计算机系统结构实验报告 实验六

王梓涵 517021911179

概述

实验名称

简单的类 MIPS 多周期流水化处理器实现

实验目的

- 理解 CPU 的流水线，了解流水线的相关（dependence）和冒险（hazard）
- 设计流水线 CPU，支持停顿（stall），通过检测竞争并插入停顿机制解决数据冒险、控制冒险和结构冒险
- 增加转发（forwarding）机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能

顶层模块

模块描述

实现指令级别的并行执行，需要将流水线分成若干阶段（stage）：取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）。通过插入四级寄存器的方式，可以将前一阶段的相应数据保存给下一阶段使用。主控模块的输出也需要被流水线记录下来，供后续阶段使用。

流水线寄存器

这里首先给出流水线寄存器的定义。对于某些寄存器的某些位，可能在流水线中具有特殊意义，用 `wire` 特别连出，方便各阶段使用。

```
// IF/ID
reg [31:0] IFID_pcPlus4, IFID_instr;
wire [4:0] IFID_INSTRS = IFID_instr[25:21], IFID_INSTRT = IFID_instr[20:16],
IFID_INSTRD = IFID_instr[15:11];

// ID/EX
reg [31:0] IDEX_readData1, IDEX_readData2, IDEX_immSext;
reg [4:0] IDEX_instrRs, IDEX_instrRt, IDEX_instrRd;
reg [8:0] IDEX_ctrl;
wire [1:0] IDEX_ALUOP = IDEX_ctrl[7:6];
wire IDEX_REGDST = IDEX_ctrl[8], IDEX_ALUSRC= IDEX_ctrl[5], IDEX_BRANCH = IDEX_ctrl[4],
IDEX_MEMREAD = IDEX_ctrl[3], IDEX_MEMWRITE = IDEX_ctrl[2], IDEX_REGWRITE = IDEX_ctrl[1],
IDEX_MEMTOREG = IDEX_ctrl[0];

// EX/EM
```

```
reg [31:0] EXMEM_aluRes, EXMEM_writeData;
reg [4:0] EXMEM_dstReg;
reg [4:0] EXMEM_ctrl;
reg EXMEM_zero;
wire EXMEM_BRANCH = EXMEM_ctrl[4], EXMEM_MEMREAD = EXMEM_ctrl[3],
EXMEM_MEMWRITE = EXMEM_ctrl[2], EXMEM_REGWRITE = EXMEM_ctrl[1],
EXMEM_MEMTOREG = EXMEM_ctrl[0];

// MEM/WB
reg [31:0] MEMWB_readData, MEMWB_aluRes;
reg [4:0] MEMWB_dstReg;
reg [1:0] MEMWB_ctrl;
wire MEMWB_REGWRITE = MEMWB_ctrl[1], MEMWB_MEMTOREG = MEMWB_ctrl[0];
```

各阶段实现

由于多条指令需要在流水线的不同阶段并行执行，为了避免冲突，需要对存储器和寄存器的读写进行同步。这里规定，在时钟的下降沿对寄存器组和数据存储器进行写入，在时钟的上升沿对流水线寄存器进行写入。这意味着对于寄存器组和数据存储器而言，在时钟的前半个周期（低电平）时写入，在后半个周期（高电平）读取，先写后读，可以解决结构冒险和一部分的数据冒险。

在各阶段的实现中，其连线不一定和指导书中给出的完全相同，这些主要是为了实现转发而做出的必要修改，之后会做介绍。此外，所有和实验五相同的部分将不再赘述。

取指

这里 PC 也是在是时钟上升沿更新的。由于插入停顿时程序不能向前执行，所以 PC 和 IF/ID 流水线寄存器此时不能更新。同时，为了解决控制冒险，在出现条件跳转时，需要将之前取出的指令清除（flush）。分支和跳转的条件将会在之后介绍。

```
reg [31:0] PC;
wire [31:0] PC_PLUS_4, BRANCH_ADDR, NEXT_PC, IF_INSTR;
wire BRANCH;
assign PC_PLUS_4 = PC + 4;
Mux32 nextPCMux(.in0(PC_PLUS_4), .in1(BRANCH_ADDR), .out(NEXT_PC), .sel(BRANCH));
InstrMemory instrMem(.address(PC), .instr(IF_INSTR));

always @ (posedge clk)
    begin
        if (!STALL)
            begin
                IFID_pcPlus4 <= PC_PLUS_4;
                IFID_instr <= IF_INSTR;
                PC <= NEXT_PC;
            end
        if (BRANCH)
            IFID_instr <= 0; // flush
    end
```

译码

这里的实现中，将跳转的判定和跳转地址的计算移到了译码阶段，只要寄存器读出的数据相等就可以跳转。这是为了减少对流水线寄存器清除的次数，提高执行效率，同时也也有利于简化流水线。

由于插入停顿时，不能对流水线寄存器以外任何模块的状态发生修改，所以此时所有存储在流水线寄存器的控制信号应该置为零。

额外保存了指令的 Rs 位用于转发时的寄存器检测。

```
wire [8:0] CTRL_OUT;
Ctrl mainCtrl(.opCode(IFID_instr[31:26]), .regDst(CTRL_OUT[8]), .aluOp(CTRL_OUT[7:6]),
              .aluSrc(CTRL_OUT[5]), .branch(CTRL_OUT[4]), .memRead(CTRL_OUT[3]),
              .memWrite(CTRL_OUT[2]), .regWrite(CTRL_OUT[1]), .memToReg(CTRL_OUT[0]));

wire [31:0] READ_DATA_1, READ_DATA_2, REG_WRITE_DATA;
Registers regs(.Clk(clk), .readReg1(IFID_INSTRS), .readReg2(IFID_INSTRT),
               .writeReg(MEMWB_dstReg), .writeData(REG_WRITE_DATA),
               .regWrite(MEMWB_REGWRITE), .reset(reset), .readData1(READ_DATA_1),
               .readData2(READ_DATA_2));

wire [31:0] IMM_SEXT;
Signext sext(.in(IFID_instr[15:0]), .out(IMM_SEXT));
wire [31:0] IMM_SEXT_SHIFT = IMM_SEXT << 2;
assign BRANCH_ADDR = IMM_SEXT_SHIFT + IFID_pcPlus4;
assign BRANCH = (READ_DATA_1 == READ_DATA_2) & CTRL_OUT[4];

always @ (posedge clk)
    begin
        IDEX_ctrl <= STALL ? 0 : CTRL_OUT;
        IDEX_readData1 <= READ_DATA_1;
        IDEX_readData2 <= READ_DATA_2;
        IDEX_immSext <= IMM_SEXT;
        IDEX_instrRs <= IFID_INSTRS;
        IDEX_instrRt <= IFID_INSTRT;
        IDEX_instrRd <= IFID_INSTRD;
    end
```

执行

该部分和单周期处理器区别不大。然而由于需要实现转发，ALU 的两个操作数 `ALU_SRC_A` 和 `ALU_SRC_B` 以及待写入内存的数据 `MEM_WRITE_DATA` 都暂时不能确定，其表达式将在之后给出。

```
wire [3:0] ALU_CTRL_OUT;
ALUCtr aluCtr(.funct(IDEX_immSext[5:0]), .aluOp(IDEX_ALUOP), .aluCtrOut(ALU_CTRL_OUT));

wire [31:0] ALU_RES;
wire ZERO;
Alu alu(.in1(ALU_SRC_A), .in2(ALU_SRC_B), .aluCtr(ALU_CTRL_OUT), .zero(ZERO),
        .aluRes(ALU_RES));

wire [4:0] DST_REG;
Mux5 dstRegMux(.in0(IDEX_instrRt), .in1(IDEX_instrRd), .sel(IDEX_REGDST), .out(DST_REG));

always @ (posedge clk)
```

```
    begin
        EXMEM_ctrl <= IDEX_ctrl[4:0];
        EXMEM_zero <= ZERO;
        EXMEM_aluRes <= ALU_RES;
        EXMEM_writeData <= MEM_WRITE_DATA;
        EXMEM_dstReg <= DST_REG;
    end
```

访存和写回

除了更新流水线寄存器之外，和单周期处理器几乎相同，这里直接给出实现：

```
// MEM
wire [31:0] MEM_READ_DATA;
DataMemory dataMem(.Clk(clk), .address(EXMEM_aluRes), .writeData(EXMEM_writeData),
                   .memRead(EXMEM_MEMREAD), .memWrite(EXMEM_MEMWRITE),
                   .readData(MEM_READ_DATA));

always @ (posedge clk)
    begin
        MEMWB_ctrl <= EXMEM_ctrl[1:0];
        MEMWB_readData <= MEM_READ_DATA;
        MEMWB_aluRes <= EXMEM_aluRes;
        MEMWB_dstReg <= EXMEM_dstReg;
    end

// WB
Mux32 writeDataMux(.in1(MEMWB_readData), .in0(MEMWB_aluRes), .sel(MEMWB_MEMTOREG),
                   .out(REG_WRITE_DATA));
```

转发机制

转发条件

转发就是将之前周期的执行阶段或访存阶段得到的数据，直接作为当前执行阶段的操作数之一的机制。就被转发数据的来源而言，既可以来自上一个周期执行阶段的运算结果，也可以来自上一周期访存阶段待写回寄存器的数据。就被转发数的目的地而言，ALU 的两个操作数都可以通过转发机制得到。这使得转发机制的实现变得相对复杂。好在课本《计算机组成与设计（第五版）》（Computer Organization and Design Fifth Edition）中已经给出了相应的逻辑表达式，这里可以直接使用：

```
wire FWD_EX_A = EXMEM_REGWRITE & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_instrRs;
wire FWD_EX_B = EXMEM_REGWRITE & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_instrRt;
wire FWD_MEM_A = MEMWB_REGWRITE & MEMWB_dstReg != 0
    & !(EXMEM_REGWRITE & EXMEM_dstReg != 0 & EXMEM_dstReg != IDEX_instrRs)
    & MEMWB_dstReg == IDEX_instrRs;
wire FWD_MEM_B = MEMWB_REGWRITE & MEMWB_dstReg != 0
    & !(EXMEM_REGWRITE & EXMEM_dstReg != 0 & EXMEM_dstReg != IDEX_instrRt)
    & MEMWB_dstReg == IDEX_instrRt;
```

下面可以得出 ALU 两个源操作数的表达式了，这里直接嵌套了若干条件表达式，这是为了直接使用之前定义的转发信号，不需要额外的编码和译码工作。注意 ALU 的第二个操作数由于还可能接受立即数，所以还需要额外再嵌套一层关于 ALUSrc 的条件表达式。由于立即数不能通过转发得到，所以 ALUSrc 信号的优先级应该高于转发的信号，这样可以避免第二个操作数被错误地换成了其它数据。

```
wire [31:0] ALU_SRC_A =
    FWD_EX_A ? EXMEM_aluRes :
    FWD_MEM_A ? REG_WRITE_DATA :
    IDEX_readData1;
wire [31:0] ALU_SRC_B =
    FWD_EX_B ? EXMEM_aluRes :
    FWD_MEM_B ? REG_WRITE_DATA :
    IDEX_ALUSRC ? IDEX_immSext :
    IDEX_readData2;
```

下面来解决待写入数据存储器的数据 `MEM_WRITE_DATA`。该信号在课本上是直接从 `ALU_SRC_B` 连出来的，但如果考虑立即数就会出现问题。这是因为在访存指令（`lw` 和 `sw`）里立即数表示的是地址的偏移量而不是数据，写入数据存储器的数据不可能来自立即数。如果和 `ALU_SRC_B` 直接连接，就会出现将立即数写入数据存储器的情况。然而存入内存的数据有可能转发自前面的指令的执行结果，所以转发信号依然是有效的。在 `MEM_WRITE_DATA` 的表达式中，只要将 `ALU_SRC_B` 中和立即数相关的部分去除即可。

```
wire [31:0] MEM_WRITE_DATA =
    FWD_EX_B ? EXMEM_aluRes :
    FWD_MEM_B ? REG_WRITE_DATA :
    IDEX_readData2;
```

插入停顿

下面只有一种情况需要插入停顿，即在 `lw` 指令后的一条指令直接访问 `lw` 所加载的寄存器的数据。由于此时数据还在数据存储器内，所以无法通过转发解决数据冒险，只能暂停流水线。这样可以得到插入停顿的条件：

```
wire STALL = IDEX_MEMREAD & (IDEX_instrRt == IFID_INSTRS | IDEX_instrRt == IFID_INSTRT);
```

仿真测试

测试程序

为了测试流水线处理器的正确性，需要设计涉及若干数据相关和控制相关的测试程序。用于本次测试的指令和数据如下所示：

```
lw  $4, 0($0)
LOOP:
lw  $1, 0($0)
lw  $2, 4($0)
add $3, $2, $0
or  $1, $3, $1
add $5, $2, $4
sw  $5, 16($0)
slt $6, $2, $5
beq $2, $3, LOOP
add $1, $1, $4
```

数据寄存器中初始化的值为：

```
0x00-0x03   5
0x04-0x07   7
```

1. 程序最初将数据存储器中地址为 0 的数据（即 5）加载到 4 号寄存器中；
2. 在循环 `LOOP` 中，将 5 加载到 `$1`，将 7 加载到 `$2`；
3. `add` 指令将 `$2` 和 `$0` 相加存入 `$2` 中，该指令依赖上一条指令的 `$2`，`$3` 中应该为 7 而非 0；
4. `or` 指令将 `$1` 和 `$3` 执行或运算存入 `$1` 中，该指令依赖上一条指令的 `$2`，`$1` 中应为 7 而非 0；
5. `add` 指令将 `$2` 和 `$4` 相加得到 12，存入 `$5`；
6. `sw` 指令将 12 存入数据存储器地址 16 的位置，该指令依赖上一条指令的 `$5`，地址 16 的位置存储的应为 12 而非 0 或 16（16 是由于错误地将立即数作为写入数据）；
7. `slt` 指令判断 `$2` 是否小于 `$5` 并存入 `$6`，该指令依赖 `add` 指令中的 `$5`，由于 7 < 12，所以 `$6` 应该为 1 而非 0。
8. `beq` 指令根据 `$2` 和 `$3` 是否相等决定是否跳转到 `LOOP`（即第二条指令），由于 `$2` 和 `$3` 相等，应该跳转；
9. `add` 指令是为了测试处理器能否避免控制冒险，由于 `beq` 一定会跳转，所以该指令不应该执行，`$1` 寄存器中不应该出现值为 12 的情况。

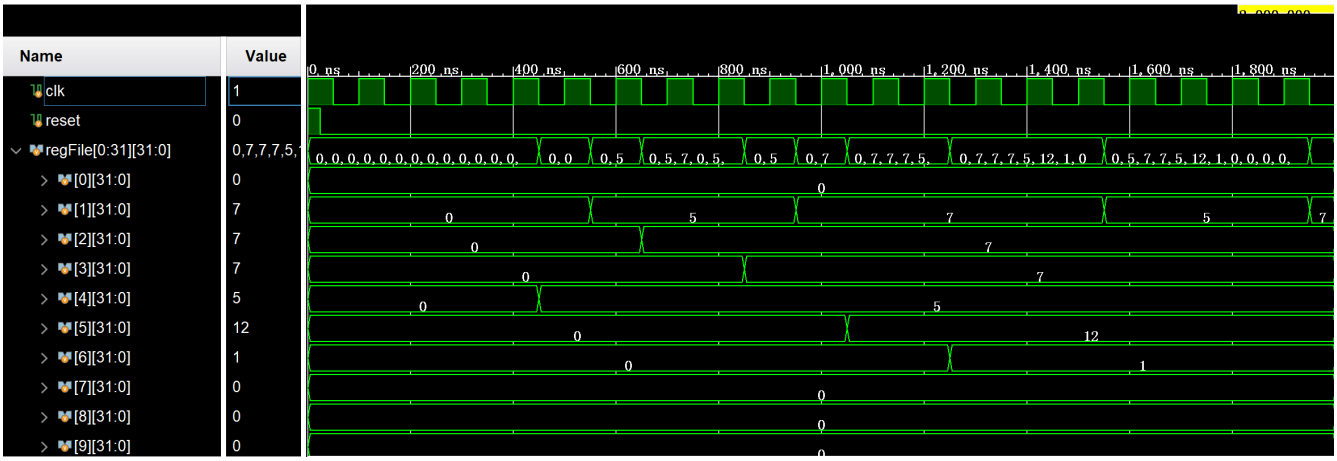
激励信号

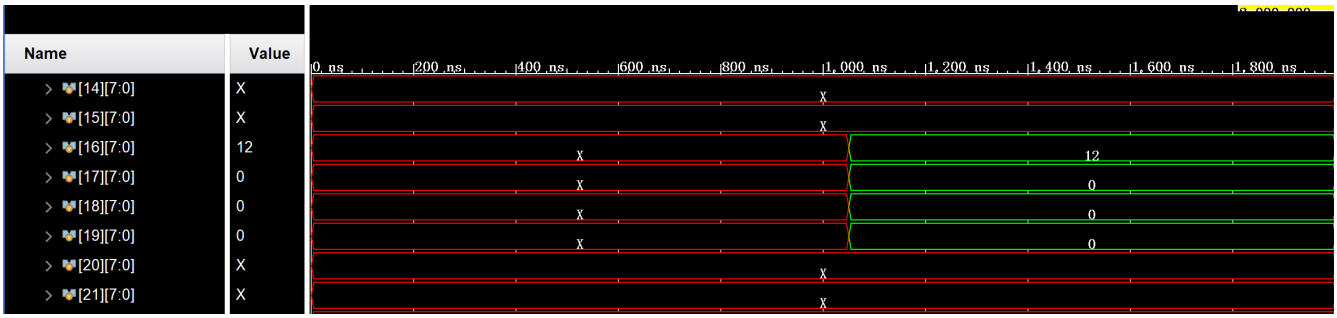
将指令汇编成二进制码，将指令和数据加载到相应寄存器，将时钟周期设为 100 ns，重置信号高电平保持 25 ns。

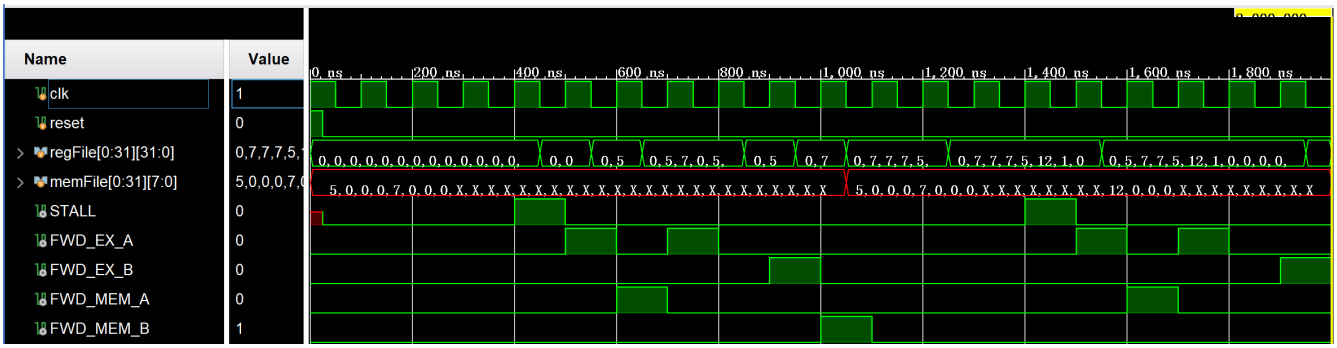
```
module Top_tb();

    reg clk, reset;
    always #50 clk = !clk;

    Top top(.clk(clk), .reset(reset));

    initial begin
        $readmemh("code.txt", top.instrMem.instrFile);
        $readmemh("data.txt", top.dataMem.memFile);
        clk = 1;
        reset = 1;
        #25 reset = 0;
    end
```

```
endmodule
```

仿真波形







上面三张图分别展示了部分寄存器、部分数据存储器单元及转发、停顿信号的波形。可以看到，寄存器和数据存储器的状态均符合预期。

在各循环中，出现一次停顿（`lw` 到 `add`），两次从 EX 阶段到第一个操作数的转发（`add` 到 `or` 及 `add` 到 `sw`），一次从 EX 阶段到第二个操作数的转发（`sw` 到 `slt`，该转发不是必要的，但也没有错误），一次从 MEM 阶段到第一个操作数的转发（`lw` 到 `add`），一次从 MEM 阶段到第二个操作数的转发（`add` 到 `slt`）。

总结与反思

重点与难点

- 在本次实验中由于线路和寄存器数量较多，正确地对这些数据进行命名是非常重要的。在我的实验中，对 `reg` 类型的数据一律用小驼峰命名法（类比变量），`wire` 类型的数据一律用大写加下划线命名（类比常量），对流水线寄存器及其线路，加上流水线名称作为前缀。这样使得代码可读性更高，更不容易出错。

- 由于流水线处理器要实现指令级别并行，同一时间会有若干指令在不同阶段执行，这给查错造成了较大难度。对于这种情况，需要从出错的地方开始，根据课本上的线路图，向前追溯和其相关的信号和数据，逐个检查正确性。这需要对流水线足够的熟悉，并且需要一定的耐心。

待改进之处

- 目前实现的指令数目较少，可以考虑扩展指令，丰富处理器的功能。
- 由于把条件跳转移到了译码阶段，两个待比较操作数的相关性未知。如果该指令存在数据依赖，必须要在汇编时插入停顿，处理器无法处理冒险。可以考虑为该指令也增加转发和停顿机制。

体会

为期六次的体系结构实验到此结束了。在本课程中，我有了不少新的收获，在此作一概述。

本实验课程是对理论课程很好的补充。在理论课中，一些知识学习得比较粗浅，缺乏深入的思考。在实验中，为了把功能正确地实现出来，我需要对课上所学的内容进行巩固和加深。例如处理器的流水化执行这一部分，由于线路复杂，指令之间相关的情况多，学理论课时总有一种没有吃透的感觉。在实现流水线处理器的过程中，我必须将所有这些的连线的条理全部弄清楚。在排错的过程中，更是要一遍遍地理清各阶段的关系，找到错误的源头。这对全面而细致地掌握体系结构时是十分有益处的。

本实验也培养了我坚韧不拔、刻苦钻研的精神。随着本课程的进行，越到后面工程的复杂性就越高。在指导书提供的信息有限的情况下，需要自己额外查阅资料并且摸索。没有足够的耐心、意志力，是完不成这样的工作的。单周期处理器的上板验证部分，我前前后后花了近五个小时，尝试了各种可能方法，才让数码管和 LED 显示正确的内容；流水线处理器的，我也是花了近一天的时间，不断调试，追溯各种错误的根源，才得到了正确的仿真波形。对于以后遇到的种种难题，也应该有这样迎难而上的韧性，才能完成好自己的工作。