浙江大学

# 本科实验报告

| | |
|---|---|
| 课程名称: | 计算机组成 |
| 实验名称: | Lab3：复杂操作实现（乘法器、除法器与浮点加法器） |
| 姓　　名: | 李宇晗 |
| 学　　号: | 3240106155 |
| 专　　业: | 计算机科学与技术 |
| 实验地点: | 东教 509 |
| 指导教师: | 林芃 |
| 报告日期: | 2025/11/04 |

# 实验一：32 位无符号乘法器

## 一、 实验目的和要求

1. 复习二进制乘法的基本法则，理解乘法器 V3（优化版本）的硬件实现原理。
2. 掌握使用 Verilog HDL 设计时序逻辑电路的方法。
3. 设计并实现一个 32 位无符号乘法器，使其能够正确计算两个 32 位无符号数的乘积。
4. 通过仿真验证乘法器功能的正确性。

## 二、 实验内容和原理

**2.1) 32** 位乘法器设计（实验任务）

本次实验的任务是设计并实现一个 32 位的无符号乘法器。输入为两个 32 位的无符号数 `multiplicand` (被乘数) 和 `multiplier` (乘数)，一个时钟信号 `clk`，一个复位信号 `rst`，以及一个开始信号 `start`。输出为一个 64 位的结果 `product` (乘积) 和一个计算完成信号 `finish`。

**2.2)** 乘法器原理（实验原理）

本次实验采用的是优化后的乘法器 V3 设计。相比于模拟竖式乘法的 V1 版本，V3 版本极大地节省了硬件资源。其核心思想是将 2N 位（64 位）的乘积寄存器一分为二：高 N 位（32 位）初始为 0，用于累加部分积；低 N 位（32 位）初始加载乘数。

算法流程如下：在 32 次迭代中，每次都检查乘积寄存器的最低位（即当前乘数的最低位）。如果该位为 1，则将高 32 位与被乘数相加，并将结果存回高 32 位；如果为 0，则高 32 位保持不变。随后，将整个 64 位的乘积寄存器逻辑右移 1 位。这样，乘数被逐渐移出，而乘积的高位和低位在寄存器中自动“就位”。

在我们的 `mul32.v` 实现中，为了精确处理加法（33 位+32 位）可能产生的进位，`product_reg` 寄存器被扩展到了 65 位 (`[64:0]`)。加法操作变为 `product_reg[64:32] + multiplicand`，结果存放在 33 位中，然后整体右移，确保了进位不会丢失。

## 三、 实验实现方法、步骤与调试

本次实验通过 Verilog 实现了一个时序逻辑控制的乘法器。核心逻辑在一个 `always @(posedge clk or posedge rst)` 块中实现，并使用一个 2 位的 `flag` 寄存器作为简单的状态机来管理控制流。

`flag` 定义了三个状态：`2'b00` 为空闲状态，等待 `start` 信号；`2'b01` 为计算状态，在此状态下执行 32 次迭代；`2'b10` 为完成状态，此时拉高 `finish` 信号。

当 `start` 信号变为高电平时，模块从空闲态进入计算态，将 `multiplier` 加载到 `product_reg` 的低 32 位，高 33 位和 `cnt` 计数器清零。

在计算状态 (`flag[0] == 1`) 时，每个时钟周期执行一次乘法迭代：首先检查 `product_reg[0]`，如果为 1，则执行加法和右移；如果为 0，则仅执行右移。`cnt` 计数器在每次迭代后加 1，当 `cnt` 计满 31（即已执行 32 次）时，进入完成状态 (`flag <= 2'b10`)。

关键代码 (**mul32.v**)：

```
always@(posedge clk or posedge rst)begin
    if(rst == 1) begin
        // 复位：清空计数器、乘积寄存器和状态标志
        cnt <= 6'b0;
        product_reg <= 65'b0;
        flag <= 0;
    end

    else begin
        if(start == 1) begin
            // 开始信号：加载乘数到低32位，乘积高位清零
            product_reg [31:0] <= multiplier;
            // 修正：应为 product_reg [64:32]
            product_reg [64:32] <= 33'b0; // 高33位清零（包含一位额外进位）
            cnt <= 0;            // 计数器清零
            flag[0] <= 1;        // 进入计算状态
            flag[1] <= 0;        // 拉低 finish 完成信号
        end
        else begin
            if(flag[0] == 1) begin // 状态1：计算中

                // 检查 product_reg[0]（当前乘数的最低位）
                if(product_reg[0] == 1) begin
                    // 最低位为1：执行 加法并右移
                    // { (高33位 + 被乘数)，低32位 } >> 1
                    product_reg <= { (product_reg[64:32] + multiplicand),
product_reg[31:0] } >> 1;
                end
                else begin
                    // 最低位为0：仅右移
                    product_reg <= product_reg >> 1;
                end

                // 检查是否已执行32次迭代（0到31）
                if(cnt == 31) begin
                    flag <= 2'b10; // 进入完成状态
                    cnt <= 0;
                end
                else begin
                    cnt <= cnt + 1; // 迭代次数加1
                end
            end
        end
    end
end

assign finish = flag[1]; // 完成信号（状态2）
assign product = product_reg[63:0]; // 输出64位乘积
```

## 四、 实验结果与分析

对 `mul32.v` 模块进行仿真，结果如下：

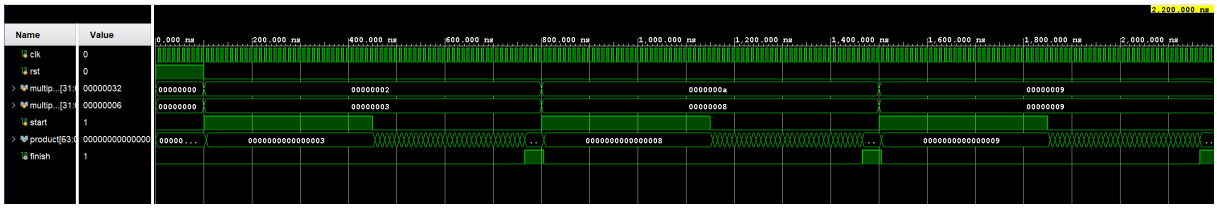

Figure 1: mul32 仿真波形图

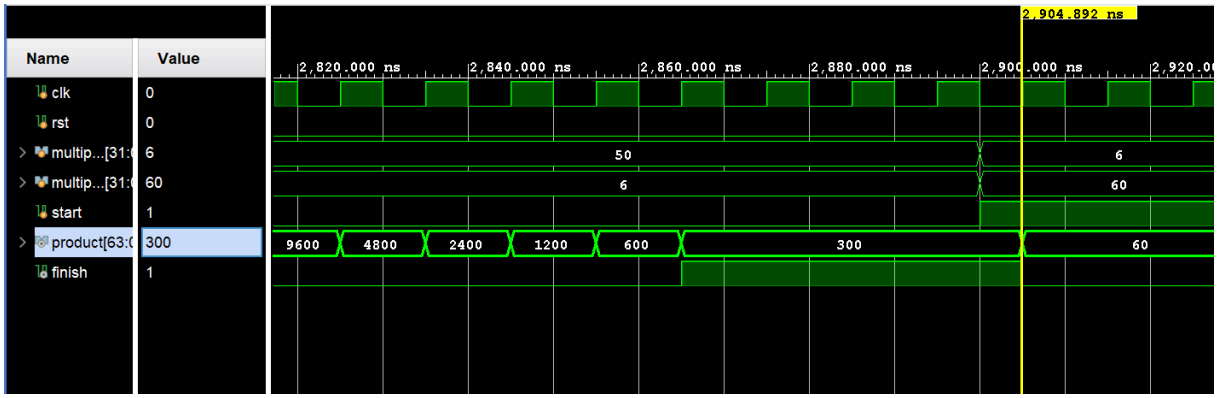


Figure 2: 50*6 波形图局部

从仿真波形图分析可知：当 `rst` 复位后，`start` 信号拉高一个时钟周期即可启动一次乘法运算。`flag` 随之进入计算状态 (`2'b01`)，`finish` 信号保持为低。在 `start` 触发 32 个时钟周期后，`cnt` 计满，`flag` 进入完成状态 (`2'b10`)，`finish` 信号 (`flag[1]`) 随之拉高。`product` 寄存器在 `finish` 拉高时显示正确的 64 位乘法结果。例如，当输入 `multiplicand = 50` 和 `multiplier = 6` 时，`product` 输出 `300` (`64'h12C`)。所有测试用例均通过，仿真结果与预期一致。

## 五、 实验讨论、心得

本次实验设计的乘法器是一个多周期时序电路。在实现过程中，有几个关键点值得注意：

1. 状态控制：使用了 `flag` 寄存器进行简单的状态控制，区分“空闲”、“计算中”和“完成”三个状态，确保了 `start` 和 `finish` 信号的正确时序。
2. 无符号数：本设计实现的是无符号乘法。如果要支持有符号乘法（如 Booth 算法或先转绝对值再处理符号位），逻辑将更为复杂。

# 实验二：32 位无符号除法器

## 一、 实验目的和要求

1. 复习二进制除法的基本法则，理解恢复余数法的硬件实现原理。
2. 掌握使用状态机（FSM）设计复杂时序逻辑电路。
3. 设计并实现一个 32 位无符号除法器，使其能够计算两个 32 位无符号数的商和余数。
4. 通过仿真验证除法器功能的正确性。

## 二、 实验内容和原理

**2.1) 32** 位除法器设计（实验任务）

设计并实现一个 32 位的无符号除法器。输入为 32 位被除数 `dividend` 和 32 位除数 `divisor`，以及 `clk`, `rst`, `start` 信号。输出为 32 位的商 `quotient`，32 位的余数 `remainder` 和一个计算完成信号 `finish`。

**2.2)** 除法器原理（实验原理）

本次实验采用的是恢复余数法（Restoring Division）的优化实现（V3 版本）。该设计巧妙地使用一个 64 位寄存器 `res` 来同时存放余数和商。其高 32 位 `res[63:32]` 用于存放余数 (Remainder)，低 32 位 `res[31:0]` 用于存放商 (Quotient)。

初始化时，将 `res` 寄存器的高 32 位（余数部分）清零，低 32 位加载被除数 `dividend`。

随后进行 32 次迭代。每次迭代包含三个步骤:

1. 左移: 将 `res` 寄存器（包含余数和商）整体逻辑左移 1 位。这相当于将余数左移 1 位，同时将被除数的最高位移入余数的最低位。
2. 尝试减法: 从左移后的余数 `res[63:32]` 中减去除数 `divisor`。
3. 检查结果: 如果减法结果为负（即发生借位），说明除数大于当前余数，减法失败。此时，需要恢复余数（即不更新高 32 位），并将商的最低位（`res[0]`）置为 0。如果减法结果为正或零，说明减法成功。此时，将减法结果更新回 `res[63:32]`，并将商的最低位置为 1。

在 `div32.v` 的实现中，通过 `sub_result` 和 `borrow` 信号来合并执行 b 和 c:

如果 `borrow == 1` (减法失败)，则 `R_next` (新余数) 等于 `R_from_shift` (原余数)，`Q_bit_next` (新商位) 为 0；否则，`R_next` 等于 `sub_result` (减法结果)，`Q_bit_next` 为 1。

循环 32 次后，`res[63:32]` 中存放的是最终的余数，`res[31:0]` 中存放的是最终的商。

## 三、 实验实现方法、步骤与调试

我们使用一个三状态的状态机 (`state`) 来控制除法流程。

- `state 2'b00` (空闲): `finish` 拉低。等待 `start` 信号为高。当 `start` 触发时，加载 `dividend` 到 `res` 低 32 位，`res` 高 32 位清零，`cnt` 清零，进入 `2'b01` 状态。

- `state 2'b01` (计算)：执行上述恢复余数法的单步迭代。`cnt` 计数器加 1。当 `cnt == 31` (完成 32 次迭代) 时，进入 `2'b10` 状态，并拉高 `finish`。
- `state 2'b10` (完成)：`finish` 保持为高。等待 `start` 信号变低，然后返回 `2'b00` 空闲状态。

关键代码 (**div32.v**)：

```
always @(posedge clk or posedge rst) begin
    if (rst == 1) begin
        // 复位逻辑：状态归0，清空所有寄存器
        state  <= 2'b00;
        cnt    <= 6'b0;
        res    <= 64'b0;
        finish <= 1'b0;
    end
    else begin
        case (state)
            2'b00: begin // 状态0：空闲
                finish <= 1'b0;
                if (start == 1) begin
                    // 开始：加载被除数到低32位，高32位(余数)清零
                    res    <= {32'b0, dividend};
                    cnt    <= 6'b0;
                    state  <= 2'b01; // 进入计算状态
                end
            end

            2'b01: begin // 状态1：计算中
                // 步骤a：将64位res寄存器（余数+商）整体左移一位
                res_shifted = res << 1;
                R_from_shift = res_shifted[63:32]; // 获取左移后的余数部分

                // 步骤b：尝试减法（余数 - 除数）
                sub_result = {1'b0, R_from_shift} - {1'b0, divisor};
                borrow = sub_result[32]; // 检查借位（borrow=1 表示减法失败）

                // 步骤c：检查结果
                if (borrow == 1'b1) begin
                    // 减法失败：恢复余数，商位置0
                    R_next = R_from_shift;
                    Q_bit_next = 1'b0;
                end else begin
                    // 减法成功：更新余数为减法结果，商位置1
                    R_next = sub_result[31:0];
                    Q_bit_next = 1'b1;
                end

                // 将新的余数(R_next)和新的商位(Q_bit_next)组合回res寄存器
                res_next_calc = {R_next, res_shifted[31:1], Q_bit_next};
                res <= res_next_calc;

                // 检查是否完成32次迭代
                if (cnt == 31) begin
```

```
                    state  <= 2'b10; // 进入完成状态
                    finish <= 1'b1; // 拉高 finish 信号
                end else begin
                    cnt <= cnt + 1; // 迭代次数加1
                end
            end

            2'b10: begin // 状态2: 完成
                finish <= 1'b1; // 保持 finish 为高
                if (start == 0) begin // 等待 start 信号变低后，返回空闲状态
                    state  <= 2'b00;
                    finish <= 1'b0;
                end
            end
        endcase
    end
end

// 输出赋值: 商在低32位，余数在高32位
assign quotient = res[31:0];
assign remainder = res[63:32];
```

## 四、 实验结果与分析

对 `div32.v` 模块进行仿真，结果如下：

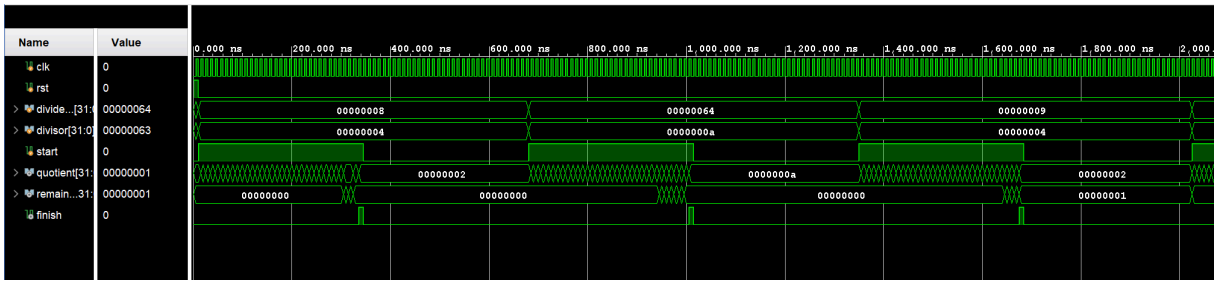


Figure 3: div32 仿真波形图

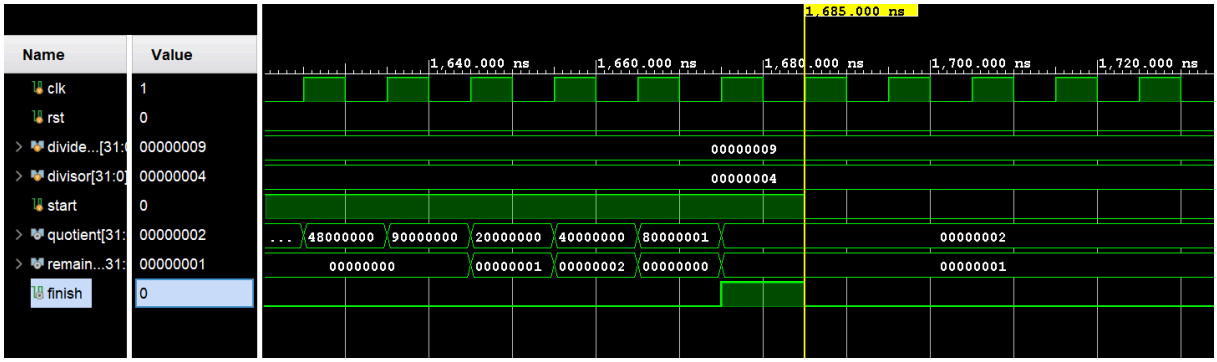


Figure 4: 9/4 波形图局部

从仿真波形图分析可知：当 `rst` 复位后，`start` 信号拉高一个时钟周期即可启动一次除法运算。状态机进入 `2'b01` (计算)，`finish` 保持为低。在 `start` 触发 32 个时钟周期后，`cnt` 计满，状态机进入 `2'b10` (完成)，`finish` 信号随之拉高。`quotient` 和 `remainder` 寄存

器在 finish 拉高时显示正确的商和余数。例如，当输入 dividend = 9 和 divisor = 4 时，quotient 输出 2，remainder 输出 1。100 / 99 的测试用例也正确得到了商 1 和余数 1。所有测试用例均通过。

## 五、 实验讨论、心得

除法器的设计比乘法器更依赖于状态机的精确控制。

1. 恢复余数法: 本次实现的恢复余数法逻辑较为清晰: 先减，再根据结果判断是否恢复。
2. **64** 位寄存器的妙用: res 寄存器同时存储余数和商，并在每次迭代中整体左移，这个设计非常精妙。它避免了单独的商寄存器和左移逻辑，简化了数据通路。
3. 有符号除法和除 **0**: 本设计只针对无符号除法。若要支持有符号除法，需要预先处理操作数的符号，并在最后恢复结果的符号。此外，也没有处理除数为 0 的异常情况，在实际应用中需要添加除 0 检测逻辑。

# 实验三：32 位浮点数加法器

## 一、 实验目的和要求

1. 了解 IEEE-754 单精度浮点数的表示方法。
2. 掌握浮点数加法运算的完整步骤：对阶、尾数相加、规格化、舍入和溢出处理。
3. 设计并实现一个 32 位浮点数加法器。
4. 通过仿真验证浮点加法器功能的正确性。

## 二、 实验内容和原理

**2.1) 32** 位浮点加法器设计（实验任务）

设计一个 32 位 IEEE-754 标准浮点数加法器。输入为两个 32 位数 `A` 和 `B`，以及 `clk`, `rst`, `en` (使能) 信号。输出为 32 位的结果 `result` 和一个完成脉冲信号 `fin`。

**2.2)** 浮点加法器原理（实验原理）

浮点数加法是一个复杂的多步骤过程，本实验 `float_add32.v` 将其实现为一个大型组合逻辑（计算）和一个时序逻辑（锁存结果和 `fin` 信号）。

核心步骤在 `always @(*)` 块中实现：

1. 准备工作： 从输入 A 和 B 中提取符号位 `signA/signB`、阶码 `expA_raw/expB_raw` 和尾数 `fracA/fracB`。
2. **0** 操作数处理及剪枝： 处理特殊情况以简化后续逻辑或返回正确结果：
   - NaN (阶码=255)：`result = QNaN`。
   - A = 0：`result = B`。
   - B = 0：`result = A`。
   - A = -B (符号相反，阶码和尾数相同)：`result = 0`。
3. 处理尾数（转换为补码）：
   - 为尾数添加隐藏位（`1.M`）和保护位（Guard bit, `_g`）。对于规格化数，尾数变为 `1.fracA`；对于非规格化数，尾数变为 `0.fracA`。
   - 代码中 `pA = {2'b0, 1'b1, fracA, 1'b0};` (27 位，`00.1_fracA_0`)。`expA_align` 存阶码。
   - 如果 `signA` 为 1 (负数)，将 `pA` 转换为补码 (`pA = ~pA + 1;`)。`pB` 同理。这使得后续加法统一为补码加法。
4. 对阶：
   - 比较 `expA_align` 和 `expB_align`。
   - 计算阶差 `expBsubA`。
   - 将阶码较小的数的尾数（`pA` 或 `pB`）进行算术右移，移动位数为阶差，使其阶码与较大的阶码 `resExp` 对齐。
5. 尾数相加：
   - 执行补码加法 `tempRes = pA + pB;`。

6. 规格化：
   - 右规 (**Overflow**)：如果 `tempRes` 发生溢出（如 `01.xxx` 或 `10.xxx`），则将 `tempRes` 算术右移 1 位，`resExp` 加 1。
   - 左规 (**Underflow**)：如果 `tempRes` 结果太小（如 `00.0xxx` 或 `11.1xxx`），则需要左移。代码中使用一个 `if-else` 链来查找最高位的 1（或 0，对于负数补码），计算出需要左移的位数 `shift_amount`。
   - `tempRes` 左移 `effective_shift` 位，`resExp` 减去 `effective_shift` 位。同时处理非规格化转变（`resExp` 减到 0）。
7. 溢出和舍入：
   - 溢出判断：检查 `resExp` 是否上溢 (`>= 255`) 或下溢 (`resExp=0` 且 `flag=1`)。
   - 舍入：本设计实现了一种“向正无穷舍入”的变体。
     - `result[22:0] = ... tempRes[23:1];` (截断，取 23 位尾数)。
     - `if (result_comb[31] == 0 && tempRes[0] == 1) ... result_comb = result_comb + 1;` (如果结果为正，且被舍弃的保护位 `tempRes[0]` 为 1，则进位)。

时序逻辑： `always @(posedge clk or posedge rst)` 块用于锁存结果和生成 `fin` 脉冲。

- `en` 为高电平时，`result_reg <= result_comb;`。
- 当 `en` 信号启动（从 0 到 1）或 `en` 保持为 1 且输入 `A` 或 `B` 变化时，`fin` 拉高一个时钟周期。

## 三、 实验实现方法、步骤与调试

关键代码 (**`float_add32.v`**)：

对阶逻辑：

```
// (3) 对阶：比较阶码，将小阶的尾数右移
expBsubA = $signed(expB_align) - $signed(expA_align); // 计算阶差
if ($signed(expBsubA) >= 0) begin // B的阶码 >= A的阶码
    pA = $signed(pA) >>> expBsubA; // A的尾数算术右移
    resExp = expB_align;           // 结果的阶码以B为准
end
else begin // A的阶码 > B的阶码
    pB = $signed(pB) >>> (-expBsubA); // B的尾数算术右移
    resExp = expA_align;           // 结果的阶码以A为准
end
```

规格化 (右规)：

```
// (5) 规格化：处理尾数相加后的结果
// 右规 (结果溢出，如 01.xxx 或 10.xxx)
if (tempRes[26:25] == 2'b10 || tempRes[26:25] == 2'b01) begin
    tempRes = $signed(tempRes) >>> 1; // 尾数算术右移1位
    resExp = resExp + 1;              // 阶码加1
end
```

规格化 (左规)：

```
// (5) 规格化：处理尾数相加后的结果
// 左规 (正数，结果过小，如 00.0xxx)
else if (tempRes[26:24] == 3'b000 && tempRes[23:0] != 0) begin
    // ... (if-else 链计算 shift_amount，即左移位数) ...
    effective_shift = shift_amount;
    resExp = resExp - shift_amount; // 阶码减小
    tempRes = tempRes << effective_shift; // 尾数左移
end
// (负数左规，逻辑类似)
```

舍入：

```
// (6) 溢出和舍入
...
else begin
    result_comb[31] = tempRes[26];      // 符号位
    result_comb[30:23] = resExp;    // 阶码
    // 尾数：如果是负数(补码)，转回原码 (取反+1)
    result_comb[22:0] = (result_comb[31]) ? ~tempRes[23:1] + 1 : tempRes[23:1];

    // 舍入：向正无穷舍入 (仅当结果为正且保护位tempRes[0]为1)
    if (result_comb[31] == 0 && tempRes[0] == 1) begin
        result_comb = result_comb + 1;
    end
end
```

## 四、 实验结果与分析

对 `float_add32.v` 模块进行仿真，结果如下：

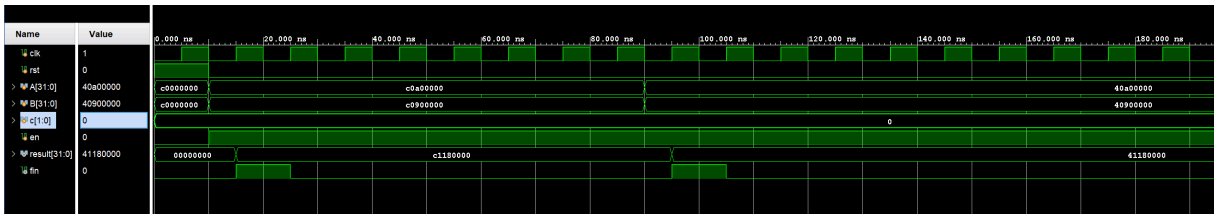


Figure 5: float_add 波形图

从仿真波形图分析可知：当 `en` 拉高，且 `A` 和 `B` 输入稳定后，在下一个时钟周期的上升沿，组合逻辑的计算结果 `result_comb` 被锁存到 `result_reg` (即 `result` 输出)。同时，`fin` 信号拉高一个时钟周期，表示一次有效的计算已完成。仿真结果与预期一致。

## 五、 实验讨论、心得

浮点数加法器是本次实验中最复杂的模块，它的设计几乎涵盖了计算机组成原理中关于浮点运算的所有难点。

1. 组合逻辑 **vs** 时序逻辑：`float_add32.v` 将核心算法实现为纯组合逻辑，这意味着只要输入变化，结果立刻（经过一定的门延迟）可用。时序逻辑部分仅用于同步输出和产生 `fin` 信号。这种设计在单周期处理器中很常见，但也可能导致组合逻辑路径过长，难以满足高时钟频率的要求。

2. 补码尾数：将正负尾数统一转换为补码再进行加法，极大地简化了“尾数相加”步骤的逻辑，无需根据符号位判断是做加法还是减法。
3. 规格化：规格化是浮点运算的核心。左规逻辑中，使用 `if-else` 链来查找最高有效位（Leading One Detector）虽然可行，但在硬件实现上效率低下，会综合成很长的逻辑链。更优的实现是使用优先级编码器（Priority Encoder）。
4. 舍入：代码实现的舍入逻辑比较简单，只对正数且保护位为 1 的情况进行了进位。标准的 IEEE-754 舍入（如“就近舍入”）会更复杂，需要考虑粘滞位（Sticky bit）。