实验六 CPU综合设计

1. **实验目的**

1 掌握复杂系统设计方法。

2 深刻理解计算机系统硬件原理。

**二、实验内容**

1）设计一个基于MIPS指令集的CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；

2）CPU需要包含寄存器组、RAM模块、ALU模块、指令译码模块；

3）该CPU能运行基本的汇编指令；（D~C+）

以下为可选内容：

4）实现多周期CPU（B-~B+）；

5）实现以下高级功能之一（A-~A+）：

(1)实现5级流水线CPU；

(2)实现超标量；

(3)实现4路组相联缓存；

可基于RISC V 、ARM指令集实现。

**如发现代码为抄袭代码，成绩一律按不及格处理。**

**三、实验要求**

编写相应测试程序，完成所有指令测试。

**四、实验代码及结果**

**1、实验代码分析**

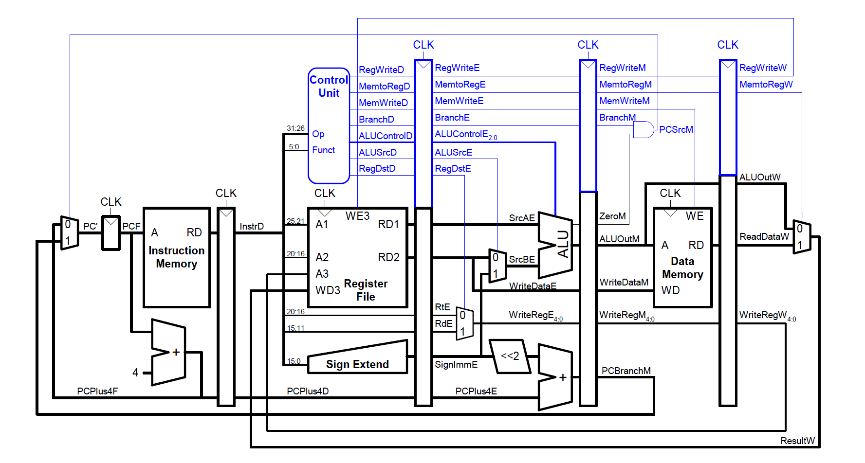
实现了五级流水线MIPS指令CPU。传统的五级流水线阶段，分别是取指、译码、执行、访存、回写。CPU主要分为controller（控制器）模块、数据通路（datapath）模块和冒险控制模块（hazard\_unit）。接下来介绍主要模块：

1. Controller

模块内部的逻辑包括使用maindec\_new模块对Op和Funct进行解码生成ControlsD和ALUOp，使用aludec模块对Funct和ALUOp进行解码生成ALUControlD，以及使用触发器模块对控制信号进行寄存器传递和更新。最后，将这些控制信号分配给输出端口。利用寄存器存储各个阶段的控制信号，提供给冒险控制模块判断是否需要流水线前推和暂停。



1. Datapath

引入多路选择器、加法器等门级组件，通过对原理图的理解，分析单条（单类型）指令在数据通路中的执行路径，依次连接对应端口。根据五个流水线阶段在每个执行阶段加入触发器，使得每个周期执行一个阶段，得到的结果送往下一个周期进行执行，同时下一条指令执行一个阶段。将控制信号接入各个多路选择器实现数据信号的选择，冒险信号同理接入多路选择器实现数据的前推和暂停。

实验代码：

module datapath\_new(

input wire clka,rst,

output wire [31:0] instr,

output wire [31:0]mem\_rdata,

output wire [31:0] pc, alu\_out,WriteDataM,

input wire jump,regwriteE,regwriteM,regdst,alusrc,branch,memtoregM,memtoregE,memtoregW,regwriteW,

input wire [2:0] alucontrol,

output wire zero,

output wire[1:0] forwardAE,forwardBE,

output wire[31:0] pc\_plusD,pc\_plusE,pc\_branch,pc\_branchM,

output wire stallD,stallF,flushE,lwstall,branch\_stall

);

wire [31:0] pc\_plus,rd1,rd2,imm\_extend,pc\_jump;

wire [31:0] alu\_srcB,wd3,imm\_sl2,pc\_next;

wire [4:0] write2regE,write2regM,write2regW;

wire zeroM,pcsrc;

wire [31:0] eq1,eq2;

wire eqout;

assign pcsrc = branch & eqout;

//assign mem\_wdata = rd2;

wire [31:0] instrD,rd1D,rd2D,rd1E,rd2E,imm\_extendE,alu\_outM,alu\_outW,mem\_rdataW;

wire[4:0] rsD,rtD,rdD,rsE,rtE,rdE,rdM,rtM,rtW,rdW;

wire forwardAD,forwardBD;

//pc

pc\_new pc\_1(

.clka(clka),

.rsta(rst),

.en(~stallF),

.pc\_in(pc\_jump),

.pc\_out(pc)

);

// 跳转目标由(PC+4) 的最高4 位与立即数instr[25:0]) 左移2 位后的值拼接得到

mux mux\_pc\_jump(

.A({pc\_plus[31:28], instr[25:0], 2'b00}),

.B(pc\_next),

.sel(jump), //jump

.Y(pc\_jump)

);

//左移两位得到跳转pc所需立即数

sl2 sl2(

.a(imm\_extendE),

.y(imm\_sl2)

);

//获得pc\_branch分支

adder adder\_branch(

.A(pc\_plusE),

.B(imm\_sl2),

.SUM(pc\_branch)

);

//选择下一个pc信号

mux mux\_pc(

.A(pc\_branch),

.B(pc\_plus),

.sel(pcsrc), //pcsrc

.Y(pc\_next)

);

//

mux mux\_forward(rd1D,alu\_outM,forwardAD,eq1);

mux mux\_forward2(rd2D,alu\_outM,forwardBD,eq2);

equal equal(eq1,eq2,eqout);

//adder

adder adder(

.A(pc),

.B(32'd4),

.SUM(pc\_plus)

);

//F-D stage

flopenrc #(32) floprc\_1 (clka,rst,~stallD,1'b0,instr,instrD);

flopenrc #(32) floprc\_2 (clka,rst,~stallD,1'b0,pc\_plus,pc\_plusD);

//inst\_ram 指令寄存器

inst\_ram inst\_ram(

.Addr(pc/4),

.CLK(clka),

.RD(instr)

);

//sign\_extend 符号拓展

sign\_extend sign\_extend(

.in(instrD[15:0]),

.out(imm\_extend)

);

//

assign rtD = instrD[20:16];

assign rdD = instrD[15:11];

assign rsD = instrD[25:21];

//regfile

regfile regfile(

.CLK(clka),

.WE3(regwrite), //regwrite

.RA1(instrD[25:21]), //base

.RA2(instrD[20:16]), //load rt

.WA3(write2regW), //store rt

.WD3(wd3),

.RD1(rd1D),

.RD2(rd2D)

);

//stage 2 D-E

flopenrc #(32) floprc\_1E(clka,rsta,1'b1,flushE,rd1D,rd1E);

flopenrc #(32) floprc\_2E(clka,rsta,1'b1,flushE,rd2D,rd2E);

flopenrc #(5) floprc\_3E(clka,rsta,1'b1,flushE,rtD,rtE);

flopenrc #(5) floprc\_4E(clka,rsta,1'b1,flushE,rdD,rdE);

flopenrc #(32) floprc\_5E(clka,rsta,1'b1,flushE,pc\_plusD,pc\_plusE);

flopenrc #(32) floprc\_6E(clka,rsta,1'b1,flushE,imm\_extend,imm\_extendE);

flopenrc #(5) floprc\_7E(clka,rsta,1'b1,flushE,rsD,rsE);

//mux2 for wa3

mux\_5 mux\_wa3(

.A(rtE),

.B(rdE),

.sel(regdst), //regdst

.Y(write2regE)

);

//mux2 for alu\_srcB

mux mux(

.A(imm\_extendE),

.B(rd2),

.sel(alusrc), //alusrc

.Y(alu\_srcB)

);

//三选一

mux\_3 #(32) mux\_3\_0(rd1E,wd3,alu\_outM,forwardAE,rd1);

mux\_3 #(32) mux\_3\_1(rd2E,wd3,alu\_outM,forwardBE,rd2);

//alu

alu alu(

.A(rd1),

.B(alu\_srcB),

.ALU\_CTRL(alucontrol),//alucontrol

.ALU\_OUT(alu\_out),

.zero(zero), //

.overflow()

);

//stage3 E-M

floprc #(32) floprc\_1M(clka,rsta,1'b0,alu\_out,alu\_outM);

floprc #(1) floprc\_2M(clka,rsta,1'b0,zero,zeroM);

floprc #(32) floprc\_3M(clka,rsta,1'b0,rd2E,WriteDataM);

floprc #(32) floprc\_4M(clka,rsta,1'b0,pc\_branch,pc\_branchM);

floprc #(5) floprc\_5M(clka,rsta,1'b0,write2regE,write2regM);

// floprc #(5) floprc\_6M(clka,rsta,1'b0,rdE,rdM);

//mux2 for wd3

mux mux\_wd3(

.A(mem\_rdataW),

.B(alu\_outW),

.sel(memtoregW), //memtoreg

.Y(wd3)

);

//data\_ram

data\_ram data\_ram(

.WriteData(WriteDataM),

.ReadData(mem\_rdata),

.Addr(alu\_outM/4),

.R\_W(R\_W),

.CLK(clk)

);

//stage4 m-w

floprc #(32) floprc\_1W (clka,rst,1'b0,alu\_outM,alu\_outW);

floprc #(32) floprc\_2W(clka,rst,1'b0,mem\_rdata,mem\_rdataW);

floprc #(5) floprc\_3W(clka,rst,1'b0,write2regM,write2regW);

// floprc #(5) floprc\_4W(clka,rst,1'b0,rdM,rdW);

//hazrd detection

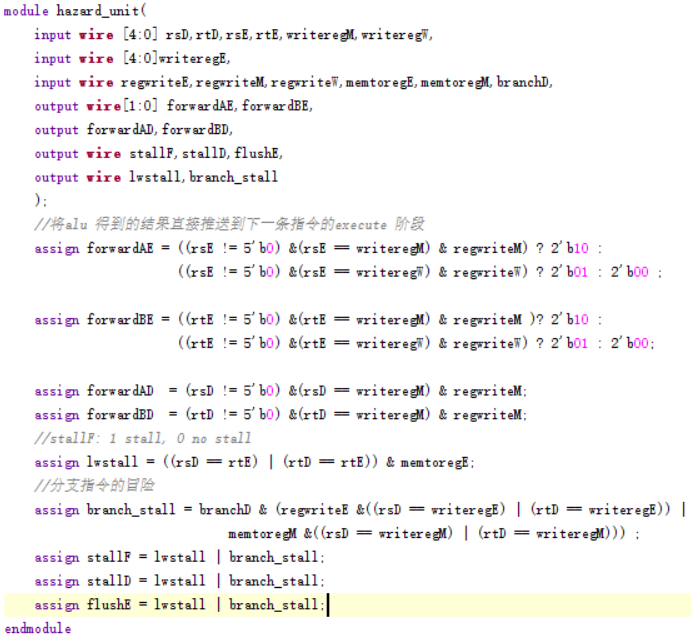
hazard\_unit hazard(rsD,rtD,rsE,rtE,write2regM,write2regW,write2regE,regwriteE,regwriteM,regwriteW,memtoregE,memtoregM,branch,forwardAE,forwardBE,forwardAD,forwardBD,stallF,stallD,flushE,lwstall,branch\_stall);

endmodule

1. hazard\_unit

而在流水线处理器中，由于当前指令可能取决于前一条指令的结果，但此时前一条指令并未执行到产生结果的阶段，这时候，就产生了冒险。冒险分为：数据冒险：寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据；控制冒险：下一条要执行的指令还未确定，就按照PC自增顺序执行了本不该执行的指令(由分支指令引起)。

1. 针对前一条指令还未到达执行阶段回写但后续指令依赖该指令结果的问题：通过判断写寄存器堆的地址和ALU地址是否构成依赖，模块输出前推信号给多路选择器将已计算出来的输入到ALU中。
2. 针对分支指令产生的控制冒险问题，因为分支在访存阶段才决定是否跳转，需要消除这三条指令产生的影响。通过将分支指令的判断提前至译码阶段，增加数据前推和流水线暂停模块实现。



**2、仿真分析**

****

1. 整体分析

有时序仿真图可以看出每隔一个时钟周期，PC根据指令执行 +4 或跳转。Instr输出每一次读取的指令。各指令代表：

1. 00000000 NOP

2. 02309021 (ADDU 8,11, 12):执行无符号整数加法，将寄存器11和12的值相加，结果存储到寄存器8中。

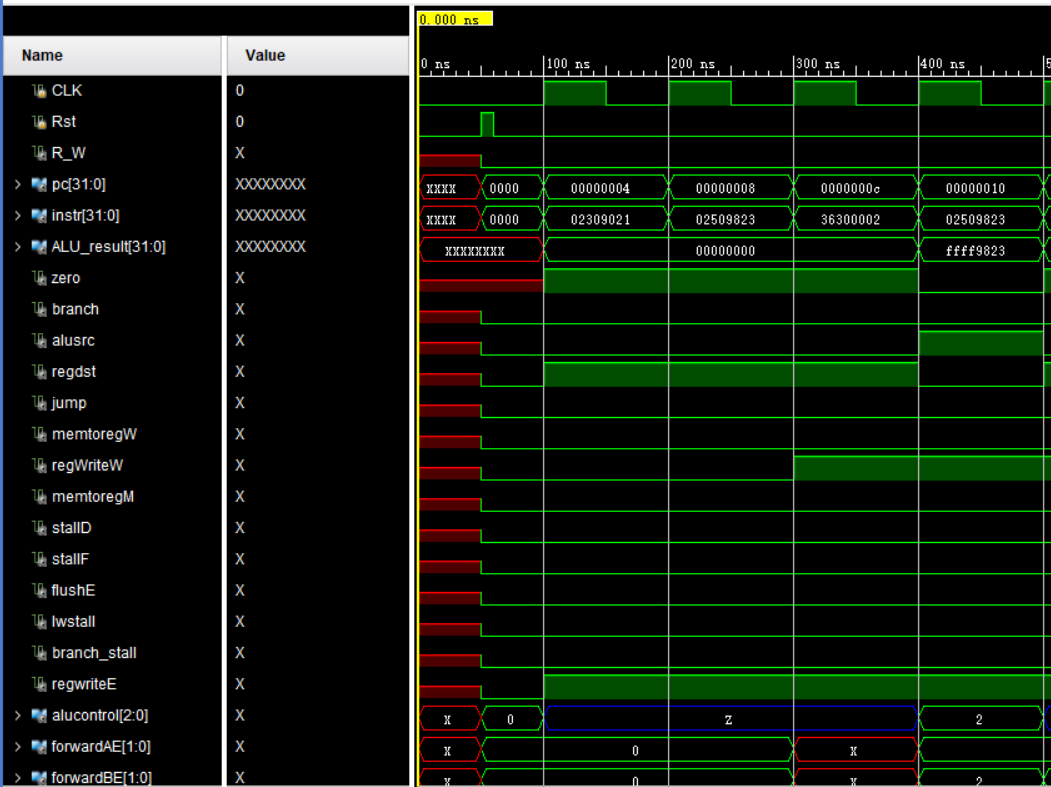
3. 02509823 (SUBU 16,12, 9):执行无符号整数减法，将寄存器12的值减去9的值，结果存储到寄存器16中。

4. 36300002 (ORI 16,16, 2): 执行按位或运算，将寄存器16的值和立即数2进行按位或操作，并将结果存储回16寄存器中。

5. 00138880 (SLL 7,4,2): 执行逻辑左移操作，将寄存器4的值左移两位，并将结果存储到寄存器4的值左移两位，并将结果存储到寄存器7中。

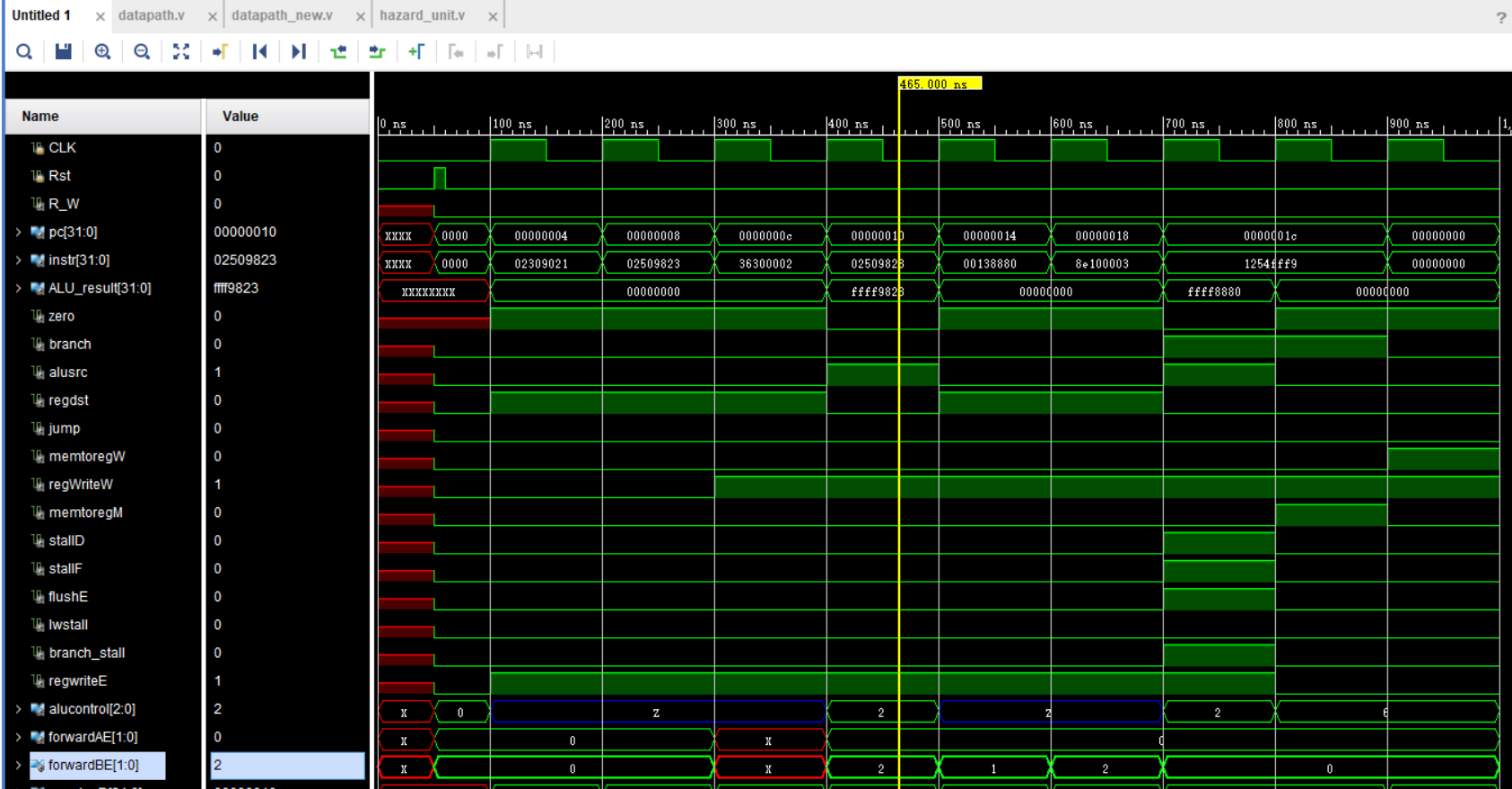
6. 8e100003 (LW 16,3(8)): 从内存中加载一个字（32位数据）到寄存器16中，地址为寄存器16中，地址为寄存器8中的值加上偏移量3。

7. 1254fff9 (BEQ 10，20, -7): 分支相等指令，如果10和20中的值相等，则跳转到当前指令的前7条指令处执行。

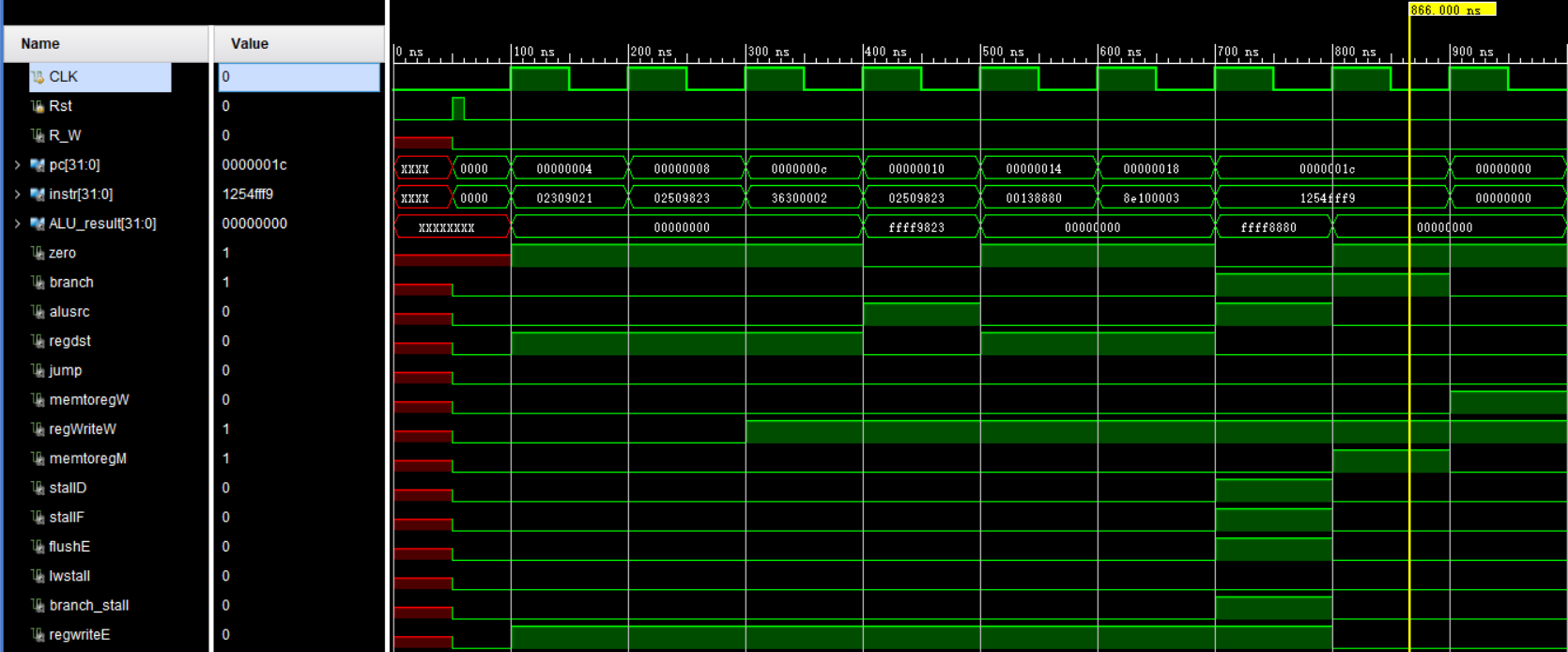


以(ADDU 8,11, 12)指令为例：将寄存器11和12的值相加，结果存储到寄存器8中。对应十六进制指令为02309021。

其指令运行过程为：首先进行取指令，pc对应地址在指令存储器中取出指令；接着交由Controller进行译码，产生控制信号。由于addu为R型指令，其十六进制指令中op部分对应为000000，产生信号为regdst置为1，regdst则是控制mux使另一个操作数类型为寄存器。各个控制信号控制从RegFile中读出数据后，然后将数据输入ALU进行运算，最后并将结果写回目标寄存器。由于两个寄存器的值均初始化为0，所以计算结果也为0。

1. 数据冒险

当ADDU指令执行到执行阶段计算出ALU\_result的值时，再次执行SUBU指令,两个指令的值都要用到寄存器12的值，因寄存器12的值尚未回写，需进行数据前推，置前推信号forwardBE为2，完成计算结果前推。

1. 控制冒险

当执行到跳转指令时将分支指令的判断提前至译码阶段，此时需设置取指、译码阶段暂停信号stallD，stallF为1，跳转信号branch为1。实现跳转，由仿真图可知，跳转到了开始位置。从最初位置开始重新取指令。

**五、调试和心得体会**

在实际设计过程中，发现自己对书上这部分理论内容还是不太熟悉，最开始只是一头雾水照线路图连线，在实际的调整错误中才逐步熟练了五级流水线cpu指令的全过程，针对五级流水线CPU中出现的数据冒险和控制冒险的问题，通过流水线前推和暂退解决，理解了指令执行的各个阶段的结构。