中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 专业 计算机科学与技术

实验项目编号 2 实验名称 简单功能型处理器设计

注 1:	撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下 (注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中 "-" 为英文标点符号的短横线。文件命名举例: `prj5-dma.pdf`。具体要求详见实验项目 5 讲义。
注 2:	使用 `git add` 及 `git commit` 命令将实验报告 `PDF` 文件添加到本地仓库 master 分支, 并通过 `git push` 推送到实验课 SERVE GitLab 远程仓库 master 分支(具体命令详见实验报告)。
注 3:	实验报告模板下列条目仅供参考, 可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明(比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

- **ALU 设计的修改**。

本部分要求修改 project1 中完成的 ALU 模块, 要求增加 3 种逻辑操作: 逻辑按位异或(XOR)、逻辑按位或非(NOR)、无符号数整数比较(SLTU)(A < B 时, Result = 1; A B 时, Result = 0)。对应的 ALUop 分别为 100, 101, 011。

由于 XOR 和 NOR 操作比较简单, 直接修改 result 即可。

SLTU 操作则需要先判断无符号数 A 和 B 的大小, 再将 result 的最低位按照结果来进行写。我们仍然采用 A-B 来判断两个数的大小, 当 A<B 时结果应该为负, 但是无符号数不能表示正负, 则可以用是否借位来表示正负, 当减法 CarryOut=1 时, 表示减法的借位, 即 A<B, CarryOut=2 时, 不借位, 即 A>=B。

```
1  //求得比较的结果：若最高位的值与符号位相同，则A>=B，否则A<B
2  assign re_SLT = {31'b0,(re_ADDSUB[31] ^ Overflow)};
3  //无符号数比较：若小于，则置1，否则置0
4  assign re_SLUT = {31'b0,CarryOut};
```

图 1: SLT 和 SLTU 的代码

• **Shifter 设计**。

本部分要求设计移位运算部件，实现三类移位操作：左移、算术右移、逻辑右移。其中左移和逻辑右移都是相应的补 0 即可；而算数右移则需要补符号位，即负数右移在左侧补 1，正数右移在左侧补 0。

本部分也比较简单，只需要设计类似于 ALUop 的控制信号 Shiftop，然后根据 Shiftop 信号控制相应的移位操作即可。

```
1  assign Result = ({32{LS}} & A << B) |
2                  ({32{ARS}} & $signed(arithmetic_shift >>> B)) |
3                  ({32{LRS}} & A >> B);
```

图 2: Shifter 移位器结果部分代码

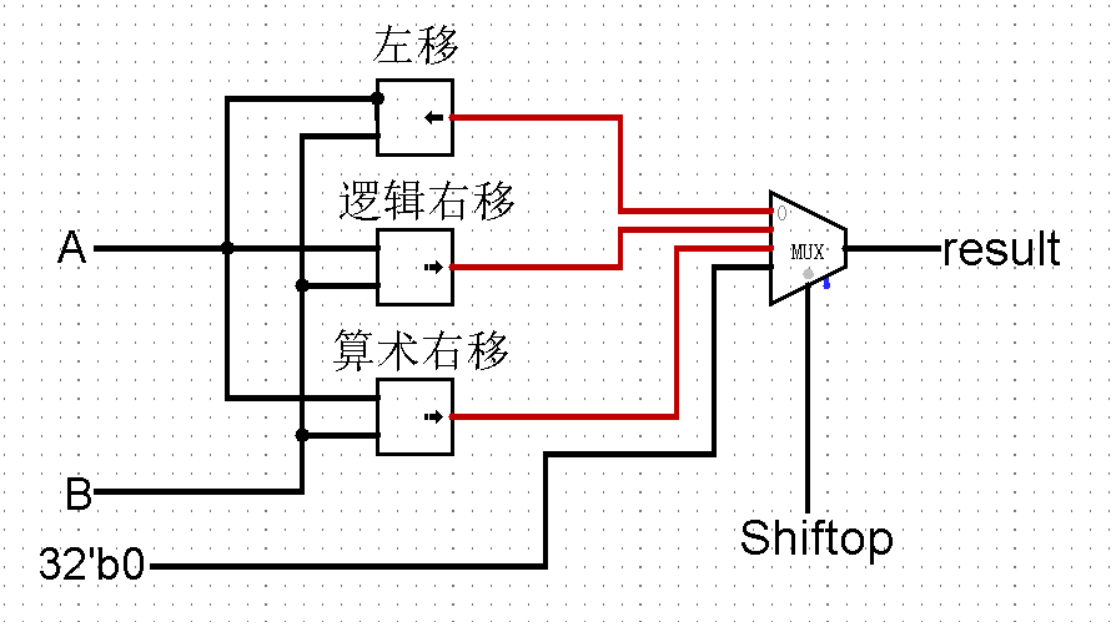


图 3: Shifter 移位器逻辑电路图

• **Simple_CPU 设计**。

此部分的设计是本实验中工作量最大也是最难的一部分。要求我们根据设计好的寄存器，ALU，Shifter，以及给定的理想内存来完成整个单周期 CPU 的设计。该 CPU 需要完成 45 条 MIPS 指令，因此，将这 45 条指令的功能以及共性了解清楚是首要的任务。

指令分为四大类，分别为 R-Type,REGIMM,J-Type,I-Type，不同类型指令各不相同，我们需要阅读 MIPS 指令集，并根据其特点，将这些指令之间的关联利用起来，从而避免通过逐一比对来实现这 45 条指令。

首先应当了解的是各个指令的指令码：

可以看出所有 R-Type 指令的 opcode 都是 6'b0，因此在设计译码表的时候应该根据其 func 码的不同以及不同指令 func 码的关系来设计。

指令类型	指令分类	序号	指令名	指令码					
				opcode(6-bit)	rs(5-bit)	rt(5-bit)	rd(5-bit)	shamt(5-bit)	func(6-bit)
R-Type	计算指令	1	addu	000000	rs	rt	rd	00000	100001
		2	subu						100011
		3	and						100100
		4	or						100101
		5	xor						100110
		6	nor						100111
		7	slt						101010
		8	sltu						101011
	移位指令	9	sll	000000	00000	rt	rd	sa	000000
		10	sra						000011
		11	srl						000010
		12	sllv		rs			00000	000100
		13	srav						000111
		14	srlv						000110
	跳转指令	15	jr	000000	rs	00000	00000	00000	001000
		16	jalr				rd		001001
	mov指令	17	movz	000000	rs	rt	rd	00000	001010
		18	movn						001011

图 4: R-Type 指令集指令码

指令类型	指令分类	序号	指令名	指令码			
				opcode(6-bit)	rs(5-bit)	REG(5-bit)	imm(16-bit)
REGIMM	跳转指令	19	bltz	000001	rs	00000	offset
		20	bgez			00001	
指令类型	指令分类	序号	指令名	指令码			
				opcode(6-bit)	instr_index(25-bit)		
J-Type	跳转指令	21	j	000010	instr_index		
		22	jal	000011			

图 5: REGIMM 指令集指令码和 J-Type 指令集指令码

可以看出 REGIMM 指令的 opcode 都为 6'b000001,但由于两条指令的功能可以用同一种操作完成,所以在译码时可直接根据 opcode 进行选择。

J-Type 指令的 opcode 高五位相同,为 5'b00001,而这两条指令与 REGIMM 的两条指令功能类似,可以一起译码。

指令类型	指令分类	序号	指令名	指令码			
				opcode(6-bit)	rs(5-bit)	rt(5-bit)	imm(16-bit)
I-Type	分支指令	23	beq	0001 00	rs	rt	offset
		24	bne	0001 01			
		25	blez	0001 10		00000	
		26	bgtz	0001 11			
	计算指令	27	addiu	001001	rs	rt	imm
		28	lui	001111			
		29	andi	001100			
		30	ori	001101			
		31	xori	001110			
		32	slti	001010			
		33	sltiu	001011			
	内存读指令	34	lb	100 000	base	rt	offset
		35	lh	100 001			
		36	lw	100 011			
		37	lbu	100 100			
		38	lhu	100 101			
		39	lwl	100 010			
		40	lwr	100 110			
	内存写指令	41	sb	101 000	base	rt	offset
		42	sh	101 001			
		43	sw	101 011			
		44	swl	101 010			
		45	swr	101 110			

图 6: I-Type 指令集指令码

I-Type 指令的 opcode 与其属于的指令分类有关,分支指令高四位为 4'b0001,计算指令高三位为 3'b001,内存读指令高三位为 3'b100,内存写指令高三位为 3'b101,可以根据各自功能的关联以及相同的高位 opcode 码来

进行译码。

接着我们需要了解各个指令所能实现的功能,知道每个指令需要调用哪些模块,从而将其分类。

指令类型	指令含义
R-Type	rd <- rs + rt (不进行溢出检查)
	rd <- rs - rt (不进行溢出检查)
	rd <- rs && rt (逻辑与运算)
	rd <- rs || rt (逻辑或运算)
	rd <- rs ^ rt (逻辑异或运算)
	rd <- !(rs || rt) (逻辑或非运算)
	rd <- (rs < rt) (有符号比较, 小于[rd]置1, 大于等于[rd]置0)
	rd <- (rs < rt) (无符号比较, 小于[rd]置1, 大于等于[rd]置0)
	rd <- rt << sa (逻辑左移)
	rd <- rt >>> sa (算术右移)
	rd <- rt >> sa (逻辑右移)
	rd <- rt << rs[4:0] (逻辑左移, 左移位数由{rs}0-4bit决定)
	rd <- rt >>> rs[4:0] (算术右移, 右移位数由{rs}0-4bit决定)
	rd <- rt >> rs[4:0] (逻辑右移, 右移位数由{rs}0-4bit决定)
	pc <- rs (将{rs}赋给PC, 作为新指令地址)
	rd <- return address, pc <- rs (将{rs}赋给PC, 作为新指令地址, 同时将跳转指令后面第二条指令的地址作为返回地址保存到{rd}, 若没有在指令中指明rd(rd为0), 则默认将返回地址保存到寄存器31中)
	if rt = 0 then rd <- rs (条件数据转移)
	if rt ≠ 0 then rd <- rs (条件数据转移)
指令类型	指令含义
REGIMM	if rs < 0 then branch (条件跳转)
	if rs ≥ 0 then branch (条件跳转)
指令类型	指令含义
J-Type	pc <- {(pc+4)[31,28], instr_index, 00} (转移到新地址, 新地址低28位为instr_index左移两位, 高4位为跳转指令后指令地址的高4位)
	pc <- {(pc+4)[31,28], instr_index, 00} (转移到新地址, 新地址低28位为instr_index左移两位, 高4位为跳转指令后指令地址的高4位, 同时将跳转指令后第二条指令的地址作为返回地址保存到寄存器31)

图 7: R-Type、REGIMM、J-Type 指令集指令含义

从表中的指令含义我们可以推断出每条指令会调用哪些模块,例如 R-Type 计算指令都需要调用 ALU,所有指令都不需要进行内存访问,只需要寄存器变量即可等等。

指令类型	指令含义
I-Type	if rs = rt then branch (条件跳转)
	if rs ≠ rt then branch (条件跳转)
	if rs ≤ 0 then branch (条件跳转)
	if rs > 0 then branch (条件跳转)
	rt <- rs + (arithmetic_extend)immediate (不进行溢出检查)
	{rt <- immediate,16'b0} (将16bit立即数保存到[rt]高16位, 低16位用0填充)
	rt <- rs && {16'b0,immediate} ({rs}与立即数零扩展为32位后的值进行逻辑与)
	rt <- rs || {16'b0,immediate} ({rs}与立即数零扩展为32位后的值进行逻辑或)
	rt <- rs ^ {16'b0,immediate} ({rs}与立即数零扩展为32位后的值进行逻辑异或)
	rt <- (rs < {16'b0,immediate}) ({rs}与立即数零扩展为32位后的值进行有符号比较, 小于[rt]置1, 否则置0)
	rt <- (rs < {16'b0,immediate}) ({rs}与立即数零扩展为32位后的值进行无符号比较, 小于[rt]置1, 否则置0)
	rt <- memory[base + offset] (从内存中指定的加载地址读取8-bit, 符号扩展至32位, 保存到[rt])
	rt <- memory[base + offset] (从内存中指定的加载地址读取16-bit, 符号扩展至32位, 保存到[rt], 有地址对齐要求, 要求加载地址最低为为0)
	rt <- memory[base + offset] (从内存中指定的加载地址读取32-bit保存到[rt])
	rt <- memory[base + offset] (从内存中指定的加载地址读取8-bit, 零扩展至32位, 保存到[rt])
	rt <- memory[base + offset] (从内存中指定的加载地址读取16-bit, 零扩展至32位, 保存到[rt], 有地址对齐要求, 要求加载地址最低为为0)
	rt <- rt MERGE memory[base + offset] (从内存中指定的加载地址处, 找到该地址所在的存储字(加载地址后两位为0), 从加载地址开始读取知道存储字结束, 将读到的值保存到{rt}的首端, 其余保持不变)
	rt <- rt MERGE memory[base + offset] (从内存中指定的加载地址处, 找到该地址所在的存储字(加载地址后两位为0), 从加载地址开始读取知道存储字结束, 将读到的值保存到{rt}的尾端, 其余保持不变)
	memory[base + offset] <- rt (将{rt}最低字节存储到内存的指定地址)
	memory[base + offset] <- rt (将{rt}最低两个字节存储到内存的指定地址, 有地址对齐要求, 要求加载地址最低位为0)
	memory[base + offset] <- rt (将{rt}存储到内存的指定地址, 有地址对齐要求, 要求加载地址最低位为0)
	memory[base + offset] <- memory[base + offset] MERGE rt (类似lwl, 非对齐存储指令, 向左存储, 将{rt}存储到内存中的指定地址)
	memory[base + offset] <- memory[base + offset] MERGE rt (类似lwr, 非对齐存储指令, 向右存储, 将{rt}存储到内存中的指定地址)

图 8: I-Type 指令集指令含义

从表中我们可以看出只有内存读写指令涉及到了内存的访存。

接下来就是需要根据 MIPS 指令集中相关指令对模块的调用接口进行判断,重点是根据每条指令的含义来进行判断,部分指令在讲义中交代的比较清楚,我们很容易写出,有些没有访问某些模块的指令,则需要根据其含义自行判断。

从表中我们可以看出,R-Type 指令与寄存器的接口对应是:rs–raddr1,rt–raddr2, 与 ALU 接口对应与指令分类有关,而跳转指令和 mov 指令则不需要调用 ALU 模块。

指令类型	寄存器堆读		ALU/shifter			内存访问				
	raddr1	raddr2	A	B	ALUop	Addres	MemRead	MemWrit	Write_Data	Write_strb
R-Type	rs	rt	rdata1	rdata2	ALUop 译码表	/				
	rs	rt	rdata2(shifter)	sa(shifter) 5'b0	Shiftop 译码表	/				
	rs	00000	/			/				
	rs	rt	/			/				
指令类型	寄存器堆读		ALU/shifter			内存访问				
	raddr1	raddr2	A	B	ALUop	Addres	MemRead	MemWrit	Write_Data	Write_strb
REGIMM	rs	/	rdata1	32'b0	SLT	/				
指令类型	寄存器堆读		ALU/shifter			内存访问				
	raddr1	raddr2	A	B	ALUop	Addres	MemRead	MemWrit	Write_Data	Write_strb
J-Type	/		/			/				

图 9: R-Type、REGIMM、J-Type 指令集与寄存器读模块、ALU 模块和内存模块的接口

REGIMM 指令是跳转指令，因此只需要将 rs 的内容传递给 PC 即可，所以只要将 rs 与 raddr1 连接即可，通过寄存器后的输出 rdata1 与 ALU 的 A 输入相连，寄存器的输出 rdata2 为 32'b0，与 ALU 的 B 输入相连。

J-Type 指令也是跳转指令，但是其 PC 值只与立即数有关，因此不调用寄存器和 ALU。

指令类型	寄存器堆读		ALU/shifter			内存访问				
	raddr1	raddr2	A	B	ALUop	Addres	MemRead	MemWrit	Write_Data	Write_strb
I-Type	rs	rt或5'b0	rdata1 32'b0	rdata2或32'b0 rdata1	ALUop 译码表	/				
	rs	/	rdata1	immediate	ALUop 译码表	/				
	base	rt	rdata1	immediate	ALUop 译码表	ALU_result	opcode[5] & ~opcode[3]	opcode[5] & opcode[3]	rdata2	Write_strb 译码表
	base	rt	rdata1	immediate	ALUop 译码表	ALU_result	opcode[5] & ~opcode[3]	opcode[5] & opcode[3]	rdata2	Write_strb 译码表

图 10: I-Type 指令集与寄存器堆读模块、ALU 模块和内存模块的接口

I-Type 分支指令实现的功能和 REGIMM 指令类似，因此有相似的接口。同理，计算指令与 R-Type 计算指令类似。

而内存读写指令则是所有指令中唯二与内存相关的指令类型。根据其 opcode 的特点，设计出了相应如表中所示的内存读写使能信号。

指令类型	跳转		寄存器堆写		
	跳转地址	地址更新条件	wen	waddr	wdata
R-Type	/	0	1	rd	ALU_result
	/	0	1	rd	Shifter_result
	rdata1	1	0	rd	PC+8
			1	rd/31	
	/	0	func[0] ^ (rdata2 == 32'b0)	rd	rdata1
指令类型	跳转		寄存器堆写		
	跳转地址	地址更新条件	wen	waddr	wdata
REGIMM	{14'b(offset[15]),(offset<<2),00} + (PC+4)	REG[0] ^ ~Zero	0	rd	/
指令类型	跳转		寄存器堆写		
	跳转地址	地址更新条件	wen	waddr	wdata
J-Type	{(PC+4)[31,28],instr_index,00}	1	opcode[0]	31	PC+8

图 11: R-Type、REGIMM、J-Type 指令集与跳转模块和寄存器堆写模块的接口

R-Type 计算指令、移位指令和 mov 指令与跳转无关,即不参与 PC 的更改功能。其余类型指令的跳转地址与其指令自身含义有关。

每种指令写入寄存器中的数据和指令类型有关,将各自指令所实现的操作结果写入寄存器中。一般都存到 rd 寄存器中,只有跳转指令需要将返回地址写入 rd 寄存器且 rd 寄存器为 0 时才会写入 31 号寄存器。

指令类型	跳转		寄存器堆写		
	跳转地址	地址更新条件	wen	waddr	wdata
I-Type	{14'b(offset[15]),(offset<<2),00} + (PC+4)	opcode[0] ^ Zero	0	/	
	/	0	1	rt	ALU_result
	/	0	1	rt	Read_result
	/	0	0	/	

图 12: I-Type 指令集与跳转模块和寄存器堆写模块的接口

I-Type 中只有分支指令与跳转有关,具体条件与指令含义有关,和 REGIMM 指令类似。且只有计算指令和内存读指令需要将数据写入寄存器堆中,即将 ALU 运算的结果或者从内存中读出的数据存入寄存器中。

根据上述指令集表的表述,下面为各类操作的译码表,主要是根据指令的含义进行组合。

ALUop译码表					
序号	指令	opcode	func	ALUop	译码
1	addu	000000	100001	010	ADD/SUB：func[3:2] == 2'b00; ALUop = {func[1],2'b10}
2	subu		100011	110	
3	and		100100	000	逻辑运算：func[3:2] == 2'b01; ALUop = {func[1],1'b0,func[0]}
4	or		100101	001	
5	xor		100110	100	
6	nor		100111	101	
7	slt		101010	111	比较运算：func[3:2] == 2'b10; ALUop = {~func[0],2'b11}
8	sltu		101011	011	
9	addiu	001001	/	010	ADD: opcode[2:1] == 2'b00; ALUop = 3'b010
10	andi	001100		000	逻辑运算：opcpde[2] == 1'b1 && opcode[1:0]!=2'b11; ALUop = {opcode[1],1'b0,opcode[0]}
11	ori	001101		001	
12	xori	001110		100	
13	lui	001111		/	单独处理
14	slti	001010		111	比较运算：opcode[2:1] == 2'b01; ALUop = {~opcode[0],2'b11}
15	sltiu	001011		011	
16	bltz	000001		111	比较运算：opcode == 6'b000001 || opcode[5:1] == 5'b00011; ALUop = 3'b111
17	bgez				
18	blez	000110			
19	bgtz	000111			
20	beq	000100		110	SUB: opcode[5:1] == 5'b00010; ALUop = 3'b110
21	bne	000101			
22-28	内存读写	opcode[5]=1		010	ADD：opcode[5] == 1'b1; ALUop = 3'b010

图 13: ALU 译码表

Jump译码表					
序号	指令	opcode	func	Jump	译码
35	jr	000000	001000	1	opcode == 6'b0 && func[5:3] == 3'b001 && func[1] == 0; Jump = 1
36	jalr		001001		
37	j	000010	/		opcode[5:1] == 5'b00001; Jump = 1
38	jal	000011			

图 14: Shifter 译码表

Write_strb译码表				
序号	指令	opcode	Write_strb	译码
39	sb	101000	0001/0010/0100/1000	4'b1000 >> (~ALU_result[1:0])
40	sh	101001	0011/1100	{{2{ALU_result[1]}},{2{~ALU_result[1]}}}
41	sw	101011	1111	4'b1111
42	swl	101010	0111/0011/0001	{ALU_result[1] & ALU_result[0],ALU_result[1], ALU_result[1] | ALU_result[0],1'b1}
43	swr	101110	1110/1100/1000	{1'b1,(~ALU_result[1]) | (~ALU_result[0]), ~ALU_result[1],(~ALU_result[1]) & (~ALU_result[0])}

图 15: Jump 操作译码表

Branch译码表					
序号	指令	opcode	func	Branch	译码
16	bltz	000001	/	1	opcode == 6'b000001 && (rt[0] ^ ALU_result[0])
17	bgez				
18	blez	000110			opcode[5:2] == 4'b0001 && (opcode[0] ^ Zero)
19	bgtz	000111			
20	beq	000100			
21	bne	000101			

图 16: Branch 操作译码表

Shiftop译码表					
序号	指令	opcode	func	Shiftop	译码
29	sll	000000	000000	00	func[5:3] == 3'b000; Shiftop = func[1:0]
30	sra	000000	000011	11	
31	srl	000000	000010	10	
32	sllv	000000	000100	00	
33	srav	000000	000111	11	
34	srlv	000000	000110	10	

图 17: 内存写操作译码表

下面我们将分别对重点部分的内容实现其电路图。

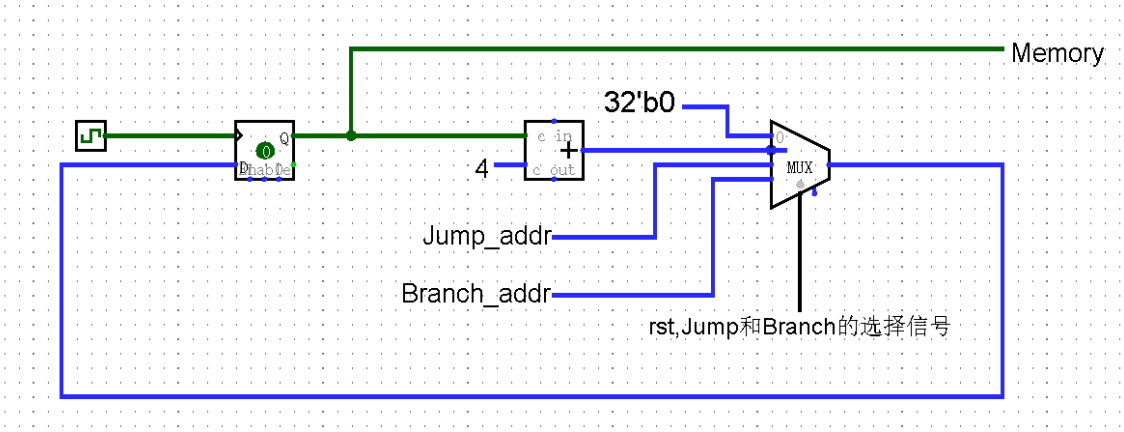


图 18: PC 时序电路部分电路示意图

PC 在时钟上升沿到来时会根据前一时钟周期的结果进行更新下一指令地址，具体是若 rst=1，则同步置零；若 Jump=1，则下一指令地址为 Jumpaddr；若 Branch=1，则下一指令地址为 Branchaddr；若都不是，则正常 PC+4。同时将根据指令地址进行访存取指。

```
//PC_next
assign PC_next  = PC + 4;

//时序逻辑电路
always@(posedge clk) begin
    if (rst) begin
        PC <= 32'b0;                          //rst同步的高电平复位信号
    end
    else begin
        PC <= Jump ? Jump_addr : (Branch ? Branch_addr : PC_next);  //根据操作不同，选择相应的下一条指令的PC地址
    end
end
```

图 19: PC 时序电路部分代码

上述代码中，我将跳转指令分为 Jump 和 Branch 两个指令控制，因为两类指令虽然都与 PC 下一周期指令地址有关，但是译码条件不相同，所有分开译码。

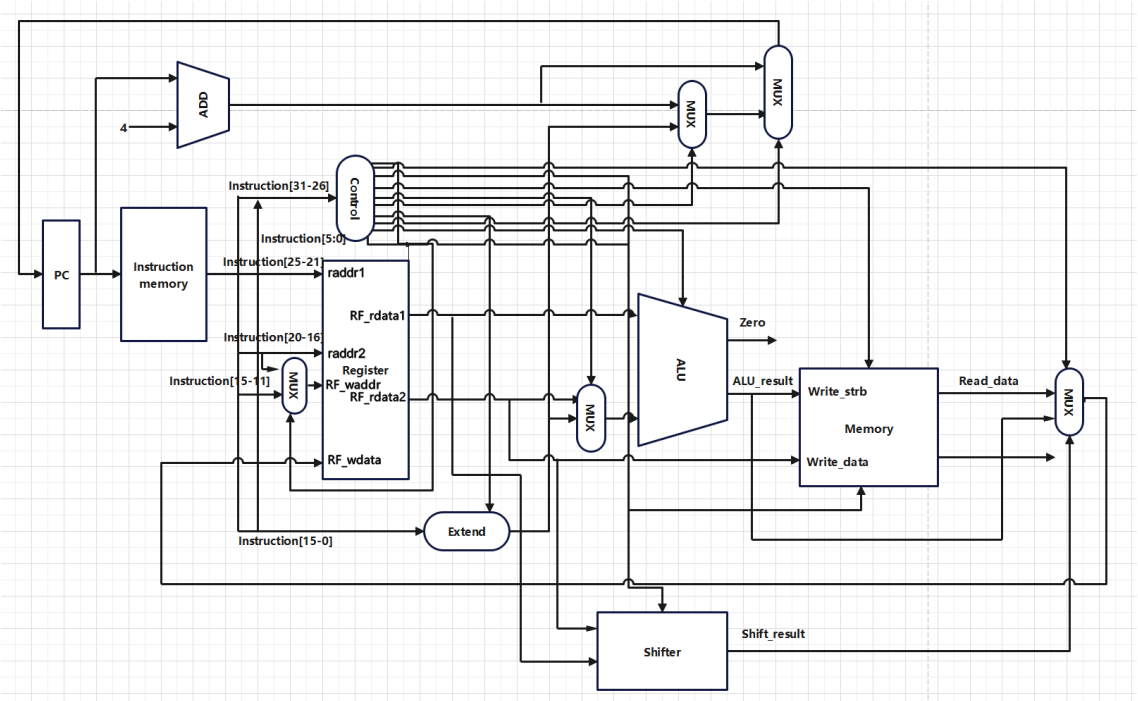


图 20: CPU 整体电路示意图

上图为对每个部件封装后组合而成的整体的 CPU 电路示意图，与课件上的绝大部分相同，但又有细微差别，比如将整个控制系统放在 Conrtol 模块当中，而课件上则是分成总 Conrtol 和 ALUcontrol 两部分。

整体 CPU 的运行分为五个部分，分别是取指，译码，执行，访存，写回。

```
//实例化
reg_file reg_file_module(
    .clk(clk),
    .waddr(RF_waddr),
    .raddr1(rs),
    .raddr2(rt),
    .wen(RF_wen),
    .wdata(RF_wdata),
    .rdata1(RF_rdata1),
    .rdata2(RF_rdata2)
);
alu alu_module(
    .A(ALU_A),
    .B(ALU_B),
    .ALUop(ALUop),
    .Result(ALU_result),
    .Overflow(),
    .CarryOut(),
    .Zero(Zero)
);
shifter shifter_module(
    .A(RF_rdata2),
    .B(Shifter_B),
    .Shiftop(Shiftop),
    .Result(Shift_result)
);
```

图 21: CPU 各模块实例化接口设计

代码中介绍了各模块实例化接口对应，方便在电路图中理解各路线的含义。

关于具体的译码表的设计以及各模块之间的具体连接在代码中体现的并不是特别明显，主要的原因是大量的三目运算符的设计，目前没有想到较好的代替在许多选择情况下用别的方法能够较好的将这些进行选择的方法。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法(比如 RTL 代码中出现的逻辑 bug,逻辑仿真和 FPGA 调试过程中的难点等)

• **Simple_CPU 设计问题**。

在设计代码的过程中遇到了很多问题,具体分为以下几类:

1. 如何将众多的指令按照其特点进行译码。
2. 各个模块之间具体该怎么连接。
3. 具体实现的过程中遇到的问题以及仿真过程中遇到的错误。

1. 对于译码的问题,主要是由从内存中读出的 Instruction 来确定的,而不同类型的指令在分解 Instruction 的时候会将其分解为不同的部分。有的部分是作为寄存器的读写地址,比如 R-Type 和 I-Type 指令都有 rs 和 rt 部分,作为寄存器的读地址 raddr1, raddr2 的接口。R-Type 有 rd 部分,作为寄存器写地址 waddr 的接口;而 REGIMM 指令只需要一个读地址,因为另一个操作数是立即数 32'b0;J-Type 则不需要访问寄存器,而是直接与 PC 值以及 Instruction 的立即数段相关。

由于所有指令高六位都是 opcode,所以我们主要根据这部分来进行译码,但是 R-Type 的 opcode 都是 6'b0,因此我们还需要额外的 Instruction 部分来区别不同的 R-Tpye 类型指令。我们注意到 R-Type 指令 Instruction 的低六位是 func 段,这是其他指令没有的,因此我们可以根据这个独特的部分来进行辅助译码。具体的译码方式在讲义中已经给我们提示了。

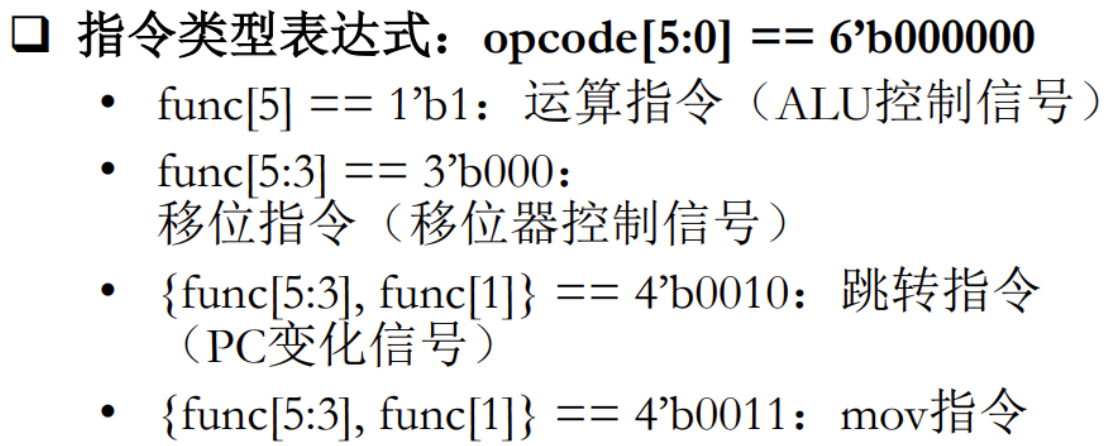


图 22: R-Type 类型指令译码方式

其中运算指令还需要细分,根据 func 其他位的数来进行译码,从而选择相应的 ALUop,进行所需要的 ALU 运算。具体见图 13: ALU 译码表。移位指令也需要根据 func 的其他位进行细分,分为左移、算术右移和逻辑右移。剩余两类只分别有两条指令,根据功能区别进行选择即可。

REGIMM 指令也是需要通过 ALU 的 SLT 操作进行完成的,因此也要将其加入到 ALUop 的译码当中,由于其跟 32'b0 进行比较,所以我们在设置 ALU 的操作数的时候需要单独判断一下这类指令的某一个操作数是 32'b0。

J-Type 指令与 R-Type 跳转指令功能类似,都是对 PC 值进行操作,但是 JALR 和 JAL 指令还需进行 31 号寄存器写,需要在寄存器堆写部分单独再添加。

I-Type 指令类型丰富,但都需要进行 ALU 的运算,在讲义中也有体现。

<table>
<tr><th>R-Type 指令</th><th>func （6-bit）</th><th>ALUop （3-bit）</th><th>ALUop编码</th><th>opcode （6-bit）</th><th>I-Type 指令</th><th>ALUop编码</th></tr>
<tr><td>add</td><td>10 00 00</td><td rowspan="2">010</td><td rowspan="3">ADD/SUB：func[3:2] == 2’b00
ALUop = {func[1], 2’b10}</td><td>001 0 00</td><td>addi</td><td rowspan="2">ADD：opcode[2:1] == 2’b00
ALUop = {opcode[1], 2’b10}</td></tr>
<tr><td>addu</td><td>10 00 01</td><td>001 0 01</td><td>addiu</td></tr>
<tr><td>sub</td><td>10 00 10</td><td rowspan="2">110</td><td></td><td></td><td></td></tr>
<tr><td>subu</td><td>10 00 11</td><td></td><td></td><td></td></tr>
<tr><td>and</td><td>10 01 00</td><td>000</td><td rowspan="4">逻辑运算：func[3:2] == 2’b01
ALUop = {func[1], 1’b0, func[0]}</td><td>001 1 00</td><td>andi</td><td rowspan="3">逻辑运算：opcode[2] == 1’b1
ALUop = {opcode[1], 1’b0, opcode[0]}</td></tr>
<tr><td>or</td><td>10 01 01</td><td>001</td><td>001 1 01</td><td>ori</td></tr>
<tr><td>xor</td><td>10 01 10</td><td>100</td><td>001 1 10</td><td>xori</td></tr>
<tr><td>nor</td><td>10 01 11</td><td>101</td><td>001 1 11</td><td>lui</td><td>非运算类指令，需单独处理</td></tr>
<tr><td>slt</td><td>10 10 10</td><td>111</td><td rowspan="2">比较运算：func[3:2] == 2’b10
ALUop = {~func[0], 2’b11}</td><td>001 0 10</td><td>slti</td><td rowspan="2">比较运算：opcode[2:1] == 2’b01
ALUop = {~opcode[0], 2’b11}</td></tr>
<tr><td>sltu</td><td>10 10 11</td><td>011</td><td>001 0 11</td><td>sltiu</td></tr>
</table>

图 23: R-Type 和 I-Type 指令 ALUop 编码

同时 I-Type 内存写指令还需要根据 ALU 计算结果和 opcode 来进行 Writestrb 译码。具体见图 17:内存写操作译码表。

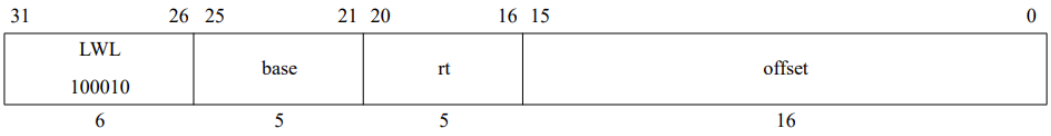
以上译码问题是本实验中个人认为最关键的部分，当我们把讲义上指令集所需要的信息进行列表并完成之后，我们再进行译码工作，当完成译码表之后，整个实现就豁然开朗。接下来我们只需要根据讲义上的电路图和我们写好的译码表来进行电路的串联即可。

2. 对于如何实现这么庞大的 CPU，我的想法是按照电路图从左到右进行实现，因此我先实现了 PC 部分的电路，而像代码中的 Jump 和 Branch 两个指令，我一开始是将其放在一起的，因为电路图中也就只有 Branch 一条指令控制着 PC 的值。但是后来发现 j 指令和 b 指令虽然都是指令跳转，但是有所区别，即其跳转地址的组成有较大区别，所以后来就将两类指令分开。

接着就是将读到的 Instruction 按照相应规则进行划分，然后再按照不同的指令所需要的译码操作进行分别译码。

然后是将不同模块根据译码得到的结果进行连接，我这里设置了 ALUop，Shiftop，Jump，Branch 和 Writestrb 这几部分，同时一个模块一个模块的实现对应的 wen 使能和输入。对应的 wen 使能和输入接口见图 21:CPU 各模块实例化接口设计。

3. 在写代码的过程中遇到的最大的问题就是 LWL，LWR，SWL，SWR 指令的实现，这几条指令在 MIPS 指令集中的意思较为抽象，不是很好理解。以 LWL 为例，该指令在 MIPS 指令集中的意思是：



Format: `LWL rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load the most-significant part of a word as a signed value from an unaligned memory address

Description: `rt ← rt MERGE memory[base+offset]`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *EffAddr*. This part of *W* is loaded into the most-significant (left) part of the word in GPR *rt*. The remaining least-significant part of the word in GPR *rt* is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word

Figure 3-2 Unaligned Word Load Using LWL and LWR

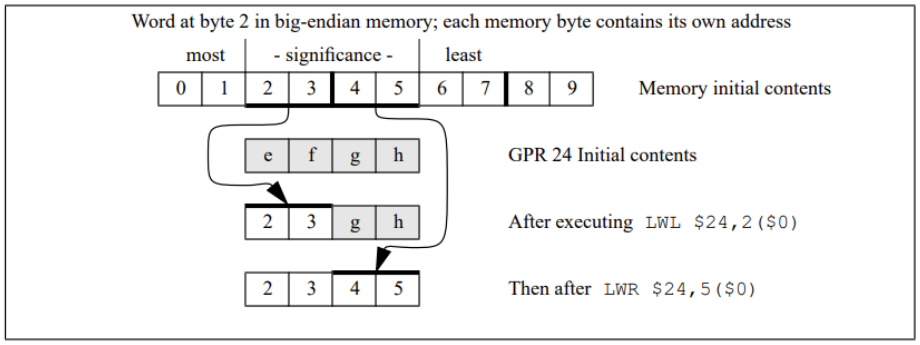


图 24: LWL 指令

从描述中我们可以知道，这个指令是将 16bit 立即数偏移量添加到通用寄存器 GPR 的 base 地址中，从而形成有效地址 EffAddr，进而从内存中读出相应的数据，将这些数据按照 most-significant 原则插入到寄存器 rt 所读出的数据 rdata2 中，而这个 EffAddr 则是根据 ALU 的计算结果得出的，因为 base 读出的数据 rdata1 和 imm 作为 ALU 的两个输入，从而计算出结果，根据这个结果我们才能知道将内存读出的数据中的几个字节插入到 rdata2 中。需要注意的是，MIPS 指令集中的例子是大尾端，而本实验要求的是小尾端。

其余三个指令与 LWL 指令类似，只不过 S 和 L 指令的读取和写入的位置相反。

另一个问题是，我设计的 Shifter 在仿真的时候存在一些问题，在 30 个测试问题中有两个问题始终没办法通过，其中有两个问题 mul-longlong 和 shift 存在错误。

```
40  ================================================
41  ERROR: at                    610ns.
42  Yours:     RF_waddr = 06, (RF_wdata & 0xffffffff) = 0x00000001
43  Reference: RF_waddr = 06, (RF_wdata & 0xffffffff) = 0xffffffff
44  ================================================
```

图 25: mul-longlong 仿真错误

```
39 ================================================
40 ERROR: at                    2430ns.
41 Yours:     RF_waddr = 04, (RF_wdata & 0xffffffff) = 0x04c3b2a1
42 Reference: RF_waddr = 04, (RF_wdata & 0xffffffff) = 0xfcc3b2a1
43 ================================================
```

图 26: shift 仿真错误

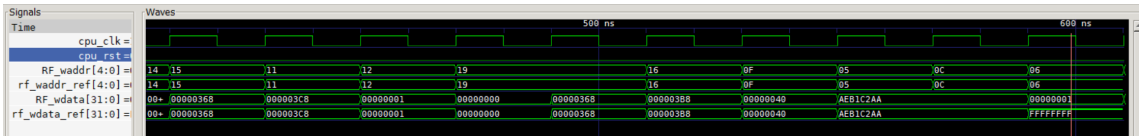


图 27: mul-longlong 错误波形

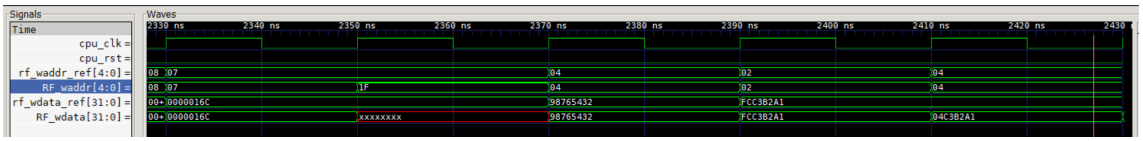


图 28: shift 错误波形

从上述两种错误仿真中我们可以看出,都是 RFwdata 存在的问题,而且都是和移位有关,因此我就想到可能是移位器存在问题。但是移位器逻辑相对简单,“这么简单会出什么问题”的思考困扰着我,最终在同学的提醒以及查阅网上的资料之后才发现了在简单的 Shifter 中存在的不起眼的问题。

因为算术移位是有符号的,而我们正常定义的 wire 类型变量是无符号的,因此我们必须将其转化成有符号的 wire 才可以直接使用算术右移操作符直接来进行算术右移操作。详见图 19:PC 时序电路部分代码。

三、 对讲义中思考题的理解和回答

- **思考题 1:ALUop 的编码有什么规律?**。

由于需要 ALU 的计算指令主要由 opcode 和 func(R-Type 类) 控制,而这些指令的 opcode 和 func 高位一般都是相同的,因此要具体译码到每条指令的话,需要根据低位的数据来判断。从图 23:R-Type 和 I-Type 指令 ALUop 编码来看,仅需要对 R-Type 的 func 低二位和 I-Type 的 opcode 低二位来进行判断即可。

- **思考题 2:表格中的 ALUop 编码是否还有优化空间?**。

我们可以观察图 23 的左右两列,发现其对应的 ALUop 的区别仅仅是 opcode[i] 还是 func[i],其他部分都相同,因此我们可以用 2 选 1 选择器来进行判断并选取相对应的指令。

四、 对于本实验的感想

本实验的目的是完成单周期 cpu,在实验中的关键地方就是解决将指令集转化成 HDL 语言的译码问题,当我们解决好这个问题之后本实验我们就已经成功了一半了,在讲义中,助教建议我们按照给出的示例自己将所有 45 条 MIPS 指令先整理在一个表格中,表格包括指令的基本信息(如指令码、指令类型等),同时还包括我们需要从 MIPS 指令集讲义中自己总结出来的一些内容(如指令的含义、指令与内存或寄存器、ALU 的接口问题等等),所以本实验近乎一半的时间我都用在了绘制这个指令集表格。在译码表中最困难的可谓是与内存读写相关的指

令,在学习了多周期 CPU 之后,我们可以看出单周期 CPU 的速度较慢的主要原因是访存和写内存指令的原因,在写单周期 CPU 的过程中这些指令也同样困扰着我最长时间。

以 LWL 等指令最为困难, 正如在第二部分中写到的 “这个指令是将 16bit 立即数偏移量添加到通用寄存器 GPR 的 base 地址中, 从而形成有效地址 EffAddr, 进而从内存中读出相应的数据, 将这些数据按照 most-significant 原则插入到寄存器 rt 所读出的数据 rdata2 中,而这个 EffAddr 则是根据 ALU 的计算结果得出的,因为 base 读出的数据 rdata1 和 imm 作为 ALU 的两个输入,从而计算出结果,根据这个结果我们才能知道将内存读出的数据中的几个字节插入到 rdata2 中。”我们需要处理四条这样类似的指令。

另外一个感想是对于门延迟的,在与助教交流的过程中,他建议我将 shifter 的组合逻辑按照 register 一样用与或非逻辑门来进行移位操作的选择,而不是用三目运算符,这是因为三目运算符的真是逻辑电路是一个选择器,其门延迟要高于普通的与或非逻辑门,因此我也按照助教的建议将 shifter 的选择部分代码改成了与或非逻辑操作来进行选择,具体见图 2:Shifter 移位器结果部分代码。

五、 实验所耗时间

在课后,你花费了大约____10____ 小时完成此次实验。