# 50 PROGRAMS-
# VERILOG MODULE


## ARUNACHALA COLLEGE OF ENGINEERING FOR WOMEN

VAISHNAVI S M

ECE

# VERILOG

Verilog is a Hardware Description Language (HDL) used to design and model digital circuits. Unlike regular programming languages, Verilog describes how hardware (like logic gates, multiplexers, registers, and processors) should function. It allows engineers to:

Design digital circuits (e.g., adders, multiplexers, processors)

Simulate circuits before manufacturing them

Implement hardware on FPGAs or ASICs

**Verilog allows circuit design using three different modeling styles**:

Structural Modeling (Gate-Level)

Dataflow Modeling (Equation-Based)

Behavioral Modeling (High-Level Code)

**Structural Modeling** = Like physically wiring gates in hardware.
**Dataflow Modeling** = Like writing equations for circuits.
**Behavioral Modeling** = Like coding circuit behavior with if-else and loops.

## AND GATE BEHAVIOR

```
module or_gate (
    input A, B,
    output reg Y
);
    always @(A or B) begin
        Y = A&&B;
    end
endmodule
```

## STRUCTURAL

```
module and_gate_structural
(input A, input B, output Y);

    and (Y, A, B);

endmodule
```

## OR GATE BEHAVIOUR

```
module or_gate (
    input A, B,
    output reg Y
);
    always @(A or B) begin
        Y = A | B;
    end
endmodule
```

## STRUCTURAL

```
module OR_Gate_Structural
(input A, input B, output Y);
 or (Y, A, B);
endmodule
```

## NOT GATE

## BEHAVIOUR

```
module NOT_gate (
    input A,
    output reg Y
);
    always @(A ) begin
        Y =~A;
    end
endmodule
```

## STRUCTURAL

```
module NOT_GATE_Structural
(input A, output Y);
 not (Y, A);
endmodule
```

## NAND GATE BEHAVIOUR

```
module NOT_gate (
    input A,B,
    output reg Y
);
    always @(A or B) begin
        Y =~(A & B);
    end
endmodule
```

## STRUCTURAL

```
module NAND_GATE_Structural (input A,inputB output Y);
 nand (Y, A, B);
endmodule
```

## NOR GATE BEHAVIOUR

```
module NOR_gate (
    input A,B,
    output reg Y
);
    always @(A or B) begin
        Y =~(A|B);
    end
endmodule
```

## STRUCTURAL

```
module NOR_GATE_Structural
(input A,input B output Y);
 nor (Y, A, B);
endmodule
```

## XOR GATE BEHAVIOUR

```
module XOR_gate (
    input A,B,
    output reg Y
);
    always @(A or B) begin
        Y =(A^B);
    end
endmodule
```

## STRUCTURAL

```
module XOR_GATE_Structural
(input A,input B output Y);
 xor (Y, A, B);
endmodule
```

## XNOR GATE BEHAVIOUR

```
module XNOR_gate (
    input A,B,
    output reg Y
);
    always @(A or B) begin
        Y =^(A~B);
    end
endmodule
```

## STRUCTURAL

```
module XNOR_GATE_Structural (input A,input B output Y);
 xnor (Y, A, B);
endmodule
```

## HALF ADDER

## BEHAVIOUR

```
module half_adder_behavioral (
    input  A, B,
    output reg sum, carry
);
    always @(A or B) begin
        sum = A ^ B;
        carry = A & B;
    end
endmodule
```

## DATA FLOW

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

## STRUCTURAL

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

## FULL ADDER

## BEHAVIOUR

```
module full_adder_behavioral (
    input  A, B, Cin,
    output reg Sum, Cout
);
    always @(A or B or Cin) begin
        Sum  = A ^ B ^ Cin;
        Cout = (A & B) | (B & Cin) |
(A & Cin);
    end
endmodule
```

## DATA FLOW

```
module full_adder_dataflow (
    input  A, B, Cin,
    output wire Sum, Cout
);
    assign Sum  = A ^ B ^ Cin;
    assign Cout = (A & B) | (B &
Cin) | (A & Cin);
endmodule
```

## STRUCTURAL

```
module full_adder_structural (
    input A, B, Cin,
    output Sum, Cout
);
    xor (Sum, A, B, Cin);
    or (Cout, (A & B), (B & Cin),
(A & Cin));
endmodule
```

## SUBTRACTOR

```
module subtractor(a, b, diff,
borrow);
    input a, b;
    output diff, borrow;
    reg y;

    always @ (a or b)
        {borrow, diff} = a - b;
   endmodule
```

## COMPARATOR

```
module comparator(a, b, greater,
equal, less);
    input [3:0] a, b;
    output reg greater, equal, less;

    always @(*) begin
        if (a > b) begin
            greater = 1;
            equal = 0;
            less = 0;
        end else if (a == b) begin
            greater = 0;
            equal = 1;
            less = 0;
        end else begin
            greater = 0;
            equal = 0;
            less = 1;
        end
    end

endmodule
```

## PARITY ENCODER

```
module encoder_4to2(input [3:0] data, output reg [1:0] encoded);


    always @(data) begin
        case (data)
            4'b1000: encoded = 2'b11;
            4'b0100: encoded = 2'b10;
            4'b0010: encoded = 2'b01;
            4'b0001: encoded = 2'b00;
            default: encoded = 2'bxx;
        endcase
    end
    endmodule
```

## ARRAY MULTIPLIER

```
module array_multiplier (
    input  [3:0] A, B,
    output [7:0] P
);
    assign P = A * B;
endmodule
```

## T FLIPFLOP

```
module T_flipflop ( input T, clk,
output reg Q
);
    always @(posedge clk) begin
        if (T == 1)
            Q <= ~Q;
        else
            Q <= Q;
    end
endmodule
```

## JK FLIPFLOP

```
module jk_flipflop (
    input clk,
    input j,
    input k,
    input reset,
    output reg q,
    output reg qbar
);


always @(posedge clk or posedge reset) begin
```

```
    if (reset) begin
        q <= 0;
        qbar <= 1;
    end else begin
        case ({j, k})
            2'b00: q <= q;
            2'b01: q <= 0;
            2'b10: q <= 1;
            2'b11: q <= ~q;
        endcase
        qbar <= ~q;
    end
end
endmodule
```

## SHIFT REGISTERS

```
module shift_register (
    input clk,
    input reset,
    input d,
    output reg [3:0] q
);

always @(posedge clk or posedge reset) begin
    if (reset)
        q <= 4'b0000;
    else
        q <= {q[2:0], d};
end


endmodule
```

## UP COUNTER

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

always @(posedge clk or posedge reset) begin
    if (reset)
        q <= 4'b0000;
    else
        q <= q + 1;
end
```

```
endmodule
```

## UP DOWN COUNTER

```
module updown_counter (
    input clk,
    input reset,
    input up_down,
    output reg [3:0] q
);

always @(posedge clk or
posedge reset) begin
    if (reset)
        q <= 4'b0000;
    else if (up_down)
        q <= q + 1;
    else
        q <= q - 1;
end

endmodule
```

## D LATCH

## BEHAVIOUR MODEL

```
module d_latch (
    input d,
    input enable,
    output reg q,
    output qbar
);
assign qbar = ~q;

always @(*) begin
    if (enable)
        q <= d;
end
endmodule
```

## STRUCTURE MODEL

```
module d_latch_structural (
    input d, clk,
    output q, qbar
);
wire dbar, r1, r2;
not (dbar, d);
nand (r1, d, clk);
nand (r2, dbar, clk);
nand (q, r1, qbar);
nand (qbar, r2, q);
endmodule
```

## D FLIPFLOP

## BEHAVIOUR MODEL

```
module d_flipflop (
    input clk,
    input reset,
    input d,
    output reg q,
    output qbar
);

assign qbar = ~q;

always @(posedge clk or
posedge reset) begin
    if (reset)
        q <= 0;
    else
        q <= d;
end
endmodule
```

## STRUCTURAL MODEL

```
module d_flipflop_structural (
    input d, clk, reset,
    output q, qbar
);

wire dbar, r1, r2, r3, r4, s1, s2;
not (dbar, d);
nand (r1, d, clk);
nand (r2, dbar, clk);
nand (s1, r1, s2);
nand (s2, r2, s1);

nand (r3, s1, reset);
nand (r4, s2, reset);
nand (q, r3, qbar);
nand (qbar, r4, q);
endmodule
```

## SR FLIPFLOP

```
module sr_flipflop (
    input clk,
    input reset,
    input s,
    input r,
    output reg q,
    output qbar
);
```

```
assign qbar = ~q;

always @(posedge clk or
posedge reset) begin
    if (reset)
        q <= 0;
    else begin
        case ({s, r})
            2'b00: q <= q;
            2'b01: q <= 0;
            2'b10: q <= 1;
            2'b11: q <= 1'bx;
        endcase
    end
end
endmodule
```

## MULTIPLIER

```
module multiplier (
    input [3:0] a,
    input [3:0] b,
    output [7:0] product
);
assign product = a * b;

endmodule
```

## 1:2 DEMULTIPLEXER

```
module demux_1to2 (
    input din,
    input sel,
    output reg d0,
    output reg d1
);

always @(din, sel) begin
    if (sel == 0) begin
        d0 = din;
        d1 = 0;
    end else begin
        d0 = 0;
        d1 = din;
    end
end

endmodule
```

## DEMULTIPLEXER

```
module demux_1to4 (
    input din,
    input [1:0] sel,
    output reg [3:0] dout
);

always @(din or sel) begin
    dout = 4'b0000;
    case (sel)
        2'b00: dout[0] = din;
        2'b01: dout[1] = din;
        2'b10: dout[2] = din;
        2'b11: dout[3] = din;
    endcase
end

endmodule
```

## 4:1 MULTIPLEXER

### BEHAVIORAL MODEL (USING CASE STATEMENT)

```
module mux_4to1_behavioral (
    input [3:0] din,
    input [1:0] sel,
    output reg dout
);

always @(din or sel) begin
    case (sel)
        2'b00: dout = din[0];
        2'b01: dout = din[1];
        2'b10: dout = din[2];
        2'b11: dout = din[3];
    endcase
end
endmodule
```

### BEHAVIORAL MODEL (USING IF-ELSE STATEMENT)

```
module mux_4to1_if_else (
    input [3:0] din,
    input [1:0] sel,
    output reg dout
);

always @(din or sel) begin
    if (sel == 2'b00)
```

```
        dout = din[0];
    else if (sel == 2'b01)
        dout = din[1];
    else if (sel == 2'b10)
        dout = din[2];
    else
        dout = din[3];
end

endmodule
```

## DATAFLOW MODEL (USING ASSIGN STATEMENT)

```
module mux_4to1_dataflow (
    input [3:0] din,
    input [1:0] sel,
    output dout
);

assign dout = (sel == 2'b00 & din[0]) |
              (sel == 2'b01 & din[1]) |
              (sel == 2'b10 & din[2]) |
              (sel == 2'b11 & din[3]);

endmodule
```

## STRUCTURAL MODELING

```
module mux_4to1_structural (
    input [3:0] din,
    input [1:0] sel,
    output dout
);

wire s0, s1, y0, y1, y2, y3;

not (s0, sel[0]);
not (s1, sel[1]);

and (y0, din[0], s1, s0);
and (y1, din[1], s1, sel[0]);
and (y2, din[2], sel[1], s0);
and (y3, din[3], sel[1], sel[0]);

or (dout, y0, y1, y2, y3);

endmodule
```

## 2:1 MULTIPLEXER

## BEHAVIORAL MODEL (USING CASE)

```
module mux_2to1_case (
    input d0, d1,
    input sel,
    output reg dout
);

always @(d0 or d1 or sel) begin
    case (sel)
        1'b0: dout = d0;
        1'b1: dout = d1;
    endcase
end
endmodule
```

## BEHAVIORAL MODEL (USING IF-ELSE)

```
module mux_2to1_if_else (
    input d0, d1,
    input sel,
    output reg dout
);

always @(d0 or d1 or sel) begin
    if (sel == 0)
        dout = d0;
    else
        dout = d1;
end
endmodule
```

## DATAFLOW MODEL

```
module mux_2to1_dataflow (
    input d0, d1,
    input sel,
    output dout
);
assign dout = (sel & d1) | (~sel & d0);
endmodule
```

## STRUCTURAL MODEL

```
module mux_2to1_structural (
    input d0, d1,
    input sel,
    output dout
);

wire not_sel, y0, y1;
```

```
not (not_sel, sel);
and (y0, d0, not_sel);
and (y1, d1, sel);
or (dout, y0, y1);
endmodule
```

## MULTIPLEXER

## BEHAVIORAL MODEL (USING CASE)

```
module mux_8to1_case (
    input [7:0] din,
    input [2:0] sel,
    output reg dout
);
always @(din or sel) begin
    case (sel)
        3'b000: dout = din[0];
        3'b001: dout = din[1];
        3'b010: dout = din[2];
        3'b011: dout = din[3];
        3'b100: dout = din[4];
        3'b101: dout = din[5];
        3'b110: dout = din[6];
        3'b111: dout = din[7];
    endcase
end
endmodule
```

## BEHAVIORAL MODEL (USING IF-ELSE)

```
module mux_8to1_if_else (
    input [7:0] din,
    input [2:0] sel,
    output reg dout
);
always @(din or sel) begin
    if (sel == 3'b000) dout = din[0];
    else if (sel == 3'b001) dout = din[1];
    else if (sel == 3'b010) dout = din[2];
    else if (sel == 3'b011) dout = din[3];
    else if (sel == 3'b100) dout = din[4];
    else if (sel == 3'b101) dout = din[5];
    else if (sel == 3'b110) dout = din[6];
```

```
    else dout = din[7];
end
endmodule
```

## DATAFLOW MODEL

```
module mux_8to1_dataflow (
    input [7:0] din,
    input [2:0] sel,
    output dout
);
assign dout = (sel == 3'b000 & din[0]) |
              (sel == 3'b001 & din[1]) |
              (sel == 3'b010 & din[2]) |
              (sel == 3'b011 & din[3]) |
              (sel == 3'b100 & din[4]) |
              (sel == 3'b101 & din[5]) |
              (sel == 3'b110 & din[6]) |
              (sel == 3'b111 & din[7]);
Endmodule
verilog
```

CopyEdit

## STRUCTURAL MODEL

```
module mux_8to1_structural (
    input [7:0] din,
    input [2:0] sel,
    output dout
);
wire s0, s1, s2;
wire y0, y1, y2, y3, y4, y5, y6, y7;

not (s0, sel[0]);
not (s1, sel[1]);
not (s2, sel[2]);

and (y0, din[0], s2, s1, s0);
and (y1, din[1], s2, s1, sel[0]);
and (y2, din[2], s2, sel[1], s0);
and (y3, din[3], s2, sel[1], sel[0]);
and (y4, din[4], sel[2], s1, s0);
and (y5, din[5], sel[2], s1, sel[0]);
and (y6, din[6], sel[2], sel[1], s0);
and (y7, din[7], sel[2], sel[1], sel[0]);
```

```
or (dout, y0, y1, y2, y3, y4, y5,
y6, y7);
endmodule
```

## Binary to Gray code converter (behavioral model)

```
module binary_to_gray (
    input [3:0] bin,
    output reg [3:0] gray
);

always @(bin) begin
    gray[3] = bin[3];
    gray[2] = bin[3] ^ bin[2];
    gray[1] = bin[2] ^ bin[1];
    gray[0] = bin[1] ^ bin[0];
end
endmodule
```

## Gray to binary code converter (behavioral model)

```
module gray_to_binary (
    input [3:0] gray,
    output reg [3:0] bin
);

always @(gray) begin
    bin[3] = gray[3];
    bin[2] = bin[3] ^ gray[2];
    bin[1] = bin[2] ^ gray[1];
    bin[0] = bin[1] ^ gray[0];
end
endmodule
```

## Ripple carry adder behavioral model (using always block)

```
module
ripple_carry_adder_behavioral (
    input [3:0] a, b,
    input cin,
    output reg [3:0] sum,
    output reg cout
);

always @(a or b or cin) begin
    {cout, sum} = a + b + cin;
```

```
end
endmodule
```

## Dataflow Model (Using Assign)

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

## Structural Model (Using Full Adders)

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

```
module ripple_carry_adder_structural (
    input [3:0] a, b,
    input cin,
    output [3:0] sum,
    output cout
);
wire c1, c2, c3;
full_adder FA0 (a[0], b[0], cin, sum[0], c1);
full_adder FA1 (a[1], b[1], c1, sum[1], c2);
full_adder FA2 (a[2], b[2], c2, sum[2], c3);
full_adder FA3 (a[3], b[3], c3, sum[3], cout);
endmodule
```

## Booth Array Multiplier

```
module booth_multiplier (
    input [3:0] multiplicand,
    input [3:0] multiplier,
    output [7:0] product
);
```

```
reg [7:0] p;
integer i;
reg prev_bit;
always @(multiplicand,
multiplier) begin
    p = 0;
    prev_bit = 0;
    for (i = 0; i < 4; i = i + 1) begin
        case ({multiplier[i],
prev_bit})
            2'b10: p = p -
(multiplicand << i);
            2'b01: p = p +
(multiplicand << i);
            2'b00: p = p;
            2'b11: p = p;
        endcase
        prev_bit = multiplier[i];
    end
end
assign product = p;
endmodule
```