

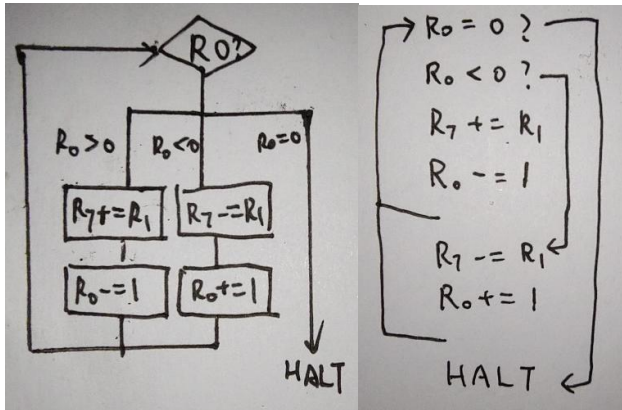
Lab 01 实验报告

PB20000296 郑腾飞

L 版本:

1 初始架构

核心的思路是，既然需要更少行数的代码，就用整数乘法的原始含义，靠连续增减得到目标，因此构成了下方左侧的流程图，改为 LC-3 可实现的流程为右侧：



具体代码为：

```
1001 010 001 111111
0001 010 010 1 00001 ; R2 = -R1
0001 000 000 1 00000 ; set NZP to the condition of R0
0000 010 000000111 ; if R0 = 0, end the program
0000 100 000000011 ; if R0 < 0, skip the next part
0001 111 111 000 001 ; R7 += R1
0001 000 000 1 11111 ; R0 -= 1
0000 111 111111011 ; go back
0001 111 111 000 010 ; R7 += R2
0001 000 000 1 00001 ; R0 += 1
0000 111 111111000 ; go back
```

共 11 行。

2 优化尝试

首先想到的思路是，若 $R0 < 0$ ，则直接翻转 $R0$ 与 $R1$ ，之后便不用再进行讨论。这个思路的具体代码为：

```
0001 000 000 1 00000 ; set NZP to the condition of R0
0000 001 000000100 ; if R0 > 0, skip the next part
1001 001 001 111111
0001 001 001 1 00001 ; R1 = -R1
1001 000 000 111111
```

```

0001 000 000 1 00001 ; R0 = -R0
0000 010 00000011 ; if R0 = 0, end the program
0001 111 111 000 001 ; R7 += R1
0001 000 000 1 11111 ; R0 -= 1
0000 111 111111100 ; go back

```

共 10 行。

接着，一个更大胆的想法诞生了：如果不顾一切，只要 R_0 不为 0 就将它直接减 1，接着 $R_7 += R_1$ ，计算结果会是如何？注意到， R_0 为负数时，补码表示的实质是 $2^{16} + R_0$ 也即 R_0 在 $\text{mod } 2^{16}$ 下的余数，因此可直接计算得此时计算结果为 $((2^{16} + R_0) \times R_1) = R_1 \times 2^{16} + R_0 \times R_1$ 。由于此式在 $\text{mod } 2^{16}$ 下的余数即为 $R_0 \times R_1$ ，因此结果很有可能是正确的，写出代码开始测试。

```

0001 000 000 1 00000 ; set NZP to the condition of R0
0000 010 00000011 ; if R0 = 0, end the program
0001 111 111 000 001 ; R7 += R1
0001 000 000 1 11111 ; R0 -= 1
0000 111 111111100 ; go back

```

Registers			Memory			
R0	xFFE9	-23	▶ x3000	x1020	4128	0001000000100000
R1	x007B	123	▶ x3001	x0407	1031	0000010000000111
R2	x0000	0	▶ x3002	x1FC1	8129	0001111110000001
R3	x0000	0	▶ x3003	x103F	4159	0001000000111111
R4	x0000	0	▶ x3004	x0FFC	4092	0000111111111100
R5	x0000	0	▶ x3005	xF025	-4059	1111000000100101
R6	x0000	0	▶ x3006	x0000	0	
R7	x0000	0	▶ x3007	x0000	0	

或许是由于循环的次数过多，虽然 R_7 最终得到了正确的结果，但运行极为缓慢。

Registers			Memory			
R0	x0000	0	▶ x3000	x1020	4128	0001000000100000
R1	x007B	123	▶ x3001	x0403	1027	0000010000000011
R2	x0000	0	▶ x3002	x1FC1	8129	0001111110000001
R3	x0000	0	▶ x3003	x103F	4159	0001000000111111
R4	x0000	0	▶ x3004	x0FFC	4092	0000111111111100
R5	x0000	0	▶ x3005	xF025	-4059	1111000000100101
R6	x0000	0	▶ x3006	x0000	0	
R7	xF4F3	-2829	▶ x3007	x0000	0	

此后，又发现判断和返回可以合成为一步，进一步精简：

```

0001 111 111 000 001 ; R7 += R1
0001 000 000 1 11111 ; R0 -= 1
0000 101 111111101 ; if R0 != 0, go back

```

这个程序当 R_0 初始为 0 时，要足足进行 65536 次循环才得出结论，虽然如此，这个程序确实足够短。对所有测试样例检查后，发现均通过，由此以这个三行的代码作为最终版本提交。

P 版本：

1 初始架构

核心的思路是，为减少执行指令数，我们每次计算 R0 对 R1 每一位的乘积，然后再将结果相加。

为了得到每一位，需要一个从 1 开始，每次提升一位的变量与 R1 取 AND，由此使用寄存器 R2 与 R3，得到代码：

```
0001 010 010 1 00001 ; R2 = 1
0101 011 010 000 001 ; R3 = R2 AND R1
0000 010 000000001 ; if R3 = 0, skip the adding
0001 111 111 000 000 ; R7 += R0
0001 000 000 000 000 ; R0 *= 2
0001 010 010 000 010 ; R2 *= 2
0000 101 111111010 ; if R2 != 0, go back
```

由于 R2 必然循环 16 次，每次执行指令为 5 或 6，包含第一句后，总指令数可能为 81 到 97，由此平均指令数亦在此两数之间。

2 优化尝试

首先，注意到，当 R0 在某次乘法后已经为 0 时，已不需要再循环下去，由此可增加判断：

```
0001 010 010 1 00001 ; R2 = 1
0101 011 010 000 001 ; R3 = R2 AND R1
0000 010 000000001 ; if R3 = 0, skip the adding
0001 111 111 000 000 ; R7 += R0
0001 000 000 000 000 ; R0 *= 2
0000 010 000000010 ; if R0 = 0, end the program
0001 010 010 000 010 ; R2 *= 2
0000 101 111111001 ; if R0 != 0, go back
```

但是，分析可发现，此判断在提前三轮及以上结束时才能减少指令数，因此可能无法起到实际效果，不过，进一步分析可发现，事实上可直接以 R0 判断终点：

```
0001 010 010 1 00001 ; R2 = 1
0101 011 010 000 001 ; R3 = R2 AND R1
0000 010 000000001 ; if R3 = 0, skip the adding
0001 111 111 000 000 ; R7 += R0
0001 010 010 000 010 ; R2 *= 2
0001 000 000 000 000 ; R0 *= 2
0000 101 111111010 ; if R0 != 0, go back
```

这种情况下，当 R0 与 R1 都取遍 -32768 到 32767 时，平均执行指令数约为 83.5，因

此已经达到了一定的优化。

此后，虽然有思路类似作业题中的方式将 **R1** 亦每次乘 2 已控制位数，但此时需要加的位并不正确，因此未能成功。将最后的代码作为最终版本提交。

Lab 02 实验报告

PB20000296 郑滕飞

递推关系：

实验中给出数据的递推式是 $F_n = F_{n-1} + 2F_{n-3} \bmod 1024$ ，为了以更少的代码行数实现递推，有两点值得注意：

首先，由于递推需要前三项的值，共需要同时保存四项，这里依次保存在 R4 到 R7。每次运算结束后，R4 到 R7 即存着连续的四项。进行下一步运算时，须先把前三项前移，也就是：

```
ADD R4, R5, #0 ;R5->R4
ADD R5, R6, #0 ;R6->R5
ADD R6, R7, #0 ;R7->R6
```

由于 F_n 的递推式中有 F_{n-1} 这一项，此时的 R7 恰好为 F_{n-1} ，因此不用重新清零，只需要：

```
ADD R7, R7, R4
ADD R7, R7, R4 ;R7 += 2R4
```

其次，关于 $\bmod 1024$ ，分析二进制可发现，这实质为保留最后的 10 位，前方清零，也即与 1023 取 AND。然而，1023 并不能用 immediate 模式直接表出，因此需要寄存器实现：

```
LD R1, INIT
.....
INIT .FILL #1023
```

在 R1 中存放 1023 后，递推中就可以直接：

```
AND R7, R7, R1
```

处理较小的 n：

程序中，真正运算的部分是从第 3 项开始，但测试的 n 可能为 1 或 2，因此需要单独考虑这两种情况。

由于 $F_0 = 1, F_2 = 2$ ，如果构造 F_{-1} ，值为 0.5，不能整数，更导致了 n 为 1 或 2 时情况的特殊。巧合的是， $F_n = n$ 对 1 与 2 都成立，因此可以一起处理。

将 R4 到 R7 赋予初值 1,1,2,2，然后进行判断：

```
ADD R0, R0, #-3
BRzp AGAIN
ADD R7, R0, #1
```

```
AGAIN [R7 = (R7 + 2R4) % 1024, R5->R4, R6->R5, R7->R6]
ADD R0, R0, #-1
BRzp AGAIN
```

如果 $n-3$ 不为负数，直接跳进循环中，每次将 n 减少 1，直到循环出结果。

当 n 为 1 或 2 时， $R7$ 必然循环一次后出来，值增加 2，只要使 $R7$ 变为 $R0 + 1$ ，最终输出的就是 $R0 - 3 + 1 + 2 = R0$ ，也就是我们想要的结果。

程序测试：

为确定程序是否正确，用 C++ 语言直接写出了对应的程序，并观察结果是否相同。

<pre>#include <iostream> using namespace std; int main(void) { int a[101]; a[0] = 1; a[1] = 1; a[2] = 2; for (int i = 3; i <= 100; i++) a[i] = (a[i-1] + 2 * a[i-3]) % 1024; for (int i = 0; i <= 100; i++) cout << i << " " << a[i] << endl; }</pre>	<pre>11 246 12 418 13 710 14 178 15 1014 16 386 17 742 18 722 19 470 20 930 21 326 22 242 23 54 24 706 25 166</pre>
--	---

在结果确定正确后，将学号 PB20000296 拆分为 20,00,02,96，对应的数字分别为 930,1,2,258 填在程序的最后。

Lab 03 实验报告

PB20000296 郑腾飞

Translate:

转化为汇编语言后可发现，程序的主要部分是一个循环：

```
AGAIN ADD R7, R6, #0
ADD R6, R5, #0
ADD R5, R4, #0
ADD R4, R5, R7
ADD R4, R4, R7
AND R4, R4, R1
ADD R0, R0, #-1
BRzp AGAIN
```

通过每次将新的位数下移以运算结果，可直接对较小初值进行处理，避免了单独处理引起的麻烦。

Guess:

将程序稍加修改，即可得到用于猜测的程序。

在程序的开头加上：

```
AND R3, R3, #0
ADD R3, R3, #-1
```

主循环的最后几行变为：

```
ADD R3, R3, #1
ADD R2, R0, R7
BRnp AGAIN
```

这样，在 R0 中存储想猜测的数的相反数后，当退出循环时，R3 就能显示其第一次出现的位置（往往也就是学号了）。

将最后四个数 930,10,146,34 的相反数输入后，得到 20,05,10,36，由此可知学号。

Optimize:

注意到，每次进行取模和全部运算完后再取模的结果是一样的，这是因为，即使产生溢出，溢出前后与 2^{16} 同余，而 1024 是 2^{16} 的因数，因此不影响同余。

其次，为了减少指令数，可以将大量其实没有实际价值的移动过程省去，每次改变三个数，具体操作为，将主循环改为：

```
AGAIN ADD R4, R4, R4
ADD R4, R4, R6
ADD R5, R5, R5
```

```
ADD R5, R5, R4
ADD R6, R6, R6
ADD R6, R6, R5
ADD R0, R0, -3
BRzp AGAIN
```

假设 R4,R5,R6 在上一次存储第 a,a+1,a+2 项, 一次循环后则存储 a+3,a+4,a+5 项。由于测试样例没有过小的 n, 不妨使 R4,R5,R6 存储 0,1,2 三项, 并在循环前先将 R0 减 3。这样, 可通过出循环时 R0 为 -3, -2, -1 来判断 R7 读取 R4,R5,R6 中的哪个。具体操作为:

```
ADD R0, R0, #2
BRn N
BRz Z
ADD R7, R6, #0
AND R7, R7, R1
HALT
N ADD R7, R4, #0
AND R7, R7, R1
HALT
Z ADD R7, R5, #0
AND R7, R7, R1
HALT
```

由此, 每次循环需要 8 条指令, 前进 3 位, n 很大时指令数近似 $\frac{8}{3}n$, 是优化前 8 条指令前进 1 位的指令数的三分之一。

Lab 04 实验报告

PB20000296 郑滕飞

Task 1:

第一个未知 bit 前的两行为：

```
LEA R2, #14 ; 由 PC 现在为 x3001 可知 R2 被赋值为 x300F
AND R0, R0, #0 ; 将 R0 清零
```

第三行为 JSR #x, 其中 x 可能为 0 或 1, 下一行 TRAP x25 表示停止, 由于不可能在程序刚开始运行就停止 (否则 R0 应该为 0 而非 5), 而是需要调用之后的子函数, x 必须为 1。此时, PC 的值 x3003 被存入 R7。

第五行将 R7 的值 x3003 存入了 R2 对应的地址, 也就是 x300F 中。

第六行, ADD R2, R2, #(8x+1), 此处暂时无法判断, 先搁置。

接下来两行是：

```
ADD R0, R0, #1 ; R0 增加 1, 现在为 1
LD R1, #17 ; R1 取第 x3008+x11=x3019 处的值#5
```

下一行为 ADD R1, R(4x+1), #-1, 此处没有任何理由使用到 R5, 因此 x 必为 0, R1 成为 4, 再然后 ST R1, #15, 将 R1 的值存回 x3019。

第十一行, BRz #1, 当前 R1 是 4, 因此进行下一行, JSR #-8, 回到第五行。这段循环为：

```
STR R7, R2, #0
ADD R2, R2, #(8x+1)
ADD R0, R0, #1
LD R1, #17
ADD R1, R1, #-1
ST R1, #15
BRz #1
JSR #-8
```

第十四行若为 JSR, 会有荒谬的结果, 由此其为 LDR, 从而写出循环后的部分为：

```
ADD R2, R2, #-1
LDR R7, R2, #0
RET
```

这其实构成了另一个循环, 当前面的 JSR 多次将地址写入 R2 对应的地址后, 此处的每个 -1 会使跳转的地址更前一位, 直到回到 x300F 所存储的 x3003。因此, 这两组循环实质上构成了一个递归。由此, 第二个 x 处必然为 0, 才能保证地址的正确跳转。

综合以上部分, 完成了这个递归程序后, 测试发现结果正确, 因此程序正确。

Task 2:

由于这个程序的功能已经写明了是计算 $\text{mod } 7$ 的余数，这里直接先把它转化为汇编语言，保留暂时未知的部分（方便阅读，将标签加粗表示）：

```
LD R1, INIT
JSR FUNCTION
AND R2, R1, #7
ADD R1, R2, R4
ADD R0, R?, #-7
BRp ?
ADD R0, R?, #-7
BRn ED
ADD R1, R1, #-7
ED HALT
FUNCTION AND R2, R2, #0
AND R3, R3, #0
AND R4, R4, #0
ADD R2, R2, #1
ADD R3, R3, #8
BACK AND R5, R3, R1
BRz NEXT
ADD R4, R2, R4
NEXT ADD R2, R2, R2
ADD R?, R3, R3
BR? BACK
RET
INIT .FILL #288
```

先分析 FUNCTION 中的部分。在初始化 R2 为 1，R3 为 8，R4 为 0 后，函数进入循环。循环中共有两个问号。由于整个循环中除了问号处并没有更改 R3 值的机会，问号必须为 R3，否则循环将永远进行或直接退出。

接下来分析循环的作用。由于 $8 \bmod 7$ 余 1， $8a+b \bmod 7$ 的余数与 $a+b$ 相同。因此，R3 实际上把所有第四位以上的部分右移三位后相加。这样的右移将进行到最高位，因此除非 R3 被左移为 0，函数都将继续，因此第二个问号处为 BRnp。

由此，我们得到了函数中循环的完整操作：

```
BACK AND R5, R3, R1
BRz NEXT
ADD R4, R2, R4
NEXT ADD R2, R2, R2
ADD R3, R3, R3
BRnp BACK
```

在一次调用函数结束后，R4 成为了 $R1 / 8$ （此处除法为保留整数部分），再经过程序

三四两行后，R1 变为了 $R1 / 8 + R1 \% 8$ ，mod 7 的余数与原来的 R1 相同，但当 $R1 > 7$ 时，最高非零位有所减少。

但是，此时的减少依然无法保证其直接为余数，由此，需要检测其是否还大于 7，因此接下来的两个问号处应该为：

```
ADD R0, R1, #-7
```

```
BRp #-5 ; 跳回调用 FUNCTION 处
```

如此循环后，R1 必然只剩后三位非零，这时的最后一步是：如果 R1 是 7，其 mod 7 的余数应为 0，而非 7，因此最后的问号仍为 R1，由此得出完整程序。

Lab 05 实验报告

PB20000296 郑腾飞

平方实现：

先考察程序中 $i * i \leq R0$ 的判断的实现方式。

首先，由于 $R0$ 范围的限制，此处的 i 为 100 以内的整数，因此可直接使用正数乘法的程序。此处假设 $R2$ 用于存储循环指标（即 i ），需要两个寄存器 $R3$ 、 $R4$ 进行操作：

```
ADD R3, R2, #0
AND R4, R4, #0
SQUARE ADD R4, R4, R2
ADD R3, R3, #-1
BRp SQUARE
```

在循环结束后， $R4$ 存放了 $R2$ 的平方。为了对比其与 $R0$ 的值，先将 $R4$ 取相反数，再与 $R0$ 相加：

```
NOT R4, R4
ADD R4, R4, #1
ADD R4, R4, R0
BRn RETURN
```

`RETURN` 标签指向 `RET` 处，代表当 $R0 - R2 * R2 < 0$ 时直接返回，由此即完成了 `while` 中条件的判断。

取模实现：

由于此处的 i 无法确定具体的值，没有简便的取模方式，因此需要将 $R0$ 反复减去 i ，直到结果可判断模。先以 $R3$ 保存 $R0$ ， $R4$ 保存 $-R2$ ：

```
ADD R3, R0, #0
NOT R4, R2
ADD R4, R4, #1
```

每次将 $R3$ 减 $R2$ ，直到结果为 0 或负数：

```
MOD ADD R3, R3, R4
BRp MOD
```

此时，如果 $R3$ 为 0，则说明 $R0 \% R2$ 为 0，因此可判断不为素数，将 $R1$ 的值重新变为 0，退出函数，否则重新开始循环：

```
BRn CYCLE
```

```
AND R1, R1, #0
RETURN RET
```

完整程序：

为与取模部分的程序相符合，选择在每次循环开始时增加 R2 的值，由此 R2 的初值应赋为 1，每次循环中增加。完整程序为：

```
ADD R1, R1, #1
JSR JUDGE
HALT
```

```
JUDGE ADD R2, R2, #1
CYCLE ADD R2, R2, #1
[R4 = R2 * R2 - R0]
BRn RETURN
```

```
[R3 = 0 if R0 % R2 = 0, else R3 < 0]
BRn CYCLE
```

```
AND R1, R1, #0
RETURN RET
```

中间的 CYCLE 部分相当于一个 do while 循环，如果终止时也没有将 R1 改为 0，R1 就会保留 1 的初值，代表 R0 为素数，否则 R1 成为 0，代表 R0 不为素数。

Lab 06 实验报告

PB20000296 郑腾飞

PART I Programs:

Lab 1 version L:

Lab 1 的 L 版本中，最终提交的是三行代码（由于二补数表示取模的原因，这三行代码在 R0 初值为负时仍可正确计算乘法，在第一篇报告中已具体证明，故此处不赘述）：

```
0001 111 111 000 001 ; R7 += R1
0001 000 000 1 11111 ; R0 -= 1
0000 101 111111101 ; if R0 != 0, go back
```

这实质上相当于一个 do while 循环（由于使用 BR 循环控制时，往往是在后方放置 BR 跳转至前方，比起 while 循环，do while 循环更常出现）。利用 C++，这三行代码可以合并为一步：

```
do r7 += r1; while (--r0);
```

在给 short 类型的 r0, r1 赋值，r7 初始化为 0 后，即可得到正确结果。

Lab 1 version P:

Lab 1 的 P 版本中，通过不断左移，与每一位取 AND 以完成乘法：

```
0001 010 010 1 00001 ; R2 = 1
0101 011 010 000 001 ; R3 = R2 AND R1
0000 010 000000001 ; if R3 = 0, skip the adding
0001 111 111 000 000 ; R7 += R0
0001 010 010 000 010 ; R2 *= 2
0001 000 000 000 000 ; R0 *= 2
0000 101 111111010 ; if R0 != 0, go back
```

这里的 AND 对应 C++ 中的 &（按位与），而判断是否相加与之后的右移在循环中的具体表达则为：

```
do {
    r3 = r2 & r1;
    if (r3) r7 += r0;
    r2 = r2 << 1;
    r0 = r0 << 1;
}
while (r0);
```

可以发现，LC-3 中的 BR 的逻辑往往与常用的 if 相反。if 的逻辑是【当满足条件，执

行某段代码】，而 BR 往往需要写成【当满足条件，跳过某段代码】。

Lab 2:

Lab 2 中，通过保存前三位所对应的数，与 1023 取 AND 以完成了对应的递推：

```
AGAIN ADD R7, R7, R4
ADD R7, R7, R4
AND R7, R7, R1
ADD R4, R5, #0
ADD R5, R6, #0
ADD R6, R7, #0
ADD R0, R0, #-1
BRzp AGAIN
```

利用 C++，这个循环可以写为：

```
do {
    r7 = (r7 + 2*r4) & r1;
    r4 = r5;
    r5 = r6;
    r6 = r7;
}
while (--r0 >= 0);
```

在循环之外，对初值较小的情况进行处理即可得到最终程序。

Lab 3:

优化后，主循环成为了每次三位的步进，可以写成：

```
do {
    r4 = 2 * r4 + r6;
    r5 = 2 * r5 + r4;
    r6 = 2 * r6 + r5;
    r0 -= 3;
}
while (r0 >= 0);
```

而最后的判断部分可以写为：

```
r0 += 2;
if (r0 < 0) r7 = r4 & r1;
else if (!r0) r7 = r5 & r1;
else r7 = r6 & r1;
cout << r7 << endl;
```

将两部分结合，增添初始化部分，即可得到完整程序。

Lab 4 task 1:

这段程序两次调用函数，利用递归把 R1 的值挪到了 R0 处。转化为 C++的主函数部分非常简单（cin 与 cout 是为了方便展示，原程序中 R1 即为 5）：

```
int main(void) {
    int r0 = 0, r1;
    cin >> r1;
    Recur(r0, r1);
    cout << r0 << endl;
}
```

而程序的主要部分，转化为汇编语言后是这样的：

```
STR R7, R2, #0
ADD R2, R2, #1
ADD R0, R0, #1
LD R1, #17
ADD R1, R1, #-1
ST R1, #15
BRz #1
JSR #-8
ADD R2, R2, #-1
LDR R7, R2, #0
RET
```

其中 R2,R7 都是为了控制递归的地址而设置，真正操作的是 R0 与 R1，由此，这部分递归的 C++代码为：

```
void Recur(int &r0, int &r1) {
    r0++;
    r1--;
    if (r1) Recur(r0, r1);
}
```

将主函数与递归部分结合，得到了最终的程序。

Lab 4 task 2:

这个程序的函数部分（已将机器语言转为汇编语言）利用 R2 与 R3 同时平移实现了右移操作，将 R4 储存为 R1 右移三位的结果：

```
FUNCTION AND R2, R2, #0
```



```

AND R3, R3, #0
AND R4, R4, #0
ADD R2, R2, #1
ADD R3, R3, #8
BACK AND R5, R3, R1
BRz NEXT
ADD R4, R2, R4
NEXT ADD R2, R2, R2
ADD R3, R3, R3
BRnp BACK
RET

```

将这段代码转换为 C++后如下：

```

int Devide8 (int r1) {
    short r2 = 1, r3 = 8, r4 = 0;
    do {
        if (r3 & r1) r4 += r2;
        r2 = r2 << 1;
        r3 = r3 << 1;
    }
    while (r3);
    return r4;
}

```

而主函数部分的循环是这样的：

```

do {
    r4 = Devide8(r1);
    r2 = r1 & 7;
    r1 = r2 + r4
}
while (r1 > 7);

```

最后，再判断 R1 是否为 7 即可处理输出。

Lab 5:

虽然是汇编语言从高级语言的程序进行的实现，但这个程序中涉及乘法、取模的部分都无法直接完成。

```

CYCLE ADD R2, R2, #1

```

```

ADD R3, R2, #0
AND R4, R4, #0

```

```

SQUARE ADD R4, R4, R2
ADD R3, R3, #-1
BRp SQUARE ; 计算平方
NOT R4, R4
ADD R4, R4, #1
ADD R4, R4, R0 ; 比较
BRn RETURN

```

```

ADD R3, R0, #0
NOT R4, R2
ADD R4, R4, #1
MOD ADD R3, R3, R4
BRp MOD ; 取模
BRn CYCLE

```

分为这几段后，可以写出对应的 C++ 程序中的循环：

```

do {
    r2++;
    r4 = 0;
    r3 = r2;
    do r4 += r2; while(--r3);
    r4 = r0 - r4;
    if (r4 < 0) return r1;
    r3 = r0;
    do r3 -= r2; while (r3 > 0);
}
while (r3);

```

补上对 R1 为 0 与 1 的控制即可得到完整的程序。

PART II Summary:

1. 评价高级语言程序的性能

值得注意的是，不能简单按照总共执行的指令条数来判定时间性能，这是由于指令本身所需要的复杂程度可能有很大的差别。例如，如果需要用 LC3 汇编，对整型数，右移会比左移多消耗时间，乘法的复杂程度可能是加减的数十倍，而除法的复杂程度更高。当循环次数很大时，这之间的差别是不能忽视的。

不过，对于 C++ 中的普通指令（即不调用函数的情况下），其具有较为固定的线性差别（例如需要 1 个 Cycle 与 100 个 Cycle 的差距），因此在估算时间复杂度时利用指令条数仍然是合理的，因为线性的差别不会影响相对 n 的量级， $O(n)$ 的程序在循环中的乘法变为加法时仍为 $O(n)$ 。

有关空间复杂度，我们所学习的程序在 LC3 中主要由寄存器进行操作，但当涉及计算

较为复杂时，往往仍需要利用内存。

2. 高级语言与 LC3 汇编对比

高级语言的方便之处主要有几点：

首先，它封装了一些基本的算术、逻辑运算，并且允许直接写较为复杂的表达式，而不用每次按步实现。头文件中也提供了大量功能各异的函数（如输入/输出），避免了和底层的直接交互引起的复杂。控制指令封装为 `if`、`for` 等，在逻辑上更易于理解。

其次，在函数调用方面，高级语言避免了手工对栈进行处理，使得嵌套调用、递归调用可以直接完成，减少了执行过程中的错误，也使得函数封装变得简单。区分地址传递、值传递，及时释放函数中的变量，可以方便进行模块化处理，不用每次都观察对整体的影响。

此外，高级语言利用结构化的思想，将数据分为不同类型，以不同方式组织、管理，减少了全部针对二进制码操作引起的模糊，也得以实现复杂的操作。

3. LC3 中值得添加的指令

单纯的 `JSR` 命令无法处理嵌套调用的情况，而每次 `JSR`、`LDR` 与 `STR` 的配合形成的操作实质上是将内存中的一列储存为地址序列，在调用结束后依次返回。由此，如果增加栈控制的命令，会更加方便处理嵌套、

此外，为编写更为复杂的程序，如减、乘、除等简单运算也值得成为命令。更进一步地，如果出现不同数据类型的情况，可能会需要更复杂的命令来完成。

数据移动指令方面，由于目前的指令都是内存与寄存器交互，可以考虑增添能直接在内存中控制数据的指令，这样可以省略寄存器的中转，也留给寄存器更多操作的余地。

在 `pseudo-instructions` 中，有一些利用 `TRAP` 进行的和字符串有关的操作，这方面操作或许也有增加指令的空间。

4. 从汇编语言中学到的思路

学到的最重要的思路或许是，对于位操作的一些妙用。在之前写高级语言程序时，往往习惯于以数据为单位进行运算、处理，但其中的有些部分用位操作其实可以更加优化。在 LC3 中，由于对数据整体的处理不多，很多时候被迫利用位进行操作，感受到了很多时候位的特殊使用方式带来的简化（这有点使我联想到了之前看过的《雷神之锤 III》中巧妙的快速平方根倒数算法，就是巧妙利用浮点数的位操作的结果）。

此外，汇编语言中对函数与跳转的操作，也增进了对高级语言中函数堆栈的理解，更加清楚其中的嵌套、递归的实现方式，通过优化函数结构来减少对函数栈的使用。

致谢

感谢老师、助教一学期以来的付出，让初入计科辅修的数院人了解了更加底层的内容，也感受到了计算机系统的乐趣^_^

Lab A 实验报告

PB20000296 郑滕飞

Trim:

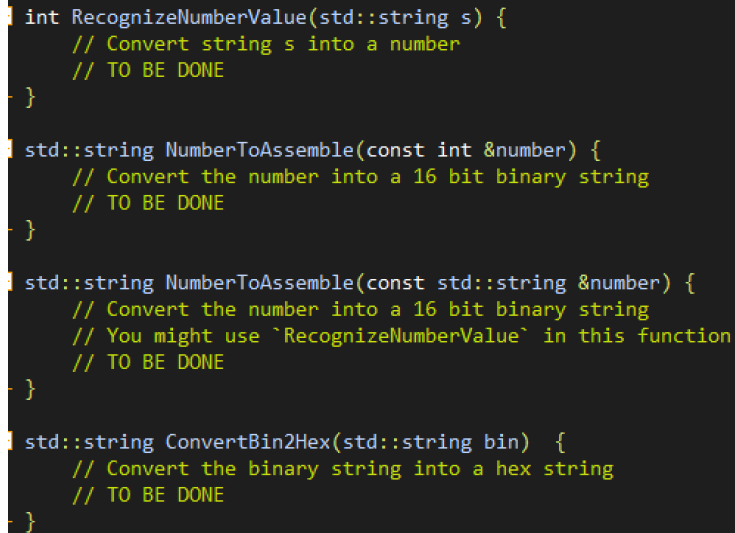
assembler.h 中需要补充去除左右空白的 Trim 方法。使用 string 类中的 find_first_not_of、find_last_not_of 与 erase 可以直接实现：

```
inline std::string &LeftTrim(std::string &s, const char *t = " \t\n\r\f\v")
{
    s.erase(0, s.find_first_not_of(t));
    return s;
}

inline std::string &RightTrim(std::string &s, const char *t = " \t\n\r\f\v")
{
    s.erase(s.find_last_not_of(t) + 1);
    return s;
}
```

Convert:

此部分的四个函数：



```
int RecognizeNumberValue(std::string s) {
    // Convert string s into a number
    // TO BE DONE
}

std::string NumberToAssemble(const int &number) {
    // Convert the number into a 16 bit binary string
    // TO BE DONE
}

std::string NumberToAssemble(const std::string &number) {
    // Convert the number into a 16 bit binary string
    // You might use `RecognizeNumberValue` in this function
    // TO BE DONE
}

std::string ConvertBin2Hex(std::string bin) {
    // Convert the binary string into a hex string
    // TO BE DONE
}
```

根据此后的调用分析可得，第一个函数需要将#开头的十进制数或X开头（由于在后方过程中已全部转为大写）的十六进制数转化成对应的数字（需要讨论十进制的正负与十六进制表示的二补数的正负）。后三个函数则不存在歧义，直接按照要求写出即可（其中字符串数转为十六位二进制码只需将前两个函数进行复合）。

Translate Operand:

根据之后的调用，此函数是为了将操作数转化为二进制码。在操作数为寄存器与直接数

时，直接调用之前的函数即可。否则，需要找到 Label 对应的地址后利用当前地址进行计算。根据之后对 Label 的处理可写出代码：

```
if (!(item.getType() == vAddress && item.getVal() == -1)) {
    // str is a label
    int diff = item.getVal() - (current_address + 1);
    return NumberToAssemble(str).substr(16 - opcode_length);
}
```

由于 PC 已经移动，计算 offset 时需要使用 current_address + 1。

Scan #0:

第一次扫描中，将文档的注释部分去除，其中有两步需要完成，一个是将字符串小写部分转为大写，一个是，当字符串的中间有分号时，将分号前后分为两部分分别保存，均可以直接写出代码。

Scan #1:

Pseudo Command: 第一次对 Pseudo Command 的扫描是为了记录不同的伪命令对应的地址数。FILL 对应一个地址，BLKW 对应此后操作数个数的地址，由此可计算结果。

Operation: 如果第一个词就代表操作码，直接将此行的类型设置为 lOperation 并返回。

Label: 当出现 Label 时，需要先将 Label 存入：

```
label_map.AddLabel(label_name, value_tp(vAddress, line_address - 1));
```

再对行地址进行调整，值得注意的是，由于在此前 line_address 已经增加了 1，之后的增加量需要比第一轮 Pseudo Command 时减少 1。

Scan #2:

Pseudo Command: 对 BLKW，需要重复插入“000000000000”或“x0000”，对 STRINGZ，则需要再读入一个词，然后按照次序插入每个字符（含\0）对应的 ASCII 码：

```
else if (word == ".STRINGZ") {
    // Fill string here
    line_stringstream >> word;
    for (int i = 1; i < word.length() - 1; i++) {
        auto output_line = NumberToAssemble((int)word[i]);
        if (gIsHexMode) output_line = ConvertBin2Hex(output_line);
        output_file << output_line << std::endl;
    }
    if (gIsHexMode) output_file << "x0000" << endl;
    else output_file << "0000000000000000" << endl;
}
```

Replace: 利用函数将逗号替换为空格即可。

LC3 Command: 类似给出的 ADD、BRNZP 与 ST 的例子直接转换即可, 如:

```
case 11:
    // "JSR"
    result_line += "01001";
    if (parameter_list_size != 1) {
        // @ Error parameter numbers
        return -30;
    }
    result_line += TranslateOprand(current_address, parameter_list[0], 11);
break;
case 12:
    // "JSRR"
    result_line += "0100000";
    if (parameter_list_size != 1) {
        // @ Error parameter numbers
        return -30;
    }
    result_line += TranslateOprand(current_address, parameter_list[0]);
    result_line += "000000";
break;
```

由此即补完了代码。

Lab S 实验报告

PB20000296 郑滕飞

main.cpp:

由于电脑没有 boost 库，为了正确用命令行控制输入，利用 Lab A 的主函数写作思路完成了所需的功能，并且保留了核心部分的代码：

```
42
43     auto input_info = getCmdOption(argv, argv + argc, "-f");
44     auto output_info = getCmdOption(argv, argv + argc, "-o");
45     auto register_info = getCmdOption(argv, argv + argc, "-r");
46
47     if (input_info.first) gInputFileName = input_info.second;
48
49     if (output_info.first) gOutputFileName = output_info.second;
50
51     if (register_info.first) gRegisterStatusFileName = register_info.second;
52
53
54     if (cmdOptionExists(argv, argv + argc, "-s")) {
55         gIsSingleStepMode = true;
56     }
57     if (cmdOptionExists(argv, argv + argc, "-d")) {
58         gIsDetailedMode = true;
59     }
60
61     virtual_machine tp virtual_machine(gBeginningAddress, gInputFileName, gRegisterStatusFileName);
62     int halt_flag = true;
63     int time_flag = 0;
64     while(halt_flag) {
65         // Single step
66         virtual_machine.NextStep();
67         if (gIsDetailedMode)
68             std::cout << virtual_machine.reg << std::endl;
69         ++time_flag;
70     }
71
72     std::cout << virtual_machine.reg << std::endl;
73     std::cout << "cycle = " << time_flag << std::endl;
74     return 0;
75 }
```

主函数中的 To Be Done 只有一处，也即模拟器前进一步，检查发现其为 NextStep 函数。此后，由于发现程序无法在 halt 时结束，将 halt_flag 传入 NextStep 作为参数，并且当程序到达 halt 时将其设置为 false。

memory.cpp:

对于 ReadMemoryFromFile，只需要每次从文件中读入一行，再用 memory.h 中提供的 TranslateInstruction 函数存入 memory 的对应位置即可，其他两函数均为直接返回 memory[address]，代表取出地址中的数。

SignExtend:

此函数根据后方调用可发现是对应类型的 int 数扩展至 16 位 int，利用左右移位可以直接实现：

```
if (x >> (B - 1)) return (-1 << B) | x;
return x;
```

UpdateCondRegister:

根据之后调用分析，CondRegister 可能在 n 位，z 位，p 位为 1，由于调用实质是取 AND，这三种情况分别对应 4,2,1，由此可写出代码：

```
if (reg[regname] > 0) reg[R_COND] = 1;
else if (reg[regname] < 0) reg[R_COND] = 4;
else reg[R_COND] = 2;
```

VM 函数部分:

类似已经给出的 VM_ADD, VM_BR 与 VM_LD, 可类似写出此部分的函数, 如:

```
void virtual_machine_tp::VM_JSR(int16_t inst) {
    int flag = inst & 0x0800;
    reg[R_R7] = reg[R_PC];
    if (flag) {
        //JSR
        int16_t pc_offset = SignExtend<int16_t, 11>(inst & 0x7FF);
        if (gIsDetailedMode)
            std::cout << reg[R_PC] << "+=" << pc_offset << std::endl;
        reg[R_PC] += pc_offset;
    }
    else {
        //JSRR
        int baseR = (inst >> 6) & 0x7;
        if (gIsDetailedMode)
            std::cout << reg[R_PC] << "->" << reg[baseR] << std::endl;
        reg[R_PC] = reg[baseR];
    }
}
```

NextStep:

直接将 case O_AND 的部分复制, 修改为对应指令即可。

由此即补完了代码。