

# 中国科学技术大学计算机学院 《计算机系统概论》实验报告



**RISC-V LAB 01 : Bits Rotation**

姓名:王晨      学号:PE20060014

完成日期:2020 年 11 月 22 日

## 一、实验要求：

用RISC-V指令集重现二进制串的2bit(或任意位数)的翻转。

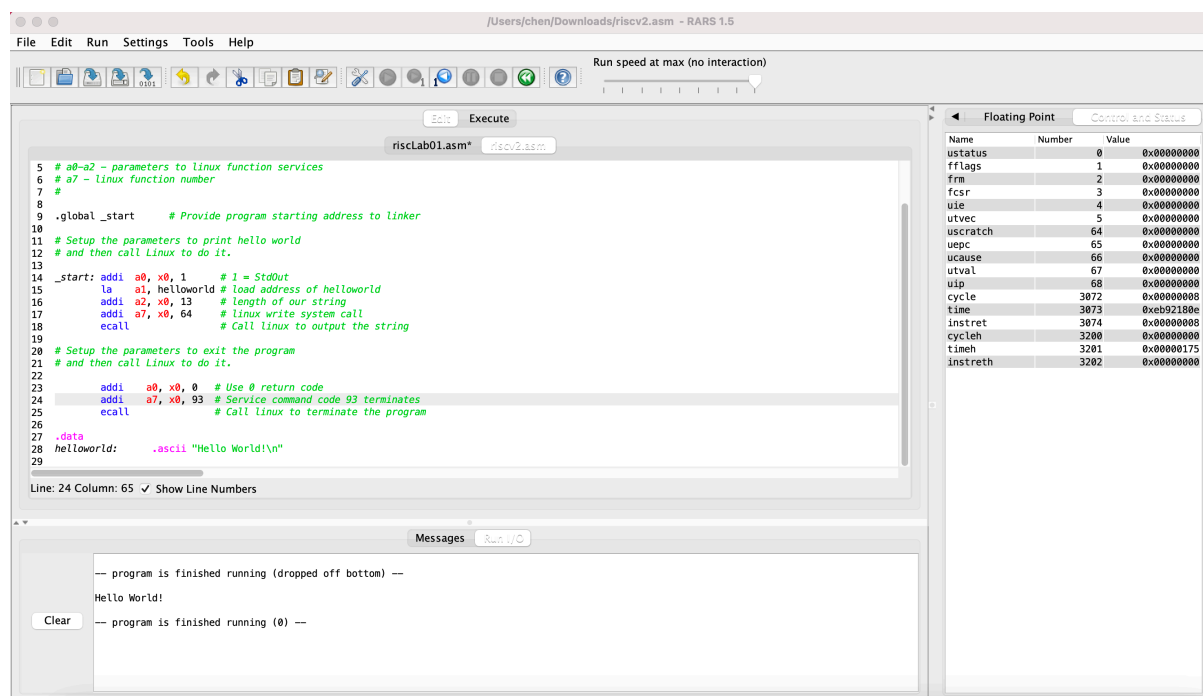
## 二、实验环境及搭建方法：

我分别在Mac环境和Linux环境搭建了RISC-V的编译和运行环境：

Mac OS Big Sur	Linux 20.04.1虚拟机
Rars 1.5 (RISC-V Simulator based on Java SDK)	Spike

### 1. Mac环境搭建

Mac的环境搭建非常简单,我在网上尝试了一些不同公司的嵌入式集成开发环境，最后发现了一个非常简单迷你RISC-V模拟器rars。[URL:https://github.com/TheThirdOne/rars](https://github.com/TheThirdOne/rars).根据说明文件安装即可。这是一个类似于LC3的模拟器，可以单步调试运行，监视寄存器和内存的状态等等，比较适合刚开始入门RISC-V汇编，因此本次实验选择了用此模拟器进行。安装完后可以看到界面如下，用一个HelloWorld.s的汇编文件测试一下，可以编译运行。



### 2. Linux环境的搭建

linux环境下需要安装risc-toolchain和spike，相对比较繁琐。根据文档一步一步编译安装即可。成功后可以使用一下命令测试spike是否能够运行：

```
chen@ubuntu:~/Desktop/ICS$ riscv32-unknown-elf-as -march=rv32im HelloWorld.s -o HelloWorld.o
```

```
chen@ubuntu:~/Desktop/ICS$ riscv32-unknown-elf-ld -o HelloWorld HelloWorld.o
```

```
chen@ubuntu:~/Desktop/ICS$ spike --isa=rv32im /home/chen/Desktop/RISCV/riscv-pk/build/pk HelloWorld
```

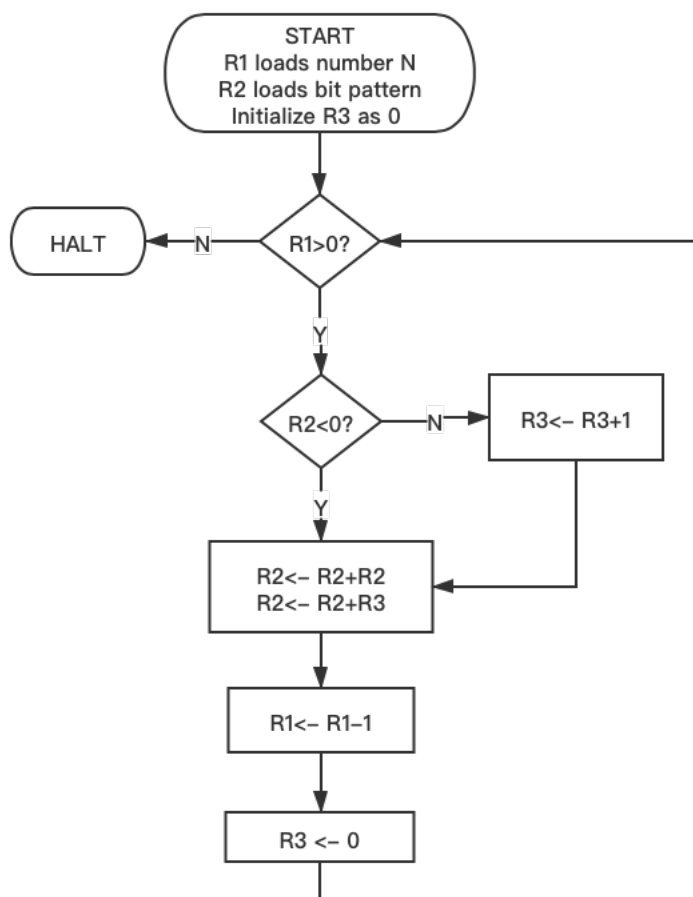
```
chen@ubuntu:~/Desktop/ICS$ riscv32-unknown-elf-as -march=rv32im HelloWorld.s -o Hello.o
chen@ubuntu:~/Desktop/ICS$ riscv32-unknown-elf-ld -o HelloWorld Hello.o
chen@ubuntu:~/Desktop/ICS$ spike --isa=rv32im /home/chen/Desktop/RISCV/riscv-pk/build/pk HelloWorld
bbl loader
Hello World!
chen@ubuntu:~/Desktop/ICS$
```

可以看到输出Hello World就成功了。

### 三、算法思路：

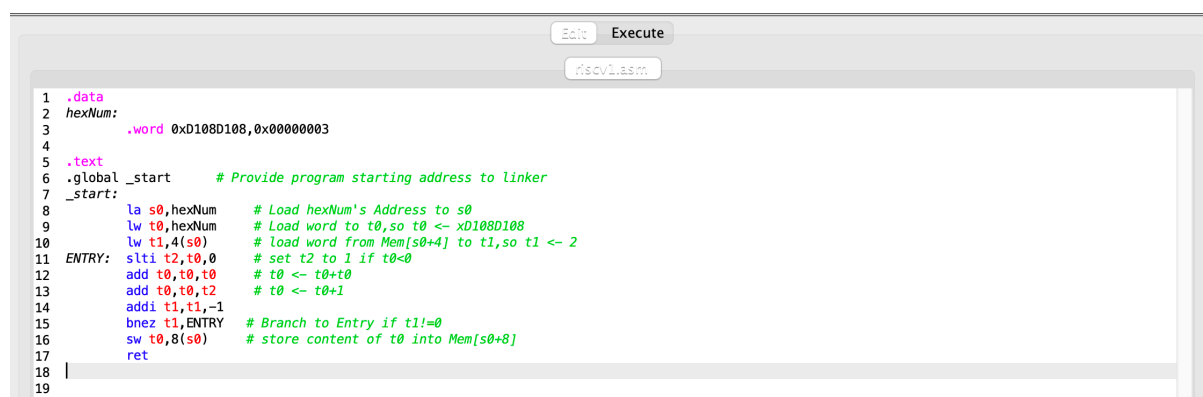
这里讲两种方法，第一种方法最简单，因为riscv直接提供shift指令，那么直接使用shift指令左移再加上二进制串首位即可。

第二种方法和之前用LC指令完成的方法是一样的，把shift指令拆分成用ADD加条件跳转的方式实现，流程图是基本一样的，只是指令集有了不同，另外一个word的长度变成了32位。本次实验报告采用这种基础的方法实现，具体实现在下一部分阐述。



LC3指令集下的算法流程示意图

### 三、程序代码及注释：



```
1 .data
2 hexNum:
3     .word 0xD108D108,0x00000003
4
5 .text
6 .global _start # Provide program starting address to linker
7 _start:
8     la s0,hexNum # Load hexNum's Address to s0
9     lw t0,hexNum # Load word to t0,so t0 <- xD108D108
10    lw t1,4(s0) # load word from Mem[s0+4] to t1,so t1 <- 2
11 ENTRY: slti t2,t0,0 # set t2 to 1 if t0<0
12    add t0,t0,t0 # t0 <- t0+t0
13    add t0,t0,t2 # t0 <- t0+t2
14    addi t1,t1,-1
15    bnez t1,ENTRY # Branch to Entry if t1!=0
16    sw t0,8(s0) # store content of t0 into Mem[s0+8]
17    ret
18
19
```

- ① .data
- ② hexNum:
- ③ .word 0xD108D108,0x00000003
- ④
- ⑤ .text
- ⑥ .global \_start # Provide program starting address to linker
- ⑦ \_start:
- ⑧ la s0,hexNum # Load hexNum's Address to s0
- ⑨ lw t0,hexNum # Load word to t0,so t0 <- xD108D108
- ⑩ lw t1,4(s0) # load word from Mem[s0+4] to t1,so t1 <- 3
- ⑪ ENTRY: slti t2,t0,0 # set t2 to 1 if t0<0
- ⑫ add t0,t0,t0 # t0 <- t0+t0
- ⑬ add t0,t0,t2 # t0 <- t0+t2
- ⑭ addi t1,t1,-1
- ⑮ bnez t1,ENTRY # Branch to Entry if t1!=0
- ⑯ sw t0,8(s0) # store content of t0 into Mem[s0+8]
- ⑰ ret # return

#### 解释：

1. 这里与LC指令所不同的是标记ENTRY的第11行，这里使用了slti指令，如果t0内的值小于0，也就是t0内的值首位是1，那么就将t2寄存器的值置成1，否则置成0，之后在第13行用add t0,t0,t2实现了首位的bit翻转到尾。

- 第15行使用了bnez指令，如果t1不等于0，那么跳转到循环入口ENTRY，否则就存储数据到memory了，这样的实现方式类似于C语言的do...while语句。
- hexNum段中定义了待操作的二进制串，操作次数，编译后可以看到它们在memory中的存储位置，最后操作完成后的结果(比如0x88468846)将通过第16行的sw指令存储到0x10010008的数据段中。

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xd108d108	0x00000003	0x88468846	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

## 四、测试用例和结果分析：

由于RISC-V字长32位，这里的测试用例使用了xD108D108，于LC3实验中使用xD108类似的：

- xD108D108左移翻转1位应该得到xA211A211
- 左移翻转2位应该得到x44234423
- 左移翻转3位应该得到x88468846
- 左移翻转32位应该仍然得到xD108D108
- FFFFFFFF和00000000左移翻转任意位均为其本身

1.单步执行到Line10的时候可以看到待操作的二进制串和操作次数都装载到寄存器中了。执行到第11行时，由于判断了 $t0 < 0$ ，因此t2被置成了1。

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xd108d108	0x00000003	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers			Floating Point	Control and Status
Name	Number	Value		
zero	0	0x00000000		
ra	1	0x00000000		
sp	2	0x7ffffefc		
gp	3	0x10008000		
tp	4	0x00000000		
t0	5	0xd108d108		
t1	6	0x00000003		
t2	7	0x00000001		
s0	8	0x10010000		



6.将initial bit pattern存为全1或全0的二进制串进行测试，N=任意数，结果也是正确的：

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xffffffff	0x00000000	0xffffffff	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

## 五、实验心得：

1. 本次实验相对比较简单，通过本次实验熟悉了RISCV模拟器的使用和调试方法，也熟悉了RISCV的一些基本操作指令。
2. 因为riscv直接提供shift指令，本实验可以直接使用shift指令完成，这样只需要几行，不需要循环。
3. 本实验所用的方法是比较简单的方法，也是比较通用的方法，时间复杂度为 $O(N)$ ，仅与移位翻转的位数N有关，代码也非常简短，因此综合来看这种方法还是比较好的。
4. 本次实验实现了任意位的左移翻转，结果是正确的，实验成功。实现了任意位的左移翻转后，就可以实现任意位的右移翻转，右移翻转1位，就相当于左移翻转31位。