Combinational Circuits Verilog

1. Half Adder

1. Gateleve Model:

```
module ha_gt(input a,b,output s,c);
    and(c,a,b);
    xor(s,a,b);
endmodule
```

2. Data-flow Model:

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

3. Another Method

```
module ha(input a,b,output s,c);
{s,c}=a+b;
endmodule
```

4. Test-Bench:

```
module ha_tb;
    reg a,b;
    wire s,c;
    ha_df DUT(.a(a),.b(b),.c(c),.s(s));
    initial begin
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    end
    initial begin
    $monitor("Time=%0t|a=%b,b=%b,s=%b,c=%b",$time,a,b,s,c);
    end
endmodule
```

2. Full Adder:

1. Data-flow model

```
module fa_df(input a,b,cin,output s,cout);
    assign s=a^b^cin;
    assign c=(a&b)|(cin&(a^b));
endmodule
```

2. Gate-level model:

```
module fa_gt(input a,b,cin,output s,cout);
    wire x,y,z;
    and(x,a,b);
    xor(y,a,b);
    xor(s,y,cin);
    and(z,cin,y);
    or(cout,x,z);
endmodule
```

3. Testbench:

```
module tb_fa;
    reg a, b, cin;
    wire s_df, cout_df, s_gt, cout_gt;

    // Instantiate both models
    fa_df dut_df(.a(a), .b(b), .cin(cin), .s(s_df), .cout(cout_df));
    fa_gt dut_gt(.a(a), .b(b), .cin(cin), .s(s_gt), .cout(cout_gt));

    initial begin
        $monitor("Time = %0t | a = %b, b = %b, cin = %b | s_df = %b, cout_df = %b | s_gt = %
                 $time, a, b, cin, s_df, cout_df, s_gt, cout_gt);

        // Apply test cases
        a = 0; b = 0; cin = 0; #10;
        a = 0; b = 0; cin = 1; #10;
        a = 0; b = 1; cin = 0; #10;
        a = 0; b = 1; cin = 1; #10;
        a = 1; b = 0; cin = 0; #10;
        a = 1; b = 0; cin = 1; #10;
        a = 1; b = 1; cin = 0; #10;
        a = 1; b = 1; cin = 1; #10;
    end
endmodule
```

4. Other method:

```
module fa_truth_table(input a, b, cin, output reg s, cout);
    always @(*) begin
        case ({a, b, cin})
            3'b000: {cout, s} = 2'b00;
            3'b001: {cout, s} = 2'b01;
            3'b010: {cout, s} = 2'b01;
            3'b011: {cout, s} = 2'b10;
            3'b100: {cout, s} = 2'b01;
            3'b101: {cout, s} = 2'b10;
            3'b110: {cout, s} = 2'b10;
            3'b111: {cout, s} = 2'b11;
            default: {cout, s} = 2'b00;
        endcase
    end
endmodule
```

```
module fa_arithmetic_split(input a, b, cin, output s, cout);
    wire [1:0] result;
    assign result = a + b + cin;
    assign s = result[0];
    assign cout = result[1];
endmodule
```

3. Half Subtractor

1. Data-flow Model:

```
module hs_df(input a,b,output d,bout);
    assign d=a^b;
    assign bout=~a&b;
endmodule
```

2. Gate-level:

```
module hs_gt(input a, b, output d, bout);
    wire x, y;   // Intermediate signals

    // XOR gate for the difference (d)
    xor(x, a, b);  // x = a ^ b
    assign d = x;   // Difference output

    // AND and NOT gates for the borrow-out (bout)
    not(y, a);      // y = ~a
    and(bout, y, b); // bout = ~a & b
endmodule
```

3. Testbench:

```
module hs_tb;
    reg a, b;
    wire d, bout;

    // Instantiate the half-subtractor module
    hs_gt DUT (.a(a), .b(b), .d(d), .bout(bout));

    initial begin
        // Apply test cases
        $monitor("Time = %0t | a = %b, b = %b | d = %b, bout = %b", $time, a, b, d, bout);
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $finish;
    end
endmodule
```

4. Full subtractor

1. Gate-flow:

```
module fs(input a,b,bin,output d,bout);
    assign d=a^b^bin;
    assign bout=(~a&b)|(bin&(a~^b));
endmodule
```

Comparator

1. 1-bit comparator

```
module comp_1(input a,b,output altb,agtb,aeqb);
    assign aeqb=a~^b;
    assign altb=~a&b;
    assign agtb=~b&a;
endmodule
```

2. Other Methods:

```
module comp_1(input a, b, output altb, agtb, aeqb);
    always @(*) begin
        if (a == b) begin
            aeqb = 1;
            altb = 0;
            agtb = 0;
        end else if (a < b) begin
            aeqb = 0;
            altb = 1;
            agtb = 0;
        end else begin
            aeqb = 0;
            altb = 0;
            agtb = 1;
        end
    end
endmodule
```

```
module comp_1(input a, b, output altb, agtb, aeqb);
    wire eq_ab, gt_ab, lt_ab;

    xor(eq_ab, a, b);        // aeqb = a ~^ b
    and(altb, ~a, b);        // altb = ~a & b
    and(agtb, a, ~b);        // agtb = a & ~b

    assign aeqb = ~eq_ab;    // aeqb is 1 when a == b
    assign lt_ab = altb;     // altb is 1 when a < b
    assign gt_ab = agtb;     // agtb is 1 when a > b
endmodule
```

2. n-bit comparator

```
module n_comp(input [n-1:0] a,b,output reg aeqb,altb,agtb);
always @(*) begin
    if (a == b) begin
            aeqb = 1;
            altb = 0;
            agtb = 0;
        end else if (a < b) begin
            aeqb = 0;
            altb = 1;
            agtb = 0;
        end else begin
            aeqb = 0;
            altb = 0;
            agtb = 1;
        end
    end
endmoduled
```

Decoder

1. 2to4 decoder:

```
module decoder_2to4(
    input [1:0] a,   // 2-bit input
    output reg [3:0] y  // 4-bit output
);
    always @(*) begin
        case(a)
            2'b00: y = 4'b0001; // output 0001 when a = 00
            2'b01: y = 4'b0010; // output 0010 when a = 01
            2'b10: y = 4'b0100; // output 0100 when a = 10
            2'b11: y = 4'b1000; // output 1000 when a = 11
            default: y = 4'b0000; // default case (optional)
        endcase
    end
endmodule
```

```
module decoder_2to4(input a,b, output [3:0] y);
    assign y[0]=~a&~b;
    assign y[1]=~a&b;
    assign y[2]=a&~b;
    assign y[3]=a&b;
endmodule
```

3. Test bench

```
module tb_decoder_2to4;
    reg [1:0] a;      // 2-bit input
    wire [3:0] y;     // 4-bit output

    // Instantiate the decoder module
    decoder_2to4 uut (
        .a(a),
        .y(y)
    );

    initial begin
        // Test all possible input values for a
        $monitor("a = %b, y = %b", a, y);

        a = 2'b00; #10;  // a = 00, expected y = 0001
        a = 2'b01; #10;  // a = 01, expected y = 0010
        a = 2'b10; #10;  // a = 10, expected y = 0100
        a = 2'b11; #10;  // a = 11, expected y = 1000
        $finish;
    end
endmodule
```

Multiplexer

1. 2to1

```
module mux_2to1(input s,a,b,output y);
    assign y=~s&a | s&b;
endmodule
```

```
module mux_2to1(input s, a, b, output reg y);
    always @(*) begin
        case(s)
            1'b0: y = a;  // When s is 0, output is a
            1'b1: y = b;  // When s is 1, output is b
            default: y = 1'b0; // Default case (usually not needed in this scenario)
        endcase
    end
endmodule
```

```
module mux_2to1(input s, a, b, output y);
    assign y = (s == 1'b0) ? a : b;
endmodule
```

```
module mux_4to2(input [1:0] s, input [3:0] a, output reg [1:0] y);
    always @(*) begin
        case(s)
            2'b00: y = a[1:0];  // Select a[1:0] when s = 00
            2'b01: y = a[3:2];  // Select a[3:2] when s = 01
            2'b10: y = a[1:0];  // Select a[1:0] when s = 10
            2'b11: y = a[3:2];  // Select a[3:2] when s = 11
            default: y = 2'b00;  // Default case
        endcase
    end
endmodule
```

Demultiplexer:

```
module demux_4to2(input a,input [1:0] s,output reg y[3:0]);
always @(*) begin
    case(s)
        2'b00:y[0]=a;
        2'b01:y[1]=a;
        2'b10:y[2]=a;
        2'b11:y[3]=a;
    endcase
end
endmodule
```

Encoder

1. 4to2

```
module en_4to2(input [3:0] a,output reg [1:0] y);
always @(*) begin
    case(a)
        4'b0001: y=2'b00;
        4'b0010: y=2'b01;
        4'b0100: y=2'b10;
        4'b1000: y=2'b11;
    endcase
endmodule
```

2. Priority Encoder 4to2

```
module priority_en_4to2(input [3:0] a, output reg [1:0] y);
always @(*) begin
    if (a[3])        // Check the highest-priority input
        y = 2'b11;   // If `a[3]` is high, set output to 11
    else if (a[2])   // Check the next-highest-priority input
        y = 2'b10;   // If `a[2]` is high, set output to 10
    else if (a[1])   // Check the next priority
        y = 2'b01;   // If `a[1]` is high, set output to 01
    else if (a[0])   // Check the lowest-priority input
        y = 2'b00;   // If `a[0]` is high, set output to 00
    else
        y = 2'bxx;   // Invalid case (no input is high)
end
endmodule
```

SR Flip-Flop

```
module sr_ff(input s,r,clk,output reg q,q_bar);
    always @(posedge clk) begin
        case({s,r})
            2'b00:
            begin
                q<=q;
                q_bar<=q_bar;
            end
             2'b01:
            begin
                q<=1'b0;
                q_bar<=1'b1;
            end
             2'b10:
            begin
                q<=1'b1;
                q_bar<=1'b0;
            end
             2'b11:
            begin
                q<=1'bx;
                q_bar<=1'bx;
            end
            default:
            begin
                q<=1'b0;
                q_bar<=1'b1;
            end
        endcase
endmodule
```

2. Testbench

```
module sr_flip_flop_tb;
    reg S, R, clk;
    wire Q, Q_bar;

    // Instantiate the SR flip-flop
    sr_flip_flop DUT (.S(S), .R(R), .clk(clk), .Q(Q), .Q_bar(Q_bar));

    // Generate clock signal
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // Clock with 10 time unit period
    end

    // Apply test cases
    initial begin
        // Test Case 1: Set = 0, Reset = 0 (Maintain previous state)
        S = 0; R = 0; #10;

        // Test Case 2: Set = 1, Reset = 0 (Set Q to 1)
        S = 1; R = 0; #10;

        // Test Case 3: Set = 0, Reset = 1 (Set Q to 0)
        S = 0; R = 1; #10;

        // Test Case 4: Set = 1, Reset = 1 (Invalid state)
        S = 1; R = 1; #10;

        // End simulation
        $stop;
    end

    // Monitor the output
    initial begin
        $monitor("Time = %0t | S = %b | R = %b | Q = %b | Q_bar = %b",
                 $time, S, R, Q, Q_bar);
    end
endmodule
```

D-flip flop

```
module d_ff(input d,clk,output reg q,q_bar);
```

```
    always @(posedge clk) begin
        case(d)
            1'b0:begin
                q<=1'b0
                q_bar<=1'b1
            end
            1'b1:begin
                q<=1'b1
                q_bar<=1'b0
            end
            default:begin
                q<=1'b0
                q_bar<=1'b0
            end
        endcase
    end
endmodule
```

Jk Flip flop

```
module jk_ff(input j,k,clk,output reg q,q_bar);
    always @(posedge clk) begin
        case({j,k})
            2'b00:
            begin
                q<=q;
                q_bar<=q_bar;
            end
             2'b01:
            begin
                q<=1'b0;
                q_bar<=1'b1;
            end
             2'b10:
            begin
                q<=1'b1;
                q_bar<=1'b0;
            end
             2'b11:
            begin
                q<=~q;
                q_bar<=~q_bar;
            end
            default:
            begin
                q<=1'b0;
                q_bar<=1'b1;
            end
        endcase
    end
endmodule
```

T-flip flop

```
module t_ff(input t, clk, output reg q, q_bar);
    always @(posedge clk) begin
        case(t)
            1'b0: begin
                q <= q;            // No change in state when t = 0
                q_bar <= q_bar;    // No change in state
            end
            1'b1: begin
                q <= ~q;           // Toggle the flip-flop when t = 1
                q_bar <= ~q_bar;   // Toggle the inverse output
            end
            default: begin
                q <= 1'b0;         // Reset to 0 (if ever needed)
                q_bar <= 1'b1;     // Inverse of reset
            end
        endcase
    end
endmodule
```

Up-counter

- Synchronous:

```
module sync_upcounter(input reset,clk,output reg [3:0] count);
    always @(posedge clk or posedge reset) begin
        if(reset) begin
            count<=4'b0000;
        end
        else begin
            count<=count+1;
        end
    end
endmodule
```

- Asynchronous:

```
module async_upcounter(input reset,clk,output reg [3:0] count);
    always @(posedge clk or posedge reset) begin
        if(reset) begin
            count<=4'b0000;
        end
        else begin
            count[0]=~coutn[0];
        end
    end
    always @(q[0]) begin
        if(q[0]==1'b1) begin
            count[1]=~count[1];
        end
    end
    always @(q[1]) begin
        if(q[1]==1'b1) begin
            count[2]=~count[2];
        end
    end
    always @(q[2]) begin
        if(q[2]==1'b1) begin
            count[3]=~count[3];
        end
    end
endmodule
```

Down Counter

- Synchronous

```
module sync_down_counter(input clk, reset, output reg [3:0] q);
    always @(posedge clk or posedge reset) begin
        if (reset)
            q <= 4'b1111;  // Reset the counter to 15 (1111)
        else
            q <= q - 1;    // Decrement the counter on each clock cycle
    end
endmodule
```

- Asychronous

```
module async_down_counter(input clk, reset, output reg [3:0] q);
    always @(posedge clk or posedge reset) begin
        if (reset)
            q <= 4'b1111;  // Reset the counter to 15 (1111)
        else
            q[0] <= ~q[0];  // Toggle the least significant bit
    end

    always @(q[0]) begin
        if (q[0] == 1'b0)
```

```
            q[1] <= ~q[1];  // Toggle the next bit when q[0] changes
    end

    always @(q[1]) begin
        if (q[1] == 1'b0)
            q[2] <= ~q[2];  // Toggle the next bit when q[1] changes
    end

    always @(q[2]) begin
        if (q[2] == 1'b0)
            q[3] <= ~q[3];  // Toggle the next bit when q[2] changes
    end
endmodule
```

Up-Down Counter

```
module sync_up_down_counter(
    input clk,         // Clock input
    input reset,       // Reset input
    input up_down,     // Control input for direction: 1 for up, 0 for down
    output reg [3:0] q // 4-bit counter output
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            q <= 4'b0000;  // Reset the counter to 0
        end else begin
            if (up_down)    // If up_down is 1, count up
                q <= q + 1;
            else            // If up_down is 0, count down
                q <= q - 1;
        end
    end

endmodule
```

Shift Register

```
module shift_register_siso(
    input clk,          // Clock
    input reset,        // Reset
    input serial_in,    // Serial input
    output reg [3:0] q  // 4-bit output
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
```

```
            q <= 4'b0000;  // Reset to 0000
        end else begin
            q <= {q[2:0], serial_in};  // Shift left and insert serial_in at LSB
        end
    end
endmodule
```

Traffic Light:

```
module traffic_light(input clk, output reg [2:0] light);
    // States definition
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;
    // Light signals
    parameter RED = 3'b100, GREEN = 3'b010, YELLOW = 3'b001;

    // State register
    reg [1:0] states;

    always @(posedge clk) begin
        case(states)
            S0: begin
                light <= GREEN;  // Green light for S0
                states <= S1;    // Transition to S1
            end
            S1: begin
                light <= YELLOW; // Yellow light for S1
                states <= S2;    // Transition to S2
            end
            S2: begin
                light <= RED;    // Red light for S2
                states <= S0;    // Transition to S0
            end
            default: begin
                light <= RED;    // Default to RED
                states <= S0;    // Reset to S0
            end
        endcase
    end
endmodule


module tb_traffic_light;

    reg clk;  // Clock signal
    wire [2:0] light;  // Output wire for light signals
    // Instantiate the traffic_light module
    traffic_light uut (
        .clk(clk),
        .light(light)
    );

    // Generate clock signal
    always begin
        #5 clk = ~clk;  // Toggle clock every 5 time units
    end

    // Initial block to initialize values and monitor the output
    initial begin
        // Initialize signals
        clk = 0;

        // Monitor the output signals
        $monitor("At time %t, light = %b", $time, light);

        // Simulate for 20 time units to observe the light transitions
        #20;

        // Finish simulation
        $finish;
    end

endmodule
```

Vending Machine

```
module vending_machine(input [1:0] D_N, output reg bottle);
    // State and input definitions
    parameter I0 = 2'b00, I1 = 2'b01, I2 = 2'b10;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

    reg [1:0] state;  // 2-bit register to store current state

    // Initial state setup
    initial begin
        state = S0;
    end

    // Always block that changes states based on D_N input
    always @(D_N) begin
        case(state)
            S0: begin
                case(D_N)
                    I0: begin
                        state <= S0;
                        bottle <= 1'b0;
                    end
                    I1: begin
                        state <= S1;
                        bottle <= 1'b0;
                    end
                    I2: begin
                        state <= S2;
                        bottle <= 1'b0;
                    end
                    default: begin
                        state <= S0;
                        bottle <= 1'b0;
                    end
                endcase
            end

            S1: begin
                case(D_N)
                    I0: begin
                        state <= S1;
                        bottle <= 1'b0;
                    end
                    I1: begin
                        state <= S2;
                        bottle <= 1'b0;
                    end
                    I2: begin
                        state <= S3;
                        bottle <= 1'b1;  // Dispense bottle
                    end
                    default: begin
                        state <= S1;
                        bottle <= 1'b0;
                    end
                endcase
            end

            S2: begin
                case(D_N)
                    I0: begin
                        state <= S2;
                        bottle <= 1'b0;
```

```
                    end
                    I1: begin
                        state <= S3;
                        bottle <= 1'b1;  // Dispense bottle
                    end
                    I2: begin
                        state <= S3;
                        bottle <= 1'b1;  // Dispense bottle
                    end
                    default: begin
                        state <= S2;
                        bottle <= 1'b0;
                    end
                endcase
            end

            S3: begin
                case(D_N)
                    I0: begin
                        state <= S0;
                        bottle <= 1'b0;
                    end
                    I1: begin
                        state <= S1;
                        bottle <= 1'b0;
                    end
                    I2: begin
                        state <= S1;
                        bottle <= 1'b0;
                    end
                    default: begin
                        state <= S0;
                        bottle <= 1'b0;
                    end
                endcase
            end

            default: begin
                state <= S0;
                bottle <= 1'b0;
            end
        endcase
    end
endmodule
```

Parity Detector

```
module parity(input x, clk, output reg z);
    // Parameter definitions for EVEN and ODD states
    parameter EVEN = 1'b0, ODD = 1'b1;

    // Register to store the current state
    reg state;

    // Initial block to initialize the state to EVEN
    initial begin
        state <= EVEN;
    end

    // Always block that detects parity on the rising edge of the clock
    always @(posedge clk) begin
        case(state)
            EVEN: begin
                if(x) begin
                    state <= ODD;  // Transition to ODD state if x is 1
                    z <= 1;        // Set output z to 1 when transitioning to ODD
                end else begin
                    state <= EVEN; // Stay in EVEN state if x is 0
                    z <= 0;        // Set output z to 0 when staying in EVEN
                end
            end
            ODD: begin
                if(x) begin
                    state <= EVEN; // Transition to EVEN state if x is 1
                    z <= 0;        // Set output z to 0 when transitioning to EVEN
                end else begin
                    state <= ODD;  // Stay in ODD state if x is 0
                    z <= 1;        // Set output z to 1 when staying in ODD
                end
            end
            default: begin
                state <= EVEN; // Default state to EVEN if something goes wrong
                z <= 0;        // Default output z to 0
            end
        endcase
    end
endmodule


module tb_parity;
```

```
    reg x;          // Input signal
    reg clk;        // Clock signal
    wire z;         // Output signal

    // Instantiate the parity detector
    parity uut (
        .x(x),
        .clk(clk),
        .z(z)
    );

    // Generate clock signal
    always begin
        #5 clk = ~clk; // Toggle clock every 5 time units
    end

    // Test sequence
    initial begin
        // Initialize signals
        clk = 0;
        x = 0;

        // Monitor the output
        $monitor("At time %t, x = %b, z = %b", $time, x, z);

        // Test different input values
        #10 x = 1;   // Input x = 1
        #10 x = 0;   // Input x = 0
        #10 x = 1;   // Input x = 1
        #10 x = 0;   // Input x = 0
        #10 x = 1;   // Input x = 1
        #10 x = 0;   // Input x = 0

        // Finish the simulation
        $finish;
    end

endmodule
```

Simple Arbiter Priority to req 0 when both asserted

```
module arbiter(input reg req0, req1, input clk, rst, output reg gnt0, gnt1);
    // State encoding
    parameter GNT0 = 2'b00, GNT1 = 2'b01, IDLE = 2'b10;
    reg [1:0] state;
```

```
// Initial block for state and grants initialization
initial begin
    state = IDLE;
    {gnt0, gnt1} = 2'b00;
end

// Always block for state transitions and grant logic
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;
        {gnt0, gnt1} <= 2'b00;
    end
    else begin
        case (state)
            IDLE: begin
                if (req0) begin
                    state <= GNT0;  // Grant to req0
                    gnt0 <= 1'b1;
                    gnt1 <= 1'b0;  // Ensure req1 is not granted
                end
                else if (req1) begin
                    state <= GNT1;  // Grant to req1
                    gnt0 <= 1'b0;
                    gnt1 <= 1'b1;
                end
                else if (req0 && req1) begin
                    state <= GNT0;  // If both are requested, prioritize req0
                    gnt0 <= 1'b1;
                    gnt1 <= 1'b0;
                end
            end

            GNT0: begin
                if (req0) begin
                    state <= GNT0;  // Keep granting to req0
                    gnt0 <= 1'b1;
                end
                else if (~req0) begin
                    state <= IDLE;  // No more request from req0, go to IDLE
                    gnt0 <= 1'b0;
                end
            end

            GNT1: begin
                if (req1) begin
                    state <= GNT1;  // Keep granting to req1
                    gnt1 <= 1'b1;
                end
                else if (~req1) begin
                    state <= IDLE;  // No more request from req1, go to IDLE
                    gnt1 <= 1'b0;
                end
            end

            default: begin
                state <= IDLE;  // Fallback to IDLE state
                {gnt0, gnt1} <= 2'b00;
            end
        endcase
    end
end
endmodule
```