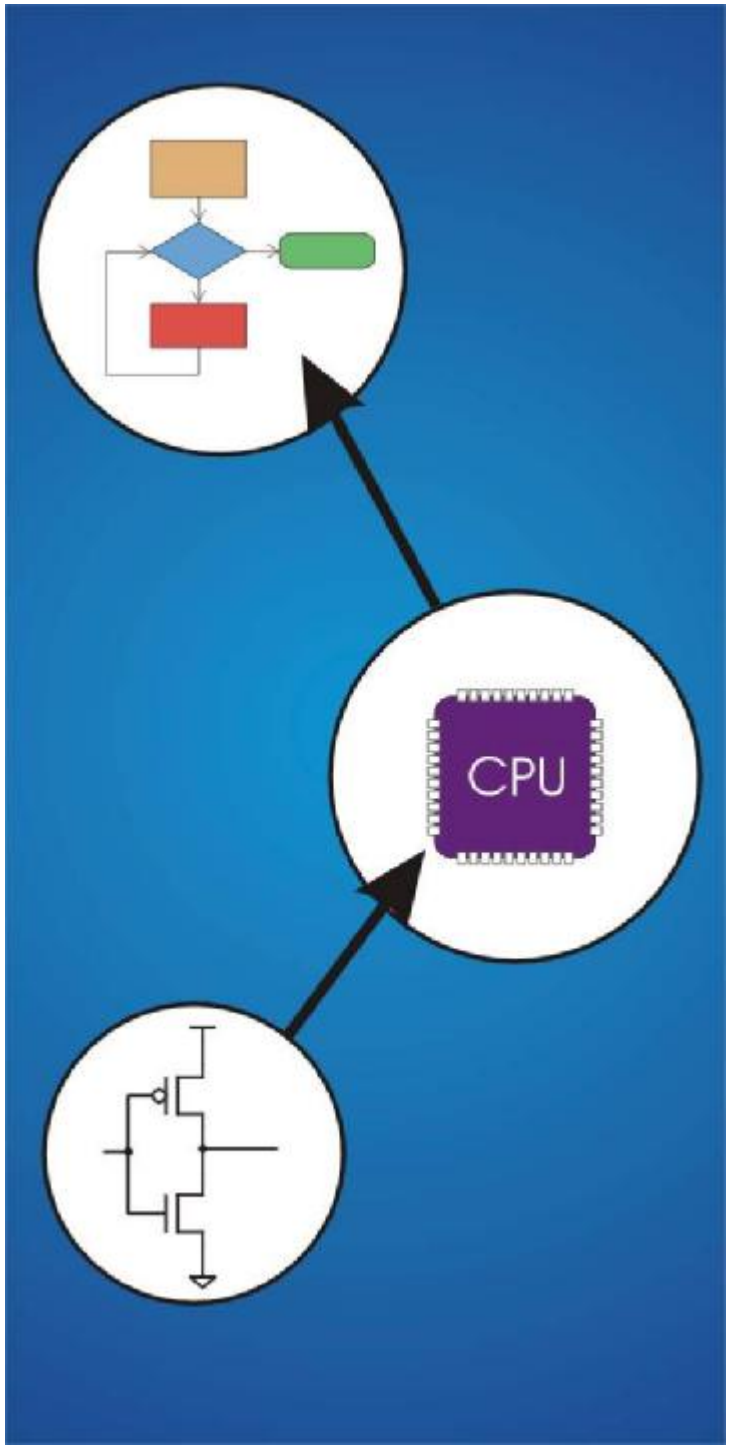


Chapter 6
编程和调试

编程方法

通过编程方式让计算机为我们解决问题

问题求解

- 问题描述
- 将问题描述转化为算法. 算法特性：
  1）有限性　2）确定性　3）可计算性
- 将算法用LC-3的机器代码实现。（或借助高级语言和编译器）

程序调试

- 程序不工作？怎样消除程序的错误（bug）
- 通过观察程序运行过程中寄存器和相关内存的变化情况。设置断点等。

Time spent on the first can reduce time spent on the second!

问题描述

问题的描述一般采用自然语言,但有时候可能在某些地方表述的不是很明确甚至可能不完整。

具体到“字符统计的问题”:“计算一个文件中某个特定字符的出现次数,该字符由键盘输入;结果在显示器上回显”

- “文件”存放在哪里? 文件的长度是多少,或则我怎么能知道什么时候到达文件的尾部?
- 统计结果的输出形式是什么? 是十进制吗?
- 当字符是字母时, 统计要区分大小写吗?

怎么解决?

- 询问提出问题的人,或者
- 作个决定然后记录下来.

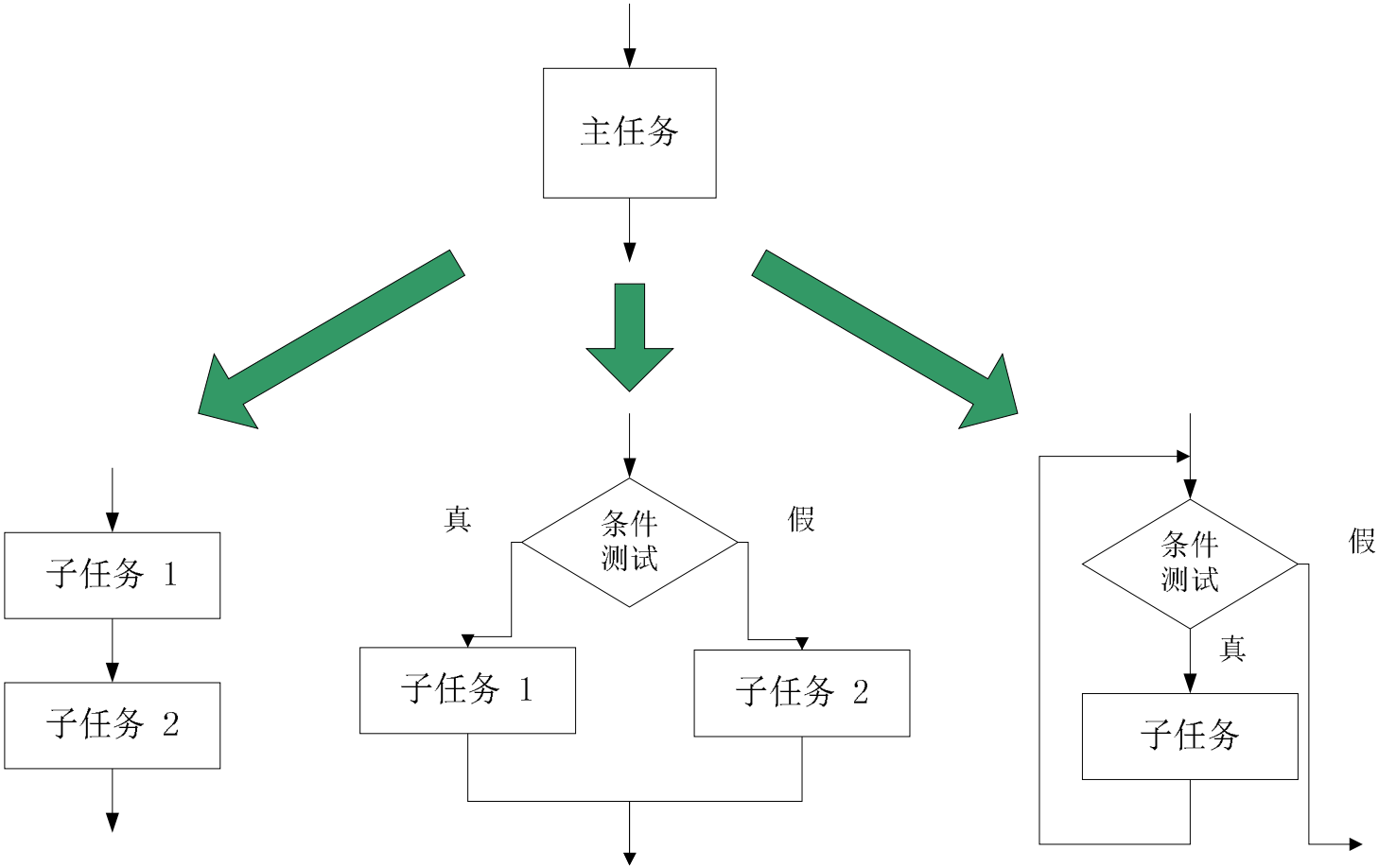
方法：系统分解

系统分解: 利用计算机编程求解复杂问题的一个重要方法

原则: 将一个复杂的任务分解成若干个子任务;再将每个子任务进一步分解成更小的任务;如此反复,直到任务小到方便编程实现（机器指令）。

子任务的实现结构：三种特定的程序结构，顺序、条件和循环

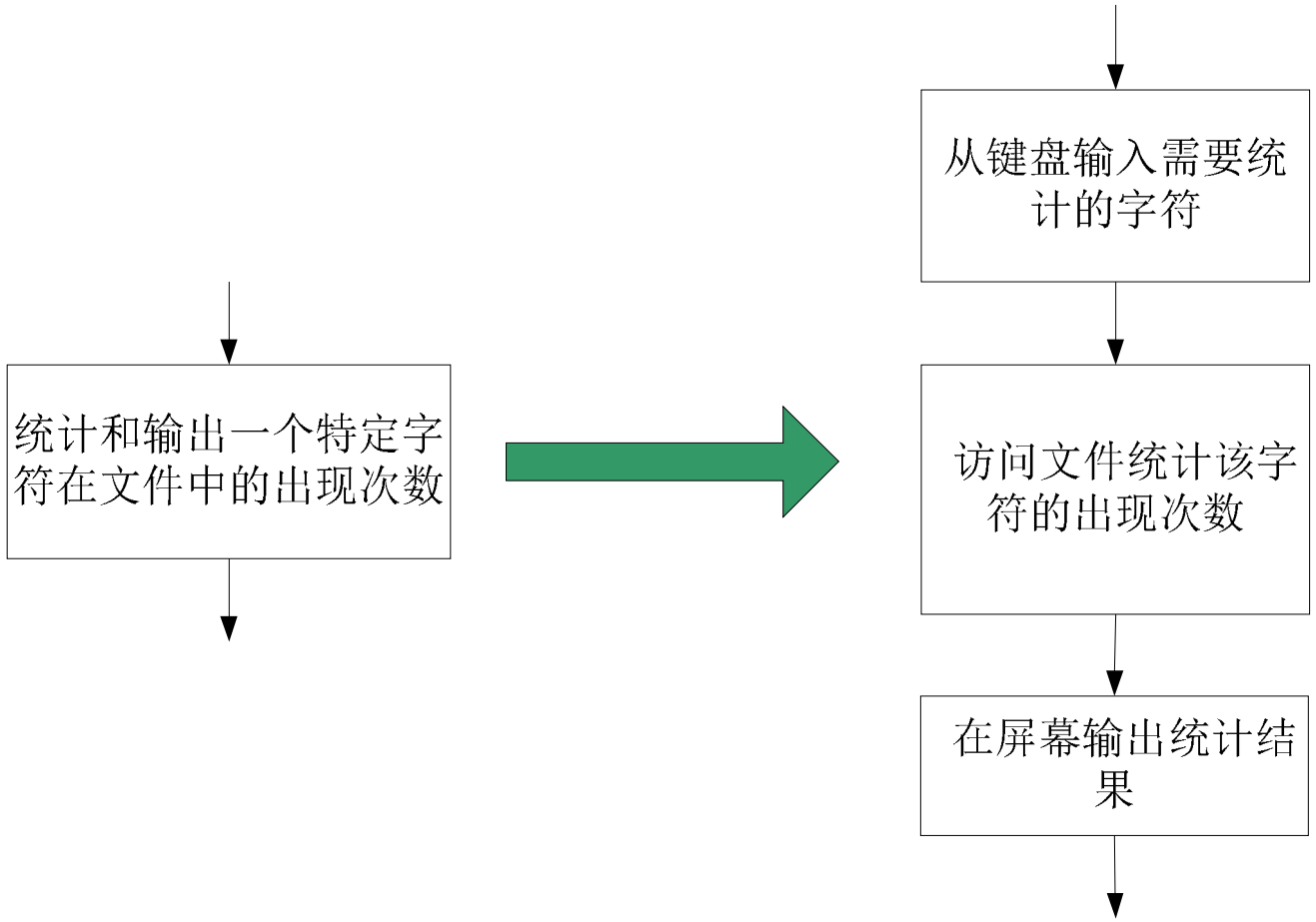
三种基本的程序构建结构



顺序执行结构　　*条件执行结构*　　*循环执行结构*

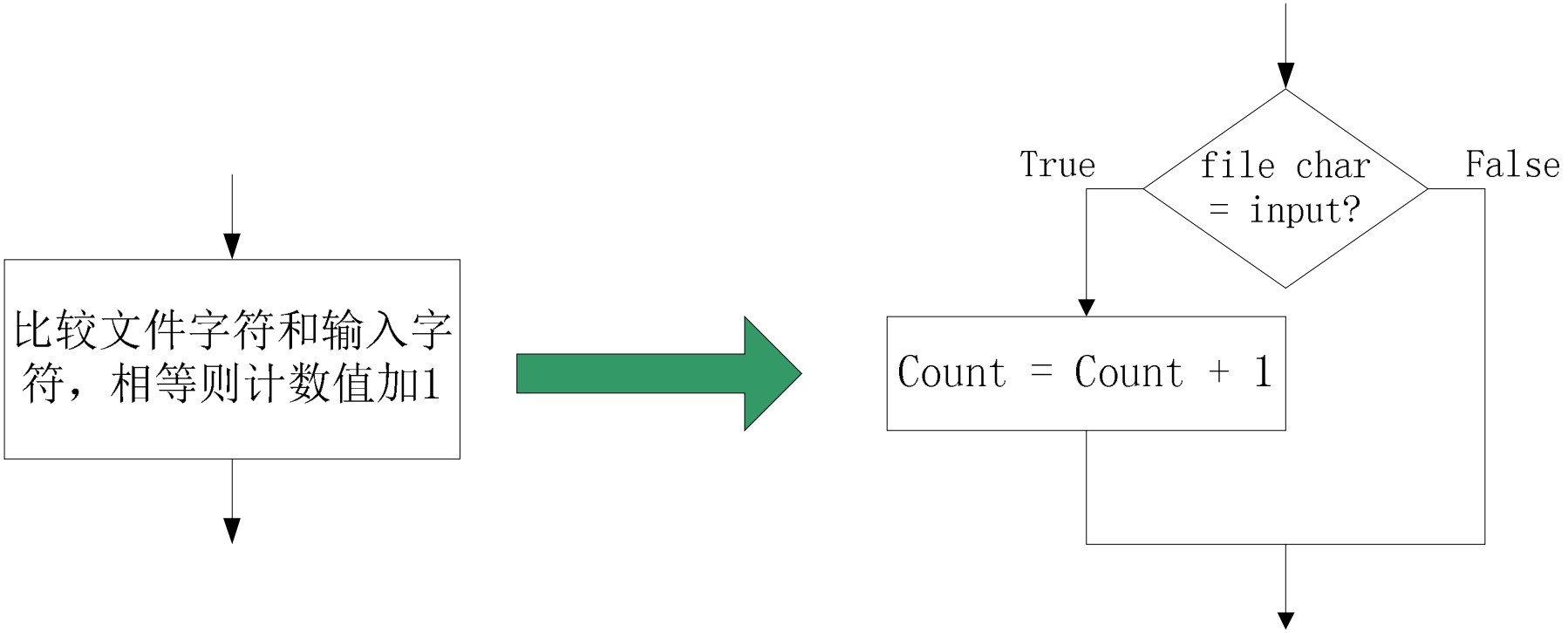
顺序执行结构

先完成子任务1,再完成子任务2, 以此类推。。。。



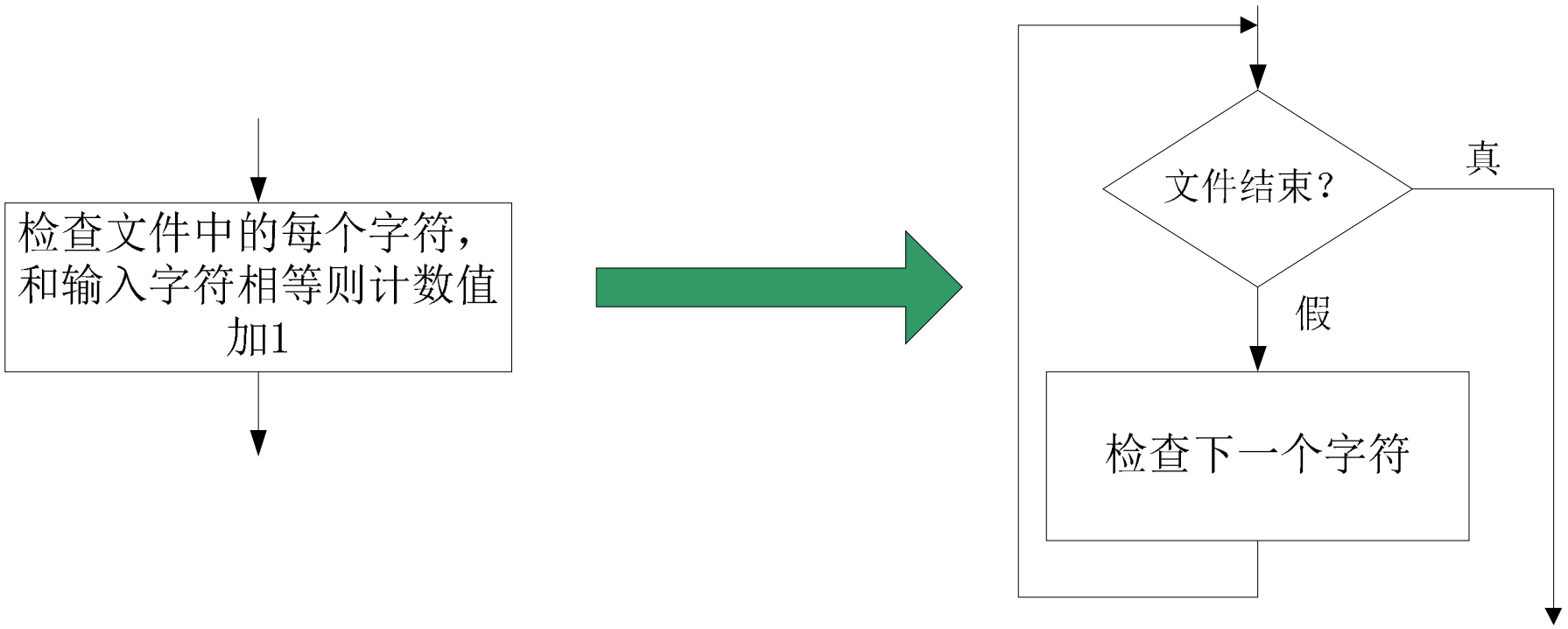
条件执行结构

如果条件成立则执行子任务1，否则执行子任务2



循环

反复执行一个子任务，直到某个满足特定条件成立才退出，



解决问题的技巧

学会把实际问题转换为用一个个子任务逐步解决的过程。

- 类似字谜游戏.
  - Ø系统的初始状态是什么
  - Ø期望的结束状态是什么？
  - Ø怎么从一个状态转移到令一个状态

- 自然语言和三种基本结构的对应关系
  - Ø“先做 A 然后做 B” Þ 顺序
  - Ø“如果 G成立, 则做 H” Þ 条件
  - Ø“对每一个存在的X, 做 Y” Þ 循环
  - Ø“做 Z 直到 W成立” Þ 循环

LC-3 基本执行结构的实现

我们怎么利用 LC-3的控制指令来实现三种基本的执行结构？

顺序执行结构

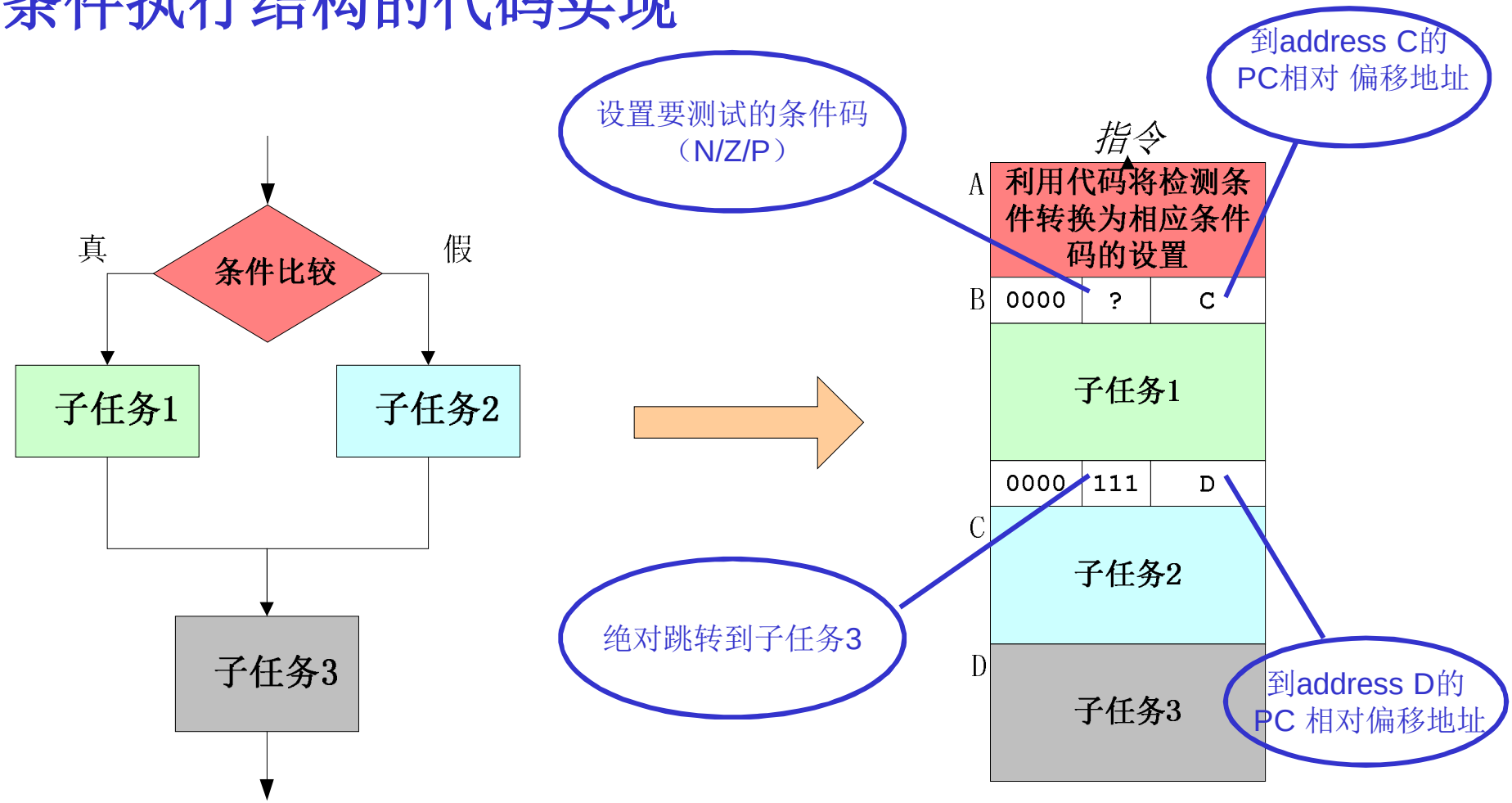
- 指令默认顺序执行,当前指令执行完后自动执行下一条指令
  因此对于顺序执行结构来说不需要特殊的指令支持

条件和循环执行结构

- 利用代码将检测的条件转换为N,Z,P的条件码.
  Example:
  条件: **“Is R0 = R1?”**
  代码: 先**R1** 减 **R0**; 如果相等, 条件码**Z** 将会设置为‘**1**’.

- 然后利用**BR**指令判断相应的条件码跳转到对应子任务的入口.

条件执行结构的代码实现

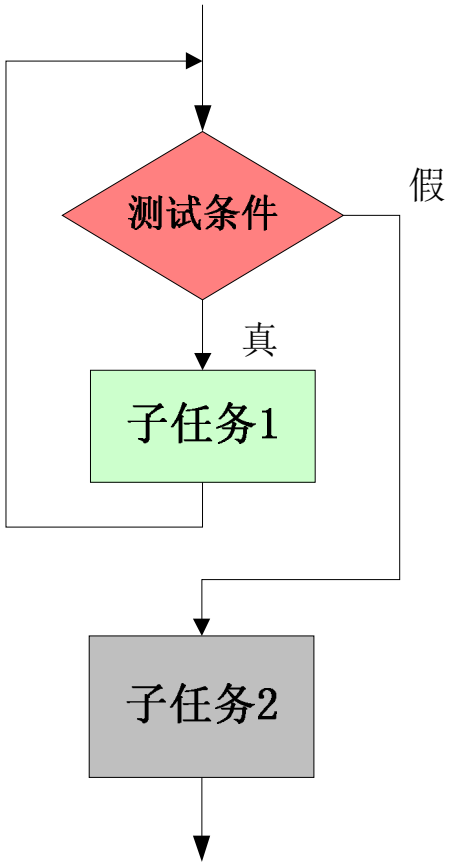


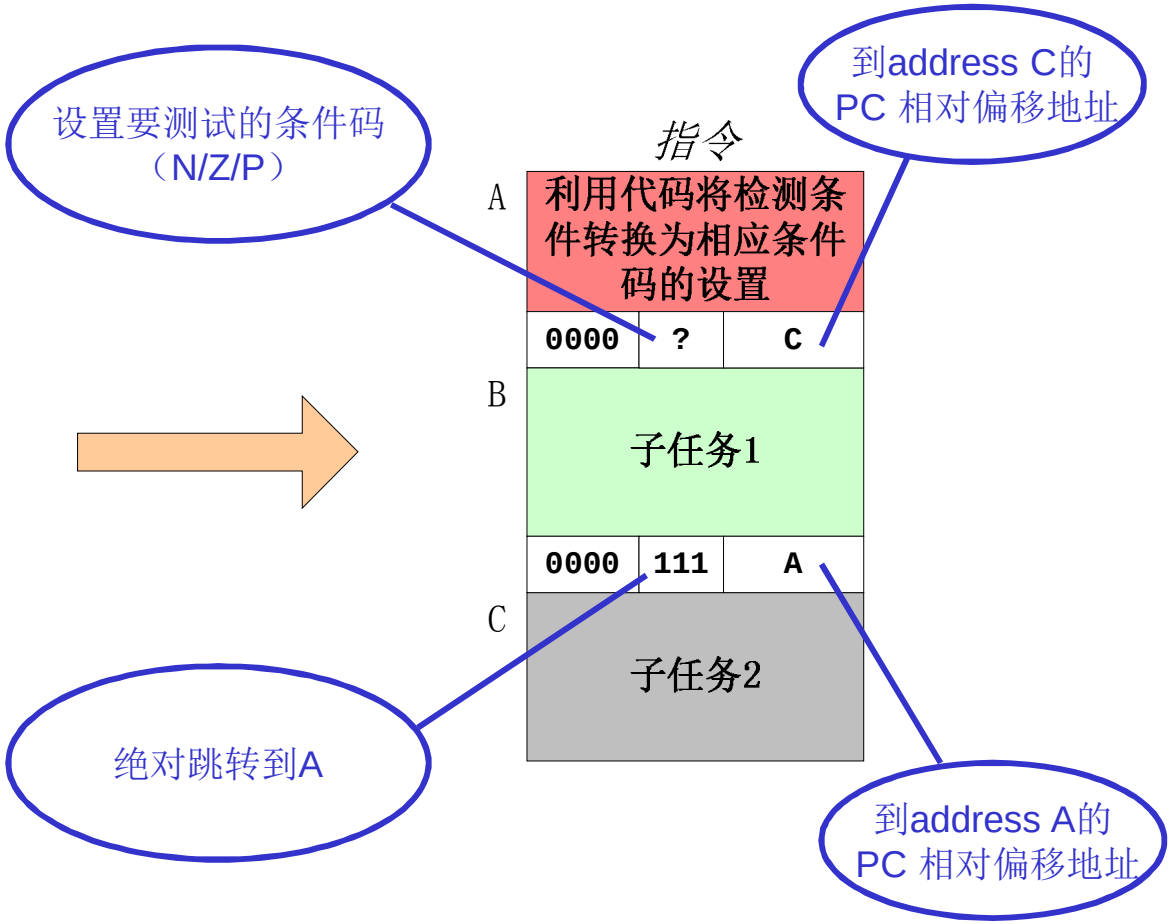
假设所有子任务的偏移都在PC相对寻址的地址范围内

假设判断条件为R0=R1则跳转到子任务2，否则执行子任务1

假设A=x3000 C=x3020 D=x3040 写出相应的代码

循环执行结构的代码实现





假设所有子任务的偏移都在PC相对寻址的地址范围内

假设判断条件为R0=30则跳转到子任务2，否则一直执行子任务1

假设A=x3000 C=x3040 写出相应的代码

IF-THEN-ELSE

In C：

```
if (count < 0)
count = count + 1;
```

In LC-3：

```
        LD R0, count   //LDI
        BRpz endif
        ADD R0, R0, #1
Endif:
        ST R0,count
```

WHILE-DO

In C：

```
while (count > 0)
{
a = a + count;
count--;
}
```

In LC-3：

```
        LD R1, a //假定PC相对寻址可访问
        LD R0, count
while: BRnz endwhile
        ADD R1, R1, R0
        ADD R0, R0, #-1
        BRnzp while
Endwhile:
        ST R1, a
        ST R0, count
```

DO-WHILE

In C:

```
do
{
if (a < b)
a++;
if (a > b)
a--;
} while (a != b)
```

In LC-3:

```
        LD R0, a
        LD R1, b
        <R2=a-b>
repeat: BRpz secondif
        ADD R0, R0, #1
        <R2=a-b>
secondif:
        BRnz until
        ADD R0, R0, #-1
until:
       <R2=a-b>
        BRnp repeat
```

FOR

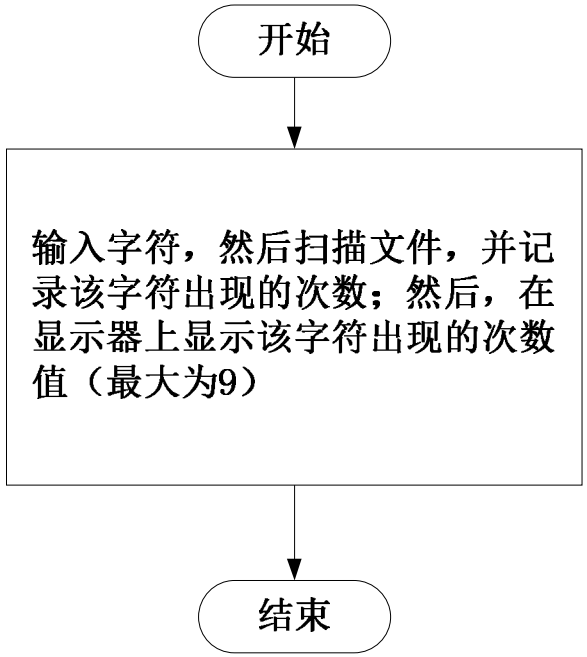
In C:

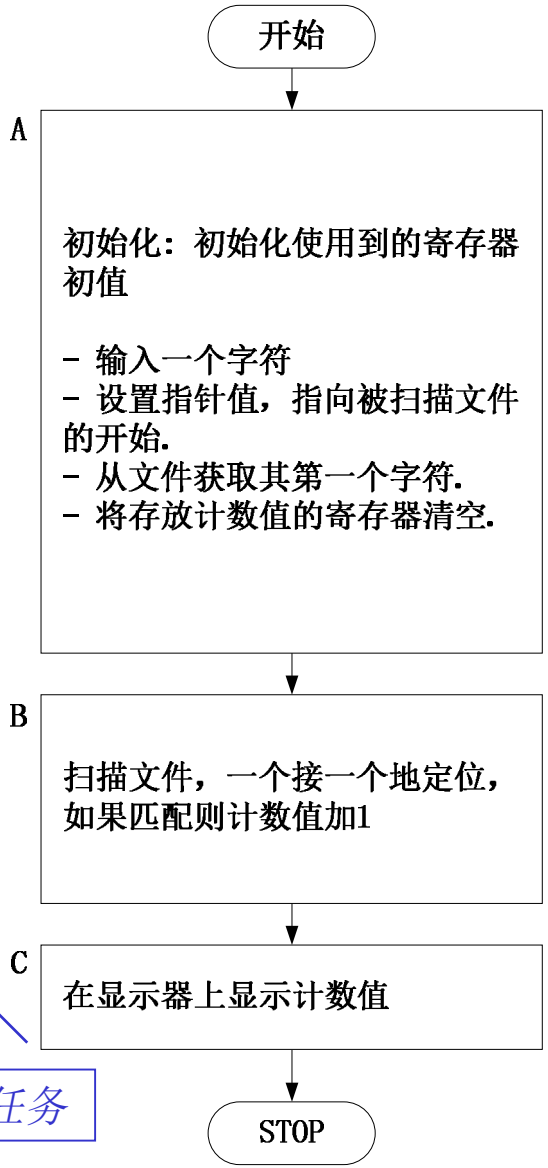
```
for (i = 3; i <= 8; i++)
{
a = a + i;
}
```

In LC-3:

```
        LD R0, a
        AND R1, R1, #0
        ADD R1, R1, #3
for:    ADD R2, R1, #-8
        BRp endfor
        ADD R0, R0, R1
        ADD R1, R1, #1
        BRnzp for
endfor:
```

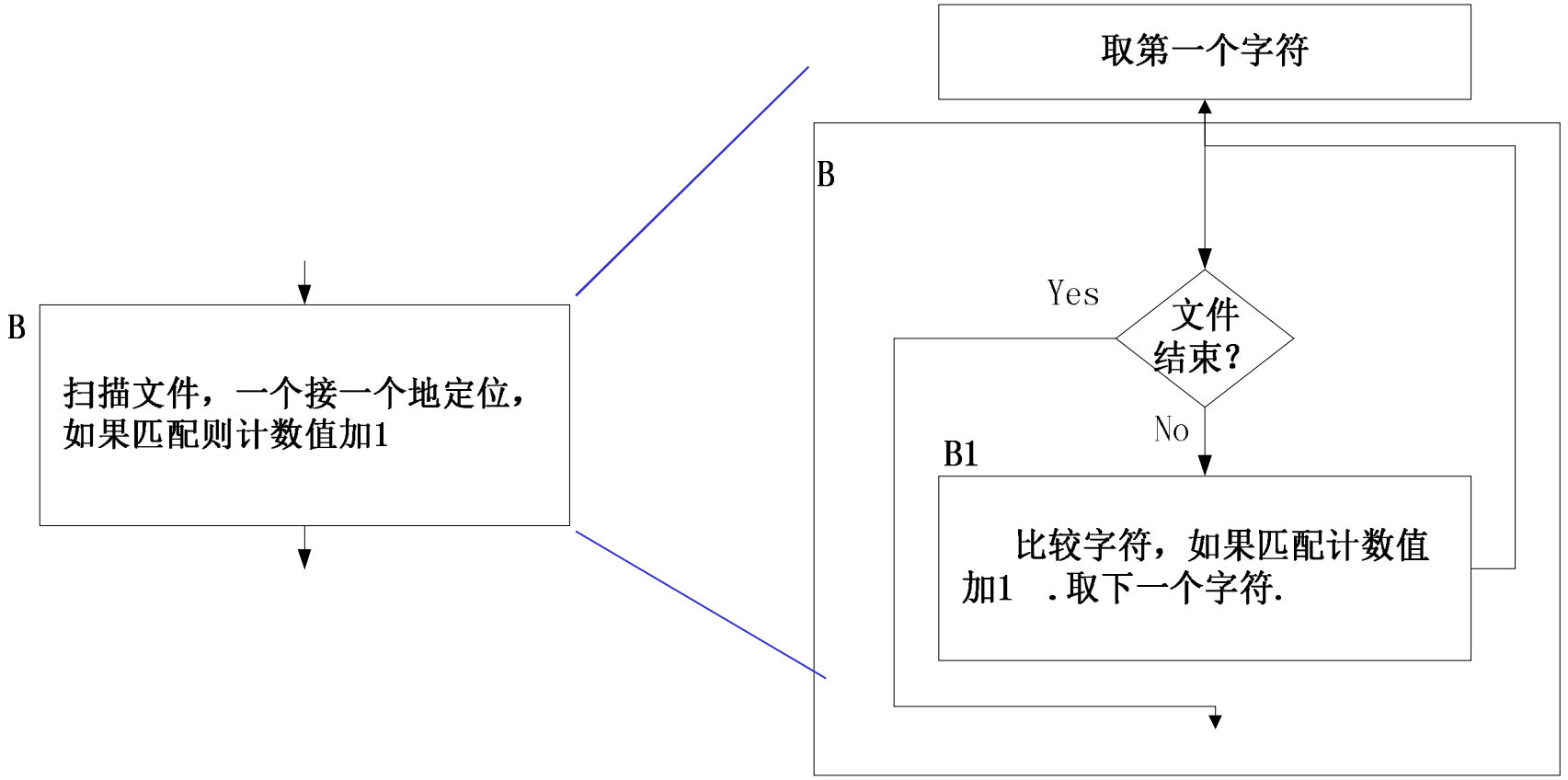
逐步细化解决实例: 字符统计





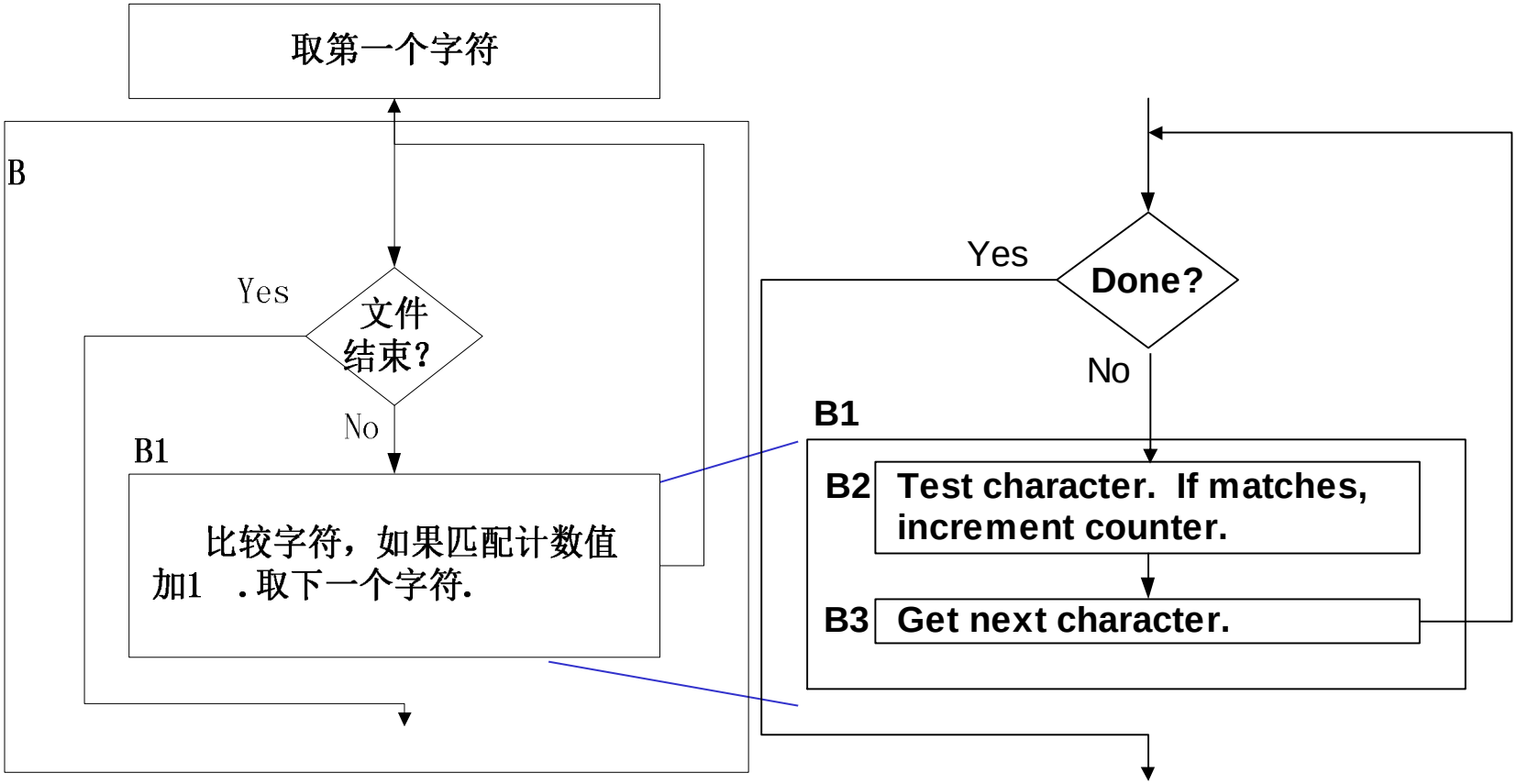
第一次细化:将一个大的问题分解为三个顺序执行的子任务

子任务B细化



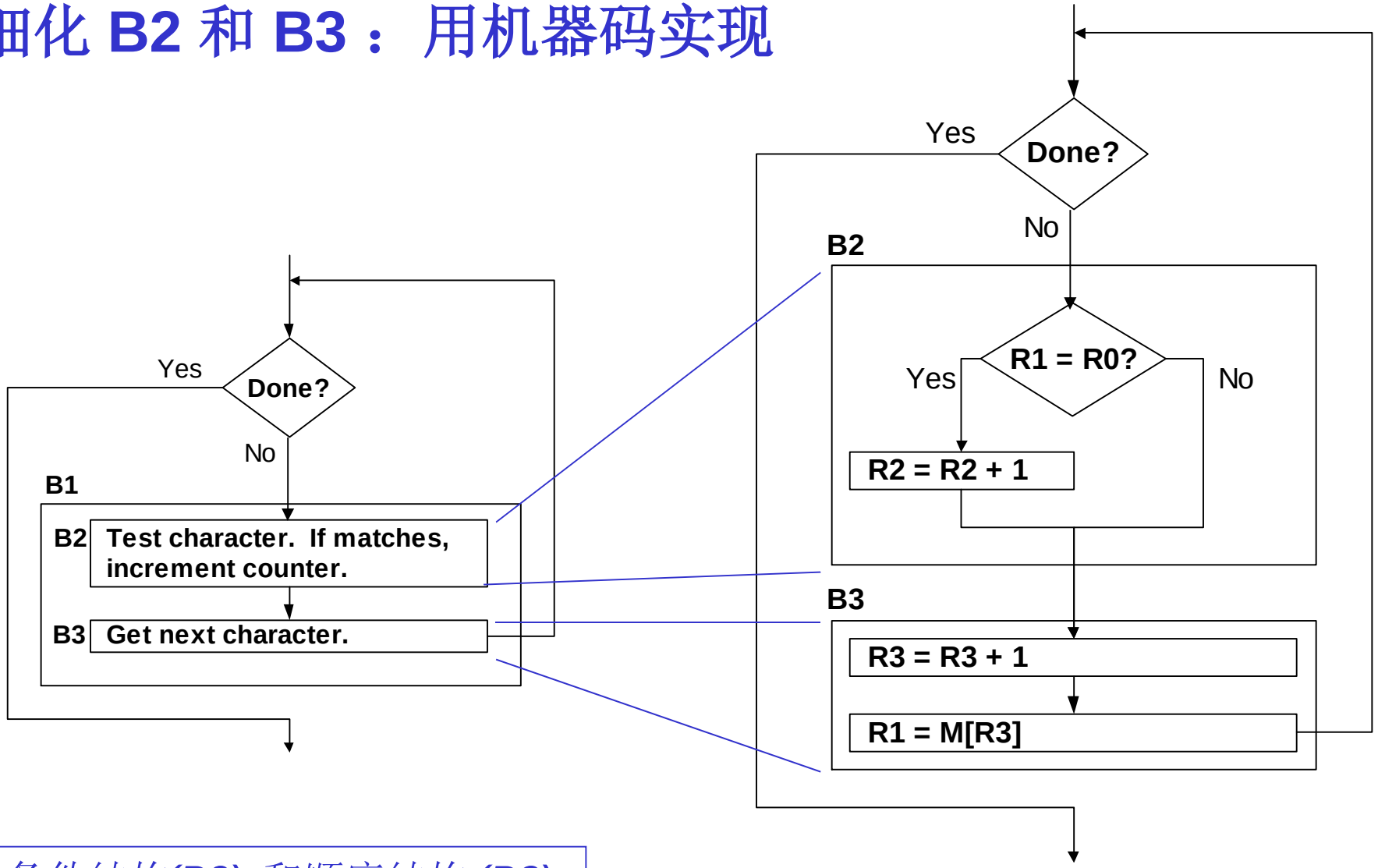
将子任务B细化为循环结构

子任务B1细化



将子任务B1细化为两个顺序执行的子任务B2和B3

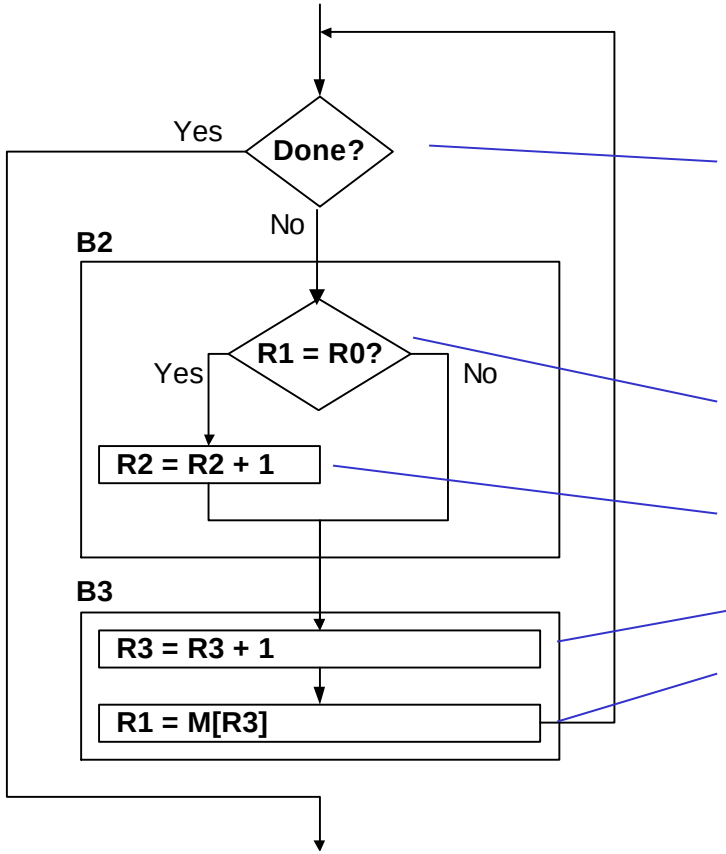
细化 B2 和 B3： 用机器码实现



条件结构(B2) 和顺序结构 (B3). 使用机器码实现

最后一步: LC-3 指令

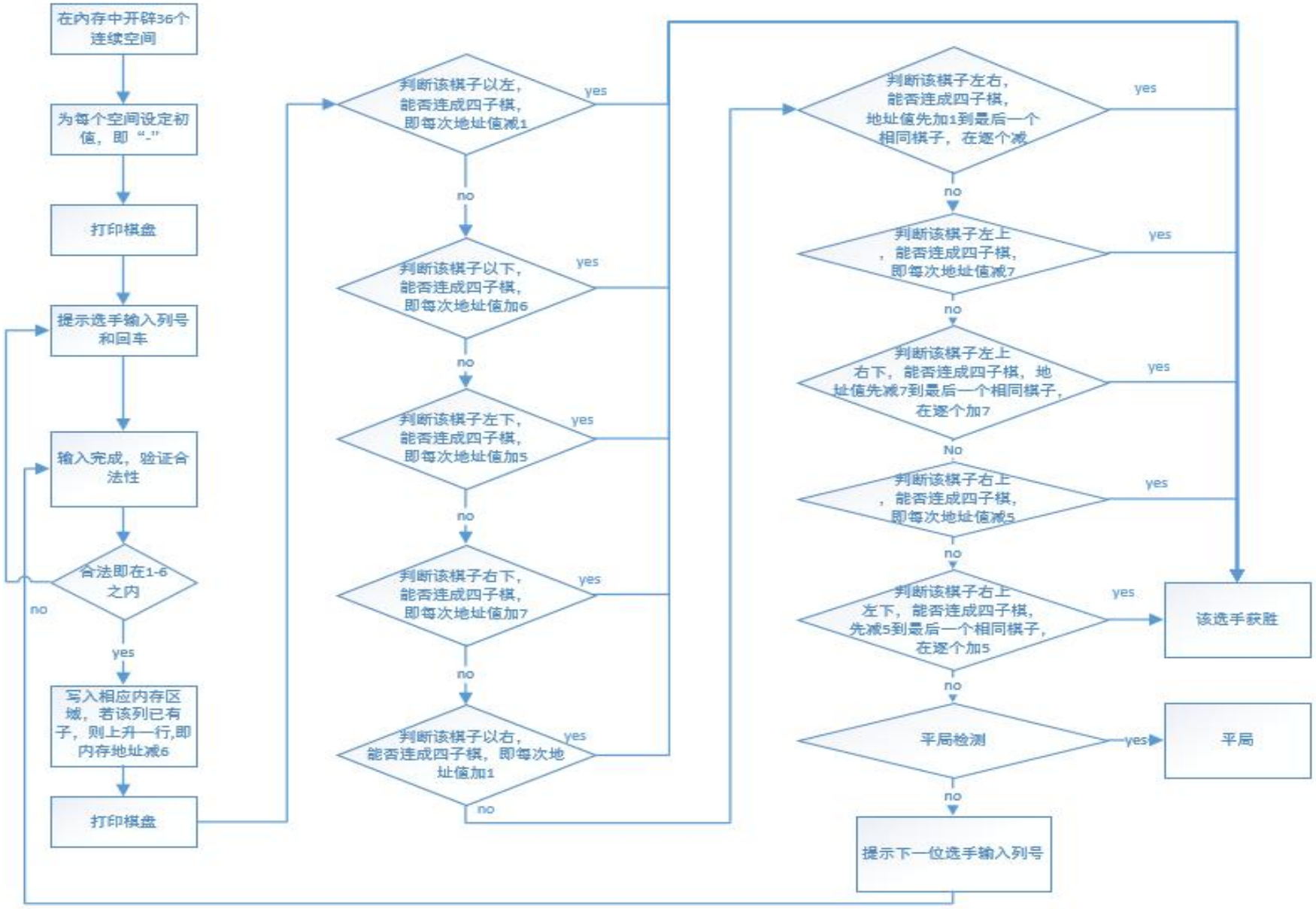
为程序添加注释是一个良好的习惯。注释以‘；’开头




```
; Look at each char in file.
0001100001111100  ; is R1 = EOT?
0000010xxxxxxxxx  ; if so, exit loop
; Check for match with R0.
1001001001111111  ; R1 = -char
0010010011000001
0010010000000001  ; R1 = R0 – char
0000101xxxxxxxxx  ; no match, skip incr
0010100101000001  ; R2 = R2 + 1
; Incr file ptr and get next char
0010110111000001  ; R3 = R3 + 1
0110001011000000  ; R1 = M[R3]
```

Pcoffset现在还不知道，
等完成所有代码后确定

简单游戏的任务分解


在内存中开辟36个连续空间
为每个空间设定初值，即“-”
打印棋盘
提示选手输入列号和回车
输入完成，验证合法性
合法即在1-6之内
no
yes
写入相应内存区域，若该列已有子，则上升一行,即内存地址减6
打印棋盘
判断该棋子以左，能否连成四子棋，即每次地址值减1
yes
no
判断该棋子以下，能否连成四子棋，即每次地址值加6
yes
no
判断该棋子左下，能否连成四子棋，即每次地址值加5
yes
no
判断该棋子右下，能否连成四子棋，即每次地址值加7
yes
no
判断该棋子以右，能否连成四子棋，即每次地址值加1
yes
no
判断该棋子左右，能否连成四子棋，地址值先加1到最后一个相同棋子，在逐个减
yes
no
判断该棋子左上，能否连成四子棋，即每次地址值减7
yes
no
判断该棋子左上右下，能否连成四子棋，地址值先减7到最后一个相同棋子，在逐个加7
yes
No
判断该棋子右上，能否连成四子棋，即每次地址值减5
yes
no
判断该棋子右上左下，能否连成四子棋，先减5到最后一个相同棋子，在逐个加5
yes
该选手获胜
no
平局检测
yes
平局
no
提示下一位选手输入列号

调试

写好了程序，却发现执行结果和预期的不一样？

怎么办?

当你在城市里迷路了你会怎么做?

- ✘ 到处乱窜，希望能找到目的地?
- P 回到一个认识的地方?然后查看地图？

在程序调试过程中,追踪程序的执行和迷路时查看地图一样重要

回到一个确认正确的启示点

- 跟踪每条指令的执行顺序是否正确
- 观察指令执行后对应的执行结果
- 并和期望的结果相比较

调试的基本操作

任何一个调试环境都应提供以下的调试手段:

1. 能显示当前寄存器或内存单元中的值
2. 能改变当前寄存器或内存单元中的值
3. 按顺序单步执行程序
4. 需要时停止程序执行

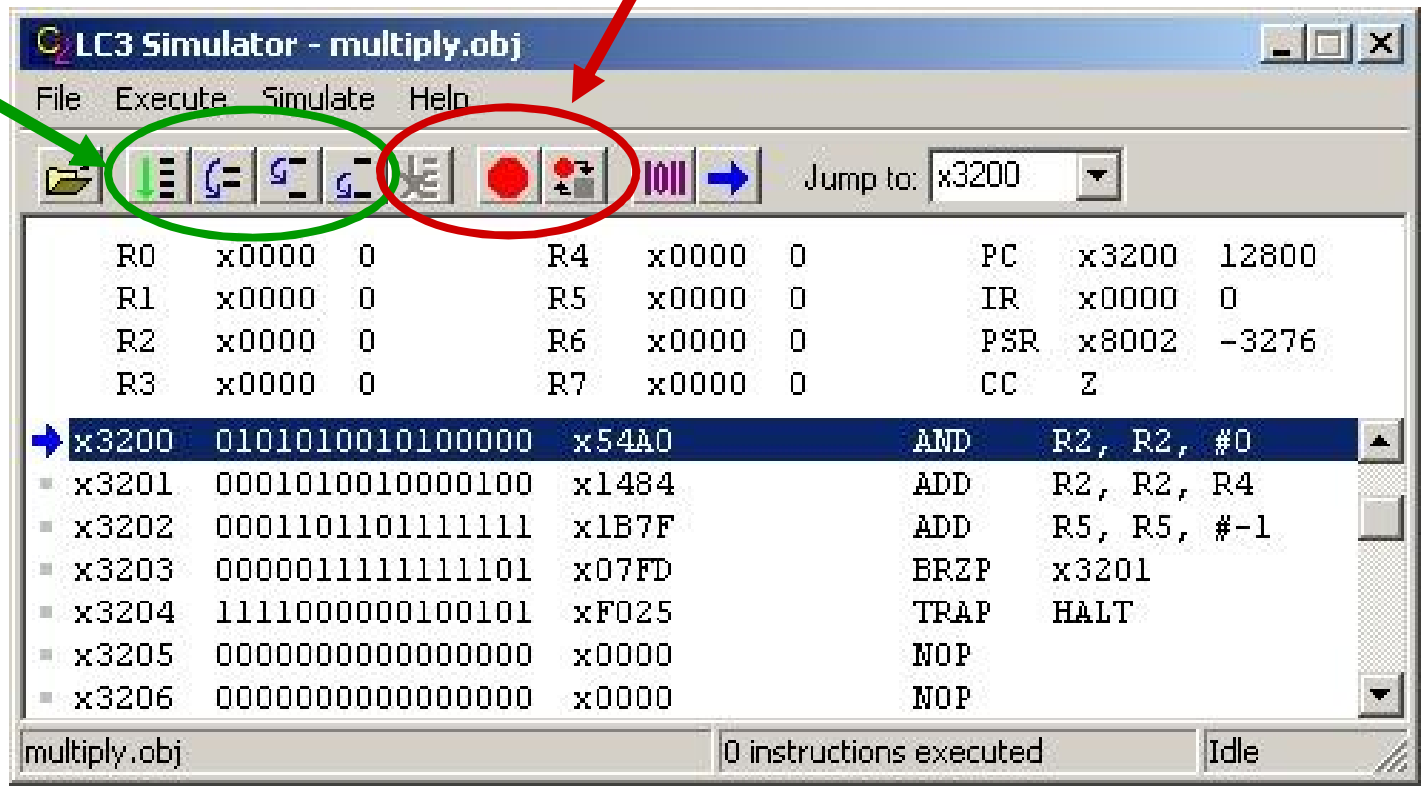
不同级别的编程语言提供不同的调试工具

- 高级语言(C, Java, ...)
  通常具备源代码级的调试工具
- 机器语言的调试工具
  - Ø 软件仿真器
  - Ø 操作系统 “监控” 工具
  - Ø 硬件在线仿真器 (ICE)
    - – 通过额外的硬件来提供机器语言级的控制

LC-3 仿真器

停止执行，
设置断点

控制指令
执行顺序



设置、显示
寄存器或
内存的值

常见的错误

语法错误

- 输入错误造成的非法操作。（**scanf -> scvnf**，错误的函数）
- 在机器语言级别程序设计时很少发生. **(**操作码 **0010** 误写成**0011**，但是合法的**)**
- 高级语言写源代码时容易发生。此时通常编译不通过

此类错误按编译器提示比较容易修正

逻辑错误

- 程序没有语法错误，能编译执行，但得不到正确的结果
- 需要利用调试器跟踪程序的执行，确认究竟在哪里出现问题。

比较难修正

数据错

- 输入为某些特定数据组合时运行不正常，程序设计人员考虑的不是很周全
- 用大量不同的数据集进行程序测试

比较难修正

跟踪程序的执行

每次执行一部分程序，观察每次执行后寄存器或内存值是否正确

单步

- 每次执行一条指令
- 冗长乏味的,但是非常有效的

断点

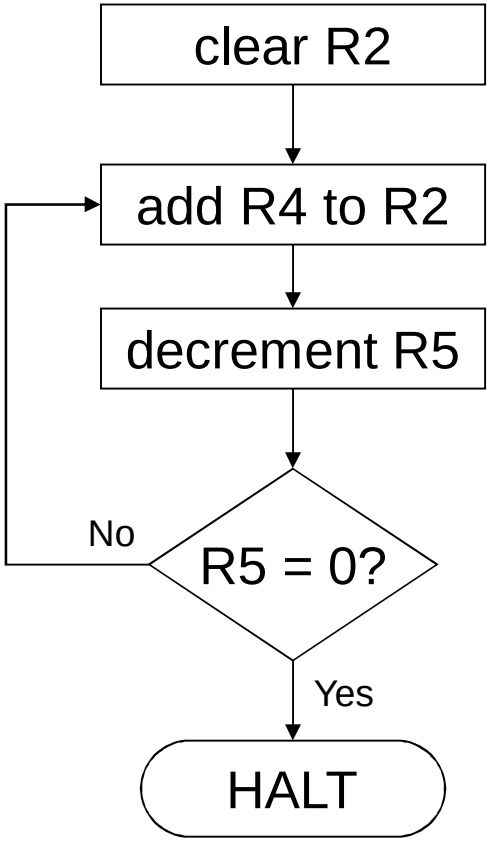
- 执行到特定指令时终止程序执行
- 对具体某个怀疑的点进行测试和观察
  - Ø 定位速度比较快，当你的怀疑是正确的

监视点

- 检测寄存器或内存值发生变化或达到某个特定值时停止程序执行。
- 在你不知道什么时候或什么地点值会发生变化时特别有用

例1: 乘法

将存放在**R4**和**R5**中的两个正整数相乘，结果存放在**R2**中




```
x3200 0101010010100000
x3201 0001010010000100
x3202 0001101101111111
x3203 0000011111111101
x3204 1111000000100101
```

Set R4 = 10, R5 =3.
Run program.
Result: R2 = 40, not 30.

乘法程序的调试

PC and registers at the beginning of each instruction

PC	R2	R4	R5
x3200	--	10	3
x3201	0	10	3
x3202	10	10	3
x3203	10	10	2
x3201	10	10	2
x3202	20	10	2
x3203	20	10	1
x3201	20	10	1
x3202	30	10	1
x3203	30	10	0
x3201	30	10	0
x3202	40	10	0
x3203	40	10	-1
x3204	40	10	-1
	40	10	-1

单步调试

分支断点(x3203)

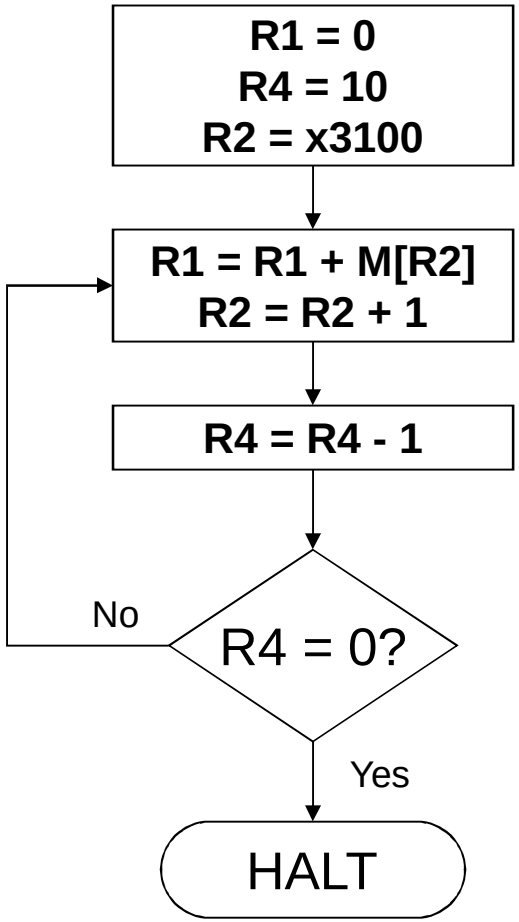
PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	-1
	40	10	-1

应该在这里停止循环

循环执行次数超过了预期。
x3203处的条件语句设置的判断条件有错误

例2：一列数的求和

将存放在起始内存单元为**X3100**的 **10**个数求和，结果存放在**R1**



x3000	**0101001001100000**
x3001	**0101100100100000**
x3002	**0001100100101010**
x3003	**0010010011111100**
x3004	**0110011010000000**
x3005	**0001010010100001**
x3006	**0001001001000011**
x3007	**0001100100111111**
x3008	**0000001111111011**
x3009	**1111000000100101**

开始运行和调试

存放的数据如下表，得到结果R1 = x0024,
正确结果应该为x8135. 出什么问题了?

Address	*Contents*
x3100	**x3107**
x3101	**x2819**
x3102	**x0110**
x3103	**x0310**
x3104	**x0110**
x3105	**x1110**
x3106	**x11B1**
x3107	**x0019**
x3108	**x0007**
x3109	**x0004**

开始单步调试

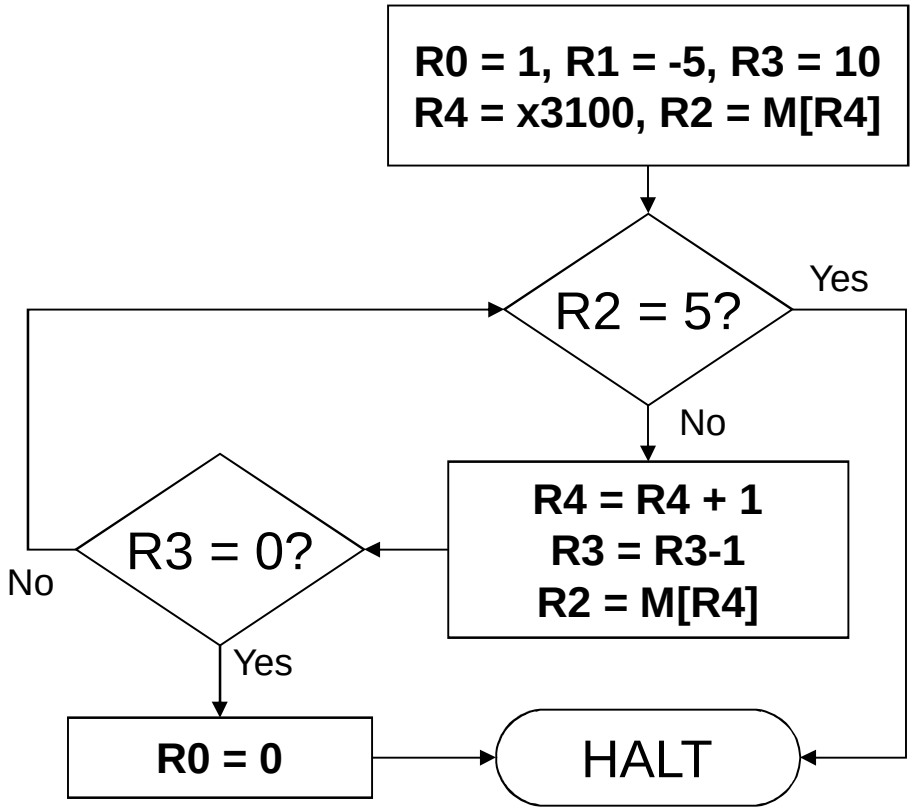
PC	R1	R2	R4
x3000	--	--	--
x3001	0	--	--
x3002	0	--	0
x3003	0	--	10
x3004	0	x3107	10

应该是x3100!

实际得到的是M[x3100], 而不是地址.
怎么修改？（操作码LD和LEA）

例3:查看内存区域是否包含5

以x3100为起始内存单元连续存放十个整数，如果其中包含5则设置R0=1，否则设置 R0为 0。




```
x3000 0101000000100000
x3001 0001000000100001
x3002 0101001001100000
x3003 0001001001111011
x3004 0101011011100000
x3005 0001011011101010
x3006 0010100000001001
x3007 0110010100000000
x3008 0001010010000001
x3009 0000010000000101
x300A 0001100100100001
x300B 0001011011111111
x300C 0110010100000000
x300D 0000001111111010
x300E 0101000000100000
x300F 1111000000100101
x3010 0011000100000000
```

开始运行和调试

数据如下表， x3108内存单元存放数据‘5’，但结果 R0 = 0, 而不是 R0 = 1.出什么问题了?

Address	*Contents*
x3100	**9**
x3101	**7**
x3102	**32**
x3103	**0**
x3104	**-8**
x3105	**19**
x3106	**6**
x3107	**13**
x3108	**5**
x3109	**61**

是不是没有把所有数据都检查到?
在x300D 处设置个断点，观察条件语句是否正确地跳转

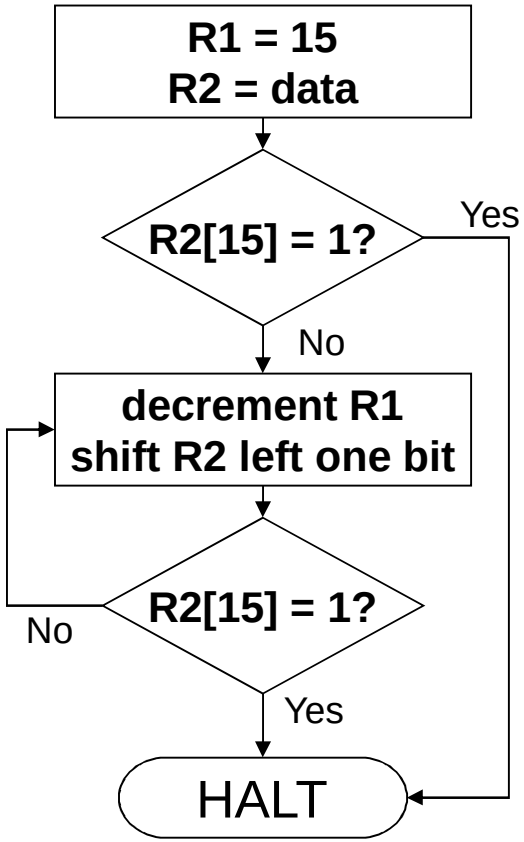
PC	R0	R2	R3	R4
x300D	1	7	9	x3101
x300D	1	32	8	x3102
x300D	1	0	7	x3103
	0	0	7	x3103

← Didn’t branch back, even though R3 > 0?

Branch uses condition code set by loading R2 with M[R4], not by decrementing R3.
Swap x300B and x300C, or remove x300C and branch back to x3007.

Example 4: 查找字中的第一个‘1’

内存单元x3009存放一个字. 程序返回该数据中第一个‘1’的位置在R1中（0~15），如果没有1，则R1=-1




```
x3000 0101001001100000
x3001 0001001001101111
x3002 1010010000000110
x3003 0000100000000100
x3004 0001001001111111
x3005 0001010010000010
x3006 0000100000000001
x3007 0000111111111100
x3008 1111000000100101
x3009 0011000100000000
```

开始运行和调试

程序大多情况都运行正常，但当检查的数据为0时……

在跳回的分支语句(x3007)处设置断点并观察

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5

PC	R1
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	-1
x3007	-2
x3007	-3
x3007	-4
x3007	-5

If no ones, then branch to HALT never occurs!

This is called an “infinite loop.”

Must change algorithm to either

(a) check for special case (R2=0), or

(b) exit loop if R1 < 0.

调试: 总结

跟踪程序看具体发生了什么事情.

- 断点, 单步执行

跟踪程序时,要去观察程序运行时真实发生的情况，而不是你想象的要发生什么

- 在10个数求和的程序中, 很容易就可能忽视实际装入的地址是**x3107** 而不是 **x3100.**

尽量用所有可能的输入数据去测试程序.

- 在 **Examples 3**和 **4**中, 程序的输入数据可能有很多种组合形式或形态
- 确定测试到所有极端的情况 **(**都为**1,** 都为**0, ...).**

作业

Ex 6.4, 6.5, 6.7, 6.8

Ex 6.11 写代码并在仿真器中仿真

Ex 6.12 写代码并在仿真器中仿真

Ex 6.14, 6.15, 6.16, 6.17, 6.19

Ex 6.13