

实验二 汇编以及程序的执行

一、实验目的

通过LC-3汇编程序的编写，理解：

1. 虚拟机运行的代码；
2. 虚拟机程序的执行过程。

二、实验内容

1. 汇编

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 the
program will be loaded
LEA R0, HELLO_STR          ; load the address of the HELLO_STR string
into R0
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
```

和C语言类似，这段程序从最上面开始，每次执行一条声明（statement）。但和C语言不同的是，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实验一中定义的操作码（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。

Assembly (text)

```
ADD R0, R0, 3
```

Assembler

Machine code (binary)

```
0001 000 000 1 00011
```

CPU/VM

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` 会在它所在其所在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 R1
BRn LOOP                ; go back to LOOP if the result was negative
...                     ; R0 is now 10!
```

2. 执行程序

前面的例子从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，将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事情。图灵曾经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

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。

“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}
}
```