****

**《CPU设计与仿真》**

**课程设计报告**

学 号： 20141002648

班级序号： 111141-14

姓 名： 吴子鸿

指导教师： 杨林权、罗忠文

成 绩：

**中国地质大学信息工程学院**

**2015年 6 月**

**实习题目一**

**·题目要求：**

通过在文件proj1.cpp中增加代码来完成整个程序。你的仿真器必须能仿真以下MIPS机器指令:

addu Rdest, Rsrc1, Rsrc2

addiu Rdest, Rsrc1, imm

subu Rdest, Rsrc1, Rsrc2

sll Rdest, Rsrc, shamt

srl Rdest, Rsrc, shamt

and Rdest, Rsrc1, Rsrc2

andi Rdest, Rsrc, imm

or Rdest, Rsrc1, Rsrc2

ori Rdest, Rsrc, imm

lui Rdest, imm

slt Rdest, Rsrc1, Rsrc2

beq Rsrc1, Rsrc2, raddr

bne Rsrc1, Rsrc2, raddr

j address

jal address

jr Rsrc

lw Rdest, offset (Radd)

sw Rsrc, offset (Radd)

任务是编写disassembled （反汇编）和 simulateInstr（指令仿真）函数的代码(”proj1.cpp”文件中), 当然可能还有一些相关的你希望用到的辅助函数的代码。可以假设仿真指令不会出现寻址错误. **如果程序遇到没有列出的指令，它将退出.**

**·实现过程：**

首先，大致知道了题目中各项函数的作用，然后开始构思如何写函数。

对于”disassembled”函数，“instr”变量中存的是10进制的机器码，pc变量中存储的是当前运行到的PC的值。我先到网上下载mips指令格式对照表，将题目中所需要的指令查表翻译成机器码进行对照，通过将instr数字右移26位，截取opcode段，若opcode=0，则是R指令，截取funct段来判断是哪条R指令，否则则不是R指令，通过判断opcode段来看是哪条指令，然后通过截取相应的段（rs、rt、rd、shamt、funct、immediate、address等），再将其转化为字符串，如有需要进行10进制、16进制转换或者符号位的判断和值的改变，或者进行10进制转换为16进制地址。最终通过strcpy以及strcat函数将相对应的字符串拼接到字符串s中，最终返回s，即可，若遇到无法判别的指令，则直接通过exit（0）退出程序。

对于”simulateinstr”函数，与”disassembled”函数判断指令的做法大致相同，在进行操作时，若有寄存器值改变，则\*changedReg值则为改变的寄存器编号，并且将相对应的寄存器的值进行改变，否则\*changedReg=-1；若有内存的改变则\*changedMem为改变的内存地址，并且将相对应的内存地址值进行改变，否则\*changedMem=-1,若有PC的值改变，则改变当前mips->pc的值，否则将pc+4模拟进行下一条指令。

综上方法，完成了代码的编写。

附：mips指令格式对照

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| MIPS 指令集(共31条） | | | | | | | | | |
| 指令 | 指令格式 | | | | | | 示例 | 示例含义 | 操作及其解释 |
| Bit # | 31..26 | 25..21 | 20..16 | 15..11 | 10..6 | 5..0 |  |  |  |
| R-ty，pe | op | rs | rt | rd | shamt | func |  |  |  |
| addu | 000000 | rs | rt | rd | 00000 | 100001 | addu $1,$2,$3 | $1=$2+$3 | rd <- rs + rt   ；其中rs＝$2，rt=$3, rd=$1,无符号数 |
| subu | 000000 | rs | rt | rd | 00000 | 100011 | subu $1,$2,$3 | $1=$2-$3 | rd <- rs - rt   ；其中rs＝$2，rt=$3, rd=$1,无符号数 |
| and | 000000 | rs | rt | rd | 00000 | 100100 | and $1,$2,$3 | $1=$2 & $3 | rd <- rs & rt   ；其中rs＝$2，rt=$3, rd=$1 |
| or | 000000 | rs | rt | rd | 00000 | 100101 | or $1,$2,$3 | $1=$2 | $3 | rd <- rs | rt   ；其中rs＝$2，rt=$3, rd=$1 |
| slt | 000000 | rs | rt | rd | 00000 | 101010 | slt $1,$2,$3 | if($2<$3)    $1=1 else    $1=0 | if (rs < rt) rd=1 else rd=0 ；其中rs＝$2，rt=$3, rd=$1 |
| sll | 000000 | 00000 | rt | rd | shamt | 000000 | sll $1,$2,10 | $1=$2<<10 | rd <- rt << shamt  ；shamt存放移位的位数，   也就是指令中的立即数，其中rt=$2, rd=$1 |
| srl | 000000 | 00000 | rt | rd | shamt | 000010 | srl $1,$2,10 | $1=$2>>10 | rd <- rt >> shamt ；(logical) ，其中rt=$2, rd=$1 |
| jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 | jr $31 | goto $31 | PC <- rs |
| I-type | op | rs | rt | immediate | | |  |  |  |
| addiu | 001001 | rs | rt | immediate | | | addiu $1,$2,100 | $1=$2+100 | rt <- rs + (zero-extend)immediate ；其中rt=$1,rs=$2 |
| andi | 001100 | rs | rt | immediate | | | andi $1,$2,10 | $1=$2 & 10 | rt <- rs & (zero-extend)immediate ；其中rt=$1,rs=$2 |
| ori | 001101 | rs | rt | immediate | | | andi $1,$2,10 | $1=$2 | 10 | rt <- rs | (zero-extend)immediate ；其中rt=$1,rs=$2 |
| lui | 001111 | 00000 | rt | immediate | | | lui $1,100 | $1=100\*65536 | rt <- immediate\*65536 ；将16位立即数放到目标寄存器高16          位，目标寄存器的低16位填0 |
| lw | 100011 | rs | rt | immediate | | | lw $1,10($2) | $1=memory[$2  +10] | rt <- memory[rs + (sign-extend)immediate] ；rt=$1,rs=$2 |
| sw | 101011 | rs | rt | immediate | | | sw $1,10($2) | memory[$2+10]  =$1 | memory[rs + (sign-extend)immediate] <- rt ；rt=$1,rs=$2 |
| beq | 000100 | rs | rt | immediate | | | beq $1,$2,10 | if($1==$2)    goto PC+4+40 | if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2 |
| bne | 000101 | rs | rt | immediate | | | bne $1,$2,10 | if($1!=$2)  goto PC+4+40 | if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2 |
| J-type | op | address | | | | |  |  |  |
| j | 000010 | address | | | | | j 10000 | goto 10000 | PC <- (PC+4)[31..28],address,0,0   ；address=10000/4 |
| jal | 000011 | address | | | | | jal 10000 | $31<-PC+4;  goto 10000 | $31<-PC+4；PC <- (PC+4)[31..28],address,0,0    ；address=10000/4 |

**·程序代码：**

int tenc(int instr) //十进制正负数转换

{

int x,im;

im=(instr<<16)>>16;

x=im>>15;

if (x==1)

im=-((im-1) ^ 0xffff);

return im;

};

char \*reverse(char \*str)

{

if( !str )

{

return NULL;

}

int len = strlen(str);

char temp;

for( int i = 0; i < len / 2; i++ )

{

// 交换前后两个相应位置的字符

temp = \*(str + i);

\*(str + i) = \*(str + len - 1 - i);

\*(str + len - 1 - i) = temp;

}

return str;

}

char \* tentohex(int imx) //instr截取立即数十进制转换为16进制

{

int k,x,im;

char a[100],b[100];

im=imx;

if (im==0)

return "0";

if (im<0)

im=im+65536;

x=0;

strcpy(a,"");

while (im!=0)

{

k=im%16;

im=im/16;

switch (k)

{

case 0: strcpy(b,"0"); break;

case 1: strcpy(b,"1"); break;

case 2: strcpy(b,"2"); break;

case 3: strcpy(b,"3"); break;

case 4: strcpy(b,"4"); break;

case 5: strcpy(b,"5"); break;

case 6: strcpy(b,"6"); break;

case 7: strcpy(b,"7"); break;

case 8: strcpy(b,"8"); break;

case 9: strcpy(b,"9"); break;

case 10: strcpy(b,"a"); break;

case 11: strcpy(b,"b"); break;

case 12: strcpy(b,"c"); break;

case 13: strcpy(b,"d"); break;

case 14: strcpy(b,"e"); break;

case 15: strcpy(b,"f"); break;

}

x++;

strcat(a,b);

};

strcpy(a,reverse(a));

return a;

};

char \* disassembled (unsigned int instr, unsigned int pc) {

int opcode,rs,rt,rd,shamt,funct,im,ad;

char a[10],b[10]="$",c[10]="$",d[10]="$",e[10],s[100],k[10];

opcode=instr>>26;

if (opcode==0)

{

funct=(instr<<26)>>26; //切割

rs=(instr<<6)>>27;

rt=(instr<<11)>>27;

rd=(instr<<16)>>27;

shamt=(instr<<21)>>27;

sprintf(k, "%d", rs);

strcat(b,k);

sprintf(k,"%d",rt);

strcat(c,k);

sprintf(k,"%d",rd);

strcat(d,k);

sprintf(e,"%d",shamt);

switch (funct)

{

case 33: strcpy(a,"addu"); break;

case 35: strcpy(a,"subu"); break;

case 36: strcpy(a,"and"); break;

case 37: strcpy(a,"or"); break;

case 8: strcpy(a,"jr"); break;

case 0: strcpy(a,"sll"); break;

case 2: strcpy(a,"srl"); break;

case 42: strcpy(a,"slt"); break;

};

if ((funct==33)||(funct==35)||(funct==36)||(funct==37)||(funct==42))

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,d);

strcat(s,", ");

strcat(s,b);

strcat(s,", ");

strcat(s,c);

}

if ((funct==0)||(funct==2))

{

if ((rd==0)&&(rt==0)&&(rd==0))

exit(0);

strcpy(s,a);

strcat(s,"\t");

strcat(s,d);

strcat(s,", ");

strcat(s,c);

strcat(s,", ");

strcat(s,e);

}

if (funct==8)

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,b);

}

}

if ((opcode==9)||(opcode==12)||(opcode==13)||(opcode==15)||(opcode==4)||(opcode==5)||(opcode==35)||(opcode==43))

{

rs=(instr<<6)>>27;

rt=(instr<<11)>>27;

if ((opcode==9)||(opcode==35)||(opcode==43))

im=tenc(instr);

else

im=(instr<<16)>>16;

sprintf(k, "%d", rs);

strcat(b,k);

sprintf(k,"%d",rt);

strcat(c,k);

if ((opcode==12)||(opcode==13)||(opcode==15))

strcpy(e,tentohex(im));

else if ((opcode==4)||(opcode==5))

strcpy(e,tentohex(tenc(im)\*4+pc+4));

else

sprintf(e,"%d",im);

switch (opcode)

{

case 9: strcpy(a,"addiu"); break;

case 12: strcpy(a,"andi"); break;

case 13: strcpy(a,"ori"); break;

case 15: strcpy(a,"lui"); break;

case 4: strcpy(a,"beq"); break;//

case 5: strcpy(a,"bne"); break;//

case 35: strcpy(a,"lw"); break;

case 43: strcpy(a,"sw"); break;

};

if ((opcode==9)||(opcode==12)||(opcode==13))

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,c);

strcat(s,", ");

strcat(s,b);

strcat(s,", ");

if ((opcode==12)||(opcode==13))

strcat(s,"0x");

strcat(s,e);

}

if ((opcode==35)||(opcode==43))

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,c);

strcat(s,", ");

strcat(s,e);

strcat(s,"(");

strcat(s,b);

strcat(s,")");

}

if ((opcode==4)||(opcode==5))

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,b);

strcat(s,", ");

strcat(s,c);

strcat(s,", 0x00");

strcat(s,e);

}

if (opcode==15)

{

strcpy(s,a);

strcat(s,"\t");

strcat(s,c);

strcat(s,", ");

strcat(s,"0x");

strcat(s,e);

}

}

if ((opcode==2)||(opcode==3))

{

ad=(instr<<6)>>6;

switch (opcode)

{

case 2: strcpy(a,"j"); break;//

case 3: strcpy(a,"jal"); break;//

};

strcpy(e,tentohex(ad\*4));

strcat(s,a);

strcat(s,"\t0x00");

strcat(s,e);

}

/\* You replace this code by the right stuff. \*/

//if (/\*instruction isn't supported \*/) exit (0); // Your program must exit when an unsupported instruction is detected

if (s=="")

exit(0);

return s;

}

/\*

\* Simulate the execution of the given instruction, updating the

\* pc appropriately.

\*

\* If the instruction modified a register--i.e. if it was lw,

\* addu, addiu, subu, sll, srl, and, andi, or, ori, lui, or slt

\* to list a few examples-- return the index of the modified

\* register in \*changedReg, otherwise return -1 in \*changedReg.

\* Note that you should never return 0 in \*changedReg, since

\* $0 cannot be changed! Note that even if the instruction

\* changes the register back to it's old value

\* (e.g. addu $3, $3, $0) the destination register ($3 in the

\* example) should be marked changed!

\*

\* If the instruction was sw, return the address of the

\* updated memory location in \*changedMem, otherwise return -1

\* in \*changedMem.

\*/

void simulateInstr (Computer mips, unsigned int instr, int \*changedReg, int \*changedMem) {

int opcode,rs,rt,rd,shamt,funct,im,ad;

opcode=instr>>26;

\*changedReg = -1;

\*changedMem = -1;

mips->pc = mips->pc + 4;

if (opcode==0)

{

funct=(instr<<26)>>26; //切割

rs=(instr<<6)>>27;

rt=(instr<<11)>>27;

rd=(instr<<16)>>27;

shamt=(instr<<21)>>27;

if (funct==33)

if(rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rs]+mips->registers[rt];

}

if (funct==35)

if(rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rs]-mips->registers[rt];

}

if (funct==0)

if(rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rt]<<shamt;

}

if (funct==2)

if (rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rt]>>shamt;

}

if (funct==36)

if (rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rs]&mips->registers[rt];

}

if (funct==37)

if (rd!=0)

{

\*changedReg=rd;

mips->registers[rd]=mips->registers[rs]|mips->registers[rt];

}

if (funct==42)

if (rd!=0)

{

\*changedReg=rd;

if (mips->registers[rs]<mips->registers[rt])

mips->registers[rd]=1;

else

mips->registers[rd]=0;

}

if (funct==8)

{

mips->pc=mips->registers[rs];

}

}

if ((opcode==9)||(opcode==12)||(opcode==13)||(opcode==15)||(opcode==4)||(opcode==5)||(opcode==35)||(opcode==43))

{

rs=(instr<<6)>>27;

rt=(instr<<11)>>27;

if ((opcode==9)||(opcode==35)||(opcode==43))

im=tenc(instr);

else

im=(instr<<16)>>16;

if (opcode==9)

if ((rt!=0))

{

\*changedReg=rt;

mips->registers[rt]=mips->registers[rs]+im;

}

if (opcode==12)

if (rt!=0)

{

\*changedReg=rt;

mips->registers[rt]=mips->registers[rs]&im;

}

if (opcode==13)

if (rt!=0)

{

\*changedReg=rt;

mips->registers[rt]=mips->registers[rs]|im;

}

if (opcode==15)

if (rt!=0)

{

\*changedReg=rt;

mips->registers[rt]=im\*65536;

}

if (opcode==4)

{

if (mips->registers[rt]==mips->registers[rs])

{

mips->pc=mips->pc+im\*4;

}

}

if (opcode==5)

{

if (mips->registers[rt]!=mips->registers[rs])

{

mips->pc=mips->pc+im\*4;

}

}

if (opcode==35)

if (rt!=0)

{

\*changedReg=rt;

mips->registers[rt]=mips->memory[((mips->registers[rs]+im)-0x00400000)/4];

}

if (opcode==43)

if (rt!=0)

{

if (rt==17)

rt=rt;

\*changedMem=mips->registers[rs]+im;

mips->memory[(\*changedMem-0x00400000)/4]=mips->registers[rt];

}

}

if ((opcode==2)||(opcode==3))

{

ad=(instr<<6)>>6;

if (opcode==2)

{

mips->pc=ad\*4;

}

if (opcode==3)

{

\*changedReg=31;

mips->registers[31]=mips->pc;

mips->pc=ad\*4;

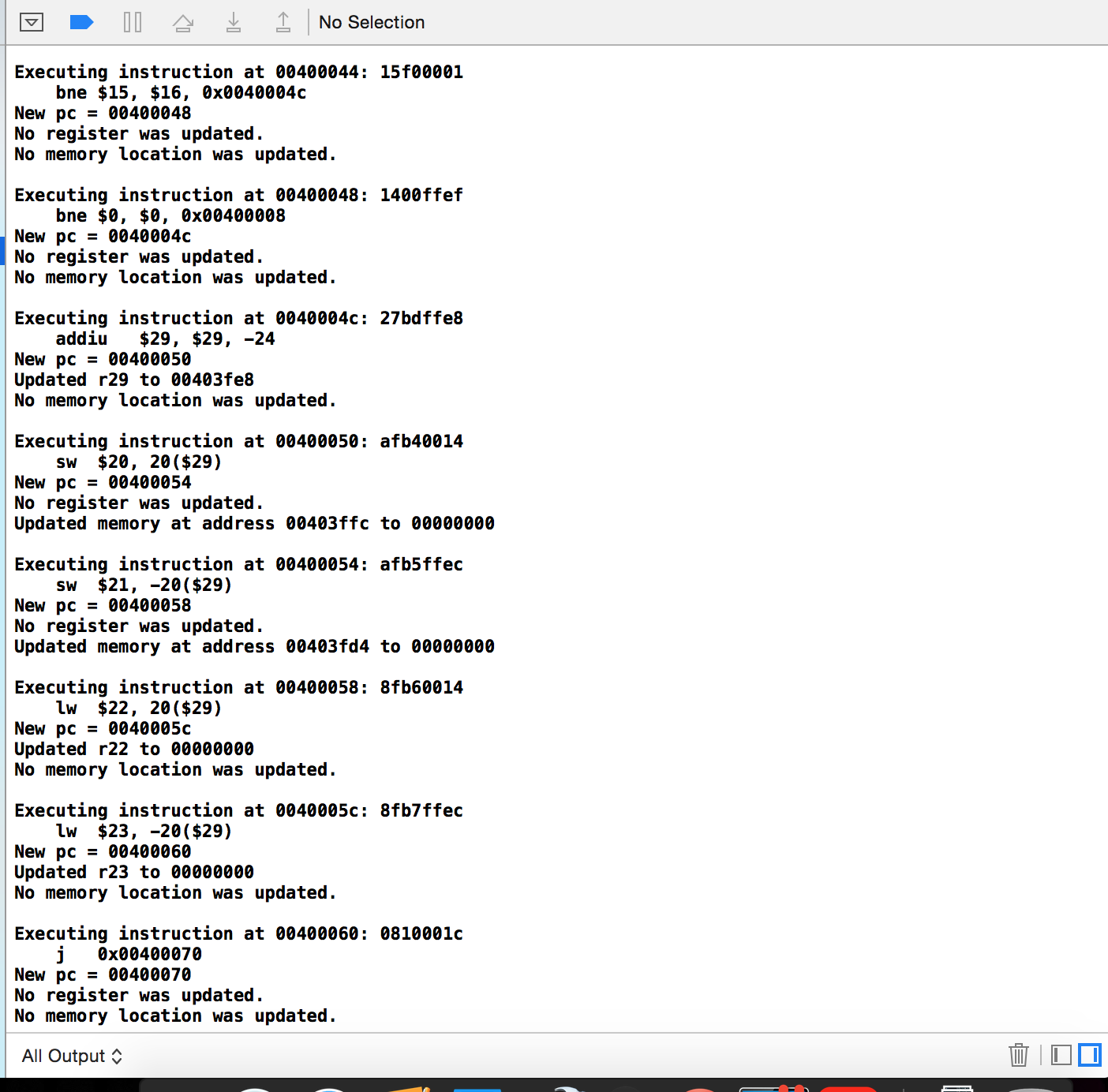
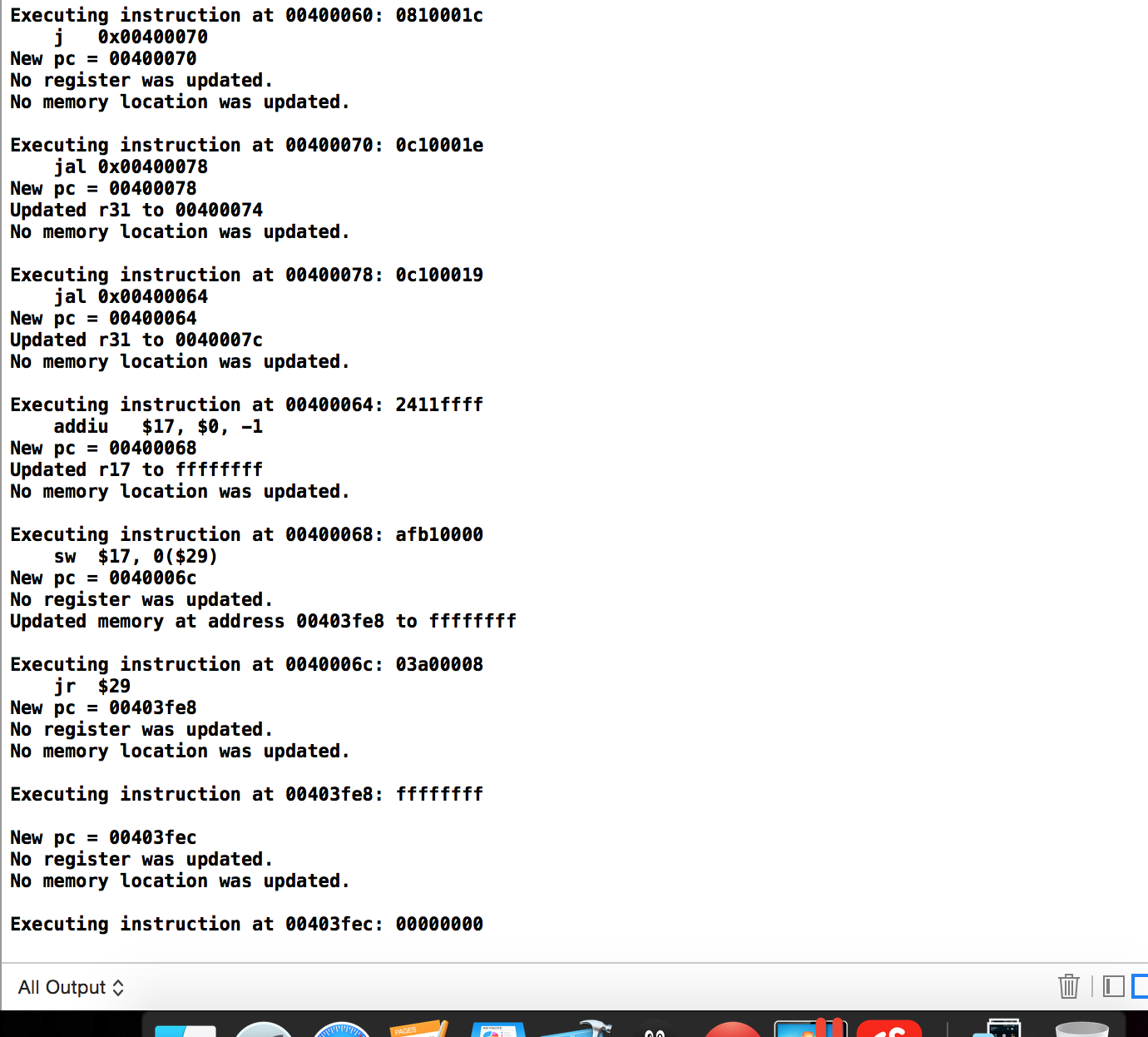
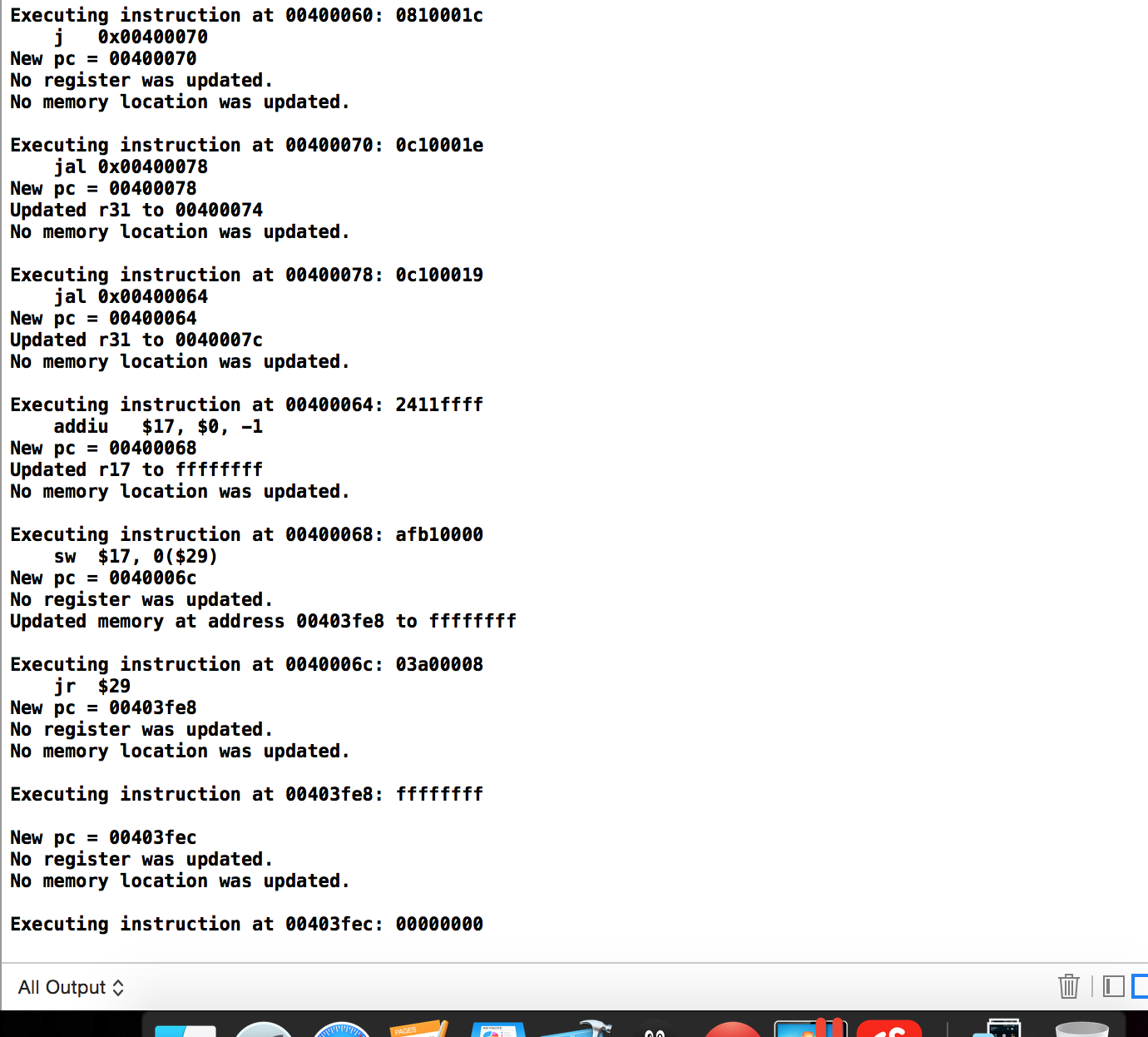
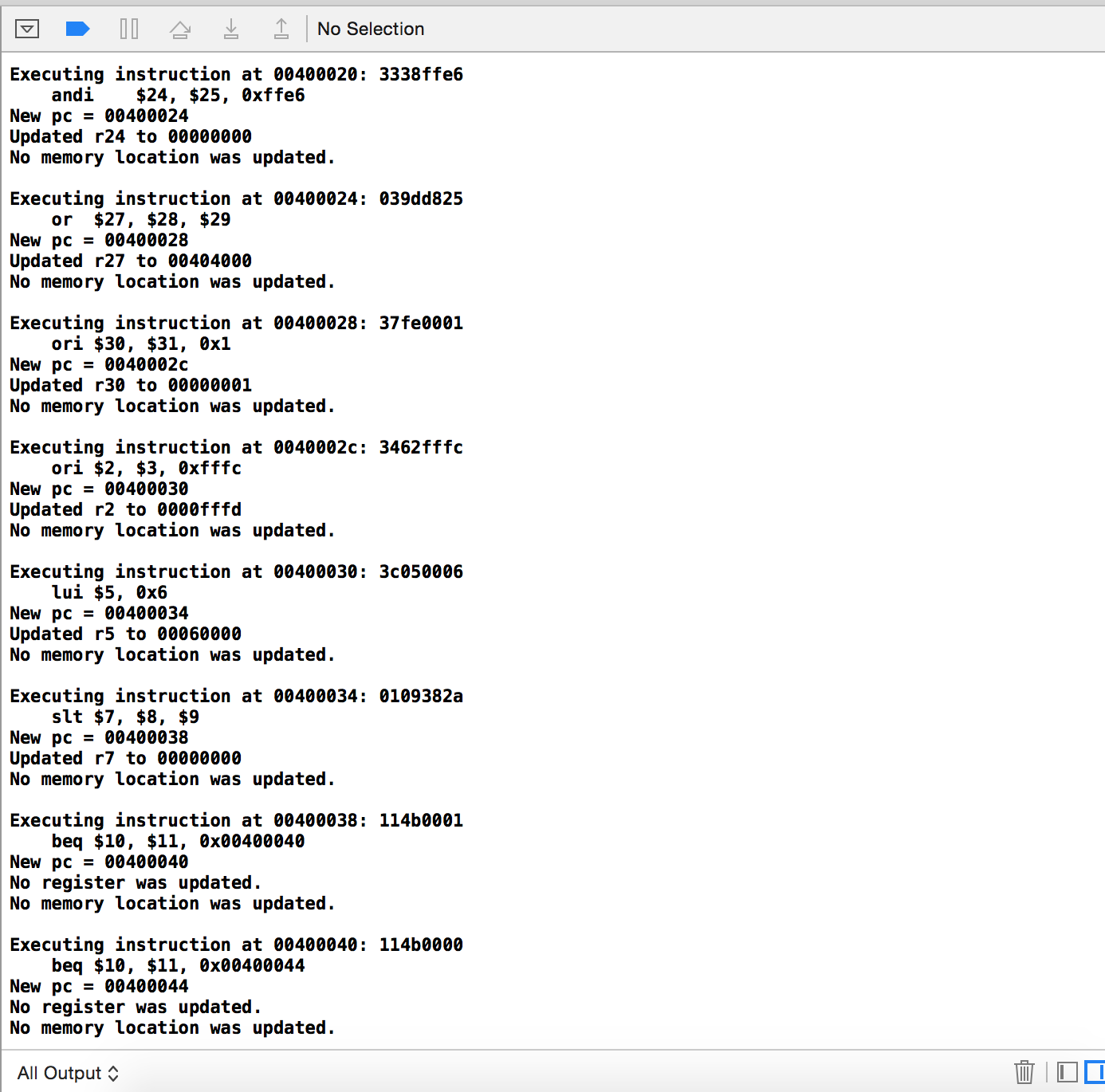
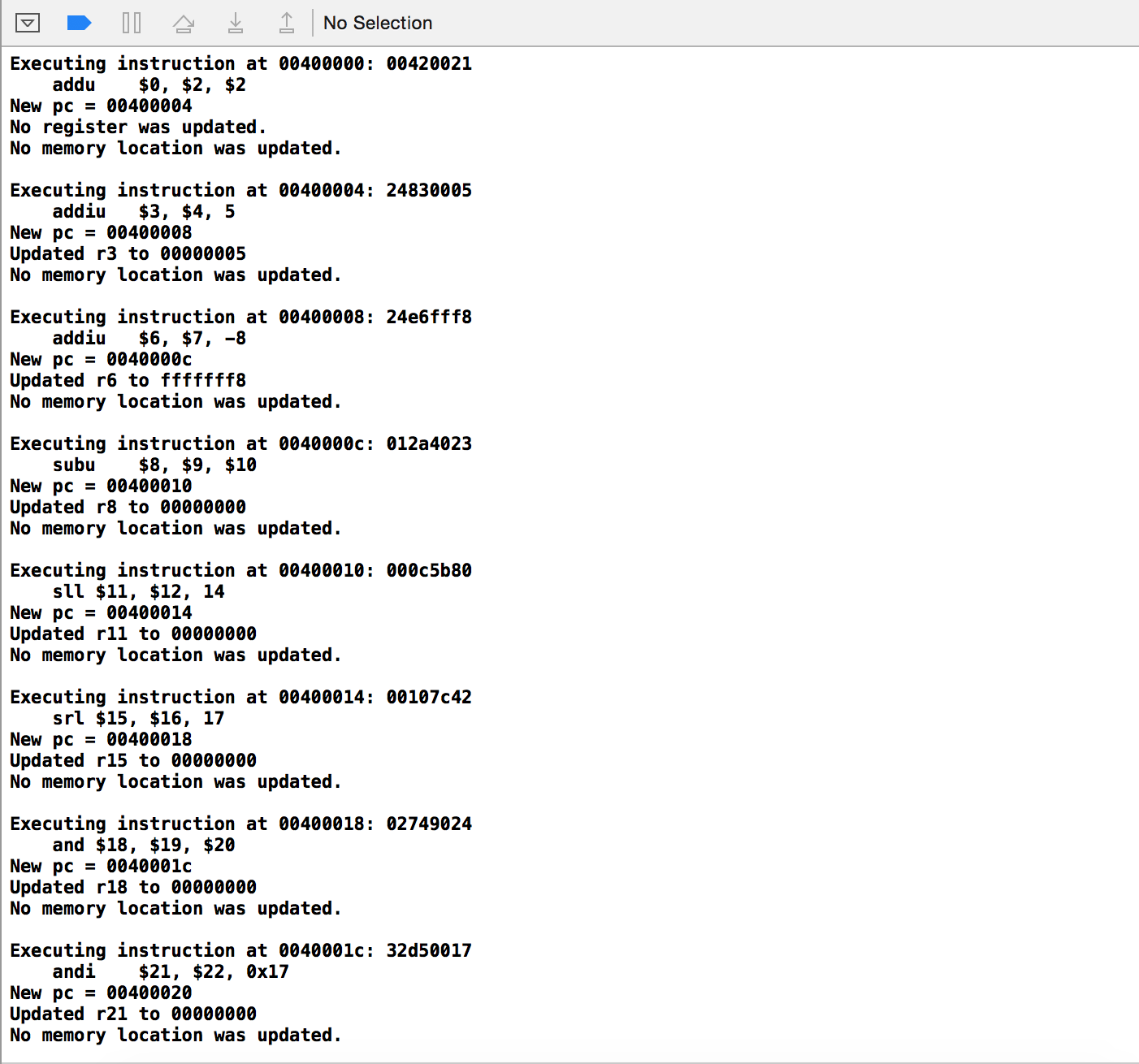
}

}

/\* You replace this code by the right stuff. \*/}

}

运行结果：



**实习题目二**

**·题目要求：**

在本次大作业中，需要使用[Logisim](http://ozark.hendrix.edu/%7Eburch/logisim/)来创建一个16-位单时钟周期CPU. 该CPU是一个全新的CPU,与老师上课所讲的有很大的不同,因此,请*仔细* 阅读本文档，其中包含了很多(课上没有讲的)新内容，同时也回答了各种可能的疑问。

指令集结构Instruction Set Architecture (ISA)

需要实现一个简单的16-位处理器(即每个指令字长为16位,寄存器也是16位)，该处理器有四个寄存器($r0到$r3). 具有独立的数据和指令内存(即有两个内存,一个指令内存,一个数据内存)。

重要注意事项: 由于Logisim的限制，也为了让事情更简单一些，我们以半字(16位)为单位 对内存编址! 这和MIPS 不同，MIPS指令是字长是32位,而内存是以字节(8位)为单位编址.

下表给出了指令编码表。通过查询 opcode字段(高四位，即15-12位)的值，可知半字编码所对应的指令 .注意，表中的opcode不到16个，而funct也不到8个. 原因是指令少一些，使同学们更容易实现(呵呵,好象比老师上课讲的CPU指令数还是多了很多).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15-12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | rs | | rt | | rd | | party bits! | | | funct | | | 参见R-type Instructions |
| 1 | rs | | rt | | immediate-u | | | | | | | | disp: DISP[imm] = $rs |
| 2 | rs | | rt | | immediate-u | | | | | | | | lui: $rt = imm << 8 |
| 3 | rs | | rt | | immediate-u | | | | | | | | ori: $rt = $rs | imm |
| 4 | rs | | rt | | immediate-s | | | | | | | | addi: $rt = $rs + imm |
| 5 | rs | | rt | | immediate-u | | | | | | | | andi: $rt = $rs & imm |
| 6 | rs | | rt | | immediate-s | | | | | | | | lw: $rt = MEM[$rs + imm] |
| 7 | rs | | rt | | immediate-s | | | | | | | | sw: MEM[$rs+imm] = $rt |
| 8 | jump address | | | | | | | | | | | | jump |
| 9 | rs | | rt | | offset | | | | | | | | beq |
| 10 | rs | | rt | | offset | | | | | | | | bne |

R-Type Instructions

|  |  |
| --- | --- |
| funct | meaning |
| 0 | or: $rd = $rs | $rt |
| 1 | and: $rd = $rs & $rt |
| 2 | add: $rd = $rs + $rt |
| 3 | sub: $rd = $rs - $rt |
| 4 | sllv: $rd = $rs << $rt |
| 5 | srlv: $rd = $rs >> $rt |
| 6 | srav: $rd = $rs >> $rt |
| 7 | slt: $rd = ($rs < $rt) ? 1 : 0 |

即：制作一个16位简易的cpu，包含4个寄存器，让其可以完美运行上述指令即可。

**·实现过程：**

首先按照题目要求，完成一个可以用来计算加减法、与或的ALU，然后做出RAM部件用来存储指令信息，还有4个寄存器，可以进行选择输入和选择输出。然后制作简单的PC，只需当每次时钟信号到达上升沿时+1即可。将这几个部件联系起来，做出最简单的cpu，然后考虑复杂的情形。

在制作ALU的过程中，我只做了有关R格式指令的运算器，对于beq和bne的判断无法处理，所以我又做了一个进行beq和bne指令的ALU，用来运算beq和bne的结果和对结果进行处理。

对于如何判断执行ALU中的哪一种运算，我做了一个ALU选择部件，通过不同的指令信息，选择相对应的计算结果。

由于我直接将4寄存器的值显示在4个数码管中，所以无需进行disp指令的操作和运算。

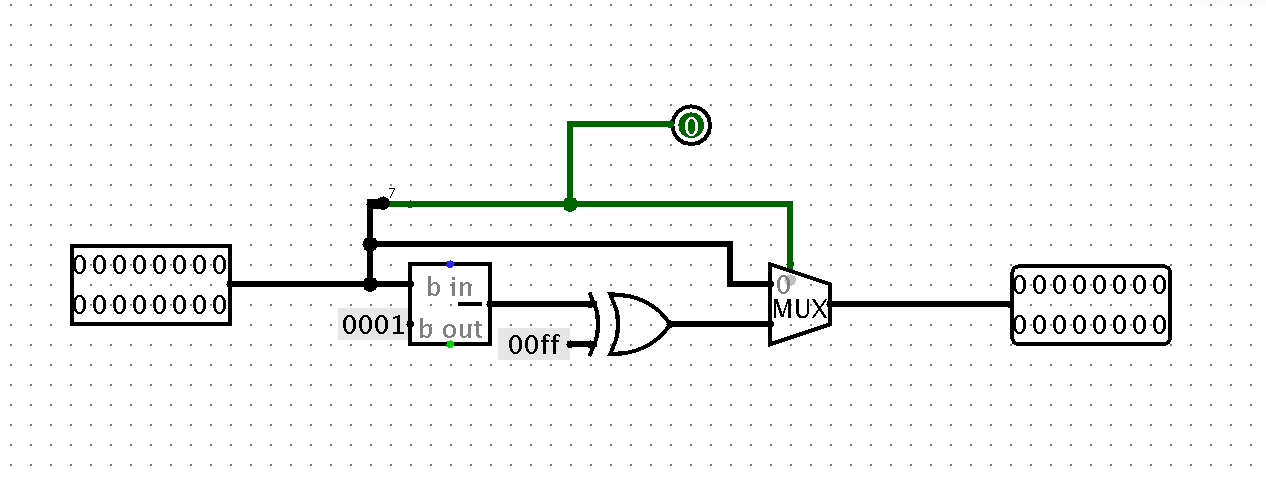
同时，我也将pc的值显示在一个数码管中。总共用了5个数码管。

最终，将数据通道连接完成，使用了n个数据选择器和解复用器。还有分离器、扩展器，最终将线路连好。

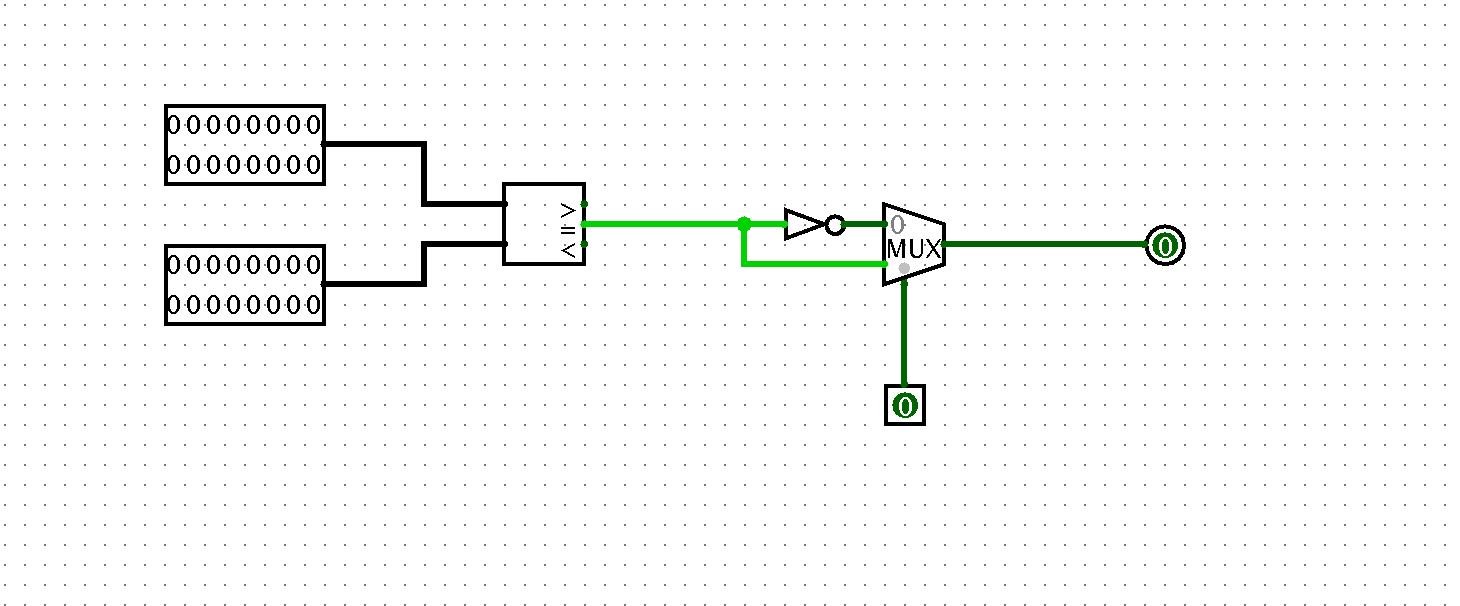
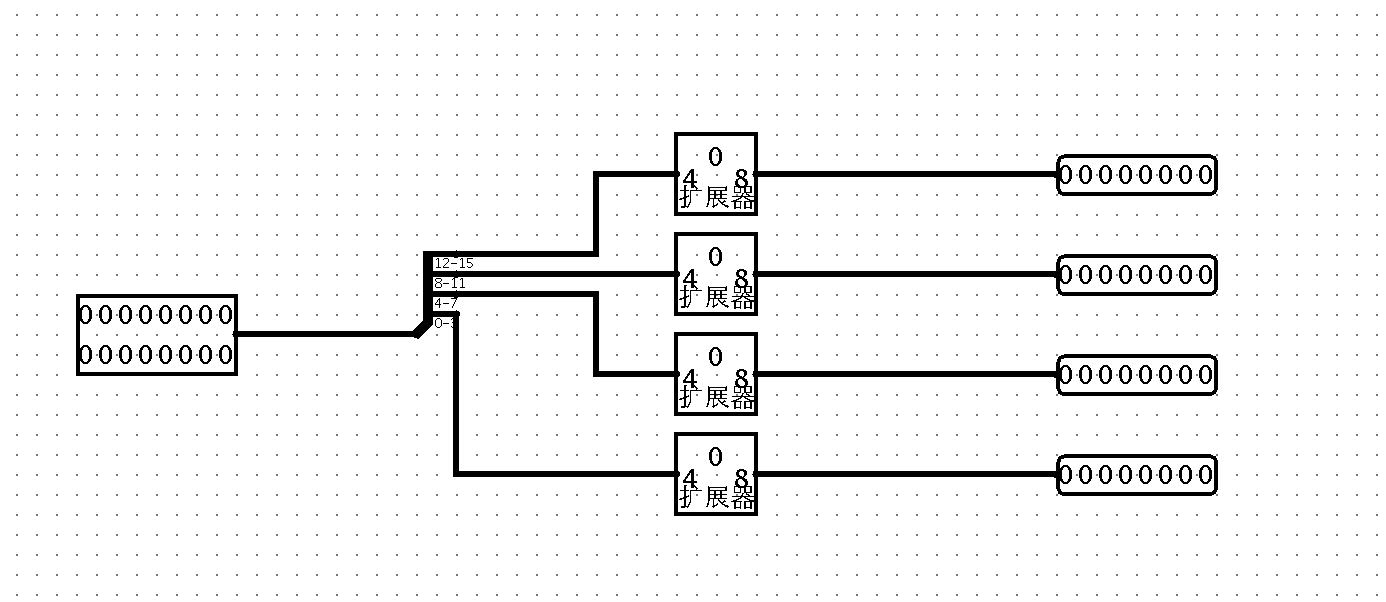
我的16位简易cpu是使用logisim 2.7.1汉化版制作和完成。经测试，可以完美运行。

**电路展示：**

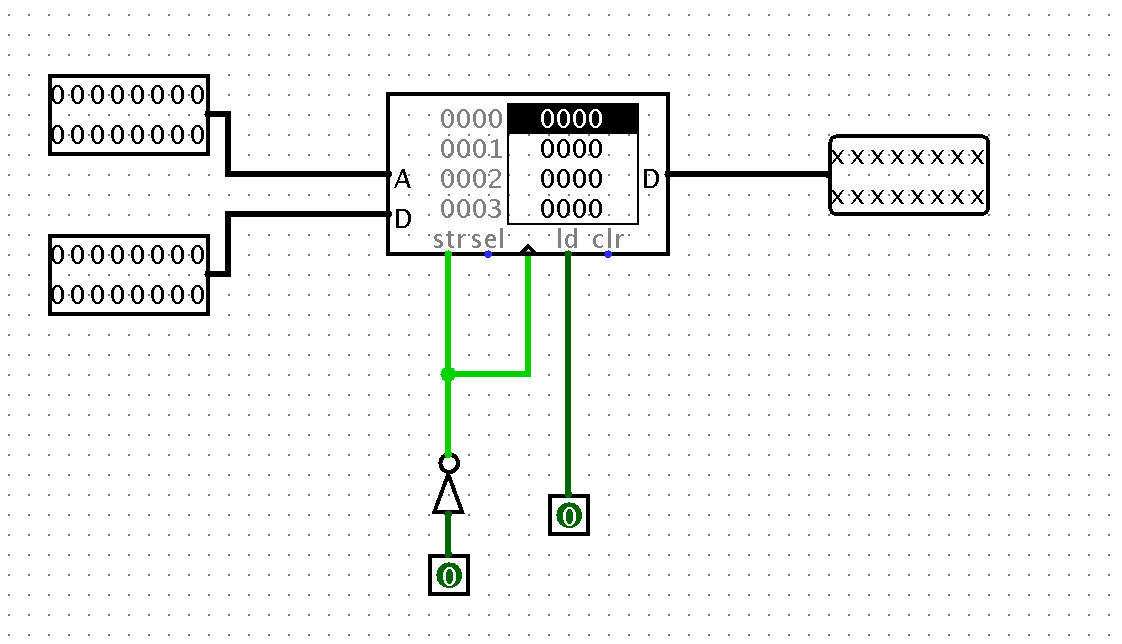
这个电路是用来判断立即数的正负的，由于负数是由二进制补码表示，符号位为1，所以先判断符号位，若为1则为负数，将补码转换为原码的方法就是原码转化为补码的方法倒过来，先-1，再将每一个位于1异或，即可。由于立即数只有8位，所以处理时只处理16位中的后8位。



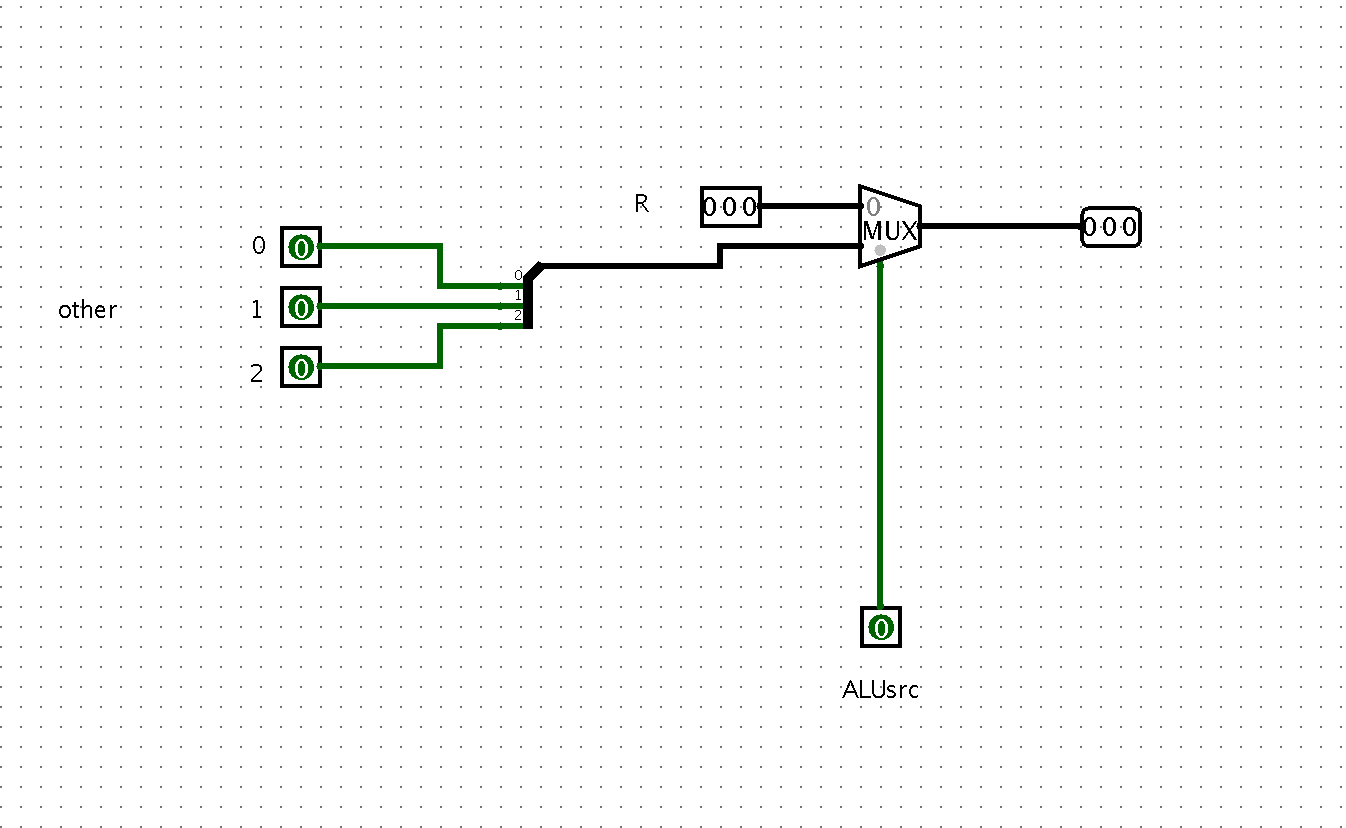
这个电路是用来展示的，将16位的数字分成4份，每相邻4位显示到一个数码管中，由于数码管是8位输入，所以用0扩展器将4位扩展为8位，连4个输出。



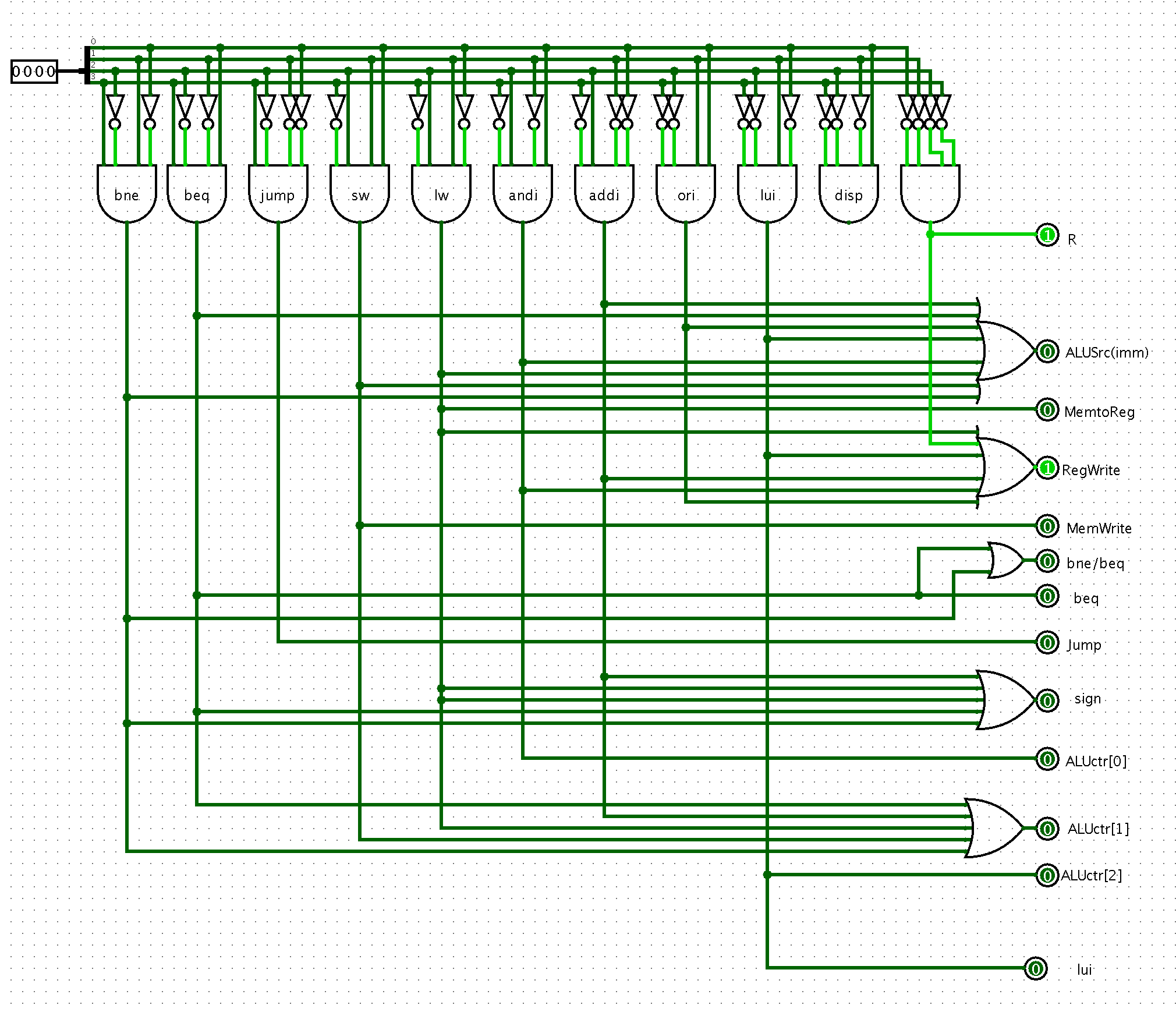
这个电路是用来操作内存的，当下边左边的输入是用来控制写内存，而下边右边的输入时用来控制输出内存内容，左边两个分别是输入时的地址和输入的内容，右边的是读内存时的输出。



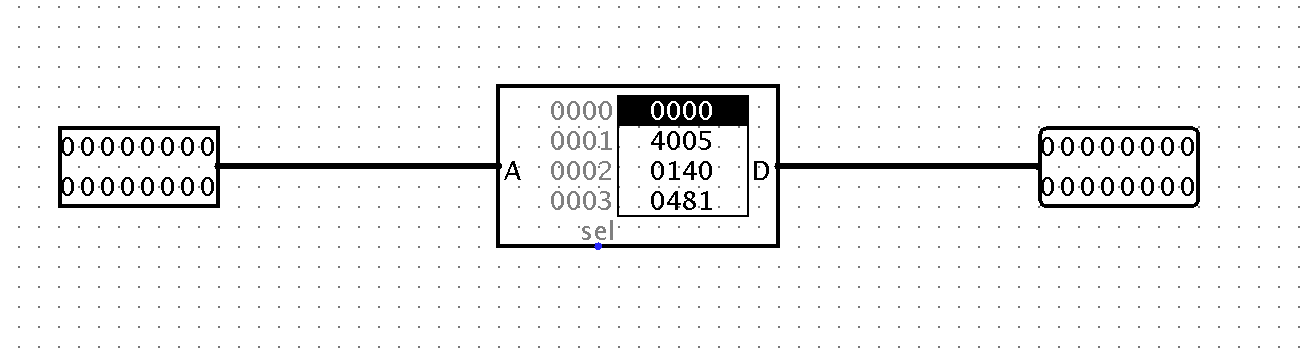
这个电路是用来控制ALU的运算的输出的，由于ALU的8个运算方式是由R格式指令的顺序完成的，所以若为R格式则直接取它的funct进行ALU运算，其他格式的则通过三位输入来控制运算方式。



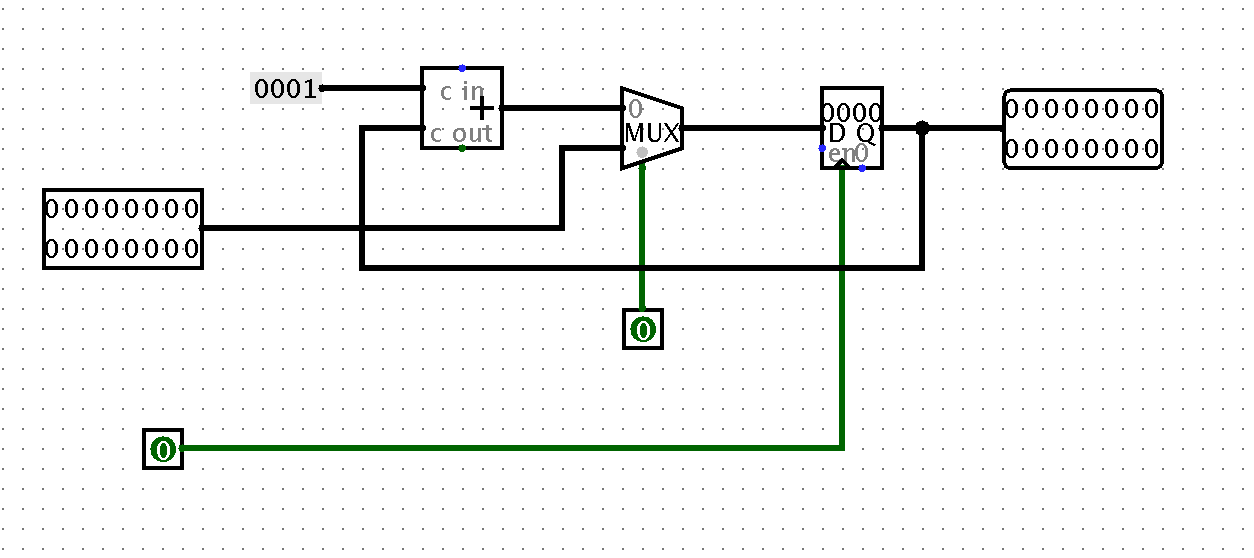
这个是用来控制的，通过12-15位的opcode来判断是什么指令,然后输出相应指令所需要对应的操作。



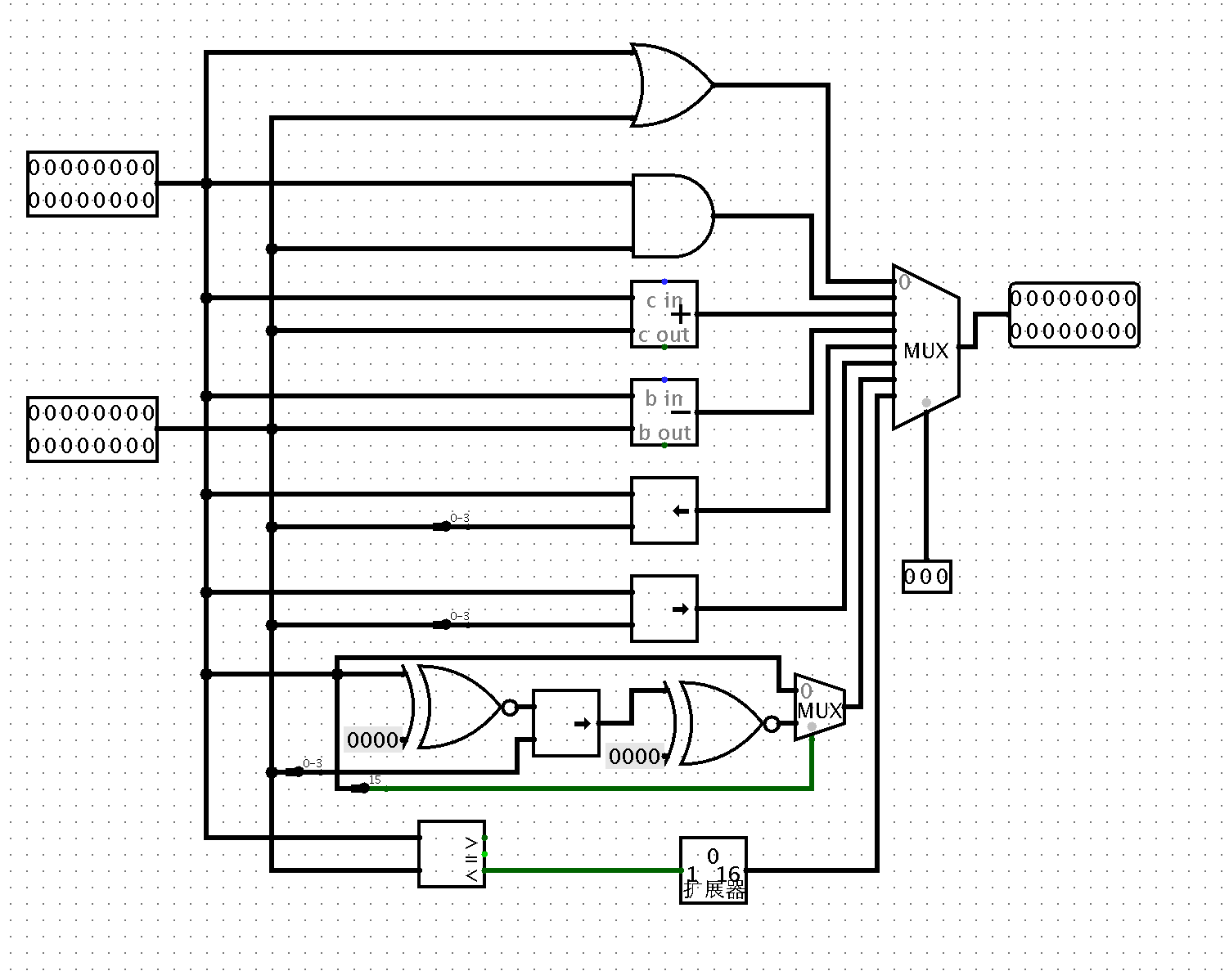
这个是ROM，左边是PC输入，右边是输出16位2进制的指令。



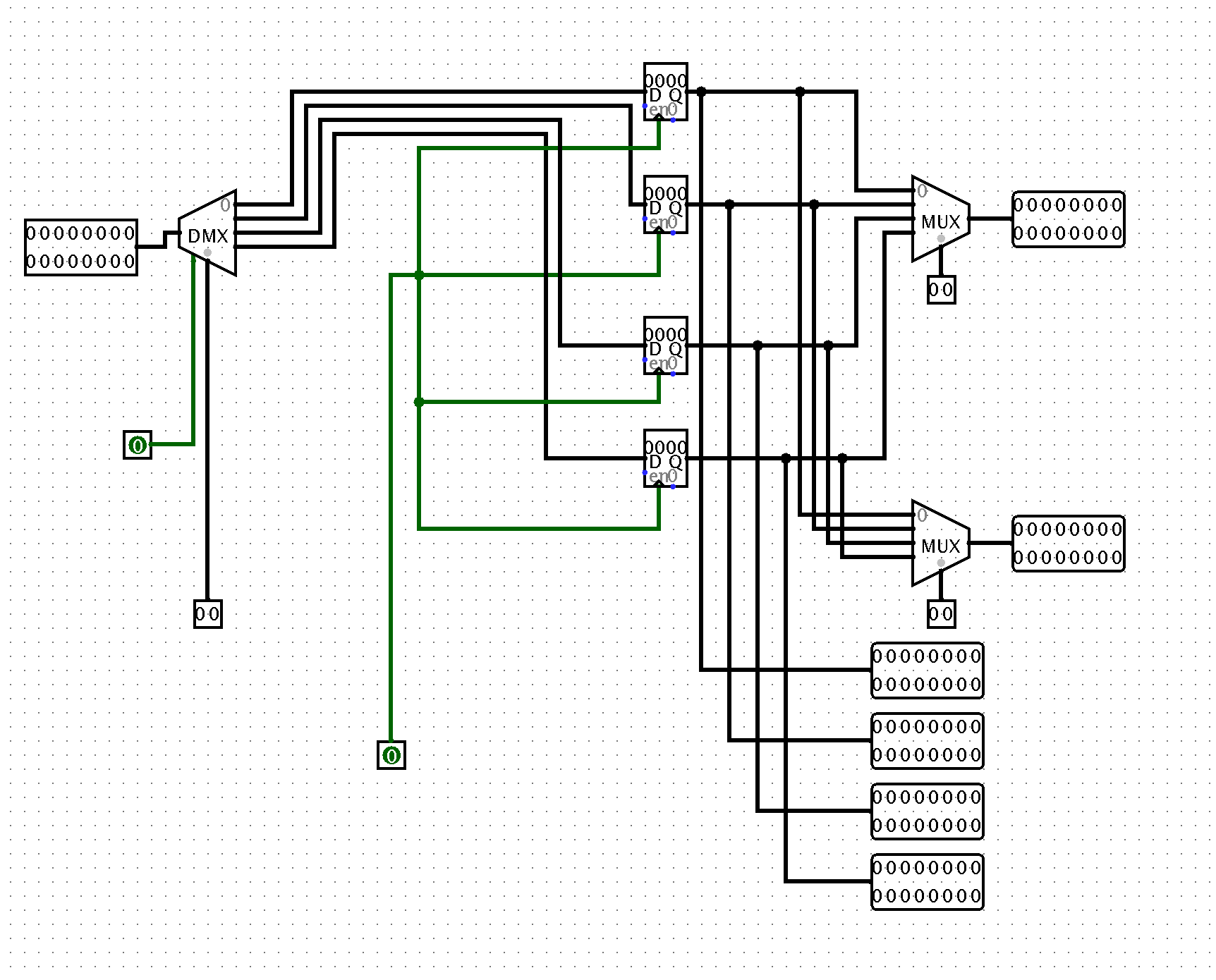
这个电路是PC的电路，若没有碰到beq、bne、j指令，则每次加1执行，反之则输入新的pc所要到的地址。



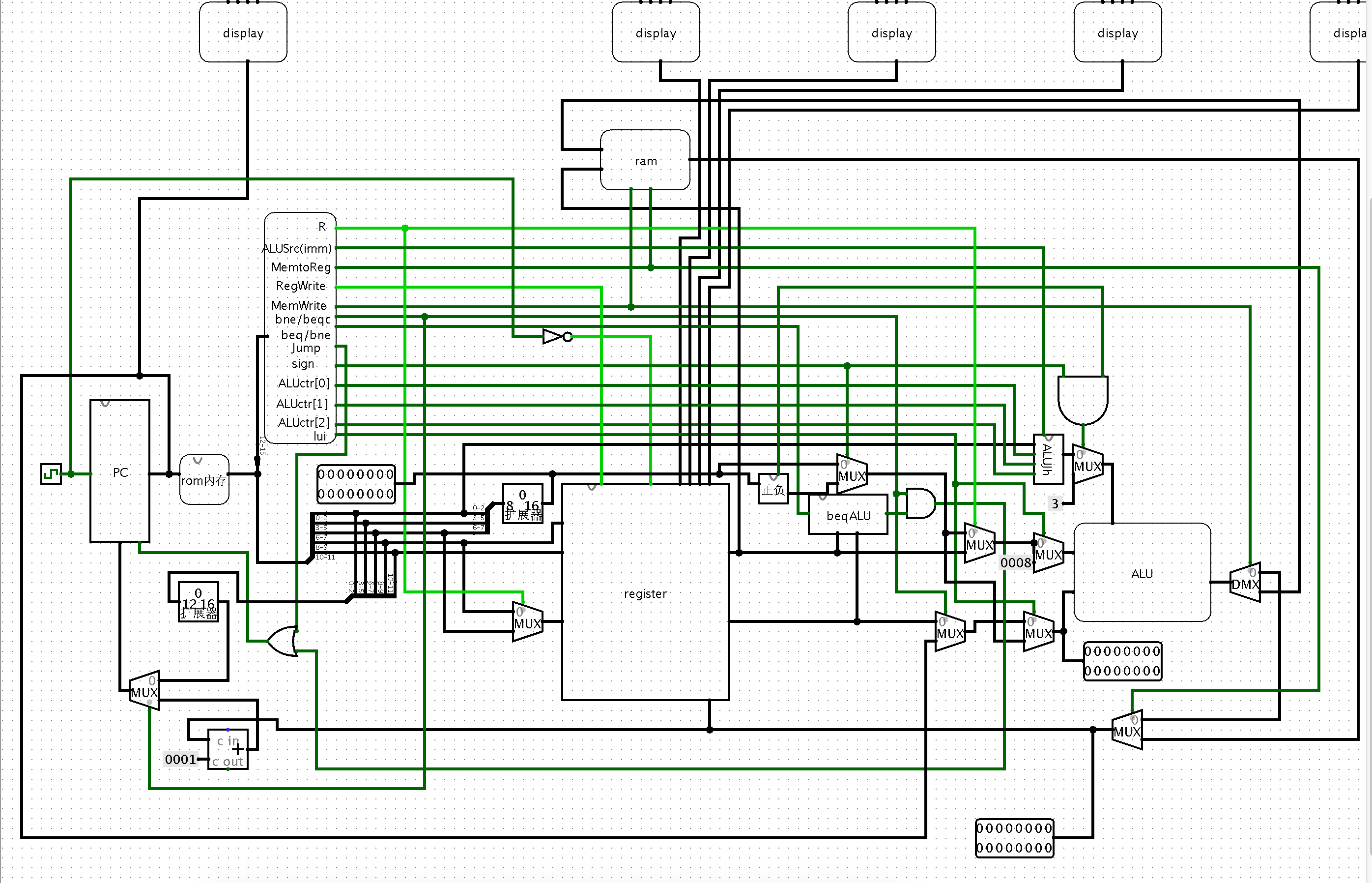
这个是ALU，用来计算的，左边输入是rs和rt或者rt和immediate，右边选择运算结果进行输出。



这个是寄存器，左边是输入，使能端通电后，通过2位二进制输入来选择所要写的寄存器，时钟信号到达时将数据存入寄存器，右边是两个选择输出寄存器的值，下方连接的是数码管用来展示寄存器的数字。



这个是主电路，上方display的上边连接的是数码管，左边是用来显示PC的值，右边4个则是分别用来显示4个寄存器的值。通过数据通道将电路联通，经测试，每条指令均可正确运行



**电路运行及调试：**

首先，写出指令，用来检测电路是否可以正常工作，然后转化为16进制存储在rom中。

指令：(每条指令下方的4位数字则是这条指令所对应4位的16进制数，左边则代表每条指令的PC值)

0001:addi $0,$0,5

0x4005

0002:or $1,$0,$1

0x 0140

0003:and $2,$1,$0

0x 0481

0004:add $3,$2,$1

0x 09c2

0005:sub $2,$1,$0

0x 0483

0006:sllv $2,$1,$1

0x 0584

0007:srl $2,$2,$1

0x 0985

0008:lui $0,128

0x 2080

0009:sra $0,$0,$1

0x 0106

000a:slt $0,$2,$1

0x 0907

000b:lui $0,85

0x 2055

000c:ori $0,$0,68

0x 3044

000d:andi $0,$0,0

0x 5000

000e:sw $2,0($0)

0x 7200

000f:lw $0,0,($0)

0x 6000

0010:j 0x0012

0x 8012

0011:空

0x 0000

0012:beq $0,$0,1

0x 9002

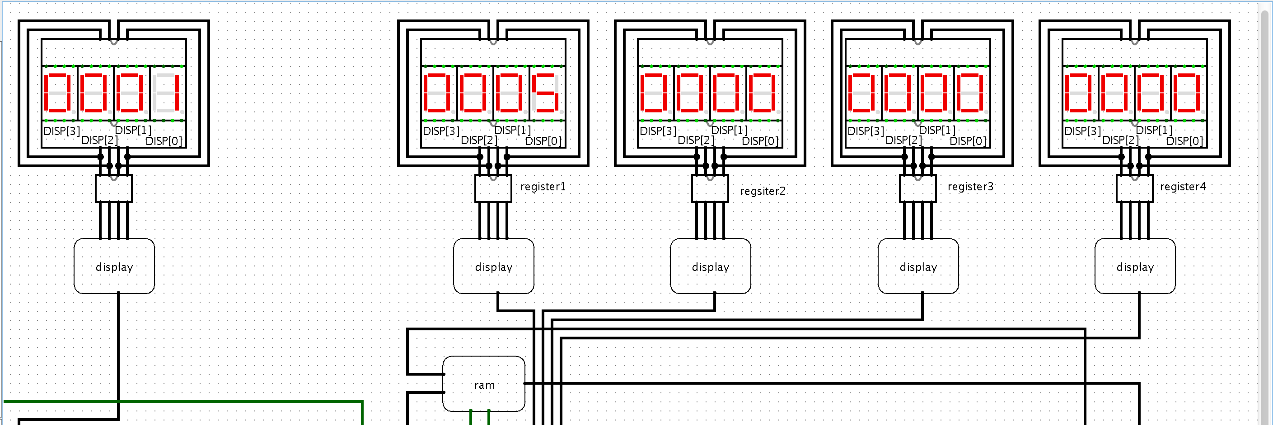
0013:空

0x 0000

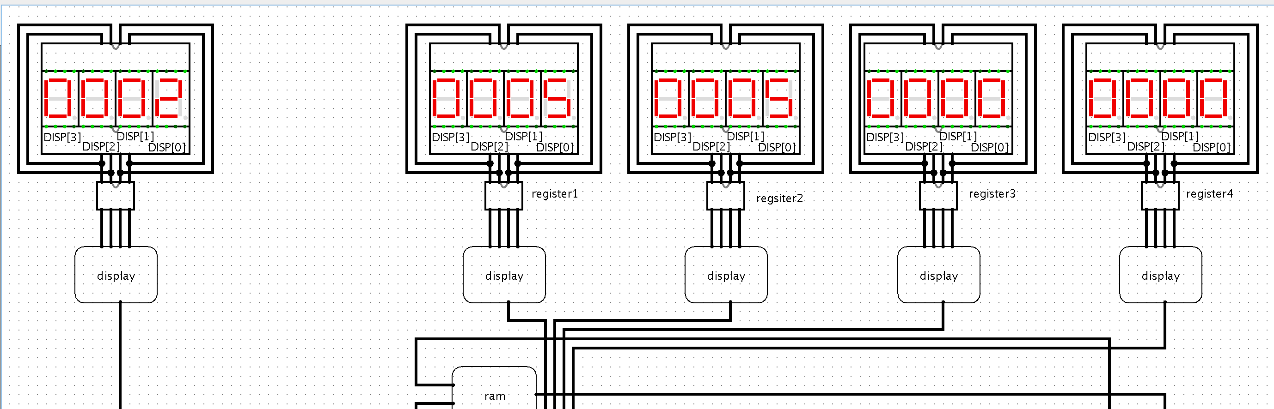
0014:bne $1,$2,-1

0x 9313

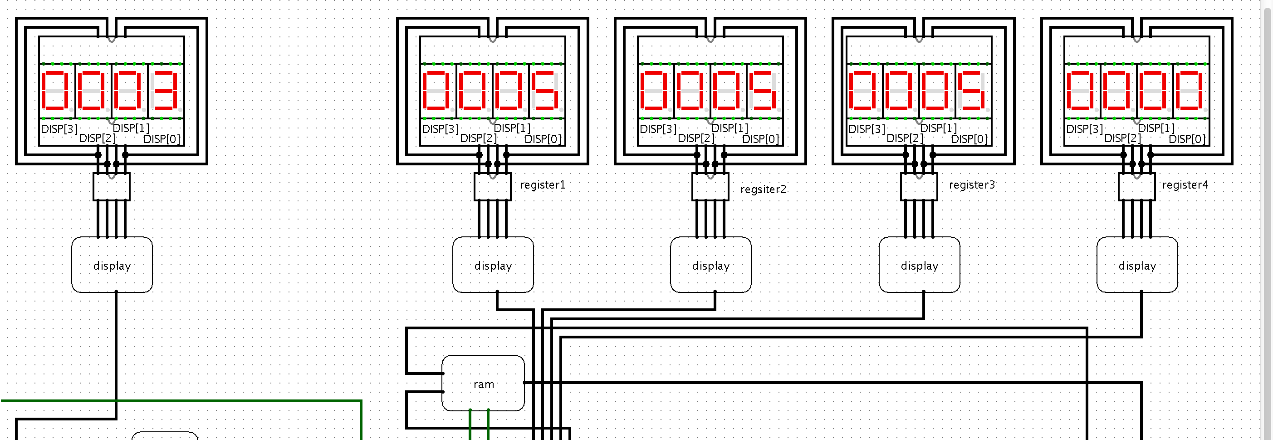
电路显示：



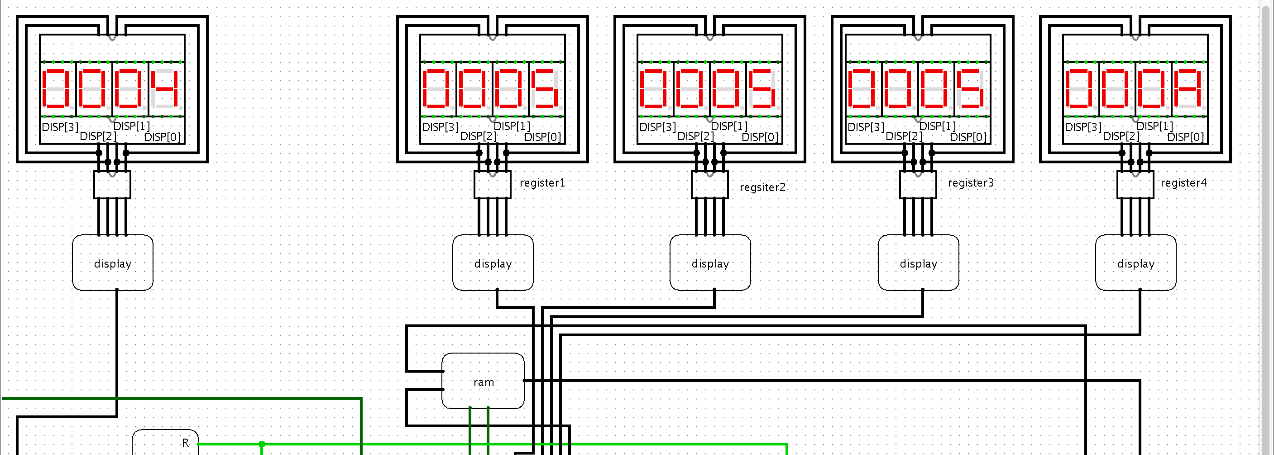
addi $0,$0,5 将0号寄存器的值+5（改变为5）



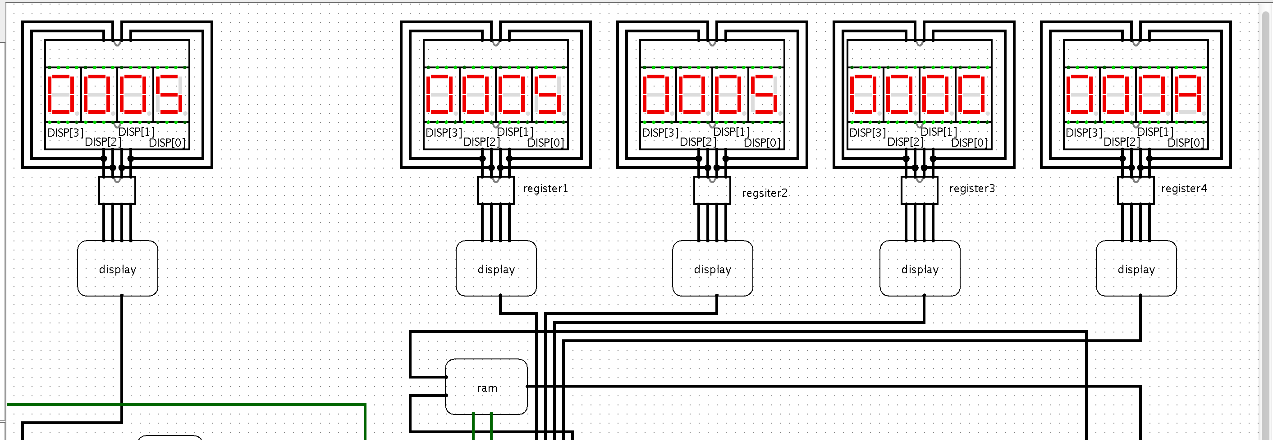
or $1,$0,$1 将1号寄存器和0号寄存器按位或，得到的结果为5，存在1号寄存器中



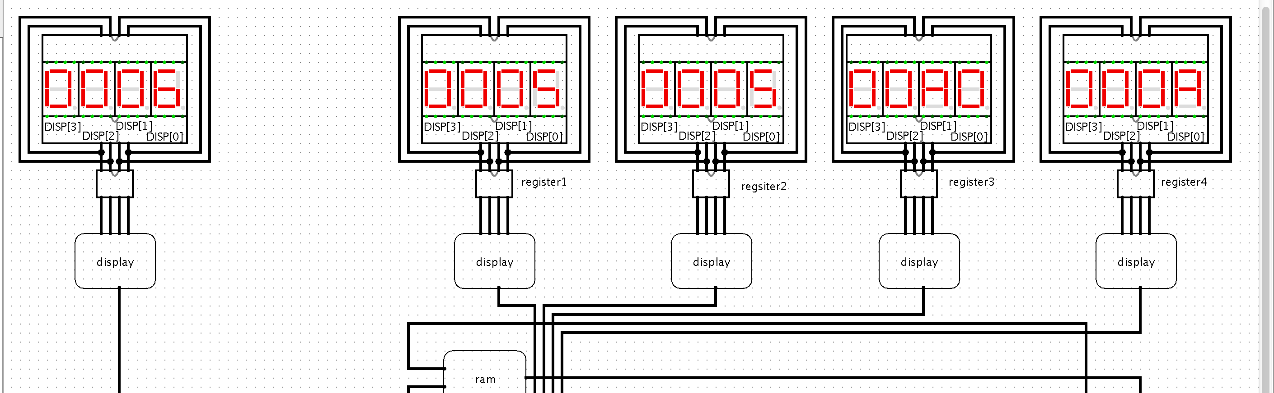
and $2,$1,$0将1号寄存器和0号寄存器按位或，得到的结果为5，存在2号寄存器中



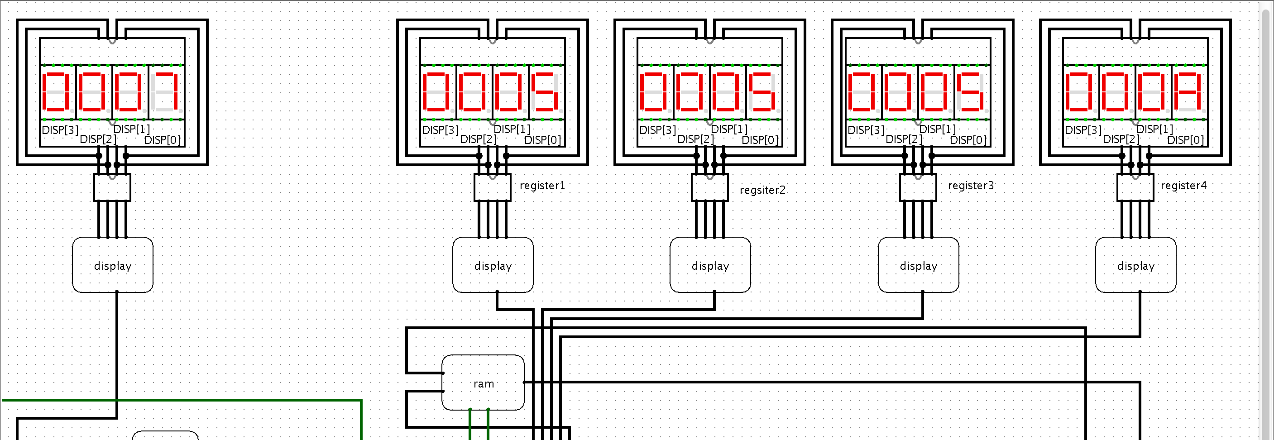
add $3,$2,$1将1号寄存器和0号寄存器的值相加，得到的结果为10，存在2号寄存器中，16进制表示为a



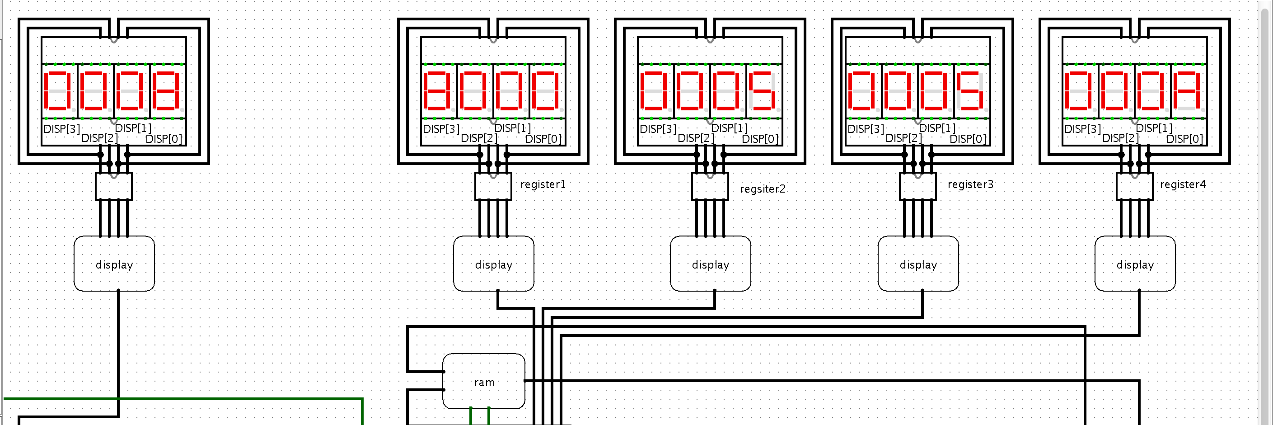
sub $2,$1,$0 将1号寄存器和0号寄存器的值相减，得到的结果为0，存在2号寄存器中，16进制表示为0



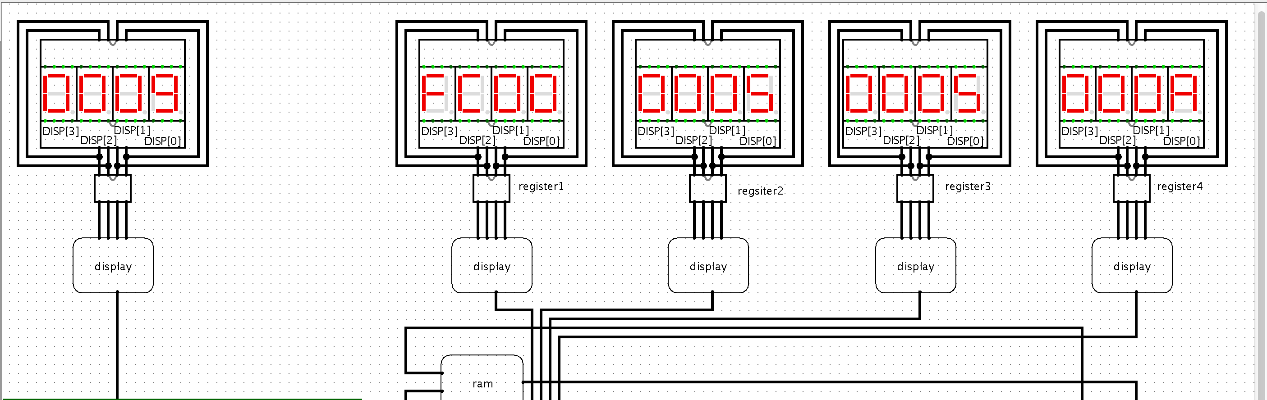
sllv $2,$1,$1 将一号寄存器的值左移5位（$1的值）存在2号寄存器中，结果为256，16进制表示为A0



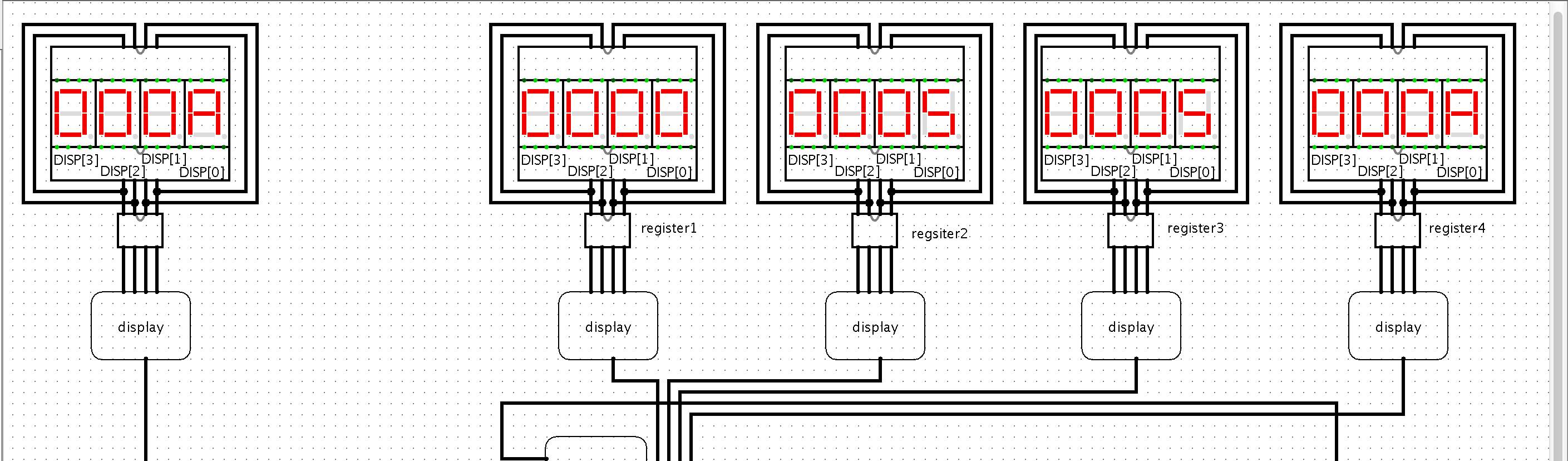
srlv $2,$2,$1将2号寄存器的值右移5位（$1的值）存在2号寄存器中，结果为5，16进制表示为5



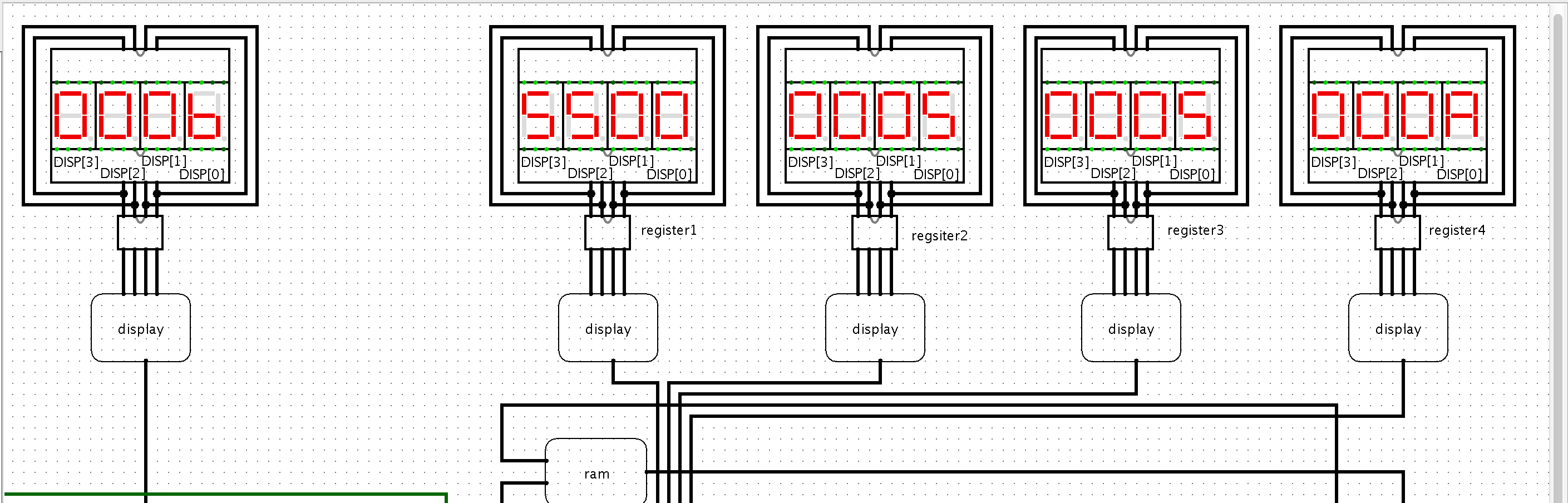
lui $0,128 将数字128转化为16进制，存在0号寄存器的高8位上，结果为16进制的8000



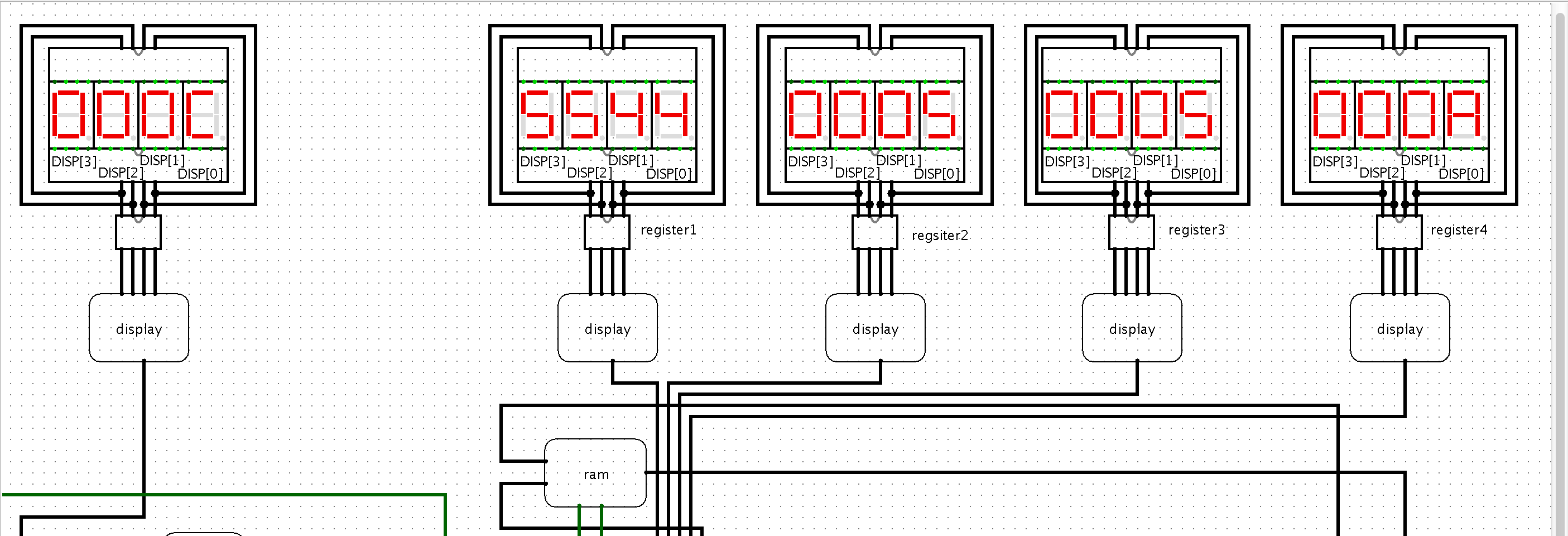
srav $0,$0,$1 将0号寄存器的值带符号向右移动5位（$1寄存器的值），结果存在$0中，16进制表示为FC00



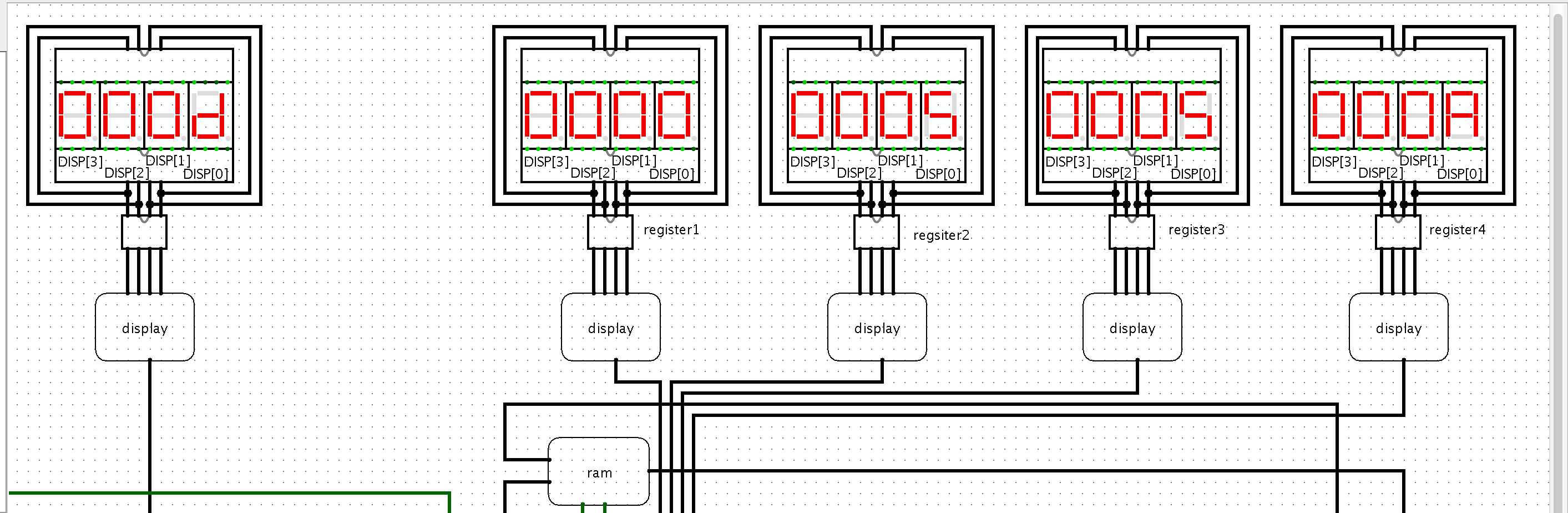
slt $0,$2,$1 判断2号寄存器的值是否小于1号寄存器，结果为0，存在0号寄存器中



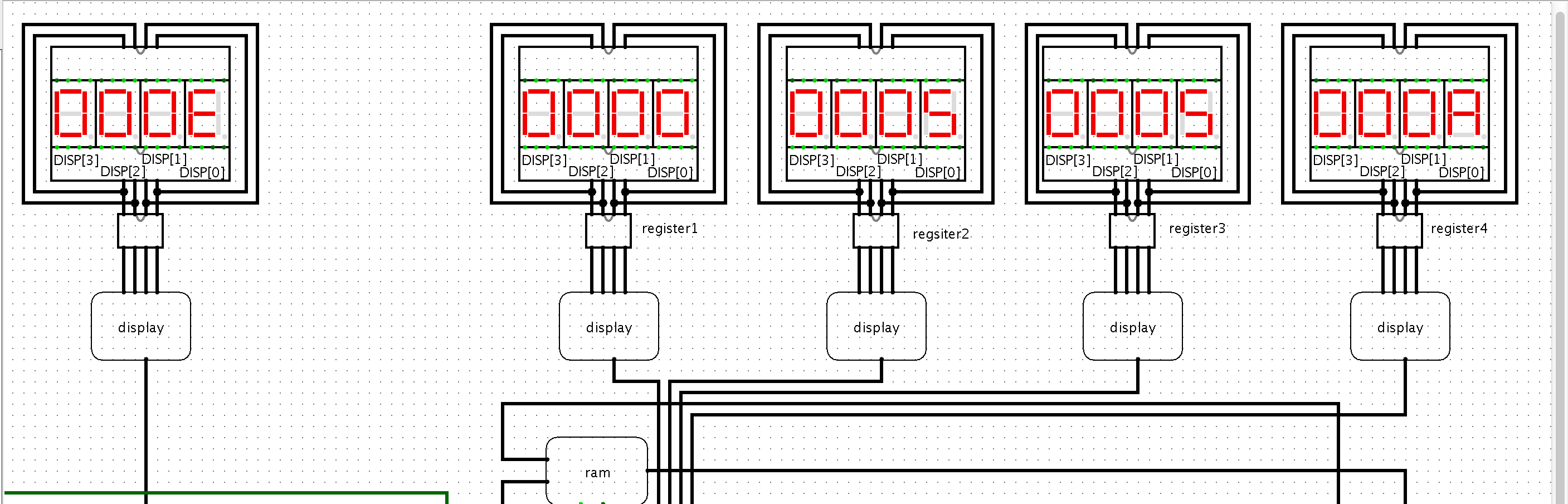
lui $0,85 将85转化为2进制，放在0号寄存器的高8位中，16进制表示为0x5500



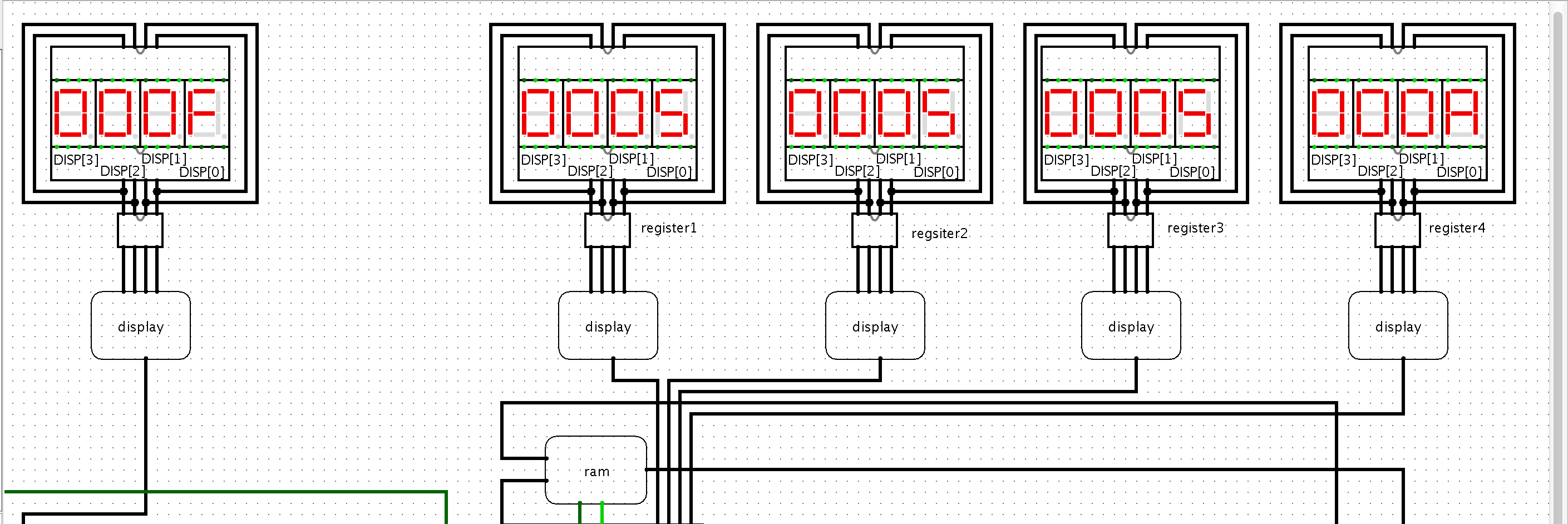
ori $0,$0,68 将0号寄存器与立即数68执行或运算，结果存在0号寄存器中，16进制表示为0x5544



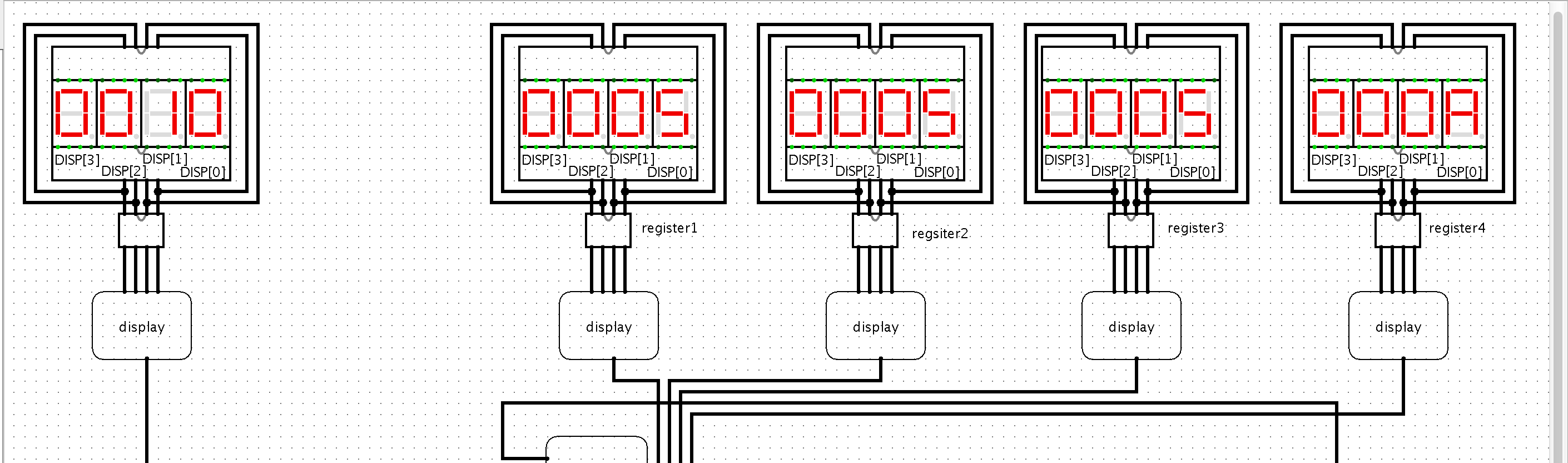
andi $0,$0,0 将0号寄存器的值与立即数0执行与运算, 结果存在0号寄存器中，16进制表示为0x0



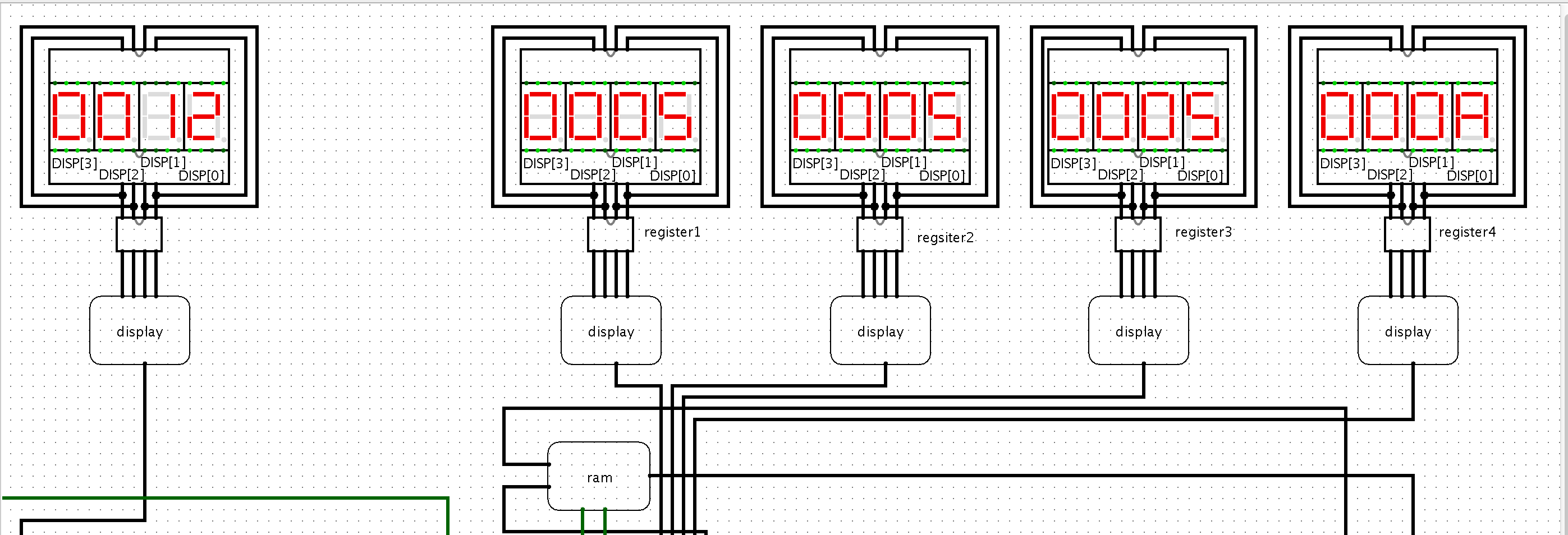
sw $2,0($0) 将2号寄存器的值，存入内存，地址为0号寄存器的值偏移量为0



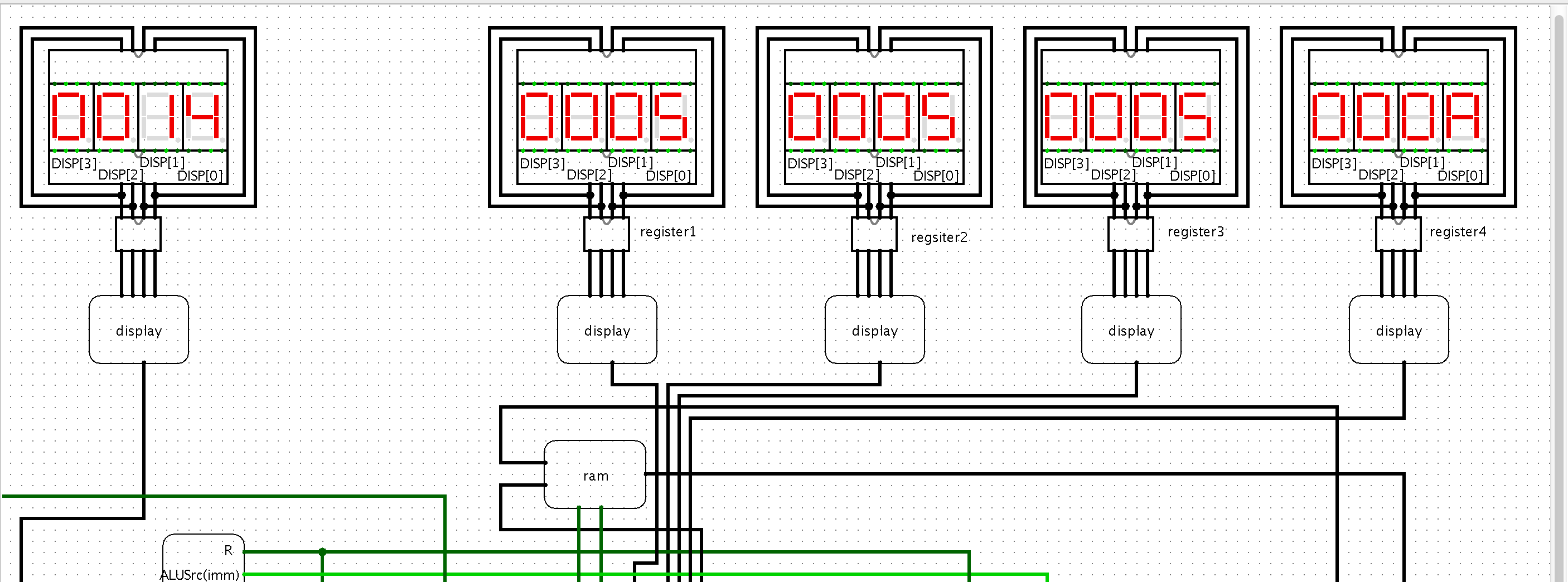
lw $0,0,($0) 将0号寄存器中值偏移量为0的地址的值，取出放到0号寄存器中，为5



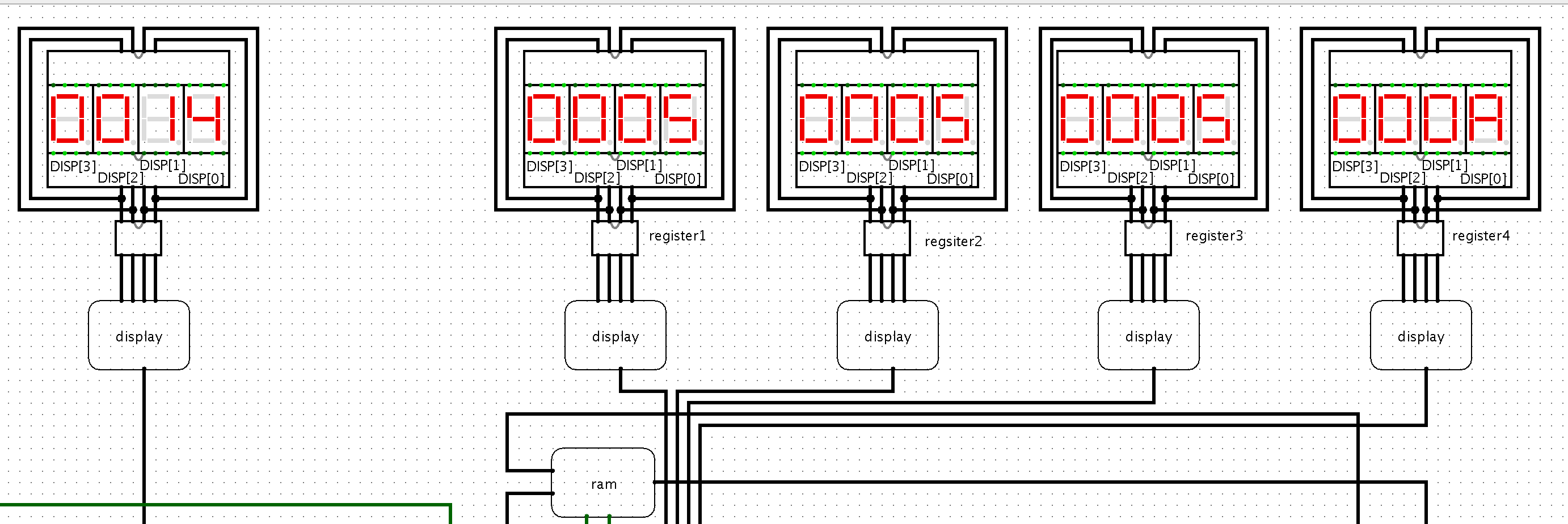
j 0x0012 直接跳转到pc中的第18行，16进制表示地址为0x0012



beq $0,$0,1 判断0号寄存器和0号寄存器的值是否相等，若相等则pc=pc+1+1



bne $1,$2,-1判断0号寄存器和0号寄存器的值是否不相等，若不相等则pc=pc+1-1，即继续执行这条指令由于条件成立，发现时钟信号前进过程中pc的值不会再改变，即不会执行下一条指令。



**至此，题目中所给的每条指令检测完毕，均与理想的结果无任何差别。**

**总 结**

通过这次cpu仿真设计，让我掌握了很多没有记住的知识，充分巩固了上课学会的知识。

身为工科学生，能力需要亲身实践来提升，通过这次课程设计的第一题，我感觉自己已经充分掌握了mips指令转化为机器码和机器码转化为mips指令的过程和具体操作，对于指令格式的充分掌握，并且知道了每一条指令对应的寄存器和pc值以及内存的变化。通过这次课程设计的第二题，我对cpu设计和各部分组成都有了详细的了解，通过cpu的制作，我不再是一味的照着书来做，而是根据题目的需求，完成相应的操作，自己制作和添加相应的小电路来完成整个大电路，在制作过程中，遇到了很多困难，比如beq和bne指令无法通过我所制作的ALU来判断，解决方法是做出另一个独立的ALU小部件来专门判断beq和bne的结果，还遇到的困难有提前做好了pc，在做j指令和beq、bne电路时发现需要的功能无法实现，结果导致我重新设计了自己的PC，当然，遇到的困难还有很多，不过通过自己的独立思考和查资料，这些问题都迎刃而解了。

在第一次上机组课程，接触mips语言时，感觉真的好难接受，很难消化课程中的内容，但是当我把第一题的mips反汇编和指令仿真两个函数写完后，发现原来难以接受和理解的东西都已化为自己所消化的知识点，已经深深地记在脑海中，充分掌握了他们。包括cpu的设计和制作也一样，由于一次接触cpu的课程，感觉接触的东西混乱而复杂，但是当cpu做好后，发现曾经学的知识点都被用上，可以连接起来组成一体，在进行调试时，先自己写汇编指令，然后翻译成机器码，转化为16进制数字后存在rom中，把知识点全部联系了起来，对掌握机组中所学的东西十分有用。

学会动手，在实践操作中掌握所学的知识点，让它们联系起来，不再是零碎的知识点，而是一个整体，可以被自己充分吸收和掌握的东西。希望自己的能力可以越来越高。