计算机系统结构实验 Lab6

David Wang

2020 年 5 月

1 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 流水线，了解流水线冒险及相关性，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的机制解决数据冒险、控制冒险、结构冒险
3. 在2的基础上，增加转发机制解决数据竞争，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 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

1.3 实验内容

1. CPU 的流水化设计与软硬件实现
2. 功能仿真

2 整体概括

2.1 实验描述

前几次实验已完成单周期处理器各部分的主要功能模块。而流水线处理器在顶层模块、控制器模块等与单周期处理器有微小差别。除此之外，需要设计流水线寄存器，转发机制以及冲突检测机制。

2.2 顶层模块概述

实现流水线执行，需要将一条指令分为取指令 (IF)、译码 (ID)、执行 (EX)、内存访问 (MEM)、写回 (WB) 五个阶段。同时，在两个阶段中间需要设置流水线寄存器，用来存储前一个阶段的执行所产生的信息。从图中可以看出，流水线的执行步骤大致为：

- 取指阶段，CPU 根据 PC 给出的地址，访问指令存储器，将指令取出后存放在 IF/ID 寄存器中，计算 PC+4 的值，供之后的几个阶段使用。

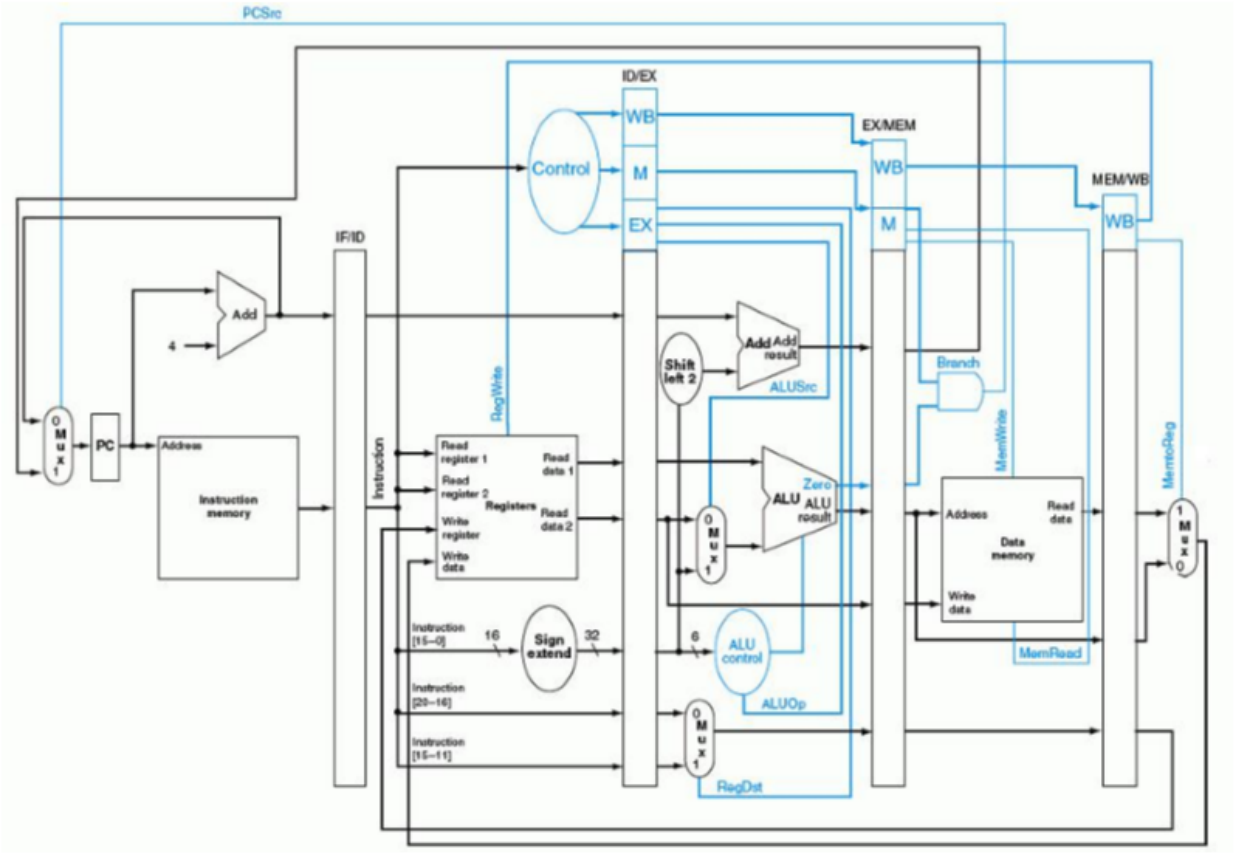
PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
M u x
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

图 1: 流水线示意图

- 译码阶段，根据 IF/ID 寄存器中的指令，依据 rs、rt 访问寄存器，将指令输入控制单元进行相关解析，决定该指令是否写寄存器，读写内存，跳转，等等。对于指令的低 16 位进行符号拓展，作为立即数供执行阶段选用。需要注意的是控制单元将后面三个阶段所需要的控制信息一并解析出来并存放到流水线寄存器中，随后逐级使用。
- 执行阶段，根据 ID/EX 寄存器中的控制信息，决定算术逻辑单元两个操作数的来源。如果有转发单元参与，则也需要考虑是否转发。计算跳转地址，供之后可能发生的跳转操作使用。
- 内存访问阶段，如果该指令需要内存访问，则需要向数据存储器提供地址，并保存数据存储器给出的数据。如果该指令为跳转指令 (beq、jr、jal 等)，则需要将跳转目标地址写入 PC，并且刷新流水线，抛弃随后不应该执行的几条指令。
- 写回阶段，需要根据控制信号确定是否向寄存器写值，写值的来源是算术逻辑单元的运算结果还是内存访问得到的数据。

3 流水线寄存器模块

下面给出流水线寄存器的实现。

```
    module IFID(
    input clock,
    input reset,
    input [31:0] instruction_in,
    input [31:0] PC_Plus_4,
    output reg [31:0] IFID_PCplus4,
    output reg [31:0] IFID_instruction
    );
    always @ (posedge clock)
    if(!reset)
    begin
        IFID_PCplus4<=PC_Plus_4;
        IFID_instruction<=instruction_in;
    end
    else
    begin
        IFID_PCplus4=0;
        IFID_instruction=0;
    end
endmodule

module IDEX(
    input clock,
    input reset,
    input [31:0] IFID_PCPlus4,
    input [4:0] IFID_rs,
    input [31:0] readData1,
    input [31:0] readData2,
    input [31:0] signextended,
    input [4:0] IFID_rt,
    input [4:0] IFID_rd,
    input [1:0] wb,//RegWrite
```

```
    input [4:0] m,// Jumptarget,Jump,Branch,memRead,memWrite
    input [4:0] ex,// D4:ALUSrc D3-D1:ALUOp D0:RegDst
    input call,
    output reg [31:0] IDEX_PCPlus4,
    output reg [31:0] IDEX_readData1,
    output reg [31:0] IDEX_readData2,
    output reg [31:0] IDEX_signextended,
    output reg [4:0] IDEX_rs,
    output reg [4:0] IDEX_rt,
    output reg [4:0] IDEX_rd,
    output reg [1:0] IDEX_wb,
    output reg [4:0] IDEX_m,
    output reg [4:0] IDEX_ex,
    output reg IDEX_call
    );
    always @ (posedge clock)
    if(!reset)
        begin
        IDEX_PCPlus4<=IFID_PCPlus4;
        IDEX_readData1<=readData1;
        IDEX_readData2<=readData2;
        IDEX_signextended<=signextended;
        IDEX_rs<=IFID_rs;
        IDEX_rt<=IFID_rt;
        IDEX_rd<=IFID_rd;
        IDEX_m<=m;
        IDEX_wb<=wb;
        IDEX_ex<=ex;
        IDEX_call<=call;
        end
    else
        begin
        IDEX_PCPlus4<=0;
        IDEX_readData1<=0;
        IDEX_readData2<=0;
        IDEX_signextended<=0;
        IDEX_rt<=0;
        IDEX_rd<=0;
        IDEX_m<=0;
        IDEX_wb<=0;
        IDEX_ex<=0;
        IDEX_call=0;
        end
endmodule

module EXMEM(
    input clock,
    input reset,
    input [1:0] IDEX_wb,
    input [4:0] IDEX_m,
```

```
    input IDEX_call,
    input [31:0] IDEX_PCPlus4,
    input [31:0] EX_branchTarget,
    input [31:0] EX_jumpTargetAddress,
    input [31:0] EX_aluResult,
    input EX_zero,
    input [31:0] EX_memWriteData,
    input [4:0] EX_writeReg,

    output reg [1:0] EXMEM_wb,//RegWrite
    output reg [4:0] EXMEM_m,
    output reg [31:0] EXMEM_branchTarget,
    output reg [31:0] EXMEM_jumpTargetAddress,
    output reg EXMEM_zero,
    output reg [31:0] EXMEM_aluResult,
    output reg [31:0] EXMEM_memWriteData,
    output reg [4:0] EXMEM_writeReg,
    output reg [31:0] EXMEM_PCPlus4,
    output reg EXMEM_call
);
    always @ (posedge clock)
    if(!reset)
    begin
        EXMEM_wb<=IDEX_wb;
        EXMEM_m<=IDEX_m;
        EXMEM_aluResult<=EX_aluResult;
        EXMEM_branchTarget<=EX_branchTarget;
        EXMEM_jumpTargetAddress<=EX_jumpTargetAddress;
        EXMEM_zero<=EX_zero;
        EXMEM_memWriteData<=EX_memWriteData;
        EXMEM_writeReg<=EX_writeReg;
        EXMEM_PCPlus4<=IDEX_PCPlus4;
        EXMEM_call<=IDEX_call;
    end
    else
    begin
        EXMEM_wb<=0;
        EXMEM_m<=0;
        EXMEM_aluResult<=0;
        EXMEM_branchTarget<=0;
        EXMEM_jumpTargetAddress<=0;
        EXMEM_zero<=0;
        EXMEM_memWriteData<=0;
        EXMEM_writeReg<=0;
        EXMEM_PCPlus4<=0;
        EXMEM_call<=0;
    end
endmodule

module MEMWB(
```

```
    input clock,
    input reset,
    input [1:0] EXMEM_wb,
    input [31:0] memReadData,
    input [31:0] EXMEM_aluResult,
    input [4:0] EXMEM_writeReg,
    input [31:0] EXMEM_PCPlus4,
    input EXMEM_call,

    output reg [1:0] MEMWB_wb,
    output reg [31:0] MEMWB_aluResult,
    output reg [31:0] MEMWB_memReadData,
    output reg [4:0] MEMWB_writeReg,
    output reg [31:0] MEMWB_PCPlus4,
    output reg MEMWB_call
);
    always @ (posedge clock)
    if(!reset)
    begin
        MEMWB_wb<=EXMEM_wb;
        MEMWB_aluResult<=EXMEM_aluResult;
        MEMWB_memReadData<=memReadData;
        MEMWB_writeReg<=EXMEM_writeReg;
        MEMWB_call<=EXMEM_call;
        MEMWB_PCPlus4<=EXMEM_PCPlus4;
    end
    else
    begin
        MEMWB_wb<=0;
        MEMWB_aluResult<=0;
        MEMWB_memReadData<=0;
        MEMWB_writeReg<=0;
        MEMWB_call<=0;
        MEMWB_PCPlus4<=0;
    end

endmodule
```

对于流水线寄存器来说，它们的读取和写入应该是同步的，因为在流水线中执行的每一条指令它们进入下一个阶段的时刻应该是相同的，否则就会发生复写。在这里我们约定流水线寄存器总是在时钟的上升沿写入。

3.1 IF/ID 寄存器

IF/ID 寄存器比较简单，只需要存储从指令存储器读取的指令以及 PC+4。

3.2 ID/EX 寄存器

在译码阶段，寄存器需要从 IF/ID 寄存器读取 rs、rt 以读取数据，对指令的低 16 位进行符号扩展，将指令输入到控制单元决定是否写寄存器、是否跳转、是否分支、执行何种运算、运算数来源等。这些信息被保存在 ID/EX 寄存器中，供之后的几个阶段使用。

3.3 EX/MEM 寄存器

执行阶段 ALU 计算出的结果以及跳转或分支地址需要被保存下来，连通其他的控制信息需要被 EX/MEM 寄存器保留。

3.4 MEM/WB 寄存器

内存访问阶段从数据存储器读取的数据和执行阶段的计算结果需要被存储，供写回阶段写入寄存器。写回所需要的控制信息也同样需要存储在 MEM/WB 寄存器中。

4 五个流水线阶段的实现

4.1 基本信号线的定义

```
    reg [31:0] PC;
    wire REG_DST,JUMP,JUMP1,BRANCH,MEM_READ;
    wire MEM_TO_REG,MEM_WRITE,JUMPTARGET,CALL;
    wire [2:0] ALU_OP;
    wire ALU_SRC,REG_WRITE;//Signals generated by control unit

    wire [3:0] ALUCTR;

    wire [31:0] INST;//PC and instruction memory

    wire [4:0] WRITEREG;
    wire [4:0] READREG1;
    wire [4:0] READREG2;
    wire [31:0] REGREADDATA1;
    wire [31:0] REGREADDATA2;
    wire [31:0] REGWRITEDATA;//register file

    wire [31:0] INSTSHIFTED;
    wire [31:0] SIGNEXTENDED;
    wire [31:0] EXTENDSHIFTED;

    wire [31:0] ALUSRC1;
    wire [31:0] ALUSRC2;

    wire ZERO;
    wire [31:0] ALURSLT;//main ALU

    wire [31:0] MEM_DATA;
    wire [31:0] PC_PLUS_4;

    wire [31:0] IFID_PCPLUS4,IFID_INST;
    wire [31:0] IDEX_PCPLUS4,IDEX_READDATA1;
    wire [31:0] IDEX_READDATA2,IDEX_SIGNEXTENDED;
    wire [4:0] IDEX_RS;
    wire [4:0] IDEX_RT,IDEX_RD,IDEX_EX;
    wire [4:0] IDEX_M;
```

```
    wire [1:0] IDEX_WB;
    wire IDEX_CALL;
    wire [1:0] EXMEM_WB;
    wire [4:0] EXMEM_M;
    wire [31:0] EXMEM_BRANCHTARGET,EXMEM_JUMPTARGETADDRESS;
    wire [31:0] EXMEM_ALURESULT,EXMEM_MEMWRITEDATA,EXMEM_PCPLUS4;
    wire [4:0] EXMEM_WRITEREG;
    wire EXMEM_ZERO,EXMEM_BRANCH,EXMEM_CALL;
    wire CTRL_FLUSH;
    wire STALL;
    wire MEMWB_REGWRITE,MEMWB_MEMTOREG,MEMWB_CALL;
    wire [31:0] MEMWB_MEMREADDATA,MEMWB_ALURESULT,MEMWB_PCPLUS4;
    wire [4:0] MEMWB_WRITEREG;
    wire [1:0] FORWARDA,FORWARDB;
    wire [31:0] INPUT1,INPUT2;
```

4.2 取指令阶段

取指令阶段比较简单。指令存储器根据 PC 值取对应地址所存储的指令，存放到 IF/ID 寄存器中，PC+4 的值也同样需要被存储，供后续跳转相关操作使用。然后更新 PC 值，为下一个周期的取指令做准备。

```

instrMemory instruction_memory(
        .readAddress(PC),

        .instruction(INST)
    );

    assign PC_PLUS_4=PC+4;
    assign CTRL_FLUSH=(EXMEM_ZERO&&EXMEM_M[2])||EXMEM_M[3];
    assign STALL=IDEX_M[1]&&
    ((IDEX_RT==IFID_INST[25:21])||(IDEX_RT==IFID_INST[20:16]));
    always @ (posedge clock)
    begin
        if(reset) PC<=0;
        else PC<=STALL?PC:
        ((EXMEM_ZERO&&EXMEM_M[2])?EXMEM_BRANCHTARGET:
         (EXMEM_M[3]?EXMEM_JUMPTARGETADDRESS:PC_PLUS_4));
    end
```

4.3 译码阶段

译码阶段需要根据 IF/ID 寄存器中的指令，读取寄存器中的值，并将指令送入控制单元解析之后几个阶段的控制信号，顺便对指令的低 16 为进行符号扩展。需要注意的是，写回阶段也需要寄存器的参与，其主要行为是根据控制信号决定是否写寄存器，以及写入数据来自于内存还是 ALU 的计算结果。

```
    Ctr ctr(
        .opCode(IFID_INST[31:26]),
```

```
3
4           .regDest(REG_DST),
5           .aluSrc(ALU_SRC),
6           .memToReg(MEM_TO_REG),
7           .regWrite(REG_WRITE),
8           .memRead(MEM_READ),
9           .memWrite(MEM_WRITE),
10          .Branch(BRANCH),
11          .ALUop(ALU_OP),
12          .Jump(JUMP),
13          .Call(CALL),
14          .jumpTarget(JUMPTARGET)
15      );
16
17      Registers registers(
18          .clock(clock),
19          .reset(reset),
20          .readReg1(IFID_INST[25:21]),
21          .readReg2(IFID_INST[20:16]),
22          .writeReg(MEMWB_WRITEREG),
23          .writeData(MEMWB_MEMTOREG?
24          MEMWB_MEMREADDATA:
25          (MEMWB_CALL?MEMWB_PCPLUS4:MEMWB_ALURESULT)),
26          .regWrite(MEMWB_REGWRITE),
27
28          .readData1(REGREADDATA1),
29          .readData2(REGREADDATA2)
30      );
31      signext signext(
32          .inst(IFID_INST[15:0]),
33          .data(SIGNEXTENDED)
34      );
```

4.4 执行阶段

执行阶段需要 ALU、ALU 控制单元以及一个加法器（用来计算跳转或分支地址）。上一步控制器解析控制信号后，在 IDEX_EX 域中存储了 ALU_SRC、ALU_OP、REG_DST 信号，用来决定 ALU 运算数的来源、ALU 所执行的操作、写回阶段的目标寄存器。

```
1       ALUCtr aluctr(
2           .ALUop(IDEX_EX[3:1]),
3           .functField(IDEX_SIGNEXTENDED[5:0]),
4           .operation(ALUCTR),
5           .Jump(JUMP1)
6       );
7       ALU alu
8           .input1(INPUT1),
9           .input2(INPUT2),
10          .aluCtr(ALUCTR),
11          .zero(ZERO),
```

```
        .aluRes(ALURSLT)
    );
```

其中 INPUT1 和 INPUT2 的控制逻辑过于复杂，且需转发机制的参与，所以在随后补充。

4.5 内存访问阶段

内存访问阶段需要根据内存访问阶段所需要的控制信号以及执行阶段 ALU 计算结果作为输入，去访问数据存储器中的数据。跳转指令和分支指令改变 PC 值也发生在这一步。

```
    dataMemory data_memory(
        .clock(clock),
        .memWrite(EXMEM_M[0]),
        .memRead(EXMEM_M[1]),
        .address(EXMEM_ALURESULT),
        .writeData(EXMEM_MEMWRITEDATA),
        .readData(MEM_DATA)
    );
```

4.6 写回阶段

写回阶段需要寄存器的参与。根据 MEM/WB 寄存器中的控制信号决定是否写回、写回 ALU 计算结果还是内存访问的数据。

寄存器单元中的这几行：

```
    .writeReg(MEMWB_WRITEREG),
    .writeData(MEMWB_MEMTOREG?
    MEMWB_MEMREADDATA:
    (MEMWB_CALL?MEMWB_PCPLUS4:MEMWB_ALURESULT))
```

展示了根据控制信号写回数据的过程。

5 冒险

5.1 结构冒险

结构冒险主要是因为在同一个时刻有多条指令试图访问同一个结构单元。在 MIPS 指令中，存储器可能会出现这种情况。我们的解决方案是同时设置指令存储器和数据存储器。

5.2 数据冒险

在单周期 CPU 中，一条指令执行完毕之前，不会有其他指令占用 CPU。而流水线 CPU 中，一条指令还未执行完毕，下一条指令就已经开始执行。由于写回操作是每一条指令的最后一步操作，所以当前一条指令试图写、后一条指令试图读同一个寄存器时，会发生数据尚未写入寄存器的情况。这时后一条指令会读取错误的数据。因此需要设置某种机制来解决这种问题。数据冒险主要分为加载-使用冒险、写后读冒险、写后写冒险。

5.2.1 加载-使用冒险 (load-use hazard)

```
    lw $1,4($0)
    add $3,$1,$1
    add $4,$1,$1
    add $5,$1,$1
    add $6,$1,$1
```

加载-使用冒险可能会涉及加载指令之后的四条指令。这四条指令的处理方式不同。

1. 加载指令后的第三条指令的译码阶段与加载指令的写回阶段重合。可以要求寄存器在前半周期写，后半周期读，这样可以很便捷地解决这种问题。

```
    always @ (negedge clock)
    begin
        if (reset)
          begin
            integer i;
            for(i=0;i<32;i=i+1) regFile[i]=0;

         end
        if(regWrite==1&&writeReg)
        begin
            regFile[writeReg]=writeData;
            if(writeReg==readReg1) readData1=writeData;
            if(writeReg==readReg2) readData2=writeData;
        end
    end
```

2. 加载指令之后的第二条指令的执行阶段在加载指令内存访问阶段的后面。可以通过转发机制解决。代码在转发机制部分的描述中给出。

3. 加载指令之后的第一条指令的执行阶段与加载指令内存访问阶段重合，无法通过转发机制解决。只能暂停流水线一个周期，使得后面的第一条指令“变成”第二条指令，通过转发机制解决。

5.2.2 写后读冒险 (read-after-write hazard)

写后读冒险主要发生在写相同寄存器的连续两条指令中。

```
    add $3,$1,$2
    add $4,$3,$3
```

上面的代码中，前一条指令所要写的寄存器，在被写入之前就被后一条指令所读。这种冒险可以通过转发机制解决。

5.2.3 写后写冒险 (write-after-write hazard)

```
    lw $1,4($0)
    add $1,$2,$3
    add $4,$1,$1
```

这时需要根据控制信息，使得第二条加法指令的结果被转发到第三条指令中。

5.3 控制冒险

控制冒险主要发生在跳转指令和分支指令中。在这里我们总是预测跳转或分支不会发生。如果的确需要跳转或分支，再刷新流水线，将不可以执行的指令用 nop 替代。

6 处理数据冒险的方式——转发机制

转发机制的控制逻辑在《计算机组成与设计：软硬件接口》中有详细的描述。这里给出实现。

```
    module forwardingUnit(
    input clock,
    input [4:0] IDEX_rs,
    input [4:0] IDEX_rt,
    input [4:0] EXMEM_regDest,
    input [4:0] MEMWB_regDest,
    input EXMEM_regWrite,
    input MEMWB_regWrite,

    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB

    );
    always @ (negedge clock)
    begin
        ForwardA=2'b00;
        ForwardB=2'b00;
        if(EXMEM_regWrite &&
                EXMEM_regDest
                && (EXMEM_regDest==IDEX_rs))
                ForwardA=2'b10;
        if(EXMEM_regWrite &&
           EXMEM_regDest&&
           (EXMEM_regDest==IDEX_rt))
                ForwardB=2'b10;
        if(MEMWB_regWrite &&
           MEMWB_regDest &&
           !(EXMEM_regWrite &&
           EXMEM_regDest &&
           (EXMEM_regDest!=IDEX_rs)))
                ForwardA=2'b01;
        if(MEMWB_regWrite &&
           MEMWB_regDest &&
           !(EXMEM_regWrite &&
           EXMEM_regDest &&
           (EXMEM_regDest!=IDEX_rt)))
                ForwardB=2'b01;
    end
endmodule
```

相应地，在执行阶段，也需要对 ALU 运算数来源进行控制。值得注意的是，ALU 的第二个运算数可能来自于立即数。所以如果在执行阶段发现当前指令需要使用立即数，则应该优先使用立即

数作为操作数。在实验要求的 16 条指令中，左移和右移指令比较特殊。被左移的运算数来自 rt，而 rs 域总为 0。因此可以使用 INPUT1 来传递左移的位数。

```
    assign INPUT1=
    (FORWARDA==2'b10)?
        EXMEM_ALURESULT:
       ((FORWARDA==2'b01)?
           (MEMWB_MEMTOREG?MEMWB_MEMREADDATA:MEMWB_ALURESULT):
           ((ALUCTR==4'b0011||ALUCTR==4'b0100)?
           {27'h0,IDEX_SIGNEXTENDED[10:6]}:IDEX_READDATA1));
    assign INPUT2=IDEX_EX[4]?IDEX_SIGNEXTENDED:
                   ((FORWARDB==2'b10)?EXMEM_ALURESULT:
                     ((FORWARDB==2'b01)?
                     (MEMWB_MEMTOREG?
                     MEMWB_MEMREADDATA:
                     MEMWB_ALURESULT):
                     IDEX_READDATA2));
    ALU alu(
        .input1(INPUT1),
        .input2(INPUT2),
        .aluCtr(ALUCTR),
        .zero(ZERO),
        .aluRes(ALURSLT)
    );
```

7 刷新流水线和暂停流水线机制

刷新流水线机制主要应用于跳转指令和分支指令。主要操作是将流水线寄存器刷新为 0，也就是用 nop 来替代不应执行的指令。暂停流水线机制主要应用于加载-使用冒险。暂停流水线机制把 PC 和 ID/EX 寄存器的值重新写回，并且把 nop 写入 ID/EX 寄存器中，插入流水线气泡使流水线暂停。

```
    wire CTRL_FLUSH;
    wire STALL;
    assign CTRL_FLUSH=(EXMEM_ZERO&&EXMEM_M[2])||EXMEM_M[3];
    assign STALL=IDEX_M[1]&&
    ((IDEX_RT==IFID_INST[25:21])||(IDEX_RT==IFID_INST[20:16]));
    always @ (posedge clock)
    begin
        if(reset) PC<=0;
        else PC<=STALL?PC:
        ((EXMEM_ZERO&&EXMEM_M[2])?
        EXMEM_BRANCHTARGET:
        (EXMEM_M[3]?EXMEM_JUMPTARGETADDRESS:PC_PLUS_4));
    end
    IFID ifid(//
        .clock(clock),
        .reset(reset),
        .instruction_in(CTRL_FLUSH?0:(STALL?IFID_INST:INST)),
        .PC_Plus_4(CTRL_FLUSH?0:(STALL?IFID_PCPLUS4:PC_PLUS_4)),
```

```
19
20          .IFID_PCplus4(IFID_PCPLUS4),
21          .IFID_instruction(IFID_INST)
22      );
23          IDEX idex(//
24          .clock(clock),
25          .reset(reset),
26          .IFID_PCPlus4(CTRL_FLUSH||STALL?0:IFID_PCPLUS4),
27          .IFID_rs(CTRL_FLUSH||STALL?0:IFID_INST[25:21]),
28          .readData1(CTRL_FLUSH||STALL?0:REGREADDATA1),
29          .readData2(CTRL_FLUSH||STALL?0:REGREADDATA2),
30          .signextended(CTRL_FLUSH||STALL?0:SIGNEXTENDED),
31          .IFID_rt(CTRL_FLUSH||STALL?0:IFID_INST[20:16]),
32          .IFID_rd(CTRL_FLUSH||STALL?0:IFID_INST[15:11]),
33          .ex(CTRL_FLUSH||STALL?0:{ALU_SRC,ALU_OP,REG_DST}),
34          .m(CTRL_FLUSH||STALL?0:
35              {JUMPTARGET,JUMP,BRANCH,MEM_READ,MEM_WRITE}),
36          .wb(CTRL_FLUSH||STALL?0:{REG_WRITE,MEM_TO_REG}),
37          .call(CALL),
38          .IDEX_PCPlus4(IDEX_PCPLUS4),
39          .IDEX_readData1(IDEX_READDATA1),
40          .IDEX_readData2(IDEX_READDATA2),
41          .IDEX_signextended(IDEX_SIGNEXTENDED),
42          .IDEX_rs(IDEX_RS),
43          .IDEX_rt(IDEX_RT),
44          .IDEX_rd(IDEX_RD),
45          .IDEX_m(IDEX_M),
46          .IDEX_wb(IDEX_WB),
47          .IDEX_ex(IDEX_EX),
48          .IDEX_call(IDEX_CALL)
49      );
```

8 仿真验证

8.1 验证程序

本次实验使用了如下的验证程序：

```
1   lw $1,4($0) # $1=1, a nop should be inserted
2   add $1,$1,$1 # $1=2
3   add $2,$1,$1 # $2=4, solve data hazard by forwarding
4   lw $2,8($0) # $2=2
5   lw $3,12($0) # $3=3, a nop should be inserted
6   add $4,$2,$3 # $4=5
7   add $4,$4,$4 # $4=10, solve data hazard by forwarding
8   addi $2,$2,1 # $2=3
9   sll $3,$3,2 # $3=6
10  beq $0,$0,1 # predict not taken, otherwise flush the pipeline
11  xor $3,$3,$3 # $3=0
12  sw $2,4($0) #(4)=3
```

```
    lw $5,4($0)
    slt $9,$0,$1
    jal 0 # solve control hazard by inserting bubbles
    xor $1,$1,$1
    xor $2,$2,$2
    xor $3,$3,$3
```

经过观察，我们不难发现，这个程序涉及到了加载-使用冒险、写后读冒险、控制冒险，而且涉及了很多特殊的运算指令，可以测试到很多流水线中的关键点，不能不说十分有趣。而如果这个流水线能够通过这个程序的考验，应该就可以认为设计比较成功。

8.2 顶层模块控制程序

```
    module top_tb();
    reg clock,reset;
    always #50 clock=!clock;
    Top top(.clock(clock),.reset(reset));
    initial begin
        $readmemh("mem_data.txt",top.data_memory.memFile);
        $readmemb("demo-5.txt",top.instruction_memory.instrFile);
        clock=0;
        reset=1;
        #75
        reset=0;
    end
endmodule
```

8.3 仿真波形图

图 2:

从图2、图3、图4可以看出，流水线 CPU 成功地读入了指令存储器中存储的指令，机器代码与汇编代码相符合。


ardingUnit.v
top_tb.v
ALUCtr.v
dataMemory.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
clock
reset
PC[31:0]
readAddress[31:0]
instruction[31:0]
instrFile[31:0][31:0]
instruction_in[31:0]
PC_Plus_4[31:0]
IFID_PCplus4[31:0]
IFID_instruction[31:0]
opCode[5:0]
regDest

图 3:

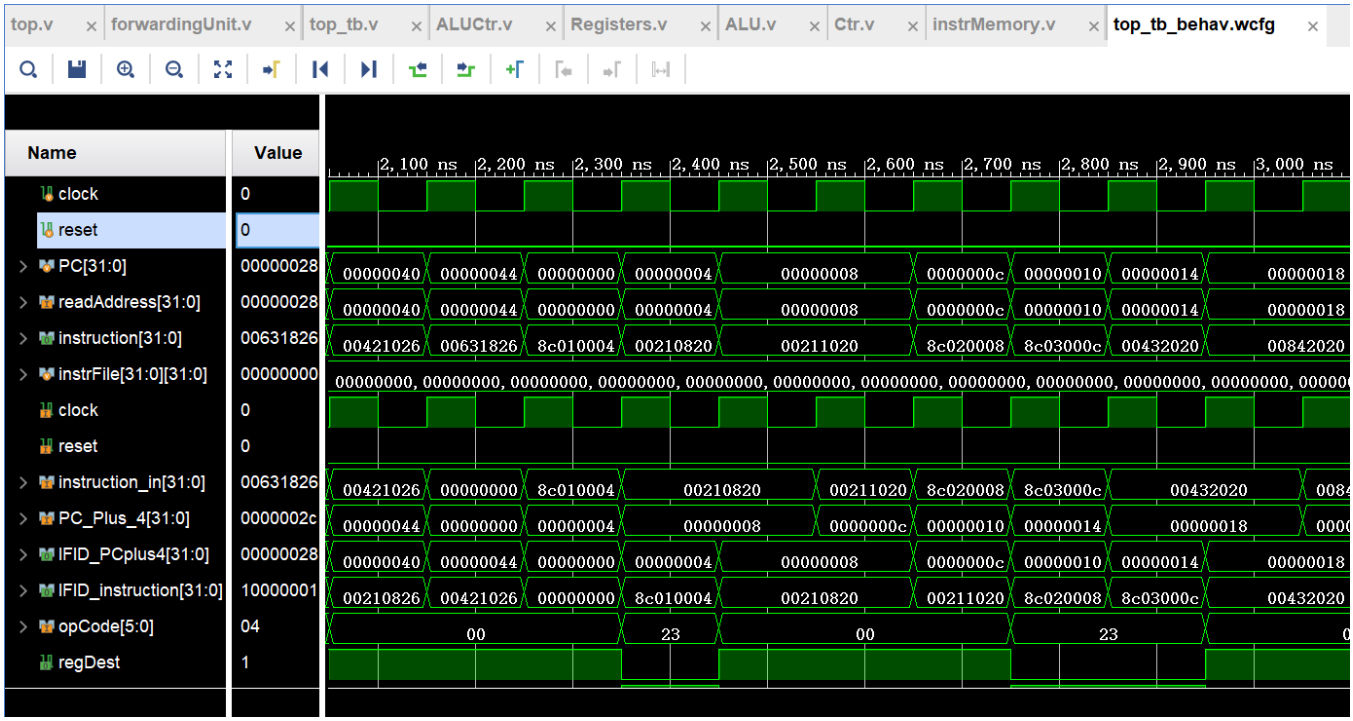
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
clock
reset
PC[31:0]
readAddress[31:0]
instruction[31:0]
instrFile[31:0][31:0]
instruction_in[31:0]
PC_Plus_4[31:0]
IFID_PCplus4[31:0]
IFID_instruction[31:0]
opCode[5:0]
regDest

图 4:

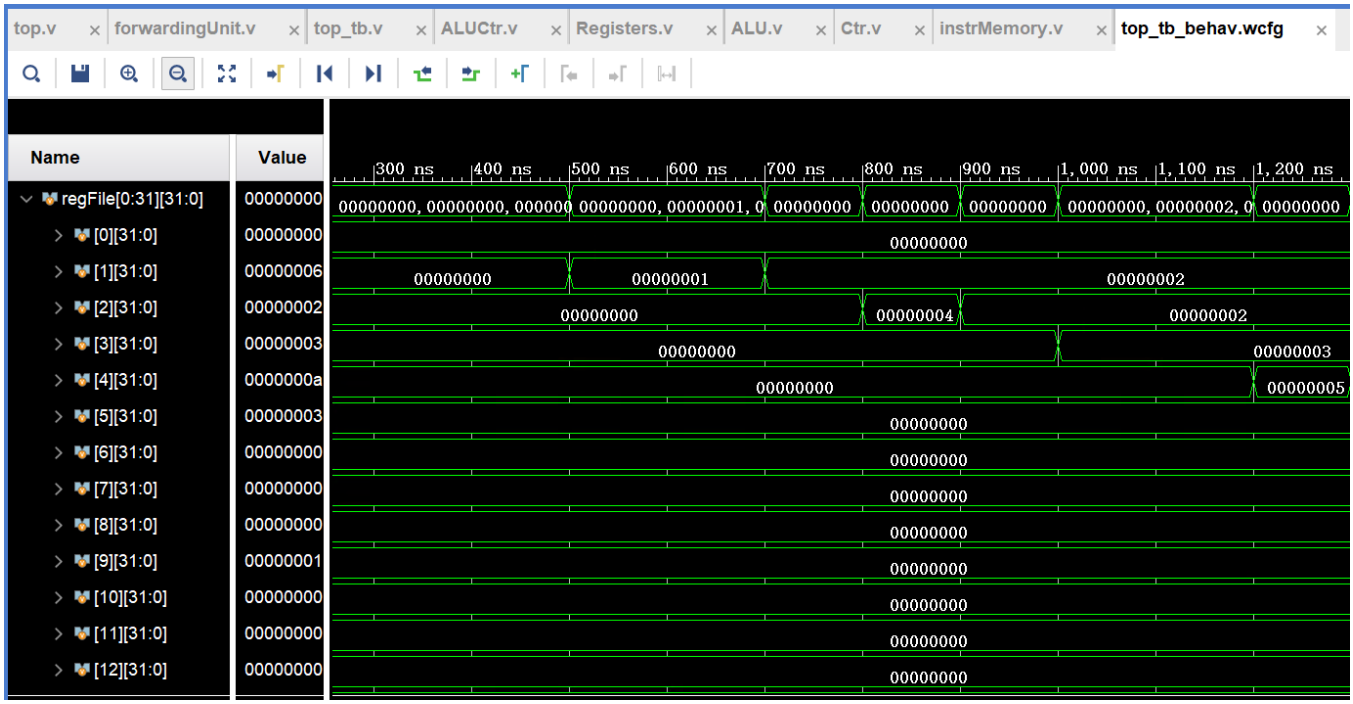
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
regFile[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
00000000
00000006
00000002
00000003
0000000a
00000001
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000 ns
1,100 ns
1,200 ns

图 5:

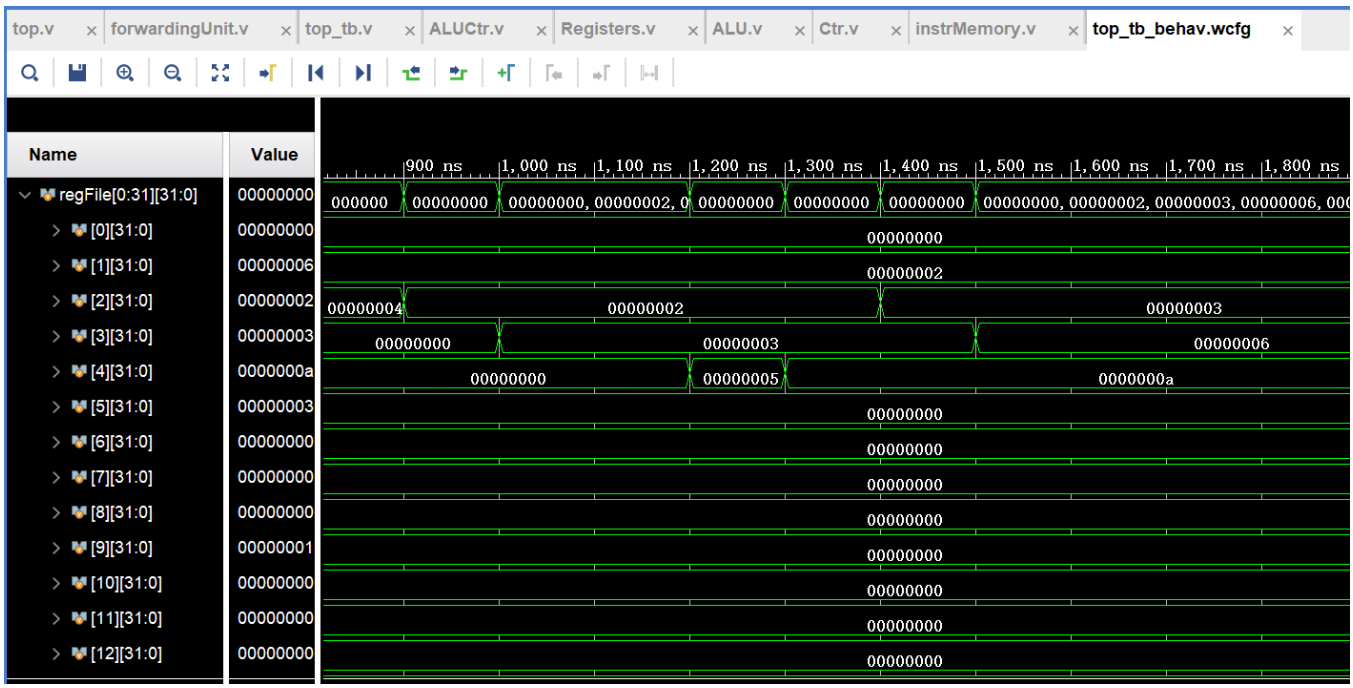
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
regFile[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
00000000
00000006
00000002
00000003
0000000a
00000001
900 ns
1,000 ns
1,100 ns
1,200 ns
1,300 ns
1,400 ns
1,500 ns
1,600 ns
1,700 ns
1,800 ns

图 6:

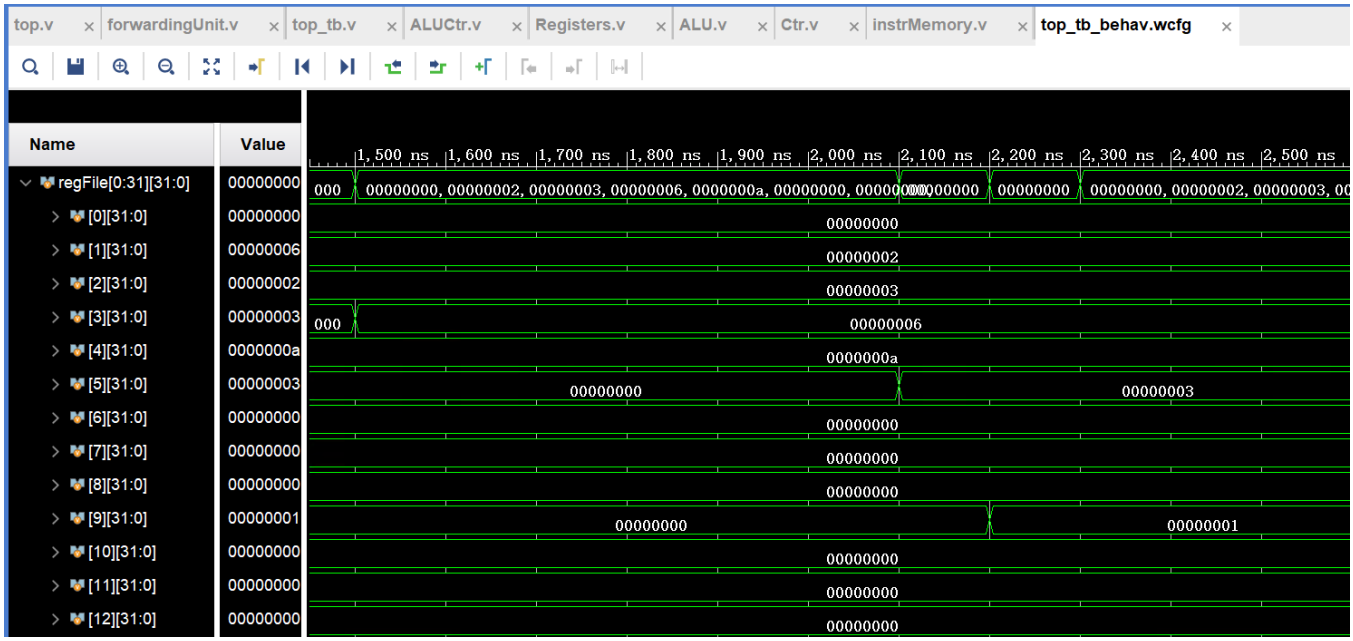
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
1,500 ns
1,600 ns
1,700 ns
1,800 ns
1,900 ns
2,000 ns
2,100 ns
2,200 ns
2,300 ns
2,400 ns
2,500 ns
regFile[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
00000000
00000006
00000002
00000003
0000000a
00000001
000
00000000, 00000002, 00000003, 00000006, 0000000a, 00000000, 00000000
00000000
00000002
00000003
00000006
0000000a
00000003
00000001

图 7:

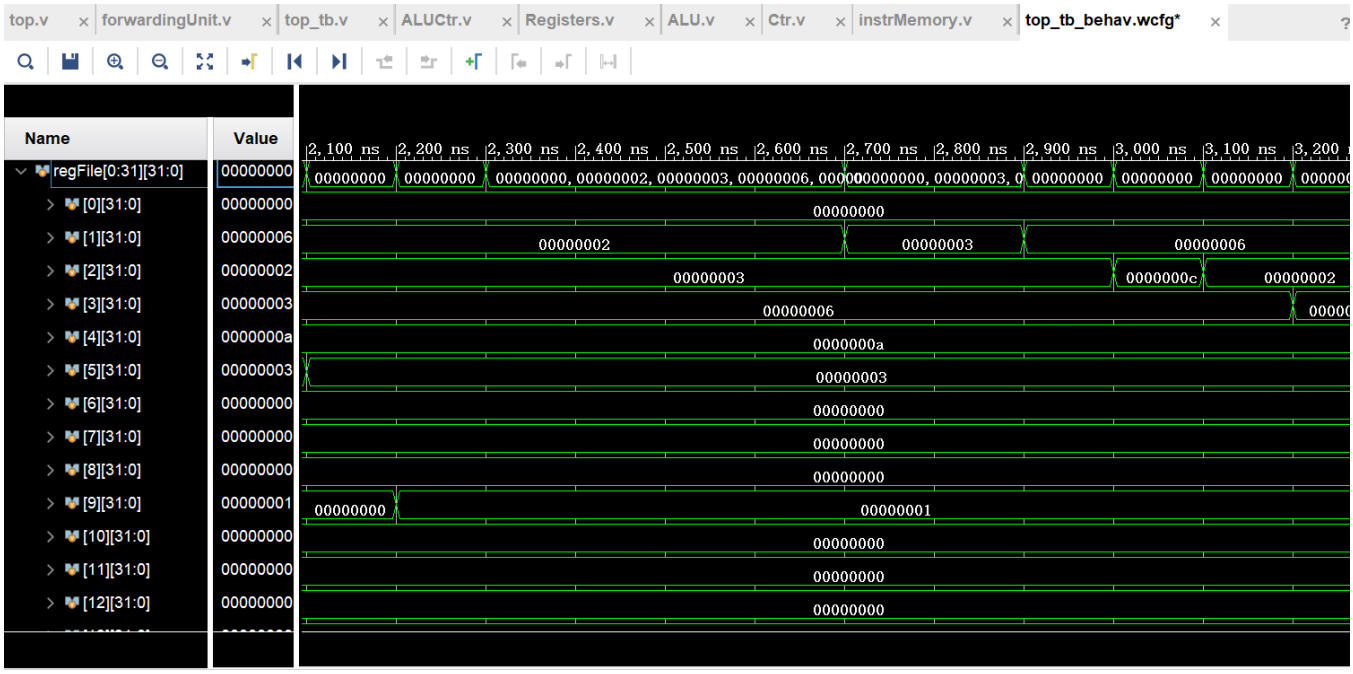
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg*
Name
Value
2,100 ns
2,200 ns
2,300 ns
2,400 ns
2,500 ns
2,600 ns
2,700 ns
2,800 ns
2,900 ns
3,000 ns
3,100 ns
regFile[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
00000000
00000006
00000002
00000003
0000000a
00000001
00000000, 00000002, 00000003, 00000006
00000000, 00000003
00000000
00000002
00000003
00000006
0000000c
00000003
00000001

图 8:

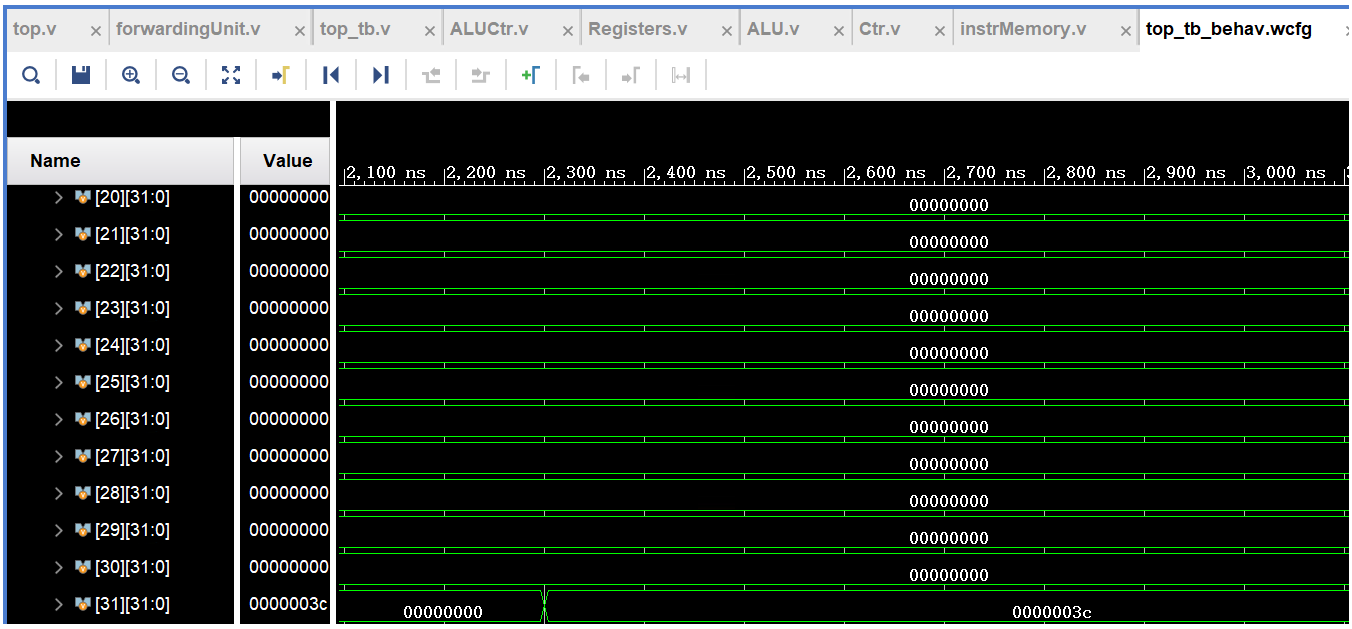
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
[20][31:0] 00000000
[21][31:0] 00000000
[22][31:0] 00000000
[23][31:0] 00000000
[24][31:0] 00000000
[25][31:0] 00000000
[26][31:0] 00000000
[27][31:0] 00000000
[28][31:0] 00000000
[29][31:0] 00000000
[30][31:0] 00000000
[31][31:0] 0000003c
2,100 ns
2,200 ns
2,300 ns
2,400 ns
2,500 ns
2,600 ns
2,700 ns
2,800 ns
2,900 ns
3,000 ns
00000000
0000003c

图 9:

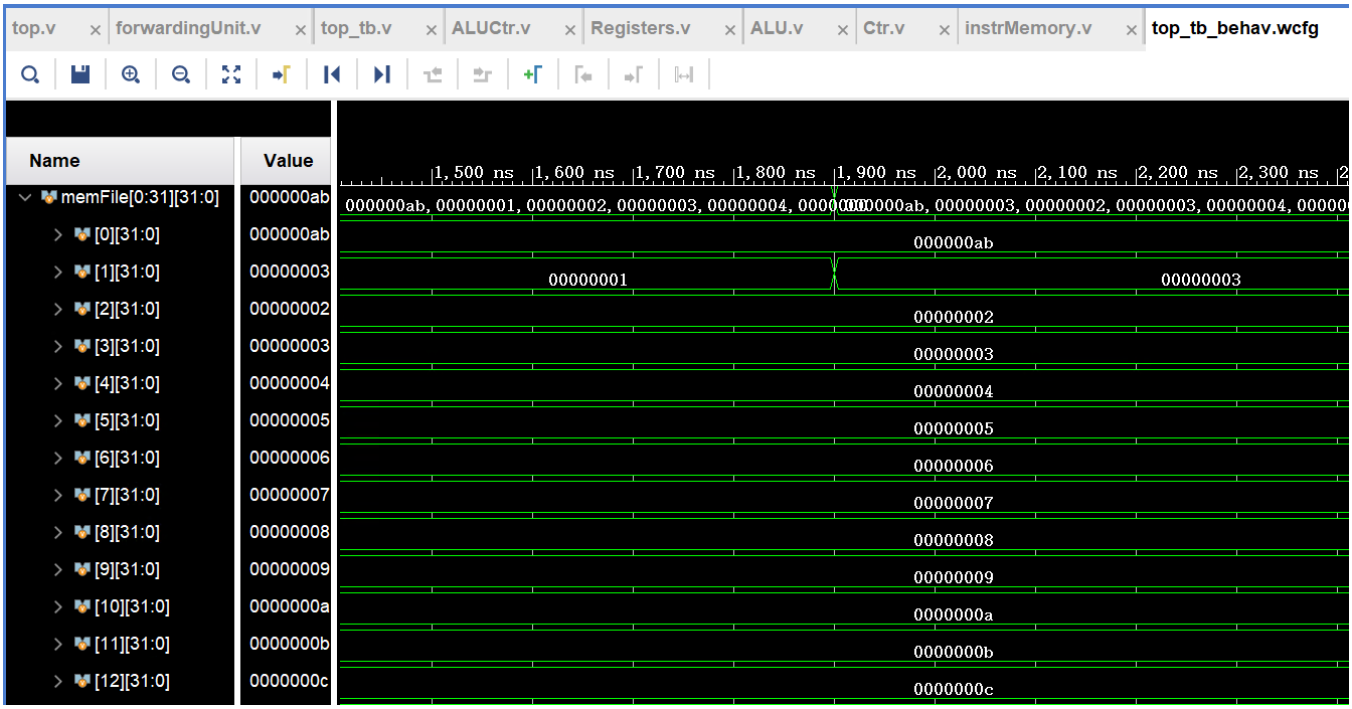
top.v
forwardingUnit.v
top_tb.v
ALUCtr.v
Registers.v
ALU.v
Ctr.v
instrMemory.v
top_tb_behav.wcfg
Name
Value
memFile[0:31][31:0] 000000ab
[0][31:0] 000000ab
[1][31:0] 00000003
[2][31:0] 00000002
[3][31:0] 00000003
[4][31:0] 00000004
[5][31:0] 00000005
[6][31:0] 00000006
[7][31:0] 00000007
[8][31:0] 00000008
[9][31:0] 00000009
[10][31:0] 0000000a
[11][31:0] 0000000b
[12][31:0] 0000000c
1,500 ns
1,600 ns
1,700 ns
1,800 ns
1,900 ns
2,000 ns
2,100 ns
2,200 ns
2,300 ns
000000ab, 00000001, 00000002, 00000003, 00000004,
000000ab, 00000003, 00000002, 00000003, 00000004,
00000001
00000003

图 10:

从图5、图6、图7、图8、图9、图10可以看出，寄存器所存储的值的变化和变化时间比较符合预期：

1. 验证程序第 2 行、第 6 行的加法指令，写回操作比前一行晚两个时钟周期。这说明插入了一个流水线气泡。
2. 第 6 行和第 7 行写回时间相差一个周期，写回数据正确，说明通过转发机制成功传递了数据。
3. 第 10 行和第 16 行的分支、跳转指令之后紧跟几条将寄存器设置为 0 的指令，其破坏性较大。而从波形图中可以看出这些寄存器并未写入 0。而如果将第 10 行改为”beq $0,$1,1”，则可以看到三号寄存器被清零。且清零的时间符合没有控制流转移时理论上应该写入的时间。这可以看出流水线采用了 branch-not-taken 策略，减少了流水线气泡的数量。

9 拓展: 使 CPU 支持 32 种指令

9.1 概述

前面已经实现了支持 16 种指令的 CPU，现在我们希望将支持的指令拓展到 32 种。通过观察参考资料中 MIPS 的指令，我们可以看出，算数运算中多出了有符号和无符号的分别。对于常见的 beq 指令，多出了 bne 等指令。总的来说，我们只需要针对新出现的指令，对控制单元、ALU、ALU 控制单元进行扩展。

9.2 控制单元

控制单元的变化主要有：新增两位信号线 Signed 与一位信号线 branchEqual。其中 Signed 用来表示是否为有符号运算。若 Signed==2’b01，则为有符号运算。若 Signed==2’b00，则为无符号运算。若 Signed==2’b10，则专门为 lui 指令设计，表示将寄存器的值左移 16 位。左移功能交给符号扩展单元实现。branchEqual 用来表示如果指令为分支指令，应该是 beq 还是 bne。

```
1    module Ctr(
2    input [5:0] opCode,
3
4
5    output reg regDest,
6    output reg aluSrc,
7    output reg memToReg,
8    output reg regWrite,
9    output reg memRead,
10   output reg memWrite,
11   output reg Branch,
12   output reg [2:0] ALUop,
13   output reg Jump,
14   output reg jumpTarget,
15   output reg Call,
16   output reg [1:0] Signed,
17   output reg branchEqual
18   );
19   always @(opCode)
20   begin
21       case(opCode)
22           6'b000000://I-type
```

```
23              begin
24                  regDest=1;
25                  aluSrc=0;
26                  memToReg=0;
27                  regWrite=1;
28                  memRead=0;
29                  memWrite=0;
30                  Branch=0;
31                  Jump=0;
32                  jumpTarget=0;
33                  Call=0;
34                  ALUop=3'b010;
35                  Signed=0;
36              end
37              6'b000010://jump
38              begin
39                  regDest=0;
40                  aluSrc=0;
41                  memToReg=0;
42                  regWrite=0;
43                  memRead=0;
44                  memWrite=0;
45                  Branch=0;
46                  Jump=1;
47                  jumpTarget=1;
48                  Call=0;
49                  ALUop=3'b000;
50                  Signed=0;
51              end
52              6'b000011://jal
53              begin
54                  regDest=1;
55                  aluSrc=0;
56                  memToReg=0;
57                  regWrite=1;
58                  memRead=0;
59                  memWrite=0;
60                  Branch=0;
61                  Jump=1;
62                  jumpTarget=1;
63                  Call=1;
64                  ALUop=3'b000;
65                  Signed=0;
66              end
67              6'b001000://addi
68              begin
69                  regDest=0;
70                  aluSrc=1;
71                  memToReg=0;
72                  regWrite=1;
```

```
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
                Signed=0;
            end
            6'b001001://addiu
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
                Signed=1;
            end
            6'b001100://andi
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b000;
                Signed=0;
            end
            6'b001101://ori
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
```

```
123                 Call=0;
124                 ALUop=3'b000;
125                 Signed=0;
126             end
127             6'b001110://xori
128             begin
129                 regDest=0;
130                 aluSrc=1;
131                 memToReg=0;
132                 regWrite=1;
133                 memRead=0;
134                 memWrite=0;
135                 Branch=0;
136                 Jump=0;
137                 jumpTarget=0;
138                 Call=0;
139                 ALUop=3'b000;
140                 Signed=0;
141             end
142             6'b001111://lui
143             begin
144                 regDest=0;
145                 aluSrc=1;
146                 memToReg=0;
147                 regWrite=1;
148                 memRead=0;
149                 memWrite=0;
150                 Branch=0;
151                 Jump=0;
152                 jumpTarget=0;
153                 Call=0;
154                 ALUop=3'b000;
155                 Signed=2'b10;
156             end
157             6'b100011://lw
158             begin
159                 regDest=0;
160                 aluSrc=1;
161                 memToReg=1;
162                 regWrite=1;
163                 memRead=1;
164                 memWrite=0;
165                 Branch=0;
166                 Jump=0;
167                 jumpTarget=0;
168                 Call=0;
169                 ALUop=3'b000;
170                 Signed=1;
171             end
172             6'b101011://sw
```

```
173             begin
174                 regDest=1;
175                 aluSrc=1;
176                 memToReg=0;
177                 regWrite=0;
178                 memRead=0;
179                 memWrite=1;
180                 Branch=0;
181                 Jump=0;
182                 jumpTarget=0;
183                 Call=0;
184                 ALUop=3'b000;
185                 Signed=1;
186             end
187             6'b000100://beq
188             begin
189                 regDest=1;
190                 aluSrc=0;
191                 regWrite=0;
192                 memToReg=0;
193                 memRead=0;
194                 memWrite=0;
195                 Branch=1;
196                 Jump=0;
197                 jumpTarget=0;
198                 Call=0;
199                 ALUop=3'b001;
200                 Signed=1;
201                 branchEqual=1;
202             end
203             6'b000101://bne
204             begin
205                 regDest=1;
206                 aluSrc=0;
207                 regWrite=0;
208                 memToReg=0;
209                 memRead=0;
210                 memWrite=0;
211                 Branch=1;
212                 Jump=0;
213                 jumpTarget=0;
214                 Call=0;
215                 ALUop=3'b001;
216                 Signed=1;
217                 branchEqual=0;
218             end
219             6'b001010://slti
220             begin
221                 regDest=0;
222                 aluSrc=1;
```

```
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b111;
                Signed=1;
            end
            6'b001011://sltiu
            begin
                regDest=0;
                aluSrc=1;
                memToReg=0;
                regWrite=1;
                memRead=0;
                memWrite=0;
                Branch=0;
                Jump=0;
                jumpTarget=0;
                Call=0;
                ALUop=3'b010;
                Signed=0;
            end
        endcase
    end
endmodule
```

9.3 ALUCtr

ALUCtr 的任务是根据 ALUop 和 funct 域决定 ALU 的运算种类。

```
module ALUCtr(
    input [2:0] ALUop,
    input [5:0] functField,
    output reg [3:0] operation,
    output reg Jump
    );
    always @ (ALUop or functField)
    begin
        Jump=0;
        casex({ALUop,functField})
            9'b000xxxxxx:operation=4'b0010;//lw,sw,addi,jump,jal +
            9'b001xxxxxx:operation=4'b1001;//beq,bne -u
            9'b011xxxxxx:operation=4'b0000;//andi &
            9'b100xxxxxx:operation=4'b0001;//ori |
            9'b101xxxxxx:operation=4'b0011;// lui <<
            9'b110xxxxxx:operation=4'b1010;//sltiu
```

```
            9'b111xxxxxx:operation=4'b0111;//slti

            9'b010100100:operation=4'b0000;//and &
            9'b010100101:operation=4'b0001;//or |
            9'b010100000:operation=4'b0010;//add +
            9'b010000000:operation=4'b0011;//sll <<
            9'b010000010:operation=4'b0100;//srl >>
            9'b010100110:operation=4'b0101;//xor
            9'b010100010:operation=4'b0110;//sub -
            9'b010101010:operation=4'b0111;//slt a<b?1:0
            9'b010100001:operation=4'b1000;//addu +u
            9'b010100011:operation=4'b1001;//subu -u
            9'b010101011:operation=4'b1010;//sltu
            9'b010100111:operation=4'b1011;//nor
            9'b010000011:operation=4'b1100;//sra
            9'b010000100:operation=4'b0011;//sllv
            9'b010000110:operation=4'b0100;//srlv
            9'b010000111:operation=4'b1100;//srav
            9'b010001000:
            begin
            operation=4'b0010;//jr
            Jump=1;
            end
        endcase
    end
endmodule
```

9.4 ALU

ALU 的变化主要是需要支持有符号运算与无符号运算。两种运算的不同体现在比较大小和是否关注溢出上。有符号运算会关注加减运算中的溢出现象，而无符号不关注。

对于有符号数比较大小的操作，若两数符号位相同，则可以当作无符号数直接使用 Verilog 中的比较符号进行操作。否则需要考虑符号位，符号为为 1 的数更小。

同样,在加减运算中需要考虑溢出现象。对于两有符号整数 a 和 b,溢出的判定条件为 (a>0&&b>0&&a

对于算数右移操作，维护一个计数器，每次逻辑右移后在最高位补 1。

```
    module ALU(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output reg zero,
    output reg overflow,
    output reg [31:0] aluRes
    );
    reg [31:0] complement;
    integer i;
    always @(input1 or input2 or aluCtr)
    begin
        case(aluCtr)
            4'b0000:
```

```
15                  begin
16                      aluRes=input1 & input2;
17                      if(aluRes==0) zero=1;
18                      else zero=0;
19                      overflow=0;
20                  end
21              4'b0001:
22                  begin
23                      aluRes=input1 | input2;
24                      if(aluRes==0) zero=1;
25                      else zero=0;
26                      overflow=0;
27                  end
28              4'b0010:
29                  begin
30                      aluRes=input1+input2;
31                      zero=aluRes?0:1;
32                      overflow=((input1[31]&input2[31]&!aluRes[31])|
33                               (!input1[31]&!input2[31]&aluRes[31]))?
34                                  1:0;
35                  end
36              4'b0011:
37                  begin
38                      aluRes=input2<<input1;
39                      if(aluRes==0) zero=1;
40                      else zero=0;
41                      overflow=0;
42                  end
43              4'b0100:
44                  begin
45                      aluRes=input2>>input1;
46                      if(aluRes==0) zero=1;
47                      else zero=0;
48                      overflow=0;
49                  end
50              4'b0101:
51                  begin
52                      aluRes=input1 ^ input2;
53                      if(aluRes==0) zero=1;
54                      else zero=0;
55                      overflow=0;
56                  end
57              4'b0110:
58                  begin
59                      if(input2==32'h80000000)
60                          begin
61                              aluRes=input1-input2;
62                              overflow=input1[0]?1:0;
63                          end
64                      else
```

```
65                          begin
66                              complement=(~input2)+1;
67                              aluRes=input1+complement;
68                              overflow=
69                              ((input1[31]&complement[31]&!aluRes[31])||
70                              (!input1[31]&!complement[31]&aluRes[31]))?
71                              1:0;
72                          end
73                      zero=aluRes?0:1;
74                  end
75              4'b0111:
76                  begin
77                      overflow=0;
78                      if(input1[31]==input2[31])
79                      begin
80                          aluRes=(input1<input2)?1:0;
81                          zero=aluRes?0:1;
82                      end
83                      else
84                      begin
85                          aluRes=input1[31]?1:0;
86                          zero=aluRes?0:1;
87                      end
88
89                  end
90              4'b1000:
91                  begin
92                      aluRes=input1+input2;
93                      zero=aluRes?0:1;
94                      overflow=((input1[31]&complement[31]&!aluRes[31])|
95                               (!input1[31]&!input2[31]&aluRes[31]))?
96                                  1:0;
97                  end
98              4'b1001:
99                  begin
100                     aluRes=input1-input2;
101                     zero=aluRes?0:1;
102                     overflow=0;
103                 end
104             4'b1010:
105                 begin
106                     aluRes=(input1<input2)?1:0;
107                     zero=aluRes?0:1;
108                     overflow=0;
109                 end
110             4'b1011:
111                 begin
112                     aluRes=~(input1|input2);
113                     if(aluRes==0) zero=1;
114                     else zero=0;
```

```
                    overflow=0;
                end
            4'b1100:
                begin
                    complement=input2;
                    for(i=input1;i>0;i=i-1)
                    begin
                        complement=complement>>1;
                        complement=({input2[31],31'h0})|complement;
                    end
                    aluRes=complement;
                    overflow=0;
                    zero=aluRes?0:1;
                end
        endcase
    end
endmodule
```

9.5 其他模块

流水线寄存器需要为新增的控制信号保留空间。此外，符号扩展单元如下：

```
    module signext(
    input [15:0] inst,
    output [31:0] data,
    output [31:0] zeroextended,
    output [31:0] leftshifted
    );
    assign data= inst[15]?{16'hffff,inst}:{16'h0000,inst};
    assign zeroextended={16'h0000,inst};
    assign leftshifted={inst,16'h0000};
endmodule
```

10 总结与反思

10.1 实验难点

1. 实验中所需要的数据线比较多，因此如果不设置一个统一的命名规则，则很容易发生错误，浪费调试时间。在本次实验中，我统一规定 wire 类型的数据线统一使用大写字母加下划线的方式，每一条线都标明在哪一个阶段使用，用处是什么。在模块内部，我也将数据线的来源标明。

2. 流水线机制比单周期处理器复杂。因此在调试时需要对照教材中的流水线示意图，还需要逐个检查每一条数据线上的数据，还需要画流水线五个阶段的示意图逐步检查数据依赖和转发过程。在转发过程中的控制逻辑比较复杂，还需要考虑特殊指令需要使用特殊来源的 ALU 操作数，特殊指令带来的特殊的控制流设计需要。这些需要极大的耐心、百折不挠的毅力以及周全的考虑。

10.2 待改进之处

1. 流水线寄存器需要存储控制信息，而教材和实验指导书给出的示意图中将同一个阶段所需信息封装到一起。这利于读者从整体上理解，但不利于工程人员写代码和调试。我在完成实验时试图将这些信息封装起来，但效果不是很好。书写其他模块的代码时需要时不时回顾封装之后的接口，浪费了一些时间。因此或许将每一种控制信息单独命名维护可以加快进度。

2. 对于分支指令的处理可以提前到执行阶段，教材中有相关介绍，但因为各种原因没能在实验中展现出来。

11 实验总结与感想

为期六周的系统结构实验结束了，回顾这六周的实验，我有很多收获。首先，这是对计算机系统结构课程的重要补充。在计算机系统结构课程中，我们学习了一种类 MIPS 指令集架构 CPU 的实现，但这种实现只是停留在理论上。对于单周期 CPU 和流水线 CPU 的很多设计细节，我们只停留在了解阶段，对于具体实现我们依然不很了解，依然有一种很懵懂的感觉。而在实验中我们亲自动手写两个 CPU，这使得我们对于课上所学的指令集、单周期处理器、流水线、指令级并行等概念与实现有了更加深刻的了解。

第二，这次实验课程极大地锻炼了我们的自学能力。在本次实验中，我学会了硬件描述语言 verilog。verilog 使得我们即使不学习数字逻辑设计也能体验处理器设计。相对于《深入理解计算机系统》中 HCL 语言的那种隔靴搔痒的感觉，使用 verilog 进行处理器设计与工业界接轨，思维与底层硬件更接近。这也加深了我们对于硬件的理解。

第三，这次课程历练了我们刻苦钻研，百折不挠，精益求精，缜密思维的能力。硬件代码远比软件代码更复杂。而且对于流水线 CPU 这类比较复杂的硬件，在 debug 的过程中更加需要细心、缜密思考以及对于理论知识的理解。这次实验中我通过查阅资料，仔细思考，遇到 bug 仔细 debug，提高了自己书写硬件代码的能力。面对难以理解的 bug，不气馁，一点一点地查找问题的源头。在成功排除故障之后，自豪感油然而生。总之，经过实验课程的磨练，我提高了知识水平、工程能力，磨练了意志品质。感谢这次实验！