中国科学院大学
《计算机组成原理（研讨课)》实验报告

姓名 王谊康 江宇涵 孔令源 箱子号 20 专业 计算机科学与技术

实验项目编号exp10 11实验名称 算术逻辑运算指令和乘除法运算指令添加与转移指令和访存指令添加

一、 总体设计思路

本次实验的目标是在已有的五级流水线 CPU 基础上，添加若干算术逻辑运算指令和乘除法运算指令。根据指令的特性，我们将设计思路分为以下两个部分：

算术逻辑运算指令的添加 本部分需要添加的指令包括 slti, sltui, andi, ori, xori, sll.w, srl.w, sra.w 以及 pcaddu12i。通过分析指令功能，我们发现这些指令的数据通路和控制逻辑可以大量复用已有指令的设计。

- 对于 slti, sltui, andi, ori, xori 等立即数指令，其与 slt, sltu, and, or, xor 的核心区别仅在于第二个源操作数来自立即数而非寄存器。因此，我们可以在译码阶段（ID）正确地解析出立即数，并在执行阶段（EXE）将 ALU 的输入选择为立即数，而无需修改 ALU 本身的运算逻辑。
- 对于 sll.w, srl.w, sra.w 等移位指令，其与 slli.w, srli.w, srai.w 的数据通路在执行、访存和写回阶段完全一致。我们只需在译码阶段正确识别这些指令，并生成与立即数移位指令相同的控制信号即可。
- 对于 pcaddu12i 指令，其功能是将 PC 的值与一个 12 位立即数相加。这条指令的数据通路可以复用 add.w 的加法器，但其操作数来源比较特殊：一个来自 PC，另一个来自指令中的立即数。这部分可以通过复用分支指令中获取 PC 值和 lu12i.w 中处理立即数的通路来实现。

乘除法运算类指令的添加 本部分需要添加 mul.w, mulh.w, mulh.wu, div.w, mod.w, div.wu, mod.wu 指令。由于 CPU 中没有现成的乘除法单元，因此核心任务是设计并集成独立的乘法器和除法器。

- **硬件设计**：需要在执行阶段（EXE）添加一个乘法器模块和一个除法器模块，并扩展 ALU，使其能够根据译码阶段传来的 `alu_op` 控制信号，将运算任务分发给乘法器或除法器，并将计算结果返回。
- **流水线控制**：考虑到除法运算通常需要多个时钟周期才能完成，为了避免数据冲突和结构冲突，必须对流水线进行处理。我们将在除法器模块中设计一个 `complete` 信号。当除法指令进入执行阶段时，若运算尚未完成，该信号将阻塞 EXE 阶段以及之前的所有阶段，直到运算完成，流水线才恢复正常执行。

下图为我们设计的 cpu 结构框图:

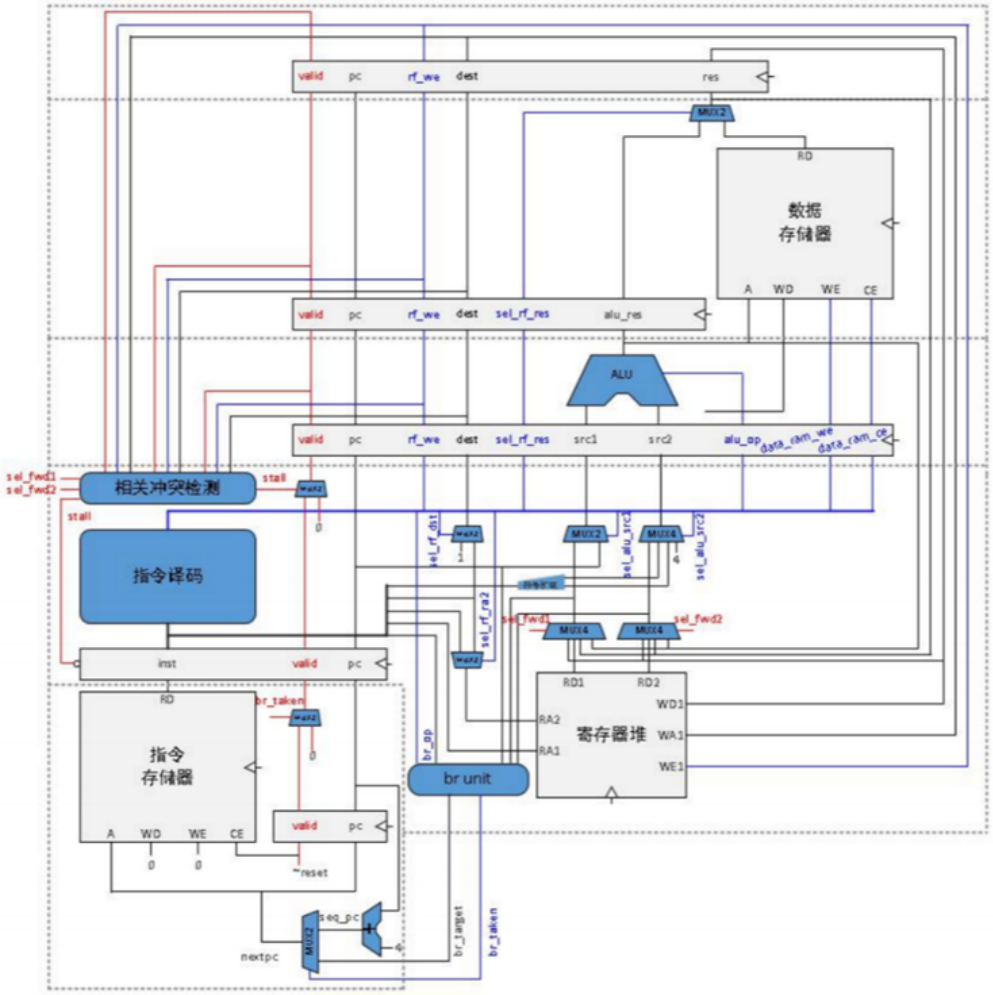
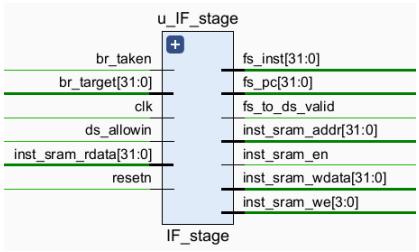
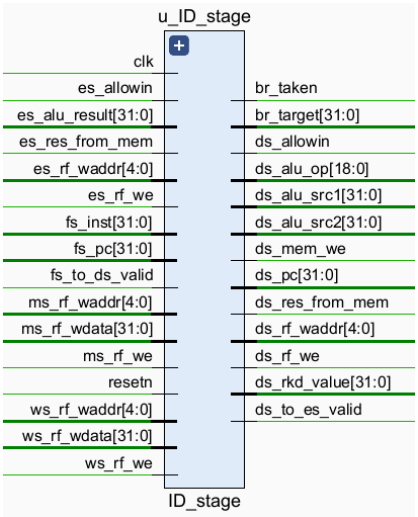


图 1: 结构框图

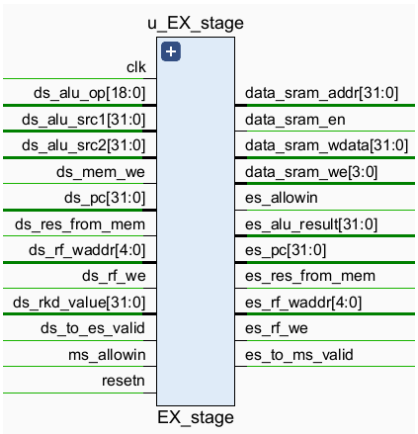
具体部件和模块:



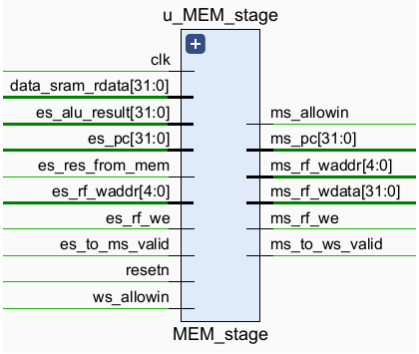
(a) if 模块



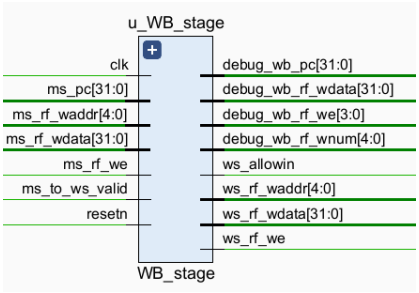
(b) id 模块



(c) ex 模块

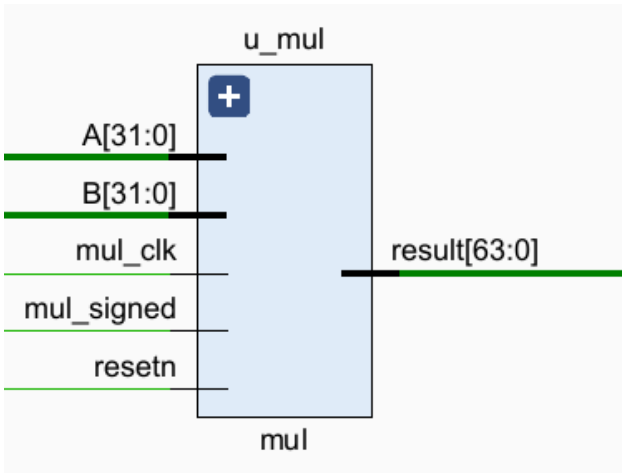


(d) mem 模块

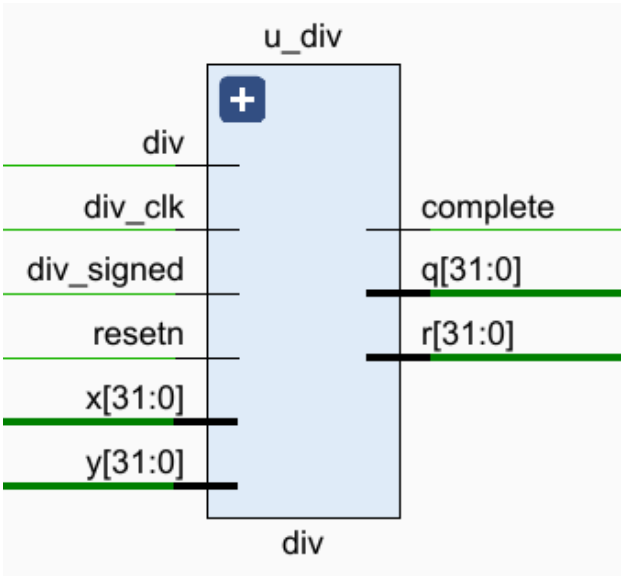


(e) wb 模块

图 2: 流水线各模块



(a) mul



(b) div

图 3: 乘除法器

二、 具体实现与修改

(一) 指令译码阶段 (ID Stage)

ID 阶段是所有改动的起点。首先，我们为所有新指令定义了唯一的 `inst_XXX` 信号，并根据指令集手册完成了译码逻辑的添加。

Listing 1: 为 R-Type 型乘除法与移位指令添加译码逻辑

```
// inst_sll_w, inst_srl_w, inst_sra_w
// inst_mul_w, inst_mulh_w, inst_mulh_wu
// inst_div_w, inst_div_wu, inst_mod_w, inst_mod_wu
case(inst_field.op)
    6b000100: begin
        case(inst_field.func)
            7b0000001: inst_mul_w = 1b1;
            7b0000010: inst_mulh_w = 1b1;
            7b0000011: inst_mulh_wu = 1b1;
            7b0001001: inst_div_w = 1b1;
            7b0001011: inst_div_wu = 1b1;
            7b0001101: inst_mod_w = 1b1;
            7b0001111: inst_mod_wu = 1b1;
        endcase
    end
    6b000101: begin
        case(inst_field.func)
            7b0100001: inst_sll_w = 1b1;
            7b0101001: inst_srl_w = 1b1;
            7b0101101: inst_sra_w = 1b1;
        endcase
    end
endcase
```

Listing 2: 为 I-Type 型算术逻辑指令添加译码逻辑

```
// inst_slti, inst_sltui, inst_andi, inst_ori, inst_xori
case(inst_field.op)
    6b001010: inst_slti = 1b1;
    6b001011: inst_sltui = 1b1;
    6b001101: inst_andi = 1b1;
    6b001110: inst_ori = 1b1;
    6b001111: inst_xori = 1b1;
endcase
```

Listing 3: 为 pcaddu12i 指令添加译码逻辑

```
// inst_pcaddu12i
case(inst_field.op)
    6b011110: inst_pcaddu12i = 1b1;
endcase
```

译码后，需要根据指令类型生成正确的控制信号，引导数据在后续流水线阶段正确流动。

ALU 操作类型 (alu_op) 修改 为了控制新增的乘除法运算，我们扩展了 alu_op 信号，为每种新运算分配了唯一的编码。

```
assign ds_alu_op[ 0] = inst_add_w | inst_addi_w | inst_ld_w | inst_st_w
                     | inst_jirl | inst_bl | inst_pcaddu12i;
assign ds_alu_op[ 1] = inst_sub_w;
assign ds_alu_op[ 2] = inst_slt | inst_slti;
assign ds_alu_op[ 3] = inst_sltu | inst_sltui;
assign ds_alu_op[ 4] = inst_and | inst_andi;
assign ds_alu_op[ 5] = inst_nor;
assign ds_alu_op[ 6] = inst_or | inst_ori;
assign ds_alu_op[ 7] = inst_xor | inst_xori;
assign ds_alu_op[ 8] = inst_slli_w | inst_sll_w;
assign ds_alu_op[ 9] = inst_srli_w | inst_srl_w;
assign ds_alu_op[10] = inst_srai_w | inst_sra_w;
assign ds_alu_op[11] = inst_lu12i_w;
assign ds_alu_op[12] = inst_mul_w ;
assign ds_alu_op[13] = inst_mulh_w;
assign ds_alu_op[14] = inst_mulh_wu;
assign ds_alu_op[15] = inst_div_w;
assign ds_alu_op[16] = inst_div_wu;
assign ds_alu_op[17] = inst_mod_w;
assign ds_alu_op[18] = inst_mod_wu;
```

图 4: 为新增运算扩展 alu_op 编码

源操作数选择逻辑修改 根据新指令的特点，我们修改了源操作数的选择逻辑，例如 pcaddu12i 的第一个操作数应为 PC 值，而各类立即数指令的第二个操作数应为立即数。

```
assign ds_src1_is_pc    = inst_jirl | inst_bl | inst_pcaddul2i;

assign ds_src2_is_imm   = inst_slli_w |
                          inst_srli_w |
                          inst_srai_w |
                          inst_addi_w |
                          inst_ld_w   |
                          inst_st_w   |
                          inst_lu12i_w|
                          inst_jirl   |
                          inst_bl     |
                          inst_pcaddul2i|
                          inst_andi   |
                          inst_ori    |
                          inst_xori   |
                          inst_slti   |
                          inst_sltui;
```

图 5: 修改源操作数选择逻辑以支持新指令

(二) 执行阶段 (EXE Stage)

(二).1 ALU 模块扩展

为了支持乘除法，ALU 模块是修改的核心。我们首先设计了独立的乘法器和除法器两个硬件单元，然后在 ALU 内部定义了相应的 `alu_op` 编码，并对这两个单元进行例化集成。

乘法器设计 只实现了单周期乘法器，完全由组合逻辑构成的“Radix-4 Booth 编码 + 华莱士树”架构。整个流程可分为三个主要阶段：

1. 部分积生成

此阶段的目标是减少后续求和的复杂度。采用了 2 位 Booth 编码（Radix-4 Booth Algorithm），它通过对乘数进行分组编码，可以将 32 位乘法的部分积数量从 32 个显著减少到 17 个。为了统一处理有符号与无符号乘法，我们将 32 位的乘数操作数扩展至 34 位进行运算。首先预先计算出被乘数 A 的 +A, -A, +2A, -2A 四种倍数形式。

Listing 4: Booth 算法所需操作数的预计算

```
1 // Pre-computation of multiplicand multiples
2 wire [63:0] A_add;   // +A (sign-extended to 64 bits)
3 wire [63:0] A_sub;   // -A (2's complement of A)
4 wire [63:0] A2_add;  // +2A (A shifted left by 1)
5 wire [63:0] A2_sub;  // -2A (2's complement of 2A)
6
7 assign A_add = {{32{A[31] & mul_signed}}, A};
8 assign A_sub = ~A_add + 1'b1;
9 assign A2_add = A_add << 1;
```

```
assign A2_sub = ~A2_add + 1'b1;
```

随后，通过一个 generate 循环，并行地生成所有 17 个部分积。每个部分积的值是根据 Booth 编码的选择信号，从上述预计算的操作数中选出的。

Listing 5: 17 个部分积的并行生成

```
wire [63:0] P [16:0]; // 17 unaligned partial products

genvar i;
generate
    for (i = 0; i < 17; i = i + 1) begin : gen_partial_products
        // Select one of {0, +A, -A, +2A, -2A} based on Booth encoding
        assign P[i] = (sel_x_val[i] ? A_add : 64'b0) |
                      (sel_neg_x_val[i] ? A_sub : 64'b0) |
                      (sel_2x_val[i]  ? A2_add : 64'b0) |
                      (sel_neg_x_val[i] ? A2_sub : 64'b0);
    end
endgenerate
```

2. 部分积压缩

此阶段的核心是利用华莱士树对 17 个部分积进行高效的并行求和。华莱士树的构建单元是进位保存加法器。我们设计了 Adder 模块来简化计算，它能将 3 个输入压缩为 2 个输出（一个和向量 S，一个进位向量 C），且无横向进位传播。

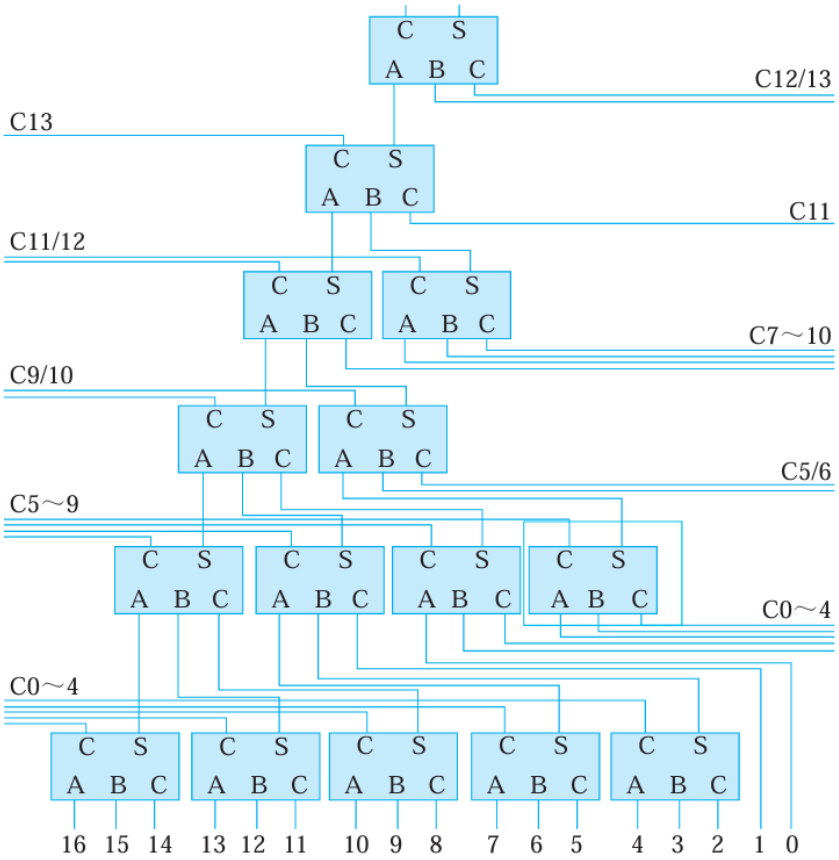


图 6: 华莱士树

Listing 6: 进位保存加法器

```
module Adder (
    input  [63:0] in1,
    input  [63:0] in2,
    input  [63:0] in3,
    output [63:0] C, // Carry vector
    output [63:0] S // Sum vector
);
    assign S = in1 ^ in2 ^ in3;
    assign C = {(in1 & in2 | in1 & in3 | in2 & in3), 1'b0} ;
endmodule
```

我们通过多级 CSA 网络，将 17 个部分积逐级规约。如下图所示，从 17 个输入压缩到 12 个，再到 8 个的规约。整个树状结构持续压缩，直至最终只剩下两个 64 位的数。

Listing 7: 华莱士树的前两级规约示例（17 -> 12 -> 8）

```
// Level 1: 17 inputs -> 12 outputs
wire [63:0] level_1 [11:0];
Adder adder1_1 (.in1(P[15] << 30), .in2(P[14] << 28), .in3(P[13] << 26), .C(level_1[0]),
```

```
    .S(level_1[1]));
Adder adder1_2 (.in1(P[12] << 24), .in2(P[11] << 22), .in3(P[10] << 20), .C(level_1[2]),
    .S(level_1[3]));
// ... other level 1 adders ...
assign level_1[10] = P[0];
assign level_1[11] = P[16] << 32;

// Level 2: 12 inputs -> 8 outputs
wire [63:0] level_2 [7:0];
Adder adder2_1 (.in1(level_1[0]), .in2(level_1[1]), .in3(level_1[2]), .C(level_2[0]),
    .S(level_2[1]));
Adder adder2_2 (.in1(level_1[3]), .in2(level_1[4]), .in3(level_1[5]), .C(level_2[2]),
    .S(level_2[3]));
// ... other level 2 adders ...
```

经过华莱士树的并行压缩后，17 个部分积的求和问题被成功转化为了两个 64 位数（`level_6[0]` 和 `level_6[1]`）相加的问题。最后，我们使用一个高速的进位传播加法器（由 Verilog 的 '+'运算符综合而成）对这两个数进行求和，得出最终的 64 位乘法结果即可。

除法器设计 与采用纯组合逻辑的高速乘法器不同，除法运算本质上是一个迭代过程，因此除法器被设计为一个多周期的时序逻辑电路。本次实验中，我们实现了恢复余数法除法器。该设计由一个计数器控制的状态机驱动，整个除法过程（包括初始化）共需要 33 个时钟周期。

1. 初始化与符号预处理（Cycle 0）

当外部 `div` 信号有效时，除法器在第一个时钟周期（`count == 0`）进行初始化。如 Listing 8 所示，此阶段主要完成三项工作：

- 对输入的被除数 `x` 和除数 `y` 取绝对值，因为核心的迭代算法只处理无符号数。
- 根据 `x` 和 `y` 的原始符号位，预先计算出最终商 `q` 和余数 `r` 的符号。
- 初始化一个 65 位的核心寄存器 `Q_R_Reg`。该寄存器的高 33 位用于存放余数，低 32 位用于存放商，初始值为 `{33'b0, abs_x}`。

Listing 8: 除法器的初始化与符号预处理逻辑

```
// --- Pre-computation for sign handling ---
assign abs_x = div_signed ? (x[31] ? ~x + 1'b1 : x) : x;
assign abs_y = div_signed ? (y[31] ? ~y + 1'b1 : y) : y;
assign sign_q = x[31] ^ y[31];
assign sign_r = x[31];

// --- Initial value for the main register ---
assign initial_Q_R = {33'b0, abs_x};

// --- State logic for initialization ---
```

```
always@(posedge div_clk) begin
    // ...
    if (div) begin
        if (count == 6'b0) begin
            Q_R_Reg <= initial_Q_R;
        end
            //...
    end
end
```

2. 32 轮“移位-减法-恢复”迭代（Cycle 1 to 32）

从第 1 到第 32 个周期，除法器执行核心的恢复余数算法。每一轮迭代都包含以下步骤：

1. **逻辑左移**：将 65 位的 `Q_R_Reg` 整体左移一位。这个操作有两个目的：a) 将余数部分左移，为“试减”做准备；b) 将商的部分左移，为存入新的商位腾出空间。
2. **试探性减法**：将左移后的余数部分（`shifted_R`）减去除数（`y_extended`）。
3. **判断与恢复**：检查减法结果的借位（`sub_borrow`）。
   (a) 如果 `sub_borrow` 为 1，说明余数不够减（结果为负），本次减法无效。此时，需要恢复余数，即下一周期的余数仍使用减法前的值（`shifted_R`）。同时，本轮的商位记为 0。
   (b) 如果 `sub_borrow` 为 0，说明减法成功。下一周期的余数更新为减法后的结果（`try_r_sub`）。同时，本轮的商位记为 1。
4. **上商**：将计算出的新商位（`~sub_borrow`）填入 `Q_R_Reg` 的最低位。

这个过程会重复 32 次。

Listing 9: 单轮迭代的核心逻辑

```
// --- Combinational logic for one iteration ---
assign shifted_Q_R = Q_R_Reg << 1; // 1. Shift left
assign shifted_R = shifted_Q_R[64:32];

assign try_r_sub = shifted_R - y_extended; // 2. Trial subtraction
assign sub_borrow = try_r_sub[32];

// 3. Judge and restore
assign next_r_val = sub_borrow ? shifted_R : try_r_sub;

// --- State logic for updating the register during iterations ---
always@(posedge div_clk) begin
    // ...
    if (count >= 1 && count <= 32) begin
        Q_R_Reg <= {
            next_r_val,          // Updated remainder
```

```
            shifted_Q_R[31:1],   // Shifted quotient part
            ~sub_borrow          // 4. Set new quotient bit
        };
    end
    // ...
end
```

3. 完成与结果输出（Cycle 33）

在 32 轮迭代结束后（count 到达 33），complete 信号置为高电平，通知 CPU 流水线运算完成。此时，Q_R_Reg 的高 32 位（Q_R_Reg[63:32]）存放的是无符号的余数，低 32 位（Q_R_Reg[31:0]）是无符号的商。最后的输出逻辑会根据第一步预计算的符号 sign_q 和 sign_r，对商和余数进行必要的求补操作，从而得到最终的、符号正确的结果。

Listing 10: 最终结果的符号校正与输出

```
// Signal completion after 32 iterations (count becomes 33)
assign complete = ~div || count == 6'b100001;

// Final sign correction for quotient and remainder
assign q = div_signed ? (sign_q ? ~Q_R_Reg[31:0] + 1'b1 : Q_R_Reg[31:0]) : Q_R_Reg[31:0];
assign r = div_signed ? (sign_r ? ~Q_R_Reg[63:32] + 1'b1 : Q_R_Reg[63:32]) : Q_R_Reg[63:32];
```

设计完成后，我们对这两个模块进行例化，如下所示。

```
mul u_mul(
    .mul_clk(clk),
    .resetn(resetn),
    .mul_signed(op_mulh|op_mul),
    .A(alu_src1),
    .B(alu_src2),
    .result(mul_result)
);
```

(a) 乘法器模块例化

```
// DIV, MOD result
div u_div(
    .div_clk(clk),
    .resetn(resetn),
    .div(op_div|op_mod|op_divu|op_modu),
    .div_signed(op_div|op_mod),
    .x(alu_src1),
    .y(alu_src2),
    .q(div_result),
    .r(mod_result),
    .complete(complete)
);
```

(b) 除法器模块例化

图 7: 在 ALU 中例化乘法器与除法器

最终的 ALU 结果 alu_result 通过一个 ‘case‘语句进行选择。当 alu_op 对应乘法或除法运算时，结果分别来自乘法器或除法器的输出。

```
// final result mux
assign alu_result = ({32{op_add|op_sub   }} & add_sub_result)
                  | ({32{op_slt          }} & slt_result)
                  | ({32{op_sltu         }} & sltu_result)
                  | ({32{op_and          }} & and_result)
                  | ({32{op_nor          }} & nor_result)
                  | ({32{op_or           }} & or_result)
                  | ({32{op_xor          }} & xor_result)
                  | ({32{op_lui          }} & lui_result)
                  | ({32{op_sll          }} & sll_result)
                  | ({32{op_srl|op_sra   }} & sr_result)
                  | ({32{op_mul          }} & mul_result[31:0])
                  | ({32{op_mulh|op_mulhu}} & mul_result[63:32])
                  | ({32{op_div|op_divu  }} & div_result)
                  | ({32{op_mod|op_modu  }} & mod_result);
```

图 8: 根据 alu_op 选择最终运算结果

（二）.2 处理多周期除法指令

除法器需要多个时钟周期完成计算。为处理这一特性，除法器模块会输出一个 `complete` 信号。在 EXE 阶段，我们将这个信号引入到流水线的 `ready_go` 控制逻辑中。

```
alu u_alu(
    .clk        (clk),
    .resetn     (resetn),
    .alu_op     (es_alu_op),
    .alu_src1   (es_alu_src1),
    .alu_src2   (es_alu_src2),
    .alu_result(es_alu_result),
    .complete   (alu_complete)
);
```

图 9: 从 ALU 模块获取运算完成信号 `complete_o`

只有当 `complete` 信号为高电平（表示运算已完成）时，`ready_go` 才能有效，否则 EXE 阶段以及之前的所有阶段都会被暂停（stall），直到运算结束。

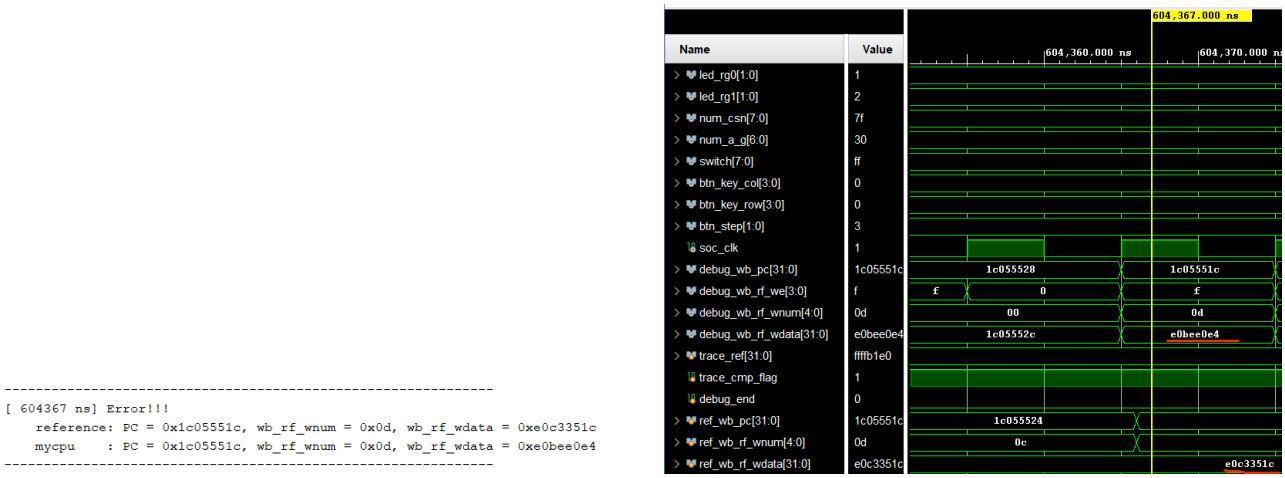
```
assign es_ready_go = alu_complete;
assign es_allowin  = !es_valid || es_ready_go && ms_allowin;
assign es_to_ms_valid = es_valid && es_ready_go;
```

图 10: 使用 ‘complete‘ 信号阻塞流水线

三、 调试过程分析

问题 1

在进行功能仿真时，pcaddu12i 指令的测试案例未能通过，gpr 中出现了非预期的 x 值。



(a) 仿真报错截图

(b) 错误值写入 GPR

图 11: 仿真过程中遇到的错误

通过追溯数据通路，我们定位到问题发生在 EXE 阶段。检查发现，当执行 pcaddu12i 指令时，送入 ALU 的第一个源操作数（op1）的值不正确，导致计算结果错误。

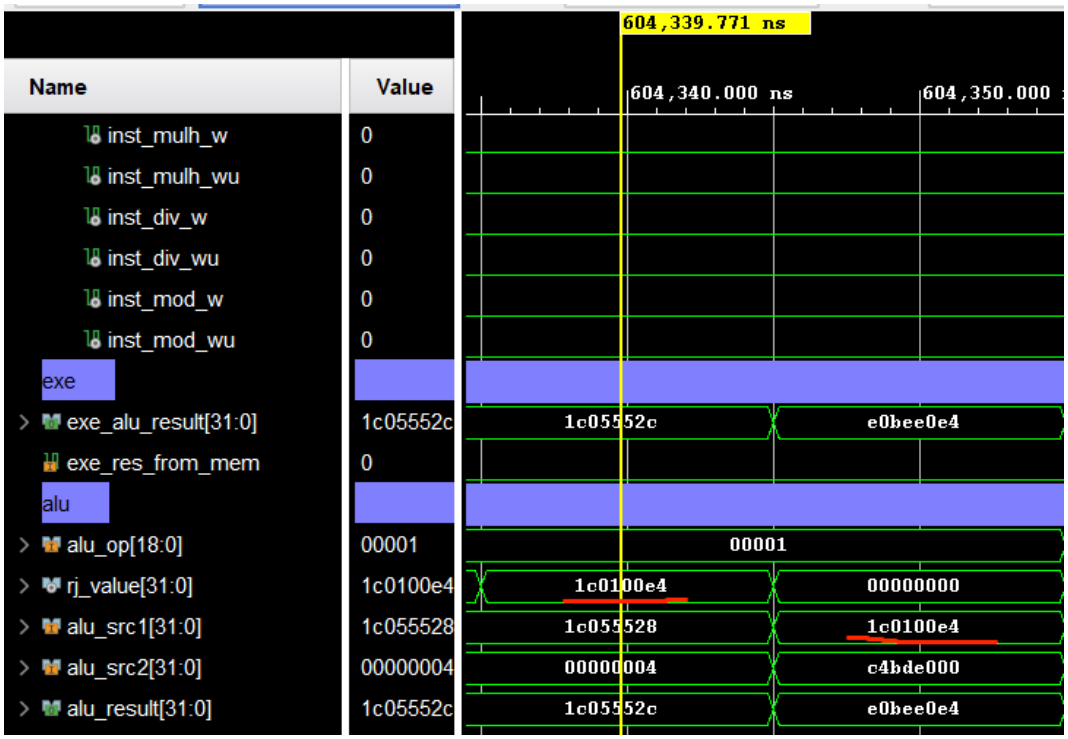


图 12: EXE 阶段的错误数据分析

我们重新查阅了指令集手册，确认 pcaddu12i 指令的第一个源操作数应该是 PC 寄存器的值。检

查 ID 阶段的代码后发现，控制信号 `src1_is_pc` 的生成逻辑中遗漏了对 `inst_pcaddu12i` 的判断。在添加相应逻辑后，问题得到解决。

Listing 11: 修复 src1_is_pc 信号的生成逻辑

```
// 原始错误代码:
// assign src1_is_pc = inst_bl | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu;

// 修复后代码:
assign src1_is_pc = inst_bl | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu |
    inst_pcaddu12i;
```

问题 2

在完成了初步的功能添加后，我们对包含多周期除法指令的程序进行了功能仿真。在调试过程中，发现了一个与流水线控制相关的典型问题。

问题现象：除法指令结果未能正确写回 在仿真测试中，我们发现 `div.w` 等除法指令虽然能够正确计算出结果，但该结果最终未能被写入目标寄存器。通过查看波形图，我们定位到了问题的直接原因：当除法指令的结果数据到达写回（WB）阶段时，对应的寄存器堆写使能信号 `rf_we` 已经变为 0，导致写操作被忽略。

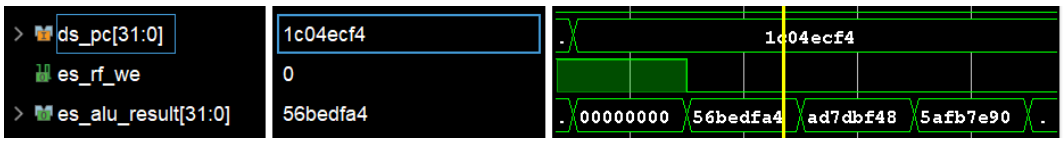


图 13: 意外置零的写使能信号

问题追溯与根本原因分析 `rf_we` 信号是在指令译码（ID）阶段生成的，并随着指令在流水线寄存器中逐级传递。我们追溯信号的传播路径后发现，问题出在 ID 级到 EX 级的流水线控制上。

在最初的设计中，当 `div.w` 指令进入 EX 阶段并开始长达 33 个周期的计算时，虽然流水线发出了暂停信号，但这个信号没能正确地阻止后续指令的控制信号“污染”ID/EX 流水线寄存器。具体流程如下：

1. **Cycle N**：`div.w` 指令在 ID 阶段，生成了正确的控制信号（包括 `rf_we = 1`），并送入 ID/EX 流水线寄存器。
2. **Cycle N+1**：`div.w` 指令进入 EX 阶段并开始计算，流水线检测到多周期操作，开始发出暂停（stall）信号。同时，下一条指令（例如一条非写回指令 `beq`）进入了 ID 阶段。
3. **关键错误点**：在 ID 阶段的 `beq` 指令被译码，生成了它自己的控制信号（包括 `rf_we = 0`）。由于暂停逻辑不完善，这个错误的 `rf_we = 0` 信号在下一个时钟沿覆盖了 ID/EX 流水线寄存器中本应为 `div.w` 指令保留的 `rf_we = 1` 信号。

4. **后续周期**:这个错误的 `rf_we = 0` 信号随着流水线空转,一步步地传递到 EX/MEM 和 MEM/WB 寄存器，最终导致写回失败。

解决方案：基于握手协议的空泡注入（Bubble Injection） 为了解决这个问题，我们修改了 ID/EX 流水线寄存器的更新逻辑：

- **正常传递**：当 ID 级有效且 EX 级允许时（`ds_to_es_valid && es_allowin`），指令和控制信号正常从 ID 传递到 EX。
- **流水线暂停/冻结**：当 EX 级不允许接收时（`es_allowin` 为 0），寄存器更新的两个条件分支都不满足，`always` 块不执行任何赋值操作。这在 Verilog 中意味着所有寄存器（`es_rf_we` 等）会 ** 保持其原有值 **，有效地“冻结”了 `div.w` 指令的正确控制信号。
- **空泡注入**：当 EX 级允许接收，但 ID 级因为上游暂停而没有有效指令发出时（`!ds_to_es_valid && es_allowin`），我们会向 EX 级主动注入一个“空泡”。这个空泡是一个无害的指令，其关键控制信号（如 `es_rf_we` 和 `es_res_from_mem`）被强制清零，确保它在后续流水线阶段中不会产生任何副作用。

Listing 12: ID/EX 级流水线寄存器的“空泡注入”控制逻辑

```
// es_allowin is low when the execute stage is busy (e.g., during division)
// ds_to_es_valid is high when the decode stage has a valid instruction

always @(posedge clk) begin
    if (!resetn) begin
        // ... (Reset logic) ...
        es_rf_we      <= 1'b0;
    end
    // Case 1: Normal operation - Pass instruction from Decode to Execute
    else if (ds_to_es_valid && es_allowin) begin
        es_rf_we      <= ds_rf_we; // Pass the correct rf_we
        // ... (Pass other signals) ...
    end
    // Case 2: Bubble Injection - EX is ready, but DS is not sending
    else if(es_allowin) begin
        // Inject a bubble by clearing critical control signals
        es_rf_we      <= 1'b0;
        es_res_from_mem <= 1'b0;
        // Other signals become "don't care" as no write will happen
    end
    // Case 3: Stall - es_allowin is false. The register freezes,
    // holding its previous value (the div.w instruction's signals).
end
```

通过这套完善的控制逻辑，当 `div.w` 在 EX 级执行时，其正确的 `rf_we = 1` 信号会被可靠地“冻结”在 ID/EX 寄存器中，直到计算完成。后续指令的控制信号无法再对其进行污染。修改后再次仿真，除法指令的结果能够被正确地写入目标寄存器，问题解决。

四、 额外转移和访存指令的添加

(一) 指令的译码与行为

在本次实验中还需要额外添加转移指令 blt、bge、bltu、bgeu 和额外访存指令 ld.b、ld.h、ld.bu、ld.hu、st.b、st.h。

通过查询手册，得到这几条指令的译码逻辑，如下所示：

Listing 13: 添加的指令的译码逻辑

```
assign inst_blt  = op_31_26_d[6'h18];
assign inst_bge  = op_31_26_d[6'h19];
assign inst_bltu = op_31_26_d[6'h1a];
assign inst_bgeu = op_31_26_d[6'h1b];

assign inst_ld_b = op_31_26_d[6'h0a] & op_25_22_d[4'h0];
assign inst_ld_h = op_31_26_d[6'h0a] & op_25_22_d[4'h1];
assign inst_ld_bu = op_31_26_d[6'h0a] & op_25_22_d[4'h8];
assign inst_ld_hu = op_31_26_d[6'h0a] & op_25_22_d[4'h9];

assign inst_st_b = op_31_26_d[6'h0a] & op_25_22_d[4'h4];
assign inst_st_h = op_31_26_d[6'h0a] & op_25_22_d[4'h5];
```

各指令的行为如下图所示：

(a) (b) (c)

图 14: 添加的访存指令的功能

BLT 将通用寄存器 rj 和通用寄存器 rd 的值视作有符号数进行比较，如果前者小于后者则跳转到目标地址，否则不跳转。

```
BLT:
    if signed(GR[rj]) < signed(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, 32)
```

BGE 将通用寄存器 rj 和通用寄存器 rd 的值视作有符号数进行比较，如果前者大于等于后者则跳转到目标地址，否则不跳转。

```
BGE:
    if signed(GR[rj]) >= signed(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, 32)
```

BLTU 将通用寄存器 rj 和通用寄存器 rd 的值视作无符号数进行比较，如果前者小于后者则跳转到目标地址，否则不跳转。

```
BLTU:
    if unsigned(GR[rj]) < unsigned(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, 32)
```

BGEU 将通用寄存器 rj 和通用寄存器 rd 的值视作无符号数进行比较，如果前者大于等于后者则跳转到目标地址，否则不跳转。

```
BGEU:
    if unsigned(GR[rj]) >= unsigned(GR[rd]) :
        PC = PC + SignExtend({offs16, 2'b0}, 32)
```

图 15: 添加的转移指令的功能

（二） 指令的具体实现

（二）.1 转移指令的添加

本次实验添加的四条转移指令，与之前的转移指令的主要区别在于，本次实验的四条指令都是具体的大于小于，而不是等于不等于。

通过分析 blt、bge、bltu 和 bgeu 指令的功能定义，可以得知它们的功能与 beq、bne 非常相似，区别仅在于是否跳转的判断条件。因此添加 blt、bge、bltu 和 bgeu 指令只需要对流水线中已有的转移指令是否跳转的判断逻辑进行扩展即可。其余功能均可直接复用实现 beq 和 bne 的数据通路，相关控制信号的生成也可以参照 beq 和 bne 的控制信号进行。

因此在实现的时候，只需要复用之前 alu 的值，添加小于的判断，大于等于只用 < 即可，所以添加 less 和 less-u 信号，并修改 brtaken 信号。

Listing 14: 转移指令的修改

```
1 assign rj_eq_rd = (rj_value == rkd_value);
2 assign rj_less_rd = ($signed(rj_value) < $signed(rkd_value));
3 assign rj_less_rd_u = (rj_value < rkd_value);
4
5 assign br_taken = ( inst_beq && rj_eq_rd
6                 || inst_bne && !rj_eq_rd
```

```
                || inst_blt && rj_less_rd
                || inst_bge && !rj_less_rd
                || inst_bltu && rj_less_rd_u
                || inst_bgeu && !rj_less_rd_u
                || inst_jirl
                || inst_bl
                || inst_b
               ) && ds_valid;
```

（二）.2 访存指令的添加

在 loadstore 指令这一部分，有许多指令需要传递到下一级进行判断，因此我们首先设计了信号用于传递指令信息。

Listing 15: 传递 loadstore 指令

```
assign mem_inst = {inst_st_b, inst_st_h, inst_st_w, inst_ld_b, inst_ld_bu,inst_ld_h,
    inst_ld_hu, inst_ld_w};//传？loadstore

 else if (ds_to_es_valid && es_allowin) begin
       es_alu_op     <= ds_alu_op;
       es_res_from_mem <= ds_res_from_mem;
       es_alu_src1   <= ds_alu_src1;
       es_alu_src2   <= ds_alu_src2;
       es_rkd_value  <= ds_rkd_value;

       es_rf_we      <= ds_rf_we;
       es_rf_waddr   <= ds_rf_waddr;
       es_pc         <= ds_pc;
       es_ld_inst    <= mem_inst[4:0];
       es_st_inst    <= mem_inst[7:5];
   end

   assign {op_st_b,op_st_h,op_st_w} = es_st_inst;

       else if (es_to_ms_valid && ms_allowin) begin
       ms_pc         <= es_pc;
       ms_alu_result <= es_alu_result;
       ms_res_from_mem <= es_res_from_mem;
       ms_rf_waddr   <= es_rf_waddr;
       ms_rf_we      <= es_rf_we;
       ms_ld_inst    <= es_ld_inst;
   end

   assign {op_ld_b, op_ld_bu,op_ld_h, op_ld_hu, op_ld_w} = ms_ld_inst;
```

传递的指令统一命名为 op-形式

1. ld.b、ld.h、ld.bu、ld.hu 指令的添加

分析 ld.b、ld.h、ld.bu、ld.hu 指令的功能定义并将其与 ld.w 的定义进行比较，可知：

1）它们计算虚地址的操作数来源、地址计算方法、虚实地址映射的规则是完全一样的。

2）它们得到的访存结果都是写回第 rd 项寄存器中。

3）它们和 ld.w 指令的差异仅在于从内存取回的数据位宽不同.

因此我们在实现这四条指令的时候，需要添加用于选择存入数据的代码。

Listing 16: 转移指令的修改

```
assign shift_rdata = {24'b0,data_sram_rdata} >> {ms_alu_result[1:0],3'b0};//
assign ms_mem_result[7:0] = shift_rdata[7:0];
assign ms_mem_result[15:8]= {8{op_ld_b}} & {8{shift_rdata[7]}}|
                        {8{op_ld_bu}} & 8'b0|
                        {8{~op_ld_bu & ~op_ld_b}} & shift_rdata[15:8];
assign ms_mem_result[31:16]={16{op_ld_b}} & {16{shift_rdata[7]}} |
                        {16{op_ld_h}} & {16{shift_rdata[15]}}|
                        {16{op_ld_bu | op_ld_hu}} & 16'b0 |
                        {16{op_ld_w}} & shift_rdata[31:16];
```

shiftrdata 是新定义的信号，其主要效果是将目标字节移到数据的低八位，因此 result 的低八位不需要进行选择，直接移用 shiftrdata 的低八位即可。

b 和 bu 指令只进行一个字节的读取，因此其 result 的 31 到 15 位只需要进行符号的补全，b 是有符号的补全，bu 是零扩展。

对于 h 和 hu 指令，取两个字节，故其 15 到 8 位可以直接选取 shiftrdata 的 15 到 8 位，而 31 到 16 位只需要如前所述进行符号补全即可。

对于 w 指令，取 32 位数据即可。

剩余数据通路，可以直接移用 ld.w 指令的数据通路。

2.st.b 和 st.h 指令的添加

对 st.b 和 st.h 指令的添加，主要修改点在于 mem-we 信号的修改。

Listing 17: we 信号的修改

```
wire [3:0] es_mem_we;
assign es_mem_we[0]  = op_st_w | op_st_h & ~es_alu_result[1] | op_st_b & ~es_alu_result[0]
    & ~es_alu_result[1];
assign es_mem_we[1]  = op_st_w | op_st_h & ~es_alu_result[1] | op_st_b & es_alu_result[0] &
    ~es_alu_result[1];
assign es_mem_we[2]  = op_st_w | op_st_h & es_alu_result[1] | op_st_b & ~es_alu_result[0] &
    es_alu_result[1];
assign es_mem_we[3]  = op_st_w | op_st_h & es_alu_result[1] | op_st_b & es_alu_result[0] &
    es_alu_result[1];
```

对于 st.w 指令，写使能总为 1.

对于 st.h 指令, 写使能在 aluresult 低二位为 00 或 01 时, 低两位为 1, 存入低十六位; 在 aluresult 低二位为 10 或 11 时，低两位为 0，存入高十六位。

对于 st.b 指令，aluresult 为 00 时存入低八位，01 时存入 8 到 15 位，10 时存入 16 到 23 位，11 时存入 24 到 31 位。

剩余数据通路，可以直接移用 st.w 指令的数据通路。