P3课下-Logisim搭建单周期CPU

处理器为 32 位单周期处理器，支持指令集：`add, sub, ori, lw, sw, beq, lui, nop`。其中 `nop` 为空指令，机器码 `0x00000000`； `add, sub` 按无符号处理，不考虑溢出；顶层有效驱动信号包括且仅包括 **异步复位信号 reset** 。

IFU（取指令单元）

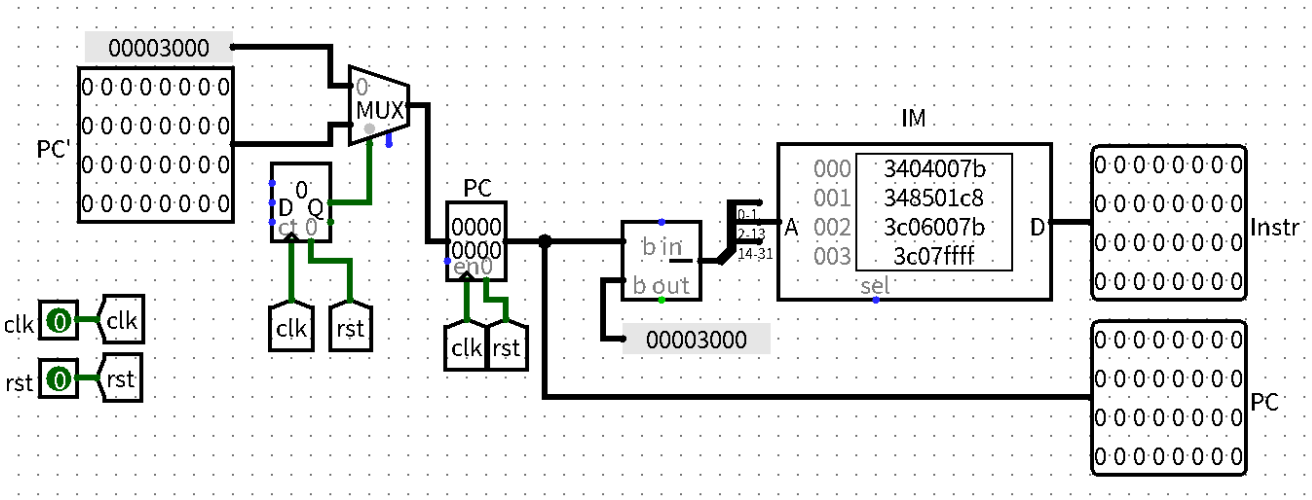
PC（程序计数器）

- 用寄存器实现，输入信号为指令地址；
- 提供**异步复位**，复位为**起始地址 0x00003000**，地址范围 0x00003000 ~ 0x00006FFF；
- **注意**：在 32 位架构中，PC 从一条指令指向下一条指令时，递增的量为 4 （1字节），而 ROM 按字计算地址，PC 指令传给 ROM 时需要除以 4 ，在 Logisim 中，使用 Splitter 舍弃后两位；
- PC 的首地址 0x00003000 与 ROM 的首地址 0 对应，且复位为 0x00003000，考虑对输入端口 PC' 减去 0x00003000，这么处理使得寄存器 PC 的值符合要求，且直接可以实现异步复位。

IM（指令存储器）

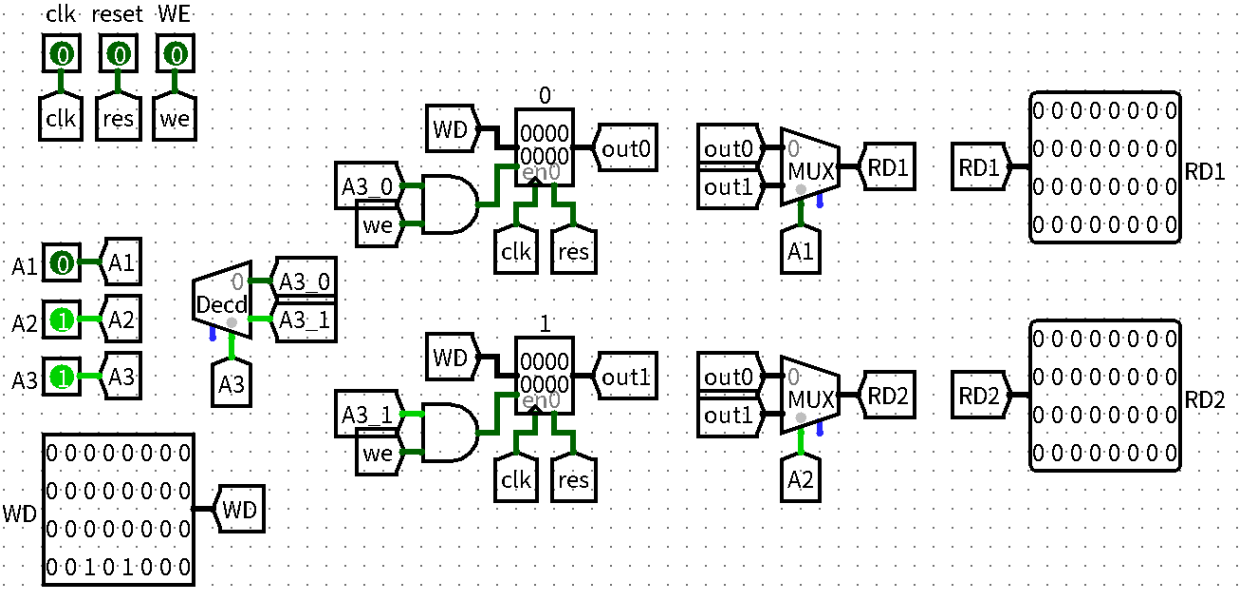
- 用 ROM 实现，容量 4096 × 32bit，起始地址 0；
- 根据 PC 地址范围推算，ROM 实际地址宽度 12 位，与 PC 地址的 [13: 2] 位相连。

信号名	方向	描述
clk	I	时钟信号
rst	I	异步复位信号，将 PC 寄存器复位
PC'	I	32 位程序计数器信号
Instr	O	32 位指令信号

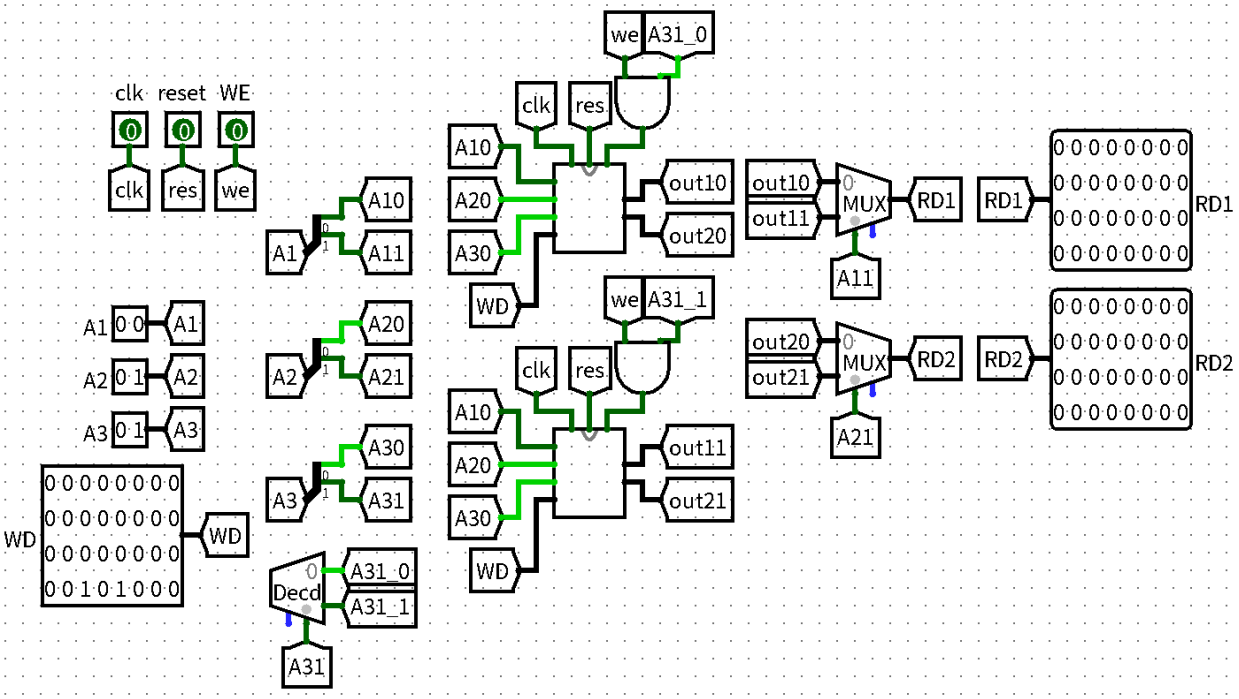


GRF（通用寄存器组）

- 用具有写使能的寄存器实现，寄存器总数 32 个，提供**异步复位**；
- 0 号寄存器的值始终为 0，其他寄存器初始值为 0 ，在 P0 课下已经实现，可以直接套用；
- 实现思路简述：先搭建包含 2 个寄存器的子电路，再进行地址扩展，扩展为 4reg、8reg、16reg、32reg。

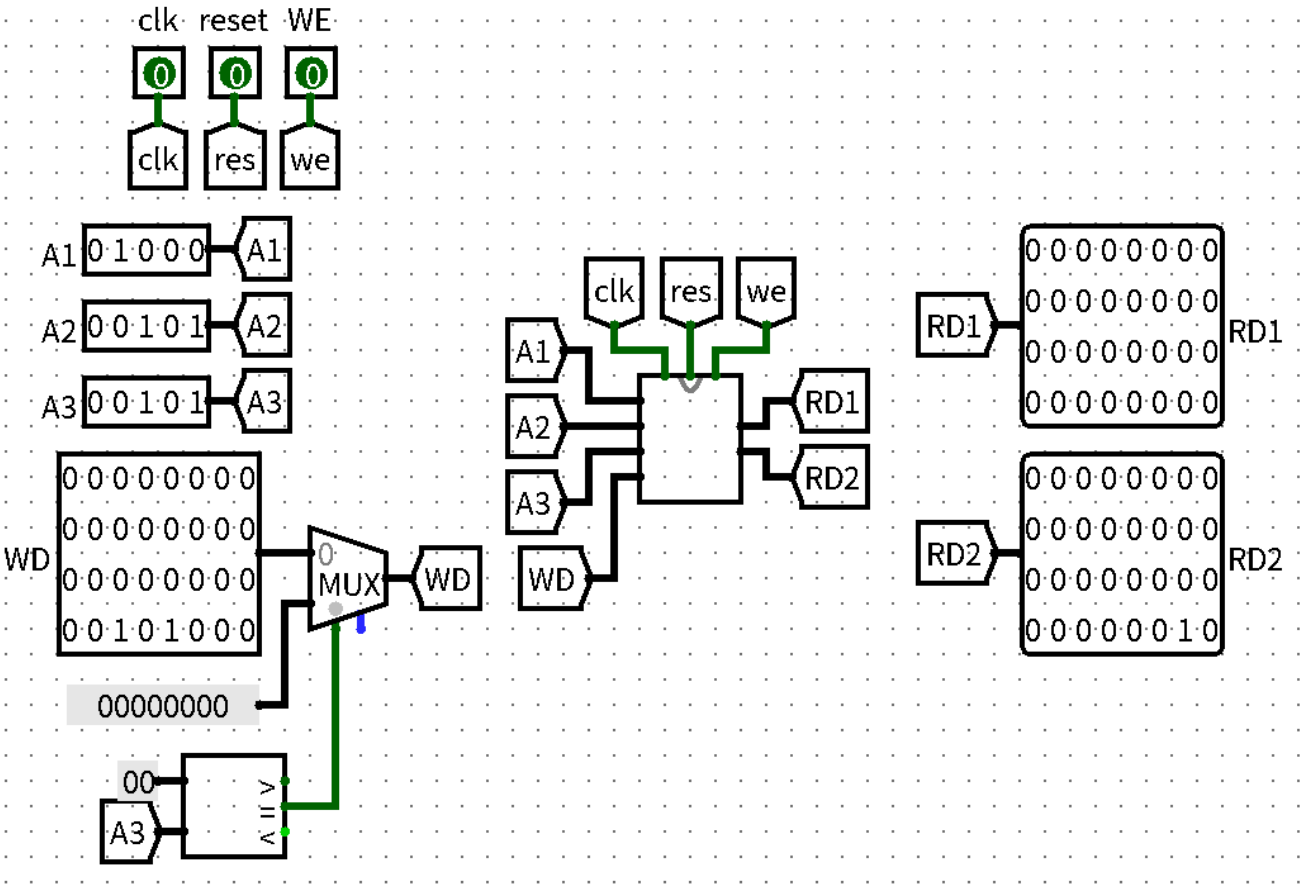


这是最底层的情况，仅有两个寄存器，地址 1 位。



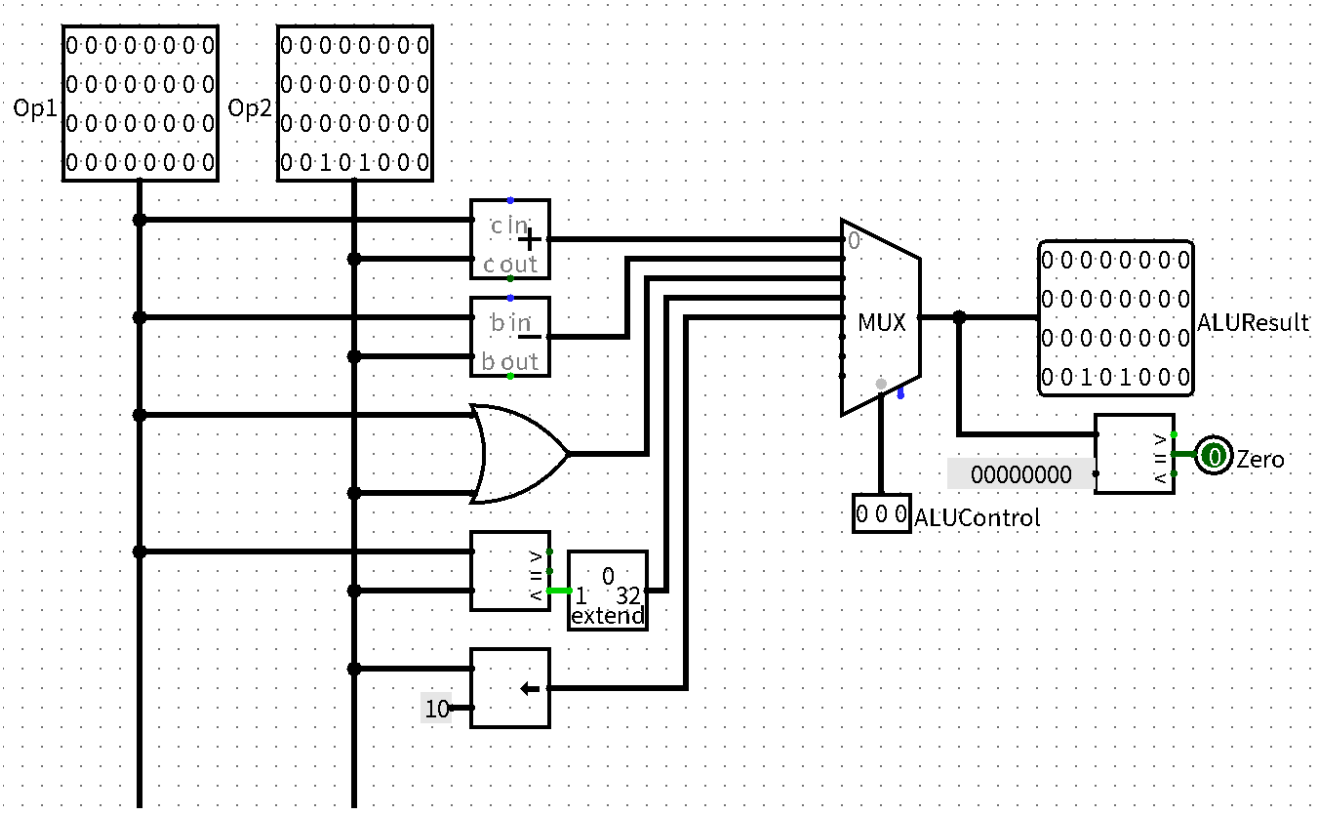
这是第二层的情况，实际有 4 个寄存器，地址 2 位。其余高层类推，不再赘述。

信号名	方向	描述
clk	I	时钟信号
reset	I	复位信号，高电平有效，将 32 个寄存器的值全部清零
WE	I	写使能信号，高电平时允许向 GRF 写入数据
A1	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中数据输出到 RD1
A2	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中数据输出到 RD2
A3	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其作为写入的目标寄存器
WD	I	32 位数据输入信号
RD1	O	输出 A1 指定的寄存器中的 32 位数据
RD2	O	输出 A2 指定的寄存器中的 32 位数据



ALU（算术逻辑单元）

信号名	方向	描述
ALUControl	I	3 位逻辑运算符选择信号 000：加法 001：减法 010：或 011：小于置1 100：加载至立即数最高位
Op1	I	32 位操作数 1
Op2	I	32 位操作数 2
Zero	O	运算结果判 0 信号，高电平表示为 0
ALUResult	O	32 位算术逻辑运算结果输出信号

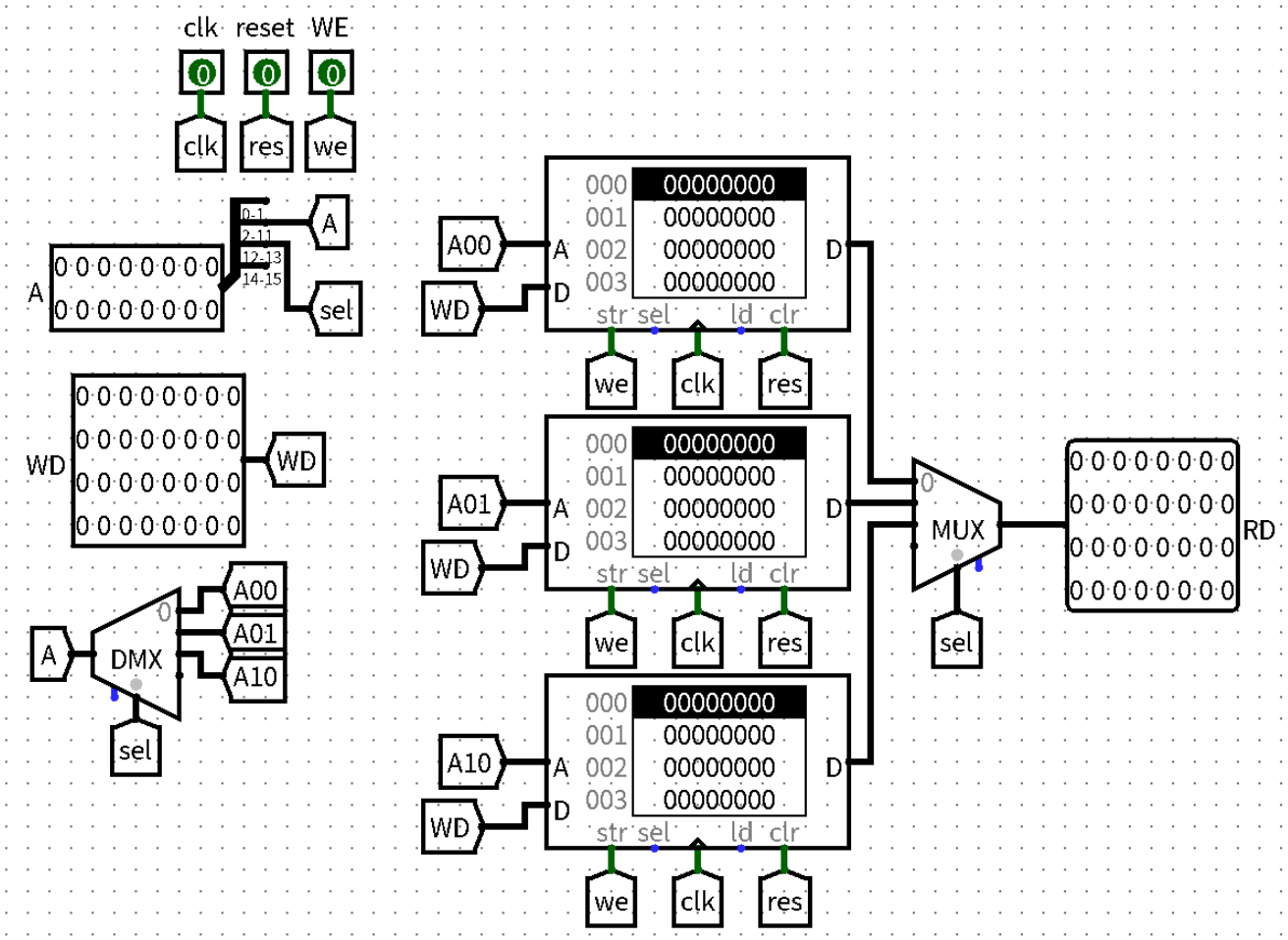


这里有个 011 判断小于的指令，在课下实际用不到（照博客弄的，后来发现没有用）但我相信以后会有用的，先保留。（）

DM（数据存储器）

- 用 RAM 实现，容量 3072 × 32bit，提供**异步复位**，**起始地址 0x00000000**，地址范围 0x00000000 ~ 0x00002FFF；
- RAM 使用双端口模式，**Seperate load and store ports**；
- 地址线 $3 \times 2^{10} = 3072$，机器码 16 位输入，使用 3 个 1K × 32 RAM 进行字扩展，由于输入一定是 4 的倍数，舍弃最低两位，接着 10 位寻址 RAM ，最高两位用于片选。

信号名	方向	描述
clk	I	时钟信号
rst	I	异步复位信号，将 RAM 复位
WE	I	RAM 写入使能信号
A	I	16 位机器码地址
WD	I	32 位数据输入信号
RD	O	32 位数据输出信号



很规范很完美的片选和位扩展！有不少同学图省事直接弄了个大 RAM 。

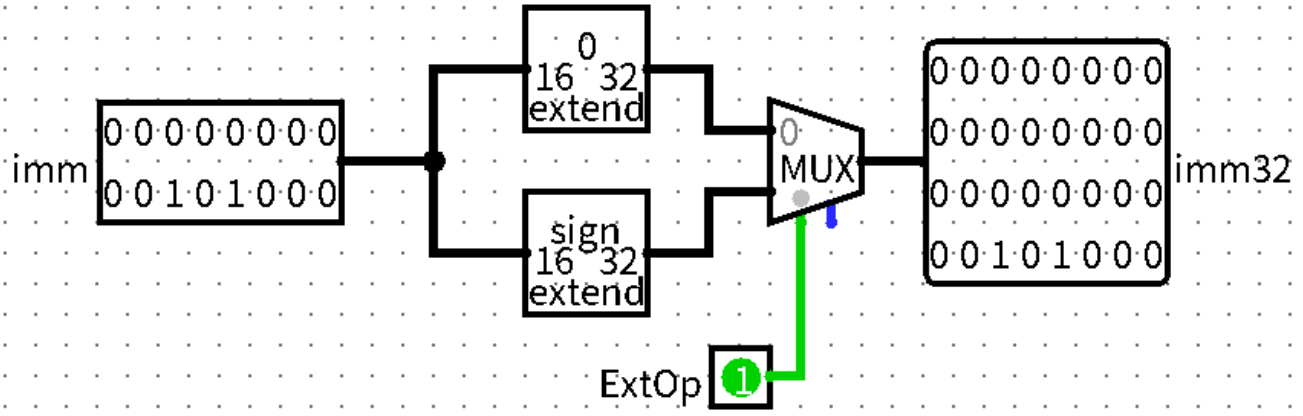
> 2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

回答第二个思考题。似乎是合理的？ROM 可读而不可写，用在 IM 似乎符合我们课程内对测试的需求，但如果考虑实际应用，对 CPU 输入的指令应该是能够实时增加的（毕竟人要操作），所以现实中可能这个模块更像个 RAM。DM 使用 RAM ，应该是没有是没问题的。GRF 使用 Register ，似乎也没有什么问题，毕竟在架构中确确实实是 32 个寄存器，但我认为或许换成一个容量足够的 RAM 也是可以的?

EXT（扩展单元）

- 使用 Bit Extender 。

信号名	方向	描述
imm	I	16位 Instr 读取的原始立即数
ExtOp	I	扩展方式信号
imm32	O	32 位扩展后的立即数



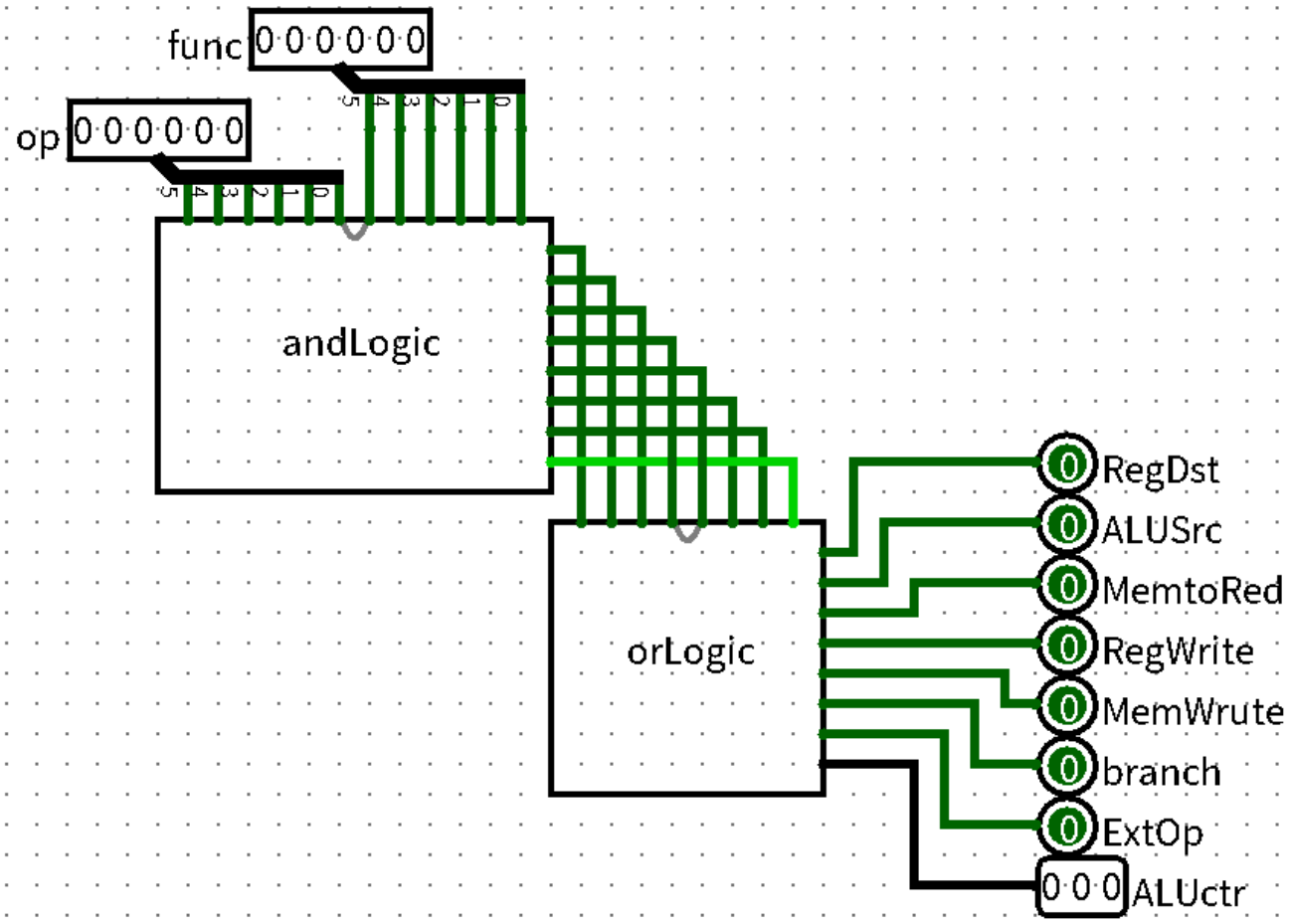
弄了个判断 0 扩展和符号扩展的小电路。

Controller（控制器）

See MIPS Green Sheet							
	func	10 0000	10 0010	n/a			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100
		add	**sub**	**ori**	**lw**	**sw**	**beq**
Control Signals	**RegDst**	1	1	0	0	X	X
	ALUSrc	0	0	1	1	1	0
	MemtoReg	0	0	0	1	X	X
	RegWrite	1	1	1	1	0	0
	MemWrite	0	0	0	0	1	0
	nPC_sel	0	0	0	0	0	1
	ExtOp	X	X	0	1	1	X
	ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract
		All Supported Instructions					

信号名	方向	描述
func	I	6 位 func 码信号
op	I	6 位 op 码信号
RegDst	O	控制信号，确定目标寄存器的指定方式 0：将结果写入 `rt` 寄存器（I 型指令） 1：将结果写入 `rd` 寄存器（R 型指令）
ALUSrc	O	控制信号，选择 ALU 的第二个源操作数 0：来自寄存器文件的一个寄存器 1：来自指令中的立即数字段

信号名	方向	描述
MemtoReg	O	控制信号，控制数据从内存到寄存器的传输 0：执行指令后，不将内存中的数据写入寄存器文件（通常发生在 ALU 指令） 1：执行指令后，将内存中的数据写入寄存器文件（通常发生在执行加载指令）
RegWrite	O	控制信号，控制数据是否从内部数据路径写入寄存器文件 0：不应该将执行结果写入寄存器文件 1：应该将执行结果写入寄存器文件中的一个指定寄存器
MemWrite	O	控制信号，指示是否应该将数据从数据缓存写入DM
branch	O	控制信号，表示是否跳转（是否 beq）
ExtOp	O	控制信号，操作数扩展信号 0：0扩展 1：符号扩展
ALUctr	O	3 位控制信号，选择 ALU 运算符



和逻辑

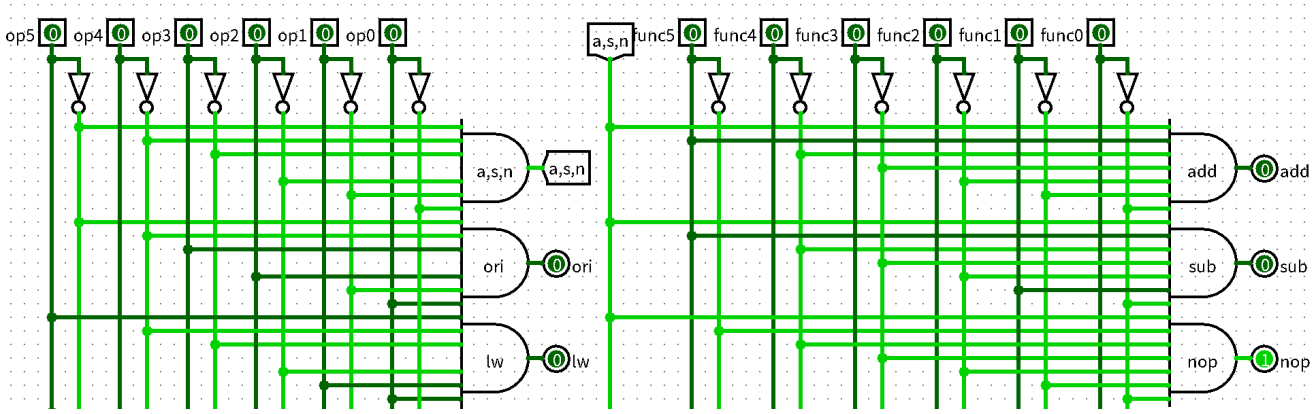
根据机器码中的 op 和 func 进行译码，判断出指令的类型，传递给或逻辑：

指令	op5	op4	op3	op2	op1	op0
add	0	0	0	0	0	0
sub	0	0	0	0	0	0
ori	0	0	1	1	0	1
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0
lui	0	0	1	1	1	1
nop	0	0	0	0	0	0

其中，使用 func 区分 `add, sub, nop`：

指令	func5	func4	func3	func2	func1	func0
add	1	0	0	0	0	0
sub	1	0	0	0	1	0
nop	0	0	0	0	0	0

有了真值表，可以直接在 Logisim 中自动生成组合逻辑电路。（太复杂！还是自己搭吧……）

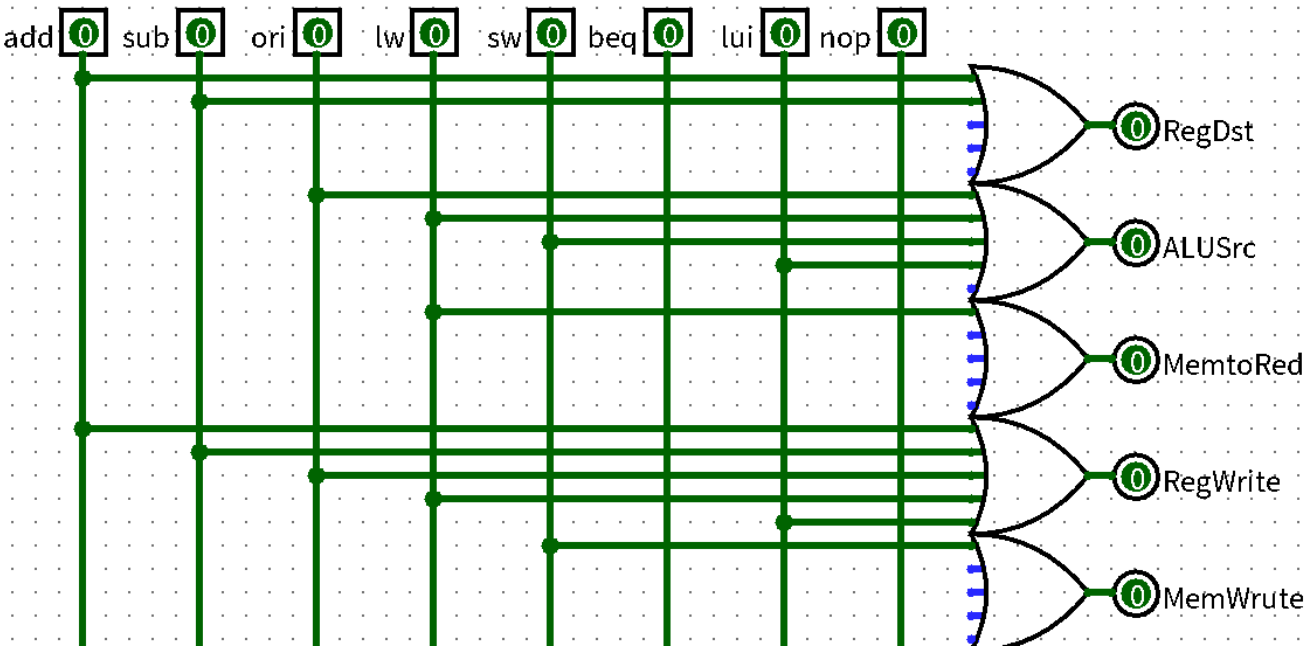


只展示一部分了。这里 beq 的和逻辑和或逻辑我一开始都搭错了，喜提两个 WA 。

或逻辑

指令	add	sub	ori	lw	sw	beq	lui	nop
RegDst	1	1						
ALUSrc			1	1	1		1	
MemtoReg				1				

指令	add	sub	ori	lw	sw	beq	lui	nop
RegWrite	1	1	1	1			1	
MemWrite					1			
branch						1		
ExtOp				1	1	1		
ALUctr	000	001	010	000	000	001	100	

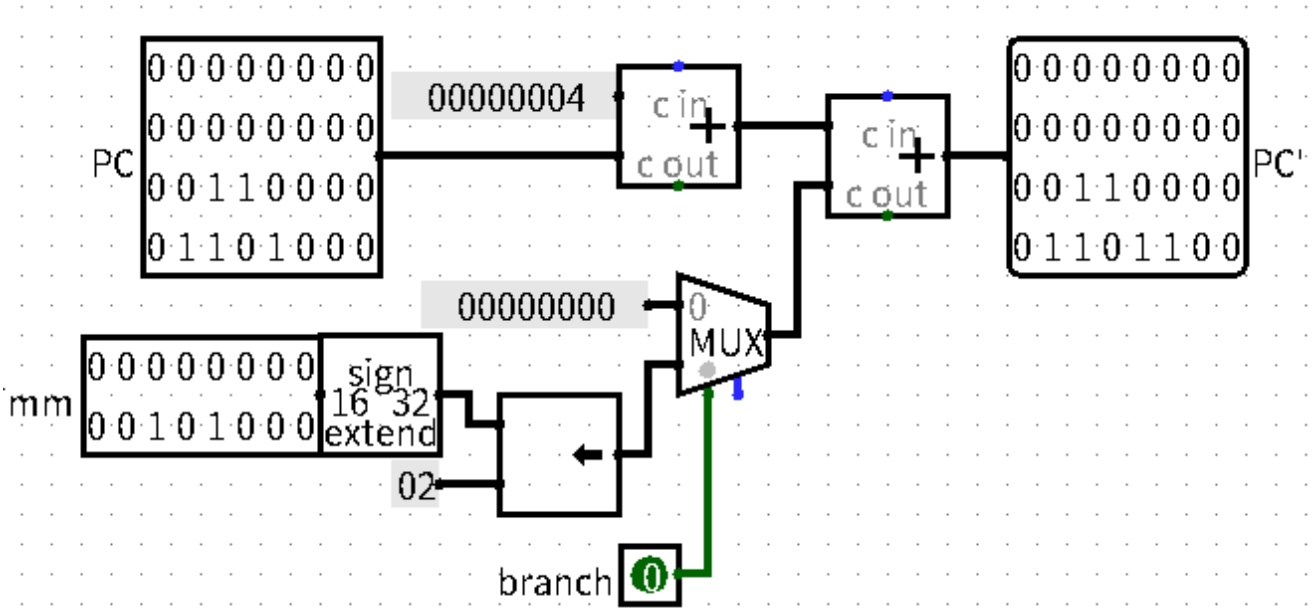


> 4. 事实上，实现 `nop` 空指令，我们并不需要将它加入控制信号真值表，为什么？

回答第四个思考题。虽然我在模块中添加了 nop ，但一通操作下来 nop 没有起到任何作用，没有输出任何控制信号，因此确实是可以删除的。在这个 CPU 中，控制读写有严格的使能信号约束，不需要担心如果不考虑 nop 会发生错误。

NPC（PC 跳转模块）

信号名	方向	描述
PC	I	32 位初始 PC 信号
Imm	I	16 位立即数信号，表示偏移量
branch	I	是否按照立即数偏移
PC‘	O	32 位更新 PC 信号



> 3. 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。

回答了第三个思考题，这个模块用于计算下一个 PC 的值。注意到，如果没有发生跳转（不是 beq），那么直接在原基础上+4；如果需要跳转，那么 PC'=PC + 4 + offset * 4，为什么要乘 4 呢，因为偏移 1 相当于 PC 增加 4 ，并且注意是符号扩展，要处理负数偏移的情况。

main（顶层模块组装）

机器码指令

R 类型指令

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

其中，op 为**操作码**，funct 为**函数**，所有 R 类型指令操作码都为 0 ，根据 funct 确定对应操作（在上文已经给出），在本次项目中，`add, sub` 是 R 类型指令。
rs 和 rt 是源寄存器，rd 是目的寄存器，形象理解为：

```
add rd, rs, rt
sub rd, rs, rt
```

shamt 是偏移数，本次不涉及，为 0 。

I 类型指令

op	rs	rt	offset or imm
6bit	5bit	5bit	16bit

本次项目中，`ori, lw, sw, beq, lui` 是 I 类型指令。
在不同指令中，目标寄存器和源寄存器指定不同，形象理解为：

```
ori rt, rs, imm
lw rt, offset(rs)
sw rt, offset(rs)
beq rs, rt, offset
lui rt, imm
```

J 类型指令

op	address
6bit	26bit

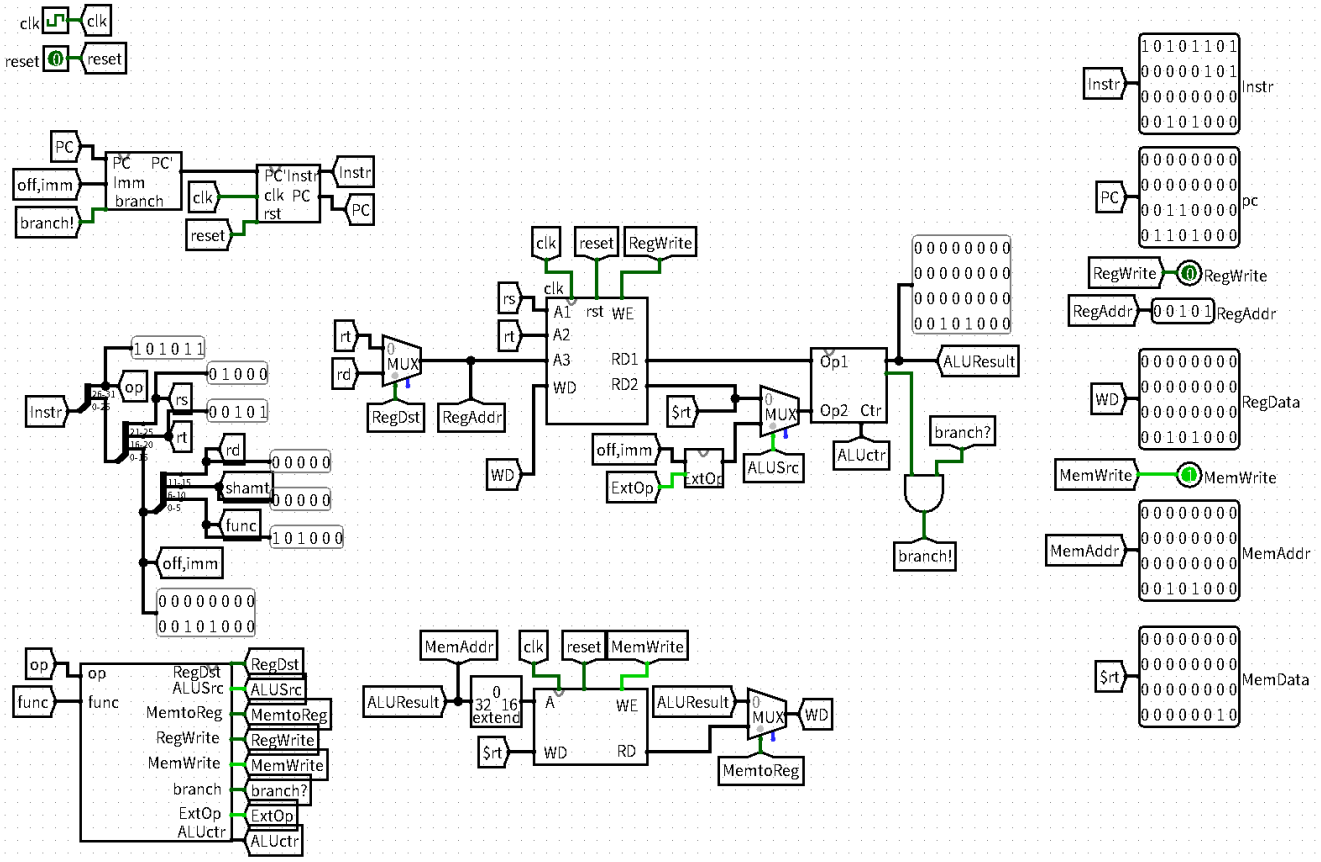
暂时不涉及。

寄存器编码

reg	name	usage
$0	$zero	常量 0
$1	$at	保留给汇编器使用的临时变量
$2-$3	$v0-$v1	函数调用返回值
$4-$7	$a0-$a3	函数调用参数
$8-$15	$t0-$t7	临时变量
$16-$23	$s0-$s7	需要保存的变量
$24-$25	$t8-$t9	临时变量
$26-$27	$k0-$k1	留给操作系统使用
$28	$gp	全局指针
$29	$sp	堆栈指针
$30	$fp	帧指针
$31	$ra	返回地址

要点简析

获取到 Instr，使用 Splitter 将 32 位机器码分成不同的部分，参考上面提到的机器码格式。使用 op 判断指令类型，并输出相应的控制信号。对于 rt 和 rd，需要判断指令的类型，再作进一步筛选输入 GRF 模块。注意立即数的扩展方式。理解 DM 和 GRF 的区别和联系：GRF 是 32 个寄存器，DM 是内存，寄存器中存有 DM 的基地址，根据这个和偏移量去对 DM 操作。形象理解：GRF 是 `$v0` 这种东西，DM 是 `.data` 字段的数组啥啥的。（可能不准确，但目前看来似乎可以这么理解）



> 1. 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。

回答第一个思考题，状态存储：GRF、DM；状态转移：Controller。

> 5. 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。

回答最后一个思考题。指令覆盖情况：没有测试 sub 指令，没有测试 nop 指令。对于单一指令的行为：add 测试了正正、正负、负负，没有测试 0；ori 测试了 0 正，0 负，没有测试正负，且没有测试立即数为 0 或负数的情况；sw 和 lw 没有测试负偏移量的情况；beq 仅测试了跳转到后续指令的情况，没有测试跳转到之前指令的情况。