

计算机系统

LAB1实验报告

班级：人工智能2103
学号：202107030125
姓名：姚丁钰

实验1.1 原型机I

- 实验目的
- 实验资源
- 实验准备
- 练习内容
 - 1.config 加法
 - 2.config 找出最大值
 - 3.config 除法
- 思考问题
 - (1)
 - 乘法
 - 除法
 - (2)

实验1.2 原型机II-扩充指令集

- 实验目的
- 实验资源
- 实验准备
- 练习内容
 - 乘法
 - 思路一
 - 思路二
 - 除法
- 思考问题
 - (1)
 - (2)
 - (3)
 - (4)

总结

- 实验中出现的問題
- 心得体会

实验1.1 原型机I

实验目的

- 了解冯诺伊曼体系结构；
- 理解指令集结构及其作用；
- 理解计算机的运行过程，就是指令的执行过程，并初步掌握调试方法。

实验资源

操作系统： `Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-39-generic x86_64)`

实验准备

- 阅读教材，掌握冯诺伊曼体系的相关内容；
- 学习课程《最小系统与原型机I》。

指令格式	例子	说明
0	0	停机指令，原型机停止运行
1	1	输入一个整数，这个整数必须大于等于0小于127，输入后，此数值保存在R0
2 Ra Rb	2 R0 R1	加法指令，将Ra和Rb的值相加，结果放在Rb中
3 Ra Rb	3 R2,R1	减法指令Ra, Rb。其意义是将寄存器Rb的值减去和Ra中的值，结果放到Rb中，这两个寄存器不能为R3，当结果大于0时，R3中赋值为1，当结果小于0时，R3中赋值为-1，当结果等于0时，R3中赋值为0
4 \$1 Ra	4 10 R1	寄存器直接赋值指令。其中Ra为寄存器名称，\$1为常数，意义是将常数值\$1直接放到寄存器Ra中
5 A B	5 R0 R1 5 R0 0000	传送指令，其中A, B为寄存器编号或内存地址意义是将A处的值传送至B处
6 bias	6 -2 6 3	判断跳转指令。其中 bias为一个整数（可以为负），意义是如果R3的值为1，则跳转当前指令+bias条指令处执行，否则执行下一条指令
7 bias	7 2 7 -3	直接跳转指令。其中bias为一个整数（可以为负），直接跳转当前指令+bias条指令处执行
8 Ra	8 R0	输出指令。将Ra的值发送至显存，并输出

练习内容

1.config 加法

基本思路：累加和 $5 + 4 + 3 + 2 + 1$

使用 `./vm64 1.config` 来运行原型机的模拟器，其中 `1.config` 文件为配置文件，一共有四行，其格式如下：

```
1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000, 0001, 0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 a.txt //表示指令内容在a.txt中
```

指令内容 `d.txt`

```
1 1
2 5 R0 0000
3 4 1 R2
4 2 R0 R1
5 3 R2 R0
6 6 -2
7 5 R1 0001
8 8 R1
9 0
```

结论：输入一个大于1的数字a，计算1+2+.....+a的值并显示出来

```
#include <stdio>

void main()
{
    int a;
    int sum;
    int i;
    sum=0;
    scanf("%d",&a);
    for(i=a;i>=1;i--)
        sum=sum+i;
    printf("%d",sum);
}
```

编译转换成原型系统能够运行的指令集合，在运行将变量a分配内存地址0000，为变量sum分配内存地址0001，为变量i分配内存地址0010 所有内存地址及寄存器初始值为0

1. 输入
2. 数据拷贝 R0, 0000
3. 寄存器赋值 \$1,R2
4. loop: 做加法 R0,R1
5. 做减法 R2,R0
6. 判断跳转 loop
7. 数据拷贝 R1,0001
8. 输出 R1
9. 停机

- 分析过程

1. 此时输入 `ir`，可以查看各个寄存器的值，而输入 `x 6 0000`则表示查看从0000开始的连续6个内存地址值

```

vm>i r
R0: 0 0x0
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0011
vm>x 6 0000
0000: 0
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 4 1 R2

```

2. 输入**si**，表示运行一条指令，例如此时运行的指令是1，表示等待输入，输入一个值5，在输入完成后将此数值存至**R0**寄存器

运行完成后，再运行**i r**指令，就可以看到输入的值5确实是已经存在**R0**寄存器中，每个寄存器的值都用十进制和十六进制表示

```

vm>si
5
0100: 5 R0 0000
vm>i r
R0: 5 0x5
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0100

```

3. 继续使用**si**指令单步执行指令 5 R0 0000，表示将R0寄存器中的值5传送到内存地址0000处

```

vm>x 8 0000
0000: 0
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 4 1 R2
0110: 2 R0 R1
0111: 3 R2 R0
vm>si
0101: 4 1 R2
vm>i r
R0: 5 0x5
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0101
vm>x 8 0000
0000: 00000101
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 4 1 R2
0110: 2 R0 R1
0111: 3 R2 R0

```

4. 继续si执行下一条指令 4 1 R2，表示将立即数1传送到寄存器R2中，寄存器R2直接赋值

```

vm>si
0110: 2 R0 R1
vm>i r
R0: 5 0x5
R1: 0 0x0
R2: 1 0x1
R3: 0 0x0
PC: 0110

```

5. 继续si执行下一条指令 2 R0 R1，表示将R0与R1的值相加，结果放于R1中

```

vm>si
0111: 3 R2 R0
vm>i r
R0: 5 0x5
R1: 5 0x5
R2: 1 0x1
R3: 0 0x0
PC: 0111

```

6. 继续si执行下一条指令 3 R2 R0，表示将R0的值减去R2的值，结果放于R0中

```

vm>si
1000: 6 -2
vm>i r
R0: 4 0x4
R1: 5 0x5
R2: 1 0x1
R3: 1 0x1
PC: 1000

```

7. 继续si执行下一条指令 6 -2，即有条件跳转，如果R3的值为1，则需要向前或向后跳转，此时跳转的值为-2，则表示需要向前跳转两条指令，再去执行 2 R0 R1指令

```

vm>si
0110: 2 R0 R1
vm>i r
R0: 4 0x4
R1: 5 0x5
R2: 1 0x1
R3: 1 0x1
PC: 0110

```

从整体上来看，我们的任务是要执行累加和 $5+4+3+2+1$ ，截止到目前，我们已经完成了 $5+4+3+2+1$ 中的第一步计算，并保存结果到了R1中，现在准备去执行+4这一操作，执行完 $6-2$ 这一操作后，PC值发生变化，在下图中观察到了PC值的改变。

8. 在计算完 $5+4+3+2+1$ 后，这时候R0的值也变成了1，执行 3 R2 R0时，R0的值变为0，而R3的值也变成了0，表示刚才相减的结果为0

```

vm>si
1000: 6 -2
vm>i r
R0: 1 0x1
R1: 14 0xe
R2: 1 0x1
R3: 1 0x1
PC: 1000

vm>si
0110: 2 R0 R1
vm>i r
R0: 1 0x1
R1: 14 0xe
R2: 1 0x1
R3: 1 0x1
PC: 0110

vm>si
0111: 3 R2 R0
vm>i r
R0: 1 0x1
R1: 15 0xf
R2: 1 0x1
R3: 1 0x1
PC: 0111

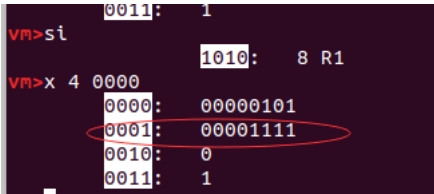
```

9. 这时候执行下一条指令 6 -2时，由于R3的值为0，不满足跳转的条件，因此会执行下一条语句 5 R1 0001，表示将寄存器的值保存到内存中，以便于其他的代码调用

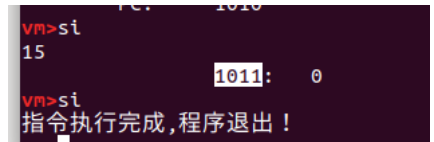
```

vm>si
1000: 6 -2
vm>i r
R0: 0 0x0
R1: 15 0xf
R2: 1 0x1
R3: 0 0x0
PC: 1000

```



10. 继续si或者使用si 4等来批量执行指令，或者是使用c来执行指令直到程序结束。由于最后我们使用了8 R1来输出最后结果，因此会打印出15这一结果



11. 程序执行完毕后，可以使用q退出

2.config 找出最大值

使用 ./vm64 2.config 来运行原型机I的模拟器，其中 2.config 文件为配置文件，一共有四行，其格式如下：

```

1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000，0001，0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 b.txt //表示指令内容在b.txt中

```

指令内容b.txt

```

1 1//输入
2 5 R0 0000
3 5 R0 R1
4 1//输入
5 5 R0 0001
6 3 R1 R0
7 6 3
8 5 0000 0010
9 7 2
10 5 0001 0010
11 5 0010 R0
12 8 R0
13 0

```

现有如下三种情况

- 第一次输入大于第二次输入
- 第一次输入小于第二次输入
- 第一次输入等于第二次输入

结论：输入两个数，保存这两个数，并找出其中的最大值

```

void main()
{
    int a,b;
    int max=0;
    scanf("%d,%d",&a,&b);
    max=a;
    if(b>a)
        max=b;
    printf("%d",max);
}

```

变量分配内存地址:

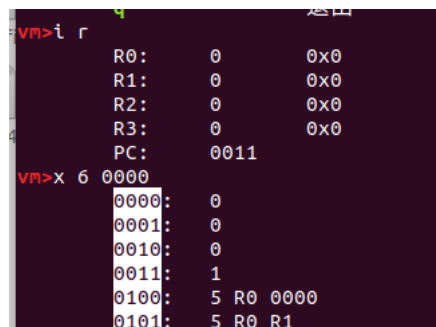
a:0000,b:0001,max:0010

- 1.等待输入; //输入数字a, 并保存在R0中
- 2.数据拷贝R0,0000; //将数字a的值保存在内存地址0000中
- 3.数据拷贝R0,R1; //将数字a的值保存在寄存器R1中
- 4.等待输入; //输入数字b, 并保存在R0中
- 5.数据拷贝R0,0001; //将数字b的值保存在内存地址0001中
- 6.做减法R1,R0; //做减法操作, 结果保存在R0中, 并根据结果对R3进行赋值
- 7.判断跳转 L1; //如果R0的值大于R1, 即b>a, 则跳转到L1, max=b
- 8.数据拷贝0000, 0010; //否则max=a
- 9.直接跳转L2
- 10.L1:数据拷贝0001,0010; //输出最大值
- 11.L2: 数据拷贝0010,1111 //max送至显存输出
- 12.停机 //程序执行完成

- 分析过程

第一次输入大于第二次输入

1. 输入 `i r`, 可以查看各个寄存器的值, 而输入 `x 6 0000` 则表示查看从0000开始的连续6个内存地址值



The screenshot shows a debugger window with the following content:

```

vm>i r
R0: 0 0x0
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0011
vm>x 6 0000
0000: 0
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1

```

2. 此时运行的指令是“1”, 表示等待输入, 输入一个值4, 在输入完成后将此数值存至R0寄存器, 运行完成后, 再运行 `i r` 指令, 就可以看到输入的值4确实是已经存在R0寄存器中, 每个寄存器的值都用十进制和十六进制表示

```
vm>si
4
0100: 5 R0 0000
vm>x 6 0000
0000: 0
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>i r
R0: 4 0x4
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0100
```

3. 继续使用si指令单步执行指令 5 R0 0000，表示将R0寄存器中的值3传送到内存地址0000处

```
vm>si
0101: 5 R0 R1
vm>x 6 0000
0000: 00000100
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>i r
R0: 4 0x4
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0101
```

4. 继续si执行下一条指令 5 R0 R1，表示将R0寄存器中的值3传送到内存地址R1处

```
vm>si
0110: 1
vm>i r
R0: 4 0x4
R1: 4 0x4
R2: 0 0x0
R3: 0 0x0
PC: 0110
```

5. 然后运行的指令是“1”，表示等待输入，输入一个值3，在输入完成后将此数值存至R0寄存器，运行完成后，再运行i r指令，就可以看到输入的值3确实是已经存在R0寄存器中，每个寄存器的值都用十进制和十六进制表示

```
vm>si
0111: 5 R0 0001
vm>i r
R0: 3 0x3
R1: 4 0x4
R2: 0 0x0
R3: 0 0x0
PC: 0111
```

6. 继续si执行下一条指令 5 R0 0001，表示将R0寄存器中的值3传送到内存地址0001处

```
vm>si
1000: 3 R1 R0
vm>x 6 0000
0000: 00000100
0001: 00000011
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>i r
R0: 3 0x3
R1: 4 0x4
R2: 0 0x0
R3: 0 0x0
```


7. 继续si执行下一条指令 `3 R1 R0`，表示将R0的值减去R1的值，结果放于R0中，结果小于0时，R3中赋值为-1

```

PC: 1000
vm>si
1001: 6 3
vm>i r
R0: -1 0xffffffff
R1: 4 0x4
R2: 0 0x0
R3: -1 0xffffffff
PC: 1001

```

8. 继续si执行下一条指令 `6 3`，即有条件跳转，如果R3的值为-1，执行下一条语句 `7 2`

```

1011: 7 2
vm>x 6 0000
0000: 00000100
0001: 00000011
0010: 00000100
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>i r
R0: -1 0xffffffff
R1: 4 0x4
R2: 0 0x0
R3: -1 0xffffffff
PC: 1011

```

9. 继续使用si指令单步执行指令 `7 2`，直接跳转当前指令+2条指令处执行

```

PC: 1011
vm>si
1101: 5 0010 R0
vm>i r
R0: -1 0xffffffff
R1: 4 0x4
R2: 0 0x0
R3: -1 0xffffffff
PC: 1101

```

10. 继续使用si指令单步执行指令 `5 0010 R0`，表示将0010寄存器中的值传送到内存地址R0处

```

vm>x 6 0000
0000: 00000100
0001: 00000011
0010: 00000100
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>i r
R0: 4 0x4
R1: 4 0x4
R2: 0 0x0
R3: -1 0xffffffff
PC: 1110

```

11. 继续使用si指令单步执行指令 `8 R0`，输出R0的值为4，即第二次输入的值

```

PC: 1110
vm>si
4
1111: 0

```

12. 程序执行完毕后，可以使用q退出

第一次输入小于第二次输入

1. 输入 `i r`，可以查看各个寄存器的值，而输入 `x 6 0000`则表示查看从0000开始的连续6个内存地址值

```

vm>i r
R0: 0 0x0
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0011
vm>x 6 0000
0000: 0
0001: 0
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1

```

2. 此时运行的指令是“1”，表示等待输入，输入一个值3，在输入完成后将此数值存至R0寄存器，运行完成后，再运行i r指令，就可以看到输入的值3确实是已经存在R0寄存器中，每个寄存器的值都用十进制和十六进制表示

```

vm>si
3
0100: 5 R0 0000
vm>si
0101: 5 R0 R1
vm>i r
R0: 3 0x3
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 0101

```

3. 继续使用si指令单步执行指令 5 R0 0000，表示将R0寄存器中的值4传送到内存地址0000处
4. 继续si执行下一条指令 5 R0 R1，表示将R0寄存器中的值4传送到内存地址R1处
5. 然后运行的指令是“1”，表示等待输入，输入一个值4，在输入完成后将此数值存至R0寄存器，运行完成后，再运行i r指令，就可以看到输入的值4确实是已经存在R0寄存器中，每个寄存器的值都用十进制和十六进制表示

```

PC: 0110
vm>si
4
0111: 5 R0 0001
vm>i r
R0: 4 0x4
R1: 3 0x3
R2: 0 0x0
R3: 0 0x0
PC: 0111

```

6. 继续si执行下一条指令 5 R0 0001，表示将R0寄存器中的值4传送到内存地址0001处

```

vm>x 6 0000
0000: 00000011
0001: 00000100
0010: 0
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
vm>si

```

7. 继续si执行下一条指令 3 R1 R0，表示将R0的值减去R1的值，结果放于R0中，结果大于0时，R3中赋值为1

```

PC: 1000
vm>si
1001: 6 3
vm>i r
R0: 1 0x1
R1: 3 0x3
R2: 0 0x0
R3: 1 0x1
PC: 1001

```

8. 继续si执行下一条指令 6 3，即有条件跳转，如果R3的值为1，则需要向前或向后跳转，此时跳转的值为3，则表示需要向后跳转3条指令，再去执行 5 0001 0010 指令

```
VM>si
1100: 5 0001 0010
VM>i r
R0: 1 0x1
R1: 3 0x3
R2: 0 0x0
R3: 1 0x1
PC: 1100
```

- 9. 继续使用si指令单步执行指令 5 0001 0010，表示将0001寄存器中的值传送到内存地址0010处
- 10. 继续使用si指令单步执行指令 5 0010 R0，表示将0010寄存器中的值传送到内存地址R0处

```
VM>x 6 0000
0000: 00000011
0001: 00000100
0010: 00000100
0011: 1
0100: 5 R0 0000
0101: 5 R0 R1
```

```
PC: 1101
VM>si
1110: 8 R0
VM>i r
R0: 4 0x4
R1: 3 0x3
R2: 0 0x0
R3: 1 0x1
PC: 1110
```

- 11. 继续使用si指令单步执行指令 8 R0，输出R0的值为4，即第二次输入的值

```
PC: 1110
VM>si
4
1111: 0
```

- 12. 程序执行完毕后，可以使用q退出

3.config除法

使用 ./vm64 3.config 来运行原型机的模拟器，其中 3.config 文件为配置文件，一共有四行，其格式如下：

```
1 5 //表示原型机的地址为5位，即内存有20个字节
2 3 //表示数据段为3个字节——00000, 00001, 00010
3 00011 //表示从第四个字节，也就是00011开始存放指令
4 c.txt //表示指令内容在c.txt中
```

指令内容 c.txt

```
1 1
2 5 R0 00000
3 1
4 5 00000 00001
5 5 00001 R1
6 3 R0 R1
7 6 2
8 7 7
9 5 00010 R2
10 4 1 R3
11 2 R3 R2
12 5 R2 00010
```

```
13 5 R1 00001
14 7 -9
15 5 00001 R1
16 5 R0 R2
17 3 R1 R2
18 6 5
19 5 00010 R2
20 4 1 R3
21 2 R3 R2
22 5 R2 00010
23 5 00010 R1
24 8 R1
25 0
```

• 分析过程

1. 输入指令1，即等待输入一个整数值4

```
vm>si
4
vm>i r
00100: 5 R0 00000
R0: 4 0x4
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 00100
```

2. 5 R0 00000 传送指令，其中A, B为寄存器编号或内存地址意义是将R0处的值传送到00000处

```
vm>x 6 00000
00000: 00000100
00001: 0
00010: 0
00011: 1
00100: 5 R0 00000
00101: 1
```

3. 输入指令1，即等待输入一个整数值2

```
vm>si
2
vm>i r
00110: 5 00000 00001
R0: 2 0x2
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 00110
```

4. 5 00000 00001 传送指令，其中A, B为寄存器编号或内存地址意义是将00000处的值传送到00001处

```
vm>si
vm>i r
00111: 5 00001 R1
R0: 2 0x2
R1: 0 0x0
R2: 0 0x0
R3: 0 0x0
PC: 00111
vm>x 6 00000
00000: 00000100
00001: 00000100
00010: 0
00011: 1
00100: 5 R0 00000
00101: 1
```

5. 5 00001 R1 传送指令，其中A, B为寄存器编号或内存地址意义是将00001处的值传送到R1处

```

vm>si
01000: 3 R0 R1
vm>i r
R0: 2 0x2
R1: 4 0x4
R2: 0 0x0
R3: 0 0x0
PC: 01000
vm>x 6 00000
00000: 00000100
00001: 00000100
00010: 0
00011: 1
00100: 5 R0 00000
00101: 1

```

6. 3 R0 R1 减法指令Ra, Rb。其意义是将寄存器Rb的值减去和Ra中的值，结果放到Rb中，这两个寄存器不能为R3，当结果大于0时，R3中赋值为1，当结果小于0时，R3中赋值为-1，当结果等于0时，R3中赋值为0。4-2=2>0，R3赋值为1

```

vm>si
01001: 6 2
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 0 0x0
R3: 1 0x1
PC: 01001

```

7. 6 2 判断跳转指令。其中 bias 为一个整数（可以为负），意义是如果R3的值为1，则跳转当前指令+bias条指令处执行，否则执行下一条指令。因此跳转到 5 00010 R2

```

vm>si
01011: 5 00010 R2
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 0 0x0
R3: 1 0x1
PC: 01011

```

8. 5 00010 R2 传送指令，其中A, B为寄存器编号或内存地址意义是将00010处的值传送至R2处

```

PC: 01011
vm>si
01100: 4 1 R3
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 0 0x0
R3: 1 0x1
PC: 01100

```

9. 4 1 R3 寄存器直接赋值指令。其中Ra为寄存器名称，\$1为常数，意义是将常数值\$1直接放到寄存器R3中

```

PC: 01100
m>si
01101: 2 R3 R2
m>i r
R0: 2 0x2
R1: 2 0x2
R2: 0 0x0
R3: 1 0x1
PC: 01101

```

10. 2 R3 R2 加法指令，将R3和R2的值相加，结果放在R2中

```

vm>si
01110: 5 R2 00010
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 1 0x1
R3: 1 0x1
PC: 01110

```

11. 5 R2 00010 传送指令，其中A, B为寄存器编号或内存地址意义是将R2处的值传送至00010处

```
PC: 01110
vm>si
01111: 5 R1 00001
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 1 0x1
R3: 1 0x1
PC: 01111
vm>x 6 00000
00000: 00000100
00001: 00000100
00010: 00000001
00011: 1
00100: 5 R0 00000
00101: 1
```

12. 5 R1 00001 传送指令，其中A, B为寄存器编号或内存地址意义是将R1处的值传送至00001处

```
vm>si
10000: 7 -9
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 1 0x1
R3: 1 0x1
PC: 10000
vm>x 6 00000
00000: 00000100
00001: 00000010
00010: 00000001
00011: 1
00100: 5 R0 00000
00101: 1
```

13. 7 -9 直接跳转指令。其中bias为一个整数（可以为负），直接跳转当前指令+bias条指令处执行

```
vm>si
00111: 5 00001 R1
vm>i r
R0: 2 0x2
R1: 2 0x2
R2: 1 0x1
R3: 1 0x1
PC: 00111
```

14. 继续执行，最后输出2

```
vm>si
01000: 3 R0 R1
vm>si
01001: 6 2
vm>si
01010: 7 7
vm>si
10001: 5 00001 R1
vm>si
10010: 5 R0 R2
vm>si
10011: 3 R1 R2
vm>si
10100: 6 5
vm>si
10101: 5 00010 R2
vm>si
10110: 4 1 R3
vm>si
10111: 2 R3 R2
vm>si
11000: 5 R2 00010
vm>si
11001: 5 00010 R1
vm>si
11010: 8 R1
vm>si
2
11011: 0vm>
```

- 运行过程

```
00011: 1
vm>si
4
00100: 5 R0 00000
vm>si
00101: 1
vm>si
2
00110: 5 00000 00001
vm>si
00111: 5 00001 R1
vm>si
01000: 3 R0 R1
vm>si
01001: 6 2
vm>si
01011: 5 00010 R2
vm>si
01100: 4 1 R3
vm>si
01101: 2 R3 R2
vm>si
```

```
vm>si
01101: 2 R3 R2
vm>si
01110: 5 R2 00010
vm>si
01111: 5 R1 00001
vm>si
10000: 7 -9
vm>si
00111: 5 00001 R1
vm>si
01000: 3 R0 R1
vm>si
01001: 6 2
vm>si
01010: 7 7
vm>si
10001: 5 00001 R1
vm>si
10010: 5 R0 R2
vm>si
10011: 3 R1 R2
vm>si
10100: 6 5
```

```
vm>si
10100: 6 5
vm>si
10101: 5 00010 R2
vm>si
10110: 4 1 R3
vm>si
10111: 2 R3 R2
vm>si
11000: 5 R2 00010
vm>si
11001: 5 00010 R1
vm>si
11010: 8 R1
vm>si
2
11011: 0 vm>
```

思考问题

(1)

- 如果基于这些指令实现两个整数的乘法与除法？

乘法

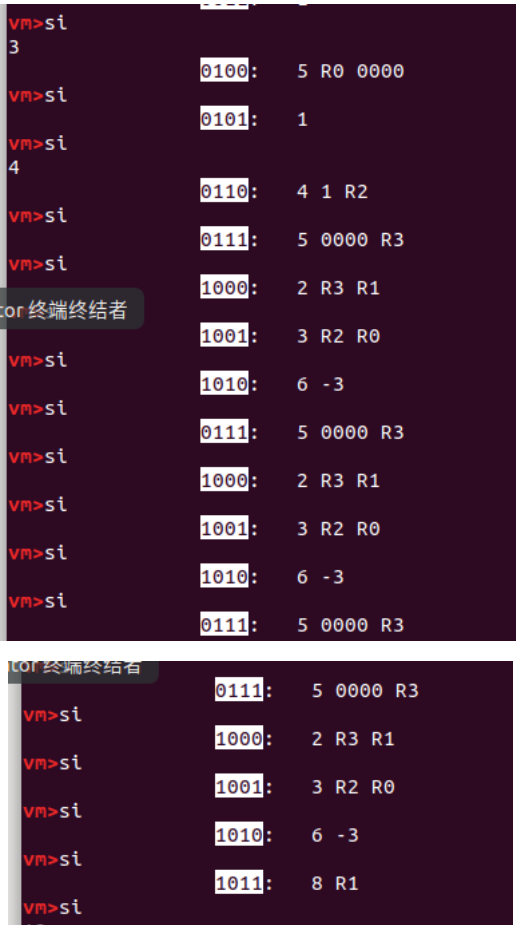
使用 `./vm64 4.config` 来运行原型机的模拟器，其中 `4.config` 文件为配置文件，一共有四行，其格式如下：

```
1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000，0001，0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 d.txt //表示指令内容在d.txt中
```

指令内容 `d.txt`

```
1 1
2 5 R0 0000 //输入乘数，并保存在内存地址0000中
3 1 //输入另一个乘数，保存在R0中
4 4 1 R2 //将R2寄存器赋值为1
5 5 0000 R3 //从地址0000中取出乘数的值5
6 2 R3, R1 //将R1的值与R3的值相加，结果保存在R1中
7 3 R2, R0 //被乘数减1
8 6 -3 //如果被乘数不为0，则结果还需要再加一次乘数
9 8 R1 //被乘数已经为0，此时R1中即为两数相乘的结果，输出此结果
10 0 //停机
```

• 分析过程



除法

使用两种不同的指令如下

• 法一

使用 `./vm64 2.config` 来运行原型机的模拟器，其中 `2.config` 文件为配置文件，一共有四行，其格式如下：

```
1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000, 0001, 0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 e.txt //表示指令内容在e.txt中
```

指令内容 `e.txt`

```
1 1
2 5 R0 0000
3 1
4 4 0 R1
5 4 1 R3
6 5 0000 R2
7 2 R3 R1
8 3 R0 R2
9 6 -2
10 2 R3 R1
11 8 R1
12 0
```

• 运行结果

```
初始化内存.....OK!
初始化寄存器.....OK!
将指令装配至内存.....OK!
准备执行指令，将要执行的地址及指令为：
0011: 1

vm>si
4
vm>si
0100: 5 R0 0000
vm>si
0101: 1
vm>si
0110: 4 0 R1
vm>si
0111: 4 1 R3
vm>si
1000: 5 0000 R2
vm>si
1001: 2 R3 R1
vm>si
1010: 3 R0 R2
vm>si
1011: 6 -2
vm>si
1001: 2 R3 R1
vm>si
1010: 3 R0 R2
vm>si
1011: 6 -2
vm>si
1100: 2 R3 R1
vm>si
1101: 8 R1
vm>si
2
1110: 0vm>
```

MobaTextEditor

File Edit Search View

e.txt x 2.config

```
1 1
2 5 R0 0000
3 1
4 4 0 R1
5 4 1 R3
6 5 0000 R2
7 2 R3 R1
8 3 R0 R2
9 6 -2
10 2 R3 R1
11 8 R1
12 0
```

• 法二

使用 `./vm64 3.config` 来运行原型机的模拟器，其中 `3.config` 文件为配置文件，一共有四行，其格式如下：

```

1 5 //表示原型机的地址为5位，即内存有20个字节
2 3 //表示数据段为3个字节——00000, 00001, 00010
3 00011 //表示从第四个字节，也就是00011开始存放指令
4 c.txt //表示指令内容在c.txt中

```

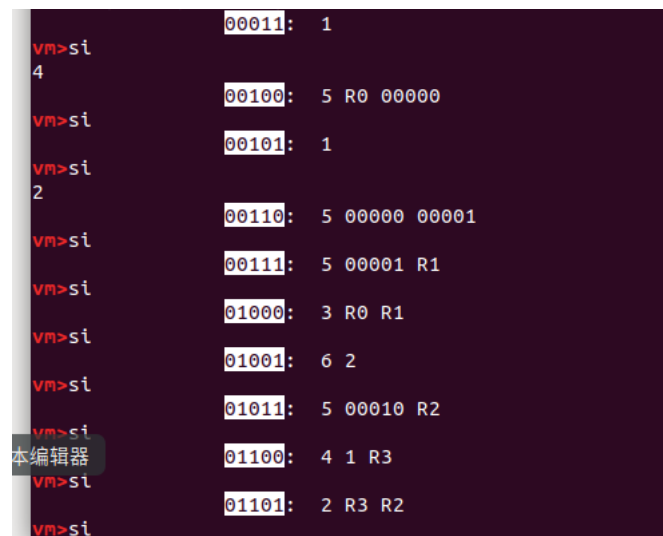
指令内容 c.txt

```

1 1
2 5 R0 00000
3 1
4 5 00000 00001
5 5 00001 R1
6 3 R0 R1
7 6 2
8 7 7
9 5 00010 R2
10 4 1 R3
11 2 R3 R2
12 5 R2 00010
13 5 R1 00001
14 7 -9
15 5 00001 R1
16 5 R0 R2
17 3 R1 R2
18 6 5
19 5 00010 R2
20 4 1 R3
21 2 R3 R2
22 5 R2 00010
23 5 00010 R1
24 8 R1
25 0

```

• 分析过程



```

vm>si 00011: 1
4
vm>si 00100: 5 R0 00000
vm>si 00101: 1
2
vm>si 00110: 5 00000 00001
vm>si 00111: 5 00001 R1
vm>si 01000: 3 R0 R1
vm>si 01001: 6 2
vm>si 01011: 5 00010 R2
本编辑器 vm>si 01100: 4 1 R3
vm>si 01101: 2 R3 R2
vm>si

```

```

VM>sl      01101: 2 R3 R2
VM>si      01110: 5 R2 00010
VM>si      01111: 5 R1 00001
VM>si      10000: 7 -9
VM>si      00111: 5 00001 R1
VM>si      01000: 3 R0 R1
VM>si      01001: 6 2
VM>si      01010: 7 7
VM>si      10001: 5 00001 R1
VM>si      10010: 5 R0 R2
VM>si      10011: 3 R1 R2
VM>si      10100: 6 5

VM>si      10100: 6 5
VM>si      10101: 5 00010 R2
VM>si      10110: 4 1 R3
VM>si      10111: 2 R3 R2
VM>si      11000: 5 R2 00010
VM>si      11001: 5 00010 R1
VM>si      11010: 8 R1
2
VM>si      11011: 0 VM>

```

(2)

- 原型机的指令集是否完备？如果是，那么如何证明（提示：搜索并阅读“可计算性理论”）？如果不是，那么要增加哪些指令？

原型机的指令集是计算机中央处理器机器码所使用的指令的集合以及其背后的寄存器体系、总线设计等逻辑框架。指令集可以分为复杂指令集（CISC）和精简指令集（RISC）。

可计算性理论是计算理论的一个分支，研究在不同的计算模型下哪些算法问题能够被解决。可计算性理论中有一个重要概念是图灵机，它是一种抽象的计算模型，可以模拟任何有效的算法。

如果一个问题可以用图灵机解决，那么它就是可计算的。如果一个问题不能用图灵机解决，那么它就是不可计算的。因此，要判断原型机的指令集是否完备，就要看它能否模拟图灵机或者与图灵机等价。

假设原型机的指令集是RISC类型的，也就是说它只有一些基本的指令，比如整数的加减乘除、数据的读写、条件跳转等。

如果我们想用原型机来计算一个矩阵的乘法，我们就需要用多条基本指令来组合实现这个功能。例如，我们可以先用读取指令从内存中读取两个矩阵的元素到寄存器中，然后用乘法和加法指令来计算每个元素的乘积和求和，最后用写入指令把结果存储到内存中。

这样，我们就用原型机的指令集实现了一个复杂的功能。如果原型机的指令集能够模拟图灵机或者与图灵机等价，那么它就是完备的。

假设原型机的指令集是CISC类型的，也就是说它有一些复杂的指令，比如字符串处理、浮点运算、矩阵乘法等。如果我们想用原型机来计算一个字符串的长度，我们就可以直接用字符串处理指令来实现这个功能。

例如，我们可以用读取指令从内存中读取一个字符串到寄存器中，然后用字符串处理指令来计算它的长度，并把结果存储到另一个寄存器中。这样，我们就用原型机的一条复杂指令实现了一个简单的功能。

可以用寄存器直接赋值指令和传送指令来模拟图灵机的读写头，用输入一个整数和输出指令来模拟图灵机的输入输出带，用加法指令和减法指令来模拟图灵机的移动头，用判断跳转指令和直接跳转指令来模拟图灵机的状态转换。

因此，原型机I的指令集是完备的。

实验1.2 原型机II-扩充指令集

实验目的

- 理解指令集结构及其作用；
- 理解计算机的运行过程，对指令集进行修改；

实验资源

操作系统：Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-39-generic x86_64)

实验准备

- 阅读教材，掌握冯诺伊曼体系的相关内容；
- 学习《最小系统与原型机I》内容，完成实验1.1

练习内容

乘法

思路一

最后的一个函数ExecuteInstruction即为CPU的指令执行过程，其中strncpy即表示从内存中取出一条指令（每条指令为字符串型，占20个字节），然后用split函数来对指令进行分割，再根据分割后的结果进行判断指令类型，调用前面的各种函数来进行操作

对指令集进行扩充，增加一条乘法指令，其格式为 9 Ra Rb，即将寄存器Ra的值与寄存器Rb的值相乘，结果放在Rb寄存器中，因此需要增加一个ExecuteMul函数

在ExecuteInstruction增加一个判断分支，从而能够识别此条指令

- 修改后的代码cpu.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 struct machine_info
5 {
6     int memory_num; //内存大小
7     int data_bytes; //数据段大小，从0开始
8     char start_address[20]; //第一条指令的地址
```

```
9     char instruct_file_name[20]; //指令文件名称
10 };
11 extern struct machine_info *pmi;
12 extern char * memory; //内存
13 extern char R0,R1,R2,R3; //寄存器
14 extern unsigned int PC;
15 void ExecuteAdd(char source[],char dest[],int *result)
16 {
17     char op;
18     if(0==strcmp(source,"R0"))
19         op=R0;
20     else if(0==strcmp(source,"R1"))
21         op=R1;
22     else if(0==strcmp(source,"R2"))
23         op=R2;
24     else if(0==strcmp(source,"R3"))
25         op=R3;
26     else
27         *result=-1;
28     if(0==strcmp(dest,"R0\n"))
29         R0+=op;
30     else if(0==strcmp(dest,"R1\n"))
31         R1+=op;
32     else if(0==strcmp(dest,"R2\n"))
33         R2+=op;
34     else if(0==strcmp(dest,"R3\n"))
35         R3+=op;
36     else
37         *result=-1;
38 }
39 }
40 void ExecuteSub(char source[],char dest[],int *result)
41 {
42     char op;
43     if(0==strcmp(source,"R0"))
44         op=R0;
45     else if(0==strcmp(source,"R1"))
46         op=R1;
47     else if(0==strcmp(source,"R2"))
48         op=R2;
49     else
50         *result=-1;
51     if(0==strcmp(dest,"R0\n"))
52     {
53         R0-=op;
54         if(R0>0) R3=1;
55         else if(R0<0) R3=-1;
56         else R3=0;
57     }
58     else if(0==strcmp(dest,"R1\n"))
59     {
60         R1-=op;
61         if(R1>0) R3=1;
62         else if(R1<0) R3=-1;
```

```
63         else R3=0;
64     }
65     else if(0==strcmp(dest,"R2\n"))
66     {
67         R2-=op;
68         if(R2>0) R3=1;
69         else if(R2<0) R3=-1;
70         else R3=0;
71     }
72     else
73         *result=-1;
74
75
76 }
77 void ExecuteMov(char source[],char dest[],int *result)
78 {
79     char op;
80     op=atoi(source);
81     if(0==strcmp(dest,"R0\n"))
82         R0=op;
83     else if(0==strcmp(dest,"R1\n"))
84         R1=op;
85     else if(0==strcmp(dest,"R2\n"))
86         R2=op;
87     else if(0==strcmp(dest,"R3\n"))
88         R3=op;
89     else
90         *result=-1;
91
92
93 }
94 void ExecuteCopy(char source[],char dest[],int *result)
95 {
96     char op;
97     char buf[9];
98     int loop;
99     int i;
100     if(0==strcmp(source,"R0"))
101         op=R0;
102     else if(0==strcmp(source,"R1"))
103         op=R1;
104     else if(0==strcmp(source,"R2"))
105         op=R2;
106     else if(0==strcmp(source,"R3"))
107         op=R3;
108     else
109     {
110         i=BCharsToInt(pmi->memory_num,source);
111         // printf("%d\n",i);
112         strncpy(buf,memory+(i-1)*20,8);
113         buf[8]='\0';
114         op=BCharsToInt(8,buf);
115         // printf("%s,%s,%s,%d\n",source,memory-20,buf,op);
116     }
```

```
117 // printf("%d\n",op);
118
119 if(0==strcmp(dest,"R0\n"))
120     R0=op;
121 else if(0==strcmp(dest,"R1\n"))
122     R1=op;
123 else if(0==strcmp(dest,"R2\n"))
124     R2=op;
125 else if(0==strcmp(dest,"R3\n"))
126     R3=op;
127 else
128 {
129     IntToBchars(op,8,buf);
130     i=BCharsToInt(pmi->memory_num,dest);
131     for(loop=0;loop<8;loop++)
132         memory[(i-1)*20+loop]=buf[loop];
133     memory[(i-1)*20+8]='\n';
134     memory[(i-1)*20+9]='\0';
135 }
136
137 }
138 void ExecuteOutput(char source[],int *result)
139 {
140     char op;
141     *result=0;
142     if(0==strcmp(source,"R0\n"))
143         op=R0;
144     else if(0==strcmp(source,"R1\n"))
145         op=R1;
146     else if(0==strcmp(source,"R2\n"))
147         op=R2;
148     else if(0==strcmp(source,"R3\n"))
149         op=R3;
150     else
151         *result=-1;
152     if(*result!=-1)
153         printf("%d\n",op);
154 }
155
156 void ExecuteMul(char source[],char dest[],int *result)
157 {
158     char op;
159     if(0==strcmp(source,"R0"))
160         op=R0;
161     else if(0==strcmp(source,"R1"))
162         op=R1;
163     else if(0==strcmp(source,"R2"))
164         op=R2;
165     else if(0==strcmp(source,"R3"))
166         op=R3;
167     else
168         *result=-1;
169     if(0==strcmp(dest,"R0\n"))
170         R0*=op;
```

```
171     else if(0==strcmp(dest,"R1\n"))
172         R1*=op;
173     else if(0==strcmp(dest,"R2\n"))
174         R2*=op;
175     else if(0==strcmp(dest,"R3\n"))
176         R3*=op;
177     else
178         *result=-1;
179
180 }
181 void ExecuteDiv(char source[],char dest[],int *result)
182 {
183     char op;
184     if(0==strcmp(source,"R0"))
185         op=R0;
186     else if(0==strcmp(source,"R1"))
187         op=R1;
188     else if(0==strcmp(source,"R2"))
189         op=R2;
190     else if(0==strcmp(source,"R3"))
191         op=R3;
192     else
193         *result=-1;
194     if(0==strcmp(dest,"R0\n"))
195         R0/=op;
196     else if(0==strcmp(dest,"R1\n"))
197         R1/=op;
198     else if(0==strcmp(dest,"R2\n"))
199         R2/=op;
200     else if(0==strcmp(dest,"R3\n"))
201         R3/=op;
202     else
203         *result=-1;
204
205 }
206 void ExecuteInstruction(int * result)
207 {
208     //取值
209     char instruction_buffer[20];
210     char *revbuf[8] = {0};
211     int num;
212     int op1,op2,op3;
213     int i;
214     strncpy(instruction_buffer,memory+(PC-1)*20,20);
215     // printf("%s\n",instruction_buffer);
216     switch(instruction_buffer[0])
217     {
218         case '0':
219             *result=0;
220             PC++;
221             break;        //停机
222         case '1':
223             scanf("%d",&i);
224             R0=(unsigned char)i;
```



```
225     safe_flush(stdin);
226     *result=1;
227     PC++;
228     break;
229 case '2':          //加法
230     split(instruction_buffer," ",revbuf,&num);
231     if(3>num)
232         *result=-1; //出错
233     else
234         ExecuteAdd(revbuf[1],revbuf[2],result);
235     if(*result!=-1) *result=2;
236     PC++;
237     break;
238 case '3':          //减法
239     split(instruction_buffer," ",revbuf,&num);
240     if(3>num)
241         *result=-1; //出错
242     else
243         ExecuteSub(revbuf[1],revbuf[2],result);
244     if(*result!=-1) *result=3;
245     PC++;
246     break;
247 case '4':          //赋值
248     split(instruction_buffer," ",revbuf,&num);
249     if(3>num)
250         *result=-1; //出错
251     else
252         ExecuteMov(revbuf[1],revbuf[2],result);
253     if(*result!=-1) *result=4;
254     PC++;
255     break;
256 case '5':          //拷贝
257     split(instruction_buffer," ",revbuf,&num);
258 //     printf("%d\n",num);
259     if(3>num)
260         *result=-1; //出错
261     else
262         ExecuteCopy(revbuf[1],revbuf[2],result);
263     if(*result!=-1) *result=5;
264     PC++;
265     break;
266 case '6':          //判断跳转
267     split(instruction_buffer," ",revbuf,&num);
268     if(2!=num)
269         *result=-1; //出错
270     else
271     {
272         i=atoi(revbuf[1]);
273         if(R3==1) PC+=i;
274         else PC++;
275     };
276     if(*result!=-1) *result=6;
277 //     PC++;
278     break;
```

```
279     case '7':          //直接跳转
280         split(instruction_buffer," ",revbuf,&num);
281         if(2>num)
282             *result=-1; //出错
283         else
284         {
285             i=atoi(revbuf[1]);
286             PC+=i;
287         };
288         if(*result!=-1) *result=6;
289     //     PC++;
290     break;
291     case '8':
292         split(instruction_buffer," ",revbuf,&num);
293         if(2>num)
294             *result=-1; //出错
295         else
296             ExecuteOutput(revbuf[1],result);
297         if(*result!=-1) *result=8;
298     //     safe_flush(stdin);
299     PC++;
300     break;
301     case '9':          //乘法
302         split(instruction_buffer," ",revbuf,&num);
303         if(3>num)
304             *result=-1; //出错
305         else
306             ExecuteMul(revbuf[1],revbuf[2],result);
307         if(*result!=-1) *result=9;
308         PC++;
309         break;
310     case 'a':          //除法
311         split(instruction_buffer," ",revbuf,&num);
312         if(3>num)
313             *result=-1; //出错
314         else
315             ExecuteDiv(revbuf[1],revbuf[2],result);
316         if(*result!=-1) *result=10;
317         PC++;
318         break;
319     default:
320         *result=-1; //出错
321         break;
322
323 }
324
325 }
```

```

1      case '9':          //乘法
2          split(instruction_buffer," ",revbuf,&num);
3          if(3>num)
4              *result=-1; //出错
5          else
6              ExecuteMul(revbuf[1],revbuf[2],result);
7          if(*result!=-1) *result=9;
8          PC++;
9          break;

```

使用 `./vm64 1.config` 来运行原型机的模拟器，其中 `1.config` 文件为配置文件，一共有四行，其格式如下：

```

1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000, 0001, 0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 d.txt //表示指令内容在d.txt中

```

指令内容 `d.txt`

```

1 1
2 5 R0 R1
3 1
4 9 R0 R1
5 8 R1
6 0

```

• 分析过程

```

准备执行指令，将要执行的地址及指令为：
0011: 1
vm>si
3
0100: 5 R0 R1
vm>si
0101: 1
vm>si
4
0110: 9 R0 R1
vm>si
0111: 8 R1
vm>si
12
1000: 0
vm>

```

思路二

增加一个 `e.txt` 文件，基于原型机的指令完成了两个数的乘法操作，其基本思路是将乘法分解为加法，例如对于 $5*6$ ，执行6次加5的操作： $5*6=5+5+5+5+5+5$

使用 `./vm64 4.config` 来运行原型机的模拟器，其中 `4.config` 文件为配置文件，一共有四行，其格式如下：

```

1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000, 0001, 0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 e.txt //表示指令内容在e.txt中

```

指令内容 `e.txt`

```

1 1
2 5 R0 0000 //输入乘数，并保存在内存地址0000中
3 1 //输入另一个乘数，保存在R0中
4 4 1 R2 //将R2寄存器赋值为1
5 5 0000 R3 //从地址0000中取出乘数的值5
6 2 R3, R1 //将R1的值与R3的值相加，结果保存在R1中
7 3 R2, R0 //被乘数减1
8 6 -3 //如果被乘数不为0，则结果还需要再加一次乘数
9 8 R1 //被乘数已经为0，此时R1中即为两数相乘的结果，输出此结果
10 0 //停机

```

- 分析过程

准备执行指令，将要执行的地址及指令为：

```

0011: 1
vm>si
3
0100: 5 R0 0000
vm>si
0101: 1
vm>si
2
0110: 4 1 R2
vm>si
0111: 5 0000 R3
vm>si
1000: 2 R3 R1
vm>si
1001: 3 R2 R0
vm>si
1010: 6 -3
vm>si
0111: 5 0000 R3
vm>si
1000: 2 R3 R1
vm>si
1001: 3 R2 R0
vm>si
1010: 6 -3
vm>si
1011: 8 R1
vm>si
6
1100: 0
vm>si
指令执行完成，程序退出！

```

除法

```

1 case 'a': //除法
2     split(instruction_buffer, " ", revbuf, &num);
3     if(3>num)
4         *result=-1; //出错
5     else
6         Executediv(revbuf[1], revbuf[2], result);
7     if(*result!=-1) *result=10;
8     PC++;
9     break;

```

使用 `./vm64 1.config` 来运行原型机的模拟器，其中 `1.config` 文件为配置文件，一共有四行，其格式如下：

```

1 4 //表示原型机的地址为4位，即内存有16个字节
2 3 //表示数据段为3个字节——0000, 0001, 0010
3 0011 //表示从第四个字节，也就是0011开始存放指令
4 e.txt //表示指令内容在e.txt中

```

指令内容 e.txt

```
1 1
2 5 R0 R1
3 1
4 a R0 R1
5 8 R1
6 0
```



思考问题

(1)

- 原型机I与原型机II完成乘法和除法操作的方式有何不同？

指令不同

在原型机I中，乘法和除法通过原型机指令互相协调，乘法变为加法，除法变为减法，进行循环加减

在原型机II中，乘法和除法通过C语言代码直接实现乘除

1. 原型机I

乘法

```

1 1
2 5 R0 0000 //输入乘数，并保存在内存地址0000中
3 1 //输入另一个乘数，保存在R0中
4 4 1 R2 //将R2寄存器赋值为1
5 5 0000 R3 //从地址0000中取出乘数的值5
6 2 R3, R1 //将R1的值与R3的值相加，结果保存在R1中
7 3 R2, R0 //被乘数减1
8 6 -3 //如果被乘数不为0，则结果还需要再加一次乘数
9 8 R1 //被乘数已经为0，此时R1中即为两数相乘的结果，输出此结果
10 0 //停机

```

除法

```

1 1
2 5 R0 00000
3 1
4 5 00000 00001
5 5 00001 R1
6 3 R0 R1
7 6 2
8 7 7
9 5 00010 R2
10 4 1 R3
11 2 R3 R2
12 5 R2 00010
13 5 R1 00001
14 7 -9
15 5 00001 R1
16 5 R0 R2
17 3 R1 R2
18 6 5
19 5 00010 R2
20 4 1 R3
21 2 R3 R2
22 5 R2 00010
23 5 00010 R1
24 8 R1
25 0

```

2. 原型机II

乘法

```

1 1
2 5 R0 R1
3 1
4 9 R0 R1
5 8 R1
6 0

```

除法

```

1 1
2 5 R0 R1
3 1
4 a R0 R1
5 8 R1
6 0

```

(2)

- 在指令集中增加乘法、除法等指令时，原型机中需要增加代码，那么硬件实现上需要增加什么样的部件？

在指令集中增加乘法、除法等指令时，硬件实现上需要增加相应的运算器，比如乘法器、除法器或者乘除混合单元。这些运算器可以根据不同的算法设计，比如基于移位、逼近或者迭代等。另外，还需要考虑数据格式（定点或浮点）、精度、溢出处理等因素。

乘法和除法用到的电路元件主要有门电路、加法器、移位寄存器等。根据网络搜索结果，这里简单介绍一些常用的电路元件：

- 门电路是实现逻辑运算的基本单元，有与门、或门、非门、异或门等。门电路可以组合成半加器、全加器等更复杂的运算单元。
- 加法器是实现加法运算的电路，有半加器、全加器、并行加法器等。加法器可以用来求和部分积或商位。
- 移位寄存器是实现数据移位的电路，有左移寄存器、右移寄存器等。移位寄存器可以用来对乘数或被除数进行对齐或规格化。

乘法和除法的电路设计有以下几种方法：

- 乘法器可以采用**阵列乘法器**、**改进的booth编码乘法器**或者**Wallace tree压缩结构**等方式，提高运算效率和降低功耗。
- 除法器可以采用**定点运算**的方式，类似于十进制除法，每次取被除数的高几位数据与除数作比较，得到对应位的商和余数。也可以采用**带预缩放的除法**、**模除法器**、**分段除法器**或者**组合除法器设计（包括阵列除法器）**等方式，优化运算过程。

(3)

- 如果一台计算机只支持加法、减法操作，那么能否计算三角函数，对数函数？（提示：搜索并阅读“泰勒级数展开”等内容）

泰勒级数展开是一种将函数变成无穷级数的方法，如果级数收敛，那么就可以用级数来近似函数的值。一些常见的三角函数和对数函数都有泰勒级数展开式，例如：

$$\sin x = x - x^3/3! + x^5/5! - \dots$$

$$\ln(1+x) = x - x^2/2 + x^3/3 - \dots$$

因此，如果一台计算机只支持加法、减法操作，那么它可以用泰勒级数展开式来计算三角函数和对数函数的近似值，只要给定一个足够小的误差范围，并且保证级数收敛。

计算机可以用以下步骤来用泰勒级数计算三角函数和对数函数：

1. 选择一个合适的展开点，一般是0或者接近给定的自变量值。
2. 根据已知的泰勒级数展开式，计算出各项的系数和指数。
3. 用加法、减法操作，将各项相加，得到一个多项式近似。
4. 判断多项式近似的误差是否在可接受的范围内，如果不是，就增加更多的项，直到满足要求。

例如，如果要计算 $\sin(0.5)$ ，可以用以下步骤：

1. 选择展开点为0。
2. 根据 $\sin x = x - x^3/3! + x^5/5! - \dots$ ，得到系数为1, -1/3!, 1/5!, ..., 指数为1, 3, 5, ...
3. 将 $x = 0.5$ 代入多项式近似，得到 $\sin(0.5) \approx 0.5 - 0.5^3/3! + 0.5^5/5! - \dots$
4. 计算多项式近似的值，并与真实值比较。如果误差太大，就增加更多的项。例如：

$$\sin(0.5) \approx 0.4794 \text{ (只取前两项)}$$

$$\sin(0.5) \approx 0.4794 + 0.0026 = 0.482 \text{ (取前三项)}$$

$$\sin(0.5) \approx 0.482 - 8e-05 = 0.4819 \text{ (取前四项)}$$

...

真实值为 $\sin(0.5) = 0.4794$

因此，只取前两项就可以达到很高的精度。

(4)

- 对于某个需要完成的功能，如果既可以通过硬件上增加电路来实现，也可以通过其他已有指令的组合来实现，那么如何判断哪一种比较合适？（提示：搜索并阅读RISC与CISC）

CISC和RISC是两种不同的CPU指令集类型，它们有以下区别：

- CISC的设计思路是用一条指令完成一个复杂的基本功能，例如乘法、除法等；
- RISC的设计思路是一条指令完成一个基本“动作”，多条指令组合完成一个复杂的基本功能，例如加法、移位等；
- CISC的优点是指令数目少，编译器简单，程序长度短，节省存储空间；
- RISC的优点是指令执行速度快，流水线效率高，芯片面积小，功耗低；
- CISC的缺点是指令执行时间长，流水线难以实现或效率低，芯片面积大，功耗高；
- RISC的缺点是指令数目多，编译器复杂，程序长度长，占用存储空间；

选择哪一种更合适取决于具体的应用场景和需求。一般来说：

- 如果需要完成的功能比较简单或常见，并且对存储空间有限制，则CISC可能更合适；
- 如果需要完成的功能比较复杂或不常见，并且对执行速度和功耗有要求，则RISC可能更合适；

总结

实验中出现的问题

1. 权限不足

解决办法： `chmod 777 xx`（文件名称）

2. 实验1.1中 `3.config` 指令运行出现错误

后面老师进行了修改

心得体会

ubuntu使用还不够熟练，导致出现很多错误，不过通过自己上网查找从而解决这些问题

在调试指令的时候需要耐心，一步一步分析

通过这次实验，我更熟悉了linux的操作，并且能够分析指令，了解cpu.c的原理

要与同学和老师多交流，思维碰撞，找到解决方法