《计算机结构与组成》
课程设计报告

班级序号：

学　　号：

姓　　名：AnDJ

指导教师：

成　　绩：

中国地质大学信息工程学院

2017 年 12 月

实习题目一

<概述题目要求>

在已有代码框架下，完成一个 MIPS 汇编器的初步开发。

需求分析：

1.实现 Mars 解释出来的机器码文件(即.dump 文件)进行反汇编(即 disassembled 函数)

2.实现 Mars 解释出来的机器码文件(即.dump 文件)进行具体的逻辑上的实现，包括给出的各种命令。

<框架代码的解读>

1.sim.cpp 文件

这个文件是整个工程的 main 函数的入口，对于整个软件的如何执行都在这里显示。看的出来，这个执行文件是需要命令参数的。

/* Argument is an option, we hope one of -r, -m, -i, -d. */这一行注释对应了课设大纲里的各个参数的那个表格。之后的命令参数用 fopen 函数打开初始化了一个 FILE 指针，这个就是该软件执行功能所需要的.dump 文件，即机器码文件。最后用读取到的命令参数和文件初始化了 computer 类，并且调用了 simulator 函数。程序终止。

2.Proj.h 文件

该文件定义了 simulatedComputer 结构体，从名字中可以看出是对 mips 运行计算机，即 32 位计算机的模拟结构体，该结构体中开辟了 MAXNUMINSTRS(1024 条)存储指令的空间和 MAXNUMDATA(3072 个 word)内存空间一起作为 mips 运行时的空间。同时指定了 pc 指针用于指定 mips 运行时需要的 pc 指针。同时声明了所需要的四个重要的函数: printInfo(), simulateInstr(), disassembled(), contents()函数。

3.computer.h 文件

该文件声明了 computer 和函数 simulate。Simulate 函数的参数就是 computer。

4.computer.cpp 文件

该文件中含有最主要的信息。

```
/*
 *  Return an initialized computer with the stack pointer set to the
 *  address of the end of data memory, the remaining registers initialized
 *  to zero, and the instructions read from the given file.
 *  The other arguments all govern how the program interacts with the user.
 */
```

从注释中可以看出 computer 结构体的功能。返回一个初始化的 computer，也就是模拟一个 32 位计算机。这个 computer 的栈指针已经指向的内存的最后面，寄存器被初始化的 0，并且指令被读入内存。

```
/* stack pointer starts at end of data area and grows down */
   mips->registers[29] = 0x00400000 + (MAXNUMINSTRS+MAXNUMDATA)*4;
```

这里出现了这个所谓的虚拟地址，即 0x00400000 是初始地址，这个是对应 mars 的，因为整个软件所需要的数据文件是从 mars 的 assembly 产生的，要做到一一对应。

接下来的循环是将指令读入内存，即读到一开始开辟的内存空间

MAXNUMINSTR 大小中去。这里有一个很有意思的东西，就是取指令的时候是这样一个 for 循环：

```
Int instr = 0;
unsigned char temp = 0;
for (i = 0; i < 4; i++) {
            if (!fread(&temp, 1, 1, filein)) {
                i = -1;
                break;
            }
            instr |= temp << (i << 3);
        }
```

即每次是拿一个 char 类型的空间取出 32 位机器码的 8 个拿到 instr 中去，但是先拿出来的 8 位反而是存往 32 位的 instr 中的靠近后面的位置，即会是这样的情况，假如一条汇编语言正常翻译为 0xaabbccdd，则 mars 产生的.dump 文件中的顺序是 0xddccbbaa，然后在该框架代码中又将其拼接而成 0xaabbccdd 的正常顺序。这个也是十分重要的。以上为 computer 的初始化函数。

Simulate 函数。该函数就是 main 函数中执行的最后一个函数。这个里面的逻辑是整个该软件的核心。首先初始化了 pc 指针为 0x00400000,然后是一个循环读取之前存入的 instruction 进行操作。

该循环的主体是，读取一条指令，（使用 content 函数，该函数在下面实现，用于取出虚拟的内存地址相对应的真正的物理内存地址中的内容），将现在 pc 指向的内存地址打印出来，然后打印 disassembled 函数将该条指令相对应的反汇编结果，接着调用 simulateInstr 函数执行该条指令，最后将执行完该指令后寄存器和内存的状态变化信息打印出来。这就指定了最重要的两个函数:disassembled 和 simulateInstr.

printInfo 函数。该函数用来打印一条指令执行过后寄存器和内存的存储变化。通过注释可以知道，这里主要是通过两个变量来确定前两者的变化情况，一个是 changeReg,即 change register，一个是 changeMem,即 change memory，changeReg 的值就是变化的寄存器标号，changeMem 的值就是存储变化的相应内存地址。如果两者为-1,则是没有发生变化。这两个变量是十分重要的。

Contents 函数。Return the contents of memory at the given address. 该函数用来返回对应虚拟地址对应的具体的内容。这里有一些对于虚拟内存地址和真实物理地址的转换是需要注意和借鉴的。

```
int index = (addr-0x00400000)/4;
    if (((addr & 0x3) != 0) || (index < 0) || (index >= (MAXNUMINSTRS+MAXNUMDATA))) {
        printf("Invalid Address: %8.8x", addr);
        exit(0);
    }
```

真实内存的转换其实就是转换到 memory 数组的正确下标。Memory 数组是一个 int 类型的数组，换句话说，就是一个 32 位数据为一个单位长的数组，而 mips 是以字节（8 位）进行编址的，所以要除以 4。同时，这要保证地址是 4 的倍数，所以有 addr& 0x3 ==0;并且要在内存中对应相对应的位置，

所以有之后的两个条件。

5. proj1. cpp

这个文件就是需要完成自己的逻辑的文件。该文件中要完成两个核心的函数，disassembled 和 simulateInstr.

<设计思路>

1.封装 disassembled 函数，

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

该函数在读取一条指令后被调用，输入是一条 32 位指令和 pc 指针，输出是一条反汇编之后的汇编指令，类型为 char*, 供外层 print 函数打印。抽象该过程为： 0x12345678 ---> add $1, $2, $3

指令格式有 R 格式、I 格式和 J 格式。实现二重封装，封装三个函数 R_Instruction, I_Instruction, J_Instruction 分别解析三种对应的指令，通过 opcode 进行区分。针对每一种指令再次封装函数，在上述函数中分情况调用。R 格式指令通过 funct 进行区分，J 格式指令只有两种，通过指定 opcode 进行区分，I 格式指令通过 opcode 进行区分。这样做的好处是，便于解决某个指令出现问题后能方便修复和维护。就是有点冗余。

这里传入的 PC 指针是为了方便计算跳转指令，这在跳转相关指令中需要用到。主要操作应该是字符串的拼接并返回。

2.封装 simulateInstr 函数

void simulateInstr (Computer mips, unsigned int instr,

int *changedReg, int *changedMem)

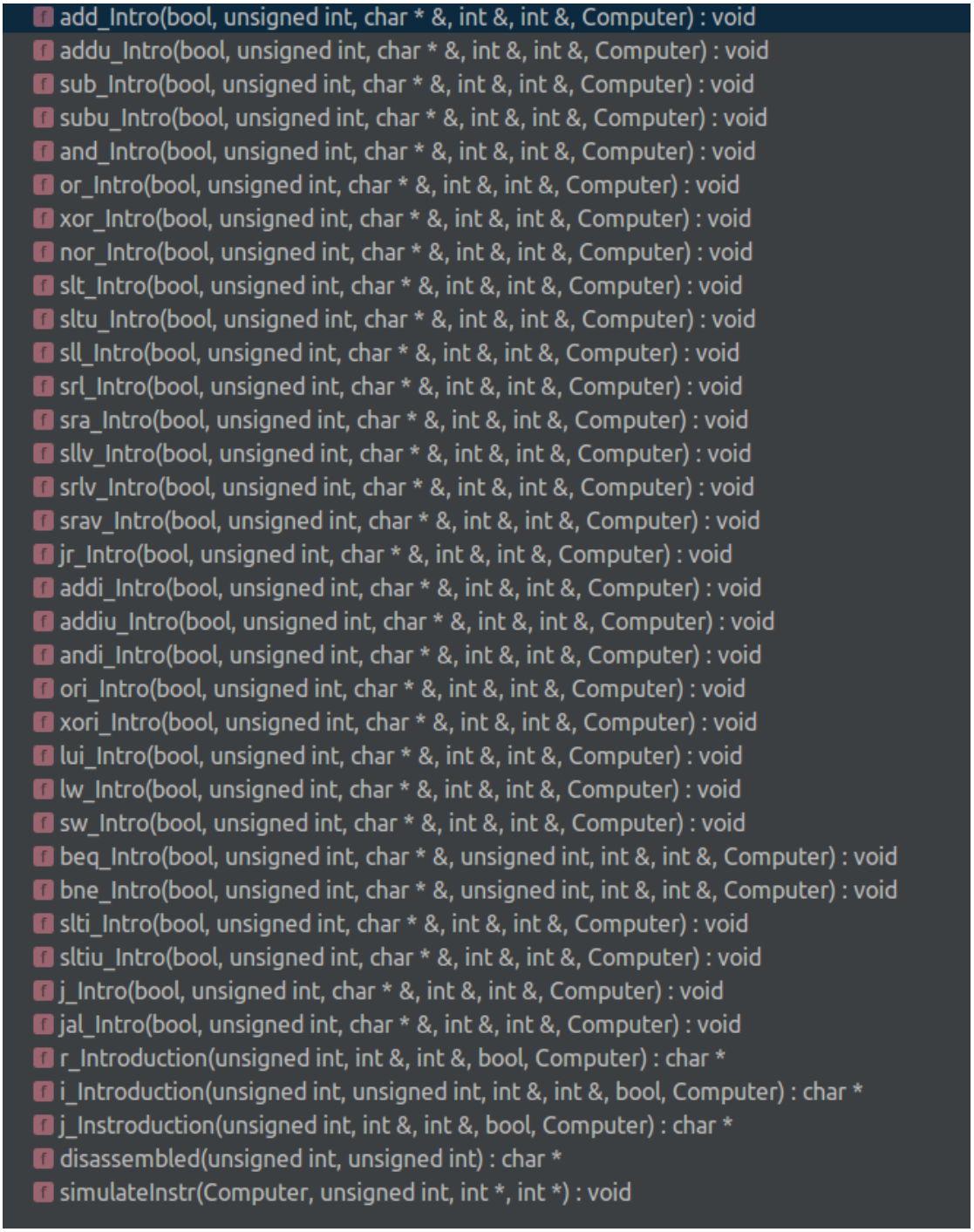
该函数执行于读取一条指令并反汇编输出后，主要作用是真正处理相关的逻辑。传入值为 mips, 指令 instr, 以及两个反馈寄存器和内存变化的两个参数 changedReg, changedMem。执行该函数产生实际的相关指令的效果，并且返回对应的 changedReg, changedMem.

这里为了降低冗余，我同样将三种不同格式的指令封装在三个不同的函数中，同时每一条指令封装一个函数，在三种不同的函数中和 disassembled 函数中一样的策略进行区分。这样的话，我将 disassembled 函数和 simulateInstr 函数的子函数全部抽象为一个函数，即 add 的反汇编函数和 add 的操作函数放在一个函数中，给该函数中分两种实现模式，mode 为 true 实现 disassembled 功能，mode 为 false 实现 simulateInstr 功能，这样也便于维护。

重点需要注意的地方就是两个返回值：changedReg 和 changedMem。

<实现和测试>

代码整体结构：



看起来有点冗余，但是方便维护。

R 格式指令示例实现：add

```
R-type
add $1, $2, $3    $1=$2+$3
    rd  rs  rt    rd=rs+rt
id add_Intro(bool mode, unsigned int introduction, char*& outChar,
             int &changedReg,int &changedMem,Computer mips){
  int rs=(introduction<<6>>27);
  int rt=introduction<<11>>27;
  int rd=introduction<<16>>27;
  if(mode){
      strcat(outChar,"add\t");
      char registerName[3];
      strcat(outChar,"$");
      sprintf(registerName,"%d",rd);
      strcat(outChar,registerName);
      strcat(outChar,", $");
      sprintf(registerName,"%d",rs);
      strcat(outChar,registerName);
      strcat(outChar,", $");
      sprintf(registerName,"%d",rt);
      strcat(outChar,registerName);
  }else{
      mips->registers[rd]=mips->registers[rs]+mips->registers[rt];
      int temp=mips->registers[rd];
      if(temp!=mips->registers[rd])
          changedReg=rd;
      else
          changedReg=-1;
      changedMem=-1;
      mips->pc+=4;
  }
```

Mode 为 true 执行反汇编操作，返回值为 outChar，通过位操作拿到相对应的 rs,rt,rd。然后拼接字符串。Mode 为 false 执行逻辑操作，通过 mips 指针操作相应的寄存器进行相加计算。如果有 register 的值发生变化则将其序号返回给 changedReg.内存无变化。最后，pc 指针自增 4 保证 simulateInstr 函数外层循环能够继续下去。

add 与 addu 的区别就是在相加的时候 addu 为

```
mips->registers[rd]=(unsigned int)mips->registers[rs]+(unsigned int)mips->registers[rt];
```

Sub subu and or xor nor slt sltu sll 等函数均为此方式实现。

立即数指令示例：addi

```
//I-type
void addi_Intro(bool mode, unsigned int introduction, char*& outChar
     ,int &changedReg,int &changedMem,Computer mips){
    int rs=(introduction<<6>>27);
    int rt=introduction<<11>>27;
    int immediate=introduction<<16>>16;
    if(immediate>>15){
        immediate|=(-1>>16<<16);
    }
    if(mode){
        strcat(outChar,"addi\t");

        char registerName[3];
        char immediateName[10];
        strcat(outChar,"$");
        sprintf(registerName,"%d",rt);
        strcat(outChar,registerName);
        strcat(outChar,", $");
        sprintf(registerName,"%d",rs);
        strcat(outChar,registerName);
        strcat(outChar,", ");
        sprintf(immediateName,"%d",immediate);
        strcat(outChar,immediateName);
    }else{
        mips->registers[rt]=mips->registers[rs]+immediate;
        changedReg=rt;
        changedMem=-1;
        mips->pc+=4;
    }
}
```

Lw 和 sw 两条指令涉及内存的计算，利用到了 Content 函数，代码示例：

Sw

```
void sw_Intro(bool mode, unsigned int introduction, char*& outChar ,int &changedReg,int
&changedMem,Computer mips){
    int rs=(introduction<<6>>27);
    int rt=introduction<<11>>27;
    int immediate=introduction<<16>>16;
    if(immediate>>15){
        immediate|=(-1>>16<<16);
    }
    if(mode){
        strcat(outChar,"sw\t");


        char registerName[3];
        char immediateName[10];
        strcat(outChar,"$");
        sprintf(registerName,"%d",rt);
        strcat(outChar,registerName);
        strcat(outChar,", ");
```

```
            sprintf(immediateName,"%d",immediate);
            strcat(outChar,immediateName);

            strcat(outChar,"($");
            sprintf(registerName,"%d",rs);
            strcat(outChar,registerName);
            strcat(outChar,")");
        }else{
            int index = (mips->registers[rs]+immediate-0x00400000)/4;
            if ((((mips->registers[rs]+immediate) & 0x3) != 0) || (index < 0) || (index >=
(MAXNUMINSTRS+MAXNUMDATA))) {
                printf("Invalid Address: %8.8x", mips->registers[rs]);
                exit(0);
            }
            mips->memory[index]=mips->registers[rt];
            changedReg=-1;
            changedMem=mips->registers[rs]+immediate;
            mips->pc+=4;
        }
}
```

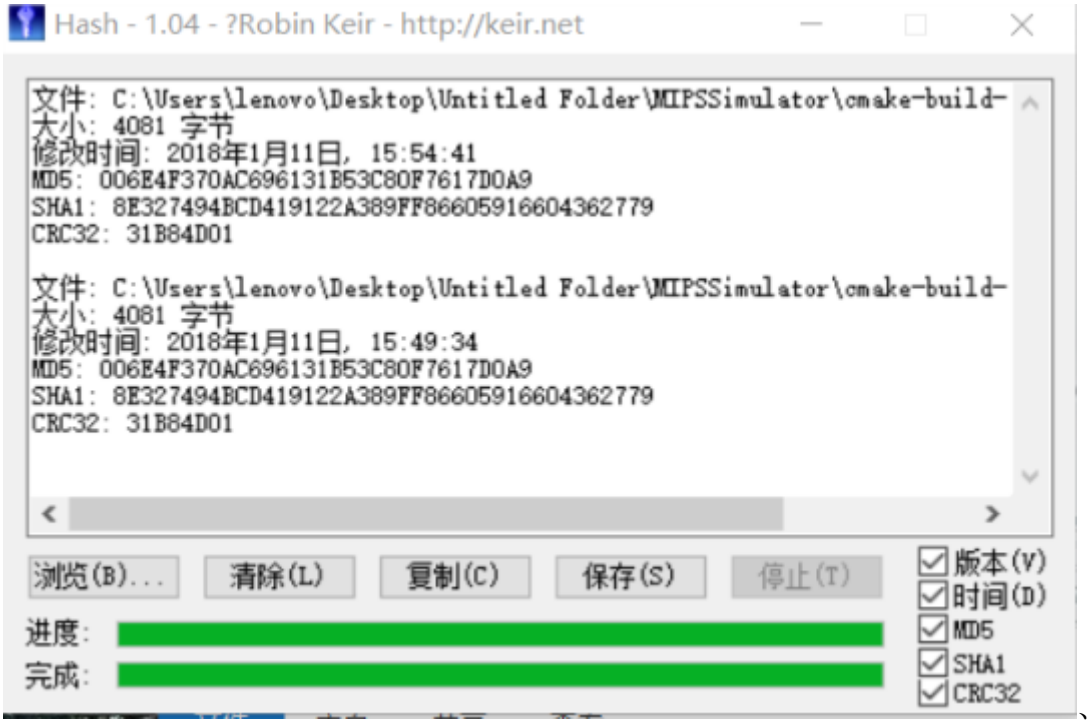
LW

```
void lw_Intro(bool mode, unsigned int introduction, char*& outChar ,
                    int &changedReg,int &changedMem,Computer mips){
    int rs=(introduction<<6>>27);
    int rt=introduction<<11>>27;
    int immediate=introduction<<16>>16;
    if(immediate>>15){
            immediate|=(-1>>16<<16);
    }
    if(mode){
            strcat(outChar,"lw\t");


            char registerName[3];
            char immediateName[10];
            strcat(outChar,"$");
            sprintf(registerName,"%d",rt);
            strcat(outChar,registerName);
            strcat(outChar,", ");
            sprintf(immediateName,"%d",immediate);
            strcat(outChar,immediateName);

            strcat(outChar,"($");
            sprintf(registerName,"%d",rs);
            strcat(outChar,registerName);
```

```
            strcat(outChar,")");

        }else{
            mips->registers[rt]=contents(mips,mips->registers[rs]+immediate);
            changedReg=rt;
            changedMem=-1;
            mips->pc+=4;
        }
}
```

跳转指令 J 和 Jal 直接把地址赋给 pc 即可，这里不再粘贴。

<测试>



最终的输出文件 md5 匹配完全一致。

实习题目二

<概述题目要求>

需求：使用 Logisim 来创建一个 16-位单时钟周期 CPU.

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)。

以半字(16 位)为单位对内存编址。

Instruction Set Architecture(ISA)

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	5	srlv: $rd = $rs >> $rt
3	sub: $rd = $rs - $rt	6	srav: $rd = $rs >> $rt
4	sllv: $rd = $rs << $rt	7	slt: $rd = ($rs < $rt) ? 1 : 0

测试数据：编写两个测试用例并在 CPU 上测试。

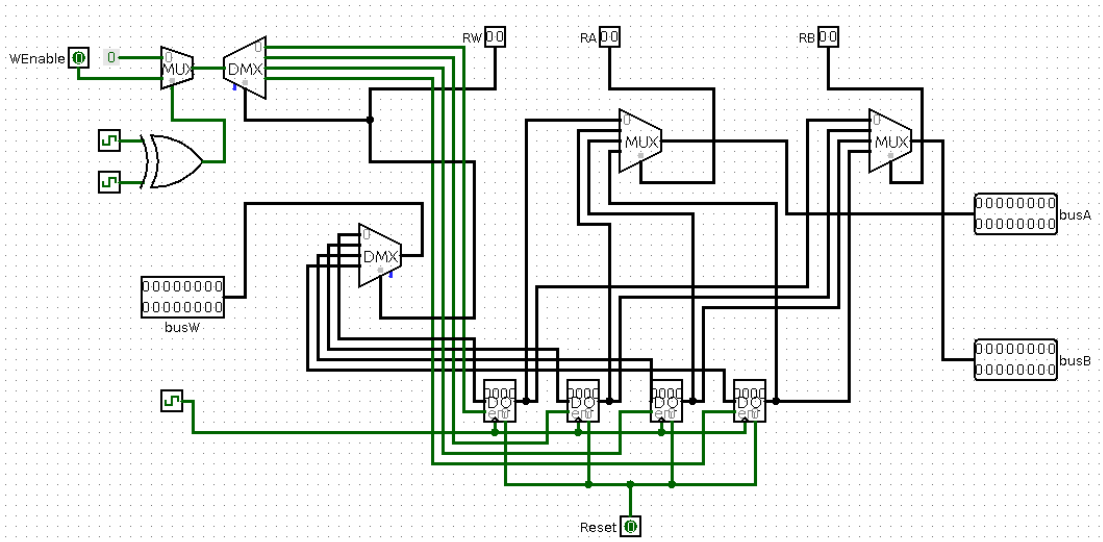
<设计思路>

Datapath:

1. 设计寄存器组

寄存器文件中有 4 个寄存器，分别编号为$r0, $r1, $r2, $r3。给该寄存器文件设置接线，RW, RA, RB 接线分别对应相应的寄存器标号，RW 为写入操作时对应的寄存器，RA 和 RB 对应输出总线 busA 和 busB，用于向外部输出相应寄存器的数据。Reset 接线用于清空寄存器中的数据，busW 为输入总线，将 16 位数传给 RW 指定寄存器。WEnable 为该寄存器组的写使能。

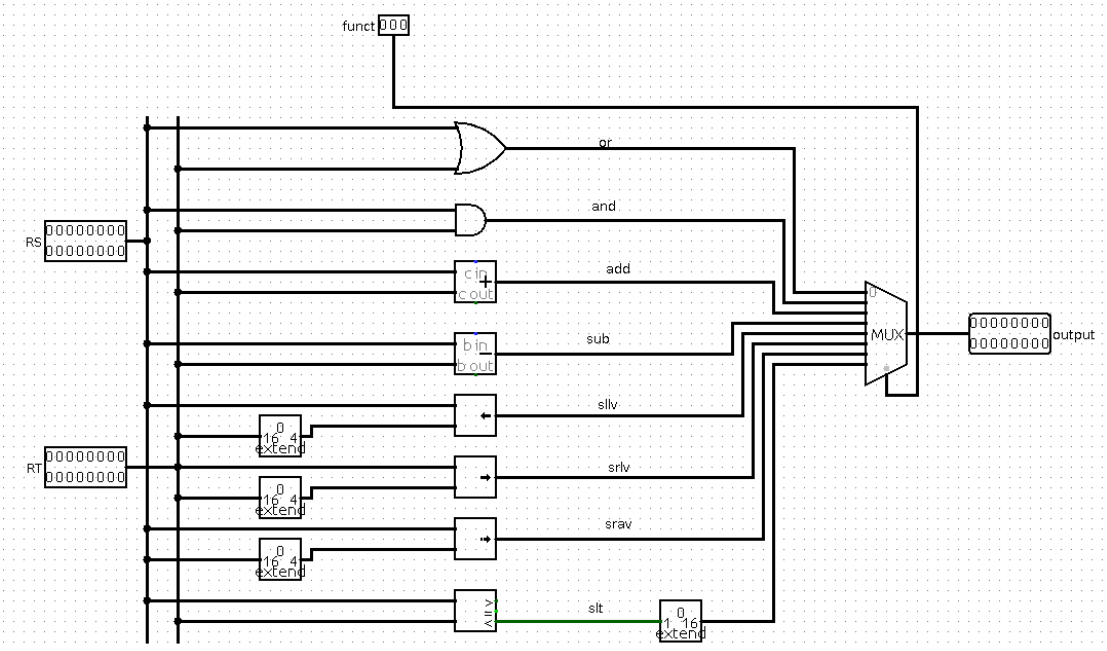
该寄存器文件中存在多个多路选择器，用于传导相应的信号。WEnable 处的多路选择器用于决定哪一个寄存器可以写入，busW 处的多路选择器用于决定哪一个寄存器接受该数据，RA,RB 处的用于输出相应寄存器的数据。



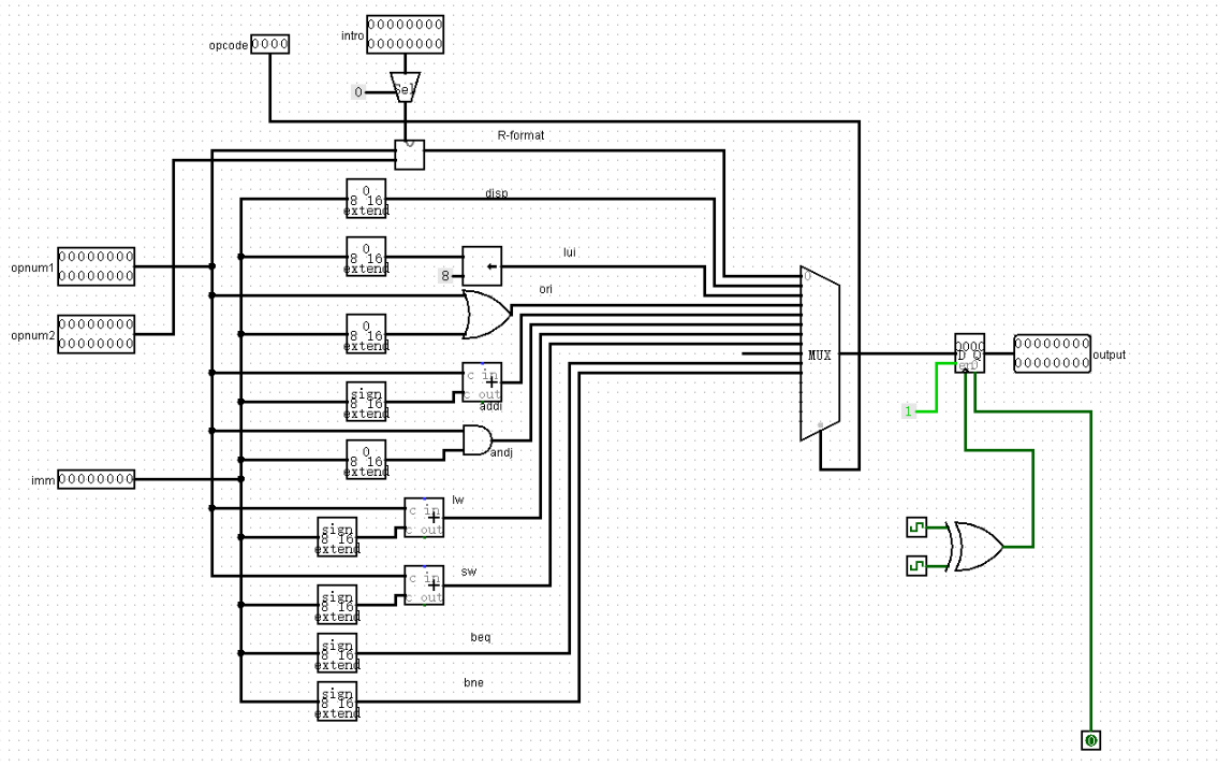
2.设计 ALU

操作中的指令有大概三种，即 R-format I-format J-format。其中前两者是需要 ALU 单元的，那么我先组件 ALU_R，其次利用 ALU_R 组织完整的 ALU。

ALU_R

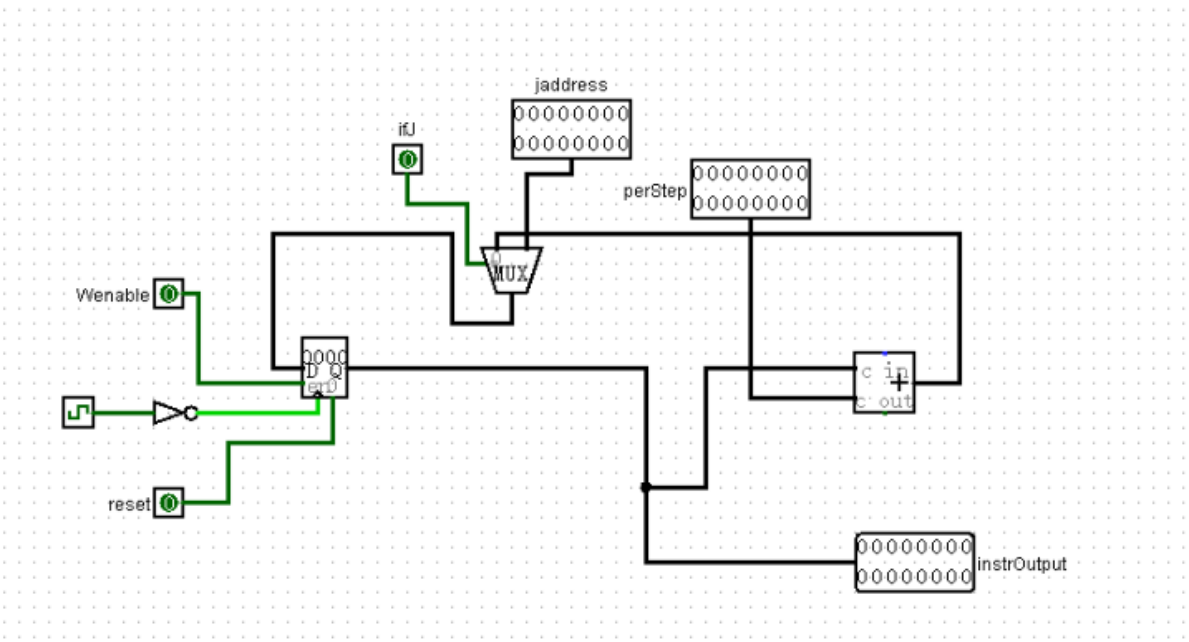


ALU



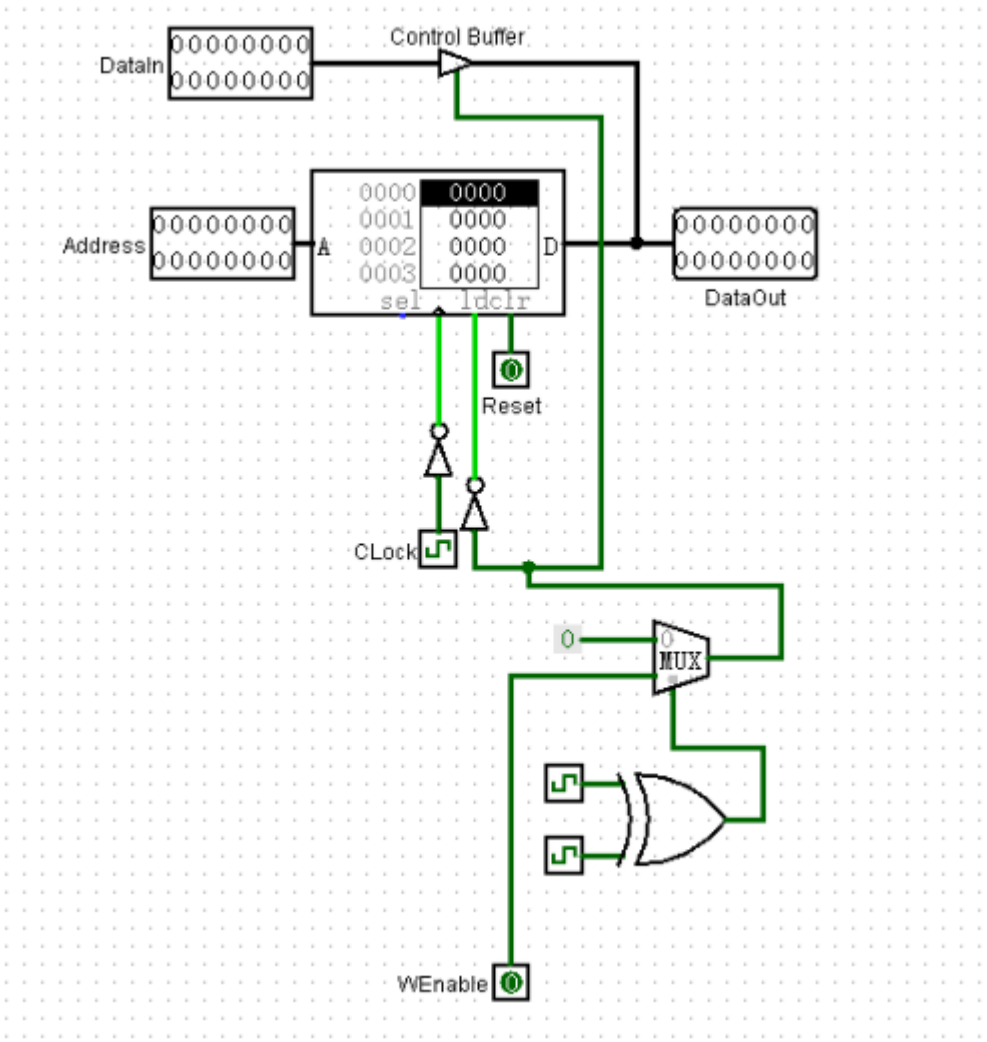
3.设计 PC 自动指向下一条指令

这里要实现 PC 自动切换下一条指令，同时还必须保证 J 格式和 beq 和 bne 指令对 PC 的影响。Perstep 指定一次 pc 往后跳几个。中间的多路选择器实现了跳转到指定地址或者自动执行下一条。



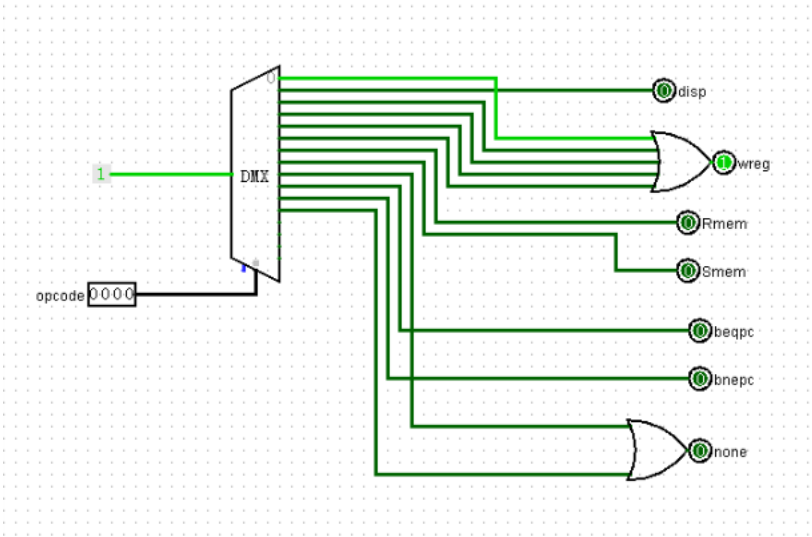
4.RAM 的实现。

这里采用了大纲中的RAM模块，address为操作的地址，wEnable控制写使能，datain和dataout输入输出。

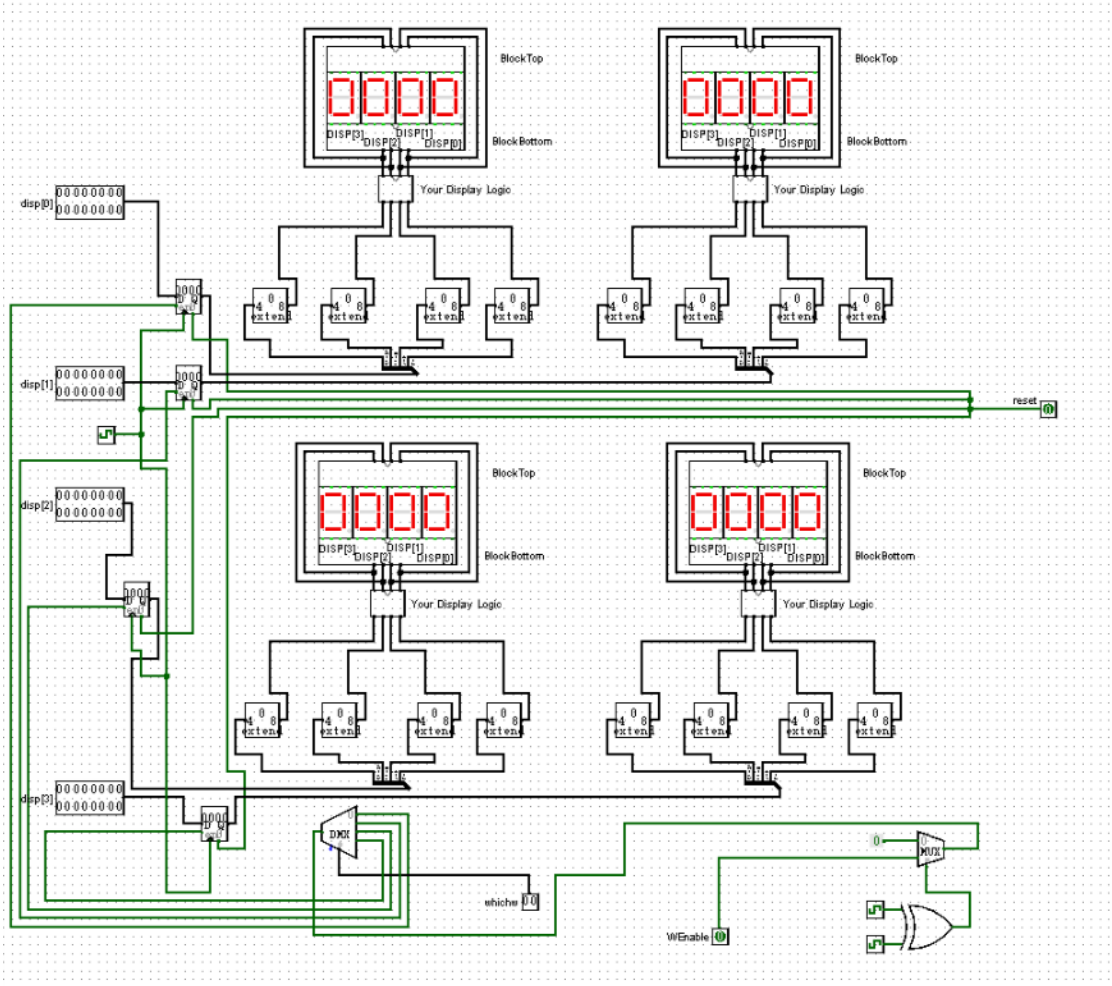


5.信号控制

这个通过 opcode 指定 ALU 计算的结果怎样进行操作，disp wreg rmem smem beqpc bnepc none 分别对应将结果进行显示，写入寄存器，读入与结果匹配的地址的内容，写入与结果匹配的地址的内容，以及 beq 和 bne 跳转到结果的地址。这一层是我自己封装起来的。

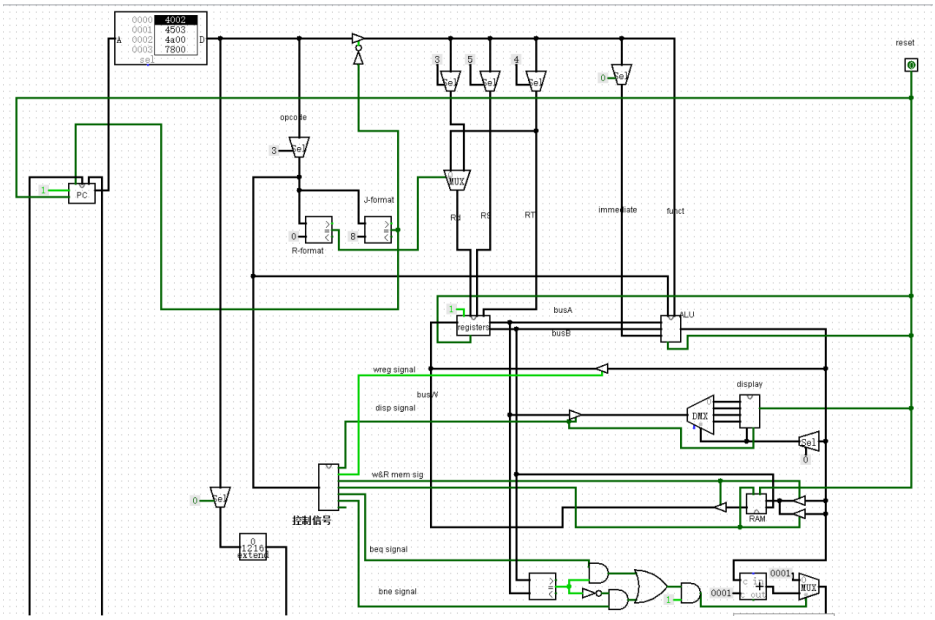


6．显示部分



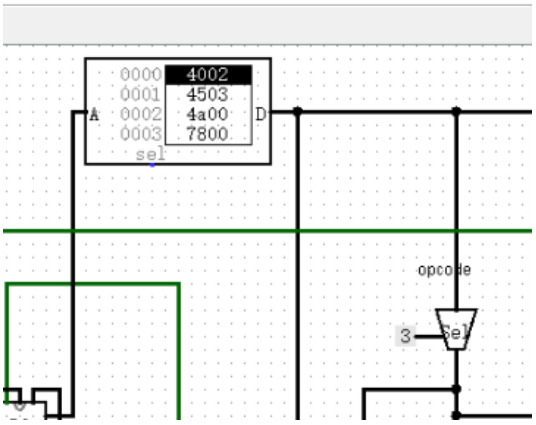
这里我扩展了老师给的显示，将一个显示器的四个七位晶体管分别显示一个寄存器的全部内容。要显示的寄存器为 16 位，分成四部分，每一部分为 4 位，然后将该四位扩展为 8 位进行显示。

CPU 整体预览：

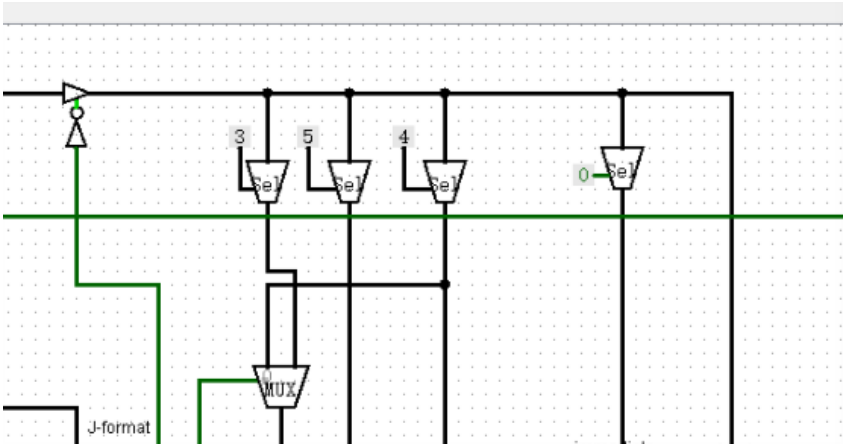


<过程解析>

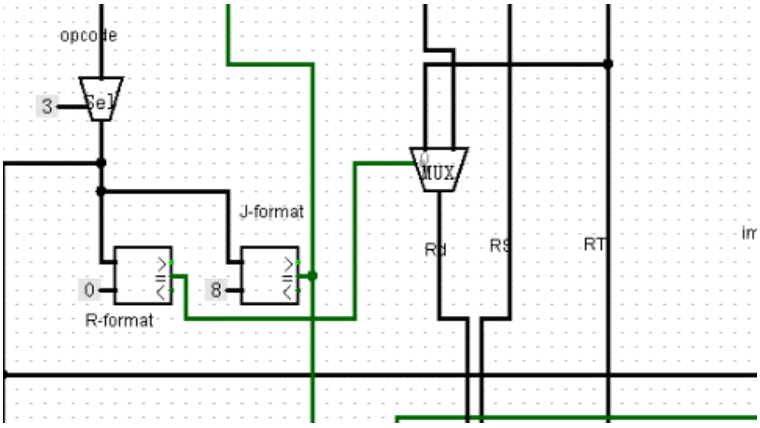
左上角的 ROM 存储指令



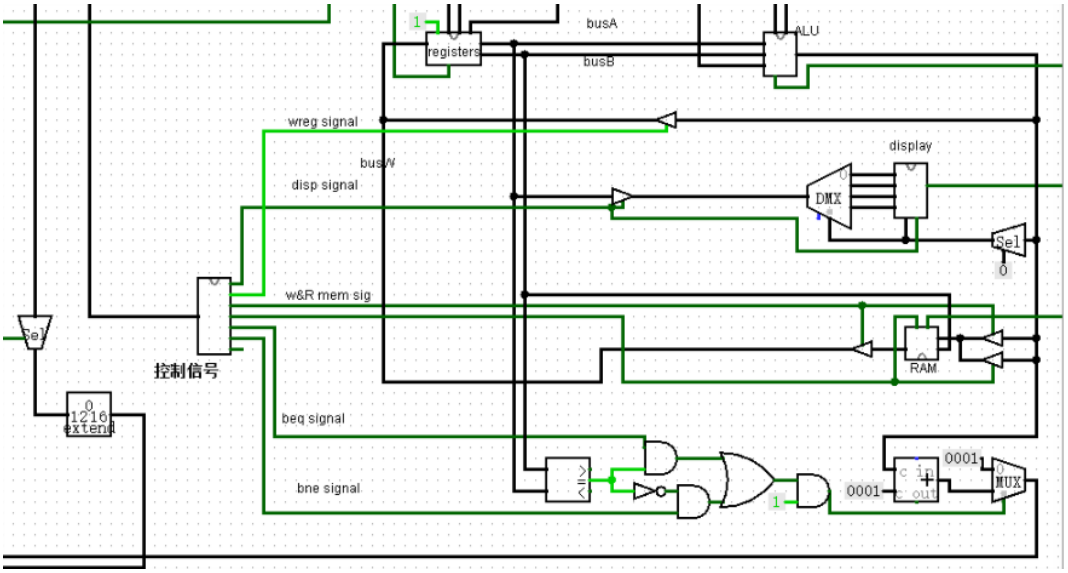
取出对应字段



三种指令的分处理：左边的处理为 R 指令，RS RT RD 直接对应，如果为 I 指令则通过中间的多路选择器切换 RT，J 格式直接对 PC 进行操作。



信号控制

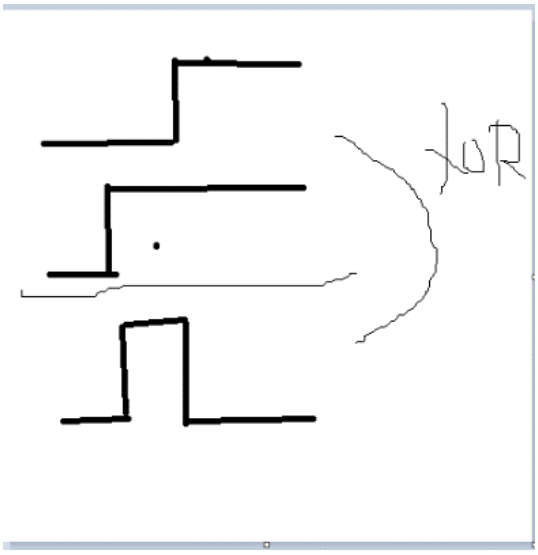


第一行控制了写入寄存器，第二行控制显示，第三行控制了写入 RAM 之中，第四行控制控制 beq 和 bne 进行操作 PC 指针。

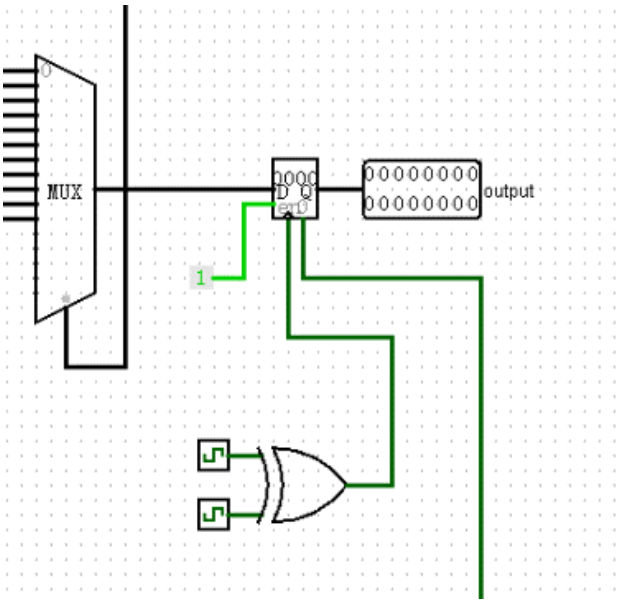
<实现和测试>

时钟周期设置：这里我采用自己的方式实现了所有的写入的操作。解决的问题主要是：addi $r1, $r1, 10 这条指令如果直接经过 ALU 计算写入寄存器文件中，寄存器文件 写使能 全程开放，那么在 PC 跳转的周期内，寄存器文件的时钟周期为其四分之一，则会连续写入四次，即最后的结果为 r1 寄存器结果为 40. 内存写入时也会出现这样的情况。

我的解决办法是：PC 经过 8 个时钟周期进行跳转，寄存器文件、RAM、PC（执行 jump）这三个要允许写入的部分的时钟必须尽可能小，即一个时钟周期。那么为了不出现 ALU 直接计算出的结果重复写入文件，我在 ALU 之后放入一个寄存器，该寄存器的时钟周期为：



XOR 组合出的新波形，用于让 ALU 的寄存器在第 3 个周期开始的时候将结果写入寄存器。



ALU 之后的寄存器的时钟控制

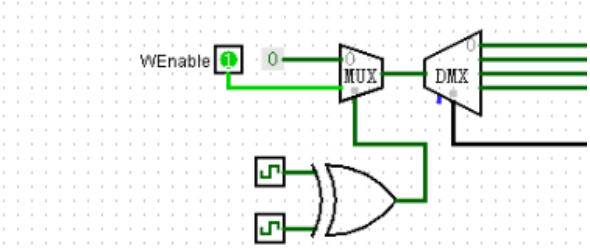
以下为这两个时钟各自的周期：

Clock	
Facing	East
High Duration	4 Ticks
Low Duration	4 Ticks
Label	
Label Location	West
Label Font	SansSerif Plain 12

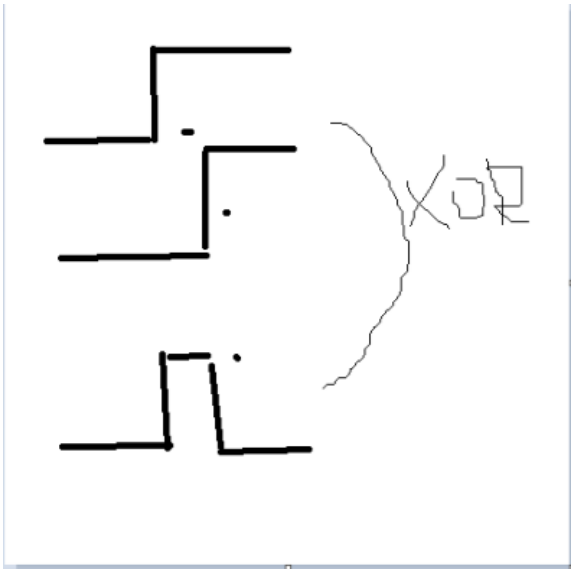
Clock	
Facing	East
High Duration	6 Ticks
Low Duration	2 Ticks
Label	
Label Location	West
Label Font	SansSerif Plain 12

同时，在前面叙述的可写入的部件的写使能不全程开放，让其在第 5 个时钟周期能够写入。这样，让 ALU 的计算和可写入部件在周期上错开，保证不会重复写入。

如寄存器组的写使能时钟控制：



……



Clock	
'acing	East
ligh Duration	4 Ticks
.ow Duration	4 Ticks
.abel	
.abel Location	West
.abel Font	SansSerif Plain 12

Clock	
Facing	East
High Duration	2 Ticks
Low Duration	6 Ticks
Label	
Label Location	West
Label Font	SansSerif Plain 12

<时钟周期整体情况>

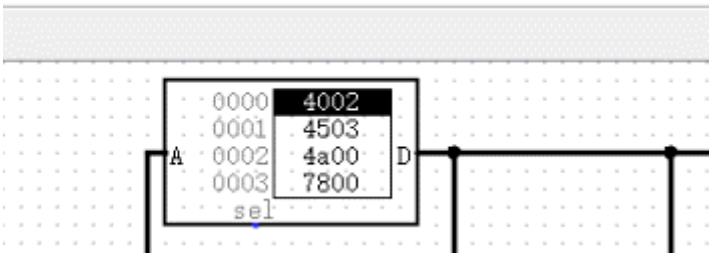
PC：每 8 个周期跳一次

ALU：第 3 个周期开始时产生结果

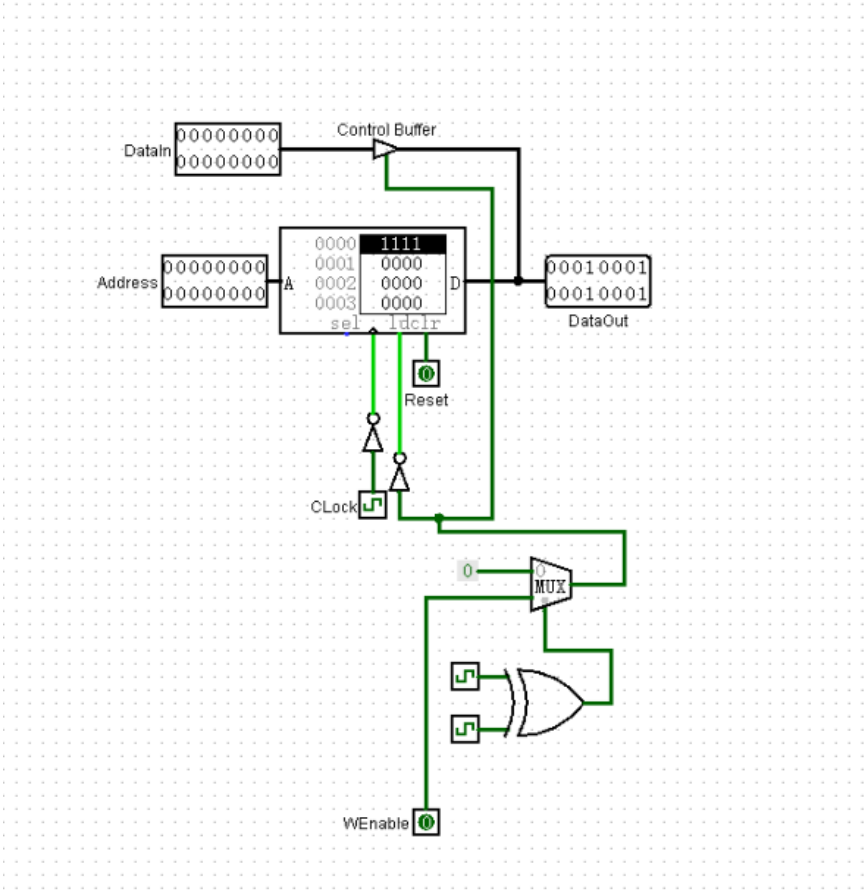
寄存器和内存：第 5 个周期开始时将 ALU 结果写入

<CPU 运行过程>

1. 将指令导入 main 的左上角的 ROM 中



2. 将内存中的数据导入 RAM_Unit 中



3. 在菜单中打开 Simualte 下的 ticks enable 进行执行。

<运行 mult.hex>

Mult 代码将存在 RAM 第一块的数据 2 和存在第二块的数据 3 相乘，存入内存第三块中 6.

Mult 代码：

```
addi $r0, $r0, 2
addi $r1, $r1, 3
addi $r2, $r2, 0
sw $r0, 0($r2)
sw $r1, 1($r2)
sub $r0, $r0, $r0
sub $r1, $r1, $r1
lw $r0, 0($r2)
lw $r1, 1($r2)
beq $r2, $r1, 8
sub $r3, $r3, $r3
lw $r2, 2($r3)
add $r2, $r2, $r0
sw $r2, 2($r3)
sub $r2, $r2, $r2
addi $r3, $r3, 1
sub $r1, $r1, $r3
bne $r2, $r1, -9
beq $r1, $r1, -1
```

导入 mult.hex 并且执行即可。
<运行 octal.hex>
Octal 操作的是 RAM 第一块的数据。
样例操作：手动将 RAM 第一块的数据更改为 0x829f，然后导入指令到 ROM 中开始执行。
执行结果为在 display 模块的 disp[0]显示为　　0237。

总　结

这次实习有一种很浓重的美国高校计算机课设的风格，第一个是直接给出框架代码编写两个函数来进行配合，与我之前做过的 CMU 大学的 CSAPP 课程很类似，第二题直接是用 logisim 编写 CPU，我看了一个多星期的 logisim 官方文档才能够掌握这个工具，这也大大锻炼了我阅读英文的能力。两次实习给我很大的感触，让我拔高了自己的能力，同时打好了计算机底层的基础。