计算机系统结构实验报告实验 6

小卷王
123456781234
xiaojuanwang@sjtu.edu.cn

目录

1 实验目的

- 实现一个简单的类 MIPS 多周期流水线处理器，能够完成多周期流水线对于指令处理的模拟。
- 能够检测竞争并用 Stall 机制来解决冒险。
- 能够通过 Forwarding 机制减少流水线停顿。
- 能够通过 predict-not-taken 机制减少流水线停顿。
- 能够支持 31 条指令。

2 实验步骤

实现一个简单的类 MIPS 多周期流水线处理器。首先我们会对原来的单周期的处理器进行改进，把单周期处理器模块全部导入进来在此基础上进一步的修改。我们先会实现最基本的多周期处理器，主要方法就是加流水线寄存器，在这一个部分的难点就是如何处理时序逻辑的思路。然后在后面我们会对 CPU 进行优化包括 stall、forwarding 和 predict-not-taken 机制，其中注意的地方是保证数据的一致性，传输正确。最后再对指令进行一些拓展。

2.1 多周期处理器实现

实现多周期处理器最大的改进就在于需要设置流水线寄存器保存上一个模块内的功能部件的有效信息，并且传递给下一个模块。除此之外我们还有一些细节的地方注意处理到就可以了。

2.1.1 流水线寄存器

添加流水线寄存器在各个模块之间我们设置流水线处理器，这是一个五阶段流水线包含一下几个部分 IF、ID、EX、MEM、WB 各个部分在每一个部分它们所用的部件分别是

- IF: muxPC, PC, Inst memory, Pcadd4
- ID: 控制器，主寄存器，符号位扩展单元，提前转移相关部件
- EX: ALU 控制器，ALU，muxALU，muxWriteReg
- MEM: 内存
- WB: muxmemory

对于每一个流水线寄存器，都设定在时钟的上沿触发，将上一级单元的内容传送到下一级单元，使用时序逻辑。设计的时候很简单，检查一下自己下一级各个单元或者之后的各个单元要什么东西，那就传给它们就可以了。我们总共有 4 个流水线寄存器，分别是在 IF/ID, ID/EX, EX/MEM, MEM/WB 而第一个我们不作为特殊的流水线，但是也是同样在时钟上沿出发的就是 PC，这样五个时序部件控制每个时钟周期的指令传递，形成一个多周期的流水线。

每一个流水线寄存器传入什么，传出什么这里就不多赘述了，我们可以结合代码查看所需要的内容，实现非常简单。对于我们所有要输出的内容，我们都做一个初始化。然后再每个时钟上沿到来的时候进行传递就可以了，使用时序逻辑。

IF/ID 寄存器的代码如下

```
module IFID(
    input Clk,
    input [31:0] instaddress,
    input [31:0] instruction,
    output reg [31:0] instructionout,
    output reg [31:0] instaddressout
    );
    initial begin
        instructionout=0;
        instaddressout=0;
    end
    always @ (posedge Clk)
    begin
          instaddressout<=instaddress;
          instructionout<=instruction;
    end
endmodule
```

ID/EX 寄存器的代码如下

```
module IDEX(
    input Clk,
    input [31:0] signext,readdata1,readdata2,instaddress,instruction,
    input regDst,aluSrc,regWrite,memToReg,
          memRead, memWrite,Jump,jr,jal,
    input [3:0] ALUOp,
    input [4:0]currentreg1,currentreg2,
    input [4:0] lastregdistin,
    input [31:0] lastaluresultin,
    input lastlastisload,
    input [4:0] loadregdist,
    input [31:0] loadmemresult,
    output reg regDstout,aluSrcout,regWriteout,
               memToRegout, memReadout,memWriteout,
                Jumpout,jrout,jalout,
    output reg [3:0]ALUOpout,
    output reg [31:0]instaddressout,readdata1out,readdata2out,
                       signextout,
    output reg [4:0] instruction2016out,instruction1511out
    );
    reg [31:0]lastlastaluresult,lastaluresult,lastaluresulttemp;
    reg [4:0] lastlastregdist,lastregdist,lastregdisttemp;
    reg lastisload;
    initial begin
        regDstout=0;
        aluSrcout=0;
```

```
27          regWriteout=0;
28          memToRegout=0;
29          memReadout=0;
30          memWriteout=0;
31          Jumpout=0;
32          jrout=0;
33          jalout=0;
34          ALUOpout=0;
35          instaddressout=0;
36          readdata1out=0;
37          readdata2out=0;
38          signextout=0;
39          instruction2016out=0;
40          instruction1511out=0;
41      end
42      always @(posedge Clk)
43      begin
44              regDstout<=regDst;
45              aluSrcout<=aluSrc;
46              regWriteout<=regWrite;
47              memToRegout<=memToReg;
48              memReadout<=memRead;
49              memWriteout<=memWrite;
50              Jumpout<=Jump;
51              jrout<=jr;
52              jalout<=jal;
53              ALUOpout<=ALUOp;
54              instaddressout<=instaddress;
55              //readdata1out<=readdata1;
56              //readdata2out<=readdata2;
57              signextout<=signext;
58              instruction2016out<=instruction[20:16];
59              instruction1511out<=instruction[15:11];
60              if(!lastisload && currentreg1==lastregdist)
61                  readdata1out<=lastaluresult;
62              else if(!lastisload && currentreg1==lastlastregdist)
63                  readdata1out<=lastlastaluresult;
64              else if(lastlastisload && currentreg1==loadregdist)
65                  readdata1out<=loadmemresult;
66              else
67                  readdata1out<=readdata1;
68              if(!lastisload && currentreg2==lastregdist)
69                  readdata2out<=lastaluresult;
70              else if(!lastisload && currentreg2==lastlastregdist)
```

```
                readdata2out<=lastlastaluresult;
            else if(lastlastisload && currentreg2==loadregdist)
                readdata2out<=loadmemresult;
            else
                readdata2out<=readdata2;
            lastregdisttemp<=lastregdist;
            lastaluresulttemp<=lastaluresult;
            lastisload<=memRead;
    end
    always @(negedge Clk)
    begin
            lastlastregdist<=lastregdisttemp;
            lastlastaluresult<=lastaluresulttemp;
            lastregdist<=lastregdistin;
            lastaluresult<=lastaluresultin;
    end
endmodule
```

其中有部分内容用于 forwarding 机制，会在之后进行介绍。

EX/MEM 寄存器的代码如下

```
module EXMEM(
    input Clk,
    input  regWrite,memToReg,memRead,memWrite,
                jal,
    input [31:0] instaddress,aluresult,readdata2,
    input [4:0] regdistaddress,
    output reg regWriteout,memToRegout,jalout,memWriteout,memReadout,
    output reg [31:0] aluresultout,readdata2out,instaddressout,
    output reg [4:0]regdistaddressout
    );
    initial begin
        regWriteout=0;
        memToRegout=0;
        jalout=0;
        aluresultout=0;
        readdata2out=0;
        instaddressout=0;
        regdistaddressout=0;
        memWriteout=0;
        memReadout=0;
   end
   always @ (posedge Clk)
   begin
            regWriteout<=regWrite;
            memToRegout<=memToReg;
```

```
            jalout<=jal;
            aluresultout<=aluresult;
            readdata2out<=readdata2;
            instaddressout<=instaddress;
            regdistaddressout<=regdistaddress;
            memWriteout<=memWrite;
            memReadout<=memRead;
    end
endmodule
```

MEM/WB 寄存器的代码如下

```
module MEMWB(
    input Clk,
    input regWrite,memToReg,jal,
    input [31:0] instaddress,regwritedata1,regwritedata2,
    input [4:0] regdist,
    output reg regWriteout,memToRegout,jalout,
    output reg [31:0] instaddressout,regwritedata1out,regwritedata2out,
    output reg [4:0] regdistout
    );
    initial begin
        regWriteout=0;
        memToRegout=0;
        jalout=0;
        instaddressout=0;
        regwritedata1out=0;
        regwritedata2out=0;
        regdistout=0;
    end
    always @(posedge Clk)
    begin
            regWriteout<=regWrite;
            memToRegout<=memToReg;
            jalout<=jal;
            instaddressout<=instaddress;
            regwritedata1out<=regwritedata1;
            regwritedata2out<=regwritedata2;
            regdistout<=regdist;
    end
endmodule
```

2.1.2 PC 与 PCmux 改进

为了实现多周期流水线的特性我们要对整个 PC 的生成和 +4 功能做一下改变，在单周期处理器设计的时候比较简单，但是这里则需要能够将 PC 功能的单元设定在 IF 阶段完成。实现 PC 的功能我

们有三个功能部件，分别是 PC、PCadd4 和 PCmux。接下来一一介绍。

PC 主模块的思想很简单，PCmux 会把已经判别过的下一跳需要执行的指令送进去，PC 要做的就是完成一个时序逻辑的工作，在下一个时钟周期到来的时候把指令地址送出去就可以了。相比于单周期模块没有什么改进，关于 stall 和 flush 机制的内容之后会介绍。

PCadd4 模块，一条简单语句

```
assign instaddressout=flush?instaddressin:instaddressin+4;
```

实现 PC+4。这里引入三目运算符之后会说到。

PCmux 模块，虽然名字叫做 Mux 但是实际是一个三路选择器，会接受到三个不同的地址，分别是 branch 指令相关的 branchaddress，jump 指令相关的 jumpaddress 和本身的 PC+4 地址。同时根据 branch 和 jump 的情况选择输出的地址。核心代码很简单。

```
    initial begin
    output1=0;
    end
    always @ (*)
    begin
        output1=pcadd4address;
        if(jump==1)
            output1=jumpaddress;
        if(branch==1)
            output1=branchaddress;
    end
```

2.1.3 其他功能部件改进

为了实现多周期处理器，我们之前的一些功能部件需要进行一些微调以满足多周期情况下的一些要求。

关于 jr 指令和 jal 指令，为了满足能够准确识别出 jr 和 jal 指令，我们对于控制器的输入将原来的六位改成了全部的指令都作为控制器的输入。当识别出 jr 指令以后因为在 ID 阶段本身就要读 register 所以没有影响，具体操作在 predict-not-taken 章节介绍。jal 指令需要传输控制信号 isjal 下去，保证在写回寄存器的内容的正确性。

为了避免结构冒险，对于寄存器和内存还需要进行一些改进

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	

对于寄存器部件来说有可能同时在第一条指令的 WB 和第四条指令的 ID 阶段使用，为了避免如果两者操作指向同一个寄存器发生先读后写的情况，我们需要避免这种结构冒险。因此我们将寄存器设定为在时钟的上半个周期进行写入在后半个周期进行读取，因此我们将寄存器模块进行修改。

内存本身因为我们已经将指令内存和数据内存分开来，不存在这种问题但是为了保证协调性也一起将读操作设定到了后半个周期进行。

各模块初始化，为了保证指令能够正确运行我们需要对所有的模块增加初始化，默认将所有模块所有的变量都初始化为 0。并且需要特别注意的是在控制器模块如果读到的指令内容是 0，也就是一条空指令也要把所有的单元全部都重置为 0。

2.1.4 整理检查 Top 模块

最后我们进行连线，因为在本实验中变量实在是太多了，因此我们设定了一套命名规则。对于除了 TOP 模块以外的模块，同一个变量输入我们不变，输出叫做 out。例如 IDEX 寄存器模块中输入我们命名 instruction 输出叫 instructionout。

在 TOP 模块中我们有许多数据线需要进行连接，首先我们先按各个单元的位置把它们分门别类排列好，然后数据线也是如此。然后我们规定下面的命名规则

- 同一个信号，如果要被多次传递起名信号名称 + 输出它的模块。
  例如 instaddresspcmux,instaddressPC,instaddresspcadd4 等等。
- 如果某一个信号从某一个模块中传出以后再也不会继续把这个内容传递下去，省略输出它的模块的名字，例如 IDEX 寄存器输出的 regDst 和 aluSrc。
- 如果某个信号是一个大信号的子部分直接用数字表示 instruction1511,instruction2016 等等。

然后我们按每个模块的输出排好线，然后依次实例化各个模块并且接线，这个过程是非常仔细还要很耐心而且容易出错的地方。在本作者实现的过程有一个小的心得，设定线路的时候一定要注意位数的问题。例如 aluResultalu 应该是一个 32 位的数字，我们要把线路设定为 32 位线宽才是正确的。但是本作者实现的时候一开始忘记写 [31:0] 导致时钟获得不了结果，最终排查发现是位宽指定错误浪费了大量的时间。因此在时间的过程中要非常的小心而且仔细。最后再检查一遍各个模块和 Top 确认无误!

2.2 Stall 机制的加入

在上一章节我们已经解决了结构冒险，而我们此时还剩下数据冒险和控制冒险的问题。

首先关于数据冒险，因为我们在下一章节将使用 forwarding 机制，因此不会产生 R&R，或者 R&I 型指令的数据冒险，因为 R 型指令能在 ALU 算出结果以后就通过前向通路把结果传输到下一级的 ID 单元。同样的大部分的 I 型指令也不会。但是唯独涉及到 memory 的指令不能够成功运行，因为

IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB

我们可以看到如果两句语句是紧接着的也没有办法把 MEM 读到的东西马上传给下一跳的 EX 单元，因此在这种情况下我们只能加入一条空指令浪费一个时钟周期，在这种情况下我们也人就是需要前向通路的。所以如果我们检测到是 load 指令加上下一条指令需要读的寄存器和现在 load 需要写入的寄存器是同一个寄存器我们就需要进行 stall。因为 memory 我们也设计了前向通路，所以只要 stall 一条空指令就可以了。

这里比较容易忽略的一种情况是如果前面一条指令是 R 型或者其他型的指令，而后面是 jr、bne、beq 三条指令就算使用了 forwarding 机制，因为我们采取了 predict not taken 机制因此我们在第二个阶段 decoding 阶段就需要计算了，所以需要走空一条指令。

```
1 //load 指令
2 //load&R
3 if(Instruction[31:26]==6'b100011 && nextinstruction[31:26]==6'b000000 &&
4 (Instruction[20:16]==nextinstruction[25:21] ||
5 [20:16]==nextinstruction[20:16])) flag=1;
6 //load &I
7 if(Instruction[31:26]==6'b100011 && (!nextinstruction[31:26]==6'b000000
8 && !nextinstruction[31:26]==6'b000010 &&!nextinstruction[31:26]==6'b000011)
```

```
&&(Instruction[20:16]==nextinstruction[25:21])) flag=1;
//R&bne beq
if(Instruction[31:26]==6'b000000 &&
    (Instruction[31:26]==6'b000100 || Instruction[31:26]==6'b000101)
    &&(Instruction[15:11]==nextinstruction[25:21]||
    Instruction[15:11]==nextinstruction[20:16]))
    flag=1;
//R&jr
if(Instruction[31:26]==6'b000000&&
(Instruction[31:26]==6'b0000000 && Instruction[5:0]==6'b001000)&&
(Instruction[15:11]==nextinstruction[25:21]))
    flag=1;
```

关于控制冒险，因为我们在接下来的章节会使用 predict-not-taken 机制，因此在 ID 阶段 jump 或者 branch 指令的计算都会提前的完成，但是在这种情况下我们就需要将后面一条指令 flush 掉，因此为了避免控制冒险的发生我们也采用 stall 机制，如果这是一条 Jump 或者 branch 指令我们插入一条空指令然后继续，避免控制冒险的发生。因为我们可以在 ID 阶段就做出决定，因此也只需要 stall 一条空指令。

```
if(Instruction[31:26]==6'b000100 || Instruction[31:26]==6'b000101||
Instruction[31:26]==6'b000010 || Instruction[31:26]==6'b000011)
flag=1;
//jr
if(Instruction[31:26]==6'b0000000 && Instruction[5:0]==6'b001000)
flag=1;
```

上述的检测工作在 Instmemory 模块完成，首先按照地址获得指令如果检测到这是一条我们需要使用 stall 机制的指令那么设定 flag 为 1，然后此时 Instmemory 模块不再输出按照地址获得的指令，而是输出一条空指令。同时将 flush 信号设置为 1 传输给 PC 告知此时本条指令使用了 stall 机制。

```
if(flag)
   begin
        flush=1;
        Instruction=0;
    end
else
    nstruction=nextinstruction;
```

同时在 PC 模块内当我们收到了 flush 的信号当前地址的这条指令说明并没有被成功执行，因此我们在下一个周期还是要继续使用我现在的这条指令，因此在 PC 模块内我们进行如下操作

```
if(flush)
    begin
        if(instaddress==address)
            instaddress<=instaddress;
        else
            instaddress<=address;
            flushdeal<=1;
    end
```

```
9  else
10     begin
11         instaddress<=address;
12         flushdeal<=0;
13     end
```

如果我们收到了进行 stall 的信号下一个周期继续输出当前指令，否则我们还是照常进行。同时我们这里还需要设计一个额外的变量叫做 flushdeal，因为我们 PC 继续保持了一样的地址，那么 instmemory 会收到重复的地址重复的指令，然后重复检测到这是一条需要 stall 的指令，陷入死循环。因此我们需要 flushdeal，按照前面的分析我们最多也只需要 flush 一次，因此只要 PC 进行过一次 stall 就给出一个 flushdeal 为 1 的信号，否则为 0。对于 instmemory 如果 flushdeal 是 1 就说明这条看似需要 stall 的指令其实已经 stall 过了，不要重复操作。因此将 flag 重新调回 0 就可以了。

另外需要注意的一件事情在于我们需要对 PCadd4 进行适当的改进，使用 flush 以后需要记得使得 Pcadd4 不要持续的不停加 4，如果不使用那样一个三目运算符导致的结果是如果条件分支的条件没有成立会跳过下一跳指令转而运行之后的指令，而这是存在问题的。

综合上述两处修改，再结合之后两个章节的机制我们可以实现最多 stall 一次来避免数据冒险和控制冒险。

2.3 Forwarding 机制的加入

为了进一步优化 CPU，加入旁路传递机制，旁路传递机制的实现思想如下。因为我们要把前面指令的运行结果传入之后指令的 EX 阶段，也就是 ALU 运算的时候使用的两个数我们不能再使用错误的还没有写入数据的寄存器读到的数据，而是之前还没来得及存到寄存器里面的 ALU 运算结果或者内存读取结果。因此对于 ID/EX 流水线寄存器我们需要新增几个输入接口，后面会介绍具体。然后新增获得这些现在 ALU 或者内存得到的结果。那么在 ID/EX 内部判断现在这条指令的寄存器，rs 或者 rt 是不是就是前面几条指令的 rd。如果是的话那么我就需要在输出的时候不要输出 readdata 而是输出前面的结果，在下一个时钟上沿到来的时候。

首先寄存器得新增一个输入指令，同时获取 currentreg1 和 currentreg2 既可以输入也可以直接从指令选择，本实验中通过传输获取。

2.3.1 ALU 结果旁路传递

新增输入 input [4:0] lastregdistin,input [31:0] lastaluresultin 直接在 Top 总线模块把 aluresult 和 EX 阶段获得的 reg destination 传输进来。同时我们额外的还要在寄存器里面再设置六个变量

reg [31:0]lastlastaluresult,lastaluresult,lastaluresulttemp;
reg [4:0] lastlastregdist,lastregdist,lastregdisttemp;

这里 lastlast 变量很直观，因为我们的前向通路不光要把当前的运算结果传给下一条指令，在再下一条指令执行 ID 单元的时候也还没有成功把结果 write back 所以也需要一个前向通路。而另外两个变量则是为了解决时序逻辑中的数据传输正确性。

我们不能直接使用 lastaluresultin 作为 lastresult 否则我们其实读到的明明是现在这条指令的 alu 结果我们却错当成是上一条的。因此我们需要在时钟下沿的时候将 lastresult 用 lastresultin 进行赋值。同理我们也不能直接把 lastresult 传输给 lastlastresult 否则这样两者就是一个数据而不是时间往前推，因此我们要使用一个 temp 变量，在时钟下沿的时候用 temp 的内容去赋值 lastlast 变量。因为 always 内的各个语句是并行执行的，因此在 always 内我们只能用 last 变量去赋值 temp 变量，通过这样在时钟上下沿多走一步的方法，我们才可以在时序逻辑中保证，lastlast 和 last 变量都是正确的内容而不是上个回合和这个回合的内容。

另外一件事情，我们还一定要保证上一条指令不是 load 指令，因为如果上一条指令时 load 指令的话我们的 alu 结果是没有实际意义的对于这里的前向通路来说，如果错误的输入反而导致错误的处理。因此还需要一个 lastisload 变量通过上一轮的 memRead 指令就可以判断。

这是特殊处理内容

```
always @(negedge Clk)
begin
    lastlastregdist<=lastregdisttemp;
    lastlastaluresult<=lastaluresulttemp;
    lastregdist<=lastregdistin;
    lastaluresult<=lastaluresultin;
end
```

在时钟上沿的操作语句内

```
lastregdisttemp<=lastregdist;
lastaluresulttemp<=lastaluresult;
lastisload<=memRead||memWrite;
```

2.3.2 LW 结果旁路传递

新增输入 input lastlastisload,input [4:0] loadregdist,input [31:0] loadmemresult,

从 memory 中读取到的数据就没有上述的问题，因为 memory 模块并不是紧接在 ID/EX 寄存器后面的因此上述的 last 被误传这回合的内容情况不会发生。我们直接用 load 的内容就可以了。当然我们还需要特别的一个变量就是这里的 lastlastisload 指令，因此在满足寄存器指向同一处并且用 lw bypassing 机制的先决条件一定是上上条指令是一条 load 指令。因此核心的我们实现前向通路的代码只需要在 ID/EX 内进行修改就可以，如下

```
if(!lastisload && currentreg1==lastregdist)
    readdata1out<=lastaluresult;
else if(!lastisload && currentreg1==lastlastregdist)
    readdata1out<=lastlastaluresult;
else if(lastlastisload && currentreg1==loadregdist)
    readdata1out<=loadmemresult;
else
    readdata1out<=readdata1;
if(!lastisload && currentreg2==lastregdist)
    readdata2out<=lastaluresult;
else if(!lastisload && currentreg2==lastlastregdist)
    readdata2out<=lastlastaluresult;
else if(lastlastisload && currentreg2==loadregdist)
    readdata2out<=loadmemresult;
 else
    readdata2out<=readdata2;
```

分别对 currentreg1 和 2 进行判断，如果上一条指令不是 Load 指令并且寄存器重叠那就用前向通路传递正确的结果。如果上上条是 Load 指令并且重叠那前向传递内存的内容。如果都不是按照正常情况处理就可以了。至此完成前向通路的设计。

2.4 Predict-not-taken 机制的加入

为了进一步提升 CPU 性能我们采用 predict-not-taken 机制，简单来说就是将所有的跳转指令提前就在译码阶段完成，这样最多只需要浪费一个周期的时间。

实现的方法是增加三个模块分别用于获得 jump 指令相关的 jumpaddress，获得 branch 指令相关的 branchaddress 和一个判断器用于判断是否满足 branch 指令的条件。

1. jumpaddressgen 提前获得 jumpaddress，思路很简单把 instruction[25:0] 左移两位再加上原先 instaddress[31:28] 位的工作提前到这个模块进行，如果是 jr 指令那么就用输入的 readdata 跳转到该地方即可。

```
module jumpaddressgen(
input jump,jr,
input [31:0] instaddress,instruction,readdata1,
output reg [31:0] jumpaddress
);
reg [25:0] temp;
initial begin
    jumpaddress=0;
end
always @ (jump or jr or readdata1 or instaddress or instruction)
begin
        if(jr)
            jumpaddress=readdata1;
        else
         begin
               temp=instruction[25:0]<<2;
               jumpaddress={instaddress[31:28],temp};
         end
end
endmodule
```

2. addressGen 这个模块作用是获得 branchaddress，本质就是新增一个加法器，因为 signextend 的操作也是在 ID 阶段就完成的因此直接把立即数左移以后和地址相加就可以了。

```
module addressgen(
    input [31:0]instaddress,signext,
    output reg [31:0] branchaddress
    );

    initial begin
        branchaddress=0;
    end
    always @ (instaddress or signext)
    begin
         branchaddress=$signed(instaddress)+($signed(signext)<<<2);
    end
```

```
endmodule
```

3. 提前决策器 AdvancDecide 因为原先需要通过 zero 的值来判断是否满足条件，由于现在提前进行判断因此我们把 register 读出的两个数据直接进行判断是否相等，在后续我们还会加入 bne 指令，因此控制器会给出一个控制信号告知是不是 equal 的 branch 指令吗，然后提前决策器根据控制信号，读到的寄存器内容进行判断。

```
module AdvanceDecide(
    input [31:0] readdata1,readdata2,
    input isequal,
    output reg satisfy
    );
    initial begin
        satisfy=0;
    end
    always @ (readdata1 or readdata2 or isequal)
    begin
        if(isequal)
            satisfy=(readdata1==readdata2);
        else
            satisfy=!(readdata1==readdata2);
    end
endmodule
```

最后把 jump,branch&satisfy 控制信号，以及 jumpaddress 和 branchaddress 输入到 PCmux 中去，由选路器进行地址选择即可。

2.5 指令扩展

进一步丰富处理器的功能，将原本的 16 条指令扩展到 31 条指令，新增的指令包括将原来的有符号数指令进行扩展到无符号指令数，因为我们这是一个简化的 CPU，对于普通的加减位运算来说并不存在有符号数和无符号数的区别换言之不管是无符号数还是有符号数我们单纯的加减就可以了，因为我们不需要去考虑 CPU 内部的 flag 信息。唯一需要注意的对于右移操作和 slt 指令我们还是需要区分有符号数和无符号数，因为运算的结果是不一样的。因此对于新增的指令我们需要做出调整的有 xor, sltu, sra, sllv, srlv, srav, xori, lui,bne,slti, sltiu。

对于 I 型中和 R 型中一样操作的指令，我们通过扩展 aluOp 的位数来实现能够兼容更多的指令，直接在译码阶段读取到 I 型指令的前六位以后就给出相应的 aluOp，然后 aluCtr 模块会根据 aluOp 的内容进行相应的选定的 ALU 操作。这里我们为了满足所有指令的需求将 aluOp 的位数扩展到 4 位。

2.5.1 ALU 工作指令增加

ALU 有了更多的需要进行的操作，因此我们对 aluCtr 信号的位数也进行相应的扩展，扩展到 5 位使得 ALU 能够满足所有包括有符号无符号运算在内的所有操作。具体的情况如表 1所示，列举了在 ALU 中的所有运算操作。

00010	00110	00000	00001
+	-	and	or
01100	00101	00111	01110
nor	xor	slt	sltu
00011	00100	01011	01000
<<shamet	>>shamet	>>shamet signed	copy input2
01001	01010	01101	10000
<<	>>	>>signed	lui op

表 1: aluCtr 与对应操作

2.5.2 新增特殊指令

实际上只有两条特殊的指令需要加入，bne 指令和 lui 指令。bne 指令通过 isequal 信号就可以处理上一个章节已经介绍过了。lui 指令的功能是将 16 位立即数放到目标寄存器高 16，指令的 rs 部分默认是 00000 所以我们只需要再设计一种特殊的操作就可以了，其余的方法我们遵循立即数运算的方法即可。并且我们也需要把它和别的操作一样对待，因为有可能会需要用 forwarding。多增加一种 aluOp 和 aluCtr 然后具体我们在运算器内部的实现很简单，立即数扩展的结果我们只取后面 16 位数字，然后再拼接很多 0 就可以。

```
aluRes={input2[15:0],{16'h0000}};
```

至此我们完成了所有指令的扩展和多周期处理器的设计

3 实验仿真与结果

3.1 仿真准备

在仿真阶段我们需要准备好相关的指令来测试我们的 CPU 是否能够运行正确，为了能够正确检查结果我们需要观察的变量包括 registers 的内容和 instruction 的内容（用于检测跳转指令）。

3.1.1 激励模块

激励模块的设计很简单，我们设定一个时钟周期，在初始的时候进行 reset 然后一直在半个时钟周期更换 Clk 的内容即可。

```
always #(PERIOD) Clk=!Clk;
initial
begin
    Clk=1;reset=1;
    #5 reset=0;
end
```

3.1.2 内存与指令

我们需要事先准备两个文件分别是数据内存和指令内存首先我们设计内存内部的存储数据[1]，随机存放一些数据就可以了，这里就展示前 13 个数据，我们在之后测试的最多也就用到这么多没有必要多设计。

[1]在不同设备下测试的时候我们需要改动绝对路径的地址以保证导入正确性

```
6
8
1
1
1
1
1
1
1
1
1
5
8
```

接下来我们设计用来测试的四组指令，第一组指令时参考资料实验手册中的样例指令作了一些的修改，第二组指令是我们特殊设计用的来测试跳转指令的一组指令，第三组指令用来测试部分额外扩展的指令的运行效果。第四组指令沿用了 lab05 中的测试指令，含有很多的数据相关性，并且同时也可以测试基本的 16 条指令的内容下面分别阐述。

1. 第一组指令时样例指令做了一些修改主要用来测试前向通路，主要测试了一些基本的情况，汇编代码如下

```
lw $1,40($0);
lw $2,44($0);
lw $3,48($0);
add $4,$1,$2;
sub $5,$4,$1;
and $6,$5,$4;
lw $10,40($0);
lw $10,40($0);
lw $10,40($0);
or $7,$3,$1;
slt $8,$3,$1;
beq $0,$0,1;
add $9,$7,$8;
lw $10,40($0);
lw $10,40($0);
lw $10,40($0);
lw $10,40($0);
lw $10,40($0);
```

二进制代码如下

```
10001100000000010000000000001010
10001100000000010000000000001011
10001100000000110000000000001100
00000000001000100010000000100000
00000000100000010010100000100010
```

```
00000000101001000011000000100100
10001100000010100000000000001010
10001100000010100000000000001010
10001100000010100000000000001010
00000000011000010011100000100101
00000000011000010100000000101010
00010000000000000000000000000010
00000000111001000100100000100000
10001100000010100000000000001010
10001100000010100000000000001010
10001100000010100000000000001010
10001100000010100000000000001010
10001100000010100000000000001010
```

2. 第二组指令的构思也很简单，测试是否能够使用 stall 和 predict-not-taken 机制成功的处理一些跳转指令。汇编代码如下

```
beq $1,$0,4;
sll $30,$27,2;
srl $24,$30,3;
slt $21,$29,$28;
beq $24,$0,1;
j 1;
jal 7;
jr $1;
```

翻译成二进制代码

```
00010000001000000000000000000100
00000000000110111111000010000000
00000000000111101100000011000010
00000011101111001010100000101010
00010000001000000000000000000001
00001000000000000000000000000001
00001100000000000000000000000111
00000000001000000000000000001000
```

3. 第三组测试一些新指令，包括 xor,sll,lui 三条指令的测试，汇编代码如下

```
lw $1,40($0);
lw $2,44($0);
lw $3,48($0);
xor $4,$1,$2;
sll $5,$1,$2;
lui $0, 0hffff;
```

二进制代码如下

```
10001100000000010000000000001010
10001100000000010000000000001011
10001100000000011000000000001100
00000000001000100010000000100110
00000000001000100010100000000100
00111100000000001111111111111111
```

4. 第四组指令主要用于测试一个程序的完整运行情况

```
lw $4 10($0)
sw $2 10($0)
add $3,$4,$2
sub $5,$1,$3
and $6,$3,$5
or $7,$5,$3
addi $0,$3,8
andi $1,$3,11
ori $3,$0,32
beq $31,$30,4
sll $1$,$3$,2
srl $7,$1,3
slt $10,$2,$3
beq $1,$31,1
j   16
jal 10
jr  $30
```

二进制代码如下

```
10001100000001000000000000001010
10101100000000010000000000001010
00000000100000100001100000100000
00000000001000110010100000100010
00000000011001010011000000100100
00000000101000110011100000100101
00100000011000000000000000001000
00110000011000010000000000001011
00110100000000011000000000010000
00010011110111110000000000000101
00000000000000011000010001000000
00000000000000001001110001100010
00000000100000110101000000101010
00010000001111110000000000000000
00001000000000000000000000010000
00001100000000000000000000001010
00000011110000000000000000001000
```

3.2 仿真结果

接下来介绍四组指令的仿真结果

1. 第一组指令结果如图 1所示。可以看见前三条指令把数据读进来，第四条指令和前面的 lw 指令之

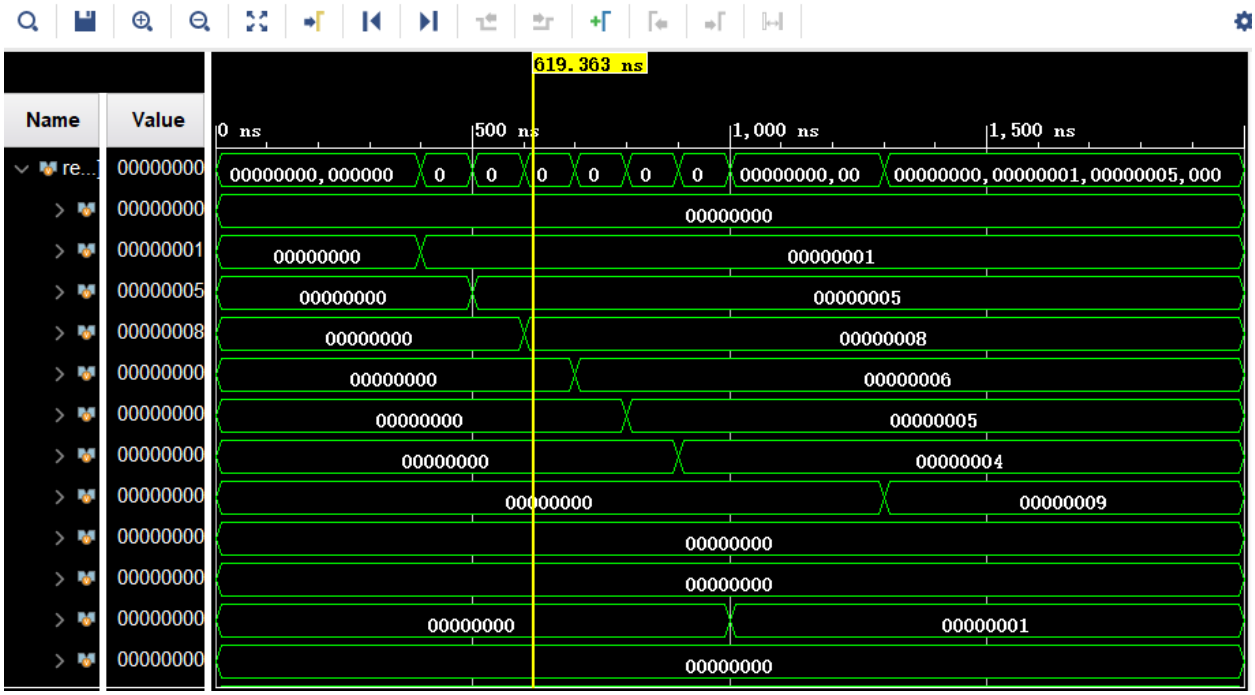

图 1: 第一组指令仿真结果

间有相关的数据依赖关系，但是通过前向通路可以成功的将结果 $2 的内容直接传过来，然后第五条指令和第六条指令都和前面的一条或者两条指令之间存在着数据依赖关系。但是能够将结果 6-1=5 和之后的 5&6=4 的结果计算出来。说明这里我们的前向通路的设计和多周期流水线的设计是成功的。

接下来的 or 指令和 slt 指令也能够返回正确的结果，最后测试 beq 指令可以看见这里 stall 机制是奏效的因为否则话就会执行后面一条的 and 指令，这里没有执行。

通过第一组测试，能够实现多周期流水线处理，forwarding,stall 机制的验证。

2. 第二组指令主要用来测试跳转指令是否能够正确运行，这里我们验证的时候就不观察寄存器的内容了，直接看一下 ID 阶段获得的指令的内容为了方便查看我们先把二进制的指令内容变成 16 进制的。

```
10200004
001BF080
001DC0C2
03BCA82A
12800001
08000001
0C000007
00200008
```

如果 CPU 能够正常运行首先会按照跳转指令先执行 beq 指令然后到第 6 条 jump 指令然后跳到第二条指令顺序执行，然后执行第五条 beq 指令然后再跳转到第七条指令，然后跳转到最后一条

指令，最后先会按照跳转指令先执行 beq 指令然后到第 6 条 jump 指令然后跳到第二条指令顺序执行，然后执行第五条 beq 指令然后再跳转到第七条指令，然后跳转到最后一条指令，最后因为要跳转到 register 0 里面存储的指令地址，因此再跳回到第一条指令的地方。依次循环。这一组指令的运行结果如图 2。我们看运行结果，在第一条 beq 指令之后跟了一条空指令说明这里可以

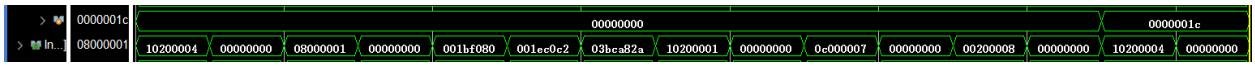

图 2: 第二组指令运行结果

成功的进行 stall 然后接下来一个周期就可以跳转到我们的目标地址，因为是 Jump 地址所以这里也进行了 stall，然后也成功的跳转过去。几条指令以后又是碰到 branch 指令成功进行 stall 并且跳转，最后的顺次剩下的几条跳转指令我们看见也都能够成功执行。最后我们可以看见 jal 指令也可以把地址成功写入第 31 号寄存器内。至此能够验证这个 CPU 对于跳转指令是可以成功运行的。

3. 第三组指令的运行结果如图 3所示，能够成功进行抑或运算，左移运算和 lui 操作。

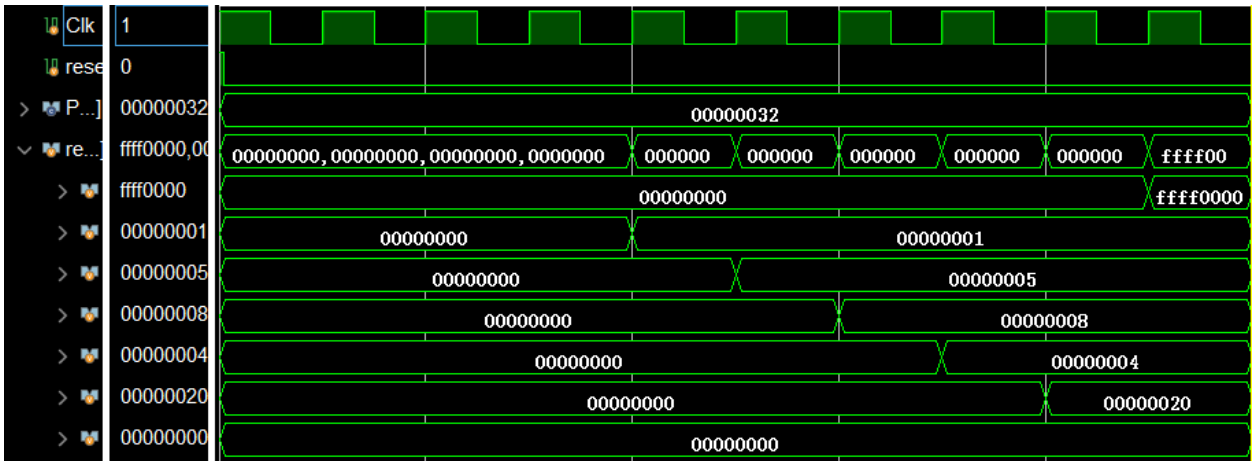

图 3: 第三组指令运行结果

4. 第四组指令的运行结果如图 4所示，能够成功运行。可以分析一下指令运行的结果，可以看见我

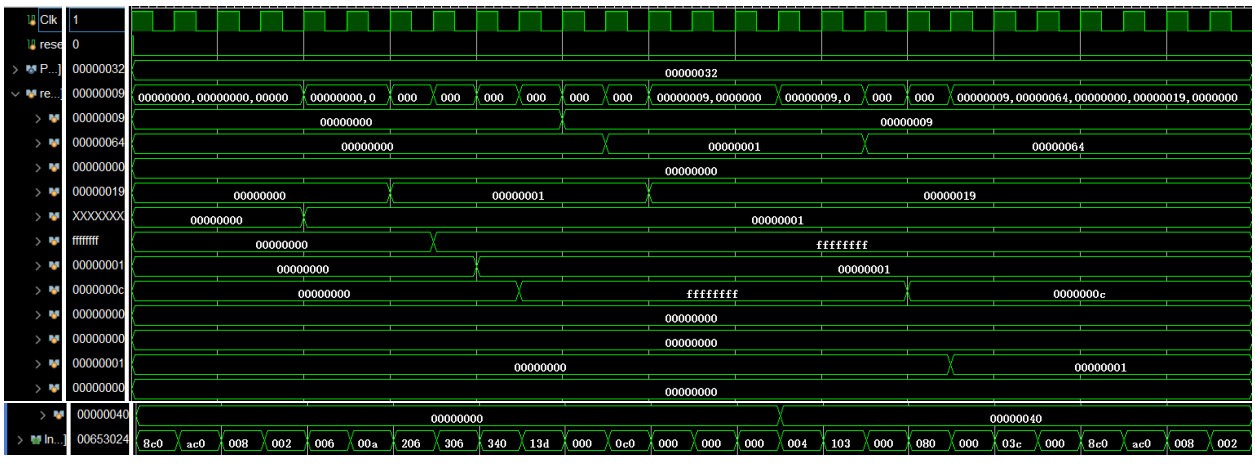

图 4: 第四组指令运行结果

们的这一组测试代码相互之间存在很多的数据相关性，很多都是前面一条算后面一条就要使用了，

有不少 RAW。但是还是可以成功的在第一条指令读取之后按照这些数字时间的关系进行相应的运算并且给出正确的结果，到了后面的 beq 指令能够成功的跳转到后面的 jal 指令并且我们可以看到在 reg 31 中也写入了对应的地址 40，然后我们又可以成功执行三条运算指令。然后进行第二个 beq 比较，这次比较分支条件不成立因此继续指令了跳转指令，并且最终执行最后一条 jr 指令。因为 reg 30 存储的地址是 0，因此又开始循环整个代码，中间所有指令都能够成功运行。完成一个完整的程序测试!

4 心得体会

这次实验完成了一个类 MIPS 多周期指令的 CPU 的设计。可以说不仅难度很高而且体量巨大。可以说实现起来还是非常的有难度的。

在实现的过程有很多的细节是要注意的，而且更多是编写代码的操作细节，在处理流水线寄存器的时候就要注意时序逻辑，一开始的时候写成了组合逻辑然后结果不对，需要使用时序逻辑的赋值方式这样才可以保障周期运行的正确性。虽然实现出来没有多少代码但是在写的时候要思考这个变量是应该放到时序逻辑里还是普通的组合逻辑就可以了。每一个流水线寄存器需要那些输入那些输出，虽然最后出来代码以看就是这些，但是在做的过程中都要思考其实也花费了不少的时间。

除此之外就像在前面说到的 Top 模块中的排线，虽然看上去没什么内容但是实际在思考怎么接线的时候还是花费了很大量的功夫进行操作整理。有很多的细节工作一定要仔细，这一部分关键的一定要有一个合理的命名变量的方法，否则后面是真的要晕过去。并且最重要的是在接线的时候不能有疏漏或者接错，在实现的过程中一开始接错了线，然后检查了很长的时间为什么 pcmux 为什么接受到的 branch 永远都是 0。

除此之外还有一个细节，就是所有的模块都需要写初始化内容，将所有的模块里面可能的变量都赋值为如果是 1 条空指令结果是什么。因为如果不初始化的话会输出 X，然后这些 X 又全部传回给 PC，后面就出不来正确的内容了，所以需要初始化。

在设计的时候还有一个关于结构冒险问题的细节，就是需要安排寄存器和内存单元能够先写后读的情况，因此要重新安排在时钟上沿进行写，而在时钟下沿进行读操作。并且其中还存在一些时序逻辑的问题。因此在一开始进行实现的时候会出现不能够正确读写结果的问题。但其实这里是不需要 care 内存的因为不存在流水线两条指令在同一个周期对内存进行同时读写结构（哈佛结构）。对于不能够正确读写，发现是对于时钟周期的安排问题，因此还需要进行适当的修改以后才可以正确运行。

除了流水线寄存器还需要把很多方方面面都考虑如何修改或者添加新的模块满足条件等等。而且在最后模拟的过程中也不是一帆风顺的，可能一会儿这里有问题要回过头去修改，哪里改了又会出现别的 Bug 等等。在实践的过程中一开始处理最基本的多周期还算成功，主要是一些前述的细节性问题。而在之后设计 stall 机制的时候出现了不能够正确检测的情况，以及持续性的输出空指令而不运行下一条指令的现象，因此需要相应的进行修改。而之后在设计 forwarding 机制的时候，因为流水线寄存器是一个时序模块，因此如何处理数据的传递调试了很长的时间，并且里面的逻辑也是有点绕的。在处理如何读到上一轮的 aluResult 并且传递下去，在这里处理的时候并不能够直接把 aluResult 输出，因为很可能发生的事情就是把这一轮的计算结果循环当成之前的结果的一种错误操作。因此要使用一个局部的 reg 变量来保存结果才能够保障结果的正确性。而之后又发现了对于 R 指令，有两条 forwarding 线路，因此还要更加细致的设计方法保存上上周期的结果。最后处理 predict-not-taken 机制的时候相

对来说比较顺利。

除了这些最后再整个代码调试的时候也遇到了一些问题，比如说一开始只考虑 forwarding 机制中 aluResult forwarding 的时候要判断前面一条语句不是 load 语句，但是忘记了也要保障前面一条语句不是 store 语句。在之后发现条件不成立的时候之间跳转到 pc+8 的语句，因此对 pcadd4 模块进行了改进。而最后对于 jr 指令不能够正确的跳转到寄存器所保存的地址处，仔细检查是 always 语句的条件存在一些问题。always 的条件需要涵盖可能的产生变化的变量，因为 always 语句内的各个语句都是并行执行的。，因此这里也是一个细节性的问题。

最后复习了一下汇编语言的编写和内容，并且还构思设计了一些指令测试。最后转换成二进制编码的时候也是眼花缭乱，感觉眼前全是 0101 差点晕过去。

总体来说这次实验极其得有难度，但是也很有收获，不仅巩固加深了所学的知识，也了解到前人是如何设计 CPU，虽然现代 CPU 以及有很大差异而且更加更加的复杂，但是通过这么一个实验能够了解到的内容和知识还是非常多的!

5 参考资料

2020 计算机系统结构实验指导书-LAB06_M.pdf
Verilog HDL 华为入门教程.pdf
夏宇闻 Verilog 教程.pdf
MIPS 指令集格式https://blog.csdn.net/u010385646/article/details/49516775