400 行 C 代码实现一个虚拟机

Linux内核那些事 2021-12-07 17:00

1. 引言

本文将教你编写一个自己的虚拟机（VM），这个虚拟机能够运行汇编语言编写的程序， 例如我朋友编写的 2048 或者我自己的 Roguelike。如果你会编程，但希望 更深入地了解计算机的内部原理以及编程语言是如何工作的，那本文很适合你。从零开始 写一个虚拟机听起来可能让人有点望而生畏，但读完本文之后你会惊讶于这件事原来如此简 单，并从中深受启发。

本文所说的虚拟机最终由 400 行左右 C 代码组成。理解这些代码只需要基本的 C/C++ 知识和二进制运算。这个虚拟机可以在 Unix 系统（包括 macOS）上执行。代码中包含少 量平台相关的配置终端（terminal）和显示（display）的代码，但这些并不是本项目的核 心。（欢迎大家添加对 Windows 的支持。）

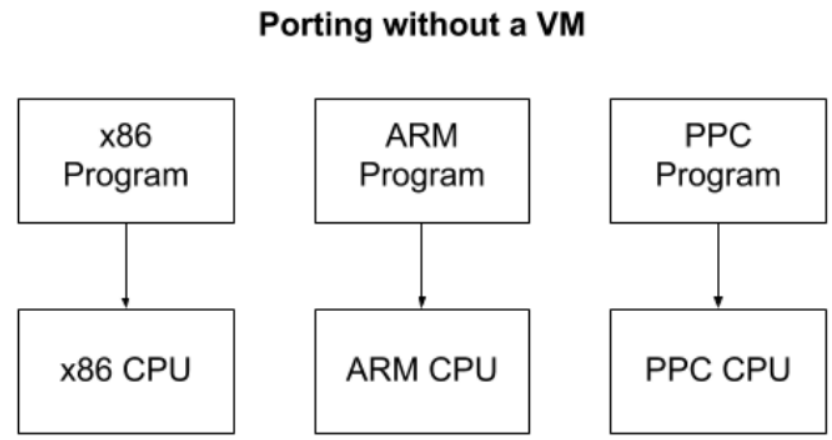
> 注意：这个虚拟机是Literate Programming 的产物。本文会解释每段代码的原理，最终的实现就是将这些代码片段连起来。

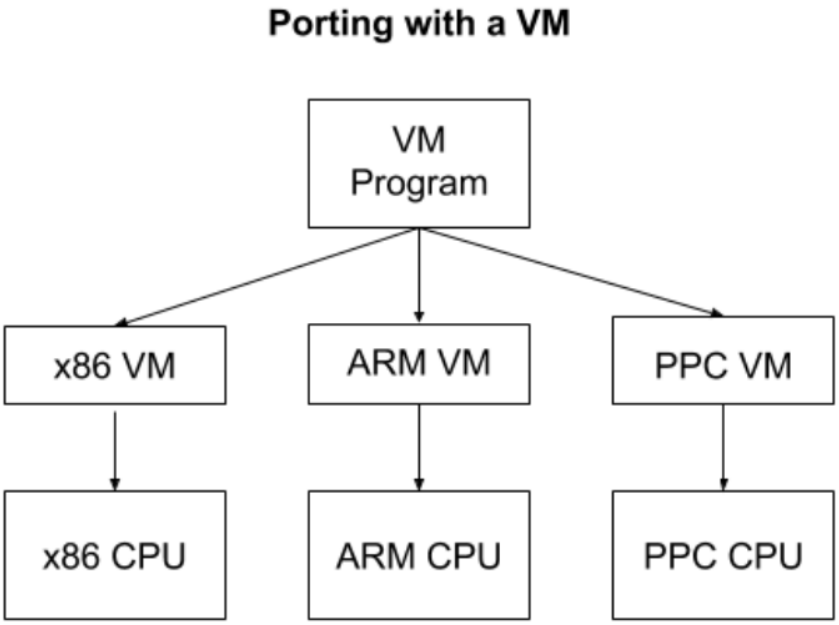
什么是虚拟机?

虚拟机就像计算机（computer），它模拟包括 CPU 在内的几个硬件组件，能够执行 算术运算、读写内存、与 I/O 设备交互。最重要的是，它能理解机器语言（machine language），因此可以用相应的语言来对它进行编程。

一个虚拟机需要模拟哪些硬件要看它的使用场景。有些虚拟机是设计用来模拟特定类型的计算设备 的，例如视频游戏模拟器。现在 NES 已经不常见了，但我们还是可以用 NES 硬件模拟器来玩 NES 游戏。这些模拟器必须能忠实地 重建每一个细节，以及原硬件的每个主要组件。

另外一些虚拟机则完全是虚构的，而非用来模拟硬件。这类虚拟机的主要用途是使软件开发更容易。例如，要开发一个能运行在不同计算架构上的程序，你无需使用每种架构特定的汇编方言来实现一遍自己的程序，而只需要使用一个跨平台的虚拟机提供的汇编语言。





> 注：编译器也解决了类似的跨平台问题，它将标准的高级语言编写的程序编译成能在不同 CPU 架构上执行的程序。相比之下，虚拟机的跨平台方式是自己创建一个标准的 CPU 架 构，然后在不同的物理设备上模拟这个 CPU 架构。编译器方式的优点是没有运行时开销 （runtime overhead），但实现一个支持多平台的编译器是非常困难的，但实现一个虚拟 机就简单多了。在实际中，人们会根据需求的不同混合使用虚拟机和编译器，因为二者工 作在不同的层次。

Java Virtual Machine (JVM) 就是一个非常成功的例子。JVM 本身是一个中等大小、程序员完全能够看懂的程序，因此很 容易将它移植到包括手机在内的上千种设备上。只要在设备上实现了 JVM，接下来任何 Java、Kotlin 或 Clojure 程序都无需任何修改就可以直接运行在这个设备上。唯一的开销 来自虚拟机自身以及机器之上的 进一步抽象。大部分情况下，这完全是可以接受的。

虚拟机不必很大或者能适应各种场景，老式的视频游戏 经常使用很小的虚拟机来提 供简单的脚本系统（scripting systems）。

虚拟机还适用于在一个安全的或隔离的环境中执行代码。一个例子就是垃圾回收（GC）。要在 C 或 C++ 之上实现一个自动垃圾回收机制并不容易 ，因为程序无法看到它自身的栈或变

量。但是，虚拟机是在它运行的程序“之外”的，因此它能够看到栈上所有的内存引用 。

另一个例子是以太坊智能合约 （Ethereum smart contracts）。智能合约是在区块链网络中被验证节点（validating node）执行的小段程序。这就要求 人们在无法提前审查这些由陌生人编写的代码的情况下，直接他们的机器上执行这些代码。为避免合约执行一些恶意行为，智能合约将它们放到一个 虚拟机 内执行，这个虚拟机没有权限访问文件系统、网络、磁盘等等资源。以太坊也很好地展现了 虚拟机的可移植性特性，因为以太坊节点可以运行在多种计算机和操作系统上。使用虚拟机 使得智能合约的编写无需考虑将在什么平台运行。

2. LC-3 架构

我们的虚拟机将会模拟一个虚构的称为 LC-3 的计算机。LC-3 在学校中比较流行，用于教学生如何用汇编编程。与 x86 相比 ， LC-3 的指令集更 加简化，但现代 CPU 的主要思想其中都包括了。

我们首先需要模拟机器最基础的硬件组件，尝试来理解每个组件是做什么的，如果 现在无法将这些组件拼成一张完整的图也不要着急。

2.1 内存

LC-3 有 65,536 个内存位置（16 bit 无符号整形能寻址的最大值），每个位置可以存储一 个 16-bit 的值。这意味着它总共可以存储 128KB 数据（64K * 2 Byte），比我们平时接触 的计算机内存小多了！在我们的程序中，这个内存会以简单数组的形式存放数据：

```
/* 65536 locations */
uint16_t memory[UINT16_MAX];
```

2.2 寄存器

一个寄存器就是 CPU 上一个能够存储单个数据的槽（slot）。寄存器就像是 CPU 的 “工作台”（workbench），CPU 要对一段数据进行处理，必须先将数据放到某个寄存器中。但 因为寄存器的数量很少，因此在任意时刻只能有很少的数据加载到寄存器。计算机的解决办 法是：首先将数据从内存加载到寄存器，然后将计算结果放到其他寄存器，最后将最终结果 再写回内存。

LC-3 总共有 10 个寄存器，每个都是 16 比特。其中大部分都是通用目的寄存器，少数几 个用于特定目的。

- 8 个通用目的寄存器（R0-R7）
- 1 个程序计数器（program counter, PC）寄存器
- 1 个条件标志位（condition flags，COND）寄存器

通用目的寄存器可以用于执行任何程序计算。程序计数器（PC）是一个无符号整数，表示内存中将要执行的下一条指令的地址。条件标记寄存器记录前一次计算结果的正负符号。

```
enum {
    R_R0 = 0,
    R_R1,
    R_R2,
    R_R3,
    R_R4,
    R_R5,
    R_R6,
    R_R7,
    R_PC, /* program counter */
```

```
    R_COND,
    R_COUNT
};
```

和内存一样，我们也用数组来表示这些寄存器：

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

2.3 指令集

一条指令就是一条 CPU 命令，它告诉 CPU 执行什么任务，例如将两个数相加。一条指令包含两部分：

- 操作码（opcode）：表示任务的类型
- 执行任务所需的参数

每个操作码代表 CPU “知道”的一种任务。在 LC-3 中只有 16 个操作码。计算机能够完成 的所有计算，都是这些简单指令组成的指令流。每条指令 16 比特长，其中最左边的 4 个 比特存储的是操作码，其余的比特存储的是参数。

我们稍后会详细介绍每条指令是做什么的，现在先定义下面的这些操作码，确保它们 是按如下顺序定义的，这样每条指令就可以获得正确的枚举值：

```
enum {
    OP_BR = 0, /* branch */
    OP_ADD,    /* add  */
    OP_LD,     /* load */
    OP_ST,     /* store */
    OP_JSR,    /* jump register */
    OP_AND,    /* bitwise and */
    OP_LDR,    /* load register */
    OP_STR,    /* store register */
    OP_RTI,    /* unused */
    OP_NOT,    /* bitwise not */
    OP_LDI,    /* load indirect */
    OP_STI,    /* store indirect */
    OP_JMP,    /* jump */
    OP_RES,    /* reserved (unused) */
    OP_LEA,    /* load effective address */
    OP_TRAP    /* execute trap */
};
```

> 注：Intel x86 架构有几百条指令，而其他的架构例如 ARM 和 LC-3 只有很少的指令 。较小的指令集称为精简指令集（RISC），较大 的指令集称为复杂指令集（CISC）。更大 的指令集本质上通常并没有提供新特性，只是使得编写 汇编更加方便 。一条 CISC 指令能做的事情可能需要好几条 RISC 才能完成。但是，对设计和制造工程 师来说，CISC 更加复杂和昂贵，设计和制造业更贵。包括这一点在内的一些权衡使得指 令设计也在不断变化。

2.4 条件标志位

R_COND 寄存器存储条件标记，其中记录了最近一次计算的执行结果。这使得程序可以完成诸如 `if (x > 0) { ... }` 之类的逻辑条件。

每个 CPU 都有很多条件标志位来表示不同的情形。LC-3 只使用 3 个条件标记位，用来 表示前一次计算结果的符号：

```
enum {
    FL_POS = 1 << 0, /* P */
    FL_ZRO = 1 << 1, /* Z */
    FL_NEG = 1 << 2, /* N */
};
```

注：<< 和 >> 表示移位操作。

至此，我们就完成了虚拟机的硬件组件的模拟。

3. 汇编示例

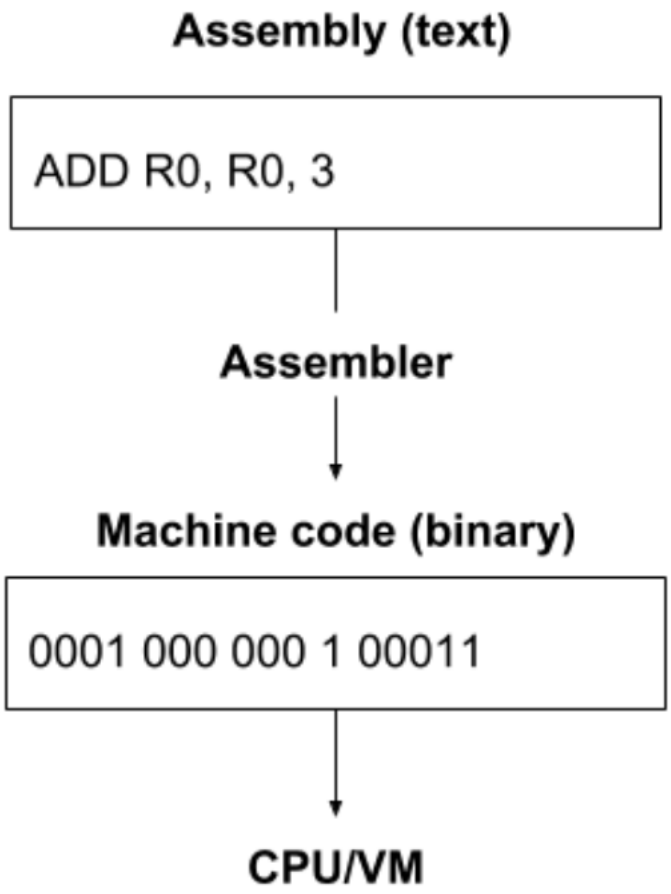
下面通过一个 LC-3 汇编程序先来感受一下这个虚拟机运行的是什么代码。这里无需知 道如何编写汇编程序或者理解背后的工作原理，只是先直观感受一下。下面是 “Hello World” 例子：

```
.ORIG x3000                         ; this is the address in memory where th
LEA R0, HELLO_STR                   ; load the address of the HELLO_STR stri
PUTs                                ; output the string pointed to by R0 to
HALT                                ; halt the program
HELLO_STR .STRINGZ "Hello World!"   ; store this string here in the program
.END                                ; mark the end of the file
```

和 C 类似，这段程序从最上面开始，每次执行一条声明（statement）。但和 C 不同的是，这里没有作用域符号 {} 或者控制结构（例如 if 和 while），仅仅是一个扁平的声 明列表（a flat list of statements）。这样的程序更容易执行。

注意，其中一些声明中的名字和我们前面的定义的操作码（opcodes）是一样的。前面 介绍到，每条指令都是 16 比特，但这里的汇编程序看起来每行的字符数都是不一样的。为什么会有这种不一致呢?

这是因为这些汇编声明都是以人类可读写的格式编写的，以纯文本的形式表示。一种称为 汇编器（assembler）的工具会将这些文本格式的指令转换成 16 比特的二进制指令， 后者是虚拟机可以理解的。这种二进制格式称为机器码（machine code），是虚拟机可以 执行的格式，其本质上就是一个 16 比特指令组成的数组。



> 注：虽然在开发中编译器（compiler）和汇编器（assembler）的角色是类似的，但二者是两个不同的工具。汇编器只是简单地将程序员编写的文本编码（encode）成二进制格式，将其中的符号替换成相应的二进制表示并打包到指令内。

`.ORIG` 和 `.STRINGZ` 看起来像是指令，但其实不是，它们称为汇编制导命令 （assembler directives），可以生成一段代码或数据。例如， `.STRINGZ` 会在它所在的 位置插入一段字符串。

循环和条件判断是通过类似 goto 的指令实现的。下面是一个如何计时到 10 的例子：

```
AND R0, R0, 0                    ; clear R0
LOOP                             ; label at the top of our loop
ADD R0, R0, 1                    ; add 1 to R0 and store back in R0
ADD R1, R0, -10                  ; subtract 10 from R0 and store back in
BRn LOOP                         ; go back to LOOP if the result was nega
... ; R0 is now 10!
```

> 注：本文不需要读者会编写汇编代码。但如果你感兴趣，你可以使用 LC-3 工具来编写和汇编你自己写的汇编程序。

4. 执行程序

前面的例子是给大家一个直观印象来理解虚拟机在做什么。实现一个虚拟机不必精通汇编编程，只要遵循正确的流程来读取和执行指令，任何 LC-3 程序都能够正确执行，不管这些程 序有多么复杂。理论上，这样的虚拟机甚至可以运行一个浏览器或者 Linux 这样的操作系 统。

如果深入地思考这个特性，你就会意识到这是一个在哲学上非常奇特的现象：程序能完成各种智能的事情，其中一些我们甚至都很难想象；但同时，所有这些程序最终都是用我们编 写的这些少量指令来执行的！我们既了解 —— 又不了解 —— 那些和程序执行相关的的事情。图灵 曾经讨探讨过这种令人惊叹的思想：

> “The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false.” — Alan M. Turing

过程（Procedure）

我们将编写的这个过程（procedure）描述如下：

- 1. 从 PC 寄存器指向的内存地址中加载一条指令
- 2. 递增 PC 寄存器
- 3. 查看指令中的 opcode 字段，判断指令类型
- 4. 根据指令类型和指令中所带的参数执行该指令
- 5. 跳转到步骤 1

你可能会有疑问：“如果这个循环不断递增 PC，而我们没有 if 或 while，那程序不会 很快运行到内存外吗？”答案是不会，我们前面提到过，有类似 goto 的指令会通过修改 PC 来改变执行流。

下面是以上流程的大致代码实现：

```
int main(int argc, const char* argv[]) {
    {Load Arguments, 12}
    {Setup, 12}

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

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

        switch (op) {
            case OP_ADD: {ADD, 6} break;
            case OP_AND: {AND, 7} break;
            case OP_NOT: {NOT, 7} break;
            case OP_BR: {BR, 7} break;
```

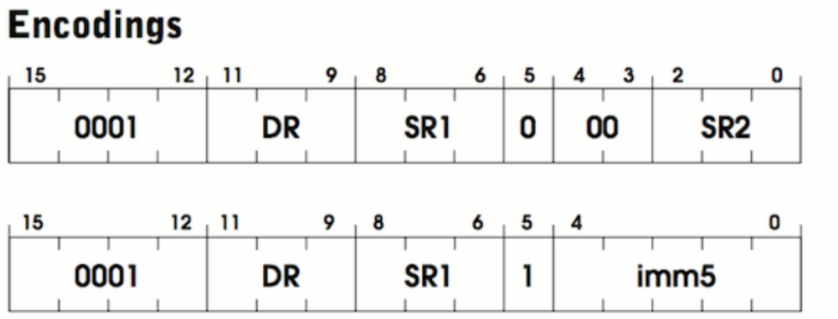
```
            case OP_JMP: {JMP, 7} break;
            case OP_JSR: {JSR, 7} break;
            case OP_LD: {LD, 7} break;
            case OP_LDI: {LDI, 6} break;
            case OP_LDR: {LDR, 7} break;
            case OP_LEA: {LEA, 7} break;
            case OP_ST: {ST, 7} break;
            case OP_STI: {STI, 7} break;
            case OP_STR: {STR, 7} break;
            case OP_TRAP: {TRAP, 8} break;
            case OP_RES:
            case OP_RTI:
            default:
                {BAD OPCODE, 7}
                break;
        }
    }
    {Shutdown, 12}
}
```

5. 指令实现

现在需要做的就是正确地实现每一条指令。每条指令的详细描述见 GitHub Repo 中附录的 PDF 文档。你需要 照着文档的描述自己实现这些指令。这项工作做起来其实比听起来要容易。下面我会拿其中 的两个作为例子来展示如何实现，其余的见下一章。

5.1 ADD

ADD 指令将两个数相加，然后将结果存到一个寄存器中。关于这条指令的描述见 526 页。ADD 指令的编码格式如下：



这里给出了两张图是因为 ADD 指令有两种不同的“模式”。在解释模式之前，先来看看两张 图的共同点：

- 1. 两者都是以 0001 这 4 个比特开始的，这是 OP_ADD 的操作码（opcode）
- 2. 后面 3 个比特名为 DR（destination register），即目的寄存器，相加的结果会放到 这里
- 3. 再后面 3 个比特是 SR1，这个寄存器存放了第一个将要相加的数字

至此，我们知道了相加的结果应该存到哪里，以及相加的第一个数字。只要再知道第二个数在哪里就可以执行加法操作了。从这里开始，这两者模式开始不同：注意第 5 比特 ，这个标

志位表示的是操作模式是立即模式（immediate mode）还是寄存器模式 （register mode）。在寄存器模式中，第二个数是存储在寄存器中的，和第一个数类似。这个寄存器称为 SR2，保存在第 0-2 比特中。第 3 和 第 4 比特没用到。用汇编代码描 述就是：

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

在立即模式中，第二个数直接存储在指令中，而不是寄存器中。这种模式更加方便，因 为程序不需要额外的指令来将数据从内存加载到寄存器，直接从指令中就可以拿到这个值。这种方式的限制是存储的数很小，不超过 2^5 = 32（无符号）。这种方式很适合对一个值 进行递增。用汇编描述就是：

```
ADD R0 R0 1 ; add 1 to R0 and store back in R0
```

下面一段解释来自 LC-3 规范：

> If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In both cases, the second source operand is added to the contents of SR1 and the result stored in DR. (Pg. 526)

这段解释也就是我们前面讨论的内容。但什么是 “sign-extending”（有符号扩展）？ 虽然立即模式中存储的值只有 5 比特，但这个值需要加到一个 16 比特的值上。因此，这些 5 比 特的数需要扩展到 16 比特才能和另一个数相匹配。对于正数，我们可以在前面填充 0， 填充之后值是不变的。但是，对于负数，这样填充会导致问题。例如， -1 的 5 比特表示 是 11111。如果我们用 0 填充，那填充之后的 0000 0000 0001 1111 等于 32！这种 情况下就需要使用有符号扩展（ sign extension），对于正数填充 0，对负数填充 1。

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

> 注：如果你如何用二进制表示负数感兴趣，可以查阅二进制补码（Two’s Complement）相关的内容。本文中只需要知道怎么进行有符号扩展就行了。

规范中还有一句：

> The condition codes are set, based on whether the result is negative, zero, or positive. (Pg. 526)

前面我们定义的那个条件标记枚举类型现在要派上用场了。每次有值写到寄存器时，我们 需要更新这个标记，以标明这个值的符号。为了方便，我们用下面的函数来实现这个功能：

```
void update_flags(uint16_t r) {
    if (reg[r] == 0) {
        reg[R_COND] = FL_ZRO;
    }
    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;
    }
}
```

现在我们就可以实现 ADD 的逻辑了：

```
{
    uint16_t r0 = (instr >> 9) & 0x7; /* destination register (DR) */
    uint16_t r1 = (instr >> 6) & 0x7; /* first operand (SR1) */
    uint16_t imm_flag = (instr >> 5) & 0x1; /* whether we are in immediate m

    if (imm_flag) {
        uint16_t imm5 = sign_extend(instr & 0x1F, 5);
        reg[r0] = reg[r1] + imm5;
    } else {
        uint16_t r2 = instr & 0x7;
        reg[r0] = reg[r1] + reg[r2];
    }

    update_flags(r0);
}
```

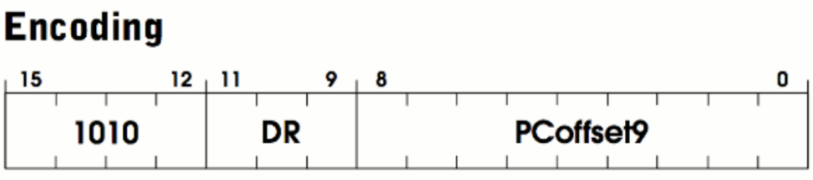
本节包含了大量信息，这里再总结一下：

- ADD 接受两个值作为参数，并将计算结果写到一个寄存器中
- 在寄存器模式中，第二个值存储在某个寄存器中
- 在立即模式中，第二个值存储在指令最右边的 5 个比特中
- 短于 16 比特的值需要执行有符号扩展
- 每次指令修改了寄存器后，都需要更新条件标志位（condition flags）

以上就是 ADD 的实现，你可能会觉得以这样的方式实现另外 15 个指令将会是一件非常繁 琐的事情。好消息是，前面的这些函数基本都是可以重用的，因为另外 15 条指令中，大部 分都会组合有符号扩展、不同的模式和更新条件标记等等。

5.2 LDI

LDI 是 load indirect 的缩写，用于从内存加载一个值到寄存器，规范见 532 页。LDI 的二进制格式如下：



与 ADD 相比，LDI 只有一种模式，参数也更少。LDI 的操作码是 1010，对应 OP_LDI 枚举类型。和 ADD 类似，它包含一个 3 比特的 DR（destination register）寄存器，用 于存放加载的值。剩余的比特组成 PCoffset9 字段，这是该指令内嵌的一个立即值（ immediate value），和 imm5 类似。由于这个指令是从内存加载值，因此我们可以猜测， PCoffset9 是一个加载值的内存地址。LC-3 规范提供了更多细节：

> An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. What is stored in memory at this address is the address of the data to be loaded into DR. (Pg. 532)

和前面一样，我们需要将这个 9 比特的 PCoffset9 以有符号的方式扩展到 16 比特，但 这次是将扩展之后的值加到当前的程序计数器 PC（如果回头去看前面的 while 循 环，就会发现这条指令加载之后 PC 就会递增）。相加得到的结果（也就是 PC 加完之后的 值）表示一个内存地址，这个地址中存储的值表示另一个地址，后者中存储的是需要加载到 DR 中的值。

这种方式听上去非常绕，但它确是不可或缺的。LD 指令只能加载 offset 是 9 位的地址， 但整个内存是 16 位的。LDI 适用于加载那些远离当前 PC 的地址内的值，但要加载这 些值，需要将这些最终地址存储在离 PC 较近的位置。可以将它想想成 C 中有一个局部变 量，这变量是指向某些数据的指针：

```
// the value of far_data is an address
// of course far_data itself (the location in memory containing the address

char* far_data = "apple";

// In memory it may be layed out like this:

// Address Label      Value
// 0x123:  far_data = 0x456
// ...
// 0x456:  string   = 'a'

// if PC was at 0x100
// LDI R0 0x023
// would load 'a' into R0
```

和 ADD 类似，将值放到 DR 之后需要更新条件标志位：

> The condition codes are set based on whether the value loaded is negative, zero, or positive. (Pg. 532)

下面是我对 LDI 的实现（后面章节中会介绍 mem_read）：

```
{
    uint16_t r0 = (instr >> 9) & 0x7; /* destination register (DR) */
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9); /* PCoffset 9*/

    /* add pc_offset to the current PC, look at that memory location to get
    reg[r0] = mem_read(mem_read(reg[R_PC] + pc_offset));
    update_flags(r0);
}
```

后面会看到，这些指令的实现中，大部分辅助功能函数都是可以复用的。

以上是两个例子，接下来就可以参考这两个例子实现其他的指令。注意本文中有两个指令是没有用到的：OP_RTI 和 OP_RES。你可以忽略这两个指令，如果执行到它们直接报错。将 main() 函数中未实现的 switch case 补全后，你的虚拟机主体就完成了！

6. 全部指令的参考实现

本节给出所有指令的实现。如果你自己的实现遇到问题，可以参考这里给出的版本。

6.1 RTI & RES

这两个指令本文没用到。

```
abort();
```

6.2 Bitwise and（按位与）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t r1 = (instr >> 6) & 0x7;
    uint16_t imm_flag = (instr >> 5) & 0x1;

    if (imm_flag) {
        uint16_t imm5 = sign_extend(instr & 0x1F, 5);
        reg[r0] = reg[r1] & imm5;
    } else {
        uint16_t r2 = instr & 0x7;
        reg[r0] = reg[r1] & reg[r2];
    }
    update_flags(r0);
}
```

6.3 Bitwise not（按位非）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t r1 = (instr >> 6) & 0x7;

    reg[r0] = ~reg[r1];
    update_flags(r0);
}
```

6.4 Branch（条件分支）

```
{
    uint16_t pc_offset = sign_extend((instr) & 0x1ff, 9);
    uint16_t cond_flag = (instr >> 9) & 0x7;
    if (cond_flag & reg[R_COND]) {
        reg[R_PC] += pc_offset;
    }
}
```

6.5 Jump（跳转）

RET 在规范中作为一个单独的指令列出，因为在汇编中它是一个独立的关键字。但是，RET 本质上是 JMP 的一个特殊情况。当 R1 为 7 时会执行 RET。

```
{
    /* Also handles RET */
    uint16_t r1 = (instr >> 6) & 0x7;
    reg[R_PC] = reg[r1];
}
```

6.6 Jump Register（跳转寄存器）

```
{
    uint16_t r1 = (instr >> 6) & 0x7;
    uint16_t long_pc_offset = sign_extend(instr & 0x7ff, 11);
    uint16_t long_flag = (instr >> 11) & 1;

    reg[R_R7] = reg[R_PC];
    if (long_flag) {
        reg[R_PC] += long_pc_offset;  /* JSR */
    } else {
        reg[R_PC] = reg[r1]; /* JSRR */
    }
    break;
}
```

6.7 Load（加载）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9);
    reg[r0] = mem_read(reg[R_PC] + pc_offset);
    update_flags(r0);
}
```

6.8 Load Register（加载寄存器）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t r1 = (instr >> 6) & 0x7;
    uint16_t offset = sign_extend(instr & 0x3F, 6);
    reg[r0] = mem_read(reg[r1] + offset);
    update_flags(r0);
}
```

6.9 Load Effective Address（加载有效地址）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9);
    reg[r0] = reg[R_PC] + pc_offset;
    update_flags(r0);
}
```

6.10 Store（存储）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9);
    mem_write(reg[R_PC] + pc_offset, reg[r0]);
}
```

6.11 Store Indirect（间接存储）

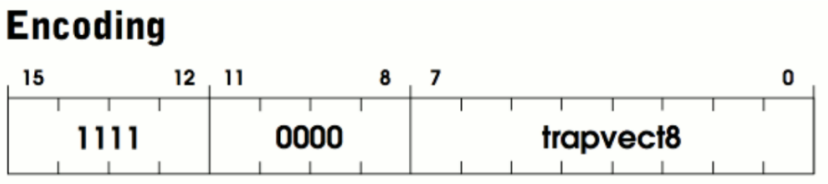
```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9);
    mem_write(mem_read(reg[R_PC] + pc_offset), reg[r0]);
}
```

6.12 Store Register（存储寄存器）

```
{
    uint16_t r0 = (instr >> 9) & 0x7;
    uint16_t r1 = (instr >> 6) & 0x7;
    uint16_t offset = sign_extend(instr & 0x3F, 6);
    mem_write(reg[r1] + offset, reg[r0]);
}
```

7. Trap Routines（中断陷入例程）

LC-3 提供了几个预定于的函数（过程），用于执行常规任务以及与 I/O 设备交换， 例如，用于从键盘接收输入的函数，在控制台上显示字符串的函数。这些都称为 trap routines，你可以将它们当做操作系统或者是 LC-3 的 API。每个 trap routine 都有一个对应的 trap code（中断号）。要执行一次捕获， 需要用相应的 trap code 执行 TRAP 指令。



定义所有 trap code：

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

你可能会觉得奇怪：为什么 trap code 没有包含在指令编码中？这是因为它们没有给 LC-3 带来任何新功能，只是提供了一种方便地执行任务的方式（和 C 中的系统函数类似 ）。在官方 LC-3 模拟器中，trap routines 是用汇编实现的。当调用到 trap code 时，PC 会移动到 code

对应的地址。CPU 执行这个函数（ procedure）的指令流，函数结束后 PC 重置到 trap 调用之前的位置。

> 注：这就是为什么程序从 0x3000 而不是 0x0 开始的原因。低地址空间是特意留出来 给 trap routine 用的。

规范只定义了 trap routine 的行为，并没有规定应该如何实现。在我们这个虚拟机中， 将会用 C 实现。当触发某个 trap code 时，会调用一个相应的 C 函数。这个函数执行 完成后，执行过程会返回到原来的指令流。

虽然 trap routine 可以用汇编实现，而且物理的 LC-3 计算机也确实是这样做的，但对虚 拟机来说并不是非常合适。相比于实现自己的 primitive I/O routines，我们可以利用操 作系统上已有的。这样可以使我们的虚拟机运行更良好，还简化了代码，提供了一个便于移 植的高层抽象。

> 注：从键盘获取输入就是一个例子。汇编版本使用一个循环来持续检查键盘有没有输入，这会消耗大量 CPU 而实际上没做多少事情！使用操作系统提供的某个合适的输入函 数的话，程序可以在收到输入之前一直 sleep。

TRAP 处理逻辑：

```
switch (instr & 0xFF) {
    case TRAP_GETC: {TRAP GETC, 9} break;
    case TRAP_OUT: {TRAP OUT, 9} break;
    case TRAP_PUTS: {TRAP PUTS, 8} break;
    case TRAP_IN: {TRAP IN, 9} break;
    case TRAP_PUTSP: {TRAP PUTSP, 9} break;
    case TRAP_HALT: {TRAP HALT, 9} break;
}
```

和前面几节类似，我会拿一个 trap routine 作为例子展示如何实现，其他的留给读者自己 完成。

7.1 PUTS

PUT trap code 用于输出一个以空字符结尾的字符串（和 C 中的 printf 类似）。规 范见 543 页。

显示一个字符串需要将这个字符串的地址放到 R0 寄存器，然后触发 trap。规范中说：

> 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. (Pg. 543)

意思是说字符串是存储在一个连续的内存区域。注意这里和 C 中的字符串有所不同：C 中每个字符占用一个 byte；LC-3 中内存寻找是 16 位的，每个字符都是 16 位，占用 两个 byte。因此要用 C 函数打印这些字符，需要将每个值先转换成 char 类型再输出：

```
{
    /* one char per word */
```

```
    uint16_t* c = memory + reg[R_R0];
    while (*c) {
        putc((char)*c, stdout);
        ++c;
    }
    fflush(stdout);
}
```

这就是 PUTS trap routine 的实现了。如果熟悉 C 的话，这个函数应该很容易理解。现 在你可以按照 LC-3 规范，自己动手实现其他的 trap routine 了。

8. Trap Routine 参考实现

本节给出所有 trap routine 的一份参考实现。

8.1 输入单个字符（Input Character）

```
/* read a single ASCII char */
reg[R_R0] = (uint16_t)getchar();
```

8.2 输出单个字符（Output Character）

```
putc((char)reg[R_R0], stdout);
fflush(stdout);
```

8.3 打印输入单个字符提示（Prompt for Input Character）

```
printf("Enter a character: ");
char c = getchar();
putc(c, stdout);
reg[R_R0] = (uint16_t)c;
```

8.4 输出字符串（Output String）

```
{
    /* one char per byte (two bytes per word) here we need to swap back to
       big endian format */
    uint16_t* c = memory + reg[R_R0];
    while (*c) {
        char char1 = (*c) & 0xFF;
        putc(char1, stdout);
        char char2 = (*c) >> 8;
        if (char2) putc(char2, stdout);
        ++c;
    }
    fflush(stdout);
}
```

8.5 暂停程序执行（Halt Program）

```
puts("HALT");
fflush(stdout);
running = 0;
```

9. 加载程序

前面提到了从内存加载和执行指令，但指令是如何进入内存的呢？将汇编程序转换为 机器码时，得到的是一个文件，其中包含一个指令流和相应的数据。只需要将这个文件的内 容复制到内存就算完成加载了。

程序的前 16 比特规定了这个程序在内存中的起始地址，这个地址称为 origin。因此 加载时应该首先读取这 16 比特，确定起始地址，然后才能依次读取和放置后面的指令及数 据。

下面是将 LC-3 程序读到内存的代码：

```
void read_image_file(FILE* file) {
    uint16_t origin; /* the origin tells us where in memory to place the im
    fread(&origin, sizeof(origin), 1, file);
    origin = swap16(origin);

    /* we know the maximum file size so we only need one fread */
    uint16_t max_read = UINT16_MAX - origin;
    uint16_t* p = memory + origin;
    size_t read = fread(p, sizeof(uint16_t), max_read, file);

    /* swap to little endian */
    while (read-- > 0) {
        *p = swap16(*p);
        ++p;
    }
}
```

注意读取前 16 比特之后，对这个值执行了 swap16()。这是因为 LC-3 程序是大端 （big-endian），但现在大部分计算机都是小端的（little-endian），因此需要做大小端 转换。如果你是在某些特殊的机器 （例如 PPC）上运行，那就不 需要这些转换了。

```
uint16_t swap16(uint16_t x) {
    return (x << 8) | (x >> 8);
}
```

> 注：大小端（Endianness）是指对于 一个整型数据，它的每个字节应该如何解释。在小端中，第一个字节是最低位，而在大端 中刚好相反，第一个字节是最高位。据我所知，这个顺序完全是人为规定的。不同的公司 做出的抉择不同，因此我们这些后来人只能针对大小端做一些特殊处理。要理解本文中大 小端相关的内容，知道这些就足够了。

我们再封装一下前面加载程序的函数，接受一个文件路径字符串作为参数，这样更加方便：

```
int read_image(const char* image_path) {
    FILE* file = fopen(image_path, "rb");
    if (!file) { return 0; };
    read_image_file(file);
    fclose(file);
```

```
    return 1;
}
```

10. 内存映射寄存器（Memory Mapped Registers）

某些特殊类型的寄存器是无法从常规寄存器表（register table）中访问的。因此，在内 存中为这些寄存器预留了特殊的地址。要读写这些寄存器，只需要读写相应的内存地址。这些称为 内存映射寄存器（MMR）。内存映射寄存器通常用于处理与特殊硬件的交互。

LC-3 有两个内存映射寄存器需要实现，分别是：

- KBSR：键盘状态寄存器（keyboard status register），表示是否有键按下
- KBDR：键盘数据寄存器（keyboard data register），表示哪个键按下了 虽然可以用 GETC 来请求键盘输入，但这个 trap routine 会阻塞执行，知道从键盘获得 了输入。KBSR 和 KBDR 使得我们可以轮询设备的状态然后继续执 行，因此程序不会阻塞。

```
enum {
    MR_KBSR = 0xFE00, /* keyboard status */
    MR_KBDR = 0xFE02  /* keyboard data */
};
```

内存映射寄存器使内存访问稍微复杂了一些。这种情况下不能直接读写内存位置，而要使 用 setter 和 getter 辅助函数。当获取输入时，getter 会检查键盘输入并更新两 个寄存器（也就是相应的内存位置）。

```
void mem_write(uint16_t address, uint16_t val) {
    memory[address] = val;
}

uint16_t mem_read(uint16_t address)
{
    if (address == MR_KBSR) {
        if (check_key()) {
            memory[MR_KBSR] = (1 << 15);
            memory[MR_KBDR] = getchar();
        } else {
            memory[MR_KBSR] = 0;
        }
    }
    return memory[address];
}
```

这就是我们的虚拟机的最后一部分了！只要你实现了前面提到的 trap routine 和指令，你 的虚拟机就即将能够运行了！

11. 平台相关的细节

本节包含一些与键盘交互以及显示相关的代码。如果不感兴趣可以直接复制粘贴。

如果不是在 Unix 类系统上运行本程序，例如 Windows，那本节内容需要替换为相应的平台实现。

```
uint16_t check_key() {
    fd_set readfds;
    FD_ZERO(&readfds);
```

```
    FD_SET(STDIN_FILENO, &readfds);

    struct timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = 0;
    return select(1, &readfds, NULL, NULL, &timeout) != 0;
}
```

下面是特定于 Unix 的设置终端输入的代码：

```
struct termios original_tio;

void disable_input_buffering() {
    tcgetattr(STDIN_FILENO, &original_tio);
    struct termios new_tio = original_tio;
    new_tio.c_lflag &= ~ICANON & ~ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &new_tio);
}

void restore_input_buffering() {
    tcsetattr(STDIN_FILENO, TCSANOW, &original_tio);
}
```

当程序被中断时，我们需要将终端的设置恢复到默认：

```
void handle_interrupt(int signal) {
    restore_input_buffering();
    printf("\n");
    exit(-2);
}
signal(SIGINT, handle_interrupt);
disable_input_buffering();
```

12. 运行虚拟机

现在你可以编译和运行这个 LC-3 虚拟机了！

使用你喜欢的 C 编译器编译这个虚拟机（ https://arthurchiao.art/assets/img/write-your-own-virtual-machine-zh/lc3-vm.c ），然后下载汇 编之后的两个小游戏：

- **2048** 下载: https://arthurchiao.art/assets/img/write-your-own-virtual-machine-zh/2048.obj
- **Rogue** 下载: https://justinmeiners.github.io/lc3-vm/supplies/rogue.obj

用如下命令执行：lc3-vm path/to/2048.obj。

```
Play 2048!
{2048 Example 13}
Control the game using WASD keys.
Are you on an ANSI terminal (y/n)? y
+-------------------------+
|                         |
|                         |
|                         |
|                    2    |
|                         |
```

```
|   2                       |
|                           |
|                           |
|                           |
+---------------------------+
```

调试

如果程序不能正常工作，那可能是你的实现有问题。调试程序就有点麻烦了。我建议通读 LC-3 程序的汇编源代码，然后使用一个调试器单步执行虚拟机指令，确保虚拟机执行到 的指令是符合预期的。如果发现了不符合预期的行为，就需要重新查看 LC-3 规范，确认你 的实现是否有问题。

13. C++ 实现（可选）

使用 C++ 会使代码更简短。本节介绍 C++ 的一些实现技巧。

C++ 有强大的编译时泛型（compile-time generics）机制，可以帮我们自动生成部分指令 的实现代码。这里的基本思想是重用每个指令的公共部分。例如，好几条指令都用到了间接 寻址或有符号扩展然后加到当前寄存器的功能。模板如下：

```
{Instruction C++ 14}
template <unsigned op>
void ins(uint16_t instr) {
    uint16_t r0, r1, r2, imm5, imm_flag;
    uint16_t pc_plus_off, base_plus_off;

    uint16_t opbit = (1 << op);
    if (0x4EEE & opbit) { r0 = (instr >> 9) & 0x7; }
    if (0x12E3 & opbit) { r1 = (instr >> 6) & 0x7; }
    if (0x0022 & opbit) {
        r2 = instr & 0x7;
        imm_flag = (instr >> 5) & 0x1;
        imm5 = sign_extend((instr) & 0x1F, 5);
    }
    if (0x00C0 & opbit) {   // Base + offset
        base_plus_off = reg[r1] + sign_extend(instr & 0x3f, 6);
    }
    if (0x4C0D & opbit) { // Indirect address
        pc_plus_off = reg[R_PC] + sign_extend(instr & 0x1ff, 9);
    }
    if (0x0001 & opbit) {
        // BR
        uint16_t cond = (instr >> 9) & 0x7;
        if (cond & reg[R_COND]) { reg[R_PC] = pc_plus_off; }
    }
    if (0x0002 & opbit) {  // ADD
        if (imm_flag) {
            reg[r0] = reg[r1] + imm5;
        } else {
            reg[r0] = reg[r1] + reg[r2];
        }
    }
    if (0x0020 & opbit) {  // AND
        if (imm_flag) {
```

```
            reg[r0] = reg[r1] & imm5;
        } else {
            reg[r0] = reg[r1] & reg[r2];
        }
    }
    if (0x0200 & opbit) { reg[r0] = ~reg[r1]; } // NOT
    if (0x1000 & opbit) { reg[R_PC] = reg[r1]; } // JMP
    if (0x0010 & opbit) {  // JSR
        uint16_t long_flag = (instr >> 11) & 1;
        pc_plus_off = reg[R_PC] +  sign_extend(instr & 0x7ff, 11);
        reg[R_R7] = reg[R_PC];
        if (long_flag) {
            reg[R_PC] = pc_plus_off;
        } else {
            reg[R_PC] = reg[r1];
        }
    }

    if (0x0004 & opbit) { reg[r0] = mem_read(pc_plus_off); } // LD
    if (0x0400 & opbit) { reg[r0] = mem_read(mem_read(pc_plus_off)); } // LI
    if (0x0040 & opbit) { reg[r0] = mem_read(base_plus_off); }  // LDR
    if (0x4000 & opbit) { reg[r0] = pc_plus_off; } // LEA
    if (0x0008 & opbit) { mem_write(pc_plus_off, reg[r0]); } // ST
    if (0x0800 & opbit) { mem_write(mem_read(pc_plus_off), reg[r0]); } // ST
    if (0x0080 & opbit) { mem_write(base_plus_off, reg[r0]); } // STR
    if (0x8000 & opbit) {  // TRAP
         {TRAP, 8}
    }
    //if (0x0100 & opbit) { } // RTI
    if (0x4666 & opbit) { update_flags(r0); }
}
{Op Table 14}
static void (*op_table[16])(uint16_t) = {
    ins<0>, ins<1>, ins<2>, ins<3>,
    ins<4>, ins<5>, ins<6>, ins<7>,
    NULL, ins<9>, ins<10>, ins<11>,
    ins<12>, NULL, ins<14>, ins<15>
};
```

这里的技巧是从 Bisqwit's NES emulator 学来的。如果你对仿真或 NES 感兴趣，强烈建议观看他的视频。

完整版 C++ 实现见:
https://justinmeiners.github.io/lc3-vm/src/lc3-alt.cpp

原文地址:
https://arthurchiao.art/blog/write-your-own-virtual-machine-zh/