1. 二进制加法器（8%）

模块代码

```
module binary_adder (
    input [3:0] a,
    input [3:0] b,
    output [4:0] sum
);

    assign sum = a + b;

endmodule
```

仿真

```
module binary_adder_tb;

    reg [3:0] a;
    reg [3:0] b;
    wire [4:0] sum;

    binary_adder dut (
        .a(a),
        .b(b),
        .sum(sum)
    );

    initial begin
        a = 4'b0000;
        b = 4'b0000;
        #10;
        a = 4'b1010;
        b = 4'b0110;
        #10;
        // End simulation
        $finish;
    end

endmodule
```

2. 十进制加法器（8%）

模块

```
module decimal_adder(
    input [3:0] A, // 4位BCD输入A
    input [3:0] B, // 4位BCD输入B
    output [3:0] Sum, // 4位BCD输出和
    output Carry // 进位输出
);
```

```
    // 内部信号
    wire [4:0] binary_sum; // A和B的二进制和，增加一位以处理可能的溢出
    wire [3:0] correction_factor; // 如果binary_sum大于9需要添加的修正因子
    wire correction_carry; // 修正加法的进位

    // 执行二进制加法
    assign binary_sum = A + B;

    // 确定是否需要修正（binary_sum的低4位 > 9）
    assign correction_factor = (binary_sum[3:0] > 9) ? 4'b0110 : 4'b0000;

    // 如果需要，添加修正因子
    assign {correction_carry, Sum} = binary_sum[3:0] + correction_factor;

    // 确定最终进位
    assign Carry = binary_sum[4] | correction_carry;

endmodule
```

仿真

```
module decimal_addertb;

    // 输入
    reg [3:0] A;
    reg [3:0] B;

    // 输出
    wire [3:0] Sum;
    wire Carry;
    decimal_adder uut (
        .A(A),
        .B(B),
        .Sum(Sum),
        .Carry(Carry)
    );

    // 初始化输入
    initial begin
        A = 4'b0000; // 初始值为0
        B = 4'b0000; // 初始值为0

        // 添加测试样例
        // 测试样例1: A = 5, B = 3
        #10; // 等待10个时间单位
        A = 4'b0101; // A设置为5
        B = 4'b0011; // B设置为3

        // 等待并观察结果
        #10;
        // 测试结束
        $finish;
    end
```

```
endmodule
```

3. 半加器（8%）

模块

```
module half_adder(
    input a,
    input b,
    output sum,
    output carry
);

    assign sum = a ^ b;
    assign carry = a & b;

endmodule
```

仿真

```
module half_addertb;

    // 输入
    reg a;
    reg b;

    // 输出
    wire sum;
    wire carry;

    // 实例化半加器模块
    half_adder dut (
        .a(a),
        .b(b),
        .sum(sum),
        .carry(carry)
    );
    // 激励信号
    initial begin
        // 测试用例1
        a = 0;
        b = 0;
        #10;

        // 测试用例2
        a = 0;
        b = 1;
        #10;

        // 测试用例3
        a = 1;
        b = 0;
        #10;
```

```
        // 测试用例4
        a = 1;
        b = 1;
        #10;

        $finish;
    end

endmodule
```

4. 一位全加器（8%）

模块

```
module full_adder(
    input a, // 第一个加数位
    input b, // 第二个加数位
    input cin, // 进位输入
    output sum, // 和输出
    output cout // 进位输出
);

    // 中间变量，用于存储各个阶段的结果
    wire sum_ab, carry_ab, carry_ac, carry_bc;

    // a和b的加法，不考虑进位
    xor(sum_ab, a, b);
    // a和b的进位
    and(carry_ab, a, b);
    // a和进位cin的进位
    and(carry_ac, a, cin);
    // b和进位cin的进位
    and(carry_bc, b, cin);
    // 最终的和，考虑a、b以及进位cin
    xor(sum, sum_ab, cin);
    // 最终的进位，来自a和b、a和cin、b和cin的任意组合
    or(cout, carry_ab, carry_ac, carry_bc);

endmodule
```

仿真

```
module full_addertb;

    // 输入
    reg a;
    reg b;
    reg cin;

    // 输出
    wire sum;
    wire cout;
```

```
    // 实例化全加器模块
    full_adder dut (
        .a(a),
        .b(b),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );

    // 测试逻辑
    initial begin
        // 初始化输入
        a = 0;
        b = 0;
        cin = 0;

        // 应用输入并显示输出
        #10 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;

        // 结束模拟
        $finish;
    end

endmodule
```

5. 2选1数选器（8%）

模块

```
module selector (
    input wire sel,
    input wire in0,
    input wire in1,
    output wire out
);
    // 选择器模块
    // 输入: sel - 选择信号
    //       in0 - 输入0
    //       in1 - 输入1
    // 输出: out - 输出信号

    // 使用选择信号 sel 控制输出
    assign out = sel ? in1 : in0;
endmodule
```

仿真

```
module selector_tb;

    // 定义测试台架的信号
    reg sel;
    reg in0;
    reg in1;
    wire out;

    // 实例化待测模块
    selector uut (
        .sel(sel),
        .in0(in0),
        .in1(in1),
        .out(out)
    );

    // 应用测试向量
    initial begin
        // 测试用例 1
        sel = 0;
        in0 = 0;
        in1 = 1;
        #10;

        // 测试用例 2
        sel = 1;
        in0 = 0;
        in1 = 1;
        #10;

        // 测试用例 3
        sel = 0;
        in0 = 1;
        in1 = 0;
        #10;

        // 测试用例 4
        sel = 1;
        in0 = 1;
        in1 = 0;
        #10;

        // 如有需要，添加更多测试用例

        $finish;
    end

endmodule
```

6. 二进制编码器（8%）

模块

```
module binary_encoder (
    input [3:0] binary_in, // 4位二进制输入
    output reg [1:0] encoded_out // 2位编码输出
);

    // 实现二进制编码的逻辑
    always @(binary_in) begin
        case (binary_in)
            4'b0001: encoded_out = 2'b00; // 如果输入为0001，则输出00
            4'b0010: encoded_out = 2'b01; // 如果输入为0010，则输出01
            4'b0100: encoded_out = 2'b10; // 如果输入为0100，则输出10
            4'b1000: encoded_out = 2'b11; // 如果输入为1000，则输出11
            default: encoded_out = 2'bxx; // 对于其他输入，输出不确定
        endcase
    end

endmodule
```

仿真

```
module binary_encodertb;

    reg [3:0] binary_in;
    wire [1:0] encoded_out;

    binary_encoder dut (
        .binary_in(binary_in),
        .encoded_out(encoded_out)
    );

    initial begin

        // 测试用例1：输入为0001
        binary_in = 4'b0001;
        #10;

        // 测试用例2：输入为0010
        binary_in = 4'b0010;
        #10;

        // 测试用例3：输入为0100
        binary_in = 4'b0100;
        #10;

        // 测试用例4：输入为1000
        binary_in = 4'b1000;
        #10;

        // 测试用例5：输入为其他值
```

```
        binary_in = 4'b1011;
        #10;
    end

endmodule
```

7. 二进制解码器（8%）

模块

```
module binary_decoder (
    input wire [1:0] binary_in,
    output wire [3:0] decoded_out
);

    assign decoded_out[0] = ~(binary_in[1] | binary_in[0]);
    assign decoded_out[1] = ~(binary_in[1] & binary_in[0]);
    assign decoded_out[2] = ~(binary_in[1] & ~binary_in[0]);
    assign decoded_out[3] = ~(~binary_in[1] & binary_in[0]);

endmodule
```

仿真

```
module binary_decoder_tb;

    reg [1:0] binary_in;
    wire [3:0] decoded_out;

    binary_decoder dut (
        .binary_in(binary_in),
        .decoded_out(decoded_out)
    );

    initial begin

        binary_in = 2'b00;
        #10;
        binary_in = 2'b01;
        #10;
        binary_in = 2'b10;
        #10;
        binary_in = 2'b11;
        #10;
        $finish;
    end

endmodule
```

8. 1位数值比较器（4%）

模块

```
module equal_comparator (
    input wire a,
    input wire b,
    output wire out
);

// 数值比较器
assign out = (a == b);

endmodule
```

仿真

```
module equal_comparatortb;

// 输入
reg a;
reg b;

// 输出
wire out;

equal_comparator eq(
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0;b = 0;
    #10
    a = 1;b = 0;
    #10
    a = 0;b = 1;
    #10
    a = 1;b = 1;
    $finish;
    end
endmodule
```

9. 2路分配器（4%）

模块

```
module data_distributor (
    input wire enable,
    input wire select_line, // 简化为1位
    input wire [7:0] input_data,
```

```
    output reg [7:0] out0,
    output reg [7:0] out1
);

// 初始化输出
initial begin
    out0 = 0;
    out1 = 0;
end

// 数据分配逻辑
always @(enable, select_line, input_data) begin
    // 当enable为低时，所有输出为0
    if (!enable) begin
        out0 = 0;
        out1 = 0;
    end else begin
        // 根据select_line的值，将input_data分配到相应的输出
        case (select_line)
            1'b0: begin
                out0 = input_data;
                out1 = 0;
            end
            1'b1: begin
                out0 = 0;
                out1 = input_data;
            end
            default: begin
                out0 = 0;
                out1 = 0;
            end
        endcase
    end
end

endmodule
```

仿真

```
`timescale 1ns / 1ps

module data_distributortb;

    // Inputs
    reg enable;
    reg select_line;
    reg [7:0] input_data;

    // Outputs
    wire [7:0] out0;
    wire [7:0] out1;

    // Instantiate the module under test
    data_distributor dut (
        .enable(enable),
```

```
        .select_line(select_line),
        .input_data(input_data),
        .out0(out0),
        .out1(out1)
    );

    // Initialize inputs
    initial begin
        enable = 0;
        select_line = 0;
        input_data = 0;
        #10;
        enable = 1;
        select_line = 0;
        input_data = 8'hFF;
        #10;
        enable = 1;
        select_line = 1;
        input_data = 8'hAA;
        #10;
        enable = 0;
        select_line = 0;
        input_data = 0;
        #10;
        $finish;
    end

endmodule
```

10. 由1位二进制全加器生成4位二进制加法器（8%）

模块

```
module adder (
    input [3:0] a, // 4位输入a
    input [3:0] b, // 4位输入b
    output [4:0] sum // 5位输出和，包括进位
);

    wire [3:0] carry; // 内部进位信号
    wire [3:0] carry_out; // 每一位加法器的进位输出

    // 1位全加器
    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin : full_adder
            full_adder fa (
                .a(a[i]), // 第i位的a输入
                .b(b[i]), // 第i位的b输入
                .carry_in(carry[i]), // 第i位的进位输入
                .sum(sum[i]), // 第i位的和输出
                .carry_out(carry_out[i]) // 第i位的进位输出
```

```
            );
        end
    endgenerate

    // 进位链
    assign carry[0] = 1'b0; // 最低位的进位输入始终为0
    assign carry[1] = carry_out[0]; // 第1位的进位输入为第0位的进位输出
    assign carry[2] = carry_out[1]; // 第2位的进位输入为第1位的进位输出
    assign carry[3] = carry_out[2]; // 第3位的进位输入为第2位的进位输出

    // 输出和的最高位，即最高位的进位输出
    assign sum[4] = carry_out[3];

endmodule
```

仿真

```
`timescale 1ns / 1ps

module addertb;

    // Inputs
    reg [3:0] a;
    reg [3:0] b;

    // Outputs
    wire [4:0] sum;

    // 实例化被测模块
    adder uut (
        .a(a),
        .b(b),
        .sum(sum)
    );

    initial begin
        // 初始化输入
        a = 0;
        b = 0;

        // 等待100ns用于全局复位
        #100;

        // 测试用例1
        a = 4'b0001; // 1
        b = 4'b0010; // 2
        #10; // 等待10ns

        // 测试用例2
        a = 4'b0100; // 4
        b = 4'b0101; // 5
        #10; // 等待10ns

        // 测试用例3
        a = 4'b1111; // 15
```

```
        b = 4'b0001; // 1
        #10; // 等待10ns

        // 测试用例4
        a = 4'b1010; // 10
        b = 4'b1011; // 11
        #10; // 等待10ns

        // 完成测试
        $finish;
    end
endmodule
```

11. 由2选1数据选择器生成4选1数据选择器（4%）

模块

```
module selector_four(
    input wire [3:0] in, // 4个输入
    input wire [1:0] sel, // 2位选择信号
    output wire out
);
    wire out_upper, out_lower;

    // 使用两个2选1选择器来实现上半部分和下半部分的选择
    selector upper_mux(
        .a(in[2]),
        .b(in[3]),
        .sel(sel[0]),
        .out(out_upper)
    );

    selector lower_mux(
        .a(in[0]),
        .b(in[1]),
        .sel(sel[0]),
        .out(out_lower)
    );

    // 使用另一个2选1选择器来根据sel[1]选择最终输出
    selector final_mux(
        .a(out_lower),
        .b(out_upper),
        .sel(sel[1]),
        .out(out)
    );
endmodule
```

仿真

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
/
// Company:
// Engineer:
//
// Create Date: 2024/07/12 10:00:00
// Design Name:
// Module Name: selector_four_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
// 用于测试selector_four模块的仿真代码，包含中文注释。
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/

module selector_four_tb;

    // 测试模块的输入
    reg [3:0] in;
    reg [1:0] sel;
    // 测试模块的输出
    wire out;

    // 实例化被测试模块
    selector_four uut (
        .in(in),
        .sel(sel),
        .out(out)
    );

    initial begin
        // 初始化输入
        in = 4'b1010; // 输入值为1010
        sel = 2'b00; // 初始选择信号为00

        // 等待100ns后更改选择信号
        #100;
        sel = 2'b01; // 更改选择信号为01

        #100;
        sel = 2'b10; // 更改选择信号为10

        #100;
        sel = 2'b11; // 更改选择信号为11
```

```
        #100;
        $finish; // 结束仿真
    end

endmodule
```

12. 由1位数值比较器生成4位数值比较器（4%）

模块

```
module equal_comparator_four(
    input [3:0] a,
    input [3:0] b,
    output equal
);

wire [0:3] equal_bits;

// Instantiate 1-bit comparators
equal_comparator one_bit_comp0(a[0], b[0], equal_bits[0]);
equal_comparator one_bit_comp1(a[1], b[1], equal_bits[1]);
equal_comparator one_bit_comp2(a[2], b[2], equal_bits[2]);
equal_comparator one_bit_comp3(a[3], b[3], equal_bits[3]);

// Output is true if all bits are equal
assign equal = (equal_bits[0] & equal_bits[1] & equal_bits[2] & equal_bits[3]);

endmodule
```

仿真

```
module equal_comparator_fourtb;

    //输入
    reg [3:0] a;
    reg [3:0] b;

    //输出
    wire equal;

    equal_comparator_four uut(
        .a(a),
        .b(b),
        .equal(equal)
        );
    initial begin
        a = 4'b1010;
        b = 4'b1010;

        #100
```

```
        a = 4'b1001;
        b = 4'b1010;

        #100
        a = 4'b1111;
        b = 4'b1111;

        #100
        a = 4'b0000;
        b = 4'b1111;
        #100
        $finish; // 结束仿真
    end
endmodule
```

13. 由2路分配器生成4路分配器（4%）

模块

```
module selector_four(
    input wire [3:0] in, // 4个输入
    input wire [1:0] sel, // 2位选择信号
    output wire out
);
    wire out_upper, out_lower;
    // 使用两个2选1选择器来实现上半部分和下半部分的选择
    selector upper_mux(
        .a(in[2]),
        .b(in[3]),
        .sel(sel[0]),
        .out(out_upper)
    );

    selector lower_mux(
        .a(in[0]),
        .b(in[1]),
        .sel(sel[0]),
        .out(out_lower)
    );

    // 使用另一个2选1选择器来根据sel[1]选择最终输出
    selector final_mux(
        .a(out_lower),
        .b(out_upper),
        .sel(sel[1]),
        .out(out)
    );
endmodule
```

仿真

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
/
// Company:
// Engineer:
//
// Create Date: 2024/07/12 10:00:00
// Design Name:
// Module Name: selector_four_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
// 用于测试selector_four模块的仿真代码，包含中文注释。
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/

module selector_four_tb;

    // 测试模块的输入
    reg [3:0] in;
    reg [1:0] sel;
    // 测试模块的输出
    wire out;

    // 实例化被测试模块
    selector_four uut (
        .in(in),
        .sel(sel),
        .out(out)
    );

    initial begin
        // 初始化输入
        in = 4'b1010; // 输入值为1010
        sel = 2'b00; // 初始选择信号为00

        // 等待100ns后更改选择信号
        #100;
        sel = 2'b01; // 更改选择信号为01

        #100;
        sel = 2'b10; // 更改选择信号为10

        #100;
        sel = 2'b11; // 更改选择信号为11
```

```
        #100;
        $finish; // 结束仿真
    end

endmodule
```

14. 2位计数器（4%）

模块

```
module counter (
    input wire clk, // 时钟信号
    input wire reset, // 异步复位信号
    output wire [1:0] count // 2位计数器输出
);

    reg [1:0] count_reg; // 内部寄存器用于存储当前计数值

    // 当时钟上升沿到来或复位信号上升沿到来时执行
    always @(posedge clk or posedge reset) begin
        if (reset)
            count_reg <= 2'b00; // 如果复位信号被激活，则计数器清零
        else
            count_reg <= count_reg + 1; // 否则，计数器值加1
    end

    assign count = count_reg; // 将内部寄存器的值赋给输出

endmodule
```

仿真

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
/
// Company:
// Engineer:
//
// Create Date:
// Design Name:
// Module Name: counter_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
// 该仿真代码用于测试counter模块。它会模拟时钟信号和复位信号，
// 并观察计数器的输出。
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/

module counter_tb;

    // 仿真模块的输入（在测试台中，输入变为reg类型）
    reg clk;
    reg reset;
    // 仿真模块的输出（在测试台中，输出变为wire类型）
    wire [1:0] count;

    // 实例化被测试的模块
    counter uut (
        .clk(clk),
        .reset(reset),
        .count(count)
    );

    // 生成时钟信号
    always #10 clk = ~clk; // 时钟周期为20ns

    initial begin
        // 初始化
        clk = 0;
        reset = 1; // 开始时激活复位信号
        #30; // 等待一段时间
        reset = 0; // 释放复位信号，开始计数

        // 观察计数器的行为
        #100; // 让仿真运行一段时间

        // 再次复位计数器
        reset = 1;
        #20;
        reset = 0;

        #50; // 再次观察计数器的行为

        $finish; // 结束仿真
    end

endmodule
```

15. 七段数字显示译码器（6%）

模块

```
module seven_segment_display_decoder(
    input [3:0] binary_input,
    output reg [6:0] segment_output
);
```

```
    always @(*)
    begin
        case(binary_input)
            4'b0000: segment_output = 7'b1111110; // 0
            4'b0001: segment_output = 7'b0110000; // 1
            4'b0010: segment_output = 7'b1101101; // 2
            4'b0011: segment_output = 7'b1111001; // 3
            4'b0100: segment_output = 7'b0110011; // 4
            4'b0101: segment_output = 7'b1011011; // 5
            4'b0110: segment_output = 7'b1011111; // 6
            4'b0111: segment_output = 7'b1110000; // 7
            4'b1000: segment_output = 7'b1111111; // 8
            4'b1001: segment_output = 7'b1111011; // 9
            default: segment_output = 7'b0000000; // Invalid input
        endcase
    end

endmodule
```

仿真

```
module seven_segment_display_decodertb;

    reg [3:0] binary_input;
    wire [6:0] segment_output;

    seven_segment_display_decoder uut (
        .binary_input(binary_input),
        .segment_output(segment_output)
    );

    initial begin

        binary_input = 4'b0000;
        #10;

        binary_input = 4'b0001;
        #10;

        binary_input = 4'b0010;
        #10;

        binary_input = 4'b0011;
        #10;

        binary_input = 4'b0100;
        #10;

        binary_input = 4'b0101;
        #10;

        binary_input = 4'b0110;
        #10;

        binary_input = 4'b0111;
```

```
        #10;

        binary_input = 4'b1000;
        #10;

        binary_input = 4'b1001;
        #10;

        binary_input = 4'b1010; // Invalid input
        #10;

        $finish;
    end

endmodule
```

16. 简单算术逻辑单元（10%）

模块

```
module ALU(input [3:0] A, input [3:0] B, input [2:0] opcode, output reg [3:0]
result);

    always @*
    begin
        case(opcode)
            3'b000: result = A + B; // Addition
            3'b001: result = A - B; // Subtraction
            3'b010: result = A & B; // Bitwise AND
            3'b011: result = A | B; // Bitwise OR
            3'b100: result = ~A;    // Bitwise NOT
            default: result = 4'b0;  // Default value
        endcase
    end

endmodule
```

仿真

```
module ALU_tb;

    // Inputs
    reg [3:0] A;
    reg [3:0] B;
    reg [2:0] opcode;

    // Outputs
    wire [3:0] result;

    // 实例化被测试的模块
    ALU uut (
        .A(A),
        .B(B),
```

```
        .opcode(opcode),
        .result(result)
    );

    initial begin
        // 初始化输入
        A = 0;
        B = 0;
        opcode = 0;

        // 等待仿真稳定
        #100;

        // 测试加法
        A = 4'b0011; // 3
        B = 4'b0101; // 5
        opcode = 3'b000; // 加法
        #10;

        // 测试减法
        A = 4'b1010; // 10
        B = 4'b0011; // 3
        opcode = 3'b001; // 减法
        #10;

        // 测试按位与
        A = 4'b1100; // 12
        B = 4'b0110; // 6
        opcode = 3'b010; // 按位与
        #10;

        // 测试按位或
        A = 4'b1001; // 9
        B = 4'b0101; // 5
        opcode = 3'b011; // 按位或
        #10;

        // 测试按位非
        A = 4'b1001; // 9
        opcode = 3'b100; // 按位非
        #10;

        // 结束仿真
        $finish;
    end

endmodule
```