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 ： 计算机 202201

学　　号 ： U202215357

姓　　名 ： 王文涛

指导教师 ： 朱　虹

2024 年 4 月 3 日

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”和“mov $1, %eax”之前的 EBX,EBP,ESI,EDI 值如下图所示。

```
(gdb) i r ebx
ebx            0x40403c            4210748
(gdb) i r ebp
ebp            0x1                 1
(gdb) i r esi
esi            0x404028            4210728
(gdb) i r edi
edi            0x404032            4210738
(gdb) c
Continuing.

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

② 记录的运行结果并与预期对比

执行到退出之前数据段开始 40 个字节的内容如下。

```
(gdb) x/40xb 0x404028
0x404028:       0x00    0x01    0x02    0x03    0x04    0x05    0x06    0x07
0x404030:       0x08    0x09    0x00    0x01    0x02    0x03    0x04    0x05
0x404038:       0x06    0x07    0x08    0x09    0x01    0x02    0x03    0x04
0x404040:       0x05    0x06    0x07    0x08    0x09    0x0a    0x04    0x05
0x404048:       0x06    0x07    0x08    0x09    0x0a    0x0b    0x0c    0x0d
```

与设想一致。

③ 修改的代码（不要所有代码贴上来）以及运行结果

lopa 前增加的代码：

```
mov $4, %eax
mov $1, %ebx
lea message, %ecx
mov $26, %edx
int $0x80
mov $3, %eax
mov $0, %edi
lea bufc, %esi
mov $1, %edx
int $0x80
```

运行结果：

```
(gdb) run
Starting program: /home/f/2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Press any key to begin!
1
[Inferior 1 (process 5496) exited normally]
```

功能正常。

（2）任务 2.2 的算法思想、运行结果等记录

①修改的代码

修改的代码部分：

```
mov $0, %ecx
lopa: mov buf1(%ecx), %al
mov %al, buf2(%ecx)
inc %al
mov %al, buf3(%ecx)
add $3, %al
mov %al, buf4(%ecx)
inc %ecx
cmp $10, %ecx
jnz lopa
```

②运行结果与预期对比

```
(gdb) x /40xb &buf1
0x404028:       0x00    0x01    0x02    0x03    0x04    0x05    0x06    0x07
0x404030:       0x08    0x09    0x00    0x01    0x02    0x03    0x04    0x05
0x404038:       0x06    0x07    0x08    0x09    0x01    0x02    0x03    0x04
0x404040:       0x05    0x06    0x07    0x08    0x09    0x0a    0x04    0x05
0x404048:       0x06    0x07    0x08    0x09    0x0a    0x0b    0x0c    0x0d
```

实验结果与预期一致。

④ 观察记录

```
    mov $0, %ecx
0x0000000000401106 <main+0>: b9 00 00 00 00  mov    $0x0,%ecx

    lopa: mov buf1(%ecx), %al
0x000000000040110b <lopa+0>: 67 8a 81 28 40 40 00    mov    0x404028(%ecx),%al

    mov %al, buf2(%ecx)
0x0000000000401112 <lopa+7>: 67 88 81 32 40 40 00    mov    %al,0x404032(%ecx)

    inc %al
0x0000000000401119 <lopa+14>:        fe c0   inc    %al

    mov %al, buf3(%ecx)
0x000000000040111b <lopa+16>:        67 88 81 3c 40 40 00    mov    %al,0x40403c(%ecx)

    add $3, %al
0x0000000000401122 <lopa+23>:        04 03   add    $0x3,%al

    mov %al, buf4(%ecx)
0x0000000000401124 <lopa+25>:        67 88 81 46 40 40 00    mov    %al,0x404046(%ecx)

    inc %ecx
0x000000000040112b <lopa+32>:        ff c1   inc    %ecx

    cmp $10, %ecx
0x000000000040112d <lopa+34>:        83 f9 0a        cmp    $0xa,%ecx

    jnz lopa
0x0000000000401130 <lopa+37>:        75 d9   jne    0x40110b <lopa>

    mov  $1, %eax
0x0000000000401132 <lopa+39>:        b8 01 00 00 00  mov    $0x1,%eax

    movl $0, %ebx
0x0000000000401137 <lopa+44>:        bb 00 00 00 00  mov    $0x0,%ebx

    int  $0x80
0x000000000040113c <lopa+49>:        cd 80   int    $0x80
```

机器指令代码内存中线性存放，每条指令都有对应的地址。mov $0, %ecx 机器指令 b9 00 00 00 00，b9 是操作码，表示将立即数加载到寄存器 ecx 中，后面四个字节是立即数 0。指令 67 8a 81 28 40 40 00,67 是前缀，表示将操作数从 32 位更改为 16 位，8a 是操作码，表示从一个内存地址加载一个字节到 %al 寄存器。81 是间接寻址模式的前缀，用于指定一个绝对地址。后面的 28 40 40 00 是一个 32 位绝对地址，即 0x404028。下一句中 88 代表目标操作数是内存地址。inc %al，fe c0 中，操作码 fe 表示执行递增操作，最后一位是 0，说明是字节操作，c0 指定操作数是%al。操作码 04 表示将立即数加到寄存器 al 上，03 即立即数。cmp $0xa,%ecx,83 f9 0a 中，83 是前缀。后面接立即数，f9 表示 ecx，表示将立即数与%ecx 比较，0a 即立即数。int $0x80, cd 80 表示系统调用。

（3）任务 2.3 的算法思想、流程图、遇到的问题及其解决方法（选择 3-4 个）、运行结果等记录

① 算法实现

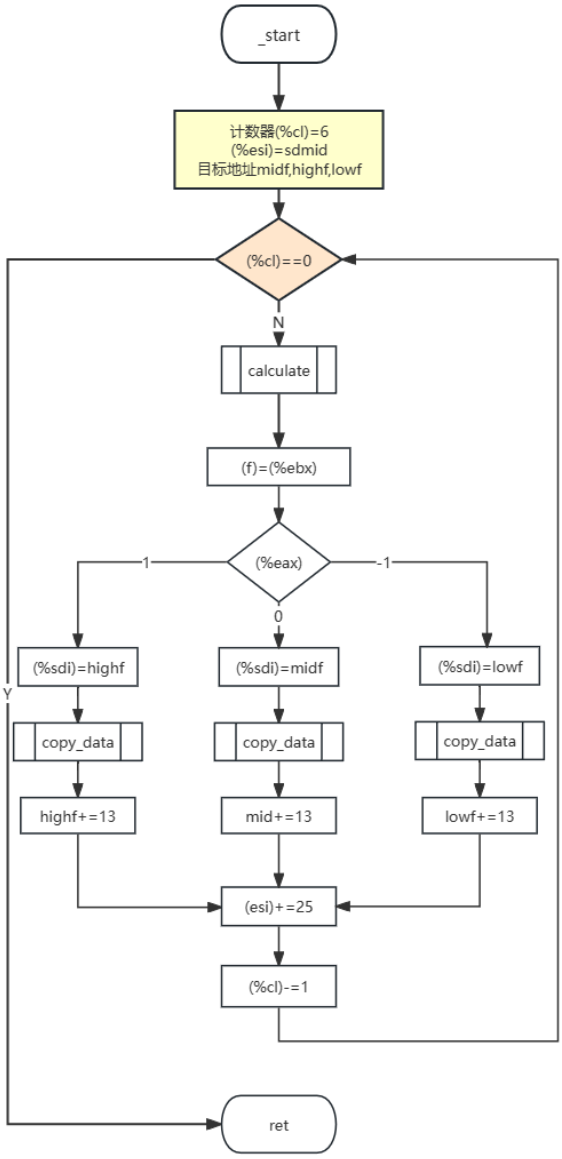
1) 主体部分_start,将 midf,highf,lowf 的地址当作局部变量存放到内存中。使用寄存器 cl 递减计数，重复六次。将 sdmid 存到 esi 中作为初始值。在循环中，先调用 calculate 函数，将返回的 f 的值存到内存中对应的位置，判断%eax 中储存的值，将对应的地址存到%edi 中，然后调用 copy_date 函数，然后将对应内存中的地址增加 13，作为下次储存的首地址。接着将源地址增加 25，作为下次操作的源地址。

2) calculate 函数部分，将 esi 压栈，根据 esi 的地址，将 a,b,c 进行计算，结果储存到 edx 中，比较结果与 100，将-1，0 或者 1 储存到 eax 中返回，esi 出栈。

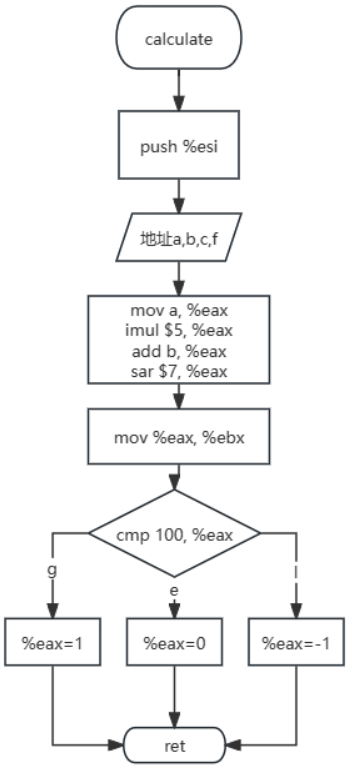
3) copy_data 函数部分，分别将 ecx,esi,edi 压栈，循环两次，将 esi 所指地址处内容复制到 edi 所指地址处，每次移动 4 字节。然后复制剩下的一个字节。接着将 esi 移动到 f 所在的位置，将 f 复制到 edi 处。寄存器出栈。

② 流程图

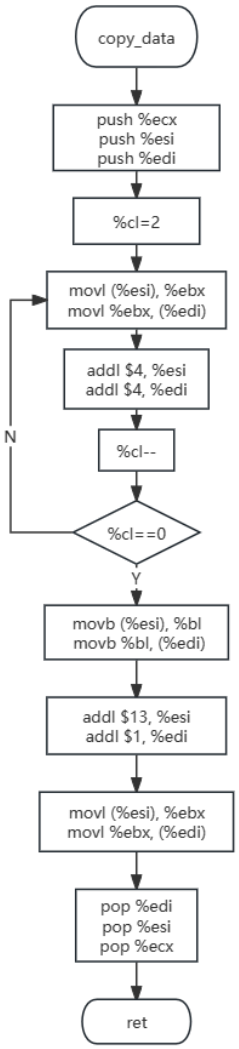
主函数：



calculate 函数:


calculate
push %esi
地址a,b,c,f
mov a, %eax
imul $5, %eax
add b, %eax
sar $7, %eax
mov %eax, %ebx
cmp 100, %eax
g
e
l
%eax=1
%eax=0
%eax=-1
ret

copy_data 函数:



③ 遇到的问题及其解决方法

1) 调试代码的时候要在终端中多次重复输入三条指令，比较浪费时间。最后选择在 linux 系统中使用 vscode，自定义 task，绑定快捷键实现一键编译。

2) 要储存三个地址，如果使用寄存器，寄存器就不够用了。最后学习了汇编中局部变量的储存形式，将三个地址储存在了内存中。

3) 在写代码时还出现了许多问题，各种报错，最后经过查询，使用 gdb 一步步调试，观察寄存器和内存中值的变化，最终都得以解决。

④ 运行结果(展示 LOWF,HIGHF,MIDF 三个存储区的数据)

```
(gdb) x /50xb &midf
0x804a096:      0x30    0x30    0x30    0x33    0x33    0x33    0x00    0x00
0x804a09e:      0x00    0x64    0x00    0x00    0x00    0x35    0x35    0x35
0x804a0a6:      0x35    0x35    0x35    0x00    0x00    0x00    0x64    0x00
0x804a0ae:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a0b6:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a0be:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a0c6:      0x00    0x00
(gdb) x /50xb &highf
0x804a0e1:      0x30    0x30    0x30    0x32    0x32    0x32    0x00    0x00
0x804a0e9:      0x00    0x12    0x27    0x00    0x00    0x36    0x36    0x36
0x804a0f1:      0x36    0x36    0x36    0x00    0x00    0x00    0x18    0x27
0x804a0f9:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a101:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a109:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a111:      0x00    0x00
(gdb) x /50xb &lowf
0x804a12c:      0x30    0x30    0x30    0x31    0x31    0x31    0x00    0x00
0x804a134:      0x00    0x02    0x00    0x00    0x00    0x30    0x30    0x30
0x804a13c:      0x34    0x34    0x34    0x00    0x00    0x00    0x02    0x00
0x804a144:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a14c:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a154:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a15c:      0x00    0x00
```

⑤ 思考

如果状态信息是无符号数，那么首先比较的时候，jg，jl 应该改为 ja，jb。进行算数运算时，乘法 imul 应改为 mul，算术右移可以改为逻辑右移。

四、体会

由于在做实验时比较缺乏汇编的相关知识，做起来比较困难。同时因为是第一次使用虚拟机和 linux 系统，安装和使用系统就耗费了很多的时间。但是经过这次实验，我对 linux 系统的使用变得比较熟悉了，也感受到了 linux 的妙处。因为对汇编不熟，我尝试自己写一些简单的 c 程序，然后经过反汇编，查看汇编代码来学习汇编语言，这种方法还是比较有效的。经过这次实验，我还掌握了 gcc 和 gdb 的各种用法，能够熟练地使用 gdb 的各种命令来进行调试了。之后写 c 程序时，我也可以通过查看汇编代码来理解程序内在的执行逻辑了。

五、源码

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
message: .ascii "Press any key to begin!\n"
bufc: .space 1
.section .text
.global  main
main:
mov $4, %eax
mov $1, %ebx
lea message, %ecx
```

```
mov $26, %edx
int $0x80

mov $3, %eax
mov $0, %edi
lea bufc, %esi
mov $1, %edx
int $0x80

mov  $buf1, %esi
mov  $buf2, %edi
mov  $buf3, %ebx
mov  $buf4, %edx
mov  $10, %ecx

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
movl $0, %ebx
int  $0x80
```

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 $0, %ecx
lopa: mov buf1(%ecx), %al
mov %al, buf2(%ecx)
inc %al
mov %al, buf3(%ecx)
add $3, %al
mov %al, buf4(%ecx)
inc %ecx
cmp $10, %ecx
jnz lopa
mov  $1, %eax
movl $0, %ebx
int  $0x80
```

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     _start
_start:
```

```
    mov %esp, %ebp
    sub $0x20, %esp
    movl $midf, -0x4(%ebp)
    movl $highf, -0x8(%ebp)
    movl $lowf, -0xc(%ebp)
    mov $num, %cl
    leal sdmid, %esi
    loop1:
    call calculate
    movl %edx, 21(%esi)
    cmp $0, %eax
    jle lee
    movl -0x8(%ebp), %edi
    call copy_data
    addl $13, -0x8(%ebp)
    jmp con
    lee:
    cmp $0, %eax
    je ee
    mov -0xc(%ebp), %edi
    call copy_data
    addl $13, -0xc(%ebp)
    jmp con
    ee:
    mov -0x4(%ebp), %edi
    call copy_data
    addl $13, -0x4(%ebp)
    con:
    add $25, %esi
    dec %cl
    jnz loop1
    mov $1, %eax
    mov $0, %ebx
    int $0x80
.type calculate @function
calculate:
```

```
    push %esi
    mov 9(%esi), %eax
    imul $5, %eax
    add 13(%esi), %eax
    sub 17(%esi), %eax
    add $100, %eax
    sar $7, %eax
    mov %eax, %edx
    cmp $100, %eax
    jle le
    mov $1, %eax
    jmp end
    le:
    cmp $100, %eax
    je eq
    mov $-1, %eax
    jmp end
    eq:
    mov $0, %eax
    jmp end
    end:
    pop %esi
    ret


.type copy_data @function
copy_data:
    push %ecx
    push %esi
    push %edi
    mov $2, %cl
    loop2:
    movl (%esi), %ebx
    movl %ebx, (%edi)
    addl $4, %esi
    addl $4, %edi
```

```
    dec %cl
    jnz loop2
    movb (%esi), %bl
    movb %bl, (%edi)
    addl $13, %esi
    addl $1, %edi
    movl (%esi), %ebx
    movl %ebx, (%edi)
    pop %edi
    pop %esi
    pop %ecx
    ret
ret
```