中国科学院大学
《计算机体系结构(研讨课)》实验报告

姓名 裴晨皓 竹彦博 纪弘璐 学号 2023K8009916003 2023K8009916001 2023K8009916002
专业 计算机科学与技术 实验项目编号 Project 3 实验名称 添加用户态指令设计专题实验

1 实验简介

本次实验的内容主要是在原有的流水线 CPU 中进一步添加算术逻辑运算、乘除法运算、转移和访存这四类用户态指令。在此基础上,本组同学又额外自行完成了除法器模块和两级流水乘法器的设计,并通过了测试。

2 设计方案介绍

2.1 总体思路

借助已有的 CPU 设计,本组同学结合教材讲解与指令集手册内容,把指令的添加分为控制通路、数据通路和功能实现三个部分,充分利用“复用”思想,遵循原有的设计逻辑,向 CPU 中添加新的指令支持。下面将按照设计时分成的三个部分进行介绍。

特别地,由于引入新的模块(乘/除法器),乘除法的功能实现将单独介绍。

下图为经过本次实验修改后的处理器结构框图:

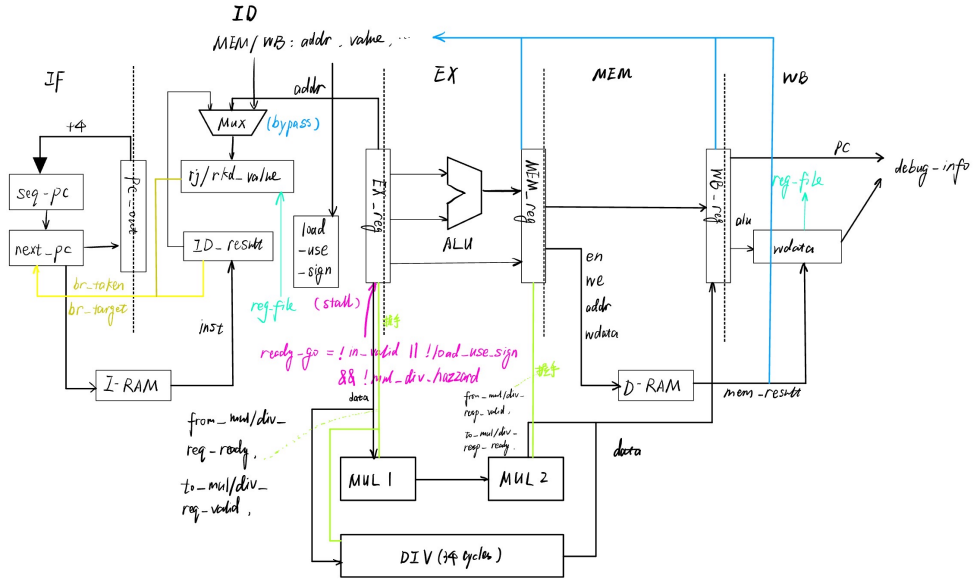


图 1: 处理器结构框图

2.2 控制通路 (宏观)

2.2.1 指令译码与各模块操作码生成 (ID)

延续原有的设计思路,结合指令的操作码,对每一条新添加的指令单独译码,生成专门的控制信号 inst_xxx(例如 inst_div_wu、inst_mod_w 等)。

新增加的指令涉及的部分运算需求，可以充分利用原有 alu 的功能，在生成 alu_op 时采用“或”逻辑，把新增指令的 inst_xxx 信号加入其中。比如新增的 ld_b 计算访存地址时要使用加法，可以用此法使之与原有的 add_w 指令共享 alu 的加法功能。

由于引入的乘除法和访存指令各自都有多条，所以采用 mul_op、div_op 和 mem_op 的方式进行区分，与 alu_op 类似，采用独热码的方式。以 mul_op 为例：

```
assign mul_op = {inst_mulh_wu, inst_mulh_w, inst_mul_w};
```

Listing 1: mul_op(独热码)

2.2.2 数据选择 (主要在 ID)

新增的指令很多都涉及读寄存器操作，所以要把新增指令中以 rd 为目标读寄存器号者的 inst_xxx 信号通过“或”运算加入 src_reg_is_rd 信号（如 bge、bgeu 等），复用 src_reg_is_rd 控制寄存器堆 2 号读端口地址是 rd 还是 rk。

```
assign src_reg_is_rd = inst_beq | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu |
    inst_st_b | inst_st_h | inst_st_w;
```

Listing 2: src_reg_is_rd 信号

同理，用这种“附加并复用”的方法处理 alu 输入操作数的选择信号 src1_is_pc、src2_is_imm，还以此法修改了标记访存指令的 res_from_mem 信号。此外，仿照 res_from_mem，我们又引入了 res_from_div 和 res_from_mul，用于在 EX 和 MEM 阶段生成握手信号。

（判定寄存器写操作指令的 gr_we、判定 store 指令的 mem_we 等信号的修改方式也类似，后文不再赘述。）

2.2.3 “写后读”冲突处理与旁路设计的修改 (ID)

根据课上老师的提示，我们自己设计的乘除法器在性能方面会劣于调用 Xilinx IP 实现出的乘除法部件——关键路径会长一些。如果再加上前递通路传回 ID 参与其他逻辑（比如分支指令条件判断），就会进一步加长关键路径，严重影响性能。因此，为了尽可能减少隐患，我们决定不对乘除法结果采用前递技术——引入控制信号 mul_div_hazzard：

```
assign mul_div_hazzard = in_valid & (
    rf_raddr1 == dest_out && !src1_is_pc && gr_we_out && (res_from_mul_out || res_from_div_out)
        && out_valid ||
    rf_raddr2 == dest_out && !src2_is_imm && gr_we_out && (res_from_mul_out || res_from_div_out)
         && out_valid ||
    rf_raddr1 == MEM_dest && !src1_is_pc && MEM_gr_we && (MEM_res_from_mul || MEM_res_from_div)
        && MEM_valid ||
    rf_raddr2 == MEM_dest && !src2_is_imm && MEM_gr_we && (MEM_res_from_mul || MEM_res_from_div)
         && MEM_valid ||
    rf_raddr1 == WB_dest && !src1_is_pc && WB_gr_we && (WB_res_from_mul || WB_res_from_div) &&
        WB_valid ||
    rf_raddr2 == WB_dest && !src2_is_imm && WB_gr_we && (WB_res_from_mul || WB_res_from_div) &&
        WB_valid
);
```

Listing 3: mul_div_hazzard 信号

这一设计在实现思路上仿照了写后读的判断逻辑，又另外加入了“与 ID 冲突的阶段内部是一条乘法或除法指令”的条件。使用这一信号即可准确判定是否有乘除法指令与当前 ID 的指令发生写后读冲突。

2.2.4 对流水线控制信号的修改 (ID、EX、MEM)

按照原有设计逻辑，每个阶段的 ready_go 信号意味着一种能力的具备，有“can”之含义（而非 will），表示该阶段工作完成，准备好进入下一个阶段（但不一定真的马上（will）进入下一阶段。特别地，如果当前阶段无效，那么 ready_go 也为 1——我们认为无效的“空穴”持续具备移动的能力）。

1. 由于 ID 阶段屏蔽掉乘除法指令的前递，所以在 ready_go 判断中引入了 mul_div_hazzard 信号。如果 mul_div_hazzard 满足，就要一直阻塞 ID 至前面冲突的乘除法指令做完 WB 阶段.

2. EX 和 MEM 涉及与乘/除法器子系统的握手。除了阶段无效可以 ready_go 外，在阶段有效时，只有“既不是没有握手成功的乘法指令，并且也不是没有握手成功的除法指令”条件满足，才可以 ready_go。

具体实现如下：

```
// ID:
assign ready_go = ~in_valid | ~load_use_sign & ~mul_div_hazzard;
// EX:
assign ready_go = !in_valid ||
                      !(res_from_mul && !(from_mul_req_ready && to_mul_req_valid)) &&
                      !(res_from_div && !(from_div_req_ready && to_div_req_valid));
// MEM:
assign ready_go = !in_valid ||
                      !(res_from_mul && !(to_mul_resp_ready && from_mul_resp_valid)) &&
                      !(res_from_div && !(to_div_resp_ready && from_div_resp_valid));
```

Listing 4: 修改后的 ready_go

2.3 数据通路 (宏观)

2.3.1 “result”的收集

原先，在 EX 得到 alu 的运算结果，WB 得到 load 的结果，要在 WB 使用选择器根据指令类型选出写回寄存器的结果。现在增加乘除法器模块后，会在 MEM 阶段收集到乘/除法结果，所以在 MEM 阶段要额外加入对 alu、乘法器、除法器（商和余数）的结果选择，然后传递到 WB 阶段和 load 的结果继续进行选择。

2.3.2 流水寄存器的数据传递 (这时“控制信号”也可视为被传递的数据)

在 ID 阶段增加的许多信号要在后续阶段使用，于是要通过流水寄存器逐级传递下去，本次新增的 mul_op、div_op 等信号要传到计算乘除法的 EX 和 MEM，mem_op 要传到 MEM 和 WB 用于发送访存请求和接收 load 结果。

2.4 功能实现 (针对具体不同类型指令特点)

2.4.1 算术逻辑运算指令

新增的移位、逻辑运算、算术运算都能够复用 alu 的功能，利用原有算术逻辑运算指令的数据和控制通路进行运算。特别地，新增的 andi、ori 和 xori 指令用到了零扩展 i12 的立即数，只要新增控制信号 need_ui12，对 imm 的生成稍作修改即可。

2.4.2 访存指令

在 MEM 阶段，由于引入了半字和字节访存，所以目标地址可能不是 4 字节对齐的，要把最低两位清 0，得到实际访存地址：

```
assign data_sram_addr  = result & ~32'b11;
```

Listing 5: 访存地址

如果是非“全字”的 store 指令，根据操作的字节数 (由 mem_op 得知) 和目标地址相对实际访问的起始地址的偏移量（目标地址 result 的末两位）形成字节掩码（字节写使能位），发送给数据内存：

```
assign data_sram_we = {4{mem_we && valid && in_valid}} & (
                        ({4{mem_op[5]}} & (4'b0001 << result[1: 0])) |  // SB
                        ({4{mem_op[6]}} & (4'b0011 << result[1: 0])) |  // SH
                        ({4{mem_op[7]}} & 4'b1111)  // SW;
                    );
```

Listing 6: store 字节写使能

有了写使能的限制，就可以采取教材中提到的较优的发送 store 数据的方法——将 Byte 重复 4 次，将 Half Word 重复 2 次，无论地址的相对偏移如何，结合写使能，选到的字节/半字总是从指令源寄存器读出的最低字节/半字：

```
assign data_sram_wdata = {32{mem_op[5]}} & {4{rkd_value[7:0]}} |
                            {32{mem_op[6]}} & {2{rkd_value[15: 0]}} |
                            {32{mem_op[7]}} & rkd_value;
```

Listing 7: store 数据

如果是非“全字”的 load 指令，依旧先根据 mem_op 对数据的字节数（Byte/Half Word）分类，再借助地址偏移（result 的末两位）形成多路选择器，在从内存得到的 4 字节数据中选出需要的部分，进行位扩展后形成“mem_result”：

```
assign mem_result   =
        {32{mem_op[0] | mem_op[3]}} &   // LB & LBU
            ({32{result[1: 0] == 2'b00}} & {{24{mem_op[0] & data_sram_rdata[7]}},
                data_sram_rdata[7: 0]} |
            {32{result[1: 0] == 2'b01}} & {{24{mem_op[0] & data_sram_rdata[15]}},
                data_sram_rdata[15: 8]} |
            {32{result[1: 0] == 2'b10}} & {{24{mem_op[0] & data_sram_rdata[23]}},
                data_sram_rdata[23: 16]} |
            {32{result[1: 0] == 2'b11}} & {{24{mem_op[0] & data_sram_rdata[31]}},
                data_sram_rdata[31: 24]}) |
        {32{mem_op[1] | mem_op[4]}} &   // LH & LHU
            ({32{result[1: 0] == 2'b00}} & {{16{mem_op[1] & data_sram_rdata[15]}},
                data_sram_rdata[15: 0]} |
            {32{result[1: 0] == 2'b10}} & {{16{mem_op[1] & data_sram_rdata[31]}},
                data_sram_rdata[31: 16]}) |
        {32{mem_op[2]}} & data_sram_rdata;  // LW
```

Listing 8: load 数据

位扩展的方法比较巧妙，以 LH/LHU 的情况为例：将 mem_op[1] 和 load 出的半字数据“与”在一起，然后扩展。mem_op[1] 为 1，代表 LH，是有符号的，“与”运算结束后得到的就是数据的符号位；若是无符号的 LHU，则 mem_op[1] 为 0，“与”之后必为 0，位扩展时也自然全用 0 填充。

2.4.3 转移指令

新增的转移指令条件判断需要比较两个寄存器值的大小关系，而非仅仅判断是否相等。我们仿照 rj_eq_rd 信号，引入了另外两个用于无符号数和有符号数比大小的信号：

```
assign rj_lt_rd = ($signed(rj_value) < $signed(rkd_value));
assign rj_ltu_rd = (rj_value < rkd_value);
```

Listing 9: 转移指令条件判断信号

然后根据具体指令的 inst_xxx 和条件判断结果，复用判断转移与否的 br_taken 和 br_target 信号即可。

2.4.4 乘除法指令

首先是 CPU 与乘除法器这两个“子系统”的交互——握手 (EX、MEM)：

做乘除法指令时，CPU 利用两个流水级进行运算，在 EX 级向相应运算模块发送数据，在 MEM 级接收结果。如果把乘/除法器视为正常的 (例如物理内存等) 子系统，那么它们与 CPU 进行交互时就需要用到握手信号。

在 EX 阶段，以除法指令为例，CPU 使用 req_valid 发送运算请求，申请向除法器发送数据：

```
assign to_div_req_valid = in_valid && res_from_div;
```

Listing 10: CPU 的除法运算请求

只要 EX 有效 (in_valid) 且内部是除法指令 (res_from_div)，就应当发请求。

进而，在 MEM 阶段，CPU 使用 resp_ready 向除法器表明自己准备好接收结果：

```
assign to_div_resp_ready = in_valid && res_from_div;
```

Listing 11: CPU 的除法运算接收准备

同理，如果 MEM 有效且在做除法指令，就能够接收结果。

乘/除法器的具体设计见后文阐述。

2.5 乘法器

按照教材上提到的采用 3:2 压缩的华莱士树，我们按照给出的电路图进行连线先实现了组合逻辑乘法器模块 (module multiplier)，然后在汪老师的建议下从华莱士树的第 2、3 层之间切分为了二级流水乘法器。

2.5.1 乘法器接口

模块接口的定义如下：

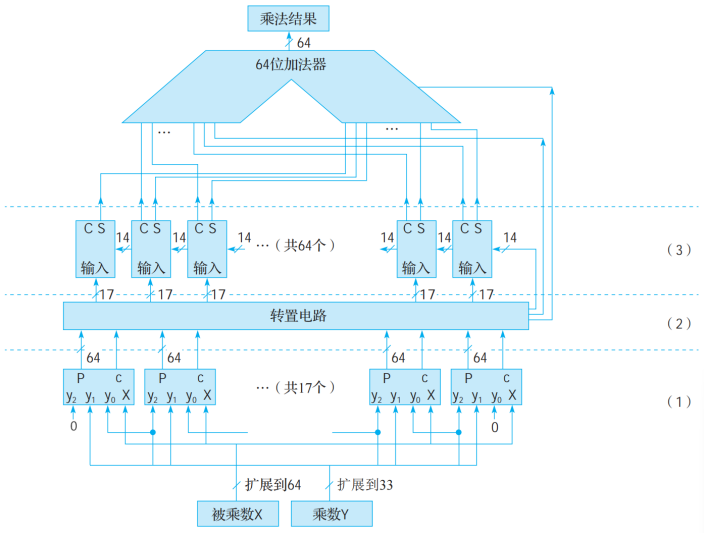
名称	方向与类型	位宽	含义
mul_clk	in(wire)	1	时钟信号
reset	in(wire)	1	复位信号
mul_op	in(wire)	3	操作码
x	in(wire)	32	乘数 1
y	in(wire)	32	乘数 2
to_mul_req_valid	in(wire)	1	握手信号，表明输入有效
from_mul_req_ready	out(wire)	1	握手信号，表明准备好接收输入
to_mul_resp_ready	in(wire)	1	握手信号，表明外界准备好接收乘法器的输出
from_mul_resp_valid	out(wire)	1	握手信号，表明运算完成，输出有效
result	out(wire)	64	积

表 1: 乘法器模块接口

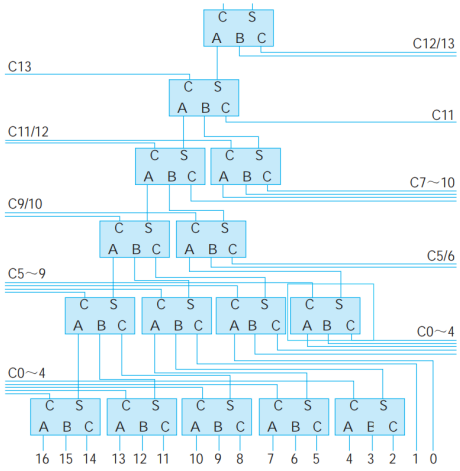
2.5.2 组合逻辑乘法器结构框图及实现原理

按照教材的说明，我们实现了一个既可以做有符号乘法运算，又可以做无符号乘法运算的 33 位有符号乘法器 (模块输入仍为 32 位，在内部增加一位是为了兼具有无符号运算的本领)。

此处引用书上的内容：下面是乘法器以及其核心部件“华莱士树”的结构框图：



(a) 33 位定点补码乘法器结构



(b) 17 个部分积相加的 1 位华莱士树 (左侧“C5~9”应为“C5~8”)

图 2: “乘法器总体结构”与“华莱士树”框图

注：教材未给出优化后的 33 位定点补码乘法器结构图，此处的结构图（b）由教材的 32 位版本修改而来。

实现上，在除法器模块 multiplier 内部，我们分别设计了 wallace（1 位华莱士树）模块、booth（两位 booth 乘法部分积生成）模块以及起运算辅助作用的 full_adder（1 位全加器）模块。基本完全按照框图所示的电路图进行连线，故此处不额外展示代码。

如上图中的图（a）所示，第（1）部分实例化了 17 个 booth 模块，运算得到 17 个 64 位的部分积，送给第（2）部分的转置电路（在顶层模块 multiplier 内部直接实现），转化成 64 组 (在部分积相加时) 每一位上的 17 个加数，送给第（3）部分。第（3）部分实例化 64 个 wallace，wallace 内部利用“保留进位加法”原理进行 3:2 压缩，通过 6 层压缩，将 17 个加数变成 2 个（最终的和 S 与进位 C），而中间产生的 14 个进位信号 C0 ~C13 则传递给对更高 1 位进行计算的 wallace。

最后用一个 64 位加法器将每一位的 S 与来自低位的进位 C 相加，得到最终的乘积 result。

2.5.3 流水化改进

在华莱士树内部进行流水化拆分时，每一层产生的 S 信号直接向更高层传递即可，跨流水级处使用流水寄存器进行过渡，逻辑十分简单。而每一棵华莱士树运算过程中产生的进位信号 C0 ~C13，需要传给下一棵树，这就涉及到不同华莱士树模块之间的数据通路设计，应当特别注意。为了使情况更加直观，下面给出相邻华莱士树数据通路示意图：

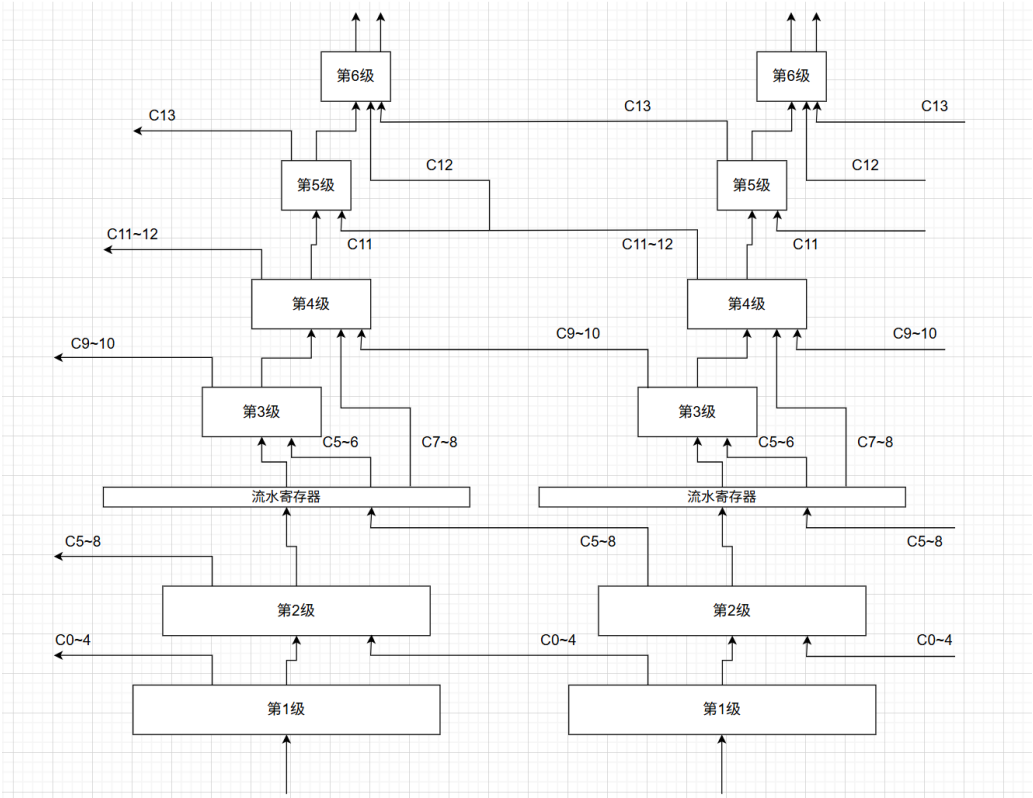


图 3: 相邻两棵华莱士树的数据通路（流水化）

切分流水时，我们不能以“棵”为单位看待华莱士树，而是应该以另一种视角，将这 64 棵树视作“森林”，考虑森林上下两个部分（第 1、2 层和第 3、4、5、6 层）之间的数据通路：在这两个部分之间插入流水寄存器，所有下半部分输出的数据都应当先存入流水寄存器，然后才能给上半部分使用。结合上面流水化后的数据通路示意图，代码自然不难写出。

结合乘法器“当前周期输入，下一周期输出”的特点，设计如下的乘法器控制信号（带有 req 和 resp 的为握手信号）：

```
    assign do_mul = to_mul_req_valid && from_mul_req_ready;
    always @(posedge mul_clk) begin
        if(reset) begin
            from_mul_resp_valid_reg <= 1'b0;
        end
        else if(do_mul) begin
            from_mul_resp_valid_reg <= 1'b1;
        end
    end
    assign from_mul_resp_valid = from_mul_resp_valid_reg;
    assign from_mul_req_ready = to_mul_resp_ready;
```

Listing 12: 乘法器控制信号

1. 如果输入握手成功（req），则 do_mul 为 1，允许乘法器流水线流动（进行运算），接受当前新的输入，在第二级算出新的结果。
2. 进行了乘法运算（do_mul），那么输出结果（resp）就一定会变为有效（valid）。
3. 如果外界能够接受（resp_ready）乘法器第二级的输出结果，那么乘法器也就做好了接收新输入的准备（req_ready）。

下面是流水寄存器的设计：S2_reg 保存第 1 级流水的最终输出 S2；Cin_reg 保存前一棵树送来的进位信号 Cin：

```
    // register for pipeline
    reg [13:0] Cin_reg;
    reg [ 3:0] S2_reg;
    always @(posedge mul_clk) begin
        if(reset) begin
            Cin_reg <= 14'd0;
            S2_reg <= 4'd0;
        end
        else if(do_mul)begin
            Cin_reg <= Cin;
            S2_reg <= S2;
        end
    end
```

Listing 13: 华莱士树流水寄存器

然后结合图 3 按需连线，形成第二级流水中华莱士树每一层的输入——以第 4 层为例：

```
assign in4 = {S3, Cin[10:9], Cin_reg[8:7]};
```

Listing 14: 华莱士树第 4 层的输入

需要特别注意：第二级流水内，并非所有的进位输入都来自流水寄存器，有些进位信号是跨越流水级传递的（C7、C8），还有一些只在同一流水级内传递（C9、C10）——同级内部自然也没有流水寄存器可言。。

2.6 除法器

除法器作为单独的一个模块 (module Div)，由本组同学尝试使用 chisel 进行实现，转化成 Verilog 后接入到流水线 CPU 中。

按照教材的指导，采用的运算方法为：做无符号数除法时，采用恢复余数法；如果是有符号数除法，则先用两个输入操作数的绝对值进行无符号数除法，然后根据被除数和除数的符号确定商和余数的符号。

2.6.1 除法器接口

模块接口定义如下：

名称	方向与类型	位宽	含义
clock	in(wire)	1	时钟信号
reset	in(wire)	1	复位信号
io_in_valid	in(wire)	1	握手信号，表明输入有效
io_in_ready	out(wire)	1	握手信号，表明准备好接收输入
io_out_ready	in(wire)	1	握手信号，表明外界准备好接收除法器的输出
io_out_valid	out(wire)	1	握手信号，表明运算完成，输出有效
io_in_bits_divOp	in(wire)	4	操作码
io_in_bits_dividend	in(wire)	32	被除数
io_in_bits_divisor	in(wire)	32	除数
io_out_bits_quotient	out(wire)	32	商
io_out_bits_remainder	out(wire)	32	余数

表 2: 除法器模块接口

其中，握手信号作为控制信号，在握手成功 (in(或者 out) 的 valid 和 ready 同时拉高) 时，数据通路对数据进行传递，in 握手成功则数据进入除法器，out 握手成功则从除法器输出计算结果。

2.6.2 除法器结构框图

下图为除法器结构框图：

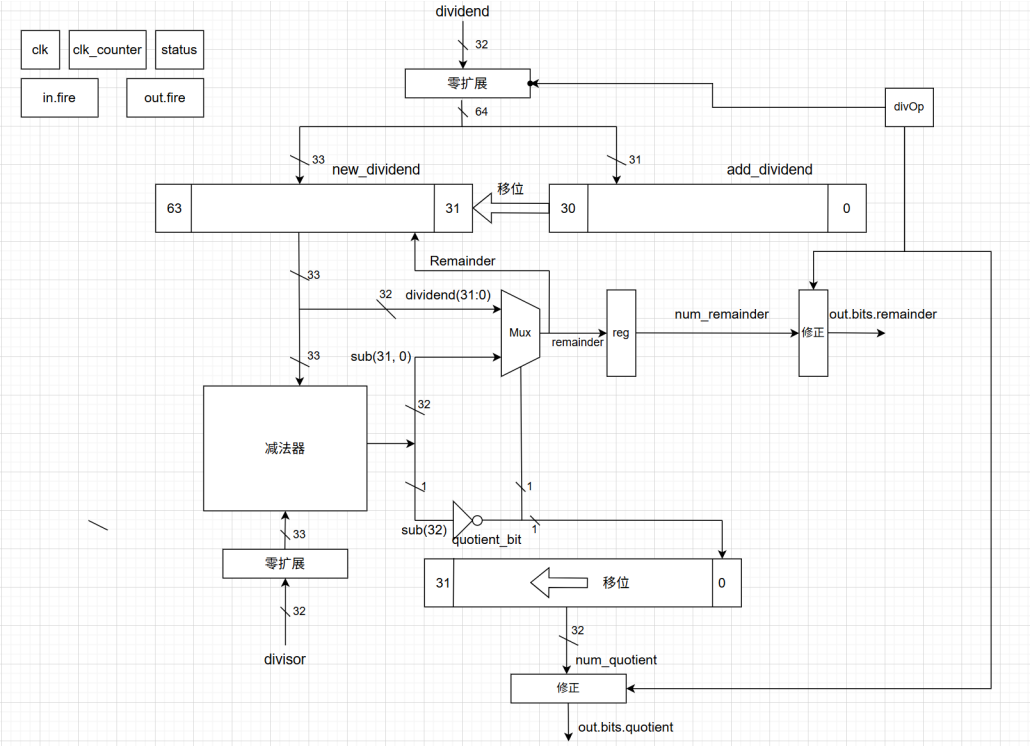


图 4: 除法器结构框图

2.6.3 除法器设计思路

我们引入了 Chisel3 基本构件，借助 Decoupled(握手)、Cat(按位拼接)、Fill(重复位填充) 工具进行设计。此外，定义了 BitUtils、Status、DivOp 这三个对象作为辅助，分别用于操作数调整、状态定义和操作码定义。BitUtils 中，实现了用于对操作数进行 0 扩展的 zext 函数，以及取绝对值的 abs 函数。Status 对象内定义的两个状态用于区分除法器空闲 (IDLE) 还是正在工作 (BUSY)；DivOp 对象内用独热码区分了四种除法操作 DIV(div.w)、DIVU(div.wu)、REM(mod.w)、REMU(mod.wu)。这部分代码逻辑较为直白，此处不浪费篇幅。

在除法器模块内部，把 divOp(操作码)、dividend(被除数) 和 divisor(除数) 三者打包进 DivReq，作为“输入数据包 (in)”；把 quotient(商) 和 remainder(余数) 打包进 DivResp，作为“输出数据包 (out)”。借助 Flipped 和 Decoupled 工具，使输入输出附带握手信号，io.in.fire 代表输入握手成功，io.out.fire 代表输出握手成功。

Listing 15: IO 接口

```
val io = IO(new Bundle {
        val in = Flipped(Decoupled(new DivReq()))
        val out = Decoupled(new DivResp())
    })
```

除法器发给外界的握手信号为 io.in.ready 和 io.out.valid——当除法器在 IDLE 空闲状态时，显然应该准备好接收输入；当除法器工作 (BUSY) 到完成全部计算时 (clk_counter 用于记录迭代运算周期数，到 32 拍时完成)，输出结果有效：

Listing 16: 除法器发向外部的握手信号

```
io.in.ready := status === Status.IDLE
io.out.valid := (clk_counter === 32.U) && (status === Status.BUSY)
```

试减操作是恢复余数除法的核心内容，专门定义一个组合逻辑模块来完成：

Listing 17: 试减模块 (div_iter)

```
def div_iter(dividend: UInt, divisor: UInt): (Bool, UInt) = {
        val sub = Wire(UInt(33.W))
        sub := dividend - divisor
        val quotient_bit = (sub(32) === 0.B)
        val remainder = Wire(UInt(32.W))
        remainder := Mux(quotient_bit, sub(31, 0), dividend(31, 0))
        (quotient_bit, remainder)
}
```

这一模块把 64 位被除数中参与相减的 33 位与 33 位的除数相减，得到结果 sub，根据 sub 的符号位确定当前上商的一位商值 quotient_bit(sub 为正则够减，上商 1；反之上商 0)，然后根据 quotient_bit 确定当前步骤的余数 remainder，利用 Mux 选择减法结果 sub 或是还原成原来的 dividend(恢复余数)。

完整运算的过程则通过 when-elsewhen 的组合，用时序逻辑实现 (类似状态机，不被阻塞的情况下共 34 拍)：

1. 数据输入：

Listing 18: 数据输入

```
    val status = RegInit(Status.IDLE)
    when (io.in.fire) {
        status := Status.BUSY
        dividend := io.in.bits.dividend
        divisor := io.in.bits.divisor
        divOp := io.in.bits.divOp
        clk_counter := 0.U
        val zext_in_dividend = Wire(UInt(64.W))
        zext_in_dividend := BitUtils.zext(Mux(io.in.bits.divOp === DivOp.DIV || io.in.bits.
            divOp === DivOp.REM, BitUtils.abs(io.in.bits.dividend), io.in.bits.dividend),
            32, 64)
        new_dividend := zext_in_dividend(63, 31)
        add_dividend := zext_in_dividend(30, 0)
    }
```

初始时，状态机处于空闲的 IDLE 状态。当输入握手成功时，进入 BUSY 工作状态，将输入的操作数和 divOp 保存在寄存器中，并把取绝对值后的被除数零扩展至 64 位，其中高 33 位 new_dividend 作为被除数中试减的部分，其余 31 位放在 add_dividend 中用于后续补充，用于记录运算周期数的 clk_counter 计数器初始化为 0——在第 1 拍，完成了准备工作。

2. 迭代运算 (试减)：

Listing 19: 迭代试减 (num_quotient 和 num_remainder 用于保存最终结果)

```
    .elsewhen (clk_counter < 32.U && status === Status.BUSY) {
        when (divisor === 0.U) {
            clk_counter := 32.U
        }.otherwise {
            val (quotient_bit, remainder) = div_iter(new_dividend, zext_divisor)
            num_quotient := (num_quotient << 1) | quotient_bit
```

```
            num_remainder := remainder
            new_dividend := Cat(remainder(31, 0), add_dividend(30))
            add_dividend := add_dividend << 1
            clk_counter := clk_counter + 1.U
        }
    }
```

在 BUSY 状态下，如果 clk_counter 小于 32(一共要算 32 拍，此时相当于未完成运算)，则继续运算：

(a) 如果除数为 0，是非法的运算，直接把计数器置为 32，使得下一拍直接结束运算，退出。

(b) 其他正常情况下，调用 div_iter 产生当前商位 (quotient_bit) 和余数 (remainder)，由于商位要拼接到已得到的部分商值 (num_quotient) 的最低位，所以采用把部分商值先左移一位，再用或运算拼接的方式来处理。

还要更新被除数中用于试减的 33 位部分 (new_dividend)：舍弃 new_dividend 的最高位，取用于补充的 add_dividend 的最高位拼接在 new_dividend 的最低位。为了便于在更新 new_dividend 时，每次都取 add_dividend 的最高位进行补充，所以 add_dividend 在迭代阶段每次左移一位。

此外，计数器 clk_counter 也要加 1。

3. 结束处理 (1 拍)：

```
    .elsewhen (clk_counter === 32.U && io.out.fire) {
        status := Status.IDLE
        clk_counter := 0.U
        num_quotient := 0.U
        new_dividend := 0
    }
```

如果计数器达到 32(已完成运算)，且输出握手成功，则把状态机重新置为 IDLE 并清空保存最终结果的寄存器，等待新的输入。

最后，使用 Mux 选择器根据除法操作的类型和被除数、除数的符号，确定商和余数的符号作为最终输出即可。

特别地，除 0 在 LoongArch 指令集中是未定义行为，在我们的除法器中采用了和 risc-v 相同的处理：商为 0xFFFF_FFFF，余数与被除数相同。用 Mux 把这种情况加入到最终输出的生成逻辑即可。

3 Debug 记录

3.1 IF 阶段 seq_pc 与 nextpc 逻辑的修正

在检查 IF.v 的相关代码时，我们发现了可通过仿真测试用例但不够严谨的控制逻辑。

在之前的设计中：

```
    assign seq_pc       = PC_out + 32'h4: PC_out;
    assign nextpc       = br_taken ? br_target : seq_pc;
```

事实上，如果有一条跳转指令在 ID 阶段被阻塞，next_pc 仍会根据跳转结果更新，并进行取指，这将导致取回的指令无法被 ID 接收。因此，取指的请求应该在 out_ready = 1 时才可以发送。因此逻辑更改为

```
    assign seq_pc       = out_ready ? PC_out + 32'h4: PC_out;
    assign nextpc       = out_ready && br_taken ? br_target : seq_pc;
```

3.2 除法器输入信号未保持

最初的设计中，我们未把在 EX 输入除法器的信号单独存入 reg，而是直接用传入除法器的 wire 信号开始运算，导致出现了信号未保持的 bug。

具体情况为：EX 的除法指令把数据传给除法器。时钟上升沿到来，MEM 中的指令向后流动，同时 EX 也流动，ID 也流动，从 ID 进入到 EX 的另一条指令覆盖了流水寄存器原来的内容，导致 EX 传给除法器的 wire 信号发生变化，不再是原来那条除法指令正确的操作数和操作码，导致除法器算出错误结果。

3.3 其他

由于添加了多条指令，要对数据通路和控制通路的许多信号进行修改，细节较多，所以出了一些类似于数据 SRAM 字节写使能设置错误、ALUop 信号未覆盖新增指令、booth 乘法部分积生成逻辑错误等低级失误，带来了不少调试上的麻烦，此处不再赘述。

4 合作说明

本实验由本组同学共同完成，组内成员同等贡献。