AT&T汇编语法

Ted posted @ 2008年9月01日 19:17 in asm with tags ASM , 3980 阅读

0.3.1 Overview

开 发一个OS，尽管绝大部分代码只需要用C/C++等高级语言就可以了，但至少和硬件相关部分的代码需要使用汇编语言，另外，由于启动部分的代码有大小限 制，使用精练的汇编可以缩小目标代码的Size。另外，对于某些需要被经常调用的代码，使用汇编来写可以提高性能。所以我们必须了解汇编语言，即使你有可 能并不喜欢它。

如果你是计算机专业的话，在大学里你应该学习过Intel格式的8086/80386汇编，这里就不再讨论。如果我们选择的OS开发工具是GCC以及GAS的话，就必须了解AT&T汇编语言语法，因为GCC/GAS只支持这种汇编语法。

本书不会去讨论8086/80386的汇编编程，这类的书籍很多，你可以参考它们。这里只会讨论AT&T的汇编语法，以及GCC的内嵌汇编语法。

0.3.2 Syntax

1.寄存器引用

引用寄存器要在寄存器号前加百分号%,如“movl %eax, %ebx”。

80386有如下寄存器：

- 8个32-bit寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp；
- 8个16-bit寄存器，它们事实上是上面8个32-bit寄存器的低16位：%ax，%bx，%cx，%dx，%di，%si，%bp，%sp；
- 8个8-bit寄存器：%ah，%al，%bh，%bl，%ch，%cl，%dh，%dl。它们事实上是寄存器%ax，%bx，%cx，%dx的高8位和低8位；
- 6个段寄存器：%cs(code)，%ds(data)，%ss(stack), %es，%fs，%gs；
- 3个控制寄存器：%cr0，%cr2，%cr3；
- 6个debug寄存器：%db0，%db1，%db2，%db3，%db6，%db7；
- 2个测试寄存器：%tr6，%tr7；
- 8个浮点寄存器栈：%st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)。

2. 操作数顺序

操作数排列是从源（左）到目的（右），如“movl %eax(源)，%ebx(目的)”

3. 立即数

使用立即数，要在数前面加符号$, 如“movl $0x04, %ebx”

或者：

para = 0x04

movl $para, %ebx

指令执行的结果是将立即数04h装入寄存器ebx。

4. 符号常数

符号常数直接引用 如

value: .long 0x12a3f2de

movl value , %ebx

指令执行的结果是将常数0x12a3f2de装入寄存器ebx。

引用符号地址在符号前加符号$, 如“movl $value, % ebx”则是将符号value的地址装入寄存器ebx。

5. 操作数的长度

操作数的长度用加在指令后的符号表示b(byte, 8-bit), w(word, 16-bits), l(long, 32-bits)，如“movb %al, %bl”，“movw %ax, %bx”，“movl %eax, %ebx ”。

如果没有指定操作数长度的话，编译器将按照目标操作数的长度来设置。比如指令 “mov %ax, %bx”，由于目标操作数bx的长度为word，那么编译器将把此指令等同于“movw %ax, %bx”。同样道理，指令“mov $4, %ebx”等同于指令“movl $4, %ebx”，“push %al”等同于“pushb %al”。对于没有指定操作数长度，但编译器又无法猜测的指令，编译器将会报错，比如指令“push $4”。

6. 符号扩展和零扩展指令

绝大多数面向80386的AT&T汇编指令与Intel格式的汇编指令都是相同的，符号扩展指令和零扩展指令则是仅有的不同格式指令。

符号扩展指令和零扩展指令需要指定源操作数长度和目的操作数长度，即使在某些指令中这些操作数是隐含的。

在AT&T语法中，符号扩展和零扩展指令的格式为，基本部分 "movs"和"movz"（对应Intel语法的movsx和movzx），后面跟上源操作数长度和目的操作数长度。movsbl意味着movs （from）byte （to）long；movbw意味着movs （from）byte （to）word；movswl意味着movs （from）word （to）long。对于movz指令也一样。比如指令“movsbl %al, %edx”意味着将al寄存器的内容进行符号扩展后放置到edx寄存器中。

其它的Intel格式的符号扩展指令还有：

- cbw -- sign-extend byte in %al to word in %ax；
- cwde -- sign-extend word in %ax to long in %eax；
- cwd -- sign-extend word in %ax to long in %dx:%ax；
- cdq -- sign-extend dword in %eax to quad in %edx:%eax；

对应的AT&T语法的指令为cbtw，cwtl，cwtd，cltd。

7. 调用和跳转指令

段内调用和跳转指令为"call"，"ret"和"jmp"，段间调用和跳转指令为"lcall"，"lret"和"ljmp"。

段间调用和跳转指令的格式为“lcall/ljmp $SECTION, $OFFSET”，而段间返回指令则为“lret $STACK-ADJUST”。

8. 前缀

操作码前缀被用在下列的情况：

- 字符串重复操作指令(rep,repne)；
- 指定被操作的段(cs,ds,ss,es,fs,gs)；
- 进行总线加锁(lock)；
- 指定地址和操作的大小(data16,addr16)；

在AT&T汇编语法中，操作码前缀通常被单独放在一行，后面不跟任何操作数。例如，对于重复scas指令，其写法为：

```
repne
scas
```

上述操作码前缀的意义和用法如下：

- 指定被操作的段前缀为cs,ds,ss,es,fs,和gs。在AT&T语法中，只需要按照section:memory-operand的格式就指定了相应的段前缀。比如：lcall %cs:realmode_swtch
- 操作数 / 地址大小前缀是“data16”和"addr16"，它们被用来在32-bit操作数 / 地址代码中指定16-bit的操作数 / 地址。

- 总 线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅 仅 对 ADD，ADC，AND，BTC，BTR，BTS，CMPXCHG,DEC，INC，NEG，NOT，OR，SBB，SUB，XOR，XADD,XCHG指令有效，如果将Lock前缀用在其它指令之前，将会引起异常。
- 字符串重复操作前缀"rep","repe","repne"用来让字符串操作重复“%ecx”次。

9. 内存引用

Intel语法的间接内存引用的格式为：

section:[base+index*scale+displacement]

而在AT&T语法中对应的形式为：

section:displacement(base,index,scale)

其中，base和index是任意的32-bit base和index寄存器。scale可以取值1，2，4，8。如果不指定scale值，则默认值为1。section可以指定任意的段寄存器作为段前 缀，默认的段寄存器在不同的情况下不一样。如果你在指令中指定了默认的段前缀，则编译器在目标代码中不会产生此段前缀代码。

下面是一些例子：

-4(%ebp)：base=%ebp，displacement=-4，section没有指定，由于base = %ebp，所以默认的section=%ss，index,scale没有指定，则index为0。

foo(,%eax,4)：index=%eax，scale=4，displacement=foo。其它域没有指定。这里默认的section=%ds。

foo(,1)：这个表达式引用的是指针foo指向的地址所存放的值。注意这个表达式中没有base和index，并且只有一个逗号，这是一种异常语法，但却合法。

%gs:foo：这个表达式引用的是放置于%gs段里变量foo的值。

如果call和jump操作在操作数前指定前缀“*”，则表示是一个绝对地址调用/跳转，也就是说jmp/call指令指定的是一个绝对地址。如果没有指定"*"，则操作数是一个相对地址。

任何指令如果其操作数是一个内存操作，则指令必须指定它的操作尺寸(byte,word,long），也就是说必须带有指令后缀(b,w,l)。