AT&T汇编语言

提纲

- AT&T汇编语言
- GCC内嵌汇编

AT&T汇编语言

- 在Linux中，以.S(或.s)为扩展名的文件是包含汇编语言代码的文件。
- 在Linux下有两种方式对AT&T汇编进行编译链接，一种是使用汇编程序GAS和连接程序LD，一种是使用GCC

AT&T汇编的编译方式

- 使用汇编程序GAS和连接程序LD
  第一步： as sourcecode.s -o objfile.o
  将汇编源文件编译成目标文件
  第二步：ld objfile.o -o execode
  将目标文件链接成可执行文件
- 使用GCC
  gcc -o execode sourcecode.S
  使用GCC编译一步就可以编译成可执行文件

AT&T汇编示例

```
.data
output: .ascii "hello
    world\n"
.text
.globl  _start
_start:
        movl $4, %eax
        movl $1, %ebx
        movl $output, %ecx
        movl $12, %edx
        int $0x80
        movl $1, %eax
        int $0x80
```

这段程序在linux上编译执行后会输出“hello world”。可以看到AT&T与Intel汇编在格式上有着显著的不同

AT&T中的节(Section)

在AT&T的语法中，一个节由.section关键词来标识，当你编写汇编语言程序时，至少需要有以下三种节：

- .data节
  这种节包含程序已初始化的数据，也就是说，包含具有初值的那些变量
- .text节
  这个节包含程序的代码。需要指出的是，该节是只读节

AT&T中的节(Section)

- .bss节
- 这个节包含程序还未初始化的数据，也就是说，包含没有初值的那些变量。当操作系统装入这个程序时将把这些变量都置为0
- 使用.bss比使用.data的优势在于，.bss节在编译后不占用磁盘的空间，这样编译、连接生成的代码的尺寸会比较小。
- 尽管在磁盘上不占空间，但是在可执行文件被读入内存后系统还是会为.bss节分配内存

拥有三个节的AT&T汇编程序示例

```
.data
output: .ascii "hello world\n"
.text
.globl  _start
_start:
        movl $4, %eax
        movl $1, %ebx
        movl $output, %ecx
        movl $12, %edx
        int $0x80
        movl $3, %eax
        movl $1, %ebx
        movl $sentence, %ecx
        movl $30, %edx
        int $0x80
        movl $4, %eax
        movl $30, %edx
        int $0x80
        movl $1, %eax
        int $0x80
.bss
    sentence: .fill 30
```

程序的功能是首先打印“hello world”，然后让用户输入字符然后将输入的字符打印出来

AT&T汇编语言常见指令

- .ascii
  语法：.ascii　“string”….
  ascii 表示零个或多个(用逗号隔开)字符串，并把每个字符串(结尾不自动加“\0”字符)中的字符放在连续的地址单元。于此类似的 .asciz指令定义的字符串会在结尾处自动加 “\0”字符
- .fill
  语法：.fill repeat , size , value
  含义是反复拷贝size个字节，重复repeat次，其中size和value是可选的，默认值分别为1和0

AT&T汇编语言常见指令

- .globl
  语法：.globl symbol
  .globl使得连接程序(ld)能够看到symbol。如果你的局部程序中定义了symbol，那么，与这个局部程序连接的其他局部程序也能存取symbol
- .rept endr
  语法：.rept count
  .....
  .endr
  把.rept指令与.endr指令之间的行重复count次

AT&T汇编语言常见指令

- .space
  语法：.space size , fill
  这个指令保留size个字节的空间，每个字节的值为fill
- .byte/.word/.long
  语法：.byte/.word/.long expressions
  预留1个字节/字/双字，并将这个字节的内容赋值为expression，若是用逗号隔开的多个expression，则为预留多个这样的字节/字/双字，并将它们的内容依次赋值。
- .set
  设定常数，就好像C程序中的#define的作用一样

AT&T 与Intel的汇编语言语法区别

AT&T和Intel汇编语言的语法区别主要体现在操作数前缀、赋值方向、间接寻址语法、操作码的后缀上

- 操作数前缀

Intel语法	AT&T语法
Mov eax,8	movl $8,%eax
Mov ebx,0ffffh	movl $0xffff,%ebx
int 80h	int $0x80

从表中可以看到在AT%T汇编中诸如"%eax"、"%ebx"之类的寄存器名字前都要加上"%"；"$8"、"$0xffff"这样的立即数之前都要加上"$"。

AT&T 与Intel的汇编语言语法区别

- 源/目的操作数顺序

Intel语法	AT&T语法
MOV EAX,8	movl $8,%eax

在Intel语法中，第一个操作数是目的操作数，第二个操作数源操作数。而在AT&T中，第一个数是源操作数，第二个数是目的操作数。

AT&T 与Intel的汇编语言语法区别

- 寻址方式

  Intel的指令格式是segreg：[base+index*scale+disp]，而AT&T的格式是%segreg：disp(base,index,scale)。

Intel语法	AT&T语法
[eax]	(%eax)
[eax + _variable]	_variable(%eax)
[eax*4 + _array]	_array(,%eax,4)
[ebx + eax*8 + _array]	_array(%ebx,%eax,8)

在AT&T中，当立即数用在scale/disp中时，不应当在其前冠以“$”前缀，而且scale,disp不需要加前缀“&”。另外在Intel中基地址使用“[”、“]”，而在AT&T中则使用

AT&T 与Intel的汇编语言语法区别

- 标识长度的操作码前缀

  在AT&T汇编中远程跳转指令和子过程调用指令的操作码使用前缀“l”，分别为ljmp，lcall，与之相应的返回指令伪lret。例如：

Intel语法	AT&T语法
CALL SECTION:OFFSET	lcall $secion:$offset
JMP FAR SECTION:OFFSET	ljmp $secion:$offset
RET FAR STACK_ADJUST	lret $stack_adjust

AT&T 与Intel的汇编语言语法区别

- 标识长度的操作码后缀

在AT&T的操作码后面有时还会有一个后缀，其含义就是指出操作码的大小。“l”表示长整数（32位），“w”表示字（16位），“b”表示字节（8位）。而在Intel的语法中，则要在内存单元操作数的前面加上byte ptr、 word ptr,和dword ptr，“dword”对应“long”。

Intel语法	AT&T语法
Mov al,bl	movb %bl,%al
Mov ax,bx	movw %bx,%ax
Mov eax,ebx	movl %ebx,%eax
Mov eax, dword ptr [ebx]	movl (%ebx),%eax

GCC内嵌汇编

- Linux操作系统内核代码绝大部分使用C语言编写，只有一小部分使用汇编语言编写，例如与特定体系结构相关的代码和对性能影响很大的代码。GCC提供了内嵌汇编的功能，可以在C代码中直接内嵌汇编语言语句，大大方便了程序设计。

基本行内汇编

- 基本行内汇编很容易理解，一般是按照下面的格式：
  asm(“statements”);
- 在“asm”后面有时也会加上“__volatile__”表示编译器不要优化代码，后面的指令保留原样
  __asm__ __volatile__("hlt");

基本行内汇编

- 如果有很多行汇编，则每一行后要加上“\n\t”：

```
asm( "pushl %eax\n\t"
"movl $0,%eax\n\t"
"popl %eax");
```

- 或者我们也可以分成几行来写，如：

```
asm("movl %eax,%ebx");
asm("xorl %ebx,%edx");
asm("movl $0,_booga);
```

扩展的行内汇编

- 在扩展的行内汇编中，可以将C语言表达式指定为汇编指令的操作数，而且不用去管如何将C语言表达式的值读入寄存器，以及如何将计算结果写回C变量，你只要告诉程序中C语言表达式与汇编指令操作数之间的对应关系即可， GCC会自动插入代码完成必要的操作。

扩展的行内汇编

- 使用内嵌汇编，要先编写汇编指令模板，然后将C语言表达式与指令的操作数相关联，并告诉GCC对这些操作有哪些限制条件。例如下面的汇编语句：
  __asm__ __violate__ ("movl %1,%0" : "=r" (result) : "r" (input));
- “movl %1,%0”是指令模板；“%0”和“%1”代表指令的操作数，称为占位符，“=r”代表它之后是输入变量且需用到寄存器，指令模板后面用小括号括起来的是C语言表达式，其中input是输入变量，该指令会完成把input的值复制到result中的操作

扩展的行内汇编

- 若把刚才的内嵌汇编语句改成如下：
  __asm__ __volatile__ ("movl %1,%0" : "=m" (result) : "m" (input));
- 只是把“=r”改成了“=m”，“r”改成了“m”，然而在编译这条改过的语句的时候编译器便会报错，因为“r”代表复制的时候借助了寄存器，而“m”则代表直接从内存复制到内存，这样的操作显然是非法的

扩展的行内汇编的语法

- 内嵌汇编语法如下：
  __asm__(
  汇编语句模板:
  输出部分:
  输入部分:
  破坏描述部分);
- 即格式为asm ("statements" : output_regs : input_regs : clobbered_regs)

扩展的行内汇编的语法

- 扩展行内汇编共分四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。

```
int main(void)
{
    int dest;
    int value=1;
    asm(
    "movl  %1, %0"
    : "=a"(dest)
    : "c" (value)
    : "%ebx");
    printf("%d\n", dest);
    return 0;
}
```

扩展的行内汇编的语法

- 汇编语句模板
  汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n”或“\n\t”分开。指令中的操作数可以使用占位符引用C语言变量，操作数占位符最多10个，名称如下：%0，%1…，%9。指令中使用占位符表示的操作数，总被视为long型（4，个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在%和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h1。

扩展的行内汇编的语法

- 输出部分

  输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和C语言变量组成。每个输出操作数的限定字符串必须包含“=”表示它是一个输出操作数。例如：__asm__ __volatile__ ("pushfl ; popl %0 ; cli":"=g" (x))

  在这里“x”便是最终存放输出结果的C程序变量，而“=g”则是限定字符串，限定字符串表示了对它之后的变量的限制条件

扩展的行内汇编的语法

- 输入部分

  输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符同样也由限定字符串和C语言表达式或者C语言变量组成。例：

  __asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));

扩展的行内汇编的语法

- 限定字符

限定字符便是内嵌汇编中放在引用的C变量之前的字符，它们的作用是指示编译器如何处理其后的C语言变量与指令操作数之间的关系，例如是将变量放在寄存器中还是放在内存中等，常用的如下：

限定字符	描述
a、b、c、d、s、D	具体的一个寄存器
q、r、A	混合的寄存器
m、o、V、p	内存
g、X	寄存器或内存
I、J、N、i、n	立即数
=、+	操作数类型

内嵌汇编示例

- 例1

```
int main(void)
{
    int result = 2;
    int input = 1;
    __asm__ __volatile__ ("addl %1, %0": "=r"(result): "r"(input));
    printf("%d\n", result);
    return 0;
}
```

这段内嵌汇编原本的目的是输出1+2=3的结果，也就是将input变量的值与result变量的值相加之后再存入result中。可以看到在汇编语句模板中的%1与%0分别代表input与result变量，而“=r”与“r”则表示两个变量在汇编中应该对应两个寄存器，“=”表示result是输出变量。然而实际运行后发现结果实际上是2。这是为什么呢？

内嵌汇编示例

- 我们用(objdump -j .text –S 可执行文件名)这样的命令来查看编译生成后的代码发现这段内嵌汇编经GCC翻译后所对应的AT&T汇编是：

```
movl   $0x2,0xfffffffc(%ebp)
movl   $0x1,0xfffffff8(%ebp)
movl    0xfffffff8(%ebp),%eax
addl    %eax,%eax
movl    %eax,0xfffffffc(%ebp)
```

前两句汇编分别是为result和input变量赋值。input 为输入型变量，而且需要放在寄存器中，GCC给它分配的寄存器是%eax，在执行addl之前%eax的内容已经是input的值。读入input后执行addl，显然addl %eax,%eax 的值不对。

内嵌汇编示例

- 之所以会出现以上的结果是因为：
- 使用“r”限制的输入变量，GCC先分配一个寄存器，然后将值读入寄存器，最后用该寄存器替换占位符
- 使用“r”限制的输出变量，GCC会分配一个寄存器，然后用该寄存器替换占位符，但是在使用该寄存器之前并不将变量值先读入寄存器，GCC认为所有输出变量以前的值都没有用处，不读入寄存器，最后GCC插入代码，将寄存器的值写回变量

  因为第二条，这样内嵌汇编指令不能奏效，因为在执行addl之前result的值没有被读入寄存器

内嵌汇编示例

- 修改后的指令如下：

```
int main(void)
{
    int result  = 2;
    int input  = 1;
      __asm____volatile__
 ("addl  %2,%0":"=r"(result):"r"(result),"m"(input));
    printf("%d\n", result);
    return 0;
}
```

这段内嵌汇编所对应的AT&T汇编如下：

```
movl   $0x2,0xfffffffc(%ebp)
movl   $0x1,0xfffffff8(%ebp)
movl    0xfffffffc(%ebp),%eax
addl   0xfffffff8(%ebp),%eax
movl    %eax,0xfffffffc(%ebp)
```

内嵌汇编示例

- 上面的代码应该可以正常工作，因为我们知道%0和%1都和result相关，应该使用同一个寄存器，而且事实上在实际结果中GCC也确实是使用了同一个寄存器eax，所以可以得到正确的结果3。但是为了更保险起见，为了确保%0与%1与同一个寄存器关联我们可以使用如下的方法：

```
int main(void)
{
    int result  = 2;
    int input  = 1;
     __asm__ __volatile__
("addl  %2,%0":"=r"(result):"0"(result),"m"(input));
    printf("%d\n", result);
    return 0;
}
```

在上面的程序中我们使用了占位符“0”表示%0与%1是使用的同一个寄存器，这样就确保了程序的正确性。

内嵌汇编示例

- 例2

```
int main(void)
{
    int count=3;
    int value=1;
    int buf[10];
    asm(
   "cld \n\t"
   "rep \n\t"
   "stosl"
    :
    : "c" (count), "a" (value) , "D" (buf) );
    printf("%d %d %d\n",
     buf[0],buf[1],buf[2]);
}
```

经GCC翻译后所对应的AT&T汇编是：

```
movl    0xfffffff4(%ebp),%ecx
movl    0xfffffff0(%ebp),%eax
lea     0xffffffb8(%ebp),%edi
cld
repz stos %eax,%es:(%edi)
```

在这里count、value和buf是三个输入变量，它们都是C程序中的变量，“c”、“a”和“D”表示这三个输入值分别被存放入寄存器ECX、EAX与EDI；“cld rep stosl”是需要执行的汇编指令；而“%ecx、%edi”表示这两个寄存器在指令中被改变了。这段内嵌汇编要做的就是向buf中写count个value值。

内嵌汇编示例

- 例3

```
int main(void)
{
    int input, output, temp;

    input = 1;
    __asm__ __volatile__ ("movl $0, %%eax;\n\t"
          "movl %%eax, %1;\n\t"
          "movl %2, %%eax;\n\t"
          "movl %%eax, %0;\n\t"
          : "=m" (output), "=m"(temp)
          : "r" (input)
          :"eax");
    printf("%d %d\n", temp,output);
    return 0;
}
```

内嵌汇编示例

- 这段内嵌汇编经由GCC转化成的汇编代码如下：

```
movl   $0x1,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%edx
mov    $0x0,%eax
mov    %eax,0xfffffff4(%ebp)
mov    %edx,%eax
mov    %eax,0xfffffff8(%ebp)
```

可以看到，由于input、output、temp都是程序局部整型数变量，于是它们实际上是存放在堆栈中的，也就是内存中的某个部分。其中output和temp是输出变量，而且“=m”表明它们应该在内存中，input是输入变量，“r”表明它应存放在寄存器中，于是首先把1存入input变量，然后将变量的值复制给了edx寄存器，在这里我们可以看到内嵌汇编中使用了破环描述符“eax”，这是告诉编译器在程序中eax寄存器已被使用，这样编译器为了避免冲突会将输入变量存放在除eax以外别的寄存器中，如像我们最后看到的edx寄存器。