实验报告

智能 1602 班 201608010623 李路

实验名称:

RISC-V 基本指令集模拟器设计与实现

实验目标:

设计一个 CPU 模拟器，能模拟 CPU 指令集的功能

实验要求 :

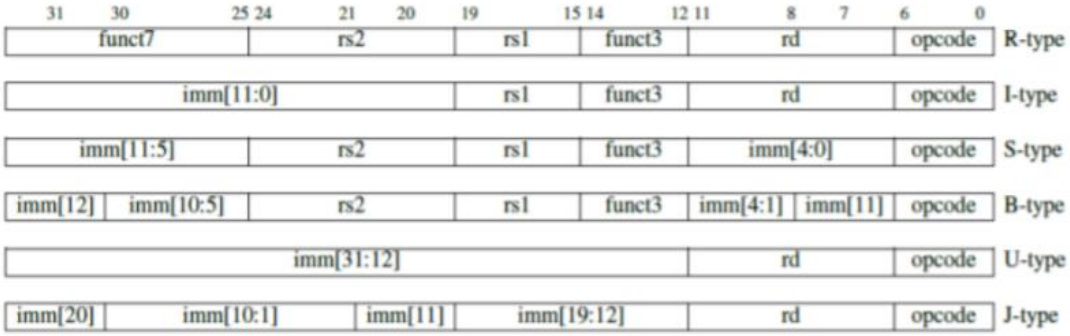
采用 C/C++编写程序 ·模拟器的输入是二进制的机器指令文件，模拟器的输出是 CPU 各个寄存器的状态和相关的存储器单元状态

实验内容 :

1. RISC-V 指令集

RISC-V（英文发音为"risk-five"）是一个全新的指令集架构，该架构最初由美国加州大学伯克利分校的 EECS 部门的计算机科学部门的 Krste Asanovic 教授、Andrew Waterman 和 Yunsup Lee 等开发人员于 2010 年发明。 其中"RISC"表示精简指令集，而其中"V"表示伯克利分校从 RISC I 开始设计的第五代指令集。 2010 年，加州大学伯克利分校的研究团队分析了 ARM、MIPS、SPARC、X86 等多种指 令集，发现这些指令集不仅复杂度不断提升，且还存在知识产权风险，而处理器架构种类和 处理能力并无直接关联。针对以上问题，该小组设计并推出了一套基于 BSD 协议许可的免 费开放的指令集架构 RISC-V，其原型芯片也于 2013 年 1 月成功流片。RISC-V 指令集具有 性能优越，彻底免费开放两大特征。RSIC-V 的设计目标是能够满足从微控制器到超级计算 机等各种复杂程度的处理器需求，支持从 FPGA、ASIC 乃至未来器件等多种实现方式，同时 能够高效地实现各种微结构，支持大量定制与加速功能，并与现有软件及编程语言可良好适 配。RISC-V 产业生态正进入快速发展期。加州大学伯克利分校在 2015 年成立非盈利组织 RISC-V 基金会，该基金会旨在聚合全球创新力量共同构建开放、合作的软硬件社区，打造 RISC-V 生态系统。三年多来，谷歌、高通、IBM、英伟达、NXP、、西部数据、Microsemi、 中科院计算所、麻省理工学院、华盛顿大学、英国宇航系统公司等 100 多个企业和研究机构 先后加入了 RISC-V 基金会。

2. RISC-V 指令集编码格式



3. RISC-V 指令集

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADDIW	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
ADDW	*rd*=x0	2^{10}	
SUBW	*rd*=x0	2^{10}	
SLLW	*rd*=x0	2^{10}	
SRLW	*rd*=x0	2^{10}	
SRAW	*rd*=x0	2^{10}	
FENCE	*pred*=0 or *succ*=0	$2^{5}-1$	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{11}	
SRLI	*rd*=x0	2^{11}	
SRAI	*rd*=x0	2^{11}	
SLLIW	*rd*=x0	2^{10}	
SRLIW	*rd*=x0	2^{10}	
SRAIW	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

模拟器程序框架:

cpu 执行指令的流程为

1. 取指 2. 译码 3. 执行

整个模拟器的运行封装在一个 while 循环中，当输入为 n 的时候，表示停止模拟器。

每执行一条指令就输入是否继续执行，getchar()是用来消去回车的，每次循环依次取指，设置 NextPC，解析指令，根据解析的指令执行相应的操作， 其中 IR 是指令寄存器，用来保存指令，PC 是程序计数器，用来指示指令在存储 器中的位置

```
while(c != 'n') {
    cout << "Registers bofore executing the instruction @0x" << std::hex << PC << endl;
    showRegs();
    IR = readWord(PC);
    NextPC = PC + WORDSIZE;
    decode(IR);
    switch(opcode) {
        case LUI:
```

下面是 decode 的具体实现，根据上面的指令的类型格式取出指令中某些位，比 如 imm11_5s 表示的是 S 类型指令中立即数 5 到 11 位的数据，在后面和 imm4_0s 一起构成了 S 类型指令中的立即数 Imm11_0StypeSignExtended

```
void decode(uint32_t instruction) {//decode是译码的意思，RV32I指令4个字节
    // Extract all bit fields from instruction  从指令中提取所有位字段
    opcode = instruction & 0x7F;//获取低7位，即0~6位
    rd = (instruction & 0x0F80) >> 7;//获取从低至高第7~11位
    rs1 = (instruction & 0xF8000) >> 15;//获取第15~19位，得到第一个寄存器
    zimm = rs1;//zimm是我们定义的一个unsigned int，把rs1赋值给了它
    rs2 = (instruction & 0x1F00000) >> 20;//获取第20~24位，得到第二个寄存器
    shamt = rs2;//shamt是我们定义的一个unsigned int，  把rs2赋值给了它
    funct3 = (instruction & 0x7000) >> 12;//获取第12~14位
    funct7 = instruction >> 25;//获取25~31位?
    imm11_0i = ((int32_t)instruction) >> 20;//转化成有符号的再移动，对应着Itype类型的地址
    csr = instruction >> 20;//获取20~31位，应该与上面的imm11_0i差不多，不过是无符号类型的
    imm11_5s = ((int32_t)instruction) >> 25;//获取第25~31位数据，对应着Stype类型的地址
    imm4_0s = (instruction >> 7) & 0x01F;//获取第7~11位数据，对应Stype类型的地址
    imm12b = ((int32_t)instruction) >> 31;//获取第31位数据，对应Btype类型的地址
    imm10_5b = (instruction >> 25) & 0x3F;//获取第25~30位数据，对应Btype类型的地址
    imm4_1b = (instruction & 0x0F00) >> 8;//第8~11位，对应Btype类型的地址
    imm11b = (instruction & 0x080) >> 7;//第7位，对应Btype类型的地址
    imm31_12u = instruction >> 12;//第12~31位，对应Utype类型的地址
    imm20j = ((int32_t)instruction) >> 31;//第31位，对应jtype类型的地址
    imm10_1j = (instruction >> 21) & 0x3FF;//第21~31位，对应jtype类型的地址
    imm11j = (instruction >> 20) & 1;//第20位，对应jtype类型的地址
    imm19_12j = (instruction >> 12) & 0x0FF;//第12到19位，对应jtype类型的地址
    pred = (instruction >> 24) & 0x0F;
    succ = (instruction >> 20) & 0x0F;
```

具体指令的实现如下(以 LUI,AUIPC,JAL,JALR 为例) 可以看到 LUI 和 AUIPC 是写寄存器的指令，LUI 是把立即数写入 rd 寄存器，AUIPC 把程序计数器和一个立即数相加的写入 rd 寄存器，这个指令的作用是构造 PC 相 对地址。JAL 和 JALR 是无条件跳转指令，是通过对 NextPC 赋值来实现的。我自 己理解 JAL 是相对跳转即相对 PC 跳转，而 JALR 是绝对跳转，即跳转到由 rs1 指 定的指令上去，我们可以先对某一个寄存器赋值，然后再调用 JALR 指令跳转到 我们想跳转到的地方

```
        switch(opcode) {
            case LUI:
                cout << "Do LUI" << endl;
                R[rd] = Imm31_12UtypeZeroFilled;
                break;
            case AUIPC:
                cout << "Do AUIPC" << endl;
                cout << "PC = " << PC << endl;
                cout << "Imm31_12UtypeZeroFilled = " << Imm31_12UtypeZeroFilled << endl;
                R[rd] = PC + Imm31_12UtypeZeroFilled;
                break;
            case JAL:
                cout << "Do JAL" << endl;
                R[rd]=PC+4;
                NextPC = PC+ Imm20_1JtypeSignExtended;
                break;
            case JALR:
                cout << "DO JALR" << endl;
                R[rd]=PC+4;
                NextPC=R[rs1]+Imm20_1JtypeSignExtended;
                break;
```

测试:

部件	配置
CPU	core i7-6300U
内存	8GB
操作系统	windows 10

测试记录:

我用于测试的指令集如下

```
void m_progMem(){
    writeWord(0, (0x666 << 12) | (2 << 7) | (LUI));//指令功能在第2个寄存器写入0x666
    writeWord(4, (1 << 12) | (3 << 7) | (AUIPC));//指令功能在第3个寄存器中写入PC+0x1000
    writeWord(8, (0x66 << 12) | (5 << 7) | (LUI));//指令功能在第5个寄存器写入6
    writeWord(12, (0x0<<25) | (5<<20) | (0<<15) | (SW << 12) | (0x1a << 7) | (STORE));//向(0号寄存器的值加
    writeWord(16, (0x10<<20) | (0<<15) | (LBU<<12) | (4<<7) | (LOAD));//读取0x10地址上的1byte取最后8位写入4
```

第一条指令，在第 2 个寄存器写入 0x666000

可以看到程序打印出了指令执行前后的 PC,IR 值，内存值和寄存器值。显示出了 执行的指令，也可以看到第 2 个寄存器的值由 0 变为了 0x666

```
Registers bofore executing the instruction @0x0
PC=0x0 IR=0x0

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x0 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0 R[c]=0x0 R
]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18]=0x0 R[19
0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Do LUI
Registers after executing the instruction
PC=0x4 IR=0x666137

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0 R[c]=
0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18]=0x0
[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Continue simulation (Y/n)? [Y]
```

第二条指令，在第 3 个寄存器中写入 PC+0x1000，结果如下：

```
y
Registers bofore executing the instruction @0x4
PC=0x4 IR=0x666137

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x0 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0 R[c]=0x
0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18]=0x0 R
[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Do AUIPC
PC = 4
Imm31_12UtypeZeroFilled = 1000
Registers after executing the instruction
PC=0x8 IR=0x1197

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0 R[c]
=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18]=0x
0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Continue simulation (Y/n)? [Y]
y
```

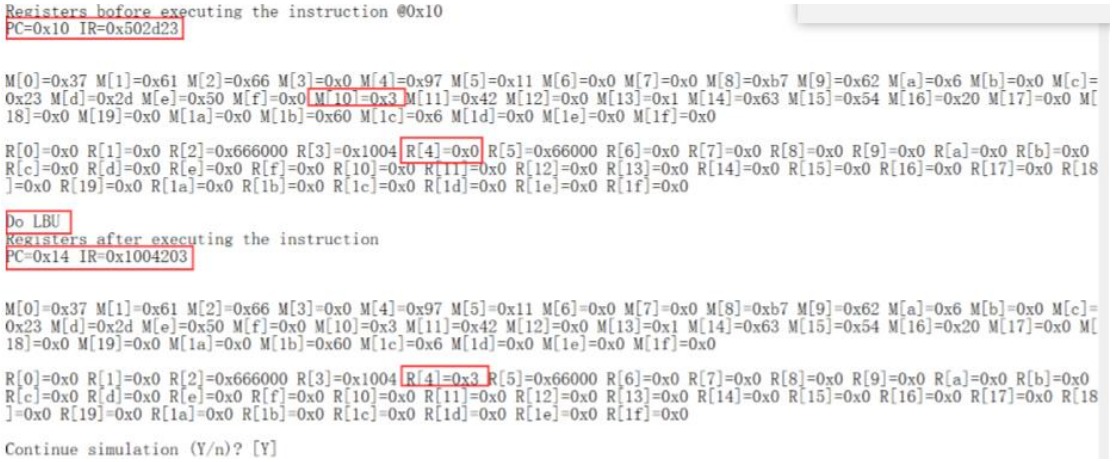
第三条指令，在第 5 个寄存器写入 0x66000，

```
Registers bofore executing the instruction @0x8
PC=0x8 IR=0x1197

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x0 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0 R[c]
=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18]=0x
0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Do LUI
Registers after executing the instruction
PC=0xc IR=0x662b7

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x66000 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0
R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18
]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Continue simulation (Y/n)? [Y]
```

第四条指令，向(0 号寄存器的值加上 0x1a)地址写入 5 号寄存器中的值

```
Registers bofore executing the instruction @0xc
PC=0xc IR=0x662b7

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x0 M[1c]=0x0 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x66000 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0
R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18
]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

DO SW
SW Addr and Data are: 1a, 66000
Registers after executing the instruction
PC=0x10 IR=0x502d23

M[0]=0x37 M[1]=0x61 M[2]=0x66 M[3]=0x0 M[4]=0x97 M[5]=0x11 M[6]=0x0 M[7]=0x0 M[8]=0xb7 M[9]=0x62 M[a]=0x6 M[b]=0x0 M[c]=
0x23 M[d]=0x2d M[e]=0x50 M[f]=0x0 M[10]=0x3 M[11]=0x42 M[12]=0x0 M[13]=0x1 M[14]=0x63 M[15]=0x54 M[16]=0x20 M[17]=0x0 M[
18]=0x0 M[19]=0x0 M[1a]=0x0 M[1b]=0x60 M[1c]=0x6 M[1d]=0x0 M[1e]=0x0 M[1f]=0x0

R[0]=0x0 R[1]=0x0 R[2]=0x666000 R[3]=0x1004 R[4]=0x0 R[5]=0x66000 R[6]=0x0 R[7]=0x0 R[8]=0x0 R[9]=0x0 R[a]=0x0 R[b]=0x0
R[c]=0x0 R[d]=0x0 R[e]=0x0 R[f]=0x0 R[10]=0x0 R[11]=0x0 R[12]=0x0 R[13]=0x0 R[14]=0x0 R[15]=0x0 R[16]=0x0 R[17]=0x0 R[18
]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0

Continue simulation (Y/n)? [Y]
```

第五条指令，读取 0x10 地址上的 1byte 取最后 8 位写入 4 号寄存器



分析和结论:

从测试记录来看，模拟器实现了对二进制指令文件的读入、指令功能的模拟，CPU 和存储器 状态的输出。 根据分析结果，可以认为编写的模拟器实现了所要求的功能，完成了实验目标。