

# 华中科技大学

## 课程实验报告

课程名称： 汇编语言程序设计实践

专业班级： 计算机科学与技术 202003 班

学 号： U202015360

姓 名： 胡沁心

指导教师： 李海波

实验时段： 2022 年 3 月 7 日~4 月 29 日

实验地点： 东九 A314

### 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：2022.6.6

### 实验报告成绩评定：

一（50 分）	二（35 分）	三（15 分）	合计（100 分）

指导教师签字：

日期：

# 目录

一、程序设计的全程实践 .....	1
1.1 目的与要求 .....	1
1.2 实验内容 .....	1
1.3 内容 1.3 的实验过程 .....	1
1.3.1 设计思想 .....	1
1.3.2 流程图 .....	3
1.3.3 源程序 .....	7
1.3.4 实验记录与分析 .....	9
1.4 内容 1.2 的实验过程 .....	13
1.4.1 实验方法说明 .....	13
1.4.2 实验记录与分析 .....	14
1.5 小结 .....	15
二、利用汇编语言特点的实验 .....	16
2.1 目的与要求 .....	16
2.2 实验内容 .....	16
2.3 实验过程 .....	16
2.3.1 实验方法说明 .....	16
2.3.2 实验记录与分析 .....	16
2.4 小结 .....	21
三、工具环境的体验 .....	23
3.1 目的与要求 .....	23
3.2 实验过程 .....	23
3.2.1 WINDOWS10 下 VS2019 等工具包 .....	23
3.2.2 DOSBOX 下的工具包 .....	26
3.2.3 QEMU 下 ARMv8 的工具包 .....	27
3.3 小结 .....	29
参考文献 .....	30

## 一、程序设计的全程实践

### 1.1 目的与要求

- 1.掌握汇编语言程序设计的全周期、全流程的基本方法与技术；
- 2.通过程序调试、数据记录和分析，了解影响设计目标和技术方案的多种因素。

### 1.2 实验内容

内容 1.1：采用子程序、宏指令、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，给出完整的建模描述、方案设计、结果记录与分析。（依托：任务 3.1）

内容 1.2：初步探索影响设计目标和技术方案的多种因素，主要从指令优化对程序性能的影响，不同的约束条件对程序设计的影响，不同算法的选择对程序与程序结构的影响，不同程序结构对程序设计的影响，不同编程环境的影响等方面进行实践。

### 1.3 内容 1.3 的实验过程

#### 1.3.1 设计思想

##### 1.内存分配

```
用户名: user_name  db 'huqinxin',0
密码字符串: pw  db '12345678',0
需要输入的用户名: str1  db 10 dup(0)
需要输入的密码: str2  db 10 dup(0)
需要输入的指令: str9  db 10 dup(0)
提示字符串:
```

```
str3  db 'wrong user name!',0ah,0dh,0
str4  db 'wrong password!',0ah,0dh,0
str5  db 'input user name:',0
str6  db 'input password:',0
str7  db 'exit!',0
str8  db 'input order:',0
```

data 10 组数据(见源程序)

储存区:

```
lowf  samples 10 dup(<>)
midf  samples 10 dup(<>)
highf samples 10 dup(<>)
```

##### 2.数据结构

```
samples struct
    samid  db 9 dup(0)
```

# 汇编语言程序设计实验报告

```
sda dd ?  
sdb dd ?  
sdc dd ?  
sf dd ?
```

samples ends

## 3. 算法设计

程序分为两个模块，main.asm 包含 main 函数和宏定义，func.asm 包含计算、复制、打印功能的函数。

函数及宏定义说明：

### 1、打印函数

函数声明：print proc

- 1) ebx 赋值为 0，作为偏移量
- 2) 进入循环，如果 samid 不为空，按 samid，sda，sdb，sdc，sf 的顺序输出整组数据
- 3) 遍历完十组 samid 后退出循环

### 2、复制函数

函数声明：copy proc stdcall arg: dword

- 1) 形参 arg 为计算函数的计算结果。ebx 为当前这组数据的偏移量
- 2) esi 赋值为 data.samid[ebx] 的首地址，即整组数据的首地址。
- 3) 比较 arg 和 100 的大小：  
小于 100：edi 赋值为 low.samid[ebx] 的首地址  
等于 100：edi 赋值为 mid.samid[ebx] 的首地址  
大于 100：edi 赋值为 high.samid[ebx] 的首地址

### 4) ecx 赋值为 samples 的内存大小

### 5) cld 加 rep 指令复制 ecx 大小的数据

### 3、计算函数

函数声明：cal proc near

- 1) 定义内部变量 x，赋值为 5。ebx 为偏移量。
- 2) eax 赋值为 data.sda[ebx]，计算  $(eax * x + sdb - sdc + 100) / 128$  的值
- 3) eax 赋给 data.sf[ebx]，调用 copy 赋值，传递实参 eax

### 4、宏定义比较字符串

宏定义：strcmp macro buf1, buf2

1) esi 赋值为 0，作为偏移量。dh 赋值为 0，作为标记，若字符串不相等则赋值为 1。参数 buf1, buf2 为需要比较的两个字符串

### 2) 进入循环

3) eax 赋值为 buf1[esi]，ebx 赋值为 buf2[esi]

4) 比较 eax 与 ebx，如果相同则 esi=esi+1，比较下一个字符；否则 dh 赋值为 1，退出循环

### 5、主函数

函数声明：main proc

1) edi 赋值为 0，用来计数。dh 作为用户名和密码任一是否正确的标志，dl 作为两个是否都正确的标志。

# 汇编语言程序设计实验报告

- 2) 进入循环, dl 赋值为 0
- 3) 提醒输入用户名和密码
- 4) 分别调用宏定义比较字符串如果两次 dh 均为 0, 则 dl 为 0, 跳出循环; 否则输出相应的报错信息, 则 dl 为 1, edi=edi+1, 重新输入
- 5) edi=3 时已三次输入错误, 输出报错, 直接退出程序。
- 6) 调用 cal 函数, 计算并拷贝所有数据
- 7) 提醒输入 R 或 Q
  - R: 从计算开始重来一遍
  - Q: 退出
  - 其他: 重新输入

## 1.3.2 流程图

### 1、打印函数

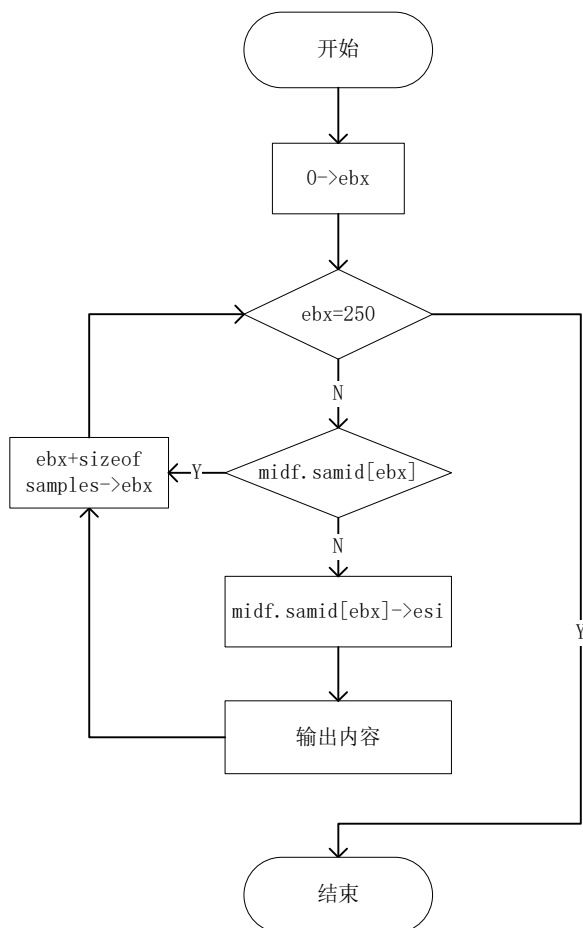


图 1.1 print 流程图

### 2、复制函数

# 汇编语言程序设计实验报告

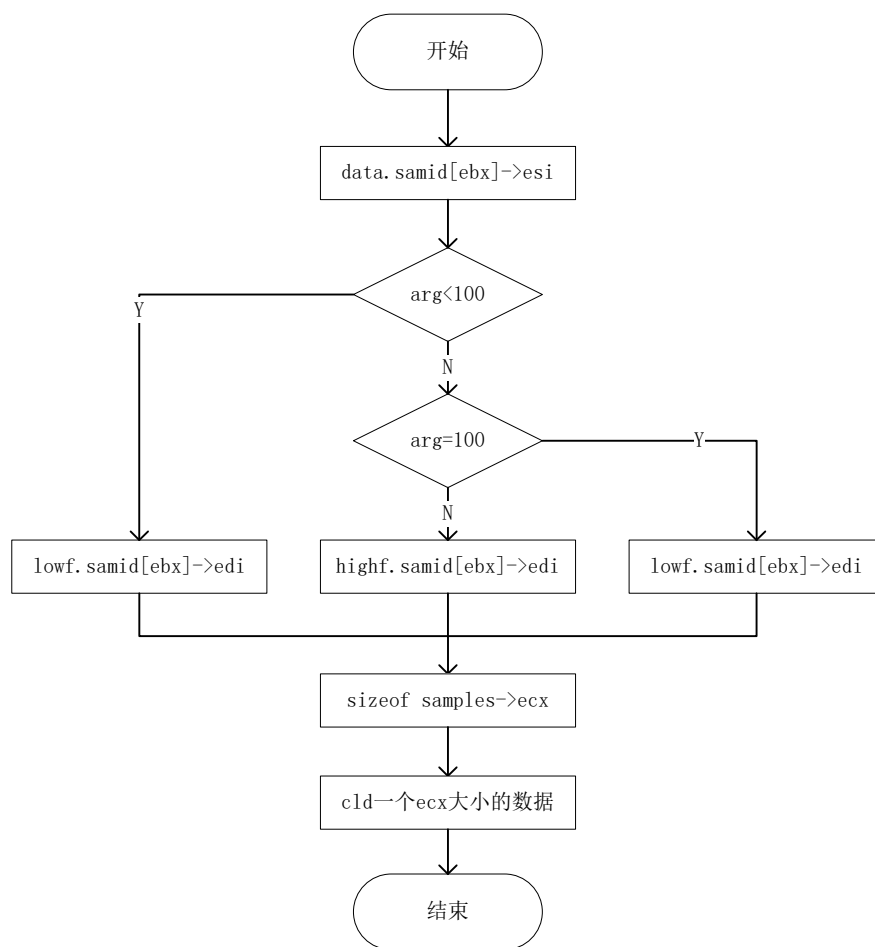


图 1.2 copy 流程图

## 3、计算函数

# 汇编语言程序设计实验报告

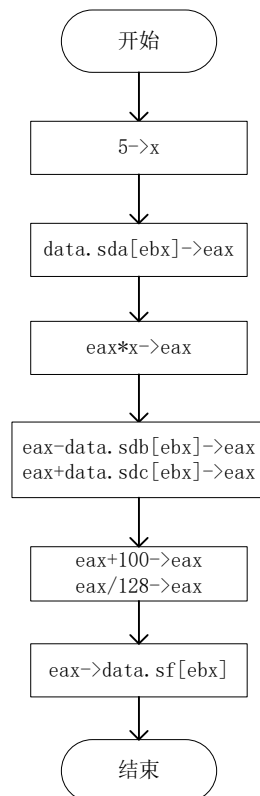


图 1.3 cal 流程图

## 4、宏定义比较字符串

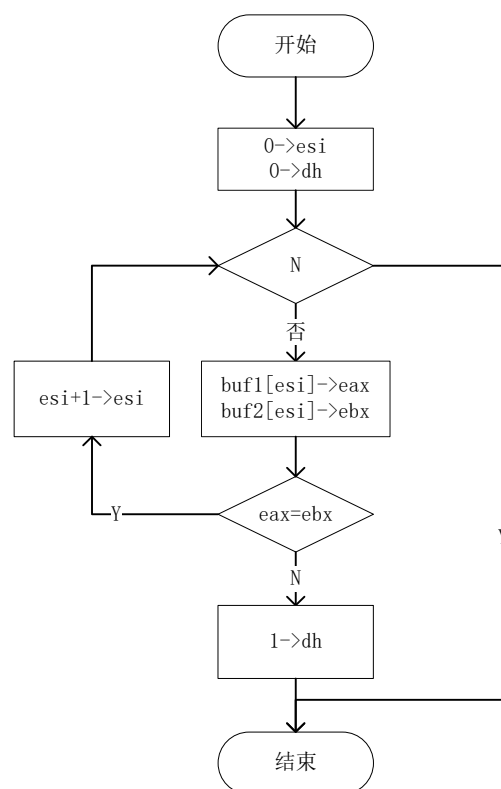


图 1.4 strcmp 流程图

# 汇编语言程序设计实验报告

## 5、主函数

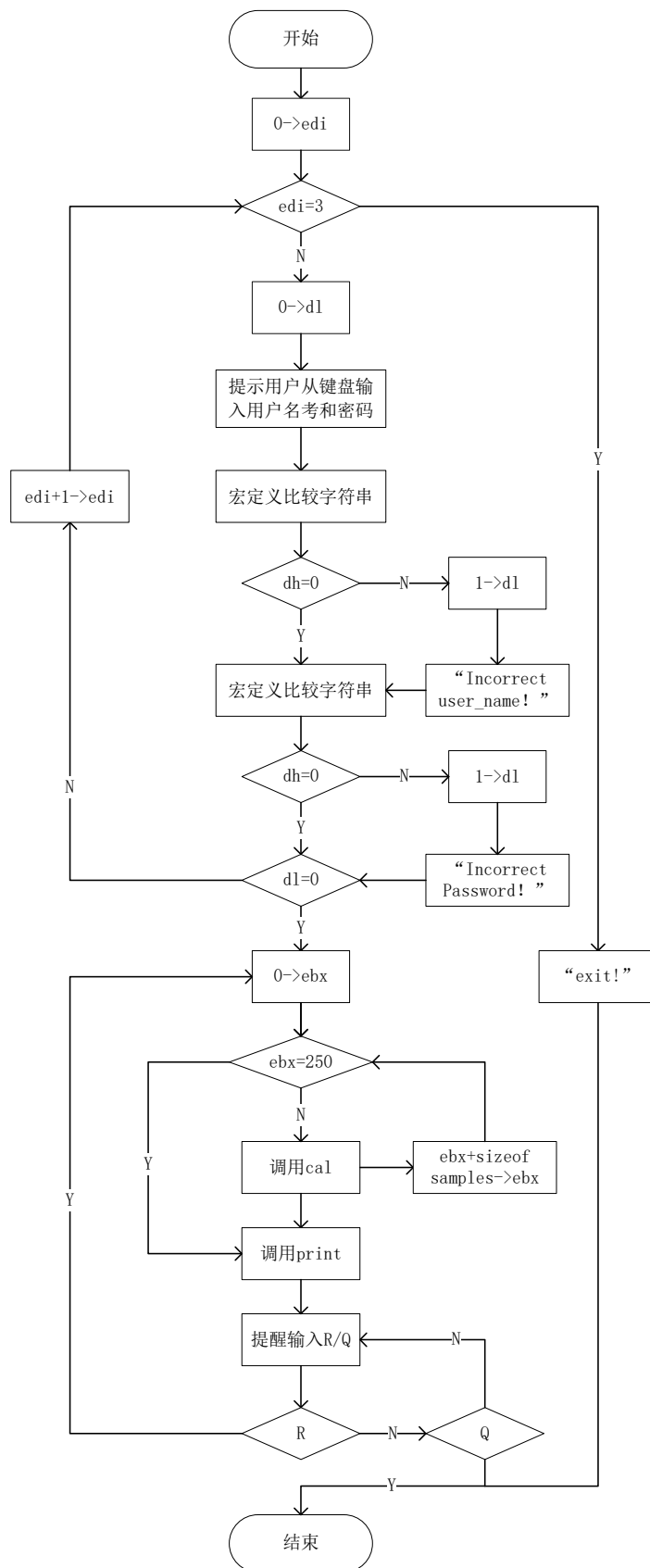


图 1.5 主函数流程图



# 汇编语言程序设计实验报告

## 1.3.3 源程序

```
main.asm
.686
.model flat, stdcall
ExitProcess PROTO STDCALL :DWORD
includelib kernel32.lib
includelib libcmnt.lib
includelib legacy_stdio_definitions.lib
printf PROTO C :VARARG
scanf PROTO C :VARARG
cal proto
print proto

.data
lpSmt db "%s",0
lpSSmt db "%s",0
lpFmt db "%s",0
user_name db 'huqinxin',0
pw db '12345678',0
str1 db 10 dup(0)
str2 db 10 dup(0)
str3 db 'wrong user name!',0ah,0dh,0
str4 db 'wrong password!',0ah,0dh,0
str5 db 'input user name:',0
str6 db 'input password:',0
str7 db 'exit!',0
str8 db 'input order:',0
str9 db 10 dup(0)

.STACK 200
.CODE

strcmp macro buf1, buf2
    local p
    local l
    local exit
    mov esi, 0
    mov dh, 0
p: cmp esi, 9
    je exit
    movsx eax, byte ptr buf1[esi]
    movsx ebx, byte ptr buf2[esi]
    cmp eax, ebx
    jne l
    inc esi
    jmp p
l: mov dh, 1
exit:
    endm

main proc c
    mov edi, 0
l1: cmp edi, 3
    je e1
    invoke printf, offset lpFmt, offset str5
    invoke scanf, offset lpSmt, offset str1
    invoke printf, offset lpFmt, offset str6
    invoke scanf, offset lpSmt, offset str2
    mov dl, 0
    strcmp str1, user_name
    cmp dh, 0
    jne l2
```

# 汇编语言程序设计实验报告

```
l4:strcmp str2, pw
    cmp dh, 0
    jne l3
l5:cmp dl, 0
    je e2
    inc edi
    jmp l1
l2:invoke printf, offset lpFmt, offset str3
    mov dl, 1
    jmp l4
l3:invoke printf, offset lpFmt, offset str4
    mov dl, 1
    jmp l5
e1:invoke printf, offset lpFmt, offset str7
    jmp e
e2:mov ebx, 0
lp:call cal
    add ebx, 25
    cmp ebx, 250
    jne lp
    call print
e3:invoke printf, offset lpFmt, offset str8
    invoke scanf, offset lpSSmt, offset str9
    cmp byte ptr str9[0], 82
    je e2
    cmp byte ptr str9[0], 81
    je e
    jmp e3
e:invoke ExitProcess, 0
main endp
end
```

func.asm

.686

.model flat, stdcall

printf           PROTO C :VARARG

public cal

public print

.data

lpFmt db "%s", 20h, 0

lpFFmt db "%d", 20h, "%d", 20h, "%d", 20h, "%d", 0ah, 0dh, 0

samples struct

samid db 9 dup(0)

sda dd ?

sdb dd ?

sdc dd ?

sf dd ?

samples ends

data samples <"s1", 2568099999, -1023, 1256, ?>,

<"s3", 425, 784, 0, ?>,

<"s2", 2540, 0, 0, ?>,

<"s4", 2540, -74, 987, ?>,

<"s5", 2540, 80765, -347, ?>,

<"s6", 2540, 6, -32224, ?>,

<"s7", 1249, 87, 2358, ?>,

<"s8", 2759, 0, 0, ?>,

<"s9", 0, 12700, 0, ?>,

<"s10", 23, -453, 27, ?>

lowf samples 10 dup(<>)

midf samples 10 dup(<>)

highf samples 10 dup(<>)

.STACK 200

# 汇编语言程序设计实验报告

```
.CODE

print proc near
    mov ebx, 0
lp: cmp midf.samid[ebx], 0
    je e
    lea esi, midf.samid[ebx]
    invoke printf, offset lpFmt, esi
    invoke printf, offset lpFFmt, midf.sda[ebx], midf.sdb[ebx], midf.sdc[ebx], midf.sf[ebx]
e: add ebx, sizeof samples
    cmp ebx, 250
    jne lp
    ret
print endp

copy proc stdcall arg: dword
    lea esi, data.samid[ebx]
    cmp arg, 64h
    jl l
    je m
    lea edi, highf.samid[ebx]
    jmp cpy
l: lea edi, lowf.samid[ebx]
    jmp cpy
m: lea edi, midf.samid[ebx]
cpy: mov ecx, sizeof samples
    cld
    rep movsb
    ret
copy endp

cal proc near
    local x: dword
    mov x, 05h
    mov eax, data.sda[ebx]
    imul eax, x
    add eax, data.sdb[ebx]
    sub eax, data.sdc[ebx]
    add eax, 64h
    shr eax, 7
    mov data.sf[ebx], eax
    invoke copy, eax
    ret
cal endp
end
```

## 1.3.4 实验记录与分析

### 1. 实验环境条件

INTEL 处理器 2GHz, 16G 内存; WINDOWS11 下 VS2022 社区版。

### 2. 汇编、链接中的情况

汇编过程中出现的主要的一个问题是：用函数形参作为偏移量是不可取的。我尝试在复制函数 `copy` 中传递一个 `dword` 类型的参数，在复制时作为当前这组数据相对于基址偏移量，实现取地址的指令，可以通过编译，但运行时会发生内存冲突。偏移量可以用参数来储存和比较，但当取地址时只能用寄存器。

### 3. 程序基本功能的验证情况

用户名为 huqinxin, 密码为 12345678

# 汇编语言程序设计实验报告

```
input user name:Huqinxin
input password:12345678
wrong user name!
input user name:huqinxin
input password:1234567
wrong password!
input user name:huqinxin
input password:123456
wrong password!
exit!
C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\3.1.exe (进程 27560)已退出, 代码为 0。
```

图 1.6 用户名密码三次错误

```
input user name:huqinxin
input password:12345678
s2 2540 0 0 100
s9 0 12700 0 100
input order:R
s2 2540 0 0 100
s9 0 12700 0 100
input order:Q
C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\3.1.exe (进程 6044)已退出, 代码为 0。
```

图 1.7 用户名密码正确

## 4.使用调试工具观察、探究代码的情况

1) 单步执行调用宏定义时，可以发现宏定义的反汇编代码直接跟在调用语句之后

mov dl, 0		
004E8596 B2 00	mov	dl, 0
strcmp str1, user_name		
004E8598 BE 00 00 00 00	mov	esi, 0 已用时间 <= 1ms
004E859D B6 00	mov	dh, 0
004E859F 83 FE 09	cmp	esi, 9
004E85A2 74 17	je	11+76h (04E85BBh)
004E85A4 0F BE 86 EB D4 55 00	movsx	eax, byte ptr str1 (055D4EBh)[esi]
004E85AB 0F BE 9E D9 D4 55 00	movsx	ebx, byte ptr user_name (055D4D9h)[esi]
004E85B2 3B C3	cmp	eax, ebx
004E85B4 75 03	jne	11+74h (04E85B9h)
004E85B6 46	inc	esi
004E85B7 EB E6	jmp	11+5Ah (04E859Fh)
004E85B9 B6 01	mov	dh, 1
cmp dh, 0		
004E85BB 80 FE 00	cmp	dh, 0
jne 12		
004E85BE 75 33	jne	11+0AEh (04E85F3h)

图 1.8 宏定义观察

2) 单步执行 main 函数调用无参外部函数 print 时，可以发现调用指令运行后先跳转到一个名为 \_print@0 的储存区，再跳转到该函数的储存区，调用堆栈段在两次跳转时都有改变，分别加入当时跳转到储存区的地址。

# 汇编语言程序设计实验报告

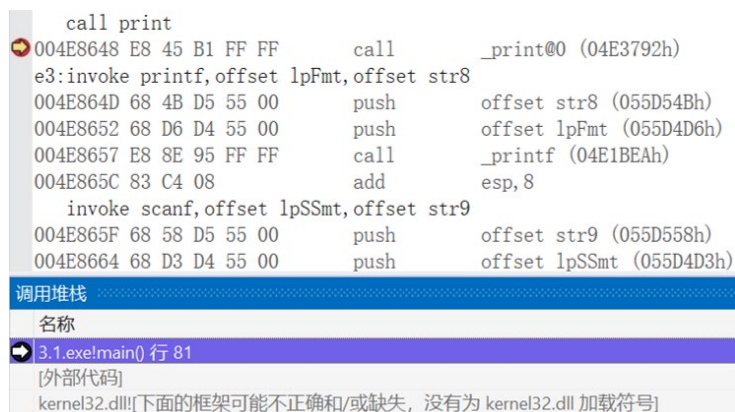


图 1.9 外部程序调用观察 1

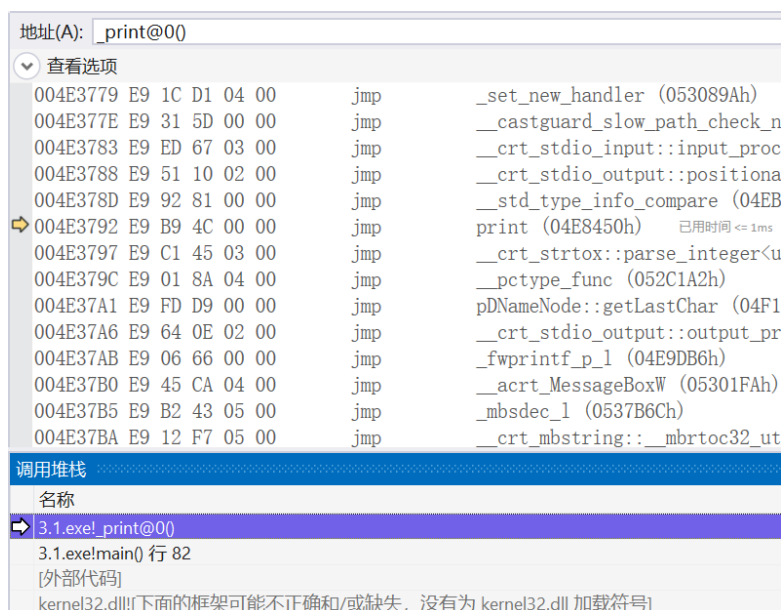


图 1.10 外部程序调用观察 2

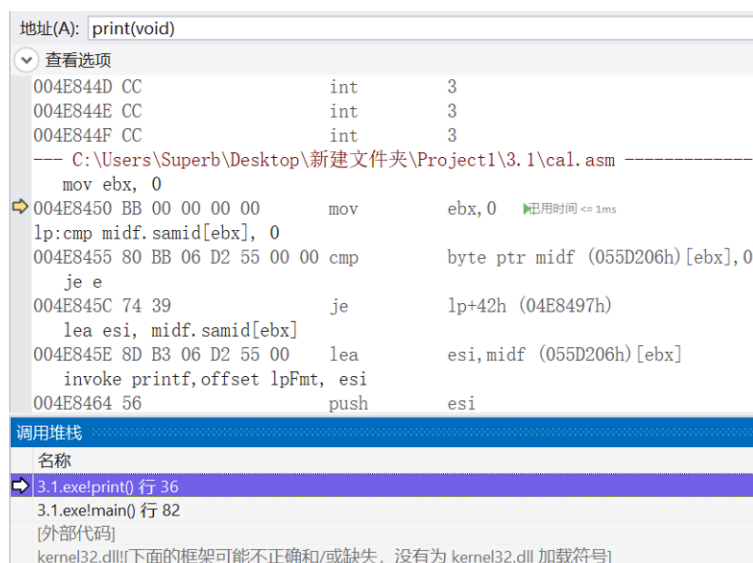


图 1.11 外部程序调用观察 3

3) 单步执行 cal 函数调用带参内部函数 copy 时, 观察到调用时只跳转了一次, 调用堆栈会将

# 汇编语言程序设计实验报告

所有线程上的函数按顺序推入调用堆栈中。调用的同时将 `eax` 推入栈中，可以在栈中观察到。

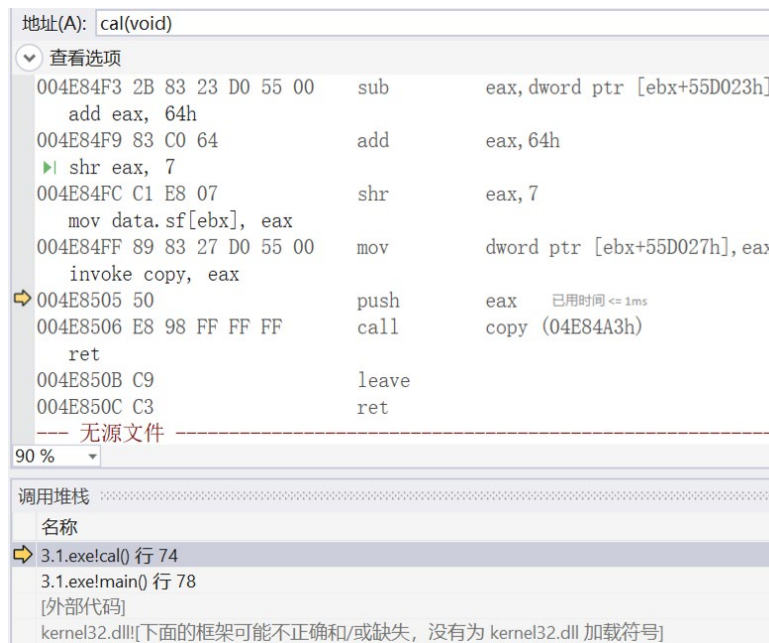


图 1.12 内部程序调用观察调用堆栈

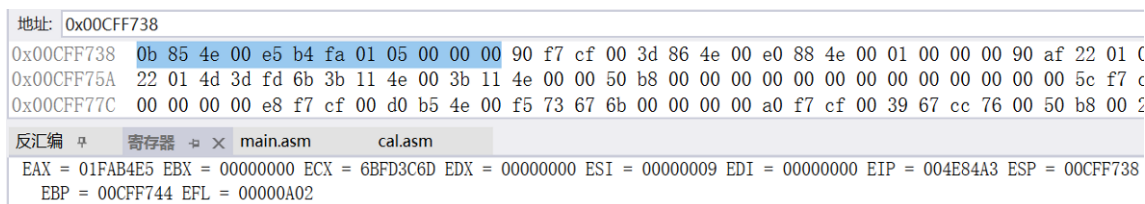


图 1.13 内部程序调用观察栈内容

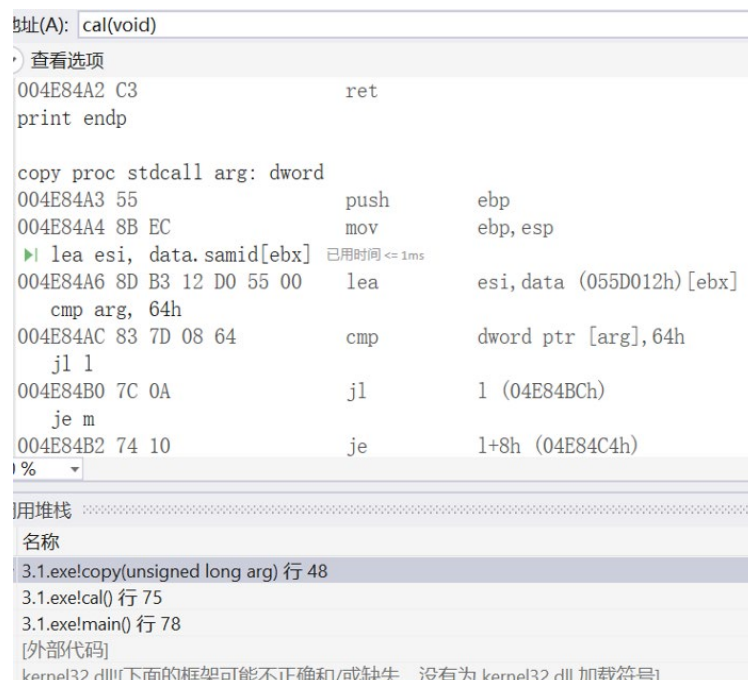


图 1.14 内部程序调用观察 3

4) 通过监视窗口观察 `cal` 函数和 `copy` 函数中执行时寄存器、内部变量和形参的值

# 汇编语言程序设计实验报告

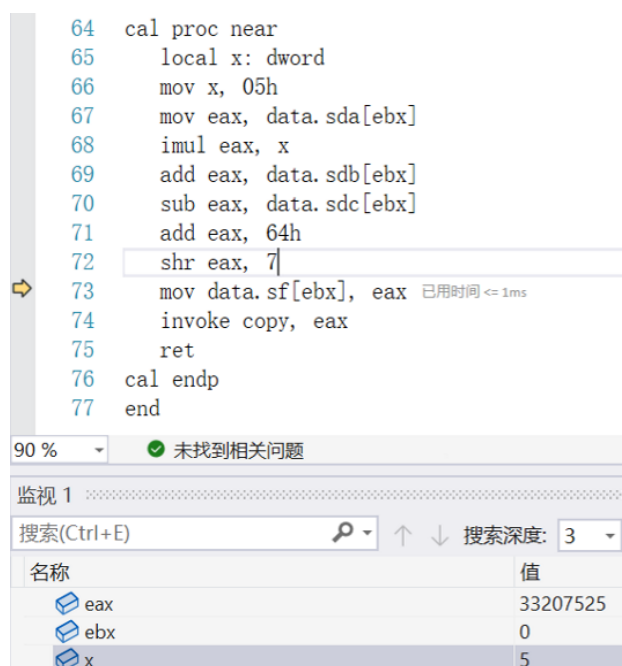


图 1.15 监视窗口和内存窗口观察 1

5) lea 赋给寄存器的偏移量可以在内存窗口中查找到相应的内存

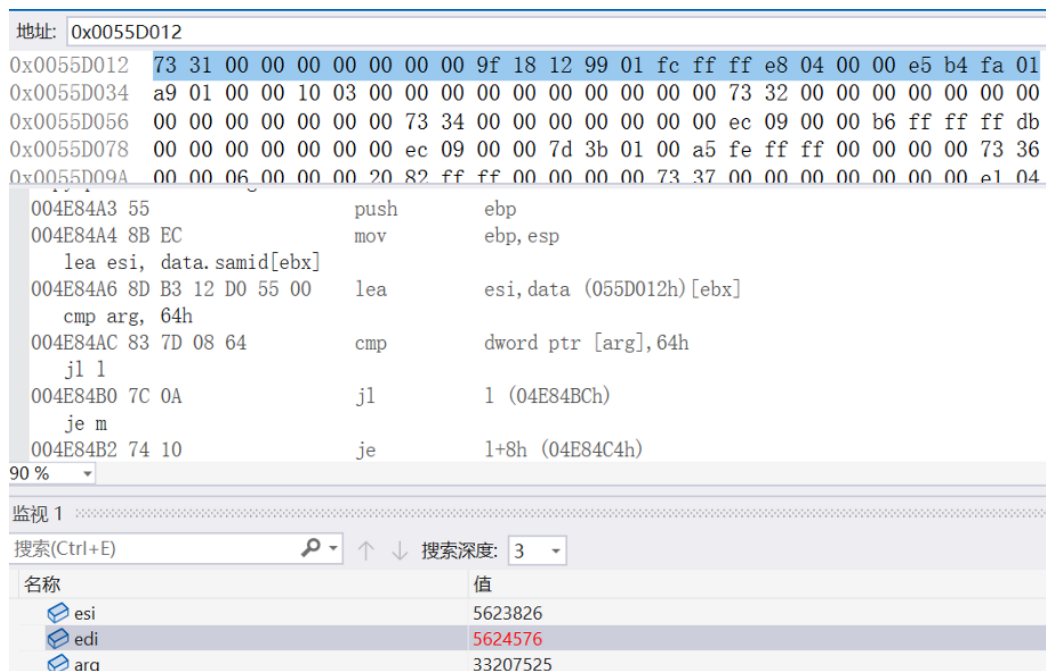


图 1.16 内部程序调用观察 2

## 1.4 内容 1.2 的实验过程

### 1.4.1 实验方法说明

1. 指令优化对程序的影响

- 1)将除以 128 的 sub 指令变为右移 7 位的 shr 指令
- 2)将 mov 指令实现的将内存位置赋给寄存器的功能用 lea 指令来实现。

# 汇编语言程序设计实验报告

3)将按字节循环将数据复制到储存区的功能用 `cld rep movsb` 来实现

## 2. 约束条件、算法与程序结构的影响

1)改成多模块：在算法优化的基础上，1)将字符串比较功能用宏定义来实现。2)将计算结果、复制数据和到储存区分别用两个函数来实现，由主函数调用计算函数，再由计算函数调用复制数据函数，在由主函数调用来实现功能。

2)C 语言和汇编语言混合编程，计算、复制和打印功能用汇编语言来实现，其他用 c 语言实现。

## 3. 编程环境的影响

改为 VS2019 下的 x64 程序。将解决方案平台切换至 X64，在项目“属性-链接-高级”里指定 start 为入口点。

## 1.4.2 实验记录与分析

### 1. 优化实验的效果记录与分析

```
3313
C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\Project1.4.exe (进程 15692)已退出，代码为 0。
```

图 1.17 算法优化前的运行时间

```
1578
C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\2.exe (进程 15548)已退出，代码为 0。
```

图 1.18 算法优化后的运行时间

算法优化后运行时间变短的原因：

- 1) 位移操作比除法的运行时间更短
- 2) `cld rep` 指令比循环指令更简短，寄存器和数据的传递更少，运行时间更短

## 2. 不同约束条件、算法与程序结构带来的差异

### 一、改为多模块

```
1875
C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\timer.exe (进程 7868)已退出，代码为 0。
```

图 1.19 改为多模块后更改后的运行时间

程序结构修改后运行时间变长原因：

- 1) 有堆栈的保存和恢复
- 2) 带有跳转的机器指令
- 3) 运行环境的保存与恢复

### 二、改为混合编程

1) C 语言定义的 `samples` 的内存大小发生变化，由 25 个字节变为 28 个字节。原因是 C 语言定义的 `sample` 内存后有三个空字节。

```
00 73 31 00 00 00 00 00 00 00 9f 18 12 99 01 fc ff ff e8 04 00 00 e5 b4 fa 01 73
```

图 1.20 汇编内存

```
00 73 31 00 00 00 00 00 00 00 00 00 a3 2f 27 00 01 fc ff ff e8 04 00 00 cb 87 01 00 73
```

图 1.21 C 语言内存

### 2) 代码变长

3) 在 c 语言中定义的变量赋值时，不是直接赋值，而是先赋给 `eax`，再赋给变量。这导致运



# 汇编语言程序设计实验报告

行时间变长。

## 4. 几种编程环境中程序的特点记录与分析

在 VS2019 下调试以下 x64 程序，观察与 32 位程序的不同之处。

### 1) 观察寄存器窗口

寄存器，标志位的符号名称发生改变

```
RAX = 00007FF6A3B71005 RBX = 0000000000000000 RCX = 000000BA07AF8000 RDX = 00007FF6A3B71005 RSI = 0000000000000000  
RDI = 0000000000000000 R8 = 000000BA07AF8000 R9 = 00007FF6A3B71005 R10 = 00007FFD6BDF7C90 R11 = 0000000000000000  
R12 = 0000000000000000 R13 = 0000000000000000 R14 = 0000000000000000 R15 = 0000000000000000 RIP = 00007FF6A3B71020  
RSP = 000000BA07CFFCD8 RBP = 0000000000000000 EFL = 00000244
```

```
CS = 0033 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B
```

```
OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0
```

图 1.22 x64 程序寄存器窗口

### 2) 基本指令、调用函数时的寄存器和堆栈操作都没有发生变化

### 3) 运行结果截图



图 1.23 x64 程序运行截图

## 1.5 小结

1. 获得的知识点：知道了如何在内存窗口中查找数据，在反汇编窗口中进行逐语句调试和观察。在实验的预习中复习了反码和补码，以及标志位。了解并学会运用许多汇编语句，如自加 `inc`，自减 `dec`，取地址 `lea`，跳转 `je`，`jne`，`jmp`，复制 `cld` 指令，`offset` 运算符返回数据标号的偏移量。学会了外部和内部变量的定义和使用，循环语句、比较语句的运用宏指令的编写，函数的调用、传递参数和返回值，多模块的使用。还学会了调用 `GetTickCount` 函数来获取程序运行的时间，进行程序优化前后的比较。

2. 在程序编写和调试过程中没有经验，比较仓促，很多基础还需要巩固。最大的感受是在第一次实验中感觉第一次开始深入了解汇编语言，了解计算机的原理，对高级语言如何编译也有了一点了解。综合之前学的一些课程，计算机内部的模样开始在大脑中有了清晰的轮廓和构想。

## 二、利用汇编语言特点的实验

### 2.1 目的与要求

掌握编写、调试汇编语言程序的基本方法与技术，能根据实验任务要求,设计出较充分利用了汇编语言优势的软件功能部件或软件系统。

### 2.2 实验内容

在编写的程序中，通过加入内存操控，反跟踪，中断处理，指令优化，程序结构调整等实践内容，达到特殊的效果。

### 2.3 实验过程

#### 2.3.1 实验方法说明

##### 1.中断处理程序的设计思想与实验方法

设计思想：在运行界面右上角显示时钟，程序功能退出后时钟不消失。

实验方法：将驻留的相关代码片段添加在程序功能结束之后、程序退出之前，并在程序开头设置 START 节点。再次编译执行，会发现程序结束之后，时钟仍不会消失。

##### 2.反跟踪程序的设计思想与实验方法

1) 将数据段的密码进行加密，对各个字节进行加减乘除、与、或、异或、位移等运算，最后寻址使用数据时再解密。

2) 动态修改代码，添加干扰代码片段，进行多次干扰跳转，使程序可读性下降。

3) 加入计时模块来抵制动态调试跟踪，如果跳转时耗时过长直接退出。

4) 将跳转的地址储存在数据段中，跳转时先寻址再解密再跳转。

##### 3.指令优化及程序结构的实验方法

指令优化：

1) 将除以 128 的 sub 指令变为右移 7 位的 shr 指令

2) 将 mov 指令实现的将内存位置赋给寄存器的功能用 lea 指令来实现。

3) 将按字节循环将数据复制到储存区的功能用 cld rep movsb 来实现

程序结构更改：

1) 改成多模块：在算法优化的基础上将字符串比较功能用宏定义来实现，将计算结果、复制数据和到储存区分别用两个函数来实现，由主函数调用计算函数，再由计算函数调用复制数据函数，在由主函数调用来实现功能。

2) C 语言和汇编混合编程，计算、复制和打印功能用汇编语言来实现，其他用 c 语言实现。

#### 2.3.2 实验记录与分析

##### 1.中断处理程序的特别之处

# 汇编语言程序设计实验报告

## 1)将时钟停留在界面上的程序段

```
MOV DX, OFFSET START +15 ; 计算中断处理程序占用的字节数, +15 是为了在计算节数时能向上取整
MOV CL, 4
SHR DX, CL ; 把字节数换算成节数(每节代表 16 个字节)
ADD DX, 10H ; 驻留的长度还需包括程序段前缀的内容(100H 个 字节)
MOV AL, 0 ; 退出码为 0
MOV AH, 31H ; 退出时, 将(DX)节的主存单元驻留(不释放)
INT 21H
```

图 2.2 中断源程序 1

## 2)设置 start 断点

```
.386
STACK SEGMENT USE16 STACK ; 主程序的堆栈段
DB 200 DUP(0)
STACK ENDS
;
CODE SEGMENT USE16
ASSUME CS: CODE, DS: CODE, SS: STACK ; 新的 INT 08H 使用的变量
START:
COUNT DB 18 ; “滴答”计数
```

图 2.2 中断源程序 2

## 3)当调试到 00bb 时, 指令为 int 16h, 调试中断, 回到运行界面。然后输入 q, 重新回到调试

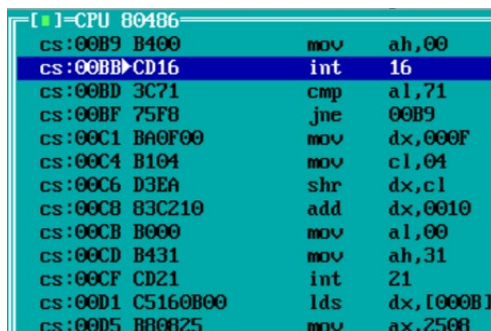


图 2.3 中断调试 1

## 4)运行到 00cf 时, 指令为 int 21h, 程序退出。

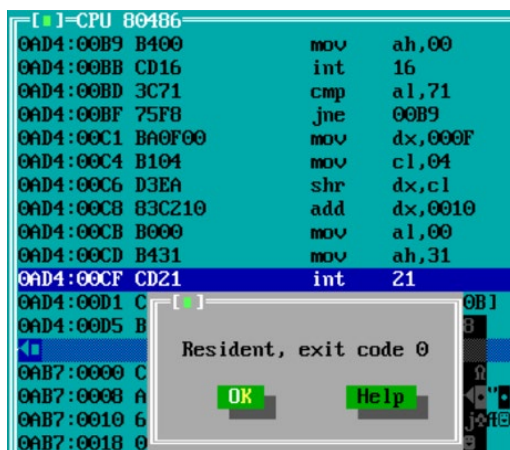


图 2.4 中断调试 2

## 2.反跟踪效果的验证

### 1)数据加密

# 汇编语言程序设计实验报告

```
pw db '1' xor 'a'
db '2' xor 's'
db '3' xor 'm'
db 34h
db 35h
db 36h
db 37h
db 38h
db 0
```

图 2.5 反跟踪-数据加密

## 2)添加干扰代码和跳转

```
mov x, 05h
jmp l1
12:add eax, data.sdb[ebx]
jmp l3
11:mov eax, data.sda[ebx]
mov edx, 0
imul eax, x
inc esi
mov buf, 12
xor edi, edx
jmp buf
13:lea esi, data
sub eax, data.sdc[ebx]
cmp esi, edi
je l4
15:add eax, 64h
dec ecx
sar eax, 7
14:mov data.sf[ebx], eax
```

图 2.6 反跟踪-干扰

## 3)加入计时模块来抵制动态调试跟踪，如果跳转时耗时过长直接退出。

```
invoke GetTickCount
mov startTime, eax

invoke GetTickCount
sub eax, startTime
mov ecx, eax

lp:call cal
cmp ecx, 0
jne t

t :invoke printf, offset lpFmt, offset str0
e :invoke ExitProcess, 0
main endp
end
```

图 2.7 反跟踪-计时

实现效果：如果在 cal 函数中逐语句调试或设置断点调试，会直接结束程序

```
input user name:huqinxin
input password:12345678
wrong time!
```

C:\Users\Superb\Desktop\新建文件夹\Project1\Debug\4.2.exe (进程 22420)已退出，代码为 0。

图 2.8 反跟踪-计时 2

## 4) 将跳转的地址储存到数据段中，跳转时先寻址再解密再跳转。

# 汇编语言程序设计实验报告

```
buf dd ?  
mov buf, 12  
inc esi  
xor edi,edx  
jmp buf
```

图 2.9 反跟踪-跳转

## 3.跟踪与破解程序

三人小组: U202015360, U202015372, U202015381。破解的是 U202015381 的反汇编程序。

1) 首先运行 exe 程序查看输出的提示信息, 这段字符串在数据段中

Please input the user's name:

图 2.10 破解 1

2) 查看 hex 数据段显示, 可以找到代码段信息, 得到加密后的密码和储存地址

```
:0047DBF0 00 00 00 00 00 00 00 45 .....E  
:0047DBF8 C0 6F 8A 3F 39 30 B7 93 .o.?90..
```

图 2.11 破解 2

3) 打开串式参考, 找到反汇编代码的位置

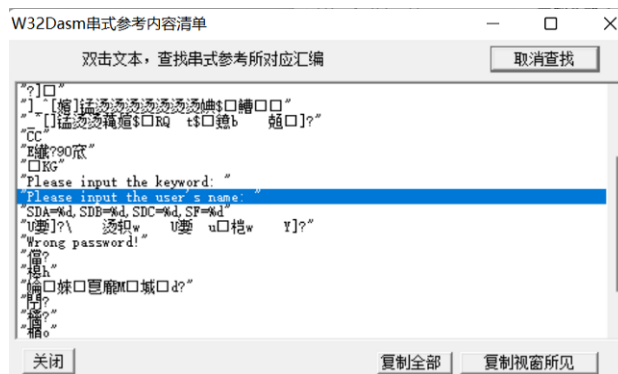


图 2.12 破解 3

4) 得到密码只有 3 位, 数据段密码的前三位有用

```
:0043DCFE 00000000      BYTE 4 DUP(0)  
  
:0043DD02 8A1503DC4700    mov dl, byte ptr [0047DC03]  
:0043DD08 50              push eax  
:0043DD09 A32CDB4700    mov dword ptr [0047DB2C], eax  
:0043DD0E BE52000000    mov esi, 00000052  
:0043DD13 53              push ebx  
:0043DD14 52              push edx  
:0043DD15 83E801          sub eax, 00000001  
:0043DD18 2BD0            sub edx, eax  
:0043DD1A 83FA00          cmp edx, 00000000  
:0043DD1D 74BD            je 0043DCDC  
:0043DD1F 8AC8            mov cl, al
```

图 2.13 破解密码

5) 再继续往下读, 得到密码的加密为 $(x-0x24)*3$ 。因为加密后为 45 c0 6f, 所以密码为 d1R

# 汇编语言程序设计实验报告

```
:0043DD23 8B1D28DB4700    mov ebx, dword ptr [0047DB28]
:0043DD29 8A9100DC4700    mov dl, byte ptr [ecx+0047DC00]
:0043DD2F 3B03            cmp eax, dword ptr [ebx]
:0043DD31 74A9            je 0043DCDC
:0043DD33 8BC2            mov eax, edx
:0043DD35 83E824          sub eax, 00000024
:0043DD38 83C203          add edx, 00000003
:0043DD3B 0FAFD0          imul edx, eax
:0043DD3E BF03000000    mov edi, 00000003
:0043DD43 F7EF            imul edi
:0043DD45 8A91F7DB4700    mov dl, byte ptr [ecx+0047DBF7]
:0043DD4B 3BC2            cmp eax, edx
:0043DD4D 758D            jne 0043DCDC
:0043DD4F 6683F901        cmp cx, 0001
:0043DD53 75CC            jne 0043DD21
:0043DD55 740C            je 0043DD63
```

图 2.14 破解密码

6) 查找密码比较完后跳转的地址 0043DD63, 可以得到计算方法为 $(5*sda+sdb)/2-sdc$

```
:0043DD63 5A            pop edx
:0043DD64 833500DB470064  xor dword ptr [0047DB00], 00000064
:0043DD6B A100DB4700    mov eax, dword ptr [0047DB00]
:0043DD70 BA00000000    mov edx, 00000000
:0043DD75 BB05000000    mov ebx, 00000005
:0043DD7A F7EB            imul ebx
:0043DD7C 833504DB470049  xor dword ptr [0047DB04], 00000049
:0043DD83 58            pop eax
:0043DD84 030504DB4700    add eax, dword ptr [0047DB04]
:0043DD8A D1F8            sar eax, 1
:0043DD8C 833508DB470052  xor dword ptr [0047DB08], 00000052
:0043DD93 2B0508DB4700    sub eax, dword ptr [0047DB08]
:0043DD99 83F864          cmp eax, 00000064
```

图 2.15 破解计算方法

## 4. 特定指令及程序结构的效果

特定指令:

1) shr eax, 7: 将 eax 右移 7 位来代替  $eax/128$

```
add eax, 64h
shr eax, 7
mov data.sf[ebx], eax
```

图 2.16 特定指令 1

2. lea esi, data.samid[ebx]: 用 lea 指令语句来代替以下语句

```
mov data.sf[ebx], eax
lea esi, data.samid[ebx]
cmp eax, 64h
jl l
je m
lea edi, highf.samid[ebx]
jmp copy
l: lea edi, lowf.samid[ebx]
jmp copy
m: lea edi, midf.samid[ebx]
jmp copy
```

图 2.17 特定指令 2

3. 用 cld rep 语句来代替循环语句

```
mov esi, offset data
add ecx, sizeof samples
imul ecx, ebx
mov esi, ecx
```

# 汇编语言程序设计实验报告

```
copy: mov ecx, sizeof samples
      add ebx, ecx
      cld
      rep movsb
      jmp lp
```

图 2.18 特定指令 3

## 二、程序结构:

### 1.多模块编程

宏定义及函数调用关系如图

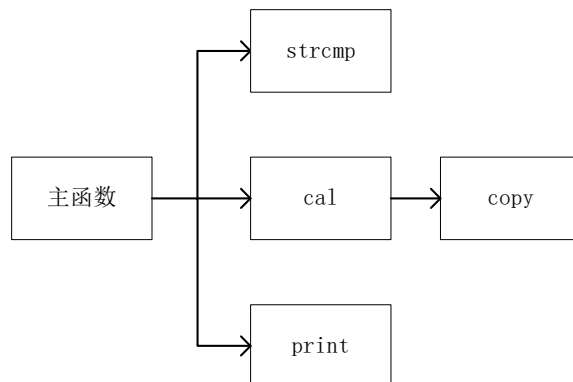


图 2.19 多模块

### 2.C 语言和汇编混合编程

在 C 语言中嵌入\_\_asm 部分，用汇编语言实现功能

```
if (i == 3) return 0;
r:
__asm {
    mov ebx, 0
    lp:
    ► mov eax, data.sda[ebx]
    imul eax, 05h
    add eax, data.sdb[ebx]
    sub eax, data.sdc[ebx]
    add eax, 64h
    shr eax, 7
    mov data.sf[ebx], eax
    lea esi, data.samid[ebx]
    cmp eax, 64h
    jl l
    je m
    lea edi, highf.samid[ebx]
    jmp copy
    l:
    lea edi, lowf.samid[ebx]
    jmp copy
```

图 2.20 混合编程

## 2.4 小结

1. 在该实验中，我深入体会了指令对程序执行时间的影响，内外部函数及含参函数的声明、定义和调用，学会了编写宏定义，学会了使用模块，并对多个文件多个模块进行汇编。

## 汇 编 语 言 程 序 设 计 实 验 报 告

---

2.对于大规模程序进行调试时，可以通过对各个模块进行调试来减少工作量。

3..通过这次试验，我观察到 C 语言中会在很多地方都用到寄存器，对于变量的一些操作，很多情况下编译器会先将变量的值赋值给 `eax`，用 `eax` 处理完，再赋值给变量，所以在混合编程中，用汇编子程序调用 C 的函数时候，先观察会用到哪些寄存器，避免使用或使用后要复原。

4.这次实验我学会了如何实现混合编程，知道了不同的编程语言是可以协同解决一个问题的，不同语言的特点有很大的不同。混合编程有时可以更好地解决问题。

5.通过模仿学习了如何编写反跟踪程序，并如何破解别人的反跟踪程序。



# 汇编语言程序设计实验报告

## 三、工具环境的体验

### 3.1 目的与要求

熟悉支持汇编语言开发、调试以及软件反汇编的主流工具的功能、特点与局限性及使用方法。

### 3.2 实验过程

#### 3.2.1 WINDOWS10 下 VS2019 等工具包

任务 1.1 从 C 语言到汇编语言

对于下列给定的 C 语言程序，使用 VS2019 进行编译、链接和调试。通过实验，回答如下问题：

(1) 显示反汇编窗口，了解 C 语言与汇编语句的对应关系。在反汇编窗口中的“查看选项”下有“显示符号名”，指出勾选与不勾该项选时，反汇编窗口显示内容的差异；

差异为是否显示变量的名字

```
00801ACC 8D 7D E8      lea      edi,[ebp-18h]
00801ACF B9 06 00 00 00    mov      ecx,6
00801AD4 B8 CC CC CC CC    mov      eax,0CCCCCCCCh
00801AD9 F3 AB          rep stos dword ptr es:[edi]
00801ADB B9 08 C0 80 00    mov      ecx,offset _173D59EB_源@c (080C008h)
00801AE0 E8 36 F8 FF FF    call     @_CheckForDebuggerJustMyCode@4 (080131Bh)
  int i;
  int result = 0;
00801AE5 C7 45 EC 00 00 00 00 mov      dword ptr [result],0
  for (i = 0; i < length; i++)
00801AEC C7 45 F8 00 00 00 00 mov      dword ptr [i],0
00801AF3 EB 09          jmp      ___$EncStackInitStart+32h (0801AFEh)
```

图 3.1 勾选

```
00801ACC 8D 7D E8      lea      edi,[ebp-18h]
00801ACF B9 06 00 00 00    mov      ecx,6
00801AD4 B8 CC CC CC CC    mov      eax,0CCCCCCCCh
00801AD9 F3 AB          rep stos dword ptr es:[edi]
00801ADB B9 08 C0 80 00    mov      ecx,80C008h
00801AE0 E8 36 F8 FF FF    call     0080131B
  int i;
  int result = 0;
00801AE5 C7 45 EC 00 00 00 00 mov      dword ptr [ebp-14h],0
  for (i = 0; i < length; i++)
00801AEC C7 45 F8 00 00 00 00 mov      dword ptr [ebp-8],0
```

图 3.2 不勾选

(2) 显示寄存器窗口。在该窗口中设置显示 寄存器、段寄存器、标志寄存器等；

```
寄存器
EAX = 0000001B EBX = 0058E000 ECX = 25B7774F EDX = 56F05A84 ESI = 00681023 EDI = 0078F758 EIP = 00681914 ESP = 0078F668 EBP = 0078F758 EFL = 00000212
CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B |
OV = 0 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 1 PE = 0 CY = 0
```

图 3.3 寄存器窗口

(3) 显示监视窗口，观察变量的值；显示内存窗口，观察变量的值（整型值、字符串等）在内存中的具体表现细节。

# 汇编语言程序设计实验报告

监视 1		
搜索(Ctrl+E) <input type="text"/> 搜索深度: 3		
名称	值	类型
x	100	short
y	-32700	short
z	15	short
argc	1	int
argv	0x000002127925a510 {0x000002127925a520 "C:\\U...	char **
添加要监视的项		

图 3.4 监视窗口

地址: 0x00D0A014
0x00D0A014 64 00 00 00

图 3.5 整型 x 内存

地址: 0x00D07B30
0x00D07B30 54 68 65 20 65 6e 64 21

图 3.6 字符串"the end!"内存

(4) 有符号与无符号整型数是如何存储的:

int result=0 和 unsigned result=0 的指令相同

```
int result = 0;
00901AE5 C7 45 EC 00 00 00 00 mov     dword ptr [result],0
```

图 3.7 有符号数储存

```
unsigned result = 0;
00701AE5 C7 45 EC 00 00 00 00 mov     dword ptr [result],0
```

图 3.8 无符号数储存

(5) 有符号数和无符号数的加减运算有无差别，是如何执行的？执行加法运算指令时，标志寄存器是如何设置的？执行比较指令时又有什么差异？

有符号数和无符号数的加减运算无差别，减法指令都为 2B C1,加法指令都为 03 C1,但 mov 指令有差别，有符号为 movsx 指令，无符号为 movzx 指令。

```
z = x - y;
007A1911 0F BF 05 14 A0 7A 00 movsx     eax,word ptr [x (07AA014h)]
007A1918 0F BF 0D 18 A0 7A 00 movsx     ecx,word ptr [y (07AA018h)]
007A191F 2B C1                sub      eax,ecx
007A1921 66 89 45 F4                mov     word ptr [z],ax
```

图 3.9 有符号减法运算

```
z = x + y;
00991911 0F B7 05 14 A0 99 00 movzx     eax,word ptr [x (099A014h)]
00991918 0F B7 0D 18 A0 99 00 movzx     ecx,word ptr [y (099A018h)]
0099191F 03 C1                add      eax,ecx
00991921 66 89 45 F4                mov     word ptr [z],ax
```

图 3.10 无符号加法运算

寄存器:

OV = 0 UP = 0 EI = 1 PL = 1 ZR = 0 AC = 0 PE = 0 CY = 1

图 3.11 无符号减法

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE = 0 CY = 0

图 3.12 无符号加法

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE = 0 CY = 1

图 3.13 有符号减法

OV = 0 UP = 0 EI = 1 PL = 1 ZR = 0 AC = 0 PE = 0 CY = 0

图 3.14 有符号加法

# 汇编语言程序设计实验报告

比较指令的差异体现在：无符号比较跳转指令为 `jae`，有符号比较跳转指令为 `jge`

```
001A1B01 3B 45 0C      cmp     eax,dword ptr [length]
001A1B04 73 11      jae     ___$EncStackInitStart+4Bh (01A1B17h)
```

图 3.15 无符号数比较

```
007B1B01 3B 45 0C      cmp     eax,dword ptr [length]
007B1B04 7D 11      jge     ___$EncStackInitStart+4Bh (07B1B17h)
```

图 3.16 有符号数比较

(6) 观察思考：程序在编译时，在 `sum` 函数的 `for (i = 0; i < length; i++)` 处会给出警告信息：有符号/无符号不匹配。请问，`i < length` 最终采用的是有符号数比较还是无符号数比较？若将语句“`z = sum(a, 5);`”换成“`z = sum(a, 0x90000000);`”运行结果如何？为什么？如果将 `sum` 函数的参数 `unsigned length` 改为 `int length`，执行“`z = sum(a, 0x90000000);`”运行结果又如何？为什么？

1. 采用的是无符号数比较，指令为 `jae`

```
00F71B01 3B 45 0C      cmp     eax,dword ptr [length]
00F71B04 73 11      jae     ___$EncStackInitStart+4Bh (0F71B17h)
```

图 3.17

2. 引发了异常：读取访问权限冲突。

原因：`length` 的大小超出了数组 `a` 的长度。

3.

```
sum : 0
condition1: 100 > -32700
condition2: (100) - (-32700) = -32736
condition3: & 100 < & -32700
The end!
C:\Users\Superb\Desktop\新建文件夹\Project2\Debug\Project2.exe (进程 184)已退出，代码为 0。
```

图 3.15 运行截图

原因：`0x90000000` 为负数，没有进入 `sum` 函数中的 `for` 循环。

## 任务 1.2 观察汇编语言程序

对下列汇编语言源程序（其功能是：定义了一个数据段，并用指定的内存寻址方式，将 `buf1` 缓冲区中的 12 个字节内容拷贝到 `buf2` 中，并显示两个缓冲区中的字符串），完成下列要求：

使用 VS2019 进行编译、链接和调试（新建工程选项为控制台的 x86 WIN32 无源码。后续实验在没有另外指定时，均采用该工具的该选项）。完成反汇编窗口显示，了解汇编源程序中的语句与反汇编语句之间的关系。同时，完成同任务 1.1 的寄存器窗口、监视窗口、内存窗口的操作。观察数据段的存储结果，观察存储规律、各个变量的地址之间的关系；观察堆栈段内容。熟悉单步、断点等执行操作，尝试在调试状态下修改变量的值、指令的代码等。

分别将该程序不同部分的代码随意修改，观察编译器提示的错误信息的特点。


尝试将访问 `buf1` 的寻址方式由寄存器间接寻址方式改成其他的寻址方式。

```
寄存器
EAX = 009FFF74 EBX = 00F50000 ECX = 00000000 EDX = 7215246F ESI = 012CAF00 EDI = 012CDAE0 EIP = 0098B780 ESP = 00DFF768 EBP = 00DFF7AC EFL = 00000202

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B
|
OV = 0 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE = 0 CY = 0
```

图 3.16 寄存器窗口

# 汇编语言程序设计实验报告

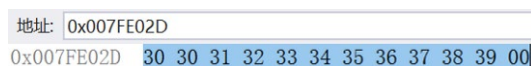


名称	值	类型
X	10 '\n'	unsigned char
Y	10	unsigned short
U	3	unsigned short
STR1	71 'G'	unsigned char
P	10477573	unsigned long
Q	5 'x5'	unsigned char
buf1+5	53	int

图 3.17 监视窗口

```
MOV ESI, OFFSET buf1
0078B780 BE 2D E0 7F 00      mov     esi, offset buf1 (07FE02Dh)
MOV EDI, OFFSET buf2
0078B785 BF 39 E0 7F 00      mov     edi, offset buf2 (07FE039h)
MOV ECX, 0
0078B78A B9 00 00 00 00      mov     ecx, 0
```

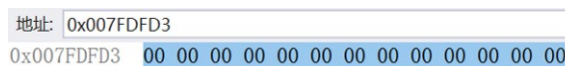
图 3.18 字符串和整型储存指令



地址: 0x007FE02D

0x007FE02D 30 30 31 32 33 34 35 36 37 38 39 00

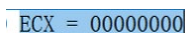
图 3.19 buf1 内存



地址: 0x007FDFD3

0x007FDFD3 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 3.20 buf2 内存



ECX = 00000000

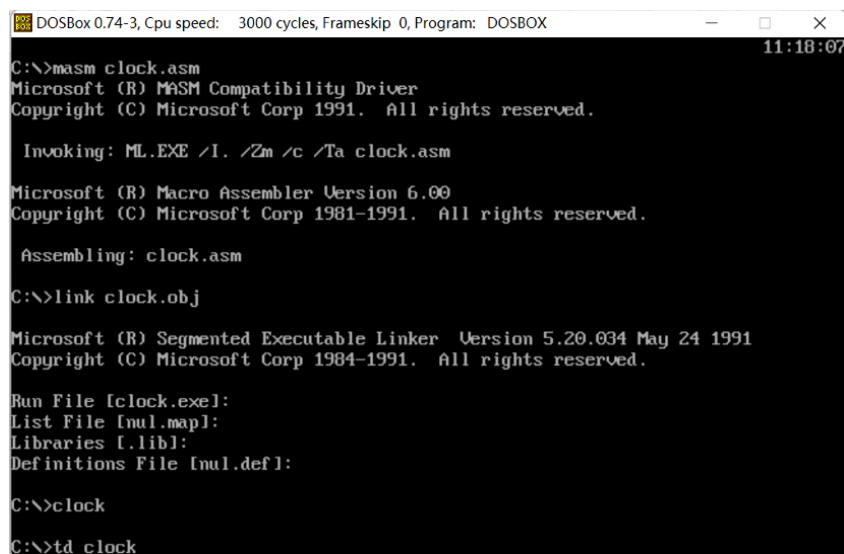
图 3.21 ecx 数值

栈寄存器: ESP = 00DFF768

改变 buf1 寻址方式: lea esi, buf1

## 3.2.2 DOSBOX 下的工具包

DOSBOX 下, 将源代码转化为可执行文件需要经过编译、链接的过程, 运行后的输出会直接显示在命令行界面中。



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
C:\>masm clock.asm
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1991. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta clock.asm

Microsoft (R) Macro Assembler Version 6.00
Copyright (C) Microsoft Corp 1981-1991. All rights reserved.

Assembling: clock.asm

C:\>link clock.obj

Microsoft (R) Segmented Executable Linker Version 5.20.034 May 24 1991
Copyright (C) Microsoft Corp 1984-1991. All rights reserved.

Run File [clock.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:

C:\>clock
C:\>td clock
```

图 3.22 时钟程序运行结果

调试功能:

# 汇编语言程序设计实验报告

1)界面：左上为代码段，左下为数据段，右上为寄存器，右下为堆栈段

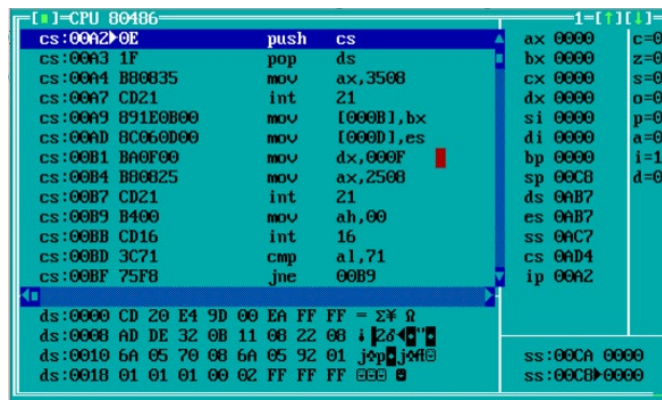


图 3.23 调试界面

2)在调试过程中，可以通过 breakpoints 设置断点，通过 run 的 step over 或 fn+F8 进行逐语句调试，变化的内存数据会由黑色变为白色

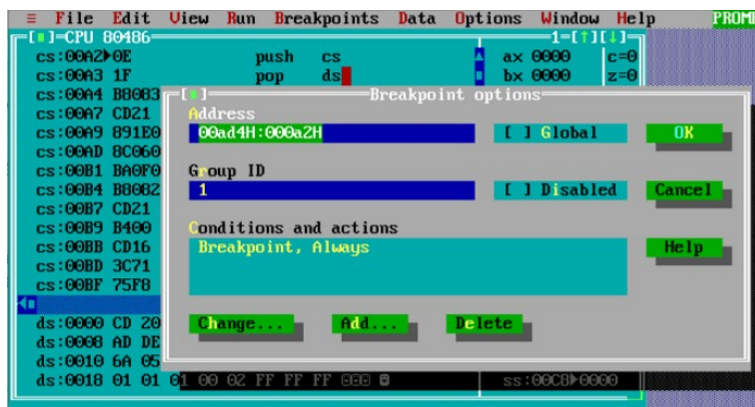


图 3.23 断点设置

## 3.2.3 QEMU 下 ARMv8 的工具包

1.hello world 汇编语言程序的显示、编译和运行

```
[root@localhost ~]# as hello.s -o hello.o
[root@localhost ~]# ld hello.o -o hello
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
[root@localhost ~]# ./hello
Hello World!
```

图 3.24 helloworld 的编译和运行

2.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的编译和运行

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memorycopy time is 510486304 ns
```

图 3.25 混合编程程序的编译和运行

3.对上一程序的优化程序

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
memorycopy time is 127053200 ns
```

图 3.26 优化程序的编译和运行

4.用 gdb 进行反汇编调试

64 位寄存器为 x0~x30，对应低 32 位 w0~w30



# 汇编语言程序设计实验报告

基本指令：

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将寄存器的高 24 位清零。

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中，同时将寄存器的高 16 位清零。

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。

LDP/STP 指令是 LDR/STR 的衍生，可以同时读/写两个寄存器，并访问 16 个字节的内存数据。

指令示例：

```
LDR R0, [R1] ; 将存储器地址为 R1 的字数据读入寄存器 R0。
LDR R0, [R1, R2] ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0。
LDR R0, [R1, #8] ; 将存储器地址为 R1+8 的字数据读入寄存器 R0。
LDR R0, [R1, R2] ! ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0，并
将新地址 R1 + R2 写入 R1。
LDR R0, [R1, #8] ! ; 将存储器地址为 R1+8 的字数据读入寄存器 R0，并
将新地址 R1 + 8 写入 R1。
LDR R0, [R1], R2 ; 将存储器地址为 R1 的字数据读入寄存器 R0，并将新地
址 R1 + R2 写入 R1。
LDR R0, [R1, R2, LSL # 2] ! ; 将存储器地址为 R1 + R2×4 的字数据读入寄存器 R0，并
将新地址 R1 + R2×4 写入 R1。
LDR R0, [R1], R2, LSL # 2 ; 将存储器地址为 R1 的字数据读入寄存器 R0，并将新地
址 R1 + R2×4 写入 R1。
STR R0, [R1], #8 ; 将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8
写入 R1。
STR R0, [R1, #8] ; 将 R0 中的字数据写入以 R1 + 8 为地址的存储器中。
STR R0, [R1, #8] ! ; 将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8
写入 R1。
```

图 3.27 基本指令

下图为反汇编 c 语言程序的指令和界面

```
(gdb) disassemble
No frame selected.
(gdb) b main
Breakpoint 1 at 0x40067c
(gdb) r
Starting program: /root/m21
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.28-36.oe1.aarch64

Breakpoint 1, 0x00000000040067c in main ()
(gdb) disassemble main
Dump of assembler code for function main:
0x000000000400674 <+0>: stp x29, x30, [sp, #-64]!
0x000000000400678 <+4>: mov x29, sp
=> 0x00000000040067c <+8>: str wzr, [x29, #60]
0x000000000400680 <+12>: b 0x4006a4 <main+48>
```

图 3.27 反汇编指令

下图为 main 函数中调用 memcpy(dst,src,len1)的过程。3 个 adrp 指令分别传递 3 个参数。bl 为跳转指令。

# 汇编语言程序设计实验报告

```
0x00000000004006d8 <+100>: adrp    x0, 0x420000 <clock_gettime@got.plt>
0x00000000004006dc <+104>: add     x0, x0, #0x38
0x00000000004006e0 <+108>: ldr     x2, [x0]
0x00000000004006e4 <+112>: adrp    x0, 0x420000 <clock_gettime@got.plt>
0x00000000004006e8 <+116>: add     x1, x0, #0x48
0x00000000004006ec <+120>: adrp    x0, 0x3d58000 <src+59998136>
0x00000000004006f0 <+124>: add     x0, x0, #0x748
0x00000000004006f4 <+128>: bl      0x40072c <memorycopy>
```

图 3.28 函数调用

下图为 memorycopy 函数的反汇编代码，bne 指令为不等时跳转，ret 为返回指令。

```
Breakpoint 1, 0x000000000040072c in memorycopy ()
(gdb) disassemble memprycopy
No symbol table is loaded. Use the "file" command.
(gdb) disassemble memorycopy
Dump of assembler code for function memorycopy:
=> 0x000000000040072c <+0>:      ldrb     w3, [x1], #1
0x0000000000400730 <+4>:      str     w3, [x0], #1
0x0000000000400734 <+8>:      sub     x2, x2, #0x1
0x0000000000400738 <+12>:     cmp     x2, #0x0
0x000000000040073c <+16>:     b.ne    0x40072c <memorycopy> // b.any
0x0000000000400740 <+20>:     ret
0x0000000000400744 <+24>:     nop
End of assembler dump.
```

图 3.29 函数返回

## 3.3 小结

1. VS2019 的功能在 3 个编译器中功能最多，可以很方便地编译、运行、设置断点、调试和查看各种信息窗口。交互性也是最好的，可以同时查看多个项目工程，64 位和 32 位可以直接切换。混合编程很方便操作，而且可以用 invoke 语句调用函数。但每新建一个项目，如果需要编译汇编语言文件，都需要设置 MASM 依赖项，这一点上很不方便。

2. DOSBOX 是一个使用 SDL 库的 DOS 模拟器，这使得 DOSBox 可以很容易地移植到不同的平台。将需要编译运行的 asm 文件放在目录下，同时使用 MASM，LINK，TD 等对汇编程序进行编译和连接等。可以使用 TD 对汇编程序进行观察和 Debug。界面中不能查看和修改源代码，且界面不能上下滚动，TD 界面难以使用鼠标。

3. ARMv8 指令与 X86/X64 指令相差较大，反汇编代码难以看懂。可以在界面中直接查看、新建和修改程序，但界面也不能上下滚动，如果代码较长，就不能在同一界面中显示。使用 gcc 编译，gdb 调试，调试过程较方便。

# 汇编语言程序设计实验报告

---

## 参考文献

- [1]许向阳. x86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2020
- [2]许向阳. 80X86 汇编语言程序设计上机指南. 武汉: 华中科技大学出版社, 2007
- [3]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
- [4]汇编语言课程组. 《汇编语言程序设计实践》任务书与指南, 2022
- [5]王爽. 汇编语言. 第 4 版. 北京: 清华大学出版社, 2020