

高级语言程序设计

课程设计报告

学 院 班 级

姓 名 学 号

成 绩 指导老师

年 月 日

**课程设计答辩记录表**

|  |  |
| --- | --- |
| **课程设计题目及要求** | |
| **基于软件技术的简易模拟计算机**  **要求：**学习机器语言编程并编写几个机器语言程序，我们用基于软件的模拟技术建立一台计算机，并在这台计算机上执行所编写的机器语言程序。  建立一台称为Simpletron的计算机。这是一台很简单的机器，但在后面你回发现它的功能同样很强大。Simpletron只能运行用它可以理解的语言编写的程序，也就是Simpletron机器语言，简称为SML。  Simpletron含有一个累加器（一种特殊的寄存器，Simpletron在使用某个信息进行计算或对它进行各方面的检查时先把它存放在这里）。Simpletron中的所有信息都是按“字”（Word）来处理的，每个字都是一个带符号的四位十进制数，如+3364、+0007、-0001等等。Simpletron中100个字的内存，这些字是按其位置序号00单元、01单元、…、99单元来引用的。  在运行一个SML程序之前，必须先把程序装入内存，每个SML程序的第一条指令总是放在00单元处。  用SML编写的每一条指令都占用Simpletron内存的一个字，因此指令是带符号的四位十进制数。我们假定SML指令的符号总是正，但数字的符号可正可负。Simpletron内存中的每个单元可能含有一条指令，也可能是程序使用的一个数据值，还可能是一块未使用的内存区（未被定义）。每条SML指令的前两位数是操作码，表示要被执行的操作。SML的操作码见附录B。  每条SML指令的后两位是操作数，表示被操作的字所在内存单元的地址。现在我们来考虑几个简单的SML程序。  范例程序1  内存单元 数 指令  00 +1007 （Read A）  01 +1008 （Read B）  02 +2007 （Load A）  03 +3008 （Add A）  04 +2109 （Store C）  05 +1109 （Write C）  06 +4300 （Halt）  07 +0000 （Variable A）  08 +0000 （Variable B）  09 +0000 （Result C）  这个SML程序从键盘上读取两个数，计算并打印出它们的和。指令+1007从键盘读取1个数并把它存在内存单元07中（已被初始化为0），然后+1008读取下一个数并把它存在内存单元08中。指令+2007把读取的第一个数存在累加器中，加法指令+3008把第二个数与累加器中的数相加，所有SML算术运算指令都把计算结果存在累加器中存储指令+2109把计算结果存放在内存单元09中，写指令+1109从中取出结果并以带符号的四位十进制数形式打印出该结果。停止指令+4300终止程序的执行。  用一个含有100个元素的数组memory模拟Simpletron的内存。运行是首先将程序装入内存，并提示“Program loading complete， Program execution begins”表示SML程序亿载入内存，接下来Simpletron执行这个SML程序，像C语言一样，除非通过控制转移到程序中的其它部分，否则该程序从内存单元00中的指令开始按顺序向下执行。  用结构体变量proccessor模拟Simpletron的处理器，成员accumulator代表累加器的寄存器，成员instructionCounter跟踪正在执行的指令内存单元，成员operationCode表示当前正在执行的操作，即指令字左边的两位数字，成员operand表示当前指令所操作的内存单元，也就是正在执行的指令字的右边两位数字。不要直接从内存中执行指令，而是把下一条要执行的指令从内存中取出来放在成员instructionRegister中，执行时再将其左边两位数字放到operationCode中，右边两位数字放在operand中。  当Simpletron开始执行时，个寄存器按如下方式初始化：  accumulator +0000  instructionCounter 00  instructionRegister +0000  operationCode 00  operand 00  在一个指令执行周期中，instructionCounter告诉我们下一条要执行的指令所在的内存单元；我们可以使用如下的C语句从数组中取出该内存单元中的内容；  proccessor. instructionRegister=memory[proccessor. instructionCounter];  用如下语句从指令寄存器中取出操作码和操作数；  proccessor.operationCode= proccessor. instructionRegister/100;  proccessor. operand= proccessor. instructionRegister%100;  接下来Simpletron必须确定操作码为读、写或加载等操作，可以用switch语句来区分SML的12种操作。  然后在打印出各寄存器的名字和内容、以及内存的全部内容。这种打印输出通常称为“计算机的转储”（computer dump）。  完成了第一条指令的模拟，下面的任务就是准备让Simpletron执行下一条指令，因为刚才执行的第一条指令不是控制转移指令，那么只需要将指令计数寄存器的值加1就行了；  proccessor. instructionCounter++;  控制转移指令的模拟；如“如果累加器为0则转移”可模拟为；  If (proccessor. Accumulator==0) proccessor. instructionCounter= proccessor. Operand;  这样就实现了Simpletron模拟器指令执行的过程；  由键盘输入范例1的SML程序，执行并在屏幕上显示逐条指令执行过程中的“计算机转储”信息。转储格式范例如下：  简易模拟计算机转储范例  将范例程序2（见附录B中）编写在文本文件fl2.txt中，Simpletron运行时从文件中读取程序再执行，并最终将执行过程的“计算机转储”信息保存在另一个文本文件中。  程序运行时，有键盘读入文本文件名，再打开指定的文件执行完成以下任务的SML程序：  1）用标记控制的循环读取10个正数，计算并打印出它们的和；  2）用计数器控制的循环读取7个数，其中既有正数，也有负数，计算并打印出它们的平均值。  3）读取一组数，判断并打印出最大的数。读取的第一个数表示要处理的数的个数。  此外，该模拟器应该能够检测到各种错误，如程序加载阶段，用户键入的SML语言的程序应该符合语法要求，且必须在-9999到+9999之间。如果出现输入错误，提示用户重新键入数据，直到数据正确为止。如在程序执行阶段，你的模拟器应该可以检测各类型严重的错误，如被0除，执行非法操作码、累加器溢出等，这些严重错误应该能够打印输出，并输出Simpletron内存的所有数据，以助用户定位程序的错误。 | |
| **完成情况：** | |
| 基本数据结构设计 |  |
| 算法设计及流程图 |  |
| 输入输出设计 |  |
| 代码规范及注释 |  |
| 数据测试及调试 |  |
| 代码理解与说明 |  |
| 评语 |  |
| 综合评分 |  |

**目 录**

[一、功能模块结构 1](#_Toc25142)

[二、数据结构设计及用法说明 3](#_Toc8965)

[三、程序结构（流程图） 5](#_Toc22821)

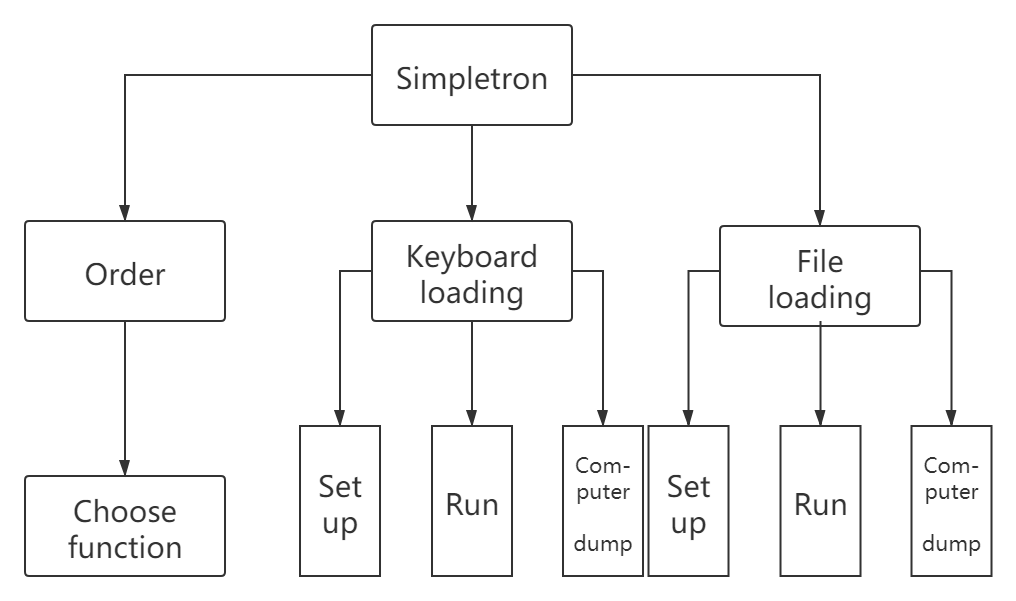
[四、各模块的功能 9](#_Toc14026)

[五、试验结果（包括输入数据和输出结果） 10](#_Toc8384)

[六、体会 19](#_Toc4448)

[七、参考文献 20](#_Toc20069)

## 一、功能模块结构

简易模拟计算机（Simpletron）由两大功能模块组成：键盘载入执行，文件载入执行。载入执行包括从对应的输入端载入相应的SML语言，执行相应的SML语言（输入/输出操作，加载/存储操作，算术运算，控制转移操作），计算机的转储。具体如下图所示：

## 二、数据结构设计及用法说明

### 1.寄存器结构

struct CPU{

int accumulator = +0000; //累加器

int instructionCounter = 00; //指令计数器

int instructionRegister = +0000; //指令寄存器

int operationCode = 00; //操作码

int operand = 00; //操作数

}proccessor;

Simpletron的灵魂之一，用结构定义的一个非常简易的寄存器，麻雀虽小五脏俱全。

Accumulator（累加器）：执行加载/存储操作和算术运算时，数据暂时储存和进行算术运算的场所。

StructionCounter（指令计数器）：将要执行指令的内存位置，指令寄存器通过指令计数器可获得接下来需要执行的语句。

StructionRegister（指令寄存器）：储存当前执行的语句。

OperationCode（操作码）：将指令寄存器中的语句处理过后得到的操作码，代表接下来将要执行的指令。

Operand（操作数）：将指令寄存器中的语句处理过后得到的操作数，代表接下来执行指令操作的对应的内存位置。

### 2.SML语言

Simpletron的灵魂之一，完整的处理器程序还需要有完整的由SML语言编写的SML程序来运行，通过编写可完成简单的标记循环，计数循环，比较大小，算数运算等基本操作。

操作码 含义

输入/输出操作

READ 10 把一个字从终端读入指定的内存单元

WRITE 11 把指定内存单元中的字写入终端

加载/存储操作

LOAD 20 把指定内存单元中的字加载到累加器

STORE 21 把累加器中的字存储在指定内存单元中

算术运算

ADD 30 把指定内存单元中的字与累加器中的字相加，结果保存在累加器中

SUBTRACT 31把累加器中的字与指定内存单元中的字相减，结果保存在累加器中

DEVIDE 32 把累加器中的字除指定内存单元中的字，结果保存在累加器中

MULTIPLY 33把累加器中的字与指定内存单元中的字相乘，结果保存在累加器中

控制转移操作

BRANCH 40 转移到指定的内存单元

BRANCHNEG 41 若累加器为负，转移到指定的内存单元

BRANCHZERO 42 若累加器为0，转移到指定的内存单元

HALT 43 停止，即程序的任务完成

### 3．内存

Simpletron的内存由一个大小为100的数组来实现，SML语句和数据均在里面存储，根据提供的SML程序对相应的内存做出各种相应的操作。

## 三、程序结构（流程图）

### 1．主函数流程图

### 2．模式选择流程图

### 3．键盘载入流程图

### 4．文件载入流程图

### 5.运行SML程序流程图

### 6．计算机的转储（键盘载入的转储和文件载入的转储类似）

## 四、各模块的功能

1.Choose\_Order\_Sign：输入并选择模式

2.SetUp1：从键盘输入SML程序并载入到内存中

3.SetUp2：从文件输入SML程序并载入到内存中

4.Run：执行载入内存的SML程序

READ 把一个字从终端读入指定的内存单元

WRITE 把指定内存单元中的字写入终端

LOAD 把指定内存单元中的字加载到累加器

STORE 把累加器中的字存储在指定内存单元中

ADD 把指定内存单元中的字与累加器中的字相加，结果保存在累加器中

SUBTRACT 把累加器中的字与指定内存单元中的字相减，结果保存在累加器中

DEVIDE 把累加器中的字除指定内存单元中的字，结果保存在累加器中MULTIPLY 把累加器中的字与指定内存单元中的字相乘，结果保存在累加器中

BRANCH 转移到指定的内存单元

BRANCHNEG 若累加器为负，转移到指定的内存单元

BRANCHZERO 若累加器为0，转移到指定的内存单元

HALT 停止，即程序的任务完成

5.Computer\_Dump1：将计算机的转储打印到终端

6.Computer\_Dump2：将计算机的转储打印到文件里

## 五、试验结果（包括输入数据和输出结果）

### 1.标记循环的正数加法运算

输入数据：

//模式选择

2

//打开文件

ADD.TXT

//文件载入的SML程序（输入负数结束）

/\*

00 +1009

01 +2009

02 +4107

03 +2010

04 +3009

05 +2110

06 +4000

07 +1110

08 +4300

\*/

//输入的测试数据

1

2

3

4

5

6

7

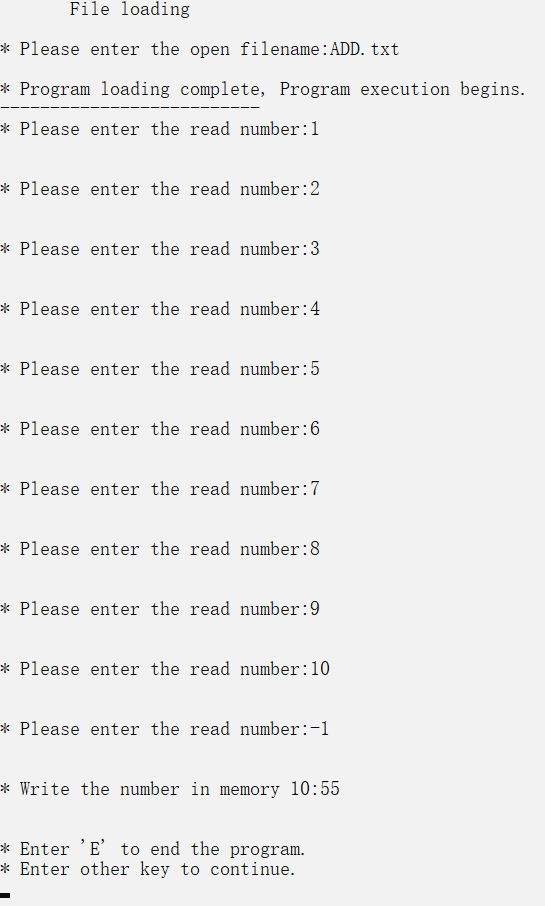
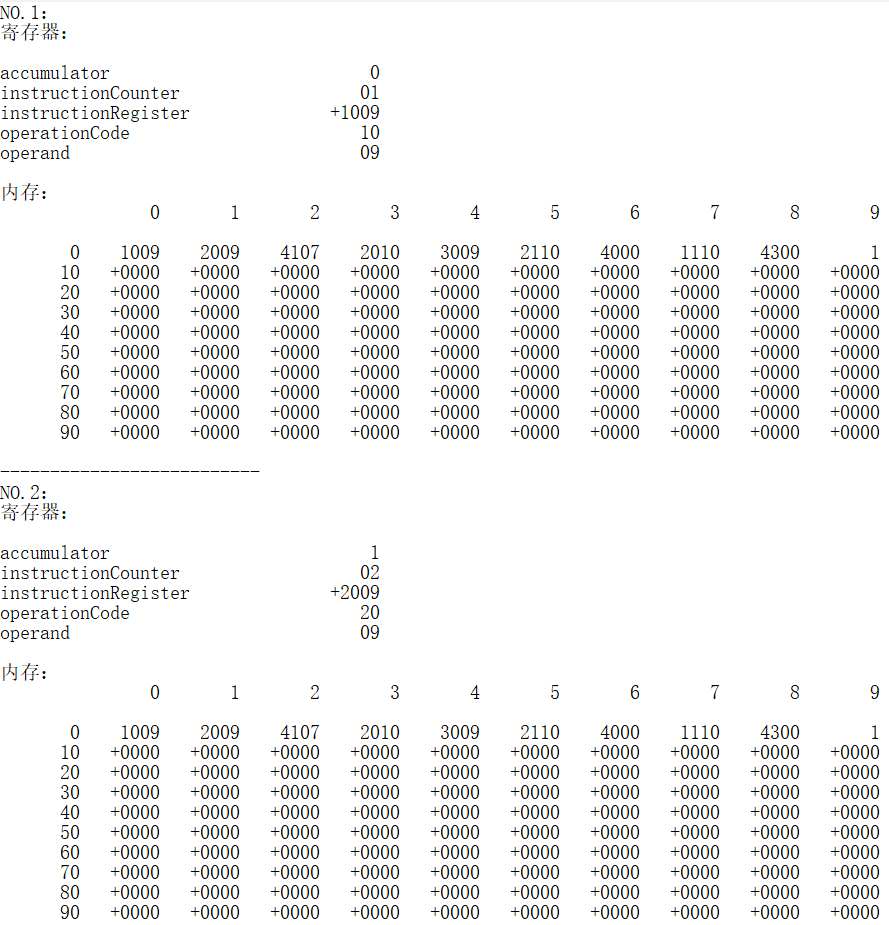
8

9

10

-1

测试结果：



### 2.计数循环求平均数

输入数据：

//模式选择

2

//打开文件

AVENGER.TXT

//文件载入的SML程序

/\*

00 +1099

01 +1098

02 +2097

03 +3199

04 +4213

05 +2096

06 +1096

07 +3096

08 +2196

09 +2097

10 +3098

11 +2197

12 +4003

13 +2096

14 +3299

15 +2196

16 +1196

17 +4300

\*/

//输入的测试数据

7//首先输入循环次数

1//计次循环每次循环加的“1”

1

2

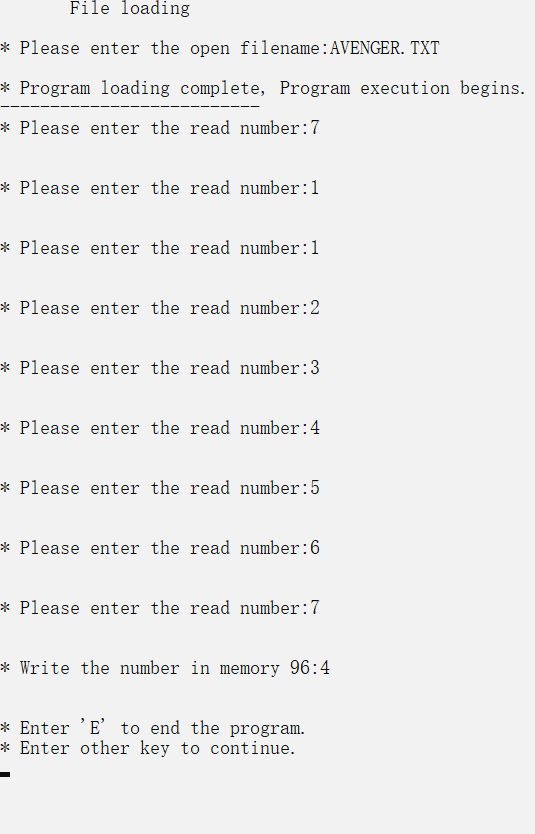
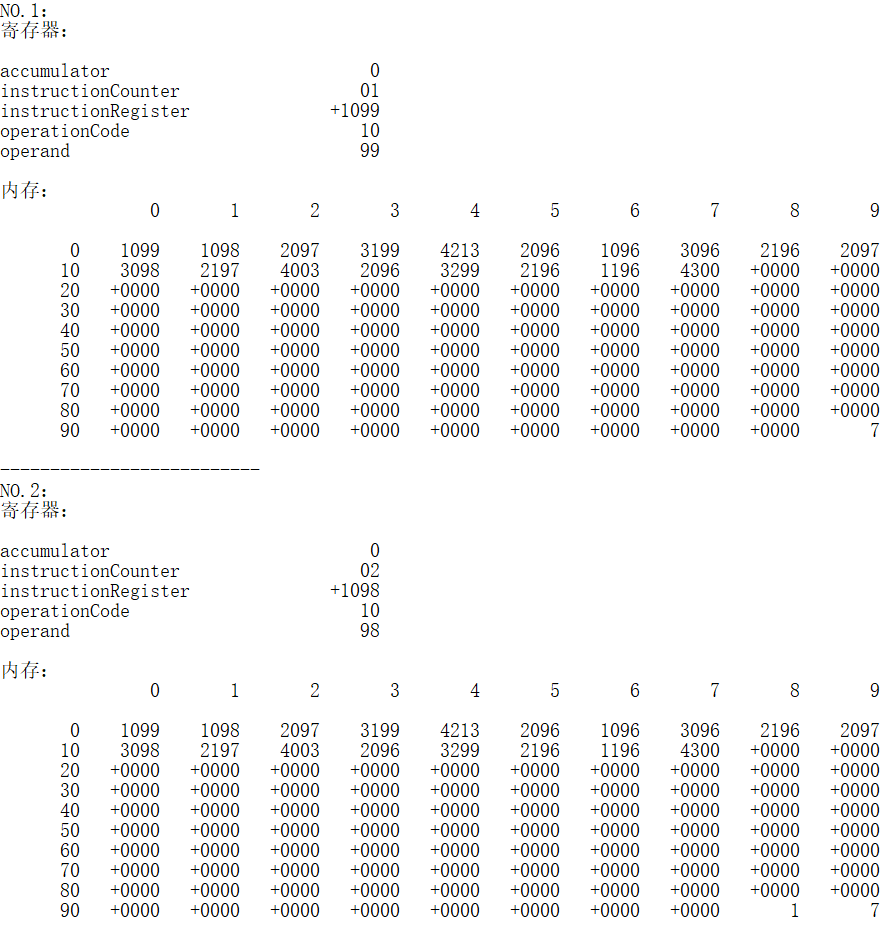
3

4

5

6

7

测试结果：

### 3.计数循环比较大小

输入数据：

//模式选择

2

//打开文件

MAX.TXT

//文件载入的SML程序

/\*

00 +1099

01 +1098

02 +2097

03 +3199

04 +4216

05 +2096

06 +1095

07 +3195

08 +4110

09 +4012

10 +2095

11 +2196

12 +2097

13 +3098

14 +2197

15 +4003

16 +2096

17 +1196

18 +4300

\*/

//输入的测试数据

7//首先输入循环次数

1//计次循环每次循环加的“1”

5

6

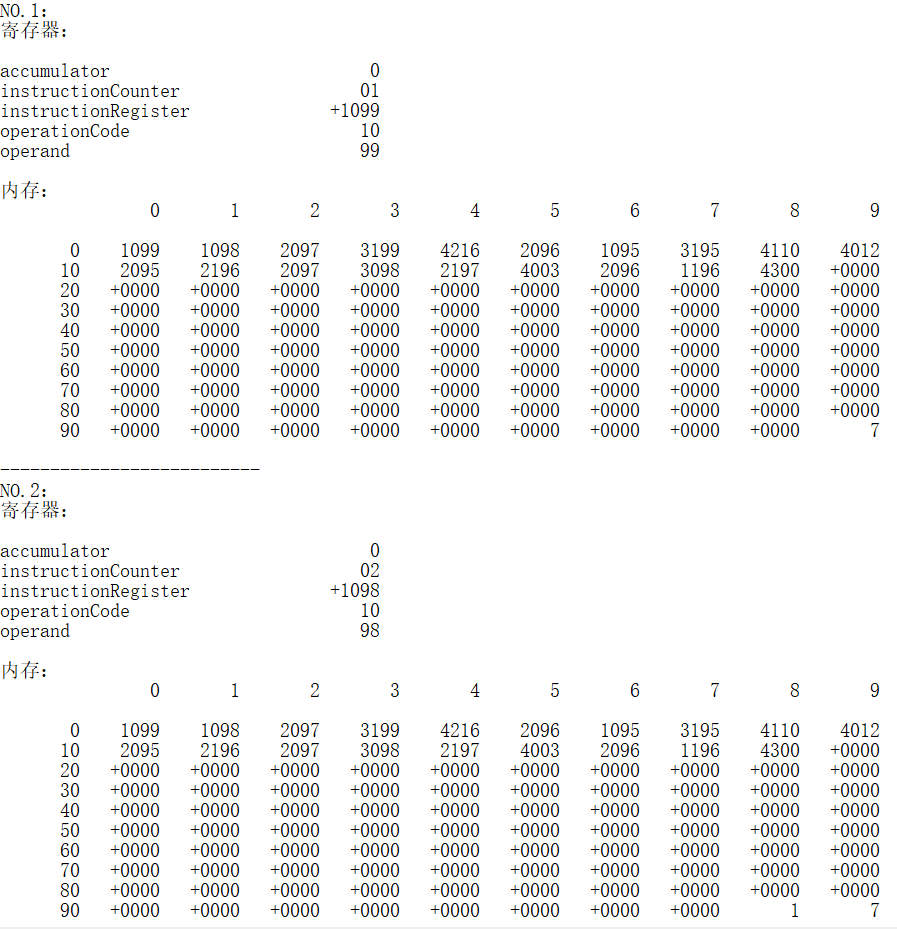
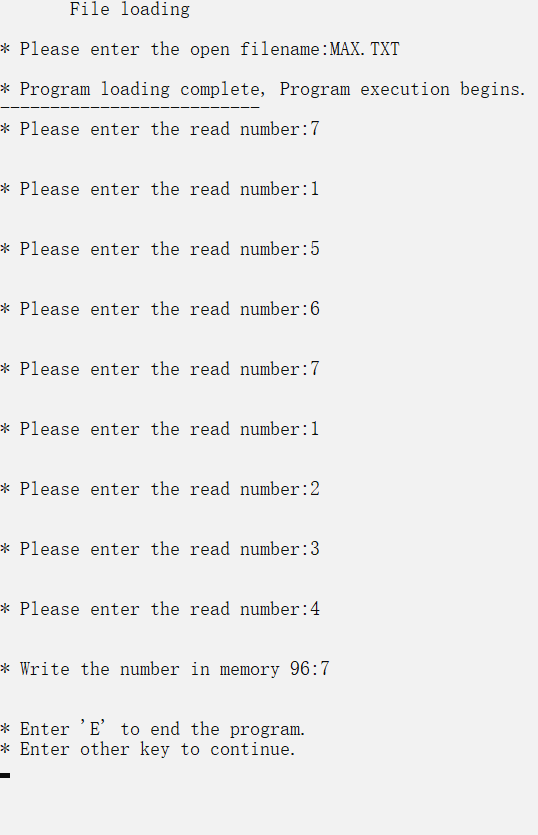
7

1

2

3

4

测试结果：

### 4.界面

### 5.错误性检查

## 六、体 会

最开始并不是想做这个课设，因为它那又臭又长的题目令人望而生畏。等到开始做课设的时候，对于本来没选定的课题还没有想法，时间又还有剩余，无聊之余看起来了这个课设的题目。这不看不要紧，一看竟然就入迷了。渐渐明白了题目想要我做的事，理解完了就觉得有成就感，并且觉得题目也很有意思，于是就转换了自己的课设题目。

这个课设其实并没有想象中的那么艰难，唯一比较绕的地方就是异常状态和SML语句执行函数的细节问题。大部分都是在模拟，甚至涉及到数据结构的地方也很少（也有可能是因为自己没有去用到的原因）。在程序编写快完成的时候就突然觉得其实没做什么只是做了一个题目看上去很厉害的课设，并没有什么实质性的意义。

然而当课设程序写完后我开始编写SML程序时，我才发现自己编写的Simpletron其实并没有那么简单。一串串SML语句实现的SML程序就好像我在自己在C语言编译器上敲的代码形成的程序一样，那种创造力，是普通课设所没有的。好像递归一样，你只是编写，剩下的交给电脑来做的那种感觉。又一瞬间，感觉自己像是体会到了计算机发展的源头与过程，感受到了其实计算机，就是在已有的东西基础上不断再创造而得到的中间产物。这种再创造在当前这个时代，还在不断地进行，再创造产生的中间产物会带来什么，会改变什么，人类无法提前预知，就像计算机诞生之前，人类根本不知道它对人类会带来这样的价值。

课设很有意思，虽然很累，但是还是很有意思。

## 七、参考文献

**[1]** 《C语言程序设计》何钦铭，颜晖主编。 高等教育出版社