

彙編語言程序設計讀書筆記（1） - 相關工具

linux下彙編語言採用的是AT&T語法，可使用GNU工具，包括彙編器gas，連接器ld，編譯器gcc，調試器gdb或kdbg，objdump的反彙編功能，簡檔器gprof。以簡單的例子分別對每個工具在彙編語言開發中的用法進行簡單說明。

這些工具都要在linux環境下使用，先建立linux的開發環境，可參考文章「[windows7 64位系統安裝VMware Centos 64位系統搭建開發環境](#)」。

假設有以下簡單的c程序test1.c。

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("hello, world!\n");
```

```
    exit(0);
```

```
}
```

1. gcc用法介紹

gcc用於編譯源文件，加上參數可以生成中間文件。

用gcc把test1.c編譯成執行文件test1：

```
[root@ken test1]# gcc -o test1 test1.c
test1.c: 在函数‘main’中:
test1.c:6: 警告: 隱式聲明與內建函数‘exit’不兼容
[root@ken test1]# |
```

用gcc把test1.c編譯成文件test1-g，帶調試參數-g：

```
[root@ken test1]# gcc -g -o test1-g test1.c
test1.c: 在函数‘main’中:
test1.c:6: 警告: 隱式聲明與內建函数‘exit’不兼容
```

用gcc把test1.c生成彙編語言文件test1-s.s，用參數-S：

```
[root@ken test1]# gcc -S -o test1-s.s test1.c
test1.c: 在函数‘main’中:
test1.c:6: 警告: 隱式聲明與內建函数‘exit’不兼容
```

用gcc生成目標代碼test1.o，用參數-c：

```
[root@ken test1]# gcc -c test1.c
test1.c: 在函数‘main’中:
test1.c:6: 警告: 隱式聲明與內建函数‘exit’不兼容
```

疑問：gcc帶調式的參數除了-g外，還有-gstabs，-gstabs+，-ggdb和調式有關的參數，man gcc得到以下的說明。從說明中，我還是不能瞭解這幾個參數的具體區別以及何時該用哪個，如果哪位朋友知道，請給於解釋。

-g Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.

-gstabs

Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output which is not understood by DBX or SDB. On System V Release 4 systems this option requires the GNU assembler.

-gstabs+

Produce debugging information in stabs format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

-ggdb

Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF 2, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

彙編語言程序設計這本書是使用-gstabs這個參數。為什麼不用其它參數？

最後，test1.c用這四個參數生成的test1的大小記錄如下，可見參數不一樣生成的文件大小是不一樣的。

-g和-ggdb生成的結果大小一樣。

```
-rwxr-xr-x 1 root root 7851 11月 4 17:22 test1-g
-rwxr-xr-x 1 root root 7851 11月 4 17:48 test1-ggdb
-rwxr-xr-x 1 root root 12051 11月 4 17:48 test1-gstabs
-rwxr-xr-x 1 root root 12289 11月 4 17:48 test1-gstabs+
```

2. objdump用於反彙編的用法

上面產生的目標文件test1.o和可執行文件test1，可使用objdump反彙編。

```
[root@ken test1]# objdump -d test1.o
test1.o:          file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:    55                      push    %rbp
   1:   48 89 e5                mov     %rsp,%rbp
   4:   bf 00 00 00 00         mov     $0x0,%edi
   9:   e8 00 00 00 00         callq   e <main+0xe>
   e:   bf 00 00 00 00         mov     $0x0,%edi
  13:   e8 00 00 00 00         callq   18 <main+0x18>
[root@ken test1]# |
```

```
[root@ken test1]# objdump -d test1
test1:          file format elf64-x86-64

Disassembly of section .init:

00000000004003c8 <_init>:
 4003c8:  48 83 ec 08              sub     $0x8,%rsp
 4003cc:  e8 7b 00 00 00          callq   40044c <call_gmon_start>
 4003d1:  e8 0a 01 00 00          callq   4004e0 <frame_dummy>
 4003d6:  e8 e5 01 00 00          callq   4005c0 <__do_global_ctors_aux>
 4003db:  48 83 c4 08              add     $0x8,%rsp
 4003df:  c3                      retq

Disassembly of section .plt:

00000000004003e0 <puts@plt-0x10>:
 4003e0:  ff 35 b2 04 20 00        pushq   0x2004b2(%rip)          # 600898 <_GLOBAL_OFFSET_TABLE_+0x8>
 4003e6:  ff 25 b4 04 20 00        jmpq    *0x2004b4(%rip)        # 6008a0 <_GLOBAL_OFFSET_TABLE_+0x10>
 4003ec:  0f 1f 40 00              nopl    0x0(%rax)

00000000004003f0 <puts@plt>:
 4003f0:  ff 25 b2 04 20 00        jmpq    *0x2004b2(%rip)        # 6008a8 <_GLOBAL_OFFSET_TABLE_+0x18>
 4003f6:  68 00 00 00 00          pushq   $0x0
 4003fb:  e9 e0 ff ff ff          jmpq    4003e0 <_init+0x18>

0000000000400400 <exit@plt>:
 400400:  ff 25 aa 04 20 00        jmpq    *0x2004aa(%rip)        # 6008b0 <_GLOBAL_OFFSET_TABLE_+0x20>
 400406:  68 01 00 00 00          pushq   $0x1
```

3. gas彙編用法

gas把彙編程序彙編成目標文件，命令是as，找到以下的彙編程序test1.s：

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
movl $1, %eax
```

```
movl $0, %ebx
```

```
int $0x80
```

如下彙編，生成test1.o文件：

```
[root@ken test1]# as -o test1.o test1.s
[root@ken test1]# ll test1.o
-rw-r--r-- 1 root root 696 12月 4 21:44 test1.o
[root@ken test1]#
```

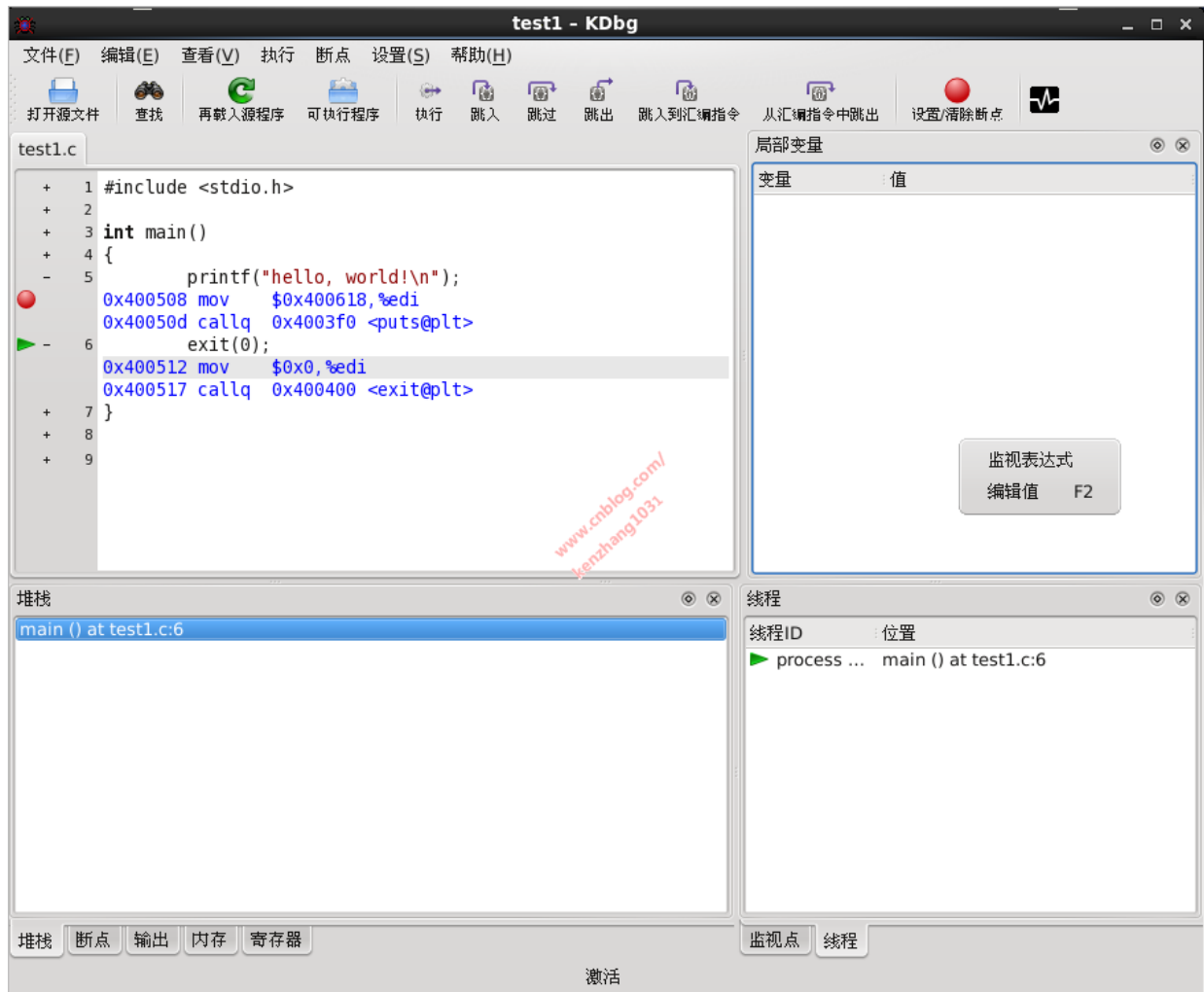
4. ld的用法

把前面的test1.o連接成可執行文件test1。

```
[root@ken test1]# ld -o test1 test1.o
[root@ken test1]# ll test1
-rwxr-xr-x 1 root root 664 12月 4 21:45 test1
[root@ken test1]#
```

5. gdb和kdbg用法

gdb用的是命令方式，用慣了像VS之類的IDE調試的，感覺gdb非常不好用，尤其是對於較複雜的工程，不如kdbg好用。



從界面看，比較方便操作，不用記複雜的命令，一目瞭然，可以把注意力放在軟件的查錯上。注意編譯或彙編時一定要帶調試的參數，比如gcc -gstabs或者as -gstabs，否則無法調式。kdbg需要kde和qt支持，而且在用ssh2登錄linux後的secureCRT中執行會提示無法連接連接到X server，需要在啟動圖形界面後的，輸入命令運行。
疑問：在secureCRT中怎麼執行？

6. 簡檔器gprof用法

簡檔器用於分析程序中所有函數的執行時間，編譯時要用-pg參數，但是查閱了幫助，as和ld都不支持-pg參數，因此只能使用gcc -pg。

用以下的test2.c為例：

```
/******
```

```
* test2.c
```

```
*****/
```

```

#include <stdio.h>

void func1(void)
{
    int i, j;

    for(i=0,j=0; i<1000000; i++)
        j++;
}

void func2(void)
{
    int i, j;

    func1();

    for(i=0,j=0; i<2000000; i++)
        j++;
}

int main()
{
    int i;

    for(i=0; i<100; i++)
        func1();

    for(i=0; i<200; i++)
        func2();

    return 0;
}

```

先要帶參數-pg編譯程序生成可執行文件test2，然後執行test2，執行一次就會生成一個新的gmon.out文件，執行gprof test2可執行文件就可以輸出簡檔，可以重定向到test2-gprof.txt，方便查看。如下：

```

[root@ken test2]# gcc -o test2 test2.c -pg
[root@ken test2]# ./test2
[root@ken test2]# ll gmon.out
-rw-r--r-- 1 root root 49200 1月 4 22:29 gmon.out
[root@ken test2]# gprof test2 > test2-gprof.txt
[root@ken test2]# |

```

最後得到的test2-gprof.txt文件摘錄如下：

Each sample counts as 0.01 seconds.

%	cumulative	self	self	total	
time	seconds	seconds	calls	ms/call	ms/call name
58.24	0.54	0.54	200	2.68	3.99 func2
42.86	0.93	0.39	300	1.31	1.31 func1

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 1.08% of 0.93 seconds

index	%	time	self	children	called	name
<spontaneous>						
[1]	100.0	0.00	0.93		main [1]	
		0.54	0.26	200/200		func2 [2]
		0.13	0.00	100/300		func1 [3]

		0.54	0.26	200/200		main [1]
[2]	85.9	0.54	0.26	200		func2 [2]
		0.26	0.00	200/300		func1 [3]

		0.13	0.00	100/300		main [1]
		0.26	0.00	200/300		func2 [2]
[3]	42.4	0.39	0.00	300		func1 [3]

可以分析到func1和func2執行的時間。

%time：表示函數佔總運行時間的百分比。

cumulative seconds：表示改行及其以上的行所用的時間。

self seconds：表示函數自己本身執行的時間。

calls：函數被調用的次數。

self ms/call：函數自己每次調用所用的執行時間。不包含函數中調用的別的函數執行的時間。

total ms/call：表示函數每次調用所用的執行時間，包含調用的別的函數的時間。

name：函數名。

疑問：彙編程序.s怎麼產生簡檔？

彙編語言程序設計讀書筆記（2） - 相關工具

64位系統篇

彙編語言程序設計一書，在32位系統下應該不會有什麼問題，然而在64位系統下，則會有些不一樣的地方。有些程序範例還會彙編錯誤或者執行錯誤。

博主所用系統為CentOS v6.4 x64。本文主要解決32位的彙編程序如何在64位環境下彙編、連接，而不論述64位彙編語言如何設計。

1. 64位系統下編譯32位的C程序

以程序test5.c為例，程序代碼很簡單，如下：

▢

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char str[4];
```

```
    str[0]='f';
```

```
    str[1]='g';
```

```
    str[2]='j';
```

```
    str[3] = 0;
```

```
    printf("CPU id is %s\n", str);
```

```
    exit(0);
```

```
}
```

這個C源程序沒有什麼32位還是64位之說，用gcc編譯後，在64位系統下就得到64位的elf文件，執行也不會有問題，如下圖所示：


```

[root@ken test1]# gcc -o test5 test5.c
[root@ken test1]# file test5
test5: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
[root@ken test1]# ./test5
CPU id is fgj
[root@ken test1]# |

```

上圖用file test5很清楚的看到是64-bit的文件。

那麼怎麼編譯成32位的程序呢？用-m32參數。會發現有錯誤，錯誤如下：

```

在包含目 /usr/include/features.h: 385 的文件中，
    从 /usr/include/stdio.h: 28，
    从 test5.c: 1:
/usr/include/gnu/stubs.h:7:27: 错误: gnu/stubs-32.h: 没有那个文件或目录
test5.c: 在函数‘main’中:
test5.c:11: 警告: 隐式声明与内建函数‘exit’不兼容

```

對於警告和exit不兼容，可以包含頭文件stdlib.h就可以解決。

對於gnu/stubs-32.h：沒有哪個文件或目錄，需要安裝glibc-devel和glibc-devel.i686。對於CentOS，安裝命令為：**yum install glibc-devel**和**yum install glibc-devel.i686**。

yum install glibc-devel，如下圖：

```

[root@ken test1]# yum install glibc-devel
Loaded plugins: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
 * base: mirrors.grandcloud.cn
 * extras: mirrors.grandcloud.cn
 * updates: mirrors.grandcloud.cn
http://mirrors.grandcloud.cn/centos/6.4/updates/x86_64/repodata/83126d8c763e8bb681c7f11cc25bab255b053d8cbd4228035ea146254bf7462-
R 7 - "couldn't connect to host"
Trying other mirror.
http://mirrors.stuhome.net/centos/6.4/updates/x86_64/repodata/83126d8c763e8bb681c7f11cc25bab255b053d8cbd4228035ea146254bf74624-
7 - "couldn't connect to host"
Trying other mirror.
ACACAHAAHAAHAA[3~^[[3~
  current download cancelled, interrupt (ctrl-c) again within two seconds
to exit.
ACACACACACACACACAChttp://centos.ustc.edu.cn/centos/6.4/updates/x86_64/repodata/83126d8c763e8bb681c7f11cc25bab255b053d8cbd4228
o 14] PYCURL ERROR 22 - "The requested URL returned error: 404 Not Found"
Trying other mirror.
updates/primary_db
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package glibc-devel.x86_64 0:2.12-1.107.el6 will be updated
--> Package glibc-devel.x86_64 0:2.12-1.107.el6_4.5 will be an update
--> Processing Dependency: glibc-headers = 2.12-1.107.el6_4.5 for package: glibc-devel-2.12-1.107.el6_4.5.x86_64
--> Processing Dependency: glibc = 2.12-1.107.el6_4.5 for package: glibc-devel-2.12-1.107.el6_4.5.x86_64
--> Running transaction check
--> Package glibc.i686 0:2.12-1.107.el6 will be updated
--> Processing Dependency: glibc = 2.12-1.107.el6 for package: glibc-common-2.12-1.107.el6.x86_64
--> Package glibc.x86_64 0:2.12-1.107.el6 will be updated
--> Package glibc.i686 0:2.12-1.107.el6_4.5 will be an update
--> Package glibc.x86_64 0:2.12-1.107.el6_4.5 will be an update
--> Package glibc-headers.x86_64 0:2.12-1.107.el6 will be updated
--> Package glibc-headers.x86_64 0:2.12-1.107.el6_4.5 will be an update
--> Running transaction check
--> Package glibc-common.x86_64 0:2.12-1.107.el6 will be updated
--> Package glibc-common.x86_64 0:2.12-1.107.el6_4.5 will be an update
updates/filelists_db
--> Finished Dependency Resolution

Dependencies Resolved

```

yum install glibc-devel.i686，如下圖：

```
[root@ken test1]# yum install glibc-devel.i686
Loaded plugins: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
 * base: mirrors.grandcloud.cn
 * extras: mirrors.grandcloud.cn
 * updates: mirrors.grandcloud.cn
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package glibc-devel.i686 0:2.12-1.107.el6_4.5 will be installed
--> Finished Dependency Resolution

Dependencies Resolved
```

之後使用-m32編譯，就不會再發生問題了。如下圖：

```
[root@ken test1]# gcc -m32 -o test5 test5.c
[root@ken test1]# file test5
test5: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
[root@ken test1]# ./test5
CPU id is fgj
[root@ken test1]# |
```

上圖用file test5很清楚的看到是32-bit的文件。

總結以上主要有三點：



64位系統下編譯32位程序使用-m32參數，即gcc -m32 -o output_file input_file.c



提示隱式聲明與內建函數『exit』不兼容的警告，增加#include <stdlib.h>來解決



gnu/stubs-32.h：沒有哪個文件或目錄的錯誤，需要安裝glibc-devel和glibc-devel.i686來解決

2. 64位系統下彙編32位的彙編程序

1). 64位彙編和32位彙編不同

彙編語言64位和32位是很不一樣的，這裡提供一份Intel官方的對64位的彙編簡單介紹的pdf文檔下載：[Introduction_to_x64_Assembly](#)。雖然該文檔按照微軟的MASM的格式來說明的，但是還是可以得到一些我們需要的信息，從該文檔中可以知道64位的寄存器已經和32位的不一樣了，比如64位寄存器是rax, rbx等，低32位的才是使用eax, ebx。

對於系統調用，64位系統和32位系統大大不一樣了，比如sys_write的系統調用，32位系統和64位系統分別如下：

32位的sys_write(stdout, str, length)的彙編調用

▢

```
# sys_write(stdout, str, length)的彙編調用
movl $4, %eax
movl $stdout, %ebx
movl $str, %ecx
movl $length, %edx
int $0x80
```

64位的sys_write(stdout, str, length)的彙編調用

▢

```
# 64位的sys_write(stdout, str, length)的彙編調用
movq $1, %rax
movq $stdout, %rdi
movq $str, %rsi
movq $length, %rdx
syscall
```

雖然出於兼容性，32位系統的調用仍然可以在64位系統上運行，但兩者已經大大不一樣了。本文不是論述如何寫64位彙編語言的，而是為瞭解決32位的彙編代碼可以在64位環境下運行。

2). 64位系統下用as和ld彙編32位的彙編程序

那麼怎樣在64位系統中彙編32位的彙編程序呢？以以下例子cpuid2.s為例，代碼為：



```
# cpuid2.s file
.section .data
output:
    .asciz "CPUID is '%s'\n"
.section .bss
    .lcomm buffer, 12
.section .text
.globl _start
_start:
    nop
    movl $0, %eax
    cpuid

    movl $buffer, %edi
    movl %ebx, (%edi)
    movl %edx, 4(%edi)
    movl %ecx, 8(%edi)

    pushl $buffer
    pushl $output
    call printf

    addl $8, %esp

    pushl $0
    call exit
```

這個代碼的具體實現，後續的章節會介紹。目前只知道是用於輸出CPU廠商ID字符串的就行了。

在64位系統下，如果按照書中所述的那樣as和ld，會產生錯誤，如下圖：

```
[root@ken test1]# as -o cpuid2.o cpuid2.s
cpuid2.s: Assembler messages:
cpuid2.s:19: Error: suffix or operands invalid for `push'
cpuid2.s:20: Error: suffix or operands invalid for `push'
cpuid2.s:25: Error: suffix or operands invalid for `push'
[root@ken test1]#
```

說是push操作無效。

如果不修改源代碼為64位的彙編，要解決這個問題，就需要命令64位系統按照32位的去彙編，as參數是--32，ld參數是-m elf_i386。如下圖：

```
[root@ken test1]# as --32 -o cpuid2.o cpuid2.s
[root@ken test1]# ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 -L/lib -lc cpuid2.o
[root@ken test1]# file cpuid2
cpuid2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#
```

file cpuid2可以看到確實是32位的文件，而且執行也沒有問題。這裡要特別對ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 -L/lib -lc cpuid2.o這行命令進行說明。

-m elf_i386：表示按照elf_i386的模塊進行連接，即32位的。

-lc：因為程序中調用了標準的C庫函數printf和exit，因此需要連接C動態庫libc.so，所以需要參數-lc來指定連接的庫文件，一般而言libxxx.so採用-lxxx的參數。

-L/lib：博主的系統有多個libc.so，包括64位的，32位的，arm結構的，如下圖所示，能搜出很多個libc.so。

```
[root@ken test1]# find / -name libc.so
/lib/libc.so
/usr/lib/libc.so
/usr/lib/x86_64-redhat-linux5E/lib64/libc.so
/usr/lib64/libc.so
/usr/local/arm/2.95.3/arm-linux/lib/libc.so
/opt/FriendlyARM/toolschain/4.5.1/arm-none-linux-gnueabi/sys-root/usr/lib/libc.so
```

而/lib/libc.so才是32位x86系統所需要的動態庫，所以使用-L/lib來指定庫文件的路徑，那麼-L/lib -lc就指定了連接的是/lib/libc.so。

-dynamic-linker /lib/ld-linux.so.2：用於運行時動態加載libc.so動態庫的。否則執行生成的可執行文件時會出錯。

3). 64位系統下用gcc彙編32位的彙編程序

還是上面的cpuid2.s為例，使用gcc -m32的參數進行彙編成32位的系統文件，特別的，這裡由於沒有main函數，而是用_start做入口點，因此需要使用參數 -nostdlib，如下圖：

```
[root@ken test1]# gcc -nostdlib -m32 -o cpuid2 cpuid2.s -L/lib -lc
[root@ken test1]# file cpuid2
cpuid2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV) dynamically linked (uses shared libs), not stripped
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#
```

對**gcc -nostdlib -m32 -o cpuid2 cpuid2.s -L/lib -lc**進行說明。

-m32：按照32位的編譯。

-nostdlib：如果沒有這個參數，gcc會連接gnu庫的函數，該函數會以_start為進入點，執行一段程序後，跳到main執行，而這個彙編源程序中用_start做進入點，而且沒有main，因此，沒有這個參數的話，會提示沒有main定義以及重複定義_start的錯誤。加了這個參數後，則不會去連接gnu函數。

綜上所述，64位系統下彙編32位彙編程序的做法是：

as --32 -o output_file.o input_file.s

ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o output_file -L/path -l libname input_file.s

或者

gcc -m32 -nostdlib -o output_file -L/path -l libname input_file.s

彙編語言程序設計讀書筆記（3） - 程序範例

主要描述三方面的內容：第一是彙編語言的程序模版，以及模版涉及到的一些知識點；第二是如何調試彙編語言；第三是如何在彙編語言中調用C庫函數。

1. 彙編語言的組成

彙編語言由段（section）組成，一個程序中執行的代碼，叫文本段（text），程序還可能有定義變量，有付給初始值的變量放在數據段（data）中，沒有賦初值或者付給零初值的放在bss段中。text段一定是要有的，data和bss可以沒有。

2. 段的定義

用.section語法定義段。比如：

.section .text定義文本段，
.section .data定義數據段，
.section .bss定義bss段。

順序沒有必須的要求，但為了便於別人接手和理解你的程序，書中建議採用從上到下按照data，bss，text段的順序定義。

3. 定義程序起始點

文本段必須要定義一個程序執行的起始點，ld默認為_start；而gcc默認會連接標準的庫代碼，該代碼入口為_start，執行一段程序後跳到main執行，因此gcc默認要求外部源程序定義main，且不能定義_start，但如果使用參數-nostdlib，那麼就不會默認連接標準的庫代碼，此時入口點用_start也沒問題；此外，ld和gcc都支持-e參數來指定入口點，此時任意的標號都可以用作入口點。

下面以例子來說明。以[彙編語言程序設計讀書筆記（2） - 相關工具64位系統篇](#)中的cpuid2.s為例子進行說明，目前可能讀者還不理解該程序的細節，但本文後面會論述，讀完本文後，理解該程序不是問題。目前只要清楚該程序用於輸出CPU的ID廠商的字符串。源程序入口為_start。如下：

▢

```
# cpuid2.s file
```

```
.section .data
```

```
output:
```

```
.asciz "CPUID is '%s'\n"
```

```
.section .bss
```

```
.lcomm buffer, 12
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    nop
```

```
    movl $0, %eax
```

```
    cpuid
```

```
    movl $buffer, %edi
```

```
    movl %ebx, (%edi)
```

```
    movl %edx, 4(%edi)
```

```
    movl %ecx, 8(%edi)
```

```
    pushl $buffer
```

```
    pushl $output
```

```
    call printf
```


addl \$8, %esp

pushl \$0

call exit

下面分別對入口為_start, main, xxxx（任意標籤）這三種情況下，用as, ld和gcc彙編cpuid2.s, 生成可執行文件。

1). 入口為_start

用as, ld生成可執行文件（入口為_start），如下圖所示：

```
[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# as --32 -o cpuid2.o cpuid2.s
[root@ken test1]# ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 -L/lib -lc cpuid2.o
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#
```

gcc生成可執行文件（入口為_start），如下圖：

```
[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# gcc -m32 -nostdlib -o cpuid2 -L/lib -lc cpuid2.s
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#
```

2). 入口為main

源程序把標籤_start改為main。此時ld可以使用-e main參數，那麼就會以main為程序起始點；gcc則有兩種方式，一是使用默認的編譯方式，即不帶-nostdlib參數，此時gcc會連接庫代碼，所以生成的執行文件的大小較大，另一種方式是使用-nostdlib，但是用-e main指定入口點。

用as, ld生成可執行文件（入口為main），如下圖所示：

```
[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# as --32 -o cpuid2.o cpuid2.s
[root@ken test1]# ld -m elf_i386 -e main -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 cpuid2.o -L/lib -lc
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#
```

gcc按第一種方式生成可執行文件（入口為main），如下圖：

```

[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# gcc -m32 -o cpuid2 cpuid2.s
[root@ken test1]# ll cpuid2
-rwxr-xr-x 1 root root 4799 11月 6 22:01 cpuid2
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]#

```

gcc按第二種方式生成可執行文件（入口為main），如下圖：

```

[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# gcc -m32 -nostdlib -e main -o cpuid2 cpuid2.s -L/lib -lc
[root@ken test1]# ll cpuid2
-rwxr-xr-x 1 root root 2133 11月 6 22:03 cpuid2
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]# |

```

第一種方法生成的大小為4799，第二生成的大小為2133。

3). 入口為xxxx（任意的標號）

cpuid2.s源程序把_start改為xxxx，即入口為xxxx。此時必須使用-e xxxx來彙編或者編譯。用as, ld生成可執行文件（入口為xxxx），如下圖所示：

```

[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# as --32 -o cpuid2.o cpuid2.s
[root@ken test1]# ld -m elf_i386 -e xxxx -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 cpuid2.o -L/lib -lc
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]# |

```

gcc生成可執行文件（入口為xxxx），如下圖：

```

[root@ken test1]# rm -f cpuid2 cpuid2.o
[root@ken test1]# gcc -m32 -nostdlib -e xxxx -o cpuid2 cpuid2.s -L/lib -lc
[root@ken test1]# ./cpuid2
CPUID is 'GenuineIntel'
[root@ken test1]# |

```

可見入口為xxxx的方式包含了以上入口為_start和main的情況。

綜合以上情況，無論入口點是什麼標籤，無論是運行在32位系統或者64位系統，都可以按照以下的命令來彙編和編譯彙編程序。

假設要彙編或編譯的彙編程序有n個，為input_file1.s, input_file2.s, ..., _filen.s。n大於等於1。

輸入文件列表input_file1.s, ...input_filen.s使用{input_file.s}表示，同樣,{input_file.o}表示一系列的.o文件

輸出可執行文件為output_file。

libc.so所在的絕對路徑用/libc_path表示，ld-linux.so.2所在的絕對路徑用ld-linux_path表示。

入口點標籤為entry_point。

[]括起來的是調用了C庫函數才需要的部分，去過沒有調用C庫，則不需要。

那麼as, ld生成可執行文件的命令如下：

as --32 -o input_file1.o input_file1.s

```
as --32 -o input_file2.o input_file2.s
```

...

```
as --32 -o input_filen.o input_filen.s
```

```
ld -m elf_i386 -e entry_point [-dynamic-linker /ld-linux_path/ld-linux.so.2] -o  
output_file [-L/libc_path -lc] {input_file.o}
```

gcc生成可執行文件的命令如下：

```
gcc -m32 -e entry_point -nostdlib -o output_file [-L/libc_path -lc] {input_file.s}
```

#註：如果源程序沒有調用C庫函數而又使用了[]中的指令連接或編譯，那麼會產生錯誤「/usr/lib/libc.so.1:
bad ELF interpreter: 沒有那個文件或目錄」

4. 外部程序標籤聲明

如果一個彙編程序文件中的代碼調用了另一個彙編程序文件的標號或者函數，那麼必須聲明這個標號或函數為.globl（應該是global的縮寫，全局的，可以跨文件調用）的。

比如彙編程序test3.s和test4.s，test3.s調用了test4.s的函數fun4，如果test4.s沒有.globl fun4這行，那麼編譯會提示錯誤，加上這行則沒有任何問題。

☐

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    call fun4
```

```
    movl $1, %eax
```

```
    movl $0, %ebx
```

```
    int $0x80
```

☐

.section .text

#.globl fun4

fun4:

movl \$1, %eax

movl \$2, %ebx

addl %ebx, %eax

ret

#符號把**.globl fun4**註釋了，那麼彙編成**test3.o**和**test4.o**後連接，會提示在函數**_start**中，沒有定義**fun4**。如下圖：

```
[root@ken test1]# rm -f test3 test3.o test4 test4.o
[root@ken test1]# as --32 -o test3.o test3.s
[root@ken test1]# as --32 -o test4.o test4.s
[root@ken test1]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o test3 -L/lib -lc test3.o test4.o
test3.o: In function `_start':
(.text+0x1): undefined reference to `fun4'
[root@ken test1]#
```

把#去掉後，讓**.globl fun4**起作用，則沒有任何問題，如下圖：

```
[root@ken test1]# rm -f test3 test3.o test4 test4.o
[root@ken test1]# as --32 -o test4.o test4.s
[root@ken test1]# rm -f test3 test3.o test4 test4.o
[root@ken test1]# as --32 -o test3.o test3.s
[root@ken test1]# as --32 -o test4.o test4.s
[root@ken test1]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o test3 -L/lib -lc test3.o test4.o
[root@ken test1]#
```

綜上，任何的標號或者函數，如果要準備給別的文件調用，那麼一定要用**.globl**聲明。

5. 彙編程序模版

經過上面的論述後，很容易得到了彙編程序的模版，如下：

.section .data

<初始化值的數據在這裡>

.section .bss

<未初始化的數據在這裡>

```
.section .text
```

```
.globl entry_point
```

```
entry_point:
```

<代碼指令在這裡>

其中，entry_point為程序起始點。

6. 彙編程序範例

書中的範例是用CPUID彙編指令去讀取CPU的廠商ID（Vendor ID）。瞭解這個程序之前先簡單說明幾個知識點。

1). 關於CPUID指令

輸入參數通過寄存器EAX傳入，執行CPUID後，輸出通過EBX，ECX，EDX傳出。這裡只要瞭解EAX=0時，ECX，EDX，EBX分別得到廠商ID的字符串的高4字節，中間4字節，低4字節。廠商ID的字符串按照小端排列，即先放低字節，即廠商ID為[EBX][EDX][ECX]。這可以通過對CPUID的測試代碼test_cpuid.s來進一步瞭解，如下代碼：

▢

```
# test_cpuid.s program
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    nop
```

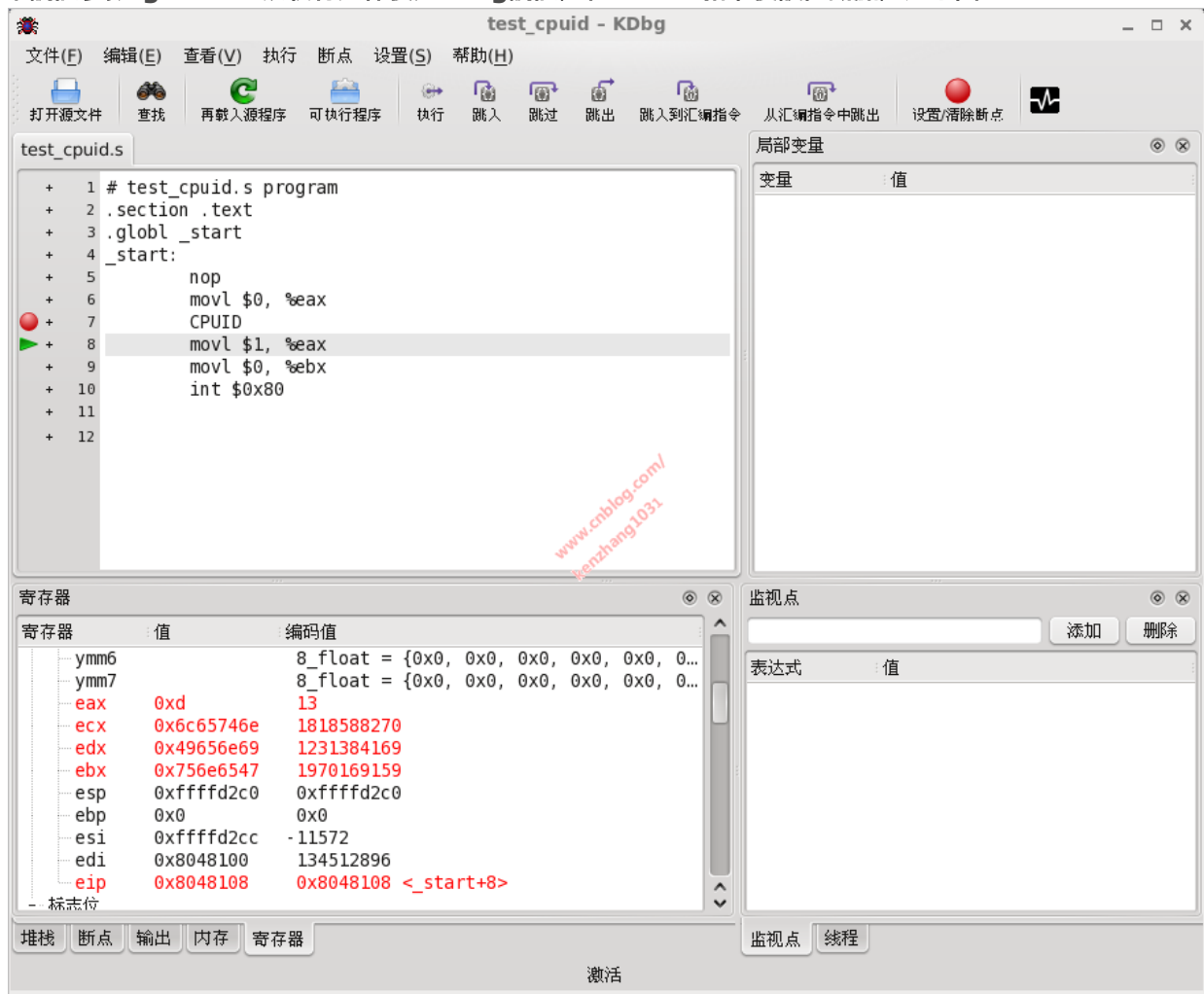
```
    movl $0, %eax
```

```
    CPUID
```

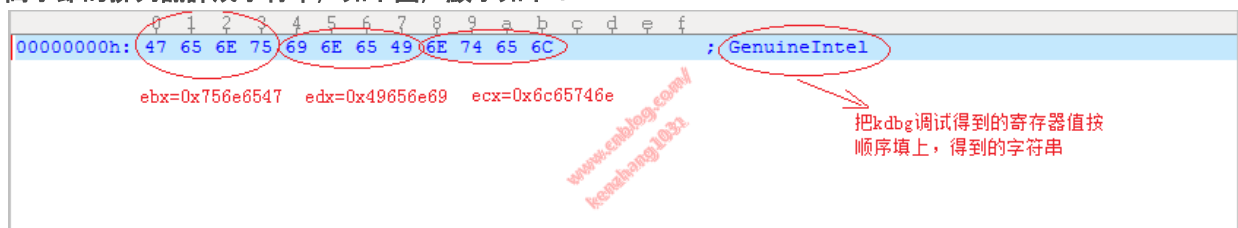
```
    movl $1, %eax
```

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

帶調試參數-gstabs生成執行文件後用kdbg調試，在CPUID指令後設置斷點，如下圖：



寄存器ebx, edx, ecx, 每個字節都是ASCII的編碼，把這些字節編碼按照ebx, edx, ecx從低字節到高字節的排列翻譯成字符串，如下圖，顯示如下：



即廠商ID是「GenuineIntel」。

2). 關於Linux系統調用

通過0x80號的軟件中斷（int \$0x80），可以調用Linux的內核函數，具體是哪個內核函數，由EAX寄存器決定，而傳遞給函數的參數則根據調用的函數而有不同的含義，一般由EBX，ECX，EDX來傳遞。書籍要到第十二章才會進一步討論。目前使用到的兩個系統調用先要簡單瞭解。

第一個是第1號調用，調用的是退出函數sys_exit(ret)，EAX=1表示調用號，EBX=ret傳遞第一個參數，表示返回給父進程的返回值，即sys_exit(ret)相當於如下的彙編代碼：

sys_exit(ret)系統調用的彙編代碼

```
movl $1, %eax
```

```
movl $ret, %ebx
```

int \$0x80

第二個調用是第4號調用，調用的是函數sys_write(int fd, const void *buf, size_t count)，三個參數分別用ebx，ecx，edx傳入，分別代表文件描述符，要寫的緩衝區首地址，緩衝區字節長度。眾所周知，Linux中用文件描述符1用於表示標準輸出（stdout），默認即顯示終端，因此要往顯示終端打印長度為length的字符串str，即sys_write(1, str, length)相當於以下的彙編代碼：

sys_write(1, str, length)系統調用的彙編代碼

```
movl $4, %eax
```

```
movl $1, %ebx
```

```
movl $str, %ecx
```

```
movl $length, %edx
```

int \$0x80

3). 完整的代碼範例cpuid.s

如下的代碼cpuid.s，讀取CPU的廠商ID，然後打印到屏幕上。代碼為：

▢

cpuid.s程序，打印CPU的廠商ID

.section .data

output:

```
.ascii "CPU ID is 'xxxxxxxxxxxx'\n" #
```

在data段定義字符串，最後的結果取代
xxxxxxxxxxxx後就是要輸出的內容

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    movl $0, %eax                                #獲
```

取CPU廠商ID到ebx,edx,ecx

```
    cpuid
```

```
    movl $output, %edi
```

```
    movl %ebx, 11(%edi)
```

```
    movl %edx, 15(%edi)
```

```
    movl %ecx, 19(%edi)                            #更
```

新xxxxxxxxxxxx，在字符串中是第11個字
節開始

```
    movl $4, %eax
```

```
    movl $1, %ebx
```

```
    movl $output, %ecx
```



```
movl $25, %edx
```

```
int $0x80
```

#輸

出字符串，包括換行在內長度為25字節

```
movl $1, %eax
```

```
movl $0, %ebx
```

```
int $0x80
```

#退

出程序，返回0值

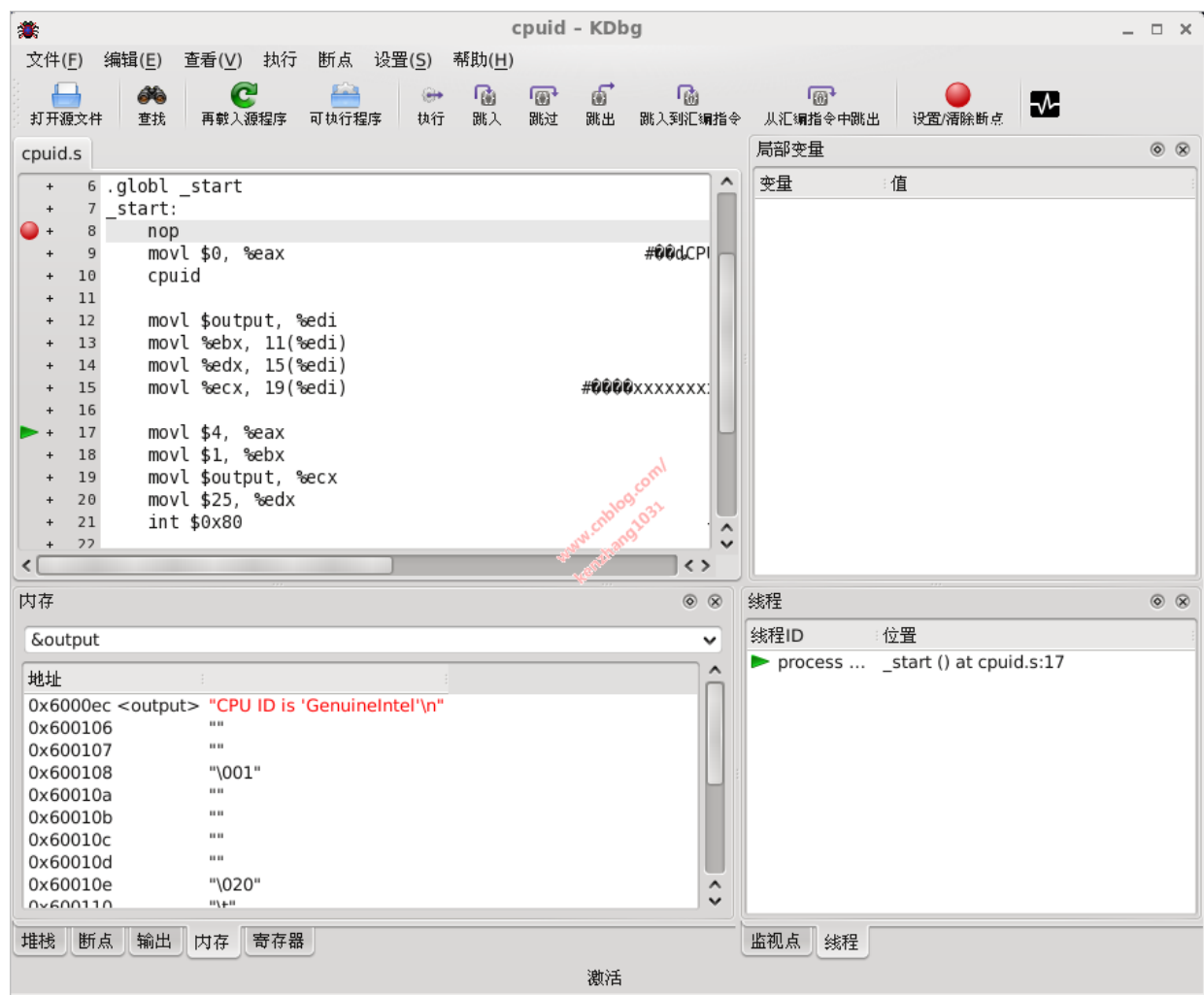
生成可執行文件以及執行的結果如下圖：

```
[root@ken test1]# rm -f cpuid cpuid.o
[root@ken test1]# as --32 -o cpuid.o cpuid.s
[root@ken test1]# ld -m elf_i386 -e _start -o cpuid cpuid.o
[root@ken test1]# ./cpuid
CPU ID is 'GenuineIntel'
[root@ken test1]# |
```

理解了上述的1)和2)所述的知識，加上代碼中的註釋，這段代碼不難理解，前提是需要有一點點彙編語言的基礎（不會movl都不知道吧？），不再贅述。

7. 調試彙編程序

用kdbg調試程序可以不用記gdb的指令，kdbg整個操作界面十分明朗，要單步運行，設置斷點，觀察變量，寄存器，內存，都沒有問題，如下圖：



但有一個特別需要注意的地方，`_start`之後的第一個`nop`指令，如果沒有，沒法觀察內存，會提示超出範圍，後來把`nop`加上則一切正常，該書上提到沒有這個`nop`會造成無法在`_start`處設置斷點，為gdb的bug，而我在kdbg中則是無法查看內存，斷點倒是可以設置。所以還是建議調試時，先增加這個`nop`指令。

8. 調用C庫函數

上面的`cpuid.s`代碼範例使用軟中斷調用linux內核函數來實現打印和退出程序，還有另外的方法實現打印和退出，那就是調用C語言的標準庫函數，打印是`printf`，退出是`exit`。

假如`CPUID=「GenuineIntel」`，那麼可以使用`printf(「CPU ID is 『%s'\n」,CPUID)`來打印和`cpuid.s`一樣的輸出信息。下面就以`printf`為例來描述彙編程序怎麼調用C庫函數。

調用子程序的指令是`call`，那麼調用`printf`就是用`call printf`，如果要實現`printf(「CPU ID is 『%s'\n」,CPUID)`，那裡面的兩個參數怎麼傳遞？這個要使用堆棧（stack），一般而言，C語言採用從右到左的順序入棧，即字符串`CPUID`的首地址先入棧，而後「CPU ID is 『%s'\n」的首地址入棧。入棧的彙編代碼是`pushl`。因此，可以按照如下的代碼實現`printf(「CPU ID is 『%s'\n」,CPUID)`。

假設「CPU ID is 『%s'\n」的首地址是`output`，讀回來的`CPUID`字符串首地址為`buffer`

那麼以下代碼實現C庫函數調用：`printf(output, buffer)`

```
pushl $buffer
```

```
pushl $output
```

```
call printf
```

注意，C語言堆棧底部用高地址，`pushl`入棧後堆棧指針`esp`寄存器變小了4，因此，如果在`call printf`後，`buffer`和`output`不再使用了，可以把`esp`設置為指向入棧前的地址，以便這兩個參數佔用的堆棧空間可以使用。即`pushl`兩次後，`esp`減少了8，因此`esp`需要加上8才能回覆原來的堆棧位置，即`addl $8, %esp`。

這裡可以簡單的理解：對於C庫函數`func(param1, param2, ..., paramn)`，調用的方法是先參數從右到左的順序入棧，然後`call func`。對於要深入研究的話，提供[abi接口文檔下載](#)。

`exit(ret)`的調用方式輕易實現：

```
# exit(ret)的彙編調用代碼
```

```
pushl $ret
```

```
call exit
```

32位系統的abi和64位的abi (*application binary interface*) 是不一樣的，因此x64的系統不能如此調用，所以在64位系統上運行32位的程序，必須按照32位的彙編和連接，否則會發生錯誤。具體可參考[彙編語言程序設計讀書筆記（2） - 相關工具64位系統篇](#)一文。

9. 通過調用C庫函數改寫cpuid.s-cpuid2.s範例

其實這個範例就是「3. 定義程序起始點」內容中的範例。理解了`cpuid.s`的意思，理解了怎麼調用C庫函數，那麼這個範例就可以輕易的理解了。另外對於這個範例還需說明兩點。

`.asiz`: 因為`printf`打印的是0結尾的字符串，因此需要定義0結尾的字符串，所以用`.asciz`，而不用`.ascii`。

`.lcomm`: 聲明留出一塊本地內存，這裡`.lcomm buffer, 12`表示留出12個字節大小的一塊本地內存，首地址用`buffer`表示。

代碼很明了，先是讀取CPUID到`ebx`, `edx`, `ecx`寄存器，然後通過`edi`寄存器為索引，把讀到的的字符串拼接到`buffer`中，然後分別以`output`和`buffer`為參數調用`printf`，相當於調用了`printf(「CPU ID is 『%s'\n」, buffer)`來打印，最後調用`exit(0)`返回。

10. 結束語

通過本篇的描述，應該知道怎樣設計一個彙編程序，包括系統調用和C庫調用怎樣使用，最後在32位系統或64位系統上彙編連接運行。

彙編語言程序設計讀書筆記（4） - 程序設計基礎之一

目錄：

- 一、數據定義
 - 1、變量數據定義
 - 2、常量數據定義
 - 3、緩衝區定義
- 二、尋址方式
 - 1、立即數尋址
 - 2、寄存器尋址
 - 3、直接尋址
 - 4、寄存器間接尋址
 - 5、寄存器相對尋址
 - 6、變址尋址
- 三、數據傳送和mov指令
 - 1、數據傳送規則
 - 2、mov指令
- 四、條件傳送數據cmov指令
 - 1、狀態標誌位
 - 2、cmov指令
- 五、交換數據
 - 1、xchg指令
 - 2、bswap指令
 - 3、xadd指令
 - 4、cmpxchg指令
 - 5、cmpxchg8b指令
- 六、堆棧
 - 1、堆棧簡介
 - 2、入棧指令push
 - 3、出棧指令pop
 - 4、所有寄存器入棧出棧
 - 5、堆棧的另一種用法
- 七、優化內存訪問

程序設計基礎部分主要內容包含數據定義，數據傳輸，尋址方式，彙編指令等等。涉及的內容較多，用多篇文章才可敘述完。

一、數據定義

彙編程序可以定義賦予了初始值的數據，且該數據在程序代碼中是可改變值的，類似於變量，也可以是不可改變值的，類似常量，還可以是程序使用的緩衝區，類似於函數中的無初值的局部變量。下面具體描述。

1、變量數據定義

前面的文章提過，`.data`段用於存放有初始值的數據，因此定義有初始值的變量數據就在`.data`段中定義。定義的模板如下：

```
.section .data
```

```
label:
```

```
type digit1, digit2, ..., digitn模板
```

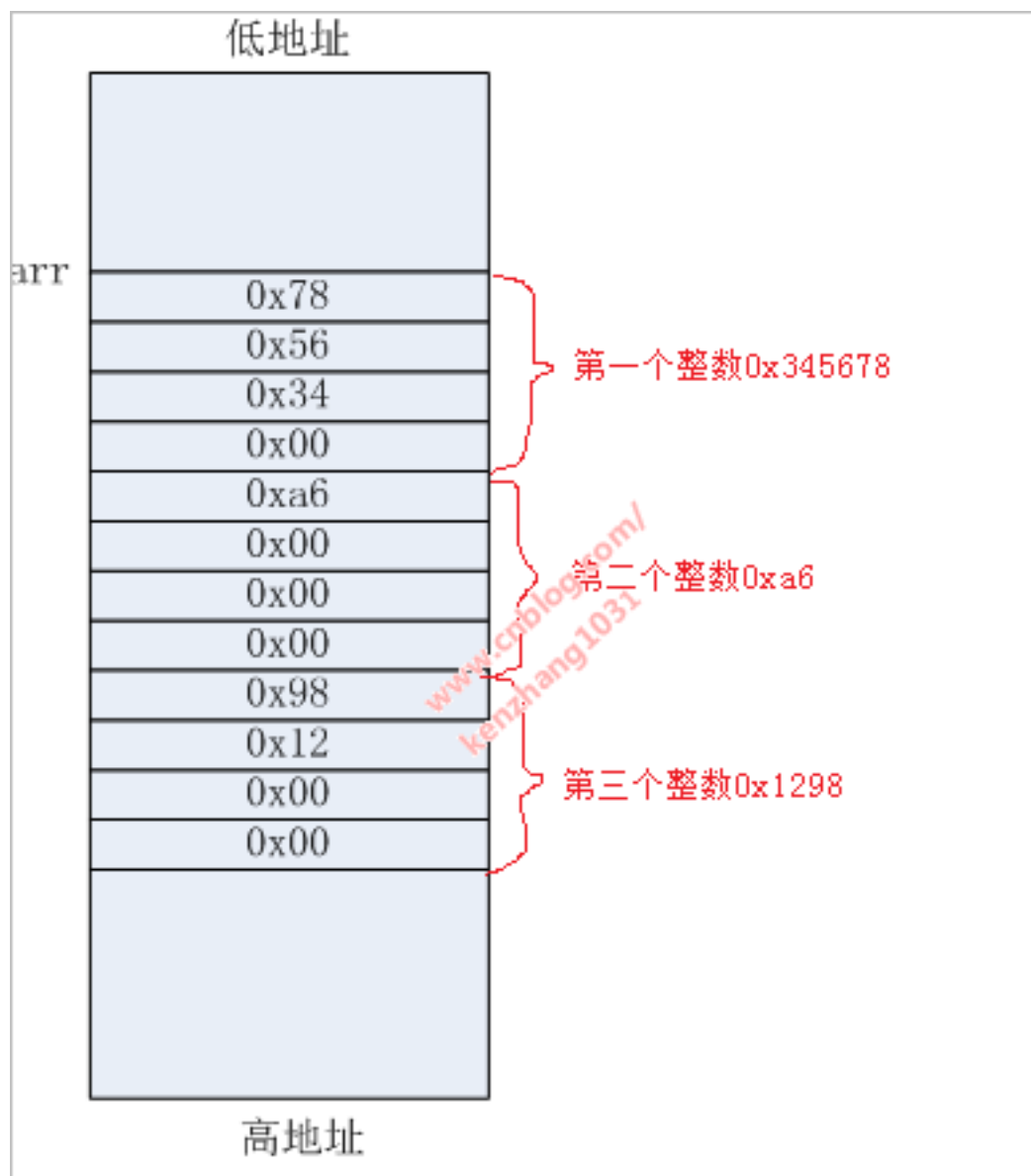
上面的語句在`.data`定義了`n`個數據`digit1`到`digitn`，首地址用標籤`label`表示，表示在`label`地址開始的內存空間定義了`n`個數據，這`n`個數據類型都是`type`表示的類型（`type`具體可能是`int`或`short`之類的，後面詳述），類型決定了每個數據所佔的內存空間是多少字節。而且對於IA-32的處理器而言，這`n`個數據是按小端對齊的方式存放在內存中的。比如以下的例子：

```
.section .data
```

```
arr:
```

```
.int 0x345678, 0xa6, 0x1298
```

定義了三個整型數據，起始地址為標籤`arr`，這三個數據按照從小端對齊的方式存儲於內存中，單個數據所佔的存儲空間是整型的大小，即32位，或說是4字節。內存圖示如下：



程序中要引用數據，就必須使用標籤，如上圖，arr是首地址，那麼arr+1的地址就表示下一個字節的地址，因為整數大小為4字節，因此下一個整數的起始地址就是arr+4，第三個整數的起始地址為arr+8，後面尋址方式那節會說到直接尋址，通過對內存地址的直接尋址，可以獲得該內存地址所存儲的數據；這個例子中，數據是按照.int定義的，然而並非必須按照int的長度來訪問這些數據，事實上，訪問的長度是由指令來決定，比如，用訪問一個字節的指令在arr地址訪問，那訪問到的數據就是0x78，按照兩個字節來訪問就是0x5678，下面的var_test1.s程序演示了如何在程序中使用這些定義的數據。

```

.section .data
arr:
    .int 0x345678, 0xa6, 0x1298
output:
    .asciz "The number is 0x%x\n"
.section .text
.globl _start
_start:
    nop
    pushl arr
    pushl $output
    call printf          # Print the first integer

    addl $8, %esp
    pushl arr+4
    pushl $output
    call printf          # Print the second integer

    addl $8, %esp
    pushl arr+8
    pushl $output
    call printf          # Print the third integer

    addl $8, %esp
    movl $0, %eax
    movw arr+1, %ax
    pushl %eax
    pushl $output
    call printf          # Print the word of ttt+1 starting address

    addl $8, %esp
    movl $0, %eax
    movb arr+1, %al
    pushl %eax
    pushl $output
    call printf          # Print the byte of ttt+1 starting address

    addl $8, %esp
    pushl $0
    call exit

```

編譯，連接，最後運行結果如下：

```

[root@ken test4]# rm var_test1 var_test1.o -f
[root@ken test4]# as --32 -gstabs -o var_test1.o var_test1.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o var_test1 -L/lib -lc var_test1.o
[root@ken test4]# ./var_test1
The number is 0x345678
The number is 0xa6
The number is 0x1298
The number is 0x3456
The number is 0x56
[root@ken test4]#

```

在var_test.s程序中的前三小段，通過32位的指令直接尋址arr，arr+4，arr+8內存地址的內容，併入棧作為printf的參數，打印出定義的三個整數；程序最後兩小段，在arr+1的地址處直接尋址，分別通過16位（movw）和8位（movb）的指令來訪問內存，對照上面所畫的內存圖，很容易理解，arr+1起始位置的內存數據字節為0x56，字則是0x3456，獲取到數據後入棧作為printf的參數，打印出來。因而得到如上的運行結果。

回到最開始定義的模版，類型type代表具體的一些類，包括整型，浮點型和字符串，具體可列表如下。

★ 整型數據類型：

type	類型說明
.byte	字節值（1字節）
.short	16位整數（2字節）
.int	32位整數（4字節）
.long	32位整數（4字節）
.quad	64位整數（8字節）
.octa	128位整數（16字節）

★ 浮點型數據類型：

type	類型說明
------	------

.float	單精度浮點型
.single	單精度浮點型
.double	雙精度浮點型

★ 字符串類型：

type	類型說明
.ascii	文本字符串（結尾不會自動加0）
.asciz	0結尾的字符串

2、常量數據定義

類似於C語言中用**#define**宏定義來定義常量，比如：

```
#define WEIGHT 60
```

定義**WEIGHT**為常量符號，代表**60**。彙編中也有類似的定義，定義模板為：

```
.section .data
```

```
.equ const_data, value
```

.equ是指令（英文單詞**equal**的縮寫，等於的意思），**const_data**是常量符號，用戶可以任意定義名稱，只要滿足變量命名規則則可，**value**就是**const_data**符號所代表的常量值，不可以在程序中被改動，使用時，要按照立即數的方式**constdata**來使用常量符號，比如**movl const_data, %eax**，就相當於**movl \$value, %eax**。

3、緩衝區定義

緩衝區是一段內存，內存中的數據沒有被初始化，以前文章說過，**.bss**段用於存儲未初始化或者全零的數據，因此緩衝區在**.bss**段中定義，指令**.comm**和**.lcomm**用於定義緩衝區，兩者的區別在於**.lcomm**定義僅供本地使用的緩衝區，前面的字母**l**正是單詞**local**的首字母，表示本地，局部的意思。定義模板為：

```
.section .bss
```

```
.comm buffer, length
```

或者

```
.section .bss
```

```
.lcomm buffer, length
```

buffer表示緩衝區首地址，**length**表示緩衝區字節長度。比如：

```
.section .bss
```

```
.comm buf, 10000
```

定義了10000個字節的緩衝區，首地址為**buf**。

緩衝區不會包含在可執行程序中，而**.data**中定義的變量數據會包含在可執行程序中，比較以下兩段代碼，**buff_test.s**和**data_test.s**，兩者都定義了10000個字節的長度，所不同的是**buff_test.s**在**.bss**定義，**data_test.s**在**.data**定義。最後觀察兩者的可執行文件的長度，**data_test**文件長度很長，超過了10000個字節，而**buff_test**文件才幾百字節，可見在**.data**定義的10000字節包含在可執行文件中了，而**.bss**定義的10000字節則沒有包含到可執行文件中。

代碼如下：

▢

```
.section .bss  
.comm buff1, 10000  
.section .text  
.globl _start
```

```
_start:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```



```
.section .data
buffer:
    .fill 10000
.section .text
.globl _start
_start:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

說明：.fill 10000表示用0填充10000個字節長度的數據。

彙編連接後，可見一個長度為574字節，另一個為10575字節，如下圖。

```
[root@ken test4]# as --32 -o buff_test.o buff_test.s
[root@ken test4]# ld -m elf_i386 -e _start -o buff_test buff_test.o
[root@ken test4]# as --32 -o data_test.o data_test.s
[root@ken test4]# ld -m elf_i386 -e _start -o data_test data_test.o
[root@ken test4]# ll buff_test data_test
-rwxr-xr-x 1 root root 574 11月 20 00:20 buff_test
-rwxr-xr-x 1 root root 10575 11月 20 00:20 data_test
[root@ken test4]#
```

二、尋址方式

上面敘述了數據的定義，數據定義後，需要在程序中使用它，即在程序中如何訪問它，比如需要讀取它，或者把其它的數值寫給它，就涉及到要通過什麼樣的方式去訪問這些數據，一般分為立即數尋址，寄存器尋址，直接尋址，寄存器間接尋址，寄存器相對間接，變址尋址，下面分別詳述。

1、立即數尋址

立即數是指數據在指令碼語句中直接指定的，而且在運行時不能改動的，用符號開始的數據，比如**int0x80**中的**0x80**就是立即數，以及指令中類似**1**，**\$0**之類以美元符號開頭的數據都是立即數。

立即數尋址的格式為：**\$imm**

2、寄存器尋址

是指數據存在於寄存器中，比如數據在寄存器**eax**中，那麼讀取寄存器**eax**的內容就得到了寄存器**eax**中的數據，像這樣通過讀取寄存器來獲取寄存器內的數據的方式就叫寄存器尋址。比如**movl %eax, %ebx**語句，指令中的**%eax**和**%ebx**都是寄存器尋址，即從寄存器**eax**中讀取源數據，寫到寄存器**%ebx**中；又比如**movl 1,1**是立即數尋址，即把立即數**1**寫到寄存器中。

寄存器尋址的格式為：**%reg**

3、直接尋址

是指通過內存的地址直接去訪問內存中的數據，直接尋址和立即數尋址要區分開，比如定義了標籤**label**，那麼**label**就表示一個內存的地址，地址裡面存放數據**digit**，因此指令中通過**label**這個內存地址來訪問數據就是直接尋址的方式，而如果使用**label**就是把地址當成立即數來使用，述立即數尋址，**digit**也是把內存中的數值當立即數使用，也屬立即數尋址。舉個完整程序的例子：

1: **.section .data**

2: **label:**

3: **.int 0x123456**

4: **.section .text**

5: **.globl _start**

6: **_start:**

7: **movl \$label, %ebx**

8: **movl label, %edx**

9:

10: **movl \$1, %eax**

11: **movl \$0, %ebx**

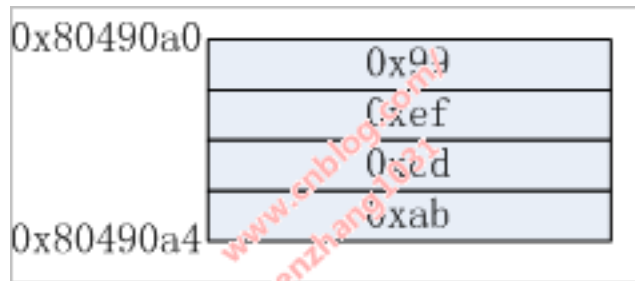
12: **int \$0x80**

第7行中的**\$label**表示立即數尋址，**ebx**中得到的是**label**的地址值，第8行中的**label**表示直接尋址，**edx**中得到的是**label**地址所對應內存中的數據，即**0x123456**。

直接尋址的格式為：**address**

4、寄存器間接尋址

寄存器間接尋址是指寄存器中存儲的是內存地址，通過寄存器間接的訪問到地址中所對應的內存數據的這樣一種尋址方式，比如寄存器**eax**存儲的是數據在內存中的地址**addr**，通過(**%eax**)的間接的方式（小括號括著寄存器），可以訪問到地址**addr**所對應的內存中的數據，比如如下圖內存地址**0x80490a0**的位置存儲了數據**0x99**，**0xef**，**0xcd**，**0xab**。



那麼如果`eax=0x80490a0`，那麼通過寄存器`eax`間接尋址(`%eax`)，可以訪問到整數數據 `0xabcdef99`。

寄存器間接尋址要和寄存器尋址區別開來，如上圖，如果是寄存器尋址，那麼通過`eax`獲取到的是寄存器裡的數值，即地址`0x80490a0`，即下面兩個指令結果是不一樣的：

1: `movl %eax, %ebx` # 寄存器尋址, `ebx=0x80490a0`

2: `movl (%eax), %ebx` # 寄存器間接尋址, `ebx=0xabcdef99`

寄存器間接尋址的格式為：**`(%reg)`**

5、寄存器相對尋址

寄存器相對尋址是在寄存器間接尋址的基礎上，加上一個相對當前地址的偏移來尋址。比如`eax`存儲了地址 `addr`，`(%eax)` 是間接尋址，如果往內存的高地址的方向偏移4個字節，即要訪問地址`addr+4`的數據，那麼可以通過`4(%eax)`這樣的相對尋址的方式，來訪問`addr+4`這個地址所對應的內存的數據，同樣的，`-4(%eax)`也是相對尋址，訪問的是 `addr-4`這個地址所對應的內存的數據。

比如，`eax=0x80490a0`，那麼`4(%eax)`訪問的就是地址為`0x80490a4`的內存位置的數據。在指令：
`movl 4(%eax), %ebx` 中，**`4(%eax)`**這樣的方式就表示寄存器相對尋址。

相對尋址的格式為：**`offset(%reg)`**

6、變址尋址

要說清楚變址尋址，要從它的格式說起。

變址尋址的格式為：**base(offset, index, size)**

base指的是內存的基址，offset指的是偏移地址，index指的是數據的序號，size指的是單個數據的字節長度，可以拿C語言中的數組的訪問來比喻，比如整型數組a[10]，那麼base可以認為是數組的首地址a，offset可以認為是0（因為數組的下標從0開始），index是索引，相當於數組的下標，即從0到9的索引值，size則是數組中單個數據的字節長度，整數字節長度就是4，那麼對數組a[i]的訪問，就使用a(0, index, 4)這樣的形式。舉個具體的例子來說明：

```
.section .data
```

```
arr:
```

```
    .int 1, 7, 9, 46, 68, 13, 93, 88, 2, 555
```

arr開始的地址定義了10個整數，如果基址base=arr，如果偏移為0，即offset=0，整數字節長度為4字節，那麼arr(0, 0, 4)就可以訪問第一個數據1，arr(0, 3, 4)就可以訪問第4個數據46。這裡只是為了便於說明而採用了偽代碼arr(0, 3, 4)的方式，實際上格式中的offset和index必須採用寄存器的方式，比如eax=0，edi=3，那麼指令中要採用arr(%eax, %edi, 4)的方式，格式中，如果有哪個值為0值，可以不寫，但逗號不可以省略，比如寫成arr(, %edi, 4)的方式，省略了offset，因為它是0。

實際上，變址尋址訪問的數據地址位於**base + offset + index * size**的位置。

下面的程序為變址尋址的範例，程序作用是打印數組的值。



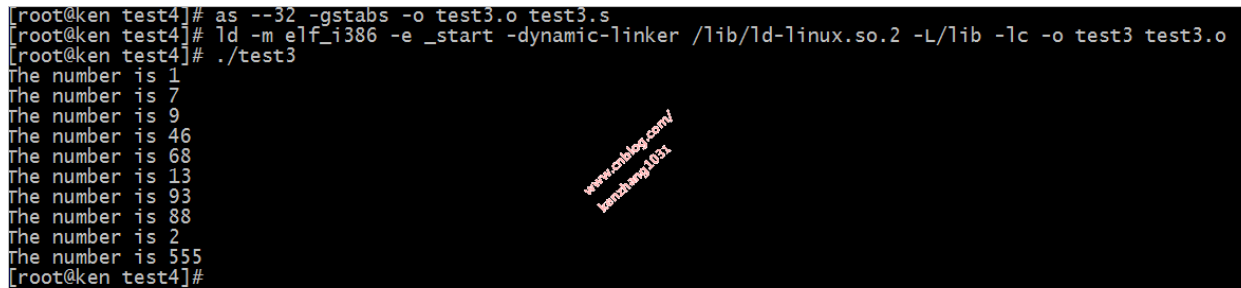
```
.section .data
output:
    .asciz "The number is %d\n"
arr:
    .int 1, 7, 9, 46, 68, 13, 93, 88, 2, 555
length:
    .int (. - arr) >> 2
.section .text
.globl _start
_start:
    nop
    movl $0, %edi
loop:
    movl arr(, %edi, 4), %eax
    pushl %eax
    pushl $output
```

```
call printf
addl $8, %esp
inc %edi
cmpl length, %edi
jne loop

pushl $0
call exit
```

文章第一次出現`(. - arr) >> 2`這樣的語句，其中的`.`表示的是當前的地址，減去`arr`得到的差值就是`arr`到當前地址這一段內存空間的字節大小，右移2位表示除以4，得到的是`arr`到當前地址這一段內存空間的整數個數（因為一個整數佔4字節，所以字節數除以4），總言之，可認為是為了求出數組`arr`的整數的個數。建議採用這樣的方式來獲取數據數目或者字符串長度，雖然這個例子中是10個整數很好數，如果是成千上萬甚至更多的話，那怎麼數？

程序編譯鏈接後執行如下圖所示，數據被正確訪問並打印。

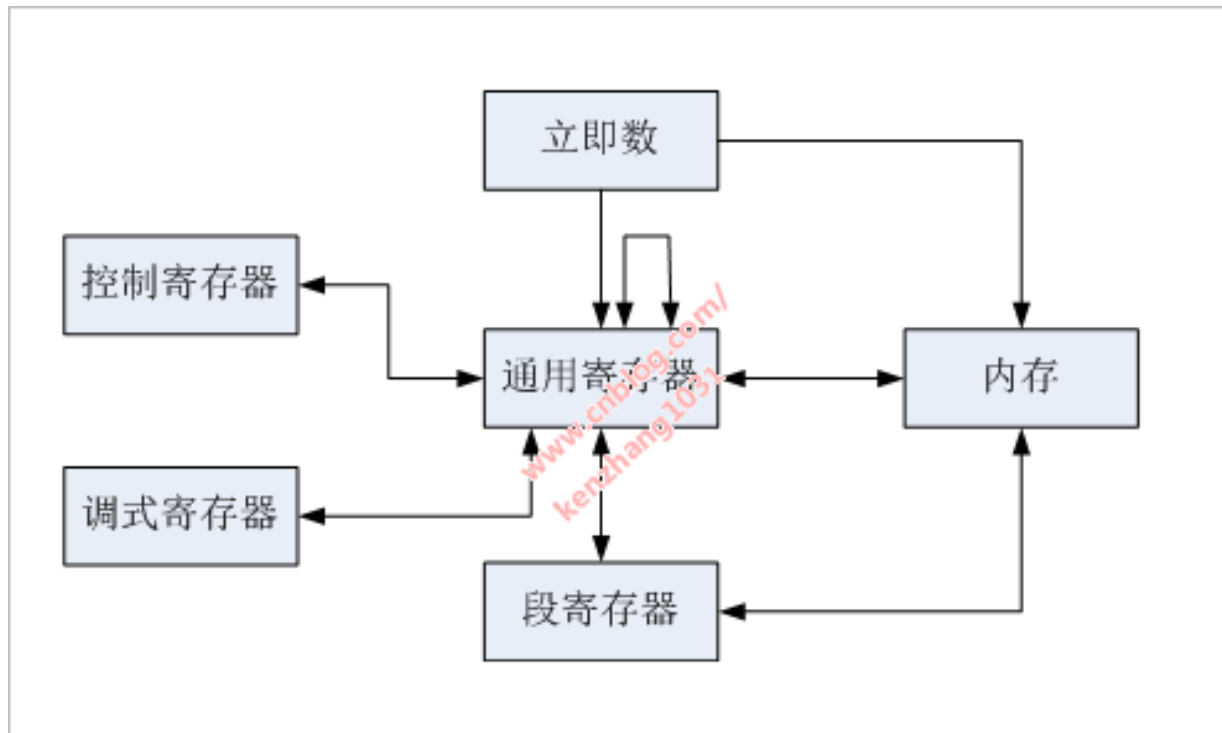


```
[root@ken test4]# as --32 -gstabs -o test3.o test3.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -L/lib -lc -o test3 test3.o
[root@ken test4]# ./test3
The number is 1
The number is 7
The number is 9
The number is 46
The number is 68
The number is 13
The number is 93
The number is 88
The number is 2
The number is 555
[root@ken test4]#
```

三、數據傳送和MOV指令

1、數據傳送規則

傳數據最常用的指令是`mov`指令。學過單片機彙編或者對8086彙編有所瞭解的人都知道，傳送數據是有一定規則的，比如一般而言，不能從內存到內存，即兩個內存之間直接進行傳送，而要通過寄存器。部件之間傳送數據的關係可以用下圖表示：



圖示說明除了立即數以外，其它部件的和通用寄存器之間都可以相互傳數據。



有一個要特別說明的，就是movs指令可以在內存之間傳送字符串。後續說到字符串處理的內容時會詳述。

2、mov指令

mov指令的格式是：**movx src, dst**

這裡x代表的是字母q, l, w, b其中之一，分別表示傳送的是64位，32位，16位，8位的數據。linux中採用的是at&t的指令格式，指令最後的字母q, l, w, b就表示傳送的數據的長度。

src表示源操作數，dst表示目的操作數，它們的規則看上面傳送關係的圖示。比如可以是從立即數到寄存器，那麼src就是立即數，dst就是寄存器。

無論src是內存，或是dst是內存時，內存的尋址都可以採用尋址方式一段說述的尋址方式，下面例子演示了mov指令的用法：

1: movl \$123, %eax # 立即數到寄存器

2: movl \$123, arr # 立即數到內存,內存用直接尋址

3: movl \$123, (%eax) # 立即數到內存,內存用間接尋址

4: movl \$123, 4(%eax) # 立即數到內存,內存用相對尋址

5: movl \$123, arr(, %edi, 4) # 立即數到內存,內存用變址尋址

6: movl %eax, %ebx # 寄存器到寄存器

7: movl %eax, arr # 寄存器到內存,內存用直接尋址

8: movl %eax, (%ebx) # 寄存器到內存,內存用間接尋址

9: movl %eax, 4(%ebx) # 寄存器到內存,內存用相對尋址

10: movl %eax, arr(, %edi, 4) # 寄存器到內存,內存用變址尋址

11: movl arr, %eax # 內存到寄存器,內存用直接尋址

12: movl (%ebx), %eax # 內存到寄存器,內存用間接尋址

13: movl 4(%ebx), %eax # 內存到寄存器,內存用相對尋址

14: movl arr(, %edi, 4), %eax # 內存到寄存器,內存用變址尋址

四、條件傳送數據cmov指令

條件傳送和上面所述的數據傳送在傳送上是一樣的，所不同的是要滿足某個或某些條件時才執行傳送數據。

條件傳送傳送數據的格式為：**cmovx src, dst**

前面的字母**c**為英文**condition**（意思是條件）的首字母，後面的**x**表示條件，不同的條件有不同的表示，通常是一到三個字母，比如無符號數大於這個條件，用字母**a**表示（**above**的第一個字母），也就是指令碼為**cmova**；後面的**src**和**dst**和數據傳送中論述的一樣，不再詳述。

比如在C語言中經常見到類似如下的語句：

```
if(a > b)
```

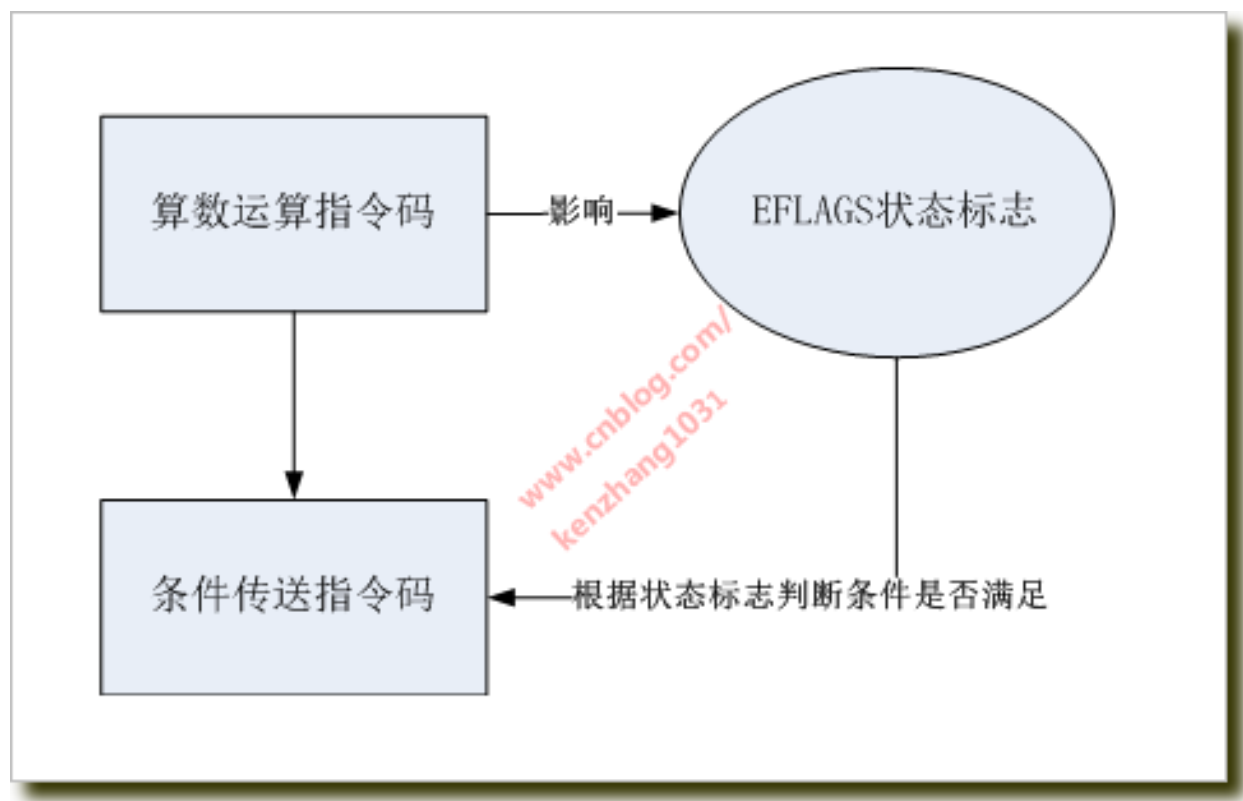
```
    max = a;
```

那麼在彙編語言中，如果**eax=a**，**ebx=b**，**max**在內存中，上面的語句就是：如果**eax**大於**ebx**這個條件滿足，那麼就把數據從**eax**寄存器傳送到內存**max**中，這就要用到條件傳送，此時用彙編的語句為：

```
cmpl %ebx, %eax
```

```
cmova %eax, max
```

cmpl是比較指令，以後的文章會詳述，這裡只要理解是比較**eax**和**ebx**，判斷是否滿足**eax**大於**ebx**條件的，**cmova**是表示如果滿足大於（字母**a**）的這個條件（字母**c**），就把**eax**中的數據傳送到**max**地址所對應的內存中。那麼電腦是怎麼通過**cmpl**的指令結果來判斷是否滿足大於這個條件的呢？對**IA-32**的**cpu**而言，是通過一個叫**EFLAGS**的標誌寄存器中的狀態位來判斷的，指令中的算數運算的結果會反映到狀態標誌中，而條件傳送語句則判斷**EFLAGS**中的狀態標誌是否滿足條件來決定是否傳送數據。這個關係可以用如下的圖示來描述：



下面對EFLAGS寄存器的狀態標誌做進一步的敘述。

1、狀態標誌位

eflags是32位的寄存器，常用於條件判斷的狀態標誌位有6個，用於指示算數運算結果的，列表如下：

狀態標誌	位號	名稱	描述
CF	bit0	進位標誌 (Carry Flag)	算數運算的結果在最高有效位產生了進位或者借位，該標誌置1，否則清0。 或者說該位用於指示無符號數是否溢出，比如n位（n為8，16或者32）無符號數範圍是[0, 2的n次方-1]的閉合區間，如果對應位數的指令（指令中最後的字母為b，w，l分別指示是8位，16位，32位）運算的結果超出了這個範圍，那麼該標誌為1。否則為0。如果結果考慮了進位或者借位，那結果還是正確的，可以用於多字（節）的運算。

PF	bit2	奇偶標誌 (Parity Flag)	運算結果的最低字節有偶數個1，該位置1，否則該位清0。即結果的最低字節和該位所包含的位中1的數目一定是奇數個。
AF	bit4	輔助進位標誌 (Auxiliary Carry flag)	如果算數運算中bit3這一位產生了進位或者借位，那麼該位置1，否則清0。
ZF	bit6	零標誌 (Zero flag)	如果結果為0，那麼該位置1，否則清0。
SF	bit7	符號標誌 (Sign flag)	等於有符號數的符號位（有符號數用最高位表示數值的正負符號，1表示負數，0表示正數），結果為負，該位置1，否則清0。
OF	bit11	溢出標誌 (Overflow flag)	該位用於指示有符號數是否溢出，比如n位（n為8，16或者32）無符號數範圍是 $[-(2^{n-1}), 2^{n-1}-1]$ 的閉合區間，如果對應位數的指令（指令中最後的字母為b，w，l分別指示是8位，16位，32位）運算的結果超出了這個範圍，那麼結果溢出，該標誌為1。否則為0。如果結果溢出，那麼結果不正確了。

以上標誌只有CF標誌可以用指令直接置位或者清零，其它標誌只能通過算數運算的指令來影響。

其實上面的**cmpl %ebx, %eax**指令就是執行**eax-ebx**這樣的算數運算，結果會反映到狀態標誌中。如果作為無符號數的形式，**eax**大於**ebx**，那麼就會把CF清零，如果等於就會把ZF置1，如果小於，就會引起CF置1；同樣作為有符號數則會影響到SF，ZF，OF標誌。**cmpl**這樣的比較指令以後會說到。

2、cmov指令

如上所述，指令碼**cmov**之後接一個字母到三個字母來表示條件，先複習一下英文單詞，再理解指令碼中的條件為什麼用這樣的字母表示，這樣更容易理解，無符號數用**above**（高）和**below**（低）表示大於和小於，有符號數用**greater**（更大）和**less**（更少）表示大於和小於，**equal**表示等於，**zero**表示零，**sign**表示符號（上面敘述了符號位為1表示為負數），**carry**表示進位，**parity**表示奇偶校驗，**even**表示偶數，**odd**表示奇數，**overflow**表示溢出，**no**表示非，不。通過這些單詞的第一個字母的組合可以產生指令碼中的條

件，比如na，表示no above，即無符號數的不大於，也就是小於或等於，所以和below or equal是等價的，即na和be等價；同樣parity even簡寫為pe，表示偶校驗，等等。

基於這樣的字母組合，得到一系列的條件羅列如下表格，等價的條件羅列在一塊：

條件	描述	狀態標誌
a/nbe	無符號數大於（above）/不小於等於（no below equal）	CF=0 且 ZF=0
ae/nb	無符號數大於等於（above equal）/不小於（no below）	CF=0
b/nae	無符號數小於（below）/不大於等於（no above equal）	CF=1
be/na	無符號數小於等於（below equal）/不大於（no above）	CF=1 或 ZF=1
c	有進位（carry）	CF=1
nc	無進位（no carry）	CF=0
g/nle	有符號數大於（greater）/不小於等於（no less equal）	SF=OF 且 ZF=0
ge/nl	有符號數大於等於（greater equal）/不小於（no less）	SF=OF
l/nge	有符號數小於（less）/不大於等於（no greater equal）	SF=^OF
le/ng	有符號數小於等於（less equal）/不大於（no greater）	SF=^OF 或 ZF=1
o	溢出（overflow）	OF=1
no	未溢出（no overflow）	OF=0

s	帶符號 (sign) , 即負數	SF=1
ns	無符號 (no sign) , 即非負數	SF=0
e/z	相等 (equal) / 為零 (zero)	ZF=1
ne/nz	不等於 (no equal) / 不為零 (no zero)	ZF=0
p/pe	奇偶校驗位置一 (parity) / 偶校驗 (parity even)	PF=1
np/po	奇偶校驗位清零 (no parity) / 偶校驗 (parity odd)	PF=0

cmov 接著上面的條件字母就得到了所有的條件傳送數據的指令碼，比如 **cmova**, **cmovnbe**, 等等，不一一列舉。

下面通過打印有符號整數數組中最大值和最小值為例，說明條件傳輸的用法。程序如下：



```
.section .data
output_max:
    .asciz "the max number is %d\n"
output_min:
    .asciz "the min number is %d\n"
arr:
    .int 3456, -5678, -3, 99999, 4, -1234567, 7564321, 34, -789, 6982
size:
    .int (. - arr) >> 2
.section .bss
    .lcomm max, 4
    .lcomm min, 4
.section .text
.globl _start
_start:
    nop
    movl %ecx, arr        # ecx save max
    movl %edx, arr        # edx save min
    movl $0, %edi
loop:
    movl arr(, %edi, 4), %eax
    cmpl %ecx, %eax
    cmovg %eax, %ecx
    cmpl %edx, %eax
    cmovl %eax, %edx
```

```

inc %edi
cmpl size, %edi
jne loop
movl %ecx, max
movl %edx, min
pushl max
pushl $output_max
call printf
addl $8, %esp
pushl min
pushl $output_min
call printf
addl $8, %esp
pushl $0
call exit

```

彙編連接後執行如下圖所示：

```

[root@ken test4]# as --32 -gstabs -o cmov_test.o cmov_test.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o cmov_test -L/lib -lc cmov_test.o
[root@ken test4]# ./cmov_test
the max number is 7564321
the min number is -1234567
[root@ken test4]#

```

五、交換數據

如果要交換兩個位置的數據，如果僅使用mov指令的話，就需要一個臨時的緩衝區，先把位置1的數據傳輸給臨時緩衝區，再把位置2的數據傳送到位置1，最後把臨時緩衝區的數據傳給位置2。相當於分了3步。彙編語言提供瞭解決這個問題的指令，一個指令就可以實現數據交換的操作。詳述如下。

1、xchg指令

xchg格式為：**xchg op1, op2**

op1和op2表示操作數1和操作數2，兩者可以是寄存器或者內存，但不可以同時都是內存，而且兩個操作數必須位數一樣，即同是8位，16位，或32位的操作數。

當操作數之一為內存時，處理器會lock住這個內存位置，防止交換過程中被其它處理器訪問，lock處理很耗時間，可能對性能會有不良影響。

下面以對一個數組進行倒序來演示這個指令的用法，代碼如下，主要是執行第一個數據和最後一個數據交換，第二個和倒數第二個交換，...，如此，直到把所有的數據都交換了，數組順序就倒過來了。

☐

```
.section .data
output:
    .asciz "the number is %d\n"
arr:
    .int 1, 2, 3, 4, 5, 6, 7, 8, 9
size:
    .int (. - arr) >> 2
.section .text
.globl _start
_start:
    nop
    movl $0, %esi
    movl size, %edi
    dec %edi
loop:
    movl arr(, %esi, 4), %eax      # Reverse Array
    xchg arr(, %edi, 4), %eax
    movl %eax, arr(, %esi, 4)
    inc %esi
    dec %edi
    cmpl %edi, %esi
    jl loop

    movl $0, %edi                # print
loop2:
    movl arr(, %edi, 4), %eax
    pushl %eax
    pushl $output
    call printf
    addl $8, %esp
    inc %edi
    cmpl size, %edi
    jne loop2

    pushl $0                     # exit
    call exit
```

彙編連接執行結果如下圖，數組反轉了。

```
[root@ken test4]# as --32 -gstabs -o reverse.o reverse.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o reverse -L/lib -lc reverse.o
[root@ken test4]# ./reverse
the number is 9
the number is 8
the number is 7
the number is 6
the number is 5
the number is 4
the number is 3
the number is 2
the number is 1
[root@ken test4]#
```

2、bswap指令

bswap指令格式為：**bswap reg32**

reg32指的是32位的寄存器。如果reg32的四個字節從高到低記為byte3-byte0，那麼這個指令的作用是把byte3和byte0交換，byte2和byte1交換，即實現了大小端的數據轉換。

以下例子示範這個指令的使用，程序為把num地址中的數據0x12345678轉為大端的排列方式，即把0x12345678轉為0x78563412，源代碼為：



```
.section .data
output:
    .asciz "the number is 0x%x\n"
num:
    .int 0x12345678
.section .text
.globl _start
_start:
    nop
    movl num, %eax
    bswap %eax
    pushl %eax
    pushl $output
    call printf
    addl $8, %esp

    pushl $0
    call exit
```

彙編連接後執行結果如下圖：

```
[root@ken test4]# as --32 -gstabs -o bswap_test.o bswap_test.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o bswap_test -L/lib -lc bswap_test.o
[root@ken test4]# ./bswap_test
the number is 0x78563412
```

3、xadd指令

該指令格式為：**xadd src, dst**

src必須是寄存器，dst可以是寄存器或者內存，該指令是交換src和dst的數據，然後把兩個數據的和存到dst中，即執行後，src=dst，dst=src+dst。

例子程序如下：



```
.section .data
output:
    .asciz "eax = 0x%x, ebx = 0x%x\n"
.section .text
.globl _start
_start:
    nop
    movl $0x12, %eax
    movl $0x34, %ebx
    xadd %ebx, %eax
    pushl %ebx
    pushl %eax
    pushl $output
    call printf
    addl $12, %esp

    pushl $0
    call exit
```

彙編連接後運行，如下圖所示：

```
[root@ken test4]# as --32 -gstabs -o xadd_test.o xadd_test.s
[root@ken test4]# ld -m elf_i386 -e _start -dynamic-linker /lib/ld-linux.so.2 -o xadd_test -L/lib -lc xadd_test.o
[root@ken test4]# ./xadd_test
eax = 0x46, ebx = 0x12
```

4、cmpxchg指令

該指令格式為：**cmpxchg src, dst**

src必須是寄存器，**dst**可以是寄存器或者內存，而且**src**位數要和**dst**一致，如果**dst**是32（16/8）位的，那麼**dst**和**eax**（**ax/al**）比較，相等的话就把**src**傳送到**dst**，如果不等就把**dst**傳送到**eax**（**ax/al**）。

示例如下：

▢

```
.section .text
.globl _start
_start:
    nop
    movl $0x12, %eax
    movl $0x34, %ebx
    movl $0x12, %ecx
    movl $0x56, %edx
    cmpxchg %ebx, %ecx
    cmpxchg %ebx, %edx

    pushl $0
    call exit
```

用kdbg看調試的過程如下圖：

```

+ 1 .section .text
+ 2 .globl _start
+ 3 _start:
+ 4     nop
+ 5     movl $0x12, %eax
+ 6     movl $0x34, %ebx
+ 7     movl $0x12, %ecx
+ 8     movl $0x56, %edx
+ 9     cmpxchg %ebx, %ecx
+10     cmpxchg %ebx, %edx
+11
+12     pushl $0
+13     call exit
+14

```

寄存器

寄存器	值	编码值
GP和其它		
eax	0x12	18
ecx	0x12	18
edx	0x56	86
ebx	0x34	52

開始賦給初值：eax=0x12, ebx=0x34, ecx=0x12, edx=0x56

執行cmpxchg %ebx, %ecx後，由於ecx等於eax，那麼就會把ebx傳到ecx中，即ecx變成了0x34，下圖的調試結果證實如此：

```

+ 1 .section .text
+ 2 .globl _start
+ 3 _start:
+ 4     nop
+ 5     movl $0x12, %eax
+ 6     movl $0x34, %ebx
+ 7     movl $0x12, %ecx
+ 8     movl $0x56, %edx
+ 9     cmpxchg %ebx, %ecx
+ 10    cmpxchg %ebx, %edx
+ 11
+ 12    pushl $0
+ 13    call exit
+ 14

```

寄存器

寄存器	值	编码值
GP和其它		
eax	0x12	18
ecx	0x34	52
edx	0x56	86
ebx	0x34	52

再執行 `cmpxchg %ebx, %edx`，由於 `edx` 不等於 `eax`，因此，會把 `edx` 傳送到 `eax`，即 `eax` 變成了 `0x56`，下圖的調試結果也證實了如此：

```

+ 1 .section .text
+ 2 .globl _start
+ 3 _start:
+ 4     nop
+ 5     movl $0x12, %eax
+ 6     movl $0x34, %ebx
+ 7     movl $0x12, %ecx
+ 8     movl $0x56, %edx
+ 9     cmpxchg %ebx, %ecx
+ 10    cmpxchg %ebx, %edx
+ 11
+ 12    pushl $0
+ 13    call exit
+ 14

```

寄存器

寄存器	值	编码值
GP和其它		
eax	0x56	86
ecx	0x34	52
edx	0x56	86
ebx	0x34	52

5、cmpxchg8b指令

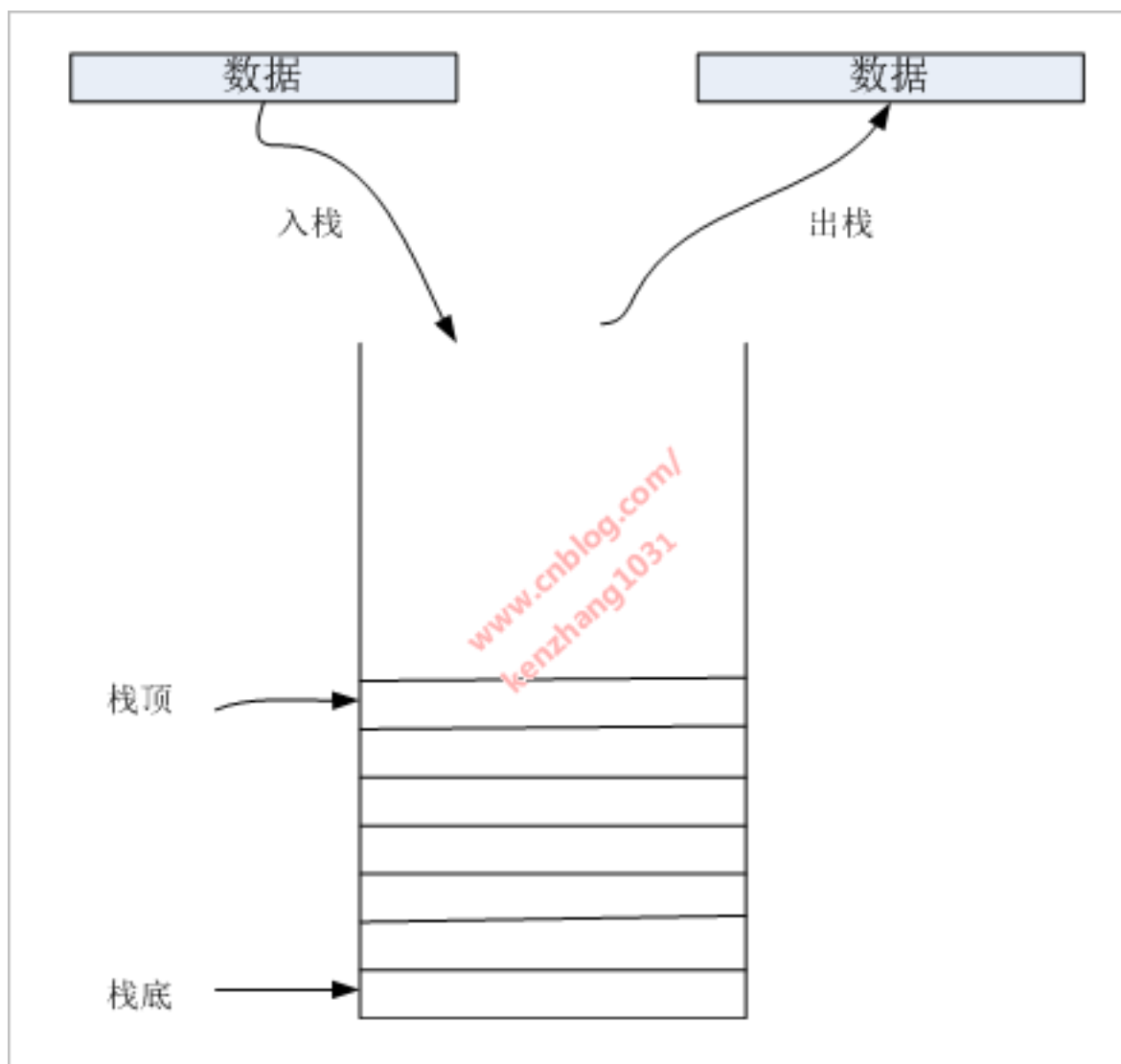
指令格式為：**cmpxchg8b dst**

功能和上述的cmpxchg類似，只是處理8字節位數據而已，指令格式中的dst是8字節的內存，和cmpxchg指令中的dst類似，指令中沒有src，是因為src默認是ecx:ebx組合的8字節寄存器對，比較的是edx:eax組合的8字節寄存器對。即dst和edx:eax比較，相等的話，就把ecx:ebx（即src）傳送到dst，不等的話就把dst傳送給edx:eax。有了cmpxchg的實例，這個很容易理解，不需再舉例。

六、堆棧

1、堆棧簡介

堆（heap）和棧（stack）本來是兩種不同的數據結構，而我們所說的堆棧，其實就是棧（stack），是一種串行形式的數據結構，通過在內存中指定一段存儲空間來存放這樣的數據結構：該結構如下面圖示所示，只能從該串行結構的一端存入或者取出數據。



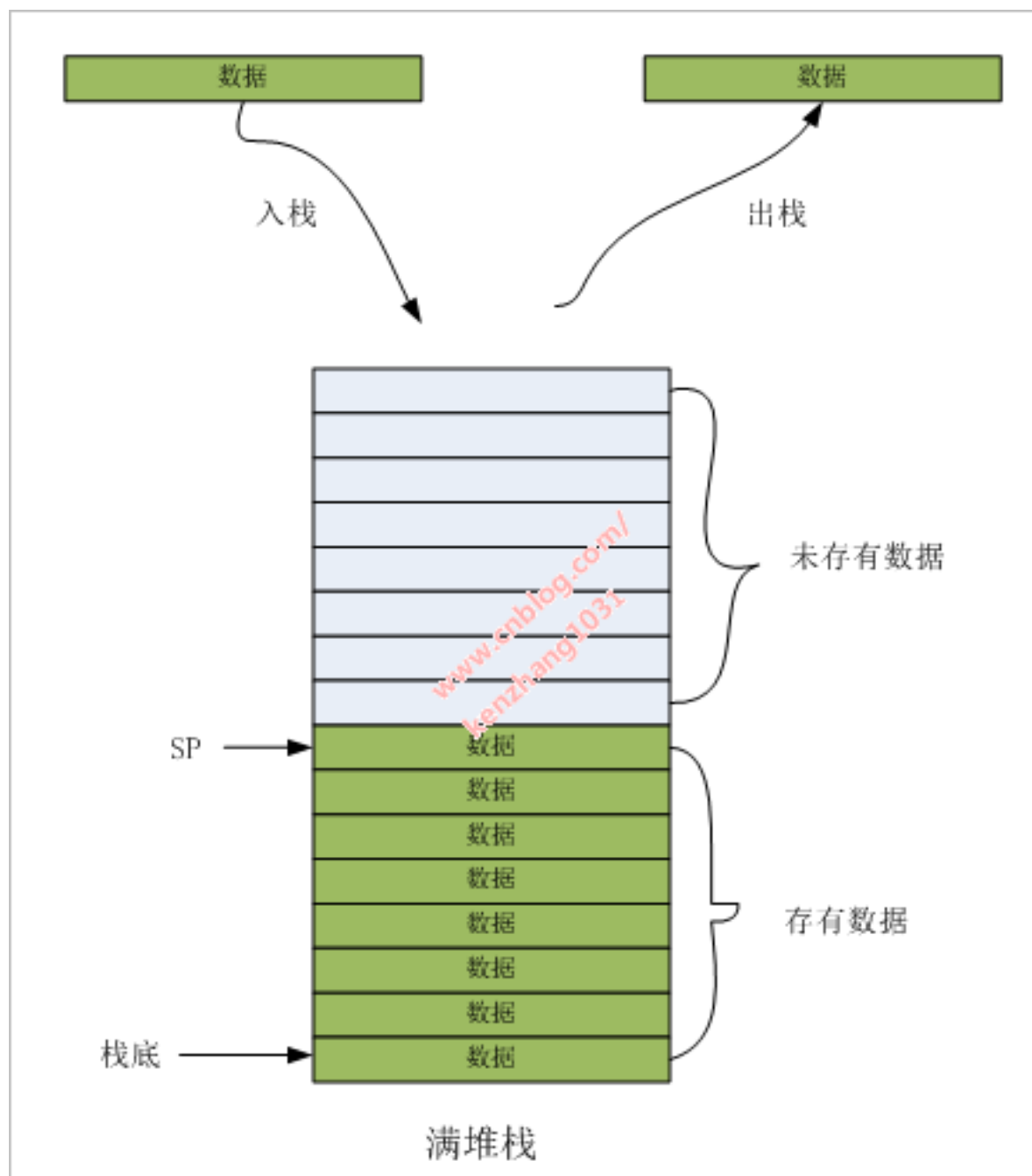
如圖示，上面開口的容器就是棧，最下面的叫棧底，數據進出的那一端叫棧頂，數據從棧頂存入堆棧，如果數據一直存入而不取出，就會裝滿容器，造成棧溢出，反之如果只取不存入，就會造成棧空。故通常，堆棧在使用上是有存有取的，而且壓入棧和彈出棧只能按順序從棧頂存取，即不能存到容器的中間或者底部（該說法僅針對push的壓棧和pop的彈出方式，手動操作堆棧的方式不受這個限制），取數據時也一樣，顯而易見，最後存入的數據會被最先取走，這個叫後進先出（Last In First Out，簡稱LIFO）。

大多數CPU都有用作堆棧指針（stack pointer）的寄存器，簡稱SP，比如原來8086的16位CPU結構上叫SP寄存器，IA-32結構CPU擴展到32位，叫ESP（extended stack pointer）寄存器，它的低16位還是SP寄存器。sp寄存器指向堆棧活動的那一端，即棧頂。這裡要特別說明，不同的CPU結構，SP操作堆棧的方式有些區別，根據這些區別，堆棧有滿堆棧和空堆棧的區別，有遞增型和遞減型的區別。

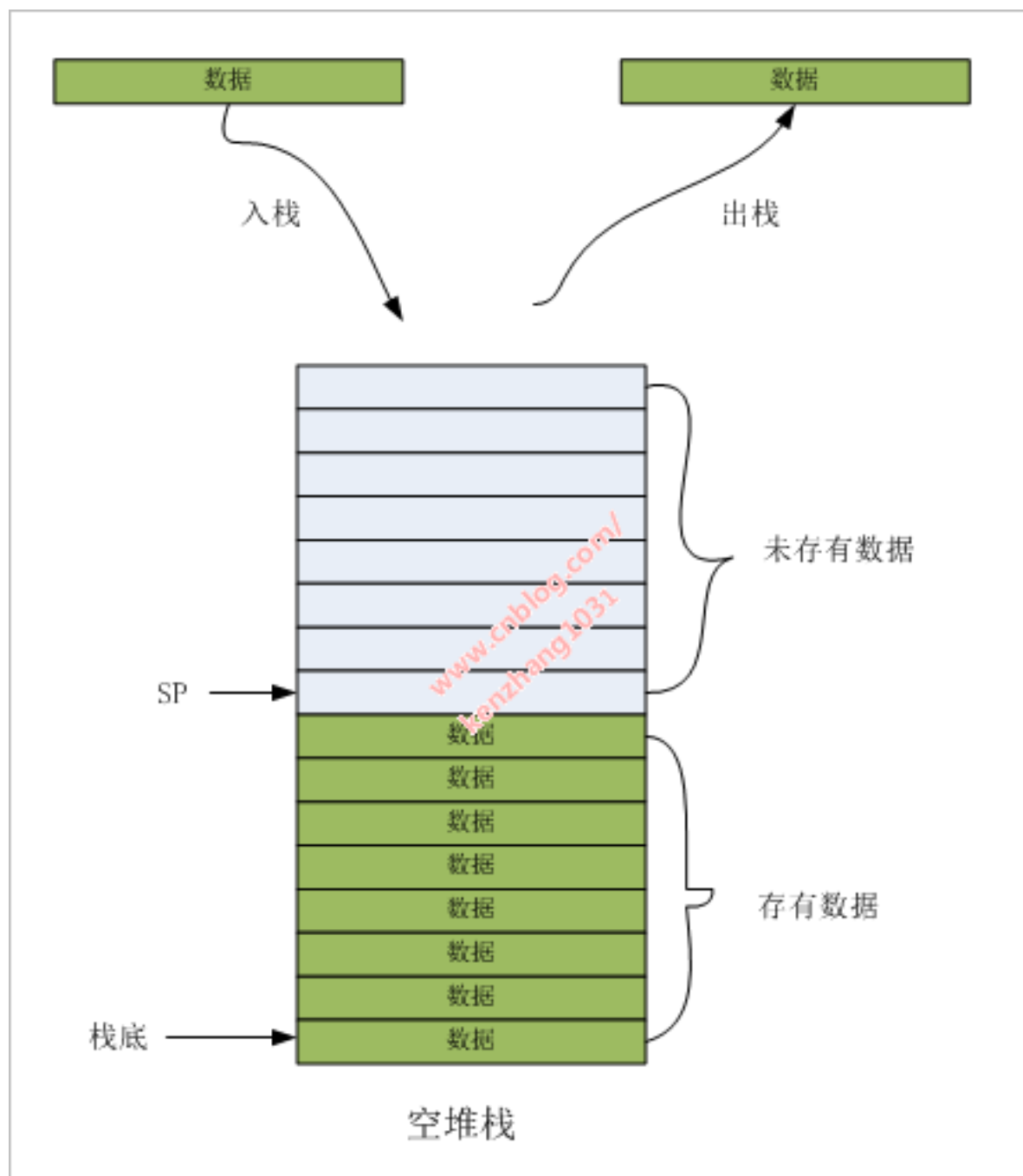
遞增型堆棧：堆棧由內存低地址向高地址方向生長，也叫向上生長型堆棧，即上面的圖示中的棧底在內存的低地址，棧頂在內存的高地址。

遞減型堆棧：堆棧由內存高地址向低地址方向生長，也叫向下生長型堆棧，即圖示中的棧底在內存的高地址，棧頂在內存的低地址。

滿堆棧：如果SP指針指向的是棧頂的最後一個數據的位置，就叫滿堆棧。滿堆棧入棧時，sp要先指向下一個空位置（對遞增型堆棧而言，該空位置就是下一個高地址，對遞減型則是下一個低地址），然後數據再入棧，出棧時，數據先出棧，然後sp再指向新的棧頂的數據。即入棧和出棧後，sp最終都指向棧頂的最後一個數據的位置。如下圖示：



空堆栈： 如果SP指向的是栈顶的空位置，就叫空堆栈。空堆栈入栈时，数据先入栈，然后sp指向下一个新的空位置（对递增型堆栈而言，该空位置就是下一个高地址，对递减型则是下一个低地址），出栈时，sp要先指向栈顶的最后一个数据，然后数据再出栈，出栈后，sp所指的位置就成了栈顶的新的空位置。即入栈和出栈后，sp最终都指向栈顶的空位置，如下图示：



IA-32結構的CPU採用的是遞減型滿堆棧，舉例來說，比如起始時ESP指向地址0xffffd2c0的位置，如果整數（4字節）0x12345678要入棧，那麼ESP先變成0xffffd2bc，然後4字節的數據再進入到地址0xffffd2bf-0xffffd2bc的4字節的堆棧中，此時ESP指向的是最後入棧的整數0x12345678，出棧時數據先出棧，然後ESP變成0xffffd2c0。

堆棧的操作通常有兩種方式，一種是手動操作ESP寄存器或者EBP寄存器，通過寄存器的間接尋址或者相對尋址的方式來操作堆棧，比如下面的代碼手動操作堆棧，把數據0x12345678入棧：

```
subl $4, %esp
```

```
movl $0x12345678, (%esp)
```

而另一種方式是使用彙編語言提供的push指令入棧，pop指令出棧，比如下面的代碼也是把0x12345678入棧：

```
pushl $0x12345678
```

堆棧是怎麼來的？程序中直接使用push和pop來操作堆棧，完全沒有看到有建立堆棧的相關代碼。如果做過單片機或者像arm之類的程序的話，應該知道，裸機程序最先運行的是一個彙編啟動程序，或者說加載操作系統前是先跑bootloader程序，通常這樣的程序在跳轉到main函數執行前會先初始化堆棧。但是這裡彙編程序設計是在IA-32結構中，是在linux系統中跑的，可以理解為linux下的彙編程序所使用的堆棧是linux操作系統建立的，畢竟這裡的彙編程序不是在裸機上跑的程序，而是在系統中的一個應用程序，因此不需要像裸機程序一樣，程序開始要先初始化一段內存區作為堆棧。

堆棧用途廣泛，比如函數中的局部變量，32位系統的參數傳遞，異常或者中斷以及任務切換的現場保護，或者上下文保護，用的都是堆棧。

2、入棧指令push

push是指把數據壓入堆棧。

指令格式為：**pushx src**

x可以表示l或者w，分別代表壓入32位或者16位的數據，位數必須和後面的src匹配，src可以是寄存器，內存，立即數。

例如以下的指令都是push的合法指令：

```
1: pushl $0x12345678          # 32-bits immediate
```

```
2: pushw $0xabcd             # 16-bits immediate
```

3: pushl %eax # reg32

4: pushw %ax # reg16

5: pushl lab1 # mem32

6: pushw lab2 # mem16

3、出棧指令pop

和push類似，pop的指令格式為：**popx dst**

x可以表示l或者w，分別代表壓入32位或者16位的數據，dst表示接收數據的目的位置，位數要和指令碼對應，可以是寄存器或者內存，不可以是立即數。

例如下面的指令都是pop的合法指令：

1: popl %eax

2: popw %ax

3: popl mem32

4: popw mem16

4、所有寄存器入棧出棧

有4種指令格式，列表如下：

指令	描述
pusha/popa	壓入/彈出所有16位的寄存器，入棧順序為：DI，SI，BP，BX，DX，CX，AX；出棧順序則反過來。

pushad/popad	壓入/彈出所有32位的寄存器，入棧順序為：EDI, ESI, EBP, EBX, EDX, ECX, EAX；出棧順序則反過來。
pushf/popf	壓入/彈出flags寄存器
pushfd/popfd	壓入/彈出eflags寄存器

以上指令中的a，是all（全部）的縮寫；d可能是double，雙倍的意思，即16位變成了32位；f表示flags寄存器，fd表示後面加double後為32位，表示eflags寄存器。



注意：

根據處理器的操作模式，**POPF/POPFD** 指令對 **EFLAGS** 寄存器的影響略有差異。處理器在保護模式中操作時，如果特權級別為 **0**（或是在實地址模式中操作，它相當於特權級別 **0**），則 **EFLAGS** 寄存器中除 **VIP**、**VIF** 以及 **VM** 標誌之外的所有非保留位都可以修改。**VIP** 與 **VIF** 標誌被清除，**VM** 標誌不受影響。

在保護模式中操作時，如果特權級別大於 **0** 但小於或等於 **IOPL**，則除 **IOPL** 字段與 **VIP**、**VIF** 以及 **VM** 標誌之外的所有標誌都可以修改。這裡，**IOPL** 標誌不受影響，**VIP** 與 **VIF** 標誌被清除，**VM** 標誌不受影響。只有在至少與 **IOPL** 相等的特權級別下執行時，才可以更改中斷標誌（**IF**）。如果在特權不夠高的情況下執行 **POPF/POPFD** 指令，則不會發生異常，但特權位也不會改變。

在虛 **8086** 模式中操作時，**I/O** 特權級別（**IOPL**）必須等於 **3** 才能使用 **POPF/POPFD** 指令，此時 **VM**、**RF**、**IOPL**、**VIP** 以及 **VIF** 標誌不受影響。如果 **IOPL** 小於 **3**，**POPF/POPFD** 指令將導致一般保護性異常（**#GP**）。

具體可以參考intel的手冊。

5、堆棧的另一種用法

即上面堆棧簡介中提到的手動操作堆棧的方法，通常在函數中，會安排一段堆棧空間給函數使用，包括存儲函數的局部變量或者臨時空間，一般使用**ESP**作為函數中堆棧棧頂的指針，**EBP**作為棧底的指針，通過**EBP**的間接尋址或者相對尋址來操作堆棧，存儲臨時變量或者局部變量，此時**ebp**是不變的，存儲的是要恢復的**esp**的值，調用子函數則通過**ESP**入棧，比如類似以下的代碼的用法，仔細閱讀註釋應該不難理解。

```
1: func:                # func函數
```

```
2:  pushl %ebp          # ebp保存到堆棧

3:  movl %esp, %ebp     # ebp為func函數所用堆棧的棧底

4:  subl $24, %esp      # esp為func函數所用堆棧的棧頂，即開闢了24個字節給func函數使用

5:  ...                 # func處理

6:  movl local_var, -4(%ebp) # 通過ebp相對尋址來操作func的堆棧底部來保存局部或臨時變量，此時ebp是不變的，

7:  movl temp_var, -8(%ebp) # ebp保持著func函數所用堆棧底的值，即是函數返回後需恢復的堆棧指針值

8:  ...                 # func處理

9:  pushl param         # 參數入棧,使用的是func堆棧的頂部

10: call other_func     # 調用別的函數

11: ...                 # func處理

12:

13: movl %ebp, %esp     # 恢復esp堆棧指針

14: popl %ebp           # 恢復ebp寄存器

15: ret                 # 函數返回
```

七、優化內存訪問

內存的訪問比寄存器慢很多，往往造成CPU執行慢。因此，要優化程序的處理速度，按以下幾點：

- 儘量使用寄存器，而不用內存，
- 如果需要使用內存，把使用頻繁的數據存放到連續的內存塊，因為CPU訪問內存時，會把一塊內存讀到緩存中，緩存比內存快多了，這麼做可以提高緩存的命中率，也就提高了速度
- 內存對齊，因為CPU讀取內存是按照一定的粒度去讀取的，比如先讀取地址0-3的4個字節，下一次可能讀取地址4-7的4字節，因此，如果數據在內存中跨邊界，比如用了地址2-5，則要讀取兩次才可取到數據，如果對齊邊界存放，則能一次讀回來，因此建議是n字節的數據對齊到n的倍數的基址上存放，比如32位數據（4字節）對齊到4的倍數的基址上。彙編語言提供了.align指令用於對齊
- 避免小數據傳輸，比如要複製內存的數據到另一塊內存，對IA-32而言，一次複製4字節肯定比按字節去複製要快
- 避免在堆棧中使用大的數據長度（比如80位和128位的浮點數）



注意：

gas中，指令**.align n**對不同的cpu結構意義不一樣，對於a29k，HPPA，M68K，m88k，W65，SPARC，Xtensa和瑞薩/的SuperH SH，以及i386的ELF格式這樣的結構而言，表示對齊到n的倍數的基址上，比如**.align 4**表示對齊到地址為4的倍數的基址上；而對於其他系統，包括使用a.out格式的i386，ARM和StrongARM的結構，表示對齊到2的n次方的倍數的基址上，比如**.align 3**表示對齊到8的倍數的基址上。