AT&T的汇编语法，以及GCC的内嵌汇编语法。

**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”。