: i = 0 --> y0 = 0, y1 = 0
sel = 0: i = 1 --> y0 = 1, y1 = 0
sel = 1: i = 0 --> y0 = 0, y1 = 0
sel = 1: i = 1 --> y0 = 0, y1 = 1
```

1:4 Demultiplexer

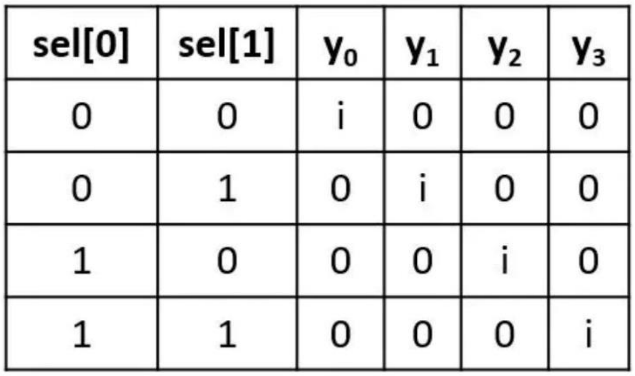
1:4 DEMUX has one select line and 4 output lines.

Block Diagram



1:4 DEMUX

Truth Table



sel[0]	sel[1]	y_0	y_1	y_2	y_3
0	0	i	0	0	0
0	1	0	i	0	0
1	0	0	0	i	0
1	1	0	0	0	i

1:4 Demux Verilog Code

```
module demux_1_4(
  input [1:0] sel,
  input  i,
  output reg y0,y1,y2,y3);

  always @(*) begin
    case(sel)
      2'h0: {y0,y1,y2,y3} = {i,3'b0};
      2'h1: {y0,y1,y2,y3} = {1'b0,i,2'b0};
      2'h2: {y0,y1,y2,y3} = {2'b0,i,1'b0};
      2'h3: {y0,y1,y2,y3} = {3'b0,i};
      default: $display("Invalid sel input");
    endcase
  end
endmodule
```

Testbench Code

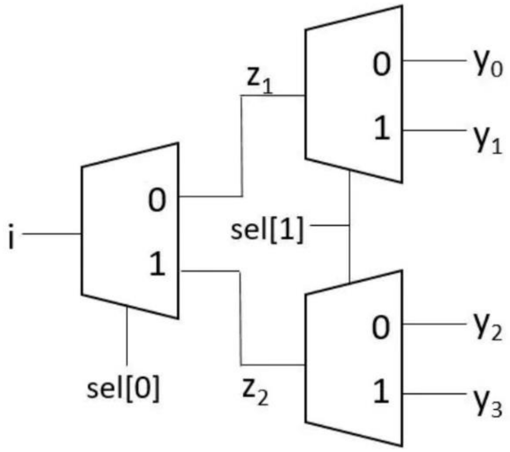
```
module tb;
  reg [1:0] sel;
  reg i;
  wire y0,y1,y2,y3;

  demux_1_4 demux(sel, i, y0, y1, y2, y3);

  initial begin
    $monitor("sel = %b, i = %b -> y0 = %0b, y1 = %0b ,y2 = %0b, y3 = %0b",
sel,i, y0,y1,y2,y3);
    sel=2'b00; i=0; #1;
    sel=2'b00; i=1; #1;
    sel=2'b01; i=0; #1;
    sel=2'b01; i=1; #1;
    sel=2'b10; i=0; #1;
    sel=2'b10; i=1; #1;
    sel=2'b11; i=0; #1;
    sel=2'b11; i=1; #1;
  end
endmodule
```

Output:

```
sel = 00, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel = 00, i = 1 -> y0 = 1, y1 = 0 ,y2 = 0, y3 = 0
sel = 01, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel = 01, i = 1 -> y0 = 0, y1 = 1 ,y2 = 0, y3 = 0
sel = 10, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel = 10, i = 1 -> y0 = 0, y1 = 0 ,y2 = 1, y3 = 0
sel = 11, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel = 11, i = 1 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 1
```

1:4 DEMUX using 1:2 DEMUXes

Block Diagram



1:4 DEMUX using 1:2 DEMUXes

1:4 DEMUX using 1:2 DEMUXes Verilog Code

```
module demux_2_1(
  input sel,
  input i,
  output y0, y1);

  assign {y0,y1} = sel?{1'b0,i}: {i,1'b0};
endmodule

module demux_1_4(
  input sel0, sel1,
  input  i,
  output reg y0, y1, y2, y3);

  wire z1,z2;

  demux_2_1 d1(sel0, i, z1, z2);
  demux_2_1 d2(sel1, z1, y0, y1);
  demux_2_1 d3(sel1, z2, y2, y3);
endmodule
```

Testbench Code

```
module tb;
  reg sel0, sel1;
  reg i;
  wire y0,y1,y2,y3;

  demux_1_4 demux(sel0, sel1, i, y0, y1, y2, y3);

  initial begin
```

```
    $monitor("sel0 = %b, sel1 = %b, i = %b -> y0 = %0b, y1 = %0b ,y2 = %0b,
y3 = %0b", sel0, sel1, i, y0,y1,y2,y3);
    sel0=0; sel1=0; i=0; #1;
    sel0=0; sel1=0; i=1; #1;
    sel0=0; sel1=1; i=0; #1;
    sel0=0; sel1=1; i=1; #1;
    sel0=1; sel1=0; i=0; #1;
    sel0=1; sel1=0; i=1; #1;
    sel0=1; sel1=1; i=0; #1;
    sel0=1; sel1=1; i=1; #1;
  end
endmodule
```

Output:

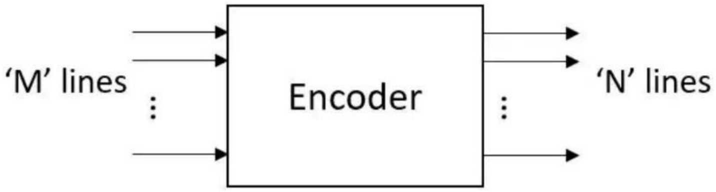
```
sel0 = 0, sel1 = 0, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel0 = 0, sel1 = 0, i = 1 -> y0 = 1, y1 = 0 ,y2 = 0, y3 = 0
sel0 = 0, sel1 = 1, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel0 = 0, sel1 = 1, i = 1 -> y0 = 0, y1 = 1 ,y2 = 0, y3 = 0
sel0 = 1, sel1 = 0, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel0 = 1, sel1 = 0, i = 1 -> y0 = 0, y1 = 0 ,y2 = 1, y3 = 0
sel0 = 1, sel1 = 1, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0
sel0 = 1, sel1 = 1, i = 1 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 1
```

Encoder

An encoder basically converts ‘M’ input lines (can be decimal, hex, octal, etc) to coded ‘N’ output lines. Various encoders can be designed like decimal-to-binary encoders, octal-to-binary encoders, decimal- to-BCD encoders, etc

An encoder basically shrinks receiving many input data lines to the output lines. This is helpful in reducing input data lines for further processing.

Block Diagram



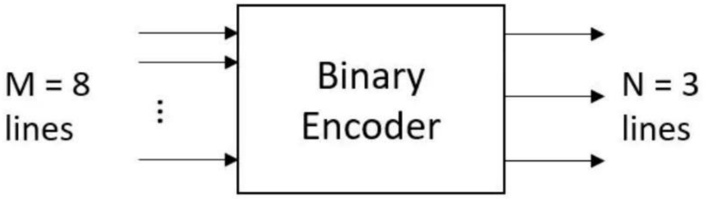
Binary Encoder

The binary encoder converts M (=2^n) input lines to N (=n) coded binary code. It is also known as a digital encoder. A single input line must be high for valid coded output, otherwise, the output line will be invalid. To address this limitation, the priority encoder prioritizes each input line when multiple input lines are set to high.

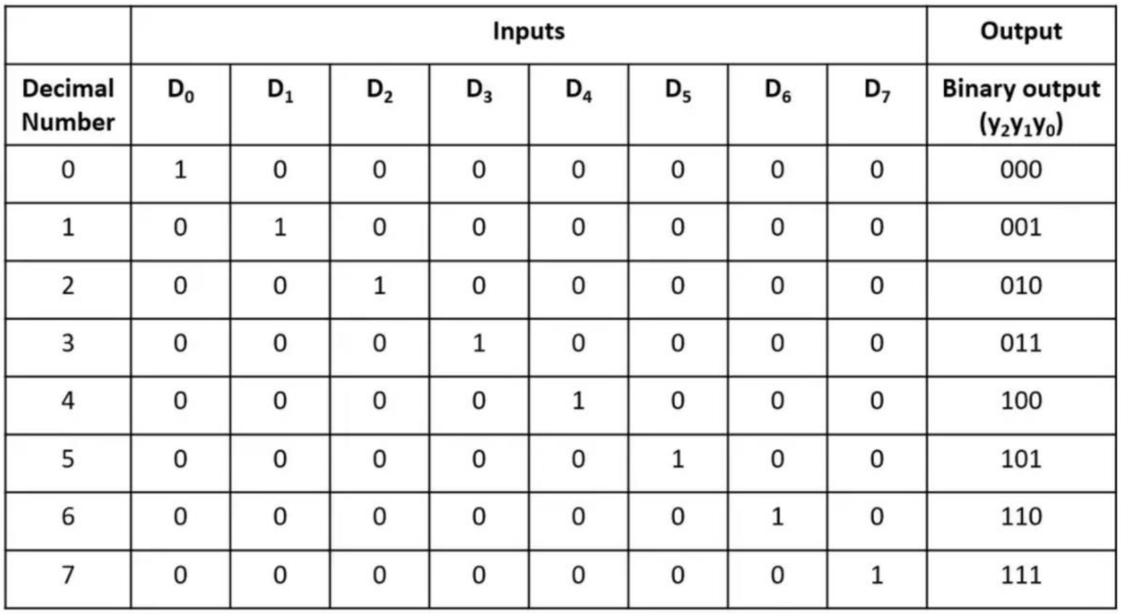
Nomenclature: M: N encoder where M denotes input lines and N denotes coded output lines.

8:3 Binary Encoder

Block Diagram



Truth Table



	Inputs								Output
Decimal Number	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Binary output ($y_2y_1y_0$)
0	1	0	0	0	0	0	0	0	000
1	0	1	0	0	0	0	0	0	001
2	0	0	1	0	0	0	0	0	010
3	0	0	0	1	0	0	0	0	011
4	0	0	0	0	1	0	0	0	100
5	0	0	0	0	0	1	0	0	101
6	0	0	0	0	0	0	1	0	110
7	0	0	0	0	0	0	0	1	111

Logical Expression

y2 = D4 + D5 + D6 + D7

y1 = D2 + D3 + D6 + D7

y0 = D1 + D3 + D5 + D7

8:3 Binary Encoder Verilog Code

```
module binary_encoder(
  input [7:0] D,
  output [2:0] y);

  assign y[2] = D[4] | D[5] | D[6] | D[7];
  assign y[1] = D[2] | D[3] | D[6] | D[7];
  assign y[0] = D[1] | D[3] | D[5] | D[7];
endmodule
```

Testbench Code

```
module tb;
  reg [7:0] D;
  wire [2:0] y;
  int i;

  binary_encoder bin_enc(D, y);

  initial begin
    D=8'b1; #1;
    for(i=0; i<8; i++) begin
      $display("D = %h(in dec:%0d) -> y = %0h", D, i, y);
      D=D<<1; #1;
    end
  end
endmodule
```

Output:

```
D = 01(in dec:0) -> y = 0
D = 02(in dec:1) -> y = 1
D = 04(in dec:2) -> y = 2
D = 08(in dec:3) -> y = 3
D = 10(in dec:4) -> y = 4
D = 20(in dec:5) -> y = 5
D = 40(in dec:6) -> y = 6
D = 80(in dec:7) -> y = 7
```

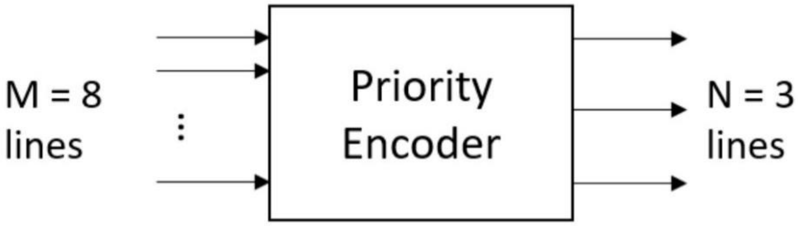
Priority Encoder

The priority encoder overcome the drawback of binary encoder that generates invalid output for more than one input line is set to high. The priority encoder prioritizes each input line and provides an encoder output corresponding to its highest input priority.

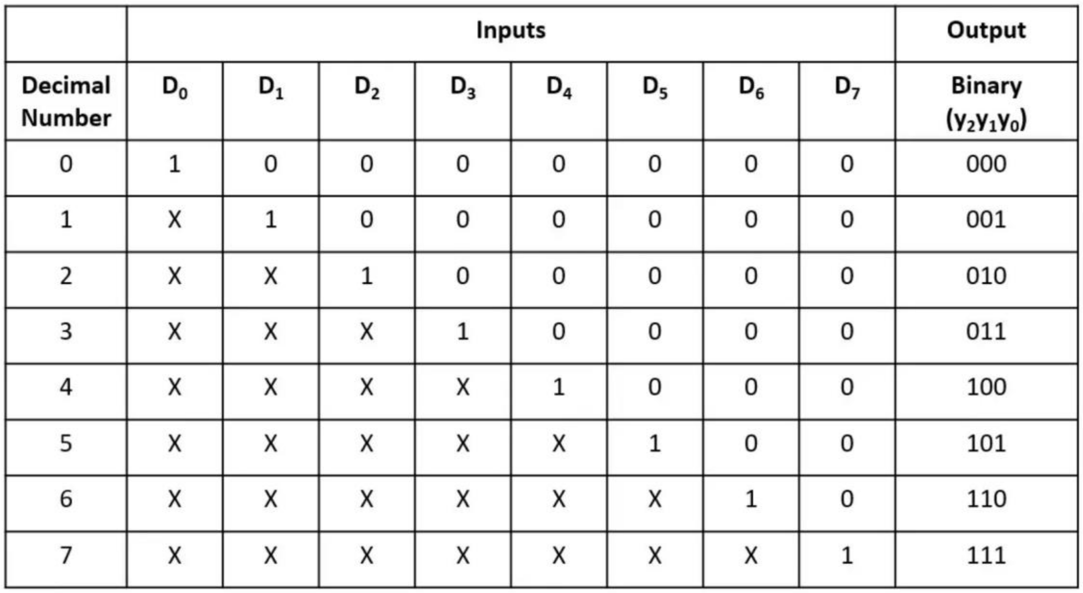
The priority encoder is widely used in digital applications. One common example of a microprocessor detecting the highest priority interrupt. The priority encoders are also used in navigation systems, robotics for controlling arm positions, communication systems, etc.

8:3 Priority Encoder

Block Diagram



Truth Table



	Inputs								Output
Decimal Number	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Binary ($y_2y_1y_0$)
0	1	0	0	0	0	0	0	0	000
1	X	1	0	0	0	0	0	0	001
2	X	X	1	0	0	0	0	0	010
3	X	X	X	1	0	0	0	0	011
4	X	X	X	X	1	0	0	0	100
5	X	X	X	X	X	1	0	0	101
6	X	X	X	X	X	X	1	0	110
7	X	X	X	X	X	X	X	1	111

8:3 Priority Encoder Verilog Code

```
module priority_encoder(
  input [7:0] D,
  output reg [2:0] y);

  always@(D) begin
    casex(D)
      8'b1xxx_xxxx: y = 3'b111;
      8'b01xx_xxxx: y = 3'b110;
      8'b001x_xxxx: y = 3'b101;
      8'b0001_xxxx: y = 3'b100;
      8'b0000_1xxx: y = 3'b011;
      8'b0000_01xx: y = 3'b010;
      8'b0000_001x: y = 3'b001;
      8'b0000_0001: y = 3'b000;
      default: $display("Invalid data received");
    endcase
  end
endmodule
```

Testbench Code

```
module tb;
  reg [7:0] D;
  wire [2:0] y;

  priority_encoder pri_enc(D, y);

  initial begin
    $monitor("D = %b -> y = %0b", D, y);
    repeat(5) begin
      D=$random; #1;
    end
  end
endmodule
```

Output:

```
D = 00100100 -> y = 101
D = 10000001 -> y = 111
D = 00001001 -> y = 11
D = 01100011 -> y = 110
D = 00001101 -> y = 11
```

Decoder

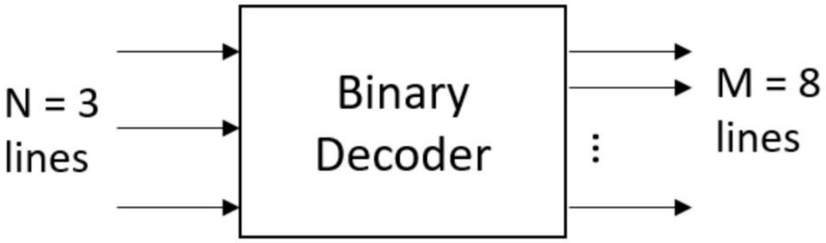
The decoder behaves exactly opposite of the encoder. They decode already coded input to its decoded form. The ‘N’(=n) input coded lines decode to ‘M’(=2^n) decoded output lines. The decoder sets exactly one line high at the output for a given encoded input line.

Nomenclature: N: M decoder where N denotes coded input lines and M denotes decoded output lines

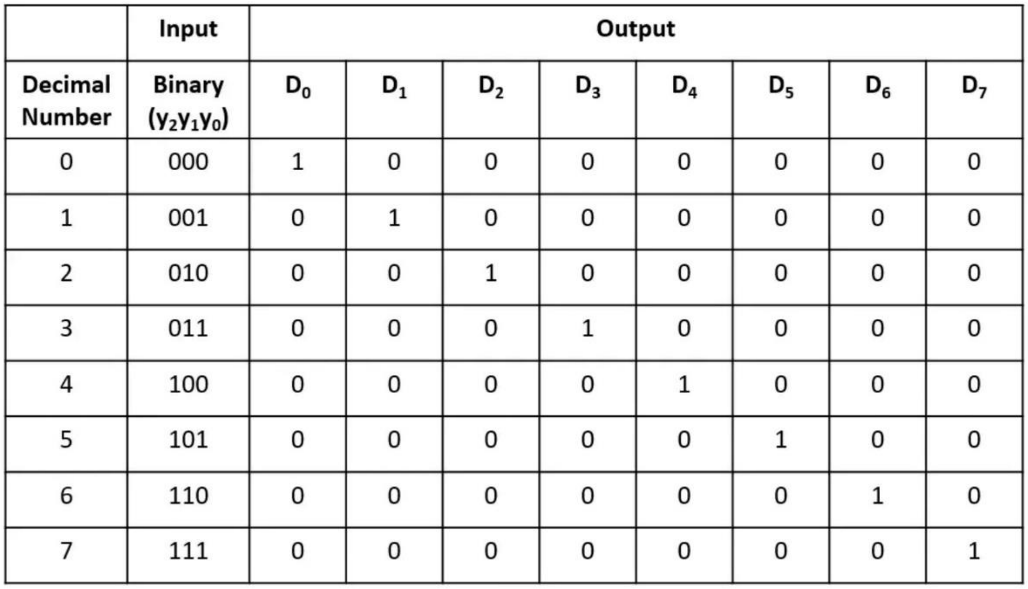
3:8 Binary decoder

The 3 input lines denote 3-bit binary code and 8 output line represents its decoded decimal form.

Block Diagram



Truth Table



	Input	Output							
Decimal Number	Binary $(y_2y_1y_0)$	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	000	1	0	0	0	0	0	0	0
1	001	0	1	0	0	0	0	0	0
2	010	0	0	1	0	0	0	0	0
3	011	0	0	0	1	0	0	0	0
4	100	0	0	0	0	1	0	0	0
5	101	0	0	0	0	0	1	0	0
6	110	0	0	0	0	0	0	1	0
7	111	0	0	0	0	0	0	0	1

3:8 Binary Decoder Verilog Code

```
module binary_decoder(
  input [2:0] D,
  output reg [7:0] y);

  always@(D) begin
    y = 0;
    case(D)
      3'b000: y[0] = 1'b1;
      3'b001: y[1] = 1'b1;
      3'b010: y[2] = 1'b1;
      3'b011: y[3] = 1'b1;
      3'b100: y[4] = 1'b1;
      3'b101: y[5] = 1'b1;
      3'b110: y[6] = 1'b1;
      3'b111: y[7] = 1'b1;
      default: y = 0;
    endcase
  end
endmodule
```

Test Bench:

```
module tb;
  reg [2:0] D;
  wire [7:0] y;

  binary_decoder bin_dec(D, y);

  initial begin
    $monitor("D = %b -> y = %0b", D, y);
    repeat(5) begin
      D=$random; #1;
    end
  end
endmodule
```

Output:

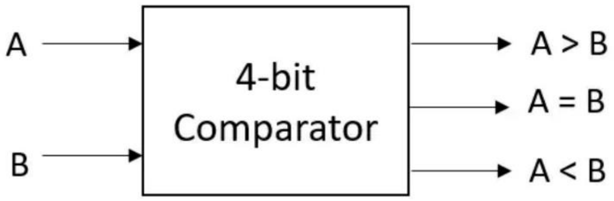
```
D = 100 -> y = 10000
D = 001 -> y = 10
D = 011 -> y = 1000
D = 101 -> y = 100000
```

Comparator

A comparator has two inputs and three output bits that say whether the first input is greater, less, or equal to the second input.

4 Bit Comparator

Block Diagram



4 Bit Comparator Verilog Code

```
module comparator(
  input [3:0] A, B,
  output reg A_grt_B, A_less_B, A_eq_B);

  always@(*) begin
    A_grt_B = 0; A_less_B = 0; A_eq_B = 0;
    if(A>B) A_grt_B = 1'b1;
    else if(A<B) A_less_B = 1'b1;
    else A_eq_B = 1'b1;
  end
endmodule
```

Testbench Code

```
module tb;
  reg [3:0] A, B;
  wire A_grt_B, A_less_B, A_eq_B;

  comparator comp(A, B, A_grt_B, A_less_B, A_eq_B);

  initial begin
    $monitor("A = %0h, B = %0h -> A_grt_B = %0b, A_less_B = %0b, A_eq_B =
%0b", A, B, A_grt_B, A_less_B, A_eq_B);
    repeat(5) begin
      A=$random; B=$random; #1;
    end
  end
endmodule
```

Output:

```
A = 4, B = 1 -> A_grt_B = 1, A_less_B = 0, A_eq_B = 0
A = 9, B = 3 -> A_grt_B = 1, A_less_B = 0, A_eq_B = 0
A = d, B = d -> A_grt_B = 0, A_less_B = 0, A_eq_B = 1
A = 5, B = 2 -> A_grt_B = 1, A_less_B = 0, A_eq_B = 0
A = 1, B = d -> A_grt_B = 0, A_less_B = 1, A_eq_B = 0
```

Array Multiplier

Array multiplier is similar to how we perform multiplication with pen and paper i.e. finding a partial product and adding them together. It is simple architecture for implementation.

Let’s understand its design implementation with a 4 x 4 unsigned array multiplier.

$$\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \text{x} & b_3 & b_2 & b_1 & b_0 \\
\cline{4-8}
 & & & & p_{30} & p_{20} & p_{10} & p_{00} \\
 & & & p_{31} & p_{21} & p_{11} & p_{01} & \text{x} \\
 & & p_{32} & p_{22} & p_{12} & p_{02} & \text{x} & \text{x} \\
 & p_{33} & p_{23} & p_{13} & p_{03} & \text{x} & \text{x} & \text{x} \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}$$

Here,

A – Multiplicand

B – Multiplier

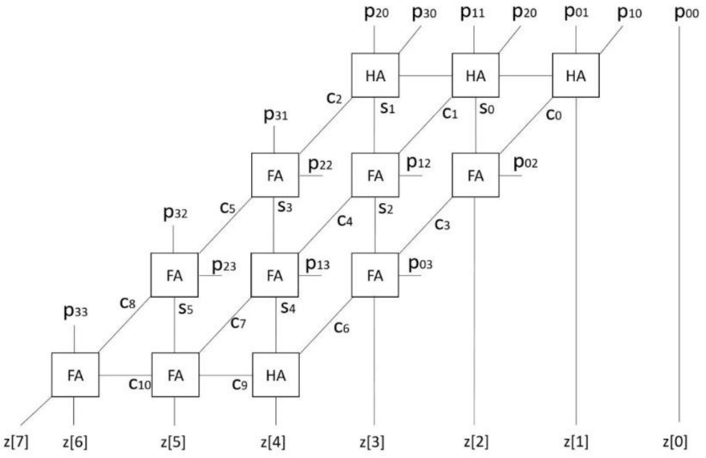
p00 – a0b0

p10 – a1b0

p20 – a2b0

.....

Block Diagram



Array Multiplier

Array Multiplier Verilog Code

```verilog
module half_adder(input a, b, output s0, c0);
  assign s0 = a ^ b;
  assign c0 = a & b;
endmodule

module full_adder(input a, b, cin, output s0, c0);
  assign s0 = a ^ b ^ cin;
  assign c0 = (a & b) | (b & cin) | (a & cin);
endmodule

module array_multiplier(input [3:0] A, B, output [7:0] z);
  reg signed p[4][4];
  wire [10:0] c; // c represents carry of HA/FA
  wire [5:0] s;  // s represents sum of HA/FA
  // For ease and readability, two diffent name s and c are used instead of
single wire name.
  genvar g;

  generate
    for(g = 0; g<4; g++) begin
      and a0(p[g][0], A[g], B[0]);
      and a1(p[g][1], A[g], B[1]);
      and a2(p[g][2], A[g], B[2]);
      and a3(p[g][3], A[g], B[3]);
    end
  endgenerate
  assign z[0] = p[0][0];

  //row 0
  half_adder h0(p[0][1], p[1][0], z[1], c[0]);
  half_adder h1(p[1][1], p[2][0], s[0], c[1]);
  half_adder h2(p[2][1], p[3][0], s[1], c[2]);

  //row1
  full_adder f0(p[0][2], c[0], s[0], z[2], c[3]);
  full_adder f1(p[1][2], c[1], s[1], s[2], c[4]);
  full_adder f2(p[2][2], c[2], p[3][1], s[3], c[5]);

  //row2
  full_adder f3(p[0][3], c[3], s[2], z[3], c[6]);
  full_adder f4(p[1][3], c[4], s[3], s[4], c[7]);
  full_adder f5(p[2][3], c[5], p[3][2], s[5], c[8]);

  //row3
  half_adder h3(c[6], s[4], z[4], c[9]);
  full_adder f6(c[9], c[7], s[5], z[5], c[10]);
  full_adder f7(c[10], c[8], p[3][3], z[6], z[7]);

endmodule
```

Testbench Code

```
module TB;
  reg [3:0] A, B;
  wire [7:0] P;

  array_multiplier am(A,B,P);

  initial begin
    $monitor("A = %b: B = %b --> P = %b, P(dec) = %0d", A, B, P, P);
    A = 1; B = 0; #3;
    A = 7; B = 5; #3;
    A = 8; B = 9; #3;
    A = 4'hf; B = 4'hf;
  end
endmodule
```

Output:

```
A = 0001: B = 0000 --> P = 00000000, P(dec) = 0
A = 0111: B = 0101 --> P = 00100011, P(dec) = 35
A = 1000: B = 1001 --> P = 01001000, P(dec) = 72
A = 1111: B = 1111 --> P = 11100001, P(dec) = 225
```

Booth’s Multiplier

In Booth’s multiplier works on Booth’s Algorithm that does the multiplication of 2’s complement notation of two signed binary numbers.

Flow chart of Booth’s Algorithm

Please note of below abbreviations used:

A – holds Multiplicand

B – holds Multiplier

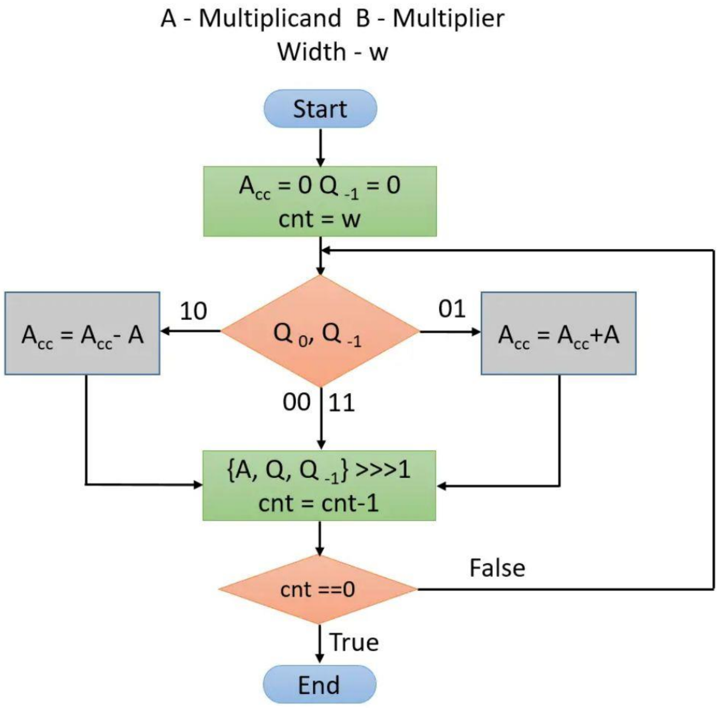
Q = B

Q0 – holds 0th bit (LSB) of Q register

Q-1 – 1-bit variable/register.

Acc – Accumulator holds the result of intermediate addition/subtraction.

Count = max(width of multiplicand register, width of multiplier register)



Booth’s Multiplier

Booth’s Multiplier Verilog Code

```
module half_adder(input a, b, output s0, c0);
  assign s0 = a ^ b;
  assign c0 = a & b;
endmodule

module full_adder(input a, b, cin, output s0, c0);
  assign s0 = a ^ b ^ cin;
  assign c0 = (a & b) | (b & cin) | (a & cin);
endmodule

module array_multiplier(input [3:0] A, B, output [7:0] z);
  reg signed p[4][4];
  wire [10:0] c; // c represents carry of HA/FA
  wire [5:0] s;  // s represents sum of HA/FA
  // For ease and readability, two diffent name s and c are used instead of
single wire name.
  genvar g;

  generate
    for(g = 0; g<4; g++) begin
      and a0(p[g][0], A[g], B[0]);
      and a1(p[g][1], A[g], B[1]);
      and a2(p[g][2], A[g], B[2]);
      and a3(p[g][3], A[g], B[3]);
    end
  endgenerate
  assign z[0] = p[0][0];

  //row 0
  half_adder h0(p[0][1], p[1][0], z[1], c[0]);
  half_adder h1(p[1][1], p[2][0], s[0], c[1]);
  half_adder h2(p[2][1], p[3][0], s[1], c[2]);

  //row1
  full_adder f0(p[0][2], c[0], s[0], z[2], c[3]);
  full_adder f1(p[1][2], c[1], s[1], s[2], c[4]);
  full_adder f2(p[2][2], c[2], p[3][1], s[3], c[5]);

  //row2
  full_adder f3(p[0][3], c[3], s[2], z[3], c[6]);
  full_adder f4(p[1][3], c[4], s[3], s[4], c[7]);
  full_adder f5(p[2][3], c[5], p[3][2], s[5], c[8]);

  //row3
  half_adder h3(c[6], s[4], z[4], c[9]);
  full_adder f6(c[9], c[7], s[5], z[5], c[10]);
  full_adder f7(c[10], c[8], p[3][3], z[6], z[7]);
endmodule
```

Testbench Code

```
module TB;
  reg [3:0] A, B;
  wire [7:0] P;

  array_multiplier am(A,B,P);

  initial begin
    $monitor("A = %b: B = %b --> P = %b, P(dec) = %0d", A, B, P, P);
    A = 1; B = 0; #3;
    A = 7; B = 5; #3;
    A = 8; B = 9; #3;
    A = 4'hf; B = 4'hf;
  end
endmodule
```

Output:

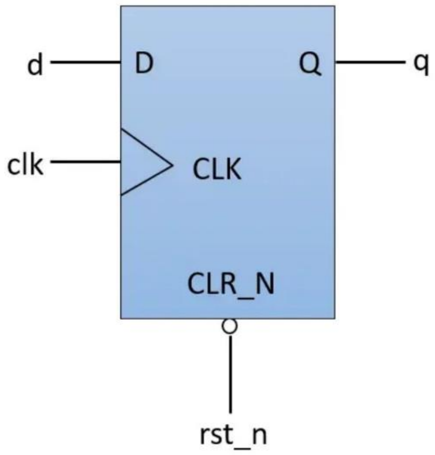
```
A = 0001: B = 0000 --> P = 00000000, P(dec) = 0
A = 0111: B = 0101 --> P = 00100011, P(dec) = 35
A = 1000: B = 1001 --> P = 01001000, P(dec) = 72
A = 1111: B = 1111 --> P = 11100001, P(dec) = 225
```

D Flip Flop with Asynchronous Reset

The D flip flop is a basic sequential element that has data input ‘d’ being driven to output ‘q’ as per clock edge. Also, the D flip-flop held the output value till the next clock cycle. Hence, it is called an edge-triggered memory element that stores a single bit.

The below D flip flop is positive edge-triggered and has asynchronous active low reset.

As soon as reset is triggered, the output gets reset



Asynchronous active low reset D flip flop

D Flip Flop with Asynchronous Reset Verilog Code

```
module D_flipflop (
  input clk, rst_n,
  input d,
  output reg q
  );

  always@(posedge clk or negedge rst_n) begin
    if(!rst_n) q <= 0;
    else       q <= d;
  end

endmodule
```

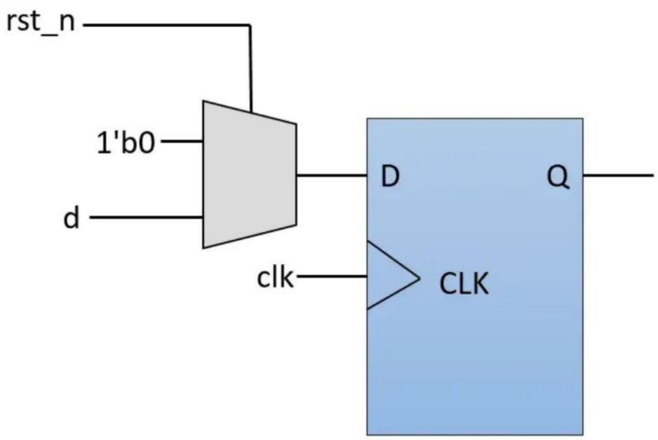
Testbench Code

```
module tb;
  reg clk, rst_n;
  reg d;
  wire q;

  D_flipflop dff(clk, rst_n, d, q);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    d = 0;

    #3 rst_n = 1;

    repeat(6) begin
      d = $urandom_range(0, 1);
      #3;
    end
    rst_n = 0; #3;
    rst_n = 1;
    repeat(6) begin
      d = $urandom_range(0, 1);
      #3;
    end
    $finish;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

D Flip Flop with Synchronous Reset

The below D flip flop is positive edge-triggered and synchronous active low reset D flip flop

As soon as reset is triggered, the output gets reset on the next posedge of a clock.



Synchronous active low reset D flip flop

D Flip Flop with Synchronous Reset Verilog Code

```
module D_flipflop (
  input clk, rst_n,
  input d,
  output reg q
  );

  always@(posedge clk) begin
    if(!rst_n) q <= 0;
    else       q <= d;
  end

endmodule
```

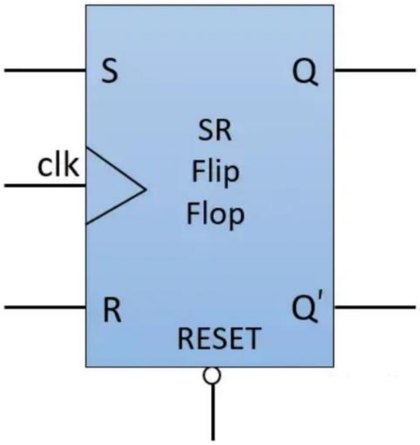
Testbench Code

```
module tb;
  reg clk, rst_n;
  reg d;
  wire q;

  D_flipflop dff(clk, rst_n, d, q);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    d = 0;

    #3 rst_n = 1;

    repeat(6) begin
      d = $urandom_range(0, 1);
      #3;
    end
    rst_n = 0; #3;
    rst_n = 1;
    repeat(6) begin
      d = $urandom_range(0, 1);
      #3;
    end
    $finish;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

SR Flip Flop

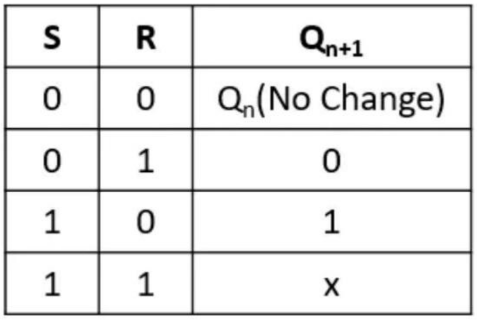
The SR flip flop has two inputs SET ‘S’ and RESET ‘R’. As the name suggests, when S = 1, output Q becomes 1, and when R = 1, output Q becomes 0. The output Q’ is the complement of Q.

Block Diagram



SR Flip Flop

Truth Table



S	R	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	x

For S = 1 and R = 1, output Q = x i.e. 1 or 0. Hence, S=1 and R=1 input combination is **invalid** and must be avoided in the SR flip flop.

SR Flip Flop Verilog Code

```
module SR_flipflop (
  input clk, rst_n,
  input s,r,
  output reg q,
  output q_bar
  );
  // always@(posedge clk or negedge rst_n) // for asynchronous reset
  always@(posedge clk) begin // for synchronous reset
    if(!rst_n) q <= 0;
    else begin
      case({s,r})
        2'b00: q <= q;    // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= 1'bx; // Invalid inputs
      endcase
    end
  end
  assign q_bar = ~q;
endmodule
```

Testbench Code

```
module tb;
  reg clk, rst_n;
  reg s, r;
  wire q, q_bar;

  SR_flipflop dff(clk, rst_n, s, r, q, q_bar);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);
    #3 rst_n = 1;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);

    drive(2'b00);
    drive(2'b01);
    drive(2'b10);
    drive(2'b11);
    #5;
    $finish;
  end

  task drive(bit [1:0] ip);
    @(posedge clk);
    {s,r} = ip;
    #1 $display("s=%b, r=%b --> q=%b, q_bar=%b",s, r, q, q_bar);
  endtask

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

Output:

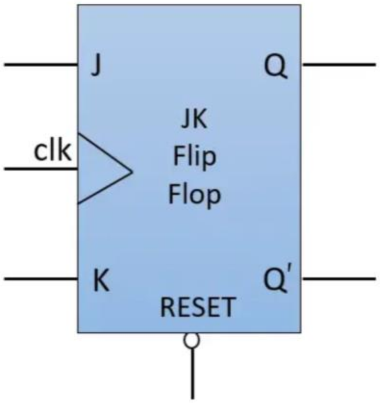
```
Reset=0 --> q=x, q_bar=x
Reset=1 --> q=0, q_bar=1
s=0, r=0 --> q=0, q_bar=1
s=0, r=1 --> q=0, q_bar=1
s=1, r=0 --> q=1, q_bar=0
s=1, r=1 --> q=x, q_bar=x
```

JK Flip Flop

The JK flip flop has two inputs 'J' and 'K'. It behaves the same as SR flip flop except that it eliminates undefined output state (Q = x for S=1, R=1).

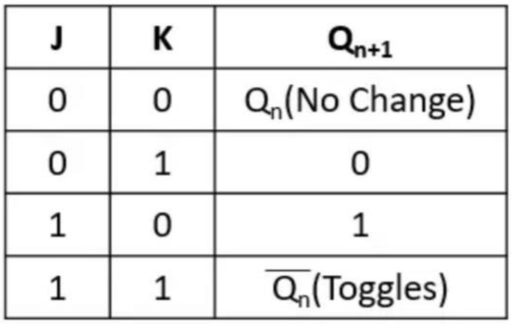
For J=1, K=1, output Q toggles from its previous output state.

Block Diagram



JK Flip Flop

Truth Table



J	K	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	$\overline{Q_n}$(Toggles)

JK Flip Flop Verilog Code

```
module JK_flipflop (
  input clk, rst_n,
  input j,k,
  output reg q,
  output q_bar
  );

  // always@(posedge clk or negedge rst_n) // for asynchronous reset
  always@(posedge clk) begin // for synchronous reset
    if(!rst_n) q <= 0;
    else begin
      case({j,k})
        2'b00: q <= q;    // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= ~q;   // Toggle
      endcase
    end
  end
  assign q_bar = ~q;
endmodule
```

Testbench Code

```
module tb;
  reg clk, rst_n;
  reg j, k;
  wire q, q_bar;

  JK_flipflop dff(clk, rst_n, j, k, q, q_bar);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);
    #3 rst_n = 1;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);

    drive(2'b00);
    drive(2'b01);
    drive(2'b10);
    drive(2'b11); // Toggles previous output
    drive(2'b11); // Toggles previous output
    #5;
    $finish;
  end

  task drive(bit [1:0] ip);
    @(posedge clk);
    {j,k} = ip;
    #1 $display("j=%b, k=%b --> q=%b, q_bar=%b",j, k, q, q_bar);
  endtask

  initial begin
```

```
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

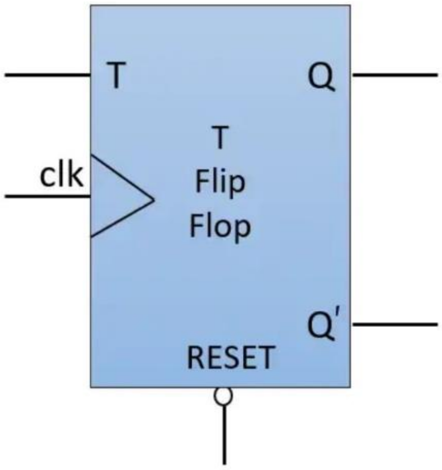
Output:

```
Reset=0 --> q=x, q_bar=x
Reset=1 --> q=0, q_bar=1
j=0, k=0 --> q=0, q_bar=1
j=0, k=1 --> q=0, q_bar=1
j=1, k=0 --> q=1, q_bar=0
j=1, k=1 --> q=0, q_bar=1
j=1, k=1 --> q=1, q_bar=0
```

T Flip Flop

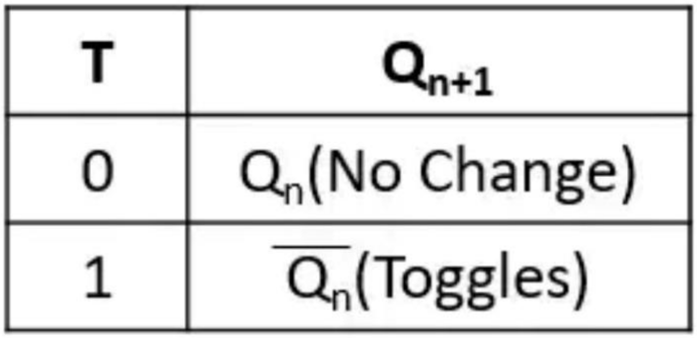
The T flip flop has single input as a ‘T’. Whenever input T=1, the output toggles from its previous state else the output remains the same as its previous state. It behaves the same as JK flip flop when J=1 and K=1. Hence, it can also be implemented by tieing both inputs (J and K) of JK flip flop.

Block Diagram



T Flip Flop

Truth Table



T	Q_{n+1}
0	Q_n(No Change)
1	$\overline{Q_n}$(Toggles)

T Flip Flop Verilog Code

```
module T_flipflop (
  input clk, rst_n,
  input t,
  output reg q,
  output q_bar
  );

  // always@(posedge clk or negedge rst_n) // for asynchronous reset
  always@(posedge clk) begin // for synchronous reset
    if(!rst_n) q <= 0;
    else begin
      q <= (t?~q:q);
    end
  end
  assign q_bar = ~q;
endmodule
```

Testbench Code

```
module tb;
  reg clk, rst_n;
  reg t;
  wire q, q_bar;

  T_flipflop dff(clk, rst_n, t, q, q_bar);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);
    #3 rst_n = 1;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);

    drive(0); // Same as previous output
    drive(1); // Toggles previous output
    drive(1); // Toggles previous output
    drive(1); // Toggles previous output
    drive(0); // Same as previous output
    #5;
    $finish;
  end

  task drive(bit ip);
    @(posedge clk);
    t = ip;
    #1 $display("t=%b --> q=%b, q_bar=%b",t, q, q_bar);
  endtask

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

Output:

```
Reset=0 --> q=x, q_bar=x
Reset=1 --> q=0, q_bar=1
t=0 --> q=0, q_bar=1
t=1 --> q=1, q_bar=0
t=1 --> q=0, q_bar=1
t=1 --> q=1, q_bar=0
t=0 --> q=1, q_bar=0
```

Universal Shift Register

A universal shift register is a sequential logic that can store data within and on every clock pulse it transfers data to the output port.

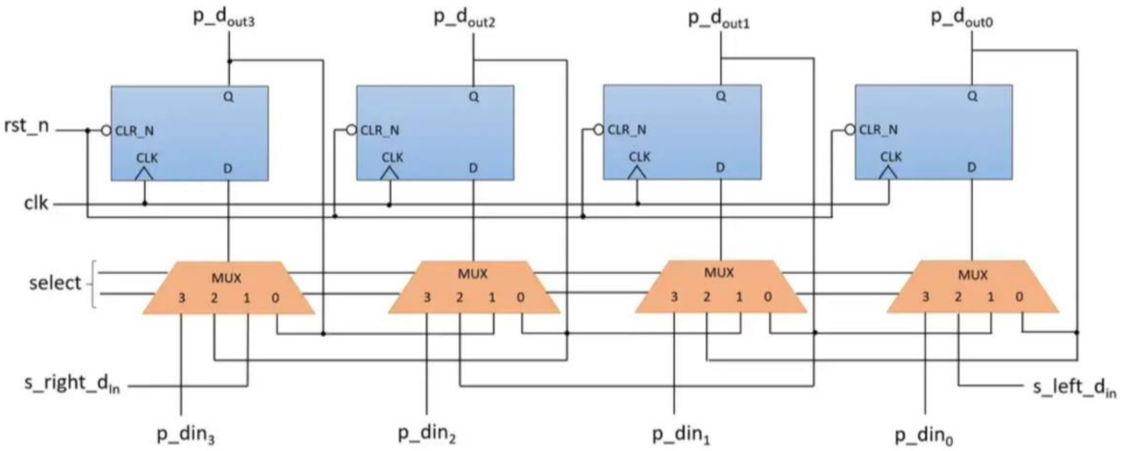
The universal shift register can be used as

1. Parallel In Parallel Out shift register
2. Parallel In Serial Out shift register
3. Serial In Parallel Out shift register
4. Serial In Serial Out shift register

Operation

Mode control (select)		**Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Block Diagram



Universal Shift Register

Universal Shift Register Verilog Code

```
module universal_shift_reg(
  input clk, rst_n,
  input [1:0] select, // select operation
  input [3:0] p_din,  // parallel data in
  input s_left_din,   // serial left data in
  input s_right_din,  // serial right data in
  output reg [3:0] p_dout, //parallel data out
  output s_left_dout, // serial left data out
  output s_right_dout // serial right data out
);
  always@(posedge clk) begin
    if(!rst_n) p_dout <= 0;
    else begin
      case(select)
        2'h1: p_dout <= {s_right_din,p_dout[3:1]}; // Right Shift
        2'h2: p_dout <= {p_dout[2:0],s_left_din};  // Left Shift
        2'h3: p_dout <= p_din; // Parallel in - Parallel out
        default: p_dout <= p_dout; // Do nothing
      endcase
    end
  end
  assign s_left_dout = p_dout[0];
  assign s_right_dout = p_dout[3];
endmodule
```

Testbench Code

```
module TB;
  reg clk, rst_n;
  reg [1:0] select;
  reg [3:0] p_din;
  reg s_left_din, s_right_din;
  wire [3:0] p_dout; //parallel data out
  wire s_left_dout, s_right_dout;

  universal_shift_reg usr(clk, rst_n, select, p_din, s_left_din, s_right_din,
p_dout, s_left_dout, s_right_dout);

  always #2 clk = ~clk;
  initial begin
    $monitor("select=%b, p_din=%b, s_left_din=%b, s_right_din=%b --> p_dout =
%b, s_left_dout = %b, s_right_dout = %b",select, p_din, s_left_din,
s_right_din, p_dout, s_left_dout, s_right_dout);
    clk = 0; rst_n = 0;
    #3 rst_n = 1;

    p_din = 4'b1101;
    s_left_din = 1'b1;
    s_right_din = 1'b0;

    select = 2'h3; #10;
    select = 2'h1; #20;
    p_din = 4'b1101;
    select = 2'h3; #10;
    select = 2'h2; #20;
    select = 2'h0; #20;

    $finish;
  end
  // To enable waveform
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end

endmodule
```

Output:

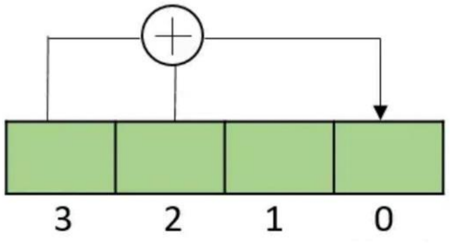
```
select=xx, p_din=xxxx, s_left_din=x, s_right_din=x --> p_dout = xxxx,
s_left_dout = x, s_right_dout = x
select=xx, p_din=xxxx, s_left_din=x, s_right_din=x --> p_dout = 0000,
s_left_dout = 0, s_right_dout = 0
select=11, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0000,
s_left_dout = 0, s_right_dout = 0
select=11, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1101,
s_left_dout = 1, s_right_dout = 1
select=01, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1101,
s_left_dout = 1, s_right_dout = 1
select=01, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0110,
s_left_dout = 0, s_right_dout = 0
select=01, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0011,
s_left_dout = 1, s_right_dout = 0
select=01, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0001,
s_left_dout = 1, s_right_dout = 0
select=01, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0000,
s_left_dout = 0, s_right_dout = 0
select=11, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0000,
s_left_dout = 0, s_right_dout = 0
select=11, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1101,
s_left_dout = 1, s_right_dout = 1
select=10, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1101,
s_left_dout = 1, s_right_dout = 1
select=10, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1011,
s_left_dout = 1, s_right_dout = 1
select=10, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 0111,
s_left_dout = 1, s_right_dout = 0
select=10, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1111,
s_left_dout = 1, s_right_dout = 1
select=00, p_din=1101, s_left_din=1, s_right_din=0 --> p_dout = 1111,
s_left_dout = 1, s_right_dout = 1
```

Linear Feedback Shift Register (LFSR)

A Linear-feedback shift register (LFSR) is another variation of shift register whose input bit is a linear function (typically XOR operation) of its previous state. It is generally used as a pseudo-random number generator, whitening sequence, pseudo-noise sequence, etc.

The bit positions that act as an input to a linear function to affect the next state are known as taps.

4-bit pseudo-random sequence generator



Linear-feedback shift register (LFSR)

At every step,

1. Q[3] xor Q[2]
2. Q = Q << 1
3. The result of the XOR operation is fed to the LSB (0th bit)

In the above pseudo-random sequence generator, taps are 4 and 3.

LFSR Verilog Code

```
module LFSR(input clk, rst, output reg [3:0] op);
  always@(posedge clk) begin
    if(rst) op <= 4'hf;
    else op = {op[2:0],(op[3]^op[2])};
  end
endmodule
```

Testbench Code

```
module TB;
  reg clk, rst;
  wire [3:0]op;

  LFSR lfsr1(clk, rst, op);

  initial begin
    $monitor("op=%b",op);
    clk = 0; rst = 1;
    #5 rst = 0;
    #50; $finish;
  end

  always #2 clk=~clk;

  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Output:

```
op=xxxx
op=1111
op=1110
op=1100
op=1000
op=0001
op=0010
op=0100
op=1001
op=0011
op=0110
op=1101
op=1010
op=0101
op=1011
```

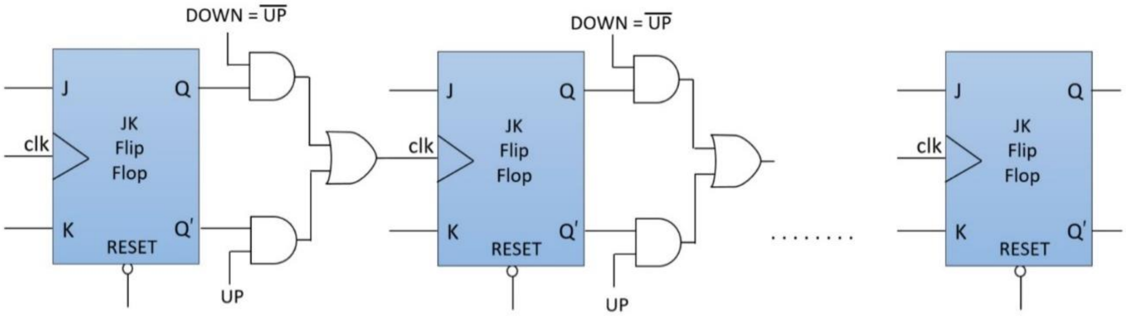
Asynchronous Counter

In asynchronous counter, within the counter output of one flip flop stage, is driven to the clock pin of the next flip flop stage. Thus, the transition of the first stage ripples till the last flip-flop stage. Hence, it is also known as the “Ripple Counter” or “Serial Counter”. In the asynchronous counter, individual flip-flop stages are triggered with the different clocks. Due to this, it is slower when compared with a synchronous counter. It is easy to design with less hardware requirement.

Disadvantages

1. Slow operation when compared with a synchronous counter.
2. High propagation delay

Block Diagram



Asynchronous Counter

4-bit Up-Down Asynchronous Counter Verilog Code

```verilog
module JK_flipflop (
  input clk, rst_n,
  input j,k,
  output reg q, q_bar
  );

  always@(posedge clk or negedge rst_n) begin // for asynchronous reset
    if(!rst_n) q <= 0;
    else begin
      case({j,k})
        2'b00: q <= q;    // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= ~q;   // Toggle
      endcase
    end
  end
  assign q_bar = ~q;
endmodule

module updown_selector(input q, q_bar, bit up, output nclk);
  assign nclk = up?q_bar:q;
endmodule

module asynchronous_counter #(parameter SIZE=4)(
  input clk, rst_n,
  input j, k,
  input up,
  output [3:0] q, q_bar
);
  wire [3:0] nclk;
  genvar g;
  /*
  // Up Counter (at output q)
  JK_flipflop jk1(clk, rst_n, j, k, q[0], q_bar[0]);
  JK_flipflop jk2(q_bar[0], rst_n, j, k, q[1], q_bar[1]);
  JK_flipflop jk3(q_bar[1], rst_n, j, k, q[2], q_bar[2]);
  JK_flipflop jk4(q_bar[2], rst_n, j, k, q[3], q_bar[3]);

  // Down Counter (at output q)
  JK_flipflop jk1(clk, rst_n, j, k, q[0], q_bar[0]);
  JK_flipflop jk2(q[0], rst_n, j, k, q[1], q_bar[1]);
  JK_flipflop jk3(q[1], rst_n, j, k, q[2], q_bar[2]);
  JK_flipflop jk4(q[2], rst_n, j, k, q[3], q_bar[3]);

  // Up and Down Counter (at output q)
  JK_flipflop jk1(clk, rst_n, j, k, q[0], q_bar[0]);
  updown_selector ud1(q[0], q_bar[0], up, nclk[0]);

  JK_flipflop jk2(nclk[0], rst_n, j, k, q[1], q_bar[1]);
  updown_selector ud2(q[1], q_bar[1], up, nclk[1]);

  JK_flipflop jk3(nclk[1], rst_n, j, k, q[2], q_bar[2]);
  updown_selector ud3(q[2], q_bar[2], up, nclk[2]);
  JK_flipflop jk4(nclk[2], rst_n, j, k, q[3], q_bar[3]);
  */

  // Using generate block
  JK_flipflop jk0(clk, rst_n, j, k, q[0], q_bar[0]);
  generate
    for(g = 1; g<SIZE; g++) begin
      updown_selector ud1(q[g-1], q_bar[g-1], up, nclk[g-1]);
      JK_flipflop jk1(nclk[g-1], rst_n, j, k, q[g], q_bar[g]);
    end
  endgenerate
endmodule
```

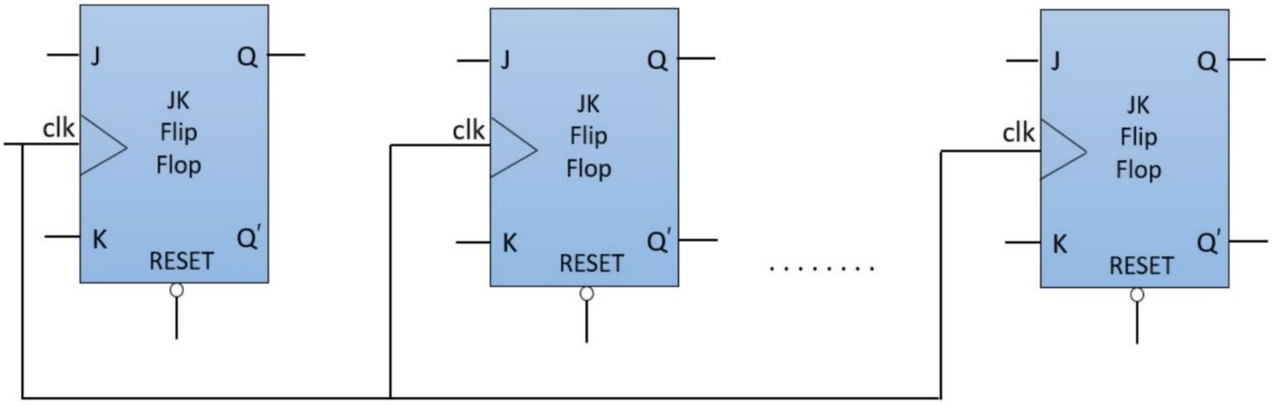
Testbench Code

```
module tb;
  reg clk, rst_n;
  reg j, k;
  reg up;
  wire [3:0] q, q_bar;
  asynchronous_counter(clk, rst_n, j, k, up, q, q_bar);

  initial begin
    clk = 0; rst_n = 0;
    up = 1;
    #4; rst_n = 1;
    j = 1; k = 1;
    #80;
    rst_n = 0;
    #4; rst_n = 1; up = 0;
    #50;
    $finish;
  end
  always #2 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Synchronous Counter

As we have discussed the asynchronous counter, output of one flip-flop stage is connected to the input clock pin of the next stage that introduces propagation delay. In a synchronous counter, all flip-flops within the counter are clocked together by the same clock which makes it faster when compared with an asynchronous counter. It is known as a parallel counter.

Block Diagram



Synchronous Counter

4-bit Up-Down Synchronous Counter

```
module synchronous_counter #(parameter SIZE=4)(
  input clk, rst_n,
  input up,
  output reg [3:0] cnt
);

  always@(posedge clk) begin
    if(!rst_n) begin
      cnt <= 4'h0;
    end
    else begin
      if(up) cnt <= cnt + 1'b1;
      else cnt <= cnt - 1'b1;
    end
  end
endmodule
```

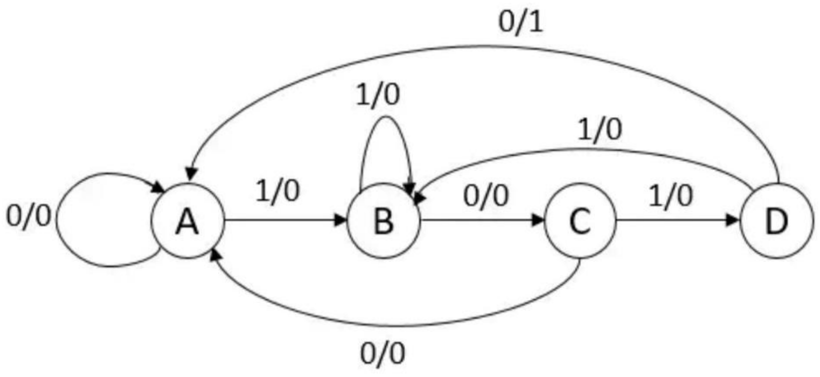
Testbench Code

```
module tb;
  reg clk, rst_n;
  reg up;
  wire [3:0] cnt;
  synchronous_counter(clk, rst_n, up, cnt);

  initial begin
    clk = 0; rst_n = 0;
    up = 1;
    #4; rst_n = 1;
    #80;
    rst_n = 0;
    #4; rst_n = 1; up = 0;
    #50;
    $finish;
  end
  always #2 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Mealy Sequence Detector

In mealy machine, output depends on the present state and current input.

1010 non-Overlapping Mealy Sequence Detector Verilog Code



1010 Non-Overlapping Mealy Sequence Detector

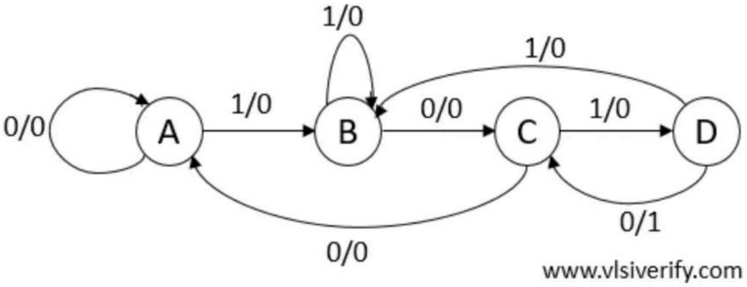
```
module seq_detector_1010(input bit clk, rst_n, x, output z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;

  bit [3:0] state, next_state;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end

  always @(state or x) begin
    case(state)
      A: begin
           if(x == 0) next_state = A;
           else       next_state = B;
         end
      B: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      C: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      D: begin
```

```
        if(x == 0) next_state = A; //This state only differs when compared 
with Mealy Overlaping Machine
           else       next_state = B;
         end
      default: next_state = A;
    endcase
  end
  assign z = (state == D) && (x == 0)? 1:0;
endmodule
```

Testbench Code

```
module TB;
  reg clk, rst_n, x;
  wire z;
  
  seq_detector_1010 sd(clk, rst_n, x, z);
  initial clk = 0;
  always #2 clk = ~clk;
  
  initial begin
    x = 0;
    #1 rst_n = 0;
    #2 rst_n = 1;
    
    #3 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #10;
    $finish;
  end
  
  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

1010 Overlapping Mealy Sequence Detector Verilog Code



1010 Overlapping Mealy Sequence Detector

```
module seq_detector_1010(input bit clk, rst_n, x, output z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;

  bit [3:0] state, next_state;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end

  always @(state or x) begin
    case(state)
      A: begin
           if(x == 0) next_state = A;
           else       next_state = B;
         end
      B: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      C: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      D: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      default: next_state = A;
    endcase
  end
  assign z = (state == D) && (x == 0)? 1:0;
endmodule
```

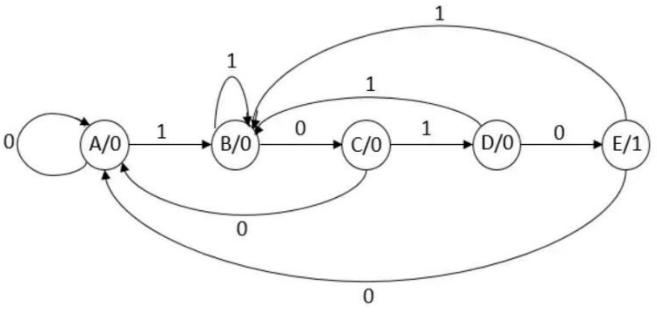
Testbench Code

```
module TB;
  reg clk, rst_n, x;
  wire z;

  seq_detector_1010 sd(clk, rst_n, x, z);
  initial clk = 0;
  always #2 clk = ~clk;

  initial begin
    x = 0;
    #1 rst_n = 0;
    #2 rst_n = 1;

    #3 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #10;
    $finish;
  end

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

Moore Sequence Detector

In moore machine, output only depends on the present state. It is independent of current input.

Moore Sequence Detector Verilog Code



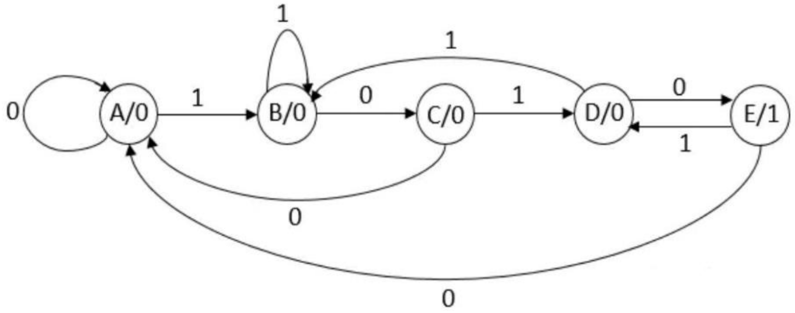
1010 Non-Overlapping Moore Sequence Detector

```
module seq_detector_1010(input bit clk, rst_n, x, output reg z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;
  parameter E = 4'h5; // extra state when compared with Mealy Machine

  bit [3:0] state, next_state;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end

  always @(state or x) begin
    case(state)
      A: begin
           if(x == 0) next_state = A;
           else       next_state = B;
         end
      B: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      C: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      D: begin
           if(x == 0) next_state = E;
           else       next_state = B;
         end
      E: begin
           if(x == 0) next_state = A;
           else       next_state = B; //This state only differs when compared
with Moore Overlaping Machine
         end
      default: next_state = A;
    endcase
  end
  //As output z is only depends on present state
  always@(state) begin
    case(state)
      A : z = 0;
      B : z = 0;
      C : z = 0;
      D : z = 0;
      E : z = 1;
      default : z = 0;
    endcase
  end
endmodule
```

Testbench Code

```
module TB;
  reg clk, rst_n, x;
  wire z;

  seq_detector_1010 sd(clk, rst_n, x, z);
  initial clk = 0;
  always #2 clk = ~clk;

  initial begin
    x = 0;
    #1 rst_n = 0;
    #2 rst_n = 1;

    #3 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #10;
    $finish;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

1010 Overlapping Moore Sequence Detector Verilog Code



1010 Overlapping Moore Sequence Detector

```
module seq_detector_1010(input bit clk, rst_n, x, output reg z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;
  parameter E = 4'h5; // extra state when compared with Mealy Machine

  bit [3:0] state, next_state;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end

  always @(state or x) begin
    case(state)
      A: begin
           if(x == 0) next_state = A;
           else       next_state = B;
         end
      B: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      C: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      D: begin
           if(x == 0) next_state = E;
           else       next_state = B;
         end
      E: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      default: next_state = A;
    endcase
  end
  //As output z is only depends on present state
  always@(state) begin
    case(state)
      A : z = 0;
      B : z = 0;
      C : z = 0;
      D : z = 0;
      E : z = 1;
      default : z = 0;
    endcase
  end
endmodule
```

Testbench Code

```
module TB;
  reg clk, rst_n, x;
  wire z;

  seq_detector_1010 sd(clk, rst_n, x, z);
  initial clk = 0;
  always #2 clk = ~clk;

  initial begin
    x = 0;
    #1 rst_n = 0;
    #2 rst_n = 1;

    #3 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #10;
    $finish;
  end

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

Synchronous FIFO

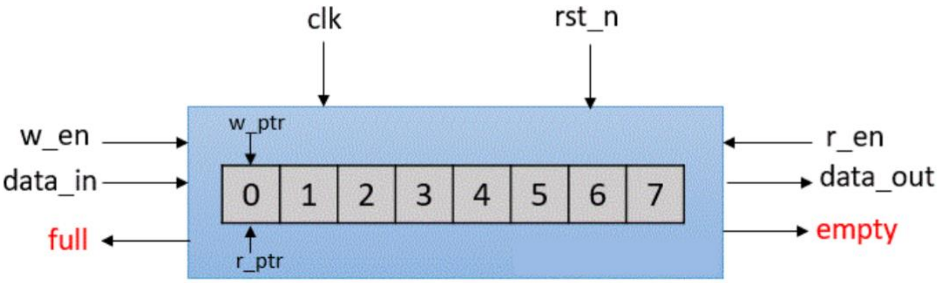
First In First Out (FIFO) is a very popular and useful design block for purpose of synchronization and a handshaking mechanism between the modules.

Depth of FIFO: The number of slots or rows in FIFO is called the depth of the FIFO.

Width of FIFO: The number of bits that can be stored in each slot or row is called the width of the FIFO.

Synchronous FIFO

In Synchronous FIFO, data read and write operations use the same clock frequency. Usually, they are used with high clock frequency to support high-speed systems.



Synchronous FIFO

Synchronous FIFO Operation

Signals:

wr_en: write enable

wr_data: write data

full: FIFO is full

empty: FIFO is empty

rd_en: read enable

rd_data: read data

w_ptr: write pointer

r_ptr: read pointer

FIFO write operation

FIFO can store/write the wr_data at every posedge of the clock based on wr_en signal till it is full. The write pointer gets incremented on every data write in FIFO memory.

FIFO read operation

The data can be taken out or read from FIFO at every posedge of the clock based on the rd_en signal till it is empty. The read pointer gets incremented on every data read from FIFO memory.

Synchronous FIFO Verilog Code

A synchronous FIFO can be implemented in various ways. Full and empty conditions differ based on implementation.

Method 1

In this method, the width of the write and read pointer = log2(depth of FIFO). The FIFO full and empty conditions can be determined as

Empty condition

w_ptr == r_ptr i.e. write and read pointers has the same value.

Full condition

The full condition means every slot in the FIFO is occupied, but then w_ptr and r_ptr will again have the same value. Thus, it is not possible to determine whether it is a full or empty condition. Thus, the last slot of FIFO is intentionally kept empty, and the full condition can be written as (w_ptr+1’b1) == r_ptr)

Verilog Code

```
module synchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input clk, rst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output full, empty
);

  reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr;
  reg [DATA_WIDTH-1:0] fifo[DEPTH];

  // Set Default values on reset.
  always@(posedge clk) begin
    if(!rst_n) begin
      w_ptr <= 0; r_ptr <= 0;
      data_out <= 0;
    end
  end

  // To write data to FIFO
  always@(posedge clk) begin
    if(w_en & !full)begin
      fifo[w_ptr] <= data_in;
      w_ptr <= w_ptr + 1;
    end
  end

  // To read data from FIFO
  always@(posedge clk) begin
    if(r_en & !empty) begin
      data_out <= fifo[r_ptr];
      r_ptr <= r_ptr + 1;
    end
  end

  assign full = ((w_ptr+1'b1) == r_ptr);
  assign empty = (w_ptr == r_ptr);
endmodule
```

Testbench Code - 1

```
module sync_fifo_TB;
  parameter DATA_WIDTH = 8;

  reg clk, rst_n;
  reg w_en, r_en;
  reg [DATA_WIDTH-1:0] data_in;
  wire [DATA_WIDTH-1:0] data_out;
  wire full, empty;

  // Queue to push data_in
  reg [DATA_WIDTH-1:0] wdata_q[$], wdata;

  synchronous_fifo s_fifo(clk, rst_n, w_en, r_en, data_in, data_out, full,
empty);

  always #5ns clk = ~clk;
  initial begin
    clk = 1'b0; rst_n = 1'b0;
    w_en = 1'b0;
    data_in = 0;

    repeat(10) @(posedge clk);
    rst_n = 1'b1;

    repeat(2) begin
      for (int i=0; i<30; i++) begin
        @(posedge clk);
        w_en = (i%2 == 0)? 1'b1 : 1'b0;
        if (w_en & !full) begin
          data_in = $urandom;
          wdata_q.push_back(data_in);
        end
      end
      #50;
    end
  end

  initial begin
    clk = 1'b0; rst_n = 1'b0;
    r_en = 1'b0;

    repeat(20) @(posedge clk);
    rst_n = 1'b1;

    repeat(2) begin
      for (int i=0; i<30; i++) begin
        @(posedge clk);
        r_en = (i%2 == 0)? 1'b1 : 1'b0;
        if (r_en & !empty) begin
          #1;
          wdata = wdata_q.pop_front();
          if(data_out !== wdata) $error("Time = %0t: Comparison Failed: expected wr_data = %h, rd_data = %h", $time, wdata, data_out);
          else $display("Time = %0t: Comparison Passed: wr_data = %h and rd_data = %h",$time, wdata, data_out);
        end
      end
      #50;
    end

    $finish;
  end

  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Output:

```
Time = 206: Comparison Passed: wr_data = 13 and rd_data = 13
Time = 226: Comparison Passed: wr_data = 70 and rd_data = 70
Time = 246: Comparison Passed: wr_data = fd and rd_data = fd
Time = 266: Comparison Passed: wr_data = e2 and rd_data = e2
Time = 286: Comparison Passed: wr_data = 97 and rd_data = 97
Time = 306: Comparison Passed: wr_data = f1 and rd_data = f1
Time = 326: Comparison Passed: wr_data = c5 and rd_data = c5
Time = 346: Comparison Passed: wr_data = ec and rd_data = ec
Time = 366: Comparison Passed: wr_data = 48 and rd_data = 48
Time = 386: Comparison Passed: wr_data = 0c and rd_data = 0c
Time = 406: Comparison Passed: wr_data = 2c and rd_data = 2c
Time = 426: Comparison Passed: wr_data = 6b and rd_data = 6b
Time = 446: Comparison Passed: wr_data = 1b and rd_data = 1b
Time = 466: Comparison Passed: wr_data = 45 and rd_data = 45
Time = 486: Comparison Passed: wr_data = f4 and rd_data = f4
Time = 546: Comparison Passed: wr_data = 6c and rd_data = 6c
Time = 566: Comparison Passed: wr_data = 67 and rd_data = 67
Time = 586: Comparison Passed: wr_data = 8c and rd_data = 8c
Time = 606: Comparison Passed: wr_data = 4a and rd_data = 4a
Time = 626: Comparison Passed: wr_data = a6 and rd_data = a6
Time = 646: Comparison Passed: wr_data = a3 and rd_data = a3
Time = 666: Comparison Passed: wr_data = 9d and rd_data = 9d
Time = 686: Comparison Passed: wr_data = 7c and rd_data = 7c
Time = 706: Comparison Passed: wr_data = b8 and rd_data = b8
Time = 726: Comparison Passed: wr_data = eb and rd_data = eb
Time = 746: Comparison Passed: wr_data = 5b and rd_data = 5b
Time = 766: Comparison Passed: wr_data = f3 and rd_data = f3
Time = 786: Comparison Passed: wr_data = 4d and rd_data = 4d
Time = 806: Comparison Passed: wr_data = 5c and rd_data = 5c
Time = 826: Comparison Passed: wr_data = f6 and rd_data = f6
```

Testbench Code - 2

```
module sync_fifo_TB;
  reg clk, rst_n;
  reg w_en, r_en;
  reg [7:0] data_in;
  wire [7:0] data_out;
  wire full, empty;

  synchronous_fifo s_fifo(clk, rst_n, w_en, r_en, data_in, data_out, full,
empty);
  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    w_en = 0; r_en = 0;
    #3 rst_n = 1;
    drive(20);
    drive(40);
    $finish;
  end

  task push();
    if(!full) begin
      w_en = 1;
      data_in = $random;
      #1 $display("Push In: w_en=%b, r_en=%b, data_in=%h",w_en,
r_en,data_in);
    end
    else $display("FIFO Full!! Can not push data_in=%d", data_in);
  endtask

  task pop();
    if(!empty) begin
      r_en = 1;
      #1 $display("Pop Out: w_en=%b, r_en=%b, data_out=%h",w_en,
r_en,data_out);
    end
    else $display("FIFO Empty!! Can not pop data_out");
  endtask

  task drive(int delay);
    w_en = 0; r_en = 0;
    fork
      begin
        repeat(10) begin @(posedge clk) push(); end
        w_en = 0;
      end
      begin
        #delay;
        repeat(10) begin @(posedge clk) pop(); end
        r_en = 0;
      end
    join
  endtask
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Method 2

In order to avoid an empty slot as mentioned in method 1, the width of write and read pointers is increased by 1 bit. This extra bit helps determine empty and full conditions when FIFO is empty (w_ptr == r_ptr when all slots are empty) and FIFO is full (w_ptr == r_ptr when all slots are full).

Empty condition

w_ptr == r_ptr i.e. write and read pointers has the same value. MSB of w_ptr and r_ptr also has the same value.

Full condition

w_ptr == r_ptr i.e. write and read pointers has the same value, but the MSB of w_ptr and r_ptr differs.

Verilog Code with an extra bit in write/read pointers

```
module synchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input clk, rst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output full, empty
);

  parameter PTR_WIDTH = $clog2(DEPTH);
  reg [PTR_WIDTH:0] w_ptr, r_ptr; // addition bit to detect full/empty
condition
  reg [DATA_WIDTH-1:0] fifo[DEPTH];
  reg wrap_around;

  // Set Default values on reset.
  always@(posedge clk) begin
    if(!rst_n) begin
      w_ptr <= 0; r_ptr <= 0;
      data_out <= 0;
    end
  end

  // To write data to FIFO
  always@(posedge clk) begin
    if(w_en & !full)begin
      fifo[w_ptr[PTR_WIDTH-1:0]] <= data_in;
      w_ptr <= w_ptr + 1;
    end
  end
  // To read data from FIFO
  always@(posedge clk) begin
    if(r_en & !empty) begin
      data_out <= fifo[r_ptr[PTR_WIDTH-1:0]];
      r_ptr <= r_ptr + 1;
    end
  end
  assign wrap_around = w_ptr[PTR_WIDTH] ^ r_ptr[PTR_WIDTH]; // To check MSB
of write and read pointers are different

  //Full condition: MSB of write and read pointers are different and
remainimg bits are same.
  assign full = wrap_around & (w_ptr[PTR_WIDTH-1:0] == r_ptr[PTR_WIDTH-1:0]);

  //Empty condition: All bits of write and read pointers are same.
  //assign empty = !wrap_around & (w_ptr[PTR_WIDTH-1:0] == r_ptr[PTR_WIDTH-
1:0]);
  //or
  assign empty = (w_ptr == r_ptr);
endmodule
```

Output:

```
Push In: w_en=1, r_en=0, data_in=24
Push In: w_en=1, r_en=0, data_in=81
Push In: w_en=1, r_en=0, data_in=09
Push In: w_en=1, r_en=0, data_in=63
Push In: w_en=1, r_en=0, data_in=0d
Push In: w_en=1, r_en=1, data_in=8d
Pop Out: w_en=1, r_en=1, data_out=24
Push In: w_en=1, r_en=1, data_in=65
Pop Out: w_en=1, r_en=1, data_out=81
Push In: w_en=1, r_en=1, data_in=12
Pop Out: w_en=1, r_en=1, data_out=09
Push In: w_en=1, r_en=1, data_in=01
Pop Out: w_en=1, r_en=1, data_out=63
Push In: w_en=1, r_en=1, data_in=0d
Pop Out: w_en=0, r_en=1, data_out=0d
Pop Out: w_en=0, r_en=1, data_out=8d
Pop Out: w_en=0, r_en=1, data_out=65
Pop Out: w_en=0, r_en=1, data_out=12
Pop Out: w_en=0, r_en=1, data_out=01
Pop Out: w_en=0, r_en=1, data_out=0d
Push In: w_en=1, r_en=0, data_in=76
Push In: w_en=1, r_en=0, data_in=3d
Push In: w_en=1, r_en=0, data_in=ed
Push In: w_en=1, r_en=0, data_in=8c
Push In: w_en=1, r_en=0, data_in=f9
Push In: w_en=1, r_en=0, data_in=c6
Push In: w_en=1, r_en=0, data_in=c5
Push In: w_en=1, r_en=0, data_in=aa
FIFO Full!! Can not push data_in=170
FIFO Full!! Can not push data_in=170
Pop Out: w_en=0, r_en=1, data_out=76
Pop Out: w_en=0, r_en=1, data_out=3d
Pop Out: w_en=0, r_en=1, data_out=ed
Pop Out: w_en=0, r_en=1, data_out=8c
Pop Out: w_en=0, r_en=1, data_out=f9
Pop Out: w_en=0, r_en=1, data_out=c6
Pop Out: w_en=0, r_en=1, data_out=c5
Pop Out: w_en=0, r_en=1, data_out=aa
FIFO Empty!! Can not pop data_out
FIFO Empty!! Can not pop data_out
```

Method 3

The synchronous FIFO can also be implemented using a common counter that can be incremented or decremented based on write to the FIFO or read from the FIFO respectively.

Empty condition

count == 0 i.e. FIFO contains nothing.

Full condition

count == FIFO_DEPTH i.e. counter value has reached till the depth of FIFO

Verilog Code using counter

```
module synchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input clk, rst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output full, empty
);

  reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr;
  reg [DATA_WIDTH-1:0] fifo[DEPTH];
  reg [$clog2(DEPTH)-1:0] count;

  // Set Default values on reset.
  always@(posedge clk) begin
    if(!rst_n) begin
      w_ptr <= 0; r_ptr <= 0;
      data_out <= 0;
      count <= 0;
    end
    else begin
      case({w_en,r_en})
        2'b00, 2'b11: count <= count;
        2'b01: count <= count - 1'b1;
        2'b10: count <= count + 1'b1;
      endcase
    end
  end

  // To write data to FIFO
  always@(posedge clk) begin
    if(w_en & !full)begin
      fifo[w_ptr] <= data_in;
      w_ptr <= w_ptr + 1;
    end
  end

  // To read data from FIFO
  always@(posedge clk) begin
    if(r_en & !empty) begin
```

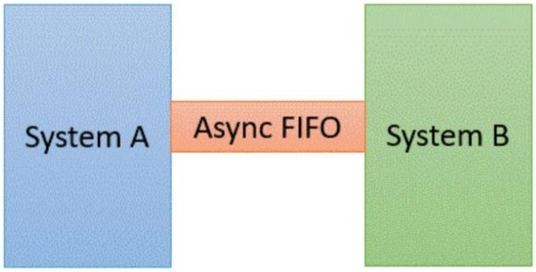
```
      data_out <= fifo[r_ptr];
      r_ptr <= r_ptr + 1;
    end
  end

  assign full = (count == DEPTH);
  assign empty = (count == 0);
endmodule
```

Output:

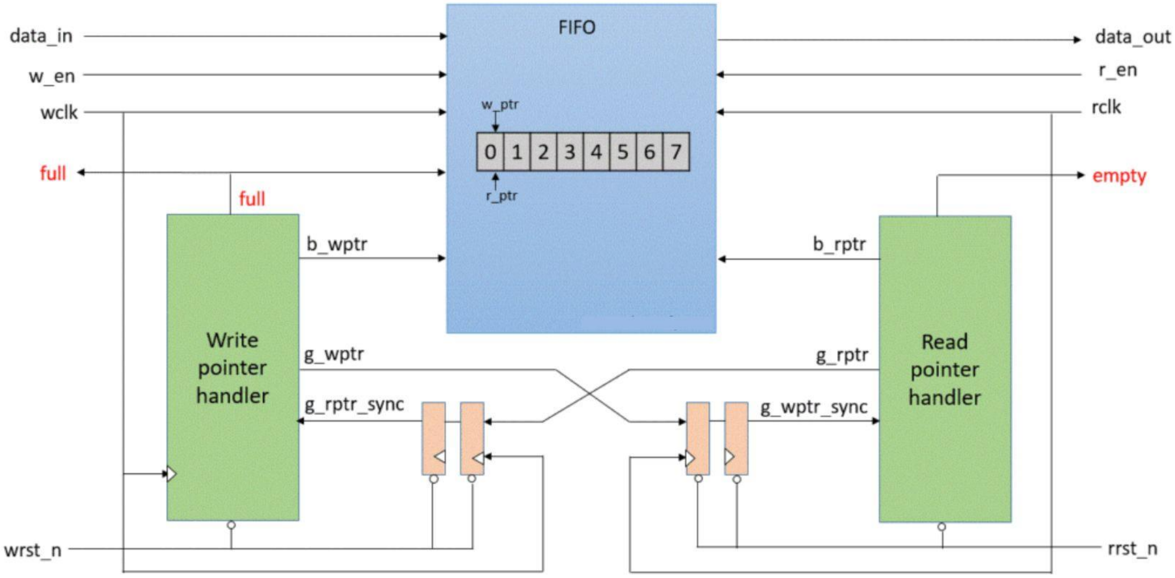
```
Time = 206: Comparison Passed: wr_data = 13 and rd_data = 13
Time = 226: Comparison Passed: wr_data = 70 and rd_data = 70
Time = 246: Comparison Passed: wr_data = fd and rd_data = fd
Time = 266: Comparison Passed: wr_data = e2 and rd_data = e2
Time = 286: Comparison Passed: wr_data = 97 and rd_data = 97
Time = 306: Comparison Passed: wr_data = f1 and rd_data = f1
Time = 326: Comparison Passed: wr_data = c5 and rd_data = c5
Time = 346: Comparison Passed: wr_data = ec and rd_data = ec
Time = 366: Comparison Passed: wr_data = 48 and rd_data = 48
Time = 386: Comparison Passed: wr_data = 0c and rd_data = 0c
Time = 406: Comparison Passed: wr_data = 2c and rd_data = 2c
Time = 426: Comparison Passed: wr_data = 6b and rd_data = 6b
Time = 446: Comparison Passed: wr_data = 1b and rd_data = 1b
Time = 466: Comparison Passed: wr_data = 45 and rd_data = 45
Time = 486: Comparison Passed: wr_data = f4 and rd_data = f4
Time = 546: Comparison Passed: wr_data = 6c and rd_data = 6c
Time = 566: Comparison Passed: wr_data = 67 and rd_data = 67
Time = 586: Comparison Passed: wr_data = 8c and rd_data = 8c
Time = 606: Comparison Passed: wr_data = 4a and rd_data = 4a
Time = 626: Comparison Passed: wr_data = a6 and rd_data = a6
Time = 646: Comparison Passed: wr_data = a3 and rd_data = a3
Time = 666: Comparison Passed: wr_data = 9d and rd_data = 9d
Time = 686: Comparison Passed: wr_data = 7c and rd_data = 7c
Time = 706: Comparison Passed: wr_data = b8 and rd_data = b8
Time = 726: Comparison Passed: wr_data = eb and rd_data = eb
Time = 746: Comparison Passed: wr_data = 5b and rd_data = 5b
Time = 766: Comparison Passed: wr_data = f3 and rd_data = f3
Time = 786: Comparison Passed: wr_data = 4d and rd_data = 4d
Time = 806: Comparison Passed: wr_data = 5c and rd_data = 5c
Time = 826: Comparison Passed: wr_data = f6 and rd_data = f6
```

Asynchronous FIFO

In asynchronous FIFO, data read and write operations use different clock frequencies. Since write and read clocks are not synchronized, it is referred to as asynchronous FIFO. Usually, these are used in systems where data need to pass from one clock domain to another which is generally termed as 'clock domain crossing'. Thus, asynchronous FIFO helps to synchronize data flow between two systems working on different clocks.



Asynchronous FIFO Block Diagram



Asynchronous FIFO

Signals:

wr_en: write enable

wr_data: write data

full: FIFO is full

empty: FIFO is empty

rd_en: read enable

rd_data: read data

b_wptr: binary write pointer

g_wptr: gray write pointer

b_wptr_next: binary write pointer next

g_wptr_next: gray write pointer next

b_rptr: binary read pointer

g_rptr: gray read pointer

b_rptr_next: binary read pointer next

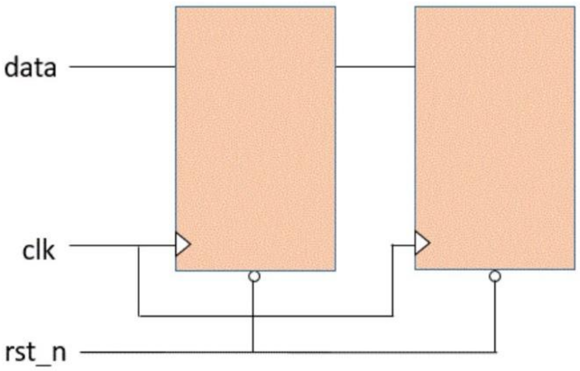
g_rptr_next: gray read pointer next

b_rptr_sync: binary read pointer synchronized

b_wptr_sync: binary write pointer synchronized

Asynchronous FIFO Operation

In the case of synchronous FIFO, the write and read pointers are generated on the same clock. However, in the case of asynchronous FIFO write pointer is aligned to the write clock domain whereas the read pointer is aligned to the read clock domain. Hence, it requires domain crossing to calculate FIFO full and empty conditions. This causes metastability in the actual design. In order to resolve this metastability, 2 flip flops or 3 flip flops synchronizer can be used to pass write and read pointers. For explanation, we will go with 2 flip-flop synchronizers. Please note that a single “2 FF synchronizer” can resolve metastability for only one bit. Hence, depending on write and read pointers multiple 2FF synchronizers are required.



2 flip-flop synchronizer

```
module synchronizer #(parameter WIDTH=3) (input clk, rst_n, [WIDTH:0] d_in,
output reg [WIDTH:0] d_out);
  reg [WIDTH:0] q1;
  always@(posedge clk) begin
    if(!rst_n) begin
      q1 <= 0;
      d_out <= 0;
    end
    else begin
      q1 <= d_in;
      d_out <= q1;
    end
  end
endmodule
```

Usage of Binary to Gray code converter and vice-versa in Asynchronous FIFO

Till now, we discussed how to get asynchronous write and read pointers in respective clock domains. However, we should not pass binary formatted write and read pointer values. Due to metastability, the overall write or read pointer value might be different.

Example: When binary value wr_ptr = 4'b1101 at the write clock domain is transferred via 2FF synchronizer, at the read clock domain wr_ptr value may receive as 4'b1111 or any other value that is not acceptable. Whereas gray code is assured to have only a single bit change from its previous value. Hence, both write and read pointers need to convert first to their equivalent gray code in their corresponding domain and then pass them to an opposite domain. To check FIFO full and empty conditions in another domain, we have two ways.

Way 1

Convert received gray code formatted pointers to binary format and then check for the full and empty conditions.

FIFO full condition

```
g2b_converter g2b_wr(g_rptr_sync, b_rptr_sync);

wrap_around = b_rptr_sync[PTR_WIDTH] ^ b_wptr[PTR_WIDTH];

wfull = wrap_around & (b_wptr[PTR_WIDTH-1:0] == b_rptr_sync[PTR_WIDTH-
1:0]);
```

FIFO empty condition

```
g2b_converter g2b_rd(g_wptr_sync, b_wptr_sync);

rempty = (b_wptr_sync == b_rptr_next);
```

Way 2

Check for full and empty conditions directly with the help of gray coded write and read pointer received. This is efficient as it does not need extra hardware for converting gray-coded write and read pointers to equivalent binary forms.

FIFO full condition

```
wfull = (g_wptr_next == {~g_rptr_sync[PTR_WIDTH:PTR_WIDTH-1],
g_rptr_sync[PTR_WIDTH-2:0]});
```

FIFO empty condition

```
rempty = (g_wptr_sync == g_rptr_next);
```

Asynchronous FIFO Verilog Code

Write Pointer Handler

The output of synchronizer g_rptr_sync is given as an input to ‘write pointer handler’ module used to generate the FIFO full condition. The binary write pointer (b_wptr) is incremented if it satisfies (w_en & !full) condition. This b_wptr value is fed to the fifo_mem module to write data into the FIFO.

```
module wptr_handler #(parameter PTR_WIDTH=3) (
  input wclk, wrst_n, w_en,
  input [PTR_WIDTH:0] g_rptr_sync,
  output reg [PTR_WIDTH:0] b_wptr, g_wptr,
  output reg full
);

  reg [PTR_WIDTH:0] b_wptr_next;
  reg [PTR_WIDTH:0] g_wptr_next;

  reg wrap_around;
  wire wfull;

  assign b_wptr_next = b_wptr+(w_en & !full);
  assign g_wptr_next = (b_wptr_next >>1)^b_wptr_next;

  always@(posedge wclk or negedge wrst_n) begin
    if(!wrst_n) begin
      b_wptr <= 0; // set default value
      g_wptr <= 0;
    end
    else begin
      b_wptr <= b_wptr_next; // incr binary write pointer
      g_wptr <= g_wptr_next; // incr gray write pointer
    end
  end

  always@(posedge wclk or negedge wrst_n) begin
    if(!wrst_n) full <= 0;
    else        full <= wfull;
  end

  assign wfull = (g_wptr_next == {~g_rptr_sync[PTR_WIDTH:PTR_WIDTH-1],
g_rptr_sync[PTR_WIDTH-2:0]});

endmodule
```

Read Pointer Handler

The output of synchronizer g_wptr_sync is given as an input to the ‘read pointer handler’ module to generate FIFO empty condition. The binary read pointer (b_rptr) is incremented if it satisfies (r_en & !empty) condition. This b_rptr value is fed to the fifo_mem module to read data from the FIFO.

```verilog
module rptr_handler #(parameter PTR_WIDTH=3) (
  input rclk, rrst_n, r_en,
  input [PTR_WIDTH:0] g_wptr_sync,
  output reg [PTR_WIDTH:0] b_rptr, g_rptr,
  output reg empty
);

  reg [PTR_WIDTH:0] b_rptr_next;
  reg [PTR_WIDTH:0] g_rptr_next;

  assign b_rptr_next = b_rptr+(r_en & !empty);
  assign g_rptr_next = (b_rptr_next >>1)^b_rptr_next;
  assign rempty = (g_wptr_sync == g_rptr_next);

  always@(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) begin
      b_rptr <= 0;
      g_rptr <= 0;
    end
    else begin
      b_rptr <= b_rptr_next;
      g_rptr <= g_rptr_next;
    end
  end

  always@(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) empty <= 1;
    else        empty <= rempty;
  end
endmodule
```

FIFO Memory

Based on binary coded write and read pointers data is written into the FIFO or read from the FIFO respectively.

```
module fifo_mem #(parameter DEPTH=8, DATA_WIDTH=8, PTR_WIDTH=3) (
  input wclk, w_en, rclk, r_en,
  input [PTR_WIDTH:0] b_wptr, b_rptr,
  input [DATA_WIDTH-1:0] data_in,
  input full, empty,
  output reg [DATA_WIDTH-1:0] data_out
);
  reg [DATA_WIDTH-1:0] fifo[0:DEPTH-1];

  always@(posedge wclk) begin
    if(w_en & !full) begin
      fifo[b_wptr[PTR_WIDTH-1:0]] <= data_in;
    end
  end
  /*
  always@(posedge rclk) begin
    if(r_en & !empty) begin
      data_out <= fifo[b_rptr[PTR_WIDTH-1:0]];
    end
  end
  */
  assign data_out = fifo[b_rptr[PTR_WIDTH-1:0]];
endmodule
```

Top Module

```
`include "synchronizer.v"
`include "wptr_handler.v"
`include "rptr_handler.v"
`include "fifo_mem.v"

module asynchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input wclk, wrst_n,
  input rclk, rrst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output reg full, empty
);

  parameter PTR_WIDTH = $clog2(DEPTH);

  reg [PTR_WIDTH:0] g_wptr_sync, g_rptr_sync;
  reg [PTR_WIDTH:0] b_wptr, b_rptr;
  reg [PTR_WIDTH:0] g_wptr, g_rptr;

  wire [PTR_WIDTH-1:0] waddr, raddr;
  synchronizer #(PTR_WIDTH) sync_wptr (rclk, rrst_n, g_wptr, g_wptr_sync);
//write pointer to read clock domain
  synchronizer #(PTR_WIDTH) sync_rptr (wclk, wrst_n, g_rptr, g_rptr_sync);
//read pointer to write clock domain

  wptr_handler #(PTR_WIDTH) wptr_h(wclk, wrst_n,
w_en,g_rptr_sync,b_wptr,g_wptr,full);
  rptr_handler #(PTR_WIDTH) rptr_h(rclk, rrst_n,
r_en,g_wptr_sync,b_rptr,g_rptr,empty);
  fifo_mem fifom(wclk, w_en, rclk, r_en,b_wptr, b_rptr, data_in,full,empty,
data_out);

endmodule
```

Testbench Code

```
module async_fifo_TB;

  parameter DATA_WIDTH = 8;

  wire [DATA_WIDTH-1:0] data_out;
  wire full;
  wire empty;
  reg [DATA_WIDTH-1:0] data_in;
  reg w_en, wclk, wrst_n;
  reg r_en, rclk, rrst_n;

  // Queue to push data_in
  reg [DATA_WIDTH-1:0] wdata_q[$], wdata;

  asynchronous_fifo as_fifo (wclk, wrst_n,rclk,
rrst_n,w_en,r_en,data_in,data_out,full,empty);

  always #10ns wclk = ~wclk;
  always #35ns rclk = ~rclk;

  initial begin
    wclk = 1'b0; wrst_n = 1'b0;
    w_en = 1'b0;
    data_in = 0;

    repeat(10) @(posedge wclk);
    wrst_n = 1'b1;

    repeat(2) begin
      for (int i=0; i<30; i++) begin
        @(posedge wclk iff !full);
        w_en = (i%2 == 0)? 1'b1 : 1'b0;
        if (w_en) begin
          data_in = $urandom;
          wdata_q.push_back(data_in);
        end
      end
      #50;
    end
```

```
 end

 initial begin
  rclk = 1'b0; rrst_n = 1'b0;
  r_en = 1'b0;

  repeat(20) @(posedge rclk);
  rrst_n = 1'b1;

  repeat(2) begin
   for (int i=0; i<30; i++) begin
    @(posedge rclk iff !empty);
    r_en = (i%2 == 0)? 1'b1 : 1'b0;
    if (r_en) begin
     wdata = wdata_q.pop_front();
     if(data_out !== wdata) $error("Time = %0t: Comparison Failed:
expected wr_data = %h, rd_data = %h", $time, wdata, data_out);
     else $display("Time = %0t: Comparison Passed: wr_data = %h and
rd_data = %h",$time, wdata, data_out);
    end
   end
   #50;
  end

  $finish;
 end

 initial begin
  $dumpfile("dump.vcd"); $dumpvars;
 end
endmodule
```

Output:

```
Time = 1575: Comparison Passed: wr_data = 51 and rd_data = 51
Time = 1715: Comparison Passed: wr_data = cd and rd_data = cd
Time = 1855: Comparison Passed: wr_data = 0e and rd_data = 0e
Time = 1995: Comparison Passed: wr_data = db and rd_data = db
Time = 2135: Comparison Passed: wr_data = 71 and rd_data = 71
Time = 2275: Comparison Passed: wr_data = 63 and rd_data = 63
Time = 2415: Comparison Passed: wr_data = e9 and rd_data = e9
Time = 2555: Comparison Passed: wr_data = 98 and rd_data = 98
Time = 2695: Comparison Passed: wr_data = 03 and rd_data = 03
Time = 2835: Comparison Passed: wr_data = a4 and rd_data = a4
Time = 2975: Comparison Passed: wr_data = a7 and rd_data = a7
Time = 3115: Comparison Passed: wr_data = 45 and rd_data = 45
Time = 3255: Comparison Passed: wr_data = 00 and rd_data = 00
Time = 3395: Comparison Passed: wr_data = 4f and rd_data = 4f
Time = 3535: Comparison Passed: wr_data = 3e and rd_data = 3e
Time = 3675: Comparison Passed: wr_data = e7 and rd_data = e7
Time = 3815: Comparison Passed: wr_data = d8 and rd_data = d8
Time = 3955: Comparison Passed: wr_data = 31 and rd_data = 31
Time = 4095: Comparison Passed: wr_data = 8b and rd_data = 8b
Time = 4235: Comparison Passed: wr_data = 07 and rd_data = 07
Time = 4375: Comparison Passed: wr_data = a1 and rd_data = a1
Time = 4515: Comparison Passed: wr_data = 15 and rd_data = 15
Time = 4655: Comparison Passed: wr_data = e6 and rd_data = e6
Time = 4795: Comparison Passed: wr_data = 80 and rd_data = 80
Time = 4935: Comparison Passed: wr_data = 01 and rd_data = 01
Time = 5075: Comparison Passed: wr_data = 72 and rd_data = 72
Time = 5215: Comparison Passed: wr_data = c8 and rd_data = c8
Time = 5355: Comparison Passed: wr_data = dc and rd_data = dc
Time = 5495: Comparison Passed: wr_data = d7 and rd_data = d7
```