Lab3: ALU, Comparator, RegFile

3220106039 李瀚轩

一、操作方法与实验步骤

1.1 ALU模块

ALU 是 CPU 中的主要算术单元， 是一个组合逻辑部件。本次实验我们的 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。我们实现的操作为：

- **ADD**： A + B
- **SUB**： A - B
- **SLL**：逻辑左移，因为操作数只有 32 位，所以移位时只取 B 的低五位，移位时右侧直接补 0，则该操作要实现 A << B[4:0]
- **SLT**: set less than，将 A 和 B 当作有符号数，判断 A < B？当 A < B 时，结果为 0x00000001，否则，结果为 0x00000000
- **SLTU**：无符号版本的 SLT，即将 A 和 B 当作无符号数
- **XOR**：对 A 和 B 进行按位异或操作，即 A ^ B
- **OR**： A | B
- **AND**： A & B
- **SRA**：算术右移，移位时只取 B 的低五位，右移时补上 A 的符号位，则该操作数要实现有符号的 A >> B[4:0]。
- **SRL**：逻辑右移，移位时只取 B 的低五位，右移时直接补上零即可，则该操作实现无符号的 A >> B[4:0]。

根据上述功能描述，可以比较容易地写出对应 verilog 代码。需要注意的是， verilog 语法中有算术右移运算符 `>>>` 。

```
module Alu(
        input [31:0] a_val,
        input [31:0] b_val,
        input [3:0] ctrl,
        output [31:0] result
    );
    `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
```

```
    wire[31:0] a, b, c, d, e, f, g, h, i, j;
    assign a = a_val + b_val; //ADD
    assign b = a_val - b_val; //SUB
    assign c = a_val << b_val[4:0]; //SLL
    assign d = (a_val[31] == 1 && b_val[31] == 0) ? 32'b1 :
               (a_val[31] == 0 && b_val[31] == 1) ? 32'b0 :
               (a_val < b_val) ? 32'b1 : 32'b0; //SLT
    assign e = (a_val < b_val); //SLTU
    assign f = a_val ^ b_val; //XOR
    assign g = a_val | b_val; //OR
    assign h = a_val & b_val; //AND
    assign i = ($signed(a_val) >>> $signed(b_val[4:0])); //SRA
    assign j = a_val >> b_val[4:0];

    assign result = (ctrl == `ALU_ADD) ? a:
                    (ctrl == `ALU_SUB) ? b:
                    (ctrl == `ALU_SLL) ? c:
                    (ctrl == `ALU_SLT) ? d:
                    (ctrl == `ALU_SLTU) ? e:
                    (ctrl == `ALU_XOR) ? f:
                    (ctrl == `ALU_OR) ? g:
                    (ctrl == `ALU_AND) ? h:
                    (ctrl == `ALU_SRA) ? i:
                    j;

endmodule
```

1.2 Comparator 模块

该模块主要是比较器，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。本次实验中我们实现如下操作：

- EQ：判断 A==B 是否为真
- NE：判断 A!=B 是否为真
- LT: 判断 A < B 是否为真，此时为有符号数
- LTU： 无符号数的情况下，判断 A < B 是否为真
- GE：判断 A>=B 是否为真，此时为有符号数
- GEU：判断 A>=B 是否为真，此时为无符号数

根据上述原理，可以比较容易写出 verilog 代码：

```
module Comparator(
        input [31:0] a_val,
        input [31:0] b_val,
        input [2:0] ctrl,
        output result
    );

    `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

    wire a,b;
    assign a = (a_val == b_val);
    assign b = ~(a_val == b_val);
    wire [31:0] sub;
    assign sub = a_val - b_val;
    assign result = (ctrl == `CMP_EQ) ? a :
                    (ctrl == `CMP_NE) ? b :
                    (ctrl == `CMP_LT) ? (sub[31] == 1) :
                    (ctrl == `CMP_LTU) ? (a_val < b_val) :
                    (ctrl == `CMP_GE) ? (sub[31] == 0) :
                    (ctrl == `CMP_GEU) ? (a_val >= b_val) :
                    1'b0;
endmodule
```

1.3 RegFile 模块

该模块我们实现寄存器的读写操作：

- 将指定数据内容 `i_data` 写入目标寄存器 `rd` 中。
- 读取指定源寄存器 `rs1` `rs2` 的值。

需要注意的是，对于写入操作，我们需要在时钟上升沿判断写使能信号是否为 1，并且需要保证目的寄存器不是 0 寄存器(0 号寄存器不允许修改)。

根据上述原理，可以得到对应的 verilog 代码：

```
module RegFile(
        `Core_DBG_Definitions
        input clk,
        input rst,
        input wen,
        input [4:0] rs1,
        input [4:0] rs2,
        input [4:0] rd,
        input [31:0] i_data,
        output [31:0] rs1_val,
        output [31:0] rs2_val
    );

     reg [31:0] regs [31:0];
    `RegFile_DBG_Assignment

    integer i;
    initial begin
        for(i = 0; i < 32; i = i + 1) begin
            regs[i] = 0;
        end
    end
```

```
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            for(i = 0; i < 32; i = i + 1) begin
                regs[i] = 0;
            end
        end
        else begin
            if(wen && rd != 0) begin
                regs[rd] <= i_data;
            end
        end
    end

    assign rs1_val = regs[rs1];
    assign rs2_val = regs[rs2];

endmodule
```

其中，我自定义 32 个 32 位寄存器，并将初始化为 0。随后在时序块中判断是否对目的寄存器进行赋值，在组合电路中实现读取操作。

二、实验结果与分析

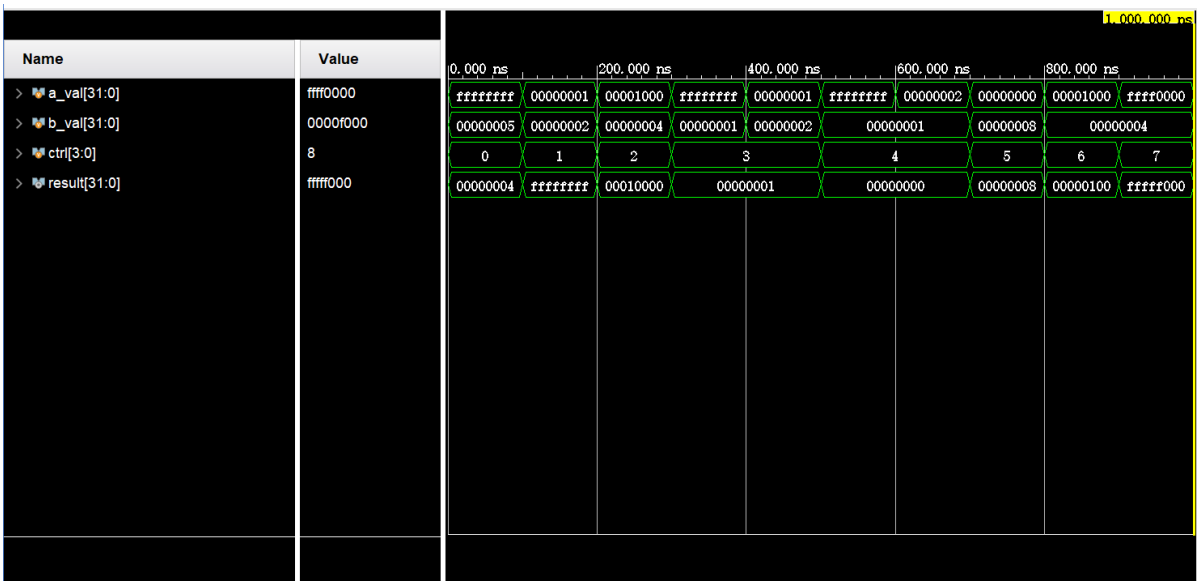
1. 通过仿真验证 ALU 模块的正确性。

在仿真代码中，我通过改变 ctrl 信号，遍历十种运算，查看结果的正确性。仿真代码如下：

```
module Alu_sim(

    );
    reg [31:0] a_val;
    reg [31:0] b_val;
    reg [3:0] ctrl;
    wire [31:0] result;
    Alu a0(
        .a_val(a_val),
        .b_val(b_val),
        .ctrl(ctrl),
        .result(result)
    );

    initial begin
        a_val = 32'b0;
        b_val = 32'b0;
        ctrl = 4'b0;
        a_val = 32'hffffffff;
        b_val = 32'd5; #100;

        ctrl = 4'd1;
        a_val = 32'b1;
        b_val = 32'd2; #100;

        ctrl = 4'd2;
```

```
        a_val = 32'h00001000;
        b_val = 32'd4; #100;

        ctrl = 4'd3;
        a_val = 32'hffffffff;
        b_val = 32'd1; #100;
        a_val = 32'd1;
        b_val = 32'd2; #100;

        ctrl = 4'd4;
        a_val = 32'hffffffff;
        b_val = 32'd1; #100;
        a_val = 32'd2;
        #100;

        ctrl = 4'd5;
        a_val = 32'b0;
        b_val = 32'd8; #100

        ctrl = 4'd6;
        a_val = 32'h00001000;
        b_val = 32'd4; #100;

        ctrl = 4'd7;
        a_val = 32'hffff0000;
        b_val = 32'd4; #100;

        ctrl = 4'd8;
        a_val = 32'hffff0000;
        b_val = 32'h0000f000; #100;

        ctrl = 4'd9;
        #100;
    end
endmodule
```

仿真波形如下：



- Ctrl 信号为 0 时，进行加法运算，-1 + 5 = 4 结果正确。

- Ctrl 信号为 1 时，进行减法运算，1 - 2 = -1 结果正确。
- Ctrl 信号为 2 时，进行逻辑左移操作，将 `a_val` 向左移 4 位，结果正确。
- Ctrl 信号为 3 时，进行 SLT 操作，可以看到结果正确。

可以看到，其余所有运算结果也都符合预期，这里不再赘述。

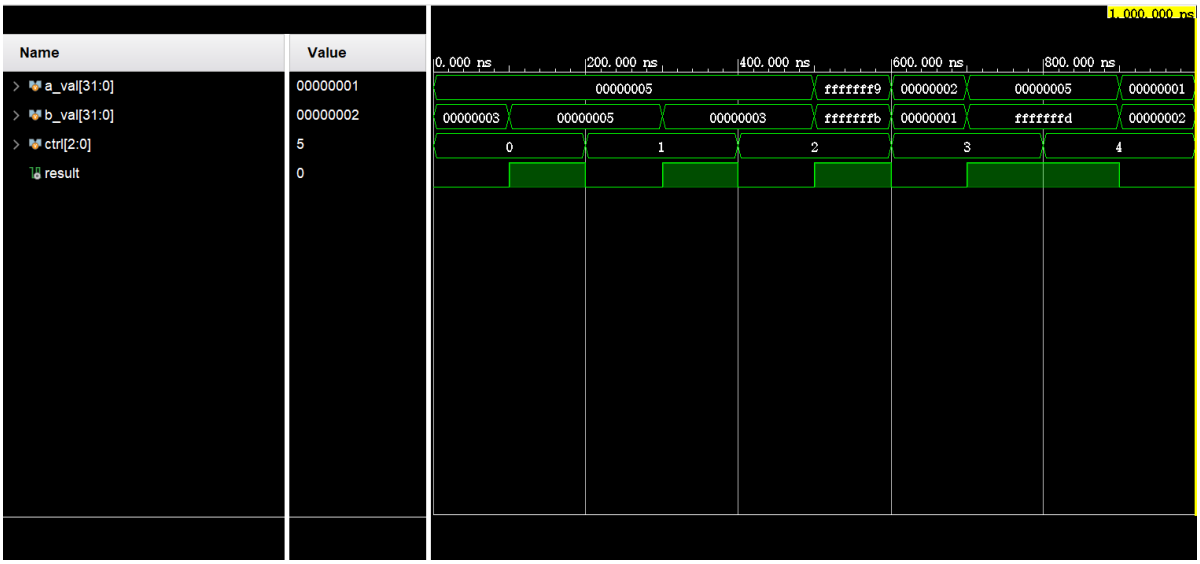
2. 通过仿真验证 Comparator 模块正确性。

在仿真代码中，我通过遍历 ctrl 信号，遍历五种操作，查看 result 的值判断是否符合预期。代码如下：

```
module Comparator_sim(

    );

    reg [31:0] a_val;
    reg [31:0] b_val;
    reg [2:0] ctrl;
    wire result;
    Comparator a0(
        .a_val(a_val),
        .b_val(b_val),
        .ctrl(ctrl),
        .result(result)
    );
    initial begin
        a_val = 32'd5;
        b_val = 32'd3;
        ctrl = 3'b0;
        #100;
        a_val = 32'd5;
        b_val = 32'd5;
        #100;
        ctrl = 3'd1;
        #100;
        a_val = 32'd5;
        b_val = 32'd3;
        #100;
        ctrl = 3'd2;
        #100
        a_val = -32'd7;
        b_val = -32'd5;
        #100
        ctrl = 3'd3;
        a_val = 32'd2;
        b_val = 32'd1;
        #100
        a_val = 32'd5;
        b_val = -32'd3;
        #100
        ctrl = 3'd4;
        #100
        a_val = 32'd1;
        b_val = 32'd2;
        #100;
```

```
        ctrl = 3'd5;
        #100
        a_val = 32'd5;
        b_val = 32'd3;
        #100;
    end
endmodule
```

仿真波形如下：



ctrl 信号从 0 到 4 分别代表相等，不等，LT, LTU, GT, GTU。可以看到仿真结果符合预期。

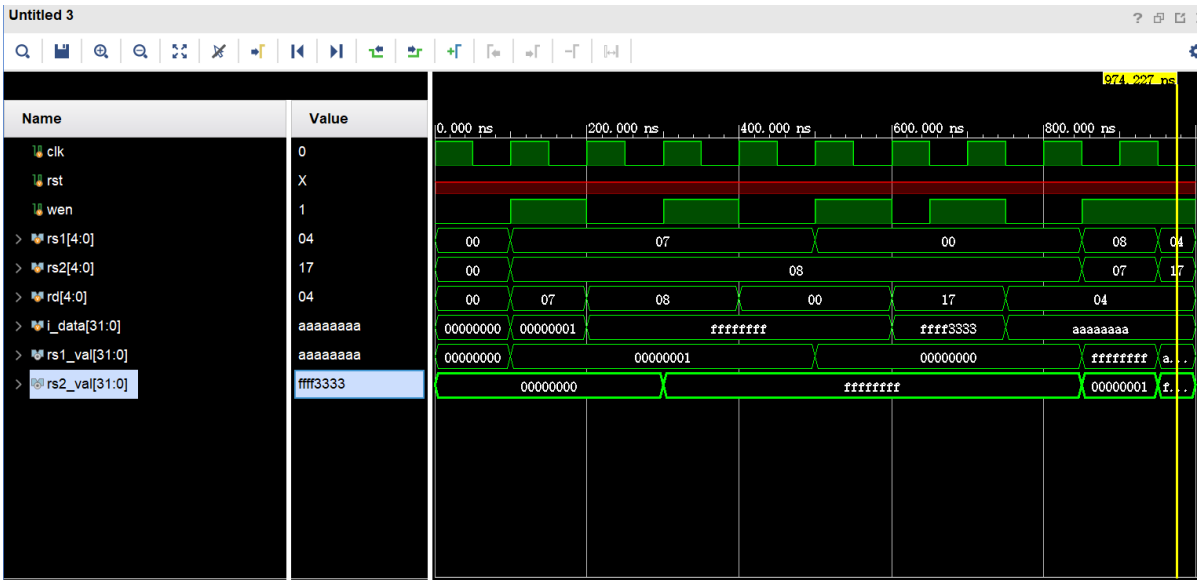
3. 通过仿真验证 RegFile 模块的正确性。

在仿真代码中，我首先验证不能往 0 寄存器里写入的逻辑；然后在使能信号为 1 时，往特定的寄存器里写入数据，并且在后续读取已写入数据的寄存器的值，以验证寄存器的读写功能。仿真代码如下：

```
module RegFile_sim(

    );
    reg clk;
    reg rst;
    reg wen;
    reg[4:0] rs1;
    reg[4:0] rs2;
    reg[4:0] rd;
    reg[31:0] i_data;
    wire[31:0] rs1_val;
    wire[31:0] rs2_val;

    RegFile uut (
        .clk(clk),
        .rst(rst),
        .wen(wen),
        .rs1(rs1),
        .rs2(rs2),
        .rd(rd),
        .i_data(i_data),
        .rs1_val(rs1_val),
        .rs2_val(rs2_val)
```

```
    );

    always begin
        clk <= 1'b1; #50;
        clk <= 1'b0; #50;
    end

    initial begin
        i_data = 32'b0;
        rs1 = 0;
        rs2 = 0;
        rd = 0;
        wen = 0;
        #100;
        i_data = 32'b1;
        rd = 5'd7;
        wen = 1;
        rs1 = 5'd7;
        rs2 = 5'd8;
        #100;
        wen = 0;
        rd = 5'd8;
        i_data = 32'hffffffff;
        #100;
        wen = 1;
        #100;
        wen = 0;
        rd = 5'b0;
        #100;
        wen = 1;
        rs1 = 5'b0;
        #100;
        wen = 0;
        rd = 5'd23;
        i_data = 32'hffff3333;
        #50;
        wen = 1;
        #100;
        wen = 0;
        rd = 5'd4;
        i_data = 32'haaaaaaaa;
        #100;
        wen  = 1;
        rs1 = 5'd8;
        rs2 = 5'd7;
        #100;
        rs1 = 5'd4;
        rs2 = 5'd23;
        #50;
    end

endmodule
```

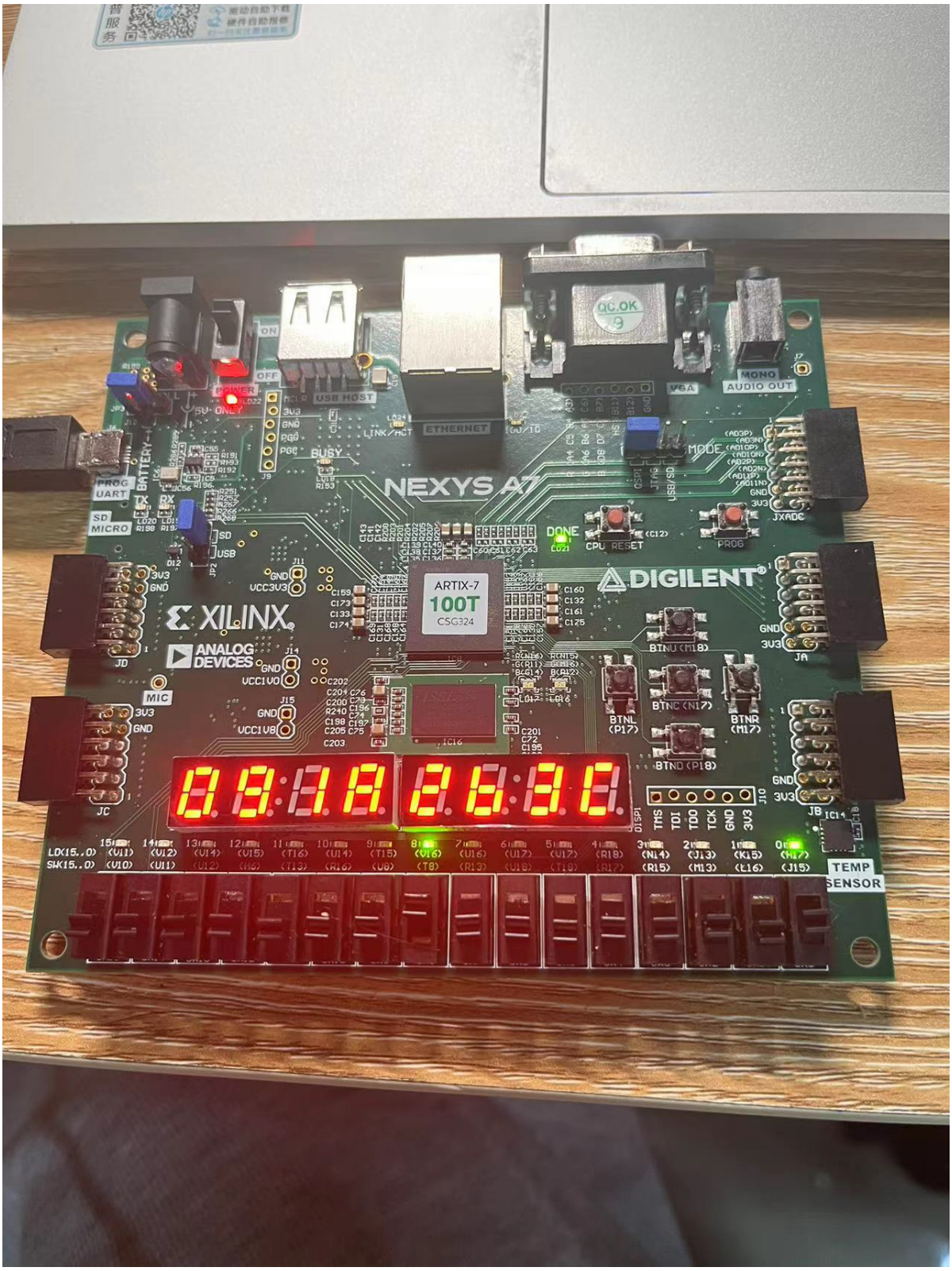
仿真代码如下：



可以看到， `rs1_val` 和 `rs2_val` 都能正确读取对应源寄存器的值。结果符合预期。

4. 上板结果

```
x01=0x12345678
x02=0xFE000000
x03=0x00000004
x04=0x00000001
x05=0x00000000
x06=0x12345670
x07=0x12345679
x08=0x00000008
x09=0x2468ACF0
x10=0x091A2B3C
x11=0xFFE00000
x12=0x1234567C
x13=0x12345674
x14=0x2468ACF0
x15=0x00000001
x16=0x00000000
x17=0x1234567C
x18=0x091A2B3C
x19=0xFFE00000
x20=0x1234567C
x21=0x00000008
x22=0x00000006
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
PC      =0x000000A0
INST    =0x01212023
MEMADDR=0x00000000
MEMDATA=0x00000000
```



可以看到，所有寄存器的值结果均正确，并且在 PC = 0x000000A0 时，x22 寄存器的值为 6，并且七段数码管显示数字正确。

三、思考题与心得

3.1 思考题

> 在实现比较器时，如何使用**一次减法**（或者说只使用一个减法器）就能实现两个 32 位数据的四种比较类型（LT，LTU，GE，GEU）？

首先将两个 32 位数据 (设为 `a`，`b`)进行有符号减法 （`a-b`），得到结果 `sub`。设 `a[31]` 和 `b[31]` 异或的结果为 `sign`，

则可以根据以上信息得到四种比较类型的结果：

1. `LT`：如果 `sign` 为 1， 则为 `a[31]` 的值，否则为 `sub[31]` 的值。(即 `sign` 为 1 说明 `a` 和 `b` 异号，因此谁是负数谁小； 若 `sign` 为 0 说明同号，则根据减法结果的最高位判断)
2. `LTU`：如果 `sign` 为 1， 则为 `b[31]` 的值，否则为 `sub[31]` 的值。（`sign` 为 1说明 `a` 和 `b` 最高位一个为 1 一个为 0，由于是无符号数，因此谁为 0 谁小；若 `sign` 为 0 则直接根据减法结果最高位判断）
3. `GE`：与 `LT` 的情况相反。
4. `GEU`：与 `LTU` 的情况相反。

3.2 心得

本次实验感觉相对比较简单，整个过程也相对比较顺利，唯一出现问题的就是 RegFile 模块的仿真，我最开始设置时钟变化周期为 50ns， 但是仿真模块使能信号与读写操作的变化周期是 10ns，导致变化周期过小，无法到达时钟上升沿，最后仿真结果读取的结果全为零。

总而言之，本次实验进行得比较顺利，可能是暴风雨前的平静/(ㄒoㄒ)/~~。下个实验就是单周期 CPU 啦，真正的挑战要来力。