赵俊宇 PB24061271

从零开始的机器码生存指南

——大作业实验报告

一． 前言

在经历一个月设计、半个月编程和半个月 debug 之后，这套机器码以及配套的运行器终于正式完工了。

实验要求部分，这些基本的指令，您可能暂时不知道我在说什么，这些都是指令名，到了操作手册部分我还会解释：加法有 ADD；减法可以使用符号位 ADD 一条指令，也可以使用 NOT+ADD，还可以使用 RES+ADD；乘法、取余共用一条指令 RTI，也可以采用循环即条件跳转 BR 与 ADD 等等实现；赋值可以用 ADD，也可以使用好几种 ST 与 LD 的搭配，甚至可以使用 LEA；条件跳转，我提供了 BR；至于返回，可以使用 TRAP 里的 end，如果您想输入、输出什么，TRAP 里当然也提供了这些功能。

值得单独说明，我把负数与溢出处理、输入方式、自定义的一些机器码规则以及测试文件的描述都放在了操作手册和思路介绍部分，请在编写和运行测试文件前阅读这两部分,当然我也把自己准备的样例放了进去，您可以直接使用。但是需要指出，由于各个系统的差异，平台兼容性成为了不小的问题。如果想要获得最佳体验，我推荐您在 Windows 的 powershell 里运行程序，因为全程采用 vscode 里的命令行测试，Linux 也有比较好的稳定性，但保留某些 bug 的可能性，~~边角的 bug 可能需要 Linux 用户自适应一下~~，我并不确定 Mac 能不能较好的实现这些功能。

和许多同学一样，在我刚开始看见大作业 4 的实验要求的时候，也是头皮发麻，不知所云，直到一点一点去参考已经有的机器码运行器，从网络上查资料自学位运算、研究 nzp 条件、编写各种测试程序，反复尝试数据类型转换，不断去查 LC-3 机器码的含义和功能，虽然在聊起大作业选题的时候，会反复收获看见外星人一样的眼神，不过总之最后，还是跌跌撞撞终于完成了实验。

至于这次作业，如果让我自己打分的话，我会给 9.5，它固然实现的并不完美，优化空间也很大，极端情况也会出现 bug。至少我已经探索到了自己现阶段能力的上限，所以它在我心里值得作为一个 10 分，献给从零开始学习 C 语言、从零开始了解机器码的自己，至于剩下的 0.5，就扣给那个当初纯凭兴趣、不知死活的自己。

二． 编程思路与实验方法介绍

总体介绍:

程序采用自顶向下设计，main.c 管理菜单，function.c 中集成了所有参与程序的函数，hardware.c 模拟了计算机内的硬件，function.h 中声明了函数，由一个函数来调用相应的指令，每个指令可以调用更加底层的小指令来处理数据，比如从 operand（操作码）里提取前五位转

化为 imm5（5 位立即数），这些函数的功能将会在后面介绍。，setting.txt 和 gdbfile.txt 有点类似于磁盘，他们的内容不会因为程序重启（类比于计算机重启）消失。整个程序的变量命名规则深度参考 LC-3，包括寄存器的顺序、指令等等也以其为框架，对部分指令进行了魔改。接下来，我已经迫不及待介绍我最最得意的两个功能实现。

一个最底层的更新条件码函数示例

```
void new_condition(unsigned short oper){
    short oper_= (short)oper;
    if(oper_>0) regis[9]=1;
    else if(oper_<0) regis[9]=4;
    else regis[9]=2;
}
```

1.1 数据读取：

这是已经迭代过一版的处理方法，在最开始的时候，每个 operand 是用一个 int a[16]表示，数据计算（以 imm5 为例）是 a[0]+2*a[1]+4*a[2]+8*a[3]，单看这样还好，那 PCoffset11（11 位程序计数器偏移量）呢？不仅繁琐，而且一个 int 只储存一个 1/0，是对计算机资源的浪费，在现在的程序中，还能看见一点点这种很笨的方案的残留代码发挥一部分功能。

于是，在学习实现位运算的时候，现在的方案出现了。一个 unsigned short 数据恰好是 16 位，利用率极高，一个数组就可以存储所有机器码。那我又该怎么获得 imm5 呢？没错，是位运算里的‘&’运算，我可以用 31（0b11111）和 operand 做与运算，这样就能获得 imm5，至于符号位，我们留到下一小节揭开悬念。

那想获得 12-16 位的指令码又该怎么做呢？既然是位运算，相信你已经猜到了，使用位运算里的‘>>’运算，例如，0001000000100001，右移位 12 位再与 15（0b1111）‘&’，就得到了 0001，即 ADD。在我的程序里，你会看见许多罗马数字命名的函数，比如 unsigned short IX_XI(unsigned short operand)就是对 10-12 位进行这样的运算，这些是最低层的计算函数。

1.2 负数处理

在这个程序中，要是说位运算是我第二得意的想法，那负数的实现就是第一。刚开始，我对于它的实现毫无头绪，直到我查了致死量的资料，决定参考真实的计算机。所以，相信你已经猜到了，我对 unsigned short 的运算是符号位扩展，原理是补码。我参考了：

计算机组成原理----原码，反码与补码_计算机组成原理补码-CSDN 博客

补码的计算方法 - 知乎

补码/反码、零扩展和符号位扩展（Zero extension and Sign extension）-CSDN 博客

在我的代码中，实现方法是先扩展符号位，再运算。读取符号位，还是以 imm5 为例，0001000000110010 是 5 位，那么 1U（无符号 1）左移 5 位 -1 就是 0b11111，与运算后是

我的程序实现。至于 unsigned short 和 short 的转换，以及 short 的函数类型，是我在 test 文件里遗留下的，事实上不会改变 operand，也就无伤大雅。

```
short pre_preparation(unsigned short oper,int num){
    short oper_=(short)(((1U << num)-1)&oper);
    if ((1&(oper_ >>(num-1))) ==1){
        oper_ =oper_ | (0b1111111111111111 << num);
    }
    else ;
    return (oper_);
}
```

0b0000000000010010。程序再次利用移位读取第五位，如果是 1，就把 0b1111111111111111 左移 5 位变为 0b1111111111100000，再与运算，最后是 0b1111111111110010。那为什么这么一通折腾就可以实现负数运算了呢？原因是，计算机就像是一个只有分针、只能顺时针的钟表，只会加，不会减，不在乎小时是否会+1，现在是 25 分，如果我想-1，我只需要等上 59 分钟。是的，这样运算，计算机会自动替我们处理溢出，这一部分就是补码的应用。还记得刚才的 imm5：0b1111111111110010 吗？采用补码的规则，最后它是 signed（0b1000000000001110），即-14。就这样，我们完成了负数的运算。

2.1 输入：

这部分的代码实现很常规，在这个部分，我提供了两种方式、四个选择，您可以使用手动输入，这种方式以空格分割，以回车结尾；您也可以选择从文件中读取，需要先输入含后缀名的文件名，再读取文件，这种方式以回车分割，以最后一个机器码结尾，不需要回车。需要说明的是，为了后面数据处理的时候好过一点，我的程序的输入方式比较严格，很多时候会因为某些幽灵字符或者'0'、'1'、'\n'，' '以外的字符报错。

```
****************************************
*Please choose the way you put in:       *
*I.  write by file(press 1)              *
*II. write in person(press 2)            *
*III.write by file(debug mode)(press 3)  *
*IV.write in person(debug mode)(press 4) *
****************************************
```

2.2 debug 模式：

在经历了毫无头绪的 debug 和测试文件编写中，这个模式出现了。这个运行器本质是一个黑盒，除非有 TRAP 指令输出，否则它不会输出任何内容，为了打开这个黑箱、方便调试，我增加了这个模式。在这个模式中，您可以看见 11 个 Register 运行前后的状态、您的每一条指令被简单的翻译成类似 LC-3 指令集的表述与编号、每个指令参与的 Register 的前后变化。这会大大方便您验证我的程序和您编写的测试文件的正确性。

```
before running:
0001111111101110 0
0001111111101010 1
1111000000000001 2
1111000000000010 3
0001111111111101 4
1111000000000001 5
1111000000000010 6
0001111111100111 7
1111000000000001 8
1111000000000010 9
1111000000000010 10
0001111111100011 11
1111000000000001 12
1111000000000010 13
0001111111110001 14
0001111111110001 15
0001111111110001 16
0001111111110001 17
0001111111110111 18
1111000000000001 19
1111000000000010 20
1111000000000100 21
```

```
regis_cd:0
regis_trap:0
regis_pc:0
regis[0] :0
regis[1] :0
regis[2] :0
regis[3] :0
regis[4] :0
regis[5] :0
regis[6] :0
regis[7] :0
```

```
nzp:4
after running
regis_cd:4
regis_trap:65498
regis_pc:18
regis[0] :-38
regis[1] :-9
regis[2] :9
regis[3] :0
regis[4] :0
regis[5] :0
regis[6] :0
regis[7] :-38
```

```
13
STI regis[3] offset:-1 ram[12]:14 ram[14]:4644

nzp:1
14
ADD resig[1] regis[0] imm5:4 4

nzp:1
15
ADD resig[7] regis[7] imm5:15 15

nzp:1
16
LDR regis[4] resig[4]:1

nzp:1
17
BR offest:1 regis_pc:17 regis_pc:17

nzp:1
18
STR regis[4] regis[7]:15 ram[19]:-4092 offset:4 1

nzp:1
```

```
//0000
void op_br(unsigned short operand){
    int target=nzp(operand);  //printf("$$%d$$",target);
        if(choice) {printf("BR ");}
    unsigned short pcoffset9 =U_IX(operand);
        if(choice) { printf("offest:%d regis[8]:%d",(short)pcoffset9,regis[8]);}
    if(target) regis[8]=regis[8]+pcoffset9;
        if(choice) {printf(" regis[8]:%d\n",(short)regis[8]);}
```

为什么要把输出语句写这么碎？

当然是为了方便确定每个函数有没有正常工作啦。

2.2.1 default mode：

在亿点点更新之后，我的程序现在可以"记住"你的选择，这样每次你不再需要每次运行都设置一堆选项，当然，你也可以选择 setting 选项来选择你的 reference。这些选项会保存在"setting.txt"中，等到你选择 default mode ，这些选择会被自动读取。值得注意的是，程序目录下需要有"setting.txt"，否则会报错，请按指引输入数字，输入字母会引起问题。

```
*V.default mode(press '5')              *
*VI.default mode setting(press '6')     *
```

```
****************************************
*the setting menu: (press 0 or 1 to switch)*
*I.  debug mode:       OFF               *
*II. GDB:              OFF               *
*III.HALT step by step:OFF               *
*IV.the input way:     FILE              *
*0: back to the former menu              *
****************************************
```

2.2.2 逐步暂停与 GDB 断点

相信你已经看见了，在 default 模式下，你可以打开逐步暂停和简易的 GDB 断点，需要提醒的是，如果这两个选项同时打开，那么相当于 GDB 模式不起作用，程序会优先逐步暂停。我的程序实现了 GDB 断点的追加、删除、创建和展示，这部分会在下面介绍，Linux 由于缓冲区问题，有时候可能需要按 1-2 下继续，请按指引输入数字，输入字母会引起问题。

```
1 codes have been done.Here is the registers:

regis_cd:1
regis_trap:0
regis_pc:1
regis[0] :9
regis[1] :0
regis[2] :0
regis[3] :0
regis[4] :0
regis[5] :0
regis[6] :0
regis[7] :0

Press any bottom to continue the program!
```

2.2.3 创建你自己的 GDB 断点文件

主菜单的这个选项“create your GDB setting file(press '7') “，可以打开断点调试界面。

```
****************************************
*please select choice for the gdb file: *
*I.create a new file(press '1')         *
*II.add more point(press '2')           *
*III.delete some point(press '3')       *
*IV.show me all the point(press '4')    *
*0.back to the former menu(press '0')   *
****************************************
```

这些分别是：创建新的 gdb 文件；增加更多的断点；删除一些断点；展示所有断点。

```
Press '0' to end
Please input the number of the code to cancel the point:
1
Error:the point doesn't exist!
Please input the number of the code to cancel the point:
6
Error:the point doesn't exist!
Please input the number of the code to cancel the point:
12
The point of instruction 12 has been canceled.
```

```
press '0' to end
Input the 3rd point:10
After the 10th instruction is done ,the program will be paused.
Input the 4th point:15
After the 15th instruction is done ,the program will be paused.
Input the 5th point:21
After the 21st instruction is done ,the program will be paused.
Input the 6th point:0
```

```
The 1st point is 21.
The 2th point is 14.
The 3rd point is 10.
The 4th point is 15.
The 5th point is 21.
All the point have been listed!Total: 5.
```

> 各种各样的断点功能展示
> 它们将被保存在“gdbfile.txt”中

2.3 register 与 nzp 条件:

在这个部分，我会介绍一些寄存器的含义：0-7 是七个最基本的寄存器，pc 是程序计数器，cd 是条件码寄存器，trap 是一个为了 TRAP 指令而特殊设计的寄存器，具体将在 TRAP 中提到。

关于 nzp 条件，简单来说就是通过 DR（目的寄存器）的正负更新条件码（函数在上文已出现），negative 是 4（0b100），zero 是 2（0b010），positive 是 1（0b001）。以 BR（0000）为例，如果是 0b0000100000000010，如果此时条件码是 0b100，就向后跳转两个，即忽略 BR 后的一个指令，如果是 0b010 或 0b001，就不去跳转。当然，你可以利用 PCoffset9 的符号位往回跳，做一个简单的一重循环，我的测试文件实现了这一点，还可以反复使用实现多重循环。

三． 操作手册

1.0 负数要怎么写:

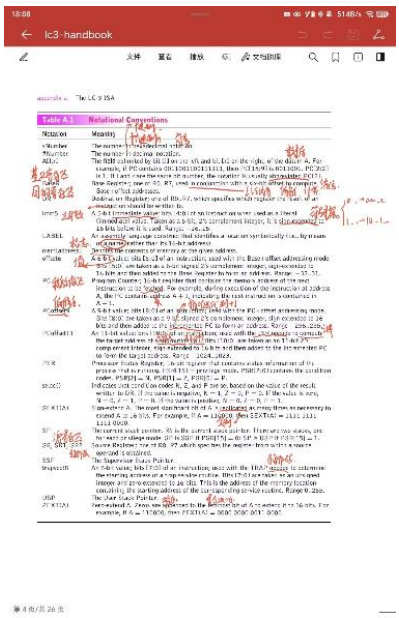
以 imm5 为例，如果是+4，就是 00100，那按照我的实现方法，-4 的符号位是 1，4 是 0b0100，取反是 0b1011，再加 1 是 0b1100，那么合起来就是 11100。

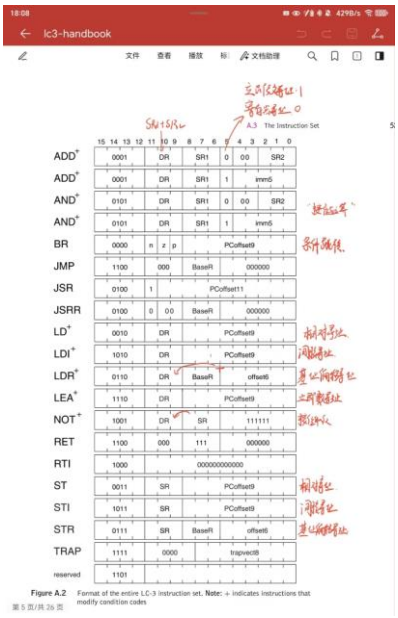
再比如 pcoffset9，如果是+1，就是 000000001，如果是-1，符号位是 1，1 是

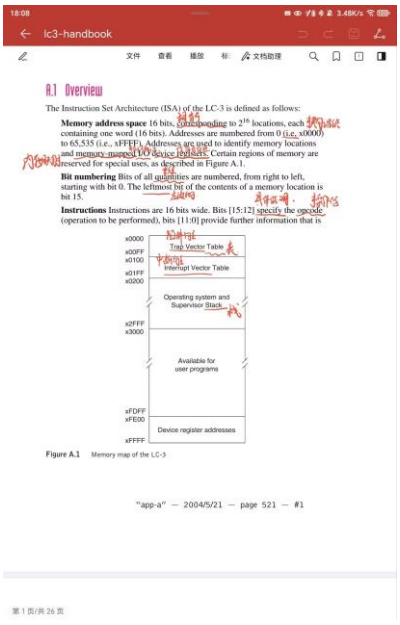
0b00000001，取反是 0b11111110，再加 1 是 0b11111111，合起来就是 111111111。

1.1 指令集：

参考 LC-3 isa，大部分指令的功能基本和它相似，代码里的命名规则也参考它。这里我贴上我的学习笔记和指令集图片做参考，具体功能按下面这个链接的描述实现。







[LC-3 机器语言 指令集-CSDN 博客]

1.2 和指令集的不同之处：

跳转：注意，第一个指令码在数组中是 RAM[1]，在跳转时尽量不要越界，原因见段尾。为什么是 RAM[1]而非 0？原因是为了寻址指令做了让步，比如你想把第 11 个机器码用 LEA 存进 DR，或是用 STI 的过程中想存第 11 个，那这样会更方便你，不需要考虑从 0 开始还要-1 的问题，第几个机器码就是它在 RAM 中的地址。问，为什么选择从 1 开始而不是像程序空间一样预

留一些？我的程序使用了很多 unsigned short ， 意味着你完全可以通过这种方式从 1 往回跳，即跳转到 65535 附近，还是类似于我上面提到的钟表比喻，当然，除非你知道自己在做什么，否则会有些危险。

RTI：由于我的运行器并没有实现一个函数判断运行器是否正常的功能，更不用提特权模式和用户模式了，因此这个指令被我魔改了，类似 ADD，是 RTI+（+的意思是会改变 Regis_cd，即 nzp 条件的寄存器），1000[DR][SR1][0/1]00[SR2]，由于咱们的要求“第二个操作数不允许是一个数字”，所以这个指令没有像 ADD 一样提供立即数 imm5 的运算，当 target 位是 0 的时候，Regis[dr]=Regis[sr1]*Regis[sr2]，更新条件位；当 target=1，Regis[dr]=Regis[sr1]%Regis[sr2]，更新条件位，特别注意，除数是 0 会结束虚拟机。

Reserved：1101：在 LC3 中，并没有这条指令，所以我改成了 RES+: 1101[DR][后面随意]，功能是将 regis[DR]取负，它和 NOT 区别在哪？区别在，由于我的机器码规则，位运算的取反‘~’0b0000000000001110（14）取反是 0b1111111111110000（-15），而 RES 则是-14，原因在与 NOT 只是单纯的将 0 变成 1，1 变成 0，取补码的运算的那个 “1”会改变结果，而 RES 乘了-1，计算机在处理的时候自动帮我们做了这些麻烦的事，做到了只改变符号但不改变结果。

TRAP：这是最魔改的指令，具体表现为，我将原来的 8 位改成了 3 位，后三位为 001：将 Regis[DR]（1111 0000 [DR]0 0_ _ _）的值拷贝到特殊的 TRAP 寄存器 Regis_trap 中；后三位为 010：将 Regis_trap 的值加上 48，作为 ASCII 的值输出相应字符；后三位为 011：输入一个字符，并将它与 48 的差存储在 Regis_trap 中；后三位为 100：更改计算机状态，结束运行器，要注意的是，如果没有这条指令结束，那么程序将继续读下去，后面的指令全是 0000……，因此只会进行无意义的 BR 直到 RAM 数组溢出；后三位为 101，反向读取，将 Regis_trap 中的值存储到 Regis[dr]中，可以认为是 001 的逆过程。为什么要算 48 的偏移量？原因是，48 对应 ‘0’ ，这是我在几版设计中认为最方便的一版，其他设计有：从 0——EOF‘\0’开始，从 128 开始，从 64 ‘@’开始，最后还是认为以数字 ‘0’为基准最符合逻辑最舒服，同时用更少的偏移量就能输出常用的数字和字母。至于除法，后三位 000，我自己觉得对整数除法是非常危险的事情，运算复杂起来会积累很多的误差，所以我把他放到了 TRAP 里作为一个边缘化的功能，Regis[DR]（1111 0000 [DR]00000）会与 Regis_trap 相除，结果保留在 Regis[dr]中，更新条件位。

1.3 我准备了一些测试样例，要拖到和可执行文件同一个文件夹下：从简单到复杂有：

0.fudu.txt 描述：检验 TRAP，您输入一个字母，程序也会输出一模一样的结果。

1.hello.txt 描述：检验 ADD、TRAP，输出 HELLO\n

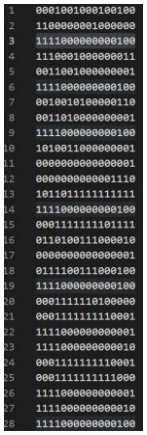
2.circle.txt 描述：检验 ADD、TRAP、BR，用上面的这些指令，我构建了一个简单的循环，输出 1-9，每个数字后有一个回车，你可以扩展我的这个文件，再增加一层循环就可以输出两位数啦。

3.calculate.txt 描述：检验 ADD，RTI，TRAP 里的 devide，注意，除数和取余为 0 会直接结束运行器，这个文件要在 debug 模式下才能看有没有在正常进行，有趣的是，这个文件里，我没写

end 结束码，这个程序是靠取余 0 报错结束的。

4.calculating.txt 描述：检验 ADD、AND、NOT、RES，注意，这个文件要在 debug 模式下才能看有没有在正常进行。

5.loadandstore.txt 描述：检验 ADD、LEA、LD、ST、STI、LDI、LDR 和 SDR，这部分值得拿来说一下。如右图所见，我在这个文件里插入了许多 1111000000000010，即 TRAP 的 end，在最后一个正常的结束之前，我要求这个文件输出“！\n”，那么，为了避开这一大堆不正常的 end，我分别使用了 JMP、LEA、LD 与 ST、LDI 与 STI、LDR 与 SDR，用这些寻址指令，选取其他的指令 load into register，再选取这些不正常 end 的地址，store into RAM，替换这些不正常的指令。在处理掉这些异常 end 后，程序成功输出 “！\n”，说明这些寻址指令没有问题。



我准备的测试文件效果都很一般，也很粗糙，这些都是我照着上面提到的 LC-3 指令集一个 0 一个 1 手搓的，会有一些地方不够精简、有些地方多余，欢迎你编写测试并向我反馈 bug。

四． 总结

最后，还是要感叹一句，16 位运行器能力真弱啊！一次加减的数字极小，寻址也要通过弯弯绕绕的方式实现。图灵奖得主 C.A.R. Hoare，这位老爷子说：“进行软件设计有两种方式。一种是让它尽量简单，让人看不出明显的不足。另一种是弄得尽量复杂，让人看不出明显的缺陷。”运行器也是如此。简单如 LC-3 指令集，简单的加减存取，也能把一群 0、1 乱码化零为整；复杂如 X86 指令集架构，据说有几乎上万条可以和 CPU 许多模块互动的指令，真是难以想象。

我们已经不能追溯，当年那群对着纸带打孔、对着手册敲 01010 的程序员要怎么编码、调试，但是，可以确定的是，无论 GUI 多么直观，无论操作系统和软件可以多么花里胡哨，计算机的架构依旧是冯·诺依曼架构，计算机的实现依旧是电路的通和断。在那个打孔的年代，实现真的很简单，实操真的很复杂；在这个面对 GUI 的年代，实操变得省心省力，但当我打出这些文字，或是你看见这些文字的时候，我们面对的计算机，依旧是不变的乃至永恒亘古的 0 和 1。当我用相比打孔进步许多的方式在处理 0 与 1 的时候，计算机也在做一样的事。

进步在技术，不变的依旧是繁与简的对抗与迭代。总有一天，新技术会产生，新事物会出现，人类会再面对一次更高级的纸带打孔，希望那时，你我都有迎头向上的勇气，重走那代人的路。“真正的英雄主义，是在认清生活的真相后，依旧热爱生活。”

五． 致谢

我写完了！运行器是，实验报告也是。在开工之前，我对自己几乎不抱能够完成的幻想，好在互联网有各种各样的学习资源，在参考了许多博客、文章和前人对数据处理的思路之后，跌跌撞撞完成了这个程序。每个指令能实现的功能真的很简单，代码思路也很单一，可至少他们组合起来能实现一些事情。

在这个过程中，很想感谢一下互联网，许多想法和思路就默默的在互联网上，直到某天我费尽力气才查到他们。在文末，我会贴上前文没机会贴的参考资料，~~也吐槽一句，网上补码的教程有些符号位居然在变、有些就不变，弯弯绕绕，这里采用的是符号位不变，最后只记住了转换公式{X（取补）=X（取反）+1}，指令集的那几个寻址也是绕蒙我了，写完有点眼熟，不就是换皮削弱版指针吗~~。

最要感谢的还是张四海老师和沈文杰、张义、徐铭鸽三位助教老师的付出，放在一年前的高中，我是不敢相信反感高中上机课的自己短短半年就能入门 C 语言的。事实上，在搞完这些之后，好多东西我还是懵懵懂懂，测试文件的补码运算都是自己在纸上算的，好多指令在我写的时候实在感觉换汤不换药，写完才隐隐感觉到它们应用的价值。

最后，感谢你读完这篇 6k 字的报告！Thank you for your time！

赵俊宇

PB24061271

附：正文未提及的参考资料

LC-3 学习记录（一）-CSDN 博客

汇编语言符号扩展指令及应用示例-CSDN 博客

为什么符号位的扩展，不改变数值的大小_负数补码符号位扩展后代表的值为什么不变呢-CSDN 博客

位运算全面总结，关于位运算看这篇就够了-CSDN 博客

【算法详解】位运算_位运算 csdn-CSDN 博客

课本—位运算部分

LC-3 Little Computer 3 Cheatsheet 计算机结构设计 机器语言 - 知乎

Introduction to Computing Systems | LC-3 Simulator