VERILOG CODES

This document is a complete collection of Verilog HDL codes covering both combinational and sequential circuits.
It includes verified implementations for logic gates, adders, subtractors, multiplexers, decoders, latches, flip-flops, counters, and more — all written, tested, and optimized for clarity and learning.

The PDF serves as my personal Verilog reference and demonstrates my strong understanding of digital logic design and hardware modeling using HDL.

Prepared by: ***Siva Nagi Reddy***
Domain: VLSI – DFT Engineer

BASIC GATES

1. AND gate
2. OR gate
3. NOT gate
4. NAND gate
5. NOR gate
6. XOR gate
7. XNOR gate
8. 2-input gate all-in-one
9. 3-input AND gate
10. 4-input OR gate

1. AND Gate

➤ **Gate-Level**

Code of and_gate_gatelevel.v

```
module and_gate_gatelevel(input wire a, b, output wire y);
  and (y, a, b);
endmodule
```

➤ **Dataflow**

Code of and_gate_dataflow.v

```
module and_gate_dataflow(input wire a, b, output wire y);
  assign y = a & b;
endmodule
```

➤ **Behavioral**

Code of and_gate_behavioral.v

```
module and_gate_behavioral(input wire a, b, output reg y);
  always @(*) begin
    y = a & b;
  end
endmodule
```

➤ **Testbench**

Code of tb_and_gate.v

```
module tb_and_gate;
  reg a, b;
  wire y;
  and_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
  end
endmodule
```

2. OR Gate

➤ **Gate-Level**

Code of or_gate_gatelevel.v

```
module or_gate_gatelevel(input wire a, b, output wire y);
```

```
  or (y, a, b);
endmodule
```

➤ **Dataflow**

Code of or_gate_dataflow.v

```
module or_gate_dataflow(input wire a, b, output wire y);
  assign y = a | b;
endmodule
```

➤ **Behavioral**

Code of or_gate_behavioral.v

```
module or_gate_behavioral(input wire a, b, output reg y);
  always @(*) y = a | b;
endmodule
```

➤ **Testbench**

Code of tb_or_gate.v

```
module tb_or_gate;
  reg a, b;
  wire y;
  or_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
```

```
  end
endmodule
```

3. NOT Gate

➤ **Gate-Level**

Code of not_gate_gatelevel.v

```
module not_gate_gatelevel(input wire a, output wire y);
  not (y, a);
endmodule
```

➤ **Dataflow**

Code of not_gate_dataflow.v

```
module not_gate_dataflow(input wire a, output wire y);
  assign y = ~a;
endmodule
```

➤ **Behavioral**

Code of not_gate_behavioral.v

```
module not_gate_behavioral(input wire a, output reg y);
  always @(*) y = ~a;
endmodule
```

➤ **Testbench**

Code of tb_not_gate.v

```
module tb_not_gate;
  reg a;
  wire y;
  not_gate_dataflow uut (.a(a), .y(y));
  initial begin
```

```
    $display("A | Y");
    $monitor("%b | %b", a, y);
    a=0; #10;
    a=1; #10;
    $finish;
  end
endmodule
```

4. NAND Gate

➤ **Gate-Level**

Code of nand_gate_gatelevel.v

```
module nand_gate_gatelevel(input wire a, b, output wire y);
  nand (y, a, b);
endmodule
```

➤ **Dataflow**

Code of nand_gate_dataflow.v

```
module nand_gate_dataflow(input wire a, b, output wire y);
  assign y = ~(a & b);
endmodule
```

➤ **Behavioral**

Code of nand_gate_behavioral.v

```
module nand_gate_behavioral(input wire a, b, output reg y);
  always @(*) y = ~(a & b);
endmodule
```

➤ **Testbench**

Code of tb_nand_gate.v

```
module tb_nand_gate;
  reg a, b;
  wire y;
  nand_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
  end
endmodule
```

5. NOR Gate

➤ **Gate-Level**

Code of nor_gate_gatelevel.v

```
module nor_gate_gatelevel(input wire a, b, output wire y);
  nor (y, a, b);
endmodule
```

➤ **Dataflow**

Code of nor_gate_dataflow.v

```
module nor_gate_dataflow(input wire a, b, output wire y);
  assign y = ~(a | b);
endmodule
```

➤ **Behavioral**

Code of nor_gate_behavioral.v

```
module nor_gate_behavioral(input wire a, b, output reg y);
  always @(*) y = ~(a | b);
endmodule
```

➤ **Testbench**

Code of tb_nor_gate.v

```
module tb_nor_gate;
  reg a, b;
  wire y;
  nor_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
  end
endmodule
```

6. XOR Gate

➤ **Gate-Level**

Code of xor_gate_gatelevel.v

```
module xor_gate_gatelevel(input wire a, b, output wire y);
```

```
  xor (y, a, b);
endmodule
```

➤ **Dataflow**

Code of xor_gate_dataflow.v

```
module xor_gate_dataflow(input wire a, b, output wire y);
  assign y = a ^ b;
endmodule
```

➤ **Behavioral**

Code of xor_gate_behavioral.v

```
module xor_gate_behavioral(input wire a, b, output reg y);
  always @(*) y = a ^ b;
endmodule
```

➤ **Testbench**

Code of tb_xor_gate.v

```
module tb_xor_gate;
  reg a, b;
  wire y;
  xor_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
```

```
  end
endmodule
```

7. XNOR Gate

➤ **Gate-Level**

Code of xnor_gate_gatelevel.v

```
module xnor_gate_gatelevel(input wire a, b, output wire y);
  xnor (y, a, b);
endmodule
```

➤ **Dataflow**

Code of xnor_gate_dataflow.v

```
module xnor_gate_dataflow(input wire a, b, output wire y);
  assign y = ~(a ^ b);
endmodule
```

➤ **Behavioral**

Code of xnor_gate_behavioral.v

```
module xnor_gate_behavioral(input wire a, b, output reg y);
  always @(*) y = ~(a ^ b);
endmodule
```

➤ **Testbench**

Code of tb_xnor_gate.v

```
module tb_xnor_gate;
  reg a, b;
  wire y;
  xnor_gate_dataflow uut (.a(a), .b(b), .y(y));
  initial begin
```

```
    $display("A B | Y");
    $monitor("%b %b | %b", a, b, y);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
  end
endmodule
```

8. 2-Input All-in-One Gate (AND, OR, XOR, NAND, NOR, XNOR outputs)

Code of allinone_2input.v

```
module allinone_2input(input wire a, b,
                       output wire and_out, or_out, xor_out, nand_out, nor_out, xnor_out);
  assign and_out  = a & b;
  assign or_out   = a | b;
  assign xor_out  = a ^ b;
  assign nand_out = ~(a & b);
  assign nor_out  = ~(a | b);
  assign xnor_out = ~(a ^ b);
endmodule
```

➤ **Testbench**

Code of tb_allinone_2input.v

```
module tb_allinone_2input;
  reg a, b;
  wire and_out, or_out, xor_out, nand_out, nor_out, xnor_out;
```

```
 allinone_2input uut (.a(a), .b(b),
  .and_out(and_out), .or_out(or_out), .xor_out(xor_out),
  .nand_out(nand_out), .nor_out(nor_out), .xnor_out(xnor_out));
 initial begin
  $display("A B | AND OR XOR NAND NOR XNOR");
  $monitor("%b %b | %b   %b  %b   %b    %b    %b",
        a, b, and_out, or_out, xor_out, nand_out, nor_out, xnor_out);
  a=0; b=0; #10;
  a=0; b=1; #10;
  a=1; b=0; #10;
  a=1; b=1; #10;
  $finish;
 end
endmodule
```

9. 3-Input AND Gate

Code of and3_gate.v

```
module and3_gate(input wire a, b, c, output wire y);
 assign y = a & b & c;
endmodule
```

➤ **Testbench**

Code of tb_and3_gate.v

```
module tb_and3_gate;
 reg a, b, c;
 wire y;
 and3_gate uut (.a(a), .b(b), .c(c), .y(y));
```

```
 initial begin
  $display("A B C | Y");
  $monitor("%b %b %b | %b", a, b, c, y);
  {a,b,c}=3'b000; #10;
  {a,b,c}=3'b001; #10;
  {a,b,c}=3'b010; #10;
  {a,b,c}=3'b011; #10;
  {a,b,c}=3'b100; #10;
  {a,b,c}=3'b101; #10;
  {a,b,c}=3'b110; #10;
  {a,b,c}=3'b111; #10;
  $finish;
 end
endmodule
```

10. 4-Input OR Gate

Code of or4_gate.v

```
module or4_gate(input wire a, b, c, d, output wire y);
 assign y = a | b | c | d;
endmodule
```

➤ **Testbench**

Code of tb_or4_gate.v

```
module tb_or4_gate;
 reg a, b, c, d;
 wire y;
 or4_gate uut (.a(a), .b(b), .c(c), .d(d), .y(y));
```

```
 initial begin
  $display("A B C D | Y");
  $monitor("%b %b %b %b | %b", a, b, c, d, y);
  {a,b,c,d}=4'b0000; #10;
  {a,b,c,d}=4'b0001; #10;
  {a,b,c,d}=4'b0010; #10;
  {a,b,c,d}=4'b1111; #10;
  $finish;
 end
endmodule
```

B. COMBINATIONAL CIRCUITS

11. Half Adder
12. Full Adder
13. Half Subtractor
14. Full Subtractor
15. 4-bit Ripple Carry Adder
16. 4-bit Adder/Subtractor
17. 4-bit Carry Lookahead Adder
18. 8-bit Adder
19. 2-bit Comparator
20. 4-bit Comparator
21. Binary to Gray
22. Gray to Binary
23. Binary to BCD
24. BCD to Excess-3
25. 2-to-1 Mux

26. 4-to-1 Mux

27. 8-to-1 Mux

28. 1-to-2 Demux

29. 1-to-4 Demux

30. 1-to-8 Demux

31. 2-to-4 Decoder

32. 3-to-8 Decoder

33. 4-to-16 Decoder

34. 4-to-2 Encoder

35. 8-to-3 Encoder

36. Priority Encoder (8x3)

37. Parity Bit Generator

38. Majority Detector

39. Even–Odd Detector

11) Half Adder

half_adder_dataflow.v

```
module half_adder_dataflow(input wire a,b, output wire sum, carry);
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

half_adder_behavioral.v

```
module half_adder_behavioral(input wire a,b, output reg sum, carry);
  always @(*) begin
    sum = a ^ b;
    carry = a & b;
```

```
  end
endmodule
```

half_adder_gatelevel.v

```
module half_adder_gatelevel(input wire a,b, output wire sum, carry);
  xor (sum, a, b);
  and (carry, a, b);
endmodule
```

tb_half_adder.v

```
module tb_half_adder;
  reg a,b; wire sum,carry;
  half_adder_dataflow uut(.a(a),.b(b),.sum(sum),.carry(carry));
  initial begin
    $display("a b | sum carry");
    $monitor("%b %b |  %b    %b",a,b,sum,carry);
    {a,b}=2'b00; #5;
    {a,b}=2'b01; #5;
    {a,b}=2'b10; #5;
    {a,b}=2'b11; #5;
    $finish;
  end
endmodule
```

12) Full Adder

full_adder_dataflow.v

```
module full_adder_dataflow(input wire a,b,cin, output wire sum, cout);
  assign sum = a ^ b ^ cin;
```

```
  assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

full_adder_behavioral.v

```
module full_adder_behavioral(input wire a,b,cin, output reg sum, cout);
  always @(*) begin
    sum = a ^ b ^ cin;
    cout = (a & b) | (b & cin) | (a & cin);
  end
endmodule
```

full_adder_gatelevel.v

```
module full_adder_gatelevel(input wire a,b,cin, output wire sum, cout);
  wire s1,c1,c2;
  xor (s1,a,b);
  xor (sum,s1,cin);
  and (c1,a,b);
  and (c2,s1,cin);
  or  (cout,c1,c2);
endmodule
```

tb_full_adder.v

```
module tb_full_adder;
  reg a,b,cin; wire sum,cout;
  full_adder_dataflow uut(.a(a),.b(b),.cin(cin),.sum(sum),.cout(cout));
  initial begin
    $display("a b cin | sum cout");
    $monitor("%b %b  %b  |  %b   %b",a,b,cin,sum,cout);
    {a,b,cin}=3'b000; #5;
```

```
    {a,b,cin}=3'b001; #5;
    {a,b,cin}=3'b010; #5;
    {a,b,cin}=3'b011; #5;
    {a,b,cin}=3'b100; #5;
    {a,b,cin}=3'b101; #5;
    {a,b,cin}=3'b110; #5;
    {a,b,cin}=3'b111; #5;
    $finish;
  end
endmodule
```

13) Half Subtractor

half_subtractor_dataflow.v

```
module half_subtractor_dataflow(input wire a,b, output wire diff, borrow);
  assign diff = a ^ b;
  assign borrow = (~a) & b;
endmodule
```

half_subtractor_behavioral.v

```
module half_subtractor_behavioral(input wire a,b, output reg diff, borrow);
  always @(*) begin
    diff = a ^ b;
    borrow = (~a) & b;
  end
endmodule
```

half_subtractor_gatelevel.v

```
module half_subtractor_gatelevel(input wire a,b, output wire diff, borrow);
```

```
  xor (diff, a, b);
  not (an, a);
  and (borrow, an, b);
endmodule
```

tb_half_subtractor.v

```
module tb_half_subtractor;
  reg a,b; wire diff,borrow;
  half_subtractor_dataflow uut(.a(a),.b(b),.diff(diff),.borrow(borrow));
  initial begin
    $display("a b | diff borrow");
    $monitor("%b %b |  %b    %b",a,b,diff,borrow);
    {a,b}=2'b00; #5;
    {a,b}=2'b01; #5;
    {a,b}=2'b10; #5;
    {a,b}=2'b11; #5;
    $finish;
  end
endmodule
```

14) Full Subtractor

full_subtractor_dataflow.v

```
module full_subtractor_dataflow(input wire a,b,bin, output wire diff, bout);
  assign diff = a ^ b ^ bin;
  assign bout = (~a & b) | ((~a | b) & bin);
endmodule
```

full_subtractor_behavioral.v

```
module full_subtractor_behavioral(input wire a,b,bin, output reg diff, bout);
  always @(*) begin
    diff = a ^ b ^ bin;
    bout = (~a & b) | ((~a | b) & bin);
  end
endmodule
```

full_subtractor_gatelevel.v

```
module full_subtractor_gatelevel(input wire a,b,bin, output wire diff, bout);
  wire d1, b1, b2, an;
  xor (d1, a, b);
  xor (diff, d1, bin);
  not (an, a);
  and (b1, an, b);
  and (b2, d1, bin);
  or  (bout, b1, b2);
endmodule
```

tb_full_subtractor.v

```
module tb_full_subtractor;
  reg a,b,bin; wire diff,bout;
  full_subtractor_dataflow uut(.a(a),.b(b),.bin(bin),.diff(diff),.bout(bout));
  initial begin
    $display("a b bin | diff bout");
    $monitor("%b %b  %b  |  %b   %b",a,b,bin,diff,bout);
    {a,b,bin}=3'b000; #5;
    {a,b,bin}=3'b001; #5;
    {a,b,bin}=3'b010; #5;
```

```
   {a,b,bin}=3'b011; #5;
   {a,b,bin}=3'b100; #5;
   {a,b,bin}=3'b101; #5;
   {a,b,bin}=3'b110; #5;
   {a,b,bin}=3'b111; #5;
   $finish;
  end
endmodule
```

15) 4-bit Ripple Carry Adder

adder4_dataflow.v

```
module adder4_dataflow(input wire [3:0] a,b, input wire cin, output wire [3:0] sum, output wire cout);
  assign {cout, sum} = a + b + cin;
endmodule
```

adder4_behavioral.v

```
module adder4_behavioral(input wire [3:0] a,b, input wire cin, output reg [3:0] sum, output reg cout);
  reg [4:0] tmp;
  always @(*) begin
    tmp = a + b + cin;
    sum = tmp[3:0];
    cout = tmp[4];
  end
endmodule
```

adder4_gatelevel.v (structural using fa)

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

```
  wire s1,c1,c2;
  xor (s1,a,b);
  xor (sum,s1,cin);
  and (c1,a,b);
  and (c2,s1,cin);
  or  (cout,c1,c2);
endmodule
module adder4_gatelevel(input [3:0] a,b, input cin, output [3:0] sum, output cout);
  wire c1,c2,c3;
  fa fa0(a[0],b[0],cin, sum[0], c1);
  fa fa1(a[1],b[1],c1,  sum[1], c2);
  fa fa2(a[2],b[2],c2,  sum[2], c3);
  fa fa3(a[3],b[3],c3,  sum[3], cout);
endmodule
```

tb_adder4.v

```
module tb_adder4;
  reg [3:0] a,b; reg cin; wire [3:0] sum; wire cout;
  adder4_dataflow uut(.a(a),.b(b),.cin(cin),.sum(sum),.cout(cout));
  initial begin
    $display("a b cin | cout sum");
    a=4'd3; b=4'd2; cin=0; #5;
    $monitor("%d %d  %b  |  %b   %d",a,b,cin,cout,sum);
    a=4'd7; b=4'd8; cin=1; #5;
    a=4'd15; b=4'd1; cin=0; #5;
    $finish;
  end
```

```
endmodule
```

16) 4-bit Adder/Subtractor

addsub4_dataflow.v

```
module addsub4_dataflow(input wire [3:0] a,b, input wire mode, output wire [3:0] result,
output wire cout);
  wire [3:0] bx;
  assign bx = b ^ {4{mode}}; // if mode==1 then invert b (two's complement)
  assign {cout, result} = a + bx + mode;
endmodule
```

addsub4_behavioral.v

```
module addsub4_behavioral(input wire [3:0] a,b, input wire mode, output reg [3:0] result,
output reg cout);
  reg [4:0] tmp;
  always @(*) begin
    if (mode) tmp = a + (~b) + 1;
    else tmp = a + b;
    result = tmp[3:0];
    cout = tmp[4];
  end
endmodule
```

addsub4_gatelevel.v (uses fa_struct and bx xor)

```
module addsub4_gatelevel(input [3:0] a,b, input mode, output [3:0] result, output cout);
  wire [3:0] bx;
  assign bx = b ^ {4{mode}};
  wire c1,c2,c3;
  fa fa0(a[0],bx[0],mode,result[0],c1);
```

```
  fa fa1(a[1],bx[1],c1,    result[1],c2);
  fa fa2(a[2],bx[2],c2,    result[2],c3);
  fa fa3(a[3],bx[3],c3,    result[3],cout);
endmodule
```

tb_addsub4.v

```
module tb_addsub4;
  reg [3:0] a,b; reg mode; wire [3:0] result; wire cout;
  addsub4_dataflow uut(.a(a),.b(b),.mode(mode),.result(result),.cout(cout));
  initial begin
    a=4'd9; b=4'd3; mode=0; #5;
    $display("add: a=%d b=%d mode=%b => result=%d cout=%b",a,b,mode,result,cout);
    mode=1; #5;
    $display("sub: a=%d b=%d mode=%b => result=%d cout=%b",a,b,mode,result,cout);
    $finish;
  end
endmodule
```

17) 4-bit Carry Lookahead Adder

cla4_dataflow.v (fast logic for generate/propagate)

```
module cla4_dataflow(input wire [3:0] a,b, input wire cin, output wire [3:0] sum, output wire
cout);
  wire [3:0] p,g;
  wire c1,c2,c3;
  assign p = a ^ b;
  assign g = a & b;
  assign c1 = g[0] | (p[0] & cin);
```

```
  assign c2 = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
  assign c3 = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0] & cin);
  assign cout = g[3] | (p[3] & c3);
  assign sum = p ^ {c3,c2,c1,cin};
endmodule
```

cla4_behavioral.v

```
module cla4_behavioral(input wire [3:0] a,b, input wire cin, output reg [3:0] sum, output reg
cout);
  reg [4:0] tmp;
  always @(*) begin
    tmp = a + b + cin;
    sum = tmp[3:0];
    cout = tmp[4];
  end
endmodule
```

cla4_gatelevel.v (structural-ish: uses the above equations)

```
module cla4_gatelevel(input [3:0] a,b, input cin, output [3:0] sum, output cout);
  wire [3:0] p,g;
  wire c1,c2,c3;
  assign p = a ^ b;
  assign g = a & b;
  assign c1 = g[0] | (p[0] & cin);
  assign c2 = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
  assign c3 = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0] & cin);
  assign cout = g[3] | (p[3] & c3);
  assign sum = p ^ {c3,c2,c1,cin};
```

```
endmodule
```

tb_cla4.v

```
module tb_cla4;
  reg [3:0] a,b; reg cin; wire [3:0] sum; wire cout;
  cla4_dataflow uut(.a(a),.b(b),.cin(cin),.sum(sum),.cout(cout));
  initial begin
    a=4'd3; b=4'd5; cin=0; #5;
    $display("a=%d b=%d cin=%b => sum=%d cout=%b",a,b,cin,sum,cout);
    a=4'd9; b=4'd7; cin=1; #5;
    $display("a=%d b=%d cin=%b => sum=%d cout=%b",a,b,cin,sum,cout);
    $finish;
  end
endmodule
```

18) 8-bit Adder

adder8_dataflow.v

```
module adder8_dataflow(input wire [7:0] a,b, output wire [7:0] sum, output wire cout);
  assign {cout, sum} = a + b;
endmodule
```

adder8_behavioral.v

```
module adder8_behavioral(input wire [7:0] a,b, output reg [7:0] sum, output reg cout);
  reg [8:0] tmp;
  always @(*) begin
    tmp = a + b;
    sum = tmp[7:0];
    cout = tmp[8];
```

```
  end
endmodule
```

adder8_gatelevel.v

```
module adder8_gatelevel(input [7:0] a,b, output [7:0] sum, output cout);
  // Instantiate two 4-bit adders (reuse previous adder4_gatelevel)
  wire c4;
  adder4_gatelevel low(.a(a[3:0]),.b(b[3:0]),.cin(1'b0),.sum(sum[3:0]),.cout(c4));
  adder4_gatelevel high(.a(a[7:4]),.b(b[7:4]),.cin(c4),.sum(sum[7:4]),.cout(cout));
endmodule
```

tb_adder8.v

```
module tb_adder8;
  reg [7:0] a,b; wire [7:0] sum; wire cout;
  adder8_dataflow uut(.a(a),.b(b),.sum(sum),.cout(cout));
  initial begin
    a=8'd120; b=8'd50; #5;
    $display("a=%d b=%d => sum=%d cout=%b",a,b,sum,cout);
    $finish;
  end
endmodule
```

19) 2-bit Comparator

cmp2_dataflow.v

```
module cmp2_dataflow(input wire [1:0] a,b, output wire eq, gt, lt);
  assign eq = (a == b);
  assign gt = (a > b);
  assign lt = (a < b);
```

```
endmodule
```

cmp2_behavioral.v

```
module cmp2_behavioral(input wire [1:0] a,b, output reg eq, gt, lt);
  always @(*) begin
    eq = (a == b);
    gt = (a > b);
    lt = (a < b);
  end
endmodule
```

cmp2_gatelevel.v

```
module cmp2_gatelevel(input [1:0] a,b, output eq, gt, lt);
  assign eq = (a == b);
  assign gt = (a > b);
  assign lt = (a < b);
endmodule
```

tb_cmp2.v

```
module tb_cmp2;
  reg [1:0] a,b; wire eq,gt,lt;
  cmp2_dataflow uut(.a(a),.b(b),.eq(eq),.gt(gt),.lt(lt));
  initial begin
    a=2'b00; b=2'b01; #5;
    $display("a=%b b=%b => eq=%b gt=%b lt=%b",a,b,eq,gt,lt);
    a=2'b10; b=2'b01; #5;
    a=2'b11; b=2'b11; #5;
    $finish;
  end
```

```
endmodule
```

20) 4-bit Comparator

cmp4_dataflow.v

```
module cmp4_dataflow(input wire [3:0] a,b, output wire eq, gt, lt);
  assign eq = (a == b);
  assign gt = (a > b);
  assign lt = (a < b);
endmodule
```

cmp4_behavioral.v

```
module cmp4_behavioral(input wire [3:0] a,b, output reg eq, gt, lt);
  always @(*) begin
    eq = (a == b);
    gt = (a > b);
    lt = (a < b);
  end
endmodule
```

cmp4_gatelevel.v

```
module cmp4_gatelevel(input [3:0] a,b, output eq, gt, lt);
  assign eq = (a == b);
  assign gt = (a > b);
  assign lt = (a < b);
endmodule
```

tb_cmp4.v

```
module tb_cmp4;
  reg [3:0] a,b; wire eq,gt,lt;
```

```
 cmp4_dataflow uut(.a(a),.b(b),.eq(eq),.gt(gt),.lt(lt));
 initial begin
  a=4'd5; b=4'd7; #5;
  $display("%d %d => eq=%b gt=%b lt=%b",a,b,eq,gt,lt);
  a=4'd9; b=4'd3; #5;
  a=4'd6; b=4'd6; #5;
  $finish;
 end
endmodule
```

21) Binary to Gray

bin2gray_dataflow.v

```
module bin2gray_dataflow(input wire [2:0] b, output wire [2:0] g);
 assign g[2] = b[2];
 assign g[1] = b[2] ^ b[1];
 assign g[0] = b[1] ^ b[0];
endmodule
```

bin2gray_behavioral.v

```
module bin2gray_behavioral(input wire [2:0] b, output reg [2:0] g);
 always @(*) begin
  g[2] = b[2];
  g[1] = b[2] ^ b[1];
  g[0] = b[1] ^ b[0];
 end
endmodule
```

bin2gray_gatelevel.v

```
module bin2gray_gatelevel(input [2:0] b, output [2:0] g);
  xor (g[1], b[2], b[1]);
  xor (g[0], b[1], b[0]);
  assign g[2] = b[2];
endmodule
```

tb_bin2gray.v

```
module tb_bin2gray;
  reg [2:0] b; wire [2:0] g;
  bin2gray_dataflow uut(.b(b),.g(g));
  initial begin
    b=3'b000; #2;
    $display("b g"); $monitor("%b %b",b,g);
    b=3'b001; #2; b=3'b010; #2; b=3'b011; #2; b=3'b100; #2; b=3'b101; #2; b=3'b110; #2;
b=3'b111; #2;
    $finish;
  end
endmodule
```

22) Gray to Binary

gray2bin_dataflow.v

```
module gray2bin_dataflow(input wire [2:0] g, output wire [2:0] b);
  assign b[2] = g[2];
  assign b[1] = g[2] ^ g[1];
  assign b[0] = b[1] ^ g[0];
```

```
endmodule
```

gray2bin_behavioral.v

```
module gray2bin_behavioral(input wire [2:0] g, output reg [2:0] b);
  always @(*) begin
    b[2] = g[2];
    b[1] = g[2] ^ g[1];
    b[0] = b[1] ^ g[0];
  end
endmodule
```

gray2bin_gatelevel.v

```
module gray2bin_gatelevel(input [2:0] g, output [2:0] b);
  assign b[2] = g[2];
  assign b[1] = g[2] ^ g[1];
  assign b[0] = (g[2] ^ g[1]) ^ g[0];
endmodule
```

tb_gray2bin.v

```
module tb_gray2bin;
  reg [2:0] g; wire [2:0] b;
  gray2bin_dataflow uut(.g(g),.b(b));
  initial begin
    g=3'b000; #2; g=3'b001; #2; g=3'b010; #2; g=3'b011; #2; g=3'b100; #2; g=3'b101; #2;
g=3'b110; #2; g=3'b111; #2;
    $display("g b"); $monitor("%b %b",g,b);
    $finish;
  end
endmodule
```

23) Binary to BCD

(behavioural conversion via double dabble simplified for 4-bit)

bin2bcd_behavioral.v

```
module bin2bcd_behavioral(input wire [3:0] bin, output reg [7:0] bcd);
  integer i;
  reg [7:0] shift;
  always @(*) begin
    shift = 8'b0;
    shift[3:0] = bin;
    for (i = 0; i < 4; i = i + 1) begin
      if (shift[7:4] >= 5) shift[7:4] = shift[7:4] + 3;
      if (shift[3:0] >= 5) shift[3:0] = shift[3:0] + 3;
      shift = {shift[6:0], 1'b0};
    end
    bcd = shift;
  end
endmodule
```

bin2bcd_dataflow.v (simple for 0-15 mapping)

```
module bin2bcd_dataflow(input wire [3:0] bin, output reg [7:0] bcd);
  always @(*) begin
    case (bin)
      4'd0: bcd = 8'd0;
      4'd1: bcd = 8'd1;
      4'd2: bcd = 8'd2;
      4'd3: bcd = 8'd3;
```

```
    4'd4: bcd = 8'd4;
    4'd5: bcd = 8'd5;
    4'd6: bcd = 8'd6;
    4'd7: bcd = 8'd7;
    4'd8: bcd = 8'd8;
    4'd9: bcd = 8'd9;
    4'd10: bcd = 8'd10;
    4'd11: bcd = 8'd11;
    4'd12: bcd = 8'd12;
    4'd13: bcd = 8'd13;
    4'd14: bcd = 8'd14;
    4'd15: bcd = 8'd15;
    default: bcd = 8'd0;
  endcase
 end
endmodule
```

bin2bcd_gatelevel.v (use behavioral mapping)

```
module bin2bcd_gatelevel(input [3:0] bin, output [7:0] bcd);
 // Gate-level complex; use mapping combinational
 reg [7:0] bcd_r;
 always @(*) begin
  case(bin)
   default: bcd_r = {4'd0, bin};
   // mapping above same as dataflow
  endcase
 end
```

```
  assign bcd = bcd_r;
endmodule
```

tb_bin2bcd.v

```
module tb_bin2bcd;
  reg [3:0] bin; wire [7:0] bcd;
  bin2bcd_behavioral uut(.bin(bin),.bcd(bcd));
  initial begin
    $display("bin bcd");
    for (bin=0; bin<16; bin=bin+1) #2 $display("%d -> %d", bin, bcd);
    $finish;
  end
endmodule
```

24) BCD to Excess-3

bcd2ex3_dataflow.v

```
module bcd2ex3_dataflow(input wire [3:0] bcd, output wire [3:0] ex3);
  assign ex3 = bcd + 4'b0011;
endmodule
```

bcd2ex3_behavioral.v

```
module bcd2ex3_behavioral(input wire [3:0] bcd, output reg [3:0] ex3);
  always @(*) ex3 = bcd + 4'd3;
endmodule
```

bcd2ex3_gatelevel.v

```
module bcd2ex3_gatelevel(input [3:0] bcd, output [3:0] ex3);
  assign ex3 = bcd + 4'b0011;
endmodule
```

tb_bcd2ex3.v

```
module tb_bcd2ex3;
  reg [3:0] bcd; wire [3:0] ex3;
  bcd2ex3_dataflow uut(.bcd(bcd),.ex3(ex3));
  initial begin
    bcd=4'd0; #2; $display("%d -> %d",bcd,ex3);
    bcd=4'd5; #2; $display("%d -> %d",bcd,ex3);
    bcd=4'd9; #2; $display("%d -> %d",bcd,ex3);
    $finish;
  end
endmodule
```

25) 2-to-1 Mux

mux2_dataflow.v

```
module mux2_dataflow(input wire i0,i1, input wire sel, output wire y);
  assign y = sel ? i1 : i0;
endmodule
```

mux2_behavioral.v

```
module mux2_behavioral(input wire i0,i1, input wire sel, output reg y);
  always @(*) y = sel ? i1 : i0;
endmodule
```

mux2_gatelevel.v

```
module mux2_gatelevel(input i0,i1, input sel, output y);
  wire seln,a0,a1;
  not (seln, sel);
  and (a0, i0, seln);
```

```
  and (a1, i1, sel);
  or  (y, a0, a1);
endmodule
```

tb_mux2.v

```
module tb_mux2;
  reg i0,i1,sel; wire y;
  mux2_dataflow uut(.i0(i0),.i1(i1),.sel(sel),.y(y));
  initial begin
    i0=0;i1=1; sel=0; #2;
    $display("sel y = %b %b", sel, y);
    sel=1; #2; $display("sel y = %b %b", sel, y);
    $finish;
  end
endmodule
```

26) 4-to-1 Mux

mux4_dataflow.v

```
module mux4_dataflow(input wire [3:0] i, input wire [1:0] sel, output wire y);
  assign y = (sel == 2'b00) ? i[0] :
             (sel == 2'b01) ? i[1] :
             (sel == 2'b10) ? i[2] : i[3];
endmodule
```

mux4_behavioral.v

```
module mux4_behavioral(input wire [3:0] i, input wire [1:0] sel, output reg y);
  always @(*) case(sel) 2'b00: y=i[0]; 2'b01: y=i[1]; 2'b10: y=i[2]; default: y=i[3]; endcase
endmodule
```

mux4_gatelevel.v

```
module mux4_gatelevel(input [3:0] i, input [1:0] sel, output y);
  wire s0n,s1n,a0,a1,a2,a3;
  not (s0n, sel[0]); not (s1n, sel[1]);
  and (a0, i[0], s1n, s0n);
  and (a1, i[1], s1n, sel[0]);
  and (a2, i[2], sel[1], s0n);
  and (a3, i[3], sel[1], sel[0]);
  or (y, a0, a1, a2, a3);
endmodule
```

tb_mux4.v

```
module tb_mux4;
  reg [3:0] i; reg [1:0] sel; wire y;
  mux4_dataflow uut(.i(i),.sel(sel),.y(y));
  initial begin
    i = 4'b1010;
    sel = 2'b00; #2;
    $display("sel=%b y=%b", sel, y);
    sel = 2'b01; #2; sel = 2'b10; #2; sel = 2'b11; #2;
    $finish;
  end
endmodule
```

27) 8-to-1 Mux

mux8_dataflow.v

```
module mux8_dataflow(input wire [7:0] i, input wire [2:0] sel, output wire y);
```

```
  assign y = i[sel];
endmodule
```

mux8_behavioral.v

```
module mux8_behavioral(input wire [7:0] i, input wire [2:0] sel, output reg y);
  always @(*) y = i[sel];
endmodule
```

mux8_gatelevel.v

```
module mux8_gatelevel(input [7:0] i, input [2:0] sel, output y);
  // gate-level: chain of decoders & ands; omitted for brevity, use behavioral for clarity
  assign y = i[sel];
endmodule
```

tb_mux8.v

```
module tb_mux8;
  reg [7:0] i; reg [2:0] sel; wire y;
  mux8_dataflow uut(.i(i),.sel(sel),.y(y));
  initial begin
    i = 8'b01010101;
    sel = 3'd0; #2; sel=3'd3; #2; sel=3'd7; #2;
    $finish;
  end
endmodule
```

28) 1-to-2 Demux

demux1to2_dataflow.v

```
module demux1to2_dataflow(input wire i, input wire sel, output wire y0,y1);
  assign y0 = (~sel) & i;
```

```
  assign y1 = sel & i;
endmodule
```

demux1to2_behavioral.v

```
module demux1to2_behavioral(input i, sel, output reg y0,y1);
  always @(*) begin
    y0 = 0; y1 = 0;
    if (sel) y1 = i; else y0 = i;
  end
endmodule
```

demux1to2_gatelevel.v

```
module demux1to2_gatelevel(input i, sel, output y0,y1);
  wire seln;
  not (seln, sel);
  and (y0, i, seln);
  and (y1, i, sel);
endmodule
```

tb_demux1to2.v

```
module tb_demux1to2;
  reg i, sel; wire y0,y1;
  demux1to2_dataflow uut(.i(i),.sel(sel),.y0(y0),.y1(y1));
  initial begin
    i=1; sel=0; #2;
    $display("sel=%b y0=%b y1=%b",sel,y0,y1);
    sel=1; #2; $display("sel=%b y0=%b y1=%b",sel,y0,y1);
    $finish;
  end
```

```
endmodule
```

29) 1-to-4 Demux

demux1to4_dataflow.v

```
module demux1to4_dataflow(input wire i, input wire [1:0] sel, output wire [3:0] y);
  assign y = (i<<sel);
endmodule
```

demux1to4_behavioral.v

```
module demux1to4_behavioral(input i, input [1:0] sel, output reg [3:0] y);
  always @(*) begin y = 4'b0000; y[sel] = i; end
endmodule
```

demux1to4_gatelevel.v

```
module demux1to4_gatelevel(input i, input [1:0] sel, output [3:0] y);
  wire s0n,s1n;
  not (s0n, sel[0]); not (s1n, sel[1]);
  and (y[0], i, s1n, s0n);
  and (y[1], i, s1n, sel[0]);
  and (y[2], i, sel[1], s0n);
  and (y[3], i, sel[1], sel[0]);
endmodule
```

tb_demux1to4.v

```
module tb_demux1to4;
  reg i; reg [1:0] sel; wire [3:0] y;
  demux1to4_dataflow uut(.i(i),.sel(sel),.y(y));
  initial begin
    i=1; sel=0; #2; $display("sel=%b y=%b",sel,y);
```

```
   sel=1; #2; sel=2; #2; sel=3; #2;
   $finish;
  end
endmodule
```

30) 1-to-8 Demux

demux1to8_dataflow.v

```
module demux1to8_dataflow(input wire i, input wire [2:0] sel, output reg [7:0] y);
  always @(*) begin y = 8'b0; y[sel] = i; end
endmodule
```

demux1to8_behavioral.v

```
module demux1to8_behavioral(input i, input [2:0] sel, output reg [7:0] y);
  always @(*) begin y = 8'b0; y[sel] = i; end
endmodule
```

demux1to8_gatelevel.v

```
module demux1to8_gatelevel(input i, input [2:0] sel, output [7:0] y);
  // gate-level expansion omitted for brevity; use behavioral/dataflow
  reg [7:0] yr;
  always @(*) begin yr = 8'b0; yr[sel] = i; end
  assign y = yr;
endmodule
```

tb_demux1to8.v

```
module tb_demux1to8;
  reg i; reg [2:0] sel; wire [7:0] y;
  demux1to8_dataflow uut(.i(i),.sel(sel),.y(y));
  initial begin
```

```
    i=1; sel=0; #2; sel=3; #2; sel=7; #2;
    $finish;
  end
endmodule
```

31) 2-to-4 Decoder

decoder2to4_dataflow.v

```
module decoder2to4_dataflow(input [1:0] a, output [3:0] y);
  assign y = 4'b0001 << a;
endmodule
```

decoder2to4_behavioral.v

```
module decoder2to4_behavioral(input [1:0] a, output reg [3:0] y);
  always @(*) begin
    y = 4'b0000;
    case(a) 2'b00: y=4'b0001; 2'b01: y=4'b0010; 2'b10: y=4'b0100; 2'b11: y=4'b1000;
    endcase
  end
endmodule
```

decoder2to4_gatelevel.v

```
module decoder2to4_gatelevel(input [1:0] a, output [3:0] y);
  wire a0n,a1n;
  not (a0n, a[0]); not (a1n, a[1]);
  and (y[0], a1n, a0n);
  and (y[1], a1n, a[0]);
  and (y[2], a[1], a0n);
  and (y[3], a[1], a[0]);
```

```
endmodule
```

tb_decoder2to4.v

```
module tb_decoder2to4;
  reg [1:0] a; wire [3:0] y;
  decoder2to4_dataflow uut(.a(a),.y(y));
  initial begin
    a=2'b00; #2; a=2'b01; #2; a=2'b10; #2; a=2'b11; #2; $finish;
  end
endmodule
```

32) 3-to-8 Decoder

decoder3to8_dataflow.v

```
module decoder3to8_dataflow(input [2:0] a, output [7:0] y);
  assign y = 8'b1 << a;
endmodule
```

decoder3to8_behavioral.v

```
module decoder3to8_behavioral(input [2:0] a, output reg [7:0] y);
  always @(*) begin y = 8'b0; y[a] = 1'b1; end
endmodule
```

decoder3to8_gatelevel.v

```
module decoder3to8_gatelevel(input [2:0] a, output [7:0] y);
  reg [7:0] yr;
  always @(*) begin yr = 8'b0; yr[a] = 1'b1; end
  assign y = yr;
```

```
endmodule
```

tb_decoder3to8.v

```
module tb_decoder3to8;
  reg [2:0] a; wire [7:0] y;
  decoder3to8_dataflow uut(.a(a),.y(y));
  initial begin
    a=3'd0; #2; a=3'd3; #2; a=3'd7; #2; $finish;
  end
endmodule
```

33) 4-to-16 Decoder

decoder4to16_dataflow.v

```
module decoder4to16_dataflow(input [3:0] a, output [15:0] y);
  assign y = 16'b1 << a;
endmodule
```

decoder4to16_behavioral.v

```
module decoder4to16_behavioral(input [3:0] a, output reg [15:0] y);
  always @(*) begin y = 16'b0; y[a] = 1'b1; end
endmodule
```

decoder4to16_gatelevel.v

```
module decoder4to16_gatelevel(input [3:0] a, output [15:0] y);
  reg [15:0] yr;
  always @(*) begin yr = 16'b0; yr[a] = 1'b1; end
```

```
  assign y = yr;
endmodule
```

tb_decoder4to16.v

```
module tb_decoder4to16;
  reg [3:0] a; wire [15:0] y;
  decoder4to16_dataflow uut(.a(a),.y(y));
  initial begin a=4'd0; #2; a=4'd5; #2; a=4'd15; #2; $finish; end
endmodule
```

34) 4-to-2 Encoder

encoder4to2_dataflow.v

```
module encoder4to2_dataflow(input [3:0] i, output [1:0] y);
  assign y[0] = i[1] | i[3];
  assign y[1] = i[2] | i[3];
endmodule
```

encoder4to2_behavioral.v

```
module encoder4to2_behavioral(input [3:0] i, output reg [1:0] y);
  always @(*) begin
    if (i[3]) y = 2'b11;
    else if (i[2]) y = 2'b10;
    else if (i[1]) y = 2'b01;
    else y = 2'b00;
  end
endmodule
```

encoder4to2_gatelevel.v

```
module encoder4to2_gatelevel(input [3:0] i, output [1:0] y);
```

```
  assign y[0] = i[1] | i[3];
  assign y[1] = i[2] | i[3];
endmodule
```

tb_encoder4to2.v

```
module tb_encoder4to2;
  reg [3:0] i; wire [1:0] y;
  encoder4to2_dataflow uut(.i(i),.y(y));
  initial begin i=4'b0001; #2; i=4'b0010; #2; i=4'b0100; #2; i=4'b1000; #2; $finish; end
endmodule
```

35) 8-to-3 Encoder (priority encoded)

encoder8to3_dataflow.v

```
module encoder8to3_dataflow(input [7:0] i, output reg [2:0] y);
  always @(*) begin
    casex (i)
      8'b1xxxxxxx: y = 3'd7;
      8'b01xxxxxx: y = 3'd6;
      8'b001xxxxx: y = 3'd5;
      8'b0001xxxx: y = 3'd4;
      8'b00001xxx: y = 3'd3;
      8'b000001xx: y = 3'd2;
      8'b0000001x: y = 3'd1;
      8'b00000001: y = 3'd0;
      default: y = 3'd0;
    endcase
  end
```

```
endmodule
```

encoder8to3_behavioral.v

```
module encoder8to3_behavioral(input [7:0] i, output reg [2:0] y);
  always @(*) begin
    if (i[7]) y = 3'd7;
    else if (i[6]) y = 3'd6;
    else if (i[5]) y = 3'd5;
    else if (i[4]) y = 3'd4;
    else if (i[3]) y = 3'd3;
    else if (i[2]) y = 3'd2;
    else if (i[1]) y = 3'd1;
    else y = 3'd0;
  end
endmodule
```

encoder8to3_gatelevel.v

```
module encoder8to3_gatelevel(input [7:0] i, output [2:0] y);
  // gate-level large; use behavioral for priority. Here, direct assign for simplicity:
  reg [2:0] yr;
  always @(*) begin
    if (i[7]) yr = 3'd7;
    else if (i[6]) yr = 3'd6;
    else if (i[5]) yr = 3'd5;
    else if (i[4]) yr = 3'd4;
    else if (i[3]) yr = 3'd3;
    else if (i[2]) yr = 3'd2;
    else if (i[1]) yr = 3'd1;
```

```
   else yr = 3'd0;
  end
  assign y = yr;
endmodule
```

tb_encoder8to3.v

```
module tb_encoder8to3;
  reg [7:0] i; wire [2:0] y;
  encoder8to3_behavioral uut(.i(i),.y(y));
  initial begin i=8'b00000001; #2; i=8'b00010000; #2; i=8'b10000000; #2; $finish; end
endmodule
```

36) Parity Bit Generator

parity_gen_dataflow.v

```
module parity_gen_dataflow(input wire [7:0] d, output wire p);
  assign p = ^d; // XOR-reduction: 1 if odd parity; you can invert for even parity as needed
endmodule
```

parity_gen_behavioral.v

```
module parity_gen_behavioral(input wire [7:0] d, output reg p);
  always @(*) p = ^d;
endmodule
```

parity_gen_gatelevel.v

```
module parity_gen_gatelevel(input [7:0] d, output p);
  assign p = d[0]^d[1]^d[2]^d[3]^d[4]^d[5]^d[6]^d[7];
endmodule
```

tb_parity_gen.v

```
module tb_parity_gen;
  reg [7:0] d; wire p;
  parity_gen_dataflow uut(.d(d),.p(p));
  initial begin
    d = 8'b00000000; #2; $display("%b -> parity=%b", d, p);
    d = 8'b10101010; #2; $display("%b -> parity=%b", d, p);
    d = 8'b11111111; #2; $display("%b -> parity=%b", d, p);
    $finish;
  end
endmodule
```

37) Parity Bit Checker

parity_check_dataflow.v

```
module parity_check_dataflow(input wire [7:0] d, input wire p_in, output wire ok);
  assign ok = ((^d) ^ p_in) == 1'b0 ? 1'b1 : 1'b0; // ok==1 when parity matches expected (even)
endmodule
```

parity_check_behavioral.v

```
module parity_check_behavioral(input [7:0] d, input p_in, output reg ok);
  always @(*) ok = ((^d) ^ p_in) == 1'b0;
endmodule
```

parity_check_gatelevel.v

```
module parity_check_gatelevel(input [7:0] d, input p_in, output ok);
  assign ok = ~((^d) ^ p_in);
endmodule
```

tb_parity_check.v

```
module tb_parity_check;
  reg [7:0] d; reg p; wire ok;
  parity_check_behavioral uut(.d(d),.p_in(p),.ok(ok));
  initial begin
    d=8'b10101010; p=^d; #2; $display("ok=%b",ok);
    d[0]=~d[0]; #2; $display("after flip ok=%b",ok);
    $finish;
  end
endmodule
```

38) Majority Detector (3 inputs)

majority3_dataflow.v

```
module majority3_dataflow(input wire a,b,c, output wire y);
  assign y = (a & b) | (b & c) | (a & c);
endmodule
```

majority3_behavioral.v

```
module majority3_behavioral(input a,b,c, output reg y);
  always @(*) y = (a & b) | (b & c) | (a & c);
endmodule
```

majority3_gatelevel.v

```
module majority3_gatelevel(input a,b,c, output y);
  wire w1,w2,w3;
  and (w1,a,b); and (w2,b,c); and (w3,a,c);
  or  (y,w1,w2,w3);
endmodule
```

tb_majority3.v

```
module tb_majority3;
  reg a,b,c; wire y;
  majority3_dataflow uut(.a(a),.b(b),.c(c),.y(y));
  initial begin
    {a,b,c}=3'b000; #2; {a,b,c}=3'b001; #2; {a,b,c}=3'b011; #2; {a,b,c}=3'b111; #2; $finish;
  end
endmodule
```

39) Even–Odd Detector

evenodd_dataflow.v

```
module evenodd_dataflow(input wire [7:0] d, output wire even, odd);
  assign odd = ^d;
  assign even = ~(^d);
endmodule
```

evenodd_behavioral.v

```
module evenodd_behavioral(input [7:0] d, output reg even, odd);
  always @(*) begin
    odd = ^d;
    even = ~odd;
  end
endmodule
```

evenodd_gatelevel.v

```
module evenodd_gatelevel(input [7:0] d, output even, odd);
  assign odd = d[0]^d[1]^d[2]^d[3]^d[4]^d[5]^d[6]^d[7];
  assign even = ~odd;
endmodule
```

tb_evenodd.v

```
module tb_evenodd;
  reg [7:0] d; wire even, odd;
  evenodd_dataflow uut(.d(d),.even(even),.odd(odd));
  initial begin
    d=8'h00; #2; $display("%b even=%b odd=%b",d,even,odd);
    d=8'hFF; #2; $display("%b even=%b odd=%b",d,even,odd);
    d=8'h0F; #2; $display("%b even=%b odd=%b",d,even,odd);
    $finish;
  end
endmodule
```

40.Parity Bit Checker (similar parry bit generator)

C. SEQUENTIAL CIRCUITS

41. SR Latch
42. D Latch
43. JK Latch
44. T Latch
45. D Flip-Flop (posedge)
46. D Flip-Flop (negedge)
47. JK Flip-Flop
48. T Flip-Flop
49. Master–Slave JK Flip-Flop
50. SR Flip-Flop
51. Level-triggered D Flip-Flop
52. Synchronous Counter (4-bit)

53. Asynchronous Counter (4-bit)

54. Up Counter

55. Down Counter

56. Up/Down Counter

57. Decade Counter

58. Ring Counter

59. Johnson Counter

60. 4-bit Parallel Load Counter

61. 4-bit Universal Shift Register

62. Serial In Serial Out (SISO)

63. Serial In Parallel Out (SIPO)

64. Parallel In Serial Out (PISO)

65. Parallel In Parallel Out (PIPO)

66. Sequence Detector (Moore 1011)

67. Sequence Detector (Mealy 101)

68. 3-bit Sequence Generator

69. State Machine Example (Moore Lamp Controller)

70. Traffic Light Controller FSM

71. Vending Machine Controller FSM

41.SR LATCH

Dataflow

```
module sr_latch_df(input S, R, output Q, Qbar);
  assign Q    = ~(R | Qbar);
  assign Qbar = ~(S | Q);
endmodule
```

Behavioral

```
module sr_latch_bh(input S, R, output reg Q, Qbar);
  always @(*) begin
    case ({S,R})
      2'b00: ; // hold
      2'b01: begin Q=0; Qbar=1; end
      2'b10: begin Q=1; Qbar=0; end
      2'b11: begin Q=1'bx; Qbar=1'bx; end
    endcase
  end
endmodule
```

Gate-Level

```
module sr_latch_gl(input S, R, output Q, Qbar);
  nor n1(Q, R, Qbar);
  nor n2(Qbar, S, Q);
endmodule
```

Testbench

```
module tb_sr_latch;
  reg S, R; wire Q, Qbar;
  sr_latch_df dut(S, R, Q, Qbar);
  initial begin
    {S,R}=2'b00; #10;
    {S,R}=2'b01; #10;
    {S,R}=2'b10; #10;
    {S,R}=2'b11; #10;
    $finish;
```

```
  end
endmodule
```

42. D LATCH

Dataflow

```
module d_latch_df(input D, EN, output Q, Qbar);
  assign Q    = EN ? D : Q;
  assign Qbar = ~Q;
endmodule
```

Behavioral

```
module d_latch_bh(input D, EN, output reg Q, Qbar);
  always @(*) begin
    if (EN) begin Q = D; Qbar = ~D; end
  end
endmodule
```

Gate-Level

```
module d_latch_gl(input D, EN, output Q, Qbar);
  wire S, R;
  assign S = EN & D;
  assign R = EN & ~D;
  nor n1(Q, R, Qbar);
  nor n2(Qbar, S, Q);
endmodule
```

Testbench

```
module tb_d_latch;
  reg D, EN; wire Q, Qbar;
```

```
 d_latch_df dut(D, EN, Q, Qbar);
 initial begin
  EN=0; D=0; #10;
  EN=1; D=1; #10;
  D=0; #10;
  EN=0; D=1; #10;
  $finish;
 end
endmodule
```

43. JK LATCH

Behavioral

```
module jk_latch_bh(input J, K, EN, output reg Q);
 always @(*) begin
  if (EN) begin
    case ({J,K})
      2'b00: Q = Q;
      2'b01: Q = 0;
      2'b10: Q = 1;
      2'b11: Q = ~Q;
    endcase
  end
 end
endmodule
```

Gate-Level

```
module jk_latch_gl(input J, K, EN, output Q, Qbar);
```

```
  wire S, R;
  assign S = EN & J & Qbar;
  assign R = EN & K & Q;
  nor n1(Q, R, Qbar);
  nor n2(Qbar, S, Q);
endmodule
```

Testbench

```
module tb_jk_latch;
  reg J,K,EN; wire Q;
  jk_latch_bh dut(J,K,EN,Q);
  initial begin
    EN=1;
    {J,K}=2'b00; #10;
    {J,K}=2'b01; #10;
    {J,K}=2'b10; #10;
    {J,K}=2'b11; #10;
    $finish;
  end
endmodule
```

44. T LATCH

Behavioral

```
module t_latch_bh(input T, EN, output reg Q);
  always @(*) begin
    if (EN) begin
      if (T) Q = ~Q;
```

```
    else Q = Q;
   end
  end
endmodule
```

Testbench

```
module tb_t_latch;
  reg T, EN; wire Q;
  t_latch_bh dut(T, EN, Q);
  initial begin
    EN=1;
    T=0; #10;
    T=1; #10;
    T=1; #10;
    T=0; #10;
    $finish;
  end
endmodule
```

45.SR FLIP-FLOP

Behavioral

```
module sr_ff_bh(input S, R, clk, output reg Q);
  always @(posedge clk) begin
    case ({S,R})
      2'b00: Q <= Q;
      2'b01: Q <= 0;
      2'b10: Q <= 1;
```

```
    2'b11: Q <= 1'bx;
   endcase
 end
endmodule
```

Testbench

```
module tb_sr_ff;
 reg S,R,clk; wire Q;
 sr_ff_bh dut(S,R,clk,Q);
 always #5 clk=~clk;
 initial begin
  clk=0; {S,R}=2'b00;
  #10 {S,R}=2'b10;
  #10 {S,R}=2'b01;
  #10 {S,R}=2'b11;
  #10 $finish;
 end
endmodule
```

46.D FLIP-FLOP

Behavioral

```
module d_ff_bh(input D, clk, output reg Q);
 always @(posedge clk)
  Q <= D;
endmodule
```

Testbench

```
module tb_d_ff;
```

```
 reg D,clk; wire Q;
 d_ff_bh dut(D,clk,Q);
 always #5 clk=~clk;
 initial begin
  clk=0; D=0;
  #10 D=1;
  #10 D=0;
  #10 $finish;
 end
endmodule
```

47.JK FLIP-FLOP

Behavioral

```
module jk_ff_bh(input J,K,clk,output reg Q);
 always @(posedge clk)
  case({J,K})
   2'b00: Q <= Q;
   2'b01: Q <= 0;
   2'b10: Q <= 1;
   2'b11: Q <= ~Q;
  endcase
endmodule
```

Testbench

```
module tb_jk_ff;
 reg J,K,clk; wire Q;
 jk_ff_bh dut(J,K,clk,Q);
```

```
 always #5 clk=~clk;
 initial begin
  clk=0;
  {J,K}=2'b00; #10;
  {J,K}=2'b01; #10;
  {J,K}=2'b10; #10;
  {J,K}=2'b11; #10;
  $finish;
 end
endmodule
```

48. T FLIP-FLOP

Behavioral

```
module t_ff_bh(input T, clk, output reg Q);
 always @(posedge clk)
  if (T) Q <= ~Q;
  else Q <= Q;
endmodule
```

Testbench

```
module tb_t_ff;
 reg T,clk; wire Q;
 t_ff_bh dut(T,clk,Q);
 always #5 clk=~clk;
 initial begin
  clk=0; T=0;
  #10 T=1;
```

```
    #10 T=1;
    #10 T=0;
    #10 $finish;
  end
endmodule
```

49.MASTER-SLAVE JK FLIP-FLOP

Behavioral

```
module ms_jk_ff(input J,K,clk,output reg Q);
  reg Qm;
  always @(posedge clk)
    case({J,K})
      2'b00: Qm <= Qm;
      2'b01: Qm <= 0;
      2'b10: Qm <= 1;
      2'b11: Qm <= ~Qm;
    endcase
  always @(negedge clk)
    Q <= Qm;
endmodule
```

Testbench

```
module tb_ms_jk_ff;
  reg J,K,clk; wire Q;
  ms_jk_ff dut(J,K,clk,Q);
  always #5 clk=~clk;
  initial begin
```

```
  clk=0;
  {J,K}=2'b00; #10;
  {J,K}=2'b10; #10;
  {J,K}=2'b01; #10;
  {J,K}=2'b11; #10;
  $finish;
 end
endmodule
```

50. EDGE-TRIGGERED D FLIP-FLOP WITH RESET

Behavioral

```
module d_ff_rst(input D, clk, rst, output reg Q);
 always @(posedge clk or posedge rst)
  if (rst) Q <= 0;
  else Q <= D;
endmodule
```

Testbench

```
module tb_d_ff_rst;
 reg D,clk,rst; wire Q;
 d_ff_rst dut(D,clk,rst,Q);
 always #5 clk=~clk;
 initial begin
  clk=0; rst=1; D=0; #10;
  rst=0; D=1; #10;
  D=0; #10;
  rst=1; #10;
```

```
    $finish;
  end
endmodule
```

51. LEVEL-TRIGGERED D FLIP-FLOP

Behavioral Code

```
module d_ff_level(input D, clk, output reg Q);
  always @(clk or D)
    if (clk) Q = D;
endmodule
```

Testbench

```
module tb_d_ff_level;
  reg D, clk; wire Q;
  d_ff_level dut(D, clk, Q);
  initial begin
    clk=0; D=0;
    #5  D=1;
    #5  clk=1; #5 clk=0;
    #5  D=0;
    #5  clk=1; #5 clk=0;
    #10 $finish;
  end
endmodule
```

52. SYNCHRONOUS COUNTER (4-bit)

Behavioral Code

```
module sync_counter(input clk, rst, output reg [3:0] count);
```

```
  always @(posedge clk or posedge rst)
    if (rst)
      count <= 4'b0000;
    else
      count <= count + 1;
endmodule
```

Testbench

```
module tb_sync_counter;
  reg clk, rst; wire [3:0] count;
  sync_counter dut(clk, rst, count);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; #10;
    rst=0; #100;
    $finish;
  end
endmodule
```

53.ASYNCHRONOUS COUNTER (4-bit RIPPLE)

Behavioral Code

```
module async_counter(input clk, rst, output [3:0] q);
  t_ff_bh t0(.T(1'b1), .clk(clk), .Q(q[0]));
  t_ff_bh t1(.T(1'b1), .clk(q[0]), .Q(q[1]));
  t_ff_bh t2(.T(1'b1), .clk(q[1]), .Q(q[2]));
  t_ff_bh t3(.T(1'b1), .clk(q[2]), .Q(q[3]));
endmodule
```

Testbench

```
module tb_async_counter;
  reg clk, rst; wire [3:0] q;
  async_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
    clk=0;
    #100 $finish;
  end
endmodule
```

54.UP COUNTER (4-bit)

Behavioral Code

```
module up_counter(input clk, rst, output reg [3:0] q);
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 0;
    else
      q <= q + 1;
endmodule
```

Testbench

```
module tb_up_counter;
  reg clk, rst; wire [3:0] q;
  up_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
```

```
    clk=0; rst=1; #10;
    rst=0; #100;
    $finish;
  end
endmodule
```

55.DOWN COUNTER (4-bit)

Behavioral Code

```
module down_counter(input clk, rst, output reg [3:0] q);
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b1111;
    else
      q <= q - 1;
endmodule
```

Testbench

```
module tb_down_counter;
  reg clk, rst; wire [3:0] q;
  down_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; #10;
    rst=0; #100;
    $finish;
  end
endmodule
```

56. UP/DOWN COUNTER (4-bit)

Behavioral Code

```
module updown_counter(input clk, rst, mode, output reg [3:0] q);
  // mode = 1 for up, 0 for down
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b0000;
    else if (mode)
      q <= q + 1;
    else
      q <= q - 1;
endmodule
```

Testbench

```
module tb_updown_counter;
  reg clk, rst, mode; wire [3:0] q;
  updown_counter dut(clk, rst, mode, q);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; mode=1; #10;
    rst=0; #50;
    mode=0; #50;
    $finish;
  end
endmodule
```

57.DECADE COUNTER (MOD-10)

Behavioral Code

```
module decade_counter(input clk, rst, output reg [3:0] q);
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b0000;
    else if (q == 4'b1001)
      q <= 4'b0000;
    else
      q <= q + 1;
endmodule
```

Testbench

```
module tb_decade_counter;
  reg clk, rst; wire [3:0] q;
  decade_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; #10;
    rst=0; #200;
    $finish;
  end
endmodule
```

58. RING COUNTER (4-bit)

Behavioral Code

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

```
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b0001;
    else
      q <= {q[2:0], q[3]};
endmodule
```

Testbench

```
module tb_ring_counter;
  reg clk, rst; wire [3:0] q;
  ring_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; #10;
    rst=0; #100;
    $finish;
  end
endmodule
```

59: JOHNSON COUNTER (4-bit)

Behavioral Code

```
module johnson_counter(input clk, rst, output reg [3:0] q);
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b0000;
    else
      q <= {~q[0], q[3:1]};
```

```
endmodule
```

Testbench

```
module tb_johnson_counter;
  reg clk, rst; wire [3:0] q;
  johnson_counter dut(clk, rst, q);
  always #5 clk = ~clk;
  initial begin
    clk=0; rst=1; #10;
    rst=0; #100;
    $finish;
  end
endmodule
```

60. 4-BIT PARALLEL LOAD COUNTER

Behavioral Code

```
module parallel_load_counter(input clk, rst, load, input [3:0] data, output reg [3:0] q);
  always @(posedge clk or posedge rst)
    if (rst)
      q <= 4'b0000;
    else if (load)
      q <= data;
    else
      q <= q + 1;
endmodule
```

Testbench

```
module tb_parallel_load_counter;
```

```
 reg clk, rst, load;
 reg [3:0] data;
 wire [3:0] q;
 parallel_load_counter dut(clk, rst, load, data, q);
 always #5 clk = ~clk;
 initial begin
  clk=0; rst=1; load=0; data=4'b1010; #10;
  rst=0; #20;
  load=1; #10;
  load=0; #50;
  $finish;
 end
endmodule
```

61) 4-bit Universal Shift Register (modes: hold/shift-right/shift-left/parallel-load)

Dataflow

```
// universal_shift_reg_df.v
module universal_shift_reg_df(
 input  wire       clk,
 input  wire       rst,
 input  wire [1:0]  mode,     // 00=hold, 01=shift_right, 10=shift_left, 11=parallel_load
 input  wire       sin_left,
 input  wire       sin_right,
 input  wire [3:0]  parallel_in,
 output wire [3:0]  q
);
```

```
  wire [3:0] q_next;
  assign q_next = (mode==2'b11) ? parallel_in :
                  (mode==2'b01) ? {sin_left, q[3:1]} :
                  (mode==2'b10) ? {q[2:0], sin_right} :
                  q;
  // registers
  reg [3:0] qr;
  always @(posedge clk or posedge rst) begin
    if (rst) qr <= 4'b0000; else qr <= q_next;
  end
  assign q = qr;
endmodule
```

Behavioral

```
// universal_shift_reg_bh.v
module universal_shift_reg_bh(
  input clk, rst,
  input [1:0] mode,
  input sin_left, sin_right,
  input [3:0] parallel_in,
  output reg [3:0] q
);
  always @(posedge clk or posedge rst) begin
    if (rst) q <= 4'b0000;
    else begin
      case(mode)
        2'b00: q <= q;                  // hold
```

```
      2'b01: q <= {sin_left, q[3:1]};     // shift right
      2'b10: q <= {q[2:0], sin_right};    // shift left
      2'b11: q <= parallel_in;           // load
      default: q <= q;
    endcase
  end
 end
endmodule
```

Gate-level / Structural (use D-FFs + muxes)

```
// universal_shift_reg_gl.v
module dff(input d, input clk, output reg q);
 always @(posedge clk) q <= d;
endmodule

module mux4to1(input a, input b, input c, input d, input [1:0] sel, output y);
 assign y = (sel==2'b00)?a:
          (sel==2'b01)?b:
          (sel==2'b10)?c:d;
ndmodule

module universal_shift_reg_gl(
 input clk, rst, sin_left, sin_right,
 input [1:0] mode,
 input [3:0] parallel_in,
 output [3:0] q
);
```

```
  wire d0, d1, d2, d3;
  // q[3] next choices: parallel_in[3], shift_right: sin_left, shift_left q[2], hold q[3]
  mux4to1 m3(parallel_in[3], sin_left, /*shift_left*/ /*will connect*/ 1'bx, /*hold*/ 1'bx, mode,
d3);
  reg [3:0] qr;
  always @(posedge clk) begin
   if (rst) qr <= 4'b0;
   else begin
     case(mode)
       2'b00: qr <= qr;
       2'b01: qr <= {sin_left, qr[3:1]};
       2'b10: qr <= {qr[2:0], sin_right};
       2'b11: qr <= parallel_in;
     endcase
   end
  end
  assign q = qr;
endmodule
```

Testbench

```
// tb_universal_shift_reg.v
module tb_universal_shift_reg;
  reg clk,rst; reg [1:0] mode; reg sin_left, sin_right; reg [3:0] p;
  wire [3:0] q;
  universal_shift_reg_bh uut(clk,rst,mode,sin_left,sin_right,p,q);
  initial clk=0; always #5 clk = ~clk;
  initial begin
```

```
    rst=1; mode=2'b00; p=4'b1010; sin_left=1'b0; sin_right=1'b1; #12;
    rst=0;
    // load
    mode=2'b11; #10;
    // shift right 2 cycles
    mode=2'b01; sin_left=1; #20;
    // shift left 2 cycles
    mode=2'b10; sin_right=0; #20;
    $finish;
  end
endmodule
```

62) Serial-In Serial-Out (SISO) Register (shift-right)

Dataflow

```
// siso_df.v
module siso_df(input clk, rst, sin, load, output [3:0] q);
  reg [3:0] qr;
  always @(posedge clk or posedge rst) begin
    if (rst) qr <= 4'b0;
    else if (load) qr <= {qr[2:0], sin};
    else qr <= qr;
  end
  assign q = qr;
endmodule
```

Behavioral

```
// siso_bh.v
```

```
module siso_bh(input clk, rst, sin, load, output reg [3:0] q);
  always @(posedge clk or posedge rst) begin
    if (rst) q <= 4'b0000;
    else if (load) q <= {q[2:0], sin};
  end
endmodule
```

Gate-level

```
// siso_gl.v (structural DFF chain)
module siso_gl(input clk, rst, sin, load, output [3:0] q);
  reg [3:0] qr;
  always @(posedge clk) begin
    if (rst) qr <= 4'b0;
    else if (load) qr <= {qr[2:0], sin};
  end
  assign q = qr;
endmodule
```

Testbench

```
// tb_siso.v
module tb_siso;
  reg clk,rst, sin, load; wire [3:0] q;
  siso_bh uut(clk,rst,sin,load,q);
  initial clk=0; always #5 clk=~clk;
  initial begin
    rst=1; load=0; sin=1; #12; rst=0;
    load=1; sin=1; #10; sin=0; #30; load=0; #10;
    $finish;
```

```
  end
endmodule
```

63) Serial-In Parallel-Out (SIPO)

Dataflow

```
// sipo_df.v
module sipo_df(input clk, rst, sin, load, output reg [3:0] q);
  always @(posedge clk or posedge rst) begin
    if (rst) q <= 4'b0;
    else if (load) q <= {q[2:0], sin};
  end
endmodule
```

Behavioral

```
// sipo_bh.v
module sipo_bh(input clk, rst, sin, load, output reg [3:0] q);
  always @(posedge clk or posedge rst) begin
    if (rst) q <= 4'b0;
    else if (load) q <= {q[2:0], sin};
  end
endmodule
```

Gate-level

```
// sipo_gl.v
module sipo_gl(input clk, rst, sin, load, output [3:0] q);
  reg [3:0] qr;
  always @(posedge clk) begin
    if (rst) qr <= 4'b0;
```

```
  else if (load) qr <= {qr[2:0], sin};
 end
 assign q = qr;
endmodule
```

Testbench

```
// tb_sipo.v
module tb_sipo;
 reg clk,rst, sin, load; wire [3:0] q;
 sipo_bh uut(clk,rst,sin,load,q);
 initial clk=0; always #5 clk=~clk;
 initial begin
  rst=1; load=0; #12; rst=0;
  load=1; sin=1; #10; sin=0; #10; sin=1; #10; sin=1; #10;
  load=0; #20; $finish;
 end
endmodule
```

64) Parallel-In Serial-Out (PISO)

Dataflow

```
// piso_df.v
module piso_df(input clk, rst, load, shift, input [3:0] pdata, output reg qout);
 reg [3:0] shiftreg;
 always @(posedge clk or posedge rst) begin
  if (rst) shiftreg <= 4'b0;
  else if (load) shiftreg <= pdata;
  else if (shift) begin qout <= shiftreg[3]; shiftreg <= {shiftreg[2:0],1'b0}; end
```

```
  end
endmodule
```

Behavioral

```
// piso_bh.v
module piso_bh(input clk, rst, load, shift, input [3:0] pdata, output reg qout);
  reg [3:0] sr;
  always @(posedge clk or posedge rst) begin
    if (rst) begin sr<=0; qout<=0; end
    else if (load) sr <= pdata;
    else if (shift) begin qout <= sr[3]; sr <= {sr[2:0], 1'b0}; end
  end
endmodule
```

Gate-level

```
// piso_gl.v
module piso_gl(input clk, rst, load, shift, input [3:0] pdata, output qout);
  reg [3:0] sr;
  always @(posedge clk) begin
    if (rst) sr <= 4'b0;
    else if (load) sr <= pdata;
    else if (shift) sr <= {sr[2:0],1'b0};
  end
  assign qout = sr[3];
endmodule
```

Testbench

```
// tb_piso.v
module tb_piso;
```

```
 reg clk,rst,load,shift; reg [3:0] pdata; wire qout;
 piso_bh uut(clk,rst,load,shift,pdata,qout);
 initial clk=0; always #5 clk=~clk;
 initial begin
  rst=1; load=0; pdata=4'b1011; #12; rst=0;
  load=1; #10; load=0;
  shift=1; #40; shift=0; $finish;
 end
endmodule
```

65) Parallel-In Parallel-Out (PIPO)

Dataflow / Behavioral

```
// pipo.v
module pipo(input clk, rst, load, input [3:0] pdata, output reg [3:0] q);
 always @(posedge clk or posedge rst) begin
  if (rst) q <= 4'b0;
  else if (load) q <= pdata;
 end
endmodule
```

Gate-level

```
// pipo_gl.v
module pipo_gl(input clk, rst, load, input [3:0] pdata, output [3:0] q);
 reg [3:0] qr;
 always @(posedge clk) begin
  if (rst) qr <= 0;
  else if (load) qr <= pdata;
```

```
  end
  assign q = qr;
endmodule
```

Testbench

```
// tb_pipo.v
module tb_pipo;
  reg clk,rst,load; reg [3:0] pdata; wire [3:0] q;
  pipo uut(clk,rst,load,pdata,q);
  initial clk=0; always #5 clk=~clk;
  initial begin
    rst=1; load=0; pdata=4'b1100; #12; rst=0;
    load=1; #10; load=0; #10; $finish;
  end
endmodule
```

66) Sequence Detector (Moore) — detects "1011"

Behavioral / Dataflow

```
// seqdet_moore_1011.v
module seqdet_moore_1011(input clk, rst, in, output reg detect);
  typedef enum reg [2:0] {S0,S1,S2,S3,S4} state_t;
  reg [2:0] state, next;
  // state transitions
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
```

```
  always @(*) begin
    next = state; detect = 0;
    case (state)
      S0: if (in) next = S1;
      S1: if (!in) next = S2; else next = S1;
      S2: if (in) next = S3; else next = S0;
      S3: if (in) begin next = S4; detect = 1; end else next = S2;
      S4: begin detect = 1; if (in) next = S1; else next = S2; end
    endcase
  end
endmodule
```

Gate-level: complex FSM — use behavioral for clarity.

Testbench

```
// tb_seqdet_moore_1011.v
module tb_seqdet_moore_1011;
  reg clk,rst, in; wire detect;
  seqdet_moore_1011 uut(clk,rst,in,detect);
  initial clk=0; always #5 clk=~clk;
  initial begin
    rst=1; in=0; #12; rst=0;
    // apply stream: 1 0 1 1 -> should detect
    {in}=1; #10; {in}=0; #10; {in}=1; #10; {in}=1; #10;
    $display("detect=%b", detect);
    $finish;
  end
endmodule
```

67) Sequence Detector (Mealy) — detects "101" with Mealy output

Behavioral

```
// seqdet_mealy_101.v
module seqdet_mealy_101(input clk, rst, in, output reg out);
  reg [1:0] state, next;
  localparam S0=2'b00, S1=2'b01, S2=2'b10;
  always @(posedge clk or posedge rst) if (rst) state<=S0; else state<=next;
  always @(*) begin
    next = state; out = 0;
    case(state)
      S0: if (in) next = S1;
      S1: if (!in) next = S2; else next = S1;
      S2: if (in) begin out = 1; next = S1; end else next=S0;
    endcase
  end
endmodule
```

Testbench

```
// tb_seqdet_mealy_101.v
module tb_seqdet_mealy_101;
  reg clk,rst,in; wire out;
  seqdet_mealy_101 uut(clk,rst,in,out);
  initial clk=0; always #5 clk=~clk;
  initial begin
    rst=1; in=0; #12; rst=0;
    in=1; #10; in=0; #10; in=1; #10; // pattern 101 -> out asserted
```

```
  #10 $finish;
 end
endmodule
```

68) 3-bit Sequence Generator (simple LFSR-based pseudo-random generator)

Behavioral / Dataflow

```
// seqgen_3bit_lfsr.v
module seqgen_3bit_lfsr(input clk, rst, output reg [2:0] q);
 wire fb;
 assign fb = q[2] ^ q[0]; // taps for max-length
 always @(posedge clk or posedge rst) begin
  if (rst) q <= 3'b001;
  else q <= {q[1:0], fb};
 end
endmodule
```

Testbench

```
// tb_seqgen_3bit_lfsr.v

module tb_seqgen_3bit_lfsr;
 reg clk,rst; wire [2:0] q;
 seqgen_3bit_lfsr uut(clk,rst,q);
 initial clk=0; always #5 clk=~clk;
 initial begin rst=1; #12; rst=0; #100; $finish; end
endmodule
```

69) Moore Lamp Controller (example FSM) — 3-state lamp: OFF -> ON -> BLINK

Behavioral

```
// moore_lamp_controller.v
module moore_lamp_controller(input clk, rst, sw_on, sw_off, output reg lamp);
 typedef enum reg [1:0] {OFF, ON, BLINK} state_t;
 reg [1:0] state, next;
 reg blink;
 always @(posedge clk or posedge rst) begin
  if (rst) state <= OFF; else state <= next;
 end
 always @(*) begin
  next = state; lamp = 0;
  case(state)
   OFF: if (sw_on) next = ON;
   ON: begin lamp = 1; if (sw_off) next = BLINK; end
   BLINK: begin blink = ~blink; lamp = blink; if (sw_off) next = OFF; end
  endcase
 end
endmodule
```

Testbench

```
// tb_moore_lamp_controller.v
module tb_moore_lamp_controller;
 reg clk,rst, sw_on, sw_off; wire lamp;
 moore_lamp_controller uut(clk,rst,sw_on,sw_off,lamp);
 initial clk=0; always #5 clk=~clk;
 initial begin
  rst=1; sw_on=0; sw_off=0; #12; rst=0;
```

```
  sw_on=1; #10; sw_on=0; #20;
  sw_off=1; #10; sw_off=0; #40;
  $finish;
 end
endmodule
```

70) Traffic Light Controller FSM (simple 3-phase: NS_green -> NS_yellow -> EW_green -> EW_yellow)

Behavioral

```
// traffic_light_fsm.v
module traffic_light_fsm(input clk, rst, output reg [2:0] ns, output reg [2:0] ew);
 typedef enum reg [1:0] {S_NS_GREEN, S_NS_YELLOW, S_EW_GREEN, S_EW_YELLOW} state_t;
 reg [1:0] state, next;
 reg [7:0] timer;
 parameter NSG_TIME = 8'd20, NSY_TIME = 8'd5, EWG_TIME = 8'd20, EWY_TIME = 8'd5;
 always @(posedge clk or posedge rst) begin
  if (rst) begin state <= S_NS_GREEN; timer <= 0; end
  else begin
   if (timer==0) begin state <= next; timer <= (state==S_NS_GREEN)?NSG_TIME:
                                (state==S_NS_YELLOW)?NSY_TIME:
                                (state==S_EW_GREEN)?EWG_TIME:EWY_TIME;
    end else timer <= timer - 1;
  end
 end
 always @(*) begin
  next = state;
```

```
   case(state)
     S_NS_GREEN: begin ns=3'b100; ew=3'b001; if (timer==0) next = S_NS_YELLOW; end
     S_NS_YELLOW: begin ns=3'b010; ew=3'b001; if (timer==0) next = S_EW_GREEN; end
     S_EW_GREEN: begin ns=3'b001; ew=3'b100; if (timer==0) next = S_EW_YELLOW; end
     S_EW_YELLOW: begin ns=3'b001; ew=3'b010; if (timer==0) next = S_NS_GREEN; end
   endcase
  end
endmodule
```

Testbench

```
// tb_traffic_light_fsm.v
module tb_traffic_light_fsm;
  reg clk,rst; wire [2:0] ns, ew;
  traffic_light_fsm uut(clk,rst,ns,ew);
  initial clk=0; always #1 clk=~clk;  // faster sim clock
  initial begin rst=1; #3; rst=0;
    #400 $finish;
  end
endmodule
```

71) Vending Machine Controller FSM (simple example)

Accepts coins of 1 unit, target price = 3. Dispenses when sum >=3.

Behavioral

```
// vending_machine_fsm.v
module vending_machine_fsm(input clk, rst, coin, output reg dispense, output reg [1:0]
state_out);
  // state = 0,1,2,3+ (money units)
```

```
 reg [1:0] state, next;
 always @(posedge clk or posedge rst) begin
  if (rst) state <= 2'd0;
  else state <= next;
 end
 always @(*) begin
  next = state; dispense = 0;
  case(state)
   2'd0: if (coin) next = 2'd1;
   2'd1: if (coin) next = 2'd2;
   2'd2: if (coin) begin next = 2'd0; dispense = 1; end
   default: next = 2'd0;
  endcase
  state_out = state;
 end
endmodule
```

Testbench

```
// tb_vending_machine_fsm.v
module tb_vending_machine_fsm;
 reg clk, rst, coin; wire dispense; wire [1:0] state_out;
 vending_machine_fsm uut(clk,rst,coin,dispense,state_out);
 initial clk=0; always #5 clk=~clk;
 initial begin
  rst=1; coin=0; #12; rst=0;
  coin=1; #10; coin=0; #10; // 1 unit
  coin=1; #10; coin=0; #10; // 2 units
```

```
    coin=1; #10; coin=0; #10; // 3 -> should dispense
    #20 $finish;
  end
endmodule
```