计算机系统结构实验报告 实验六

惠宇龙 518030910059

1 实验概述

1.1 实验描述

简单的类 MIPS 多周期流水化处理器实现。

特别地，由于实验要求1-4是分级逐步完善流水线CPU，因此我们将其全部完成后，在本实验报告中统一呈现。

1.2 实验目的

（本实验共完成了5个板块，4个必做+1个选做（5））

- 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
- 通过predict-not-taken策略，进一步提高处理器性能

2 部件的设计与实现

有了Lab5的基础，我们不需要对部件实现代码进行过多的修改。

2.1 各部件之间的联系

流水线处理器的设计较为复杂，各部件之间遵从如下所示的联系：

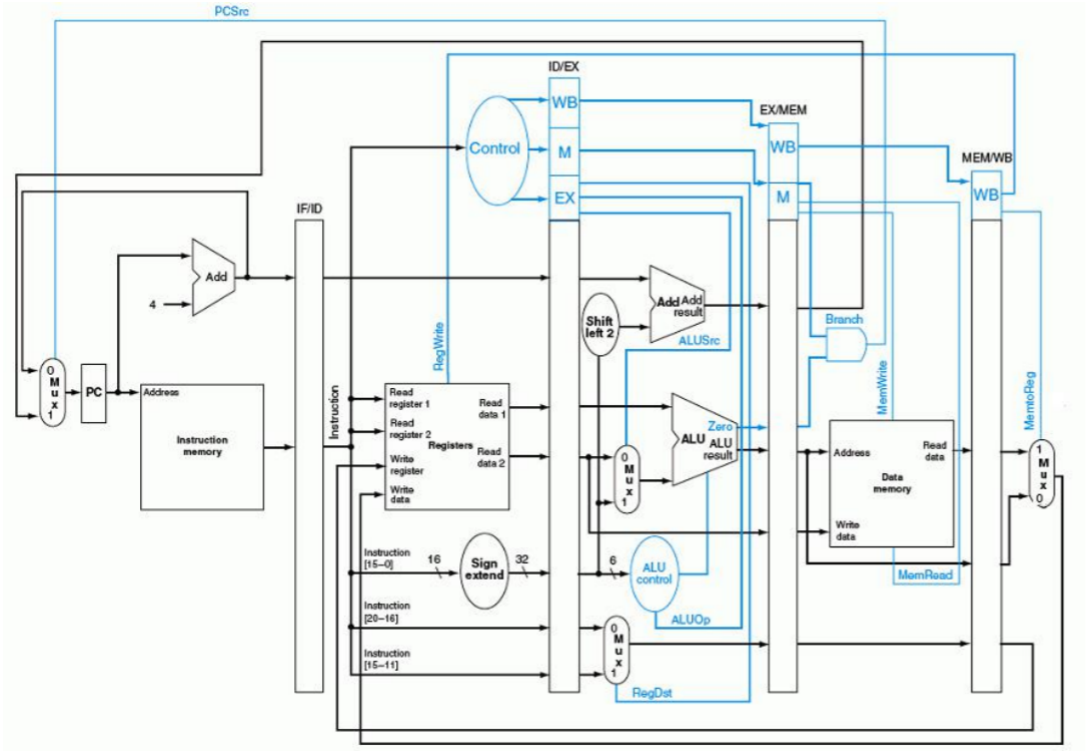


图1.流水线处理器的原理图

2.2 实现流水线寄存器

流水线设计与单周期处理器最大的差别之一，就是流水线处理器需要4个寄存器来储存当前指令的执行状态。

其原理以及所需存储的数据以及在图1中说明，具体的实现代码如下：

```
 // IF/ID
reg  [31:0] IFID_pctmp, IFID_inst;
wire [4:0]  IFID_rs = IFID_inst[25:21], IFID_rt = IFID_inst[20:16],
IFID_rd = IFID_inst[15:11];
wire STALL;

// ID/EX
reg [31:0] IDEX_readData1, IDEX_readData2, IDEX_immData;
reg [31:0] IDEX_pctmp;
reg [4:0] IDEX_rs, IDEX_rt, IDEX_rd;
reg [9:0] IDEX_ctrl;
reg IDEX_JUMP;

wire[2:0] IDEX_ALUOP = IDEX_ctrl[8:6];
wire IDEX_RegDst = IDEX_ctrl[9], 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];
wire PCsrc;
```

```
wire [31:0] Finaladdr;

// EX/MEM
reg [31:0] EXMEM_ALUres, EXMEM_writeData;
reg [4:0] EXMEM_dstReg;
reg [4:0] EXMEM_ctrl;
reg EXMEM_zero;
wire 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];
```

需要说明的是，上述代码中处了reg，还定义了一些wire元件，它们用来显示地说明reg中每一个端口的含义，并将reg的数据与其他的部件相连通。

3 流水线分阶段实现

开始之前，由于多条指令可能需要在流水线的不同阶段并行执行，为了避免冲突，我们首先需要对存储器和寄存器的读写进行同步。

考虑到我们的时钟信号如图所示： 

我们规定，在时钟的上升沿对寄存器组和数据存储器进行写入，在时钟的下降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（高电平）时写入，在后半个周期（低电平）读取。由于读写操作有时会在同一个周期内发生，这种在前半周期写，后半个周期读的安排可以避免很多异常的时序错误。

3.1 取值阶段

我们选择在上升沿（每个周期开始时）更新PC状态。在下降沿时，将根据PC值读取到的指令写入IFID的指令寄存器。当出现reset信号时，我们需要将PC和寄存器统一清零。

下列代码中已经考虑了最终实现时stall的情况（stall置1的条件后续介绍），如果注释掉stall相关的语句，就是任务1所要求的最基本的流水线。

```
//IF stage
 reg [31:0] PC,PC_PLUS_4;
 wire [31:0] PC_NEXT;
 wire [31:0] IF_INST;
 wire Jr; //用来判断是否是Jr指令，需要更新PC

  InstMemory im0(
 .readaddress(PC),
 .readinst(IF_INST)
 );


  assign PC_NEXT = Jr? IDEX_readData1:PCsrc?  Finaladdr: PC_PLUS_4;
```

```
 always @ (reset or posedge Clk)
 begin
     if(reset)
      begin
      PC=0;
      PC_PLUS_4=0;
     end
     else if(Clk)
     if (!STALL)
     begin
         PC = PC_NEXT;
         PC_PLUS_4 = PC + 4;
     end
     if (PCsrc)
       IFID_inst = 0; // flush
 end

always @ (reset or negedge Clk)
if (reset)
     begin
     IFID_pctmp <= 0;
     IFID_inst<= 0;
     end
else
     begin
     IFID_inst <= IF_INST;
     IFID_pctmp<=PC_PLUS_4;
     end
```

3.2 译码、读寄存器阶段

译码和寄存器的读取操作相对简单，只需实例化先前定义好的主控制器和寄存器。

```
 //ID
 wire [9:0] IDEX_CTRL;
 wire Jump;
ctr ctr0(
 .opcode(IFID_inst[31:26]),
 .aluop(IDEX_CTRL[8:6]),
 .branch(IDEX_CTRL[4]),
 .jump(Jump),
 .alusrc(IDEX_CTRL[5]),
 .memwrite(IDEX_CTRL[2]),
 .regwrite(IDEX_CTRL[1]),
 .memtoreg(IDEX_CTRL[0]),
 .memread(IDEX_CTRL[3]),
 .regdst(IDEX_CTRL[9])
 );

  wire [31:0] WriteDataR;
  wire [4:0] WriteReg;
  wire [31:0] IDEX_READDATA1, IDEX_READDATA2;
  wire RegWrite;
  Reg reg0(
 .Clk(Clk),
 .readReg1(IFID_rs),
```

```
    .readReg2(IFID_rt),
    .writeReg(WriteReg),
    .writeData(WriteDataR),
    .regwrite(RegWrite),
    .readData1(IDEX_READDATA1),
    .readData2(IDEX_READDATA2),
    .reset(reset)
    );

    wire [31:0] IDEX_IMMDATA;
    sign sextend0(
    .inst(IFID_inst [15:0]),
    .data(IDEX_IMMDATA)
    );

    always @ (reset or negedge Clk )
   if (reset)
        begin
        IDEX_rs <= 0;
        IDEX_readData1 <= 0;
        IDEX_readData2 <= 0;
        IDEX_immData <= 0;
        IDEX_rt <= 0;
        IDEX_rd <= 0;
        IDEX_ctrl <= 0;
        IDEX_JUMP<=0;
        end
    else
        begin
        IDEX_rs<=IFID_rs;
        IDEX_rt<=IFID_rt;
        IDEX_rd<=IFID_rd;
        IDEX_pctmp<=IFID_pctmp;
        IDEX_ctrl<= STALL ? 0:IDEX_CTRL; //SET ONE STALL
        IDEX_readData2<=IDEX_READDATA2;
        IDEX_readData1<=IDEX_READDATA1;
        IDEX_immData<=IDEX_IMMDATA;
        IDEX_JUMP<= Jump;
        end
```

3.3 运算执行阶段

下列代码中FWD相关的语句，对应着对forwarding 条件的处理，如果去除掉相关的语句，那么这就是任务一里要求实现的最基础流水线。

我们用input1和input2代表输入ALU的信号，用tmp1和tmp2代表经过预处理的输入信号。tmp1表示：是要选择正常的寄存器数据data1，还是要处理sll,srl指令而选择shamt数据。tmp2表示：是要选择正常的寄存器数据data2还是选择经过扩展的立即数。input1和input2则是根据forwarding条件, 在尚未更新的中间数据和之前读取的预处理数据之间选择。（关于forwarding条件的判断，后续会进行介绍）

为了能够提升系统的性能，将分支语句的buble减少到一个，我们将branch前移至ID和EX状态的交界处，而不再是MEM状态下解决。可以看到，我们使用EXMEM_ZERO 线路来判断是否跳转，而不是用流水线寄存器的结果进行判断。这是因为流水线寄存器需要等待写入，而wire可以在刚刚计算出结果时就立即帮助我们做出判断。

```
  wire [3:0] ALUctr;
  wire shiftmux;
  wire Jal;//用来判断是否需要更新Reg

  aluctr aluctr0(
    .ALUOp(IDEX_ALUOP),
    .Funct(IDEX_immData[5:0]),
    .Oper(ALUctr),
    .shiftmux(shiftmux),
    .Jal(Jal),
    .Jr(Jr)
    );

    wire[31:0] tmp1,tmp2,input1, input2;
    assign tmp2= IDEX_ALUSrc ? IDEX_immData :IDEX_readData2;
    assign tmp1= shiftmux? IDEX_immData[10:6] : IDEX_readData1;//for sll
    assign input1= FWD_EX_1? EXMEM_ALUres : FWD_MEM_1? WriteDataR: tmp1;
    assign input2= FWD_EX_2? EXMEM_ALUres : FWD_MEM_2? WriteDataR: tmp2;

   wire EXMEM_ZERO;
   wire [31:0] EXMEM_ALURES;
    alu alu0(
    .input1(input1),
    .input2(input2),
    .aluCtr(ALUctr),
    .zero(EXMEM_ZERO),
    .aluRes(EXMEM_ALURES)
    );

    assign PCsrc=(IDEX_Branch && EXMEM_ZERO)|| IDEX_JUMP ;
    assign Branchaddr=IDEX_pctmp+(IDEX_immData<<2);
    assign Jumpaddr[31:28]=IDEX_pctmp[31:28];
    assign Jumpaddr[27:23]=IDEX_rs;
    assign Jumpaddr[22:18]=IDEX_rt;
    assign Jumpaddr[17:2]=IDEX_immData;
    assign Jumpaddr[1:0]=0;
    assign Finaladdr=IDEX_JUMP ? Jumpaddr :Branchaddr;

   always @ (reset or negedge Clk)
   if (reset)
    begin
    EXMEM_ALUres <= 0;
    EXMEM_writeData <= 0;
    EXMEM_dstReg <= 0;
    EXMEM_ctrl <= 0;
    EXMEM_zero <= 0;
    end
   else
    begin
    EXMEM_writeData<=FWD_EX_2? EXMEM_ALUres : FWD_MEM_2?
MEMWB_readData:IDEX_readData2;
    EXMEM_dstReg<= Jal? 31:IDEX_RegDst?  IDEX_rd:IDEX_rt;
    EXMEM_ctrl<=IDEX_ctrl[4:0];
    EXMEM_ALUres<=Jal? IDEX_pctmp: EXMEM_ALURES;
    EXMEM_zero<=EXMEM_ZERO;
    end
```

3.4 访问存储器阶段

该阶段操作较为简单，只需将DataMemory实例化，并且在写入数据时设置reset对应的代码块即可。

```
//MEM stage
   wire [31:0] MEMWB_READDATA;
    DM dm0(
    .clk(clk),
    .address(EXMEM_ALUres),
    .writeData(EXMEM_writeData),
    .memwrite(EXMEM_MemWrite),
    .memread(EXMEM_MemRead),
    .readData(MEMWB_READDATA)
    );


    always @ (reset or negedge clk)
    if(reset)
        begin
        MEMWB_readData <= 0;
        MEMWB_ALUres <= 0;
        MEMWB_dstReg <= 0;
        MEMWB_ctrl <= 0;
        end
    else
        begin
        MEMWB_ALUres<=EXMEM_ALUres;
        MEMWB_dstReg<=EXMEM_dstReg;
        MEMWB_ctrl<=EXMEM_ctrl[1:0];
        MEMWB_readData<=MEMWB_READDATA;
    end
```

3.5 写入寄存器阶段

根据几个控制信号选择写回的数据即可

```
 //WB
   assign WriteDataR= MEMWB_MemtoReg ? MEMWB_readData : MEMWB_ALUres;
   assign RegWrite=MEMWB_RegWrite;
   assign WriteReg=MEMWB_dstReg;
```

4 停顿、转发、分支预测的设计与实现

4.1 停顿（stall）的设计实现

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：

```
assign STALL = IDEX_MemRead && (IDEX_rt == IFID_rs || IDEX_rt == IFID_rt);
```

STALL会带来的影响在上述分阶段实现中已经呈现。如果出现stall, 我们会使PC的更新停滞一轮，并且清空存储当前指令的控制信号的寄存器（IDEX_ctrl）。这样，当前指令无法被执行，需要下一周期才能重新执行，等效于一次停顿。

4.2 转发（forwarding）的设计实现

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。被转发数据的来源，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。

因此我们需要定义四个变量分别表示：来源于执行阶段还是访存阶段，转发给第一个操作数还是第二个操作数。特别地，在检测MEM级的forwarding条件时，需要注意：如果此时EX级的forwarding条件也成立，那么应该选择EX的执行数据，而不是MEM的访存数据。因为lw指令会在R指令之前发生，故R指令产生的结果才是最新的结果。

```
//forward unit
 wire FWD_EX_1 = EXMEM_RegWrite & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_rs;
 wire FWD_EX_2 = EXMEM_RegWrite & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_rt;
 wire FWD_MEM_1 = MEMWB_RegWrite & MEMWB_dstReg != 0
   & !(EXMEM_RegWrite & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_rs)
   & MEMWB_dstReg == IDEX_rs;
 wire FWD_MEM_2 = MEMWB_RegWrite & MEMWB_dstReg != 0
   //lw在R 之前发生，需要排除。
   & !(EXMEM_RegWrite & EXMEM_dstReg != 0 & EXMEM_dstReg == IDEX_rt)
   & MEMWB_dstReg == IDEX_rt;
```

4.3 分支预测（predict-not-taken）的设计实现

该部分的实现并不需要额外的模块或元件，只需要简单的逻辑设置即可。

当branch尚未计算出结果时，我们不要设置stall而是让PC继续更新。如果返回了branch的结果，需要跳变，则flush掉已经写入IFID寄存器的数据，将其置零，避免继续执行这一错误的指令。

```
always @ (reset or posedge Clk)
   begin
        if(reset)
        begin
        PC=0;
        PC_PLUS_4=0;
        end
       else if(Clk)
       if (!STALL)
       begin
           PC = PC_NEXT;
           PC_PLUS_4 = PC + 4;
       end

       if (PCsrc)
         IFID_inst = 0; // flush
   end
```

5 结果验证

5.1 通过基础流水线进行的软法检测

因为是软法仿真，我们在尚未设置stall和转发等功能的基础流水线上进行。通过插入nop指令，调整指令间的先后顺序，来消除hazard。

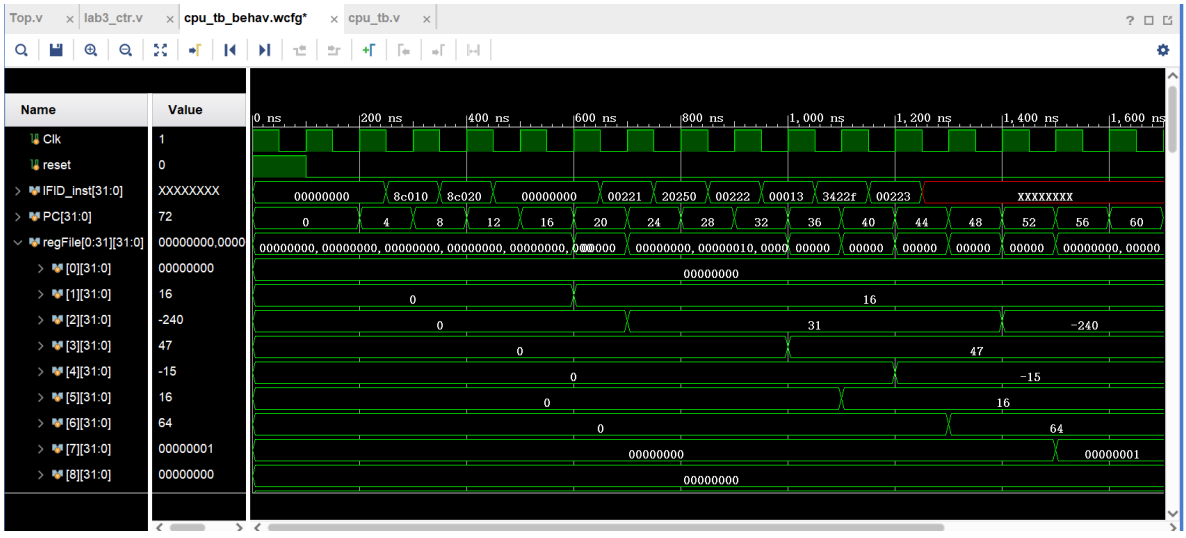
激励文件的代码如下：

```
always #50 clk=!clk;

    initial begin
    $readmemb("C:/Archlabs/inst.mem",cpu0.im0.instfile);
    $readmemh("C:/Archlabs/mem.mem",cpu0.dm0.memFile);
    clk=1;
    reset=1;
    #100 reset=0;

    end
```

对应的指令形式如下：

```
10001100000000010000000000001000 // lw $1 , 8($0)
10001100000000010000000000001100  //lw $2, 12($0)
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00000000001000100001100000100000  //add $3,$1,$2
00100000001001010000000000000000 //addi $5,$1,0
00000000001000100010000000100010  //sub $4,$1,$2
00000000000000000000000000000000 //nop
00000000000000001001100001000000  //sll $6, $1,2
00110100001000101111111100000000 //ori $1, $2, ...
00000000101000100011100000101010  //slt $7,$5,$2
```

生成的波形如下：



可以看到，在450ns-650ns之间，inst指令均为00，这是插入Nop语句以避免hazard的软法；并且我们还调整了指令的顺序以避免造成hazard（比如slt指令涉及到$5的数据，我们把它放在最后执行）。

另一方面本次实验的运算结果均符合逻辑结果。故，基础流水线的设计及软法验证是成功的。

5.1 流水线CPU完善后的硬法检测

设置了停顿、转发、分支预测等机制，硬件系统可以直接解决hazard等冲突。

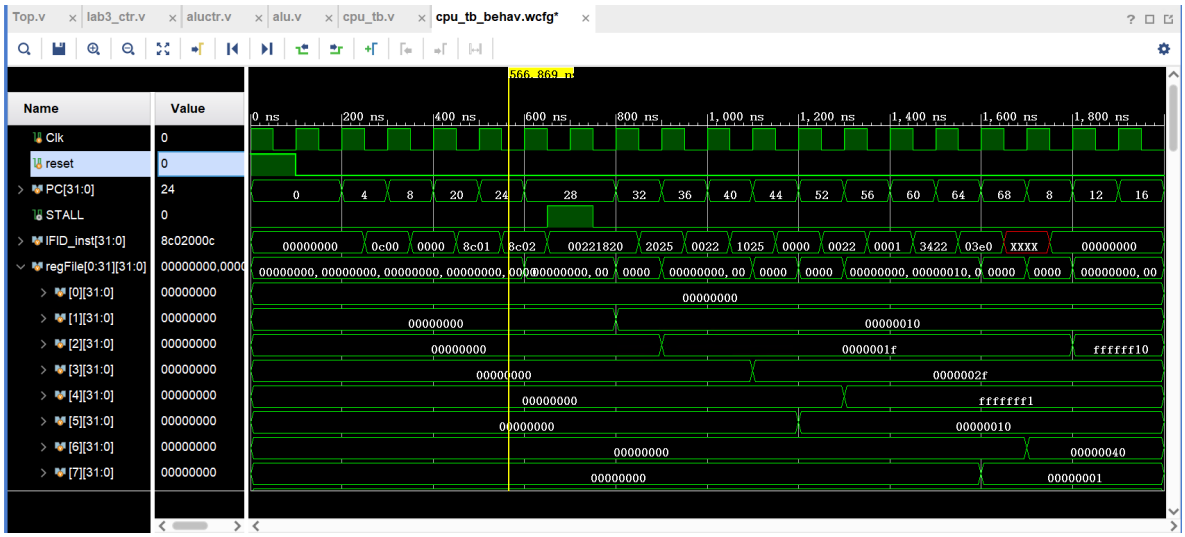
激励文件的代码与软法检测时相同。

执行的指令如下：

```
00000000000000000000000000000000 //nop
```

```
00001100000000000000000000000101 //jal 到第（5+1)个指令(lw) 并把pc+4写入$31
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
10001100000000010000000000001000  // lw $1 , 8($0) 16
10001100000000100000000000001100  //lw $2, 12($0) 20
00000000010001000011000000100000  //add $3,$1,$2
00100000001001010000000000000000 //addi $5,$1,0
00000000001000100010000000100010  //sub $4,$1,$2
00010000001001010000000000000010 //beq $1,$5, 跳1个
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00000000001000100011100000101010 //slt $7,$1,$2
00000000000000010011000010000000  //sll $6, $1,2
00110100001000101111111100000000 //ori $2, $1, ...
00000011111100000000000000001000 //jr $31
```

运行检测，得到的波形为：



可以看到，regFile呈现的结果是符合运算逻辑的(各种数学运算的输入输出和Lab5一致，这里不再赘述）。

特别地，由于引入了停滞和转发机制，所以所有指令均可正常运行（硬件正确处理了hazard冲突）。并且在执行时只出现了一次stall：这对应的是 `lw $2, 12($0) 20` 和 `add $3,$1,$2` 两行连续的指令。

此外，关注200ns-400ns的PC状态，这里对应的是 `jal` 指令，首先是更新PC的值（由4到8），当PC的跳转数据返回时，将刚刚新更新的数据置零，并将PC设置为新返回的跳转数据（由8跳变为20）。特别地，600ns时，可以看到光标的紧左侧，在regFile这一行波形出现了变化，这里是将pc+4=8的结果写入了reg[31].

关注1000ns-1200ns的PC状态，这里对应的是 `beq` 指令。由于我们设置了predict-not-taken, 所以我们先是继续执行后续指令（PC由40到44），当branch的跳转条件返回时，将刚刚新更新的数据置零（flush)，并将PC设置为新返回的跳转数据（由44跳变为52).

1700ns时，PC的值又变为8，这是因为 `jr` 指令将$31中的数据取出，并依次修改了PC的值。

综上，本次实验成功完成了多周期流水线处理器的设计与实现任务。

6 功能扩充（选做5）

6.1 基本描述

将之前已经实现的16条指令扩展为31条完整的指令集。

6.2 设计原理

由于此时要考虑到全部31种指令，opcode 的取值较多，因此很难准确分配3-bit的ALUop而不造成冲突。因此，我在本版块中，选择将ALUop的编码扩展为4-bit，这与原方案相比只是稍稍扩充了寄存器容量。考虑到这一更改使得编码更清晰简单，故硬件复杂度的微增是值得的。

因为31种指令非常繁杂，因此我们需要首先列出不同指令与ALUop、aluCtr的对应关系表，（这些编码是我个人的设计，和实际的MIPS架构可能存在些许出入）

指令	R	addi, lw , sw, jump	jal	beq	ori	andi	addiu	xori	lui	bne	slti	sltiu
ALUOP	0100	0000	0101	0001	0010	0011	1000	1010	1001	1011	1100	1101
aluCtr	-	0010	0010	0110	0001	0000	0011	1010	1011	1100	0111	1111
operation	-	add	add	sub	or	and	addiu	xor	lui	bne	slt	sltu

表1.各种指令与aluCtr、ALUOP的对应关系表

指令	add,addu	sub,subu	and	or	xor	nor	slt	sltu	sll,sllv	srl,srlv	sra, srav	jr
funct	100000, 100001	100010, 100011	100100	100101	100110	100111	101010	101011	000000, 000100	000010, 000110	000011, 000111	001000
aluCtr	0010	0110	0000	0001	1010	1110	0111	1111	1000	1001	0100	1101
operation	add	sub	and	or	xor	nor	slt	sltu	sll	srl	sra	jr

表2.R类型指令与aluCtr的对应关系表

根据上表，修改ALU, aluCtr 以及 ctr 模块，即可实现全部31条指令。

6.3 具体实现

模块改动的代码较多，且改动位置的分布较为分散，在报告中难以完全呈现改动情况，详情参见项目文件。

ALU模块的全部代码：

```
module alu(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output reg  zero,
    output reg [31:0] aluRes
     );

    integer i1,i2;
    reg [31:0] tmp;
    always @(input1 or input2 or aluCtr)
    begin
    case (aluCtr)
    4'b0000: aluRes=input1 & input2; //and
    4'b0001: aluRes=input1 | input2;  //or
    4'b0010: aluRes=input1 +input2;   //add
    4'b0110: aluRes=input1-input2;   //sub
    4'b1101: aluRes=0; //nothing
    4'b0111:
     begin
     i1=input1; i2=input2;
     aluRes=i1 < i2; //slt
     end
    4'b0011: //addiu
     begin
     tmp[15:0]=input2[15:0];
     tmp[31:16]=0;
     aluRes=input1+tmp;
     end
```

```
    4'b1010: aluRes=input1^input2;//xori
    4'b1011: //lui
     begin
     aluRes[15:0]=0;
     aluRes[31:16]=input2[15:0];
     end
    4'b1100://bne
     begin
     aluRes=input1-input2;
     if(aluRes) aluRes=0;
     end
    4'b1111: aluRes=input1 < input2; //sltu

    4'b1110: //nor
     begin
     aluRes= input1|input2;
     aluRes= ~aluRes;
     end
    4'b1000://sl
     aluRes=input2<<input1;

    4'b1001://sr
     aluRes=input2>>input1;

    4'b0100: //sra
    begin
    aluRes=input2>>input1;
    for( i1=0;i1<input1;i1=i1+1)
    aluRes[31-i1]=input2[31];
    end

    endcase

    if (aluRes) zero=0;
    else zero=1;
    end

endmodule
```

aluCtr模块的全部代码：

```
module aluctr(
    input [3:0] ALUOp,
    input [5:0] Funct,
    output [3:0] Oper,
    output reg shiftmux,
    output reg Jal,
    output reg Jr
    );

    reg [3:0]OPER;

    assign Oper=OPER;

    always@(ALUOp or Funct)
     begin
        shiftmux=0;
```

```
        Jal=0;
        Jr=0;
        casex({ALUOp,Funct})
            10'b0000xxxxxx://addi,lw,sw,j
            OPER=4'b0010;//add

            10'b0101xxxxxx://jal
            begin
            Jal=1;
            OPER=4'b0010;//add
            end

            10'b0001xxxxxx:// beq
            OPER=4'b0110;//sub

            10'b0010xxxxxx://i-or
            OPER=4'b0001;//or

            10'b0011xxxxxx:// i-and
            OPER=4'b0000;

            10'b1000xxxxxx: //addiu
            OPER=4'b0011;

            10'b1010xxxxxx: //xori
            OPER=4'b1010;

            10'b1001xxxxxx: //lui
            OPER=4'b1011;

            10'b1011xxxxxx: //bne
            OPER=4'b1100;

            10'b1100xxxxxx: //slti
            OPER=4'b0111;

            10'b1101xxxxxx: //sltiu
            OPER=4'b1111;

            10'b0100100000://r-add addu
            OPER=4'b0010;//add

            10'b0100100010: //r-sub subu
            OPER=4'b0110;//sub

            10'b0100100100://r-and
              OPER=4'b0000;//and

            10'b0100100101: //r-or
             OPER=4'b0001;//or

            10'b0100100110: //r-xor
             OPER=4'b1010;//xor

            10'b0100100111: //r-nor
             OPER=4'b1110;//nor

             10'b0100101010: //r-slt
```

```
            OPER=4'b0111;//slt

             10'b0100101011: //r-sltu
            OPER=4'b1111;//sltu

             10'b0100001000://r-jr
             begin
             Jr=1;
            OPER=4'b1101;//nothing
            end

            10'b0100000000://r-sll
            begin
            OPER=4'b1000; //sl
            shiftmux=1;
            end

            10'b0100000100: //r-sllv
             OPER=4'b1000;//sl

            10'b0100000010://r-srl
            begin
            OPER=4'b1001; //sr
            shiftmux=1;
            end

            10'b0100000110: //srlv
             OPER=4'b1001; //sr

            10'b0100000011://r-sra
            begin
            OPER=4'b0100; //sra
            shiftmux=1;
            end

            10'b0100000111://srav
            OPER=4'b0100; //sra

            default:
            begin
            OPER=4'b1101;//nothing
            end
        endcase
    end
endmodule
```

Ctr模块的部分代码：

```
 always@(opcode)
        begin
            case(opcode)
            6'b000000://R
            begin
                RegDst=1;
                ALUSrc=0;
                MemToReg=0;
                RegWrite=1;
```

```
                MemRead=0;
                MemWrite=0;
                Branch=0;
                ALUOp=4'b0100;
                Jump=0;
            end

            //  I - type
            6'b001000 ://addi
            begin
                RegDst=0;
                ALUSrc=1;
                MemToReg=0;
                RegWrite=1;
                MemRead=0;
                MemWrite=0;
                Branch=0;
                ALUOp=4'b0000;
                Jump=0;
            end

            6'b001001 ://addiu
            begin
                RegDst=0;
                ALUSrc=1;
                MemToReg=0;
                RegWrite=1;
                MemRead=0;
                MemWrite=0;
                Branch=0;
                ALUOp=4'b1000;
                Jump=0;
            end
        .......
       end
```

6.4 结果验证

此处主要检验新增指令的执行效果，此前已经检验无误的指令，这里不再赘述。

激励文件如下：

```
module cpu_tb(
    );
    reg clk;
    reg reset;

    Top cpu0(
    .clk(clk),
    .reset(reset));

    always #50 clk=!clk;

    initial begin
    $readmemb("C:/Archlabs/inst3.mem",cpu0.im0.instfile);
    $readmemh("C:/Archlabs/mem1.mem",cpu0.dm0.memFile);
    clk=1;
```

```
    reset=1;
    #100 reset=0;

    end

endmodule
```

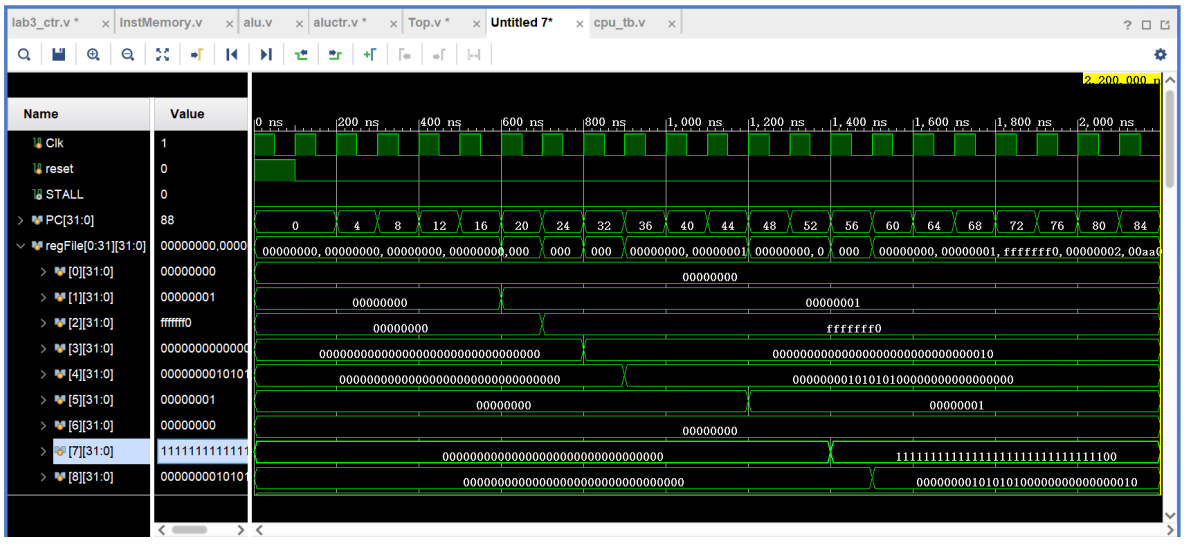
需要执行的指令，如下：

```
00000000000000000000000000000000 // nop
10001100000000010000000000000000 //lw $1, 0($0)
10001100000000010000000000000100 //lw $2, 4($0)
10001100000000011000000000001000  //lw $3, 8($0)
00111100000001000000000010101010 //lui $4, 0000000010101010
00010100001000110000000000000010 //bne $1,$3, 10b
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00101000010001010000000000000000 //slti $5, $2,00
00101100010001100000000000000000 //sltiu $6,$2,00
00000000011000100011100000000111 //srav  $7,$2,$3
00000000100000110100000000100110 //xor $8,$3,$4
```

存储器中的数据如下：

```
0x00
0x00
0x00
0x01  // 1
0xff
0xff
0xff
0xf0 //-16
0x00
0x00
0x00
0x02 // 2
```

运行上述文件，得到如下波形：



观察上述波形图：寄存器$1,$2,$3 中分别存储着内存中的数据1，-16，2 。

$4对应的是`lui`指令，即立即数0000000010101010被存放在高16位中，而低16位置零。

随后执行`bne`指令，因为$1,$3的数据不同，因此PC会跳转（从20到24再直接到32）

后面是有符号比较`slti`和无符号比较`sltiu`,在有符号比较时，$-16 < 0$，因此$5 被置为1；而无符号比较时，ffffffff0>0, 因此$6 的值依然为0.

`srav`指令，将ffffffff0左移两位，并在高两位填充上11.即得到了$7中的数据。

而$8中呈现的则是$3 和$4 异或的结果。

综上，该波形的结果均符合运算逻辑，本选做板块的实验是成功的。

7 思考总结

本实验的难度相较以往有了较大的增加，在完成此实验时我也花费了更多的时间。

本实验的难点之一是寄存器和线路复杂繁多，因此我们需要对这些元件进行正确且清晰的命名。在每个流水线寄存器中，我都先说明了所处流水线的位置，再说明储存数据的种类，如`EXEM_ALUres`这样的命名方式。这些寄存器往往需要一定的线路和其他部件相连，因此我选择了对应命名的全大写模式如`EXEM_ALURES`。其他的寄存器，我则采用了小驼峰命名法，每一个单词的首字母大写，如`MemToReg`。有了清晰的命名方法，能够给本次实验提供很大的帮助。

本次实验一开始，我并没有考虑到读写的同步问题，将所有寄存器都设置成了上升沿写入数据。但后来发现，当我们需要在一个周期内完成读写时，这会造成很多冲突。因此我设置了前半周期写入、后半周期读取的方案，成功解决了这一问题。

一开始在设计分支语句的时候，我将branch的判断放置在ID阶段的起始。但是后来经过多组样例的测试发现，这样虽然能保证bubble的数目最少，最早得到结果。但假如`beq`的比较数据是尚未写入reg的数据，那么这样的设计必然会导致错误。后来我决定将branch的判断放置在ID、EX交界处，这样就可以利用forward机制，尚未写入reg的数据会被直接转发,这保证了判断结果的准确性。特别地，经过理论分析和实际调试，我发现，这种情况下bubble的数量依然是1，并没有增加。因此我最终选择了后者的设计。

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，这给调试带来了巨大的困难。对于这种情况，需要从出错的地方开始，按照时间线逐步推理。这一过程虽然繁杂枯燥，但是同时也使我对流水线的工作流程更为熟悉，加深了我的理解。

8 最终体会

在本课程中，我有不少收获，在此作一概述。

之前我曾参加过许多科目的实验课程，但体验感和收获大多都很一般。然而本课程则不然，它在设置方面非常合理，能紧紧贴近课堂所学，在巩固课堂知识、加深我对底层架构理解的同时，鼓励我在某些设计上做出了创新和突破。另一方面，本次实验后两个lab的设计较为复杂，且细节较多，这很好地锻炼了我细心和严谨的习惯，也磨砺了我不畏困难的意志。

这是我第一次接触到硬件语言，也因此对硬件底层实现有了全新的理解和认识。遗憾的是，因为疫情的原因，我们暂时无法进行上板验证，也希望在返校后老师能给予我们上板验证的机会，加深我们对硬件的理解。

最后，由衷感谢老师对本课程内容的精心设置，感谢老师在疫情期间仍然在线上为我们耐心地答疑解惑。