《计算机组成与设计》
实验报告

姓名: 刘韬

学院: 竺可桢学院

专业: 人工智能

邮箱: 3220103422@zju.edu.cn

报告日期: 2024 年 3 月 28 日

Lab0: 实现多路选择器 MUX

一、 实验步骤

Part I 新建工程文件

点击 Create New Project， 创建名为 Project1 的工程到对应的文件目录， 后选择 RTL Project， 此时把 Do not specify at this time 勾上， 表示在新建工程时不去指定源文件。在器件(Defalut Part)界面， 选择之前添加好的板卡 Nexys A7-100T， 然后完成工程创建。

Part II 新建源文件和 MUX 的功能实现

添加 Design Sources， 选择 Create design sources， 并命名为 MUX.v， 输入下面代码:

```
module MUX(
    input [15:0]SW,
    output [3:0]LED
    );
    assign LED = SW[14]?(SW[15]?0:SW[7:4]):(SW[15]?SW[11:8]:SW[3:0]);
endmodule
```

以上代码实现了四输入 MUX， 其中 SW[15:14] 作为选择信号。

- SW[15:14]=0 时输出 SW[3:0]。
- SW[15:14]=1 时输出 SW[7:4]。
- SW[15:14]=2 时输出 SW[11:8]。
- SW[15:14]=3 时输出常数 0。
- 输出直接绑到四个 LED 灯。

上面的 assign 语句是判断 SW[14]， 然后判断 SW[15]， 选择对应的输出。

二、 仿真

Part I 仿真代码

添加 Simulation Sources， 选择 Create， 命名为 MUX_tb.v， 输入测试代码:

```
module MUX_tb;
    reg [15:0]SW;
    wire [3:0]LED;
    MUX m1(SW,LED);

    initial begin
        SW[13:12] = 2'b00;
        SW[11:0] = 12'b011100110001;
        SW[15:14] = 2'b00;#50;
```

```
        SW[15:14] = 2'b01;#50;
        SW[15:14] = 2'b10;#50;
        SW[15:14] = 2'b11;#50;
    end
endmodule
```

测试代码固定 SW[11:0] = 12'b011100110001;。那么

在 SW[15:14] == 00 时， LED = 4b'0001 对应第四个灯亮， 输出 SW[3:0];

在 SW[15:14] == 01 时， LED = 4b'0011 对应第三四个灯亮， 输出 SW[7:4];

在 SW[15:14] == 10 时， LED = 4b'0111 对应第二三四个灯亮， 输出 SW[11:8];

在 SW[15:14] == 11 时， LED = 4b'0000 对应全部灯不亮， 输出 0。

Part II 仿真分析

仿真波形截图如下:

图 1: 多路选择器仿真波形图

仿真图像上看到当 SW[15:14] == 00 时， LED = 1； 当 SW[15:14] == 01 时， LED = 3； 当 SW[15:14] == 10 时， LED = 7； 当 SW[15:14] == 11 时， LED = 0;与预期相符。(由于 SW[15:14]是在高位， 所以值为 01 时显示为 4'b0100， 10 时显示为 4'b1000， 11 时显示为 4'b1100 即 c， 此时左边 value 栏内是最后一段的值)

三、 进行下板验证

Part I 约束代码

添加约束文件 constraints， 输入以下约束代码:

```
# LED
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14
```

```
Sch=led[3]

# SW
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15
Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14
Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14
Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14
Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14
Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14
Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14
Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14
Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35
Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14
Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14
Sch=sw[15]
```

约束代码分别对应 LED 灯和 SWITCH 开关，SWITCH 作为输入，LED 作为输出显示。

Part II 验证结果

下面是验证结果:

SW[11:0]=12b'011100110001，上图中已经标出 SW[15:14]的各种取值，对应的输出，均满足预期。

四、 思考题

1. Error 是出在 Generate Bitstream 阶段的，因为有一些 I/O 端口缺少约束
2. 可能的解决方案:
   1. 指定所有 I/O 标准，完善所有的引脚约束
   2. 或者使用如下命令: set_property SEVERITY {Warning} [get_drc_checks NSTD-1]来允许未约束的端口生成比特流文件。使用这个命令的方法是将它输入到一个.tcl 文件中，然后通过右键点击

左侧 Program and Debug 后点击 Bitstream Setting 将刚刚创立的.tcl 文件指定为 tcl.pre.就可以继续 Generate Bitstream。

3. 如何得到解决方案的: 我是通过阅读报错信息直接得到以上两种解决方案的。

五、 讨论、心得

lab0 主要是对 vivado 的使用和熟悉，已经基本掌握了添加源代码文件，仿真，下板验证的流程。使用的是 vivado2022.2，配置过程中没有遇到特别大的问题。值得吐槽的是 vivado 的编辑器，配色和区分十分蛋疼，打开的第一眼就和他说拜拜了。不过总体来说，这次实验还是比较顺利的，希望之后的实验也能顺利进行（绝不是 flag）。

lab1: Warm Up

一、 ALU 设计及实现

Part I 模块实现

新建源文件，输入下面设计代码:

```
module ALU (
  input [31:0]  A,
  input [31:0]  B,
  input [3:0]   ALU_operation,
  output[31:0]  res,
  output        zero
);
// Your code
    reg [31:0] result;
    always @(*) begin
        case (ALU_operation)
            4'd0: result = A + B;   //ADD
            4'd1: result = A - B;   //SUB
            4'd2: result = A << B[4:0]; //SLL
            4'd3: result = $signed(A) < $signed(B) ? 32'b1 : 32'b0; //SLT
            4'd4: result = $unsigned(A) < $unsigned(B) ? 32'b1 : 32'b0; //SLTU
            4'd5: result = A ^ B;   //XOR
            4'd6: result = A >> B[4:0]; //SRL
            4'd7: result = $signed(A) >>> B[4:0]; //SRA
            4'd8: result = A | B;   //OR
            4'd9: result = A & B;   //AND
            default: result = 32'b0;
        endcase
    end
    assign res = result;
    assign zero = (result == 0);
endmodule
```

我们使用 case 来完成选择，根据不同的 ALU_operation 输入来选择操作的种类。

Part II 仿真测试

新建仿真文件，输入下面测试代码:

```
module ALU_tb;
    reg [31:0]  A, B;
    reg [3:0]   ALU_operation;
    wire[31:0]  res;
    wire        zero;
    ALU ALU_u(
        .A(A),
        .B(B),
        .ALU_operation(ALU_operation),
```

```
    .res(res),
    .zero(zero)
);

initial begin
    A=32'hA5A5A5A5;
    B=32'h5A5A5A5A;
    ALU_operation =4'b0000;
    #100;
    ALU_operation =4'b0001;
    #100;
    ALU_operation =4'b0010;
    #100;
    ALU_operation =4'b0011;
    #100;
    ALU_operation =4'b0100;
    #100;
    ALU_operation =4'b0101;
    #100;
    ALU_operation =4'b0110;
    #100;
    ALU_operation =4'b0111;
    #100;
    ALU_operation =4'b1000;
    #100;
    ALU_operation =4'b1001;
    #100;

    // Overflow case
    ALU_operation = 4'b0000;
    A=32'hFFFFFFFF;
    B=32'h00000001;
    #100;

    ALU_operation = 4'b0001;
    A=32'h00000001;
    B=32'h00000002;#100;
    A=32'h00000001;
    B=32'hFFFFFFFF;#100;

    ALU_operation = 4'b0010;
    A=32'h00001000;
    B=32'h00000004;#100;

    ALU_operation = 4'b0011;
    A=32'h00000001;
    B=32'h00000001;#100;
    A=32'hFFFFFFFF;
    B=32'h00000001;#100;
    A=32'h00000001;
    B=32'hFFFFFFFF;#100;


    ALU_operation = 4'b0100;
```

```
    A=32'h00000001;
    B=32'h00000001;#100;
    A=32'hFFFFFFFF;
    B=32'h00000001;#100;
    A=32'h00000001;
    B=32'hFFFFFFFF;#100;

    ALU_operation = 4'b0110;
    A=32'h00001000;
    B=32'h00000004;#100;

    ALU_operation = 4'b0111;
    A=32'hFFFF0000;
    B=32'h00000004;#100;
    A=32'h0FFF0000;#100;


  end
endmodule
```

在上面的测试代码中，我们测试了常规情况和边界情况，下面是仿真结果:

1. 常规情况下，我们测试了所有操作的结果:

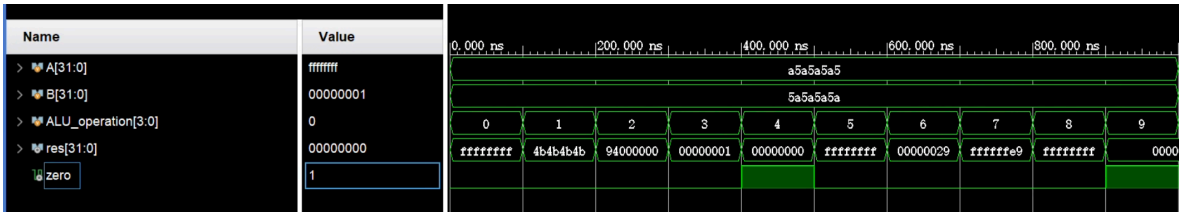

图 2: ALU 常规情况仿真波形图

两个操作数的值为

- A = 32'hA5A5A5A5 = 1010 0101 1010 0101 1010 0101 1010 0101
- B = 32'h5A5A5A5A = 0101 1010 0101 1010 0101 1010 0101 1010
  - ALU_operation = 4'b0000(ADD): A + B = 32'hFFFFFFFF
  - ALU_operation = 4'b0001(SUB): A - B = 32'h4B4B4B4B
  - ALU_operation = 4'b0010(SLL): A << B[4:0] =32'hA5A5A5A5 << 26 = 32'h94000000
  - ALU_operation = 4'b0011(SLT): A < B = 32'h00000001，此时是有符号数的比较，故$A < 0 < B$。
  - ALU_operation = 4'b0100(SLTU): A < B = 32'h00000000，此时是无符号数的比较，故$A > B$。同时 zero 输出也为 1.
  - ALU_operation = 4'b0101(XOR): A ^ B = 32'hFFFFFFFF
  - ALU_operation = 4'b0110(SRL): A >> B[4:0] = 32'hA5A5A5A5 >> 26 = 32'h00000029，逻辑右移，高位补 0。
  - ALU_operation = 4'b0111(SRA): A >>> B[4:0] = 32'hFFFFFFE9，算数右移，高位补符号位 1。

- ALU_operation = 4'b1000(OR): A | B = 32'hFFFFFFFF
- ALU_operation = 4'b1001(AND): A & B = 32'h00000000，此时 zero 输出也为 1。

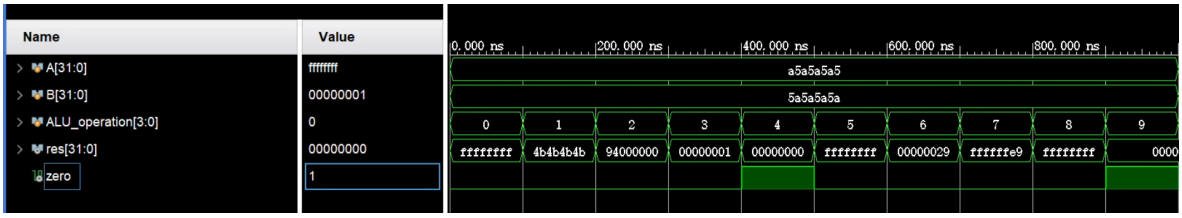

图 3: ALU 溢出情况仿真波形图

2. 边界情况下，我们测试了溢出的情况:
   - ALU_operation = 4'b0000(ADD): A = 32'hFFFFFFFF, B = 32'h00000001, A + B = 32'h00000000，此时发生了溢出，结果为 0。
   - ALU_operation = 4'b0001(SUB): A = 32'h00000001, B = 32'h00000002, A - B = 32'hFFFFFFFF，此时发生了溢出，结果为-1。
   - ALU_operation = 4'b0001(SUB): A = 32'h00000001, B = 32'hFFFFFFFF, A - B = 32'h00000002，此时发生了溢出，结果为 2。
   - ALU_operation = 4'b0010(SLL): A = 32'h00001000, B = 32'h00000004, A << B[4:0] = 32'h00008000，此时发生了溢出，结果为 32768。
   - ALU_operation = 4'b0011(SLT): A = 32'h00000001, B = 32'h00000001, A < B = 32'h00000000，此时发生了溢出，结果为 0。
   - ALU_operation = 4'b0011(SLT): A = 32'hFFFFFFFF, B = 32'h00000001, A < B = 32'h00000001，此时发生了溢出，结果为 1。
   - ALU_operation = 4'b0011(SLT): A = 32'h00000001, B = 32'hFFFFFFFF, A < B = 32'h00000000，此时发生了溢出，结果为 0。
   - ALU_operation = 4'b0100(SLTU): A = 32'h00000001, B = 32'h00000001, A < B = 32'h00000000，此时发生了溢出，结果为 0。
   - ALU_operation = 4'b0100(SLTU): A = 32'hFFFFFFFF, B = 32'h00000001, A < B = 32'h00000000，此时发生

二、 Register Files

Part I 模块实现

新建源文件，输入下面设计代码:

```
module Regs(
  input clk,
  input rst,
  input [4:0] Rs1_addr,
```

```
    input [4:0] Rs2_addr,
    input [4:0] Wt_addr,
    input [31:0]Wt_data,
    input RegWrite,
    output [31:0] Rs1_data,
    output [31:0] Rs2_data,
    output [31:0] Reg00,
    output [31:0] Reg01,
    output [31:0] Reg02,
    output [31:0] Reg03,
    output [31:0] Reg04,
    output [31:0] Reg05,
    output [31:0] Reg06,
    output [31:0] Reg07,
    output [31:0] Reg08,
    output [31:0] Reg09,
    output [31:0] Reg10,
    output [31:0] Reg11,
    output [31:0] Reg12,
    output [31:0] Reg13,
    output [31:0] Reg14,
    output [31:0] Reg15,
    output [31:0] Reg16,
    output [31:0] Reg17,
    output [31:0] Reg18,
    output [31:0] Reg19,
    output [31:0] Reg20,
    output [31:0] Reg21,
    output [31:0] Reg22,
    output [31:0] Reg23,
    output [31:0] Reg24,
    output [31:0] Reg25,
    output [31:0] Reg26,
    output [31:0] Reg27,
    output [31:0] Reg28,
    output [31:0] Reg29,
    output [31:0] Reg30,
    output [31:0] Reg31
);
// Your code here
    reg [31:0] Regdata[1:31];
    integer i;

    always @(posedge clk) begin
        if (rst)
            for (i=1; i<32; i=i+1)
                Regdata[i] <= 0;
        else
            if (RegWrite && Wt_addr != 0)
                Regdata[Wt_addr] <= Wt_data;
    end
    assign Rs1_data = (Rs1_addr ==0) ? 0 : Regdata[Rs1_addr];
    assign Rs2_data = (Rs2_addr ==0) ? 0 : Regdata[Rs2_addr];
    assign Reg00 = 0;
```

```
    assign Reg01 = Regdata[1];
    assign Reg02 = Regdata[2];
    assign Reg03 = Regdata[3];
    assign Reg04 = Regdata[4];
    assign Reg05 = Regdata[5];
    assign Reg06 = Regdata[6];
    assign Reg07 = Regdata[7];
    assign Reg08 = Regdata[8];
    assign Reg09 = Regdata[9];
    assign Reg10 = Regdata[10];
    assign Reg11 = Regdata[11];
    assign Reg12 = Regdata[12];
    assign Reg13 = Regdata[13];
    assign Reg14 = Regdata[14];
    assign Reg15 = Regdata[15];
    assign Reg16 = Regdata[16];
    assign Reg17 = Regdata[17];
    assign Reg18 = Regdata[18];
    assign Reg19 = Regdata[19];
    assign Reg20 = Regdata[20];
    assign Reg21 = Regdata[21];
    assign Reg22 = Regdata[22];
    assign Reg23 = Regdata[23];
    assign Reg24 = Regdata[24];
    assign Reg25 = Regdata[25];
    assign Reg26 = Regdata[26];
    assign Reg27 = Regdata[27];
    assign Reg28 = Regdata[28];
    assign Reg29 = Regdata[29];
    assign Reg30 = Regdata[30];
    assign Reg31 = Regdata[31];


endmodule
```

需要特别注意的是，第 0 号寄存器需要始终保持恒定值 0，无法通过写寄存器修改其值。因此在读取寄存器值的时候直接对 0 地址进行返回即可。

Part II 仿真测试

新建仿真文件，输入下面测试代码:

```
module Regs_tb;
    reg clk;
    reg rst;
    reg [4:0] Rs1_addr;
    reg [4:0] Rs2_addr;
    reg [4:0] Wt_addr;
    reg [31:0]Wt_data;
    reg RegWrite;
    wire [31:0] Rs1_data;
    wire [31:0] Rs2_data;
```

```
wire [31:0] Reg[0:31];
Regs Regs_U(
    .clk(clk),
    .rst(rst),
    .Rs1_addr(Rs1_addr),
    .Rs2_addr(Rs2_addr),
    .Wt_addr(Wt_addr),
    .Wt_data(Wt_data),
    .RegWrite(RegWrite),
    .Rs1_data(Rs1_data),
    .Rs2_data(Rs2_data),
    .Reg00(Reg[0]),
    .Reg01(Reg[1]),
    .Reg02(Reg[2]),
    .Reg03(Reg[3]),
    .Reg04(Reg[4]),
    .Reg05(Reg[5]),
    .Reg06(Reg[6]),
    .Reg07(Reg[7]),
    .Reg08(Reg[8]),
    .Reg09(Reg[9]),
    .Reg10(Reg[10]),
    .Reg11(Reg[11]),
    .Reg12(Reg[12]),
    .Reg13(Reg[13]),
    .Reg14(Reg[14]),
    .Reg15(Reg[15]),
    .Reg16(Reg[16]),
    .Reg17(Reg[17]),
    .Reg18(Reg[18]),
    .Reg19(Reg[19]),
    .Reg20(Reg[20]),
    .Reg21(Reg[21]),
    .Reg22(Reg[22]),
    .Reg23(Reg[23]),
    .Reg24(Reg[24]),
    .Reg25(Reg[25]),
    .Reg26(Reg[26]),
    .Reg27(Reg[27]),
    .Reg28(Reg[28]),
    .Reg29(Reg[29]),
    .Reg30(Reg[30]),
    .Reg31(Reg[31])
);

always #10 clk = ~clk;

initial begin
    clk = 0;
    rst = 1;
    RegWrite = 0;
    Wt_data = 0;
    Wt_addr = 0;
    Rs1_addr = 0;
```

```
        Rs2_addr = 0;
        #100
        // When RegWrite == 1, we write to the register file
        rst = 0;
        RegWrite = 1;
        Wt_addr = 5'b00101;
        Wt_data = 32'ha5a5a5a5;
        #50
        Wt_addr = 5'b01010;
        Wt_data = 32'h5a5a5a5a;
        #50
        RegWrite = 0;
        Rs1_addr = 5'b00101;
        Rs2_addr = 5'b01010;
        // When RegWrite == 0, we can not write to the register file
        Wt_addr = 5'b00101;
        Wt_data = 32'h5a5a5a5a;
        // Wt_addr = 0, keep the reg00 ==0;
        RegWrite = 1;
        Wt_addr = 5'b00000;
        Rs1_addr = 5'b00000;
        Rs2_addr = 5'b00000;
        #100 $stop();
    end

endmodule
```

测试波形如下:

三、 Finite State Machine

Part I 模块实现

新建源文件，输入下面设计代码:

```
module TruthEvaluator(
    input  clk,
    input  truth_detection,
    output trust_decision
);
    // State definition
    localparam HIGHLY_TRUSTWORTHY = 2'b00;
    localparam TRUSTWORTHY = 2'b01;
    localparam SUSPICIOUS = 2'b10;
    localparam UNTRUSTWORTHY = 2'b11;
    reg [1:0]state = 2'b00;

    // initial begin
    //     state = 2'b00;
    // end

    always @(posedge clk) begin
```

```
    if(truth_detection) begin
        case(state)
            HIGHLY_TRUSTWORTHY: state <= HIGHLY_TRUSTWORTHY;
            TRUSTWORTHY: state <= HIGHLY_TRUSTWORTHY;
            SUSPICIOUS: state <= TRUSTWORTHY;
            UNTRUSTWORTHY: state <= SUSPICIOUS;
            default: state <= 2'b00;
        endcase
    end
    else begin
        case(state)
            HIGHLY_TRUSTWORTHY: state <= TRUSTWORTHY;
            TRUSTWORTHY: state <= SUSPICIOUS;
            SUSPICIOUS: state <= UNTRUSTWORTHY;
            UNTRUSTWORTHY: state <= UNTRUSTWORTHY;
            default: state <= 2'b00;
        endcase
    end
  end

  assign trust_decision = (state[1] == 1'b0);
endmodule
```

Part II 仿真测试

新建仿真文件，输入下面测试代码:

```
module TruthEvaluator_tb;
  reg clk;
  reg truth_detection;
  wire trust_decision;

  TruthEvaluator m0(
    .clk(clk),
    .truth_detection(truth_detection),
    .trust_decision(trust_decision)
  );

  always #50 clk = ~clk;

  initial begin
    clk = 0;
    truth_detection = 0;#200;
    truth_detection = 1;#100;
    truth_detection = 0;#300;
    truth_detection = 1;#400;
    #100 $stop();
  end


endmodule
```