

**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2011班**

**学 号：**

**姓 名：**

**指导教师：**

**实验时段： 2022年3月7日~4月29日**

**实验地点： 南一楼803**

**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：2022.5.26

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 2](#_Toc97397791)

[1.1 目的与要求 2](#_Toc97397792)

[1.2 实验内容 2](#_Toc97397793)

[1.3 内容1.1的实验过程 2](#_Toc97397794)

[1.3.1 设计思想 2](#_Toc97397795)

[1.3.2 流程图 3](#_Toc97397796)

[1.3.3 源程序 5](#_Toc97397797)

[1.3.4 实验记录与分析 10](#_Toc97397798)

[1.4 内容1.2的实验过程 15](#_Toc97397799)

[1.4.1 实验方法说明 15](#_Toc97397800)

[1.4.2 实验记录与分析 16](#_Toc97397801)

[1.5 小结 20](#_Toc97397802)

[1.5.1 内容1.1小结 20](#_Toc97397802)

[1.5.2 内容1.2小结 21](#_Toc97397802)

[二、利用汇编语言特点的实验 22](#_Toc97397803)

[2.1 目的与要求 22](#_Toc97397804)

[2.2 实验内容 22](#_Toc97397805)

[2.3 实验过程 22](#_Toc97397806)

[2.3.1 实验方法说明 22](#_Toc97397807)

[2.3.2 实验记录与分析 23](#_Toc97397808)

[2.4 小结 26](#_Toc97397809)

[三、工具环境的体验 28](#_Toc97397810)

[3.1 目的与要求 28](#_Toc97397811)

[3.2 实验过程 28](#_Toc97397812)

[3.2.1 WINDOWS10下VS2019等工具包 28](#_Toc97397813)

[3.2.2 DOSBOX下的工具包 32](#_Toc97397814)

[3.2.3 QEMU下ARMv8的工具包 34](#_Toc97397815)

[3.3 小结 38](#_Toc97397816)

[参考文献 40](#_Toc97397817)

# 一、程序设计的全程实践

## 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。

## 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。

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。

## 内容1.1的实验过程

### 设计思想

**1. 登录功能**

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。对于用户名和密码的输入，用宏cmp\_str实现字符串的比较。通过循环login使得输入错误的次数不超过三次。其中，寄存器ecx表示循环的次数，由于会受到invoke的影响，故用count1变量来另外保存ecx中的值。

**2. 菜单功能**

通过main\_menu循环来实现不断选择操作的功能，实现用户的输入操作选项。

**3. 计算功能**

对N组采集到的状态信息计算f。使用calculate子程序实现，通过循环Loop1来遍历计算SF的值。其中，寄存器ecx保存循环的次数，ebx保存每次计算的结构的地址，eax保存计算SF过程中的结果。

**4. 复制功能**

将计算好结果的buf分组复制给lowf,midf,highf。使用copy子程序实现，在循环遍历buf的过程中，分别用ebx,edx,eax寄存器保存midf,lowf,highf下一次复制的首地址，用寄存器esi保存遍历buf过程中每个结构的地址，用ecx记录循环的次数。在判断完SF和0的关系后，将对应存储区的下一次复制地址送到edi寄存器中，利用movsb指令实现字符串的复制。以上功能都在文件main.asm中实现；公共变量为str8，即打印midf中数据的标准格式。

**5. 展示功能**

将midf存储区的各组数据在屏幕上显示出来。该功能在文件subp.asm中实现；使用display子程序；传递参数buff，代表需要打印的数据的首地址；在程序中定义了num局部变量，表示循环打印的次数，通过while伪指令实现循环。

### 流程图

主程序的流程图如图1.3.1所示。

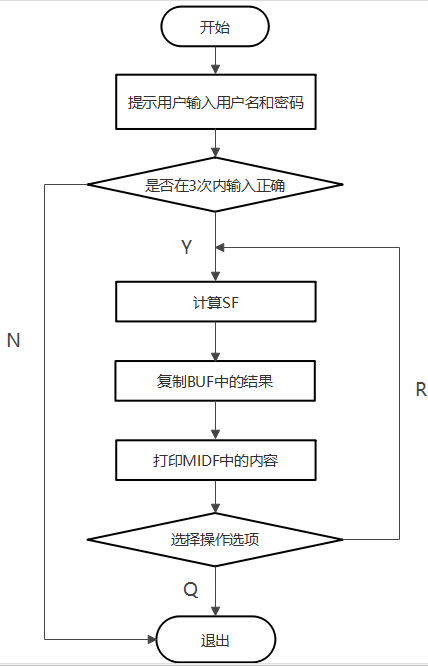


图1.3.1 主程序流程图

计算SF的值的子程序calculate的流程图如图1.3.2所示。

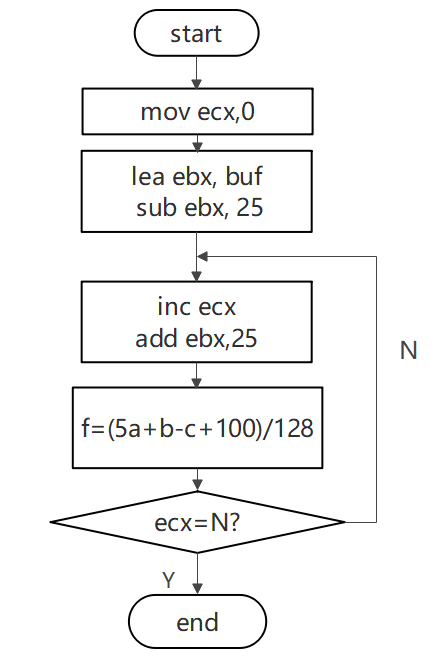


图1.3.2 calculate流程图

复制BUF中数据的子程序copy如图1.3.3所示。

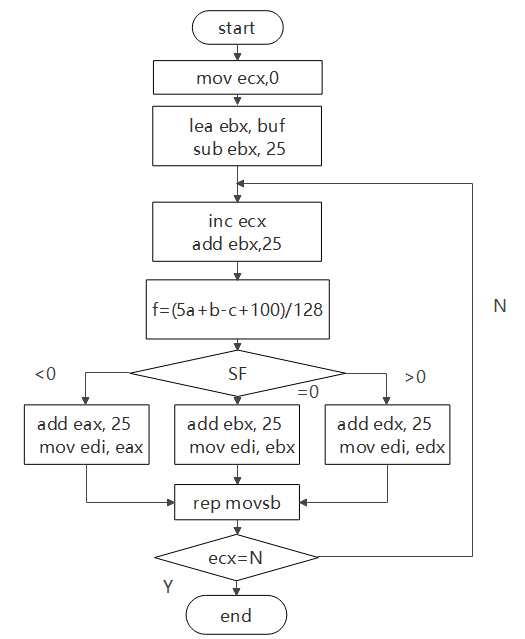


图1.3.3 子程序copy流程图

打印MIDF的子程序display如图1.3.4所示。

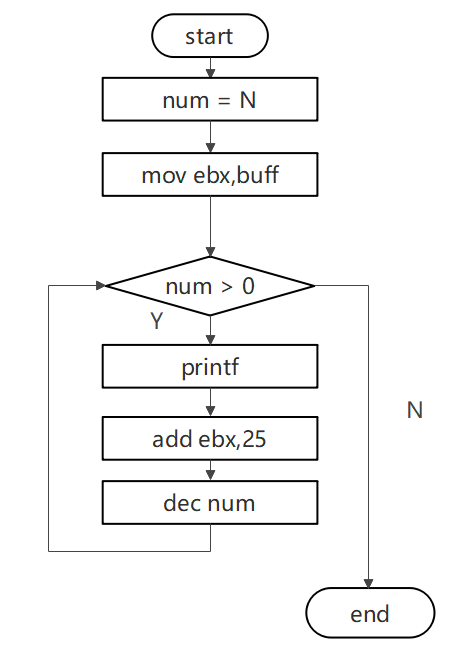


图1.3.4 子程序display的流程图

### 源程序

;任务3.1中，main.asm文件的代码如下：

.686P

.model flat,c

ExitProcess proto stdcall :dword

main\_menu\_print proto c

login proto c

add\_data proto c:dword

display proto:dword

printf proto c :ptr sbyte, :vararg

scanf proto c :ptr sbyte, :vararg

strcmp proto c :ptr sbyte, :vararg

includelib kernel32.lib

includelib libcmt.lib

includelib legacy\_stdio\_definitions.lib

;被其他文件调用的公共量一定要设为public，否则会报错

public str6,str7,str8,str10,buf,midf

;数据结构

SAMPLES STRUCT

SAMID DB 9 DUP(0) ;每组数据的流水号

SDA SDWORD 256809 ;状态信息a

SDB SDWORD -1023 ;状态信息b

SDC SDWORD 1265 ;状态信息c

SF SDWORD 0 ;处理结果f

SAMPLES ENDS

N = 5

.data

buf SAMPLES <'20220301',100,200,300,0>

SAMPLES <'20220302',-100,-200,-300,0>

SAMPLES <'20220303',1,2,3,0>

SAMPLES N-3 DUP(<>)

lowf SAMPLES N DUP(<>)

midf SAMPLES N DUP(<>)

highf SAMPLES N DUP(<>)

;用户名和密码

bname db '1',0

iname db 20 dup(0)

bpass db '1',0

ipass db 20 dup(0)

;输出用到的字符串

str1 db '用户名: ',0

str2 db '密码: ',0

str3 db '用户名不存在',0ah,0

str4 db '密码错误',0ah,0

str5 db '登录成功',0ah,0

str6 db '操作成功',0ah,0

str7 db '请选择要执行的操作：',0

str8 db 'SAMID：%s SDA：%d SDB：%d SDC：%d SF：%d',0ah,0

str9 db '请输入有效字符: ',0

str10 db 'midf的数据信息如下：',0ah,0

count1 dd 0 ;记录登录的次数

count2 dd 0

choice dw ?

flag db 3

lpfmt db '%s',0

numtype db '%hd',0

.stack 200

.code

;计算f的值

calculate proc

mov ecx, 0

lea ebx, buf

sub ebx, 25

Loop1:

inc ecx

add ebx, 25

imul eax, [ebx].SAMPLES.SDA, 5

add eax, [ebx].SAMPLES.SDB

sub eax, [ebx].SAMPLES.SDC

add eax, 100

mov [ebx].SAMPLES.SF, eax

sar [ebx].SAMPLES.SF, 7 ;优化：用sar取代idiv指令

cmp ecx, N

jnz Loop1

ret

calculate endp

;复制每组数据

copy proc

lea ebx, midf ;优化：用lea取代mov指令

sub ebx, 25

lea edx, lowf

sub edx, 25

lea eax, highf

sub eax, 25

lea esi, buf

sub esi, 25

mov ecx,0

mov count2,ecx

Loop2:

mov ecx, count2 ;记录循环次数

inc ecx

mov count2, ecx

add esi, 25

.if [esi].SAMPLES.SF>0

jmp high1

.elseif [esi].SAMPLES.SF<0

jmp low1

.else

jmp mid1

.endif

mid1:

;sf = 100的情况

add ebx, 25

mov edi, ebx

jmp over

low1:

;sf<100的情况

add edx, 25

mov edi, edx

jmp over

high1:

;sf>100的情况

add eax, 25

mov edi, eax

;复制buf中的内容

over:

mov ecx, 25

rep movsb

mov ecx, count2

cmp ecx, N

jnz Loop2

ret

copy endp

main proc

;登录

.while(flag > 0)

call login

cmp eax,0

jnz main\_menu

dec flag

.endw

jmp exit

;菜单界面

main\_menu:

call main\_menu\_print

invoke scanf,offset numtype,offset choice

cmp choice,1

jz choice1

cmp choice,2

jz choice2

cmp choice,3

jz exit

choice1:

invoke add\_data, offset buf

jmp main\_menu

choice2:

call calculate

call copy

invoke display, offset midf

jmp main\_menu

exit:

invoke ExitProcess, 0

main endp

end

;任务3.2中，func.c文件中的代码如下：

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#pragma warning(disable:4996)

#define OK 1

#define ERROR 0

#define N 5

typedef struct samples {

char SAMID[9];

long SDA, SDB, SDC, SF;

} SAMPLES;

extern char str7[20], str6, str10[20];

extern SAMPLES buf[5];

extern SAMPLES midf[5];

/\*登录界面\*/

int login(void)

{

char user\_name[20];

char password[20];

printf("请输入用户名： ");

scanf("%s", user\_name);

if (strcmp(user\_name, "xujinhui")) {

printf("用户名错误！\n");

return ERROR;

}

printf("请输入密码： ");

scanf("%s", password);

if (strcmp(password, "U202011675")) {

printf("密码错误\n");

return ERROR;

}

printf("登陆成功!\n");

return OK;

}

/\*展示菜单界面\*/

void main\_menu\_print(void)

{

system("pause");

system("cls");

printf(" 程序菜单\n");

printf("1.输入采集数据 2.计算并打印midf内容 3.退出程序\n");

printf("%s", str7);

return;

}

/\*输入并覆盖buf的第一条数据\*/

void add\_data(char\* buff)

{

printf("请输入覆盖数据信息：\n");

scanf("%s %d %d %d", buff, buff+9, buff+13, buff+17);

printf("%s", str6);

return;

}

;任务3.2中，subp.asm文件中的代码如下：

.686P

.model flat,c

printf proto c :ptr sbyte, :vararg

extern str8:sbyte

N = 5

.code

;display: 显示数据子程序

;buff: 缓冲区的首地址

display proc buff:dword

local num:dword ;定义局部变量，表示存储循环的次数

mov ecx, N

mov num, ecx

mov ebx, buff

.while(num > 0)

invoke printf, offset str8,ebx,dword ptr [ebx+9],dword ptr[ebx+13],dword ptr[ebx+17],dword ptr[ebx+21]

add ebx, 25

dec num

.endw

ret

display endp

end

### 实验记录与分析

**1. 实验环境条件**

实验采用Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz ，1.19 GHz处理器，机带RAM为8.00 GB (7.75 GB 可用)；WINDOWS10下VS2019社区版。

**2. 汇编、链接中的情况**

1. 在main.subp文件中忘记声明公共变量，导致运行时出现“无法解析的外部命令”报错。
2. 在调用display子程序时，程序总是会重复打印，经过反汇编调试后发现，是由于在编写子程序时没有添加ret返回语句，导致esp的值改变。在添加ret返回语句后，成功解决问题。
3. 在菜单模块实现循环输入R操作选项时，程序进入死循环，不断打印。经过调试后发现，是由于在子程序中使用了全局变量，但没有为变量重新置零。
4. 菜单实现后，在调试输入操作选项时，程序直接跳过选项输入部分，检查后发现是菜单输入变量choice类型（字类型）与类型说明符不符。
5. 在执行复制BUF中的数据信息到存储区时，编译器报错A2114 ：INVOKE argument type mismatch : argument : 0。是由于[ebx]前未加dword ptr。

**3. 程序基本功能的验证情况**

1. 输入用户名和密码

程序的运行结果如图1.3.5所示。可以看到，当输入错误超过三次时，系统会自动退出。对于用户名和密码输入错误的情况，程序也能够分别判断出。

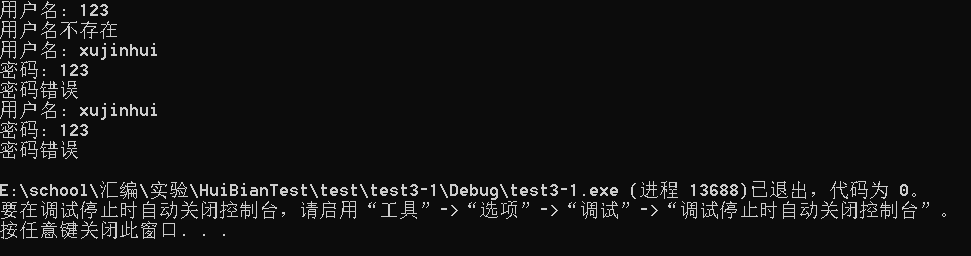


图1.3.5 用户登录界面

1. 进入菜单界面

程序的运行结果如图1.3.6所示。当用户名和密码均输入正确时，程序会执行任务（2），计算SF的结果并复制到各个存储区中，将MIDF中的数据信息打印出来。然后再进入菜单界面，获取用户输入的操作选项。

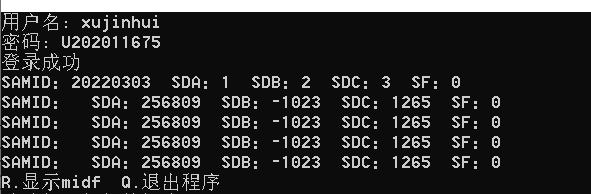


图1.3.6 菜单界面

1. R操作选项

程序的运行结果如图1.3.7所示。当用户输入R后，会重新执行任务（2），并再次等待用户输入操作选项。



图1.3.7 R操作界面

1. Q操作选项

程序的运行结果如图1.3.8所示。当用户输入Q后，退出程序。



图1.3.8 Q操作界面

**4. 使用调试工具观察、探究代码的情况**

1) 程序开始运行时，堆栈中存储内容如图1.3.9所示。

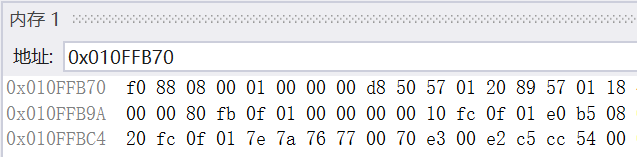


图1.3.9 程序开始时堆栈的内容

2) 利用VS2019的单步执行，在call处设立断点。可以看到在执行call语句时，call语句后的语句的地址进栈，同时esp地址-4，如图1.3.10，1.3.11所示。

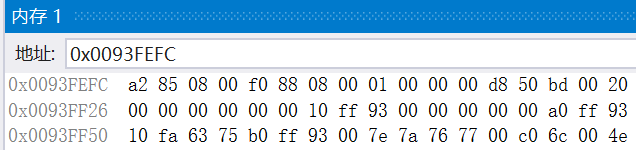


图1.3.10 执行call语句后的堆栈栈顶

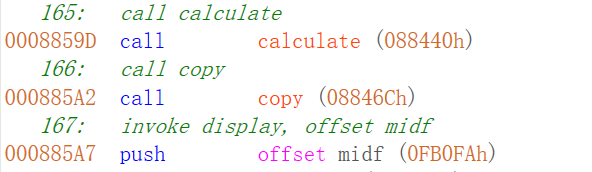


图1.3.11 栈顶对应地址的语句

执行ret语句时，栈顶地址出战，出栈数据存入EIP中，即pop EIP，同时语句跳转至EIP对应语句，即之前地址进栈的语句。如图1.3.12所示。

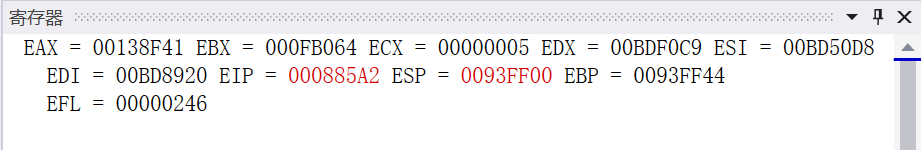


图1.3.12 执行ret语句后EIP的值变为栈顶元素值

若在执行ret前将栈顶元素修改，则执行ret语句时，程序仍会跳转至栈顶元素地址处，但如果栈顶元素的值不正确时会出现问题。例如，将栈顶元素值+5，则会跳转至call语句的后两条语句处，如图1.3.13所示。

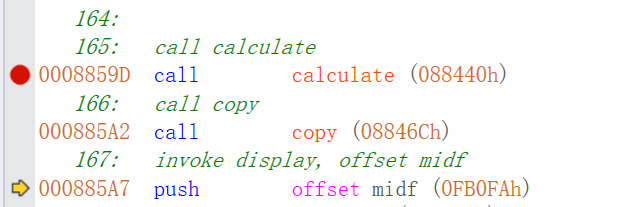


图1.3.13 栈顶元素＋5后的跳转结果

1. invoke伪指令包含将传入的参数进栈以及call语句，如图1.3.14所示。

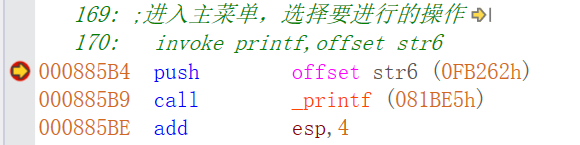


图1.3.14 invoke语句对应的反汇编

此外，函数体的开头会增加语句push ebp和语句mov ebp,esp，而ret语句前增加leave语句，如图1.3.15所示。

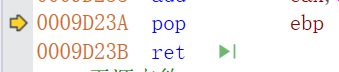


图1.3.15 ret语句反汇编结果

1. 子程序中局部变量的存储空间在堆栈中，可利用lea指令获取局部变量的地址。
2. 宏指令的调用，是直接将调用处语句替换为宏体内的语句，宏参数替换为传入的参数，如图1.3.16所示。

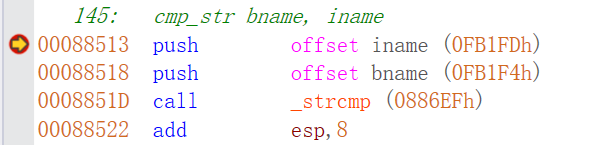


图1.3.16 一条宏指令的反汇编语句

1. 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，当其符号名不一致时会发生A2114 INVOKE argument type mismatch的报错，如图1.3.17所示。因此我们在使用公共符号时，应当格外注意声明的符号与引用的是否一致。

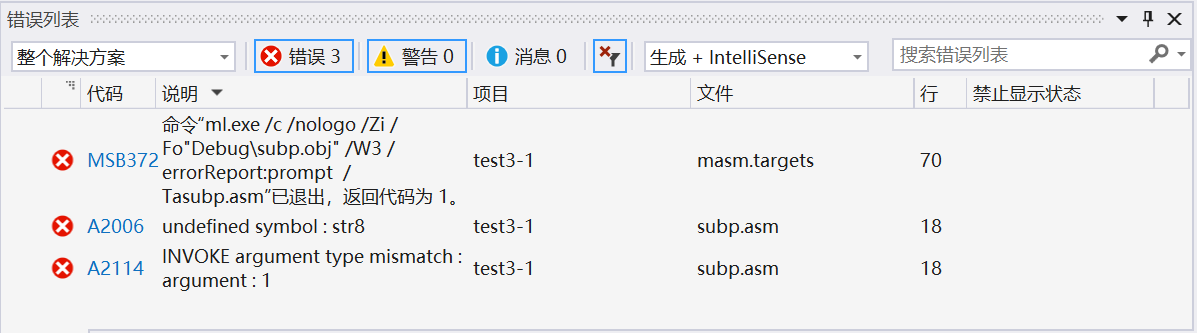


图1.3.17 公共符号的定义与引用不同的结果

**5.用C语言改造程序**

将用户登录、进入菜单界面用 C语言程序实现，并添加用C语言实现“修改采集的状态数据”的功能，即：在功能（4）时，按M键将可以输入一组采集数据（不包括f字段），用来覆盖N组采集到的状态信息中的第一组数据。实验的记录和分析如下：

1. 在C语言模块中调用汇编语言中定义的字符串时，声明语句应将其写成字符数组的形式，否则调用时会报错。如图1.3.19所示。

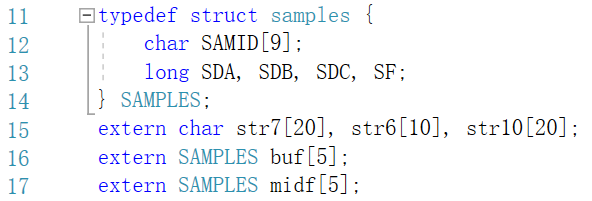


图1.3.18 C文件中数据的定义和引用

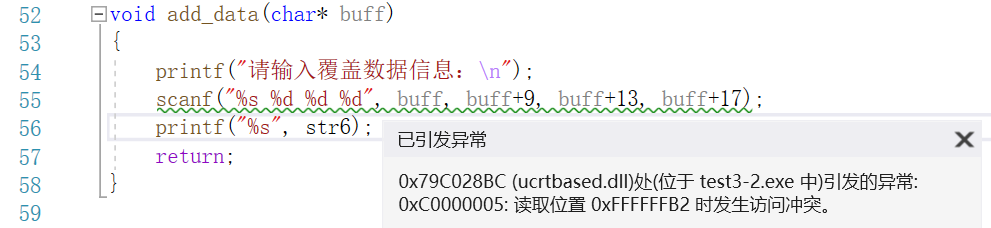


图1.3.19 声明错误的结果

1. 汇编程序中访问C中定义的变量时，未产生相应汇编指令（可能在预处理过程中完成），而是在调用时直接使用其地址。C程序中访问汇编程序中的变量时也是如此。
2. 由于在C程序中并无调用结构体的定义，因此应现在C程序中定义待引用结构体，随后用extern语句引用相应结构数组，并在”[]”中写明结构体个数。
3. 地址类型转换，即将该地址按照转换后的字节长度接受赋值。如在语句char a[10]; \*(int\*)a = 123;执行时，会用a[0]~a[3]这四个字节的空间存储数值123。

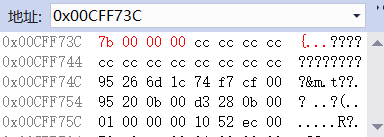


图1.3.20 a[10]的前四个字节的值发生变化

1. C语言函数调用语句与汇编语言函数调用相同，均为call语句。进入函数前，会将除eax，ecx，edx之外的寄存器入栈，return语句执行后恢复。执行return语句时，会将return后的值存入eax中，若无返回值则无上述过程。参数的传递同样是通过栈传递。

## 内容1.2的实验过程

### 实验方法说明

**1.指令优化对程序的影响**

1. 在取址时，灵活利用寻址方式的特点，用lea取代mov指令，有效减少程序运行时间。
2. 在计算SF的值时，用sar取代idiv指令，实现除法功能，可以有效优化程序。
3. 在复制BUF中的数据到各个存储区时，用rep movsb指令可以有效实现字符串的复制，并优化代码长度和执行效率。

**2.约束条件、算法与程序结构的影响**

1. 在存储BUF的数据时，将采集到的每组状态信息定义成一个结构，可以增强程序的可读性，简化代码逻辑。
2. 采用的计时方法是在待测程序执行前获取一个时间标签，执行之后再获取一个时间标签，然后通过计算这两个时间标签的差值来得到程序的执行时间。本次实验采用QueryPerformanceCount微秒级的计数器，比毫秒级的clock()精度更高，实验效果更加明显。由于CPU不仅仅是在为一个程序服务，还需要做其他操作，因此，重复执行该程序得到的计时数值会有一定的波动，需要取多次计时数值的平均值来比较。
3. 减少循环体内的代码指令。因为在循环体中，每减少一条指令，就相当于减少了“外循环次数\*内循环次数”条指令的执行时间，所以对程序的优化十分重要。
4. 减少重复代码的数量。在复制BUF中的代码到存储区时，原来的代码采用在判断后直接复制的算法，这样复制BUF到各个存储区的代码需要重复写3次。而在优化后，采用先判断再统一跳转到复制模块的算法，可以有效减少代码数量，提高程序效率。
5. 在记录循环次数时，原来的程序采用变量count保存，在每次循环的出入口都需要重复使用寄存器改变count的值。而在优化后，直接使用寄存器存储循环次数，减少了指令的数量，同时也提高了程序效率。

**3. 编程环境的影响**

将同一个程序放到不同软硬件运行环境下运行（比如：不同的计算机硬件下运行，虚拟机或非虚拟机下运行，调试环境或非调试环境下运行等），观察程序执行时间是否会随之发生变化。

### 实验记录与分析

**1.优化实验的效果记录与分析**

程序优化后的数据结构如图1.4.1所示。

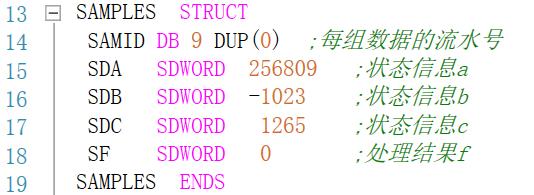


图1.4.1 数据结构的定义

处理的数据量为10000时，程序优化前的执行结果如图1.4.2所示，运行时间约为20000微秒，如图1.4.2所示。

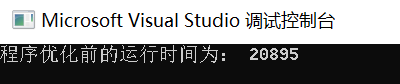


图1.4.2 程序优化前

将原本判断SF后立即复制的方法改为先判断后统一复制的方法，并减少循环体内的重复代码后，运行时间约为1300微秒，如图1.4.3所示。

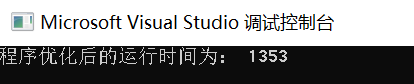


图1.4.3 第一次优化后

改变取址指令，将复制字符串的指令改为movsb，将除法用移位指令实现，运行时间明显再次缩短，约为500微秒。如图1.4.4所示。



图1.4.4 第二次优化后

在优化前，程序约有140行代码，而在优化后，程序约有100行代码，代码的行数减少了约30%，代码得到了极大的简化。



图1.4.5 优化前代码行数

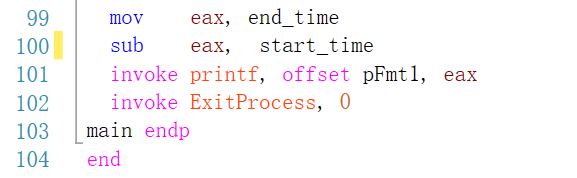


图1.4.6 优化后代码行数

**2. 不同约束条件、算法与程序结构带来的差异**

因未能发现更好优化方案，故尝试添加无效语句观察时间变化。在复制BUF数据的循环中，将一条mov指令复制3次，观察到运行时间明显延长，从原本的约20000微秒变为约22000微秒，如图1.4.5所示。由此可见，循环内的少数多余语句可能对程序运行时间造成很大影响。

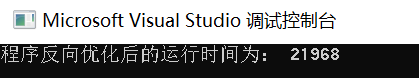


图1.4.7 反向优化后

将处理的数据量改为100，运行优化前的程序，如图1.4.6所示，运行时间为380微秒。

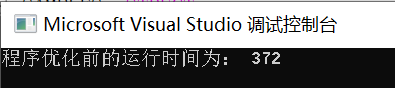


图1.4.8 程序优化前

处理的数据量为100时，程序优化后的执行结果如图1.4.7所示，运行时间为140微秒。

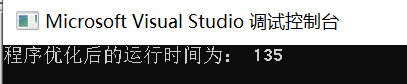


图1.4.9 程序优化后

由以上结果可以看出，在处理10000条数据时，优化后比优化前的运行时间缩短了约19500微秒；在处理100条数据时，优化后比优化前的运行时间缩短了约240微秒。故程序的优化可以极大缩短运行时间，提高程序效率。并且处理的数据量越多，优化效果越明显。

**3.几种编程环境中程序的特点记录与分析**

在调试环境下运行，观察到程序运行时间变为约21000微秒，如图1.4.10所示。

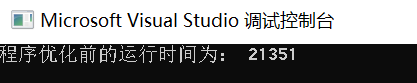


图1.4.10 调试环境下运行结果

Intel core i5 9400F环境下运行，程序运行时间约为18300微秒，如图1.4.11所示。



图1.4.11 在其他计算机环境下的运行结果

由此可以得出，影响一个系统最终能否达到设计目标的因素是有很多方面的，不仅有功能问题、算法问题、程序结构问题、指令选择问题，还有运行环境问题等等。将同一个程序放到不同软硬件运行环境下运行，程序执行时间也会随之发生变化。

**4.实验中出现的问题记录与分析**

1. 在使用mov xx,byte ptr ebx时，出现invalid use of register错误，经过调试发现是因为寄存器前不能使用xx ptr，若要使用，则应使用byte ptr [ebx]的形式。
2. 在使用跳转命令jmp low命令时，出现syntax erro low报错，经过查找资料发现是因为low和high都是关键词，不能作为跳转标志，可以修改命名消除报错
3. 在使用jg命令时，出现jump destination too far报错，查阅资料后发现，是由于je等命令有字节长度的限制，而jmp没有，故将jg转化为jmp后可消除报错。
4. 最开始使用div命令实现除法功能时，出现integer overflow，经过调试后发现是由于SF是有符号数，故应使用idiv有符号除法指令。
5. 在scan附近使用寄存器，会发现有些寄存器的值被影响而有些不会，导致输出结果错误。可以使用push和pop来保护寄存器。
6. 在复制BUF中的值到存储区时，出现了访问冲突异常，经过使用内存、监视器窗口进行调试后，发现是由于使用了esp寄存器，导致程序中断。在改变寄存器后解决了问题。部分运行时的报错信息如图1.4.5所示。

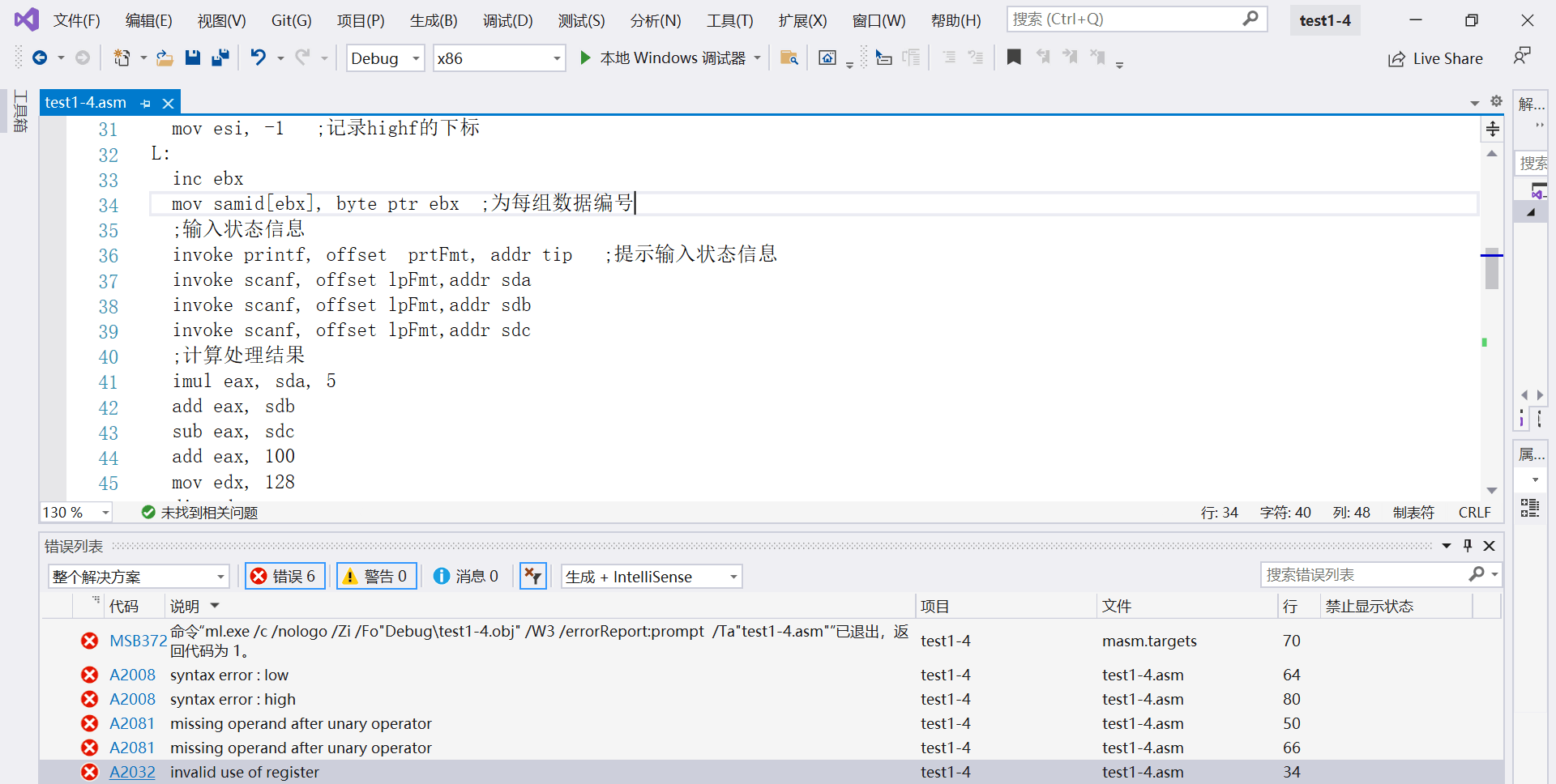


图1.4.5 部分报错信息

## 小结

### 内容1.1小结

通过编写带有菜单的程序的过程中，让我了解到利用跳转指令实现循环结构和分支结构的方法，也体会到高级语言在此方面的优越性。此外，在程序编写过程中，多次出现的寻址问题也深化了我对汇编程序中寻址方式的认识。在SF的计算过程中，除法是比较特别的一个——被除数在计算中会被扩展，DX和AX或EDX与EAX会共同构成一个被除数。在这一前提下，CDQ指令的使用便是保证计算结果正确的关键因素。另外，在汇编程序中，一个指令后不能接两个变量参数，这点也是值得注意的。

汇编程序中的一个重要部分是对寄存器的使用。在实验过程中，常常出现调用函数后某些寄存器的值发生变化的现象。这是由于调用了C库函数，而这些函数通常会改变EAX，ECX，EDX的值。因此在调用C库函数时应避开使用这些寄存器。

本次实验还探究了模块化编程方法，包括多源文件、多子程序以及汇编语言和C语言的混合编程。当程序达到一定规模时，将所有代码写在同一个源文件、同一个子程序中将会使程序难以阅读。同时，当相同的步骤需要反复进行时，会极大增大代码长度。因此，将程序分解为多个子程序、多个源文件时很有必要的。

在C语言编程中，我们很少关心调用函数时参数的传递和语句的跳转，但在汇编语言中，这些功能都需要我们自己去完成。为了防止函数调用时改变寄存器中不应被改变的值，我们需要对其进行保护，即利用堆栈储存或尽量避免使用这些寄存器。另外，由于函数的跳转等都涉及到堆栈的使用，我们也应该尽量避免对寄存器esp、ebp的使用。

对于部分结构，如嵌套的循环、分支等，用汇编语言编写会较为复杂，因此我们可以用C语言来代替这些部分。而汇编语言由于更加贴近机器语言，在设计得当的情况下执行效率又会优于C语言。因此在合适的情况下对二者进行结合有助于提高程序的编写效率和运行效率。

### 内容1.2小结

本次实验的主要内容是程序优化，任务要求中提到不只考虑纯算法方面的优化，故此次实验我分别从指令和程序算法语句两个方面进行了优化。

在指令优化方面，通过使用lea指令、移位指令、movsb指令等代替原来的指令，实现了程序的效率优化，让我了解到了不同指令的特点，以及根据程序功能选取最合适指令的重要性。

通过减少循环内重复代码和反向“负优化”，让我了解到对语句的精简，尤其是在多重循环中的语句精简的重要性，即多余的语句在大量的循环中可能会严重拖慢程序运行的时间。

另外，在实验中，我也明显地感觉到寄存器有些不够用。在后续程序设计中，我也将想办法解决这一问题，以免在程序运行中出现寄存器冲突导致的难以发现的错误。scan函数会影响寄存器的值，因此在使用时要格外小心，并且在程序中尽量不要使用esp寄存器，因为这可能会带来一些意想不到的错误。

程序的运行时间受环境影响很大，且即使在同一环境下，重复运行的时间也有所不同。对于本次的汇编实验，在调试与非调试下运行时间相差并不明显，这可能是汇编程序的优势。

此外，这次实验让我了解到了系统运行的结果不仅仅与指令、算法有关，还与系统所处的环境有关。将同一个程序放到不同软硬件运行环境下运行，程序执行时间也会随之发生变化。

# 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**

1) 在主程序中设置接管系统08H中断程序，保存原08H中断程序地址，设置程序驻留。

2) 在新的08H中断程序中，利用IO指令读取CMOS芯片中的时间信息并输出到屏幕上。

3) 程序的流程图如图2.3.1所示。

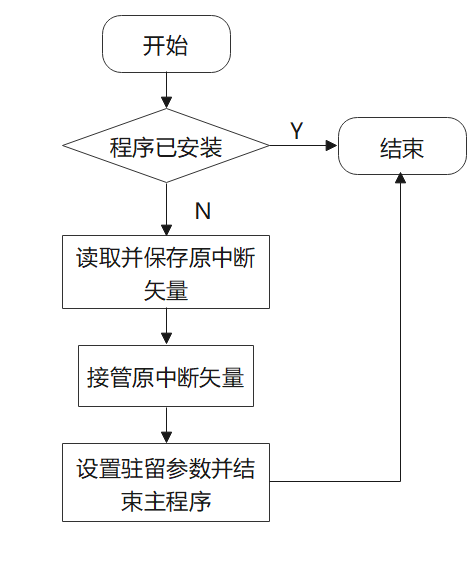


图2.3.1 任务4.1的流程图

**2.反跟踪程序的设计思想与实验方法**

1）利用逻辑运算，实现用户名和密码等信息的加密处理。

2) 采用计时、动态修改执行代码等方式实现反跟踪。

**3.指令优化及程序结构的实验方法**

1. 在取址时，灵活利用寻址方式的特点，用lea取代mov指令，有效减少程序运行时间。
2. 在计算SF的值时，用sar取代idiv指令，实现除法功能，可以有效优化程序。
3. 在复制BUF中的数据到各个存储区时，用rep movsb指令可以有效实现字符串的复制，并优化代码长度和执行效率。
4. 在存储BUF的数据时，将采集到的每组状态信息定义成一个结构，可以增强程序的可读性，优化程序结构。
5. 减少循环体内的代码指令。因为在循环体中，每减少一条指令，就相当于减少了“外循环次数\*内循环次数”条指令的执行时间，所以对程序的优化十分重要。
6. 减少重复代码的数量，可以有效减少代码数量，提高程序效率。
7. 在记录循环次数时，若程序采用变量保存，则在每次循环的出入口都需要重复使用寄存器改变变量的值。而在优化后，直接使用寄存器存储循环次数，减少了指令的数量，同时也提高了程序效率。

### 2.3.2实验记录与分析

**1.中断处理程序的特别之处**

1) 编写和调试完毕后，重复运行程序会报错，如图2.3.2所示。

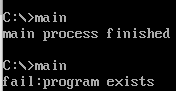


图2.3.2 重复安装时的报错信息

2) 在TD下观察到的中断矢量表如图2.3.3所示。当第二次运行时，可观察到原中断是量表被接管处被修改，如图2.3.4所示。

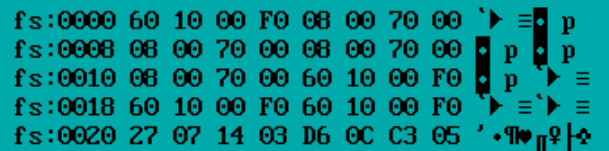


图2.3.3 接管前的中断矢量表

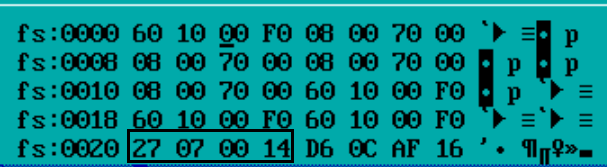


图2.3.4 接管后的中断矢量表

3) 在int语句处使用快捷键Alt+F7即可进入中断管理程序。如图2.3.5所示。

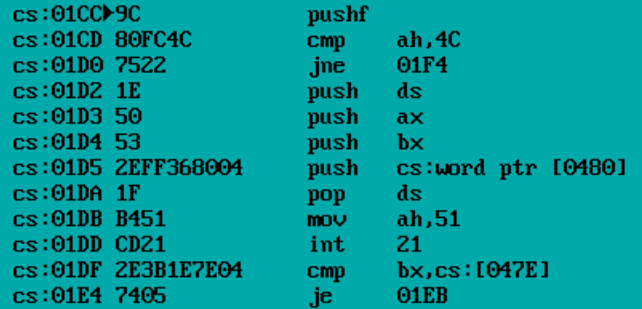


图2.3.5 中断管理程序内容

**2.反跟踪效果的验证**

1) 在数据段定义数据的过程中，使用“异或”对进用户名、密码进行加密。在使用密码时，先将输入的密码按照相同方式变换后与数据段中内容进行比较；在使用SF时，先对其及逆行相同的变换后代入SF的计算中。

2) 在部分函数中插入计时程序，防止单步执行跟踪。

3) 动态修改函数语句，将数据段中保存的部分语句的机器码复制到代码段中。

**3.跟踪与破解程序**

1) 利用W32asm反编译程序，并用串式数据参考功能读取数据内容，找到用户名和密码的位置，如图2.3.6所示。

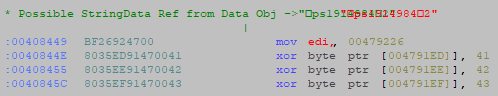


图2.3.6 代码中密码操作所在位置

2) 根据反汇编结果中的三个异或操作，对应数据段中相应位置可以计算出正确的密码。从数据段中可以得到三次异或操作结果，再次进行异或操作可以得到解密后的字符。

3) 利用前一步获得的密码登录系统，直接调用系统中的信息输出函数即可得到MIDF存储区的信，如图2.3.7所示。



图2.3.7 获取到的MIDF存储区信息

4) 反编译得到的程序，除了自己编写的源代码（经编译器处理）外，还包含大量库函数等语句，在一些位置还有大量的int 03语句，如图2.3.8所示。

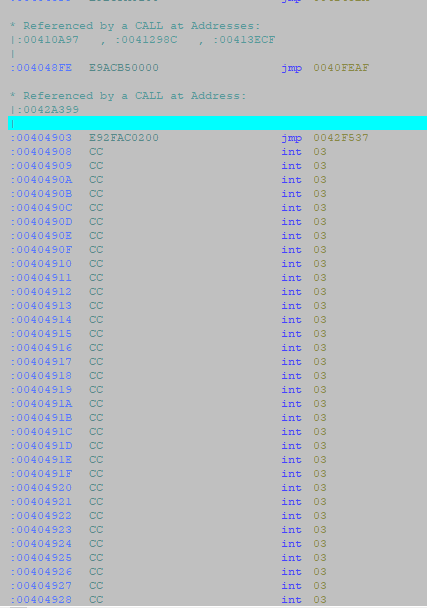


图2.3.8 反编译程序结果

**4.特定指令及程序结构的效果**

1) 处理的数据量为10000时，程序优化前的运行时间约为20000微秒。

2) 将原本判断SF后立即复制的方法改为先判断后统一复制的方法，并减少循环体内的重复代码后，运行时间约为1300微秒。

3) 改变取址指令，将复制字符串的指令改为movsb，将除法用移位指令实现，运行时间明显再次缩短，约为500微秒。

4) 在优化前，程序约有140行代码，而在优化后，程序约有100行代码，代码的行数减少了约30%，代码得到了极大的简化。

5) 添加无效语句观察时间变化。在复制BUF数据的循环中，将一条mov指令复制3次，观察到运行时间明显延长，从原本的约20000微秒变为约22000微秒。由此可见，循环内的少数多余语句可能对程序运行时间造成很大影响。

6) 将处理的数据量改为100，运行优化前的程序，运行时间为380微秒,程序优化后的运行时间约为140微秒。由以上结果可以看出，在处理10000条数据时，优化后比优化前的运行时间缩短了约19500微秒；在处理100条数据时，优化后比优化前的运行时间缩短了约240微秒。故程序的优化可以极大缩短运行时间，提高程序效率。并且处理的数据量越多，优化效果越明显。

## 2.4小结

本次实验的内容为中断与反跟踪，所涉及到的内容相比C语言更加接近操作系统甚至计算机硬件。在中断程序设计中，我们利用对中断程序和中断矢量表的修改，实现了对系统底层功能的修改和操纵；在系统时间的获取中，我们利用IO指令，实现从CMOS中读取当前时间，并保存到寄存器中的功能。这些任务的完成，使我了解到，程序的“中断”并不是简单的、字面意义上的中断，而是一种暂时停止程序的运行，并根据相应设置执行其他更加高级指令的方法。

加密与解密也是本次实验的重点。本实验中，除了利用异或等运算对明文信息进行加密外，还利用了动态修改代码的方式，即在程序运行过程中，将一部分数据段内容复制到代码段中（利用VirtualProtect函数解除保护），并将被复制内容的开头设置为下一条指令的机器码，从而增加代码迷惑性。

解密过程使用了W32asm软件。在W32asm中，软件可以分析出程序中定义的变量，并跳转到调用变量的位置，为密码的解密提供了方便。另外，软件也支持数据段等的分析，也方便了对加密信息的对照和破解。

中断是操作系统相对底层的功能。利用中断可以实现多种诸如从键盘中读取字符等底层命令。合理利用中断也能实现许多难以直接实现、以及在高级语言中通过标准库才能实现的指令。通过修改中断程序和中断矢量表，可以实现类似子程序的功能，但是实现的功能比欧通子程序更加接近操作系统。

程序的加密和解密，一直是软件设计和使用中绕不开的话题。通过合理的方法进行加密，可以大大增加不怀好意者对程序的攻击和破坏的难度，从而提高系统的安全性、阻止信息的泄露，也可以保护开发者的劳动成果不被窃取。而解密则是为了从现有的软件中破解出想得到的信息，绕开商业软件许可等目的。任何软件都会被破解，考验的则是加密者、解密者的功力，而优秀的反编译软件则是解密者的强大工具。利用W32asm或其他更高级工具，可以将编译得到的可执行文件反编译为汇编指令，从而方便解密者进行跟踪和破解。IDA等工具甚至可以将程序解析为多个模块，并展示其中的相互关系，降低了分析的难度。

中断和反跟踪，是汇编语言相对高级语言的优势。通过调用系统级指令，可以更为高效地实现某些功能，而不必调用标准库函数，提高程序运行效率。由于中断矢量表的修改可以影响到其他程序，因此合理地利用中断也可以实现不同程序间的交互，以及判断驻留后的程序是否重复运行。对于反跟踪，汇编语言可以通过复制数据段内容到代码段中，实现动态修改程序代码，从而扰乱跟踪者的注意力。这类功能均为高级语言不具备或很难实现的功能。对于跟踪破解，由于反编译软件直接获得的是汇编指令，因此利用汇编语言进行分析效率会更高，也会更准确。

总之，汇编语言与高级语言各有特点，高级语言优势在于容易理解和容易编写，而汇编语言的优势在于贴近底层、执行效率高，便于加密与跟踪。

# 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**

对于给定的C语言程序，使用VS2019进行编译、链接和调试。通过实验，可以得出以下发现：

1. 通过寄存器窗口，可以了解寄存器、段寄存器、标志寄存器等存储内容的变化；通过监视窗口，可以观察变量值的变化。通过反汇编窗口，可以了解C语言与汇编语句的对应关系， 当勾选“显示符号名”时，对变量的调用会使用[变量名]表示；而不勾选时则会直接使用地址。如图3.2.1和3.2.2所示：

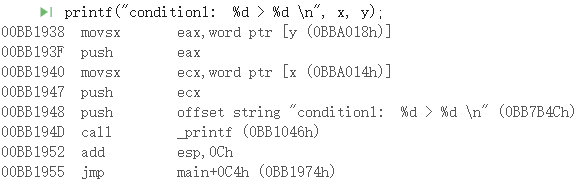


图3.2.1 勾选“显示符号名”

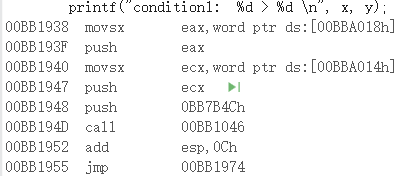


图3.2.2 不勾选“显示符号名”

1. 打开内存窗口，要注意查找变量内存时应使用取址符号。可以观察到int和unsigned型变量的存储空间为4个字节，short型变量的存储空间为2个字节，char型变量的存储空间为1个字节，char型数组的元素的存储空间连续，且所有变量都采用小端存储的方式。有符号与无符号整型数在内存中都以补码的形式存储，且存储的最小单位是字节；若有符号整型数为负数，则最高位为1，否则为0。各个数据的存储形式如图3.2.3, 图3.2.4, 图3.2.5所示。



图3.2.3 x和y在内存中的存储形式



图3.2.4 str[10] = "The end!"的存储形式



图3.2.5 a[5] = { 1, 2, 3, 4, 5 }的存储形式

1. 经过实验可以得出，系统对有符号数和无符号数的加减法都采用一样的策略，用他们的补码进行运算。以实验中的有符号数z = x – y的计算为例，系统按补码进行运算，用标志位来判断是否溢出。X原来的值为00000064H，存在EAX中，y原来的值为FFFF8044H，存在ECX中，经过相减计算后的值为00008020H，存在EAX中。在减法过程中，产生了借位，所以CY被置为1；而x和y的两个最高位不相同，所以OF为0。而在做无符号数加减法运算时，不需要考虑溢出标志位，只考虑进位标志位。此外，有符号数在执行比较指令时需要考虑最高位，而无符号数不需要考虑。

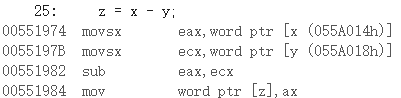


图3.2.6 有符号数的减法运算

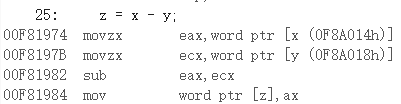


图3.2.7 无符号数的减法运算

1. 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。i<length 最终采用有符号数比较更好，因为i和length都是非负数。
2. 若将语句“z = sum(a, 5);”换成“ z = sum(a,0x90000000)；”，则会引发异常: 读取访问权限冲突。这是因为系统认为0x90000000是有符号数，而length是无符号数。如果将sum函数的参数 unsigned length改为 int length, 则不会引发异常。调试过程的部分截图如图3.2.1所示：

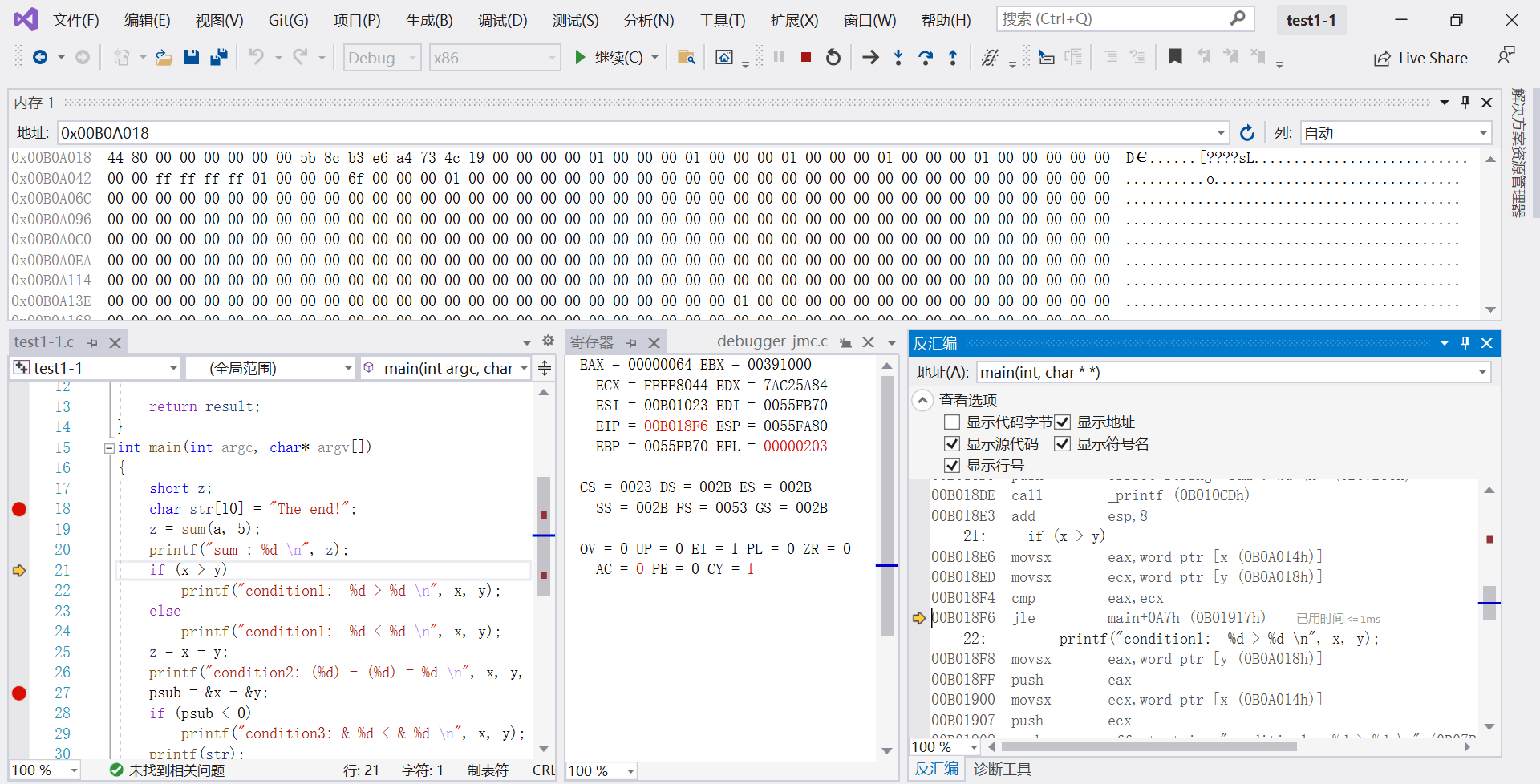


图3.2.8 调试过程截图

**任务1.2 观察汇编语言程序**

对于给定的汇编语言源程序，使用VS2019进行编译调试，得出的实验结果如下：

1. 通过反汇编窗口显示，可以看出汇编源程序中的语句与反汇编语句之间是严格一一对应的，虽然格式和大小写稍有不同。
2. 通过反汇编、内存等窗口可以看出，buf1的地址为0E7B02Dh，buf2的地址为0E7B039h，两者都采用小端存储，地址相差12个字节，刚好是buf1的长度。这说明数据段在内存中占用一片连续的存储空间，且不同变量间无分隔，如图3.2.9所示。



图3.2.9 数据段在内存中的存储

1. 原汇编程序中buf1的寻址方式为寄存器间接寻址方式，可将其改为变址寻址方式：

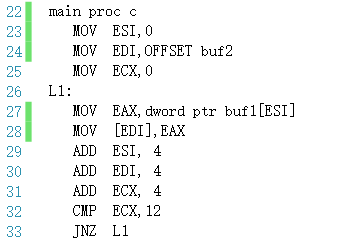


图3.2.10 修改后的寻址代码

**任务5.1 在VS2019下调试以下x64程序**

1. 在VS2019下创建项目，设置平台为X64，设置程序入口点。X64程序开头无处理器选择伪指令和存储模型说明伪指令。
2. 在声明函数原型的时候可以不声明参数表。因此可以不用proto进行声明。
3. 程序的入口默认为mainCRTStartup，也可根据需要手动设定。如果采用start为程序入口，需要在项目“属性-链接-高级”里指定start为入口点,否则会报错mainCRTStartup不能解析，如图3.2.11所示。

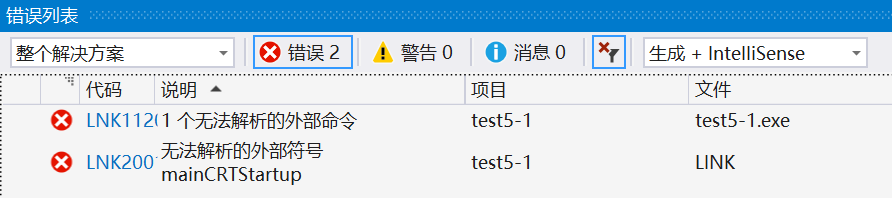


图3.2.11 程序报错示意图

1. 编译器不再支持invoke伪指令，只能使用call指令，并由调用者传递参数。
2. 增加了R8~R15共8个64位通用寄存器，原有的8个寄存器被扩展到64位，即共有16个64位通用寄存器。
3. 指定strat为程序入口后，程序可以正确运行，如图3.2.12所示。



图3.2.12 程序运行示意图

**任务5.3 查阅华为鲲鹏服务器所采用的CPU的汇编语言编程资料，体会与80X86的异同**

1. 段定义如图3.2.13和图3.2.14所示。

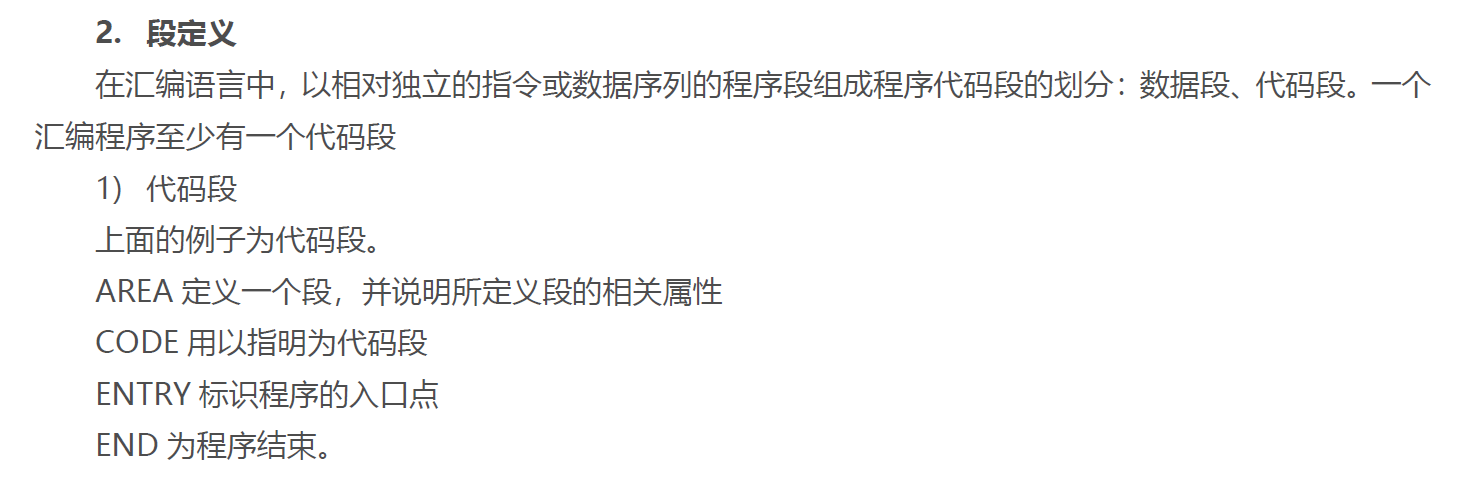


图3.2.13 代码段定义



图3.2.14 数据段定义

1. 调用子程序时使用跳转指令，如图3.2.15所示。



图3.2.15 子程序跳转说明

1. 开发环境需要为ARMv8环境，如图3.2.16所示。

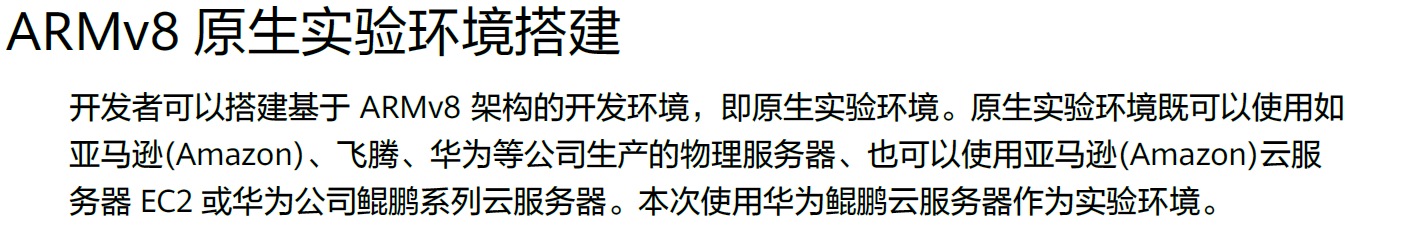


图3.2.16 开发环境说明

### DOSBOX下的工具包

1. 将DOSBOX压缩包解压并安装，在DOS 窗口中的提示符C: \>后输入：MASM DEMO.ASM，注意MASM.EXE和DEMO.ASM要在同一个目录下。如图3.2.17所示。



图3.2.17 打开DOSBOX窗口

1. 输入LINK DEMO.OBJ链接源文件，并用TD.EXE调试程序，如图3.2.18所示。

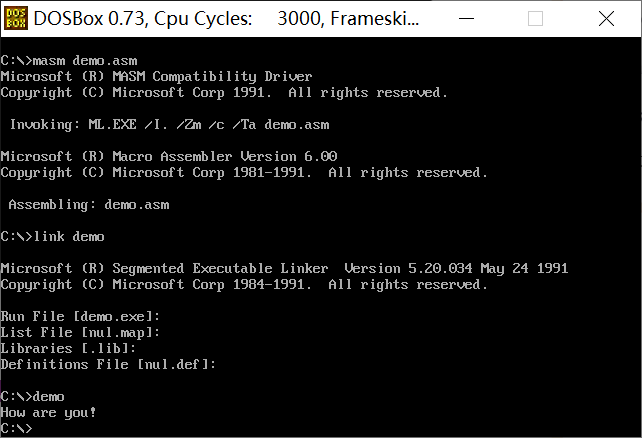


图3.2.18 编译链接源文件

1. 进入TD面板后调试。可以观察内存存放的位置与数值，是否正确。通过寄存器面板查看取出的值是否正确，单步调试（F8）查看程序的运行流程是否和想法相同。通过步入（F7）查看调用子程序内部是否按照正常逻辑执行。如图3.2.19所示。

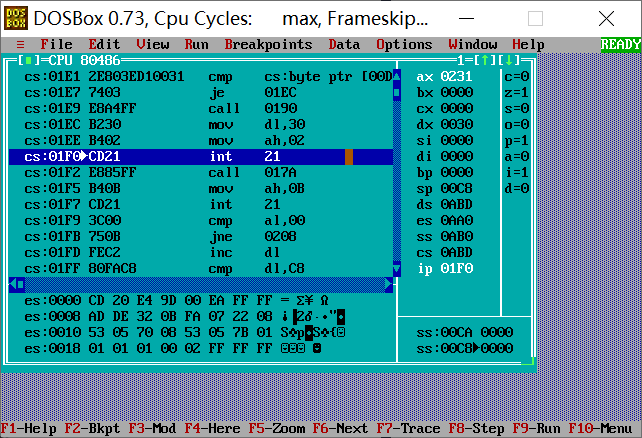


图3.2.19 TD调试过程

1. 通过标志位寄存器查看进位借位标志和程序跳转的状态等。如图3.2.20所示。

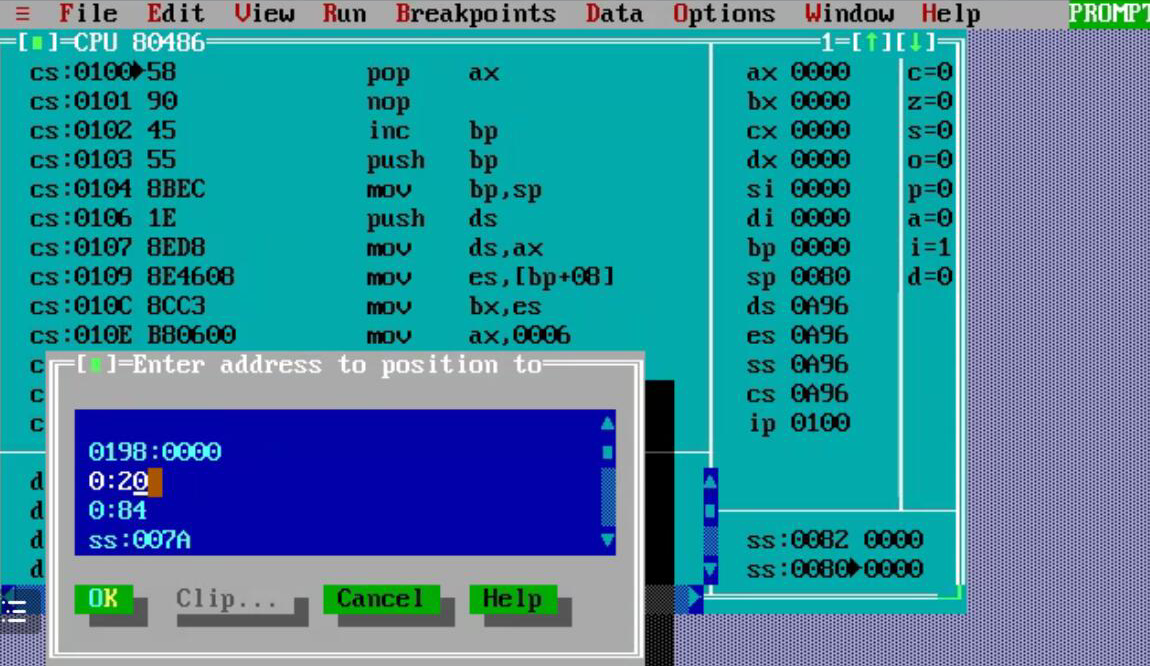


图3.2.20 通过TD查看程序跳转状态

### QEMU下ARMv8的工具包

1. 解压qumetest文件压缩包，并按使用说明修改环境变量，以管理员身份进入cmd，查看qemutest文件的子目录，如图3.2.21所示。

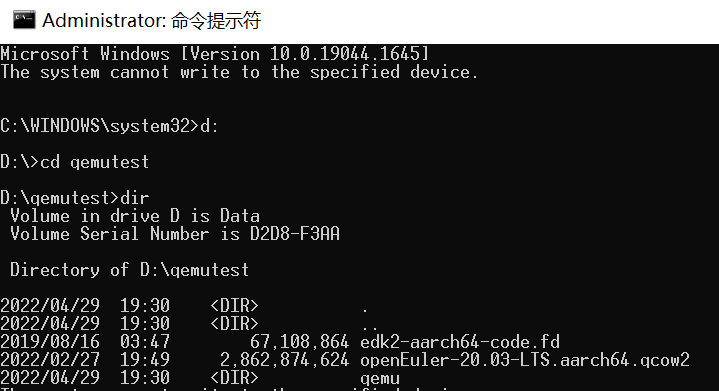


图3.2.21 进入qemutest目录

1. 启动qemu，在菜单“View”下选择Serial0，登录用户，如图3.2.22所示。

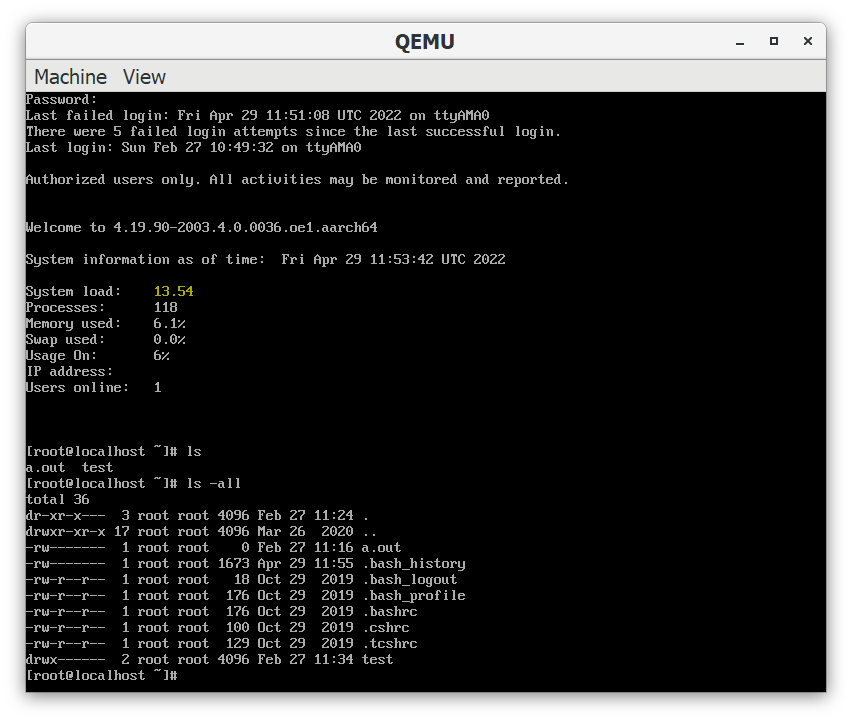


图3.2.22 qemu成功登录界面

1. 查看test文件下的目录，通过终端直接查看test1.c中的源代码，如图3.2.23所示。

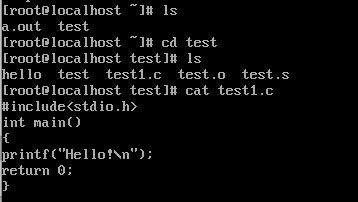


图3.2.23 通过cat查看test1.c源代码

1. 通过gcc命令可以在终端直接编译运行test1.c文件，如图3.2.24所示。

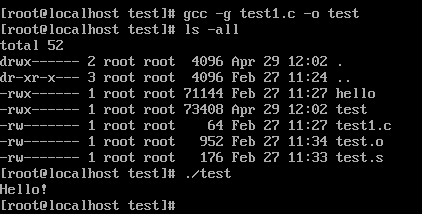


图3.2.4 编译运行test1.c

1. 可以通过(gdb)start开始执行程序，在第一条可执行语句处停下来，如图3.2.25所示。

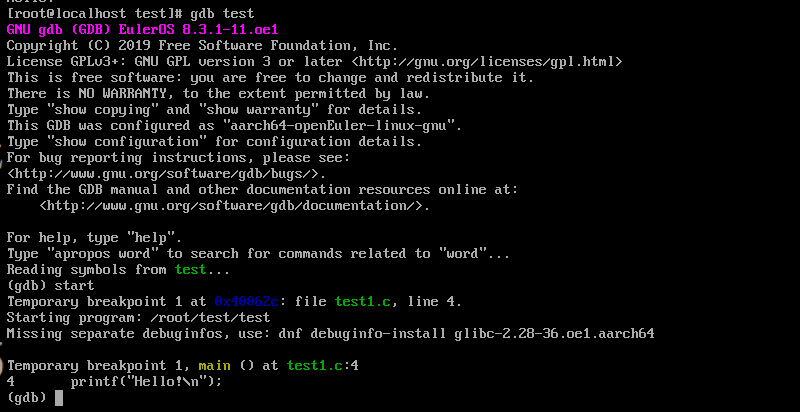


图3.2.25 gdb中运行程序

1. 在gdb中通过disass反汇编，可以用不同格式显示反汇编的内容，如图3.2.26所示。

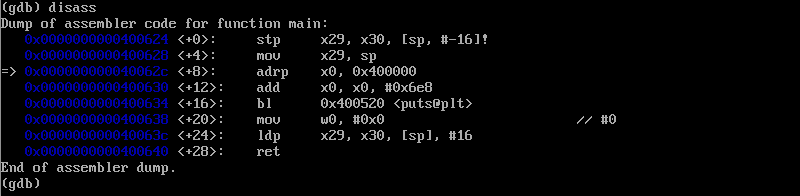


图3.2.26 gdb中反汇编

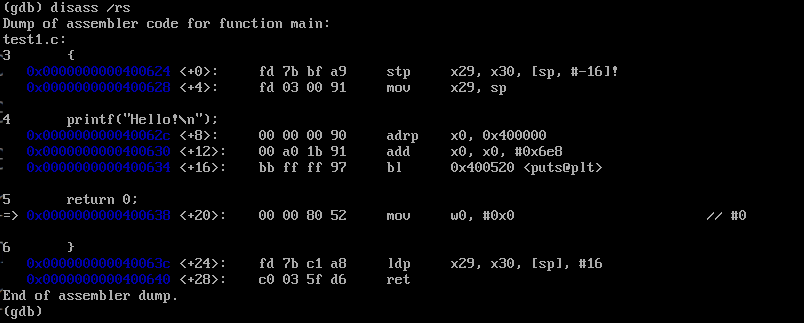


图3.3.27 gdb中查看指令机器码

1. 在gdb中可以通过reg查看寄存器中的内容，如图3.2.28所示。

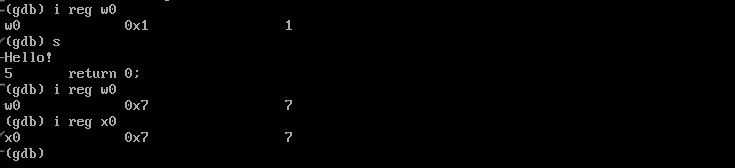


图3.2.28 gdb中查看寄存器

1. 编译执行“ARM虚拟环境安装说明”文档中“2.2.1”程序，可以看到 memorycopy 函数具体的执行时间为 47514738ns，如图3.2.29所示。

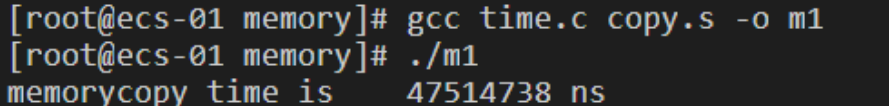


图 3.2.29 程序2.2.1执行结果

1. 对“2.2.1”程序进行优化，优化思路是一次对多个字节进行读写。这就用到了 ldp 指令和 stp 指令，这两条指令可以一次访问 16 个字节的内存数据，大大提高了内存读写效率。程序编译执行结果如图3.2.30所示。

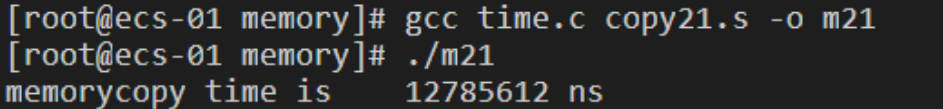


图3.2.30 程序优化后执行结果

1. ARMV8提供了X0~X30共31个64位通用寄存器，其低32位为W0~W30，是32位寄存器。执行写Wn寄存器时，会将Xn的高32位清零。
2. 在ARMV8中，用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。
3. 在指令中，立即数前要加#。
4. 指令后接3个操作数，第一个操作数保存后两个操作数执行后的结果。
5. 指令的机器码均为32位，对应CPU的位数，而X86中指令机器码长度不固定。
6. 子程序通过寄存器进行参数传递，返回值存储在R0中。
7. ARMv8汇编与C语言的混合编程需遵守ATPCS规则，在汇编语言中需用IMPORT伪指令声明C语言函数；在C语言中插入ARMv8汇编代码段与插入X86汇编代码段方法相同。

## 3.3小结

### 3.3.1 内容1.1小结

本次实验的主要内容是认识和了解汇编程序，学习汇编程序的编译、调试；利用学到的汇编指令实现分支、循环等结构。

本次实验中主要采用的是VS2019这一IDE，其好处在于简化了编译调试的过程，同时操作方便，界面也更为美观。但是其在标志寄存器窗口中使用的名称与教材有所差异，给使用造成了一定的不便。在实验中，对于同样的操作，C语言下的语句更加简单和贴近自然语言，而汇编语句则可能需要很多语句和跳转指令才能实现，体现出高级语言对编程的简化作用。

在VS2019中，对窗口菜单的编写提供了可视化页面，有助于简化菜单的编辑。但是在实际使用中，效果并不尽如人意，这可能是我对相关原理不够了解的问题。这也告诉我，在使用一个工具时，一定要尽量理解使用功能的作用。

本次实验还进行了32位/64位编程的比较。X64程序相对X86程序变化较小，主要在于代码开头部分的简化、寄存器的增加和扩展以及各种指令的64位升级，使从32位到64位的过渡更加容易，也保证了兼容性。

X86到X64的变化并不是彻底的改变，而是在原有基础上进行的增补和优化，这样做降低了开发者的学习成本、提高了CPU对软件的兼容性。但是，对兼容性的保证也导致许多开发者保持32位编程而不转向64位，一定程度上降低了软件运行效率。同时，CPU对过老指令集的兼容也会影响CPU的能效。相比之下，ARM平台一直在放弃对较老指令集的兼容，从而换来更高的能效比，大有赶超X86/64平台的趋势，这也是CPU升级发展需要做出的取舍。

ARMv8汇编语言和X86/X64差异较大。除运算指令大致相同外，大量指令的用法和功能都有所变化。在寄存器使用方面，ARMv8提供了更多的寄存器。但是与X86汇编不同，对64位寄存器的低32位进行操作时，ARMv8会将相应的高32位清零。另外，ARMv8和X86的显著差别在于，X86语句对应机器指令长度不固定，而ARMv8汇编指令的机器指令长度均为定值，这也是RISC和CISC两种模式的重要区别。

由于指令差别较大，在X86/X64环境下可以运行的程序在ARMv8下基本无法正常运行。这也告诉我们，软件的开发一定要选择正确的平台和指令集，错误的选择可能会使前期努力作废。指令集的不通用，为多平台开发维护造成了较大困难，希望未来能有更多更好的解决方案。

ARMv8指令属于RISC，即精简指令集，特点主要为指令长度均为定长。这样做以一定程度上降低语句执行的效率为代价，带来了容易编写、低能耗等优势，也是未来发展的方向。

# 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