

## 汇编语言

软件安装

1.DOSBox

2.Vim

## 编译和链接

将源代码 生成最终 的 exe 文件 然后执行

## 进制

10进制

2进制

进制快速转换

字节转换

小结

## 寄存器

通用寄存器

(地址寄存器) 指令寄存器 CS (段地址) 和IP (偏移地址)

指令的执行过程

## debug

### 寄存器 (内存访问)

3个段

数据段

1. 字的存储

3. mov, add, sub 指令

4. -d 段地址: 偏移地址

5. 在内存中存放自己定义的数据, 通过 **ds:[ ]** 来 让CPU访问数据

代码段

1. 段地址存放在cs寄存器中

2. 偏移地址存放在ip寄存器当中

3. 内存中存放代码

4. 修改cs:ip中的值就可使CPU执行代码

栈段

1. 栈的作用

2. 栈的寄存器ss:sp

3. 操作指令push&ip

4. 处理数据时要, 临时存放数据

5. 修改ss:sp中的值, 决定栈顶位置, CPU在执行的过程中把我们定义的栈段当作栈使用

6. 一段连续的内存地址

7. 栈的容量的最大极限

8.每执行 一条 -t 指令 就会将寄存器的值保存到 栈中

内存的安全访问

## 承上启下

### 第一个程序

编译和链接

程序的跟踪 debug + 程序名

快速编译

偏移地址寄存器 **bx**

### 自己分配内存

自己分配内存

总结

### 定位内存地址的方法

and 和 or

以字符的形式给出数据

[bx+idata]

SI 和 DI

[BX+SI] 和 [BX+DI]

[BX+SI+idata] 和 [bx+di+idata]

## 数据处理的两个基本问题

bx,si,di和bp

指令要处理的数据

数据位置的表达

数据的长度

div 指令

伪指令 dd (占两个字)

dup

## 转移指令

操作符 offset

jmp 指令

jmp short 标号

jmp near ptr 标号

jmp far ptr 标号

在内存中转移

jmp word ptr 标号

jmp dword ptr 标号

jcxz (短转移)

loop (短转移)

## ret 和 call

ret (用栈中的数据)

call(不能实现短转移)

- 1.根据位移进行转移
- 2.转移目的地址在指令中
- 3.转移地址在寄存器中
4. 转移地址在内存中

1

2

call 和 ret 共同应用

批量数据处理

寄存器冲突问题

mul

1. 8位

2. 16位

模块化程序设计

参数和结果的传递

## 标志寄存器

# 汇编语言

- 日期 19.07.19

## 软件安装

### 1.DOSBox

- 1. 无脑下一步
- 2. 修改配置文件  
添加以下命令

```
1 | mount c: d:\asm
2 |   c:
3 |   //d盘下的文件是自行创建其中包含debug.exe就可以了
```

## 2.Vim

- 1. 安装完后打开其文件位置
- 2. 修改配置文件  
在开头写入简单的配置文件

```
1 | set number
2 | color evening
3 | //保存退出
```

## 编译和链接

### 将源代码 生成最终 的 exe 文件 然后执行

- 这一部分刚开始跟上做就好了，不用了解清楚

DOSBox代码

```
1 | masm t1;
2 | link t1;
3 | // t1 为自己创建的asm文件
4 | //在创建txt文件把后缀改为asm
5 | //用vim编辑
```

asm文件代码

```
1 | assume cs:code
```

code segment

```
    mov bx,0B800H
mov es,bx
```

```
mov bx,16010 + 402
```

```
    mov word ptr es:[bx],5535H
```

```
    mov ax,4C00H
int 21H
```

code ends

end

1 |

# 进制

## 10进制

437->>>

$$4 * 100 + 3 * 10 + 7 * 1$$

$$4 * 10^2 + 3 * 10^1 + 7 * 10^0$$

0.1.2.3就是他在数字中的位置

## 2进制

111->>

$$1 * 2^2 + 1 * 2^1 + 1 * 2^0 \gg \text{转换成10进制}$$

数字	1	0
含义	有	无

位置	2	1	0	
有代表	4	2	1	

## 进制快速转换

拆分

## 字节转换

$$1\text{byte} = 8\text{bit}$$

$$1\text{KB} = 1024\text{byte} \gg 1\text{KB} = 2^{10}\text{byte}$$

$$1\text{MB} = 1024 * 1\text{KB} \quad 1\text{MB} = 1024 * 1024 \text{byte} \gg 1\text{MB} = 2^{20}\text{byte}$$

$$1\text{GB} = 1024 * 1\text{MB} \gg 1\text{GB} = 2^{30}\text{byte}$$

## 小结

1. 汇编指令是 机器指令的 注记符，同机器指令——对应
  2. 每一种CPU都有自己的汇编指令集
  3. CPU 可以直接使用的 信息 在存储器中 存放
    - 4.在存储器中指令和数据没用任何区别，都是二进制数
    - 5.存储器单元从 0 开始 编号
    - 6.一个 存储单元 可以存储 8 个 bit 即 8 为二进制数
    - 7.1byte=8bit 1KB=1024byte 1MB=1024KB 1GB=1024MB
- 总线

地址总线 决定 CPU 的 寻址能力  
数据总线 决定 CPU 的一次传输数据量  
控制总线 决定 CPU 的对其他器件的控制能力

## 寄存器

### 1. 小例子

- 1.1 B800: 0400 回车
- 1.2 1空格 1空格
- 1.3 2空格 2空格
- 1.4 ...

2. 汇编程序员 就是 通过 汇编语言 中的 汇编指令 去修改 寄存器的值 从而 控制 CPU 控制整个计算机

## 通用寄存器

AX,BX,CX,DX

1.

他们各自可分为两个 8 位寄存器(only)

$ax = ah + al$  ( $h == high, l == low$ )

2.

1 byte = 8 bit(8位寄存器)字节型数据

2 byte = 16 bit(16位寄存器)字型数据 2个字节

一个字型数据==2个字节型数据=高位字节+低位字节

3.

1 数据与寄存器之间 要保持一致性, 8位寄存器给8位数据, 16为寄存器给16位数据

2 不区分大小写

## (地址寄存器) 指令寄存器 CS (段地址) 和IP (偏移地址)

jmp指令 jmp 2000:0 ==> cs2000,ip===0;

mov ax,1000

jmp ax

==> ip=1000;

只能用jmp指令修改cs,ip

- 1.CPU从cs:ip 所指的内存单元中读取内容, 存取到 指令缓存器当中
- 2.然后IP跳转到下一个指令位置, 并且在执行指令缓存器当中的指令
- 3.重复1。

段地址寄存器	偏移地址寄存器
ds (内存) ,es,ss (栈) ,cs	sp (栈) ,bp,si,di,ip,bx

## 指令的执行过程

1. CPU从cs:ip所指向的内存单元 读取 指令 然后 存放到 指令缓存器当中
2.  $IP = IP + \text{所读指令的长度}$ ，从而指向下一条指令
3. 执行指令缓存器的内容，回到步骤1

## debug

---

**-r 查看和修改寄存器中的内容**

```
-r cs  
cs value  
enter
```

**-d 查看内存中的内容** 段地址加偏移地址

```
-d ss:00
```

**-v 将机器指令翻译成汇编指令**

**-a 以汇编指令的格式 在内存中写入一条汇编指令** 每次debug都的写

**-t 执行当前 cs:ip 所指的机器指令** 代码段

**-e 可以改写 内存中的内容（数据）**

**-p 快速执行完loop 指令**

**-g 地址 ===== 一直执行到 地址 的 位置**

## 寄存器（内存访问）

---

3个段

---

### 数据段

---

#### 1. 字的存储

一次存放两个字节

2.

内存地址由 **段地址** 和 **偏移地址** 构成  
其中段地址默认保存在DS寄存器当中  
偏移地址由 **[address]** 保存告知

#### 3. mov, add, sub 指令

#### 4. -d 段地址：偏移地址

5. 在内存中存放自己定义的数据，通过 **ds 和[]** 来 让CPU访问数据

### 代码段

---

#### 1. 段地址存放在cs寄存器中

## 2. 偏移地址存放在ip寄存器当中

## 3. 内存中存放代码

## 4. 修改cs:ip中的值就可使CPU执行代码

# 栈段

---

## 1. 栈的作用

1. 临时性保存数据
2. 进行数据交换

```
1  -a
2  mov ax,1000
3  mov bx,2000
4  push ax
5  push bx
6  pop ax
7  pop bx
```

## 2. 栈的寄存器ss:sp

## 3. 操作指令push&ip

push 执行过程

- 1.sp=sp-2 (栈顶标记)
- 2.传入字型数据

pop 执行过程

- 1.传出字或字节
- 2.sp=sp+2(栈顶标记)

栈顶标记 在 数据 (内存地址) 的上面 的 内存地址  
sp 偏移地址寄存器 ss 段地址寄存器

## 4. 处理数据时要 , 临时存放数据

## 5. 修改ss:sp中的值, 决定栈顶位置, CPU在执行的过程中把我们定义的栈段当作栈使用

## 6. 一段连续的内存地址

## 7. 栈的容量的最大极限

sp 的变化范围 0~ffffH 32768 个字型数据  
call 将指令IP 保存到内存的哪里? ret 可以拿回  
保存到栈中 为了让 ret 从栈中取回

## 8.每执行 一条 -t 指令 就会将寄存器的值保存到 栈中

# 内存的安全访问

1. 安全空间 0: 200~0: 2ffH
2. 内存分配的时间 1. 系统加载程序的时候 为程序分配的内存。2. 程序执行过程中，向系统再要去内存空间

## 承上启下

- 我们可以把内存任意的划分为 栈，数据，指令，他们可以是同一块内存，亦可以是不同的内存
- cpu 通过 ss:sp 所指向的 内存作为 栈
- ds:[] 所指向的 内存 作为数据
- cs:ip 所指向的 内存 作为指令

指令从哪里？数据从哪来？临时性的数据存放到哪里？

## 第一个程序

### 编译和链接

1. 编译 `masm .mas --> .obj`
2. 链接 `link .obj --> .exe`

exe 文件的描述信息中 保存的程序入口 地址  
然后 系统 通过 描述文件 来设置 cs:ip 和 其它内存

asm 文件 -- 汇编语言 (1.汇编指令2.伪指令3.符号体系)

1. 汇编指令 由编译器 翻译成010101 的机器指令 最后由 **CPU** 执行
2. 伪指令和符号体系 由编译器执行

- 程序返回功能  
把系统加载程序的时候 给程序分配的内存， 设置的寄存器 返还给系统，因为 系统资源是有限的

```
1 | mov ax,4c00
2 | int 21H
```

### 程序的跟踪 debug + 程序名

- p 执行 int 指令
- q 退出
- cx == 程序长度
- PSP区 从 **ds:0** 开始的 256 个字节

### 快速编译

- 字母型数字前面 必须加 0;

默认代码 (目前)



```

1  assum cs:code
2
3  code segment
4
5      ;填写内容
6
7      mov ax,4c00H
8      int 21H
9
10 code ends
11
12 end

```

## 偏移地址寄存器 *bx*

# 自己分配内存

## 自己分配内存

- 一个 segment 最少占据 16 个字节
- 假设 数据段 有  $N$  个字节 则 实际占用  $(N/16 + 1) * 16$  个
- 都是 16 的倍数

```

1  //实验5
2  assume cs:codesg
3
4  a segment
5      db 1,2,3,4,5,6,7,8
6  a ends
7
8  b segment
9      db 1,2,3,4,5,6,7,8
10 b ends
11
12 c segment
13     db 0,0,0,0,0,0,0,0
14 c ends
15
16 codesg segment
17
18 start:
19
20
21     mov ax,c
22     mov es,ax
23
24
25     sub cx,cx
26     sub bx,bx
27     add cx,8
28
29 addnum: mov ax,a
30         mov ds,ax

```

```

31
32         sub dx,dx
33
34         mov dl,ds:[bx] ;拿出第一个数据
35
36         mov ax,b
37         mov ds,ax
38
39         add dl,ds:[bx] ;拿出第二个数据,并且相加
40
41         mov es:[bx],dl
42
43         inc bx
44
45         loop addnum
46
47
48         mov ax,4c00h
49         int 21h
50
51 codesg ends
52
53 end start

```

## 总结

- db 字节型
- dw 字型

## 定位内存地址的方法

### and 和 or

#### 1. and 逻辑与

1. 全为 1 才出 1 否则全部为 0
2. 可用于对二进制位的数字设 0

#### 2. or 逻辑或

1. 只要有1 就为 1
2. 可用于对二进制数字设 1

## 以字符的形式给出数据

- like '.....' 其中单引号包含的内容 编译器将把 其中的内容 转化为相应的 ASCII

### [bx+idata]

- idata 是立即数

常用格式

1. mov ax,[200+bx]

2. `mov ax,200[bx]`
3. `mov ax,[bx].200`

- 可以处理数组

## SI 和 DI

- 类似于BX 但是 不能分成两个 8 为寄存器

## [BX+SI] 和 [BX+DI]

常用格式

```
mov ax,[bx] [si]
```

## [BX+SI+idata] 和 [bx+di+idata]

# 数据处理的两个基本问题

sreg 段地址寄存器

reg 寄存器

## bx,si,di和bp

- bx si/di组合
- bp si/di组合

## 指令要处理的数据

1. 保存在CPU
2. 在内存中
3. 在端口中

## 数据位置的表达

1. 立即数(idata)
2. 寄存器
3. 段地址加偏移地址

## 数据的长度

- byte 和 word

在处理数据的时候要告知 CPU 要处理的数据有多大可以通过一些方法来告知

1. 通过寄存器来指明 如 `ax`,代表对word操作而 `al`,代表对byte 操作
2. 无寄存器 则用 `X ptr` 来表示 X 为byte 或者 word 如： `mov word ptr ds:[0],1`
3. 用 `push` or `pop` 就不用声明 因为 栈就是对字进行操作

## div 指令

---

- 除数 有8位和16位 在一个reg或内存单元中
- 被除数 默认 放在 ax (16位) 或者 dx (高16位) 和 ax (低16位) 中
- 结果 al(商) ah(余数) 或者 ax(商) dx(余数)

## 伪指令 dd (占两个字)

---

- 相当于 两个 dw
- 四个 db

## dup

---

用来重复数据

- db 3 dup (0) ==> db 0 ,0, 0

## 转移指令

---

### 操作符 offset

---

- 取得标号的偏移地址

## jmp 指令

---

- 无条件转移指令
- 可同时修改 cs 和 ip 或者 ip

### jmp short 标号

---

- 在编译是就已经处理好 要偏移的地址
- 无论 本 命令在哪 只有 偏移地址

### jmp near ptr 标号

---

- 段内短转移

### jmp far ptr 标号

---

- 同时修改 cs 和 ip

## 在内存中转移

---

### jmp word ptr 标号

- jmp word ptr ds:[0]

- 只修改 IP
- 段内转移

## jmp dword ptr 标号

- 段间转移
- $ip[X+0], cs[X+2]$

## jcxz (短转移)

- jmp cx zero
- 只有在cx 为0 的情况下 才 执行 转移

## loop (短转移)

- cx 不为0 执行loop

# ret 和 call

- 指令执行过程

指令执行过程

- 1. CPU 从 CS:IP 所组合出来的地址 读取指令 读到 指令缓冲器中
- 2.  $IP = IP + \text{所读指令的字节数}$  ; IP指向了下一条指令
- 3. 执行 指令缓冲器中的内容, 回到第一步

通过栈中的数据来修改 cs 和 ip 同时还会 修改栈顶标志

## ret (用栈中的数据)

- 弹栈

1. 近转移 ret 修改 IP **pop ip**

1.  $(ip) = ((ss) * 16 + (sp))$
2.  $(sp) = (sp) + 2$

2. 远转移 retf 修改 cs:ip **pop ip, pop cs**

1.  $(ip) = ((ss) * 16 + (sp))$
2.  $(sp) = (sp) + 2$
3.  $(cs) = ((ss) * 16 + (sp))$
4.  $(sp) = (sp) + 2$

## call(不能实现短转移)

- 类似jmp
- call程序处理的数据一般要进行压栈

## 1.根据位移进行转移

```
1 | push ip
2 | jmp near ptr 标号
```

- 执行过程 原理
  - call下一条指令的IP压栈后，转到标号处

## 2.转移目的地址在指令中

---

```
1 | call far ptr
```

- 执行过程 原理
  - call下一条指令的CS:IP压栈后，转到标号处

## 3.转移地址在寄存器中

---

```
1 | call 16 位 reg
```

- 执行过程 原理
  - call下一条指令的IP压栈后，转到reg处

## 4. 转移地址在内存中

---

1

---

```
1 | call word ptr 内存单元地址
```

- 执行过程 原理
  - call下一条指令的IP压栈后，转到内存单元地址

2

---

```
1 | call dword ptr 内存单元地址
```

- 执行过程 原理
  - call下一条指令的CS:IP压栈后，转到标号处

## call 和 ret 共同应用

---

- 就像函数调用

## 批量数据处理

---

```
1 | assume cs:code,ds:data,ss:stack
```

```

2
3 data segment
4     db 'conversation'
5 data ends
6
7 stack segment
8     db 16 dup(0)
9 stack ends
10
11 code segment
12
13
14     start: mov ax,data
15             mov ds,ax
16             mov si,0
17             mov cx,12
18             call capital
19             mov ax,4c00h
20             int 21h
21
22     capital: and byte ptr ds:[si],11011111b
23             inc si;
24             loop capital
25             ret
26
27
28 code ends
29
30
31
32 end start

```

## 寄存器冲突问题

- 在子程序执行开头，把所需要用到的寄存器压栈
- 在子程序完成后，从栈中弹出各个寄存器的值

```

1 assume cs:code,ds:data,ss:stack
2
3 data segment
4     db 'word',0
5     db 'unix',0
6     db 'wind',0
7     db 'good',0
8 data ends
9
10 stack segment
11     db 128 dup(0)
12 stack ends
13
14 code segment
15
16
17     start: mov ax,data
18             mov ds,ax

```

```

19
20         mov cx,4
21         mov bx,0
22
23     s:    mov di,bx
24           call capital
25           add bx,5
26           loop s
27
28           mov ax,4c00h
29           int 21h
30
31     capital: push cx;执行子程序前压栈
32              push si
33
34     change: mov cl,ds:[si]
35              mov ch,0
36              jcxz ok
37              and byte ptr ds:[si],11011111b
38              inc si
39              jmp change
40
41     ok:    pop si;执行完后弹栈
42           pop cx
43           ret
44
45
46     code ends
47
48
49
50     end start

```

## mul

### 1. 8位

一个默认放在AL，另一个放在内存字节单元或者8位reg。  
结果 默认 AX。

### 2. 16位

一个默认放在AX，另一个放在内存字单元或者16位reg。  
结果 默认 高位在DX，低位在AX。

## 模块化程序设计

- 通过ret,call.

## 参数和结果的传递



```

1  assume cs:code,ds:data,ss:stack
2
3  data segment
4      dw 1,2,3,4,5,6,7,8
5      dd 0,0,0,0,0,0,0,0
6      db 'word',0
7      db 'unix',0
8      db 'wind',0
9      db 'good',0
10 data ends
11
12 stack segment
13     db 128 dup(0)
14 stack ends
15
16 code segment
17
18
19     start:  mov ax,data
20             mov ds,ax
21             mov si,0
22             mov bp,0
23             call r_start
24
25
26             mov ax,4c00h
27             int 21h
28
29
30     r_start:  mov bx,ds:[si]
31               call cube
32               mov ds:[16+bp],ax
33               add si,2
34               add bp,4
35               loop r_start
36               ret
37
38     cube:    mov ax,bx
39               mul bx
40               mul bx
41               ret
42

```

## 标志寄存器

---

标志	真值为1	假值为0
OF	OV	NO
SF	NG	PL
ZF	ZR	NZ
PF	PE	PO
CF	CY	NC
DF	DN	UP

- CF标志位Carry Flag

- CF CY NC CY = Carry Yes (进位) NC = not Carry
- 进位
- 和运算相关的指令会影响标志位 like add, sub

- ZF标志位Zero Flag

- ZF ZR NZ ZR = Zero NZ = not ZERO
- 判断相等

- PF标志位parity Flag

- 一的个数是否位偶数 or 奇数
- PF PE PO PE = EVEN 偶数 O = ODD

- SF标志位Sign Flag

- 正负 1 0
- SF NG PL NG = negative 负数 PL = Positive
- 计算的结果看陈整数和负数

- OF标志位Overflow

- OF 标志位 溢出 overflow
- 128 ~ 127 :有符号
- 32768 ~ 32767