计算机系统结构试验 Lab06

518030910022 杨涵章

目录

1 实验概述：

1.1 实验名称

简单的类 **MIPS** 多周期流水线处理器设计与实现

1.2 实验目的

1） 理解 CPU Pipeline，了解流水线冒险(hazard)及相关性，设计基础流水线 CPU
2） 设计支持 Stall 的流水线 CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险、控制竞争和结构冒险
3） 在 2.的基础上，增加 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
4） 在 3. 的基础上，通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能

1.3 实验步骤

1） 按照基本流程创建工程，需导入实验三、四、五中实现的各个模块
2） 对之前实验中实现的模块进行更改，使其能够整体运行
3） 创建顶层模块，对各个模块创建实例，用变量连接起来，组装模块为多周期处理器
4） 增加 Stall 机制的解决流水线 CPU 的数据冒险、控制竞争和结构冒险
5） 增加 Forwarding 机制解决解决数据竞争，减少因数据竞争带来的流水线停顿延时。

2 实验描述：

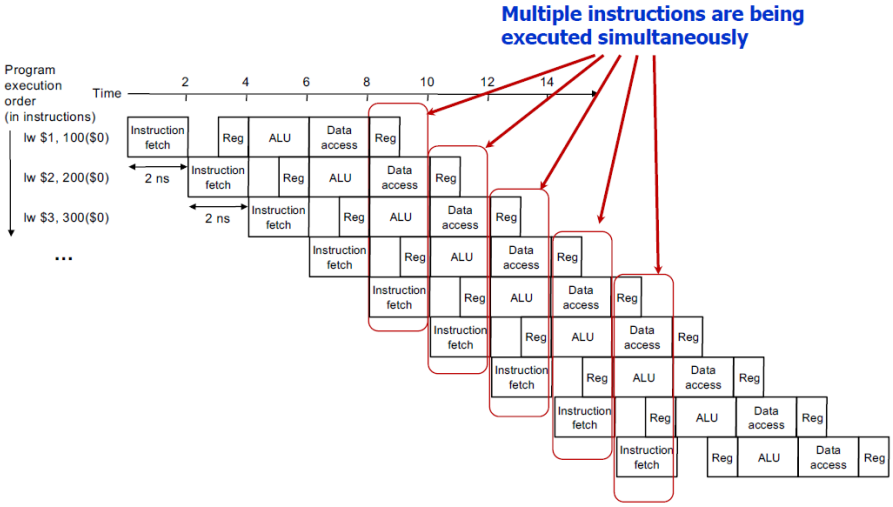
2.1 实验原理

这次实验需要完成指令级别的并行指令执行，因此需要重新实现多周期处理器。需要将流水线分为以下阶段：取值（IF）、译码（ID）、执行（EX）、访存（MEM）、写（WB）。与单周期处理器不同，多周期处理器能够在单个时钟周期内并行处理多个指令的不同阶段。通过四级寄存器的方式，可以将前一阶段的相应数据保存并留给下一阶段使用。主控模块的输出也被流水线记录下来，以供后续阶段使用。本实验将继续使用实验三、四、五中实现的阶段模块，设计顶层模块将其连接起来，组装成多周期处理器。后续增加 Stall 机制、Forwarding 机制解决冲突停顿等问题。

经设计考虑，这次试验无新增模块，但仍需对之前完成的模块进行一定的修改，事之

能够契合流水线结构。

下图为流水线操作的原理，可以看到流水线处理指令能够大大提升处理器的效率。



2.1 修改模块

2.1.1 寄存器模块

在多周期处理器完成之后我对设计的指令进行测试，发现存在部分逻辑赋值无法成功的现象，后经过仔细地排查发现存在时序和逻辑序的混乱。在寄存器模块中对于寄存器数据的读取赋值未采用 assign 赋值，导致数值的变化无法实时变更。等到读取时由于流水线寄存器数值变化较快，数值读取错误。因此这里需要更改为逻辑赋值。

```
35      reg [31:0] regFile[0:31];
36
37      always @ (readReg1 or readReg2)
38      begin
39          if(readReg1)
40              assign readData1=regFile[readReg1];
41          else
42              assign readData1=0;
43          if(readReg2)
44              assign readData2=regFile[readReg2];
45          else
46              assign readData2=0;
47      end
```

2.1.2 其余修改模块

实验过程中我对部分模块进行微调，但并不影响整体思路。因此其余模块无较大改动，存在改动的部分和 Lab05 基本一致。此外我设计的处理器寻址方式与 Lab05 保持一致，指令的读入和数据的储存均按照 32 位进行操作。

2.2 顶层模块

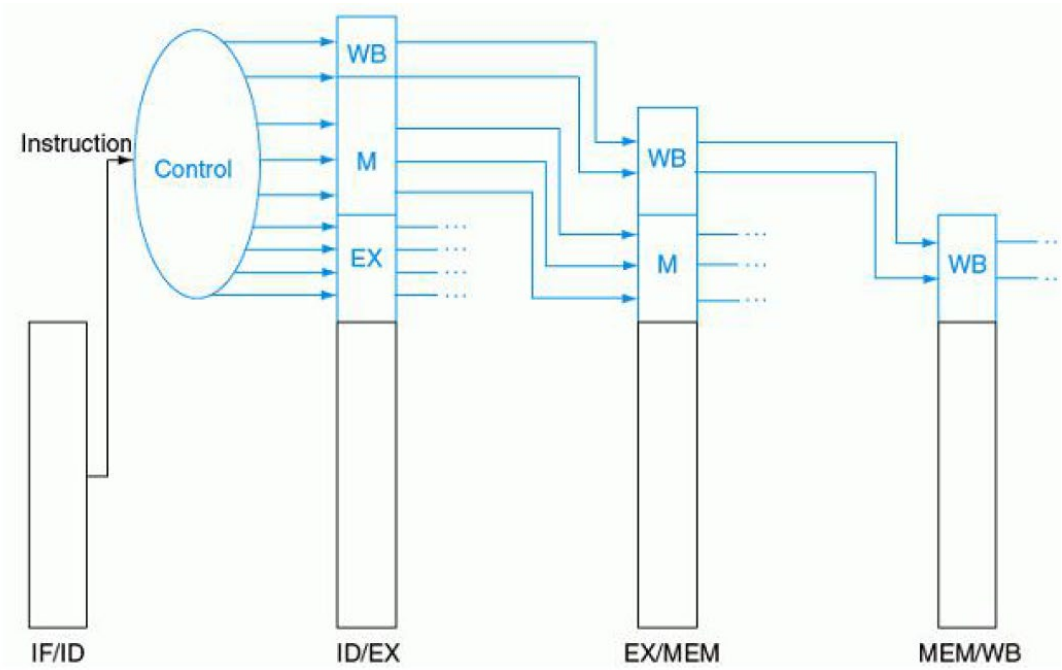
2.2.1 总体描述

由于在单个时钟周期内存在多条指令在流水线的不同阶段并行执行，为了避免寄存器或

存储器的值的读写发生冲突，需要进行时序同步。我的设计思路为在时钟的下降沿进行寄存器或数组存储器的写入，在时钟的上升沿对流水线寄存器进行写入（一定程度上参考计算机系统结构课程中的设计，将单个阶段分位写入读入两部分）。这样可以使得单个时钟周期内先写后读，可以解决结构冒险和一部分的数据冒险。

2.2.1 流水线寄存器

流水线处理器在一级操作中存在多种控制信号，无法公用统一控制信号。因此需要在 ID/EX, EX/EXM, MEM/WB 等阶段储存当前状态的控制信号和中间变量值，在后续连线中需要结合这部分控制信号进行操作数的选择和处理。下图为控制信号储存结构和代码实现。



流水线处理器控制信号储存结构

```
    // IF/ID
    reg [31:0] IFID_pc4, IFID_instr;
    wire [4:0] IFID_instrs = IFID_instr[25:21], IFID_instrt = IFID_instr[20:16],
        IFID_instrd = IFID_instr[15:11];
    wire BRANCH;

    // 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],
    wire 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],
    wire 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];
```

2.2.2 取指（Instruction Fetch）

取值阶段相较于 Lab05 的取值阶段增添了对于停顿（STALL）的判断，供后续插入停顿解决数据冒险的方式使用。在出现条件转移时，会对之前取出的指令进行清除，等待后续操作得出跳转指令结果再进行判断。此外基本和 lab05 相同。下为实现代码。

```
    // Instruction Fetch Stage
    reg [31:0] PC;
    wire [31:0] PC_PLUS_4, BRANCH_addr, NEXT_PC, IF_inst;
    assign PC_PLUS_4 = PC + 1;
    mux32 nextPCmux(.in0(PC_PLUS_4), .in1(BRANCH_addr), .out(NEXT_PC),
        .sel(BRANCH));
    InstMemory instMem(.address(PC), .instruction(IF_inst));

    always @ (posedge clk)
    begin
        if (!STALL)                 //no tall, or PC will be the same as last time cycle
        begin
            IFID_pc4 <= PC_PLUS_4;
            IFID_instr <= IF_inst;
            PC <= NEXT_PC;
        end
        if (BRANCH)
            IFID_instr <= 0;
    end
```

2.2.3 译码（Instruction Decode）

在译码阶段实现了对于跳转的判定和跳转地址的计算，只要寄存器读出的数据相等即可进行跳转。减少了流水线寄存器清楚的次数，进一步提高了执行效率。与 Lab05 相同，在该阶段实现了对于寄存器操作数和立即数的读取，在程序后半段是上部分提及的两阶段之前的控制信号转移。注意该阶段的时序操作。

下为译码阶段的实现代码

```
    // Decode Stage
    wire [8:0] CTRL_out;
    Ctr mainCtrl(.opCode(IFID_instr[31:26]), .regDst(CTRL_out[8]),
        .aluOp(CTRL_out[7:6]), .aluSrc(CTRL_out[5]), .brance(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),
        .regwrute(MEMWB_regwrite), .reset(reset), .readData1(READ_DATA_1),
        .readData2(READ_DATA_2));

    wire [31:0] IMM_sext;
    signext sext(.inst(IFID_instr[15:0]), .data(IMM_sext));
    wire [31:0] IMM_sext_SHIFT = IMM_sext << 2;
    assign BRANCH_addr = IMM_sext_SHIFT + IFID_pc4;
    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
```

2.2.4 执行（Execution）

执行部分和单周期处理方式大致相同，实例化 ALU 控制器、ALU。同样代码最后也需要添加时钟上升沿对流水线寄存器的操作。在执行阶段对于源操作数存在选择，主要为转发机制（forwarding）的选择，需要根据转发信号进行判断。转发机制将在后续部分进行介绍。

```
    // Execution Stage
    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 = IDEX_ALUSRC ? IDEX_immSext : FWD_EX_B ? EXMEM_aluRes :
        FWD_EX_B ? EXMEM_aluRes : FWD_MEM_B ? REG_WRITE_DATA : IDEX_readData2;
    wire [31:0] MEM_WRITE_DATA = FWD_EX_B ? EXMEM_aluRes :
        FWD_EX_B ? EXMEM_aluRes : FWD_MEM_B ? REG_WRITE_DATA : IDEX_readData2;

    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(.input1(ALU_SRC_A), .input2(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
```

2.2.5 访存（Memory）

该部分主要进行对于内存的读写，与 lab05 基本相同，但同样需要注意需要继续更新流水线寄存器。

```
    // Memory Stage
    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
```

2.2.6 写回（Write Back）

对寄存器进行更新，同样和单周期处理器基本相同。这一步为指令周期最后一个阶段，无需进行流水线处理器的操作。

```
    // Write Back Stage
    mux32 writeDatamux(.in1(MEMWB_readData), .in0(MEMWB_aluRes),
        .sel(MEMWB_memtoreg), .out(REG_WRITE_DATA));
```

2.2.7 reset

和单周期处理器类似，顶层模块需要接收 reset 信号对处理器中所有连线和寄存器进行初始化。将所有信号或寄存器置零即可。

```
    always @ (reset)
    begin
        PC = 0;
        IFID_pc4 = 0;
        IFID_instr = 0;
        IDEX_instrRs = 0;
        IDEX_readData1 = 0;
        IDEX_readData2 = 0;
        IDEX_immSext = 0;
        IDEX_instrRt = 0;
        IDEX_instrRd = 0;
        IDEX_ctrl = 0;
        EXMEM_aluRes = 0;
        EXMEM_writeData = 0;
        EXMEM_dstReg = 0;
        EXMEM_ctrl = 0;
        EXMEM_zero = 0;
        MEMWB_readData = 0;
        MEMWB_aluRes = 0;
        MEMWB_dstReg = 0;
        MEMWB_ctrl = 0;
    end
```

2.3 冲突处理

2.3.2 转发机制（Forwarding）

转发即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。转发数据有两种来源：1. 来自上一周期执行阶段的运算结果；2. 来自上一周期访存。同样的转发数据也可以分别去 ALU 的两个操作数。转发机制原理较为简单，但具体实现较为复杂，这里我参考了《Computer Organization and Design Fifth Edition》中的逻辑表达式（下图为相关转发和信号和解释），将其套用到我的代码中去。

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

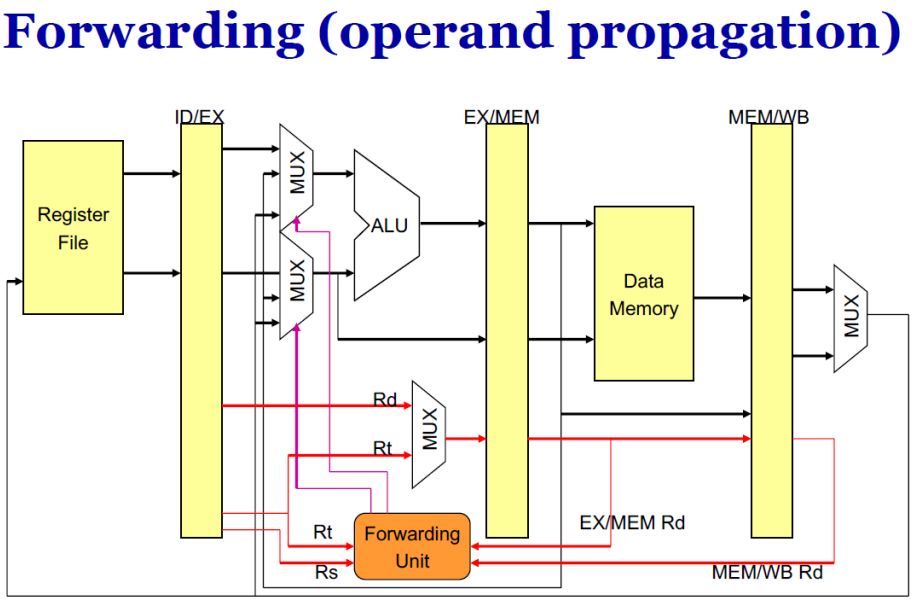
FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

代码如下（直接放置于顶层模块中，未将其书写成独立的模块）：

```
    // Forwarding Unit
    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;
```

同时由于转发机制的加入，之前顶层代码的连接也需要改变：ALU 源操作数的选取。线路的连接主要参照下方原理图（图源自计算机系统与结构课程班 PPT），并结合自己的代码

做出了细微调整。



下面是 ALU 两个源操作数的表达式，这里嵌套了若干表达式。这里的嵌套逻辑为：1. 优先考虑寄存器转发；2. 其次考虑存储器转发（从逻辑来看寄存器的值在储存之后发生，当前指令的操作数应当为寄存器最后的值，所以寄存器转发优先）；3. 无转发发生直接为寄存器读取所得数据。

涉及到转发操作的还有写入数据存储器的数据。该信号在课本上直接从 B 源操作数接出，但一旦考虑立即数操作便可能出现问题。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，写入寄存器的数据不可能来自立即数。如果连接便会发生将立即数写入存储器的情况。此时转发信号依然有效，所以我抛弃了立即数操作，在源操作数中和立即数有关的操作去掉。

转发操作的代码如下：

```
122     // Execution Stage
123     wire [31:0] ALU_SRC_A = FWD_EX_A ? EXMEM_aluRes :
124         FWD_MEM_A ? REG_WRITE_DATA : IDEX_readData1;
125     wire [31:0] ALU_SRC_B = IDEX_ALUSRC ? IDEX_immSext : FWD_EX_B ? EXMEM_aluRes :
126         FWD_EX_B ? EXMEM_aluRes : FWD_MEM_B ? REG_WRITE_DATA : IDEX_readData2;
127     wire [31:0] MEM_WRITE_DATA = FWD_EX_B ? EXMEM_aluRes :
128         FWD_EX_B ? EXMEM_aluRes : FWD_MEM_B ? REG_WRITE_DATA : IDEX_readData2;
```

2.3.1 插入停顿（STALL）

由于数据冲突，存在两种数据冒险：1. 当前指令的操作数在上一指令的执行阶段得出；2. 当前指令的操作数在上一指令的访存阶段得出。因此在当前阶段无法得出正确的操作数，需要当前指令暂停，直到上一指令完成操作才能获得正确的操作数。

由于访存机制的存在，两相邻指令之间的数据冒险能够被直接消除。但第二种冒险由于两阶段相隔较远，需要两次停顿才能消除冒险。经过转发机制，这里仅需要一次停顿。停顿的实现较为简单，这里只需要判断第二种停顿即可。

其次就是在之前提到的停顿的处理方式，如果遇到停顿（STALL=true），需要 PC 当前周期停止更新，控制信号置零。直到下一周期停顿结束才开始执行。

代码实现如下。

```
56      // Hazard detection
57      wire STALL = IDEX_MEMREAD & (IDEX_instrRt == IFID_instrs | IDEX_instrRt == IFID_instrt);
```

```
99      always @ (posedge clk)
100     begin
101         IDEX_ctrl <= STALL ? 0 : CTRL_out;
102         IDEX_readData1 <= READ_DATA_1;
103         IDEX_readData2 <= READ_DATA_2;
104         IDEX_immSext <= IMM_sext;
105         IDEX_instrRs <= IFID_instrs;
106         IDEX_instrRt <= IFID_instrt;
107         IDEX_instrRd <= IFID_instrd;
108     end
```

3 模拟仿真：

3.1 软件方法

3.1.1 MIPS 指令编写

为了方便检测验证，这次的 MIPS 指令我主要参考了实验手册提供的代码。代码主要操作为在指令中已有较为详细的注释，便不再描述。指令中可以看到 3 和 5、2 和 4 存在数据冒险。因此可以采用调换指令执行顺序的方法，将后续无关指令提前执行而不影响执行结果，调换方法已在图中标记。调换之后存在数据冲突的数据有足够的时间完成指令的执行，从而消除了数据冒险。

指令代码和数据文件如下：

```
1   lw  $1, 0($0)    ;1
2   lw  $2, 4($0)    ;5
3   lw  $3, 8($0)    ;8
4   add $4, $1, $2   ;$4=6
5   add $5, $3, $1   ;$5=7
6   add $6, $2, $1   ;$6=1
7   lw  $10, 0($0)   ;1
8   lw  $10, 0($0)   ;1
9   lw  $10, 0($0)   ;1
10  or  $7, $3, $1   ;$6=1
11  slt $8, $3, $1   ;$6=1
12  beq $0, $0, end ;to end
13  add $9, $7, $8   ;$9=9, not executed
14  end:
15  lw  $10, 0($0)   ;1
16  lw  $10, 0($0)   ;1
17  lw  $10, 0($0)   ;1
18  lw  $10, 0($0)   ;1
19  lw  $10, 0($0)   ;1
```

```
1   1
2   5
3   8
```

```
10001100000000001000000000000000
10001100000000010000000000000001
10001100000000011000000000000010
00000000001000100010000000100000
00000000011000010010100000100010
00000000010000010011000000100100
10001100000010100000000000000000
10001100000010100000000000000000
10001100000010100000000000000000
00000000011000010011100000100101
00000000011000010100000000101010
00010000000000000000000000000001
00000000111010000100100000100000
10001100000010100000000000000000
10001100000010100000000000000000
10001100000010100000000000000000
10001100000010100000000000000000
10001100000010100000000000000000
```

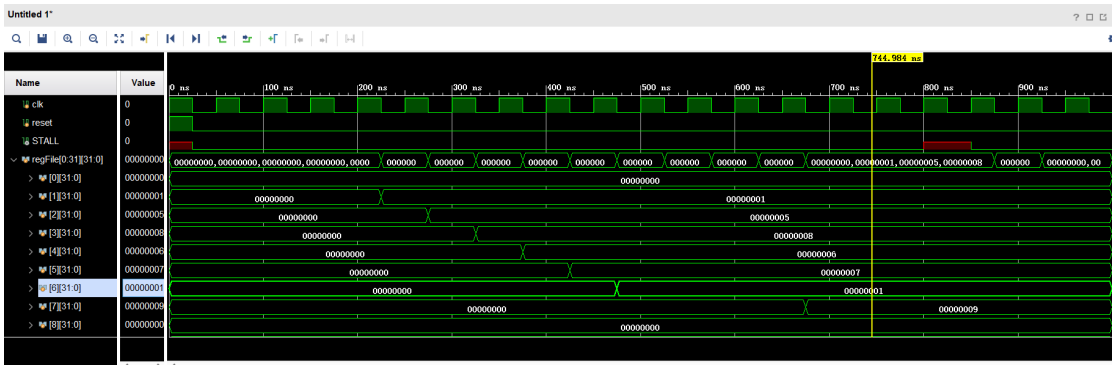
3.1.2 仿真程序

仿真程序中设置时钟信号，读取相对路径下的文件（这里我已经将文件放置在相对路径下）。设置 reset 信号，模拟运行即可。由于指令较长，我设置时钟为 25 单位变化一次。

```
module top_tb(

    );

        reg clk, reset;
    always #25 clk = !clk;

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

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

3.1.3 仿真结果

由于我将停顿（STALL）操作和多周期处理器的实现合并到一起，所以这里需要从 stall 来进行判断是否存在数据冲突。可以看到哦各个指令执行的情况正确，而且指令的执行过程中未发生停顿（中途 STALL 值无效原因不详，但我详细查看过中间变量，发现当时指令仍正常运行，无停顿）。因此通过交换指令顺序，我成功消除了数据冒险。



3.2 插入停顿（STALL）

3.2.1 MIPS 指令编写

指令整体与第一部分的指令相同。但我去除了用于消去冲突的加载指令（指令 7、8、9），同时将指令 4 与 5 交换顺序。这样便会出现 load-use 冲突。设计这样的指令预期可以观测到仿真波形出现 STALL，并且指令运行周期延缓，但同时保证指令能够正确执行。

指令如下，由于变动较小，这里便不再展示二进制代码和数据文件。

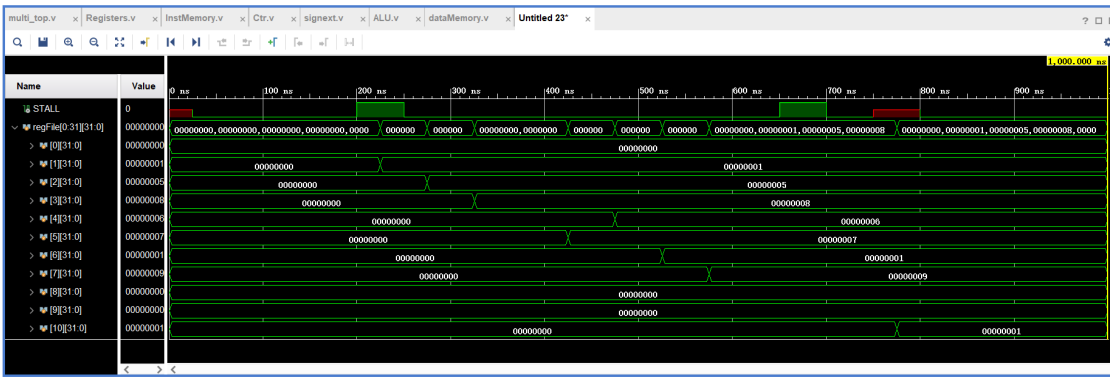
```
lw  $1, 0($0)    ;1
lw  $2, 4($0)    ;5
lw  $3, 8($0)    ;8
add $5, $3, $1   ;$5=7
add $4, $1, $2   ;$4=6
add $6, $2, $1   ;$6=1
or  $7, $3, $1   ;$6=1
slt $8, $3, $1   ;$6=1
beq $0, $0, end ;to end
add $9, $7, $8   ;$9=9, not executed
end:
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
```

3.2.2 仿真程序

仿真程序同 3.1.2.。

3.2.3 仿真结果

首先可以看到程序执行的正确性，寄存器文件的数值变化和之前正确的波形相同。同样也可以和实验手册中的波形作对比。这里可以看到 STALL 的数量为 1（若只实现 STALL），这是因为我将转发机制也加入到其中，能够消去一层 STALL，但仍然保留一层 STALL 无法消除。这里可以观察到 STALL 机制能够正确运作，使得指令暂停运行一周期，同时保证了指令运行的正确性。



3.3 转发（Forwarding）

3.3.1 MIPS 指令编写

指令整体与第一部分的指令基本相同。但我去除了用于消去冲突的加载指令（指令 7、8、9）。这样便会出现上一指令仍处于冲突。设计这样的指令预期可以观测到未出现 STALL，同时保证指令能够正确执行。

同样指令如下，由于变动较小，这里便不再展示二进制代码和数据文件。

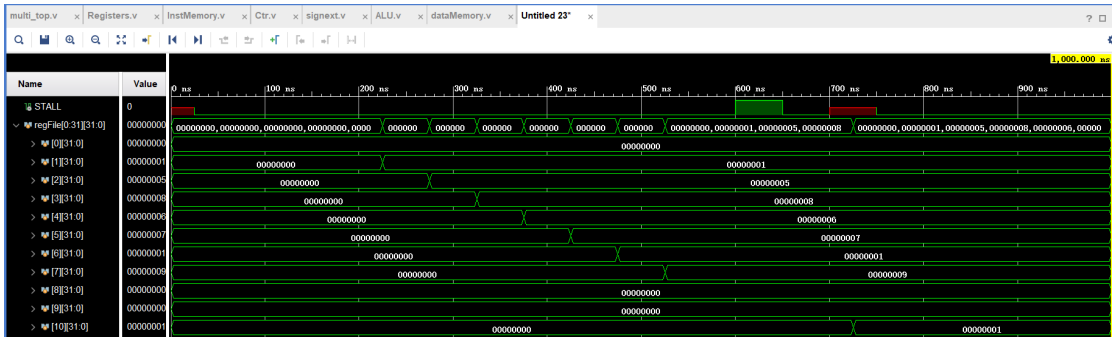
```
lw  $1, 0($0)    ;1
lw  $2, 4($0)    ;5
lw  $3, 8($0)    ;8
add $4, $1, $2   ;$4=6
add $5, $3, $1   ;$5=7
add $6, $2, $1   ;$6=1
or  $7, $3, $1   ;$6=1
slt $8, $3, $1   ;$6=1
beq $0, $0, end ;to end
add $9, $7, $8   ;$9=9, not executed
end:
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
lw  $10, 0($0)   ;1
```

3.3.2 仿真程序

仿真程序同 3.1.2.。

3.3.3 仿真结果

首先可以看到程序执行的正确性，寄存器文件的数值变化和之前正确的波形相同。同样也可以和实验手册中的波形作对比。这里可以看到冲突部分指令（指令 2、4）无 STALL 产生，表示转发机制正确执行。得到了正确的波形。



4 总结与反思：

本实验主要难点包括各部分的连接、流水线状态的储存、程序的调试。这次实验的线路相较于上一次实验增加了许多，因此连线方面十分复杂。我处理的基本思路还是按照流水线阶段处理连线。由于要求实现的流水线多周期处理器要求对流水线信号进行储存，进一步加大连线复杂程度。因此我选择先实现多周期处理器，将需要流水线信号的部分进行空出，后依次进行补充。由于线路较多，命名较为复杂，我基本采用“阶段名__操作“的方式进行命名，但仍然耗费了较长时间。最后便是程序的调试，流水线信号较多，对于每个阶段需要同时观察多个阶段的指令，依次检查程序的运行情况，程序调试难度较大。

这次设计的流水线处理器仍有许多值得改进的地方。首先便是 Lab05 中同样存在的寻址大小的问题，由于时间紧迫，这次的实验大多是在 Lab05 的基础中进行填充，因此我并未进行寻址大小改变的尝试。其次便是后续预测、指令扩张等部分我并没有实现。由于期末时间的限制，无法完整地实现多周期流水线处理器，较为遗憾。

本次实验是这次课程的最后一次实验，也是最为复杂、难度最大的一次实验。就我个人而言，在老师预设的时间之内我难以完整地完成该实验，因此我的实验并没有实现预测及以后的内容。再加上最后一个实验的周期和考试周的周期相重叠，所以时间便更加紧凑。因此实验不完整，我对此感到抱歉。其次便是我希望老师在以后的授课过程中能够对实验进行一定的总揽讲解。虽然我能感受得到老师想要培养同学自主学习探索的能力，但我也在学习的过程中犯了许多不必要的错误，耽搁了较多的时间。我相信能有较为清晰的指导能够使我更早地完成实验。

最后特别感谢老师对于实验的指导，使得我有机会将课程中学习的知识融入到具体操作中去。但仍感到稍微遗憾的是未能进行相关上板测试，只能在线进行模拟仿真实验，也希望以后能够有机会真正上板测试。