华东师范大学软件工程学院实践报告

课程名称： 计算机组成与实践	**年级：2023 级**	**上机实践成绩：**
指导教师： 谷守珍	**姓名：** 张建夫	**上机实践日期：** 6/3~6/17
实践编号： 实验4	**学号：** 10235101477	**上机实践时间：** 4 学时

一、 实验名称

单时钟周期MIPS CPU设计实验

二、 实验目的

- 掌握硬布线控制器设计的基本原理
- 能利用相关原理在logisim平台中设计实现MIPS单周期CPU

三、 实验内容

- 绘制MIPS CPU数据通路
- 实现单周期硬布线控制器
- 测试联调

四、 实验原理

本次实验需要实现的具体指令如下：

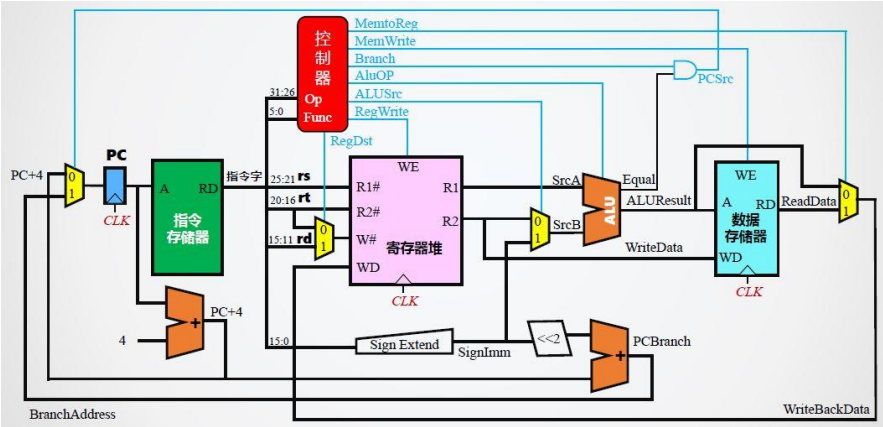
#	MIPS指令	RTL功能描述
1	add $rd,$rs,$rt	R[$rd]←R[$rs]+R[$rt]　溢出时产生异常，且不修改R[$rd]
2	slt $rd,$rs,$rt	R[$rd]←R[$rs]<R[$rt]　小于置1，有符号比较
3	addi $rt,$rs,imm	R[$rt]←R[$rs]+$SignExt_{16b}$(imm)　　溢出产生异常
4	lw $rt,imm($rs)	R[$rt]←$Mem_{4B}$(R[$rs]+$SignExt_{16b}$(imm))
5	sw $rt,imm($rs)	Mem_{4B}(R[$rs]+$SignExt_{16b}$(imm))←R[$rt]
6	beq $rs,$rt,imm	if(R[$rs] = R[$rt]) PC ← PC + $SignExt_{18b}$({imm, 00})
7	bne $rs,$rt,imm	if(R[$rs] != R[$rt]) PC ← PC + $SignExt_{18b}$({imm, 00})
8	syscall	系统调用，这里用于停机

我们先根据学习到的mips指令类别对这些指令分一个类，对于前两条指令（加法指令和小于则置位）是R型指令，剩下的除了syscall外都是i型指令，对于syscall，在硬布线控制器那个部分有一行字进行了说明：

注意R_TYPE表示R型运算指令，SYSCALL是特殊的R型指令，不属于这个类别

也就是说，syscall的opcode和r型指令的opcode一样，为0b000000，但是后面的func不一样。

在明确了需要实现的指令只有R型和i型之后，我们就可以根据之前ppt的内容进行实现，由于ppt的内容实际上还实现了j型指令，我们需要根据ppt的内容进行删减，而实验的ppt也给出了实现的大体框架：



而对于硬布线控制器的实现，实验的ppt也给出了具体的要求：

#	控制信号	信号说明	产生条件
1	MemToReg	写入寄存器的数据来自存储器	lw指令
2	MemWrite	写内存控制信号	sw指令　未单独设置MemRead信号
3	Beq	Beq指令译码信号	Beq指令
4	Bne	Bne指令译码信号	Bne指令
5	AluOP	运算器操作控制符	加法，比较两种运算
6	AluSrcB	运算器第二输入选择	Lw指令，sw指令，addi
7	RegWrite	寄存器写使能控制信号	寄存器写回信号
8	RegDst	写入寄存器选择控制信号	R型指令
9	Halt	停机信号，取反后控制PC使能端	syscall指令

控制器应该根据实验要求实现的指令给出相应的控制信号。

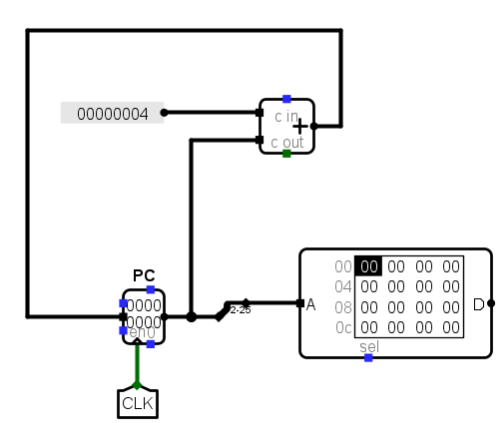
五、 实验过程

对于单时钟的数据通路，我将各个功能都进行了模块化，根据指令取指，译码，执行，写回的顺序来进行实现，由于后面的部分会影响前面的实现（如BEQ指令执行的结果可能会影响pc是+4还是跳转），我可能会在做后面的部分的时候又跳回去完善前面的部分，在全部部分搭建好后，我又按照ppt的要求实现了单周期的布线控制器，最后测试。

（1） 第一部分（实现单时钟的数据通路）

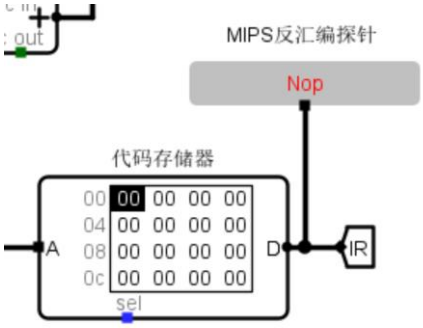
1. 指令取指：

该部分我最先进行实现的是pc简单的自增，并将pc的值传送到一个ROM中实现取指：



这里我简单地使用了一个加法器实现PC的自增，由于mips的指令要求字对齐（即四个字节对齐），因此指令地址的值一定是4的倍数，意味着PC低两位始终为0，也就是说我们不需要这两位，因此此处是从2开始传送地址的，另外，由于logism的ROM最高支持24位地址，这里就使用了24位地址，因此PC向ROM的地址输入是2~25.

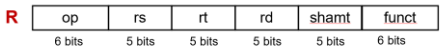
然后我们将取指的结果用隧道IR保存起来，方便我们后续译码：



2. 指令译码：

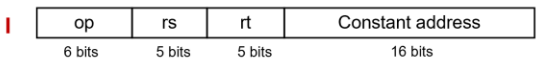
译码的部分较为简单，在这个部分我就顺便将控制器的外部连线也做了。

根据前面的分析，译码需要将指令分为R型指令和i型指令，这两种指令的结构如下图所示：



指令中各字段名称及含义如下：

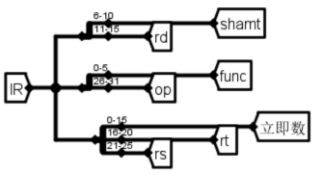
- op：指令的基本操作，通常称为操作码（opcode）
- rs：第一个源操作数寄存器
- rt：第二个源操作数寄存器
- rd：用于存放操作结果的目的寄存器
- shamt：位移量
- funct：功能码，用于指明op字段中操作的特定变式



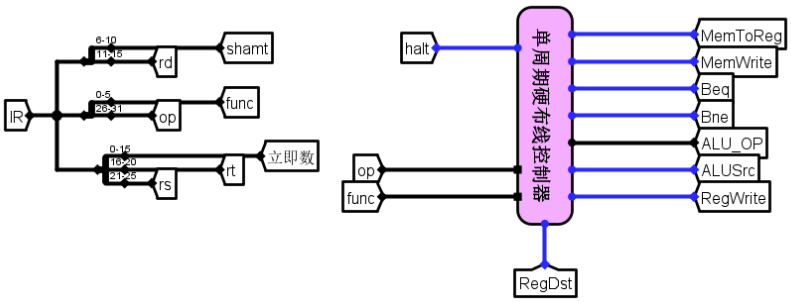
指令中各字段名称及含义如下：

- op：指令的基本操作，通常称为操作码（opcode）
- rs：源操作数寄存器
- rt：目的操作数寄存器
- Constant address：常数或地址

由于这两种指令的每个字段都可能用到（事实上本次实验中shamt字段并没用到），我用了三个splitter将每个字段都用一个隧道进行保存，方便后续使用：

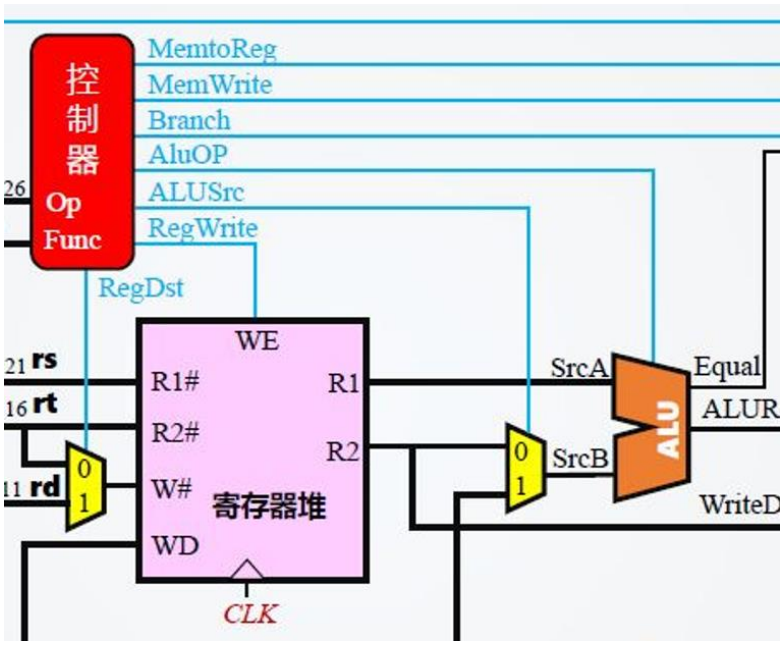


接着，有了op和func字段，就可以完成对控制器的输入了，与之前一样，为方便后续使用，控制器的每个输入都用一个隧道进行保存：

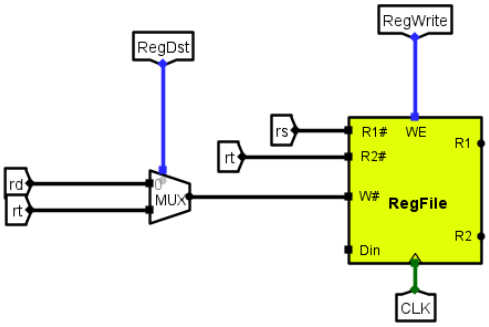


3. 指令执行：

该部分包含了从寄存器文件中读出具体寄存器的值，并根据条件传送到alu计算，我是先根据ppt的内容进行连线：



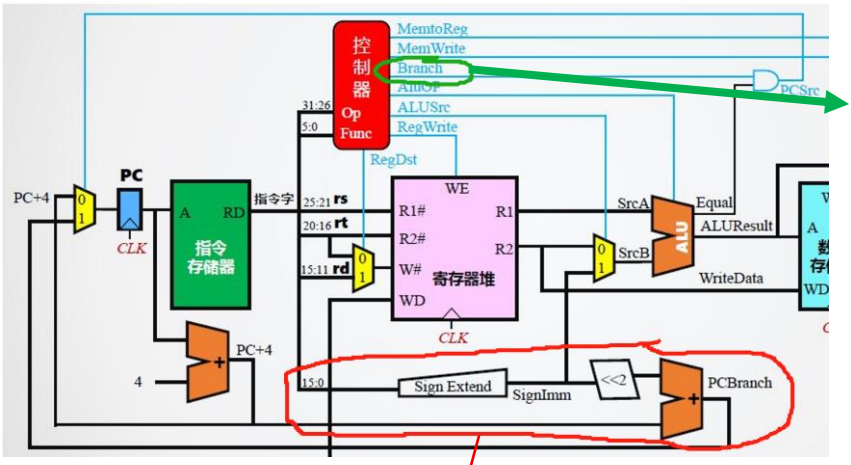
根据ppt上的数据通路，RegDst用于决定写入的目标寄存器是rd还是rt（即是load指令还是R型指令），我便先连了一个寄存器文件：



（此处我连的rd和rt是和ppt上是反的，我当时没发现，不过由于还没实现硬布线控制器，这个“错误”可以在后面纠正）

接着实现寄存器文件传送数据到alu的部分，在这个部分通过观察ppt上的数据通路，

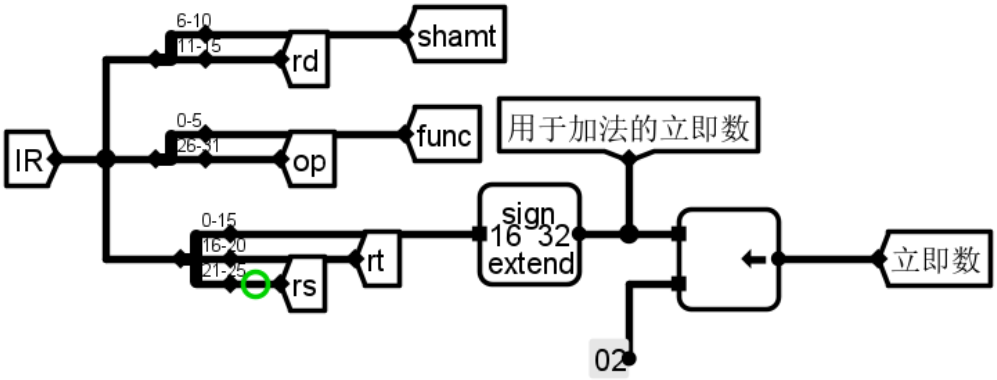
可以发现我们要改变部分先前已经做好的电路图：



控制器输出 branch 信号说明该指令是一个条件跳转指令，而在实验里面，跳转信号有两个(beq 和 bne)，因此这两个信号要分别与 alu 计算出来的 equal 进行与操作，判断 PC 下一个指令是否跳转。

对于译码出来的立即数，有两个用途，一个是 i 型指令的立即数计算，另一个是跳转指令的地址计算

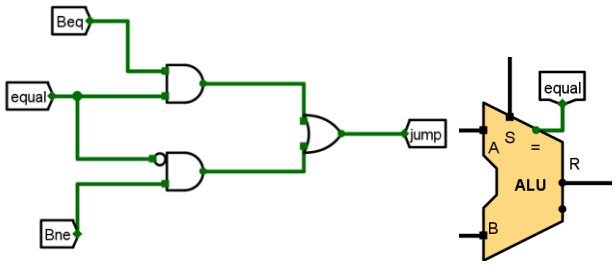
我们首先处理红色框框的部分，该部分需要我们调整立即数的译码：



在这个部分中，我将立即数的译码结果分为两个，分别对应上面分析的两个用途，同时这里有一个位扩展器的实现细节，这里应采用符号扩展而不是0扩展，addi指令可能含有负数。

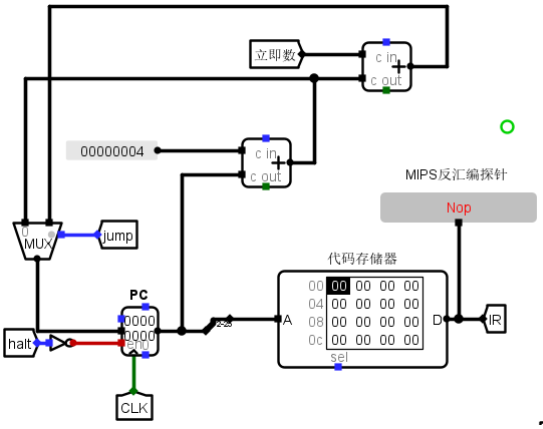
另外，这里“用于加法的立即数”左移两位的原因是 在第二个用途中 跳转指令的地址是字对齐，低两位始终为0，这里的也就保证了PC与其相加后仍然是字对齐。

接下来是绿色框框的部分讲解，由于本实验有bne指令，而图里实际只考虑了beq指令的实现，因此，如果要产生跳转信号，就有两种可能，一种是使用beq指令，且alu的计算结果产生了equal信号，另一种是bne信号，且alu未产生equal信号，因此我们能画出如下电路图：



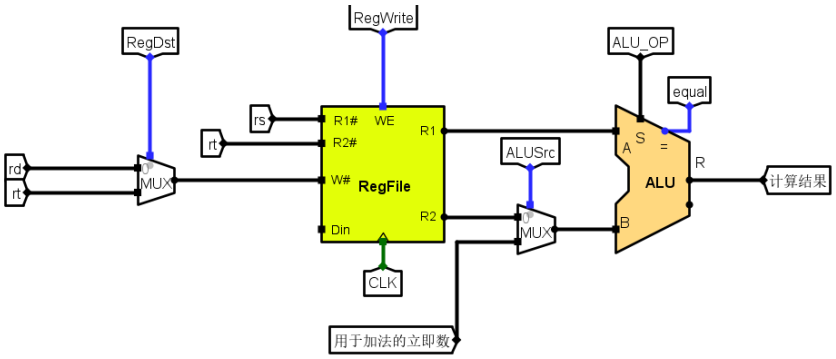
（equal由alu得出，这里我用jump表示最终的结果，如果jump为1则跳转）

接着，我们要重新修改取指的部分，来最终实现跳转指令的正确运行（而不只有PC加4）：



这里我也顺便将halt接了上去，相当于实现了syscall停机指令。

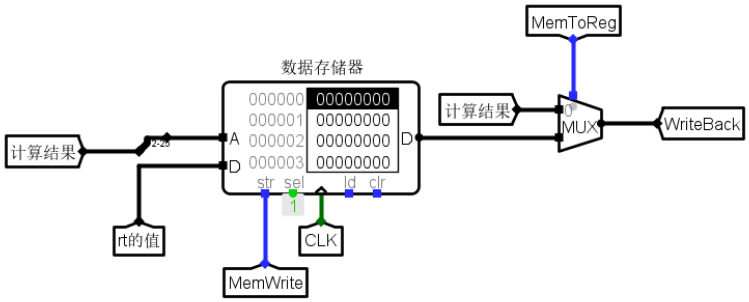
最后根据ppt的连线将alu和寄存器文件串起来：



为了保证我之前模块化的设计思路，这里alu的计算结果用了隧道保存，可以看到此处的Din（用于寄存器写回的数据）为空，这是因为在下一个部分才会得到写回的数据。

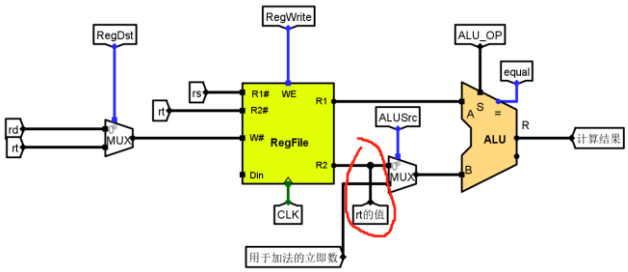
4. 写回：

这一部分也是根据ppt的内容直接连线，但是，由于原本的文件中并没有数据存储器，因此要使用logism自带的RAM：



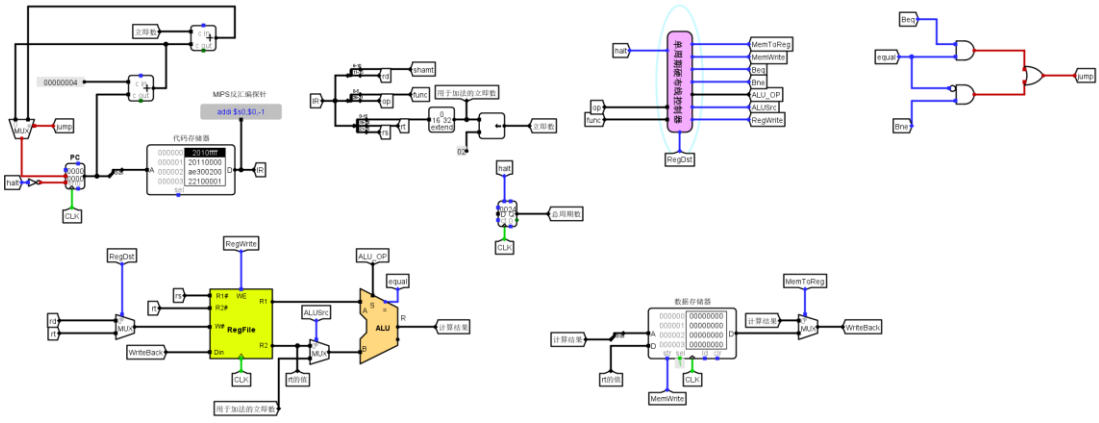
这里有几个实现细节值得说一下：

1. Alu的计算结果取的是2~25位，这是因为指令是按照字对齐取数据的，且RAM最多支持24位，和前面PC取指同理；
2. “rt的值”：这个值就是R2选出来的值，根据ppt，我在原先的寄存器文件到alu的部分加了一个隧道来取得该值：



这个值用作sw指令中写到内存里面的值。

在所有上述工作完成后，我还实现了一个周期的计数器，方便后续的调试，下面是所有模块的展示：

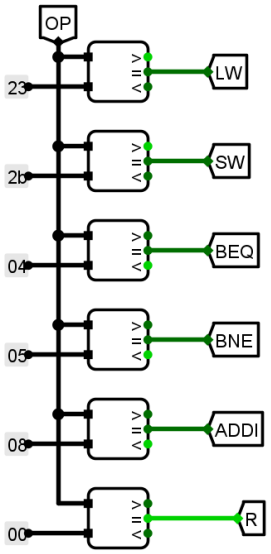


（2） 第二部分（实现单周期硬布线控制器）：

这一部分的实现要比单周期mips数据通路的实现要简单得多，我先实现的是指令译码逻辑，网上查询资料我们能够得知对应指令的opcode如下：

Mips指令	Op码
Add（两个寄存器的值相加，并将结果存储在目标寄存器中。）	000000
Slt（比较两个寄存器的值，并将结果（0 或 1）存储在目标寄存器中）第一个小于第二个置1	000000
ADDI（用于将一个立即数添加到目标寄存器的值中。）	001000
Lw（用于从内存中加载一个32位的字到目标寄存器中。）	100011
Sw（用于将一个32位的字从源寄存器存储到内存中。）	101011
Beq（比较两个寄存器的值，值相等，那么程序将跳转到指定的地址。）	000100
Bne（不相等跳转）	000101
Syscall（系统调用停机）	000000

于是可以对OP先进行译码：



这里使用logism的比较器来实现译码，这个是最方便的。

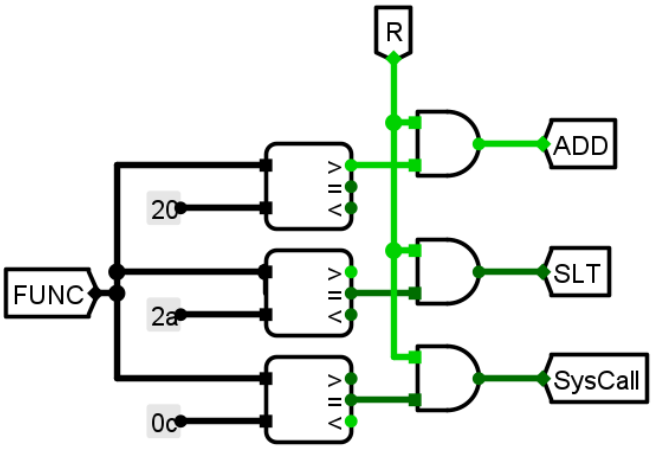
由于opcode为0的情况是指使用R型指令，我们还需要根据R型指令的func字段进一步区分是ADD指令，SLT指令还是syscall，而这三个的对应指令编号如下：

指令操作	funct字段	ALU操作	ALUcontrol
add	100000	add	0010
slt	101010	slt	0111

funct	指令助记符	指令功能描述	备注
00	sll rd,rt,shamt	R[rd]=R[rt]<<shamt	逻辑左移，注意 rs 字段未使用
02	srl rd,rt,shamt	R[rd]=R[rt]>>shamt	逻辑右移，注意 rs 字段未使用
03	sra rd,rt,shamt	R[rd]=R[rt]>>shamt	算术右移，注意 rs 字段未使用
04	sllv rd,rt,rs	R[rd]=R[rt]<<R[rs]	可变左移
08	jr rs	PC=R[rs]	R[rs] 值应是 4 的倍数，字对齐
09	jalr rs	R[31]=PC+8 PC=R[rs]	子程序调用
12	syscall	系统调用	无操作数

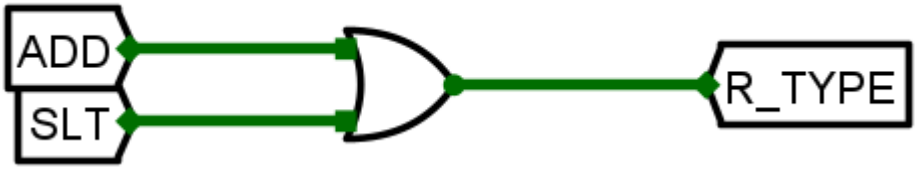
Add和slt的func字段是从ppt上摘下来的，而对于系统调用则是从网上找的。

得知对应的编码后就可以画出译码器：



显然，由于这三个指令是只有在opcode为0时才能生效，因此要通过一个选择R型指令的与门。

最后是R_TYPE的电路图：

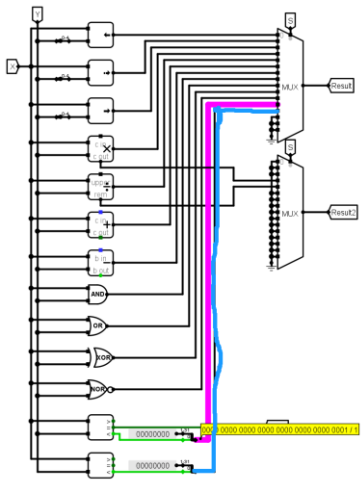


这里没有把syscall加进去是因为之前的这句话：

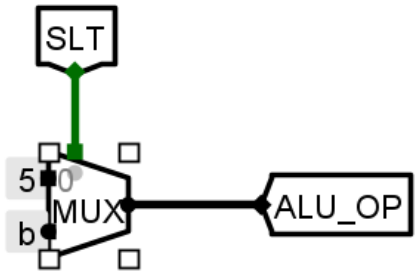
注意R_TYPE表示R型运算指令，SYSCALL是特殊的R型指令，不属于这个类别

然后我们实现alu的控制逻辑，这里有一个非常取巧的点，在需要进行alu运算的部分里面，所有的指令，除了slt外，其他指令都是执行加法（lw和sw是计算存储地址，而beq和bne是计算跳转地址，都是相加），也就是说，我们可以通过slt作为选择信号来输出对应的ALU_OP。

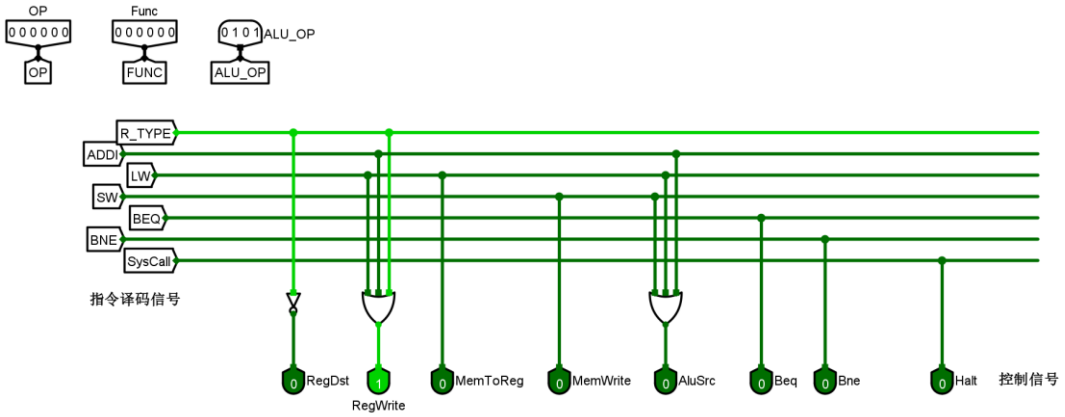
由于本实验中的ALU_OP与ppt的ALU_OP不一样，我打开了封装ALU的电路来获取正确的ALU_OP，不过在查看ALU_OP的过程中我发现了一些有意思的地方：



针对小于则置位，上面高亮的两条路径均可以输出正确的结果，16进制的表示为b或c，对于加法，16进制表示为5，于是我们只要一个简单的选择器就可以实现ALU_OP：



最后是信号的输出部分：



此处的连线参考了之前ppt的表格：

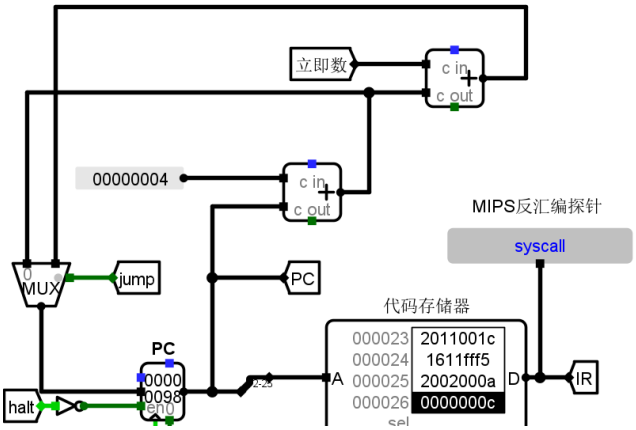
输入或输出	信号名	R型	lw	sw	beq
输入	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
输出	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

另外一点需要注意的是，RegDst此处取反是因为之前我将rd和rt的位置放反了，此处也要取反以实现正确的电路表示。

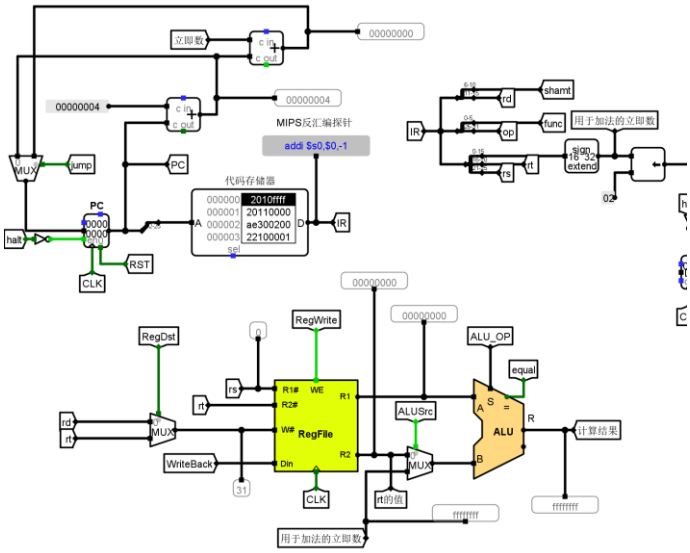
至此，所有实验电路搭建完毕。

六、 实验结果及分析

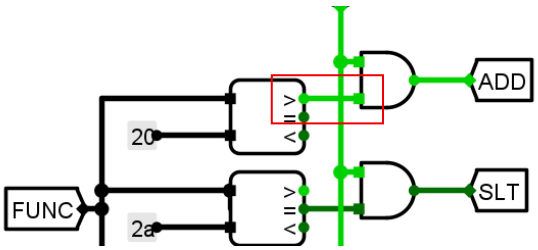
实验测试采用的是模板自带的测试方式，最开始我的测试一直停在第36个周期（理论上应该运行224个周期才结束）：



刚开始我一直以为是数据通路的实现有问题，在一堆地方加了探针：



一句一句指令执行（实际上就是很快地按两次ctrl+k，以实现单步调试），检查了多遍都感觉没有问题，后面在网上找了答案，以为是RAM加了片选信号的缘故（网上的答案没有使能片选），但显然不是，最后是从网上下了一个“答案”，一句一句用探针的值比较，才发现是控制器的信号出了问题：



（注意我的ADD译码接的是大于号）

这个问题卡了我几个小时，后面将这个改回来后实验才成功：

000080	00000006	00000005	00000004	00000003	00000002	00000001	00000000	ffffffff	0000
000090	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	0000

七、 心得体会

本次实验是这几次实验中最难的一次，上面写实验过程的时候我看起来写得思路比较清晰，实际上是试错了几次才走到正确的道路上，尤其是实验中有一些不懂的地方（不知道ppt为什么这样连线），回去复习了才大致搞懂这样连线的原因，同时，害怕连线连错，每次完成一个模块，我都会按照ppt的连线图去检查是否正确，防止全部做完了却找不出问题来，虽然最后还是没有成功一次过，但是在debug的过程中学到了很多logism的高级用法，使用探针进行debug是比较高效的，同时快速的ctrl+k能够实现单步调试，这些都是很好的使用经验。