

计算机系统结构实验 Lab6实验报告

- 姓名：夏鸿驰
- 学号：520021910965
- 日期：2022年4月26日

目录

1. 实验概述
 - 1.1 实验名称
 - 1.2 实验目的
2. 实验步骤
 - 2.1 构建简单MIPS流水线CPU
 - 2.2 构建调试通过predict-not-taken解决控制冒险的MIPS流水线CPU
 - 2.3 构建调试增加Forwarding机制解决数据竞争的MIPS流水线CPU
 - 2.4 构建调试MIPS-31指令流水线CPU
 - 2.5 构建调试应用Cache原理设计加入Cache Line的MIPS流水线CPU
 - 2.6 构建调试更加强大的分支预测的MIPS流水线CPU（自己补充的）
3. 实验心得
 - 3.1 实验重难点
 - 3.2 实验感想
4. 参考资料

Notes:

具体代码过于冗长（1000-2000+行）不会全部放入报告，仅挑选有代表性的代码，具体见工程文件即可。

并且展示的代码仅仅是中间过程的代码，不代表是最终版本的代码，最终版本见工程文件夹/**lab06-master**。

实现的步骤可能与指导书有所出入：先实现通过predict-not-taken解决控制冒险，再实现Forwarding机制解决数据竞争，主要是为了调试的方便（有些跳转语句也有数据依赖）。

1. 实验概述

1.1 实验名称

- 简单的类 MIPS 多周期流水化处理器实现

1.2 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. 构建调试通过predict-not-taken解决控制冒险的MIPS流水线CPU
3. 构建调试增加Forwarding机制解决数据竞争的MIPS流水线CPU
4. 构建调试MIPS-31指令流水线CPU
5. 构建调试应用Cache原理设计加入Cache Line的MIPS流水线CPU
6. 增加协处理器CP0，支持中断相关机制及中断指令（未做）

2. 实验步骤

2.1 构建简单MIPS流水线CPU

2.1.1 创建流水线中间寄存器文件

- 组成部分：四个中间寄存器文件，用于传递数据和控制信号
- 主要接口：
 - reset信号：初始化
 - Clk信号：每个周期的开始结束信号
 - 上个阶段的数据和控制信号，output成为下个阶段的数据和控制信号
- 大致结构：
 - 内部寄存器
 - 在Clk下降沿进行的写入操作
- 具体文件：
 - IF_IDRegister.v: 代表作为在IF阶段和ID阶段中间的寄存器
 - 主要作用：传递当前PC值和Fetch得到的指令值。
 - ID_EXRegister.v: 代表作为在ID阶段和EX阶段中间的寄存器
 - 主要作用：
 - 将ID阶段Ctr译码出的控制信号传递给EX阶段
 - 将PC更新所需要的数据（如ID_NextSeqPC, ID_JumpAddr）传递给后续阶段
 - 将ID阶段读出的寄存器值传递给EX阶段
 - 将符号拓展数据和零扩展数据传递给EX阶段
 - 将当前指令的rt,rd传递给EX阶段，用于生成将要写入的寄存器序号
 - EX_MEMRegister.v: 代表作为在EX阶段和MEM阶段中间的寄存器
 - 主要作用：
 - 将EX阶段还未用到的Ctr译码出的控制信号传递给MEM阶段
 - 将PC更新所需要的数据（如EX阶段计算出的EX_BranchAddr, EX_NextSeqPC, EX_JumpAddr）传递给后续阶段
 - 将MEM阶段可能需要写入内存中的数据EX_WriteData和可能作为地址的EX_ALURes传递给后续MEM阶段
 - 将EX阶段生成将要写入的寄存器序号传递给后续MEM阶段
 - MEM_WBRegister.v: 代表作为在MEM阶段和WB阶段中间的寄存器
 - 主要作用：
 - 将EX, MEM阶段还未用到的Ctr译码出的控制信号传递给WB阶段
 - 将MEM阶段读出的内存数据MEM_ReadData传递给后续的WB阶段

2.1.2 创建顶层文件Top_Pipeline连接各个流水线阶段

- 组成部分：
 - 各实例化的模块文件
 - 连接各个模块文件的wire和reg
- 主要接口：
 - reset信号和Clk信号
- 具体结构（若未贴出完整代码，说明和lab5中一致或相近）：
 - IF阶段：

- 根据当前PC值，从InstrMemory读出指令
- 直接将下一个PC值设置为PC+4，**待下一实验步骤解决控制冒险问题**

■ 1 | wire [31:0] IF_NextPC = IF_CurPC + 4;

- 将后续阶段需要的数据和控制信号放入IF_IDRegister

○ ID阶段：

- 根据OpCode在Ctr模块译码得到控制信号
- 实例化寄存器文件Registers，注意此时用到的部分信号是来自WB阶段的，因为从读到写中间相隔多个周期

```
■ 1 //Registers
2     reg [4:0] WB_writeReg;          //From WB
3     wire [31:0] WB_writeData;      //From WB
4     wire [31:0] ID_readData1;
5     wire [31:0] ID_readData2;
6     wire WB_RegWrite;            //From WB
7     wire WB_jr;                  //From WB
8     Registers Registers(
9         .clk(clk),
10        .readReg1(ID_Instr[25:21]),
11        .readReg2(ID_Instr[20:16]),
12        .writeReg(WB_writeReg),
13        .writeData(WB_writeData),
14        .regwrite(WB_RegWrite),
15        .jr(WB_jr),
16        .reset(reset),
17        .readData1(ID_readData1),
18        .readData2(ID_readData2)
19    );
```

- 符号拓展和零拓展
- 得到jump指令的jumpaddr
- 将后续阶段需要的数据和控制信号放入ID_EXRegister

○ EX阶段：

- 通过控制逻辑得到当前指令的ALU操作数，并且直接设置为当前阶段ALU的操作数，**待后续步骤通过forwarding解决数据冒险问题**（代码与lab5一致）。
- 通过控制逻辑得到当前指令将要写入的寄存器序号。
- 通过ALU计算得到ALURes。
- 通过PC相关控制逻辑得到分支跳转地址EX_BranchAddr。
- 将后续阶段需要的数据和控制信号放入EX_MEMRegister。

○ MEM阶段：

- 通过DataMemory写入或读出相关数据。
- 通过PC相关控制逻辑得到真正的下一PC值，**用于后续步骤解决结构冒险问题**。
- 将后续阶段需要的数据和控制信号放入MEM_WBRegister。

○ WB阶段：

- 通过WB_MemToReg信号选择将要写入的数据。
- 通过在先前已经实例化的寄存器文件中的相关信号实现数据的写入寄存器。

2.2 构建调试通过predict-not-taken解决控制冒险的MIPS流水线CPU

2.2.1 Top_Pipeline顶层文件的初步修改

- 在IF阶段直接predict-not-taken：

```
o 1 | wire [31:0] IF_NextPC = Bubble? Correct_Next_PC : IF_CurPC + 4;  
    //Sequential by default
```

2.2.2 创建新模块文件Ctr_Hazard_Handler.v用于解决结构冒险

- 模块组成：

- 输入信号：

- 当前IF阶段predict的下一条指令的PC (Predict_Next_PC)
 - 当前MEM阶段计算出的前第三条指令真实的下一条指令PC (Real_Next_PC)
 - Clk和reset常规信号

- 输出信号：

- Bubble: 是否产生bubble应对预测分支错误 (清空ID_EX, EX_MEM流水线中间寄存器)
 - Correct_Next_PC: 输出给IF阶段，用于在预测错误时的重新抓取正确位置指令。

- 内部结构：

- 一个历史PC预测表：

- 1 | reg [31:0] History_Predict_PC[0:2];

- 一个指针ptr，指向当前周期将要写入的预测地址的地址和前三个周期的预测跳转指令地址的地址（与当前输入的Real_Next_PC比较）
 - 整型数据recover_cycle：预测失败恢复过程中禁止再次触发比较的控制信号

- 具体代码（部分）：

```
o 1 //写入预测的PC  
2     always @(negedge Clk)  
3         begin  
4             History_Predict_PC[ptr] = Predict_Next_PC;  
5             ptr = (ptr + 1) % 3;  
6             recover_cycle = (recover_cycle > 0)? recover_cycle - 1 : 0;  
7         end  
8  
9     //前三个周期的预测跳转指令地址和与当前输入的Real_Next_PC的比较  
10    //产生bubble信号和设置recover_cycle  
11    always @(Real_Next_PC)  
12        begin  
13            if(Real_Next_PC != History_Predict_PC[ptr] &&  
14            recover_cycle == 0) begin  
15                bubble = 1;  
16                recover_cycle = 4;  
17            end  
18            else begin  
19                bubble = 0;  
20            end  
21        end  
22  
23     assign Correct_Next_PC = Real_Next_PC;
```

2.2.3 在Top_Pineline顶层文件再做出适配Ctr_Hazard_Handler模块的修改

- bubble状态下PC的强制置换

- 1 | wire [31:0] IF_NextPC = Bubble? correct_Next_PC : IF_CurPC + 4;
//Sequential by default

- Correct_Next_PC由后续实例化的Ctr_Hazard_Handler产生。

- ID_EX寄存器的bubble信号加入的实例化

- 1 | ID_EXRegister ID_EXRegister(
2 | ...omitted
3 | .Bubble(Bubble),
4 |
5 | .clk(clk),
6 | .reset(reset),
7 | ...omitted
8 |);

- EX_MEMORY寄存器的bubble信号加入的实例化

- 1 | EX_MEMORYRegister EX_MEMORYRegister(
2 | ...omitted
3 |
4 | .Bubble(Bubble),
5 |
6 | .clk(clk),
7 | .reset(reset),
8 |
9 | .MEM_MemToReg(MEM_MemToReg),
10 | .MEM_RegWrite(MEM_RegWrite),
11 | .MEM_writeReg(MEM_writeReg),
12 |
13 | ...omitted
14 |);

- MEM阶段中下一PC的计算 (其实和lab5一致，但是后续要用到) :

- 1 | //MEM-PC
2 | wire [31:0] MEM_PreNextPC1;
3 | assign MEM_PreNextPC1 = (MEM_Branch&&MEM_zero)?
MEM_BranchAddr:MEM_NextSeqPC;
4 | wire [31:0] MEM_PreNextPC2;
5 | assign MEM_PreNextPC2 = MEM_jr? MEM_readData1:MEM_PreNextPC1;
6 | wire [31:0] MEM_NextPC;
7 | assign MEM_NextPC = MEM_Jump? MEM_JumpAddr:MEM_PreNextPC2;

- Ctr_Hazard_Handler的实例化:

```

o 1 //For Control Hazard:
2   Ctr_Hazard_Handler Ctr_Hazard_Handler(
3     //这里用了IF阶段预测的下一PC值作为输入
4     .Predict_Next_PC(IF_NextPC),
5     //这里用了MEM阶段计算得到的真实下一PC值作为输入
6     .Real_Next_PC(MEM_NextPC),
7     .Clk(clk),
8     .reset(reset),
9     //输出Bubble信号
10    .Bubble(Bubble),
11    //输出正确下一PC数据给IF阶段
12    .Correct_Next_PC(Correct_Next_PC)
13  );

```

2.2.4 在流水线中间寄存器加入Bubble输入信号支持解决控制冒险

- 正如上面所述，ID_EX寄存器和EX_MEM寄存器需要在预测错误时接收到Bubble信号并清零，所以需要做出一些修改。
- 这里以ID_EX寄存器为例，EX_MEM类似。
- 在Clk的下降沿时，判断输入信号Bubble是否为1，若是，则其内部所有寄存器全部清零，如下：

```

• 1 always @(negedge clk)
2   begin
3     if(Bubble == 1) begin
4       RegDst = 0;
5       ALUSrc1 = 0;
6       ALUSrc2 = 0;
7       MemToReg = 0;
8       RegWrite = 0;
9       MemRead = 0;
10      Memwrite = 0;
11      Branch = 0;
12      ALUOp = 0;
13      Jump = 0;
14
15      CurPC = 0;
16      NextSeqPC = 0;
17      JumpAddr = 0;
18      readData1 = 0;
19      readData2 = 0;
20      SignExtImm = 0;
21      ZeroExtImm = 0;
22
23      rd = 0;
24      rt = 0;
25    end
26    else begin
27      RegDst = ID_RegDst;
28      ALUSrc1 = ID_ALUSrc1;
29      ALUSrc2 = ID_ALUSrc2;
30      MemToReg = ID_MemToReg;
31      RegWrite = ID_RegWrite;
32      MemRead = ID_MemRead;
33      Memwrite = ID_MemWrite;
34      Branch = ID_Branch;
35      ALUOp = ID_ALUOp;

```

```

36         Jump = ID_Jump;
37
38         CurPC = ID_CurPC;
39         NextSeqPC = ID_NextSeqPC;
40         JumpAddr = ID_JumpAddr;
41         readData1 = ID_readData1;
42         readData2 = ID_readData2;
43         SignExtImm = ID_SignExtImm;
44         ZeroExtImm = ID_ZeroExtImm;
45
46         rd = ID_rd;
47         rt = ID_rt;
48     end
49 end

```

- 从而凭此可以解决控制冒险中错误语句的问题：禁止了任何写信号，避免对任何状态的错误影响。

2.2.5 仿真和调试

加入一段没有任何数据冒险的代码，专门用于调试控制冒险的解决，如下：

```

1 lw $1,12($0)
2 lw $2,17($0)
3 add $0, $0, $0
4 add $0, $0, $0
5 add $0, $0, $0
6 add $3,$1,$2
7 sub $4,$1,$2
8 and $5,$1,$2
9 or $6,$1,$2
10 slt $7,$1,$2
11 slt $8,$2,$1
12 beq $1,$2,1
13 add $9,$0,$1
14 beq $1,$1,1
15 add $10,$0,$2
16 add $10,$0,$1
17 sw $5,-4($2)
18 j 22
19 or $11,$0,$1
20 or $11,$0,$1
21 or $11,$0,$1
22 or $11,$0,$1
23 or $12,$0,$1
24 addi $13,$8,100
25 addi $14,$9,-40
26 andi $15,$10,57
27 andi $16,$10,-32
28 ori $17,$12,63
29 ori $18,$12,-43
30 sll $19,$13,14
31 srl $20,$14,9
32 jal 35
33 or $21,$0,$1
34 or $21,$0,$1
35 or $21,$0,$1
36 or $22,$0,$2

```

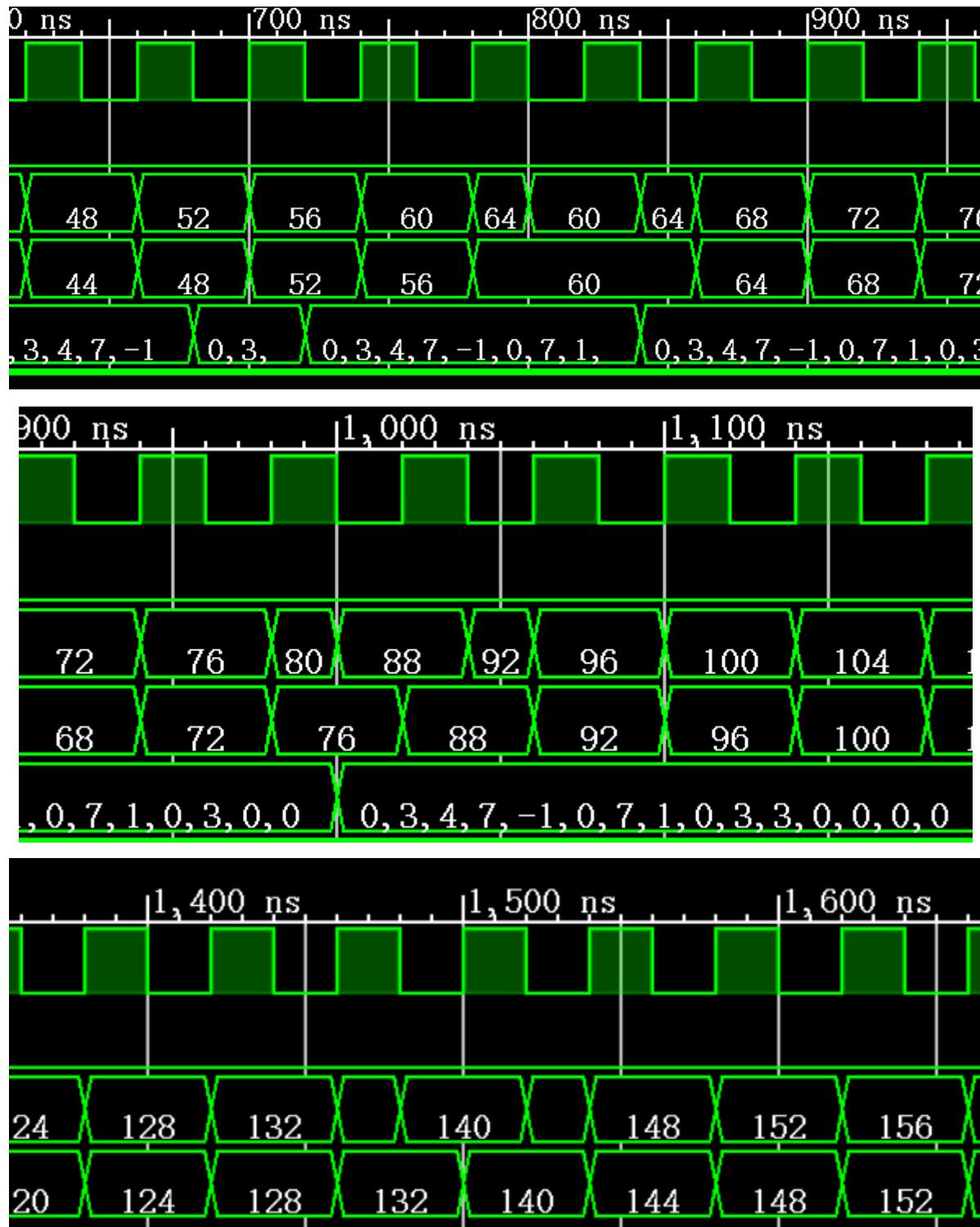
```

37 ori $23,$0,156
38 add $0, $0, $0
39 add $0, $0, $0
40 add $0, $0, $0
41 jr $23
42 or $24,$0,$1
43 or $25,$0,$2

```

具体的机器码省略，然后用之前lab5一直采用相同内存初始化，进行仿真，结果如下：

- Ctr_Hazard_Handler相关：



- 一次预测恢复的具体情况：

Name	Value	700 ns	750 ns	800 ns	850 ns
Clk	0				
reset	0				
> NextPC[31:0]	52				
> CurPC[31:0]	48				
> Predict_Next_PC[31:0]	52	52	56	60	64
> Real_Next_PC[31:0]	44	48	52	56	60
Bubble	0	40	44	48	52
> Correct_Next_PC[31:0]	44	40	44	48	52
> History_Prediction_C[0:2][31:0]	52,44,48	52,44,48	52,56,48	52,56,60	64,56,60
> [0][31:0]	52	40	52		64
> [1][31:0]	44	44		56	
> [2][31:0]	48		48		60
> ptr[31:0]	1	0	2	0	1
> recover_cycle[31:0]	0		0	4	3

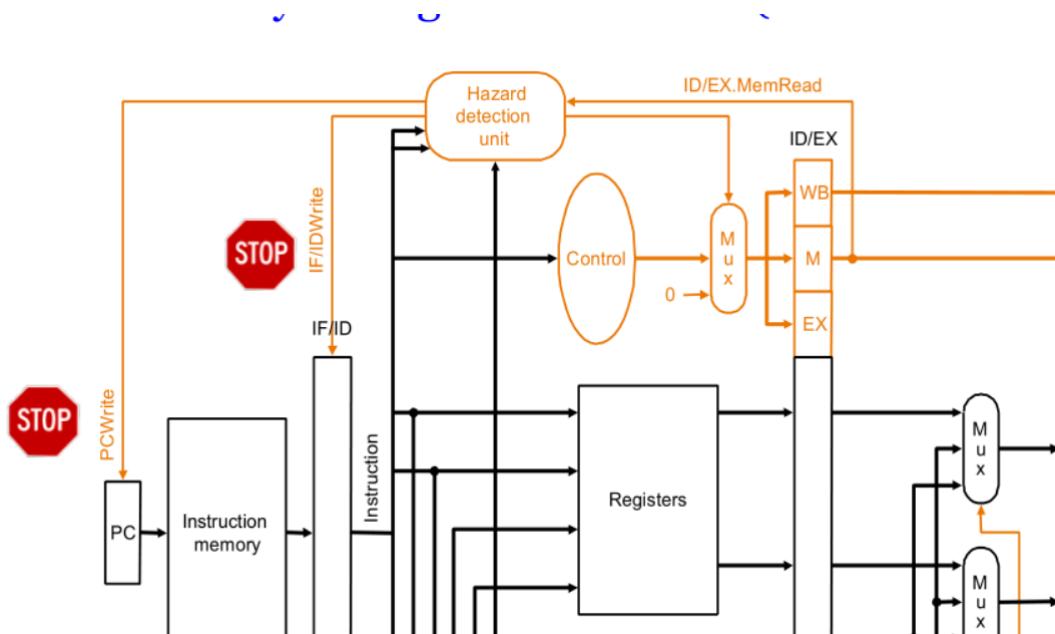
- PC=52时的beq语句需要跳转，在预测错误后Bubble信号置为1，并进行了错误恢复，采用了Correct_Next_PC，从而恢复正确执行逻辑。
- 可见当预见分支语句后，若预测失败，对应的PC值有正确的恢复过程。
- 局限性：**
 - 对于一些不必预测，百分百跳转的语句，如j, jal, jr，由于在IF过程中没有对应的译码元器件，导致其被迫需要使用这种低效的恢复手段来实现语句的正确执行，**将在后续步骤进行改进**。
 - 对于可选的分支跳转语句，仅仅可以预测不跳转，不能预测跳转，使程序变得可能的低效，**将在后续步骤进行改进**。

2.3 构建调试增加Forwarding机制解决数据竞争的MIPS流水线CPU

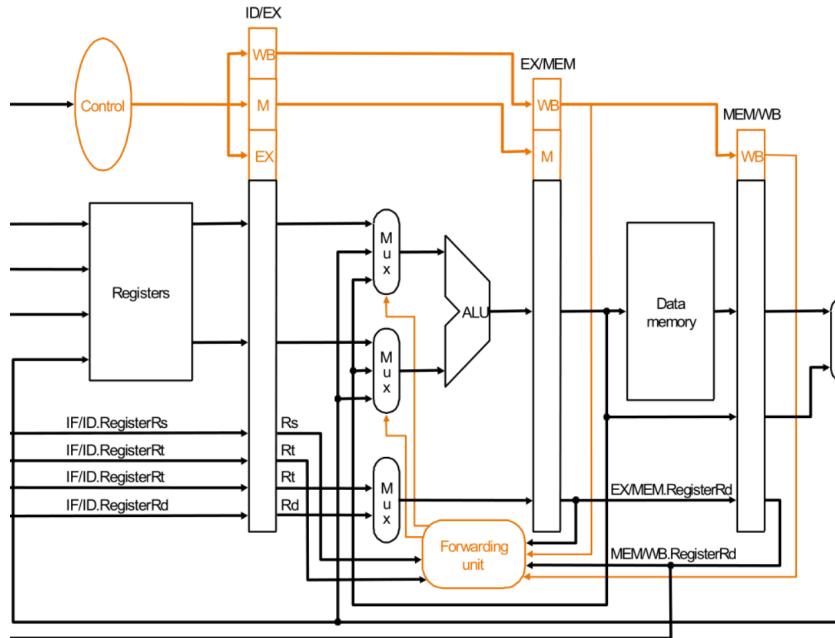
2.3.1 主要原理

Data_Hazard_Handler分为两个部分，其一是Hazard Detection Unit，解决Load-use Case下不可避免的bubble；其二是Forwarding Unit，解决Forwarding。

Hazard Detection Unit:



Forwarding Unit:



2.3.2 增加Data_Hazard_Handler进行forwarding和必要的load-use case下的bubble

- 输入输出的信号（具体作用见注释）：

```

o   1 input clk,
  2 input reset,
  3
  4 //For Forwarding:
  5 //检测此时EX阶段指令需要的寄存器
  6 input [5:0] EX_OpCode,
  7 input [5:0] EX_Funct,
  8 //检测此时MEM阶段和WB阶段是否会覆盖当前需要的寄存器
  9 input MEM_regWrite,
 10 input [4:0] MEM_writeReg,
 11 input WB_regWrite,
 12 input [4:0] WB_writeReg,
 13 input [4:0] EX_rs,
 14
 15 //ALU输入操作数数据的选择：从ID_EX寄存器，上一指令ALU产生的结果EX_MEMaluRes
 16 和上上条指令从内存中读出的MEM_WBwriteData进行三选一
 17 //00-default 01-EX_MEMaluRes, 10-MEM_WBwriteData
 18 output [1:0] Input1_Sel,
 19 output [1:0] Input2_Sel,
 20 output [1:0] sw_sel,
 21 output [1:0] jr_sel,
 22
 23 //For Load-Use Case:
 24 //判断此时在ID阶段的前一条指令是否是Load指令
 25 input EX_MemRead,
 26 //判断是否存在数据依赖
 27 input [4:0] EX_rt,
 28 input [31:0] ID_Instr,
 29 //输出必要的stall信号，不同于上面的Bubble信号
 30 output D_Bubble

```

- 具体代码（仅挑选有代表性的）：

- For Load-Use Case:

```

if(EX_MemRead == 1 && EX_rt != 0)begin
    casex({ID_Instr[31:26],ID_Instr[5:0]})
        12'b100011xxxxxx: //lw
            begin
                //rs
                if(ID_Instr[25:21] == EX_rt) d_bubble = 1;
                else d_bubble = 0;
            end
        12'b000000xxxxxx: //remaining r-type
            begin
                //rs rt
                if(ID_Instr[25:21] == EX_rt || ID_Instr[20:16] == EX_rt) d_bubble = 1;
                else d_bubble = 0;
            end
end

```

- For forwarding:

- lw的逻辑: 解释input1_sel、input_2sel (类似作用) :

```

12'b100011xxxxxx: //lw
begin
    //rs
    if(EX_rs == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        input1_sel = 2'b01;
    end
    else if(EX_rs == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        input1_sel = 2'b10;
    end
    else begin
        input1_sel = 2'b00;
    end
    input2_sel = 2'b00;
    sw_sel_reg = 2'b00;
    jr_sel_reg = 2'b00;
end

```
- sw的逻辑: 解释sw_sel:

```

12'b101011xxxxxx: //sw
begin
    //rs rt
    //rs
    if(EX_rs == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        input1_sel = 2'b01;
    end
    else if(EX_rs == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        input1_sel = 2'b10;
    end
    else begin
        input1_sel = 2'b00;
    end
    //rt-sw
    if(EX_rt == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        sw_sel_reg = 2'b01;
    end
    else if(EX_rt == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        sw_sel_reg = 2'b10;
    end
    else begin
        sw_sel_reg = 2'b00;
    end
    input2_sel = 2'b00;
    jr_sel_reg = 2'b00;
end

```
- jr的逻辑: 解释jr_sel:

```

12'b000000001000: //jr
begin
    //rs
    if(EX_rs == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        jr_sel_reg = 2'b01;
    end
    else if(EX_rs == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        jr_sel_reg = 2'b10;
    end
    else begin
        jr_sel_reg = 2'b00;
    end
    input1_sel = 2'b00;
    input2_sel = 2'b00;
    sw_sel_reg = 2'b00;
end

```

2.3.3 其他模块文件相适配的改进

- 将寄存器文件Register写入的时刻从Clk的下降沿提前至Clk的上升沿，从而更好的适配Forwarding:

```

o 1 always @(posedge clk)
  begin
  2     if(regwrite&&!jr&&writeReg!=0) regFile[writeReg] =
  3         writeData;
  4     end

```

- 将Ctr_Hazard_Handler中添加入D_Bubble的输入，并设置相应的recover_cycle:

```

o 1 always @(D_Bubble) begin
  2     if(D_Bubble == 1) recover_cycle = 2;
  3 end

```

- 在PC和IF_IDRegister中的写入操作前加入D_Bubble的判断:

```

o 1 //PC:
  2 always @ (posedge clk)
  3 begin
  4     if(D_Bubble == 0) PCReg = NextPC;
  5 end

```

```

o 1 //IF_IDRegister:
  2 always @ (negedge clk)
  3 begin
  4     if(D_Bubble == 0) begin
  5         Instr = IF_Instr;
  6         CurPC = IF_CurPC;
  7         NextSeqPC = IF_NextSeqPC;
  8     end
  9 end

```

- 在ID_EXRegister中加入D_Bubble信号，用于清零部分控制信号:

```

o 1 else begin
  2     if(D_Bubble == 0) begin
  3         RegDst = ID_RegDst;
  4         ALUSrc1 = ID_ALUSrc1;
  5         ALUSrc2 = ID_ALUSrc2;
  6         MemToReg = ID_MemToReg;
  7         Regwrite = ID_RegWrite;

```

```

8      MemRead = ID_MemRead;
9      MemWrite = ID_MemWrite;
10     Branch = ID_Branch;
11     ALUOp = ID_ALUOp;
12     Jump = ID_Jump;
13
14     readData1 = ID_readData1;
15     readData2 = ID_readData2;
16     SignExtImm = ID_SignExtImm;
17     ZeroExtImm = ID_ZeroExtImm;
18
19     rd = ID_rd;
20     rt = ID_rt;
21     rs = ID_rs;
22
23     OpCode = ID_OpCode;
24     Funct = ID_Funct;
25
26     CurPC = ID_CurPC;
27     NextSeqPC = ID_NextSeqPC;
28     JumpAddr = ID_JumpAddr;
29 end
30
31 else begin
32     RegDst = 0;
33     ALUSrc1 = 0;
34     ALUSrc2 = 0;
35     MemToReg = 0;
36     RegWrite = 0;
37     MemRead = 0;
38     MemWrite = 0;
39     Branch = 0;
40     ALUOp = 0;
41     Jump = 0;
42
43     readData1 = 0;
44     readData2 = 0;
45     SignExtImm = 0;
46     ZeroExtImm = 0;
47
48     rd = 0;
49     rt = 0;
50     rs = 0;
51
52     OpCode = 0;
53     Funct = 0;
54
55     CurPC = ID_CurPC;
56     NextSeqPC = ID_NextSeqPC;
57     JumpAddr = ID_JumpAddr;
58 end
59
60 ...omitted

```

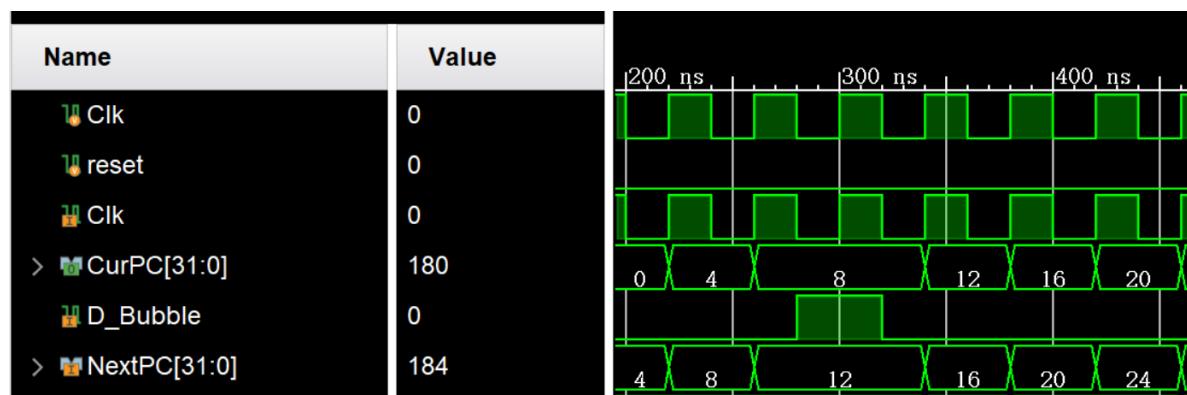
2.3.4 仿真和调试

采用了与先前结构相类似的汇编代码，不过在每个代码中都加入了很多数据依赖，以用于检测数据冒险是否被处理成功：

```
1 lw $1,12($0)
2 lw $2,17($0)
3 add $3,$1,$2
4 sub $4,$2,$3
5 and $5,$3,$4
6 or $6,$4,$5
7 slt $7,$6,$5
8 slt $8,$7,$6
9 beq $1,$2,1
10 add $9,$0,$8
11 beq $9,$9,1
12 add $10,$8,$7
13 add $10,$8,$8
14 sw $10,-1($10)
15 j 19
16 or $11,$0,$1
17 or $11,$0,$1
18 or $11,$0,$1
19 or $11,$0,$1
20 or $12,$0,$1
21 addi $13,$12,100
22 addi $14,$13,-40
23 andi $15,$14,57
24 andi $16,$15,-32
25 ori $17,$16,63
26 ori $18,$17,-43
27 sll $19,$18,14
28 srl $20,$19,9
29 jal 32
30 or $21,$0,$1
31 or $21,$0,$1
32 or $21,$0,$1
33 or $22,$0,$2
34 ori $23,$0,144
35 jr $23
36 or $24,$0,$1
37 or $25,$0,$2
```

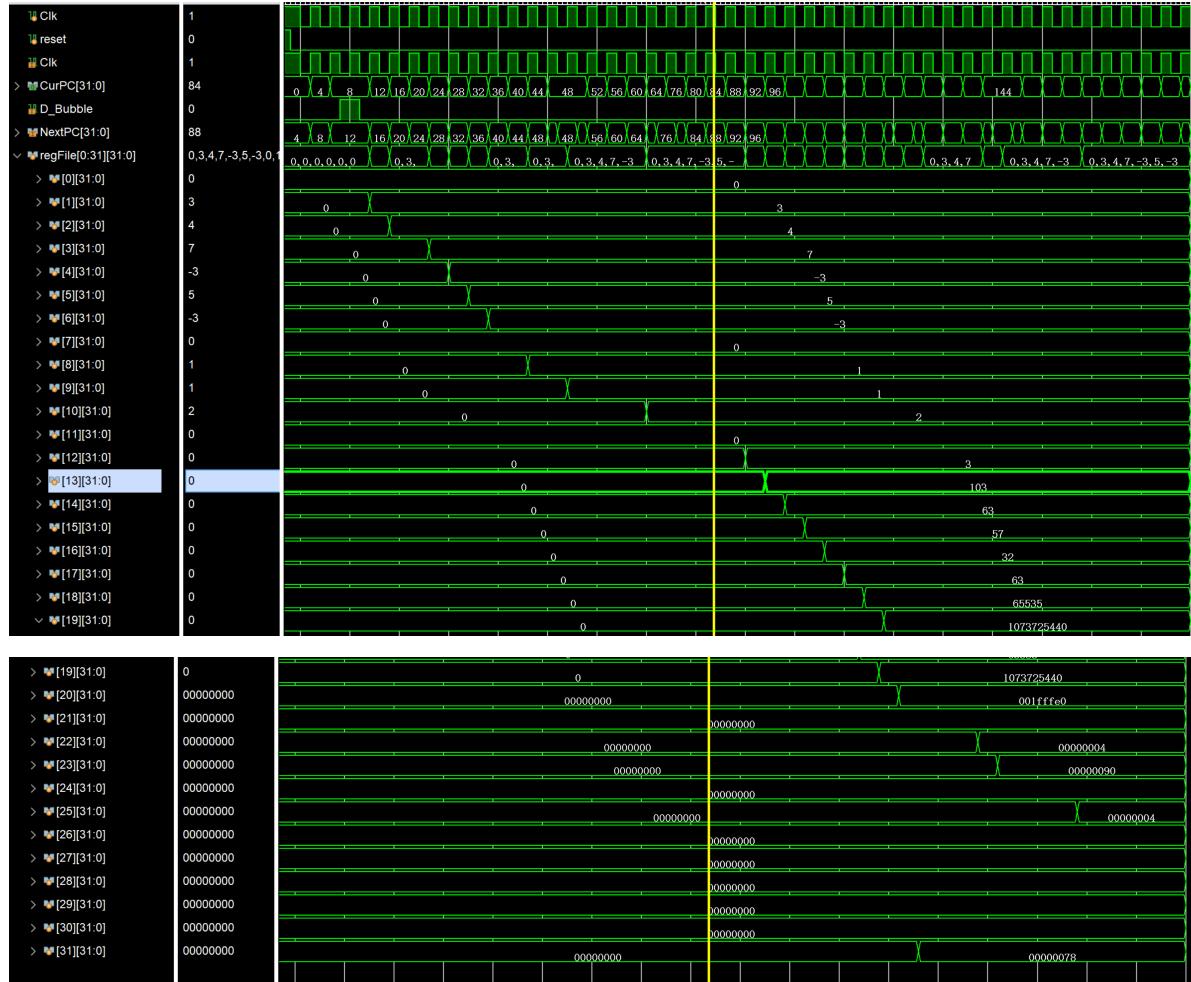
对应的内存初始化与先前一致，进行仿真：

对于Hazard Detection Unit:



检测到Load-use的情况并设置了D_Bubble，推迟了一个周期的指令执行。

对于Forwarding，得到寄存器被写的情况，如下：



符合数据的正确运算结果，故Data Hazard被成功解决。

注：此处的slt仍然是针对无符号数的比较，到下面mips31指令流水线后，slt被修改为针对有符号数的比较，而sltu是针对无符号数的比较。所以最终提交的版本中，slt和sltu的定义和运算均是正确的。此处的关注点应该在Data Hazard的解决方案上，正如上面展现的那样，数据都被正确地Forwarding了。

2.4 构建调试MIPS-31指令流水线CPU

2.4.1 添加的新指令

CORE INSTRUCTION SET NAME, MNEMONIC	FOR- MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add Imm. Unsigned addiu	I	$R[rt] = R[rs] + \text{SignExtImm}$	(2) 9 _{hex}
Add Unsigned addu	R	$R[rd] = R[rs] + R[rt]$	0 / 21 _{hex}
Branch On Equal beq	I	$\text{if}(R[rs]==R[rt])$ $PC=PC+4+\text{BranchAddr}$	(4) 4 _{hex}
Load Byte Unsigned lbu	I	$R[rt]=\{24'b0,M[R[rs]$ $+ \text{SignExtImm}(7:0)\}$	(2) 24 _{hex}
Load Halfword Unsigned lhu	I	$R[rt]=\{16'b0,M[R[rs]$ $+ \text{SignExtImm}(15:0)\}$	(2) 25 _{hex}

Load Upper Imm.	lui	I	$R[rt] = \{imm, 16'b0\}$	f_{hex}
Nor	nor	R	$R[rd] = \sim(R[rs] \mid R[rt])$	$0 / 27_{hex}$
Set Less Than Imm.	slti	I	$R[rt] = (R[rs] < SignExtImm)? 1 : 0$ (2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	$R[rt] = (R[rs] < SignExtImm)? 1 : 0$ (2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$ (6)	$0 / 2b_{hex}$
Store Byte	sb	I	$M[R[rs]+SignExtImm](7:0) = R[rt](7:0)$ (2)	28_{hex}
Store Halfword	sh	I	$M[R[rs]+SignExtImm](15:0) = R[rt](15:0)$ (2)	29_{hex}
Store Word	sw	I	$M[R[rs]+SignExtImm] = R[rt]$ (2)	$2b_{hex}$
Subtract Unsigned	subu	R	$R[rd] = R[rs] - R[rt]$ (1) May cause overflow exception (2) SignExtImm = { 16{immediate[15]}, immediate } (3) ZeroExtImm = { 16{1b'0}, immediate } (4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 } (5) JumpAddr = { PC+4[31:28], address, 2'b0 } (6) Operands considered unsigned numbers (vs. 2's comp.)	$0 / 23_{hex}$

xor	000000	rs	rt	rd	00000100110	xor \$1,\$2,\$3	\$1=\$2 ^ \$3	rd <- rs xor rt ; 其中rs=\$2, rt=\$3, rd=\$1(异或)
sra	000000000000	rt	rd	shamt	000011	sra \$1,\$2,10	\$1=\$2>>10	rd <- rt >> shamt ; (arithmetic) 注意符号位保留其中rt=\$2, rd=\$1
sllv	000000	rs	rt	rd	00000000100	sllv \$1,\$2,\$3	\$1=\$2<<\$3	rd <- rt << rs ; 其中rs=\$3, rt=\$2, rd=\$1
srlv	000000	rs	rt	rd	00000000110	srlv \$1,\$2,\$3	\$1=\$2>>\$3	rd <- rt >> rs ; (logical) 其中rs=\$3, rt=\$2, rd=\$1
sraw	000000	rs	rt	rd	00000000111	sraw \$1,\$2,\$3	\$1=\$2>>\$3	rd <- rt >> rs ; (arithmetic) 注意符号位保留其中rs=\$3, rt=\$2, rd=\$1
xori	001110	rs	rt		immediate	andi \$1,\$2,10	\$1=\$2 ^ 10	rt <- rs xor (zero-extend)immediate ; 其中rt=\$1, rs=\$2

共20条指令，加上原来的16条指令，共36条指令。

2.4.2 模块文件的修改

主要是在Ctr.v, ALUCtr.v和ALU.v三个地方进行修改。

并在Data_Hazard_Handler进行对应的添加。

- Ctr:
 - 增加了新的控制信号MByte, 表明内存操作的位数:

```
■ 1 | output [1:0] MByte //new 00-word 01-half 10-byte
```

- 拓展了一位ALUOp，支持更多的译码：

```

■ 1 | output [3:0] ALUOP,
2 | //new 0000-add 0001-sub 0010-Rtype 0011-and
3 | //0100-or 0101-slt 0110-addu 0111-sltu 1000-lui 1001-xor

```

- 对应的译码工作代码见工程文件

- ALUCtr:

- 针对在Ctr新加入的ALUOp信号进行译码
- 创建新的ALUCtr信号，支持更多的ALU运算：

```

■ 1 | //0000-and 0001-or 0010-add 0011-sll
2 | //0100-srl 0101-jr 0110-sub 0111-slt
3 | //1000-sltu 1001-addu 1010-subu 1011-lui
4 | //1100-nor 1101-xor 1110-sra 1111-

```

- 对应的译码工作不难，见工程文件

- ALU:

- 针对新的运算类型，设计运算器，举例如下：

```

■ 1 | 4'b1011://lui
2 | begin
3 |     alures = {{input2[15:0]},16'b0};
4 |     if(alures == 0)
5 |         zero = 1;
6 |     else
7 |         Zero = 0;
8 |     Overflow = 0;
9 | end
10 | 4'b1110://sra
11 | begin
12 |     signed_input2 = input2;
13 |     alures = signed_input2 >> input1[4:0];
14 |     if(alures == 0)
15 |         zero = 1;
16 |     else
17 |         zero = 0;
18 |     Overflow = 0;
19 | end

```

- 针对无符号运算和有符号运算的区别，设置了溢出位Overflow和相应的判断语句，举例如下：

```

■ 1 | 4'b0010: //add
2 | begin
3 |     alures = input1 + input2;
4 |     if(alures == 0)
5 |         zero = 1;
6 |     else
7 |         zero = 0;
8 |     signed_input1 = input1;
9 |     signed_input2 = input2;
10 |    signed_result = alures;
11 |    if(signed_input1 > 0 && signed_input2 > 0 && signed_result
|      < 0

```

```

12      || signed_input1 < 0 && signed_input2 < 0 && signed_result
13      > 0) begin
14          Overflow = 1;
15      end
16      else begin
17          Overflow = 0;
18      end
19  end
20 4'b0110: //sub
21 begin
22     alures = input1 - input2;
23     if(alures == 0)
24         zero = 1;
25     else
26         zero = 0;
27     signed_input1 = input1;
28     signed_input2 = input2;
29     signed_result = alures;
30     if(signed_input1 > 0 && signed_input2 < 0 && signed_result
31     < 0
32     || signed_input1 < 0 && signed_input2 > 0 && signed_result
33     > 0) begin
34         Overflow = 1;
35     end
36     else begin
37         Overflow = 0;
38     end
39 end

```

- Data_Hazard_Handler:

- 仅仅是仿照之前的代码进行添加，举例如下：

```

12'b000101xxxxxx: //bne new
begin
    //rs rt
    //rs
    if(EX_rs == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        input1_sel = 2'b01;
    end
    else if(EX_rs == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        input1_sel = 2'b10;
    end
    else begin
        input1_sel = 2'b00;
    end
    //rt
    if(EX_rt == MEM_writeReg && MEM_regWrite == 1 && MEM_writeReg != 0) begin
        input2_sel = 2'b01;
    end
    else if(EX_rt == WB_writeReg && WB_regWrite == 1 && WB_writeReg != 0) begin
        input2_sel = 2'b10;
    end
    else begin
        input2_sel = 2'b00;
    end
    sw_sel_reg = 2'b00;
    jr_sel_reg = 2'b00;
end

```

- Top_Pipeline.v:

- 对上述进行修改的信号进行适配的修改和实例化
- 拓宽一位Branch控制信号，用于bne的实现：

```

//MEM-PC
wire [31:0] MEM_PreNextPC1;
//assign MEM_PreNextPC1 = (MEM_Branch&&MEM_zero)? MEM_BranchAddr:MEM_NextSeqPC;
//change here:
assign MEM_PreNextPC1 = ((!MEM_Branch[1] && MEM_Branch[0] && MEM_zero) //01-eq
                         || (MEM_Branch[1] && (!MEM_Branch[0]) && (!MEM_zero)))? //10-ne
                           MEM_BranchAddr : MEM_NextSeqPC;
wire [31:0] MEM_PreNextPC2;
assign MEM_PreNextPC2 = MEM_jr? MEM_readData1:MEM_PreNextPC1;
wire [31:0] MEM_NextPC;
assign MEM_NextPC = MEM_Jump? MEM_JumpAddr:MEM_PreNextPC2;

```

2.4.3 仿真验证

向先前的汇编指令序列加入此次新加入的20条指令，得到新的汇编指令序列，如下：

```

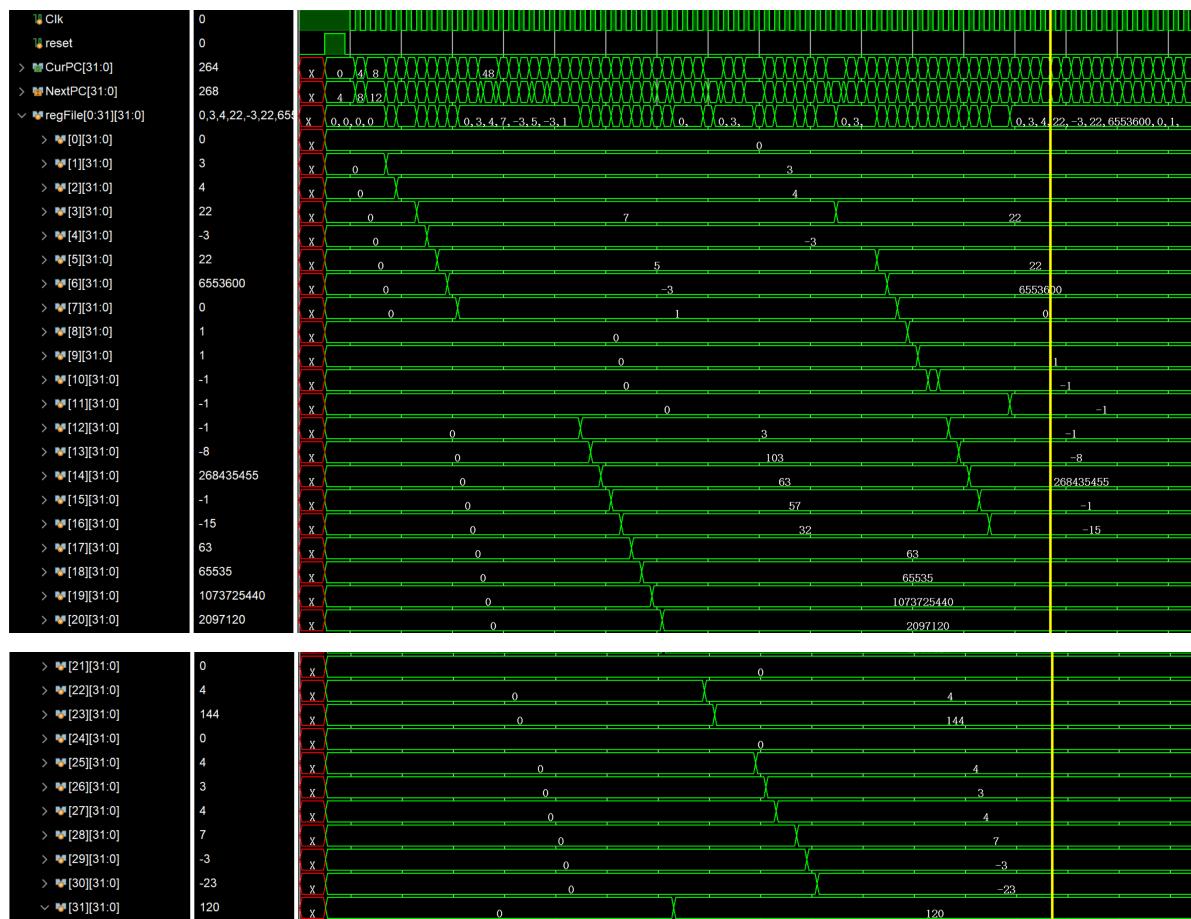
1  lw $1,12($0)
2  lw $2,17($0)
3  add $3,$1,$2
4  sub $4,$2,$3
5  and $5,$3,$4
6  or $6,$4,$5
7  slt $7,$6,$5
8  slt $8,$7,$6
9  beq $1,$2,1
10 add $9,$0,$8
11 beq $9,$9,1
12 add $10,$8,$7
13 add $10,$8,$8
14 sw $10,-1($10)
15 j 19
16 or $11,$0,$1
17 or $11,$0,$1
18 or $11,$0,$1
19 or $11,$0,$1
20 or $12,$0,$1
21 addi $13,$12,100
22 addi $14,$13,-40
23 andi $15,$14,57
24 andi $16,$15,-32
25 ori $17,$16,63
26 ori $18,$17,-43
27 sll $19,$18,14
28 srl $20,$19,9
29 jal 32
30 or $21,$0,$1
31 or $21,$0,$1
32 or $21,$0,$1
33 or $22,$0,$2
34 ori $23,$0,144
35 jr $23
36 or $24,$0,$1
37 or $25,$0,$2
38 //新指令测试从此开始:
39 lbu $26,12($0)
40 lhu $27,17($0)
41 addu $28,$26,$27
42 subu $29,$27,$28
43 addiu $30,$29,-20
44 bne $30,$30,1

```

```
45 nor $3,$30,$30
46 bne $30,$29,1
47 nor $4,$30,$30
48 nor $5,$30,$30
49 lui $6,100
50 slti $7,$6,-20
51 sltiu $8,$7,-20
52 sltu $9,$7,$8
53 xor $10,$8,$3
54 addi $10,$0,-1
55 sra $12,$10,10
56 sllv $13,$12,$1
57 sriv $14,$10,$2
58 srav $15,$10,$2
59 xori $16,$15,14
60 sb $10,10($0)
61 addi $11,$0,-1
62 sh $11,11($0)
```

采用与之前一致的数据内存初始状态，开始仿真，得到如下波形图：

寄存器文件：



不难带入数据后观察得到，各个新加入的指令（除sb和sh不在此处显示之外）产生的结果都是正确的（也可以打开工程文件慢慢检查）。

数据内存：



sw存储的是0，所以没有波形变化；

sh和sb分别存储了16个bit和8个bit，证明实现成功。

2.5 构建调试应用Cache原理设计加入Cache Line的MIPS流水线CPU

2.5.1 Cache原理

- 这里采用的是一个全相联的Cache，有两个entry，每个entry可以放置2个字长的数据。
 - entry的位定义如下：


```
// Assume that A Cache Block is 8 bytes = 2 words
// We have 2 blocks in the cached memory
// The entry of the cache is as followed: totally 69 bits per entry
o // |-----8 bytes/64 bits(cached data)-----|---4bits(addr)----|--1bit(valid)--|
// |0.....63|64.....67|.....68....|
reg [68:0] CacheData [0:1];
reg [31:0] rd;
```
 - [68]是有效位，[67:64]是地址的前四位，地址前四位一致的数据放入同一个缓存块中。[63:32]和[31:0]分别是两个字长为1的缓存数据。共有两个这样的entry。
- 关于写的策略：
 - 这里采用直写 (write-through) 和写分配 (write-allocate) 的策略。
- 关于缓存块替换：
 - 交替替换，实质上就是FIFO先进先出的策略。

2.5.2 Cache Line模块文件的编写

- 在时钟的上升沿进行缓存数据的读写（包括写的分配）
- 在时钟的下降沿进行缓存数据到主存的直写
- 具体部分代码：
 - 在缓存中连接外部的主存：

```

// Cache is connected to Main Memory:
// We apply *write-through* and *write-allocate* policy
reg [31:0] Main_Address;
reg [31:0] Main_MemWriteData;
wire [31:0] Main_ReadData;
reg Main_MemWrite;
reg Main_MemRead;
reg [1:0] Main_MByte;
■ DataMemory DataMemory(
    .Clk(Clk),
    .Address(Main_Address),
    .WriteData(Main_MemWriteData),
    .MemWrite(Main_MemWrite),
    .MemRead(Main_MemRead),
    .MByte(Main_MByte),
    .ReadData(Main_ReadData)
);

```

- 时钟上升沿的读操作，在读miss的情况下讨论冷缓存和已经warm-up的情况：

```

always @(posedge Clk) begin
    if(MemRead == 1) begin
        //read hit
        if(CacheData[0][68] == 1 && CacheData[0][67:64] == Address[4:1]) begin
            if(Address[0] == 0) rd = CacheData[0][31:0];
            else if(Address[0] == 1) rd = CacheData[0][63:32];
        end
        else if(CacheData[1][68] == 1 && CacheData[1][67:64] == Address[4:1]) begin
            if(Address[0] == 0) rd = CacheData[1][31:0];
            else if(Address[0] == 1) rd = CacheData[1][63:32];
        end
        //read miss
        else begin
            //cold cache:
            if(CacheData[0][68] == 0) begin
                CacheData[0][67:64] = Address[4:1];
                CacheData[0][68] = 1;
                // Reading from memory:
                Main_MemRead = 1;
                Main_Address = {Address[31:1],1'b0};
                #2;//wait for reading
                CacheData[0][31:0] = Main_ReadData;
                Main_Address = {Address[31:1],1'b1};
                #2;//wait for reading
                CacheData[0][63:32] = Main_ReadData;
                Main_MemRead = 0;
                // read from cache:
                if(Address[0] == 0) rd = CacheData[0][31:0];
                else if(Address[0] == 1) rd = CacheData[0][63:32];
            end
            else if(CacheData[1][68] == 0) begin
                something omitted here...
            end
        end
    end
end

```

```

    end
    //after warm-up
  else begin
    CacheData[ptr][67:64] = Address[4:1];
    // Reading from memory:
    Main_MemRead = 1;
    Main_Address = {Address[31:1],1'b0};
    #2;//wait for reading
    CacheData[ptr][31:0] = Main_ReadData;
    Main_Address = {Address[31:1],1'b1};
    #2;//wait for reading
    CacheData[ptr][63:32] = Main_ReadData;
    Main_MemRead = 0;
    // read from cache:
    if(Address[0] == 0) rd = CacheData[ptr][31:0];
    else if(Address[0] == 1) rd = CacheData[ptr][63:32];
    //update ptr
    ptr = (ptr + 1) % 2;
  end
end
Main_MemWrite = 0; // Don't write in this cycle.

```

- 时钟上升沿的写操作，在写miss的情况下从主存读入数据，分别讨论冷缓存和warm-up后的情况：

```

else if(MemWrite == 1) begin
  //write hit
  if(CacheData[0][68] == 1 && CacheData[0][67:64] == Address[4:1]) begin
    if(Address[0] == 0) begin
      case(MByte)
        2'b00: CacheData[0][31:0] = WriteData;
        2'b01: CacheData[0][15:0] = WriteData[15:0];
        2'b10: CacheData[0][7:0] = WriteData[7:0];
        default: CacheData[0][31:0] = WriteData;
      endcase
    end
    else if(Address[0] == 1) begin
      case(MByte)
        2'b00: CacheData[0][63:32] = WriteData;
        2'b01: CacheData[0][47:32] = WriteData[15:0];
        2'b10: CacheData[0][39:32] = WriteData[7:0];
        default: CacheData[0][63:32] = WriteData;
      endcase
    end
  end
  else if(CacheData[1][68] == 1 && CacheData[1][67:64] == Address[4:1]) begin
    something omitted here...
  end
end

```

```

end
//write miss
//apply write-allocate
else begin
    //cold cache:
    if(CacheData[0][68] == 0) begin
        CacheData[0][67:64] = Address[4:1];
        CacheData[0][68] = 1;
        //first read from main memory:
        Main_MemRead = 1;
        Main_Address = {Address[31:1],1'b0};
        #2;//wait for reading
        CacheData[0][31:0] = Main_ReadData;
        Main_Address = {Address[31:1],1'b1};
        #2;//wait for reading
        CacheData[0][63:32] = Main_ReadData;
        Main_MemRead = 0;
        //then write the cache memory:
        if(Address[0] == 0) begin
            case(MByte)
                2'b00: CacheData[0][31:0] = WriteData;
                2'b01: CacheData[0][15:0] = WriteData[15:0];
                2'b10: CacheData[0][7:0] = WriteData[7:0];
                default: CacheData[0][31:0] = WriteData;
            endcase
        end
        else if(Address[0] == 1) begin
            case(MByte)
                2'b00: CacheData[0][63:32] = WriteData;
                2'b01: CacheData[0][47:32] = WriteData[15:0];
                2'b10: CacheData[0][39:32] = WriteData[7:0];
                default: CacheData[0][63:32] = WriteData;
            endcase
        end
        //finally write to the main memory at the negative edge of clk
    end
    else if(CacheData[1][68] == 0) begin
        something omitted here...
        //after warm-up
    end
    else begin
        //substitute
        CacheData[ptr][67:64] = Address[4:1];
        // Reading from memory:
        Main_MemRead = 1;
        Main_Address = {Address[31:1],1'b0};
        #2;//wait for reading
        CacheData[ptr][31:0] = Main_ReadData;
        Main_Address = {Address[31:1],1'b1};
        #2;//wait for reading
        CacheData[ptr][63:32] = Main_ReadData;
        Main_MemRead = 0;
        //then write the cache memory:
        if(Address[0] == 0) begin
            case(MByte)
                2'b00: CacheData[ptr][31:0] = WriteData;
                2'b01: CacheData[ptr][15:0] = WriteData[15:0];
                2'b10: CacheData[ptr][7:0] = WriteData[7:0];
                default: CacheData[ptr][31:0] = WriteData;
            endcase
        end
        else if(Address[0] == 1) begin
            case(MByte)
                2'b00: CacheData[ptr][63:32] = WriteData;
                2'b01: CacheData[ptr][47:32] = WriteData[15:0];
                2'b10: CacheData[ptr][39:32] = WriteData[7:0];
                default: CacheData[ptr][63:32] = WriteData;
            endcase
        end
        //update ptr
        ptr = (ptr + 1) % 2;
        //finally write to the main memory at the negative edge of clk
    end
end

```

- 进行直写 (在时钟下降沿的DataMemory(Main Memory)内进行):

```

    //write through to main memory at the negative edge of clk
    Main_MemWrite = 1;
    Main_Address = Address;
    Main_MemWriteData = WriteData;
    Main_MByte = MByte;
  end

```

2.5.3 其他模块文件的更改适配

将之前Top_Pipeline中与DataMemory的连接更改为与Cache Line的连接即可。

```

//Apply Cache_Line instead of DataMemory
wire [31:0] MEM_ReadData;
Cache_Line Cache_Line(
  .Clk(Clk),
  .reset(reset),
  .Address(MEM_ALURES),
  .WriteData(MEM_WriteData),
  .MemWrite(MEM_MemWrite),
  .MemRead(MEM_MemRead),
  .MByte(MEM_MByte),
  .ReadData(MEM_ReadData)
);

```

2.5.4 仿真模拟

采用特别的汇编指令序列和特殊的数据内存初始状态：

汇编指令序列：

```

1  lw $1,0($0)
2  lw $2,1($0)
3  lw $3,0($0)
4  lw $4,1($0)
5  lw $5,2($0)
6  lw $6,3($0)
7  lw $7,2($0)
8  lw $8,3($0)
9  lw $9,4($0)
10  lw $10,5($0)
11  lw $11,6($0)
12  lw $12,7($0)
13
14  lbu $1,0($0)
15  lbu $2,1($0)
16  lbu $3,0($0)
17  lbu $4,1($0)
18  lbu $5,2($0)
19  lbu $6,3($0)
20
21  lhu $7,2($0)
22  lhu $8,3($0)
23  lhu $9,4($0)
24  lhu $10,5($0)
25  lhu $11,6($0)
26  lhu $12,7($0)
27
28  sw $1,8($0)
29  sw $2,9($0)
30  sw $3,8($0)
31  sw $4,9($0)

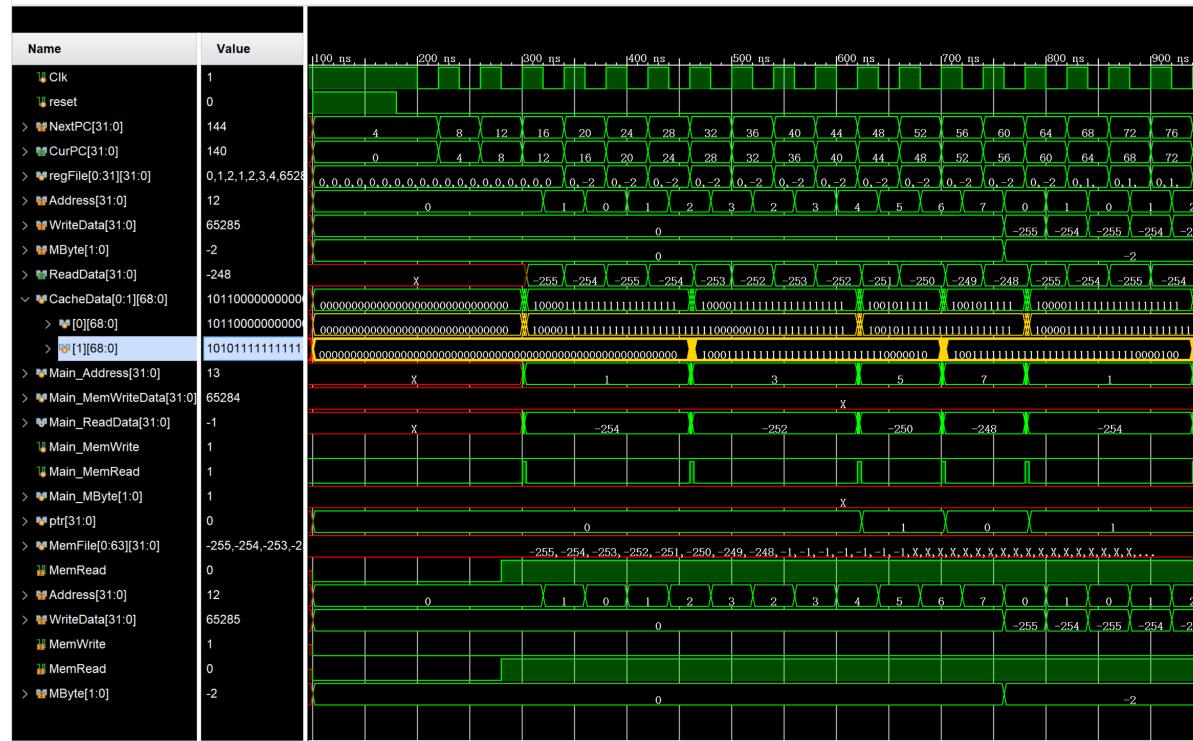
```

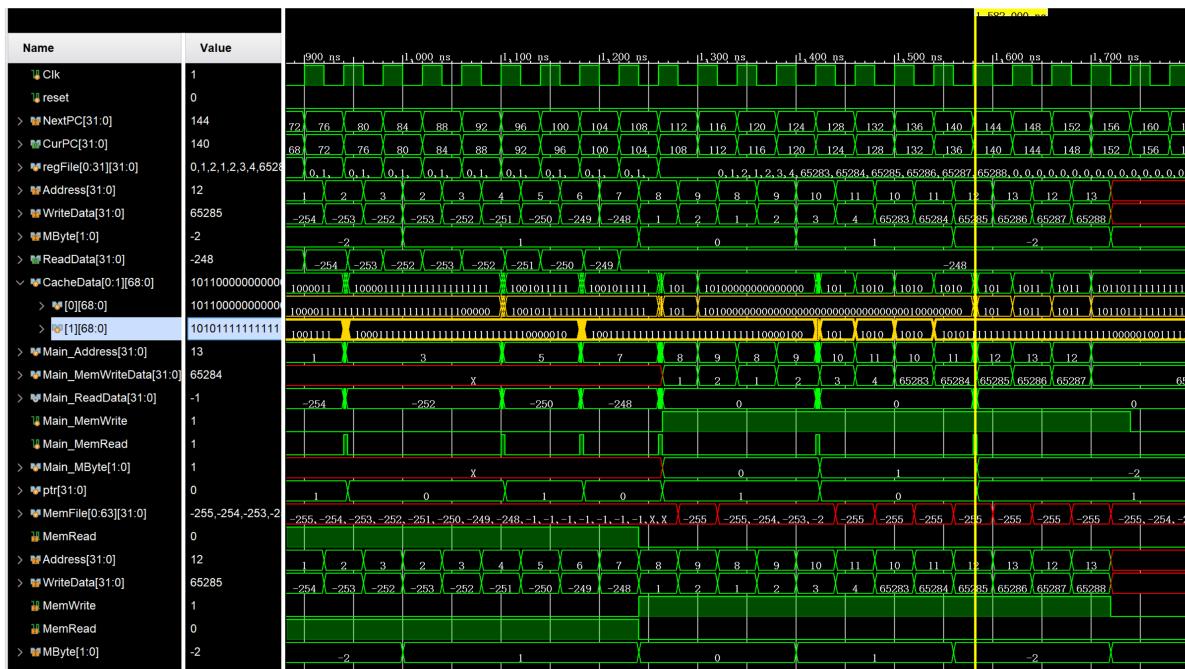
```
32  
33 sh $5,10($0)  
34 sh $6,11($0)  
35 sh $7,10($0)  
36 sh $8,11($0)  
37  
38 sb $9,12($0)  
39 sb $10,13($0)  
40 sb $11,12($0)  
41 sb $12,13($0)
```

数据内存初始状态：

仿真，得到波形图如下：

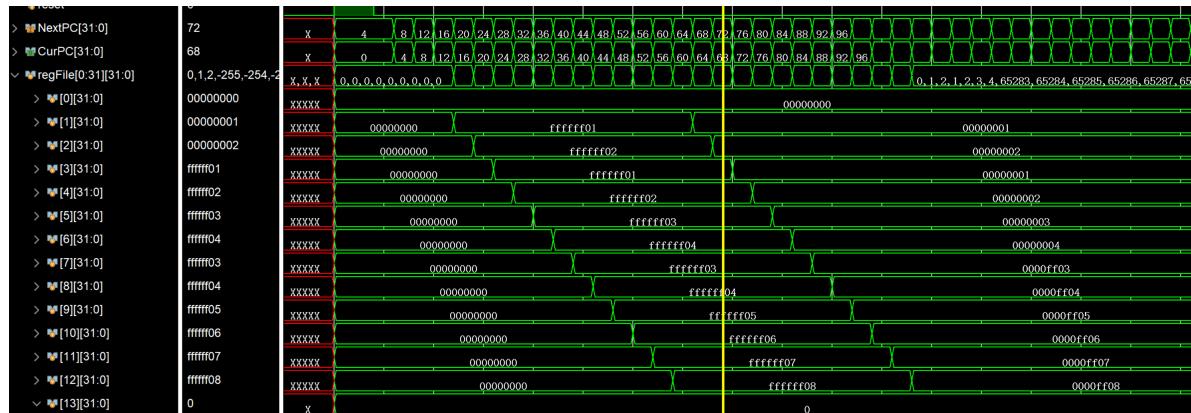
缓存相关信号波形：



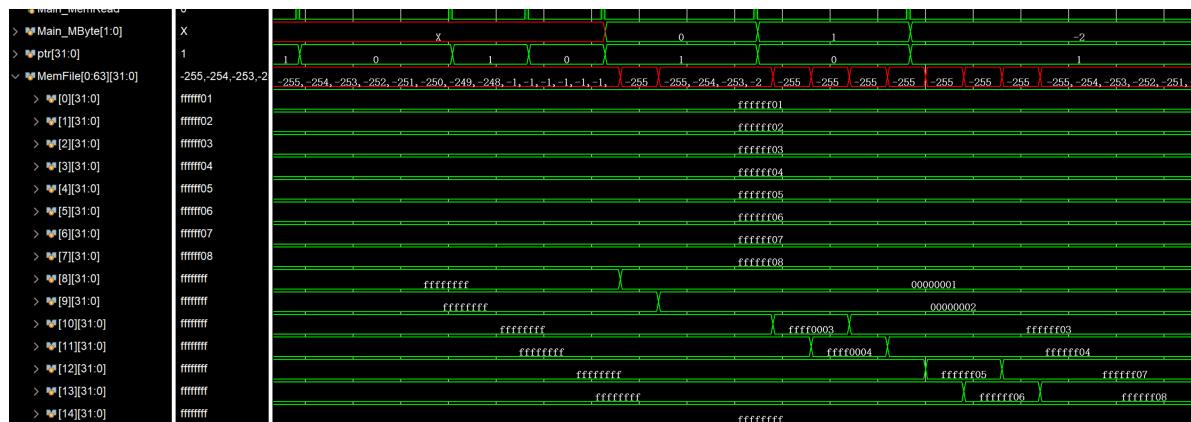


可以看到，一次主存的Main_MemRead信号脉冲可以支持多次的数据读写，说明缓存数据读取的实现成功。

寄存器文件：



数据内存:



上面两个波形图更加印证了直写、写分配策略实现的成功和之前lw/lh/lb和sw/sh/sb指令的成功实现。

2.6 构建调试更加强大的分支预测的MIPS流水线CPU

2.6.1 原理概述

- 在上面的仿真实验中，发现对于一些不必预测，百分百跳转的语句，如j, jal, jr，由于在IF过程中没有对应的译码元器件，导致其被迫需要使用这种低效的恢复手段来实现语句的正确执行。而对于可选的分支跳转语句，仅仅可以预测不跳转，不能预测跳转，使程序变得可能的低效。
- 可以在IF阶段使用一个额外的译码单元，对j, jal, jr语句直接判断出应该往哪个地址进行跳跃，而对于beq和bne计算出可能的跳转地址，并采用一个二位的预测器，动态地对下一指令地址做出预测。

2.6.2 预测模块Predict_PC和Ctr_Hazard_Handler的适配编写

- 在原有的Ctr_Hazard_Handler中，加入一个二位的预测器TwoBitPredictor，并在进行预测MEM_Predict时，根据预测的正误对TwoBitPredictor进行状态的变换，如下：

```
always @(Real_Next_PC)
begin
    #1;
    if(Real_Next_PC != History_Predict_PC[ptr] && recover_cycle == 0) begin
        bubble = 1;
        recover_cycle = 4;
        if(MEM_Predict == 1) begin
            if(Real_Next_PC == MEM_NextSeqPC) begin //predict taken and wrong
                case(TwoBitPredictor)
                    2'b11: TwoBitPredictor = 2'b10;
                    2'b10: TwoBitPredictor = 2'b01;
                    2'b01: TwoBitPredictor = 2'b00;
                    2'b00: TwoBitPredictor = 2'b00;
                endcase
            end
            else begin //predict not-taken and wrong
                case(TwoBitPredictor)
                    2'b11: TwoBitPredictor = 2'b11;
                    2'b10: TwoBitPredictor = 2'b11;
                    2'b01: TwoBitPredictor = 2'b10;
                    2'b00: TwoBitPredictor = 2'b01;
                endcase
            end
        end
    end
    else begin
        bubble = 0;
        if(MEM_Predict == 1 && recover_cycle == 0) begin
            if(Real_Next_PC == MEM_NextSeqPC) begin //predict not-taken and right
                case(TwoBitPredictor)
                    2'b11: TwoBitPredictor = 2'b10;
                    2'b10: TwoBitPredictor = 2'b01;
                    2'b01: TwoBitPredictor = 2'b00;
                    2'b00: TwoBitPredictor = 2'b00;
                endcase
            end
            else begin //predict taken and right
                case(TwoBitPredictor)
                    2'b11: TwoBitPredictor = 2'b11;
                    2'b10: TwoBitPredictor = 2'b11;
                    2'b01: TwoBitPredictor = 2'b10;
                    2'b00: TwoBitPredictor = 2'b01;
                endcase
            end
        end
    end
end
end
```

- Predict_PC模块：用此时的TwoBitPredictor状态作为输入，对bne和beq做预测，并根据jal和j直接计算出跳转地址，对于jr指令，由于可能涉及到数据冒险，工程量较大，这里没有实现。

```

module Predict_PC(
    input Clk,
    input reset,
    input [31:0] Instr,
    input [31:0] IF_CurPC,
    input [1:0] TwoBitPredictor,
    output [31:0] Predict_Next_PC,
    output IF_Predict
);

reg [31:0] pre_next_pc;
reg if_predict;

o

always @(IF_CurPC or Instr or TwoBitPredictor) begin

    case(Instr[31:26])
        6'b0000010: begin //j
            pre_next_pc = {IF_CurPC[31:28],Instr[25:0],2'b00};
            if_predict = 0;
        end
        6'b0000011: begin //jal
            pre_next_pc = {IF_CurPC[31:28],Instr[25:0],2'b00};
            if_predict = 0;
        end
        6'b000100: begin //beq
            if(TwoBitPredictor > 1) begin
                pre_next_pc = IF_CurPC + 4 + {{14{Instr[15]}},Instr[15:0],2'b00};
            end
            else begin
                pre_next_pc = IF_CurPC + 4;
            end
            if_predict = 1;
        end
        6'b000101: begin //bne
            if(TwoBitPredictor > 1) begin
                pre_next_pc = IF_CurPC + 4 + {{14{Instr[15]}},Instr[15:0],2'b00};
            end
            else begin
                pre_next_pc = IF_CurPC + 4;
            end
            if_predict = 1;
        end
        6'b000000: begin
            if(Instr[5:0] == 6'b001000) begin //jr
                pre_next_pc = IF_CurPC + 4;
                if_predict = 0;
                //to be done...
            end
            else begin
                pre_next_pc = IF_CurPC + 4;
                if_predict = 0;
            end
        end
        default: begin
            pre_next_pc = IF_CurPC + 4;
            if_predict = 0;
        end
    endcase
end

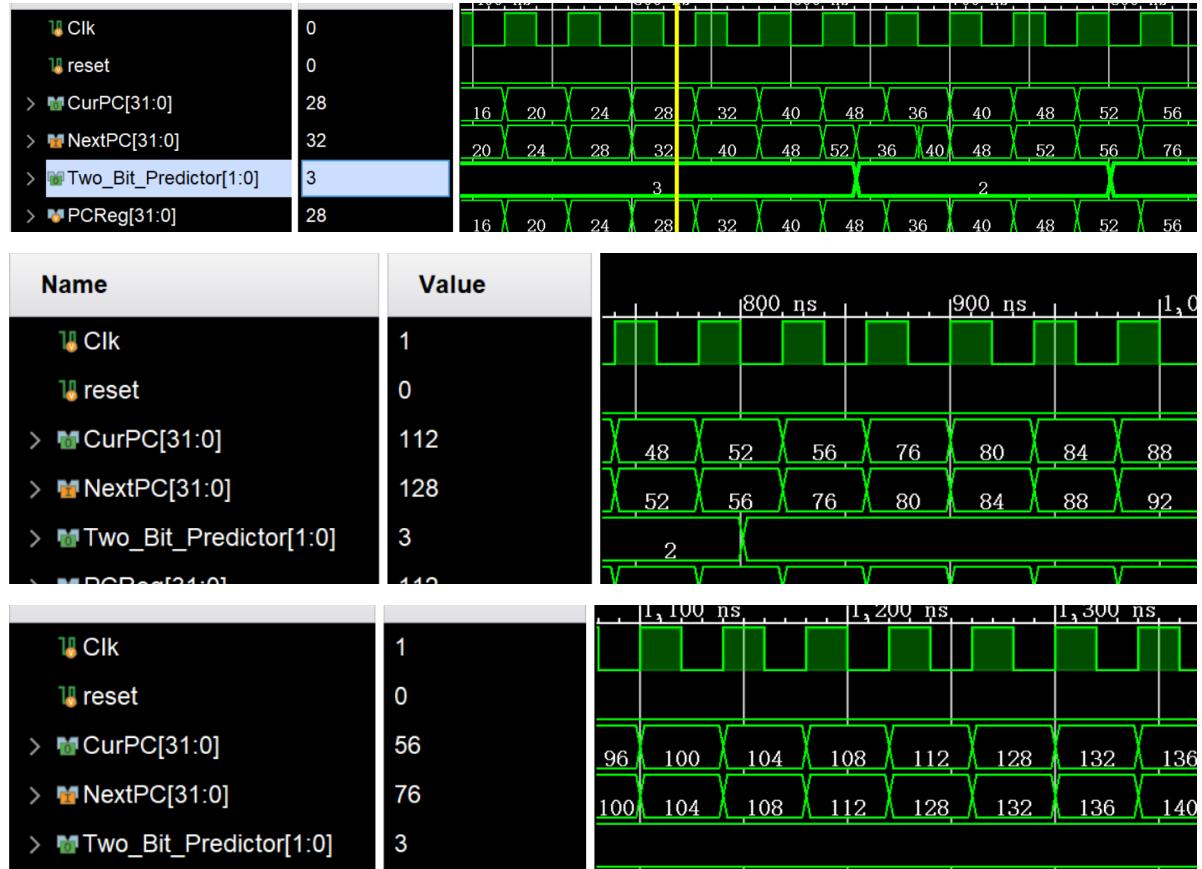
```

2.6.3 仿真与验证

采用和MIPS31指令实现相同的汇编指令序列和内存初始化状态，首先先将TwoBitPredictor的初始值设置为11，此时应在第一个beq语句选择跳转但预测失败，在第二个beq语句选择跳转并预测成功。同时，对于j和jal语句，可以直接得到跳转的PC值。

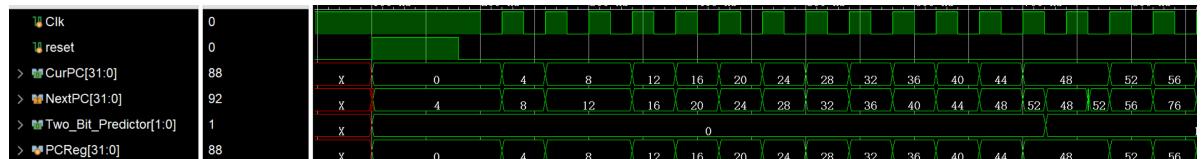
1	PC: 32 beq \$1,\$2,1
2	PC: 40 beq \$9,\$9,1
3	PC: 56 j 19
4	PC: 112 jal 32

结果如下：



而当设置TwoBitPredictor初始值为00时，此时应在第一个beq语句选择不跳转且预测成功，在第二个beq语句选择不跳转并预测失败。

结果如下：



并且可以保证预测PC未改变程序正确运行的逻辑，故实现成功。

3. 实验心得

3.1 实验重难点

过于简单，只是复杂的堆砌而已。希望加大难度，比如浮点数运算，tomasulo算法。

由于我之前用c++实现过一个risc-v的模拟器：<https://github.com/xiahongchi/Risc-v-Simulator>，这次的mips流水线实现在有这个经验的基础上相对而言简单了不少。主要难点还是在从单周期到流水线的思维的转变。

首先需要的是一个没有解决各种hazard的pipeline，构造出最初的雏形，保证各个单个语句的逻辑执行是正确无误的，再加入诸如data hazard, control hazard等等的额外因素。由于部分data hazard涉及到control hazard，所以先解决control hazard是个不错的思路。在解决这个问题的基础上，直接一步到位用forwarding来解决data hazard，这样基础部分的实现就完成了。

接下来就是增加更多指令的Mips流水线版本。需要考虑到与低版本的兼容性。这里比较好的思路就是保证各个大的模块接口做最少的更改，在内部将所需要的逻辑实现完成。关键的就在于ALU, ALUctr和Ctr三个模块。

之后就是带有cache line的实现，这一部分的实现思路就是用cache替代原本DataMemory的位置，并将缓存的思想融入进去。

最后考虑到实现的mips的流水线控制PC上仍然有点小瑕疵，就再加入了投机执行和分支预测的功能，关

键要考虑到与原有模块的兼容性的问题，仍然需要对模块做出较多的修改。

此外，debug程序也是重中之重。这里推荐采用时序方式，从五级流水一步一步来看，找到自己程序的写错的位置并更正。

3.2 实验感想

索然无味。

这个实验的感想还是挺多的。之前都是用C++来模拟的汇编代码的运行，而这次实验带我们用verilog的语言，从更加好的角度理解了计算机系统结构的知识，让知识从书本上落到了实际应用之中。但是我觉得还可以有进一步的拓展，毕竟CPU的设计绝不到此结束。在今后我还会继续挑战将浮点数运算，Scoreboard和Tomasulo等经典算法融入到我的作品之中，让我有更好的代码能力和基础知识，提升自己的综合素养。

4. 参考资料

【1】 2022计算机系统结构实验指导书lab6