使用Logism的单周期mips CPU设计文档

17373436 林昱同

# 一、模块规格

## 1、IFU(取指令单元)

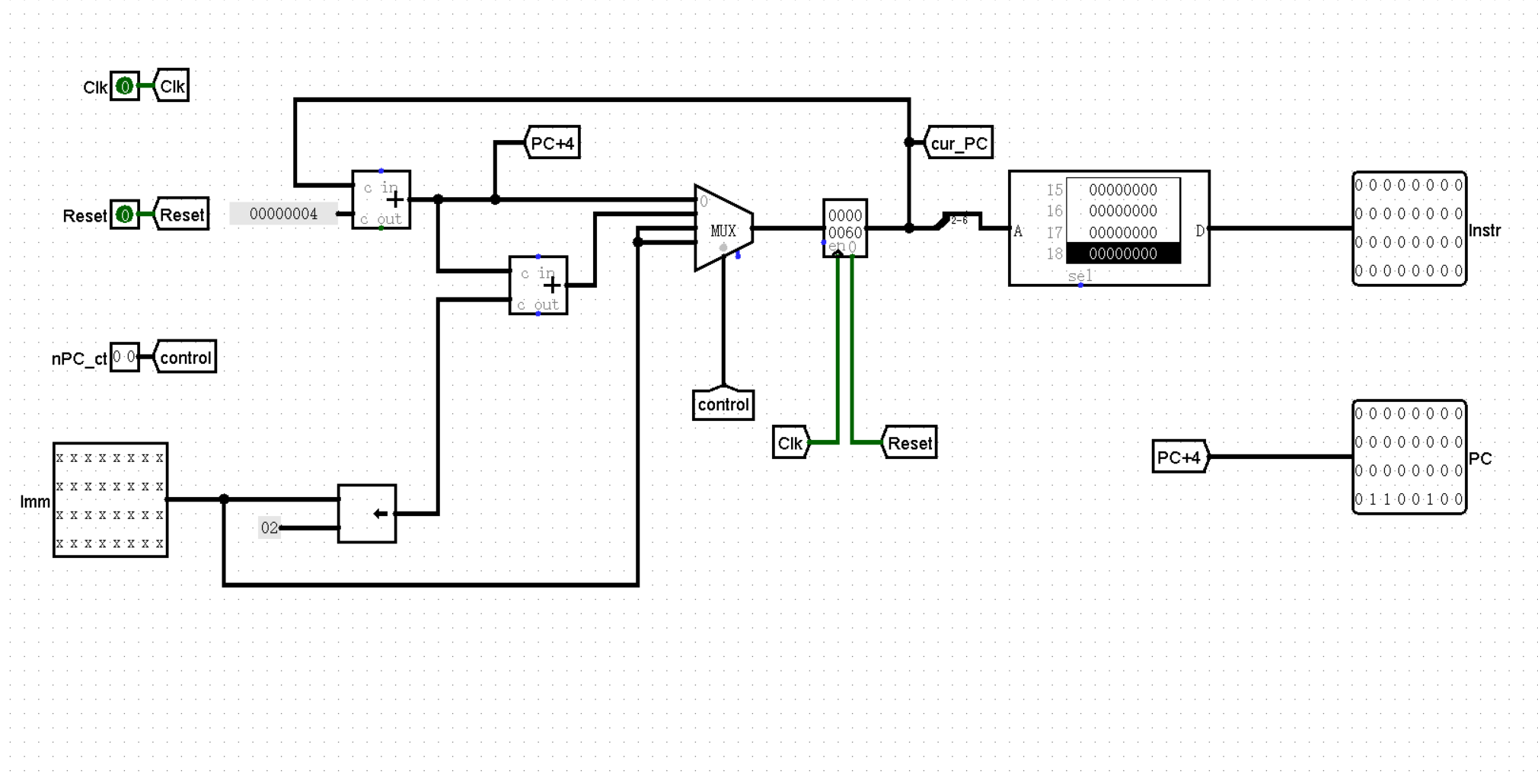
### 端口定义：

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Clk | Input | 1 | 时钟信号 |
| Reset | Input | 1 | 复位 |
| nPC\_ctrl | Input | [1:0] | 是否为分支/跳转指令 |
| Imm | Input | [31:0] | 分支/跳转指令的数 |
| Instr | Output | [31:0] | 当前的指令 |
| PC | Output | [31:0] | 当前PC值 |

### 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 复位 | 当reset为1时，PC变为0 |
| 2 | 下一条指令 | 当Clk上升沿来临时  当nPC\_ctrl为00时，PC<-PC+4  当nPC\_ctrl为01时，PC<-PC+4+{Imm,2’b00}  当nPC\_ctrl为1X时，PC<-Imm |

### 电路图



## 2、GRF单元（通用寄存器单元）

### 端口定义

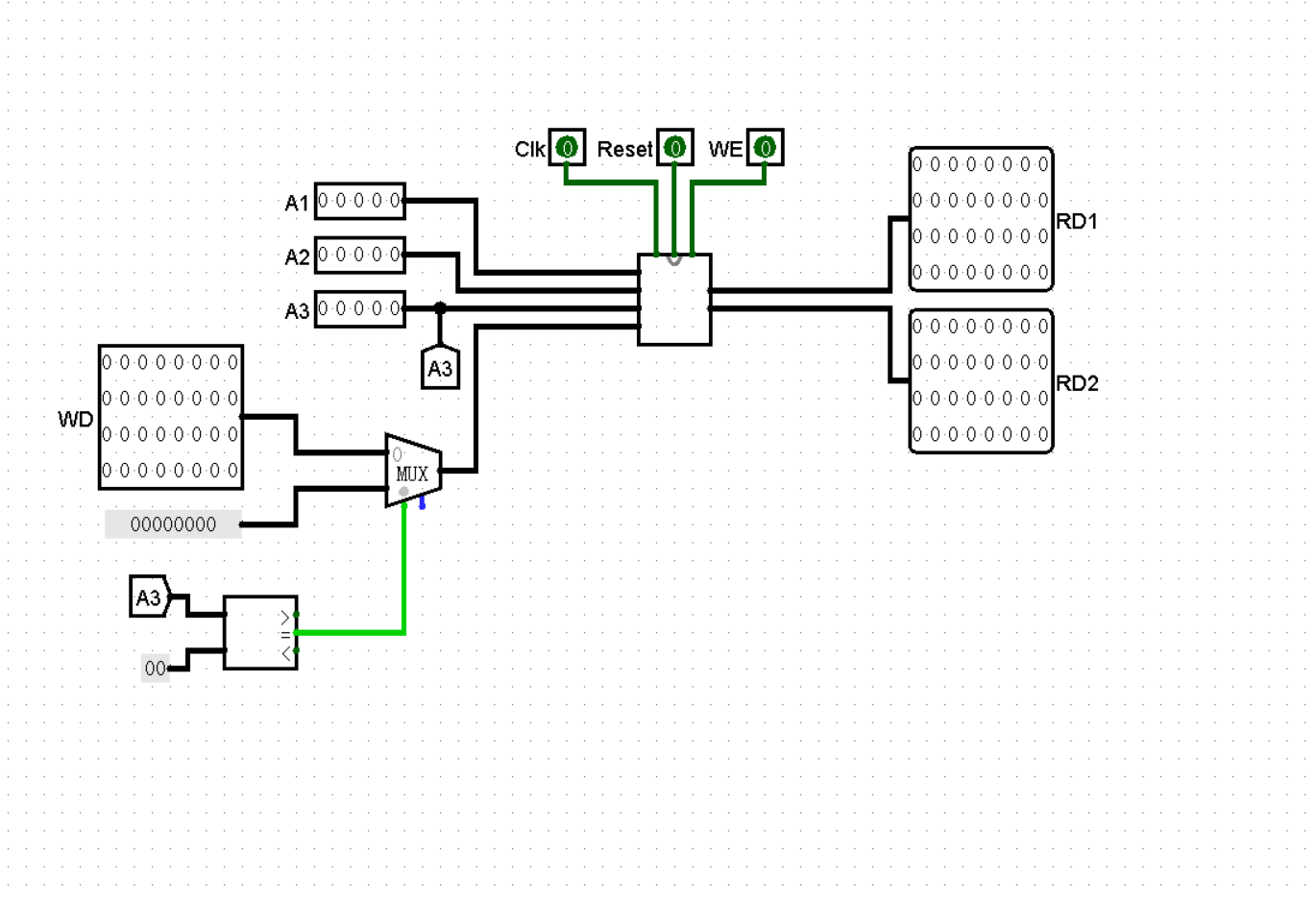
|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| A1 | Input | [4:0] | 读寄存器编号1 |
| A2 | Input | [4:0] | 读寄存器标号2 |
| A3 | Input | [4:0] | 写寄存器编号 |
| WD | Input | [31:0] | 写入数据 |
| Clk | Input | 1 | 时钟信号 |
| Reset | Input | 1 | 复位信号 |
| WE | Input | 1 | 写入使能 |
| RD1 | Output | [31:0] | 寄存器值1 |
| RD2 | Output | [31:0] | 寄存器值2 |

### 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 复位 | 当reset为1时，所有寄存器值均变为0 |
| 2 | 读取值 | RD1 RD2始终为A1和A2编号的寄存器的值 |
| 3 | 写入 | 当WE为1时，向A3号寄存器写入WD |

### 电路描述

电路使用倍增的方法搭出，因此完全不宜放于此，这里放置顶层电路。



## 3、ALU（算术逻辑单元）

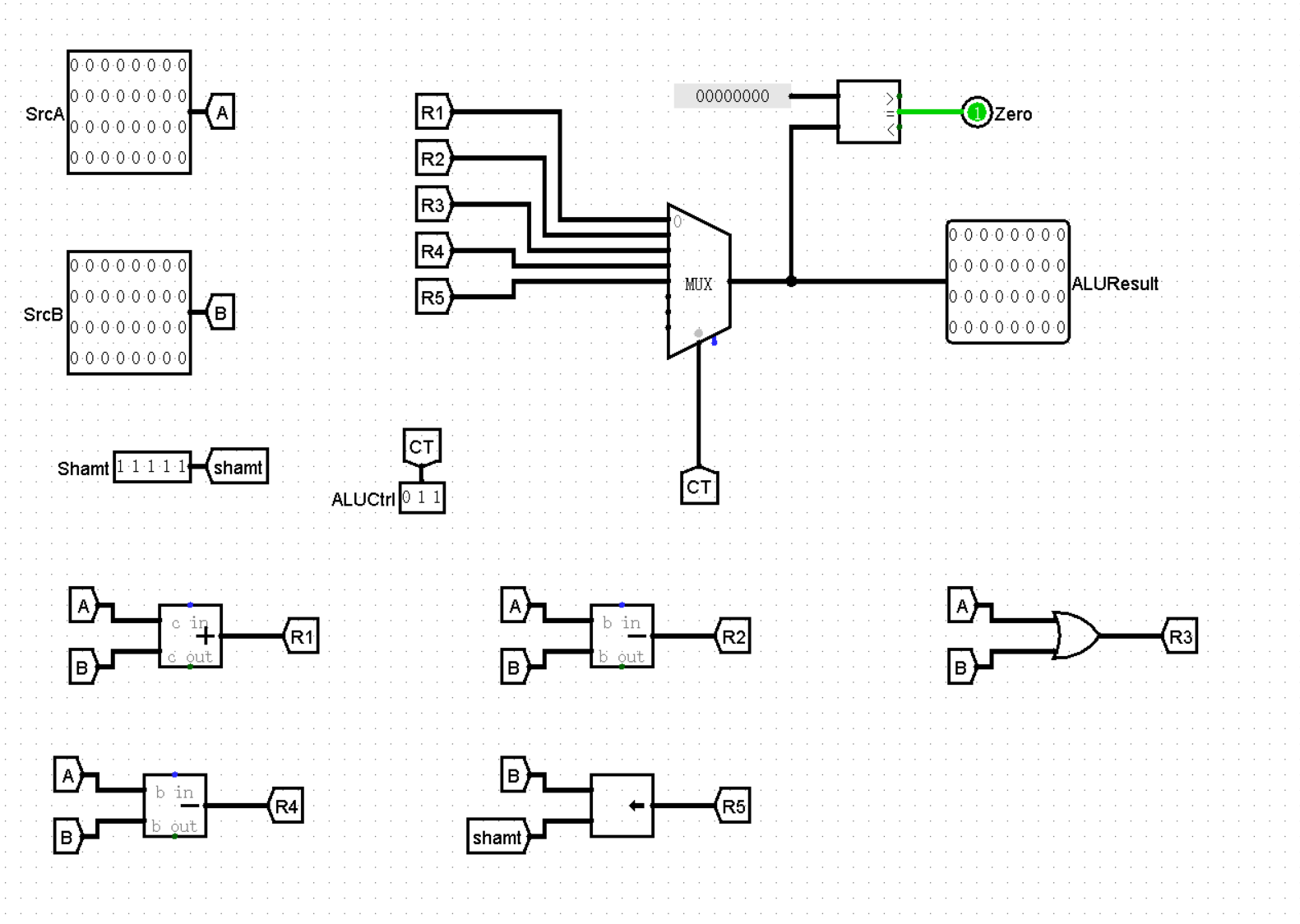
### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| SrcA | Input | [31:0] | 数据A |
| SrcB | Input | [31:0] | 数据B |
| ALUCtrl | Input | [2:0] | ALU功能控制信号 |
| Shamt | Input | [4:0] | 移位控制 |
| Zero | Output | 1 | 运算结果是否为零 |
| ALUResult | Output | [31:0] | 运算结果 |

### 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 加 | ALUCtrl=2’b000，ALUResult=SrcA+SrcB |
| 2 | 减 | ALUCtrl=2’b001，ALUResult=SrcA-SrcB |
| 3 | 或 | ALUCtrl=2’b010，ALUResult=SrcA|SrcB |
| 4 | 比较 | ALUCtrl=2’b010，Zero=SrcA==SrcB |
| 5 | 逻辑左移 | ALUCtrl=2’b100，ALUResult=SrcB<<Shamt |
|  |  |  |
|  |  |  |
|  |  |  |

### 电路实现



## 4、DM（数据储存器）

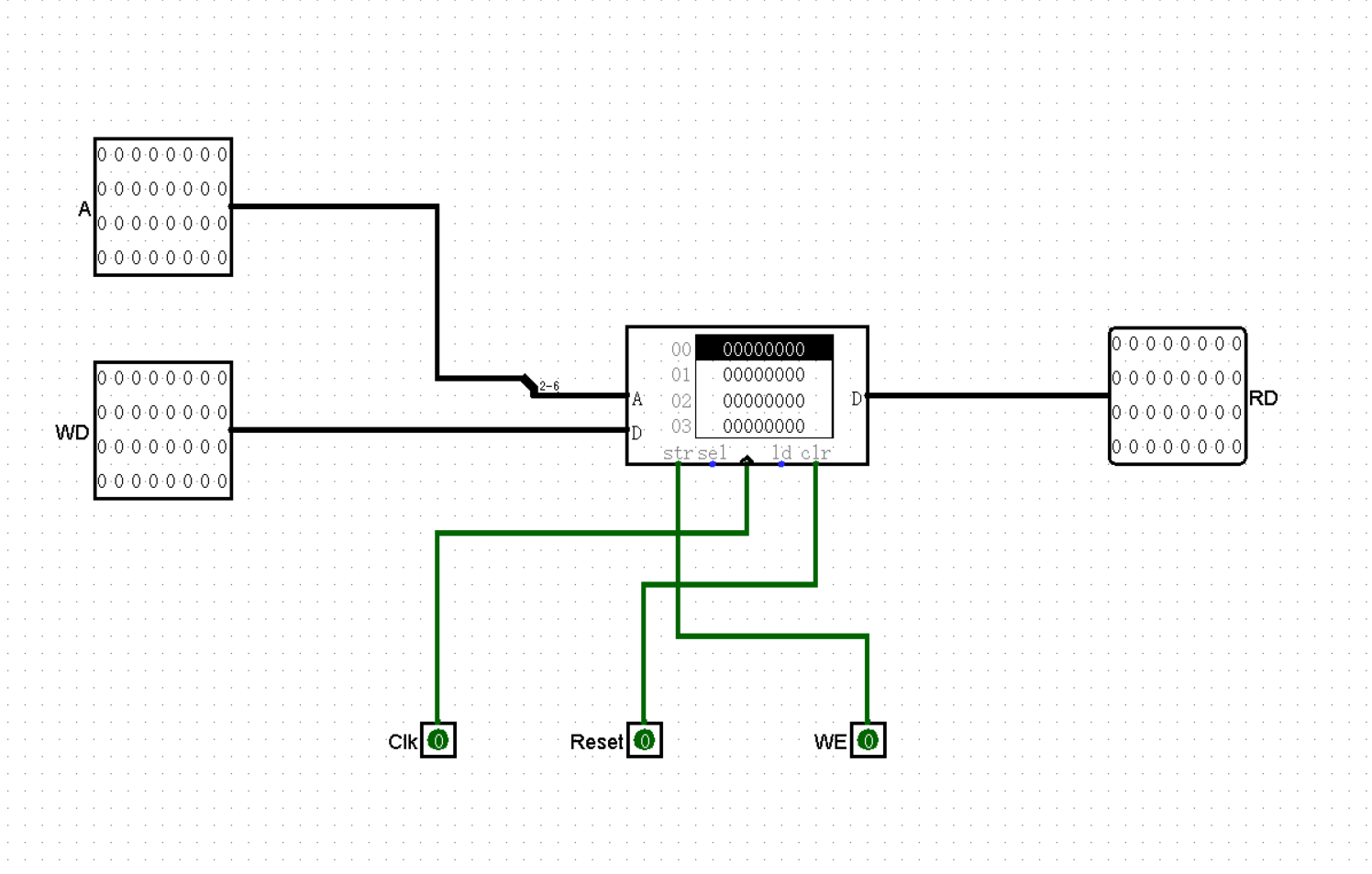
### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| A | Input | [31:0] | 地址，只有[4:0]有意义 |
| WD | Input | [31:0] | 写入数据 |
| Clk | Input | 1 | 时钟信号 |
| WE | Input | 1 | 写入使能 |
| Reset | Input | 1 | 初始化信号 |
| RD | Output | [31:0] | 读取数据 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 写入 | 当时钟上升沿来临时，如果Reset为0且WE为1，则再A的位置写入WD |
| 2 | 读取 | RD始终为地址为A的数据的值 |
| 3 | 清空 | Reset为1时，所有数据清0 |
|  |  |  |

### 电路图



## 5、EXT（拓展器）

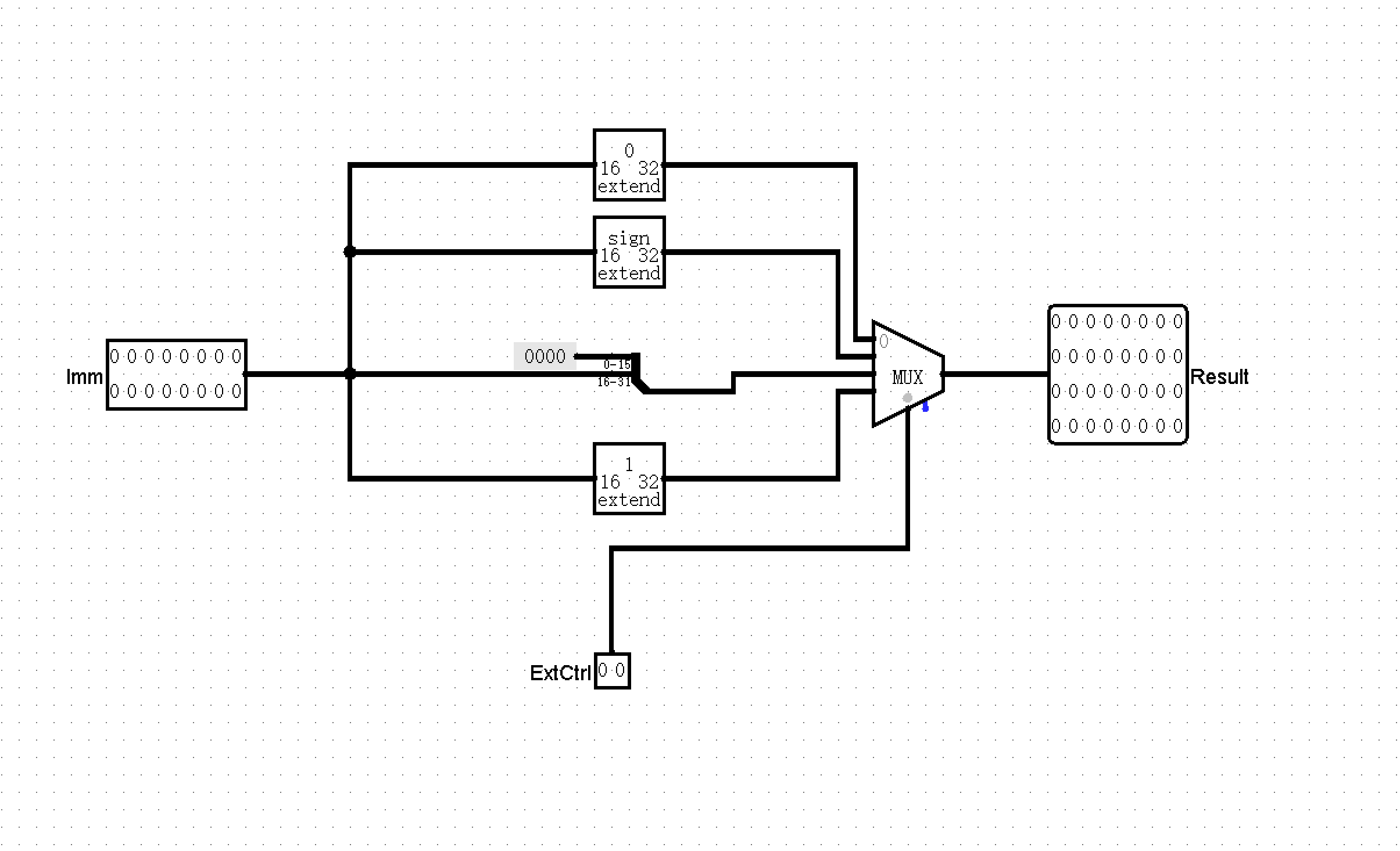
### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Imm | Input | [15:0] | 输入立即数 |
| ExtCtrl | Input | [1:0] | Extender控制信号 |
| Result | Output | [31:0] | 拓展结果 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 0拓展 | Result={{16{0}},Imm} |
| 2 | 符号拓展 | Result={{16{Imm[15]}},Imm} |
| 3 | 加载到高位 | Result={Imm, {16{0}}} |
| 4 | 1拓展 | Result={{16{1}},Imm} |

### 电路图



## 6、BC(Branch \_Control分支控制)

分支的控制信号既关乎数据流，也关乎控制信号，因此在下面定义控制信号之前定义描述。

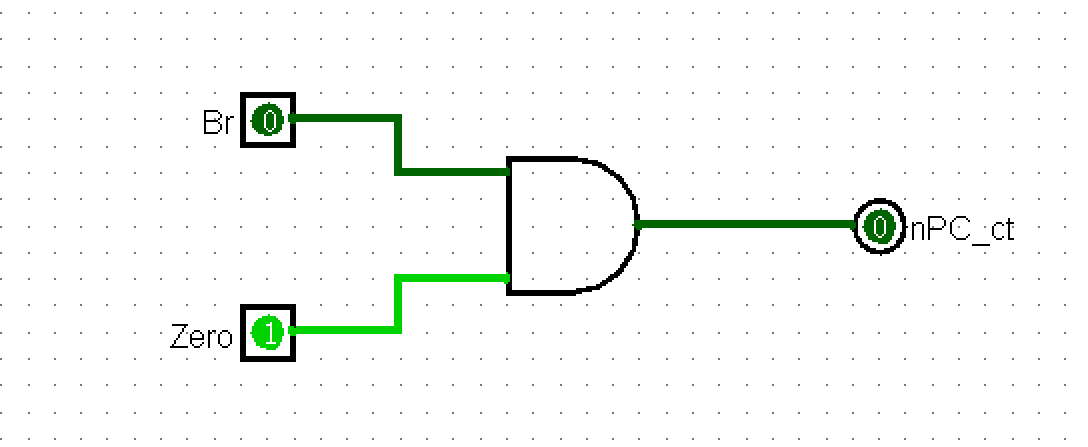
由于只需要一个beq，所以这里直接使用一个与门即可。

### 接口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位宽 | 功能简述 |
| Br | Input | 1 | 是否为分支指令 |
| Zero | Input | 1 | 结果是否为0 |
| nPC\_Ctrl | Output | 1 | 是否分支跳转 |

## 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名 | 功能描述 |
| 1 | 分支判断 | nPC\_Ctrl=Br&Zero |



# 二、控制信号

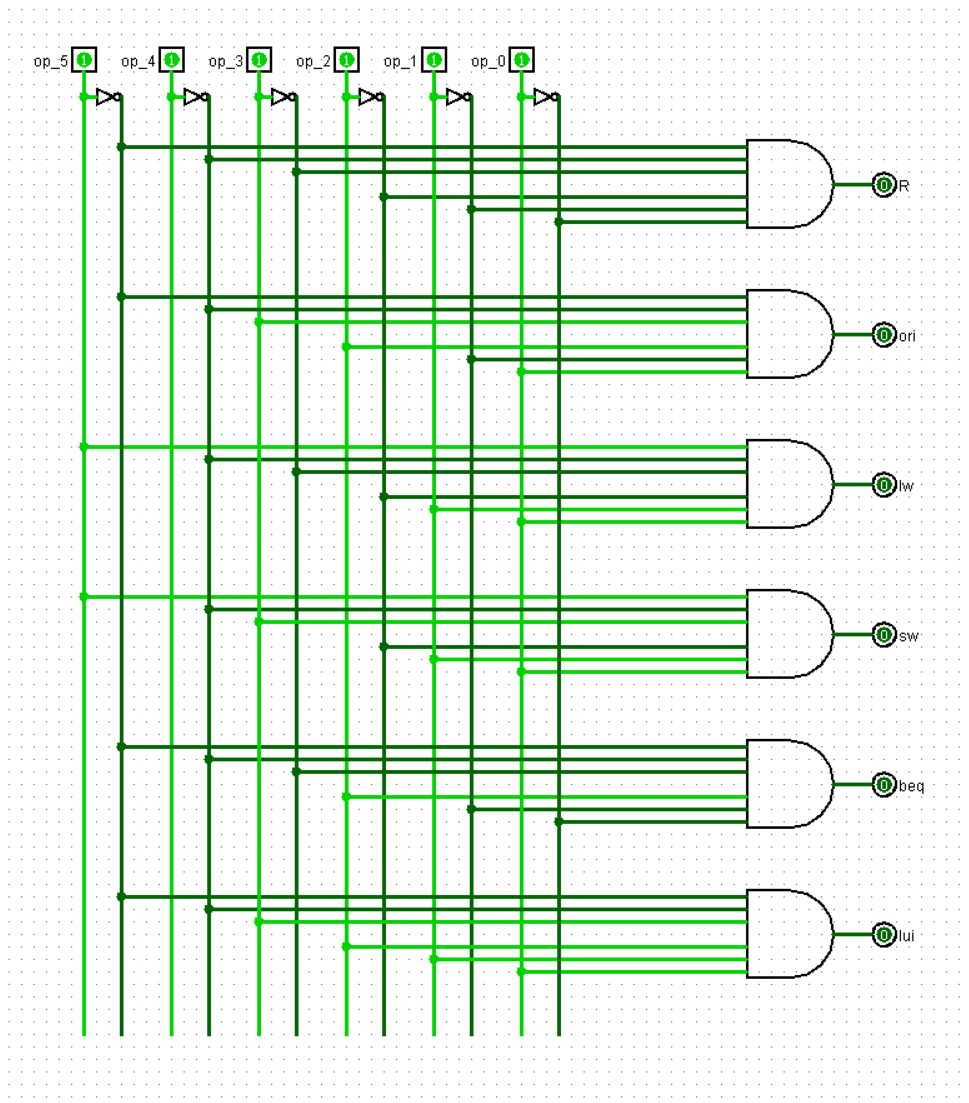
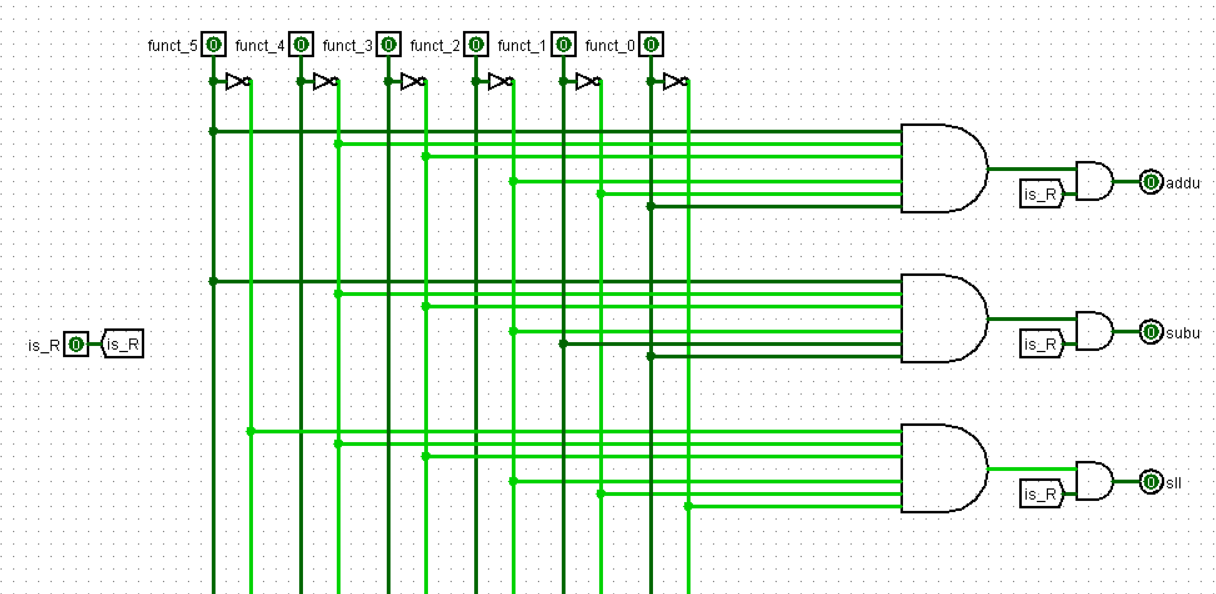
## 1、指令编码

### 各个指令的opcode和funct

|  |  |  |
| --- | --- | --- |
|  | opcode[5:0] | funct[5:0] |
| ADDU | 000000 | 100001 |
| SUBU | 000000 | 100011 |
| ORI | 001101 | \ |
| LW | 100011 | \ |
| SW | 101011 | \ |
| BEQ | 000100 | \ |
| LUI | 001111 | \ |
| NOP (SLL) | 000000 | 000000 |

由以上的控制信号的编码绘制出译码器。

### 译码电路图

可以看作是一个与电路，也可以看作是独热编码

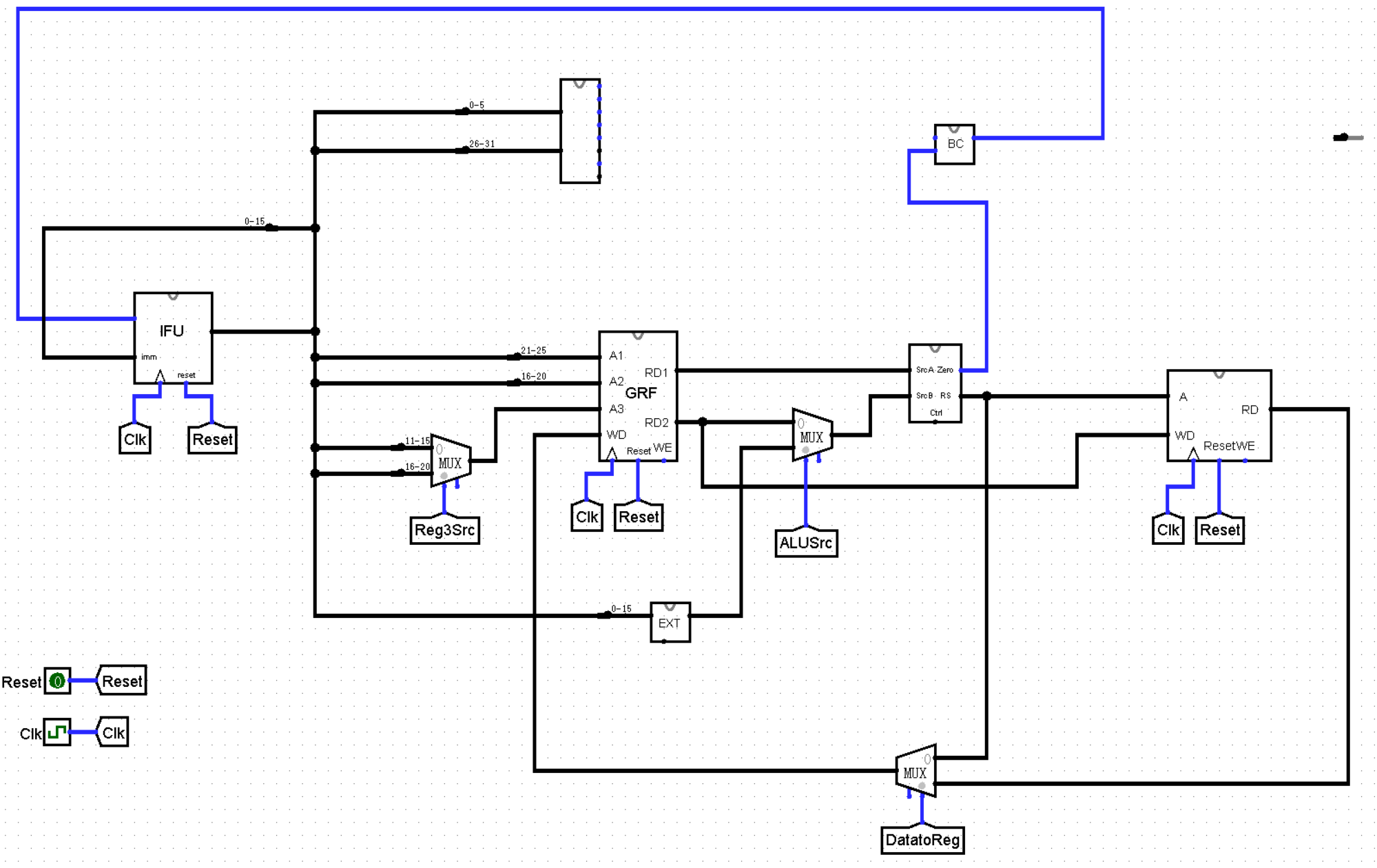
# 2、数据流与控制信号定义

### 数据流：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | IFU  .Imm | GRF  .A1 | GRF  .A2 | GRF  .A3 | GRF  .WD | ALU  .SrcA | ALU  .SrcB | DM  .A | DM  .D | EXT  .Imm | BC  .Zero |
| ADDU | \ | IFU  .instr  [25:21] | IFU  .instr  [20:16] | IFU  .instr  [15:11] | ALU  .Result | GRF  .RD1 | GRF  .RD2 | \ | \ | \ | \ |
| SUBU | \ | IFU  .instr  [25:21] | IFU  .instr  [20:16] | IFU  .instr  [15:11] | ALU  .Result | GRF  .RD1 | GRF  .RD2 | \ | \ | \ | \ |
| ORI | \ | IFU  .instr  [25:21] | \ | IFU  .instr  [20:16] | ALU  .Result | GRF  .RD1 | EXT  .Re | \ | \ | IFU  .instr  [15:0] | \ |
| LW | \ | IFU  .instr  [25:21] | \ | IFU  .instr  [20:16] | DM  .RD | GRF  .RD1 | EXT  .Re | ALU  .Re | \ | IFU  .instr  [15:0] | \ |
| SW | \ | IFU  .instr  [25:21] | IFU  .instr  [20:16] | \ | \ | GRF  .RD1 | EXT  .Re | ALU  .Re | GRF  .RD2 | IFU  .instr  [15:0] | \ |
| BEQ | IFU  .instr  [15:0] | IFU  .instr  [25:21] | IFU  .instr  [20:16] | \ | \ | GRF  .RD1 | GRF  .RD2 | \ | \ | \ | ALU  .Zero |
| LUI | \ | IFU  .instr  [25:21] | \ | IFU  .instr  [20:16] | ALU  .Result | GRF  .RD1 | EXT  .Re | \ | \ | IFU  .instr  [15:0] | \ |
| SLL | \ | \ | IFU  .instr  [20:16] | IFU  .instr  [15:11] | ALU  .Result | \ | GRF  .RD2 | \ | \ | \ | \ |
| J | {\*,IFU  .instr  [25:0]} | \ | \ | \ | \ | \ | \ | \ | \ | \ | \ |
| 汇总 | IFU  .instr  [15:0]  {\*,IFU  .instr  [25:0]} | IFU  .instr  [25:21 | IFU  .instr  [20:16] | IFU  .instr  [15:11]  IFU  .instr  [20:16] | ALU  .Result  DM  .RD | GRF  .RD1 | GRF  .RD2  EXT  .Re | ALU  .Re | GRF  .RD2 | IFU  .instr  [15:0] | ALU  .Zero |
| 对应控制信号 | IFUIM  M | \ | \ | Reg3Src | Data  toReg | \ | ALUSrc | \ | \ | \ | \ |

这里有几个显然固定的数据流没有标识，比如control的opcode和funct、ALU的shamt。 以及一些不好分辨时数据流还是控制信号的，比如nPC\_ct 。

通过这些数据通路，确定选择器的控制信号，并绘制出数据通路如下：(下图未加sll与j指令);



### 控制信号真值表：

通过以上的数据通路列表，确定选择信号，并通过器件的使用情况来确定各个元件的写使能信号和模式。

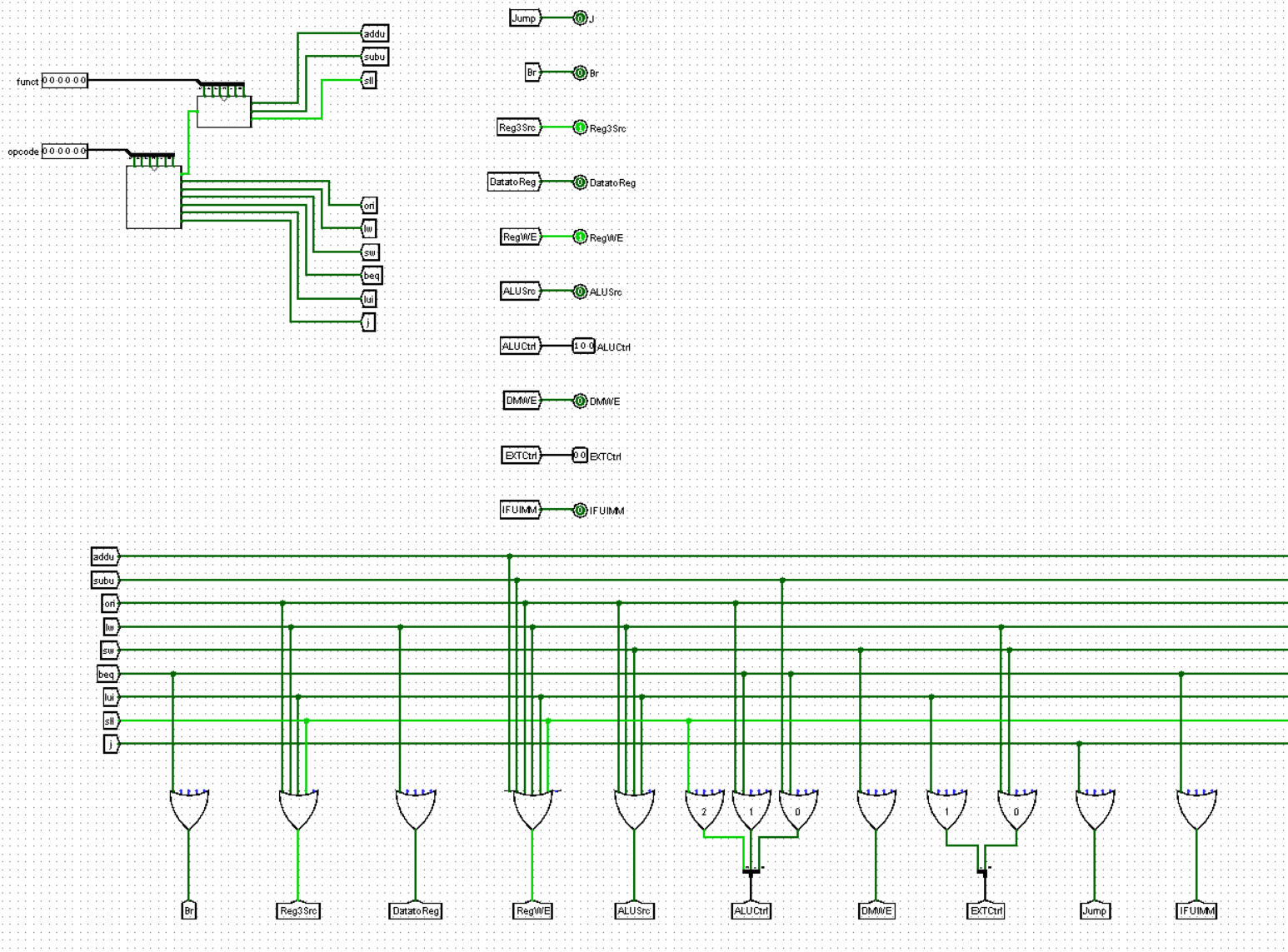
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Br | J | IFU  IMM | Reg3Src | DatatoReg | RegWE | ALUSrc | ALUCtrl | DMWE | EXTCtrl |
| ADDU | 0 | 0 | X | 0 | 0 | 1 | 0 | 000 | 0 | X |
| SUBU | 0 | 0 | X | 0 | 0 | 1 | 0 | 001 | 0 | X |
| ORI | 0 | 0 | X | 1 | 0 | 1 | 1 | 010 | 0 | 00 |
| LW | 0 | 0 | X | 1 | 1 | 1 | 1 | 000 | 0 | 01 |
| SW | 0 | 0 | X | X | X | 0 | 1 | 000 | 1 | 01 |
| BEQ | 1 | 0 | 1 | X | X | 0 | 0 | 011 | 0 | X |
| LUI | 0 | 0 | X | 1 | 0 | 1 | 1 | 000 | 0 | 10 |
| SLL  (NOP) | 0 | 0 | X | 1 | 0 | 1 | 0 | 100 | 0 | X |
| J | 0 | 1 | 0 | X | X | 0 | X | X | 0 | X |

为了方便，以上的值为X时，均取0.

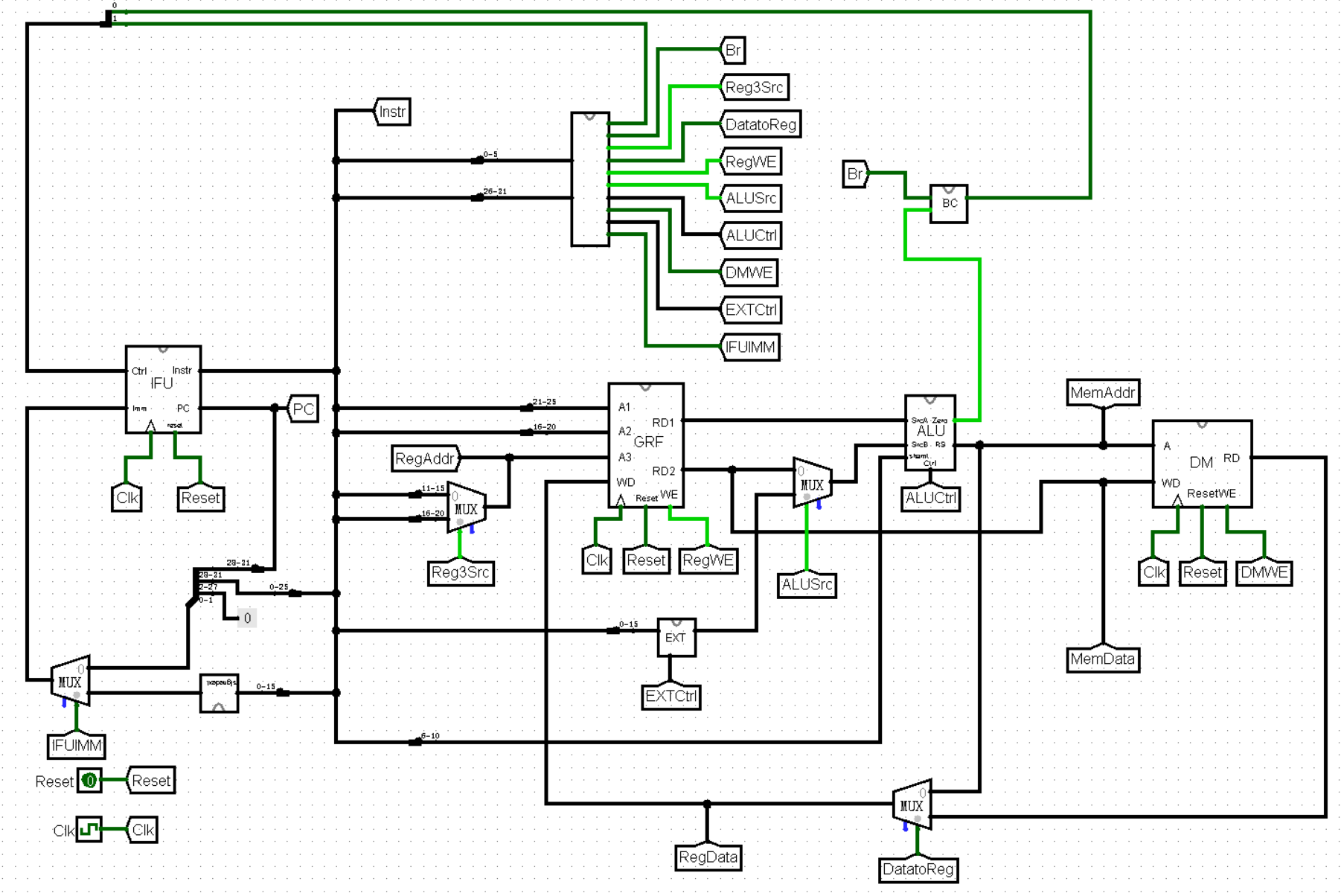
每个控制信号可以理解为指令的独热编码的或（和）。

这里的J和BC的输出信号一起构成nPC\_ct。

以下为电路：



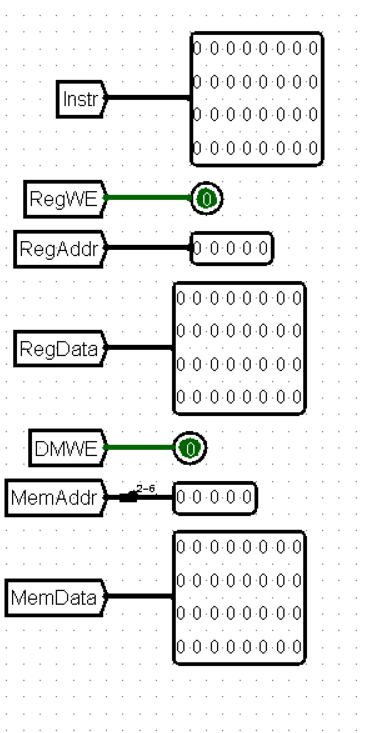
## 3、连接控制信号，完成cpu



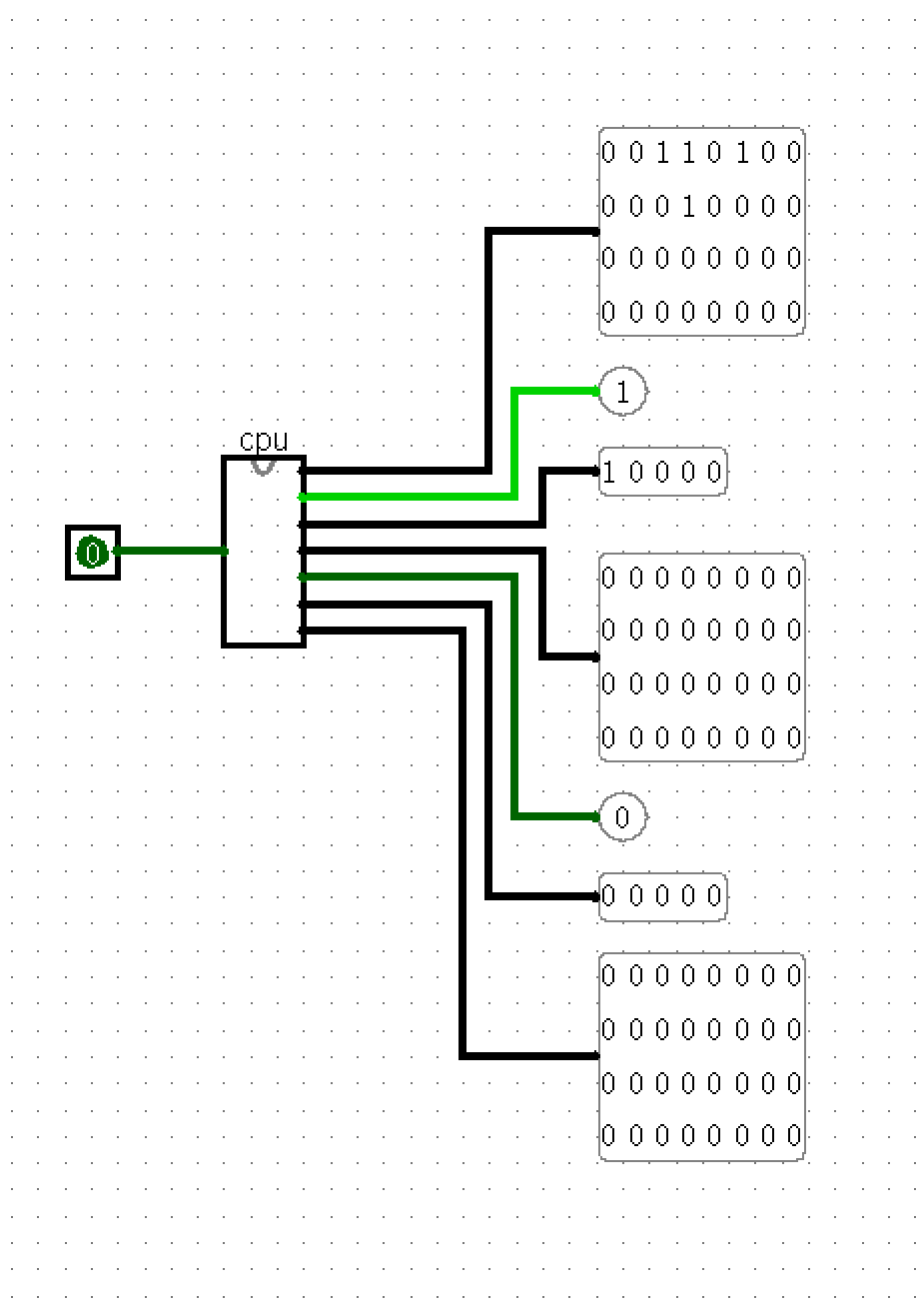
# 三、CPU的测试

## 1、测试电路

### 使用tunnel引出关键数据



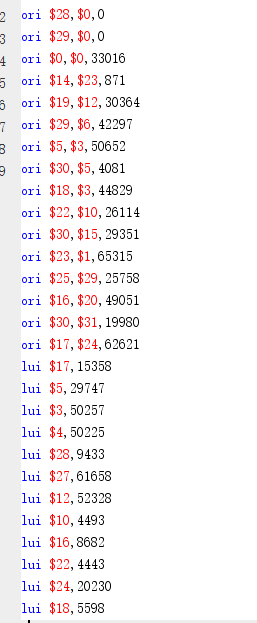
### 测试电路



## 2、测试集

### 测试用例1

题目给出的测试代码：

图源自讨论区

### 测试用例2

**ori** $s1, $0, 32

**ori** $s2, $0, 0

**ori** $s0, $0, 0

for\_1\_begin:

**beq** $s0, $s1, for\_1\_end

**sw** $s0, 0($s2)

**ori** $t0, $0, 4

**addu** $s2, $s2, $t0

**ori** $t0, $0, 1

**addu** $s0, $s0, $t0

**beq** $0, $0, for\_1\_begin

for\_1\_end:

**nop**

简单的把内存0-31填充上数字的程序。

期望下，内存中0-31地址分别被填充0-31

### 测试用例3

**ori** $s2, $0, 0 #the address of a[0]

**ori** $t0, $0, 0x7111 #to try the lui //lui is {imm,{16{0}}}, not {imm, $rs}

**lui** $t0, 0x7

**ori** $t0, 0x8000 #0x0078000([39:24]=01111000)

**sw** $t0, 0($s2) #a[0] = 0x00010000

**ori** $s0, $0, 1

**ori** $s1, $0, 32

for\_1\_begin:

**beq** $s0, $s1, for\_1\_end #for i = 1 to 31

**nop**

**ori** $t0, $0, 4

**addu** $s2, $s2, $t0

**lw** $t0, -4($s2) #get a[i-1]

**ori** $t1, $0, 0x8000 #try zero extern (16'b1000...)

**subu** $t0, $t0, $t1

**sw** $t0, 0($s2) #a[i]=a[i-1]-0x8000 // can less than 0

**ori** $t0, $0, 1

**addu** $s0, $s0, $t0

**beq** $0, $0, for\_1\_begin

**nop**

for\_1\_end:

期望下内存从0x00078000填填充至0xfff80000

# 四、思考题

## 模块规格（L0.T2）

### 若PC（程序计数器）位数为30位，试分析其与32位PC的优劣。

两种方法各有优劣：

如使用30位，可以节约一定的寄存器资源，同时在j和branch指令时也不需要移位，但是在指令储存器前需要左移两位，因为mips储存器地址按字节寻址。（当然，logisim里面的ROM和RAM是按字寻址的）同时，jr指令需要移位。

如果使用32位，j和branch指令的立即数需要移位，但是jr指令的立即数不需要移位。同时，寻址时可以直接寻址。

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

GRF使用寄存器合理，DM使用RAM也合理，但是IM使用ROM，不一定合理。因为现代的真实cpu中，指令储存器和数据储存器都在内存（RAM）或缓存（CACHE）中。同时，真是情况下，指令时可以改变的。

如果需要改进的话，如果多周期的cpu，就可以把IM和DM放在一起。

## L0.T3

### 结合上文给出的样例真值表，给出RegDst， ALUSrc， MemtoReg，RegWrite, nPC\_Sel, ExtOp与op和func有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

RegDst= ~op[5] ~op[4] ~op[3] ~op[2] ~op[1] ~op[0] func[5] ~func[4] ~func[3] ~func[2] ~func[1] ~func[0]+ ~op[5] ~op[4] ~op[3] ~op[2] ~op[1] ~op[0] func[5] ~func[4] ~func[3] ~func[2] func[1] ~func[0]

ALUSrc=~op[5] ~op[4] op[3] op[2] ~op[1] op[0]+ op[5] ~op[4] ~op[3] ~op[2] op[1] op[0]+ op[5] ~op[4] op[3] ~op[2] op[1] op[0]

MemtoReg= op[5] ~op[4] ~op[3] ~op[2] op[1] op[0]

RegWrite=~op[5] ~op[4] ~op[3] ~op[2] ~op[1] ~op[0] func[5] ~func[4] ~func[3] ~func[2] ~func[1] ~func[0]+ ~op[5] ~op[4] ~op[3] ~op[2] ~op[1] ~op[0] func[5] ~func[4] ~func[3] ~func[2] func[1] ~func[0]+ ~op[5] ~op[4] op[3] op[2] ~op[1] op[0]+ op[5] ~op[4] ~op[3] ~op[2] op[1] op[0]

nPC\_Sel= ~op[5] ~op[4] ~op[3] op[2] ~op[1] ~op[0]

ExtOp= op[5] ~op[4] ~op[3] ~op[2] op[1] op[0]+ op[5] ~op[4] op[3] ~op[2] op[1] op[0]

### 充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

RegDst= ~op[5] ~op[4]

ALUSrc= op[0]

MemtoReg= op[5]

RegWrite=~ ~op[3] ~op[2]+ op[3] op[2]

nPC\_Sel= op[2] ~op[1] ~op[0]

ExtOp=op[1] op[0]

### 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？请给出你的理由。

如果nop指令位0x00000000，其实相当于 sll $0, $0, $0。如果所搭建的指令集中包含了sll，的确就不需要了。如果不包含，则不加入真值表的话，单路的结果将会是所有控制信号全0，这样，写入信号全0，这条指令就不会有任何操作。

## L0.T4

### 前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

片选信号(chip select)即为选择使用哪个芯片的信号，一般是判断地址信号的高位是否是在范围内，然后通过一个端口决定这个存储器是否可用。

这里判断地址的前四位是否为0003，不是的话，则使用logisim的RAM自带的sel端口禁用RAM即可。

### 除了编写程序进行测试外，还有一种验证CPU设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

形式验证可以证明系统不存在某个缺陷，而软件测试不行，它只能判断系统是否可以通过当前数据集。同时，形式验证可以判断软件是否满足某些属性。

但是，形式验证有一定的限制，同时，如果采用定理证明的方法，则需要较多的用户干预。