计算机科学系

计算机组成原理

项目综合实训报告

项目名称：五级流水 RISC 指令集 CPU 设计与实现

姓名 1：李琦　　学号：17408070220

姓名 2：王雕　　学号：17408070118

指导教师：曹非、张昭昭

时间：2019 年 12 月 23 日 ~~ 2020 年 1 月 10 日

	姓名	承担主要任务	工作量比例
项目成员	李琦	指令集的设计、五级流水图和系统结构图的设计、模块的设计、最终在实验箱上的验证	5
	王雕	指令集的设计、五级流水图和主控制器和 ALU 控制器结构图设计、位选和段选信号的设计	5

一、 设计与实训内容及完成情况

第一周：①设计 RISC 指令集。②根据所设计指令集，设计 5 级流水处理器结构。包括数据通路部分和控制器部分。

完成情况：基本要求全部实现，实现了 9 条基本指令外加 IN、OUT 两条指令；设计出了 5 级流水处理器结构（包括数据通路部分和控制器部分）以及控制器部分的电路图。

第二周：使用 verilog 硬件描述语言来描述该 CPU 结构，并用 modelsim 进行各模块仿真验证；编写机器语言测试程序，modelsim 仿真 CPU 运行测试程序，验证设计的正确性。

完成情况：要求全部实现，并且用 Verilog 语言描述出了该 CPU 结构，在 modelsim 中成功的完成了仿真并且得到了正确的波形图。

第三周：使用 Quartus II 对上述 Verilog 语言描述的 CPU 进行综合，基于可编程逻辑器件实验平台（TD-CMA 或者 CVT-SOPC-IV），构成一个简单的计算机系统。运行测试程序，证明系统功能的正确性和完备性。

完成情况：要求全部实现，在实验箱上运行测试程序准确无误，证明了系统功能的正确性和完备性。

二、 指令集的设计

助记符	指令格式						用法	含义
Bit	31-26	25-21	20-16	15-11	10-6	5-0		
R-TYPE	Op	Rs	Rt	Rd	Sa	Func		
add	000001	Rs	Rt	Rd	00000	100001	add $1,$2,$3	rd=rs+rt
sub	000001	Rs	Rt	Rd	00000	100010	sub $1,$2,$3	rd=rs-rt
or	000001	Rs	Rt	Rd	00000	100011	or $1,$2,$3	rd=rs | rt
and	000001	Rs	Rt	Rd	00000	100100	and $1,$2,$3	rd=rs & rt
slt	000001	Rs	Rt	Rd	00000	100101	slt $1,$2,$3	rd=(rs < rt) ? 1 : 0
解释域	操作码	源寄存器	源寄存器	目的寄存器	用于移位指令	ALU控制字段		

表 1 R 型指令格式表

助记符	指令格式				用法	含义
Bit	31-26	25-21	20-16	15-0		
I-TYPE	Op	Rs	Rt	immediate		
lw_m	000010	Rs	Rt	immediate	lw_m $2,10($1)	rt=memory[rs+im]
lw_io	000011	Rs	Rt	immediate	lw_io $2,10($1)	rt=in
sw_m	000111	Rs	Rt	immediate	sw_m $2,10($1)	memory[rs+im]=rt
sw_io	001011	Rs	Rt	immediate	sw_io $2,10($1)	out=data[rt]
beq	000100	Rs	Rt	immediate	beq $1,$2,10	PC=(rs==rt)?PC+4+im<<2:PC
解释域	操作码	源寄存器	源寄存器	立即数		

表 2　I 型指令格式表

助记符	指令格式		用法	含义
Bit	31-26	25-0		
J-TYPE	Op	address		
j	000101	address	j 10000	PC={(PC+4)[31,28],addr,00}
解释域	操作码	地址字段		

表 3　J 型指令格式表

助记符	指令格式	含义
Bit	31-0	
NOP	32`h00000000	无操作

表 4　空指令表

指令汇总表		
指令类型	指令	含义
控制指令	beq	条件成立(寄存器1的值与寄存器2的值相等)，跳转到 pc+4 + immediate<<2
	j	无条件跳转，跳转到 { (pc+4)[31,28] , addr , 00 }
数据传输	lw_m	从存储器中读取一个数，存入到寄存器中
	lw_io	从IN单元中读取一个数，存入到寄存器中
	sw_m	将寄存器中的值，存入到存储器中
	sw_io	将寄存器中的值，直接输出到OUT单元中
算术逻辑	add、sub	将源操作数和目的操作数相加，从源操作数中减去目的操作数，并存入寄存器中
	and、or	将源操作数和目的操作数相与(或)，并将结果存入寄存器中
	slt	比较源操作数和目的操作数大小，如果条件成立，置目的寄存器值为1，否则为0

表 5　指令汇总表

三、　数据通路和控制器的设计

ALU控制器真值表								
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	x	x	x	x	x	x	0010
1	0	1	0	0	0	0	1	0010
1	0	1	0	0	0	1	0	0110
1	0	1	0	0	0	1	1	0000
1	0	1	0	0	1	0	0	0001
1	0	1	0	0	1	0	1	0111
0	1	x	x	x	x	x	x	0110
1	1	x	x	x	x	x	x	1111

表 6　ALU 控制器真值表

ALU操作说明	
ALU 操作字段	ALU操作
0000	与
0001	或
0010	加
0110	减
0111	条件设置
1111	无操作

表 7　ALU 操作说明

主控制器真值表									
Input or Output	Signal name	R-Format	lw_m	sw_m	beq	j	lw_io	sw_io	nop
Inputs	Op5	0	0	0	0	0	0	0	0
	Op4	0	0	0	0	0	0	0	0
	Op3	0	0	0	0	0	0	1	0
	Op2	0	0	0	1	1	1	0	0
	Op1	0	1	1	0	0	1	1	0
	Op0	1	0	1	0	1	1	1	0
Outputs	ALUSrc	0	1	1	0	0	1	1	x
	RegWrite	1	1	0	0	0	1	0	0
	MemRead	0	1	0	0	0	0	0	0
	MemWrite	0	0	1	0	0	0	0	0
	Branch/PCSrc	0	0	0	1	0	0	0	0
	ALUOp1	1	0	0	0	0	0	0	1
	ALUOp0	0	0	0	1	0	0	0	1
	IN	0	0	0	0	0	1	0	0
	OUT	0	0	0	0	0	0	1	0

表 8　主控制器真值表

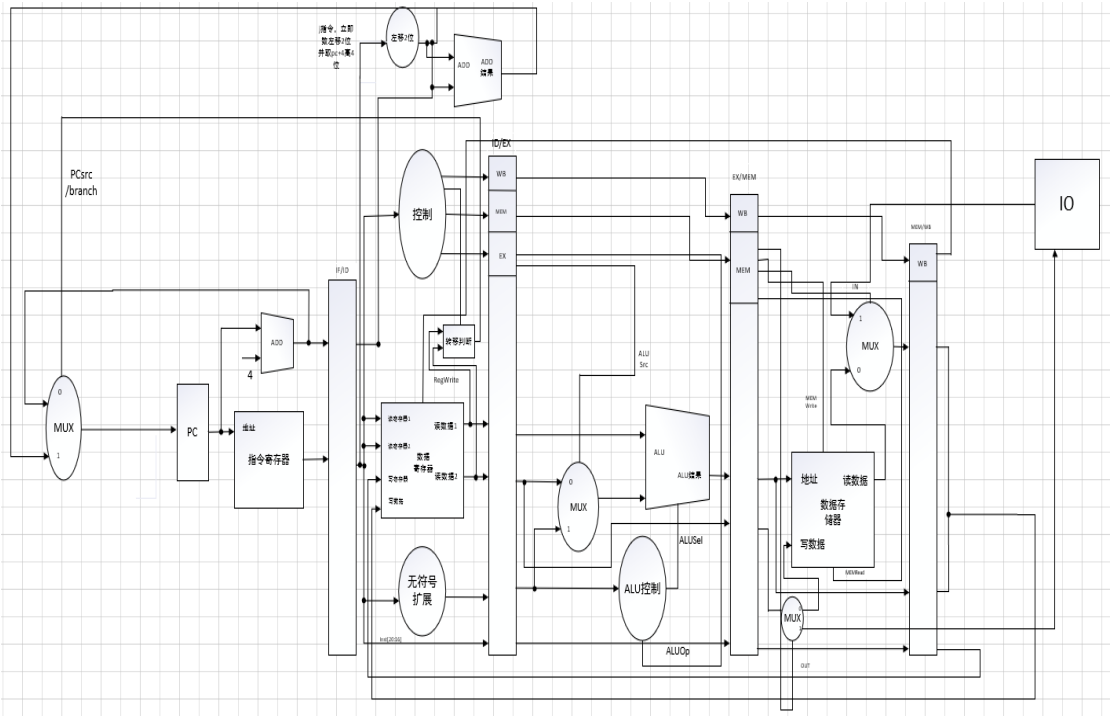

图 1　总流水级图

四、 Verilog 描述与仿真

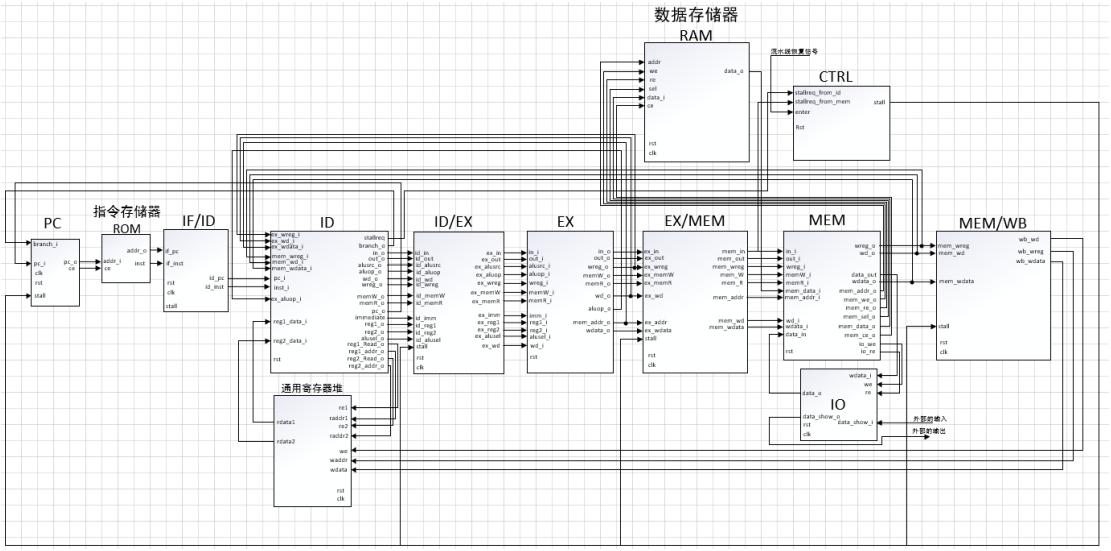

图 2　系统结构图

模块划分如图所示。本次一共设计出了 14 个模块，图上有 13 个模块，还有一个模块用来在实验箱上演示程序用的。最顶层模块 CPU 包含了 sopc 模块（包含了 OpenMIPS 模块、RAM、ROM、IO 单元）、data_io 模块（演示数据的模块）。OpenMIPS 模块又包含了 PC、IF/ID、ID、Regfile、ID/EX、EX、EX/MEM、MEM、MEM/WB、CTRL 模块。各个模块的关系及描述如下：

PC 模块：作用是给出指令地址，其接口描述如表所示

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc_i	32	输入	修改的指令地址
4	branch_i	1	输入	是否发生地址转移
5	stall	6	输入	来自控制模块，暂停信号
6	pc	32	输出	要读取的指令地址
7	ce	1	输出	指令存储器使能信号

表 9　PC 模块接口描述表

当输入的 rst 为高电平时，表示复位信号有效。

在复位的时候，输出的指令存储器使能信号为 ChipDisable，表示指令存储器禁用，其余时刻指令存储器使能信号为 ChipEnable，表示指令存储器使能。

在指令存储器禁用时，PC 的值保持为 0，当指令存储器能使用时，PC 的值

会在每时钟加 4，表示下一条的指令地址，因为一条指令是 32 位的，一条指令对应 4 个字节，所以 PC 加 4 指向下一条指令地址；如果转移分支信号 branch_i 有效，则将输出的 pc 地址修改输入的 pc 地址；如果流水级暂停，则保持输出的 pc 地址值保持不变。

IF/ID 模块：作用是暂时保存取指阶段取得的指令，以及对应的指令地址，并在下一个时钟传递到译码阶段，其接口描述如表所示

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	if_pc	32	输入	取值阶段取得的指令对应的地址
4	if_inst	32	输入	取值阶段取得的指令
5	stall	6	输入	来自控制模块，暂停信号
6	id_pc	32	输出	译码阶段的指令对应的地址
7	id_inst	32	输出	译码阶段的指令

表 10　IF/ID 模块接口描述表

IF/ID 模块只是简单地将取指阶段的结果在每个时钟周期的上升沿传递到译码阶段。当流水级暂停的时候，保持输入的值不变。

Regfile 模块：实现了 32 个 32 位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作，其接口描述如表所示

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	we	1	输入	写使能信号
6	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
7	re1	1	输入	第一个读寄存器端口度使能信号
8	rdata1	32	输出	第一个读寄存器端口读使能信号
9	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
10	re2	1	输入	第二个读寄存器端口读使能信号
11	rdata2	32	输出	第二个读寄存器端口输出的寄存器值

表 11　Regfile 模块接口描述表

Regfile 模块可以分为四段：

第一段：定义了一个二维的向量，元素个数是 32，每个元素的宽度是 32。此处定义了一个 32 个 32 位寄存器。

第二段：实现了写寄存器操作，当复位信号无效时，在写使能信号有效，且写操作目的寄存器不等于 0 的情况下，可以将写输入数据保存到目的寄存器。之所以要判断目的寄存器不为 0，是因为 MIPS32 架构规定$0 的值只能为 0，所以不要写入。

第三段：实现了第一个读寄存器端口，分以下几步依次判断：

当复位信号有效时，第一个读寄存器端口的输出始终为 0；

当复位信号有效时，如果读取读的是$0，那么直接给出 0；

如果第一个读寄存器端口要读取的目的寄存器与要写入的目的寄存器是同一个寄存器，那么直接将要写入的值作为第一个读寄存器端口的输出；

如果以上情况都不满足，那么给出第一个读寄存器端口要读取的目的寄存器地址对应寄存器的值；

第一个读寄存器端口不能使用时，直接输出 0。

第四段：实现了第二个读寄存器端口，具体过程与第三段是相似的。

读寄存器操作时组合逻辑电路，也就是一旦输入的要读取的寄存器地址 raddr1 或者 raddr2, 发生变化，那么会立即给出新地址对应的寄存器的值，这样可以保证在译码阶段取得要读取的寄存器的值，而在写寄存器操作是时序逻辑电路，写操作发生在时钟信号的上升沿。

ID 模块：作用是对指令进行译码，得到最终的指令类型、子类型、源操作数 1、第二个操作数选择信息、要写入的目的寄存器地址等信息。其中指令类型指的是存取数类型、R-type、beq，子类型指的是更加详细的运算类型，比如：当运算类型是逻辑运算时，运算子类型可以是逻辑“或”运算，逻辑“与”运算等，判断指令是否为 IN、OUT 指令，如果是，则产生相应的控制信号，在访存阶段输入与输出数据。如果是转移类型的指令，例如 beq、j 指令，那么直接在本阶段进行转移逻辑的判断，可以较少控制相关，使得延迟槽的指令只有一条，可以为空指令或者其他不想关的指令。在读寄存器值的时候，先判断当前要读的寄存器是否与前面正在执行即将要写会数据的指令的写地址是否相同，如果相同，则使用新的数据。如果不相同，则正常从通用寄存器中读取数据。其接口描述如表所示

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	pc_i	32	输入	译码阶段的指令对应的地址
3	inst_i	32	输入	译码阶段的指令
4	reg1_data_i	32	输入	从Regfile输入的第一个读寄存器端口的输入
5	reg2_data_i	32	输入	从Regfile输入的第二个读寄存器端口的输入
6	ex_wreg_i	1	输入	处于执行阶段的指令是否要写目的寄存器
7	ex_wd_i	5	输入	处于执行阶段的指令要写的目的寄存器地址
8	ex_wdata_i	32	输入	处于执行阶段的指令要写入的目的寄存器的数据
9	mem_wreg_i	1	输入	处于访存阶段的指令是否要写目的寄存器
10	mem_wd_i	5	输入	处于访存阶段的指令要写的目的寄存器地址
11	mem_wdata_i	32	输入	处于访存阶段的指令要写入的目的寄存器的数据
12	reg1_read_o	1	输出	Regfile模块的第一个读寄存器端口的读使能信号
13	reg2_read_o	1	输出	Regfile模块的第二个读寄存器端口的读使能信号
14	reg1_addr_o	5	输出	Regfile模块的第一个读寄存器端口的读地址信号
15	reg2_addr_o	5	输出	Regfile模块的第二个读寄存器端口的读地址信号
16	aluop_o	2	输出	译码阶段要进行的指令类型
17	alusel_o	4	输出	译码阶段要进行的指令具体运算类型
18	reg1_o	32	输出	端口一读出寄存器的值
19	reg2_o	32	输出	端口二读出寄存器的值
20	wd_o	5	输出	要写入的目的寄存器的地址
21	wreg_o	1	输出	是否要写入目的寄存器
22	alusrc_o	1	输出	端口二寄存器的值与立即数选择信号
23	immediate	32	输出	扩展后的立即数
24	memW_o	1	输出	数据存储单元写使能

25	memR_o	1	输出	数据存储单元读使能
26	branch_o	1	输出	是否发生地址转移
27	pc_o	32	输出	修改后的指令地址
28	in_o	1	输出	是否需要从IN单元读取数字
29	out_o	1	输出	是否将寄存器的数输出到OUT单元

表 12　ID 模块接口描述表

ID 模块中的电路都是组合逻辑电路，ID 模块与 Regfile 模块也有接口连接。ID 模块可分为三段。

第一段：实现了对指令的译码，依据指令中的特征字段区分指令，并产生对应的控制信号。

第二段：读出端口一读出的寄存器值，解决了 R 类型数据相关，判断从访存阶段和执行阶段输入的写目的寄存器地址、写使能信号，首先判断读使能是否有效，其次判断写目的寄存器地址和读寄存器地址是否一致，最后判断写使能是否有效。如果判断成功，则把即将要写回的数据作为要读的数据，从而解决了数据相关。

第三段：读出端口二读出的寄存器值，读逻辑与端口一致。

ID/EX 模块：ID 模块的输出连接到 ID/EX 模块，后者的作用是将译码阶段取得的运算类型、源操作数、要写的目的寄存器地址等结果，在下一个始终传递到流水线执行阶段。其接口描述如表所示。

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_alusel	4	输入	译码阶段要进行的指令具体运算类型
4	id_aluop	2	输入	译码阶段要进行的指令类型
5	id_reg1	32	输入	端口一读出寄存器的值
6	id_reg2	32	输入	端口二读出寄存器的值
7	id_wd	5	输入	译码阶段的指令要写入的目的寄存器地址
8	id_wreg	1	输入	译码阶段的指令是否又要写入的目的寄存器
9	id_alusrc	1	输入	译码阶段第二个操作数选择信号
10	id_imm	32	输入	译码阶段产生的扩展后立即数

11	id_memW	1	输入	译码阶段的指令是否有要写入存储单元
12	id_memR	1	输入	译码阶段的指令是否有要读存储单元
13	id_in	1	输入	译码阶段的指令是否读IN单元
14	id_out	1	输入	译码阶段的指令是否写入到OUT单元
15	stall	6	输入	来自控制模块的信息，暂停信号
16	ex_alusel	4	输出	执行阶段要进行的指令具体运算类型
17	ex_aluop	2	输出	执行阶段要进行的指令类型
18	ex_reg1	32	输出	端口一读出寄存器的值
19	ex_reg2	32	输出	端口二读出寄存器的值
20	ex_wd	5	输出	执行阶段的指令要写入的目的寄存器地址
21	ex_wreg	1	输出	执行阶段的指令是否又要写入的目的寄存器
22	ex_alusrc	1	输出	执行阶段第二个操作数选择信号
23	ex_imm	32	输出	执行阶段产生的扩展后立即数
24	ex_memW	1	输出	执行阶段的指令是否有要写入存储单元
25	ex_memR	1	输出	执行阶段的指令是否有要读存储单元
26	ex_in	1	输出	执行阶段的指令是否有要读IN单元
27	ex_out	1	输出	执行阶段的指令是否有要写入到OUT单元

表 13　ID/EX 模块接口描述表

ID/EX 模块只是简单地将译码阶段的结果在时钟周期上升沿传递到执行阶段。执行阶段将依据这些值进行运算。当流水级暂停的时候，保持输入的值不变。

EX 模块：从 ID/EX 模块得到运算类型 alusel_i、指令类型 aluop_i、源操作数 reg1_i、源操作数 reg2_i、要写入的目的寄存器地址 wd_i、alusrc_i、immediate。EX 模块会依据这些数据进行运算，其接口描述如表所示。

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	alusel_i	4	输入	执行阶段要进行的运算类型
3	aluop_i	2	输入	执行阶段要进行的指令类型
4	alusrc_i	1	输入	执行阶段要进行的第二个操作数选择信号
5	imm_i	32	输入	执行阶段要进行运算的扩展后立即数

6	reg1_i	32	输入	端口一读出寄存器的值
7	reg2_i	32	输入	端口二读出寄存器的值
8	wd_i	5	输入	执行执行要写入的目的寄存器地址
9	wreg_i	1	输入	是否有要写入的目的寄存器
10	memW_i	1	输入	是否要写入存储单元
11	memR_i	1	输入	是否要从存储单元读
12	in_i	1	输入	是否要从IN单元读
13	out_i	1	输入	是否写入到OUT单元
14	wd_o	5	输出	执行阶段的指令最终要写入的目的寄存器地址
15	wreg_o	1	输出	执行阶段的指令最终要是否有要写入的目的寄存器
16	wdata_o	32	输出	执行阶段的指令最终要写入目的寄存器的值
17	memW_o	1	输出	是否要写入存储单元
18	memR_o	1	输出	是否要从存储单元读
19	mem_addr_o	32	输出	要访问数据存储器RAM地址
20	in_o	1	输出	是否要从IN单元读
21	out_o	1	输出	是否写入到OUT单元

表 14　EX 模块接口描述表

EX 模块中都是组合逻辑电路，可分为两端理解。

第一段：依据输入的运算子类型进行运算。

第二段：给出最终的运算结果，包括是否要写目的寄存器 wreg_o、要写的目的寄存器地址 wd_o、要写入的数据 wdata_o。其中 wreg_o、wd_o 的值都直接来自译码阶段，不需要改变，wdata_o 的值要依据指令类型以及运算子类型进行选择。

EX/MEM 模块：输出连接到 EX/MEM 模块，后者的作用是将执行阶段取得的运算结果，在下一个时钟传递到流水线访存阶段。其接口描述如表所示。

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	ex_wd	5	输入	执行阶段的指令执行后要写入的目的寄存器地址
4	ex_wreg	1	输入	执行阶段的指令执行后是否有要写入的目的寄存器
5	ex_wdata	32	输入	执行阶段的指令执行后要写入目的寄存器的值或访存单元地址
6	ex_memW	1	输入	执行阶段的指令执行后是否有要写入的存储单元
7	ex_memR	1	输入	执行阶段的指令执行后是否有要从存储单元中读取
8	ex_addr	32	输入	执行阶段执行后要访问的存储单元地址
9	ex_in	1	输入	执行阶段的指令执行后是否要IN单元读取数
10	ex_out	1	输入	执行阶段的指令执行后是否要写入到OUT单元
11	stall	6	输入	来自控制模块的信息，暂停信号
12	mem_wd	5	输出	访存阶段的指令要写入的目的寄存器地址
13	mem_wreg	1	输出	访存阶段的指令是否有要写入的目的寄存器
14	mem_wdata	32	输出	访存阶段的指令要写入目的寄存器的值或访存单元地址
15	mem_W	1	输出	访存阶段是否要写入存储单元
16	mem_R	1	输出	访存阶段是否从存储单元中读取
17	mem_addr	32	输出	访存阶段要访问的存储单元地址
18	mem_in	1	输出	访存阶段是否要从IN单元读取，而不是从存储单元
19	mem_out	1	输出	访存阶段是否要写入到OUT单元，而不是存储单元

表 15　EX/MEM 模块接口描述表

EX/MEM 模块中，只有一个时序逻辑电路，在时钟上升沿，将执行阶段的结果传递到访存阶段。当流水级暂停的时候，保持输入的值不变。

MEM 模块：判断 MEMW_i、MEMR_i、MEM_in、MEM_out 信号的值，从而决定数据从 IN 单元还是存储单元给入，或者数据从 OUT 单元还是存储单元给出。其接口描述如表所示

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	wd_i	5	输入	访存阶段的指令要写入的目的寄存器地址
3	wreg_i	1	输入	访存阶段的指令是否有要写入的目的寄存器
4	wdata_i	32	输入	访存阶段的指令要写入目的寄存器的值
5	memW_i	1	输入	访存阶段的指令是否要写入存储单元
6	memR_i	1	输入	访存阶段的指令是否要从存储单元中读
7	mem_data_i	32	输入	来自外部数据存储器RAM的信息
8	mem_addr_i	32	输入	指令要访问访问存储器的地址
9	mem_in	1	输入	是否要从IN单元读取，而不是从存储单元
10	mem_out	1	输入	是否写入到OUT单元，而不是存储单元
11	data_in	32	输入	来自IN单元输入的数
12	wd_o	5	输出	访存阶段的指令最终要写入的目的寄存器的地址
13	wreg_o	1	输出	访存阶段的指令最终是否有要写入的目的寄存器
14	wdata_o	32	输出	访存阶段的指令最终要写入目的寄存器的值
15	mem_addr_o	32	输出	要访问的数据存储器的地址
16	mem_we_o	1	输出	是否是写操作，为1表示是写操作
17	mem_re_o	1	输出	是否是读操作，为1表示是读操作
18	mem_sel_o	4	输出	字节选择信号
19	mem_data_o	32	输出	要写入数据存储器的数据
20	mem_ce_o	1	输出	数据存储器使能信号
21	data_out	32	输出	输出到OUT单元的数
22	io_we	1	输出	IO单元写使能信号
23	io_re	1	输出	IO单元读使能信号

表 16　MEM 模块接口描述表

MEM 模块中只有一个组合逻辑电路，将输入的执行阶段的结果直接作为输出，MEM 模块的输出连接到 MEM/WB 模块。

MEM/WB 模块：作用是将访存阶段的运算结果，在一个时钟传递到回写阶段，其接口描述如表所示。

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_wd	5	输入	访存阶段的指令最终要写入的目的寄存器地址
4	mem_wreg	1	输入	访存阶段指令最终是否有要写入的目的寄存器
5	mem_wdata	32	输入	访存阶段的指令最终要写入目的寄存器的值
6	stall	6	输入	来自控制模块的信息，暂停信号
7	wb_wd	5	输出	回写阶段的指令要写入的目的寄存器地址
8	wb_wreg	1	输出	回写阶段的指令是否有要写入的目的寄存器
9	wb_wdata	32	输出	回写阶段的指令要写入目的寄存器的值

表 17　MEM/WB 模块接口描述表

MEM/WB 模块中是时序逻辑电路，即在时钟上升沿才发生信号传递，而 MEM 模块中的是组合逻辑电路，MEM/WB 模块将访存阶段指令是否要写目的寄存器 mem_wreg、要写的目的寄存器地址 mem_wd、要写入的数据 mem_wdata 等信息传递到回写阶段对应的接口 wb_wreg、wb_wd、wb_wdata。

数据存储器 RAM：其接口描述如表所示

序号	接口名	宽度（bit）	输入/输出	作用
1	clk	1	输入	时钟信号
2	data_i	32	输入	要写的数据
3	addr	32	输入	要访问的地址
4	we	1	输入	是否是写操作，为1表示写操作
5	re	1	输入	是否是读操作，为1表示读操作
6	sel	4	输入	字节选择信号
7	ce	1	输入	芯片使能信号
8	data_o	32	输出	读出的数据

表 18　数据存储器 RAM 接口描述表

为了方便实现对数据存储器按字节寻址，在设计的时候使用 4 个 8 位存储器代替一个 32 位存储器。

在读操作时，从 4 个 8 位存储器中各读出一个字节，组合为一个 32 位的数据输出。

写操作时，依据 sel 的值，修改其中特定存储器对应的字节即可。

指令存储器 ROM：其接口描述如表所示

序号	接口名	宽度（bit）	输入/输出	作用
1	ce	1	输入	使能信号
2	addr	32	输入	要读取的指令地址
3	inst	32	输出	读出的指令

表 19　指令存储器 ROM 接口描述表

OpenMIPS 是按字节寻址的，而此处定义的指令寄存器的每个地址是一个 32bit 的字，所以要将 OpenMIPS 给出的指令地址除以 4 再使用。

IO 单元：为了实现 IN、OUT 指令，加入 IO 单元，其接口描述如下

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	we	1	输入	写使能信号，为1表示可写入
4	re	1	输入	读使能信号，为1表示可读
5	data_i	32	输入	输出给OUT指令的数据
6	data_show_i	32	输入	来自于外部的数据
7	data_o	32	输出	输入给IN指令的数据
8	data_show_o	32	输出	将数据输出到外部
9	out	1	输出	输出数据显示使能

表 10　IO 单元接口描述表

IO 单元读操作，从外部读取数据时是组合逻辑。

IO 单元写操作，将数据写出到外部是时序逻辑。

CTRL 模块：为了实现流水线的暂停，加入 CTRL 模块。IN 指令需要暂停流水线，并且是在访存阶段实现的发出暂停请求。其接口描述如下

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	stallreq_from_mem	1	输入	处于访存阶段的指令是否请求流水线暂停
3	enter	1	输入	来自于外部的取消暂停，为1表示取消
4	stall	6	输出	暂停流水线控制信号

表 11　CTRL 模块接口描述表

输出信号 stall 是一个宽度为 6 的信号，其含义如下。

stall [0] 表示取指地址 PC 是否保持不变，为 1 表示保持不变。

stall [1] 表示流水线取值阶段是否暂停，为 1 表示暂停。

stall [2] 表示流水线译码阶段是否暂停，为 1 表示暂停。

stall [3] 表示流水线执行阶段是否暂停，为 1 表示暂停。

stall [4] 表示流水线访存阶段是否暂停，为 1 表示暂停。

stall [5] 表示流水线回写阶段是否暂停，为 1 表示暂停。

OpenMIPS 模块：顶层模块 OpenMIPS 中主要是对流水线中各个阶段的模块进行例化、连接。OpenMIPS 模块的接口描述如下

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	rom_data_i	32	输入	从指令存储器取得的指令
4	rom_addr_o	32	输出	输出到指令存储器的地址
5	rom_ce_o	1	输出	指令存储器使能信号
6	ram_data_i	32	输入	从数据存储器读取的数据
7	ram_addr_o	32	输出	要访问的数据存储器地址
8	ram_we_o	1	输出	是否对数据存储器的写操作，为1表示是写操作
9	ram_ce_o	1	输出	是否对数据存储器的读操作，为1表示是读操作
10	ram_sel_o	4	输出	字节选择信号
11	ram_data_o	32	输出	要写入数据存储器的数据
12	ram_ce_o	1	输出	数据存储器使能信号
13	data_i	32	输入	来自IN单元的输入的数

14	data_o	32	输出	输出到OUT单元的数
15	io_we_o	1	输出	输入到IO单元的写使能信号
16	io_re_o	1	输出	输入到IO单元的读使能信号
17	enter_i	1	输入	来自外部的恢复流水级信号

表 12　OpenMIPS 模块接口描述表

SOPC 模块：为了验证，需要建立一个 SOPC，其中仅含有 OpenMIPS、指令存储器 ROM、数据寄存器 RAM，所以是一个最小 SOPC。

OpenMIPS 从指令存储器 ROM 中读取指令，指令进入 OpenMIPS 开始执行。

OpenMIPS 从数据寄存器 RAM 中读取数据，数据进入 OpenMIPS 中存储到通用寄存器 Regfile 中。

最小 SOPC 的接口描述如下

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	enter	1	输入	来自外部的恢复流水级信号
4	data_show_i	32	输入	来自外部输入的数据
5	data_show_o	32	输出	输出到外部
6	pc	32	输出	指令地址
7	out_o	1	输出	输出数据显示使能

表 13　SOPC 模块接口描述表

data_io 模块：将当前 PC 地址输出到实验箱中，从实验箱中读取数据，并输出运算结果。其接口描述如下

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	k1_ten	1	输入	高位按键
4	k2_ge	1	输入	低位按键
5	out_i	1	输入	显示数据使能
6	pc_in	32	输入	pc地址
7	res	32	输入	指令运算结果
8	data	32	输出	作为程序要读取的数
9	sel	3	输出	位选
10	seg	7	输出	段选

表 14　data_io 模块接口描述表

data_io 模块，将 pc 地址取低 8 位，以 16 进制输出到两位数码管中。取实验箱中两位 16 进制数作为输入数据，并将运算结果作为 16 进制数输出。

CPU 模块：包含了 SOPC 和 data_io 模块，其接口描述如下

序号	接口名	宽度（bit）	输入/输出	作用
1	CLOCK_50	1	输入	时钟信号
2	rst	1	输入	复位信号
3	enter	1	输入	取消流水级暂停信号
4	k1	1	输入	高位按键
5	k2	1	输入	低位按键
6	sel	3	输出	位选信号
7	seg	8	输出	段选信号

表 15　CPU 模块接口描述表

测试程序测试了所有的指令，检验数据相关是否解决、IN 指令能否暂停流水级。

Modelsim 仿真中遇到的问题：由于测试文件中产生的是 50MHZ 的时钟信号，经过分频之后变为 1HZ 的时钟信号，将 1HZ 的时钟信号输入给了 sopc 模块。刚开始复位信号，还没有达到第一个上升沿，就置为无效，所以指令存储器读不出指令，读使能信号也无效及输入的指令地址也无效。但是，经过测试之后发现，复位信号要保持 1HZ 时钟信号的两个上升沿，最终才可以读出来指令。

add 指令执行过程：

在第一个时钟上升沿，pc 给出指令地址，指令存储器读出指令 32'h04222821，并进入 IF/ID 流水级。在第二时钟个上升沿，进入译码阶段，根据指令中给出的 Rs、Rt 寄存器的地址，读出相应的值。端口一（地址为 00001）读出的值为 5，端口二（地址为 00010）读出的值为 5，aluop 为 10（表示 R-type 类型的指令），alusel 为 0010（表示加法），wreg 为 1（表示要写入结果要写回寄存器），要写回寄存器地址为 00101，并将产生相应的控制信号送入 ID/EX 流水级。在第三个时钟上升沿，进入执行阶段，根据 aluop、alusel 控制信号的值，确定运算结果为 10，并将运算结果及相应的控制信号送入 EX/MEM 流水级。在第四个时钟上升沿，进入访存阶段，因为 R-type 指令不需要访存，所以直接将所有数据送入到 MEM/WB 流水级。在第五个时钟上升沿，将运算结果写回到寄存器中。

beq 指令执行过程：

在第一个时钟上升沿，pc 给出指令地址，指令存储器读出指令32'h10220004，并进入 IF/ID 流水级。在第二个时钟上升沿，进入译码阶段，根据指令中给出的 Rs、Rt 寄存器的地址，读出相应的值，并判断是否相等，如果相等则将 branch 信号置 1，计算修改后的 pc 地址，当前 beq 指令的地址+4 与指令中扩展之后的立即数相加，将修改后的 pc 地址以及 branch 控制信号给 pc 模块。

将转移判断逻辑放在了译码阶段，那么延迟槽指令只有一条，这条延迟槽指令可以是一条无相关的指令或者空指令。

IN 指令执行过程：IN 指令是将外部输入的数据，直接送入到通用寄存器堆中。

在第一个时钟上升沿，pc 给出指令地址，指令存储器读出指令32'h0C2B0001，并进入 IF/ID 流水级。在第二个上升沿，进入译码阶段，因为 IN 指令不需要从寄存器读值，所以端口一与端口二的读使能全为 0，IN 使能为 1。确定写入目的寄存器地址为 01011，写使能为 1。在第三个时钟上升沿，进入执行阶段，IN 指令不需要确定运算结果，所以直接将写使能、写目的寄存器地址送入到 EX/MEM 流水级。在第四个时钟上升沿，进入访存阶段，IN 使能信号送入到 CTRL 模块，暂停整个流水级执行，等待外部输入的数据及恢复流水级信号。将输入的数据作为 MEM/WB 流水级的输入。在第五个时钟上升沿，回写数据到通用寄存器堆。

OUT 指令执行过程：OUT 指令是将通用寄存器的值，直接输出到外部。

在第一个时钟上升沿，pc 给出指令地址，指令存储器读出指令32'h2C270001，并进入 IF/ID 流水级。在第二个上升沿，进入译码阶段，因为 OUT 指令需要从寄存器读值，所以端口二的读使能为 1，out 使能为 1。不需要写入到寄存器与存储单元中，通用寄存器写使能为 0，存储单元写使能为 1。在第三个时钟上升沿，进入执行阶段，out 指令将运算结果确定为端口 2 读出的值，并进入 EX/MEM 流水级。在第四个时钟上升沿，进入访存阶段，out 使能为 1，将输入的数据送入到外部，并进入 MEM/WB 流水级。在第五个时钟上升沿，不用写数据到通用寄存器堆。

五、 可编程逻辑器件实现

遇到的问题及解决方案，对于同一个信号，不可以在两个及以上 always 块中赋值，将赋值语句都放在了一个 always 块中。

本次实验的结果，用八位数码管展示，其中两位数码管用来输出 PC 的地址，两位数码管用来显示输入的数据，两位数码管用来显示程序输出的结果。数字都是以十六进制表示。使用了两位拨动开关，一个复位开关，一个恢复流水级开关。

使用了两个按键，一个用来控制高位的数字加 1，一个用来控制低位的数字加 1。

测试程序测试了所有的指令，并且检验数据相关是否解决、IN 指令能否暂停流水级。通过输出的 pc 地址进行观察，看是否可以暂停流水级。通过输出的结果，查看数据相关是否被解决。

硬件调试中遇到的问题，①时钟信号选择有误，最后选择核心板上的时钟信号 50MHZ 作为输入。②段选管脚设置有问题，小数点 dp 如果不给值，默认点亮，所以段选是八位。③按键消抖，最终是将 50MHZ 分为 1HZ，并在 1HZ 的时钟上升沿判断按键是否被按下。

当程序读取 IN 指令，IN 指令在访存阶段时，会暂停流水级。此时 PC 的输出保持不变，并且等待数据的输入。手动输入数据（通过按键调节数字），拨动恢复流水级开关，程序继续运行。程序的输出结果，验证了 R-type 指令数据相关得以解决，以及输入进去的数字可以被使用。

六、 总结、感想和收获

李琦的总结：

本次课程设计，实现了一个 32 位的五级流水 RISC 指令集的 CPU。一共设计了 9 条基本指令以及 IN、OUT 指令。主要的工作的内容为：参与指令集的设计、五级流水结构图的设计、系统结构图的设计、真值表的设计、负责全部模块的设计和最终实验箱的验证（采用八位数码管演示程序）。

程序设计的思路为，首先按照 R-type、I-type、J-type 的指令不同，逐步实现每条指令，并在 ModelSim 中查看波形是否正确，最后加入 IN、OUT 指令。由于最终要在实验箱上展示，所以我实现了人机交互，当执行 IN 指令的时候，暂停流水级的执行，等待数据的输入以及恢复流水级信号。并且解决 R-type 指令的数据相关，当 EX 模块得出运算结果以及访存阶段，直接将写入的地址、写入的数据、写使能送入译码阶段，在译码阶段判断当前指令要读的地址与即将要写回地址是否一致，如果一致，那么读的是“新”值，而不是寄存器中的“旧”值。最终一共设计了 18 个模块，每个模块有各自的功能，并通过不同的顶层模块将它们例化、连接。本次是通过用程序来模拟流水级的执行过程，变量都是通过用数组进行保存。但实际 CPU 是通过用硬件来存储信息，SRAM 及 SDRAM，从而提高了速度与性能。但是，最终并没有实现系统总线，所以没有达到硬件与软件的结合。

刚开始对 verilog 语言进行熟悉，了解组合电路与时序电路的不同，以及思考每个模块应该是什么电路。在设计时，遇到了许多的麻烦，通过一步步地排查，最终得到想要的结论。上课时，对于流水级的执行过程不是非常理解，但是通过这次课程，加深了理解，并有了一些自己的改进，实践是检验真理的唯一标准。

王雕的总结：

主要的工作内容为：参与指令集的设计、五级流水结构图的设计、真值表的设计；首先根据设计出来的真值表用卡诺图进行化简，从而设计出了主控制器和 ALU 控制器的电路图；其次参与了模块的设计；最后实现了分频功能和 8 段译码功能。

通过这次的课设，对 Verilog 语言进行了复习，熟练的掌握了 Verilog 语言，并且用它进行了编程，对卡诺图的化简和电路图的绘画也精通了很多。深刻理解了 5 级流水 CPU 的工作原理，了解了组合逻辑和时序逻辑的不同之处，对很多指令有了新的了解，根据书上所学的知识，对数据相关进行了解决。期间遇到了一些问题，通过网上的查找、和组长进行讨论，最终把问题都一个一个解决了。曾经我觉得设计一个 CPU 出来很不可思议，但这次在自己的努力下最终设计出来了，也渐渐有了整体框架的思想。

七、参考文献

[1]雷思磊. 自己动手写 CPU[M].北京：电子工业出版社，2014.9

[2]（美）帕特森，（美）亨尼斯著；王党辉等译. 计算机组成与设计：硬件/软件接口（原书第 5 版）[M]. 北京：机械工业出版社，2015.6

附录 1：

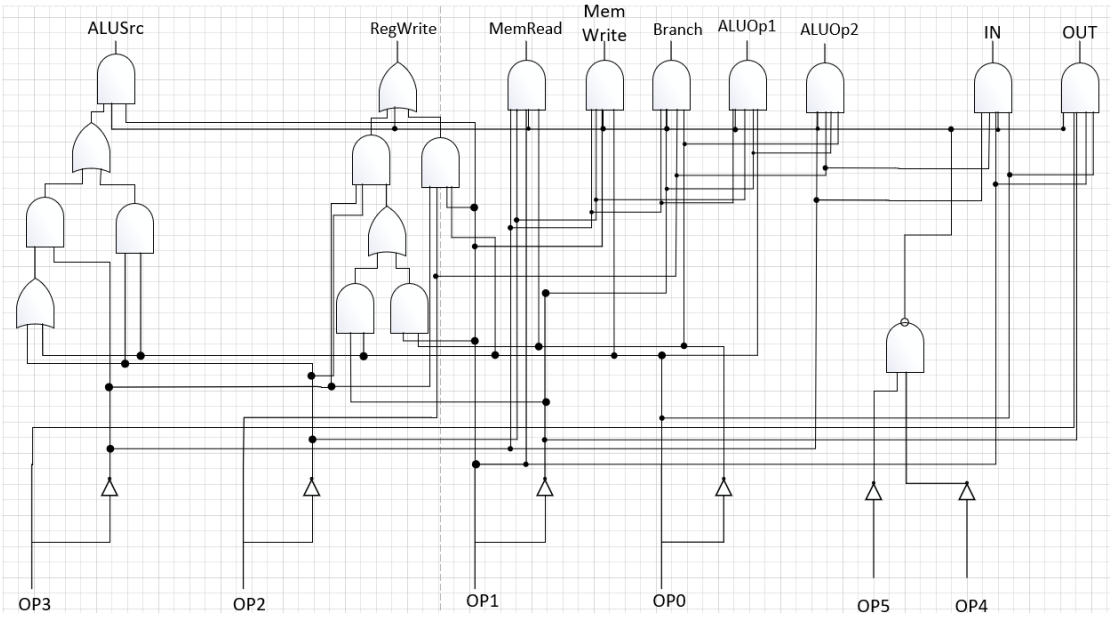

图 3　主控制器结构图

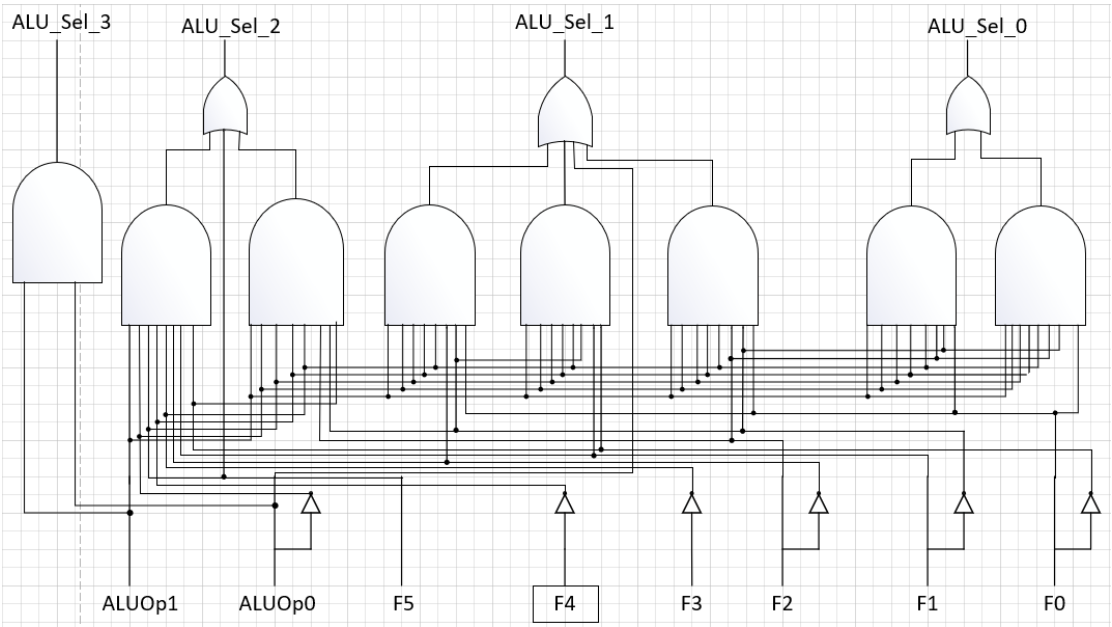

图 4　ALU 控制器结构图

附录 2：

宏定义文件：

```
//*********************     全局的宏定义     *********************
`define RstEnable              1'b1                    //复位信号有效
`define RstDisable             1'b0                    //复位信号无效
`define ZeroWord               32'h00000000            //32 位的数值 0
`define WriteEnable            1'b1                    //使能写
`define WriteDisable           1'b0                    //禁止写
`define ReadEnable             1'b1                    //使能读
`define ReadDisable            1'b0                    //禁止读
`define AluOpBus               1:0                     //译码阶段的输出 AluOp_o 的宽度
`define AluSelBus              3:0                     //执行阶段的输出 AluSel_o 的宽度
`define InstValid              1'b0                    //指令有效
`define InstInvalid            1'b1                    //指令无效
`define ChipEnable             1'b1                    //芯片使能
`define ChipDisable            1'b0                    //芯片禁止
//***************     与具体指令有关的宏定义     *****************
`define      EXE_R_type        6'b000001               //R_type 的指令码
`define EXE_lw_m               6'b000010               //lw_m 指令的指令码
`define      EXE_lw_io               6'b000011                  //lw_io 指令的指令码
`define EXE_sw_m               6'b000111               //sw_m 指令的指令码
`define EXE_sw_io              6'b001011               //sw_io 指令的指令码
`define EXE_beq                6'b000100               //beq 指令的指令码
`define EXE_j                  6'b000101               //j 指令的指令码
`define EXE_NOP                6'b000000               //nop 指令的指令码
`define EXE_ADD                6'b100001               //add 指令的功能码
`define EXE_SUB                6'b100010               //sub 指令的功能码
`define EXE_OR                 6'b100011               //or 指令的功能码
`define EXE_AND                6'b100100               //and 指令的功能码
`define EXE_SLT                6'b100101               //slt 指令的功能码
//AluOp
`define EXE_ls_op              2'b00                   //表示 lw_m , lw_io , sw_m , sw_io 指令
`define EXE_arithmetic_op 2'b10                        //表示 R-type 类型的指令
`define EXE_nop_op             2'b11                   //表示 nop 指令
//AluSel
`define EXE_AND_sel            4'b0000                 //与
`define EXE_OR_sel             4'b0001                 //或
`define EXE_ADD_sel            4'b0010                 //加
`define EXE_SUB_sel            4'b0110                 //减
`define EXE_SLT_sel            4'b0111                 //条件设置
`define EXE_NOP_sel            4'b1111                 //无操作

//*************     与指令存储器 ROM 有关的宏定义     **************
`define InstAddrBus            31:0                    //ROM 的地址总线宽度
`define InstBus                31:0                    //ROM 的数据总线宽度
`define InstMemNum             131071                  //ROM 的实际大小 128KB
`define InstMemNumLog2    17                           //ROM 的实际使用的地址线宽度
//*************     与通用寄存器 Regfile 有关的宏定义     **************
`define RegAddrBus             4:0                     //Regfile 模块的地址线宽度
`define RegBus                 31:0                    //Regfile 模块的数据线宽度
`define Regwidth               32                      //通用寄存器的宽度
`define DoubleRegWidth         64                      //两倍的通用寄存器的宽度
`define DoubleRegBus           63:0                    //两倍的通用寄存器的数据线的宽度
`define RegNum                 32                      //通用寄存器的数量
`define      RegNumLog2        5                       //寻址通用寄存器使用的地址位数
`define NOPRegAddr             5'b00000
//*************     与通用寄存器 Regfile 有关的宏定义     **************
`define DataAddrBus            31:0                    //地址总线宽度
`define      DataBus           31:0                    //数据总线宽度
`define      DataMemNum        131071                  //RAM 的大小，单位是字，此处是 128K word
`define DataMemNumLog2     17                          //实际使用的地址宽度
`define      ByteWidth         7:0                     //一个字宽度，8bit
//*************     与转移指令有关的宏定义               **************
`define      Branch            1'b1                    //转移
`define      NotBranch         1'b0                    //不转移
//*************     与 IO 单元有关的宏定义               **************
```

```
`define In                  1'b1            //表示需要从 IN 单元读取数据
`define     NotIn           1'b0            //表示不需要从 IN 单元读取数据
`define     Out             1'b1          //表示需要将数据输出到 OUT 单元
`define     NotOut          1'b0          //表示不需要将数据输出到 OUT 单元
//************     与流水线暂停有关的宏定义          **************
`define     Stop            1'b1            //流水线暂停
`define     NoStop          1'b0          //流水线继续
//************     与数码管段选有关的宏定义  gfedcba **************
`define     data_0                  8'b00111111     //0
`define     data_1                  8'b00000110     //1
`define     data_2                  8'b01011011     //2
`define     data_3                  8'b01001111     //3
`define     data_4                  8'b01100110     //4
`define     data_5                  8'b01101101     //5
`define     data_6                  8'b01111101     //6
`define     data_7                  8'b00000111     //7
`define     data_8                  8'b01111111     //8
`define     data_9                  8'b01101111     //9
`define     data_a                  8'b01110111     //a
`define     data_b                  8'b01111100     //b
`define     data_c                  8'b00111001     //c
`define     data_d                  8'b01011110     //d
`define     data_e                  8'b01111001     //e
`define     data_f                  8'b01110001     //f
```

CPU 模块：

```
`include "defines.v"
module cpu(
    input wire      CLOCK_50,
    input wire      rst,
    input wire      enter,
    input wire      k1,
    input wire      k2,
    output      wire[2:0]   sel,
    output      wire[7:0]   seg
);
    wire  clk_1_o;
    wire[`InstAddrBus]          pc_o;
    wire[`RegBus]               data_from_in;
    wire                        out;
    wire[`RegBus]               res_o;
    //例化 sopc 模块
    openmips_spoc openmips_spoc0(
            .clk(clk_1_o),                  .rst(rst),
            .enter(enter),          .data_show_i(data_from_in),

            .data_show_o(res_o),        .out_o(out),
            .pc(pc_o)
    );
    //例化 data_io 模块
    data_io data_io0(
            .clk(CLOCK_50),     .rst(rst),
            .k1_ten(k1),        .k2_ge(k2),
            .pc_in(pc_o),       .res(res_o),
            .out_i(out),
            .data_o(data_from_in),.sel(sel),
            .seg(seg)
    );
    //例化分频模块
    fenpin fenpin0(
            .clk(CLOCK_50),     .rst(rst),
            .clk_1(clk_1_o)
    );
endmodule
```

data_io 模块：

```
`include "defines.v"
module data_io(
    input wire  clk,
```

```
        input wire  rst,
        input wire  k1_ten,
        input wire  k2_ge,
        input wire[`InstAddrBus]    pc_in,
        input wire[`RegBus]         res,
        input wire                  out_i,
        output      reg[`RegBus]                data_o,
        output      reg[2:0]                    sel,
        output      reg[7:0]                    seg
);
        parameter   T1KHZ = 2_499;
        parameter   T1HZ = 24_999_999;
        integer       cnt1;
        integer           cnt2;
        reg         clk_10k;
        reg         clk_1;
        reg[`RegBus]    result;
        reg[3:0]          shiwei;
        reg[3:0]          gewei;
        wire[3:0]   pc_2;
        wire[3:0]   pc_1;
        wire[3:0]   res_2;
        wire[3:0]   res_1;
        assign      pc_2 = pc_in[3:0];
        assign      pc_1 = pc_in[7:4];
        assign      res_2 = result[3:0];
        assign      res_1 = result[7:4];
        initial begin
            sel = 3'b000;
        end
        always@ (posedge clk) begin
            if(rst == `RstEnable) begin
                cnt1  <= 0;
                clk_10k <= 0;
                cnt2  <= 0;
                clk_1<= 0;
            end
            if(cnt1 == T1KHZ) begin
                clk_10k <= ~clk_10k;
                cnt1 <= 0;
            end else begin
                cnt1 <= cnt1 + 1;
            end
            if(cnt2 == T1HZ) begin
                clk_1 <= ~clk_1;
                cnt2  <= 0;
            end else begin
                cnt2 <= cnt2 + 1;
            end
        end
        always@ (posedge clk_1) begin
            if(rst == `RstEnable) begin
                shiwei <= 4'b0000;
                gewei <= 4'b0000;
            end
            if(k1_ten == 1'b0) begin
                if(shiwei == 4'b1111) begin
                    shiwei <= 4'b0000;
                end else begin
                    shiwei <= shiwei + 1;
                end
            end
            if(k2_ge == 1'b0) begin
                if(gewei == 4'b1111) begin
                    gewei <= 4'b0000;
                end else begin
                    gewei <= gewei + 1;
                end
            end
```

```
end
always@ (posedge clk_10k) begin
	if(sel == 3'b111) begin
		sel <= 3'b000;
	end else begin
		sel <= sel + 1;
	end
end
always@ (*) begin
	if(rst == `RstEnable) begin
		seg <= `data_0;
		result <= `ZeroWord;
		data_o <= `ZeroWord;
		//shiwei <= 4'b0000;
		//gewei <= 4'b0000;
	end else begin
		if(out_i == `WriteEnable) begin
			result <= res;
		end
		data_o <= {26'd0,shiwei[3:0],gewei[3:0]};
		if(sel == 3'b000) begin
			case(pc_1)
				4'd0: begin
					seg = `data_0;
				end
				4'd1: begin
					seg = `data_1;
				end
				4'd2: begin
					seg = `data_2;
				end
				4'd3: begin
					seg = `data_3;
				end
				4'd4: begin
					seg = `data_4;
				end
				4'd5: begin
					seg = `data_5;
				end
				4'd6: begin
					seg = `data_6;
				end
				4'd7: begin
					seg = `data_7;
				end
				4'd8: begin
					seg = `data_8;
				end
				4'd9: begin
					seg = `data_9;
				end
				4'd10: begin
					seg = `data_a;
				end
				4'd11: begin
					seg = `data_b;
				end
				4'd12: begin
					seg = `data_c;
				end
				4'd13: begin
					seg = `data_d;
				end
				4'd14: begin
					seg = `data_e;
				end
				4'd15: begin
					seg = `data_f;
```

```
                end
        endcase
end
else if(sel == 3'b001) begin
        case(pc_2)
                4'd0: begin
                        seg = `data_0;
                end
                4'd1: begin
                        seg = `data_1;
                end
                4'd2: begin
                        seg = `data_2;
                end
                4'd3: begin
                        seg = `data_3;
                end
                4'd4: begin
                        seg = `data_4;
                end
                4'd5: begin
                        seg = `data_5;
                end
                4'd6: begin
                        seg = `data_6;
                end
                4'd7: begin
                        seg = `data_7;
                end
                4'd8: begin
                        seg = `data_8;
                end
                4'd9: begin
                        seg = `data_9;
                end
                4'd10: begin
                        seg = `data_a;
                end
                4'd11: begin
                        seg = `data_b;
                end
                4'd12: begin
                        seg = `data_c;
                end
                4'd13: begin
                        seg = `data_d;
                end
                4'd14: begin
                        seg = `data_e;
                end
                4'd15: begin
                        seg = `data_f;
                end
        endcase
end
else if(sel == 3'b110) begin
        case(res_1)
                4'd0: begin
                        seg = `data_0;
                end
                4'd1: begin
                        seg = `data_1;
                end
                4'd2: begin
                        seg = `data_2;
                end
                4'd3: begin
                        seg = `data_3;
                end
```

```
                4'd4: begin
                        seg = `data_4;
                end
                4'd5: begin
                        seg = `data_5;
                end
                4'd6: begin
                        seg = `data_6;
                end
                4'd7: begin
                        seg = `data_7;
                end
                4'd8: begin
                        seg = `data_8;
                end
                4'd9: begin
                        seg = `data_9;
                end
                4'd10: begin
                        seg = `data_a;
                end
                4'd11: begin
                        seg = `data_b;
                end
                4'd12: begin
                        seg = `data_c;
                end
                4'd13: begin
                        seg = `data_d;
                end
                4'd14: begin
                        seg = `data_e;
                end
                4'd15: begin
                        seg = `data_f;
                end
        endcase
end
else if(sel == 3'b111) begin
        case(res_2)
                4'd0: begin
                        seg = `data_0;
                end
                4'd1: begin
                        seg = `data_1;
                end
                4'd2: begin
                        seg = `data_2;
                end
                4'd3: begin
                        seg = `data_3;
                end
                4'd4: begin
                        seg = `data_4;
                end
                4'd5: begin
                        seg = `data_5;
                end
                4'd6: begin
                        seg = `data_6;
                end
                4'd7: begin
                        seg = `data_7;
                end
                4'd8: begin
                        seg = `data_8;
                end
                4'd9: begin
                        seg = `data_9;
```

```
                end
                4'd10: begin
                        seg = `data_a;
                end
                4'd11: begin
                        seg = `data_b;
                end
                4'd12: begin
                        seg = `data_c;
                end
                4'd13: begin
                        seg = `data_d;
                end
                4'd14: begin
                        seg = `data_e;
                end
                4'd15: begin
                        seg = `data_f;
                end
        endcase
end
else if(sel == 3'b100) begin
        case(shiwei)
                4'd0: begin
                        seg = `data_0;
                end
                4'd1: begin
                        seg = `data_1;
                end
                4'd2: begin
                        seg = `data_2;
                end
                4'd3: begin
                        seg = `data_3;
                end
                4'd4: begin
                        seg = `data_4;
                end
                4'd5: begin
                        seg = `data_5;
                end
                4'd6: begin
                        seg = `data_6;
                end
                4'd7: begin
                        seg = `data_7;
                end
                4'd8: begin
                        seg = `data_8;
                end
                4'd9: begin
                        seg = `data_9;
                end
                4'd10: begin
                        seg = `data_a;
                end
                4'd11: begin
                        seg = `data_b;
                end
                4'd12: begin
                        seg = `data_c;
                end
                4'd13: begin
                        seg = `data_d;
                end
                4'd14: begin
                        seg = `data_e;
                end
                4'd15: begin
```

```
                        seg = `data_f;
                    end
                endcase
            end
            else if(sel == 3'b101) begin
                case(gewei)
                    4'd0: begin
                        seg = `data_0;
                    end
                    4'd1: begin
                        seg = `data_1;
                    end
                    4'd2: begin
                        seg = `data_2;
                    end
                    4'd3: begin
                        seg = `data_3;
                    end
                    4'd4: begin
                        seg = `data_4;
                    end
                    4'd5: begin
                        seg = `data_5;
                    end
                    4'd6: begin
                        seg = `data_6;
                    end
                    4'd7: begin
                        seg = `data_7;
                    end
                    4'd8: begin
                        seg = `data_8;
                    end
                    4'd9: begin
                        seg = `data_9;
                    end
                    4'd10: begin
                        seg = `data_a;
                    end
                    4'd11: begin
                        seg = `data_b;
                    end
                    4'd12: begin
                        seg = `data_c;
                    end
                    4'd13: begin
                        seg = `data_d;
                    end
                    4'd14: begin
                        seg = `data_e;
                    end
                    4'd15: begin
                        seg = `data_f;
                    end
                endcase
            end
            else begin
                seg = `data_0;
            end
        end
    end
endmodule
```

分频模块：

```
`include "defines.v"
module fenpin(
    input wire      clk,
    input wire      rst,
    output  reg             clk_1
```

```
);
	parameter   T1s   = 24_999_999;
	integer             cnt;
	always@ (posedge clk) begin
		if(rst == `RstEnable) begin
			cnt    <= 0;
			clk_1 <= 0;
		end
		if(cnt == T1s) begin
			clk_1 <= ~clk_1;
			cnt <= 0;
		end else begin
			cnt <= cnt + 1;
		end
	end
endmodule
```

openmips_spoc 模块：

```
`include "defines.v"
module openmips_spoc(
	input wire  clk,
	input wire  rst,
	//来自外部输入的数据
	input wire[`RegBus]    data_show_i,
	//来自外部的取消暂停信号
	input wire                  enter,
	//送到外部输出的信息
	output      wire[`RegBus]    data_show_o,
	output      wire                out_o,
	output      wire[`InstAddrBus]     pc
);
	//连接指令存储器 ROM
	wire[`InstAddrBus]     inst_addr;
	wire[`InstBus]             inst;
	wire                        rom_ce;
	//连接数据存储器 RAM
	wire                        memW_i;
	wire                        memR_i;
	wire[`RegBus]              mem_addr_i;
	wire[`RegBus]              mem_data_i;
	wire[`RegBus]              mem_data_o;
	wire[3:0]                   mem_sel_i;
	wire                        mem_ce_i;
	//连接 IO 单元
	wire                        io_we_i;
	wire                        io_re_i;
	wire[`RegBus]              io_data;
	//例化处理器 Openmips
	openmips openmips0(
			.clk(clk),                  .rst(rst),
			.rom_addr_o(inst_addr),             .rom_data_i(inst),
			.rom_ce_o(rom_ce),
			.ram_data_i(mem_data_o),   .ram_addr_o(mem_addr_i),
			.ram_data_o(mem_data_i),   .ram_we_o(memW_i),
			.ram_re_o(memR_i),                  .ram_sel_o(mem_sel_i),
			.ram_ce_o(mem_ce_i),
			.data_i(data_show_i),          .data_o(io_data),
			.io_we_o(io_we_i),                  .io_re_o(io_re_i),
			.enter_i(enter)
	);
	//例化指令存储器 ROM
	inst_rom inst_rom0(
			.ce(rom_ce),
			.addr(inst_addr),               .inst(inst)
	);
	//例化数据寄存器 RAM
	data_ram data_ram0(
			.clk(clk),                              .ce(mem_ce_i),
			.we(memW_i),                            .re(memR_i),
```

```
                .addr(mem_addr_i),                  .sel(mem_sel_i),
                .data_i(mem_data_i),          .data_o(mem_data_o)
        );
        //例化 IO 单元
        io io0(
                .clk(clk),                          .rst(rst),
                .we(io_we_i),                       .re(io_re_i),
                .data_i(io_data),
                .data_show_i(data_show_i),
                .out(out_o)
        );
        assign data_show_o = io_data;
        assign pc = inst_addr;
endmodule
```

io 模块：

```
`include "defines.v"
module io(
        input wire              rst,
        input wire              clk,
        input wire              we,
        input wire              re,
        //来自 OUT 输出的数据
        input wire[`RegBus]     data_i,
        //来自外部输入的数据
        input wire[`RegBus]     data_show_i,
        //输入给 IN 指令的信息
        output      reg[`RegBus]    data_o,
        //输出到外部的信息
        output      reg[`RegBus]    data_show_o,
        output      wire            out
);
        assign out  = we;
        //写操作（输出到 OUT 单元）
        always@ (posedge clk) begin
                if(rst == `RstEnable) begin
                        data_show_o     <= `ZeroWord;
                end else if(we == `WriteEnable) begin
                        data_show_o     <= data_i;
                end
        end
        //读操作
        always@ (*) begin
                if(rst == `RstEnable) begin
                        data_o <= `ZeroWord;
                end else if(re == `ReadEnable) begin
                        data_o <= data_show_i;
                end
        end
endmodule
```

数据存储器 RAM 模块：

```
`include "defines.v"
module      data_ram(
        input wire  clk,
        input wire  ce,
        input wire  we,
        input wire  re,
        input wire[`DataAddrBus]    addr,
        input wire[3:0]             sel,
        input wire[`DataBus]        data_i,
        output      reg[`DataBus]           data_o
);
        //定义四个字节数组
        reg[`ByteWidth] data_mem0[0:`DataMemNum-1];
        reg[`ByteWidth] data_mem1[0:`DataMemNum-1];
        reg[`ByteWidth] data_mem2[0:`DataMemNum-1];
        reg[`ByteWidth] data_mem3[0:`DataMemNum-1];
        //写操作
```

```
        always@ (posedge clk) begin
            if(ce == `ChipDisable) begin
                //data_o <= `ZeroWord;
            end else if(we == `WriteEnable) begin
                if(sel[3] == 1'b1) begin
                    data_mem3[addr[`DataMemNumLog2+1:2]] <= data_i[31:24];
                end
                if(sel[2] == 1'b1) begin
                    data_mem2[addr[`DataMemNumLog2+1:2]] <= data_i[23:16];
                end
                if(sel[1] == 1'b1) begin
                    data_mem1[addr[`DataMemNumLog2+1:2]] <= data_i[15:8];
                end
                if(sel[0] == 1'b1) begin
                    data_mem0[addr[`DataMemNumLog2+1:2]] <= data_i[7:0];
                end
            end
        end
        //读操作
        always@ (*) begin
            if(ce == `ChipDisable) begin
                data_o <= `ZeroWord;
            end else if(re == `ReadEnable) begin
                data_o <= {data_mem3[addr[`DataMemNumLog2+1:2]],
                           data_mem2[addr[`DataMemNumLog2+1:2]],
                           data_mem1[addr[`DataMemNumLog2+1:2]],
                           data_mem0[addr[`DataMemNumLog2+1:2]]};
            end else begin
                data_o <= `ZeroWord;
            end
        end
    endmodule
```

指令存储器 ROM 模块：

```
    `include "defines.v"
    module inst_rom(
        input wire                      ce,
        input wire[`InstAddrBus]    addr,
        output      reg[`InstBus]           inst
    );
        reg[`InstBus]     inst_mem[0:50];
        reg[4:0] n;
        initial begin
            inst_mem[0] = 32'h04222821;
            inst_mem[1] = 32'h04A33822;
            inst_mem[2] = 32'h04434025;
            inst_mem[3] = 32'h04224824;
            inst_mem[4] = 32'h04445023;
            inst_mem[5] = 32'h0C2B0001;
            inst_mem[6] = 32'h10220004;
            inst_mem[7] = 32'h00000000;
            inst_mem[8] = 32'h00000000;
            inst_mem[9] = 32'h00000000;
            inst_mem[10] = 32'h00000000;
            inst_mem[11] = 32'h2C270001;
            inst_mem[12] = 32'h05676021;
            inst_mem[13] = 32'h00000000;
            inst_mem[14] = 32'h1C410001;
            inst_mem[15] = 32'h14000014;
            inst_mem[16] = 32'h2C2C0001;
            inst_mem[17] = 32'h00000000;
            inst_mem[18] = 32'h00000000;
            inst_mem[19] = 32'h00000000;
            inst_mem[20] = 32'h2C2B0001;
            inst_mem[21] = 32'h14000000;
        end
        always@ (*) begin
            if(ce == `ChipDisable) begin
                inst <= `ZeroWord;
```

```
            end else begin
                inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
            end
        end
endmodule
```

OpenMIPS 模块：

```
`include "defines.v"
module openmips(
    input wire          clk,
    input wire          rst,
    input wire[`RegBus]    rom_data_i,
    output      wire[`RegBus]    rom_addr_o,
    output      wire             rom_ce_o,
    //连接到 IO 单元
    input wire[`RegBus]    data_i,
    output      wire[`RegBus]    data_o,
    output      wire             io_we_o,
    output      wire             io_re_o,
    //来自外部的取消暂停信号
    input wire              enter_i,
    //连接数据存储器 RAM
    input wire[`RegBus]    ram_data_i,
    output      wire[`RegBus]    ram_addr_o,
    output      wire[`RegBus]    ram_data_o,
    output      wire             ram_we_o,
    output      wire             ram_re_o,
    output      wire[3:0]        ram_sel_o,
    output      wire             ram_ce_o
);
    //连接 IF/ID 模块与译码阶段 ID 模块的变量
    wire[`InstAddrBus]          pc;
    wire[`InstAddrBus]          id_pc_i;
    wire[`InstBus]              id_inst_i;
    //连接译码阶段 ID 模块输出与 ID/EX 模块输入的变量
    wire[`AluOpBus]             id_aluop_o;
    wire[`AluSelBus]        id_alusel_o;
    wire[`RegBus]           id_reg1_o;
    wire[`RegBus]           id_reg2_o;
    wire                    id_wreg_o;
    wire[`RegAddrBus]       id_wd_o;
    wire[`RegBus]           id_immediate_o;
    wire                    id_alusrc_o;
    wire                    id_memW_o;
    wire                    id_memR_o;
    wire                    id_in_o;
    wire                    id_out_o;
    //连接译码阶段 ID 模块与 PC 模块的输入的变量
    wire                    id_branch_o;
    wire[`RegBus]           id_pc_o;
    //连接 ID/EX 模块输出与执行阶段 EX 模块的输入的变量
    wire[`AluOpBus]             ex_aluop_i;
    wire[`AluSelBus]        ex_alusel_i;
    wire[`RegBus]           ex_reg1_i;
    wire[`RegBus]           ex_reg2_i;
    wire                    ex_wreg_i;
    wire[`RegAddrBus]       ex_wd_i;
    wire[`RegBus]           ex_immediate_i;
    wire                    ex_alusrc_i;
    wire                    ex_memW_i;
    wire                    ex_memR_i;
    wire                    ex_in_i;
    wire                    ex_out_i;
    //连接执行阶段 EX 模块的输出与 EX/MEM 模块的输入的变量
    wire                    ex_wreg_o;
    wire[`RegAddrBus]       ex_wd_o;
    wire[`RegBus]           ex_wdata_o;
    wire[`RegBus]           ex_mem_addr_o;
    wire                    ex_memW_o;
```

```
wire                    ex_memR_o;
wire                    ex_in_o;
wire                    ex_out_o;
//连接 EX/MEM 模块的输出与访存阶段 MEM 模块的输入变量
wire                    mem_wreg_i;
wire[`RegAddrBus]       mem_wd_i;
wire[`RegBus]           mem_wdata_i;
wire[`RegBus]           mem_addr_i;
wire                    memW_i;
wire                    memR_i;
wire                    mem_in_i;
wire                    mem_out_i;
//连接访存阶段 MEM 模块的输出与 MEM/WB 模块的输入变量
wire                    mem_wreg_o;
wire[`RegAddrBus]       mem_wd_o;
wire[`RegBus]           mem_wdata_o;
//连接 MEM/WB 模块的输出与回写阶段的输入变量
wire                    wb_wreg_i;
wire[`RegAddrBus]       wb_wd_i;
wire[`RegBus]           wb_wdata_i;
//连接译码阶段 ID 模块与通用寄存器 Regfile 模块的变量
wire                    reg1_read;
wire                    reg2_read;
wire[`RegBus]           reg1_data;
wire[`RegBus]           reg2_data;
wire[`RegAddrBus]       reg1_addr;
wire[`RegAddrBus]       reg2_addr;
//连接 CTRL 模块与各个流水级之间的变量
wire[5:0]               stall_out;
//例化 CTRL 模块
ctrl ctrl0(
        .rst(rst),
        .stallreq_from_mem(mem_in_i),
        .enter(enter_i),
        .stall(stall_out)
);
//PC_reg 例化
pc_reg pc_reg0(
        .clk(clk),    .rst(rst),     .pc(pc),      .ce(rom_ce_o),
        .branch_i(id_branch_o),              .pc_i(id_pc_o),
        //从模块传递过来的信息
        .stall(stall_out)
);
assign      rom_addr_o = pc;            //指令存储器的输入地址就是 pc 的值
//IF/ID 模块例化
if_id if_id0(
        .clk(clk),    .rst(rst),     .if_pc(pc),
        .if_inst(rom_data_i),    .id_pc(id_pc_i),
        .id_inst(id_inst_i),
        //从模块传递过来的信息
        .stall(stall_out)
);
//译码阶段 ID 模块例化
id id0(
        .rst(rst),     .pc_i(id_pc_i),    .inst_i(id_inst_i),
        //来自 Regfile 模块的输入
        .reg1_data_i(reg1_data),      .reg2_data_i(reg2_data),
        //送到 Regfile 模块的信息
        .reg1_read_o(reg1_read),      .reg2_read_o(reg2_read),
        .reg1_addr_o(reg1_addr),      .reg2_addr_o(reg2_addr),
        //处于执行阶段的指令运算结果、及指令类型
        .ex_wreg_i(ex_wreg_o),              .ex_wdata_i(ex_wdata_o),
        .ex_wd_i(ex_wd_o),
        //处于访存阶段的指令运算结果
        .mem_wreg_i(mem_wreg_o),      .mem_wdata_i(mem_wdata_o),
        .mem_wd_i(mem_wd_o),
        //送到 ID/EX 模块的信息
        .aluop_o(id_aluop_o),         .alusel_o(id_alusel_o),
        .reg1_o(id_reg1_o),                 .reg2_o(id_reg2_o),
```

```
        .wd_o(id_wd_o),                 .wreg_o(id_wreg_o),
        .immediate(id_immediate_o),     .alusrc_o(id_alusrc_o),
        .in_o(id_in_o),                 .out_o(id_out_o),
        //送到访存阶段的信息
        .memW_o(id_memW_o),                 .memR_o(id_memR_o),
        //送到 PC 模块的信息
        .branch_o(id_branch_o),         .pc_o(id_pc_o)
);
//通用寄存器 Regfile 模块例化
regfile regfile0(
        .clk(clk),                      .rst(rst),
        .we(wb_wreg_i),                 .waddr(wb_wd_i),
        .wdata(wb_wdata_i),             .re1(reg1_read),
        .raddr1(reg1_addr),             .rdata1(reg1_data),
        .re2(reg2_read),            .raddr2(reg2_addr),
        .rdata2(reg2_data)
);
//ID/EX 模块例化
id_ex id_ex0(
        .clk(clk),                              .rst(rst),
        //从译码阶段 ID 模块传递过来的信息
        .id_aluop(id_aluop_o),          .id_alusel(id_alusel_o),
        .id_reg1(id_reg1_o),            .id_reg2(id_reg2_o),
        .id_wd(id_wd_o),                        .id_wreg(id_wreg_o),
        .id_imm(id_immediate_o),        .id_alusrc(id_alusrc_o),
        .id_memW(id_memW_o),                    .id_memR(id_memR_o),
        .id_in(id_in_o),                .id_out(id_out_o),
        //从模块传递过来的信息
        .stall(stall_out),
        //传递到执行阶段 EX 模块的信息
        .ex_aluop(ex_aluop_i),          .ex_alusel(ex_alusel_i),
        .ex_reg1(ex_reg1_i),            .ex_reg2(ex_reg2_i),
        .ex_wd(ex_wd_i),                        .ex_imm(ex_immediate_i),
        .ex_alusrc(ex_alusrc_i),        .ex_wreg(ex_wreg_i),
        .ex_memW(ex_memW_i),                    .ex_memR(ex_memR_i),
        .ex_in(ex_in_i),                .ex_out(ex_out_i)
);
//EX 模块例化
ex ex0(
        .rst(rst),
        //从 ID/EX 模块传递过来的值
        .aluop_i(ex_aluop_i),           .alusel_i(ex_alusel_i),
        .reg1_i(ex_reg1_i),                     .reg2_i(ex_reg2_i),
        .wd_i(ex_wd_i),                         .imm_i(ex_immediate_i),
        .wreg_i(ex_wreg_i),                     .alusrc_i(ex_alusrc_i),
        .memW_i(ex_memW_i),                         .memR_i(ex_memR_i),
        .in_i(ex_in_i),                         .out_i(ex_out_i),
        //输出到 EX/MEM 模块的信息
        .wd_o(ex_wd_o),                             .wreg_o(ex_wreg_o),
        .wdata_o(ex_wdata_o),                   .mem_addr_o(ex_mem_addr_o),
        .memW_o(ex_memW_o),                         .memR_o(ex_memR_o),
        .in_o(ex_in_o),                         .out_o(ex_out_o)
);
//EX/MEM 模块例化
ex_mem ex_mem0(
        .clk(clk),                              .rst(rst),
        //来自执行阶段 EX 模块的信息
        .ex_wd(ex_wd_o),                        .ex_wreg(ex_wreg_o),
        .ex_wdata(ex_wdata_o),                  .ex_memW(ex_memW_o),
        .ex_memR(ex_memR_o),                    .ex_addr(ex_mem_addr_o),
        .ex_in(ex_in_o),                .ex_out(ex_out_o),
        //来自控制模块的信息
        .stall(stall_out),
        //送到访存阶段 MEM 模块的信息
        .mem_wd(mem_wd_i),                          .mem_wreg(mem_wreg_i),
        .mem_wdata(mem_wdata_i),                .mem_W(memW_i),
        .mem_R(memR_i),                             .mem_addr(mem_addr_i),
        .mem_in(mem_in_i),                      .mem_out(mem_out_i)
);
```

```
	//MEM 模块例化
	mem mem0(
		.rst(rst),
		//来自 EX/MEM 模块的信息
		.wd_i(mem_wd_i),			.wreg_i(mem_wreg_i),
		.wdata_i(mem_wdata_i),		.memW_i(memW_i),
		.memR_i(memR_i),			.mem_addr_i(mem_addr_i),
		.mem_in(mem_in_i),			.mem_out(mem_out_i),
		//送到 MEM/WB 模块的信息
		.wd_o(mem_wd_o),			.wreg_o(mem_wreg_o),
		.wdata_o(mem_wdata_o),
		//来自于 IN 单元的数据
		.data_in(data_i),
		//送到 OUT 单元的数据
		.data_out(data_o),			.io_we(io_we_o),
		.io_re(io_re_o),
		//来自数据存储器的信息
		.mem_data_i(ram_data_i),
		//送到数据寄存器的信息
		.mem_addr_o(ram_addr_o),	.mem_we_o(ram_we_o),
		.mem_re_o(ram_re_o),		.mem_sel_o(ram_sel_o),
		.mem_data_o(ram_data_o),	.mem_ce_o(ram_ce_o)
	);
	//MEM/WB 模块例化
	mem_wb mem_wb(
		.clk(clk),				.rst(rst),
		//来自访存阶段 MEM 模块的信息
		.mem_wd(mem_wd_o),			.mem_wreg(mem_wreg_o),
		.mem_wdata(mem_wdata_o),
		//来自控制模块的信息
		.stall(stall_out),
		//送到回写阶段的信息
		.wb_wd(wb_wd_i),			.wb_wreg(wb_wreg_i),
		.wb_wdata(wb_wdata_i)
	);
endmodule

PC 模块：
`include "defines.v"
module pc_reg(
	input	wire		clk,
	input wire		rst,
	//来自于译码阶段的地址转移信息
	input wire		branch_i,
	input wire[`RegBus]	pc_i,
	//来自控制模块 CTRL
	input wire[5:0]		stall,
	output	reg		ce,
	output	reg[`InstAddrBus]	pc
);
	always@ (posedge clk) begin
		if (rst == `RstEnable) begin
			ce <= `ChipDisable;		//复位的时候指令存储器禁用
			//cnt  = 0;
		end   else begin
			ce <= `ChipEnable;		//复位结束后，指令存储器使能
		end
	end
	always@ (posedge clk) begin
		if (ce == `ChipDisable) begin
			pc <= `ZeroWord;		//指令存储器禁用的时候的，pc 为 0
		end
		else if(stall[0] == `NoStop) begin
			if(branch_i == `Branch) begin
				pc <= pc_i;
			end else begin
				pc <= pc + 4'h4;
			end
		end
```

```
    end
endmodule
```

IF/ID 模块：

```
`include "defines.v"
module if_id(
    input wire  clk,
    input wire  rst,
    //来自于取指阶段的信号
    input       wire[`InstAddrBus]    if_pc,
    input wire[`InstBus]          if_inst,
    //来自于 CTRL 模块的信号
    input wire[5:0]               stall,
    //对应译码阶段的信号
    output      reg[`InstAddrBus]     id_pc,
    output      reg[`InstBus]         id_inst
);
    always@   (posedge clk) begin
        if (rst == `RstEnable) begin
            id_pc <= `ZeroWord;
            id_inst <= `ZeroWord;
        end else if(stall[1] == `Stop && stall[2] == `NoStop) begin      //暂停取指阶段，后续阶段继续执行
            id_pc<= `ZeroWord;
            id_inst      <= `ZeroWord;
        end else if(stall[1] == `NoStop) begin
            id_pc<= if_pc;
            id_inst      <= if_inst;
        end
    end
endmodule
```

id 模块：

```
`include "defines.v"
module id(
    input wire  rst,
    input wire[`InstAddrBus]          pc_i,
    input wire[`InstBus]              inst_i,
    //处于执行阶段的指令运算结果、以及指令类型
    input wire                        ex_wreg_i,
    input wire[`RegBus]               ex_wdata_i,
    input wire[`RegAddrBus]           ex_wd_i,
    //处于访存阶段的指令的运算结果
    input wire                        mem_wreg_i,
    input wire[`RegBus]               mem_wdata_i,
    input wire[`RegAddrBus]           mem_wd_i,
    //读取的 Regfile 的值
    input wire[`RegBus]               reg1_data_i,
    input wire[`RegBus]               reg2_data_i,
    //输出到 Regfile 的信息
    output      reg                         reg1_read_o,
    output      reg                         reg2_read_o,
    output      reg[`RegAddrBus]        reg1_addr_o,
    output      reg[`RegAddrBus]        reg2_addr_o,
    //送到执行阶段的信息
    output      reg[`AluOpBus]          aluop_o,
    output      reg[`AluSelBus]         alusel_o,
    output      reg[`RegBus]            reg1_o,
    output      reg[`RegBus]            reg2_o,
    output      reg[`RegAddrBus]        wd_o,
    output      reg[`RegBus]            immediate,
    output      reg                         alusrc_o,
    output      reg                         wreg_o,
    //送到访存阶段的信息
    output      reg                         memW_o,
    output      reg                         memR_o,
    output      reg                         in_o,
    output      reg                         out_o,
    //送到 PC 模块的信息
    output      reg                         branch_o,
```

```
    output      reg[`RegBus]                pc_o
);
    //取指令中的指令码、功能码
    wire[5:0] op    = inst_i[31:26];        //指令码
    wire[4:0] op2 = inst_i[10:6];       //取 sa 字段
    wire[5:0] op3 = inst_i[5:0];        //Func 字段
    wire[4:0] op4 = inst_i[20:16];          //取目的寄存器地址
    wire[15:0] imm = inst_i[15:0];          //取立即数字段
    wire[`RegBus]   imm_sll2;               //立即数左移 2
    wire[`RegBus]   pc_plus_4;
    assign      pc_plus_4 = pc_i + 4;
    assign      imm_sll2 = {14'b0,inst_i[15:0],2'b00};
    //表示指令是否有效
    reg instvalid;
    //一、对指令进行译码
    always@ (*) begin
        if(rst == `RstEnable) begin
            aluop_o         <= `EXE_nop_op;
            alusel_o    <= `EXE_NOP_sel;
            wd_o        <= `NOPRegAddr;
            wreg_o          <= `WriteDisable;
            instvalid   <= `InstInvalid;
            reg1_read_o     <= `ReadDisable;
            reg2_read_o     <= `ReadDisable;
            reg1_addr_o     <= `NOPRegAddr;
            reg2_addr_o     <= `NOPRegAddr;
            alusrc_o    <= 1'b0;
            immediate   <= 32'h0;
            memW_o          <= `WriteDisable;
            memR_o          <= `ReadDisable;
            branch_o    <= `NotBranch;
            pc_o        <= `ZeroWord;
            in_o        <= `NotIn;
            out_o       <= `NotOut;
        end   else begin
            aluop_o         <= `EXE_nop_op;
            alusel_o    <= `EXE_NOP_sel;
            wd_o        <=   inst_i[15:11];         //默认写入 15:11 位表示寄存器地址
            wreg_o          <= `WriteDisable;
            instvalid   <= `InstInvalid;
            reg1_read_o     <= `ReadDisable;
            reg2_read_o     <= `ReadDisable;
            reg1_addr_o     <=   inst_i[25:21];         //默认通过 Regfile 读端口 1 读取的寄存器地址
            reg2_addr_o     <=   inst_i[20:16];         //默认通过 Regfile 读端口 2 读取的寄存器地址
            alusrc_o    <= 1'b0;
            immediate   <= `ZeroWord;
            memW_o      <= `WriteDisable;
            memR_o      <= `ReadDisable;
            branch_o    <= `NotBranch;
            pc_o        <= `ZeroWord;
            in_o        <= `NotIn;                  //默认不需要 IN 单元
            out_o       <= `NotOut;                     //默认不需要 OUT 单元
            case(op)
                `EXE_R_type:    begin           //根据 op 的值判断是否是 R-type 指令
                case(op2)
                    5'b00000:   begin
                        case(op3)               //根据功能码判断是哪种指令
                            `EXE_ADD:begin          //add 指令
                                //运算子类型为
                                alusel_o    <= `EXE_ADD_sel;
                                //R-type 指令 需要将结果写入目的寄存器，所以 wreg_o 为
WriteEnable
                                wreg_o          <= `WriteEnable;
                                //运算类型为 R-type 运算
                                aluop_o         <= `EXE_arithmetic_op;
                                //需要通过 Regfile 的读端口 1 读取寄存器
                                reg1_read_o     <= `ReadEnable;
                                //需要通过 Regfile 的读端口 2 读取寄存器
                                reg2_read_o     <= `ReadEnable;
```

```
                        //指令执行要写的目的寄存器地址
                        wd_o        <= inst_i[15:11];
                        //R-type 指令是有效指令
                        instvalid   <= `InstValid;
                        //R-type 指令不需要立即数
                        immediate <= `ZeroWord;
                        //R-type 指令选择寄存器 2 中的值
                        alusrc_o    <= 1'b0;
                    end
                    `EXE_SUB:begin
                        alusel_o    <= `EXE_SUB_sel;
                        wreg_o          <= `WriteEnable;
                        aluop_o         <= `EXE_arithmetic_op;
                        reg1_read_o     <= `ReadEnable;
                        reg2_read_o     <= `ReadEnable;
                        wd_o        <= inst_i[15:11];
                        instvalid   <= `InstValid;
                        immediate <= `ZeroWord;
                        alusrc_o    <= 1'b0;
                    end
                    `EXE_OR:begin
                        alusel_o    <= `EXE_OR_sel;
                        wreg_o          <= `WriteEnable;
                        aluop_o         <= `EXE_arithmetic_op;
                        reg1_read_o     <= `ReadEnable;
                        reg2_read_o     <= `ReadEnable;
                        wd_o        <= inst_i[15:11];
                        instvalid   <= `InstValid;
                        immediate <= `ZeroWord;
                        alusrc_o    <= 1'b0;
                    end
                    `EXE_AND:begin
                        alusel_o    <= `EXE_AND_sel;
                        wreg_o          <= `WriteEnable;
                        aluop_o         <= `EXE_arithmetic_op;
                        reg1_read_o     <= `ReadEnable;
                        reg2_read_o     <= `ReadEnable;
                        wd_o        <= inst_i[15:11];
                        instvalid   <= `InstValid;
                        immediate <= `ZeroWord;
                        alusrc_o    <= 1'b0;
                    end
                    `EXE_SLT:begin
                        alusel_o    <= `EXE_SLT_sel;
                        wreg_o          <= `WriteEnable;
                        aluop_o         <= `EXE_arithmetic_op;
                        reg1_read_o     <= `ReadEnable;
                        reg2_read_o     <= `ReadEnable;
                        wd_o        <= inst_i[15:11];
                        instvalid   <= `InstValid;
                        immediate <= `ZeroWord;
                        alusrc_o    <= 1'b0;
                    end
                    default:begin
                    end
                    endcase
                end
            default:begin
            end
            endcase
        end
        `EXE_lw_m:      begin
            //lw_m 指令  需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteEnable
            wreg_o          <= `WriteEnable;
            //运算类型为 I-type 运算
            aluop_o         <= `EXE_ls_op;
            alusel_o    <= `EXE_ADD_sel;
            //需要通过 Regfile 的读端口 1 读取寄存器，作为寻址地址中的基值
            reg1_read_o     <= `ReadEnable;
```

```
        //不需要通过 Regfile 的读端口 2 读取寄存器
        reg2_read_o     <= `ReadDisable;
        //指令执行要写的目的寄存器地址
        wd_o        <= inst_i[20:16];
        //lw_m 指令是有效指令
        instvalid   <= `InstValid;
        //lw_m 指令需要立即数
        immediate <= inst_i[15:0];
        //lw_m 指令选择立即数，作为寻址地址中的偏移量
        alusrc_o    <= 1'b1;
        //lw_m 型指令需要访存。lw_m 不需要往存储单元写入数据，需要读出数据
        memW_o  <= `WriteDisable;
        memR_o  <= `ReadEnable;
    end
    `EXE_sw_m:      begin
        //sw_m 指令  不需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteDisable
        wreg_o          <= `WriteDisable;
        //运算类型为 I-type 运算
        aluop_o         <= `EXE_ls_op;
        alusel_o    <= `EXE_ADD_sel;
        //需要通过 Regfile 的读端口 1 读取寄存器，作为寻址地址中的基值
        reg1_read_o     <= `ReadEnable;
        //需要通过 Regfile 的读端口 2 读取寄存器，读出要保存的数
        reg2_read_o     <= `ReadEnable;
        //指令执行要写的目的寄存器地址，sw_m 不需要要写入的目的寄存器的地址
        wd_o        <= `NOPRegAddr;
        //sw_m 指令是有效指令
        instvalid   <= `InstValid;
        //sw_m 指令需要立即数
        immediate <= inst_i[15:0];
        //sw_m 指令选择立即数
        alusrc_o    <= 1'b1;
        //sw_m 指令需要访存。sw_m 需要往存储单元写入数据，不需要读出数据
        memW_o  <= `WriteEnable;
        memR_o  <= `ReadDisable;
        branch_o    <= `NotBranch;
    end
    `EXE_beq: begin
        //beq 指令  不需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteDisable
        wreg_o          <= `WriteDisable;
        //运算类型为无运算类型
        aluop_o         <= `EXE_nop_op;
        alusel_o    <= `EXE_NOP_sel;
        //需要通过 Regfile 的读端口 1 读取寄存器，作为转移判断中第一个数
        reg1_read_o     <= `ReadEnable;
        //需要通过 Regfile 的读端口 2 读取寄存器，作为转移判断中第二个数
        reg2_read_o     <= `ReadEnable;
        //指令执行要写的目的寄存器地址，beq 不需要要写入的目的寄存器的地址
        wd_o        <= `NOPRegAddr;
        //beq 指令是有效指令
        instvalid   <= `InstValid;
        //beq 指令不需要将立即数向后传递
        immediate <= `ZeroWord;
        alusrc_o    <= 1'b1;
        if(reg1_o == reg2_o) begin
            branch_o    <= `Branch;
            pc_o        <= pc_plus_4 + imm_sll2;
        end
    end
    `EXE_j: begin
        //j 指令  不需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteDisable
        wreg_o          <= `WriteDisable;
        //运算类型为无运算类型
        aluop_o         <= `EXE_nop_op;
        alusel_o    <= `EXE_NOP_sel;
        //不需要通过 Regfile 的读端口 1 读取寄存器
        reg1_read_o     <= `ReadDisable;
        //不需要通过 Regfile 的读端口 2 读取寄存器
        reg2_read_o     <= `ReadDisable;
```

```
                    //指令执行要写的目的寄存器地址，j 不需要要写入的目的寄存器的地址
                    wd_o        <= `NOPRegAddr;
                    //j 指令是有效指令
                    instvalid   <= `InstValid;
                    //j 指令不需要将立即数向后传递
                    immediate <= `ZeroWord;
                    //j 指令选择立即数
                    alusrc_o    <= 1'b1;
                    branch_o    <= `Branch;
                    pc_o        <= {pc_plus_4[31:28],inst_i[25:0],2'b00};
                end
                `EXE_lw_io: begin
                    //lw_io 指令  需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteEnable
                    wreg_o          <= `WriteEnable;
                    //运算类型为无运算
                    aluop_o         <= `EXE_ls_op;
                    alusel_o    <= `EXE_NOP_sel;
                    //不需要通过 Regfile 的读端口 1 读取寄存器，作为寻址地址中的基值
                    reg1_read_o     <= `ReadDisable;
                    //不需要通过 Regfile 的读端口 2 读取寄存器
                    reg2_read_o     <= `ReadDisable;
                    //指令执行要写的目的寄存器地址
                    wd_o        <= inst_i[20:16];
                    //lw_io 指令是有效指令
                    instvalid   <= `InstValid;
                    //lw_io 指令不需要立即数
                    immediate <= `ZeroWord;
                    //lw_io 指令不选择立即数，作为寻址地址中的偏移量
                    alusrc_o    <= 1'b1;
                    //lw_io 指令需要访存。lw_io 不需要往存储单元写入数据，不需要读出数据
                    memW_o          <= `WriteDisable;
                    memR_o          <= `ReadDisable;
                    //iw_io 指令需要 IO 单元。将 IN 单元的数字直接输入到寄存器中
                    in_o        <= `In;
                    out_o       <= `NotOut;
                end
                `EXE_sw_io: begin
                    //sw_io 指令  不需要将读出结果写入目的寄存器，所以 wreg_o 为 WriteDisable
                    wreg_o          <= `WriteDisable;
                    //运算类型为无运算
                    aluop_o         <= `EXE_ls_op;
                    alusel_o    <= `EXE_NOP_sel;
                    //不需要通过 Regfile 的读端口 1 读取寄存器，作为寻址地址中的基值
                    reg1_read_o     <= `ReadDisable;
                    //需要通过 Regfile 的读端口 2 读取寄存器，作为要输出的数据
                    reg2_read_o     <= `ReadEnable;
                    //指令执行要写的目的寄存器地址
                    wd_o        <= `NOPRegAddr;
                    //sw_io 指令是有效指令
                    instvalid   <= `InstValid;
                    //sw_io 指令不需要立即数
                    immediate <= `ZeroWord;
                    //sw_io 指令不选择立即数，作为寻址地址中的偏移量
                    alusrc_o    <= 1'b1;
                    //sw_io 指令需要访存。sw_m 不需要往存储单元写入数据，不需要读出数据
                    memW_o          <= `WriteDisable;
                    memR_o          <= `ReadDisable;
                    //sw_io 指令需要 IO 单元。将寄存器中的数据直接输出 OUT 单元
                    in_o        <= `NotIn;
                    out_o       <= `Out;
                end
            default:begin
            end
            endcase
        end
    end
    //确定进行运算的源操作数 1
    always@ (*) begin
        if(rst == `RstEnable) begin
```

```
                reg1_o      <= `ZeroWord;
            end
            else if((reg1_read_o == `ReadEnable) && (ex_wreg_i == `WriteEnable)
                        && (ex_wd_i == reg1_addr_o)) begin
                reg1_o <= ex_wdata_i;
            end
            else if((reg1_read_o == `ReadEnable) && (mem_wreg_i == `WriteEnable)
                        && (mem_wd_i == reg1_addr_o)) begin
                reg1_o <= mem_wdata_i;
            end
            else if(reg1_read_o == `ReadEnable) begin
                reg1_o      <=    reg1_data_i;      //Regfile 读端口 1 的输出值
            end
            else begin
                reg1_o      <= `ZeroWord;
            end
        end
        //确定进行运算的源操作数 2
        always@ (*) begin
            if(rst == `RstEnable) begin
                reg2_o      <= `ZeroWord;
            end
            else if((reg2_o == `ReadEnable) && (ex_wreg_i == `WriteEnable)
                        && (ex_wd_i == reg2_addr_o)) begin
                reg2_o <= ex_wdata_i;
            end
            else if((reg2_o == `ReadEnable) && (mem_wreg_i == `WriteEnable)
                        && (mem_wd_i == reg2_addr_o)) begin
                reg2_o <= mem_wdata_i;
            end
            else if(reg2_read_o == `ReadEnable) begin
                reg2_o      <= reg2_data_i;
            end
            else begin
                reg2_o      <= `ZeroWord;
            end
        end
endmodule
```

通用寄存器 regfile 模块：

```
`include "defines.v"
module regfile(
        input wire  clk,
        input wire  rst,
        //写端口
        input wire                      we,             //写使能
        input wire[`RegAddrBus]     waddr,          //写入寄存器的地址
        input wire[`RegBus]         wdata,          //写入的数据
        //读端口 1
        input wire                      re1,        //端口 1 的读使能
        input wire[`RegAddrBus]     raddr1,         //端口 1 的地址
        output      reg[`RegBus]            rdata1,             //端口 1 的数据
        //读端口 2
        input wire                      re2,        //端口 2 的读使能
        input wire[`RegAddrBus]     raddr2,         //端口 2 的地址
        output      reg[`RegBus]            rdata2              //端口 2 的数据
);
        //定义 32 个 32 个寄存器
        reg[`RegBus]     regs[0:`RegNum-1];
        initialbegin
            regs[1] = 32'd5;
            regs[2] = 32'd5;
            regs[3] = 32'd6;
            regs[4] = 32'd7;
            regs[5] = 32'd8;
        end
        //写操作
        always@ (posedge clk) begin
            if(rst == `RstDisable) begin
```

```
                if((we == `WriteEnable) && (waddr != `RegNumLog2'h0)) begin
                        regs[waddr] <= wdata;
                end
        end
    end
    //读端口 1 的读操作
    always@ (*) begin
        if(rst == `RstEnable) begin
                rdata1 <= `ZeroWord;
        end else if(raddr1 == `RegNumLog2'h0) begin
                rdata1 <= `ZeroWord;
        end else if((raddr1 == waddr) && (we == `WriteEnable)
                                && (re1 == `ReadEnable)) begin
                rdata1 <= wdata;
        end else if(re1 == `ReadEnable) begin
                rdata1 <= regs[raddr1];
        end else begin
                rdata1 <= `ZeroWord;
        end
    end
    //读端口 2 的读操作
    always@ (*) begin
        if(rst == `RstEnable) begin
                rdata2 <= `ZeroWord;
        end else if(raddr2 == `RegNumLog2'h0) begin
                rdata2 <= `ZeroWord;
        end else if((raddr2 == waddr) && (we == `WriteEnable)
                                && (re2 == `ReadEnable)) begin
                rdata2 <= wdata;
        end else if(re2 == `ReadEnable) begin
                rdata2 <= regs[raddr2];
        end else begin
                rdata2 <= `ZeroWord;
        end
    end
endmodule
```

ID/EX 模块：

```
`include "defines.v"
module id_ex(
    input wire  clk,
    input wire  rst,
    //从译码阶段传递过来的信息
    input wire[`AluOpBus]       id_aluop,
    input wire[`AluSelBus]      id_alusel,
    input wire[`RegBus]         id_reg1,
    input wire[`RegBus]         id_reg2,
    input wire[`RegAddrBus]     id_wd,
    input wire[`RegBus]         id_imm,
    input wire                  id_wreg,
    input wire                  id_alusrc,
    input wire                  id_memW,
    input wire                  id_memR,
    input wire                  id_in,
    input wire                  id_out,
    //来自控制模块的信息
    input wire[5:0]             stall,
    //传递到执行阶段的信息
    output      reg[`AluOpBus]      ex_aluop,
    output      reg[`AluSelBus]     ex_alusel,
    output      reg[`RegBus]        ex_reg1,
    output      reg[`RegBus]        ex_reg2,
    output      reg[`RegAddrBus]    ex_wd,
    output      reg[`RegBus]        ex_imm,
    output      reg                 ex_alusrc,
    output      reg                 ex_wreg,
    output      reg                 ex_memW,
    output      reg                 ex_memR,
    output      reg                 ex_in,
```

```
        output      reg                             ex_out
);
        always@   (posedge clk) begin
                if(rst == `RstEnable)    begin
                        ex_aluop    <= `EXE_nop_op;
                        ex_alusel   <= `EXE_NOP_sel;
                        ex_reg1             <= `ZeroWord;
                        ex_reg2             <= `ZeroWord;
                        ex_wd               <= `NOPRegAddr;
                        ex_wreg             <= `WriteDisable;
                        ex_imm              <= 32'h0;
                        ex_alusrc   <= 1'b0;
                        ex_memW             <= `WriteDisable;
                        ex_memR             <= `ReadDisable;
                        ex_in         <= `NotIn;
                        ex_out              <= `NotOut;
                end   else if(stall[2] == `Stop && stall[3] == `NoStop) begin
                        ex_aluop    <= `EXE_nop_op;
                        ex_alusel   <= `EXE_NOP_sel;
                        ex_reg1             <= `ZeroWord;
                        ex_reg2             <= `ZeroWord;
                        ex_wd               <= `NOPRegAddr;
                        ex_wreg             <= `WriteDisable;
                        ex_imm              <= 32'h0;
                        ex_alusrc   <= 1'b0;
                        ex_memW             <= `WriteDisable;
                        ex_memR             <= `ReadDisable;
                        ex_in         <= `NotIn;
                        ex_out              <= `NotOut;
                end else if(stall[2] == `NoStop) begin
                        ex_aluop    <=    id_aluop;
                        ex_alusel   <=    id_alusel;
                        ex_reg1             <=    id_reg1;
                        ex_reg2             <=    id_reg2;
                        ex_wd               <=    id_wd;
                        ex_wreg             <=    id_wreg;
                        ex_imm              <=    id_imm;
                        ex_alusrc   <=    id_alusrc;
                        ex_memW <=    id_memW;
                        ex_memR  <=    id_memR;
                        ex_in         <=    id_in;
                        ex_out              <=    id_out;
                end
        end
endmodule
```

EX 模块：

```
`include "defines.v"
module ex(
        input wire  rst,
        //译码阶段送到执行阶段的信息
        input wire[`AluOpBus]       aluop_i,
        input wire[`AluSelBus]      alusel_i,
        input wire[`RegBus]         reg1_i,
        input wire[`RegBus]         reg2_i,
        input wire[`RegAddrBus]     wd_i,
        input wire[`RegBus]         imm_i,
        input wire                  wreg_i,
        input wire                  alusrc_i,
        input wire                  memW_i,
        input wire                  memR_i,
        input wire                  in_i,
        input wire                  out_i,
        //执行结果
        output      reg[`RegAddrBus]      wd_o,
        output      reg           wreg_o,
        output      reg[`RegBus]          wdata_o,
        output      reg[`RegBus]          mem_addr_o,
        //送到访存阶段的信息
```

```
    output      reg                     memW_o,
    output      reg                     memR_o,
    output      reg                     in_o,
    output      reg                     out_o
);
    wire            ov_sum;                 //保存溢出情况
    wire            reg1_eq_reg2;           //第一个操作数是否等于第二个操作数
    wire            reg1_lt_reg2;           //第一个操作数是否小于第二个操作数
    reg[`RegBus]    logicout;               //保存逻辑运算的结果
    reg[`RegBus]    arithmeticres;          //保存算术运算的结果
    wire[`RegBus]   reg2_i_mux;             //保存输入的第二个操作数 reg2_i 的补码
    wire[`RegBus]   result_sum;             //保存加法结果
    wire[`RegBus]   data2;                  //ALU 第二个操作数
    wire[`RegBus]   left2_data;             //立即数左移 2
    //一、计算下列变量的值
    //如果是减法或者有符号比较运算，那么 reg2_i_mux 等于第二个操作数 reg2_i 的补码，否则就等于第二个操作数 reg2_i
    assign      reg2_i_mux = ((alusel_i == `EXE_SUB_sel)    ||
                              (alusel_i == `EXE_SLT_sel))    ?
                              (~reg2_i) + 1 : reg2_i;
    //将立即数左移 2
    assign      left2_data = imm_i << 2;
    //选择 ALU 第二个操作数
    assign      data2 = (alusrc_i == 1'b1) ? imm_i : reg2_i_mux;
    //计算加法、减法的结果
    assign      result_sum = reg1_i + data2;
    //计算是否溢出，加法指令(add)和减法指令(sub)执行的时候
    assign      ov_sum = ((!reg1_i[31] && !reg2_i_mux[31]) && result_sum[31]) ||
                         ((reg1_i[31] && reg2_i_mux[31]) && (!result_sum[31]));
    //计算操作数 1 是否小于操作数 2
    assign      reg1_lt_reg2 = ((alusel_i == `EXE_SLT_sel)) ?
                               ((reg1_i[31] && !reg2_i[31]) ||
                                (!reg1_i[31] && !reg2_i[31]) && result_sum[31]) ||
                                (reg1_i[31] && reg2_i[31] && result_sum[31])
                                :(reg1_i < reg2_i);
    //二、依据不同的运算类型，给 arithmeticres、logicout 赋值
    always@ (*) begin
        case(alusel_i)
            `EXE_ADD_sel, `EXE_SUB_sel: begin
                arithmeticres <= result_sum;        //加法、减法运算
            end
            `EXE_SLT_sel: begin
                arithmeticres <= {31'b0,reg1_lt_reg2};          //比较运算
            end
            `EXE_AND_sel: begin
                logicout <= reg1_i & reg2_i;        //逻辑与操作
            end
            `EXE_OR_sel: begin
                logicout <= reg1_i | reg2_i;        //逻辑或操作
            end
            default: begin
                arithmeticres <= `ZeroWord;
                logicout <= `ZeroWord;
            end
        endcase
    end
    //三、确定要写出的值
    always@ (*) begin
        if(rst == `RstEnable) begin
            wdata_o <= `ZeroWord;
            wd_o<= `NOPRegAddr;
            memW_o   <= `WriteDisable;
            memR_o    <= `ReadDisable;
            mem_addr_o<= `ZeroWord;
            wreg_o      <= `WriteDisable;
            in_o  <= `NotIn;
            out_o<= `NotOut;
        end else begin
            memW_o   <= memW_i;
```

```
                memR_o    <= memR_i;
                in_o  <= in_i;
                out_o<= out_i;
                //如果是 add、sub 指令，结果发生溢出，那么设置 wreg_o 为 WriteDisable，不写入目的寄
存器
                if(((alusel_i == `EXE_ADD_sel) || (alusel_i == `EXE_SUB_sel)) && (ov_sum == 1'b1)) begin
                        wreg_o <= `WriteDisable;
                end else begin
                        wreg_o <= wreg_i;
                end
                case(aluop_i)
                        `EXE_arithmetic_op: begin                       //如果是 R-type 指令
                                case(alusel_i)
                                `EXE_ADD_sel, `EXE_SUB_sel, `EXE_SLT_sel: begin
                                        wdata_o <= arithmeticres;
                                        wd_o <=    wd_i;
                                        mem_addr_o <= `ZeroWord;
                                end
                                `EXE_AND_sel, `EXE_OR_sel: begin
                                        wdata_o <= logicout;
                                        wd_o <=    wd_i;
                                        mem_addr_o <= `ZeroWord;
                                end
                                default:      begin
                                        wdata_o <= `ZeroWord;
                                        wd_o <=wd_i;
                                        mem_addr_o <= `ZeroWord;
                                end
                                endcase
                        end
                        `EXE_ls_op: begin
                                mem_addr_o <= arithmeticres;
                                wdata_o <= reg2_i;
                                wd_o <= wd_i;
                        end
                endcase
        end
    end
endmodule
```

EX/MEM 模块：

```
`include "defines.v"
module ex_mem(
    input wire  clk,
    input wire  rst,
    //来自执行阶段的值
    input wire[`RegAddrBus]    ex_wd,
    input wire                 ex_wreg,
    input wire[`RegBus]        ex_wdata,
    input wire                 ex_memW,
    input wire                 ex_memR,
    input wire[`RegBus]        ex_addr,
    input wire                 ex_in,
    input wire                 ex_out,
    //来自控制模块的信息
    input wire[5:0]            stall,
    //送到访存阶段的信息
    output     reg[`RegAddrBus]    mem_wd,
    output     reg                 mem_wreg,
    output     reg[`RegBus]        mem_wdata,
    output     reg                 mem_W,
    output     reg                 mem_R,
    output     reg[`RegBus]        mem_addr,
    output     reg                 mem_in,
    output     reg                 mem_out
);
    always@ (posedge clk) begin
        if(rst == `RstEnable) begin
            mem_wd    <= `NOPRegAddr;
```

```
                mem_wreg <= `WriteDisable;
                mem_wdata <= `ZeroWord;
                mem_W    <= `WriteDisable;
                mem_R    <= `ReadDisable;
                mem_addr   <= `ZeroWord;
                mem_in    <= `NotIn;
                mem_out   <= `NotOut;
            end else if(stall[3] == `Stop && stall[4] == `NoStop) begin
                mem_wd   <= `NOPRegAddr;
                mem_wreg <= `WriteDisable;
                mem_wdata <= `ZeroWord;
                mem_W    <= `WriteDisable;
                mem_R    <= `ReadDisable;
                mem_addr   <= `ZeroWord;
                mem_in    <= `NotIn;
                mem_out   <= `NotOut;
            end else if(stall[3] == `NoStop) begin
                mem_wd   <= ex_wd;
                mem_wreg <= ex_wreg;
                mem_wdata <= ex_wdata;
                mem_W    <=    ex_memW;
                mem_R    <=    ex_memR;
                mem_addr <= ex_addr;
                mem_in    <= ex_in;
                mem_out   <= ex_out;
            end
        end
endmodule
```

MEM 模块：

```
`include "defines.v"
module mem(
        input wire  rst,
        //来自执行阶段的信息
        input wire[`RegAddrBus]     wd_i,
        input wire                  wreg_i,
        input wire[`RegBus]         wdata_i,
        input wire                  memW_i,
        input wire                  memR_i,
        input wire[`RegBus]         mem_addr_i,
        input wire                  mem_in,
        input wire                  mem_out,
        //来自于 IO 输入的数据
        input wire[`RegBus]         data_in,
        //来自外部数据寄存器 RAM 的信息
        input wire[`RegBus]         mem_data_i,
        //访存阶段的值
        output      reg[`RegAddrBus]    wd_o,
        output      reg                 wreg_o,
        output      reg[`RegBus]        wdata_o,
        //送到数据寄存器 RAM 的信息
        output      reg[`RegBus]        mem_addr_o,
        output      reg                 mem_we_o,
        output      reg                 mem_re_o,
        output      reg[3:0]            mem_sel_o,
        output      reg[`RegBus]        mem_data_o,
        output      reg                 mem_ce_o,
        //送到 OUT 单元的数据
        output      reg[`RegBus]        data_out,
        output      reg                 io_we,              //IO 单元写使能信号
        output      reg                 io_re               //IO 单元读使能信号
);
        always@ (*) begin
            if(rst == `RstEnable) begin
                wd_o <= `NOPRegAddr;
                wreg_o <= `WriteDisable;
                wdata_o <= `ZeroWord;
                mem_addr_o <= `ZeroWord;
                mem_sel_o <= 4'b0000;
```

```
                mem_data_o <= `ZeroWord;
                mem_ce_o <= `ChipDisable;
                mem_we_o <= `WriteDisable;
                mem_re_o <= `ReadDisable;
                data_out <= `ZeroWord;
                io_we    <= `WriteDisable;
                io_re <= `ReadDisable;
            end   else begin
                wd_o <= wd_i;
                wreg_o <= wreg_i;
                mem_we_o <= memW_i;
                mem_re_o <= memR_i;
                wdata_o <= `ZeroWord;
                mem_addr_o <= `ZeroWord;
                mem_sel_o <= 4'b0000;
                mem_data_o <= `ZeroWord;
                mem_ce_o <= `ChipDisable;
                data_out <= `ZeroWord;
                io_we    <= `WriteDisable;
                io_re <= `ReadDisable;
                if(memW_i == `WriteEnable || memR_i == `ReadEnable) begin
                    wdata_o <= mem_data_i;
                    mem_addr_o <= mem_addr_i;
                    mem_sel_o <= 4'b1111;
                    mem_data_o <= wdata_i;
                    mem_ce_o <= `ChipEnable;
                end else if(mem_in == `In) begin
                    wdata_o <= data_in;
                    io_we    <= `WriteDisable;
                    io_re <= `ReadEnable;
                end else if(mem_out == `Out) begin
                    data_out <= wdata_i;
                    io_we    <= `WriteEnable;
                    io_re <= `ReadDisable;
                end else begin
                    wdata_o <= wdata_i;
                    mem_addr_o <= `ZeroWord;
                    mem_sel_o <= 4'b0000;
                    mem_data_o <= `ZeroWord;
                    mem_ce_o <= `ChipDisable;
                end
            end
        end
endmodule
```

MEM/WB 模块：

```
`include "defines.v"
module mem_wb(
        input wire  clk,
        input wire  rst,
        //访存阶段的结果
        input wire[`RegAddrBus]     mem_wd,
        input wire                  mem_wreg,
        input wire[`RegBus]         mem_wdata,
        //来自控制模块的信息
        input wire[5:0]             stall,
        //送到回写阶段的信息
        output      reg[`RegAddrBus]    wb_wd,
        output      reg                 wb_wreg,
        output      reg[`RegBus]        wb_wdata
);
        always@ (posedge clk) begin
            if(rst == `RstEnable) begin
                wb_wd    <= `NOPRegAddr;
                wb_wreg  <= `WriteDisable;
                wb_wdata <= `ZeroWord;
            end   else if(stall[4] == `Stop && stall[5] == `NoStop) begin
                wb_wd    <= `NOPRegAddr;
                wb_wreg  <= `WriteDisable;
```

```
                wb_wdata <= `ZeroWord;
            end else if(stall[4] == `NoStop) begin
                wb_wd <= mem_wd;
                wb_wreg <= mem_wreg;
                wb_wdata <= mem_wdata;
            end
        end
endmodule
```

控制 CTRL 模块：

```
`include "defines.v"
module      ctrl(
        input wire          rst,
        //来自访存阶段的暂停请求
        input wire          stallreq_from_mem,
        //来自于外部的取消暂停信号
        input wire          enter,
        output      reg[5:0]     stall
);
        wire  not_enter;
        assign not_enter = ~enter;
        always@ (*) begin
            if(rst == `RstEnable) begin
                stall   <= 6'b000000;
            end
            else if(stallreq_from_mem == `Stop && not_enter == 1'b1) begin
                stall   <= 6'b011111;
            end else begin
                stall   <= 6'b000000;
            end
        end
endmodule
```

附录 3：

```
#初始值：$1 = 5; $2 = 5; $3 = 6; $4 = 7; $5 = 8;
0x00:   add $1,$2,$5          #$5 = $1 + $2;    10
0x04:   sub $5,$3,$7          #$7 = $5 - $3;    4
0x08:   slt $2,$3,$8          #$8 = ($2 < $3) ? 1 : 0;    0
0x0C:   and $1,$2,$9          #$9 = $1 & $2;   5
0x10:   or  $2,$4,$10         #$10 = $2 | $4;   7
0x14:   IN  $11,1($1)         #data_in -> $11
0x18:   beq $1,$2,4           #pc =0x2C
0x1C:   nop
0x20:   nop
0x24:   nop
0x28:   nop
0x2C:   OUT     $7,1($1)      #data[$2] -> out; 4
0x30:   add $11,$7,$12        #$12 = $11 + $7; data + 4
0x34:   nop
0x38:   sw  $1,(1)$2          #$1 -> memory[$2 + 1]
0x3C:   j   20                #PC =0x50
0x40:   OUT     $12,1($1)     #data[$12] -> out       data + 4
0x44:   nop
0x48:   nop
0x4C:   nop
0x50:   OUT     $11,1($1)     #data[$2] -> out; data
0x54:   j   0                 #PC=0x00
```

十六进制代码：

```
0x00:04222821;                  0x04:04A33822;
0x08:04434025;                  0x0C:04224824;
0x10:04445023;                  0x14:0C2B0001;
0x18:10220004;                  0x1C:00000000;
0x20:00000000;                  0x24:00000000;
0x28:00000000;                  0x2C:2C270001;
0x30:05676021;                  0x34:00000000;
0x38:1C410001;                  0x3C:14000014;
0x40:2C2C0001;                  0x44:00000000;
0x48:00000000;                  0x4C:00000000;
0x50:2C2B0001;                  0x54:14000000;
```

考查项目	考 察 内 容	优	良	中	及格	不及格
业务能力	理论知识、分析与解决问题的能力、实际操作技能、文字处理、口头表达、计算机应用、人际关系、工作效率等					
工作学习态度	主动性、责任心、服务态度、求知欲等					
劳动纪律	遵守各项规章制度（包括业务和行政的规定），迟到、早退、旷工等情况					
实习报告	实习报告撰写的质量					

综合李琦同学自 2019 年 12 月 23 日至 2020 年 1 月 10 日在**计算机组成原理项目综合实训**的实习表现，认定其综合表现为：

优秀（ ）良好（ ）中等（ ）及格（ ）不及格（ ）

指导教师签名（章）：

年 月 日

考查项目	考 察 内 容	优	良	中	及格	不及格
业务能力	理论知识、分析与解决问题的能力、实际操作技能、文字处理、口头表达、计算机应用、人际关系、工作效率等					
工作学习态度	主动性、责任心、服务态度、求知欲等					
劳动纪律	遵守各项规章制度（包括业务和行政的规定），迟到、早退、旷工等情况					
实习报告	实习报告撰写的质量					

综合王雕同学自 2019 年 12 月 23 日至 2020 年 1 月 10 日在**计算机组成原理项目综合实训**的实习表现，认定其综合表现为：

优秀（ ）良好（ ）中等（ ）及格（ ）不及格（ ）

指导教师签名（章）：

年 月 日