中山大学数据科学与计算机学院本科生实验报告

（2019 学年秋季学期）

课程名称：**计算机组成原理实验**　　任课教师：郭雪梅　　助教：汪庭葳、刘洋旗

年级&班级	2018 级计科一班	专业(方向)	计算机类
学号	18340013	姓名	陈琮昊
电话	15734867907	Email	1062080021@qq.com
开始日期	2019.12.18	完成日期	2019.12.20

一、实验题目：实验 8 流水线 CPU 的设计

二、实验目的：

(1) 了解流水线 CPU 基本功能部件的设计与实现方法，

(2) 了解提高 CPU 性能的方法。

(3) 掌握流水线 MIPS 微处理器的工作原理。

(4) 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。

(5) 掌握流水线 MIPS 微处理器的测试方法。

三、实验内容

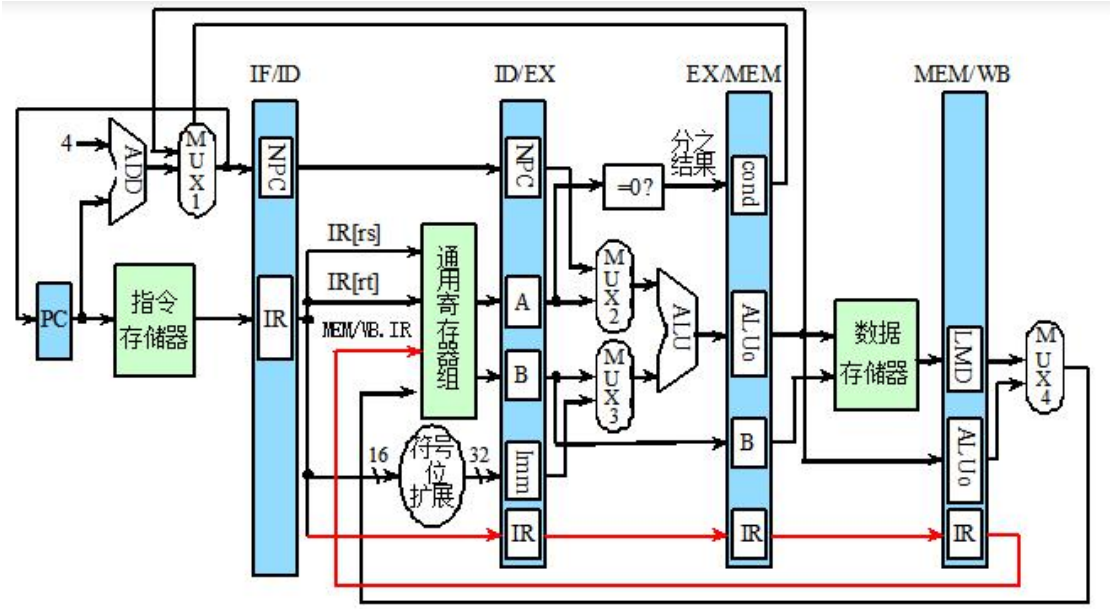
1. 实验步骤：

（1）分模块实现，给出Verilog代码；

（2）运行仿真，检测结果是否正确。

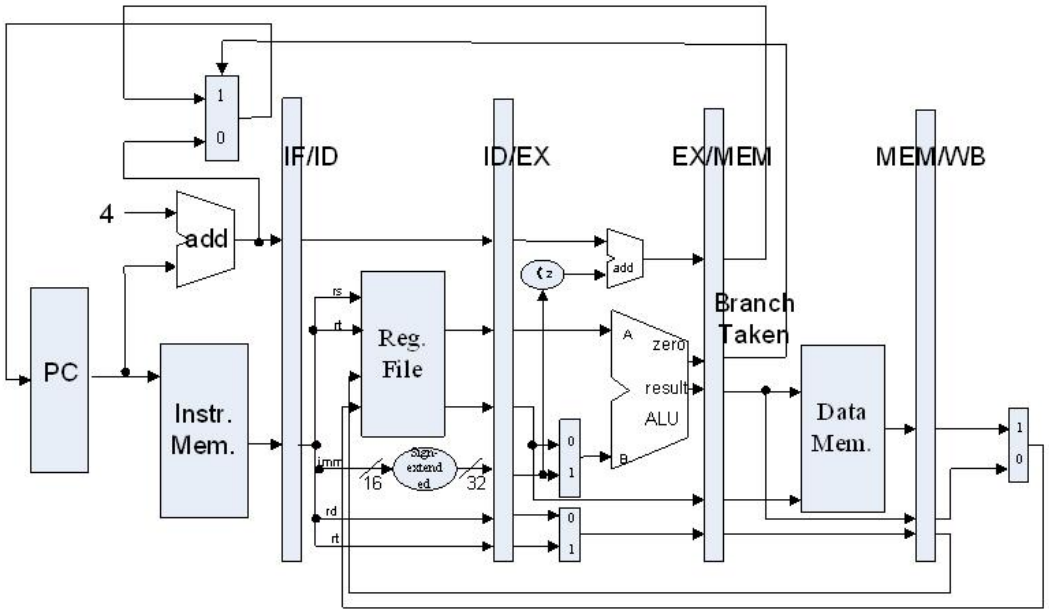
2. 实验原理：流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于现代 CPU 的架构中。根据 MIPS 处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级，对应多周期的五个处理阶段。一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

流水线实现的数据通路如下图：

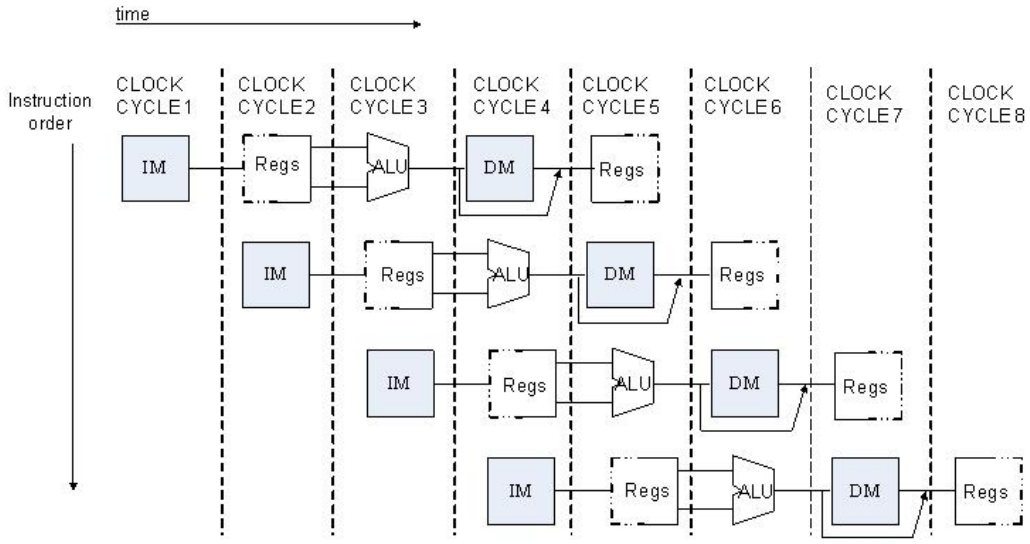


流水实现的数据通路

流水线设计如下图（**没有冒险处理**）：



指令执行过程参考如下图：



本实验涉及的相关指令如下表：

表 1 本实验所涉及的 11 条 MIPS 指令

R 型指令							
指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	功能
Add	000000	rs	rt	rd	000000	100000	寄存器加
Sub	000000	rs	rt	rd	000000	100010	寄存器减
And	000000	rs	rt	rd	000000	100100	寄存器与
Or	000000	rs	rt	rd	000000	100101	寄存器或
Xor	000000	rs	rt	rd	000000	100110	寄存器异或
I 型指令							
Addi	001000	rs	rt	immediate			立即数加
Andi	001100	rs	rt	immediate			立即数与
Ori	001101	rs	rt	immediate			立即数或
Lw	100011	rs	rt	offset			取数据
Sw	101011	rs	rt	offset			存数据
Beq	000100	rs	rt	offset			相等转移
Bne	000101	rs	rt	offset			不等转移

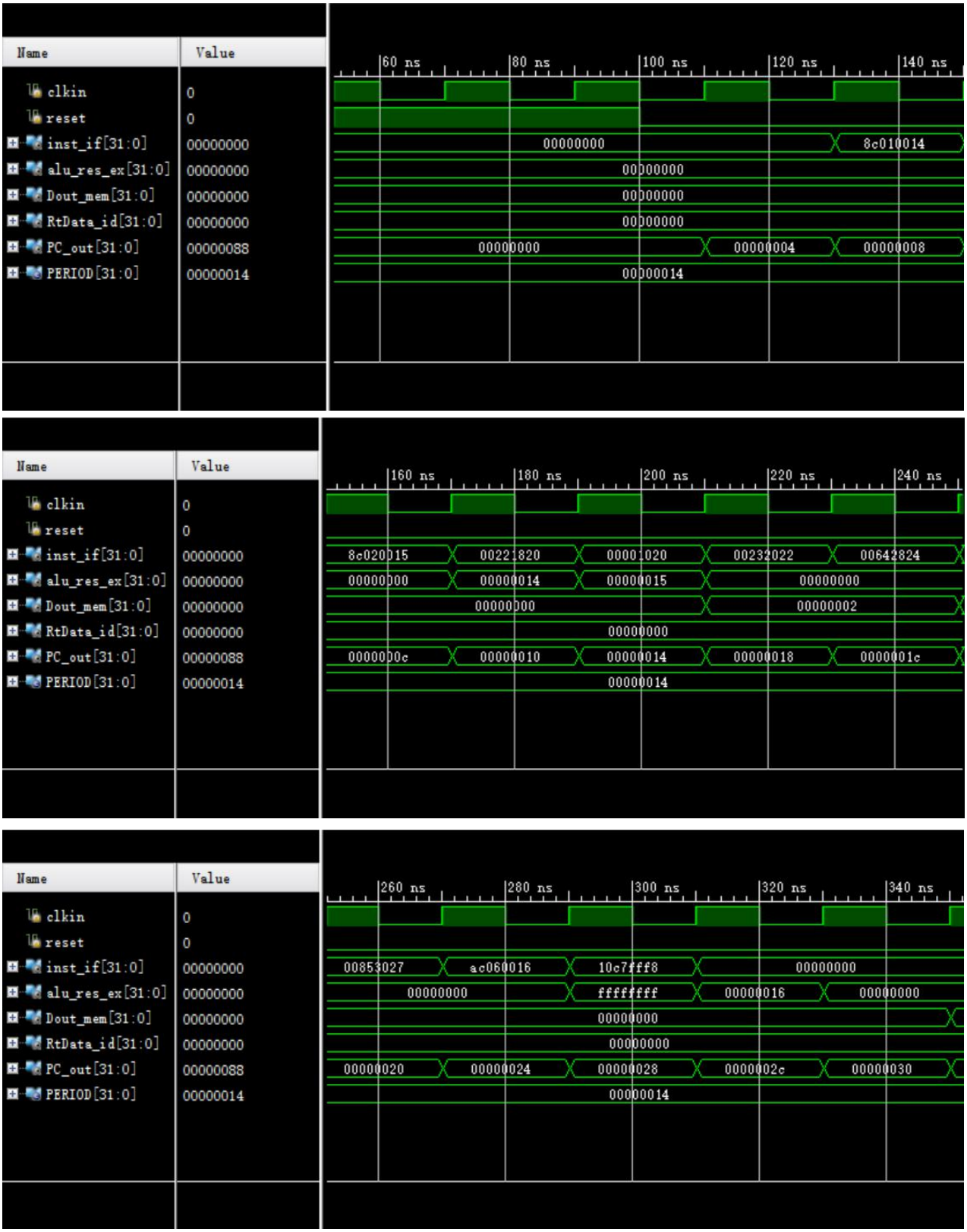
四、实验结果：

首先给出该流水线 CPU 实现的指令如下：

1.lw r1,$20(r0) 机器码：8c010014
2.lw r6,$21(r1) 机器码：8c020015
3.add r3,r0,r0 机器码：00221820
4.dd r2,r0,r0 机器码：00001020
5.sub r4,r1,r3 机器码：00232022
6.and r5,r4,r5 机器码：00642824
7.nor r6,r4,r5 机器码：00853027
8.sw r6,$22(r0) 机器码：ac060016
9.beq r6,r7,-8 机器码：10c7fff8

仿真结果：

下面给出仿真结果的截图。



首先在第一张图中可以看到经过一段时间后第一条指令才开始执行，这与后面所给代码是相吻合的；然后第一条指令 lw r1,$20(r0)和第二条指令 lw r6,$21(r1)正常执行：可以在第二张图中的 alu_res_ex 看到那里有 0x14 和 0x15 可知结果正确；由于在地址 0x15 存入了 1 个数字 2，所以可以看到在第二条指令的 WB 阶段的 Dout_mem 为 2 可知指令正确执行；然后到了第 3 条指令 add r3,r1,r2 就会出问题，因为发生了 Data Hazard，此时 r2 还没有更新，所以此时 r3

的结果仍然是 0+0=0，这与仿真结果是吻合的；同理后面几条也会出现相同的 Data Hazard，因此结果均为 0；一直到 nor r6,r4,r5 这条指令：因为 r4=0,r5=0，所以 r6=0xffffffff，从第三张图中可以看到该数出现在了该指令执行的 EX 阶段，可知指令正确执行。然后后面两条继续执行，直到结束。

五、实验感想：本学期最后一次实验主要是设计流水线 CPU，如果在理论课的时候搞清楚了流水线 CPU 的整个过程，那么根据老师给的代码实现这一设计并不难。但是需要注意的是有一些地方需要进行小的修改，而这些小的修改考察的就是细心和耐心程度，还有就是对整个代码要有一个正确的理解，才能够找到那些不易发现的 bug。一开始未进行改动时跑出来的仿真是有一些问题的：比如有的输出结果读不出来，还有就是执行的时候并不是按照正确的流水线 CPU 的原理来执行的：比如指令的某个阶段出现在了错误的周期。后来在进行了相应的改动以后成功读出了数据，每条指令的各个阶段也都是在正确的位置执行。

附录：

流水线 CPU 的实现代码：

(1) coe 文件：

(i)ROM_test.coe:

```
memory_initialization_radix=16;

memory_initialization_vector=

8c010014,

8c020015,

00221820,

00001020,

00232022,

00642824,

00853027,

ac060016,

10c7fff8;
```

(ii)test.coe:

```
memory_initialization_radix=16;
```

```
memory_initialization_vector=

00000000, 00000000, 00000000, 00000000, 00000000, 00000002,

00000002, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000001, 00000004, 00000000, 00000000;
```

(2)仿真文件 sim.v：

```
module sim;

// Inputs

reg clkin;

reg reset;

wire[31:0] inst_if;//指令,送给顶层的 data2

wire[31:0] alu_res_ex;//ALU 结果送给 data4

wire[31:0] Dout_mem;//memory 输出送给 data6,就是图里的 Data_in

wire[31:0] RtData_id;//寄存器堆的输出 B，送给 data5，就是图里的 Data_out

wire[31:0] PC_out;//pc，送给 data7


MipsPipelineCPU top(

.clk(clkin),

.reset(reset),

.inst_if(inst_if),

.alu_res_ex(alu_res_ex),

.Dout_mem(Dout_mem),

.RtData_id(RtData_id),

.PC_out(PC_out)
```

```
);


initial begin

// Initialize Inputs

clkin = 0;

reset = 1;

// Wait 100 ns for global reset to finish

#100;

reset = 0;

end

parameter PERIOD = 20;

always begin

clkin = 1'b0;

#(PERIOD / 2) clkin = 1'b1;

#(PERIOD / 2) ;

end

endmodule
```

(3)顶层文件 top.v:

```
module MipsPipelineCPU(clk,reset,inst_if,

alu_res_ex,Dout_mem,

RtData_id,PC_out

);

//CPU 模块输入：clk、reset
```

```
//CPU 模块输出：PC 地址、指令、ALU 运算结果、寄存器堆的数据输出 B、Memory 结果
//这些数据都是一开始产生就传递给输出
input clk;//100Mhz
input reset;
output[31:0] inst_if;//指令,送给顶层的 data2
output[31:0] alu_res_ex;//ALU 结果送给 data4
output[31:0] Dout_mem;//memory 输出送给 data6,就是图里的 Data_in
output[31:0] RtData_id;//寄存器堆的输出 B，送给 data5，就是图里的 Data_out
output[31:0] PC_out;//pc，送给 data7



/*IF 级*/
wire branch_or_pc_mem;//本来是 MEM 级的!
wire[31:0] Branch_addr_mem;//本来是 MEM 级的!
wire[31:0] next_pc_if;
wire[31:0] inst_if;
IF IF(
//输入
.clk(clk),
.reset(reset),
.branch_or_pc(branch_or_pc_mem),//需要 MEM 的输入，branch_or_pc_mem
.branch_addr(Branch_addr_mem),//需要 EX/MEM 的输入
//输出
.next_pc_if(next_pc_if),
```

```
.inst_if(inst_if),
.pc(PC_out)//当前 pc
);

/*IF-ID 寄存器*/
wire[31:0] next_pc_id;
wire[31:0] inst_id;
flipflop#(.WIDTH(32))IF_ID1(
.clk(clk),
.reset(reset),
.in(inst_if),//送指令
.out(inst_id)
);
flipflop#(.WIDTH(32))IF_ID2(
.clk(clk),
.in(next_pc_if),//送 pc+4
.reset(reset),
.out(next_pc_id)
);

//注意这里申明了 WB 级的东西：RegWrite 和 RegWriteAddr，有点混乱，写 WB 级注意不要重复！
wire[4:0] RtAddr_id,RdAddr_id;
wire RegWrite_wb,MemtoReg_id,RegWrite_id,MemWrite_id;
wire MemRead_id,ALUSrcB_id,RegDst_id,Branch_id;
```

```
wire[4:0] RegWriteAddr_wb;

wire[2:0] ALUCode_id;

wire[31:0] Imm_id,RsData_id,RtData_id;


/*ID 级*/

wire[31:0] RegWriteData_wb;//WB 级的东西，注意!

assign RegWriteData_wb=reg_data_wb;


ID ID(.clk(clk),.reset(reset),.inst_id(inst_id),

.RegWrite_wb(RegWrite_wb),.RegWriteAddr_wb(RegWriteAddr_wb),

.RegWriteData_wb(RegWriteData_wb),//送进来的数据要经过选择，在 WB 命名为 reg_data_wb!

.RegWrite_id(RegWrite_id),.RegDst_id(RegDst_id),.MemtoReg_id(MemtoReg_id),

.MemWrite_id(MemWrite_id),.MemRead_id(MemRead_id),

.ALUCode_id(ALUCode_id),.ALUSrcB_id(ALUSrcB_id),

.Branch_id(Branch_id),.Imm_id(Imm_id),.RsData_id(RsData_id),.RtData_id(RtData_id),

.RtAddr_id(RtAddr_id),.RdAddr_id(RdAddr_id));


/*ID-EX 级间寄存器*/

//总共 14 根线

wire[4:0] RtAddr_ex,RdAddr_ex;

wire MemtoReg_ex,RegWrite_ex,MemWrite_ex;

wire MemRead_ex,ALUSrcB_ex,RegDst_ex,Branch_ex;

wire[2:0] ALUCode_ex;

wire[31:0] Imm_ex,RsData_ex,RtData_ex,next_pc_ex;
```

```
flipflop#(.WIDTH(1))ID_EX1(
.clk(clk),
.reset(reset),
.in(RegWrite_id),//RegWrite
.out(RegWrite_ex)
);
flipflop#(.WIDTH(1))ID_EX2(
.clk(clk),
.reset(reset),
.in(RegDst_id),//RegDst
.out(RegDst_ex)
);
flipflop#(.WIDTH(1))ID_EX3(
.clk(clk),
.reset(reset),
.in(MemRead_id),//MemRead
.out(MemRead_ex)
);
flipflop#(.WIDTH(1))ID_EX4(
.clk(clk),
.reset(reset),
.in(MemWrite_id),//MemWrite
.out(MemWrite_ex)
);
```

```
flipflop#(.WIDTH(1))ID_EX5(
.clk(clk),
.reset(reset),
.in(ALUSrcB_id),//ALUSrcB_id
.out(ALUSrcB_ex)
);
flipflop#(.WIDTH(1))ID_EX6(
.clk(clk),
.reset(reset),
.in(MemtoReg_id),//MemtoReg
.out(MemtoReg_ex)
);
flipflop#(.WIDTH(1))ID_EX7(
.clk(clk),
.reset(reset),
.in(Branch_id),//Branch
.out(Branch_ex)
);
flipflop#(.WIDTH(3))ID_EX8(//注意这里的宽度是 3!
.clk(clk),
.reset(reset),
.in(ALUCode_id),//ALUCode
.out(ALUCode_ex)
);
```

```
flipflop#(.WIDTH(32))ID_EX9(//注意是 32 位!
.clk(clk),
.reset(reset),
.in(next_pc_id),//pc+4
.out(next_pc_ex)
);
flipflop#(.WIDTH(32))ID_EX10(
.clk(clk),
.reset(reset),
.in(RsData_id),//寄存器堆 A
.out(RsData_ex)
);
flipflop#(.WIDTH(32))ID_EX11(
.clk(clk),
.reset(reset),
.in(RtData_id),//寄存器堆 B
.out(RtData_ex)
);
flipflop#(.WIDTH(32))ID_EX12(
.clk(clk),
.reset(reset),
.in(Imm_id),//Imm,符号拓展
.out(Imm_ex)
);
```

```
flipflop#(.WIDTH(5))ID_EX13(//注意宽度是 5!
.clk(clk),
.reset(reset),
.in(RtAddr_id),//rt
.out(RtAddr_ex)
);
flipflop#(.WIDTH(5))ID_EX14(
.clk(clk),
.reset(reset),
.in(RdAddr_id),//rd
.out(RdAddr_ex)
);

/*EX 级*/
wire[31:0] Branch_addr_ex;
wire[31:0] alu_res_ex;
wire alu_zero_ex;
wire[4:0] RegWriteAddr_ex;
EX EX(.clk(clk),.next_pc_ex(next_pc_ex),
.ALUCode_ex(ALUCode_ex),.ALUSrcB_ex(ALUSrcB_ex),
.RegDst_ex(RegDst_ex),
.Imm_ex(Imm_ex),.RsData_ex(RsData_ex),.RtData_ex(RtData_ex),
.RtAddr_ex(RtAddr_ex),.RdAddr_ex(RdAddr_ex),
//输出
```

```
.Branch_addr_ex(Branch_addr_ex),
.alu_zero_ex(alu_zero_ex),.alu_res_ex(alu_res_ex),
.RegWriteAddr_ex(RegWriteAddr_ex)
);

/*EX-MEM 级间寄存器*/
wire RegWrite_mem;
wire MemRead_mem;
wire MemWrite_mem;
wire MemtoReg_mem;
wire[31:0] alu_res_mem;
wire alu_zero_mem;
wire[31:0] RtData_mem;
wire[4:0] RegWriteAddr_mem;
flipflop#(.WIDTH(1))EX_MEM1(
.clk(clk),
.reset(reset),
.in(RegWrite_ex),//RegWrite
.out(RegWrite_mem)
);
flipflop#(.WIDTH(1))EX_MEM2(
.clk(clk),
.reset(reset),
.in(MemRead_ex),//MemRead
```

```
.out(MemRead_mem)
);
flipflop#(.WIDTH(1))EX_MEM3(
.clk(clk),
.reset(reset),
.in(MemWrite_ex),//MemWrite
.out(MemWrite_mem)
);
flipflop#(.WIDTH(1))EX_MEM4(
.clk(clk),
.reset(reset),
.in(MemtoReg_ex),//MemtoReg
.out(MemtoReg_mem)
);
flipflop#(.WIDTH(1))EX_MEM5(
.clk(clk),
.reset(reset),
.in(Branch_ex),//Branch
.out(Branch_mem)
);
flipflop#(.WIDTH(32))EX_MEM6(//注意是 32 位!
.clk(clk),
.reset(reset),
.in(Branch_addr_ex),//Branch 地址
```

```
.out(Branch_addr_mem)//注意这里送回 IF 级!
);
flipflop#(.WIDTH(32))EX_MEM7(
.clk(clk),
.reset(reset),
.in(alu_res_ex),//alu 结果
.out(alu_res_mem)
);
flipflop#(.WIDTH(1))EX_MEM8(
.clk(clk),
.reset(reset),
.in(alu_zero_ex),//alu 的零信号
.out(alu_zero_mem)
);
flipflop#(.WIDTH(32))EX_MEM9(
.clk(clk),
.reset(reset),
.in(RtData_ex),//RtData
.out(RtData_mem)
);
flipflop#(.WIDTH(5))EX_MEM10(
.clk(clk),
.reset(reset),
.in(RegWriteAddr_ex),//写回地址
```

```
.out(RegWriteAddr_mem)
);

/*MEM 级*/
wire[31:0] Dout_mem;

MEM MEM(
.clk(clk),.MemRead_mem(MemRead_mem),.MemWrite_mem(MemWrite_mem),
.Branch_mem(Branch_mem),
.alu_zero_mem(alu_zero_mem),
.alu_res_mem(alu_res_ex),.RtData_mem(RtData_mem),
.branch_or_pc_mem(branch_or_pc_mem),.Dout_mem(Dout_mem)//注意信号要往回送，给 IF
);

/*MEM-WB 级间寄存器*/
wire[31:0] Dout_wb;
wire[31:0] alu_res_wb;

wire MemtoReg_wb;
flipflop#(.WIDTH(1))MEM_WB1(
.clk(clk),
.reset(reset),
.in(RegWrite_mem),//RegWrite
.out(RegWrite_wb)
```

```
);
flipflop#(.WIDTH(1))MEM_WB2(
.clk(clk),
.reset(reset),
.in(MemtoReg_mem),//MemtoReg
.out(MemtoReg_wb)
);
flipflop#(.WIDTH(32))MEM_WB3(//注意这里是 32 位
.clk(clk),
.reset(reset),
.in(Dout_mem),//Dout，RAM 的输出
.out(Dout_wb)
);
flipflop#(.WIDTH(32))MEM_WB4(
.clk(clk),
.reset(reset),
.in(alu_res_mem),//alu 的结果
.out(alu_res_wb)
);
flipflop#(.WIDTH(5))MEM_WB5(//注意是 5 位
.clk(clk),
.reset(reset),
.in(RegWriteAddr_mem),//RegWriteAddr
.out(RegWriteAddr_wb)
```

```
);

/*WB 级*/
reg[31:0] reg_data_wb;

always@(*)begin
case(MemtoReg_wb)
1'b0:reg_data_wb<=alu_res_wb;//来自 ALU
1'b1:reg_data_wb<=Dout_wb;//来自 RAM
endcase
end

endmodule
```

后面的代码是顶层模块中调用的一些子模块代码的实现：

(4) IF:

```
module IF(clk,reset,branch_or_pc,
branch_addr,next_pc_if,inst_if,
pc
);
input clk;
input reset;
input branch_or_pc;//Branch&ALU_zero
input[31:0]  branch_addr;//Branch 跳转地址
output[31:0]  next_pc_if;//pc+4
```

```
output[31:0]  inst_if;//从 ROM 中读的指令

output[31:0]  pc;


//PC 的多选器

reg[31:0]  pc_in;//pc 选择

always@(*)

begin

case(branch_or_pc)

1'b0:pc_in<=next_pc_if;//没有分支也没有 jump

1'b1:pc_in<=branch_addr;//有 Branch

endcase

end




//PC 寄存器

reg[31:0] pc;

always@(posedge clk)

begin

if(reset) pc<=32'b0;//复位

else pc<=pc_in;

end


//计算下一个 PC 的加法器

adder_if adder32_bits_if(
```

```
.a(pc),
.b(32'b00000000000000000000000000000100),
.c(next_pc_if)
);

//指令 ROM
//InstructionROM InstructionROM(.a(pc[11:2]),.spo(inst_if));
InstructionROM InstructionROM (
  .clka(clk),    // input wire clka
  .ena(1'b1),      // input wire ena
  .addra(pc[11:2]),  // input wire [9 : 0] addra
  .douta(inst_if)  // output wire [31 : 0] douta
);

endmodule
```

（5）adder_if:

```
module adder_if(a,b, c);
input [31:0] a,b;//偏移量
output [31:0] c;//新指令地址
assign c=a+b;
endmodule
```

(6)IF-ID flipflop:

```
module flipflop(clk,reset,in,out);
parameter WIDTH=1;//根据需要改宽度
```

```
input clk;//

input reset;

input [WIDTH-1:0]in;

output reg[WIDTH-1:0] out;//


always@(posedge clk)

if(reset)

out<={WIDTH{1'b0}};

else

out<=in;

endmodule
```

(7) ID:

```
module ID(clk,reset,inst_id,

RegWrite_wb,RegWriteAddr_wb,RegWriteData_wb,

RegDst_id,MemtoReg_id,RegWrite_id,

MemWrite_id,MemRead_id,ALUCode_id,

ALUSrcB_id,Branch_id,

Imm_id,RsData_id,RtData_id,

RtAddr_id,RdAddr_id

);

input clk;

input reset;

input[31:0] inst_id;//IF 给的指令
```

```
//WB 级的输入
input RegWrite_wb;
input[4:0] RegWriteAddr_wb;
input[31:0] RegWriteData_wb;

//八个信号输出
output RegWrite_id;
output RegDst_id;
output MemRead_id;
output MemWrite_id;
output ALUSrcB_id;
output Branch_id;
output MemtoReg_id;
output[2:0] ALUCode_id;

//其他输出
output[31:0] Imm_id;//符号拓展
output[31:0] RsData_id;//寄存器堆输出 1
output[31:0] RtData_id;//寄存器堆输出 2
output[4:0] RtAddr_id;//rt
output[4:0] RdAddr_id;//rd

assign RtAddr_id=inst_id[20:16];//rt
assign RdAddr_id=inst_id[15:11];//rd
```

```
assign Imm_id={{16{inst_id[15]}},inst_id[15:0]};//符号扩展成 32 位立即数


/*控制模块*/
CtrlUnit CtrlUnit(
//输入
.inst(inst_id),
//输出
.RegWrite(RegWrite_id),.RegDst(RegDst_id),
.Branch(Branch_id),.MemRead(MemRead_id),.MemWrite(MemWrite_id),
.ALUCode(ALUCode_id),.ALUSrc_B(ALUSrcB_id),
.MemtoReg(MemtoReg_id)
);


/*寄存器堆模块*/
RegisterFiles RegisterFiles(
//输入，由 WB 级来提供
.clk(clk),.rst(reset),.L_S(RegWrite_wb),
.R_addr_A(inst_id[25:21]),.R_addr_B(inst_id[20:16]),
.Wt_addr(RegWriteAddr_wb),.wt_data(RegWriteData_wb),
//输出
.rdata_A(RsData_id),.rdata_B(RtData_id)
);
```

```
endmodule
```

(8)ControlUnit:

```
module CtrlUnit(inst,RegWrite,RegDst,

Branch,MemRead,

MemWrite,ALUCode,

ALUSrc_B,

MemtoReg

);

input[31:0] inst;

output RegWrite;

output RegDst;

output Branch;

output MemRead;

output MemWrite;

output[2:0] ALUCode;

output ALUSrc_B;

output MemtoReg;//1:来自 mem


wire[5:0] op;

wire[5:0] func;

wire[4:0] rt;

assign op=inst[31:26];//op 字段

assign func=inst[5:0];//func 字段
```

```
//R 指令
parameter R_type_op=6'b000000;
parameter ADD_func=6'b100000;
parameter AND_func=6'b100100;
parameter XOR_func=6'b100110;
parameter OR_func=6'b100101;
parameter NOR_func=6'b100111;
parameter SUB_func=6'b100010;

//R_type
wire ADD,AND,NOR,OR,SUB,XOR,R_type;
assign ADD=(op==R_type_op)&&(func==ADD_func);
assign AND=(op==R_type_op)&&(func==AND_func);
assign NOR=(op==R_type_op)&&(func==NOR_func);
assign OR=(op==R_type_op)&&(func==OR_func);
assign SUB=(op==R_type_op)&&(func==SUB_func);
assign XOR=(op==R_type_op)&&(func==XOR_func);
assign R_type=ADD||AND||NOR||OR||SUB||XOR;

//Branch
parameter BEQ_op=6'b000100;
parameter BNE_op=6'b000101;
wire BEQ,BNE,Branch;
assign BEQ=(op==BEQ_op);
```

```
assign BNE=(op==BNE_op);

assign Branch=BEQ||BNE;


// I_type instruction decode

parameter ADDI_op=6'b001000;

parameter ANDI_op=6'b001100;

parameter XORI_op=6'b001110;

parameter ORI_op=6'b001101;

wire ADDI,ANDI,XORI,ORI,I_type;

assign ADDI=(op== ADDI_op);

assign ANDI=(op==ANDI_op);

assign XORI=(op==XORI_op);

assign ORI=(op==ORI_op);

assign I_type=ADDI||ANDI||XORI||ORI;


// SW ,LW instruction decode

parameter SW_op=6'b101011;

parameter LW_op=6'b100011;

wire SW,LW;

assign SW=(op==SW_op);

assign LW=(op==LW_op);


// Control Singal

assign RegWrite=LW||R_type||I_type;//要写寄存器
```

```
assign RegDst=R_type;//RegDst=1，选择 rd，只有 R 指令这样

assign MemWrite=SW;

assign MemRead=LW;

assign MemtoReg=LW;

assign ALUSrc_B=LW||SW||I_type;


// ALUCode

//自己定义的，只要能在 ALU 里对应的上就行

parameter alu_add=3'b010;

parameter alu_sub=3'b110;

parameter alu_and=3'b000;

parameter alu_or=3'b001;

parameter alu_xor=3'b011;

parameter alu_nor=3'b100;


reg[2:0] ALUCode;

always@(*)begin

if(op==R_type_op)begin

case(func)

ADD_func: ALUCode<=alu_add;

AND_func: ALUCode<=alu_and;

XOR_func: ALUCode<=alu_xor;

OR_func: ALUCode<=alu_or;

NOR_func: ALUCode<=alu_nor;
```

```
SUB_func: ALUCode<=alu_sub;
default: ALUCode<=alu_add;
endcase
end
else
begin
case(op)
BEQ_op: ALUCode<=alu_sub;
BNE_op: ALUCode<=alu_sub;
ADDI_op: ALUCode<=alu_add;
ANDI_op: ALUCode<=alu_and;
XORI_op: ALUCode<=alu_xor;
ORI_op: ALUCode<=alu_or;
SW_op: ALUCode<=alu_add;
LW_op: ALUCode<=alu_add;
default: ALUCode<=alu_add;
endcase
end
end

endmodule
```

(9)RegisterFiles:

```
module RegisterFiles(
input clk, rst, L_S,
```

```
input[4:0] R_addr_A, R_addr_B, Wt_addr,
input[31:0] wt_data,
output[31:0] rdata_A, rdata_B
);
reg[31:0] register [1:31];
integer i;
assign rdata_A=(R_addr_A==0)?0: register[R_addr_A];
assign rdata_B=(R_addr_B==0)?0: register[R_addr_B];

always@(posedge clk or posedge rst)begin
if(rst==1)
for(i=1; i<32; i= i+1)
register[i]<=0;
else if((Wt_addr!=0)&&(L_S==1))
register[Wt_addr]<= wt_data;
end

endmodule
```

(10)ID-EX flipflop:

```
module flipflop(clk,reset,in,out);
parameter WIDTH=1;//根据需要改宽度
input clk;//
input reset;
input [WIDTH-1:0]in;
```

```
output reg[WIDTH-1:0] out;//


always@(posedge clk)
if(reset)
out<={WIDTH{1'b0}};
else
out<=in;
endmodule
```

(11)EX:

```
module EX(clk,next_pc_ex,
ALUCode_ex,ALUSrcB_ex,
RegDst_ex,
Imm_ex,RsData_ex,RtData_ex,
RtAddr_ex,RdAddr_ex,
//输出
Branch_addr_ex,
alu_zero_ex,alu_res_ex,RegWriteAddr_ex
);
input clk;
input[31:0] next_pc_ex;
input[2:0] ALUCode_ex;
input ALUSrcB_ex;
input RegDst_ex;
input[31:0] Imm_ex;
```

```
input[31:0] RsData_ex;

input[31:0] RtData_ex;

input[4:0] RtAddr_ex;

input[4:0] RdAddr_ex;

//

output[31:0] Branch_addr_ex;

output alu_zero_ex;

output[31:0] alu_res_ex;

output reg[4:0] RegWriteAddr_ex;


//分支地址

adder_32bits adder_32bits_ex(.a(next_pc_ex),.b(Imm_ex<<2),.c(Branch_addr_ex));


//ALUSrcB 的多选器

reg[31:0] alu_in;

always@(*)begin

case(ALUSrcB_ex)

1'b0:alu_in<=RtData_ex;//来自寄存器堆第二个输出

1'b1:alu_in<=Imm_ex;//来自符号扩展

endcase

end


//ALU

ALU ALU(.input1(RsData_ex),.input2(alu_in),.aluCtr(ALUCode_ex),
```

```
.aluRes(alu_res_ex),.zero(alu_zero_ex)//overflow什么也不连
);

//写寄存器堆地址的多选器
always@(*)begin
case(RegDst_ex)
1'b0:RegWriteAddr_ex<=RtAddr_ex;//rt
1'b1:RegWriteAddr_ex<=RdAddr_ex;//rd
endcase
end

endmodule
```

(12)adder_32bits.v:

```
module adder_32bits(
input[31:0] a,
input[31:0] b,
output[31:0] c
);

assign c= a+ b;
endmodule
```

(13)ALU:

```
module ALU(
input [31:0] input1,
```

```
input [31:0] input2,
input [2:0] aluCtr,
output reg[31:0] aluRes,
output reg zero
);
always @(input1 or input2 or aluCtr) // 运算数或控制码变化时操作
begin
    case(aluCtr)
        3'b110: // 减
        begin
            aluRes = input1 - input2;
            if(aluRes == 0)
                zero = 1;
            else
                zero = 0;
        end
        3'b010: // 加
            aluRes = input1 + input2;
        3'b000: // 与
            aluRes = input1 & input2;
        3'b001: // 或
            aluRes = input1 | input2;
        3'b011: // 异或
            aluRes = input1 ^ input2;
```

```
        3'b100: // 或非
            aluRes =  ~ ( input1 | input2 );
//        4'b0111: // 小于设置
//            begin
//                if(input1<input2)
//                    aluRes = 1;
//                else
//                    aluRes = 0;
//            end
//        4'b1000: // lui
//            aluRes = input2 << 16;
//        4'b1001: // subu
//                    begin
//                        aluRes = input1 - input2;
//                        if(aluRes == 0)
//                            zero = 1;
//                        else
//                            zero = 0;
//                    end
//        4'b0011: // addu
//            aluRes = input1 + input2;
//        4'b1010: //sltu
//            begin
//                if(input1 < input2)
```

```
//                  aluRes = 1;
//               else
//                  aluRes = 0;
//           end
//       default:
//           aluRes = 0;
    endcase
end
endmodule
```

(14)EX-MEM flipflop:

```
module flipflop(clk,reset,in,out);
parameter WIDTH=1;//根据需要改宽度
input clk;//
input reset;
input [WIDTH-1:0]in;
output reg[WIDTH-1:0] out;//

always@(posedge clk)
if(reset)
out<={WIDTH{1'b0}};
else
out<=in;
endmodule
```

(15)MEM:

```
module MEM(clk,MemRead_mem,
MemWrite_mem,
Branch_mem,
alu_zero_mem,
alu_res_mem,
RtData_mem,
branch_or_pc_mem,
Dout_mem
);
input clk;
//MemRead 信号暂时不要了
input MemRead_mem;
input MemWrite_mem;
input Branch_mem;
input alu_zero_mem;
input[31:0]alu_res_mem;
input[31:0] RtData_mem;
output branch_or_pc_mem;
output[31:0] Dout_mem;

DataRAM DataRAM(
.clka(clk),//input clka
.ena(1'b1),// input wire ena
.wea(~MemRead_mem&MemWrite_mem),//input [0:0] wea
```

```
.addra(alu_res_mem[11:2]),//input [9 : 0] addra
.dina(RtData_mem),//input [31:0] dina
.douta(Dout_mem)//output [31:0] douta
);

//and 模块，确定跳转信号
and_1bit and_1bit(.a(Branch_mem),.b(alu_zero_mem),.c(branch_or_pc_mem));

endmodule
```

(16)and_1bit:

```
module and_1bit(
    input a,
    input b,
    output c
    );

    assign c = a & b;
endmodule
```

(17)MEM-WB flipflop:

```
module flipflop(clk,reset,in,out);
parameter WIDTH=1;//根据需要改宽度
input clk;//
input reset;
input [WIDTH-1:0]in;
```

```
output reg[WIDTH-1:0] out;//


always@(posedge clk)

if(reset)

out<={WIDTH{1'b0}};

else

out<=in;

endmodule
```