Implementation of RISC-V 5-Stage Pipeline CPU

Computer Architecture: Lab #2

Due on May 3, 2020 at 23:59pm

魏剑宇
PB17111586

April 30, 2020

Contents

1 实验目的

- **Stage 1**: 实现一个基本的流水线 CPU，包含一些基本的指令
- **Stage 2**: 实现 RISC-V 的大部分用户级指令，并完成流水线的 Hazard 模块，解决数据相关、控制相关等问题
- **Stage 3**: 实现 CSR 模块和 CSR 相关的指令

2 实验环境和工具

- **Host OS**: macOS Mojave 10.14.6
- **Guest OS**: Ubuntu Bionic Beaver
- **VSCode**: 1.44.2
- **Vivado**: HL WebPack Edition
- **Git**: 2.20.1 (Apple Git-117)

本次实验 Simulation 部分在 Ubuntu 虚拟机中完成 (Vivado 没有提供 macOS 版本)，代码编写在 mac 上使用 VSCode 完成.

我的做法是在虚拟机 vivado 创建项目时，将源码的文件夹添加进去的时候，不拷贝，而是映射到项目中。然后在 mac 上通过共享文件夹将源码映射到 ubuntu 上，这样，源码编辑和 Git 版本控制都可以在 mac 上完成，只需在仿真的时候切换到 Ubuntu。整个一套 workflow 还是非常流畅舒适的。

3 实验过程

3.1 Stage 1

原本我是打算跳过 stage 1 直接编写 stage 2，但想到加入 Hazard 模块后，代码会比较复杂，如果在 ALU 和一些简单的数据通路处出现了问题，可能会比较难 Debug，所以还是先做了 Stage 1.

下面在介绍的过程中，我并非描述了我所有编写的代码。例如 BranchDecision 以及 ALU 中的一部分代码，逻辑都十分简单，或者之间有共同性，都介绍的话会比较冗余.

3.1.1 Hazard

为了先消除数据相关的影响，我令代码每运行一条指令，插入三个周期的气泡，完成这一步的代码如下. 插入气泡的方式是令 bubbleF 并 flushD.

```
always @(posedge CPU_CLK) begin
    debug_hazard_cnt <= debug_hazard_cnt + 1;
end

initial begin
    debug_hazard_cnt = 0;
end

assign bubbleF = debug_hazard_cnt[0] | debug_hazard_cnt[1];
assign flushD = bubbleF | debug_flushD;
```

3.1.2 ALU

ALU 部分实现 `Parameters.v` 中几个操作. 这个只需要使用一个 case 语句就可以完成了，唯一需要注意的是 SRA 和 SLT 操作。这两个操作涉及到补码. 在实现 SRA 使，我为了图方便就直接使用 `>>>` 操作符了. BranchDecision 中同样要注意补码的问题.

```
wire signed [31:0] signed_op1 = op1;
always @(*) begin
    case (ALU_func)
        // ...
        `SRA    : ALU_out = signed_op1 >>> op2[4:0];
        // ...
        `SLT    : ALU_out = (op1 < op2)
                          & (op1[31] | ~op2[31])) | (op1[31] & ~op2[31]);
        // ...
    endcase
end
```

3.1.3 ControllerDecoder

这可以说是阶段一整个 CPU 最重要的一部分了。在下面我有选择性地介绍一下其中一些信号的生成。

- **op2_src, alu_src2**: op2 只有对于 R 类型的指令才来源于寄存器，其它情况下都来源于立即数. 对于 alu_src2 由于我在 ALU 计算时进行了截取（因为移位指令要求后一个小于 32），所以没有生成需要 reg2 地址的情况.

  ```
  assign op2_src = imm_type == `RTYPE ? 0 : 1;
  assign alu_src2 = op2_src ? 2'b10 : 2'b00;
  ```

- **load_npc**: 对于 JAL 指令和 JALR 指令，最终 WB 的数应该是 NPC，而不是 ALU 的计算结果.

  ```
  assign load_npc = opcode == `OP_JAL || opcode == `OP_JALR;
  ```

- **ALU_func**: ALU_func 只需要通过一些 if 和 case 语句，针对 opcode 和 funct3 进行分情况讨论就行了. 需要注意，如 AUIPC, Store, Load, JALR 等指令都是需要进行 ADD 操作.
- **cache_write_en**: 只有对于 Store 类指令才不为 0，并且根据类型，其每一个 bit 表示的可写性不同. 例如，对于 sh 指令，则只有低两个字节可写.

```
if (opcode == `OP_STORE)
    case (funct3)
        3'b000 : cache_write_en = 4'b0001;
        3'b001 : cache_write_en = 4'b0011;
        3'b010 : cache_write_en = 4'b1111;
        default: cache_write_en = 4'b0000;
    endcase
else
    cache_write_en = 0;
```

- **imm_type, src_reg_en, reg_write_en**: 这些都由 opcode 唯一决定，并且比较简单，我就放在一起了. 需要注意，src_reg_en 和 reg_write_en 在后面的数据相关中发挥了很重要的作用. 另外，需要特别注意，JAL 和 JALR 的 reg_write_en 也为 1.

```
`OP_JAL    : begin
    imm_type = `JTYPE;
    src_reg_en = 2'b00;
    reg_write_en = 1;
end
`OP_JALR   : begin
    imm_type = `ITYPE;
    src_reg_en = 2'b10;
    reg_write_en = 1;
end
```

3.1.4 DataExtend

框架代码给出了一个不支持非对齐的内存实现，我也就没有考虑这方面的问题. 在 DataExtend 中，需要通过 addr 判断从哪个地方开始读取，另外还需要对读取的数据进行扩展. 如下，是读取一个字节的情况

```
`LB     : begin
    case (addr)
        2'b00: data_before_extend[7:0] = data[7:0];
        2'b01: data_before_extend[7:0] = data[15:8];
        2'b10: data_before_extend[7:0] = data[23:16];
        2'b11: data_before_extend[7:0] = data[31:24];
    endcase
    dealt_data = {{24{data_before_extend[7]}}, data_before_extend[7:0]};
end
```

3.1.5 NPC Generator

在 lab1 中我们讨论过，各个跳转或分支指令是有优先级的，在 EX 段完成跳转的优先级应该高于在 ID 段完成跳转指令的优先级.

```
always @(*) begin
  if (jalr)
      NPC = jalr_target;
  else if (br)
      NPC = br_target;
  else if (jal)
      NPC = jal_target;
  else
      NPC = PC;
end
```

3.2 Stage 2

在阶段一我就已经把另外那些指令也实现了，这里也不讨论了. 在这节我将介绍一下 Hazard 模块的实现.

3.2.1 Data Hazard

阅读了框架代码后，我发现助教的思路和我的思路不同. 框架是对于两条相关的指令，考虑对于前一条指令，如果那条指令与后面的某一条指令相关，就需要插入 bubble. 而我是考虑对于后面一条指令，是否与前面的指令相关，如果是，则插入 bubble.

两种思路都是可以的. 在我的思路下，我需要更改 Hazard 的接口，wb_select 应该来自于 MEM 而不是 EX，并且 reg1_srcD, reg2_srcD, reg_dstE 将不会被用到.

对于 Data Hazard，有三种情况：

1. EX 用到的 reg 来自于 MEM，即 reg2 用到了，不是 0，MEM 有写，并且两者相同. 这种情况又分为两种情况：该条指令是通过 ALU 在 EX 计算得出结果或者在 MEM 加载得到结果. 这两种情况可以通过 wb_select 区分. 如果 EX 就能得到结果，直接 forward 即可，如果 MEM 才能得到结果，就需要插入 bubble.

```
if (src_reg_en[0] && reg2_srcE != 5'b0
    && reg_write_en_MEM && reg2_srcE == reg_dstM) begin
    if (!wb_select) begin
        op2_sel = alu_src2 == 2'b00 ? 2'b00 : 2'b11;
        reg2_sel = 2'b00;
        reg2_bubbleF = 0;
        reg2_bubbleD = 0;
        reg2_bubbleE = 0;
        reg2_flushM = 0;
    end else begin
        op2_sel = alu_src2 == 2'b00 ? 2'b01 : 2'b11;
        reg2_sel = 2'b01;
        reg2_bubbleF = 1;
        reg2_bubbleD = 1;
        reg2_bubbleE = 1;
        reg2_flushM = 1;
    end
end
```

注意，在 EX 段插入 bubble 是通过 bubbleF，bubbleD，bubbleE，flushM 来实现的.

2. EX 用到的 reg 来自于 WB, 此时同样直接转发，不需要 bubble.

```
else if (src_reg_en[0] && reg2_srcE != 5'b0
         && reg_write_en_WB && reg2_srcE == reg_dstW) begin
    op2_sel = alu_src2 == 2'b00 ? 2'b01 : 2'b11;
    reg2_sel = 2'b01;
    reg2_bubbleF = 0;
    reg2_bubbleD = 0;
    reg2_bubbleE = 0;
    reg2_flushM = 0;
end
```

3. 无数据相关，则不需要转发也不需要 bubble.

```
else begin
    op2_sel = 2'b11;
    reg2_sel = 2'b11;
    reg2_bubbleF = 0;
    reg2_bubbleD = 0;
    reg2_bubbleE = 0;
    reg2_flushM = 0;
end
```

reg1 的情况类似.

3.2.2 Control Hazard

对于 EX 段完成的跳转指令，需要冲刷流水线两个周期. 对于 ID 段完成的跳转指令，需要冲刷流水线一个周期. 我直接冲刷流水线，就没必要插入 bubble 了.

```
always @(*) begin
    if (jalr) begin
        jump_flushD = 1;
        jump_flushE = 1;
    end else if (br) begin
        jump_flushD = 1;
        jump_flushE = 1;
    end else if (jal) begin
        jump_flushD = 1;
        jump_flushE = 0;
    end else begin
        jump_flushD = 0;
        jump_flushE = 0;
    end
end
```

可以发现，上面我都是通过 jump, reg1, reg2 来表示由于各个原因引发的 bubble 或 flush. 总的我只需要把他们都或一起就行了，例如:

```
flushM = reg1_flushM | reg2_flushM | rst;
```

3.3 Stage 3

这一步为了不引进更多数据依赖（偷懒省事），我没有按照一般的方法，在 ID 段读取 CSR，在 WB 段写入 CSR，而是直接全部都在 EX 段完成. 这样，这样对 CSR 的读取不需要 bypass 或插入 bubble，只需要对 CSR 用到的通用寄存器进行 bypass. 而这一步不需要更改 Hazard 模块的代码，只需要正确生成 CSR 的 src_reg_en 和 reg_write_en 即可.

3.3.1 CSR

CSR 模块中，我塞进去了一堆寄存器和一个 ALU. 如下:

```
// ...
always @(*) begin
    case (op)
        `CSRRW : dealt_data = in_data;
        `CSRRC : dealt_data = out_data & ~in_data;
        `CSRRS : dealt_data = in_data | out_data;
        default: dealt_data = in_data;
    endcase
end

always @(posedge clk or posedge rst) begin
    if (rst)
        for (i = 0; i < 32; i = i + 1)
            reg_file[i][31:0] <= 32'b0;
    else if (write_en)
        reg_file[dealt_addr] <= dealt_data;
end
```

注意 CSRRC 应当是一个与非操作.

3.3.2 Controller Decoder

另外还需要 CSR 所需要的控制信号.

- **csr_src**: 表示来自 zimm 或者 reg1.

  ```
  assign csr_src = funct3[2];
  ```

  我发现 funct3 的高一位 bit 表示是否是立即数.

- **load_csr**: 和 load_npc 类似，表示 EX 段产生的结果来自 CSR 模块而不是 ALU 或 npc.

  ```
  assign load_csr = opcode == `OP_CSR;
  ```

- **csr_read_en, csr_write_en**: 表示 csr 的可写或者可读. 当然我这里写或者读没有副作用，所以 csr_read_en 也就没有用到.

  ```
  assign csr_write_en = opcode == `OP_CSR
                        && (rs1 != 5'd0 || funct3[1:0] == 2'b01);
  assign csr_read_en = opcode == `OP_CSR && rd != 5'd0;
  ```

- **src_reg_en, csr_write_en**: 需要更新一下这两个控制信号:

  ```
  `OP_CSR    : begin
      imm_type = `ITYPE;
      src_reg_en = csr_src ? 2'b00 : 2'b10;
      reg_write_en = 1;
  end
  ```

4 实验总结

4.1 Bug Fix

在我完成此实验的过程中，coding:debug 的比例大概在 1:2 吧. 这里，我参照着 commit message，尽量回忆一些我踩的坑吧.

- 转发时需要考虑优先级的问题. 如果 MEM 和 WB 段都有产生的结果，需要使用 MEM 而不是 WB 的，因为 MEM 产生的结果是相对 WB 的后一条指令，才是最新结果.
- 对有符号数的移位操作，除了通过数字逻辑实现之外，另一种方式是让 vivado 帮我们做，使用 `>>>` 操作符. 但此操作符要求 op1 是 signed.
- 对于 store 指令，虽然它既用到了 reg1，也用到了 reg2，但对于 reg2，是写入的数据，而不会在 ALU 中用到. 我们需要转发 reg2 而不需要转发 op2.

其它的 bug 是一些不重要的东西，比如 typo 或者意识模糊的时候写的代码，就不再这里讨论了.

4.2 收获 & 分析

这次实验进一步加强（?）了自己的 verilog 能力，当然主要是对流水线有了更深的理解. 像这样实现一次后，很多之前没有搞明白的事情现在就搞清楚了.

这次实验我花了三天时间，每天大概 5 个小时. 第一天 stage1，第二天 stage2，第三天 stage3. 总的来说，在助教搭好的框架上写，还是相对比较容易的. 毕竟只剩下完形填空的活了，也不需要自己设计.