《计算机组成原理》实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| **年级、专业、班级** |  | **姓名** | 周梓蕙 |
| **实验题目** | 实验四简单五级流水线CPU | | |
| **实验时间** | 2021 年 6 月 20 日 | **实验地点** | ds1410 |
| **实验成绩** | 优秀/良好/中等 | **实验性质** | □验证性  □G设计性  □综合性 |
| **教师评价：**  □算法/实验过程正确; □源程序/实验内容提交; □程序结构/实验步骤合理;  □实验结果正确; □语法、语义正确; □报告规范;  其他:  评价教师: | | | |
| **实验目的**   1. 掌握流水线 (Pipelined) 处理器的思想。 2. 掌握单周期处理中执行阶段的划分。 3. 了解流水线处理器遇到的冒险。 4. 掌握数据前推、流水线暂停等冒险解决方式。 | | | |

报告完成时间: 2021 年 7 月 1 日

# 实验内容

阅读实验原理实现以下模块：

1. Datapath，所有模块均可由实验三复用，需根据不同阶段，修改 mux2 为 mux3(三选一选择器)，以及带有 enable(使能)、clear(清除流水线) 等信号的触发器，
2. Controller，其中 main decoder 与 alu decoder 可直接复用，另需增加触发器在不同阶段进行信号传递
3. 指令存储器 inst\_mem(Single Port Ram)，数据存储器 data\_mem(Single Port Ram)；同实验三一致，无需改动，
4. 参照实验原理，在单周期基础上加入每个阶段所需要的触发器，重新连接部分信号。实验给出top 文件，需兼容 top 文件端口设定。
5. 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

# 实验设计

## 冒险处理模块

### 功能描述

本次实验处理的冒险主要包括数据冒险和结构冒险。

### 接口定义

介绍接口的信号名、方向、位宽，描述功能。

# 实验过程记录

记录实验的过程，完成了什么样的工作，存在的问题包括哪些，解决方案如何等。subsubsec- tion 名称自行设定。

## 完成的工作

### 五级流水线的构建

实验的整个过程主要是按照实验指导和示例代码进行。按照实验中的要求，创建多个寄存器

（每个寄存器只处理一个信号）保存每一阶段的控制信号，结果。

**表 1:** 接口定义模版

|  |  |  |  |
| --- | --- | --- | --- |
| **信号名** | **方向** | **位宽** | **功能描述** |
| clk | Input | 1-bit | 时钟周期信号 |
| rst | Input | 1-bit | 清零信号，为 1 时执行初始化操作。 |
| writereg\_m | Input | 5-bit | m 阶段的控制信号，写入寄存器的编  号。 |
| writereg\_w | Input | 5-bit | w 阶段写入的寄存器的编号。 |
| regwrite\_m | Input | 1-bit | m 阶段的控制信号，为 1 表示要写入  数据到寄存器。 |
| regwrite\_w | Input | 1-bit | w 阶段的控制信号，为 1 表示要写入  数据到寄存器。 |
| regwrite\_e | Input | 1-bit | e 阶段的控制信号，为 1 表示要写入  数据到寄存器。 |
| writereg\_e | Input | 5-bit | e 阶段写入的寄存器的地址。 |
| rs\_e | Input | 5-bit | 执行阶段的源操作数寄存器的编号。 |
| rt\_e | Input | 5-bit | 执行阶段的源操作数寄存器的编号。 |
| rs\_d | Input | 5-bit | 解码阶段的源操作数寄存器的编号。 |
| rt\_d | Input | 5-bit | 解码阶段的源操作数寄存器的编号。 |
| memtoreg\_e | Input | 1-bit | 执行阶段，为 1 表示要将存储器中的  值加载到寄存器。 |
| memtoreg\_m | Input | 1-bit | 为 1 表示要将存储器中的值加载到  寄存器。 |
| branch\_d | Input | 1-bit | 解码阶段的控制信号，为 1 表示当前  beq 指令。 |
| stall\_f | Output | 1-bit | f 阶段，为 1 表示要将 f 到 d 阶段的  寄存器挂起。 |
| stall\_d | Output | 1-bit | d 阶段，为 1 表示要将 d 到 e 阶段的  寄存器挂起。 |
| flush\_e | Output | 1-bit | e 阶段，为 1 表示要将 e 到 m 阶段的  寄存器刷新。 |
| forward\_a\_d | Output | 1-bit | d 阶段，为 1 表示 d 阶段的 beq 比较  中的一个输入值采用数据前推的值。 |
| forward\_b\_d | Output | 1-bit | d 阶段，为 1 表示 d 阶段的 beq 比较  中的一个输入值采用数据前推的值。 |
| forward\_a\_e | Output | 2-bit | e 阶段，为 00 表示 alu 的一个输入值  选用rd1 的值，为 01、10 表示选用w、 m 阶段写入寄存器的值。 |
| forward\_b\_e | Output | 2-bit | e 阶段，为 00 表示 alu 的一个输3 入值选用rd2 的值，为 01、10 表示选用w、  m 阶段写入寄存器的值。 |

### 冒险的处理

本次实验处理的冒险主要包括数据冒险和结构冒险，两者均采用了数据前推和阻塞的方法处理。处理数据冒险的情况主要有执行阶段 alu 的输入未保存到寄存器中等，此时正确的数据可能在 m 或 w 阶段。正确的数据可能来自于 alu 的计算结果或存储器。处理结构冒险的情况主要有beq 判断的两个寄存器的值被修改且未写回存储器，此时的数据可能在 e 或m 阶段。

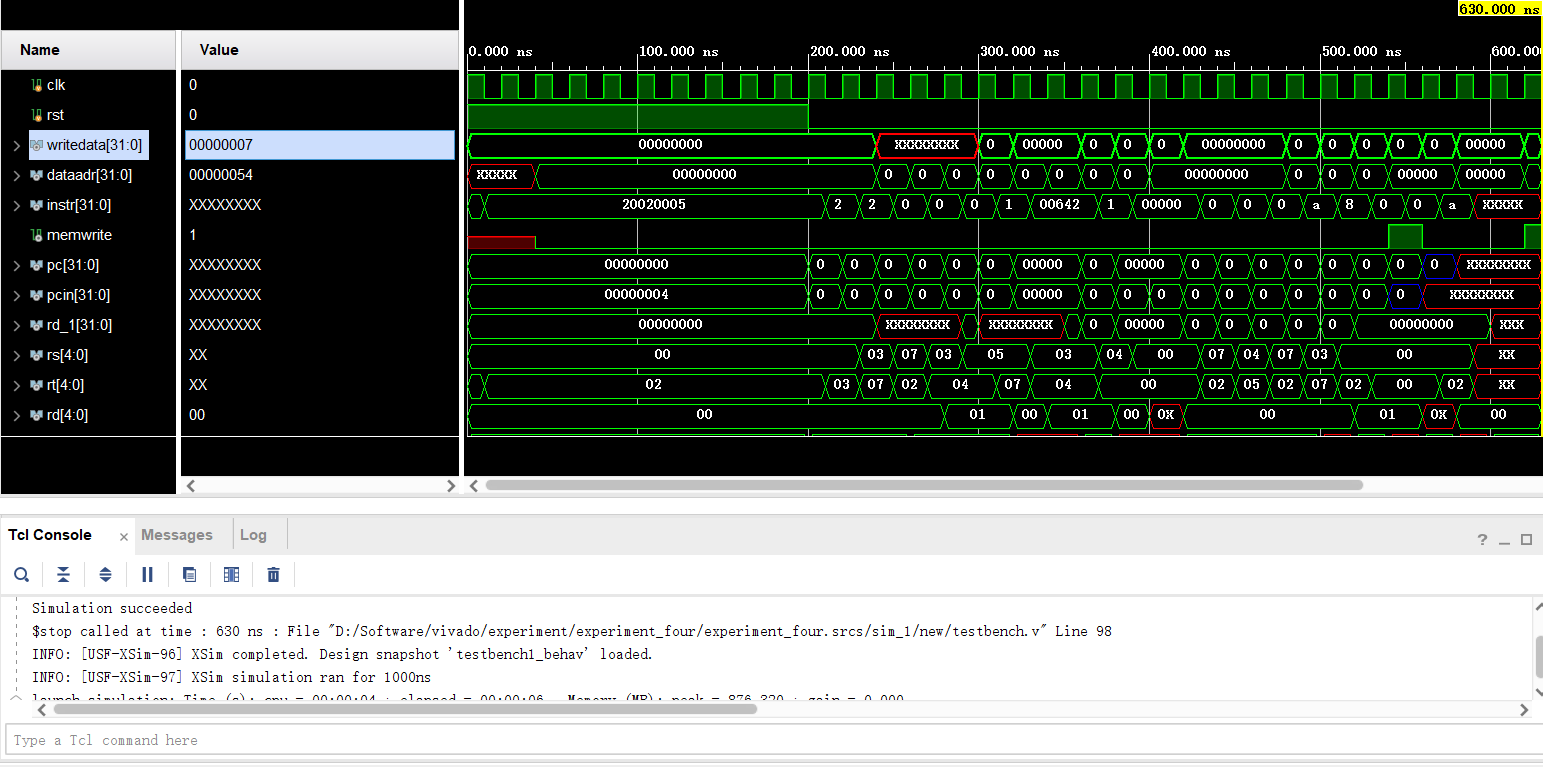
## 存在的问题和解决方案

### 调试时pcsrc\_d 慢了一个周期产生

pcsrc\_d 比 instr\_d 慢了一个周期产生，导致 j 指令无法及时跳转。经检查发现，送入 controller模块的是 instr\_d 而非 instr，因此应该将解析 instr\_d 后的控制信号直接输出，而不是保存一个周期再输出。

# 实验结果及分析

需要仿真图一张，控制台打印输出图一张，要求仿真图中包含 pc、instr、rs、rt、rd、result 信号，仿真图应在控制台打印输出Simulation succeeded 时截图。控制台打印输出图为此时截图。



**图 1**

其中，result 则为仿真图里的"dataadr"，表示 alu 的计算结果。

# Datapath 代码

module datapath (

clk , rst , memtoreg\_d , pcsrc\_d , alusrc\_d ,

regdst\_d , regwrite\_d , jump\_d , alucontrol\_d , overflow , zero\_m , pc , instr , instr\_d , aluout\_m , writedata\_m , readdata\_m , pcin , writereg\_e ,

rs\_e , rt\_e , writereg\_m , writereg\_w , regwrite\_m , regwrite\_w ,// 数 据 冒 险

forward\_a\_e , forward\_b\_e , forward\_a\_d , forward\_b\_d , rs\_d , rt\_d , memtoreg\_e ,

stall\_f , stall\_d , flush\_e , equal\_d ,

memtoreg\_m ,

regwrite\_e , rd1\_d , src\_b , src\_a\_e , src\_b\_e , rd1\_e , alusrc\_e , sign\_imm\_e , wd\_3 , aluout\_w , equal\_rd1 , equal\_rd2 , pc\_branch\_d , pc\_plus\_d , pc\_sl , memtoreg\_w , readdata\_w , pc\_jump\_ 1

);

parameter WIDTH = 32;

input clk , rst , memtoreg\_d , pcsrc\_d , alusrc\_d , regdst\_d , regwrite\_d , jump\_d

, overflow ;

input[ WIDTH - 1 : 0] instr;

input[ WIDTH - 1 : 0] readdata\_m ; input [2 : 0] alucontrol\_d ; output zero\_m ;

output[ WIDTH - 1 : 0] instr\_d , aluout\_m ; output[ WIDTH - 1 : 0] writedata\_m ; output[ WIDTH - 1 : 0] pc , pcin ;

output equal\_d ;

output[ WIDTH - 1 : 0] rd1\_d , src\_b , rd1\_e ; output [4 : 0] writereg\_e ;

output alusrc\_e ;

output [31 : 0] sign\_imm\_e ;

output [31 : 0] wd\_3 , aluout\_w , equal\_rd1 , equal\_rd2 , pc\_branch\_d , pc\_plus\_d

, pc\_sl , readdata\_w ; output memtoreg\_w ;

// hazard

input forward\_a\_d , forward\_b\_d ; input [1 : 0] forward\_a\_e , forward\_b\_e ;

output regwrite\_m , regwrite\_w , memtoreg\_m , regwrite\_e ; output [4 : 0] rs\_e , rt\_e , writereg\_m , writereg\_w ;

input stall\_f , stall\_d , flush\_e ; output memtoreg\_e ;

output [4 : 0] rs\_d , rt\_d ;

// output [31 : 0] instr\_d ;

output [31 : 0] src\_a\_e , src\_b\_e , pc\_jump\_ 1 ;

wire [ WIDTH - 1 : 0] pc\_plus\_f , pc\_branch\_1 , pc\_branch\_2 , pc\_jump\_1 , pcin , pc\_sl , src\_b , src\_a\_e , src\_b\_e ;

wire [27 : 0] pc\_jump ;

wire [ WIDTH - 1 : 0] equal\_rd1 , equal\_rd 2 ;

wire [ WIDTH - 1 : 0] rd1\_d , rd2\_d ;

wire [ WIDTH - 1 : 0] instr\_d , sign\_imm\_d , pc\_plus\_d ;

wire [4 : 0] rd\_d ;

wire alusrc\_e , regwrite\_e , memtoreg\_e , regdst\_e , zero\_e ; wire [2 : 0] alucontrol\_e ;

wire [4 : 0] writereg\_e , rs\_e , rt\_e , rd\_e ;

wire [ WIDTH - 1 : 0] rd1\_e , rd2\_e , writedata\_e , pc\_branch\_e , sign\_imm\_e , pc\_plus\_e , aluout\_e ;

wire memtoreg\_m , regwrite\_m ; wire [4 : 0] writereg\_m ; wire [ WIDTH - 1 : 0] pc\_branch\_m ;

wire memtoreg\_w , regwrite\_w ; wire [4 : 0] writereg\_w ;

wire [ WIDTH - 1 : 0] aluout\_w , readdata\_w , wd\_3 ;

// stage f

mux2 mux\_branch (. a( pc\_plus\_f), . b( pc\_branch\_d ), . select( pcsrc\_d ), . result( pc\_branch\_ 2 )); // f

mux2 mux\_jump (. a( pc\_branch\_ 2 ), . b( pc\_jump\_ 1 ), . select( jump\_d ), . result( pcin

)); // f

// PC p(. clk( clk), . ena( enable ), . pc( pc), . pcin ( pcin ), . rst( rst)); // f adder add\_pc\_plus (. a( pc), . b(32 ’ b100 ), . result( pc\_plus\_f)); // f

sl2 #(26) sl\_pc\_jump (. a( instr\_d [25 : 0]), . result( pc\_jump )); // f

PC p(. clk( clk), . ena( stall\_f), . pc( pc), . pcin ( pcin ), . rst( rst)); // f

floper #( WIDTH ) instr\_d\_f( clk , rst , stall\_d , instr , instr\_d , pcsrc\_d ); floper #( WIDTH ) pc\_plus\_d\_f( clk , rst , stall\_f , pc\_plus\_f , pc\_plus\_d ,

pcsrc\_d );

// stage d

signext sign\_ext (. a( instr\_d [15 : 0]), . result( sign\_imm\_d )); // d

regfile reg\_file (. clk (~ clk), . we3 ( regwrite\_w ), . ra1 ( instr\_d [25 : 21]) , . ra2 ( instr\_d [20 : 16]) , . wa3 ( writereg\_w ), . wd3 ( wd\_3 ), . rd1 ( rd1\_d ), . rd2 ( rd2\_d )); // d

// regfile reg\_file (. clk (~ clk), . we3 ( regwrite\_w ), . ra1 ( instr\_d [25 : 21]) , . ra2 ( instr\_d [20 : 16]) , . wa3 ( writereg\_w ), . wd3 ( wd\_3 ), . rd1 ( rd1\_d ), . rd2 ( rd2\_d )); // d

flopr #(1) regwrite\_e\_f ( clk , flush\_e , regwrite\_d , regwrite\_e ); flopr #(1) memtoreg\_e\_f ( clk , flush\_e , memtoreg\_d , memtoreg\_e );

// flopr #(1) memwrite\_e\_f ( clk , rst , memwrite\_d , memwrite\_e );

flopr #(3) alucontrol\_e\_f ( clk , flush\_e , alucontrol\_d , alucontrol\_e ); flopr #(1) alusrc\_e\_f( clk , flush\_e , alusrc\_d , alusrc\_e );

flopr #(1) regdst\_e\_f( clk , flush\_e , regdst\_d , regdst\_e ); flopr #( WIDTH ) rd 1 \_e\_f( clk , flush\_e , rd1\_d , rd1\_e ); flopr #( WIDTH ) rd 2 \_e\_f( clk , flush\_e , rd2\_d , rd2\_e ); flopr #(5) rs\_e\_f( clk , flush\_e , rs\_d , rs\_e );

flopr #(5) rt\_e\_f( clk , flush\_e , rt\_d , rt\_e );

flopr #(5) rd\_e\_f( clk , flush\_e , rd\_d , rd\_e );

flopr #( WIDTH ) sign\_imm\_e\_f ( clk , flush\_e , sign\_imm\_d , sign\_imm\_e );

// flopr #( WIDTH ) pc\_plus\_e\_f( clk , flush\_e , pc\_plus\_d , pc\_plus\_e );

mux2 #(32) mux2 \_equal\_rd 1 (. a( rd1\_d ), . b( aluout\_m ), . select( forward\_a\_d ), . result( equal\_rd 1 ));

mux2 #(32) mux2 \_equal\_rd 2 (. a( rd2\_d ), . b( aluout\_m ), . select( forward\_b\_d ), . result( equal\_rd 2 ));

// stage e

sl2 sl\_pc\_plus (. a( sign\_imm\_d ), . result( pc\_sl)); // e

adder add\_pc\_branch (. a( pc\_sl), . b( pc\_plus\_d ), . result( pc\_branch\_d )); // e mux2 #(5) mux\_wa (. a( rt\_e ), . b( rd\_e ), . select( regdst\_e ), . result( writereg\_e )

); // e

mux2 #(32) mux\_src\_b (. a( src\_b ), . b( sign\_imm\_e ), . select( alusrc\_e ), . result( src\_b\_e )); // e

mux\_3 #(32) mux3 \_src\_a\_e (. a( aluout\_m ), . b( wd\_3 ), . c( rd1\_e ), . select( forward\_a\_e ), . result( src\_a\_e ));

mux\_3 #(32) mux3 \_src\_b\_e (. a( aluout\_m ), . b( wd\_3 ), . c( rd2\_e ), . select( forward\_b\_e ), . result( src\_b ));

alu a(. a( src\_a\_e ), . b( src\_b\_e ), . clk (~ clk), . reset( rst), . op( alucontrol\_e ),

. result( aluout\_e ), . zero ( zero\_e )); // e

flopr #(1) regwrite\_m\_f ( clk , rst , regwrite\_e , regwrite\_m ); flopr #(1) memtoreg\_m\_f ( clk , rst , memtoreg\_e , memtoreg\_m ); flopr #(1) zero\_m\_f( clk , rst , zero\_e , zero\_m );

flopr #(5) writereg\_m\_f ( clk , rst , writereg\_e , writereg\_m ); flopr #( WIDTH ) aluout\_m\_f( clk , 1 ’ b0 , aluout\_e , aluout\_m );

flopr #( WIDTH ) writedata\_m\_f ( clk , rst , writedata\_e , writedata\_m );

// flopr #( WIDTH ) pc\_branch\_m\_f ( clk , rst , pc\_branch\_e , pc\_branch\_m );

// satge m

flopr #(1) regwrite\_w\_f ( clk , rst , regwrite\_m , regwrite\_w ); flopr #(5) writereg\_w\_f ( clk , rst , writereg\_m , writereg\_w ); flopr #( WIDTH ) aluout\_w\_f( clk , rst , aluout\_m , aluout\_w );

// flopr #( WIDTH ) wrdata\_m\_f( clk , rst , readdata\_e , writedata\_m ); flopr #( WIDTH ) readdate\_w\_f ( clk , rst , readdata\_m , readdata\_w ); flopr #(1) memtoreg\_m\_w ( clk , rst , memtoreg\_m , memtoreg\_w );

mux2 #(32) mux\_result (. a( aluout\_w ), . b( readdata\_w ), . select( memtoreg\_w ), . result( wd\_3 )); // w

assign pc\_jump\_ 1 = { pc\_plus\_d [31 : 28], pc\_jump }; assign rs\_d = instr\_d [25 : 21];

assign rt\_d = instr\_d [20 : 16];

assign equal\_d = equal\_rd 1 == equal\_rd 2 ? 1 : 0; assign writedata\_e = src\_b ;

// assign

assign ena = ~ stall\_d ;

assign rd\_d = instr\_d [15 : 11]; endmodule

# Hazard 代码

module hazard (

clk , rst , rs\_e , rt\_e , writereg\_m , writereg\_w , regwrite\_m , regwrite\_w , forward\_a\_e , forward\_b\_e ,

rs\_d , rt\_d , memtoreg\_e , stall\_f , stall\_d , flush\_e ,

branch\_d , memtoreg\_m , writereg\_e , regwrite\_e , forward\_a\_d , forward\_b\_d , jump\_d ,

// 调 试

stall\_lw , jump ,

// 新 增

// memtoreg\_w

);

input clk , rst;

input [4 : 0] rs\_e , rt\_e , writereg\_m , writereg\_w , rs\_d , rt\_d , writereg\_e ; input regwrite\_m , regwrite\_w , memtoreg\_e , branch\_d , memtoreg\_m , regwrite\_e ; output stall\_f , stall\_d , flush\_e , forward\_a\_d , forward\_b\_d ;

output [1 : 0] forward\_a\_e , forward\_b\_e ;

// 调 试

output stall\_lw ;

// 改

input jump , jump\_d ;

// input memtoreg\_w ;

reg [1 : 0] forward\_a\_e , forward\_b\_e ;

reg stall\_f , stall\_d , flush\_e , stall\_lw , forward\_a\_d , forward\_b\_d , stall\_branch , stall\_jump ;

always@ (\*) begin

if( rst) begin

forward\_a\_e = 2 ’ b00 ;

// forward\_b\_e = 2 ’ b00 ; forward\_a\_d = 1 ’ b0 ; forward\_b\_d = 1 ’ b0 ; stall\_lw = 1 ’ b0 ; stall\_f = 1 ’ b0 ;

end else

stall\_d = 1 ’ b0 ; flush\_e = 1 ’ b0 ;

begin

// forward\_a\_e

if( rs\_e Ị= 0 && rs\_e == writereg\_m && regwrite\_m ) begin

forward\_a\_e = 2 ’ b10 ;

end

else if( rs\_e Ị= 0 && rs\_e == writereg\_w && regwrite\_w ) begin

forward\_a\_e = 2 ’ b01 ;

end else begin

forward\_a\_e = 2 ’ b00 ;

end

// forward\_b\_e

if( rt\_e Ị= 0 && rt\_e == writereg\_m && regwrite\_m ) begin

forward\_b\_e = 2 ’ b10 ;

end

else if( rt\_e Ị= 0 && rt\_e == writereg\_w && regwrite\_w ) begin

forward\_b\_e = 2 ’ b01 ;

end else begin

forward\_b\_e = 2 ’ b00 ;

end

end

stall\_lw = ((( rs\_d == rt\_e ) || ( rt\_d == rt\_e )) && memtoreg\_e ); //

如 果e 阶 段 的lw 指 令 中 写 入 的 寄 存 器 被 访 问 到， 则 需 要 挂 起 forward\_a\_d = ( rs\_d Ị= 0) && ( rs\_d == writereg\_m ) && regwrite\_m ; forward\_b\_d = ( rt\_d Ị= 0) && ( rt\_d == writereg\_m ) && regwrite\_m ; stall\_branch = ( branch\_d && regwrite\_e && ( writereg\_e == rs\_d ||

writereg\_e == rt\_d )) || ( branch\_d && memtoreg\_m && ( writereg\_m

== rs\_d || writereg\_m == rt\_d )); stall\_jump = jump ; // 改

stall\_f = stall\_lw || stall\_branch ;

// stall\_d = stall\_lw || stall\_branch || stall\_jump && Ịjump\_d ; stall\_d = stall\_lw || stall\_branch ;

flush\_e = stall\_lw || stall\_branch ; end // else

endmodule

# Controller 代码

module controller(

op , funct , zero , memtoReg , memwrite , pcsrc\_m , alusrc , regdst , regwrite , jump , alucontrol , aluop , code , rst , pcsrc\_d , equal\_d , branch\_d , clk

);

input zero , rst , equal\_d ; input clk;

input [5 : 0] op; input [5 : 0] funct; output regwrite ; output regdst; output alusrc; output memwrite ; output memto Reg ;

output pcsrc\_m , pcsrc\_d ; output jump ;

output branch\_d ;

output [2 : 0] alucontrol; output [1 : 0] aluop ; output [8 : 0] code ;

wire [2 : 0] alucontrol; wire [1 : 0] aluop ;

wire [8 : 0] code ;

maindec md (. op( op), . regwrite ( regwrite ), . regdst( regdst), . alusrc( alusrc),

. memwrite ( memwrite ), . memto Reg ( memto Reg ), . aluop ( aluop ), . zero ( zero ), . pcsrc\_m ( pcsrc\_m ), . jump ( jump ), . code ( code ), . pcsrc\_d ( pcsrc\_d ), . equal\_d ( equal\_d ), . branch\_d ( branch\_d ), . clk( clk));

aludec ad (. funct( funct), . aluop ( aluop ), . alucontrol( alucontrol), . rst(1 ’ b0 )

);

endmodule

module maindec(

op , regwrite , regdst , alusrc , memwrite , memtoReg , aluop , zero , pcsrc\_m , jump , code , rst , pcsrc\_d , equal\_d , branch\_d , clk

);

input [5 : 0] op;

input rst , clk; input zero ; input equal\_d ;

output pcsrc\_d , pcsrc\_m ; output jump ;

output regwrite ; output regdst; output alusrc; output memwrite ; output memto Reg ; output branch\_d ; output [1 : 0] aluop ; output [8 : 0] code ;

reg [8 : 0] code ; wire branch\_d ;

wire branch\_e ; wire branch\_m ; wire memwrite\_e ; wire memwrite\_m ; wire mem\_write ; wire jump\_d ;

flopr #(1) branch\_e\_f( clk , rst , branch\_d , branch\_e ); flopr #(1) branch\_m\_f( clk , rst , branch\_e , branch\_m );

// flopr #(1) memwrite\_e\_f ( clk , rst , memWrite , memwrite\_m ); flopr #(1) memwrite\_e\_f ( clk , rst , mem\_write , memwrite\_e ); flopr #(1) memwrite\_m\_f ( clk , rst , memwrite\_e , memwrite\_m ); flopr #(1) jump\_d\_f( clk , rst , jump , jump\_d );

assign regwrite = code [7]; // 是 否 需 要 回 写 寄 存 器 堆

assign regdst = code [6]; // 写 入 寄 存 器 堆 的 地 址 是 rt 还 是 rd ,0 为 rt ,1 为 rd assign alusrc = code [5]; // 送 入ALU B 端 口 的 值 是 立 即 数 32 位 拓 展 还 是 寄 存 器 堆 里

读 出 的 值

assign branch\_d = code [4]; // 是 否 为 branch 指 令， 且 满 足 branch 的 条 件

assign mem\_write = code [3]; // 是 否 需 要 写 数 据 存 储 器

assign memto Reg = code [2]; // 回 写 的 数 据 来 自 于 ALU 计 算 的 结 果/ 存 储 器 读 取 的数 据

assign aluop [1] = code [1]; assign aluop [0] = code [0];

// 新 增

assign pcsrc\_m = branch\_m & zero ; // 下 一 个 PC 值 是 PC +4/ 跳 转 的 新 地 址 assign pcsrc\_d = branch\_d & equal\_d ; // 下 一 个 PC 值 是 PC +4/ 跳 转 的 新 地 址 assign jump = code [8]; // 是 否 为 jump 指 令

assign memwrite = memwrite\_m ;

always@ (\*) begin

end

case ( op)

6 ’ b 000000 : code = 9 ’ b 011000010 ;

6 ’ b 100011 : code = 9 ’ b 010100100 ;

// 6 ’ b 101011 : code = 9 ’ b 00 x101 x00 ; 6 ’ b 101011 : code = 9 ’ b 00 x101000 ;

// 6 ’ b 000100 : code = 9 ’ b 00 x010 x01 ; 6 ’ b 000100 : code = 9 ’ b 00 x010001 ;

6 ’ b 001000 : code = 9 ’ b 010100000 ;

// 6 ’ b 000010 : code = 9 ’ b 10 xx00 xxx; 6 ’ b 000010 : code = 9 ’ b 10 xx000 xx; default: code = 9 ’ b 000000000 ;

endcase

endmodule

module aludec(

funct , aluop , alucontrol , rst

);

input rst;

input [5 : 0] funct; input [1 : 0] aluop ;

output [2 : 0] alucontrol; reg [2 : 0] alucontrol;

always@ (\*) begin

if( rst) begin

alucontrol = 3 ’ b000 ;

end else begin

if( aluop == 2 ’ b00 ) begin

alucontrol = 3 ’ b010 ; end // if( aluop == 2 ’ b00 ) else if( aluop == 2 ’ b01 ) begin

alucontrol = 3 ’ b110 ;

end // else if( aluop == 2 ’ b01 ) else if( aluop == 2 ’ b10 )

begin

case ( funct)

6 ’ b 100000 : alucontrol = 3 ’ b010 ; 6 ’ b 100010 : alucontrol = 3 ’ b110 ;

6 ’ b 100100 : alucontrol = 3 ’ b000 ; 6 ’ b 100101 : alucontrol = 3 ’ b001 ; 6 ’ b 101010 : alucontrol = 3 ’ b111 ; default: alucontrol = 3 ’ b000 ;

endcase

end // else if( aluop == 2 ’ b10 ) else

begin

alucontrol = 3 ’ b000 ; end // else

end

end // always endmodule