

浙江大学

本科实验报告

课程名称：计算机组成与设计

姓名：汪珉凯

学院：计算机科学与技术学院

专业：AI

指导教师：刘海风

报告日期：2024 年 3 月 6 日

浙江大学实验报告

课程名称: 计算机组成与设计 实验类型: 综合

实验项目名称: 实现多路选择器 MUX


学生姓名: 汪珉凯 学号: 3220100975 同组学生姓名: 赵桢、马扬松

实验地点: 紫金港东四 509 室 实验日期: 2024 年 2 月 29 日

一、操作方法与实验步骤

0. 下载并安装 vivado2022.2.

1. 新建一个工程, 我取名为 LAB0.

 LAB0 - [E:/CO_LABS/LAB0/LAB0.xpr] - Vivado 2022.2

2. 新建源文件, 取名为 MUX.v, 具体代码如下所示:

```
`timescale 1ns / 1ps

module mux (//
    input wire [15:0] SW,
    output reg [3:0] LED
);
always @(SW) begin
    case (SW[15:14])//根据 SW[15:14]的不同取值决定选择 MUX 的哪项功能
        2'b00: LED <= SW[3:0];
        2'b01: LED <= SW[7:4];
        2'b10: LED <= SW[11:8];
        2'b11: LED <= 4'b0000;//取 0
        default: LED <= 4'b0000; // Default case for safety
    endcase
end
endmodule
```

3. 新建一个仿真文件, 取名为 mux_sim.v, 具体代码如下所示:

```
`timescale 1ns / 1ps

module mux_sim;
    reg [15:0] SW;
    wire [3:0] LED;
    mux mux1(
        .SW(SW),
```

```

        .LED(LED)
    );

initial begin
    // Initialize Inputs

    SW = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

    SW = 16'b0000000000001011; // SW[15:14]=00, expecting LED to show SW[3:0]
    #100;

    SW = 16'b0100000011110000; // SW[15:14]=01, expecting LED to show SW[7:4]
    #100;

    SW = 16'b1000_1110_1010_1010; // SW[15:14]=10, expecting LED to show SW[11:8]
    #100;

    SW = 16'b1100_1111_1111_1111; // SW[15:14]=11, expecting LED to show 0000
    #100;

    // Complete the simulation
    $finish;
end

endmodule

```

4.新建一个约束文件 constraint_of_lab0.xdc，代码如下所示：

```

# LED

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }];

# SW

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }];
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }];
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }];
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }];
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }];
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }];
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }];
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }];
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }];
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }];
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }];
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }];
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }];

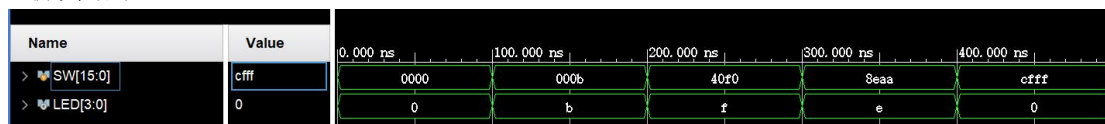
```

```
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW[15] }];
```

MUX 的输入 SW[15:0]分别对应于 PACKAGE_PIN 为 J15,L16,M13 等等的开关 (PACKAGE_PIN), MUX 的输出 LED[3:0]分别对应于 PACKAGE_PIN 为 H17,K15,J13,N14 的四盏 LED 灯 (分别编号为 0, 1, 2, 3)。

二、实验结果与分析

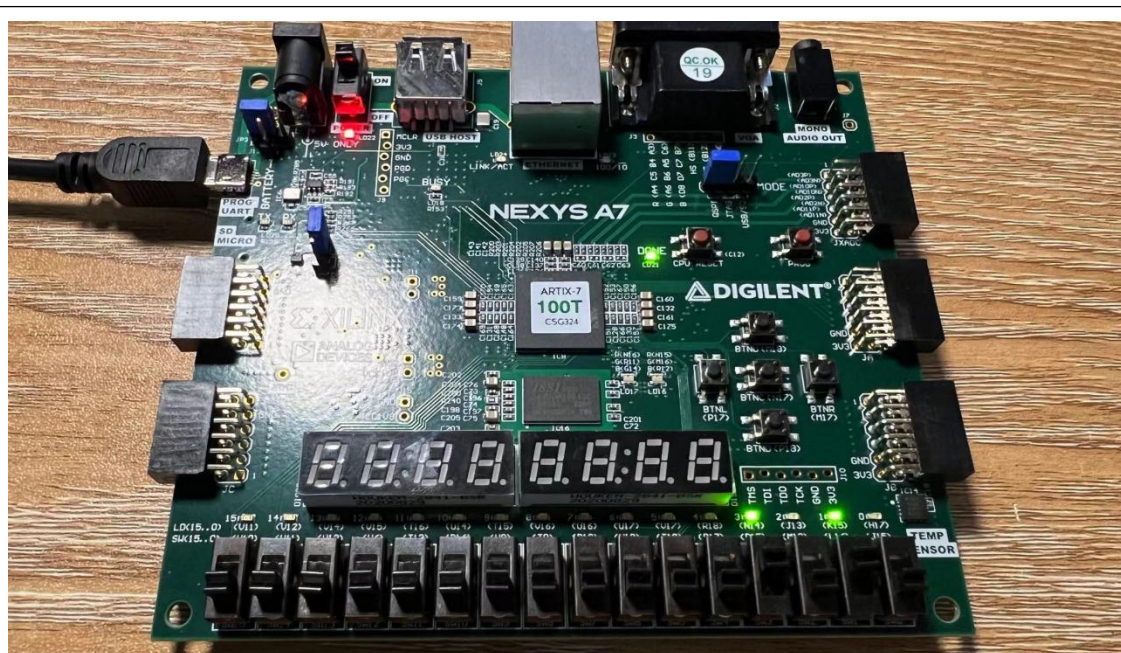
1.仿真结果



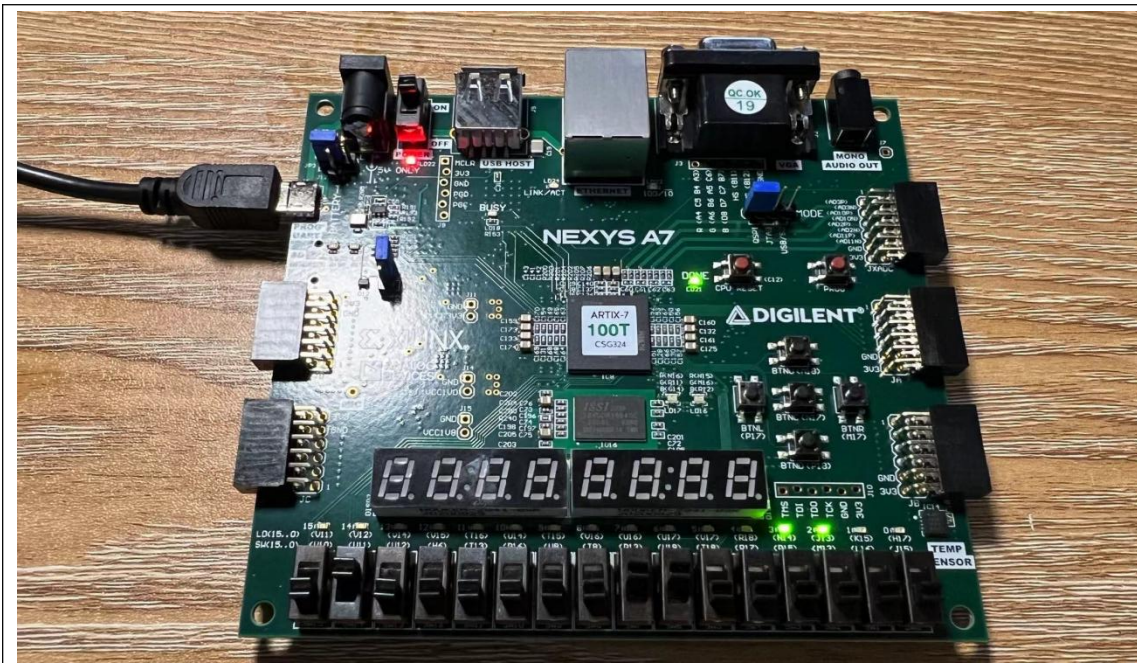
分析:

- (1) 在第 0~100ns: 状态初始化。
- (2) 在第 100~200ns: SW[15:14]=00,应该输出 SW[3:0],我们可以看到 LED 的输出值是 b, 也就是 SW 的后四位, 符合预期。
- (3) 在第 200~300ns: SW[15:14]=01,应该输出 SW[7:4],我们可以看到 LED 的输出值是 f, 也就是 SW 的第 4 位到第 7 位, 符合预期。
- (4) 在第 300~400ns: SW[15:14]=10,应该输出 SW[11:8],我们可以看到 LED 的输出值是 e, 也就是 SW 的第 8 位到第 11 位, 符合预期。
- (5) 在第 400~500ns: SW[15:14]=11,应该输出 0,我们可以看到 LED 的输出值是 0, 符合预期。

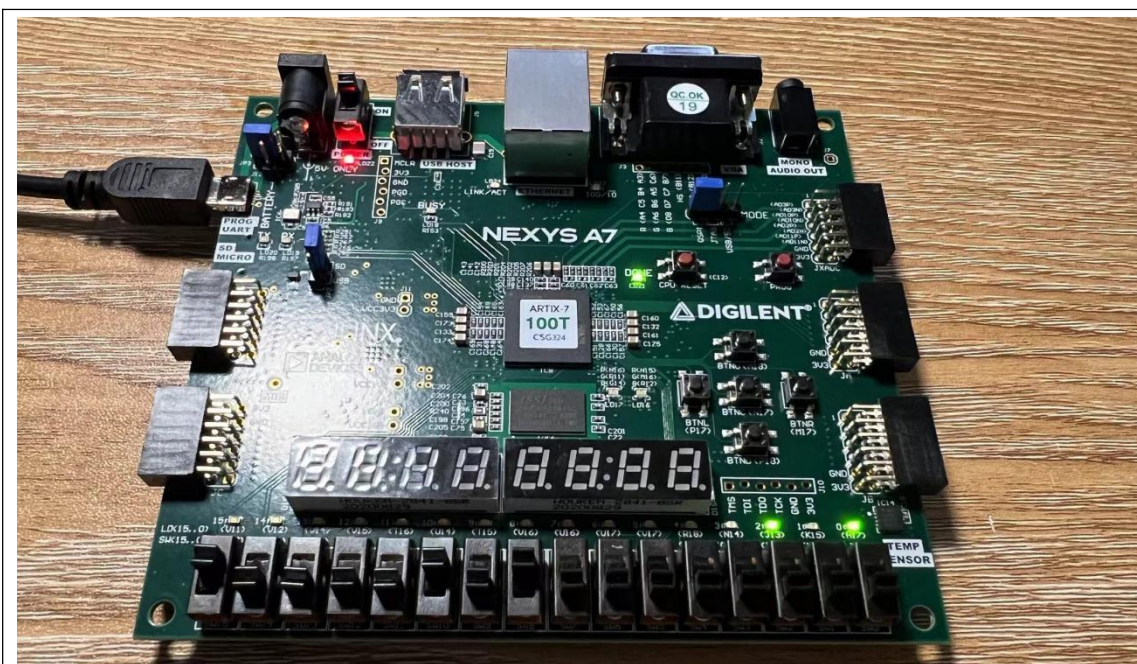
2.上板结果



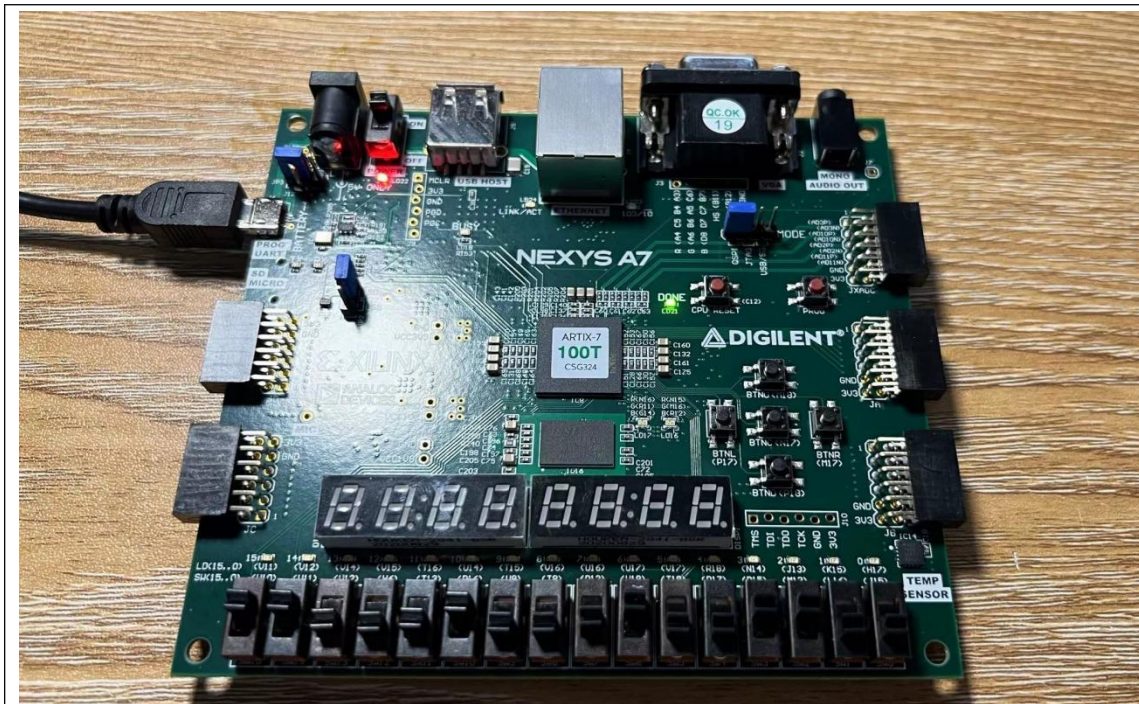
此时 SW[15:14]的值为 00,因此 4 盏 LED 灯输出 SW[3:0]代表的数。可以看到图片中 SW[3:0]=1010,而且 LED[3],LED[1]亮起, LED[2],LED[0]熄灭,与 SW[3:0]的值相同,故符合预期。



此时 SW[15:14]的值为 01,因此 4 盏 LED 灯输出 SW[7:4]代表的数。可以看到图片中 SW[7:4]=1100,而且 LED[3],LED[2]亮起, LED[1],LED[0]熄灭,与 SW[7:4]的值相同,故符合预期。



此时 SW[15:14]的值为 10,因此 4 盏 LED 灯输出 SW[11:8]代表的数。可以看到图片中 SW[11:8]=0101,而且 LED[2],LED[0]亮起, LED[3],LED[1]熄灭,与 SW[11:8]的值相同,故符合预期。



此时 SW[15:14]的值为 11,因此 4 盏 LED 灯输出 0。可以看到图片中所有 LED 灯熄灭,故符合预期。

三、讨论、心得

本次实验主要在于安装 vivado 并熟悉其使用。我的第一次安装出现了无法仿真的问题,该问题在重装系统后得以成功解决。此外,由于是第一次使用 vivado,我把这次实验的重心放在了记住 vivado 各个按钮的功能上。我相信随着实验的继续进行,我很快就能初步掌握 vivado 的使用方法。

思考题

- 1.这个 error 应该是出在 Implementation 阶段。
- 2.解决方案是为设计中的所有 I / O 添加 IOSTANDARD 和 PACKAGE_PIN 约束。
- 3.我是通过搜索引擎搜索了报错的第一句话,参阅了相关资料后了解了问题原因与解决办法。

资料链接如下: <https://cloud.tencent.com/developer/article/1759905>。

浙江大学实验报告

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称： ALU 、 RegFile、 FSM 的设计

学生姓名： 汪珉凯 学号： 3220100975 同组学生姓名： 赵桢、马扬松

实验地点： 紫金港东四 509 室 实验日期： 2024 年 3 月 6 日

一、操作方法与实验步骤

0.新建工程，命名为 LAB1.

1.设计实现 ALU 模块

1.1 建立 ALU.v 源文件，具体代码如下：

```
`timescale 1ns / 1ps

module ALU (
    input [31:0] A,
    input [31:0] B,
    input [3:0] ALU_operation,
    output[31:0] res,
    output      zero
); //输入输出
reg [31:0] res0;
reg zero0;

always @(*) begin
    case (ALU_operation) //选择不同的操作
        4'd0: res0 = A+B; //加
        4'd1: res0 = A-B; //减
        4'd2: res0 = A<<B[4:0]; //左移
        4'd3: res0 = ($signed(A)<$signed(B)) ? 1:0; //有符号判断大小
        4'd4: res0 = ($unsigned(A)<$unsigned(B)) ? 1:0; //无符号判断大小
        4'd5: res0 = A^B; //按位异或
        4'd6: res0 = A>>B[4:0]; //右移
        4'd7: res0 = $signed(A)>>>B[4:0]; //逻辑右移
        4'd8: res0 = A|B; //按位或
        4'd9: res0 = A&B; //按位与
        default: res0 = 0;
    endcase
    res = res0;
    zero = zero0;
end
```

```

        endcase

        if (res0 == 0)//检测是否为0值
            zero0 = 1;
        else zero0 = 0;
    end
    assign res = res0;
    assign zero =zero0;
endmodule

```

1.2 对 ALU 建立仿真代码 ALU_tb.v，具体代码如下：

```

`timescale 1ns / 1ps
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'b1000;
        #100;
        ALU_operation =4'b1001;
        #100;
        ALU_operation =4'b0111;
        #100;
        ALU_operation =4'b0110;
        #100;
        ALU_operation =4'b0101;
        #100;
        ALU_operation =4'b0100;
        #100;
        ALU_operation =4'b0011;
        #100;
        ALU_operation =4'b0010;
        #100;
        ALU_operation =4'b0001;
    end
endmodule

```



```

#100;

ALU_operation =4'b0000;

#100;

A=32'h01234567;
B=32'h76543210;

ALU_operation =4'b0111;

#100;

// Additional boundary tests
// Test with all zeros
ALU_operation =4'b0000;//add
A = 32'b0;
B = 32'b0;

#100;

// Test with all ones
ALU_operation =4'b0111;//and
A = 32'b1;
B = 32'b1;

#100;

// Test with maximum values
A = 32'hFFFFFFFF;
B = 32'hFFFFFFFF;

#100;

// Test with minimum values
A = 32'h80000000; // -2^31
B = 32'h80000000; // -2^31

#100;

// Test overflow in addition
ALU_operation =4'b0000;//add
A = 32'h7FFFFFFF; // 2^31 - 1
B = 32'h00000001; // 1

#100;

// Test underflow in subtraction
ALU_operation =4'b0001;//subtract
A = 32'h80000000; // -2^31
B = 32'h00000001; // 1

#100;

end
endmodule

```

2.设计 Register Files 模块

2.1 建立 Regs.v 源文件，代码如下所示：

```

`timescale 1ns / 1ps

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

);

integer i;

reg [31:0] register [31:0];//寄存器组

```

```
reg [31:0] rs1_d ;//两个中间寄存器
reg [31:0] rs2_d ;
```

```
initial begin
    register[0] = 0; //0 寄存器始终为0
end

always @(posedge clk or posedge rst or posedge RegWrite) begin
    if (rst) begin //如果 rst 信号为 1, 全部初始化为 0
        rs1_d <= 0 ;
        rs2_d <= 0 ;
        for (i=1;i<31;i=i+1)begin
            register[i]<=0;
        end
    end

    else if (RegWrite) begin //如果 regwrite 信号为 1, 则执行写操作
        if(Wt_addr != 0)
            register[Wt_addr]<=Wt_data;
        end

        else begin //否则执行读操作
            rs1_d <= register[Rs1_addr] ;
            rs2_d <= register[Rs2_addr] ;
        end
    end

end

assign Rs1_data=rs1_d;
assign Rs2_data=rs2_d;

assign Reg00 = register [0];
assign Reg01 = register [1];
assign Reg02 = register [2];
assign Reg03 = register [3];
assign Reg04 = register [4];
assign Reg05 = register [5];
assign Reg06 = register [6];
assign Reg07 = register [7];
assign Reg08 = register [8];
assign Reg09 = register [9];
assign Reg10 = register [10];
assign Reg11 = register [11];
assign Reg12 = register [12];
assign Reg13 = register [13];
assign Reg14 = register [14];
assign Reg15 = register [15];
assign Reg16 = register [16];
assign Reg17 = register [17];
assign Reg18 = register [18];
```

```

assign Reg19 = register [19];
assign Reg20 = register [20];
assign Reg21 = register [21];
assign Reg22 = register [22];
assign Reg23 = register [23];
assign Reg24 = register [24];
assign Reg25 = register [25];
assign Reg26 = register [26];
assign Reg27 = register [27];
assign Reg28 = register [28];
assign Reg29 = register [29];
assign Reg30 = register [30];
assign Reg31 = register [31];
endmodule

```

2.2 建立 Regs 模块的仿真代码 Regs_tb.v,具体代码如下所示:

```

`timescale 1ns / 1ns

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;

    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)
    );

    always #10 clk = ~clk;

```



```

initial begin
    //reset operation

    clk = 0;

    rst = 1;

    RegWrite = 0;

    Wt_data = 0;

    Wt_addr = 0;

    Rs1_addr = 0;

    Rs2_addr = 0;

    #100//write operation

    rst = 0;

    RegWrite = 1;

    Wt_addr = 5'b00101;

    Wt_data = 32'h5a5a5a5a;

    #50

    Wt_addr = 5'b01010;

    Wt_data = 32'h5a5a5a5a;

    #50//read operation

    RegWrite = 0;

    Rs1_addr = 5'b00101;

    Rs2_addr = 5'b01010;

    #50//boundary operation

    RegWrite = 1 ;

    Wt_addr = 5'b00000;

    Wt_data = 32'h11111111;

    #50

    Wt_addr = 5'b11111;

    Wt_data = 32'h22222222;

    #50

    Rs1_addr = 5'b00000;

    Rs2_addr = 5'b11111;

    #100//again reset operation

    rst = 1;

    #50

    rst = 0;

    #100 $stop();

end

endmodule

```

3.FSM 模块的设计

3.1 建立 FSM 的源代码 TruthEvaluator.v,具体代码如下:

```

`timescale 1ns / 1ps

module TruthEvaluator(

```

```

    input clk,
    input in,
    output out
);
//4 states need 2 bits
reg [1:0] curr_state ;
reg [1:0] next_state ;
// State definition
localparam
    Q1 = 2'b00,
    Q2 = 2'b01,
    Q3 = 2'b10,
    Q4 = 2'b11;
//4 states
initial begin
    curr_state = Q1;
    next_state = Q1;
end
// First segment: state transfer
always @(posedge clk) begin
    curr_state <= next_state ;
end
// Second segment: transfer condition
always @(*) begin
    case(curr_state)
        Q1: next_state = (in == 1'b0) ? Q2 : Q1;
        Q2: next_state = (in == 1'b0) ? Q3 : Q1;
        Q3: next_state = (in == 1'b0) ? Q4 : Q2;
        Q4: next_state = (in == 1'b0) ? Q4 : Q3;
        default: next_state = Q1;
    endcase
end
// Third segment: output
assign out = (Q1 == curr_state) || (Q2 == curr_state);
endmodule

```

3.2 对上述源文件建立仿真代码 TruthEvaluator_tb.v, 具体如下:

```

`timescale 1ns / 1ps

module TruthEvaluator_tb;

    reg clk;

    reg in;

    wire out;

```

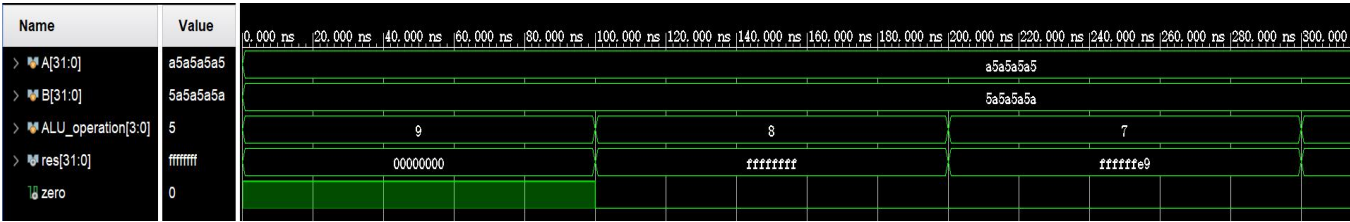
```
TruthEvaluator m0(
    .clk(clk),
    .in(in),
    .out(out)
);
always #10 clk=~clk;
initial begin
    clk=0;
    in=0;
    #100;

    in=1;
    #100;
    in=0;
    #100;
    in=1;
    #100;
    $stop;
end
endmodule
```

二、实验结果与分析

1.ALU 的仿真结果及分析:

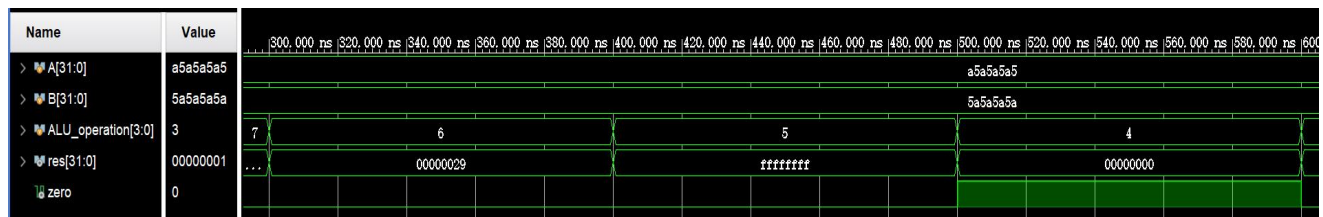
首先设置 A=32'h a5a5a5a5, B=32'h5a5a5a5a .



当 ALU_operation 为 9 时, ALU 执行 AND 操作。可以看到, 此时 A 和 B 按位与操作的结果应为 0, 恰好是 res 寄存器的值,且 zero 输出变成 1, 因此符合预期.

当 ALU_operation 为 8 时, ALU 执行 OR 操作。可以看到, 此时 A 和 B 按位或操作的结果应为 32'h ffffffff, 恰好是 res 寄存器的值, 因此符合预期.

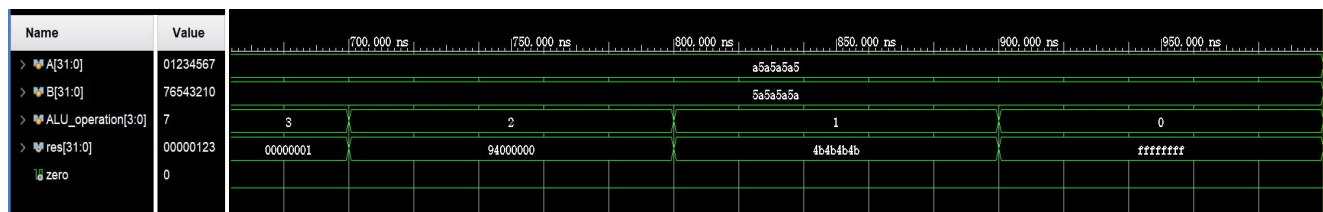
当 ALU_operation 为 7 时, ALU 执行 SRA(算术右移)操作。可以看到,B 的低 5 位是 5'b11010, 即需要将 A 的数据算术右移 26 位, 而 A 的最高位是 1, 所以最高位补 1, 因此预期结果应该是 32'h fffffff9,恰好是 res 寄存器的值, 因此符合预期.



当 ALU_operation 为 6 时，ALU 执行 SRL（逻辑右移）操作。同理需要将 A 的数据逻辑右移 26 位，最高位补 0，因此预期结果应该是 32'h 00000029,恰好是 res 寄存器的值，因此符合预期。

当 ALU_operation 为 5 时，ALU 执行 XOR 操作。可以看到，此时 A 和 B 按位异或操作的结果应为 32'h ffffffff，恰好是 res 寄存器的值，因此符合预期。

当 ALU_operation 为 4 时，ALU 执行 SLTU（无符号数比较大小）操作。显然 A 的值要大于 B 的值，因此预期结果为 0，且 zero 输出变成 1，符合实验结果。



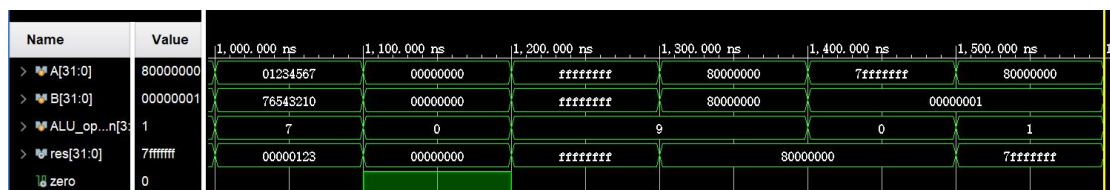
当 ALU_operation 为 3 时，ALU 执行 SLT（有符号数比较大小）操作。显然 A 是一个负数，B 是一个正数，A 要小于 B，因此预期结果为 1，符合实验结果。

当 ALU_operation 为 2 时，ALU 执行 SLL 操作。A 左移 5 位的预期结果是 32'h 94000000,与实验结果相同。

当 ALU_operation 为 1 时，ALU 执行 SUB 操作。A-B 的理论值是 32'h 4b4b4b4b,与实验结果相同。

当 ALU_operation 为 0 时，ALU 执行 ADD 操作。A+B 的理论值是 32'h ffffffff,与实验结果相同。

下面进行边界条件测试：



1100ns~1200ns:全 0 情况测试，显然 0+0 的结果还是 0，符合预期。

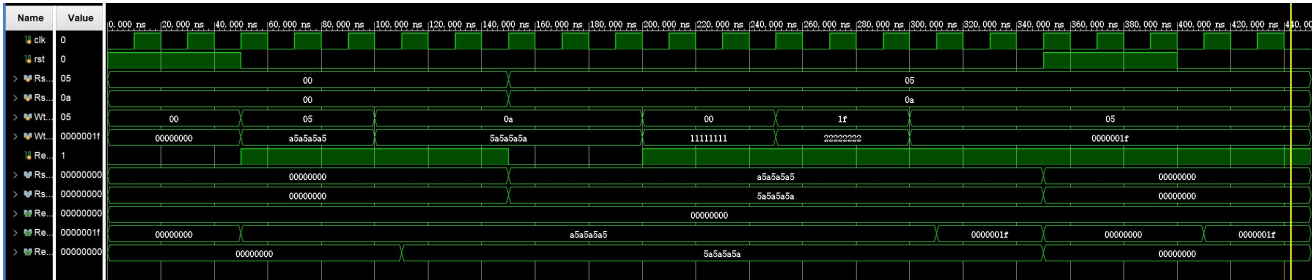
1200ns~1300ns:全 1 情况测试，显然 AND 的结果还是全 1，符合预期。

1300ns~1400ns:最小值（负值）测试，显然 AND 的结果还是 64'h 80000000,符合预期。

1400ns~1500ns:加法溢出情况检测：显然最终结果还是 64'h 80000000,符合预期。

1500ns~1600ns:减法溢出情况检测：显然最终结果是 64'h 7ffffff,符合预期。

2.Regis 的仿真结果及分析:



0-50ns:检验 rst 功能是否正常。

可以看到当 rst 为 1 时，所有寄存器的值均为 0，与预期相合。

50ns-150ns:检验 write 功能是否正常。

在 50ns-100ns 期间，被写入的地址 5，被写入的值是 64'h a5a5a5a5,写操作完成后可以看到 5 号寄存器里的值从 0 变成了 64'h a5a5a5a5，与预期相合。

在 100ns-150ns，被写入的地址是 10，被写入的值是 64'h 5a5a5a5a,写操作完成后可以看到 5 号寄存器里的值从 0 变成了 64'h 5a5a5a5a，与预期相合。

150ns-200ns:检验 read 功能是否正常。

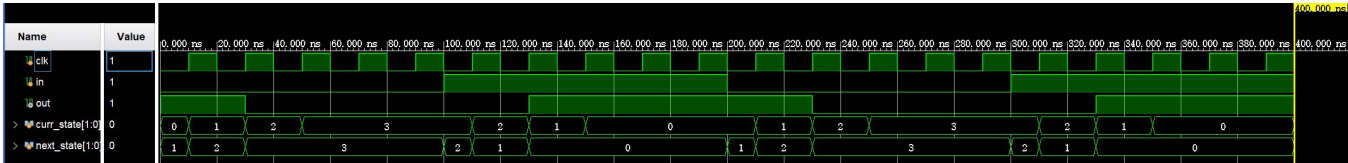
两个读地址分别是 5 和 10,可以看到 Rs1 和 Rs2 寄存器读入的数值恰好是 Reg5 和 Reg10 中储存的值，因此与预期相合。

200ns-350ns: 0 号寄存器以及边界条件的检验。

200ns-250ns 期间，将一个非零值写入 0 号寄存器，但结果 0 号寄存器的值始终保持为 0 不改变，与预期符合。

250ns-350ns:分别测试了写入地址为全 1 以及写入数据为全 1 的情况，发现寄存器堆都正常运行。

3.FSM 的仿真结果及分析:



0-100ns: 输入为 0，则 FSM 的状态不断往“更加怀疑”的方向转变。且 30ns-100ns 状态进入“可疑”、“不可信”，此时 FSM 给出的输出从 1（可信）变成了 0（不可信）。

100ns-200ns: 输入为 1，则 FSM 的状态不断往“更加相信”的方向转变。且 130ns-200ns 状态进入“非常可信”、“可信”，此时 FSM 给出的输出从 0（不可信）变成了 1（可信）。

200ns-400ns 又重复测试了上述过程，结果依然和前 200s 相同，因此与预期符合。

三、讨论、心得

这次实验内容总体比较简单，我觉得我学到的最有用的就是有限状态机的代码实现模板。主要还是复习了 verilog 的具体语法，慢慢开始找回写硬件代码的感觉。主要问题还是对硬件代码的书写不熟悉，这是因为上学期我对计算机逻辑掌握的就不是很好，希望能通过努力弥补一下。

思考题

1.关于\$signed 的思考:

我通过查看 Verilog 标准手册并参阅了 csdn 的相关文章，解决了这个问题，链接如下：

https://blog.csdn.net/kuan_/article/details/132677191

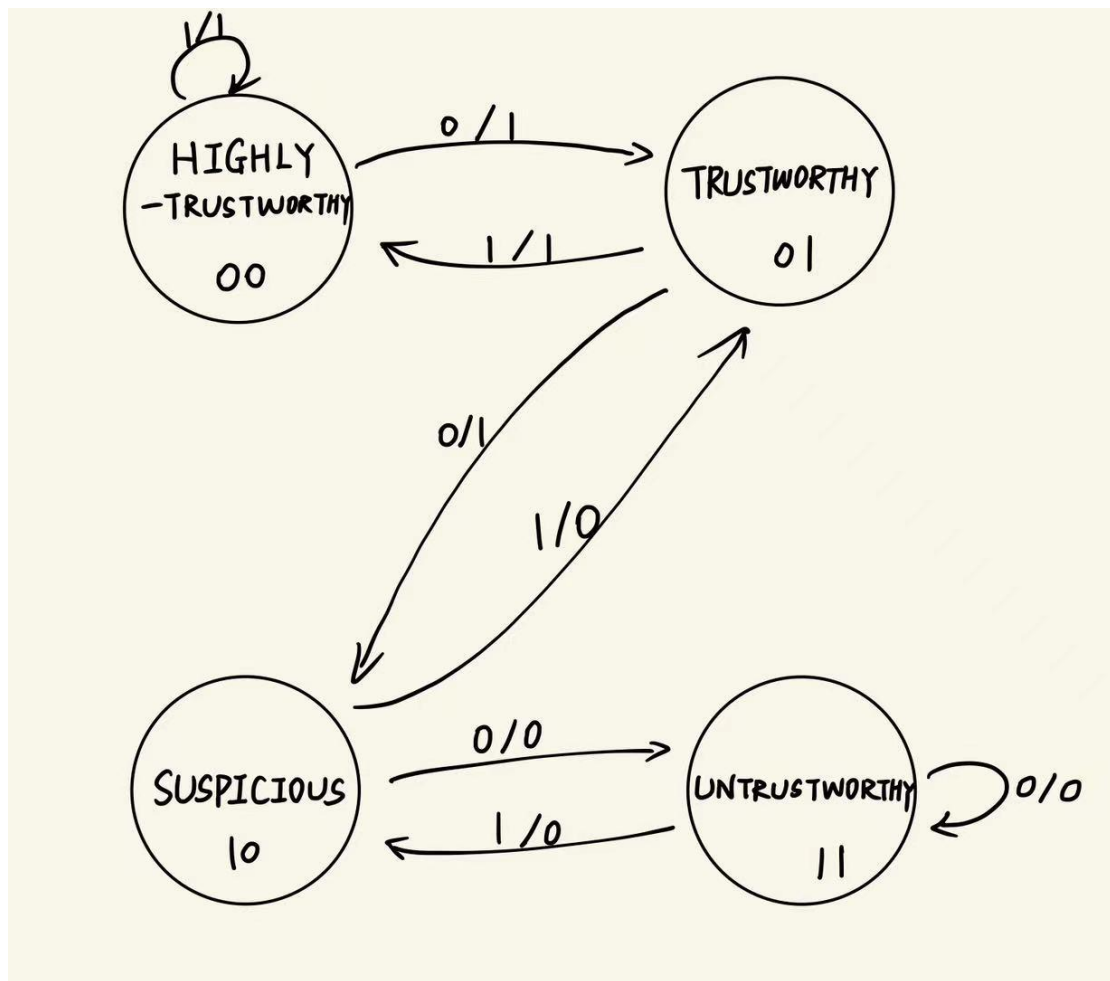
<http://staff.ustc.edu.cn/~songch/download/IEEE.1364-2005.pdf>

我认为引起该错误的原因是：\$signed 本身的作用是：在表达式求值时临时将操作数视为有符号数，或者说对原数进行符号扩展，而不会改变操作数本身的类型。而?:操作符冒号前后是并列的操作，因此

```
assign res = (...) ? ... : (ALU_operation == 4'd7) ? ($signed(A) >>> $signed(B)) : ...;
```

这里的 A、B 依然被当成了无符号数处理。

2.有限状态机的状态转移示意图



浙江大学实验报告

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称： IP core 测试框架搭建

学生姓名： 汪珉凯 学号： 330283200401240019 同组学生姓名： 赵桢、马扬松

实验地点： 紫金港东四 509 室 实验日期： 2024 年 3 月 14 日

一、操作方法与实验步骤

0.新建工程命名为 OExp02-IP2SOC，并将已经给出的源文件和 IP 核导入到当前工程目录下

1.设计顶层模块，根据图示连线，CSSTE.v 的具体代码如下：

```
`timescale 1ns / 1ps

module CSSTE(
    input        clk_100mhz,
    input        RSTN,
    input  [3:0]  BTN_y,
    input  [15:0] SW,
    output [3:0]  Blue,
    output [3:0]  Green,
    output [3:0]  Red,
    output        HSYNC,
    output        VSYNC,
    output [15:0] LED_out,
    output [7:0]  AN,
    output [7:0]  segment
);
//u1
wire MemRW_0 ;
wire [31:0] Addr_out_0, Data_out_0, PC_out_0;
//u2
wire [31:0] spo_0;
//u3
wire [31:0] douta_0;
//u4
wire data_ram_we_0, GPIOf0000000_we_0, GPIOe0000000_we_0, counter_we_0;
wire [9:0] ram_addr_0;
wire [31:0] Cpu_data4bus_0 ,ram_data_in_0 ,Peripheral_in_0;
//u5
```

```

wire [7:0] point_out_0,LE_out_0;

wire [31:0] Disp_num_0;

//u6

//u7

wire [1:0] counter_set_0;

//u8

wire Clk_CPU_0;

wire [31:0] clkdiv_0;

//u9

wire rst_0;

wire [3:0] BTN_OK_0;

wire [15:0] SW_OK_0;

//u10

wire counter0_OUT_0,counter1_OUT_0,counter2_OUT_0;

wire [31:0] counter_out_0;

SCPU U1(
    .MIO_ready(),
    .clk(Clk_CPU_0),
    .rst(rst_0),
    .Data_in(Cpu_data4bus_0), //数据输入总线
    .inst_in(spo_0), //指令输入总线
    .CPU_MIO(), // Not used
    .MemRW(MemRW_0), //存储器读写控制
    .Addr_out(Addr_out_0), //数据空间访问地址
    .Data_out(Data_out_0), //数据输出总线
    .PC_out(PC_out_0) //程序空间访问指针
);

ROM_mine U2 (
    .a(PC_out_0[11:2]),
    .spo(spo_0)
);

RAM_mine U3 ( .clka(~clk_100mhz), // 存储器时钟，与 CPU 反向
    .wea(data_ram_we_0), // 存储器读写，来自 MIO_BUS
    .addra(ram_addr_0), // 地址线，来自 MIO_BUS
    .dina(ram_data_in_0), // 输入数据线，来自 MIO_BUS
    .douta(douta_0) // 输出数据线，来自 MIO_BUS
);

MIO_BUS U4(
    . clk(clk_100mhz),
    . rst(rst_0),
    . BTN(BTN_OK_0),
    . SW(SW_OK_0),
    . mem_w(MemRW_0),

```



```

    .Cpu_data2bus(Data_out_0), //data from CPU
    .addr_bus(Addr_out_0), //addr from CPU
    .ram_data_out(douta_0),
    .led_out(LED_out),
    .counter_out(counter_out_0),
    .counter0_out(counter0_OUT_0),
    .counter1_out(counter1_OUT_0),
    .counter2_out(counter2_OUT_0),
    .Cpu_data4bus(Cpu_data4bus_0), //write to CPU
    .ram_data_in(ram_data_in_0), //from CPU write to Memory
    .ram_addr(ram_addr_0), //Memory Address signals
    .data_ram_we(data_ram_we_0),
    .GPIOf0000000_we(GPIOf0000000_we_0), // GPIOffffff00_we
    .GPIOe0000000_we(GPIOe0000000_we_0), // GPIOfffffe00_we
    .counter_we(counter_we_0), //计数器
    .Peripheral_in(Peripheral_in_0) //送外部设备总线
);

Multi_8CH32 U5(
    .clk(~Clk_CPU_0), //io_clk, 同步 CPU
    .rst(rst_0),
    .EN(GPIOe0000000_we_0), //!=1, 通道 0 显示
    .Test(SW_OK_0[7:5]), //通道选择 SW[7:5]
    .point_in({clkdiv_0[31:0],clkdiv_0[31:0]}), //针对 8 个显示通道各 8 个小数点
    .LES(64'b0),
    .Data0(Peripheral_in_0),
    .data1({2'b0,PC_out_0[31:2]}),
    .data2(spo_0),
    .data3(counter_out_0),
    .data4(Addr_out_0),
    .data5(Data_out_0),
    .data6(Cpu_data4bus_0),
    .data7(PC_out_0),
    .point_out(point_out_0), //小数点输出
    .LE_out(LE_out_0), //闪烁控制输出
    .Disp_num(Disp_num_0) //接入 7 段显示器
);

Seg7_Dev U6(
    .disp_num(Disp_num_0),
    .point(point_out_0),
    .les(LE_out_0),
    .scan(clkdiv_0[18:16]),
    .AN(An),
    .segment(segment)
);

```

```
SPIO U7(  
  .clk(~Clk_CPU_0),  
  .rst(rst_0),  
  .Start(clkdiv_0[20]),  
  .EN(GPIO00000000_we_0),  
  .P_Data(Peripheral_in_0),  
  .counter_set(counter_set_0),  
  .LED_out(LED_out),  
  .led_clk(),  
  .led_sout(),  
  .led_clrn(),  
  .LED_PEN(),  
  .GPIO00()  
);
```

```
clk_div U8(  
  .clk(clk_100mhz),  
  .rst(rst_0),  
  .SW2(SW_OK_0[2]),  
  .SW8(SW_OK_0[8]),  
  .STEP(SW_OK_0[10]),  
  .clkdiv(clkdiv_0),  
  .Clk_CPU(Clk_CPU_0)  
);
```

```
SAnti_jitter U9(  
  .clk(clk_100mhz),  
  .RSTN(RSTN),  
  .readn(),  
  .Key_y(BTN_y),  
  .SW(SW),  
  .Key_x(),  
  .Key_out(),  
  .Key_ready(),  
  .pulse_out(),  
  .BTN_OK(BTN_OK_0),  
  .SW_OK(SW_OK_0),  
  .CR(),  
  .rst(rst_0)  
);
```

```
Counter_x U10(  
  .clk(~Clk_CPU_0),  
  .rst(rst_0),  
  .clk0(clkdiv_0[6]),  
  .clk1(clkdiv_0[9]),  
  .clk2(clkdiv_0[11]),
```

```

.counter_we(counter_we_0),
.counter_val(Peripheral_in_0),
.counter_ch(counter_set_0),
.counter0_OUT(counter0_OUT_0),
.counter1_OUT(counter1_OUT_0),
.counter2_OUT(counter2_OUT_0),
.counter_out(counter_out_0)
);
VGA U11(
.clk_25m(clkdiv_0[1]),
.clk_100m(clk_100mhz),
.rst(rst_0),
.pc(PC_out_0),
.inst(spo_0),
.alu_res(Addr_out_0),
.mem_wen(MemRW_0),
.dmem_o_data(douta_0),
.dmem_i_data(ram_data_in_0),
.dmem_addr(Addr_out_0),
.hs(HSYNC),
.vs(VSYNC),
.vga_r(Red),
.vga_g(Green),
.vga_b(Blue)
);
endmodule

```

2.生成 ROM、RAM 的 IP 核。

2.1 在 VIVADO 集成菜单上从 PROJECT MANAGER 选择 IP Catalog

2.2 点击搜索栏，输入 memory generator, 选择 Distributed Memory Generator

2.3 分别选择生成 ROM 和 RAM 的生成，进行核参数设置,并关联本地文件 I_mem、D_mem。

最终结果如图所示：

The image shows two side-by-side screenshots of the Vivado IP configuration interface for memory blocks.

Left Screenshot (ROM_block):

- Component Name:** ROM_block
- memory config** tab is selected.
- Options:**
 - Depth: 1024 (Range: [16 - 65536])
 - Data Width: 32 (Range: [1 - 1024])
- Memory Type:**
 - ☒ ROM
 - ☐ Single Port RAM
 - ☐ Simple Dual Port RAM
 - ☐ Dual Port RAM

Right Screenshot (RAM_block):

- Component Name:** RAM_block
- Basic** tab is selected.
- Memory Size:**
 - Write Width: 32 (Range: 1 to 4608 (bits))
 - Read Width: 32
 - Write Depth: 1024 (Range: 2 to 1048576)
 - Read Depth: 1024
- Operating Mode:** Write First
- Enable Port Type:** Always Enabled
- Port A Optional Output Registers:**
 - ☐ Primitives Output Register
 - ☐ Core Output Register
 - ☐ SoftECC Input Register
 - ☐ REGCEA Pin

2.4 对 I_mem 中程序的思考：

I_mem 中的程序实现了斐波那契数列的计算。程序首先使用 addi 指令将寄存器 x1 的值初始化为 1，随后使用 slt 和一系列的 add 指令在循环中计算斐波那契数列的值，并将结果存储在寄存器 x2 到 x31 中。每个寄存器中的值都是前两个寄存器值的和，符合斐波那契数列的定义，即每个数字是前两个数字的和。

3.对 VGA 模块的修改

(1) 修改修改附件 VGADisplay.v 中的文件路径，和 font_8x16.mem，vga_debugger.mem 文件的路径匹配。

```
initial
$readmemh("E://CO_LABS//OExp02-IP2SOC//OExp02-IP2SOC.srcs//sources_1//imports//OExp02-IP2SOC//vga_debugger.mem", display_data);
```

```
initial
$readmemh("E://CO_LABS//OExp02-IP2SOC//OExp02-IP2SOC.srcs//sources_1//imports//OExp02-IP2SOC//font_8x16.mem", fonts_data);
```

(2) 对 VGA 端口的修改

修改 VGA.v 中 module VGA 的端口描述，将增加的输入接到 VGA 模块的 vga_debugger 实例，修改完后的代码如下：

```
`timescale 1ns / 1ps
module VGA(
    input wire clk_25m,
    input wire clk_100m,
    input wire rst,
    input wire [31:0] pc,
    input wire [31:0] inst,
    input wire [31:0] alu_res,
    input wire mem_wen,
    input wire [31:0] dmem_o_data,
    input wire [31:0] dmem_i_data,
    input wire [31:0] dmem_addr,
    input wire [4:0] rs1,
    input wire [31:0] rs1_val,
    input wire [4:0] rs2,
    input wire [31:0] rs2_val,
    input wire [4:0] rd,
    input wire [31:0] reg_i_data,
    input wire reg_wen,
    input wire is_imm,
    input wire is_auihc,
    input wire is_lui,
    input wire [31:0] imm,
    input wire [31:0] a_val,
    input wire [31:0] b_val,
```



```
input wire [3:0] alu_ctrl,
input wire [2:0] cmp_ctrl,
input wire cmp_res,
input wire is_branch,
input wire is_jal,
input wire is_jalr,
input wire do_branch,
input wire [31:0] pc_branch,
input wire mem_ren,
input wire csr_wen,
input wire [11:0] csr_ind,
input wire [1:0] csr_ctrl,
input wire [31:0] csr_r_data,
input wire [31:0] x0,
input wire [31:0] ra,
input wire [31:0] sp,
input wire [31:0] gp,
input wire [31:0] tp,
input wire [31:0] t0,
input wire [31:0] t1,
input wire [31:0] t2,
input wire [31:0] s0,
input wire [31:0] s1,
input wire [31:0] a0,
input wire [31:0] a1,
input wire [31:0] a2,
input wire [31:0] a3,
input wire [31:0] a4,
input wire [31:0] a5,
input wire [31:0] a6,
input wire [31:0] a7,
input wire [31:0] s2,
input wire [31:0] s3,
input wire [31:0] s4,
input wire [31:0] s5,
input wire [31:0] s6,
input wire [31:0] s7,
input wire [31:0] s8,
input wire [31:0] s9,
input wire [31:0] s10,
input wire [31:0] s11,
input wire [31:0] t3,
input wire [31:0] t4,
input wire [31:0] t5,
```

```

input wire [31:0] t6,
input wire [31:0] mstatus_o,
input wire [31:0] mcause_o,
input wire [31:0] mepc_o,
input wire [31:0] mtval_o,
input wire [31:0] mtvec_o,
input wire [31:0] mie_o,
input wire [31:0] mip_o,

    output wire hs,
    output wire vs,
    output wire [3:0] vga_r,
    output wire [3:0] vga_g,
    output wire [3:0] vga_b
);
wire [9:0] vga_x;
wire [8:0] vga_y;
wire video_on;
VgaController vga_controller(
    .clk      (clk_25m      ),
    .rst      (rst         ),
    .vga_x    (vga_x       ),
    .vga_y    (vga_y       ),
    .hs       (hs          ),
    .vs       (vs          ),
    .video_on (video_on    )
);
wire display_wen;
wire [11:0] display_w_addr;
wire [7:0] display_w_data;
VgaDisplay vga_display(
    .clk      (clk_100m    ),
    .video_on (video_on    ),
    .vga_x    (vga_x       ),
    .vga_y    (vga_y       ),
    .vga_r    (vga_r       ),
    .vga_g    (vga_g       ),
    .vga_b    (vga_b       ),
    .wen      (display_wen ),
    .w_addr   (display_w_addr),
    .w_data   (display_w_data)
);
VgaDebugger vga_debugger(
    .clk      (clk_100m    ),
    .display_wen (display_wen ),

```

```

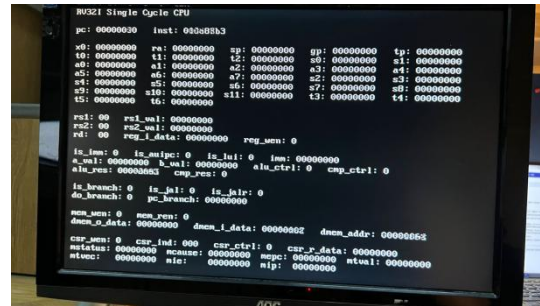
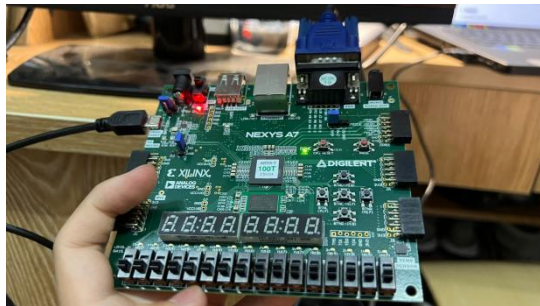
.display_w_addr(display_w_addr),
.display_w_data(display_w_data),
.pc          (pc          ),
.inst        (inst        ),
.rs1         ( rs1        ),
.rs1_val     ( rs1_val    ),
.rs2         ( rs2        ),
.rs2_val     ( rs2_val    ),
.rd          ( rd         ),
.reg_i_data  ( reg_i_data ),
.reg_wen     ( reg_wen    ),
.is_imm      ( is_imm     ),
.is_auiopc   ( is_auiopc  ),
.is_lui      ( is_lui     ),
.imm         ( imm        ),
.a_val       ( a_val      ),
.b_val       ( b_val      ),
.alu_ctrl    ( alu_ctrl   ),
.cmp_ctrl    ( cmp_ctrl   ),
.alu_res     (alu_res     ),
.cmp_res     ( cmp_res    ),
.is_branch   ( is_branch  ),
.is_jal      ( is_jal     ),
.is_jalr     ( is_jalr    ),
.do_branch   ( do_branch  ),
.pc_branch   ( pc_branch  ),
.mem_wen     (mem_wen     ),
.mem_ren     ( mem_ren    ),
.dmem_o_data (dmem_o_data ),
.dmem_i_data (dmem_i_data ),
.dmem_addr   (dmem_addr   ),
.csr_wen     ( csr_wen    ),
.csr_ind     ( csr_ind    ),
.csr_ctrl    ( csr_ctrl   ),
.csr_r_data  ( csr_r_data ),
.x0          ( x0         ),
.ra          ( ra         ),
.sp          ( sp         ),
.gp          ( gp         ),
.tp          ( tp         ),
.t0          ( t0         ),
.t1          ( t1         ),
.t2          ( t2         ),
.s0          ( s0         ),

```

```
.s1      (    s1      ),
.a0      (    a0      ),
.a1      (    a1      ),
.a2      (    a2      ),
.a3      (    a3      ),
.a4      (    a4      ),
.a5      (    a5      ),
.a6      (    a6      ),
.a7      (    a7      ),
.s2      (    s2      ),
.s3      (    s3      ),
.s4      (    s4      ),
.s5      (    s5      ),
.s6      (    s6      ),
.s7      (    s7      ),
.s8      (    s8      ),
.s9      (    s9      ),
.s10     (    s10     ),
.s11     (    s11     ),
.t3      (    t3      ),
.t4      (    t4      ),
.t5      (    t5      ),
.t6      (    t6      ),
.mstatus_o ( mstatus_o      ),
.mcause_o  ( mcause_o      ),
.mepc_o    ( mepc_o      ),
.mtval_o   ( mtval_o      ),
.mtvec_o   ( mtvec_o      ),
.mie_o     ( mie_o      ),
.mip_o     ( mip_o      )

);
endmodule
```

二、实验结果与分析



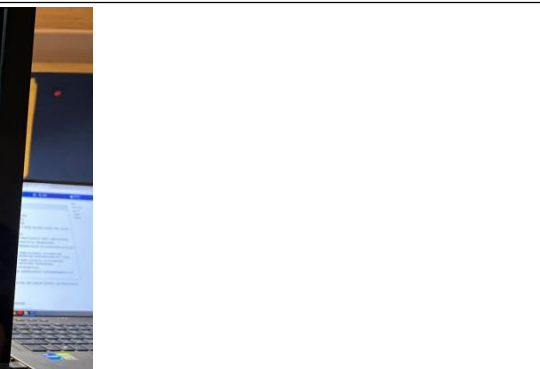
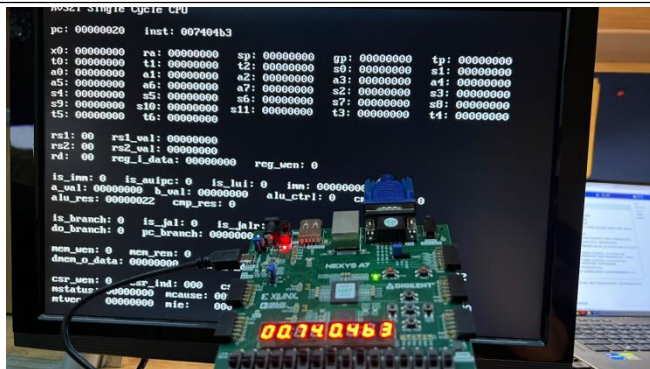
上述情况是所有开关均为 0，可以看到 VGA 显示屏上成功显示出了当前的各类数据。可以看到，由于 PC 在高速跳转，PC 的值保持模糊不清，符合预期。



令 SW[8]=1，可以看到显示屏上 PC 停止跳动，符合预期。



SW[10] 用于手动时钟模式。此时拨一下 SW[10]（即从 0 拨到 1）时钟增加一个周期，可以看到 PC 的值从'h14 变成了'h18,指令地址增加 4，符合预期。



令 SW[7:5]=010: 显示 ROM 指令输出 Inst_in，即我们正在执行的指令，可以看到七段数码

管显示出 32'h007404b3,与显示屏上 inst 的值相同。



SW[7:5]=100: 显示 CPU 数据存储地址 addr_bus。可以看到七段数码管显示出 32'h00000022,与显示屏上 dmem_addr 的值相同,符合预期。



验证斐波那契数列。令 SW[8]、SW[2]=1,SW[7:5]=101,即进入手动单步时钟模式,七段数码管成功显示 Cpu_data2bus 的值,即'h59 (F11='d89)



在上述基础上将 SW[10]拨成 1,即手动使 PC 跳转到下一条指令。可以看到,Cpu_data2bus 的值成功变成了'h90 (F12='d144)

三、讨论、心得

本次实验的难点在于准确无误地将每一条线都连对。虽然看起来简单,但实际我前前后后花了将近半个月的时间,才将本次实验的上板结果全部调试正确。希望以后在用到本次实验搭建的平台的时候不要出现问题。

浙江大学实验报告

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称： 乘除法器

学生姓名： 汪珉凯 学号： 330283200401240019 同组学生姓名： 赵桢、马扬松

实验地点： 紫金港东四 509 室 实验日期： 2024 年 3 月 21 日

一、操作方法与实验步骤

1. 乘法器模块的实现

1.1 建立源代码 multiplier.v，具体如下：

```
`timescale 1ns / 1ps
module multiplier(
    input        clk,      // 时钟信号
    input        start,    // 开始运算
    input [31:0]  A,        // 两个 32-bit 输入值
    input [31:0]  B,
    output reg    finish,   // 当结束计算时置 1，你可能需要将它改为 `output reg`
    output reg    [63:0] res // 64-bit 输出，你可能需要将它修改为 `output reg[63:0]`
);
    reg state; // 记录 multiplier 是不是正在进行运算
    reg[31:0] multiplicand; // 保存当前运算中的被乘数
    reg[4:0] cnt; // 记录当前计算已经经历了几个周期（运算与移位）

    wire[5:0] cnt_next = cnt + 6'b1; // 计数器的迭代
    wire [31:0] temp = res [63:32] + multiplicand; // 每一次加法的运算结果

    reg [31:0] temp_a; // 输入 A
    reg [31:0] temp_b; // 输入 B
    wire [31:0] temp_result; // 暂存结果
    reg sign = 0; // 记录当前运算的结果是否是负数

    initial begin // 将寄存器都初始化为 0
        res <= 0;
        state <= 0;
        finish <= 0;
        cnt <= 0;
```



```

        multiplicand <= 0;

    End

assign temp_result=~res+1;//负数取反最后的加 1

always @(posedge clk) begin

    if(~state && start ) begin//开始计算

        // Not Running

        sign <= (A[31] ^ B[31]); //判断正负

        if(A[31]==1)temp_a=~A+1; // 负数则取反

        else temp_a=A;

        if(B[31]==1)temp_b=~B+1; // 负数则取反

        else temp_b=B;

        multiplicand = temp_a;

        res[63:32] <=0 ;

        res[31:0]  <=temp_b ;

        state <= 1;

        finish <= 0;

        cnt <= 0;

    end

    else if(state) begin

        // Running

        // 处理“一次”运算与移位

        cnt <= cnt_next;

        if(res[0]==1)begin

            res[63:32] = temp;//高位相加

        end

        res<=res>>1;//右移

    end

    if(cnt==31) begin

        // 迭代结束 得到结果

        cnt <= 0;

        finish <= 1;

        state <= 0;

        if(sign==1)begin//若结果为负数，最后取反加 1

            res <= ~(res >> 1) + 1'b1;

        end else begin

            res <= res  >> 1;

        end

    end

end

endmodule

```

1.2 建立仿真代码 multiplier_tb.v, 具体代码如下:

```
`timescale 1ns / 1ps

// 定义 multiplier_tb 模块, 用于对乘法器进行仿真测试
module multiplier_tb;

    // 输入端口声明

    reg clk, start;        // 时钟信号和启动信号
    reg[31:0] A;           // 32 位输入值 A
    reg[31:0] B;           // 32 位输入值 B
    // 输出端口声明
    wire finish;           // 完成信号
    wire[63:0] res;        // 64 位输出结果

    // 实例化被测试的 multiplier 模块
    multiplier m0(.clk(clk), .start(start), .A(A), .B(B), .finish(finish), .res(res));

    // 初始化块, 用于设置仿真环境和激励信号
    initial begin

        // 设置 VCD 文件名以及要记录的变量
        $dumpfile("multiplier_signed.vcd");
        $dumpvars(0, multiplier_tb);

        // 初始化输入信号
        clk = 0;
        start = 0;

        // 第一组测试用例

        #10;                // 等待 10 个时间单位
        A = 32'd1;          // 设置输入 A 为正数 1
        B = 32'd0;          // 设置输入 B 为零
        #10 start = 1;      // 启动计算
        #10 start = 0;      // 停止计算
        #200;               // 等待 200 个时间单位, 观察结果

        // 第二组测试用例
        A = -32'd10;        // 设置输入 A 为负数 -10
        B = 32'd30;         // 设置输入 B 为正数 30
        #10 start = 1;      // 启动计算
        #10 start = 0;      // 停止计算
        #200;               // 等待 200 个时间单位, 观察结果

        // 第三组测试用例
        A = 32'd66;         // 设置输入 A 为正数 66
        B = 32'd23;         // 设置输入 B 为正数 23
        #10 start = 1;      // 启动计算
        #10 start = 0;      // 停止计算
        #200;               // 等待 200 个时间单位, 观察结果

        // 第四组测试用例
        A = -32'd6;         // 设置输入 A 为负数 -6
        B = 32'd30;         // 设置输入 B 为正数 30
        #10 start = 1;      // 启动计算
```

```

    #10 start = 0;      // 停止计算

    #300;              // 等待 300 个时间单位，观察结果

    $finish();         // 结束仿真

end

// 时钟生成

always begin

    #2 clk = ~clk;     // 每 2 个时间单位反转一次时钟信号

end

endmodule

```

2. 除法器模块的实现

2.1 建立除法器源代码 divider.v, 具体代码如下:

```

`timescale 1ns / 1ps

module divider(

    input clk,

    input rst,

    input start,

    input [31:0] dividend,

    input [31:0] divisor,

    output reg divide_zero,

    output reg finish,

    output reg [31:0] res,

    output reg [31:0] rem

);

// 内部变量定义

reg [63:0] dividend_temp;
reg [31:0] divisor_temp;
reg [5:0] count;
wire [31:0] temp;
wire [31:0] res_temp;
wire [31:0] rem_temp;

assign temp = dividend_temp[63:32] - divisor_temp;
assign res_temp = dividend_temp[31:0]; // 输出商
assign rem_temp = dividend_temp[63:32] >> 1; // 输出余数

always @(posedge clk or posedge rst) begin

    if (rst) begin

        // 复位所有输出和内部变量

        divide_zero <= 0;

        finish <= 0;

        res <= 0;

        rem <= 0;
    end
end

```

```

    dividend_temp <= 0;

    count <= 0;

end

else if (start) begin

    if (divisor == 0) begin

        // 如果除数为 0, 则设置 divide_zero 为高位, 结束计算

        divide_zero <= 1;

        finish <= 1;

    end

    else begin

        // 初始化计算

        divide_zero <= 0;

        finish <= 0;

        dividend_temp <= {32'b0,dividend}; // 扩展被除数以便于计算

        divisor_temp <= divisor;

        count <= 32; // 设置计数器为 32, 因为需要处理 32 位的数据

    end

end

else if (count >= 0 && ~finish) begin

    if(count==32)begin

        dividend_temp <= dividend_temp << 1;

    end

    // 执行除法操作

    if (temp[31] == 1) begin

        dividend_temp <= dividend_temp << 1;

    end

    else begin

        dividend_temp <= ( { temp,divident_temp[31:0] } << 1 ) + 1;

    end

    count <= count - 1; // 更新计数器

    if (count == 0) begin

        // 计算完成

        finish <= 1;

    end

end

res <= res_temp;

rem <= rem_temp;

end

endmodule

```

2.2 建立除法器的仿真激励代码 divider_tb.v, 具体代码如下:

```
`timescale 1ns / 1ps
module divider_tb();

    reg clk;
    reg rst;
    reg [31:0] dividend;
    reg [31:0] divisor;
    reg start;

    wire divide_zero;
    wire [31:0] res;
    wire [31:0] rem;
    wire finish;

    // 实例化被测模块
    divider u_div(
        .clk(clk),
        .rst(rst),
        .dividend(dividend),
        .divisor(divisor),
        .start(start),
        .divide_zero(divide_zero),
        .res(res),
        .rem(rem),
        .finish(finish)
    );

    // 生成时钟信号
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 时钟周期为 10ns
    end

    // 测试案例
    initial begin
        // 初始化测试环境
        rst = 1; start = 0; dividend = 0; divisor = 0;
        #10; // 等待一段时间以确保复位完成

        // 清除复位信号
        rst = 0;

        // 测试案例 1: 普通除法操作
        #10; dividend = 7; divisor = 2; start = 1;
        #10; start = 0; // 开始信号重置
        #400; // 等待额外的时间确保所有信号都已更新
    end
endmodule
```

```

// 测试案例 2: 除数为 0 的情况

#10; dividend = 50; divisor = 13; start = 1;

#10; start = 0; // 开始信号重置

#400;

//测试结束

$finish;

end

// 可选: 添加波形输出

initial begin

    $dumpfile("divider_tb.vcd");

    $dumpvars(0, divider_tb);

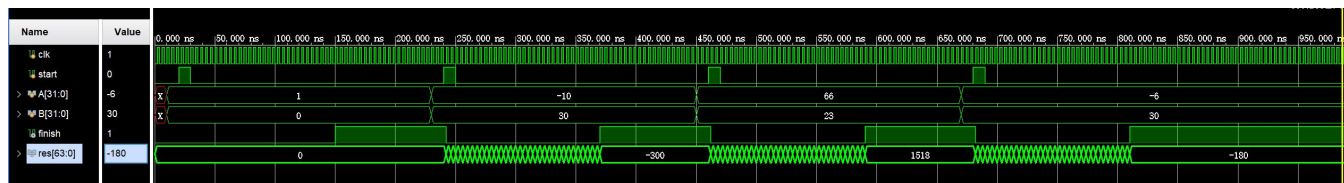
end

endmodule

```

二、实验结果与分析

1.有符号乘法器仿真结果



(仿真波形图中的数据都已经转化为 signed decimal 形式)

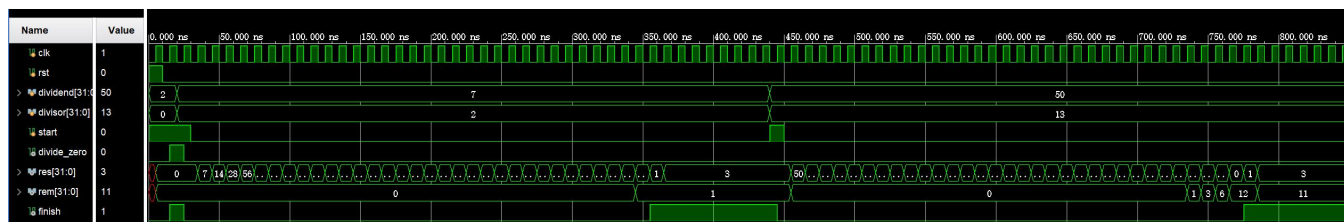
本次仿真可以被分为四个不同的阶段,进行了四次乘法。可以看到:

第一阶段:乘法器实现含 0 的乘法,结果保持为 0,符合预期

第二阶段和第四阶段:乘法器实现了正数和负数的乘法,结果能得到一个负数且正确,符合预期。

第三阶段:乘法器实现了两个正数的乘法,结果能得到一个正数且正确,符合预期。

2.无符号除法器仿真结果



(仿真波形图中的数据都已经转化为 unsigned decimal 形式)

本次仿真可以被分为四个不同的阶段,进行了四次无符号除法。可以看到:

第一阶段:验证 rst 的功能。可以看到,当 rst=1 时,所有寄存器中的值全部为 0,符合预期

第二阶段:验证除数为 0 的情况。可以看到 2/0 的情况下, divide_zero=1,符合预期。

第三阶段和第四阶段:除法器实现了两个无符号(正)数的除法,结果能得到正确的商和余,分别时: $7/2=3...1$; $50/13=3...11$,符合预期。

三、讨论、心得

本次实验总体难度并不大，特别的点在于这是我第一次写多周期的硬件模块，开始时感觉有点陌生，让我想到了原来设计有限状态机的过程。本次实验后，我对课本中不同版本的乘除法器实现都有了更深刻的认识。

思考题

1. $(x+y)+z = 1.0$; $x+(y+z) = 0.0$ ，两者的区别在于：

对于 $(x+y)+z$ 的计算：由于 x 和 y 是大数（其绝对值相等但符号相反），它们相加的结果理论上应该是 0。然后将 0 与 z （即 1.0）相加，得到的结果应该是 1.0。

对于 $x+(y+z)$ 的计算：首先计算 $y+z$ ，得到 $1.5e38 + 1.0$ 。由于 1.0 相对于 $1.5e38$ 来说非常小，它在加法操作中的影响几乎可以忽略不计，因此结果接近 $1.5e38$ 。然后将 x （即 $-1.5e38$ ）加到这个结果上，理论上应该得到 0。

2. 如果 $SOME_VALUE_0 := 0.1$, $SOME_VALUE_1 := 1000$ ，我将得到：100.0-1.4e(-12)。

如果 $SOME_VALUE_0 := 0.125$, $SOME_VALUE_1 := 800$ ，我将得到：100.0 整数。

也就是说，前者与 100 存在 $1.4e(-12)$ 的微小误差，后者则没有，原因在于：浮点数在计算机中的表示并不是永远完全精确的，特别是对于无法精确表示为二进制分数的数。比如小数 0.1，在二进制表示中是一个无限循环小数，因此不能被浮点数格式精确表示，这导致了累积误差；而 0.125 在二进制中可以精确表示（即 0.001），因此在进行多次加法操作时不会产生误差。