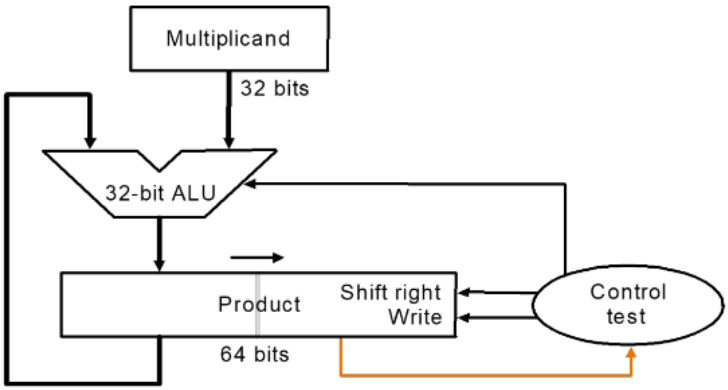
Lab2： 乘除法器、浮点加法器

3220106039 李瀚轩 2024.3.27

一、操作方法与实验步骤

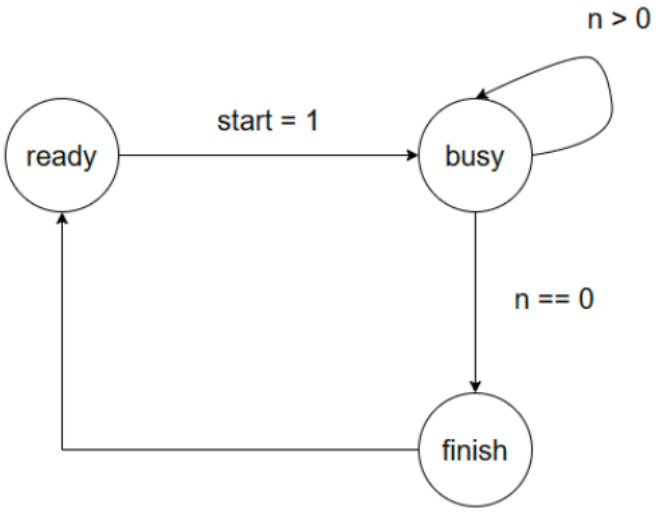
1.1 乘法器

1. 乘法器的原理。



如上图所示，我们首先将乘数置于 `Product` 寄存器的低 32 位中，高 32 位清零。随后执行 32 轮循环， 每次循环判断 `Product` 的最低位是否为 1， 若为 1， 则将 `Product` 的高 32 位与被乘数相加， 再将 `Product` 右移；若为 0，则仅作右移操作。

2. 通过有限状态机将乘法器算法原理转化为 verilog 代码。



这里展示了基本有限状态机的状态图。

- 对于 ready 状态，我们检测 start 的值，为 1 时进入 busy 状态(即开始运算)。
- 对于 busy 状态，不断执行运算操作，并用 n 记录迭代次数，每执行一次操作将 n 减 1， 当检测到 n 为 0 时进入 finish 状态。
- 对于 finish 状态， 输出模块工作完成的信息使外部电路接收后，回到 ready 状态。

具体 verilog 代码如下：

```
    reg ready_reg;
    reg finish_reg;
    reg [1:0] state;
    reg [3:0] n;

    initial begin
        ready_reg <= 1;
        finish_reg <= 0;
        n <= 0;
        state <= 0;
    end
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            state <= 2'b00;
        end
        else if ((state == 2'b00) && start) begin
            state <= 2'b01;
            ready_reg <= 1'b0;
        end
        else if((state == 2'b01) && (n == 0)) begin
            state <= 2'b10;
            finish_reg <= 1'b1;
        end
        else if(state == 2'b10) begin
            state <= 2'b00;
            ready_reg <= 1'b1;
            finish_reg <= 1'b0;
        end
    end
```

在 verilog 代码中，我们需要声明 state 寄存器来存储状态的编号， 并在 always 时序块中实现状态的转化。由于 always 块中左值不允许 wire 类型的变量，因此额外声明两个寄存器存放 ready 和 finish 信号。

3. 根据 1 中乘法器的原理可以实现乘法器的 verilog 代码。 需要注意的是我们实现的是有符号乘法器，因此需要通过乘数被乘数最高位异或判断乘积的符号，而在运算中我们只考虑无符号数(对操作数取绝对值)，对最后的结果进行处理即可。

该模块完整的 verilog 代码如下：

```
module Mul(
    input clk,
    input rst,
    input [15:0] multiplicand,
    input [15:0] multiplier,
    input start,
    output [31:0] product,
    output ready,
    output finish
```

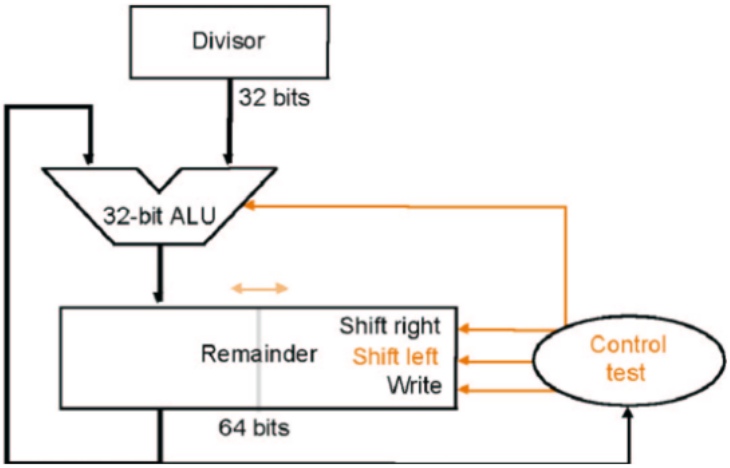
```
    );

    wire neg;
    reg [31:0] result;
    assign neg = multiplier[15] ^ multiplicand[15];
    wire [15:0] num1;
    wire [15:0] num2;
    assign num1 = (multiplicand[15] == 1) ? ~multiplicand + 1 : multiplicand;
    assign num2 = (multiplier[15] == 1) ? ~multiplier + 1 : multiplier;
    //assign result = {16'b0, multiplier};
    reg ready_reg;
    reg finish_reg;
    reg [1:0] state;
    reg [3:0] n;

    initial begin
        ready_reg <= 1;
        finish_reg <= 0;
        n <= 0;
        state <= 0;
    end
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            state <= 2'b00;
        end
        else if ((state == 2'b00) && start) begin
            state <= 2'b01;
            ready_reg <= 1'b0;
        end
        else if((state == 2'b01) && (n == 0)) begin
            state <= 2'b10;
            finish_reg <= 1'b1;
        end
        else if(state == 2'b10) begin
            state <= 2'b00;
            ready_reg <= 1'b1;
            finish_reg <= 1'b0;
        end
    end

    always@(posedge clk or posedge rst) begin
        if(rst)  result <= 32'b0;
        else if((state == 2'b00) && start) begin
            n <= 4'd15;
            result <= {16'b0, num2};
        end
        else if(state == 2'b01) begin
            if(result[0] == 1'b1) begin
                result = {result[31:16] + num1, result[15:0]};
            end
            result = result >> 1;
            n = n - 1'b1;
        end
    end
```

```
    assign product = (neg) ? ~result + 1 : result;
    assign ready = ready_reg;
    assign finish = finish_reg;

endmodule

// 我的这个markdown主题，在代码块中会把非阻塞赋值的字符自动转化成小于等于号/(ToT)/~~
// (==)看不到两个等号之间的空隙，(=)，(==)
```

如上述代码所示，当处于 `state==2'b00` 状态时，如果检测到 start 信号，则将 `n` 赋值为 15，并将 `result`（暂存乘积值的寄存器）高 16 位赋值为 0，低 16 位赋值成乘数，并进入 `2'b01` 状态，开始计算。当检测到 `n` 为 0 时，将 `finish` 置为 1，进入 `2'b10` 状态。而状态 `2'b10` 则完成对外部电路的信号输出，即将 `finish` 置为 0，`ready` 置为 1，回到初始状态。

1.2 除法器

1. 除法器的原理。



- 首先将被除数放在 `Remainder` 的低 32 位，高 32 位清零，并将整体左移一位。
- 开始循环，不断判断 `Remainder` 的高 32 位是否大于除数，若大于则减去，反之不操作，随后整体左移一位。若发生减法，则在左移后将最低位置为 1。
- 循环结束后，`Remainder` 低 32 位即为商，高 32 位右移一位后的结果为余数。
- 对于有符号数，我们仍然采用类似乘法的规则，即首先判断符号，在运算过程中用无符号数参与运算，最后处理结果。但是需要注意的是，余数的符号与被除数的符号一致。

2. 通过有限状态机将除法器算法原理转化为 verilog 代码。

具体思路与步骤与乘法器类似，这里不再赘述。

3. 根据 1 中的原理可以实现除法器的 verilog 代码。如下所示：

```
module Div(
        input clk,
        input rst,
        input start,
        input [15:0] dividend,
        input [15:0] divisor,
```

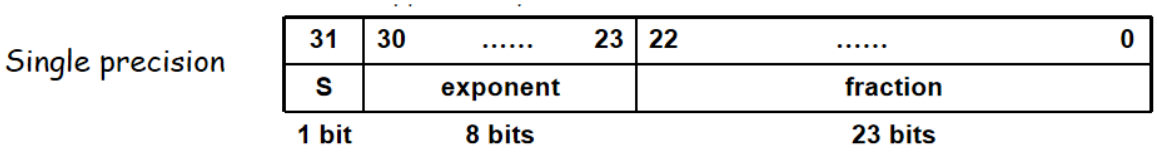
```
    output finish,
    output ready,
    output [15:0] quotient,
    output [15:0] remainder,
    output div_by_0
);

wire neg;
assign neg = dividend[15] ^ divisor[15];
wire [15:0] num1;
wire [15:0] num2;
assign num1 = (dividend[15] == 1) ? ~dividend + 1 : dividend;
assign num2 = (divisor[15] == 1) ? ~divisor + 1 : divisor;

reg [1:0] state;
reg [3:0] n;

reg ready_reg;
reg finish_reg;
reg [31:0] remainder_reg;
reg div_by_0_reg;

initial begin
    ready_reg <= 1;
    finish_reg <= 0;
    n <= 0;
    state <= 0;
end

always@(posedge clk or posedge rst) begin
    if(rst) begin
        state <= 2'b00;
    end
    else if ((state == 2'b00) && start) begin
        if(divisor == 0) begin
            state <= 2'b10;
            finish_reg <= 1'b0;
        end
        else begin
            state <= 2'b01;
            ready_reg <= 1'b0;
        end
    end
    else if((state == 2'b01) && (n == 0)) begin
        state <= 2'b10;
        finish_reg <= 1'b1;
    end
    else if(state == 2'b10) begin
        state <= 2'b00;
        ready_reg <= 1'b1;
        finish_reg <= 1'b0;
    end
end
```

```
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            remainder_reg <= 32'b0;
        end
        else if(state == 2'b00 && start) begin
            if(divisor == 0) begin
                remainder_reg <= 32'b0;
                div_by_0_reg <= 1;
            end
            else begin
                n <= 4'd15;
                div_by_0_reg <= 0;
                remainder_reg <= {15'b0, num1, 1'b0};
            end
        end
        else if(state == 2'b01) begin
            n = n - 1'b1;
            if(remainder_reg[31:16] >= num2) begin
                remainder_reg = {remainder_reg[31:16] - num2,
remainder_reg[15:0]} << 1;
                remainder_reg = remainder_reg + 1'b1;
            end
            else begin
                remainder_reg = remainder_reg << 1;
            end
        end
    end

    assign quotient = (neg) ? ~remainder_reg[15:0] + 1 : remainder_reg[15:0];
    wire [15:0] abs_remainder;
    assign abs_remainder = remainder_reg[31:16] >> 1;
    assign remainder = (dividend[15] == 1) ? ~abs_remainder + 1 :
abs_remainder;

    assign div_by_0 = div_by_0_reg;
    assign ready = ready_reg;
    assign finish = finish_reg;
endmodule
```

与乘法器不同的是，除法器需要判断除数为 `0` 的非法情况。因此在代码中进行 `divisor == 0` 的特判，若成立，则输出一个 `div_by_0` 的信号。

1.3 浮点加法器

1. 浮点数在计算机中的表示。

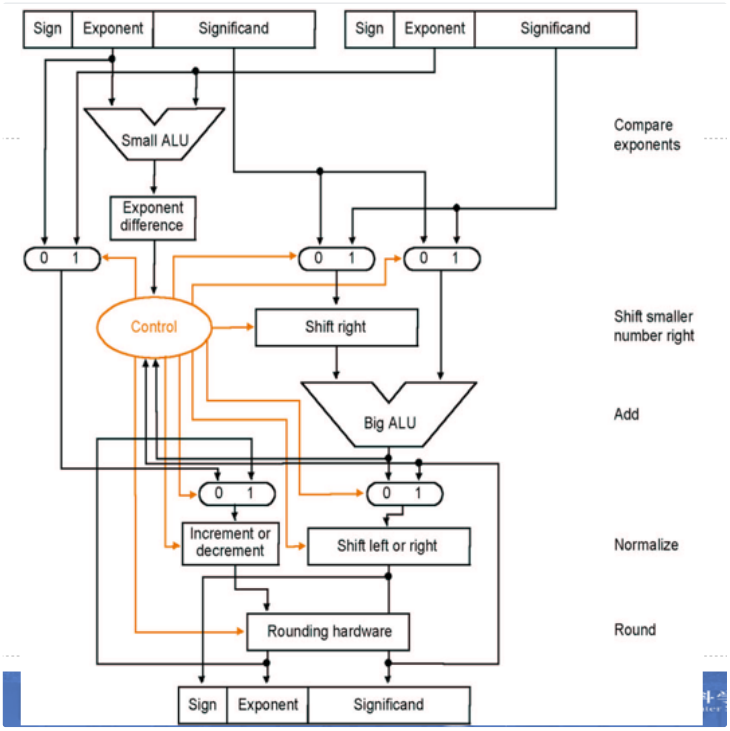


类似科学计数法。最高位是符号位，`[30:23]` 为指数部分，并加上了偏移量 `127` 。`[22:0]` 为尾数部分(significand)，舍去了规范的 leading 1。

因此将二进制浮点数转化为十进制数的公式为：

$$(-1)^{sign} \cdot (1 + significand) \cdot 2^{exponent-bias}$$

2．浮点数的加法原理。



- 首先比较两个浮点数的指数，取较大的指数作为结果指数，较小的对尾数进行移位使得指数对齐。
- 对两个尾数进行加法。
- 检测得到的结果是否满足规范化要求，若不满足则进行移位操作。
- 将结果写入，完成运算。

3．实现浮点加法器的 verilog 代码。代码如下：

```
module AddFloat(
    input clk,
    input rst,
    input start,
    input [31:0] a,
    input [31:0] b,
    output finish,
    output [31:0] result
    );

    reg [2:0] state;
    reg [7:0] a_exp, b_exp, result_exp;
    reg [24:0] a_mant, b_mant, result_mant;
    reg sign;
    reg [31:0] result_reg;
    reg finish_reg;
    reg running;

    wire neg;
    assign neg = a[31] ^ b[31];

    localparam
```

```
        state0 = 3'b000,
        state1 = 3'b001,
        state2 = 3'b010,
        state3 = 3'b011,
        state4 = 3'b100,
        state5 = 3'b101;

    always@(posedge clk or posedge rst) begin
        if(rst) begin
            result_reg <= 0;
            finish_reg <= 0;
            running <= 0;
        end
        else if(start) begin
            a_exp <= a[30:23];
            b_exp <= b[30:23];
            a_mant <= {2'b01, a[22:0]};
            b_mant <= {2'b01, b[22:0]};
            state <= state0;
            finish_reg <= 0;
            running <= 1;
        end
        else if(running && !finish_reg) begin
            case(state)
                state0: begin
                    if(a_exp == b_exp && a_mant == b_mant && neg) begin
                        sign <= 0;
                        result_exp <= 0;
                        result_mant <= 0;
                        state <= state4;
                    end
                    else if(a_exp == 0 || a_exp == 8'hFF) begin
                        sign <= a[31];
                        result_exp <= a_exp;
                        result_mant <= a_mant;
                        state <= state4;
                    end
                    else if(b_exp == 0 || b_exp == 8'hFF) begin
                        sign <= b[31];
                        result_exp <= b_exp;
                        result_mant <= b_mant;
                        state <= state4;
                    end
                    else begin
                        state <= state1;
                    end
                end

                state1: begin
                    if(a_exp == b_exp) begin
                        state <= state2;
                    end
                    else if(a_exp > b_exp) begin
                        b_exp <= b_exp + 1;
```

```
                b_mant[24:0] <= {1'b0, b_mant[24:1]};
            end
            else begin
                a_exp <= a_exp + 1;
                a_mant[24:0] <= {1'b0, a_mant[24:1]};
            end
        end

        state2: begin
            if(!neg) begin
                sign <= a[31];
                result_mant <= a_mant + b_mant;
            end
            else begin
                if(a[31]) begin
                    if(a_mant > b_mant) begin
                        sign <= 1;
                        result_mant <= a_mant - b_mant;
                    end
                    else begin
                        sign <= 0;
                        result_mant <= b_mant - a_mant;
                    end
                end
                else begin
                    if(a_mant > b_mant) begin
                        sign <= 0;
                        result_mant <= a_mant - b_mant;
                    end
                    else begin
                        sign <= 1;
                        result_mant <= b_mant - a_mant;
                    end
                end
            end
            result_exp <= a_exp;
            state <= state3;
        end

        state3: begin
            if(result_mant[24]) begin
                result_exp <= result_exp + 1;
                result_mant <= result_mant >> 1;
                state <= state3;
            end
            else if(result_mant[23] == 0) begin
                result_exp <= result_exp - 1;
                result_mant <= result_mant << 1;
                state <= state3;
            end

            else state <= state4;
        end
```

```
                state4: begin
                    result_reg <= {sign, result_exp, result_mant[22:0]};
                    finish_reg <= 1;
                    running <= 0;
                end

            endcase
        end
    end

    assign result = result_reg;
    assign finish = finish_reg;

endmodule
```

感觉伪代码里的实现不太能清晰展现浮点加法器状态变化的过程，我还是采用有限状态机的方法来实现，具体分为 5 个状态：

- state0：初始状态，首先判断是否存在相反数相加或者非规范数的存在，如果有则进行处理，处理完毕后直接进入 state4 。若没有则进入 state1。
- state1：对齐指数。完成后进入状态 state2。
- state2：根据符号判断对尾数进行相加或相减。完成后进入状态 state3。
- state3：判断结果是否符合规范化要求，若不符合则进行移位，符合要求后进入 state4。
- state4：输出结果后回到初始状态。

二、实验结果与分析

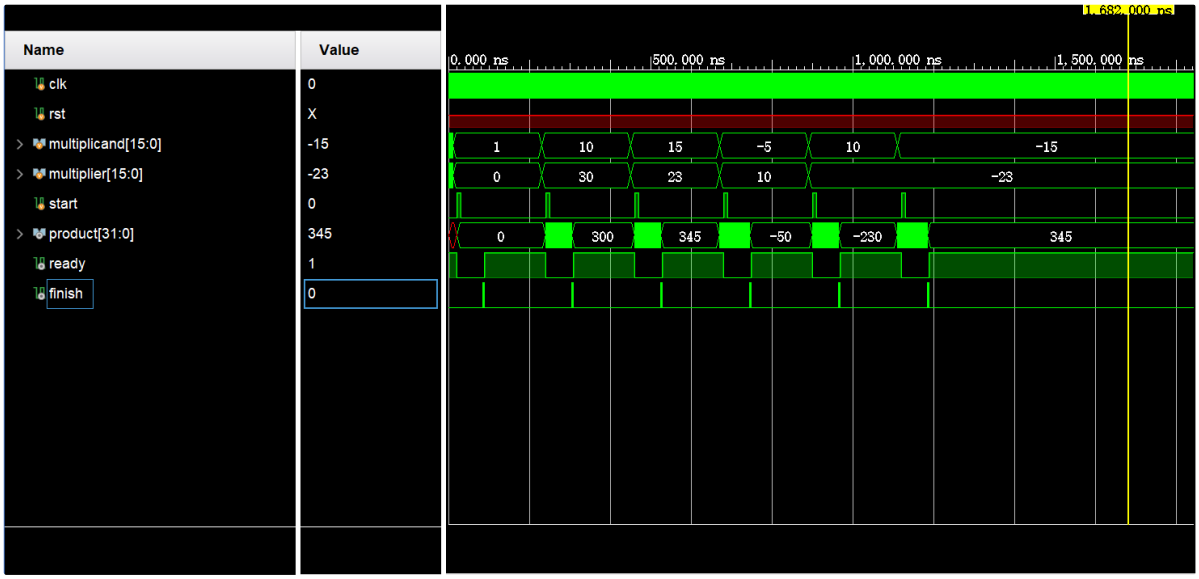
1. 通过仿真验证乘法器的正确性。

在仿真代码中，我设置四组情况：正数乘正数、正数乘负数、负数乘正数、负数乘负数。代码如下：

```
module Mul_sim(

    );

    reg clk;
    reg rst;
    reg [15:0] multiplicand;
    reg [15:0] multiplier;
    reg start;
    wire [31:0] product;
    wire ready;
    wire finish;

    Mul m0 (
        .clk(clk),
        .rst(rst),
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .start(start),
```

```
        .product(product),
        .ready(ready),
        .finish(finish)
    );

    always begin
        clk <= 1'b1; #2;
        clk <= 1'b0; #2;
    end

    initial begin
        start = 0;
        #10;
        multiplicand = 15'd1;
        multiplier = 15'd0;
        #10 start = 1;
        #10 start = 0;
        #200;

        multiplicand = 15'd10;
        multiplier = 15'd30;
        #10 start = 1;
        #10 start = 0;
        #200;

        multiplicand = 15'd15;
        multiplier = 15'd23;
        #10 start = 1;
        #10 start = 0;
        #200;

        multiplicand = -15'd5;
        multiplier = 15'd10;
        #10 start = 1;
        #10 start = 0;
        #200;

        multiplicand = 15'd10;
        multiplier = -15'd23;
        #10 start = 1;
        #10 start = 0;
        #200;

        multiplicand = -15'd15;
        multiplier = -15'd23;
        #10 start = 1;
        #10 start = 0;
        #200;

    end
endmodule
```

仿真波形如下：



可以看到运算结果全部正确，并且可以看到在 `start` 信号为高电平后，`ready` 信号变为低电平，此时模块进行计算。计算完成后 `finish` 信号变为高电平，该周期中模块即可输出正确结果，一周期后 `finish` 变回低电平，`ready` 重新变回高电平，可以说明交互信号正确。仿真波形中 `product` 有段很绿的片段，代表正在进行运算。

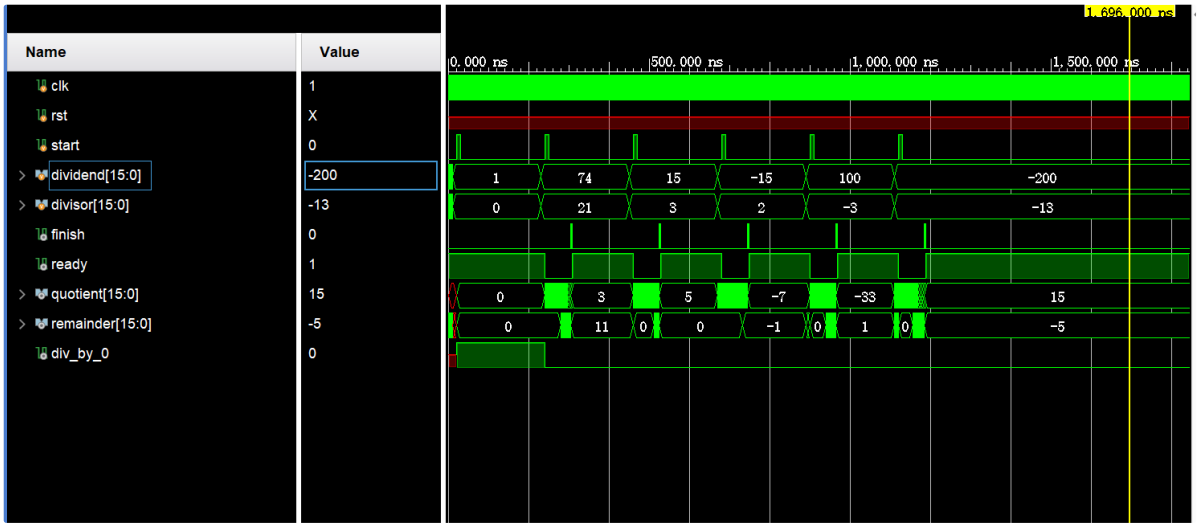
2．通过仿真验证除法器的正确性。

在仿真代码中，我设置 5 种情况：正数除以正数，正数除以负数，负数除以正数，负数除以负数，除以零。仿真代码如下：

```
module Div_sim(

    );

    reg clk;
    reg rst;
    reg start;
    reg [15:0] dividend;
    reg [15:0] divisor;
    wire finish;
    wire ready;
    wire [15:0] quotient;
    wire [15:0] remainder;
    wire div_by_0;

    Div d0 (
        .clk(clk),
        .rst(rst),
        .start(start),
        .dividend(dividend),
        .divisor(divisor),
        .finish(finish),
        .ready(ready),
        .quotient(quotient),
        .remainder(remainder),
        .div_by_0(div_by_0)
    );
```

```
    always begin
        clk <= 1'b1; #2;
        clk <= 1'b0; #2;
    end

    initial begin
        start = 0;
        #10;
        dividend = 15'd1;
        divisor = 15'd0;
        #10 start = 1;
        #10 start = 0;
        #200;

        dividend = 15'd74;
        divisor = 15'd21;
        #10 start = 1;
        #10 start = 0;
        #200;

        dividend = 15'd15;
        divisor = 15'd3;
        #10 start = 1;
        #10 start = 0;
        #200;

        dividend = -15'd15;
        divisor = 15'd2;
        #10 start = 1;
        #10 start = 0;
        #200;

        dividend = 15'd100;
        divisor = -15'd3;
        #10 start = 1;
        #10 start = 0;
        #200;

        dividend = -15'd200;
        divisor = -15'd13;
        #10 start = 1;
        #10 start = 0;
        #200;
    end

endmodule
```

仿真波形如下：



可以看到，运算结果全部正确，交互信号波形也正确。可以看到当除数为 0 时 `div_by_0` 信号为 1，并且所有结果中余数的符号与被除数相同。

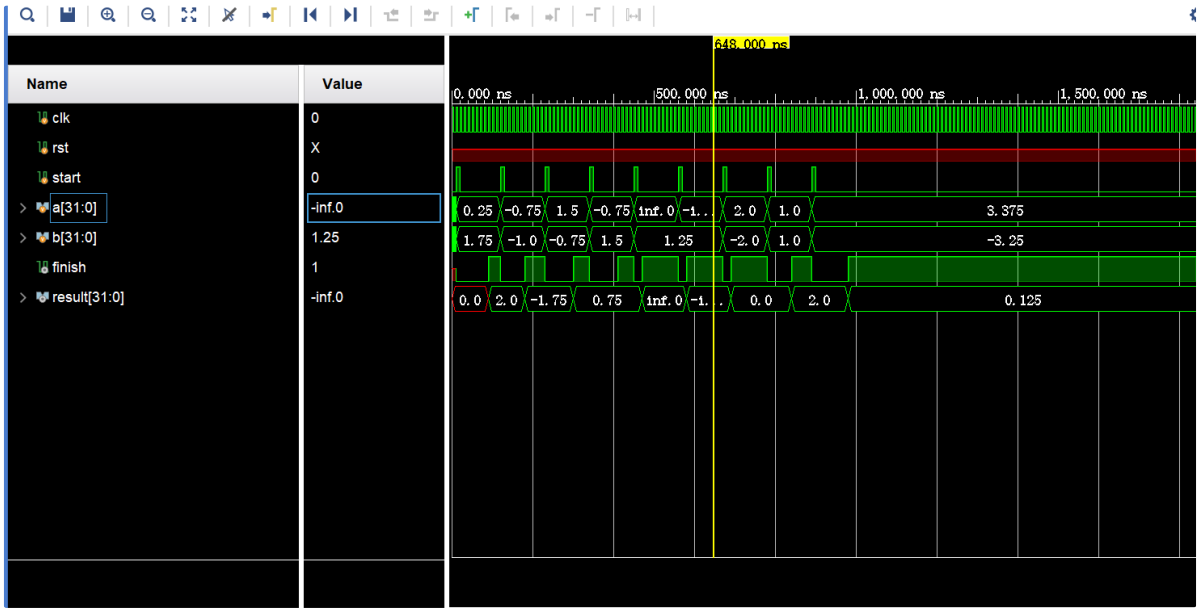
3. 通过仿真验证浮点加法器的正确性。

在仿真代码中，我设置了如下内容：正浮点数加正浮点数、正浮点数加负浮点数、负浮点数加正浮点数、负浮点数加负浮点数、正负无穷分别和一个非无穷的数做加法，结果仍为无穷、两个互为相反数的浮点数加和为 0 、两个相同的数相加，测试进位、两个数相加会产生退位。仿真代码如下：

```
module AddFloat_sim(

    );
    reg clk;
    reg rst;
    reg start;
    reg [31:0] a;
    reg [31:0] b;
    wire finish;
    wire [31:0] result;

    AddFloat a0 (
        .clk(clk),
        .rst(rst),
        .start(start),
        .a(a),
        .b(b),
        .finish(finish),
        .result(result)
    );

    always begin
        clk <= 1'b1; #5;
        clk <= 1'b0; #5;
    end
    initial begin
        start = 0;
        #10;
        a = 32'h3E800000;//0.25
        b = 32'h3FE00000;//1.75
```

```
        start = 1; #10;
        start = 0; #100;

        a = 32'hBF400000;//-0.75
        b = 32'hBF800000;//-1.0
        start = 1; #10;
        start = 0; #100;

        a = 32'h3FC00000; //1.50
        b = 32'hBF400000;//-0.75
        start = 1; #10;
        start = 0; #100;

        a = 32'hBF400000; // -3.7
        b = 32'h3FC00000; // 1.25
        start = 1; #10;
        start = 0; #100;

        a = 32'h7F800000;
        b = 32'h3fa00000;
        start = 1; #10;
        start = 0; #100;

        a = 32'hFF800000;
        b = 32'h3fa00000;
        start = 1; #10;
        start = 0; #100;

        a = 32'h40000000; //1
        b = 32'hC0000000; //-1
        start = 1; #10;
        start = 0; #100;

        a = 32'h3F800000;
        b = 32'h3F800000;
        start = 1; #10;
        start = 0; #100;

        a = 32'h40580000;
        b = 32'hC0500000;
        start = 1; #10;
        start = 0; #100;

    end

endmodule
```

仿真波形如下图所示：



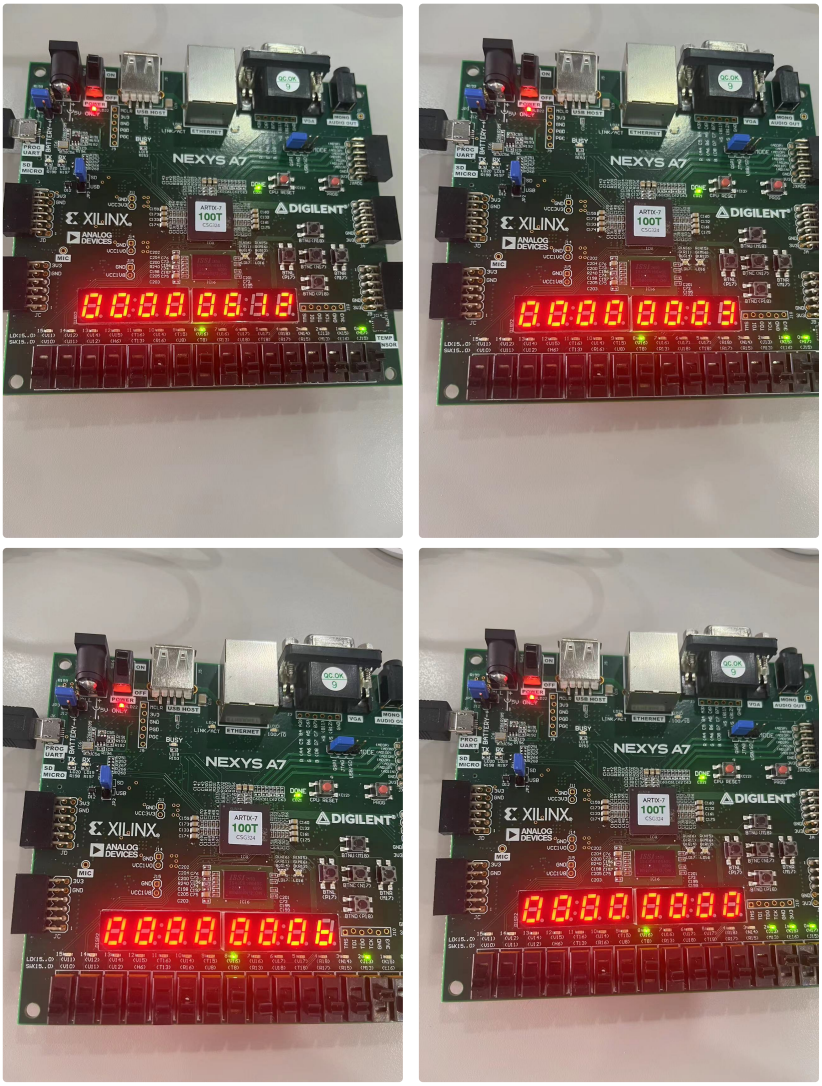
可以看到，所有运算结果全部正确。其中正负无穷与规范数相加仍为正负无穷，相反数相加结果为 0， 进位退位的正确性也得到验证。

4. 上板验证

上板结果的正确性已经在验收时得到验证，这里不一一展示所有截图，仅选取部分结果展示。

```
x01=0x00000612
x02=0x00000003
x03=0x0000000B
x04=0x00000000
x05=0x00000000
x06=0x00000000
x07=0x00000000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
A       =0x0000004A
B       =0x00000015
MEMADDR=0x00000000
MEMDATA=0x00000000
```

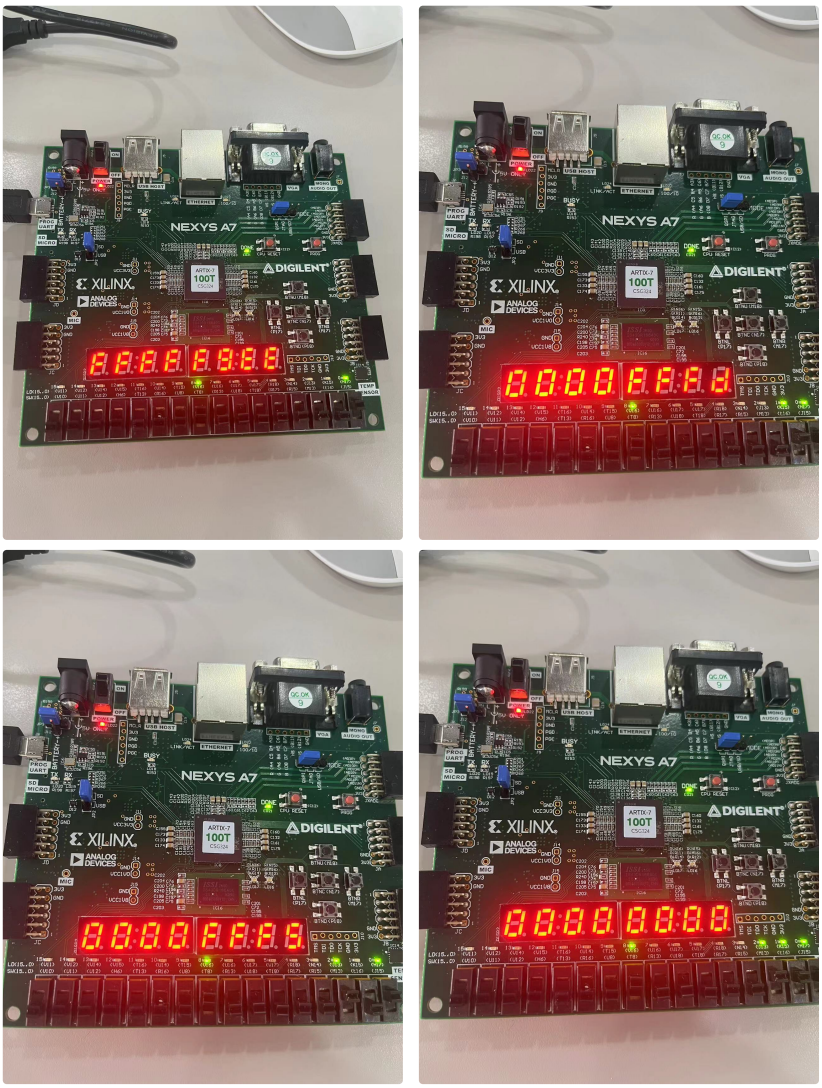
对于操作数 A,B 我们可以将其十六进制表示转化为十进制，方便计算：A = 74， B = 21。因此 A*B = 1554 = 0x00000612, A / B 的商为 3， 余数为 11 = 0X0000000B。上板结果如下图所示：



当 `SW[2:1]` 为 00, 01, 10, 11时，七段数码管上的结果分别对应乘法、除法的商、除法的余数、除数是否为0。 可以看到结果全部正确。

```
x01=0xFFFFF9EE
x02=0x0000FFFD
x03=0x0000FFF5
x04=0x00000000
x05=0x00000000
x06=0x00000000
x07=0x00000000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
A      =0x0000FFB6
B      =0x00000015
MEMADDR=0x00000000
MEMDATA=0x00000000
```

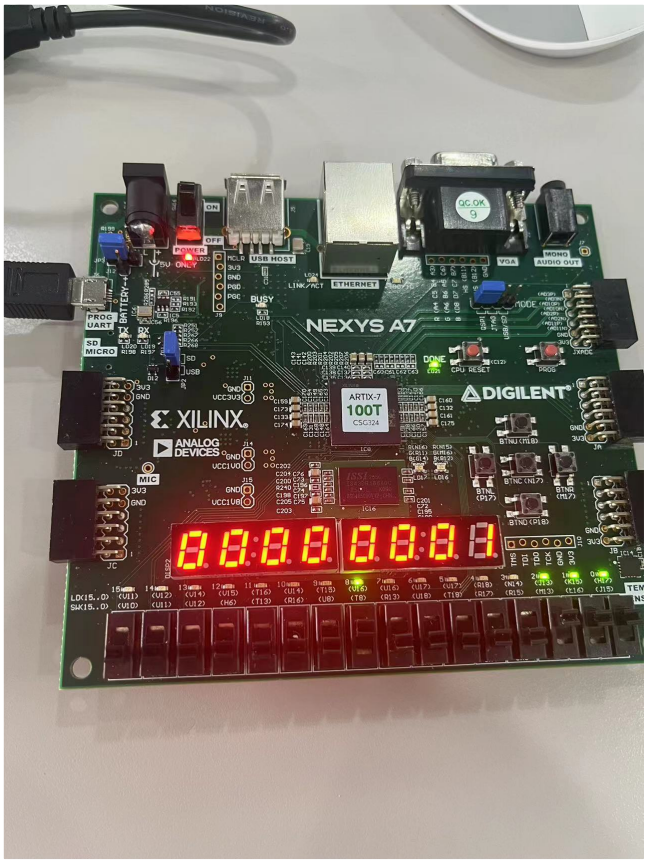
对于这种情况，我们也可以采取类似上述的方法得到正确结果。上板结果如下图所示：



可以看到上板结果正确。

```
x01=0x00000000
x02=0x00000000
x03=0x00000000
x04=0x00000001
x05=0x00000000
x06=0x00000000
x07=0x00000000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
A       =0x00000001
B       =0x00000000
MEMADDR=0x00000000
MEMDATA=0x00000000
```

在这种情况下，除数为0，因此我们应该得到 `div_by_0` 为 1 的输出，并当 `SW[2:1]==2'b11` 时，在七段数码管上看到 `00000001` 。上板结果如下图所示：



可以看到结果正确。

三、思考题与心得

3.1 思考题

> 在设计乘除法器和外部交互的接口时，为何需要单独设置一个 finish 状态，而不是合并到 ready 状态中，在恢复到 ready 状态时表示上次计算完成？

- 首先理一下正确的交互接口：当计算完成时 finish 信号为 1， 保持一个周期后 finish 恢复为 0，ready 置为 1。此时如果 start 信号为 1，则将 ready 置为 0， 电路开始计算。
- 我觉得原因可能是倘若没有 finish 信号，那么计算完成后 ready 就从 0 变为 1。 倘若在计算完成的这个周期内 start 信号变为 1，那么在下一个周期电路将直接开始新一轮的计算。但是外部电路本来应该在下一个周期读取计算的结果，这种情况下，外部电路读取到的是新一轮计算经过一轮操作后的结果，造成读取错误。

3.2 心得

在本次实验中，我巩固了理论课上乘除法器、浮点加法器的硬件实现原理，并且学习了通过有限状态机设计算法的 verilog 代码的方法，收获颇丰！

但是在实验过程中，我还是遇到了不少困难，感觉主要还是对 verilog 特性不那么熟悉：

- 阻塞赋值与非阻塞赋值：简单来说，非阻塞方法是在整个块结束时完成赋值操作，即左值不是马上改变；而阻塞方法时在赋值语句执行完后立刻完成赋值操作。在写乘法器时，刚开始如下代码块我使用的是非阻塞赋值，这样的话高位加被乘数的操作就不会执行，只会执行移位操作。

```
else if(state == 2'b01) begin
        if(result[0] == 1'b1) begin
            result = {result[31:16] + num1, result[15:0]};
        end
        result = result >> 1;
        n = n - 1'b1;
    end
```

- verilog 中不支持一条语句两个操作。在写除法器时，如下的代码块所示，我刚开始写成 `remainder_reg = {remainder_reg[31:16] - num2, remainder_reg[15:0]} << 1 + 1'b1` 。但是仿真结果全部为 0 。之后才知道得写成两个语句！

```
if(remainder_reg[31:16] ≥ num2) begin
                remainder_reg = {remainder_reg[31:16] - num2,
remainder_reg[15:0]} << 1;
                remainder_reg = remainder_reg + 1'b1;
            end
```

- 在 verilog 中一个寄存器的赋值不能出现在两个 always 块里面。我在上板的时候发现一个诡异的现象，就是仿真结果正确但是板上除法显示全为 0 。刚开始我一头雾水，还在查看除法器的逻辑是否出现问题，但是之后分析觉得既然仿真没问题，那应该是交互信号出问题。但是我自己还是没找出来，询问助教学长才知道是因为我在两个 always 块中对同一个寄存器赋值。

三面三个问题分别耗费了我巨大的时间，找 bug 的过程中也吐槽为什么 verilog 没有个类似 gdb 的 debug 工具/(ToT)/~~。但是在正确实现之后，还是蛮有成就感的，也感觉正在慢慢熟悉和适应 verilog 硬件代码的独特特性。

总而言之，本次实验我学到了很多，收获满满！