p3_log

[toc]

整体结构

- 两个mealy机 -> 单周期CPU
- 控制器生成真值表
- 抽象和模块化
- 控制器打表
- MIPS测试，设计数据
- RAM与ROM的区别

设计过程

- CPU分为两个部分，数据通路(Datapath)+控制器(control)

数据通路

- 一个MIPS数据通路又可以抽象为五步：取指令(IF)，译码(ID)，执行(EX)，访存(MEM)，回写(WB)
- 相应的应该有PC、NPC、IM、RF、ALU、DM、Controller这几个大的模块，以及在实现的过程中会存在的MUX、EXT、Splitter这些小的模块（系统与子系统的模块）
- 数据通路是连接方式的组合，介于我们需要实现指令条数，我们通过这些指令的RTL描述，倒推出这条指令所需要的功能部件，从而建立起部件之间的连接关系，以及控制信号的取值。然后把这些连接关系填入数据通路的表中，填入每一个端口所需要的输入。
- 把所有的指令分析完后，归并每个端口的输入信号，信号来源唯一的端口可以直接连接，但多输入的情况我们需要MUX+Controller的实现。
- 得到最后的数据通路图后，我们很容易能构造出数据通路。

控制器设计

- 控制器的设计过程，我们理解为通过opcode和function识别指令+输出相应的控制信号，这两步可以进一步抽象为和逻辑和或逻辑。
- 我们把表给列出来（其实这也是一种真值表）。
- 创建好以后，连线就行

具体实现

- 确定所需实现的指令：addu，subu，ori，lw，sw，beq，jal，jr（MIPS-C0），add，sub，xor，jalr，nop，sll/sllv，lui，slt。

- 尽量每一种指令都来一下：

    - R型：
        - 计算指令：add,sub,and, or
        - 移位指令：sll,sra,sllv
        - 置位指令：slt,sltu
        - 跳转指令：jr,jalr

        - 系统调用：syscall
    - I型：
        - 计算指令：addi,ori,lui
        - 置位指令：slti,sltiu
        - 分支指令：beq,bne,bgtz
        - 访存指令：lw,sw,lh,sh,lhu,lb,sb,lbu
    - J型：
        - 跳转指令：j,jal

- 把每个模块组装好，设计好端口
- 打表连接
- 在端口的命名中我们采取的一些命名缩写，它的意义是

命名	意义	备注
WD	write data	
RD	read data	
A	address	
WR	write reg	
WE	write enable	

PC/NPC实现

- PC用寄存器实现
- NPC先实现一个简单的+4功能
- 后续跳转指令加入时再看
- beq：在ALU的地方返回一个rs=rt的结果到NPC，把imm16传到NPC，再传个控制器信号，是beq就跳
- jal：在这里我们的控制信号就要多增加，而且不仅要计算PC+4+imm26的值，我们还需要计算PC+4的值，写入31号$sp寄存器
- j：与jal的区别仅在于，不用回写
- jr：PC <- GPR[rs],需要读入rs寄存器的值

信号名	方向	描述	备注
PC[31:0]	I	当前位置	
imm[26]	I	26位立即数	beq是imm16,jal是add26,可以覆盖
ra[31:0]	I	jr需要的32位返回地址	
BeqYes?	I	rs,rt相等的标志	NPCOp=001起作用

信号名	方向	描述	备注
NPCOp[2:0]	I	NPC功能选择： 000: 顺序地址 001: beq 010: j 011: jal 100: jr	
NPC[31:0]	O	输出接下来的PC	
PC4[31:0]	O	PC+4	

IM实现

- IM没有写入的要求，用ROM即可，指令为32位，地址位宽可以选取8（即256条指令）

InstructionSplitter

- 在32位的机器码中根据MIPS指令的类型我们需要得到opcode、rs、rt、rd、shamt、funt、imm16、address，在RIJ型的指令中分别会用到，但是我们并不需要判断指令类型给出相应的输出。我们把这些都给输出，需要什么是后续元件与Controller的事。

RF实现

- 这里在p0已经写过了，就直接CV了，但是注意端口的命名可以改一下。

信号名	方向	描述	备注
clk	I	时钟信号	
clr	I	异步复位	
WE	I	写使能，为1时可以写入	
A1[4:0]	I	指定32各寄存器中的一个，将值读入RD1	
A2[4:0]	I	指定32各寄存器中的一个，将值读入RD2	
A3[4:0]	I	指定32各寄存器中的一个，作为WD值的写入对象	
WD[31:0]	I	在WE为1，将内容写入A3指定的寄存器	
RD1[31:0]	O	输出A1指定的寄存器的值	
RD2[31:0]	O	输出A2指定的寄存器的值	

ALU（重点！）

- 在最后加一个MUX输出
- addu：直接用加法器
- subu：直接用减法器

- slt：比较器
- ori：有i的都可以按照这个思路来，在RD2与imm16（Zero_ext32）之间加一个MUX，剩下交给控制器
- lw：这里我们需要EXT来实现立即数的符号位拓展，而在ori指令中有非符号位拓展，把他们集成起来做一个EXT模块，剩下交给控制器去决定，R[rt] <- MEM[R[rs]+sign_ext(imm16)]，所以rt与rd到A3加一个MUX
- sw：MEM[R[rs]+sign_ext(imm16)] <- R[rt]，RD2接到DMWD
- 考虑到跳转指令需要对NPC模块进行改装，beq、j、jal、jr指令的实现就放到PC/NPC实现里了
- 下面是不同的ALUOp代表的功能

ALUOp	功能	备注
0000	addu	
0001	subu	
0010	ori	
0011	slt	
0100	lw	
0101	sw	

EXT

信号名	方向	描述	备注
ExtOp	I	选择拓展方式： 0: 0拓展 1: 符号位拓展	
imm16	I	输入的16位立即数	
imm32	O	输出的32位立即数	

DM的实现

- 用一个可写入的RAM来实现
- 用位拓展器把传入的32位值变为8位

Controller

- 可以分为两个部分，一个是识别器，一个是信号器
- 根据后面的信号表与和或逻辑来搭建
- 注意合理CV，不然会很难受
- 注意测试
- 问题：nop不需要控制为啥，因为nop的时候NPCOp=000，几个WE都是0，写不进东西，所以不需要控制

> 自此，框架已经搭好，现在开始做打表连接

数据通路表及控制信号表

指令	RTL描述
addu	R[rd] <- R[rs]+R[rt]
subu	R[rd] <- R[rs]-R[rt]
ori	R[rt] <- R[rs] | 0ext32(imm16)
slt	R[rd] <- 0ext32(R[rs] < R[rt])
lw	R[rt] <- M[R[rs]+imm32]
sw	M[R[rs]+imm32] <- GPR[rt]
beq	if (R[rs] == R[rt])
j	PC <- add26
jal	PC <- add26; R[31] <- PC + 4;
jr	PC <- R[rs];

- 下面根据RTL描述写出数据通路表
- 每一个指令需要将什么端口连接，指令级数据通路与控制器建模，先忽略控制信号的端口

部件	NPC				IM	RF				ALU		DM	
指令 / 输入信号	PC32	imm26	ra32	BeqYes?	A	A1	A2	A3	WD	A	B	A	WD
addu	PC				PC	rs	rt	rd	ALUout	RD1	RD2		
subu	PC				PC	rs	rt	rd	ALUout	RD1	RD2		
ori	PC				PC	rs		rt	ALUout	RD1	0imm16		
slt	PC				PC	rs	rt	rd	ALUout	RD1	RD2		
lw	PC				PC	rs		rt	DMout	RD1	Simm16	ALUout	
sw	PC				PC	rs	rt			RD1	Simm16	ALUout	RD2
beq	PC	imm16		Zero	PC	rs	rt			RD1	RD2		
j	PC	add26			PC								
jal	PC	add26			PC			0x1f	PC4				
jr	PC		RD1		PC	rs							
总合计	PC	imm16 add26	RD1	Zero	PC	rs	rt	rd rt 0x1f	ALUout DMout PC4	RD1	RD2 0imm16 Simm16	ALUout	RD2

- 相应的在分析每条指令的通路的时候可以把控制信号的取值记下来，暂时不包含MUX的控制

指令	opcode	funct	NPCOp	immEXTOp	RFWE	ALUOp	DMWE
addu	000000	100001	000		1	0000	0
subu	000000	100011	000		1	0001	0
ori	001101		000	0	1	0010	0
slt	000000	101010	000		1	0011	0
lw	100011		000	1	1	0100	0
sw	101011		000	1	0	0101	1
beq	000100		001		0		0
j	000010		010		0		0
jal	000011		011		1		0
jr	000000	001000	100		0		0

- 所以数据通路连接的总合计为

部件	NPC				IM	RF				ALU		DM	
指令 / 输入信号	PC32	imm26	ra32	BeqYes?	A	A1	A2	A3	WD	A	B	A	WD
总合计	PC	imm16 add26	RD1	Zero	PC	rs	rt	rd rt 0x1f	ALUout DMout PC4	RD1	RD2 0imm16 Simm16	ALUout	RD2

- 在根据数据通路表中的多输入端口，把MUX的Sel信号写入控制信号表

指令	WRsel	RFWDSel	BSel
addu	01	00	0
subu	01	00	0
ori	00	00	1
slt	01	00	0
lw	00	01	1
sw			1
beq			0
j			
jal	10	10	
jr			

debug

- ROM的寻址方式
- 加入指令lui
  - 在数据通路连接表与信号控制表里加入它，然后把该连的线连起来
  - 按照控制表打控制器
- 发现一个ori在mars里调用了ori、lui、or
- 但是or除了ALUOp不同以外其余都相同
- add,sub
- DM写入地址输出
- imm32,address移位但ra不移，所以对应到p3的处理办法，ra应该移位
- beq的imm16要符号位拓展(p4的时候才发现这个bug，可见测试数据还是很弱的，需要自己造数据)

测试

- 每条指令功能都测试一下
- 每条指令考虑正负等等
- 用disassembler的工具进行机械码转汇编语言
  - 转换注意：b -> j
  - neg -> sub
- CPU的每一步对应着MARS调试的每一步
- jr指令可以跳到所有寄存器的储存值，注意！

后续增量开发

- 思考是RIJ哪种指令，思考需不需要跳转
  - R型I型对应不同的WRSel
  - 跳转要改变NPCOp和NPC接口
  - 写对应的控制信号
- 下面由于特殊性，加上lh,lb,sh,sb, sll指令
- 对于lh,sh,lb,sb,lbu,lhu指令,下面以lb,sb,lbu为例
  - 改版DM，增加DMOp2:0
  - store操作：通过out与WD的7:0拼起来读入，根据A决定
  - load操作：只输出out的8位（根据A来决定），注意拓展
  - 其余模块都与sw，lw一致
- sll指令比较的特殊，需要加上ASel（为1时选择shamt）
- Controller指令的顺序改进

思考题

- 上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。
  - 状态存储：IM，RF，DM
  - 状态转移：NPC，ALU
- 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。
  - 正确的，合理的，无可挑剔的
- 在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。
  - 我把Controller分为了两个部分，一个是识别器，一个是信号发送器，主要考虑是可以在main电路中调试的时候看到当前执行的指令。
- 事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？
  - 因为nop的时候NPCOp=000，没有跳转，而且几个WE使能是0，写不进东西，所以不需要控制
  - 而且我写了sll，所以nop就对应着移0位的sll，等同于没有修改。
- 上文提到，MARS 不能导出 PC 与 DM 起始地址均为 0 的机器码。实际上，可以避免手工修改的麻烦。请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。
  - 可以把其中一个起始地址先设为0，然后另一个可以通过写程序实现判断与修改。
- 阅读 Pre 的 “MIPS 指令集及汇编语言” 一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。
  - 有些寄存器没有用到，有些负数的特殊情况没有涉及（比如beq的负数情况），跳转的指令没有设计