合肥工業大學

系统硬件综合设计报告

课　程　名　称　系统硬件综合设计

专　业　班　级　计算机科学与技术 17-1 班

学生学号　2017217xxx

学生姓名　superQ

指　导　教　师

报告目录

一、项目目的

通过设计并实现一个多周期或者流水的 CPU，进一步理解和掌握 CPU 设计的基本原理和过程

二、项目内容

设计并实现一个多周期和流水 CPU

– 若干段流水、可以处理冲突

– 三种类型的指令若干条

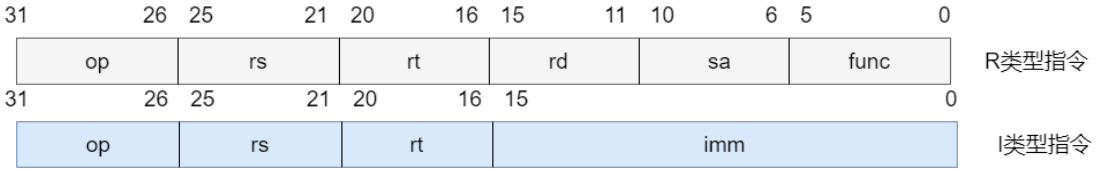
三、项目原理

3.1 指令格式

本次 CPU 设计的指令位数为 32 位

一共涉及到两种类型的指令，分别是 R 类型指令和 I 类型指令

R 类型的指令之所以包含 6 位功能码，是因为因为操作码只有有限的 6 位，而本次 CPU 设计的指令多达 50 多条，并且还考虑到后期增添指令集的情况。因此如果每个指令都使用相同的操作码的话，操作码肯定不够用，因此使用功能码来将功能近似的若干条代码进行打包，也就是这些指令使用同一个操作码，而功能码不同。这样对执行阶段的部分操作也可以进行统一设计，R 类型和 I 类型指令的具体格式与位数如下图所示



3.2 指令集

本次设计总计共完成实现了 53 条指令，包括各类逻辑运算，算术运算指令，移位指令，以及分支指令和部分访存指令，指令集简略介绍如下表所示，指令详情与原理将会在下一小节详细介绍

指令	指令格式	指令作用
ori	ori rs,rt,imm	逻辑或运算，将 rs 对应的寄存器值与立即数进行或运算放入 rt 寄存器中
or	or rd,rs,rt	逻辑或运算，将 rs 与 rt 寄存器中的值进行或运算存放到 rd 寄存器中
and	and rd,rs,rt	逻辑与运算，将 rs 与 rt 寄存器中的值进行与运算存放到 rd 寄存器中
andi	andi rt,rs,imm	逻辑与运算，将 rs 对应的寄存器
nor	nor rd,rs,rt	逻辑或非运算，将 rs 与 rt 寄存器中的值进行或非运算存放到 rd 寄存器中
xor	xor rd,rs,rt	逻辑异或运算，将 rs 与 rt 寄存器中的值进行异或运算存放到 rd 寄存器中
xori	xori rt,rs,imm	逻辑异或运算,将 rs 对应的寄存器值与立即数进行异或运算
lui	lui rt,imm	将 16 位立即数保存到地址为 rt 的通用寄存器的高 16 位
sll	sll rd,rt,sa	逻辑移位指令，将地址为 rt 的寄存器的值向右移动 sa 位

sllv	sllv rd,rt,rs	逻辑移位指令，将地址为 rt 的寄存器的值向右移动 rs 为地址的寄存器的值位
srl	srl rd,rt,sa	逻辑移位指令，将地址为 rt 的寄存器的值向左移动 sa 位
sra	sra rd,rt,sa	算术移位指令，将地址为 rt 的寄存器的值向右移动 sa 位
srlv	srlv rd,rt,rs	逻辑移位指令，将地址为 rt 的寄存器向右移动 rs 为地址的寄存器的值位
srav	srav rd,rt,rs	算术移位指令，将地址为 rt 的寄存器向右移动 rs 为地址的寄存器的值位
add	add rd,rt,rs	算术加法指令，将地址为 rt 的寄存器值与 rs 的值有符号相加结果赋给 rd
addu	addu rd,rt,rs	算术加法指令，将地址为 rt 的寄存器值与 rs 的值无符号相加结果赋给 rd
addi	addi rd,rt,imm	算术加法指令，将地址为 rt 的寄存器值与 16 位立即数有符号相加结果赋给 rd
addiu	addiu rd,rt,imm	算术加法指令，将地址为 rt 的寄存器值与 16 位立即数无符号相加结果赋给 rd
sub	sub rd rt,rs	算术减法指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值进行有符号相减，结果赋给 rd
subu	subu rd,rt,rs	算术减法指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值进行无符号相减，结果赋给 rd

slt	slt rd,rt,rs	算术比较指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值进行有符号比较，如果后者小与前者，那么将 1 保存到 rd 的通用寄存器中，否则保存 0
sltu	sltu rd,rt,rs	算术比较指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值进行无符号比较，如果后者小与前者，那么将 1 保存到 rd 的通用寄存器中，否则保存 0
slti	slti rd,rt,imm	算术比较指令，将地址为 rt 的寄存器值与 16 位立即数进行有符号比较，如果后者小与前者，那么将 1 保存到 rd 的通用寄存器中，否则保存 0
sltiu	sltiu rd,rt,imm	算术比较指令，将地址为 rt 的寄存器值与 16 位立即数进行无符号比较，如果后者小与前者，那么将 1 保存到 rd 的通用寄存器中，否则保存 0
mul	mul rd,rs,rt	算术乘法指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值有符号相乘，结果存入 rd
mult	mult rs,rt	算术乘法指令，将地址为 rt 的寄存器值与地址为 rs 的寄存器值有符号相乘，结果存入{hi,lo}中
multu	multu rs,rt	算术乘法指令，将地址为 rt 的寄存器值与地址

		为 rs 的寄存器值无符号相乘，结果存入{hi,lo}中
div	div rs,rt	算术除法指令，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行有符号除法运算，商与余数分别保存到寄存器 LO 与 HI 中
divu	divu rs,rt	算术除法指令，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行无符号除法运算，商与余数分别保存到寄存器 LO 与 HI 中
movn	movn rd,rs,rt	如果 rt 寄存器值不为 0，则将 rs 寄存器的值赋给 rd
movz	movz rd,rs,rt	如果 rt 寄存器值为 0，则将 rs 寄存器的值赋给 rd
mfhi	mfhi rd	将寄存器 hi 的值赋给 rd 寄存器
mflo	mflo rd	将寄存器 lo 的值赋给 rd 寄存器
mthi	mthi rs	将寄存器 rs 的值赋给 hi 寄存器
mtlo	mtlo rs	将寄存器 rs 的值赋给 lo 寄存器
madd	madd rs,rt	算术乘累加指令，实现有符号运算 {HI,LO}<-{HI,LO}+rs*rt
maddu	maddu rs,rt	算术乘累加指令，实现无符号运算 {HI,LO}<-{HI,LO}+rs*rt
msub	msub rs,rt	算术乘累减指令，实现有符号运算 {HI,LO}<-{HI,LO}-rs*rt

msubu	msubu rs,rt	算术乘累减指令，实现无符号运算 {HI,LO}<-{HI,LO}-rs*rt
j	j target	转移到新的指令地址 pc <- (pc+4)[31,28]||target||00
jal	jal target	转移到新的指令地址 ,并将跳转指令后面第二条指令的地址作为返回地址保存到寄存器$31 pc <- (pc+4)[31,28]||target||00
jr	jr rs	将地址为 rs 的寄存器的值赋给 PC
jalr	jalr rs	将地址为 rs 的寄存器的值赋给 PC,并且将跳转指令后面第二条的指令地址作为返回地址保存到地址为 31 的寄存器中
beq	beq rs,rt,offset	一旦 rs 和 rt 地址的寄存器的值相同，则跳转 跳转地址等于当前 pc 值+4 与扩展后的 offset 左移两位所得的结果
bgtz	bgtz rs,offset	如果 rs 寄存器的值大于 0，则进行跳转，跳转结果与 beq 中描述相同
blez	blez rs,offset	如果 rs 寄存器的值小于等于 0，则进行跳转，跳转结果与 beq 中描述相同
bne	bne rs,rt,offset	一旦 rs 和 rt 地址的寄存器的值不同，则跳转
bltz	bltz rs,offset	如果 rs 寄存器的值小于 0，则进行跳转，跳转结果与 beq 中描述相同
bltzal	bltzal rs,offset	如果 rs 寄存器的值小于 0，则进行跳转，跳转

		结果与 beq 中描述相同，并将转移指令后面第二条指令的地址作为返回地址保存到通用寄存器$31
bgez	bgez rs,offset	如果 rs 寄存器的值大于等于 0，则进行跳转，跳转结果与 beq 中描述相同
bgezal	bgezal rs,offset	如果 rs 寄存器的值大于等于 0，则进行跳转，跳转结果与 beq 中描述相同，并将转移指令后面第二条指令的地址作为返回地址保存到通用寄存器$31
lb	lb rt,offset(base)	从内存中指定的加载地址出，读取一个字节，随后符号扩展到 32 位，保存到地址为 rt 的通用寄存器中
sb	sb rt,offset(base)	将地址为 rt 的通用寄存器的最低字节存储到内存中的指定地址

3.3 模块设计

3.3.1 程序计数器模块 PC

表 3.1 PC 模块输入信号，输出信号与功能

输入	时钟信号 clk，重置信号 rst，暂停序列 stall [5:0] 分支符号 branch_flag_i, 分支目标地址 branch_target_address_i[31:0]
输出	计数值 pc,片使能信号 ce

功能	是用来确认下一条指令的地址的模块，正常情况下 PC 每次计数都是+4，但是因为下一条指令的地址还会受到分支指令的影响，因此还要引入分支的相关信号

程序计数器 PC 结构图如下图所示

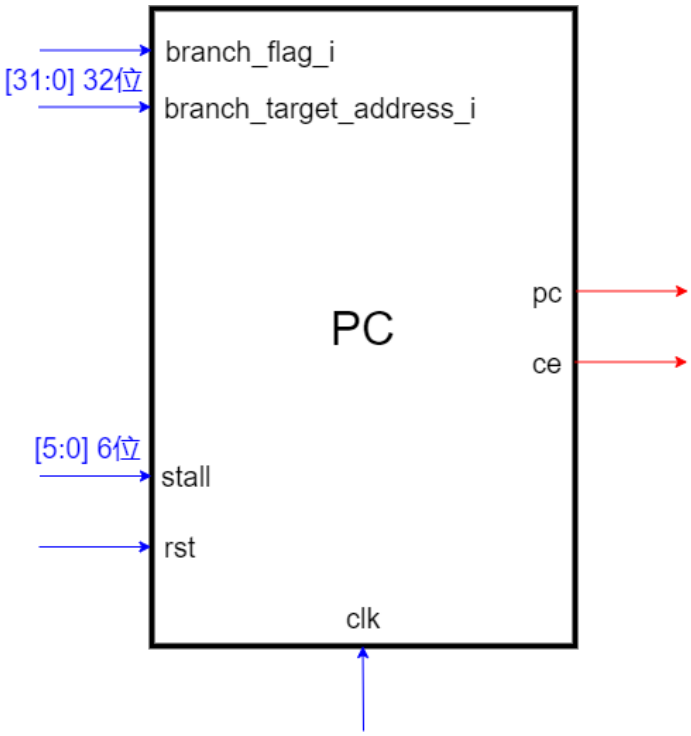


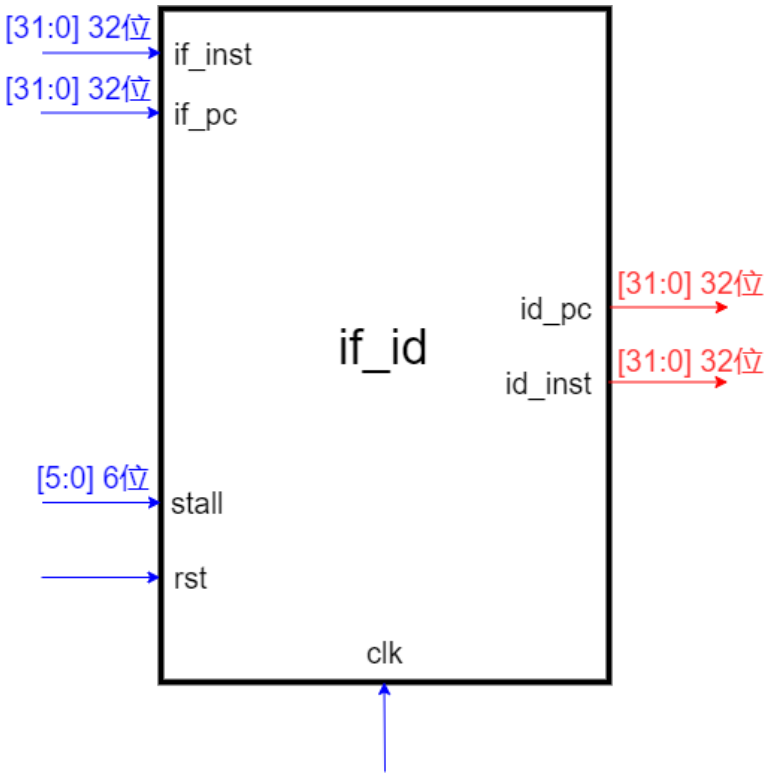
图3.3.1 程序计数器PC结构图

3.3.2 if_id 模块

表 3.2 if_id 模块输入信号，输出信号与功能

输入	时钟信号 clk，重置信号 rst，取指阶段的 pc 值 if_pc[31:0] 取指阶段的指令 if_inst [31:0]， 暂停序列 stall [5:0]
输出	译码阶段的 pc id_pc[31:0]，译码阶段的 pc id_inst[31:0]
功能	是 CPU 流水线中的第一个传递模块，用与将从第一个取指阶段中的部分信号与指令传递到下一个译码阶段中

if_id 模块结构图如下图所示



if_id模块结构图

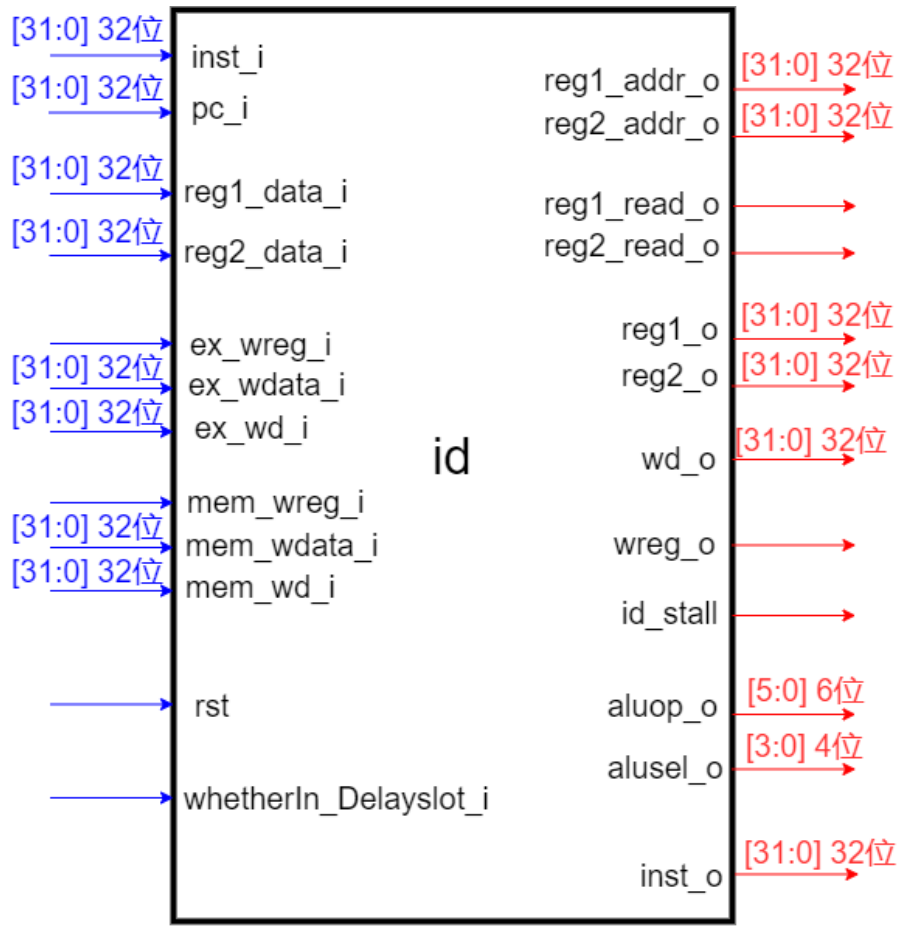
3.3.3 id 模块

表 3.3 id 模块输入信号，输出信号与功能

输入	重置信号 rst，取指阶段的 pc 值 pc_i[31:0] 取指阶段的指令 inst_i [31:0]，第一个操作数数据 reg1_data_i[31:0] 第二个操作数数据 reg2_data_i[31:0]， 执行阶段的写使能信号(数据前推) ex_wreg_i, 执行阶段的写入数据(数据前推) ex_wdata_i[31:0] 执行阶段的写入地址(数据前推) ex_wd_i[31:0] 访存阶段的写使能信号(数据前推) mem_wreg_i,

	访存阶段的写入数据(数据前推) mem_wdata_i[31:0] 访存阶段的写入地址(数据前推) mem_wd_i[31:0] 是否位于延迟槽内信号 whetherInDelayslot_i
输出	分支信号 branch_flag_o 分支目标地址 branch_target_address_o[31:0] 译码阶段是否位于延迟槽内信号 whetherIn_Delayslot_o 链接地址 link_addr_o[31:0] 后面进入译码阶段的指令是否位于延迟槽信号 next_inst_delayslot_o 译码阶段第一个读使能信号 reg1_read_o[31:0] 译码阶段第二个读使能信号 reg2_read_o[31:0] 译码阶段第一个读取地址 reg1_addr_o[31:0] 译码阶段第二个读取地址 reg2_addr_o[31:0] 运算符类型 aluop_o[5:0],运算符种类 alusel_o[3:0] 译码阶段第一个源操作数 reg1_o[31:0] 译码阶段第二个源操作数 reg2_o[31:0] 译码阶段的写使能信号(数据前推) ex_wreg_i, 译码阶段的写使能信号 wreg_o，执行阶段的写入地址 wd_o[31:0] 译码阶段第指令 inst_i[31:0], id 阶段的暂停序列 id_stall
功能	译码阶段是流水线中的第二个阶段，在这里会将从取指阶段取得的指令进行一系列分解与解析。通过对分解之后获得的操作码和功能码的判断可以将不同的信号设置为不同的值方便不同指令进行相应的后续操作与功能实现

id 模块结构图如下图所示



id模块结构图

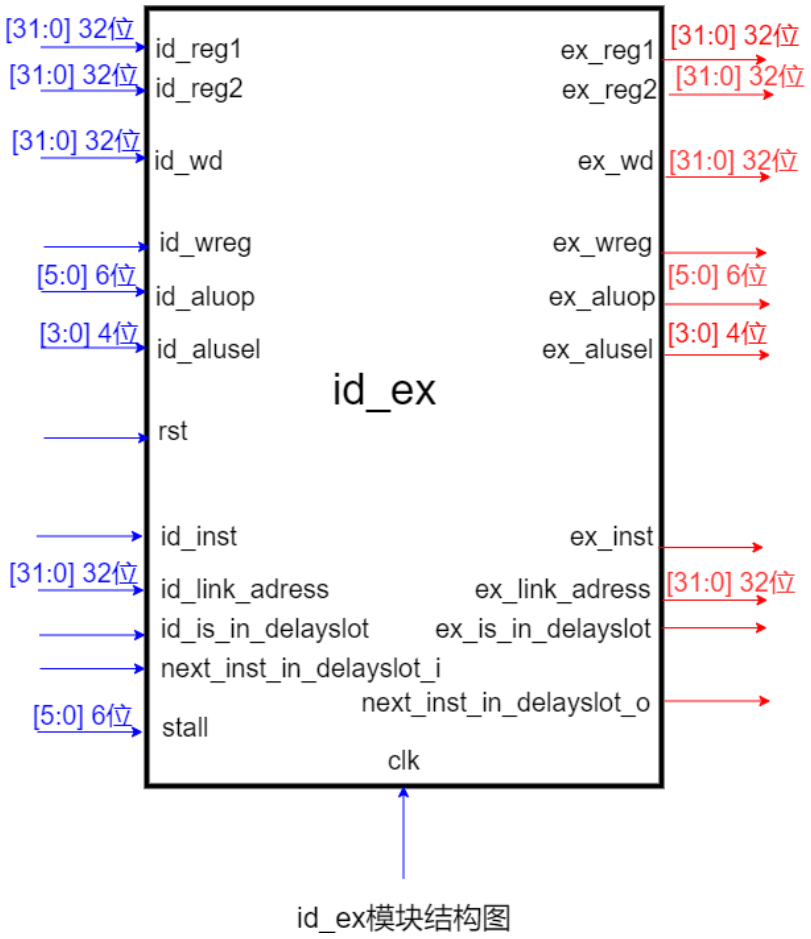
3.3.4 id_ex 模块

表 3.4 id_ex 模块输入信号，输出信号与功能

输入	时钟信号 clk，重置信号 rst 译码阶段运算符类型 id_aluop[5:0]，译码阶段运算符种类 id_alusel[3:0] 译码阶段第一个源操作数 id_reg1[31:0] 译码阶段第二个源操作数 id_reg2[31:0] 译码阶段的写使能信号 id_wreg，译码阶段的写入地址 id_wd[31:0] 暂停序列 stall[5:0]

	当前处于译码阶段的指令 id_inst[31:0]
输出	执行阶段运算符类型 ex_aluop[5:0],执行阶段运算符种类 ex_alusel[3:0] 执行阶段第一个源操作数 ex_reg1[31:0] 执行阶段第二个源操作数 ex_reg2[31:0] 执行阶段的写使能信号 ex_wreg，执行阶段的写入地址 ex_wd[31:0] 当前处于执行阶段的指令 ex_inst[31:0] 处于执行阶段的转移指令要保存的返回地址 ex_link_address[31:0] 当前处于执行阶段的指令是否位于延迟槽信号 ex_is_in_delayslot 当前处于译码阶段的指令是否位于延迟槽信号 s_in_delayslot_o
功能	该模块是本 CPU 流水线中的第二个传递模块，将从第二个译码阶段中传来的各类参数和信号传入第三个执行阶段

id_ex 模块结构图如下图所示



id_ex模块结构图

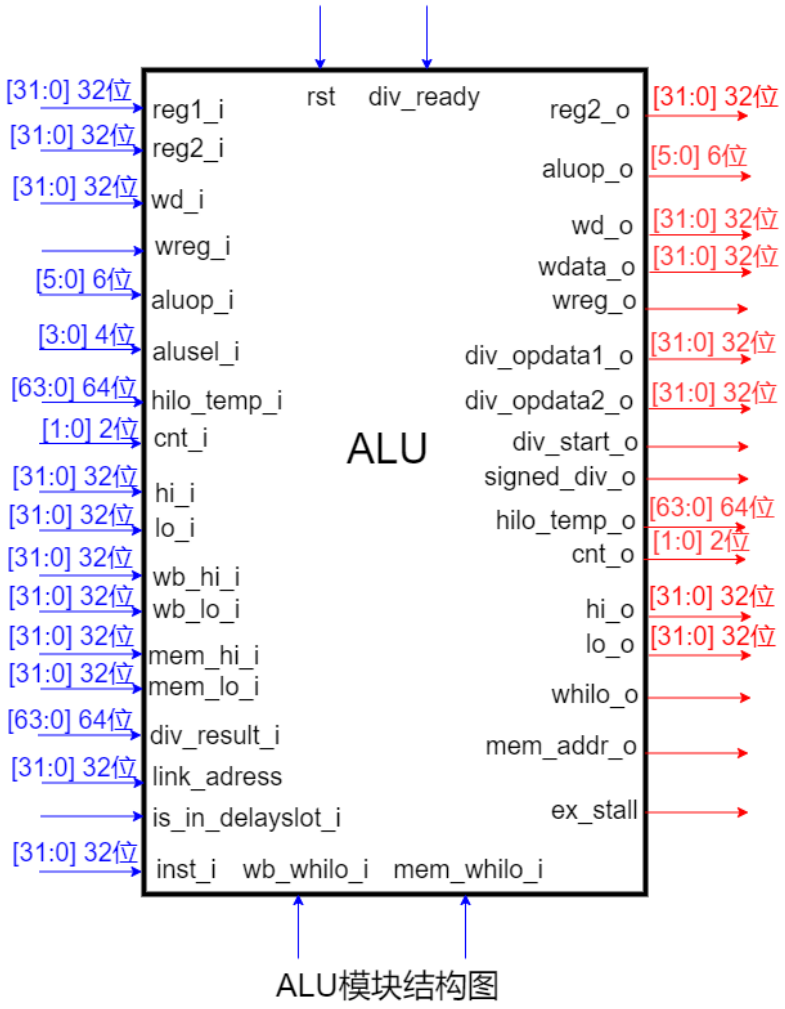
3.3.5 ALU 模块

表 3.5 ALU 模块输入信号，输出信号与功能

输入	重置信号 rst 运算符类型 aluop_i[5:0]，运算符种类 alusel_i[3:0] 第一个源操作数 reg1[31:0] 第二个源操作数 reg2[31:0] 写使能信号 wreg_i，写入地址 wd_i[31:0] HI 寄存器输入值 hi_i[31:0],LO 寄存器输入值 lo_i[31:0] 写回阶段的 HI 输入值（数据前推）wb_hi_i[31:0] 写回阶段的 LO 输入值（数据前推）wb_lo_i[31:0]

	写回阶段的 HI/LO 写使能信号（数据前推） wb_whilo_i 访存阶段的 HI 输入值（数据前推）mem_hi_i[31:0] 访存阶段的 LO 输入值（数据前推）mem_lo_i[31:0] 访存阶段的 HI/LO 写使能信号（数据前推） mem_whilo_i 第一个执行周期得到的乘法结果 hilo_temp_i[63:0] 当前处于的乘法阶段 cnt_i[1:0] 除法运算的结果 div_result_i[63:0]，除法运算完成标志 div_ready_i 当前处于执行阶段的指令 inst_i[31:0]
输出	执行阶段运算符类型 aluop_o[5:0] 存储指令对应存储器地址 mem_addr_o[31:0] 传到访存指令的第二个源操作数 reg2_o[31:0] 除法运算被除数 div_opdata1_o[31:0]，除数 div_opdata2_o[31:0] 除法开始运算标志 div_start_o，是否是有符号除法 signed_div_o 第一个周期得到的乘法结果输出 hilo_temp_o[63:0] 当前乘法所处周期[1:0] HI 输入值 hi_o[31:0],LO 输入值 lo_o[31:0],HI/LO 写使能信号 whilo_o 执行阶段写使能信号 wreg_o，执行阶段写入地址 wd_o[31:0] 要写入的数据 wdata_o[31:0],执行阶段的暂停信号 ex_stall,
功能	ALU 模块是流水线中处于第三阶段的执行阶段，主要用于各类算术，逻辑运算，移位操作，譬如加减法，除法，乘法，累乘加法等，同时根据运算的结果与类型决定传入下一个阶段的数和信号

ALU 模块结构图如下图所示



ALU模块结构图

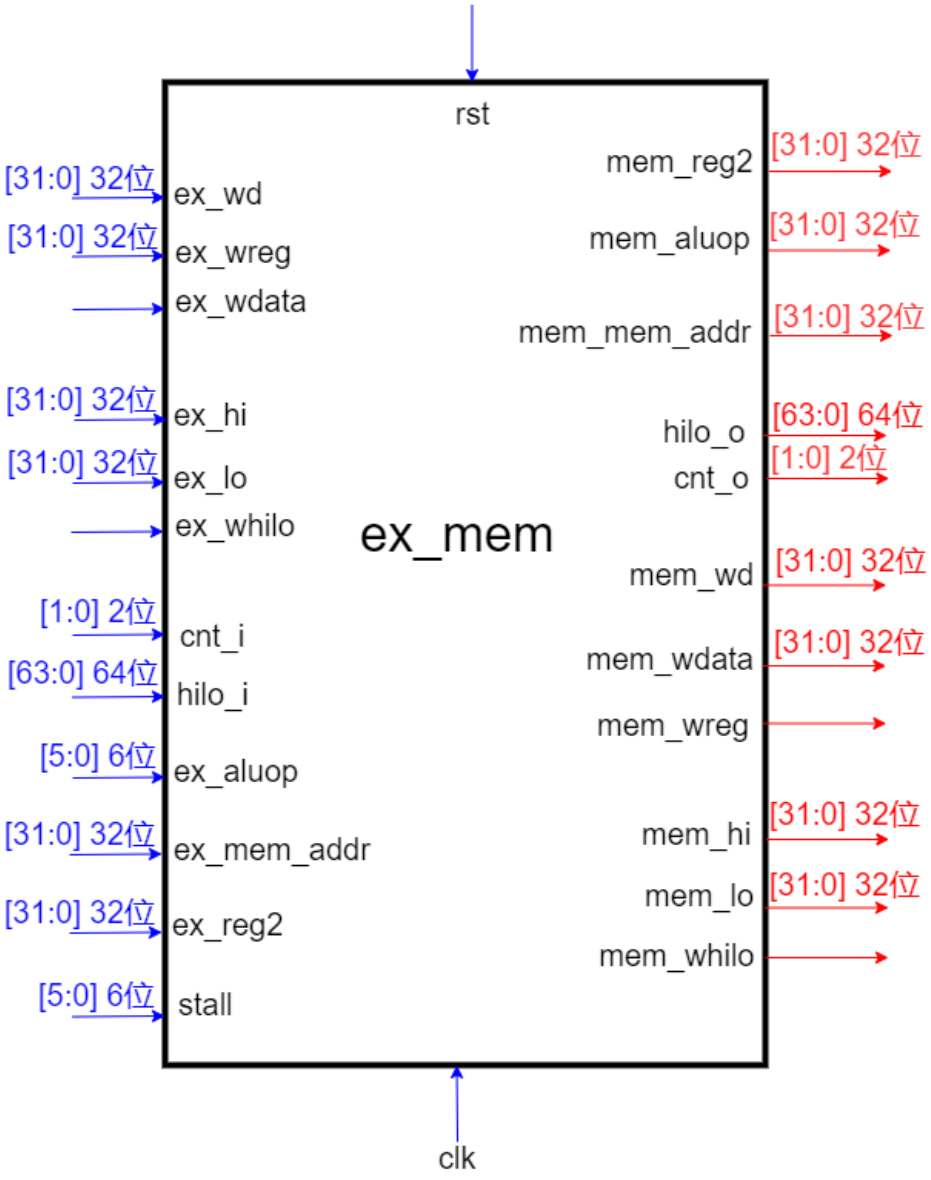
3.3.6 ex_mem 模块

表 3.6 ex_mem 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst 执行阶段运算符类型 ex_aluop[5:0] 存储指令对应存储器地址 ex_mem_addr [31:0] 传到访存阶段的第二个源操作数 ex_reg2 [31:0] 执行阶段写使能信号 ex_wreg，执行阶段写入地址 ex_wd[31:0] 执行阶段写入数据 ex_wdata[31:0] 执行阶段 HI 寄存器输入值 ex_hi [31:0]

	执行阶段 LO 寄存器输入值 ex_lo [31:0] 执行阶段的 HI/LO 写使能信号 ex_whilo 暂停序列 stall [5:0] 执行阶段乘法传递结果 hilo_i[63:0] 执行阶段传递的所处乘法阶段 cnt_i[1:0]
输出	访存阶段运算符类型 mem_aluop[5:0] 访存指令对应存储器地址 mem_mem_addr [31:0] 访存阶段的第二个源操作数 mem_reg2 [31:0] 访存阶段写使能信号 mem_wreg，访存阶段写入地址 mem_wd[31:0] 访存阶段写入数据 mem_wdata[31:0] 访存阶段 HI 寄存器输入值 mem_hi [31:0] 访存阶段 LO 寄存器输入值 mem_lo [31:0] 访存阶段的 HI/LO 写使能信号 mem_whilo 访存阶段乘法传结果 hilo_o[63:0] 访存阶段传递的所处乘法阶段 cnt_o[1:0]
功能	ex_mem 是本 CPU 中第三个传递模块，是将 ALU 模块中传来的各类参数与信号传入第四个访存阶段

ex_mem 模块结构图如下图所示



ex_mem模块结构图

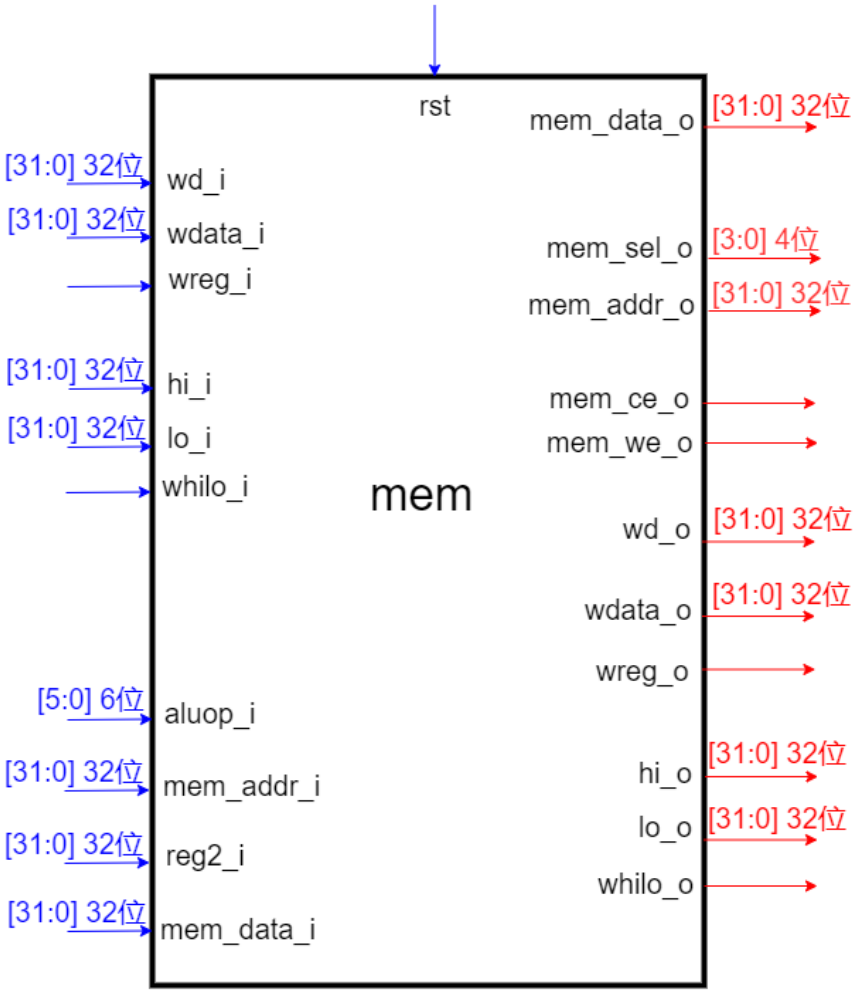
3.3.7 mem 模块

表 3.7 mem 模块输入信号，输出信号与功能

输入	重置信号 rst 运算符类型 aluop_i[5:0] 存储指令对应存储器地址 mem_addr _i[31:0] 存储数据 mem_data _i[31:0]

	第二个源操作数 reg2_i [31:0] 写使能信号 wreg_i，写入地址 wd_i[31:0] 写入数据 wdata_i[31:0] HI 寄存器输入值 hi_i [31:0] LO 寄存器输入值 lo_i [31:0] HI/LO 写使能信号 whilo_i
输出	访存指令对应存储器地址 mem_addr_o [31:0] 访存阶段写使能信号 wreg_o，访存阶段写入地址 wd_o[31:0] 访存阶段写入数据 wdata_o[31:0] 访存阶段 HI 寄存器输入值 hi_o [31:0] 访存阶段 LO 寄存器输入值 lo_o [31:0] 访存阶段的 HI/LO 写使能信号 whilo_o 访存阶段写使能信号 mem_we_o 访存阶段片使能信号 mem_ce_o 芳村阶段指令类型 mem_sel_o[3:0] 要写入数据存储器的数据 mem_data_o[31:0]
功能	访存阶段是本流水线中的第四个阶段，主要功能和其名称一致，就是访问内存，通过对从执行阶段中传过来的运算符类型等信号从而对内存中读取数据或者存储数据

mem 模块结构图如下图所示



mem模块结构图

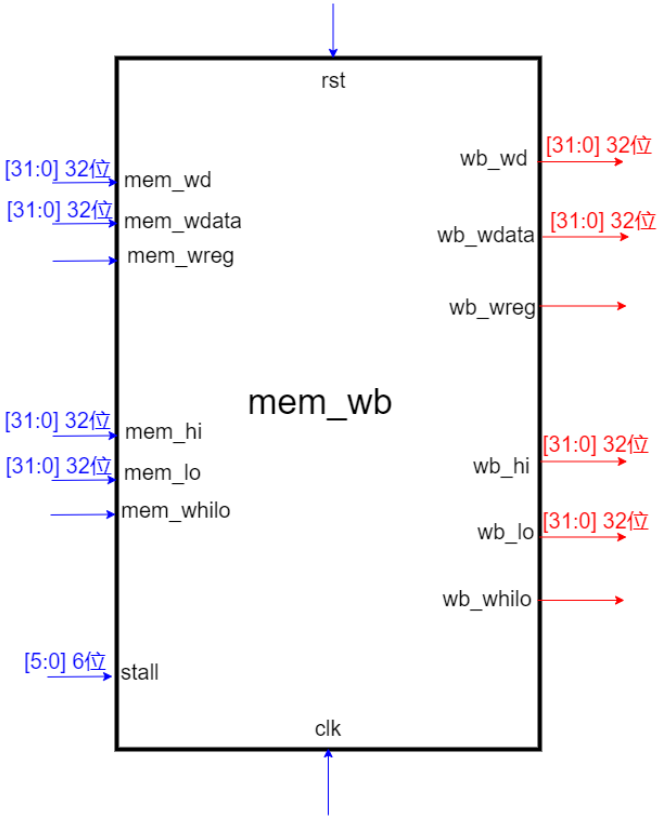
3.3.8 mem_wb 模块

表 3.8 mem_wb 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst 访存阶段写使能信号 mem_wreg，访存阶段写入地址 mem_wd[31:0] 访存阶段写入数据 mem_wdata[31:0] 访存阶段 HI 寄存器输入值 mem_hi [31:0] 访存阶段 O 寄存器输入值 mem_lo [31:0] 访存阶段 HI/LO 写使能信号 mem_whilo

	暂停序列 stall[5:0]
输出	写回阶段写使能信号 wb_wreg，写回阶段写入地址 wb_wd[31:0] 写回阶段写入数据 wb_wdata[31:0] 写回阶段 HI 寄存器输入值 wb_hi [31:0] 写回阶段 O 寄存器输入值 wb_lo [31:0] 写回阶段 HI/LO 写使能信号 wb_whilo
功能	mem_wb 模块是流水线中最后一个传递阶段，该阶段将从访存阶段中传入的从内存中读取的数据和信号传入写回阶段

mem_wb 模块结构图如下图所示



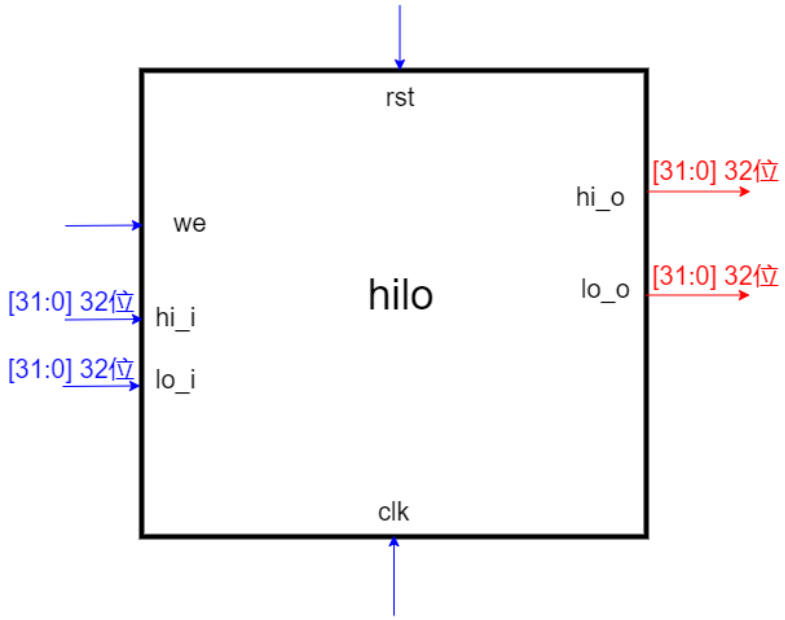
mem_wb模块结构图

3.3.9 hilo 寄存器模块

表 3.9 hilo 寄存器模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst 访写使能信号 we HI 寄存器输入值 hi_i [31:0] LO 寄存器输入值 lo_i [31:0]
输出	HI 寄存器输出值 hi_o [31:0] LO 寄存器输出值 lo_o [31:0]
功能	hilo 寄存器模块用于特殊寄存器 HI 和 LO 的读入与读出操作，包括将值从乘法运算或者 MTHI、MTLO 指令中传入 HI 或 LO 寄存器，以及当 MFHI,MFLO 指令时将值从中读取

hilo 寄存器模块结构图如下图所示



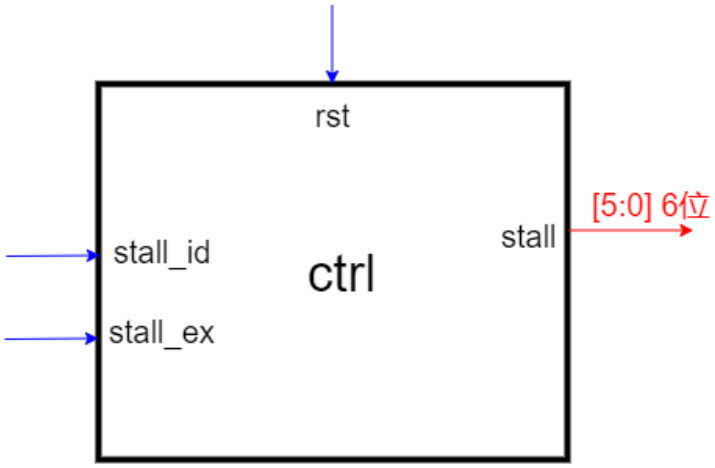
hilo模块结构图

3.3.10 ctrl 模块

表 3.10 ctrl 模块输入信号，输出信号与功能

输入	重置信号 rst 来自 id 的暂停信号 stall_id，来自 ex 的暂停信号 stall_ex
输出	暂停序列 stall[5:0]
功能	ctrl(控制)模块主要用于处理本 CPU 中的流水线暂停，根据来自译码和执行阶段的暂停信号从而生成暂停序列，输出到各个模块中完成流水线暂停

ctrl 模块结构图如下图所示



ctrl模块结构图

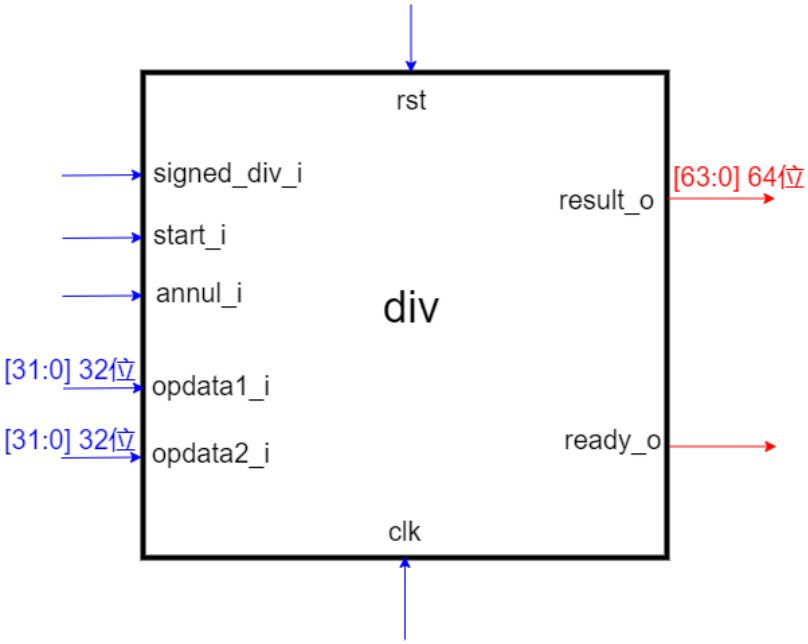
3.3.11 div 模块

表 3.11 div 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst 除法运算符号 signed_div_i 除法被除数 opdata1_i[31:0],除数 opdata2_i[31:0]

	除法开始标志 start_i，取消除法运算标志 annul_i
输出	除法运算结果 result_i[63:0]，除法是否准备好 ready_o
功能	除法 div 模块主要用于除法运算，因为本 CPU 使用的是试商法，32 位数据就需要进行 32 次除法运算，并且除法还分阶段，具体会在接下来的除法实现章节具体解释

div 模块结构图如下图所示



div模块结构图

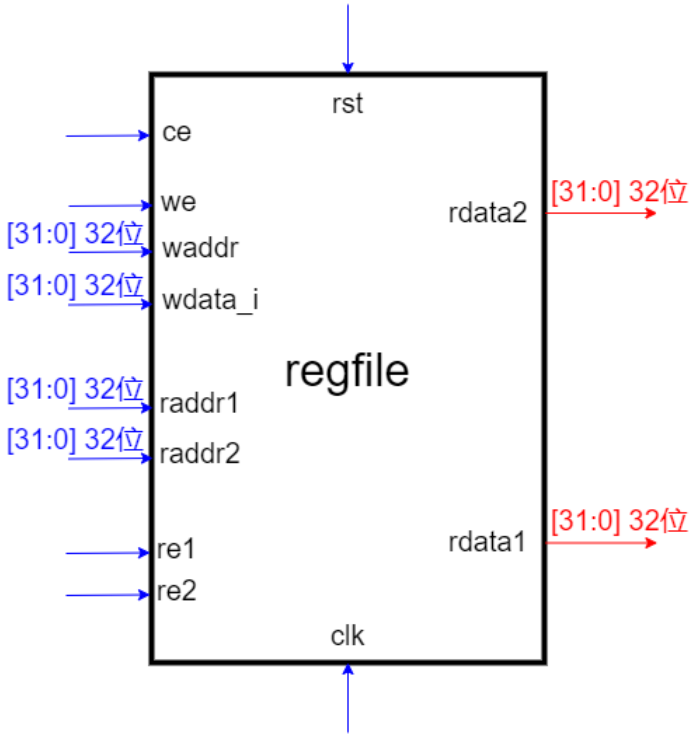
3.3.12 寄存器堆模块

表 3.12 寄存器堆模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst， 写使能信号 we 写入地址 waddr[31:0],写入数据 wdata_i[31:0] 读取地址 1 raddr1[31:0] 读使能信号 1 re1，读使能信号 2 re2

	读取地址 2 raddr2[31:0]
输出	读取数据 1 rdata1[31:0] 读取数据 2 rdata2[31:0]
功能	RAM 模块是本 CPU 的数据存储器，主要也是用于存储数据

寄存器堆模块结构图如下图所示



寄存器堆模块结构图

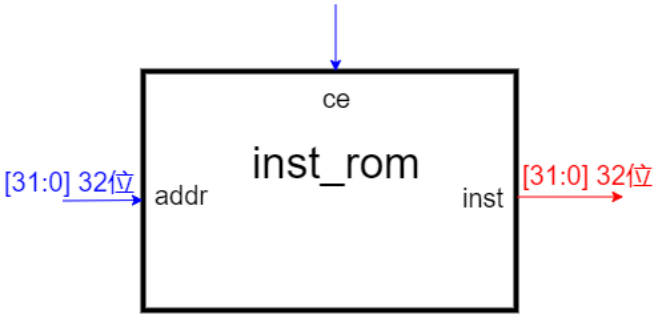
3.3.13 寄存器堆模块

表 3.13 指令寄存器模块输入信号，输出信号与功能

输入	片使能信号 ce，指令地址 addr
输出	输出指令 inst
功能	指令寄存器存储着所有待执行的指令，从 PC 中获取指令地址，并从该模块中取出要执行的指令，本 CPU 使用 readmemh/readmemb 函数从文

	本文件中读取十六进制或者二进制的指令赋给指定寄存器变量

指令寄存器模块结构图如下图所示



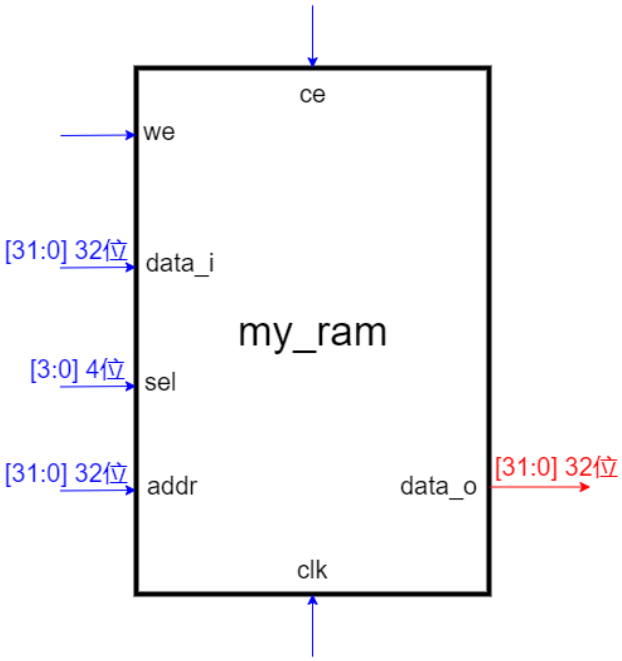
inst_rom模块结构图

3.3.14 RAM 模块

表 3.14 RAM 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst， 写使能信号 we 写入地址 addr[31:0]，指令种类 sel[3:0] 写入数据 data_i[31:0]
输出	读出数据 data_o[31:0]
功能	RAM 模块是本 CPU 的数据存储器，主要也是用于存储数据

RAM 模块结构图如下图所示



my_ram模块结构图

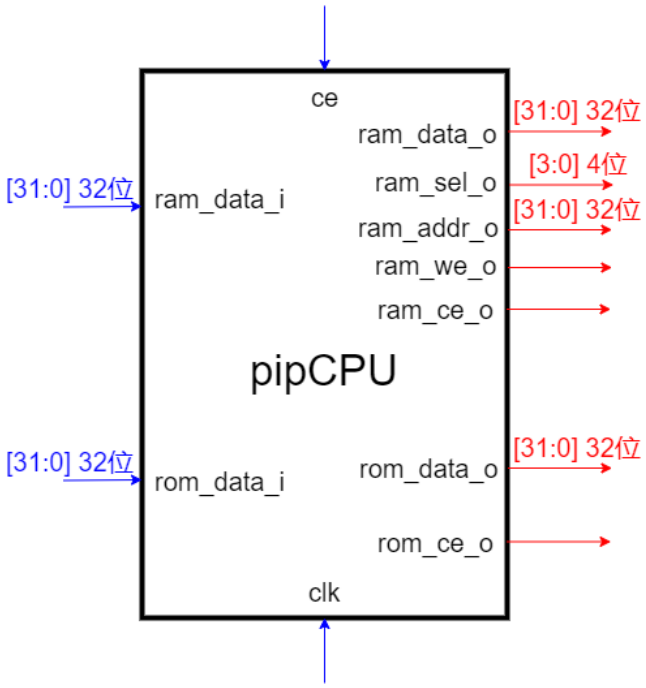
3.3.15 pip_CPU 模块

表 3.15 **pip_CPU** 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst 来自 rom 的数据(指令)rom_data_i[31:0] 来自 ram 的数据(数据)ram_data_i[31:0]
输出	传入 rom 的地址 rom_addr_o[31:0] 传入 rom 的写使能信号 rom_we_o 传入 ram 的地址 ram_addr_o[31:0] 传入 ram 的数据 ram_data_o[31:0] 传入 ram 的写使能信号 ram_we_o 传入 ram 的片使能信号 ram_ce_o 传入 ram 的指令类型 ram_sel_o

功能	该模块用于封装上述模块中的大部分模块，一共封装了 pc_reg, hilo_reg, div,if_id, id, id_ex, ALU, ex_mem, mem, mem_wb, ctrl 共计 11 个模块，并设置输入输出信号与 rom 和 ram 进行数据交换

pip_CPU 模块结构图如下图所示



pip_CPU模块结构图

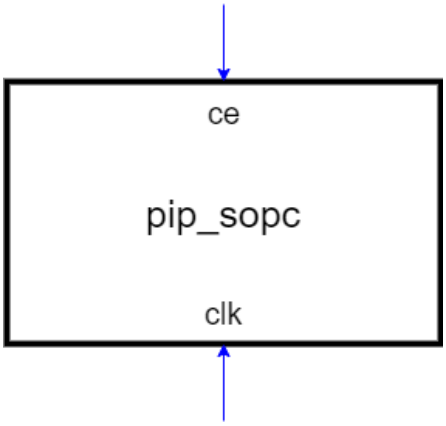
3.3.16 pip_sopc 模块

表 3.16 **pip_sopc** 模块输入信号，输出信号与功能

输入	时钟信号 clk,重置信号 rst
输出	无
功能	sopc 即可编程片上系统，本 CPU 中该模块封装了三个模块，分别是 pipCPU,inst_rom 以及 my_ram 模块，封装在同一个 SOPC 中便于三个模块之间互相通信，同时 SOPC 接收来自顶端测试模块的时钟信号和片

	使能信号

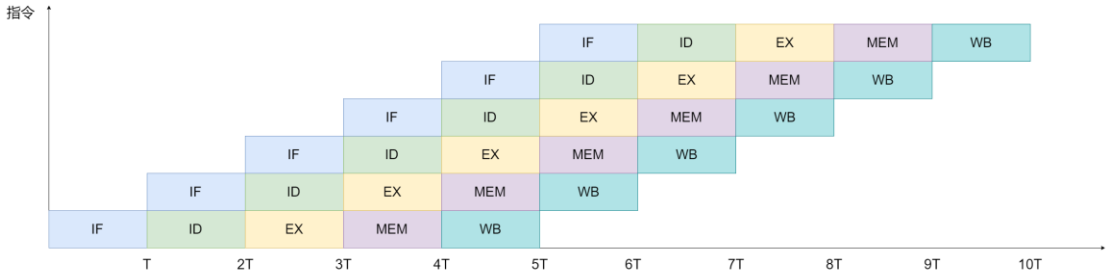
pip_sopc 模块结构图如下图所示



pip_sopc模块结构图

3.4 流水线机制

本 CPU 采用的是五段流水机制，五个阶段分别是取指阶段，译码阶段，执行阶段，访存阶段以及写回阶段，详细示意图如下图所示



五段流水线示意图

表 3.17 流水线五个阶段各详细作用表

阶段	主要作用
取指(IF)	取指阶段根据 PC 计数获取指令地址，再从 inst_rom 指令寄存器中获取当前指令，并

	通过 if_id 模块传入译码阶段
译码(ID)	对指令进行分析，根据其操作码与功能码确定指令性质，并使用分支语句来设置各个信号。将信号和值通过 id_ex 模块传入执行阶段
执行(EX)	执行阶段根据从译码阶段传来的信号进行不同种类的运算，可以进行逻辑运算，算术运算，移位操作等。并将运算结果和各个信号送入访存阶段
访存(MEM)	从上一个阶段送来的指令操作码在这一阶段是访存指令的重要依据之一，CPU 在这里根据这些来判断如何访存，并将执行阶段送来的参数与信号部分送入写回阶段
写回(WB)	将运算的结果获取他参数存取到寄存器堆中

指令流水的一个很大的优点就是在一个时钟周期内可以有多条指令在执行不同的阶段，这样会很大程度上提高系统的并行性。降低系统运行指令的时间花费，提高整个处理机的处理能力。

然而与此同时流水线带来的一个问题就是一条指令后边若干条指令如果要用到这条指令的执行结果，因为该指令仍然处于执行或者访存阶段，并未将数据写回到寄存器堆，因此后面的指令在指定的寄存器中并未找到所需的数，这会造成后边的运算直接全部出现问题。这时候本人引入学计算机体系结构时有学过的一种机制，就是数据前推机

制

3.5 数据前推机制

本 CPU 中的数据前推机制不仅考虑到了运算阶段的运算结果，还考虑到了 HI 与 LO 寄存器，因为包括乘法运算在内的多种运算要使用到 HI 与 LO 寄存器。

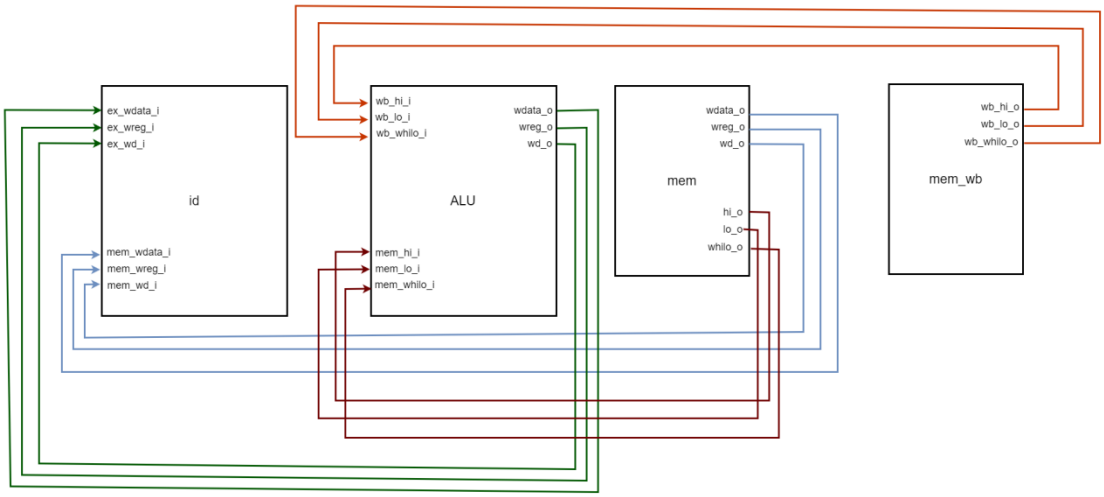
举例来说，考虑 MFLO 或者 MFHI 指令会跟随在 MULT 指令之后，如下

MULT $2,$3

MFHI $5

这时候 MFHI 需要的 HI 内容就是 MULT 运算完成之后存放到 HI 特殊寄存器中的内容，因此 HI 与 LO 寄存器也是需要进行数据前推操作的

因为五段流水线最多可能会引起三条指令之前的互相冲突，因此本数据前推机制进行了两个时期内的数据前推，具体前推对应数据流如下图所示



数据前推示意图

3.6 延迟槽

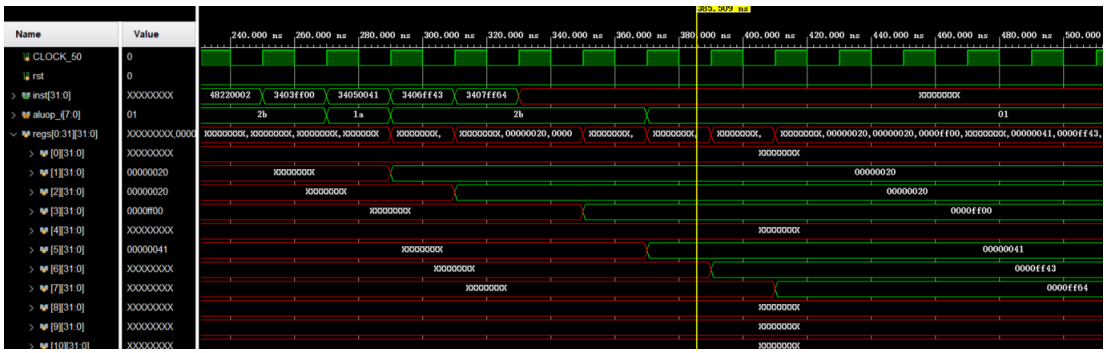
本 CPU 设计中包含部分分支指令，为了提高流水线的执行效率，对这些分支指令采取延迟槽机制，考虑以下代码

```
ori $1,$0, 0x0020
```

ori $2,$0,0x0020
beq $1,$2, 6
ori $3,$0,0xff00
.....

其中第三条指令 BNE 就是一条分支指令，因为本 CPU 使用的是延迟槽机制，因此 BNE 后一条指令，也就是这条“ori $3,$0,0xff00”就放入了延迟槽。不过需要注意的是，放入延迟槽中的指令无论分支与否都会被执行。因此延迟槽中的指令理应谨慎选择，要确保不会对分支之后的执行内容产生影响，比如上述这条延迟槽中的指令就不是很合适，因为倘若分支后的运算使用到了 3 号寄存器中的内容，那么其运算就不是原来所期望的操作数。

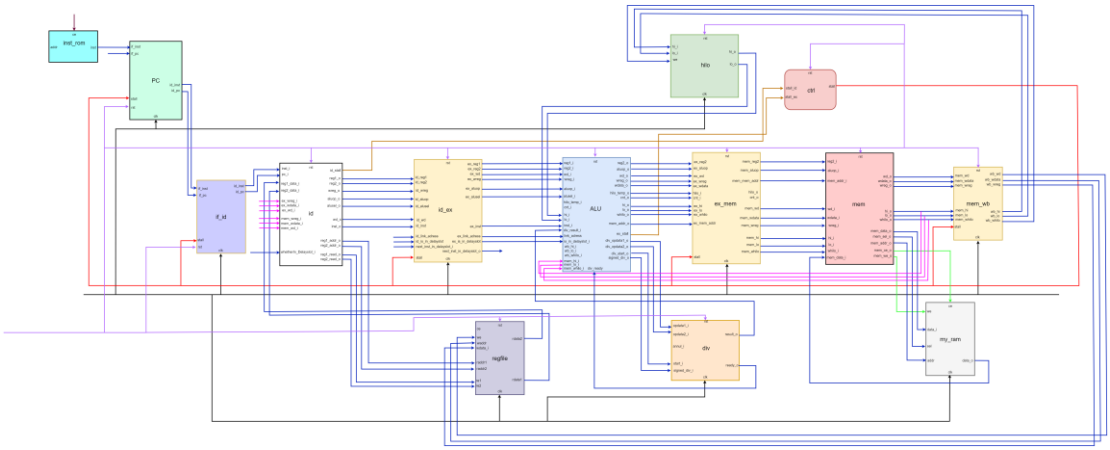
测试波形图如下图所示，可以看到，第三条是分支指令，因为这个缘故因此 3 号寄存器写回延迟了一个周期。BEQ 分支满足了条件并转移（4 号寄存器没有写入就是一个证明），但是在 BEQ 后一条指令，也就是在延迟槽里的指令依旧执行并写回



因此最好采用手动调度或者用编译器来调度，将一些相对无用或者与分支后的指令不会产生冲突的指令

3.7 最终总数据流图展示

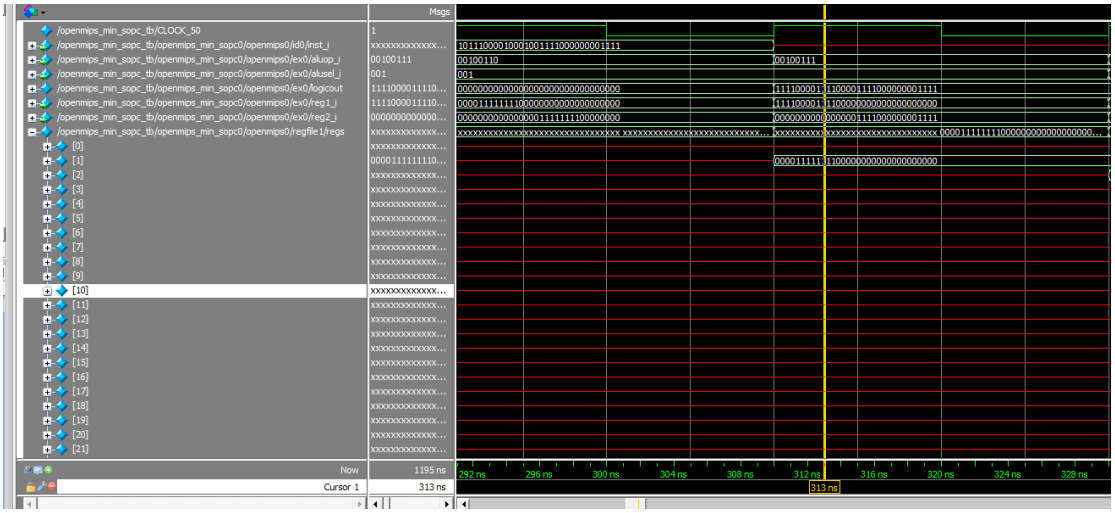
总共花费约 1 小时多的时间绘制出了这幅数据通路，因为图过于庞大，因此这里只能最多展示成这样，完整的数据流图请参考附件中的 png 图像或者 drawio 文件



四、项目步骤

4.1 概述

本项目前期采用 Modelsim SE 10.1a 完成

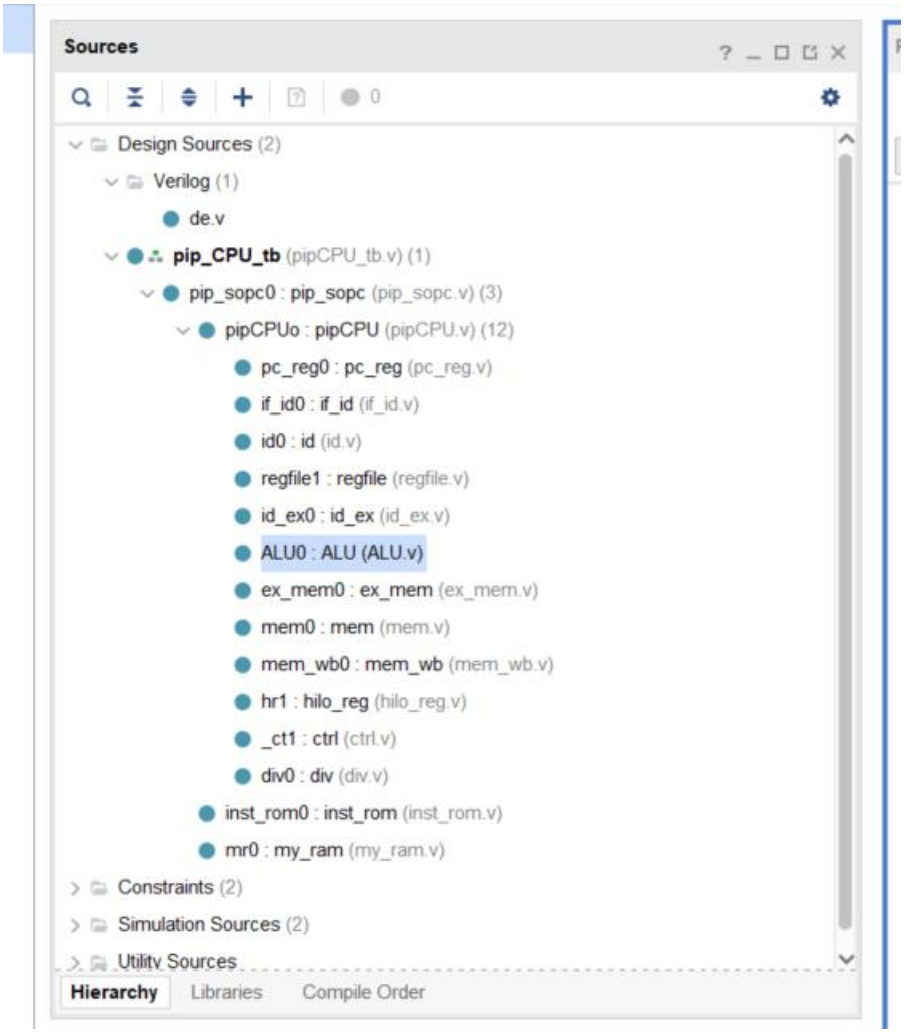


中期初步安装并学习了一下 vivado，后期全部转战 vivado 2019.2 并完善项目所有细节，因为本人觉得 vivado 的软件 UI 更加现代更美观，并且使用起来也更方便

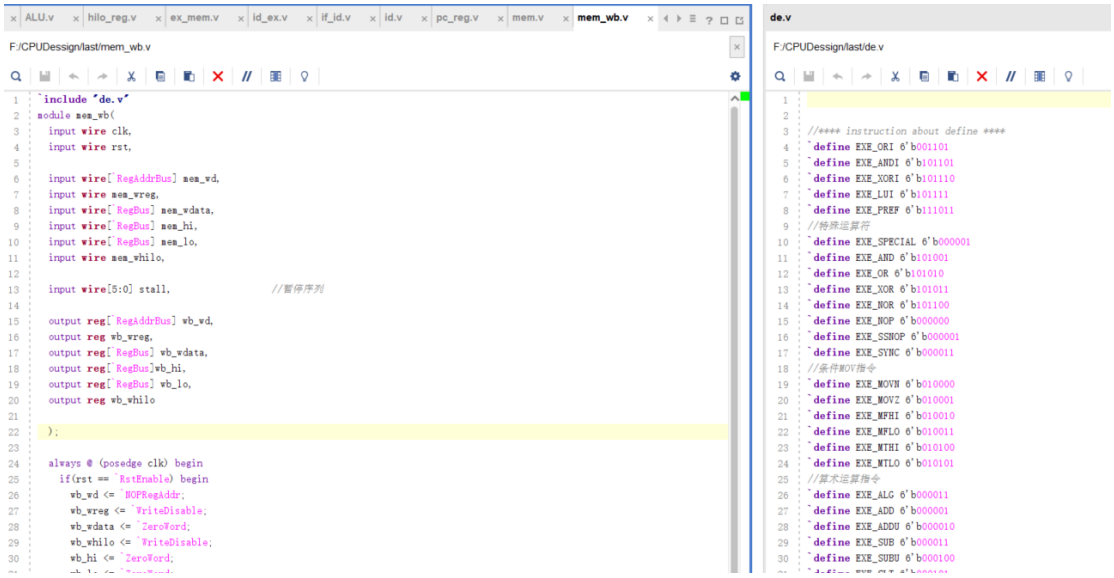
接下来将会依次测试所有指令并截取合适的图片来进行演示

4.2 vivado 窗口

Vivado 的一个很好的优点就是会把在各个模块中调用的其他模块实例整合成一个 Hierachy 资源窗口，这样可以清楚地理清各个模块之间的调用关系，如如下图所示



vivado 同时也支持分屏操作，也让本人的代码效率有了提升，不需要多次切换

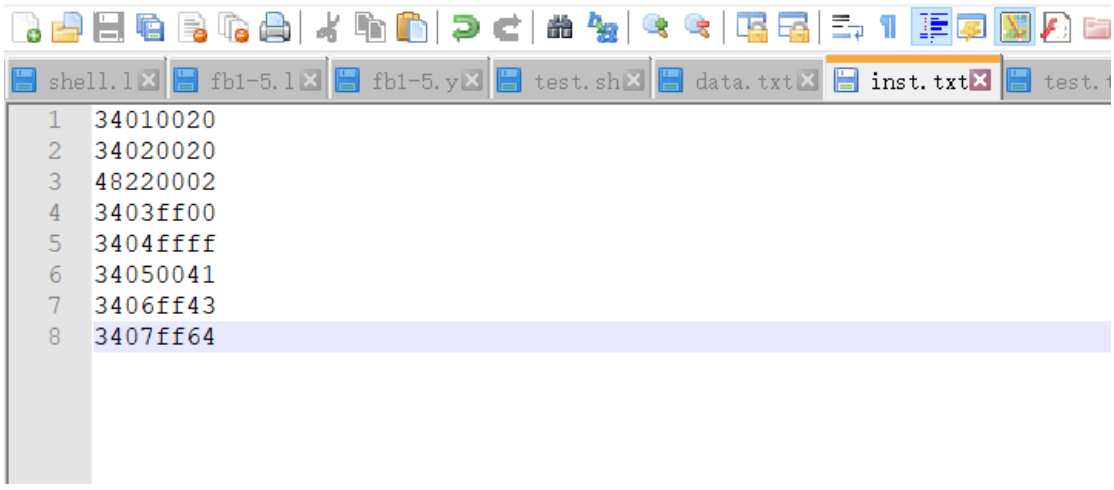


4.3 指令设置

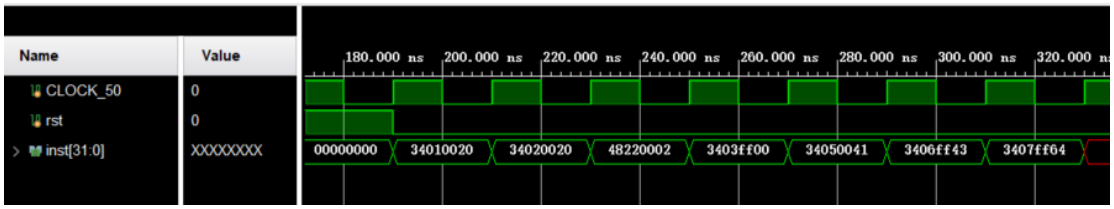
要进行测试，首先要导入指令，在 inst_rom 中有如下代码，用于从指定文件中读取指令，同时支持十六进制或者二进制输入，就是需要在文件里进行修改

```
reg[`InstBus] inst_mem[0:`InstMemNum-1];

  initial $readmemh ("F:/CPUDessign/last/inst.txt", inst_mem);

  //initial $readmemb ("F:/CPUDessign/last/inst.txt", inst_mem);
```

随后在指定的文件里放入要测试的代码，并保存



随后进入项目中点击“Run Simulation”即可进行仿真，从下图可看到指令正常读取

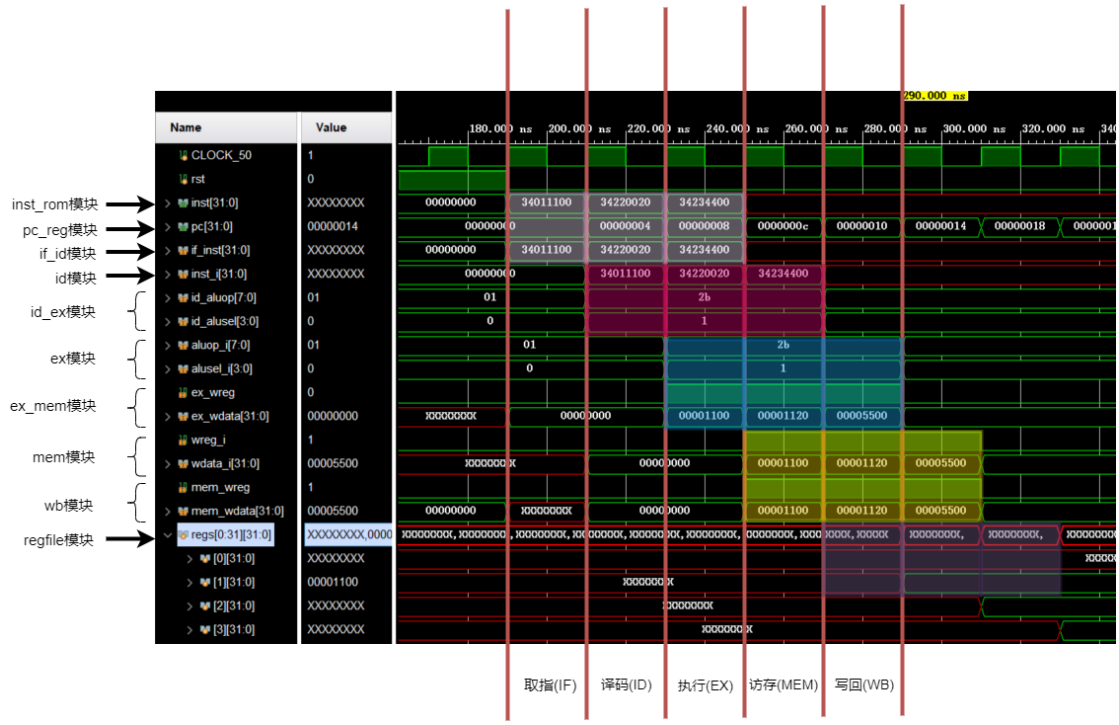


4.4 流水线演示 & 数据前推测试 & ORI 指令测试

首先进行演示的是三条 ori 指令，指令作用是分别将 1 到 3 号寄存器与 0 号寄存器进行或运算，因为 0 号寄存器是全 0.因此就相当于把指令中的立即数赋给 3 个寄存器，指令详情如下

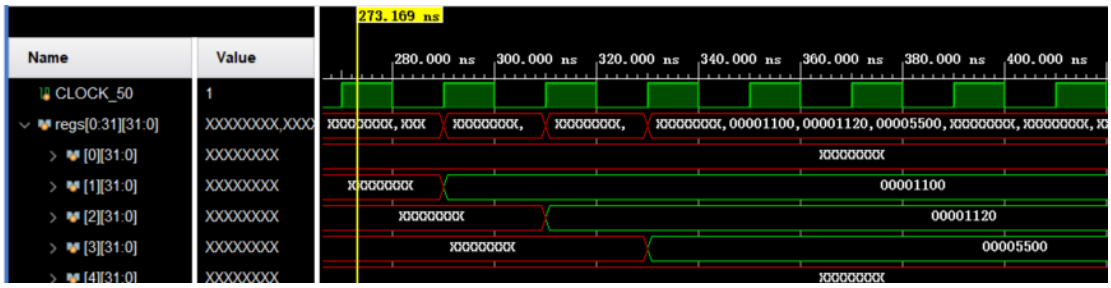
```
ori $1,$0,0x1100

ori $2,$1,0x0020

ori $3,$1,0x4400
```

这里通过截取波形图再使用 draw.io 设计软件加上了部分说明和色彩，很好的体现出了五段流水线



观察指令后可以发现，第一条和第二条指令是存在数据相关的，这种情况下如果没有数据前推会导致第二条指令执行与预期不符（二号寄存器被赋值为 0x0020）.然而在这次测试中可以看到二号寄存器被正确的进行了或运算得到了正确的结果 0x1120，正是因为数据前推解决了这个冲突

并且观察最终寄存器堆中的值，验证了三条指令的执行正确



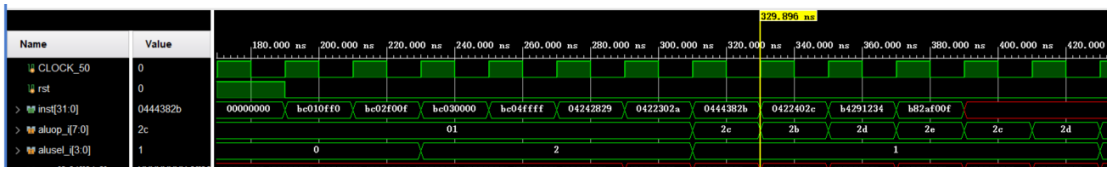
4.5 逻辑运算指令+LUI 指令测试

本次测试包含 andi, and, or, nor, nori, xor 指令,设计测试指令如下

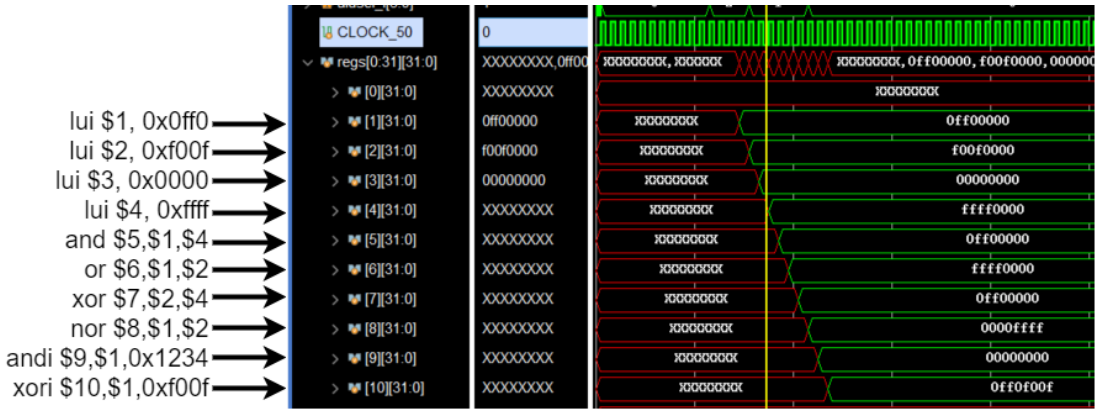
```
lui $1, 0x0ff0

lui $2, 0xf00f

lui $3, 0x0000

lui $4, 0xffff

and $5,$1,$4

or $6,$1,$2

xor $7,$2,$4

nor $8,$1,$2

andi $9,$1,0x1234

xori $10,$1,0xf00f
```

将代码送入指定文件后，启动模拟，由于波形图太过庞大，因此分段进行讲述，首先是从 inst 信号指令都依次正常读取，aluop 信号指的是运算符，而 alusel 指的是运算符种类，代码块中六条逻辑运算的指令都是 1 类，也就是逻辑运算指令

注:为了表示方便这里将寄存器都表示为“n 号”，比如三号寄存器就是 3 号



再从寄存器堆中对各条指令的写回结果进行分析



①前四条指令是 lui 指令，也就是将指令中的立即数存入对应寄存器的高位，根据运算结果来看，四条指令都成功完成了指定的功能存储了数据

②第五条指令是 and 指令，将 1 号和 4 号的值进行与运算，结果存入 5 号

0x0ff0000 与 0xffff0000 进行与运算的结果为 0x0ff00000,可以看出 5 号内容完全一致

③第六条指令是 or 指令，将 1 号和 2 号的值进行或运算,结果存入 6 号

0x0ff0000 与 0xf00f0000 进行与运算的结果为 0xffff0000,可以看出 6 号内容完全一致

④第七条指令是 xor 指令，将 2 号和 4 号的值进行异或运算,结果存入 7 号

0xf00f000 与 0xffff0000 进行与运算的结果为 0x0ff00000,可以看出 7 号内容完全一致

⑤第八条指令是 nor 指令，将 1 号和 2 号的值进行或非运算,结果存入 8 号

0x0ff0000 与 0xf00f0000 进行与运算的结果为 0x0000ffff,可以看出 8 号内容完全一致

⑥第九条指令是 andi 指令，将 1 号与立即数 0x1234 进行与运算,结果存入 9 号 0x0ff0000 与 0x00001234 进行与运算的结果为 0x00000000,可以看出 9 号内容完全一致

⑦第九条指令是 xori 指令，将 1 号与立即数 0xf00f 进行与运算,结果存入 10 号 0x0ff0000 与 0x0000f00f 进行与运算的结果为 0x0ff0f00f,可以看出 10 号内容完全一致

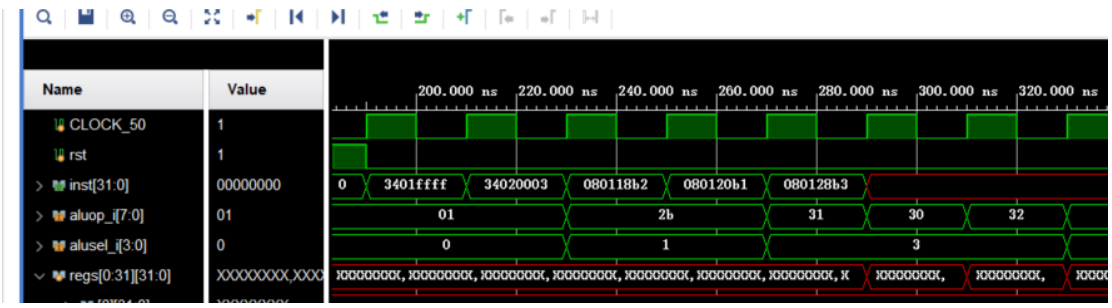
至此，本 CPU 的所有逻辑指令以及 LUI 指令测试演示完毕

4.5 移位指令测试

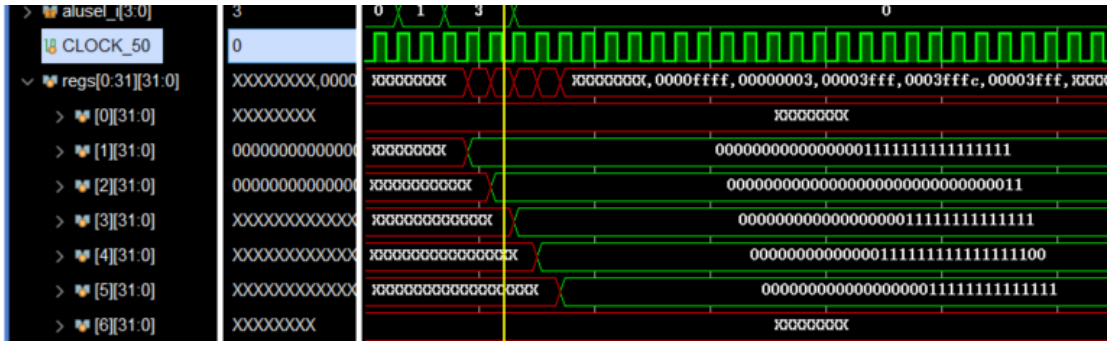
移位指令包括 SLL, SRL, SRA，SLLV, SRLV, SRAV 六条指令，SLLV,SRLV,SRAV 因为只是增添了一个读操作，其他都一样，故只测试 SLL,SRLSRA 指令，设计测试指令如下

```
ori $1,$0,0xffff

ori $2,$0,0x0003

srl $3,$1,00010

sll $4,$1,00010

sra $5,$1,00010
```

将代码送入指定文件后，启动模拟，五条指令正常读取，并且三条移位指令类型显示为 3（移位类指令）



观察寄存器堆如下



三条指令分别是将 1 号寄存器的内容算术左移，算术右移，逻辑右移操作，依次都是移动 3 位，依次存入 3,4,5 号，从 3,4,5 号中的数据观察，移位成功

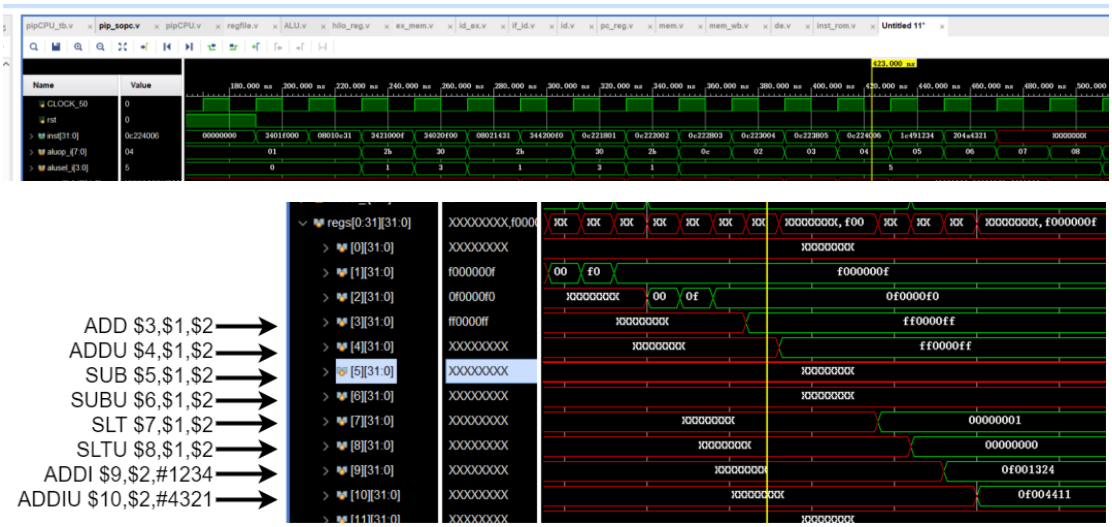
4.6 基本算术运算测试

基本算术运算包括 ADD, ADDU, SUB, SUBU, SLT, SLTU, ADDI, ADDIU 共八条指令，首先将设计好的测试指令导入制定文件，启动模拟，测试代码如下

```
ORI $1,$0,0xff00
SLL $1,$1,16
ORI $1,$1,0xf00ff                    //将 1 号寄存器设置为 0xf000000f
ORI $2,$0,0xff00
SLL $2,$2,16
ORI $2,$2,0xf00ff                    //将 2 号寄存器设置为 0x0f0000f0
ADDU $4,$1,$2
SUB $5,$1,$2
SUBU $6,$1,$2
SLT $7,$1,$2
SLTU $8,$1,$2
ADDI $9,$2,#1234
```

ADDIU $10,$2,#4321

由于代码过多，从波形图可以看出，所有指令都正常读取，并且基本算术指令的指令类型全部为第五类(算术指令)



①前六条指令是寄存器初始化指令，将 1 号和 2 号的值初始化

②第七,八条指令是 add 指令，将 1 号和 2 号的值进行有符号与无符号加法运算，结果存入 3 号和 4 号，0xf000000f 与 0x0f0000f0 进行与运算的结果为 0xff0000ff,可以看出 3 号和 4 号内容完全一致

③第九，十条指令是 sub 与 subu 指令，将 1 号和 2 号的值进行有符号与无符号减法运算,结果存入 5 号和 6 号，因为减法结果溢出，因此并未存入

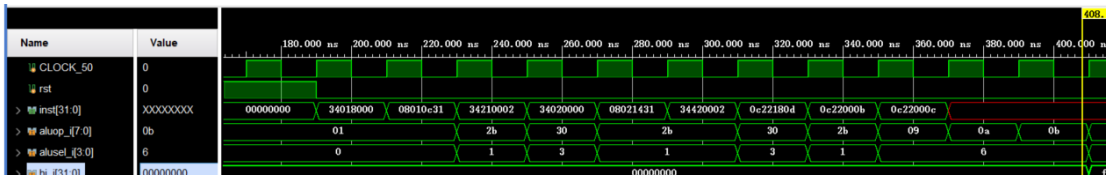
④第 11,12 条指令是 SLT,SLTU 指令，将 1 号与 2 号的值进行有符号与无符号比较，因为 1 号中的值在有符号比较中是负数，因此更小，将 1 存入 7 号。而在无符号比较中，1 号中的数肯定更大，因此将 0 存入 8 号

⑤第 13,14,条指令是 ADDI,ADDIU 指令，将 2 号寄存器分别与 0x00001234 和 0x00004321 相加，所得结果存入 9 号和 10 号

至此，基本算术指令测试演示完成

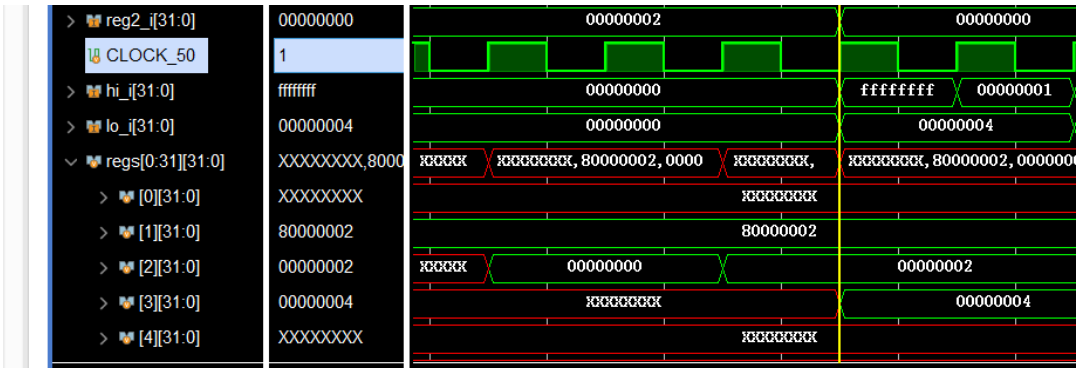
4.6 乘法运算测试

乘法运算包含 MUL,MULT,MULTU 三条指令，将测试好的代码放入文件，并启动模拟从波形图得出所有指令都正常读取，并且三条乘法指令类型全部为第六类(乘法类指令)



略去寄存器初始化命令，三条乘法指令如下图所示

```
MUL $3,$1,$2

MULT $1,$2

MULTU $1,$2
```

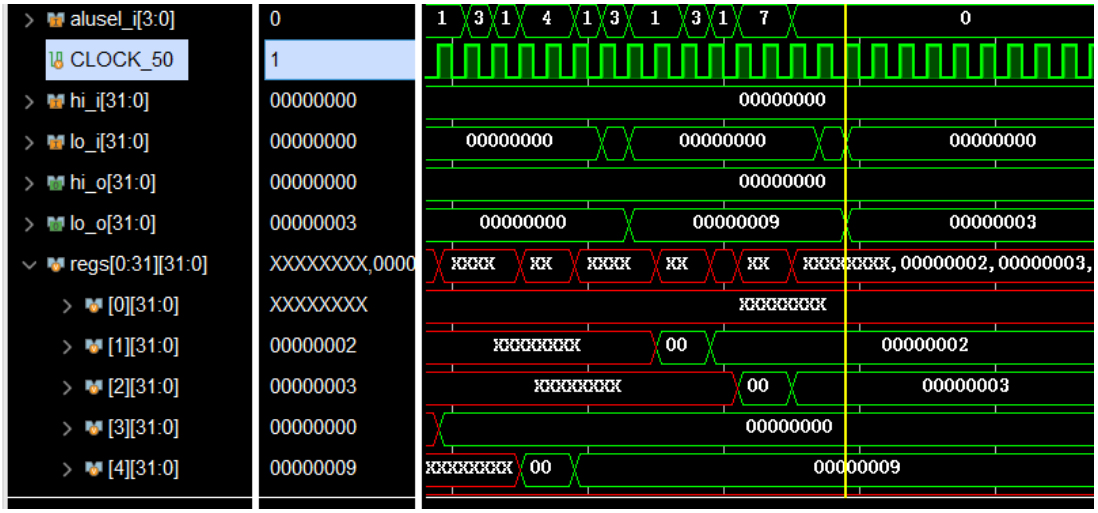


启动仿真，图像如上图所示，可以看到运行到乘法指令时，hi_i 与 lo_i 都显示出了正确的运算结果(0x80000002 * 0x00000002 = 0xffffffff00000004),并且将低 32 位放入了 3 号寄存器中，而 MULTU 指令对 0x80000002 乘以 2，等于 0x0000000100000004 也在 hi_i，lo_i 中有所体现，因此指令测试基本正确

4.7 乘累加，乘累减指令测试

累乘加，累乘减指令实际上包含四条指令，MADD, MADDU, MSUB, MSUBU，这里对其中一条 MSUB 进行测试，将测试指令导入指定文件中，启动仿真，测试指令如下

```
MSUB $1,$2
```



启动模拟之后各个关键信号如上图所示，可以看到 1 号和 2 号寄存器被赋上了初值 2 和 3，而 HI,LO 中的值为{0,9},因此根据乘累加指令运算法则，{HI,LO}<-{HI,LO}-rs*rt, 最终{HI,LO}的值为 3，正好符合运算规则所得结果

4.8 分支预测指令测试

本 CPU 中包含若干条分支预测指令，分别为 BEQ, BNE，BLEZ, BGTZ, BLTZAL, BGEZ, BGEZAL，BLTZ

因为指令过多，全部测试代码过于冗长，因此这里只挑选其中一个指令进行测试

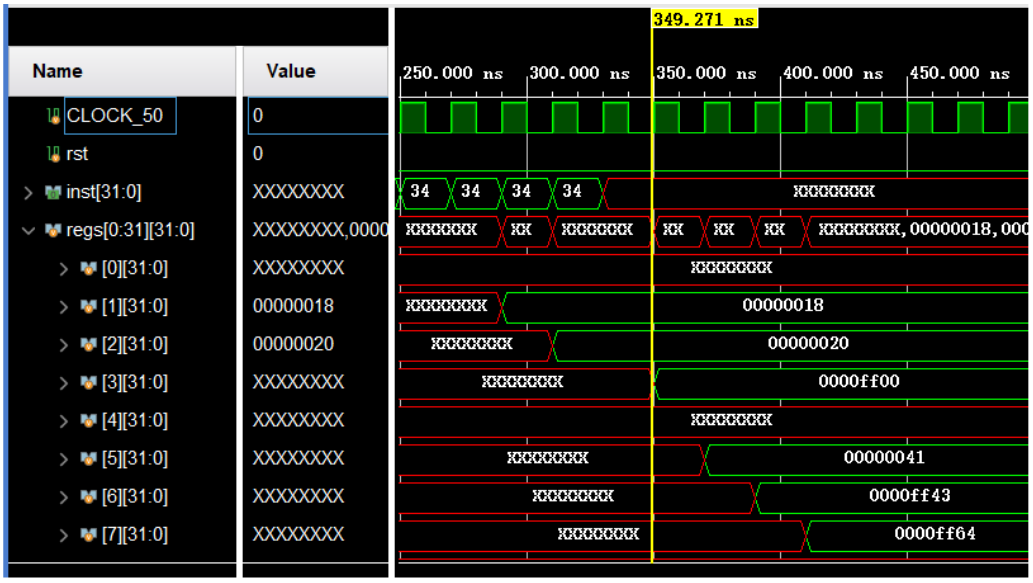
就是 BNE 指令，使用以下指令

```
ori $1,$0,0x0018

ori $2,$0,0x0020

bne $1,$2,2(offset)

ori $3,$0,0x00ff

ori $4,$0,0x0041
```

```
ori $5,$0,0xff43

ori $6,$0,0xff64
```

运行仿真程序后，理论上来说因为 1 号和 2 号的值不相同，条件满足，因此会分支跳转。

执行指令之后波形图如下

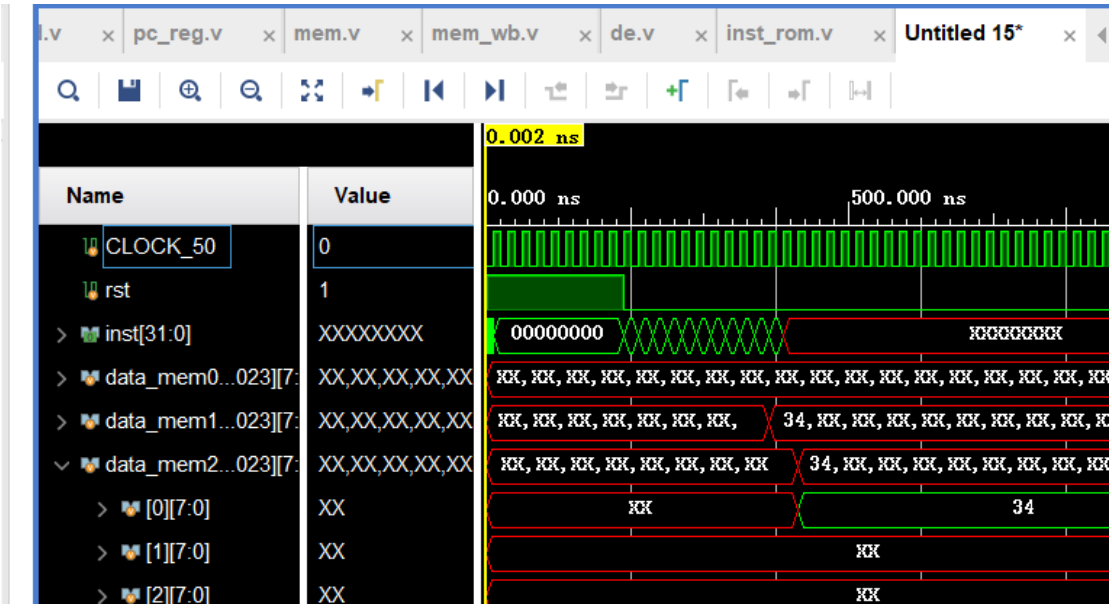


分支指令在第三条，因此 3 号寄存器的写入和 2 号寄存器的写入之间多相差一个周期，而 3 号寄存器写入是因为延迟槽的缘故，并且 4 号寄存器并没有写入数据。并且计算完成之后偏移 2 个指令也是正常的，测试成功

4.9 访存指令测试

本 CPU 中共实现的访存指令有 LB 与 SB，现编写测试代码来测其中的 SB 指令，假定存储的单元地址与读取的单元地址一致，测试代码如下

```
ori $1,$0,0x1234

sb $1,1(0)

sb $2,2(0)
```



因为 1 号寄存器中的值初始化为 0x1234,因此 sb 会将该寄存器中的低字节分别存储到指定位置，最终模拟出来查看 RAM 对应位置，发现两个位置数据写入为 34,证明 SB 指令测试成功

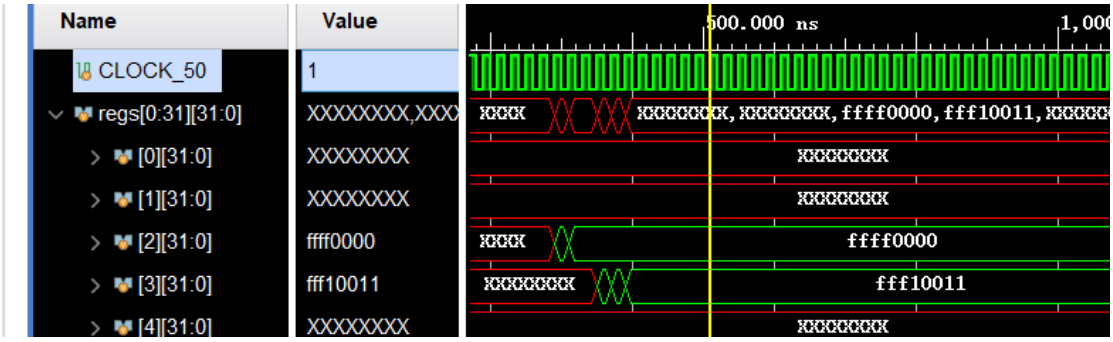
4.10 除法指令测试

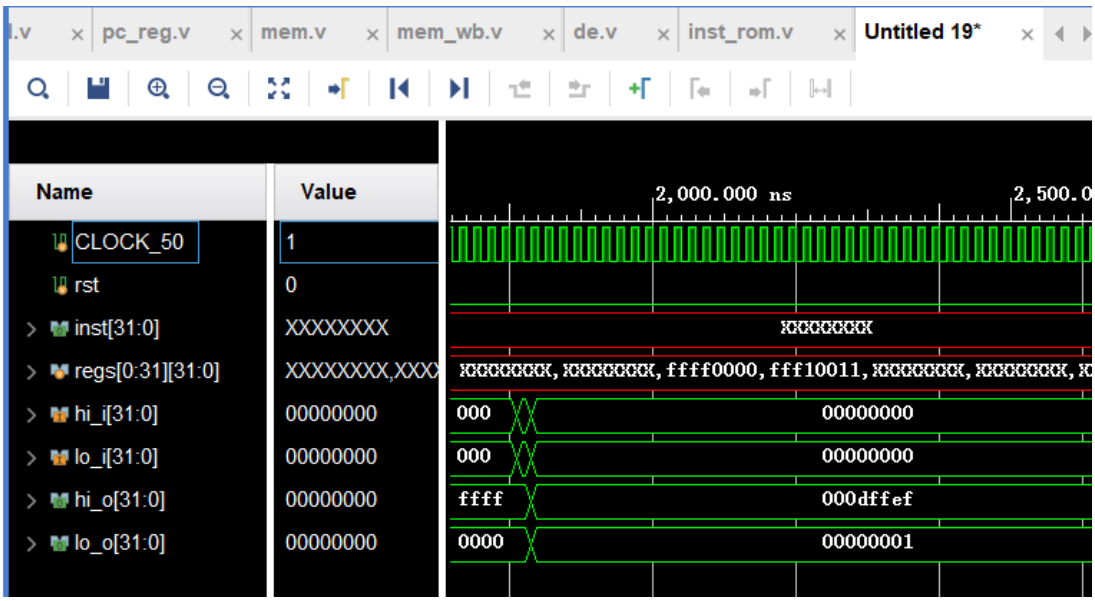
最后进行测试的两条指令是除法指令，因为除法实现起来相对复杂，因此本人也是折腾了很久才最终折腾明白，先调入测试代码（不包括寄存器初始化代码）

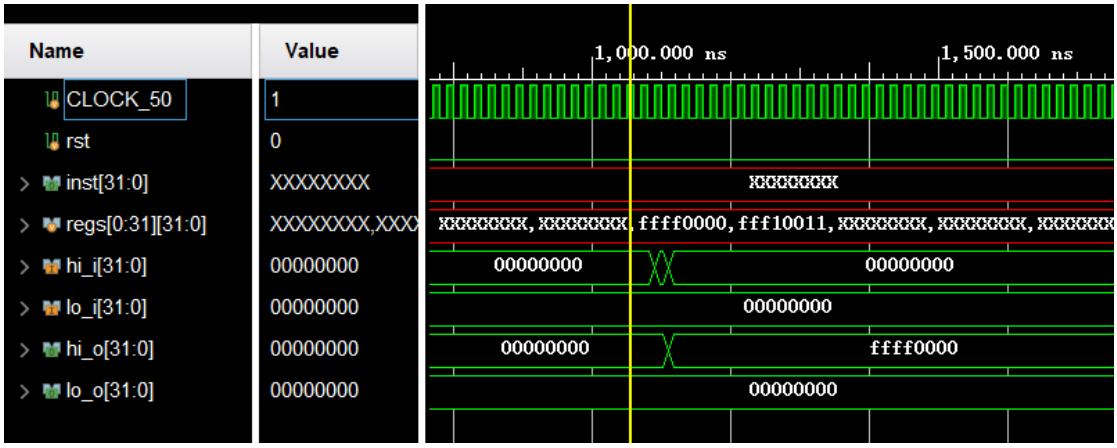
```
div $2,$3

divu $2,$3
```

这两条指令是将 2 号寄存器的内容除以 3 号寄存器的内容，最终会将结果送入{HI,LO}

二号寄存器初始化为 0xffff0000,三号寄存器初始化为 0xfff10011







点击开始仿真之后，可以看到 hi 与 lo 寄存器的值发生了两次改变，这与预估的结果完全符合（值 0xffff000000000000 与 0xffff0000fff10011），验证完成

五、总结

本次 CPU 硬件设计从今年 5 月一直持续到开学，本人从参考之前做的十条简单指令的简单 CPU，一直到后来参考了网上有些地方的简单 CPU 架构，包括 MIPS，risv-v 等架构，一步一步探索，一个模块一个模块设计。每个信号都经过反反复复的测试与调整。并且本人在设计过程中也有时时刻刻记录设计日志，单单是设置日志（包括测试代码等）就超过了 1000 行。

最终设计出来了一个五段流水的 CPU，个人感觉还是挺有成就感的，在写报告的时候光是数据通路就花了超过 1 小时 25 分钟，各模块的详细输入输出信号也花费了不少时间去绘制与阐述。并且本次 CPU 设计还解决了之前上计算机体系结构和计算机组成原理课时候的不少疑惑。就比如说延迟槽，本人在测试的时候还以为延迟槽里的指令执行是代码出现了问题，但是在网上查询并复习了所学知识之后才知道原来延迟槽中的指令也是能被执行的

总而言之，我在这次 CPU 硬件设计中认识学习到了很多，不仅仅是专业知识，同时还有对 modelsim 和 vivado 两个仿真工具的使用以及 verilog 语言的掌握，最重要的是在大三大四一次次不断的课设中本人不断增强自己的代码能力，相信这些锻炼的能力不会白费，在以后更大规模的项目，研究生阶段乃至工作时会有帮助

六、附件

6.1 源代码

souce.zip

6.2 本次实验报告所有的 drawio 绘图文件

drawfile.zip

6.3 高清完整版的数据流图

sjlt.zip