浙江大学

计算机体系结构实验报告

课程名称：	计算机体系结构
姓　　名：	周楠
学　　院：	竺可桢学院
专　　业：	计算机科学与技术
学　　号：	3220102535
指导教师：	常瑞

2024 年 10 月 9 日

浙江大学实验报告

课程名称：	计算机体系结构	实验类型：	综合		
实验项目名称：	流水线 RISC-V CPU 设计				
学生姓名：	周楠	专业	计算机科学与技术	学号：	3220102535
实验地点：	玉泉实验室	实验时间：	2024.09.17	指导教师：	常瑞

一. 操作方法与实验步骤

1.1 cmp_32 模块设计

```
assign c = (EQ & res_EQ) |
           (NE & res_NE) |
           (LT & res_LT) |
           (LTU & res_LTU) |
           (GE & res_GE) |
           (GEU & res_GEU); //to fill sth. in ()
```

上述 Verilog 代码主要用于实现一个 32 位的比较器，用于分支（branch）指令的判断。

1.2 CtrlUnit 模块设计

```
wire BEQ = Bop & funct3_0; //to fill sth. in
wire BNE = Bop & funct3_1; //to fill sth. in
wire BLT = Bop & funct3_4; //to fill sth. in
wire BGE = Bop & funct3_5; //to fill sth. in
wire BLTU = Bop & funct3_6; //to fill sth. in
wire BGEU = Bop & funct3_7; //to fill sth. in

wire LB = Lop & funct3_0; //to fill sth. in
wire LH = Lop & funct3_1; //to fill sth. in
wire LW = Lop & funct3_2; //to fill sth. in
wire LBU = Lop & funct3_4; //to fill sth. in
```

```
12    wire LHU = Lop & funct3_5; //to fill sth. in
13
14    wire SB = Sop & funct3_0; //to fill sth. in
15    wire SH = Sop & funct3_1; //to fill sth. in
16    wire SW = Sop & funct3_2; //to fill sth. in
17
18    wire LUI = opcode == 7'b0110111; //to fill sth. in
19    wire AUIPC = opcode == 7'b0010111; //to fill sth. in
20
21    wire JAL = opcode == 7'b1101111; //to fill sth. in
22    assign JALR = ((opcode == 7'b1100111) && funct3_0); //to fill sth
          . in
```

上述是指令类型的判断，通过 opcode 确定指令的类型，再通过 funct3 和 funct7 判断具体指令的类型。

```
1     // 判断跳转的类型
2     assign cmp_ctrl = (BEQ == 1) ? 3'b001 :
3                       (BNE == 1) ? 3'b010 :
4                       (BLT == 1) ? 3'b011 :
5                       (BGE == 1) ? 3'b101 :
6                       (BLTU == 1) ? 3'b100 :
7                       (BGEU == 1) ? 3'b110 : 3'b000;
8
9     assign Branch = JAL | JALR | (B_valid && cmp_res); //to fill sth.
          in
```

cmp_ctrl 信号用于判断 branch 指令的类型，传输到 cmp_32 模块用作控制信号

Branch 信号在 RV32core 模块中用于判断 PC 是否需要跳转。在 mux_IF 模块中，充当选择信号，用于选择 next_PC_IF = PC_4_IF 或者 Jump_PC_ID, 在 hazarddetectionunit 模块中，用于生成 Reg_FD_Flush 信号. 该信号用于在执行 branch 指令时，产生一个 stall 的间隔，便于将计算得到的 Jump_PC_ID 传输到 IF 阶段，用于更新 PC 值

在执行 JAL、JALR、以及 branch 指令满足条件时，才会发生跳转，此时 Branch 信号为 1，否则为 0

```
1    // 判断输入是否在PC
2    assign ALUSrc_A = AUIPC | JAL | JALR; //to fill sth. in
3
4    // I型指令，J型指令，S型指令，L型指令， U型指令 需要使用立即数
5    assign ALUSrc_B = I_valid | S_valid | L_valid | LUI | AUIPC; //to
         fill sth. in
```

ALUSrc_A, ALUSrc_B 表示输入信号的来源。对于 ALUSrc_A 表示是否需要 PC 作为 ALU 的输入，ALUSrc_B 表示是否需要立即数作为 ALU 的输入。

- ALUSrc_A: 对于 branch 指令，跳转后的地址计算发生在 ID 阶段的 ALU，对于 AUIPC、JAL、JALR 指令，跳转后的地址计算发生在 EX 阶段的 ALU，此时 ALUSrc_A = 1，需要 PC 值作为输入进行计算
- ALUSrc_B: 对于 I 型指令，立即数作为 ALU 的输入；对于 S、L、LUI、AUIPC 指令，立即数作为 ALU 的输入，此时 ALUSrc_B = 1，需要立即数作为输入进行计算

```
1    // rs1use 表示是否使用rs1，rs2use 表示是否使用rs2
2    assign rs1use = R_valid | I_valid | L_valid | S_valid | B_valid |
         JALR; //to fill sth. in
3
4    assign rs2use = R_valid | B_valid | S_valid; //to fill sth. in
```

rs1use 和 rs2use 用于判断是否使用 rs1 和 rs2。

```
1    // hazard_optype 表示hazard的类型
2    // 分为ALU计算类型，load，store类型
3    assign hazard_optype = ({2{R_valid | I_valid | JALR | JAL | LUI |
         AUIPC}} & 2'b01) |
4                           ({2{L_valid}} & 2'b10) |
5                           ({2{S_valid}} & 2'b11) ;
6                                           //to fill sth. in
```

hazard_optype 用于判断 hazard 的类型。指示当前指令的类型，是 Load 相关指令，还是 Store 相关指令，还是 ALU 单纯的计算指令

1.3 RV32core 模块设计

```
1    // 根据ID阶段ALU的计算结果，判断是否要跳转
2    // 不跳转 PC_4_IF，跳转 jump_PC_ID， 控制信号 branch_ctrl
3    MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(
         next_PC_IF)); //to fill sth. in ()
```

mux_IF 模块，根据 Branch_ctrl 的值，选择 PC_4_IF 或者 jump_PC_ID 作为下一个 PC 的值。如果需要发生跳转，则选择 jump_PC_ID，否则选择 PC_4_IF。

```
1    // 选择 传入 ID_Reg_EXE的 计算输入
2    MUX4T1_32 mux_forward_A(
3        .I0(rs1_data_reg),
4        .I1(ALUout_EXE),
5        .I2(ALUout_MEM),
6        .I3(Datain_MEM),
7        .s(forward_ctrl_A),
8        .o(rs1_data_ID)
9    ); //to fill sth. in ()
10
11   MUX4T1_32 mux_forward_B(
12       .I0(rs2_data_reg),
13       .I1(ALUout_EXE),
14       .I2(ALUout_MEM),
15       .I3(Datain_MEM),
16       .s(forward_ctrl_B),
17       .o(rs2_data_ID)
18   );
```

mux_forward_A 和 mux_forward_B 用于选择传入 ID_Reg_EXE 的计算输入。根据 forward_ctrl_A 和 forward_ctrl_B 的值，选择 rs1_data_reg、ALUout_EXE、ALUout_MEM、Datain_MEM 作为计算输入，分别对应以下四种情况：

- 不发生数据冲突
- 发生数据冲突，rd_EXE = rs1_ID. rs1 的输入变为 ALU 在 EXE 阶段的计算结果

- 发生数据冲突，rd_MEM = rs1_ID, rs1 的输入变为 ALU 在 MEM 阶段的计算结果
- 发生数据冲突，前一条指令是 load 指令，rs1 的输入变为 load 指令从内存中读取的数据

```
// 选择EX阶段的ALU的输入
MUX2T1_32 mux_A_EXE(
    .I0(rs1_data_EXE),
    .I1(PC_EXE),
    .s(ALUSrc_A_EXE),
    .o(ALUA_EXE)
); //to fill sth. in ()

MUX2T1_32 mux_B_EXE(
    .I0(rs2_data_EXE),
    .I1(Imm_EXE),
    .s(ALUSrc_B_EXE),
    .o(ALUB_EXE)
); //to fill sth. in ()

// Datain_MEM 表示从内存中读出的数据，也就是load的输出
// store是将rs2的值存储到地址rs1+imm的位置
MUX2T1_32 mux_forward_EXE(
    .I0(rs2_data_EXE),
    .I1(Datain_MEM),
    .s(forward_ctrl_ls),
    .o(Dataout_EXE)
);
```

mux_A_EXE 和 mux_B_EXE 用于选择 ALU 的输入。根据 ALUSrc_A_EXE 和 ALUSrc_B_EXE 的值，选择 rs1_data_EXE、Imm_EXE、PC_EXE 作为 ALU 的输入：

针对 load-store 数据冲突类型，由于 store 指令执行时，需要 load 指令读取的数据计算 store 的地址，所以需要根据 forward_ctrl_ls 控制信号，选择 EX 是用立即数作为 ALU 的输入，还是用 load 指令读出的数据作为 ALU 的输入。

1.4 HazardDetectionUnit 模块设计

1.4.1 冲突检测

```
1    // load-use hazard, 需要一个stall, 先排除load_store情况
2    wire rs1_stall = (rs1use_ID == 1)
3                   & (rd_EXE != 0)
4                   & (rs1_ID == rd_EXE)
5                   & (hazard_optype_EX == 2'b10)
6                   & (hazard_optype_ID != 2'b11);
7
8    // forward, rs1的输入变为EXE的结果
9    wire rs1_forward_ctrl1 = (rs1use_ID == 1)
10                          & (rd_EXE != 0)
11                          & (rs1_ID == rd_EXE)
12                          & (hazard_optype_EX == 2'b01);
13   // forward, rs1的输入变为ALU计算的在MEM的值
14   wire rs1_forward_ctrl2 = (rs1use_ID == 1)
15                          & (rd_MEM != 0)
16                          & (rs1_ID == rd_MEM)
17                          & (hazard_optype_MEM == 2'b01);
18
19   // forward, 前一条指令是load,  rs1的输入变为从内存中读出的值
20   wire rs1_forward_ctrl3 = (rs1use_ID == 1)
21                          & (rd_MEM != 0)
22                          & (rs1_ID == rd_MEM)
23                          & (hazard_optype_MEM == 2'b10);
24
25   // load-use hazard, 需要一个stall, 先排除load_store情况
26   wire rs2_stall = (rs2use_ID == 1)
27                  & (rd_EXE != 0)
28                  & (rs2_ID == rd_EXE)
29                  & (hazard_optype_EX == 2'b10)
30                  & (hazard_optype_ID != 2'b11);
31
32   // forward, rs2的输入变为EXE的结果
```

```
33    wire rs2_forward_ctrl1 = (rs2use_ID == 1)
34                           & (rd_EXE != 0)
35                           & (rs2_ID == rd_EXE)
36                           & (hazard_optype_EX == 2'b01);
37    // forward, rs2的输入变为ALU计算的在MEM的值
38    wire rs2_forward_ctrl2 = (rs2use_ID == 1)
39                           & (rd_MEM != 0)
40                           & (rs2_ID == rd_MEM)
41                           & (hazard_optype_MEM == 2'b01);
42    // forward, 前一条指令是load,  rs2的输入变为从内存中读出的值
43    wire rs2_forward_ctrl3 = (rs2use_ID == 1)
44                           & (rd_MEM != 0)
45                           & (rs2_ID == rd_MEM)
46                           & (hazard_optype_MEM == 2'b10);
```

当一条指令在 EXE 阶段从内存中加载数据（即 hazard_optype_EX == 2'b10，表示 load 指令）时，如果下一条指令需要用到这个数据（即寄存器 rs1_ID 或 rs2_ID），会导致数据冒险。因为数据还没有准备好，而下一条指令已经到达执行阶段，这种情况下需要暂停（stall）流水线，直到数据准备就绪。

当 rs1_stall 或 rs2_stall 为 1 时，表明存在 load-use hazard，需要暂停流水线，避免后续指令错误使用尚未准备好的数据。

为了尽量避免因数据冒险导致的流水线暂停（stall），可以通过数据转发（forwarding）机制，将后续指令需要的操作数从 EXE 或 MEM 阶段的计算结果直接转发给当前指令，而无需等待寄存器写回。

- rs1_forward_ctrl1、rs2_forward_ctrl1:

  如果 rs1_ID 或 rs2_ID 是当前 EXE 阶段计算结果的目标寄存器 rd_EXE，并且 EXE 阶段的指令是 ALU 运算（hazard_optype_EX == 2'b01），那么可以直接从 EXE 阶段的结果转发。

- rs1_forward_ctrl2、rs2_forward_ctrl2:

  如果 rs1_ID 或 rs2_ID 是 MEM 阶段的目标寄存器 rd_MEM，并且 MEM 阶段的指令是 ALU 运算（hazard_optype_MEM == 2'b01），那么可以从 MEM 阶段的计算结果转发。

- rs1_forward_ctrl3、rs2_forward_ctrl3:

如果 rs1_ID 或 rs2_ID 是 MEM 阶段的目标寄存器 rd_MEM，并且 MEM 阶段的指令是 load 指令（hazard_optype_MEM == 2'b10），则可以从 MEM 阶段读取的内存数据进行转发。

1.4.2 forward 控制信号

```
1    assign forward_ctrl_A = (rs1_forward_ctrl1) ? 2'b01 :
2                            (rs1_forward_ctrl2) ? 2'b10 :
3                            (rs1_forward_ctrl3) ? 2'b11 : 2'b00;
4
5    assign forward_ctrl_B = (rs2_forward_ctrl1) ? 2'b01 :
6                            (rs2_forward_ctrl2) ? 2'b10 :
7                            (rs2_forward_ctrl3) ? 2'b11 : 2'b00;
8
9    // 判断是否是load_store情况
10   assign forward_ctrl_ls = (hazard_optype_EX == 2'b11)
11                           & (hazard_optype_MEM == 2'b10)
12                           & (rs2_EXE == rd_MEM);
```

- forward_ctrl_A 和 forward_ctrl_B 用于控制数据从 EXE 或 MEM 阶段的转发。通过对源操作数 rs1_ID 和 rs2_ID 的转发控制，可以减少流水线停顿，确保操作数在正确的时间到达执行单元。
- forward_ctrl_ls 用于检测 load-store 数据冒险，当 MEM 阶段是 load 而 EXE 阶段是 store，并且 EXE 阶段的 store 依赖于 load 的结果时，触发这一信号，用于后续处理冲突。

二. 思考题

1. 添加了 Forwarding 机制后，是否观察到了 stall 延迟减少的情况？请在测试程序中给出 Forwarding 机制起到实际作用的位置，并给出仿真图加以证明。

```
1        add x1, x2, x4 # PC = 0xC, x1 = 0x00000018
2        addi x1, x1, -1 # PC = 0x10, x1 = 0x00000017
```

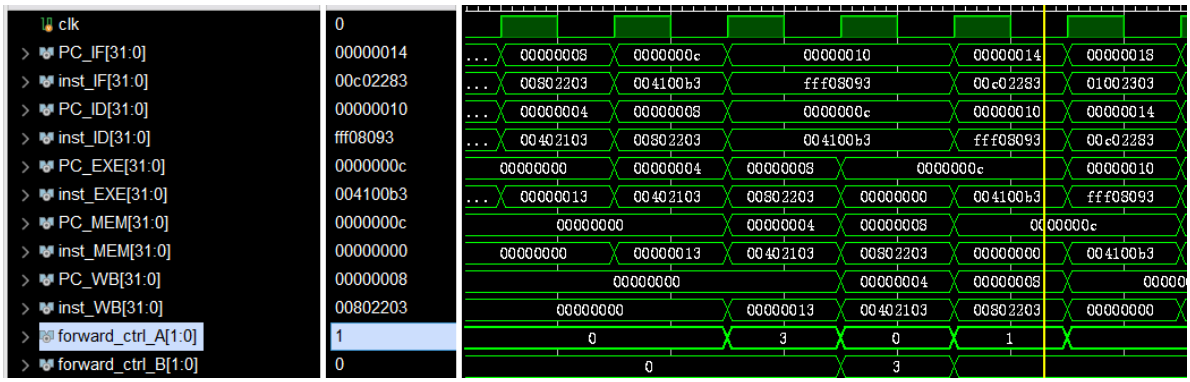

图 1: Forwarding

在上述指令中，add x1, x2, x4 的结果 x1 在 addi x1, x1, -1 中用作输入。如果没有 forwarding 机制, 此时我们需要添加 stall，只有当第一条指令执行到 WB 阶段时，第二条指令才能执行 ID 阶段，中间需要插入一个 stall。

如果存在 forwarding 机制，如上述波形图所示，此时第二条指令不需要 stall，因为此时 forward_ctrl_A 为 1，表示将 ALU 在 EXE 阶段的结果传输到第二条指令的 ID 阶段。

2. 在我们的框架中，比较器 cmp_32 处于 ID 段。请说明比较器在 ID 对比比较器在 EX 的优劣。（提示：可以从时延的角度考虑）

优势：在 ID 阶段就执行比较操作意味着可以提前判断分支条件（如条件跳转或分支指令）。这样，在 ID 阶段即可做出是否跳转的决定，并及时处理分支指令。如果分支成立，可以提前进行分支预测或流水线刷新，避免流水线继续无效地取指和解码错误指令，从而减少因错误预测而产生的分支延迟（Branch delay），这样当分支预测错误时, 我们只需 flush 一个周期, 而 EX 需要 flush 两个周期.

劣势：在典型的流水线中，ID 阶段通常执行寄存器读取和操作码解码等相对简单的任务。加入比较器可能会导致 ID 阶段的时延增加，从而限制流水线的时

钟频率。因为 ID 阶段需要从寄存器中读取操作数，然后执行比较操作，这增加了该阶段的总执行时间。