華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 汇编语言编程基础

院　　系：计算机科学与技术

专业班级：　CS2202　

学　　号：　U202215378　

姓　　名：　冯瑞琦　

指导教师：　王多强　

2024 年 **4** 月 **5** 日

infbp 一、实验目的与要求

(1) 掌握汇编源程序编辑工具、汇编程序、连接程序、调试工具的使用；

(2) 熟悉分支、循环程序、子程序的结构，掌握分支、循环、子程序的调试方法；

(3) 加深对转移指令、子程序调用和返回指令及一些常用的汇编指令的理解。

二、实验内容

任务 2.1 习题三，第 2 题。

要求：（1）分别记录执行到“mov $10, %ecx”和“mov $1, %eax”之前的 EBX,EBP,ESI,EDI 各是多少。

（2）记录程序执行到退出之前数据段开始 40 个字节的内容，指出程序运行结果是否与设想的一致。

(3)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。

任务 2.2 习题三，第 3 题。

要求：(1) 内存单元中数据的访问采用变址寻址方式。

(2) 记录程序执行到退出之前数据段开始 40 个字节的内容，检查程序运行结果是否与设想的一致。

(3) 观察并记录机器指令代码在内存中的存放形式，并与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。(相似语句记录一条即可，重点理解机器码与汇编语句的对应关系，尤其注意操作数寻址方式的形式)。

任务 2.3 设计实现一个数据处理的程序

有一个计算机系统运行状态的监测系统会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 4 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。

每组数据的定义方法可以参考如下：

```
sdmid  .fill  9, 1, 0    # 每组数据的流水号（可以从 1 开始编号）
sda    .long  256809   # 状态信息 a
sdb    .long  -1023    # 状态信息 b
sdc    .long  1265     # 状态信息 c
sf     .long  0        # 处理结果 f
```

请编写完整的汇编语言程序，并按照要求设计子程序。

(1) 编写一子程序 calculate，完成 f 的计算并保存，并且子程序的入口参数为 esi（调用子程序前后 esi 的值不变），保存状态信息 a 的地址，若 f=100，则(eax)=0, f>100, 则(eax)=1, 若 f<100 则(eax)=-1，返回 eax；先按照不考虑溢出的要求编写程序，再按照考虑溢出的要求修改程序；

(2) 编写一子程序 copy_data，要求用堆栈传递参数，实现一组存储区状态信息的复制，要求每次拷贝 4 字节，多余的 1 个字节单独拷贝，参数为待复制的存储区的首地址和拷贝的目标地址，拷贝的字节长度。

(3) 查看运行到返回操作系统的指令之前，三个存储区域 LOWF、MIDF、HIGHF 内前 50 字节的值，只需要截图即可。

(4) 熟悉加减乘除等运算指令，内存拷贝方法。思考：如果三个状态信息是无符号数，程序需要做什么调整？

三、实验记录及问题回答

（1）任务 2.1 的运行结果等记录

①运行结果

执行到“mov $10, %ecx” 之前的 EBX,EBP,ESI,EDI 如下图所示。

```
Breakpoint 1, main () at 21.s:13
13      mov $10, %ecx
(gdb) i r ebx ebp esi edi
ebx            0x40403c            4210748
ebp            0x1                 1
esi            0x404028            4210728
edi            0x404032            4210738
(gdb)
```

inad a 执行到 “mov $1, %eax”之前的 EBX,EBP,ESI,EDI 如下图所示。

```
Breakpoint 2, lopa () at 21.s:26
26      mov $1, %eax
(gdb) i r ebx ebp esi edi
ebx            0x404046            4210758
ebp            0x1                 1
esi            0x404032            4210738
edi            0x40403c            4210748
(gdb)
```

②记录的运行结果及与预期对比

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，运行结果与设想一致。

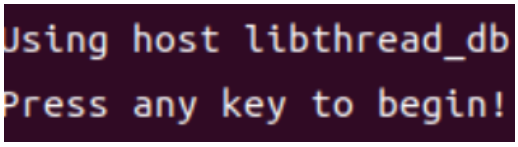
```
Breakpoint 2, lopa () at 21.s:26
26      mov $1, %eax
(gdb) info address buf1
Symbol "buf1" is at 0x404028 in a file compiled without debugging.
(gdb) x/40xb 0x404028
0x404028:       0x00    0x01    0x02    0x03    0x04    0x05    0x06
0x404030:       0x08    0x09    0x00    0x01    0x02    0x03    0x04
0x404038:       0x06    0x07    0x08    0x09    0x01    0x02    0x03
0x404040:       0x05    0x06    0x07    0x08    0x09    0x0a    0x04
0x404048:       0x06    0x07    0x08    0x09    0x0a    0x0b    0x0c
(gdb)
```

③修改的代码以及运行结果

```
prompt_msg: .asciz "Press any key to begin!\n"  # 提示信息字符串
msg_len = . - prompt_msg

.section .text
.global main
main:
    # 显示提示信息
    mov $4, %eax          # 使用系统调用号 4 来进行输出
    mov $1, %ebx          # 使用标准输出文件描述符 1
    mov $prompt_msg, %ecx # 传递提示信息的地址
    mov $msg_len, %edx    # 传递提示信息的长度
    int $0x80             # 调用系统调用
    # 等待用户按键
    mov $3, %eax        # 使用系统调用号 3 来进行输入
    mov $0, %ebx        # 使用标准输入文件描述符 0
    mov $buf1, %ecx     # 传递一个缓冲区来存储用户输入（但我们不会使用它）
    mov $1, %edx        # 传递缓冲区的大小
    int $0x80           # 调用系统调用
```

修改后运行页面如下图。



（2）任务 2.2 的算法思想、运行结果等记录

①修改的代码

算法思想：使用%esi 寄存器作为变址寄存器，依次访问 buf1、buf2、buf3 和 buf4 中的每个字节。每次循环迭代，都从 buf1 中读取一个字节到%al 寄存器中。然后将%al 中的值写入 buf2、buf3 和 buf4 中的相应位置，并对其进行加减操作。通过修改%esi 寄存器的值，实现对 buf1、buf2、buf3 和 buf4 的顺序访问。

有修改的代码片段如下。

```
lopa:
    mov  buf1(%esi), %al
    mov  %al, buf2(%esi)
    inc  %al
    mov  %al, buf3(%esi)
    add  $3,  %al
    mov  %al, buf4(%esi)
```

②运行结果与预期对比

程序执行到退出之前数据段开始 40 个字节的内容如下图，运行结果与设想的一致。

```
(gdb) info address buf1
Symbol "buf1" is at 0x404028 in a file compiled without debugging.
(gdb) x/40xb 0x404028
0x404028:       0x00    0x01    0x02    0x03    0x04    0x05    0x06    0x07
0x404030:       0x08    0x09    0x00    0x00    0x00    0x00    0x00    0x00
0x404038:       0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x404040:       0x00    0x00    0x00    0x00    0x00    0x00    0x04    0x00
0x404048:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

③观察记录

机器码如下图所示。

由此可见，机器码与汇编语言之间存在密切的关系，它们可以看作是同一条指令的不同表现形式。机器码是由二进制数字表示的指令代码，直接由计算机硬件执行。每条机器指令都对应着一系列的二进制数字，这些数字指示了特定的操作、操作数以及操作方式。但是，机器码并不直观，难以理解和记忆，因此为了方便人类程序员编写和理解程序，汇编语言应运而生。

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000401106 <+0>:     mov    $0x404028,%esi
   0x000000000040110b <+5>:     mov    $0x404032,%edi
   0x0000000000401110 <+10>:    mov    $0x40403c,%ebx
   0x0000000000401115 <+15>:    mov    $0x404046,%edx
   0x000000000040111a <+20>:    mov    $0xa,%ecx
```

使用 objdump 指令进行反汇编，结果如下图所示。

在反汇编过程中，计算机程序将二进制机器码转换回汇编指令，以便程序员能够查看和理解已编译程序的工作原理。

因此，汇编语言、机器码和反汇编之间构成了一种密切的关系。汇编语言提供了一种人类可读的表示形式，使得程序员能够更轻松地编写和理解程序；机器码是计算机实际执行的指令代码；而反汇编则是将机器码转换回汇编语言的过程，为程序分析和逆向工程提供了便利。

```
Disassembly of section .note.gnu.property:

0000000000400338 <.note.gnu.property>:
  400338:       04 00                   add    al,0x0
  40033a:       00 00                   add    BYTE PTR [rax],al
  40033c:       10 00                   adc    BYTE PTR [rax],al
  40033e:       00 00                   add    BYTE PTR [rax],al
  400340:       05 00 00 00 47          add    eax,0x47000000
  400345:       4e 55                   rex.WRX push rbp
  400347:       00 02                   add    BYTE PTR [rdx],al
  400349:       80 00 c0                add    BYTE PTR [rax],0xc0
  40034c:       04 00                   add    al,0x0
  40034e:       00 00                   add    BYTE PTR [rax],al
  400350:       01 00                   add    DWORD PTR [rax],eax
  400352:       00 00                   add    BYTE PTR [rax],al
  400354:       00 00                   add    BYTE PTR [rax],al
        ...
```

（3）任务 2.3 的算法思想、流程图、遇到的问题及其解决方法、运行结果等记录

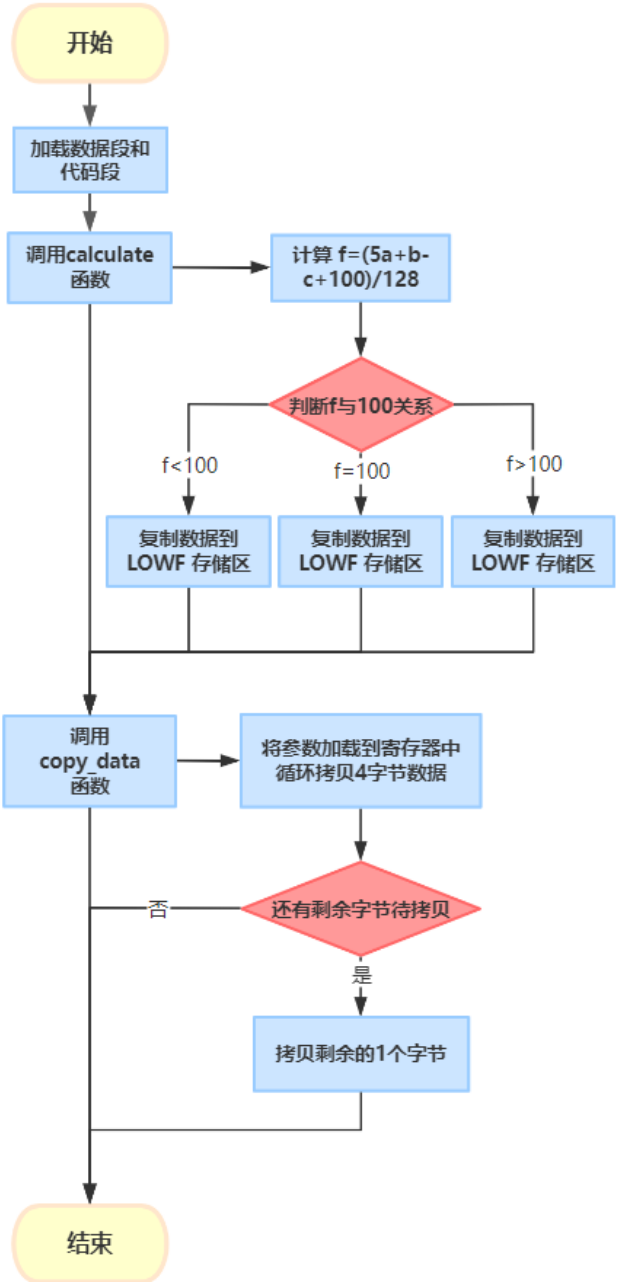
①算法实现

计算处理结果 f 时，经过 calculate 子程序，从输入参数中提取状态信息 a、b 和 c。根据公式 f=(5a+b-c+100)/128 计算得到处理结果 f。对 a 加上 100，接着加上 b，最后减去 c，再乘 5，得到中间结果(5a+b-c+100)。随后将中间结果除以 128 并保留整数部分，即(5a+b-c+100) 除以 128 得到 f。根据 f 的值设置 eax 寄存器的值：如果 f=100，则 eax=0；如果 f>100，则 eax=1；如果 f<100，则 eax=-1。

复制数据根据处理结果时，在 copy_data 子程序中提取源地址、目标地址和要复制的字节长度。通过循环以批量拷贝的方式复制每组数据的前 len/4 个字节，每次复制 4 个字节。然后判断是否还有剩余字节需要复制；若有，单独复制剩余的 1 个字节，并根据 f 的值将数据复制到不同的存储区域。当 f=100 时，将数据复制到 MIDF 存储区；当 f100 时，将数据复制到 HIGHF 存储区。最后，返回到调用者处，完成数据处理和复制的任务。

②流程图

任务 2.3 的算法流程图如下图所示。



③遇到的问题及其解决方法

问题 1：需要根据计算结果确定数据应该复制到哪个存储区域。

解决方法：在处理这个问题时，我使用了条件判断指令来比较计算结果 f 与预定值的大小关系。通过使用 cmp 指令来比较 f 的值和 100，我能够确定其相对大小。然后，根据比较结果，使用不同的跳转指令，如 jge（跳转到大于等于的情况）和 jl（跳转到小于的情况），以选择相应的跳转目标。这样，根据计算结果的不同，能够将数据复制到正确的存储区域中去。

问题 2：在子程序中需要保存和恢复寄存器状态。

解决方法：在编写子程序时，我使用了 push 和 pop 指令来保存和恢复寄存器的状态，将寄存器的当前值推入栈中，以便稍后恢复它们的状态。通过在子程序的开始和结束处使用这些指令，确保在子程序执行期间寄存器的值不会被其他操作改变，从而保证了程序的正确性。

问题 3：复制数据时需要处理多余的字节和边界情况。

解决方法：在处理数据复制时，采用了循环结构和条件判断来确保每次复制的是 4 字节数据。这样，即使在处理数据长度不是 4 的倍数或者数据存储区域的边界情况下，也能够正确地处理数据。对于多余的字节使用了额外的条件判断和指令，以确保所有数据都能够被正确地复制，从而保证了程序的完整性和准确性。

④运行结果

LOWF 存储区的数据如下图所示。

```
(gdb) x/50xb &LOWF
0x404068:       0x0c    0x00    0x00    0x00    0xff    0xff    0xff    0xff
0x404070:       0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x404078:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x404080:       0x00    0x32    0x00    0x00    0xff    0xff    0xff    0xff
0x404088:       0xcf    0x0e    0x00    0x00    0x00    0x00    0x00    0x00
0x404090:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x404098:       0x00    0x00
```

⑤思考

这次汇编代码任务要求完成两个主要功能：首先是计算一个值 f，然后是输出三个存储区域 LOWF、MIDF 和 HIGHF 内前 50 字节的值。

对我来说，这个任务是一个挑战，因为需要将抽象的任务转化为底层的机器指令来实现。在着手编写代码之前，首先要理解每一行代码的作用和逻辑。我需要考虑如何使用汇编指令进行数学运算、条件分支和系统调用。这让我认识到了汇编语言与高级语言的差异，它更贴近计算机硬件层面，需要更深入地理解计算机的内部工作原理。

在编写代码的过程中，我遇到了许多问题。例如，需要确保正确使用寄存器来保存变量和地址，并注意指令的顺序和语法。在输出存储区域内容时，我学会了如何使用系统调用与操作系统进行交互，这是一个之前未曾接触过的新领域。

代码仍然存在许多不足之处，如可读性差、可移植性低等，还有很长的路要走，但这是一个学习过程，我会继续努力改进。

四、体会

这次实验对我而言是一项具有挑战性的任务，但正是在挑战中我获得了成长和收获。

完成这次实验后，我更加深入地了解了汇编语言及其在计算机系统中的应用。汇编语言是一种低级语言，直接操作计算机硬件，因此对于理解计算机系统的工作原理至关重要。通过编写汇编程序，我不仅仅是在学习一门编程语言，更是在探索计算机的内部结构和运行机制。

在实验中，我遇到了各种各样的困难和问题。因为以前没有接触过汇编语言，理解汇编语言的语法和语义对我来说有些困难，需要反复查看 PPT 和在网上查阅相关资料，并进行反复的练习。其次是调试程序时遇到的各种错误，有些错误可能是由于我对汇编语言的理解不够深入，有些则可能是由于代码逻辑错误造成的。无论是哪一种情况，我都不得不耐心地一步步排查错误，并尝试不同的解决方案。在这个过程中，我学会了如何利用调试工具来定位问题，并通过修改代码来解决它们。

除了技术方面的挑战，我还面临着时间管理和自我调节的挑战。在规定的 4 小时内完成这项实验我很难做到，而且直言途中也出现意想不到的问题，甚至连学习怎样使用虚拟机运行程序都花费了很多时间，这需要我保持耐心和毅力。在实验过程中，我不断调整自己的学习和工作计划，尽量避免拖延并保持专注，这对我提高了时间管理能力和自我调节能力。

总之，在本次实验中，我获得了丰富的经验和知识。在解决问题的过程中，我不仅加深了对汇编语言的理解，还提升了自己的问题解决能力。

五、源码

实验任务 2.1

```
.section .data
buf1: .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
buf2: .fill 10, 1, 0
buf3: .fill 10, 1, 0
buf4: .fill 10, 1, 0
prompt_msg: .asciz "Press any key to begin!\n"  # 提示信息字符串
msg_len = . - prompt_msg

.section .text
.global main
main:
    # 显示提示信息
    mov $4, %eax            # 使用系统调用号 4 来进行输出
    mov $1, %ebx            # 使用标准输出文件描述符 1
    mov $prompt_msg, %ecx # 传递提示信息的地址
    mov $msg_len, %edx     # 传递提示信息的长度
    int $0x80               # 调用系统调用

    # 等待用户按键
```

```
    mov $3, %eax        # 使用系统调用号 3 来进行输入
    mov $0, %ebx        # 使用标准输入文件描述符 0
    mov $buf1, %ecx     # 传递一个缓冲区来存储用户输入（但我们不会使用它）
    mov $1, %edx        # 传递缓冲区的大小
    int $0x80           # 调用系统调用

lopa:
    mov (%esi), %al
    mov %al, (%edi)
    inc %al
    mov %al, (%ebx)
    add $3, %al
    mov %al, (%edx)
    inc %esi
    inc %edi
    inc %ebx
    inc %edx
    dec %ecx
    jnz lopa
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

实验任务 2.2

```
section .data
buf1: .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
buf2: .fill 10, 1, 0
buf3: .fill 10, 1, 0
buf4: .fill 10, 1, 0
.section .text
.global  main
main:
    mov  $buf1, %esi
    mov  $buf2, %edi
    mov  $buf3, %ebx
    mov  $buf4, %edx
    mov  $10, %ecx
lopa:
    mov  buf1(%esi), %al
    mov  %al, buf2(%esi)
    inc  %al
    mov  %al, buf3(%esi)
    add  $3,  %al
    mov  %al, buf4(%esi)    inc  %esi
    dec  %ecx
```

```
    jnz  lopa

    mov  $1, %eax
    movl $0, %ebx
    int  $0x80
```

实验任务 2.3

```
.section .data
sdmid:  .ascii "000111", "\0\0\0"     # 每组数据的流水号（可以从 1 开始编号）
sda:    .long  512   # 状态信息 a
sdb:    .long  -1023    # 状态信息 b
sdc:    .long  1265     # 状态信息 c
sf:     .long  0        # 处理结果 f
        .ascii  "000222","\0\0\0"
        .long  256809      # 状态信息 a
        .long  -1023    # 状态信息 b
        .long  2780     # 状态信息 c
        .long  0        # 处理结果 f
       .ascii  "000333","\0\0\0"
        .long  2513# 状态信息 a
        .long  1265    # 状态信息 b
        .long  1023     # 状态信息 c
        .long  0        # 处理结果 f
       .ascii  "000444","\0\0\0"
        .long  512   # 状态信息 a
        .long  -1023    # 状态信息 b
        .long  1265     # 状态信息 c
        .long  0        # 处理结果 f
        .ascii "555555","\0\0\0"
        .long  2513
        .long  1265
        .long  1023
        .long  0
        .ascii "666666","\0\0\0"
        .long 256800
        .long -2000
        .long 1000
        .long 0
        num = 6
midf:  .fill  9, 1, 0
          .long 0, 0, 0, 0
          .fill  9, 1, 0
          .long 0,0,0,0
       .fill 9, 1,0
       .long 0,0,0,0
highf: .fill  9, 1, 0
          .long 0, 0, 0, 0
          .fill  9, 1, 0
          .long 0,0,0,0
          .fill 9,1,0
          .long 0,0,0,0
```

```
lowf:   .fill  9, 1, 0
          .long 0, 0, 0, 0
          .fill  9, 1, 0
          .long 0,0,0,0
          .fill 9,1,0
          .long 0,0,0,0
        len=25

.section .text
.global calculate
calculate:
    push %ebx
    push %ecx
    push %edx
    movl 8(%esp), %ebx
    movl 12(%esp), %ecx
    movl 16(%esp), %edx
    leal 100(%ebx), %eax
    addl %ecx, %eax
    subl %edx, %eax         # eax = a + b - c + 100
    shll $2, %eax           # eax = (a + b - c + 100) * 4
    addl %eax, %ebx         # ebx = 5a + b - c + 100
    sarl $7, %ebx           # ebx = (5a + b - c + 100) / 128
    cmpl $100, %ebx         # 比较结果和 100
    jge greater_than_100    # 如果大于等于 100，跳转到 greater_than_100
    jl less_than_100        # 如果小于 100，跳转到 less_than_100
equal_to_100:
    movl $0, %eax           # f=100，eax=0
    jmp end                 # 跳转到 end
greater_than_100:
    movl $1, %eax           # f>100，eax=1
    jmp end                 # 跳转到 end
less_than_100:
    movl $-1, %eax          # f<100，eax=-1
end:
    # 恢复寄存器状态
    pop %edx
    pop %ecx
    pop %ebx
    ret
.global copy_data
copy_data:
    # 保存寄存器状态
    push %ebx
    push %ecx
    push %edx
```

```
    movl 8(%esp), %ebx   # 加载源地址
    movl 12(%esp), %ecx  # 加载目标地址
    movl 16(%esp), %edx  # 加载拷贝长度
    # 循环拷贝 4 字节数据
    movl %edx, %eax
    shrl $2, %eax
copy_loop:
    cmpl $0, %eax
    je end_copy_loop
    movl (%ebx), %esi
    movl %esi, (%ecx)
    addl $4, %ebx
    addl $4, %ecx
    subl $1, %eax
    jmp copy_loop
end_copy_loop:
    # 拷贝剩余的 1 个字节
    movl %edx, %eax
    andl $3, %eax
    cmpl $0, %eax
    je end
    movb (%ebx), %al
    movb %al, (%ecx)
```