

日期：2017.3.29

目录

1	实验目的与要求.....	1
2	实验内容.....	1
3	实验过程.....	2
3.1	任务 1	2
3.1.1	源代码	2
3.1.2	流程图	3
3.1.3	实验步骤	3
3.1.4	实验记录与分析	4
3.2	任务 2	6
3.2.1	优化思路	6
3.2.2	源代码	7
3.2.3	实验步骤	14
3.2.4	实验记录与分析	14
3.3	任务三	19
3.3.1	C 语言源程序	19
3.3.2	实验步骤	19
3.3.3	实验记录与分析	20
4	总结与体会.....	27
	参考文献	28

汇编语言程序设计实验报告

1 实验目的与要求

本次实验的主要目的与要求有下面 6 点，所有的任务都会围绕这 6 点进行，希望大家事后检查自己是否达到这些目的与要求。

- (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

任务 1. 观察多重循环对 CPU 计算能力消耗的影响

若有 m 个用户在同一台电脑上排队使用实验一任务四的程序，想要查询成绩列表中最后一个学生“xxxxxx”的平均成绩，那就相当于将实验一任务四的程序执行了 m 次。为了观察从第一个用户开始进入查询至第 m 个用户查到结果之间到底延迟了多少时间，我们让实验一任务四的功能二和功能三的代码重复执行 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。由于功能一和功能四需要输入、输出，速度本来就较慢，所以，没有纳入到这 m 次循环体内（但可以保留不变）。请按照上述设想修改实验一任务四的程序，并将 m 值尽量取大（建议 $m \geq 1000$ ，具体数值依据实验效果来改变，逐步增加到比较明显的程度，比如秒级延迟），以得到较明显的效果。

任务 2. 对任务 1 中的汇编源程序进行优化

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！

任务 3. 观察用 C 语言实现的实验一任务四中功能一的程序与汇编语言实现的程序的差异

用汇编语言和 C 语言分别实现实验一任务四中功能一的功能（对汇编语言而言，就是把实验一中相关程序摘取出来成为独立的程序），对比两种语言实现的程序的代码情况，观察和总结 C 语言编写程序和自己的汇编语言程序的对应关系及差异，总结其中可以简化的地方。

3 实验过程

3.1 任务 1

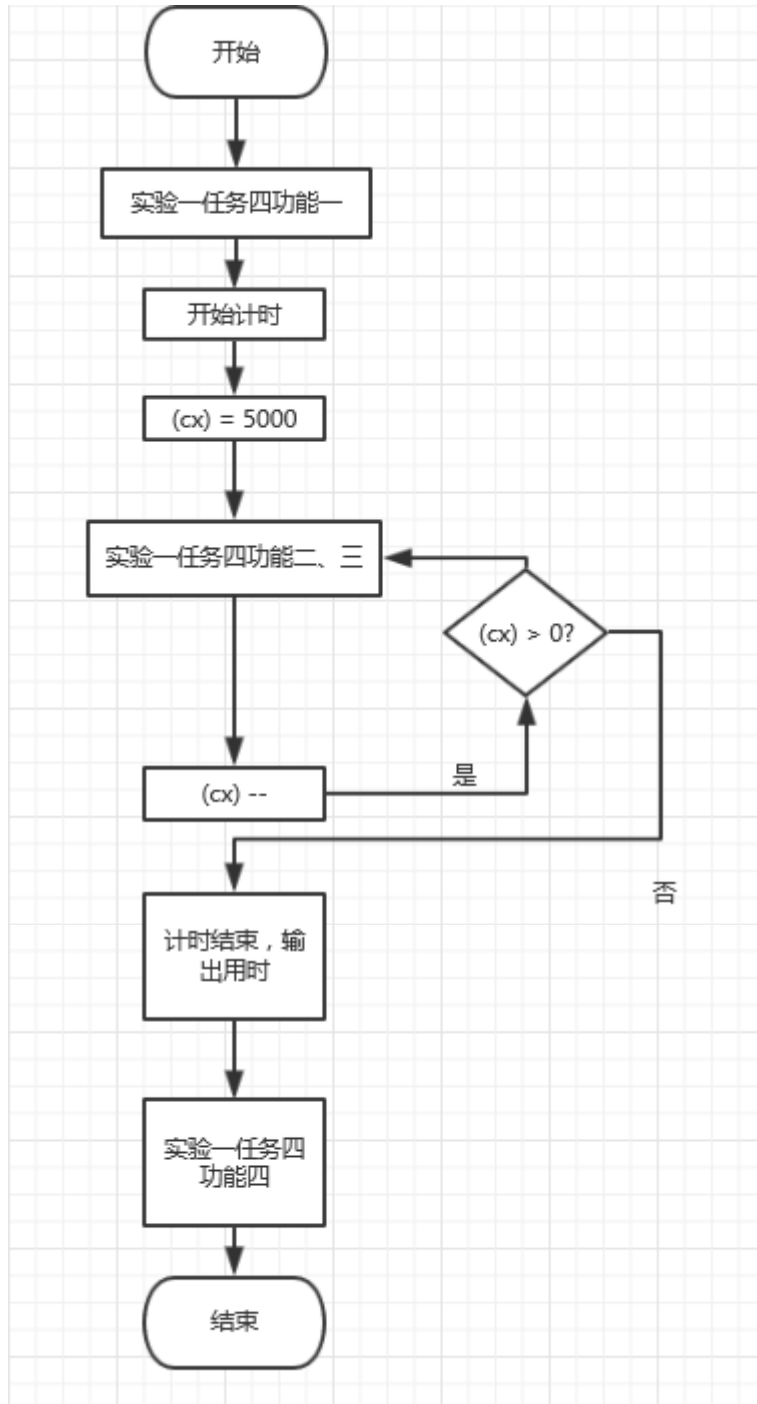
3.1.1 源代码

在汇编网站上下载 TIMER.ASM 文件, 加入到自己的程序当中, 实现计算函数执行时间的功能。

```
.....  
mov ax, 0    ; (此前为实验一任务四功能一代码)  
call TIMER  
mov cx, 1000  
work1: push cx  
.....    ; 这里为实验一任务四功能二、功能三代码  
pop cx  
mov ax, 1  
call TIMER  
.....    ; (这里为实验一任务四功能四的代码)  
将改动部分的代码摘录如上
```

汇编语言程序设计实验报告

3.1.2 流程图



3.1.3 实验步骤

1. 准备上机实验环境。
 - a) 操作系统: windows 10 下虚拟环境 DosBox 0.74
 - b) 编辑器 sublime text 3

汇编语言程序设计实验报告

2. 测试源代码所需时间
 - a) 将 m 定为 5000, 编译链接源程序
 - b) 执行程序, 搜索最后一名同学“xinjie”五次, 记录每次所需时间, 计算平均时间作为此时 m 下的当前代码效率。即 5000 次循环下未优化代码效率
3. 了解时间计算程序的原理
 - a) 研究 DOS 下的系统功能调用, 找到相关调用
 - b) 写测试程序, 测试该调用的实际功能
 - c) 查找资料确认自己的想法无误
4. 考虑 m 和程序执行时间的关系
 - a) 按倍数增加 m, 再次测试程序时间, 记录
 - b) 粗略估计 m 对时间的影响
5. 探究当前系统环境对程序执行时间的影响
 - a) 对 DOSbox cpu cycle 进行调整
 - b) 计算耗时

3.1.4 实验记录与分析

1. 实验环境条件

操作系统: windows 10 下 DOSBox 0.74

2. 编译连接源程序

```
D:\>masm STUDENT;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

49826 + 453323 Bytes symbol space free

0 Warning Errors
0 Severe Errors

D:\>link STUDENT;
Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

D:\>_
```

图 1-1 编译链接初始程序

未发生错误, 则执行程序, 查询最后一个人五次, 记录表格, 算出平均查询所需时间。测试时 循环执行次数等于 5000。

```
D:\>STUDENT.EXE
Please Input Name :xinjie
Time elapsed in ms is 1970
A
Please Input Name :xinjie
Time elapsed in ms is 1980
A
Please Input Name :xinjie
Time elapsed in ms is 1920
A
Please Input Name :xinjie
Time elapsed in ms is 1980
A
Please Input Name :xinjie
Time elapsed in ms is 1920
A
Please Input Name : ^
```

图 1-2 查询耗时测试

汇编语言程序设计实验报告

统计结果见表 1-1

表 1-1 初始程序查询耗时

第一次	第二次	第三次	第四次	第五次	平均
1970ms	1980ms	1920ms	1980ms	1920ms	1954ms

3. 研究 disptime 函数

查阅书籍了解到 DOS 功能调用中有取得时间的 2CH，调用之后，(CX):(DX)为时间，我决定写一个简单的程序来研究这个功能。

程序如下：

```
.386
codesg segment USE16
    assume cs:codesg
start:  mov ah,2CH
        int 21H
codesg ends
end start
```

使用 td 反汇编这个程序观察寄存器变化得到下图：（我是在 15: 04 的时候测试）

Address	Disassembly	Comment
cs:0000 B42C	mov ah,2C	
cs:0002 CD21	int 21	
cs:0004 00830672	or [bp+di+7206],al	
cs:0006 090A	or [bp+si],cx	

图 1-3 2CH 调用研究

将他转换为 10 进制变成容易观察的数字（每两位转换）：3845 0148 没有发现什么规律再次调用（15: 06 的时候测试）：

Address	Disassembly	Comment
cs:0000 B42C	mov ah,2C	
cs:0002 CD21	int 21	
cs:0004 00830672	or [bp+di+7206],al	
cs:0006 090A	or [bp+si],cx	

图 1-4 2CH 调用研究

转化为十进制：3847 26 65。观察到 cx 高两位不变，可能代表小时，cl 增加了 2，可能代表分钟，dx 可能代表秒级别。由于我不知道可以怎么样得到 dos 下面的时间，所以实验不太准确，于是网上查找了资料，得到下列结论：

CH=时(0-23), CL=分, DH=秒, DL=百分之几秒

可以看到猜测基本成立。这时候明白了 2CH 调用的机制，于是测试耗费的时间，我们可以得到前后两个时间，做减法即可。

4. 增加 m 测试循环次数对执行时间的影响：我们猜测他们是线性相关

保持操作系统环境不变，修改 m 的值，多次测试得到结论，结果记录见表 1-2

表 1-2 循环次数和执行时间的关系

M = 1000	M = 2000	M = 3000	M = 4000	M = 5000
390ms	770ms	1150ms	1540ms	1980ms

粗略观察可以发现基本呈线性相关。

5. 操作系统环境对速度的影响

汇编语言程序设计实验报告

CPU Cycles (speed up/slow down)
By default (cycles=auto) DOSBox tries to detect whether a game needs to be run with as many instructions emulated per time interval as possible (cycles=max, sometimes this results in game working too fast or unstable), or whether to use fixed amount of cycles (cycles=3000, sometimes this results in game working too slow or too fast). But you can always manually force a different setting in the DOSBox's configuration file.

图 1-5 dosBOX 文档

阅读文档后我们可以发现可以通过更改 cpu cycles 来改变操作系统环境

```
C:\>STUDENT.EXE
Please Input Name :xinjie
Time elapsed in ms is 1930
A
Please Input Name :q
C:\>_
```

图 1-6 默认速度

```
C:\>cycles = auto 5000 100% limit 200000
C:\>STUDENT.EXE
Please Input Name :xinjie
Time elapsed in ms is 1160
A
Please Input Name :q
C:\>cycles = auto 100000 100% limit 200000
C:\>STUDENT.EXE
Please Input Name :xinjie
Time elapsed in ms is 50
A
Please Input Name :q
```

图 1-7 调节 CPU cycles

可以看到计算时间是会受到系统环境的影响而改变的

3.2 任务 2

3.2.1 优化思路

1. 替换执行速度较慢的指令

将可以替换的乘除指令替换为移位指令、加减指令。

这一部分主要出现在计算平均值的函数中：原本的代码中发生了两次乘法、两次除法：语文成绩 * 2，总成绩 * 2，英语成绩 / 2，总成绩 / 7。对于这四个运算，我们需要找到可以替代他们的方案。乘以二和除以二可以简化为位移操作，乘以二左移，除以二右移，又由于数字不太大，我们可以放心移位而不用担心溢出。

对于除以 7 的优化，目前有这些思路，首先是将除法换位减法，利用高速的减法代替低速的除法，第二种方法观察到被除数最大为 700 不算太大，可以建立一张结果表，每次访问这张表即可。

2. 减少指令数

这一部分主要出现在查询过程中，每减少一个指令就能减少 $m * 30 * 10$ 次执行，很值得自己去减少。使用 MOVZX 可以减少指令数。将高位置为 0 这种指令全部去除。

3. 使用 32 位寄存器

汇编语言程序设计实验报告

对寄存器的访问要快于对内存的访问，在代码中使用更大的寄存器可以减少更多对内存的访问。在计算平均值时发生了三次内存访问（这些内存是连续的）所以可以优化。在查询中比对每个字母，可以每次比对多个字母，加快速度。两个 32 位寄存器，一个 16 位寄存器，三次比较节约时间。

3.2.2 源代码

```
.386
STACK SEGMENT USE16 STACK
    DB 300 DUP(0)
STACK ENDS

DATA SEGMENT USE16
N      EQU 30
POIN   DW 0

BUF     DB 'zhangsan', 0, 0
        DB 0, 0, 0, ?
        DB 'lisi', 6 DUP(0)
        DB 80, 100, 70, ?
        DB 'B', 0, 0, 0, 0, 0, 0, 0, 0
        DB 10, 20, 12, ?
        DB N-4 DUP('TempValue', 0, 80, 90, 95, ?)
        DB 'xinjie', 0, 0, 0, 0
        DB 100, 100, 100, ?

IN_NAME DB 10
        DB 0
        DB 10 DUP(0)
STRING  DB 300 DUP(0)
CRLF    DB 0DH, 0AH, '$'
MSG1     DB 0AH, 0DH, 'Please Input Name :$'
MSG2     DB 0AH, 0DH, 'Not Find This Student!:$'
MSG3     DB 'Rank: '
DIVID    DB 7 DUP(0)
        db 7 dup(1)
        db 7 dup(2)
        db 7 dup(3)
        db 7 dup(4)
        db 7 dup(5)
        db 7 dup(6)
        db 7 dup(7)
        db 7 dup(8)
        db 7 dup(9)
        db 7 dup(10)
        db 7 dup(11)
        db 7 dup(12)
```

汇 编 语 言 程 序 设 计 实 验 报 告

```
db 7 dup(13)
db 7 dup(14)
db 7 dup(15)
db 7 dup(16)
db 7 dup(17)
db 7 dup(18)
db 7 dup(19)
db 7 dup(20)
db 7 dup(21)
db 7 dup(22)
db 7 dup(23)
db 7 dup(24)
db 7 dup(25)
db 7 dup(26)
db 7 dup(27)
db 7 dup(28)
db 7 dup(29)
db 7 dup(30)
db 7 dup(31)
db 7 dup(32)
db 7 dup(33)
db 7 dup(34)
db 7 dup(35)
db 7 dup(36)
db 7 dup(37)
db 7 dup(38)
db 7 dup(39)
db 7 dup(40)
db 7 dup(41)
db 7 dup(42)
db 7 dup(43)
db 7 dup(44)
db 7 dup(45)
db 7 dup(46)
db 7 dup(47)
db 7 dup(48)
db 7 dup(49)
db 7 dup(50)
db 7 dup(51)
db 7 dup(52)
db 7 dup(53)
db 7 dup(54)
db 7 dup(55)
db 7 dup(56)
db 7 dup(57)
db 7 dup(58)
db 7 dup(59)
```

汇编语言程序设计实验报告

```
db 7 dup(60)
db 7 dup(61)
db 7 dup(62)
db 7 dup(63)
db 7 dup(64)
db 7 dup(65)
db 7 dup(66)
db 7 dup(67)
db 7 dup(68)
db 7 dup(69)
db 7 dup(70)
db 7 dup(71)
db 7 dup(72)
db 7 dup(73)
db 7 dup(74)
db 7 dup(75)
db 7 dup(76)
db 7 dup(77)
db 7 dup(78)
db 7 dup(79)
db 7 dup(80)
db 7 dup(81)
db 7 dup(82)
db 7 dup(83)
db 7 dup(84)
db 7 dup(85)
db 7 dup(86)
db 7 dup(87)
db 7 dup(88)
db 7 dup(89)
db 7 dup(90)
db 7 dup(91)
db 7 dup(92)
db 7 dup(93)
db 7 dup(94)
db 7 dup(95)
db 7 dup(96)
db 7 dup(97)
db 7 dup(98)
db 7 dup(99)
db 7 dup(100)
```

DATA ENDS

CODE SEGMENT USE16

ASSUME DS:DATA, CS:CODE, SS:STACK

START:

汇编语言程序设计实验报告

```
MOV AX, DATA
```

```
MOV DS, AX
```

INPUT:

```
MOV DX, OFFSET MSG1
```

```
MOV AH, 9
```

```
INT 21H ;功能一一小题
```

```
LEA DX, IN_NAME
```

```
MOV AH, 10
```

```
INT 21H ;功能一二小题
```

```
MOV BL, IN_NAME + 1
```

```
MOV BH, IN_NAME + 2
```

```
CMP BL, 0
```

```
JE INPUT
```

```
CMP BH, 'q'
```

```
JE DIE ;功能一 三小题
```

```
MOV BH, 0
```

```
MOV CX, 10
```

```
SUB CX, BX
```

PP: MOV [IN_NAME + BX + 2], 0

```
INC BX
```

```
LOOP PP
```

```
mov ax,0
```

```
call TIMER
```

```
mov cx, 5000
```

work1:

```
push cx
```

```
MOV DI, -14
```

FIND:

```
MOV CX, N
```

```
ADD DI, 14
```

FIND_S:

```
MOV EAX, DWORD PTR [IN_NAME + 2]
```

```
CMP EAX, DWORD PTR [BUF + DI]
```

```
JNE CON
```

```
MOV EBX, DWORD PTR [IN_NAME + 6]
```

```
CMP EBX, DWORD PTR [BUF + DI + 4]
```

```
JNE CON
```

```
MOV DX,  WORD PTR  [IN_NAME + 10]
```

汇编语言程序设计实验报告

```
CMP DX, WORD PTR [BUF + DI + 8]
JE SUCCESS_FIND
```

```
CON: CMP CX, 0
JE NOT_FIND
LOOP FIND
```

DIE:

```
MOV AH, 4CH
INT 21H
```

NOT_FIND:

```
MOV DX, OFFSET MSG2
MOV AH, 9
INT 21H
JMP INPUT
```

SUCCESS_FIND:

```
MOV WORD PTR [POIN], OFFSET BUF + 10
ADD WORD PTR [POIN], DI
CALL SET_AVERAGE_GRADE
```

```
pop cx
loop work1
```

```
mov ax,1
call TIMER
```

```
CALL G_ABCD
JMP INPUT
```

G_ABCD:

```
PUSH AX
PUSH DX
PUSH SI
MOV SI, [POIN]
ADD SI, 3
MOV AX, [SI]
MOV AH, 0
SUB AL, 90
JS G_BCD
MOV DL, 'A'
JMP SCREEN
```

G_BCD:

```
MOV AX, [SI]
MOV AH, 0
```

汇编语言程序设计实验报告

```
SUB AL, 80
JS  G_CD
MOV DL, 'B'
JMP SCREEN
G_CD:
MOV AX, [SI]
MOV AH, 0
SUB AL, 70
JS  G_D
MOV DL, 'C'
JMP SCREEN
G_D:
MOV DL, 'D'
JMP SCREEN
SCREEN:
MOV AH, 2
INT 21H
POP SI
POP DX
POP AX
RET

SET_AVERAGE_GRADE:
MOV SI, 10
MOV CX, N
MATH:
MOV EDX, DWORD PTR [BUF + SI]
MOVZX BX, DL ;Chinese Grade
SHL BX, 1
MOVZX AX, DH ;MATH GRADE
ADD BX, AX
SHR EDX, 8
MOV AL, DH ;ENGLISH
SHR AX, 1
ADD BX, AX
SHL BX, 1
MOVZX AX, BYTE PTR [BX + DIVID]
MOV [BUF + SI + 3], AL
ADD SI, 14
LOOP MATH
RET
```

;时间计数器(ms),在屏幕上显示程序的执行时间(ms)

;使用方法:

```
;      MOV  AX, 0      ;表示开始计时
;      CALL TIMER
```

汇编语言程序设计实验报告

```
;      ... .. ;需要计时的程序
;      MOV  AX, 1
;      CALL TIMER      ;终止计时并显示计时结果(ms)
;输出: 改变了 AX 和状态寄存器
TIMER  PROC
    PUSH  DX
    PUSH  CX
    PUSH  BX
    MOV   BX, AX
    MOV   AH, 2CH
    INT   21H          ;CH=hour(0-23),CL=minute(0-59),DH=second(0-59),DL=centisecond(0-100)
    MOV   AL, DH
    MOV   AH, 0
    IMUL  AX, AX, 1000
    MOV   DH, 0
    IMUL  DX, DX, 10
    ADD   AX, DX
    CMP   BX, 0
    JNZ   _T1
    MOV   CS:_TS, AX
_T0: POP  BX
    POP  CX
    POP  DX
    RET
_T1: SUB  AX, CS:_TS
    JNC   _T2
    ADD   AX, 60000
_T2: MOV  CX, 0
    MOV  BX, 10
_T3: MOV  DX, 0
    DIV  BX
    PUSH DX
    INC  CX
    CMP  AX, 0
    JNZ  _T3
    MOV  BX, 0
_T4: POP  AX
    ADD  AL, '0'
    MOV  CS:_TMSG[BX], AL
    INC  BX
    LOOP _T4
    PUSH DS
    MOV  CS:_TMSG[BX+0], 0AH
    MOV  CS:_TMSG[BX+1], 0DH
    MOV  CS:_TMSG[BX+2], '$'
    LEA  DX, _TS+2
    PUSH CS
```

汇编语言程序设计实验报告

```
POP    DS
MOV     AH, 9
INT     21H
POP     DS
JMP     _T0
_TS     DW    ?
        DB    0AH, 0DH, 'Time elapsed in ms is '
_TMSG    DB    12 DUP(0)
TIMER    ENDP

CODE ENDS
        END START
```

3.2.3 实验步骤

1. 准备上机环境，编译链接之前代码，确保无误。
 - a) 操作系统： windows 10 下虚拟环境 DosBox 0.74
 - b) 编辑器： sublime text 3
2. 对需要优化代码段进行分析，找出可以优化的部分
 - a) 只对查询部分做时间计算，计算查询耗时
 - b) 只对平均值部分做时间计算，计算平均值耗时
 - c) 对比两个子功能耗时，得出最占用系统时间的部分，大力优化，另一个在之后优化。
3. 逐步优化代码
 - a) 将乘除法换成移位运算
 - b) 将除以 7 换成减法，测试效率，换成查表，测试效率，对比取其中优化效率高的
 - c) 将高位置零语句换成 `movzx` 的语句，减少指令数
 - d) 使用 32 位寄存器来减少内存访问
4. 总结优化措施对效率的影响
 - a) 更改算数运算指令对效率的影响
 - b) 更改算法对效率的影响
 - c) 减少访问内存对效率的影响
 - d) 减少指令数对效率的影响

3.2.4 实验记录与分析

1. 编译连接源程序

汇编语言程序设计实验报告

```
q
C:\>masm youhua;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

49828 + 451274 Bytes symbol space free

0 Warning Errors
0 Severe Errors

C:\>link youhua;
Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
```

图 2-1 编译链接源程序

2. 分析需要优化的代码:

我们测试的是任务二、任务三两个任务，循环 5000 次测试的总时间在 1950ms 左右。

对比任务二的耗时和任务三的耗时

计算平均值耗时的代码:

```
mov ax,0
call TIMER
mov cx, 5000
work1:  CALL SET_AVERAGE_GRADE
loop work1
mov ax,1
call TIMER
```

```
C:\>STUDENT.EXE

Please Input Name :xin
Time elapsed in ms is 1090
```

图 2-2 计算平均值耗时测试

计算查找耗时的代码:

.....

```
mov ax,0 ; 此前为实验一任务四功能一代码
```

```
call TIMER
```

```
mov cx,1000
```

```
work1: push cx
```

```
..... ; 这里为实验一任务四功能二代码
```

```
pop cx
```

```
mov ax, 1
```

```
call TIMER
```

```
..... ; 这里为实验一任务四功能四的代码
```

汇编语言程序设计实验报告

```
C:\>STUDENT.EXE
Please Input Name :xinjie
Time elapsed in ms is 880
^
Please Input Name :q
```

图 2-3 查找耗时

通过测试两个部分代码的耗时,我们看书两个部分耗时差别不是很大,算平均值更加耗时一点。所以对这两部分的优化都很重要。

3. 逐步优化代码

a) 对算数运算的优化

表 2-1 优化乘除

优化前代码	优化后代码
MOV AH, 2 MUL AH	SHL BX, 1
MOV DL, 2 DIV DL	SHR AX, 1

优化这样子优化第一减少了一行指令,第二将乘法除法变成不了所需机器周期更少的移位操作。

b) 对除以 7 的优化

使用查表法进行优化,使用 C 语言生成查表法所需要的表.查表的时间复杂度为 $O(1)$,在算法层面上进行优化。

C 语言程序:

```
#include <stdio.h>
int main()
{
    int i = 0;
    for(i = 0; i <= 100; i++)
        printf("db 7 dup(%d)\n", i);
    return 0;
}
```

操作细节:

```
D:\ASM>gcc cTest.c
D:\ASM>a >> lingshi
D:\ASM>
```

图 2-4 生成表

汇编语言程序设计实验报告

```
1 db 7 dup(0)
2 db 7 dup(1)
3 db 7 dup(2)
4 db 7 dup(3)
5 db 7 dup(4)
6 db 7 dup(5)
7 db 7 dup(6)
8 db 7 dup(7)
9 db 7 dup(8)
10 db 7 dup(9)
11 db 7 dup(10)
12 db 7 dup(11)
13 db 7 dup(12)
14 db 7 dup(13)
```

图 2-5 表

我们定义 700 个数据，假设作为了一个数组，这个数组中第 i 个数字就是 i/7 得到的值，因为我们每隔七个这个值增加一，所以每隔七个更新这个值，这样我们计算成绩除以七的时候就可以直接访问内存单元了。

优化前代码：

```
MOV BL, 7
DIV BL
```

优化后代码：

```
MOVZX AX, BYTE PTR [BX + DIVID]
```

省去一条代码并且简化了指令

- e) 将高位置零语句换成 movzx 的语句，减少指令数

表 2-2 优化 movzx

优化前	优化后
MOV BX, 0 MOV BL, [BUF + SI + 1]	MOVZX BX, DL

代码中有四处这样的代码，可以减少四条代码，movzx 是内置的语句，cpu 周期数少于两条语句加起来的周期数，所以可以这样优化。

- f) 在求平均值的程序中共有三处访问内存的操作，可以使用 32 位寄存器只访问一次从而加快速率

```
MOV AL, [BUF + SI]
MOV AH, 2
MUL AH ;AX IS CHINESE * 2 <= 200 16 IS ENOUGH
MOV BL, [BUF + SI + 1] ;MATH GRADE
MOV BH, 0
ADD BX, AX
MOV AL, [BUF + SI + 2] ;ENGLISH
MOV AH, 0
```

图 2-6 访问三次内存

```
MATH:
MOV EDX, DWORD PTR [BUF + SI]
```

图 2-7 使用 32 位寄存器只访问内存一次

- g) 优化查找
最初版本的查找是一个一个字符比较，我们可以优化成四个四个比较，剩下的两个两

汇编语言程序设计实验报告

个比较,可以减少内存访问,减少循环,加快速度。内存访问的速度要比访问寄存器慢几百上千倍,即使根据局部性原理我们的数据都在一级缓存内,那么速度也还会满几十倍。

程序:

```
MOV DI, -14
```

```
FIND:
```

```
    MOV CX, N
```

```
    ADD DI, 14
```

```
FIND_S:
```

```
    MOV EAX, DWORD PTR [IN_NAME + 2]
```

```
    CMP EAX, DWORD PTR [BUF + DI]
```

```
    JNE CON
```

```
    MOV EBX, DWORD PTR [IN_NAME + 6]
```

```
    CMP EBX, DWORD PTR [BUF + DI + 4]
```

```
    JNE CON
```

```
    MOV DX,  WORD PTR  [IN_NAME + 10]
```

```
    CMP DX,  WORD PTR [BUF + DI + 8]
```

```
    JE SUCCESS_FIND
```

```
CON: CMP CX, 0
```

```
    JE NOT_FIND
```

```
    LOOP FIND
```

4. 效率总结

初始时间	优化后时间	优化程度
1970ms	1100ms	44.16%
1980ms	1100ms	44.44%
1980ms	1100ms	44.44%

经过上述几个步骤的优化之后,执行效率提升了大约 44%

对于程序的优化来说: 算法优化效果提升 > 算数指令优化效果提升 > 大于减少内存寻址
优化效果提升 > 减少指令数目效果提升

当然上面都是粗略估计的结果

3.3 任务三

3.3.1 C 语言源程序

```
#include <stdio.h>
int main()
{
    char in_name[10] = "";
    while(1)
    {
        printf("Please input Student's name!\n");
        gets(in_name);
        if(in_name[0] == '\0')
            continue;
        else if(in_name[0] == 'q' && in_name[1] == '\0')
            break;
        else
            break;
    }
    return 0;
}
```

3.3.2 实验步骤

1. 准备开发环境
 - a) 操作系统：windows 10 下虚拟环境 DosBox 0.74 版本
 - b) 编辑器：sublime text 3
 - c) C 语言编译器：Borland C++ 3.1
2. 编译链接 C 语言源程序，运行 EXE 文件，确定是否与汇编程序所运行情况相同
 - a) 编译链接 C 程序，运行，输入回车，回车，q，查看结果
 - b) 编译链接汇编程序，运行，输入回车，回车，q，查看结果
 - c) 两个结果相同，进行下一步，结果不同，分析原因修改程序
3. 观察两个 EXE 文件长度，若相差悬殊，思考原因
 - a) 将两个 EXE 文件大小进行对比，以一点五倍作为评判“悬殊”的标准
 - b) 思考 C 语言和汇编语言的编译过程，明白为什么造成文件大小不同
4. 反汇编 C 语言程序，观察生成的汇编代码的整体结构，看其中使用了那些段？这些段是用来干什么的？
 - a) 使用 bcc -S 命令来生成 C 语言对应的汇编代码，保存为 CTEST.ASM
 - b) 观察文件总体结构，找到各段名，根据段名推测这个段用来干什么

汇编语言程序设计实验报告

- c) 根据上述的猜测, 写简短 C 程序来确认或者否定自己的猜想
- d) 若确认成功, 思考为什么要这样分配, 若确认失败, 查阅资料了解详细情形
- 5. 观察代码段, 举例说明 C 编译器是如何将一条 C 语句翻译成机器指令的
 - a) 使用 TD 反汇编 C 语言的 EXE 文件
 - b) 首先观察变量定义的翻译过程
 - i. 不包含头文件(头文件导致了大量对此次任务无用的东西)写处测试程序
 - ii. 对各种变量类型: int , char , float , static int, char *, int [], long 进行定义
 - iii. 反汇编上述测试程序, 观察如何翻译
 - c) 然后观察加减法运算的翻译过程
 - i. 对加减乘除都进行使用, 写出测试程序
 - ii. 反汇编上述测试程序, 观察如何翻译成机器指令
 - d) 关于函数调用
 - i. 进行最简单的函数调用
 - ii. 反汇编上述程序, 观察如何翻译
- 6. 对与实验任务一相同功能的代码反汇编, 分析对应关系
 - a) 继续反汇编 CTEST.EXE 文件, 使用前文得出的结论观察反汇编的代码
 - b) 对输出字符串、比较字符串的部分重点比较
- 7. 探讨 Borland C++ 3.1 编译器对 C 语言的优化策略
 - a) 了解如何设置优化选项
 - b) 观察 Borland C++是从哪几个方面对 C 语言进行优化的

3.3.3 实验记录与分析

- 1. 准备开发环境
 - a) 从汇编网站下载好 BC 包, 测试是否可以使用

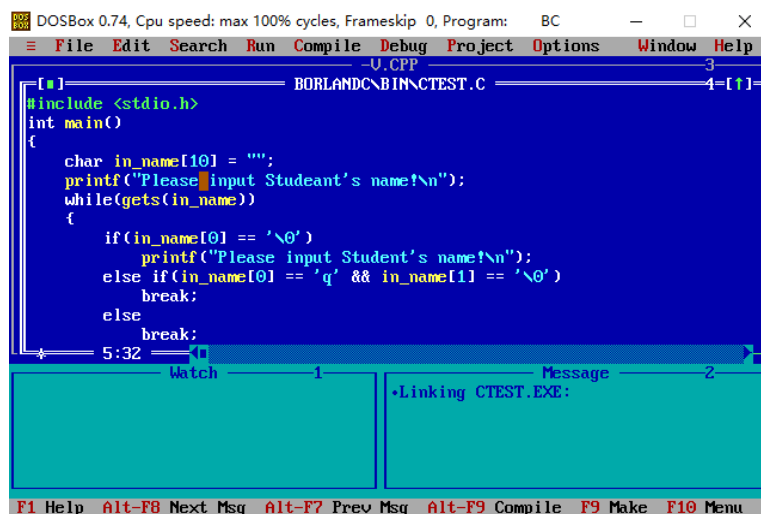


图 3-1 BC 界面

- b) 检查自己的软件版本都无误, 进行下一步

汇编语言程序设计实验报告

2. 编译 C 语言程序，执行。

```
C:\>CTEST.EXE
Please input Student's name!

Please input Student's name!

Please input Student's name!

q
```

图 3-2 C 语言执行情况

```
Please Input Student' Name :
Please Input Student' Name :
Please Input Student' Name :
q
```

图 3-3 汇编语言执行情况

C 语言使用 `gets()` 读入回车也会回显而汇编没有造成了这里的一些区别，不过对后来的实验没有什么影响。之后的实验就使用这两个文件。

3. 对比两个 EXE 文件长度

ASMTEST.EXE	2017/3/30 15:09	应用程序	1 KB
CTEST.EXE	2017/3/30 15:11	应用程序	8 KB

图 3-4 EXE 文件大小对比

相比来看，C 语言 EXE 文件是汇编语言生成的 EXE 文件的 8 倍，足以到达“悬殊”的标准。

思考原因，C 语言程序包含了“`stdio.h`”标准库，C 语言在链接标准库时可能加入了比汇编代码多得多的东西。而且经过后来的学习，C 语言还会生成调试表，这也样数据量更加增大。

4. 反汇编 C 语言程序，观察生成的汇编代码的整体结构，看其中使用了那些段？这些段是用来干什么的？

a) 使用 `bcc -S` 生成对应的汇编代码

```
C:\>bcc -S CTEST.C
Borland C++ Version 3.1 Copyright (c) 1992 Borland International
ctest.c:

Available memory 4200392
```

图 3-5 bcc 生成反汇编

b) 观察文件总体结构，找到各段名，根据段名推测这个段用来干什么

通过观察发现以下几个段名：`_TEXT`、`_DATA`、`_BSS`

通过命名推测：`_TEXT` 段中存放代码，`DATA` 段中存放数据（具体是什么数据不能轻易判断），`BSS` 段暂时未知

c) 观察文件内容，验证自己的结论

在 `_DATA` 段中发现了我们要 `printf` 的语句，以及 `DATA` 段除了定义数据以外没有其他代码，所以 `DATA` 段中的确是存放数据。

在 `_TEXT` 段中发现了函数执行的主体，`_TEXT` 段的确是代码段

通过百度、Google，了解到 `_BSS` 段存放的是未初始化的全局变量

d) 写简短的验证程序验证自己的猜想以及验证资料

```
int x = 5;
```

汇编语言程序设计实验报告

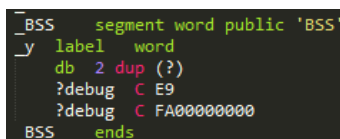
```
int y;  
main()  
{  
    int i = 5;  
    i++;  
}
```

设置了两个全局变量，一个初始化，一个未初始化，一个局部变量，一条自增语句。

预测结果：_BSS 段中有对 y 变量的说明，_DATA 段中有对 x I 变量的说明，_TEXT 段中有自增的指令

实际结果：

Y 变量在_BSS 段中, X 变量在_DATA 段中, 但是 i 却不在, 自增指令也在_TEXT 段中。思考查阅资料后获悉，_DATA 段中存放的应该是已初始化的全局变量。



```
_BSS    segment word public 'BSS'  
_y     label word  
       db 2 dup (?)  
       ?debug    C E9  
       ?debug    C FA00000000  
_BSS    ends
```

图 3-6 _BSS 段

e) 思考 C 语言这样设置的合理性

_DATA 段和_TEXT 段的必要性不言而喻，一个用来访问数据，一个存放代码，而 BSS 段经过查找资料得到应该也是为了节省内存，但具体也不是很清楚，这并不是本次实验的重点。

f) 总结：

- i. 这次观察到了 text 段，data 段，bss 段
- ii. Text 段用来存放已编译的程序的机器代码
- iii. Data 段用来存放已经初始化的全局和静态变量，局部变量存放在栈中
- iv. Bss 段存放未初始化的全局和静态 c 变量，不占用实际的空间，仅仅是占位符。Bss 是（Block Storage Start）的缩写。也可以看成 Better Save Space 方便理解。

5. 观察代码段，举例说明 C 编译器是如何将一条 C 语句翻译成机器指令的

a) 首先观察对定义变量的操作

C 语言源程序：

```
int x = 5;  
int y;  
static s = 1;  
main()  
{  
    int i = 5;  
    float f = 3.0;  
    int *p = &i;  
    char c = 'c';  
    int arr[5] = {1,2,3,4,5};  
}
```

生成对应的反汇编代码之后观察：

汇编语言程序设计实验报告

```
mov word ptr [bp-2],5
; float f = 3.0;
mov word ptr [bp-4],16448
mov word ptr [bp-6],0
; int *p = &i;
lea ax,word ptr [bp-2]
mov word ptr [bp-8],ax
; char c = 'c';
mov byte ptr [bp-9],99
; int arr[5] = {1,2,3,4,5};
lea ax,word ptr [bp-20]
```

图 3-7 变量定义

对于 x s 将他们两个放入了_DATA 段，而函数内部的变量 I, f, p, c 则都放入了栈空间。而数组 arr 中的元素首先是把字面量 1, 2, 3, 4, 5 存放到了 DATA 段中，初始化的时候又从 DATA 中取出来送到栈中。

表 3-1 定义变量分析

C 语言	汇编语言
char c = 'c';	mov byte ptr [bp - 9], 99
int i = 5;	mov word ptr [bp - 2], 5

b) 观察加减乘除指令

C 语言测试程序：

```
main()
{
int i = 5;
int j = i + 2;
j = i * 3;
j = i / 3;
}
```

```
mov ax,cx
mov dx,3
imul dx
mov si,ax
```

图 3-8 乘法

反汇编后观察汇编指令为 imul、 idiv 、 add，和汇编语言的语法基本一致。

表 3-2 数学运算分析

C 语言	汇编语言
j = j * 3	mov dx, 3 imul dx

c) 观察函数调用：

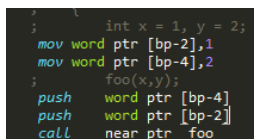
C 语言程序：

汇编语言程序设计实验报告

```
main()
{
    int x = 1, y = 2;
    foo(x,y);
}

int foo(int i, int j)
{
    return i + j;
}

调用部分:
```



```
int x = 1, y = 2;
mov word ptr [bp-2],1
mov word ptr [bp-4],2
foo(x,y):
push word ptr [bp-4]
push word ptr [bp-2]
call near ptr _foo
```

图 3-9 函数调用

可以发现是先将实参入栈，然后对栈内的元素操作。

- d) 探究了 C 语言三个基本的语法生成的反汇编代码：变量定义，数学计算，函数调用。了解了变量在段中的存放，数学计算的翻译(还实验过乘以 2 除以二，直接优化为了位移操作)，函数调用的细节。

6. 对与实验任务一相同功能的代码反汇编，分析对应关系

- a) 继续反汇编 CTEST.EXE 文件，使用前文得出的结论观察反汇编的代码

表 3- 2 C 语言代码与汇编代码对照表

C 语言语句	反汇编语句
char in_name[10] = "";	lea ax,word ptr [bp-10] mov ax,offset DGROUP:d@w+0
while(1){	@1@58:
printf("xxx");	mov ax,offset DGROUP:s@ push ax call near ptr _printf pop cx
gets(in_name);	lea ax,word ptr [bp-10] push ax call near ptr _gets pop cx
if(in_name[0] == '\0')	cmpbyte ptr [bp-10],0 jne short @1@142
continue;	jmp short @1@282

汇 编 语 言 程 序 设 计 实 验 报 告

<pre>else if(in_name[0] == 'q' && in_name[1] == '\0')</pre>	<pre>@1@142: cmpbyte ptr [bp-10],113 jne short @1@254 cmpbyte ptr [bp-9],0 jne short @1@254</pre>
<pre>break;</pre>	<pre>jmp short @1@310 jmp short @1@282</pre>
<pre>else break;}</pre>	<pre>jmp short @1@310 @1@282: jmp short @1@58 @1@310:</pre>

- b) 对输出字符串、比较字符串的部分重点比较

观察字符串比较部分：

in_name 中的数据存放到了栈空间当中，当需要比较的时候，直接比较栈空间的这部分数据，使用跳转指令 jnp 实现转移。

7. 探讨 Borland C++ 3.1 编译器对 C 语言的优化策略

- a) 了解如何设置优化选项

在网上查找了 borland c++ 3.1 的资料，找到了优化选项在 option -> compiler -> optimization 下，打开该选项，为下图：

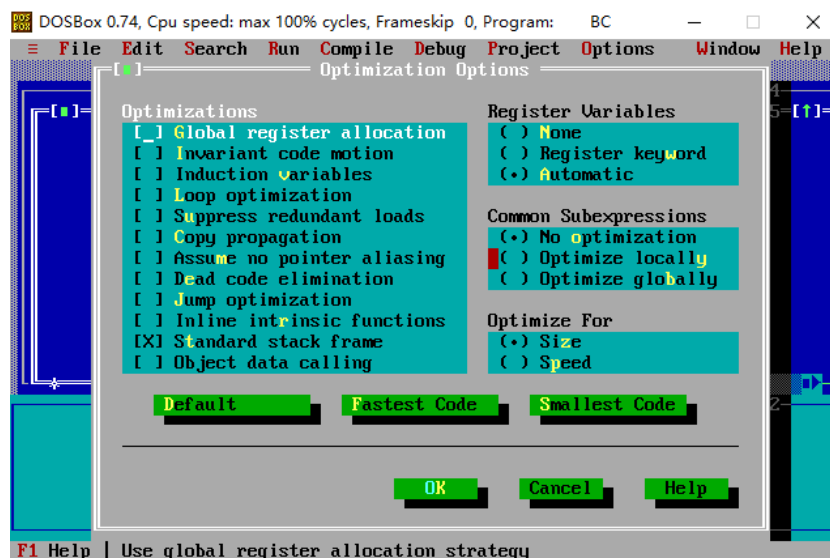


图 3-10 优化方面

- b) Borland C++是从哪些方面对 C 语言进行优化的：

- i. 全局寄存器分配(Global register allocation)CPU 寄存器数量有限，无法保存所有的变量，所以如何分配就很重要。使寄存器尽量达到最高效化工作就是这个优化。

汇 编 语 言 程 序 设 计 实 验 报 告

- ii. 不变代码移出 (Invariant code motion)
 - iii. 循环优化 (Loop optimization): 有以下几种:
 - 循环展开, 例如 `for(I = 0; I <= 3; i++) x = 1;` 优化为 `x = 1; x = 2; x = 3;`
 - 还有自动向量化和循环不变量代码移动
 - iv. Suppress redundant loads 代码垃圾移除
 - v. Copy propagation(复写传播) 比方说 `int x=1; int y = x; y++;`
可以被优化为 `int x = 1; x++;` 删去了不必要的 `y`
 - vi. Assume no pointer aliasing(不使用别名)
 - vii. Dead code elimination(无用代码消除)
 - viii. Jump optimization 跳转优化
 - ix. Inline intrinsic functions 使用内联函数 (内联函数的概念在 C++ 中有)
 - x. Standard stack frame 标准栈框架
 - xi. Object data calling 对象数据优化 (可能是对 C++ 支持的优化操作)
- c) 总结: 对 C 语言的优化主要是两个方面: 第一个是对语句的优化, 减少函数调用, 循环调用; 第二点是对硬件设施分配的优化, 寄存器分配等。

4 总结与体会

实验一是一个修改自己的程序，增加显示时间功能的任务。乍一看似乎没什么难度，不过要想细细思考其中的原理也是别有趣味。DOS 系统的时间和自己 windows 的时间是不同的，无法通过 windows 的时间来推测 dos 的时钟程序调用功能，但是 dos 下面的 date 命令在我的电脑上也没有办法使用，导致探究中出了一些问题，只好拿来前人的结论来用。

实验二是优化自己程序的题目，我也是第一次做这种题目。从拿到题目无从入手，到最后优化到百分之 44，其中还是付出了很大的努力。最开始的时候自己只会把乘以二，除以二这样的操作改成移位运算加快时间，可是这样子做最低要求都满足不了。然后发现除以 7 是一个可以大做文章的地方，一开始想着除法就是减法，算出有几个 7 来就可以避免除法了。可是这样做最多做 100 次减法，而这样子做显然比除法本身还要慢。但是当被除数和除数相差不大的时候这种优化还是可取的。然后在讨论的群里面看到了“表”这个关键字，立马觉得这个方法很实用，建立一张表直接去查，速度加快了很多很多。然后是对 32 位寄存器的使用，访问内存肯定比访问寄存器要慢很多，由于这个原则，我们加大寄存器的位数就可以减少访问了内存从而加速。根据这个原理加快了查询和访问成绩。这个优化基本就是这次实验思考最多的地方，收获的确很多。

实验三是一个观察 C 语言和汇编语言对应的题目。我对于 C 语言是如何执行的已经困惑很久，对于他里面的变量到底是在栈上还是在堆上也迷惑很长时间，借这次实验我又从汇编语言的角度深刻理解了这一点。深深感受到了开发编译器的困难。最后观察 BC 中对代码的优化选项，可以发现，开发者对代码的优化遵循的原则和我们二小题遵循的原则实际差别不是很大，只是要做的工作更加多，设计更加合理。我也因此感觉到了以前觉得神秘的东西只是因为你的知识还没有积累到那个程度，当你的知识储备够多的时候，很多神秘的东西也就不再神秘。

最后是对预习和总结的体会，预习自己做的还是不太好，分配任务步骤自己很难做到合理细致，别人看到的时候总会觉得摸不到头脑，我一定会积极改正！

汇 编 语 言 程 序 设 计 实 验 报 告

参考文献

- [1] 王爽. 汇编语言. 第三版. 清华大学出版社, 2003 01- 310
 - [2] 曹忠升 80X86 汇编语言程序设计 华中科技大学出版社
 - [3] Randal E. Bryant & David R. O' Hallaron CSAPP 机械工业出版社
-