上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

计算机系统结构实验报告lab6

姓　　名：张卫鸣

学　　号：520021911141

专　　业：计算机科学与技术

2022 年 4 月 20 日

计算机系统结构实验lab6

目录

一、实验概述

（一）实验名称

简单的类MIPS多周期流水化处理器实现

（二）实验目的

1. 理解CPU Pipeline、流水线冒险(hazard)及相关性，在lab5基础上设计简单流水线CPU
2. 在1.的基础上设计支持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/竞争、控制冒险和结构冒险
3. 在2.的基础上，增加Forwarding机制解决数据竞争，减少因为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
5. 在4.的基础上，将CPU支持的指令数量从9条或16条扩充至31条，使处理器功能更加丰富
6. 在5.的基础上，增加协处理器CP0支持中断相关机制及中断指令
7. 应用Cache原理，设计Cache Line并进行仿真验证

二、实验描述

（一）各模块分析设计

1. 主控制器模块Ctr

主控制器模块对指令产生的操作码opCode进行译码，向其他部件发送指令对应的控制信号。

在译码阶段我们可以识别R型指令和其他指令的控制信号。在本次实验中，我直接对照实验文档中所给的MIPS31条指令集开始编写控制模块的编码。同时也相较于lab5的模块增加了一些控制信号。控制信号有关的信息如下表1所示。

信号	输入/输出	位数	说明
opCode	input	6	指令操作码
func	input	6	运算功能码
regDst	output	1	目标寄存器选择信号
aluSrc	output	1	ALU第二个操作数来源
memToReg	output	1	写寄存器数据来源
regWrite	output	1	写寄存器信号
memRead	output	1	读取内存信号
memWrite	output	1	写入内存信号
aluOp	output	4	运算类型解析信号
jump	output	1	j指令跳转信号
extSign	output	1	符号扩展信号
jalSign	output	1	jal指令跳转信号
beqSign	output	1	beq指令跳转信号
bneSign	output	1	bne指令跳转信号
luiSign	output	1	载入立即数指令信号
jrSign	output	1	jr指令跳转信号

表1

在这个模块的设计中，我们需要用分支语句来实现不同控制信号的输出，主控制器模块所产生的对应控制信号如下表所示(部分信号用首字母缩写表示)：

Opcode	指令	aluOp	RD	AS	MTR	RW	MR	MW	J	E	jal	beq	bne	lui	jr
000000	R	1101	1	0	0	1	0	0	0	0	0	0	0	0	0
000010	jr	1111	1	0	0	0	0	0	0	0	0	0	0	0	1
001000	addi	0000	0	1	0	1	0	0	0	1	0	0	0	0	0
001001	addiu	0001	0	1	0	1	0	0	0	0	0	0	0	0	0
001100	andi	0100	0	1	0	1	0	0	0	0	0	0	0	0	0
001101	ori	0101	0	1	0	1	0	0	0	0	0	0	0	0	0
001110	xori	0110	0	1	0	1	0	0	0	0	0	0	0	0	0
001111	lui	1111	0	1	0	1	0	0	0	0	0	0	0	1	0
100011	lw	0001	0	1	1	1	1	0	0	1	0	0	0	0	0
101011	sw	0001	0	1	0	0	0	1	0	1	0	0	0	0	0
000100	beq	0011	0	0	0	0	0	0	0	1	0	1	0	0	0
000101	bne	0011	0	0	0	0	0	0	0	1	0	0	1	0	0
001010	slti	1000	0	1	0	1	0	0	0	1	0	0	0	0	0
001011	sltiu	1001	0	1	0	1	0	0	0	0	0	0	0	0	0
000010	j	1111	0	0	0	0	0	0	1	0	0	0	0	0	0
000011	jal	1111	0	0	0	0	0	0	1	0	1	0	0	0	0

表2

除了添加的信号，其他功能实现信息已经在lab5中进行详细描述，此处不再赘述。与之对应的ALUOp和ALUCtrOut的改进扩充在下一节详细描述。

主控制器模块的部分代码如下：

```
always @(opCode or func)
    begin
        case(opCode)
        6'b000000:          //R Type
        begin
            if(func==6'b001000)begin
                RegDst=0;
                RegWrite=0;
                JrSign=1;
                ALUOp=4'b1111;
            end
            else begin
                RegDst=1;
                RegWrite=1;
                JrSign=0;
                ALUOp=4'b1101;
            end
            // regDst=0;
            ALUSrc=0;
```

```
            MemToReg=0;
            // regWrite=0;
            MemRead=0;
            MemWrite=0;
            Jump=0;
            ExtSign=0;
            JalSign=0;
            BeqSign=0;
            BneSign=0;
            LuiSign=0;
            // jrSign=0;
            // aluOp=4'b1111;
            Jump=0;
        end
        6'b001000:          //addi
        begin
            RegDst=0;
            ALUSrc=1;
            MemToReg=0;
            RegWrite=1;
            MemRead=0;
            MemWrite=0;
            Jump=0;
            ExtSign=1;
            JalSign=0;
            BeqSign=0;
            BneSign=0;
            LuiSign=0;
            JrSign=0;
            ALUOp=4'b0000;
        end
        //其他指令类似，此处省略
```

此处代码是根据lab3的主控制器模块增添了一些信号修改得到，具体的实现可以参考lab3的主控制器模块。

2. ALU控制器模块ALUCtr

同上一个实验，ALU控制器模块的输出是根据主控制器模块输出的aluOp和指令中的func共同决定。ALU控制器模块的输出ALUCtrOut传递给ALU运算单元进行不同的计算。在这里我们扩充了指令，ALU控制器模块的解析如表3所示。

由于对于一些R型指令我们没有对应的func指令，所以我们需要用casex进行分支。

指令	ALUOp	func	ALUCtrOut	说明
add	1101	100000	0000	加法运算
addi	0000	xxxxxx	0000	加法运算
addu	1101	100001	0001	加法运算
addiu	0001	xxxxxx	0001	加法运算
lw	0001	xxxxxx	0001	加法运算
sw	0001	xxxxxx	0001	加法运算
sub	1101	100010	0010	减法运算
subu	1101	100011	0011	减法运算
beq	0011	xxxxxx	0011	减法运算
bne	0011	xxxxxx	0011	减法运算
and	1101	100100	0100	逻辑与运算
andi	0100	xxxxxx	0100	逻辑与运算
or	1101	100101	0101	逻辑或运算
ori	0101	xxxxxx	0101	逻辑或运算
xor	1101	100110	0110	逻辑异或运算
xori	0110	xxxxxx	0110	逻辑异或运算
nor	1101	100111	0111	逻辑或非运算
slt	1101	101010	1000	带符号小于时置位
slti	1000	xxxxxx	1000	带符号小于时置位
sltu	1101	101011	1001	无符号小于时置位
sltiu	1001	xxxxxx	1001	无符号小于时置位
sll	1101	000000	1010	逻辑左移运算
sllv	1101	000100	1010	逻辑左移运算
srl	1101	000010	1011	逻辑右移运算
srlv	1101	000110	1011	逻辑右移运算
sra	1101	000011	1100	算术右移运算
srav	1101	000111	1100	算术右移运算
lui	1111	xxxxxx	1111	无运算
jr	1111	001000	1111	无运算
jal	1111	xxxxxx	1111	无运算
j	1111	xxxxxx	1111	无运算

表3

ALU控制器的信号控制实现和lab5中基本一样，此处不再赘述。

信号	输入/输出	长度	说明
aluOp	input	3	运算类型解析信号
funct	input	6	指令功能码
aluCtrOut	output	4	ALU运算信号
shamtSign	output	1	移位信号

表4

与lab3类似，我们用casex语句进行分支，代码如下：

```
module ALUCtr(
    input [3:0] aluOp,
    input [5:0] func,
    output reg [3:0] aluCtrOut,
    output reg shamtSign
    );
    always @ (aluOp or func)
    begin
        casex({aluOp,func})
            // 9'b000xxxxxx://lw sw
            //     aluCtrOut = 4'b0010;
            // 9'b001xxxxxx://beq
            //     aluCtrOut = 4'b0110;
            //
            10'b0000100000:aluCtrOut=4'b0000;//add
            10'b0000100001:aluCtrOut=4'b0001;//addu
            10'b0000100010:aluCtrOut=4'b0010;//sub
            10'b0000100011:aluCtrOut=4'b0011;//subu
            10'b0000100100:aluCtrOut=4'b0100;//and
            10'b0000100101:aluCtrOut=4'b0101;//or
            10'b0000100110:aluCtrOut=4'b0110;//xor
            10'b0000100111:aluCtrOut=4'b0111;//nor
            10'b0000101010:aluCtrOut=4'b1000;//slt
            10'b0000101011:aluCtrOut=4'b1001;//sltu
            10'b0000000000:aluCtrOut=4'b1010;//sll
            10'b0000000010:aluCtrOut=4'b1011;//srl
            10'b0000000011:aluCtrOut=4'b1100;//sra
            10'b0000000100:aluCtrOut=4'b1010;//sllv
            10'b0000000110:aluCtrOut=4'b1011;//srlv
            10'b0000000111:aluCtrOut=4'b1100;//srav
            10'b1111001000:aluCtrOut=4'b1111;//jr
            10'b0010xxxxxx:aluCtrOut=4'b0000;//addi
            10'b0011xxxxxx:aluCtrOut=4'b0001;//addiu
            10'b0100xxxxxx:aluCtrOut=4'b0100;//andi
            10'b0101xxxxxx:aluCtrOut=4'b0101;//ori
            10'b0110xxxxxx:aluCtrOut=4'b0110;//xori
            10'b0111xxxxxx:aluCtrOut=4'b1111;//lui
```

```
            10'b1000xxxxxx:aluCtrOut=4'b0001;//lw,sw
            10'b1100xxxxxx:aluCtrOut=4'b0011;//beq,bne
            10'b1010xxxxxx:aluCtrOut=4'b1000;//slti
            10'b1011xxxxxx:aluCtrOut=4'b1001;//sltiu
            10'b1111xxxxxx:aluCtrOut=4'b1111;//jal,j

        endcase
        if(func==6'b000000||func==6'b000010||func==6'b000011)
        shamtSign=1;
        else shamtSign=0;
    end

endmodule
```

3. 算术逻辑运算单元ALU

算术逻辑运算单元的实现与lab5类似，我们在本次实验中加入溢出信号位，用来区分如add和addu的区别，判断在运算过程中是否有溢出现象。

其他信号控制实现与lab5一致，信号端口信息如下表5所示。

信号	输入/输出	长度	说明
input1	input	32	操作数1
input2	input	32	操作数2
aluCtr	input	4	ALU运算信号
zero	output	1	0标志信号
overflow	output	1	溢出标志信号
aluRes	output	32	ALU输出结果

表5

```
module ALU(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output zero,
    output overflow,
    output [31:0] aluRes
    );
    reg Zero;
    reg Overflow;
    reg [31:0] ALURes;
//    reg [31:0] Input1;
//    reg[31:0]Input2;
    always @ (input1 or input2 or aluCtr)
    begin
```

```
        case(aluCtr)
            4'b0000://add with overflow
            begin
                ALURes=input1+input2;
                if((input1>>31)==(input2>>31)&&(input1>>31)!=
(aluRes>>31))
                Overflow=1;
                else Overflow=0;
            end
            4'b0001:ALURes=input1+input2;//add
            4'b0010://sub with overflow
            begin
                ALURes=input1-input2;
                if((input1>>31)==((~input2+1)>>31)&&(input1>>31)!=
(aluRes>>31))
                Overflow=1;
                else Overflow=0;
            end
            4'b0011:ALURes=input1-input2;//sub
            4'b0100:ALURes=input1&input2;//and
            4'b0101:ALURes=input1|input2;//or
            4'b0110:ALURes=input1^input2;//xor
            4'b0111:ALURes=~(input1|input2);//nor
            4'b1000:ALURes=($signed(input1)<$signed(input2));//slt
            4'b1001:ALURes=(input1<input2);//unsigned slt
            4'b1010:ALURes=(input2<<input1);//left-shift
            4'b1011:ALURes=(input2>>input1);//right-shift
            4'b1100:ALURes=($signed(input2)>>>input1);
            default:ALURes=0;

        endcase
        if(ALURes==0)
            Zero=1;
        else
            Zero=0;
    end
    assign zero=Zero;
    assign overflow=Overflow;
    assign aluRes=ALURes;
//    assign input1=Input1;
//    assign input2=Input2;
endmodule
```

4. 寄存器模块Registers

寄存器模块的实现与lab5中的实现完全一致，具体参考lab5的实验报告。

5. 数据内存单元dataMemory

数据内存单元和我们在lab4中的实现完全一致，在此处不再展示具体的信号和代码。

6. 带符号扩展单元SignExt

带符号扩展单元亦和我们在lab4中的实现完全一致，此处不再展示具体的信号和代码。

7. 指令存储单元InstMemory

指令存储单元的实现和我们在lab5中的实现完全一致，在此处不再展示具体的信号和代码。

8. 多路选择器MUX实现

本实验中，多路选择器模块的实现与lab5中完全一致，不再展示相关的信号和代码实现。

在本实验中，相较于lab5,我们增添了一些数据选择器来支撑更多指令的功能实现，并且实现forward数据的选择等。选择器的具体分支效果如下表6所示。(其中的选择信号在Top顶层模块文件中以各个段寄存器所暂存的段寄存器控制信号存在，具体见下节Top顶层模块文件的实现)

多路选择器	选择信号	说明
rs_shamt_mux	EX_SHAMT_SIGN	选择是否移位操作，ALU输入1
rt_ext_mux	EX_ALU_SRC_SIG	选择ALU输出结果，ALU输入2
lui_mux	EX_LUI_SIG	选择是否是lui指令作为ALU输出
pc_jump_mux	ID_JUMP_SIGNAL	选择是否jump指令跳转
pc_jr_mux	ID_JR_SIGNAL	选择是否jr指令跳转
pc_beq_mux	EX_BEQ_SIG&EX_ALU_ZERO	选择是否beq指令跳转
pc_bne_mux	EX_BNE_SIG&(~EX_ALU_ZERO)	选择是否bne指令跳转
mem_to_reg_mux	MA_MEM_TO_REG_SIGNAL	选择返回寄存器的数据
forward_mux1	WB_REG_WRITE_SIGNAL*	选择forward数据，ALU输入1
forward_mux2	WB_REG_WRITE_SIGNAL*	选择forward数据，ALU输入2
forward_mux1_next	MA_REG_WRITE_SIGNAL*	选择forward数据，ALU输入1
forward_mux2_next	MA_REG_WRITE_SIGNAL*	选择forward数据，ALU输入2

表6

表中*部分标注此处不仅仅受WB_REG_WRITE_SIGNAL和MA_REG_WRITE_SIGNAL信号控制，这里需要进行前向通路的选择，对应的选择信号还应判断目标寄存器和所用寄存器是否相同。在这里分别有：读存储器的值送入下两条指令的EX阶段，以及ALU运算结果送入下一条指令的EX阶段执行前。更加完整的运行机制将在下面的顶层模块中进行描述。

（二）顶层模块Top

1. 流水线原理描述

MIPS处理器的执行过程我们初步分为五个阶段:取指阶段，译码阶段，执行阶段，访存阶段和写回阶段。

取指阶段(Instruction Fetch)

取指阶段主要包含指令存储器模块，从PC提供的指令输入，取出当前应该执行的指令。在这里我们把PC简化成一个寄存器通路，连接指令存储器的输入端。

```
    //IF stage
    reg[31:0] IF_PC;    //replace PC module because the update will be considered as a whole in the pipeline
    wire[31:0] IF_INST;
    InstMemory inst_mem(
        .address(IF_PC),
        .inst(IF_INST)
    );
```

译码阶段(Instruction Decode)

译码阶段主要包含主控制器模块、寄存器模块、以及符号扩展单元，以及相关的多路选择器来选择寄存器和写入数据等。

```
//ID stage
    wire[12:0] ID_CTR_SIGNALS;
    // 12:J
    // 11:JR
    // 10:EXT
    // 9:REG_DST
    // 8:JAL
    // 7:ALU_SRC
    // 6:LUI
    // 5:BEQ
    // 4:BNE
    // 3:MEM_WRITE
```

```
    // 2:MEM_READ
    // 1:MEM_TO_REG
    // 0:REG_WRITE
    //This is the sequence we will use in the next stages
    //we should use wire to realize output

    wire ID_JUMP_SIGNAL;
    wire ID_JR_SIGNAL;
    wire ID_EXT_SIGNAL;
    wire ID_REG_DST_SIGNAL;
    wire ID_JAL_SIGNAL;
    wire ID_ALU_SRC_SIGNAL;
    wire ID_LUI_SIGNAL;
    wire ID_BEQ_SIGNAL;
    wire ID_BNE_SIGNAL;
    wire ID_MEM_WRITE_SIGNAL;
    wire ID_MEM_READ_SIGNAL;
    wire ID_MEM_TO_REG_SIGNAL;
    wire ID_REG_WRITE_SIGNAL;

    wire[3:0] ID_CTR_SIGNAL_ALUOP;
    Ctr main_ctr(
        .opCode(IF2ID_INST[31:26]),
        .func(IF2ID_INST[5:0]),
        .jump(ID_JUMP_SIGNAL),
        .jrSign(ID_JR_SIGNAL),
        .extSign(ID_EXT_SIGNAL),
        .regDst(ID_REG_DST_SIGNAL),
        .jalSign(ID_JAL_SIGNAL),
        .aluSrc(ID_ALU_SRC_SIGNAL),
        .luiSign(ID_LUI_SIGNAL),
        .beqSign(ID_BEQ_SIGNAL),
        .bneSign(ID_BNE_SIGNAL),
        .memWrite(ID_MEM_WRITE_SIGNAL),
        .memRead(ID_MEM_READ_SIGNAL),
        .memToReg(ID_MEM_TO_REG_SIGNAL),
        .regWrite(ID_REG_WRITE_SIGNAL),
        .aluOp(ID_CTR_SIGNAL_ALUOP)
    );

    assign ID_CTR_SIGNALS[12]=ID_JUMP_SIGNAL;
    assign ID_CTR_SIGNALS[11]=ID_JR_SIGNAL;
    assign ID_CTR_SIGNALS[10]=ID_EXT_SIGNAL;
    assign ID_CTR_SIGNALS[9]=ID_REG_DST_SIGNAL;
    assign ID_CTR_SIGNALS[8]=ID_JAL_SIGNAL;
    assign ID_CTR_SIGNALS[7]=ID_ALU_SRC_SIGNAL;
    assign ID_CTR_SIGNALS[6]=ID_LUI_SIGNAL;
    assign ID_CTR_SIGNALS[5]=ID_BEQ_SIGNAL;
```

```
    assign ID_CTR_SIGNALS[4]=ID_BNE_SIGNAL;
    assign ID_CTR_SIGNALS[3]=ID_MEM_WRITE_SIGNAL;
    assign ID_CTR_SIGNALS[2]=ID_MEM_READ_SIGNAL;
    assign ID_CTR_SIGNALS[1]=ID_MEM_TO_REG_SIGNAL;
    assign ID_CTR_SIGNALS[0]=ID_REG_WRITE_SIGNAL;

    wire[31:0] ID_REG_READ_DATA1;
    wire[31:0] ID_REG_READ_DATA2;
    wire[4:0] WB_REG_WRITE_ID;
    //wire[4:0] WB_REG_WRITE_SELECT;//after the mux
    wire[31:0] WB_REG_WRITE_DATA;
    //wire[31:0] WB_REG_WRITE_DATA_SELECT;//after the mux
    wire WB_REG_WRITE_SIGNAL;

    wire[4:0] ID_REG_DEST;
     wire[4:0] ID_REG_RS=IF2ID_INST[25:21];//RS
     wire[4:0] ID_REG_RT=IF2ID_INST[20:16];//RT
     wire[4:0] ID_REG_RD=IF2ID_INST[15:11];//RD
    Mux_s rt_rd_mux(
        .selectSignal(ID_REG_DST_SIGNAL),
        .input1(IF2ID_INST[15:11]),
        .input2(IF2ID_INST[20:16]),
        .out(ID_REG_DEST)
    );
    Registers reg_file(
        .readReg1(IF2ID_INST[25:21]),
        .readReg2(IF2ID_INST[20:16]),
        .writeReg(WB_REG_WRITE_ID),
        .writeData(WB_REG_WRITE_DATA),
        .regWrite(WB_REG_WRITE_SIGNAL),
        .jalSign(ID_JAL_SIGNAL),
        .jalData(IF2ID_PC+4),
        .clk(clk),
        .reset(reset),
        .readData1(ID_REG_READ_DATA1),
        .readData2(ID_REG_READ_DATA2)
    );
    wire[31:0] ID_EXT_RES;
    signext sign_ext(
        .inst(IF2ID_INST[15:0]),
        .extSign(ID_EXT_SIGNAL),
        .data(ID_EXT_RES)
    );
```

执行阶段(Execution)

执行阶段主要包括ALU控制单元、算术逻辑运算单元以及一些数据多路选择器来选择ALU的输入输出结果，支持指令中对应的运算。

同时，我在这个阶段加入了分支选择和跳转的设计，整体设计参照lab5的实现，在这里加入bne指令的处理，思路是相同的。

```
    //EX stage

    wire EX_ALU_SRC_SIG=ID2EX_CTR_SIGNALS[7];
    wire EX_LUI_SIG=ID2EX_CTR_SIGNALS[6];
    wire EX_BEQ_SIG=ID2EX_CTR_SIGNALS[5];
    wire EX_BNE_SIG=ID2EX_CTR_SIGNALS[4];
    wire EX_SHAMT_SIGNAL;
    wire[3:0] EX_ALU_CTR_OUT;

    wire[31:0] EX_ALU_INPUT1;
    wire[31:0] EX_ALU_INPUT2;
    wire[31:0] EX_ALU_RES;
    wire[31:0] EX_ALU_RES_LUI;
    wire EX_ALU_ZERO;
    wire EX_ALU_OVERFLOW;

    ALUCtr alu_ctr(
        .aluOp(ID2EX_ALUOP),
        .func(ID2EX_FUNC),
        .shamtSign(EX_SHAMT_SIGNAL),
        .aluCtrOut(EX_ALU_CTR_OUT)
    );


    wire[31:0] EX_ALU_INPUT1_FORWADING;
    wire[31:0] EX_ALU_INPUT2_FORWADING;

    Mux rs_shamt_mux(
        .selectSignal(EX_SHAMT_SIGNAL),
        .input1({27'b0,ID2EX_SHAMT}),
        .input2(EX_ALU_INPUT1_FORWADING),
        .out(EX_ALU_INPUT1)
    );//FOR INPUT1

    Mux rt_ext_mux(
        .selectSignal(EX_ALU_SRC_SIG),
        .input1(ID2EX_EXT_RES),
        .input2(EX_ALU_INPUT2_FORWADING),
        .out(EX_ALU_INPUT2)
    );//FOR INPUT2
```

```
41
42      ALU alu(
43          .input1(EX_ALU_INPUT1),
44          .input2(EX_ALU_INPUT2),
45          .aluCtr(EX_ALU_CTR_OUT),
46          .aluRes(EX_ALU_RES),
47          .zero(EX_ALU_ZERO),
48          .overflow(EX_ALU_OVERFLOW)
49      );
50
51      Mux lui_mux(
52          .selectSignal(EX_LUI_SIG),
53          .input1({ID2EX_EXT_RES[15:0],16'b0}),
54          .input2(EX_ALU_RES),
55          .out(EX_ALU_RES_LUI)
56      );
57
58      wire[31:0] BRANCH_DT=ID2EX_PC+4+(ID2EX_EXT_RES<<2);
59
60      //jump or branch, updating pc
61      //we set the sequence jump->jr->beq->bne
62      wire[31:0] PC_JUMP;
63      wire[31:0] PC_JR;
64      wire[31:0] PC_BEQ;
65      wire[31:0] PC_BNE;
66
67      Mux pc_jump_mux(
68          .selectSignal(ID_JUMP_SIGNAL),
69          .input1(((IF2ID_PC+4)&32'hf0000000)+(IF2ID_INST[25:0]<<2)),
70          .input2(IF_PC+4),
71          .out(PC_JUMP)
72      );
73      Mux pc_jr_mux(
74          .selectSignal(ID_JR_SIGNAL),
75          .input1(ID_REG_READ_DATA1),
76          .input2(PC_JUMP),
77          .out(PC_JR)
78      );
79      Mux pc_beq_mux(
80          .selectSignal(EX_BEQ_SIG&EX_ALU_ZERO),
81          .input1(BRANCH_DT),
82          .input2(PC_JR),
83          .out(PC_BEQ)
84      );
85      Mux pc_bne_mux(
86          .selectSignal(EX_BNE_SIG&(~EX_ALU_ZERO)),
87          .input1(BRANCH_DT),
88          .input2(PC_BEQ),
```

```
89          .out(PC_BNE)
90      );
91      wire[31:0] PC_NEXT=PC_BNE;
```

访存阶段(Memory Access)

访存阶段主要包括存储器以及一个多路选择器选择访存结果和ALU运算结果，这个阶段主要是进行数据存储器的访问。

```
1       //MA stage
2       wire[31:0] MA_MEM_READ_DATA;
3       dataMemory data_mem(
4           .clk(clk),
5           .address(EX2MA_ALU_RES),
6           .writeData(EX2MA_REG_READ_DATA2),
7           .memWrite(MA_MEM_WRITE_SIGNAL),
8           .memRead(MA_MEM_READ_SIGNAL),
9           .readData(MA_MEM_READ_DATA)
10      );
11
12      wire[31:0] MA_MEM_DATA_OUT;
13      Mux mem_to_reg_mux(
14          .selectSignal(MA_MEM_TO_REG_SIGNAL),
15          .input1(MA_MEM_READ_DATA),
16          .input2(EX2MA_ALU_RES),
17          .out(MA_MEM_DATA_OUT)
18      );
```

写回阶段(Write Back)

写回阶段主要包括对寄存器的写回，我们将写回安排在时钟下降沿，一定程度解决了一些数据冒险。

```
1       //connect it to the registers
2       assign WB_REG_WRITE_ID=MA2WB_REG_WRITE;
3       assign WB_REG_WRITE_DATA=MA2WB_DATA_OUT;
4       assign WB_REG_WRITE_SIGNAL=MA2WB_CTR_SIGNALS;
```

2. 流水线寄存器

由于我们需要支持指令的并行，所以我们要在流水线的两个阶段之间设置一系列的段寄存器，用来保存指令在上一阶段执行的结果。

- IF2ID段寄存器：包含当前指令及其PC
- ID2EX段寄存器：包含各个控制信号、符号扩展单元的结果、解析出的指令中rs、rt、func、shamt的值、读取寄存器的结果、以及当前指令PC和目标寄存器的地址
- EX2MA段寄存器：包含一部分控制信号、ALU的运算结果、rt寄存器的值以及目标寄存器的地址
- MA2WB段寄存器：包含写回信号、最终写寄存器的数据以及目标寄存器的地址

相关代码如下：

```
    //Segment registers IF to ID
    reg[31:0] IF2ID_PC;
    reg[31:0] IF2ID_INST;

    //Segment registers ID to EX
    reg[3:0] ID2EX_ALUOP;
    reg[7:0] ID2EX_CTR_SIGNALS;
    reg[4:0] ID2EX_INST_RS;
    reg[4:0] ID2EX_INST_RT;
    reg[31:0] ID2EX_REG_READ_DATA1;
    reg[31:0] ID2EX_REG_READ_DATA2;
    reg[4:0] ID2EX_REG_DEST;
    reg[31:0] ID2EX_EXT_RES;
    reg[4:0] ID2EX_SHAMT;
    reg[5:0] ID2EX_FUNC;
    reg[31:0] ID2EX_PC;

    //Segment registers EX to MA
    reg[3:0] EX2MA_CTR_SIGNALS;
    reg[31:0] EX2MA_ALU_RES;
    reg[31:0] EX2MA_REG_READ_DATA2;
    reg[4:0] EX2MA_REG_DEST;

    //Segment registers MA to WB
    reg MA2WB_CTR_SIGNALS;
    reg[31:0] MA2WB_DATA_OUT;
    reg[4:0] MA2WB_REG_WRITE;
```

3. 前向通路forward机制

当前指令处于EX阶段时，运算输入可能来自先前指令的写回结果。进行前向传递需要有以下条件：先前指令需要进行寄存器写入操作且先前指令写入的目标寄存器与当前指令的读取寄存器相同。对于两条前向通路，我们分别1需要对rs和rt进行选择，总共需要4个多路选择器。

具体的forward代码实现如下：

```
//FORWARDING
    wire[31:0] EX_FORWARING1_TMP;
    wire[31:0] EX_FORWARING2_TMP;
    Mux forward_mux1(
        .selectSignal(WB_REG_WRITE_SIGNAL&(MA2WB_REG_WRITE==ID2EX_INST_RS)),
        .input1(MA2WB_DATA_OUT),
        .input2(ID2EX_REG_READ_DATA1),
        .out(EX_FORWARING1_TMP)
    );
    Mux forward_mux2(
        .selectSignal(WB_REG_WRITE_SIGNAL&(MA2WB_REG_WRITE==ID2EX_INST_RT)),
        .input1(MA2WB_DATA_OUT),
        .input2(ID2EX_REG_READ_DATA2),
        .out(EX_FORWARING2_TMP)
    );
    Mux forward_mux1_next(
        .selectSignal(MA_REG_WRITE_SIGNAL&(EX2MA_REG_DEST==ID2EX_INST_RS)),
        .input1(EX2MA_ALU_RES),
        .input2(EX_FORWARING1_TMP),
        .out(EX_ALU_INPUT1_FORWADING)
    );
    Mux forward_mux2_next(
        .selectSignal(MA_REG_WRITE_SIGNAL&(EX2MA_REG_DEST==ID2EX_INST_RT)),
        .input1(EX2MA_ALU_RES),
        .input2(EX_FORWARING2_TMP),
        .out(EX_ALU_INPUT2_FORWADING)
    );
```

4. STALL与BRANCH机制

在流水线的设计中，我们会遇到数据冒险，当且仅当产生了访存使用的数据冒险，此时我们需要将流水线进行停顿，我们用STALL信号来检测流水线中是否存在冒险。

```
wire STALL=ID2EX_CTR_SIGNALS[2]&((ID2EX_INST_RT==IF2ID_INST[25:21])|(ID2EX_INST_RT==IF2ID_INST[20:16]));//memRead信号
```

有时候我们也会遇到跳转这样的特殊情况如beq和bne指令，在这里我们采用预测错误则清楚相关指令的操作，并得到正确的地址继续执行。转移错误在EX阶段执行后来判断，最多会影响之后的两个指令。

我们用BRANCH信号来判断是否预测错误，如果是则进行流水线后续指令的清空，设计如下：

```
wire BRANCH=(EX_BEQ_SIG&EX_ALU_ZERO)|(EX_BNE_SIG&(~EX_ALU_ZERO));
```

5. 流水线时序实现代码

```
    initial IF_PC=0;
    always @(reset)
    begin
        if(reset)begin
            IF_PC=0;
            IF2ID_INST=0;
            IF2ID_PC=0;
            ID2EX_ALUOP=0;
            ID2EX_CTR_SIGNALS=0;
            ID2EX_EXT_RES=0;
            ID2EX_FUNC=0;
            ID2EX_INST_RS=0;
            ID2EX_INST_RT=0;
            ID2EX_REG_DEST=0;
            ID2EX_REG_READ_DATA1=0;
            ID2EX_REG_READ_DATA2=0;
            ID2EX_SHAMT=0;
            EX2MA_ALU_RES=0;
            EX2MA_CTR_SIGNALS=0;
            EX2MA_REG_DEST=0;
            EX2MA_REG_READ_DATA2=0;
            MA2WB_CTR_SIGNALS=0;
            MA2WB_DATA_OUT=0;
            MA2WB_REG_WRITE=0;
        end
    end

    always @(posedge clk)
    begin
        //if-id
        if(!STALL)
            IF_PC<=PC_BNE;
```

```
33          if(BRANCH||ID_CTR_SIGNALS[12]||ID_CTR_SIGNALS[11])begin
34              IF2ID_INST<=0;
35              IF2ID_PC<=0;
36          end else if(!STALL)begin
37              IF2ID_INST<=IF_INST;
38              IF2ID_PC<=IF_PC;
39          end
40          //id-ex
41              if(STALL||BRANCH)begin
42                  ID2EX_PC<=IF2ID_PC;
43                  ID2EX_ALUOP<=4'b1111;
44                  ID2EX_CTR_SIGNALS<=0;
45                  ID2EX_EXT_RES<=0;
46                  ID2EX_INST_RS<=0;
47                  ID2EX_INST_RT<=0;
48                  ID2EX_REG_READ_DATA1<=0;
49                  ID2EX_REG_READ_DATA2<=0;
50                  ID2EX_FUNC<=0;
51                  ID2EX_SHAMT<=0;
52                  ID2EX_REG_DEST<=0;
53              end else begin
54                  ID2EX_PC<=IF2ID_PC;
55                  ID2EX_ALUOP<=ID_CTR_SIGNAL_ALUOP;
56                  ID2EX_CTR_SIGNALS<=ID_CTR_SIGNALS[7:0];
57                  ID2EX_EXT_RES<=ID_EXT_RES;
58                  ID2EX_INST_RS<=IF2ID_INST[25:21];
59                  ID2EX_INST_RT<=IF2ID_INST[20:16];
60                  ID2EX_REG_READ_DATA1<=ID_REG_READ_DATA1;
61                  ID2EX_REG_READ_DATA2<=ID_REG_READ_DATA2;
62                  ID2EX_FUNC<=IF2ID_INST[5:0];
63                  ID2EX_SHAMT<=IF2ID_INST[10:6];
64                  ID2EX_REG_DEST<=ID_REG_DEST;
65              end
66
67          //ex-ma          EX2MA_CTR_SIGNALS<=ID2EX_CTR_SIGNALS[3:0];
68              EX2MA_ALU_RES<=EX_ALU_RES_LUI;
69              EX2MA_REG_READ_DATA2<=EX_ALU_INPUT2_FORWARDING;
70              EX2MA_REG_DEST<=ID2EX_REG_DEST;
71
72              MA2WB_CTR_SIGNALS<=EX2MA_CTR_SIGNALS[0];
73              MA2WB_DATA_OUT<=MA_MEM_DATA_OUT;
74              MA2WB_REG_WRITE<=EX2MA_REG_DEST;
75      end
76  endmodule
```

三、实验结果验证

（一）汇编代码设计与编写

为了测试我们所编写的MIPS流水线处理器能否正常工作并处理好并行，我编写了一些MIPS指令来验证。

测试用的代码如下：

```
1  00001000000000000000000000000100    j
2  00000000000000000000000000000000    nop
3  00000000000000000000000000000000    nop
4  00000000000000000000000000000000    nop
5  00001100000000000000000000000110    jal
6  00000000000000000000000000000000    nop
7  10001100000000010000000000000010    lw 1,2(0)   $1:2
8  10001100000000010000000000000110    lw 2,6(0)   $2:6
9  00000000001000100001100000100000    add 3,1,2   $3:8
10 00000000010000010010000000100010    sub 4,2,1   $4:4
11 00000000001001000001000000100010    sub 2,1,4   $2:-2
12 00000000001001000001100000100100    and 3,1,4   $3:0
13 00000000000000010000100001000000    sll 1,1,1   $1:4
14 00100000000000010000000000000010    addi 1,0,2  $1:2
15 00110100100000100000000000000011    ori 2,4,3   $2:7
16 00000000001000100001100000101010    slt 3,1,2   $3:1
17 00111100000001001111111111111111    lui 4,65535 $4:-65536
18 00100100000000010000000000001111    addiu 1,0,15  $1:15
19 00000000000000010010100111000000    sll 5,1,7   $5:1920
20 00000000011001010001100000100001    addu 3,3,5  $3:1921
21 00000000011000010001100000100110    xor 3,3,1   $3:1934
22 00000000000001000010001000000011    sra 4,4,8   $4:-256
23 00101000010000100000000000000001    slti 2,2,1  $2:0
24 00110100001000100000000000011110    ori 2,1,30  $2:31
25 00000000010000010001000000100011    subu 2,2,1  $2:16
26 10101100000000100000000000000010    sw 2,2(0)
27 00000000000000010000100011000010    srl 1,2,3   $1:2
28 00010000101000000000000000000001    beq 5,0,1
29 00010000101000000000000000000001    beq 5,0,1
30 00010100101000000000000000000001    bne 5,0,1
31 10101100000000101000000000000010    sw 2,2(0)
32 00000011111100000000000000001000    jr 31
```

内存的初始值如下：

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
```

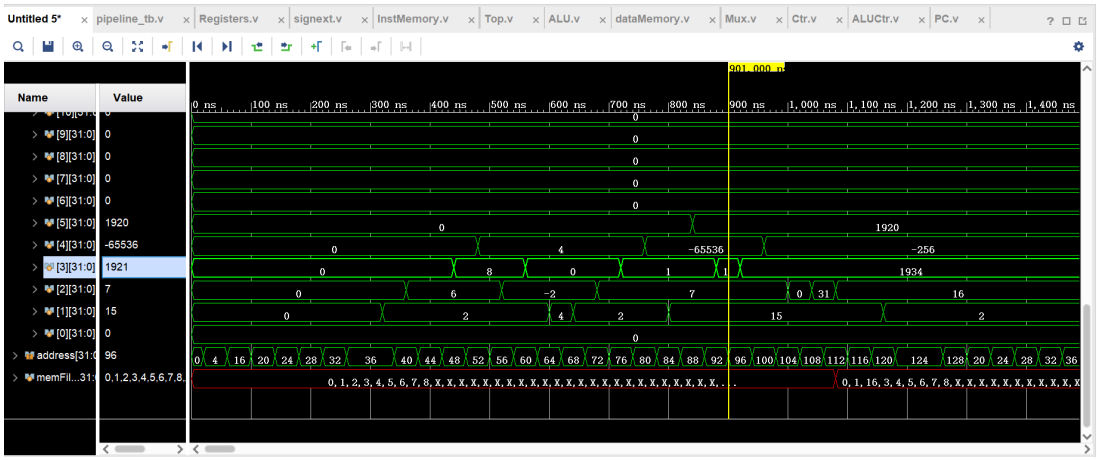
将内存的初始值写在data.dat中，将数据写在inst_data.dat中。

（二）仿真测试代码

激励文件代码如下：

```
module pipeline_tb(

    );
     reg clk;
    reg reset;
    Top processor(.clk(clk),.reset(reset));

    initial begin

 $readmemb("D:/C/Arch/lab6/inst_data.dat",processor.inst_mem.instFile);

 $readmemh("D:/C/Arch/lab6/data.dat",processor.data_mem.memFile);
        reset=1;
        clk=0;
    end

    always #20 clk=~clk;
    initial begin
        #40 reset=0;
        #1000;
        $finish;
    end
endmodule
```

（三）仿真波形



如图1所示，波形展示了寄存器和内存相关内容的变化，我们可以验证得完全符合我的测试指令，于是我们的流水线编写正确。

四、实验心得

本次实验难度空前大，比上个星期苦苦完成的lab5还要大很多。lab5完成了一个MIPS单周期处理器，在此基础之上lab6要将其改进成流水线的处理器。在刚开始做实验的时候我完全无法下手，我脑海中仅存的只有一些段寄存器的添加和线路的简单连接，对于一些特殊处理机制如forward和stall等等我还是一头雾水。

于是我先模仿着添加段寄存器，在此过程中我把lab5里相关的连线内容全部更新过了，重新命名因为lab6的连线实在太多。在这个阶段我就已经停留了很长时间因为有时在不同的阶段会用到不同的硬件，阶段直接也有联系。为了让自己的思路更清晰，我把每一个段寄存器的出口都添加了命名比较显眼的wire，来方便我在这一阶段的连线。

后期我上网搜寻有关资料，参考了一个实现停顿和预测的比较简单的方式，用信号判断特殊情况，从而在后期连流水线的代码做出相应的处理。同时我也加入了forward前向传递机制，这一部分比我想象的要简单，其实就是比lab5增加了一些连线。

最后我编写了整个流水线时序实现的代码，这个过程比较繁琐因为我前面的实现比较混杂，导致初期写的时候怎么调试都无法出现正确的结果。于是我参考了网上一些比较通俗易懂的时序连接方式自己结合自己的设计从头重新编写了一下终于成功了，这一过程特别不容易。

在本次实验中，我对流水线处理器结构有了更深一步的了解，在添加前向通路，停顿等机制的过程中我同时完善了自己对这部分理论知识的模糊点与不足，很有收获。在调试过程中我对Verilog的语法愈加熟练，与几星期前自己懵懂对照实验指导书完成lab1的我相比，实是感慨万千。

在完成报告的前一天，我又自己编写了这个MIPS指令测试集来验证我的设计。很不幸，出现了一些错误。我运用控制变量法和排除法在经过漫长的调试中终于将问题锁定在ALU,ALUCtr和Ctr模块中。由于本次实验我是直接从31条指令自己独立编码，第一次尝试的时候出现了一些错误，导致部分运算结果不正确。于是我又认真学习了MIPS指令的相关编码，发现了一些规律，比如从ALUCtr先对R型指令进行编码后对其他指令进行对应，这样终于解决了大部分问题。最后一个问题也让我印象深刻，就是func|opcode作为Ctr分支语句的条件，而不是仅有opcode。

总之，本次实验的过程非常漫长而艰难，但是在一步一步探索的过程中真的收获了很多。

最后我向计算机系统结构实验课程的黄小平老师和王莉老师与助教，以及在实验过程中指导过我的学长和同学表示感谢。

五、参考资料

2022计算机系统结构实验指导书lab6