**ICS第三次习题课**

# **HW5**

#### T1 What is the purpose of the .END pseudo-op? How does it differ from the HALT instruction?

**.END伪指令的目的是什么？它与HALT指令有何不同？**

**.END 不是⼀条指令，并不会显式地存放在内存中，只是在汇编语言中标识在哪里停止汇编的⼀个标识符，而相比起来， HALT 是⼀条指令，它会存放在内存中，当 LC-3 执行到这条命令时，会停机.**

#### T2 What are the definitions of a queue?

**队列的定义是什么？**

**队列的定义特征主要包括以下几点：**

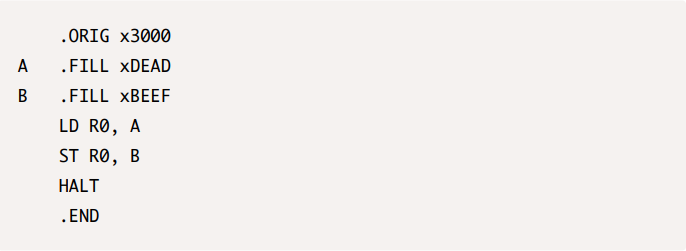
**1. 先进先出 (FIFO) 原则：队列中的元素从队尾 (rear) 进⼊，从队头 (front) 出来，最先加⼊队列的元素将是最先被移除的。**

**2. 线性结构：队列是⼀种线性结构，元素之间是⼀对⼀的关系，每个元素有且仅有⼀个前驱和⼀个后继（除了队头和队尾的特殊元素）。**

**3. 操作限定：队列的操作主要限定在队头和队尾进行，包括入队 (enqueue) 操作在队尾添加元素，出队 (dequeue) 操作在队头移除元素。**

#### T3 The following program has an error in it. What is the error? How would you fix it?

**下面的程序中有一个错误。错误是什么？你会怎么修正？**



**程序中 A 和 B 处存放的值会被当做命令执⾏，而 A 处的 OPCODE 为 1101， 是保留的指令，因此会出错；要修正，只需将 A 和 B 移到 HALT 之后即可.**

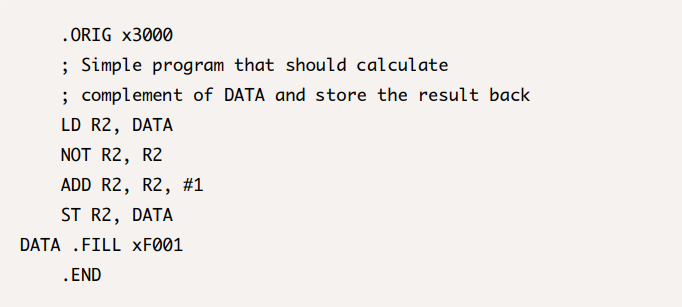
#### T4 Suppose you write two separate assembly language modules that you expect to be combined by the linker. Each module uses the label AGAIN , and neither module contains the pseudo-op .EXTERNAL AGAIN . Is there a problem using the label AGAIN in both modules? Why or why not?

**假设您编写了两个单独的汇编语言模块，希望通过链接器将它们组合在一起。每个模块都使用标签AGAIN，并且两个模块都不包含伪操作.EXTERNAL AGAIN。在两个模块中再次使用标签是否存在问题？为什么？**

**没有问题，因为每个模块会有⾃⼰的符号表，没有 .EXTERNAL 声明时，只会在自己模块的符号表内查找 AGAIN 的地址，不会出现冲突.**

#### T5 Your friend has just written a simple program intended to calculate complements, which is as follows:

计算”补码”



#### However, it does not seem to be reliable for some reason...

#### Questions:

#### 1. What's the 2's complement of xF001 in hex?

x0FFF.

#### 2. Will the program store the complement to DATA ?

Yes.

#### 3. What will happen afterwards? Why?

Infinite loop. x0FFF interpreted as "BRnzp #-1".

#### Open questions (Answer if you like, but it WILL NOT be graded): What's the root cause of this phenomenon? How can we prevent this from happening?

Root cause:LC3将数据和指令存储在一起，因此数据可能被解释为指令。

Solution: 将数据和指令分别存储。（指令内存+数据内存）

#### T6 What's the difference between pseudo-ops .FILL , .BLKW and .STRINGZ in LC3?

.FILL ⽤于存放⼀个 16 位的数据在它的地址。例如 .FILL x1234 将 x1234 存放到内存中。

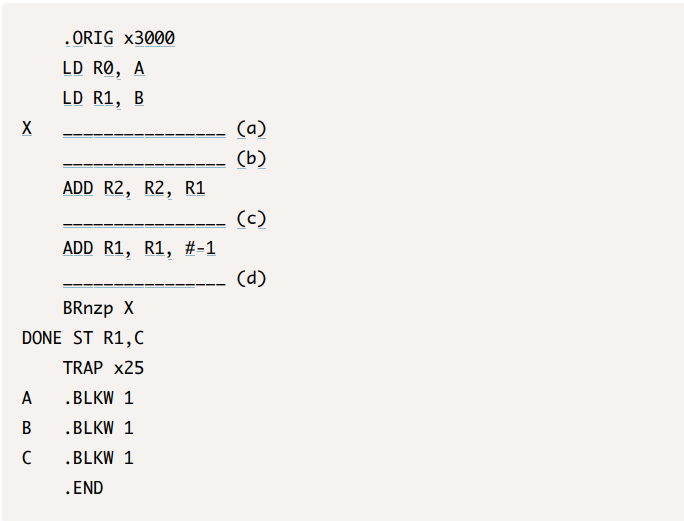
.BLKW ⽤于预留多个字。例如 .BLKW #3 将在这个位置留下 3 个字，以便后续存放数据，被预留的字在内存中默认为 x0000。

.STRINGZ ⽤于在连续的内存中存放字符串。例如 .STRINGZ "Hello" 将字 符串中的字符 ASCII 码存放到连续内存中，最后以 x0000 结尾。



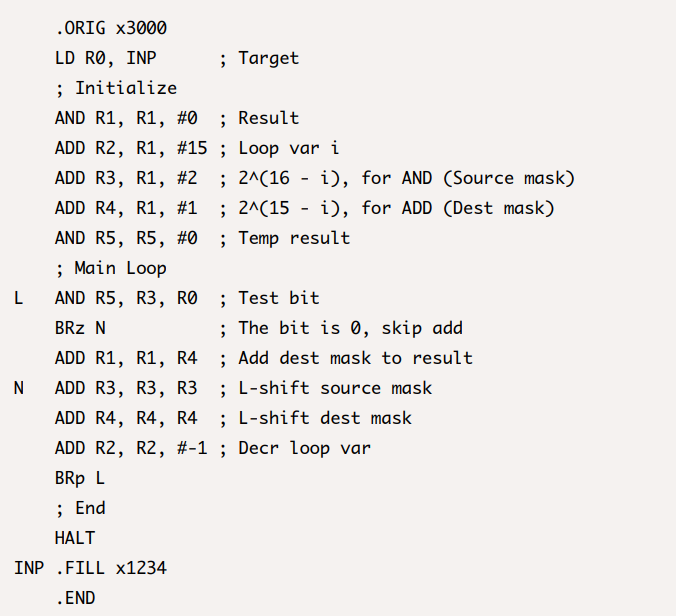
#### T7 It is often useful to find the midpoint between two values. For this problem, assume A and B are both even numbers, and A is less than B. For example, if A = 2 and B = 8, the midpoint is 5. The following program finds the midpoint of two even numbers A and B by continually incrementing the smaller number and decrementing the larger number. You can assume that A and B have been loaded with values before this program starts execution.

找到两个值之间的中点通常是有用的。对于这个问题，假设A和B都是偶数，并且A小于B。例如，如果A=2，B=8，则中点为5。以下程序通过不断递增较小的数字和递减较大的数字来找到两个偶数A和B的中点。您可以假设在这个程序开始执行之前，A和B已经加载了值。



#### T8 We all know that we can achieve left-shift by adding the number to itself. For example, ADD R0, R0, R0 will left-shift R0 by 1 bit. However, right-shift is not that easy. Complete the following LC3 program so that it will right-shift R0 by 1 bit.

我们都知道，我们可以通过将数字本身相加来实现左移。例如，ADD R0、R0和R0将使R0左移1位。然而，向右转变并不是那么容易。完成以下LC3程序，使R0右移1位。

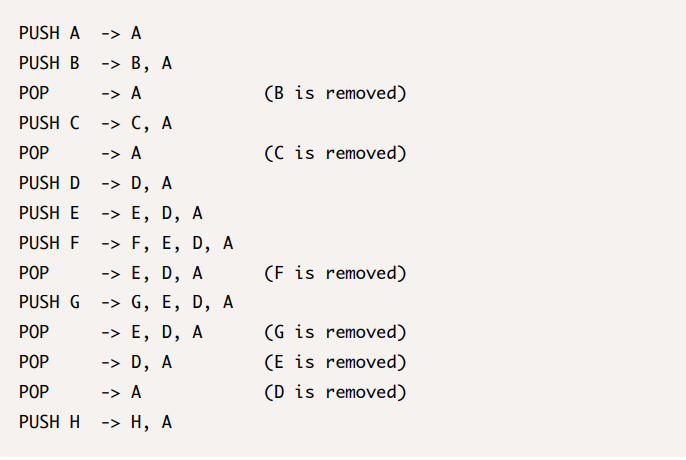


从第二位开始检测是否为1，是则加上此位降一位的值

#### T9 The following operations are performed on a stack:



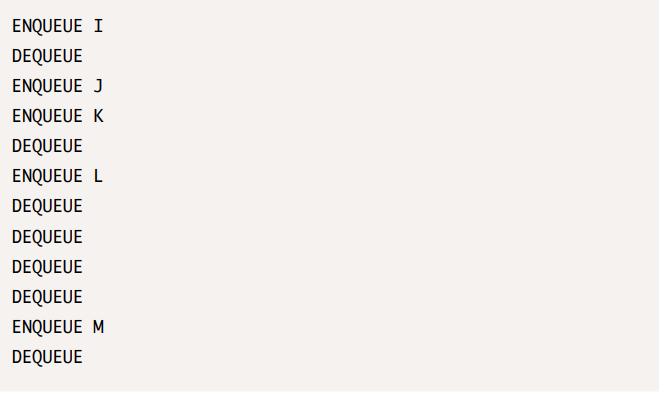
#### 1.What dose the stack contain after the PUSH H ?



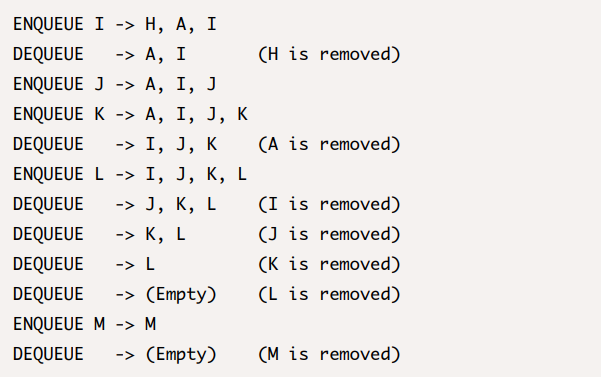
#### 2. At which point does the stack contain the most element?

最多的时候是执⾏ PUSH F 后或者 PUSH G 后，总共有 4 个元素

#### Without removing the element left in the stack from the previous operations, we change this stack to a queue (the front of queue is the top of stack), and perform:

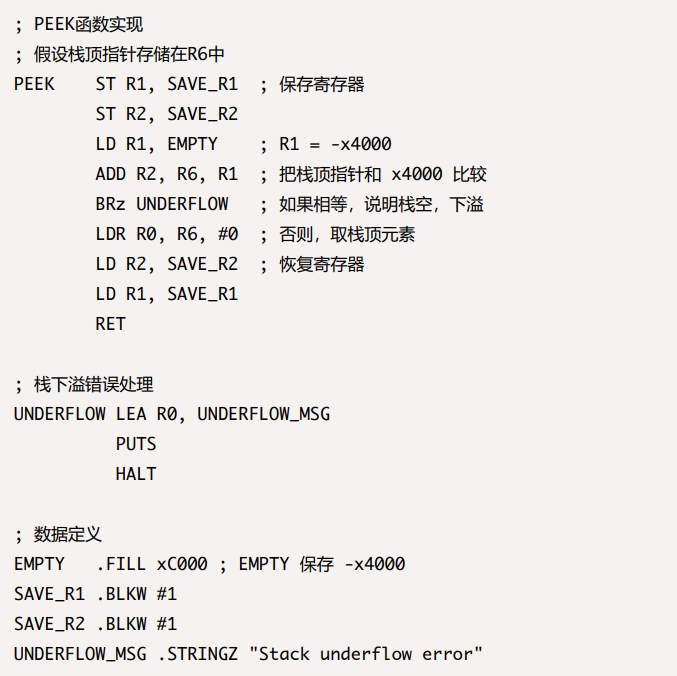


#### 3. What does the stack contain now?



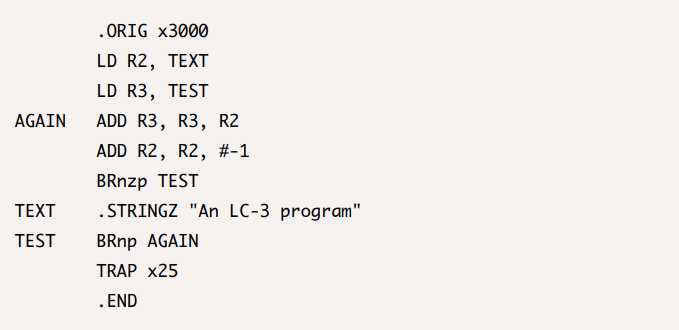
#### T10 Write a function that implements another stack function, PEEK . PEEK returns the value of the top element of the stack without removing the element from the stack. The return value is stored in R0 , so you don't need to save R0 . PEEK should also do underflow error checking: if an underflow occurs, you should output the string "Stack underflow error" and halt. (Suppose the pointer of top of the stack is in R6 , and the stack can only take up the memory space from x3FFF to x3FF0 )

编写一个实现另一个堆栈函数PEEK的函数。PEEK返回堆栈顶部元素的值，而不从堆栈中移除该元素。返回值存储在R0中，因此不需要保存R0。PEEK还应该进行下溢错误检查：如果发生下溢，您应该输出字符串“Stack underflow error”并停止。（假设栈顶的指针在R6，栈只能占用从x3FFF到x3FF0的内存空间）



# HW6

#### T1 How many times is the loop executed? When the program halts, what is the value in R3? (If you do not want to do the calculation, it is okay to answer this with a mathematical expression.)



前两⾏程序先将 TEXT 和 TEST 处内存值存到 R2 和 R3 中， TEXT 处值显然

为 A 的 ASCII 码 x41 ，但 TEST 处值需要考虑 BR 跳转值，而这⼜需要考虑

到字符串 "An LC-3 program" 的⻓度为 16，因此 TEST 处指令为 BRnp #-20 ，

对应⼗六进制 x0BEC .

之后的循环依据 R2 值是否为 0 决定终⽌，每次循环将 R2 值减小 1，因而循环

执⾏共 65 次。每次循环中将 R3 加上本次的 R2 值，因此最终值为*BECH* +(65*D* + 1 ) × 65*D* /2 =144*DH*，也就是 x144D 或⼗进制 5197.

#### T2 Two students wrote interrupt service routines for an assignment. Both service routines did exactly the same work, but the first student accidentally used RET at the end of his routine, while the second student correctly used RTI . There are three errors that arose in the first student’s program due to his mistake. Describe any two of them.

两名学生为一项程序编写了中断服务程序。两个服务程序都做了完全相同的工作，但第一个学生在程序结束时意外使用了RET，而第二个学生正确使用了RTI。由于第一个学生的错误，他的课程出现了三个错误。描述其中的任意两个。

1. 因为是中断服务，所以 R7 中并不保存返回地址，而 RET 会跳转到 R7 保

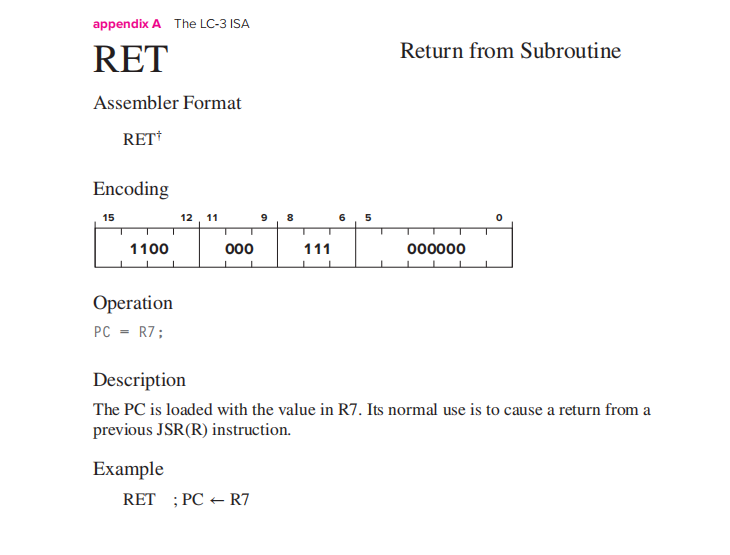
存的地址，因此在中断服务中直接使⽤ RET 会跳转到错误的位置

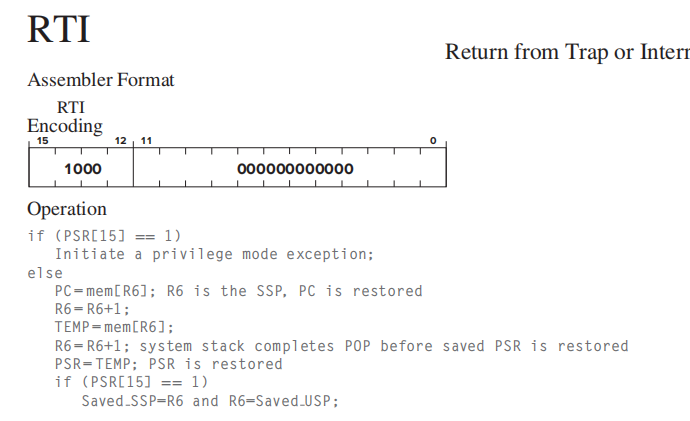
2. 直接调⽤ RET 并没有恢复在中断服务中被保存下来的 Processor Status

Register，因而并没有从特权模式恢复到⽤⼾模式，之前的状态码也没有被

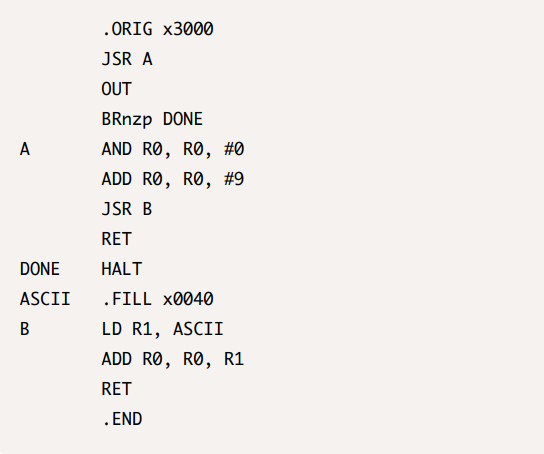
恢复

3. RET 也不涉及对栈的操作，但从中断服务恢复，应该要恢复栈指针





#### T3 After learning the instruction JSR , a student wrote the following program to print a character to the console.



#### 1. What does the student intend to print?

想打印的是字符 I ，因为如果不考虑实际跳转，逻辑上在 OUT 前 R0 中

保存的是 x0049，对应 ASCII 码表中的字符 I

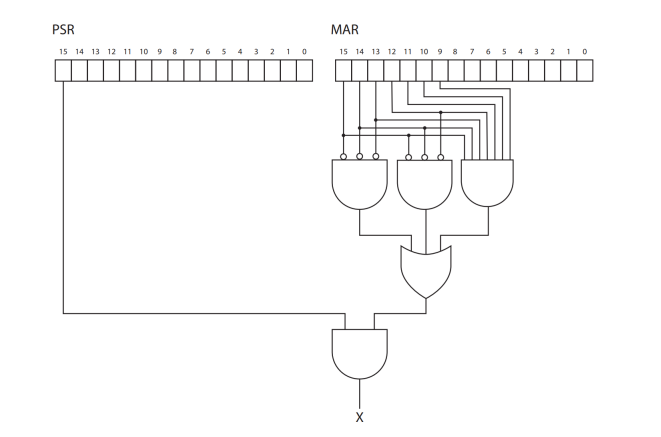
#### 2. Will the program print the character? Can you explain why?

但实际上什么也不会输出，因为调⽤了⼀次 JSR 后再次调⽤ JSR 时没有

保存原本的返回地址 R7 ，因此返回地址已经被覆盖，因而从 B 中回到 A

中再次执⾏ RET 会在 A 中出现死循环，⽆法真正执⾏到 OUT

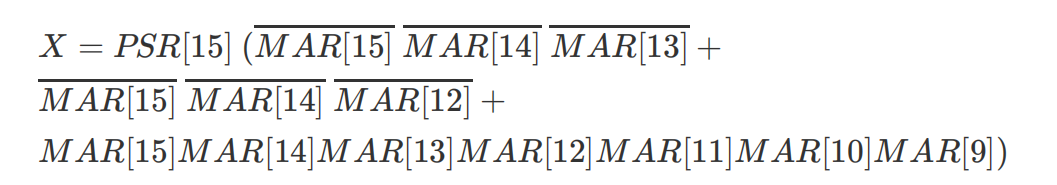
#### T4 The LC-3 contains the following logic.



#### Can you tell what signal X is? When will X be set to 1?

根据逻辑写出表达式，以下简记XY表⽰ X AND Y， X+Y表⽰ X OR Y，

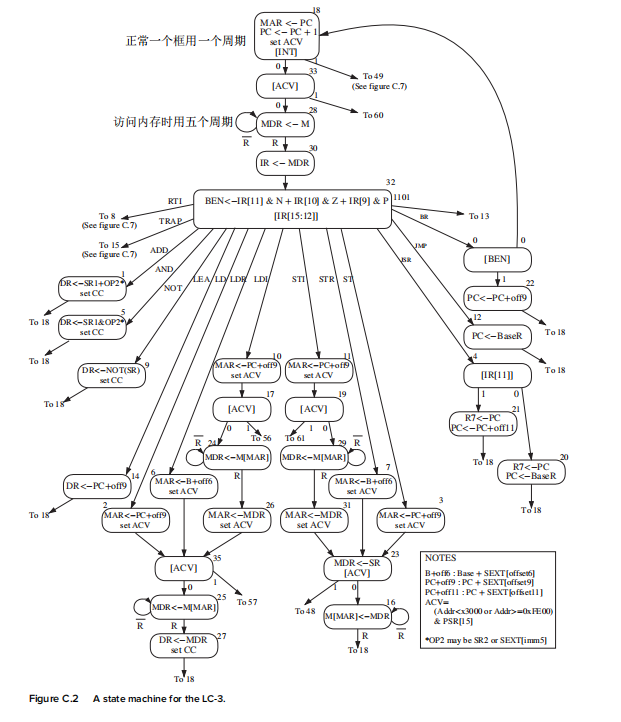
表⽰ X[N] 的第 N 位



判断当 PSR 的最⾼位为 1 时 (此时位于⽤⼾模式)，且当 MAR 的值处于

[x0000,x3000)∪ [xFE00,xFFFF] 时 (此时 MAR 位于特权内存区)， X 为 1，否则

为 0. 由 Figure C.2 中所述， X 为信号 ACV .



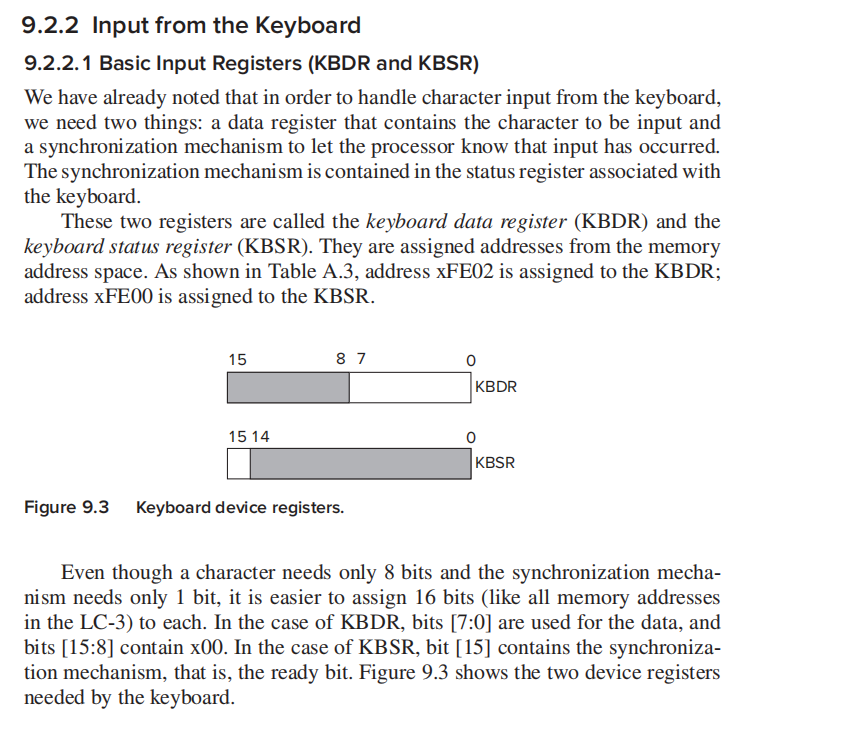
#### T5 Imagine that you are writing a simple LC-3 program that is designed to receive a character from the keyboard and then display that character on the screen.

#### 1. How do you check in LC-3 if there is a new character input from the keyboard?

KBSR 最⾼位为1

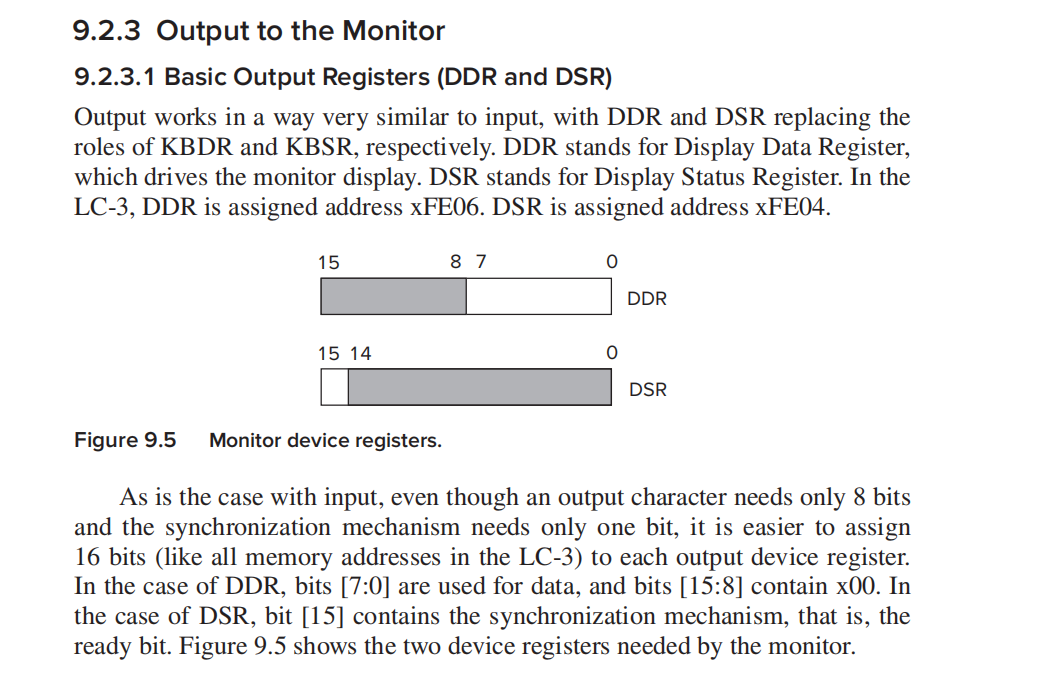
#### 2. Once a new character input is detected, how do you read this character from the keyboard?

读取 KBDR ，并清除 KBSR 最⾼位

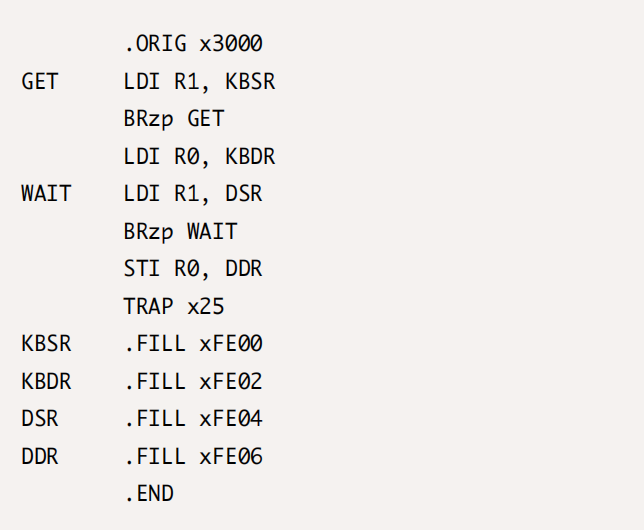


#### 3. How do you display the read character on the screen?

当 DSR 最⾼位为 1 的时候，写⼊ DDR



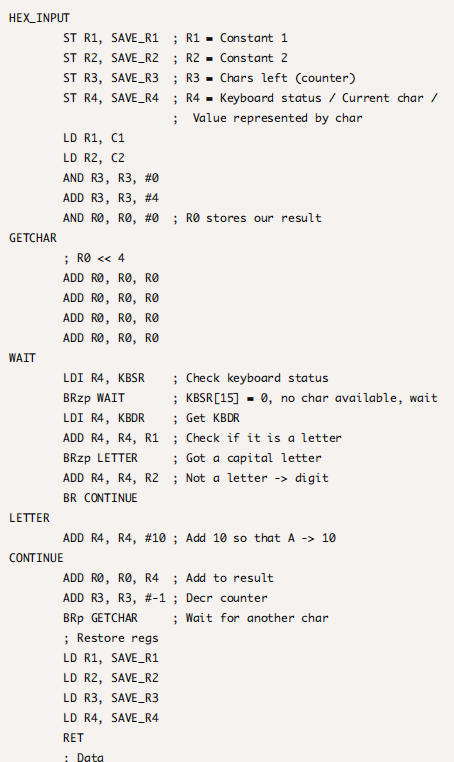
#### 4. Provide a simple LC-3 assembly code snippet that demonstrates this process.

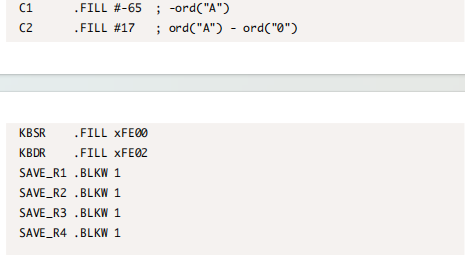


#### T6 Here's a subroutine that takes 4 chars in hex from keyboard and store the value they represent in R0 using polling technique. Note that it assumes all possible input characters are in 0123456789ABCDEF .

这里有一个子程序，它从键盘上取4个十六进制字符，并使用轮询技术将它们表示的值存储在R0中。请注意，它假设所有可能的输入字符都在0123456789ABCDEF中。

#### 1. Fill in the blanks (denoted by underlines \_ ) to complete the program.





#### 2. Briefly explain what the four consecutive ADD R0, R0, R0 do.

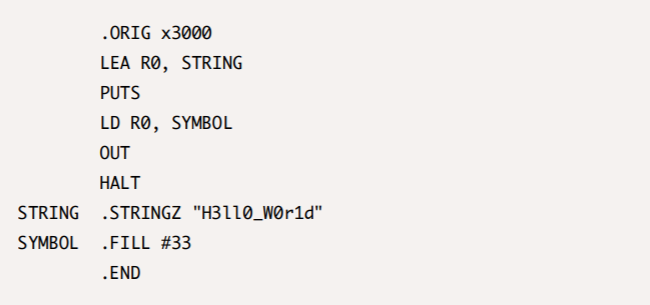
R0 << 4, so that we can add the next value to the result.

#### 3. We have no idea what R0 stores before the subroutine is called, so we placed the instruction AND R0, R0, #0 before the label GETCHAR in order to clear R0 . Is this instruction necessary? Why or why not?

在调用子例程之前，我们不知道R0存储了什么，所以我们将指令AND R0，R0，#0放在标签GETCHAR之前，以清除R0。这个指示有必要吗？为什么？

It is not necessary. Dirty data will get shifted out

#### T7 The following program needs to be assembled and stored in LC-3 memory:



#### 1. What is the output of the program?

33：！

H3ll0\_W0r1d!

#### 2. How many bytes of memory does the program occupy?

18 × 2 = 36 bytes. (Each instruction takes 2 bytes; don't forget the \0 at

the end of the string.)

#### T8

#### 1. What problem might arise if a program does not check KBSR before reading KBDR ?

It might reads an input character more than once.

#### 2. What problem might arise if the keyboard does not check KBSR before writing to KBDR ?

It might overwrite an input character before it is processed.

#### 3. Which one of the two problems mentioned above is more likely to occur? Justify your answer.

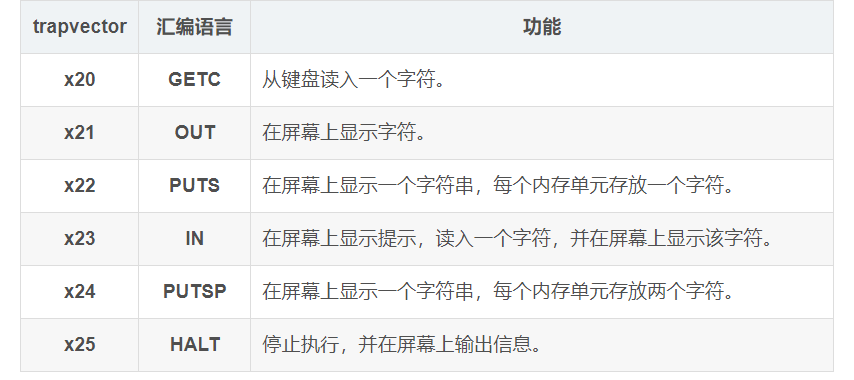
The first scenario is more likely to happen, because CPU is much faster

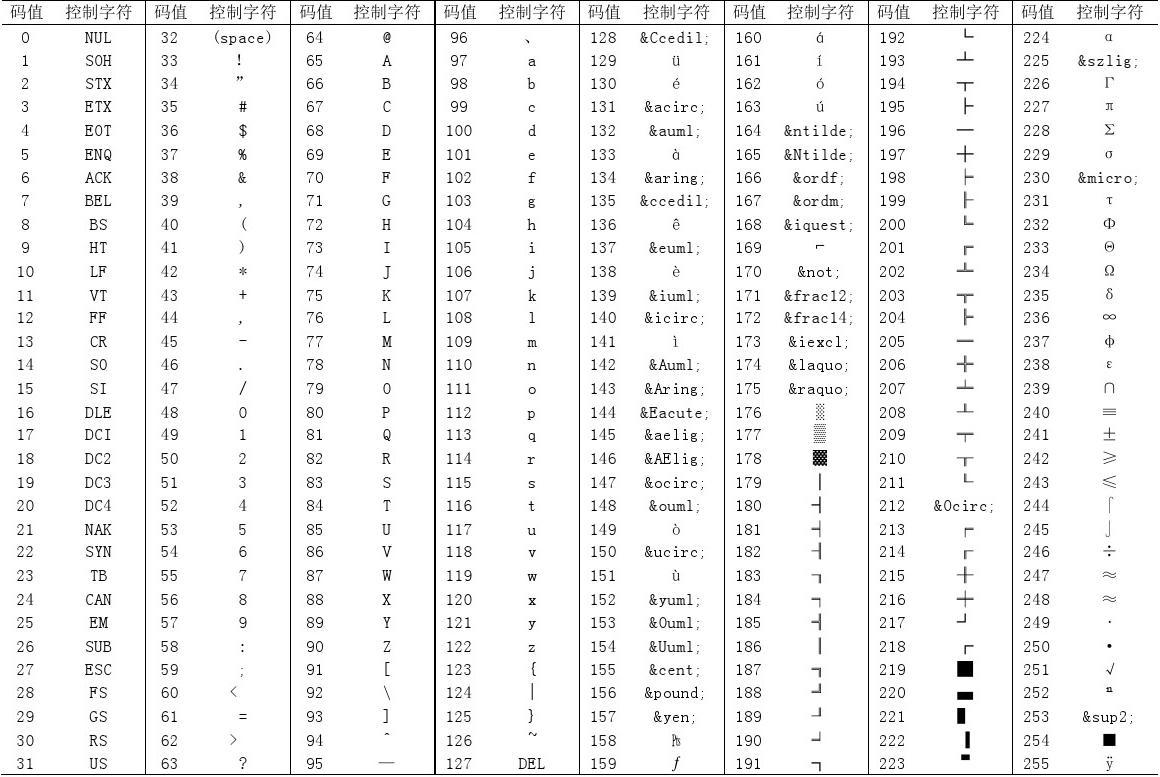
than human input.

#### T9 What is the output of this program? Assume all registers are initialized to 0 before the program executes.

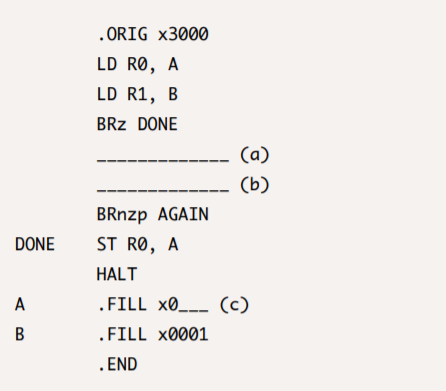


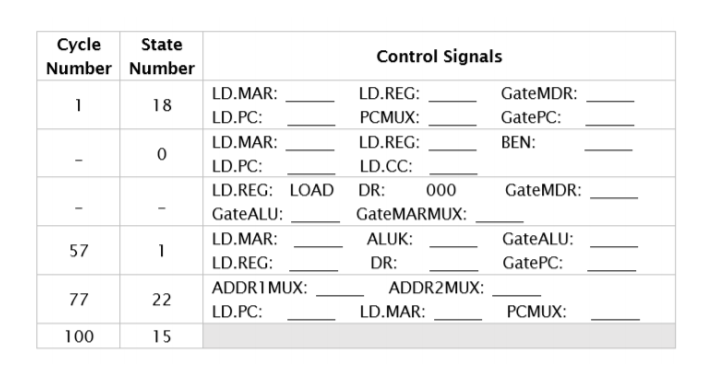
F !





#### T10





该程序仅使用R0和R1。请注意，第（a）行和第（b）行指示两个缺失的指令，第（c）行包括一些缺失的数字。还请注意，程序中的一条指令必须再次标记为in，但现在该标签已丢失。在执行程序之后，A的内容是x1800。总共执行了9条指令。在执行过程中，计算机的一些快照是在某些时钟周期拍摄的。下表按拍摄快照的周期号排序。请注意，我们没有说明内存访问需要多少时钟周期。

#### 1. How many clock cycles does a memory access take? (We define the clock cycles of a memory access as the cycles during which the computer stays at a state that does either MDR<−M[MAR] or M[MAR]<-MDR )

一次内存访问需要多少个时钟周期？（我们将内存访问的时钟周期定义为计算机处于MDR＜−M[MAR]或M[MAR]<-MDR状态的周期）

4

#### 2. Given values for A and B , what does the program do?

因为最后存到 A 处的值为 x1800，因此循环需要被执⾏，题⼲⼜提到⼀共执⾏

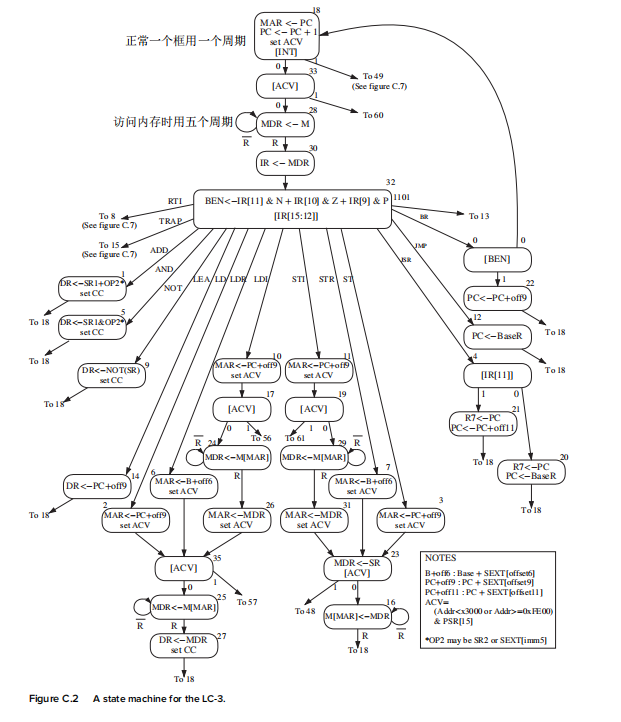
了 9 条指令，因此循环只被执⾏了⼀次。同时，可以确定在 BRnzp AGAIN 之

后，就紧跟着⼀条 BR 指令，且正确跳转到了 DONE 位置。

假设每次访存需要经过 *x* 个时钟周期，在状态机中数⼀遍，可以发现

* LD 指令需要 7 + 2*x* 个时钟周期，
* 未进⾏跳转的 BR 指令需要 5 + *x* 个时钟周期，
* 进⾏跳转的 BR 指令需要 6 + *x* 个时钟周期，
* ST 指令需要 6 + 2*x* 个时钟周期，
* HALT也就是 TRAP 指令需要 5 + *x* 个时钟周期，

这样已经有了 9 条指令中的 7 条经过的时钟周期数。



从表中可以得知，第 77 个周期下，处于状态 22，查表看到此时位于 BR 指令执

⾏过程中最后⼀个状态，因为⼀共就 100 个周期，所以可以猜测此时为最后⼀个BR 指令，这样就得到⽅程 6 + 2*x* + 5 + *x* = 100 − 77，解得 *x* = 4. (其实也可

以尝试猜测为倒数第⼆个 BR 指令，这时有⽅程 6 + *x* + 6 + 2*x* + 5 + *x* =

100 − 77，得到 *x* = 1.5 不可能；而再往前就会出现 *x* < 0 那就更不能成⽴了)

因此可以确定每次访存需要 4 个时钟周期。

从第 77 个周期往回推，可以得到第 57 个周期为循环中的最后⼀条指令，此处位

于状态 1，也即是 ADD ，再往前⼀⾏的 DR 为 000，也就是此处在更新 R0 ，因

此均不可能为跳转，因而 AGAIN 只能位于 BRz DONE 这⾥. 表格再往前推，是

状态 0，对应的是 BR 指令，因此这只能为第⼀次执⾏到 BRz DONE ，由之前推

理，对应的周期数应为 2(7 + 2*x*) + 5 + *x* = 19 + 5*x* = 39.

考虑到执⾏完挖空的两⾏后，执⾏到 BRz DONE 时会跳转到 DONE ，也就是 57

个周期处的指令执⾏完后会得到结果为 0，而且是 ADD ，而因为跳转后 R0 存

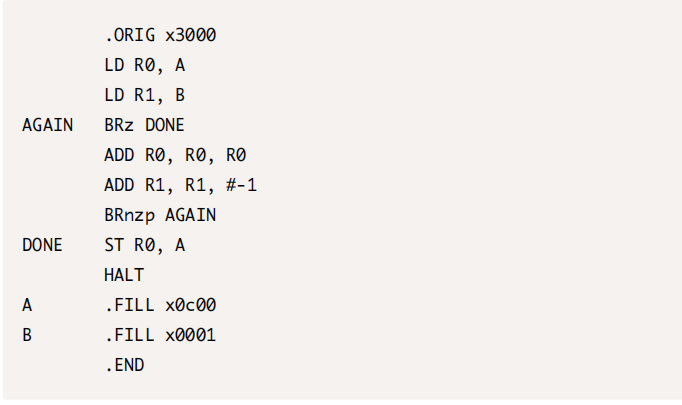
的是 x1800，因而此处更新的是 R1 ，而 R1 之前为 x0001，因此 (b) 处空的指

令为 ADD R1, R1, #-1 .

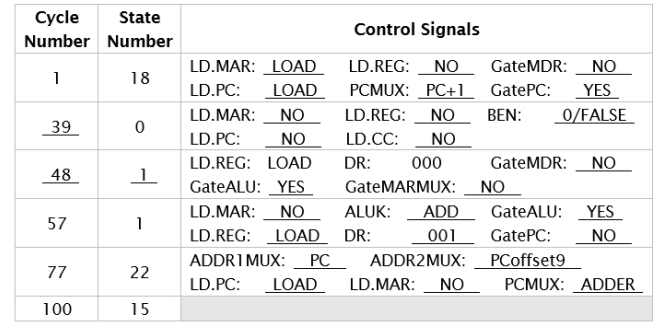
对于 (a) 处的空，因为整个程序只涉及 R0 和 R1 ，而只能使⽤ ADD , AND ,

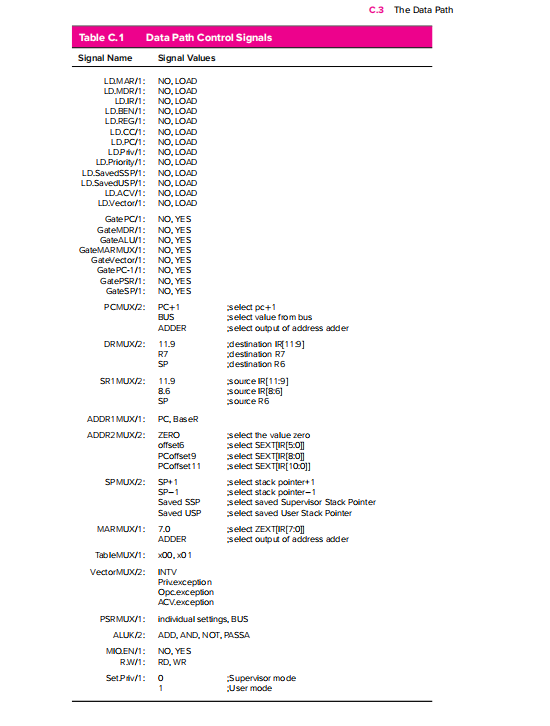
NOT ，从 x0\_\_\_ 变为 x1800 不能通过 AND 或是 NOT ，所以 (a) 处只能是

ADD R0, R0, R0 ，且 (c) 处为 c00 .



LD.XX 实际上是对 XX 的写使能信号，该程序功能为计算 *A* ⋅ 2*B* 并存到 A 处。





LAB1