实验三 指令的实现

一、实验目的

正确地实现每一条指令，指令可以参考实验相关材料中的文档（学习通7.1节中的`The LC-3 ISA`）。需要参照上述文档的描述实现LC-3的指令。

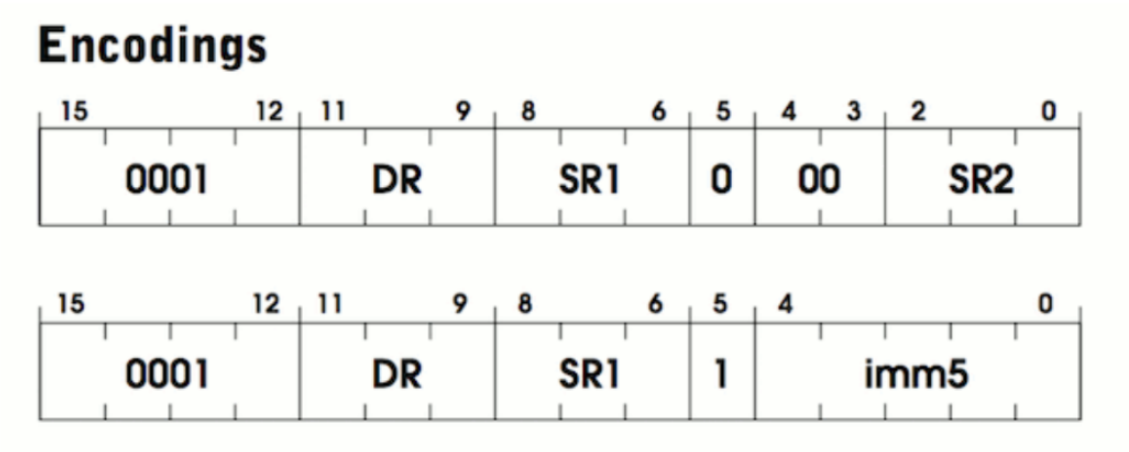
1. 实现全部指令；
2. 实现中断例程。

二、实验内容

1. 指令的实现

1.1 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，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)

这段解释也就是之前讨论的内容。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：更多二进制表示负数的相关信息，可以查询二进制补码（Two's Complement）。

规范中还有一处需要注意：

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

之前定义的那个条件标记枚举类型需要在此处应用。每次有值写到寄存器时，需要更新这个标记，以标明这个值的符号。为了方便，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;
    }
}
```

基于上述的功能，可以实现`ADD`的逻辑：

```
{
    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
mode */

    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);
}
```

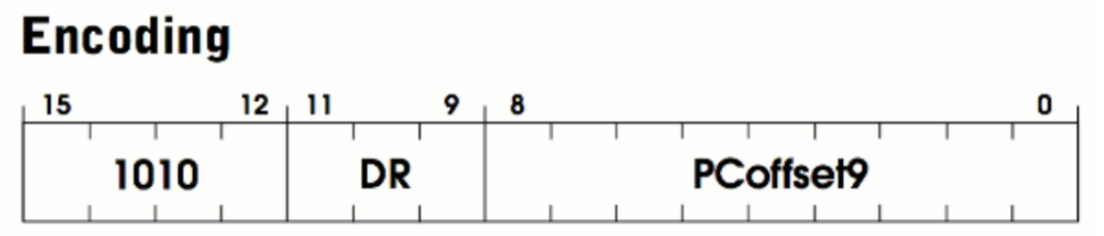
进行总结：

- `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，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。

1. 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，`LDI` 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，因此可以推测，`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，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，但它却又是不可或缺的。`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语言中有一个局部变量，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) has
an 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)

下面是对 `LDI` 的参考实现（后面章节中会介绍 mem_read），同学们也可以根据自己的理解进行实现：

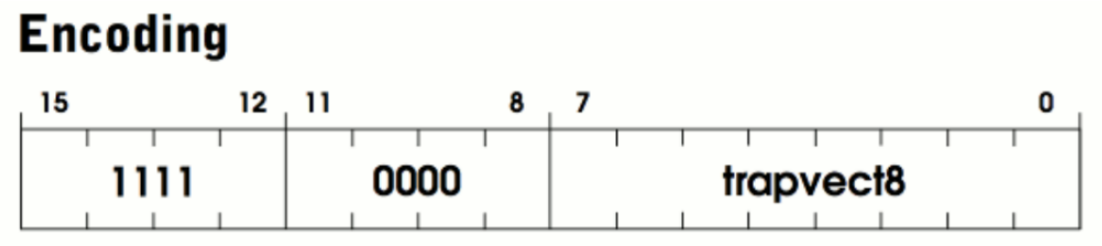
```
{
    uint16_t r0 = (instr >> 9) & 0x7;                       /* destination
register (DR) */
    uint16_t pc_offset = sign_extend(instr & 0x1ff, 9);     /* PCoffset 9*/

    /* add pc_offset to the current PC, look at that memory location to get the
final address */
    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它的指令。注意本文中有两个指令是没有用到的：`OP_RTI` 和 `OP_RES`。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，即可完成虚拟机的主体。

2. Trap Routines（中断陷入例程）

LC-3提供了几个预定的函数（过程），用于执行常规任务以及与I/O设备的交换。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，可以将它们视作操作系统或者是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
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 */
};
```

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语言中的系统函数类似）。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语言实现。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为良好，并且简化了代码，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;
}
```

与指令实现的类似，此处给出一个trap routine作为例子展示如何实现，其它的留给同学们自己完成。

2.1 PUTS

PUT trap code用于输出一个以空字符结尾的字符串（和C语言中的`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`。规范如下：

> 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)

字符串是存储在一个连续的内存区域。注意这里和C语言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语言的话，这个函数应该很容易理解。现在同学们可以按照LC-3规范，自己动手实现其它的trap routine了。

三、全部指令的参考实现

本节给出所有指令的参考实现。如果在实验中遇到问题，可以参考这里给出的版本。

1. RTI & RES

这两个指令本实验中没有用到。

```
abort();
```

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);
}
```

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);
}
```

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;
    }
}
```

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`RET`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. 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;
}
```

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);
}
```

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);
}
```

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);
}
```

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]);
}
```

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]);
}
```

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]);
}
```

四、Trap Routine的参考实现

本节给出所有trap routine的参考实现。

1. 输入单个字符（Input Character）

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

2. 输出单个字符（Output Character）

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

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;
```

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);
}
```

5. 暂停程序执行（Halt Program）

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