

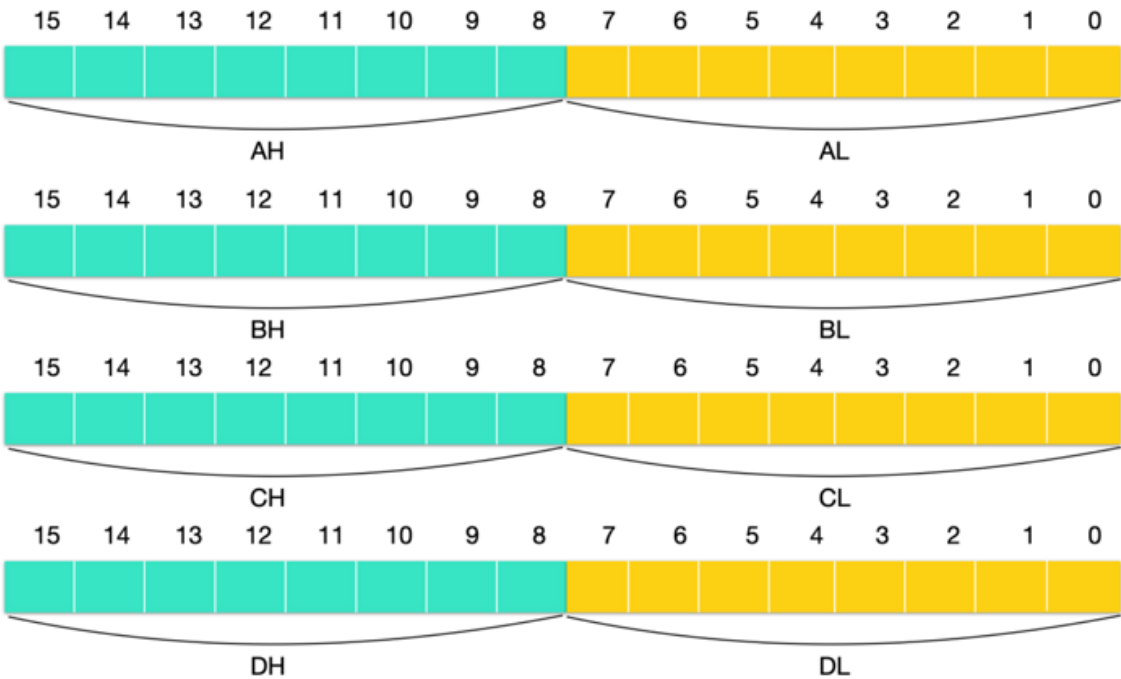
王爽汇编第二章, 寄存器

- 1. 寄存器知识
- 2. 通用寄存器
 - 2.1. AX寄存器(累加寄存器)
 - 2.2. BX寄存器(寻址寄存器)
 - 2.3. CX寄存器(计数寄存器)
 - 2.4. DX寄存器
- 3. 段寄存器
- 4. 偏移(索引)寄存器
 - 4.1. BP基础指针寄存器
 - 4.2. SP栈指针寄存器
 - 4.3. SI变址寄存器
 - 4.4. DI目标变址寄存器
- 5. IP寄存器
- 6. 标志寄存器

1. 寄存器知识

一个典型的CPU由运算器、控制器、寄存器（CPU工作原理）等器件构成，这些器件靠内部总线相连。

- 运算器进行信息处理(运算单元)
- 寄存器进行信息存储(存储单元)
- 控制器负责控制各种器件工作(控制单元)



几乎所有的冯·诺伊曼型计算机的 CPU，其工作都可以分为5个阶段：**取指令、指令译码、执行指令、访存取数、结果写回。**

1. 取指令：取出内存中OPCode字节码。
2. 指令译码：将OPCode字节码翻译成汇编指令。
3. 指令执行：执行汇编指令。
4. 访存取数：根据汇编指令需要,可能会访问内存数据,根据地址码得到内存位置并读取该操作数。
5. 结果写回：运行结果数据写回到 CPU 的内部寄存器，方便后续指令快速存取。

8086CPU中一共有14 个寄存器，所有寄存器都是16位宽，存2个字节，因为是完全的16位微处理器。

AX, BX, CX, DX, SP, BP, SI, DI, IP, FLAG, CS, DS, SS, ES

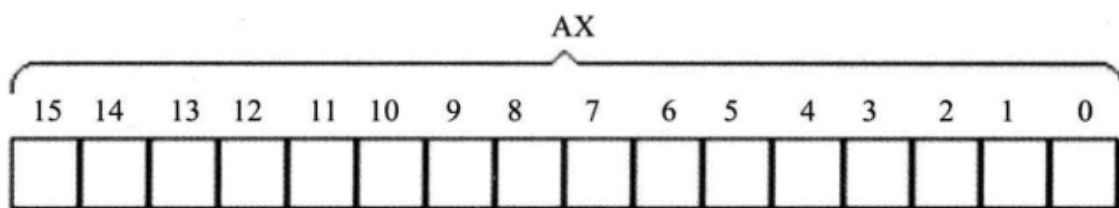


图 2.1 16 位寄存器的逻辑结构

这 14 个寄存器有可能进行具体的划分，按照功能可以分为五种：

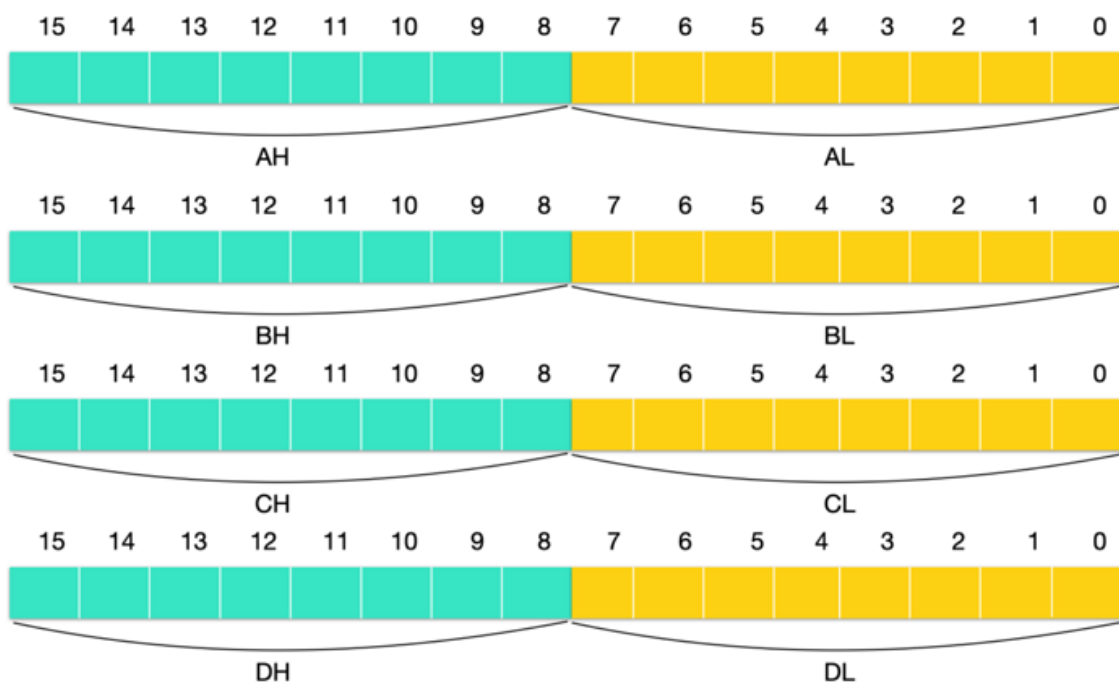
- 通用寄存器
- 段寄存器
- 偏移寄存器
- IP寄存器
- 标志寄存器

2. 通用寄存器

通用寄存器有4个：AX, BX, CX, DX，一般用来存放数据，也被称为数据寄存器。

它们可分为两个可独立使用的8位寄存器

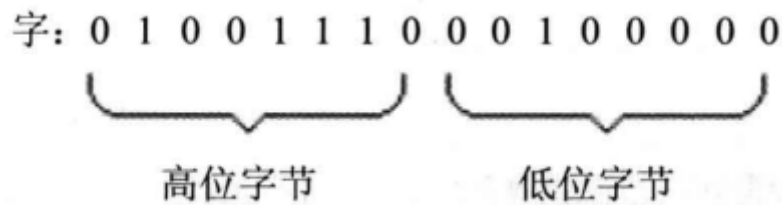
如下图所示。



8086CPU可以一次性处理以下两种尺寸的数据。

- 字节：记为byte，一个字节由8个bit组成，可以存在8位寄存器中。

- 字：记为word，一个字由两个字节组成，可以存在一个16位寄存器中(16位CPU)



8086采用小端模式：高地址存放高位字节，低地址存放低位字节。

一个8位寄存器所能存储的数据范围是：0~255 (2的8次方-1)

```
>>> hex(pow(2,8)-1)
'0xff'
```

8位寄存器

```
>>> hex(pow(2,32)-1)
'0xffffffff'
```

32位寄存器

```
>>> hex(pow(2,64)-1)
'0xffffffffffffffff'
```

64位寄存器

2.1. AX寄存器(累加寄存器)

AX也叫做累加寄存器，主要用于输入/输出大规模的指令运算。

以下例子是我在Windows下编译的32位汇编，其中AX扩展成了EAX，我们可以看见乘法(累加)后的结果他其实就是保存到了EAX中，然后再将EAX赋值给整型变量c，所以 乘法(imul) 和 除法(div) 运算都会用到AX寄存器，除法会将商保存到ax中，余数保存到dx中。

编译命令： `cl -FAS .\1.cpp`

```

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4
_main PROC
; 4 : {
    push    ebp
    mov     ebp, esp
    sub     esp, 12 ; 0000000cH

5   :    int a = 2;
    mov     DWORD PTR _a$[ebp], 2

; 6   :    int b = 3;
    mov     DWORD PTR _b$[ebp], 3

; 7   :    int c = a * b;
    mov     eax, DWORD PTR _a$[ebp]
    imul    eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _c$[ebp], eax

; 8   :    return 0;
    xor     eax, eax

; 9   : }

    mov     esp, ebp
    pop     ebp
    ret     0
_main ENDP
_TEXT ENDS
END

```

① 变量a

② 变量b

③ 2x3

eax作为累加结果 传给变量C

可以利用文章 <https://www.cnblogs.com/VxerLee/p/15264290.html> 中的方法将C语言转换成DOS汇编代码，然后对比源码和汇编代码发现乘法结果确实会放入到ax中。

乘法：

Assembly code (left):

```
23  _main  proc  far
24      push  bp
25      mov  bp,sp
26      sub  sp,2
27      push  si
28      push  di
29      ?debug  L 6
30      mov  si,2
31      ; ?debug  L 7
32      mov  di,3
33      ; ?debug  L 8
34      mov  ax,si
35      mul  di
36      mov  word ptr [bp-2],ax
37      ; ?debug  L 9
38      push  ds
39      mov  ax,offset DGROUP:s@
40      push  ax
41      call  far ptr _printf
```

C source code (right):

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a,b,c;
6     a = 2;
7     b = 3;
8     c = a * b;
9     printf("Hello,World!\r\n");
10    return 0;
11 }^Z
```

Red boxes and arrows indicate the mapping: Assembly line 29 points to C line 5, 32 to 6, 34-35 to 8, and 36 to 9.

除法(取余数):

Assembly code (left):

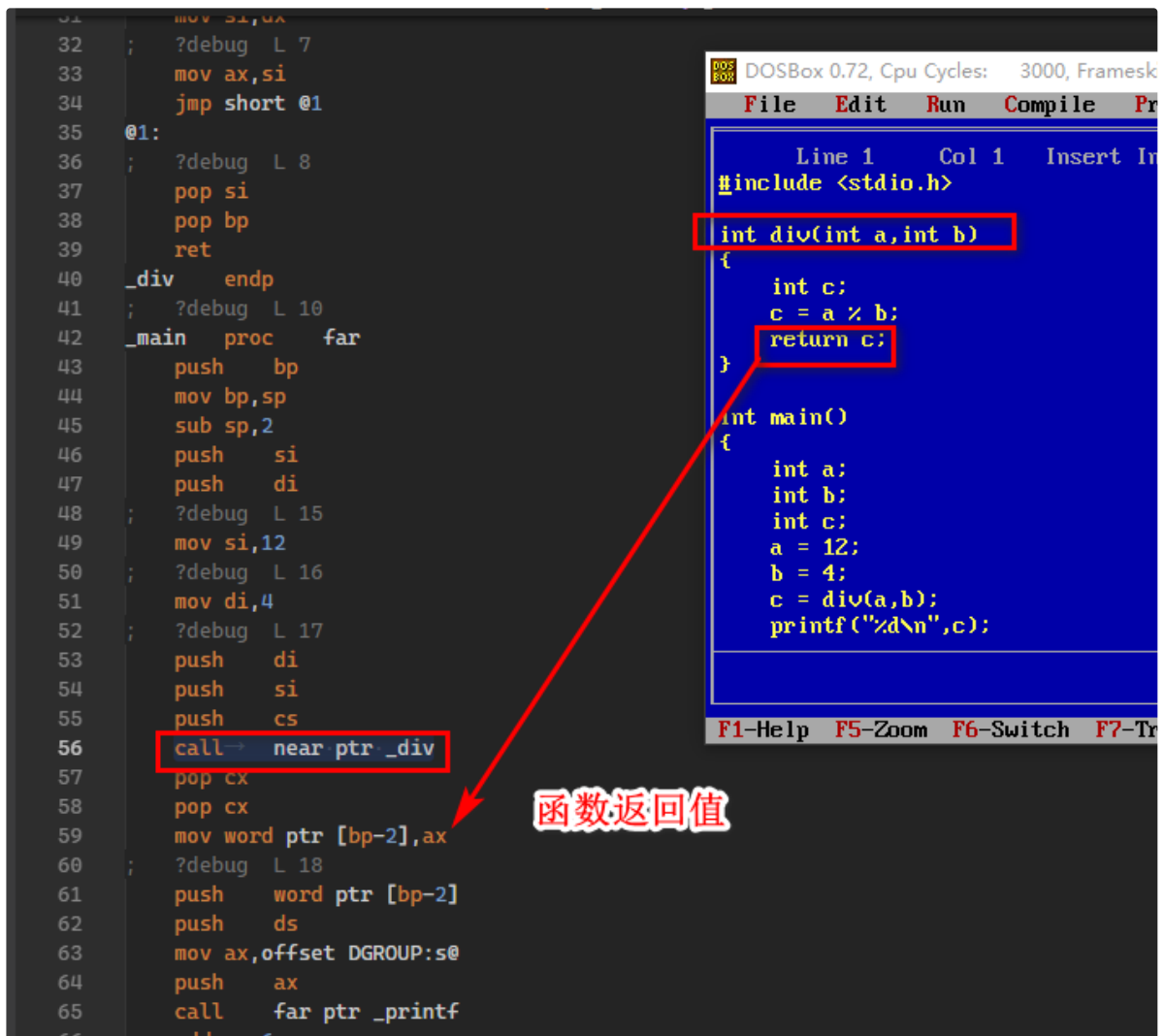
```
15  b@  label  byte
16  b@w  label  word
17      ?debug  C E9CE592E53056469762E63
18      ?debug  C E92A6C2D53112E2F696E636C75646
19      ?debug  C E900501D1110696E636C7564652F
20  _BSS  ends
21  DIV_TEXT  segment byte public 'CODE'
22      ; ?debug  L 2
23  _main  proc  far
24      push  bp
25      mov  bp,sp
26      sub  sp,2
27      push  si
28      push  di
29      ?debug  L 7
30      mov  si,12
31      ; ?debug  L 8
32      mov  di,4
33      ; ?debug  L 9
34      mov  ax,si
35      cwd
36      idiv  di
37      mov  word ptr [bp-2],dx
```

C source code (right):

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int b;
6     int c;
7     a = 12;
8     b = 4;
9     c = a % b;
10    printf("%d\n",c);
11 }
```

Red boxes and arrows indicate the mapping: Assembly line 34 points to C line 4, 36 to 9, and 37 to 10. A red box around line 37 is labeled "余数dx" (Remainder dx).

ax还常常被用作函数的返回值。



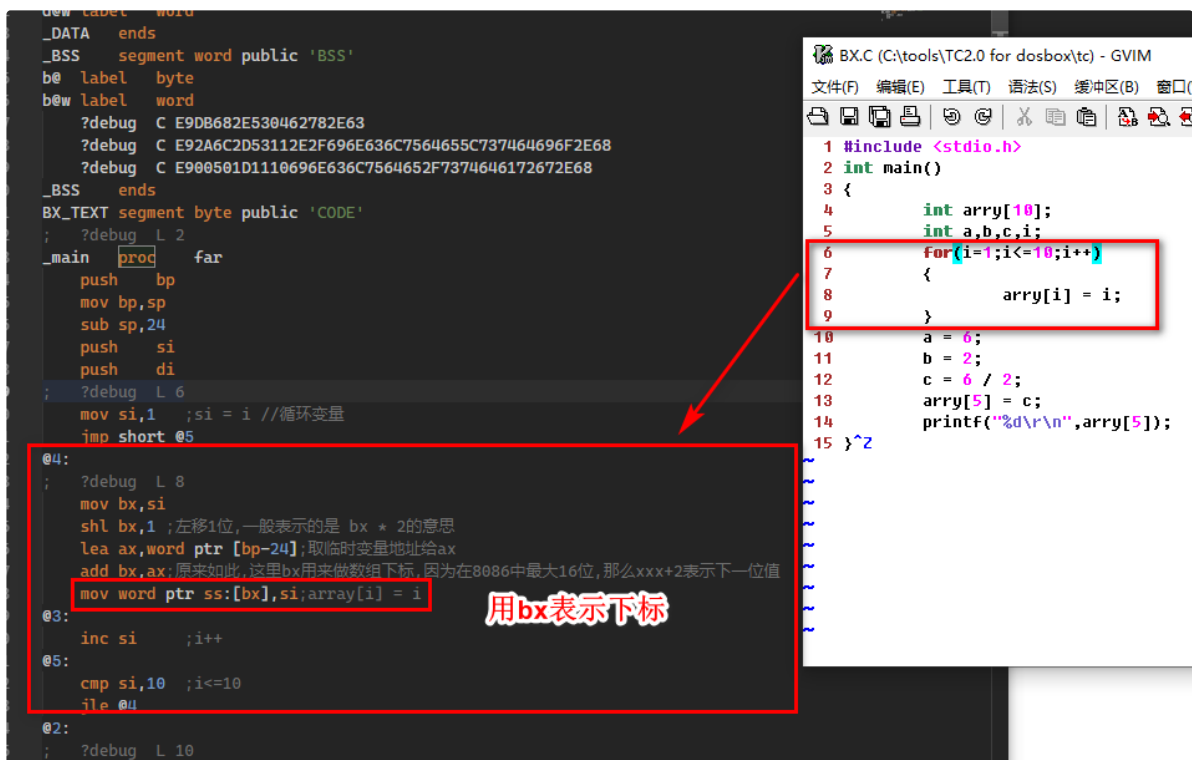
2.2. BX寄存器(寻址寄存器)

bx叫做数据寄存器，用来暂存一般数据，不过用的最多的还是用来寻址，bx存放偏移地址，然后根据基地址+偏移进行物理内存地址的定位。

暂存一般数据：

```
1 mov bx,2
2 add bx,bx
3 mov ax,bx
```

用bx寻址：



2.3. CX寄存器(计数寄存器)

cx也是叫做数据寄存器,也能暂存一般数据,不过cx还有个专门的用途,根据字面意思c -> "Count",用来在Loop(循环)时候,用cx寄存器来进行计数指定循环的次数。

```
1 mov cx,10 ;循环10次
2 xor ax,ax
3 xor bx,bx ;存放sum
4 s:
5     int ax ;
6     add bx,ax;/bx = 1+2+3+4+5+6+7+8+9+10
7     loop s
```

除此之外在Win32汇编中,ecx经常会作为类指针进行传入。

```
1 #include <stdio.h>
2 #include <iostream>
3 using namespace std;
4
5 class Cat{
6     public:
7         int age;
8         char *name;
9
10    public:
11        void run();
12 };
13
14 void Cat::run()
15 {
16     cout << "I'm a Cat." << endl;
17     cout << "My Name is " << this->name << endl;
18     printf("age=%d\n",this->age);
19     printf("I'm Running....\n");
20 }
21 int main()
```

```

22 {
23     Cat maomi;
24     maomi.age = 2;
25     maomi.name = "xiaohui";
26     maomi.run();
27     return 0;
28 }

```

进行反汇编后可以发现代码调用 `maomi.run()` 的时候会把 `this` 指针 传入 `ecx` 寄存器。 `lea ecx,dword ptr ss:[x]`

The screenshot shows the x32dbg interface. The assembly window displays the following instructions:

```

8B55 FC mov edx,dword ptr ss:[ebp-0x4]
8B02 mov eax,dword ptr ds:[edx]
50 push eax
68 B8E04000 push class.40E008
E8 56330000 call class.404437
83C4 08 add esp,0x8
68 C0E04000 push class.40E0C0
E8 49330000 call class.404437
83C4 04 add esp,0x4
8BE5 mov esp,ebp
50 pop ebp
C3 ret
55 push ebp
8BEC mov ebp,esp
83EC 08 sub esp,0x8
C745 F8 02000000 mov dword ptr ss:[ebp-0x8],0x2
C745 FC D4E04000 mov dword ptr ss:[ebp-0x4],class.40E004
8D40 F8 lea ecx,dword ptr ss:[ebp-0x8]
E8 60FFFFF call class.40107E
33D0 xor eax,edx
8BE5 mov esp,ebp
50 pop ebp

```

The register window on the right shows the following values:

```

EAX 007C14C0
EBX 7EFDE000
ECX 0018FF40
EDX 007C2688
EBP 0018FF48
ESP 0018FFA0
ESI 00000000
EDI 00000000
EIP 0040110C

```

The memory dump at the bottom shows the following data:

```

地址 值 注释
0018FF40 00000002 ...
0018FF44 0040E004 0x0. "xiaohui"
0018FF48 0018FF88 ...
0018FF4C 0040496F 0x0. 返回到 class.0040496F 自 class.004010F5
0018FF50 00000001 ...
0018FF54 007C1478 x.1. &"C:\Users\Administrator\Desktop\class.exe"

```

A red arrow points from the assembly instruction `lea ecx,dword ptr ss:[ebp-0x8]` to the register window where `ECX` is highlighted with the value `0018FF40`. Below the register window, the memory dump shows the value `"xiaohui"` at address `0018FF40`, with `age` and `name` labels pointing to it.


```

354 this$ = -4
355 ?run@Cat@@QAEXXZ PROC NEAR ; Cat::run
356
357 ; 15 : {
358
359     push    ebp
360     mov     ebp, esp
361     push    ecx
362     mov     DWORD PTR _this$[ebp], ecx
363
364 ; 16 :     cout << "I'm a Cat." << endl;
365
366     push    OFFSET FLAT:?endl@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@1@AAV21@@Z ; std::endl
367     push    OFFSET FLAT:$SG7428
368     push    OFFSET FLAT:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
369     call    ??6std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<
370     add     esp, 8
371     mov     ecx, eax
372     call    ??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAAV01@AAV01@@Z@Z ; std::basic_ostrea
373
374 ; 17 :     cout << "My Name is " << this->name << endl;
375
376     push    OFFSET FLAT:?endl@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@1@AAV21@@Z ; std::endl
377     mov     eax, DWORD PTR _this$[ebp]
378     mov     ecx, DWORD PTR [eax+4]
379     push    ecx
380     push    OFFSET FLAT:$SG7521
381     push    OFFSET FLAT:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
382     call    ??6std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<
383     add     esp, 8
384     push    eax
385     call    ??6std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<
386     add     esp, 8
387     mov     ecx, eax
388     call    ??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAAV01@AAV01@@Z@Z ; std::basic_ostrea
389
390 ; 18 :     printf("age=%d\n",this->age);
391
392     mov     edx, DWORD PTR _this$[ebp]
393     mov     eax, DWORD PTR [edx]
394     push    eax
395     push    OFFSET FLAT:$SG7522
396     call    _printf
397     add     esp, 8
398
399 ; 19 :     printf("I'm Running.....\n");
400
401     push    OFFSET FLAT:$SG7523
402     call    _printf
403     add     esp, 4
404
405 ; 20 : }

```

ecx 给 this 指针

this+4 = this->name

this+0 = this->age

2.4. DX寄存器

dx寄存器和之前的寄存器一样，都是能暂存一般数据，在之前的汇编代码中可以发现编译器会将(通用寄存器)和si、di都拿来暂存一般的数据，除此之外之前还介绍过乘法和除法的值也会写入dx中。

dx用作乘法的时候，用来存储ax寄存器不够存储的高位数据

```

1  assume cs:codesg,ds:datasg
2  ;>>>>>>数据段>>>>>>
3  datasg segment
4
5  datasg ends
6
7  ;>>>>>>代码段>>>>>>
8  codesg segment
9  start:
10     mov ax,datasg
11     mov ds,ax
12
13     ;乘法指令 0x190 * 0x190 = 0x27100 超过0xFFFF
14     mov ax,190h
15     mov bx,190h
16     mul bx ;0x190 * 0x190
17     ;此时来看看ax 和 dx的值
18
19     mov ax,4c00h
20     int 21h
21
22 codesg ends
23 end start

```

乘法前

DX=0000

DX=7100

DX=2000

dx存乘法结果的高位数据

dx用作除法的时候前面介绍过了，用来存储余数。

3. 段寄存器

CPU中包含了四个段寄存器，用作程序指令，数据或栈的基础位置。ps:(不过在Windows中好像这些段寄存器没什么用，因为Windows用了平坦模式，每个段的地址都一样都是0，直接用偏移来定位)

段寄存器主要的功能如下：

- **CS(Code Segment)** : 代码寄存器，程序代码的基础位置
- **DS(Data Segment)** : 数据寄存器，变量的基本位置
- **SS(Stack Segment)** : 栈寄存器，栈的基础位置
- **ES(Extra Segment)** : 其他寄存器，内存中变量的其他基本位置。

```
1 mov cs:[xxx],ax
2 mov ds:[xxx],ax
3 mov ss:[xxx],ax
4 mov es:[xxx],ax
```

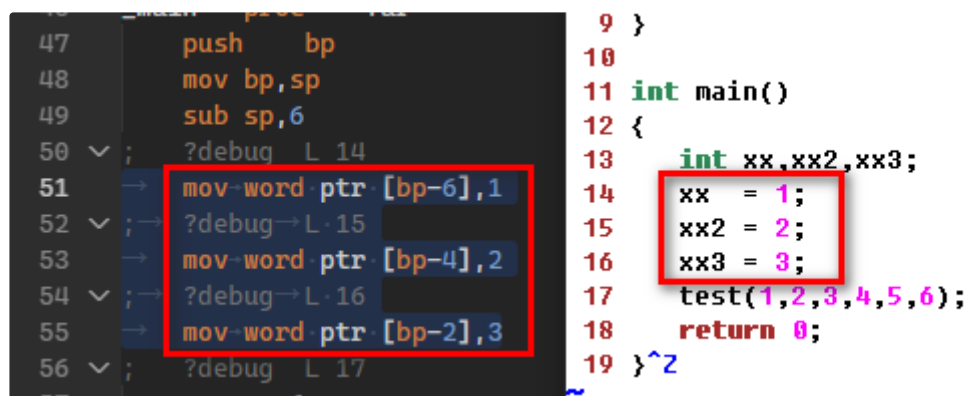
4. 偏移(索引)寄存器

偏移寄存器或者叫索引寄存器，主要是包含段地址的偏移量，用来进行内存地址的定位。

- **BP(Base Pointer)** : 基础指针，它是栈寄存器上的偏移量，用来定位栈上变量
- **SP(Stack Pointer)** : 栈指针，它是栈寄存器上的偏移量，用来定位栈顶
- **SI(Source Index)** : 变址寄存器，用来拷贝源字符串
- **DI(Destination Index)** : 目标变址寄存器，用来复制到目标字符串

4.1. BP基础指针寄存器

用 **[bp-xx]** 表示变量



```
47 push bp
48 mov bp,sp
49 sub sp,6
50 ; ?debug L 14
51 mov word ptr [bp-6],1
52 ; ?debug→L 15
53 mov word ptr [bp-4],2
54 ; ?debug→L 16
55 mov word ptr [bp-2],3
56 ; ?debug L 17
57 mov ax,6

9 }
10
11 int main()
12 {
13     int xx,xx2,xx3;
14     xx = 1;
15     xx2 = 2;
16     xx3 = 3;
17     test(1,2,3,4,5,6);
18     return 0;
19 }^2
```

用 **[bp+xx]** 表示函数参数

```

7      mov ax,6
8      push ax
9      mov ax,5
9      push ax
1     mov ax,4
2     push ax
3     mov ax,3
4     push ax
5     mov ax,2
5     push ax
7     mov ax,1
8     push ax
9     push cs
9     call near ptr _test

```

push 6,5,4,3,2,1,cs

堆栈数据中 `bp+6` 开始就是 `test` 函数的第一个参数直到 `bp+16` 为止。

```

21 BP_TEXT segment byte public 'CODE'
22 ; ?debug L 2
23 _test proc far
24     push bp
25     mov bp,sp
26     push si
27     ?debug L 5
28     mov si,word ptr [bp+6]
29     add si,word ptr [bp+8]
30     ?debug L 6
31     mov si,word ptr [bp+8]
32     add si,word ptr [bp+10]
33     ?debug L 7
34     mov si,word ptr [bp+10]
35     add si,word ptr [bp+12]
36     ?debug L 8
37     mov si,word ptr [bp+14]
38     add si,word ptr [bp+16]
39 @1:
40     ?debug L 9
41     pop si
42     pop bp
43     ret
44 _test endp
45 ; ?debug L 11

```

```

1 #include <stdio.h>
2 void test(int a,int b,int c,int d,int e,int f)
3 {
4     int x;
5     x = a + b;
6     x = b + c;
7     x = c + d;
8     x = e + f;
9 }
10
11 int main()
12 {
13     int xx,xx2,xx3;
14     xx = 1;
15     xx2 = 2;
16     xx3 = 3;
17     test(1,2,3,4,5,6);
18     return 0;
19 }^Z

```

4.2. SP栈指针寄存器

这个其实主要是栈顶指针，无论何时sp指针都执行栈的顶部，[栈是一个从高地址向下生长的内存]。

push数据时候 sp指针情况(sp地址会减少)

此时 sp 地址 = 0x0018FEFC

接着我们执行完push看看

地址	汇编指令	操作数
004010F6	nop	
004010F7	nop	
004010F8	nop	
004010F9	nop	
004010FA	nop	
004010FB	nop	
004010FC	nop	
004010FD	nop	
004010FE	push	0x3
00401100	push	0x2
00401102	push	0x1
00401104	nop	
00401105	nop	
00401106	nop	
00401107	nop	
00401108	nop	

寄存器	值
EAX	00000001
EBX	00020000
ECX	00010000
EDX	0018FF40
EBP	0018FF38
ESP	0018FEFC
ESI	00000000
EDI	00000000

执行完push后的sp地址 = 0x0018FEF0

得出结论：当执行push操作后bp地址会减少

注意这地方是我们push前的地址

push的值

地址	汇编指令	操作数
004010F5	nop	
004010F6	nop	
004010F7	nop	
004010F8	nop	
004010F9	nop	
004010FA	nop	
004010FB	nop	
004010FC	nop	
004010FD	nop	
004010FE	push	0x3
00401100	push	0x2
00401102	push	0x1
00401104	nop	
00401105	nop	
00401106	nop	
00401107	nop	
00401108	nop	

寄存器	值
EAX	00000001
EBX	00020000
ECX	00010000
EDX	0018FF40
EBP	0018FF38
ESP	0018FEF0
ESI	00000000
EDI	00000000

pop数据时候 sp指针情况(sp地址会增加)

执行完pop后地址又变回去了 0x0018FEFC

得出结论：执行pop执行是，sp会增加

地址	汇编指令	操作数
004010F5	nop	
004010F6	nop	
004010F7	nop	
004010F8	nop	
004010F9	nop	
004010FA	nop	
004010FB	nop	
004010FC	nop	
004010FD	nop	
004010FE	push	0x3
00401100	push	0x2
00401102	push	0x1
00401104	pop	eax
00401105	pop	ebx
00401106	pop	ecx
00401107	nop	
00401108	nop	
00401109	lea	ecx, dword ptr ss:[ebp-0x8]
0040110C	call	class.40107E

寄存器	值
EAX	00000001
EBX	00000002
ECX	00000003
EDX	0018FF40
EBP	0018FF38
ESP	0018FEFC
ESI	00000000
EDI	00000000

4.3. SI变址寄存器

si寄存器是变址寄存器，可以用来存放寻址用的偏移，此外si还被用作隐含的源串地址，默认在DS段中。
[无论是si还是di这两个寄存器总感觉和字符串都有点关系,c++的string?]

```
1 char source[]="hello,world"; (假装=esi寄存器)
2 char dest[]={0};           (假装=edi寄存器)
3 ;-----
4 mov ecx,strlen(source) ;字符串长度
5 rep movsb ;重复循环 并且传输字符串
6 ;-----
7 [dest] = "hello,world"
```

地址	十六进制	ASCII
0040D01F	68 65 6C 6C 6F 00 00 00 00 00 00 00 00 00 00 00	hello.....
0040D02F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040D03F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040D04F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

4.4. DI目标变址寄存器

di寄存器是目标变址寄存器，和si一样可以用来存放寻址用的偏移，此外di还被用作隐含的目的串地址，默认在ES段中。

先来看看STOS指令的介绍：

```
1 字符串存储指令 STOS
2
3 格式: STOS OPRD
4
5 功能: 把AL(字节)或AX(字)中的数据存储到DI为目的串地址指针所寻址的存储器单元中去.指针DI将根据DF的值进行自动调整.
```

接下来我们来看看，在汇编语言中是如何实现高级语言中清空数组的操作，以下是一个初始化字符数组的高级代码。

```
1 char szMsg[500]={0};
```

翻译后的汇编代码。

```
edi.00401000
push ebp
mov ebp,esp
sub esp,0x1F4
push edi
mov byte ptr ss:[ebp-0x1F4],0x0
mov ecx,0x7C ; 7C:'|'
xor eax,eax ; eax:&"ALLUSERSPROFILE=C:\\ProgramData"
lea edi,dword ptr ss:[ebp-0x1F3]
rep stosd
stosw
stosb
xor eax,eax ; eax:&"ALLUSERSPROFILE=C:\\ProgramData"
pop edi
mov esp,ebp
pop ebp
ret
```

ecx循环0x7c*4次
清空eax

szMsg数组 给edi

rep.循环操作
stos 将eax内容复制到edi内存

上面的汇编代码执行完后，可以发现szMsg数组的内容全部被清空了。

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code is as follows:

```
sub esp,0x1F4
push edi
mov byte ptr ss:[ebp-0x1F4],0x0
mov ecx,0x7C ; 7C:'\
xor eax,eax ; eax:&"ALLUSERSPROFILE=C:\\ProgramData"
lea edi,dword ptr ss:[ebp-0x1F3]
rep stosd
stosw
stosb
xor eax,eax ; eax:&"ALLUSERSPROFILE=C:\\ProgramData"
pop edi
mov esp,ebp
pop ebp
ret
```

A red arrow points from the `lea edi,dword ptr ss:[ebp-0x1F3]` instruction to the `EDI` register in the right-hand pane, which shows the value `0040101E`. A label `szMsg数组` is placed near the arrow. The memory dump on the right shows the state of memory addresses from `0018FD50` to `0018FDA0`, with values ranging from `00000000` to `00000128`. The text `执行前` (Before Execution) is written in the center of the memory dump area.

The screenshot shows the same debugger window after execution. The assembly code is the same as in the previous screenshot. The `EDI` register now shows the value `00401020`, which is the address of the `szMsg` array. A label `edi地址+数组大小` is placed near the `EDI` register. The memory dump on the right shows the state of memory addresses from `0018FD50` to `0018FDA0`, with values ranging from `00000000` to `00000000`. The text `执行后` (After Execution) is written in the center of the memory dump area.

5. IP寄存器

IP(Instruction Pointer)：指令指针寄存器，它是从 Code Segment 代码寄存器处的偏移来存储执行的下一条指令。

6. 标志寄存器

就剩下两种寄存器还没聊了，这两种寄存器是指令指针寄存器和标志寄存器：

- **FLAG** : Flag 寄存器用于存储当前进程的状态, 这些状态有
 - 位置 (Direction): 用于数据块的传输方向, 是向上传输还是向下载传输
 - 中断标志位 (Interrupt) : 1 - 允许; 0 - 禁止
 - 陷入位 (Trap) : 确定每条指令执行完成后, CPU 是否应该停止。1 - 开启, 0 - 关闭
 - 进位 (Carry) : 设置最后一个无符号算术运算是否带有进位
 - 溢出 (Overflow) : 设置最后一个有符号运算是否溢出
 - 符号 (Sign) : 如果最后一次算术运算为负, 则设置 1 = 负, 0 = 正
 - 零位 (Zero) : 如果最后一次算术运算结果为零, 1 = 零
 - 辅助进位 (Aux Carry) : 用于第三位到第四位的进位
 - 奇偶校验 (Parity) : 用于奇偶校验

参考文献:

https://blog.csdn.net/qq_39654127/article/details/88698911 《王爽汇编笔记》

<https://segmentfault.com/a/1190000037478310> 《十一假期偷了八天寄存器的相关知识》

<https://docs.microsoft.com/zh-cn/cpp/build/reference/fa-fa-listing-file?view=msvc-160> 《/FA/Fa使用》

https://blog.csdn.net/bagboy_taobao_com/article/details/6203705 《了解寄存器:ESI EDI 变址寄存器》

<http://c.biancheng.net/view/3679.html> 《汇编语言字符串基本指令简介》

https://blog.csdn.net/weixin_43216249/article/details/110728729 《汇编语言学习笔记--串操作篇 (c++的string? ? ?) 》