

```

1      always @(posedge clk) begin
2          if(WR == 1 && mod_reg != 0)
3              begin
4                  register[mod_reg] <= in_data;
5                  $display("@%h: %d <= %h", programCounter - 8, mod_reg,
in_data);
6              end
7          end
8      end

```

数据冒险

通过一个Forwarding Unit模块解决。具体代码如下：

```
1      always @(*) begin
2          out_forwardA <= 3'b010;
3          out_forwardB <= 3'b010;
4
5          if (EXE_MEM_RegWriteW && (EXE_MEM_Write_Reg != 0) && (in_rr.RegS
== EXE_MEM_Write_Reg)) begin
6              out_forwardA <= 3'b001;
7          end
8          if (EXE_MEM_RegWriteW && (EXE_MEM_Write_Reg != 0) && (in_rr.RegT
== EXE_MEM_Write_Reg)) begin
9              out_forwardB <= 3'b001;
10         end
11         if (MEM_WB_RegWriteW && (MEM_WB_Write_Reg != 0) && !
(EXE_MEM_RegWriteW && ( EXE_MEM_Write_Reg !=0 ) && (EXE_MEM_Write_Reg ==
in_rr.RegS)) && (in_rr.RegS == MEM_WB_Write_Reg)) begin
12             out_forwardA <= 3'b100;
13         end
14         if (MEM_WB_RegWriteW && (MEM_WB_Write_Reg != 0) && !
(EXE_MEM_RegWriteW && ( EXE_MEM_Write_Reg !=0 ) && (EXE_MEM_Write_Reg ==
in_rr.RegT)) && (in_rr.RegT == MEM_WB_Write_Reg )) begin
15             out_forwardB <= 3'b100;
16         end
17     end
18 end
```

根据 $RegS$ 和 $RegT$ 的冲突与 EXE_MEM 流水线寄存器和 MEM_WB 流水线寄存器中需要写的寄存器的冲突情况，如果 EXE 需要使用的寄存器和 MEM 或 WB 阶段的冲突，那就需要用旁路转发，使 ALU 选择正确的数值。

还有 $AdventureDetect$ 中的一部分代码，用于检测 lw 造成的数据冒险：

```
1      if ((ID_EXE_opcode == 6'b100011 && (ID_EXE_writeReg != 0 &&
(ID_EXE_writeReg == RegS || ID_EXE_writeReg == RegT)))) // lw
2          begin
3              PC_write <= 3'b000;
4              IF_ID_write <= 3'b010;
5              ID_EX_write <= 3'b000;
6          end
7
```

如果发现 EXE 阶段的指令是 lw ，且 ID 阶段需要读的寄存器和 EXE 阶段需要写的寄存器一样，则需要让 ID 阶段的指令 $stall$ 一个周期。

分支冒险

```
1      if ((ID_EXE_opcode[5:0] == 6'b000011) || EXE_MEM_opcode == 6'b000010
|| MEM_WB_opcode == 6'b000010 || EXE_MEM_opcode == 6'b000011 ||
MEM_WB_opcode == 6'b000011 || (EXE_MEM_opcode == 6'b000000 && EXE_MEM_func ==
6'b001000) || (MEM_WB_opcode == 6'b000000 && MEM_WB_func == 6'b001000))
// j
```

```

2         begin
3             f ((ID_EXE_opcode[5:0] == 6'b000011 || ( EXE_MEM_opcode
== 6'b000011 && (IF_ID_ins_addr_plus_4 - EXE_MEM_ins_addr_plus_4 == 3'b100)
)|| (MEM_WB_opcode == 6'b000011 && (IF_ID_ins_addr_plus_4 -
MEM_WB_ins_addr_plus_4 == 3'b100))) && (RegS == 5'b11111 || RegT ==
5'b11111))
4                 begin // jal指令对31号寄存器进行操作, jal的下一条指令会
读到31号寄存器
5                     if (ID_EXE_opcode == 6'b000011 || EXE_MEM_opcode
== 6'b000011)
6                         begin
7                             PC_Write <= 3'b000;
8                             IF_ID_Write <= 3'b010;
9                             ID_EX_Write <= 3'b000;
10                        end
11                    if (MEM_WB_opcode == 6'b000011)
12                        begin
13                            PC_Write <= 3'b111;
14                            IF_ID_Write <= 3'b010;
15                            ID_EX_Write <= 3'b000;
16                        end
17                    end
18
19                else if ((EXE_MEM_opcode == 6'b000000 && EXE_MEM_func ==
6'b001000) || EXE_MEM_opcode == 6'b000010 || EXE_MEM_opcode == 6'b000011)
20                    begin
21                        PC_Write <= 3'b000;
22                        IF_ID_Write <= 3'b000;
23                        ID_EX_Write <= 3'b000;
24                    end
25                else if ((MEM_WB_opcode == 6'b000000 && MEM_WB_func ==
6'b001000) || MEM_WB_opcode == 6'b000010 || MEM_WB_opcode == 6'b000011 )
26                    begin
27                        PC_Write <= 3'b111;
28                        IF_ID_Write <= 3'b010;
29                        ID_EX_Write <= 3'b000;
30                    end
31                end
end

```

这段代码处理的是*j*, *jal*, *jr*三条指令的相关情况。

第一个分支为*jal*的下一条指令会读到31号寄存器的特殊情况, 这种情况下, 需要让*IF_ID*寄存器中的指令*stall*三个周期, 这里的实现方法比较笨拙, 其实可以通过旁路转发, 来达到不*stall*的效果, 这在之后的50条指令中会改进。

第二个分支为检测到*EXE*阶段的操作作为三个跳转之一时, 要将下一次进入*ID_EX*和*IF_ID*流水寄存器的值清零, 并让*PC*的值保持不变 (这里的设计是, *PC_Write*为3'b000时*PC*的值保持不变)。

第三个分支为与第二个分支类似, 只是需要让*PC*写入新值。

另一部分代码用于处理*beq*指令, 分为两个部分来分析。

```

1         if (EXE_MEM_opcode == 6'b000100)
2             begin

```

```

3         if (EXE_MEM_zero_flag == 1'b0)
4             begin
5                 if (ID_EXE_opcode == 6'b100011 &&
6                     ((IF_ID_ins[31:26] != 6'b100011 && ID_EXE_WriteReg != 0 && (ID_EXE_WriteReg
7                     == RegS || ID_EXE_WriteReg == RegT))) || (IF_ID_ins[31:26] == 6'b100011 &&
8                     ID_EXE_WriteReg && ID_EXE_WriteReg == RegS))
9                     begin
10                        PC_write <= 3'b000;
11                        IF_ID_Write <= 3'b010;
12                        ID_EX_Write <= 3'b000;
13                    end
14                else if ((RegS == 5'b11111 || RegT ==
15                5'b11111) && ID_EXE_opcode == 6'b000011)
16                    begin
17                        PC_write <= 3'b000;
18                        IF_ID_Write <= 3'b010;
19                        ID_EX_Write <= 3'b000;
20                    end
21                else
22                    begin
23                        PC_write <= 3'b111;
24                        IF_ID_Write <= 3'b111;
25                        ID_EX_Write <= 3'b111;
26                    end
27            end
28        else if (EXE_MEM_zero_flag == 1'b1)
29            begin
30                PC_write <= 3'b000;
31                IF_ID_Write <= 3'b000;
32                ID_EX_Write <= 3'b000;
33            end
34    end

```

第一部分，当 EXE_MEM 中的指令为 beq 时，分两种情况讨论。

当不需要跳转时，分三种情况：之后的连续两条指令都为 lw 且前一条 lw 要写的寄存器是第二条 lw 要读的寄存器或只有后一条指令为 lw 但产生了数据冲突。这样就需要 PC 的值保持不变， IF_ID 寄存器中的值也不改变，清空 ID_EX 中的值；当 ID_EX 中的指令为 jal 且产生了数据冲突时，做同样的处理。其余情况，按照正常状态执行。

当需要跳转时，让两个流水寄存器清零，并且 PC 保持不变。

```

1         if (MEM_WB_opcode == 6'b000100)
2             begin
3                 if (MEM_WB_zero_flag == 1'b1)
4                     begin
5                         PC_write <= 3'b111;
6                         IF_ID_Write <= 3'b000;
7                         ID_EX_Write <= 3'b111;
8                     end
9                 else if (MEM_WB_zero_flag == 1'b0)
10                    begin

```

```

11         if (ID_EXE_opcode == 6'b000011 ||
ID_EXE_opcode == 6'b000010 || (ID_EXE_opcode == 6'b000000 && ID_EXE_func ==
6'b001000))
12             begin
13                 if ((RegS == 5'b11111 || RegT ==
5'b11111) && ID_EXE_opcode == 6'b000011)
14                     begin
15                         PC_Write <= 3'b000;
16                         IF_ID_Write <= 3'b010;
17                         ID_EX_Write <= 3'b000;
18                     end
19                 else
20                     begin
21                         PC_Write <= 3'b111;
22                         IF_ID_Write <= 3'b000;
23                         ID_EX_Write <= 3'b111;
24                     end
25             end
26         else if ((ID_EXE_opcode == 6'b100011 &&
(ID_EXE_WriteReg != 0 && (ID_EXE_WriteReg == RegS || ID_EXE_WriteReg ==
RegT))))
27             begin
28                 PC_Write <= 3'b000;
29                 IF_ID_Write <= 3'b010;
30                 ID_EX_Write <= 3'b000;
31             end
32         else
33             begin
34                 PC_Write <= 3'b111;
35                 IF_ID_Write <= 3'b111;
36                 ID_EX_Write <= 3'b111;
37             end
38         end
39     end
40 end

```

这部分与之前的情况类似，只需要让 PC 寄存器写入新值即可。

需要说明的是，这个冒险处理的设计实在是太过笨拙，在50条指令的设计中会改成统一简洁的模式对冒险处理单元进行设计，可以通过旁路转发和增加新的组件来进行极大的优化。

实验结果

