

前言

了解计算机之自底向上式。大家好，我是later。本文会介绍一个能运行“2048”游戏的虚拟机，注：并不需要自己写 2048游戏，需要自己模拟 该游戏的运行环境。这是一个有意思的话题，什么叫做 游戏的运行环境呢？从上面给出的游戏链接，我们可以获取一个2048.obj，这就是要运行的2048游戏程序，但是我们不能将它运行在windows或者linux上，额，这是为什么呢？这里简单说就是 obj不是一个windows和linux的可执行程序，obj里面的指令不是 x86和arm等常见平台的，这个obj是需要运行到 咱们自己实现的虚拟机中的。就像下面这样：

Shell ▾

```
1 ./lc3 2048.obj
```

```
+-----+
|       |
|   4    32    64    2  |
|       |
|   2    16    4     16  |
|       |
|  16   128   512    4   |
|       |
|   2     8    64     2   |
|       |
+-----+

You lost :(
```

Note

注：本文内容参考 [Write your Own Virtual Machine \(jmeiners.com\)](https://jmeiners.com) 

目录

- [1 虚拟机介绍](#)
- [2 LC-3 架构](#)
- [3 LC-3汇编](#)
- [4 执行程序](#)
- [5 指令翻译](#)
- [6 指令备忘单](#)
- [7 trap routines](#)
- [8 LC-3 endianness](#)
- [9 内存映射寄存器](#)
- [10 优化](#)
- [11 总结](#)

1. 虚拟机介绍

如果要从虚拟化谈起，会 太太太 冗长了(实际是later对虚拟化也不是很了解)。本文所介绍的虚拟机实际就是一个Linux下面的可执行程序，它可以解析2048.obj这个二进制。进一步的，这个虚拟机的名字叫做：[lc3-vm](#) (little computer)，它就像是一个“计算机”，它模拟了一个cpu和一些硬件，可以做算术运算、读写内存、与I/O设备交互。再进一步，一个程序中有什么呢？有人说：代码段、数据段..... 嗯，later觉得这样的同学基础非常扎实！有些同学可能用的是java或者python，但归根结底，咱们写的程序经过编译被翻译成了二进制，这个二进制里面包含了一些数据，更重要的是，包含了一些"cpu（可能是物理的cpu，可能是软件模拟的cpu）"能执行的指令。在本文中的虚拟机能够执行 2048.obj 中的指令，额，其实这么说是没有错的，但是实际上有点怪怪的，因为lc3-vm是一个遵循 "[The IC-3 ISA](#)"的虚拟机，而2048.obj就是为了在 "The IC-3 ISA" 上运行的程序。

另外，后面later还会介绍其他的模拟器：能玩 超级马里奥 的NES 模拟器（NES模拟器模拟了咱们小时候的 小霸王电脑，这个小电脑里面有一颗 [6502cpu](#)）、运行 riscv 程序的模拟器、大名鼎鼎的 [QEMU](#)（谈探索linux的利器）。

为什么要写虚拟机

写虚拟机可以更深入的了解计算机的工作原理。

所需要做的工作

就像前言所说的那样我们需要写一个虚拟机，大概就是500行左右的c程序。注：2048游戏不需要我们写，我们就把这个2048.obj当做是运行在虚拟机中的程序就好，除了2048还有其他的游

戏，例如：[justinmeiners/lc3-rogue: Roguelike tunnel generator in LC3 assembly. \(Homework\) \(github.com\)](#) 



环境

本文使用wsl2 ubuntu20.04作为开发环境。

2. LC-3 架构

LC-3相比 x86 架构而言，指令集非常简单，但包含了几乎所有的现代CPU的核心内容。

LC-3相关资源

一些关于 LC-3 的相关资源：[Introduction to Computing Systems \(mheducation.com\)](#) , 有很多国外学校的本科生计算机入门课程用到了LC-3(或者lc3)。推荐一本书：[Introduction To Computing Systems: From Bits & Gates To C/C++ & Beyond, Third Edition \(icourse.club\)](#) 。本书类似一本计算机的教科书，其实它就是。是非常好的入门资料，全书依据 **自底向上** 的方法，从逻辑门电路开始讲起，到冯诺依曼，再到LC-3架构，最后c语言等等。

Memory

首先，我们需要模拟计算机的基本硬件组件。LC-3是基于冯诺依曼架构的，我们先从 memory 开始。

LC-3的地址空间是0 ~ 65536，也就是 2^{16} ，用一个16位的无符号整型刚好可以表示，每个地址存放16bit数据。所以 一共的地址空间能够表示： $65536 * 2 = 128KB$ ，这比我们现在使用的电脑小太多了，但这也意味着它足够简单。

“ Memory Storage

```
1  #define MEMORY_MAX (1 << 16)

uint16_t memory[MEMORY_MAX]; /* 65536 locations */
```

上面就是 用c语言表示的 LC-3 的memory， $1 << 16$ 是65536，uint16_t 是16位整型，即：长度位65536的16位整型数组来表示memory。

Registers

现代处理器都会有寄存器，寄存器存在的原因是，CPU直接访问内存或者外设，其访问速度相比CPU的处理速度是非常慢的，于是在CPU内部有一些存储单元，CPU访问这些存储单元会比较快。这些存储单元就是 Register。

LC-3也有寄存器，寄存器通常分为 通用寄存器、标志寄存器、特殊寄存器（PC、SP等）。将控制器从内存读取的数据放到寄存器中，或者将寄存器中的数据放到内存中，这一般用到的是通用寄存器。标志寄存器和特殊寄存器一方面是为了表示指令执行的状态或CPU的状态，另一方面是为了编程使用。

LC-3共有10个寄存器，每一个都可以存放16bit数据，这正好是每个内存中存放的数据的大小。

- 8个通用寄存器，以R0~R7编号。
- 程序计数器PC，这是处理器的标配。
- 一个条件标志寄存器：COND

PC的作用很单纯，就是指向下一条将要执行的指令的地址，所以PC肯定是大于等于16bit的，答案是 16bit，所有寄存器都是16bit的。COND会提供每次执行指令之后的相关信息，比如做了一个减法，通过COND可以知道减法的结果是正数还是0，或者负数。

“ Registers

```
1  enum {  
2      R_R0 = 0,  
3      R_R1,  
      R_R2,  
      R_R3,  
      R_R4,  
      R_R5,  
      R_R6,  
      R_R7,  
      R_PC,  
      R_COUNT  
}
```

```
uint16_t reg[R_COUNT];
```

同样用一个寄存器数组来表示 LC-3的寄存器，用枚举的最后一个值来表示寄存器的数量，这是一个非常好的方法，一方面避免了单独定义一个宏来说明数量，另一方面别人在阅读代码的时候，会很容易将寄存器的相关定义都找到。

指令集

程序员和处理器沟通的方式是指令。每款cpu都有其能够识别的指令，说cpu听得懂的话(指令)，cpu才能干活儿。

LC-3有一套自己的指令集， 2048.obj 这个游戏里面 就包含了 LC-3能够识别的指令。

指令由 **opcode**(操作码：指令要做啥) + **operands**(操作数：谁去做，一般是寄存器)，ITCS 的 4.3.1章节有详细描述。

简而言之，LC-3的指令是由 **16bit** 组成的，前面4bit表示的是 opcode，后面是operands。

LC-3共有16个opcode，分为三类：

- 数据搬移：寄存器之间、寄存器与内存之间。
- 控制：为了改变指令序列，一般指令是一条接着一条指令，控制类指令可以完成跳转，不按顺序执行，这是支持函数和条件判断的基础。
- 操作：这类指令最好理解，就是加法和两个逻辑运算：AND、NOT

详细的指令解释和含义在 ITCS的第五章。

“ Opcodes

```
1  enum
2  {
3      OP_BR = 0, /* branch */
4      OP_ADD,    /* add */
5      OP_LD,     /* load */
6      OP_ST,     /* store */
7      OP_JSR,    /* jump register */
8      OP_AND,    /* bitwise and */
9      OP_LDR,    /* load register */
10     OP_STR,    /* store register */
11     OP_RTI,    /* unused */
12     OP_NOT,    /* bitwise not */
```

```

13     OP_LDI,    /* load indirect */
14     OP_STI,    /* store indirect */
        OP_JMP,    /* jump */
        OP_RES,    /* reserved (unused) */
        OP_LEA,    /* load effective address */
        OP_TRAP   /* execute trap */
};

```

C

每个操作码都有对应的名字，用枚举定义出来，这样既方便知道是哪个操作码(根据枚举的名字)，又方便在写代码是判断(枚举值实际都是整型)，这是一个很好的方法 😊。

条件标志

在写程序时，往往会有很多判断，这时就会用上 **R_COND** 寄存器，该寄存器存储了上一条指令执行情况的相关信息。条件判断就会用到该寄存器。

LC-3只有三种条件标志：

```

1  enum {
2      FL_POS = 1 << 0, /* p */
3      FL_ZRO = 1 << 1, /* Z */
4      FL_NEG = 1 << 2, /* N */
5  };

```

C

如上所示，这个寄存器的第一个bit位表示正数，第二个表示0，第三个表示负数。如果上一次计算的结果是0，那第二个bit位为1。

基于以上，硬件组件的基本定义已经完成了。

3. LC-3汇编

汇编语言是比c语言还底层的语言，它和指令一一对应。lc-3的汇编有对应的汇编器翻译成lc-3的指令。下面先看一个 Hello World 的例子：

“ Hello World Assembly

```

1  .ORIG x3000                                ; this is the address in
2  memory where the program will be loaded

```

Asm

```

3  LEA R0, HELLO_STR                ; load the address of the
4  HELLO_STR string into R0
5  PUTs                             ; output the string pointed
    to by R0 to the console
    HALT                           ; halt the program
    HELLO_STR .STRINGZ "Hello World!" ; store this string here in
    the program
    .END                           ; mark the end of the file

```

别说话，先把上面的代码跑起来吧。先从 [lc3-tools](#) 这里下载编译工具，将上面代码保存为 `hello_world.asm`，接下来我们将这个 `lc-3` 的汇编文件编译为 `lc-3` 的可执行二进制文件。

编译 `lc3-tools`

上面链接给到的工具里面包含了 编译和模拟器的源码，我们先工具都编译出来。

注：`late`是在 `WSL2` 里面完成。

```

Shell  v
1  cd      lc3tools
2  make
sh

```

可能需要根据提示完成一些 `chmod`，最终看到

```

STARTING PASS 1
0 errors found in first pass.
STARTING PASS 2
0 errors found in second pass.

```

就编译好了。目录下面的 `lc3sim` 就是 `lc3` 的模拟器。

编译 `hello_world.asm`

```

~/code/lc3tools
> ./lc3as ../later-lc3-vm/hello_world.asm
STARTING PASS 1
0 errors found in first pass.
STARTING PASS 2
0 errors found in second pass.

```

我们在 `hello_world.asm` 的目录下得到了 `hello_world.obj`，即 `lc-3` 的可执行文件。

运行它：

```
Have fun.

--- halting the LC-3 ---

PC=x0494 IR=xB1AE PSR=x0400 (ZERO)
R0=x0000 R1=x7FFF R2=x0000 R3=x0000 R4=x0000 R5=x0000 R6=x0000 R7=x0490
      x0494 x0FF9 BRNZP TRAP_HALT
Loaded "../later-lc3-vm/hello_world.obj" and set PC to x3000
(lc3sim) c
Hello World!
```

太棒了，**Hello World!** 成功输出了。

.ORIG和.STRINGZ是伪指令，它们指导汇编器生成一些数据或代码，并不是CPU所识别的指令。
.STRINGZ 插入一段字符串到程序的二进制中。

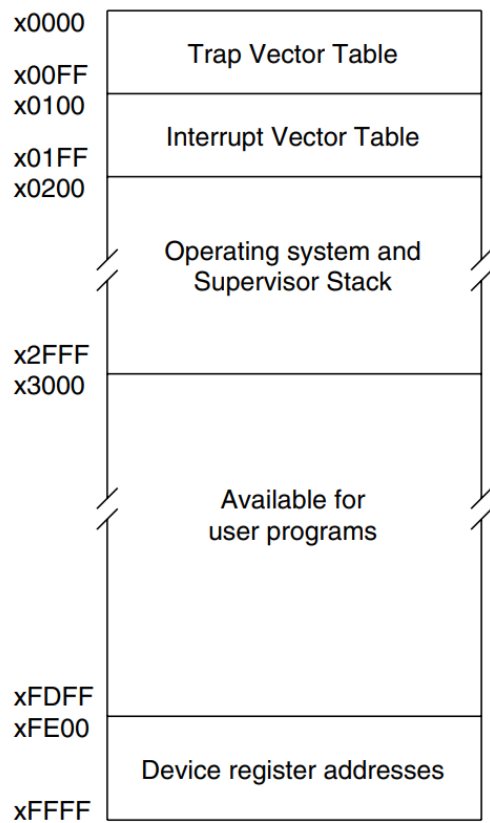
4. 执行程序

写虚拟机并不需要写汇编代码，所以快速结束了上面的例子。我们真正需要关心的是 LC-3 到底是如何执行程序的。宇宙始于一个奇点爆炸，从大爆炸结束那一刻，所有宇宙中的事务，有一个初始的状态，然后就是随着时间不停变化，变化的原因多种多样，就像现在的人类社会，一个人出生时，他的家庭就是他的初始状态，后面家庭发生变化，自己随着年龄增长，发生变化，不停改变状态。抽象的看，整个过程就像是一个程序。奇点爆炸就是开机的一瞬间，然后处理器从第一条代码开始执行，一直不断的执行代码，执行每条时，都处于一个状态，执行完又是一个状态，是一个状态机。 [pat67509_appc.pdf \(georgetown.edu\)](#) 中C.2章节有详细描述。

LC-3开机的时候，初始状态是怎么样的呢？（有兴趣可以了解下 x86的开机执行顺序，0x7c00）。

这部分参考 [ITCS第4.4章节和附录A app-a \(colostate.edu\)](#)：

- PC指针初始化为0x3000，也就是取内存0x3000处的指令。



也就是最开始执行的是OS的代码，其余寄存器状态为0。

执行程序流程

1. 加载PC寄存器指向的内存的指令。
2. 将PC寄存器中的值加一。
3. 译码。
4. 执行。
5. 回到第一步。

根据上面的叙述，我们在main函数中完成这些内容。

“ Main Loop

```
1  int main(int argc, const char* argv[])
2  {
    @{\Load Arguments}
    @{\Setup}

    /* since exactly one condition flag should be set at any
```

```

given time, set the Z flag */
reg[R_COND] = FL_ZR0;

/* set the PC to starting position */
/* 0x3000 is the default */
enum { PC_START = 0x3000 };
reg[R_PC] = PC_START;

int running = 1;
while (running)
{
    /* FETCH */
    uint16_t instr = mem_read(reg[R_PC]++);
    uint16_t op = instr >> 12;

    switch (op)
    {
        case OP_ADD:
            @{ADD}
            break;
        case OP_AND:
            @{AND}
            break;
        case OP_NOT:
            @{NOT}
            break;
        case OP_BR:
            @{BR}
            break;
        case OP_JMP:
            @{JMP}
            break;
        case OP_JSR:
            @{JSR}
            break;
        case OP_LD:
            @{LD}
            break;
        case OP_LDI:
            @{LDI}

```

```

        break;
    case OP_LDR:
        @{LDR}
        break;
    case OP_LEA:
        @{LEA}
        break;
    case OP_ST:
        @{ST}
        break;
    case OP_STI:
        @{STI}
        break;
    case OP_STR:
        @{STR}
        break;
    case OP_TRAP:
        @{TRAP}
        break;
    case OP_RES:
    case OP_RTI:
    default:
        @{BAD_OPCODE}
        break;
    }
}
@{Shutdown}
}

```

C

在进入主循环前，我们需要先处理参数，作为一个计算机模拟器，我们的任务是把计算机程序跑起来，所以我们至少有一个输入参数是 可执行文件的路径。

“ Load Arguments

```

1  if (argc < 2)
2  {
3      /* show usage string */
4      printf("lc3 [image-file1] ...\n");

```

C ▾

```
5     exit(2);
6 }
7
8 if (!read_image(argv[1]))
9 {
10     printf("failed to load image: %s\n", argv[1]);
11     exit(1);
12 }
```

读取image也就是读取 2048.obj 等二进制可执行文件，这里是要将二进制读取到 lc-3 的 memory 中，读取到偏移0x3000开始的地方。

5. 指令翻译

把image读取到内存后，开始执行这个程序了，按照那5个步骤，开始循环起来。所以我们要完成每条指令的工作，即指令翻译，这是模拟器核心的工作。但这其实并不复杂。

ADD

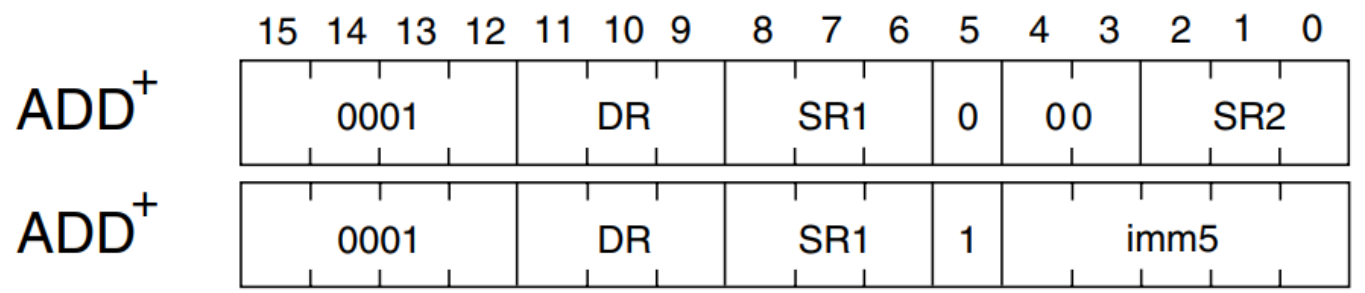
先看看指令怎么使用的，这部分可以参考ITCS的第五章。

“ Add Register Assembly

Asm ▾

```
1  ADD R2 R0 R1 ; add the contents of R0 to R1 and store in R2.
2  ADD R0 R0 1 ; add 1 to R0 and store back in R0
```

ADD可以将两个寄存器相加，也可以是寄存器和立即数（立即数就是一个数字，仅此而已）。来看看 ADD的指令编码吧（参考[app-a \(jmeiners.com\)](#)）：



通过第5个bit位来区分ADD的两种情况，参考资料甚至将代码都写出来了。所以我们获取相应的位就能得到对应的数据。

符号扩展

从上图可以看到，立即数模式下的立即数只有0~4个bit，所以立即数只有5个bit位。但处理器需要的是16bit数据（这也是c语言中的隐式类型转换，处理器只能进行整型算术运算）。所以需要
将5bit展开到16bit。对于正数而言，前面填0，对于负数，前面填1。（这里涉及到补码知识）

“ Sign Extend

```
1 uint16_t sign_extend(uint16_t x, int bit_count)
2 {
    if ((x >> (bit_count - 1)) & 1) {
        x |= (0xFFFF << bit_count);
    }
    return x;
}
```

先右移4bit，取出最高位，如果不是1那就是正数，如果是1就是负数，与0xFFFF或运算，补齐16bit。

条件标记寄存器

每次执行完指令后，我们需要更新条件标记寄存器，就判断保存计算结果的寄存器就好了。

“ Update Flags

```
1 void update_flags(uint16_t r)
2 {
3     if (reg[r] == 0)
4     {
5         reg[R_COND] = FL_ZR0;
6     }
7     else if (reg[r] >> 15) /* a 1 in the left-most bit
    indicates negative */
    {
        reg[R_COND] = FL_NEG;
    }
    else
```

```

    {
        reg[R_COND] = FL_POS;
    }
}

```

C

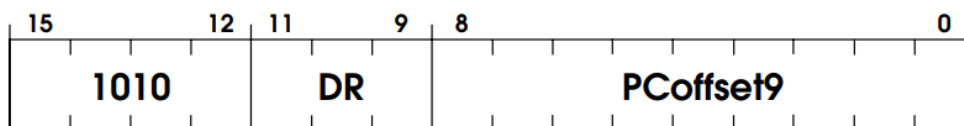
现在 ADD 的代码对你来说，已经没有什么秘密了。

LDI

从二进制的指令编码到汇编的字母缩写，是一一对应的，对于汇编的缩写，我们将其展开，就能更好的了解其意义。

“ LDI: load indirect 将内存中的值加载到寄存器。

Encoding



Operation

$$DR = \text{mem}[\text{mem}[PC^{\dagger} + \text{SEXT}(\text{PCoffset9})]];$$

取当前PC的值，加上扩展后的偏移，，然后获取其内存的值，这个值仍然是一个地址，或者说是一个指针，最后获取指针的值。

例子：

“ LDI Sample

```

1 // the value of far_data is an address
2 // of course far_data itself (the location in memory containing
3 the address) has an address
4 char* far_data = "apple";
5
6 // In memory it may be layed out like this:
7

```

C

```

8 // Address Label      Value
9 // 0x123: far_data = 0x456
10 // ...
11 // 0x456: string    = 'a'
12
13 // if PC was at 0x100
14 // LDI R0 0x023
15 // would load 'a' into R0

```

c

6. 指令备忘单

我们需要完成剩下的指令翻译，对照下面内容看代码实现。

“ RTI & RES

直接 `abort()`，因为这两个指令在LC-3根本没有。

“ AND

Encodings

15	12	11	9	8	6	5	4	3	2	0
0101	DR	SR1	0	00	SR2					

15	12	11	9	8	6	5	4			0
0101	DR	SR1	1	imm5						

这和ADD几乎一致，仅仅只是把加法操作换成了 &

Note

Examples

AND R2, R3, R4 ;R2 ← R3 AND R4

AND R2, R3, #7 ;R2 ← R3 AND 7

“ NOT

Encoding

15	12	11	9	8	6	5	4	3	2	0
1001	DR	SR	1	1111						

Note

Examples

NOT R4, R2 ; R4 ← NOT(R2)

将SR寄存器的值取反，放到DR中，更新条件标志寄存器。

“ BR

控制指令改变指令执行的序列。条件分支跳转要么改变PC，要么啥都不做，这取决于前一条指令执行的结果。

由于标志寄存器只记录三种情况，所以一共就只有四种条件分支条件。

- 判断N，如果是负数则跳转 → BRn
- 判断Z，如果是0则跳转 → BRz
- 判断P，如果是正数则跳转 → BRp
- 判断NZP，这是无条件跳转，因为一共只有三种情况。

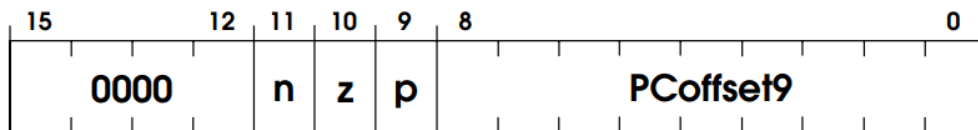
上面描述等价于：

```
1  if ((n AND N) OR (z AND Z) OR (p AND P))
2      PC = PC + SEXT(PCoffset9)
```

C v

C

Encoding

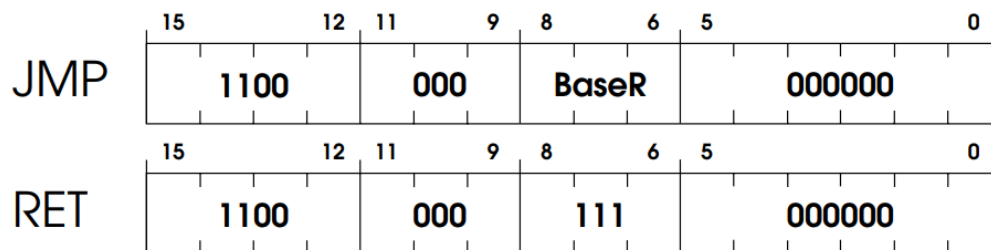


取 9~11 位，与条件标志寄存器 &，为真就跳转（PC + PCoffset）

“ RET 和 JMP

这两个跳转指令很简单，ret直接跳转到R7寄存器的地址，JMP跳转到指定的寄存器的地址。

Encoding



Note

Examples

JMP R2 ; PC ← R2

RET ; PC ← R7

二进制的111就是十进制的7，所以这两条指令可以用同样的代码实现。

“ JSR

Jump Register

类似 CALL 指令，将当前PC保存到 R7中，然后根据 第11位判断 如何处理PC。

Note

Operation

$R7 = PC; t$

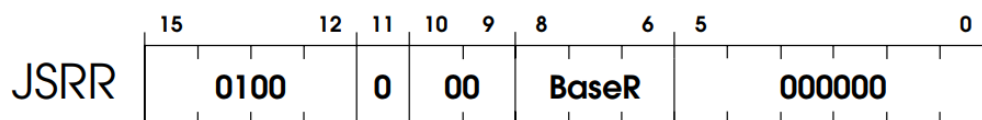
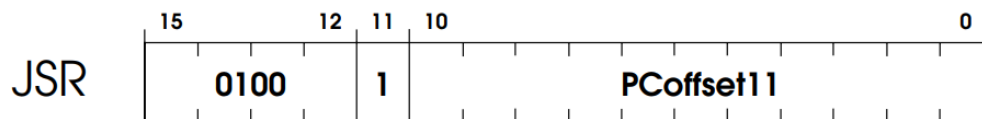
if (bit[11] == 0)

----PC = BaseR;

else

----PC = $PCt + \text{SEXT}(\text{PCoffset11})$;

Encoding



“ LD

Load

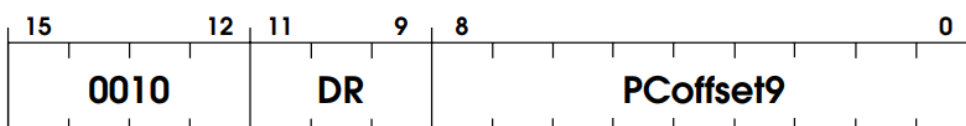
将内存中的数据加载到寄存器。

Note

Example


LD R4, VALUE ; $R4 \leftarrow \text{mem}[\text{VALUE}]$

Encoding



“ LDR

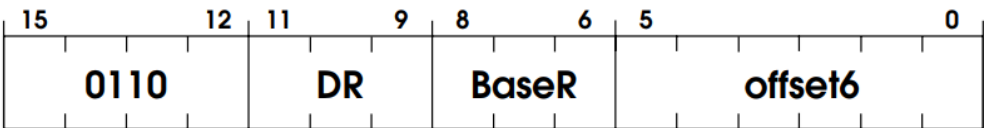
Load Register
看例子就清楚了。

 **Note**

Example


LDR R4, R2, #-5 ; R4 ← mem[R2 - 5]

Encoding



 LEA

Load effective addres
不多说了，看Operation

 **Note**

DR = PC† + SEXT(PCoffset9);

Encoding



 ST

store
将寄存器中的值存到内存中。

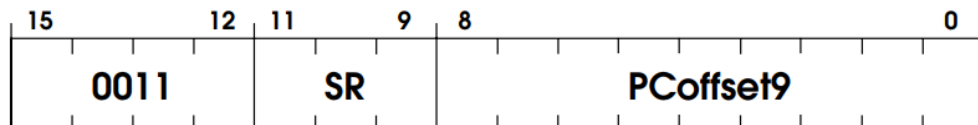
Note

Operation

$\text{mem}[\text{PC} + \text{SEXT}(\text{PCoffset9})] = \text{SR};$

代码的就是对Operation的实现。

Encoding



“ STI

store indirect

所谓间接存储，就是内存里面的值还是一个地址，或者说指针，需要再取其值。

Note

```
1 mem[mem[PC + SEXT(PCoffset9)]] = SR;
```

“ STR

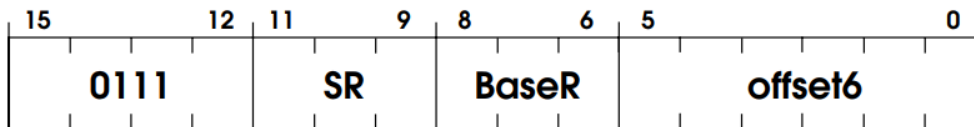
store register

基于寄存器中保存的地址加上偏移获取到一个地址，然后将寄存器中的值保存进去。

Note

```
1 mem[BaseR + SEXT(offset6)] = SR;
```

Encoding

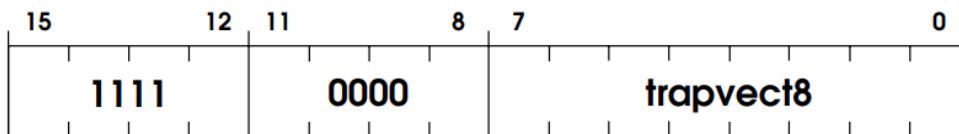


7. trap routines

LC-3提供了一些程序，它们可以完成一些通用的任务和与IO设备交互。比如说，有从键盘获取输入、往屏幕上输出字符串。它们都称为 **trap routines**. 每个 **trap routine** 都有一个对应的 **trap code**。其实简单来说，就是预先实现了一些函数，通过类似 函数名 的方式来调用。还可以用 **TRAP #imm** 这样调用。

实际类似 系统调用，BIOS也给OS提供了类似的一些功能。调用 **TRAP**，就是调用函数，首先保存 PC 到R7，然后改变PC指针到 对应 **trap** 函数。

Encoding



有6个trap routines

Table A.2 Trap Service Routines		
Trap Vector	Assembler Name	Description
x20	GETC	Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x21	OUT	Write a character in R0[7:0] to the console display.
x22	PUTS	Write a string of ASCII characters to the console display. The characters are contained in consecutive memory locations, one character per memory location, starting with the address specified in R0. Writing terminates with the occurrence of x0000 in a memory location.
x23	IN	Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console monitor, and its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x24	PUTSP	Write a string of ASCII characters to the console. The characters are contained in consecutive memory locations, two characters per memory location, starting with the address specified in R0. The ASCII code contained in bits [7:0] of a memory location is written to the console first. Then the ASCII code contained in bits [15:8] of that memory location is written to the console. (A character string consisting of an odd number of characters to be written will have x00 in bits [15:8] of the memory location containing the last character to be written.) Writing terminates with the occurrence of x0000 in a memory location.
x25	HALT	Halt execution and print a message on the console.

获取到TRAP指令后，继续判断需要哪个trap routine

Note

TRAP Codes

```
1 enum
2 {
    TRAP_GETC = 0x20, /* get character from keyboard, not
    echoed onto the terminal */
    TRAP_OUT = 0x21, /* output a character */
    TRAP_PUTS = 0x22, /* output a word string */
    TRAP_IN = 0x23, /* get character from keyboard,
    echoed onto the terminal */
    TRAP_PUTSP = 0x24, /* output a byte string */
    TRAP_HALT = 0x25 /* halt the program */
};
```

这也就是为啥PC的初始值是 0x3000 而不是0x0的原因，前面存放 trap routines

在裸机上实现这些函数，肯定得用汇编，而且还得和外设的寄存器打交道，会稍微复杂。但我们现在是在实现模拟器，完全可以利用现有的编程接口，比如，往屏幕输出用puts就好了（用printf也可以的，我觉得）。

trap routine 的实现非常简单，对照上面trap table就行。

8. LC-3 endianness

LC-3是大端的，现在的计算机大多数都是小端的，我们需要大小转换。

9. 内存映射寄存器

如同arm一样，将寄存器映射到寻址空间中，就如同访问内存一样，读写这写地址就如同读写外设的寄存器。

键盘有两个寄存器：

- keyboard status register → KBSR

- keyboard data register —→ KBDR

KBSR 表示是否有键被按下，KBDR表示哪个键被按下了。

尽管可以使用 GETC 请求键盘输入，但这会阻止执行，直到接收到输入为止。KBSR 和 KBDR 允许您轮询设备的状态)并继续执行，因此程序可以在等待输入时保持响应。

通常按键都是触发中断，而这里使用轮询的方式。当2048游戏开始，第一副游戏棋盘摆好后，就会等待用户的按键，此时会一直读取键盘的KBSR，判断是否有按键被按下。

键盘寄存器的地址：

Table A.3 Device Register Assignments		
Address	I/O Register Name	I/O Register Function
xFE00	Keyboard status register	Also known as KBSR. The ready bit (bit [15]) indicates if the keyboard has received a new character.
xFE02	Keyboard data register	Also known as KBDR. Bits [7:0] contain the last character typed on the keyboard.
xFE04	Display status register	Also known as DSR. The ready bit (bit [15]) indicates if the display device is ready to receive another character to print on the screen.
xFE06	Display data register	Also known as DDR. A character written in the low byte of this register will be displayed on the screen.
xFFFE	Machine control register	Also known as MCR. Bit [15] is the clock enable bit. When cleared, instruction processing stops.

10. 优化


有些函数的功能later也没有搞得很清楚，但那些只是为了更好的和键盘互动，和虚拟机无关. So feel free to copy paste!

Note

function

```
1 void disable_input_buffering()
2 void restore_input_buffering()
3 uint16_t check_key()
```

11. 总结

本文介绍了lc3-vm的很多细节，详细读者能够轻松的读懂整个几百行的代码，[仓库在这里](#)  但lc3背后的故事更加吸引人，这是从十多年前就开始的一个项目，[再次强调](#)，lc3的作者以自底向上的思想，完成了一个16bit 计算机的设计和实现，从门电路到ISA，到指令集设计，并完成了lc3的虚拟机实现，汇编到指令的翻译，甚至c语言到lc3指令的翻译，还有一个later还未探索的lc3os!

向这些先驱致敬，提供了这么好的资源，并且later十分认同 自底向上 的学习方法。

Enjoy 😊

参考

参考原文：[Write your Own Virtual Machine \(jmeiners.com\)](#) 

代码仓库：[justinmeiners/lc3-vm: Write your own virtual machine for the LC-3 computer! \(github.com\)](#) 

[|ITCS: Introduction To Computing Systems: From Bits & Gates To C/C++ & Beyond, Third Edition \(icourse.club\)](#) 

[Lecture_10-310h.ppt \(utexas.edu\)](#) 

[CS 345 \(byu.edu\)](#) 

[ECE120: Introduction to Computing, Spring 2019 ZJUI Section \(illinois.edu\)](#) 