

Verilog PPT 例程

二选一多路转换器

```
module mux2addr(a,b,s,out);
    input a,b,s;
    output reg out;
    always @(a,b,s) begin
        if (s) begin
            out <= a;
        end else begin
            out <= b;
        end
    end
endmodule
```

比较器

```
module compare (a,b,equal);
    input a,b;
    output equal;
    assign equal = (a==b) ? 1:0;
endmodule
```

奇偶校验电路

```
module parity(data,out)
    input [7:0] data;
    output out;
    assign out = ^data; //若Data中有偶数个1，则Dout=0
endmodule
```

总线选择电路

```
module sel4_1(BusA,BusB,BusC,BusD,Se1,Dout);
    input [3:0] BusA,BusB,BusC,BusD;
    input [1:0] Se1;
    output [3:0] Dout;
    assign Dout = (Se1==2'b00) ? BusA:
        (Se1==2'b01) ? BusB:
        (Se1==2'b10) ? BusC:
        BusD;
endmodule
```

循环移位

```
module cycle_shift(shiftDat, clk, rst_n);
    output reg[7:0] shiftDat;
    input clk, rst_n;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            shiftDat <= 8'b0110_0010;
        end else begin
            shiftDat <= {shiftDat[6:0], shiftDat[7]};
        end
    end
end
endmodule
```

3-8 译码器

- 例程1

```
module Decoder(out, in, en, clk, rst_n)
    output reg[7:0] out;
    input [2:0] in;
    input en;
    input clk, rst_n;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            out <= 8'b0000_0000;
        end else if (en) begin
            out <= 8'b1 << in;
        end else begin
            out <= 8'b0000_0000;
        end
    end
end
endmodule
```

- 例程2

```
module Decoder(out, in, en)
    output reg[7:0] out;
    input [2:0] in;
    input en;
    always @(in) begin
        case(in)
            3'b000 : out = 8'b1111_1110;
            3'b001 : out = 8'b1111_1101;
            3'b010 : out = 8'b1111_1011;
            3'b011 : out = 8'b1111_0111;
            3'b100 : out = 8'b1110_1111;
            3'b101 : out = 8'b1101_1111;
            3'b110 : out = 8'b1011_1111;
            3'b111 : out = 8'b0111_1111;
            default: out = 8'b0000_0000;
        endcase
    end
end
endmodule
```

二分频器

```
module div_2 (clk, rst, out);
    input clk, rst;          // 输入时钟和复位信号
    output out;              // 输出信号
    reg q;                   // 寄存器用于存储状态

    always @(posedge clk or negedge rst) begin
        if (!rst)
            q <= 1'b0;       // 异步复位，输出置为 0
        else
            q <= ~q;         // 翻转状态，实现二分频
        end

    assign out = q;          // 将寄存器 q 的值赋给输出
endmodule
```

移位寄存器

```
module pipeb2(Q3,D,clk);
    output [7:0] Q3;
    input [7:0] D;
    input clk;
    reg [7:0] Q3,Q2,Q1;
    always @(posedge clk) begin
        Q3 <= Q2;
        Q2 <= Q1;
        Q1 <= D;
    end
endmodule
```

模10计数器

```
module cnt10 (clk, rst, cnt);
    parameter WIDTH = 8;
    input clk, rst;
    output [3:0] cnt;
    reg [3:0] cnt;
    always @(posedge clk or posedge rst) begin
        if (rst)
            cnt <= 0;
        else begin
            if(cnt == 9)
                cnt<=0;
            else
                cnt <= cnt + 1;
        end
    end
endmodule
```

数据选择器

```
module select(sel, a,b,c,d, mux);
    output reg mux;
    input [1:0] sel;
    input a,b,c,d;
    always @(*) begin
        case (sel)
            2'b00: mux = a;
            2'b01: mux = b;
            2'b10: mux = c;
            2'b11: mux = d;
            default: mux = 1'b0; // 默认值, 防止综合工具报错
        endcase
    end
endmodule
```

- case语句必须在一个过程块中, 如 always @(*)

这里sel可以用wire型吗,mux可以用wire型吗,a,b,c,d要指定位宽吗

1. sel 是否可以用 wire 型?

- 可以: sel 是一个输入信号, 输入信号默认是 wire 类型, 因此可以直接使用 wire 型。
- 注意: 如果 sel 是组合逻辑的输入信号, 默认 wire 类型即可, 无需显式声明为 wire。

2. mux 是否可以用 wire 型?

- 不可以: 在你的代码中, mux 是通过 always 块中的 case 语句赋值的, 而 always 块只能对 reg 类型的信号赋值。因此, mux 必须是 reg 类型。
- 总结: 如果 mux 的值是通过 always 块赋值的, 必须定义为 reg 类型; 如果是通过 assign 语句赋值, 则可以使用 wire 类型。

3. a, b, c, d 是否需要指定位宽?

- 需要: 虽然 a, b, c, d 默认是 1 位宽, 但为了代码的可读性和明确性, 建议显式定义它们的位宽。
- 示例: 如果它们是单比特信号, 可以定义为:

```
input a, b, c, d; // 单比特信号
```

如果它们是多比特信号 (例如 8 位宽), 则需要明确指定位宽:

```
input [7:0] a, b, c, d; // 多比特信号
```

总结

- sel: 可以用 wire 型, 默认是 wire。
- mux: 不能用 wire 型, 必须用 reg 型, 因为它在 always 块中赋值。
- a, b, c, d: 建议显式指定位宽, 即使是单比特信号, 也可以定义为 input [0:0] 或 input。

循环语句

★ 除了forever语句外,其他语句都是可综合,但循环次数必须在程序编译时确定,动态改变循环次数的语句是不可综合的。

- forever

forever循环语句常用于产生周期性的波形,用来作为仿真测试信号。它与 always语句不同之处在于不能独立写在程序中,而必须写在initial块中。

```
forever
begin
    #((ClockPeriod/2)-2) clk_i = 0;
    #((ClockPeriod/2)+2) clk_i = 1;
end
```

- repeat

在repeat语句中,其表达式通常为常量表达式。下面的例子中使用repeat循环语句及加法和移位操作来实现一个乘法器。

```
parameter size=8, longsize=16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin mult
    reg[longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat(size)
        begin
            if(shift_opb[1]) begin
                result = result + shift_opa;
                shift_opa = shift_opa << 1;
                shift_opb = shift_opb >> 1;
            end
        end
    end
end
```

- for

```
begin :init_mem
    reg[7:0] tempi;
    for(tempi=0; tempi<memsize; tempi=tempi+1) begin
        memory[tempi] = 0;
    end
end
```

- while

```

module CountOnes(input [7:0] a, output reg[3:0] count )
    always @(*) begin : countls
        tempreg = a;
        count = 0;
        while(tempreg != 0) begin
            if (tempreg[0])
                count = count + 1;
            tempreg = tempreg >> 1;
        end
    end
endmodule

```

`begin : countls` 是一个命名块, `countls` 是块的名称。

testbench

3-8译码器testbench

这个 testbench 会:

1. 包含 `timescale` 定义。
2. 声明 testbench 模块。
3. 声明连接到 `Decoder` 模块的信号 (`reg` 用于输入, `wire` 用于输出)。
4. 定义时钟周期参数。
5. 实例化被测模块 (`Decoder`)。
6. 生成时钟信号 (`clk`)。
7. 生成复位信号 (`rst_n`) 序列。
8. 生成输入激励 (`en`, `in`), 覆盖各种情况 (复位、使能关闭、使能打开并遍历所有输入)。
9. 使用 `$monitor` 或 `$display` 来观察输入和输出的变化。
10. 在适当的时候使用 `$finish` 结束仿真。

Verilog

```

// timescale 定义: 时间单位为 1ns, 精度为 1ps
`timescale 1ns / 1ps

// Testbench 模块
module decoder_tb;

    // 参数定义
    parameter CLK_PERIOD = 10; // 时钟周期为 10ns (100 MHz)

    // 信号声明
    // 输入信号 (reg)
    reg clk;
    reg rst_n;
    reg en;
    reg [2:0] in;

    // 输出信号 (wire)
    wire [7:0] out;

    // 实例化被测模块 (DUT - Design Under Test)
    Decoder dut (
        .out      (out),    // 连接输出端口
        .in       (in),     // 连接输入端口

```

```

        .en      (en),      // 连接使能端口
        .clk     (clk),     // 连接时钟端口
        .rst_n   (rst_n)   // 连接复位端口
    );

// 时钟生成
initial begin
    clk = 0; // 初始时钟为 0
    // forever 循环产生时钟，每半个周期翻转一次
    forever #(CLK_PERIOD / 2) clk = ~clk;
end

// 激励和仿真控制
initial begin
    // 1. 初始化和复位
    $display("Time=%0t: Starting simulation, Initializing inputs.", $time);
    rst_n = 1'b1; // 先置为高电平
    en     = 1'b0; // 初始使能为低
    in     = 3'b000; // 初始输入为 0
    # (CLK_PERIOD / 4); // 短暂等待

    $display("Time=%0t: Asserting Reset (rst_n = 0).", $time);
    rst_n = 1'b0; // 拉低复位信号，开始复位
    # (CLK_PERIOD * 2); // 保持复位 2 个时钟周期

    $display("Time=%0t: De-asserting Reset (rst_n = 1).", $time);
    rst_n = 1'b1; // 释放复位
    # (CLK_PERIOD / 2); // 等待半个周期确保复位完全释放

    // 2. 测试使能 (en) 为低的情况
    $display("Time=%0t: Test Case 1 - Enable is LOW (en = 0).", $time);
    en = 1'b0;
    in = 3'b010; // 改变输入，但输出应保持为 0
    # CLK_PERIOD;
    in = 3'b101;
    # CLK_PERIOD;
    // 预期输出: out 应保持为 8'b0000_0000

    // 3. 测试使能 (en) 为高的情况，遍历所有输入
    $display("Time=%0t: Test Case 2 - Enable is HIGH (en = 1). Cycling
through inputs.", $time);
    en = 1'b1; // 拉高使能信号
    for (integer i = 0; i < 8; i = i + 1) begin
        in = i; // 施加输入 000 到 111
        $display("Time=%0t: Applied in = %b", $time, in);
        # CLK_PERIOD; // 等待一个时钟周期让输出更新
        // 预期输出: out 应该等于 8'b1 左移 in 位
    end

    // 4. 再次测试使能 (en) 为低的情况
    $display("Time=%0t: Test Case 3 - Enable is LOW again (en = 0).",
$time);
    en = 1'b0;
    in = 3'b000; // 即使输入改变，输出也应变为 0
    # CLK_PERIOD;
    // 预期输出: out 应变为 8'b0000_0000

    // 5. 结束仿真

```

```

# (CLK_PERIOD * 5); // 再等待几个周期
$display("Time=%0t: Simulation Finished.", $time);
$finish; // 结束仿真
end

// (可选) 使用 $monitor 监控信号变化
initial begin
    // $monitor 会在任何被监控信号发生变化时打印信息
    $monitor("Time=%0t: rst_n=%b en=%b in=%b => out=%b",
        $time, rst_n, en, in, out);
end

endmodule

```

如何使用这个 Testbench:

1. 将 Decoder 模块的代码和这个 decoder_tb 模块的代码保存在同一个文件 (例如 decoder.v) 或两个不同的文件 (例如 decoder.v 和 decoder_tb.v) 中。
2. 使用 Verilog 仿真器 (如 ModelSim, VCS, Xcelium, Icarus Verilog 等) 编译并运行 testbench 模块 (decoder_tb)。

- 如果使用 Icarus Verilog (免费):

Bash

```

iverilog -o decoder_sim decoder.v decoder_tb.v
vvp decoder_sim

```

- 如果使用 ModelSim/QuartaSim:

Tcl

```

vlib work          # 创建 work 库 (如果需要)
vlog decoder.v decoder_tb.v # 编译 verilog 文件
vsim work.decoder_tb      # 加载 testbench 进行仿真
run -all               # 运行仿真直到 $finish

```

3. 观察仿真器输出的 \$display 和 \$monitor 信息, 验证 out 信号是否在 en 为高时根据 in 的值正确地产生了 one-hot 编码, 并且在复位或 en 为低时输出为 0。

对于22-23年期末程序设计题第1题

四选一数据选择器的testbench

```

`timescale 1ns/1ns
module tb_top_mux;
    parameter DELAY = 10;
    reg [1:0] sel;
    reg a,b,c,d;
    wire mux;
    select uut(
        .sel(sel),
        .a(a),
        .b(b),
        .c(c),

```



```

        .d(d),
        .mux(mux)
    );
    initial begin
        $display("test1");
        a = 1'b0;
        b = 1'b1;
        c = 1'b0;
        d = 1'b1;
        sel = 2'b01;
        # DELAY;
        $display("test2");
        a = 1'b1;
        b = 1'b0;
        c = 1'b0;
        d = 1'b1;
        sel = 2'b10;
        #DELAY;
        $display("test3");
        a = 1'b1;
        b = 1'b0;
        c = 1'b1;
        d = 1'b0;
        sel = 2'b10;
        #DELAY;
        $ display("finish test");
        $finish; // 结束仿真
    end
endmodule

```

Q&A

always @(posedge clk or negedge rst) 后可以没有begin吗? if 后可以没有begin吗?

在 Verilog 中, `always` 和 `if` 后是否需要 `begin` 取决于后面语句的数量:

1. `always @(posedge clk or negedge rst)` **后是否需要 begin**
 - **不需要 begin**: 如果 `always` 块中只有一条语句, 则可以省略 `begin` 和 `end`。
 - **需要 begin**: 如果 `always` 块中有多条语句, 则必须使用 `begin` 和 `end` 将这些语句包裹起来。

示例:

```

always @(posedge clk or negedge rst)
    q <= 1'b0; // 单条语句, 不需要 begin 和 end

always @(posedge clk or negedge rst) begin
    q <= 1'b0; // 多条语句, 需要 begin 和 end
    d <= 1'b1;
end

```

2. if 后是否需要 begin

- **不需要 begin**：如果 if 或 else 后只有一条语句，则可以省略 begin 和 end。
- **需要 begin**：如果 if 或 else 后有多条语句，则必须使用 begin 和 end。

示例：

```
if (rst)
    q <= 1'b0; // 单条语句，不需要 begin 和 end

if (rst) begin
    q <= 1'b0; // 多条语句，需要 begin 和 end
    d <= 1'b1;
end
```

3. 你的代码分析

在你的代码中：

```
always @(posedge clk or negedge res_n) begin
    if (!res_n) begin
        out <= 8'b 0000_0000;
    end else if (en) begin
        out <= 8'b1 << in[2:0];
    end else begin
        out <= 8'b0000_0000;
    end
end
end
```

- always 块中有多条语句，因此需要 begin 和 end。
- if 和 else 块中也有多条语句，因此也需要 begin 和 end。

如果你省略了 begin 和 end，代码会导致语法错误或逻辑错误。

什么时候用wire，什么时候用reg，定义reg 和定义wire有什么区别

在 Verilog 中，wire 和 reg 是两种不同的数据类型，用于不同的场景，不能任意替换。以下是它们的区别和使用场景：

1. wire

- **定义**：wire 是一种连续赋值的信号类型，表示硬件中的连接线。
- **特点**：
 - 只能通过 **连续赋值语句**（assign）或 **模块实例化的端口** 驱动。
 - 无法在 always 块中赋值。
 - 默认值是高阻态（z）。
- **使用场景**：
 - 用于组合逻辑的输出。
 - 用于模块之间的连接信号。
 - 用于驱动 assign 语句。

示例：

```
wire a, b, c;
assign c = a & b; // 组合逻辑, c 是 wire 类型
```

2. reg

- **定义:** reg 是一种存储类型信号, 表示硬件中的寄存器。
- **特点:**
 - 必须在 **过程块** (如 always 或 initial) 中赋值。
 - 可以存储值, 直到下一次赋值。
 - 默认值是未知态 (x) 。
- **使用场景:**
 - 用于时序逻辑的输出。
 - 用于 always 块中存储信号的值。

示例:

```
reg q;
always @(posedge clk) begin
    q <= d; // 时序逻辑, q 是 reg 类型
end
```

3. 区别总结

特性	wire	reg
赋值方式	通过 assign 或模块端口驱动	在 always 或 initial 块中赋值
存储能力	无存储能力	有存储能力
默认值	高阻态 (z)	未知态 (x)
使用场景	组合逻辑、模块连接	时序逻辑、存储信号

4. 是否可以任意定义?

- **不能任意定义:** wire 和 reg 的使用场景不同, 必须根据信号的用途选择合适的类型。
 - 如果信号需要存储值 (如在 always 块中赋值), 必须使用 reg。
 - 如果信号是组合逻辑的输出或模块间的连接, 必须使用 wire。

5. 你的代码分析

在你的代码中:

```
output reg [7:0] out;
```

- out 被定义为 reg 类型, 因为它在 always 块中被赋值。
- 如果将 out 定义为 wire, 代码会报错, 因为 wire 不能在 always 块中赋值。

如果你需要组合逻辑输出, 可以使用 wire, 例如:

```
wire [7:0] out;
assign out = 8'b1 << in[2:0]; // 组合逻辑
```

定义 input output 时，一定要定义位宽吗

在 Verilog 中，定义输入和输出信号时是否需要指定位宽取决于信号的具体需求：

1. 是否必须定义位宽

- **不必**：如果信号是单比特（1 位宽），可以省略位宽定义，默认位宽为 1 位。

```
input en; // 等价于 input [0:0] en;
output out; // 等价于 output [0:0] out;
```

- **必须**：如果信号是多比特（宽度大于 1 位），必须明确指定位宽，否则会导致语法错误或逻辑错误。

2. 你的代码分析

在你的代码中：

```
input [2:0] in; // 输入信号 in 是 3 位宽，必须定义位宽
output reg [7:0] out; // 输出信号 out 是 8 位宽，必须定义位宽
input en; // 输入信号 en 是单比特，可以省略位宽
```

- **in 和 out**：这些信号是多比特信号，必须定义位宽，否则工具无法知道它们的具体宽度。
- **en**：这是单比特信号，可以省略位宽定义。

3. 示例

(1) 定义位宽的正确写法

```
module Example(input [3:0] a, input b, output [7:0] c);
    // a 是 4 位宽, b 是单比特, c 是 8 位宽
    assign c = {4'b0000, a} + b; // 示例逻辑
endmodule
```

(2) 省略单比特信号的位宽

```
module Example(input a, input b, output c);
    // a, b, c 都是单比特信号
    assign c = a & b; // 示例逻辑
endmodule
```

4. 总结

- **单比特信号**：可以省略位宽定义，默认是 1 位。
- **多比特信号**：必须明确指定位宽，否则会导致错误。
- **建议**：为了代码清晰，即使是单比特信号，也可以显式定义为 `[0:0]`，但这不是强制要求。

rst_n和clk是什么类型的

如果 `clk` 和 `rst_n` 是模块的输入信号，并且没有显式声明为其他类型（如 `reg`），它们默认是 `wire` 类型。

为什么在testbench里rst,clk输入要用reg型而在模块内rst,reg输入默认是wire型

在 Verilog 中，`reg` 和 `wire` 的使用取决于信号的用途和赋值方式。以下是为什么在 Testbench 中 `rst` 和 `clk` 通常定义为 `reg` 类型，而在模块内它们默认是 `wire` 类型的原因：

1. 模块内的 `rst` 和 `clk` 默认是 `wire` 类型

- 输入信号默认是 `wire` 类型：
 - 在 Verilog 中，模块的输入信号（如 `rst` 和 `clk`）默认是 `wire` 类型，表示这些信号是由外部驱动的。
 - `wire` 类型的信号不能存储值，只能通过连续赋值（`assign`）或模块端口驱动。
- 模块内的用途：
 - 在模块内部，`rst` 和 `clk` 通常用于驱动时序逻辑（如触发器或寄存器），它们的值由外部 Testbench 或其他模块提供，因此不需要存储值。

示例：模块内的 `rst` 和 `clk`

```
module example(  
    input clk,           // 默认是 wire 类型  
    input rst_n          // 默认是 wire 类型  
);  
    reg [7:0] counter;  
  
    always @(posedge clk or negedge rst_n) begin  
        if (!rst_n)  
            counter <= 0;  
        else  
            counter <= counter + 1;  
        end  
endmodule
```

- `clk` 和 `rst_n` 是输入信号，由外部驱动，因此默认是 `wire` 类型。

2. Testbench 中的 `rst` 和 `clk` 通常是 `reg` 类型

- Testbench 中需要手动驱动信号：
 - 在 Testbench 中，`rst` 和 `clk` 的值需要通过 `initial` 或 `always` 块手动赋值，以模拟输入信号的变化。
 - 只有 `reg` 类型的信号才能在过程块（如 `initial` 或 `always`）中赋值。
- Testbench 的用途：
 - Testbench 的作用是模拟外部环境，驱动被测模块（DUT: Device Under Test）的输入信号。
 - 因此，`rst` 和 `clk` 在 Testbench 中通常定义为 `reg` 类型，以便在过程块中赋值。

示例：Testbench 中的 rst 和 clk

```
module tb_example;
    reg clk;           // 定义为 reg 类型，用于手动驱动
    reg rst_n;         // 定义为 reg 类型，用于手动驱动
    wire [7:0] counter;

    example uut (
        .clk(clk),
        .rst_n(rst_n),
        .counter(counter)
    );

    initial begin
        // 初始化信号
        clk = 0;
        rst_n = 0;

        // 释放复位
        #5 rst_n = 1;

        // 模拟时钟信号
        forever #5 clk = ~clk;
    end
endmodule
```

- 在 Testbench 中，clk 和 rst_n 是 reg 类型，因为它们需要在 initial 和 always 块中赋值。

3. 总结对比

场景	信号类型	原因
模块内输入信号	wire	输入信号由外部驱动，默认是 wire 类型，不能在模块内存储值。
Testbench 信号	reg	Testbench 中需要手动驱动信号，reg 类型可以在 initial 或 always 块中赋值。

4. 关键点

- 模块内：rst 和 clk 是输入信号，默认是 wire 类型，由外部驱动。
- Testbench 中：rst 和 clk 是手动驱动的信号，因此需要定义为 reg 类型。

这种设计是 Verilog 的基本规则，确保模块和 Testbench 的职责分离：模块描述硬件逻辑，而 Testbench 模拟外部环境。