浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	熊子宇
学　　号：	3200105278
学　　院：	竺可桢学院
专　　业：	混合班
指导教师：	姜晓红/叶大源
报告日期：	2022 年 3 月 17 日

实验二 Warm Up

一、实验目的

1. 实现 ALU 并仿真验证
2. 实现 Comparator 并仿真验证
3. 实现 RegFile 并仿真验证
4. 替换 Lab1 工程的 Core 部件，上板验证

二、实验方法与步骤

1 实现 ALU 并仿真验证

1.1 ALU 功能介绍

我们设计的 ALU 接受两个 32-bit 输入，依据 Control Unit 的 ALU 控制信号，输出运算结果。一共能够执行 10 种不同的运算：

```
`define ALU_ADD  4'd0
`define ALU_SUB  4'd1
`define ALU_SLL  4'd2
`define ALU_SLT  4'd3
`define ALU_SLTU 4'd4
`define ALU_XOR  4'd5
`define ALU_SRL  4'd6
`define ALU_SRA  4'd7
`define ALU_OR   4'd8
`define ALU_AND  4'd9
```

1.2 ALU 功能实现

基本思路是用多路选择器，根据控制信号选择不同的运算结果。多路选择器在 Verilog 中可以用 case 语句实现。而各种运算如果真的用基本门电路实现，存在一定困难（尤其是加减法器），在这里直接使用了 Verilog 内嵌的各种运算符，大大降低了我们设计电路的难度。

对我而言特别要注意的是如下几种运算：

一，slt 和 sltu 的判断。对于 slt 可以直接使用 signed int 的减法结果的最高位判断。而在 sltu 情况下，若要判断 unsigned int A 和 B 孰大孰小，我令 A - B + 2^33，如果未发生借位（即结果的第 33 位仍为 1），则说明 A>B，sltu 置 0；如果发生借位（即结果的第 33 位为 0），则 sltu 置 1。

二，Verilog 中内置了 signed 类型。使用$signed(a_val)即可显式地表明 a_val 为 signed int。

三，混淆了逻辑与(&&)和按位与(&)的符号，以及逻辑或(||)和按位或(|)的符号。

主要代码如下（本报告中只有这里的代码比较长）

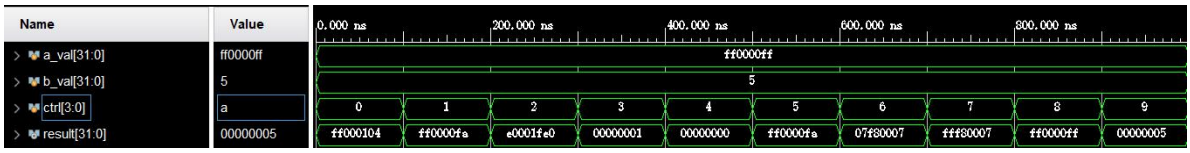
```
module Alu(
 input [31:0] a_val, // a 操作数
 input [31:0] b_val, // b 操作数
 input [3:0] ctrl, // ALU 的 Control 信号
 output reg [31:0] result // ALU 运算的结果
);
    wire [31:0]add_res;
    wire [31:0]sub_res;
    wire [31:0]sll_res;
    wire [31:0]slt_res;
    wire [31:0]sltu_res;
    wire [31:0]xor_res;
    wire [31:0]srl_res;
    wire [31:0]sra_res;
    wire [31:0]or_res;
    wire [31:0]and_res;

    assign add_res = a_val + b_val;
    assign sub_res = a_val - b_val;
    assign sll_res = a_val << b_val[4:0];
    assign slt_res = (sub_res[31] == 1) ? 1 : 0;
    wire [32:0]temp_add;
    assign temp_add = a_val - b_val + 33'h100000000;
    assign sltu_res = (temp_add[32] == 0) ? 1 : 0;//注意这个判断方法
    assign xor_res = a_val ^ b_val;
    assign srl_res = a_val >> b_val[4:0];
    assign sra_res = $signed(a_val) >>> b_val[4:0];//本来想用 SEXT 位
扩展，语法错误
    assign or_res = a_val | b_val;//应该是按位或|，而不是逻辑或||
    assign and_res = a_val & b_val;//应该是按位与&,而不是逻辑与&&

    always @* begin
        case (ctrl)
            `ALU_ADD: result = add_res;
            `ALU_SUB: result = sub_res;
            `ALU_SLL: result = sll_res;
            `ALU_SLT: result = slt_res;
            `ALU_SLTU: result = sltu_res;
            `ALU_XOR: result = xor_res;
            `ALU_SRL: result = srl_res;
            `ALU_SRA: result = sra_res;
            `ALU_OR: result = or_res;
            `ALU_AND: result = and_res;
        endcase
    end
endmodule
```

1.3 仿真验证

仿真验证关键代码如下：

```
    a_val = 32'hff0000ff;
    b_val = 32'd5;
    for (ctrl = 0; ctrl < 10; ctrl = ctrl + 1) begin
        #100;
```

```
    end
```




```
ALU_ADD: ff0000ff + 5 = ff000104
ALU_SUB: ff0000ff - 5 = ff0000fa
ALU_SLL:
11111111000000000000000011111111 << 5 = 11100000000000000011111111
ALU_SLT: ff0000ff < 5，所以 slt = 1
ALU_SLTU: ff0000ff > 5，所以 sltu = 0
ALU_XOR: ff0000ff ^ 5 = ff0000fa
ALU_SRL:
11111111000000000000000011111111 >> 5 = 00000111000000000000000000111
ALU_SRA:
11111111000000000000000011111111 >> 5 = 11111111000000000000000000111
ALU_OR: ff0000ff | 5 = ff0000ff
ALU_AND: ff0000ff & 5 = 5
```

由仿真波形图可知，各运算结果均正确。

2 实现 Comparator 并仿真验证

2.1 Comparator 功能介绍

用于比较两个数的大小并输出真值信号。在 CPU 中，比较器被用来做分支跳转语句的判断控制。共有 6 种判断模式：

```
`define CMP_EQ  3'd0
`define CMP_NE  3'd1
`define CMP_LT  3'd2
`define CMP_LTU 3'd3
`define CMP_GE  3'd4
`define CMP_GEU 3'd5
```

2.2 Comparator 功能实现

基本思路和 ALU 非常类似，同样是多路选择器+控制信号选择不同比较结果，不再赘述。各种比较也是利用 Verilog 的内嵌语法。比如使用$signed(a_val)和<组合就可以轻松得到 stl 的结果（这么说来在 ALU 中我仍然选择了稍微底层一点的方式实现 stl 和 stlu）。此外也要注意到 6 种判断模式两两互补，所以实际上只要写出 3 种模式的判断方法即可。

关键代码如下：

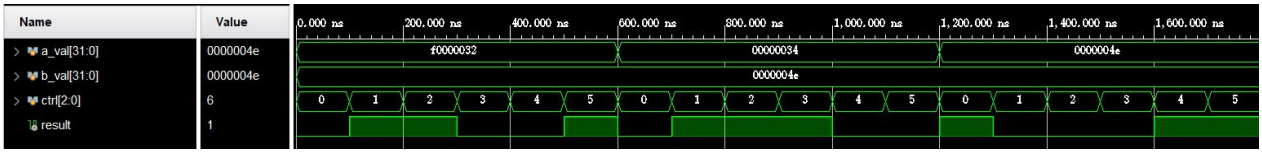
```
assign EQ_res = (a_val == b_val) ? 1 : 0;
assign NE_res = ~EQ_res;
assign LT_res = ($signed(a_val) < $signed(b_val)) ? 1 : 0;
assign LTU_res = (a_val < b_val) ? 1 : 0;
assign GE_res = ~LT_res;
assign GEU_res = ~LTU_res;
```

2.3 仿真验证

仿真验证主要代码如下：

```
a_val = 32'hf0000032;
b_val = 32'd78;
for (ctrl = 0; ctrl < 6; ctrl = ctrl + 1) begin
    #100;
end
a_val = 32'd52;
b_val = 32'd78;
for (ctrl = 0; ctrl < 6; ctrl = ctrl + 1) begin
    #100;
end
a_val = 32'd78;
b_val = 32'd78;
for (ctrl = 0; ctrl < 6; ctrl = ctrl + 1) begin
    #100;
end
```

如果仅使用一组 a 和 b 的值，无法完善地测试所有比较结果，因此我分别采用了 a > b，a < b 和 a = b 三组数据进行仿真验证。



对照宏定义可知，各组比较结果均正确。

```
`define CMP_EQ  3'd0
`define CMP_NE  3'd1
`define CMP_LT  3'd2
`define CMP_LTU 3'd3
`define CMP_GE  3'd4
`define CMP_GEU 3'd5
```

3 实现 RegFile 并仿真验证

3.1 RegFile 功能介绍

RegFile 是存放和读写寄存器的模块。我们要实现的 CPU 中有 32 个寄存器，每个寄存器有 32bit。要注意 0 号寄存器只读，由硬件置 0。

3.2 RegFile 功能实现

首先我们要定义寄存器组。0 到 31 号共 32 个寄存器，每个寄存器 32 位。

再分析 RegFile 的功能，主要有三个方面：

第一，读 rs1 和 rs2 对应的寄存器的数据；

第二，当 rst 为 1 时异步清零寄存器；

第三，当 wen 置 1 且处于时钟正边沿时改写 rd 寄存器内容。

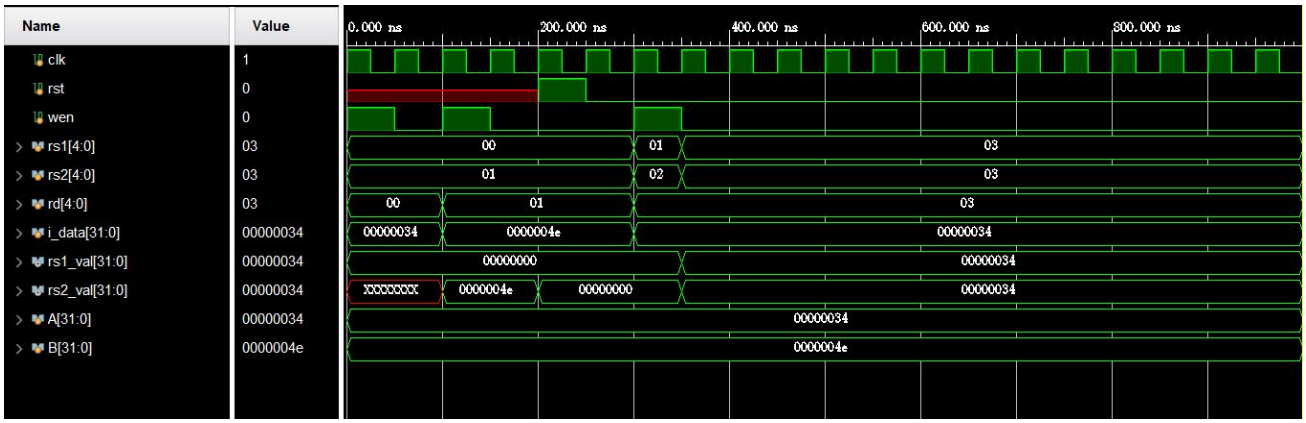
代码如下：

```
module RegFile(
 input clk, rst,
```

```
    input wen, // 写使能
    input [4:0] rs1, // 源寄存器 1 的编号
    input [4:0] rs2, // 源寄存器 2 的编号
    input [4:0] rd, // 目的寄存器的编号
    input [31:0] i_data, // 写入的数据
    output [31:0] rs1_val, // 源寄存器 1 的输出数据
    output [31:0] rs2_val // 源寄存器 2 输出的数据
       );
       reg [31:0] register [0:31];
       integer i;
       assign rs1_val = (rs1 == 0) ? 0 : register[rs1];
       assign rs2_val = (rs2 == 0) ? 0 : register[rs2];
       always @(posedge clk or posedge rst) begin
           if (rst == 1)
               for (i = 1; i < 32; i = i+1)
                   register[i] <= 0;
           else if ((wen == 1) && (rd != 0))
                   register[rd] <= i_data;
       end
    endmodule
```

3.3 仿真验证

仿真验证代码均是简单的赋值语句，因此省略，看波形图即可。



周期序号	写行为	读行为(寄存器)	其他行为
1	x0 ← A	x0, x1	
2		x0, x1	
3	x1 ← B	x0, x1	
4		x0, x1	
5		x0, x1	rst ← 1
6		x0, x1	
7	x3 ← A	x1, x2	
8		x3, x3	

周期 1，x0 <- A 行为失败，rs1_val = 0。x1 并未初始化，所以返回 XXXXXXXX。

周期 3，x1 <- B 成功，rs2_val = B = 0000004e。

周期 5，rst = 1 后，所有寄存器被清零。rs1_val = rs2_val = 0。从周期 6、7 返回值可见至少 x0, x1, x2 均为 0。

周期 8，由于在周期 7 完成了 x3 <- A，所以周期 8 时 rs1_val = rs2_val = A = 00000034。

4 替换 Lab1 工程的 Core 部件，上板验证

4.1 删除 Lab1 提供的 Core 组件，导入 CoreOthers 文件

4.2 导入已验证过的 ALU、Comparator 和 RegFile 组件

在这里我通过 Add Files 的方式导入了 Comparator 和 RegFile，通过生成带源文件 IP 核的方式导入了 alu。文件结构如下图所示。

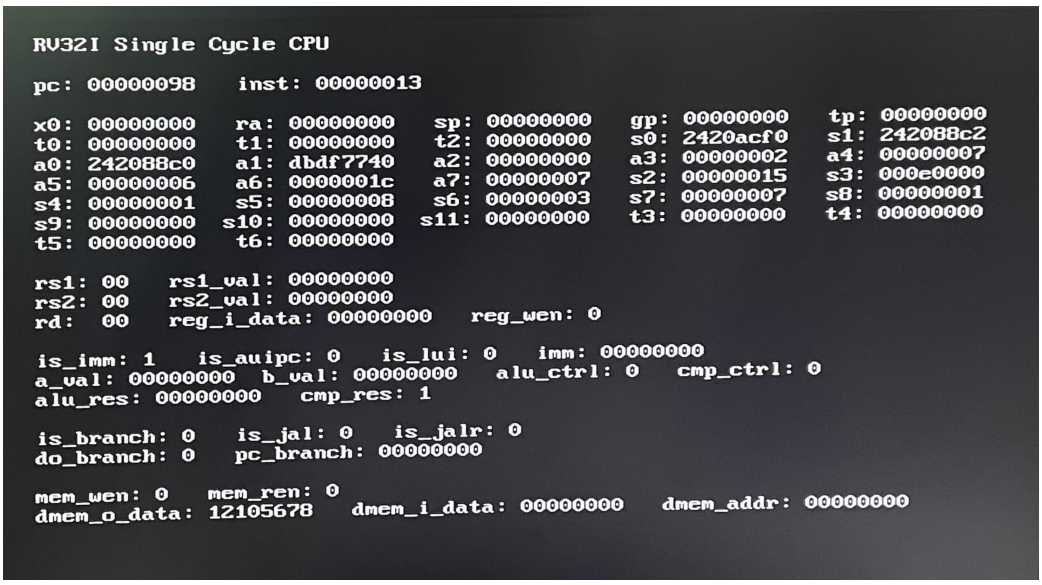
4.3 增加 RegFile 中的 Debug 宏定义

把`VGA_DBG_RegFile_Outputs 放在 RegFile.v 的参数表中，把`VGA_DBG_RegFile_Assignments 放在 RegFile.v 的内容中。

4.4 生成 CPU 测试框架的 bit 流，上板验证

点击 Generate Bit Stream 后，工程会依次进行 Synthesis, Run Implementation 和 Generate Bit Stream 三步。生成 bit 流后点击 Open Hardware Manger，点击 Auto Connect，Device 选中 Sword 板。

上板后，pc 不断跳动，Debug 信号也有相应值输出，操作 SW[0]可以控制单步/连续，按动行列键盘可以单步跳动。

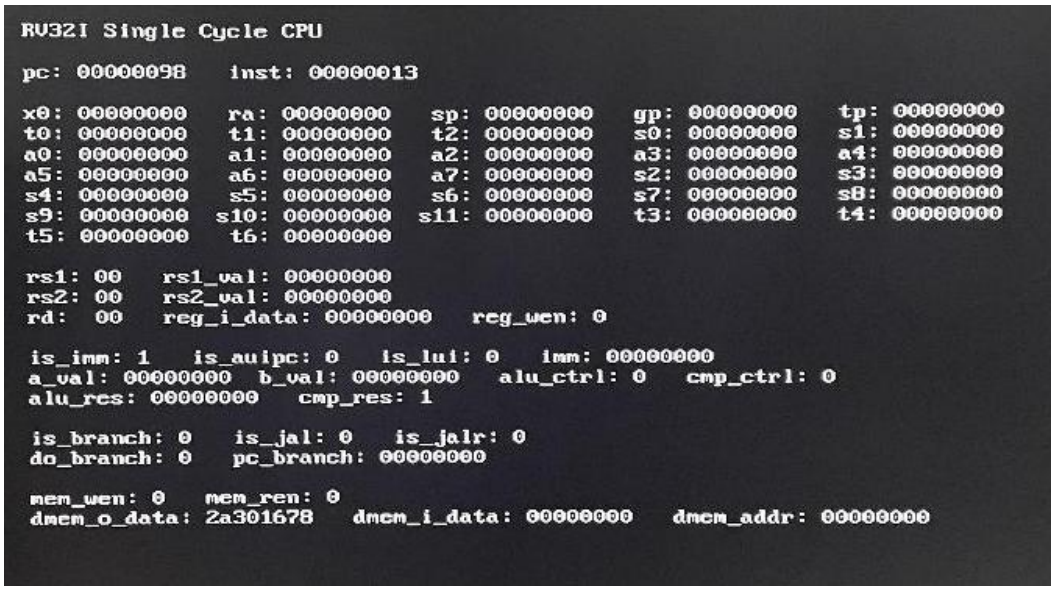


三、实验心得

这一部分主要叙述我做 Lab2 时遇到的 Bug 或困难。

困难一：Verilog 语法不熟悉，从而导致 ALU 仿真时部分计算有误。比如$sign()这种符号是查阅网上资料得知的。还有在 ALU 部分已经叙述过。

困难二：没有意识到要添加 RegFile 的 debug 信号，也不知道怎么使用两个宏定义。最初我上板子验证的结果如下：



可以看到所有寄存器都为 0。然而我没有意识到不对，因为我看不懂 VGA 显示的各个信号分别代表什么意思。经过同学提醒我才把两个宏定义加入 RegFile.v 中。我觉得能看懂调试信息是非常必要的。

问题三：Add Files 操作会使得修改一个工程中的文件，联动另一个引用该文件的工程。这是与 ISE 的 Add Copy of Source 的显著不同之处。Add Copy of Source 实际上是生成了一份副本，从而两个工程中的同名文件已经相互独立；而 Vivado 没有添加副本的功能，所以我在 SCPU 工程中引入 RegFile.v 后，若在 RegFile 工程中注释了宏定义，SCPU 中的 RegFile.v 会自动更新，从而在 Synthesis 步骤报错。而如果不注释宏定义，又无法仿真模拟。解决这个问题的方法，我认为如有必要，最好手动添加副本。

困难四：Auto Connect 时报错 No Target Device。

我尝试了三台 Sword 板，发现都会报错找不到板子，还以为自己电脑出问题。后来发现是实验室的 Sword 板可能存在问题。