

SHANGHAI JIAOTONG UNIVERSITY



计算机系统结构实验报告 – Lab6

姓名：刘一凡

学号：518021910609

完成时间：2020/5/19

目录

[1. 概述 3](#_Toc40977760)

[1.1 实验名称 3](#_Toc40977761)

[1.2 实验目的 3](#_Toc40977762)

[2. 实验描述 3](#_Toc40977763)

[2.1 主要结构 3](#_Toc40977764)

[2.2 顶层模块Top 5](#_Toc40977765)

[2.2.1 模块描述 5](#_Toc40977766)

[2.2.2 变量定义 5](#_Toc40977767)

[2.2.3 停顿部分stall 6](#_Toc40977768)

[2.2.4 取指阶段（IF） 6](#_Toc40977769)

[2.2.5 译码阶段（ID） 7](#_Toc40977770)

[2.2.6 执行阶段（EXE） 9](#_Toc40977771)

[2.2.7 访存阶段（MEM） 10](#_Toc40977772)

[2.2.8 写回阶段（WB） 10](#_Toc40977773)

[2.2.9 Forwarding机制 11](#_Toc40977774)

[2.3 其他模块 12](#_Toc40977775)

[3. 仿真验证 13](#_Toc40977776)

[3.1 仿真代码 13](#_Toc40977777)

[3.2 指令内容 14](#_Toc40977778)

[3.3 仿真波形 16](#_Toc40977779)

[4. 实验心得 17](#_Toc40977780)

# 1. 概述

## 1.1 实验名称

简单的类MIPS 多周期流水线处理器设计与实现

## 1.2 实验目的

1．理解CPU Pipeline，了解流水线冒险(hazard)及相关性，设计基础流水线CPU

2．设计支持Stall 的流水线CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险、控制竞争和结构冒险

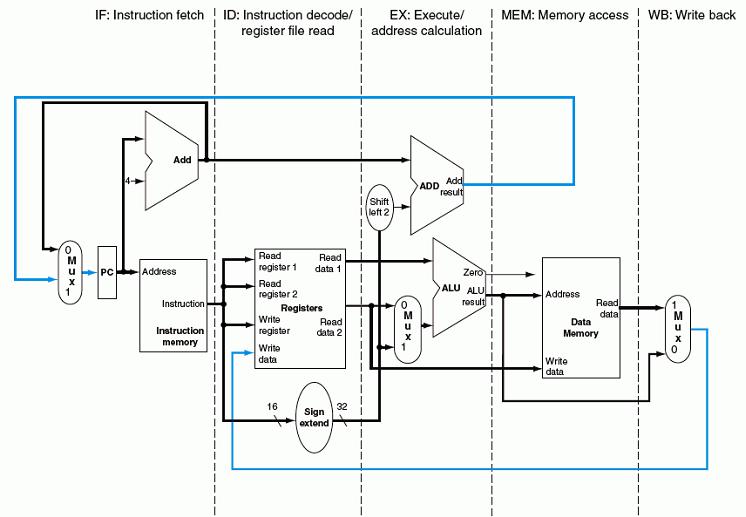
3．在2的基础上，增加Forwarding 机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能

4．在3的基础上，通过predict-not-taken 或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能

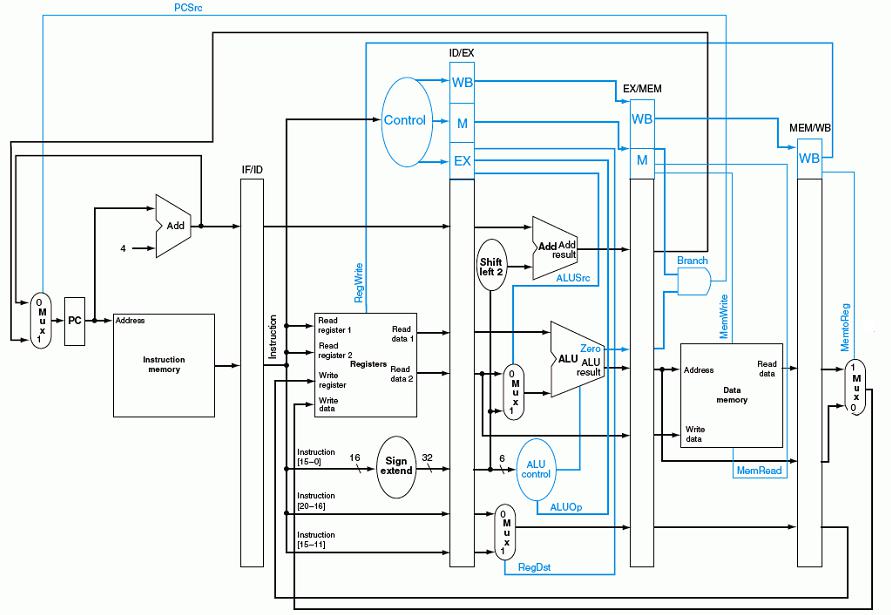
# 2. 实验描述

## 2.1 主要结构

在类MIPS多周期流水线处理器中，我们把一个指令分为5个阶段：取指令（IF），译码（ID），执行（EX），访存（MEM）和写回（WB），各个阶段完成不同的工作。一般而言，在前一指令完成某一阶段后，后一指令可以进行这一阶段，而不必等待前一指令完成所有阶段，这样可以大大提升处理器的效率。每个阶段进行的工作如下图所示。



为了使得各个阶段互不干扰地工作，我们需要将前一阶段产生的后一阶段所需要的数据和指令存储起来，因此，我们在每两个阶段之间引入了一些寄存器，如下图所示。例如，在IF/ID寄存器中，我们存储了PC+4的结果以及当前取出的指令，供译码及以后的阶段使用。此外控制单元所生成的控制信号也需要被存储在寄存器中，以控制相关的模块。



## 2.2 顶层模块Top

### 2.2.1 模块描述

顶层模块Top将一次指令的执行分为5个阶段：IF,ID,EX,MEM,WB，通过插入四级寄存器的方式，实现多周期流水线处理器。同时，引入了Stall机制解决数据冒险，控制竞争和结构冒险，增加Forwarding机制减少因数据竞争带来的停顿，提升流水线性能，并且采用了延时转移策略进一步提高处理器性能。在这次实验中，我尝试不加入实验5中所加入的Adder，Shift模块等比较简单的模块，而将它们直接放在Top模块中实现。Top模块以时钟信号Clk和复位信号reset作为输入。

### 2.2.2 变量定义

我们定义了寄存器中的变量。这些变量分为两类，一类是普通的寄存器中存储的变量，以寄存器名称（如IF/ID）+“\_”+小写字符表示；另一类是比较特殊的在流水线中需要使用的变量，以寄存器名称+“\_”+大写字符表示，并用wire定义，以表示其不同于普通寄存器变量的用途。可以认为这些特殊的变量一般是控制信号。具体的定义如以下代码所示。

1. //some stage registers
2. //IF/ID
3. reg [31:0] IFID\_pc4;//register pc+4
4. reg [31:0] IFID\_instruction;//register
5. //signals
6. wire [4:0] IFID\_INSTR\_RS=IFID\_instruction[25:21];
7. wire [4:0] IFID\_INSTR\_RT=IFID\_instruction[20:16];
8. wire [4:0] IFID\_INSTR\_RD=IFID\_instruction[15:11];
9. wire BRANCH;
11. //ID/EX
12. //registers
13. reg [31:0] IDEX\_read\_data1;
14. reg [31:0] IDEX\_read\_data2;
15. reg [31:0] IDEX\_sign\_extend;
16. reg [4:0] IDEX\_instruction\_rs, IDEX\_instruction\_rt, IDEX\_instruction\_rd;
17. reg [8:0] IDEX\_ctr;
19. //signals
20. wire [1:0] IDEX\_ALUOP=IDEX\_ctr[7:6];
21. wire IDEX\_MEM\_TO\_REG=IDEX\_ctr[0];
22. wire IDEX\_REG\_WRITE=IDEX\_ctr[1];
23. wire IDEX\_MEM\_WRITE=IDEX\_ctr[2];
24. wire IDEX\_MEM\_READ=IDEX\_ctr[3];
25. wire IDEX\_BRANCH=IDEX\_ctr[4];
26. wire IDEX\_ALUSRC=IDEX\_ctr[5];
27. wire IDEX\_REG\_DST=IDEX\_ctr[8];
29. //EX/MEM
30. //registers
31. reg [31:0] EXMEM\_write\_data,EXMEM\_alures;
32. reg [4:0] EXMEM\_ctr, EXMEM\_dst\_addr;
33. reg EXMEM\_zero;
34. //signals
35. wire EXMEM\_MEM\_TO\_REG=EXMEM\_ctr[0];
36. wire EXMEM\_REG\_WRITE=EXMEM\_ctr[1];
37. wire EXMEM\_MEM\_WRITE=EXMEM\_ctr[2];
38. wire EXMEM\_MEM\_READ=EXMEM\_ctr[3];
39. wire EXMEM\_BRANCH=EXMEM\_ctr[4];
41. //MEM/WB
42. //registers
43. reg [31:0] MEMWB\_read\_data;
44. reg [31:0] MEMWB\_alures;
45. reg [4:0] MEMWB\_dst\_addr;
46. reg [1:0] MEMWB\_ctr;
47. //signals
48. wire MEMWB\_REG\_WRITE=MEMWB\_ctr[1];
49. wire MEMWB\_MEM\_TO\_REG=MEMWB\_ctr[0];

### 2.2.3 停顿部分stall

在本次实验中，插入的停顿发生只需要考虑一种情况，就是在lw指令后的一条指令直接访问lw所加载的寄存器中的数据，由于此时数据还在中间的寄存器内，无法通过Forwarding机制解决冒险，只能暂时使流水线停顿以解决冒险。

1. //stalling
2. wire STALL=(IFID\_INSTR\_RT==IDEX\_instruction\_rt|IFID\_INSTR\_RS==IDEX\_instruction\_rt)&IDEX\_MEM\_READ;

### 2.2.4 取指阶段（IF）

在取指阶段，需要将下一步指令取出并进行PC+4的操作以及更新内部寄存器。需要注意的是，由于PC在时钟上升沿更新，插入停顿时程序不能向前执行，所以需要检测停顿信号。当检测到有停顿时，需要清除当前取出的指令。

1. //IF STAGE
2. reg [31:0] PC\_NOW;wire[31:0] PC4, PC\_NEXT, BRANCH\_ADDRESS, INSTRUCTION;
3. assign PC4=PC\_NOW+4;
4. //update next pc
5. Mux32 nextpc(
6. .SEL(BRANCH),
7. .INPUT1(PC4),
8. .INPUT2(BRANCH\_ADDRESS),
9. .OUTPUT(PC\_NEXT)
10. );
11. //fetch instruction
12. InstrMemory fetch\_instr (
13. .address(PC\_NOW),
14. .instr(INSTRUCTION)
15. );
17. always @ (posedge clk)
18. begin
19. **if**(!STALL)
20. begin
21. //update registers
22. IFID\_pc4 <= PC4;
23. IFID\_instruction <= INSTRUCTION;
24. PC\_NOW <= PC\_NEXT;
25. end
26. **if**(BRANCH)
27. //if there is a branch, we need to stop
28. IFID\_instruction<=0;
29. end

### 2.2.5 译码阶段（ID）

在译码阶段中，流水线的控制模块生成控制信号，寄存器模块读入地址输入寄存器中的数据，更新相关的内部寄存器。此外还进行了有符号扩展以及跳转的判定操作（我们将跳转操作在这一阶段进行了实现）。由于在插入停顿时，不能对寄存器以外的其他模块的状态进行修改，所以此时所有存储在流水线寄存器中的控制信号需要设置为0。

1. //DECODE STAGE
2. wire [8:0] CTR\_SIGNAL;
3. wire [31:0] READ\_DATA1,READ\_DATA2,WRITE\_DATA\_REG;
4. wire [31:0] SIGN\_EXTEND;
5. wire [31:0] EXTEND\_SHIFT;
6. //generate control signals
7. Ctr uu0(
8. .opcode(IFID\_instruction[31:26]),
9. .RegDst(CTR\_SIGNAL[8]),
10. .ALUOp(CTR\_SIGNAL[7:6]),
11. .ALUSrc(CTR\_SIGNAL[5]),
12. .Branch(CTR\_SIGNAL[4]),
13. .MemRead(CTR\_SIGNAL[3]),
14. .MemWrite(CTR\_SIGNAL[2]),
15. .RegWrite(CTR\_SIGNAL[1]),
16. .MemToReg(CTR\_SIGNAL[0])
17. );
18. //update register file
19. Registers update\_reg(
20. .reset(reset),
21. .clock\_in(clk),
22. .readReg1(IFID\_INSTR\_RS),
23. .readReg2(IFID\_INSTR\_RT),
24. .writeReg(MEMWB\_dst\_addr),
25. .writeData(WRITE\_DATA\_REG),
26. .regWrite(MEMWB\_REG\_WRITE),
27. .readData1(READ\_DATA1),
28. .readData2(READ\_DATA2)
29. );
30. //sign extend module
31. signext extt(
32. .inst(IFID\_instruction[15:0]),
33. .data(SIGN\_EXTEND)
34. );
36. assign EXTEND\_SHIFT=SIGN\_EXTEND<<2;//sign extended result
37. assign BRANCH=CTR\_SIGNAL[4]&(READ\_DATA1==READ\_DATA2);//check whether should branch
38. assign BRANCH\_ADDRESS=EXTEND\_SHIFT+IFID\_pc4;//branch address
39. //update registers
40. always @ (posedge clk)
41. begin
42. IDEX\_read\_data1<=READ\_DATA1;
43. IDEX\_read\_data2<=READ\_DATA2;
44. IDEX\_sign\_extend<=SIGN\_EXTEND;
45. IDEX\_instruction\_rs<=IFID\_INSTR\_RS;
46. IDEX\_instruction\_rt<=IFID\_INSTR\_RT;
47. IDEX\_instruction\_rd<=IFID\_INSTR\_RD;
48. IDEX\_ctr<=STALL? 0:CTR\_SIGNAL;//operate the stalling
49. end

### 2.2.6 执行阶段（EXE）

该部分的功能与单周期处理器基本相同，生成ALU控制信号以及执行对应的ALU操作，更新相关的内部寄存器。但是由于需要考虑Forwarding机制，ALU的两个操作数以及待写入内存中的数据都不能确定，我将在之后的代码中给出确定它们的表达式。

1. wire [31:0] ALU\_RES;wire [3:0] ALU\_CTR; wire [4:0] REG\_DST; wire ZERO;
2. //generate alu control signal
3. AluCtr generate\_alu(
4. .aluOp(IDEX\_ALUOP),
5. .funct(IDEX\_sign\_extend[5:0]),
6. .aluCtr(ALU\_CTR)
7. );
8. //alu execution
9. Alu alu(
10. .input1(ALUSRC1),
11. .input2(ALUSRC2),
12. .aluCtr(ALU\_CTR),
13. .zero(ZERO),
14. .aluRes(ALU\_RES)
15. );
17. Mux5 muxx(
18. .SEL(IDEX\_REG\_DST),
19. .INPUT1(IDEX\_instruction\_rt),
20. .INPUT2(IDEX\_instruction\_rd),
21. .OUTPUT(REG\_DST)
22. );
23. //update
24. always @ (posedge clk)
25. begin
26. EXMEM\_write\_data<=WRITE\_DATA\_MEM;
27. EXMEM\_alures<=ALU\_RES;
28. EXMEM\_ctr<=IDEX\_ctr[4:0];
29. EXMEM\_dst\_addr<=REG\_DST;
30. EXMEM\_zero<=ZERO;
31. end

### 2.2.7 访存阶段（MEM）

访存阶段的实现和单周期处理器基本一致，需要进行内存的读写操作以及更新内部寄存器。

1. //MEMORY STAGE
2. wire [31:0] READ\_DATA\_MEM;
3. //memory access
4. DataMemory readd(
5. .Clk(clk),
6. .address(EXMEM\_alures),
7. .writeData(EXMEM\_write\_data),
8. .memWrite(EXMEM\_MEM\_WRITE),
9. .memRead(EXMEM\_MEM\_READ),
10. .readData(READ\_DATA\_MEM)
11. );
12. //update
13. always @ (posedge clk)
14. begin
15. MEMWB\_read\_data<=READ\_DATA\_MEM;
16. MEMWB\_alures<=EXMEM\_alures;
17. MEMWB\_dst\_addr<=EXMEM\_dst\_addr;
18. MEMWB\_ctr<=EXMEM\_ctr[1:0];
19. end

### 2.2.8 写回阶段（WB）

写回阶段的实现和单周期处理器基本一致，需要更新内部寄存器。

1. //WRITE BACK STAGE
2. Mux32 muxx2(
3. .SEL(MEMWB\_MEM\_TO\_REG),
4. .INPUT1(MEMWB\_alures),
5. .INPUT2(MEMWB\_read\_data),
6. .OUTPUT(WRITE\_DATA\_REG)
7. );
8. //reset
9. always @ (reset)
10. begin
11. PC\_NOW=0;
12. IFID\_pc4=0;//register pc+4
13. IFID\_instruction=0;
14. IDEX\_read\_data1=0;
15. IDEX\_read\_data2=0;
16. IDEX\_sign\_extend=0;
17. IDEX\_instruction\_rs=0;
18. IDEX\_instruction\_rt=0;
19. IDEX\_instruction\_rd=0;
20. IDEX\_ctr=0;
21. EXMEM\_write\_data=0;
22. EXMEM\_alures=0;
23. EXMEM\_ctr=0;
24. EXMEM\_dst\_addr=0;
25. EXMEM\_zero=0;
26. end

### 2.2.9 Forwarding机制

Forwarding机制是多周期流水线处理器中非常重要的一个机制。在Forwarding机制中，我们需要判断当前指令和下一条指令是否符合forwarding的条件，若满足条件，则根据指令的具体内容得出ALU两个操作数以及访存数据的表达式。判断是否满足forwarding条件的表达式在系统结构书中已经给出。由于ALU第二个操作数可能接受立即数，因此需要额外嵌套一层关于ALUSRC的条件表达式。再考虑将要写入内存的数据，由于该数据是从ALUSRC2连出来的且不存在立即数，所以把第二个操作数的立即数逻辑运算部分删去就可以得到写入数据的逻辑表达式了。

1. //forward
2. wire FORWARD\_MEM1,FORWARD\_MEM2,FORWARD\_EX1,FORWARD\_EX2;
3. assign FORWARD\_MEM1=MEMWB\_REG\_WRITE & MEMWB\_dst\_addr!=0 & !(EXMEM\_REG\_WRITE & EXMEM\_dst\_addr != 0 & EXMEM\_dst\_addr != IDEX\_instruction\_rs) &
4. MEMWB\_dst\_addr == IDEX\_instruction\_rs;
5. assign FORWARD\_MEM2=MEMWB\_REG\_WRITE & MEMWB\_dst\_addr != 0 & !(EXMEM\_REG\_WRITE & EXMEM\_dst\_addr != 0 & EXMEM\_dst\_addr != IDEX\_instruction\_rt) &
6. MEMWB\_dst\_addr == IDEX\_instruction\_rt;
7. assign FORWARD\_EX1=EXMEM\_REG\_WRITE & EXMEM\_dst\_addr!=0 & EXMEM\_dst\_addr == IDEX\_instruction\_rs;
8. assign FORWARD\_EX2=EXMEM\_REG\_WRITE & EXMEM\_dst\_addr!=0 & EXMEM\_dst\_addr == IDEX\_instruction\_rt;
10. //EXECUTION STAGE
11. //use forward to update src and write\_data
12. wire [31:0] WRITE\_DATA\_MEM= FORWARD\_EX2 ? EXMEM\_alures :
13. FORWARD\_EX2 ? EXMEM\_alures : FORWARD\_MEM2 ? WRITE\_DATA\_REG : IDEX\_read\_data2;
14. wire [31:0] ALUSRC1=FORWARD\_EX1 ? EXMEM\_alures : FORWARD\_MEM1 ? WRITE\_DATA\_REG : IDEX\_read\_data1;
15. wire [31:0] ALUSRC2=IDEX\_ALUSRC ? IDEX\_sign\_extend : FORWARD\_EX2 ? EXMEM\_alures :
16. FORWARD\_EX2 ? EXMEM\_alures : FORWARD\_MEM2 ? WRITE\_DATA\_REG : IDEX\_read\_data2;

## 2.3 其他模块

本次实验中所用到的Alu模块，AluCtr模块，Ctr模块，Mux模块，inst\_memory模块，data\_memory模块，signext模块，Registers模块都与之前的基本相同。不同的是，由于在此次实验中，我尝试采用一行2个十六进制数（8个二进制数）的方式存储数据和指令，数据存储模块和指令存储模块需要稍加修改以适应这一改动。

1. module DataMemory(
2. input Clk,
3. input [31:0] address,
4. input [31:0] writeData,
5. input memWrite,
6. input memRead,
7. output [31:0] readData
8. );
10. reg [7:0] memFile[0:31];
11. reg [31:0] readData;
13. always @ (address or memRead)
14. begin
15. **if** (memRead)
16. readData = {memFile[address+3], memFile[address+2], memFile[address+1], memFile[address]};
17. **else**
18. readData = 0;
19. end
21. always @ (negedge Clk)
22. begin
23. **if** (memWrite)
24. begin
25. memFile[address] <= writeData[7:0];
26. memFile[address+1] <= writeData[15:8];
27. memFile[address+2] <= writeData[23:16];
28. memFile[address+3] <= writeData[31:24];
29. end
30. end
32. endmodule
33. module InstrMemory(
34. input [31:0] address,
35. output [31:0] instr
36. );
38. reg [7:0] instrFile[0:63];
40. assign instr = {instrFile[address+3], instrFile[address+2],
41. instrFile[address+1], instrFile[address]};
43. endmodule

# 3. 仿真验证

## 3.1 仿真代码

仿真代码部分将时钟周期设为100ns，重置高电平的信号保持50ns。

1. module Top\_tb(
3. );
4. reg clk,reset;
5. always #50
6. clk=!clk;
8. Top top(.clk(clk),.reset(reset));
10. initial begin
11. //read file, absolute path
12. $readmemh("C:/Archlabs/Lab06/inst.txt", top.fetch\_instr.instrFile);
13. $readmemh("C:/Archlabs/Lab06/mem.txt", top.readd.memFile);
14. clk = 1;
15. reset = 1;
16. #50 reset = 0;
17. end
18. endmodule

## 3.2 指令内容

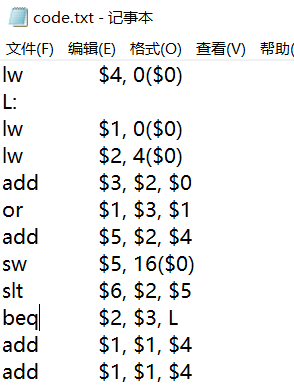
初始的指令文件如下所示：

1. 00
2. 00
3. 04
4. 8c
5. 00
6. 00
7. 01
8. 8c
9. 04
10. 00
11. 02
12. 8c
13. 20
14. 18
15. 40
16. 00
17. 25
18. 08
19. 61
20. 00
21. 20
22. 28
23. 44
24. 00
25. 10
26. 00
27. 05
28. ac
29. 2a
30. 30
31. 45
32. 00
33. f8
34. ff
35. 43
36. 10
37. 20
38. 08
39. 24
40. 00
41. 20
42. 08
43. 24
44. 00

初始的内存数据如下所示：

1. 03
2. 00
3. 00
4. 00
5. 04
6. 00
7. 00
8. 00
9. 00
10. 00
11. 00
12. 00

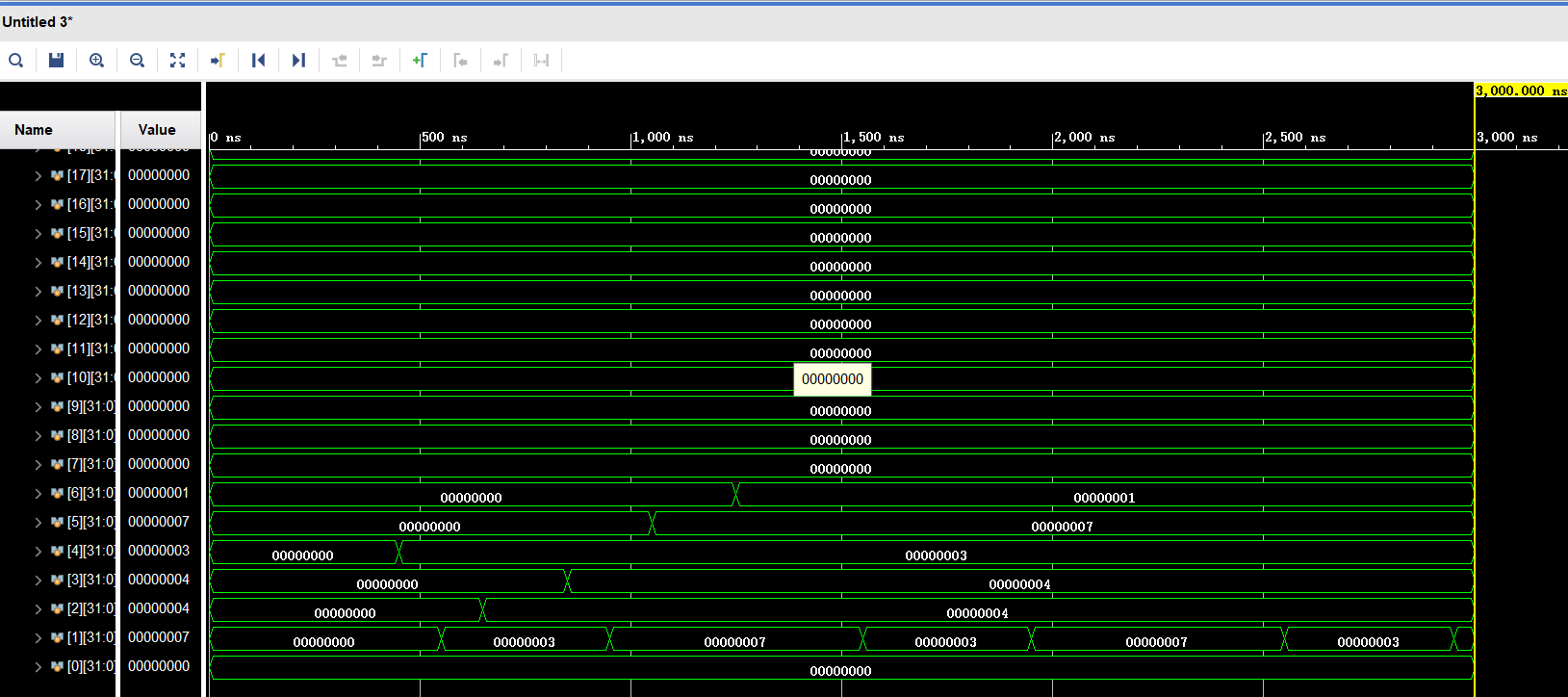
实际的MIPS指令如下所示：

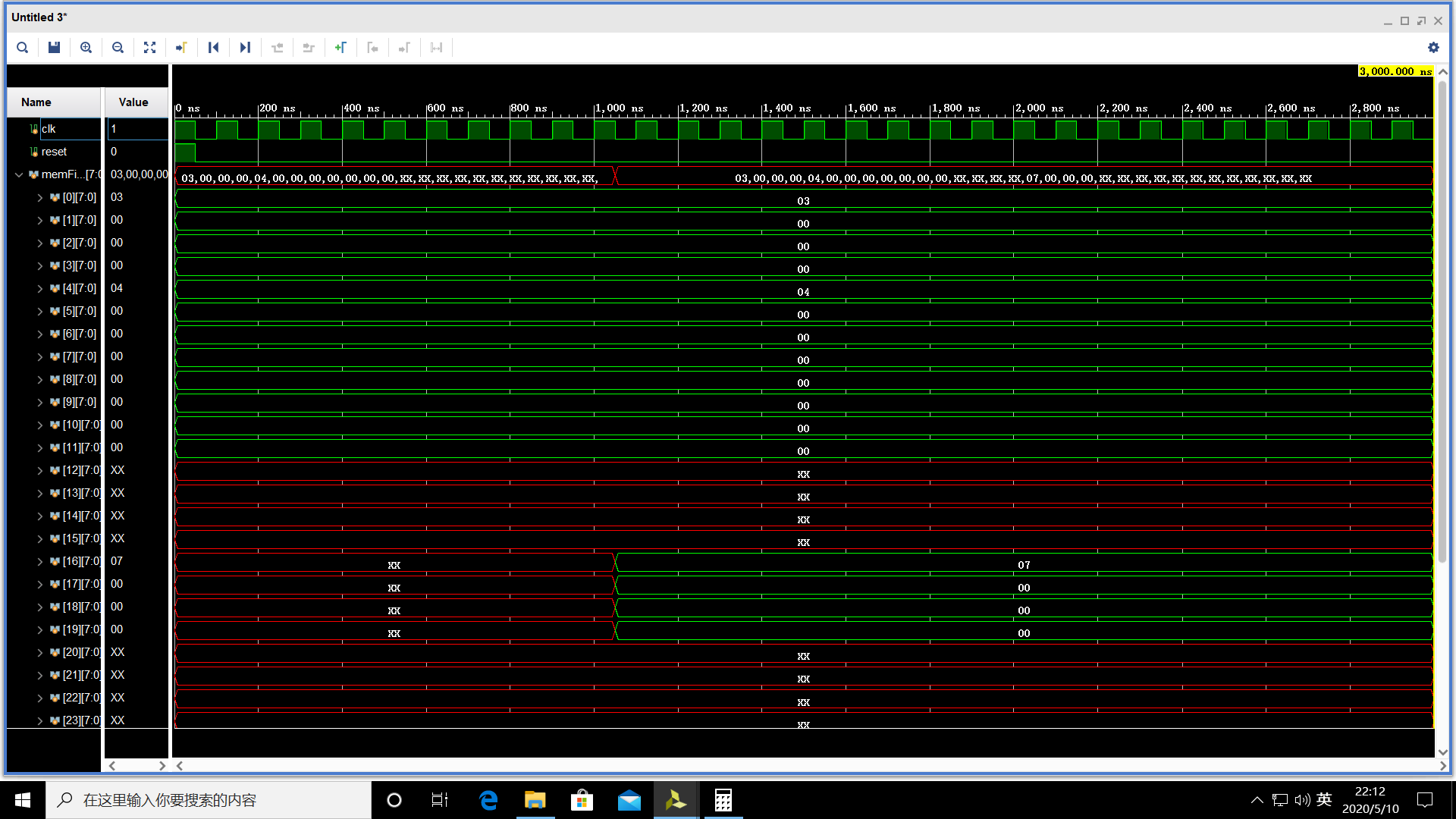


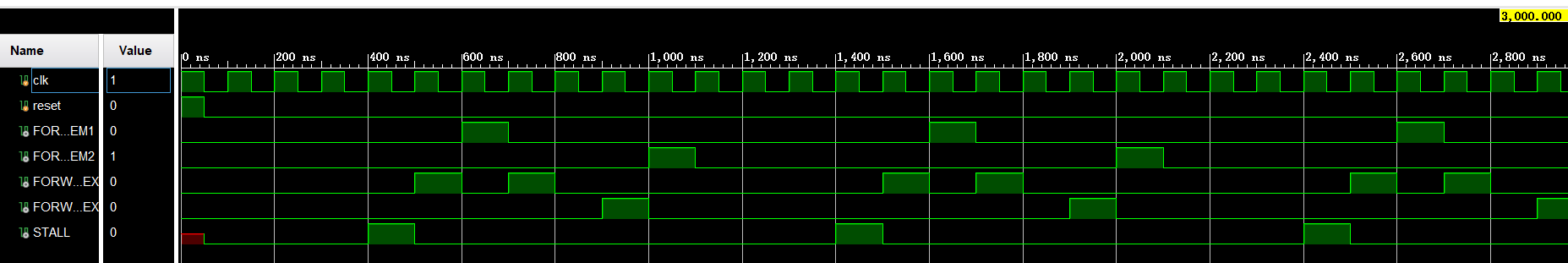
执行第1条指令后，$4为3；执行第2条指令后，$1为3；执行第3条指令后，$2为4；执行第4条指令后，$3为4；执行第5条指令后，$1变为7；执行第6条指令后，$5为7；执行第7条指令后，内存中地址为16的数据应变为7；执行第8条指令后，$6为1；执行第9条指令后，跳转到L处继续执行；beq后面的两个add指令不会执行。需要注意的是，这里的txt文件在代码中都以绝对路径进行读取，所以在验证的时候可能需要修改路径。

## 3.3 仿真波形

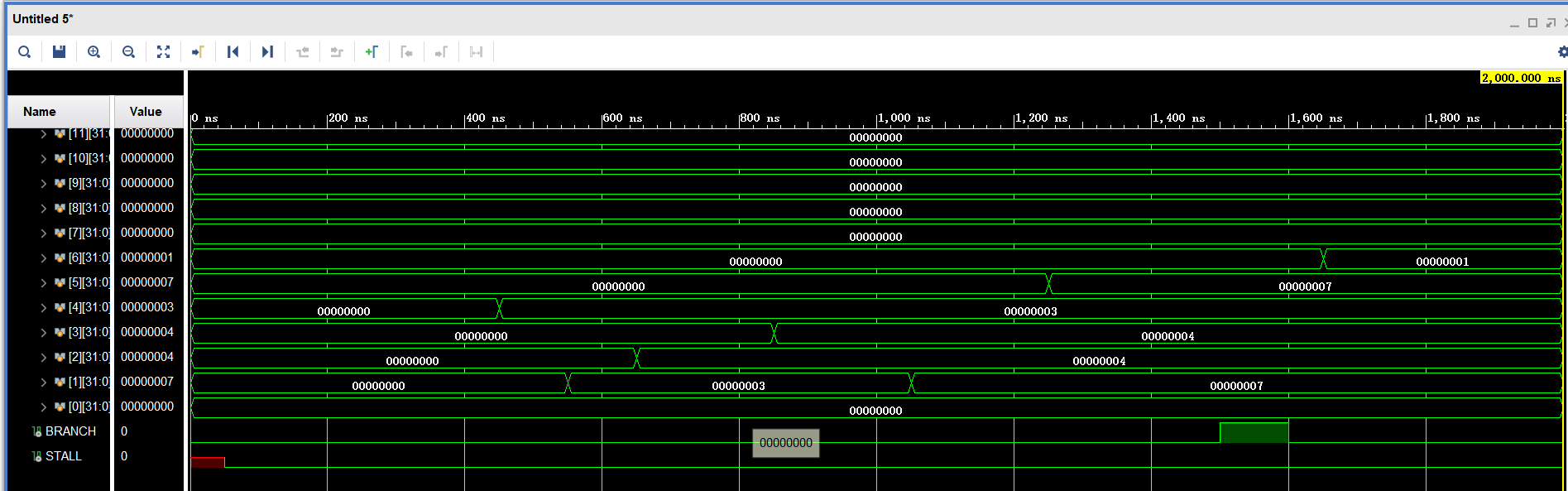
下面是此流水线运行3000ns后的仿真波形图。从图中可以看出，仿真实验运行的结果和预期相一致，并且可以看到stall和forwarding中使用的变量的值和变化均符合预期。成功地用硬件方法（加入stall机制，forwarding机制和延时转移机制）实现了多周期流水线处理器。

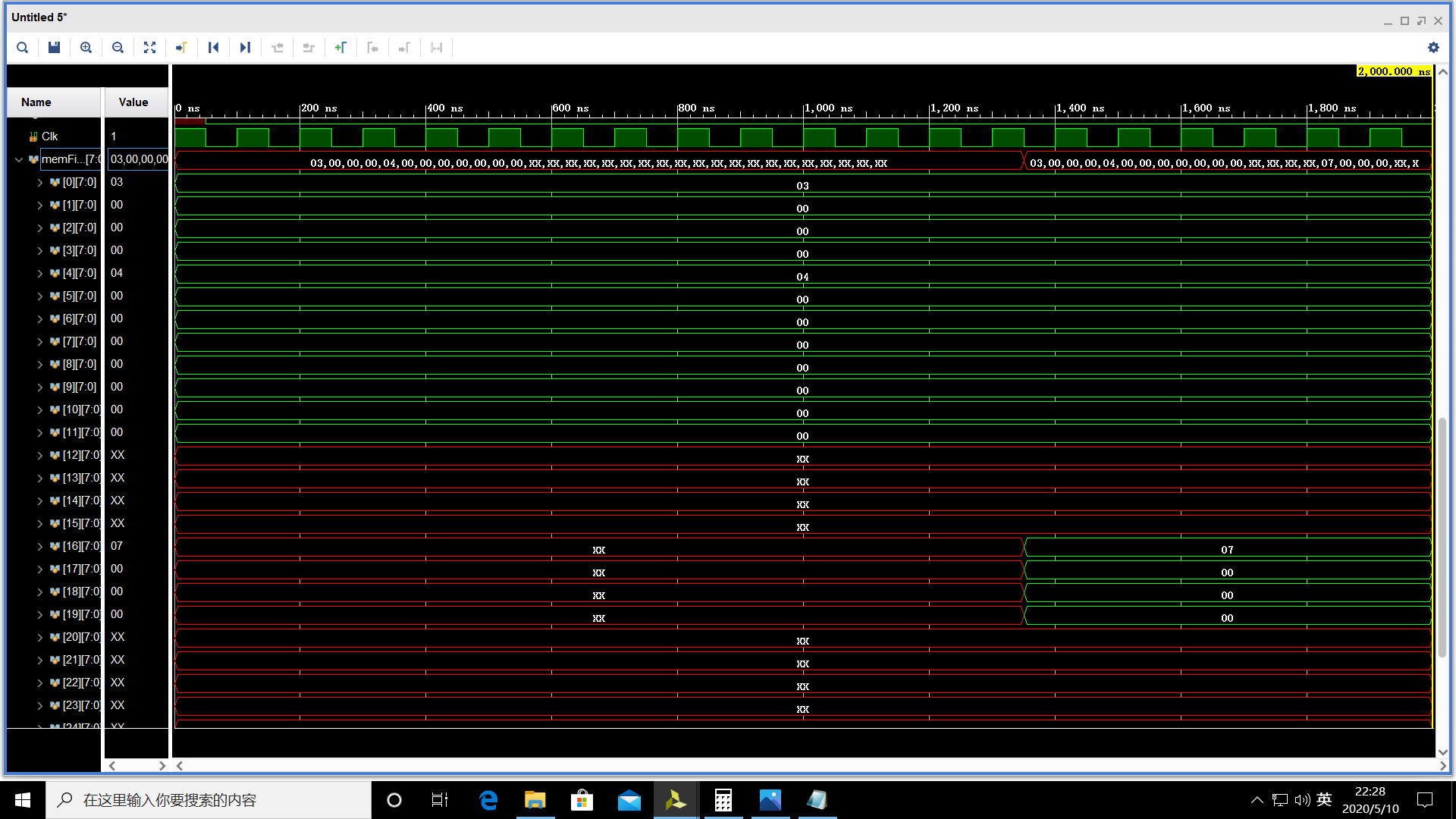






下面是用软件法（修改指令以解决冲突和冒险）实现多周期流水线处理器的仿真波形图。为了模拟方便起见，我在可能产生冒险冲突的两条指令之间加入了一些nop指令（指令数字全为0，什么事都不做）并进行了仿真模拟2000ns。可以看到，这样就不会有stall了，也可以正确地执行指令并得到正确的结果。软件法实验成功。





# 4. 实验心得

本次实验是对实验5的继承改进。实验5实现了单周期处理器，实验6要求实现多周期处理器，实现难度上比实验5难了不少，需要考虑逻辑上的合理性与许多细节的实现。

本次实验的第一个难点是变量的命名。由于需要四组额外的寄存器去存储两个阶段之间的数据和控制信号，所以本次实验中的变量是很多的，如果不制定一套命名规则的话，在之后实例化模块的阶段就很有可能出错。因此，经过反复思考与查阅一般的命名规则，我采用了变量所属的寄存器类型（如IF/ID）加上具体的变量名的命名规则，这样可以区分不同内部寄存器的变量。为了区分一般的存储内容和控制信号等在后面的阶段具有特殊功能的变量，我用reg类型和wire类型作了区分，并且前者用小写命名，后者用大写命名，这样在之后使用变量的时候就十分清晰了。

本次实验的第二个难点是对于冒险的处理，虽然在理论课上我已经学过了stall和forwarding机制，但是在实现时仍然遇到了一些困难。特别是在forwarding的实现上，我一开始还不是很明白它的逻辑表达式以及在整个流水线中起到的作用。在查阅了课本以及网络上的一些讲解后，我终于会根据我自己定义的变量和信号写出若干forwarding条件下的逻辑表达式了。

本次实验的第三个难点是流水线的逻辑和实现。多周期流水线处理器不同于简单的单周期处理器，由于需要同时完成不同指令的不同阶段，因此需要分清每个阶段流水线进行的工作，不能有交叉和重复，数据和信号在不同阶段间的传递和保护也是特别需要注意的，稍不留神就可能出现前一阶段产生的信号在传输过程中丢失或者被改变的情况。

本次实验的第四个难点是出现问题后的调试。由于流水线本来就较为复杂，所以难免在模拟时会出错。出错后首先需要定位具体出错的指令和阶段。由于变量非常多，所以只能抓住几个主要的变量以及一些控制信号进行查错。在一次进行仿真模拟时我的lw指令始终无法正确执行，在一步步进行debug后，我最终发现是我的寄存器写入的控制信号在传递的过程中丢失了，纠正了这一错误后，仿真波形基本符合预期了。在调试过程中，需要非常耐心和细心，才能找出错误的原因。

美中不足的是，由于时间问题，我没能做后面的选做实验，这还是有些遗憾的。但是在这次实验中，我对于多周期流水线处理器有了很深入的理解，不仅知道了它的工作原理，还自己实现了类似的功能，在看到正确的波形后内心还是有成就感的。实验的过程也引导我将课堂上所学的知识运用到实践中，进行深入思考和探究，这对于我之后的学习和科研都很有帮助。

实验课程到这里也结束了。这六次实验课难度由易到难，逐渐上升。前几次实验主要是为了让我们了解Verilog的基本语法和基本的仿真实验步骤，按部就班地做就可以了。但是随着课程的进行，内容难度逐渐升高，复杂性也越来越高，这就需要我们查阅更多其他的资料，自己学习更多相关的知识，将理论课程的内容融会贯通，进行主动的探索和创新，才能正确完成实验。这次的实验课程让我学会了很多，从简单的模块实现到流水线处理器的整体实现，从学习Verilog语法到进行程序的debug……同时这些实验也培养了我的耐心和韧性，让我在遇到困难时积极思考，想出应对方法。最后，我也要十分感谢老师和同学在我遇到难以解决的问题时给予我的帮助。