

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

学 号：

姓 名：

指导教师：

实验时段：

实验地点：

原创性声明

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

特此声明！

学生签名：

报告日期：

实验报告成绩评定：

	1	2	3	4	5
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

课程总体说明	- 3 -
0.1 课程目标	- 3 -
0.2 成绩构成	- 3 -
0.3 实验任务的总体描述	- 3 -
1 编程基础	1
1.1 实验目的与要求	1
1.2 实验内容	1
1.3 任务 1.1 实验过程	5
1.3.1 实验方法说明	5
1.3.2 实验记录与分析	5
1.4 任务 1.2 实验过程	7
1.4.1 实验方法说明	7
1.4.2 实验记录与分析	7
1.5 任务 1.3 的实验过程	8
1.5.1 设计思想及存储单元分配	8
1.5.2 流程图	9
1.5.3 源程序	9
1.5.4 实验步骤	11
1.5.5 实验记录与分析	12
1.6 小结	14
1.6.1 主要收获	14
1.6.2 主要看法	14
2 程序优化	15
2.1 实验目的与要求	15
2.2 实验内容	15
2.3 任务 2.1 实验过程	15
2.3.1 实验方法说明	15
2.3.2 实验记录与分析	16
2.4 小结	17
2.4.1 主要收获	17

汇编语言程序设计实验报告

2.4.2 主要看法	18
3 模块化程序设计	18
3.1 实验目的与要求	18
3.2 实验内容	18
3.3 任务 3.1 实验过程	20
3.3.1 设计思想及存储单元分配	20
3.3.2 流程图	20
3.3.3 源程序	21
3.3.4 实验步骤	22
3.3.5 实验记录与分析	23
3.4 任务 3.2 的实验过程	25
3.4.1 实验方法说明	25
3.4.2 实验记录与分析	25
3.5 小结	26
3.5.1 主要收获	26
3.5.2 主要看法	27
4 中断与反跟踪	27
4.1 实验目的与要求	27
4.2 实验内容	27
4.3 任务 4.1 实验过程	28
4.3.1 设计思想及存储单元分配	28
4.3.2 流程图	29
4.3.3 源程序	29
4.3.4 实验步骤	30
4.3.5 实验记录与分析	30
4.4 任务 4.2 实验过程	31
4.4.1 实验方法说明	31
4.4.2 实验记录与分析	31
4.5 任务 4.3 实验过程	33
4.5.1 实验方法说明	33
4.5.2 实验记录与分析	33
4.6 小结	35
4.6.1 主要收获	35
4.6.2 主要看法	36

汇编语言程序设计实验报告

5 16/32/64 位编程比较	36
5.1 实验目的与要求	36
5.2 实验内容	36
5.3 任务 5.1 实验过程	36
5.3.1 实验方法说明	36
5.3.2 实验记录与分析	37
5.4 任务 5.2 实验过程	38
5.4.1 实验方法说明	38
5.4.2 实验记录与分析	38
5.5 任务 5.3 实验过程	39
5.5.1 实验方法说明	39
5.5.2 实验记录与分析	39
5.6 小结	40
5.6.1 主要收获	40
5.6.2 主要看法	41
参考文献	43

汇编语言程序设计实验报告

课程总体说明

0.1 课程目标

下表是本课程的目标及与支撑的毕业要求指标点之间的关系。请大家关注下表中最后一列“实验中的注意事项”的内容，以便更有针对性的满足课程目标的要求。

课程目标	支撑的毕业要求指标点	实验中的注意事项
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，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。	3.1 掌握与计算机复杂工程问题有关的工程设计和软硬件产品开发全周期、全流程的基本设计/开发方法和技术，了解影响设计目标和技术方案的多种因素。	不能只写代码完成功能，还要有设计、调试、记录、分析等部分的内容。
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。	3.2 能为计算机复杂工程问题设计方案设计满足特定需求的软/硬件模块。	要思考与运用汇编语言的优势编写某些程序。
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。	5.1 了解计算机专业常用的现代仪器、信息技术工具、工程工具和模拟软件的使用原理和方法，并理解其局限性。	熟悉实验中使用的工具，把对工具的看法记录在案。

0.2 成绩构成

实验课程综合成绩由实验过程成绩和实验报告成绩二部分构成。**实验过程成绩：30%**。主要考察各实验完成过程中的情况，希望大家做到预习准备充分，操作认真熟练，在规定的时间内完成实验任务，结果正确，积极发现和提出问题，交流讨论时描述问题准确、清晰。**实验报告成绩：70%**。主要考核报告体现的实验完成质量(含问题的分析、设计思想与程序、针对问题的实验方法与步骤、实验记录、实验结果分析等方面)和报告格式规范等撰写质量方面的内容。

0.3 实验任务的总体描述

本课程安排了8次4学时的课内实验课时，将实现一个具有一定复杂程度的系统。对该系统的相关要求被划分成了**5个主题**：1) 搭建原型系统；2) 在原型系统基础上探索程序指令级别的优化；3) 通过模块化方法调整与优化原型系统的程序结构；4) 通过中断、内存数据和地址操纵、跟踪与反跟踪、加密等措施增强系统安全性；5) 程序在不同平台上的实现。

针对这5个主题，对应地布置了5次实验。**实验1(编程基础)**安排8个课内学时熟悉汇编语

汇 编 语 言 程 序 设 计 实 验 报 告

言程序设计的基本方法、技术与工具，设计实现指定原型系统的主要功能。针对原型系统的搭建，实验报告中要有全周期、全流程的描述。**实验 2（程序优化）**安排 4 个课内学时探索如何通过选择不同的指令及组合关系来优化程序的性能或代码长度。**实验 3（模块化程序设计）**安排 8 个课内学时，利用子程序、模块化程序设计方法、与 C 语言混合编程等，调整与优化程序结构。**实验 4（中断与反跟踪）**安排 8 个课内学时，通过利用中断机制、内存数据和地址操纵技术、跟踪与反跟踪技巧、加密等措施增强系统安全性。**实验 5（16/32/64 位编程比较）**安排 4 个课内学时，熟悉在不同软硬件平台上移植实现指定功能的基本方法。每次实验的侧重面有所不同，但都会涉及到课程目标的三个方面，因此，需要大家在实验过程中以及实验报告中有所注意和体现。

汇编语言程序设计实验报告

1 编程基础

1.1 实验目的与要求

(1) 熟练掌握程序开发平台 (VS2019) 的基本用法, 包括程序的编译、链接和调试; 掌握 DOSBox 下 16 位汇编语言程序开发工具的基本用法;

(2) 熟悉编程的基础知识, 包括数据在计算机内的表现形式、寻址方式、常用指令等;

(3) 熟悉程序运行的基本原理;

(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;

(6) 掌握设计实现一个原型系统的基本方法。

1.2 实验内容

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”, 指出勾选与不勾选该项选时, 反汇编窗口显示内容的差异;

(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;

(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。

(4) 有符号与无符号整型数是如何存储的;

(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?

```
#include <stdio.h>

int a[5] = { 1, 2, 3, 4, 5 };
short x = 100;
short y = -32700; //注意观察初始值较大带来的问题
int psub;
int sum(int a[], unsigned length)
{
    int i;
    int result = 0;
    for (i = 0; i < length ; i++)
        result += a[i];

    return result;
}

int main(int argc, char* argv[])
```

汇编语言程序设计实验报告

```
{
    short z;
    char str[10] = "The end!";
    z = sum(a, 5);
    printf("sum : %d \n", z);
    if (x > y)
        printf("condition1: %d > %d \n", x, y);
    else
        printf("condition1: %d < %d \n", x, y);
    z = x - y;
    printf("condition2: (%d) - (%d) = %d \n", x, y, z);
    psub = &x - &y;
    if (psub < 0)
        printf("condition3: & %d < & %d \n", x, y);
    printf(str);
    return 0;
}
```

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：

（1）使用 VS2019 进行编译、链接和调试。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，说明存储规律，说明各个变量地址之间的关系。如何观察堆栈段？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。

给定的汇编语言源程序如下：

```
.386
.model flat, stdcall
ExitProcess PROTO STDCALL :DWORD
includelib kernel32.lib ;ExitProcess 在 kernel32.lib 中实现
printf PROTO C :VARARG
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

.DATA
lpFmt db "%s", 0ah, 0dh, 0
X db 10, 255, -1
Y dw 10, 255, -1
Z dd 10, 255, -1
U dw ($-Z)/4
STR1 db 'Good', 0
P dd X, Y
Q db 2 dup(5, 6)
buf1 db '00123456789', 0
buf2 db 12 dup(0) ; 12 个字节的空间，初值均为 0
```


汇编语言程序设计实验报告

```
.STACK 200
.CODE
main proc c
    MOV ESI,OFFSET buf1
    MOV EDI,OFFSET buf2
    MOV ECX,0
L1:
    MOV EAX,[ESI] ;如果总数不是 12 个字节，还能每次传送 4 个字节吗？
    MOV [EDI],EAX
    ADD ESI, 4
    ADD EDI, 4
    ADD ECX, 4
    CMP ECX,12
    JNZ L1
    invoke printf,offset lpFmt,OFFSET buf1
    invoke printf,offset lpFmt,OFFSET buf2
    invoke ExitProcess, 0
main endp
END
```

(2) 在 DOSBox 下使用“MASM 6.0, LINK.EXE, TD.EXE, 完整段定义”改造上述汇编语言程序（源程序可以采用记事本等编辑），体会 16 位段程序的编译、链接和调试过程。尝试在调试状态下直接录入或修改代码。

```
.386
DATA SEGMENT USE16
lpFmt db 0ah, 0dh, "$"
X DB 10, 255, -1
Y DW 10, 255, -1
Z DD 10, 255, -1
U DW ($-Z)/4
STR1 DB 'Good', 0
P DD X, Y
Q DB 2 DUP(5, 6)
buf1 DB '00123456789','$' ;结束符号为$
buf2 DB 12 dup(0)
DATA ENDS
STACK SEGMENT USE16 STACK
DB 200 DUP(0)
STACK ENDS
CODE SEGMENT USE16
ASSUME CS:CODE,SS:STACK,DS:DATA
START: MOV AX,DATA
MOV DS,AX
MOV ESI,OFFSET buf1
MOV EDI,OFFSET buf2
MOV ECX,0
L1:
MOV EAX,[ESI]
MOV [EDI],EAX
ADD ESI, 4
```

汇编语言程序设计实验报告

```
ADD EDI, 4
ADD ECX, 4
CMP ECX, 12
JNZ L1
MOV DX, OFFSET buf1 ;采用 DOS 功能调用显示字符串
MOV AH, 9
INT 21H
MOV DX, OFFSET lpFmt
MOV AH, 9
INT 21H
MOV DX, OFFSET buf2
MOV AH, 9
INT 21H
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

任务 1.3 设计实现一个网店商品信息后台管理系统

有一个老板在网上开了一个网店，通过后台管理系统管理相关信息。网店里 n 种商品销售。每种商品的信息包括：商品名称（最长名称 9 个字节，其后加一个数值 0 表示名称结束）、进货价（字类型）、销售价（字类型）、进货数量（字类型）、已售数量（字类型）、利润率（%）【=（销售价*已售数量-进货价*进货数量）*100/（进货价*进货数量），字类型（有符号数）】。老板管理网店信息时需要输入自己的名字和密码，老板登录后可查看指定商品的信息、出货（出售指定数量的某种商品）、补货（给某种商品增加一定的进货数量）、计算商品的利润率，按利润率从高到低显示商品信息等。

该系统被执行后，首先提示输入用户名（即老板名称）和密码，在用户名和密码正确时，显示一个主菜单界面。当用户名和密码错误时，显示用户名错误或者密码错误的提示信息后，退出系统。主菜单界面信息包括：

请输入数字 1...9 选择功能：

1. 查找指定商品并显示其信息
2. 出货
3. 补货
4. 计算商品的利润率
5. 按利润率从高到低显示商品的信息
9. 退出

根据系统的基本需求，可以制定如下的数据段的定义（供参考）：

BNAME DB 'ZHANGSAN', 0 ; 老板姓名

BPASS DB 'U20190001', 0 ; 密码

N EQU 30

GA1 DB 'PEN', 7 DUP(0) ; 商品 1 名称

DW 15, 20, 70, 25, ? ; 进货价、销售价、进货数量、已售数量，利润率（尚未计算）

汇编语言程序设计实验报告

GA2 DB 'PENCIL', 4 DUP(0) ; 商品 2 名称

DW 2, 3, 100, 50, ?

GA3 DB 'BOOK', 6 DUP(0) ; 商品 3 名称

DW 30, 40, 25, 5, ?

GA4 DW 'RULER', 5 DUP(0) ; 商品 4 名称

DW 3, 4, 200, 150, ?

GAN DB N-4 DUP('TempValue', 0, 15, 0, 20, 0, 30, 0, 2, 0, ?, ?);除了 4 个已经具体定义了的商品信息以外, 其他商品信息暂时假定为一样的。

本次实验主要是利用分支、循环程序的结构, 在 VS2019 下实现该系统的部分功能, 并熟悉全周期、全流程地设计实现一个原型系统的基本方法。本次实验要具体实现的功能要求如下:

查找指定商品并显示其信息: 提示用户输入商品名称; 用户输入名称后, 在商店中寻找是否存在该商品; 若存在, 显示找到的商品信息; 若没有找到, **提示没有找到**。最后都返回到主菜单界面。

出货: 输入商品的名称及本次销售数量。判断输入数据的有效性, 即剩余数量应大于等于本次销售数量, 修改商品的已售数量。若商品未找到或数据无效, 则提示错误。最后都返回到主菜单界面。

补货: 输入商品的名称及本次增加的数量。找到该商品, 修改商品的进货数量。若商品未找到, 则提示错误。最后都返回到主菜单界面。

计算商品的利润率: 按照利润率计算公式依次计算所有商品的利润率。

1.3 任务 1.1 实验过程

1.3.1 实验方法说明

1. 将给定 C 语言程序在 VS2019 上进行编译、链接和调试。
2. 进行逐行运行, 在反汇编窗口、寄存器窗口、监视窗口、内存窗口观察各内容的显示。观察并了解各窗口显示信息内容。
3. 分别进行有符号数和无符号数的加减运算, 并查看各窗口显示内容。
4. 观察有符号数储存在无符号类型变量中的状态。
5. 观察地址差值的计算处理细节。

1.3.2 实验记录与分析

1. 实验环境条件: Intel Core i5-8250U 1.6GHz, 8M 内存; WINDOWS 10 下 Visual Studio 2019。
2. 在反汇编窗口下, 查看勾选符号名和未勾选的区别。查看包含 int 类型的局部变量 i 的相关反汇编语句, 如图 1.1a、图 1.1b 所示, 该语句为 i=0 的反汇编语句。

```
mov     dword ptr [i], 0
```

图 1.1a 勾选显示符号名的反汇编语句

```
mov     dword ptr [ebp-8], 0
```

图 1.1b 未勾选显示符号名的反汇编语句

由于局部变量存储在堆栈中, 应该通过堆栈的栈帧 ebp 进行查找, 如图 1.1b 所示。但对观察

汇编语言程序设计实验报告

者十分不友好，因为需要通过计算才能弄清该位置单元获取的数据信息，VS2019 提供了显示符号命名的功能，极大方便了对局部变量、跳转地址标号等的观察与分析。

3. 显示各个窗口，显示各类寄存器，如标志寄存器（如图 1.4 所示）。显示监视窗口。显示内存窗口，具体观察如下。

4. 在此环境中，设置 short 类型全局变量 x, y，并将其赋值 100 与 -32700。观察其内存存放数据，前 4 字节为 x 数据，后 4 字节为 y 数据（如图 1.2 所示）。

0x0086A014 64 00 00 00 44 80 00 00

图 1.2 short 类型全局变量 x, y 在内存中的存储

发现 short 为 2 字节类型，而在内存中占据 4 个字节，这是为方便对齐而进行的占位操作，超出字节使用 0 进行占位，造成了存储浪费。

再尝试将 x, y 的类型改变为 unsigned short 类型，其内存存放数据与图 1.2 相同。

5. 当进行减法运算时，反汇编中显示，对 short 类型全局变量 x, y 进行了扩展传输指令后扩展为 4 字节数据（如图 1.3 所示）。故进行运算时使用 32 位寄存器来运算，结果为 00008020，通过标志寄存器中的标志位显示，无溢出而有进位（如图 1.4 所示）。

EAX	00000064
ECX	FFFF8044

图 1.3 寄存器存储 short 类型数据（eax 存储 100，ecx 存储 -32700）

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

图 1.4 运算后的标志寄存器的各标志位（OV 显示有无溢出，CY 显示有无进位）

发现 C 语言进行运算操作时，不足 4 字节位数自动进行符号扩展。而此时判断溢出时，则根据寄存器最高位决定，而非该数据类型最高位。

6. 对与 short 类型有符号数 x, y 进行加减运算及比较运算时，都会进行符号扩展传送指令操作。对于加减运算后，最终取得运算结果的后 16 位作为最终结果（如图 1.5 所示）。而在无符号数 x, y 进行运算时，则使用无符号扩展指令，相应操作类似。（如图 1.6 所示）

EAX	00008020
ECX	FFFF8044
EBP	012FF7BC
AX	8020

图 1.5 结果进行去位

```
movsx    eax, word ptr [x (065A014h)]    movzx    eax, word ptr [x (055A014h)]
movsx    ecx, word ptr [y (065A018h)]    movzx    ecx, word ptr [y (055A018h)]
sub      eax, ecx                        sub      eax, ecx
mov      word ptr [z], ax                mov      word ptr [z], ax
```

图 1.6 符号扩展指令与无符号扩展指令（前者为有符号数运算，后者为无符号数运算）

在进行比较指令时，与加减法指令相同，有符号数进行比较时，进行 movsx 有符号扩展比较；无符号数进行比较时，进行 movzx 无符号扩展比较。

7. 在计算 x 与 y 的地址差时，查看差值 psub 的计算过程。（如图 1.7 所示）

汇编语言程序设计实验报告

```
28:      psub = &x - &y;
008A19AA B8 14 A0 8A 00      mov     eax,offset x (08AA014h)
008A19AF 2D 18 A0 8A 00      sub     eax,offset y (08AA018h)
008A19B4 D1 F8              sar     eax,1
008A19B6 A3 58 A1 8A 00      mov     dword ptr [psub (08AA158h)],eax
```

图 1.7 地址差值计算

发现出现 sar 算数右移指令，等同于将 x 与 y 的地址差值除以 2。分析得知 x 与 y 为 short 类型，占用 2 个字节，故 C 语言计算地址差值时会自动计算字节长度，类似的有数组地址等。

1.4 任务 1.2 实验过程

1.4.1 实验方法说明

1. 分别使用 VS2019 和 DOXBox 对 1.2 实验汇编语言源程序进行编译、链接和调试。
2. 对于在 VS2019 下操作：逐行运行，通过反汇编窗口、寄存器窗口、监视窗口和内存窗口的观察与理解。特别观察数据存储地址，查看各变量地址位置关系。并将寄存器间接寻址改为变址寻址。
3. 对于在 DOSBox 下进行操作：通过 DOSBox 进行编译、链接和运行，通过 TD 指令进入调试界面，掌握相关操作。

1.4.2 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 Visual Studio 2019。
2. 观察数据在内存中的存储方式，字节类型数据 X 的存储数据为 10，255，-1。发现数据 255 与 -1 的存储内容一致。（如图 1.8 所示）分析发现，内存存在存储数据时，若视为有符号数超出存储范围，则会将其作为不超出范围的无符号数存储。但其差别会在对应数据处理时产生。

0x00D48005 0a ff ff

图 1.8 X 定义中数据在内存中的存储

3. 观察各个变量地址间的位置关系。不难看出，地址之间的位置关系以字节为单位，X 与 Y 相隔 3 个字节数据，故地址相差 3；Y 与 Z 相隔 3 个字类型数据，故地址相差 6。以此类推。
4. 观察堆栈段的内容，需要得知 esp 或 ebp 的值，可以通过寄存器窗口查询。在 invoke 调用函数时，会出现入栈操作，在此将修改后的 ebp 的值代入到内存查询窗口处，即可观察到堆栈内容。
5. 将寄存器间接寻址改为变址寻址。实验发现，由于 buf1 为字节类型数据，则使用 AL 来存储对应数据。（如图 1.9 所示）

```
MOV ESI,OFFSET buf1      MOV ECX,0
MOV EDI,OFFSET buf2
MOV ECX,0
L1:
MOV EAX,[ESI]            MOV AL,buf1+[ECX]
MOV [EDI],EAX            MOV buf2+[ECX],AL
ADD ESI,4                ADD ECX,1
ADD EDI,4                CMP ECX,12
ADD ECX,4                JNZ L1
CMP ECX,12
JNZ L1
```

图 1.9 寄存器间接寻址（左） 变址寻址（右）

汇编语言程序设计实验报告

6. 在 DOSBox 下生成运行程序，并运行（如图 1.10 所示）。进入 TD 界面，查看 CPU 界面窗口（如图 1.11 所示），并进行数据段查找等操作。

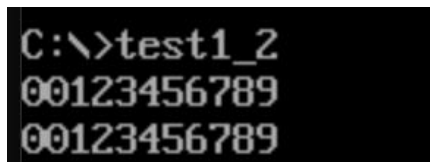


图 1.10 DOSBox 下的运行界面

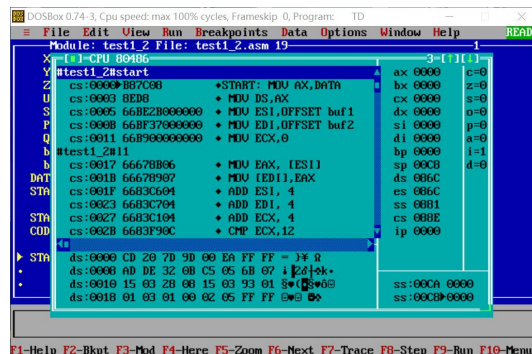


图 1.11 TD 模式下的 CPU 界面

1.5 任务 1.3 的实验过程

1.5.1 设计思想及存储单元分配

设计一个网店商品信息后台管理系统的大致思路为：验证用户名及密码，循环操作，退出程序。循环操作则为输入对应操作码进入相应流程中，运行完成后再返回到主流程中，如此循环。

对于用户名及密码验证阶段需要分配储存空间存储正确的用户名和密码以及输入的用户名和密码，循环操作中需要分配空间存储提示信息以及操作码、输入值。

1. 存储单元分配

1) 商品信息

变量名称：GA1, GA2, ...

商品名：使用 10 个字节存储空间, 最后以 0 作结；

进货价、销售价、进货数量、已售数量、利润率：分别用 2 个字节存储空间存储

2) 用户名及密码验证

ANAME: (字节变量) 存放键入的用户名。

APASS: (字节变量) 存放 x 的立方值。

BNAME: (字节变量) 存放正确的用户名。

BPASS: (字节变量) 存放正确的密码。

Note_1-4: (字节变量) 存放需要使用的提示信息。

3) 循环操作

Note5-13: (字节变量) 用于存放提示信息。

num: (字节变量) 用于存放输入的操作码。

GOOD: (字节变量) 用于存放待操作商品名称。

DATA: (字变量) 用于存放输入的数据。

2. 寄存器分配

EBX: 用于计数读取商品的数量。

汇编语言程序设计实验报告

ESI：用于存放查找商品的偏移地址。

EAX、ECX：临时寄存器。

1.5.2 流程图

图 1.12 是任务 1.3 实现一个网店商品信息后台管理系统的程序流程图。

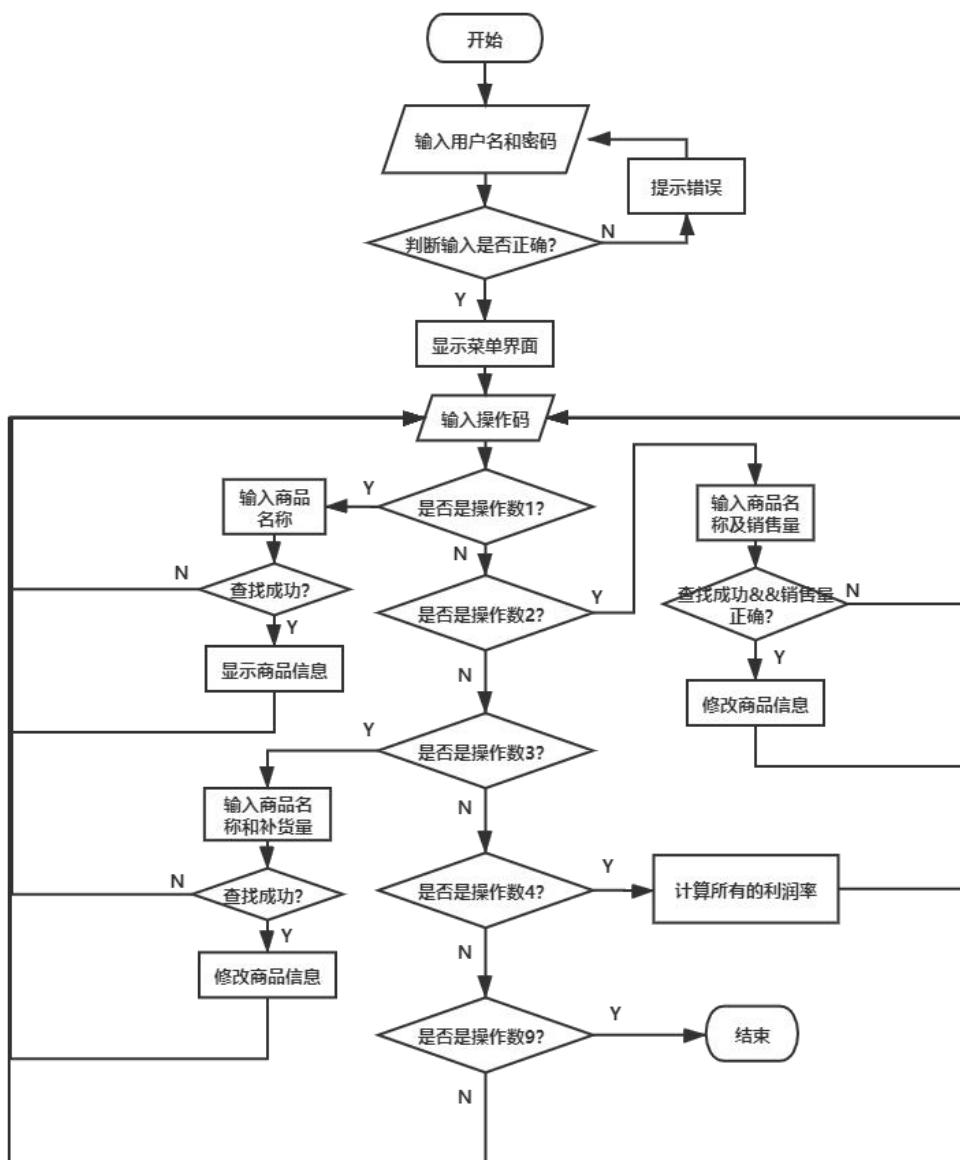


图 1.12 网店商品信息后台管理系统的程序流程图

1.5.3 源程序

```
.686P
.model flat, c
```

汇编语言程序设计实验报告

```
.....
.data
.....
ANAME DB 20 DUP(0) ;储存输入名称
APASS DB 20 DUP(0) ;存储输入密码
BNAME DB 'DONGCHENGCHENG',0 ;老板姓名
BPASS DB 'U201914984',0 ;密码
N EQU 30
GA1 DB 'PEN', 7 DUP(0) ;商品 1 名称
DW 15,20,70,25,? ;进货价、销售价、进货数量、已售数量, 利润率(尚未计算)
GA2 DB 'PENCIL', 4 DUP(0) ;商品 2 名称
DW 2,3,100,50,?
GA3 DB 'BOOK', 6 DUP(0) ;商品 3 名称
DW 30,40,25,5,?
GA4 DB 'RULER',5 DUP(0) ;商品 4 名称
DW 3,4,200,150,?
GA5 DB 'WATER',5 DUP(0) ;商品 5 名称
DW 2,4,150,140,?
GAN DB N-5 DUP('TempValue',0,15,0,20,0,30,0,2,0,?,?)

.stack 200
.code
main proc
start:
    invoke printf,offset Note_1 ;输入名称
    invoke scanf,offset lpfmt1,offset ANAME
    invoke printf,offset Note_2 ;输入密码
    invoke scanf,offset lpfmt1,offset APASS
    invoke strcmp,offset ANAME,offset BNAME ;比较用户名
    cmp eax,0
    jne errorname
    invoke strcmp,offset APASS,offset BPASS ;比较密码
    cmp eax,0
    jne errorpass
    invoke printf,offset Note_5
input:
    invoke printf,offset Note_6 ;输入操作数
    invoke scanf,offset lpfmt2,offset num
    cmp num,1
    je func1
    cmp num,2
    je func2
    cmp num,3
    je func3
    cmp num,4
    je func4
    cmp num,9
    je exit

..... ;信息查找错误跳转内容

func1: ;操作 1: 查找商品
.....
Loop1:
    invoke strcmp,offset GOOD,ESI
    ADD ESI,20
    ADD EBX,1
```


汇编语言程序设计实验报告

```
    cmp EBX,N
    jge errorgood
    cmp EAX,0
    jne Loop1
    sub ESI,20
    movsx eax,word ptr[ESI+18]
    invoke printf,offset lpfmt3,ESI,word ptr[ESI+10],word ptr[ESI+12],word ptr[ESI+14],word
ptr[ESI+16],eax
    jmp input

func2: ;操作 2: 出货操作
    .....
    jmp input

func3: ;操作 3: 进货操作
    .....
    jmp input

func4: ;计算利润率 = (销售价*已售数量-进货价*进货数量)*100/(进货价*进货数量) 【有符号数】
    MOV ESI,offset GA1
Loop4:
    invoke strcmp,offset GAN,ESI
    CMP EAX,0
    je next
    MOVZX EAX,word ptr [ESI+12]
    MOVZX ECX,word ptr [ESI+16]
    imul EAX,ECX ;eax 存储销售额
    imul EAX,100
    MOVZX EBX,word ptr [ESI+10]
    MOVZX ECX,word ptr [ESI+14]
    imul ECX,EBX ;ecx 存储进货额
    MOV EBX,ECX
    imul EBX,100
    sub EAX,EBX ;eax 存储利润
    cdq
    idiv ECX
    MOV word ptr[ESI+18],AX
    ADD ESI,20
    jmp Loop4
next:
    invoke printf,offset Note_13
    jmp input

exit:
    invoke ExitProcess,0
main endp
end
```

1.5.4 实验步骤

1. 准备上机实验环境，使用 Visual Studio Code 编写汇编源程序，在 Visual Studio2019 上进行调试运行。

2. 运行该程序。分别进行正确指令和错误指令的操作，查看相应输出是否正确。

3. 详细操作步骤如下。

(1) 分别输入错误的用户名和密码，查看输出。输入正确的用户名和密码，查看输出。

汇编语言程序设计实验报告

- (2) 进入菜单界面，输入错误操作码，查看输出。
 - (3) 输入操作码 4，进入商品利润率计算界面。
 - (4) 输入操作码 1，进入查找商品界面。输入错误商品名称，查看输出；输入正确商品名称，查看输出。
 - (5) 输入操作码 2，进入出货操作界面。输入错误信息，查看输出；输入正确信息，查看输出。
 - (6) 输入操作码 3，进入补货操作界面。输出错误信息，查看输出；输入正确信息，查看输出。
 - (7) 输入操作码 9，查看是否正确退出系统。
4. 对于细节方面，进行如下操作调试程序错误：
- (1) 查看反汇编窗口，在进行商品名称查找时，逐行调试，查看寄存器数值变化规律。
 - (2) 在进行商品利润率计算时，逐行调试，查看各步骤有符号数运算结果及运算寄存器存储数值的变化。

1.5.5 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 Visual Studio 2019。
2. 运行该程序，进行详细操作。

(1) 输入错误的用户名或密码，显示如图 1.13 所示。输入正确的用户名和密码，显示如图 1.14 所示。

```
请输入用户名: ZHANGSAN  
请输入密码: 123456  
用户名错误, 请重新输入!  
  
请输入用户名: DONGCHENGCHENG  
请输入密码: 123456  
密码错误, 请重新输入!
```

图 1.13 验证用户名或密码错误显示

```
请输入用户名: DONGCHENGCHENG  
请输入密码: U201914984  
  
菜单: 1. 查找商品; 2. 出货; 3. 补货; 4. 利润率; 9. 退出;  
输入操作:
```

图 1.14 验证用户名或密码正确显示

(2) 输入错误操作码，查看界面。（如图 1.15 所示）

```
菜单: 1. 查找商品; 2. 出货; 3. 补货; 4. 利润率; 9. 退出;  
输入操作: 5  
操作码输入错误! 请重新输入!
```

图 1.15 操作码错误显示

(3) 输入操作码 4，进行商品利润率计算后，再输入操作数 1，输入错误商品名称，输出查找错误（如图 1.16 所示）；输入正确商品名称，输出商品信息（如图 1.17 所示）。

```
输入操作: 1  
输入商品名: GLASS  
未查找到商品!
```

图 1.16 查找错误显示

```
输入操作: 1  
输入商品名: PEN  
商品名: PEN, 进货价: 15, 销售价: 20, 进货数量: 70, 已售数量: 25, 利润率: -52%
```

图 1.17 显示商品信息

(4) 输入操作码 2，进行出货操作。输入错误信息，如输入错误的进货值，输入错误的商品名（如图 1.18 所示）；输入正确信息，输出修改后的信息。（如图 1.19 所示）

汇编语言程序设计实验报告

```
输入操作: 2
输入商品名: GLASS
未查找到商品!

输入操作: 2
输入商品名: PEN
请输入销售数量: 100
数据输入错误!
```

图 1.18 输入错误显示

```
输入操作: 2
输入商品名: PEN
请输入销售数量: 15
商品名: PEN, 进货价: 15, 销售价: 20, 进货数量: 70, 已售数量: 40
```

图 1.19 输入正确显示

(5) 输入操作码 3，进入补货操作。输入错误商品名，显示如图 1.20 所示。输入正确商品名和补货数，显示如图 1.21 所示。

```
输入操作: 3
输入商品名: GLASS
未查找到商品!
```

图 1.20 输入错误显示

```
输入操作: 3
输入商品名: PENCIL
请输入进货数量: 10
商品名: PENCIL, 进货价: 2, 销售价: 3, 进货数量: 110, 已售数量: 50
```

图 1.21 输入正确显示

3. 进行调试时，遇到如下错误。

(1) 在进行输出函数调用时，若在函数语句中放置休止符，而非数据段定义中，则会出现错误输出。数据段输出信息定义如图 1.22 所示。

```
lpfmt3 DB '商品名:%s, 进货价:%d, 销售价:%d, 进货数量:%d, 已售数量:%d, 利润率:%d%%'
lpfmt4 DB '商品名:%s, 进货价:%d, 销售价:%d, 进货数量:%d, 已售数量:%d'

invoke printf, offset lpfmt3, ESI, word ptr [ESI+10], word ptr [ESI+12], word ptr [ESI+14], word ptr [ESI+16], eax, 0
```

图 1.22 输出信息数据段定义和函数调用

分析发现，在 `printf` 引用语句时，会以休止符 0 为结尾，而当读取 `lpfmt3` 地址下的内容时，会一直读取直至遇到休止符，故出现错误显示。所以，在数据定义时，要养成以休止符作结的习惯。

(2) 在进行负数输出时，如果直接调用 `printf` 输出字类型数据，则会当成无符号数处理。错误代码如图 1.23 所示。

```
invoke printf, offset lpfmt3, ESI, word ptr [ESI+10], word ptr [ESI+12], word ptr [ESI+14], word ptr [ESI+16], word ptr [ESI+18]
```

图 1.23 输出负数错误调用

分析后，`printf` 函数会将 16 位数据进行无符号扩展传送，因而会将其转化为正数形式，因而需要在 `printf` 前，将该 16 位数据进行有符号扩展传送到寄存器中，再进行输出就能正常输出负数了。

(3) 在进行除法运算时，未对 `edx` 进行对应处理。导致除法运算结果总是出现错误。

分析可知，除法运算的被除数又两个寄存器内的数据组成，高 32 位为 `edx`，低 32 位为 `eax`。故需要对 `edx` 做相应处理，如 `eax` 最高位为 0 则 `edx` 置 0；若 `eax` 最高位为 1，则 `edx` 的置 0ffffffffH。对应的指令为 `cdq`，故在进行除法运算前应使用 `cdq` 指令。

汇编语言程序设计实验报告

1.6 小结

1.6.1 主要收获

通过任务 1.1 的实验，让我了解了 C 语言与汇编语言的联系，以及在 VS2019 中各窗口的运用。通过任务 1.2 的实验，则是在两个不同的环境中编译和运行源程序，掌握了 DOSBox 的基础操作，以及寄存器间接寻址与变址寻址的转换。通过任务 1.3 的实验，了解了汇编源程序的组成以及各个指令的使用。

任务 1.1 的实验为探究性实验，通过观察反汇编窗口，了解了 C 语言定义变量、数组等数据在内存中的存储方式，以及通过指令实现各类语句的方法。如 C 语言中 short 类型的数据在内存中不仅占有 2 个字节的空間，还会有 2 个字节被 0 占据；在进行运算时，会将有符号数通过 movsx 指令进行传输，而无符号数则通过 movzx 指令进行传输；对地址差值进行计算时，是通过右移指令对其差值进行除以类型字节长度处理，从而得到 C 语言概念中地址的差值。这些在 C 语言学习中忽略的细节，在汇编语言的学习中得到了巩固。

任务 1.2 中，则主要学习的通过 VS2019 和 DOSBox 编译、链接、调试、运行汇编语言源程序。主要了解了 VS2019 的运行环境，学会了使用逐行调试。另外，还掌握了寄存器间接寻址和变址寻址的转换。在 MOV 操作时，特别要注意源操作数和目的操作数的寻址方式不能全为存储器寻址。

任务 1.3 是最为关键性的一个实验，它使我了解了整个汇编语言程序的搭建框架，调试运行流程。除此之外，还收获了许多经验教训。

首先，要学会流程分解。此系统包含多个操作流程分支，分支集结到主流程中。则要理清主流程和分支流程的关系，此系统主流程为：验证用户名和密码 -> 分支流程入口选择 -> 退出程序；而分支流程包含 4 个操作，则根据编号进行对应分支流程设计。通过分解，可以将大问题分化成多个小问题，逐个击破。

其次，在调用 C 语言函数库时，EAX、ECX、EDX 会发生变化，这是一种潜移默化的规定。通过此次实验，让我了解了在今后进行函数编写时，调动 EAX、ECX 和 EDX 寄存器是规范化的操作。因此，在编写程序前，需要了解一些规范化规则，这样才会使自己的程序更加泛用。

最后，则是在负数输出上出现了问题，负数保存后使用 printf 并未输出正确的负数，而是将其作为无符号数输出。发现，printf 输出 %d 时，会将数据扩展为 32 位，而此时扩展方式为无符号扩展传输，故需要编写指令先将该负数进行有符号扩展为 32 位，再进行输出则显示正确。

通过本次实验，掌握了 VS2019 的使用；通过观察反汇编、内存和寄存器窗口显示，对汇编语言源代码进行调试；学会了使用汇编语言实现存储空间的分配，函数的调用以及分支、循环的使用。

1.6.2 主要看法

本次实验进行了 VS2019 和 DOSBox 的使用对比。两者差别十分显著，VS2019 对调试和运行比较友好，不需要复杂的界面操作，直接点击即可，并且界面美观度与适用度比 DOSBox 好，操作难度不大，对新手十分友好。而 DOSBox 则在各界面的显示上略胜一筹，查看寄存器、内存和 CPU 比 VS2019 方便。我认为此次实验的任务 3 搭建系统实验设计非常好，让初学者能够在实验过程中学会各类指令的使用、函数调用以及分支、循环的实现等，十分全面。

2 程序优化

2.1 实验目的与要求

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

2.2 实验内容

任务 2.1 对任务 1.3 的程序进行完善和优化

先实现“5. 按利润率从高到低显示商品信息”的功能（该功能先按照商品利润率的大小排序，然后显示排序后的商品信息，显示完毕后返回主菜单），然后对功能“4”和“5”的程序做些改造，以便计时与多次循环（多次循环的目的仅仅是为了更明显地观察程序执行的时间）。

改造之后的程序流程为：当输入“4”时，在原来“4”的代码之前先执行新增的计时开始的代码，然后再执行新增的一段计数的循环控制程序，接着执行原来“4”的功能，执行完后不要返回到主菜单，而是直接执行“5”的功能，“5”的功能执行完后，也不要返回到主菜单，而是回到计数的循环控制程序；当计数未结束时，重复执行“4”和“5”功能，直到计数完毕，才跳转到新增的计时结束的代码处，显示了计时的时间之后回到主菜单。

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）。同时，为增强程序的可读性，要求将商品信息定义成一个结构，参考下列示例。

```
GOODS STRUCT
    GOODSNAME db 10 DUP(0)
    BUYPRICE  dw 0
    SELLPRICE dw 0
    BUYNUM    dw 0
    SELLNUM   dw 0
    RATE      dw 0
GOODS ENDS
```

对商品信息的访问都通过该结构进行。

2.3 任务 2.1 实验过程

2.3.1 实验方法说明

1. 首先优化代码的可读性，将商品信息统一用结构定义，并将商品信息的访问进行对应的修改。同时将操作 4 和操作 5 设计为调用函数的方式，采用堆栈法传递参数。
2. 设计利润率从高到低排列算法，采用插入排序算法构造初始功能。另外，为了避免数据的整体交换导致操作十分复杂，故创建新的结构体，用于存放排序后的各数据首地址，进而通过首地址

汇编语言程序设计实验报告

寻址到对应位置按序输出即可。这种方式将不会更改存储单元的数值，从而十分便捷。

3. 进行相关优化，采用 timeGetTime 计时方式，操作 4 与操作 5 循环 1 000 000 次记录运行时间。

4. 优化环节包括（1）指令层面：用等价语句改变相关操作，对循环语句中的代码长度进行改变；（2）运行环境层面：软件环境：分别在 VS2019 下的调试环境和非调试环境运行；硬件环境：分别在 Core i5-8250U 环境下和 AMD Ryzen R5 2500U 环境下运行。

2.3.2 实验记录与分析

1. 初始情况下：使用插入排序，设计算法为将数据从小到大排序。故设计数据为从大到小的最差情况，测试运行用时。（如图 2.1 所示）

2. 而发现再拔取电源后，测试运行时间由大幅度增加。（如图 2.2 所示）得知，在运行程序时，为接近最优效果，应该在外部供电的环境中运行程序。故为了在后续优化减少变量的干扰，统一使用在充电的环境下进行测试。



图 2.1 初始运行时间（充电环境下）

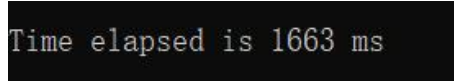


图 2.2 无充电下的初始代码运行时间

3. 指令层面。

由于统计的时间在一定范围内波动，故采用测量 5 次取平均的方式进行时间测试。

（1）在排序时，进行计数，不采用 32 位寄存器而改用 16 位寄存器后，运行时间如图 2.3 所示。

						平均值
32位(ms)	400	413	406	406	432	411.4
16位(ms)	482	433	483	455	437	458

图 2.3 使用 32 位寄存器与 16 位寄存器的差别

发现，使用 16 位寄存器时程序运行时间比 32 位寄存器时要长，运行效率不如 32 位寄存器。分析可知，直接使用 32 位寄存器时计算机 CPU 中无需进行多余的操作，而使用 16 位时，则需进行如扩展等的操作造成时间的浪费。并且，32 位寄存器能够存储更多的数据，故使用 32 位寄存器即可。

（2）减少外层循环语句数。

进行计数时可不进行基数的加法，而直接进行存储单位长度的加法。（如图 2.4 所示）

<pre>mov EAX, 1 Loop5: CMP EAX, 28 ja next5 MOV EDI, ORIID mov ebx, eax imul ebx, 6 ADD EDI, EBX SUB EAX, 1 MOV ESI, ORIID mov ebx, eax imul ebx, 6 ADD ESI, EBX ADD eax, 1 MOV EDX, [EDI].GOODSID.ID MOV CX, [EDI].GOODSID.RATES</pre>	<pre>mov EAX, 6 Loop5: CMP EAX, 168 ja next5 MOV EDI, ORIID ADD EDI, EAX SUB EAX, 6 MOV ESI, ORIID ADD ESI, EAX ADD eax, 6 MOV EDX, [EDI].GOODSID.ID MOV CX, [EDI].GOODSID.RATES</pre>
---	--

图 2.4 改变计数方式（左图更改前，右图更改后）

汇编语言程序设计实验报告

更改前后的平均运行时间分别为，更改前：411.4ms，更改后：322.8ms。实现了极大的提升。虽然损失了一些代码的可读性，但使其运行速度得到了极大提升，利大于弊。

(3) 减少内层循环语句数。

在第(2)步优化的基础上，对内层循环语句进行删减。如进行比较语句时，直接采用变址寻址的做法。(如图 2.5 所示)

<code>mov BX, [ESI].GOODSID.RATES</code>	<code>CMP CX, [ESI].GOODSID.RATES</code>
<code>CMP CX, BX</code>	<code>jge next6</code>
<code>jge next6</code>	

图 2.5 改变内层循环语句数(左图更改前，右图更改后)

更改前后运行时间分别为，更改前：320.3ms，更改后：308.6ms。仅仅减少了一条指令，却带来较为明显的效率提升。由此可见，优化的核心在于内层循环的语句优化。

4. 运行环境层面。

(1) 软件环境。

在上述优化的基础上，在 VS2019 下的调试环境和非调试环境运行进行对比。(如图 2.6 所示)

Time elapsed is 303 ms

Time elapsed is 301 ms

图 2.6 左图为调试环境，右图为非调试环境

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

(2) 硬件环境。

在上述优化基础上，分别在 Core i5-8250U 环境下和 AMD Ryzen R5 2500U 环境下运行对比。(如图 2.7 所示)

Time elapsed is 306 ms

Time elapsed is 584 ms

图 2.7 左图为 Core i5-8250U，右图为 AMD Ryzen R5 2500U

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

2.4 小结

2.4.1 主要收获

任务 2.1 包含两个过程：实现函数和程序优化。实现函数过程，则是将 C 语言学习的排序算法用汇编语言实现。而程序优化则是一次通过运行时间来判断运行效率的过程，从而达到效率最优化。

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

汇编语言程序设计实验报告

行比较，可以明显发现，汇编语言中的基数并非 C 语言中数组的基数，而是数据存放的字节数，这样可以使效率显著提升。

而在程序优化环节，则主要考虑在指令层面进行优化。首先对寄存器的运用进行比对，发现 32 位寄存器优于 16 位寄存器，这与 CPU 的寄存器为 32 位有关，故在后续程序设计时，使用 32 位寄存器较优。而在指令删改过程中，发现指令的删减能够提升不少运行效率，尤其是对内层循环的优化，上述实验中，内层语句仅删减一条就可以得到较优的运行效率。故在往后的程序设计中，优化要将内层循环视为核心优化点。

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

2.4.2 主要看法

本次实验主要考虑指令层面的优化，发现在程序编写完成后实现优化比较困难。故希望能够通过实验规定一些要求的基础上，再进行程序的构建与优化。可以更好的考察优化处理细节，从而避免了“无话可说”的情况。并且在规定的同时，也能够有一定的优化空间，而非造成在构建程序时已经构成了最优运行效率，迫使负优化的情况。

3 模块化程序设计

3.1 实验目的与要求

- (1) 掌握子程序设计的方法与技巧，熟悉子程序的参数传递方法和调用原理；
- (2) 掌握宏指令、模块化程序的设计方法；
- (3) 掌握汇编语言程序与 C 语言程序混合编程的方法；
- (4) 掌握较大规模程序的开发与调试方法；理解模块之间的信息传递与组装的基本方法；
- (5) 完成指定功能的程序设计与调试。

3.2 实验内容

任务 3.1 采用子程序、宏指令、多模块等编程技术调整经过任务 2.1 完善后的商品信息后台管理系统。

要求：

- (1) 实现字符串比较的子程序，用户名、密码及商品名的比较均调用该子程序。
- (2) 出售商品、计算商品的利润率、利润排序等均用子程序实现。
- (3) 要有一个有参数的子程序，其调用使用 invoke 伪指令，并且在该子程序有形参，定义并

汇编语言程序设计实验报告

使用了局部变量。

(4) 至少定义一个带有形参的宏指令。

(5) 将汇编源代码至少分解到两个不同的源文件中。

(6) 在实验中, 要对回答如下问题: 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? invoke 伪指令对应的汇编语句有哪些? 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限制?

(7) 探究:

a) 对一个 NEAR 类型子程序强制使用 FAR 调用 (即 CALL FAR PTR 子程序名) 会怎样? 反之, 对一个 FAR 类型的子程序 (子程序可以与主程序在同一个代码段, 也可以在不同的代码段) 强制使用 NEAR 调用又会怎样?

b) 观察不同模块的可合并段合并后变量偏移地址的变化情况。观察不同段在内存里的放置次序。体会模块间段的定义及其对应的装配方法。观察段合并与不合并时对程序的影响。

c) 观察模块间的参数的传递方法, 包括公共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?

d) 通过调试工具观察宏指令在执行程序中的替换和扩展, 解释宏和子程序的调用有何不同。

任务 3.2 C 语言和汇编语言混合编程

对任务 3.1 中的程序, 进行改造。

1、用户登录功能、主菜单的显示功能用 C 语言程序实现;

2、用 C 语言实现, 增加 “6. 添加新商品” 的功能;

3、其他功能模块仍用汇编语言实现。

回答如下问题:

(1) 在 C 语言程序、汇编语言程序中, 分别是如何说明外部变量和函数的? 汇编指令访问 C 的变量时是如何翻译的 (观察对应的反汇编代码)? C 语言语句访问汇编语言定义的变量时是如何翻译的?

(2) 如何保证在 C 语言程序和汇编语言程序中, 正确访问商品信息的结构数组?

(3) 观察不同变量地址之间的关系; 根据该关系, 实现一个变量名称不出现在语句中的情况下, 修改该变量值的功能。 (比如, 已知 int x, y; 假设这两个变量在内存中相邻, 就可以用表达式 *(&x-1)=20 修改 y 的值)

(4) 地址类型转换的含义是什么? (比如, char a[10]; 一种地址类型转换的做法: *(int *)a=123;)

(5) 函数调用语句对应的汇编语句有哪些? 调用函数与被调用函数之间是如何传递信息的?

对混合编程形成的执行程序, 用调试工具观察由 C 语言形成的程序代码与由汇编语言形成的程序代码之间的相互关系, 包括段、偏移的值等。

3.3 任务 3.1 实验过程

3.3.1 设计思想及存储单元分配

此次实验要求运用多模块的方式来对系统进行再构建。根据要求，由于需构建多个函数，故采用构建两个模块的方式对系统进行管理，即构建主模块与子模块。

主模块中：包含创建子程序声明，公共符号的声明，结构体的声明，数据段存放主程序说明变量定义，代码段中则存放主流程程序。

子模块中：外部符号声明，数据段中存放构建函数时待续的变量，代码段中存放构建的子程序定义。

模块之间的通信“协议”包含两个部分，分别为子程序与公共变量。

构建主要子程序如下：

(1) 字符串比较子程序 `my_strcmp`；函数类型 NEAR；语言类型 C；形参设置两个 `dword` 类型 `buf1` 与 `buf2`。

协议描述：主程序中调用该函数时，传递待比较字符串的存放首地址；函数中设置局部变量用于记录读取长度。若两个字符串相同，则将 `eax` 置 0；否则，将 `eax` 置 1。在主程序中通过 `eax` 的取值判断运行结果。

(2) 出售商品子程序 `sell_goods`；函数类型 NEAR；语言类型 C；对 `ESI` 进行现场保护；形参设置 2 个，分别为 `dword` 类型 `TARID` 与 `word` 类型 `SellData`。

协议描述：主程序传递待修改商品单元存储首地址与销售量至子程序中；函数中进行商品数据的修改，修改成功，则将 `eax` 置 1；反之置 0。主程序中对 `eax` 进行比较处理，进而进行对应后续流程。

(3) 计算利润率子程序 `count_profits`；函数类型 NEAR；语言类型 C；对 `EBX, ESI` 进行现场保护；形参设置 1 个 `dword` 类型 `GOODID`。

协议描述：主程序传递待计算利润率商品单元的首地址；函数进行利润率计算将计算值存放在商品单位对应位置，计算完成后，返回主程序流程中。形参 `GOODID` 存放待计算利润商品单元首地址以及计算完成后商品首地址。

(4) 利润排序子程序 `my_sort`；函数类型 NEAR；语言类型 `stdcall`；对 `EBX, ESI, EDI` 进行现场保护；形参设置 1 个 `dword` 类型 `ORIID`。

协议描述：主程序传递待商品存储单元结构的首地址；函数进行排序完成后，返回主程序流程中。形参 `ORIID` 存放商品单元结构首地址及排序后的商品结构首地址。

由于本程序暂无必要的公共符号，故协议中不含该内容。

另外，在进行利润率计算时，调用宏指令 `save_imul` 实现两个存储器寻址变量相乘的操作。宏定义 `save_imul` 中形参设置 `buf1, buf2` 与 `product`。`buf1` 和 `buf2` 存放存储器中指定的存储单元数据，将其相乘结果存放至 `product` 中。

3.3.2 流程图

图 3.1 是任务 3.1 的模块结构和子程序调用说明。

汇编语言程序设计实验报告

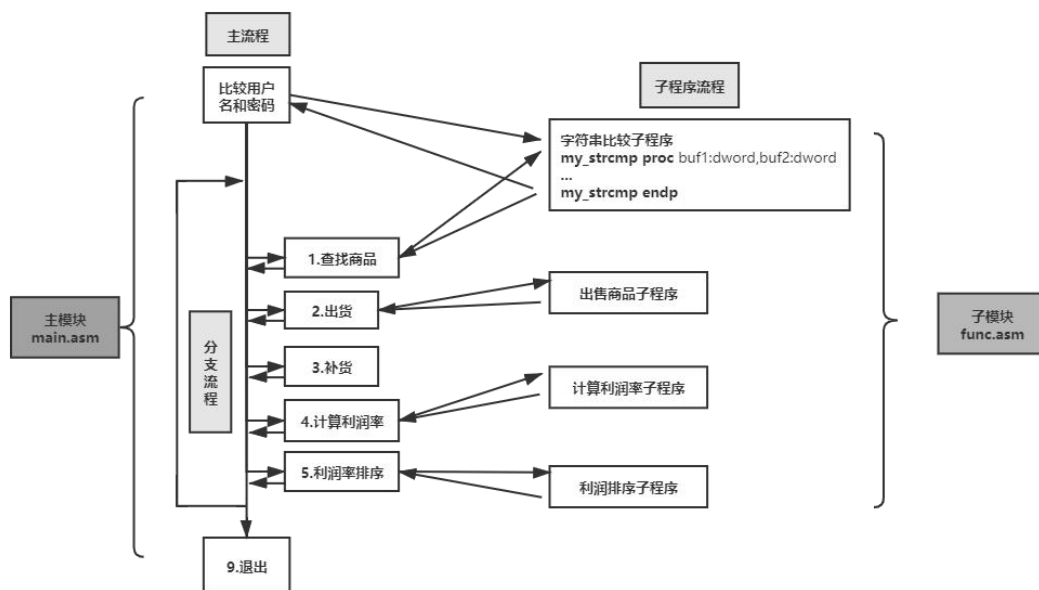


图 3.1 模块结构图以及子程序调用关系

3.3.3 源程序

```
;save_imul 存储器数据乘法宏指令
;形参设置 buf1 与 buf2 为存储器数据，product 存放相乘结果
save_imul macro buf1,buf2,product
    push ebx
    push edx
    movzx ebx,buf1
    movzx edx,buf2
    imul ebx,edx
    mov product,ebx
    pop edx
    pop ebx
Endm
```

```
;my_strcmp 字符串比较函数
;buf1 与 buf2 为待比较字符串存放的首地址
;局部变量 count_num 用于计数当前比较位数
;堆栈保护寄存器 esi, edi
my_strcmp proc,buf1:dword,buf2:dword
    local count_num:dword
    push esi
    push edi
    .....
    pop edi
    pop esi
    ret
my_strcmp endp

;sell_goods 出售商品函数
;TARID 为待出售商品的首地址,同时也存放处理后的首地址;SellData 为销售数量
;ESI 存放首地址
sell_goods proc uses ESI,TARID:dword,SellData:word
    .....
endp
```

汇编语言程序设计实验报告

```
ret
sell_goods endp

;count_profits 计算利润率函数
;GOODID 为待计算商品的首地址；同时也存放处理后的首地址
;ESI 存放首地址
count_profits proc uses ESI EBX,GOODID:dword
    .....
    ret
count_profits endp

;my_sort 排序函数
;ORIID 为待排序结构的首地址；同时也存放排序后的结构首地址
;eax 用于外部循环的计数；esi 用于直接内部循环的计数
my_sort proc stdcall uses ESI EDI EBX,ORIID:dword ;eax 为外层循环，esi 为内层循环
    mov EAX,6
Loop5:
    CMP EAX,24
    ja next5
    MOV EDI,ORIID
    ADD EDI,EAX
    SUB EAX,6
    MOV ESI,ORIID
    ADD ESI,EAX
    ADD eax,6
    MOV EDX,dword ptr [EDI]
    MOV CX,word ptr [EDI+4]
Loop6:
    CMP CX,word ptr [ESI+4]
    jge next6
    mov ebx,dword ptr [ESI]
    mov dword ptr [ESI+6],ebx
    mov bx,word ptr [ESI+4]
    mov word ptr [ESI+10],bx
    sub ESI,6
    CMP ESI,ORIID
    jb next6
    jmp Loop6
next6:
    ADD ESI,6
    mov dword ptr [ESI],EDX
    mov word ptr [ESI+4],CX
    ADD eax,6
    jmp Loop5
next5:
    ret
my_sort endp
```

3.3.4 实验步骤

1. 准备上机实验环境，使用 Visual Studio2019 进行程序编写、调试、运行。
2. 在子模块中构建上述设计子程序，然后进行逐一调试运行，使其运行结果正确。
3. 观察细节。
 - (1) 在反汇编窗口中逐步运行，查看子程序与主程序信息传递方式。
 - (2) 分别使用 call 指令与 invoke 指令，查看其反汇编对应的语句与运行方式。

汇编语言程序设计实验报告

(3) 观察 ret 指令的运行操作, 并在 ret 前改变栈顶元素, 查看 ret 运行结果。

(4) 观察局部变量在子程序中的存储方式, 以及使用方式。

4. 探究。

(1) 对 NEAR 类型子程序强制使用 FAR 调用, 观察结果。

(2) 观察公共符号与外部符号的使用方式。观察当公共符号的符号名不一致时产生的现象。

(3) 通过反汇编窗口, 观察宏调用与函数调用的差别。

3.3.5 实验记录与分析

1. 实验环境条件: Intel Core i5-8250U 1.6GHz, 8M 内存; WINDOWS 10 下 Visual Studio 2019。

2. 调试成功后, 运行时查看反汇编窗口中的子程序调用方式。如观察 my_strcmp 的调用方式, 其需传递 2 个参数。如果采用 invoke 方式调用, 如图 3.2 所示。

```
78: invoke my_strcmp, offset ANAME, offset BNAME ;比较用户名
0050837E 68 0E 82 57 00      push     offset BNAME (057820Eh)
00508383 68 D0 81 57 00      push     offset ANAME (05781D0h)
00508388 E8 38 95 FF FF       call     _my_strcmp (05018C5h)
0050838D 83 C4 08             add     esp, 8
```

图 3.2 invoke 调用方式下的反汇编语句

发现 invoke 指令执行 4 条语句分别为两个参数进栈传参以及 call 指令和调用结束后的栈顶参数回退。当将其转化为 call 指令时也必须加入最后的退栈指令 add esp,n, 避免多次调用栈溢出的情况。

3. 当执行 call 指令时, 将进行如图 3.3 所示相关操作。执行完成后栈顶数据变为主程序断点地址, 如图 3.4 所示, 横线为断点地址, 方框中的数据为两个入栈参数。

```
00FE8388 E8 38 95 FF FF      call     _my_strcmp (0FE18C5h)
_my_strcmp:
00FE18C5 E9 36 69 00 00      jmp     my_strcmp (0FE8200h)

23: my_strcmp proc, buf1:dword, buf2:dword
> 00FE8200 55                push     ebp      已用时间 <= 1ms
00FE8201 8B EC            mov     ebp, esp
-----
```

图 3.3 call 指令运行流程

```
8d 83 fe 00 d0 81 05 01 0e 82 05 01
```

图 3.4 call 指令运行后栈中数据

4. 结尾处 ret 指令在反汇编中显示两条指令 (如图 3.5 所示)。执行前堆栈部分数据如图 3.6 所示, 横线部分为局部变量, 中间数据为前 EBP 数值, 灰色区域则为主流程断点地址。

```
48: ret
00FE8234 C9                leave
00FE8235 C3                ret
```

图 3.5 ret 指令的反汇编语句

```
00 00 00 00 c4 fd 16 01 8d 83 fe 00
```

图 3.6 ret 执行前堆栈部分数据

发现, leave 语句与 ret 语句分别进行不同的操作。其中, leave 语句执行 ESP 返回至 EBP 指向地址的下一偏移地址以及返回 EBP 的操作 (如图 3.7 所示)。而 ret 语句则执行返回主流程, 即赋

汇编语言程序设计实验报告

值 EIP 操作（如图 3.8 所示）。

ESP = 0116FD74 EBP = 0116FDC4 8d 83 fe 00

图 3.7 leave 语句执行后的 EBP 与堆栈数据

} EIP = 00FE838D ESP = 0116FD74

图 3.8 ret 语句执行后的 EIP

无论是否有局部变量，在 ret 语句前改变栈顶数据，仍可正确运行。其原因是仅改变了局部变量相关数值或原 EBP 数值，并未改变断点数据 EIP 的数值。

5. 在定义局部变量时，ESP 会进行相应的改变（如图 3.9 所示），留出局部变量所占空间。而其对应地址则为[EBP-n]的数据，故在获取其地址时可以使用 offset 语句，亦可使用[EBP-n]的方式获取地址。

```
23: my_strcmp proc, buf1:dword, buf2:dword
004F8200 55                push     ebp
004F8201 8B EC            mov      ebp, esp
004F8203 83 C4 FC        add      esp, 0FFFFFFCh
```

图 3.9 在堆栈中为局部变量预留空间

6. 在 X86-32 位环境下，当对 NEAR 类型子程序强制使用 FAR 调用时，程序运行时出现报错，如图 3.10 所示。



图 3.10 强制使用 FAR 调用报错显示

分析可知，在该环境下使用 flat 内存模式，代码段从地址 0 开始，4G 处结束，这样一来，所有的子程序调用都处在同一个段中，所以都是 NEAR 调用。

7. 由于本次实验并无需求定义公共符号，故为探究公共符号的运用，定义公共符号 public Note; 外部符号声明为 extern Note:word。Note 在主模块中定义为字类型数据 3132H。在子模块中使用 mov ax,Note 指令，调用结果如图 3.11 所示。

mov ax,word ptr [Note (01B81C6h)] 0x001B81C6 32 31

图 3.11 外部符号调用结果

而当外部符号声明中，改变申明类型为 extern Note:byte，则会在 mov 指令处进行报错。原因为此时 Note 为 byte 类型数据，与 ax 类型不匹配，而与模块间类型是否一致无关。而当改变符号名称时，则会进行报错，原因为无法解析外部符号，与模块间符号名不一致有关。

8. 在计算利润率子程序中，调用带参数的宏指令，其宏定义与反汇编如图 3.12 所示。

```
save_imul macro buf1,buf2,product
push ebx
push edx
movzx ebx,buf1
movzx edx,buf2
imul ebx,edx
mov product,ebx
pop edx
pop ebx
endm

77: save_imul word ptr [ESI+12],word ptr [ESI+16],eax
00BE826A 53                push     ebx
00BE826B 52                push     edx
00BE826C 0F B7 5E 0C      movzx    ebx,word ptr [esi+0Ch]
00BE8270 0F B7 56 10      movzx    edx,word ptr [esi+10h]
00BE8274 0F AF DA        imul     ebx,edx
00BE8277 8B C3            mov      eax,ebx
00BE8279 5A                pop      edx
00BE827A 5B                pop      ebx
```

图 3.12 带参宏定义（左）与宏调用反汇编（右）

汇编语言程序设计实验报告

分析发现,在进行带参宏调用时,会将定义语句中的参数替换为调用参数,且反汇编时自动进行语句扩展。而与子程序调用不同,子程序调用会进行模块间或流程间的跳转,并且传参方式明显也不相同。

3.4 任务 3.2 的实验过程

3.4.1 实验方法说明

1. 该实验中将进行 C 语言与汇编语言的混合使用。故对该系统流程的构建划分为两部分: C 语言部分与汇编语言部分。

2. 构建 C 语言部分: 构造用户登录功能与主菜单显示功能, 以及添加新商品功能。

3. 构建汇编语言部分, 以及混合部分的联系设置。

4. 调试、运行。使其产生正确的输出结果。

5. 观察细节。

(1) 在 C 语言模块中, 通过不同方式进行赋值, 查看其反汇编语句执行结果;

(2) 查看 C 语言模块中的函数调用的反汇编语句, 逐行调试观察运行过程;

(3) 观察结构数组在 C 语言模块与汇编语言模块的联系。

(4) 观察其他可能出现的现象。

3.4.2 实验记录与分析

1. 实验环境条件: Intel Core i5-8250U 1.6GHz, 8M 内存; WINDOWS 10 下 Visual Studio 2019。

2. 在 C 语言模块中, 定义全局变量 `int x,y`。采取两种赋值方式, 一种为 `y=0`, 其反汇编语句为 `MOV dword ptr[_y],0`。另一种为 `*(&x+1)=0`, 则会自动进行扩展, 在 `x` 的地址值上加 4 为目标地址。C 语言在存储全局变量时为紧凑的存储方式, 故可以采取第 2 种赋值形式。

3. 在 C 语言模块中, 进行汇编语言模块的函数调用, 如调用汇编语言编写的 `my_strcmp` 函数, 查看反汇编窗口, 如图 3.13 所示。

```
22:      flat = my_strcmp(ANAME, BNAME);
0361A0E 68 1E A2 36 00      push     offset BNAME (036A21Eh)
0361A13 8D 45 E4             lea      eax, [ANAME]
0361A16 50                  push     eax
0361A17 E8 7F F6 FF FF      call     _my_strcmp (036109Bh)
0361A1C 83 C4 08             add      esp, 8
0361A1F 89 45 BC             mov      dword ptr [flat], eax
```

图 3.13 C 语言模块中调用汇编语言模块函数

分析发现,对比任务 3.1 内的汇编语言函数 `invoke` 调用方式十分相似。其中,在 `ANAME` 地址进栈操作时,进行了两次指令, `lea eax,[ANMAE]`与 `push eax`,分析可知, `ANAME` 为定义的局部变量,只能通过 `lea` 指令获取局部变量地址。

4. 观察结构数组的联系,比较汇编语言与 C 语言构建结构的联系。两类模块下的结构定义如图 3.14 所示。

汇编语言程序设计实验报告

<pre>GOODS struct GOODSNAME DB 10 DUP(0) BUYPRICE DW 0 SELLPRICE DW 0 BUYNUM DW 0 SELLNUM DW 0 RATE DW 0 GOODS ENDS</pre>	<pre>struct GOODS { unsigned char GOODSNAME[10]; unsigned short BUYPRICE; unsigned short SELLPRICE; unsigned short BUYNUM; unsigned short SELLNUM; unsigned short RATE; };</pre>
---	--

图 3.14 汇编模块下（左）与 C 语言模块下（右）的结构定义

在进行 C 语言模块与汇编语言模块的结构关联时，仅传输结构首元素地址。在汇编语言模块中，访问结构各元素是根据字节长度进行对应访问，指令为结构变量地址值加上对应元素在结构中的相对字节位置值。在 C 语言模块中，结构数组定义为 GA1[i]，观察反汇编，访问结构数组元素由 GA 地址值与 i 值决定。如访问 GOODSNAME 元素时，进行如图 3.15 的指令操作；如果访问 BUYPRICE 元素时，进行如图 3.16 指令操作。

```
imul    eax, dword ptr [i], 14h
add     eax, offset GA1 (036A238h)
```

图 3.15 访问 GA1[i]的 GOODSNAME 元素的反汇编语句

```
imul    eax, dword ptr [i], 14h
add     eax, 36A242h
```

图 3.16 访问 GA1[i]的 BUYPRICE 元素的反汇编语句

分析可知，对于 C 语言访问 GA1[i]元素时，会先进行 i 乘法查询，其乘法数组为结构长度，提供数组相对位置。再进行访问元素地址的加法查询。将两者结合即可得到对应元素的地址。

3.5 小结

3.5.1 主要收获

此次实验完成的两项任务整体围绕模块编程展开，任务 3.1 实现汇编多模块编程，任务 3.2 实现汇编语言模块与 C 语言模块混合编程。通过对多模块的编程实验，主要掌握汇编语言多模块与混合编程的设计思想，以及模块联系、调用的具体实现。

任务 3.1 中，主要掌握了多模块编程思想，其次则是掌握了对函数调用和宏指令调用的设计与使用。将主流程与子程序设计分别放置在两个不同的模块中，可以使模块信息传递具有条理性，不至于多模块处理混乱。在多模块编程中，主要思想则是主模块简单化，子模块为主模块提供辅助函数支撑。另外，在进行模块间联立时，函数的调用规范也不容小觑。在主流程中调用函数可使用 call 指令与 invoke 指令，且使用堆栈法传参与保护现场可以极大程度减少记忆与编程压力。在对 invoke 指令进行反汇编时，发现其为 call 指令的延伸，即包含 call 指令以及进栈，退栈操作。所以，在单独进行 call 指令操作时，要注意退栈处理以避免多次调用栈溢出。

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

汇编语言程序设计实验报告

值，在不同的子模块中的使用则取决于外部符号说明时的类型，但不允许变量名称存在差异。

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

任务 3.2 中，则是熟悉汇编语言与 C 语言的混合编程模式。实验发现，在不同语言模块进行多模块联立时，外部符号的传递仅仅是传递变量地址值，故其类型根据外部符号说明的类型为准。在进行结构数组联立时，则只需保证结构定义的字节段相同即可，在反汇编时会进行对应地址处理，从而访问所需元素。而 C 语言模块调用汇编模块函数时则与汇编语言调用子模块函数无异。

最后，掌握到对相邻元素进行赋值时，可以通过其中一元素对另一元素进行存储值改变，从而间接改变相邻元素值。关键则是弄清元素之间的相对位置关系。

3.5.2 主要看法

本次实验主要以模块思想为主，分别进行了汇编多模块编程、汇编 C 语言混合模块编程的操作。设计十分合理，且对两个模式中出现的函数和外部符号联立进行了考察与对比。加深了对模块联立的不同操作以及相似思想。但对于一些探究则会显得无意义，如对 NEAR 和 FAR 的强制转换函数调用探究，在目前 X86-32 位 flat 模式下显得并无实际意义，故可以考虑进行改进。

4 中断与反跟踪

4.1 实验目的与要求

- (1) 通过观察与验证，理解中断矢量表的概念；
- (2) 熟悉 I/O 访问，BIOS 功能调用方法；
- (3) 掌握实方式下中断处理程序的编制与调试方法；
- (4) 进一步熟悉内存的一些基本操纵技术；
- (5) 熟悉跟踪与反跟踪的技术，熟悉动态与静态反汇编工具，深刻理解汇编语言的特殊能力；
- (6) 完成指定功能的程序设计与调试，提升对计算机系统的理解与分析能力。

4.2 实验内容

任务 4.1：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，并能避免重复安装。

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；能在 TD 下调试中断处理程序(给出调试方法和关键的调试记录)。

汇编语言程序设计实验报告

任务 4.2: 数据加密与反跟踪

在任务 3.1 完成的程序的基础上,老板的密码采用密文的方式存放在数据段中,各种商品的进货价也以密文方式存放在数据段中。加密方法自选(但不应选择复杂的加密算法)。当进货价被加密后,原来对该数据进行处理的功能在执行处理操作前就需要解密进货价。

可以采用计时、动态修改执行代码、间接寻址、代码中穿插数据定义或无关代码、堆栈检查等反跟踪方法中的几种方法组合起来进行反跟踪(要求:采用包括动态修改执行代码在内的不少于两种反跟踪方法,重点是深入理解和运用好所选择的反跟踪方法)。

为简化录入和处理的工作量,只需要定义三种商品的信息即可。

任务 4.3: 跟踪与数据解密

解密同组同学的加密程序,获取各个商品的进货价。

注意:两人一组(在实验记录中说明同组是谁),每人实现一套自己选择的加密与反跟踪方法,把执行程序交给对方解密(解密时间超过半小时的,说明反跟踪方法基本有效)。如何设计反跟踪程序以及如何跟踪破解,是本次实验报告中重点需要突出的内容。

4.3 任务 4.1 实验过程

4.3.1 设计思想及存储单元分配

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。即,若为首次安装,则旧中断地址处前 2 个字节的标记不为‘#’,则进行后续安装环节;若为二次及以上安装,则始终为首次安装地址,地址处且前 2 个字节标记‘#’驻留不会消失,从而判断已经首次安装,跳过安装环节。在主程序中加入显示信息 T,若进入安装环节,则显示 T。

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

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

中断处理程序设计包含 3 个环节:执行原中断程序,获取时间信息,显示。

汇编语言程序设计实验报告

(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。

4.3.2 流程图

图 4.1 是任务 4.1 的主流程处理程序流程图。

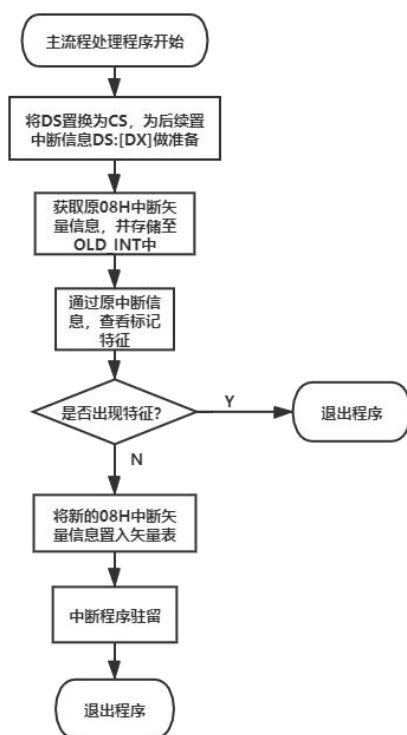


图 4.1 主流程处理程序流程图

4.3.3 源程序

```
.386
CODE SEGMENT USE16
    ASSUME CS:CODE,DS:CODE,SS:STACK
;新的 INT 08H 使用变量
...
flag DW '#'
```

汇编语言程序设计实验报告

```
;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
```

4.3.4 实验步骤

1. 准备上机实验环境，使用 Visual Studio Code 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。

4.3.5 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 DOSBox0.74-3 TD.EXE。
2. 运行程序，首先进行安装中断处理程序，观察结果；运行其他程序，观察结果；再次运行安装程序，观察结果。观察结果如图 4.2 所示。

汇编语言程序设计实验报告

```
C:\>test4
T
C:\>hello
HELLO WORLD!
C:\>test4
```

图 4.2 程序运行结果截图

发现，程序运行成功，当第二次及多次运行程序时，未输出安装检验显示 T，表明设置标记特征能够达到避免多次重复安装的目的。

3. 观察中断矢量表，在内存单元 0000:0020H 处，储存此次实验变化矢量信息。首次运行安装程序前的该地址内存如图 4.3(a)所示，运行后的该地址内存如图 4.3(b)所示。

```
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 4.3(a) 首次运行前的中断矢量表

```
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 4.3(b) 首次运行后的中断矢量表

4. 查询第 2 次运行后的中断矢量表查询 08 中断号信息，访问中断处理程序，如图 4.4 所示。

```
cs:0011 9C          pushf
cs:0012 2EFF1E0B00  call cs:far [000B1
cs:0017 2EFE0E0000  dec  cs:byte ptr I6
cs:001C 0F840100    je   0021
cs:0020 CF          iret
cs:0021 2EC606000012 mov  cs:byte ptr I6
cs:0027 FB          sti
cs:0028 60          pusha
cs:0029 4F          push
```

图 4.4 查询中断矢量表访问中断处理程序

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

4.4 任务 4.2 实验过程

4.4.1 实验方法说明

1. 该任务分为两个步骤，即加密与反跟踪。但需将主要精力放在反跟踪上。
2. 首先进行加密操作。采取异或、添加多余数据、简单的运算的加密手段，将用户密码和各商品进货量以密文形式存储。
3. 进行反跟踪。采用 4 种反跟踪方式，即动态修改执行代码、计时、间接转移以及插入数据定义和无关代码的方式进行反跟踪操作。有效抵制静态跟踪与动态跟踪。

4.4.2 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 Visual Studio 2019。
2. 在加密步骤时，对密码采取异或和添加多余数据的手段进行加密。同时在密码比对时，对外部输入的待验密码进行同等加密，通过密文进行比对。而对进货价则采取简单运算的加密手段，对进货价进行加 10H 的操作，进而保存至数据段中。加密手段如图 4.5 所示。

汇编语言程序设计实验报告

```
BPASS DB 'U' xor 'A','2' xor 'B','0' xor 'C','1914984',12H,32H,0
GA1 GOODS <'PEN',15+10H,20,70,25,>
GA2 GOODS <'PENCIL',2+10H,3,100,50,>
GA3 GOODS <'BOOK',30+10H,40,25,5,>
```

图 4.5 对密码和进货价进行加密

3. 首先进行动态修改执行代码方式反跟踪。在 Windows 系统下,使用 API 函数 VirtualProtect 来改变调用程序的一段内在区域的保护属性。对调用 my_strcmp 函数指令进行机器码替换,替换的机器码如图 4.6 所示。

```
machine_code DB 0E8H,0A1H,94H,0FFH,0FFH ;call my_strcmp的机器码
```

图 4.6 替换机器码数据段

其中 0E8H 为 call 指令的机器码,而 0xFFFF94A1H 为该指令的相对位移量,该值在程序中是固定的,故可使用该机器码进行动态修改执行代码反跟踪。

4. 通过计时的方式来抵制动态调试跟踪。该方法的主要思想为人工单步调试一段代码的时间远远大于连续执行该程序段所用时间,故可使用计时程序,来检验执行一段代码的时长,设置时长分界 55ms。计时反跟踪如图 4.7 所示。

```
INVOKE winTimer,0 ;计时开始
mov ecx,0
mov esi,offset APASS
mov edi,offset BPASS
xor APASS,'A'
xor APASS[1],'B'
xor APASS[2],'C'
mov P5,str_loop1
mov str_end,str_exit
str_loop1:
mov al,[esi]
cmp al,[edi]
INVOKE winTimer,1 ;计时结束,时间差为eax
cmp eax,55 ;当计时大于55ms时则转移到特定指令区
```

图 4.7 计时反跟踪方法

当执行该段程序超过 55ms 时,将偏移正确进程,进入特定错误指令程序段,干扰跟踪者的动态调试。使用实验 4.2 中的 winTimer 函数,计算一段时间的时间差,从而进行反跟踪。

5. 间接转移/调用的方式抵制静态反汇编跟踪。该方法包括 jmp 和 call 指令的间接转移和调用,如图 4.8(a)、图 4.8(b)所示。

```
mov eax,1 ;间接转移方式
mov ebx,offset P5
mov edx,1
jmp dword ptr[ebx+edx*4]
```

图 4.8(a) 间接转移方式

```
lea eax,P1 ;间接调用方式
push ESI
call dword ptr[eax]
ADD ESP,4
```

图 4.8(b) 间接调用方法

在进行 jmp 指令间接转移时,P5 与目标地址存储地址相邻,故使用寄存器间接寻址的方式降低代码可读性。而在进行函数间接调用时,P1 存放子程序入口地址,采用寄存器间接寻址的方式,进行 call 指令转移。通过间接转移/调用的方式增大代码阅读难度,干扰跟踪者静态跟踪。

6. 插入数据定义和无关代码干扰视线。该方式主要与间接转移/调用方式同时使用,从而进一步增大阅读难度,干扰视线。添加干扰代码如图 4.9 所示。

汇编语言程序设计实验报告

```
pop ebx
call other_road ;在处理堆栈中途进行其他指令执行, 干扰视线
msg DB 'Hello,world',0 ;添加多余数据定义, 干扰视线
pop edi
pop esi
```

图 4.9 插入干扰代码

在进行堆栈处理时, 插入干扰函数调用 other_road 与无用数据段, 组合成干扰堆栈跟踪方式。other_road 函数为通过堆栈读取多余数据, 直至数据末尾 0。再此中断退栈操作, 进行其他堆栈特殊化使用, 极大程度干扰了跟踪者的动态调试, 加大了阅读难度, 达到一举多得的目的。

4.5 任务 4.3 实验过程

4.5.1 实验方法说明

1. 该任务分为两个步骤, 即跟踪与解密。
2. 利用静态反汇编工具 W32Dasm, 将执行程序反汇编。
3. 进行静态跟踪, 观察执行文件中的数据信息, 尝试发现关键破绽, 尝试查找反跟踪程序。
4. 进行动态跟踪, 输入错误信息, 逐行运行, 观察运行过程发生的变化。

4.5.2 实验记录与分析

1. 实验环境条件: Intel Core i5-8250U 1.6GHz, 8M 内存; WINDOWS 10 下 W32Dasm。实验小组成员: 陆云龙、王家顺。
2. 利用静态反汇编工具 W32Dasm, 将执行程序反汇编, 首先进行密码破解。首先观察数据段, 读取姓名信息。如图 4.10 所示。观察到使用计时反跟踪方式, 如图 4.11 所示。

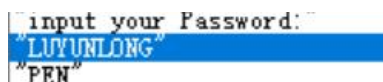


图 4.10 使用 W32Dasm 工具破解字符串数据

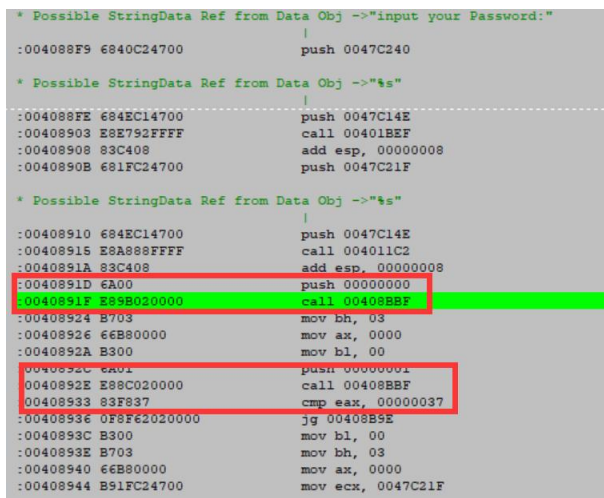


图 4.11 静态观察获取计时反跟踪程序

汇编语言程序设计实验报告

分析发现，比较 eax 与 37 的指令为检验时间间隔，当时间间隔大于 37ms 时，跳转至偏离程序段 (00408B9Eh) 中，进行错误显示，干扰我的观察。

通过静态观察，发现动态修改执行代码，如图 4.12 所示。可在动态调试中避免被其干扰。紧接着，通过该动态执行修改代码，推测其紧挨着密码的输入操作。发现，两步 push 指令，推测其为密码与输入密码的比较函数参数传递，故尝试通过该指令查找加密密码数据段。成功获取加密信息。如图 4.13 所示。

```
00408965 B805000000 mov eax, 00000005
0040896A B840000000 mov ebx, 00000040
0040896F 8D099994000 lea ecx, dword ptr [00408999]
00408975 687C24700 push 0047C27C
0040897A 53 push ebx
0040897E 50 push eax
0040897C 51 push ecx

* Reference To: KERNEL32.VirtualProtect, Ord:05D0h
|
0040897D E839FFFFFF Call 004087BB
00408982 B805000000 mov ecx, 00000005
00408987 BF9994000 mov edi, 00408999

* Possible StringData Ref from Data Obj -> "???"
|
0040898C BE77C24700 mov esi, 0047C277
00408991 8A06 mov al, byte ptr [esi]
00408993 8B07 mov byte ptr [edi], al
00408995 4E inc esi
00408996 47 inc edi
00408997 E2F8 loop 00408991
00408999 0000000000 BYTE 5 DUP(0)
```

图 4.12 静态观察获取动态修改执行程序

```
0040895B 681FC24700 push 0047C21F
* Possible StringData Ref from Data Obj -> "糖神 菜"
|
00408960 68A04700 push 0047C0AA
```

```
0047C218 00 00 00 00 00 00 00 00
0047C220 00 00 00 00 00 00 00 00
0047C228 00 00 00 00 00 00 00 69
0047C0A8 47 00 F9 90 8A 8D A5 8D
0047C0B0 96 A5 A2 9F 00 00 00 00
```

图 4.13 静态观察获取加密信息

通过合理推测，成功查询到加密密码的可能十位信息。接着需获取加密手段，利用该动态修改执行程序中的 esi 赋值，查找机器码数据为 E8 3A AE FF FF，E8 为 call 指令的机器码，相对位置为 0xFFFFAE3A。如图 4.14 所示。

```
0040898C BE77C24700 mov esi, 0047C277
00408991 8A06 mov al, byte ptr [esi]
00408993 8B07 mov byte ptr [edi], al
```

```
0047C270 73 21 0A 0D 0A 0D 0C E8
0047C278 3A AE FF FF 00 00 00 00
```

图 4.14 密码比对函数入口地址相对位置

通过该相对位置，计算出 call 指令跳转地址为 0x4037D3。不过当进行静态跟踪后，遇到困难，无法人工解析其后的代码，故转变动态调试。

3. 在动态调试中，观察到间接调用来干扰视线，如图 4.15 所示。

```
0038A89E 03048D18E43B00 add eax, dword ptr [4*ecx+003BE418]
0038A8A2 50 push eax

* Reference To: KERNEL32.LeaveCriticalSection, Ord:03C1h
|
0038A8A3 FF155CF03B00 call dword ptr [003BF05C]
```

图 4.15 间接调用函数干扰视野

由于此工具对动态调试时的 scanf 与 printf 的操作跳转过多，增大了单步调试压力，故在短时间内无法查询到代码预期动态处理过程。由此可以思考，增加无关调用函数，可以极大程度干扰动态调用的进程。除此之外，还设置了计时的方法来阻止动态观察，如图 4.16 所示，增加了动态调试难度。

```
* Reference To: WINMM.timeGetTime, Ord:0094h
|
0040BB09 FF35ACF14700 jmp dword ptr [0047F1AC]
0040BB0F CC int 03
```

图 4.16 计时方式阻碍动态调试

4.6 小结

4.6.1 主要收获

此次实验完成了两项工作：中断处理程序的置换与驻留，加密反跟踪的设置与解密。由于是首次接触此类程序编写，进一步加深了对汇编程序的认知与理解。尤其对于中断处理和反跟踪处理，有了更深刻的认识，并且也逐渐掌握了对中断和反跟踪程序的应用。

任务 4.1 则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，在进行后续操作。为避免破坏寄存器数值，则还需进行现场保护。

任务 4.2 与任务 4.3 分别进行反跟踪与跟踪操作，简单了解了信息的加密处理，掌握了通过反跟踪程序来抵制静态跟踪，以及使用反汇编工具对加密程序的静态跟踪。任务 4.2 中，简单进行了加密操作，了解了对加密信息的细节操作，如对输入密码进行加密再对比，而无需在代码段中进行解密操作。而对于反跟踪程序的执行，包含静态抵制和动态抵制两个方面。此次实验中抵制静态反汇编采用了 3 种方法：间接转移/调用、动态修改执行代码以及插入无关代码。其主要思想是降低代码可读性，增加代码复杂度。并且通过动态修改代码，了解了 API 函数 VirtualProtect 的使用，可通过机器码代换汇编语句进行程序编写，扩展了知识面和编程思想。

而进行跟踪操作时，则了解了静态跟踪与动态调试结合的方式。在静态反汇编过程中，借助反汇编工具 W32Dasm 的数据段查询功能，可直接转移到密码处理相关代码段中。通过观察与推测，跟踪出了反跟踪手段：间接调用干扰，动态修改代码段以及计时反跟踪。对于动态修改代码，可以根据代码的执行过程，查询出替换机器码，从而通过偏移值跳转至对应位置。而计时反跟踪则可通过自动运行进行跳过，通过工具自带动态调试功能，能够跳过计时程序。

通过此次实验，让我充分了解到汇编语言的强大功能，不仅体现在其对系统中断功能的修改，还体现在加密与反跟踪的多样性与细节性。相比与 C 语言而言，汇编语言在微观上对程序进行修改更为方便，一句 C 语言的修改往往对应多条汇编语句的改变，从而会露出许多破绽。而通过汇编语言进行加密与反跟踪，能够通过一系列汇编语句的穿插，来加大程序的安全性，对防破解作用巨大。

汇编语言程序设计实验报告

4.6.2 主要看法

本次实验进一步阐述了汇编语言的强大功能，其对修改系统内部特点功能，以及处理程序加密与反跟踪做出了巨大贡献。在更低程度上，通过此次实验，加深了对系统指令细节操作上的认知。展示了其与其他高级语言的区别，可以在更加细微程度上处理程序。在加密与反跟踪阶段，更细微的操作更能加强程序的安全性。

5 16/32/64 位编程比较

5.1 实验目的与要求

- (1) 了解 16/32/64 位环境下程序设计的不同特点及配套的开发工具；
- (2) 通过完成指定的程序设计，观察并理解汇编语言在不同环境下的基本特点。

5.2 实验内容

任务 5.1 编写一个基于窗口的 WIN32 程序，实现**网店商品信息后台管理系统**的部分功能。也即：以任务 3.1 的程序为基础，将其部分功能移植过来，具体要求如下描述。

编写一个基于窗口的 WIN32 程序的菜单框架，具有以下的下拉菜单项：

File	Action	Help
Exit	Compute Rate	About
	List Sort	

点菜单 File 下的 Exit 选项时结束程序；点菜单 Help 下的选项 About，弹出一个消息框，显示本人信息。点菜单 Action 下的选项 Compute Rate、List Sort 将分别实现计算利润率或在窗口中显示按利润率排序后的所有商品各项信息的功能。

任务 5.2 在 VS2019 下调试一个 x64 程序，观察与 32 位程序的不同之处。参考程序见附件。

任务 5.3 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。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。参考阅读材料见附件（如：ARM 汇编技术简介.pdf，ARM 基础实验手册.pdf 等）。

5.3 任务 5.1 实验过程

5.3.1 实验方法说明

1. 编写基于窗口的 WIN32 程序时，先搭出主程序，窗口主程序，窗口消息处理程序，用户程序的大致框架。

汇编语言程序设计实验报告

2. 补充窗口消息处理程序与用户处理程序，使其具有要求功能。
3. 定义 5 种商品，对窗口菜单程序进行检验。
4. 比较基于窗口的应用程序的调试方式与以往程序调试的不同。
5. 观察点击菜单项时产生的信息，并比较控制台和窗口风格下信息输出方式的差别。

5.3.2 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 Visual Studio 2019。
2. 对菜单进行定义，要求中包含 4 种操作信息，故定义四种资源名称与子消息号，如图 5.1 所示。其需通过在头文件 menuID.inc 中定义为汇编语言可识别的符号常量后，方可使用，如图 5.2 所示。两者定义一一对应。

```
#define IDM_FILE_EXIT 10001
#define IDM_ACTION_COMPUTE 10101
#define IDM_ACTION_LIST 10102
#define IDM_HELP_ABOUT 10201
```

图 5.1 资源名称与子消息号的定义

```
IDM_FILE_EXIT equ 10001
IDM_ACTION_COMPUTE equ 10101
IDM_ACTION_LIST equ 10102
IDM_HELP_ABOUT equ 10201
```

图 5.2 头文件中定义汇编语言中的符号常量

3. 在补充用户处理程序时，除补充实验 3.1 的部分程序功能外，由于 TextOut 函数的输出的限制，在输出窗口界面时，数字类型数据需转化为字符类型，如数字 1 需转化为字符数据 31H，且还需把握输出的字符串位数。在此次数据中包含个位，十位，百位的数据，以及正负数和 0 的数据，如若单独对其进行转化会十分困难。故采用固定位数输出的方法，在无位处填写空格字符(ASCII 码为 32)即可，字符数据暂存空间如图 5.3 所示。同时通过数据转字符数的子程序来实现转换与存储功能。

```
goods_name db 32,32,32,32,32,32,0
goods_buyprice db 32,32,32,32,0
goods_sellprice db 32,32,32,32,0
goods_buynum db 32,32,32,32,0
goods_selinum db 32,32,32,32,0
goods_rate db 32,32,32,32,32,0
```

图 5.3 用于数字转字符的存储空间

4. 定义 5 类商品种类，对窗口菜单程序进行检验。首先查看窗口显示界面，分别测试每个操作菜单是否达到要求。点击菜单栏下 Help 的 About 子菜单选项，输出窗口信息如图 5.4 所示，以消息框的形式显示信息。先后点击菜单栏下 Action 下的 Compute Rate 和 List Sort 子菜单选项，输出窗口消息如图 5.5 所示，在主窗口中显示消息。

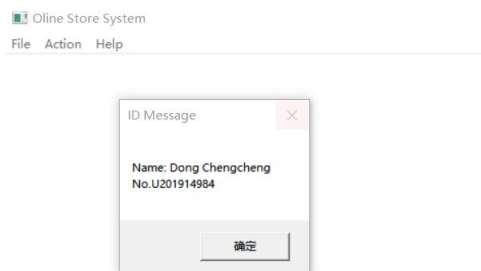


图 5.4 在消息框中显示信息

Online Store System					
File Action Help					
GoodsName	BuyPrice	SellPrice	BuyNum	SellNum	Rate
WATER	2	4	150	140	86 %
RULER	3	4	200	150	0 %
PENCIL	2	3	100	50	-25 %
PEN	15	20	70	25	-52 %
BOOK	30	40	25	5	-73 %

图 5.5 在主窗口中显示信息

汇编语言程序设计实验报告

5. 通过对基于窗口的应用程序的调试发现，同样可以通过设置断点的方式进行调试。在对用户程序进行调试时，可以通过先在需要调试的程序段设置断点，再运行程序，且需要点击对应菜单选项使其接收消息，跳转到处理对应消息的程序中，进而运行到断点处，即可进行调试。这与控制台下的程序调试有相似之处，控制台下的消息指令则为输入消息，而窗口风格下的调试为点击或者输入信息。

6. 在点击对应菜单时，会进行消息处理环节，识别消息为命令信息，将消息编号 `uMsg` 置为 `WM_COMMAND` 符号常量，再将子消息编号 `wParam` 置为对应符号常量，如 `IDM_FILE_EXIT`。通过窗口消息处理程序，跳转到对应分支处。进而完成消息的处理。

7. 比较控制台和窗口风格下的输出方式，可以发现明显差别。在控制台下的消息输出时，通过一些输出消息的函数，如输出函数 `printf`，在控制台界面输出信息。而在窗口风格下的输出，则通常在主窗口中输出信息，如运用 API 函数中的 `TextOut` 函数，并且该类函数只可输出字符类型的数据。故在进行相关窗口风格输出数字类型消息时，需要进行数串转化。

5.4 任务 5.2 实验过程

5.4.1 实验方法说明

1. 通过 VS2019 调试 x64 参考程序。
2. 观察与以往 32 位程序的不同之处。
3. 并对不同之处进行记录与分析。

5.4.2 实验记录与分析

1. 实验环境条件：Intel Core i5-8250U 1.6GHz，8M 内存；WINDOWS 10 下 Visual Studio 2019。
2. 观察整体，x64 程序很多高级用法不再支持。没有了处理器选择伪指令，如 .686P；没有了存储器模型说明伪指令，如 .model flat, c；没有了 .stack 堆栈段定义。如图 5.6a 与图 5.6b，对 32 位程序与 64 位程序的整体结构进行了比较。

```
.386
.model flat, stdcall
ExitProcess PROTO STDCALL :DWORD
includelib kernel32.lib

.DATA
lpFmt db "%s", 0ah, 0dh, 0

.STACK 200
.CODE
main proc c
```

图 5.6a 32 位程序的整体结构

```
.data
MessageBoxA proto :DWORD, :DWORD
extrn ExitProcess : proc ;编译器不再检查参数
lpContent db '你好! 3+6='
sum db 0, 0
lpTitle db 'My first x86-64 Application', 0

.code
mainCRTStartup proc ;这是控制台情况下默认的执行
;start proc ;需要在项目“属性-链接-
mov ax, 3
```

图 5.6b 64 位程序的整体结构

3. 观察细节，可发现以下细节差别。

(1) 首先在函数声明部分，32 位程序中只能通过 `proto` 进行函数的声明，而在 64 位程序中，编译器不再检查参数是否合规，可以使用除 `proto` 外的声明语句如 `extern` 来声明外部函数。

汇编语言程序设计实验报告

(2) 在 64 位程序中，控制台的默认执行入口点为 `mainCRTStartup`，而 32 位程序的控制台默认执行入口点为 `main`，且可以使用 `end+[开始地址]` 的用法进行程序入口的设置。64 位程序舍弃了 `end+[入口地址]` 的用法，要在 VS2019 中的项目属性中设置程序的入口点。

(3) 舍弃 `invoke` 的伪指令使用，参数传递需要在语句中体现。

(4) 64 位程序将 32 位寄存器扩展为 64 位，如 `ecx` 扩展为 `rcx`，且添加了更多的通用寄存器，如 `r8`, `r9`。且在进行对 C 语言函数及 API 函数时，需要遵循寄存器调用规则，只有一个参数时，只使用 `rcx` 存放参数，如图 5.7 所示；当有 4 个参数需要传递时，则需按照 `rcx`、`rdx`、`r8`、`r9` 的顺序进行参数存放，如图 5.8 所示，且需注意在入栈时是按照从右向左的顺序进行入栈。

```
sub    rsp, 18h
mov    rcx, 0
call   ExitProcess
```

图 5.7 调用单参数 `ExitProcess` 函数

```
sub    rsp, 28h
xor     r9d, r9d
lea     r8, lpTitle
lea     rdx, lpContent
xor     rcx, rcx
call    MessageBoxA
add     rsp, 28h
```

图 5.8 调用 4 个参数的 `MessaBoxA` 函数

5.5 任务 5.3 实验过程

5.5.1 实验方法说明

1. 查阅华为鲲鹏服务器采用的 CPU (ARMv8 系列) 的汇编语言编程资料，体会与 80x86 体系的异同。
2. 在 CPU 内寄存器、段定义方法、指令语句及格式的特点、子程序调用的参数传递和返回方式、与 C 语言混合编程、开发环节等方面进行比较。
3. 重点观察并记录段定义方法、子程序参数传递与返回的方法、开发环境的差异。

5.5.2 实验记录与分析

1. 对于段定义的方式，在 80x86 的 32 位环境中，一个汇编程序能够包含数据段 `.data`，代码段 `.code` 和堆栈段 `.stack` 的定义。而 ARMv8 系列的段定义方法为以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：数据段、代码段，且一个汇编程序至少有一个代码段。具体定义如图 5.9 所示。

汇编语言程序设计实验报告

1) 代码段 上面的例子为代码段。 AREA 定义一个段，并说明所定义段的相关属性 CODE 用以指明为代码段 ENTRY 标识程序的入口点 END 为程序结束。	2) 数据段 AREA DATAAREA, DATA, BIINIT, ALLGN=2 DISPBUF SPACE 200 RCVBUF SPACE 200 DATA 用以指明为数据段， SPACE 分配 200 字节的存储单元并初始化为 0
--	--

图 5.9 ARMv8 的具体段定义方式

2. 对于子程序参数传递与返回方法，在 80x86 的 32 位环境中，参数传递可通过寄存器或者存储单元传递，更多使用堆栈法传参，且子程序返回时则是通过修改 esp，ebp 以及 eip 的方式进行返回。而 ARMv8 的子程序调用与返回则显得十分简朴，可以看出其通过实参的方式进行参数传递，且仅通过单模块下的跳转指令 BL 实现子程序的调用，且无返回指令的使用，而是直接修改 PC 实现返回指令。如图 5.10 所示。

```
使用 BL 指令进行调用，该指令会把返回的 PC 值保存在 LR
AREA Example, CODE, READONLY      @声明代码段 Example
ENTRY                             @程序入口
Start
MOV R0, #0                        @设置实参, 将传递给子程序的实参存放在 r0 和 r1 内
MOV R1, #10
BL ADD_SUM                        @调用子程序 ADD_SUM
B OVER                            @跳转到 OVER 标号处, 进入结尾
ADD_SUM
ADD R0, R0, R1                    @实现两数相加
MOV PC, LR                       @子程序返回, R0 内为返回的结果
OVER
END
```

图 5.10 ARMv8 的子程序参数传递与返回方式

3. 对于两者的开发环境。对于 80x86 的使用则包含 DOS 实模式环境以及 Win32 环境，分别可以在 DOSBox 和 VS2019 的开发环境中编写汇编程序。而 ARMv8 处理器内核使用 ARM 结构，该结构包含 32 位的 ARM 指令集和 16 位 Thumb 指令集，因此 ARM 有两种操作状态。在进行开发时，配置的软硬件环境如图 5.11 所示。

1.4 实验环境说明

- 华为鲲鹏云主机、openEuler20.03 操作系统；
- 安装 gcc7.3+ 版本；

图 5.11 ARMv8 的开发环境

5.6 小结

5.6.1 主要收获

本次实验完成了对基于窗口的 Win32 程序的实现、对 64 位与 32 位程序的对比以及对华为鲲鹏服务器使用的 ARMv8 系列 CPU 下的汇编语言编程的了解。概括下来则主要包含两个实验环节，即基于窗口的 Win32 程序的实现和不同环境的对比。

汇编语言程序设计实验报告

首先,在实现基于窗口的 Win32 程序时,第一步并非从消息处理程序与用户处理程序入手,而是应该先弄清整体框架,再进行搭建。各个程序的调用关系为主程序-窗口主程序-窗口消息处理程序-用户程序,从这个角度出发,则包括两个步骤:搭建整体框架及添加窗口消息处理程序与用户程序。搭建完框架后,在添加窗口消息处理程序前,还需对资源文件进行再定义,即添加 4 个所需菜单栏与对应的消息号。在进行窗口消息输出时,需注意 API 函数 TextOut 只能输入字符类型数据,且还需考虑字符长度。考虑到数据类型十分多样,包含正负 0,以及个十百位等数据,故通过创建固定存储空间的方式,将转换数据存放到该空间内,剩余位数由空格填充,即解决了转化的统一性还解决了位数的统一性,可谓一举两得。再与控制台的信息输出进行比较,可以明显发现,控制台的信息输出是在控制台界面进行的,而窗口风格下的信息输出是基于创建窗口输出对于菜单栏和数据的。且控制台输出可实现数字类型的输出,而窗口下输出为字符类型,需要进行数串转换。

其次,则是对不同环境的对比。X86 下 64 位程序与 32 位程序存在些许差别,大致可分为整体和细节上的差别。整体上,删除了诸多伪指令的使用,如 `.model`, `.stack`。在细节上,则添加和扩展了通用寄存器,以及修改了对 C 语言函数和 API 函数的调用方式,制定了严格的调用规则。接着是对 ARMv8 与 80x86 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。

5.6.2 主要看法

对 16 位、32 位以及 64 位程序进行比较,不妨以 32 位程序作为中间基准。三者的根本差异在 CPU 可寻址宽度上可以明显体现,同时为了满足该要求在编程规范上也做了调整。比较 16 位与 32 位程序的区别,16 位程序一般在实方式环境中运行,如 DOSBox 环境,同时为了便于连接软硬件层面,比 Win32 位程序多了 DOS 系统功能调用 INT 指令。由于 16 位程序比 32 位程序更加细微,故在设置中断等软硬件结合方面更加方便。64 位程序则是在 32 位程序下进行了一些取舍,增加和扩展了通用寄存器,同时也舍弃了诸多伪指令用法。虽然在一定方面扩大了可查找范围,但对于简单编程来说,由于 C 语言函数即 API 函数调用的规范化限制,也增加了编程难度。

对 DOSBox 下的编程环境、WINDOWS 下的 VS2019、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 VS2019 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 VS2019 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 VS2019 作用区间的环境,保留了软硬件的链接

汇 编 语 言 程 序 设 计 实 验 报 告

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。

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, 2021