

华中科技大学

课程实验报告

课程名称： 计算机系统基础实验

专业班级： 大数据 2201 班

学 号： U202215566

姓 名： 刘师言

指导教师： 李海波

实验时段： 2022 年 9 月 27 日~11 月 15 日

实验地点： 南一楼 808 室

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： 刘师言

报告日期：2023 年 10 月 29 日

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:

汇编语言程序设计实验报告

目录

1 汇编语言编程基础	1
1.1 实验目的和要求	1
1.2 实验内容	1
1.3 实验 1.1 实验过程	2
1.3.1 实验方法说明	2
1.3.2 实验记录与分析	3
1.4 实验 1.2 实验过程	6
1.4.1 实验方法说明	6
1.4.2 实验记录与分析	6
1.5 实验 1.3 实验过程	8
1.5.1 实验方法说明	8
1.5.2 实验记录与分析	8
1.6 实验 1.4 实验过程	11
1.6.1 设计思想及存储单元分配	11
1.6.2 流程图	12
1.6.3 实验步骤说明	13
1.6.4 实验记录与分析	13
1.7 实验小结	17
2 程序优化	18
2.1 实验内容	18
2.2 任务 2.1 实验过程	18
2.2.1 设计思想	18
2.2.2 实验步骤说明	19
2.2.3 实验记录与分析	19
2.3 实验小结	21
3 二进制炸弹破解	23
3.1 实验目的和要求	23
3.2 实验内容	23
3.3 任务 3.1 实验过程	23
3.3.1 实验方法说明	23
3.3.2 实验记录与分析	23

汇编语言程序设计实验报告

2.1 实验小结	38
4 模块化程序设计	40
4.1 实验内容	40
4.2 任务 4.1 实验过程	40
4.2.1 实验方法说明	40
4.2.2 实验记录与分析	41
4.3 实验小结	47
5 中断处理	48
5.1 实验目的和要求	48
5.2 实验内容	48
5.3 任务 5.1 实验过程	48
5.3.1 设计思想	48
5.3.2 流程图	49
5.3.3 源程序	50
5.3.4 实验步骤	51
5.3.5 实验记录与分析	51
5.4 实验小结	52
6 LINUX 和鲲鹏环境编程	54
6.1 实验目的和要求	54
6.2 实验内容	54
6.3 任务 6.1 实验过程	54
6.3.1 实验方法说明	54
6.3.2 实验记录与分析	54
6.4 实验小结	59
参考文献.....	61

汇编语言程序设计实验报告

1 汇编语言编程基础

1.1 实验目的和要求

- (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。

1.2 实验内容

上机实验环境说明：使用 Linux 环境（建议使用 Linux 虚拟机）。源程序编辑工具可以使用 vi（或使用 windows 下的 vscode、记事本等编辑器，然后共享到虚拟机中）；汇编程序使用 as；连接程序使用 ld；（或使用 gcc 进行编译和链接）；调试工具使用 gdb。

任务 1.1 已知 8 位二进制数 x1 和 x2 的值，求[x1]补 + [x2]补，并指出结果的符号，判断是否产生了溢出和进位。

x1 = + 0110011b, x2 = +1011010b

x1 = - 0101001b, x2 = -1011101b

x1 = + 1100101b, x2 = -1011101b

要求：(1) 请事先指出执行指令后(AH)、标志位 SF、OF、CF、ZF 的内容。

(2) 记录上机执行后的结果，与（1）中对应的内容比较。

(3)求差运算中，若将 A、B 视为有符号数，且 A>B，标志位有何特点？

若将 A、B 视为无符号数，且 A>B，标志位又有何特点？

任务 1.2 课堂习题三，第 2 题。

要求：（1）分别记录执行到“mov \$10, %ecx”和“mov \$1, %eax”之前的 EBX,EBP,ESI,EDI 各是多少。

（2）记录程序执行到退出之前数据段开始 40 个字节的内容，指出程序运行结果是否与设想的一致。

(3)在标号 lopa 前加上一段程序，实现新的功能：先显示提示信息“Press any key to begin!”，然后，在按了一个键之后继续执行 lopa 处的程序。

操作提示：先单步执行各个语句，每执行一条语句，都应观察数据段中的内容以及相应寄存器的变化。其次，单步执行循环体两遍且正确理解了循环体语句的含义后，可在“mov \$1, %eax”处设置断点，然后直接执行到断点处，回答(1)和(2)的问题。

汇编语言程序设计实验报告

任务 1.3 课堂习题三，第 3 题。

要求：(1) 内存单元中数据的访问采用变址寻址方式。

(2) 记录程序执行到退出之前数据段开始 40 个字节的内容，检查程序运行结果是否与设想的一致。

(3) 观察并记录机器指令代码在内存中的存放形式，并与反汇编语句及自己编写的源程序语句进行对照，也与任务 2 做对比。（相似语句记录一条即可，重点理解机器码与汇编语句的对应关系，尤其注意操作数寻址方式的形式）。

(4) 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，结果怎样？理解 IP/EIP 指明指令起始位置的重要性。

任务 1.4 设计实现一个数据处理的程序

有一个计算机系统运行状态的监测系统会按照要求收集三个状态信息 a, b, c（均为有符号双字整型数）。假设 n 组状态信息已保存在内存中。对每组的三个数据进行处理模型是 $f = (5a + b - c + 100) / 128$ （最后结果只保留整数部分）。当 $f < 100$ 时，就将该组数据复制到 LOWF 存储区，当 $f = 100$ 时，就将该组数据复制到 MIDF 存储区，当 $f > 100$ 时，就将该组数据复制到 HIGHF 存储区。

每组数据的定义方法可以参考如下：

```
sdmid .fill 9, 1, 0    # 每组数据的流水号（可以从 1 开始编号）
sda   .long 256809     # 状态信息 a
sdb   .long -1023      # 状态信息 b
sdc   .long 1265       # 状态信息 c
sf    .long 0          # 处理结果 f
```

要求：熟悉加减乘除等运算指令，内存拷贝方法，为后续优化做准备（本次实验请按照直观的表达式求解，不必做优化）。先按照不考虑溢出的要求编写程序，再按照考虑溢出的要求修改程序；思考：如果三个状态信息是无符号数，程序需要做什么调整？

1.3 任务 1.1 实验过程

1.3.1 实验方法说明

1. 准备上机实验环境，对实验用到的软件进行安装、运行，通过试用初步了解软件的基本功能、操作等。

2. 在 Linux 环境下的终端中使用 vim 编写汇编程序，实现两个 8 位二进制数的加法，使用 as 编译和 ld 链接，生成二进制可执行文件后使用 gdb 进行单步调试，在执行完两数相加的指令后观察并记录结果的符号和标志位的特点。

3. 编写汇编程序将两数进行求差运算，编译、链接和调试过程同上，先将两数视为有符号数，并单步调试，在执行完两数相减的指令后观察并记录标志位的特点；再将两数视为无符号数，并单步调试，在执行完两数相减的指令后再次观察并记录标志位的特点，比较前后差异。

汇编语言程序设计实验报告

4. 尝试按照自己思考的其他语句及输入格式等进行操作，积累更多的经验。

1.3.2 实验记录与分析

1. 实验环境条件：

Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存；

Linux 下 vim, as, ld, gdb

2. 实验记录与分析：

在 Linux 环境下的终端中使用 vim 编写两个 8 位二进制数相加的汇编程序，先对第一组数值 ($x1 = +0110011b$, $x2 = +1011010b$) 进行实验，其中在表示有符号数时需要使用补码的形式，源码如图 1.1.1 所示。依次使用 as、ld 进行编译和链接后，使用 gdb 打开生成的可执行文件进行单步调试。

```
.section .text
.global _start
_start:
mov $0b00110011, %al
mov $0b01011010, %bl
add %bl, %al
mov $1, %eax
mov $0, %ebx
int $0x80
```

图 1.1.1 两数相加汇编程序

在执行完加法指令后，使用指令 `info r` 查看寄存器和标志位的信息，如图 1.1.2 所示。观察标志位 SF、OF、CF、ZF 的内容，发现 SF 和 OF 的值为 1，CF 和 ZF 的值为 0，说明运算结果符号为负 (SF = 1)，发生了溢出 (OF = 1)，未发生进位 (CF = 0)，且结果不为 0 (ZF = 0)。

汇编语言程序设计实验报告

```
Breakpoint 1, _start () at test.s:4
4      mov $0b00110011, %al
(gdb) n
5      mov $0b01011010, %bl
(gdb)
6      add %bl, %al
(gdb)
7      mov $1, %eax
(gdb) info r
eax            0x8d            141
ecx            0x0            0
edx            0x0            0
ebx            0x5a            90
esp            0xffffd080     0xffffd080
ebp            0x0            0x0
esi            0x0            0
edi            0x0            0
eip            0x8049006       0x8049006 <_start+6>
eflags        0xa86          [ PF SF IF OF ]
cs             0x23            35
ss             0x2b            43
ds             0x2b            43
es             0x2b            43
fs             0x0            0
gs             0x0            0
(gdb) █
```

图 1.1.2 单步调试并查看标志位的内容

由于该程序使用的两个二进制数（ $x1 = +0110011b$ ， $x2 = +1011010b$ ）均为有符号数正数，而两数相加得到的结果却为负数（ $SF = 1$ ），说明运算过程发生了溢出，且为正溢出， $OF = 1$ 即可以验证我们的结论。这是因为两数相加后由于进位导致最高位变为 1，而最高位为有符号数的符号位，符号位为 1 表示该有符号数为负数，因此运算结果为负数，即发生了溢出。

改变汇编程序中二进制数的值，依次对剩下的几组数值进行同样的操作，其中需要注意的是表示有符号数时需使用补码，有符号数负数的补码等于其反码加 1，如对于二进制数 $x1 = -0101001b$ ，该数的补码为 $1101011b$ 。

对第二组数值（ $x1 = -0101001b$ ， $x2 = -1011101b$ ）进行实验时，在使用 `gdb` 进行调试的过程中观察到标志位 `CF` 和 `OF` 的值为 1，`SF` 和 `ZF` 的值为 0（如图 1.1.3 所示），说明了运算结果不为 0 且符号为正，运算过程中发生了进位和溢出。

```
eflags        0xa03          [ CF IF OF ]
```

图 1.1.3 查看标志位的内容

对第三组数值（ $x1 = +1100101b$ ， $x2 = -1011101b$ ）进行实验时，在使用 `gdb` 进行调试的过程中观察到标志位 `CF` 值为 1，`SF`、`OF` 和 `ZF` 的值均为 0（如图 1.1.4 所示），说明了运算结果不为 0 且符号为正，运算过程中发生了进位，但没有发生溢出。

```
eflags        0x203          [ CF IF ]
```

图 1.1.4 查看标志位的内容

接下来对两个二进制数进行求差运算，先将两二进制数视为有符号数，表示方法同上。编写相应的汇编程序，使用 `as` 和 `ld` 分别进行编译和链接，再使用 `gdb` 进行单步调试，在执行完减法指令后查看标志位的内容，观察到第一组数值和第二组数值运算完毕后上述四个标

汇编语言程序设计实验报告

志位值均为 0，分别如图 1.1.5、图 1.6 所示。

eflags 0x206 [PF IF]

图 1.1.5 第一组数值实验后标志位的内容

eflags 0x202 [IF]

图 1.1.6 第二组数值实验后标志位的内容

而第三组数值（ $x1 = +1100101b$ ， $x2 = -1011101b$ ）运算完毕后标志位 CF、SF 和 OF 的值均为 1，ZF 的值为 0（如图 1.1.7 所示），说明运算结果为负数（ $SF = 1$ ），且发生了进位（ $CF = 1$ ）和溢出（ $OF = 1$ ）；

eflags 0xa83 [CF SF IF OF]

图 1.1.7 第三组数值实验后标志位的内容

再将两二进制数视为无符号数，编写相应的汇编程序，注意在表示无符号数时，最高位不再表示该数的符号，而是表示该数的实际数值，因此存放时需要将无符号数存放在位数较少的寄存器（如 al、bl）中，运算时再使用相对应的位数较高的寄存器（如 ax，bx），这样就能保证计算机在运算时将该数作为无符号数来进行操作，而不会将数的最高位作为其符号位。

编写完相应汇编程序（如图 1.1.8 所示）后，使用 as 和 ld 分别进行编译和链接，再使用 gdb 进行单步调试，在执行完减法指令后查看标志位的内容，观察第一组数值运算完毕后上述四个标志位值均为 0（如图 1.1.9 所示），即该组数据实验结果与做有符号数减法时相同，这是因为对于正数而言有符号数和无符号数的表示方式相同。

```
.section .text
.global _start
_start:
mov $0b01011010, %al
mov $0b00110011, %bl
sub %bx, %ax
mov $1, %eax
mov $0, %ebx
int $0x80
```

图 1.1.8 两无符号数相减汇编程序

eflags 0x297 [CF PF AF SF IF]

图 1.1.9 视作无符号数时第一组数值实验后标志位的内容

在对第二组和第三组数值进行时，观察到两组数值运算完毕后四个标志位的值同样均为 0，这是因为无符号数恒为非负数，在令 $A > B$ 进行求差运算时，得到的结果一定也为非负数且不会发生进位或溢出。

3.实验小结：

汇编程序在表示二进制数的值的时候要使用立即数，并写成以 \$0b 开头的形式。在 AT&T 汇编格式下，减法指令 `sub %ebx, %eax` 表示用寄存器 `eax` 中的值减去寄存器 `ebx` 中的值并将结果保存在寄存器 `eax` 中，注意减数和被减数的顺序。

在表示有符号数时，需要使用补码的形式，其中正数的原码最高位即符号位为 0，补码等于其原码；负数的原码符号位为 1，补码等于其反码加 1。而表示无符号数时，则没有原码、反码和补码的概念，因为无符号数的最高位不再表示数的符号，而是表示数的实际数值大小。在编写相应的汇编程序时需要对有符号数和无符号数的表示加以区分。

汇编语言程序设计实验报告

1.4 任务 1.2 实验过程

1.4.1 实验方法说明

1. 使用 `as` 和 `ld` 对现有的汇编程序进行编译和链接，再使用 `gdb` 进行单步调试，观察相应步骤中寄存器的内容和数据段的内容。

2. 使用系统调用将字符串打印在屏幕上，并读取键盘输入，在完成输入后继续执行程序。

1.4.2 实验记录与分析

1. 实验环境条件：

Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存；

Linux 下 `vim`, `as`, `ld`, `gdb`

2. 实验记录与分析：

编译和链接汇编程序后，使用 `gdb` 进行单步调试，在执行到“`mov $10,%ecx`”和“`mov $1,%eax`”之前时，分别使用指令 `info r` 查看寄存器 `ebx`, `ebp`, `esi`, `edi` 的内容并进行记录，如图 1.2.1、图 1.2.2 所示。

```
Breakpoint 1, _start () at test.s:13
13      mov $10,%ecx
(gdb) info r
eax                0x0                0
ecx                0x0                0
edx                0x804a01e           134520862
ebx                0x804a014           134520852
esp                0xffffd0d0         0xffffd0d0
ebp                0x0                0x0
esi                0x804a000           134520832
edi                0x804a00a           134520842
eip                0x8049014           0x8049014 <_start+20>
eflags             0x202              [ IF ]
cs                 0x23               35
ss                 0x2b               43
ds                 0x2b               43
es                 0x2b               43
fs                 0x0                0
gs                 0x0                0
(gdb) █
```

图 1.2.1 查看寄存器的内容

汇编语言程序设计实验报告

```
Breakpoint 2, lopa () at test.s:26
26      mov $1, %eax
(gdb) info r
eax             0xd             13
ecx             0x0             0
edx             0x804a028        134520872
ebx             0x804a01e        134520862
esp             0xffffd0d0       0xffffd0d0
ebp             0x0             0x0
esi             0x804a00a        134520842
edi             0x804a014        134520852
eip             0x804902c        0x804902c <lopa+19>
eflags         0x246            [ PF ZF IF ]
cs              0x23            35
ss              0x2b            43
ds              0x2b            43
es              0x2b            43
fs              0x0             0
gs              0x0             0
(gdb)
```

图 1.2.2 查看寄存器的内容

由汇编程序源码可以看出，程序执行到退出之前数据段开始 40 字节的内容为数组 buf1 到 buf4 经过运算之后各元素的值，其中每个数组包含 10 个元素，每个元素占 1 个字节。通过阅读汇编程序源码，发现程序主要功能是通过循环在 buf1 的基础上为 buf2、buf3 和 buf4 三个数组中的元素赋值，推理得出程序执行到退出之前 buf1 到 buf4 的内容分别为整数 0 到 9、0 到 9、1 到 10 和 4 到 13，接着使用 gdb 对设想结果进行验证。使用 gdb 先在程序末尾处设置断点，然后执行程序，使用指令“x /40bd \$buf1”查看数据段开始 40 个字节的内容（如图 1.2.3 所示），发现程序运行结果与设想一致。

```
(gdb) x /40bd &buf1
0x804a02c:  0      1      2      3      4      5      6      7
0x804a034:  8      9      0      1      2      3      4      5
0x804a03c:  6      7      8      9      1     2      3      4
0x804a044:  5      6      7      8      9     10     4      5
0x804a04c:  6      7      8      9     10    11    12    13
(gdb)
```

图 1.2.3 查看数据段开始 40 个字节的内容

接下来要使汇编程序在屏幕上打印出字符串和从键盘读取输入，上网查阅了有关资料后得知可以通过调用 C 库函数和进行系统调用等方法来实现，这里采用系统调用的方法，在汇编程序中使用中断指令 int \$0x80 进行系统调用，参数分别存入寄存器 eax、ebx、ecx、和 edx 中。其中，eax 中的参数为系统调用号，打印字符串的系统调用号为 4，读取键盘输入的系统调用号为 3；ebx 中的参数表示标准输入或输出；ecx 中的参数为事先在数据段定义的字符串的首地址，用于输入输出；edx 中的参数为字符串的长度。按照系统调用的格式编写汇编程序（如图 1.2.4 所示），经编译和链接后执行，成功在屏幕上打印出字符串和从键盘读取输入，如图 1.2.5 所示。

汇编语言程序设计实验报告

```
.section .data
msg: .ascii "Press any key to begin!\n"
input: .space 20
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:

//print message
movl $4, %eax
movl $1, %ebx
movl $msg, %ecx
movl $24, %edx
int $0x80

//press any key to begin
mov $3, %eax
mov $0, %ebx
mov $input, %ecx
mov $20, %edx
int $0x80
```

图 1.2.4 汇编程序中使用系统调用

```
vboxuser@Linux:~/Desktop/computer system$ ./test
Press any key to begin!
a
vboxuser@Linux:~/Desktop/computer system$
```

图 1.2.5 在屏幕上打印字符串和从键盘读取输入

3.实验小结:

在进行标准输入输出系统功能调用时需注意传入寄存器 `ecx` 的参数为字符串的首地址，在字符串名前要加`&`符号表示取其地址。若只将字符串的值存入寄存器 `ecx` 中，则得不到预期的结果。

1.5 任务 1.3 实验过程

1.5.1 实验方法说明

1. 采用变址寻址方式编写相应汇编程序，使用 `as` 和 `ld` 对汇编程序进行编译和链接，再使用 `gdb` 进行单步调试，观察相应步骤中寄存器的内容和数据段的内容。

2. 观察并记录机器指令代码在内存中的存放形式，并与反汇编语句及自己编写的源程序语句进行对照。

1.5.2 实验记录与分析

1. 实验环境条件:

Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存;

Linux 下 `vim`, `as`, `ld`, `gdb`

汇编语言程序设计实验报告

2. 实验记录与分析:

变址寻址的汇编语句形式如 `buf(%eax)`所示, 其中 `buf` 表示变量的首地址, 寄存器 `eax` 中存储要访问地址的偏移量。内存单元中数据的访问采用变址寻址, 对实验 2 的汇编程序进行改写, 如图 1.3.1 所示。

```
.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 $0, %esi
mov $10, %ecx
lopa: mov buf1(%esi), %al
mov %al, buf2(%esi)
inc %al
mov %al, buf3(%esi)
add $3, %al
mov %al, buf4(%esi)
inc %esi
dec %ecx
jnz lopa
mov $1, %eax
movl $0, %ebx
int $0x80
```

图 1.3.1 采用变址寻址方式的汇编程序

使用 `as` 和 `ld` 对汇编程序进行编译和链接, 再使用 `gdb` 进行调试, 在程序末尾位置设置断点, 观察并记录程序结束前数据段开始 40 个字节的内容 (如图 1.3.2 所示), 发现程序结束前数据段开始 40 个字节的内容与实验 2 采用寄存器间接寻址时数据段的内容一致, 说明程序编写正确。

```
(gdb) x /40bd &buf1
0x804a000: 0      1      2      3      4      5      6      7
0x804a008: 8      9      0      1      2      3      4      5
0x804a010: 6      7      8      9      1     2      3      4
0x804a018: 5      6      7      8      9     10     4      5
0x804a020: 6      7      8      9     10     11     12     13
```

图 1.3.2 采用变址寻址方式时程序结束前数据段内容

在 `gdb` 调试过程中, 使用指令 `disassemble/r` 可以查看反汇编语句及其对应的机器码。在实验二和实验三中分别使用该指令观察寻址赋值过程中机器码在内存中的存放形式, 记录下机器码与汇编语句的对应关系, 如图 1.3.3 和图 1.3.4 所示。

汇编语言程序设计实验报告

```
(gdb) disassemble/r lopa
Dump of assembler code for function lopa:
0x08049045 <+0>: 8a 06 mov (%esi),%al
0x08049047 <+2>: 88 07 mov %al,(%edi)
0x08049049 <+4>: fe c0 inc %al
0x0804904b <+6>: 88 03 mov %al,(%ebx)
0x0804904d <+8>: 04 03 add $0x3,%al
0x0804904f <+10>: 88 02 mov %al,(%edx)
0x08049051 <+12>: 46 inc %esi
0x08049052 <+13>: 47 inc %edi
0x08049053 <+14>: 43 inc %ebx
0x08049054 <+15>: 42 inc %edx
0x08049055 <+16>: 49 dec %ecx
0x08049056 <+17>: 75 ed jne 0x8049045 <lopa>
0x08049058 <+19>: b8 01 00 00 00 mov $0x1,%eax
0x0804905d <+24>: bb 00 00 00 00 mov $0x0,%ebx
0x08049062 <+29>: cd 80 int $0x80
End of assembler dump.
```

图 1.3.3 实验二汇编语句对应的机器码

```
(gdb) disassemble/r lopa
Dump of assembler code for function lopa:
0x0804900a <+0>: 8a 86 00 a0 04 08 mov 0x804a000(%esi),%al
0x08049010 <+6>: 88 86 0a a0 04 08 mov %al,0x804a00a(%esi)
0x08049016 <+12>: fe c0 inc %al
0x08049018 <+14>: 88 86 14 a0 04 08 mov %al,0x804a014(%esi)
0x0804901e <+20>: 04 03 add $0x3,%al
0x08049020 <+22>: 88 86 1e a0 04 08 mov %al,0x804a01e(%esi)
0x08049026 <+28>: 46 inc %esi
0x08049027 <+29>: 49 dec %ecx
0x08049028 <+30>: 75 e0 jne 0x804900a <lopa>
0x0804902a <+32>: b8 01 00 00 00 mov $0x1,%eax
0x0804902f <+37>: bb 00 00 00 00 mov $0x0,%ebx
0x08049034 <+42>: cd 80 int $0x80
End of assembler dump.
```

图 1.3.4 实验三汇编语句对应的机器码

由两个实验机器码和汇编语句的对应关系可以看出，在采用不同寻址方式时，执行的功能相同，但汇编语句对应的机器码不同。实验二采用寄存器间接寻址方式，机器码没有有关变量地址的内容（如第一行反汇编语句“mov (%esi),%al”对应的机器码为“8a 06”），说明在寻址时直接对存储在寄存器中变量的值进行操作，没有涉及到内存中的具体地址；而实验三采用变址寻址方式，机器码中存在变量的地址（如第一行反汇编语句“mov 0x804a000(%esi),%al”对应的机器码为“8a 86 00 a0 04 08”），说明寻址时通过访问变量的地址来对变量进行操作，涉及到了内存中的具体地址。实验二与实验三的比较，说明了汇编语句操作数的寻址方式影响其对应的机器码。

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，结果是均会在终端输出整个分支的代码块（如图 1.3.5 所示）。这是因为 IP/EIP 指明了指令的起始位置，从不同字节位置开始反汇编时，在 IP/EIP 的作用下调试器都会从其指示的起始位置开始反汇编。该寄存器确保了 CPU 执行时能准确获取指令的起始位置，从而使程序能稳定地运行，表明了其重要性。

```
(gdb) disassemble/r 0x0804901e
Dump of assembler code for function lopa:
0x0804900a <+0>: 8a 86 00 a0 04 08      mov     0x804a000(%esi),%al
0x08049010 <+6>: 88 86 0a a0 04 08      mov     %al,0x804a00a(%esi)
0x08049016 <+12>: fe c0      inc     %al
0x08049018 <+14>: 88 86 14 a0 04 08      mov     %al,0x804a014(%esi)
0x0804901e <+20>: 04 03      add     $0x3,%al
0x08049020 <+22>: 88 86 1e a0 04 08      mov     %al,0x804a01e(%esi)
0x08049026 <+28>: 46      inc     %esi
0x08049027 <+29>: 49      dec     %ecx
0x08049028 <+30>: 75 e0      jne     0x804900a <lopa>
0x0804902a <+32>: b8 01 00 00 00      mov     $0x1,%eax
0x0804902f <+37>: bb 00 00 00 00      mov     $0x0,%ebx
0x08049034 <+42>: cd 80      int     $0x80
End of assembler dump.
(gdb) █
```

图 1.3.5 从不同字节位置开始反汇编

1.6 任务 1.4 的实验过程

1.6.1 设计思想及存储单元分配

题目要求：有一个计算机系统运行状态的监测系统会按照要求收集三个状态信息 a, b, c (均为有符号双字整型数)。假设 n 组状态信息已保存在内存中。对每组的三个数据进行处理模型是 $f=(5a+b-c+100)/128$ (最后结果只保留整数部分)。当 $f<100$ 时，就将该组数据复制到 LOWF 存储区，当 $f=100$ 时，就将该组数据复制到 MIDF 存储区，当 $f>100$ 时，就将该组数据复制到 HIGHF 存储区。

由于 n 组状态信息自行在数据段定义，因此可以不用考虑数据的输入。在数据段提前定义好所有状态信息的总组数，以及各存储区已存放数据组的个数。先将所有状态信息的总组数存放在寄存器中作为循环计数器；循环过程中每次对数据组的三个状态信息进行处理，并将处理结果存储在寄存器中，与 f 进行比较，判断将要存放的存储区后进行跳转；最后根据循环计数器判断所有数据组是否都处理完，若未处理完则继续循环，重新跳转到循环开头进行一系列操作。

1.存储单元分配

n: 状态信息的总组数。

lowfn: LOWF 存储区中已存放数据组的个数。

midfn: MIDF 存储区中已存放数据组的个数。

highfn: HIGHF 存储区中已存放数据组的个数。

MSG: 状态信息数据组的首地址。数据组中共 n 组数据，每组数据包含 1 个数据组 id 和 3 个状态信息，其中数据组 id 为 9 个占 1 个字节的整型数，每个状态信息和处理结果为占 4 个字节的有符号双字整型数，因此可以得到每组状态信息共占 $9+4*4=25$ 个字节。由于每个数据单元的字节数已知，在程序设计过程中便能够通过变址寻址的方式方便地访问相应的数据元素。

LOWF: 存放处理结果小于 100 的数据组的存储区，存储区初始化时用若干个占 1 个字节的数 0 填充，以达到为各存储区分配空间的目的。

MIDF: 存放处理结果等于 100 的数据组的存储区，存储区初始化方法同上。

汇编语言程序设计实验报告

HIGHF: 存放处理结果大于 100 的数据组的存储区, 存储区初始化方法同上。

2. 寄存器分配

ESI: 存放地址偏移量, 利用变址寻址方式访问状态信息数据组。

EDI: 存放状态信息的总组数, 作为程序总循环计数器。

EAX: 存放每组数据的处理结果。

EBX: 存放地址偏移量, 利用变址寻址方式访问各存储区。

ECX: 作为拷贝数据时的循环计数器。

1.6.2 流程图

图 1.4.1 是任务 1.4.1 处理状态信息组的程序流程图。

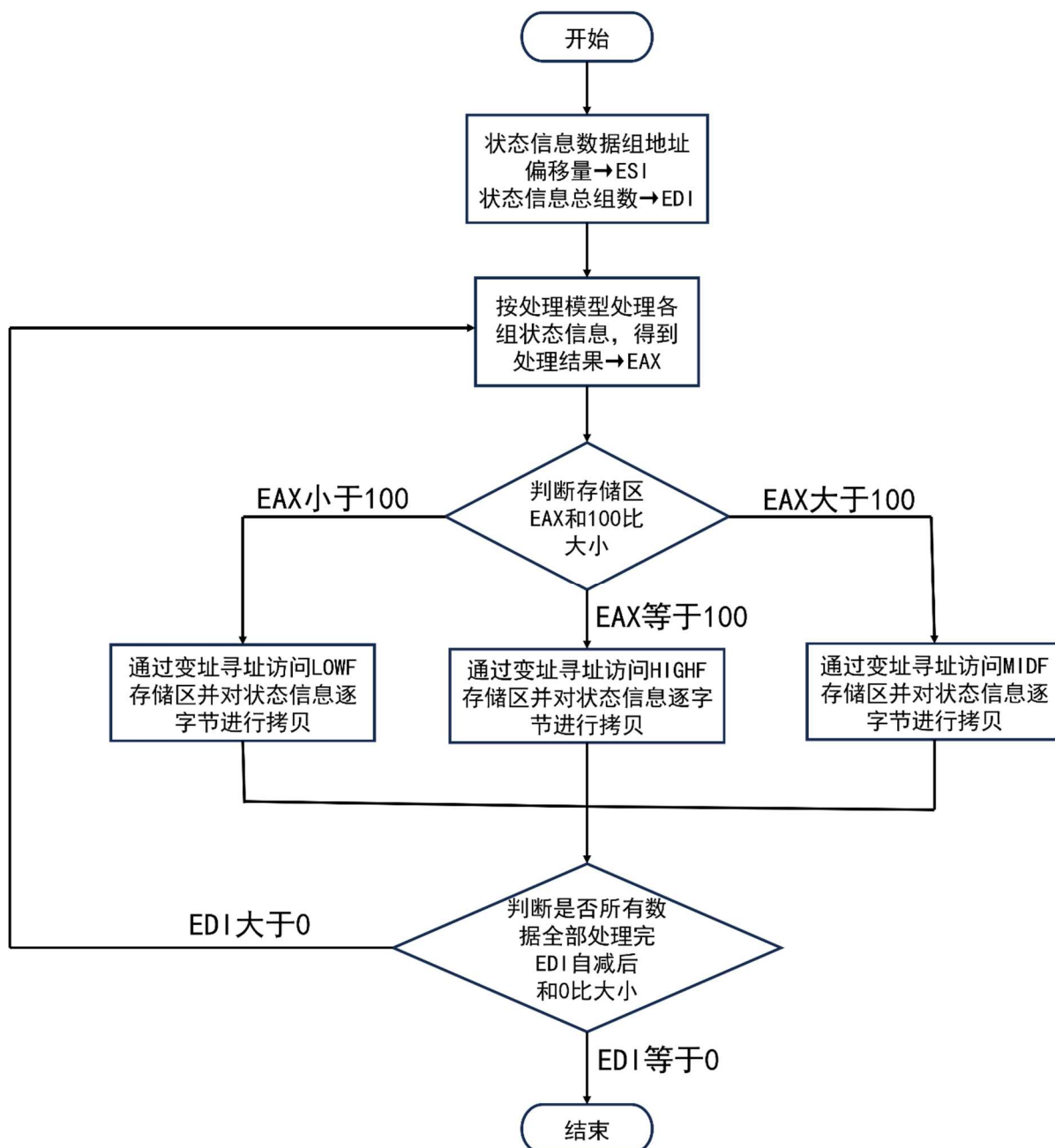


图 1.4.1 处理状态信息组的程序流程图

1.6.3 实验步骤说明

1. 准备上机实验环境。
2. 使用 vim 编辑汇编源文件,将汇编源文件命名为 `opt_before.s`(此处命名为“优化前”,便于后续与优化后的程序进行对比)。使用 `as` 汇编源文件。即 `as -g --32 -o opt_before.o opt_before.s`; 观察提示信息,若出错,则用 vim 修改源文件中的错误,保存后重新汇编,直至不再报错为止。
3. 使用链接程序 `ld` 将汇编生成的 `opt_before.o` 文件链接成执行文件。即 `ld -m elf_i386 -o opt_before opt_before.o`; 若链接时报错,则依照错误信息修改源程序。之后重新汇编和连接,直至不再报错并生成 `opt_before` 可执行二进制文件。
4. 执行该程序。即在命令行提示符后输入 `./opt_before` 后回车,观察执行现象。
5. 使用 `gdb` 观察 `opt_before` 的执行情况。即 `gdb opt_before` 后回车
 - (1) 单步执行 `mov MSG(%esi), %eax` 指令,观察 EAX 的值,即观察通过变址寻址得到的值是否为第一组状态信息的第一条状态数值。
 - (2) 单步执行乘法指令,观察 EAX 的值是否是第一条状态数值的五倍。
 - (3) 单步执行加法指令,观察 EAX 的值是否是原来的值加上第二条状态数值。
 - (4) 单步执行减法指令,观察 EAX 的值是否是原来的值减去第三条状态数值
 - (5) 单步执行 `cdq` 指令,观察 EAX, EDX 的值的变化情况。
 - (6) 执行有符号除法指令前,观察各寄存器存放的内容。
 - (7) 执行完除法指令后,观察 EAX 的值是否是原来的值除以 128。
 - (8) 执行完除法指令后,观察各寄存器值的变化情况
 - (9) 单步执行 `mov %eax, MSG(%esi)`,到数据段内存中观察 MSG 中第一组数据的最后四个字节是否是得到的处理结果值。
 - (10) 单步执行跳转指令,观察 EIP 的值是否是代码段相应存储区分支开始语句的地址。
 - (11) 执行拷贝循环结束后,到数据段相应存储区内存中观察前 25 个字节是否对应第一组状态信息组的状态数值。
 - (12) 单步执行 `dec %edi` 指令,观察标志位的变化情况
 - (13) 单步执行跳转指令,观察 EIP 的值是否重新变为总循环开始语句的地址。
 - (14) 单步执行 `mov MSG(%esi), %eax` 指令,观察 EAX 的值是否是第二组状态信息的第一条状态数值。
 - (15) 在程序结尾处设置断点,执行到该断点处后,到内存中观察所有数据组拷贝完成后各存储区的内容是否与预期一致。
6. 使用 vim 重新编辑汇编源文件,去除汇编源文件中有符号除法处的 `cdq` 指令,再次汇编、链接文件,使用 `gdb` 调试。单步执行除法指令,观察现象。

1.6.4 实验记录与分析

1. 实验环境条件: Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存; Linux 下 vim,

汇编语言程序设计实验报告

as, ld, gdb。

2. 汇编源程序过程没有发生异常。

3. 链接过程没有发生异常。

4. 执行之后程序直接退出到命令行提示符(如图 1.4.2 所示),说明程序执行基本正常,但由于结果没有显示,所以需要用 gdb 调试去观察程序执行过程是否正确。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp1/mission1_4$ as -g --32 -o
opt_before.o opt_before.s
vboxuser@Linux:~/Desktop/computer system experiment/exp1/mission1_4$ ld -m elf_i
386 -o opt_before opt_before.o
vboxuser@Linux:~/Desktop/computer system experiment/exp1/mission1_4$ ./opt_before
vboxuser@Linux:~/Desktop/computer system experiment/exp1/mission1_4$
```

图 1.4.2 程序汇编、链接和执行结果

5. 用 gdb 调入 opt_before 可执行文件后

(1) 本次实验数据段中状态信息组 MSG 的定义如图 1.4.3 所示,即第一组状态信息组的第一条状态数值为 256809,第二条状态数值为-1023,第三组状态数值为 1265,预期的处理结果为 10014。单步执行 mov MSG(%esi), %eax 指令后,观察到 EAX 的值为 256809 (如图 1.4.4 所示),说明变址寻址执行正确。

```
MSG:
.byte 0,0,0,0,0,0,0,0,0
.long 256809
.long -1023
.long 1265
.long 0

.byte 0,0,0,0,0,0,0,0,1
.long 1234
.long -1234
.long 5678
.long 0
```

图 1.4.3 数据段中状态信息组 MSG 定义

```
Breakpoint 1, lopa () at test.s:34
34  mov MSG(%esi), %eax
(gdb) n
35  mov $5, %ecx
(gdb) i r
eax                0x3eb29          256809
```

图 1.4.4 单步执行 mov 指令后 EAX 的值

(2) 单步执行乘指令后,观察到 EAX 的值为(如图 1.4.5 所示),即第一条状态数值的五倍,说明乘法指令执行正确。

```
36  imul %ecx, %eax
(gdb) n
37  add $4, %esi
(gdb) i r
eax                0x1397cd        1284045
```

图 1.4.5 单步执行乘法指令后 EAX 的值

(3) 单步执行加法指令后,观察到 EAX 的值为 1283022,即原来的值加上第二条状态值,说明加法指令执行正确。

(4) 单步执行减法指令后,观察到 EAX 的值为 1281857,即原来的值减去第三条状态

汇编语言程序设计实验报告

值，说明减法指令执行正确。

(5) 单步执行 `cdq` 指令后，观察到 `EAX` 和 `EDX` 的值均不变，这里尚不清楚该指令的具体含义，先继续往后调试程序。

(7) 执行有符号除法指令前，观察到 `EAX` 的值为 1281857，`EBX` 的值为 128，`EDX` 的值为 0。

(8) 执行完有符号除法指令后，观察到 `EAX` 的值变为 10014，即原来的值除以 128，`EBX` 的值不变，`EDX` 的值变为 65（如图 1.4.6 所示），说明有符号除法指令执行正确。而 1281857 除以 128 的余数正好为 65，我们可以推断出执行有符号除法指令 `idiv %ebx` 时各寄存器的作用，即 `EAX` 中存放的是被除数和除法运算完成的商，操作数 `EBX` 中存放的是除数，`EDX` 中存放的是除法运算完成后的余数。

```
(gdb) i r
eax          0x271e          10014
ecx          0x5             5
edx          0x41            65
ebx          0x80            128
```

图 1.4.6 执行完有符号除法指令后各寄存器的值

(9) 单步执行 `mov %eax, MSG(%esi)` 指令后，到数据段内存中观察 `MSG` 中第一组数据的最后四个字节（如图 1.4.7 所示），即存放的处理结果，可以看到在十六进制表示下该结果值为 `0x271e`，将其转换为十进制数为 10014，与此前预期处理结果相同，说明程序模型处理流程完全正确。

```
48      mov %eax, MSG(%esi)
(gdb) n
51      sub $21, %esi
(gdb) x /25bx &MSG
0x804a010:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x804a018:  0x00  0x29  0xeb  0x03  0x00  0x01  0xfc  0xff
0x804a020:  0xff  0xf1  0x04  0x00  0x00  0x1e  0x27  0x00
0x804a028:  0x00
(gdb)
```

图 1.4.7 到内存中观察 `MSG` 中第一组数据最后四个字节的值

(10) 单步执行跳转指令后，观察到 `EIP` 的值为 `0x8049123`，对应代码段中 `highf` 分支开始语句的地址（如图 1.4.8 所示），由于处理结果 10014 比 100 大，说明程序经过比较后即将跳转到正确的存储区分支。

```
0x0804904d <+66>:  jg      0x8049123 <highf>
0x08049053 <+72>:  cmp     $0x64,%eax
0x08049056 <+75>:  je      0x80490bf <midf>
End of assembler dump.
(gdb) i r $eip
eip          0x8049123          0x8049123 <highf>
```

图 1.4.8 执行完跳转指令后 `EIP` 的值

(11) 在拷贝循环结束处设置断点，执行拷贝循环结束后，到内存中观察数据段相应存储区前 25 个字节的值，如图 1.4.9 所示。再到内存中观察数据段 `MSG` 前 25 个字节的值，即第一组状态信息组的内容，如图 1.4.10 所示。可以发现在十六进制表示下两者前 25 个字节的内容完全相同，说明拷贝循环流程执行正确，各状态信息成功地从 `MSG` 拷贝到了相应的存储区。

汇编语言程序设计实验报告

```
Breakpoint 4, next () at test.s:141
141      add $4, %esi
(gdb) x /25bx &HIGHF
0x804ee62: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804ee6a: 0x00 0x29 0xeb 0x03 0x00 0x01 0xfc 0xff
0x804ee72: 0xff 0xf1 0x04 0x00 0x00 0x1e 0x27 0x00
0x804ee7a: 0x00
```

图 1.4.9 拷贝循环结束后到内存中观察相应存储区中的内容

```
(gdb) x /25bx &MSG
0x804a010: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804a018: 0x00 0x29 0xeb 0x03 0x00 0x01 0xfc 0xff
0x804a020: 0xff 0xf1 0x04 0x00 0x00 0x1e 0x27 0x00
0x804a028: 0x00
```

图 1.4.10 到内存中观察数据段 MSG 中第一组状态信息组的内容

(12) 单步执行 `dec %edi` 指令后，观察到标志位 ZF 不为 1，而此后 `jne` 跳转成立的条件为标志位 ZF = 1，因此此时满足跳转条件，程序将跳转到总循环开始处。这一步比较的作用是判断是否全部状态信息组均被处理完，若没有处理完则继续执行总循环，直到处理完为止。

(13) 单步执行跳转指令后，观察到 EIP 的值为总循环开始处的地址，说明程序跳转到总循环开始处，即将重新执行循环语句，上述推断正确。

(14) 单步执行 `mov MSG(%esi), %eax` 指令后，观察到 EAX 的值为 1234，即第二组状态信息的第一条状态数值，说明程序循环执行正确，达到了依次处理各状态信息组的目的。

(15) 在程序结尾处设置断点，执行到该断点处后，到内存中观察所有数据组拷贝完成后各存储区的内容。根据预先计算得出，第一组数据存储在高地址存储区，第二组数据存储在中地址存储区，查看这两个存储区的内容（如图 1.4.11、图 1.4.12 所示），并与 MSG 进行比较，如图 4.13 所示，可以发现在十六进制表示下各存储区内容与 MSG 中对应状态信息组的数据完全一致，说明程序整体执行正确，达成了实验最终目的。

```
(gdb) x /25bx &HIGHF
0x804ee62: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804ee6a: 0x00 0x29 0xeb 0x03 0x00 0x01 0xfc 0xff
0x804ee72: 0xff 0xf1 0x04 0x00 0x00 0x1e 0x27 0x00
0x804ee7a: 0x00
```

图 1.4.11 程序执行结束前 HIGHF 存储区内容

```
(gdb) x /25bx &LOWF
0x804a042: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804a04a: 0x01 0xd2 0x04 0x00 0x00 0x2e 0xfb 0xff
0x804a052: 0xff 0x2e 0x16 0x00 0x00 0xfb 0xff 0xff
0x804a05a: 0xff
```

图 1.4.12 程序执行结束前 LOWF 存储区内容

```
(gdb) x /50bx &MSG
0x804a010: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804a018: 0x00 0x29 0xeb 0x03 0x00 0x01 0xfc 0xff
0x804a020: 0xff 0xf1 0x04 0x00 0x00 0x1e 0x27 0x00
0x804a028: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804a030: 0x00 0x01 0xd2 0x04 0x00 0x00 0x2e 0xfb
0x804a038: 0xff 0xff 0x2e 0x16 0x00 0x00 0xfb 0xff
0x804a040: 0xff 0xff
```

图 1.4.13 到内存中观察数据段 MSG 中全部状态信息组的内容

6. 使用 vim 重新编辑汇编源文件，去除汇编源文件中有符号除法处的 `cdq` 指令，再次汇编、链接文件，使用 gdb 调试。单步执行除法指令后，观察到处理第一组状态信息组结果与之前实验完全相同，而处理第二组状态信息组时程序发生了异常，如图 4.14 所示。


```
Program received signal SIGFPE, Arithmetic exception.  
lopa () at test.s:46  
46      idiv %ebx
```

图 1.4.14 去除 cdq 指令后处理第二组状态信息组时执行除法发生异常

经过上网查阅资料，了解到 cdq 指令的作用是将一个 32 位有符号数扩展为 64 位有符号数，数据能表示的数不变。具体实现过程是把 EAX 的最高位，也就是其符号位，全部复制到 edx 的每一个位上。这样的扩展确保了有符号数除法在处理负数时的正确性，使除法流程不会产生异常。去除 cdq 指令后，由于执行除法指令时，第一组状态信息组的被除数为正数，符号位为 0，扩展后对 EDX 无影响，因此对有符号数除法也无影响，所以虽然缺少该扩展执行程序也能够正常执行；而第二组状态信息组的被除数为负数，需要扩展符号位来确保有符号数除法正常执行，因此缺少该扩展指令使程序发生了错误。

经过如此一番学习后，便填补了之前实验中对 cdq 指令具体作用不理解的空白。

1.7 实验小结

通过本次实验，我主要学习到了 AT&T 汇编语言中各基础指令的作用，熟悉了加减乘除等运算指令，内存拷贝方法，学会了使用 gdb 等调试工具调试程序的步骤及方法，掌握了汇编语言设计程序的基本方法，能够利用循环、跳转等结构实现程序的主要功能，加深了对诸如符号数除法 idiv、符号位扩展 cdq、条件判断 cmp、条件跳转 jne 等重要指令的理解，收获颇深。

程序设计过程并非一帆风顺，在调试过程中遇到了大大小小的问题与 BUG，比如操作数位置不当、内存访问方式不当等，但是通过上网查阅资料、询问老师同学等方法这些问题均得到了很好的解决，我个人对汇编程序的理解能力与也得到了一定程度的提高，思维水平得到了锻炼，可以说本次实验是一次很好的学习与检验的过程。

从上述对实验过程的逐步解析可以发现，本次实验目的达成情况良好，程序最终执行过程没有出现异常，且能够实现实验题目中的要求。但程序在效率、代码简洁程度方面仍有一定的改善空间，需要结合后期的学习不断完善与提升。

实验最后提出了一道思考题：如果三个状态信息是无符号数，程序需要做什么调整。如果状态信息是无符号数，则此时需要考虑溢出的情况，在存储数据时要保证有足够的字节空间，此外在需要时还应该进行位数的扩展。在进行乘除法时，原本的 imul、idiv 指令应替换成无符号数乘法 mul 和无符号数除法 div 指令。在使用跳转指令时，原本的 jl、jg 指令则应替换成比较无符号数大小的 jb、ja 指令。

在实验操作的过程中，我也得到了不少经验教训，比如操作数位置时常摆放不当，造成结果大相径庭，除法指令中涉及的寄存器不熟悉，导致寄存器内容混乱等等。同时发现了自己还存在的一些问题，例如不能将一些基础且重要的指令或寄存器的作业牢记脑中，每次都要重新查阅课本或上网查阅相关资料，比较浪费时间，希望在今后的学习中能得到改善，并一步步提高自己对汇编程序的阅读能力、设计能力和代码的编写能力。

2 程序优化

2.1 实验内容

在任务 1.4 描述的背景下，假设在输入缓冲区中已经存放了 N 组采集到的状态信息，需要对这 N 组数据分别计算对应的 f 并依据分组原则将 N 组数据复制到对应的 LOWF、MIDF、HIGHF 存储区中。

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）。请尝试对 f 的计算过程以及数据的复制过程的代码进行优化，通过对这些代码执行时间的计时来判断优化的效果。

为体现程序的优化效果，对任务 1.4 的程序主体，重复执行 m 次。m 的大小可以自行设定，直到总的执行时间和优化时间有明显的效果。

在程序主体执行前使用 clock 函数计时，计时结果保存在变量 start_t 中。在程序主体执行结束后再次调用 clock 计时，结果保存在 end_t 中。调用 printf 输出(end_t - start_t)，即程序主体的执行时间。

分别实现未优化和优化的程序，对比两者的执行时间。要求优化效果提高 10%以上。

2.2 任务 2.1 实验过程

2.2.1 设计思想

为了直观地得到程序的优化效果，采取调用 C 函数 clock 的方式获取程序总运行时间，并调用 C 函数 printf 对该时间进行输出，这样便能清晰地看到程序每次执行所花费的总时间，从而对程序优化效果进行评估。

然而，当状态信息的总组数较小时，程序执行很快便会结束，每次花费的时间差异极不明显，无法判断优化效果。因此，通过对任务 1.4 的程序主体重复执行多次，直到程序总执行时间优化前后出现明显差异，从而能很好地体现程序的优化效果。这里我将程序主体重复执行了十万次，为方便数据段原始状态信息组数据定义，我采用了 fill 伪指令填充的方式对 MSG 进行定义。

准备工作全部完成后，接下来设计具体的程序优化方法：

（1）将执行慢的指令替换为执行快的指令，例如用移位和加法指令替代汇编程序中的除法指令。

（2）将循环、分支结构展开成顺序结构，充分利用 CPU 的流水线特性，提高程序的执行效率。

（3）使用宽字节指令传送，例如用 movl 替代 movb、movw 等，减少循环中指令的执行次数。

（4）算法设计上的优化，用加法替代乘法，例如 EAX*5 可以写成“add %eax, %eax”“add %eax, %eax”和“add MSG(%esi), %eax”三条语句，使用多条快指令的执行效率仍比

汇编语言程序设计实验报告

使用一条慢指令要高。

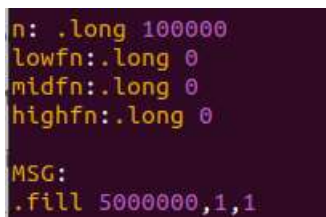
2.2.2 实验步骤说明

1. 准备上机实验环境。

2. 使用 vim 编辑汇编源文件，将汇编源文件命名为 `opt_after.s`（此处命名为“优化后”，便于与优化前的程序进行对比）。

3. 汇编源程序编写及优化过程如下：

（1）为方便数据段原始状态信息组数据定义，我采用了 `.fill` 伪指令填充的方式对 `MSG` 进行定义，并令程序的总循环次数为十万次，如图 2.1 所示。



```
n: .long 100000
lowfn: .long 0
midfn: .long 0
highfn: .long 0

MSG:
.fill 5000000, 1, 1
```

图 2.1 数据段 `MSG` 定义及总循环次数定义

（2）在优化前的原汇编代码基础上，调用 C 函数 `clock` 和 `printf` 记录程序的总执行时间，并在程序结束前打印在屏幕上。这一部分编写完成后，先对调用 C 函数的效果和优化前程序执行效率进行测试和记录，汇编、链接源文件后，执行 `opt_after`，在终端观察现象。

（3）将原来的乘法指令 “`mov $5, %ecx`” 和 “`imul %ecx, %eax`” 替换为加法指令 “`add %eax, %eax`” “`add %eax, %eax`” 和 “`add MSG(%esi), %eax`”。

（4）将原来的除法指令 “`mov $128, %ebx`” 和 “`idiv %ebx`” 替换为移位和加法指令 “`xor %edx, %edx`” “`add %edx, %eax`” 和 “`sar $7, %eax`”。

（5）将循环结构展开成顺序结构，即去除跳转操作，使用多条重复指令替代原有的循环结构。

（6）使用宽字节指令传送，将原有的传送指令 `movb` 替换为 `movl`（必要时仍使用 `movb`，如只需传送单个字节）。

4. 每完成一个优化步骤，就汇编、链接优化后的汇编文件，执行 `opt_after`，在终端观察现象并记录下完成此次优化后程序的总执行时间。

5. 所有优化步骤均完成后，记录下优化后程序的总执行时间，与原来优化前的总执行时间进行对比，计算出程序优化效果提升的百分比。

2.2.3 实验记录与分析

1. 实验环境条件：Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存；Linux 下 vim, as, ld, gdb。

2. 汇编源程序编写及优化过程：

（1）调用 C 函数 `clock` 和 `printf` 部分编写完成后，汇编、链接程序，执行 `opt_after`，观察到在终端中打印出了程序的总执行时间为 5756 毫秒，如图 2.2 所示。这个时间便是优化

汇编语言程序设计实验报告

前程序的总执行时间，对其进行记录，之后与优化后程序的总执行时间进行比较。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp2/mission2_1$ ./opt_after
run time: 5756
```

图 2.2 优化前程序总执行时间

(2) 将原来的乘法指令“`mov $5, %ecx`”和“`imul %ecx, %eax`”替换为加法指令“`add %eax, %eax`”“`add %eax, %eax`”和“`add MSG(%esi), %eax`”。汇编、链接程序，执行 `opt_after`，观察到在终端中打印出了完成此次优化后程序的总执行时间为 5320 毫秒，相比优化前程序效率得到一定提升，说明此次优化效果良好。

(3) 将原来的除法指令“`mov $128, %ebx`”和“`idiv %ebx`”替换为移位和加法指令“`xor %edx, %edx`”“`add %edx, %eax`”和“`sar $7, %eax`”。由于除数 128 正好为 2 的 7 次方，根据二进制数移位操作的本质，每次向左（向右）移动一位，则二进制数等于原数乘以 2（除以 2）后的结果。因此这里汇编程序中的除法指令可以直接用向右移动 7 位替代，与除以 128 达到的效果相同，但是移位指令执行速度更快，因此程序的效率比使用除法指令更高。汇编、链接程序，执行 `opt_after`，观察到在终端中打印出了完成此次优化后程序的总执行时间为 4945 毫秒，相比上次优化程序效率得到一定提升，说明此次优化效果良好。

(4) 将循环结构展开成顺序结构，即去除跳转操作，使用多条重复指令替代原有的循环结构，如图 2.3 所示。汇编、链接程序，执行 `opt_after`，观察到在终端中打印出了完成此次优化后程序的总执行时间为 4677 毫秒，相比上次优化程序效率得到一定提升，说明此次优化效果良好。


```
lowf:
//去除循环结构, 优化为流水线模式
//拷贝每个数据组id(9个字节)
//采用顺序结构, 每次拷贝4个字节, 耗时较短
movl lowfn, %ebx
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %esi
add $4, %ebx
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %esi
add $4, %ebx
movb MSG(%esi), %al
movb %al, LOWF(%ebx)
inc %esi
inc %ebx
//依次拷贝每个数据组的数据a,b,c,sf(每个占4个字节)
//采用顺序结构, 每次拷贝4个字节, 耗时较短
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %esi
add $4, %ebx
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %esi
add $4, %ebx
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %esi
add $4, %ebx
movl MSG(%esi), %eax
movl %eax, LOWF(%ebx)
add $4, %ebx
movl %ebx, lowfn
jmp next
```

图 2.3 将循环结构展开成顺序结构

(5) 使用宽字节指令传送, 将原有的传送指令 `movb` 按需替换为 `movl`。汇编、链接程序, 执行 `opt_after`, 观察到在终端中打印出了完成此次优化后程序的总执行时间为 4430 毫秒, 相比上次优化程序效率得到一定提升, 说明此次优化效果良好。

(6) 所有优化工作均完成后, 再次汇编、链接程序, 执行 `opt_after`, 观察到在终端中打印出了完成所有优化后程序的总执行时间为 4253 毫秒, 如图 2.4 所示。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp2/misson2_1$ ./opt_after
run time: 4253
```

图 2.4 优化后程序总执行时间

3. 将优化前后程序的总执行时间进行比较, 计算出程序优化效果提升的百分比为 $(5756 - 4253) \div 5756 \times 100\% = 26.1\%$, 满足了实验要求中优化效果提升 10% 以上的要求, 说明程序整体优化效果非常好。

2.3 实验小结

本次实验我了解了程序计时的方法以及运行环境对程序执行情况的影响。深刻理解了 CPU 执行指令的过程, 明白了流水线模式的工作特性。熟悉了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。初步了解了 C 函数的

汇编语言程序设计实验报告

调用方法和操作过程，进一步提升了汇编程序的阅读理解能力和设计能力。

每完成一步优化操作后，程序执行效率均得到了一定程度的提升，通过实验进一步验证了每个优化步骤的正确性。而完成所有优化操作后，观察到程序执行效率相比优化前有着较大幅度的提升，远远超出了实验要求优化效果提升的百分比，说明从不同角度完成优化后，程序总体优化效果非常好，此次实验十分成功。

然而，本次实验仍存在着些许不足。例如对一些陌生的指令不够熟悉（如 `xor` 指令、`sar` 指令等等），且不能完全独立自主地想到相应的程序优化策略，部分优化方法需要通过老师指点、向身边同学请教或者上网查阅有关资料才能意识到。希望通过日后的不断学习，能够加强自己的思维能力，以及对汇编指令的熟悉程度，更进一步提升对汇编语言的理解能力和设计能力。

3 二进制炸弹破解

3.1 实验目的和要求

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。

3.2 实验内容

上机实验环境说明：使用 Linux 环境，需要使用 gdb 和 objdump 工具。

破解“二进制炸弹”：

“二进制炸弹”是一个 Linux 可执行程序。每个同学按自己学号的后七位数字，到群文件中下载自己的压缩包，解压后可得到 bomb 执行程序和 bomb.c 总控源程序。

bomb 执行程序由六个阶段组成。每个阶段都需要输入特定的字符串。如果输入了正确的字符串，那么该阶段就“解除”。否则，炸弹会通过打印“BOOM!!!”。每个同学的目标是解除尽可能多的阶段。

3.3 任务 3.1 实验过程

3.3.1 实验方法说明

1. 准备上机实验环境，对实验用到的软件进行安装、运行，通过试用初步了解软件的基本功能、操作等。
2. 使用 gdb 调试 bomb 二进制可执行文件，熟悉 gdb 中反汇编的方法。
3. 根据反汇编得到的汇编代码尝试理解代码中各指令的含义、寄存器的作用和存储的内容、程序的机器级表示以及程序的执行过程。
4. 单步调试程序，分析程序的执行过程，到相关地址的内存中查看可能需要的数据，尝试破解各个炸弹。

3.3.2 实验记录与分析

1. 实验环境条件：

Oracle VM VirtualBox 虚拟机 Ubuntu22.04.3 4G 内存；

Linux 下 gdb, odjdump 工具

2. 实验记录与分析：

使用 gdb 调试 bomb 二进制可执行文件，先使用指令 l 初步查看 bomb 的源代码内容，发现是包含若干头文件的 C 语言程序，程序只定义了主函数（如图 3.1 所示），而其中使用的函数（如 initialize_bomb(), phase_1()等）均在其他头文件中定义，因此无法查看，只能

汇编语言程序设计实验报告

通过对本文件的调试和反汇编尝试逐一破解各函数的具体内容。

```
(gdb) l
23      #include <stdio.h>
24      #include <stdlib.h>
25      #include "support.h"
26      #include "phases.h"
27
28      /*
29      * Note to self: Remember to erase this file so my victims will have no
30      * idea what is going on, and so they will all blow up in a
31      * spectacular fiendish explosion. -- Dr. Evil
32      */
(gdb)
33
34      FILE *infile;
35
36      int main(int argc, char *argv[])
37      {
38          char *input;
```

图 3.1 初步查看 bomb 源代码

接下来使用 gdb 对 bomb 进行单步调试，逐关卡进行破解。

(1) 破解 phase_1:

在 phase_1 处设置断点，执行程序。终端上显示第一关的提示语，并要求输入答案，先输入任意字符串进行观察，这里我输入的是“abcdefg”。程序执行到 phase_1 处后，使用指令 disassemble phase_1 对 phase_1 中的内容进行反汇编，得到反汇编后的汇编代码如图 3.2 所示。

```
Breakpoint 1, 0x565564cd in phase_1 ()
(gdb) disassemble phase_1
Dump of assembler code for function phase_1:
=> 0x565564cd <+0>:      push    %ebx
0x565564ce <+1>:      sub     $0x10,%esp
0x565564d1 <+4>:      call   0x56556240 <__x86.get_pc_thunk.bx>
0x565564d6 <+9>:      add     $0x4a8e,%ebx
0x565564dc <+15>:     lea     -0x2e20(%ebx),%eax
0x565564e2 <+21>:     push    %eax
0x565564e3 <+22>:     push    0x1c(%esp)
0x565564e7 <+26>:     call   0x56556a2c <strings_not_equal>
0x565564ec <+31>:     add     $0x10,%esp
0x565564ef <+34>:     test   %eax,%eax
0x565564f1 <+36>:     jne     0x565564f8 <phase_1+43>
0x565564f3 <+38>:     add     $0x8,%esp
0x565564f6 <+41>:     pop     %ebx
0x565564f7 <+42>:     ret
0x565564f8 <+43>:     call   0x56556b44 <explode_bomb>
0x565564fd <+48>:     jmp     0x565564f3 <phase_1+38>
End of assembler dump.
(gdb)
```

图 3.2 对 phase_1 中的内容进行反汇编

其中，屏幕上的“=>”箭头符号指示的就是下一条将要执行的语句。使用 ni 指令对汇编代码进行单步调试（注意此时单步调试不能使用 n 指令，ni 和 n 的区别在单步的跨度上，n 指令执行的是程序源代码，而 ni 指令执行的是单条机器指令），在程序执行完 phase_1<+22> 处的语句后，到内存中观察 EAX 的内容，如图 3.3 所示。

汇编语言程序设计实验报告

```
(gdb) x /100bc $eax
0x56558144: 72 'H' 111 'o' 117 'u' 115 's' 101 'e' 115 's' 32 ' ' 119 'w'
0x5655814c: 105 'i' 108 'l' 108 'l' 32 ' ' 98 'b' 101 'e' 103 'g' 97 'a'
0x56558154: 116 't' 32 ' ' 106 'j' 111 'o' 98 'b' 115 's' 44 ',' 32 ' '
0x5655815c: 106 'j' 111 'o' 98 'b' 115 's' 32 ' ' 119 'w' 105 'i' 108 'l'
0x56558164: 108 'l' 32 ' ' 98 'b' 101 'e' 103 'g' 97 'a' 116 't' 32 ' '
0x5655816c: 104 'h' 111 'o' 117 'u' 115 's' 101 'e' 115 's' 46 '.' 0 '\000'
```

图 3.3 到内存中观察 EAX 中的内容

由于接下来将会进入 `strings_not_equal` 函数，通过该函数名称可以推测该函数的作用是
比较用户输入的字符串与答案字符串，进而可以推断刚才观察到的 EAX 中的内容就是答案
字符串。在 `strings_not_equal` 函数处设置断点，继续单步调试，进入 `strings_not_equal` 函数
内部对刚刚的推断进行验证。反汇编得到 `strings_not_equal` 函数的汇编代码，如图 3.4 所示。

```
Breakpoint 2, 0x56556a2c in strings_not_equal ()
(gdb) disassemble strings_not_equal
Dump of assembler code for function strings_not_equal:
=> 0x56556a2c <+0>:      push    %edi
    0x56556a2d <+1>:      push    %esi
    0x56556a2e <+2>:      push    %ebx
    0x56556a2f <+3>:      mov     0x10(%esp),%ebx
    0x56556a33 <+7>:      mov     0x14(%esp),%esi
    0x56556a37 <+11>:     push    %ebx
    0x56556a38 <+12>:     call    0x56556a0e <string_length>
    0x56556a3d <+17>:     mov     %eax,%edi
    0x56556a3f <+19>:     mov     %esi,(%esp)
    0x56556a42 <+22>:     call    0x56556a0e <string_length>
    0x56556a47 <+27>:     add     $0x4,%esp
    0x56556a4a <+30>:     mov     %eax,%edx
    0x56556a4c <+32>:     mov     $0x1,%eax
    0x56556a51 <+37>:     cmp     %edx,%edi
    0x56556a53 <+39>:     jne     0x56556a80 <strings_not_equal+84>
    0x56556a55 <+41>:     movzbl (%ebx),%eax
    0x56556a58 <+44>:     test    %al,%al
    0x56556a5a <+46>:     je      0x56556a74 <strings_not_equal+72>
    0x56556a5c <+48>:     cmp     %al,(%esi)
    0x56556a5e <+50>:     jne     0x56556a7b <strings_not_equal+79>
    0x56556a60 <+52>:     add     $0x1,%ebx
    0x56556a63 <+55>:     add     $0x1,%esi
--Type <RET> for more, q to quit, c to continue without paging--c
    0x56556a66 <+58>:     movzbl (%ebx),%eax
    0x56556a69 <+61>:     test    %al,%al
    0x56556a6b <+63>:     jne     0x56556a5c <strings_not_equal+48>
    0x56556a6d <+65>:     mov     $0x0,%eax
    0x56556a72 <+70>:     jmp     0x56556a80 <strings_not_equal+84>
    0x56556a74 <+72>:     mov     $0x0,%eax
    0x56556a79 <+77>:     jmp     0x56556a80 <strings_not_equal+84>
    0x56556a7b <+79>:     mov     $0x1,%eax
    0x56556a80 <+84>:     pop     %ebx
    0x56556a81 <+85>:     pop     %esi
    0x56556a82 <+86>:     pop     %edi
    0x56556a83 <+87>:     ret
```

图 3.4 对 `strings_not_equal` 中的内容进行反汇编

由于 EBX 和 ESI 中的内容均由 ESP 变址寻址得到的内容传送而来，因此可以判断 EBX
和 ESI 中存储的是进入该函数前传入的参数。单步调试到 `call string_length` 处时，到内存中
查看 EBX 和 ESI 中的内容（如图 3.5、图 3.6 所示），发现 EBX 的内容正好是所输入的字符
串“abcdefg”，而 ESI 的内容与刚才 EAX 的内容（即所推测的答案字符串）相同，因此可

汇编语言程序设计实验报告

以验证我们之前的推测基本正确。strings_not_equal 中的 strings_length 函数和剩下部分即为对 EBX（内容为我们输入的字符串）和 ESI（内容为答案字符串）中内容的长度和各个位字符的比较，最后将比较结果通过寄存器 EAX 返回，再由 phase_1 函数根据 EAX 的值决定是否成功解除炸弹。

```
(gdb) x /100bc $ebx
0x5655b3a0 <input_strings>:  97 'a' 98 'b' 99 'c' 100 'd' 101 'e' 102 'f' 103 'g' 0 '\000'
```

图 3.5 到内存中查看 EBX 的内容

```
(gdb) x /100bc $esi
0x56558144:  72 'H' 111 'o' 117 'u' 115 's' 101 'e' 115 's' 32 ' ' 119 'w'
0x5655814c:  105 'i' 108 'l' 108 'l' 32 ' ' 98 'b' 101 'e' 103 'g' 97 'a'
0x56558154:  116 't' 32 ' ' 106 'j' 111 'o' 98 'b' 115 's' 44 ',' 32 ' '
0x5655815c:  106 'j' 111 'o' 98 'b' 115 's' 32 ' ' 119 'w' 105 'i' 108 'l'
0x56558164:  108 'l' 32 ' ' 98 'b' 101 'e' 103 'g' 97 'a' 116 't' 32 ' '
0x5655816c:  104 'h' 111 'o' 117 'u' 115 's' 101 'e' 115 's' 46 '.' 0 '\000'
```

图 3.6 到内存中查看 ESI 的内容

接下来取消所有断点，重新执行 bomb 程序，在关卡 1 输入我们所推测的答案字符串“House will begat jobs, jobs will begat houses.”后回车，发现程序提示“Phase 1 defused”，表明该关卡炸弹拆除成功（如图 3.7 所示），说明我们关卡 1 的破解思路及调试方法基本正确。最后对关卡 1 进行总结，其涉及到的知识点主要为字符串的比较。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
```

图 3.7 关卡 1 成功破解

（2）破解 phase_2:

在 phase_2 处设置断点，执行程序。由于前面的关卡 1 我们已经成功破解，因此可以把答案保存在名为 answer 文件中，后面每次执行程序时，在 r 指令后附带该文件名 answer，这样可以直接将该文件中的内容作为输入给程序，从而跳过已经破解过的关卡，节约时间。

同样的，在提示输入关卡 2 的答案时，我先输入任意一个字符串“abcdefg”进行初步观察。然后使用指令 disassemble phase_2 对 phase_2 的内容进行反汇编，如图 3.8 所示。


```

Breakpoint 3, 0x565564ff in phase_2 ()
(gdb) disassemble phase_2
Dump of assembler code for function phase_2:
=> 0x565564ff <+0>:      push    %edi
    0x56556500 <+1>:      push    %esi
    0x56556501 <+2>:      push    %ebx
    0x56556502 <+3>:      sub     $0x28,%esp
    0x56556505 <+6>:      call    0x56556240 <__x86.get_pc_thunk.bx>
    0x5655650a <+11>:     add     $0x4a5a,%ebx
    0x56556510 <+17>:     mov     %gs:0x14,%eax
    0x56556516 <+23>:     mov     %eax,0x24(%esp)
    0x5655651a <+27>:     xor     %eax,%eax
    0x5655651c <+29>:     lea     0xc(%esp),%eax
    0x56556520 <+33>:     push    %eax
    0x56556521 <+34>:     push    0x3c(%esp)
    0x56556525 <+38>:     call    0x56556b79 <read_six_numbers>
    0x5655652a <+43>:     add     $0x10,%esp
    0x5655652d <+46>:     cmpl    $0x0,0x4(%esp)
    0x56556532 <+51>:     jne     0x5655653b <phase_2+60>
    0x56556534 <+53>:     cmpl    $0x1,0x8(%esp)
    0x56556539 <+58>:     je      0x56556540 <phase_2+65>
    0x5655653b <+60>:     call    0x56556b44 <explode_bomb>
    0x56556540 <+65>:     lea     0x4(%esp),%esi
    0x56556544 <+69>:     lea     0x14(%esp),%edi
    0x56556548 <+73>:     jmp     0x56556551 <phase_2+82>
    0x5655654a <+75>:     add     $0x4,%esi
    0x5655654d <+78>:     cmp     %edi,%esi
    0x5655654f <+80>:     je      0x56556562 <phase_2+99>
    0x56556551 <+82>:     mov     0x4(%esi),%eax
    0x56556554 <+85>:     add     (%esi),%eax
    0x56556556 <+87>:     cmp     %eax,0x8(%esi)
    0x56556559 <+90>:     je      0x5655654a <phase_2+75>
    0x5655655b <+92>:     call    0x56556b44 <explode_bomb>
    0x56556560 <+97>:     jmp     0x5655654a <phase_2+75>
    0x56556562 <+99>:     mov     0x1c(%esp),%eax
    0x56556566 <+103>:    sub     %gs:0x14,%eax
    0x5655656d <+110>:    jne     0x56556576 <phase_2+119>
    0x5655656f <+112>:    add     $0x20,%esp
    0x56556572 <+115>:    pop     %ebx
    0x56556573 <+116>:    pop     %esi
    0x56556574 <+117>:    pop     %edi
    0x56556575 <+118>:    ret
    0x56556576 <+119>:    call    0x56557950 <__stack_chk_fail_local>
End of assembler dump.

```

图 3.8 对 phase_2 的内容进行反汇编

从 phase_2 的反汇编代码中可以看出，其包含一个名为 read_six_numbers 的函数，推测该函数的作用是读取用户输入的六个数字。因此现在我们重新执行 bomb 程序，在提示输入关卡 2 的答案时，我输入“1 2 3 4 5 6”后再次观察程序的执行情况。单步执行完 phase_2<+43> 指令后，到内存中观察 ESP 中的内容，如图 3.9 所示。

```

(gdb) x /40bd $esp
0xfffffcf60:   100    -81    85    86    1    0    0    0
0xfffffcf68:    2     0     0     0    3    0    0    0
0xfffffcf70:    4     0     0     0    5    0    0    0
0xfffffcf78:    6     0     0     0    0   -86   102   21
0xfffffcf80:   100    -81    85    86  -124  -48   -1   -1

```

图 3.9 到内存中观察 ESP 中的内容

发现 ESP 中从第 5 个字节开始即是我们所输入的六个数字。知道了 ESP 中各偏移地址

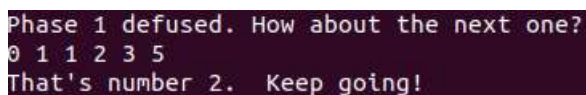
汇编语言程序设计实验报告

的内容后，再对反汇编语句进行分析，由 `phase_2<+46>` 处的语句可以看出，指令将 `ESP+4` 处的内容，也就是我们输入的第一个数字与 0 进行比较，若不相等则跳转到 `phase_2<+60>` 处引爆炸弹。说明答案字符串的第一个数字为 0。

接下来由 `phase_2<+53>` 处的语句可以看出，指令将 `ESP+8` 处的内容，也就是我们输入的第二个数字与 1 进行比较，若相等则跳转到 `phase_2<+65>` 处继续执行程序，不相等则不跳转，继续顺序执行语句，而按顺序下一条语句即为引爆炸弹，这表明了若两者不相等则会引爆炸弹。说明答案字符串的第二个数字为 1。

继续分析剩下的部分，可以发现程序进入了一个循环。循环的内容是从我们输入的第三个数字开始，每次取该数字的值，判断它与它之前的两数之和是否相等，若不相等则引爆炸弹，若相等则继续下一个循环，直到第六个数字取完为止。这说明了答案字符串是一个前两位分别为 0 和 1 的一个斐波那契数列，从而推出答案字符串为 “0 1 1 2 3 5”。

下面取消所有断点，通过附带 `answer` 文件的方式重新执行程序，在提示输入关卡 2 的答案时，输入刚刚推出的答案字符串 “0 1 1 2 3 5” 后，观察到终端中提示关卡 2 的炸弹成功解除，顺利进入阶段 3（如图 3.10 所示）。说明我们关卡 2 的破解思路及调试方法基本正确。最后对关卡 2 进行总结，其涉及到的主要是循环。



```
Phase 1 defused. How about the next one?  
0 1 1 2 3 5  
That's number 2. Keep going!
```

图 3.10 关卡 2 成功破解

（3）破解 `phase_3`：

将关卡 2 的答案同样保存在文件 `answer` 中，在 `phase_3` 处设置断点。在执行程序之前，先使用 `disassemble phase_3` 对 `phase_3` 进行反汇编（如图 3.11 所示），事先观察关卡 3 要求输入的答案形式是怎样的，这样可以避免盲目尝试，提高我们破解关卡的效率。

汇编语言程序设计实验报告

```
(gdb) disassemble phase_3
Dump of assembler code for function phase_3:
0x5655657b <+0>:      push    %ebx
0x5655657c <+1>:      sub     $0x18,%esp
0x5655657f <+4>:      call   0x56556240 <__x86.get_pc_thunk.bx>
0x56556584 <+9>:      add     $0x49e0,%ebx
0x5655658a <+15>:     mov     %gs:0x14,%eax
0x56556590 <+21>:     mov     %eax,0xc(%esp)
0x56556594 <+25>:     xor     %eax,%eax
0x56556596 <+27>:     lea     0x8(%esp),%eax
0x5655659a <+31>:     push    %eax
0x5655659b <+32>:     lea     0x8(%esp),%eax
0x5655659f <+36>:     push    %eax
0x565565a0 <+37>:     lea     -0x2c81(%ebx),%eax
0x565565a6 <+43>:     push    %eax
0x565565a7 <+44>:     push    0x2c(%esp)
0x565565ab <+48>:     call   0x56556140 <__isoc99_sscanf@plt>
0x565565b0 <+53>:     add     $0x10,%esp
0x565565b3 <+56>:     cmp     $0x1,%eax
0x565565b6 <+59>:     jle     0x565565ce <phase_3+83>
0x565565b8 <+61>:     cmpl    $0x7,0x4(%esp)
0x565565bd <+66>:     ja      0x5655661c <phase_3+161>
0x565565bf <+68>:     mov     0x4(%esp),%eax
0x565565c3 <+72>:     mov     %ebx,%edx
0x565565c5 <+74>:     add     -0x2dc0(%ebx,%eax,4),%edx
0x565565cc <+81>:     jmp     *%edx
0x565565ce <+83>:     call   0x56556b44 <explode_bomb>
0x565565d3 <+88>:     jmp     0x565565b8 <phase_3+61>
0x565565d5 <+90>:     mov     $0xa3,%eax
0x565565da <+95>:     cmp     %eax,0x8(%esp)
0x565565de <+99>:     jne     0x5655662f <phase_3+180>
0x565565e0 <+101>:    mov     0xc(%esp),%eax
0x565565e4 <+105>:    sub     %gs:0x14,%eax
0x565565eb <+112>:    jne     0x56556636 <phase_3+187>
0x565565ed <+114>:    add     $0x18,%esp
0x565565f0 <+117>:    pop     %ebx
0x565565f1 <+118>:    ret
0x565565f2 <+119>:    mov     $0x91,%eax
0x565565f7 <+124>:    jmp     0x565565da <phase_3+95>
0x565565f9 <+126>:    mov     $0x23d,%eax
0x565565fe <+131>:    jmp     0x565565da <phase_3+95>
0x56556600 <+133>:    mov     $0x1fa,%eax
0x56556605 <+138>:    jmp     0x565565da <phase_3+95>
0x56556607 <+140>:    mov     $0x26d,%eax
0x5655660c <+145>:    jmp     0x565565da <phase_3+95>
0x5655660e <+147>:    mov     $0x147,%eax
0x56556613 <+152>:    jmp     0x565565da <phase_3+95>
0x56556615 <+154>:    mov     $0x26c,%eax
0x5655661a <+159>:    jmp     0x565565da <phase_3+95>
0x5655661c <+161>:    call   0x56556b44 <explode_bomb>
--Type <RET> for more, q to quit, c to continue without paging--c
0x56556621 <+166>:    mov     $0x0,%eax
0x56556626 <+171>:    jmp     0x565565da <phase_3+95>
0x56556628 <+173>:    mov     $0x2bf,%eax
0x5655662d <+178>:    jmp     0x565565da <phase_3+95>
0x5655662f <+180>:    call   0x56556b44 <explode_bomb>
0x56556634 <+185>:    jmp     0x565565e0 <phase_3+101>
0x56556636 <+187>:    call   0x56557950 <__stack_chk_fail_local>
End of assembler dump.
```

图 3.11 对 phase_3 的内容进行反汇编

通过观察反汇编语句,发现 phase_3 中包含对 scanf 的调用,因此需要知道 scanf 中的参

汇编语言程序设计实验报告

数格式如何，从而确定我们输入字符串的格式。先任意输入一个字符串“1 2 3 4 5 6”，执行程序，单步执行完 `call __isoc99_sscanf@plt` 语句后，到内存观察反汇编语句中出现的 `-0x2c81(%ebx)` 中的内容，如图 3.12 所示。

```
(gdb) x /40bc $ebx-0x2c81
0x565582e3: 37 '%' 100 'd' 32 ' ' 37 '%' 100 'd' 0 '\000' 69 'E' 114 'r'
```

图 3.12 到内存中观察相应地址中的内容

发现该地址中前几个字节正好对应 C 函数 `scanf` 中的参数格式，即从用户输入中接收两个整型数值。重新执行程序，这次输入字符串“1 2”再次进行尝试。单步执行到 `phase_3<+56>` 处的语句，观察到 EAX 的值为 2，而该处语句为比较 EAX 与 1 的大小，若 EAX 小于等于 1 则引爆炸弹，因此推断此时 EAX 中的内容为我们输入字符串中包含的数字的个数。说明我们此前得到的 `scanf` 要求输入两个整数的参数格式正确。

通过观察接下来的反汇编语句，发现涉及到对 ESP 偏移地址相应内容的操作，因此现在再到内存中观察 ESP 中的内容，如图 3.13 所示。

```
(gdb) x /40bd $esp
0xfffffcf70: 64 -76 85 86 1 0 0 0
0xfffffcf78: 2 0 0 0 0 50 -27 120
```

图 3.13 到内存中观察 ESP 中的内容

可以看到 ESP 的第 5 个字节到第 8 个字节，即 `ESP+4`，是我们输入的第一个数字；ESP 的第 9 个字节到第 12 个字节，即 `ESP+8`，是我们输入的第二个数字。反汇编语句中 `phase_3<+61>` 处将我们输入的第一个数字与 7 进行比较，若大于 7 则引爆炸弹，说明答案第一个数字不能大于 7。

观察到反汇编语句中存在 `jmp *edx` 语句，该语句的作用是跳转到 EDX 中地址对应的语句处（EDX 中的内容为一个地址值）。而由上面两行语句得到 EDX 的值为 EBX 的值加偏移地址 `EBX-0x2dc0+EAX`（此时 EAX 中是输入的第一个数字）*4 对应的值。通过计算我们可以发现，当第一个数字是 1 到 7 时，EDX 的值正好分别对应 `phase_3<+90>`，`phase_3<+124>`，`phase_3<+131>`，`phase_3<+131>`，`phase_3<+138>`，`phase_3<+145>`，`phase_3<+152>` 处语句的地址。由此可见，该跳转指令类似 C 语言中 `switch` 的机器级表示，根据输入的数的值跳转到不同分支执行。

由于我们输入的的第一个数字为 1，所有按上面的理论推理程序将跳转到 `phase_3<+90>` 处。继续单步调试完 `phase_3<+81>` 处语句，观察 EDX 中的值，如图 3.14 所示。

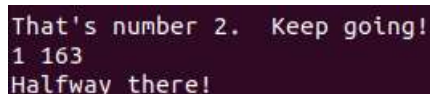
```
edx 0x565565d5 1448437205
```

图 3.14 执行跳转指令时 EDX 的内容

发现 EDX 的值正好对应 `phase_3<+90>` 处语句的地址，说明推理正确，程序即将跳转到 `phase_3<+90>` 处执行。继续分析反汇编语句，看到接下来将输入的第二个数字与十六进制数 `0xa3` 进行比较，若不相等则引爆炸弹，说明答案的第二个数字为答案第一个数字跳转到的对应分支所给定的要比较的十六进制数值。那么当答案第一个数字为 1 时，第二个数字就应该是 `0xa3`，转换为十进制后为 163，因此一组可能的答案就是“1 163”，下面重新执行程序，对得出的这组答案进行验证。

取消所有断点，重新执行程序，在提示输入关卡 3 的答案时，输入“1 163”观察执行结果，如图 3.15 所示。可以看到关卡 3 炸弹成功拆除，说明我们推理过程思路基本正确。

汇编语言程序设计实验报告



```
That's number 2. Keep going!
1 163
Halfway there!
```

图 3.15 关卡 3 成功破解

接下来再验证当输入的第一个数字为 2 到 7，第二个数字为对应分支所给定的数时，炸弹是否也能被成功拆除。根据反汇编语句，我们可以得到当第一个数字为 2 时，第二个数字应为 0x91，即 145，则另一组可能的答案“2 145”。重新执行程序，将改组答案输入到关卡 3 中，发现炸弹也被成功拆除，说明我们的推理完全正确。同理可得，当第一个数字为 3,4,5,6,7，第二个数字为对应分支所给定的数时，炸弹也能被成功拆除。那么得到关卡 3 的所有答案为“1 163”“2 145”“3 573”“4 506”“5 621”“6 327”“7 620”。最后对关卡 3 进行总结，其涉及的主要是条件跳转。

(4) 破解 phase_4:

在 phase_4 处设置断点，先使用 `disassemble phase_4` 指令观察 phase_4 的反汇编语句，如图 3.16 所示。观察到其反汇编语句中同样包含 `scanf` 函数，因此需要采用和关卡 3 相同的方法查看 `scanf` 中参数的格式来确定输入字符串的格式。执行程序，在提示输入关卡 4 的答案时，先任意输入字符串“1 2”，单步执行完 `call __isoc99_sscanf@plt` 语句后，到内存中观察反汇编语句中出现的 `-0x2c81(%ebx)` 中的内容，如图 3.17 所示。


```
(gdb) disassemble phase_4
Dump of assembler code for function phase_4:
0x5655667c <+0>:    push    %ebx
0x5655667d <+1>:    sub     $0x18,%esp
0x56556680 <+4>:    call   0x56556240 <__x86.get_pc_thunk.bx>
0x56556685 <+9>:    add     $0x48df,%ebx
0x5655668b <+15>:   mov     %gs:0x14,%eax
0x56556691 <+21>:   mov     %eax,0xc(%esp)
0x56556695 <+25>:   xor     %eax,%eax
0x56556697 <+27>:   lea     0x4(%esp),%eax
0x5655669b <+31>:   push    %eax
0x5655669c <+32>:   lea     0xc(%esp),%eax
0x565566a0 <+36>:   push    %eax
0x565566a1 <+37>:   lea     -0x2c81(%ebx),%eax
0x565566a7 <+43>:   push    %eax
0x565566a8 <+44>:   push    0x2c(%esp)
0x565566ac <+48>:   call   0x56556140 <__isoc99_sscanf@plt>
0x565566b1 <+53>:   add     $0x10,%esp
0x565566b4 <+56>:   cmp     $0x2,%eax
0x565566b7 <+59>:   jne     0x565566c5 <phase_4+73>
0x565566b9 <+61>:   mov     0x4(%esp),%eax
0x565566bd <+65>:   sub     $0x2,%eax
0x565566c0 <+68>:   cmp     $0x2,%eax
0x565566c3 <+71>:   jbe     0x565566ca <phase_4+78>
0x565566c5 <+73>:   call   0x56556b44 <explode_bomb>
0x565566ca <+78>:   sub     $0x8,%esp
0x565566cd <+81>:   push    0xc(%esp)
0x565566d1 <+85>:   push    $0x5
0x565566d3 <+87>:   call   0x5655663b <func4>
0x565566d8 <+92>:   add     $0x10,%esp
0x565566db <+95>:   cmp     %eax,0x8(%esp)
0x565566df <+99>:   jne     0x565566f3 <phase_4+119>
0x565566e1 <+101>:  mov     0xc(%esp),%eax
0x565566e5 <+105>:  sub     %gs:0x14,%eax
0x565566ec <+112>:  jne     0x565566fa <phase_4+126>
0x565566ee <+114>:  add     $0x18,%esp
0x565566f1 <+117>:  pop     %ebx
0x565566f2 <+118>:  ret
0x565566f3 <+119>:  call   0x56556b44 <explode_bomb>
0x565566f8 <+124>:  jmp     0x565566e1 <phase_4+101>
0x565566fa <+126>:  call   0x56557950 <__stack_chk_fail_local>
End of assembler dump.
```

图 3.16 对 phase_4 的内容进行反汇编

```
(gdb) x /40bc $ebx-0x2c81
0x565582e3:  37 '%' 100 'd' 32 ' ' 37 '%' 100 'd' 0 '\000' 69 'E' 114 'r'
```

图 3.17 到内存中观察相应地址的内容

发现该地址的内容与关卡 3 中该地址的内容相同，说明关卡 4 中 scanf 参数的格式也为接收两个整数，所以输入答案时应输入两个整数。而我们初步测试时正好输入的是字符串“1 2”，符号关卡 4 要求的输入格式，因此我们继续向下执行程序。单步执行完 phase_4<+53> 语句后，观察后面的反汇编语句，发现同样涉及到对 ESP 偏移地址对应内容的操作，所有先到内存中查看 ESP 中的内容，如图 3.18 所示。

```
(gdb) x /40bd $esp
0xffffcf70:  -112  -76   85   86    2    0    0    0
0xffffcf78:    1    0    0    0    0  -20    0   64
```

图 3.18 到内存中观察 ESP 中的内容

发现 ESP+4 对应的是输入的第二个数字，ESP+8 对应的是输入的第二个数字。接下来

汇编语言程序设计实验报告

将 EAX 与 2 进行比较, 若不相等则引爆炸弹。观察到 EAX 的值为 2, 推测这里 EAX 中的内容应该是我们输入的字符串中包含数字的个数, 因此程序得以继续向后执行。接下来程序比较输入的第二个数字的数减 2 后的结果与 2 的大小, 若结果小于等于 2 则跳转到引爆炸弹的语句之后, 反之则不跳转并且引爆炸弹, 说明答案中的第二个数字应小于等于 4。

然后程序将输入的第二个数字与 5 压栈, 作为调用 func4 函数时的参数。调用 func4 函数时, 通过观察 func4 函数的反汇编语句可以看出, 返回值存储在寄存器 EAX 中。回到 phase_4 函数后, 程序将 func4 函数的返回值与输入的的第一个数字进行比较, 若不相等则引爆炸弹。说明输入的的第一个数字应与由第二个数字作为 func4 函数参数得到的返回值相等, 那么单步执行到 phase_4<+95>语句处, 此时将存储于 EAX 中的 func4 函数的返回值与输入的的第一个数字进行比较, 通过观察 EAX 的值, 便能得到对应的答案, 如图 3.19 所示。

eax	0x18	24
-----	------	----

图 3.19 到内存中观察 EAX 中的内容

因此可以得出, 关卡 4 的一组可能的答案为“24 2”。下面将对该答案进行验证。取消所有断点, 执行程序。在提示输入关卡 4 的答案时, 输入字符串“24 2”, 观察到关卡 4 炸弹成功被拆除, 如图 3.20 所示。说明我们的推理思路基本正确。

```
Halfway there!  
24 2  
So you got that one. Try this one.
```

图 3.20 关卡 4 成功破解

根据刚刚的推理思路, 第二个数字的取值还可以是小于等于 4 的任意数值。对应的第一个数字的值只需要把第二个数字作为参数传入 func4 函数后, 观察存储于 EAX 中的返回值即可得出。因此, 我们还能得到若干组可能的答案, 如“36 3”“48 4”等等。将上述几组答案带到关卡 4 中一一进行验证, 发现炸弹均被成功拆除, 说明我们的推理过程完全正确。最后对关卡 4 进行总结, 其涉及到的主要是函数调用, 且该函数为一个递归函数。

(5) 破解 phase_5:

在 phase_5 处设置断点, 先使用 disassemble phase_5 指令观察 phase_5 的反汇编语句, 如图 3.21 所示。

```
(gdb) disassemble phase_5
Dump of assembler code for function phase_5:
0x565566ff <+0>:    push    %esi
0x56556700 <+1>:    push    %ebx
0x56556701 <+2>:    sub     $0x20,%esp
0x56556704 <+5>:    call   0x56556240 <__x86.get_pc_thunk.bx>
0x56556709 <+10>:   add     $0x485b,%ebx
0x5655670f <+16>:   mov     0x2c(%esp),%esi
0x56556713 <+20>:   mov     %gs:0x14,%eax
0x56556719 <+26>:   mov     %eax,0x18(%esp)
0x5655671d <+30>:   xor     %eax,%eax
0x5655671f <+32>:   push    %esi
0x56556720 <+33>:   call   0x56556a0e <string_length>
0x56556725 <+38>:   add     $0x10,%esp
0x56556728 <+41>:   cmp     $0x6,%eax
0x5655672b <+44>:   jne     0x56556782 <phase_5+131>
0x5655672d <+46>:   mov     $0x0,%eax
0x56556732 <+51>:   lea     -0x2da0(%ebx),%ecx
0x56556738 <+57>:   movzbl (%esi,%eax,1),%edx
0x5655673c <+61>:   and     $0xf,%edx
0x5655673f <+64>:   movzbl (%ecx,%edx,1),%edx
0x56556743 <+68>:   mov     %dl,0x5(%esp,%eax,1)
0x56556747 <+72>:   add     $0x1,%eax
0x5655674a <+75>:   cmp     $0x6,%eax
0x5655674d <+78>:   jne     0x56556738 <phase_5+57>
0x5655674f <+80>:   movb    $0x0,0xb(%esp)
0x56556754 <+85>:   sub     $0x8,%esp
0x56556757 <+88>:   lea     -0x2dca(%ebx),%eax
0x5655675d <+94>:   push    %eax
0x5655675e <+95>:   lea     0x11(%esp),%eax
0x56556762 <+99>:   push    %eax
0x56556763 <+100>:  call   0x56556a2c <strings_not_equal>
0x56556768 <+105>:  add     $0x10,%esp
0x5655676b <+108>:  test    %eax,%eax
0x5655676d <+110>:  jne     0x56556789 <phase_5+138>
0x5655676f <+112>:  mov     0xc(%esp),%eax
0x56556773 <+116>:  sub     %gs:0x14,%eax
0x5655677a <+123>:  jne     0x56556790 <phase_5+145>
0x5655677c <+125>:  add     $0x14,%esp
0x5655677f <+128>:  pop     %ebx
0x56556780 <+129>:  pop     %esi
0x56556781 <+130>:  ret
0x56556782 <+131>:  call   0x56556b44 <explode_bomb>
0x56556787 <+136>:  jmp     0x5655672d <phase_5+46>
0x56556789 <+138>:  call   0x56556b44 <explode_bomb>
0x5655678e <+143>:  jmp     0x5655676f <phase_5+112>
0x56556790 <+145>:  call   0x56557950 <__stack_chk_fail_local>
End of assembler dump.
```

图 3.21 对 phase_5 的内容进行反汇编

观察到关卡 5 没有像前几个关卡一样包含对 scanf 函数的调用，而是与第一关具有相同的 string_length 和 strings_not_equal 函数，可以推测关卡 5 要求输入的是一串连续的字符串。通过观察 phase_5<+41>处的语句，调用完 string_length 函数后将 EAX 与 6 进行比较，若不相等则引爆炸弹。推测该函数返回的是输入字符串的长度，且将返回值存储在 EAX 中进行返回，那么我们输入的字符串长度应为 6。

执行程序，当提示输入 phase_5 的答案时，先任意输入六个字符组成的字符串“abcdef”进行初步观察。单步调试完 string_length 函数后，发现反汇编语句中出现了一个循环，该循环共执行 6 次，通过分析可以发现循环依次对输入的字符串的 6 个字符进行操作。下面逐语

汇编语言程序设计实验报告

句分析循环的具体操作，循环先将偏移地址 EBX-0x2da0 传送给了 ECX，单步调试完该操作后，观察存储于 ECX 的偏移地址所对应的值，如图 3.22 所示。

```
(gdb) i r $ecx
ecx          0x565581c4          1448444356
(gdb) x /40bc 0x565581c4
0x565581c4 <array.0>: 109 'm' 97 'a' 100 'd' 117 'u' 105 'i' 101 'e' 114 'r' 115 's'
0x565581cc <array.0+8>: 110 'n' 102 'f' 111 'o' 116 't' 118 'v' 98 'b' 121 'y' 108 'l'
```

图 3.22 到内存中观察 ECX 中偏移地址所对应的值

发现该偏移地址所对应的值是一个存有若干字符的数组。继续观察反汇编语句，可以看到下面通过基址加变址寻址的方式从 ESI 中偏移地址取值传送给 EDX，所以再去观察 ESI 中偏移地址所对应的内容，如图 3.23 所示。

```
(gdb) i r $esi
esi          0x5655b4e0          1448457440
(gdb) x /40bc 0x5655b4e0
0x5655b4e0 <input_strings+320>: 97 'a' 98 'b' 99 'c' 100 'd' 101 'e' 102 'f' 0 '\000' 0 '\000'
```

图 3.23 到内存中观察 ESI 中偏移地址所对应的值

发现该偏移地址所对应的值正好是我们输入的字符串。循环依次取我们输入的每个字符进行操作。第一次循环取的是我们输入的第一个字符‘a’传送给 EDX。接下来的 `and $0xf, %edx` 语句的作用是取 EDX 的值的最低位，由于 EDX 的值对应字符‘a’ASCII 值的十六进制表示 0x61，因此该语句执行完后 EDX 的值变为 0x1。然后再根据 EDX 的值到 ECX 中偏移地址对应的数组中取相应的元素，并传送给 EDX。数组第 1 个元素（数组从第 0 个元素开始）为 a，因此推出该语句执行完后 EDX 的值应为 a 对应的 ASCII 值 0x61。单步调试完该语句，观察 EDX 的值（如图 3.24 所示），发现 EDX 的值与我们的推测一致，说明推理思路基本正确。

```
(gdb) i r $edx
edx          0x61              97
```

图 3.24 到内存中观察 EDX 中的内容

然后程序将得到的 EDX 的值，即数组中的一个字符，存放到 ESP 对应偏移地址中，循环总共会得到 6 个字符，这 6 个字符组成的字符串后续将作为参数传递给 `strings_not_equal` 函数。单步调试完 `phase_5<+88>` 后，观察 EAX 中的内容（如图 3.25 所示），推测应该是 `strings_not_equal` 函数要比较的答案字符串。

```
(gdb) i r $eax
eax          0x5655819a          1448444314
(gdb) x /40bc 0x5655819a
0x5655819a: 102 'f' 108 'l' 121 'y' 101 'e' 114 'r' 115 's' 0 '\000' 0 '\000'
```

图 3.25 到内存中观察 EAX 中的内容

因此，我们可以对关卡 5 进行如下梳理：先输入一个包含 6 个字符的字符串，程序依次取输入的字符串中的每个字符，并取其 ASCII 值十六进制表示下的最低位作为数组下标访问指定的数组，从该数组中获取相应的元素，这样根据输入的字符串总共可以获取 6 个字符，再将获取到的这 6 个字符所组成的字符串与答案字符串“flyers”进行比较，若相同则炸弹拆除成功。可以发现这一系列操作比较类似 C 语言中的指针。

所以，我们现在就需要分别获取答案字符串“flyers”中每个字符在指定数组中对应的下标，得到下标的十六进制表示分别为 9、f、e、5、6、7，然后再根据 ASCII 表找到在十六进制表示下分别以这六个下标作为 ASCII 值结尾的 6 个字符，便是我们输入时的答案字符串。这里我以 yonefg 为例（ASCII 值分别为 0x79、0x6f、0x6e、0x65、0x66、0x67），这

汇编语言程序设计实验报告

6 个字符的结尾满足刚刚的规律, 将其代入关卡 5 进行验证, 发现炸弹被成功拆除 (如图 3.26 所示), 说明我们的推理基本正确。

```
Halfway there!  
yonefg  
So you got that one. Try this one.
```

图 3.26 关卡 5 成功破解

同样地, 我们选取其他满足上述 ASCII 规律的 6 个字符进行测试, 依然能成功拆除炸弹, 说明我们的推理完全正确。最后对关卡 5 进行总结, 其涉及到的主要是指针的机器级表示。

(6) 破解 phase_6:

先在 phase_6 处设置断点, 使用 disassemble phase_6 指令观察 phase_6 的内容, 发现关卡 6 的反汇编语句非常之长, 所以先整体阅读、理解关卡 6 的反汇编代码, 发现关卡 6 总共可以分成若干个部分, 下面逐部分对关卡 6 进行分析:

第一部分 (如图 3.27 所示): 输入六个数字, 判断输入的每个数字是否均小于 6, 且是否两两不相同, 如果不满足则引爆炸弹。

```
0x565567bc <+39>: call 0x56556b79 <read_six_numbers>  
0x565567c1 <+44>: mov %esi,0x18(%esp)  
0x565567c5 <+48>: add $0x10,%esp  
0x565567c8 <+51>: movl $0x0,0x4(%esp)  
0x565567d0 <+59>: mov %esi,%ebp  
0x565567d2 <+61>: jmp 0x565567f7 <phase_6+98>  
0x565567d4 <+63>: call 0x56556b44 <explode_bomb>  
0x565567d9 <+68>: jmp 0x5655680b <phase_6+118>  
0x565567db <+70>: add $0x1,%esi  
0x565567de <+73>: cmp $0x6,%esi  
0x565567e1 <+76>: je 0x565567f2 <phase_6+93>  
0x565567e3 <+78>: mov 0x0(%ebp,%esi,4),%eax  
0x565567e7 <+82>: cmp %eax,(%edi)  
0x565567e9 <+84>: jne 0x565567db <phase_6+70>  
0x565567eb <+86>: call 0x56556b44 <explode_bomb>  
0x565567f0 <+91>: jmp 0x565567db <phase_6+70>  
0x565567f2 <+93>: addl $0x4,0x8(%esp)
```

图 3.27 关卡 6 拆解的第一部分

第二部分 (如图 3.28 所示): 按照输入的 6 个数字的序列对程序内部链表的 6 个结构体结点重新进行排序, 得到重建后的新链表。


```

0x565567f2 <+93>: jmp 0x56556800 <phase_6+110>
0x565567f7 <+98>: addl $0x4,0x8(%esp)
0x565567fb <+102>: mov 0x8(%esp),%eax
0x565567fd <+104>: mov %eax,%edi
0x565567ff <+106>: mov (%eax),%eax
0x56556803 <+110>: mov %eax,0xc(%esp)
0x56556806 <+113>: sub $0x1,%eax
0x56556809 <+116>: cmp $0x5,%eax
0x5655680b <+118>: ja 0x565567d4 <phase_6+63>
0x5655680d <+123>: addl $0x1,0x4(%esp)
0x56556810 <+127>: mov 0x4(%esp),%esi
0x56556814 <+130>: cmp $0x5,%esi
0x56556817 <+132>: jle 0x565567e3 <phase_6+78>
0x56556819 <+137>: mov $0x0,%esi
0x5655681e <+139>: mov %esi,%edi
0x56556820 <+143>: mov 0x1c(%esp,%esi,4),%ecx
0x56556824 <+148>: mov $0x1,%eax
0x56556829 <+154>: lea 0x168(%ebx),%edx
0x5655682f <+157>: cmp $0x1,%ecx
0x56556832 <+159>: jle 0x5655683e <phase_6+169>
0x56556834 <+162>: mov 0x8(%edx),%edx
0x56556837 <+165>: add $0x1,%eax
0x5655683a <+167>: cmp %ecx,%eax
0x5655683c <+169>: jne 0x56556834 <phase_6+159>
0x5655683e <+173>: mov %edx,0x34(%esp,%edi,4)
0x56556842 <+176>: add $0x1,%esi
0x56556845 <+179>: cmp $0x6,%esi
0x56556848 <+181>: jne 0x5655681e <phase_6+137>
0x5655684a <+185>: mov 0x34(%esp),%esi
0x5655684e <+189>: mov 0x38(%esp),%eax
0x56556852 <+192>: mov %eax,0x8(%esi)
0x56556855 <+196>: mov 0x3c(%esp),%edx
--Type <RET> for more, q to quit, c to continue without paging--c
0x56556859 <+199>: mov %edx,0x8(%eax)
0x5655685c <+203>: mov 0x40(%esp),%eax
0x56556860 <+206>: mov %eax,0x8(%edx)
0x56556863 <+210>: mov 0x44(%esp),%edx
0x56556867 <+213>: mov %edx,0x8(%eax)
0x5655686a <+217>: mov 0x48(%esp),%eax
0x5655686e <+220>: mov %eax,0x8(%edx)
0x56556871 <+227>: movl $0x0,0x8(%eax)
0x56556878 <+232>: mov $0x5,%edi
0x5655687d <+242>: jmp 0x56556887 <phase_6+242>

```

图 3.28 关卡 6 拆解的第二部分

第三部分(如图 3.29 所示):通过循环对链表中每个结点结构体中包含的数值进行比较,若不满足前一个数大于后一个数,则引爆炸弹,说明此处是判断链表中每个结点结构体中包含的数值构成的序列是否降序排列。

汇编语言程序设计实验报告

```
0x56556878 <+227>: mov    $0x5,%edi
0x5655687d <+232>: jmp    0x56556887 <phase_6+242>
0x5655687f <+234>: mov    0x8(%esi),%esi
0x56556882 <+237>: sub    $0x1,%edi
0x56556885 <+240>: je     0x56556897 <phase_6+258>
0x56556887 <+242>: mov    0x8(%esi),%eax
0x5655688a <+245>: mov    (%eax),%eax
0x5655688c <+247>: cmp    %eax,(%esi)
0x5655688e <+249>: jge    0x5655687f <phase_6+234>
0x56556890 <+251>: call   0x56556b44 <explode_bomb>
0x56556895 <+256>: jmp    0x5655687f <phase_6+234>
0x56556897 <+258>: mov    0x4c(%esp),%eax
0x5655689b <+262>: sub    %gs:0x14,%eax
0x565568a2 <+269>: jne    0x565568ac <phase_6+279>
```

图 3.29 关卡 6 拆解的第三部分

第四部分(如图 3.30 所示):到内存中观察关卡 6 程序内部原链表的各结点结构体序列,作为降序排序的依据。

0x5655b0cc <node1>:	0x52	0x01	0x00	0x00	0x01	0x00	0x00	0x00
0x5655b0d4 <node1+8>:	0xd8	0xb0	0x55	0x56	0x39	0x02	0x00	0x00
0x5655b0dc <node2+4>:	0x02	0x00	0x00	0x00	0xe4	0xb0	0x55	0x56
0x5655b0e4 <node3>:	0x8f	0x01	0x00	0x00	0x03	0x00	0x00	0x00
0x5655b0ec <node3+8>:	0xf0	0xb0	0x55	0x56	0xbb	0x00	0x00	0x00
0x5655b0f4 <node4+4>:	0x04	0x00	0x00	0x00	0xfc	0xb0	0x55	0x56
0x5655b0fc <node5>:	0xd1	0x00	0x00	0x00	0x05	0x00	0x00	0x00
0x5655b104 <node5+8>:	0x68	0xb0	0x55	0x56	0x00	0x00	0x00	0x00
0x5655b068 <node6>:	0xbb	0x03	0x00	0x00	0x06	0x00	0x00	0x00
0x5655b070 <node6+8>:	0x00	0x00	0x00	0x00	0x8e	0xce	0x21	0x00

图 3.30 原链表各结点结构体序列

通过前三部分的推理和对第四部分原链表各节点结构体序列的观察,我们可以对原链表作出排序,当原链表中各结点编号序列为“6 2 3 1 5 4”时,链表结构体包含的数值构成的序列正好是降序序列(第六个结点 0x03bb、第二个结点 0x0239、第三个结点 0x018f、第一个结点 0x0152、第五个结点 0x00d1、第四个结点 0x00bb)。因此,推测“6 2 3 1 5 4”就是关卡 6 的答案。

取消所有断点,在提示输入关卡 6 的答案时,输入“6 2 3 1 5 4”后观察到关卡 6 炸弹被成功拆除,如图 3.31 所示。说明我们对关卡 6 复杂语句及数据结构的分析完全正确。最后,对关卡 6 进行总结,其涉及到的主要是 C 语言链表、指针和结构体的机器级表示,分析起来有不小的难度。

```
Good work! On to the next...
6 2 3 1 5 4
Congratulations! You've defused the bomb!
```

图 3.31 关卡 6 成功破解

至此,六个关卡便全部破解完毕,炸弹已被我们成功拆除。

3.4 实验小结

本实验是一个大型的反汇编实验,共设计了 6 个阶段,每个阶段对应考察一种不同的程序技术。我通过使用反汇编工具与调试工具,观察程序经过反汇编后的汇编代码,并对其进行分析,推理得出能够使程序顺利运行,不会引爆炸弹的正确输入,从而逐个解决各个阶段的问题。

汇编语言程序设计实验报告

在本次实验中，通过阅读大量的汇编代码，我熟练掌握了 AT&T 格式的汇编语言的各自基本操作，包括从字符串比较、循环、条件分支、递归调用和栈，到链表、指针、结构的灵活运用。为了完成实验目标，必须理解各个阶段的汇编代码的执行流程与计算细节，从而推导出程序期望的正确输入。这给我的逻辑思考、代码阅读能力和对计算机系统基础的理论理解都带来了不小的考验，也花费了我大量的时间，但同时也很大程度上提高了我这些方面的能力，整个实验内容非常丰富，使我收获颇丰。

4 模块化程序设计

4.1 实验内容

采用子程序、多模块等编程技术设计实现一个较为完整的计算机系统运行状态的监测系统。

主要功能为(以任务 1.4 为基础):

(1) 程序执行后,提示输入用户名(用户名请定义成自己的姓名拼音)和密码,如果不正确,提示出错信息后再次重新输入,最多三次出错机会,三次都出错时程序退出。若用户名和密码都正确,则继续后续处理。

(2) 在接下来的处理中,先对 N 组状态信息分别计算 f,并进行分组复制;

(3) 然后将 MIDF 存储区的各组数据在屏幕上显示出来。

(4) 最后,按 R 键重新从“(2)”处执行,按 Q 键退出程序。

要求:

(1) 计算 f、复制每组的数据、显示 MIDF 存储区的各组数据等功能均分别用子程序实现。

(2) 功能(1)(3)(4)用 C 语言程序实现。

(3) 通过实验操作,观察:子程序与主程序之间是如何传递信息的?刚进入堆栈时,堆栈栈顶及之下存放了一些什么信息?执行 CALL 指令及 RET 指令,CPU 完成了哪些操作?若执行 RET 前把栈顶的数值改掉,那么 RET 执行后程序返回到何处?子程序中的局部变量的存储空间在什么位置?如何确定局部变量的地址?访问局部变量时的地址表达式有何特点?

(4) 将汇编源代码至少分解到两个不同的源文件中。

(5) 进一步可观察的内容:

观察模块间的参数的传递方法,包括全局符号的定义和外部符号的引用,若符号名不一致或类型不一致会有什么现象发生?

4.2 任务 4.1 实验过程

4.2.1 实验方法说明

1.准备上机实验环境,对实验用到的软件进行安装、运行,通过试用初步了解软件的基本功能、操作等。

2.先尝试编写 C 语言程序实现实验要求中的功能(1)(3)(4)。

汇编语言程序设计实验报告

- 4.再在任务 1.4 汇编程序的基础上改写汇编源代码，设计出满足实验需求的子程序。
- 3.最后通过 C 语言与汇编语言的混合编程的方法将 C 语言的模块与汇编语言的模块组合到一个工程里，编译连接成一个程序，实现程序的完整功能。
4. 使用 gdb 对混合编程的程序进行调试，观察子程序与主程序之间传递信息的方式及调用子程序过程中堆栈的内容等。
5. 通过调试混合编程的程序，体会与纯粹汇编语言编写的程序的调试过程的差异。

4.2.2 实验记录与分析

编写 C 语言程序，命名为 `main.c`，实现实验要求中的功能（1）（3）（4），如图 4.1 所示。可以发现，C 语言程序在本实验混合编程中作流程控制器的作用。使用指令“`gcc -c -g -m32 -o main.o main.c`”对 C 语言程序进行编译，再使用指令“`gcc -m32 main.o -o main`”对生成的 `main.o` 文件进行链接，进而生成 `main` 可执行二进制文件。

接下来执行可执行文件 `main`，测试功能（1）（3）（4）能否正常运行。在终端输入指令 `./main` 后，观察到屏幕上弹出“请输入用户名和密码”的提示，按照初始设定的用户名（自己的名字拼音“`liushiyan`”）和密码（`123456`）输入后，可以看到程序提示登录成功，并正常打印出了 `MIDF` 一行文字（此时还没有将汇编子程序连接到 C 语言程序中来，只是打印了一串字符串进行功能测试），接着当按下 `R` 键后，`MIDF` 能够重新打印，当按下 `Q` 键后，程序正常退出，如图 4.2 所示。说明我们的 C 语言程序的功能编写正确。

汇编语言程序设计实验报告

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<string.h>

char username[20] = "liushiyan";
char password[20] = "123456";

int main() {
    char username_input[20];
    char password_input[20];
    int times = 0;

    while (strcmp(username, username_input)!=0 || strcmp(password, password_input)!=0) {
        if (times == 3) {
            printf("Failed too many times, the program will exit!\n");
            return 0;
        }
        if (times) {
            printf("User name or password is incorrect, please input again!\n");
        }
        printf("Please input user name and password!\n");
        printf("username: ");
        scanf("%s", username_input);
        printf("password: ");
        scanf("%s", password_input);
        times++;
    }

    printf("Login successfully!\n");

    char input='R';
    while(input=='R'){
        printf("MIDF:\n");
        //cal();
        input=getchar();
        while(!(input=='R' || input=='Q')){
            input=getchar();
        }
    }

    return 0;
}
```

图 4.1 C 语言程序 main.c 源代码

```
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ ./main
Please input user name and password!
username: liushiyan
password: 123456
Login successfully!
MIDF:
R
MIDF:
R
MIDF:
Q
vboxuser@Linux:~/Desktop/computer system experiment/exp4$
```

图 4.2 C 语言程序功能正常

接下来在任务 1.4 源程序的基础上对原有汇编程序进行改写,使其成为能被 C 语言程序调用的子程序,这里我把改写后的汇编文件命名为 cal.s。在声明一个汇编程序为子程序时,在文件的代码段需要加上“.type cal, @function”和“.globl cal”两行指令,其作用是将 cal 汇编程序定义为一个可以被其他程序调用的子程序。在调用子程序时,还需要做到“保护现场”,即子程序调用过程中非必要不能改变主程序原有寄存器的内容,同时还要对堆栈进行

汇编语言程序设计实验报告

管理，从而实现对局部变量的空间分配和全局变量的获取。因此，在子程序开头使用 `push` 指令将子程序涉及到的原有寄存器内容压栈，以便后续恢复，如图 4.3 所示。

```
.section .text
.type cal, @function
.global cal
cal:
pushl %ebp
movl %esp, %ebp
pushl %edi
pushl %esi
pushl %ebx
```

图 4.3 子程序声明及寄存器压栈实现保护现场

在结束对子程序的调用前，即子程序执行 `ret` 指令返回主程序之前，还需要对之前已经保护的现场进行恢复，即将栈中内容依次出栈给相应的寄存器，如图 4.4 所示。这样便达到了保护现场的目的，在参数传递和返回值传递的过程中，也可以使用堆栈来对其进行管理，通过顺序压栈和出栈的方式来在主程序和子程序之间传递信息。需要注意的是，由于栈的特性为“后进先出”，因此出栈时的顺序应与压栈时的顺序相反，避免恢复时寄存器的内容出现错乱。

```
popl %ebx
popl %esi
popl %edi
movl %ebp, %esp
popl %ebp
ret
```

图 4.4 寄存器出栈实现恢复现场

根据实验要求，在子程序中还需要实现对 MIDF 存储区中内容的打印。我们可以通过一个循环的方式按照状态信息组的相应格式对 MIDF 存储区中每个状态信息组的内容进行打印，如图 4.5 所示。打印过程调用了 C 函数的 `printf` 函数，其参数通过压栈的方式进行传递，这里总共两个参数，先压栈的是要打印的实际数值，后压栈的是要打印的字符串，其中数值用占位符“`%d`”代替。通过 `call printf` 指令便能够将传入的参数打印在屏幕上。要打印的字符串定义在数据段中，如图 4.6 所示。注意在字符串末尾要手动加上“`\0`”，否则在打印时会出现内存溢出，导致意料之外的错误。

```
mov $0, %esi
lop2print:
add $9, %esi
pushl MIDF(%esi)
push $d
call printf
add $4, %esi
pushl MIDF(%esi)
push $d
call printf
add $4, %esi
pushl MIDF(%esi)
push $d
call printf
add $4, %esi
pushl MIDF(%esi)
push $d
call printf
add $4, %esi
cmp midfn, %esi
jne lop2print
```

图 4.5 调用 printf 函数对存储区内容进行打印

```
d: .ascii "%d\n\0"
```

图 4.6 要打印的字符串在数据段的定义

最后，我们在子程序数据段总共创建十组状态信息组数据，并设计其中一组数据刚好满足处理结果 $f=100$ （如图 4.7 所示），这样就能使该组数据最后存储于 MIDF 存储区中，便于后续测试时作为参考。

```
n: .long 10
lowfn: .long 0
midfn: .long 0
highfn: .long 0

MSG:
.fill 225,1,1
.byte 0,0,0,0,0,0,0,1,0
.long 0
.long 12800
.long 100
.long 0
```

图 4.7 子程序数据段中初始状态信息组定义

这样我们的汇编子程序就改写完毕了，下面要在 C 语言程序中增加对该子程序的调用。方法是在 C 程序的 main 函数外部，先声明对汇编子程序 cal 函数的引用，即使用“void cal();”语句（如图 4.8 所示），然后即可在 main 函数体中调用 cal 函数（如图 4.9 所示），实现主程序对子程序的调用，从而通过混合编程的方式实现相应的功能。

```
void cal();
```

图 4.8 主程序中引用子程序

汇编语言程序设计实验报告

```
char input='R';
while(input=='R'){
    printf("MIDF:\n");
    cal();
    input=getchar();
    while(!(input=='R' || input=='Q')){
        input=getchar();
    }
}

return 0;
```

图 4.9 主程序 main 函数体中调用子程序

汇编程序和 C 语言程序均编写完成后,接下来对汇编程序进行汇编。使用指令“as -g --32 -o cal.o cal.s”对汇编源文件进行汇编,没有发生异常,说明我们汇编程序初步编写正确。再使用指令“gcc -c -g -m32 -o main.o main.c”对 C 语言程序进行编译,也没有发生异常,说明我们 C 语言程序初步编写正确。最后对生成的两个.o 文件,即可重定位文件进行链接,从而生成可执行的二进制文件。按照实验指导书中的方法,使用指令“gcc -m32 main.o cal.o -o main”对两个可重定位文件进行链接,发现终端中弹出警告提示,如图 4.10 所示。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ gcc -m32 main.o cal.o -o main
/usr/bin/ld: cal.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
```

图 4.10 程序链接过程弹出警告提示

通过上网查阅资料,我了解了 PIE 是 gcc 编译器的功能选项,作用于程序(ELF)编译过程中。是一个针对代码段(.text)、数据段(.data)、未初始化全局变量段(.bss)等固定地址的一个防护技术,如果程序开启了 PIE 保护的话,在每次加载程序时都变换加载地址。而当我们不需要编译器开启 PIE 保护时,在构建可执行文件时将-no-pie 命令添加到指令结尾,如图 4.11 所示。再次执行构建指令,发现这次没有任何报错或警告,说明我们的混合编程过程基本正确。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ gcc -m32 main.o cal.o -o main -no-pie
vboxuser@Linux:~/Desktop/computer system experiment/exp4$
```

图 4.11 关闭 PIE 后再次构建可执行文件

接下来就要验证混合编程程序的功能是否正确。使用“./main”指令执行生成好的可执行文件,输入正确的用户名和密码后,发现终端中不出所料地打印出了我们想要的 MIDF 存储区的结果,如图 4.12 所示。说明我们子程序和主程序的功能编写完全正确,实现了主程序对子程序的调用,达到了通过混合编程解决实际问题的目的,顺利完成了实验对于模块化程序设计的要求。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ ./main
Please input user name and password!
username: liushiyan
password: 123456
Login successfully!
MIDF:
0
12800
100
100
```

汇编语言程序设计实验报告

图 4.12 终端中成功打印出 MIDF 状态信息组

通过混合编程，观察以下几个方面：

(1) 子程序与主程序之间是如何传递信息的？

由刚才对子程序改写时的分析可以得出，主程序通过参数压栈的方式将参数传递给子程序，子程序通过出栈的方式将参数从栈中取出；同样地，子程序也可以通过将返回值压栈的方式将参数返回给主程序，还可以通过寄存器传递参数，这时该寄存器在保护现场的过程中就不需要对其进行保护和恢复。

(2) 刚进入堆栈时，堆栈栈顶及之下存放了一些什么信息？

刚进入堆栈时，到内存中对堆栈进行观察可以发现，堆栈栈顶存放的是返回到主程序后将要执行语句的地址，这个地址在子程序调用 RET 指令时会出栈给 EIP。堆栈栈顶之下存放的则是主程序传入的若干参数，他们由主程序进行压栈存放在堆栈中，从而实现主程序向子程序传递信息。

(3) 执行 CALL 指令及 RET 指令，CPU 完成了哪些操作？

通过对 CALL 指令及 RET 指令的单步调试可以发现，执行 CALL 指令时，CPU 将主程序下一条即将执行的指令地址压栈，并将子程序首址传送给 EIP，此时程序就将跳转到子程序处执行，这样就达到了调用子程序的目的。执行 RET 指令时，CPU 将事先压栈的主程序下一条即将执行的指令地址出栈传送给 EIP，此时程序就将跳转到主程序调用完子程序处的位置执行，这样就达到了从子程序结束调用返回主程序的目的。

(4) 若执行 RET 前把栈顶的数值改掉，那么 RET 执行后程序返回到何处？

由上述分析可知，RET 执行后程序返回到栈顶数值地址对应的语句处。因此可能会造成程序执行的混乱，导致发送意料之外的错误。

(5) 子程序中的局部变量的存储空间在什么位置？如何确定局部变量的地址？访问局部变量时的地址表达式有何特点？

通过调用子程序时对内存中堆栈内容的观察可以发现，子程序中的局部变量存储在栈底之上，也就是由高字节的栈底向低字节的栈顶扩展进行存储。

局部变量的地址可以通过栈底的位置和局部变量所占空间字节数进行确定，例如该局部变量为子程序中定义的第一个局部变量，且是占 4 个字节的一个整型数，那么该局部变量则存储在栈底上方（即向低字节方向）4 字节处，即该局部变量的地址比栈底地址少 4 个字节，且向低字节方向扩展共 4 个字节的空间。

访问局部变量时的地址表达式通常是采用变址寻址或基址加变址寻址的方式，一般为 EBP 减去地址偏移量的形式，如上述的整型局部变量的地址表达式为 `-0x4(%ebp)`，若此时再定义第二个局部变量，则该局部变量的地址表达式就为 `-0x8(%ebp)`。

进一步可以观察模块间的参数的传递方法，包括全局符号的定义和外部符号的引用。若

汇编语言程序设计实验报告

符号名不一致或类型不一致，如我将 main.c 中的 cal 符号修改为 bal，单独编译时没有发生异常，而在将 cal.o 与 main.o 进行链接时观察到发生引用未定义错误，如图 4.13 所示。说明了链接器对各可重定位文件进行重定位时，全局符号的定义和外部符号的引用符号名和类型必须一致。

```
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ gcc -c -g -m32 -o main.o main.c
vboxuser@Linux:~/Desktop/computer system experiment/exp4$ gcc -m32 main.o cal.o -o main -no-pie
/usr/bin/ld: main.o: in function `main':
/home/vboxuser/Desktop/computer system experiment/exp4/main.c:36: undefined reference to `bal'
collect2: error: ld returned 1 exit status
```

图 4.13 全局符号定义和外部符号引用符号名不一致时发生错误

4.3 实验小结

本次实验我主要掌握了 C 语言调用汇编程序，即主程序调用子程序的方法，和模块化子程序的设计方法。通过调试混合编程的程序，体会到了与纯粹汇编语言编写的程序的调试过程的差异。通过本次实验，我还明白了不同的编程语言是可以协同解决一个问题的，而且可以利用不同语言的特点来更好地解决问题；利用汇编语言的知识，能够更好地理解高级语言的内部处理原理与策略，为编写更好的 C 语言程序、用好 C 编译器提供支持。

此外，我还掌握了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，掌握了模块化程序的设计方法、汇编语言程序与 C 语言程序混合编程的方法，理解了模块之间的信息传递与组装的基本方法，收获颇丰。

然而，在设计和调试混合编程的程序的过程中，仍遇到了大大小小的问题，例如编译链接过程发生错误、环境不兼容等等，但最终都通过上网查阅资料或询问老师同学的方式得到了较好的解决。此外，一开始在对子程序进行设计的过程中，由于对堆栈的概率及运行原理了解得不是特别清楚，所以产生了很多意想不到的 BUG。然而在 DEBUG 的过程中，我不断学习和巩固子程序、堆栈、参数传递等混合编程的概念，进而一步步对混合编程、模块化程序设计的模式有了更深的理解，不失为一个很好的学习过程。并且，在实验最后很好地解决了一开始难以解决的 BUG 和诸多问题，成功地编写出了满足实验要求的程序，较好地达到了本次试验任务的要求。

希望通过本次实验对所学汇编和混合编程的知识进行总结和巩固，并对尚未熟悉的知识点进行学习和掌握，在日后的学习中能对其有更好的理解与灵活的运用，不断提高自己对汇编代码的理解能力与设计能力，加深对主程序与子程序之间调用与信息传递工作原理的理解与自主运用。

5 中断处理

5.1 实验目的和要求

- (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；

5.2 实验内容

任务 5.1：利用中断实现实时时间显示。

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。

5.3 任务 5.1 实验过程

5.3.1 设计思想

任务 4.1 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。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。

汇编语言程序设计实验报告

中断处理程序设计包含 3 个环节：执行原中断程序，获取时间信息并显示。

（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。

5.3.2 流程图

图5.1 是任务5.1 的主流程处理程序流程图。

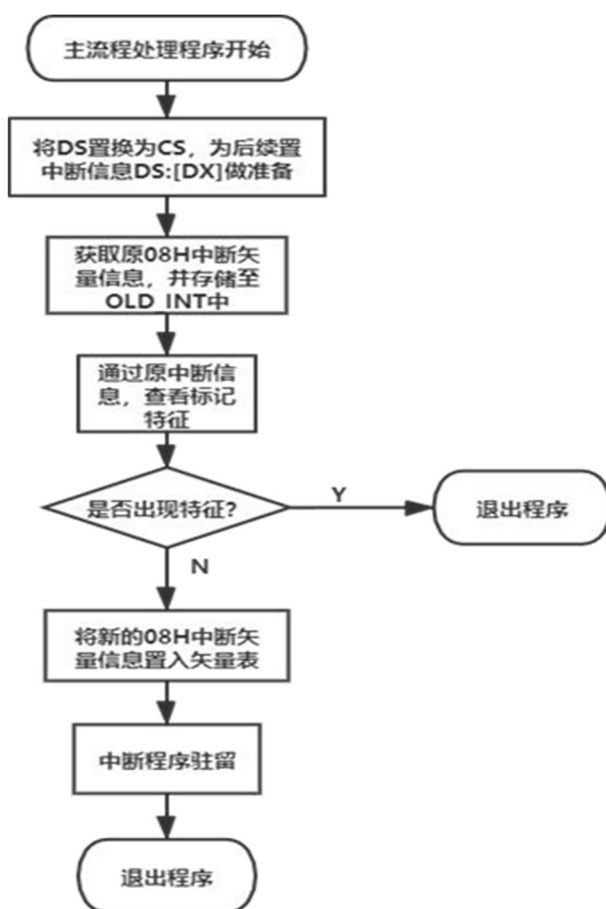


图 5.1 主流程处理程序流程图

5.3.3 源程序

```
.386
CODE SEGMENT USE16
ASSUME CS:CODE,DS:CODE,SS:STACK
;新的 INT 08H 使用变量
... ..
flag DW '#'
;新的 INT 08H 代码
NEW08H PROC FAR
... ..
NEW08H ENDP
;取时间子程序
GET_TIME PROC
... ..
GET_TIME ENDP
;初始化（中断处理程序的安装）及主程序
BEGIN:
... ..
MOV SI,BX
SUB SI,2    ;获取标记特征位置地址
CMP WORD PTR [SI],'#'    ;比较标记特征
JZ EXIT
... ..
MOV AH,02H
MOV DL,'T'
INT 21H;判断是否重复安装，安装时显示 T
MOV DX,OFFSET BEGIN+15
MOV CL,4
SHR DX,CL
ADD DX,10H
MOV AL,0
MOV AH,31H
INT 21H;中断处理程序驻留
EXIT:
MOV AH,4CH
INT 21H ;退出
CODE ENDS
END BEGIN
```

汇编语言程序设计实验报告

5.3.4 实验步骤

1. 准备上机实验环境，使用Notepad进行程序编写,使用 DOSBox 下调试、运行。
2. 编写程序，调试通过后运行，在指定位置正确显示时间信息。
3. 运行其他程序，查看中断程序是否驻留正常。
4. 在TD 下，再次运行该程序，查看是否重复安装中断程序。
5. 在TD 下观察中断矢量表，观察某个中断处理程序代码。
6. 在TD 下调试中断处理程序。

5.3.5 实验记录与分析

1.实验环境条件： 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz, 16G 内存;
WINDOWS 11 下 DOSBox0.73-3 TD.EXE。

2.运行程序，首先进行安装中断处理程序，观察结果，如图5.2所示。发现终端中显示了信息T，说明第一次安装中断处理程序运行成功。



图5.2 首次安装中断处理程序结果

3.运行其他程序，观察结果；再次运行安装程序，观察结果（如图5.3所示）发现终端中没有输出安装检验显示T，说明了设置标记特征成功，通过这种方式能够达到避免重复多次安装中断处理程序的目的。

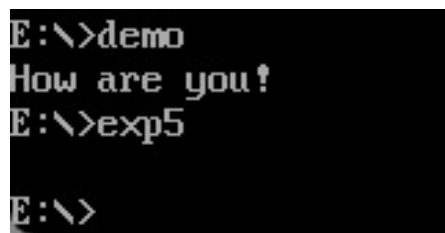


图5.3 再次安装中断处理程序结果

4.安装完中断处理程序后，在TD中观察中断矢量表。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中断矢量表内存如图5.4所示，运行安装程序后的中断矢量表内存如图5.5所示，发现安装中断处理前后中断矢量表中矢量信息发生了变化。

```
fs:0008 08 00 70 00 08 00 70 00
fs:0010 08 00 70 00 60 10 00 F0
fs:0018 60 10 00 F0 60 10 00 F0
fs:0020 A5 FE 00 F0 D6 0C 11 08
```

图5.4 运行安装程序前的中断矢量表

```
fs:0008 08 00 70 00 08 00 70 00
fs:0010 08 00 70 00 60 10 00 F0
fs:0018 60 10 00 F0 60 10 00 F0
fs:0020 11 00 A2 01 D6 0C 37 08
```

图5.5 运行安装程序后的中断矢量表

5.查询第二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 5.6 所示。从访问结果可以看出,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。说明实验基本要求已经达成。

cs:0011 9C	pushf
cs:0012 2EFF1E0B00	call cs:far [000B]
cs:0017 2EFE0E0000	dec cs:byte ptr [000B]
cs:001C 0F840100	je 0021
cs:0020 CF	iret
cs:0021 2EC606000012	mov cs:byte ptr [000B], 12
cs:0027 FB	sti
cs:0028 60	pusha
cs:0029 4F	push 4F

图 4.6 查询中断矢量表访问中断处理程序

5.4 实验小结

此次实验完成了中断处理程序的置换与驻留。由于是首次接触此类程序编写,进一步加深了我对汇编程序的认知与理解。尤其对于中断处理,有了更深刻的认识,并且也逐渐掌握了对中断的应用。通过此次实验,我充分认识到汇编语言的功能强大,主要体现在其对系统中断功能的修改上。

本次实验任务是对中断程序置换与驻留的有关处理。进一步掌握和实操了主程序安装与驻留的细节与中断处理程序的细节,并且理解了 16 位程序的关键指令 INT 的使用方式。对中断处理进行置换时,包含两个关键过程,安装与驻留。安装前,应该判断中断程序是否重复安装。由于代码驻留的特殊性,驻留后无法直接通过变量查询驻留单元偏移地址,只能通过中断矢量

汇编语言程序设计实验报告

表中信息查找。故可在指向位置的前一个存储单元处，放置一个标记特征值‘#’，每次读取中断矢量表时读取其指向空间的前一个单元，若发现该标记则表示重复安装，则直接退出安装程序。另外，INT 指令的用法十分丰富，安装与驻留都使用 INT 21H 指令操作，并且可通过调用号进一步采取对应操作。而在程序驻留阶段，不仅需要驻留中断处理程序的全部程序段内容，还需包括程序段前缀的 100H 个字节的内容。

而在进行中断处理程序的编写时，先进行标志寄存器的保护，此次实验需在原中断的基础上进行操作，故还需完成原功能，此时则需要指定地址偏移量是在 CS 的基础上进行偏移，必须加前缀 CS。在执行置换的中断处理程序时，必须先开中断，再进行后续操作。为避免破坏寄存器数值，还需进行现场保护。

6 Linux 和鲲鹏环境编程

6.1 实验目的和要求

- (1) 了解 ARM + Linux 环境下程序设计的特点及配套的开发工具；
- (2) 观察并理解 80X86 和 ARM 下，不同的“指令集体系结构”的基本特点。

6.2 实验内容

任务 6.1 安装 QEMU 等环境，编译执行示例程序。

需要编译执行的程序包括：

(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。

查阅华为鲲鹏服务器所采用的 CPU（即 ARMv8 系列）的汇编语言编程资料，体会与 80X86 体系的异同。主要关注 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程、开发环境等方面。

6.3 任务 6.1 实验过程

6.3.1 实验方法说明

1. 准备上机实验环境，对实验用到的软件进行安装、运行，通过试用初步了解软件的基本功能、操作等。

2. 安装 QEMU 等环境，编译执行示例程序：“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），“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。观察执行结果。

3. 查阅华为鲲鹏服务器所采用的 CPU（即 ARMv8 系列）的汇编语言编程资料，体会与 80X86 体系的异同。

4. 主要从 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程、开发环境等方面对不同架构的指令集体系结构和编程环境进行比较和记录。

6.3.2 实验记录与分析

在 QEMU 上用基于 linux 内核的 openEuler 操作系统让我们得以在同一台电脑上体验不

汇编语言程序设计实验报告

```
"time.c" 22L, 505C written
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memorycopy time is 185259381 ns
[root@localhost ~]#
```

图 6.4 ARM 虚拟环境下程序运行结果

```
"copy21.s" 12L, 133C written
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
memorycopy time is 49182205 ns
[root@localhost ~]#
```

图 6.5 ARM 虚拟环境下程序运行结果

从上述几个程序的运行结果可以看出，测试内存拷贝函数的执行时间在不断优化，说明我们的程序编译执行过程正确。通过以上几个示例程序的执行和编译，我对 ARM + Linux 环境下程序设计的特点有了进一步的认识，并对其相配套的开发工具有了初步的了解和尝试。

在编译执行完示例程序之后，我还使用了 QEMU 上的 gdb 调试工具对程序进行了简单调试的尝试，如图 6.6 和图 6.7 所示。

QEMU		
Machine View		
x1	0x0	0
x2	0xffffffff7f90000	281474842034176
x3	0x3b9ac9ff	999999999
x4	0x21f02	139010
x5	0x317f	12671
x6	0x113889f0000000	4847239692288000
x7	0x629afb2c	1654324012
x8	0x17	23
x9	0xb89b48f50c	792879232268
x10	0x0	0
x11	0x8000000	134217728
x12	0xffffffff9d653653	-1654311341
x13	0xffffffff7ff0000	281474842427392
x14	0xffffffff7e14908	281474840480008
x15	0xffffffff7e071f8	281474840424952
x16	0xffffffff7ee6ed0	281474841341648
x17	0x420000	4325376
x18	0x694	1684
x19	0x400748	4196168
x20	0x0	0
x21	0x400580	4195712
x22	0x0	0
--Type <RET> for more, q to quit, c to continue without paging--c		
x23	0x0	0
x24	0x0	0
x25	0x0	0
x26	0x0	0
x27	0x0	0
x28	0x0	0
x29	0xfffffffff9d0	281474976709072
x30	0x4006d8	4196056
sp	0xfffffffff9d0	0xfffffffff9d0
pc	0x4006d8	0x4006d8 <main+100>
cpsr	0x1000	[EL=0]
fpsr	0x0	0
fpcr	0x0	0
(gdb)		

图 6.6 gdb 调试功能部分展示

```

QEMU
Machine View
15 clock_gettime(CLOCK_MONOTONIC, &t1);
16 memcpy(dst, src, len1);
17 clock_gettime(CLOCK_MONOTONIC, &t2);
18 printf("memcpy time is %llu ns\n", t2.tv_nsec-t1.tv_nsec);
19 return 0;
(gdb) x/20w &t1
0xfffffffff9f0: -136167644 65535 4196168 0
0xfffffffffa00: 0 59999999 0 0
0xfffffffffa10: 4195764 0 0 0
0xfffffffffa20: 0 0 4195712 0
0xfffffffffa30: 0 0 4195712 0
(gdb) n
16 memcpy(dst, src, len1);
(gdb) info args
No arguments.
(gdb) x/20w &t1
0xfffffffff9f0: 12671 0 582899600 0
0xfffffffffa00: 0 59999999 0 0
0xfffffffffa10: 4195764 0 0 0
0xfffffffffa20: 0 0 4195712 0
0xfffffffffa30: 0 0 4195712 0
(gdb) info locals
t1 = {tv_sec = 12671, tv_nsec = 582899600}
t2 = {tv_sec = 0, tv_nsec = 281474976709136}
i = 59999999
j = <optimized out>
(gdb) info frames
Undefined info command: "frames". Try "help info".
(gdb) info frame
Stack level 0, frame at 0xfffffffffa10:
pc = 0x4006d8 in main (time.c:16); saved pc = 0xfffff7e23f60
source language c.
Arglist at 0xfffffffff9d0, args:
Locals at 0xfffffffff9d0, Previous frame's sp is 0xfffffffffa10
Saved registers:
x29 at 0xfffffffff9d0, x30 at 0xfffffffff9d8
(gdb)
  
```

图 6.7 gdb 调试功能部分展示

从对 gdb 的初步尝试过程中，我发现 gdb 的功能还是十分强大的，可以像 Visual Studio 中一样设置断点，单步调试，还可以输入指令实时查看当前运行到的代码段部分，也可以显示出当前函数中的各个局部变量的值和栈帧情况，寄存器中的值也能同时以十六进制和十进制显示。然而美中不足的是，由于没有图形界面且不能查看过早的操作，为程序的调试带来了些许不便。

接下来观察并理解 80X86 和 ARM 下，不同的“指令集体系结构”的基本特点。通过上网查阅有关资料，我了解到 80X86 和 ARM 两个不同的指令集体系结构的基本特点，在架构目标、指令集、寄存器、内存访问和扩展性等方面存在明显的差异，主要有如下几点：

- 1.架构目标：ARMv8 和 80x86 体系的目标不同。ARMv8 旨在提供高效的能耗比，而 80x86 体系则更注重性能。
- 2.指令集：ARMv8 和 80x86 体系使用的指令集不同。ARMv8 使用的是精简指令集(RISC)，而 80x86 体系使用的是复杂指令集(CISC)。
- 3.寄存器：ARMv8 和 80x86 体系使用的寄存器数量和类型不同。ARMv8 使用较少的寄存器，而 80x86 体系则使用更多的寄存器。
- 4.内存访问：ARMv8 和 80x86 体系对内存的访问方式也不同。ARMv8 通常使用内存映射的方式来访问内存，而 80x86 体系则使用专门的内存访问指令。
- 5.扩展性：ARMv8 和 80x86 体系的扩展性也不同。ARMv8 通常使用较小的核心来构建较大的系统，而 80x86 体系则使用多个核心来构建较大的系统。

在了解了两个不同指令集体系结构的基本特点后，我主要关注 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程、

汇编语言程序设计实验报告

开发环境等方面，查阅华为鲲鹏服务器所采用的 CPU（即 ARMv8 系列）的汇编语言编程资料，体会与 80X86 体系的异同，并从上述不同方面得到了如下几点比较：

1.CPU 内寄存器

ARMV8：寄存器数量较多，均为通用寄存器，可用于多种数据类型。

80X86：寄存器数量较少，分为通用寄存器、段寄存器和专用寄存器。其中通用寄存器数量有限，需要频繁使用栈进行内存访问。

2.段的定义方法

ARMV8：采用基于寄存器的寻址方式，不使用段的概念。

80X86：采用基于段的寻址方式，将内存划分为不同的段，每个段有一个段基址和一个段限长，通过段寄存器进行访问。

3.指令语句及格式的特点

ARMV8：指令格式简洁明了，操作码较短，易于实现流水线操作。

80X86：指令格式较为复杂，操作码较长，需要多个时钟周期才能完成一条指令。

4.子程序调用的参数传递与返回方法

ARMV8：参数通过寄存器传递，返回值也存储在寄存器中。可以使用多个寄存器进行参数传递和返回值存储。

80X86：参数通过栈传递，返回值存储在专用寄存器或内存中。需要频繁使用栈进行内存访问。

5.与 C 语言混合编程

ARMV8：支持 C 语言编程，可以使用 C 语言进行大部分应用程序开发。与汇编语言混合编程也较为方便。

80X86：同样支持 C 语言编程，但由于指令集和寻址方式的差异，与汇编语言混合编程较为复杂。

6.开发环境

ARMV8：拥有丰富的开发工具和支持，包括编译器、调试器、仿真器等。开发环境较为完善。

80X86：同样拥有较多的开发工具和支持，但由于历史原因和架构差异，部分工具可能存在兼容性问题。同时，由于架构的复杂性，开发难度相对较高。

将以上几个方面的比较通过表格的方式直观展示出来，如表 6.1 所示。

表 6.1 ARMv8 与 80X86 体系多方面比较

	ARMv8	80X86
CPU 内寄存器	寄存器数量较多，通用寄存器可用于多种数据类型	寄存器数量较少，通用寄存器数量有限，需要频繁使用栈进行内存访问
段的定义方法	不使用段的概念，基于寄存器的寻址方式	采用基于段的寻址方式，将内存划分为不同的段，通过段寄存器进行访问

汇编语言程序设计实验报告

	ARMv8	80X86
指令语句及格式的特点	指令格式简洁明了，操作码较短，易于实现流水线操作	指令格式较为复杂，操作码较长，需要多个时钟周期才能完成一条指令
子程序调用的参数传递与返回方法	参数通过寄存器传递，返回值也存储在寄存器中，可以使用多个寄存器进行参数传递和返回值存储	参数通过栈传递，返回值存储在专用寄存器或内存中，需要频繁使用栈进行内存访问
与 C 语言混合编程	支持 C 语言编程，可以使用 C 语言进行大部分应用程序开发，与汇编语言混合编程也较为方便	支持 C 语言编程，但由于指令集和寻址方式的差异，与汇编语言混合编程较为复杂
开发环境	拥有丰富的开发工具和支持，包括编译器、调试器、仿真器等，开发环境较为完善	同样拥有较多的开发工具和支持，但由于历史原因和架构差异，部分工具可能存在兼容性问题，同时开发难度相对较高

总的来说，ARMV8 和 80X86 体系在 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程和开发环境等方面存在明显的差异。这些差异使得两种体系在性能、功耗、扩展性等方面具有各自的优势和适用场景。

6.4 实验小结

本次实验我使用以前从未使用过的 QEMU 环境进行编程实验，通过几个示例程序的编译和执行，我对 ARMv8 指令集结构体系有了更加深刻的认识。在对 ARM 虚拟环境下的开发工具有了初步的尝试和体验后，我将其与熟悉的 80X86 环境进行了比较，通过上网查阅资料和阅读相关汇编语言编程资料后，我对 ARMv8 和 80X86 两个不同的指令集结构体系进行了多方面的对比和总结，在这一系列的比较、总结学习中我对两个指令集结构体系有了更新、更全面的认识体会，收获颇丰。

对 ARMv8 与 80x86 两个指令集结构体系的对比，着重在段的定义方法、子程序参数传递与返回方法、开发环境这三个方面进行对比。发现段的定义方式中，ARM 采用相对独立的指令或数据序列的程序段组成程序代码段的划分，通过伪指令进行代码段与数据段的区分。在子程序的调用中，ARM 则直接通过跳转指令进行跳转到子程序空间，通过修改 PC 值，进行子程序的返回，且传参则直接通过寄存器进行实参传递。而 ARMv8 的开发环境是基于 ARM 平台的华为鲲鹏云主机、openEuler20.03 操作系统，以及 gcc 软件支持的。且 ARM 采用的是 RISC 精简指令集，可将数据线和指令线分离。

通过此次实验，除了收获到上述知识外，也让我感悟到学无止境的道理。在不同环境下编写程序，可以达到不同的要求与目的，如果只了解一两个环境远远无法满足程序的用户要求。在掌握更多的环境后，才能够让程序最优化，才能够实现更多程序功能。

汇编语言程序设计实验报告

结合《计算机系统基础实验》的以上六个实验，通过对 DOSBox 下的编程环境、WINDOWS 下的 VS、支持 ARM 的 gcc 进行比较，可以发现 DOSBox 下环境为实方式下的虚拟环境，同时使用可视化调试 TD 方便了调试，但由于程序功能较为简单，操作也较不方便，故其更适合用于理解汇编底层运行方式，如中断、驻留等。WINDOWS 下的 VS 的功能十分强大，也满足了大部分的实验要求，通过可视化窗口的提供极大提高了编程效率，如反汇编窗口、内存窗口的提供等。用 VS 实现实验大部分功能的效率是优于 DOSBox 的，但正因其编译能力的强大，对于理解汇编连接软硬件层面的作用体现不是很多。最后则是 ARM 的 gcc，通过了解实验手册的相关功能，发现其是介于 DOSBox 和 VS 作用区间的环境，保留了软硬件的链接功能，也保留了底层语言与高级语言的结合功能。虽然在指令层面进行了精简化处理，使得编程可能有些困难，但提高了 CPU 的运行效率，使其作用区间更偏向于开发对应环境下的程序。

三者各有优劣，关键在于对它们作用区间的把握，“工欲善其事，必先利其器”说的就是这样的道理，只有了解各个工具的使用方式，才能在编程工作中如鱼得水。

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
- [3]汇编课程组. 《汇编语言程序设计实践》任务书与指南, 2023