浙江大學
ZHEJIANG UNIVERSITY

计算机组成 实验报告

姓　　名:	NoughtQ
学　　号:	1145141919810
专　　业:	计算机科学与技术
课程名称:	计算机组成
指导教师:	赵莎
实验地点:	东四 509 教室

2024 年 10 月 3 日

Lab3：复杂操作实现——乘法器、除法器、浮点加法

一、 实验目的和要求

1. 复习二进制加减、乘除的基本法则
2. 掌握补码的基本原理和作用
3. 了解浮点数的表示方法及加法运算法则
4. 进一步了解计算机系统的复杂运算操作

二、 实验设备和环境

- 操作系统：Windows 11
- 开发工具：Xilinx VIVADO 2023.2

三、 实验内容和原理

3.1 乘法器

乘法器的基本原理是模拟手工的乘法运算，不过这样得到的乘法器浪费较多的空间，且效率不高。经过一定的改进后，我们得到了下面的 64 位的乘法器，即理论课上介绍过的 V3 乘法器：

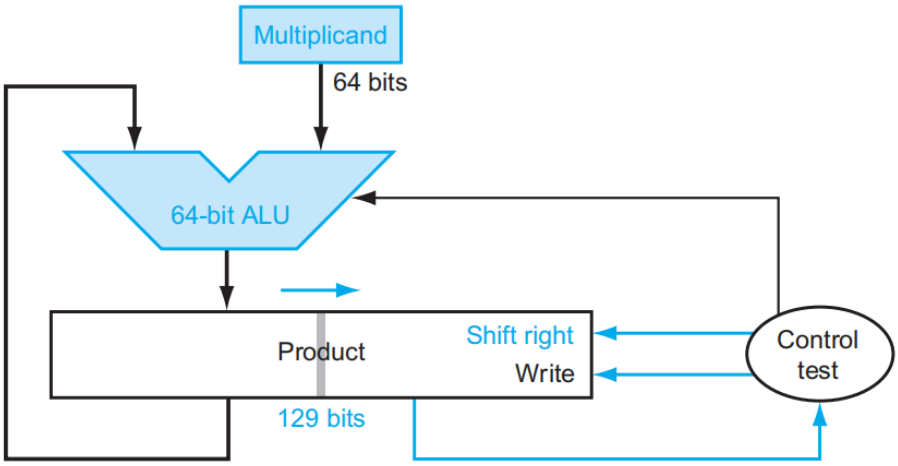

Figure 1: 乘法器

该乘法器包括一个 64 位的被乘数(multiplicand)寄存器，一个 64 位的 ALU（算术逻辑单元），一个 128 位的乘积(product)寄存器，以及一个控制器。对于乘积寄存器，在初始状态下，左半边存放乘积，右半边存放乘数(multiplier)。

它的工作流程为：

1. 检查乘数的最低位数字
2. 如果是 1，就将被乘数加到乘积的寄存器内的左半边；如果是 0，继续下一步
3. 将乘积寄存器的值右移一位
4. 如果已经完成 64 次计算，完成整个乘法运算；否则的话回到第一步继续计算

本次实验中，我们需要设计的是 32 位的乘法器，原理与上面的乘法器一致，只需将寄存器和 ALU 的存储空间以及循环次数削减一半即可。

3.2 除法器

除法器的基本原理也是模拟手工的除法运算，不过这样得到的除法器浪费较多的空间，且效率不高。经过一定的改进后，我们得到了下面的 64 位的除法器：

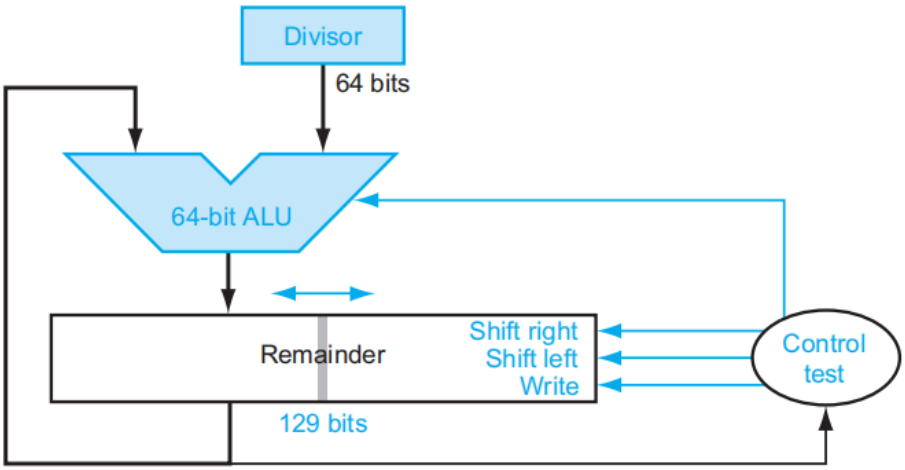

Figure 2: 除法器

该乘法器包括一个 64 位的除数(divisor)寄存器，一个 64 位的 ALU（算术逻辑单元），一个 128 位的余数(remainder)寄存器，以及一个控制器。对于余数寄存器，在初始状态下，左半边存放被除数(dividend)，右半边存放商(quotient)。

它的工作流程为：

1. 用余数寄存器左半边的值（初始为被除数）减去除数寄存器的值，结果放在余数寄存器的左半边
2. 检查余数寄存器左半边的值
   - 如果其值 ≥ 0（即符号位（最高位）的数字为 0），将余数寄存器里的值左移一位，并将最右端的位设成 1
   - 如果其值 < 0，则恢复余数寄存器左半边的值；然后将余数寄存器里的值左移一位，并将最右端的位设成 0
4. 如果上面的步骤已经重复了 64 次，中止整个流程，否则返回第一步

本次实验中，我们需要设计的是 32 位的除法器，原理与上面的除法器一致，只需将寄存器和 ALU 的存储空间以及循环次数削减一半即可。

3.3 浮点加法器

本次实验完成的是 32 位浮点数的加法。首先来了解一下 32 位浮点数的构成：

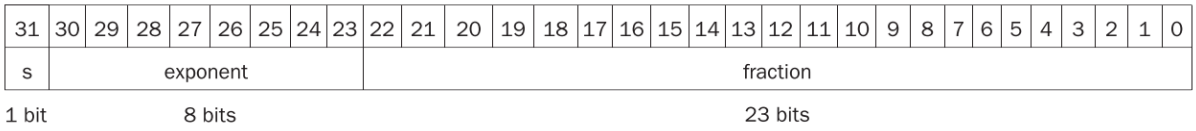

Figure 3: 32 位浮点数的结构

- 最高位为**符号位** s
- 之后 8 位为**指数(exponent)位** E
- 最后低 23 位为**尾数(fraction)位** F

用数学语言表述为:

$$(-1)^S \times F \times 2^E$$

我们讨论的是 IEEE 754 标准下的浮点数，该标准下的浮点数还有以下特征:

- 原本 F 的格式形如 $1.XXXX...$，也就是说小数点左边的数字一定是 1，因此在实际使用中我们不会将这个 1 存入尾数部分，而只记录小数点右边的部分
- 指数也有可能是负数，为了让负指数也能像正指数一样比较，我们引入了偏移值(biased notation)的概念: 新的指数 = 实际指数（补码）- 偏移值（在 32 位的浮点数中，偏移值 = 127）

所以浮点数的实际表示法为:

$$(-1)^S \times (1 + F) \times 2^{E - \text{bias}}$$

有了浮点数的基础知识，那么下面就简单介绍一下浮点数加法的流程:

1. **对齐 (alignment)**: 比较两个数的指数部分，将较小的数往右移，直到两个数指数部分的数量级一致为止
2. **加法 (addition)**: 将两个数的有效数字 (significand) 相加
3. **规范化 (normalization)**: 使结果规范化，要么右移来增加指数，要么左移减小指数

> 注
>
> 前往下一步之前还要判断一下数字是否溢出，如果是的话抛出异常，否则继续下一步

4. **舍入 (rounding)**: 如果舍入后结果变得不规范，则需要返回到第 3 步再做调整，否则的话完成整个加法运算

四、 实验步骤

> 注意
>
> 我用 Typst 撰写本篇报告，但 Typst 会将代码块中的 <= 自动转化为 ⩽，但我在代码中多次用到这类非阻塞赋值的符号，因此可能会影响到代码的阅读，请见谅~

4.1 乘法器

在 Vivado 中打开已经给出的工程文件 `mul32.xpr`，找到 Verilog 文件 `mul32.v`，可以看到已经给出了相应接口。接下来便根据上面的实验原理补充下列代码：

```
`timescale 1ns / 1ps

module mul32(
  input clk,
  input rst,
  input[31:0] multiplicand,
  input[31:0] multiplier,
  input start,
  output reg[63:0] product,
  output reg finish
);

  reg state;      // 标记是否位于计算状态
  reg[4:0] cnt;   // 循环计数，31 + 1 ⇒ 0

  // 初始赋值
  initial begin
    product ⩽ 0;
    finish ⩽ 0;
    state ⩽ 0;
    cnt ⩽ 0;
  end

  always @(posedge clk) begin
    // 当复位或重新开始时，重新初始化
    if (rst || start && ~state) begin
        product ⩽ {32'b0, multiplier};
        state ⩽ 1;
        finish ⩽ 0;
        cnt ⩽ 0;
    // 开始计算
```

```
    end else if (~start && state) begin
      if (product[0] == 1'b1) begin          // 如果被乘数最低位值为 1
        product[63:32] = product[63:32] + multiplicand;     // 将被乘数加给乘积
      end
      product = product >> 1;      // 乘积寄存器整体向右移 1 位
      cnt = cnt + 1'b1;
    end
    if (~start && cnt == 0 && state) begin   // 如果已经循环了 32 次，计算完毕
       finish <= 1;
       state <= 0;
    end
  end
endmodule
```

本次实验已给出对应的参考仿真代码，如下所示：

```
`timescale 1ns / 1ps

module tb();
    reg clk;
    reg rst;
    reg[31:0] multiplicand;
    reg[31:0] multiplier;
    reg start;
    wire[63:0] product;
    wire finish;
    initial begin
    clk = 0;
    rst = 1;
    multiplicand = 0;
    multiplier   = 0;
    start        = 0;
    #100
    rst = 0;
    start = 1;
    multiplicand = 32'd2;
    multiplier   = 32'd3;
    #350
    start = 0;
    #350
```

```
    start = 1;
    multiplicand = 32'd10;
    multiplier   = 32'd8;
    #350
    start = 0;
    #350
    start = 1;
    multiplicand = 32'd9;
    multiplier   = 32'd9;
    #350
    start = 0;
    #350
    start = 1;
    multiplicand = 32'd50;
    multiplier   = 32'd6;
    #350
    start = 0;
    #350
    start = 1;
    multiplicand = 32'd6;
    multiplier   = 32'd60;
    #350
    start = 0;
    #4000 $finish();
    end

    always #5 clk = ~clk;

    mul32 mul32_u(
    clk,rst,multiplicand,multiplier,start,product,finish
    );
endmodule
```

补充完 `mul32.v` 的代码后只需点击左侧“Run Simulation”即获得仿真波形（仿真波形图片及其分析在第五节给出）。

4.2 除法器

在 Vivado 中打开已经给出的工程文件 `div32.xpr`，找到 Verilog 文件 `div32.v`，可以看到已经给出了相应接口。接下来便根据上面的实验原理补充下列代码：

```
`timescale 1ns / 1ps

```

```
module div32(
    input    clk,
    input    rst,
    input    start,
    input    [31:0] dividend,
    input    [31:0] divisor,
    output reg finish,
    output   [31:0] quotient,
    output   [31:0] remainder
);

  reg state;           // 检查当前是否位于计算状态
  reg[4:0] cnt;        // 循环计数
  reg[63:0] rem_reg;   // 模拟 64 位余数寄存器，左半部分存放余数（初始为被除数），右半部分存放商

  assign quotient = rem_reg[31:0];
  assign remainder = {1'b0, rem_reg[63:33]};

  // 初始赋值
  initial begin
    rem_reg <= 0;
    finish <= 0;
    state <= 0;
    cnt <= 0;
  end

  always @(posedge clk or posedge rst) begin
    // 当位于复位状态，或重新开始下一次计算时，重新初始化
    if (rst || start && ~state) begin
        rem_reg <= {31'b0, dividend, 1'b0};
        state <= 1;
        finish <= 0;
        cnt <= 0;
    // 开始计算
    end else if (~start && state) begin
      if (rem_reg[63:32] >= divisor) begin              // 如果余数大于等于除数
        rem_reg[63:32] = rem_reg[63:32] - divisor;      // 直接用除数减去余数
        rem_reg = {rem_reg[62:0], 1'b1};                // 右移余数寄存器，末尾补 1
```

```
      end else begin                                    // 若余数小于
除数
        rem_reg = {rem_reg[62:0], 1'b0};                // 右移余数寄
存器，末尾补 0
      end
      cnt = cnt + 1'b1;
    end
    if (~start && cnt == 0 && state) begin              // 如果循环了
32 次，计算结束
       finish <= 1;
       state <= 0;
    end
    if (rst) begin
        finish <= 0;
    end
  end

endmodule
```

本次实验已给出对应的参考仿真代码，如下所示：

```
`timescale 1ns / 1ps

module div32_tb();
      reg clk;
      reg rst;
      reg [31:0] dividend;
      reg [31:0] divisor;
      reg start;

      wire [31:0] quotient;
      wire [31:0] remainder;
      wire finish;
      div32   u_div(
         .clk(clk),
         .rst(rst),
         .dividend(dividend),
         .divisor(divisor),
         .start(start),
         .quotient(quotient),
         .remainder(remainder),
         .finish(finish)
```

```
    );
    always #5 clk = ~clk;

    initial begin
     clk =0;
     rst = 1;
     start = 0;
     #10
     rst = 0;
     start = 1;
         dividend = 32'd8;
         divisor  = 32'd4;
         #335
         dividend = 32'd100;
         divisor  = 32'd10;
         #335
         dividend = 32'd9;
         divisor  = 32'd4;
         #340
         dividend = 32'd100;
         divisor  = 32'd99;
         #350 $stop();

    end
endmodule
```

补充完 div32.v 的代码后只需点击左侧“Run Simulation”即获得仿真波形（仿真波形图片及其分析在第五节给出）。

4.3 浮点加法器

在 Vivado 中打开已经给出的工程文件 float_add.xpr，找到 Verilog 文件 add.v，可以看到已经给出了相应接口。接下来便根据上面的实验原理补充下列代码：

```
`timescale 1ns / 1ps

module float_add(
  input clk,
  input rst,
  input [31:0] A,
  input [31:0] B,
  input [1:0] c,       // 00 +, 01 -, 10 *, 11 /
  input en,      // en = 1, begin
```

```
  output reg [31:0] result,
  output reg fin     // fin = 1 when finish
);

  // 用于表示各阶段的常量
  localparam
    S1 = 3'b001,
    S2 = 3'b010,
    S3 = 3'b011,
    S4 = 3'b100;

  reg state;                                  // 是否进入计算状态
  reg [2:0] stage;                            // 浮点数运算阶段（共 4 步）
  reg A_sign, B_sign, res_sign;               // 符号位（1 位）
  reg [7:0] A_exp, B_exp, res_exp;            // 指数位（8 位）
  reg [24:0] A_frac, B_frac, res_frac;   // 尾数位（23 位）
  // 这里额外加上两位，一位用于存放被忽略的尾数小数点左边的 1；另一位备用，存放可能的进位

  always @(posedge clk or posedge rst) begin
    // 当处于复位状态时，进行总的初始化
    if (rst) begin
      result <= 0;
      fin <= 0;
      state <= 0;
    end
    // 当接收到开始信号时，进一步初始化其他值
    if (en) begin
      A_sign <= A[31];
      A_exp <= A[30:23];
      A_frac <= {2'b01, A[22:0]};
      B_sign <= B[31];
      B_exp <= B[30:23];
      B_frac <= {2'b01, B[22:0]};
      stage <= S1;
      fin <= 0;
      state <= 1;
    end
    // 开始计算
    if (state && ~fin) begin
      case (stage)
        S1: begin                                 // 阶段 1：对齐
```

```
            if (A_exp > B_exp) begin              // 若 A 的指数大
              B_exp <= B_exp + 1;                 // 增加 B 的指数
              B_frac <= B_frac >> 1;              // 且 B 的指数往右移
            end else if (B_exp > A_exp) begin     // 若 B 的指数大
              A_exp <= A_exp + 1;                 // 增加 A 的指数
              A_frac <= A_frac >> 1;              // 且 A 的指数往右移
            end else begin                        // 否则直接进入阶段 2
              stage <= S2;
            end
          end
          S2: begin                               // 阶段 2：加法
            if (A_sign ^ B_sign == 0) begin       // 若两者符号位相同，直
接相加
              res_sign <= A_sign;
              res_frac <= A_frac + B_frac;
            end else if (A_sign == 1) begin       // 若 A 负 B 正
              res_sign <= (A_frac > B_frac);      // 符号位为绝对值较大者的
符号位
               if (A_frac >= B_frac) begin        // 用绝对值较大的尾数 -
绝对值较小的尾数
                 res_frac <= A_frac - B_frac; // 避免下溢问题
              end else begin
                res_frac <= B_frac - A_frac;
              end
            end else begin
              res_sign <= (B_frac > A_frac);      // 若 B 负 A 正，与上面同理
              if (B_frac >= A_frac) begin
                res_frac <= B_frac - A_frac;
              end else begin
                res_frac <= A_frac - B_frac;
              end
            end
            res_exp <= A_exp;                     // 指数位
            stage <= S3;                          // 进入阶段 3
          end
          S3: begin                               // 阶段 3：处理溢出
            if (res_frac[24] == 1) begin          // 仅处理上溢问题，下溢
问题应该不存在
              res_exp <= res_exp + 1;
              res_frac <= res_frac >> 1;
            end
            stage <= S4;                          // 进入阶段 4
```

```
        end
        S4: begin                                    // 阶段 4：汇总，结束
计算
          result <= {res_sign, res_exp, res_frac[22:0]};
          fin <= 1;
          state <= 0;
        end
      endcase
    end
  end
endmodule
```

本次实验已给出对应的参考仿真代码，如下所示：

```
`timescale 1ns / 1ps

module tb;

  // Inputs
  reg clk;
  reg rst;
  reg [31:0] A;
  reg [31:0] B;
  reg [1:0] c;
  reg en;

  // Outputs
  wire [31:0] result;
  wire fin;


  // Instantiate the Unit Under Test (UUT)
  float_add add (
    .clk(clk),
    .rst(rst),
    .A(A),
    .B(B),
    .c(c),
    .en(en),
    .result(result),
    .fin(fin)
  );
```

```
  always #5 clk = ~clk;

  initial begin
    // Initialize Inputs
    clk = 0;
    rst = 1;
    A = 32'hc0000000;
        B = 32'hc0000000;
        c = 2'b00;
        en = 0;
    #10;
    rst = 0;
    A = 32'hc0a00000; //-5.0
    B = 32'hc0900000; //-4.5
    c = 2'b00;        // +
    en = 1;           // c1180000 (-9.5)
    #60;

    // en = 0;
    // #80;

    A = 32'h40a00000; //+5.0
    B = 32'h40900000; //+4.5
    en = 1;           //41180000 (+9.5)
    // #80;
        // en = 0;
        #60;

    $stop();
  end
endmodule
```

补充完 `mul32.v` 的代码后只需点击左侧“Run Simulation”即获得仿真波形（仿真波形图片及其分析在第五节给出）。

五、 实验结果与分析

5.1 乘法器

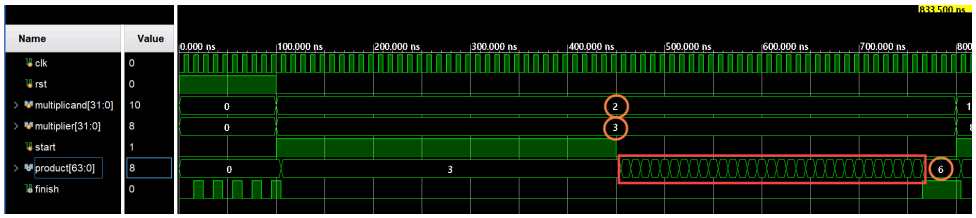

Figure 4: $2 \times 3 = 6$

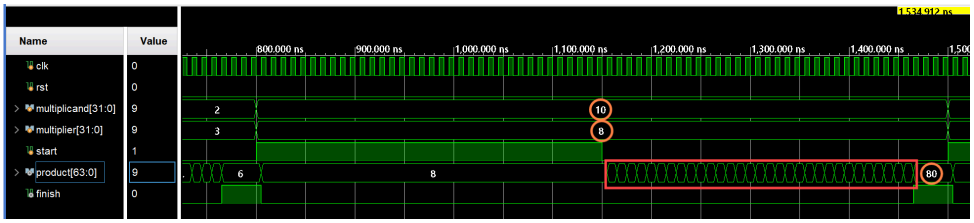

Figure 5: $10 \times 8 = 80$

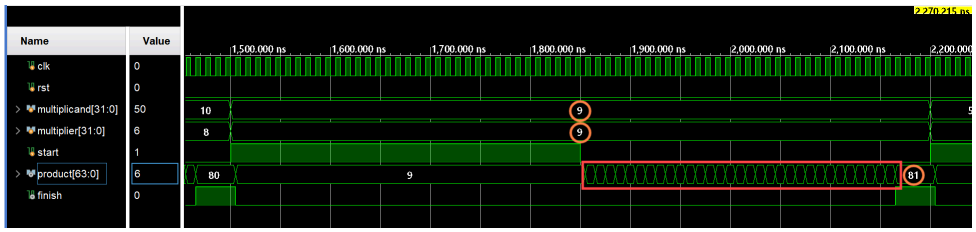

Figure 6: $9 \times 9 = 81$

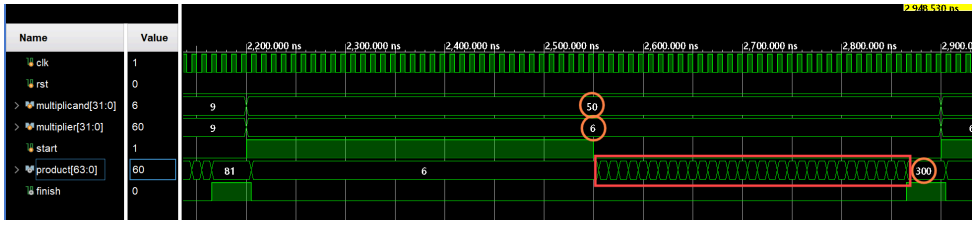

Figure 7: $50 \times 6 = 300$

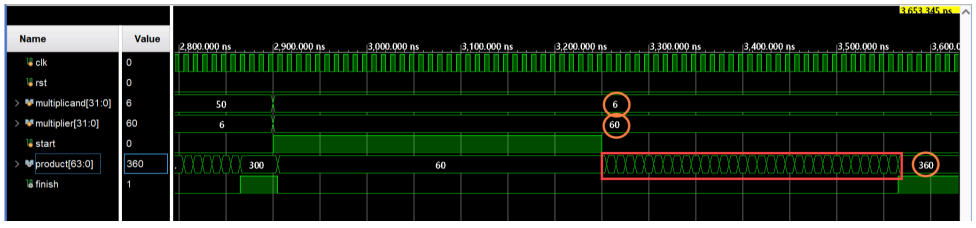

Figure 8: $6 \times 60 = 360$

由于仿真波形过长，因此我将整个仿真波形分为 5 张图片展示，每张图片对应其中一个乘法运算，其中橙色圆圈表示被乘数、乘数与积，红色方框表示乘法运算过程中的移位操作。可以看到乘法器能够正常完成每种乘法运算，说明乘法器的设计是正确的。

5.2 除法器

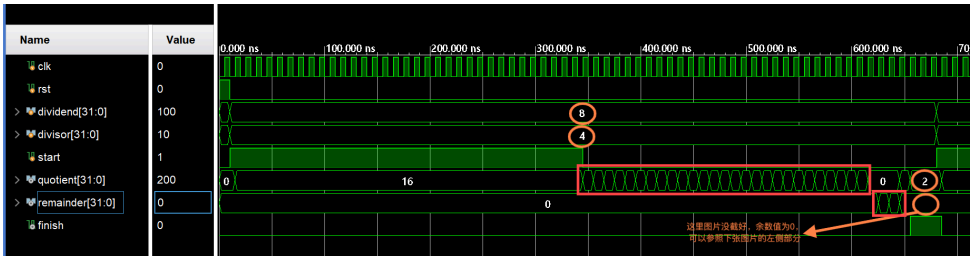

Figure 9: $8 \div 4 = 2 \cdots 0$

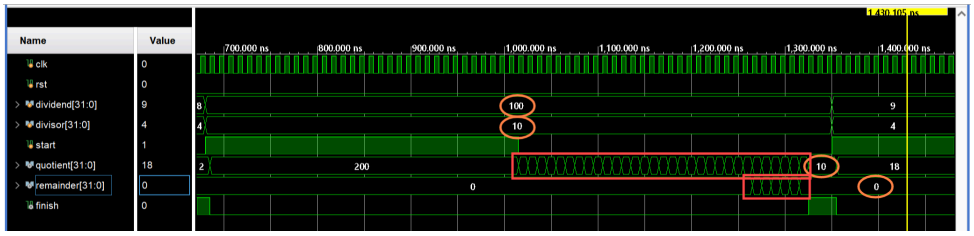

Figure 10: $100 \div 10 = 10 \cdots 0$

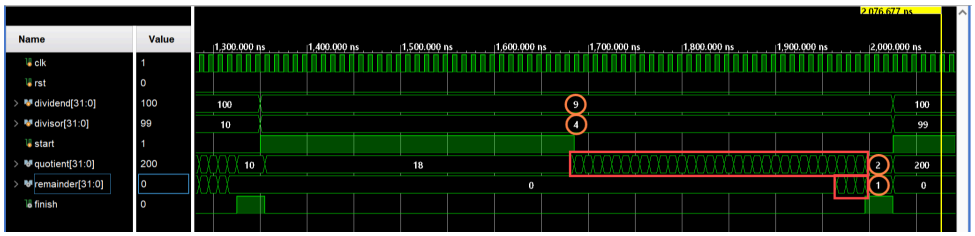

Figure 11: $9 \div 4 = 2 \cdots 1$

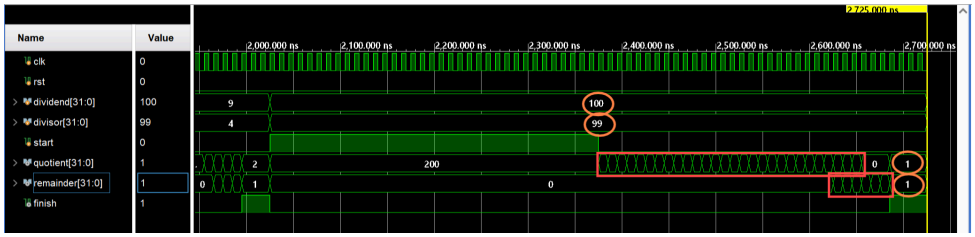

Figure 12: $100 \div 99 = 1 \cdots 1$

由于仿真波形过长，因此我将整个仿真波形分为 4 张图片展示，每张图片对应其中一个除法运算，其中橙色圆圈表示被除数、除数、商和余数，红色方框表示除法运算过程中的移位操作（商和余数）。可以看到除法器能够顺利完成每种除法运算，说明除法器的设计是正确的。

5.3 浮点加法器

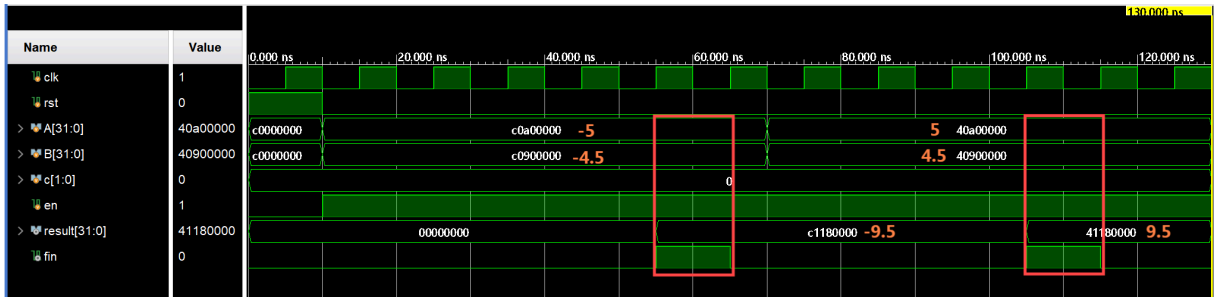

Figure 13: $-5 + (-4.5) = -9.5$

$5 + 4.5 = 9.5$

该仿真波形涉及到两个浮点数加法计算，且经过浮点加法器的运算后，最终都能得到正确的结果。

六、 实验心得

本次实验是对理论课上所讲的乘法器、除法器和浮点加法器逻辑步骤的复现。虽然理论知识不难，但实际写代码的时候还是遇到了不少的阻碍，比如在设计浮点加法器的时候，我最开始为尾数寄存器赋予理论上的 23 位空间，忘记了计算的时候要算上小数点前面的 1，以及可能产生的进位，从而导致计算结果的异常，这暴露了我理论知识掌握得还不够扎实的问题。不过经过这一次的磨砺，我对课上所讲的原理有了更清晰而深入的了解，同时也巩固了 Verilog 代码编写的技能。希望在今后的实验中，我能够提前打好理论基础，吸取过往的教训，避免低级错误，从而更加顺利地完成实验任务。