CSE-COA-LAB-006

计算机组成与系统实验

马叶晟[1] [2]

5140209064

指导老师：黄小平

2016 年 4 月 7 日

[1] email: kimi.ysma@gmail.com

[2] 上海交通大学，计算机科学与工程系.

1 实验介绍

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

理解CPU的pipeline，对data hazard，branch hazard有初步认识

1.3 实验范围

本次试验将覆盖以下范围:

1. ISE的使用
2. FPGA实验板的使用
3. 使用Verilog HDL进行逻辑设计

2 实验内容

2.1 多周期流水线MIPS处理器整体实现概述

在之前完成的ALU,ALUCtr, Ctr, register, dataMem, instMem, signExt的基础上完成多周期流水线CPU的实现. 总体来说，多周期流水线的CPU的实现难度和复杂度远远大于之前的单周期CPU，在上面调试的时间也远远多于lab5. 当然，这也使我更深刻地理解了流水线的思想和解决各种hazard的方法.

2.2 MIPS流水线概述

流水线的概念的提出成倍地提高了CPU工作的效率. 严格地来说，流水线并没有提高CPU执行每一个指令的时间，但是大大地提高CPU的吞吐率. 在我们实现的简化版的MIPS处理器中，整个处理器被分割成了四个阶段: IF, ID, EX, MEM, WB. 在各级流水线之间，我们插入了一些寄存器，这些寄存器对于下一阶段流水线的正常工作具有重要的意义.

下面具体说一下各个阶段CPU的工作:

- IF(instruction fetch): 获取即将执行的指令
- ID(instruction decode): 对得到的指令进行译码，得到一系列控制信号
- EX(execution): 执行指令，进行相关的算术运算并判断branch等情况
- MEM(memory): 从内存中读取或写入数据
- WB(write back): 对于lw等指令，将从MEM级中得到的数据写入寄存器堆中

MIPS是一种为了流水线的而特意设计的机器语言，它有如下的特征：所有指令都有相同的长度、只有比较有限的几种指令形式、不能同时从内存中读取和写入.

MIPS多周期流水线架构的数据通路示意图如下所示：

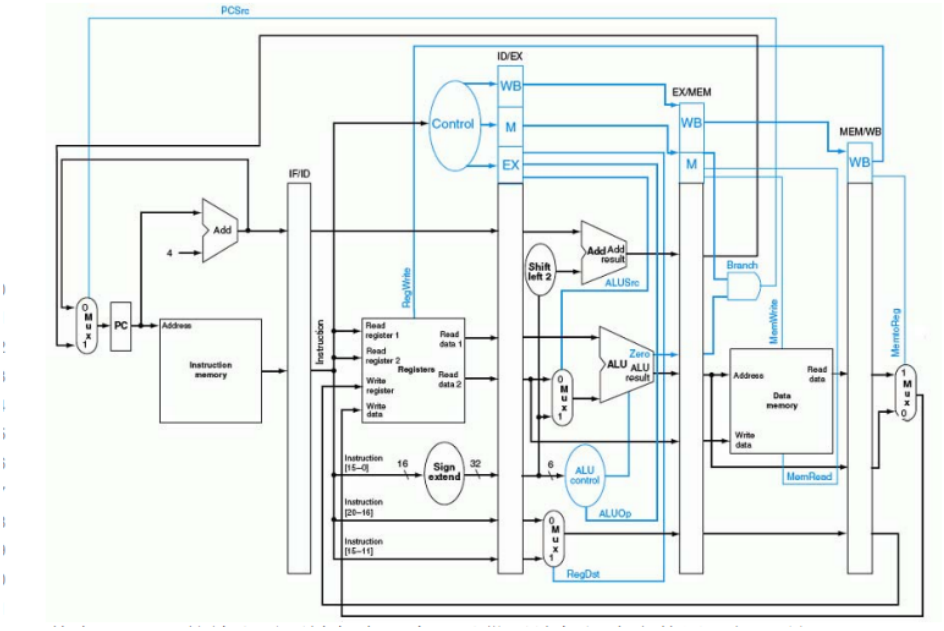

图 1: MIPS多周期流水线的数据通路示意图

2.3 流水线中线网的命名规则

对于流水线中的变量命名，必须有一套较为统一的规则，否则到最后调试的时候会非常混乱. 我采取的命名方式是级数_变量名或级数_级数_变量名，如IF_PCPlusFour和IF_ID_PCPlusFour. 这样的命名方式能够较清楚地看出变量的意义以及变量在整个流水线中所占据的位置.

2.4 多周期流水线CPU的端口

多周期流水线CPU的输入端口是: CLOCK_IN, RESET.

2.5 多周期流水线中的top模块的实现

在top模块的实现中，我们首先完成各级流水线之间的线网连接. 按照流水线的设计，在每一个周期中，流水线都会向前推进一级，这就牵涉到了各级流水线之间是采用组合逻辑还是时序逻辑的问题. 虽然有一些寄存器可以采用组合逻辑的方式来实现，但是这就会大大提高我们编程的复杂性，整个top模块中也会新增许多变量. 因此，我还是采用统一的在时钟的上升沿进行相应的寄存器的推进工作，这样就大大减轻了编程的负担.

其次，我们要完成获取下一个PC的逻辑. 这个过程与lab5类似，同样是通过组合逻辑，使用Verilog的三目运算符来实现数据选择的功能. 这一部分与lab5类似，就不赘述了.

到这里为止，我们已经实现了一个简略版的多周期流水线的MIPS处理器，之后我将分析流水线CPU中的各种hazard并解决其中的control hazard.

2.6 流水线CPU中的hazard的分析和解决

流水线虽然提高了CPU的吞吐率，但是同时也带来了几种hazard. 这些hazard包括: hardware hazard, data hazard, control hazard.

首先，对于hardware hazard的解决，由于MIPS是为流水线特别设计的机器语言，因此设计者在设计这个语言的时候就已经避免了hardware hazard. 因此，我们就没有必要去考虑hardware hazard了.

其次，我们要处理data hazard. Data hazard的解决需要分情况来讨论，一种是普通的data hazard，这种hazard与内存无关，因此可以直接通过转发来解决；另一种是load-use data hazard，这种hazard无论如何也需要流水线的停滞来完成，等到load完成之后再开始use的运算. 当然，我们也可以用编程的手段来避免这种data hazard，但是这无疑增加了汇编程序员的负担了.

最后一种要处理的hazard是control hazard. 引起control hazard的原因是由于`beq`或`jump`指令的跳转，CPU应当执行的下一条指令改变了，但是由于流水线的特性导致有一条或者两条并不会被执行的指令已经加载到CPU当中去了，这时就需要我们排空流水线，根据跳转的地址，重新从新的PC处读取指令. 这样就能使CPU依旧正常地工作.

在我实现的多周期流水线CPU中，我解决了control hazard，而data hazard则需要通过具体的编程来避免. 为了解决hazard，我添加了一个模块`hazardDetect`，这个模块接受输入`jump`、`PcSrc`，输出信号`IF_ID_Flush`、`ID_EX_Stall`、`EX_MEM_Stall`. 受到影响的流水线会根据各自的信号值加载合适的数据. 这样在发生跳转的时候就总是能排空流水线并执行正确的指令. `HazardDetect`模块的代码实现见附录1，top模块的代码实现见附录2.

2.7 多周期流水线MIPS处理器的仿真波形

在lab6的仿真中，我采用了与lab5类似的testbench. 测试的指令集如下所示:

```
1     memBuf[0] = 8'b10001100;
2     memBuf[1] = 8'b00000000;
3     memBuf[2] = 8'b00000000;
4     memBuf[3] = 8'b00000000; // nop
5     memBuf[4] = 8'b00010000;
6     memBuf[5] = 8'b00000000;
7     memBuf[6] = 8'b00000000;
8     memBuf[7] = 8'b00000000; // beq 0 0
9     memBuf[8] = 8'b10001100;
10    memBuf[9] = 8'b00000000;
11    memBuf[10] = 8'b00000000;
12    memBuf[11] = 8'b00000000; // nop
13    memBuf[12] = 8'b10001100;
14    memBuf[13] = 8'b00000000;
15    memBuf[14] = 8'b00000000;
16    memBuf[15] = 8'b00000000; // nop
17    memBuf[16] = 8'b10001100;
18    memBuf[17] = 8'b00000001;
19    memBuf[18] = 8'b00000000;
20    memBuf[19] = 8'b00000001; // regFile[1]=memFile[1]
21    memBuf[20] = 8'b10001100;
22    memBuf[21] = 8'b00000010;
23    memBuf[22] = 8'b00000000;
24    memBuf[23] = 8'b00000010; // regFile[2]=memFile[2]
25    memBuf[24] = 8'b00000000;
```

```
26      memBuf[25] = 8'b00000000;
27      memBuf[26] = 8'b00000000;
28      memBuf[27] = 8'b00000000; // nop
29      memBuf[28] = 8'b00000000;
30      memBuf[29] = 8'b00000000;
31      memBuf[30] = 8'b00000000;
32      memBuf[31] = 8'b00000000; // nop
33      memBuf[32] = 8'b00000000;
34      memBuf[33] = 8'b00000000;
35      memBuf[34] = 8'b00000000;
36      memBuf[35] = 8'b00000000; // nop
37      memBuf[36] = 8'b00000000;
38      memBuf[37] = 8'b00100010;
39      memBuf[38] = 8'b00011000;
40      memBuf[39] = 8'b00100000; // regFile[3]=regFile[1]+regFile[2]
41      memBuf[40] = 8'b00000000;
42      memBuf[41] = 8'b01000001;
43      memBuf[42] = 8'b00011000;
44      memBuf[43] = 8'b00100010; // regFile[3]=regFile[1]-regFile[2]
45      memBuf[44] = 8'b00000000;
46      memBuf[45] = 8'b01000001;
47      memBuf[46] = 8'b00011000;
48      memBuf[47] = 8'b00100101; // regFile[3]=regFile[1]|regFile[2]
49      memBuf[48] = 8'b00000000;
50      memBuf[49] = 8'b01000001;
51      memBuf[50] = 8'b00011000;
52      memBuf[51] = 8'b00100100; // regFile[3]=regFile[1]&regFile[2]
```

仿真的波形图如下所示：

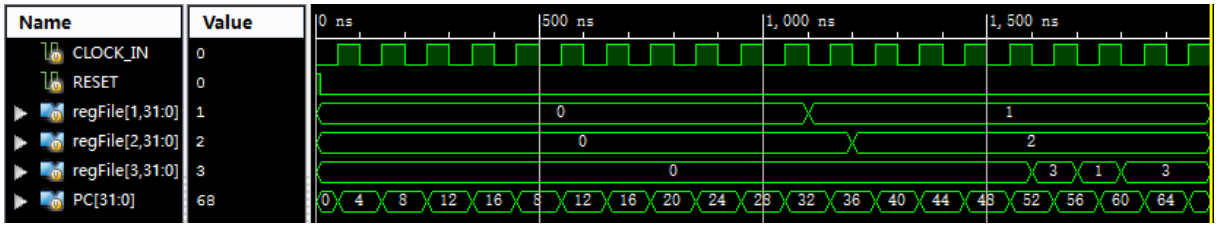

图 2: 多周期流水线CPU的仿真波形图

从仿真图像中我们可以看到，第一个beq指令使得流水线停滞了两个阶段，之后加载到了PC=8的位置，这是为了解决control hazard而导致的. 之后，我在设计指令集的时候特意设置了一个load-use data hazard，而在编程解决load-use data hazard时，必须加入3个nop指令来使数据写入到内存当中. 从波形图中可以看到，各个算术运算均得到了正确的结果而没有受到data hazard的影响.

2.8 多周期流水线MIPS的上板验证

在lab6的上班验证中，与lab5类似，我们仍然需要完成一个分频器的实现来降低时钟频率以便观察，并将led与相关需要观测的寄存器连接. 此外，由于lab6整个工程比较大，在synthesize的过程中会耗费大量的时间，因此一种比较好的策略是修改完一些部分之后统一上板验证，这样效率会有大幅度

的提高. 还有一点是在上板验证的时候最好把内存模块做的小一些，这样不仅能减少硬件的使用率，而且减少了综合的时间.

在用多周期流水线CPU进行上板验证的时候，需要注意的一点是多周期流水线CPU由于要解决control hazard和data hazard，因此在某些阶段led可能会很长时间没有变化，这时候我们需要耐心地等待，因为这时候流水线需要停滞一段时间或者正在执行nop指令，在解决完hazard之后流水线依旧会按照正常速率进行吞吐.

3 心得体会

lab6是计算机组成与结构实验的最后一个实验，也是难度最高、最复杂的实验. 为了完成lab6，我花费了大量时间在课余时间调试多周期流水线CPU. 因为ISE编译综合需要要耗费大量的时间，而在自己的笔记本电脑上，CPU的性能也比不上实验室里的台式机，因此也耗费了不少的时间. 而在上板验证的时候，一开始使用数电的小板子的时候没有经验，为了让指令能够成功地加载进去就耗费了我不少时间，而经过了一段时间的练习，我对于小板子的管脚、下载等就比较熟悉了，开发上的效率也就提高了不少. 也体会到了FPGA作为可编程逻辑器件的灵活多样性.

这次实验给我的经验教训是，首先，做任何事情都要认真仔细，一开始我实现的`ALU`模块就有一些微小的错误，但是当时没有发现，而到最后综合调试的时候，这些小错误就被无限地放大，产生让人疑惑的错误，所幸我之后耐心地跟踪这些错误排除了它们，但是在中间耗费的时间还是很可观的. 还有一点是做什么事情都要有一点提前意识，在最后做lab5和lab6的过程中，我明显感觉到时间有些紧，后悔没有在做lab3和lab4的时候就顺带着把lab5做完，导致我最后用了两个完整的周六周日来做lab5和lab6. 在最后一次实验的之前还熬夜了好几天. 当然，最后做出结果的时候，看到led按照我的设想在跳动的时候还是很有成就感的.

在写lab6的代码的时候，一种比较好的写代码的流程是把代码分成各个阶段，把变量命名和时序逻辑的实现都按照流水线的各级来进行，这样在寻找变量定义和代码的时候比较方便快捷，也不容易出错，可以减少编程时的烦躁感. 同时，在编写Verilog代码的时候应当注意代码风格的养成. 良好的代码风格包括缩进、端口的书写等，我采用的代码风格是一行写一个模块端口，这是比较推荐的Verilog代码风格，也能较为方便地看出一个模块的端口. 此外，由于lab中的代码需要检测观察data hazard和control hazard，因此相关的MIPS代码的生成是一件比较繁琐的事情，与lab5一样我们可以使用SPIM汇编器这个有力的工具，生成相关的机器码. 更进一步我们可以写一些脚本，把这里的机器码转变成Verilog代码. 这些脚本都是十分好写的，写完之后也能极大地提高生产力.

我发现对于很多问题，有时候自己思考没有思路的时候，可以在网上寻找相关的资源. 对于Verilog而言，比较有用的主要有AsicWorld, Stack Overflow, Xilinx forum等资源，通过这些资源我们能够获得比较详尽，权威的资料作为参考，尤其是在Verilog这种偏向工程的语言上，前人的经验是尤其珍贵的.

总而言之，lab6使我对于多周期流水线CPU的优越性有了深刻的理解，它并没有减少执行每一个指令的时间，但是成倍地提高了指令的吞吐率，因此在效率上有了比较大的提升. 此外，lab6也使我知道了control hazard和data hazard究竟是如何在硬件层面上得到解决的，终于将我在计算机组成理论课上学习到的知识运用到了实际当中，也解决了我在理论课上遇到的一些困惑.

4 致谢

感谢老师在实验过程中的耐心指导. 感谢同学之间的互相勉励, 同时帮助别的同学解决一些硬件上的问题也使我很有成就感. 也感谢那些在网络上为他人解答疑惑的工程技术人员们.

通过这门课我确实学习到了一些很有用的知识，对于Verilog硬件描述语言也扩展了我的眼界. 这一定是我大学中最难忘的课程之一.

附录1

HazardDetect模块的代码如下所示：

```
module hazardDetect(
    jump,
    PcSrc,
    IF_ID_Flush,
    ID_EX_Stall,
    EX_MEM_Stall
);
    input jump;
    input PcSrc;
    output reg IF_ID_Flush;
    output reg ID_EX_Stall;
    output reg EX_MEM_Stall;

    initial begin
        IF_ID_Flush = 0;
        ID_EX_Stall = 0;
        EX_MEM_Stall = 0;
    end

    always @ (jump or PcSrc) begin
    if (jump) begin
        IF_ID_Flush = 1;
        ID_EX_Stall = 0;
        EX_MEM_Stall = 0;
    end
    else if (PcSrc) begin
        IF_ID_Flush = 1;
        ID_EX_Stall = 1;
        EX_MEM_Stall = 1;
    end
    else begin
        IF_ID_Flush = 0;
        ID_EX_Stall = 0;
        EX_MEM_Stall = 0;
    end
end
endmodule
```

附录2

top模块的代码如下所示：

```
module Top(
    CLOCK_IN,
    RESET
);
    input CLOCK_IN;
    input RESET;

    initial begin
    PC <= 0;
    IF_ID_PCPlusFour <= 0;
    IF_ID_Inst <= 0;
    ID_EX_RegDst <= 0;
```

```
13      ID_EX_AluSrc <= 0;
14      ID_EX_MemToReg <= 0;
15      ID_EX_RegWrite <= 0;
16      ID_EX_MemRead <= 0;
17      ID_EX_MemWrite <= 0;
18      ID_EX_Branch <= 0;
19      ID_EX_AluOp <= 0;
20      ID_EX_PCPlusFour <= 0;
21      ID_EX_ReadData1 <= 0;
22      ID_EX_ReadData2 <= 0;
23      ID_EX_Signext <= 0;
24      IF_ID_Inst_20_16 <= 0;
25      IF_ID_Inst_15_11 <= 0;
26      EX_MEM_MemToReg <= 0;
27      EX_MEM_RegWrite <= 0;
28      EX_MEM_MemRead <= 0;
29      EX_MEM_MemWrite <= 0;
30      EX_MEM_Branch <= 0;
31      EX_MEM_AddRes <= 0;
32      EX_MEM_Zero <= 0;
33      EX_MEM_AluRes <= 0;
34      EX_MEM_ReadData2 <= 0;
35      EX_MEM_WriteReg <= 0;
36      MEM_WB_ReadData <= 0;
37      MEM_WB_AluRes <= 0;
38      MEM_WB_WriteReg <= 0;
39      MEM_WB_MemToReg <= 0;
40      MEM_WB_RegWrite <= 0;
41      end
42
43      // IF
44      reg [31:0] PC;
45
46      // IF_ID
47      wire [31:0] IF_PcPlusFour;
48      wire [31:0] IF_INST;
49      wire [31:0] IF_BeqPc;
50      wire [31:0] IF_NextPc;
51      reg [31:0] IF_ID_PCPlusFour;
52      reg [31:0] IF_ID_Inst;
53
54
55      //ID->EX
56      wire ID_RegDst;
57      wire ID_AluSrc;
58      wire ID_MemToReg;
59      wire ID_RegWrite;
60      wire ID_MemRead;
61      wire ID_MemWrite;
62      wire ID_Branch;
63      wire ID_Jump;
64      wire [1:0] ID_AluOp;
65      wire [31:0] ID_ReadData1;
66      wire [31:0] ID_ReadData2;
67      wire [31:0] ID_Signext;
68      reg ID_EX_RegDst;
69      reg ID_EX_AluSrc;
70      reg ID_EX_MemToReg;
71      reg ID_EX_RegWrite;
72      reg ID_EX_MemRead;
73      reg ID_EX_MemWrite;
```

```
74      reg ID_EX_Branch;
75      reg [1:0] ID_EX_AluOp;
76      reg [31:0] ID_EX_PCPlusFour;
77      reg [31:0] ID_EX_ReadData1;
78      reg [31:0] ID_EX_ReadData2;
79      reg [31:0] ID_EX_Signext;
80      reg [20:16] IF_ID_Inst_20_16;
81      reg [15:11] IF_ID_Inst_15_11;
82
83      // EX_MEM
84      wire [31:0] EX_AddRes;
85      wire [31:0] EX_AluSrc_input_2;
86      wire [5:0] EX_WriteReg;
87      wire EX_Zero;
88      wire [31:0] EX_AluRes;
89      wire [3:0] EX_AluCtr;
90
91      //reg
92      reg EX_MEM_MemToReg;
93      reg EX_MEM_RegWrite;
94      reg EX_MEM_MemRead;
95      reg EX_MEM_MemWrite;
96      reg EX_MEM_Branch;
97      reg [31:0] EX_MEM_AddRes;
98      reg EX_MEM_Zero;
99      reg [31:0] EX_MEM_AluRes;
100     reg [31:0] EX_MEM_ReadData2;
101     reg [4:0] EX_MEM_WriteReg;
102
103
104     // MEM_WB
105     wire [31:0] MEM_ReadData;
106     wire MEM_PCSrc;
107     reg [31:0] MEM_WB_ReadData;
108     reg [31:0] MEM_WB_AluRes;
109     reg [4:0] MEM_WB_WriteReg;
110     reg MEM_WB_MemToReg;
111     reg MEM_WB_RegWrite;
112     wire [31:0] WB_WriteData;
113
114
115     assign IF_PcPlusFour = PC + 4;
116     assign IF_BeqPc = MEM_PCSrc ? EX_MEM_AddRes : IF_PcPlusFour;
117     assign IF_NextPc = ID_Jump?{IF_ID_PCPlusFour[31:28],IF_ID_Inst
            [25:0],0}:IF_BeqPc;
118     assign EX_AddRes = ID_EX_PCPlusFour + (ID_EX_Signext << 2);
119     assign EX_AluSrc_input_2 = ID_EX_AluSrc ? ID_EX_Signext :
            ID_EX_ReadData2;
120     assign EX_WriteReg = ID_EX_RegDst ? IF_ID_Inst_15_11 :
            IF_ID_Inst_20_16;
121     assign MEM_PCSrc = EX_MEM_Branch & EX_MEM_Zero;
122     assign WB_WriteData = MEM_WB_MemToReg ? MEM_WB_ReadData :
            MEM_WB_AluRes;
123
124
125
126
127
128     instMem mainInstMem(
129         .addr(PC),
130         .clk(CLOCK_IN),
```

```
131         .reset(RESET),
132         .inst(IF_INST)
133     );
134
135     register mainRegsiter(
136         .clk(CLOCK_IN),
137         .readReg1(IF_ID_Inst[25:21]),
138         .readReg2(IF_ID_Inst[20:16]),
139         .writeReg(MEM_WB_WriteReg),
140         .writeData(WB_WriteData),
141         .regWrite(MEM_WB_RegWrite),
142         .readData1(ID_ReadData1),
143         .readData2(ID_ReadData2),
144         .reset(RESET)
145     );
146
147     signext mainSignext(
148         .inst(IF_ID_Inst[15:0]),
149         .signextOut(ID_Signext)
150     );
151
152     Ctr mainCtr(
153         .opCode(IF_ID_Inst[31:26]),
154         .regDst(ID_RegDst),
155         .aluSrc(ID_AluSrc),
156         .memToReg(ID_MemToReg),
157         .regWrite(ID_RegWrite),
158         .memRead(ID_MemRead),
159         .memWrite(ID_MemWrite),
160         .branch(ID_Branch),
161         .aluOp(ID_AluOp),
162         .jump(ID_Jump)
163     );
164
165     aluCtr mainAluCtr(
166         .aluOp(ID_EX_AluOp),
167         .funct(ID_EX_Signext[5:0]),
168         .aluCtr(EX_AluCtr)
169     );
170
171     Alu mainAlu(
172         .input1(ID_EX_ReadData1),
173         .input2(EX_AluSrc_input_2),
174         .aluCtr(EX_AluCtr),
175         .zero(EX_Zero),
176         .aluRes(EX_AluRes)
177     );
178
179     dataMemory mainDataMem(
180         .clk(CLOCK_IN),
181         .address(EX_MEM_AluRes),
182         .writeData(EX_MEM_ReadData2),
183         .memWrite(EX_MEM_MemWrite),
184         .memRead(EX_MEM_MemRead),
185         .readData(MEM_ReadData)
186     );
187
188     wire IF_ID_Flush;
189     wire ID_EX_Stall;
190     wire EX_MEM_Stall;
```

```
191
192     hazardDetect mainHazardDetect(
193         .jump(ID_Jump),
194         .PcSrc(MEM_PCSrc),
195         .IF_ID_Flush(IF_ID_Flush),
196         .ID_EX_Stall(ID_EX_Stall),
197         .EX_MEM_Stall(EX_MEM_Stall)
198     );
199
200
201     always @ (posedge CLOCK_IN) begin
202     if (RESET)
203         PC <= 0;
204     else
205         PC <= IF_NextPc;
206
207     //IF_ID
208     IF_ID_PCPlusFour <= IF_PcPlusFour;
209     if (IF_ID_Flush) IF_ID_Inst <= 0;
210     else IF_ID_Inst <= IF_INST;
211
212     //ID_EX
213     if (ID_EX_Stall) begin
214         ID_EX_RegDst <= 0;
215         ID_EX_AluSrc <= 0;
216         ID_EX_MemToReg <= 0;
217         ID_EX_RegWrite <= 0;
218         ID_EX_MemRead <= 0;
219         ID_EX_MemWrite <= 0;
220         ID_EX_Branch <= 0;
221         ID_EX_AluOp <= 0;
222         ID_EX_PCPlusFour = IF_ID_PCPlusFour;
223         ID_EX_ReadData1 <= 0;
224         ID_EX_ReadData2 <= 0;
225         ID_EX_Signext <= ID_Signext;
226         IF_ID_Inst_20_16 <= 0;
227         IF_ID_Inst_15_11 <= 0;
228     end else begin
229         ID_EX_RegDst <= ID_RegDst;
230         ID_EX_AluSrc <= ID_AluSrc;
231         ID_EX_MemToReg <= ID_MemToReg;
232         ID_EX_RegWrite <= ID_RegWrite;
233         ID_EX_MemRead <= ID_MemRead;
234         ID_EX_MemWrite <= ID_MemWrite;
235         ID_EX_Branch <= ID_Branch;
236         ID_EX_AluOp <= ID_AluOp;
237         ID_EX_PCPlusFour = IF_ID_PCPlusFour;
238         ID_EX_ReadData1 <= ID_ReadData1;
239         ID_EX_ReadData2 <= ID_ReadData2;
240         ID_EX_Signext <= ID_Signext;
241         IF_ID_Inst_20_16 <= IF_ID_Inst[20:16];
242         IF_ID_Inst_15_11 <= IF_ID_Inst[15:11];
243     end
244
245     //EX_MEM
246     if (EX_MEM_Stall) begin
247         EX_MEM_MemToReg <= 0;
248         EX_MEM_RegWrite <= 0;
249         EX_MEM_MemRead <= 0;
250         EX_MEM_MemWrite <= 0;
251         EX_MEM_Branch <= 0;
```

```
252             EX_MEM_AddRes <= 0;
253             EX_MEM_Zero <= 0;
254             EX_MEM_AluRes <= 0;
255             EX_MEM_ReadData2 <= 0;
256             EX_MEM_WriteReg <= 0;
257         end else begin
258             EX_MEM_MemToReg <= ID_EX_MemToReg;
259             EX_MEM_RegWrite <= ID_EX_RegWrite;
260             EX_MEM_MemRead <= ID_EX_MemRead;
261             EX_MEM_MemWrite <= ID_EX_MemWrite;
262             EX_MEM_Branch <= ID_EX_Branch;
263             EX_MEM_AddRes <= EX_AddRes;
264             EX_MEM_Zero <= EX_Zero;
265             EX_MEM_AluRes <= EX_AluRes;
266             EX_MEM_ReadData2 <= ID_EX_ReadData2;
267             EX_MEM_WriteReg <= EX_WriteReg;
268         end
269
270         // MEM_WB
271         MEM_WB_MemToReg <= EX_MEM_MemToReg;
272         MEM_WB_RegWrite <= EX_MEM_RegWrite;
273         MEM_WB_ReadData <= MEM_ReadData;
274         MEM_WB_AluRes <= EX_MEM_AluRes;
275         MEM_WB_WriteReg <= EX_MEM_WriteReg;
276 end
277 endmodule
```