RISC-V 基本指令集模拟器设计与实现

班级：计科 1601
学号：201608010112
姓名：庞姝颖

一、实验目标

完成一个模拟 RISC-V 的基本整数指令集 RV32I 的模拟器设计。

二、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单元状态

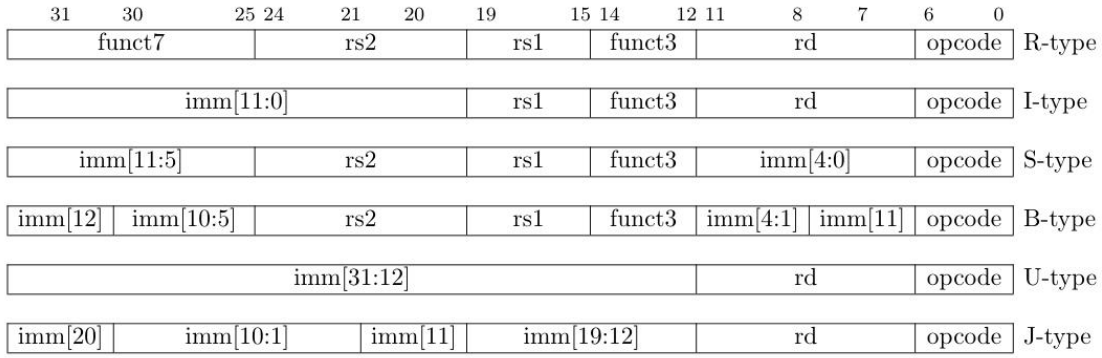
三、实验内容

1、RISC-V 指令集内容

本次实验采用的是 RV32I 指令集，它支持 32 位寻址空间，支持字节地址访问，仅支持小端格式，寄存器也是 32 位整数寄存器。RV32I 指令集的目的是尽量简化硬件的实施设计，含有 47 条指令（现在应该有 40 条，在最新的规范中，一些 csr 指令被放在扩展指令集中）。着重实现 37 条。

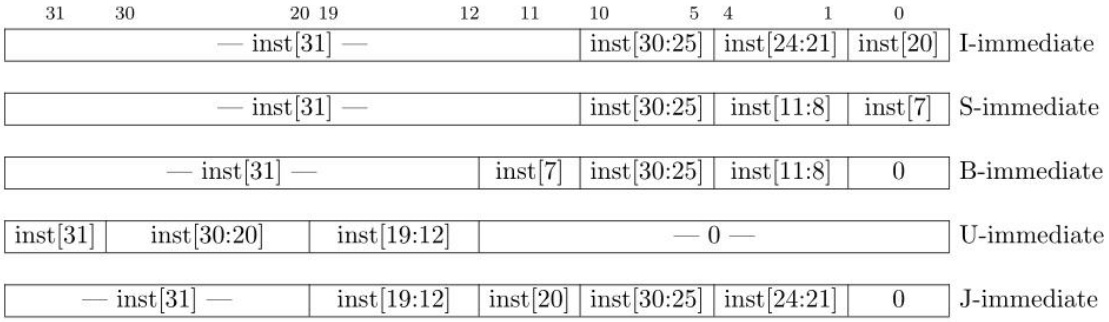
2、RISC-V 指令集编码格式

其中包含了六种基本指令格式，每种指令格式都是固定的 32 位指令，所以指令在内存中必须 4 字节对齐。Rd 表示目的寄存器，rs1 是源操作数寄存器 1，rs2 是源操作数寄存器 2。



Imm 表示指令中的立即数，比如 imm[11:0]，表示一个 12 位的立即数，它的高 20 位会符号位扩展，也就是最左边的位 imm[11]来进行扩展。Imm[31:12]表示一个 32 位的立即数，它的低 12 位会补 0。

下图为各种指令格式扩展后的 32 位立即数。



3、RISC-V 指令集

1. load 和 store 指令

Load/store 指令在 memory 和寄存器之间传输数据，load 指令编码为 I 型，store 指令编码为 S 型。计算 memory 地址时候，imm 都会符号扩展成 32 位，然后和 rs1 相加，得到 memory 地址。为了提高性能，load/store 指令应该尽量对齐地址，比如 lw 指令，访问地址应该 4 字节对齐，lh 访问地址应该双字节对齐。根据微架构实现的不同，不对齐地址的访问可能会比较慢，而且地址对齐访问，能够确保是原子操作，不对齐的话为了读取和存储数据正确，还要进行额外的同步操作。

category	fmt	RV32I base	Machine code
Load byte	I	Lb rd,rs1,imm	[31-20,imm][19-15,rs1]000[11-7,rd]0000011
Load half word	I	Lh rd,rs1,imm	[31-20,imm][19-15,rs1]001[11-7,rd]0000011
Load word	I	Lw rd,rs1,imm	[31-20,imm][19-15,rs1]010[11-7,rd]0000011
Load byte unsigned	I	Lbu rd,rs1,imm	[31-20,imm][19-15,rs1]100[11-7,rd]0000011
Load half unsigned	I	Lhu rd,rs1,imm	[31-20,imm][19-15,rs1]101[11-7,rd]0000011
Store byte	S	sb rs1, rs2, imm	[31-25,imm[11-5]][24-20,rs2,][19-15,rs1]000[11-7,imm[4-0]]0100011
Store half word	S	sh rs1, rs2, imm	[31-25,imm[11-5]][24-20,rs2,][19-15,rs1]001[

			11-7,imm[4-0]]0100011
Store word	S	sw rs1, rs2, imm	[31-25,imm[11-5]][24-20,rs2,][19-15,rs1]010[11-7,imm[4-0]]0100011

2. 整数计算指令（算术，逻辑指令，比较指令及移位指令）

计算指令在寄存器和寄存器之间，或者在寄存器和立即数之间进行算术或逻辑运算。指令格式为 I，R，U。整数计算指令不会产生异常。

category	fmt	RV32I base	Machine code
arithmetic			
add	R	add rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]000[11-7,rd]0110011
Add imm	I	addi rd, rs1, imm	[31-20,imm][19-15,rs1]000[11-7,rd]0010011
subtract	R	sub rd, rs1, rs2	0100000[24-20,rs2][19-15,rs1]000[11-7,rd]0110011
Load upper imm	U	lui rd, imm	[31-12,imm][11-7,rd]0110111
Add upper imm to pc	U	auipc rd, imm	[31-12,imm][11-7,rd]0010111
logical			
xor	R	xor rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]100[11-7,rd]0110011
Xor imm	I	xori rd, rs1, imm	[31-20,imm][19-15,rs1]100[11-7,rd]0010011
or	R	or rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]110[11-7,rd]0110011
Or imm	I	ori rd, rs1, imm	[31-20,imm][19-15,rs1]110[11-7,rd]0010011
and	R	and rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]111[11-7,rd]0110011
And imm	I	andi rd, rs1, imm	[31-20,imm][19-15,rs1]111[11-7,rd]0010011
shifts			
shift left	R	SLL rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]001[11-7,rd]0110011
shift left immediate	I	SLLI rd, rs1, shamt	0000000[24-20,imm][19-15,rs1]001[11-7,rd]0010011
shift right	R	SRL rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]101[11-7,rd]0110011
shift right immediate	I	SRLI rd, rs1, shamt	0000000[24-20,imm][19-15,rs1]101[11-7,rd]0010011
shift right arithmetirc	R	SRA rd, rs1, rs2	0100000[24-20,rs2][19-15,rs1]101[11-7,rd]0110011
shift right arith imm	I	SRAI rd, rs1, shamt	0100000[24-20,imm][19-15,rs1]101[11-7,rd]0010011
compare			

set <	R	slt rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]010[11-7,rd]0110011
set < immediate	I	slti rd, rs1, imm	[31-20,imm][19-15,rs1]010[11-7,rd]0010011
set < unsigned	R	sltu rd, rs1, rs2	0000000[24-20,rs2][19-15,rs1]011[11-7,rd]0110011
set < imm unsigned	I	sltiu rd, rs1, imm	[31-20,imm][19-15,rs1]011[11-7,rd]0010011

3. 控制指令，包括无条件跳转指令和条件跳转指令

category	fmt	RV32I base	Machine code
branch			
branch =	B	beq rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]000[11-7, imm[4:1][11]]1100011
branch <>	B	bne rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]001[11-7, imm[4:1][11]]1100011
branch <	B	blt rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]100[11-7, imm[4:1][11]]1100011
branch >=	B	bge rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]101[11-7, imm[4:1][11]]1100011
branch < unsigned	B	bltu rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]110[11-7, imm[4:1][11]]1100011
branch >=unsigned	B	bgeu rs1, rs2,imm	[31-25, imm[12][10:5]][24-20, rs2][19-15, rs1]111[11-7, imm[4:1][11]]1100011
jump and link			
J&L	J	JAL rd, imm	[31-12, imm[20][10:1][11][19:12]][11-7,rd]1101111
Jump and link register	I	JALR, rd, rs1, imm	[31-20,imm[11:0]][19-15,rs1]000[11-7,rd]1100111

4. 同步指令

Risc-V 在多个 hart（硬件线程）之间使用的是松散一致性模型，所以需要存储器 fence 指令。fence 指令能够保证存储器访问的执行顺序。在 fence 指令之前的所有存储器访问指令，比该 fence 之后的所有数据存储器访问指令先执行。

fence.i 指令用于同步指令和数据流。如果程序中添加一个 fence.i,则该指令能够保证 fence.i 之前所有指令的访存结果能被 fence.i 之后的所有指令访问到。通常说来，处理器的微架构硬件实现时，一旦遇到一条 fence.i 指令，便会先等到之前的所有访存指令执行完，然后冲刷流水线，包括 Icache，使其后的所有指令，能够重新取指，从而得到最新的值。

category	fmt	RV32I base	Machine code
Synch			

synch thread	I	FENCE iorw，iorw	0000[27-24,pred][23-20,succ]00000000000000001111
synch instr and data	I	FENCE.i	00000000000000000001000000001111

5. 控制状态寄存器指令

category	fmt	RV32I base	Machine code
CSR			
CSRRW	I	CSRRW	[31-20,csr][19-15, rs1]001[11-7,rd]1110011
CSRRS	I	CSRRS	[31-20,csr][19-15, rs1]010[11-7,rd]1110011
CSRRC	I	CSRRC	[31-20,csr][19-15, rs1]011[11-7,rd]1110011
CSRRWI	I	CSRRWI	[31-20,csr][19-15,zimm]101[11-7,rd]1110011
CSRRSI	I	CSRRSI	[31-20,csr][19-15,zimm]110[11-7,rd]1110011
CSRRCI	I	CSRRCI	[31-20,csr][19-15,zimm]111[11-7,rd]1110011

6. 环境调用和断点指令

System call I ECALL 00000000000000000000000001110011

System break I EBREAK 00000000000100000000000001110011

这两条指令能够产生环境调用异常和生成断点异常，产生异常时，当前指令的 pc 值被写入 mepc 寄存器。

四、实验设计

模拟器程序框架

考虑到 CPU 执行指令的流程为：

取指

译码

执行（包括运算和结果写回）

模拟器的框架程序如下：

```
while(1){
    inst=fetch(cpu.pc);
    cpu.pc=cpu.pc+4;

    inst.decode();
    switch(inst.opcode){
        case ADD:
            cpu.regs[inst.rd]=cpu.regs[rs]+cpu.regs[rt];
```

```
            break;
        case /*其他操作码*/:
            break;
        default:
            cout<<"无法识别的操作码："<<inst.opcode;
    }
}
```

其中 while 循环条件可以根据需要改为模拟终止条件。

五、测试

1、测试环境

部件	配置
CPU	Core i5-6200U
内存	DDR3 4GB
操作系统	Windows 10

2、测试内容

提前写入的测试指令，主要测试了 load 和 store 指令、整数计算指令、控制转移指令，而同步指令、控制状态寄存器指令、环境调用和断点指令只是做了大致了解，并没有仔细地设计实现。

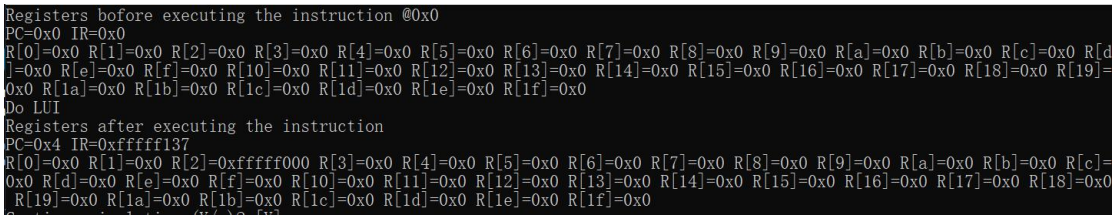
```
void progMem() {
    // Write starts with PC at 0
    writeWord(0, (0xfffff << 12) | (2 << 7) | (LUI));
    writeWord(4, (1 << 12) | (5 << 7) | (AUIPC));
    writeWord(8, (0x20<<25) | (5<<20) | (0<<15) | (SW << 12) | (0 << 7) | (STORE));
    writeWord(12, (0x400<<20) | (0<<15) | (LB<<12) | (3<<7) | (LOAD));
    writeWord(16, (0x400<<20) | (0<<15) | (LBU<<12) | (7<<7) | (LOAD));
    writeWord(20, (0x0<<25) | (2<<20) | (0<<15) | (BGE<<12) | (0x8<<7) | (BRANCH));
    writeWord(28, (0x8<<20) | (3<<15) | (SLTIU<<12) | (8<<7) | (ALUIMM));
    writeWord(32, (SRAI<<25) | (0x2<<20) | (0x2<<15) | (SHR<<12) | (9<<7) | (ALUIMM));

    writeWord(36,(0x400)<<20|(1<<15)|(JALRfun<<12)|(4<<7)|(JALR));
    writeWord(40, (0x20<<25) | (7<<20) | (0<<15) | (SH << 12) | (9 << 7) | (STORE));
    writeWord(44, (0x0<<25) | (4<<20) | (1<<15) | (BGEU<<12) | (0x8<<7) | (BRANCH));
    writeWord(48, (0x400<<20) | (2<<15) | (ORI<<12) | (4<<7) | (ALUIMM));
    writeWord(52, (SUB<<25) | (4<<20) | (2<<15) | (ADDSUB << 12) | (9 << 7) | (ALURRR));

    writeWord(56, (1<<31) |(0<<25) | (8<<20) | (0<<15) | (BLTU << 12) | (0 << 11) |(0 << 7) | (BRANCH));
    writeWord(60, (0x20<<25) | (8<<20) | (0<<15) | (SB << 12) | (0 << 7) | (STORE));
    writeWord(64, (0x100<<20) | (3<<15) | (XORI << 12) | (9 << 7) | (ALUIMM));
    writeWord(68, (ADD<<25) | (3<<20) | (1<<15) | (ADDSUB << 12) | (10 << 7) | (ALURRR));
    writeWord(72, (1 << 31) |(1 << 23) |(1 << 22) |(1 << 12) | (7 << 7) | (JAL));

    writeWord(76, 0x0013ab73);// CSRRS
    writeWord(80, 0x0013db73);//CSRRWI
    writeWord(84, 0x0013fb73);//CSRRCI
    writeWord(88, 0x0000100f);//FENCE_I
//  writeWord(16, 0x00100073);//EBREAK,默认跳转到 pc为4的位置
}
```

3、测试结果

1. 整数计算指令

算术指令：

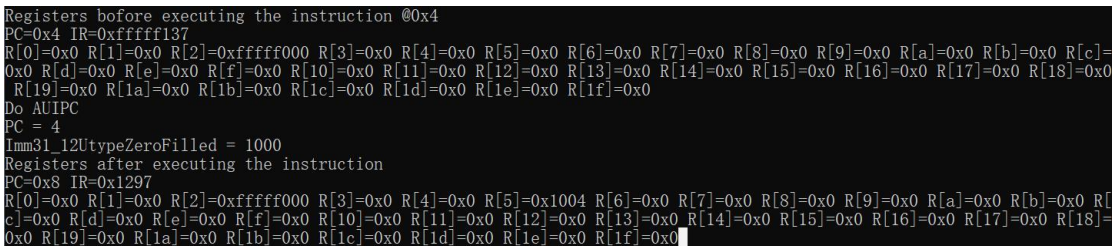
LUI 指令



测试　writeWord(0, (0xfffff << 12) | (2 << 7) | (LUI));

20 位立即数 imm 左移 12 位，并将低 12 位置零，写入 r2 寄存器中，可以看到执行指令前 r2 寄存器储值为 0，执行指令后 r2 寄存器储值为 0xfffff000，结果正确。

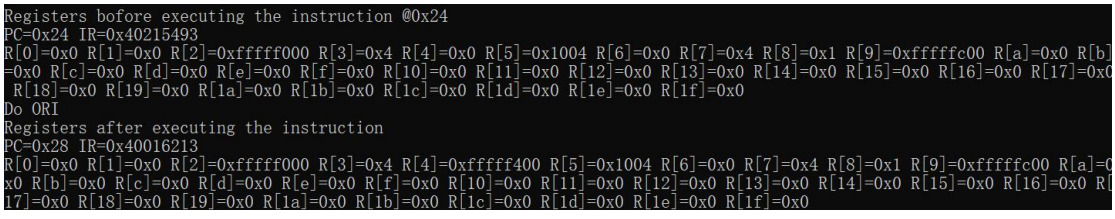
AUIPC 指令



测试　writeWord(4, (1 << 12) | (5 << 7) | (AUIPC));

该指令用 imm 构建一个偏移量的高 20 位，低 12 位填 0，并将此偏移加到 pc 上，将结果写入 rd。我们可以观察到，执行指令前 r5 寄存器储值为 0，执行指令后 r2 寄存器储值为 0x1004（=PC+imm<<12），结果正确。

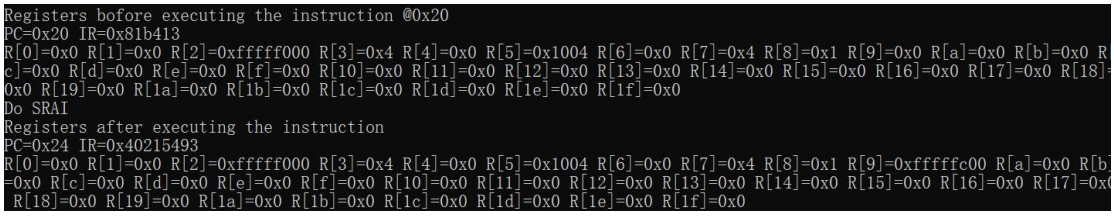
逻辑指令：



测试　writeWord(36, (0x400<<20) | (2<<15) | (ORI<<12) | (4<<7) | (ALUIMM));

ORI 指令，rd=rs1|imm，符号扩展 12bit 数 imm（0x400）或 rs1（r2）存储的值，结果放在 rd。可以看到，r4 存储的值从 0x0 变为 0xfffff400，结果正确。

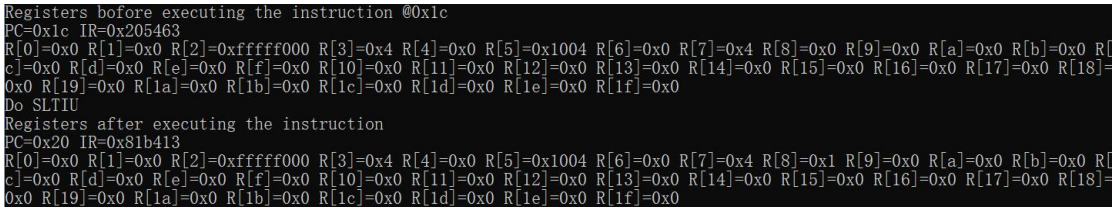
移位指令：



测试　writeWord(32, (SRAI<<25) | (0x2<<20) | (0x2<<15) | (SHR<<12) | (9<<7) | (ALUIMM));

SRAI 指令，立即数算术右移，将寄存器 r2 右移 shamt 位（本例为 2 位），空位用 x[r2]中最高位填充，结果存在 rd（本例为 r9）中。可以看到，r9 存储的值从 0x0 变为 0xfffffc00，结果正确。

比较指令：

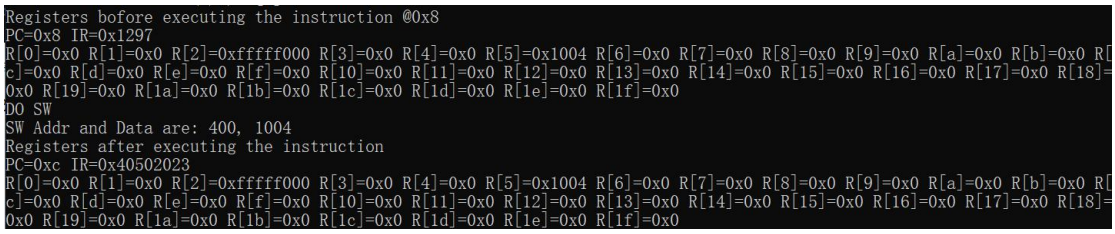


测试　writeWord(28, (0x8<<20) | (3<<15) | (SLTIU<<12) | (8<<7) | (ALUIMM));

SLTIU 指令，无符号数小于立即数则置位，比较 x[r3]和有符号扩展的 imm，比较时视为无符号数。如果 x[r3]更小，向 rd（x[r8]）中写入 1，否则写入 0。可以看到，r3 寄存器中的值为 4 小于立即数 8，所以向目的寄存器中写入 1，即 r8 中值为 1。

2. load 和 store 指令

STORE 类指令

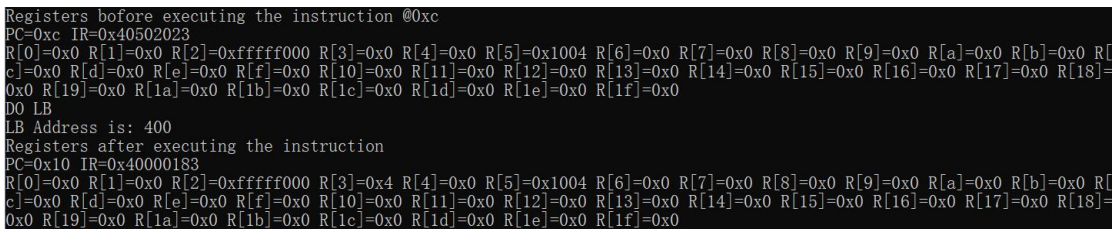


测试　writeWord(8, (0x20<<25) | (5<<20) | (0<<15) | (SW << 12) | (0 << 7) | (STORE));

SW/SH/SB 指令，分别将寄存器 rs2 中的低 32/16/8/位存储到 mem[rs1+imm]中。可以看到，执行指令前 r5 寄存器储值为 0x1004，执行指令后 M[1024]储值为 0x1004，结果正确。

注：SW Addr and Data are: 400, 1004，这些数字都是 16 进制,如 0x400=1024。

LOAD 类指令



测试　writeWord(12, (0x400<<20) | (0<<15) | (LB<<12) | (3<<7) | (LOAD));

LB 指令，读取存储器 8 位，然后用符号位扩展到 32 位，再保存到 rd 中。可以看到，执行指令前 r3 寄存器储值为 0x0，执行指令后 r3 寄存器储值为 0x4，结果正确。

```
Registers bofore executing the instruction @0x10
PC=0x10 IR=0x40000183
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x0 R[5]=0x1004 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 LBU
Registers after executing the instruction
PC=0x14 IR=0x40004383
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x0 R[5]=0x1004 R[6]=0x0 R[7]=0x4 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
```

测试　writeWord(16, (0x400<<20) | (0<<15) | (LBU<<12) | (7<<7) | (LOAD));

LBU 指令，读取存储器 8 位，然后用 0 扩展到 32 位，再保存到 rd 中。我们可以观察到，执行指令前 r7 寄存器储值为 0x0，执行指令后 r3 寄存器储值为 0x4，结果正确。

3. 控制转移指令

有条件跳转指令

```
Registers bofore executing the instruction @0x14
PC=0x14 IR=0x40004383
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x0 R[5]=0x1004 R[6]=0x0 R[7]=0x4 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 BGE
Registers after executing the instruction
PC=0x1c IR=0x205463
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x0 R[5]=0x1004 R[6]=0x0 R[7]=0x4 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
```

测试　writeWord(20, (0x0<<25) | (2<<20) | (0<<15) | (BGE<<12) | (0x8<<7) | (BRANCH));

BGE 指令，如果 r0 寄存器存储值大于或等于 r2 寄存器存储值，则跳转，因为该指令是有符号比较，而 r2 中数字为负数，所以发生跳转，PC 变为 0x1c。

无条件跳转

```
Registers bofore executing the instruction @0x24
PC=0x24 IR=0x40215493
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x0 R[5]=0x1004 R[6]=0x0 R[7]=0x4 R[8]=0x1 R[9]=0xfffffc00 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 JALR
Registers after executing the instruction
PC=0x400 IR=0x40008267
R[0]=0x0 R[1]=0x0 R[2]=0xfffff000 R[3]=0x4 R[4]=0x28 R[5]=0x1004 R[6]=0x0 R[7]=0x4 R[8]=0x1 R[9]=0xfffffc00 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]=0x
0 R[18]=0x0 R[19]=0x0 R[1a]=0x0 R[1b]=0x0 R[1c]=0x0 R[1d]=0x0 R[1e]=0x0 R[1f]=0x0
```

测试　writeWord(36,(0x400)<<20|(1<<15)|(JALRfun<<12)|(4<<7)|(JALR));

JALR 指令，将 pc+4 写入到 rd 寄存器（也就是 r4），pc 变为 r[1]存储值加 0x400。

六、实验总结

通过本次实验，我深入了解了 RV32I 基础整数指令，对各个指令的作用及实现方式也有了较为清晰的认识。参考老师的代码，发现了许多问题，通过查阅网络和指令集手册，理解各个指令的具体实现，还是很有趣的。而测试过程十分枯燥，花费很多时间，但是这个也是十分有必要的。