

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验

实验名称: 实验二 程序执行时间与代码长度优化

实验时间: 2018-4-9, 14: 00-17: 30 实验地点: 南一楼

指导教师: 朱虹

专业班级: 计算机科学与技术 1601 班

学 号: U201614531 姓 名: 刘本嵩

同组学生: 无 报告日期: 2018 年 4 月 9 日

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:

日期:

成绩评定

实验完成质量得分 (70 分) (实验步骤清晰详细深入, 实验记录真实完整等)	报告撰写质量得分(30 分) (报告规范、完整、通顺、详实等)	总成绩 (100 分)

指导教师签字:

日期:

目录

1 实验目的与要求	1
2 实验内容	2
2.1 任务 1：观察多重循环对 CPU 计算能力消耗的影响	2
2.2 任务 2：对任务 1 中的汇编源程序进行优化	2
3 实验过程	4
3.1 任务 1	4
3.1.1 实验步骤	4
3.1.2 修改后源程序	4
3.1.3 实验记录与分析	7
3.2 任务 2	8
3.2.1 实验步骤	8
3.2.2 优化后源程序	9
3.2.3 实验记录与分析	11
4 总结与体会	12
5 参考文献	13

1 实验目的与要求

1. 熟悉汇编语言指令的特点，掌握代码优化的基本方法；
2. 理解高级语言程序与汇编语言程序之间的对应关系。

2 实验内容

2.1 任务 1：观察多重循环对 CPU 计算能力消耗的影响

要求：

应用场景介绍：以实验一任务四的背景为基础，只要顾客买走了网店中的一件商品，老板就需要重新获得全部商品的平均利润率。现假设在双十一零点时，SHOP1 网店中的“Bag”商品共有 m 件，有 m 个顾客几乎同时下单购买了该商品。请模拟后台处理上述信息的过程并观察执行的时间。

上述场景的后台处理过程，可以理解为在同一台电脑上有 m 个请求一起排队使用实验一任务四的程序。为了观察从第 1 个顾客开始进入购买至第 m 个顾客购买完毕之间到底花费了多少时间，我们让实验一任务四的功能三调整后的代码重复执行 m 次，通过计算这 m 次循环执行前和执行后的时间差，来感受其影响。功能三之外的其他功能不纳入到这 m 次循环体内（但可以保留不变）。

提示：

1. 在进入功能二之前增加 m 次循环的初始化工作，在功能三结束之后增加 m 次循环的条件判断和转移语句。

2. 学校汇编教学网站的软件下载中提供了显示当前时间“秒和百分秒”的子程序。若在 m 次循环前调用一下该子程序， m 次循环执行完之后再调用一下该子程序，就能在屏幕上观察并感受到执行循环前后的时间差（时间差值需要自行手工计算，当然，你也可以选用网站上另一个计时程序，它是可以帮你计算好差值的）。注意，由于虚拟机环境下 CPU 会被分时调度，故该时间差值会因计算机运行环境与状态的不同而不同。

2.2 任务 2：对任务 1 中的汇编源程序进行优化

要求：

优化工作包括代码长度的优化和执行效率的优化，本次优化的重点是执行效率的优化。请通过优化 m 次循环体内的程序，使程序的执行时间尽可能减少 10% 以上。减少的越多，评价越高！

提示：

首先是通过选择执行速度较快的指令来提高性能，比如，把乘除指令转换成移位指令、加法指令等；其次，内循环体中每减少一条指令，就相当于减少了 $m \times n$ 条指令的执行时间，需要仔细斟酌；第三，尽量采用 32 位寄存器寻址，能有更多的机会提高指令执行效率。

3 实验过程

3.1 任务 1

3.1.1 实验步骤

1. 根据任务的要求将实验一的程序改写，去除与本次实验无关的部分，增加循环次数，为程序添加计时子程序并且调用。

2. 确保程序没有问题后汇编、连接。运行连接后的汇编程序，观察多重循环对 CPU 计算能力消耗的影响。

3.1.2 修改后源程序

首先，添加新的用于控制计时的 proc。它会将特定代码段重复执行特定次，并给出计时结果。

```
raddon_perf proc
    push ax
    push bx
    push cx
    push dx
    push di
    push si

    mov bx, 3000
    call rlib_showtime
raddon_perf_loop_begin:
    push bx
    ; manual longjmp
    jmp tmp10
raddon_perf_callback:
    pop bx
    dec bx
    jne raddon_perf_loop_begin
    call rlib_showtime

    pop si
    pop di
    pop dx
    pop cx
```

```
    pop bx
    pop ax
    ret
raddon_perf endp
```

然后，在程序刚开始运行时就直接调用 `raddon_perf`，在被测试代码段执行结束时自动跳回到 `raddon_perf_callback`，以便准备下一次循环。

```
...
mov ds,bx

; performance test
call raddon_perf

lea bx,tmpbuf2
...
;call rlib_print
; performance return
jmp raddon_perf_callback
jmp tmp1
```

为了自动化输入输出，我直接将要查询商品的名字写到 `.data` 缓冲区内，并将 `rlib_print/rlib_readstr` 全部转换为 `raddon_fake_print` 和 `raddon_fake_readstr`

```
...
raddon_fake_readstr proc
    pop ax
    pop bx
    push ax
    ret
raddon_fake_readstr endp
raddon_fake_print proc
    pop ax
    add sp, 2
    push ax
    ret
raddon_fake_print endp
...
...
push bx

;performance test
call raddon_fake_print
call raddon_fake_print
call raddon_fake_readstr
```

```
    call raddon_fake_print
    jmp tmp28
    ...

...
        db 8 dup(0)
m_product db 4, 'book', 8 dup(0)
pr1 dw 0
...
```

rlib_showtime 的实现如下:

```
rlib_showtime proc
; hour:min:sec:sec/100
    push ax
    push bx
    push cx
    push dx
    mov ah, 2ch
    int 21h
    mov bx, dx

    mov al, ch
    xor ah, ah
    call print2Digits
    mov dl, '.'
    mov ah, 02h
    int 21h

    mov al, cl
    xor ah, ah
    call print2Digits
    mov dl, '.'
    mov ah, 02h
    int 21h

    mov al, bh
    xor ah, ah
    call print2Digits
    mov dl, '.'
    mov ah, 02h
    int 21h

    mov al, bl
    xor ah, ah
    call print2Digits
    mov dl, 10
```

```

mov ah, 02h
int 21h

pop dx
pop cx
pop bx
pop ax
ret
rlib_showtime endp

print2Digits:
;; input in AX (0-99)
;; clobbers AX and DX, save them if needed
MOV DL, 0Ah ; divide by: 10
DIV DL ; first digit in AL (quotient), second digit in AH (remainder)
MOV DX, AX ; save the digits
ADD AL, 30h ; ASCII '0'
MOV AH, 0Eh ; set up print
INT 10h ; print first digit.
MOV AL, DH ; retrieve second digit
ADD AL, 30h
INT 10h ; print it
RET

```

3.1.3 实验记录与分析

1. 程序的改写主要是将实验一中任务四的源程序中的功能一和功能四删除，同时去除在功能二和功能三中无意义的输出的系统调用。在主程序中设定 m 次循环，并且在循环中依次调用功能二和功能三的子程序，并且在循环开始直接和循环结束之后调用相关的子程序，以统计循环执行的时间。修改之后的程序以及改动的部分如 2.1.2 所示，计时子程序的原理如图 3.1.2 所示。

2. 程序编译和连接的情况如下图所示。



```

C:\>make good
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1991. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta good.asm

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

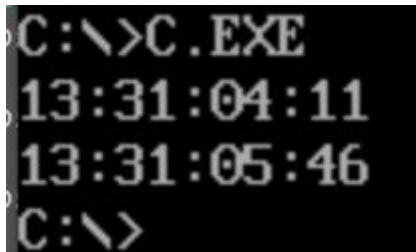
Assembling: good.asm

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

C:\>

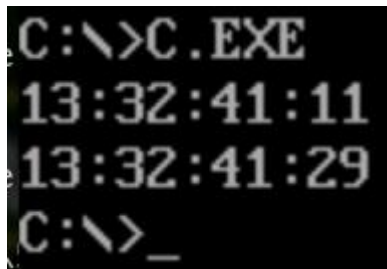
```

3. 设置 m 为 30000，运行后结果如下图所示(CPU 3MHz)。

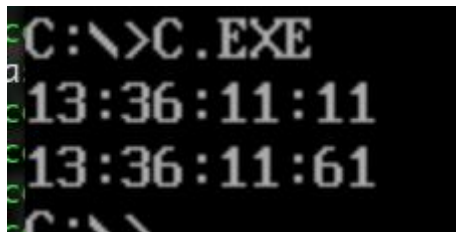


```
C:\>C.EXE
13:31:04:11
13:31:05:46
C:\>
```

4. 分别设置 m 为 3000/10000，运行后结果如下图所示。



```
C:\>C.EXE
13:32:41:11
13:32:41:29
C:\>_
```



```
C:\>C.EXE
13:36:11:11
13:36:11:61
C:\>
```

3.2 任务 2

3.2.1 实验步骤

1. 根据任务 1 中的源程序进行相应的修改，并编译到 bin，观察优化的效果，并且根据结果进行相关的分析。由于无法进行 profiling，我们无法确定应当重点优化的热点，只能尽可能猜测热点并进行优化。

2. 对可能的热点的优化的思路如下：

基本逻辑：例如不重复搜索，不重复计算。这是优化的最大因素，但在性能瓶颈往往不能使用。

缓存：由于程序数据段体积较小，在这里没有较大优化余地。

分支预测：这也是一个惩罚较高的优化项，在下面对每一个条件跳转进行了分析和适当修改。

考虑到循环的存在，我尽可能使是否跳转的结果为 true，从而使使用 global branch prediction 的现代 CPU 始终处于 strongly taken 的状态。

SIMD 和 SSE/AVX：不可使用。

并行优化：不可使用。

使用较低开销的指令：例如使用 `xor ah, ah` 代替 `mov ah, 0`，使用 `CMPS/CMPSB/CMPSW/CMPSD` 代替循环 `CMP`，使用加法或移位代替乘法，使用乘法代替除法等 C 编译器惯用的手法。优化余地一般不大。

只使用等效变换保证了优化的正确性。

3.2.2 优化后源程序

```
...
    mov ah,2
    xor dx, dx
    mov dl,al
...
    mov bx,[bx+1]
    xor bh, bh
```

```

        cmp bx,4
...
        mov ch,1[di][bx]
        and ch,ch
        cmp ch,cl
...
        sub m_product[1],0
jne tmp9
        jmp __rlib_start
tmp9:
        mov di,offset s1
...

        push cx
        mov si, bx
        lea di, inname
        mov cx, 5
        cld
        repz cmpsw
        je short queryfound
        pop cx
...

        m_product db 4, 'book', 8 dup(0)
        product_index db 0
        pr1 dw 0
...
        pop cx ; Use cache rather than calculate again.
        mov product_index, cx
...

        -jmp tmp13
        mov cx, product_index
...

        call raddon_fake_print
        call raddon_fake_print
        call raddon_fake_readstr
        call raddon_fake_print
...

        add ax,cx
        sar ax,1
        mov pr_sum,ax
...

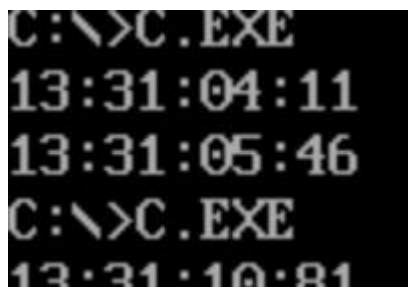
        mov ax,61h
        jl _tmp1
        jmp tmp19
_tmp1:

```

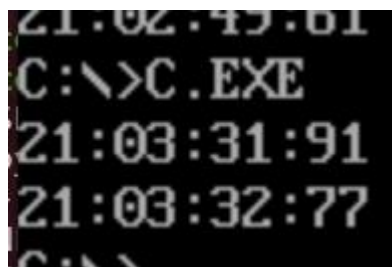
```
inc ax
cmp dx,50
jl _tmp2
jmp tmp19
_tmp2:
inc ax
cmp dx,20
jl _tmp3
jmp tmp19
_tmp3:
inc ax
...
etc...
```

3.2.3 实验记录与分析

1. 优化前的运行时间(m=30000)



2. 优化后的运行时间(m=30000)



经计算，优化后程序的运行时间由 1250ms 减少到了 860ms，效果较为符合预期。其中基本逻辑的优化(缓存)用最简单的方法贡献了大多数优化效果。其他常数级的小优化大多没有命中热点，没有很大的效果。

4 总结与体会

通过这次实验，我加深了对于这些知识的了解，能够更加熟练地在实践中运用已经学到的汇编知识，从底层发现程序的问题所在。同时，我强化练习了 TD 的使用方式，能够更加熟练地使用 TD 反汇编并且调试程序。

在任务 1 中，我主要根据第一次实验的任务四对程序进行了改写，由于第一次实验的任务四采用了子程序这种结构化的设计，所以对于程序的改写非常顺利，几乎没有遇到什么错误，这体现出了高度模块化，结构化的程序对于开发的积极作用。在该任务中，我编写了 `rlib_showtime`，更加了解了 DOS 系统调用。

在任务 2 中，我们需要对程序做优化。通过该任务，我掌握了汇编语言优化的基本方法，我主要尝试了以下方法：逻辑优化，分支预测优化，指令优化。例如使用单独的内存缓存结果，避免重复计算。利用现代 CPU 的分支预测机制，避免几十个周期的分支预测惩罚。利用复杂的指令去替代多个简单指令，包括使用串查找取代循环比较，使用 `movsx/movzx` 等。它们都取得了很好的效果。但如果能在 DOS 上应用 profiling 工具确定热点，将会更容易对昂贵操作进行更细致的优化。

在该任务中，我曾经遇到过性能优化超过 99% 的情况。我本来以为是偶然的跨循环缓存数据造成的优化结果并准备修正，但在后来的检查中，我很快发现这是由于我的优化错误地修改了程序的逻辑，使得程序提早地跳出了比较分支造成的。在经历这种错误之后，我在每一次优化完成之后都会使用 WD 加载程序并观察程序是否是正确执行。

5 参考文献

- [1] Intel(R) 64 and IA-32 Architectures Software Developer's Manual(<https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>)
- [2] (out-of-date) INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986(<https://css.csail.mit.edu/6.858/2015/readings/i386.pdf>)
- [3] (out-of-date) Open Watcom Toolset (<http://www.openwatcom.org/>)
- [4] (out-of-date) Microsoft ASM Language for MS-DOS (https://en.wikipedia.org/wiki/Microsoft_Macro_Assembler)
- [5] 80386 instruction set indexed by MIT.edu (<https://pdos.csail.mit.edu/6.828/2017/readings/i386/c17.htm>)
- [6] (out-of-date) MASM directives (<http://stanislavs.org/helppc/directives.html>)
- [7] (out-of-date) DOS Interrupts Reference by SCU.edu.au (<http://spike.scu.edu.au/~barry/interrupts.html>)
- [8] DOSBox: DOS Simulator for modern computer(not a VM) (<https://www.dosbox.com/wiki/>)
- [9] x86 arch introduction by wikibooks.org (https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)