沒想到《Linux　Debugging:使用反彙編理解C++程序函數調用棧》發表了收到了大家的歡迎。但是有網友留言說不熟悉彙編，因此本書列了彙編的基礎語法。這些對於我們平時的調試應該是夠用了。

# 1 AT&T與Intel彙編語法對比

本科時候大家學的基本上都是Intel的8086彙編語言，微軟採用的就是這種格式的彙編。GCC採用的是AT&T的彙編格式, 也叫GAS格式(Gnu ASembler GNU彙編器)。

1、寄存器命名不同

| AT&T | Intel | 說明 |
| --- | --- | --- |
| %eax | eax | Intel的不帶百分號 |

2、操作數順序不同

| AT&T | Intel | 說明 |
| --- | --- | --- |
| movl %eax, %ebx | mov ebx, eax | Intel的目的操作數在前,源操作數在後；AT&T相反 |

3、常數/立即數的格式不同

| AT&T | Intel | 說明 |
| --- | --- | --- |
| movl $_value,%ebx | mov eax,_value | Intel的立即數前面不帶$符號 |
| movl $0xd000d,%ebx | mov ebx,0xd000d | 規則同樣適用於16進制的立即數 |

4、操作數長度標識

| AT&T | Intel | 說明 |
| --- | --- | --- |
| movw %ax,%bx | mov bx,ax | Intel的彙編中,操作數的長度並不通過指令符號來標識。<br>AT&T的格式中,每個操作都有一個字符後綴,表明操作數的大小.例如:mov指令有三種形式:<br>movb傳送字節<br>movw傳送字<br>movl傳送雙字<br>如果沒有指定操作數長度的話，編譯器將按照目標操作數的長度來設置。比如指令“mov %ax, %bx”， |

由於目標操作數bx的長度為word，那麼編譯器將把此指令等同於“movw %ax, %bx”。

5、尋址方式

| AT&T | Intel | 説明 |
| --- | --- | --- |
| imm32(basepointer, indexpointer, indexscale) | [basepointer + indexpointer*indexscale + imm32) | 兩種尋址的實際結果都應該是<br>imm32 + basepointer + indexpointer*indexscale |

例如: 下面是一些尋址的例子：

| AT&T | Intel | 説明 |
| --- | --- | --- |
| mov 4(%ebp), %eax | mov eax, [ebp + 4] | 基址尋址（Base Pointer Addressing Mode）,用於訪問結構體成員比較方便，例如一個結構體的基地址保存在eax寄存器中，其中一個成員在結構體內的偏移量是4字節，要把這個成員讀上來就可以用這條指令 |
| data_items(,%edi,4) | [data_items+edi*4 | 變址尋址（Indexed Addressing Mode），訪問數組 |
| movl $addr, %eax | mov eax, addr | 直接尋址（Direct Addressing Mode） |
| movl (%eax), %ebx | mov ebx, [eax] | 間接尋址（Indirect Addressing Mode）,把eax寄存器的值看作地址，把內存中這個地址處的32位數傳送到ebx寄存器 |
| mov $12, %eax | mov eax, 12 | 立即數尋址（Immediate Mode） |
| mov $12, %eax | mov eax, 12 | 寄存器尋址（Register Addressing Mode |

6.跳轉方式不同

AT&T 彙編格式中，絕對轉移和調用指令（jump/call）的操作數前要加上'*'作為前綴，而在Intel 格式中則不需要。

| AT&T | Intel | 説明 |
| --- | --- | --- |
| jmp *%eax | jmp %eax | 用寄存器%eax中的值作為跳轉目標 |
| jmp *(%eax) | jmp (%eax) | 以%eax中的值作為讀入的地址, 從存儲器中讀出跳轉目標 |

# 2 求一個數組最大數

**通過求一個數組的最大數，來進一步學習AT&T的語法**

**[cpp]**
view plain copy

```
1.  #PURPOSE: This program finds the maximum number of a
2.  # set of data items.
3.  #
4.  #VARIABLES: The registers have the following uses:
5.  #
6.  # %edi - Holds the index of the data item being examined
7.  # %ebx - Largest data item found
8.  # %eax - Current data item
9.  #
10. # The following memory locations are used:
11. #
12. # data_items - contains the item data. A 0 is used
13. # to terminate the data
14. #
15.  .section .data #全局變量
16. data_items: #These are the data items
17.  . long  3,67,34,222,45,75,54,34,44,33,22,11,66,0
18.
19.  .section .text
20.  .globl _start
21. _start:
22.  movl $0, %edi # move 0 into the index  register
23.  movl data_items(,%edi,4), %eax # load the first byte of data
24.  movl %eax, %ebx # since  this  is the first item, %eax is
25.            # the biggest
26.
27. start_loop: # start loop
28.  cmpl $0, %eax # check to see  if  we've hit the end
29.  je loop_exit
```

```
30.  incl %edi # load next value
31.  movl data_items(,%edi,4), %eax
32.  cmpl %ebx, %eax # compare values
33.  jle start_loop # jump to loop beginning  if  the  new
34.            # one isn't bigger
35.  movl %eax, %ebx # move the value as the largest
36.  jmp start_loop # jump to loop beginning
37.
38. loop_exit:
39.  # %ebx is the status code for the _exit system call
40.  # and it already has the maximum number
41.  movl $1, %eax #1 is the _exit() syscall
42.  int  $0×80
```

彙編程序中以.開頭的名稱並不是指令的助記符，不會被翻譯成機器指令，而是給彙編器一些特殊指示，稱為彙編指示（Assembler Directive）或偽操作（Pseudo-operation），由於它不是真正的指令所以加個“ 偽 ”字。.section指示把代碼劃分成若干個段（Section），程序被操作系統加載執行時，每個段被加載到不同的地址，操作系統對不同的頁面設置不同的讀、寫、執行權限。.data段保存程序的數據，是可讀可寫的，相當於C++程序的全局變量。

.text段保存代碼，是只讀和可執行的，後面那些指令都屬於.text段。

.long指示聲明一組數，每個數佔32；.quad類似，佔64位；.byte是8位；.word是16位。.ascii，例如.ascii "Hello world"，聲明11個數，取值為相應字符的ASCII碼。

參考資料：


1.

最簡單的彙編程序

2.

第二個彙編程序

3. http://blog.chinaunix.net/uid-27717694-id-3942757.html


最後復習一下lea命令：

mov 4(%ebp) %eax #將%ebp+4地址處所存的值，mov到%eax

leal 4(%ebp) %eax #將%ebp+4的地址值， mov到%eax

leal 可以被mov取代：

addl $4, %ebp

mov. %ebp, %eax