课 程 实 验 报 告

课程名称： 计算机系统基础实验

专业班级：

学　　号：

姓　　名：

指导教师： 李海波

实验时段： 2022 年 9 月 27 日~11 月 15 日

实验地点： 南一楼 808 室

原创性声明

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

特此声明！

学生签名：

报告日期：

实验报告成绩评定：

	1	2	3	4	5	6
实验完成质量（70%），报告撰写质量（30%），每次满分 20 分。						
合计（100 分）						

备注：实验完成质量从实验目的达成程度，设计方案、实验方法步骤、实验记录与结果分析论述清楚等方面评价；报告撰写质量从撰写规范、完整、通顺、详实等方面评价。

指导教师签字：

日期：

汇 编 语 言 程 序 设 计 实 验 报 告

目录

1 汇编语言编程基础

1.1 实验内容

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

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

(3) 熟悉程序运行的基本原理;

(4) 熟悉分支、循环程序的结构及控制方法，掌握分支、循环程序的调试方法;

(5) 加深对转移指令及一些常用的汇编指令的理解;

1.2 任务 1.1 实验过程

1.2.1 实验方法说明

1. 准备上机实验环境:安装了 dosbox 虚拟机环境(版本 0.73), 源程序编辑程序采用 IDE Masm for Windows 集成实验环境 2015；汇编程序使用 MASM 6.0；连接程序使用 LINK；调试工具使用 TD。在安装过程中遇到一些问题。向指导老师李老师请教解决了把dosbox的虚拟机C盘挂载到win10系统下 D:\masm 目录下的问题（使用 mount 命令）。此外，通过在网络上学习检索，学会了通过更改配置文件改编 dosbox 虚拟机窗口大小的技巧。

2. 在 TD 的代码窗口中的当前光标下输入第一个运算式对应的两个 8 位数值对应的指令语句 MOV AH,01001101B；MOV AL,-01110010B；ADD AH,AL；观察代码区显示的内容与自己输入字符之间的关系；然后确定 CS:IP 指向的是自己输入的第一条指令的位置，单步执行三次，观察寄存器内容的变化，记录标志寄存器的结果，和执行前预计的结果进行比对；保存文件名为 001.asm。

3. 输入：MOV al,+0110011B；ADD al，+1011010B；预计结果 CF=0,ZF=0,SF=1,OF=1。输入 MOV al,-0101001B；ADD al，-1011101B；预计结果（AH）=7A，SF=0、OF=1、CF=1、ZF=0；输入；预计结果（AH）=C2，SF,OF,CF=1、ZF=0。

4. 尝试按照自己想的其他语句及输入格式等进行操作，自主设计了几个算例以研究在有符号数和无符号数运算中几个标志寄存器变化的情况，积累了更多的经验。

1.2.2 实验记录与分析

1. 实验环境条件：WINDOWS 10 下 DOSBox0.73； TD.EXE 5.0。

2. 输入指令 MOV al,+0110011B；ADD al，+1011010B。执行指令后的结果如图 3.1.1 所示。可以看出，计算结果在(AL)=8DH 与标志位的状态(CF=0,ZF=0,SF=1,OF=1)与事前预期的是一致的。

图 1.1 执行完测试语句后的状态

3. 输入指令 MOV al,-0101001B；ADD al，-1011101B。执行指令后的结果如图 1.2 所示。可以看出，计算结果在(AL)=7AH 与标志位的状态(CF=1,ZF=0,SF=0,OF=1)与事前预期的是一致的。

图 1.2 执行完测试语句后的状态

4. 输入 MOV al,+1100101B；ADD al，-1011101B。执行指令后的结果如图 1.3 所示。可以看出，计算结果 (AL)=08H 与标志位的状态(CF=1,ZF=0,SF=0,OF=0)与事前预期的是一致的。

图 1.3 执行完测试语句后的状态

5. 接下来测试减法。输入指令 MOV AH,+0110011B；SUB AH，+1011010B。执行指令后的结果如图 3.1.4 所示。可以看出，计算结果在(AH)=D9H 与标志位的状态(CF=1,ZF=0,SF=1,OF=0)与事前预期的是一致的。

图 1.4 执行完测试语句后的状态

6. 输入指令 MOV AL,-0101001B；SUB AL，-1011101B。执行指令后的结果如图 1.5 所示。可以

看出，计算结果在(AL)=34H 与标志位的状态(CF=0,ZF=0,SF=0,OF=0)与事前预期的是一致的。

图 1.5 执行完测试语句后的状态

7. 接下来探究“求差运算中，若将 A、B 视为有符号数，且 A>B, 标志位有何特点”这一问题。考虑到有符号数（大小关系已确定）的加减主要地有三种情况：正减正、负减负、正减负，本人分别涉及三对数的运算来探究这一问题。

8. 设计三对数字为+1110110，+1001000；+1110110，-1001000；-1001000，-1111010。

图 1.6 执行完测试语句后的状态

结论：求差运算中，若将 A、B 视为有符号数，且 A>B, 标志位 CF=0;

进一步观察、思考、分析发现 OF,CF,ZF 符号位的状态关系：(SFxorOF) == 0 且 ZF == 0，有 A>B。

9. 用同样方式分析 7 中问题对于无符号数的情况，得出相似结论：

结论：求差运算中，若将 A、B 视为有符号数，且 A>B, 标志位 CF=0;

还可以得到其他符号位的关系：CF == 0 且 ZF == 0 则有 A>B。

10. 思考为什么两种运算比较存在差异？可能的原因：处理器处理有符号负数时用补码进行运算。

1.3 任务 1.2 实验过程

1.3.1 实验方法说明

1. 准备上机实验环境。

2. 使用 Masm for Windows 集成实验环境编写源程序，保存文件名为 1-2.asm。使用 MASM 6.0 汇编源文件。在 dosbox 命令行窗口中运行命令 masm 1-2.asm；观察提示信息，若出错，则用编辑程序修改错误，保存后重新汇编，直至不再报错为止。

3. 使用连接程序 LINK.EXE 将汇编生成的 1-2.OBJ 文件链接成执行文件。运行命令 link 1-2.obj；

若连接时报错，则依照错误信息修改源程序。之后重新汇编和连接，直至不再报错并生成102.exe 文件。

4. 执行该程序。即在命令行提示符后输入 1-2.后回车，观察执行现象。

5. 使用 TD.EXE 观察 1-2.exe 的分步执行情况。运行命令 td 1-2.exe。

1.3.2 实验记录与分析

1. 上机实验环境说明：WINDOWS 10 系统下使用 DOSBox0.73。汇编程序使用 MASM 6.0;连接程序使用 LINK；调试工具使用 TD。

2. 汇编源程序（masm）时没有发生异常。

3. 连接过程（link）没有发生异常。

4. 用 TD 调入 102.EXE 后，F7 单步执行各个语句，每执行一条语句，观察数据段中的内容以及相应寄存器的变化，在“MOV CX，000A”和“INT 21H”设置断点，执行两遍循环体后，然后F9 直接执行到断点处。

记录执行到“MOV CX，10”和“INT 21H”之前的(BX)，(BP),(SI),(DI) 值。

图 1.7

执行到“INT 21H”之前(BX)=001EH,(BP)=0028H,(SI)=000AH,(DI)=0014H。

图 1.8

5. 在数据区右击 goto，输入 ds：0，记录程序执行到退出之前数据段开始 40 个字节的内容如下图所示，程序运行结果与设想的一致。

图 1.9

6. 解决需求:“在标号 LOPA 前加上一段程序，实现新的功能：先显示提示信息“Press any key to begin!”，然后，在按了一个键之后继续执行 LOPA 处的程序”。

图 1.10 操作成功

1.4 任务 1.3 实验过程

1.4.1 实验方法说明

1. 准备上机实验环境。

2. 使用 Masm for Windows 集成实验环境编写源程序，根据实验要求在原有的 103.asm 程序上做修改以实现目标的功能。使用 MASM 6.0 汇编源文件。在 dosbox 命令行窗口中运行命令 masm 103.asm；观察提示信息，若出错，则用编辑程序修改错误，保存后重新汇编，直至不再报错为止。

3. 使用连接程序 LINK.EXE 将汇编生成的 103.OBJ 文件链接成执行文件。运行命令 link 103.obj；

若连接时报错，则依照错误信息修改源程序。之后重新汇编和连接，直至不再报错并生成 103.exe 文件。

4. 执行该程序。即在命令行提示符后输入 103 后回车，观察执行现象是否符合要求。

5. 使用 TD.EXE 观察 103.exe 的分步执行情况。运行命令 td 103.exe。

1.4.2 实验记录与分析

1. 上机实验环境说明：WINDOWS 10 系统下使用 DOSBox0.73。汇编程序使用 MASM 6.0;连接程序使用 LINK；调试工具使用 TD。

2. 汇编源程序（masm）时没有发生异常。

3. 连接过程（link）没有发生异常。

4. 用 TD 调入 103.EXE 后,用 F9 执行到程序末尾，在数据区右击 goto，输入 ds：0，记录程序执行到退出之前数据段开始 40 个字节的内容如下图所示，不符合预期。认为应该是源程序中某处出现了错误，经过一段时间的排查发现，原因在于用变址寻址方式时错把+10（0AH）写成了“+10H”（16）。修改后，按照原定方案重新执行步骤 4。

图 1.11 执行完测试语句后的状态

图 1.12 执行完测试语句后的状态

5. 改写程序后观测到的 ds 段前四十个字节的结果与实验 1.2 中的结果相同，符合预期。

图 1.13 执行完测试语句后的状态

6. 在 TD 代码窗口中观察并记录机器指令代码在内存中的存放形式，并与 TD 中提供的反汇编语句及自己编写的源程序语句进行对照，也与任务 2 做对比。td 102.exe 如图所示。td 103.exe 如图所示。观察机器指令代码在内存中的存储方式，并与自己编写的源程序代码相比较，不难发现：

（1）START，LOPA 等都被翻译成具体的地址；

（2）源程序中的常量被翻译成了与参与运算寄存器相同类型的十六进制数据，如 mov cx,10 中的 10 被翻译成 000A;

（3）细致观察发现：相近地址对应的机器指令是相似的，如 td 102.exe 中的 mov si,bi 等对应的指令有一定的相似度；td 103.exe 中亦然。

（4）观察 td102.exe 和 td103.exe 的不同。102 和 103 程序一个重要的不同点在于 103 中用的都是 32 位的 esi 寄存器，102 中用的都是 16 位的 si 寄存器。因为寄存器位数的不同，反汇编产生的代码也有不同，但仔细观察，发现有相同的片段。

（5）变址寻址中字符串名称在反汇编中变成了偏移地址，例如 MOV 14H[ESI],AL 变成了 MOV [ESI+14],AL；

（6）如果从串的错误位置开始汇编，可能导致后续反汇编得到的指令紊乱，体现了 IP/EIP 的重要性。

1.5 任务 1.4 实验过程

1.5.1 设计思想及存储单元分配

设计一个计算机系统运行状态的监测系统，按照要求收集三个状态信息 a，b，c（均为有符号双字整型数）。假设 n 组状态信息已保存在内存中。对每组的三个数据进行处理的模型是 f=(5a+b-c+100)/128（最后结果只保留整数部分）。当 f<100 时，就将该组数据复制到 LOWF 存储区，当 f=100 时，就将该组数据复制到 MIDF 存储区，当 f>100 时，就将该组数据复制到 HIGHF 存储区。

设计思想：根据数据在内存中的先后地址来访问每个状态信息的数值，以一组数据为每次处理的一个循环，在循环中实现结果的运算、写入以及内存单元的拷贝。数据定义方式：SAMID DB 9 DUP(0)；每组数据的流水号（可以从 1 开始编号）SDA DD 256809；状态信息 aSDB DD -1023；状态信息 bSDC DD 1265；状态信息 cSF DD ？；处理结果 f 在数据段中定义字长为 120 个字节的三段存储空间，分别存储处理结果大于、小于等于 100 的数据的各项信息。存储单元分配：如以上数据定义方式说明，每组数据占据的存储空间为 25 字节，一共设计十组数据，占 250 字节空间；三段用于存储拷贝数据的存储空间分别为 120 字节，总共 360 字节；此外还有三个 DW 型变量用来标识每段存储空间的拷贝起始位置，一共占 6 个字节，故数据段共使用 616 字节的存储空间。

1.5.2 实验步骤说明

1.准备上机实验环境。

2.使用 masm for windows 集成开发环境录入源程序，存盘文件名为 104.ASM。使用 MASM 6.0 汇编源文件。即 MASM 104；观察提示信息，若出错，则用编辑程序修改错误，存盘后重新汇编，直至不再报错为止。

3.使用连接程序 LINK.EXE 将汇编生成的 104.OBJ 文件连接成执行文件。即 LINK 104；若连接时报错，则依照错误信息修改源程序。之后重新汇编和连接，直至不再报错并生成 104.EXE 文件。

4.执行该程序。查看该程序能否正常执行结束。

5.使用 TD.EXE 观察 104.exe 的执行情况。即 TD 104.EXE 回车

（1）观察被调试的程序是否与自己编写的 asm 源程序一致。

（2）单步执行开始 2 条指令，观察 DATA 的实际值，以及 DS 的改变情况。

（3）观察 SS：0 至 SS：SP 区域的数据值。

（4）断点执行到 cmp 前的一条指令，观察 DS：0 开始数据区，找到各变量在数据段中的位置和值。

（5）将 ax 寄存器的值与预想中的数值比较。

（6）执行到程序结束中断前，在下方内存观察窗口观察内存从 ds:0 开始各个字节的情况，与

预想中的数值比较。

1.5.3 实验记录与分析

1.实验环境条件:上机实验环境说明:Linux 系统下使用 DOSBox0.73。汇编程序使用 MASM 6.0;连接程序使用 LINK；调试工具使用 TD。

2.汇编源程序时，源程序没有发生异常。这是由于 masm 之前，在集成开发环境上先运行了几次，消除了语法错误。

3.连接过程没有发生异常。

4.该程序可以执行结束。

5.用 TD 调入程序后

（1）观察到开始几条指令与预想中不同，类似于赋值语句而非期望中的语句：

图

图 1.14

排查错误语句经过检查发现，这是写错了一条指令导致的，虽然汇编链接时没有报错，但会影响程序得到正确结果。修改后正确。

（2）单步执行开始 2 条指令。

图 1.15 单步执行两步

（3）观察 SS：0 至 SS：SP 区域的数据值。

图 1.16 观察 SS 区的值

（4）断点执行到 cmp 前的一条指令，观察 DS：0 开始数据区，找到各变量在数据段中的位置和值。可以看到，前九个字节都为 0，这是正确的。前 25 个字节是第一组数据的各个数据，都与预想中相同。

图 1.17 计算第一组数据后的 data 段

（5）观察 ax 寄存器的值，这是程序主要的计算结果。第一组数据计算结果为 0064H，即十进制下的 100，是正确的。

图 1.18 计算第二组数据前的 data 段

（6）继续断点调试，执行到第二次 cmp 指令前。观察对应数据区发现，第一组数据的各个数

据都已经成功写入内存区域，位于 0172H，与预想计算的数值（250+120）相同。证明操作正确。

图 1.19 计算第二组数据后的 data 段

（7)接下来多次执行,直到程序终止前。我们观察 ds 对应的内存区域,分别到 HIGHF/MIDF/LOWF 对应的存储区域中观察数据是否写入。结果符合预期，程序实现了设计目的。

图 1.20 程序退出前的 data 段

图 1.21 程序退出前的 data 段

1.6 小结

1.6.1 实验完成情况

成功安装并且运行了汇编程序的编辑，编译，调试工具，使用 td 完成了对 1.14 题中程序的调试，同时对程序中的变量以及标志位进行了观察跟踪。

完成了汇编程序的改写，初步了解了汇编程序的编写格式，运行了自己写的第一个汇编程序，并用 td 对该程序进行了调试。

完成了数据处理程序的设计，能够实现实验要求的功能，同时运行正常。

基本完成了实验的要求。

1.6.2 实验收获

掌握了汇编源程序编辑工具、汇编程序、连接程序、调试工具 TD 的使用，学会了如何将汇编程序编译为可运行程序以及如何使用 TD 对汇编程序调试；

初步理解了数、符号、寻址方式等在计算机内的表现形式，以及汇编语言下如何合理的调用寄存器以及寄存器所存储的内容；

理解了指令执行与标志位改变之间的关系；

掌握并应用了常用的 DOS 功能调用；

掌握了如何使用汇编语言编写分支、循环程序的结构及控制方法，掌握了分支、循环程序的调试方法，成功编写出一个数据处理程序。

2 程序优化

2.1 实验内容

（上机实验环境说明：本次实验需要调用系统函数，例如 printf 和 clock 函数。因此本次实验使用 VS2019 开发汇编程序和 C 程序。(使用 VS2019 开发汇编程序的操作，参见:许向阳.x86 汇编语言程序设计,第 19 章)

任务 2.1 对任务 1.4 的程序进行优化在任务 1.4 描述的背景下，假设在输入缓冲区中已经存放了 N 组采集到的状态信息，需要对这 N 组数据分别计算对应的 f 并依据分组原则将 N 组数据复制到对应的 LOWF、MIDF、HIGHF 存储区中。优化工作包括代码长度的优化和执行效率的优化等等（本次以执行效率/性能的优化为主）。请尝试对 f 的计算过程以及数据的复制过程的代码进行优化，通过对这些代码执行时间的计时来判断优化的效果。

为体现程序的优化效果，对任务 1.4 的程序主体，重复执行 m 次。m 的大小可以自行设定，直到总的执行时间和优化时间有明显的效果。在程序主体执行前使用 clock 函数计时，计时结果保存在变量 start_t 中。在程序主体执行结束后再次调用 clock 计时，结果保存在 end_t 中。调用 printf 输出(end_t-start_t)，即程序主体的执行时间。分别实现未优化和优化的程序，对比两者的执行时间。要求优化效果提高 10%以上。

2.2 任务 2.1 实验过程

2.2.1 实验方法说明

1.准备上机实验环境:Visual studio 2019。

2.使用 Visual studio 2019 编写源程序，保存解决方案（项目）名为 test，修改项目-生成依赖项，添加依赖项为 masm,之后编辑并命名汇编文件为 2-1.asm，修改 Debug 生成模式为 x86。

3.对在 dosbox 实模式下的汇编程序进行适当修改，使其能在 visualstudio 的保护模式下运行并调用 C 语言中的标准库函数实现输出、计时等功能。

4.使用调试-直接运行不调试运行程序；若运行程序时报错，则依照错误信息查找资料，向老师、同学请教，修改源程序。之后重新汇编和连接，直至不再报错并在 Debug 目录下生成 test.exe 文件。

5.使用调试-逐步调试进入调试界面，通过汇编窗口下的反汇编等功能观察反汇编代码等。

6.运行程序，记录程序执行时间。

7. 修改程序中的部分语句后，记录程序执行时间，实现程序运行时间上的优化。

2.2.2 记录与分析

1.把在 dosbox 上的程序直接移植到 VISUAL STUDIO 上的解决方案中，运行时出现大量报错信息。经检查发现，这些错误是由于在程序中使用的大量十六位寄存器没有改为三十二位寄存器导致的。在 dosbox 的实模式（16 位）下能运行的程序在 visual studio 的保护模式（32 位）下不一

定能够运行，将所有寄存器都改成 32 位寄存器后错误解决。

2.编写了用于记录程序运行时间的计时函数（基于 C 标准库中的 timeGetTime 函数）。运行程序后发现运行时间为 0ms。经反复检查代码逻辑无问题，该现象是由于程序运行时间太短造成的。在源程序以外再套上一个循环，让程序执行 1000000 次，得到了比较明显的运行时间记录。

图 2.1 初始程序的时间性能改编前的源程序：2857ms

3.第一次改编尝试：把除以 128 的指令改成逻辑右移 7 位的指令。

图 2.2 第一次改编后的时间性能

4.第二次改编尝试：把乘 5 的指令改成逻辑左移两位后再加一倍，

图 2.3 第二次改编后的时间性能

5.第三次改编尝试：把乘 16 指令改成位运算，乘 5 的指令改成原始的乘法指令。

图 2.4 第三次改编后的时间性能

2.3 小结

2.3.1 实验完成情况

将在 Linux 系统下，在 dosbox 上可以进行编译运行的汇编程序，经过对部分结构的改写后，成功在 VISUAL STUDIO 环境下运行。

解决了本任务中程序计时的问题，由于程序运行速度过快，因此通过增加程序运行次数，成功实现了程序运行的计时功能，并且能够比较改写部分语句后的程序速度。

实现了几种优化汇编程序的方法，并进行了多次测量，比较了不同写法下程序的运行速度，对不同写法进行了评估。

基本完成了任务要求的优化功能。

2.3.2 实验收获

通过本次实验，我掌握了在汇编程序中调用 C 语言库函数的方法，其中调用了计时函数，实现了对程序的计时。

掌握了对汇编程序代码进行优化的基本方法，并且根据该程序的实际情况，选择了几种优化方法成功对该程序进行了优化。

对汇编语言中的指令语句，以及寄存器的使用有了更加深入的理解。

3 模块化程序设计

3.1 实验内容

（1）掌握子程序设计的方法与技巧，熟悉子程序的参数传递方法和调用原理；

（2）掌握模块化程序的设计方法;

（3）掌握如何使用 C 语言与汇编语言混合编程；

（4）掌握较大规模程序的开发与调试方法；理解模块之间的信息传递与组装的基本方法，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；

（5） 完成指定功能的程序设计与调试。

3.2 任务 3.1 实验过程

3.2.1 实验方法说明

1. 准备上机实验环境。

2. 使用 VS2019 集成实验环境编写源程序，根据实验要求在以实验 2 中的程序作为子程序，编写一个较为完整的系统汇编程序。将不同功能的程序在不同的文件中编写，最终使主程序实现满足要求的功能。

3. 观察提示信息，若出错，则用编辑程序修改错误，保存后重新运行，直至不再报错为止。

4. 执行该程序，进行功能测试。

3.2.2 实验记录与分析

1. 上机实验环境说明：WINDOWS 10 系统下使用 Visual Studio 2019。

2. 编译过程中未发生异常

3. 错误输入用户名和密码三次，测试报错功能。其中以子程序的形式编写了 StrCmp 函数进行字符串比较，并在内部声明了局部变量 u，并用 u 控制了该程序内部循环的终止，局部变量 u 存储于栈中，其地址表达式为使用 esp 或 ebp 的变址表达式。为 StrCmp 传参时，使用了 invoke 伪指令进行动态传参，参数以存入堆栈的形式进行传递，堆栈栈顶存放子程序出口地址，下面存放参数地址。CPU 执行 CALL 指令时，将当前 IP 压入栈中，而执行 RET 指令时，将栈顶数据弹出到 IP 中，从而完成子程序的进入与弹出。

图 3.1 测试 3.1 报错功能 1

图 3.2 测试 3.1 报错功能 2

4. 重新运行程序，输入正确的用户名和密码，程序正确执行。

图 3.3 测试 3.1 检验功能

5. 多次输入’r’，测试显示功能，功能正常。成功显示了 MIDF 存储区中的内容，其中显示功能使用的是 C 语言的 printf 进行实现，该函数在汇编程序中使用 invoke 进行调用。Invoke 对应的汇编语句分别有，将参数入栈 PUSH 以及调用子程序 CALL。

图 3.4 测试 3.1 显示功能

6. 输入’q’，测试退出功能，功能正常。

图 3.5 测试 3.1 退出功能

3.3 任务 3.2 实验过程

3.3.1 实验方法说明

1. 准备上机实验环境。
2. 应用 C 语言与汇编语言混合编程，改造在(2)中实现的程序，在 C 语言中 声明变量与函数，并在汇编程序中进行调用。
3. 观察提示信息，若出错，则用编辑程序修改错误，保存后重新运行，直至不再报错为止。
4. 执行该程序，进行功能测试。

3.3.2 实验记录与分析

1. 上机实验环境说明：WINDOWS 10 系统下使用 Visual Studio 2019。
2. 编译过程中未发生异常
3. 错误输入用户名和密码三次，测试报错功能。

图 3.6 测试 3.2 报错功能 1

图 3.7 测试 3.2 报错功能 2

4. 重新运行程序，输入正确的用户名和密码，程序正确执行。

图 3.8 测试 3.2 检验功能

5. 使用键盘输入’r’，测试显示功能，功能正常，并计算出计算过程所消耗的时间。

```
r
00000064
00000a28
00000000
0000012c
00000064
00000000
0000319c
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000

Time elapsed is 211 ms

 Waiting
```

图 3.9 测试 3.2 显示功能

6. 输入’m’，测试修改功能，输入数据 0，12800，100，使 f=100，成功修改 MIDF 中的数据，功能正常，。

```
m

Insert data:
0
0
Insert data:
12800
12800
Insert data:
100
100Waiting
r
00000064
00000000
00003200
00000064
00000064
00000a28
00000000
0000012c
00000064
00000000
0000319c
00000000
00000000
00000000
00000000
00000000

Time elapsed is 214 ms
```

图 3.10 测试 3.2 修改功能

7. 输入’q’，测试退出功能，功能正常。

```
 Waiting
q

E:\test\Debug\test.exe (进程 13648)已退出，代码为 0。
```

图 3.11 测试 3.2 退出功能

3.4 小结

3.3.3 思考问题

实验 3.1 以子程序的形式编写了 StrCmp 函数进行字符串比较，并在内部声明了局部变量 u，并用 u 控制了该程序内部循环的终止，局部变量 u 存储于栈中，其地址表达式为使用 esp 或 ebp 的变址表达式。

为 StrCmp 传参时，使用了 invoke 伪指令进行动态传参，参数以存入堆栈的形式进行传递，堆栈栈顶存放子程序出口地址，下面存放参数地址。

CPU 执行 CALL 指令时，将当前 IP 压入栈中，而执行 RET 指令时，将栈顶数据弹出到 IP 中，从而完成子程序的进入与弹出。

Invoke 对应的汇编语句分别有，将参数入栈 PUSH 以及调用子程序 CALL。

3.3.4 实验完成情况

实验 3.1 实现的功能为：用户名与密码输入，字符串验证，报错功能，数据处理功能，数据显示功能，程序循环功能，程序退出功能。

字符串验证功能和数据处理功能使用了子程序进行实现，数据显示功能使用 invoke 调用 C 语言的 printf 进行实现，使其打印传入的参数。

实验 3.2 实现的功能为：用户名与密码输入，字符串验证，报错功能，数据处理功能，数据处理计时功能，数据修改功能，数据显示功能，程序循环功能，程序退出功能。

实验 3.2 以 3.1 为基础，主要实现了将数据迁移到 C 语言程序中，并在 C 语言中实现了数据修改功能，在汇编程序中对其进行了调用。

实验要求的功能基本已经实现，并对部分问题进行了思考回答。

3.3.5 实验收获

掌握了子程序设计的方法与技巧，成功编写了用于字符串比较的 CmpStr 与用于登录功能的 LOGIN 子程序，熟悉了子程序的参数传递方法和调用原理，调用了 C 语言中的 printf 函数，并进行传参，同时回答了关于 invoke 传参的相关问题；

掌握了模块化程序的设计方法，将一个程序拆分成了多个子程序，并放在了不同的文件中，从而编写出一个较为稳固的系统程序;

掌握了汇编语言程序与 C 语言程序混合编程的方法，本实验中同时使用了 C 语言和汇编语言，并且变量和函数都实现了 C 语言和汇编语言的互相声明与调用；

掌握了较大规模程序的开发与调试方法，本实验设计的是一个相对完整的系统程序，从整个项目着手容易出错，因此实际操作过程中，先编写各个功能的程序，并编写响应代码对功能进行测试，将局部功能全部较好的完成之后，才编写 MAIN 程序对其进行汇总；

基本实现了实验所要求的全部功能，功能全部正常，并且符合预期。

4 二进制炸弹破解

4.1 实验内容

（1）熟悉动态与静态反汇编工具 gdb；
（2）熟悉程序的机器级表示，掌握逆向工程的原理与技能；
（3）破解程序 bomb，提升对计算机系统的理解与分析能力。

4.2 实验方法说明

1. 准备上机实验环境。
2. 使用环境为 CentOS7 + gdb。
3. 反汇编得到 bomb 的汇编程序 txt 文件，使用 gdb 调试 bomb，结合汇编代码进行分析。
4. 执行该程序，破译密码。

4.3 任务 1 实验过程

1. 找到炸弹爆炸的代码

```
08048b20 <phase_1>:
 8048b20:    55                      push   %ebp
 8048b21:    89 e5                   mov    %esp,%ebp
 8048b23:    83 ec 08                sub    $0x8,%esp
 8048b26:    8b 45 08                mov    0x8(%ebp),%eax
 8048b29:    83 c4 f8                add    $0xfffffff8,%esp

// put two string into stack and compare them in <strings_not_equal>
// EAX stroe the 0x8(%ebp) which is what we entered.
// and the string in 0x80497c0 is 'Public speaking is very easy.'
 8048b2c:    68 c0 97 04 08          push   $0x80497c0
 8048b31:    50                      push   %eax
 8048b32:    e8 f9 04 00 00          call   8049030 <strings_not_equal>
 8048b37:    83 c4 10                add    $0x10,%esp
 8048b3a:    85 c0                   test   %eax,%eax
 8048b3c:    74 05                   je     8048b43 <phase_1+0x23>
 8048b3e:    e8 b9 09 00 00          call   80494fc <explode_bomb>
 8048b43:    89 ec                   mov    %ebp,%esp
 8048b45:    5d                      pop    %ebp
 8048b46:    c3                      ret
 8048b47:    90                      nop
```

图 4.1 任务 1 炸弹爆炸代码

2. 分析炸弹爆炸的条件，当 eax 的值为 0 时，炸弹不爆炸

```
8048b3a:    85 c0                   test   %eax,%eax
8048b3c:    74 05                   je     8048b43 <phase_1+0x23>
```

图 4.2 任务 1 炸弹爆炸条件

3. 分析 eax 的值，把我们输入的参数放进%eax 中，然后放进(%esp)，再调用函数<strings_not_equal>，如果输入的内容与传入的字符串相等，则返回 0，这个就是 eax 为 0 的条件。

```
8048b31:    50                          push   %eax
8048b32:    e8 f9 04 00 00              call   8049030 <strings_not_equal>
```

图 4.3 任务 1 分析 eax 的值

4. 找出 strings_not_equal 的参数，即地址 0x80497c0 中的内容

```
8048b2c:    68 c0 97 04 08              push   $0x80497c0
8048b31:    50                          push   %eax
```

图 4.4 任务 1 找到函数的参数

4. 进入 gdb 调试模式，输出 0x80497c0 中的内容，得到答案

```
(gdb) x/s 0x80497c0
0x80497c0:      "Public speaking is very easy."
```

图 4.5 任务 1 答案

5. 验证答案

```
(gdb) s
Single stepping until exit from function __kernel_vsyscall,
which has no line number information.
Public speaking is very easy.
0xf7edf953 in __read_nocancel () from /lib/libc.so.6
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
```

图 4.6 验证任务 1 答案

4.4 任务 2 实验过程

1. 找到炸弹爆炸的代码

```
8048b69:    e8 8e 09 00 00              call   80494fc <explode_bomb>
8048b83:    e8 74 09 00 00              call   80494fc <explode_bomb>
```

图 4.7 任务 2 炸弹爆炸代码

2. 分析炸弹爆炸的条件，第一个条件需要(ebp-0x18)中的值为 1，第二个条件需要(esi+4*ebx-0x4)*eax 与 esi+4*ebx 的值相等

```
8048b63:    83 7d e8 01                 cmpl   $0x1,-0x18(%ebp)
8048b67:    74 05                       je     8048b6e <phase_2+0x26>
```

图 4.8 任务 2 炸弹爆炸条件 1

```
8048b79:    0f af 44 9e fc           imul   -0x4(%esi,%ebx,4),%eax
8048b7e:    39 04 9e                 cmp    %eax,(%esi,%ebx,4)
8048b81:    74 05                    je     8048b88 <phase_2+0x40>
8048b83:    e8 74 09 00 00           call   80494fc <explode_bomb>
```

图 4.9 任务 2 炸弹爆炸条件 2

3. 分析(ebp-0x18)的值，分析子程序 read_six_numbers 可知，(ebp-0x18)为输入的六个数字中的第一个数字，且必须输入六个数字，否则该子程序中内置的爆炸函数会运行。

```
8048b5b:    e8 78 04 00 00           call   8048fd8 <read_six_numbers>
```

图 4.10 任务 2 分析 ebp-0x18 的值

4. 分析 eax 中的值，该段程序遍历了输入的六个数字，每次遍历比较 eax*(i) 与 (i+1)，其中 eax 从 1 开始，并且再每次循环开始前加 1。由第一个数字为 1，可以推出第二个数字为 1*2=2，第三个数字为 2*3=6，第四个数字为 6*4=24，第五个数字为 24*5=120，第六个数字为 120*6=720.

```
 8048b76:    8d 43 01                 lea    0x1(%ebx),%eax
// compare (esi+4*ebx-0x4)*eax and esi+4*ebx
// a begin from 2
// answer is '1 2 6 24 120 720'
 8048b79:    0f af 44 9e fc           imul   -0x4(%esi,%ebx,4),%eax
 8048b7e:    39 04 9e                 cmp    %eax,(%esi,%ebx,4)
 8048b81:    74 05                    je     8048b88 <phase_2+0x40>
 8048b83:    e8 74 09 00 00           call   80494fc <explode_bomb>
// ebx++
 8048b88:    43                       inc    %ebx
 8048b89:    83 fb 05                 cmp    $0x5,%ebx
 8048b8c:    7e e8                    jle    8048b76 <phase_2+0x2e>
 8048b8e:    8d 65 d8                 lea    -0x28(%ebp),%esp
```

图 4.11 任务 2 分析 eax 的值

5. 验证答案

```
Continuing.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
```

图 4.12 任务 2 答案

4.5 任务 3 实验过程

1. 分析第一个爆炸点，若 eax 大于等于 0x2，则爆炸

```
8048bbf:    83 f8 02                cmp    $0x2,%eax

8048bc2:    7f 05                   jg     8048bc9 <phase_3+0x31>
8048bc4:    e8 33 09 00 00          call   80494fc <explode_bomb>
```

图 4.13 任务 3 炸弹爆炸代码 1

2. 分析 eax 的值，分析调用的函数 sscanf@plt 可知，eax 为函数的返回值，需要该函数正常运行。其中函数为数据输入，输入的格式为 edx 中的内容，使用 gdb 打印 0x80497de 即 edx 中存放的内容”%d %c %d”可知，需要输入数字，字符，数字。

```
// $0x80497de = "%d %c %d"
// breakpoint
 8048bb1:    68 de 97 04 08          push   $0x80497de
 8048bb6:    52                      push   %edx
// scanf %d %c %d
 8048bb7:    e8 a4 fc ff ff          call   8048860 <sscanf@plt>
 8048bbc:    83 c4 20                add    $0x20,%esp
// compare return value
 8048bbf:    83 f8 02                cmp    $0x2,%eax
```

图 4.14 任务 3 分析 eax

3. 分析输入数据的存放地址，函数 sscanf@plt 使用堆栈法进行传参，(ebp-0xc)为输入的第一个数据，(ebp-0x5)为输入的第二个数据，(ebp-0x4)为输入的第三个数据

```
// push 3 data
 8048ba5:    8d 45 fc                lea    -0x4(%ebp),%eax
 8048ba8:    50                      push   %eax
 8048ba9:    8d 45 fb                lea    -0x5(%ebp),%eax
 8048bac:    50                      push   %eax
 8048bad:    8d 45 f4                lea    -0xc(%ebp),%eax
 8048bb0:    50                      push   %eax
```

图 4.15 任务 3 数据地址

4. 分析第一个数据的要求，(ebp-0xc)为输入的第一个数据，当 0x7>第一个数据时，炸弹爆炸，同时由于为无符号数比较，第一个数不能为负数。

```
// $ebp-0xc is the first input. The first data must below 7
 8048bc9:    83 7d f4 07             cmpl   $0x7,-0xc(%ebp)
 8048bcd:    0f 87 b5 00 00 00       ja     8048c88 <phase_3+0xf0>
```

图 4.16 任务 3 第一个数据

5. 第一个数据的作用，根据数据 1 的不同，程序会跳转到不同的程序段，即该程序为一个 switch 程序，在每一段程序中会对第二、三个数据进行不同的比较，即二、三数据与第一个数据相关，该任务有多个答案。

```
// jump to 0x80497e8+4eax = 0x80497e8+4*first data
 8048bd3:    8b 45 f4                  mov    -0xc(%ebp),%eax
 8048bd6:    ff 24 85 e8 97 04 08      jmp    *0x80497e8(,%eax,4)
 8048bdd:    8d 76 00                  lea    0x0(%esi),%esi
```

图 4.17 任务 3 数据 1 作用

6. 分析第一个数据为 0 时的情况，首先(ebp-0x4)需要与 0x309 相等，因此第三个数据的值为 777，其次(ebp-0x5)需要与 0x71 相等，即数据 2 为’q’。

```
// 0 q 777
 8048be0:    b3 71                     mov    $0x71,%bl
 8048be2:    81 7d fc 09 03 00 00      cmpl   $0x309,-0x4(%ebp)
 8048be9:    0f 84 a0 00 00 00         je     8048c8f <phase_3+0xf7>
 8048bef:    e8 08 09 00 00            call   80494fc <explode_bomb>
 8048bf4:    e9 96 00 00 00            jmp    8048c8f <phase_3+0xf7>
// jump from switch
 8048c8f:    3a 5d fb                  cmp    -0x5(%ebp),%bl
 8048c92:    74 05                     je     8048c99 <phase_3+0x101>
 8048c94:    e8 63 08 00 00            call   80494fc <explode_bomb>
```

图 4.17 任务 3 数据 1 为 0 时

7. 验证答案。

```
That's number 2.  Keep going!
0 q 777
Halfway there!
```

图 4.18 任务 3 答案

4.6 任务 4 实验过程

1. 分析炸弹爆炸条件 1，其中 sscanf@plt 与任务 3 中相同，0x8049808 为”%d”，可以判断出本任务的输入为一个数字，且该数需要大于 0。

```
 8048cf0:    68 08 98 04 08            push   $0x8049808
 8048cf5:    52                        push   %edx
 8048cf6:    e8 65 fb ff ff            call   8048860 <sscanf@plt>
 8048cfb:    83 c4 10                  add    $0x10,%esp
 8048cfe:    83 f8 01                  cmp    $0x1,%eax
 8048d01:    75 06                     jne    8048d09 <phase_4+0x29>

'/ paramter must > 0
 8048d03:    83 7d fc 00               cmpl   $0x0,-0x4(%ebp)
 8048d07:    7f 05                     jg     8048d0e <phase_4+0x2e>
 8048d09:    e8 ee 07 00 00            call   80494fc <explode_bomb>
```

图 4.19 任务 4 炸弹爆炸代码 1

2. 分析炸弹爆炸的条件 2，当 eax 的值不等于 0x37(55)时，炸弹爆炸，而 eax 的值显然与调用

的方法 func4 相关，而 func4 上 push 了一个参数，显然又与 func4 相关

```
8048d14:    50                      push   %eax
8048d15:    e8 86 ff ff ff          call   8048ca0 <func4>
8048d1a:    83 c4 10                add    $0x10,%esp
// fibonacci(9) = 55 = 0x37
8048d1d:    83 f8 37                cmp    $0x37,%eax
8048d20:    74 05                   je     8048d27 <phase_4+0x47>
8048d22:    e8 d5 07 00 00          call   80494fc <explode_bomb>
```

图 4.20 任务 4 炸弹爆炸代码 2

3. 分析 func4，其中 push 入栈的 eax 的值，作为参数放到了 ebx 中，将程序拆分后可以发现该程序为一个递归程序，当参数小于等于 1 时调出，否则将参数分别减 1 和减 2 后再次调用 func4，并且将函数的返回值以 add 的方式放在 eax 中。可以发现 func4 实现了 f(x)=f(x-1)+f(x-2)即 fabonacci 数列，其中 f(0)=1,f(1)=1,f(2)=2,f(3)=3,f(4)=5,f(5)=8,f(6)=13,f(7)=21,f(8)=34,f(9)=55。

```
08048ca0 <func4>:
 8048ca0:    55                      push   %ebp
 8048ca1:    89 e5                   mov    %esp,%ebp
 8048ca3:    83 ec 10                sub    $0x10,%esp
 8048ca6:    56                      push   %esi
 8048ca7:    53                      push   %ebx

// ebx <= 1 break
 8048ca8:    8b 5d 08                mov    0x8(%ebp),%ebx
 8048cab:    83 fb 01                cmp    $0x1,%ebx
 8048cae:    7e 20                   jle    8048cd0 <func4+0x30>

// eax-1 and recursion
 8048cb0:    83 c4 f4                add    $0xfffffff4,%esp
 8048cb3:    8d 43 ff                lea    -0x1(%ebx),%eax
 8048cb6:    50                      push   %eax
 8048cb7:    e8 e4 ff ff ff          call   8048ca0 <func4>

// eax-2 and recursion
 8048cbc:    89 c6                   mov    %eax,%esi
 8048cbe:    83 c4 f4                add    $0xfffffff4,%esp
 8048cc1:    8d 43 fe                lea    -0x2(%ebx),%eax
 8048cc4:    50                      push   %eax
 8048cc5:    e8 d6 ff ff ff          call   8048ca0 <func4>

 8048cca:    01 f0                   add    %esi,%eax
 8048ccc:    eb 07                   jmp    8048cd5 <func4+0x35>
 8048cce:    89 f6                   mov    %esi,%esi
 8048cd0:    b8 01 00 00 00          mov    $0x1,%eax
 8048cd5:    8d 65 e8                lea    -0x18(%ebp),%esp
 8048cd8:    5b                      pop    %ebx
 8048cd9:    5e                      pop    %esi
 8048cda:    89 ec                   mov    %ebp,%esp
 8048cdc:    5d                      pop    %ebp
 8048cdd:    c3                      ret
 8048cde:    89 f6                   mov    %esi,%esi
```

图 4.21 任务 4 func4 代码

4. 得到答案，可知当 eax 的值为 0x37(55)时，答案不爆炸，因此 func4 运行的结果应为 0x37，因此 func4 的参数应为 9，所以任务 4 的答案为 9

```
// fibonacci(9) = 55 = 0x37
 8048d1d:    83 f8 37                cmp    $0x37,%eax
 8048d20:    74 05                   je     8048d27 <phase_4+0x47>
 8048d22:    e8 d5 07 00 00          call   80494fc <explode_bomb>
```

图 4.22 任务 4 答案

5. 验证答案

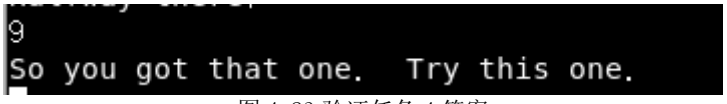

图 4.23 验证任务 4 答案

4.7 小结

4.7.1 实验完成情况

学习并掌握了 Linux 系统下，调试工具 gdb 的使用，包括运行程序，暂停程序，单步运行，设置断点，打印变量及寄存器，查看当前堆栈，显示源码，查看地址，监视内存等功能。

掌握了如何对一个程序进行反汇编，得到其机器级的代码，从而破译程序。

本实验共有 6 个任务，其中完全解决了 4 个任务，并对每个任务的汇编代码有了全面深入的理解，完全理解了代码运行的逻辑以及结果，知道每一行代码的作用。

总体而言，本次实验较好的完成了。

4.7.2 实验收获

学会了如何在 Linux 系统下使用 gdb 调试工具。

通过破译程序 bomb，加强了我对汇编程序的理解，是一次对汇编语言很好的应用，本来一些不熟悉的汇编指令，在实际应用分析后，明白了汇编语言指令的运行过程，同时也对汇编中变量的存储，以及寄存器的应用有了全新的理解

通过本次的练习，我的汇编语言能力获得了很好的锻炼，对于一些重要知识点（如跳转表，循环）的知识点，掌握的更加牢靠。而本实验中包含的许多有趣实用的汇编语言技巧（如一些精巧的中间变量的使用、灵活的 jump 跳转指令的运用）使我更加注意编程技巧的学习。

5 Linux 和鲲鹏环境编程

5.1 实验内容

（1）了解 ARM + Linux 环境下程序设计的特点及配套的开发工具；

（2）观察并理解 80X86+windows 和 ARM+Linux 下，不同的“指令集体系结构+编程环境”的基本特点。

（3）安装 QEMU 等环境，编译执行示例程序，完成鲲鹏开发环境的测试。

5.2 实验方法说明

1. 准备上机实验环境，在 WINDOWS10 系统下安装 QEMU 虚拟机；准备 openEuler 20.03 操作系统； 配套的编辑器、编译器、调试器等。

2. 按照说明对 qemu 环境进行配置。

3. 在 qemu 环境下执行示例程序，测试环境。

5.3 任务 5.1 实验过程

1. 环境变量配置。

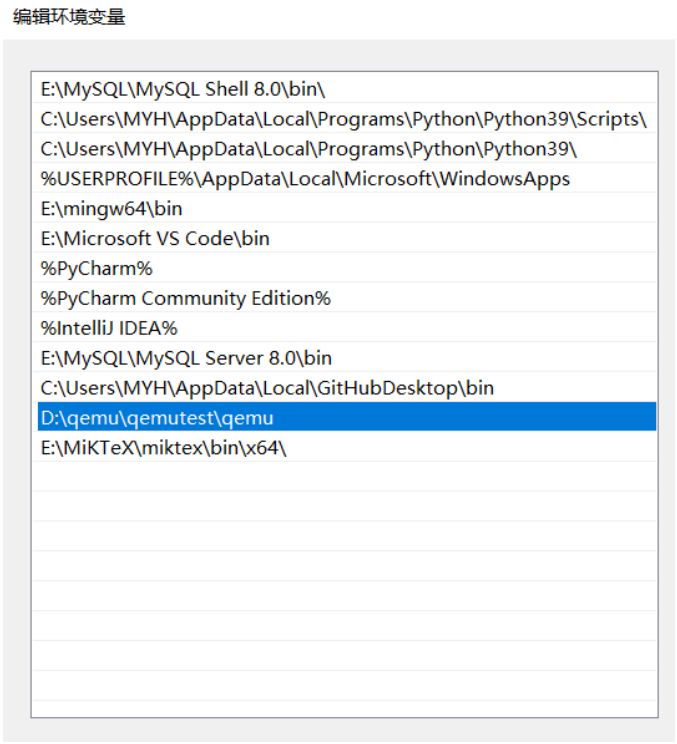

图 5.1 配置 qemu 环境变量

2. 解压 qemu 环境安装包，输入启动指令” qemu-system-aarch64 -m 4096 -cpu cortex-a57

-smp 4 -M virt -bios edk2-aarch64-code.fd -hda openEuler-20.03-LTS.aarch64.qcow2 -serial vc:800x600”启动环境。

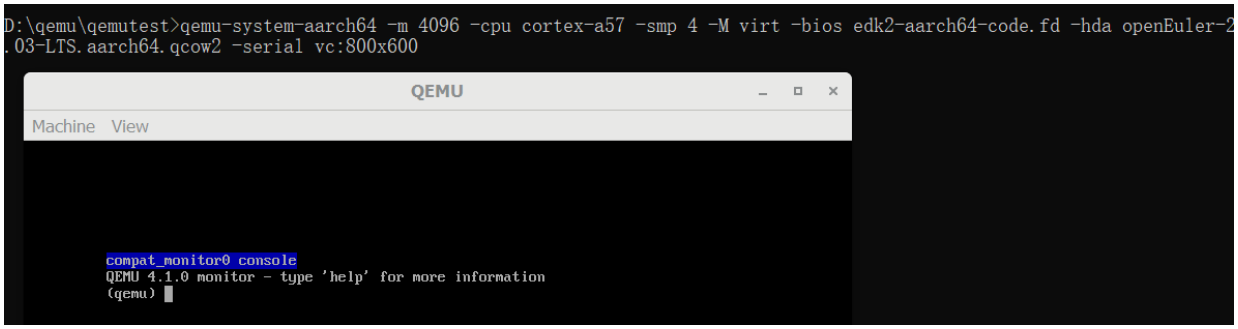

图 5.2 启动 qemu 环境

3. 输入初始的用户名”root”和密码”openEuler12#$”,登录系统。

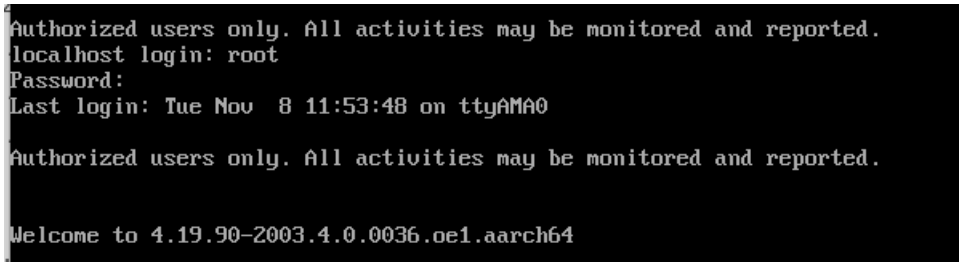

图 5.3 登录 openEuler 虚拟机

4. 网络链接测试。输入命令”ifup eth0”，打开网口。在输入命令”ifconfig”检测网络配置。

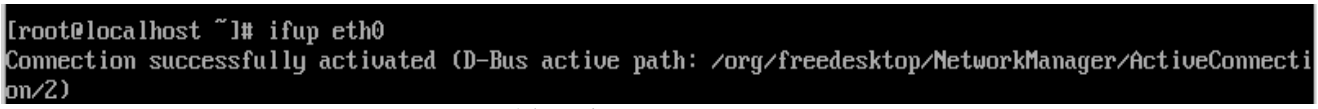

图 5.4 打开网口

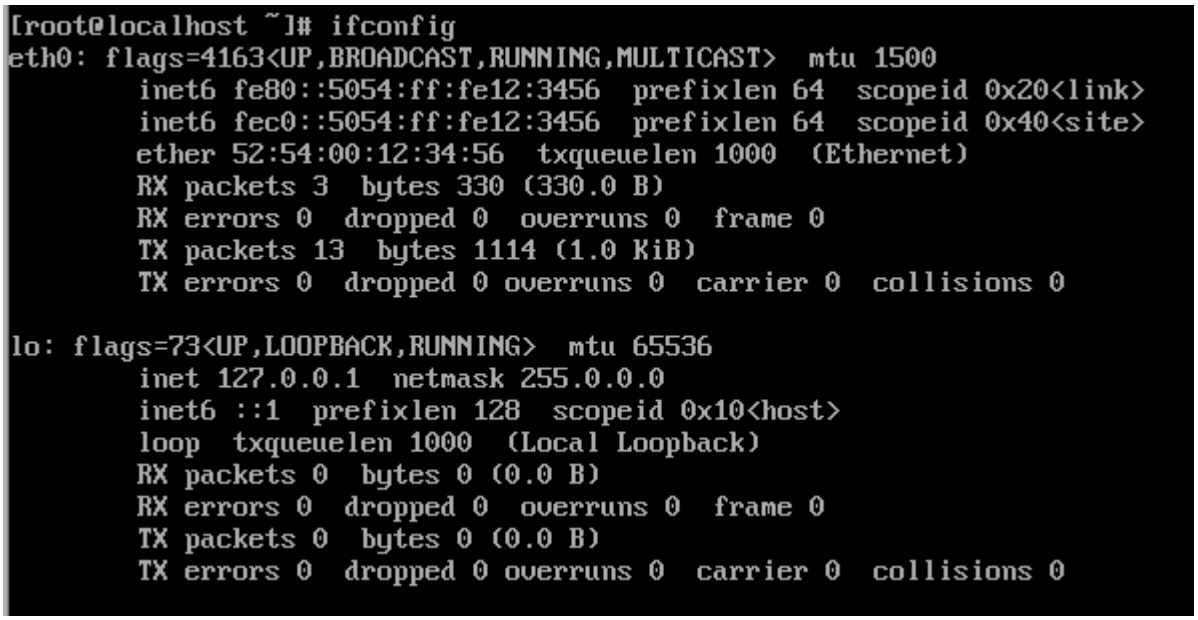

图 5.5 IP 配置成功

5. yum 源配置。

```
[root@localhost ~]# cd /etc
[root@localhost etc]# cd yum.repos.d
[root@localhost yum.repos.d]# cat openEuler_aarch64.repo
#Copyright (c) [2019] Huawei Technologies Co., Ltd.
#generic-repos is licensed under the Mulan PSL v1.
#You can use this software according to the terms and conditions of the Mulan PSL v1.
#You may obtain a copy of Mulan PSL v1 at:
#    http://license.coscl.org.cn/MulanPSL
#THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR
#IMPLIED, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT, MERCHANTABILITY OR FIT FOR A PARTICULAR
#PURPOSE.
#See the Mulan PSL v1 for more details.
[base]
name=openEuler20.03LTS
baseurl=https://repo.openeuler.org/openEuler-20.03-LTS/OS/aarch64/
enabled=1
gpgcheck=0
```

图 5.6 查看 yum 源

```
[root@localhost yum.repos.d]# yum makecache
openEuler20.03LTS                               0.0  B/s |   0  B     00:00
Error: Failed to download metadata for repo 'base': Cannot download repomd.xml: Cannot download repo
data/repomd.xml: All mirrors were tried
```

图 5.7 更新 yum 源

6. 测试程序。编写 Hello World 程序后，编译并运行。

```
[root@localhost test]# cat test1.c
#include<stdio.h>
int main()
{
printf("Hello!\n");
return 0;
}
```

图 5.8 Hello world 测试程序

```
[root@localhost test]# gcc test1.c -o hello
[root@localhost test]# ./hello
Hello!
```

图 5.8 Hello world 运行

6. 测试另外两个程序，运行成功。

```
[root@localhost test]# ./m1
memorycopy time is   238150800 ns
[root@localhost test]# ./m21
memorycopy time is  3358644688 ns
```

图 5.9 测试其他程序

5.4 小结

5.4.1 实验完成情况

成功在 x86 系统上搭建出能够兼容 ARM v8 指令集的模拟环境，为鲲鹏处理器的学习提供环境。目前 Windows 系统是主流，因此本次实验在 x86+Windows 平台上运行与 ARM v8 指令集兼容的模拟环境的方法。

学习并了解了 QEMU 的原理和使用，它通过动态翻译来模拟 CPU，将客户操作系统的指令 翻译给真正的硬件执行，实现对另一种体系结构计算机的模拟。经过 QEMU 的翻译，客户操作系 统可以间接地同真实主机中的 CPU、网卡、硬盘等硬件设备进行交互。由于程序执行过程需要 QEMU 的翻译，程序执行的性能与速度会比在真实主机上差。

掌握了 openEuler 操作系统的使用，初步了解了以 Linux 为内核的操作系统的运作方式。

搭建了鲲鹏开发环境，从 QEMU 模拟器的安装到操作系统的安装，以及对系统功能完成了配置，可以正常使用。

成功对实验要求的程序完成了测试，环境运行正常。

基本完成了实验要求的全部内容。

5.4.2 实验收获

通过本次实验，我初步了解如何在 WINDOWS 系统下使用虚拟机，并且学习了 Linux 操作系统的使用，成功配置了 Linux 操作系统，并且搭建了实验所需的环境。

掌握了 QEMU 的原理和使用，明白了 QEMU 的工作机理，并且了解了 QEMU 虚拟机的优缺点。

掌握了 openEuler 操作系统的使用，初步了解了以 Linux 为内核的操作系统的运行机理，以及其相对于常用的 WINDOWS 系统的优势。

掌握了如何在 Linux 操作系统下进行开发。

6 中断处理

6.1 实验内容

（1）通过观察与验证，理解中断矢量表的概念；
（2）熟悉 I/O 访问，BIOS 功能调用方法；
（3）掌握实方式下中断处理程序的编制与调试方法；
（4） 进一步熟悉内存的一些基本操纵技术；

6.2 任务 5.1 实验过程

1. 准备上机实验环境:安装了 dosbox 虚拟机环境(版本 0.73)，源程序编辑程序采用 IDE Masm for Windows 集成实验环境 2015；汇编程序使用 MASM 6.0；连接程序使用 LINK；调试工具使用 TD。

2. 编译例 6.2 的汇编程序，得到可运行程序。

```
.386
STACK SEGMENT USE16 STACK
      DB 200 DUP(0)
STACK ENDS

CODE SEGMENT USE16
      ASSUME CS:CODE,DS:CODE,SS:STACK
      COUNT DB 18
      HOUR DB ?,?,':'
      MIN DB ?,?,':'
      SEC DB ?,?
      BUF_LEN = $-HOUR
      CURSOR DW ?
      OLD_INT DW ?,?
NEW08H PROC FAR
      PUSHF
      CALL DWORD PTR CS:OLD_INT
      DEC CS:COUNT
      JZ DISP
      IRET
DISP: MOV CS:COUNT,18
      STI
      PUSHA
      PUSH DS
      PUSH ES
      MOV AX,CS
      MOV DS,AX
      MOV ES,AX
      CALL GET_TIME
      MOV BH,0
      MOV AH,3
      INT 10H
      MOV CURSOR,DX
      MOV BP,OFFSET HOUR
      MOV BH,0
      MOV DH,0
      MOV DL,80-BUF_LEN
      MOV BL,07H
      MOV CX,BUF_LEN
      MOV AL,0
      MOV AH,13H
```

图 6.1 例 6.2 源程序 1

```
        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 MIN,AX
        MOV AL,0
        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 SEC,AX
        RET
GET_TIME ENDP
BEGIN:PUSH CS
        POP DS
        MOV AX,3508H
        INT 21H
        MOV OLD_INT,BX
        MOV OLD_INT+2,ES
        MOV DX,OFFSET NEW08H
        MOV AX,2508H
        INT 21H
NEXT: MOV AH,0
        INT 16H
        CMP AL,'q'
        JNE NEXT
        LDS DX,DWORD PTR OLD_INT
        MOV AX,2508H
        INT 21H
        MOV AH,4CH
        INT 21H
CODE ENDS
        END BEGIN
```

图 6.2 例 6.2 源程序 2

3. 使用 td 调试例 6.2 的程序。

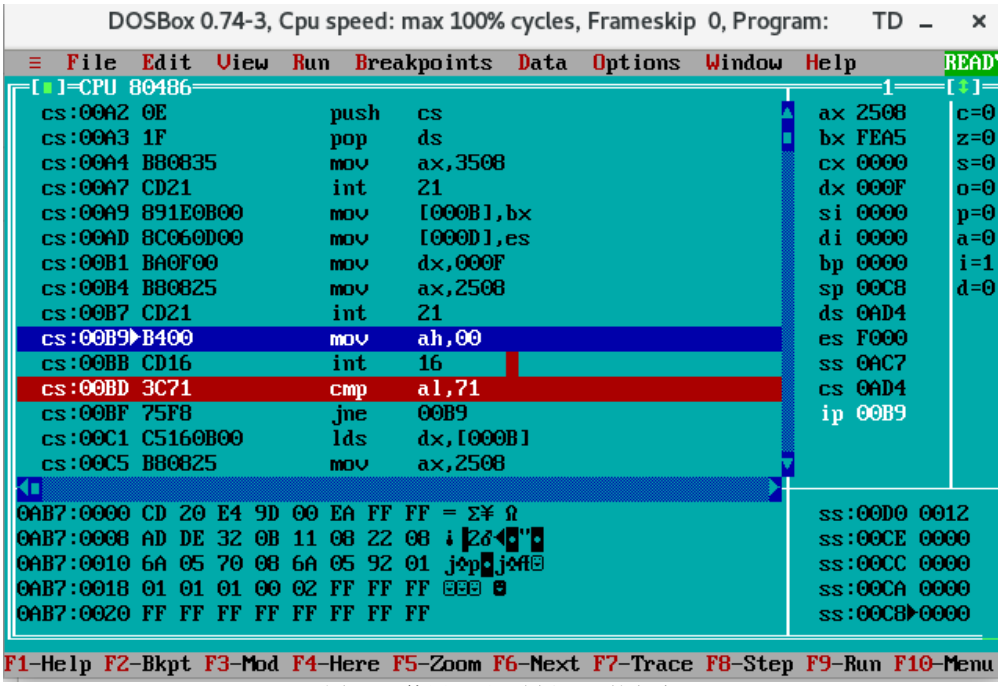

图 6.3 使用 td 调试例 6.2 的程序

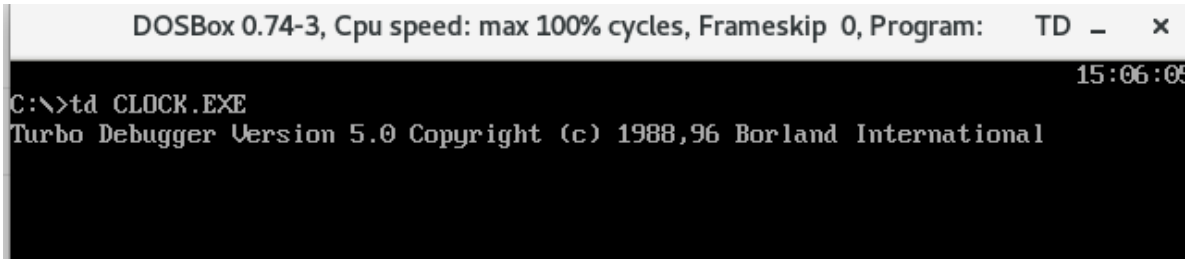

图 6.4 例 6.2 程序正常运行

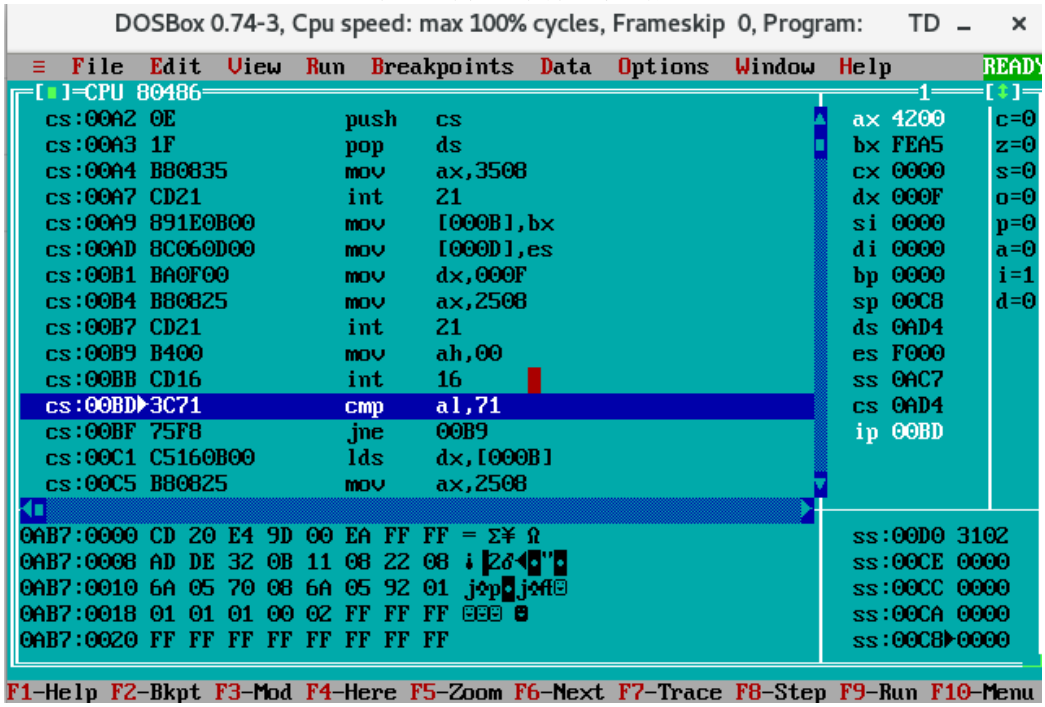

图 6.5 例 6.2 循环功能正常运行

4. 实现常驻功能，将该程序留在内存中保持运行。

```
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    mov ax,351ch
    int 21h                    ;获取原1ch中断入口地址并进行保存
    mov old_int,bx
    mov old_int+2,es
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    mov dx,offset new1ch       ;置新的1ch中断的入口地址
    mov ax,251ch
    int 21h
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
NEXTN: MOV AH,0
      INT 16H
      CMP AL,'q'
      JNE NEXTN
    mov dx,offset start
    add dx,15
    mov cl,4
    shr dx,cl                   ;将新的1ch中断子程序常驻内存
    add dx,10h
    mov al,0
    mov ah,31h
    int 21h
```

图 6.6 编写常驻内存程序

5. 测试常驻内存的功能。运行程序，右上角正常显示时间。输入’q’退出程序后，程序的功能保持运行，驻留成功。

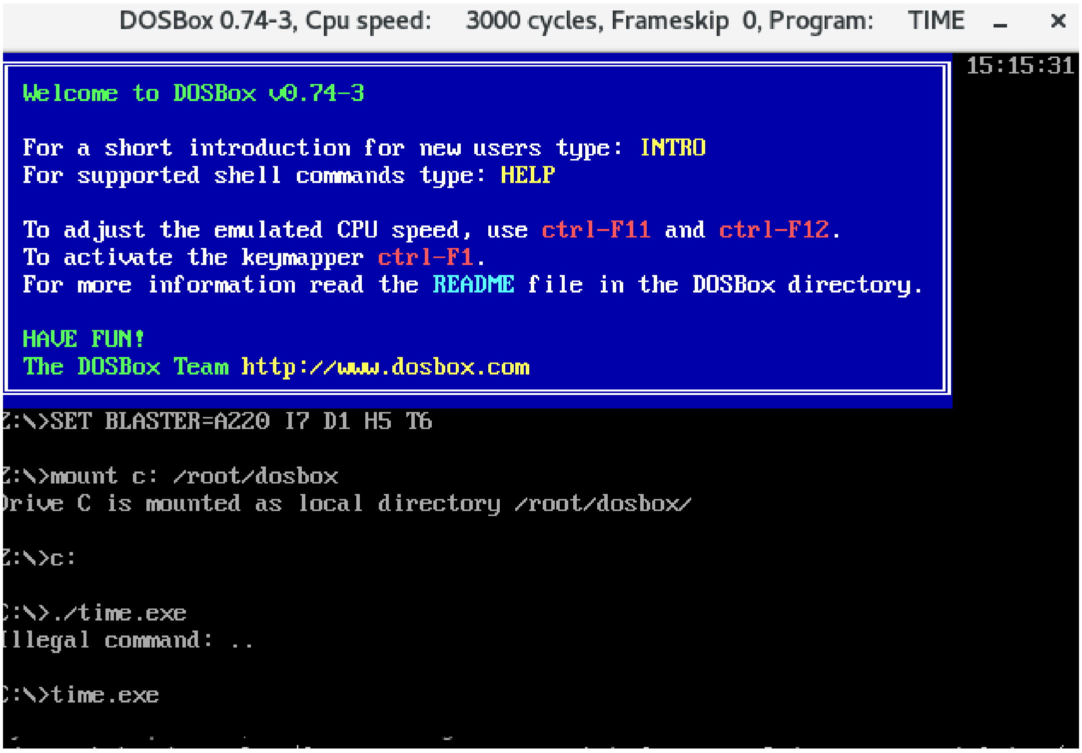

图 6.7 程序正常运行

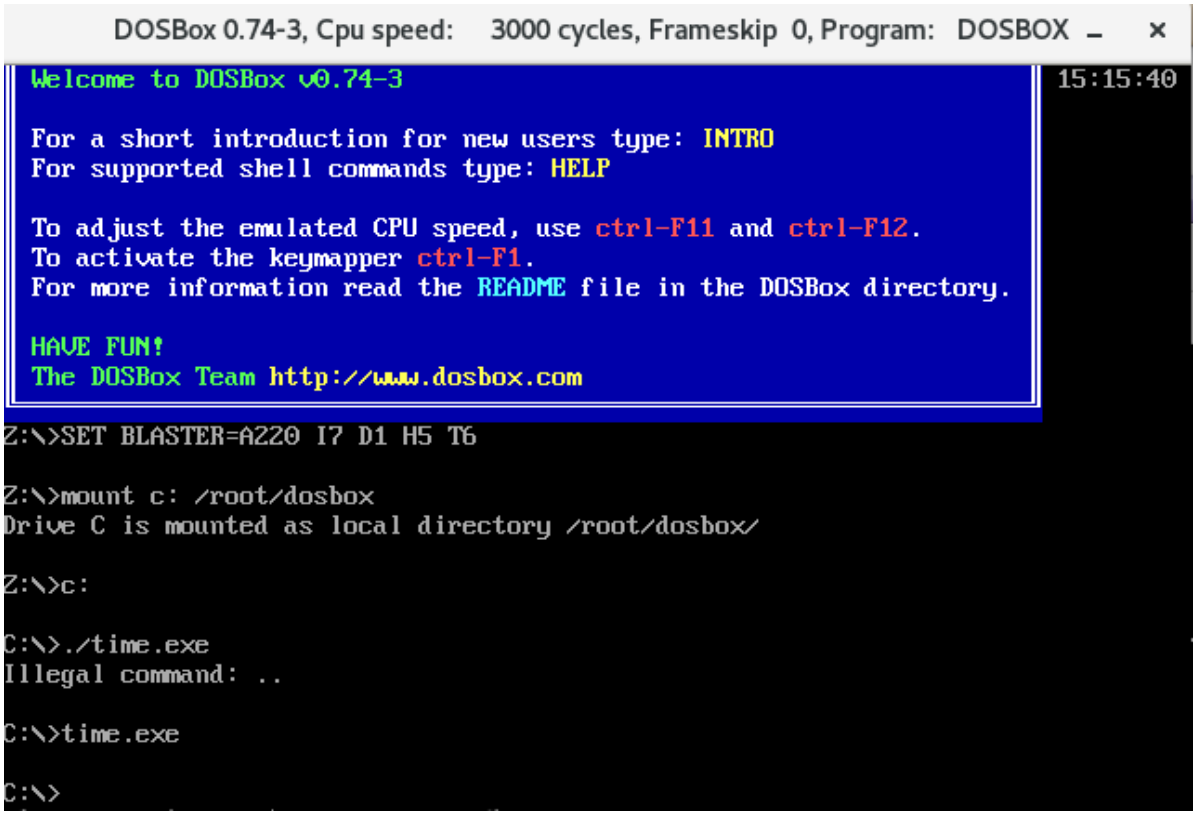

图 6.8 程序退出后

6.3 小结

6.3.1 实验完成情况

完成了对例 6.2 程序的调试以及分析。

完成了内存驻留功能的编写，成功将程序驻留在内存中，当程序退出后，仍有功能在内存中保持运行。

基本完成了实验要求。

6.3.2 实验收获

通过本次实验中对程序的调试与分析，初步理解了中断矢量表的概念。

掌握了如何编写内存驻留程序，并成功在例 6.2 源程序的基础上。进行改写，成功实现了该程序的内存驻留。

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（第 19 章，第 3 章、第 4 章、第 5 章）

[2]许向阳. 80X86 汇编语言程序设计上机指南. 武汉：华中科技大学出版社， 2007

[3]王元珍，曹忠升，韩宗芬. 80X86 汇编语言程序设计. 武汉：华中科技大学出版社，2005

[4]汇编语言课程组. 《汇编语言程序设计实践》任务书与指南，2021

[5]ARM 虚拟环境安装说明 V1.1.pdf 等参考资料，2022