

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 202107 班

学 号： U202115538

姓 名： 陈侠锬

指导教师： 李海波

实验时段： 2023 年 3 月 10 日~4 月 21 日

实验地点： 东九 A310

原创性声明

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

特此声明！

学生签名：

报告日期：2023. 5. 1

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 1.1 的实验过程	1
1.3.1 设计思想	1
1.3.2 流程图	1
1.3.3 源程序	3
1.3.4 实验记录与分析	3
1.4 内容 1.2 的实验过程	8
1.4.1 指令优化对程序的影响	8
1.4.2 约束条件、算法与程序结构的影响	10
1.4.3 编程环境的影响	13
1.5 小结	13
二、利用汇编语言特点的实验	14
2.1 目的与要求	14
2.2 实验内容	14
2.3 实验过程	14
2.3.1 中断处理程序	14
2.3.2 反跟踪程序	16
2.3.3 指令优化及程序结构	19
2.4 小结	20
三、工具环境的体验	21
3.1 目的与要求	21
3.2 实验过程	21
3.2.1 WINDOWS10 下 VS2019 等工具包	21
3.2.2 DOSBOX 下的工具包	25
3.2.3 QEMU 下 ARMv8 的工具包	26
3.3 小结	28
参考文献	29

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。

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 1.1 的实验过程

1.3.1 设计思想

任务 3.1 在任务 1.3、1.4、2.1 的基础上完成，整个程序大致分为输入用户名密码模块（宏定义实现，输入用户名密码并检查是否正确，最多输入三次）、计算模块（子函数 `cal_f`，计算每一组数据的 SF 值）、数据复制模块（包含三个复制函数 `copy_to_HIGHF`、`copy_to_MIDF`、`copy_to_LOWF`，根据 SF 值的不同将原始数据复制到对应模块中）、显示数据模块（子函数 `show_MIDF`，将 MIDF 存储区中的数据打印出来）、等待输入模块（在主程序中，在 `show_MIDF` 函数执行完后等待用户的输入）。

程序分为 `3.1.asm` 和 `3.1_functions.asm` 两个文件，复制模块和计算模块存放在 `3.1_functions.asm` 文件中，主程序与其他模块放在 `3.1.asm` 文件中。两文件的公共变量有原始数据数组 `data_buf`、三个存储区 `HIGHF`、`MIDF`、`LOWF` 以及原始数据数组指针 `buf_point`。

寄存器分配：`esi` 作为访问 `data_buf` 时使用的指针，`ebx` 作为访问三个存储区时使用的指针，`edx` 在 `cal_f` 函数中临时存储计算结果（此外还在显示数据模块作为变址寄存器），`eax` 在数据复制过程中作为中继。

1.3.2 流程图

任务 3.1 主程序流程图如图 1.1 所示，数据复制模块的流程图如图 1.2 所示。

汇编语言程序设计实验报告

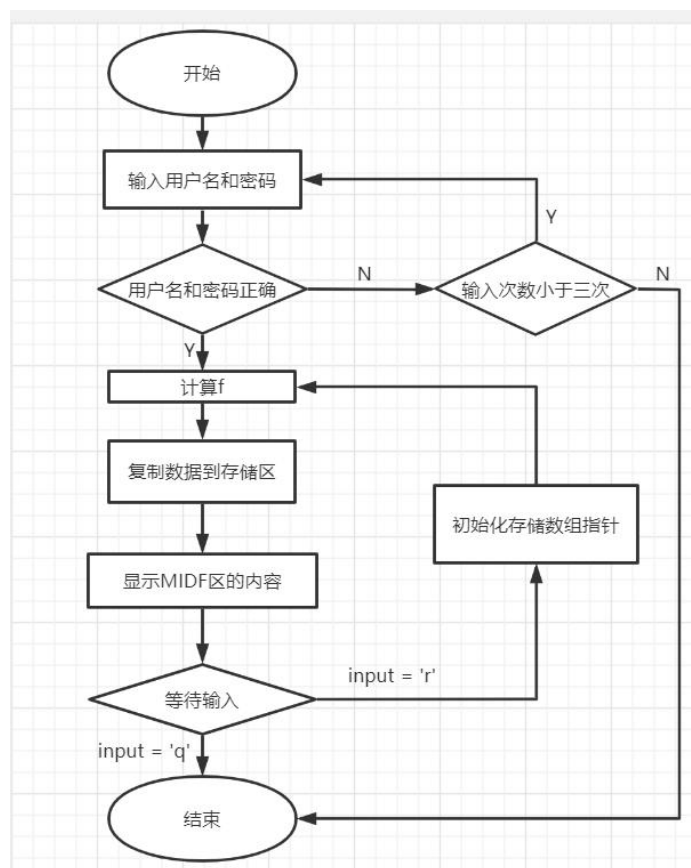


图 1.1 主程序流程图

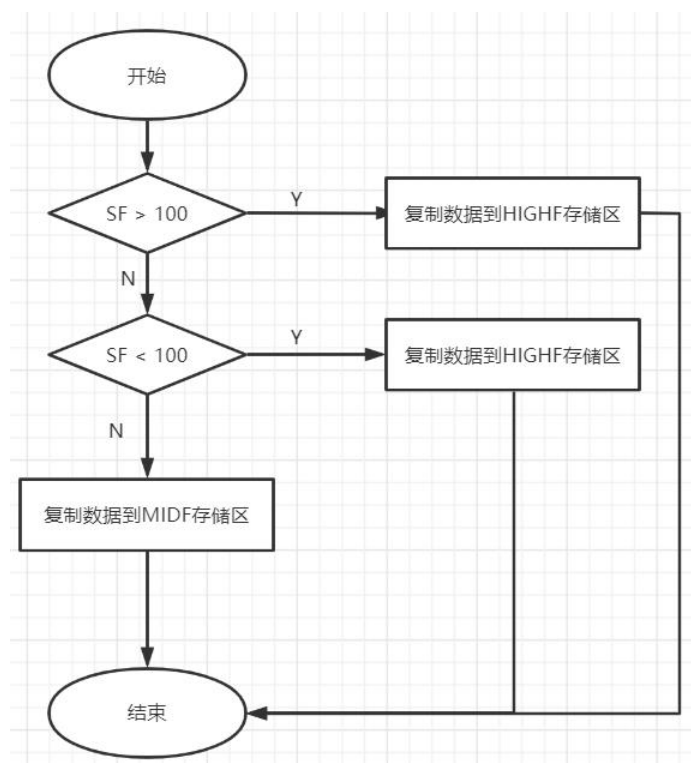


图 1.2 数据复制模块流程图

汇编语言程序设计实验报告

1.3.3 源程序

见电子版文件“3.1.asm”、“3.1_functions.asm”。

1.3.4 实验记录与分析

1. 实验环境条件

INTEL 处理器 2.69GHz, 16G 内存; WINDOWS11 下 VS2019 社区版。

2. 汇编、链接中的情况

在程序进行输入时, 首先出现了问题, 具体表现为在使用 scanf 读取一个数据并按回车之后, 必须再输入一个数据程序才能够进行到下一条语句 (整个程序中只有一条 scanf 语句), 通过与同学程序的反复比较, 我发现这是由于 lpFmt 这个变量在定义时出现了问题 (lpFmt 是存储字符 “%s” 的变量, 专门用于输入输出), 修改前后的 lpFmt 变量如图 1.3 所示。

```
lpFmt db '%s', 0; 正确定义, 输入和输出用  
;lpFmt db '%s', 0aH, 0dH, 0; 错误定义, 输出时无问题, 输入时会有问题
```

图 1.3 修改前后的 lpFmt 变量

修改前的变量同时存储了 0aH 和 0dH, 用于在输出时输出一个换行, 但是会导致用 scanf 读取数据时出现问题。

随后, 我在主程序中两次调用实现字符串比较功能的宏定义, 程序报错如图 1.4 所示。

✖ A2005 symbol redefinition: LO

图 1.4 程序报错 (之一)

LO 是我在宏中定义的跳转字符之一, 报错原因因为我两次调用同一个宏, 但没有将宏中的跳转字符用 local 声明。用 local 声明后此问题得以解决。(最终的程序中, 我把宏改写为先后进行用户名和密码的验证, 这样在主程序中只调用一次宏即可)

之后, 在实现输出 MIDF 存储区这一功能遇到了问题, 我在该子函数中定义了一个 tem_point 作为 MIDF 的临时指针, 一开始想要通过使用 tem_point 进行直接寻址, 但是程序会报访问冲突的错误, 在观察反汇编语句的寻址指令之后, 推测可能是因为局部变量 tem_point 占据内存空间导致直接寻址访问到的地址并不是 MIDF 存储区的地址, 因此我改用基址加变址的寻址方式访问 MIDF, ebx 作为基址寄存器, edx 作为变址寄存器, 如图 1.5 所示。同时我观察到在 printf 函数过后, 寄存器中的值可能会改变 (尤其是 edx), 因此每次访问 MIDF 之前都要重新往寄存器中赋值。

```
invoke printf, offset lpFmtc, MIDF[ebx + edx]; 基址加变址寻址
```

图 1.5 基址加变址寻址访问 MIDF

在初步完成了输出 MIDF 区数据这一功能后, 我尝试利用结构体的特点将输出简化, 即以 MIDF[ebx].SDA 这种形式访问第 ebx 个结构体的 SDA 数据, 从 SDA 至 SF 都可以正常输出, 但使用 MIDF[ebx].SAMID 输出流水号时会报错访问冲突, 我推测是因为 SAMID 使用 printf 作为字符串输出时, 结尾没有 0 而导致的报错。最终我还是选择以逐个字节输出的方式输出流水号。

输出 MIDF 存储区内容的代码如图 1.6 所示。

汇编语言程序设计实验报告

```
;输出流水号
NEXT_PRINT:
    invoke printf, offset lpFmt, offset samid
    mov tem_cnt, 0
    mov ebx, tem_point
PRINT_SAMID:
    mov edx, tem_cnt
    invoke printf, offset lpFmtc, MIDF[ebx + edx] ;基址加变址寻址
    inc tem_cnt
    cmp tem_cnt, 6
    JNE PRINT_SAMID
    invoke printf, offset lpFmtc, 0aH ;输出一个换行
    ;invoke printf, offset samid, MIDF
;输出剩余内容(4个字节一个内容)
    invoke printf, offset sda, MIDF[ebx].SDA
    invoke printf, offset sdb, MIDF[ebx].SDB
    invoke printf, offset sdc, MIDF[ebx].SDC
    invoke printf, offset sf, MIDF[ebx].SF
    add tem_point, 22
    ;add edx, 22
    invoke printf, offset lpFmtc, 0aH
    mov ebx, tem_point
    cmp ebx, MIDF_point
    jb NEXT_PRINT
```

图 1.6 输出 MIDF 存储区内容的代码

3. 程序基本功能的验证情况

程序启动后，首先需要输入用户名和密码，由宏定义实现的字符串比较功能将会对输入的内容与事先设定好的用户名和密码进行比较，用户最多有三次输入机会，三次输入错误后程序将会终止，如图 1.7 所示。

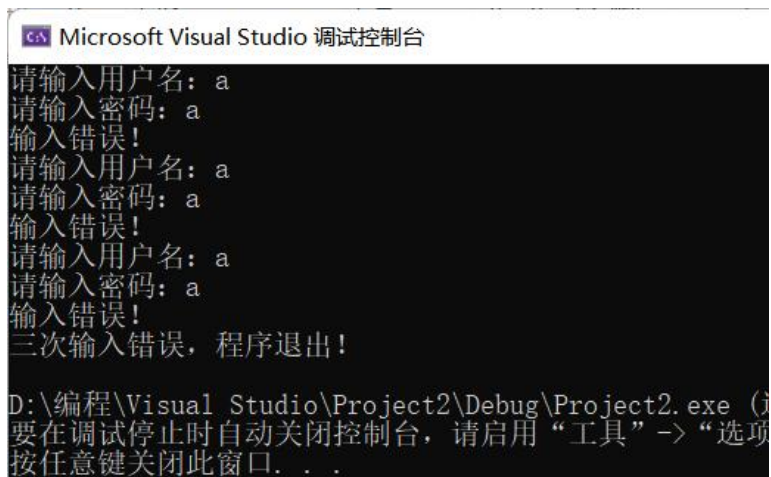


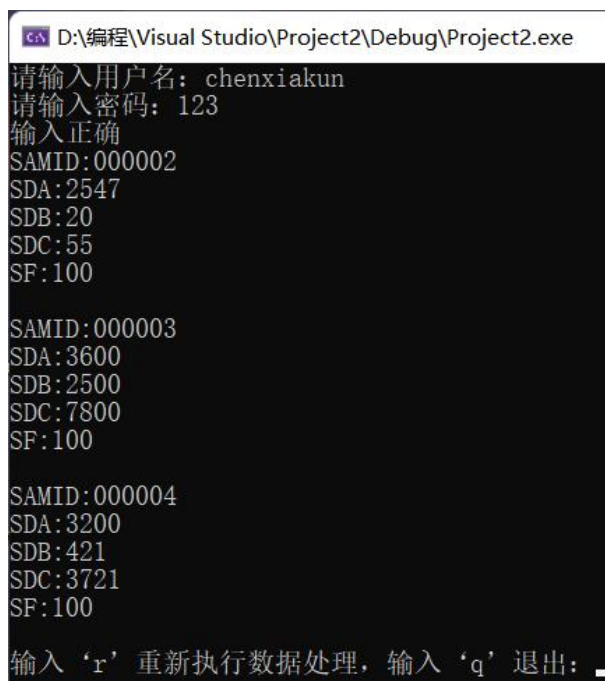
图 1.7 三次输入错误的情况

正确输入用户名和密码之后，程序将自动计算各组原始数据的 SF 值并根据此值存储到不同的区域，之后，程序会显示 MIDF 存储区中的内容，原始数据定义如图 1.8 所示，显示内容如图 1.9 所示。

data_buf	SAMPLES	<' 000001', 32100, 43200, 10, ?>	;HIGHF
	SAMPLES	<' 000002', 2547, 20, 55, ?>	;MIDF
	SAMPLES	<' 000003', 3600, 2500, 7800, ?>	;MIDF
	SAMPLES	<' 000004', 3200, 421, 3721, ?>	;MIDF
	SAMPLES	<' 000005', 1998, 200, 50, ?>	;LOWF

图 1.8 原始数据定义

汇编语言程序设计实验报告



```
D:\编程\Visual Studio\Project2\Debug\Project2.exe
请输入用户名: chenxiakun
请输入密码: 123
输入正确
SAMID:000002
SDA:2547
SDB:20
SDC:55
SF:100

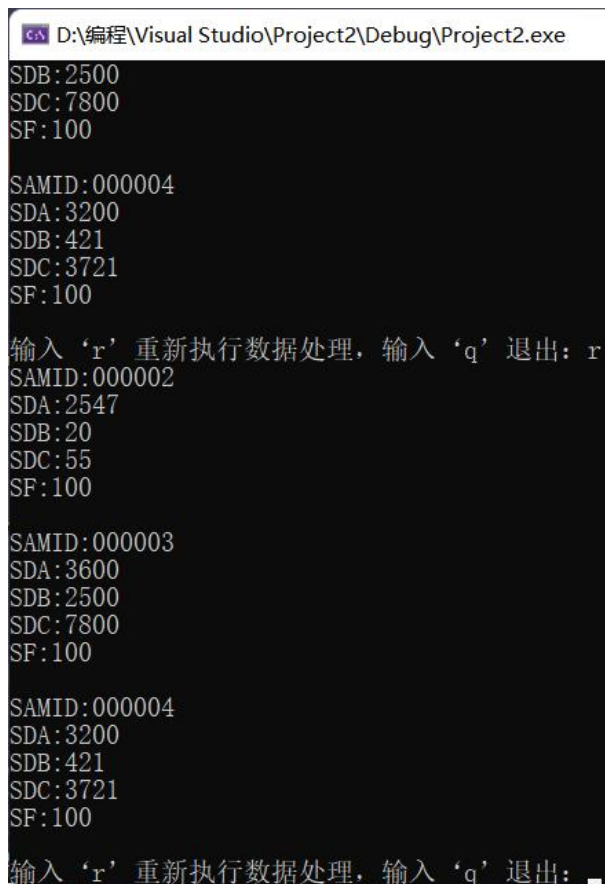
SAMID:000003
SDA:3600
SDB:2500
SDC:7800
SF:100

SAMID:000004
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: _
```

图 1.9 显示内容

之后, 根据提示, 按 “r” 键可以重新执行数据处理, 如图 1.10 所示。



```
D:\编程\Visual Studio\Project2\Debug\Project2.exe
SDB:2500
SDC:7800
SF:100

SAMID:000004
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: r
SAMID:000002
SDA:2547
SDB:20
SDC:55
SF:100

SAMID:000003
SDA:3600
SDB:2500
SDC:7800
SF:100

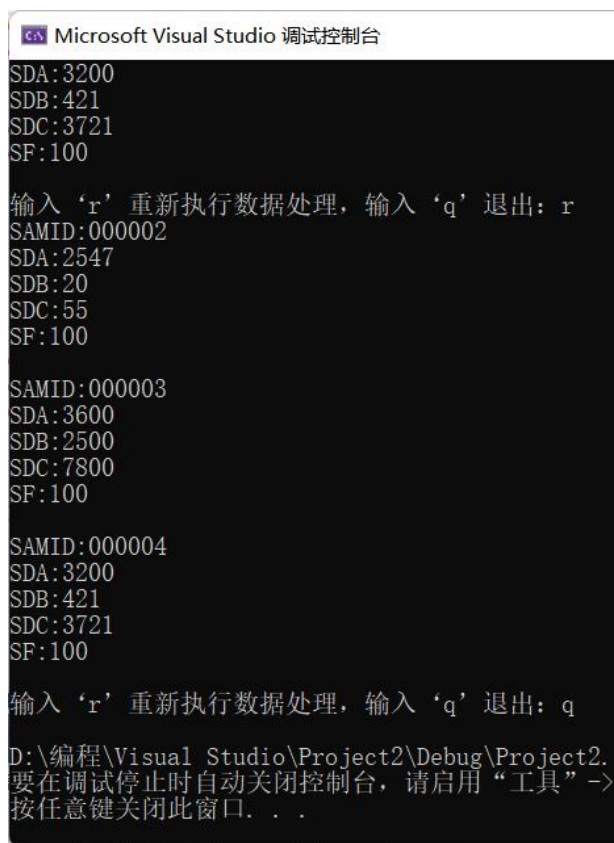
SAMID:000004
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: _
```

图 1.10 按 “r” 键以重新执行数据处理

按 “q” 键可以退出程序, 如图 1.11 所示。

汇编语言程序设计实验报告



```
Microsoft Visual Studio 调试控制台
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: r
SAMID:000002
SDA:2547
SDB:20
SDC:55
SF:100

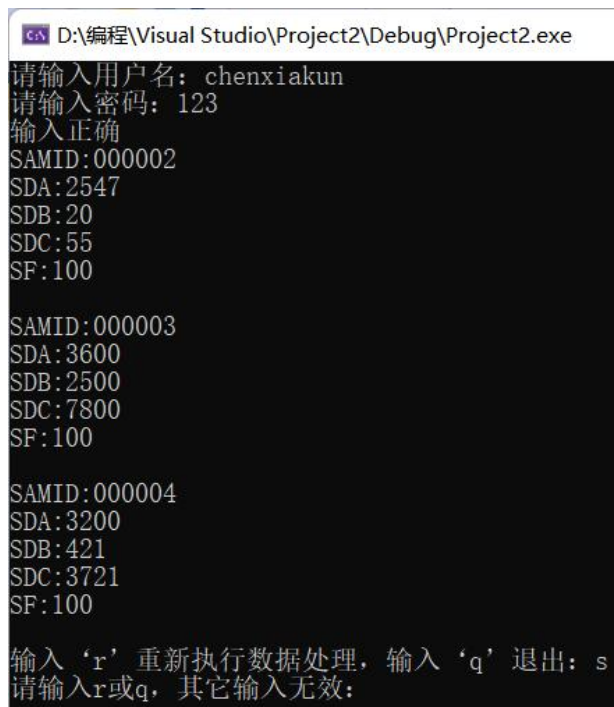
SAMID:000003
SDA:3600
SDB:2500
SDC:7800
SF:100

SAMID:000004
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: q
D:\编程\Visual Studio\Project2\Debug\Project2.exe 要在调试停止时自动关闭控制台, 请启用 "工具" -> 按任意键关闭此窗口. . .
```

图 1.11 按“q”键以退出程序

另外，在等待输入状态下，如果用户输入了“r”“q”以外的内容，程序也会进行相关提示，如图 1.12 所示。



```
D:\编程\Visual Studio\Project2\Debug\Project2.exe
请输入用户名: chenxiakun
请输入密码: 123
输入正确
SAMID:000002
SDA:2547
SDB:20
SDC:55
SF:100

SAMID:000003
SDA:3600
SDB:2500
SDC:7800
SF:100

SAMID:000004
SDA:3200
SDB:421
SDC:3721
SF:100

输入 'r' 重新执行数据处理, 输入 'q' 退出: s
请输入r或q, 其它输入无效:
```

图 1.12 提示内容

程序运行过程中，首先会遇到调用宏的过程，其对应反汇编代码如图 1.13 所示。

```
string_compare username, username_input, len_username, code, code_input, len_code)
00FB81BE push    edx
00FB81BF push    eax
00FB81C0 mov     edx, dword ptr [len_username (01028014h)]
00FB81C6 cmp     byte ptr [edx+102801Fh], 0
00FB81CD je      REINPUT+0C1h (0FB8237h)
00FB81CF cmp     byte ptr username_input (01028020h)[edx], 0
00FB81D6 jne     REINPUT+0C1h (0FB8237h)
LO:
00FB81D8 cmp     edx, 0
00FB81DB je      REINPUT+78h (0FB81EEh)
00FB81DD dec     edx
00FB81DE mov     al, byte ptr username (01028009h)[edx]
00FB81E4 cmp     al, byte ptr username_input (01028020h)[edx]
00FB81EA jne     REINPUT+0C1h (0FB8237h)
00FB81EC jmp     REINPUT+62h (0FB81D8h)
OK:
00FB81EE mov     edx, dword ptr [len_code (0102801Ch)]
00FB81F4 cmp     byte ptr [edx+102802Bh], 0
00FB81FB je      REINPUT+0C1h (0FB8237h)
00FB81FD cmp     byte ptr code_input (0102802Ch)[edx], 0
```

之后，程序会运行到用 `call` 指令调用 `cal_f` 函数这一句，`call` 指令首先将断点地址保存到堆栈的栈顶，然后将函数所在地址传给 `eip` 寄存器，实现程序的跳转。如图 1.14 至图 1.16 所示。

之后，程序会运行到用 `call` 指令调用 `cal_f` 函数这一句，`call` 指令首先将断点地址保存到堆栈顶，然后将函数所在地址传给 `eip` 寄存器，实现程序的跳转。如图 1.14 至图 1.16 所示。

```

    call cal_f ;计算f
00FB828F call    _cal_f@0 (0FB1F8Ch)
    mov esi, buf_point
00FB8294 mov     esi, dword ptr [buf_point (01028108h)]
    mov edx, dword ptr data_buf[esi + 18]

```

寄存器

EAX = 00000001 EBX = 00BD6000 ECX = 00000005 EDX = 00000003 ESI = 0133B480 EDI = 0133DCA0 EIP = 00FB3624 ESP = 00CFF8CC
EBP = 00CFF8DC EFL = 00000246

CS = 0023 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

内存 3				
地址:	0x00CFF8CC	列: 自动		
0x00CFF8CC	94 82 fb 00 00 00 00 00 00 00 00 00 00 00 00 00 00 24 f9 cf 00 a1 88 fb 00 01	???.?????.\$??.\$??.		
0x00CFF8E5	00 00 00 80 b4 33 01 a0 dc 33 01 44 73 bb b6 67 44 fb 00 67 44 fb 00 00 60	...€?3.??3.Ds??gD?.gD?..		
0x00CFF8FE	bd 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 f8 cf 00 00 00 00 7c f9 cf	?.....???.???..??		
0x00CFF917	00 e0 b3 fb 00 d8 e7 76 b7 00 00 00 00 00 34 f9 cf 00 d9 6b 33 75 00 60 bd 00	.???.??v?....4??7.7k3u.?.		
0x00CFF930	00 6b 33 75 8c f9 cf 00 d2 8f 2f 77 00 60 bd 00 9b f9 4f 56 00 00 00 00 00	?k3u???.??/w.?.??OV....		
0x00CFF949	00 00 00 00 60 bd 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00?		
0x00CFF962	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 f9 cf 00 00 00 00@???		
0x00CFF97B	00 94 f9 cf 00 10 d2 30 77 87 f2 b9 21 00 00 00 00 9c f9 cf 00 9d 8f 2f 77	.???.?0w???.???..???.??/w		
0x00CFF994	ff ff ff ff 19 91 32 77 00 00 00 00 00 00 00 00 67 44 fb 00 60 bd 00 00?2w.....gD?.?.		

可以看出，压入堆栈栈顶的地址实际上是使用 `call` 指令的下一句，使得从子函数返回后从使用 `call` 的下一句开始执行。子函数最后使用 `ret` 指令时，必须要保证 `esp` 指向的地址与刚进入子函数时的一致。

可以看出，压入堆栈栈顶的地址实际上是使用 `call` 指令的下一句，使得从子函数返回后从使用 `call` 的下一句开始执行。子函数最后使用 `ret` 指令时，必须要保证 `esp` 指向的地址与刚进入子函数时的一致。

汇编语言程序设计实验报告

之后,程序会运行到数据复制模块,对于第一组数据,将会使用 invoke 指令调用 copy_to_HIGHF 函数,此函数有一个参数(形参为 point,实参为 highf_point),对应的反汇编语句如下图所示。

```
invoke copy_to_HIGHF, highf_point
00FB82B4 push     dword ptr [highf_point]
00FB82B7 call     _copy_to_HIGHF@4 (0FB4115h)
mov highf_point, ebx
00FB82BC mov     dword ptr [highf_point], ebx
```

图 1.17 invoke 调用函数对应的反汇编语句

程序先将实参 highf_point 压入栈,然后再调用 call 指令,断点与之前的情况一样,设置在调用函数的后一句。

子函数中首先将 ebp 压栈,然后将栈顶指针的内容存储到 ebp 中,若在子函数中定义了局部变量,还会有一个 add 指令改变栈顶指针 esp 留出堆栈空间用于子程序内部定义的局部变量的存储(从观察调用 MIDF_point 函数的过程中可以得出),如图 1.18 所示。

```
copy_to_HIGHF proc point:dword
> 00FB8510 push     ebp      已用时间 <= 1ms
00FB8511 mov     ebp, esp
```

图 1.18 子函数开头的反汇编语句

在执行 ret 指令时,编译器首先执行 leave 语句,其功能相当于“mov esp, ebp”“pop ebp”还原堆栈的状态,之后的“ret 4”清除参数所占用的堆栈空间(形参 point 是 dword 类型,占了 4 个字节的堆栈空间)。如图 1.19 所示。

```
;mov point, ebx
ret
00FB8568 leave  ►
;mov point, ebx
ret
00FB8569 ret     4
```

图 1.19 ret 指令对应的反汇编语句

此外,在 show_MIDF 函数中,在不勾选“显示符号名”的情况下,还可以观察子函数中局部变量的访问形式,以局部变量 tem_point 和 tem_cnt 为例,如图 1.20 所示。

```
mov ebx, tem_point
00FB8381 mov     ebx, dword ptr [ebp-4]
PRINT_SAMID:
mov edx, tem_cnt
00FB8384 mov     edx, dword ptr [ebp-8]
```

图 1.20 子函数中局部变量的访问形式

可见,编译器以栈底指针 ebp 为基准,采用[ebp-n]这种变址寻址的方式访问子程序中的局部变量。

1.4 内容 1.2 的实验过程

1.4.1 指令优化对程序的影响

在任务 2.1 中,需要从指令角度优化任务 1.4 中的程序。我主要针对计算 f 过程中的乘除运算与复制到存储区的过程进行了优化,如下所述。

1. 除法运算优化

汇编语言程序设计实验报告

两种除法的汇编代码如图 1.21 所示。

```
;移位除法
sar eax, 7 ;除以128 (算术右移操作, 符号位保持不变)
];普通除法
;mov ecx, 128
;xor edx, edx ;清空高32位寄存器
;idiv ecx
```

图 1.21 两种除法的汇编代码

2. 复制过程优化

两种复制过程的汇编代码如图 1.22 所示。（以复制到 MIDF 存储区的代码为例）

```
;复制数据到MIDF
mov ebx, midf_point
;非循环复制 (多字节)
mov eax, 0
mov ax, word ptr input[esi]
mov word ptr MIDF[ebx], ax
mov eax, dword ptr input[esi + 2]
mov dword ptr MIDF[ebx + 2], eax
mov eax, dword ptr input[esi + 6]
mov dword ptr MIDF[ebx + 6], eax
mov eax, dword ptr input[esi + 10]
mov dword ptr MIDF[ebx + 10], eax
mov eax, dword ptr input[esi + 14]
mov dword ptr MIDF[ebx + 14], eax
mov eax, dword ptr input[esi + 18]
mov dword ptr MIDF[ebx + 18], eax
];循环复制 (单字节)
;mov ecx, 0 ;计数器
;COPY1:
;mov eax, 0
;mov al, byte ptr input[esi+ecx]
;mov byte ptr MIDF[ebx+ecx], al
;inc ecx
;cmp ecx, 22
;jl COPY1
```

图 1.22 两种复制过程的汇编代码

为便于观察到不同指令导致的程序运行时间差异，我设置了 200 组原始数据，并对这 200 组数据进行 200000 次计算 f 与复制操作（即执行计算 f 与复制操作共计 2 千万次），使用 C 语言中的 clock 函数进行计时，对以上 4 种可能的指令组合形式分别进行了 20 次测试，从结果来看，移位除法+非循环复制（多字节）的指令组合方式运行时间最短，效率最高，而普通除法+循环复制（单字节）的指令组合方式运行时间最长，效率最低。

得到以上结果后，我对乘法指令也进行了改进，如下所述。

3. 乘法指令优化

两种乘法的汇编代码如图 1.23 所示。

```
;普通乘法
imul eax, 5
];无符号乘法
;mul eax, 5
;移位乘法
;sal eax, 2
;add eax, eax
```

图 1.23 两种乘法的汇编代码

测试结果总表如图 1.24 所示。

汇编语言程序设计实验报告

循环复制（单字节）				非循环复制（多字节）						次数：2000 0000次	
普通除法	时间（ms）	移位除法	时间（ms）	普通除法	时间（ms）	移位除法	时间（ms）	移位除法+移位乘法	时间（ms）		
1	663	1	321	1	100	1	93	1	93		
2	672	2	317	2	98	2	95	2	94		
3	677	3	304	3	100	3	94	3	95		
4	664	4	326	4	103	4	96	4	95		
5	665	5	318	5	99	5	94	5	95		
6	662	6	320	6	99	6	94	6	94		
7	669	7	322	7	98	7	96	7	94		
8	665	8	338	8	98	8	91	8	94		
9	675	9	324	9	98	9	94	9	93		
10	675	10	314	10	99	10	96	10	93		
11	674	11	309	11	103	11	95	11	95		
12	662	12	311	12	98	12	94	12	94		
13	666	13	313	13	99	13	94	13	94		
14	670	14	319	14	105	14	94	14	94		
15	670	15	318	15	100	15	96	15	94		
16	665	16	318	16	100	16	97	16	94		
17	670	17	328	17	99	17	95	17	95		
18	674	18	321	18	99	18	94	18	93		
19	668	19	313	19	98	19	95	19	94		
20	669	20	308	20	99	20	95	20	94		
平均值	668.75	平均值	318.1	平均值	99.6	平均值	94.6	平均值	94.05		

图 1.24 测试结果总表

从表中可以看出，在将普通乘法改为移位乘法的方法后，程序运行时间有小幅减少，推测为：移位乘法确实起到了优化作用，但由于本任务中乘的数太小，因此优化不明显。

综上所述，在大批量数据下，指令优化对程序的影响还是比较大的。

1.4.2 约束条件、算法与程序结构的影响

1. 任务 1.3 中，需要考虑不同的字符串比较方法的差异。字比较相对于字节比较可以减少程序的比较次数，从而减少程序运行时间。但是，在字符串位数为奇数时，在比较最后一个字时，容易因为其第二个字节内容的不确定性造成误判。我考虑的解决方法有两种：一、使用字比较与字节比较结合的方式，若字符串位数为奇数，在比较到最后一个字时改用字节比较的方法；二、在为字符串开辟存储空间时，多开辟几个空间并放入相同的字符，保证最后一个字比较时第二个字节的内容一致。

另外，字节比较和字比较需要的寄存器与汇编指令也稍有不同，如图 1.25 所示。（代码中 key 是设置好的密码，array 是输入的密码，ecx 存储数组下标）

```

L:
    inc ecx
    ;inc ecx ;字比较多加一条
    dec edx ;计数器（循环次数）
    jz 0
    ;dec edx ;字比较多加一条
    ;jz 0 ;判断终止
    mov al, key[ecx] ;字节比较
    ;mov ax, word ptr key[ecx] ;字比较
    cmp al, array[ecx] ;字节比较
    ;cmp ax, word ptr array[ecx] ;字比较
    jz L ;相等则循环
    
```

图 1.25 字节比较和字比较差异

此外，若在串比较之前优先考虑两串长度是否相等，则可在串不相等的情况下直接终止程序，不进行比较，从而节省程序运行时间。

2. 任务 1.4 中，需要考虑数据溢出与有/无符号数的乘除法指令差异。如不对数据溢出做任何处理的话，溢出的数据可能写入一些不允许随意修改的内存从而导致程序报错，因此，在写程序时

汇编语言程序设计实验报告

一般都需要对溢出情况做一些特殊处理，如主动中止程序，拒绝写入内存等。

对于有/无符号数的乘除法指令差异，如图 1.26 所示。

```
imul eax, 5 ;有符号乘法      sar eax, 7 ;除以128 (算术右移操作, 符号位保持不变, 即有符号除法)
;mul eax, 5 ;无符号乘法      ;shr eax, 7 (逻辑右移操作, 当作无符号数除法)
```

图 1.26 有/无符号数的乘除法指令差异

其中，除法操作也可用 idiv（有符号）和 div（无符号）指令实现。

3. 在任务 1.4 中，数据的存储采用的是数组形式，如图 1.27 所示。

```
buf db '000001' ;占6个字节
dd 256809, -1023, 1265, 0
db '000002'
dd 112586, -1204, 1563, 0
db '000003'
dd 3586, -12256, 2451, 0
db '000004'
dd 2520, -600, -700, 0
;db '000005'
;dd 354687, 2002, 3569, 0
```

图 1.27 任务 1.4 的数据存储方式

在这种存储方式下，想要访问数据必须使用下标，当数据过多时可读性不强。但在任务 2.1 中，我采用结构体的形式存储数据，如图 1.28 和图 1.29 所示。

```
SAMPLES STRUCT
    SAMID DB 6 DUP(0) ;每组数据的流水号
    SDA DD 0 ;状态信息a
    SDB DD 0 ;状态信息b
    SDC DD 0 ;状态信息c
    SF DD 0 ;处理结果f
SAMPLES ENDS ;数据结构结束
```

图 1.28 结构体声明

```
input SAMPLES <'1', 321, 432, 10, ?>
SAMPLES <'2', 12654, 544, 342, ?>
SAMPLES <'3', 32100654, 432, 10, ?>
SAMPLES 197 dup(<>)
```

图 1.29 结构体数组 input 定义

相应地，HIGHF、MIDF、LOWF 三个存储数组也应该定义为结构体数组，因此，在访问其中内容是可以如图 1.30 所示的方式访问，可以使数据的访问更加清晰。

```
invoke printf, offset sda, MIDF[ebx].SDA
invoke printf, offset sdb, MIDF[ebx].SDB
invoke printf, offset sdc, MIDF[ebx].SDC
invoke printf, offset sf, MIDF[ebx].SF
```

图 1.30 任务 3.1 中输出 SDA 至 SF

4. 任务 3.1 中，采用了模块化设计，将不同功能用宏、子函数等方式设计，使主程序的可读性更强，代码更具条理性。如图 1.31 和图 1.32 所示。

汇编语言程序设计实验报告

```
;实现字符串比较的宏定义
string_compare macro username, username_input, len_username, code, code_input, len_code ;str1是既定用户名/密码, len1是str1的长度
    push edx
    push eax
    mov edx, len_username
    cmp username_input[edx - 1], 0
    je INCORRECT
    cmp username_input[edx], 0
    jne INCORRECT
L0:
    cmp edx, 0
    je OK
    dec edx
    mov al, username[edx] ;字节比较
    cmp al, username_input[edx] ;字节比较
    jne INCORRECT
    jmp L0 ;相等则循环
OK: ;继续比较密码
    mov edx, len_code
    cmp code_input[edx - 1], 0
    je INCORRECT
    cmp code_input[edx], 0
    jne INCORRECT
L02:
    cmp edx, 0
    je OK2
```

图 1.31 实现字符串比较的宏定义（部分）

```
;子程序: 计算f
cal_f proc
    push esi
    push edx
    mov esi, buf_point
    mov edx, dword ptr data_buf[esi + 6]
    imul edx, 5
    add edx, dword ptr data_buf[esi + 10]
    sub edx, dword ptr data_buf[esi + 14]
    add edx, 100
    sar edx, 7 ;除以128 (算术右移操作, 符号位保持不变)
    mov dword ptr data_buf[esi + 18], edx
    pop edx
    pop esi
    ret
cal_f endp
```

图 1.32 实现计算 f 的子程序

5. 任务 3.2 采用了汇编语言与 C 语言混合编程。我将输入用户名密码模块、显示数据模块、等待输入模块分别用 C 语言子函数实现，进一步缩短了汇编文件中主程序的代码量，使条理更加清晰。在实现“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”这一功能时，我发现结构体在 C 语言和汇编语言中存储方式的差异将会带来很大问题。

在任务 3.1 中及之前的实验中，在汇编程序中只为流水号 SAMID 开辟了 6 个字节的空间，这对于汇编程序来讲是没有问题的，但是一旦在 C 语言中以“%s”的形式输出 SAMID 时，会由于该字符串没有结尾 0 而输出意想不到的内容。因此，需要在汇编程序中流水号定义时多开辟一些空间并赋 0 值，以保证 C 语言中流水号的输出正确。

在解决了上一个问题后，我又考虑到 C 语言的结构体存在内存对齐的情况，若 SAMID 在汇编中定义时开辟空间为 7 个字节，则 C 语言中 SAMID 实际占 8 个字节，在输入新数据时将会出现问题。因此，最终汇编语言的结构体声明如图 1.33 所示，C 语言的结构体声明如图 1.34 所示。

<pre>SAMPLES STRUCT SAMID DB 8 DUP(0) SDA DD 0 SDB DD 0 SDC DD 0 SF DD 0 SAMPLES ENDS</pre>	<pre>struct SAMPLES2 { char name[8]; int SDA, SDB, SDC, SF; };</pre>
---	--

图 1.33—图 1.34 汇编语言结构体声明（左）和 C 语言结构体声明（右）

汇编语言程序设计实验报告

1.4.3 编程环境的影响

在所有实验中，我们共使用了三种编程环境进行了编程练习，即 VS2019、DOSBOX、QEMU。

可以看出，VS2019 易于调试及运行 32 位及 64 位的汇编程序，并且具有很好的中断和异常处理程序；DOSBOX 易于调试 16 位的汇编程序，且可以很好地观察二进制代码；QEMU 这个环境对 gcc 编译更加灵活，可以将函数和主程序分开编写和编译，这样对调用函数非常的灵活方便。

1.5 小结

通过实验一，我开始真正熟悉起汇编程序的书写，也加深了对 C 语言与汇编语言的联系的理解。反汇编窗口、寄存器窗口等窗口对于调试程序观察数据提供了很大的帮助，有助于我理解各类数据在内存中的存储方式。同时，我掌握了在汇编语言中对有/无符号数的运算方法，加深了对四种不同寻址方式的理解，掌握了在汇编语言中如何调用 C 语言函数。

通过实验二，我掌握了在汇编语言中使用结构体的方法，同时针对乘除法方式、复制数据方式展开了不同指令的测试。在大批量数据下，不同指令所花费的时间差别将会非常显著，这要求我们在平时编写程序时就要注意掌握那些运行时间短的指令及方法。

通过实验三，我掌握了汇编语言的模块化设计和 C 语言与汇编语言混合编程的方法，掌握了在汇编语言中定义宏、使用宏的方法，进一步加深了对外部函数的声明和使用的理解。在任务 3.2 的调试过程中，我由于对 C 语言中结构体存在内存对齐现象这一情况不够了解，导致走了许多弯路，这个知识点今后我将会牢牢记住。另外，在不断的调试过程中，我更加清楚地了解到函数调用时所进行的一系列过程，也更加清楚函数形参、局部变量的存储方式。

综上所述，实验一到实验三的过程，是我不断认识汇编语言，加深对数据存储方式、函数调用过程的理解，以及明确汇编语言与 C 语言的差异的过程，实验过程令我受益匪浅。

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 中断处理程序

1. 设计思想与实验方法

使用 INT xxH 指令来调用相关中断处理程序。

在任务 4.1 中，我多次使用了 INT 21H 的功能调用，灵活使用 INT 21H 对应的 AH 的值对应的功能，以使得程序更加高效的进行。

任务 4.1 的主要任务是编写一个新的中断处理程序，接管 8 号时钟中断，以使得 DOSBOX 界面右上角可以显示实时时间，并且要求在退出程序之后，该时间显示功能保留（通过驻留退出程序来实现）。在每一次运行程序后，都需要判断新的中断处理程序是否安装，若安装则直接退出程序。

我判断新中断处理程序是否安装的方法，是每次运行程序后，使用 INT 21H 获取 8 号中断处理程序的地址作为 OLD08H，将程序中代码段新中断处理程序的入口地址作为 NEW08H，然后比较两个地址是否相同，若相同则说明新的程序已经安装。

2. 记录与分析

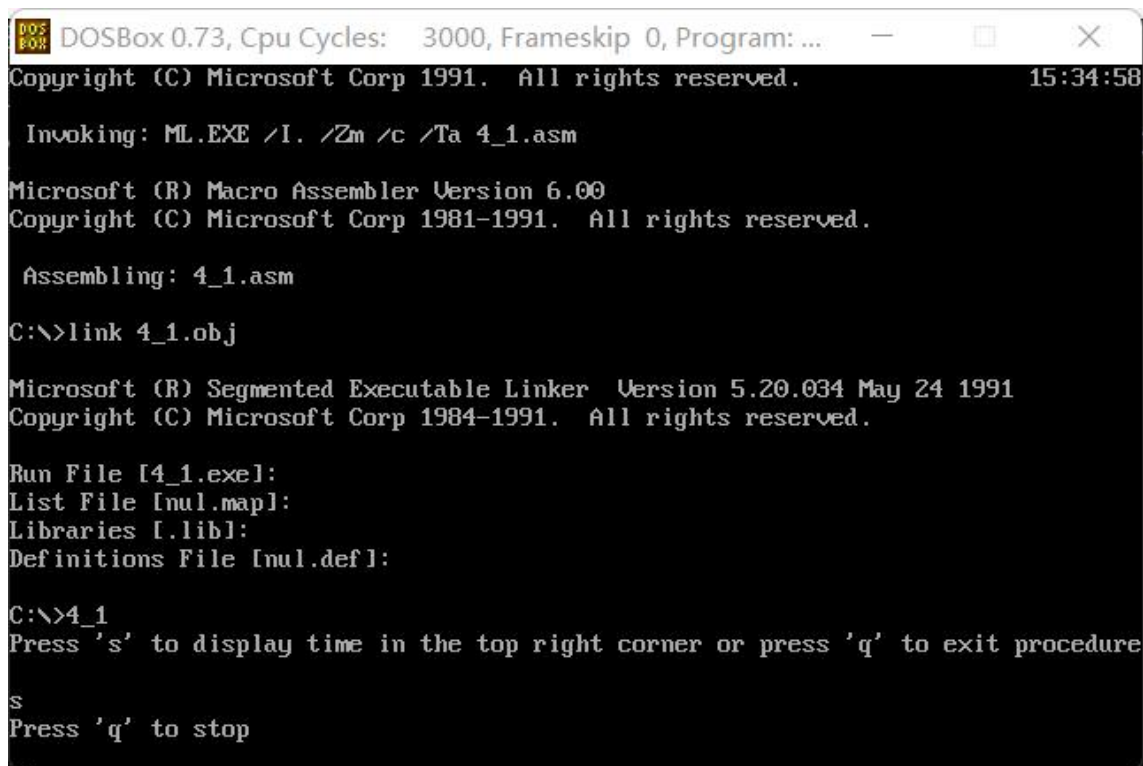
判断新程序是否安装及安装新程序的代码如图 2.1 所示。

```
MOV AX, 3508H
INT 21H      ;获取原08H的中断向量
MOV OLD_INT, BX ;保存中断向量
MOV OLD_INT+2, ES
MOV DX, OFFSET NEW08H
CMP DX, OLD_INT ;在不关DOSBOX的情况下，安装的新中断程序是不会变的
;只要已经存在的08H中断程序和NEW08H的地址相同，就是已经安装过了
JE HAVE_INSTALLED
MOV AX, 2508H
INT 21H      ;设置新的08H中断向量
```

图 2.1 判断新程序是否安装及安装新程序代码

汇编语言程序设计实验报告

在 DOSBOX 下执行程序后，界面右上角将显示时间，如图 2.2 所示。



```
DOSBox 0.73, Cpu Cycles: 3000, Frameskip 0, Program: ... 15:34:58
Copyright (C) Microsoft Corp 1991. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta 4_1.asm

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

Assembling: 4_1.asm

C:\>link 4_1.obj

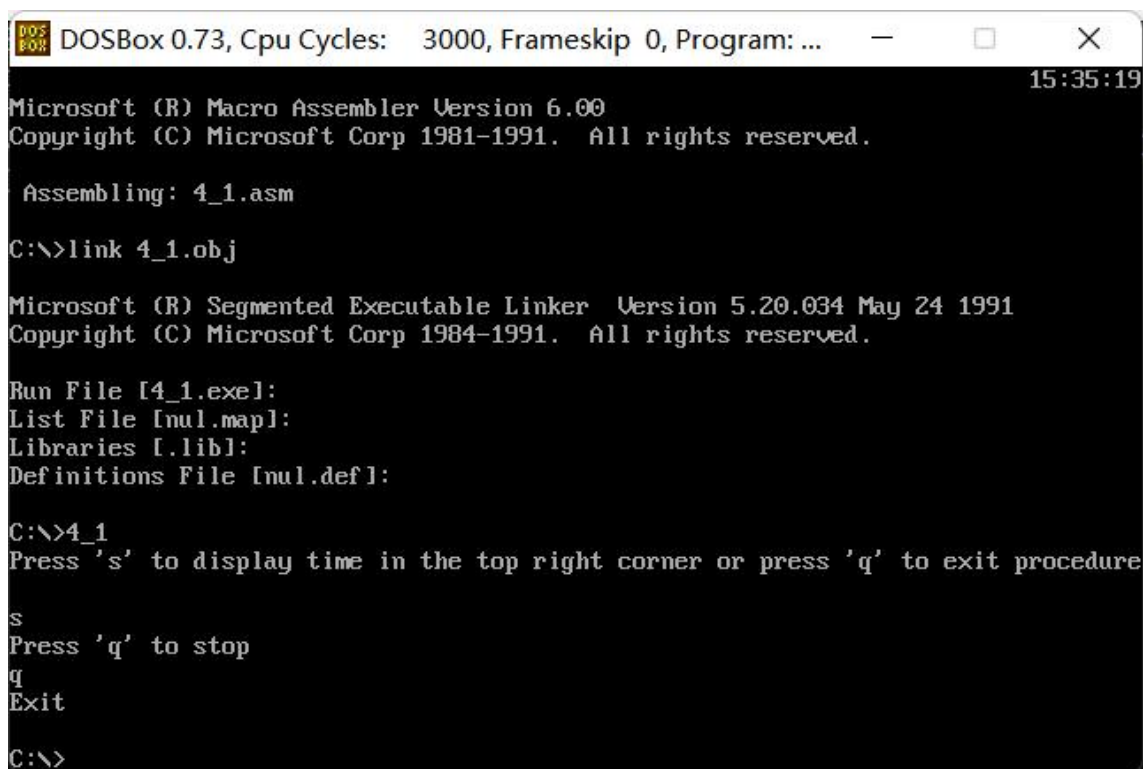
Microsoft (R) Segmented Executable Linker Version 5.20.034 May 24 1991
Copyright (C) Microsoft Corp 1984-1991. All rights reserved.

Run File [4_1.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:

C:\>4_1
Press 's' to display time in the top right corner or press 'q' to exit procedure
s
Press 'q' to stop
```

图 2.2 在 DOSBOX 下运行程序

退出程序后，中断处理程序将会驻留，即界面右上角仍然实时显示时间，如图 2.3 所示。



```
DOSBox 0.73, Cpu Cycles: 3000, Frameskip 0, Program: ... 15:35:19
Microsoft (R) Macro Assembler Version 6.00
Copyright (C) Microsoft Corp 1981-1991. All rights reserved.

Assembling: 4_1.asm

C:\>link 4_1.obj

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

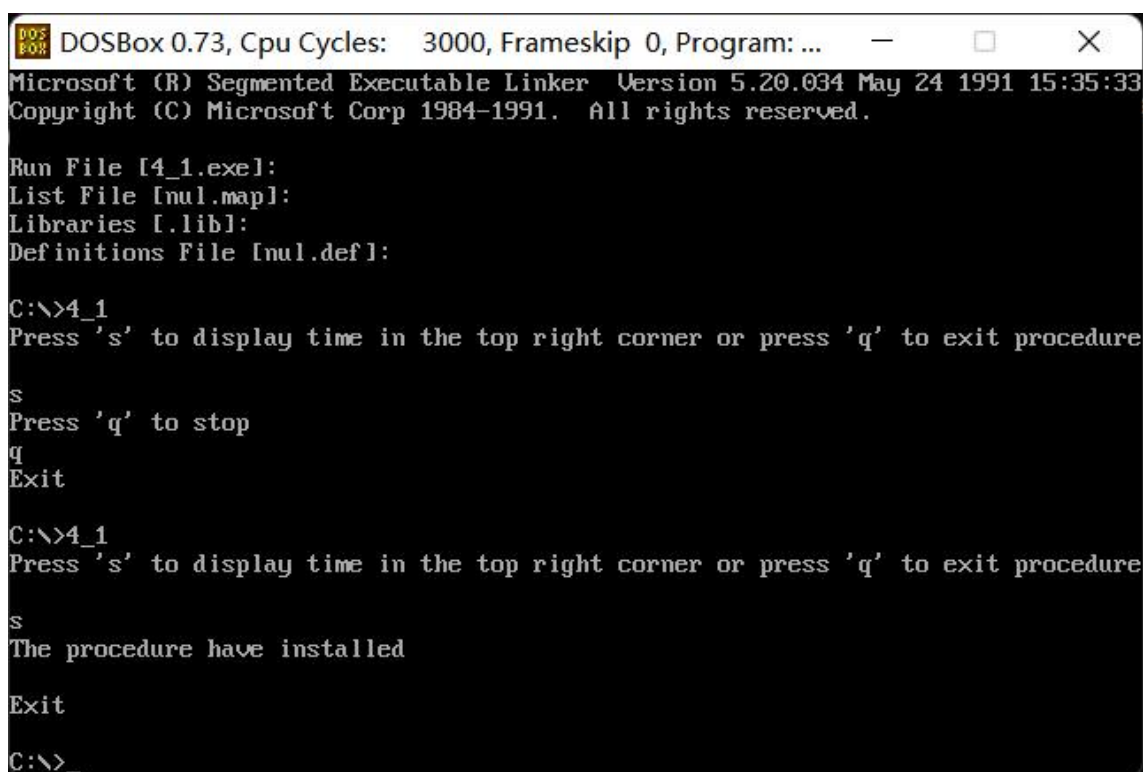
Run File [4_1.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:

C:\>4_1
Press 's' to display time in the top right corner or press 'q' to exit procedure
s
Press 'q' to stop
q
Exit
C:\>
```

图 2.3 中断处理程序驻留的效果

再次运行程序，会显示中断程序已安装的提示，并直接退出程序，如图 2.4 所示。

汇编语言程序设计实验报告



```
DOSBox 0.73, Cpu Cycles: 3000, Frameskip 0, Program: ...
Microsoft (R) Segmented Executable Linker Version 5.20.034 May 24 1991 15:35:33
Copyright (C) Microsoft Corp 1984-1991. All rights reserved.

Run File [4_1.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:

C:\>4_1
Press 's' to display time in the top right corner or press 'q' to exit procedure

s
Press 'q' to stop
q
Exit

C:\>4_1
Press 's' to display time in the top right corner or press 'q' to exit procedure

s
The procedure have installed

Exit

C:\>_
```

图 2.4 再次运行程序后的相关提示

2.3.2 反跟踪程序

1. 设计思想与实验方法

①密码加密

采用不公开的算法加密，对密码进行异或操作，使得保存或传输的数据是编码后的密文数据

②动态修改代码

对程序中需保密的计算 SF 的函数，我尝试采用动态修改代码的方案进行加密和反汇编，此举可使得别人在静态反汇编时无法正确获取到相关代码的值，即代码时动态生成的，必须执行过一段程序后才能把后续要执行的程序代码恢复出来，此方法可以有效地抵制静态汇编工具。但是由于未知原因，我的代码始终显示有内存问题，至今未解决。

③加入冗余代码

为了扰乱视线，可以在 JMP 或 CALL 指令之后定义一些初始值随意的变量存储区，由于在程序正常控制下是不会执行到这段存储区的，所以，反汇编程序把这段存储区的数据反汇编成指令语句是没有任何意义的，这就达到了扰乱视线的目的。

④间接转移/调用

无论是 JMP 指令还是 CALL 指令，都支持间接转移/调用，尤其是可以使用寄存器寻址方式实现间接转移/调用。我首先建立一张地址表，在整个程序一开始将计算 SF 函数的地址送入，当需要调用它时，将此地址表中的地址送入某个寄存器，然后再用 JMP 或 CALL 去转移，以此降低程序可读性，加大人工动态调试跟踪时理解程序的难度。

⑤通过检查堆栈来抵制动态调试跟踪

动态调试过程中会产生单步中断，该中断响应过程在没有跨越特权级时会使用被调试程序的堆

汇编语言程序设计实验报告

栈，也就是说被调试程序栈顶以上几个字存储区的内容会被中断相应过程修改。因此，在代码中某处先 push 一个变量，然后 pop 掉这个变量，正常运行过程中 pop 只改变栈顶指针的位置，而 pop 的变量实际仍然存在于 pop 之前的内存中，通过再取出那块内存的数据，与原变量比较，若不同则可以说明程序处在单步调试状态。

⑥通过计时的方法来抵制动态调试跟踪

人工单步调试时的指令执行间隔很容易达到秒级，而连续执行的程序中的指令执行间隔一般不会达到毫秒级，因此，通过计算指定的几条指令执行所花费的时间，就可以发现该程序是否被调试。在某些关键可以获取信息容易被设置成断点的地方进行计时可以有效的察觉跟踪。

2. 反跟踪效果的验证

①加密密码的相关代码如图 2.5 所示。

```
username db 'chenxiakun', 0
len_username dd $ - username - 1
code db 'a' xor 'A', 'b' xor 'B', 'c' xor 'C', 'd' xor 'D', 'e' xor 'E', 'f' xor 'F', 'g' xor 'G', 0
```

图 2.5 加密密码

②动态修改用到的变量如图 2.6 所示，相关代码如图 2.7 所示。

```
.DATA
MACHINE_CODE DB 8BH, 45H, 0FCH, 6BH, 0C0H, 05H, 03H, 45H, 0F8H, 83H, 0C0H, 64H, 2BH, 45H, 0F4H, 0C1H, 0F8H, 07H, 8BH, 4DH, 0F0H, 89H, 01H
LEN=$-MACHINE_CODE
OLDPROTECT DWORD ?
```

图 2.6 动态修改用到的变量

```
;动态修改代码
MOV EAX, LEN
MOV EBX, 40H
LEA ECX, COPE_HERE
INVOKE VirtualProtect, ECX, EAX, EBX, OFFSET OLDPROTECT
MOV ECX, LEN
MOV EDI, OFFSET COPE_HERE
MOV ESI, OFFSET MACHINE_CODE
COPY_CODE:
MOV AL, [ESI]
MOV [EDI], AL
INC ESI
INC EDI
LOOP COPY_CODE
COPE_HERE:
DB LEN DUP(0)

ret 16
```

图 2.7 动态修改相关代码（未调通）

③冗余代码如图 2.8 所示。

```
CONFUSE: ; 反跟踪方法一：冗余代码
dec ecx
cmp ecx, -1
jne CONFUSE
;冗余代码结束
```

图 2.8 冗余代码

④间接转移/调用的相关代码如图 2.9 所示。

```
mov eax, dword ptr address
call eax ; 反跟踪方法二：间接调用cal_f

;call cal_f ;计算f
```

图 2.9 间接转移/调用的相关代码

汇编语言程序设计实验报告

⑤检查堆栈的相关代码如图 2.10 所示。

```
    ;反跟踪方法三：通过检查堆栈来抵制动态调试跟踪
    cli      ;cli会报错？
    push PASS
    mov eax, 0
    mov ebx, 0
    pop eax
    mov ebx, [esp - 4] ;正常运行情况下，pop指令只
    jmp ebx
    sti
```

图 2.10 检查堆栈的相关代码

⑥计时反跟踪的相关代码如图 2.10 所示。

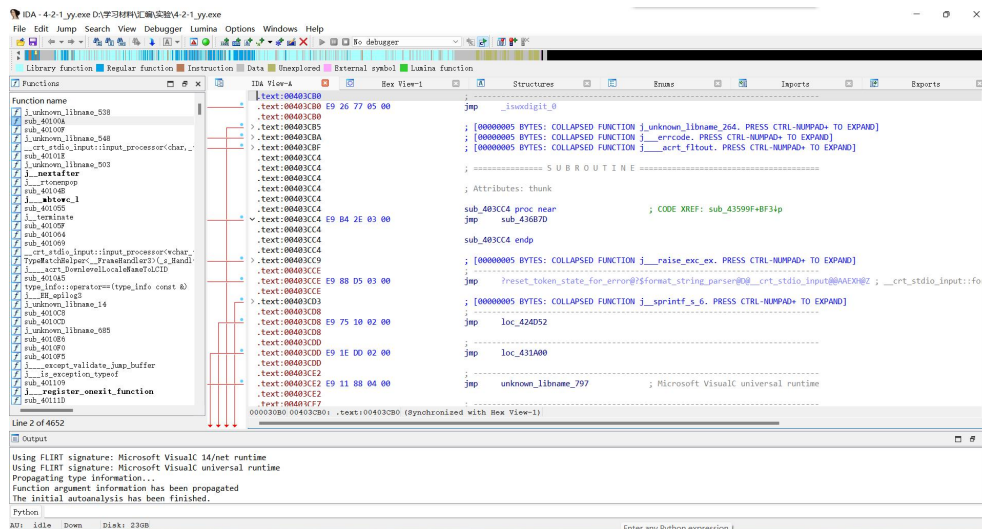
```
    invoke clock      ;反跟踪方法四：计时反跟踪
    mov start_time, eax
    ENCRYPTION:      ;加密输入的密码
    mov cl, 'A'
    mov eax, 0
    NEXT_LETTER:      ;加密下一个字母
    cmp code_input[edx], 0
    je COMPARE
    xor code_input[edx], cl
    inc cl
    inc eax
    jmp NEXT_LETTER
    invoke clock
    mov end_time, eax
    sub eax, start_time
    mov stay_time, eax
    cmp stay_time, 55
    jae NORMAL_ENDING
    cmp stay_time, -1 ;stay_time初始值设定为-1，如果这里还为-1，说明直接跳过了计算stay_time的步骤
    je NORMAL_ENDING
    ;加密结束
```

图 2.11 计时反跟踪相关代码

3. 跟踪与破解程序

任务 4.3 要求与同学组队，尝试破解对方的程序，本任务中与我组队的同学是杨阳，我使用以下技术进行了破解。

①利用静态反汇编工具 IDA 将执行程序反汇编成源程序并观察其特点。在 IDA 中，可以观察模块与模块之间的联系获取信息。如图 2.12 所示。



汇编语言程序设计实验报告

图 2.12 IDA 反汇编 exe 文件

②利用二进制文件编辑工具，这里我使用的是 VS2019，通过直接修改代码进行暴力破解。包括：直接观察和修改执行文件中的数据信息、直接修改执行程序的机器码等。二进制编辑界面如图 2.13 所示。

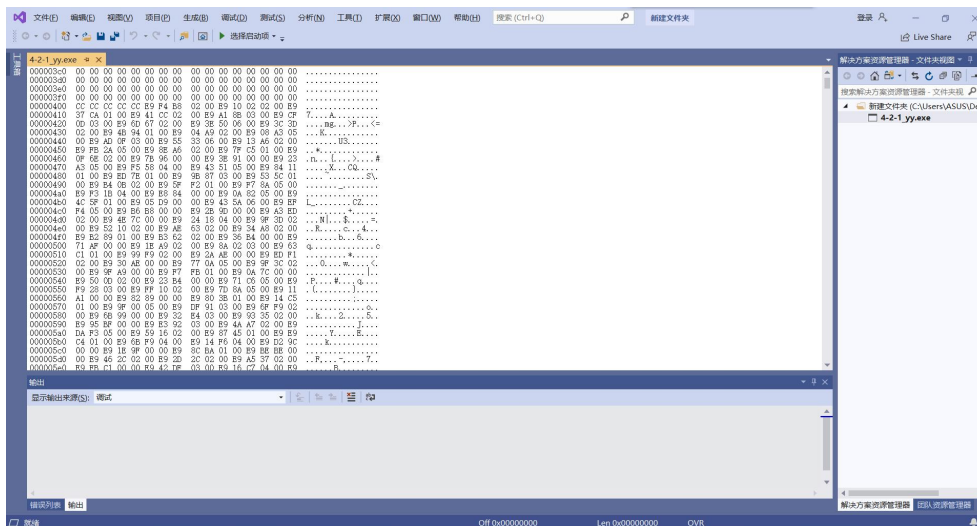


图 2.13 二进制文件编辑工具打开 exe 文件

③动态跟踪调试

可使用 VS2019（对可执行程序的调试能力极为有限）或 IDA，OLLYDBG 等工具进行动态跟踪调试。此部分需单步调试程序，并识别代码中反跟踪的部分以此跳过该部分防止自己的调试过程被识别。

在使用了以上跟踪与破解方法的基础上，30 分钟之内没有破解出密码，在与队友讨论并再次尝试之后破解成功。

2.3.3 指令优化及程序结构

1. 实验方法

①在 1.4 节中已经对实验 2、3 在指令优化及程序结构方面的修改方法进行了说明，这里不再重述。常见的优化思路有：选择执行较快的指令；减少循环体中的指令条数；灵活利用机器指令的特点、寻址方式的特点等。

②在实验 5.2 中，要求运行“ARM 虚拟环境安装说明”文档中“2.2.1”（一个测试内存拷贝函数的执行时间的 C 语言与汇编语言混合编程的程序）和“2.2.3”（对前面“2.2.1”程序的优化）的程序。由于内存存在连续读/写多个数据时，其性能要优于非连续读/写的数据的方式，因此在“2.2.3”这一程序中，使用了一次对多个字节进行读写的优化思路，使用了 ldp 指令和 stp 指令，使其可以一次访问 16 个字节的内存数据，大大提高了内存读写效率。

2. 特定指令及程序结构的效果

①在 1.4 节中已经对实验 2、3 在指令优化及程序结构方面修改前后的效果进行了说明，这里不再重述。需要强调的是，在不同模块中若要使用同一个变量，一定要保证变量定义或声明时的一致性，在使用 C 语言和汇编语言混合编程时，还要注意两种语言默认规定的差异（如 C 语言结构体有内存对齐现象而汇编语言结构体没有）。

汇编语言程序设计实验报告

②在实验 5.2 中优化前后的程序运行结果分别如图 2.14 和图 2.15 所示。

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memorycopy time is 203765596 ns
```

图 2.14 优化前的运行结果

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
memorycopy time is 61430227 ns
```

图 2.15 优化后的运行结果

可以看出，一次对 16 字节读写的程序执行效率明显优于单字节读写。

2.4 小结

这一部分的实验内容为编写和使用中断处理程序以及加密、跟踪和反跟踪。

通过实验四，我编写了新的中断处理程序，通过 IO 指令的使用从 CMOS 芯片中读取当前时间的时、分、秒信息，了解到在实方式下，系统定时器被初始化成每秒产生 18.2 次中断，我选择通过计数 18 次的方式来判断是否需要执行显示时间的程序。在这个实验中我对中断处理程序的编写、如何替换原有中断处理程序、如何使用 INT 指令来调用中断处理程序等都有了更为深刻的认识。

在任务 4.2 及 4.3 中，我了解到了不同的加密方法以及反跟踪方法，实验中的加密方法较为简单，而反跟踪方法包括添加冗余代码、计时、检查堆栈、检查中断向量表、动态修改执行代码等等，每一项的增加都为后面尝试破解程序增加了不小的难度。在跟踪破解程序这一环节，我既是进攻方，也防守方，更加体会到了信息安全的重要性，也激发了我对破解程序的兴趣。

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

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境(IDE)等等。本课程实验部分内容使用 VS2019 完成。

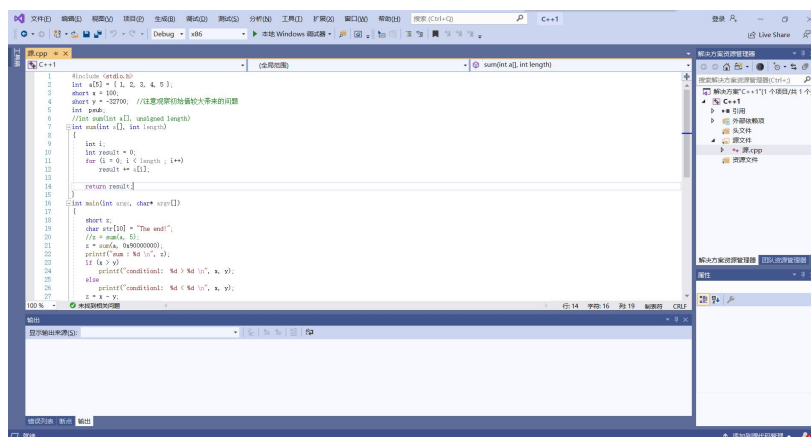


图 3.1 VS2019 主界面

在任务 1.1 中，使用到了 VS2019 中的多种窗口，如反汇编窗口（图 3.2）、寄存器窗口（图 3.3）、监视窗口（图 3.4）、内存窗口（图 3.5）

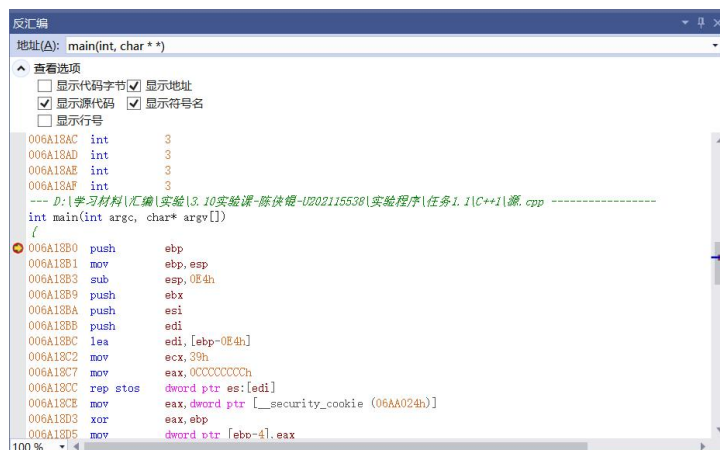


图 3.2 反汇编窗口

汇编语言程序设计实验报告

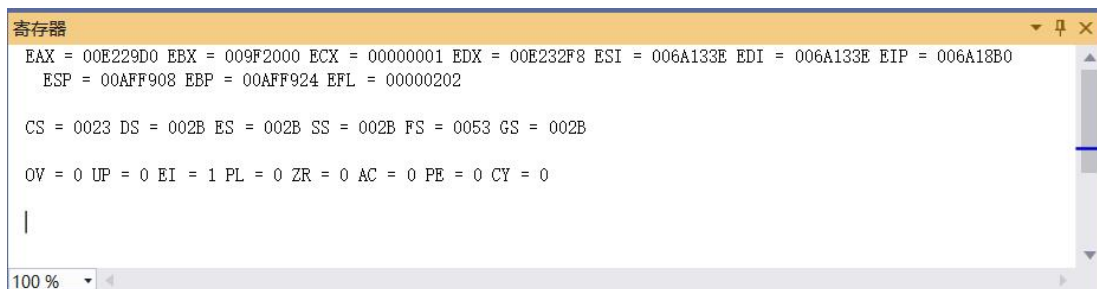


图 3.3 寄存器窗口

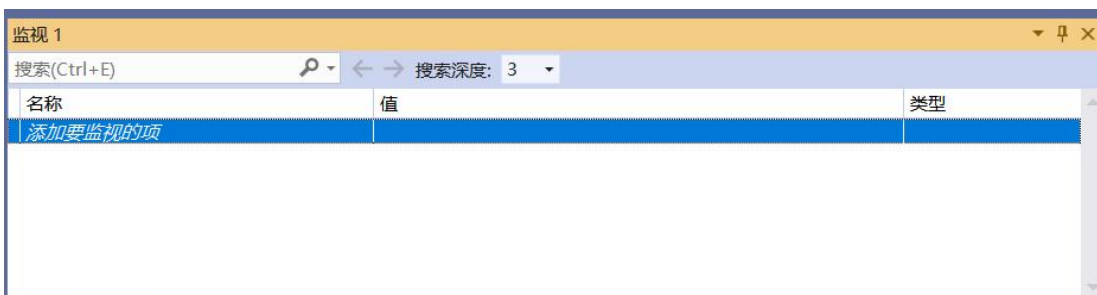


图 3.4 监视窗口

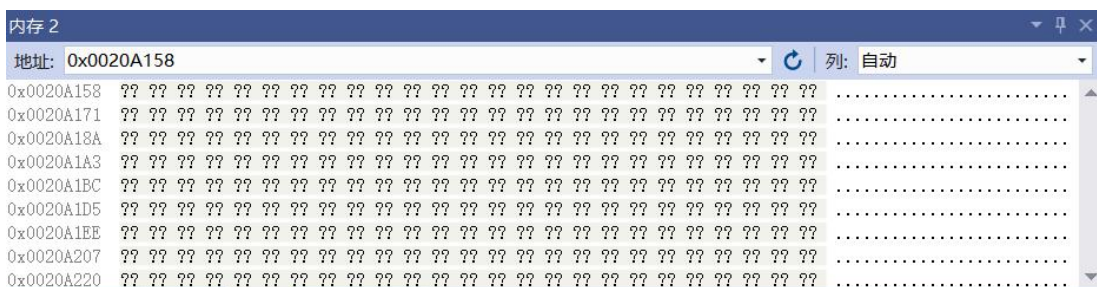


图 3.5 内存窗口

其中，反汇编窗口可以用于观察编译器生成的反汇编代码，其不显示符号名的情况如图 3.6 所示。

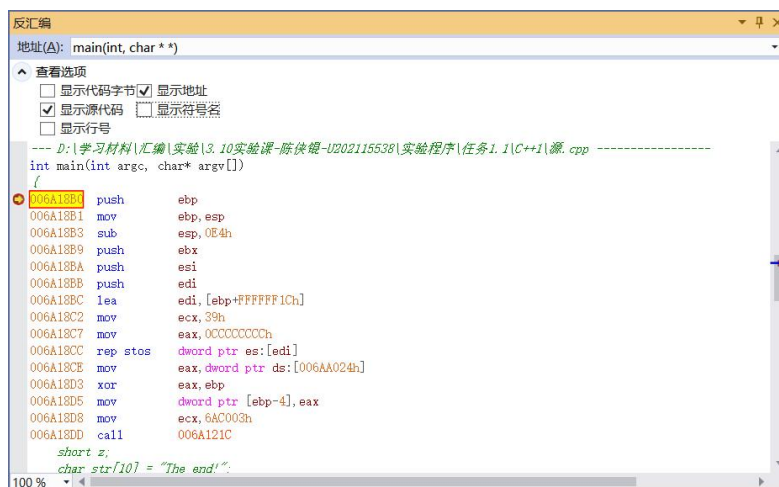


图 3.6 反汇编窗口（不显示符号名）

可以看出，在不显示符号名的情况下，所有符号将会被替换为地址，令代码可读性变差，不利于调试过程的观察。因此在调试过程中，建议勾选“显示符号名”这一项。

汇编语言程序设计实验报告

此外，寄存器窗口可用于观察通用寄存器、段寄存器和标志寄存器中的值；监视窗口可在调试过程中随时观察具体变量（或变量地址）的值；内存窗口在输入正确内存地址之后，可以看到该地址中存储的值（图 3.7），如内存地址不正确，多数情况下该地址中的值将以“？”代替（见图 3.5）

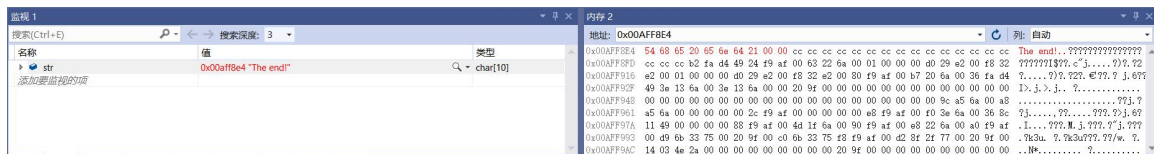


图 3.7 变量 str 的地址及地址中存储的数据

任务 1.1（4）中，要求对有符号和无符号变量的存储方式进行观察，如图 3.8 所示，两种变量的存储方式是相同的。

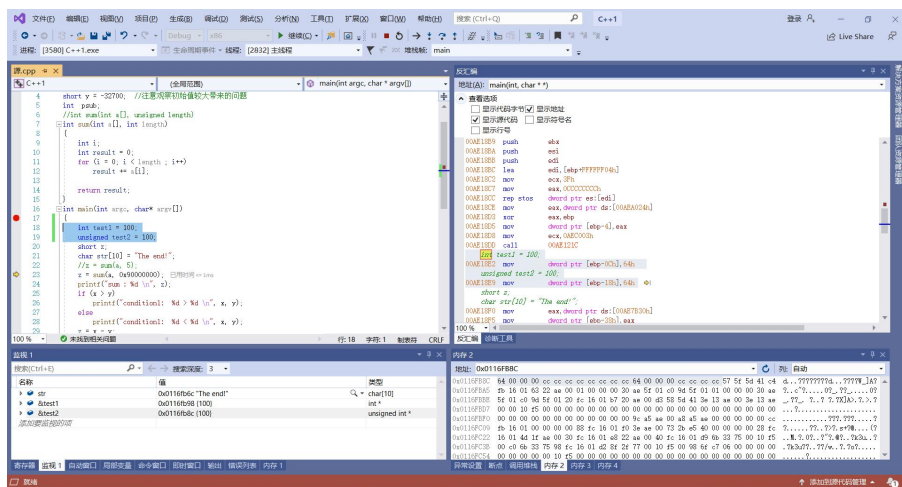


图 3.8 有符号和无符号变量的存储方式比较

任务 1.1（5）中，要求对有符号和无符号变量的运算进行观察比较，C 语言编写如图 3.9 所示，观察反汇编情况如图 3.10 所示，标志寄存器情况如图 3.11 所示。

```
int main(int argc, char* argv[])
{
    int t = 0, tt = 0;
    unsigned int p = 0, pp = 0;
    int ttt = t + tt;
    unsigned int ppp = p + pp;

    if (t > 0) ttt = 0;
    if (p > 0) ppp = 0;
}
```

图 3.9 有符号和无符号变量的运算比较（C 语言代码）



图 3.10 有符号和无符号变量的运算比较（反汇编观察）

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

图 3.11 标志寄存器情况

汇编语言程序设计实验报告

可以看出，两者的加法运算没有区别，都是先将第一个加数放入寄存器中，然后用 add 指令将寄存器中的值与第二个加数相加，再通过 mov 指令把结果赋值给变量。而二者的比较过程中，跳转指令会有所不同。

任务 1.1 (6) 的观察结果如图 3.12 至 3.14 所示。

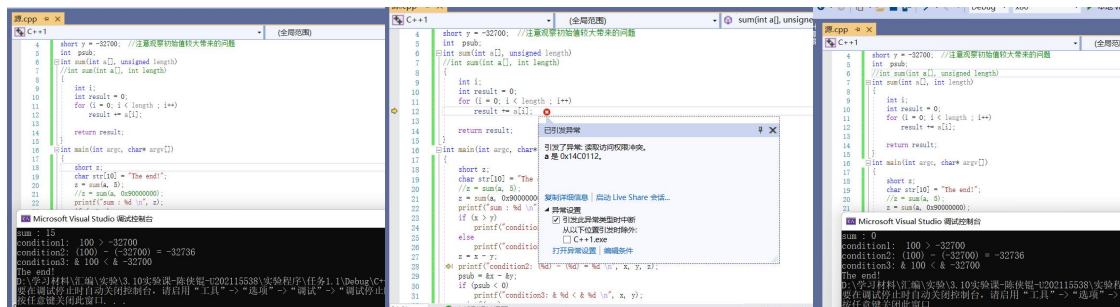


图 3.12-图 3.14 任务 1.1 (6) 观察结果

- ①在 $z = \text{sum}(a, 5)$ ，使用 unsigned length 时，程序会给出“有符号/无符号不匹配”的警告信息，但可以正常运行；
- ②在 $z = \text{sum}(a, 0x90000000)$ ，使用 unsigned length 时，由于循环次数过大导致数组溢出；
- ③在 $z = \text{sum}(a, 0x90000000)$ ，使用 int length 时，0x90000000 被解释为负数，程序可以正常运行，但 sum 函数中的循环不执行。

此外，对情况①中反汇编窗口观察可以得知，当有符号/无符号不匹配时，比较采用无符号比较（图中为 jae 指令）

```
for (i = 0; i < length; i++)
00B11DFF mov     dword ptr [ebp-8], 0
00B11E06 jmp     00B11E11
00B11E08 mov     eax, dword ptr [ebp-8]
00B11E0B add     eax, 1
00B11E0E mov     dword ptr [ebp-8], eax
for (i = 0; i < length; i++)
00B11E11 mov     eax, dword ptr [ebp-8]
00B11E14 cmp     eax, dword ptr [ebp+0Ch]
00B11E17 jae     00B11E2A
```

图 3.15 比较指令的反汇编观察

任务 1.2 中，将 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，如图 3.16 所示。

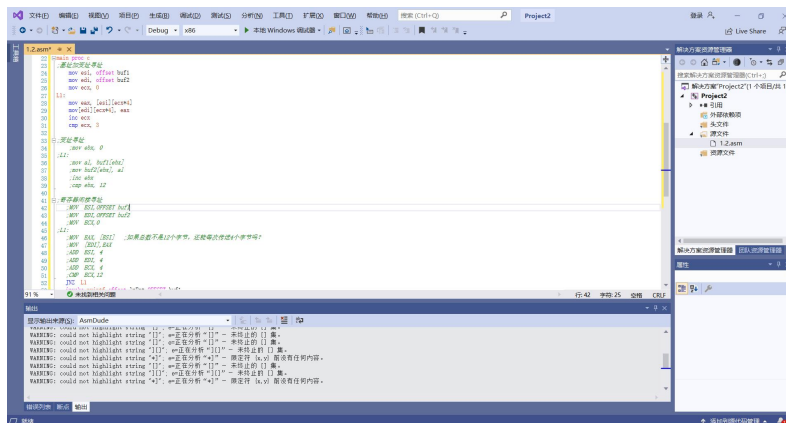


图 3.16 改写 buf1 的寻址方式

此外，在任务 1.2 中，我了解到了字符串在数据段定义的正确方法，了解到 C 语言函数如 printf 在汇编语言中的声明和调用，加深了对于汇编语言中方括号内“下标”的认识，其表示第几个字节而非像 C 语言一样表示第几个元素，若此“下标”不慎用错，程序更可能输出错误结果而非报错。

汇编语言程序设计实验报告

3.2.2 DOSBOX 下的工具包

DOSBox 是一个软件。它是当前在 Windows、Linux, macOS, Android 系统运行 DOS 游戏的较为完美的解决方案。由于它采用的是 SDL 库, 所以可以很方便的移植到其他的平台。其主界面如图 3.17 所示。

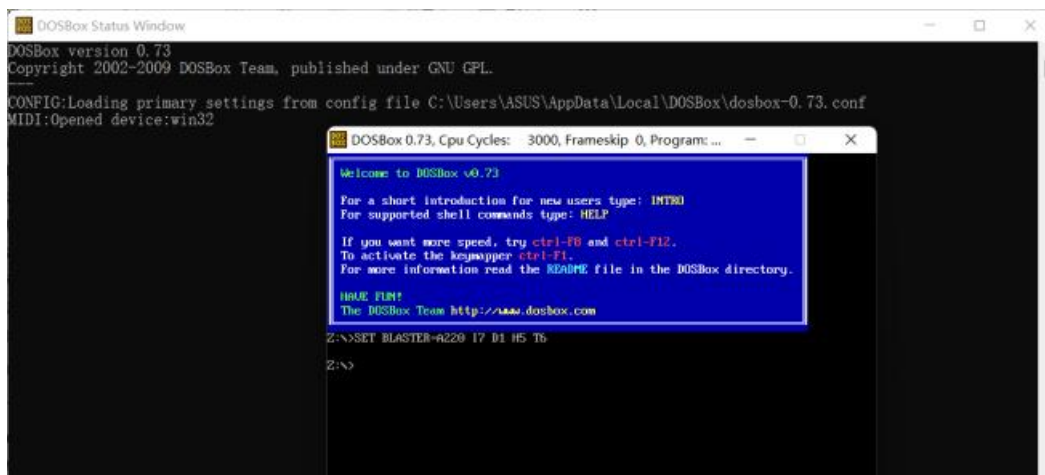


图 3.17 DOSBOX 主界面

在 DOSBOX 环境下, 想要编译程序必须先使用 mount 指令将存储汇编文件的文件夹挂载为 C 盘, 进入此“C 盘”之后, 先使用 masm 编译汇编文件, 再使用 link 链接 obj 文件, 最后才可以运行, 如图 3.18 所示。

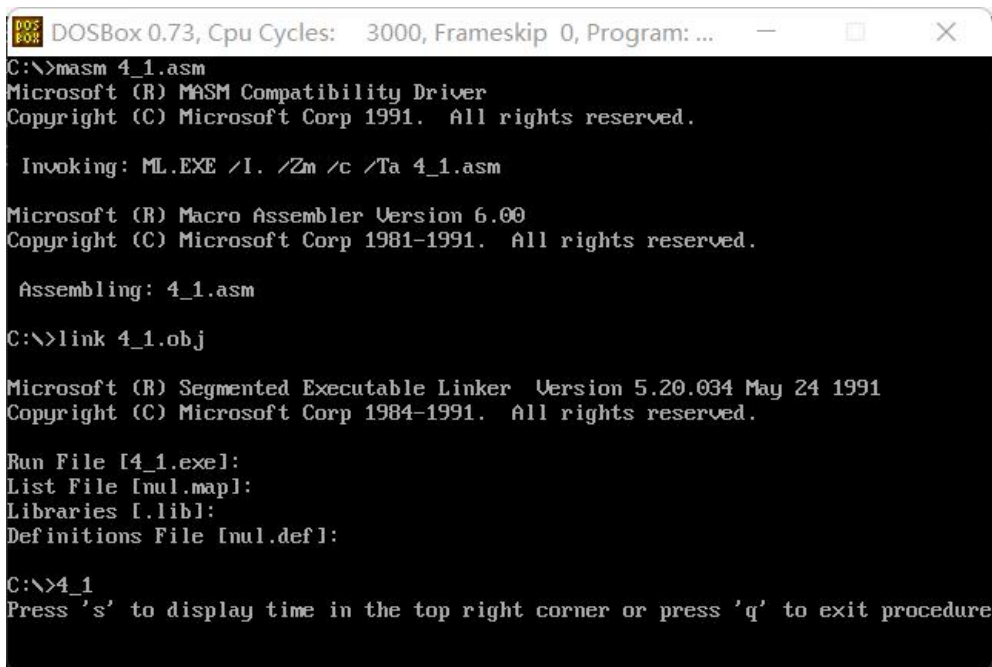


图 3.18 DOSBOX 环境下编译并运行汇编程序

此外, 我们还可以使用 td 工具进行调试, 在 td 中, 我们可以观察中断矢量表, 从而更好的对中断处理程序进行观察。td 界面中, 程序的代码段、数据段、栈、寄存器将分别在左上角、左下角、右下角、右上角显示出来, 如图 3.19 所示。

汇编语言程序设计实验报告

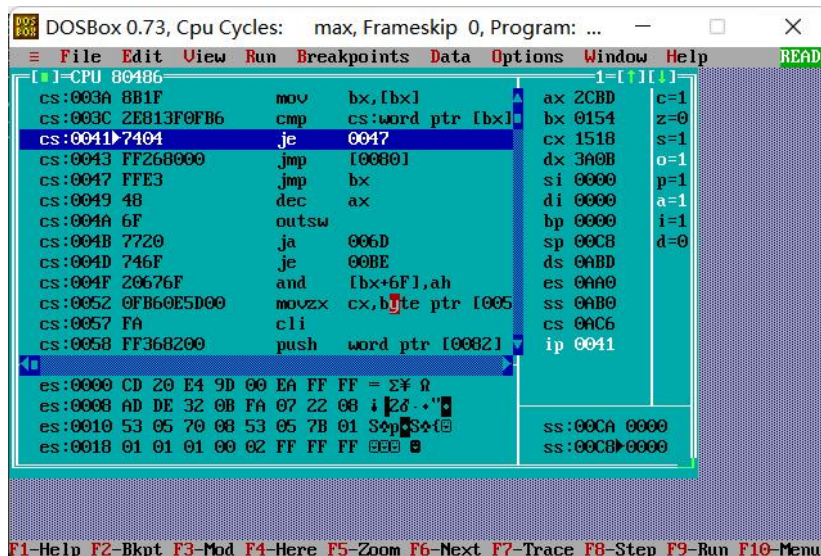


图 3.19 使用 td 调试汇编程序

3.2.3 QEMU 下 ARMv8 的工具包

QEMU 是一套由法布里斯·贝拉(Fabrice Bellard)所编写的以 GPL 许可证分发源码的模拟处理器软件，在 GNU/Linux 平台上使用广泛。在 View 菜单将串口改为 Serial0 后，界面如图 3.20 所示。

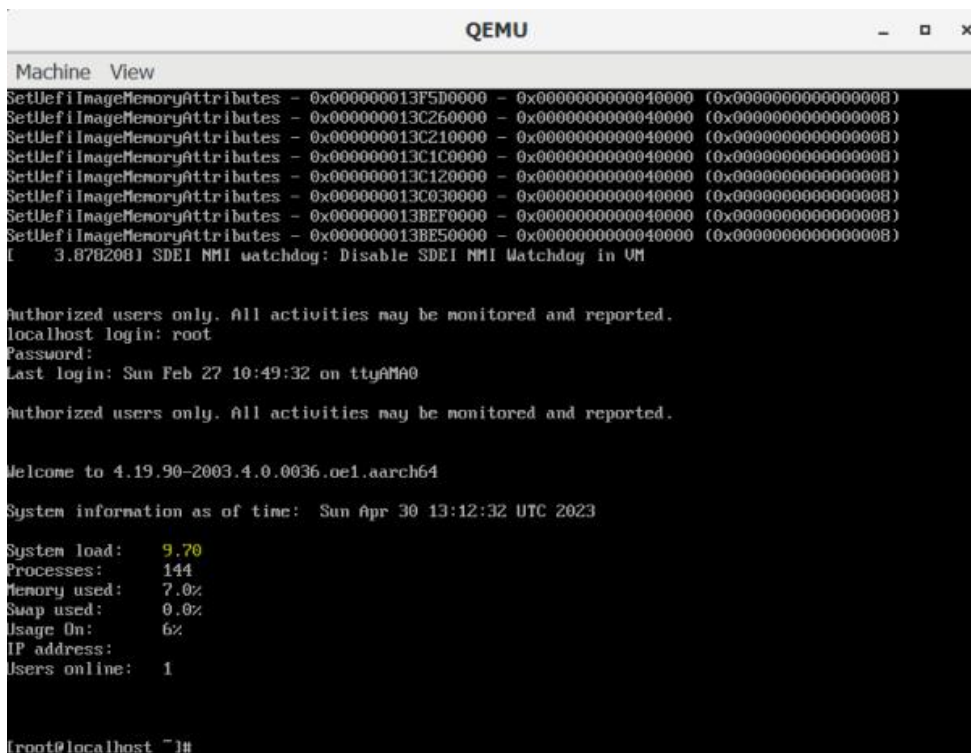
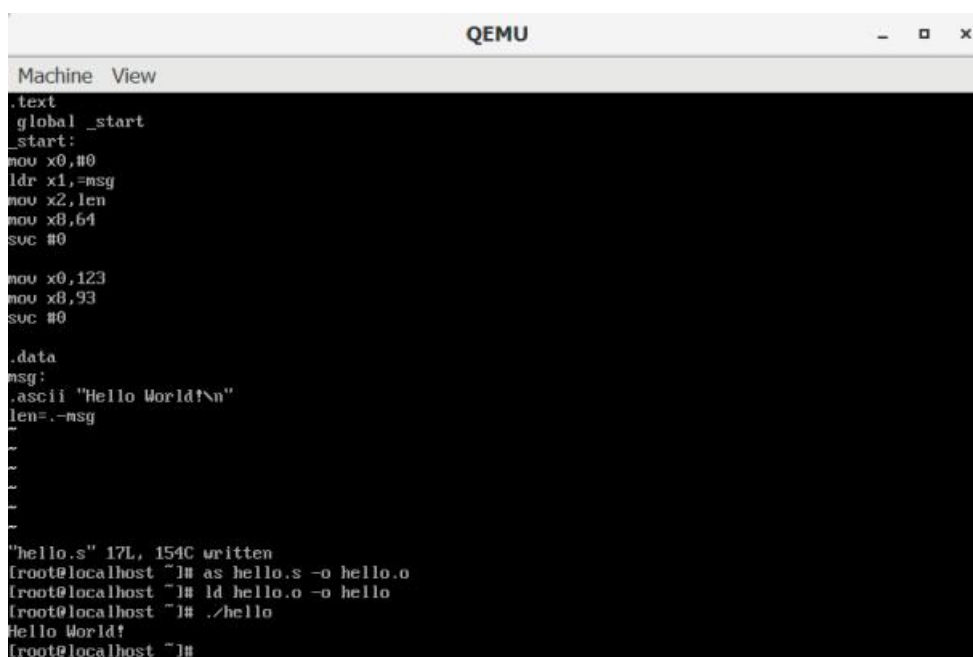


图 3.20 QEMU 界面（串口 Serial0）

对于纯汇编语言编写的程序，我们可以使用 as 指令进行编译，再使用 ld 指令进行链接，最后执行程序，如图 3.21 所示。

汇编语言程序设计实验报告



```
QEMU
Machine View
.text
.global _start
_start:
mov x0,#0
ldr x1,msg
mov x2,len
mov x8,64
svc #0

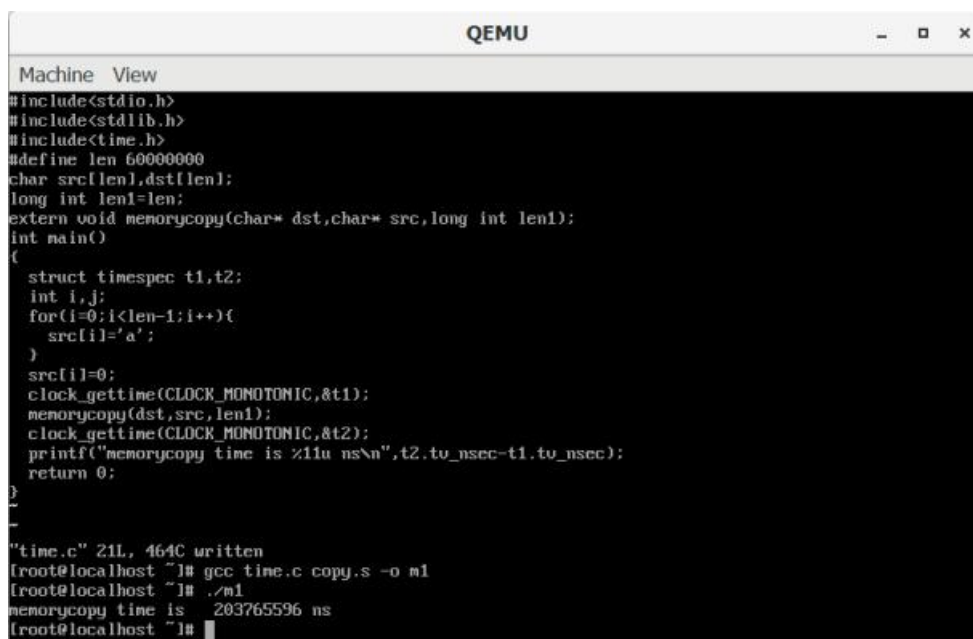
mov x0,123
mov x8,93
svc #0

.data
msg:
.ascii "Hello World!\n"
len=.-msg

"hello.s" 17L, 154C written
[root@localhost ~]# as hello.s -o hello.o
[root@localhost ~]# ld hello.o -o hello
[root@localhost ~]# ./hello
Hello World!
[root@localhost ~]#
```

图 3.21 编写、编译、链接纯汇编程序

对于 C 语言与汇编语言混合编程的程序，可使用 gcc 指令进行编译，并且其可以主动选择调用的函数文件，使得调用函数更加方便，如图 3.22 所示。



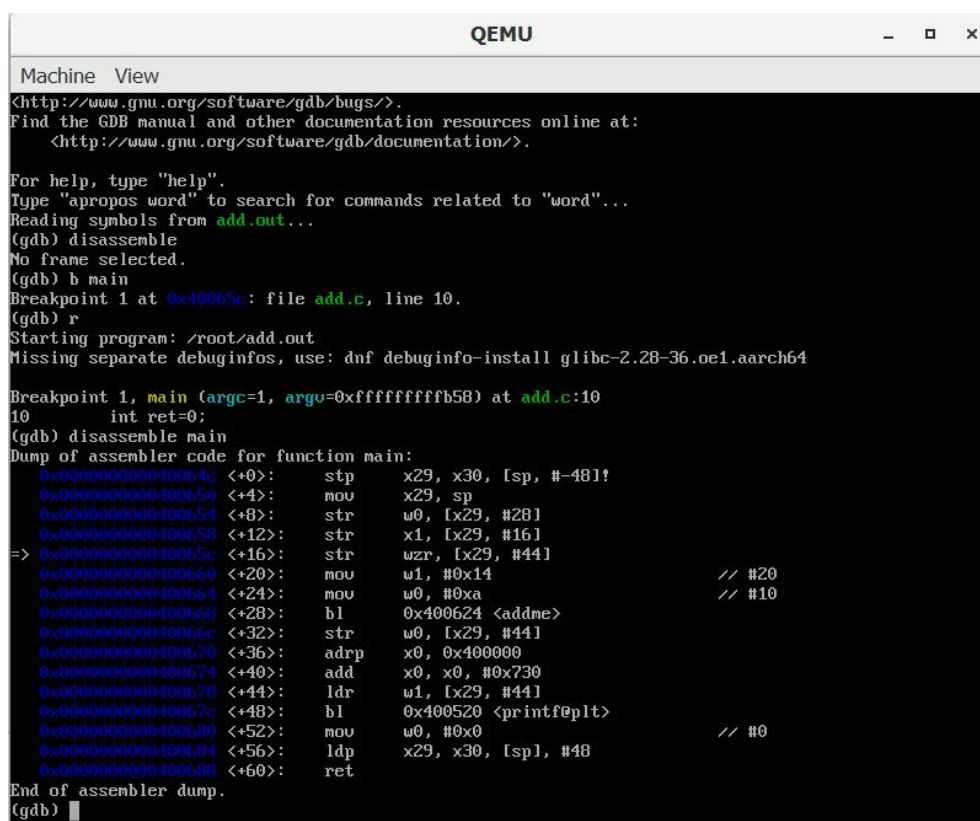
```
QEMU
Machine View
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define len 60000000
char src[len],dst[len];
long int len1=len;
extern void memcpy(char* dst,char* src,long int len1);
int main()
{
    struct timespec t1,t2;
    int i,j;
    for(i=0;i<len-1;i++){
        src[i]='a';
    }
    src[i]=0;
    clock_gettime(CLOCK_MONOTONIC,&t1);
    memcpy(dst,src,len1);
    clock_gettime(CLOCK_MONOTONIC,&t2);
    printf("memcpy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
    return 0;
}

"time.c" 21L, 464C written
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memcpy time is 203765596 ns
[root@localhost ~]#
```

图 3.22 gcc 编译 C 语言与汇编语言的混合程序

此外，在 QEMU 环境中，还可以使用 gdb 等工具调试，如图 3.23 所示。

汇编语言程序设计实验报告



```
Machine View
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from add.out...
(gdb) disassemble
No frame selected.
(gdb) b main
Breakpoint 1 at 0x40065c: file add.c, line 10.
(gdb) r
Starting program: /root/add.out
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.28-36.oe1.aarch64

Breakpoint 1, main (argc=1, argv=0xffffffffb58) at add.c:10
10      int ret=0;
(gdb) disassemble main
Dump of assembler code for function main:
0x00000000-00064c <+0>:      stp     x29, x30, [sp, #-48]!
0x00000000-000650 <+4>:      mov     x29, sp
0x00000000-000654 <+8>:      str     w0, [x29, #28]
0x00000000-000658 <+12>:     str     x1, [x29, #16]
=> 0x00000000-00065c <+16>:     str     wzr, [x29, #44]
0x00000000-000660 <+20>:     mov     w1, #0x14           // #20
0x00000000-000664 <+24>:     mov     w0, #0xa           // #10
0x00000000-000668 <+28>:     bl      0x400624 <addme>
0x00000000-00066c <+32>:     str     w0, [x29, #44]
0x00000000-000670 <+36>:     adrp    x0, 0x400000
0x00000000-000674 <+40>:     add     x0, x0, #0x730
0x00000000-000678 <+44>:     ldr     w1, [x29, #44]
0x00000000-00067c <+48>:     bl      0x400520 <printf@plt>
0x00000000-000680 <+52>:     mov     w0, #0x0           // #0
0x00000000-000684 <+56>:     ldp     x29, x30, [sp], #48
0x00000000-000688 <+60>:     ret
End of assembler dump.
(gdb) █
```

图 3.23 gdb 工具调试界面

使用 gdb 调试，可以像 vs2019 一样设置断点，逐步调试，还可以输入指令实时查看当前运行到的代码段部分，也可以显示出当前函数中的各个局部变量的值和栈帧情况，寄存器中的值也能同时以十六进制和十进制显示，可以看出其功能之强大。但同时，其没有图形化界面这一点也为调试带来了一些困难。

3.3 小结

这部分的实验主要是对不同编程环境的体验。

VS2019 是我们比较常用的 IDE，在 C/C++ 等高级语言中所体现出的功能比较强大，但是对于汇编语言来说稍有欠缺，在一开始使用时没有高亮显示，必须自己安装插件才可以。但是，其在调试和异常处理等方面具有很优秀的功能，可以同时查看反汇编、寄存器、内存、监视等多种窗口，更贴合现代的图形化界面也使得编程与调试工作更加舒适。

DOSBOX 在调试 16 位汇编程序这一点上较为优秀，使用 td 调试工具可以做到和 VS2019 类似的功能，但其调试与鼠标操作的适配度不高，因此需要键盘来辅助调试。在该环境中，可以对中断和编译链接的步骤更加清楚。但其图形化界面设计过于古老，且 16 位汇编程序在实际工程中并不多用，因此该工具用途并不广泛。

QEMU 是 64 位下的处理程序，在该环境中，编写源文件代码的方法较于其他 IDE 或环境更加复杂，且因其没有图形化界面，以及没有自动配置和调试检错的功能，使得调试的难度增加。但是，它的 gcc 编译功能在多文件程序中比较好用，可以将几个文件进行不同的组合然后编译，这样一是可以使得一个代码多用，增加了利用率和空间使用率，二是使调用函数这一功能更加灵活。

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, 2023