设计草稿

注意事项

1. 仔细阅读RTL语言
2. 搭电路时不要影响到其他指令的实现
3. 注意各种器件的参数，如有符号/无符号，零扩展/符号扩展，计数器的溢出时操作等
4. 仔细读报错信息
5. 除以不常用器件：奇偶校验，优先编码，求补器，逐位加法器，位查找器

设计单周期cpu前的准备

1. 整体浏览了一遍p3教程 **模块化、层次化**
2. 复习了Logisim中各部件的用法及功能
3. 重新看了一遍 Pre **MIPS指令集及汇编语言**一节

核心思想

本部分内容主要来自于阅读肖利民老师的PPT以及阅读学长往年的博客

核心思想：CPU的功能是控制指令执行，过程是**取指、取数、执行。**

> ***注意一定要仔细读RTL语言！！！***

MIPS模型机指令集

模型机指令编码

R 类型格式	OP (31 ~ 26)	Rs (25 ~ 21)	Rt (20 ~ 16)	Rd (15 ~ 11)	Shamt (10 ~ 6)	Funct (5 ~ 0)
add rd, rs, rt	000000	rs	rt	rd	XXXXX	100000
sub rd, rs, rt	000000	rs	rt	rd	XXXXX	100010
and rd, rs, rt	000000	rs	rt	rd	XXXXX	100100
or rd, rs, rt	000000	rs	rt	rd	XXXXX	100101

I 类型格式	OP (31 ~ 26)	Rs (25 ~ 21)	Rt (20 ~ 16)	16 bits immediate or address (15 ~ 0)
lw rt, rs, imm16	100011	rs	rt	imm16
sw rt, rs, imm16	101011	rs	rt	imm16
beq rs, rt, imm16	000100	rs	rt	imm16

J 类型格式	OP (31 ~ 26)	26 bits address
j target	000010	target

- 对于R型指令，处理流程为：

  从IM中读指令——送Controller中译码——从GRF中取数——送ALU运算——送GRF存结果——PC加4

- 对于I型指令，处理流程为：
  从IM中读指令——送Controller中译码——从GRF中取数&在EXT中扩展立即数——（送ALU运算）（从DM中读数）——送GRF存结果|送DM存结果|计算nPC——PC+4|PC变到指定值
- 对于J型指令，处理流程为：
  从IM中读指令——送Controller中译码——从GRF中取数|将立即数扩展到32位——改变NextPC的值

子电路具体实现方法

目前共有IFU、GRF、ALU、DM、EXT、Controller、nextPC七个单元

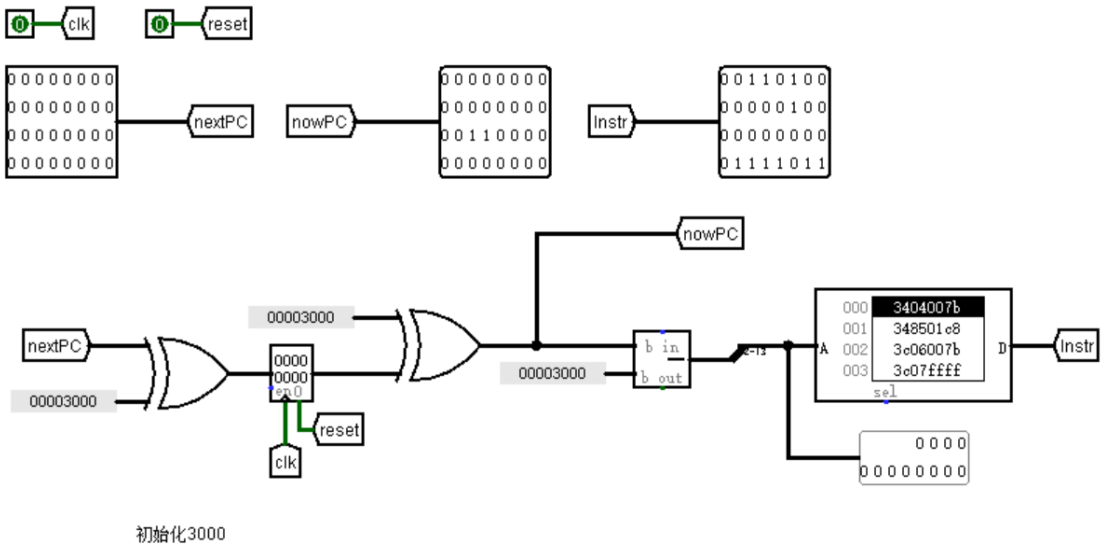
IFU（取指令单元）

*指令的地址范围为0x0000_3000~0x0000_6FFF,ROM容量为4096*32bit*

- 功能：1. 根据PC值在IM（存储code的ROM，预先从mips中导出，在第一行加上v2.0 raw）中取出要执行的指令; 2. 具有异步复位功能；
- 端口：

端口名称	方向	描述
clk	input	时钟信号
reset	input	异步复位信号，1时将PC复位到0x0000_30000
nextPC[31:0]	input	下一周期的PC值
nowPC[31:0]	output	本周期执行指令的PC值
Instr[31:0]	output	本周期执行指令的二进制编码

- 电路图：



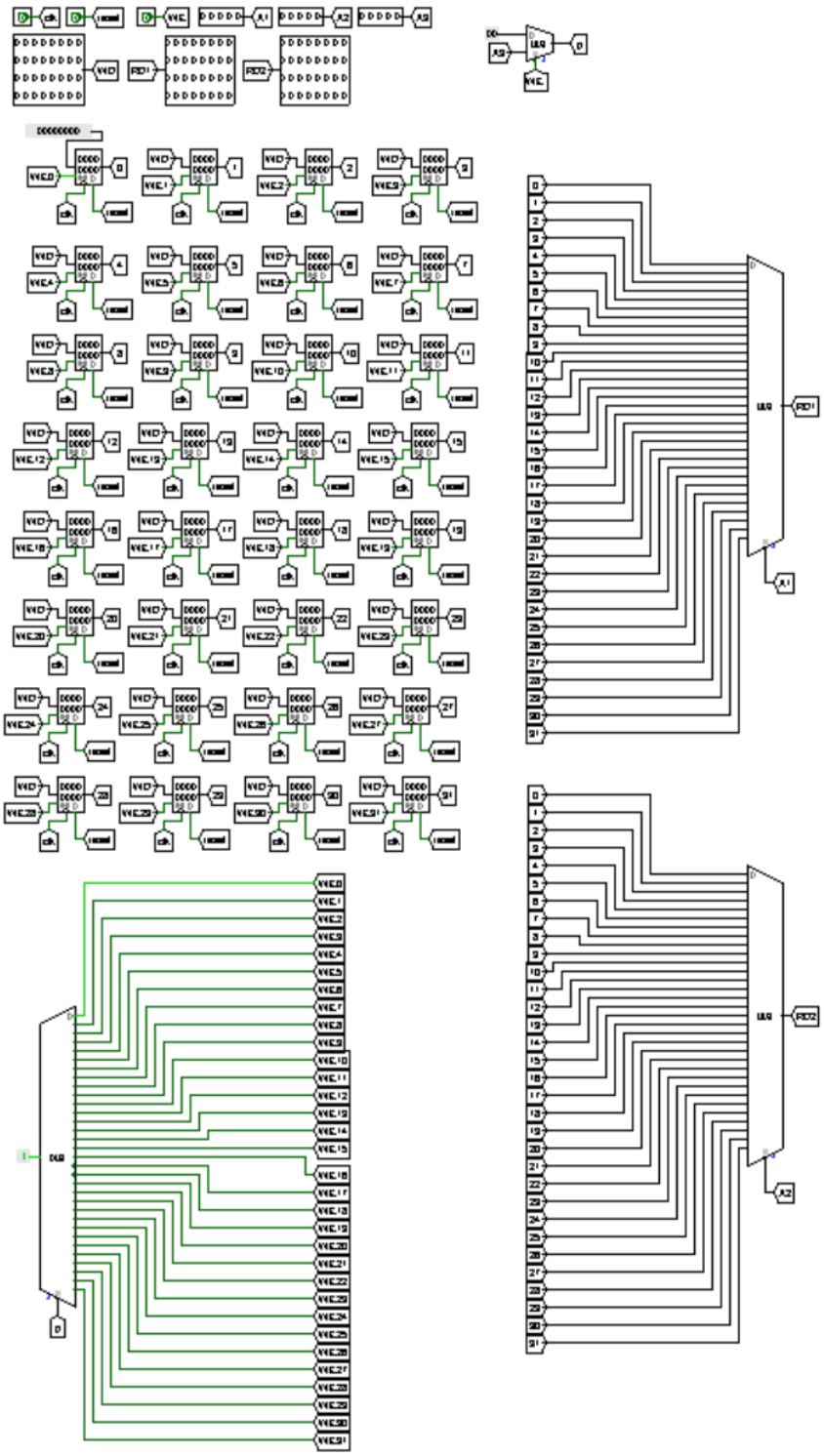
初始化3000

GRF（通用寄存器组）

- 功能：
  - 具有异步复位功能；
  - 可以同时读两个寄存器内存储的值；
  - 可以向一个寄存器写入值；
- 端口（与P0_L0_GRF基本一致）：

信号名	方向	描述
clk	input	时钟信号
reset	input	异步复位信号，1时将32个寄存器中的值全部清零
WE	input	写使能信号，1时可以向GRF中写入数据，0时不可
WD[31:0]	input	需要写入的数据
A1[4:0]	input	地址输入信号，对应R1
A2[4:0]	input	地址输入信号，对应R2
A3[4:0]	input	地址输入信号，将对应的寄存器写入的目标
RD1[31:0]	output	输出A1指定的寄存器中的值
RD2[31:0]	output	输出A2指定的寄存器中的值

- 电路图：



ALU（算术逻辑单元）

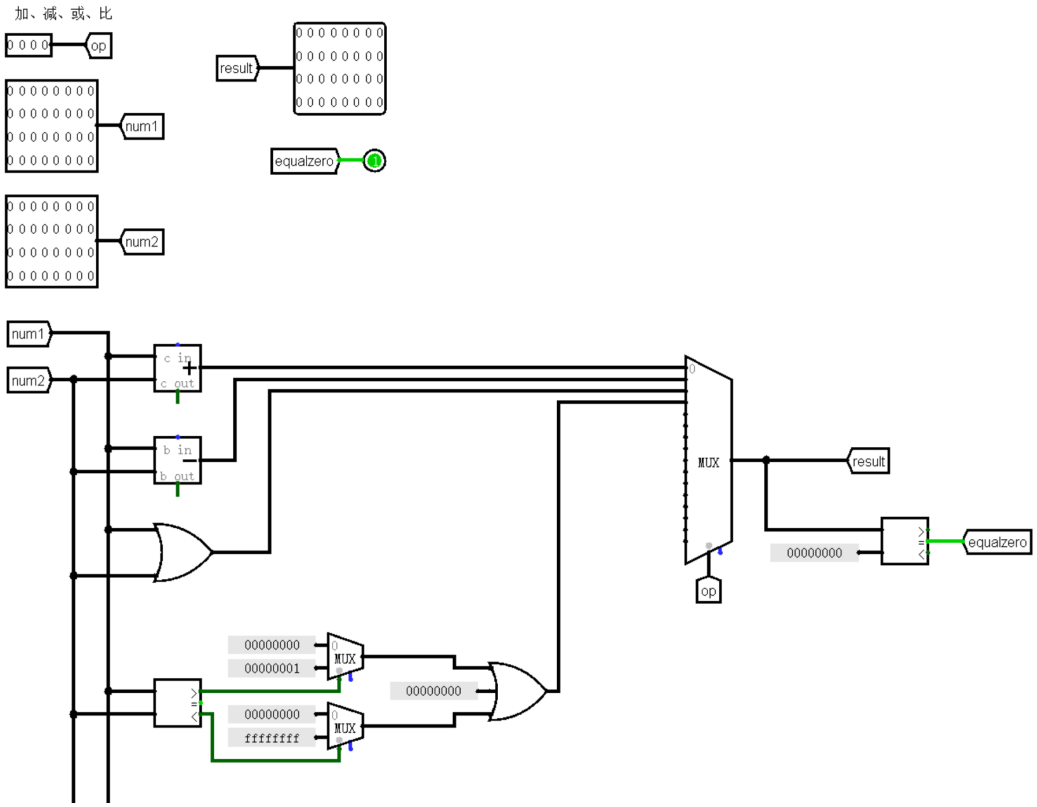
- 功能： 提供32位加、减、或运算及大小比较功能
- 端口：

信号名	方向	描述
op[3:0]	input	操作选择信息（最多支持8种操作）
num1[31:0]	input	操作数1
num2[31:0]	input	操作数2

信号名	方向	描述
result[31:0]	output	结果数
equalzero	output	判断结果是否为零，1时为0

op	描述
0000	不考虑溢出的无符号加
0001	不考虑溢出的无符号减
0010	求两个操作数的按位或
0011	判断大小，0:相等　1:num1>num2　-1:num1<num2

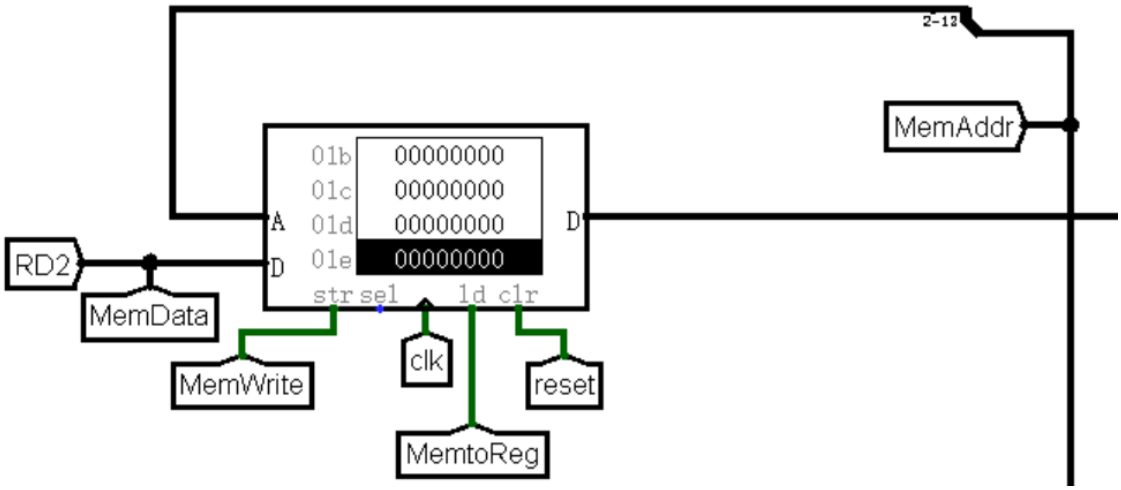
- 电路图：



DM(数据存储器)

- *介绍：* *在Logisim中，DM可用一个RAM实现，容量为3072*bit，具有异步复位功能，RAM的Data Interface属性为Separate load and store ports,地址范围为0x0000_0000~0x0000_2FFFF。*

- 电路图：



EXT(扩展器)

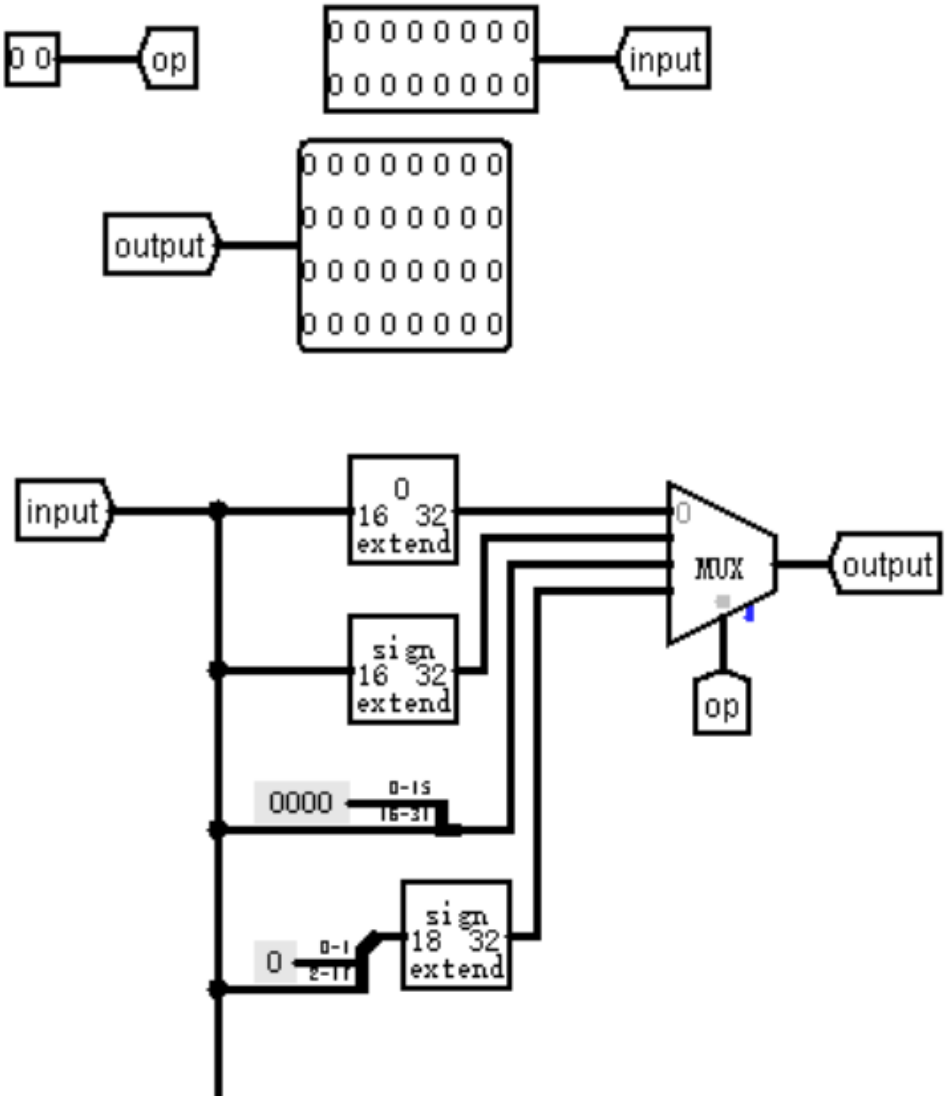
- 功能： 实现16位立即数的0扩展,移到高位，符号扩展，补两位0再符号扩展，得到32位数
- 端口：

信号名	方向	描述
op[1:0]	input	操作选择信号
input[15:0]	input	16位操作数
output[31:0]	output	32为结果数

op	描述
00	0扩展
01	符号扩展
10	将16位立即数移到高位，后16位补0
11	后补两个0，然后进行符号扩展

- 电路图：

00:0扩展　01:符号扩展　10：移到高位　11：后补两个0，然后符号扩展



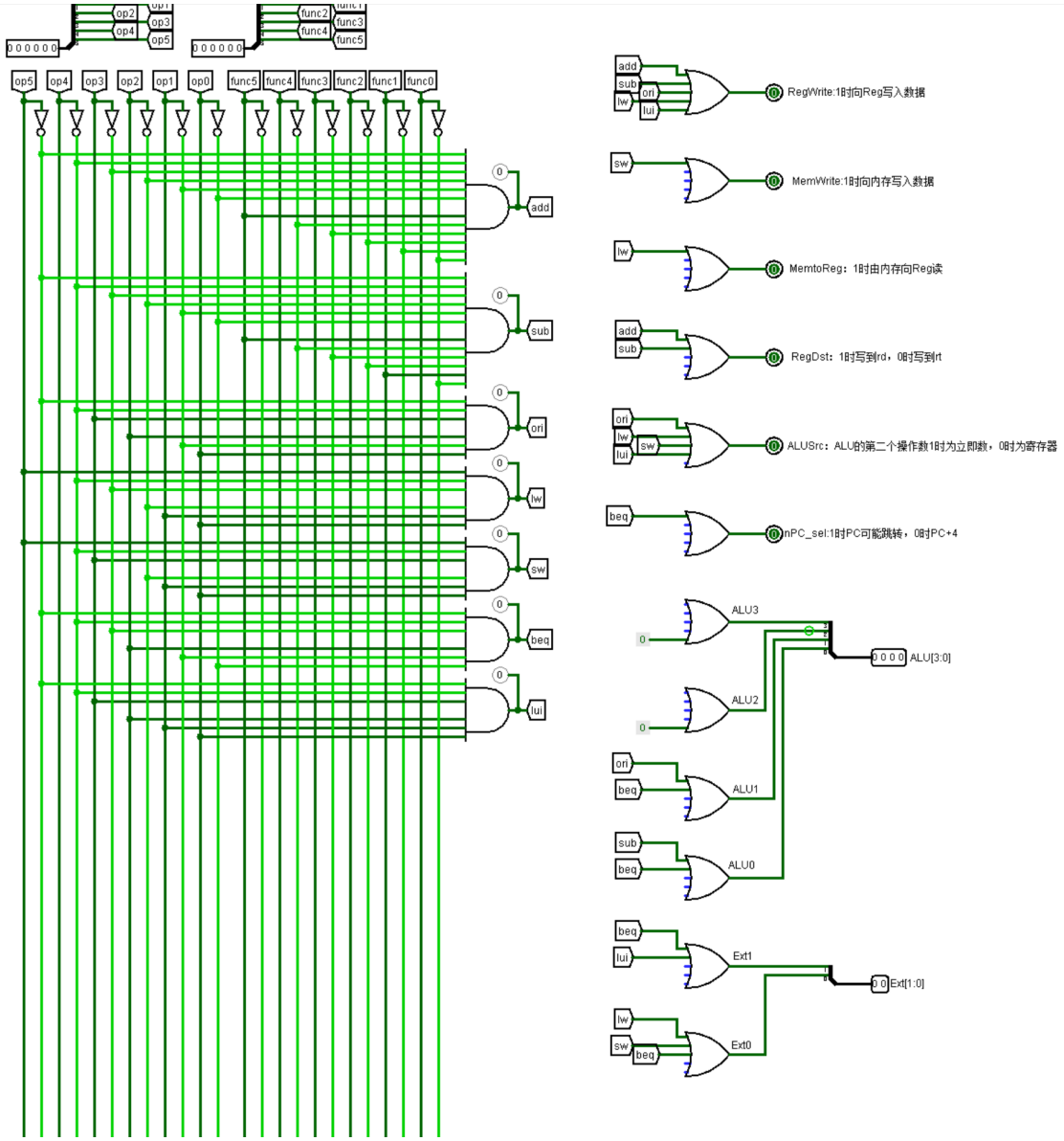
Controller(控制器)

- 功能：通过识别Opcode和Funct两段二进制数据来确定各种控制信号的高低电平。
- 端口：Dst指destination，Src指source

信号名	方向	描述
OpCode[5:0]	input	对应Instr[31:26]
Func[5:0]	input	对应Instr[5:0]
RegWrite	output	1时可以向Reg写入数据
MemWrite	output	1是可以向DM写入数据
MemtoReg	output	1时可以从DM读出数据存入Reg
RegDst	output	1时写到rd，0时写到rt
ALUSrc	output	1时ALU的第二个操作数为立即数，否则为寄存器
nPC_sel	output	1时PC可能跳转，0时PC+4

信号名	方向	描述
ALU[3:0]	output	ALU操作选择信号
Ext[1:0]	output	Ext操作选择信号

- 电路图：*左侧为and logic，右侧为or logic，并且可以扩展*



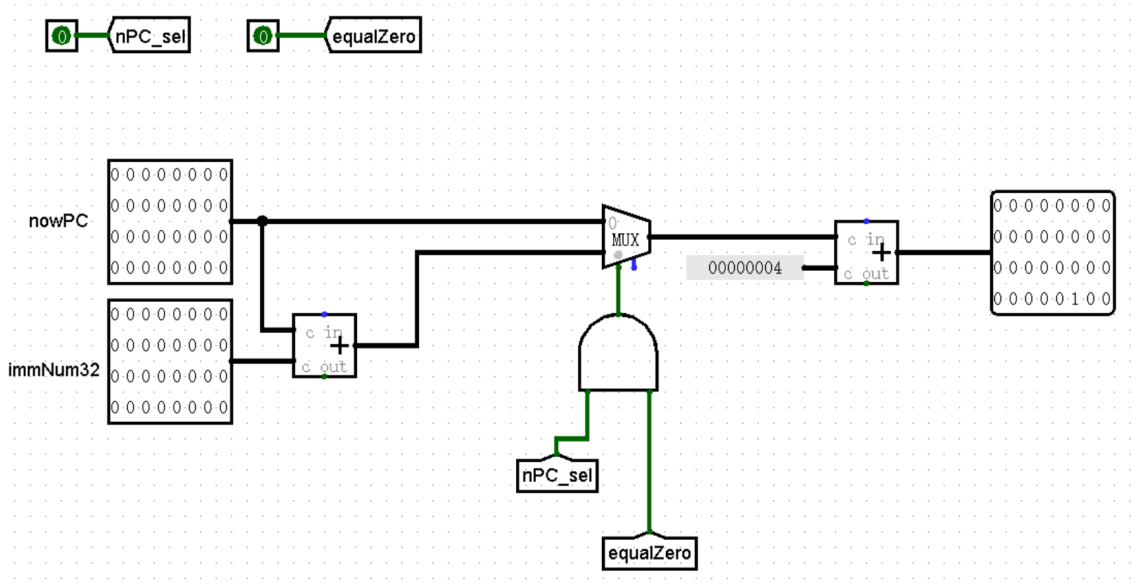
nextPC(指令地址计算单元)

- 功能：计算得出下一个要执行的指令的地址
- 端口：

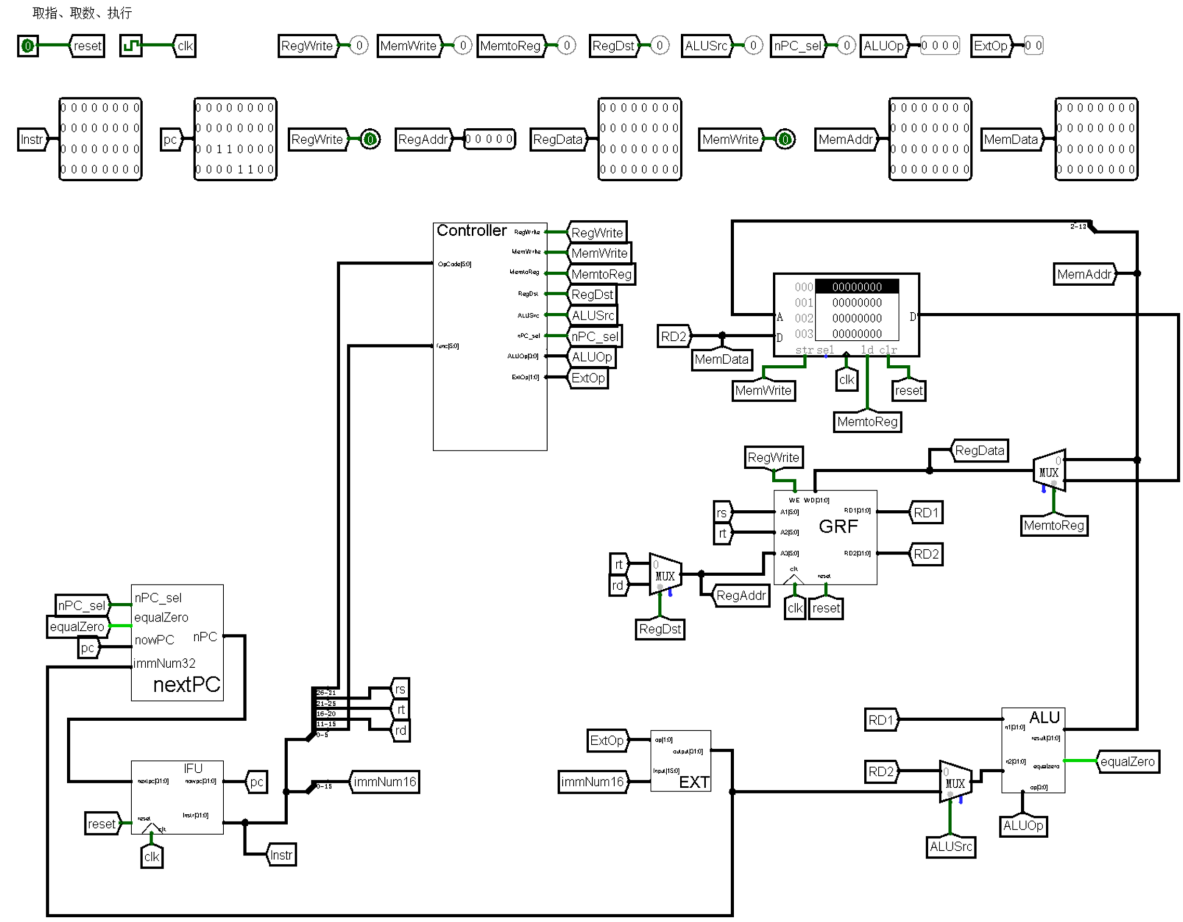
信号名	方向	描述
nPC_sel	input	nextPC选择控制信号
equalZero	input	nextPC选择控制信号
nowPC[31:0]	input	目前正在执行的指令的地址
immNum32[31:0]	input	nPC_sel与equalZero均为1时将加nowPC再加4作为nextPC

信号名	方向	描述
nextPC[31:0]	outPC	下一个要执行的指令的地址

- 电路图：



顶层设计电路



最上面两行为输入和输出以及一些关键数据的Probe;

顶层设计电路主要就是子电路+splitter+Multiplexer+Tunnel+wire的组合。

思考题

第一题 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。

- 状态存储模块： IM、DM 、GRF
- 状态转移功能：EXT、ALU、nextPC

第二题 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。

这种做法是合理的

- IM是用来存储指令的，ROM的特点是可读但不可写，而指令要求不能在运行过程中被更改，这与ROM的特点相符合。
- DM是数据存储器，RAM的特点使可读可写并且可以复位，数据存储器的数据要求在程序运行过程中可以被读出、被写入还有被复位为0，这与RAM的特点相符合。
- GRF使用Register即寄存器，寄存器是计算机可以操作的速度最快的存储单元，CPU需要频繁快速地存储数据，这与寄存器的特点相符合，另外寄存器价格昂贵，而GRF所需寄存器数量是非常小的，这种花销是可接受的。

第三题 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。

设计了nextPC模块，设计思路*子电路具体实现方法*一节~~（不过好像教程中也有提示hh）~~

第四题 事实上，实现nop空指令，我们并不要将它加入控制信号真值表，为什么？

nop空指令在执行期间，CPU实际上不做任何改变寄存器、内存的操作。不将nop加入控制信号真值表，那么控制寄存器写入、内存写入等的控制信号均会为0，这样便实现了nop的功能。

第五题 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。

- 该测试是合理的，测试了所有已经实现的指令，并且对于每个指令都举了多种情况进行测试（如正正 正负 负负等）~~（实际上在刚搓出cpu的时候，这个测试代码帮我找出了一个bug）~~
- 该测试强度还是不够高的
    - ori：应该尝试一下对$0操作
    - lui：应该尝试一下对$0操作
    - add：
        - 尝试一下三个寄存器有两个是相同的以及三个都是相同的情况，尝试一些较大的数
        - 尝试一下接近边界的数
    - sub：同add
    - sw：
        - 尝试一下offset是负数的情况
        - 尝试一下base寄存器是非0数的情况
        - 最好存的数据每byte都不为0x00
    - lw：同上 + 尝试一下0号寄存器
    - beq：应该尝试一些跳转或者不跳转并且目标在跳转指令之前或就是跳转指令的情况

- nop：没有测试nop

测试方案

基于pre提供的测试方案，考虑到自己想出的pre代码的不足之处，写出了新的测试代码。

```
.text
#lui
lui $0,0x1234
ori $t0,0xabcd
lui $t0,0xffff
#ori
ori $t1,$t0,0x1234
ori $0,$t1,0xffff
#add
ori $t2,0x1234
add $t2,$t2,$t2
add $0,$t2,$t2
ori $t3,0x0001
lui $t4,0xffff
ori $t4,$t4,0xfffe
add $t4,$t3,$t4
nop
#sub
lui $t5,0xffff
ori $t5,0xffff
ori $t6,0xabcd
sub $t6,$t5,$t6
sub $t5,$t5,$t5
nop
#sw
ori $t7,4
sw $t7,-4($t7)
sw $t0,4($0)
sw $t1,8($0)
sw $t2,12($0)
sw $t3,12($0)
sw $0,16($0)
nop
#beq
ori $s0,0x1
ori $s1,0xffff
beq $s0,$s1,b1
b2:
sub $s1,$s1,$s1
beq $0,$s1,b3
b1:
beq $0,$0,b2
b3:
ori $s2,11451
add $s3,$s2,$0
b4:
beq $s3,$s2,b4
```