

《计算机组成与设计》

实验报告

姓名: 刘韬

学院: 竹可桢学院

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;
    end
endmodule
```

```

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:

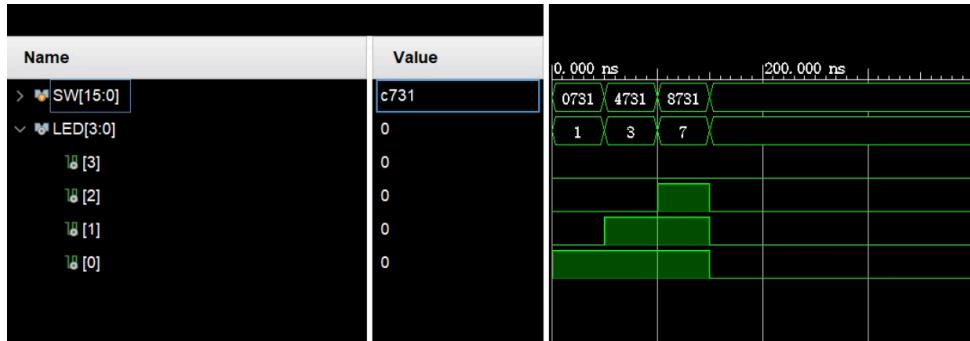


图 1: 多路选择器仿真波形图

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 LVCMS3 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]
set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMS3 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15
Sch=led[1]
set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMS3 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15
Sch=led[2]
set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMS3 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14

```

```

Sch=led[3]

# SW
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVC MOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15
Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVC MOS33 } [get_ports { SW[1] }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVC MOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14
Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVC MOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14
Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVC MOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14
Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVC MOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14
Sch=sw[5]
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVC MOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14
Sch=sw[6]
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVC MOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14
Sch=sw[7]
set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVC MOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVC MOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVC MOS33 } [get_ports { SW[10] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVC MOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14
Sch=sw[11]
set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVC MOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35
Sch=sw[12]
set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVC MOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14
Sch=sw[13]
set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVC MOS33 } [get_ports { SW[14] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVC MOS33 } [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：

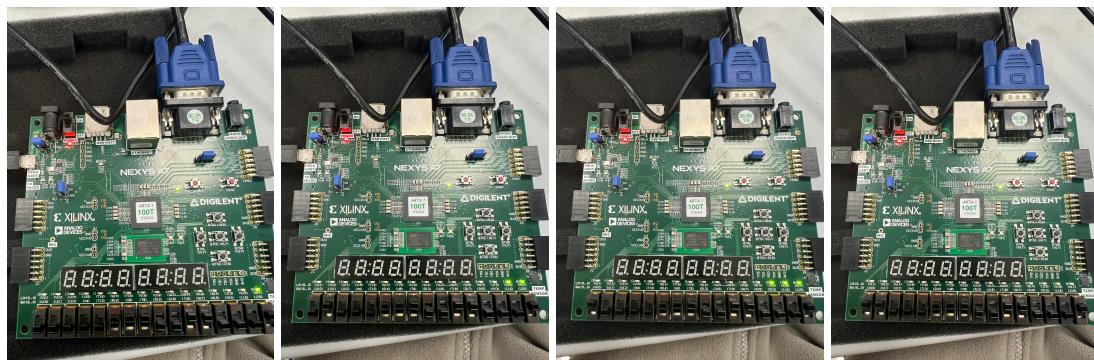


图 2: SW[15:14] = 00

图 3: SW[15:14] = 01

图 4: SW[15:14] = 10

图 5: SW[15:14] = 11

SW[11:0]=12b'011100110001, 分别第一, 第一二, 第一二三灯亮, 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'hFFFFFF;
B=32'h00000001:#100;
A=32'h00000001;
B=32'hFFFFFF:#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：

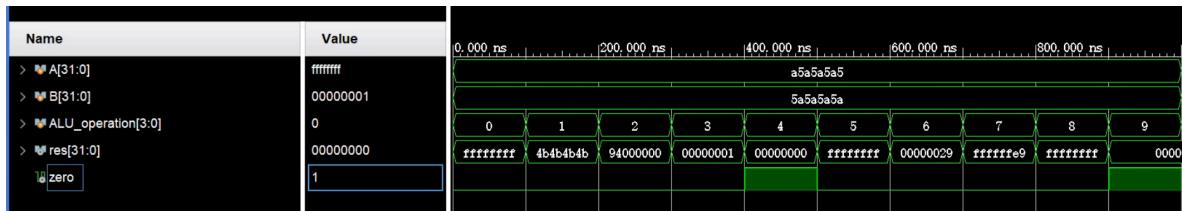


图 6：ALU 常规情况仿真波形图

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'hFFFFFF
- ▷ 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'hFFFFFF
- ▷ 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'hFFFFFE9, 算数右移，高位补符号位 1。

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

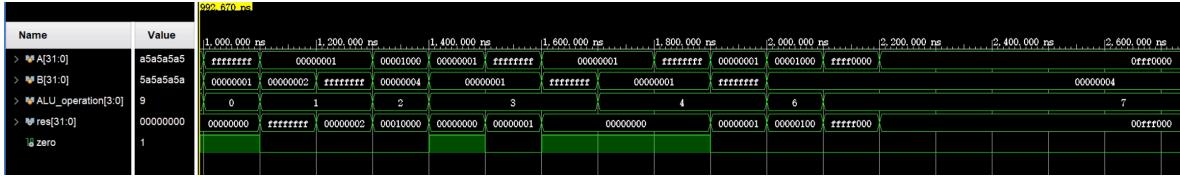


图 7: ALU 溢出情况仿真波形图

2. 边界情况下，我们测试了溢出的情况:

- ◊ ALU_operation = 4'b0000(ADD): A = 32'hFFFFFFFFF, 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'hFFFFFFFFF,
此时发生了溢出，结果为 -1。
- ◊ ALU_operation = 4'b0001(SUB): A = 32'h00000001, B = 32'hFFFFFFFFF, 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,
结果为 0。
- ◊ ALU_operation = 4'b0011(SLT): A = 32'hFFFFFFFFF, B = 32'h00000001, A < B = 32'h00000001,
结果为 1。
- ◊ ALU_operation = 4'b0011(SLT): A = 32'h00000001, B = 32'hFFFFFFFFF, A < B = 32'h00000000,
结果为 0。
- ◊ ALU_operation = 4'b0100(SLTU): A = 32'h00000001, B = 32'h00000001, A < B = 32'h00000000,
结果为 0。
- ◊ ALU_operation = 4'b0100(SLTU): A = 32'hFFFFFFFFF, B = 32'h00000001, A < B = 32'h00000000,
结果为 0。
- ◊ ALU_operation = 4'b0100(SLTU): A = 32'h00000001, B = 32'hFFFFFFFFF, A < B = 32'h00000001,
结果为 1。
- ◊ ALU_operation = 4'b0110(SRL): A = 32'h00001000, B = 32'h00000004, A >> B[4:0] =
32'h00000100, 将 A 逻辑右移 4 位，结果为 1。
- ◊ ALU_operation = 4'b0111(SRA): A = 32'hFFFF0000, B = 32'h00000004, A >>> B[4:0] =
32'hFFFF0000, 将 A 算术右移 4 位，高位补了 4 个 1，所以结果为 32'hFFFF0000。
- ◊ ALU_operation = 4'b0111(SRA): A = 32'h0FFF0000, B = 32'h00000004, A >>> B[4:0] =
32'h0000FFF0, 将 A 算术右移 4 位，高位补了 4 个 0，所以结果为 32'h00FFF000。

Part III 思考题

问题出在?:运算符。在二选一时如果有一路是 unsigned 类型，就会把另一路也转化为 unsigned。同时>>>运算的方式由结果类型 signed/unsigned 决定，而结果类型被决定的优先级是：

- 1.作为某些运算符的操作数时，由运算符决定
- 2.运算符不能决定的，与左操作数类型相同

在这里是第一种情况，由运算符决定。而(ALU_operation == 4'd7)是 unsigned 类型，所以尽管(\$signed(A) >>> \$signed(B))使用了类型转换也会被转化为 unsigned 类型，导致结果不符合预期。

参考链接：<https://www.zhihu.com/question/646666165>

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'h5a5a5a5a;
    #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:

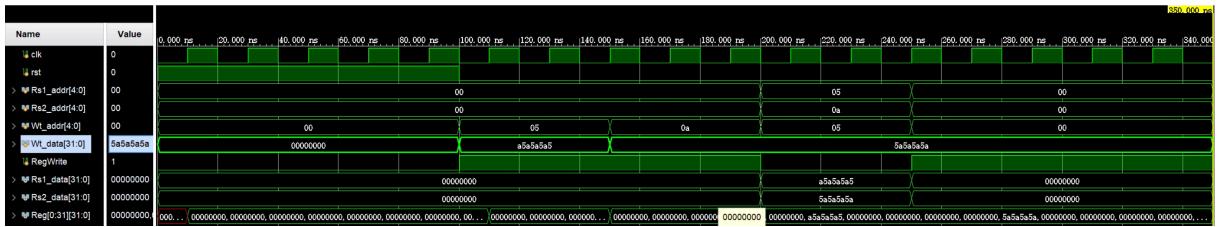


图 8: Regs 仿真波形图

在波形中我们测试的功能如下:

1. 0 - 100 ns : $\text{rst} = 1$, 初始化寄存器, 所有寄存器的值都为 0。
2. 100 - 150 ns : $\text{rst} = 0$, $\text{RegWrite} = 1$, $\text{Wt_addr} = 5'b00101$, $\text{Wt_data} = 32'h5a5a5a5a5$, 写入寄存器 5 的值为 $32'h5a5a5a5a5$ 。
3. 150 - 200 ns : 写入寄存器 10 的值为 $32'h5a5a5a5a5a$ 。
4. 200 - 250 ns : 查看写入的寄存器 5 和 10 的值, $\text{Rs1_addr} = 5'b00101$, $\text{Rs2_addr} = 5'b01010$, $\text{Rs1_data} = 32'h5a5a5a5a5$, $\text{Rs2_data} = 32'h5a5a5a5a5a$ 。
5. 250ns + : 查看并写入寄存器 0 的值, 可以看到寄存器 0 的值一直为 0, 无法进行写入。

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

```

测试波形如下：



图 9: TruthEvaluator 仿真波形图

状态说明:

- ◊ state[1:0]的值代表状态, 即 00 代表 HIGHLY_TRUSTWORTHY, 01 代表 TRUSTWORTHY, 10 代表 SUSPICIOUS, 11 代表 UNTRUSTWORTHY。
- ▷ trust_decision = 1 when state[1] = 0, 即 HIGHLY_TRUSTWORTHY 和 TRUSTWORTHY 状态下输出 1
- ▷ trust_decision = 0 when state[1] = 1, 即 SUSPICIOUS 和 UNTRUSTWORTHY 状态下输出 0
- ◊ truth_detection 代表状态转移的方向
- ◊ 状态在每一个时钟周期的上升沿发生更新

波形说明:

1. 0 - 50ns 初始状态为 HIGHLY_TRUSTWORTHY, 输出为 1
2. 50 - 150ns truth_detection = 0, 状态变为 01, 输出为 1
3. 150 - 250ns truth_detection = 0, 状态变为 10, 输出为 0
4. 250 - 350ns truth_detection = 1, 状态变为 01, 输出为 1
5. 350 - 450ns truth_detection = 0, 状态变为 10, 输出为 0
6. 450 - 550ns truth_detection = 0, 状态变为 11, 输出为 0
7. 550 - 650ns truth_detection = 0, 状态保持 11, 输出为 0
8. 650 - 750ns truth_detection = 1, 状态变为 10, 输出为 0
9. 750 - 850ns truth_detection = 1, 状态变为 01, 输出为 1
10. 850 - 950ns truth_detection = 1, 状态变为 00, 输出为 1
11. 950ns+ truth_detection = 1, 状态保持 00, 输出为 1

Part III 思考题

下图中圈内数字为状态, 箭头上的数字为输入信号

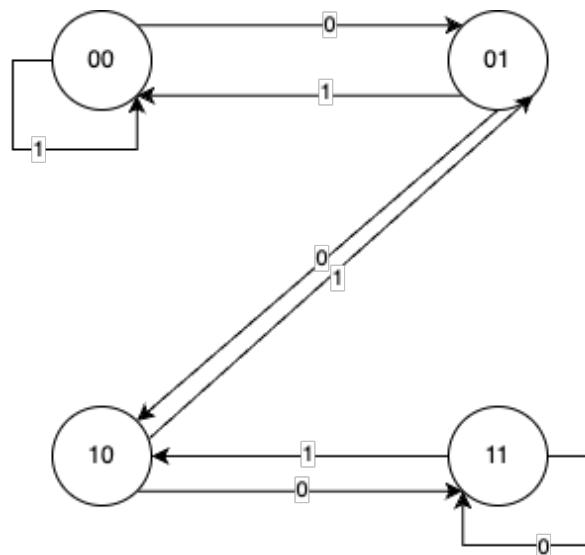


图 10: 信任评估器状态转移图

四、讨论、心得

在做 ALU 的时候出现了赋值的问题，导致左移跟右移没有效果，在助教 Q 的帮助下找到了问题，并且修改了仿真代码，最终顺利通过验收。在写寄存器的时候，开始并没有将 0 地址寄存器赋为 0（上课不听导致的），后来看实验文档才发现要赋值为 0.



图 11: 伟大, 无须多言

lab2:实验平台搭建

一、CSSTE

以下是 CSSTE.v 的完整代码:

```
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
);

    wire U1_MemRW;
    wire [31:0] U1_Addr_out;
    wire [31:0] U1_Data_out;
    wire [31:0] U1_PC_out;

    wire [31:0] U2_spo;

    wire [31:0] U3_douta;

    wire [31:0] U4_Cpu_data4bus;
    wire [31:0] U4_ram_data_in;
    wire [9:0] U4_ram_addr;
    wire U4_data_ram_we;
    wire U4_GPIOf0000000_we;
    wire U4_GPIOe0000000_we;
    wire U4_counter_we;
    wire [31:0] U4_Peripheral_in;

    wire [7:0] U5_point_out;
    wire [7:0] U5_LE_out;
    wire [31:0] U5_Disp_num;

    // wire [7:0] U6_AN;
    // wire [7:0] U6_segment;

    wire [1:0] U7_counter_set;
    wire [15:0] U7_LED_out;

    wire [31:0] U8_clkdiv;
    wire U8_Clk_CPU;
```

```

    wire [3:0] U9_BTN_OK;
    wire [15:0] U9_SW_OK;
    wire U9_RST;

    wire U10_counter0_OUT;
    wire U10_counter1_OUT;
    wire U10_counter2_OUT;
    wire [31:0] U10_counter_out;

SCPU U1(
    .clk(U8_Clk_CPU),
    .rst(U9_RST),
    .Data_in(U4_Cpu_data4bus),
    .inst_in(U2_spo),
    .MemRW(U1_MemRW),
    .Addr_out(U1_Addr_out),
    .Data_out(U1_Data_out),
    .PC_out(U1_PC_out)
);

ROM_I U2(
    .a(U1_PC_out[11:2]),
    .spo(U2_spo)
);

RAM_B U3(
    .clka(~clk_100mhz),
    .wea(U4_data_ram_we),
    .addra(U4_ram_addr),
    .dina(U4_ram_data_in),
    .douta(U3_douta)
);

MIO_BUS U4(
    .clk(clk_100mhz),
    .rst(U9_RST),
    .BTN(U9_BTN_OK),
    .SW(U9_SW_OK),
    .mem_w(U1_MemRW),
    .Cpu_data2bus(U1_Data_out),
    .addr_bus(U1_Addr_out),
    .ram_data_out(U3_douta),
    .led_out(U7_LED_out),
    .counter_out(U10_counter_out),
    .counter0_out(U10_counter0_OUT),
    .counter1_out(U10_counter1_OUT),
    .counter2_out(U10_counter2_OUT),
    .Cpu_data4bus(U4_Cpu_data4bus),
    .ram_data_in(U4_ram_data_in),
    .ram_addr(U4_ram_addr),
    .data_ram_we(U4_data_ram_we),
    .GPIOf0000000_we(U4_GPIOf0000000_we),
    .GPIOe0000000_we(U4_GPIOe0000000_we),

```

```

.counter_we(U4_counter_we),
.Peripheral_in(U4_Peripheral_in)
);

Multi_8CH32 U5(
.clk(~U8_Clk_CPU),
.rst(U9_rst),
.EN(U4_GPIOe0000000_we),
.Test(SW[7:5]),
.point_in({U8_clkdiv[31:0],U8_clkdiv[31:0]}),
.LES(64'b0),
.Data0(U4_Peripheral_in),
.data1({2'b0,U1_PC_out[31:2]}),
.data2(U2_spo),
.data3(U10_counter_out),
.data4(U1_Addr_out),
.data5(U1_Data_out),
.data6(U4_Cpu_data4bus),
.data7(U1_PC_out),
.point_out(U5_point_out),
.LE_out(U5_LE_out),
.Disp_num(U5_Dispatcher)
);

Seg7_Dev_0 U6(
.disp_num(U5_Dispatcher),
.point(U5_point_out),
.les(U5_LE_out),
.scan({U8_clkdiv[18:16]}),
.AN(AN),
.segment(segment)
);

SPIO U7(
.clk(~U8_Clk_CPU),
.rst(U9_rst),
.Start(U8_clkdiv[20]),
.EN(U4_GPIOf0000000_we),
.P_Data(U4_Peripheral_in),
.counter_set(U7_counter_set),
.LED_out(U7_LED_out)
);

assign LED_out = U7_LED_out;

clk_div U8(
.clk(clk_100mhz),
.rst(U9_rst),
.SW2(U9_SW_OK[2]),
.SW8(U9_SW_OK[8]),
.STEP(U9_SW_OK[10]),
.clkdiv(U8_clkdiv),
.Clk_CPU(U8_Clk_CPU)
);

```

```

SAnti_jitter U9(
    .clk(clk_100mhz),
    .RSTN(RSTN),
    .Key_y(BTN_y),
    .SW(SW),
    .BTN_OK(U9_BTN_OK),
    .SW_OK(U9_SW_OK),
    .rst(U9_rst)
);

Counter_x U10(
    .clk(~U8_Clk_CPU),
    .rst(U9_rst),
    .clk0(U8_clkdiv[6]),
    .clk1(U8_clkdiv[9]),
    .clk2(U8_clkdiv[11]),
    .counter_we(U4_counter_we),
    .counter_val(U4_Peripheral_in),
    .counter_ch(U7_counter_set),
    .counter0_OUT(U10_counter0_OUT),
    .counter1_OUT(U10_counter1_OUT),
    .counter2_OUT(U10_counter2_OUT),
    .counter_out(U10_counter_out)
);

VGA U11(
    .clk_25m(U8_clkdiv[1]),
    .clk_100m(clk_100mhz),
    .rst(U9_rst),
    .pc(U1_PC_out),
    .inst(U2_spo),
    .alu_res(U1_Addr_out),
    .mem_wen(U1_MemRW),
    .dmem_o_data(U3_douta),
    .dmem_i_data(U4_ram_data_in),
    .dmem_addr(U1_Addr_out),
    .hs(HSYNC),
    .vs(VSYNC),
    .vga_r(Red),
    .vga_g(Green),
    .vga_b(Blue)
);

endmodule

```

二、处理 VGA

在 VGA.v 中增加了部分输入端口，并接到模块 vga_debugger 中，完成填空。以下是 VGA.v 的完整代码：

```

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_auipc,
    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 [31:0] alu_res,
    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_wen,
    input wire mem_ren,
    //  input wire [31:0] dmem_o_data,
    //  input wire [31:0] dmem_i_data,
    //  input wire [31:0] dmem_addr,
    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_auipc (is_auipc  ),
    .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

```

另外还要对 VgaDisplay.v 进行修改，把其中读取 ROM 和 RAM 对地址进行更新，更改为本地地址。以下是 VgaDisplay.v 的完整代码：

```

module VgaDisplay(
  input wire clk,
  input wire video_on,
  input wire [9:0] vga_x,
  input wire [8:0] vga_y,
  output wire [3:0] vga_r,
  output wire [3:0] vga_g,
  output wire [3:0] vga_b,
  input wire wen,
  input wire [11:0] w_addr,
  input wire [7:0] w_data
);

(* ram_style = "block" *) reg [7:0] display_data[0:4095];
initial $readmemh("D:/Xilinx/CO_Lab/CO_Lab2/OExp02-IP2SOC/vga_debugger.mem", display_data);

```

```

wire [11:0] text_index = (vga_y / 16) * 80 + vga_x / 8;
// I don't know why I need this '-(vga_y / 16)' ...
wire [7:0] text_ascii = display_data[text_index] - (vga_y / 16);
wire [2:0] font_x = vga_x % 8;
wire [3:0] font_y = vga_y % 16;
wire [11:0] font_addr = text_ascii * 16 + font_y;

(* ram_style = "block" *) reg [7:0] fonts_data[0:4095];
initial $readmemh("D:/Xilinx/CO_Lab/CO_Lab2/OExp02-IP2SOC/font_8x16.mem", fonts_data);
wire [7:0] font_data = fonts_data[font_addr];

assign { vga_r, vga_g, vga_b } = (video_on & font_data[7 - font_x]) ? 12'hfff : 12'h0;

always @(posedge clk) begin
    if (wen) begin
        display_data[w_addr] <= w_data;
    end
end

endmodule

```

三、 I_mem.pdf 的程序的功能

对前面几条指令进行解释

0x0	0x00100093	addi x1 x0 1	loop:addi x1,x0,1	#x1=00000001
0x4	0x00102133	slt x2 x0 x1	slt x2,x0,x1	#x2=00000001
0x8	0x002101B3	add x3 x2 x2	add x3,x2,x2	#x3=00000002
0xc	0x00218233	add x4 x3 x2	add x4,x3,x2	#x4=00000003

1. 第一条指令是 x0 和立即数 1 进行加法，结果存到 x1 中，x1 的结果为 1
2. 第二条指令是 x0 < x1，然后把这个式子的结果存到 x2 中，x2 的结果为 1
3. 第三条指令是 x2 + x2，结果存到 x3 中，x3 的结果为 2
4. 第四条指令是 x3 + x2，结果存到 x4 中，x4 的结果为 3
5. 后面的指令都是前一项与前一项的和，结果存到下一个寄存器中

0x78	0x01DF0FB3	add x31 x30 x29	add x31,x30,x29	#x31=00621963
0x7c	0xF80002E3	beq x0 x0 -124	beq x0,x0,loop	#pc---->loop

◊ 最后一条指令是跳转指令，回到循环开始的位置。

所以这个程序的功能是计算斐波那契数列，并且不断重复。

四、 下板验证

烧录完成后，连接板子，根据按钮介绍验证。我们主要验证程序的正确性。

Part I 第一条指令的执行情况

首先我们设置 SW[8]SW[2] = 10, 这代表手动单步时钟; SW[7"5]=101, 输出寄存器 B 的值, 在 VGA 对应 dmem_i_data 的值, 也是我们要观察的主要的运算结果。



```
00321 Single Cycle CPU
pc: 00000000 inst: 00100093
x0: 00000000 rx: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_limm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ran: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000001

car_wen: 0 car_ldi: 000 car_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

图 14: VGA 显示

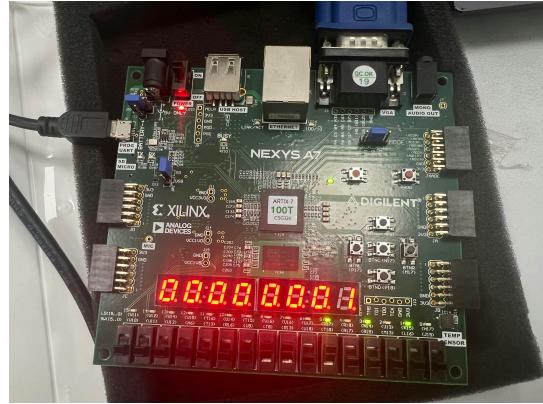
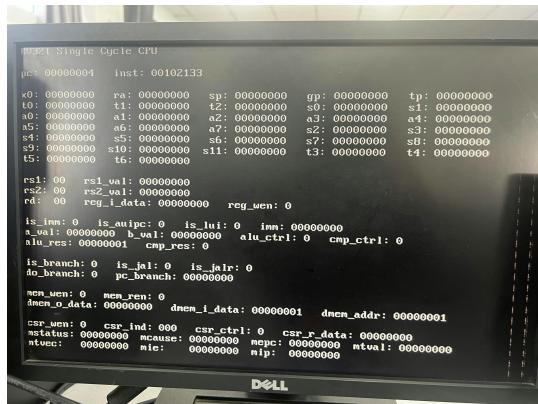


图 15: 板子显示

可以看到此时 pc=0, 显示的是执行完第一条指令 (x0 和立即数 1 进行加法) 的结果, 此时 ALU 结果为 1, dmem_i_data = 1

Part II 第二条指令的执行情况

我们搬动 SW[10]一次, 时钟单步一个周期。



```
00321 Single Cycle CPU
pc: 00000004 inst: 00102133
x0: 00000000 rx: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_limm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ran: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000001

car_wen: 0 car_ldi: 000 car_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

图 16: VGA 显示

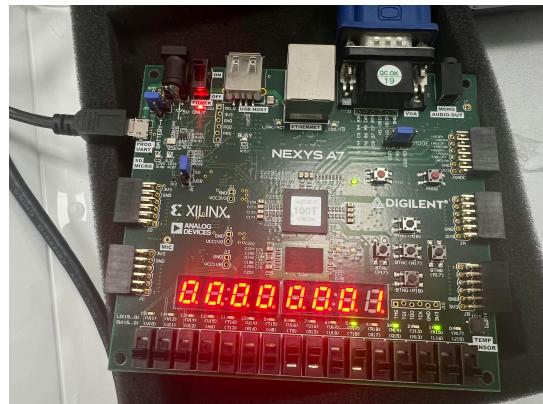


图 17: 板子显示

可以看到此时 pc=4, 显示的是执行完第二条指令 (x0 < x1) 的结果, 此时 ALU 结果为 1,dmem_i_data = 1

Part III 第三条指令的执行情况

我们搬动 SW[10]一次, 时钟单步一个周期。

```

iM32I Single Cycle CPU
pc: 00000008 inst: 002101b3
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 a8: 00000000 a9: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_lwm: 0 is_uwipc: 0 is_lui: 0 imm: 00000000
u_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_rren: 0
mem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000002
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
status: 00000000 cause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

图 18: VGA 显示

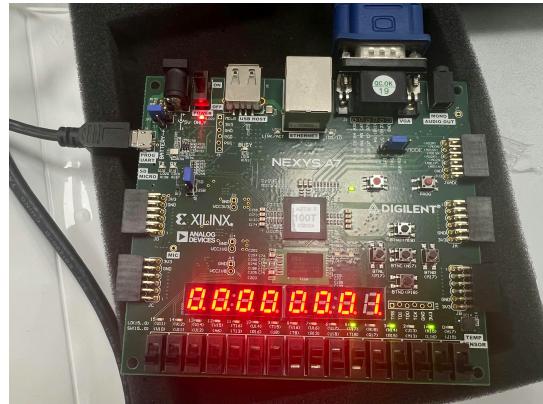


图 19: 板子显示

可以看到此时 $pc=8$, 显示的是执行完第三条指令 ($x_2 + x_2$) 的结果, 此时 ALU 结果为 2, $dmem_i_data = 1$

Part IV 第四条指令的执行情况

我们搬动 SW[10]一次, 时钟单步一个周期。

```

iM32I Single Cycle CPU
pc: 0000000c inst: 00210233
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 a8: 00000000 a9: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_lwm: 0 is_uwipc: 0 is_lui: 0 imm: 00000000
u_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000003 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_rren: 0
mem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000003
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
status: 00000000 cause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

图 20: VGA 显示

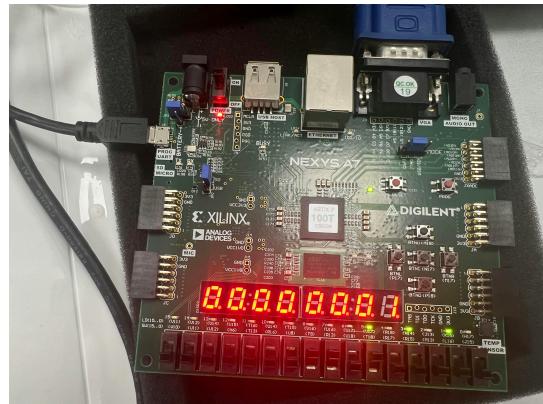


图 21: 板子显示

可以看到此时 $pc=12$, 显示的是执行完第四条指令 ($x_3 + x_2$) 的结果, 此时 ALU 结果为 3, $dmem_i_data = 1$

Part V 后续指令的执行情况

执行 1

我们搬动 SW[10]若干次，时钟单步一个周期。



图 22: VGA 显示



图 23: 板子显示

可以看到此时 pc=38, 此时 ALU 结果为 262,dmem_i_data = E9

执行 2

接着搬动 SW[10]若干次，时钟单步一个周期。



图 24: VGA 显示

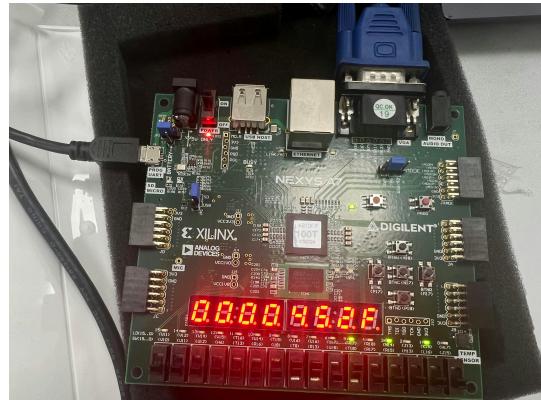


图 25: 板子显示

可以看到此时 pc=5C, 此时 ALU 结果为 b520,dmem_i_data = 452F

执行 3

接着搬动 SW[10]若干次，时钟单步一个周期。

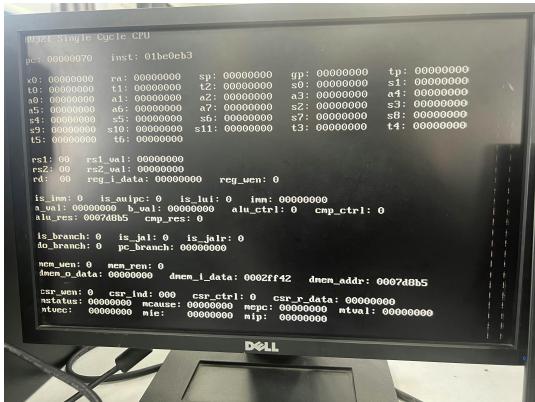


图 26: VGA 显示

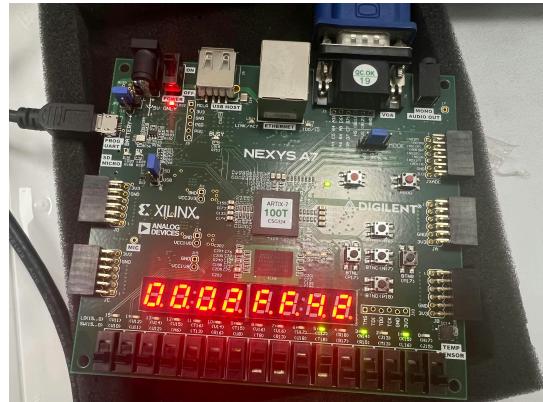


图 27: 板子显示

可以看到此时 pc=70, 此时 ALU 结果为 7d8b5,dmem_i_data = 2ff42

Part VI 总结

以上指令运行情况均满足 I_mem.pdf 的程序功能，说明下载验证也是满足预期的。此外这里展示的 VGA 有些问题，打印的左边出了点问题，但是在之前一次显示的时候是正常的，可能是因为 VGA 显示屏的问题。

五、讨论、心得

这次连线相当痛苦，在 LED_out 处没有看清楚输出，导致悬空线，最后在助教 Q 的帮助下解决了问题。好在上板验证一次就通过了，没有经受路径问题的折磨。



图 28: 伟大, 无须多言

lab3:乘除法器、浮点加法器

一、 乘法器

Part I 模块实现

我实现的是有符号，无溢出判断乘法器，以下是乘法器完整代码：

```
module multiplier(
    input clk,
    input start,
    input[31:0] A,
    input[31:0] B,
    output reg finish,
    output reg[63:0] res
);

reg state; // 记录 multiplier 是否正在进行运算
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;

reg sign = 0; // 记录当前运算的结果是否是负数

wire [31:0] unsighed_A;
wire [31:0] unsighed_B;
assign unsighed_A = (A[31] == 1'b1) ? ~A + 1 : A;
assign unsighed_B = (B[31] == 1'b1) ? ~B + 1 : B;

initial begin
    res <= 0;
    state <= 0;
    finish <= 0;
    cnt <= 0;
    multiplicand <= 0;
end

always @(posedge clk) begin
    if(~state && start) begin
        // Not Running
        sign <= A[31] ^ B[31];
        multiplicand <= unsighed_A;
        res <= {32'b0, unsighed_B};
        state <= 1'b1;
        finish <= 1'b0;
        cnt <= 1'b0;
    end else if(state) begin
        // Running
        // Why not "else"?
        // 你需要在这里处理“一次”运算与移位
    end
end
```

```

if(res[0] == 1'b1) begin
    res[63:32] = res[63:32] + multiplicand;
end
res = res >>> 1;
cnt = cnt + 1'b1;
end

// 填写 cnt 相关的内容，用 cnt 查看当前运算是否结束
if(cnt == 0 && state == 1 ) begin
// 得到结果
    finish <= 1;
    state <= 0;
    if(sign) begin
        res <= ~res + 1'b1;
    end
end
end

endmodule

```

我们这里仅是把 start 作为开始信号，一旦 start 置于高位，那么无论后面是否为 1，都会开始运算。

Part II 仿真测试

仿真代码如下：

```

module multiplier_tb;

reg clk, start;
reg[31:0] A;
reg[31:0] B;

wire finish;
wire[63:0] res;

multiplier m0(.clk(clk), .start(start), .A(A), .B(B), .finish(finish), .res(res));

initial begin
    $dumpfile("multiplier_signed.vcd");
    $dumpvars(0, multiplier_tb);

    clk = 0;
    start = 0;
    #10;
    A = 32'd1;
    B = 32'd0;
    #10 start = 1;
    #10 start = 0;
    #200;

```

```

A = 32'd10;
B = 32'd30;
#10 start = 1;
#10 start = 0;
#200;

A = -32'd10;
B = 32'd30;
#10 start = 1;
#10 start = 0;
#200;

A = 32'd66;
B = 32'd23;
#10 start = 1;
#10 start = 0;
#500;
$finish();
end

always begin
#2 clk = ~clk;
end

endmodule

```

仿真波形如下：

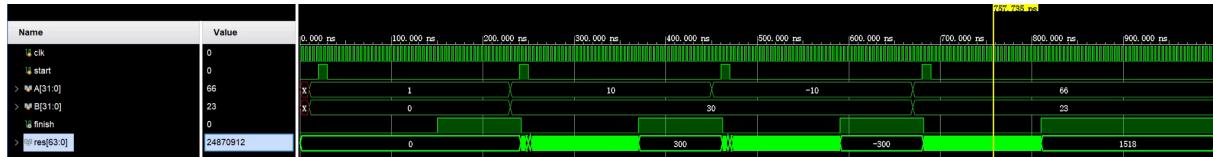


图 29：乘法器仿真波形图

仿真图内已经选择 10 进制表示，可以看到有 0 情况和符号数的情况都是正确的。验收的时候，助教问的阻塞赋值和非阻塞赋值的问题，在这里我们对 res 的运算时的赋值时阻塞赋值，这是因为这时候的运算有先后顺序的要求。还有一点没做好的是应该对同一个变量要始终使用同一种赋值方式，最好不要混合使用。

二、除法器

Part I 模块实现

我实现的是无符号，无溢出判断、有除零异常判断除法器，以下是除法器完整代码：

```

module divider(
    input  clk,
    input  rst,

```

```

input start,      // 开始运算
input[31:0] dividend, // 被除数
input[31:0] divisor, // 除数
output reg divide_zero, // 除零异常
output reg finish,    // 运算结束信号
output[31:0] res,    // 商
output[31:0] rem     // 余数
);

reg [63:0] remainder;
reg [31:0] quotient;
reg [63:0] high_divisor;
reg [5:0] count;
reg state;

assign res = quotient;
assign rem = remainder[31:0];

always @(posedge clk or posedge rst) begin
  if(rst) begin
    divide_zero <= 0;
    finish <= 0;
    state <= 0;
    count <= 0;
    remainder <= 0;
    quotient <= 0;
    high_divisor <= 0;
  end
  else if(start) begin
    state <= 1;
    if(divisor == 0) begin
      finish <= 1;
      divide_zero <= 1;
    end
    else begin
      remainder = {32'b0, dividend};
      high_divisor = {divisor, 32'b0};
      count = 0;
      quotient = 0;
      finish = 0;
    end
  end
  else if(state && ~finish) begin
    if(remainder >= high_divisor) begin
      remainder = remainder - high_divisor;
      quotient[0] = 1;
    end
    quotient = quotient << 1;
    high_divisor = high_divisor >> 1;
    count = count + 1;
  end
  if(count == 32) begin

```

```

    state = 0;
    finish = 1;
end
end
endmodule

```

Part II 仿真测试

仿真代码如下:

```

module divider_tb;

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

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

divider
d0(.clk(clk), .rst(rst), .start(start), .dividend(dividend), .divisor(divisor), .divide_zero(divide_zero), .finish(finish), .res(res), .rem(rem));

initial begin
$dumpfile("divider.vcd");
$dumpvars(0, divider_tb);

clk = 0;
rst = 1;
start = 0;
#10;
rst = 0;

dividend = 32'd1;
divisor = 32'd0;
#10 start = 1;
#10 start = 0;
#200;

dividend = 32'd60;
divisor = 32'd30;
#10 start = 1;
#10 start = 0;
#200;

dividend = 32'd60;
divisor = 32'd80;
#10 start = 1;
#10 start = 0;
#200;

```

```

dividend = 32'd323;
divisor = 32'd23;
#10 start = 1;
#10 start = 0;
#500;
$finish();
end

always begin
#2 clk = ~clk;
end
endmodule

```

仿真波形如下:

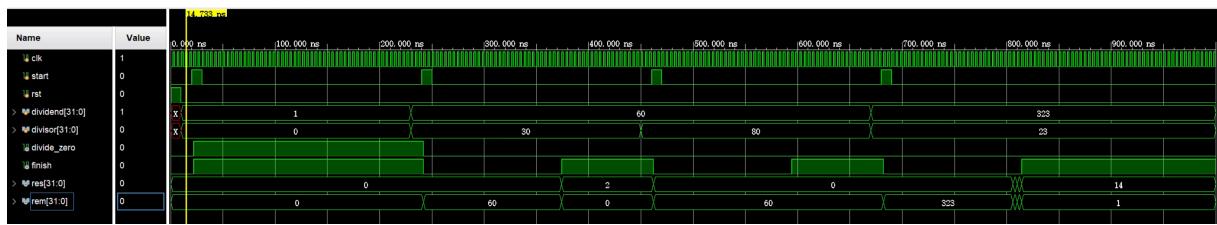


图 30: 除法器仿真波形图

上面仿真同样是 10 进制表示，先验证了除 0 判断，然后验证了正常的除法运算，最后验证了余数的正确性。

我们这里将 start 作为启动信号，但与乘法器不同的是，只有当 start 置于高位再置于低位时，才会开始运算，尽管这也实现了除 0 判断，但是这样的实现方式不够优雅，应该在 start 置于高位时，就开始运算，并且将我们的除 0 判断更新放在每次除法的过程中。

三、浮点加法器

Part I 模块实现

我实现的是单精度浮点加法器，以下是浮点加法器完整代码：

```

module fpadder(
    input clk,
    input rst,
    input [31:0] A,
    input [31:0] B,
    input start,
    output reg [31:0] result,
    output reg finish
);
    reg [2:0] state;
    reg sign,sign1,sign2;
    reg [7:0] exp,exp1,exp2;
    reg [24:0] frac,frac1,frac2;

```

```

localparam
S0 = 3'b000, //check denormal number
S1 = 3'b001,
S2 = 3'b010,
S3 = 3'b011,
S4 = 3'b100,
S5 = 3'b101;

always @(posedge clk or posedge rst) begin
if(rst)begin
    result <= 0;
    finish <= 0;
end
if(start)begin
    sign1 <= A[31];
    exp1 <= A[30:23];
    frac1 <= {2'b01, A[22:0]};
    sign2 <= B[31];
    exp2 <= B[30:23];
    frac2 <= {2'b01, B[22:0]};
    state <= S0;
    finish <= 0;
end
else if(~finish)begin
    case(state)
        S0: begin
            if(exp1 == 8'b0 || exp1 == 8'b11111111)begin
                exp <= exp1;
                frac <= frac1;
                sign <= sign1;
                state <= S5;
            end
            else if(exp2 == 8'b0 || exp2 == 8'b11111111)begin
                exp <= exp2;
                frac <= frac2;
                sign <= sign2;
                state <= S5;
            end
            else begin
                state <= S1;
            end
        end
        S1: begin
            if(exp1 == exp2)begin
                state <= S2;
            end
            else if(exp1 > exp2)begin
                exp2 <= exp2 + 1;
                frac2 <= frac2 >> 1;
                if(frac2 == 0)begin
                    exp <= exp1;
                    frac <= frac1;
                    state <= S5;
                end
            end
        end
    endcase
end
end

```

```

    end
else begin
    exp1 <= exp1 + 1;
    frac1 <= frac1 >> 1;
    if(frac1 == 0)begin
        exp <= exp2;
        frac <= frac2;
        state <= S5;
    end
end
end
S2: begin
    if(sign1 ^ sign2 == 0)begin//same sign
        sign <= sign1;
        frac <= frac1 + frac2;
    end
    else if(sign1 == 1)begin
        sign <= (frac2 > frac1) ? 0 : 1;
        frac <= (frac2 > frac1) ? (frac2 - frac1) : (frac1 - frac2);
    end
    else begin
        sign <= (frac2 > frac1) ? 1 : 0;
        frac <= (frac2 > frac1) ? (frac2 - frac1) : (frac1 - frac2);
    end
    exp <= exp1;
    state <= S3;
end
S3: begin
    if(frac[24] == 1) begin
        exp <= exp + 1;
        frac <= frac >> 1;
    end
    else if(frac[23] == 0)begin
        exp <= exp -1;
        frac <= frac << 1;
    end
    else begin
        state <= S4;
    end
end
S4: begin
    state <= S5;
end
S5: begin
    result <= {sign, exp, frac[22:0]};
    finish <= 1;
end
endcase
end
end
endmodule

```

我们根据浮点数加法算法的五个阶段进行了实现，其中 S0 阶段是检查是否为非规格化数，S1 阶段是对齐，S2 阶段是加法，S3 阶段是规格化，S4 阶段是进行舍入但实际上啥也没干，S5 阶段是输出结果。

Part II 仿真测试

仿真代码如下：

```
module fpadder_tb;
reg clk;
reg rst;
reg [31:0] A;
reg [31:0] B;
reg start;
wire [31:0] result;
wire finish;

// Instantiate the Unit Under Test (UUT)
fpadder uut (
    .clk(clk),
    .rst(rst),
    .A(A),
    .B(B),
    .start(start),
    .result(result),
    .finish(finish)
);
initial begin
    clk = 0;
    rst = 1;
    A = 0;
    B = 0;
    start = 0;
    #50;
    rst = 0;

    A=32'hc0a00000;
    B=32'hc0900000;
    start = 1; #10;
    start = 0; #80;

    A=32'h40a00000;
    B=32'h40900000;
    start = 1; #10;
    start = 0; #80;

    A = 32'h3fa00000;
    B = 32'hc06ccccd;
    start = 1; #10;
    start = 0; #80;

    A = 32'h3fa00000;
```

```

B = 0;
start = 1; #10;
start = 0; #80;

A = 0;
B = 32'h3fa00000;
start = 1; #10;
start = 0; #80;

$stop();
end

always #5 clk = ~clk;

endmodule

```

仿真波形如下:

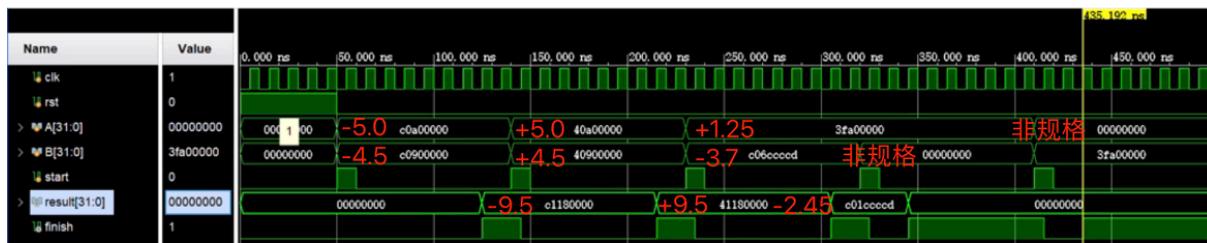


图 31: 浮点加法器仿真波形图

在图中已经详细地标出了每一段的运算值和结果，可以看到包括了非规格化数的情况，正常的加法运算，也验证了需要对齐的情况，均得到了满足。

Part III 思考题

- ◊ 1. 1.0
- 2. 0.0
- 3. 两者是有区别的，第一个先进行 $x+y=0$ ，所以最终结果是 1.0，第二个先进行了 $y+z$ 由于对齐导致实际上 z 的值被舍入时被舍去了，所以最终结果是 0.0。
- ◊ 1. -0.000954
- 2. 0.000000
- 3. 在上面程序中，我们的预期结果是 0.0。但是根据浮点数的存储方式，有些数据并不能精确的被存入内存中。在这里 0.1 只能被近似地表示为

$$0x3DCCCCCD = (1 + 0.600000023841858) \times 2^{\{123-127\}} = 0.1000000149011612$$

，所以在计算时会有一定的误差，导致最终结果不是 0.0。

而对于 0.125，它的二进制表示是精确的

$$0x3E000000 = (1 + 0.0) \times 2^{\{126-127\}} = 0.125$$

，所以它的结果是 0.125。

四、讨论、心得

这次的实验算是熟悉了操作环境后，最轻松的一次实验了，只根据理论课上的算法进行逐一实现即可，也没有遇到太多的问题，后面就要到实验四了，希望能够顺利完成。