LoongArch 5 级流水线 CPU 实验报告

PB22020514 郭东昊

2024.4.19

一、实验目的与内容

此次实验的主要内容为:完成一个支持 32 位 LoongArch 精简指令集中 39 条指令的 5 级流水线 CPU，并完成电路的仿真、分析。

实验的目的为:

- 熟练掌握流水线 CPU 的结构和工作原理，特别是对流水线中的数据冒险和控制冒险的处理
- 掌握流水线 CPU 的设计和调试方法
- 熟练掌握数据通路和控制器的设计和 Verilog 描述方法
- 提高用 Verilog 描述时序逻辑电路的能力
- 深化分模块和分层次的逻辑电路设计思维

二、数据通路与状态图

本实验中，我编写的流水线 CPU 的数据通路如下（除少量控制信号外与 PPT 一致）:

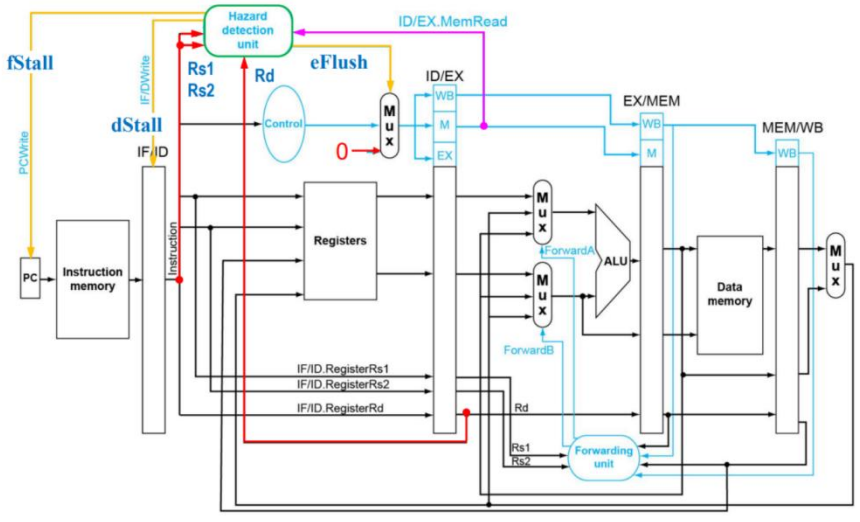


图 1 支持 40 条指令的 LoongArch 5 级流水 cpu 数据通路

其中，InstMem，DataMem，regfile 均为异步读同步写。

此次实验在上次的单周期 CPU 基础上做的修改有:

增加了大量流水线寄存器，用于在两个流水线级之间传递数据及控制信号。在此基础上，对大量的模块接口进行了重接，使得模块在正确的流水线寄存器处读写数据与控制信号。

修改了寄存器堆逻辑，使得读取的寄存器正好为即将写入的寄存器时，读到的是即将写入的值，以此解决旧有实现的结构冒险问题（寄存器写优先）。

添加了数据前递单元，即 forwarding_unit。若 EX 级产生的数据即将写入某非零寄存器，而下一条指令 ID 级中 rs1（rs2）端口正好读的是该写入值，则 EX 级结果为最新数据，设 afwd（bfwd）为 2'b10;若 MEM 级产生的数据即将写入某非零寄存器，而下一条指令（会被阻塞一周期，在阻塞模块里实现）ID 级中 rs1（rs2）端口正好读的是该写入值，则 MEM 级结果为最新数据，设 afwd（bfwd）为 2'b01;其他情况设 afwd（bfwd）为 2'b00。最终，根据 afwd（bfwd）将寄存器堆读出值设为最新的值以供之后器件（如 ALU）利用。

添加了阻塞模块，详见核心代码对 hazard_detect 模块的解析。

三、核心代码

译码逻辑

译码模块参照龙芯官方指导书提供的译码单元代码，可读性强，结构清晰，易于维护。下面以指令 add.w 的译码为例阐释该代码的工作原理。

指令 add.w 的指令码如下图：

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RDCNTID.W	rj	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	rj				0	0	0	0	0
RDCNTVL.W	rd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	rd			
RDCNTVH.W	rd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	rd			
ADD.W	rd, rj, rk	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	rk						rj					rd			

根据龙芯架构的指令码设计特色，将指令码 inst 分为[31:26]， [25:22]，[21:20]， [19:15]四段，分别命名为 op_31_26，op_25_22，op_21_20，op_19_15，通过译码器，将其译为独热码，如位宽为 6 的 op_31_26，经过 6-64 位译码器，变成位宽为 64 的 op_31_26_d，以此类推，生成 op_25_22_d，op_21_20_d，op_19_15_d。这样做的好处是，被截取的指令片段的值，成为了其对应独热码中为 1 的那一位的下标。这样，add.w 指令的信号变量 inst_add_w 便可被唯一表示为

```
inst_add_w      = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
                  op_21_20_d[2'h1] & op_19_15_d[5'h00];
```

以此类推，所有指令的信号变量都可以这样表示。

```
decoder_6_64 u_dec0(.in(op_31_26),  .co(op_31_26_d));
decoder_4_16 u_dec1(.in(op_25_22),  .co(op_25_22_d));
decoder_2_4  u_dec2(.in(op_21_20),  .co(op_21_20_d));
decoder_5_32 u_dec3(.in(op_19_15),  .co(op_19_15_d));

assign inst_add_w       = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h00];
assign inst_sub_w       = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h02];
assign inst_slt         = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h04];
assign inst_sltu        = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h05];
assign inst_nor         = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h08];
assign inst_and         = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h09];
assign inst_or          = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h0a];
assign inst_xor         = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h0b];
assign inst_slli_w      = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] & op_19_15_d[5'h01];
assign inst_srli_w      = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] & op_19_15_d[5'h09];
assign inst_srai_w      = op_31_26_d[6'h00] & op_25_22_d[4'h1] & op_21_20_d[2'h0] & op_19_15_d[5'h11];
assign inst_addi_w      = op_31_26_d[6'h00] & op_25_22_d[4'ha];
assign inst_ld_w        = op_31_26_d[6'h0a] & op_25_22_d[4'h2];
assign inst_st_w        = op_31_26_d[6'h0a] & op_25_22_d[4'h6];
assign inst_jirl        = op_31_26_d[6'h13];
assign inst_b           = op_31_26_d[6'h14];
assign inst_bl          = op_31_26_d[6'h15];
assign inst_beq         = op_31_26_d[6'h16];
assign inst_bne         = op_31_26_d[6'h17];
assign inst_lu12i_w     = op_31_26_d[6'h05] & ~inst[25];
assign inst_pcaddu12i   = op_31_26_d[6'h07] & ~inst[25];
assign inst_mul_w       = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h18];
assign inst_slti        = op_31_26_d[6'h00] & op_25_22_d[4'h8];
```

图 2 译码单元代码展示

如此一来，alu 单元的运算符，立即数单元的选择信号，一些重要控制信号等，都可以用指令信号变量 inst_**的简单或运算来得到，易于代码检查和维护。这些指令信号将随着级间流水寄存器逐步往下传递，告诉每一级正在服务于哪一条指令，以便某些级内控制信号的生成。

Forwarding Unit

Forwarding Unit 管先指令（EX/MEM 级或 MEM/WB 级）的写回寄存器号 wb_dest（rd/1）与后指令（ID/EX 级）的读寄存器号 rf_raddr1（rj）或 rf_raddr2（rk/rd）的冲突。有以下几点问题需要考虑。

几乎所有的指令都含 rd，且几乎所有需要写回的指令的 wb_dest 都为 rd。但并不是所有含 rd 的指令都会写回，如 bne 指令取的是 rj， rd 作操作数进行比较，并不写回，st 指令的 rd 也不写回。所以，除了检查 wb_dest 与 rf_raddr 的冲突，还应该检查 regfile 的写使能以保证确实先指令需要写回。

如果汇编程序不规范，先指令写回永远是 0 的 0 号寄存器，后指令用到了 0 号寄存器作为操作数，那么也无需前递，否则会产生错误。所以，还需检查 wb_dest 是否为 0。

如果 EX/MEM 级与 MEM/WB 级恰巧都想前递数据，那么 EX/MEM 级的优先级应该更高，因为它的指令更新。

在我的代码中，forward1 信号为 32 位位宽的，经过 Forwarding Unit 的选择，最终送向 alu 的数据信号。forward1_sel 为 forward1 的选择信号，forward1_en1 与 forward1_en2 为 forward1 的前递使能信号。关于 forward1 的部分代码实现如下，forward2 同理。

```
//forward1管先指令(EX/MEM级或MEM/WB级)的写回寄存器号wb_dest(rd)与后指令(ID/EX级)的读寄存器号rf_raddr1(rj)的冲突。
//对先指令，需要排除不含rd，或尽管含rd，但不写回rd的指令(如st)
//对后指令，需要排除不含rj的指令
//foward1_en1管EX/MEM的rd与ID/EX的rj的冲突，为1时，有可能发生冲突，产生前递
//foward1_en2管MEM/WB的rd与ID/EX的rj的冲突，为1时，有可能发生冲突，产生前递
assign forward1_en1 = ( !EXMEM_inst_st_w     & !EXMEM_inst_st_h      & !EXMEM_inst_st_b  & !EXMEM_inst_bne       & !EXMEM_inst_beq
                      & !EXMEM_inst_blt      & !EXMEM_inst_bge       & !EXMEM_inst_bltu  & !EXMEM_inst_bgeu      & !EXMEM_inst_b)
                      & (!IDEX_inst_lu12i_w    & !IDEX_inst_pcaddu12i  & !IDEX_inst_b      & !IDEX_inst_bl);

assign forward1_en2 = ( !MEMWB_inst_st_w     & !MEMWB_inst_st_h      & !MEMWB_inst_st_b  & !MEMWB_inst_bne       & !MEMWB_inst_beq
                      & !MEMWB_inst_blt      & !MEMWB_inst_bge       & !MEMWB_inst_bltu  & !MEMWB_inst_bgeu      & !MEMWB_inst_b)
                      & (!IDEX_inst_lu12i_w    & !IDEX_inst_pcaddu12i  & !IDEX_inst_b      & !IDEX_inst_bl);
```

图 3 forward1_en 的生成

```
always @(*) begin
    if ((EXMEM_wb_dest == IDEX_rf_raddr1)      && forward1_en1  && EXMEM_wb_dest)//EXMEM_wb_dest不得为0
        forward1_sel = 2'b01;
    else if((MEMWB_wb_dest == IDEX_rf_raddr1)  && forward1_en2  && EXMEM_wb_dest)//上下两个条件不得调换，if后的条件优先级高，else if后的条件优先级低
        forward1_sel = 2'b10;
    else
        forward1_sel = 2'b00;
```

图 4 forward1_sel 的生成

```
    case (forward1_sel)
        2'b00:
            forward1 = IDEX_rf_rdata1;
        2'b01:
            forward1 = EXMEM_alu_result;
        2'b10:
            forward1 = wb_data;
        default:
            forward1 = IDEX_rf_rdata1;
    endcase
```

图 5 forward1 的选择

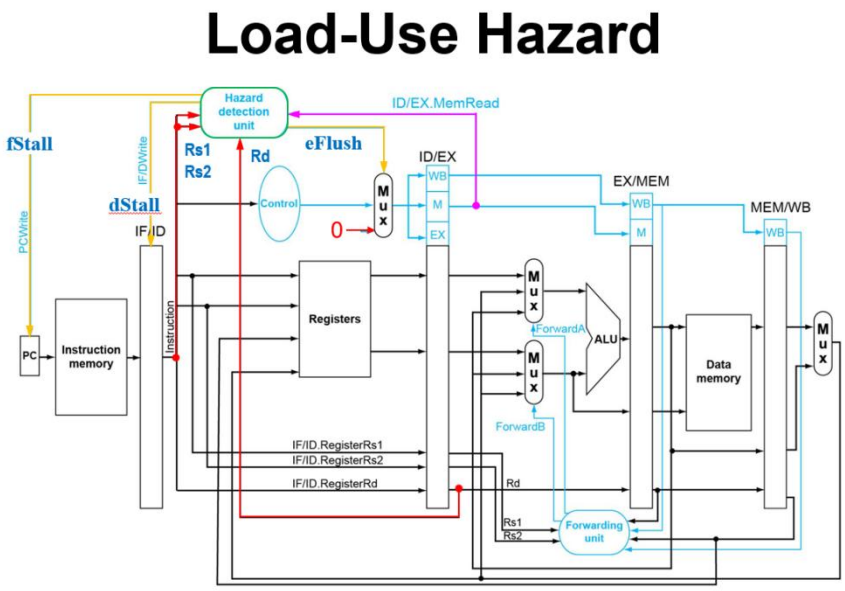
```
//alu_src的选择，此时经过forward unit的筛选，forward信号已经完全替代了rf_rdata1和rf_rdata2
assign alu_src1     = ctr_EX_alu_src1_sel ? IDEX_pc : forward1;
assign alu_src2     = ctr_EX_alu_src2_sel ? IDEX_imm : forward2;
```

图 6 alu_src 的选择

Hazard Detection Unit

Hazard Detection Unit 主要解决单靠 Forwarding Unit 无法解决的问题。尽管 Forwarding Unit 已经尽最大的努力不让流水线暂停，但是遇到某些先后指令的冲突，流水线不得不暂停。主要分为两类 Hazard。

1. Load-Use Hazard

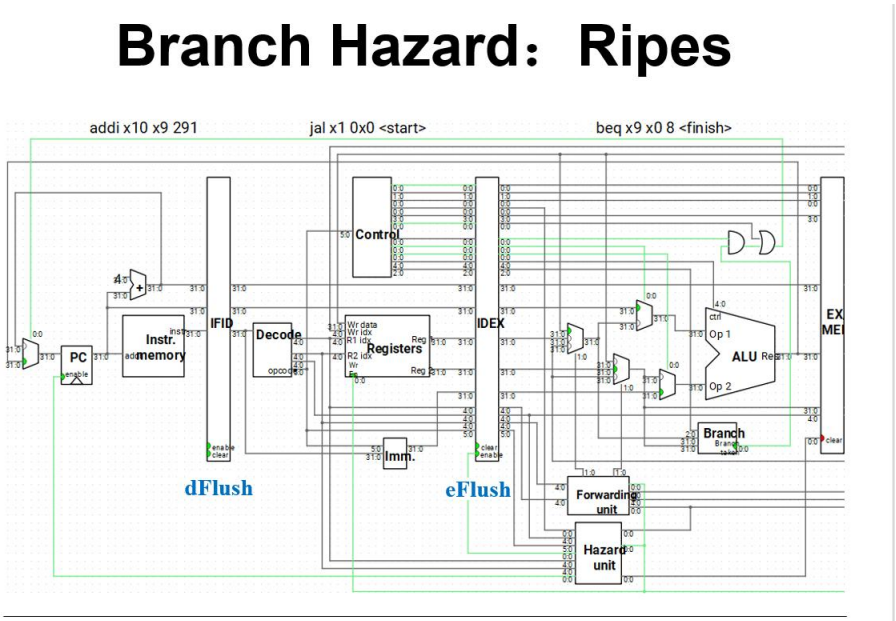


先指令为 ld 类型指令，写回的数据在 MEM/WB 级才给出，若紧接着的指令需要用到 ld 从主存中加载的数据，必须等待一个周期才能获得。

参考 Forward Unit 的写法，负责 Load-Use Hazard 部分的代码如下，给出了对各级间流水寄存器的控制信号 fStall， dStall， eFlush。

```
always @(*) begin
    if(ld_hazard_rj_en && (MEMWB_wb_dest == IDEX_rf_raddr1)
    || ld_hazard_rk_en && (MEMWB_wb_dest == IDEX_rf_raddr2)
    || ld_hazard_rd_en && (MEMWB_wb_dest == IDEX_rf_raddr2))begin
        fStall = 1;
        dStall = 1;
        eFlush = 1;
    end
    else begin
        fStall = 0;
        dStall = 0;
        eFlush = 0;
    end
end
```

2. Branch Hazard



先指令为 b 类型指令，且在 EX 阶段决定跳转，那么在它之后已经有两条指令偷偷跑到了 IF 阶段和 ID 阶段，要把他们清理掉。

若允许跳转，给出 dFlush 和 eFlush 信号。

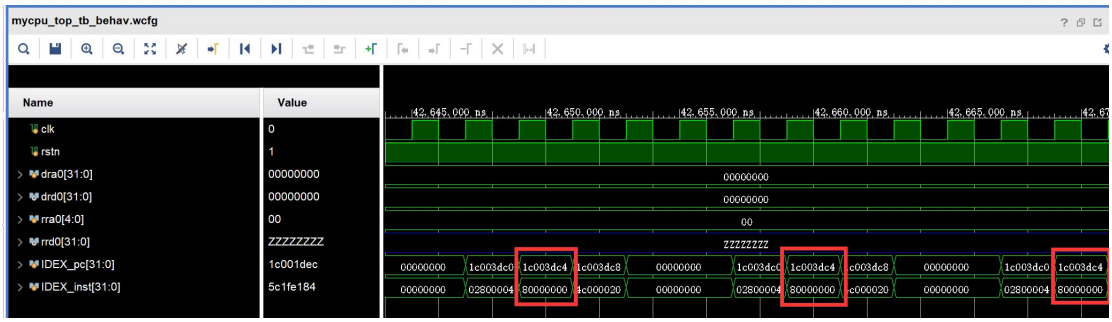
```
assign br_hazard_en = br_en;

always @(*) begin
    if(br_hazard_en)begin
        dFlush = 1;
        eFlush = 1;
    end
    else begin
        dFlush = 0;
        eFlush = 0;
    end
end
```

各级间流水寄存器的写使能 we，清零信号 cl 与上述信号的关系如下。

```
assign pc_we    = !fStall;
assign IFID_we  = !dStall;
assign IFID_cl  = dFlush;
assign IDEX_we  = 1;
assign IDEX_cl  = eFlush;
```

四、仿真结果与分析

主要使用波形图进行 debug。向 InstMem 中初始化 picola32r.coe 中的数据，经过压力测试，观察波形图，pc 成功跳转到成功运行 39 条指令的标志地址 1c003dc4，对应指令码为 80000000。仿真测试通过。



五、电路资源使用情况

经过测试，WNS 非负，电路可以正常运行。电路总共使用了 4920 个 LUT（查找表）、2120 个 LUTRAM（查找表 RAM）和 1134 个 FF（触发器），其中 CPU 模块使用了 3605 个 LUT（查找表）、2120 个 LUTRAM（查找表 RAM）和 514 个 FF（触发器）。总共只比单周期 CPU 多花了约三百个 LUT 和四百个触发器。

六、总结

由于上学期我的单周期 cpu 并未成功上板实现，代码存在诸多问题。在上学期，我也只实现了 5 条指令，所以本次实验中，我花费好几十个小时完成了这个实验从编程到 debug 到仿真通过的全过程。实验的成功，标志着我更加熟练地掌握了 Verilog 的数字系统编程，跟上了课程进度。

这次实验带我们揭开了现代体系结构下的 CPU 的神秘面纱，并为更近一步的优化实现，如缓存、分支预测甚至是多发射铺平了道路，相信这将为我在计算机组成原理上的理论学习带来不少帮助！