扉页

目的

本文的目的是为了使读者在底层更加理解编程语言的工作机制。通过更深层次的理解，读者通常在高层时能够更高效。学习汇编编程是一个好的方法。其他PC编译语言任然介绍怎样。这本书介绍怎样在80386编程。这个模式支持现代操作系统的功能。

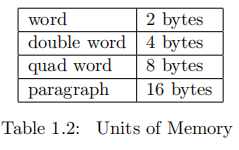
这本书使用，NASM编译器，这些都可以同网上下载。这本书也介绍怎样在Linux操作系统和windows操作系统下编程。如果你想要运行本书中的例子，你必须从我的网页下载。

注意这本书并不打算介绍汇编的所有方面，而是尝试介绍最常用的topic。

1. 介绍
   1. 数字系统
   2. 内存

内存的基本单元是bype，每个byte都有一个地址。

在一个PC结构中，具有下面的名字



补充ASCII的是Unicode，ASCII是一个byte，Unicode将两个byte映射成一个字符。

1.2.2 CPU

CPU直接操作寄存器中的数据。一般情况下，每种类型CPU有自己特殊的机器语言。

一个指令需要的beats的次数依赖CPU生成模型。

1.2.4 8086 16位寄存器

原始的8086 提供16位寄存器。这些寄存器可以分解为。通常AH和AL独立用作1byte的寄存器。

Index 寄存器：SI DI

1. 通常用作指针，也可以用作通用寄存器
2. 不能分成两个8位

BP、SP:

1. 指向stack

CS,DS，SS,ES:指向程序不同段，

IP

FLAGS:存储之前指令的结果，每个结果都是bit

1.2.5 32-bit寄存器

80386有扩展寄存器。例如AX是32位。具有向后兼容。不能直接获取EAX的高16位。

段寄存器还是16位。还有两个寄存器，FG和GS，他们的名字没有意义，他们只是额外的暂时段寄存器，类似ES。

Word 表示CPU数据寄存器的大小。对于80x86家族，现在有点confusing。80386发展之后，他们决定不改变word的意义。

1.2.6 实模式

在实模式，只能访问1M内存。20位数字不能用8086的16位寄存器获取，为了解决这个问题，Intel使用两个16位的值，Selector：offset，selector必须存储在段寄存器中（offset 应该是在IP中）。

然后使用下面的公式来访问20位地址。



效果上，selector就是第几个paragraph。

实模式具有下面的不足：

1. 单个selector只能引用64K内存，但一个程序的如果比较大，则需要多个selector，CS需要切换。
2. 同一个地址可能被多个逻辑地址同时reference。比如物理地址04808可以被引用047C:0048,047D:0038引用

1.2.7 16位保护模式

Selector值与实模式下完全不同，一个selector是GDT的index，在这两个模式下，程序分成段。在实模式下，segments固定在特定的物理位置，保护模式使用虚拟内存计数。

在保护模式下，一个segment有在gdt中有一个描述符。Entry具有segment需要的所有信息，包括位置，获取权限。

16位模式的一个缺点是，offset还是16位，从而，segment的最多64K。

1.2.8 32位保护模式

80386引入，32位保护模式，这与80286的16-位保护模式有两个不同。

1. offset扩展为32位，从而offset能够有4G/
2. Segment 分成4K的page，虚拟内存系统现在以页为单位，而不是segment，这意味着，segment的部分会在内存中。

1.2.9 中断

有事程序的正常流必须被快速响应。计算机的影响提供一个中断机制来处理这些事情，例如，当一个鼠标移动，鼠标中断当前的程序来处理鼠标移动。中断后会进入终端处理器，终端是routines来处理中断。每个类型的终端分配一个整数，在物理内存开始的开始部分。终端号是table的index。

外部中段从CPU外面得到。很多IO设备raise 终端。内部终端是CPU的。Error中断叫做traps，从中断指令的叫做软件中断。DOS使用这些中断来实现API，现在使用基于C的接口。

很多中断将控制在控制结束后将控制返回到程序，然后回复所有的寄存器。Traps一般不会返回，一般会停止程序。

* 1. 汇编语言
     1. 机器语言

每个CPU理解自己的语言，每个指令有自己的操作代码。80x86的指令的大象不一样。Opcode总是在指令的开头，很多指令号可数据。

机器语言很难直接编程。比如

1.3.2 汇编语言

一个每个汇编指令表示一个机器指令。

一个assembler是一个读取汇编语言代码并转化为机器语言的程序。Assembler比编译器简单得多，每个汇编语言指令直接表示一个机器语言。高阶语言更加复杂也需要更多的机器指令。

另外一个汇编语言和高阶语言的另外一个区别是每种类型的CPU有自己的机器语言。每个CPU也有自己的汇编语言。将汇编程序移植到不同的架构比高阶语言更难。

1.3.3 指令操作

机器代码有不同的类型和数量也不一样。通体上，每个指令具有固定的操作数。操作数一般有下面的类型。

寄存器：这些操作符引用CPU的寄存器内容

内存：这些指向内存中的数据。地址都是segment的offset

立即数：这些事固定的数，存储在指令本身中。不在data segment中

Implied：这些没有显式地显示，例如，增加指令，会对寄存器或者内存加1.

1.3.4 基本指令

最基本的指令是MOV，将数据从一个位置移动到另外一个位置。

Src复制到dest中，一个限制是，这两个操作数都不能是内存操作数。这说明了汇编的一个问题。操作数也必须是同样的大小。

1.3.5 指令

（从硬件上想，指令是除了CPU实际执行的用于程序的指令）

一个指令是assembler的artifact而不是CPU的。他们通常要么用来instruct assembler来做一些东西或者告诉assembler一些东西。他们并不翻译成机器代码。指令的常见作用为：

1. 定义常数
2. 定义存储的内存位置
3. 将内存编程segment
4. 条件性包含某些代码
5. 包含其他文件

NASM代码类似C传递。

Equ指令，equ指令可以用来定义symbol。Symbols在汇编程序中使用的常数。



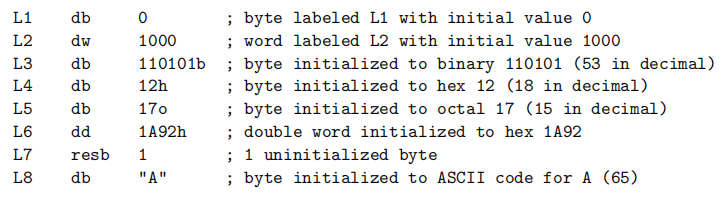
The %指令

这个指令类似C的#define 指令。用法



数据指令

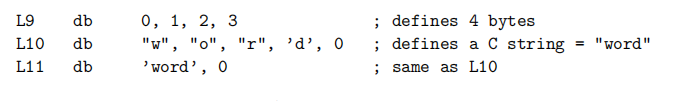
数据指令用在数据段中，来定义内存的空间。有两个方式来reserve内存：1）定义数据的大小，用RES X指令，X用字母代替，来决定对象的大小；2）定义大小和初始值，使用DX指令。例子如下



将内存带有标签很常用，label允许方便引用内存。

双信号和单引号的意义相同，连续的数据定义在内存中按序存放。也就是说，L1存在L2后面。

内存的序列可能被定义。



TIME指令重复操作符特定的次数



Label有两种使用方式：

1. 直接使用，被解释为数据的地址
2. 在[]中，解释为数据内容。

第七行显示NASM的特性。Assembler并不跟踪label的数据的类型。

考虑下面的指令，这个指令产生一个错误，因为assember不知道1是一个byte word韩式double word



应该这样修改。



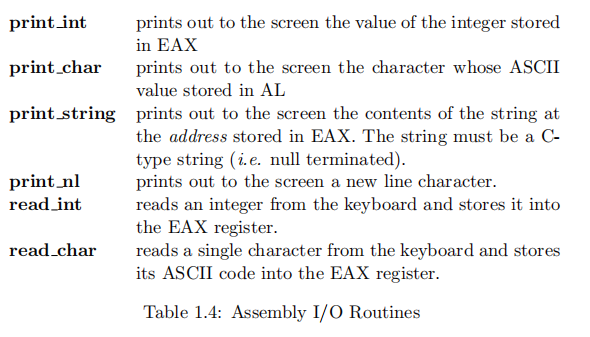
1.3.6 输入和输出

输入和输出依赖于系统活动。这设计与系统的硬件打交道。高阶语言，比如C提供标准库来进行统一的编程接口。汇编语言不提供标准库。他们要么直接获取硬件，要么使用操作系统提供的缔结routine。

汇编器与C互动非常常见。优点是汇编语言可以使用C的IO方法。然而，我们必须知道C语言的传递参数方式。为了简化IO,作者提供了自己的方式来隐藏复杂的C rules并提供更加复杂的接口。所有的routine保持寄存器的值，除了read 寄存器。这些routine确实会修改EAX寄存器的值。为了使用这些routine，需要告诉assember



为了使用其中的print routine，我们用加载正确的值到EAX中，并使用call指令来激活。



* 1. 创建程序

汇编很难移植，今天编程序已经不用汇编了，但是为什么我们还是要学汇编？

1. 有时汇编比较快
2. 汇编能够直接获取系统的硬件特征
3. 更好理解计算机
4. 更好理解高阶语言

实际上， 作者从不用汇编，但是总是使用ideas of 汇编

* + 1. 第一个程序

汇编中可以用C，C中也可以用汇编

代码中需要注意，函数调用前参数放入EAX，从键盘读取也是放入EAX。

未初始化的数据应该在bss段，block started by symbol的简称。

注意代码标签有一个下划线，这是C的calling convention。这个convention生么了C使用的规则。后面会程序完整的convention。现在只需知道所有的C符号都会有下划线。

Global指令告诉汇编器使之成为全局的。不想C中，label带有内部局部性，这意味着只有一个模块中的代码可以使用label。Global指令给标签一个外部范围。

1.4.2 汇编依赖

1.4.6 连接

Bss编译后没有offset，在link之后有offset，占用内存。

大端： 大数字先放。比如00000004 在内存中00 00 00 04（IBM）

小端：小的先放。00000004 在内存中 04 00 00 00 (INTEL)

1. 基本汇编语言
   1. 结合整数
      1. 整数表示

整数有两种方式，signed和unsigned。

无符号型的是最简单的模式，它将帧数表示成两种形式。符号位和值。整数的mag可以用表示，最简单的方式是，正数0，负数1，但是需要令指令能够进行减操作，这使得CPU复杂。

One’s complement

第二种方法叫做One’s complement表示，一个数的One’s complement，通过将数中的bit进行翻转得到。例如，

有一个方法可以不convert bit，就是用F减去正数。

Two’s complement

上面描述的两种方法用于早期的计算器。现代计算器使用two’s complement。现代计算器计算第三个叫做two’s complement的表示。这个two’s complement有两个步骤。

1. 得到one’s complement
2. 结果加1

在这种情况下，可以用加表示减。

证明：y-x = y+0xF-x+1, 其中0xF+1等于0x10，从而等于y-x

当在two’s complement中实现加法，最左边的加法可能会溢出，这个一处没有使用。

使用two’s complement , 可以表示-128到127.

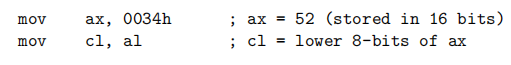
CPU不知道应该表示什么类型，汇编语言不知道高阶语言的类型。FF表示成-1还是+255依赖于高阶语言。

C语言定义符号和无符号类型，这允许C编译器决定使用的正确的数据。

* + 1. 符号扩展

在汇编中，所有的数据都有固定的大小，改变数据大小并不常见，减小数据大小比较简单。

减小数据大小：



增加数据大小

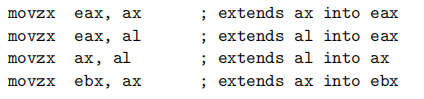
如果将一个字节FF扩展为一个word，那么word的值应该是多少，这依赖于FF是五符号性还是有符号性的。

总体上，扩展一个无符号数字。因此FF变成，然而，为了扩展一个有符号数字，我们必须扩展符号位。这意味着新的位复制符号位。

注意，计算机并不知道一个数字是符号性还是无符号性的，程序需要设定这个转换过程。



然而，使用MOV指令来讲无符号word转化为一个有符号double是不可能的，为什么？因为我们不能声明EAX的高位。80386使用MOVZX来解决这个问题。这个指令有两个操作数，dest必须是16位或者32位，src必须是8位或者16位。Source必须是一个8为或者16位bit寄存器。其他限制是dest必须比src大。

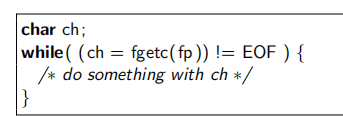


对于符号数字，CWDE指令讲AX扩展成EAX，CDQ指令讲EAX扩展成EDX:EAX（64位），最后MOVSX指令类似MOVZX，不过它使用针对符号数字。

应用到C编程

讲无符号和符号整数进行扩展在C中也有。C中的变量可以直接声明有无符号。第3行使用MOVZX，第四行使用MOVSX。

有一个常见的C语言编程bug，考虑图2.2中的代码。



如果Fgetc返回000000ff，那么会转型成ff，而EOF（-1）在比较时也会转型成FF，从而会出现不想要的错误。

Extendtion 也会出错。

这个问题的解决方法是将ch定义为int，而不是char。

* + 1. two’s complement arithmetic

Carry flag：进位标志，如果加法有进位或者减法有借位，则设置1

Overflow flag:运算结果无法放置在dest

MUL用来乘以unsigned数，imul用来乘以signed数。

乘法操作的具体内容依赖于source的大小，如果source是byte，那么乘以AL，结果存为16位的AX中。如果source是16位，那么乘以AX，结果存到DX:AX中，如果source是32位，那么结果存到EDX:EAX。

imul指令与mul有相同的形式，也增加了两种其他形式

除法操作，

2.1.5 扩展精度数学

汇编语言提供指令，允许我们来实现大于double word的加法与剑法，这些指令使用 carry flag，正如上面所述，加法与减法会修改carry flag。

ADC与SBB指令使用carry flag中的信息。ADC指令实现下面的运算。

* 1. 控制结构

高阶语言提供高阶控制结构

* + 1. 比较语句

控制结构决定基于数据的比较来做什么。在汇编中。

cmp：对于符号操作数，结果影响ZF和CF。

对于无符号操作数，结果影响ZF，OF，SF。

可以通过这些符号来判断对比结果

* + 1. 分支指令

分支指令可以将执行转移到程序的任意位置。换句话说，他们就像goto，有两种类型分支，无条件与条件分支。

Jmp 无条件。

汇编语言与链接器会将label调整为正确的指令地址。

第三章

3.1 位移操作

汇编语言允许程序员处理数据的某一位。比如shift操作。

3.1.1 逻辑位移

逻辑位移是最简单的位移，

注意，出来的位是zero。这些指令允许我们位移多个为。唯一的数量要么是常数要么存储在CL寄存器中。输出的位存储来carry flag中。

3.1.2 shift的使用

快速乘法与除法最常用。

实际上，logical shift可以用来乘以以及除以unsigned值。但是不适合signed值。

3.1.3 数学位移

SAL与SHL完全相同。

SAR：并不位移sign bit。另外一个bit正常的使用，不过incoming 位是sign位。

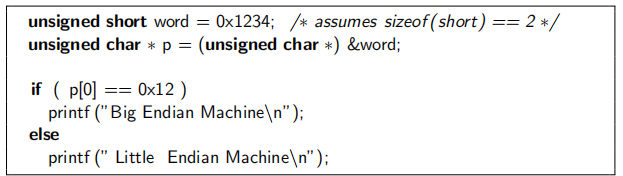
3.2 bool 操作

3.3.2 在C中使用位操作

C中

3.4 大端和小端表示

判断的依据

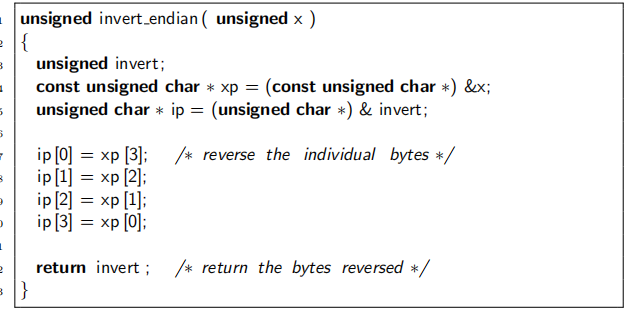


3.4.1 什么时候关心大端与小端？

一般的编程中，大端与小端并不重要，最常见的时候是二进制代码在不同的计算机中传输的时候，比如通过网络或者物理数据媒介。ASCII数据是单个byte，大端与小端并不重要。

所有外部TCP/IP头文件以大端形式存储整数。TCP/IP库提供C函数来解决段。例如，htonl()函数讲double转化，。Ntohl()函数进行反向转换。对于大端系统，这两个函数的输入没有改变，这允许我们来写网络编程，并在任何系统中可以运行。对于更多信息。

下面是C函数讲word转换的方法



80486寄存器提供了bswap指令来调整32位寄存器。这个指令不能用于16位寄存器，然而，xchg指令可以用于swap 16位的寄存器。

3.5 counting bit

上面计算一个double word中的1的数量。这部分看其他方法。

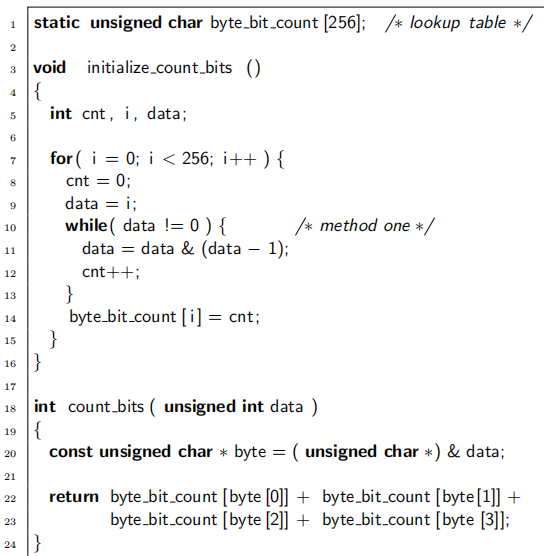
3.5.1 第一个方法

第一个方法很简单。

3.5.2 方法二

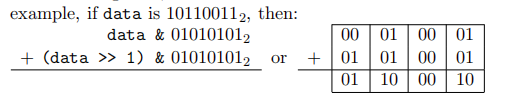
Lookup table来计算。直接的方法是预计算每个然而，但是太多的值。

更加真实的方法是预计算所有byte的位的个数，然后将这些放入数组中。然后double word可以分成四个byte。



3.5.3 方法三

这个方法将数据中1和0相加，结果等于1的数量。例如



右边的加法显示了实际相加的bits。Byte的bits分成四个2-bit域，来显示有四个独立地加法。由于这些加法的最大数是2，那么加法不可能一处，并且影响其他域。

# 第四章 子程序

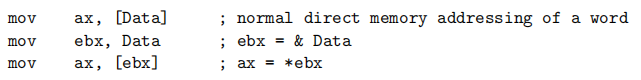
这一章节介绍使用子程序，以及与高阶语言的结构。函数以及流程是高阶子程序的高阶语言例子。

调用子程序的代码以及子程序本身必须在数据传输上达成一致。这些规则叫做calling conventions。这一章的很多部分会解决标准C中的calling convention。这可以用来使汇编和C结合使用。

4.1 间接寻址

间接寻址允许寄存了类似之后着呢。为了说明一个寄存器用于指针，用[]表示。

由于AX有一个word，第三行从



由于AX两个byte，第三行。如果AX被AL代替，那么制度一个byte。需要注意，寄存器并不知道数据的类型。EBX被假定了。

汇编中，一个寄存器中的是当作指针还是数值依赖于操作指令

4.2 简单的子程序例子

如果子程序被程序的不同部分使用，那么必须返回到调用的地方。。

$返回当前行的地址

4.3 stack

Ss寄存器声明stack的segment。数据都是double word。

Push

4.5 callingconventions

当一个子程序invoked，调用代码和子程序必须统一数据的传输，高阶语言有标准的方法。对于高阶语言，汇编语言必须使用与高阶语言相同的calling conventions。Calling conventions对于不同的编译器不一样。一样的是用call调用，然后用ret返回。

4.5.1 在栈上传递参数

参数在call之前push到栈上。如果参数被改变，那么需要传数据的地址。如果参数大小于double word，那么找转换。

子程序不会pop堆栈上的参数，而是从堆栈本身访问它们。

1. 因为他们是先放的
2. 他们会被多次适应，因此不能够放在寄存器中。

考虑把单个参数传递到stack上。当子程序invoke之后，栈看起来想图4.1，参数可以使用间接寻址。

如果栈子程序内存储数据，ESP会改变（所有stack都在一个地方）。用EBP可以解决这个问题。这样子程序的参数可以在EBP+8的任何位置而无需担心什么放到栈上面了。

在子程序结束之后，push到stack上的参数必须被删除，C的calling convention是调用代码必须完成这个，其他的convention并不一下。例如，Pascal 声明子程序必须去除参数。有其他类型的ret指令，能够使得这个见到那。一些C也支持。

图4.5声明了使用C calling convention是怎么被调用的。第三行将参数从栈中去除。也可以使用POP指令来完成，但是者需要结果从年初到寄存器中。实际上，对于这个例子，很多会使用POP ECX指令来去除参数。编译器会使用POPO而不是ADD因为ADD需要更多的bytes。然而，POP也改变ECX的值，下面是另外一个例子，两个子程序使用上面讨论的C calling convention。第54行说明了多个数据以及text segment可在单个源文件中声明。他么也可合成单个数据或者text segment。将数据和代码分成separate segment使得子程序使用的数据被子程序定义，

4.5.2 在栈上的局部变量

局部变量存储在EBP后面。当前的EBP用来获取局部变量。

艾玛通过ESP减去需要的byte得到。图4.6显示了新的子程序框架。EBP寄存器用来获取局部变量，考虑4.7中的C函数，图4.8显示了等价的子程序。

图4.9显示了栈的样子，栈的这个部分包含参数，返回信息以及局部变量存储，叫做stackframe。每个C函数的invocation 都会创建一个新的stack frame。

子程序可以使用两个特殊的指令来简化。ENTER指令以及LEAVEENTER的操作数被局部变量需要。LEAVE指令没有操作数。图4.10显示这些指令是怎么被使用的。

4.6 多模块程序

一个多模块程序包含多个obj文件。为了使A中的模块使用B中的label，必须使用extern指令。这个指令告诉汇编器这些label可以被使用（虽然汇编的时候还没有，连接之后才有）。

在汇编中，label默认情况下不能被外界使用，如果label可以在其他模块中使用，那么在当前模块必须使用global声明。Global做下面内容。

图1.7指令将\_asm\_main声明为global，如果没有这个声明，那么会出现链接错误，为什么？因为C并不能够参考内部\_asm\_main变迁。

接下来是之前的例子，重写来使用两个模块。这两个子程序在独立的源文件中。

4.7 C与汇编的接口

现在，很少程序用会编写。编译器能够将高级语言转化为汇编语言。因为用高级语言写代码更加简单。

当使用汇编，通常只是用小部分代码。通常者有两种方式：从C调用汇编inline代码。这种方式，程序员可以将汇编直接放到C中，汇编代码必须用编译器使用的方式写。

调用一个汇编子代码在PC上比较标准。汇编代码通常与C结合，是因为下面的原因：

1. 用C完成不了
2. 快速

汇编会有移植性差的致命缺点，第二个原因不再成立。

大部分C calling convention已经被声明了。然而，还有一些其他需要的功能。

4.7.1 保存寄存器

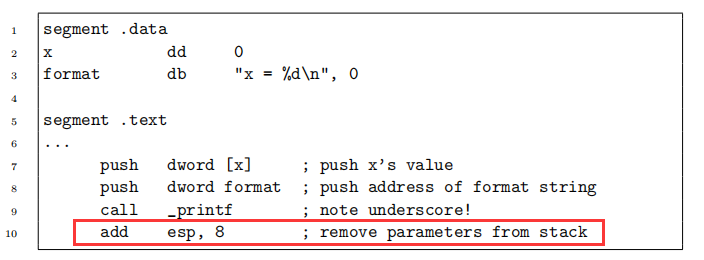
首先，C假设一个子程序保留下面的寄存器。这些并不意味着子程序不能在内部操作，相反，这意味着，如果这不改变他们的值，那么必须在原始程序返回前恢复。EBX，ESI和EDI值必须没有被修改。通常，栈用来保留这些寄存器的原始值。

4.7.2 函数的label

大部分编译器会在函数名字以及全局或者静态变量前加一个下划线，例如一个f的函数的label会是\_f。Linux编译器并不预置任何字符，在linux ELF可执行文件下，我们对C函数f只需使用f label。然而，。

4.7.3 传递参数

在c中，参数从右往左push



4.7.4 计算局部变量的地址

寻找data和bss中变量的地址比较简单，通常链接器会做这个事，然而，计算局部变量的地址并不直观。考虑传递一个变量的地址，如果X在EBP-8，我们不能够直接使用：



为甚，使用MOV将内容存储到EAX的数必须被编译器计算，也就是说，最后必须是常数。不过，有一个指令，能够执行，加载有效地址



现在EAX将x的地址保存，并能够在调用函数的时候push到栈中。

4.7.5 返回值

返回值通过寄存器，所有的有效类型（char,int,enum,etc）在EAX寄存器中。

4.7.6 其他calling convention

所有80x86都支持描述标准C的calling convention.

一个函数的calling convention可以通过\_\_attrbute\_\_扩展来声明。例如，为了声明一个void函数



Gcc支持标准calling convention。上面的函数时刻将cdecl换成stdcall。

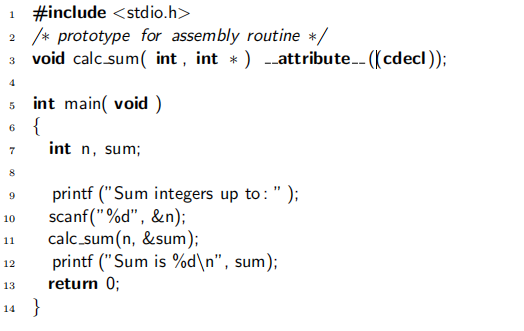
GCC也支持regparm，告诉编译器，最多传递三个参数给一个函数，来达到优化的目的。

微软通过下面方式声明calling convention



每个calling convention有优点和缺点，cdecl主要的有点是这比较简单，并且灵活。可以用于很多C函数和C编译器。

4.7.7



为什么push ebx重要？因为C calling conven 要求EBX在函数调用中没有修改。

第25行证明了dump\_stack宏的工作原理。注意，第一个参数是一个数值标签，第二个和第三个参数决定了要显示多少个double word。

4.7.8 从汇编语言调用C

一个需要注意的关键点是scanf。这意味着，要保存EBX,ESI和EDI寄存器。然而，EAX,ECX和EDX寄存器可能被修改，实际上EAX肯定会被修改。

4.8 可重入和递归子程序

可重入子程序必须满足下面的属性

1. 必须不修改任何代码指令。在高阶语言中，这比较困难，但是在汇编中并不困难。
2. 不能够修改global数据。所有变量都存在stack中。

写可重入代码时的有点

1. 可以递归调用
2. 被多个进程调用，在很多多线程操作系统中，如果有一个程序的多个实例，那么内存中只有一份copy。DLL就使用了这个想法
3. 在多线程中工作的比较好

4.8.1 递归子程序

这些类型的子程序调用他们自己。递归可以是直接递归或者间接递归。直接递归在子程序中，调用自己。间接递归当子程序没有被自己调用，而是被其他子程序调用，例如子程序foo调用bar，但是bar也调用foo。

递归子程序必须有终止条件，当终止条件为真，那么就没有递归。

1. 数组

5.1 介绍

数组是内从中的连续内存块。每个元素的类型必须相同，并使用同样数量的bytes进行存储。任何元素的地址可以通过三种方式获取。

1. 第一个元素的地址
2. 每个element的子捷速
3. 一个element的index

5.1.1 定义数组

在data和bss中定义数组

为了定义一个初始化数组，使用db，dw。NASM也提供有用的指令，叫做TIMES,这可以用来重复元素。在图5.1显示了多个例子。

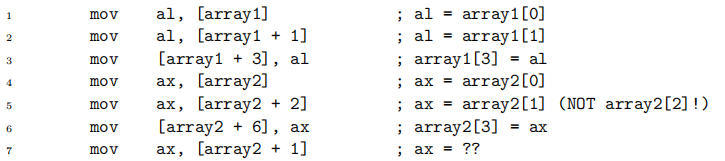
为了定义bss中未初始化的数组，私用resb，resw指令。注意这些指令声明保存的大暖内从。

把数组定义为局部变量

没有只记得方式在stack上定义数组。和前面一样，我们计损局部变量需要的总的bytes。然而，从ESP的字节数应该是4的辈数。

5.1.2 获取数组的元素

为了获取数组的元素，必须计算地址，考虑下面两个定义



注意第5行，word 1被引用，而不是2，为什么，因为array2的每个元素是两个byte。

在汇编中，移动的都是字节。

图5.3显示了一个代码，将array1中所有的元素加到之前。

5.1.3 更多高级的间接寻址

不出意外，间接寻址总是与数组结合在一起，最常用的间接寻址的放肆是。

第七章 结构与C++

7.1 结构

7.1.1 介绍

结构技术有一些有点

1. 把相关的数据放在一起
2. 简化数据传输
3. 增加代码的局部性

从汇编的角度，结构可以看作一个数组。真实数组的元素总是同样大小和同样类型。这个属性允许我们计算任何元素的地址。

一个结构内的元素大小并不一样，因此结构中的每个元素必须有一个名字tag，恶如是数值索引。

在汇编中，讨论结构的元素的方式与数组的元素相同。为了获取元素，我们必须知道结构的开始地址。然而，与数组不同的是，结构内的元素的offset并不能通过计算得到，而是编译器直接计算得到的。

ANSI C标准说明，一个结构的元素在内从中的方式与结构中定义的顺序相同。这也说明了结构的元素在内存中。也定义了ing一个有用的hong文件。这个宏有两个参数，第一个参数是结构的类型，第二个参数是名字。结果会返回offset。

7.1.2 内存alignment

如果使用offset来寻找offset，结果返回4，而不是2，为什么？因为gcc分类得基本单位是double word。图7.2显示了S结构，编译器讲两个没有使用的类型插入结构中来align y。这显示了为什么使用来计算offset而不是自己计算。

当然，如果结构只在汇编中使用，程序员可以决定offset。然而，如果讲汇编与C接口，那么汇编也需要内存alignment。一个复杂的是，不同的C编译器可能给不同的offset，，

Gcc编译器有着灵活以及复杂的方法来声明alignment，编译器允许哦我们来声明alignment类型。



等一一个新的类型叫做ualgnment int，在两个byte。1可以被其他幂次代替。如果y换成，然而，z任然。

Gcc编译器也允许我们打包结构，者告诉编译器来使用最小的可能空间。

微软支持用下面的方法。

这个指令告诉编译器来将元素的结构以字节为单位。指令保持，直到被其他指令重写。者会导致问题。如果头文件在其他头文件之前。这个头文件的结构可能被冈。一个程序的不哦同模块可能有不同的。没有办法来避免这个问题，微软只会吃一个方法来保持当前的alignment状态，并在后面回复。

7.1.3 bit 域

Bit域允许我们来声明结构的数量，只是用bit的特殊部分。Bits的大小并不是eigth的倍数。一个bit域定义为unsigned int或者int。然后是一个分号。这定义了一个32为变量，分成下面的部分。

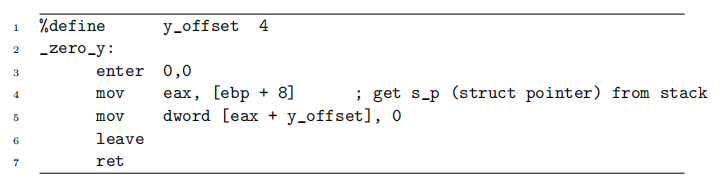
第一个bitfield分配到double word中最重要的部分。

然而，这个形式并不简单，如果我们看这个bit在内存中的具体存储。当bitfield扩展到边界是，会出现问题，因为在小端处理器中，内存是反向的。例如，S的bitfield会看起来像现模。

F2l变迁指向最后五个bits。

7.1.4 用汇编写结构

正如上面讨论的，获取结构与数组非常像。一个简单的例子，考虑我将结构S的y元素设置为零。



C允许我们pass by value传结构体到一个函数，然而这是个bad idea。当传入值，结构体中的整个数据必须复制到stack中，然后从routine中提取。

C也允许结构体类型用作函数的返回值，明显，一个结构体不能够在EAX寄存器中返回。不同的编译器通过不同的方式处理这个情况。一个创建的方案是，编译器重写这个函数，用结构体指针传入。

7.2 汇编和C++

C++编程语言是c的扩展，汇编与C的大部分接口也是用C++，然而，一些规则需要被修改。一些C++的扩展更加容易理解，

7.2.1 重载和名字

C++允许不同的函数定义同样的名字，当多个函数共享。

C++hi用名字来进行链接过程，但是通过name mangling来修改symbol。某种程度上，C也使用了name mangling。这给C函数的名字增加了一个先化学。C\_\_使用更加复杂的name mangling过程，例如，第一个函数，第二个函数

不幸的是，没有标准的方法来进行name mangling。Signature通过参数的顺序和类型定义的。

函数的返回类型不是名字signature的一部分。这个事实解释了C++的命名。只有signature唯一的函数才会被重载。如果两个函数有一个名字，signature定义在C++宏，他们会产生同样的名字，从而产生链接error。魔心情况下，C++