

**课 程 实 验 报 告**

**课程名称： 计算机系统基础实验**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师： 李海波**

**实验时段： 2023年9月27日~11月15日**

**实验地点： 东九楼A310**

**原创性声明**

  本人郑重声明：本报告的内容由本人独立完成，有关观点、方法、数据和文献等的引用已经在文中指出。除文中已经注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品或成果，不存在剽窃、抄袭行为。

特此声明！

学生签名：

报告日期：

实验报告成绩评定：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 实验完成质量（70%），报告撰写质量（30%），每次满分20分。 |  |  |  |  |  |  |
| 合计（100分） |  | | | | | |

备注：实验完成质量从实验目的达成程度，设计方案、实验方法步骤、实验记录与结果分析论述清楚等方面评价；报告撰写质量从撰写规范、完整、通顺、详实等方面评价。

指导教师签字：

                    日期：

**目录**

[1 汇编语言编程基础*（这之后内容只参考写法与格式，内容可改）* 3](#_Toc11073)

[1.1 实验内容 3](#_Toc18852)

[1.2 任务1.1实验过程 *（这里展示“验证性、研究性”任务的写法）* 3](#_Toc1177)

[1.2.1 实验方法说明 *（描述你认为需要注意的关键做法和环节）* 3](#_Toc10445)

[1.2.2 实验记录与分析 *（记录你认为不应缺少的信息，重要现象及其分析）* 3](#_Toc13993)

[1.3 任务1.2的实验过程 *（这里展示“设计型”任务的写法）* 4](#_Toc28828)

[1.3.1 设计思想及存储单元分配 *（描述算法、程序框架、功能模块、接口等）* 4](#_Toc24411)

[1.3.2 流程图*（后续类似的设计任务可以不用画流程图）* 4](#_Toc10184)

[1.3.3 实验步骤说明 5](#_Toc4407)

[1.3.4 实验记录与分析 6](#_Toc19127)

[1.4 小结 7](#_Toc17155)

[2 程序优化 7](#_Toc6441)

[2.1 实验内容 7](#_Toc4424)

[2.2 任务3.1实验过程*（设计型、综合性任务）* 7](#_Toc21030)

[。。。。。。。。 7](#_Toc2313)

[3 二进制炸弹破解 8](#_Toc22616)

[3.1 实验内容 8](#_Toc11571)

[3.2 任务4.1实验过程*（研究型任务）* 8](#_Toc19003)

[。。。。。。。。 8](#_Toc13454)

[4 模块化程序设计 8](#_Toc13510)

[4.1 实验内容 8](#_Toc11319)

[4.2 任务2.1实验过程 *（设计型、综合型任务）* 8](#_Toc31877)

[4.2.1 实验方法说明 8](#_Toc19180)

[4.2.2 实验记录与分析 8](#_Toc14383)

[4.3 小结 8](#_Toc25521)

[5 中断处理 8](#_Toc20797)

[5.1 实验内容 8](#_Toc6802)

[5.2 任务5.1实验过程 *（研究性任务）* 8](#_Toc8732)

[。。。。。。。。 8](#_Toc21650)

[6 Linux和鲲鹏环境编程 9](#_Toc18144)

[6.1 实验内容 9](#_Toc25615)

[6.2 任务5.1实验过程 *（研究性任务）* 9](#_Toc15531)

[。。。。。。。。 9](#_Toc15633)

[参考文献 10](#_Toc3358)

# 汇编语言编程基础

## 实验内容

本次实验的主要目的与要求有以下几点，所有的任务都会围绕这几点进行，希望大家事后检查自己是否达到这些目的与要求。

1. 掌握汇编源程序编辑工具、汇编程序、连接程序、调试工具的使用；
2. 理解数、符号、寻址方式等在计算机内的表现形式；
3. 理解指令执行与标志位改变之间的关系；
4. 熟悉分支、循环程序的结构及控制方法，掌握分支、循环程序的调试方法；
5. 加深对转移指令及一些常用的汇编指令的理解。

## 任务1.1实验过程

### 实验记录与分析

(1)理论结果：

标志位AH=0xa03，SF=0，OF=1，CF=1，ZF=0

标志位AH=0x203，SF=0，OF=0，CF=1，ZF=0

标志位AH=0x206，SF=0，OF=0，CF=0，ZF=0

（2）观测结果：

如图所示，理论结果与实验观测结果一致。







A、B为有符号数与无符号数时的标志位特征分别如下两图所示。



有符号

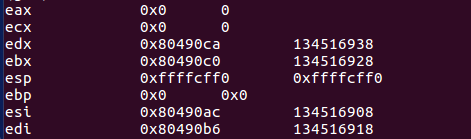


无符号

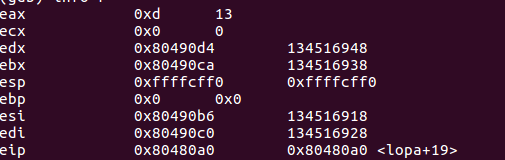
## 任务1.2的实验过程

### 实验记录与分析

“mov $10, %ecx”和“mov $1, %eax”之前的EBX, EBP, ESI, EDI 中保存的结果如下图所示。

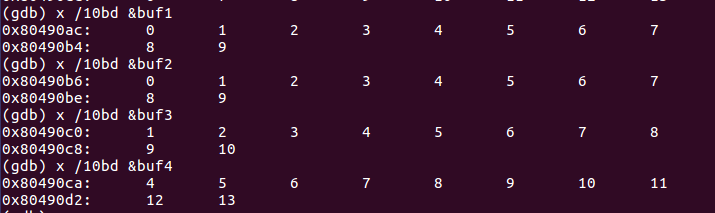


执行“mov $10, %ecx”前寄存器状态



执行“mov $1, %eax”前寄存器状态

程序执行到退出之前数据段开始40个字节的内容如下图所示：



40个字节的内容

可以看到，程序执行结果与设想结果基本一致。

## 任务1.3的实验过程

## 1.4.1 采用变址寻址方式的代码

.section .data

buf1: .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

buf2: .fill 10, 1, 0

buf3: .fill 10, 1, 0

buf4: .fill 10, 1, 0

.section .text

.global \_start

\_start:

mov $buf1, %esi

mov $10, %ecx

lopa: mov (%esi), %al

mov %al, 10(%esi)

inc %al

mov %al, 20(%esi)

add $3, %al

mov %al, 30(%esi)

inc %esi

dec %ecx

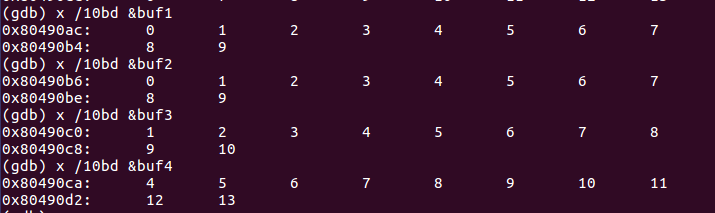
jnz lopa

mov $1, %eax

movl $0, %ebx

int $0x80

## 1.4.2 观测结果



寻址方式改变后的40个存储单元中的内容

## 1.4.3 反汇编起始位置与效果

在观察连续存放的二进制串在反汇编成汇编语言语句时，从不同字节位置开始反汇编会得到不同结果，因为指令解析是从指令指针所指向的地址开始的。

## 任务1.4的实验过程

## 1.5.1 设计思想及存储单元分配

存储单元分配

状态信息存储数组：sdmid

LOWF、MIDF、HIGHF存储区：各使用 40 个字节存储空间,最后以 0 作结；

寄存器分配

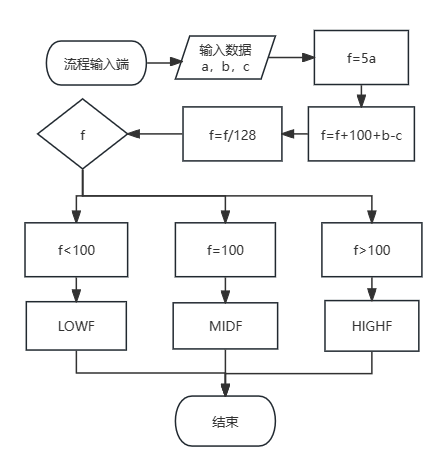
EDI、EBX、EDX：用于存放三个存储区的初始地址。

ESI：用于存放sdmid数组的起始地址。

EAX：临时寄存器。

ECX：计数寄存器，用来遍历simid数组中的元素。

## 1.5.2 程序框架与程序源代码



程序框架

.section .data

LOWF: .fill 10,4,0 # 存储低于100的数据

MIDF: .fill 10,4,0 # 存储等于100的数据

HIGHF: .fill 10,4,0 # 存储高于100的数据

sdmid: .long 256809, -1023, 1065

.long 6786, 67, 87

.long 2000, 2800, 100

.section .text

.global \_start

\_start:

mov $3, %ecx # 设置组数，这里假设有10组数据

lea sdmid, %esi # 加载数据的地址

mov $LOWF, %edi # 设置初始的存储区

mov $MIDF, %ebx

mov $HIGHF, %edx

process\_data:

movl 0(%esi), %eax # 加载a的值到寄存器

# 计算f = (5a + b - c + 100) / 128

imull $5, %eax # 5a

addl 4(%esi), %eax # 5a + b

subl 8(%esi), %eax # 5a + b - c

addl $100, %eax # 5a + b - c + 100

sarl $7, %eax # (5a + b - c + 100) / 128 0x804916c

# 判断f的值，根据条件复制到不同的存储区

cmp $100, %eax

jl copy\_to\_LOWF

je copy\_to\_MIDF

jg copy\_to\_HIGHF

copy\_to\_LOWF:

movl 0(%esi), %eax

add %eax, (%edi)

movl 4(%esi), %eax

add %eax, 4(%edi)

movl 8(%esi), %eax

add %eax, 8(%edi)

add $12, %edi

jmp end\_of\_loop

copy\_to\_MIDF:

movl 0(%esi), %eax

add %eax, (%ebx)

movl 4(%esi), %eax

add %eax, 4(%ebx)

movl 8(%esi), %eax

add %eax, 8(%ebx)

add $12, %ebx

jmp end\_of\_loop

copy\_to\_HIGHF:

movl 0(%esi), %eax

add %eax, (%edx)

movl 4(%esi), %eax

add %eax, 4(%edx)

movl 8(%esi), %eax

add %eax, 8(%edx)

add $12, %edx

jmp end\_of\_loop

end\_of\_loop:

add $12, %esi # 移动到下一组数据

dec %ecx # 减小循环计数器

jnz process\_data # 继续处理下一组数据

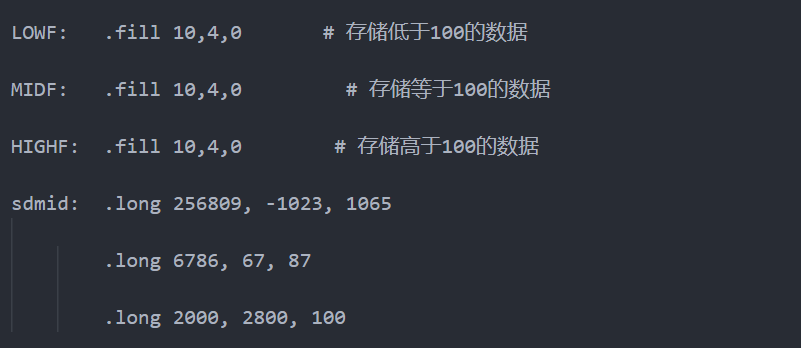
# 退出程序

mov $1, %eax

int $0x80

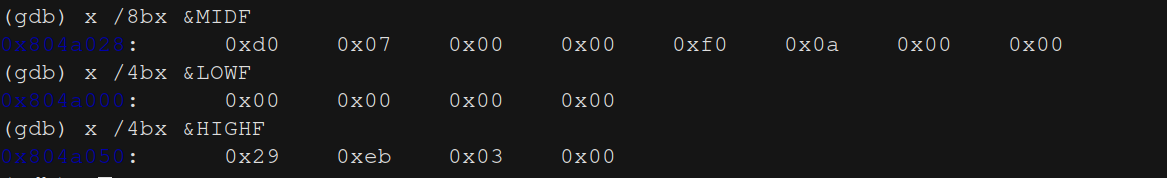
## 1.5.3 实验记录与分析

如下图所示，状态信息存储数组中存储信息如下。预测，会有两组数据存储在MIDF区，一组数据存储在HIGHF区域。



Simid数组预存信息

如下图所示，实验观测结果与理论预测结果相符。



各个存储区存储状态信息的状态

## 1.6 小结

通过实验一的四个任务，我了解了 C 语言与汇编语言的联系。我还掌握了 Linux 操作系统的部分基础指令，以及寄存器间接寻址与变址寻址的转换。实验让我充分熟悉汇编源程序的组成以及各个指令的使用，学会使用汇编语言实现存储空间的分配，函数的调用以及分支、循环的使用。

# 程序优化

## 实验内容

1. 了解程序计时的方法以及运行环境对程序执行情况的影响。
2. 深刻理解 CPU 执行指令的过程，不同特点的编程技巧和指令序列组合对程序长度及执行效率的影响，掌握代码优化的基本方法。

## 任务2.1实验过程

优化工作：包括代码长度的优化和执行效率的优化等等（本次以执行效率/性能的优化为主）。

优化环节：

（1）指令层面：用等价语句改变相关操作，对循环语句中的代码长度进行变；

（2）运行环境层面：

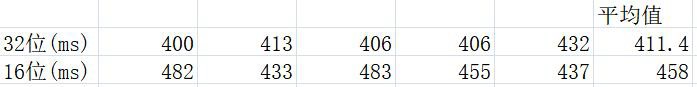
软件环境：分别在 VS2019 下的调试环境和非调试环境运行；

硬件环境：分别在 Core i5-8250U 环境下和 AMD Ryzen R5 2500U 环境下运行。

### 实验记录与分析

指令层面：

（1）由于统计的时间在一定范围内波动，故采用测量 5 次取平均的方式进行时间测试。发现，使用 16 位寄存器时程序运行时间比 32 位寄存器时要长，运行效率不如 32 位寄存器。分析可知，直接使用 32 位寄存器时计算机 CPU 中无需进行多余的操作，而使用 16 位时，则需进行如扩展等的操作造成时间的浪费。并且，32 位寄存器能够存储更多的数据，故使用 32 位寄存器即可。



使用 32 位寄存器与 16 位寄存器的差别

（2）使用更加高效的指令，如将乘法指令转换为移位指令。

由于程序未使用多层循环，因此没有在循环语句上进行更多改动。

运行环境层面：

（1）软件环境。

在上述优化的基础上，在 VS2019 下的调试环境和非调试环境运行进行对比。

 左图为调试环境，右图为非调试环境

发现，在调试环境和非调试环境下运行程序时间接近。故优化程度不大，但通过多次运行可以发现，非调试环境下的运行时间比较稳定，且都保持在 300ms 左右；而调试环境下的运行时间波动幅度较大。

（2）硬件环境。

 在上述优化基础上，分别在 Core i5-8250U 环境下和 AMD Ryzen R5 2500U 环境下运行对比。

左图为 Core i5-8250U，右图为AMD Ryzen R5 2500U

测试结果分析得知，硬件环境对程序运行效率影响很大。故在测试程序运行时，要考虑多个环境中的运行效率，尽可能提升用户体验舒适度。

## 小结

### 主要收获

我深入了解到了堆栈传参的用途以及实现方法，并且还对 C 语言和汇编语言的类比有了更为深入的了解。在第一次实验中，采取的是寄存器存储流程值，因而会导致寄存器使用的麻烦，如不够用、使用时修改数值的情况时有发生，而采用函数构建与堆栈法保护寄存器时则不会出现上述情况，大大减轻的记忆难度和编程难度。

而在程序优化环节，则主要考虑在指令层面进行优化。首先对寄存器的运用进行比对，发现 32 位寄存器优于 16 位寄存器，这与 CPU 的寄存器为 32 位有关，故在后续程序设计时，使用 32 位寄存器较优。而在指令删改过程中，发现指令的删减与选取能够提升不少运行效率，尤其是选取高效的指令与删减无效的指令。在往后的程序设计中，要更加灵活地采用多种优化方法。

总的来说，通过此次实验，让我了解到对程序的优化，不仅要考虑程序的算法实现层面，指令层面的优化也不可忽视。在超量运算过程中，这些不注意的指令层面可能会造成许多资源的浪费。另外，不同的 CPU 的运算效率不同，在生产程序时也应考虑在不同 CPU 环境下的运行效率差异。从而扩大用户面，提高用户体验舒适度。

# 二进制炸弹破解

## 实验内容

### 实验概述

实验目的：

1. 熟悉动态与静态反汇编工具；

2. 熟悉程序的机器级表示，掌握逆向工程的原理与技能；

3. 完成执行程序的调试，提升对计算机系统的理解与分析能力。

实验要求：

1. 熟练使用 gdb 调试器和 objdump；

2. 单步跟踪调试每一阶段的机器代码；

3. 理解汇编语言代码的行为或作用；

4. “推断”拆除炸弹所需的目标字符串。

5. 在各阶段的开始代码前和引爆炸弹函数前设置断点，便于调试。

实验安排：

1. 实验语言：C语言， AT&T汇编语言

2. 实验环境：32位Linux

主要内容：

一个“Binary Bombs”（二进制炸弹，简称炸弹）是一个 Linux 可执行 C 程序，包含phase1~phase6 共6个阶段。

炸弹运行各阶段要求输入一个字符串，若输入符合程序预期，该阶段炸弹被 “拆除”，否则“爆炸” 。

每个炸弹阶段考察机器级语言程序不同方面，难度递增阶段 1：字符串比较

阶段 2：循环

阶段 3：条件/分支：含 switch 语句阶段 4：递归调用和栈

阶段 5：指针

阶段 6：链表/指针/结构

隐藏阶段，第 4 阶段之后附加特定字符串后出现

## 任务3.1实验过程

## 3.2.1 阶段1字符串比较

任务描述：输入一个字符串，并对字符串进行比较，若为正确的字符串则进入下一阶段，若字符串错误则炸弹爆炸。

实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的字符串信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_1 函数内，出现“string\_not\_equal”函数的调用，且调用前传递两个参数，如图 3.1 所示。

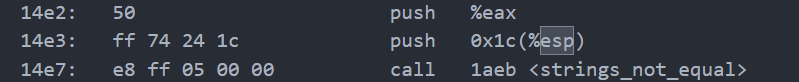


图 3.1 调用string\_not\_equal 函数前的准备阶段

发现第一个入栈的参数为一个立即数，推测为密码存储单元，而后续入栈的参数为在 main 函数里进入 phase\_1 函数准备阶段的参数， 且前面紧跟着 read\_line 函数，显而易见的是文本内容的存储单元首地址。

（3）：使用 gdb 指令，查找 0x56558144 存放的字符串数据，得到如图 3.2 所示的结果。



图3.2

## 3.2.2 阶段2 循环

任务描述：输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_2 函数内，出现“read\_six\_numbers”函数的调用，可知要输入 6 个数字的密码，设为x1, x2, x3, x4, x5, x6。

（3）： 根据图 3.3 所示信息，可以得知正确的密码中 4(%esp)=0，8（%esp）

=1，故后续 lea 0x4(%esp),%ebx 中，为对输入数字的判断，即 x1=0，x2=1。

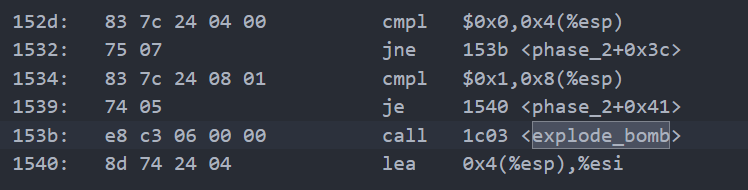


图 3.3

（4）： 由图3.4，根据循环体内的信息，可知 x1 + x2 = x3 。由此可推测是一个Fibonacci sequence。则有：x1 = 0，x2 = 1, x3 = 1, x4 = 2, x5 = 3, x6 = 5。

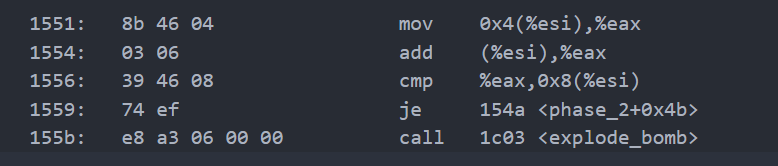


图 3.4

## 3.2.3 阶段3 条件/分支

任务描述：输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的字符串信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_3 函数内，出现“ scanf ”函数的调用，如图 3.5 所示，其中通过 gdb 查找存储单元 0x565581a2 的信息，为“%d %c %d”，可知要输入 2 个整型数字与 1 个字符。



图 3.5

（3）： 对两个数和一个字符进行判断，通过多种比较逐一确定它们的内容。如图 3.6 所示，若第一个数大于7就会跳转到炸弹爆炸处，所以第一个数小于等于7。

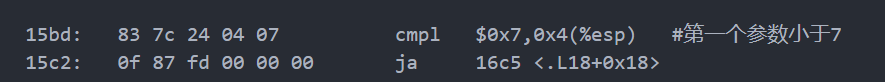


图 3.6

（4）：选取第一个数为 1，经调试发现程序经 switch 语句跳转到如图 3.7所示所在代码段。结合最后图 3.8 所示的返回语句，可推断出1 p 996是一组密码。

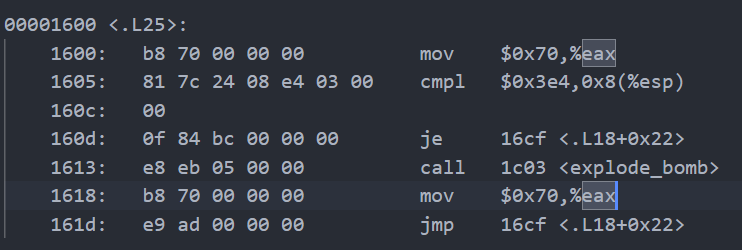


图 3.7

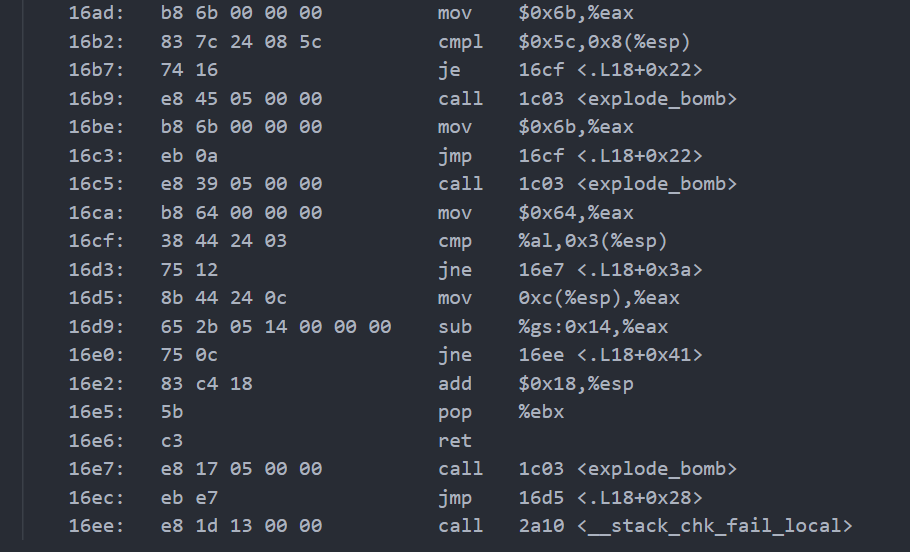


图 3.8

## 3.2.4 阶段4 递归调用和栈

任务描述：输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

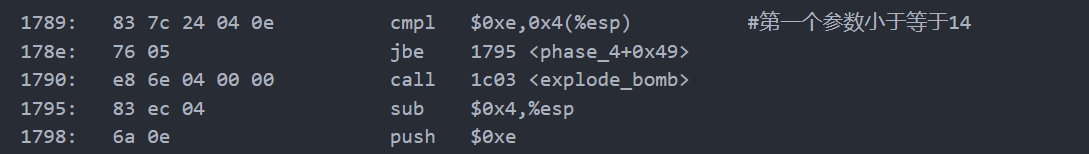
实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_4 函数内，出现 scanf 函数，通过 gdb 查找存储单元信息，为“%d %d”，可知要输入 2 个整型数字，设为x1, x2。

（3）：如图 3.9 所示，x1 必须小于等于 14。

 图 3.9

进入函数 func4 前的准备阶段传参，如图 3.10 所示，为x1，0，14。

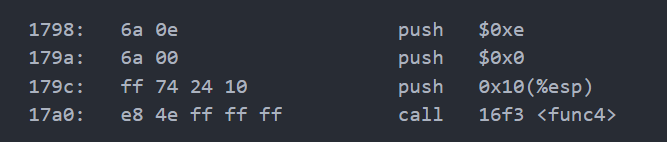


图 3.10

（4）：观察如图 3.11 所示的信息，可以发现返回参数 eax 与 37 进行比较，且 x2 也与 37 进行比较，当返回值 eax 和 x2 不为 37 时，炸弹爆炸。故 x2==37，func4（x1，0，14）==37。

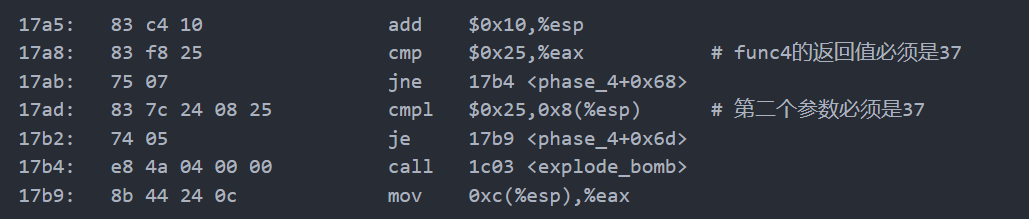


图 3.11 传回参数的比较

（5）：进入 func4 函数体观察，它的功能中包含二分查找思想。根据最后多次尝试的结果得出当 x1 = 10 时满足输出结果，所以最终x1 = 10, x2 = 37。

## 3.2.5 阶段5 指针

任务描述：输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_5 函数内，出现 “ string\_length ” 函数，可知要输入一个字符串。

（3）：由图 3.12 知，字符串的长度为 6。字符串中的每一个字符转化为整型数字，最终作为变址指针指向内存中的数据。由图 2.12 知，6次指针所指内存中的数相加所得和为48。

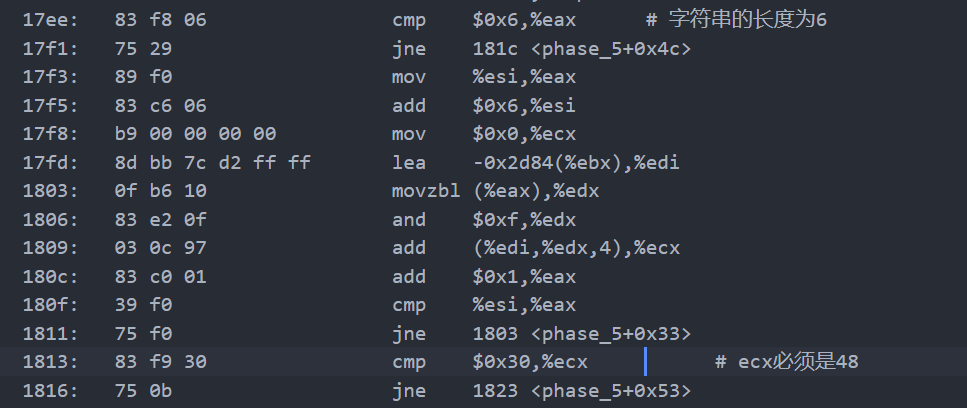


图 3.12

（4）：由图 3.13 可知指针所指向内存内容。可以得出121212是一个答案，即 (0x02 + 0x0a) \* 0x3 = 0x30。

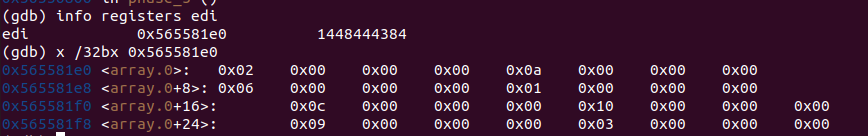


图 3.13 存储单元内容

## 3.2.6 阶段6 链表/指针/结构

任务描述：输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

实验设计：采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的信息。

实验过程及结果：

（1）：对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）：查看反汇编中函数调用过程前后的关键信息，在 phase\_6 函数内，出现 “ read\_six\_numbers ” 函数，可知要输入6个数字，设为x1, x2, x3, x4, x5, x6。

（3）：如图 3.14 所示，6个数为无符号数。该段代码代表有两层循环，外层循环用来判断每个数是否都小于等于6，大于6则炸弹爆炸；内层循环（红色方框标注）判断6个数是否相同，若存在则炸弹爆炸。

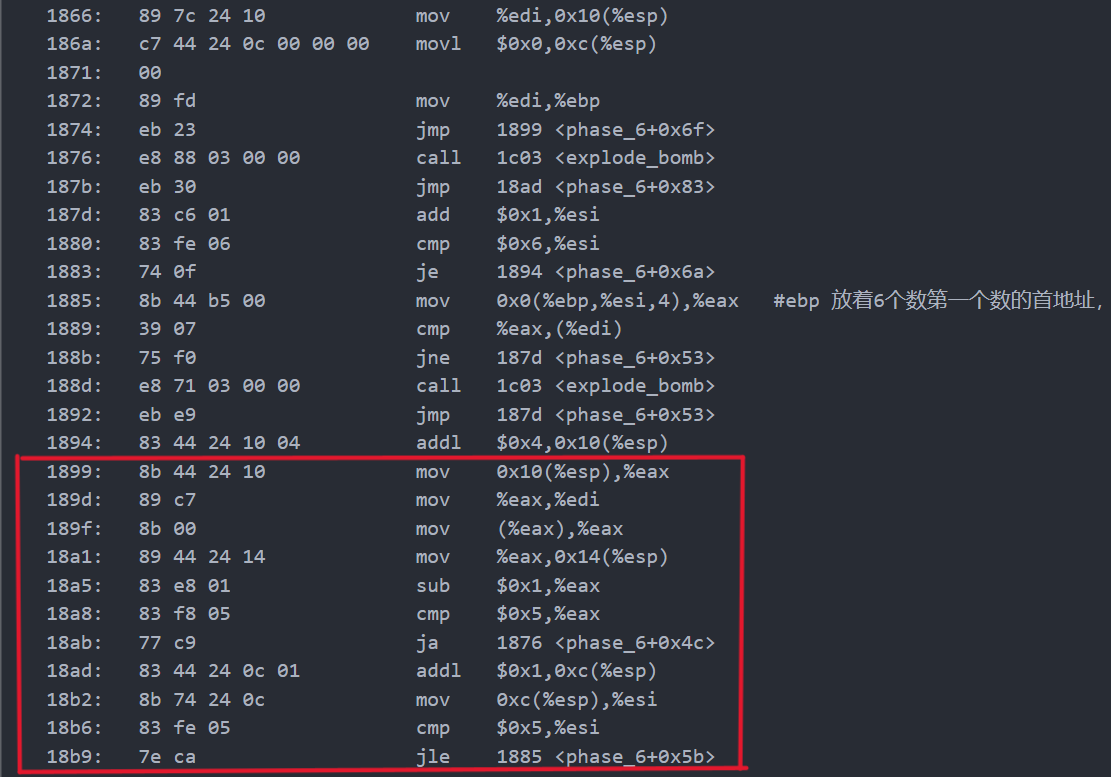
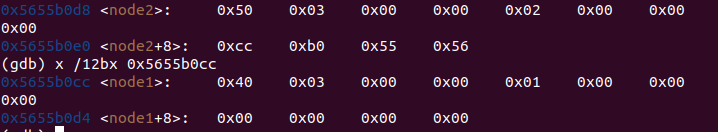
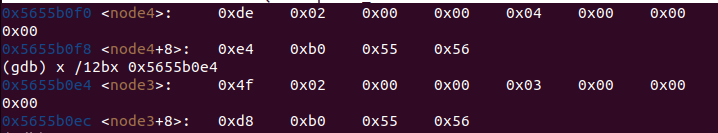


图 3.14

（4）：根据后续代码推断出存在不连续的多个数据块，推测是链表。链表的首地址为 0x5655b0d8，使用 gdb 指令，查看链表存储单元内容，包含 3 个元素，最后一个元素为指向下一结构体的指针，如图 3.15 所示。





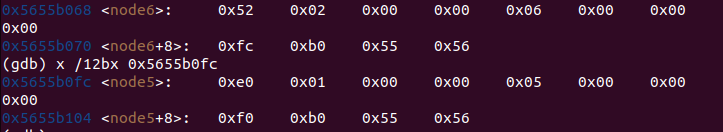


图 3.15

（5）：后续程序对保存的地址进行处理，实现链表的重排，使链表升序排列。则最后不难得出密码为5、6、3、1、4、2.。如图 3.16 所示。

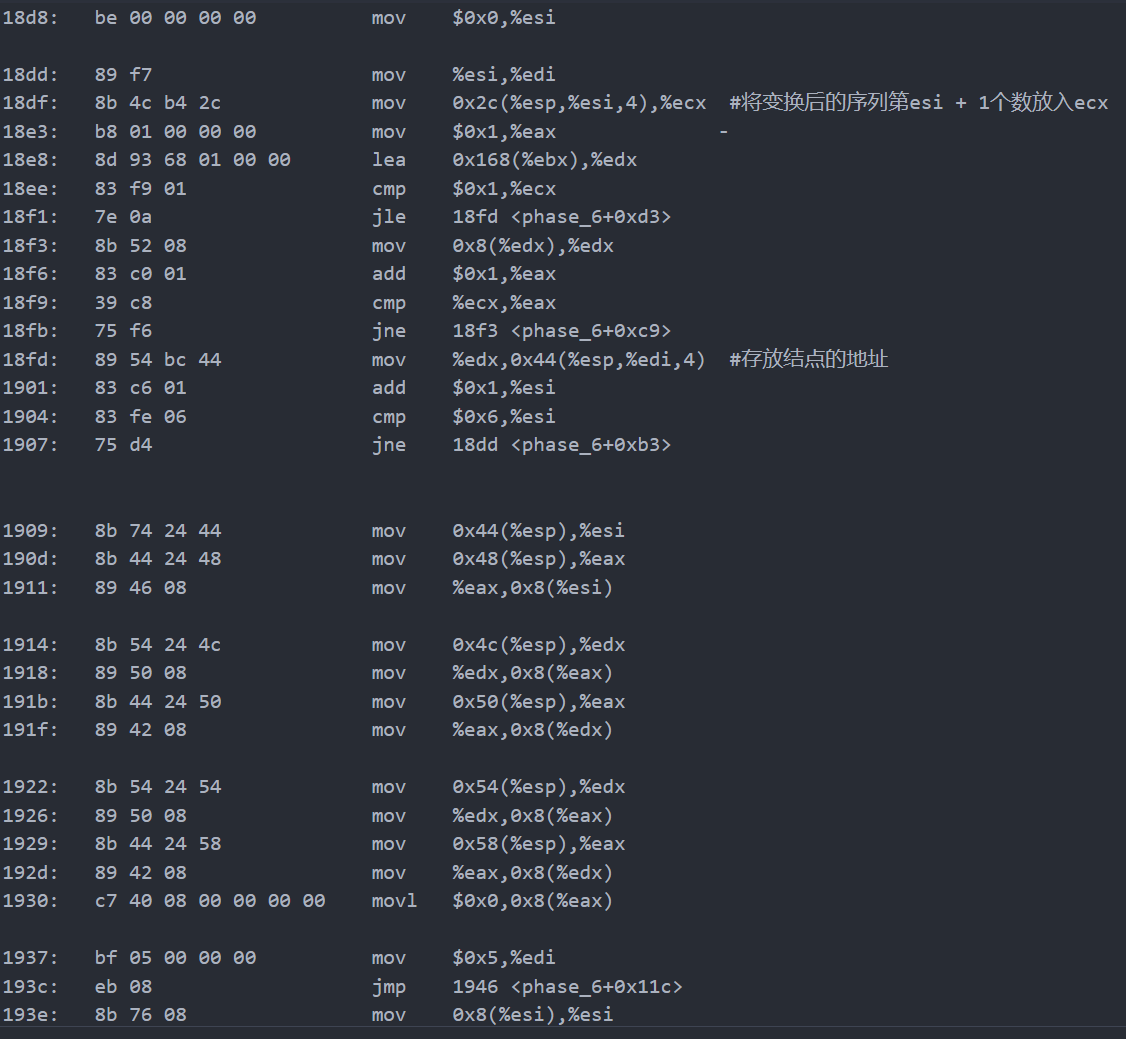


图 3.16

## 3.2.7 阶段7 隐藏阶段

实验分析：这里重点分析后面的二叉链表结构，隐藏阶段需要在第四个阶段输入密码后，再输入字符串“DrEvil”才会触发隐藏阶段。由 图3.17 所示的反汇编代码，这里进行了一个二叉搜索树中的查找操作。经 gdb 指令查看相关存储单元中的内容，最终可以确定该二叉排序树为{38，8，50，6，22，45，109，1，7，20，36，40，47，99，1001}。而程序要求必须查找二叉排序树最右分支端点才不会使炸弹爆炸，故隐藏阶段的密码即为二叉排序树中的最大数，即1001。

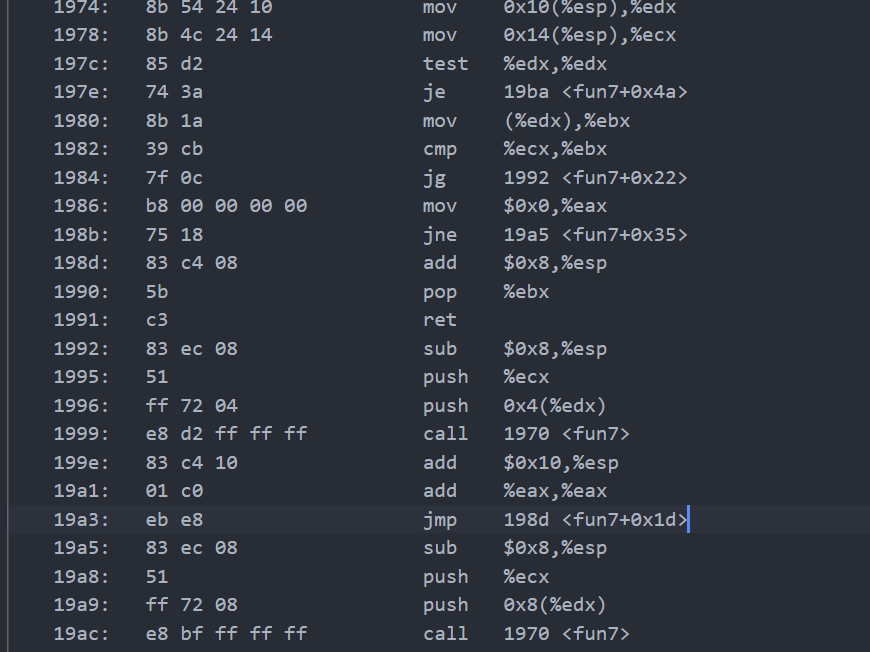


图 3.17

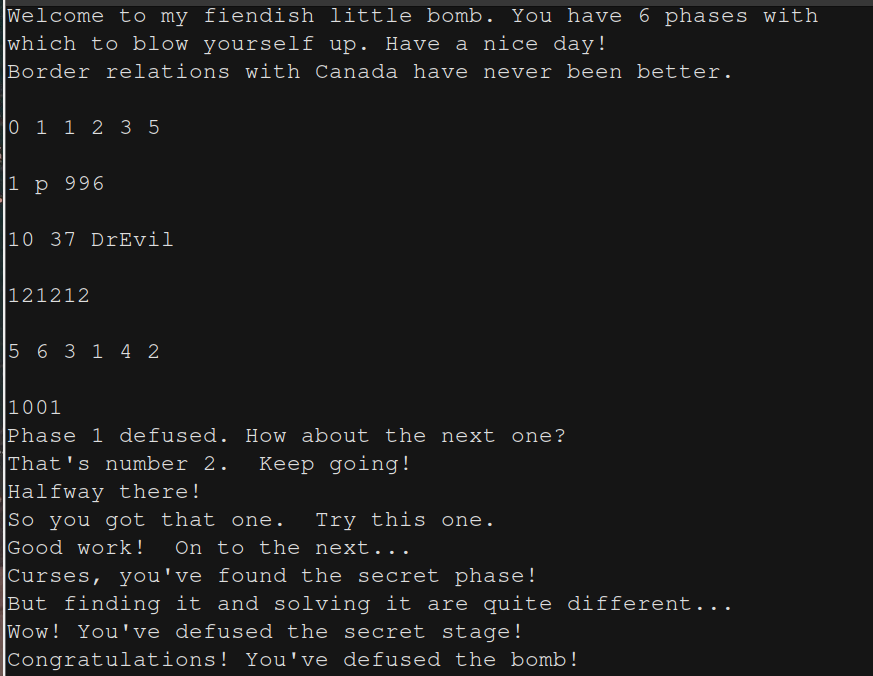


图 3.18 通关图

# 模块化程序设计

## 实验目的与要求

1. 掌握子程序设计的方法与技巧，熟悉子程序的参数传递方法和调用原理；
2. 掌握模块化程序的设计方法;
3. 掌握汇编语言程序与C语言程序混合编程的方法；
4. 理解模块之间的信息传递与组装的基本方法；

## 任务4.1实验过程

### 实验方法说明

### 1. 在对实验1.4进行改进的基础上，进行程序调整以满足首要条件：首先，提醒用户输入其用户名（将用户名定义为个人姓名的拼音）和密码。若输入不正确，显示错误信息并要求用户重新输入，最多允许三次错误机会。当用户三次都未正确输入时，程序将退出。如果用户名和密码均正确，则继续进行后续处理。

2. 在MIDF中实现R键继续Q键返回

3. 在 x86 汇编语言中，信息在主程序与子程序之间的传递主要通过寄存器和内存完成。

参数和返回值的传递方式包括寄存器传递和堆栈传递。在调用子程序之前，主程序将参数存储在不同的寄存器中，如 eax、ebx、ecx、edx 等。子程序通过读取这些寄存器中的值来获取参数。返回值通常被保存在 eax 寄存器中，以便供主程序读取。

此外，参数也可以通过堆栈传递。主程序将参数推入堆栈，而子程序则从堆栈中弹出这些参数。类似地，返回值也可以通过堆栈传递，并被放置在堆栈的顶端。

关于返回地址，在调用子程序时，主程序的下一条指令地址（即返回地址）被推入堆栈，以确保子程序执行完毕后能够正确返回到主程序的位置。ret 指令会弹出并跳转到这个返回地址。

至于局部变量，通常它们会被存储在堆栈空间。在子程序调用时，会在堆栈上为局部变量分配空间。这些局部变量的地址通常是相对于当前堆栈帧的偏移量。通过这些调整，实现了相同的功能，但采用了一种不同的表达方式。

4. 研究在模块之间实现参数传递的方式时，我们在处理汇编和C语言时可以采用如下技巧：

全局符号的定义和使用：我们可以在整个程序范围内使用全局符号，使其在不同模块中可见。在先前的代码片段中，像LOWF、MIDF、HIGHF这样的全局变量可以在不同的模块中引用，因为它们在整个程序中都是可见的。

外部符号的引用：当一个模块需要调用另一个模块中定义的全局变量或函数时，可以通过引用外部符号来实现。在当前任务的代码中，C语言程序通过声明extern void calculateF(int a, int b, int c, int \*result) {}和extern void copyData(int \*source, int \*destination, int size) {}，以引用包含在汇编代码中的子程序。

确保符号名和类型的一致性：在引用时，必须确保符号名和类型是一致的。如果存在不一致的符号名或类型，可能会导致链接错误或者运行时错误的发生。

在实际操作中，必须确保模块间接口的一致性，以确保参数传递和函数调用的正确性。在链接阶段，链接器将这些模块合并为一个可执行文件。一旦发现符号名或类型不匹配，链接阶段可能会失败。因此，确保在链接阶段一切一致性都得到了维护是非常重要的。

### 实验记录与分析

完成准备工作构建程序后，运行程序后得到如图4.1所示的结果。

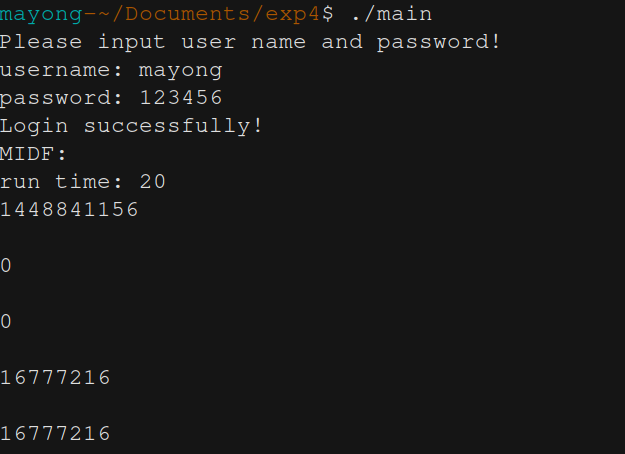


图 4.1 运行成功截图

## 小结

此次实验完成的任务整体围绕模块编程展开。通过对多模块的编程实验，主要掌握汇编语言多模块与混合编程的设计思想，以及模块联系、调用的具体实现。

任务 4.1 中，主要掌握了多模块编程思想，其次则是掌握了对函数调用和宏指令调用的设计与使用。将主流程与子程序设计分别放置在两个不同的模块中，可以使模块信息传递具有条理性，不至于多模块处理混乱。在多模块编程中，主要思想则是主模块简单化，子模块为主模块提供辅助函数支撑。另外，在进行模块间联立时，函数的调用规范也不容小觑。在主流程中调用函数可使用 call 指令与 invoke 指令，且使用堆栈法传参与保护现场可以极大程度减少记忆与编程压力。在对 invoke 指令进行反汇编时，发现其为 call 指令的延伸，即包含 call 指令以及进栈，退栈操作。所以，在单独进行 call 指令操作时，要注意退栈处理以避免多次调用栈溢出。

同时，call 指令的执行包含 2 个过程，即跳转至存储函数内存读取函数指向数据段，再由读取到的内存地址跳转至子程序定义数据段的位置。此过程中，函数集合内存起到了中间过渡的作用，这也是模块间能够联立的重要原因之一。对 ret 进行反汇编发现，如果存在传参，则在 ret 指令时会进行 leave 指令操作，其作用为将 ESP 返回至 EBP 指向位置，并赋值 EBP 前存放值，进行 EBP， ESP 归位操作，进而返回 EIP，回到主流程断点位置。在进行外部符号使用时，仅仅传递变量地址值，在不同的子模块中的使用则取决于外部符号说明时的类型，但不允许变量名称存在差异。

另外，宏指令的调用相比与函数调用更为简单，在反汇编时会自动进行语句扩展，并将对应参数直接代入其中，不需要进行语句跳转等操作。但需注意的是，宏指令便于处理简单频繁语句列，而对于某功能的定义则使用函数定义最佳，原因则是使用函数定义更符合编程思想，在进行管理时也更为方便。

实验发现，在不同语言模块进行多模块联立时，外部符号的传递仅仅是传递变量地址值，故其类型根据外部符号说明的类型为准。在进行结构数组联立时，则只需保证结构定义的字节段相同即可，在反汇编时会进行对应地址处理，从而访问所需元素。而 C 语言模块调用汇编模块函数时则与汇编语言调用子模块函数无异。

# 中断处理

## 实验内容

## 5.1.1 实验目的与要求

（1）通过观察与验证，理解中断矢量表的概念；

（2）熟悉I/O访问，BIOS功能调用方法；

（3）掌握实方式下中断处理程序的编制与调试方法；

（4）进一步熟悉内存的一些基本操纵技术；

## 5.1.2 实验内容

在DOSBox环境中实现时分秒信息在窗口指定位置的显示。其中，指定位置信息来源于程序中定义的变量的内容；所实现的程序运行后需要驻留退出。

要求能在TD下观察中断矢量表、观察已有的某个中断处理程序的代码、读取CMOS中某个单元内容；能通过某种方式在TD下调试中断处理程序。

## 任务5.1实验过程

## 5.2.1 设计思想与存储单元分配

任务 **5.1** 要求在 DOSBox 环境下实现时分秒信息在窗口指定位置的显示，且实现的程序首次安装完成后需驻留，并避免重复安装。故设计思路分为两大部分：主流程处理程序设计与中断处理程序设计。

主流程处理程序包含 3 个环节：判断是否重复安装中断处理程序，安装中断处理程序，驻留退出。

（1）判断是否重复安装中断处理程序环节：由于第一次程序驻留后无法直接访问驻留内存位置，进而难以获取驻留在内存中的数据。故采用设置标记特征的办法，在数据段末、中断处理程序段前存储 DW 类型标记符号‘#’。在主流程处理程序中，当读取中断信息时，将中断程序地址值减 2 即为标记符号存储内存处，只需判断是否为‘#’则可避免重复安装中断处理程序。即，若为首次安装，则旧中断地址处前 2 个字节的标记不为‘#’，则进行后续安装环节；若为二次及以上安装，则始终为首次安装地址，地址处且前 2 个字节标记‘#’驻留不会消失，从而判断已经首次安装，跳过安装环节。在主程序中加入显示信息 T，若进入安装环节，则显示 T。

（2）安装中断处理程序环节：使用 INT 21H 功能，使用 25H 调用号，将 8 号时钟中断替换为自己编写的新的 08 中断处理程序。

（3）驻留退出环节：使用 INT 21H 功能，使用 31H 调用号，将主流程处理程序外的程序段驻留，即驻留中断处理程序所需的所有信息。退出主流程处理程序。

中断处理程序设计包含 3 个环节：执行原中断程序，获取时间信息，显示。

（1）执行原中断程序环节：在首次安装中断处理程序时，已将原中断处理程序地址保存至内存单元 OLD\_INT 中，故将标志寄存器保护后，直接使用 CALL 指令调用原中断处理程序。由于该内存单元在主流程中已驻留，故无需担心被清除。通过原中断执行次数来实现计数，当原时钟中断执行 18 次后进入后续获取时间及显示环节。

（2）获取时间信息环节：首先重置计数器，再开中断，保护现场。调用读取时间子程序 GET\_TIME，使用端口号 70H，设定访问单元，再读取相应时间，并进行 BCD 码的转换。获取的时间信息分别存储到对应的内存单元。

（3）显示环节：使用 INT 10H，首先获取原光标位置，存储到对应内存单元。再设置显示光标的位置参数，显示存储时间信息的内存单元。显示结束后，还原光标。现场退栈，退出中断处理程序。

## 5.2.2 实验步骤

编写程序，调试通过后运行，在指定位置正确显示时间信息。

运行其他程序，查看中断程序是否驻留正常。

在 TD 下，再次运行该程序，查看是否重复安装中断程序。

在 TD 下观察中断矢量表，观察某个中断处理程序代码。

在 TD 下调试中断处理程序。

## 5.2.2 实验记录与分析

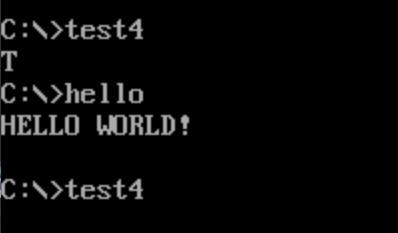
 1.运行程序，首先进行安装中断处理程序，观察结果；运行其他程序，观察结果；再次运行安装程序，观察结果。观察结果如图 5.1 所示。发现，程序运行成功，当第二次及多次运行程序时，未输出安装检验显示 T，表明设置标记特征能够达到避免多次重复安装的目的。

图 5.1

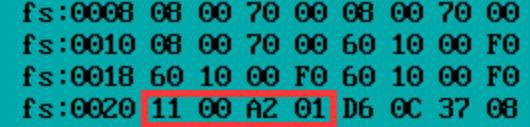
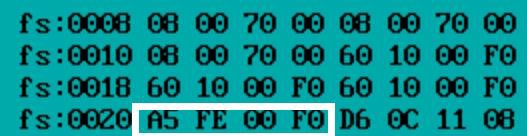
2.观察中断矢量表，在内存单元 0000:0020H 处，储存此次实验变化矢量信息。首次运行安装程序前的该地址内存如图 5.2(a)所示，运行后的该地址内存如图 5.2(b)所示。

图 5.2（a） 图 5.2（b）

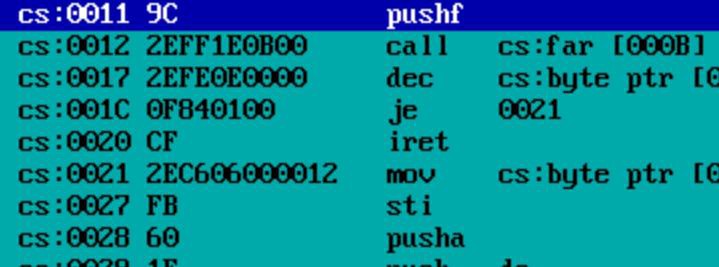
3. 查询第 2 次运行后的中断矢量表查询 08 中断号信息，访问中断处理程序，如图 5.3 所示。

图 5.3 查询中断矢量表访问中断处理程序

表明，该内存信息已驻留成功。且 8 号中断处理程序已替换为该实验的中断处理程序。通过该中断地址，可访问该中断处理程序。

# Linux和鲲鹏环境编程

## 实验内容

## 6.1.1 实验目的

了解ARM + Linux环境下程序设计的特点及配套的开发工具。

并理解80X86和ARM下，不同的“指令集体系结构”的基本特点。

## 6.1.2 实验需要编译执行的程序

（1）“ARM虚拟环境安装说明”文档中“1.4.1”的程序（一个显示Hello World!的汇编语言程序）；

（2）“ARM虚拟环境安装说明”文档中“2.2.1”（一个测试内存拷贝函数的执行时间的C语言与汇编语言混合编程的程序）和“2.2.3”（对前面“2.2.1”程序的优化）的程序。

## 任务6.1实验过程

1.按照教程安装ARM虚拟环境.

2.确定程序是否自带gcc、编译调试、gdb等功能。

3.编译并运行图 6.1的程序

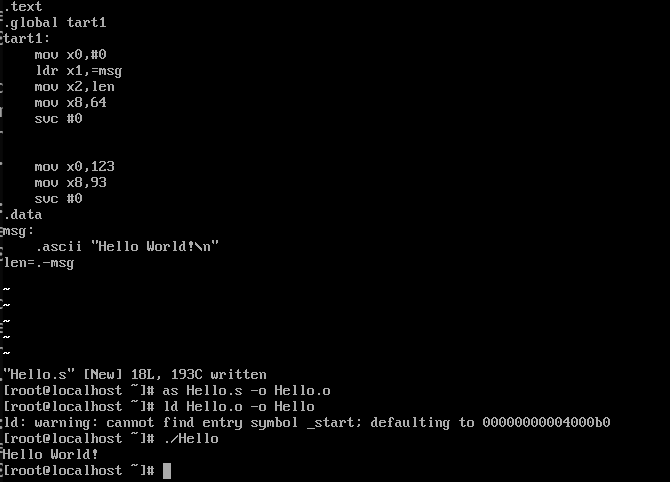


图 6.1

4.创建time.c文件 执行 vi time.c 编写 c 语言计时程序。通过clock\_gettime 函数来计算时间差，从而得出所求代码的执行时间。如图 6.2 所示。

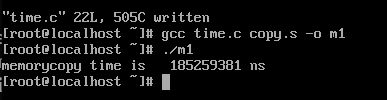


图 6.2

5. 对上述程序进行优化。优化结果如图 6.3 所示。

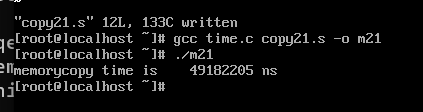


图 6.3 优化结果

## 6.3 小结

通过完成实验，我对整个实验总体流程有了更清晰的认识。按照提供的PDF文档配置实验环境并进行编译测试的过程相对来说并不复杂，但在实践中，我发现了一些挑战，这使我更深入地了解了ARM虚拟环境。

首先，我注意到不同的指令集和编程环境与我之前所学的知识有一些差异。这让我意识到在不同的技术领域和平台上，开发和测试的方法可能会有所不同。通过解决这些差异，我不仅仅是在完成实验，更是在拓宽我的技术视野。

在配置环境的过程中，我遇到了一些新的工具和设置，这些工具和设置不仅仅是为了完成实验任务，还有助于我更全面地理解ARM虚拟环境的工作原理。这种深入了解对于未来的项目和学习都将是非常有价值的。

编译测试阶段也为我提供了一个学习的机会。我发现不同的编程语言和编译器可能会在性能和执行方面产生差异，这让我更注重选择适合特定情境的工具和技术。

总的来说，虽然实验总体并不难，但通过完成实验，我获得了更多的实践经验，加深了对ARM虚拟环境的理解，并为未来的技术探索和项目实施打下了坚实的基础。

# 参考文献

[1]袁春风. 计算机系统基础. 北京：机械工业出版社，2023

[2]王元珍，曹忠升，韩宗芬. 80X86 汇编语言程序设计. 武汉：华中科技大学出版社，2005