

LC-3 模拟器和 LC3 编辑器 Windows 版本的使用指南

LC-3 是一种硬件, 所以您可能想知道为什么我们需要一个模拟器。原因在于 LC-3 实际上并不存在(尽管它可能有一天会存在)。现在这只是一个计划——一个指令集结构 (ISA) 和一个微体系结构将实现 ISA。模拟器让我们看到程序执行期间发生在寄存器和内存中的一个“真实的”LC-3。

本指南是如何安排的

对于你们中那些喜欢潜水和立刻尝试一些东西的人来说, 第一部分会带你进入你的第一个程序, 用机器语言, 进入文本编辑器(称为 LC3Edit)。你还会发现关于编写汇编语言程序的信息, 但是你可能会跳过这部分, 直到在本学期的后面, 你学会了 LC-3 汇编语言。

第二部分简要介绍了仿真器的接口。

第三部分向您展示了如何使用模拟器看您刚刚编写的程序效果。

第四部分通过几个例子在仿真器中进行调试。

最后两部分是作为参考材料, LC3 编辑器, 模拟器本身。

换句话说:

第一章: 创建模拟器方案	2
第二章: 模拟器, 你在屏幕上看到的	6
第三章: 在模拟器上运行一个程序	9
第 4 章: 在模拟器中调试程序	12
第五章: LC3 编辑器参考	19
第六章: LC-3 模拟器参考, Windows 版本	22
第七章: LC-3 汇编程序快速参考	30

第一章：创建模拟器方案

这个例子也在教科书上,介绍计算机系统:从比特、盖茨到 C 及其以后!你会发现它在第六章,从 166 页开始。这里的主要区别是,我们将检查程序 x3003 行的错误给予纠正。一旦我们理解了“正确的方法”来做事情,我们将接触到一个调试示例。

问题陈述

我们的目标是取十个数字,存储在内存位置 x3100 到 x3109,并将它们添加在一起,把结果留在寄存器 1 中。

使用 LC3 编辑器

如果您使用的是 Windows 版本,将有另一个程序在同一文件夹下来作为模拟器,称为 LC3Edit.exe。通过双击它的图标来启动程序,您将看到一个简单的文本编辑器和一些特殊的添加。

用机器语言键入你的程序

你可以选择三种方式之一输入您的程序到 LC3 编辑器中:二进制、十六进制、或 LC-3 汇编语言。这就是我们用二进制编写的小程序:

```
0011000000000000
0101001001100000
0101100100100000
0001100100101010
1110010011111100
0110011010000000
0001010010100001
0001001001000011
0001100100111111
0000001111111011
1111000000100101
```

当你键入代码到 LC3 编辑器中,你可能会看到一个图表告诉你每条指令的格式,例如在教科书的封底(表后面)。所以你可能会更容易阅读自己的代码,如果你在每个指令的不同部分之间预留空间。此外,你可以在每一行代码的分号后面紧跟着注释,这将使它更简单可供你记得你正在做什么。在这种情况下你的二进制应该像这样:

```
0011 0000 0000 0000 ;程序的起始位置在 x3000
0101 001 001 1 00000 ;清 R1, 用来存放运行的总和
0101 100 100 1 00000 ;清 R4, 用来作为一个计数器
0001 100 100 1 01010 ; 用#10 装载 R4, 添加的次数
```

```

1110 010 011111100 ;装载数据的起始地址
0110 011 010 000000 ;装载下一个数被相加
0001 010 010 1 00001 ;递增指针
0001 001 001 0 00 011 ;添加下一个数到运行总和
0001 100 100 1 11111 ;递减计数器
0000 001 111111011 ;如果计数器不为零，继续做
1111 0000 00100101 ;停止

```

无论哪种方式对 LC3Edit 来说都是好的。无论如何，它忽略了空间。第二种方法就是更容易阅读。程序也可以看起来像这样的，如果你选择用十六进制键入(分号之后的注释仍是一个选项)：

```

3000
5260
5920
192A
E4FC
6680
14A1
1243
193F
03FB
F025

```



保存你的程序

点击这个按钮或者选择文件菜单下“保存”。当你把你的程序变成一个目标文件，你可能想要一个新文件夹保存，因为你将创建多个文件在同一个地方。如果你输入 0 和 1，把你的程序命名为 addnums. bin。如果你输入十六进制，把你的程序命名为 addnums. hex。

为你的程序创建. obj 文件

在模拟器运行你的程序之前，您需要将程序转换成 LC-3 模拟器可以理解的语言。模拟器不理解您刚输入到 LC3Edit 的十六进制或二进制 ASCII 表示。它只懂得真正的二进制，因此您需要将您的程序转换成实际的二进制文件，并将其保存在一个文件名称为 addnums. obj 中。如果你使用 LC3Edit，这些按钮中一个且只有一个将发生：



你将如何知道是哪一个？这取决于你是否用 1 和 0 键入您的程序（B 和一个箭头），用十六进制（X 和一个箭头），或用汇编语言（asm 和一个箭头）。

当你按下相应的按钮时，将创建一个新文件在同一文件夹，你保存了你的原始 addnums

程序。它会自动具有相同的名称,但其文件扩展名(除名称部分,在“.”之后)将是.obj。


如果你用0和1,或十六进制输入你的程序,只有一个新文件将会出现:addnums.obj(目标文件)。

如果你不知道LC-3汇编语言,现在你准备跳到第2章,并了解模拟器。一旦你学习汇编语言,在这个学期晚一点,您可以完成第1章,用更多可读的方式,了解进入程序的细节。

用LC-3汇编语言键入你的程序

所以这学期到一半,你已经引进了汇编语言。现在键入你的程序将是非常容易的。这就是添加了十个数字的程序可能看起来像,它仅利用了虚运算,标签,和注释。

```
.ORIG x3000
AND R1,R1,x0 ;清 R1, 用来存放运行总和
AND R4,R4,x0 ;清 R4, 用来作为一个计数器
ADD R4,R4,xA ; 用#10 装载 R4, 添加的次数
LEA R2,x0FC ; 装载数据的起始地址
LOOP LDR R3,R2,x0 ; 装载下一个数被相加
ADD R2,R2,x1 ; 递增指针
ADD R1,R1,R3 ; 添加下一个数到运行总和
ADD R4,R4,x-1 ; 递减计数器
BRP LOOP ; 如果计数器不为零, 继续做
HALT ; 停止
.END
```

你仍然需要改变你的程序为.obj文件,它现在被称为“汇编”程序。为此,单击按钮 。

因为你使用了更漂亮的汇编语言方法,你已经获得不只是一个,但少量的文件:

addnums.obj, 目标文件

addnums.bin, ASCII 码0和1编写的程序

addnums.hex, ASCII 码十六进制格式

addnums.sym, 在汇编程序第一次通过的时候创建的符号表格

addnums.lst, 程序的列表文件

The .bin and .hex 文件看起来和这一章节(不带注释)的前部分展现的一样。最后两个文件是值得看的。

addnums.sym

如果你在文本编辑器中打开它,这个文件看起来就像下面这样:

```
//Symbol Name Page Address
//-----
```

```
// LOOP                                3004
```

在您的程序你只有一个标签:LOOP。因此在符号表中,这就是唯一的入口。3004 是标签 LOOP 的地址,或内存位置。换句话说,当汇编程序看着一行接一行的通过时,它到达了这一行

```
LOOP LDR R3, R2, x0 ;装载下一个数被相加
```

看到标签“LOOP”,注意到位置计数器值 x3004,并放置单一入口到符号表。

第二步,每当汇编在声明中见到所指的标签时,

```
BRP LOOP
```

它用十六进制值 3004 取代了 LOOP 循环。如果你有更多的标签在你的程序中,他们会被列在符号名中,它们的位置将被列在页面地址。

addnums.lst

如果你使用任何文本编辑器打开文件列表,你会看到:

```
(0000) 3000 0011000000000000 ( 1)          .ORIG x3000
(3000) 5260 0101001001100000 ( 2)          AND    R1 R1 #0
(3001) 5920 0101100100100000 ( 3)          AND    R4 R4 #0
(3002) 192A 0001100100101010 ( 4)          ADD    R4 R4 #10
(3003) E4FC 1110010011111100 ( 5)          LEA    R2 x3100
(3004) 6680 0110011010000000 ( 6)    LOOP  LDR    R3 R2 #0
(3005) 14A1 0001010010100001 ( 7)          ADD    R2 R2 #1
(3006) 1243 0001001001000011 ( 8)          ADD    R1 R1 R3
(3007) 193F 0001100100111111 ( 9)          ADD    R4 R4 #-1
(3008) 03FB 0000001111111011 (10)          BRP    LOOP
(3009) F025 1111000000100101 (11)          TRAP   x25
```

让我们选择一行,把它分开。自第六行有一个标签,这是最有趣的一个。因此让我们来看这部分。

```
(3004) 6680 0110011010000000 ( 6) LOOP LDR    R3 R2 #0
```

(3004)

这是指令将被放置在内存中的地址,当您的程序将被加载到模拟器中。

6680

这是指令本身的十六进制值

0110011010000000

这是指令的二进制形式

(6)

这条指令是汇编语言程序中的第六行。因为标记着 (1) 的那一行刚好指定着开始位置，所以在模拟器中一旦这条指令被加载到内存中，它实际上是这个程序的第五行。但是我们现在计算的是汇编语言程序的行数，而不是内存的位置，所以这是第六行

LOOP

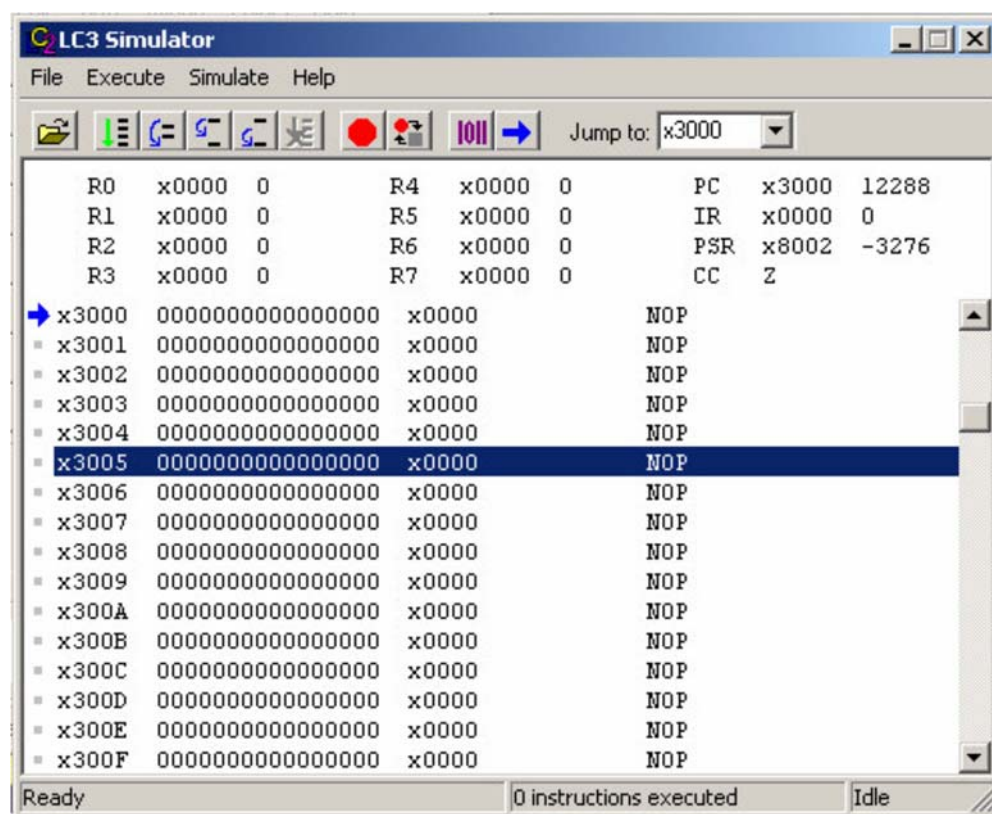
这是与这一行相关的标签

LDR R3 R2 #0

最后，这是这条指令的汇编语言版本。注意到这条指令后面的注释现在没有了，那些都只是对你自己有用的信息（或者对其他程序员）。模拟器不会关心它们。

第二章：模拟器，你在屏幕上看到的

当你启动 LC-3 模拟器的 windows 版本后，你会看到如下图所示：



第六章会更详细介绍这个界面的所有部分。如果你想知道所有的细节，去看第六章。如果你只是想知道足够的部分细节以继续看懂这一步接一步的例子，就继续看下去。

寄存器

注意在菜单项和工具条按钮下面的寄存器列表。

R0	x0000	0	R4	x0000	0	PC	x3000	12288
R1	x0000	0	R5	x0000	0	IR	x0000	0
R2	x0000	0	R6	x0000	0	PSR	x8002	-3276
R3	x0000	0	R7	x0000	0	CC	Z	

从左边开始，你会看到 R0 到 R3，然后跳到第四列，看到 R4 到 R7。这些就是 LC-3 指令用来存放源数据和结果数据的目的地 8 个寄存器。X0000 和 0 这两列是这些寄存器的内容，前一个是十六进制表示（通常这样的数字前面带有 x 代表后面跟着的是一个十六进制的数字），后面那个是 10 进制表示。当你启动模拟器的時候，这些暂时的寄存器总是包含零。

如果在程序的执行期间，R2 包含了十进制值 129，你会看到这样的：

R2 x0081 129

在模拟器顶部最后那三列展示了 LC-3 控制单元的 5 个重要寄存器的名字和内容。这些寄存器分别是 PC, IR, N, Z 和 P 状态码寄存器。

PC	x3000	12288
IR	x0000	0
PSR	x8002	-3276
CC	Z	

PC，也叫程序计数器，指向下一条将要执行的指令。当模拟器载入你的程序时，PC 寄存器会包含你的第一条指令的地址。地址默认值为 x3000。

IR，也叫指令寄存器，包含着当前指令的值。当你启动模拟器时，它保持为 0，因为此时还没有指令。

PSR，也叫程序状态寄存器，包含了当前处理器的状态，可能是用户状态或者是特许状态或者是状态码的值。

CC，也叫标识位，是由确定的一些指令来设置的（例如 ADD, AND, OR, LEA, LD, LDI, 和 LDR）。这些状态码由 3 个寄存器组成：N, Z 和 P。因为在任一时间，3 个寄存器中只有 1 个的值为 1，所以模拟器仅仅显示给我们的是当前值为 1 的寄存器的名字。（所以当你打开模拟器的時候，默认情况下 N=0, Z=1, P=0）。

内存

在寄存器的下面，你会看到一长串数字的紧凑表，类似下面这样的：

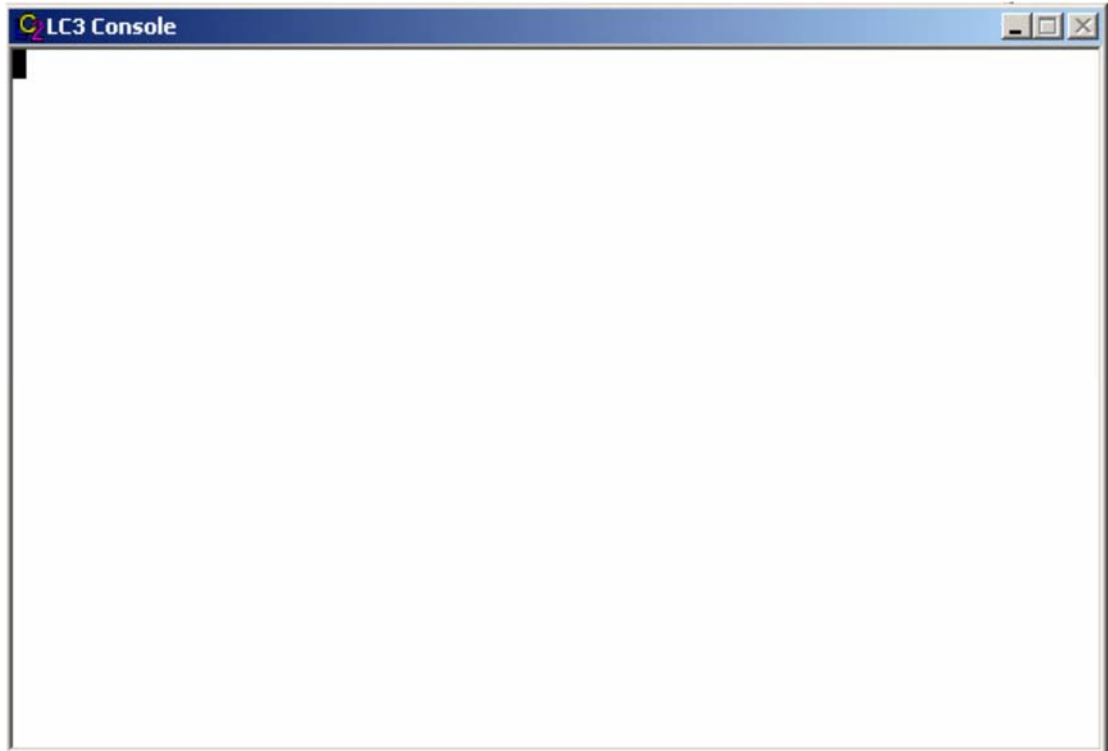
x3000	0000000000000000	x0000	NOP
x3001	0000000000000000	x0000	NOP
x3002	0000000000000000	x0000	NOP
x3003	0000000000000000	x0000	NOP
x3004	0000000000000000	x0000	NOP
x3005	0000000000000000	x0000	NOP
x3006	0000000000000000	x0000	NOP

使用右边的滚动条来在 LC-3 的内存中上下滚动。记住 LC-3 有 2^{16} 大小的地址空间，即总共 65536 个存储单元，这是一个很长的滚动列表。你可能会找丢了，如果你真的找丢了，去到界面顶部附近的”Jump to”方框，输入你想定位到的地址（记得在 16 进制表示的地址前要加小写 x）。


在这么一大长串存储单元的列表中，第一列告诉了你存储单元的地址。第二列以二进制的形式告诉了你一个存储单元中的内容，第三列也代表这内存位置中的内容，但是是以十六进制来表示的，十六进制表示有时候能够更容易表示。第四列是一个内存位置的内容的汇编语言解释。如果一个存储单元上有一条指令，这条汇编语言的解释就会有效。如果一个存储单元的内容是数据，那只需要完全忽略第四列就可以了。

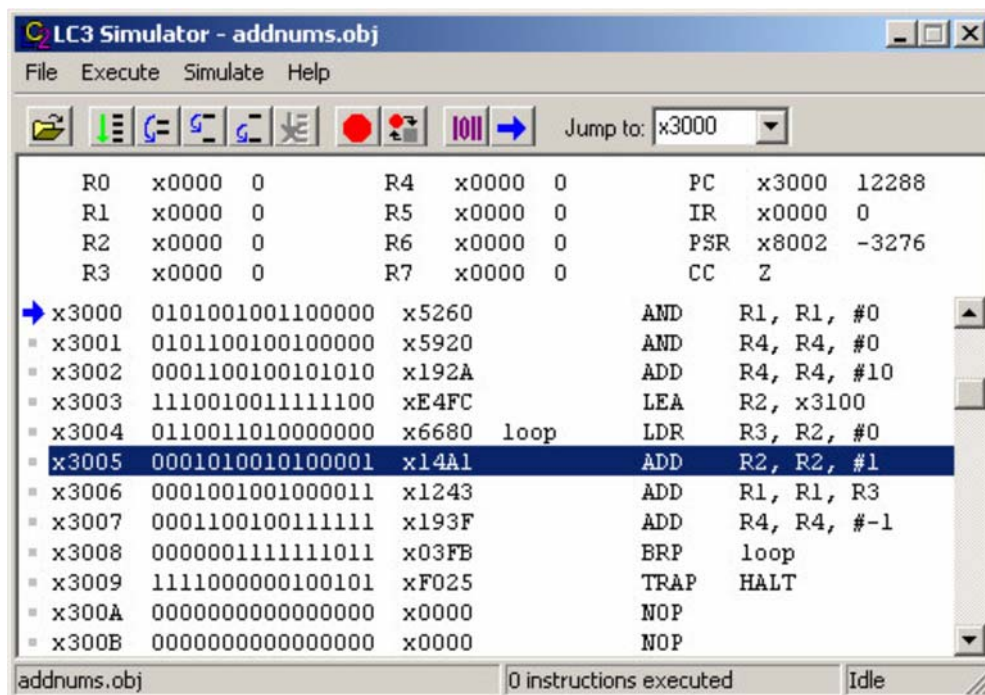
控制台窗口

当你启动模拟器后还会出现第二个窗口，这个窗口相当不显眼，有个模糊的标题“LC-3 Console”。这个窗口会给你输出一些信息比如“Halting the processor”。如果你在你的程序中使用输入输出程序，你会在这个窗口中看到你的输入和输出。



第三章：在模拟器上运行一个程序

现在你准备再模拟器上运行一个程序。打开模拟器，然后点击打开程序按钮 。浏览且选择 addnums.obj 文件。注意到在模拟器中，你只能选择打开.obj 这样的文件类型。当你的程序被装进来后，你会看到如下图所示：



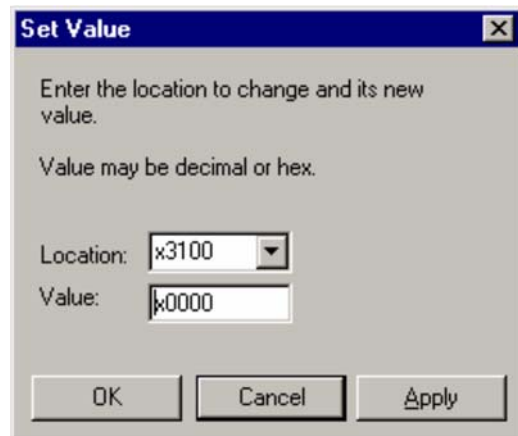
注意到你程序的第一行，不管你原来是使用什么格式写的，第一行都消失了。这第一行指定标记了程序应该被装载在 x3000 开始的内存位置。如果你向上滚动一行左右，你会看到，在 x3000 单元以前的仍然全是 0。因为现在还没有任何事发生（你还没开始运行程序或者单步运行程序），这些临时的寄存器（R0 到 R7）仍然保持全为 0，PC 寄存器指向了你程序的第一行（如上图蓝色标记的那行），IR 寄存器是空的。

加载数据（十个数字）到内存

有很多方法可以把你准备的数据加到 LC-3 仿真器的内存中。你想把他们放在 x3100 开始的位置。

第一个方法：点击工具条上“Jump to”框右边的输入框，输入十六进制数字 x3100, 当你按下回车后，在内存信息显示出来前，你会跳越 x100 个内存位置。所以 x3100 这个内存单元位置会第一个显示。

现在在 x100 处双击任何地方，你会看到如下这个弹出来的窗口：



在 Value 选项框中，填入 16 进制数 X3107 然后选择 OK 这时你的起始数据位置显示如下：


■ x3100 00110000100000111 x3107 ST R0, x3107

在上面表达式中，出现有二进制表达式，十六进制表达式，以及一些汇编语言表达式 (ST R0, x3107). 当然，这些都是数据，而不是指令，但是 LC-3 simulator 不知道这些。事实上，在 LC-3simulator 眼中所有内存位置的内容都是一样的。除非被告知作为指令运行，或者作为数据加载。。。。。。因为在离你输入数据很远的地方有一个停止指令，通常在此处是不被视为指令的，所以忽略掉那些汇编语言解释


你可以双击每一行并输入数据，如果你只想打开弹出窗口一次，即 Location 区域的值自动变化，在 value 区域输入下一个值，然后单击 apply 按钮，做完单击操作后，单击 OK


第二种方式： 返回到 LC3Edit. 程序，输入下面 16 进制代码：

```
3100          ;数据在内存地址 X3100 处开始
3107          ;从此处开始，添加的 10 个数字
2819
0110
0310
0110
1110
11B1
0019
0007
0004
```

单击 ，将代码保存为 data.hex

通常第一行是我们想要添加数据的起始地址。后面若干行是我们想要导入内存的实际数据。

因为我们使用 16 进制编程，因此单击 ，此时，在你保存 “.hex” 的位置将会产生一个名为 data.obj 的文件

现在返回模拟器，单击  导入项目，选择 data.obj。注意，你可以导入多个.obj 文件，他们可以同时存在于 LC-3 simulator 内存中。X3100 开始的内存显示如下：

▪ x3100	00110001000000111	x3107	ST	R0, x3008
▪ x3101	00101000000011001	x2819	LD	R4, x311B
▪ x3102	0000000100010000	x0110	NOP	
▪ x3103	0000001100010000	x0310	BRP	x3014
▪ x3104	0000000100010000	x0110	NOP	
▪ x3105	0001000100010000	x1110	ADD	R0, R4, R0
▪ x3106	0001000110110001	x11B1	ADD	R0, R6, #-15
▪ x3107	00000000000011001	x0019	NOP	
▪ x3108	00000000000000111	x0007	NOP	
▪ x3109	00000000000000100	x0004	NOP	
▪ x310A	00000000000000000	x0000	NOP	

现在，你的数据已经到位，程序运行已准备就绪

运行你的程序

单击，” Jump to” 选项框，选择 X3000 作为你想要到达的位置

下一步非常重要：双击地址 X3009 哪一行前面的小灰色方块：

```

▪ x3009  1111000000100101  xF025          TRAP  HALT
↑
double-click
here!

```

它将在这里设置一个断点，如果不这么做，在 R1 中你将看不到你的运行结果，这里我们设置了陷阱例程，使得 R1 关闭模拟器之前暂停。(在下一章节中将解释断点的更多详细情况。)

当你双击后，这一行将显示如下：

```

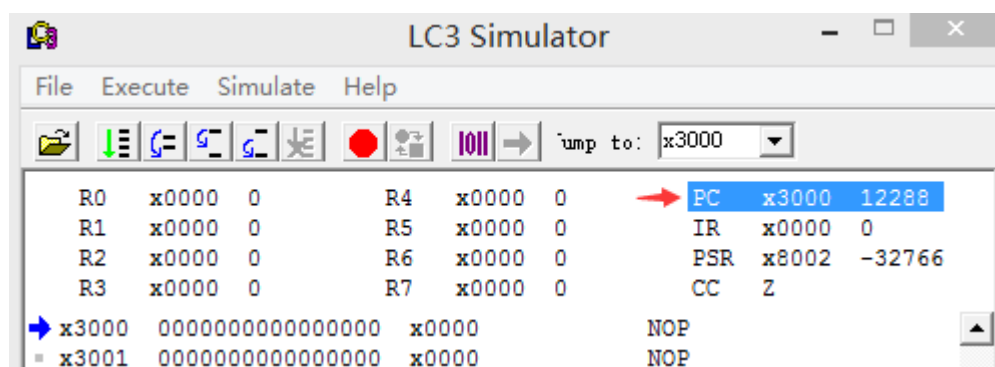
● x3009  1111000000100101  xF025          TRAP  HALT

```

那个红色的小点是一个停止标志。即我们的指令将在 X3009 那一行停止

现在，你可以开始运行程序了，确保 PC 值设置为 X3000. 因为那是第一个指令开始的地方。

如果没有，双击 PC 哪一行，将值设置为 X3000



现在，单击 ，运行程序


如果你已经添加了 10 个数字在你的程序的数据段中，你知道 x8235 是期望值。即当程序停止在断点处时，你将会看到的 R1 的值（在十进制中，结果是-32,459. 它是一个负值因为在 x8135 中第一个 bit 位是 1, 这是一个二进制的补码）。到达断点后将会弹出一个提示窗口，


在本例中，当 PC 值是 X3009 时会产生该事件

单步执行程序

现在你知道你的程序在运行，你也知道他们如何工作的。但这里并不会给你一个直观的感受，关于 LC3 在执行每一个指令期间发生了什么。通过一行一行的执行，观察模拟器发生了什么非常有趣。你会经常这样做来调整一些不良代码。让我们来实验吧


首先，你需要重将程序计数器重置为你的程序的第一个位置。即设置 PC 为 X3000. 可以双击

它，然后输入新的值。当然，你也可以使用快速方法，单击 x3000 哪一行，然后单击 ，这样就将 PC 指向了这个位置，现在你就可以单步执行程序了


单击  一步，第一次。，一些有趣的事情将会发生：

1. R1 被清 0，
2. 2，蓝色箭头，以及 PC 的值 都指向了 x3001 的位置，这也是下一个将运行的指令
3. IR 的值设置为了 x5260, 看看 x3000 处 16 进制的值，这里也是 x5260. IR 中存储了当前运行的指令。因为我们已经结束了第一个指令，并且还没有运行第二个指令，所以第一个指令依然是当前指令

单击  一步，第二次。观察 PC 和 IR 中的新的值，第二次指令 将 R4 清零

单击  一步，第三次。PC 和 IR 的值再一次更新，现在 R4 拥有了值 X0A, 即 10 进制数 10. 即我们需要重复循环 10 次，这个指令仅仅执行添加 x000A 到 x0000，并且将结果放置在 R4 中

继续单步执行，观察每一个指令执行后的结果，确保结果是期望看到的

如果你想结束单步，直接运行。你可以单击  按钮，这将直接执行程序到断点处
现在你知道一次就写出一个完美的程序，并成功运行是多么的美妙。但是，通常那是难以实现的。下一章将带领你在模拟器中调试一些程序

第 4 章：在模拟器中调试程序

现在，你已经熟练了理想状态下一次即完美运行，但是你不得不面对一个非常现实的问题，如果一个程序有一些问题，你必须尝试追踪这个问题并修复

Example1: 调试这个不适用乘法指令的乘法程序

这个例子来自课本，将在 129-130 讨论。这个程序支持两个正数相乘，在 R2 中保存结果

输入程序：

首先需要在 LC3Edit 中输入程序，如下显示：

0011 0010 0000 0000 ;程序起始地址：x3200


```

0101 010 010 1 00000 ; R2 复位
0001 010 010 0 00 100 ;R4中值与R2相加 结果放置与R2中
0001 101 101 1 11111 ;R5中值减去1
0000 011 111111101 ;如果结果>=0 转移至x3201
1111 0000 00100101 ;停止


```

研究程序发现：R4 与 R5 中的内容将会相乘，通过将 R4 中的值加上自身的 a 倍，这个 a 由 R5 中的值决定，举例来讲，如果 R4 中值为 7，R5 中值为 6，第一次 0+7 第二次 7+7 第三次 14+7 ... 第六次 35+7。在程序结束时 最后的值放置于 R2 中

将程序转换为.obj 格式

当你 在 LC3Edit 中输入完程序 并保存为如 multiply.bin 后，单击  转化为.obj 文件

将程序加载如模拟器

开启模拟器，然后单击  加载你的程序: multiply.obj, 现在模拟器的内存中部分内容显示如下：

➡ x3200	0101010010100000	x54A0	AND	R2, R2, #0
▪ x3201	0001010010000100	x1484	ADD	R2, R2, R4
▪ x3202	0001101101111111	x1B7F	ADD	R5, R5, #-1
▪ x3203	0000011111111101	x07FD	BRZP	x3201
▪ x3204	1111000000100101	xF025	TRAP	HALT
▪ x3205	0000000000000000	x0000	NOP	
▪ x3206	0000000000000000	x0000	NOP	

此时 PC 值为 x3200，蓝色箭头指向的那一行，即下一条即将执行的指令，因为程序尚未执行，该指令也为你的程序的第一条指令。

在 halt 指令处设置断点

断点有很多中用途，我们将会解除到其中一些，最好养成在 halt 除设置断点的习惯，如果不设置，程序就会运行到 halt 子程序然后结束，这时有可能会改变寄存器的值。因此，首先在行 x3204 处设置断点，双击改行最前面的灰色方框

```

▪ x3204 1111000000100101 xF025 TRAP HALT
↑
double-click
here!

```

这时，行 x3204 处如下：

```

● x3204 1111000000100101 xF025 TRAP HALT

```


红色的标识表示该行存在断点，如果一个程序运行时，当 PC 值为 x3204 时，模拟器就会暂停，等待进一步处理。

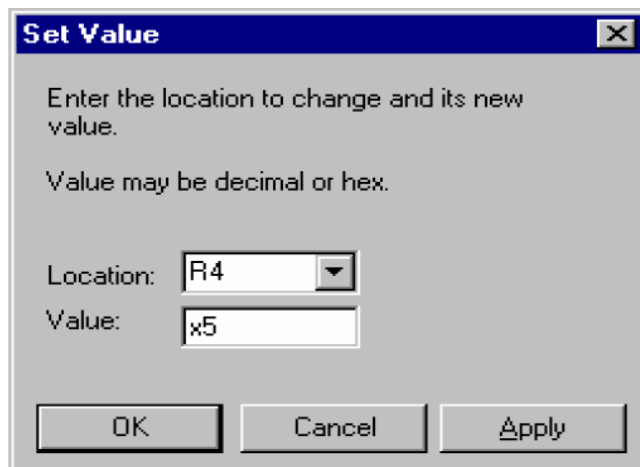
运行乘法程序

在第一次运行程序之前，需要为 R4、R5 设置相应的值，以便他们相乘。怎样选择合适的值以便利于测试，一般而言，0 和 1 不是好的选择，如果为 R5 设置很大的值，后面就会循环

好多次，因此选择两个较小的不同的值，如 5 和 3。




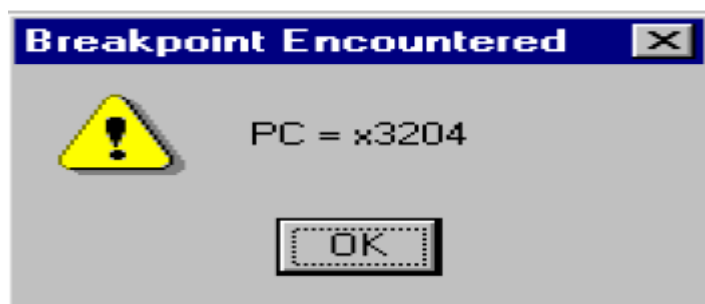
点击 ，弹出 Set Value 窗口，在 Location 字段中选择 R4，在 Value 处输入“x5”，如下：



点击 Apply，然后选择 R5，输入 x3，点击 OK，接下来运行程序。



点击 ，运行程序，稍后，会弹出如下窗口：



该窗口会弹出是因为在 halt 行处设置的断点，点击 OK 关闭窗口，查看 R2，应该包含最后的结果，十进制 $3 \times 5 = 15$ ，但是 R2 中包含十进制 20（十六进制 x14），程序存在问题，下面发现问题所在。

逐步调试乘法程序

方法之一是从头到尾逐行调试程序，因为程序存在循环，所以采用另一种方法，首先，让程序循环执行一次以确保每条指令正确执行。


双击 R5，然后在弹出的窗口中设置 R5 为 x3，然后点击 OK。



然后点击内存区域 x3200 处，接着点击 ，设置 PC 值为 x3200。

现在蓝色的箭头指向第一行，两个寄存器也已经初始化为目标值，接下来调试程序。




点击 ，Step Over，这时 PC 指向下一条指令 X3201，IR 中内容为第一条指令，

X54A0, R2 被清零。这些正是程序的期望目标值，下一步接着调试。




点击, PC 和 IR 值如约改变, 此时 R2 包含 5 (十进制和十六进制相同), 运行正确, 继续。



接着点击, R5 从 x3 变为 x2, R5 有两个作用, 既是乘数, 又是计数器 (告诉模拟器还要执行多少次循环, 因此每次循环结束, R5 自减), 继续。




点击, 触发分支指令的执行, 每次分支指令执行, 二者必选其一, 此时, 分支被执行, 因为状态码被 add 指令设置, add 之后结果为 x2, 正数, 因此状态码 P 为 1, 如果状态码中 Z 或 P 为 1, 分支执行, 因此分支被执行, 此时 PC 指向 X3201, 等待下一次循环。

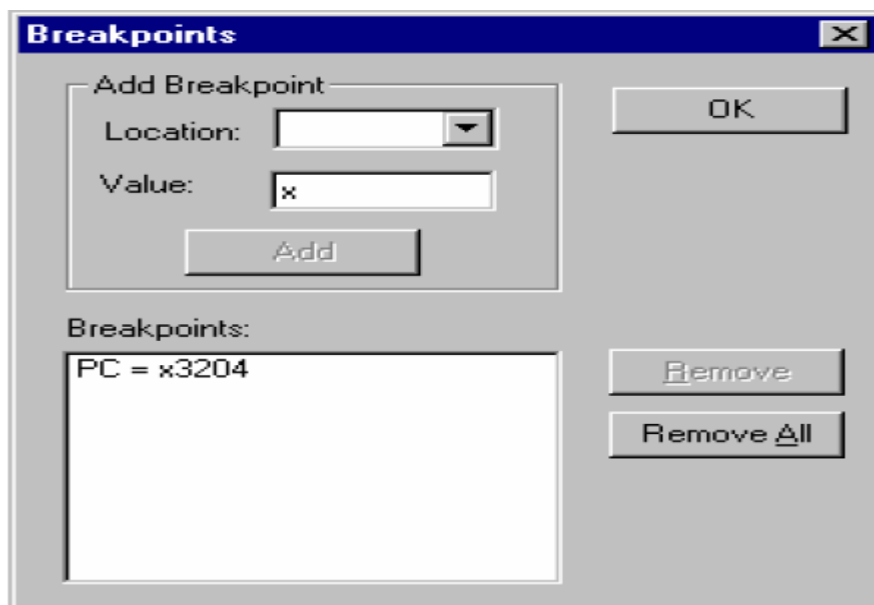
经过对程序的一次循环调试发现每条指令都没有问题, 这样问题有可能在循环设置的地方。

利用断点调试循环

发现一个循环是否被过多执行的好方法就是在分支指令处设置断点。这样在每次循环迭代的结尾处都会暂停, 此时有利于查看寄存器的状态。



这里尝试另一种设置断点的方法, 点击, 弹出如下窗口:



在 Location 处点击下拉箭头, 可以看到所有选项:


PC
X
PSR
IR
CC

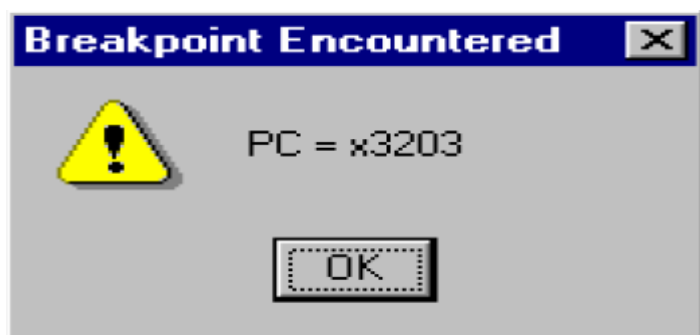
R0
R1
R2
R3
R4
R5
R6
R7

基本上，可以让模拟器在上述任意地方达到预设值时暂停运行，这样在 R0 的值为 X00FF 时，或者状态码 Z 为 0 时，或者内存区域 x4000 值为 x1234 时设置断点，程序就会相应暂停。

在本次调试中，设置 PC 为 x3203，然后点击 Add，这样断点列表中就有两条，两条都和 PC 有关，在 PC 为 x3203 时或 x3204 时，模拟器都会暂停，然后点击 ok 即可。




现在设置 PC 为 x3200，R5 为 x3，然后点击 ，运行程序，接着就会弹出如下窗口：




点击 OK，注意观察寄存器的值，蓝色箭头和 PC 均指向 x3203，R4 未变，R5 则变为 x2，R2 变为 x5。状态码 p 为 1，意味着继续执行程序时，分支会跳转。



点击 ，关闭弹出的窗口，如上次一样，观察寄存器，尤其是 R2 和 R5，目前已经循环两次，R2 内容为 x1，R2 为十进制 10，状态码 P=1，因此循环将继续执行一次。




点击 ，然后点击 OK，此时 R5 为 0，R2 为十进制 15，因为 $3 \times 5 = 15$ ，此时应该停止，但是状态码 Z=1，分支指令将继续执行，多做一次，这里错了问题。

通过修改分支指令使只有当 P=1 时，循环就会执行正确的次数，为验证其正确性，用 LC3Edit 修改分支指令如下：

0000 001 111111101 ;跳转到 location x3201 如果结果为正值

保存并转成 .obj 格式，装载程序到模拟器，如果不想修改源代码，可以在模拟器中直接修改，双击行 x3203，在 Set Value 窗口中将其值从 x0601 改为 x0201（此种方法仅对此次装载有效，下次装载时，该 bug 仍然存在，因此还要修改源代码）。



现在把 PC 值重设为 x3200，R5 设为 x3，双击行 x3202 取消断点，点击 ，关闭

断点弹窗后，可以看到十进制 15 出现在 R2 中，程序调试成功。

Example 2: 调试程序使其输入输入并求和

如果未学习汇编语言，可以等到学习后在做本实验。

在 LC3Edit 中输入程序

本程序的目的是让用户输入两个数（0 到 9），然后求和，然后打印（同样介于 0 和 9）
在 Console 窗口中，程序如下：


```
.ORIG x3000
TRAP x23 ;the trap instruction which is also known as "IN"
ADD R1,R0,x0 ;move the first integer to register 1
TRAP x23 ;another "IN"
ADD R2, R0, R1 ;两个整数相加
LEA R0, MSG ;载入字符串的地址
TRAP x22 ;输出字符串
ADD R0, R2, x0 ;sum 保存到 R0 中，并准备输出
TRAP x21 ;显示结果
HALT
MSG .STRINGZ "The sum of those two numbers is"
.END
```



将程序保存在 LC3Edit 中，并通过点击来编译这个程序。

在 simulator 中运行错误的程序

打开 simulator，载入程序。注意：halt 在 x3008 行，从 x3009 行开始，你会在每行都能看到一个 ASCII 码值。在 x3009 行，你会看到 x54，这个是表示字母 T 的 ASCII 码。在 x300A 行，你会看到 x68，代表“h”的 ASCII 码。整个字符串，“The sum of those two number is”存储在内存地址从 x3009 到 x3028 中，最后一个地址存储着空格。

双击 x3008 行前的小灰色方块来设置断点。现在点击运行你的程序。第一个指令是陷阱程序，提示你在 console 窗口输入字符，就像下面：

Input a character>

Simulator 在这个陷阱程序中会一直等待。（注意：“____instructions executed”信息会出现在 simulator 窗口的下方，并且____是快速增长的数字。知道你输入一个字符，它才会停止无期限的变化。）通过点击 console 窗口，你可以使它变成活动窗口，然后输入 0~9。试试输入“4”。

注意，当你输入“4”，R0 中会出现 x34。（寄存器 R0 保存的是你在 IN 陷阱程序中键盘的输入。如果你查看课本的附录 E 中的 ASCII 表，你会发现整数 4 在 ASCII 码中就是用 x34 表示的。

程序的第二个指令是将 x34 传递给 R1，然后你会被提醒再次输入一个字符。因为这是一个

非常非常简单的程序，你需要指定另一个整数使得它们相加的和最多为 9。所以再次点击 console 窗口，输入“3”。

一旦你输入了第二个数字，你会在 console 窗口中看到下面的信息：

The sum of those two numbers is g

你知道 3+4=7。错在哪里？

调试程序

关于为什么程序给了错误的结果，上面的一些段落给了一个大的暗示。记住，当你在 console 窗口输入“4”时，R0 中给出的值是 x34。当你输入的是“3”时，显示的是 x33。我们把这些值相加，结果是 x67。查看 ASCII 表，x67 代表的是“g”。所以你的输出也是有道理的，只是不是我们希望的。

你只需要在程序中围绕数据段添加几行代码，确保它正确。对数字 0~9 的 ASCII 码需要做一些窍门。“0”被表示为 x30，“1”被表示为 x31。这种模式一直延续到“9”被表示为 x39。所以，我们怎么将 x30 从整数的 ASCII 值提取出来，得到它原来的数值呢？

你只需要下面的数据段：

```
ASCII      .FILL x30          ;mask: 转换成 ASCII
MEGASCII   .FILL xFFD0        ; mask: -x30
```

需要添加 5 个指令：两个是载入两个 mask，一个是将 -x30 加到第一个数中，一个是对另一个数做同样的操作，最后一个是在输出前将结果加上 x30。你的程序现在是这样的（新添加的行显示为黑体）：

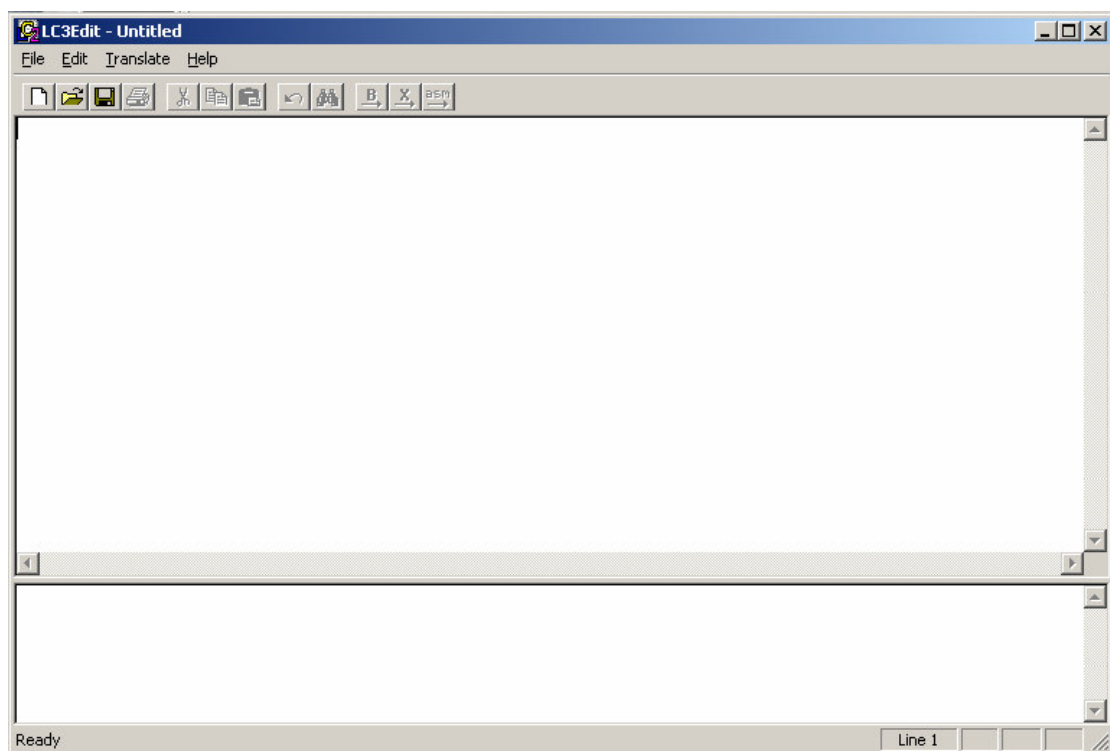
```
.ORIG x3000
LD R6, ASCII
LD R5, NEGASCII
TRAP x23 ;输入
ADD R1,R0,x0 ;将第一个整数传给 R0
ADD R1,R1,R5 ;将第一个 ASCII 数字转换成数值
TRAP x23 ;另一个驶入
ADD R0,R0,R5 ;将另一个 ASCII 数字转换成数值
ADD R2,R0,R1 ;将这两个整数相加
ADD R2,R2,R6 ;将和转换成 ASCII 表示
LEA R0,MESG ;载入字符串的地址
TRAP x22 ;输出字符串
ADD R0,R2,x0 ;结果传给 R0
TRAP x21 ;显示结果
HALT
ASCII .FILL x30 ;mask: 转换成 ASCII
NEGASCII .FILL xFFD0 ;mask: -x30
MESG .STRINGZ "The sum of those two numbers is "
.END
```

在 LC3Edit 中保存你的程序，并且编译。当你在 simulator 中尝试这一版本，你会得到你想要的结果。

关于这个程序的改进版本的一点建议就是：这个程序事实上可以更短一点。我们现在的程序就是所有的步骤都讲的很清楚而且很分离。如果你想让你的程序执行起来效率更高，试着重写，表达意义一样，但是可以减少 4 行。

第五章：LC3 编辑器参考

当你打开 LC3Edit，你会看到下面界面：



上窗口和下窗口

上窗口是我们可以输入机器语言(十进制或者十六进制)或者汇编语言程序的地方，有光标。例如：可以参见第三和第四章节。

下窗口是显示信息的，当你将程序转化成目标文件。如果你的程序没有任何问题，你将看到像这样的信息：

```
Assembling C:\WINNT\Profiles\kathy\Desktop\test.asm...
Starting Pass 1...
Pass 1 - 0 error(s)
Starting Pass 2...
Pass 2 - 0 error(s)
```

如果你不够幸运，你会看到这样的信息代替上面的信息：

**Assembling C:\WINNT\Profiles\kathy\Desktop\test.asm...
Starting Pass 1...
Line 8: Unrecognized opcode or syntax error at or before 'r0'
Pass 1 - 1 error[s]**

在这种情况下，你必须调试在你运行程序在 simulator 之前。

搞清楚错误信息代表了什么

选择 LC3Edit 的 Help 菜单，然后选择 “Contents...” 。点击显示 Index 的表，你会看到一系列的提议。双击 “Error Messages”，然后会出现一系列所有你可能遇到的错误的描述，当你转化或者编译成 .obi 文件的时候。

工具栏上的按键

菜单上的这些命令很容易使用，只要简单的点击命令栏上的按键。当你使用 LC3Edit 的时候，你可以将光标停留在任何一个按键上，然后一个 “工具提示” 会出现来告诉你这个按键的作用。



创建

你可以通过创建新的文件来清除上窗口。如果你没有保存你的当前文件，你会被提醒先保存。



打开

通过点击这个按键，你可以浏览电脑上的文本文件。默认能打开的文件扩展名为 .bin, .hex 和 .asm。但是如果你用了不同的文件扩展名来命名，你也可以选择 “all files”。



保存

如果你点击这个按键并且之前没有保存你的文件，你会得到一个弹出窗口叫 “Save Source Code As”。另外，这个按键会自动替换你文件的之前版本。



打印

这个会出现一个特有的弹出窗口来打印你的文件，以至于能让你在打印前设置你希望的属性。



剪切

这个按键会复制突出显示文本到剪切板，然后移动它。



复制

这个按键会复制突出显示文本到剪切板，但是不会移动。



粘贴

将会把剪贴板中的文本插入到光标位置



撤消

你可以撤消上次操作（当然要是合理的、、、、、没有撤消保存和打印!），再次单击，会重做你刚才撤消的操作。



搜索

这将会弹出一个窗口让你定义是从当前光标位置向上还是向下搜索，是否想要匹配例子，（匹配例子意思是如果你输入“add”，将会匹配“add”而不会匹配“Add”，“ADD”或者“aDd”）



2 进制转换

当你完成了编码 1 和 0 的程序，你想要（尝试）转换这个程序为一个.obj 文件，点击这个按钮。如果你的程序有错误，在界面窗口的下面你会看到关于错误的相关信息。



16 进制转换

当你完成了 16 进制的编码，你想要（尝试）转换这个程序为一个.obj 文件，点击这个按钮。如果你的程序有错误，在界面窗口的下面你会看到关于错误的相关信息。



汇编

如果你在 LC-3 中用汇编语言完成了编码，你想要（尝试）转换这个程序为一个.obj 文件，也成为汇编文件，点击这个按钮。如果你的程序有错误，在界面窗口的下面你会看到关于错误的相关信息。

菜单

大部分菜单上的项目会做上述所述的按钮一样的事情，然而，一些选项只能通过菜单完成。

文件菜单

“New” 做同以上按钮相同的事情

“Open” 做同以上按钮相同的事情

“Save” 做同以上按钮相同的事情

“Save as” 保存你的文件与当前名字不同的名字

“Print” 做同以上按钮相同的事情

“Print Setup...” 会弹出一个窗口，你可以改变你打印的性质，比如打印机，你想要多大的纸

“Exit” 关闭 LC3 编辑

编辑菜单

“Undo” 做同以上按钮相同的事情

“Cut” 做同以上按钮相同的事情

“Copy” 做同以上按钮相同的事情

“Paste” 做同以上按钮相同的事情

“Find...” 做同以上按钮相同的事情

“Find Next” 搜索在 “Find.” 下你最近定义的搜索文字

“Replace...” 会弹出一个窗口，你可以搜索某些词，然后用什么词去替代它，你可以只更换一个实例，也可以更换你文件里的所有实例。

转换菜单

“Convert Base 2” 做同以上按钮相同的事情

“Convert Base 16” 做同以上按钮相同的事情

“Assemble” 做同以上按钮相同的事情

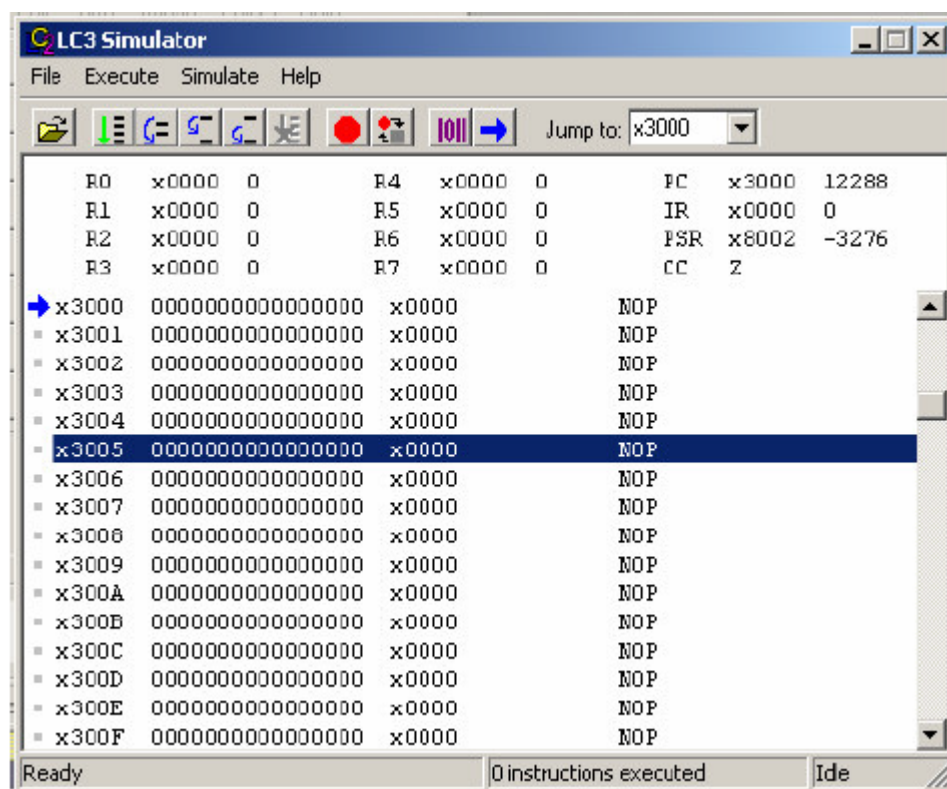
帮助菜单

“Contents...” 会弹出一个帮助窗口的概述和大量的主题。如果你需要主题更多详细的说明，你可以点击[这里](#)查看。

“About LC3Edit...” 可以给你 LC3Edit 的作者和版本的相关信息。

第六章：LC-3 模拟器参考，Windows 版本

这里你所看到得是当你启动 windows 版本的模拟器时：



这个界面有几个部分，现在一个一个的来看这些部分。

寄存器

接近界面的顶端，你会看到 LC-3 中最重要的寄存器，如下内容。

R0	x0000	0	R4	x0000	0	PC	x3000	12288
R1	x0000	0	R5	x0000	0	IR	x0000	0
R2	x0000	0	R6	x0000	0	PSR	x8002	-3276
R3	x0000	0	R7	x0000	0	CC	Z	

R0 到 R7 是 8 个寄存器，LC-3 指令可以从它们那里获取资源或者存储资源到它们那里，x0000 和 0 的列是寄存器里的内容，第一个是 16 进制，第二个是 10 进制。
最后的四列显示 PC、IR 和 PSR 的名字和内容，N、Z、P 是寄存器的状态码。

正如你所知，PC 或程序计数器指向的是当前要完成的指令的下一条将被执行的指令，当你启动模拟器，PC 的值总是 x3000（十进制是 12288，但是我们从不用地址为十进制），因为一些原因，汇编语言的教授对地址 x3000 有着特殊的爱好，他们喜欢从这里开始程序。某一天可能会有人发现为什么。

IR，指令寄存器，保存当前正在被执行的指令。当启动模拟器的时候，它的值总是零，因为在你运行程序之前这里面没有“当前指令”存储。如果有一种方式去看在 6 个步骤做成的一个指令循环中，在 LC-3 发生了什么，那么你会发现 IR 的值在 6 个步骤中都是相同的。模拟器不会让我们看到指令的“内部”。所以如果你正在一步一步的执行完你的程序，IR 实际上包含的是刚刚执行完的指令。它将会一直考虑“当前的”直到下一个指令开始，执行的第一个步骤-取指令-会赋给 IR 一个新的值。

PSR，处理器状态寄存器，包含关于处理器当前状态的信息。特别是，处理器处于用户或特

权模式都是通过当前状态编码设置。

CC，状态码，包含三个寄存器：N、Z、P，你不会看到三个都列出来，因为模拟器的作者很聪明，他意识到任何时候，三个寄存器只会有一个寄存器的值为 1，其他的两个寄存器一定为 0。所以，对于 CC，你会知道看 N、Z、P 中一个字母。当你启动模拟器时，Z 寄存器值设为 1，N 和 P 设为 0，所以你会看到 Z 出现在状态码中。当你运行任何指令都会改变状态码 (ADD, AND, OR, LEA, LD, LDI, or LDR)。

内存

下面的寄存器，你会看到一个长长的、数字密集列表，用滚动条上下滚动调整 LC-3 的内存。记住，LC-3 有 216 个地址空间，或者是 65536 个存储地址。有很长的列表去滚动，你可能会迷失，（这就是为什么我们在顶部有一个“Jump to”指令，你可能会用到）。

x3000	0000000000000000	x0000	NOP
x3001	0000000000000000	x0000	NOP
x3002	0000000000000000	x0000	NOP
x3003	0000000000000000	x0000	NOP
x3004	0000000000000000	x0000	NOP
x3005	0000000000000000	x0000	NOP
x3006	0000000000000000	x0000	NOP

当你启动模拟器时，大部分的内存是“空”，意思是说大部分的地址值为 0。你会注意到每个的地址后面是 16 个 0 或 1。这是地址的 16 位 2 进制值，2 进制的表示后面你会看到 16 进制的表示，因为这很方面的阅读。

内存长表的最后一行包含了文字或内存，这行是模拟器的解释翻译成 LC-3 的汇编语言。当你加载一个程序时，这些翻译会非常的有帮助，因为你会很快的查看你的程序，知道发生了什么。然而有时候，这些汇编语言解释毫无意义。记住，电脑和模拟器一样很笨，他们不会明白你的想法。所以程序的数据版块会翻译为最后一行某种形式的汇编语言。冯诺依曼模型重要的一方面是指令和数都会存储在计算机内存。我们唯一告诉他们的方式就是我们怎么使用它们，如果程序计数器在一个特殊的位置加载数据，那么数据会被解释为一条指令。如果不，那么不会的，所以当它把程序的数据给了很多无意义的解释，忽视最后一行的信息。你可能会注意到，如果你有时候游览内存，如果你没有放任何东西在内存，内存的地址也不会设为全 0，内存的特定部分用来保存操作系统需要的指令和数据，比如，地址 x20 到 xFF 是保存陷阱进程的地址。这里有这部分的一小部分：

x0020	0000010000000000	x0400	BRZ	x0000
x0021	0000010000110000	x0430	BRZ	x0030
x0022	0000010001010000	x0450	BRZ	x0050
x0023	0000010010100000	x04A0	BRZ	x00A0
x0024	0000010011100000	x04E0	BRZ	x00E0
x0025	1111110101110000	xFD70	TRAP	x70
x0026	1111110100000000	xFD00	TRAP	x00
x0027	1111110100000000	xFD00	TRAP	x00
x0028	1111110100000000	xFD00	TRAP	x00

内存的其他地方保存可以引发进程的指令,不要替换这些地方的值,或当你运行你的程序时,奇怪的行为会不定期的发生。可能这些信息会给你一个提示为什么教授喜欢内存地址 x3000 作为程序开始的地址。这离任何操作系统内存区都很远,所以你不会想要意外替换重要的指令和数据。

蓝色箭头

当你启动模拟器时,在执行程序期间,你会看到一个蓝色箭头指向一个内存单元。

➡ x3000 0000000000000000 x0000 NOP

这是一个提示,蓝色箭头现在正指向地址 x3000.

PC x3000 12288

箭头让你知道在你的程序里下一个指令是什么。由于程序计数器当前值是告诉我们在地址 x3000 的指令是我们将要执行的下一个指令,箭头所指的就是 x3000 在内存的位置。

那个'Jump to:' 选项

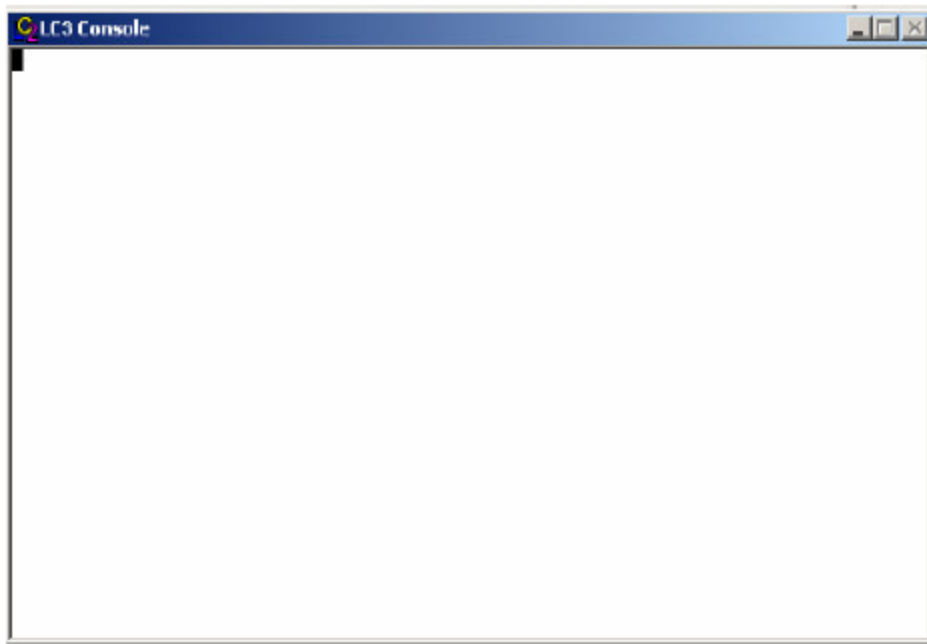
在模拟器顶部右侧角落,你将会看到一个有用的接口。

Jump to: x3000 ▼

由于内存是这么大(2^{16} 个地址),下拉滚动条不会列出全部内存地址让你选择,你将会发现你会经常用到这个技巧。点击 16 进制数(这里是 x3000),输入一个新的位置,当你按下 Enter 键的时候,你的内存将会被导向输入地址,如果你按下下拉滚动条将会记录你最近输入的内存坐标。

一个提醒:模拟器会识别十进制和十六进制数,你经常会使用十六进制数来引用一个内存位置,所以别忘了在你输入的数字前面加上" x",否则将会被模拟器当成十进制数处理,此时你的引用的错误的内存位置。

The Console Window



在我们详细地浏览过模拟器窗口前，我们应该注意到另一个窗口和模拟器运行时候一起弹出，它很不显眼，有着模糊的标题” LC3 Console”，这个窗口会给你诸如 “Halting the processor.” 等信息，如果你在你的程序中使用输入输出，你将会在这个窗口看到你的输出和执行你的输入。

工具栏上的按钮

让我们一起来看看每个按钮的功能（如果你把鼠标放置在按钮上方，会弹出一个工具提示，告诉你按钮名字防止你不知道或忘记）



Load Program

这个按钮告诉你浏览和打开一个文件，文件类型只能是.obj 文件，这些文件可以通 LC3Edit 生成（详情参见 LC3Edit 部分）



Run Program

这个按钮会执行你的程序，直到遇到两个指令会停止：HALT（停止机器时钟），breakpoint 你设置的断点



Step Over

这个按钮会执行一条指令，然后停止，等待执行下一个语句。如果执行的语句是像 JSR 或 JSRR 或 TRAP 等指令，模拟器将会一次性执行全部子程序包含的全部指令，然后等待主程序下一条指令。” Step Over” 直接执行完全部子程序，而不会等待你在子程序中慢慢单步执行。



Step Into

这个按钮会执行一条指令，和” Step Over”不同的是如果执行的语句是像 JSR 或 JSRR 或 TRAP 等指令，你将单步执行子程序的每一条指令，而不是全部执行。



Step Out

如果你的程序正执行到子程序中的某条指令(由于 JSR 或 JSRR 或 TRAP 指令引起进去子程序的)，你希望一次性执行完全部子程序，返回调用子程序的程序中的下一条指令，那么久使用这个按钮，你将会直接返回到调用程序中 JSR 或 JSRR 或 TRAP 的下一条指令。更切确的说你将会返回到当子程序被调用时候放入寄存器 R7 内的那个地址。



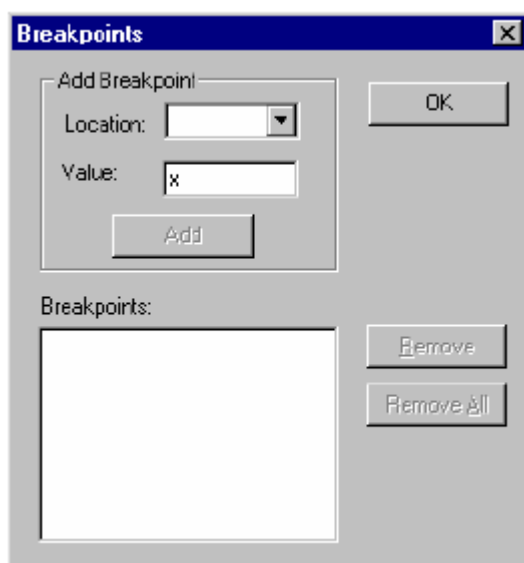
Stop Execution

如果你的程序陷入了无限循环，不用恐慌，这个按钮会停止全部执行。



Breakpoints

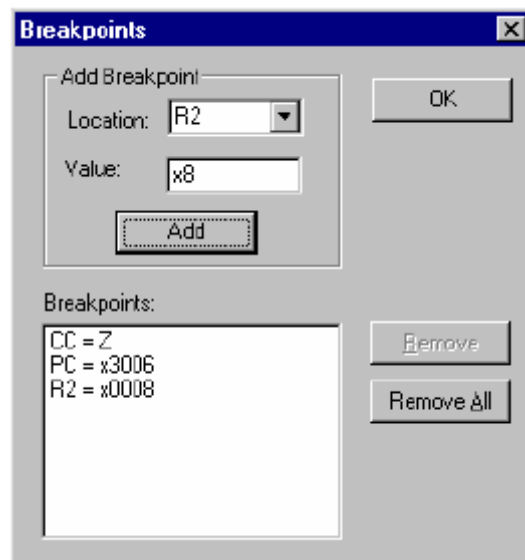
这个按钮带来了下面的对话框，这里你可以设置断点，基于 PC, PSR, IR, CC, R0-R7 或者内存位置的值。断点和它们的使用将在 debugging 章节讨论。



点击滚动下拉列表，你将会看到那些寄存器的列表，和一个神秘选项” x. ”，这个选项是用于指定某个特定内存地址，它使用十六进制。例如你想要定位选项去读” x3008”，如果你想要模拟器当到 x3008 位置的内存存放你在 value 框指定的特定值停下来等待你的输入。

Value 框内的值可以使十六进制数(以 x 开头)或十进制数(不以 x 开头)，它们都可以用 16bit 的二进制数表示，这边有一个特例，如果你选择 Location 是 CC，你在 Value 框内值只能是’ N’ ， ’ Z’ ， ’ P’ 中的一个。

添加一个断点，关闭弹出窗口，点击 OK。添加多个断点，设置 Location 和 Value 后按 Add。例如，假设你添加了多个断点：一个是暂停模拟器当 CC 的值为 Z，一个是暂停模拟器当指令执行到 x3006 内存地址时候，一个是暂停模拟器当 R2 里面值为 x8，你的对话框应该像下面一样。



现在你可以选择框框中列出的列表中任何断点，可以选择 delete 删除，Remove All 即可删除全部断点。



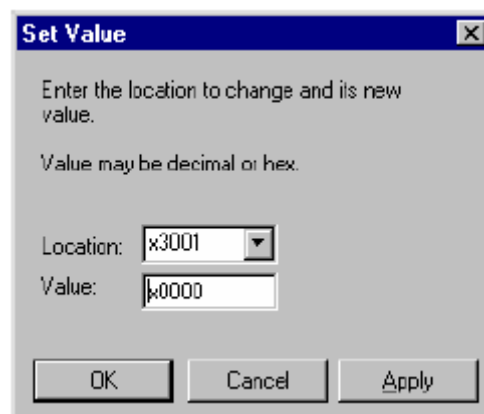
Toggle Breakpoint

如果你点击你程序的任何行，然后点击这个按钮，它会在该行上创建一个断点，或删除该行上的断点。



Set Value

这个按钮带来了一个弹出窗口如下，它可以让你改变任何寄存器或内存的值。



Location 滚动下拉列表列出了寄存器，和神秘的”x.”，就是上面说的十六进制的内存位

置地址, Value 可以使十六进制数或十进制数, 记住 x 开头 16 进制数, 非 x 开头 10 进制数。

当你点击 Apply 按钮, 相应位置的值将变成你指定的值。但是窗口还不会关掉, 方便你指定更多内存或寄存器的值。当你设置完后点击 OK, 退出对话框。



Set PC to selected location

这个按钮是很有用的, 假如你在调试整个程序, 现在你要指定特定的某个部分开始。在 jump to 中指定好地址后, 点击这个按钮, 就会指向相应内存位置, PC 会变成指定的相应值。

The menus

大部分菜单项目和前面介绍的按钮做一样的事情, 一些选项只能通过菜单实现。

File menu

“Load Program...” 和前面讲的按钮功能一样。

“Reinitialize Machine”, 重置全部寄存器和内存位置为它们的默认值。如果你的环境乱七八糟, 想从一个干净的环境开始, 这是一个好的选项。

“Reload Program”, 装载你现在正在处理的程序, 重置 PC 到开始位置。如果你已经编辑好你的程序, 产生了新的 .obj 文件, 使用这个选项来装载新的版本。如果你现在没有正在处理的程序, 它是不可用状态。

“Randomize Machine” 是我个人最喜欢的一个选项。它吧全部寄存器和全部内存位置 (没有存有重要值, 像 TRAP 子程序) 为随机值。当你认为你已经算出全部 bug 时候, 在装载你的程序前使用这个命令是个很好的主意。这样一来, 你假定一些内存位置以 0 开始的, 新的 bug 将会呈现出自己。

“Clear Console” 擦除 console window 上的全部文本。

Execute menu

“Run”, “Stop”, “Step Over,” “Step Into,” and “Step Out” 和上面的按钮功能一样。

Simulate menu

“Set Value”, “Set PC to selection location”, “Breakpoints” and “Toggle Breakpoint” 和上面按钮功能一样。

“Display Follows PC” 它是一个触发器, 可以检查标志。如果选中它, 当前运行指令的内存位置将会被显示。如果没被选中, 你可以滚动到特定的内存位置, 不管当前指令在哪里执行。

Help menu

“Contents...” 带来一个索引主题, 选择一个解释对于主题。如果你发现指导复杂难懂, help menu 是一个发现更多解释的好地方。

“About Simulate”显示关于版权信息和作者的名字和 Email 地址。

第七章：LC-3 汇编程序快速参考

标识

标识是对大小写敏感的，它必须以字母或下划线开头，后面接不多于 9 个字母数字字符。它们不可以为汇编语言的保留单词，例如一个操作码或伪操作，标识可以以冒号结束也可以不以冒号结束。如果以冒号结束，该标识在后面的指令中应该被引用到。每个内存空间位置只能有一个标识。

指令语法

指令必须包括操作码和操作数，操作数可以以逗号划分也可以不以逗号划分。虽然汇编程序标识对大小写敏感。

伪操作

在每个汇编程序中伪操作只有两个 `.org` 和 `.end`，标识可以和 `.Fill` 一起使用，用来填充标识位置所在内存的值。伪操作可以再汇编程序 `.org` 和 `.end` 之间的任何位置。

常量表达

汇编程序接受以 `#`（十进制）或 `x`（十六进制）开头的数字，对于十六进制不可以用 `'-'`（表示负数）为先导符号，对于十进制则可以。