

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 2104 班

学 号： U202115452

姓 名： 刘凯欣

指导教师： 金良海

实验时段： 2023 年 3 月 6 日~4 月 28 日

实验地点： 东九 A312

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. 16

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	3
1.3.3 源程序	5
1.3.4 实验记录与分析	5
1.4 内容 1.2 的实验过程	9
1.4.1 指令优化对程序的影响	9
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	11
1.5 小结	12
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	22
2.4 小结	23
三、工具环境的体验	25
3.1 目的与要求	25
3.2 实验过程	25
3.2.1 WINDOWS10 下 VS2022 等工具包	25
3.2.2 DOSBOX 下的工具包	27
3.2.3 QEMU 下 ARMv8 的工具包	28
3.3 小结	32
参考文献	33

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。主要功能为：

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

(2) 在接下来的处理中，先对 N 组（N 的值取 5 左右即可）采集到的状态信息计算 f，并进行分组复制；

(3) 然后将 MIDF 存储区的各组数据在屏幕上显示出来（应让该区的数据不少于 2 组）。

(4) 最后，进入等待按键状态，按 R 键重新从“(2)”处执行，按 Q 键退出程序（按其他键则回到等待按键状态）。

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

此次实验以任务 1.3，1.4，2.1 为基础，在实现相关功能的基础上，进一步拓展功能。需要实现的功能有：用户名和密码比较；流水号信息计算与存放；流水号信息显示；结束等待判断。实现这些功能的段最后将合并起来。

首先，用户进行登入，需要输入正确的用户名和密码，连续三次错误程序即退出。对于用户名和密码的判断，我设定了宏定义 `stringcmp` 进行比较，主程序中循环调用宏。在循环时，选取变量计数，实现三次错误退出功能。

登入成功后，进入流水信息的处理部分。在这一部分，我定义了子程序 `calcul_f` 去实现 f 的计算，定义了子程序 `copy_f` 去实现不同信息区的内容拷贝，定义了 `printf_infor` 去实现 MIDF 的信息显示。在主程序的 `main` 函数中，完成基本的交互和判断，并在相应位置调用这上述三个函数。要注意的是，由于 f 值的不同，对应的流水信息存放位置也不同，故在主程序调用 `calcul_f` 和 `copy_f` 时，设置了相关的标志变量进行判断。

在 MIDF 信息打印完成后，进入等待界面等待用户指令。此时用户键入“r”，则程序重新打印

汇 编 语 言 程 序 设 计 实 验 报 告

MIDF 信息, 输入”q”则退出程序, 其他形式的键入将回到等待界面。这部分在主程序 main 中实现, 通过在流水信息处理部分和等待界面处理部分设置标号实现。

这就是 3.1 实验的主要思想, 下面是是对函数的一些描述。

宏定义 stringcmp: 带有两个参数, 以密码核验为例, 接受设定的密码和用户输入的密码两个变量, 主程序中 invoke 调用。采用逐字节比较, 实现对输入用户名 username 和密码 password 的检查功能。主函数中, 在调用 stringcmp 的同时, 设置变量进行循环计数, 实现三次报错即退出程序的功能。

子程序 calcul_f: 带有三个参数, 接受 SDA,SDB,SDC,计算 f 表达式的值, 通过主程序 invoke 传参和调用。在表达式的运算, 对除法和乘法有一定优化探讨。

子程序 copy_f: 函数不带参数, 但通过主程序 push 传递参数, 采用循环, 并逐字节进行拷贝, 实现数据拷贝功能, call 调用。

子程序 print_infor: 不带参数, 循环输出 MIDF 中存储的信息, 由于在主程序中, 故直接 invoke 调用。

上述子程序中, stringcmp、print_infor 在主模块中, calcul_f、copy_f 在子模块中。

公共符号如下:

username: 预设用户名

password: 预设用户密码

input_name: 用户输入的用户名

input_key: 用户输入的密码

除此之外, 还有所有系统输出的提示信息字符串以及流水信息的结构体定义和变量声明。

汇编语言程序设计实验报告

1.3.2 流程图

图 1.1，主程序流程图。

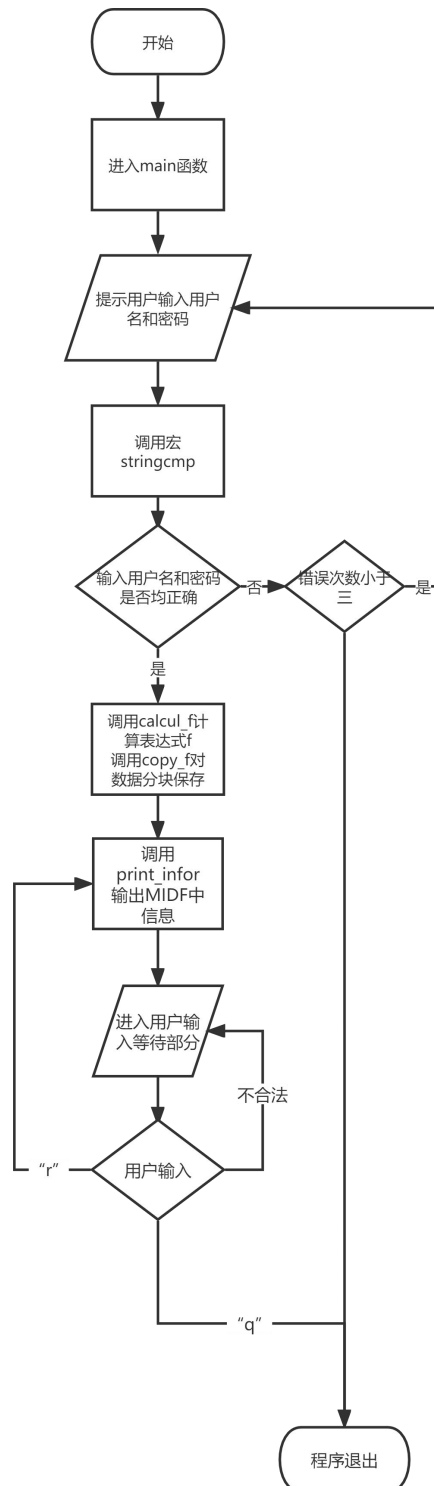


图 1.1 主程序流程图

汇编语言程序设计实验报告

图 1.2，宏定义 strcmp 流程图

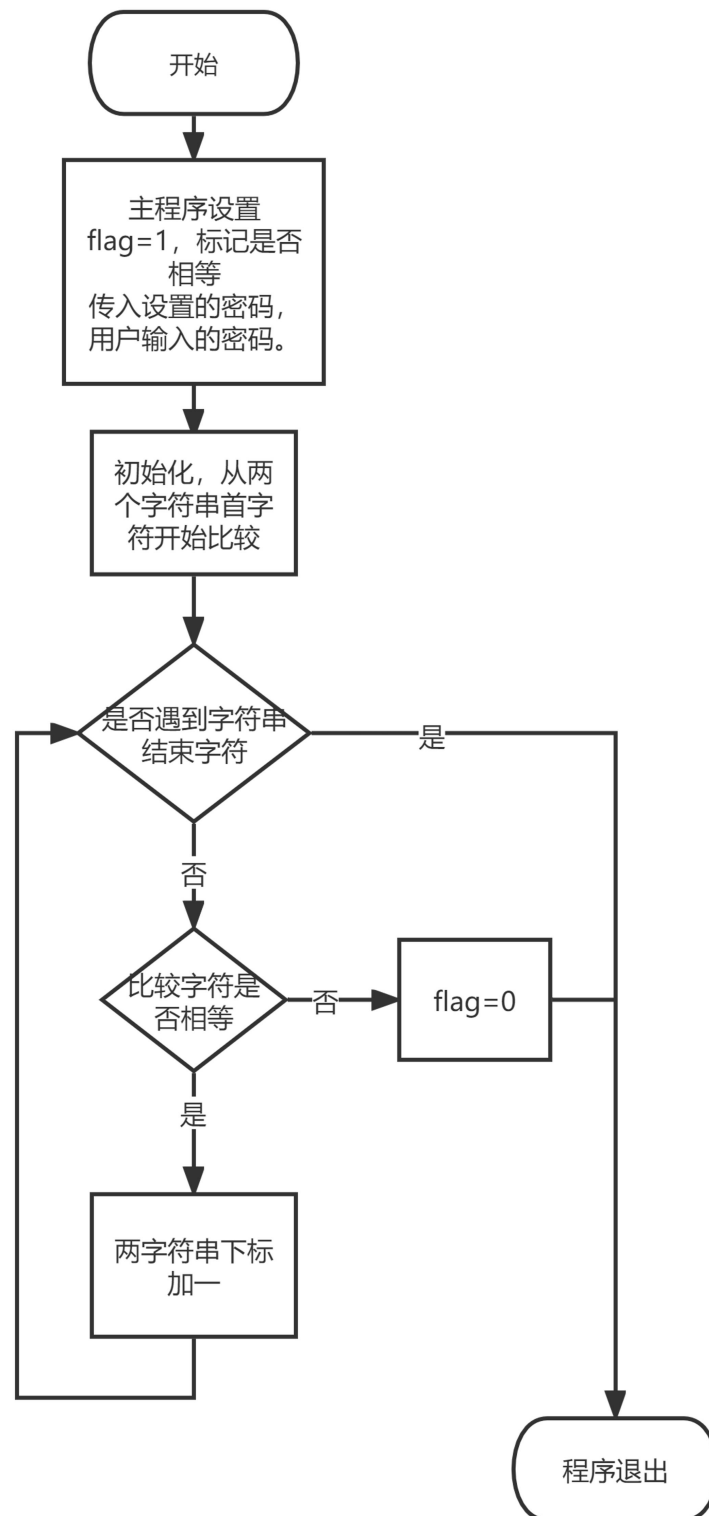


图 1.2 宏定义 strcmp 比较流程（以密码比较为例）

汇编语言程序设计实验报告

1.3.3 源程序

见电子版文件“part3.1.asm”和“part3.1_func.asm”。

1.3.4 实验记录与分析

1. 实验环境条件

AMD 处理器 3.2GHz, 16G 内存; WINDOWS10 下 VS2022 社区版。

2. 汇编、链接中的情况

汇编过程中出现了下面几个问题:

- 1) 关于用 `username` 和 `password` 分配长度的问题。一开始我规定最大长度为 10 个字节, 于是分配了 `db 10 dup(?)`, 显然, 这没有考虑到字符串结尾有结束字符的存在。这一点在 1.3 实验, 先比较密码长度中有所体现。同时要注意, 用户输入的规范性问题。在检测功能时, 我检测了输入长度超过变量分配的内存空间和带有空格输入的情况。针对前者, 考虑到内存访问溢出, 我将存放用户输入的变量放到段末尾; 针对后者, 结合 C 语言中 `scanf` 的性质, `scanf` 读取空格和换行即停止读入, 所以尽量不要输入空格。
- 2) 关于编译器找不到 `main` 函数的问题。该问题源于汇编主程序引用了另一个单独定义结构体的文件 `struct.asm`。无论该文件是否被排除在生成项外, 程序始终报错。经过老师检查纠错发现, 这可能是编译器的问题, 因为无论是上述操作, 还是将该文件后缀名改成 `.h` 和 `.inc`, 都会进行编译。而一旦发生编译, 编译器便会要求在文件结尾加入 ``end``, 显然, 此时主程序若调用该程序, 该调用指令后的所有指令被 ``end`` “注释”掉了。最后我将结构体定义放到主程序中, 可正常运行。该问题和 vs2022 编译器有关, 不太清楚是何种原因。
- 3) 宏定义中标号 `local` 问题。在宏定义中, 我使用了标号, 但在编译过程中程序报错 “symbol redefinition”。查阅后发现是宏的伪标号问题, 宏中伪标号需要 `local` 定义才可使用。而在主程序调用时, 伪标号会替换成别的标号。

3. 程序基本功能的验证情况

(1) 用户名和密码模块测试

设置密码为 “liukaixin”, 进行下列测试:

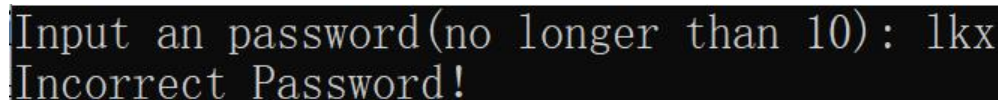
输入正确的密码



```
Input an password(no longer than 10): liukaixin
OK!
```

图 1.3 正确密码结果

输入错误密码:



```
Input an password(no longer than 10): lkx
Incorrect Password!
```

图 1.4 错误密码结果

前缀正确但长度超出密码:

汇编语言程序设计实验报告

```
Input an password(no longer than 10): liukaixin1111
Incorrect Password!
```

图 1.5 前缀正确长度超出密码结果

(2) 数据处理模块

设置三个数据，分别代表 $f < 100$, $f = 100$, $f > 100$ 。结果如下

 LOWF.SF	0x00000063
 MIDF.SF	0x00000064
 HIGHF.SF	0x0000006b

图 1.6 对不同数据的存储情况

(3) 用户登录模块

该部分需要同时输入用户名和密码，只有都正确才可以进入。

用户名设置为“liukaixin”，密码设置为“bestfriend”

输入正确的用户名和密码：

```
Please input username and password (no longer than 10):
liukaixin
bestfriend
OK!
```

图 1.6 用户界面正确输入

输入错误的用户名或密码，以及错误三次程序退出

```
Please input username and password (no longer than 10):
lkx
friend
Wrong username or password!

Please input username and password (no longer than 10):
liukaix
best
Wrong username or password!

Please input username and password (no longer than 10):
liukaixin
friend
Wrong username or password!

Flase three times!Process exits!
```

图 1.7 用户界面错误输入

(4) MIDF 输出测试

我设置了两组数据求得的 $f = 100$ 。分别是

'00000002', 2540, 1, 1 和 '00000005', 2460, 600, 200

输出结果如下：

汇编语言程序设计实验报告

```
SAMID:00000002
SDA:2540
SDB:1
SDC:1
SF:0

SAMID:00000005
SDA:2460
SDB:600
SDC:200
SF:0
```

图 1.8 MIDF 信息输出

(5) 程序等待界面

在上述打印完 MIDF 之后，程序进入等待界面，需要用户输入“q”或“r”进行下一步操作，其他视为非法输入。

```
Input 'r' to replay,'q' to exit:
r
SAMID:00000002
SDA:2540
SDB:1
SDC:1
SF:0

SAMID:00000005
SDA:2460
SDB:600
SDC:200
SF:0

Input 'r' to replay,'q' to exit:
w
Input 'r' to replay,'q' to exit:
q
```

图 1.9 等待界面功能测试

4. 其他

进一步观察的内容：

- (1) 观察不同模块里的段（都采用默认的简化段定义方法）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。再采用完整段定义方法定义两个段，段名不同，观察这两个段在内存里的放置次序。

汇编语言程序设计实验报告

不同模块的可合并段合并后，合并后的段的变量的偏移地址是依次连续存放的。

- (2) 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？

模块间参数的传递包括两种公共型 PUBLIC 和引用型 EXTRN，PUBLIC 型变量将变量暴露给其他模块，向外部授权该变量的使用，而 EXTRN 型变量则是说明这个变量是从外部模块的 PUBLIC 型变量引用过来进行访问的。当符号名或类型不一致时，会产生编译和链接的错误。

- (3) 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？

模块间参数的传递包括两种公共型 PUBLIC 和引用型 EXTRN，PUBLIC 型变量将变量暴露给其他模块，向外部授权该变量的使用，而 EXTRN 型变量则是说明这个变量是从外部模块的 PUBLIC 型变量引用过来进行访问的。当符号名或类型不一致时，会产生编译和链接的错误。

- (4) 通过调试工具观察宏指令在执行程序中的替换和扩展，观察宏和子程序的调用有何不同。宏指令是在编译时，编译器将相应的宏指令进行指令的替换和展开。子程序调用则是程序在运行时，将子程序的 EA 赋值给 IP/EIP，是代码段读取位置的变化。

- (5) 在进行对除法的优化时，除以 128 的操作，可以优化为右移 7 位，在运算上有简化。但是位移操作是否没有考虑到溢出的问题？在 idiv 操作进行双字除法时，我们会先 mov edx, 0，使得溢出位有保证，但是位移指令的相关操作我并不了解多少。

- (6) 关于程序计时使用的时间函数。在实验前，通过搜索资料，我发现使用 GetTickCount PROTO stdcall 也可以实现 ms 级别的计时。

汇编语言程序设计实验报告

1.4 内容 2.1 的实验过程

以实验 2.1 为例，对实验 1.4 的程序进行优化。下面的实验环境均为 AMD 处理器 3.2GHz，16G 内存；WINDOWS10 下 VS2022 社区版。

程序运行 2e8 次，实验采用五组数据。如下表

表 1.1 优化实验的数据表格

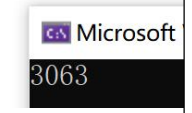

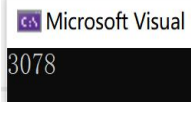
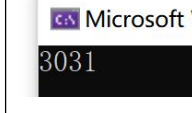
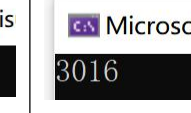
SDA	SDB	SDC
2540	1	100
2540	1	1
2147483647	2147483647	-2147483648
1	2	3
2	3	4

1.4.1 指令优化对程序的影响

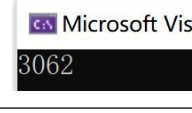
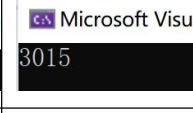
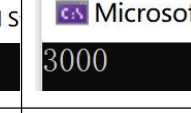


实验中，主要探讨了三个方面的优化：循环中 add 和 inc 的效率比较；除法和移位的效率比较；乘法的优化。在 SAMID 的拷贝中，由于实验开始我就采用了一次循环拷贝 4 字节的方式，所以没有进行相关优化。

1. add 指令和 inc 指令效率比较

add 指令

					平均
3063	3094	3078	3031	3016	3056.4

inc 指令

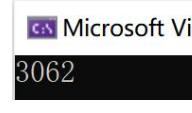
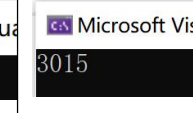
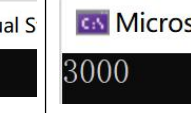


					平均
3062	3015	3000	3031	3062	3034

3034<3056.4，优化了 0.7%，由此，inc 指令更快，后续实验采用 inc 指令。

2. 除法和移位的效率比较






机器在执行指令时，移位操作是要比除法更快的。带着这样的理论分析，对实验中除以 128 的指令进行忧患。

idiv 除法指令

					平均
3062	3015	3000	3031	3062	3034

shr 移位指令

汇编语言程序设计实验报告

 Microsoft V	 Microsoft \	 Microsoft Vis	 Microsoft Visu	 Microsoft V	平均
2875	2844	2890	2734	2828	
2875	2844	2890	2734	2828	2834.2

3034>2834.2, 后者相对前者优化 6.6%, 故 shr 指令更快, 后续采用 shr 指令。

3. 乘法的优化

;原先为了实现 $5*a$, 我采用的是 imul 三操作数指令,

```
imul eax, a, 5
```

;这里可以考虑将 a 移入 eax, 对 a 进行移位操作, 该部分整体重写一下, 伪代码如下

F1:

```
mov eax, 0
```

```
mov eax, a
```

```
shl eax, 2
```

```
add eax, a
```

;或者是, 采用 lea 取偏移地址, 最后基址加变址, 此时可以将 100 优化进去

F2:

```
lea eax, SAMP[ecx]
```

```
mov eax, [eax].SAMPLES.SDA
```

```
mov eax, 100[eax+eax*4]
```

这里只给出最后结果:

方案	5 次平均时间
imul 三操作数	2834.2
F1 移位操作	2912.6
F2 基址加变址	2612.4

2612.4<2834.2<2912.6, 故而方法 F2 更快, 即取偏移地址, 基址加变址运算。

分析认为, 基址加变址, 通过直接访问地址的方式提高了速率, 同时将+100 的操作优化进去, 所以更快。

而 F1 的移位操作, 由于乘 5 的操作不是 2 的幂次, 所以会拆分成两次操作, 故而相对 imul 三操作数会更慢。

1.4.2 约束条件、算法与程序结构的影响

由于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, 若我们利用倒计时进行循环计数控制, 我们便可以在循环体中减少一条指令, 这也是优化时间性能的一个方向。实验中, 进行 LOWF, MIDF, HIGHF 各个区的拷贝信息时, 在访问内存安全的前提下, 可以通过一次性拷贝两个字节甚至四个字节的 (而非逐字节拷贝), 可以减少循环次数, 加快与运行时间。

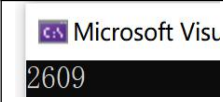



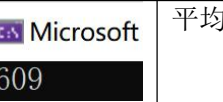
同时我们可以利用模块化的程序设计模式, 以及 C 语言和汇编语言混合的模块化设计模式来设计程序, 让程序的逻辑更加清晰, 功能的添加和删除更加便捷, 降低程序的耦合性, 让程序更加易于维护。C 语言与汇编语言的混合使用可以通过建立两个不同的文件, 然后通过全局变量等方式进行数据通讯的方式来将这两个模块的程序连接到一起。

汇 编 语 言 程 序 设 计 实 验 报 告

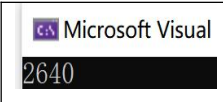


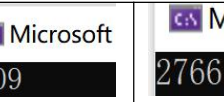
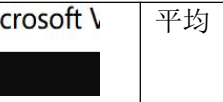
1.4.3 编程环境的影响

显然，对于同一个程序，在不同的编程环境下，运行时间几乎不同；编译器不同，对应的时间也会不同。由于缺少不同的实验环境，为了检测环境的影响，我在同一台电脑上，通过选择电池的不同模式，对程序运行时间进行了测试。

在性能模式下

					平均
2609	2656	2578	2563	2609	2603

在省电模式下

					平均
2640	2719	2625	2609	2766	2671.8

从结果来看，性能模式运行更快。这也是显然的，在性能模式下，电脑追求性能最佳，cpu 利用率更高，从而程序运行更快。

汇编语言程序设计实验报告

1.5 小结

3.1 思考题

(1) 子程序和主程序之间是如何传递信息的？

主程序将下一条指令的地址即 EIP 的值压入堆栈，将要传入子程序的参数依次压栈；子程序通过寄存器或者默认单元将信息传给主程序，在结束时将堆栈中原有的 EIP 的值弹出，传给 EIP，执行主程序下一条指令。

(2) 刚进入堆栈时，堆栈栈顶及之下存放了一些什么信息？

有形参时，堆栈栈顶存放的是形参的值，栈顶之下存放的是其他形参的值及主程序中下一条待执行语句的地址；没有形参时，栈顶存放的是主程序中下一条待执行语句的地址。

(3) 执行 CALL 指令和 RET 指令，CPU 中完成了哪些操作？

执行 CALL 指令时，CPU 将当前 EIP 的值压入堆栈，将子程序的地址送入 EIP；

执行 RET 指令时，CPU 将当前栈顶的数据（双字）送入 EIP，程序回到主程序继续运行。

(4) 若在执行 RET 前把栈顶数值修改掉，那么 RET 执行后程序返回到哪里？

RET 执行后程序返回到被修改后的栈顶数据对应的地址。一般出现这种情况，程序会出现错误。

(5) invoke 伪指令对应的汇编语句有哪些？

3.1 的实验程序中有许多用到了 invoke 的伪指令，例如下面是调用计算 f 函数的 invoke 指令：

```
invoke calcul_f , SAMP[ecx]. SDA, SAMP[ecx]. SDB, SAMP[ecx]. SDC
```

这相当于

```
push      SAMP[ecx]. SDC
```

```
push      SAMP[ecx]. SDB
```

```
push      SAMP[ecx]. SDA
```

```
call      calcul_f
```

(6) 子程序中的局部变量的存储空间在什么位置？

在 [ebp-4] 中。

(7) 如何确定局部变量的地址？

局部变量采用 lea 指令读取地址。

(8) 访问局部变量时的地址表达式有何限制？

单个局部变量作为源操作数或目的操作数，对应的寻址方式是变址寻址方式。

对于数组型的局部变量，使用基址加变址的寻址方式

体会

在这几次实验中，我尝试了运用汇编语言以及汇编语言与 C 语言的结合编写了一些基础的功能，并实现了分模块开发的任务。

工欲善其事必先利其器，在实验中，我熟悉了编译器的调试模式，与 cpp 编译不同的是，汇编程序的调试，几乎都建立在对每条语句的跟踪观察，以及在不同的监视窗口对不停变化的计算机内存内容进行监视与分析。这些内容相当底层，但与此同时又提高了我对汇编的理解。

在利用汇编语言实现功能的过程中，我熟悉了常用的机器指令，不同的寻址方式，掌握了基础的顺序与分支程序设计方法，还利用子程序以及宏来分模块实现了一些功能，这个程序任务中的每

汇编语言程序设计实验报告

一个步骤都不可避免地涉及到字符串的输入输出判断处理、变量地址的存放和查询。在我熟练利用常用机器指令的同时，还逐渐了解了不同的数据在内存中存储的位置以及规律，以通过此来编写程序来对内存中存储的字符串进行比较和操作。在实现特定功能时，循环指令也是必不可少的，这需要我们掌握各种跳转指令以及条件判断语句，存储特定形式的数据时，基础的数据类型已经无法满足我们对这类数据的存储要求，我们便需要利用复合数据类型对相应数据进行封装，批量进行操作。

我们还要对不同的数据类型进行声明与操作，不同数据类型的操作数所对应的指令也略有差别，全局变量和局部变量也需要注意。在汇编和 C 语言混合编程实验中，我就遇到结构如何成为全局变量的问题，甚至因此导致了程序一直无法找到“main”函数的报错。基于这些问题，以及对汇编传递地址的了解，我在书写子程序时，更多的是通过各种方式的传参，将全局变量传递过去。

总的来说，这几次汇编实验以来，我重新翻了几次书，查阅了大量的资料，理论知识更扎实了；在将课本中的知识通过自己理解由键盘来实现的过程中，实践知识更丰富了。

在模块化实验程序之前，我首先在同一个程序文件中编写并调试了所有代码，在调试无误后在进行分模块操作，其中涉及到不同程序间数据的通讯。程序间的通讯问题是个很麻烦的事情，虽然书上和 ppt 上说利用 public 和 extern 的组合即可实现，但是实际操作时始终有这有那的问题。具体问题还是得具体分析啊！这里也有一个很有意思的事情，在实验 1.4 完成后，我们理论课学习了结构体的知识，在进行实验 2 之前，我突然发现 1.4 的内容更应该结构体来完成，于是转头重新构建了 1.4 的代码。这步操作，相当于是提前完成了实验 3.1 的主题部分。

汇编代码实现程序虽然代码量较大，但是由于是各个模块组合而成，构造清晰，没有复杂的指令。每个模块仅执行一个很简单的操作，所有的模块组合在一起就实现了较为复杂的功能。这让人十分有成就感。

在任务 3.1 中，我们需要构造汇编语言的主程序和子程序，我选择主体部分放在主程序中，子程序中则都是函数部分。在 3.2 中，我们需要利用 C 语言和汇编语言进行混合编程，这一部分同样要进行分模块操作，我选择了将 C 语言和汇编语言分成两个不同的模块，仍然是汇编语言作为主模块，是主体部分，而 C 语言的部分则是函数部分。在与同学的交流中，我发现实验 3.2 可以采用 C 语言部分为主体部分，汇编语言为函数部分。这样的分块又该如何写？这很令人感兴趣。

而如何打通 C 语言以及汇编语言的模块也涉及到数据通讯的问题。

在实验中也遇到了一些问题，有些很玄乎，有些纯属是自己的知识或是习惯的问题，在这里就不再赘述。

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

实验主要有以下需求：需要利用中断处理程序实现时间显示并驻留；再次运行程序时需要检测驻留功能是否实现。

对于第一个大问题，进一步分为：插入中断处理程序；从 CMOS 芯片中获取时间信息，并转化成对应的格式进行输出（CMOS 中为 BCD 码）；执行中断处理程序后程序驻留。由于这三者之间是共同处理的，故放在一起说。

在实方式下，终端类型码为 8 的中断为系统时间中断，同时在计算机系统中，每秒会产生 18.2 次中断（近似为 18 次）。因此，我执行原 8 号中断功能，用 IO 指令读取 CMOS 芯片中当前时间的时、分、秒信息（对应地址为 70H 和 71H），同时设计计数函数，通过倒计时 18 次判断是否需要执行显示时间的程序，如果为 18 则中断返回并重新从 8 开始计时，反之调用显示程序并将其存入到对应的内存中。获得时间信息后，再通过 BIOS 软中断指令，int 10H，在屏幕上显示获取的时间信息并驻留。

对于第二个问题，由于中断处理程序会修改原先中断地址，所以我们在再次调用程序时，可以直接通过对比 3508H 21H 获取的原中断地址和即将设置的终端地址是否相同，从而给予反馈。

2. 记录与分析

（1）程序正确性检测

实验需要检查是否实现时间显示驻留，以及时间驻留后，再次调用程序是否会对时间驻留进行反馈。

首先我们看看初始界面。图 2.1 是进入 dosbox 的初始界面。

汇编语言程序设计实验报告

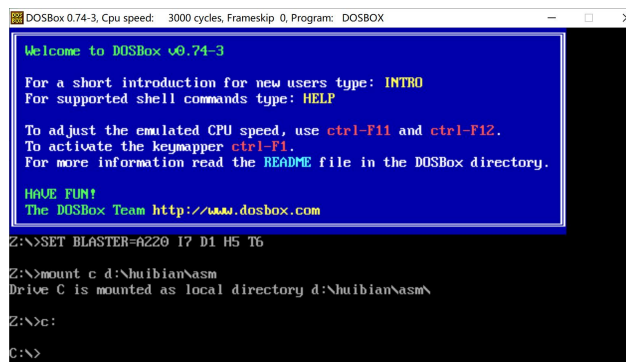


图 2.1 dosbox 初始界面

我的程序命名为 timer.asm, 此前已经编译生成了对应的 exe 文件, 这里直接运行查看结果。键入“2”后, 程序运行, 并在右上角开始显示当前的时间, 并与现实中的时间同步。

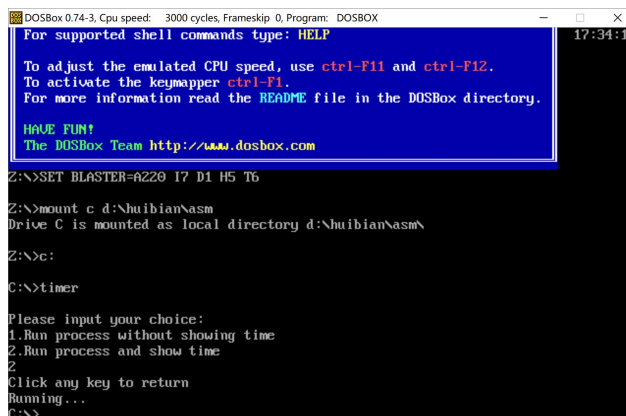


图 2.2 运行中断处理程序后显示时间

随后, 再次运行程序, 查看检查驻留功能。键入后, 提示“Time displays is already installed”。

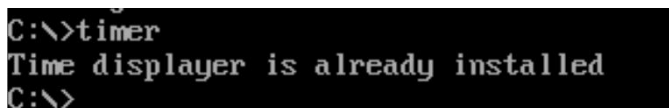


图 2.3 检测功能

(2) 程序特别之处

第一处是如何获取 CMOS 芯片中的时间信息。下面是获取时信息的代码:

```
MOV AL, 4
OUT 70H, AL
JMP $+2
IN AL, 71H
MOV AH, AL
AND AL, 0FH
SHR AH, 4
ADD AX, 3030H
XCHG AH, AL
MOV WORD PTR HOUR, AX
```

CMOS 芯片时间信息是在 70H 和 71H 中。在获得时间信息后, 还需要转换成光标输出。

第二处是如何判断是否已经实现驻留。这里通过先调用一次 8 号中断, 获得其终端地址, 并与将要执行的修改 8 号中断指令设置的地址比较。

```
PUSH CS
```

汇编语言程序设计实验报告

```
POP    DS
MOV     AX, 3508H
INT     21H
CMP     BX, OFFSET NEW08H
JNE     NEXT
LEA     DX, installed
MOV     AH, 9
INT     21h
JMP     EXIT
```

2.3.2 反跟踪程序

1. 设计思想与实验方法

为什么要反跟踪？是为了保证程序运行的安全性，通过加密、扰乱视野（冗余代码）、复杂化操作等方式实现。能够有效避免他人通过反汇编直接跟踪到程序的关键信息。

实验中，我采用了四种方式进行反跟踪：对用户名和密码进行异或加密；设置无效的时间函数；使用 virtual 函数动态修改函数入口；添加冗余代码；。

2. 反跟踪效果的验证

(1) 对用户名和密码进行异或加密

对用户名“liukaixin”和密码“bestfriend”使用“X”异或进行加密。

```
;username db "liukaixin",0
username db 'l' xor 'X','i' xor 'X','u' xor 'X','k' xor 'X','a' xor 'X','i' xor 'X','x' xor 'X','i' xor 'X','n' xor 'X',0
;password db "bestfriend",0
password db 'b' xor 'X','e' xor 'X','s' xor 'X','t' xor 'X','f' xor 'X','r' xor 'X','i' xor 'X','e' xor 'X','n' xor 'X','d' xor 'X',0
```

图 2.4 异或加密

(2) 设置无效的时间函数

在 main 程序的开始和结束时，我设置了 GetTickCount 函数进行计时。实际上这些计时功能并没有作用。

```
main proc c
;功能1, 用户输入正确用户名和密码
invoke GetTickCount
mov starttime, eax
```

图 2.5 程序开始前开始计时

```
EXIT:|
invoke GetTickCount
mov starttime, eax
invoke ExitProcess,0
main endp
```

图 2.6 程序结束时暂停计时

(3) 使用 virtual 函数动态修改函数入口

我使用 virtual 函数动态修改显示 MIDF 的函数的入口地址

```
FINAL:
mov     edx, esi
;call   print_info
mov     eax, lenTH
mov     ebx, 40h
lea     ecx, CopyHere
invoke VirtualProtect, ecx, eax, ebx, offset oldprotect
mov     ecx, lenTH
mov     edi, offset CopyHere
mov     esi, offset machine_code
CopyCode:
mov     al, [esi]
mov     [edi], al
inc     esi
inc     edi
loop    CopyCode
CopyHere:
db     lenTH dup(0)
```

图 2.7 动态修改函数入口地址

(4) 添加冗余代码

汇编语言程序设计实验报告

在不影响程序功能的地方加上一些代码，这些代码实际并无作用，只是用来扰乱视野的。一般来说，jmp 类的跳转语句和 call 类的函数调用语句在这方面有较好效果。

可以在程序的任何位置添加无关代码，比如在计算 f 时，在定义变量时。也可以熟练使用 push pop 操作来添加冗余代码。

cmp	eax,	100			
je	EQUAL		;相等		
jg	GREAT		;大于		
jl	LESS		;小于		
;无用代码					
mov	eax,	10h			
int	21h				
push	eax				
invoke	copy_f				

name	dd	"kaixin",0
key	dd	"bad",0
;username		
username	db	"liukaixin",0
;password		
password	db	"bestfriend",0
;password		
	db	'b' xor 'X','e'

图 2.8 添加无用代码

(5) 反跟踪程序的验证。

我使用的是 W32DSM89 静态反汇编工具。由于对自己的程序较为了解，所以反汇编自己的 exe 文件时，可以较快的得出结果。

3. 跟踪与破解程序

与周汝凡同学组队完成。对她的 debug 版本的 exe 文件进行反汇编。

周汝凡同学的反跟踪设置很成功，我在 30 分钟内并没有找全她的程序的关键信息。在之后的查找和交流中才找全。下面是记录反汇编过程中，跟踪到关键信息的步骤。

在 W32DASM 反汇编工具中，可以通过右上角的 Refs 功能快速查询到程序中的字符串信息。程序中的字符串信息见图 2.9。

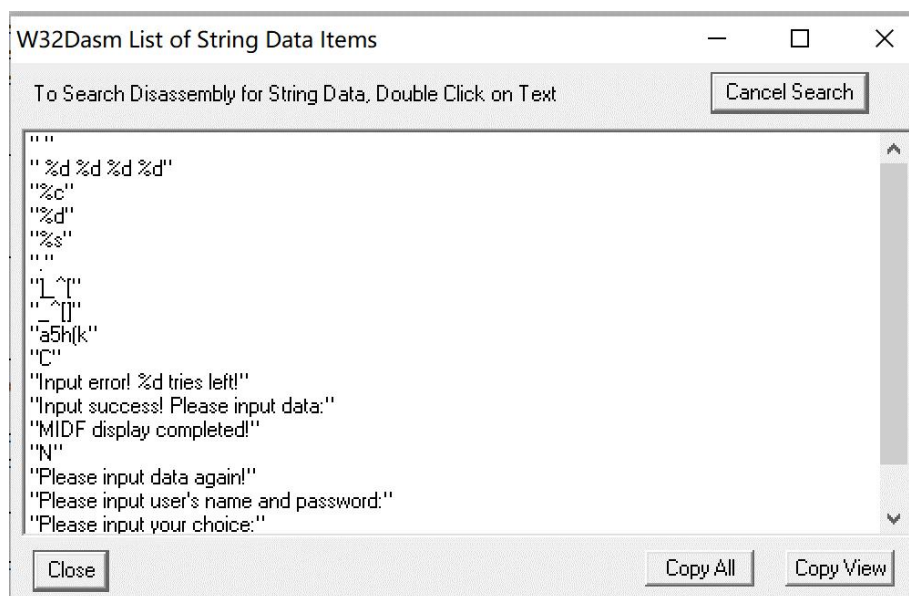


图 2.9 查看 exe 文件的字符串信息

分析途中字符串信息，有一条关键信息 “Input success!Please input data:”。由于程序是按照逻辑进行交互的，所以这必然是在用户名和密码都输入正确后才输出的语句，这就说明如果我们从这句语句入手，就很有可能找到对应的用户名和密码。而上面的 “%s” 对应的就应该是用户名和密码的输入调用了。

汇编语言程序设计实验报告

点击“Input”这条语句，W32DASM 跳转到程序中输出这条语句的地方。

```
* Possible StringData Ref from Data Obj ->"Input success! Please input data:"
|
:00408828 686FD14700      push 0047D16F
:0040882D E8C293FFFF      call 00401BF4
:00408832 83C404          add esp, 00000004
:00408835 EB4F          jmp 00408886
```

图 2.10 跳转到 Input success 语句

从此处往前查询，由于用户名和密码的检查需要用到 cmp 语句，且一般而言会涉及到数值拷贝的操作，如果我们找到这样的代码块，那么这一块将大概率是用户名登陆界面。

果不其然，在这条语句的前不远处，找到了有上述特征的代码块。

```
|
:004087CE 890D2BD14700      mov dword ptr [0047D12B], ecx
:004087D4 3B0D27D14700      cmp ecx, dword ptr [0047D127]
:004087DA 7524             jne 00408800
:004087DC 33C9             xor ecx, ecx
:004087DE 8A140E           mov dl, byte ptr [esi+ecx]
:004087E1 8A340F           mov dh, byte ptr [edi+ecx]
-----
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004087F7 (U)
|
:004087E4 3B0D27D14700      cmp ecx, dword ptr [0047D127]
:004087EA 7D0D             jge 004087F9
:004087EC 3AD6             cmp dl, dh
:004087EE 7510             jne 00408800
:004087F0 41              inc ecx
:004087F1 8A140E           mov dl, byte ptr [esi+ecx]
:004087F4 8A340F           mov dh, byte ptr [edi+ecx]
:004087F7 EBEB             jmp 004087E4
```

图 2.11 有比较特征的代码块

实际上，这样的代码块有两处，说明一处是用户名检验，一处是密码检验。仔细查看后，两处的处理几乎相同，故只以图 2.11 中所示的部分为例进行记录。

继续从图 2.11 部分的代码往上看，我看到了图 2.12 中比较奇怪的语句。

```
* Possible StringData Ref from Data Obj ->"a5h(k"
|
:004087A3 BEC2D04700      mov esi, 0047D0C2
:004087A8 33C9             xor ecx, ecx

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004087B6 (U)
|
:004087AA 8A140E           mov dl, byte ptr [esi+ecx]
:004087AD 80FA00           cmp dl, 00
:004087B0 7406             je 004087B8
:004087B2 41              inc ecx
:004087B3 8A140E           mov dl, byte ptr [esi+ecx]
:004087B6 EBF2             jmp 004087AA
```

图 2.12 比较奇怪的输出符号

“a5h(k”可能是某种冗余代码，也有可能是某种字符串经过处理后显示出来的奇怪模样。但结合图中下面的代码（mov esi, 0047D0C2 和 mov dl, byte ptr [esi+ecx]）和图 2.11，显然是程序试图从 data 段（0047D0C2 是 data 段的地址）取出某种字符并参与后续的比较！那么可以肯定这部分涉及到关键信息了！同时分析知道，这部分是求对应字符串的长度，并将结果存放到 ecx 中。

汇编语言程序设计实验报告

接下来利用 W32DASM 的功能，可以直接查看到 data 段的信息。点击上方的 HexData，选择第一行“of Data”，查看 data 段信息。并找到 0047D0C2 处存放的信息。

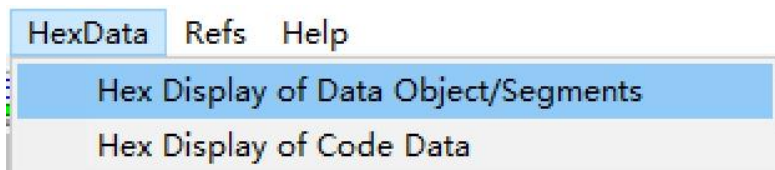


图 2.13 W32DASM 查看 data 的功能

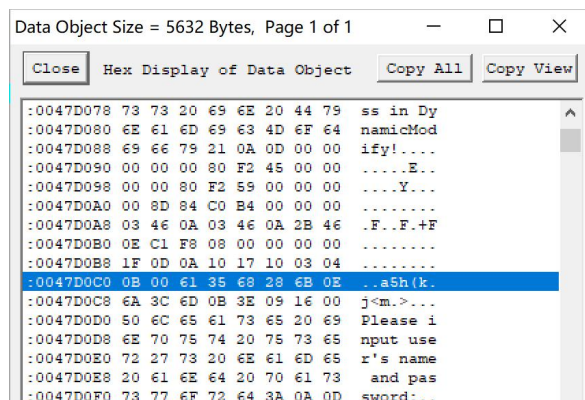


图 2.14 查看 0047D0C2 处存放的信息

从 0047D0C2 开始的字符串的信息为“61 35 68 28 6B 0E 6A 3C 6D 0B 3E 09 16”（16 进制），记作 STR。于是可得这一部分为设定的密码或用户名。在图 2.12 下方还有一部分求字符串长度的代码，但查询对应的 data 段后发现对应内容为 0，可以推断出这部分为求用户输入内容的长度。

接着往下看，如图 2.15。分析发现，这部分进入了字符比较的部分。在一开始比较长度是否相同（如，cmp ecx, dword ptr [0047D127]，0047D127 是程序分配的存放长度的变量地址），随后进入循环比较字符的部分。

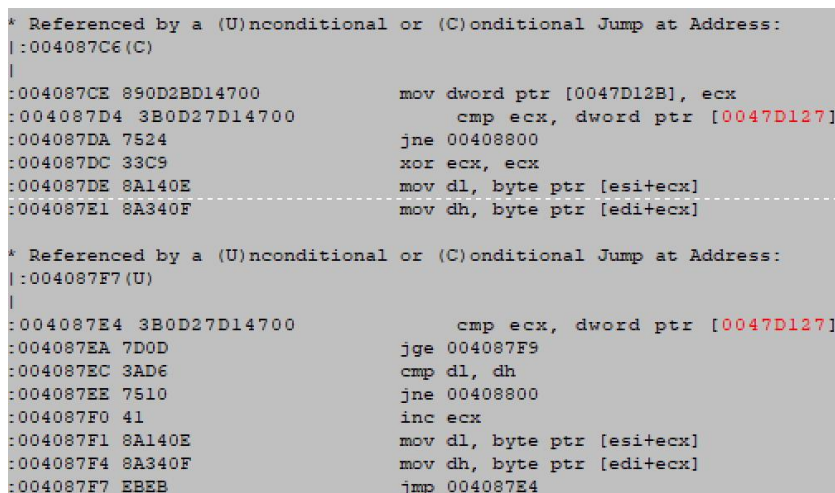


图 2.15 进入比较部分

这里好像没有对字符串进行异或的操作？我对上面的 STR 直接输出看看如何。

汇编语言程序设计实验报告

Microsoft Visual Studio 调试



图 2.16 STR 输出

显然这不是真正的密码。

既然没有看到对 STR 进行操作，那必然是对输入的部分进行了加密操作，接下来就是跟踪相关内容。后续实验中，查找输入部分操作代码的过程较为复杂，也正是因为这一步，没有在 30 分钟内完成跟踪。下面简述一下相关过程。

在意识到问题的关键在于查找到修改输入字符串的代码后，我继续查询关键字字符串的信息，进而找到程序用户名登陆的真正入口，随后从 scanf 部分的%s 处开始跟踪代码。在用户输入完代码后，输入字符串进入了 virtual 函数（动态修改代码）中。在该函数中，还调用了一个在程序中定义的有三字节长度的变量。这个变量拷贝了 data 段中“0047D09A”的三个字节，对应内容如图 2.17。

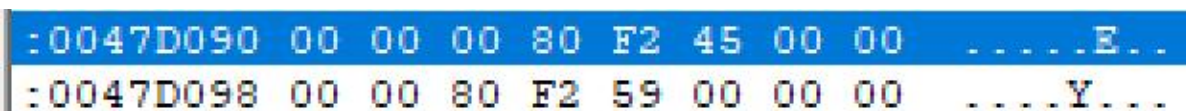


图 2.17 0047D09A 处三个字节的内容

大胆猜测，小心验证。于是我猜测程序是对输入字符串异或了“Y”。随着我继续跟踪相关信息，发现了图 2.18 中，virtual 函数的部分。

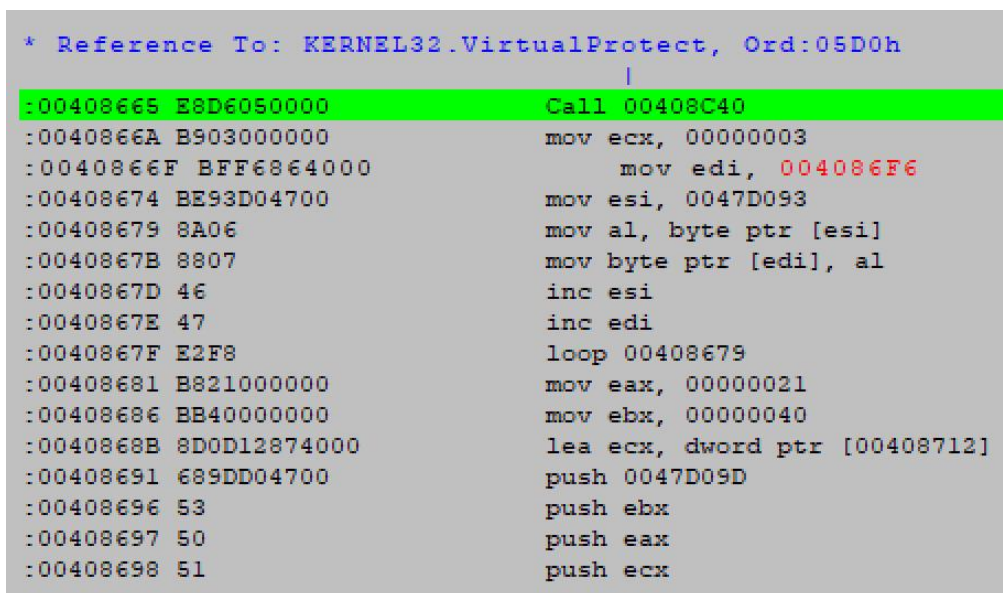


图 2.18 virtual 函数

这里唯一一个 call 函数非常关键！很有可能会找到对应的加密操作。但是当我 call 进去后，进入了一段非常复杂而且有无法跳转到的代码。如图 2.19。

汇编语言程序设计实验报告

```
* Reference To: KERNEL32.VirtualProtect, Ord:05D0h
|
:00408C40 FF2504004800      Jmp dword ptr [00480004]
:00408C46 56                push esi
:00408C47 6A01             push 00000001
:00408C49 E8C291FFFF      call 00401E10
:00408C4E E8F18FFFFF      call 00401C44
:00408C53 50                push eax
:00408C54 E8D4BAFFFF      call 0040472D
:00408C59 E88B95FFFF      call 004021E9
```

图 2.19 进入 call 函数后的部分代码

于是我的跟踪部分到这里就无法进行下去了（由于 W32DASM 的限制，无法读入完整的 exe 二进制文件，最多读到 283640 行，这点在“3.2.2 DOSBOX 下的工具包”中会谈到）。因此，只能大胆猜测 STR 这部分是异或的“Y”。异或 Y 后的结果如图 2.20。

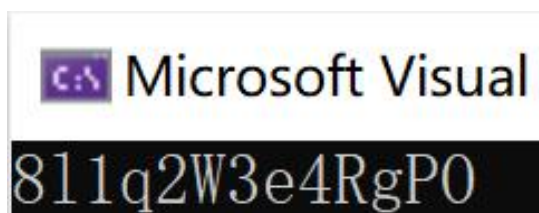


图 2.20 STR 异或“Y”的结果

与周汝凡同学交流后确实是如此。其中的“81”和“gP0”甚至还是她的冗余加密部分。

这时候另一部分“1F 0D 0A 10 17 10 03 04 0B”就猜测是与“E”进行异或了。对应的结果为“ZHOURUFAN”，是正确的。后续跟踪 f 的计算公式，结果为 $f = (9a + 180 + 2b - c) / 256 + 100$ 。

在上述反汇编跟踪程序完成后，我查看了周汝凡同学的代码，并和她进行了沟通，这样大致了解了她的反跟踪的手段。

在用户登录界面，对用户名和密码均使用了动态修改内存的方式调用函数。在函数传递参数时，传递存放三个关键参数的变量首址，其中最后一个参数就是对输入进行异或的字母，这点与图 2.17 中查看到的情况相同。

```
Modify_name_code db 080H, 0F2H, 45H
len_modifyname = $ - Modify_name_code
oldprotect_name dd ?

Modify_password_code db 080H, 0F2H, 59H
len_modifycode = $ - Modify_password_code
oldprotect_code dd ?
```

图 2.21 存放三个关键参数的变量

汇编语言程序设计实验报告

下面是相对应的 virtual 函数调用部分。

```
mov eax, len
mov ebx, 40H
lea ecx, DynamicModify_name

invoke VirtualProtect, ecx, eax, ebx, offset oldprotect

mov ecx, len_modifyname
mov edi, offset DynamicModify_name
mov esi, offset Modify_name_code

CopyCode_name: ;动态修改执行代码
mov al, [esi]
mov [edi], al
inc esi
inc edi
loop CopyCode_name
```

图 2.22 相关函数调用部分

后面计算 f 的模块，周汝凡同学同样使用了动态修改内存的方式进行调用。

总的来说，周汝凡同学的反跟踪代码设置的十分成功，在跟踪程序的过程中，从中我也学习到了诸多汇编的相关知识和跟踪技巧。

2.3.3 指令优化及程序结构

1. 实验方法

实验 2 设计的优化部分在 1.4 中已经有说明。下面仅对实验 3.2、5.2 的优化进行说明。

实验 3.2 是 C 语言和汇编语言的混合编程。高级语言相较汇编语言来说，在逻辑和代码上就简单了很多，这是一个主要的优化。比如在汇编中定义的宏（用来比较用户名和密码输入是否正确），在 C 语言中是可以直接调用 strcmp() 函数进行比较的。还有重新输入信息覆盖函数，用 C 语言实现，用变量代替了大量的寄存器取值、地址运算操作等等，实现了程序的“瘦体”。3.2 实验中，我仍以汇编文件为主体，将大量的函数、子程序部分“外包”到 C 语言文件中，使得整体的结构更为简洁。

实验 5.2 采用的优化方法为，将按照字节为单位的读写，转换成多个按照双字为单位进行读写。这一点与实验 3.1 中拷贝信息优化相同。通过一次多字节操作，减少循环次数，提高了运行速度。

2. 特定指令及程序结构的效果

关于实验 3.2 的相关效果，可以参考 3_2.exe 程序。

关于实验 5.2 的测试结果，可以参考“三. 工具环境的体验” - “3.2.3 QEMU 下 ARMv8 的工具包”。

汇编语言程序设计实验报告

2.4 小结

3.2 思考题

(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？

在 C 语言程序中，外部变量和外部函数均需在声明语句前添加 `extern` 表明其可以被外部访问。在汇编程序中，外部变量使用 `extern` 声明，外部函数使用 `proto` 声明，同时声明的参数需要类型一致。

(2) 如何保证在 C 语言程序和汇编程序中，正确访问采集到的状态信息结构中的数据？

实验中我尝试通过“`extern struct SAMPLES`”的方式调用结构体，但多次尝试均失败了。在调用时始终报错：“表达式必须指向完整对象类型的指针”。考虑到，汇编调用 C 语言函数时，传递的都是参数的地址，故而通过在 C 语言程序中，定义与汇编主程序相同存储结构的结构体，即可实现对应的功能。

在汇编程序中，结构体中的变量在内存中连续，而在 C 语言程序中，结构体中的变量默认按照对齐的方式存储，在内存中不一定连续，这一点至关重要。在实验中，我就因为设置流水号的长度为 9 个字节，发现最后输出的 MIDF 信息时钟不正确。后来我修改流水号长度为 8 个字节，结果才正确。通过搜集资料发现，还可以通过 `#pragma pack(n)` 的方式在 C 语言程序中强制设置对齐方式，当 `n=1` 时，就是不进行对齐，内存中连续。

(3) (地址) 类型转换的含义是什么？

实际上对地址的不同解释方式，其实际的地址值没有发生改变，只是指向地址的指针类型发生改变，指针指向对象的数据大小读取范围发生改变。

(4) 如果汇编语言子程序不是 C 语言风格，被 C 语言程序调用时是否会出错？

通过实验观察并查询相应资料，我发现，当汇编语言子程序语言风格不是 C 语言风格时，会引起函数调用规范的冲突，导致程序无法按预期执行

体会

通过实验 3，我对分模块的设计有了进一步的理解。分模块的设计，除了汇编与汇编，C 语言与 C 语言，还有两者的混合汇编。经过实践后，我愈发感觉到高级语言相对于汇编语言给编程带来了极大的简便性，但其中如指针的内存操作等思想，汇编语言则给出了最好的解答。两者相辅相成，互助前行。

通过实验 4，我对中断处理程序、反跟踪处理的理解更加深刻，对静态反汇编跟踪程序的操作有了一定的掌握。在任务 4.1 中，我基于样例代码做出了一定修改，能够读懂相关代码，实现了相关功能。我还通过反复查阅资料，浏览关于中断表的使用方法，设置好了中断处理程序。但是平心而论，我对中断矢量表的还有很多地方不清楚的，相关的中断操作也有待提高。在任务 4.2 中，通过查看书中内容，我了解到了两个特殊函数：一个是自动修改返回地址的子程序；一个是动态修改执行代码的程序。前者通过栈操作是 `eip` 指向目的机器指令，后者涉及到的部分我的理解还有所欠缺。任务 4.3 是我耗时最长的一个实验，一方面是 W32DASM 在网上较难找到相关的使用说明，第二个是 W32DASM 有所缺陷，不能完整的读入 `exe` 文件，导致我的跟踪操作进行到某一步将不得不停止。但是在跟踪的过程中，有大量的地址操作，堆栈操作，`call` 和 `jmp` 操作，很好的锻炼了我的相关

汇编语言程序设计实验报告

能力。

实验中，涉及到的 W32DASM 静态反汇编工具，在反汇编功能上比较强大，比如可以直接反应出程序输出的字符串信息，可以查看 data 段和 code 段查看信息。比较遗憾的是，在网上关于 W32DASM 的使用教程较少，其中的大多数功能我在实验中不会用或者用的不太熟练。而且，W32DASM 无法读入完整的 exe 文件，导致我在跟踪程序的过程中遇到了一定的困难。

经过跟踪同学的 exe 程序，我意识到实验 4.2 我的反跟踪代码还不能打到安全性的要求。有一部分原因是我的用户登录模板并没有设置成子程序，而是放在 main 代码块中运行的。后续也没有将其设置成函数，用动态修改代码的方式进行反跟踪。

总的来说，从实验 3 和实验 4，我收获了很多知识，也仍有些疑惑，但是终归是收获更多。

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。

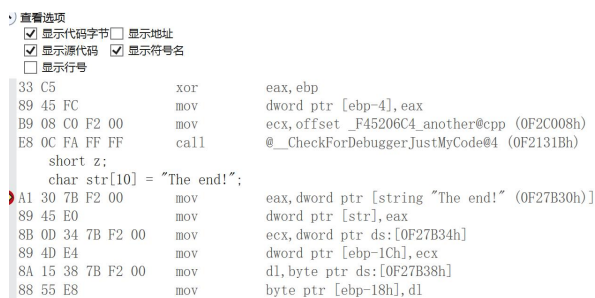
3.2 实验过程

3.2.1 WINDOWS10 下 VS2022 等工具包

AMD 处理器 3.2GHz, 16G 内存; WINDOWS10 下 VS2022 社区版。

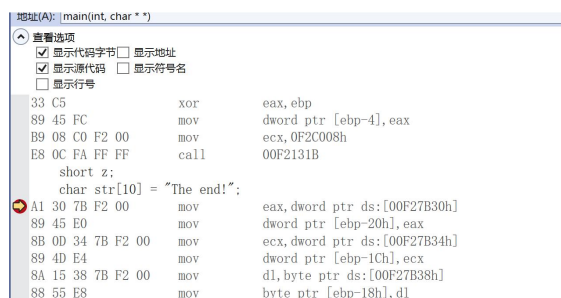
(1)显示反汇编窗口, 了解 C 语言与汇编语句的对应关系。

cpp 文件需要在步进调试的时候才可以显示反汇编窗口。按下 F10 或者 F11 进入调试, 通过上方调试→窗口→反汇编的顺序进入反汇编窗口, 下面显示的是 main 函数前后指令的反汇编语句。



```
33 C5          xor     eax, ebp
89 45 FC       mov     dword ptr [ebp-4], eax
B9 08 C0 F2 00 mov     ecx, offset _F45206C4_another@cpp (0F2C008h)
E8 0C FA FF FF call    @_CheckForDebuggerJustMyCode@4 (0F2131Bh)
short z;
char str[10] = "The end!";
A1 30 7B F2 00 mov     eax, dword ptr [string "The end!" (0F27B30h)]
89 45 E0       mov     dword ptr [str], eax
8B 0D 34 7B F2 00 mov     ecx, dword ptr ds:[0F27B34h]
89 4D E4       mov     dword ptr [ebp-1Ch], ecx
8A 15 38 7B F2 00 mov     dl, byte ptr ds:[0F27B38h]
88 55 E8       mov     byte ptr [ebp-18h], dl
```

图 3.1 显示符号名的反汇编



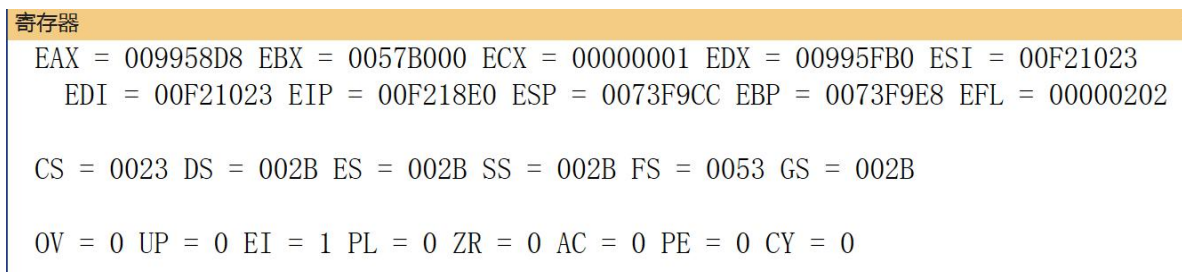
```
33 C5          xor     eax, ebp
89 45 FC       mov     dword ptr [ebp-4], eax
B9 08 C0 F2 00 mov     ecx, 0F2C008h
E8 0C FA FF FF call    00F2131B
short z;
char str[10] = "The end!";
A1 30 7B F2 00 mov     eax, dword ptr ds:[00F27B30h]
89 45 E0       mov     dword ptr [ebp-20h], eax
8B 0D 34 7B F2 00 mov     ecx, dword ptr ds:[00F27B34h]
89 4D E4       mov     dword ptr [ebp-1Ch], ecx
8A 15 38 7B F2 00 mov     dl, byte ptr ds:[00F27B38h]
88 55 E8       mov     byte ptr [ebp-18h], dl
```

图 3.2 不显示符号名的反汇编

通过左右图的比较发现, 显示符号名, 编译器会用“()”将地址包括在其中, 将地址存放的变量以及内容显示出来, 比如 ptr [string “The end!” (0F27B30h)]和 ptr ds:[00F27B30h].

(2)显示寄存器窗口。

通过调试→窗口→寄存器的顺序进入寄存器窗口。在这个界面可以实时观察各个寄存器的值。



```
寄存器
EAX = 009958D8 EBX = 0057B000 ECX = 00000001 EDX = 00995FB0 ESI = 00F21023
EDI = 00F21023 EIP = 00F218E0 ESP = 0073F9CC EBP = 0073F9E8 EFL = 00000202

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053 GS = 002B

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

图 3.3 显示寄存器、段寄存器、标志寄存器

第一行是通用寄存器, 有: EAX, EBX, ECX, EDX, ESI, EDI, EIP, ESP, EBP, EFL

第二行是段寄存器, 有: CS, DS, ES, SS, FS, GS

第三行是标志寄存器, 有: OV, UP, EI, PL, ZR, AC, PE, CY

(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。

汇编语言程序设计实验报告

下面是监视整形变量 `x`, `y` 和寄存器 `eax`, `ebx` 的图。汇编中, 调用变量名, 就是访问内存给该变量分配的地址, 这与寄存器不同。在监视中要注意究竟是访问地址, 还是访问地址指向的值。

监视 1		
搜索(Ctrl+E) < -> 搜索深度: 3		
名称	值	类型
▸ a	0x00f2a000 {0x00000001, 0x00000002, 0x00000003, 0x0...	int[0x00000005]
x	0x0064	short
y	0x8044	short
eax	0x00db58d8	unsigned int
ebx	0x0083c000	unsigned int
添加要监视的项		

图 3.4 显示监视窗口

观察汇编指令, 最直接的当然还是直接观察内存, 图 35 是观察整形变量 `x` 存放的数据。

内存 1															
地址: 0x00F2A014													列: 自动		
0x00F2A014	64	00	00	00	44	80	00	00	00	00	00	fc	9a	cd	d2
0x00F2A024	03	65	32	2d	00	00	00	00	01	00	00	00	01	00	00
0x00F2A034	01	00	00	00	01	00	00	00	01	00	00	00	00	00	00
0x00F2A044	ff	ff	ff	ff	01	00	00	00	2f	00	00	00	01	00	00
0x00F2A054	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00F2A064	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

图 3.5 显示内存窗口

内存的观察很方便, 但是不会主动对用户输入的地址表达式判断正错。同时也有点要注意, 如果通过表达式 “`eax`” 寻访 `eax` 的内容, 一旦 `eax` 变化, 内存地址这一行的值不会主动变化, 需要重新输入 “`&eax`”, 这一点在使用上容易出错且有些麻烦。

(4)有符号和无符号整型数存储方式。

有符号数和无符号数都是以补码的形式存放到内存中。机器直接读取内存, 此时是没有有无符号数的区别的。实验中, 我做了如下定义 `short x=-100; unsigned short x1=-100;`通过监视器查看两个变量的值。结果如下图:

x	0xff9c	short
x1	0xff9c	unsigned sh...

图 3.6 有符号数和无符号数相同值时的内存显示

再输出显示两个变量, `printf("%d %d",x,x1);`此时输出 “-100 65436”, 但是 `x` 和 `x1` 的储存地址中, 两者都是 `0FF9CH`。由此有个直观的反应, CPU 只会根据输入信号进行逻辑运算, 在硬件级别上是没有无符号有符号的概念, 运算结束会根据运算前的信号和输出信号来设置一些标志位。

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

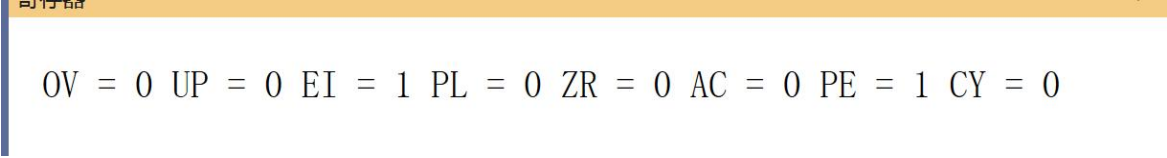
有符号数和无符号数的加减运算无差别, 都是先进行补码运算, 然后根据有无符号数以及相关标志寄存器, 来输出相对应的结果。

汇编语言程序设计实验报告

执行比较指令时，先进性减法运算，再根据结果的正负（标志寄存器）跳转到对应语句。

在实验中，对有无符号数的加减法我进行了一些探究。我设置了 `short x=100; short y=-32700; short z=x-y;`

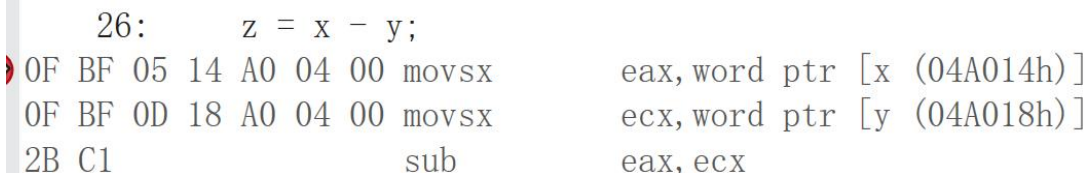
理论分析上述指令，在运算到 `x-y` 时，结果为 32800，此时应当溢出，OF=1.但是实际程序运行到这里的时候，标志寄存器却为 0！如下图：



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

图 3.7 运行到“`x-y`”时的标志寄存器

这里询问老师，老师说可以结合汇编指令查看，图 3.8 是 `z=x-y` 的汇编指令



```
26:      z = x - y;
0F BF 05 14 A0 04 00 movsx    eax, word ptr [x (04A014h)]
0F BF 0D 18 A0 04 00 movsx    ecx, word ptr [y (04A018h)]
2B C1                      sub     eax, ecx
```

图 3.8 `z=x-y` 处的汇编指令

这里临时存储 `xy` 的寄存器是 `eax` 和 `ecx`，这是 32 位的！也就是说，`xy` 的符号位有拓展，在运算时的溢出是对应 `eax` 的溢出，而对于低 16 位，没有溢出！这和编译器有一定的关系。

(6)关于有无符号数的其他问题。

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。汇编中，进行比较时，只会看二进制的补码，并不关注符号。但到了 `c` 语言层面，会判断符号问题。故程序在运行时有警告信息但不报错。故而，`i<length` 最终采用的是无符号数比较。

若将语句“`z = sum(a, 5);`”换成“`z = sum(a, 0x90000000);`”此时可以运行，程序视 `length` 为正数：`0x90000000`.故运行时时间较长。

如果将 `sum` 函数的参数 `unsigned length` 改为 `int length`，执行“`z = sum(a, 0x90000000);`”此时，`length` 为负数，不进入循环，直接结束。结果为 0.

3.2.2 DOSBOX 下的工具包

(1) 工具包的基本使用

我是在学习王爽老师的《汇编语言》时安装的 `dosbox`，在实验之前已经配置好了环境。这里说一下配置环境时的一个简化后续操作的步骤。在每次进入 `dosbox` 时，需要输入“`mount c dosbox` 的路径”和“`c:`”。这两步是将 `dosbox` 挂载在 C 盘上运行。通过在 C 盘中找到“`dosbox-0.74-3.conf`”配置文件，在文件最下面加上上述两条语句。之后每次进入 `dosbox` 就不再需要重新输入了。

`Dosbox` 中，有许多功能，其中常用的有 `debug` 功能，`td` 功能。此次实验主要使用 `td` 功能。

(2) 汇编程序的生成过程

当我们编写了一个汇编文件，如 `demo.asm`，如何在 `dosbox` 中生成 `exe` 文件呢？

汇编语言程序设计实验报告

首先，输入 `masm demo.asm`，生成 `demo.obj` 文件。再输入 `link demo.obj` 或者 `link demo`，就可以生成 `demo.exe` 文件。随后，输入 `demo` 或者 `demo.exe` 就可以运行程序了。

(3) 对实验 4.1 的程序 `td` 查看。

a. 查看中断矢量表

程序运行前，8 号中断程序对应的地址为“`F0 00 FE A5`”，运行后为“`01 AF 01 0B`”

可见下图。

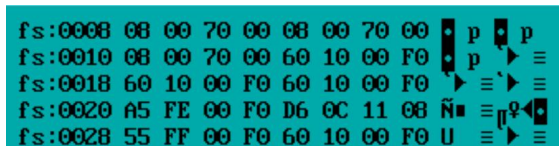


图 3.9 运行程序前 8 号矢量表

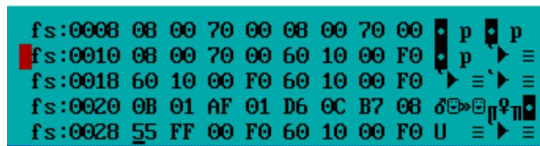


图 3.10 运行程序后 8 号矢量表

b. 查看驻留判断代码

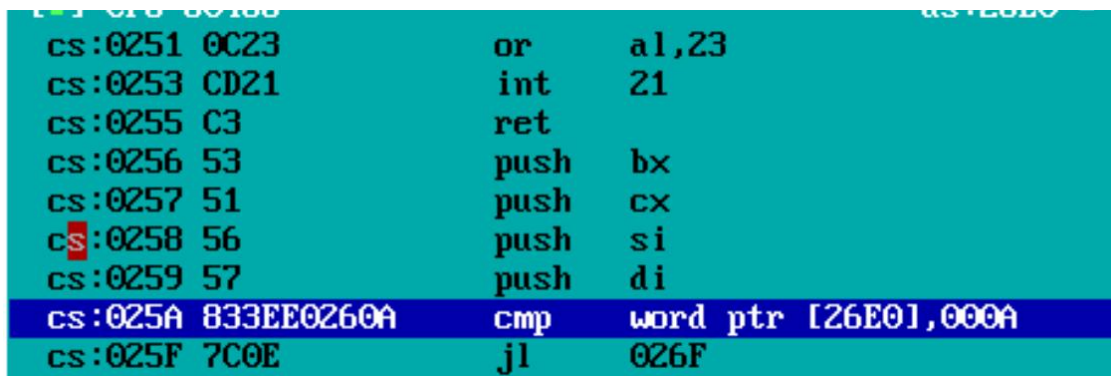


图 3.11 判断驻留代码

c. 查看 CMOS 指定端口的内容

查看时钟信息

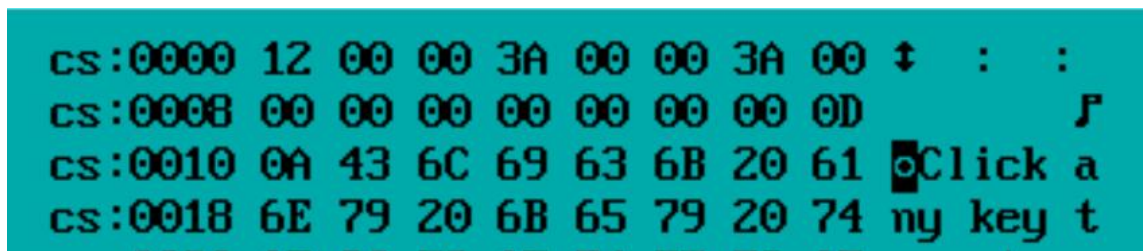


图 3.12 时钟信息存放在 `cs:0000` 处

直观可看右上角



图 3.13 `td` 中右上角的时间信息

3.2.3 QEMU 下 ARMv8 的工具包

1. 显示“Hello World”的汇编语言程序实现

汇编语言程序设计实验报告

(1) 进入登陆界面

```
System information as of time: Tue May 16 03:21:09 UTC 2023
System load: 5.92
Processes: 93
Memory used: 4.9%
Swap used: 0.0%
Usage On: 6%
IP address:
Users online: 1
```

图 3.14 qemu 登陆界面

(2) 运行 hello 程序

通过 vi hello.s 创建并写入代码后，通过 as 和 ld 指令生成对应的可运行程序 hello

```
[root@localhost ~]# ll
total 8.0K
-rw----- 1 root root 0 May 2 14:14 a.out
-rw----- 1 root root 156 May 2 14:28 hello.s
drwx----- 2 root root 4.0K Feb 27 2022 test
[root@localhost ~]# as hello.s -o hello.o
[root@localhost ~]# ld hello.o -o hello
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
[root@localhost ~]# ./hello
Hello World!
```

图 3.15 成功运行 hello 程序

2. 内存拷贝函数的测试与优化实现

实验中需要创建四个文件并进行时间检测。由于运行时间会有波动，故运行多次进行记录。综合来看，优化是有效果的。见下面各图。

(1) 优化前的 copy.s 文件的运行

```
[root@localhost ~]# gcc time.c copy.s -o m1
[root@localhost ~]# ./m1
memorycopy time is 207857417 ns
[root@localhost ~]# ./m1
memorycopy time is 190795869 ns
[root@localhost ~]# ./m1
memorycopy time is 209029660 ns
[root@localhost ~]# ./m1
memorycopy time is 210105115 ns
```

图 3.16 copy.s 的运行

(2) 2 倍展开优化的 copy121.s 文件的运行

```
[root@localhost ~]# gcc time.c copy121.s -o m121
[root@localhost ~]# ./m121
memorycopy time is 3485101136 ns
[root@localhost ~]# ./m121
memorycopy time is 163373163 ns
[root@localhost ~]# ./m121
memorycopy time is 187278915 ns
[root@localhost ~]# ./m121
memorycopy time is 187561631 ns
```

图 3.17 2 倍展开优化的 copy121.s 的运行时间

汇编语言程序设计实验报告

(3) 4 倍展开优化的 copy122.s 文件的运行

```
[root@localhost ~]# gcc time.c copy122.s -o m122
[root@localhost ~]# ./m122
memorycopy time is 184956205 ns
[root@localhost ~]# ./m122
memorycopy time is 181029329 ns
[root@localhost ~]# ./m122
memorycopy time is 181333342 ns
```

图 3.18 4 倍展开优化的 copy122.s 的运行时间

(4) 内存突发传输方式优化 copy21.s 文件的运行

```
[root@localhost ~]# gcc time.c copy21.s -o m21
[root@localhost ~]# ./m21
memorycopy time is 49281969 ns
[root@localhost ~]# ./m21
memorycopy time is 50557321 ns
[root@localhost ~]# ./m21
memorycopy time is 74270665 ns
[root@localhost ~]# ./m21
memorycopy time is 72728085 ns
[root@localhost ~]# ./m21
memorycopy time is 53153626 ns
```

图 3.19 内存突发传输方式优化 copy21.s 的运行时间

上述过程中生成的所有文件如图 3.20

```
[root@localhost ~]# ll
total 100K
-rw-r--r-- 1 root root 0 May 2 14:14 a.out
-rw-r--r-- 1 root root 161 May 16 03:39 copy121.s
-rw-r--r-- 1 root root 227 May 16 03:45 copy122.s
-rw-r--r-- 1 root root 112 May 16 03:53 copy21.s
-rw-r--r-- 1 root root 104 May 16 03:32 copy.s
-rwxr-xr-x 1 root root 1.2K May 16 03:23 hello
-rw-r--r-- 1 root root 952 May 16 03:23 hello.o
-rw-r--r-- 1 root root 156 May 2 14:28 hello.s
-rwxr-xr-x 1 root root 70K May 16 03:49 m1
-rwxr-xr-x 1 root root 70K May 16 03:50 m121
-rwxr-xr-x 1 root root 70K May 16 03:50 m122
-rwxr-xr-x 1 root root 70K May 16 03:54 m21
drwxr-xr-x 2 root root 4.0K Feb 27 2022 test
-rw-r--r-- 1 root root 438 May 16 03:48 time.c
```

图 3.20 实验中创建的所有文件

3.使用 gdb 工具

在本次实验中，我使用 gdb 工具调试了内存拷贝函数的未优化版本的文件的可调试 exe 文件。相关步骤如下。

(1) 生成可调式的 exe 文件，并使用 gdb 工具进行调试

```
[root@localhost ~]# gcc -g time.c copy21.s -o example
[root@localhost ~]# gdb -q example
Reading symbols from example...
(gdb) █
```


汇编语言程序设计实验报告

图 3.21 进入 gdb 调试模式

(2) list 查看代码

```
[root@localhost ~]# gdb -q example
Reading symbols from example...
(gdb) list
1      #include<stdio.h>
2      #include<stdlib.h>
3      #include<time.h>
4      #define len 600000000
5      char src[len],dst[len];
6      long int len1=len;
7      extern void memcpy(char *dst,char *src,long int len1);
8      int main(){
9          struct timespec t1,t2;
10         int i,j;
(gdb)
11         for(i=0;i<len-1;i++)
12             src[i]='a';
13             src[i]=0;
14             clock_gettime(CLOCK_MONOTONIC,&t1);
15             memcpy(dst,src,len1);
16             clock_gettime(CLOCK_MONOTONIC,&t2);
17             printf("memcpy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
18             return 0;
19         }
20
(gdb)
Line number 21 out of range; time.c has 20 lines.
```

图 3.22 list 查看代码

(3) 设置断点

由图 3.22 可知，调用 memcpy 函数的语句在第十五行，故在此处设置断点。

```
(gdb)
Line number 21 out of range; time.c has 20 lines.
(gdb) b 15
Breakpoint 1 at 0x4006d8: file time.c, line 15.
```

图 3.23 设置断点

(4) 开始调试

首先输入 r 指令，程序运行，并在断点处停止。随后输入 s 指令，程序开始单步运行。我们通过输入 backtrace 指令观察函数使用情况。

```
(gdb) r
Starting program: /root/example
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.28-36.oe1.aarch64

Breakpoint 1, main () at time.c:15
15      memcpy(dst,src,len1);
(gdb) s
memcpy () at copy21.s:3
3      ldp x3,x4,[x1],#16
```

图 3.24 开始运行

```
(gdb) backtrace
#0  memcpy () at copy21.s:3
#1  0x00000000004006f8 in main () at time.c:15
```

图 3.25 函数调用

(5) 观察参数传递和程序单步运行

汇编语言程序设计实验报告

如下图,显示出函数的参数 dst,src 的地址为“0x3d58748”“0x420048”,参数 len1 值为“60000000”

```
(gdb) print &dst
$3 = (char (*)[60000000]) 0x3d58748 <dst>
(gdb) print &src
$4 = (char (*)[60000000]) 0x420048 <src>
(gdb) print len1
$5 = 60000000
```

图 3.26 函数参数的地址与值

继续单步运行,经过四轮循环后,观察 dst 的值

```
(gdb) s
3      ldp x3,x4,[x1],#16
(gdb) x dst
0x3d58748 <dst>:      0x61616161
```

图 3.27 四轮循环后

此后,程序将要退出。

3.3 小结

首先,关于 16 位、32 位、64 位程序的特点,三者主要的区别在于运行环境的机器字长不同。在操作中的直观反应就是,各类寄存器的命名都不相同了:ax,eax,rax。在不同环境中,要注意数据的取值范围,避免产生错误。

Dosbox 在 windows 系统中,可以模拟 dos 系统,在实模式下进行汇编编程。在 vs 的编译器上有汇编的功能,相比于 dosbox 更为直观、简洁。但不可否认的是,通过使用 dosbox 的 debug 功能和 td 功能,仍然是了解汇编语言的最好的方式。唯一比较遗憾的是 dosbox 中的 td 功能网上教程较少,且大多数不够准确,导致实验时花费了大量的时间进行了无结果的搜索。

QEMU 下 ARMv8 的使用。本实验需要一定的 linux 知识,同时还要学习 gdb 和 gcc 的基本指令。这是我第一次在 linux 环境下使用 gdb 语句,体验上比较有趣。同时,通过指令行进行操作的方式,也十分符合计算机学生的习惯。在 gdb 的指令中,诸如“r”指令是程序开始运行,“s”指令是程序单步运行,在调试的过程中,这些都给我留下了较深的印象。

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
- [5]王爽. 汇编语言 (第四版). 清华大学出版社, 2019
- [6]Wuliwuliii. INT 21H 指令说明及使用方法 (汇编语言学习)
https://blog.csdn.net/qg_41730082/article/details/103194044, 2019
- [7]jackailson. BIOS int 10H 中断介绍
<https://blog.csdn.net/jackailson/article/details/82619729> , 2018
- [8]Npgw. c/c++下取消结构体字节对齐方法
<https://blog.csdn.net/wanxuexiang/article/details/86658855>, 2019
- [9]東纸. 设置 Dosbox 自动挂载 https://blog.csdn.net/qg_14813265/article/details/105783915, 2020