中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　专业：计算机科学与技术

实验序号：2　　实验名称：简单功能型处理器设计——基于 MIPS 32 位指令集

一、实验目的

理解 MIPS 指令格式，实现基于理想内存($Ideal\ Memory$)的简单功能型处理器数据通路和控制单元。

二、实验设计原理与思路

处理器核的组成结构包括了：1.**数据通路**（含运算器、片上互联总线等）；2.**控制器和控制部件**。

数据通路是主要的计算单元，包括算术逻辑单元(ALU)、移位器($shifter$)、寄存器堆($register\ file$)等模块，以及各个模块之间的数据传输电路，用于执行各种计算任务。

控制器和控制部件主要负责指令的解析和分发，管理数据通路以执行对应的操作，一般包含指令寄存器($Instruction\ Register, IR$)，程序计数器($Program\ Counter, PC$)，译码器($Decoder$)，控制信号发生器($Control\ Signal\ Generator$)和定时器($Timer$)。控制信号的生成可以通过多路选择器和与或操作实现。

单周期处理器($single\ cycle\ processer$)一种计算机中央处理器架构，每条指令的执行都在固定的一个时钟周期内完成。每条指令的执行可以划分为取指、译码、执行、访存、写回五个阶段，这 5 个阶段都在一个周期完成，因此CT($clock\ cycle\ time$)取决于指令集中执行时间最长的指令。

下面是按照处理器设计流程对于设计过程的记录。

第一步：分析每条指令的功能，并用RTL来表示。

这一环节比较方便，只要对照$MIPS$指令集手册将RTL代码整理到表格中就可以了。

指令类型	opcode		指令名称	RTL
R-Type	000000	计算指令	addu(func=100001)	R[rd] ← R[rs] + R[rt]
			subu(func=100011)	R[rd] ← R[rs] - R[rt]
			and(func=100100)	R[rd] ← R[rs] & R[rt]
			or(func=100101)	R[rd] ← R[rs] | R[rt]
			xor(func=100110)	R[rd] ← R[rs] ^ R[rt]
			nor(func=100111)	R[rd] ← ~(R[rs] | R[rt])
			slt(func=101010)	if R[rs]< R[rt]then R[rd]←32'b1 else R[rd] ← 32^' b0
			sltu(func=101011)	if (0 || R[rs]) < (0 || R[rt]) then R[rd] ← 32'b1 else R[rd] ← 32'b0
		移位指令	sll(Shift Word Left Logical)	R[rd] ← {R[rt] [(31-shamt) : 0], shamt'b0}
			sra(Shift Word Right Arithmetic)	R[rd] ← {(shamt){R[rt] [31]} , R[rt] [31 : shamt]}
			srl(Shift Word Right Logical)	R[rd] ← {(shamt){1'b0} , R[rt] [31 : shamt]}
			sllv(Shift Word Left Logical Variable)	s ← R[rs] [4 : 0] R[rd] ← {R[rt] [31-s : 0] , (s){1'b0}}
			srav(Shift Word Right Arithmetic Variable)	s ← R[rs] [4 : 0] R[rd] ← {(s){R[rt]31} , R[rt] [31 : s]}

			srlv(Shift Word Right Logical Variable)	s ← R[rs] [4 : 0] R[rd] ← {(s){1'b0} , R[rt] [31 : s]}
		跳转指令	jr(Jump Register)	PC ← R[rs]
			jalr(Jump and Link Register)	R[rd] ← PC + 8 PC ← R[rs]
		MOV 指令	movz(Move Conditional on Zero)	if R[rt] = 0 then R[rd] ← R[rs]
			movn(Move Conditional on Not Zero)	if R[rt] ≠ 0 then R[rd] ← R[rs]
REGIMM-Type	000001		bltz(Branch on Less Than Zero)	target_offset ← sign_extend(offset : 2'b0) If R[rs] < 0 then PC ← PC + target_offset
			bgez(Branch on Greater Than or Equal to Zero)	target_offset ← sign_extend(offset , 2'b0) if R[rs] >= 0 then PC ← PC + target_offset
J-Type	00001_		j(Jump)	PC ← {PC[32 : 28] , instr_index , 2'b0}
			jal(Jump and Link)	R[31]← PC + 8 PC ← {PC[32 : 28] , instr_index , 2'b0}
I-Type	0001__	分支指令	beq(Branch on Equal)	target_offset ← sign_extend(offset , 2'b0) if R[rs] = R[rt] then PC ← PC + target_offset
			bne(Branch on Not Equal)	target_offset ← sign_extend(offset , 2'b0) if R[rs] ≠ R[rt] then PC ← PC + target_offset
			blez(Branch on Less Than or Equal to Zero)	target_offset ← sign_extend(offset , 2'b0) if R[rs] ≤ 0 then PC ← PC + target_offset
			bgtz(Branch on Greater Than Zero)	target_offset ← sign_extend(offset , 2'b0) if R[rs] > 0 then PC ← PC + target_offset
	001001	计算指令	addiu(Add Immediate Unsigned Word)	R[rt] ← R[rs] + sign_extend(immediate)
	001111		lui(Load Upper Immediate)	R[rt] ← {immediate : 16'b0}
	001100		andi(And Immediate)	R[rt] ← R[rs] & zero_extend(immediate)
	001101		ori(Or Immediate)	R[rt] ← R[rs] | zero_extend(immediate)
	001110		xori(Exclusive OR Immediate)	R[rt] ← R[rs] ^ zero_extend(immediate)
	001010		slti(Set on Less Than Immediate)	if R[rs] < sign_extend(immediate) then R[rd] ← 32'b1 else R[rd] ← 32'b0
	001011		sltiu(Set on Less Than Immediate Unsigned)	if (0 || R[rs]) < (0 || sign_extend(immediate)) then R[rd] ← 32'b1 else R[rd] ← 32'b0
	100000	内存读指令	lb(Load Byte)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt]← sign_extend(MemRead[7+8*byte : 8*byte])
	100001		lh(Load Halfword)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt]← sign_extend(MemRead[15+8*byte : 8*byte])
	100011		lw(Load Word)	address ← sign_extend(offset) + R[rs]

				R[rt] ← MemRead
	100100		lbu(Load Byte Unsigned)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt]← zero_extend(MemRead[7+8*byte : 8*byte])
	100101		lhu(Load Halfword Unsigned)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt]← zero_extend(MemRead[15+8*byte : 8*byte])
	100010		lwl(Load Word Left)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt] ←{MenRead[7+8*byte : 0] , R[rt] [23-8*byte:0]}
	100110		lwr(Load Word Right)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] R[rt] ←{MenRead[31 : 32-8*byte] , R[rt] [31-8*byte:0]}
	101000	内存写指令	sb(Store Byte)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] WriteData ← {R[rt] [31–8*byte : 0] , (8*byte)0}
	101001		sh(Store Halfword)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] WriteData ← {R[rt] [31–8*byte : 0] , (8*byte)0}
	101011		sw(Store Word)	address ← sign_extend(offset) + R[rs] WriteData ← R[rt]
	101010		swl(Store Word Left)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] WriteData ← {(24–8*byte)0 , R[rt] [31 : 24–8*byte]}
	101110		swr(Store Word Right)	temp ← sign_extend(offset) + R[rs] address ← {temp[31 : 2] , 2'b0} byte ← temp[1 : 0] WriteData ← {R[rt] [31–8*byte : 0] , (8*byte)0}

第二步：根据指令的功能给出所需的原件，考虑时钟方案。

实验中需要完成的指令在上面的表格中列出了，共有 45 条$MIPS$指令。根据指令单格式可以分为 4 类：R_Type，$REGIMM\_Type$，J_Type，I_Type。

而根据指令的功能，有计算$(calculate)$、移位$(shift)$指令，有分支$(branch)$、跳转$(jump)$指令，有移动$(move)$指令，有访存$(load)$指令，还有存储$(store)$指令。

这些指令的译码、执行阶段显然需要寄存器堆、ALU和移位器，因此我们要先分别声明它们的输入输出端口，然后例化他们。因为本实验中取指阶段和访存阶段都在外部完成，指令内存$(Instruction\ Memory)$和数据内存$(Data\ Memory)$由外界提供，不需要在代码中实现，它们的输入输出端口则作为整个处理器模块的输入输出。

同时还要考虑PC值的变化。在单周期处理器中，一条指令执行过程中ALU模块只能有一个输入，ALU只能

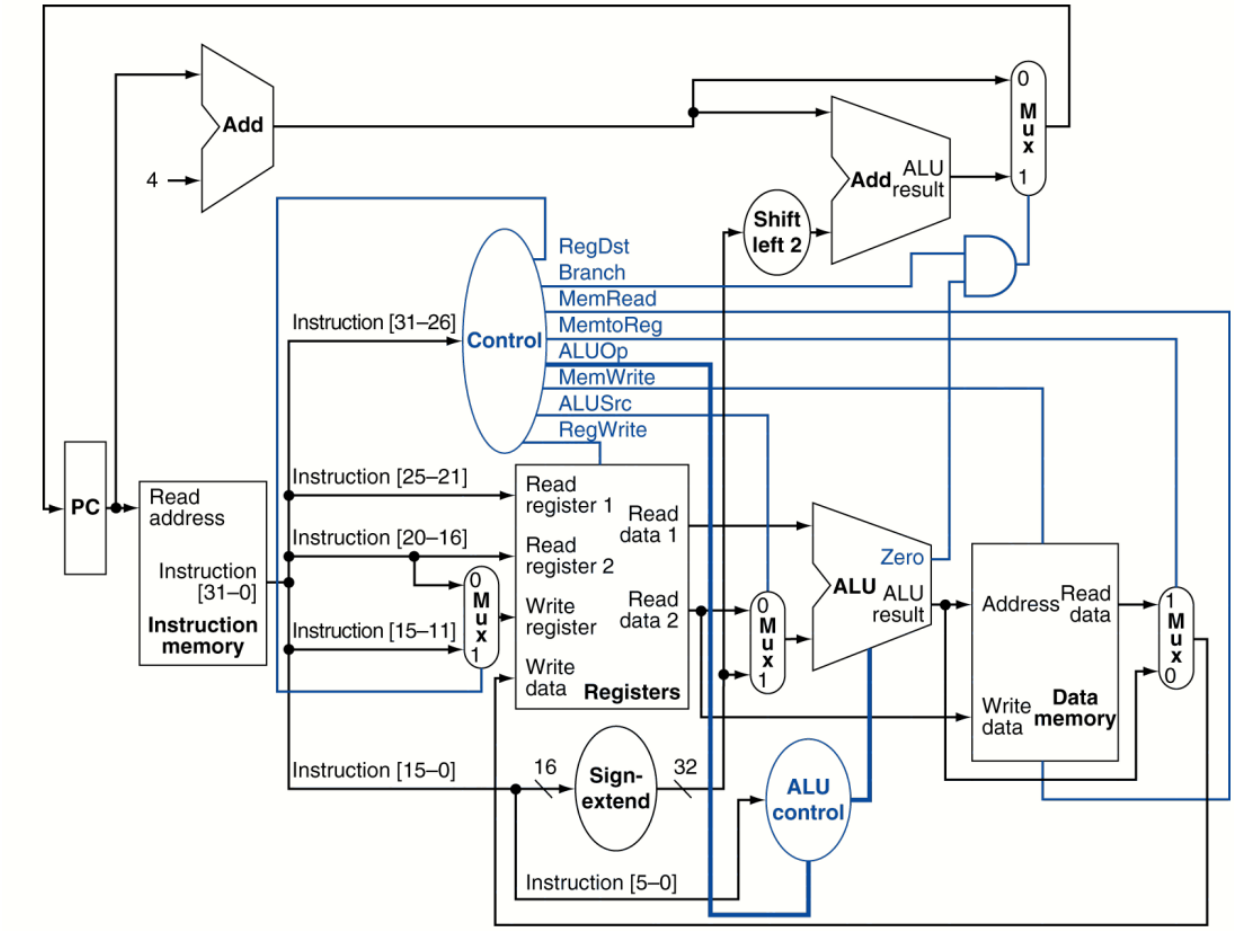
执行一次运算，因此PC值的累加和跳转还需要单独的加法器来实现。所有指令中都需要$PC+4$，而对于$branch$指令，$PC+4$后还需要加扩展后的立即数，因此硬件上还需要额外的两个加法器。

时钟控制上，只有PC和寄存器堆写是时钟同步控制的，其他操作都是异步的，完全由组合逻辑实现。PC和寄存器堆共用一个时钟信号，每个周期时钟上升沿PC的值更新，同时进行指令的写回阶段，即将数据写入寄存器堆。

第三步：将数据通路互连

下面就需要考虑各个模块输入输出端口有关的信号如何互连。理论课上，老师讲解使用的例子是支持 9 条指令的单周期处理器，感觉 9 条指令下的CPU数据通路就比较简洁清晰，译码的过程很有层次感。根据指令的$opcode$就能够得出大多数的选择控制信号，然后传到各个模块的输入端口的选择器上，能够将译码的硬件部分集中在$control$部件中,而将控制信号与数据信号区分得很清楚。同时，大多数模块的输入就只有 2 种甚至 1 种数据源，因此只要一个 2 选 1 多路选择器即可，选择时比较方便。

下面是从张老师课件中搬过来的支持 9 条指令的CPU数据通路示意图。



但是到了这次实验中，指令数量一下增加到了 45 条，而且不同指令都会有不同的操作方式。因此在考虑数据通路时我就遇到了一些困难。

首先是有些模块的输入情况有很多，控制起来很麻烦，感觉什么信号都需要，导致连线也非常复杂。

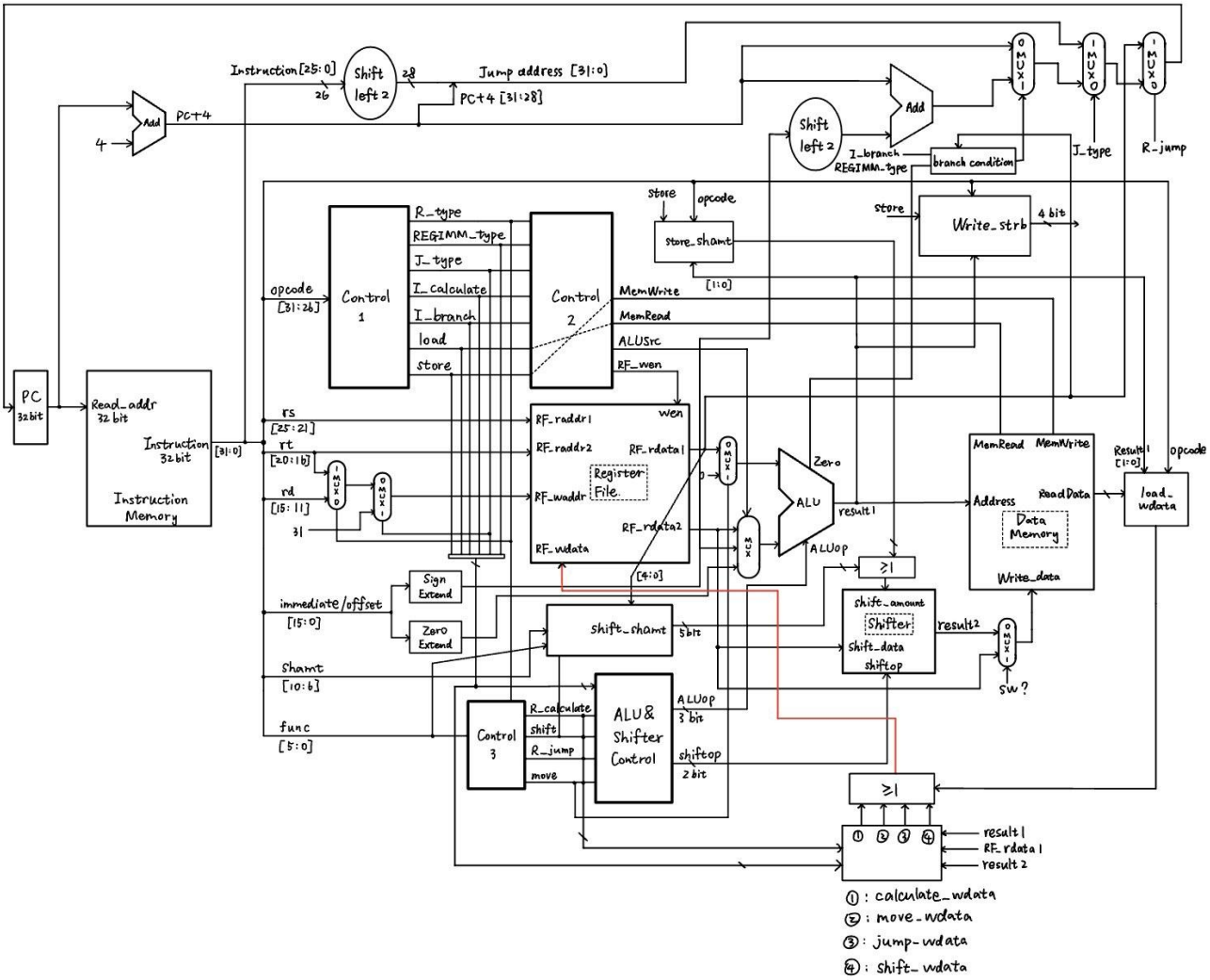
比如说我印象深刻的一个信号是寄存器堆的写入数据RF_wdata，它的数据有时候从立即数过来，有时候从寄存器堆过来，有时候从PC过来，有时候从ALU过来，有时候从移位器过来，还会从内存中过来，就几乎是所有地方都可能会传数据到寄存器中。特别是从内存中过来的数据，每一条指令的操作方式都有区别。要将这些所有情况与对应的指令匹配起来，并实现选择电路的选择需涉及到很多信号（不仅仅是$opcode$），数据通路会比较复杂。

然后就是我没有把译码部分单独地作为一块来处理。在上图中译码部分体现在椭圆形的$control$，它输入指令的$opcode$，输出各选择器的选择信号。但是到了我们这里，我就有些不清楚$control$部分要产生什么。

这可能是因为我在写代码前并没有先把数据通路先画出来，这张图是我写完之后才画的。我在写代码时很多时候没有把用来选择数据的使能信号命名为一个$wire$型的信号，而是直接将它们用在了数据选择的过程中。

所以我在画图时也将使能信号的产生和使用放在了一起，这样就感觉把译码的部分分散到了各个输入端口，不是很清楚。

不过最终经过不懈的努力，我还是画出了我的数据通路示意图。它看起来有点乱，而且*control*部分只是将指令分了一个类，并没有产生所有的控制信号和使能信号。不过我至少吸取了教训，下次应该先画出数据通路，然后给控制信号命名，这样才能做到和张老师的图中一样思路清晰。



第四步：确定每个原件所需控制信号的取值，生成一张反映指令与控制信号之间关系的表。

这一环节的过程还是比较麻烦的，需要每一条指令对照上面指令的*RTL*代码表格，考虑它要用到的模块的输入信号。最终做了一张非常宽的表格，但在这里放不下，所以我把指令分类以后拆分成几张表格。

表 1 *R_Type*计算指令

指令类型	opcode			寄存器堆读		ALU			寄存器堆写		
				raddr1	raddr2	A	B	ALUop	wen	waddr	wdata
R-Type	000000	计算指令	addu(func=100001)	rs	rt	rdata1	rdata2	{func[1],2'b0}	1	rd	result1
			subu(func=100011)								
			and(func=100100)					{func[1],1'b0,func[0]}			
			or(func=100101)								
			xor(func=100110)								
			nor(func=100111)								
			slt(func=101010)					{~func[0],2'b11}			
			sltu(func=101011)								

表 2 *R_Type*移位指令

指令类型	opcode			寄存器堆读		寄存器堆写			移位器		
				raddr1	raddr2	wen	waddr	wdata	A	B	Shiftop
R-Type	000000	移位指令	sll	rs	rt	1	rd	result2	rdata2	sa	func[1:0]
			sra								
			srl								
			sllv							rdata1[4:0]	
			srav								
			srlv								

表 3 *R_Type*跳转和*MOV*指令

指令类型	opcode			寄存器堆读		ALU			寄存器堆写		
				raddr1	raddr2	A	B	ALUop	wen	waddr	wdata
R-Type	000000	跳转指令	jr	rs					0		
			jalr	rs					1	rd	PC+8
		MOV指令	movz	rs	rt	32'b0	rdata2	010	zero	rd	rdata1
			movn	rs	rt				!zero	rd	rdata1

表 4 *REGIMM_Type*指令

指令类型	opcode			寄存器堆读		跳转		备注
				raddr1	raddr2	跳转地址	地址更新条件	
REGIMM-Type	000001	分支指令	bltz	rs		PC+4+target_offset	rdata1[31]==1	(target_offset ← sign_extend(offset || 00)
			bgez	rs		PC+4+target_offset	rdata1[31]==0	

表 5 *J_Type*指令

指令类型	opcode			跳转		寄存器堆写		
				跳转地址	地址更新条件	wen	waddr	wdata
J-Type	00001_	跳转指令	j	PC_GPRLEN..28 || instr_index || 00		0		
			jal	PC_GPRLEN..28 || instr_index || 00		1	5'b11111	PC+8

表 6 *I_Type*分支指令

指令类型	opcode			寄存器堆读		ALU			跳转	
				raddr1	raddr2	A	B	ALUop	跳转地址	地址更新条件
I-Type	0001__	分支指令	beq	rs	rt	rdata1	rdata2	110	PC+4+target_offset	zero==1
			bne	rs	rt	rdata1	rdata2	110	PC+4+target_offset	zero==0
			blez	rs	rt	rdata1	rdata2	010	PC+4+target_offset	rdata1[31]||zero
			bgtz	rs	rt	rdata1	rdata2	010	PC+4+target_offset	!(rdata1[31]||zero)

表 7 *I_Type*计算指令

指令类型	opcode			寄存器堆读		ALU			寄存器堆写		
				raddr1	raddr2	A	B	ALUop	wen	waddr	wdata
I-Type	001001	计算指令	addiu	rs		rdata1	sign_extend(immediate)	010	1	rt	result1
	001111		lui						1	rt	offset||0^16
	001100		andi	rs		rdata1	zero_extend(immediate)	000	1	rt	result1
	001101		ori	rs		rdata1	zero_extend(immediate)	001	1	rt	result1

	001110		xori	rs		rdata1	zero_extend(immediate)	100	1	rt	result1
	001010		slti	rs		rdata1	sign_extend(immediate)	111	1	rt	result1
	001011		sltiu	rs		rdata1	sign_extend(immediate)	011	1	rt	result1

表 8 *I_Type load*指令

指令类型	opcode			寄存器堆读	ALU			内存访问			寄存器堆写		
				raddr1	A	B	ALUop	Address	Mem Read	Mem Write	wen	waddr	wdata
I-Type	100000	内存读指令	lb	rs	rdata1	Signextended (offset)	010	target_addr	1	0	1	rt	sign_extended (target_byte)
	100001		lh	rs	rdata1		010		1	0	1	rt	sign_extended (target_halfword)
	100011		lw	rs	rdata1		010		1	0	1	rt	Read_data
	100100		lbu	rs	rdata1		010		1	0	1	rt	zero_extended (target_byte)
	100101		lhu	rs	rdata1		010		1	0	1	rt	zero_extended (target_halfword)
	100010		lwl	rs	rdata1		010		1	0	1	rt	temp
	100110		lwr	rs	rdata1		010		1	0	1	rt	temp

表 9 *I_Type store*指令

指令类型	opcode			寄存器堆读		ALU			内存访问					移位器		
I-Type	101000	内存写指令	sb	rs	rt	rdata1	Signextended (offset)	010	target_addr	0	1	result2	result1[1]?(result1[0]?4'b1000:4'b0100): (result1[0]?4'b0010:4'b0001)	rdata2	result1[1:0]||000	00
	101001		sh	rs	rt	rdata1		010		0	1	result2	result1[1]?(4'b1100):(4'b0011)	rdata2	result1[1:0]||000	00
	101011		sw	rs	rt	rdata1		010		0	1	rdata2	4'b1111			
	101010		swl	rs	rt	rdata1		010		0	1	result2	result1[1]?(result1[0]?4'b1111:4'b0111): (result1t[0]?4'b0011:4'b0001)	rdata2	(~result1[1:0])||000	10
	101110		swr	rs	rt	rdata1		010		0	1	result2	result1[1]?(result1[0]?4'b1000:4'b1100): (result1[0]?4'b1110:4'b1111)	rdata2	result1[1:0]||000	00

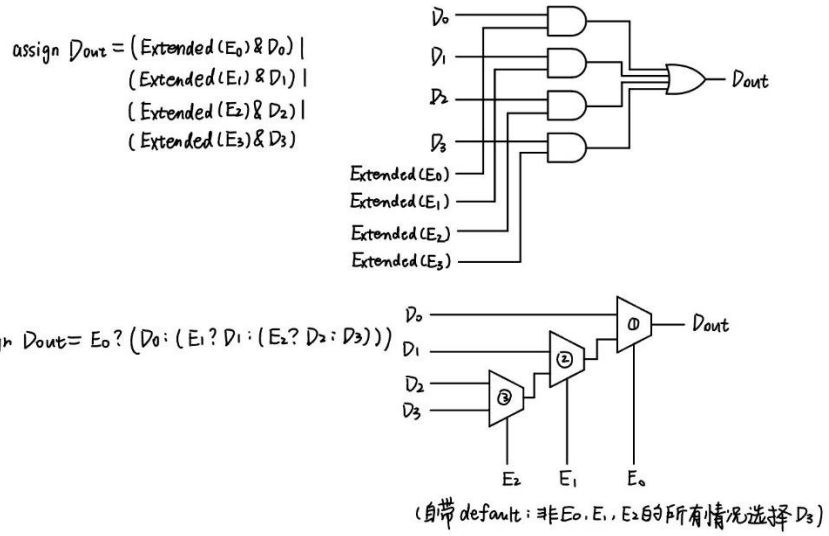
第五步：根据表得到每个控制信号的逻辑表达式，涉及控制电路。

这一环节其实就是把上面表格中的每一列进行逻辑表达式的整理和化简，将相同输入的指令归为一种情况，根据这些指令共有的特征写出这种输入的使能信号，在进行输入数据的选择。

每个模块的输入端口都有很多种信号源，涉及到多选一，因此需要使用n选 1 多路选择器(MUX)。n选 1 多路选择器在$verilog$代码中的实现有 2 种方式。

第一种是通过将使能信号与数据信号的每一位按位与，在将所以信号源的数据信号按位或，这其实类似于n选 1MUX内部在硬件上的实现方式。第二种是进行多次二选一，每次分出一种情况。由于 2 选 1MUX在$verilog$代码中可以通过三目运算符直观地表现，因此这里可以套用多层三目运算符来实现，而在硬件上就是多个 2 选 1 选择器一层层叠加起来。

下面是 4 选 1MUX用两种方法实现的例子。

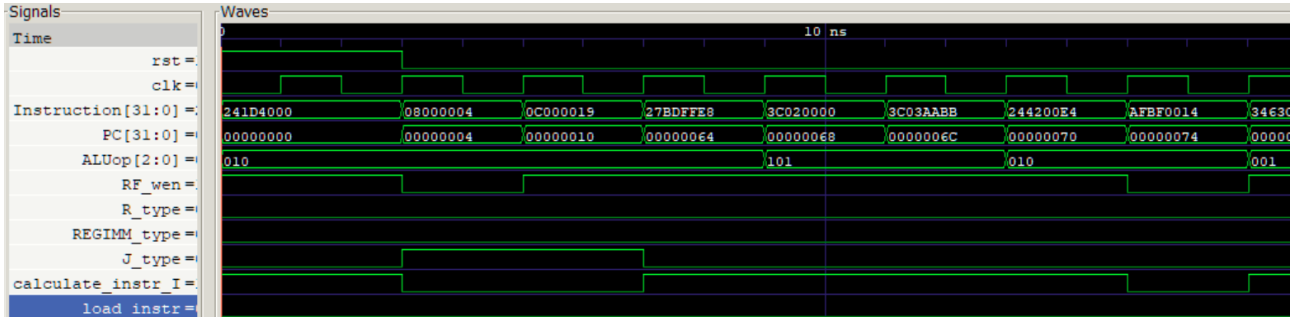


在功能上n个 2 选 1MUX与 1 个$n+1$选 1 的MUX是一样的。而这两种方法也各有利弊。第一种方法的优势在于体现在代码中会显得非常清晰，每一种情况的使能信号与数据信号相对应起来，可以很容易读懂代码；同时因为它是并行的选择，它的延迟更低。第二种方法的好处在于它自带了$defult$情况，只要将特殊的几种情况先分出来，剩下的所有情况就不需要使能信号来归类。但是它的缺陷也很明显，首先如果一行代码中出现了好几层三目运算符会显得很乱，也有可能会出错；同时它是串行的选择，下一个MUX的选择信号需要等待上一个MUX的输出才能选择，所以它的延迟会更长。

三、信号波形仿真

这次实验的$verilog$代码比较长，不方便在文档中。而且这次实验仿真测试有 30 项，我前后调试改错了二十几次才通过了仿真测试，下载了很多次波形，对照着波形一次次地检查自己可能出错的地方，确实是一次非常难忘的改错体验。

但是我很难用几张波形图片就展现出处理器的功能，因为涉及的信号非常多，我很难全放上去，那我就“装模作样”地放一张之前改错时下载的波形吧。



四、实验中遇到的问题

这次实验我调试修改代码过程中遇到的问题真的有一箩筐，一会儿这里错，一会儿那里错。错误原因的话，要么是写的时候手残打错了信号名称，要么就是对指令的操作理解的不对。我总结一下几个印象深刻的错误吧。

首先是ALU里的一个错误。因为ALU加了$SLTU$，它也需要使用用减法，需要对操作数B去补码。但是我原来取补码的译码操作对这条指令是不适用的，所以导致了本来做减法的时候做了加法。这个错误藏得很深，我找了好久才想到。

还有一个是$write_strb$输出的错误。这个信号应该几乎是情况最麻烦的一个了，所以我就把它的 4 位拆分成 4 个信号，每个信号单独译码。而且我没有采用之前的两种选择器实现方式，而是直接先写成最小项形式，然后化简为最简表达式，这样虽然长度上变短了，但是我检查错误的时候也巨麻烦。最后我真的是在化简的时候出了一个错，改了好久才改好。

五、实验心得

这次实验首先让我对$MIPS$指令集更熟悉了，几种类型的指令以及它们的格式、功能，然后我也更加了解了

处理器的工作方式，它如何去实现将各条指令要做到事情转化成控制信号，传输给各个功能模块去完成，而且还要尽量追求硬件上的精简，不能说逐条执行去译码。

当然，我做完之后发现其实自己不应该着急去编写代码。虽然我做了指令单的译码表，但我没有先画出数据通路的结构图，导致我对各个模块需要的使能信号没有一个归类和总结，这样也增加了我$verilog$代码的调试改错时间。