一、设计草稿

CPU顶层架构模型

整个单周期CPU的工作流程如下

- 根据程序计数器PC从ROM中取出指令
- 译码，确定指令类型
- 根据指令类型执行操作
    - 读写寄存器
    - 读写内存
    - 更新程序计数器PC

> 依据上述流程，我们设计的 CPU 应包含以下几个功能模块
>
> - IFU（取指令单元，从存储指令的 ROM 模块中取出下一条指令的32位二进制码）
> - $Split$（将每条指令的 32 位二进制码分成分别表示寄存器号、`OpCode`码等的二进制码段）
> - $Controller$（控制器，根据 splitter 得到的 6 位`OpCode`码和 6 位`func`码确定指令的类型并输出对应的控制信号）
> - GRF（寄存器堆）
> - ALU（算术逻辑单元，实现指令需要的数学运算）
> - EXT（位扩展器，根据需要进行相应的位扩展）等基本部件
> - DM（数据存储器，内存）

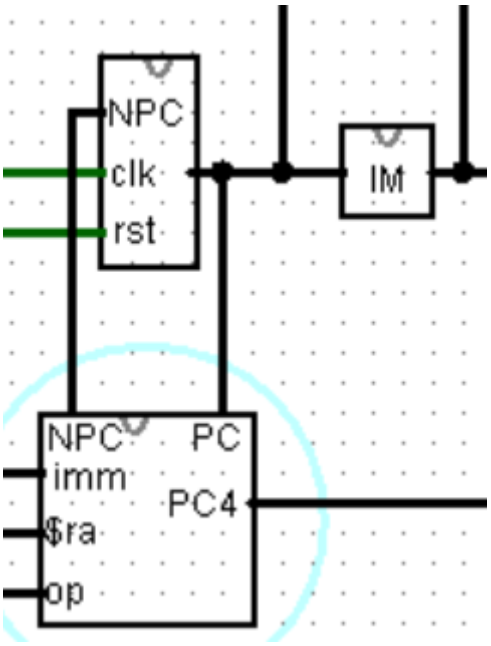
对本次实验所实现的指令，依据其执行的操作进行分类

功能	指令
读、写寄存器	add，sub，lui
读内存、写寄存器	lw
读寄存器、写内存	sw
读寄存器	beq

下面我们将结合各个指令的功能，完成各个功能模块的设计

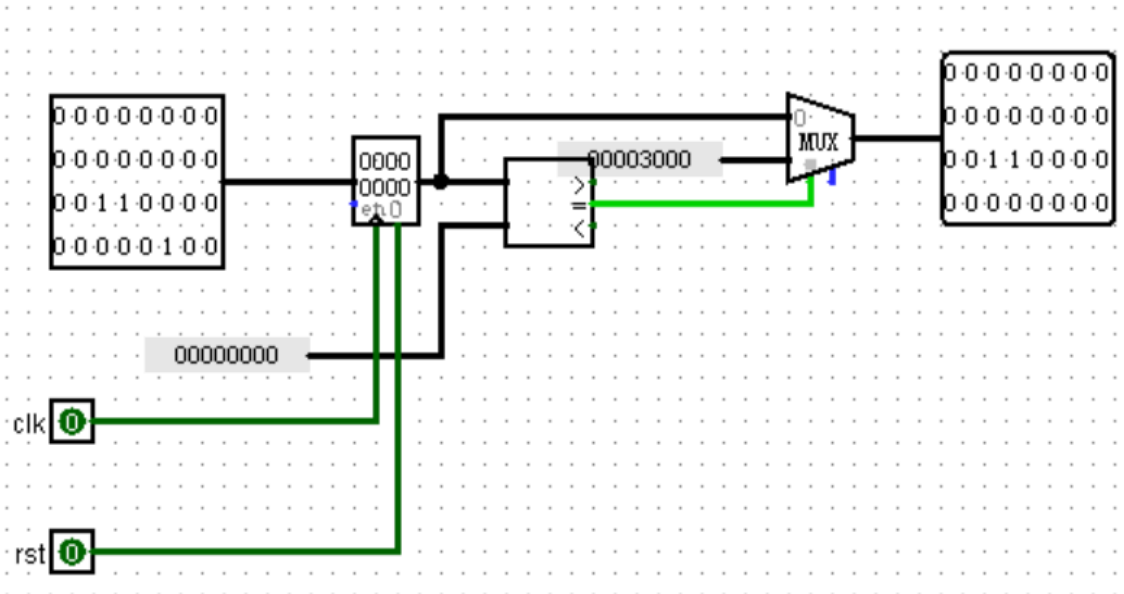
（一）IFU

根据PC从ROM中取出指令。由于每执行一条指令后，PC的值会随之更新，所以需要设计一个FSM实现PC的状态转移。由此，我们将IFU细分为PC（状态存储）、NPC（次态逻辑）、IM（取指令）三个模块



- **PC：**

端口名称	I/O	位宽	功能
NPC	I	32	输入寄存器的次态PC值
clk	I	1	时钟信号
rst	I	1	异步复位信号
PC	O	32	当前的PC值



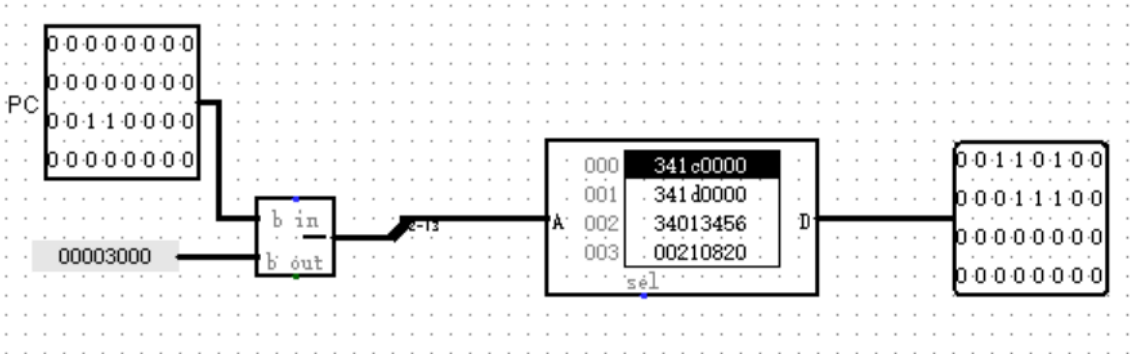
- **IM：**

  需要注意PC和ROM中指令地址的映射关系

  $addr = (PC - 0x00003000)/4$

端口名称	I/O	位宽	功能
PC	I	32	当前的PC值

端口名称	I/O	位宽	功能
instruction	O	32	取出的指令

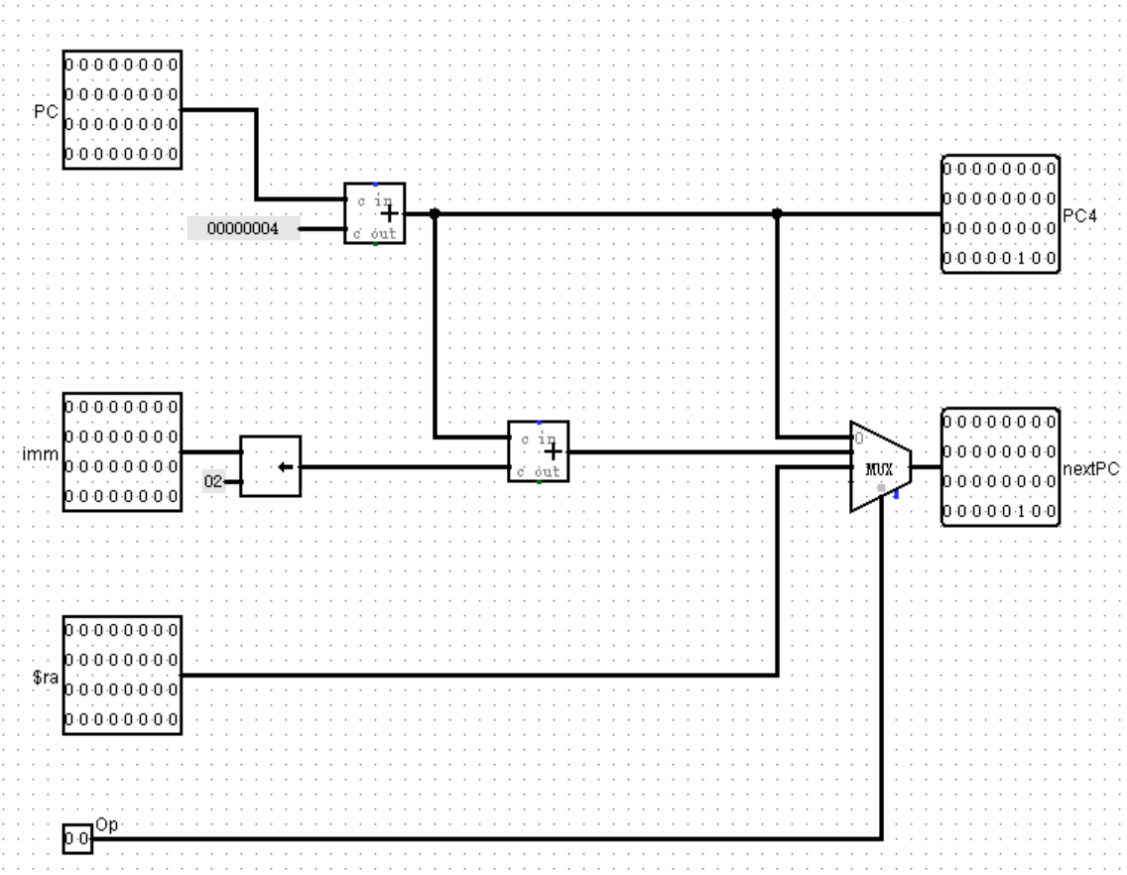


- **NPC:**

对于PC的次态逻辑，根据指令的类型，有以下四种情况：

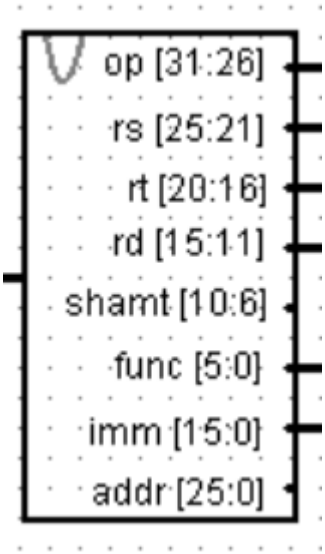
  - 通常情况下，如add、lw、sw等指令，$PC = PC + 4$
  - 分支指令，如beq、bne等指令，$PC = PC + 4 + signExtend(offset||0^2)$，即需要和一个符号扩展后的32位立即数运算
  - 跳转指令，如j、jal指令，$PC = (PC[31:28]||instrIndex||0^2)$，也是需要和一个符号扩展后的32位立即数运算，同时jal指令执行时，会将当前$PC + 4$的值储存在寄存器`$ra`当中
  - 跳转并链接指令，如jr、jalr指令，$PC = GPR[rs]$

端口名称	I/O	位宽	功能
PC	I	32	当前PC
Op	I	2	选择信号 00 : 计算顺序地址 01 : 计算beq地址 10 : 计算jr地址 11 : 计算jal地址
imm	I	32	参与计算的立即数（已符号扩展为32位）
$ra	I	32	jr所跳转的寄存器
PC4	O	32	PC + 4
nextPC	O	32	次态PC



（二）Split

利用分位器分出指令的各部分编码，封装在模块中，使电路更整洁

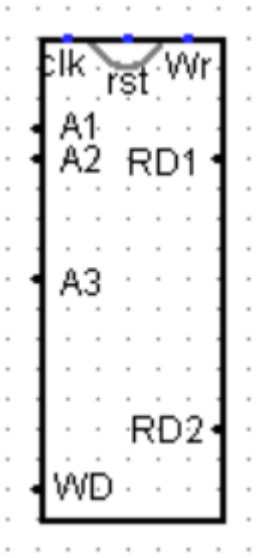


（三）GRF

已在P0课下时实现

端口名称	I/O	位宽	功能
clk	I	1	时钟信号
rst	I	1	异步复位信号
Wr	I	1	写使能信号

端口名称	I/O	位宽	功能
A1	I	5	5位地址读取信号
A2	I	5	5位地址读取信号
A3	I	5	5位地址写入信号，指定一个寄存器作为写入数据的目标
WD	I	32	32位数据写入信号
RD1	O	32	输出A1指定的寄存器中的32位数据
RD2	O	32	输出A1指定的寄存器中的32位数据



(四）ALU

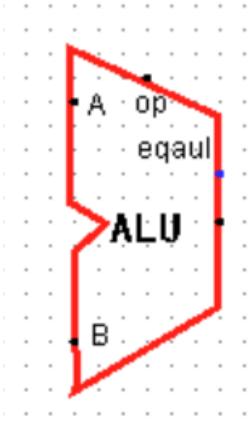
此模块主要是实现寄存器和立即数之间的运算。下表是本次实验的指令所涉及的运算

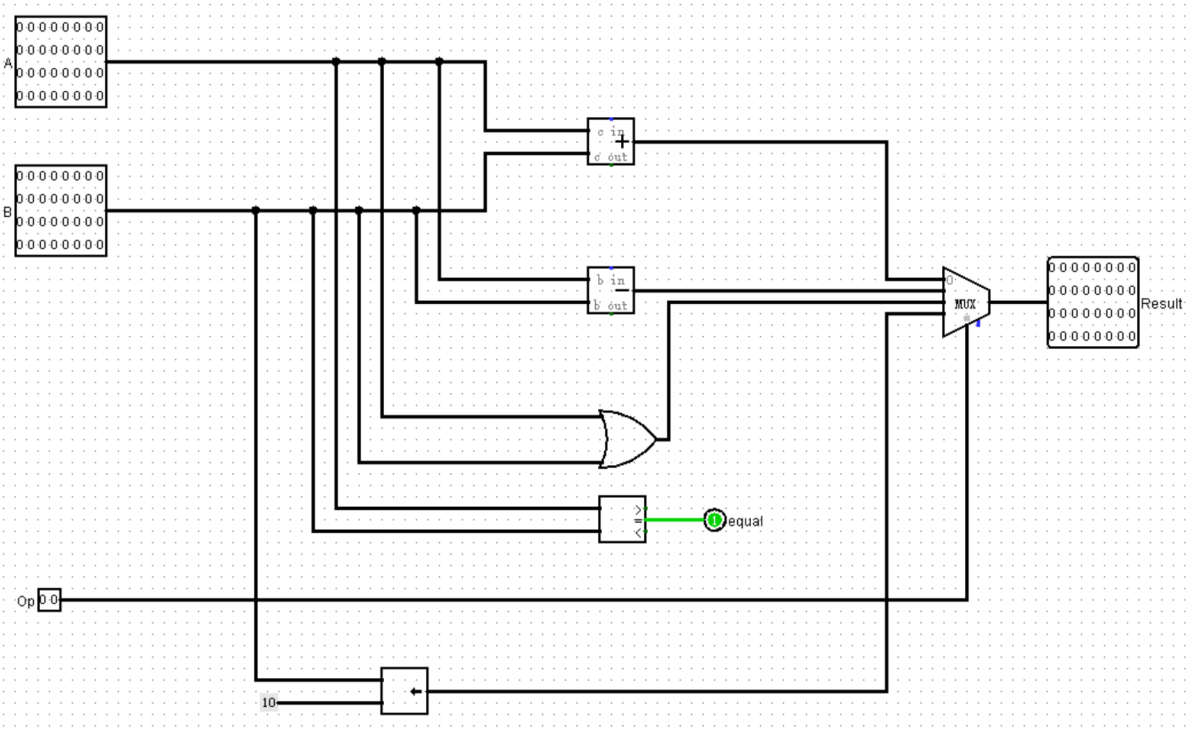
指令	运算
add	加（不考虑溢出）
lw，sw	加（寄存器加立即数得到内存地址）
sub	减（不考虑溢出）
ori	或
lui	左移（立即数加载至高位）
beq	判断是否相等

beq比较特殊，其不需要写寄存器或内存，且运算结果为0或1，我们可将其运算结果单独作为模块的一个输出；而剩下的运算结果用多路选择器输出。

端口名称	I/O	位宽	功能
A	I	32	输入运算数A

端口名称	I/O	位宽	功能
B	I	32	输入运算数B
Op	I	2（后续实验可能会扩展）	选择信号 00 : A + B 01 : A - B 10 : A | B 11 : B<<16
Result	O	32	运算结果
equal	O	1	0 ： A != B 1 : A == B

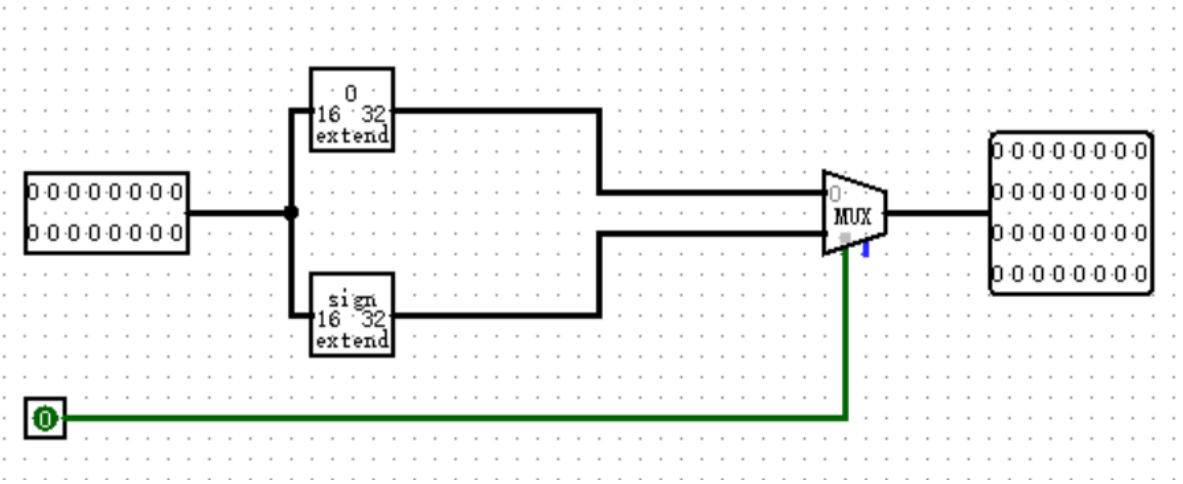




(五) Ext

端口名称	I/O	位宽	功能
imm	I	16	16位立即数

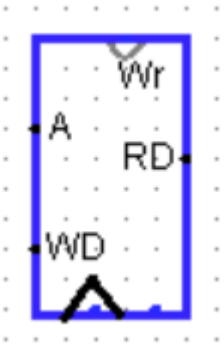
端口名称	I/O	位宽	功能
Op	I	1	0： 0扩展 1： 符号扩展
Result	O	32	扩展结果

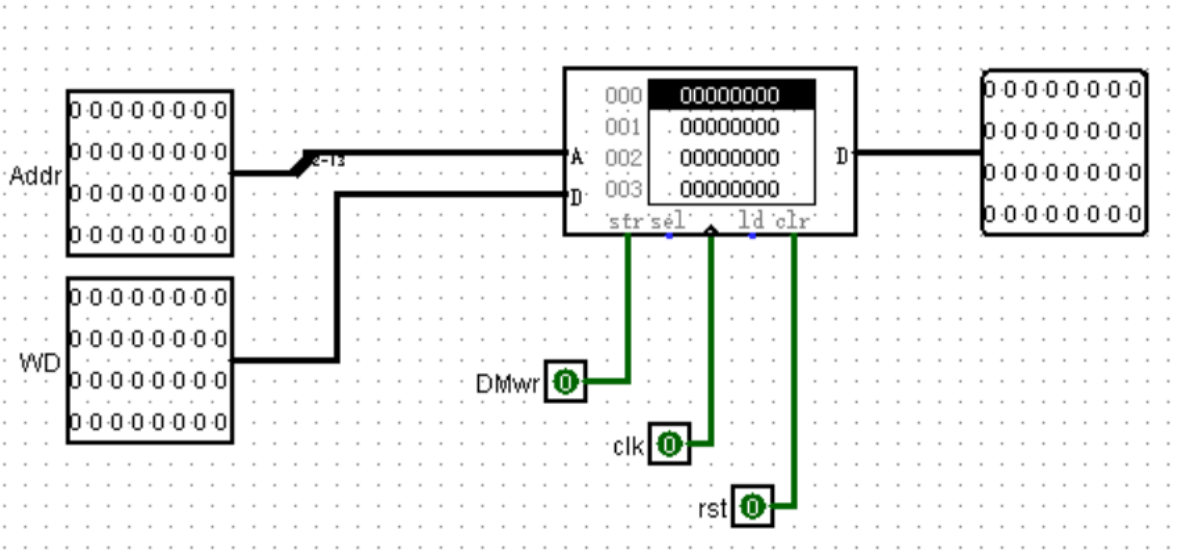


(六) DM

端口名称	I/O	位宽	功能
Addr	I	32	待操作的内存地址
WD	I	32	写入内存的数据
DMWr	I	1	写使能信号
clk	I	1	时钟信号
rst	I	1	异步复位信号
RD	O	32	Addr中储存的数据

需要注意Addr和RAM内存中地址的映射关系





（七）数据通路

至此已经完成了 $Controller$ 外各个模块的搭建，下面我们要结合控制信号，完成顶层电路的连接，最后再实现 $Controller$ 模块

- $Split$ 和 GRF 的连接

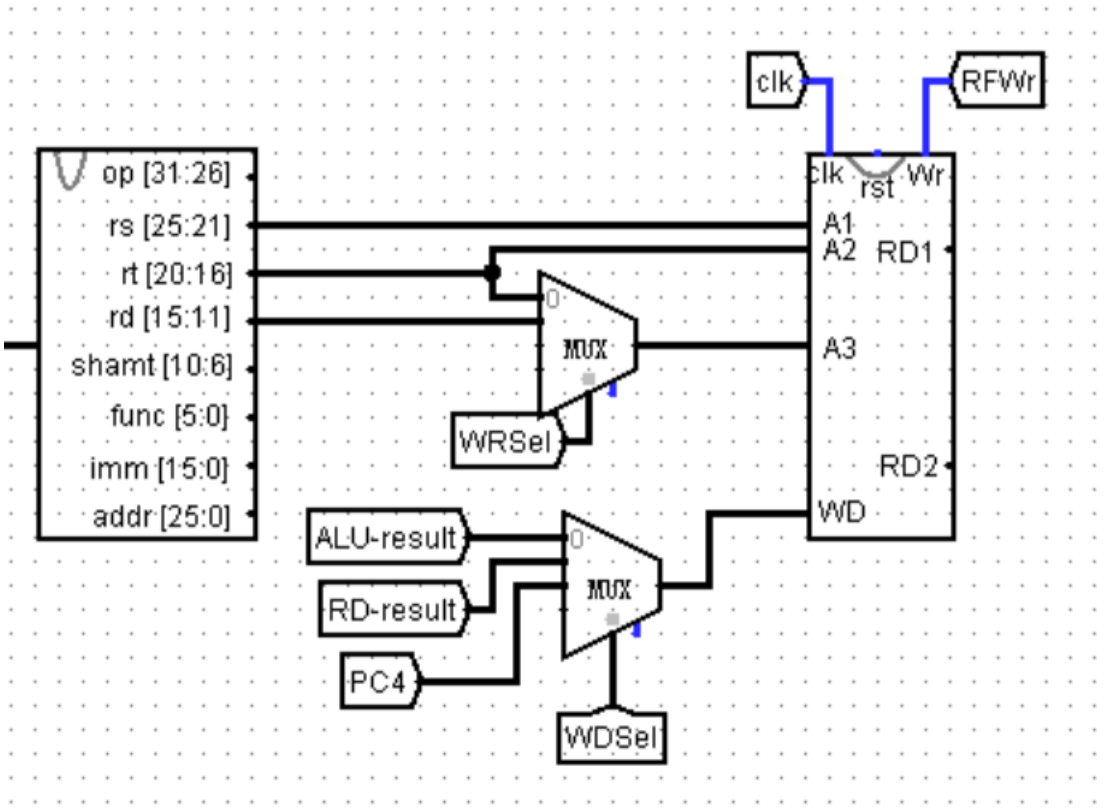
  首先设置**RFWr**信号选择是否执行写寄存器操作

  读写寄存器，需要仔细研究相关指令的RTL语言

  rs [25 : 21] 和rt [20 : 16] 总是被读取的寄存器，默认依次接在AD1和AD2端口；

  被写入的寄存器，根据指令类型，或是rt [20 : 16] ，或是rd [15 : 11]，于是设置**WRSel**信号进行选择；

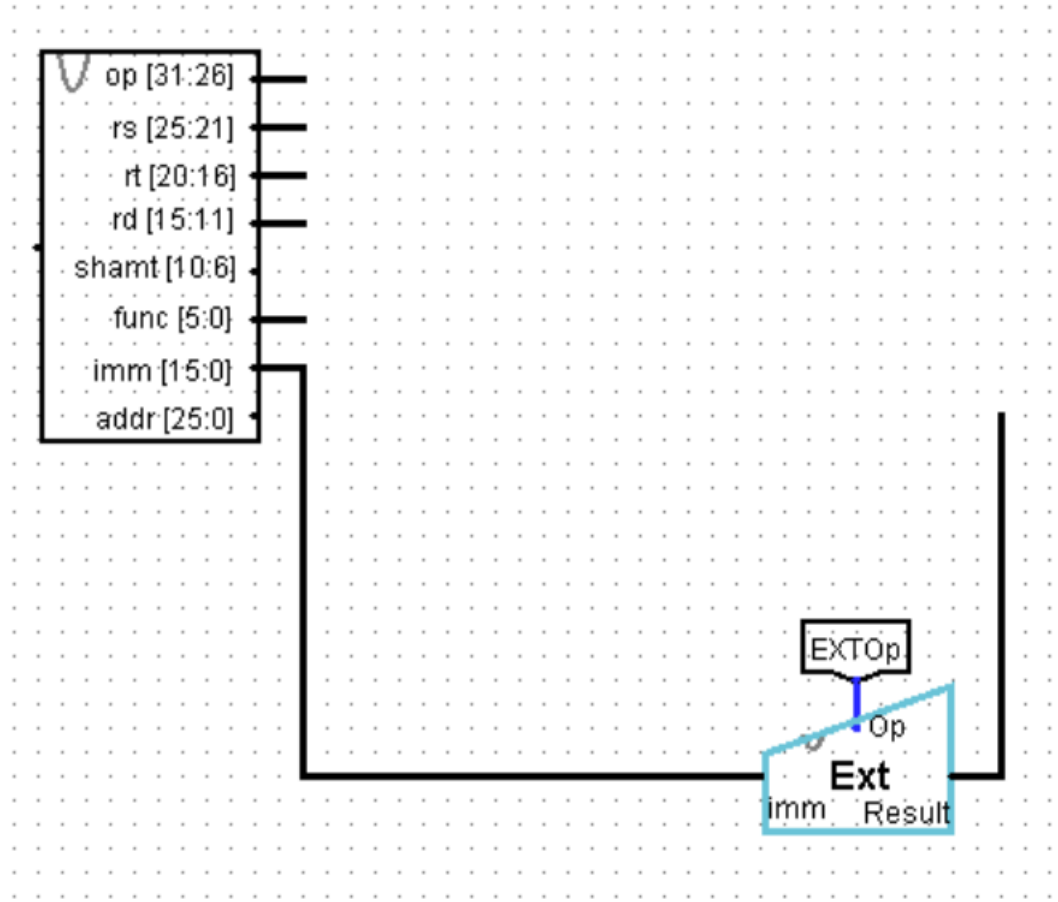
  被写入的数据，根据指令类型，或是ALU运算结果，或是RAM内存中的数据，或是PC4，于是设置**WDSel**进行选择。



- $Split$ 和 Ext 的连接

  将 imm [15 : 0] 接入Ext 输入端口

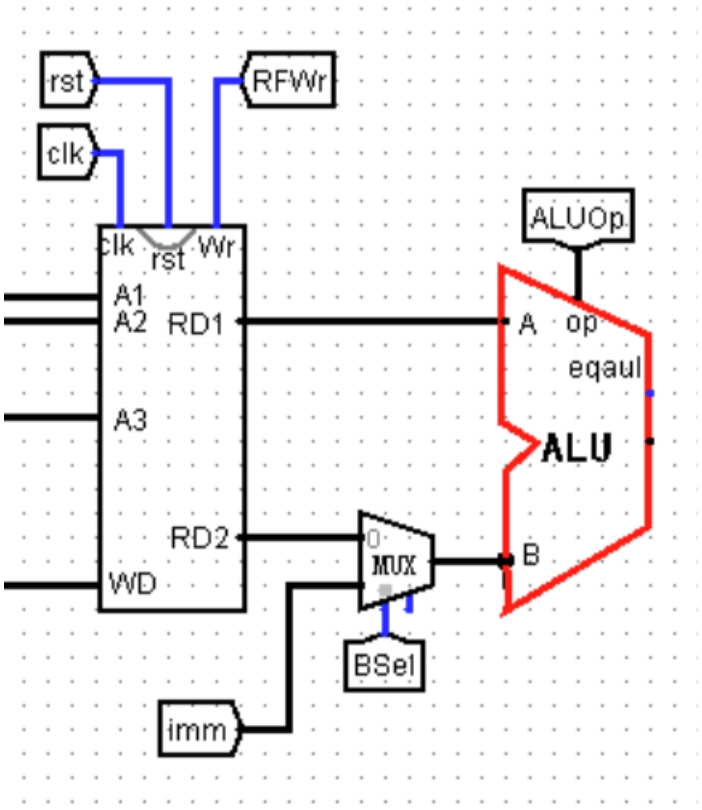
  设置**EXTOp**信号选择扩展方式



- GRF 和 ALU 的连接

  首先设置**ALUOp**信号选择是否执行运算操作

运算的对象，依据指令的不同，或是rs [25 : 21] 和rt [20 : 16] 运算，或是rs [25 : 21] 和 imm [15 : 0] 扩展后运算，或是 imm [15 : 0] 左移运算，默认将rs [25 : 21] 接到A端口，于是设置**BSel**信号对接入B端口的数据进行选择



- GRF、ALU 和 DM 的连接

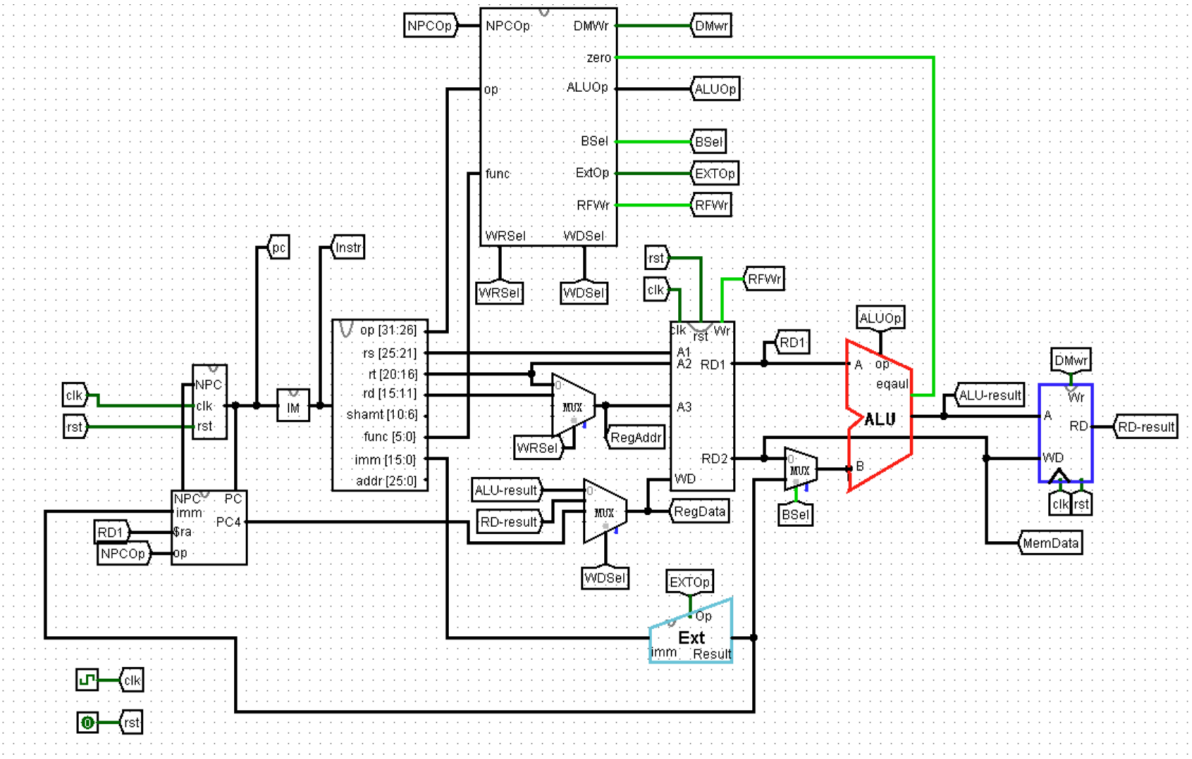
  加载和存储操作

  首先设置**DMWr**信号选择是否执行存储操作

  存储到内存的总是 rt [20 : 16] 寄存器中的数据，因此将 $GRF.RD2$ 端口与 $DM.WD$ 端口相连；

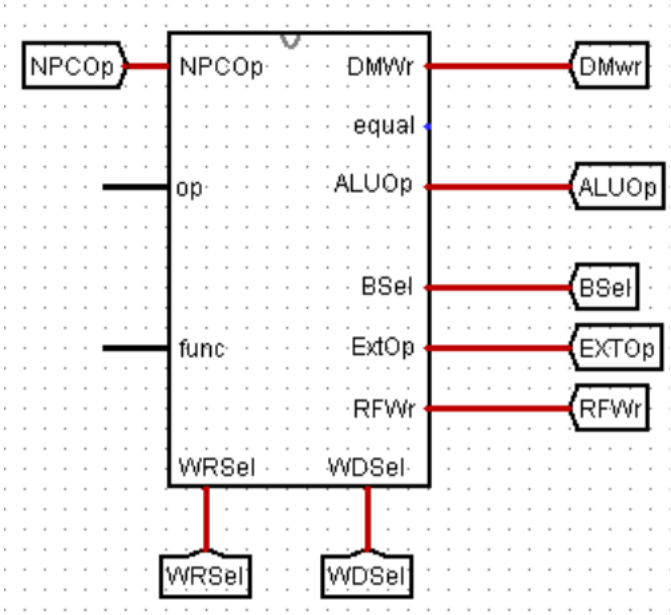
  待操作的内存地址由 ALU 计算得到，因此将 ALU输出接到 $DM.A$ 端口

整个顶层电路如下：

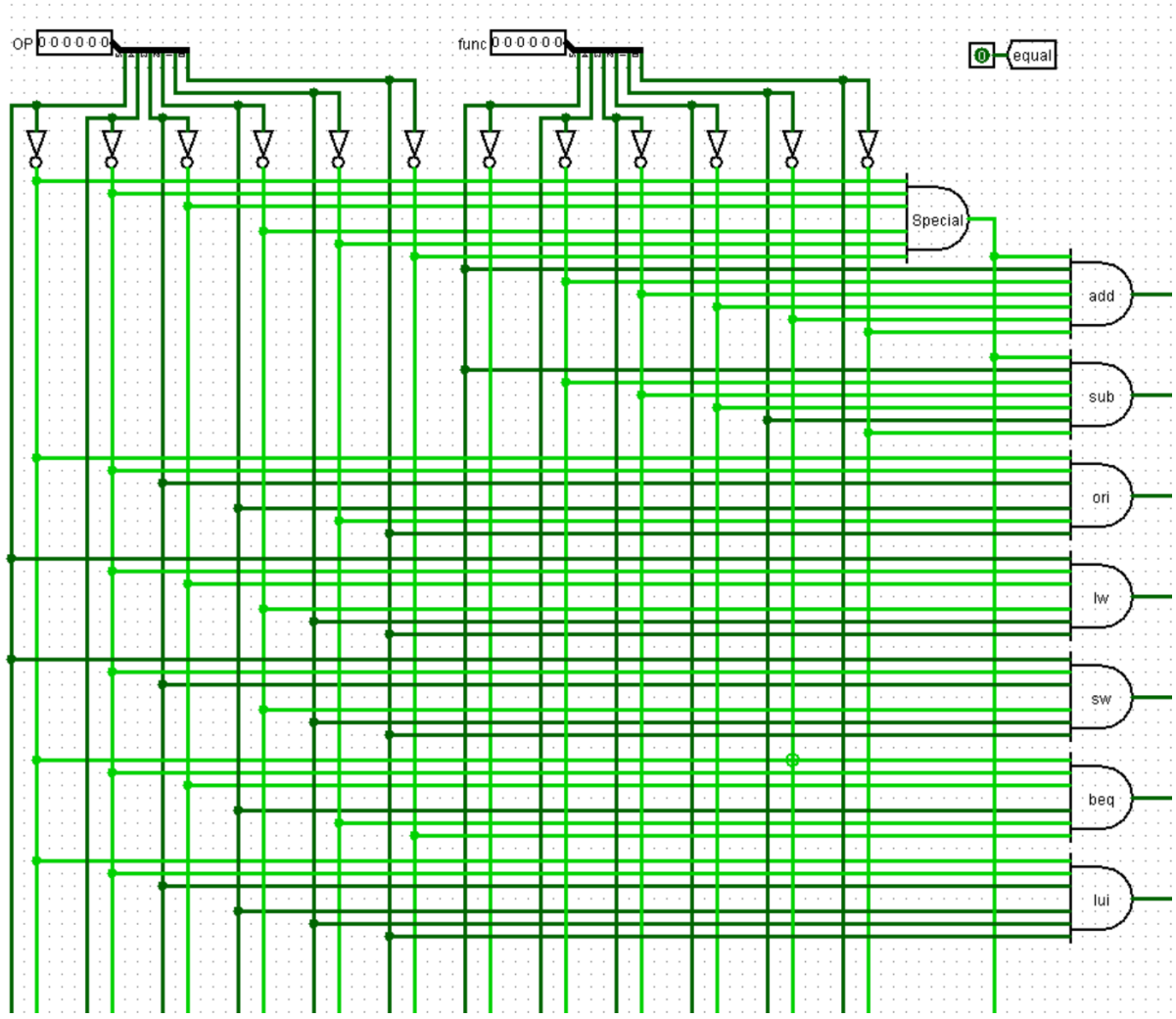


(八) Controller

输入端口名称	I/O	位宽
OP	I	6
func	I	6
equal	I	1

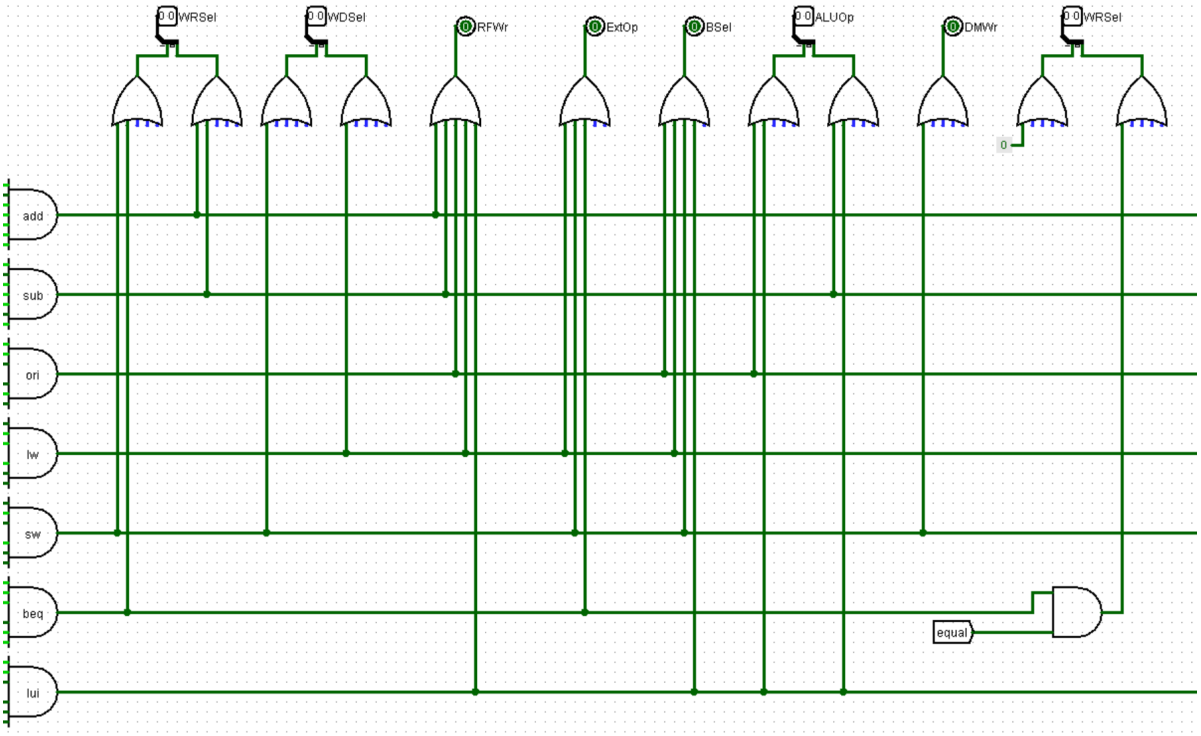


首先通过**与逻辑**确定指令类型



根据真值表完成**或逻辑**

指令	WRSel	WDSel	RFWr	ExtOp	BSel	ALUOp	DMWr	NPCOp
add	01	00	1	X	0	00	0	00
sub	01	00	1	X	0	01	0	00
ori	00	00	1	0	1	10	0	00
lw	00	01	1	1	1	00	0	00
sw	10	XX	0	1	1	00	1	00
beq	10	XX	0	X	0	XX	0	equal=0 : 00 equal=1 : 01
lui	00	00	1	0	1	11	0	00


WRSel
WDSel
RFWr
ExtOp
BSel
ALUOp
DMWr
WRSel
0
add
sub
ori
lw
sw
beq
lui
equal

（二）测试方案

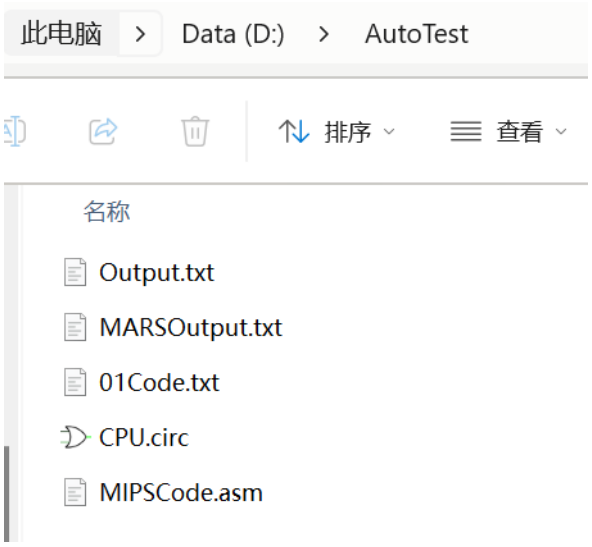
本人用python写了自动化测评程序，自动生成测试数据，将自己电路运行结果与Mars运行结果对拍

使用说明

评测机将自动生成测试数据，评测规则与cscore平台相同

你需要进行如下操作：

- 请在D盘下建立一个新目录，命名为AutoTest
- 将你的.circ文件置于其中，命名为CPU.circ
- 点击运行AutoTest.exe，评测结果将会显示在控制台上，同时在"D:\AutoTest"中生成此次测试的数据点、你的输出和正确输出



如果你想保存本次测试数据，请将这几个文件复制到其他文件夹中。本目录中的数据，在下次评测时将会被新的数据覆盖

工具原理

可划分为三个部分

- 生成测试数据

  利用随机数生成指令，规模可在源码中调整

  保证了beq指令不会造成死循环

  保证了sw、lw指令中，计算得到的地址是字对齐的

  其他各种细节不再详述，总归保证了生成的MIPS代码可以在Mars上正常运行，不会产生数据溢出、越界等错误
- 运行电路和Mars_CO_v0.5.0，得到并格式化二者输出
- 比较输出，返回测评结果

部分核心代码

```
#-------------------------------------------------------------------------------------------
# 格式化Logisim输出
def Format(outp):
    OutLines=outp.readlines()
    outp.seek(0)                        # 回到文件开头，准备清空文件
    outp.truncate()                     # 清空文件内容
    for line in OutLines:
        line=re.sub(r"\s+","",line)     # 去除空白字符
        if(line[64]=='1'):
            if(line[65:70]=="00000"):
                continue
            outp.write("Instr ["+"{:08x}".format(int(line[0:32],2))+"]\t")
            outp.write("@"+"{:08x}".format(int(line[32:64],2))+": ")
            outp.write("$"+"{:d}".format(int(line[65:70],2))+" <= "+"
{:08x}".format(int(line[70:102],2))+"\n")
        if(line[102]=='1'):
```

```
            outp.write("Instr ["+"{:08x}".format(int(line[0:32],2))+"]\t")
            outp.write("@"+"{:08x}".format(int(line[32:64],2))+": ")
            outp.write("*"+"{:08x}".format(int(line[103:135],2))+" <= "+"
{:08x}".format(int(line[135:167],2))+"\n")
###-----------------------------------------------------------------------------------------
# 切换工作路径至当前文件所在目录
current_path = os.path.abspath(__file__)     # 获取当前文件的绝对路径
new_path = os.path.dirname(current_path)     # 获取当前文件所在目录的路径
os.chdir(new_path)                           # 切换工作路径至当前文件所在目录

# 载入数据文件
content = open("test.txt").read()
file="D:\\AutoTest\\CPU.circ"
Circ = open(file, encoding="utf-8").read()
Circ = re.sub(r"addr/data: 12 32([\s\S]*?)</a>", "addr/data: 12 32\n" + content + "</a>", Circ)
with open(file, "w", encoding="utf-8") as file:
    file.write(Circ)

# 运行logisim，并格式化输出
os.system("java -jar logisim-generic-2.7.1.jar D:\\AutoTest\\CPU.circ -tty table > Output.txt")
Output = open("Output.txt","r+")
Format(Output)

# 运行MARS，并格式化输出
os.system("java -jar Mars_CO_v0.5.0.jar test.asm 250 mc CompactLargeText coL1 > MARSOutput.txt")
MarsOutput = open("MARSOutput.txt","r+")
Format_Mars(MarsOutput)

# 比较输出
Compare(Output, MarsOutput)
###-----------------------------------------------------------------------------------------
```

参考资料

Mars_CO_v0.5.0是从https://github.com/Toby-Shi-cloud/Mars-with-BUAA-CO-extension中获取的，感谢伟大的助教😙

（三）思考题

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。

> 状态存储：IM、GRF、DM
>
> 状态转移：NPC、ALU

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。

> 合理。
>
> 程序段是不可更改的，所以放入ROM中，只读
>
> 数据段既可读也可写，所以放入RAM中
>
> GRF本身就是寄存器堆，用寄存器来实现显然合理

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。

> 我将分位器作了封装，并且把IFU模块细分为了PC、NPC、IM三个模块，具体设计思路参见“（一）设计草稿”

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？

> nop指令不作任何处理，即写寄存器信号和写内存信号全部置零，而默认情况下即如此，所以无需将其加入控制信号表

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。

> 指令类型覆盖率达到75%，缺少对sub和nop的测试，对GRF的测试不够全面
>
> 单一指令：
>
> - ori：测试了与0和较小的非零数的运算，但ori对立即数进行无符号扩展，缺少对最高位为1的立即数的测试

- lui :　　已充分测试
- add:　　测试了正正、正负、负负，缺少0的测试
- lw、sw：缺少偏移量为负数的测试
- beq:　　测试了相等和不相等两种情况，但缺少偏移量为负数的测试