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：丁文、汪庭葳

年级&班级	2019 级 04 班	专业(方向)	计算机科学与技术（超算方向）
学号	19335112	姓名	李钰
电话	19847352856	Email	1643589912@qq.com
开始日期	2021. 1.1	完成日期	2021.1.8

一、实验题目

流水线 CPU 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。

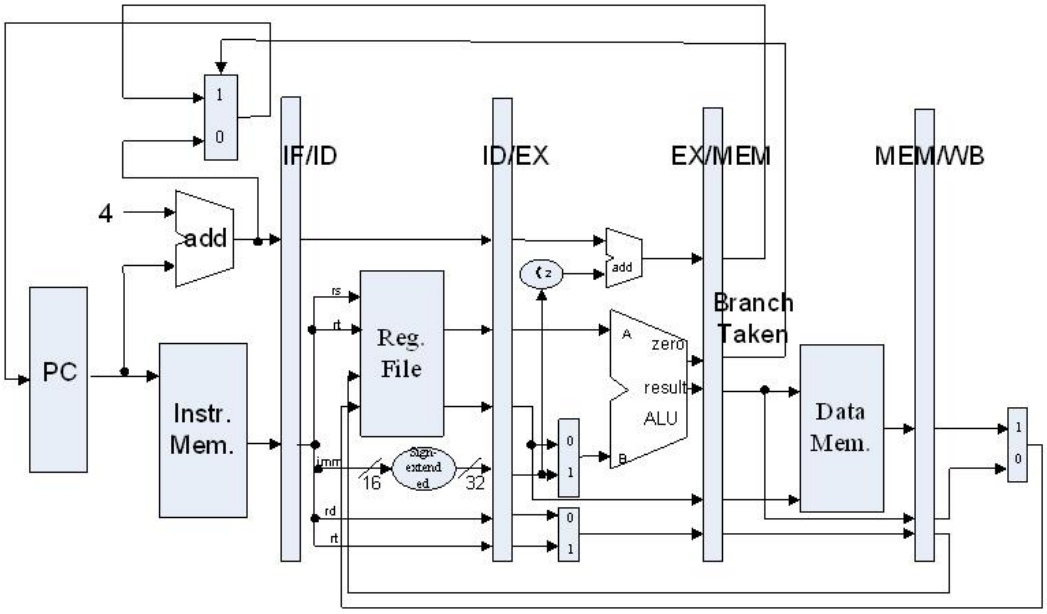
三、实验内容

1. 实验步骤

在原有单周期CPU基础上，增加4个级间寄存器，分别是IF/ID、ID/EX、EX/MEM、MEM/WB模块，用于存放前一阶段操作之后的指令数据以及相应控制信号，向后传递

2. 实验原理

流水线CPU相较于单周期CPU主要增加的是级间寄存器。其结构图如下



（1） IF级：取指令部分。

包括指令储存器和PC寄存器及其更新模块，负责根据PC寄存器的值从指令存储器中取出指令编码和对PC的值进行更新。

（2） ID级：指令译码部分。

根据独处的指令编码形成控制信号和读寄存器堆输出的寄存器的值。

（3） EX级：执行部分。

根据指令的编码进行算数或者逻辑运算或者计算条件分支指令的跳转目标地址。此外LW、SW指令所用的RAM访问地址也是在本级上实现。控制信号有ALUCode、ALUSrcA、ALUScrB和RegDst，根据这些信号确定ALU操作、选择两个ALU操作数A、B，并确定目标寄存器。

（4） MEM级：存储器访问部分。

只有在执行LW、SW指令时才对存储器进行读写，对其他指令只起到一个周期的作用。该级只需存储器写操作允许信号MemWrite。

（5） WB级：寄存器堆写回部分。

该级把指令执行的结果回写到寄存器文件中。该级设置寄存器写操作允许信号RegWrite。

四、实验结果

本次实现了最简单的几条基本操作指令，add，sub，lw等，可以看出有数据冲突时，经过流水线的计算结果是不正确的。

五、实验感想

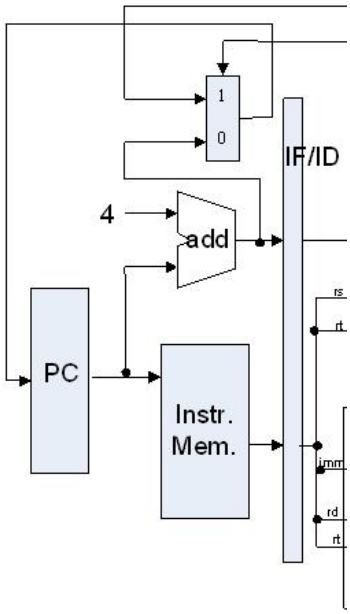
本次实验没有实现冒险冲突的改善，待课后学有余力时再进行完善。通过这次实验我体会到了流水线与单周期CPU的不同之处，更加深刻地掌握了级间寄存器的用处。

附录（流程图，注释过的代码）：

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

```
reg[WIDTH-10] out;
always@(posedeg clk)
if(reset)
out<={WIDTH{1'b0}};
else
out<=in;
endmodule
```

1.取指令部分（IF）



IF/ID.IR←Mem[PC]（将 PC 的指令取出放入指令寄存器表示当前运行的指令）

NPC←PC+4（PC 指向下一条指令（32 位加 4））

PC 模块功能

实现思路

由于 PC 是 32 位，所以增加一个 32 位加法器，固定与 32 位的立即数 4 进行相加，PC+4 结果在时钟信号的上升沿更新写进 PC 寄存器。

主要实现代码

1.PCAdd4

```
module adder_if(a,b, c);
input [31:0] a,b;//偏移量
output [31:0] PCadd4;//新指令地址
assign c=a+b;
endmodule
```

2.指令存储器

INSTMEM 参照单周期 CPU

MUX

REG_ifid

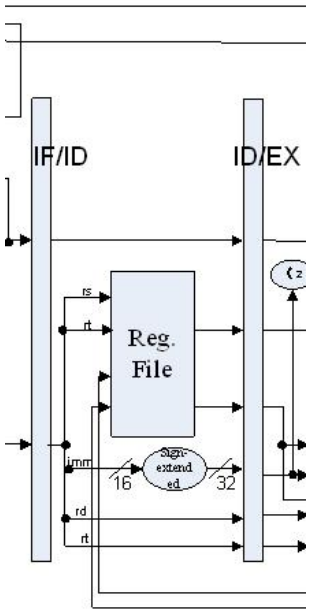
参考代码

```
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));

endmodule
```

其中 next_pc_if(NPC) 和 inst_if(IR) 需要送到下个阶段

2) Design ID



ID means "instruction fetch". This module includes the CPU controller, which is responsible for decoding, and register files. The codes of MEM are as follow:

```
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
```

The following shows the design details for control unit:

```
  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
elsebegin
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
```

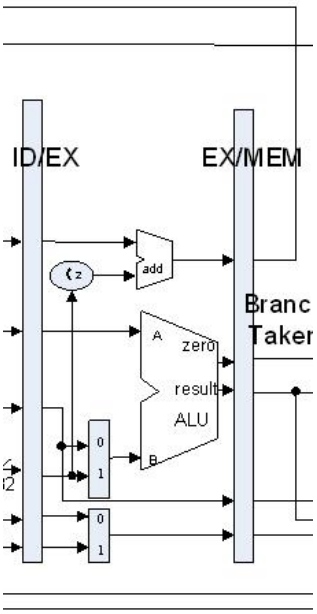
The details for register files:

```
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 orposedge rst)begin
if(rst==1)
```

```
for(i=1; i<32; i= i+1)
register[i]<=0;
elseif((Wt_addr!=0)&&(L_S==1))
register[Wt_addr]<= wt_data;
end

endmodule
```

The register files are the same as the previous (used inComputer Organization course)

3) Design EX



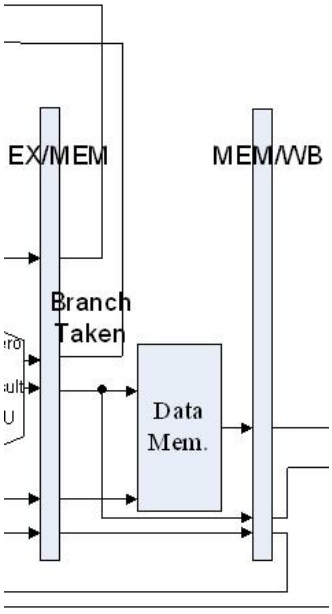
EX means "execution". It contains ALU and an adder.

The codes for EX module are shown below:

```
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;
outputreg[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(.ALU_operation(ALUCode_ex),.A(RsData_ex),.B(alu_in),
.res(alu_res_ex),.zero(alu_zero_ex),.overflow()//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



4) Design 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
.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
```

Also an adder should be included, which is serving for calculating the proper address of writing back.

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

assign c= a+ b;
endmodule
```

5) Design WB

The stage is simple and can be done in top:

```
/*WB 级*/
//只有一个多选器，直接在顶层实现
//选择写回的内容
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
```

6) Design top module

Finally we come to the top module, this module is actually for connecting lines between registers and different stages, which is shown as below:

```
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_mem),.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
```