中国科学院大学
《计算机体系结构(研讨课)》实验报告

姓名 陈琛、王荦🄴、张钧玮 箱子号 80 专业 计算机科学与技术
实验项目编号 3 实验名称 添加用户态指令设计专题

一、 实验任务

本次实验的目标是在已有的简单流水线 CPU 基础上,添加更多的普通用户态指令。具体需要实现:

- 算术逻辑运算类指令。包括 slti、sltui、andi、ori、xori、sll、srl、sra 以及 pcaddu12i。
- 乘除运算类指令。包括 mul.w、mulh.w、mulh.wu、div.w、mod.w、div.wu 和 mod.w。
- 转移指令。包括 blt、bge、bltu 和 bgeu。
- 访存指令的扩展。包括 ld.b、ld.h、ld.bu、ld.hu、st.b 和 st.h。

表 1: 新增指令分类

指令类型	**指令说明**
算术逻辑运算类	包括立即数运算、移位操作及 PC 相对地址计算等指令
乘除运算类	实现 32 位整数的乘法和除法运算, 支持有符号和无符号操作
转移指令	添加条件分支指令, 支持有符号和无符号比较跳转
访存指令扩展	支持字节和半字的加载存储操作, 包含符号扩展

二、 设计详细分析

1 总体设计思路

在本次实验中,我们采用了模块化的设计方法,将新增指令的实现分为三个层次进行。首先,通过修改指令译码逻辑, 添加算术逻辑运算、移位操作、条件分支以及访存指令的译码支持, 并完善 ALU 模块以支持这些操作。其次,设计独立的 mul.v 和 div.v 模块,分别实现乘法和除法运算的基本功能。这些模块采用多周期执行的方式,通过移位相加算法实现乘法,通过恢复余数法实现除法,为上层提供了清晰的接口。最后,将独立的乘除法模块集成到流水线 CPU 中,同时协调新增指令与原有流水线结构的兼容性(核心在于处理乘除法指令的多周期执行特性与流水线单周期推进的矛盾)。

2 处理器结构设计框图

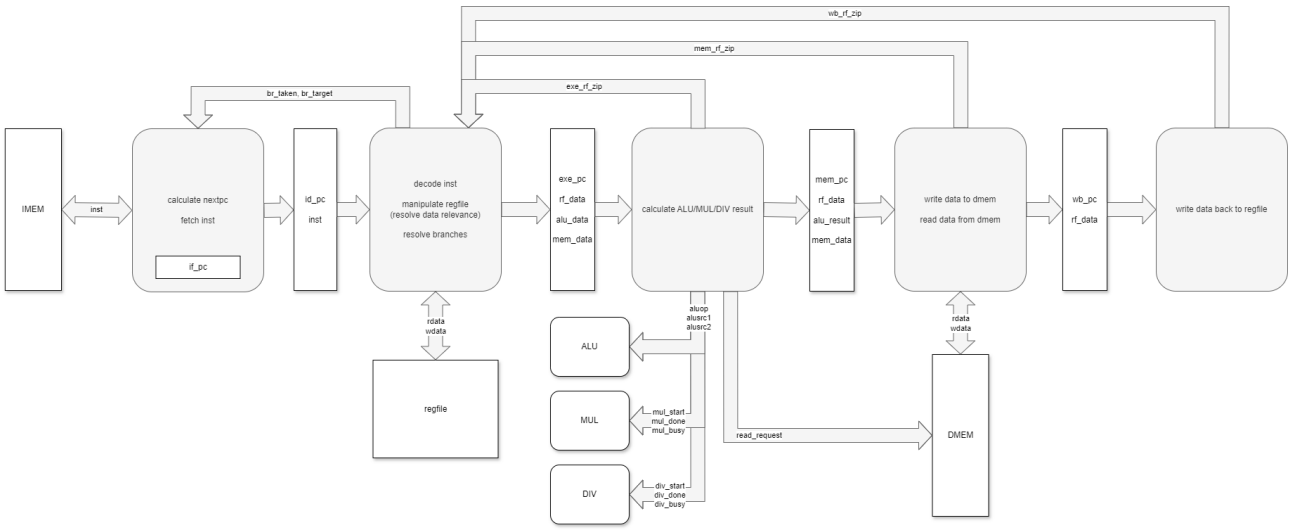

图 1: 处理器结构设计框图

3 关键设计模块

3.1 简单算数逻辑运算、转移与访存指令的实现

- **简单算数逻辑运算**

  对于简单算数逻辑运算，其需要用到的功能部件都已经在之前的实验中实现，故只需要添加相应的译码逻辑和控制信号即可。其中，pcaddu12i 指令需要使用 PC 作为其中一个操作数，为此可以复用转移指令计算链接地址的通路。

- **转移指令**

  新增的四条转移指令都是条件转移，与已经实现的 beq, bne 指令类似，可以复用跳转和冲刷的相关逻辑，只需要添加额外的转移条件计算及选择逻辑。

- **访存指令**

  字节和半字的访存指令需要对总线数据进行额外的处理，以保证访存请求总是按照四字节对齐。对于 load 指令，根据地址最低 2 位选择合适的数据切片，并进行零扩展（ld.bu, ld.hu）或符号扩展（ld.b, ld.h）；对于 store 指令，根据地址最低 2 位将待写入的数据进行移位，并设置写掩码。

3.2 乘除法模块的设计与封装

为支持 mul.w/mulh.w/mulh.wu 与 div.w/mod.w/div.wu/mod.wu，本项目将乘法与除法拆分为两个独立的多周期运算单元，并通过清晰的握手信号与 EXE 阶段对接。二者均以 32 位宽（可参数化 XLEN=32）为核心数据通道，使用移位-累加与恢复余数两类经典迭代算法，保证硬件资源可控与时序友好。

- **模块接口与时序约束**

  两个模块的基本接口如下（节选接口，省略若干内寄存器与中间连线）：

```
// mul.v (32x32 → 64, 支持有/无符号)
module mul #(
    parameter XLEN = 32
```

```
)(
    input wire               clk,
    input wire               resetn,
    input wire               start,      // 单拍启动脉冲
    input wire               signed_mode, // 1: 有符号; 0: 无符号
    input wire [XLEN-1:0]    op_a,
    input wire [XLEN-1:0]    op_b,
    output wire              busy,       // 运算进行中
    output wire              done,       // 单次运算完成 (脉冲)
    output wire [2*XLEN-1:0] product     // 64 位乘积
);
```

```
// div.v (32/32 → 32 商 + 32 余数, 支持有/无符号)
module div #(
    parameter XLEN = 32
)(
    input wire          clk,
    input wire          resetn,
    input wire          start,         // 单拍启动脉冲
    input wire          signed_mode,   // 1: 有符号; 0: 无符号
    input wire [XLEN-1:0] dividend,
    input wire [XLEN-1:0] divisor,
    output wire         busy,          // 运算进行中
    output wire         done,          // 单次运算完成 (脉冲)
    output wire         divide_by_zero, // 除数为 0 标志
    output wire [XLEN-1:0] quotient,   // 商
    output wire [XLEN-1:0] remainder   // 余数
);
```

start 仅在指令发射到 EXE 的首拍拉高；单元据此锁存操作数并拉高 busy，直至迭代结束拉低 busy、同时产生单拍 done。流水线侧通过 exe_ready_go/allowin 与 busy/done 联动，保障多周期期间暂停推进，完成后再释放(对应 3.3 小节的多周期 FSM 控制)。

- **符号处理与绝对值通道**

  两单元均以统一的无符号迭代内核实现有/无符号两种语义。做法是：

  1. 在 `signed_mode=1` 时，先对被乘数/被除数求绝对值(两数取幅值)，内部以无符号路径迭代；

  2. 乘法：最终结果的符号由两操作数符号异或得到；若为负，则对 64 位乘积做二进制补码取反加一；

  3. 除法：内部先得到无符号的商 | 余，最终根据被除数/除数符号恢复商的符号，余数符号与被除数一致(与指令集定义相容)；同时在 `divisor=0` 时置位 `divide_by_zero` 并给出约定结果(实现中为合法的零/保持模式)。

- **乘法：移位-部分积累加(Shift–Add)**

  乘法器采用一位/一拍的移位-累加结构：

  • 迭代寄存器：`acc_r`(累加器，承载部分积)、`multiplicand_r`、`multiplier_r` 与 `count_r`(循环计数，宽度 `COUNT_WIDTH=$clog2(XLEN)+1`)；

  • 每拍操作：检查 `multiplier_r[0]`，若为 1 则 `acc_r += multiplicand_r`；随后 `multiplicand_r` 左移一位，`multiplier_r` 右移一位，`count_r--`；

• 结束条件：当 count_r==1 时进入最后一次累加/移位，下一拍置 done=1、busy=0，product 输出稳定（含已完成的符号修正）。

该结构每次只做一次 32/64 位加法与移位，时序友好、面积较小；代价是潜在的 32 拍延迟。

- **除法：恢复余数法（Restoring Division）**

  除法器采用恢复余数法，一拍处理一位，被除数自高位至低位移入：

  • 关键寄存器：remainder_mag_r（当前余数幅值）、quotient_mag_r（当前商幅值）、dividend_shift_r（被除数移位源）、divisor_mag_r（除数幅值）、count_r；

  • 每拍流程：

  1. 余数左移一位并移入 dividend_shift_r 的下一位；
  2. 试减 divisor_mag_r 得到 remainder_sub；
  3. 若试减结果非负（ge_candidate），则记 1（当前位商为 1），余数采纳 remainder_sub；否则当前位商为 0，余数保留；
  4. count_r--，直到 last_cycle；

3.3 乘除法指令的多周期实现

首先，将 ALU 操作码的位宽从原有的 12 位扩展到 19 位，以支持新增的乘除法指令。在 IDU 模块中，修改 alu_op 的定义，将乘除法指令编码到高位。具体实现中，alu_op[12] 到 alu_op[18] 分别对应 mul.w、mulh.w、mulh.wu、div.w、mod.w、div.wu 和 mod.wu 指令：

```
assign alu_op[12] = inst_mul_w; // MUL.W: 32x32->32 (low part)
assign alu_op[13] = inst_mulh_w; // MULH.W: 32x32->32 (high part, signed)
assign alu_op[14] = inst_mulh_wu; // MULH.WU: 32x32->32 (high part, unsigned)
assign alu_op[15] = inst_div_w; // DIV.W: signed division
assign alu_op[16] = inst_mod_w; // MOD.W: signed modulo
assign alu_op[17] = inst_div_wu; // DIV.WU: unsigned division
assign alu_op[18] = inst_mod_wu; // MOD.WU: unsigned modulo
```

在指令译码方面，添加对乘除法指令类型的识别逻辑如下：

```
wire ty_MD     = op_31_26_d[0] & op_25_22_d[0] & ((inst[21:18] == 4'b0111) | (inst[21:18] ==
    4'b1000));
wire inst_mul_w = ty_MD & op_21_20_d[1] & (inst[17:15] == 3'b000);
wire inst_mulh_w = ty_MD & op_21_20_d[1] & (inst[17:15] == 3'b001);
wire inst_mulh_wu = ty_MD & op_21_20_d[1] & (inst[17:15] == 3'b010);
wire inst_div_w = ty_MD & op_21_20_d[2] & (inst[17:15] == 3'b000);
wire inst_mod_w = ty_MD & op_21_20_d[2] & (inst[17:15] == 3'b001);
wire inst_div_wu = ty_MD & op_21_20_d[2] & (inst[17:15] == 3'b010);
wire inst_mod_wu = ty_MD & op_21_20_d[2] & (inst[17:15] == 3'b011);
```

在执行阶段，由于乘除法的特殊性，需要设计多周期状态机以及握手机制来处理乘除法指令的执行。这部分工作是乘除法指令实现的核心。首先识别出哪些操作是多周期的：

```
assign is_mul_op = exe_alu_op[12] | exe_alu_op[13] | exe_alu_op[14];
assign is_div_op = exe_alu_op[15] | exe_alu_op[16] | exe_alu_op[17] | exe_alu_op[18];
assign is_multicycle_op = is_mul_op | is_div_op;
```

然后设计状态机来控制流水线的暂停和恢复。状态机通过 multicycle_executing 信号来跟踪当前是否正在执行多周期指令：

```
always @(posedge clk) begin
    if (~resetn) begin
        multicycle_executing <= 1'b0;
    end else begin
        if (start_multicycle) begin
            // Start multi-cycle execution
            multicycle_executing <= 1'b1;
        end else if (multicycle_executing) begin
            // Wait for completion of multi-cycle operation
            if ((is_mul_op & mul_done) | (is_div_op & div_done))
                multicycle_executing <= 1'b0;
        end
    end
end
```

流水线的控制逻辑也相应调整，确保在多周期指令执行期间，流水线能够正确暂停：

```
assign exe_ready_go = ~start_multicycle & ~multicycle_executing;
assign exe_allowin = ~exe_valid | (exe_ready_go & mem_allowin);
assign exe_to_mem_valid = exe_valid & exe_ready_go;
```

乘除法模块的启动信号逻辑如下。启动信号只在指令开始执行时维持一个周期：

```
always @(posedge clk) begin
    if (~resetn) begin
        mul_start <= 1'b0;
        div_start <= 1'b0;
    end else begin
        // Start signal active for one cycle when new multicycle instruction arrives
        mul_start <= is_mul_op & start_multicycle;
        div_start <= is_div_op & start_multicycle;
    end
end
```

对于不同的乘除法指令，根据指令类型从乘法或除法的结果中选择正确的数据：

```
assign final_result = exe_alu_op[12] ? mul_product[31:0] : // MUL.W: low 32 bits
                      exe_alu_op[13] ? mul_product[63:32] : // MULH.W: high 32 bits (signed)
                      exe_alu_op[14] ? mul_product[63:32] : // MULH.WU: high 32 bits (unsigned)
                      exe_alu_op[15] ? div_quotient :    // DIV.W: quotient (signed)
                      exe_alu_op[16] ? div_remainder :   // MOD.W: remainder (signed)
                      exe_alu_op[17] ? div_quotient :    // DIV.WU: quotient (unsigned)
                      exe_alu_op[18] ? div_remainder :   // MOD.WU: remainder (unsigned)
                      exe_alu_result;                    // Regular ALU result
```

三、 实验过程中遇到的问题

1 EXEU 多周期流水线信号问题

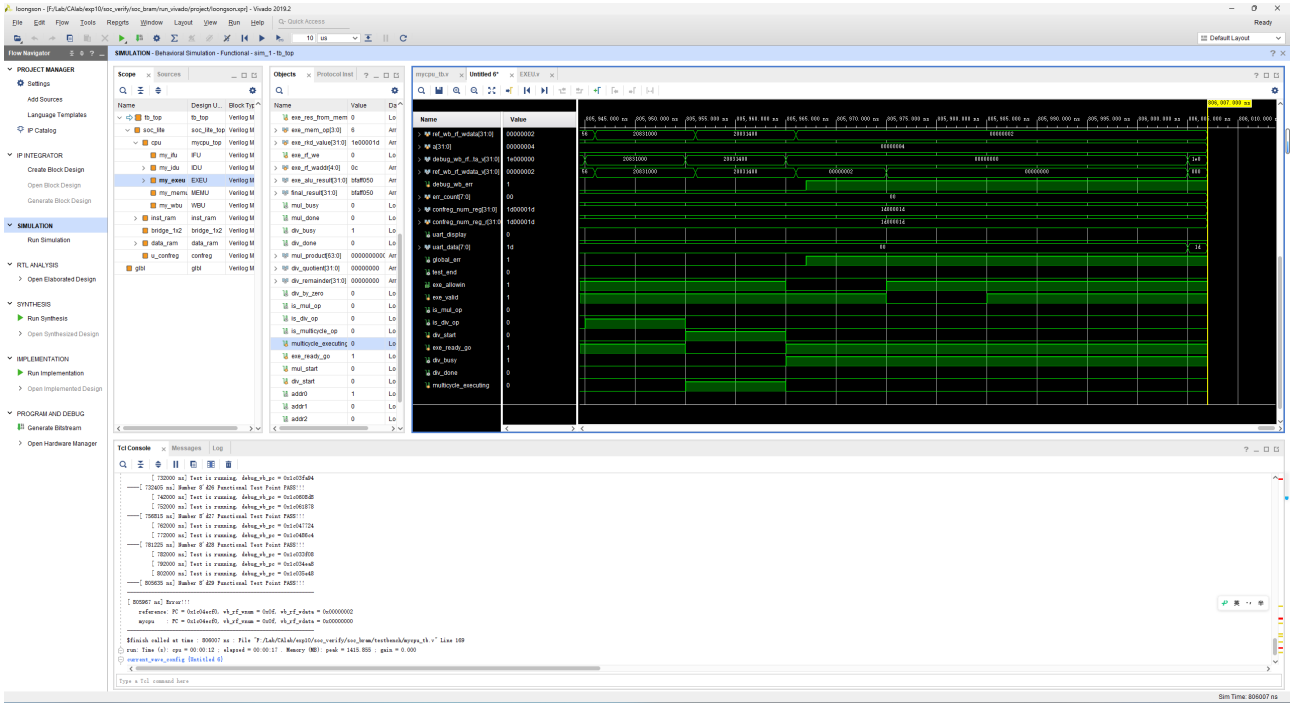

图 2: EXEU 多周期流水线信号问题

在设计多周期 EXEU 模块的初期，流水线 allowin、ready_go 等信号的逻辑频频出错，在波形图中反映为多周期乘除法指令过程中 exe 阶段的 valid、allowin 等信号异常，导致与前后流水级的数据传递出现问题。仔细检查代码后发现，多周期下的 EXEU 不能继续保持原有的 ready_go 信号赋值逻辑（由于原先单周期的特性，ready_go 信号始终被设置为 1）；此外，valid 等其他信号均需要根据多周期特性进行重新修正。

为了更好地区分多周期 EXEU 的执行阶段，同时避免时序逻辑常常“慢一拍”的典型问题，我们引入了 start_exe 寄存器，并通过组合逻辑定义 start_multicycle 信号，用于标记多周期执行的开始。

```
reg start_exe;
always @(posedge clk) begin
    if (id_to_exe_valid & exe_allowin) begin
        {exe_alu_op, exe_res_from_mem, exe_alu_src1, exe_alu_src2, exe_mem_op, exe_rf_we,
            exe_rf_waddr, exe_rkd_value, exe_pc} <= id_to_exe_zip;
        start_exe <= 1'b1;
    end else
        start_exe <= 1'b0;
end
wire  start_multicycle;
assign start_multicycle = is_multicycle_op & start_exe; // Special time stamp
```

此后，根据 start_multicycle 信号，便可以正常对流水线控制信号进行赋值（具体见上文设计分析部分，这里不再赘述）。

2 访存级前递路径过长问题

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	-1.443	16	257	data_...RDCLK	inst_r...DDR[8]	20.299	4.942	15.357	20.0
Path 2	-1.427	16	67	data_...RDCLK	inst_...ARDEN	20.402	4.942	15.460	20.0
Path 3	-1.425	17	29	data_...RDCLK	inst_r...ARDEN	20.856	5.066	15.790	20.0
Path 4	-1.423	16	257	data_...RDCLK	inst_r...DDR[7]	20.279	4.942	15.337	20.0
Path 5	-1.422	17	29	data_...RDCLK	inst_...ARDEN	20.398	5.066	15.332	20.0
Path 6	-1.400	16	257	data_...RDCLK	inst_r...DR[9]	20.439	4.942	15.497	20.0
Path 7	-1.397	16	257	data_...RDCLK	inst_r...DR[8]	20.439	4.942	15.497	20.0
Path 8	-1.394	16	67	data_...RDCLK	inst_r...ARDEN	20.564	4.942	15.622	20.0
Path 9	-1.386	17	16	data_...RDCLK	inst_r...ARDEN	20.366	5.066	15.300	20.0
Path 10	-1.383	16	257	data_...RDCLK	inst_r...DDR[8]	20.430	4.942	15.488	20.0

图 3: 时序违例情况

最初的设计在进行布局布线时，出现了时序严重违例的情况。查看具体路径发现：如果要完全消除 load-to-use 冲突，考虑访存结果参与条件转移的情况，则存在以下的组合逻辑路径：

Data BRAM 输出端 ⟶ 符号扩展 ⟶ IDU 多路选择器 ⟶ 转移条件生成 ⟶ 转移目标生成 ⟶ Inst BRAM 输入端

因此考虑截断上述组合逻辑。出现 load-to-use 冲突时，在 ID 级插入一个气泡，同时不前递 MEM 级的访存结果（但仍需传递 MEM 级指令在 EXE 级生成的结果）。实际访存结果在 WB 级前递。

四、 实验结果

所有测试均通过验证，证明了我们实现的 CPU 能够正确支持所有新增的用户态指令。

五、 小组成员分工

陈琛：实现了除了乘除法以外的用户态指令，进行了仿真和上板测试，并修复了时序违例。

张钧玮：负责乘除法运算单元的设计与调试，调整运算算法以简化电路确保时序友好。

王荤🄵：将乘除法模块集成到流水线 CPU 中，处理乘除法指令的多周期执行特性与流水线单周期推进的矛盾，协调新增指令与原有流水线结构的兼容性。