简明 x86 汇编语言教程

第〇章 写在前面

我不想夸大或者贬低汇编语言。但我想说，汇编语言改变了 20 世纪的历史。与前辈相比，我们这一代编程人员足够的幸福，因为我们有各式各样的编程语言，我们可以操作键盘、坐在显示器面前，甚至使用鼠标、语音识别。我们可以使用键盘、鼠标来驾驭“个人计算机”，而不是和一群人共享一台使用笨重的继电器、开关去操作的巨型机。相比之下，我们的前辈不得不使用机器语言编写程序，他们甚至没有最简单的汇编程序来把助记符翻译成机器语言，而我们可以从上千种计算机语言中选择我们喜欢的一种，而汇编，虽然不是一种“常用”的具有“快速原型开发”能力的语言，却也是我们可以选择的语言中的一种。

每种计算机都有自己的汇编语言——没必要指望汇编语言的可移植性，选择汇编，意味着选择性能而不是可移植或便于调试。这份文档中讲述的是 x86 汇编语言，此后的“汇编语言”一词，如果不明示则表示 ia32 上的 x86 汇编语言。

汇编语言是一种易学，却很难精通的语言。回想当年，我从初学汇编到写出第一个可运行的程序，只用了不到 4 个小时；然而直到今天，我仍然不敢说自己精通它。编写快速、高效、并且能够让处理器“很舒服地执行”的程序是一件很困难的事情，如果利用业余时间学习，通常需要 2-3 年的时间才能做到。这份教材并不期待能够教给你大量的汇编语言技巧。对于读者来说，x86 汇编语言"就在这里"。然而，不要僵化地局限于这份教材讲述的内容，因为它只能告诉你汇编语言是“这样一回事”。学好汇编语言，更多的要靠一个人的创造力于悟性，我可以告诉你我所知道的技巧，但肯定这是不够的。一位对我的编程生涯产生过重要影响的人曾经对我说过这么一句话：

> 写汇编语言程序**不是**汇编语言最难的部分，创新才是。

我想，愿意看这份文档的人恐怕不会问我“为什么要学习汇编语言”这样的问题；不过，我还是想说几句：首先，汇编语言非常有用，我个人主张把它作为 C 语言的先修课程，因为通过学习汇编语言，你可以了解到如何有效地设计数据结构，让计算机处理得更快，并使用更少的存储空间；同时，学习汇编语言可以让你熟悉计算机内部运行机制，并且，有效地提高调试能力。就我个人的经验而言，调试一个非结构化的程序的困难程度，要比调试一个结构化的程序的难度高很多，因为“结构化”是以牺牲运行效率来提高可读性与可调试性，这对于完成一般软件工程的编码阶段是非常必要的。然而，在一些地方，比如，硬件驱动程序、操作系统底层，或者程序中经常需要执行的代码，结构化程序设计的这些优点有时就会被它的低效率所抹煞。另外，如果你想真正地控制自己的程序，只知道源代码级的调试是远远不够的。

浮躁的人喜欢说，用 C++写程序足够了，甚至说，他不仅仅掌握 C++，而且精通 STL、MFC。我不赞成这个观点，掌握上面的那些是每一个编程人员都应该做到的，然而 C++只是我们"常用"的一种语言，它不是编程的全部。低层次的开发者喜欢说，嘿，C++是多么的强大，它可以做任何事情——这不是事实。便于维护、调试，这些确实是我们的追求目标，但是，写程序不能仅仅追求这个目标，因为我们最终的目的是满足设计需求，而不是个人非理性的理想。

这份教材适合已经学习过某种结构化程序设计语言的读者。其内容基于我在 1995 年给别人讲述汇编语言时所写的讲义。当然，如大家所希望的，它包含了最新的处理器所支持的特性，以及相应的内容。我假定读者已经知道了程序设计的一些基本概念，因为没有这些是无法理解汇编语言程序设计的；此外，我希望读者已经有了比较良好的程序设计基础，因为如果你缺乏对于结构化程序设计的认识，编写汇编语言程序很可能很快就破坏了你的结构化编程习惯，大大降低程序的可读性、可维护性，最终让你的程序陷于不得不废弃的代码堆之中。

基本上，这份文档撰写的目标是尽可能地便于自学。不过，它对你也有一些要求，尽管不是很高，但我还是强调一下。

学习汇编语言，你需要

- 胆量。不要害怕去接触那些计算机的内部工作机制。
- 知识。了解计算机常用的数制，特别是二进制、十六进制、八进制，以及计算机保存数据的方法。
- 开放。接受汇编语言与高级语言的差异，而不是去指责它如何的不好读。
- 经验。要求你拥有任意其他编程语言的一点点编程经验。
- 头脑。

祝您编程愉快！

第一章 汇编语言简介

先说一点和实际编程关系不太大的东西。当然，如果你迫切的想看到更实质的内容，完全可以先跳过这一章。

那么，我想可能有一个问题对于初学汇编的人来说非常重要，那就是：

汇编语言到底是什么？

汇编语言是一种最接近计算机核心的编码语言。不同于任何高级语言，汇编语言几乎可以完全和机器语言一一对应。不错，我们可以用机器语言写程序，但现在除了没有汇编程序的那些电脑之外，直接用机器语言写超过 1000 条以上指令的人大概只能算作那些被我们成为“圣人”的牺牲者一类了。毕竟，记忆一些短小的助记符、由机器去考虑那些琐碎的配位过程和检查错误，比记忆大量的随计算机而改变的十六进制代码、可能弄错而没有任何提示要强的多。熟练的汇编语言编码员甚至可以直接从十六进制代码中读出汇编语言的大致意思。当然，我们有更好的工具——汇编器和反汇编器。

简单地说，汇编语言就是机器语言的一种**可以被人读懂的形式**，只不过它更容易记忆。至于宏汇编，则是包含了宏支持的汇编语言，这可以让你编程的时候更专注于程序本身，而不是忙于计算和重写代码。

汇编语言除了机器语言之外最接近计算机硬件的编程语言。由于它如此的接近计算机硬件，因此，它可以最大限度地发挥计算机硬件的性能。用汇编语言编写的程序的速度通常要比高级语言和C/C++快很多--几倍，几十倍，甚至成百上千倍。当然，解释语言，如解释型 LISP，没有采用 JIT 技术的 Java 虚机中运行的 Java 等等，其程序速度更无法与汇编语言程序同日而语 。

永远不要忽视汇编语言的高速。实际的应用系统中，我们往往会用汇编彻底重写某些经常调用的部分以期获得更高的性能。应用汇编也许不能提高你的程序的稳定性，但至少，如果你非常小心的话，它也不会降低稳定性；与此同时，它可以大大地提高程序的运行速度。我强烈建议所有的软件产品在最后 Release 之前对整个代码进行 Profile，并适当地用汇编取代部分高级语言代码。至少，汇编语言的知识可以告诉你一些有用的东西，比如，你有多少个寄存器可以用。有时，手工的优化比编译器的优化更为有效，而且，你可以完全控制程序的实际行为。

我想我在罗嗦了。总之，在我们结束这一章之前，我想说，不要在优化的时候把希望完全寄托在编译器上——现实一些，再好的编译器也不可能总是产生最优的代码。

第二章 认识处理器

中央处理器(CPU)在微机系统处于“领导核心”的地位。汇编语言被编译成机器语言之后，将由处理器来执行。那么，首先让我们来了解一下处理器的主要作用，这将帮助你更好地驾驭它。

典型的处理器的主要任务包括

- 从内存中获取机器语言指令，译码，执行
- 根据指令代码管理它自己的寄存器
- 根据指令或自己的的需要修改内存的内容
- 响应其他硬件的中断请求

一般说来，处理器拥有对整个系统的所有总线的控制权。对于 Intel 平台而言，处理器拥有对数据、内存和控制总线的控制权，根据指令控制整个计算机的运行。在以后的章节中，我们还将讨论系统中同时存在多个处理器的情况。

处理器中有一些寄存器，这些寄存器可以保存特定长度的数据。某些寄存器中保存的数据对于系统的运行有特殊的意义。

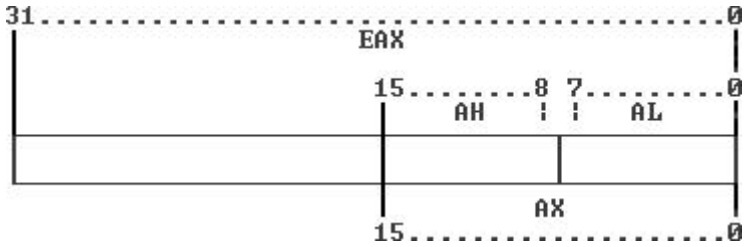
新的处理器往往拥有更多、具有更大字长的寄存器，提供更灵活的取指、寻址方式。

寄存器

如前所述，处理器中有一些可以保存数据的地方被称作寄存器。

寄存器可以被装入数据，你也可以在不同的寄存器之间移动这些数据，或者做类似的事情。基本上，像四则运算、位运算等这些计算操作，都主要是针对寄存器进行的。

首先让我来介绍一下 80386 上最常用的 4 个通用寄存器。先瞧瞧下面的图形，试着理解一下：



上图中，数字表示的是位。我们可以看出，EAX 是一个 32-bit 寄存器。同时，它的低 16-bit 又可以通过 AX 这个名字来访问；AX 又被分为高、低 8bit 两部分，分别由 AH 和 AL 来表示。

对于 EAX、AX、AH、AL 的改变同时也会影响与被修改的那些寄存器的值。从而事实上只存在一个 32-bit 的寄存器 EAX，而它可以通过 4 种不同的途径访问。

也许通过名字能够更容易地理解这些寄存器之间的关系。EAX 中的 E 的意思是“扩展的”，整个 EAX 的意思是扩展的 AX。X 的意思 Intel 没有明示，我个人认为表示它是一个可变的量 。而 AH、AL 中的 H 和 L 分别代表高和低 。

为什么要这么做呢？主要由于历史原因。早期的计算机是 8 位的，8086 是第一个 16 位处理器，其通用寄存器的名字是 AX，BX 等等；80386 是 Intel 推出的第一款 IA-32 系列处理器，所有的寄存器都被扩充为 32 位。为了能够兼容以前的 16 位应用程序，80386 不能将这些寄存器依旧命名为 AX、BX，并且简单地将他们扩充为 32 位——这将增加处理器在处理指令方面的成本。

Intel 微处理器的寄存器列表（在本章先只介绍 80386 的寄存器，MMX 寄存器以及其他新一代处理器的新寄存器将在以后的章节介绍）

通用寄存器

下面介绍通用寄存器及其习惯用法。顾名思义，通用寄存器是那些你可以根据自己的意愿使用的寄存器，修改他们的值通常不会对计算机的运行造成很大的影响。通用寄存器最多的用途是计算。

EAX 32-bit 宽	通用寄存器。相对其他寄存器，在进行运算方面比较常用。在保护模式中，也可以作为内存偏移指针（此时，DS 作为段 寄存器或选择器）
EBX 32-bit 宽	通用寄存器。通常作为内存偏移指针使用（相对于 EAX、ECX、EDX），DS 是默认的段寄存器或选择器。在保护模式中，同样可以起这个作用。
ECX 32-bit 宽	通用寄存器。通常用于特定指令的计数。在保护模式中，也可以作为内存偏移指针（此时，DS 作为 寄存器或段选择器）。
EDX 32-bit 宽	通用寄存器。在某些运算中作为 EAX 的溢出寄存器（例如乘、除）。在保护模式中，也可以作为内存偏移指针（此时，DS 作为段 寄存器或选择器）。

上述寄存器同 EAX 一样包括对应的 16-bit 和 8-bit 分组。

用作内存指针的特殊寄存器

ESI 32-bit 宽	通常在内存操作指令中作为“源地址指针”使用。当然，ESI 可以被装入任意的数值，但通常没有人把它当作通用寄存器来用。DS 是默认段寄存器或选择器。
EDI	通常在内存操作指令中作为“目的地址指针”使用。当然，EDI 也可以被装入任

32-bit 宽	意的数值，但通常没有人把它当作通用寄存器来用。DS 是默认段寄存器或选择器。
EBP 32-bit 宽	这也是一个作为指针的寄存器。通常，它被高级语言编译器用以建造‘堆栈帧’来保存函数或过程的局部变量，不过，还是那句话，你可以在其中保存你希望的任何数据。SS 是它的默认段寄存器或选择器。

注意，这三个寄存器没有对应的 8-bit 分组。换言之，你可以通过 SI、DI、BP 作为别名访问他们的低 16 位，却没有办法直接访问他们的低 8 位。

段寄存器和选择器

实模式下的段寄存器到保护模式下摇身一变就成了选择器。不同的是，实模式下的“段寄存器”是 16-bit 的，而保护模式下的选择器是 32-bit 的。

CS	代码段，或代码选择器。同 IP 寄存器(稍后介绍)一同指向当前正在执行的那个地址。处理器执行时从这个寄存器指向的段（实模式）或内存（保护模式）中获取指令。除了跳转或其他分支指令之外，你无法修改这个寄存器的内容。
DS	数据段，或数据选择器。这个寄存器的低 16 bit 连同 ESI 一同指向的指令将要处理的内存。同时，所有的内存操作指令 默认情况下都用它指定操作段(实模式)或内存(作为选择器，在保护模式。这个寄存器可以被装入任意数值，然而在这么做的时候需要小心一些。方法是，首先把数据送给 AX，然后再把它从 AX 传送给 DS(当然，也可以通过堆栈来做).
ES	附加段，或附加选择器。这个寄存器的低 16 bit 连同 EDI 一同指向的指令将要处理的内存。同样的，这个寄存器可以被装入任意数值，方法和 DS 类似。
FS	F 段或 F 选择器(推测 F 可能是 Free?)。可以用这个寄存器作为默认段寄存器或选择器的一个替代品。它可以被装入任何数值，方法和 DS 类似。
GS	G 段或 G 选择器(G 的意义和 F 一样，没有在 Intel 的文档中解释)。它和 FS 几乎完全一样。
SS	堆栈段或堆栈选择器。这个寄存器的低 16 bit 连同 ESP 一同指向下一次堆栈操作(push 和 pop)所要使用的堆栈地址。这个寄存器也可以被装入任意数值，你可以通过入栈和出栈操作来给他赋值，不过由于堆栈对于很多操作有很重要的意义，因此，不正确的修改有可能造成对堆栈的破坏。

* 注意 一定不要在初学汇编的阶段把这些寄存器弄混。他们非常重要，而一旦你掌握了他们，你就可以对他们做任意的操作了。段寄存器，或选择器，在没有指定的情况下都是使用默认的那个。这句话在现在看来可能有点稀里糊涂，不过你很快就会在后面知道如何去做。

特殊寄存器(指向到特定段或内存的偏移量)：

EIP	这个寄存器非常的重要。这是一个 32 位宽的寄存器 ，同 CS 一同指向即将执行的那条指令的地址。不能够直接修改这个寄存器的值，修改它的唯一方法是跳转或分支指令。(CS 是默认的段或选择器)
ESP	这个 32 位寄存器指向堆栈中即将被操作的那个地址。尽管可以修改它的值，然而并不提倡这样做，因为如果你不是非常明白自己在做什么，那么你可能造成

	堆栈的破坏。对于绝大多数情况而言，这对程序是致命的。(SS 是默认的段或选择器)

IP: Instruction Pointer, 指令指针
SP: Stack Pointer, 堆栈指针

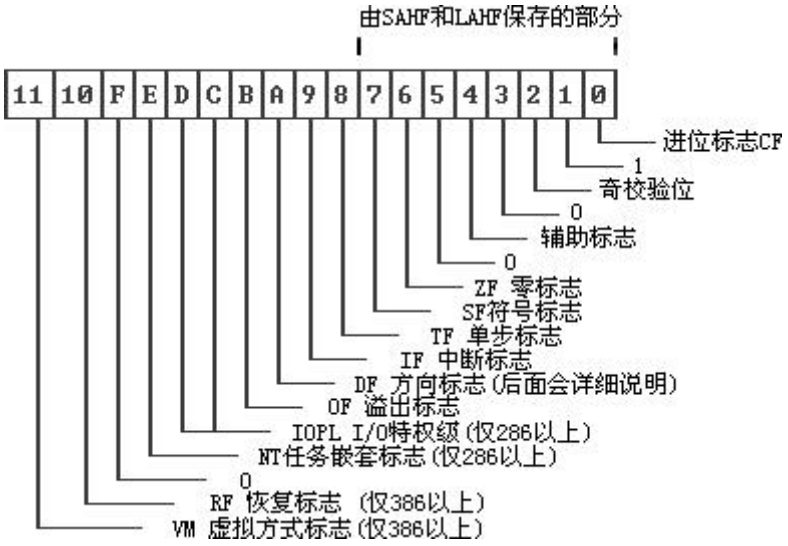
好了，上面是最基本的寄存器。下面是一些其他的寄存器，你甚至可能没有听说过它们。(都是 32 位宽)：

CR0, CR2, CR3(控制寄存器)。举一个例子，CR0 的作用是切换实模式和保护模式。

还有其他一些寄存器，D0, D1, D2, D3, D6 和 D7(调试寄存器)。他们可以作为调试器的硬件支持来设置条件断点。

TR3, TR4, TR5, TR6 和 TR? 寄存器(测试寄存器)用于某些条件测试。

最后我们要说的是一个在程序设计中起着非常关键的作用的寄存器：标志寄存器。



本节中部份表格来自 David Jurgens 的 HelpPC 2.10 快速参考手册。在此谨表谢意。

2.2 使用寄存器

在前一节中的 x86 基本寄存器的介绍，对于一个汇编语言编程人员来说是不可或缺的。现在你知道，寄存器是处理器内部的一些保存数据的存储单元。仅仅了解这些是不足以写出一个可用的汇编语言程序的，但你已经可以大致读懂一般汇编语言程序了（不必惊讶，因为汇编语言的祝记符和英文单词非常接近），因为你已经了解了关于基本寄存器的绝大多数知识。

在正式引入第一个汇编语言程序之前，我粗略地介绍一下汇编语言中不同进制整数的表示方法。如果你不了解十进制以外的其他进制，请把鼠标移动到这里。

汇编语言中的整数常量表示

- **十进制整数**
  这是汇编器默认的数制。直接用我们熟悉的表示方式表示即可。例如，1234 表示十进制的 1234。不过，如果你指定了使用其他数制，或者有凡事都进行完整定义的小爱好，也可以写成[十进制数]d 或[十进制数]D 的形式。
- **十六进制数**
  这是汇编程序中最常用的数制，我个人比较偏爱使用十六进制表示数据，至于为什么，以后我会作说明。十六进制数表示为 0[十六进制数]h 或 0[十六进制数]H，其中，如果十六进制数的第一位是数字，则开头的 0 可以省略。例如，7fffh, 0ffffh，等等。
- **二进制数**
  这也是一种常用的数制。二进制数表示为[二进制数]b 或[二进制数]B。一般程序中用二进制数表示掩码（mask code）等数据非常的直观，但需要些很长的数据（4 位二进制数相当于一位十六进制数）。例如，1010110b。
- **八进制数**
  八进制数现在已经不是很常用了（确实还在用，一个典型的例子是 Unix 的文件属性）。八进制数的形式是[八进制数]q、[八进制数]Q、[八进制数]o、[八进制数]O。例如，777Q。

需要说明的是，这些方法是针对宏汇编器（例如，MASM、TASM、NASM）说的，调试器默认使用十六进制表示整数，并且不需要特别的声明（例如，在调试器中直接用 FFFF 表示十进制的 65535，用 10 表示十进制的 16）。

现在我们来写一小段汇编程序，修改 EAX、EBX、ECX、EDX 的数值。

我们假定程序执行之前，寄存器中的数值是全 0：

	?	X	
		H	L
EAX	0000	00	00
EBX	0000	00	00
ECX	0000	00	00
EDX	0000	00	00

正如前面提到的，EAX 的高 16bit 是没有办法直接访问的，而 AX 对应它的低 16bit，AH、AL 分别对应 AX 的高、低 8bit。

```
mov eax, 012345678h                    ; 将 012345678h 送入 eax
mov ebx, 0abcdeffeh                    ; 将 0abcdeffeh 送入 ebx
mov ecx, 1                             ; 将 000000001h 送入 ecx
mov edx, 2                             ; 将 000000002h 送入 edx
```

则执行上述程序段之后，寄存器的内容变为：

	?	X	
		H	L
EAX	1234	56	78
EBX	abcd	ef	fe
ECX	0000	00	01
EDX	0000	00	02

那么，你已经了解了 mov 这个指令（mov 是 move 的缩写）的一种用法。它可以将数送到寄存器中。我们来看看下面的代码：

```
mov eax, ebx                    ; ebx 内容送入 eax
mov ecx, edx                    ; edx 内容送入 ecx
```

则寄存器内容变为：

	?	X	
		H	L
EAX	abcd	ef	fe
EBX	abcd	ef	fe
ECX	0000	00	02
EDX	0000	00	02

我们可以看到，“move”之后，数据依然保存在原来的寄存器中。不妨把 mov 指令理解为“送入”，或“装入”。

练习题

把寄存器恢复成都为全 0 的状态，然后执行下面的代码：

```
mov eax, 0a1234h                ; 将 0a1234h 送入 eax
mov bx, ax                      ; 将 ax 的内容送入 bx
mov ah, bl                      ; 将 bl 内容送入 ah
mov al, bh                      ; 将 bh 内容送入 al
```

思考：此时，EAX 的内容将是多少？[答案]

下面我们将介绍一些指令。在介绍指令之前，我们约定：

使用 Intel 文档中的寄存器表示方式

- reg32 32-bit 寄存器（表示 EAX、EBX 等）
- reg16 16-bit 寄存器（在 32 位处理器中，这 AX、BX 等）
- reg8 8-bit 寄存器（表示 AL、BH 等）

- imm32 32-bit 立即数（可以理解为常数）
- imm16 16-bit 立即数
- imm8 8-bit 立即数

在寄存器中载入另一寄存器，或立即数的值：

```
mov reg32, (reg32 | imm8 | imm16 | imm32)
mov reg32, (reg16 | imm8 | imm16)
mov reg8, (reg8 | imm8)
```

例如，mov eax, 010h 表示，在 eax 中载入 00000010h。需要注意的是，如果你希望在寄存器中装入 0，则有一种更快的方法，在后面我们将提到。

交换寄存器的内容：

```
xchg reg32, reg32
xchg reg16, reg16
xchg reg8, reg8
```

例如，xchg ebx, ecx，则 ebx 与 ecx 的数值将被交换。由于系统提供了这个指令，因此，采用其他方法交换时，速度将会较慢，并需要占用更多的存储空间，编程时要避免这种情况，即，尽量利用系统提供的指令，因为多数情况下，这意味着更小、更快的代码，同时也杜绝了错误（如果说 Intel 的 CPU 在交换寄存器内容的时候也会出错，那么它就不用卖 CPU 了。而对于你来说，检查一行代码的正确性也显然比检查更多代码的正确性要容易）刚才的习题的程序用下面的代码将更有效：

```
mov eax, 0a1234h                    ; 将 0a1234h 送入 eax
mov bx, ax                          ; 将 ax 内容送入 bx
xchg ah, al                         ; 交换 ah, al 的内容
```

递增或递减寄存器的值：

```
inc reg(8,16,32)
dec reg(8,16,32)
```

这两个指令往往用于循环中对指针的操作。需要说明的是，某些时候我们有更好的方法来处理循环，例如使用 loop 指令，或 rep 前缀。这些将在后面的章节中介绍。

将寄存器的数值与另一寄存器，或立即数的值相加，并存回此寄存器：

```
add reg32, reg32 / imm(8,16,32)
add reg16, reg16 / imm(8,16)
add reg8, reg8 / imm(8)
```

例如，add eax, edx，将 eax+edx 的值存入 eax。减法指令和加法类似，只是将 add 换成 sub。

需要说明的是，与高级语言不同，汇编语言中，如果要计算两数之和（差、积、商，或一般地说，运算结果），那么必然有一个寄存器被用来保存结果。在 PASCAL 中，我们可以用 nA := nB + nC 来让 nA 保存 nB+nC 的结果，然而，汇编语言并不提供这种方法。如果你希望保持寄存器中的结果，需要用另外的指令。这也从另一个侧面反映了“寄存器”这个名字的意义。数据只是“寄存”在那里。如果你需要保存数据，那么需要将它放到内存或其他地方。

类似的指令还有 and、or、xor（与，或，异或）等等。它们进行的是逻辑运算。

我们称 add、mov、sub、and 等称为为指令助记符（这么叫是因为它比机器语言容易记忆，而起作用就是方便人记忆，某些资料中也称为指令、操作码、opcode[operation code]等）；后面的参数成为操作数，一个指令可以没有操作数，也可以有一两个操作数，通常有一个操作数的指令，这个操作数就是它的操作对象；而两个参数的指令，前一个操作数一般是保存操作结果的地方，而后一个是附加的参数。

我不打算在这份教程中用大量的篇幅介绍指令——很多人做得比我更好，而且指令本身并不是重点，如果你学会了如何组织语句，那么只要稍加学习就能轻易掌握其他指令。更多的指令可以参考 Intel 提供的资料。编写程序的时候，也可以参考一些在线参考手册。Tech!Help 和 HelpPC 2.10 尽管已经很旧，但足以应付绝大多数需要。

聪明的读者也许已经发现，使用 sub eax, eax，或者 xor eax, eax，可以得到与 mov eax, 0 类似的效果。在高级语言中，你大概不会选择用 a=a-a 来给 a 赋值，因为测试会告诉你这么做更慢，简直就是在自找麻烦，然而在汇编语言中，你会得到相反的结论，多数情况下，以由快到慢的速度排列，这三条指令将是 xor eax, eax、sub eax, eax 和 mov eax, 0。

为什么呢？处理器在执行指令时，需要经过几个不同的阶段：取指、译码、取数、执行。

我们反复强调，寄存器是 CPU 的一部分。从寄存器取数，其速度很显然要比从内存中取数快。那么，不难理解，xor eax, eax 要比 mov eax, 0 更快一些。

那么，为什么 a=a-a 通常要比 a=0 慢一些呢？这和编译器的优化有一定关系。多数编译器会把 a=a-a 翻译成类似下面的代码(通常，高级语言通过 ebp 和偏移量来访问局部变量；程序中，x 为 a 相对于本地堆的偏移量，在只包含一个 32-bit 整形变量的程序中，这个值通常是 4)：

```
mov eax, dword ptr [ebp-x]
sub eax, dword ptr [ebp-x]
mov dword ptr [ebp-x],eax
```

而把 a=0 翻译成

```
mov dword ptr [ebp-x], 0
```

上面的翻译只是示意性的，略去了很多必要的步骤，如保护寄存器内容、恢复等等。如果你对与编译程序的实现过程感兴趣，可以参考相应的书籍。多数编译器（特别是 C/C++编译器，如 Microsoft Visual C++）都提供了从源代码到宏汇编语言程序的附加编译输出选项。这种情况下，

你可以很方便地了解编译程序执行的输出结果；如果编译程序没有提供这样的功能也没有关系，调试器会让你看到编译器的编译结果。

如果你明确地知道编译器编译出的结果不是最优的，那就可以着手用汇编语言来重写那段代码了。怎么确认是否应该用汇编语言重写呢？

使用汇编语言重写代码之前需要确认的几件事情

- 首先，这种优化**最好**有**明显的效果**。比如，一段循环中的计算，等等。一条语句的执行时间是很短的，现在新的 CPU 的指令周期都在 0.000000001s 以下，Intel 甚至已经做出了 4GHz 主频（主频的倒数是时钟周期）的 CPU，如果你的代码自始至终只执行一次，并且你只是减少了几个时钟周期的执行时间，那么改变将是无法让人察觉的；很多情况下，这种“优化”并不被提倡，尽管它确实减少了执行时间，但为此需要付出大量的时间、人力，多数情况下得不偿失（极端情况，比如你的设备内存价格非常昂贵的时候，这种优化也许会有意义）。
- 其次，确认你已经使用了**最好的算法**，并且，你优化的程序的实现是**正确**的。汇编语言能够提供同样算法的最快实现，然而，它并不是万金油，更不是解决一切的灵丹妙药。用高级语言实现一种好的算法，不一定会比汇编语言实现一种差的算法更慢。不过需要注意的是，时间、空间复杂度最小的算法不一定就是解决某一特定问题的最佳算法。举例说，快速排序在完全逆序的情况下等价于冒泡排序，这时其他方法就比它快。同时，用汇编语言优化一个不正确的算法实现，将给调试带来很大的麻烦。
- 最后，确认你**已经**将高级语言编译器的性能**发挥到极致**。Microsoft 的编译器在 RELEASE 模式和 DEBUG 模式会有差异相当大的输出，而对于 GNU 系列的编译器而言，不同级别的优化也会生成几乎完全不同的代码。此外，在编程时对于问题的严格定义，可以极大地帮助编译器的优化过程。如何优化高级语言代码，使其编译结果最优超出了本教程的范围，但如果你不能确认已经发挥了编译器的最大效能，用汇编语言往往是一种更为费力的方法。
- **还有一点非常重要，那就是你明白自己做的是什么。**好的高级语言编译器有时会有一些让人难以理解的行为，比如，重新排列指令顺序，等等。如果你发现这种情况，那么优化的时候就应该小心——编译器很可能比你拥有更多的关于处理器的知识，例如，对于一个超标量处理器，编译器会对指令序列进行“封包”，使他们尽可能的并行执行；此外，宏汇编器有时会自动插入一些 nop 指令，其作用是将指令凑成整数字长（32-bit，对于 16-bit 处理器，是 16-bit）。这些都是提高代码性能的必要措施，如果你不了解处理器，那么最好不要改动编译器生成的代码，因为这种情况下，盲目的修改往往不会得到预期的效果。

曾经在一份杂志上看到过有人用纯机器语言编写程序。不清楚到底这是不是编辑的失误，因为一个头脑正常的人恐怕不会这么做程序，即使它不长、也不复杂。首先，汇编器能够完成某些封包操作，即使不行，也可以用 db 伪指令来写指令；用汇编语言写程序可以防止很多错误的发生，同时，它还减轻了人的负担，很显然，“完全用机器语言写程序”是完全没有必要的，因为汇编语言可以做出完全一样的事情，并且你可以依赖它，因为计算机不会出错，而人总有出错的时候。

此外，如前面所言，如果用高级语言实现程序的代价不大（例如，这段代码在程序的整个执行过程中只执行一遍，并且，这一遍的执行时间也小于一秒），那么，为什么不用高级语言实现呢？

一些比较狂热的编程爱好者可能不太喜欢我的这种观点。比方说，他们可能希望精益求精地优化每一字节的代码。但多数情况下我们有更重要的事情，例如，你的算法是最优的吗？你已经把程序在高级语言许可的范围内优化到尽头了吗？并不是所有的人都有资格这样说。汇编语言是这样一件东西，它足够的强大，能够控制计算机，完成它能够实现的任何功能；同时，因为它的强大，也会提高开发成本，并且，难于维护。因此，我个人的建议是，如果在软件开发中使用汇编语言，则应在软件接近完成的时候使用，这样可以减少很多不必要的投入。

第二章中，我介绍了 x86 系列处理器的基本寄存器。这些寄存器对于 x86 兼容处理器仍然是有效的，如果你偏爱 AMD 的 CPU，那么使用这些寄存器的程序同样也可以正常运行。

不过现在说用汇编语言进行优化还为时尚早——不可能写程序，而只操作这些寄存器，因为这样只能完成非常简单的操作，既然是简单的操作，那可能就会让人觉得乏味，甚至找一台足够快的机器穷举它的所有结果（如果可以穷举的话），并直接写程序调用，因为这样通常会更快。但话说回来，看完接下来的两章——内存和堆栈操作，你就可以独立完成几乎所有的任务了，配合第五章中断、第六章子程序的知识，你将知道如何驾驭处理器，并让它为你工作。

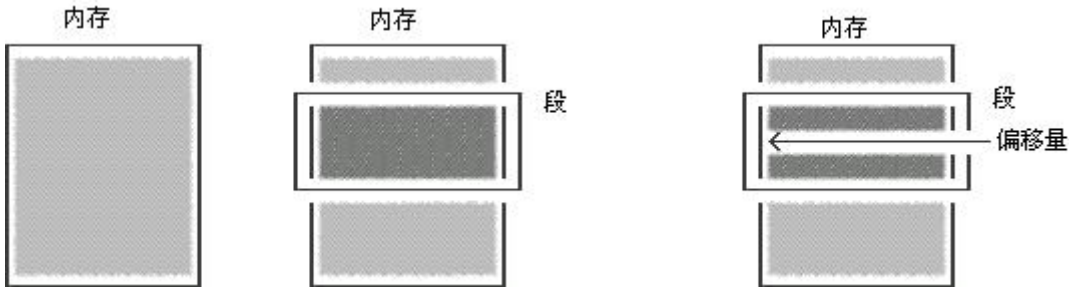
第三章 操作内存

在前面的章节中，我们已经了解了寄存器的基本使用方法。而正如结尾提到的那样，仅仅使用寄存器做一点运算是没有什么太大意义的，毕竟它们不能保存太多的数据，因此，对编程人员而言，他肯定迫切地希望访问内存，以保存更多的数据。

我将分别介绍如何在保护模式和实模式操作内存，然而在此之前，我们先熟悉一下这两种模式中内存的结构。

3.1 实模式

事实上，在实模式中，内存比保护模式中的结构更令人困惑。内存被分割成段，并且，操作内存时，需要指定段和偏移量。不过，理解这些概念是非常容易的事情。请看下面的图：

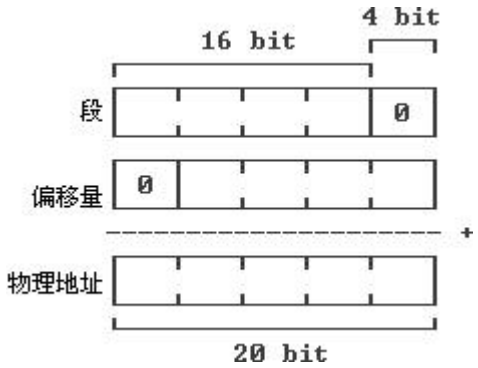


段-寄存器这种格局是早期硬件电路限制留下的一个伤疤。地址总线在当时有 20-bit。

然而 20-bit 的地址不能放到 16-bit 的寄存器里，这意味着有 4-bit 必须放到别的地方。因此，为了访问所有的内存，必须使用两个 16-bit 寄存器。

这一设计上的折衷方案导致了今天的段-偏移量格局。最初的设计中，其中一个寄存器只有 4-bit 有效，然而为了简化程序，两个寄存器都是 16-bit 有效，并在执行时求出加权和来标识 20-bit 地址。

偏移量是 16-bit 的，因此，一个段是 64KB。下面的图可以帮助你理解 20-bit 地址是如何形成的：



段-偏移量标识的地址通常记做 **段:偏移量** 的形式。

由于这样的结构，一个内存有多个对应的地址。例如，0000:0010 和 0001:0000 指的是同一内存地址。又如，

0000:1234 = 0123:0004 = 0120:0034 = 0100:0234
0001:1234 = 0124:0004 = 0120:0044 = 0100:0244

作为负面影响之一，在段上加 1 相当于在偏移量上加 16，而不是一个“全新”的段。反之，在偏移量上加 16 也和在段上加 1 等价。某些时候，据此认为段的“粒度”是 16 字节。

练习题
尝试一下将下面的地址转化为 20bit 的地址：

```
2EA8:D678 26CF:8D5F 453A:CFAD 2933:31A6 5924:DCCF
694E:175A 2B3C:D218 728F:6578 68E1:A7DC 57EC:AEEA
```

稍高一些的要求是，写一个程序将段为 AX、偏移量为 BX 的地址转换为 20bit 的地址，并保存于 EAX 中。

[上面习题的答案]

我们现在可以写一个真正的程序了。

经典程序：Hello, world

```
;;; 应该得到一个 29 字节的.com 文件
```

```
.MODEL TINY                                ; .COM 文件的内存模型是‘TINY’
.CODE                                      ; 代码段开始

CR equ 13                                  ; 回车
LF equ 10                                  ; 换行
TERMINATOR equ '$'                         ; DOS 字符串结束符

ORG 100h                                   ; 代码起始地址为 CS:0100h

Main PROC
mov dx,offset sMessage                     ; 令 DS:DX 指向 Message
mov ah,9                                   ; int 21h(DOS 中断)功能 9 -
int 21h                                    ; 显示字符串到标准输出设备
mov ax,4c00h                               ; int 21h 功能 4ch -
int 21h                                    ; 终止程序并返回 AL 的错误代码
Main ENDP

sMessage:
  DB 'Hello, World!'
  DB CR,LF,TERMINATOR
                                           ; 程序结束的同时指定入口点为 Main
END Main
```

那么，我们需要解释很多东西。

首先，作为汇编语言的抽象，C 语言拥有“指针”这个数据类型。在汇编语言中，几乎所有对内存的操作都是由对给定地址的内存进行访问来完成的。这样，在汇编语言中，绝大多数操作都要和指针产生或多或少的联系。

这里我想强调的是，由于这一特性，汇编语言中同样会出现 C 程序中常见的缓冲区溢出问题。如果你正在设计一个与安全有关的系统，那么最好是仔细检查你用到的每一个串，例如，它们是否一定能够以你预期的方式结束，以及（如果使用的话）你的缓冲区是否能保证实际可能输入的数据不被写入到它以外的地方。作为一个汇编语言程序员，你有义务检查每一行代码的可用性。

程序中的 equ 伪指令是宏汇编特有的，它的意思接近于 C 或 Pascal 中的 const（常量）。多数情况下，equ 伪指令并不为符号分配空间。

此外，汇编程序执行一项操作是非常繁琐的，通常，在对与效率要求不高的地方，我们习惯使用系统提供的中断服务来完成任务。例如本例中的中断 21h，它是 DOS 时代的中断服务，在 Windows 中，它也被认为是 Windows API 的一部分（这一点可以在 Microsoft 的文档中查到）。中断可以被理解为高级语言中的子程序，但又不完全一样——中断使用系统栈来保存当前的机器状态，可以由硬件发起，通过修改机器状态字来反馈信息，等等。

那么，最后一段通过 DB 存放的数据到底保存在哪里了呢？答案是紧挨着代码存放。在汇编语言中，DB 和普通的指令的地位是相同的。如果你的汇编程序并不知道新的助记符（例如，新的处理器上的 CPUID 指令），而你很清楚，那么可以用 DB 机器码的方式强行写下指令。这意味着，你可以超越汇编器的能力撰写汇编程序，然而，直接用机器码编程是几乎肯定是一件费力不讨好的事——汇编器厂商会经常更新它所支持的指令集以适应市场需要，而且，你可以期待你的汇编其能够产生正确的代码，因为机器查表是不会出错的。既然机器能够帮我们做将程序转换为代码这件事情，那么为什么不让它来做呢？

细心的读者不难发现，在程序中我们没有对 DS 进行赋值。那么，这是否意味着程序的结果将是不可预测的呢？答案是否定的。DOS（或 Windows 中的 MS-DOS VM）在加载.com 文件的时候，会对寄存器进行很多初始化。.com 文件被限制为小于 64KB，这样，它的代码段、数据段都被装入同样的数值（即，初始状态下 DS=CS）。

也许会有人说，“嘿，这听起来不太好，一个 64KB 的程序能做得了什么呢？还有，你吹得天花乱坠的堆栈段在什么地方？”那么，我们来看看下面这个新的 Hello world 程序，它是一个 EXE 文件，在 DOS 实模式下运行。

```
;;; 应该得到一个 561 字节的 EXE 文件

.MODEL SMALL                          ; 采用“SMALL”内存模型
.STACK 200h                           ; 堆栈段

CR equ 13                             ; 回车
LF equ 10                             ; 换行
TERMINATOR equ '$'                    ; DOS 字符串结束符

.DATA                                 ; 定义数据段

Message DB 'Hello, World !'           ; 定义显示串
  DB CR,LF,TERMINATOR

.CODE                                 ; 定义代码段

Main PROC                             ; 将数据段
mov ax, DGROUP                        ; 加载到 DS 寄存器
mov ds, ax

                                      ; 设置 DX
mov dx, offset Message                ; 显示
mov ah, 9
int 21h

mov ax, 4c00h                         ; 终止程序
int 21h
Main ENDP
```

```
END main
```

561 字节？实现相同功能的程序大了这么多！为什么呢？我们看到，程序拥有了完整的堆栈段、数据段、代码段，其中堆栈段足足占掉了 512 字节，其余的基本上没什么变化。

分成多个段有什么好处呢？首先，它让程序显得更加清晰——你肯定更愿意看一个结构清楚的程序，代码中 hard-coded 的字符串、数据让人觉得费解。比如，mov dx, 0152h 肯定不如 mov dx, offset Message 来的亲切。此外，通过分段你可以使用更多的内存，比如，代码段腾出的空间可以做更多的事情。exe 文件另一个吸引人的地方是它能够实现“重定位”。现在你不需要指定程序入口点的地址了，因为系统会找到你的程序入口点，而不是死板的 100h。

程序中的符号也会在系统加载的时候重新赋予新的地址。exe 程序能够保证你的设计容易地被实现，不需要考虑太多的细节。

当然，我们的主要目的是将汇编语言作为高级语言的一个有用的补充。如我在开始提到的那样，真正完全用汇编语言实现的程序不一定就好，因为它不便于维护，而且，由于结构的原因，你也不太容易确保它是正确的；汇编语言是一种非结构化的语言，调试一个精心设计的汇编语言程序，即使对于一个老手来说也不啻是一场恶梦，因为你很可能掉到别人预设的“陷阱”中——这些技巧确实提高了代码性能，然而你很可能不理解它，于是你把它改掉，接着就发现程序彻底败掉了。使用汇编语言加强高级语言程序时，你要做的通常只是使用汇编指令，而不必搭建完整的汇编程序。绝大多数（也是目前我遇到的全部）C/C++编译器都支持内嵌汇编，即在程序中使用汇编语言，而不必撰写单独的汇编语言程序——这可以节省你的不少精力，因为前面讲述的那些伪指令，如 equ 等，都可以用你熟悉的高级语言方式来编写，编译器会把它转换为适当的形式。

需要说明的是，在高级语言中一定要注意编译结果。编译器会对你的汇编程序做一些修改，这不一定符合你的要求（附带说一句，有时编译器会很聪明地调整指令顺序来提高性能，这种情况下最好测试一下哪种写法的效果更好），此时需要做一些更深入的修改，或者用 db 来强制编码。

3.2 保护模式

实模式的东西说得太多了，尽管我已经删掉了许多东西，并把一些原则性的问题拿到了这一节讨论。这样做不是没有理由的——保护模式才是现在的程序（除了操作系统的底层启动代码）最常用的 CPU 模式。保护模式提供了很多令人耳目一新的功能，包括内存保护（这是保护模式这个名字的来源）、进程支持、更大的内存支持，等等。

对于一个编程人员来说，能“偷懒”是一件令人愉快的事情。这里“偷懒”是说把“应该”由系统做的事情做的事情全都交给系统。为什么呢？这出自一个基本思想——人总有犯错误的时候，然而规则不会，正确地了解规则之后，你可以期待它像你所了解的那样执行。对于 C 程序来说，你自己用 C 语言写的实现相同功能的函数通常没有系统提供的函数性能好（除非你用了比函数库好很多的算法），因为系统的函数往往使用了更好的优化，甚至可能不是用 C 语言直接编写的。

当然，“偷懒”的意思是说，把那些应该让机器做的事情交给计算机来做，因为它做得更好。我们应该把精力集中到设计算法，而不是编写源代码本身上，因为编译器几乎只能做等价优化，而实现相同功能，但使用更好算法的程序实现，则几乎只能由人自己完成。

举个例子，这样一个函数：

```
int fun(){
  int a=0;
  register int i;
  for(i=0; i<1000; i++) a+=i;
  return a;
}
```

在某种编译模式[DEBUG]下被编译为

```
push ebp                                ; 子程序入口
mov ebp,esp
sub esp,48h
push ebx                                ; 保护现场
push esi
push edi
lea edi,[ebp-48h]                       ; 初始化变量-调试版本特有。
mov ecx,12h                             ; 本质是在堆中挖一块地儿，存 CCCCCCCC。
mov eax,0CCCCCCCCh                      ; 用串操作进行，这将发挥 Intel 处理器优势
rep stos dword ptr [edi]                ; ‘a=0’
mov dword ptr [ebp-4],0                 ; ‘i=0’
mov dword ptr [ebp-8],0
jmp fun+31h                             ; 走着
mov eax,dword ptr [ebp-8]               ; i++
add eax,1
mov dword ptr [ebp-8],eax
cmp dword ptr [ebp-8],3E8h              ; i<1000?
jge fun+45h
mov ecx,dword ptr [ebp-4]
add ecx,dword ptr [ebp-8]               ; a+=i;
mov dword ptr [ebp-4],ecx
jmp fun+28h                             ; return a;
mov eax,dword ptr [ebp-4]
pop edi                                 ; 恢复现场
pop esi
pop ebx
mov esp,ebp
pop ebp                                 ; 返回
ret
```

而在另一种模式[RELEASE/MINSIZE]下却被编译为

```
xor eax,eax                             ; a=0;
xor ecx,ecx                             ; i=0;
add eax,ecx                             ; a+=i;
inc ecx                                 ; i++;
cmp ecx,3E8h                            ; i<1000?
jl fun+4                                ; 是->继续继续
ret                                     ; return a
```

如果让我来写，多半会写成

```
mov eax, 079f2ch
                                        ; return 499500
ret
```

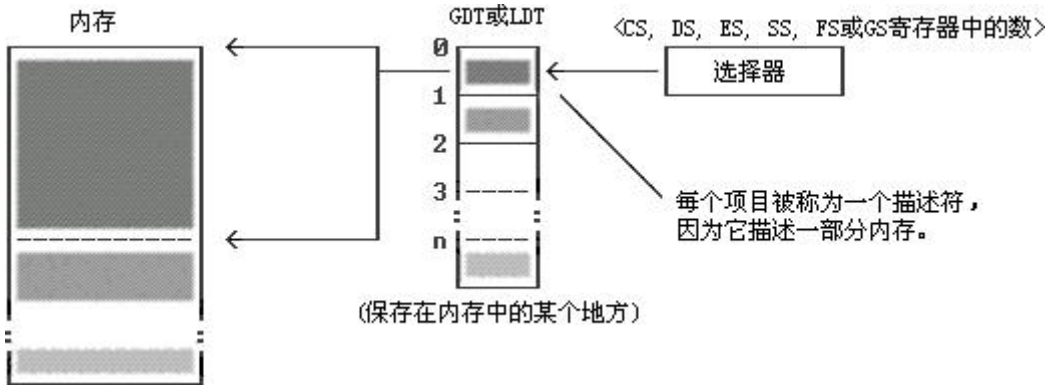
为什么这样写呢？我们看到，i 是一个外界不能影响、也无法获知的内部状态量。作为这段程序来说，对它的计算对于结果并没有直接的影响——它的存在不过是方便算法描述而已。并且我们看到的，这段程序实际上无论执行多少次，其结果都不会发生变化，因此，直接返回计算结果就可以了，计算是多余的（如果说一定要算，那么应该是编译器在编译过程中完成它）。

更进一步，我们甚至希望编译器能够直接把这个函数变成一个符号常量，这样连操作堆栈的过程也省掉了。

第三种结果属于“等效”代码，而不是“等价”代码。作为用户，很多时候是希望编译器这样做的，然而由于目前的技术尚不成熟，有时这种做法会造成一些问题（gcc 和 g++的顶级优化可以造成编译出的 FreeBSD 内核行为异常，这是我在 FreeBSD 上遇到的唯一一次软件原因的 kernel panic），因此，并不是所有的编译器都这样做（另一方面的原因是，如果编译器在这方面做的太过火，例如自动求解全部“固定”问题，那么如果你的程序是解决固定的问题“很大”，如求解迷宫，那么在编译过程中你就会找锤子来砸计算机了）。然而，作为编译器制造商，为了提高自己的产品的竞争力，往往会使用第三种代码来做函数库。正如前面所提到的那样，这种优化往往不是编译器本身的作用，尽管现代编译程序拥有编译执行、循环代码外提、无用代码去除等诸多优化功能，但它都不能保证程序最优。最后一种代码恐怕很少有编译器能够做到，不信你可以用自己常用的编译器加上各种优化选项试试:)

发现什么了吗？三种代码中，对于内存的访问一个比一个少。这样做的理由是，尽可能地利用寄存器并减少对内存的访问，可以提高代码性能。在某些情况下，使代码既小又快是可能的。

书归正传，我们来说说保护模式的内存模型。保护模式的内存和实模式有很多共同之处。



毫无疑问，以'protected mode'(保护模式), 'global descriptor table'(全局描述符表), 'local descriptor table'(本地描述符表)和'selector'(选择器)搜索，你会得到完整介绍它们的大量信息。

保护模式与实模式的内存类似，然而，它们之间最大的区别就是保护模式的内存是“线性”的。

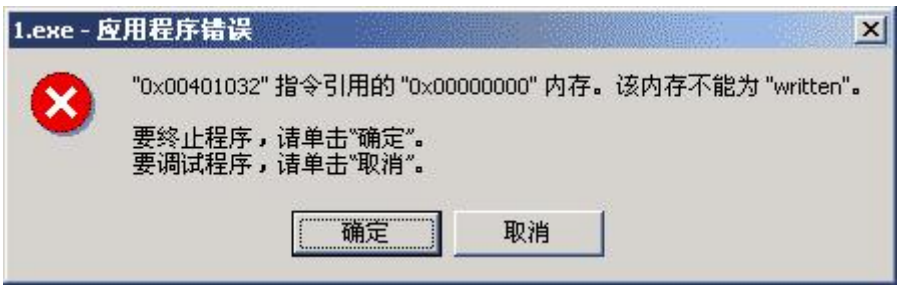
新的计算机上，32-bit 的寄存器已经不是什么新鲜事（如果你哪天听说你的 CPU 的寄存器不是 32-bit 的，那么它——简直可以肯定地说——的字长要比 32-bit 还要多。新的个人机上已经开始逐步采用 64-bit 的 CPU 了），换言之，实际上段/偏移量这一格局已经不再需要了。尽管如此，在继续看保护模式内存结构时，仍请记住段/偏移量的概念。不妨把段寄存器看作对于保护模式中的选择器的一个模拟。选择器是全局描述符表(Global Descriptor Table, GDT)或本地描述符表(Local Descriptor Table, LDT)的一个指针。

如图所示，GDT 和 LDT 的每一个项目都描述一块内存。例如，一个项目中包含了某块被描述的内存的物理的基地址、长度，以及其他一些相关信息。

保护模式是一个非常重要的概念，同时也是目前撰写应用程序时，最常用的 CPU 模式（运行在新的计算机上的操作系统很少有在实模式下运行的）。

为什么叫保护模式呢？它“保护”了什么？答案是进程的内存。保护模式的主要目的在于允许多个进程同时运行，并保护它们的内存不受其他进程的侵犯。这有点类似于 C++中的机制，然而它的强制力要大得多。如果你的进程在保护模式下以不恰当的方式访问了内存（例如，写了“只读”内存，或读了不可读的内存，等等），那么 CPU 就会产生一个异常。这个异常将交给操作系统处理，而这种处理，假如你的程序没有特别说明操作系统该如何处理的话，一般就是杀掉做错了事情的进程。

我像这样的对话框大家一定非常熟悉（临时写了一个程序故意造成的错误）：



好的，只是一个程序崩溃了，而操作系统的其他进程照常运行（同样的程序在 DOS 中几乎是板上钉钉的死机，因为 NULL 指针的位置恰好是中断向量表），你甚至还可以调试它。

保护模式还有其他很多好处，在此就不一一赘述了。实模式和保护模式之间的切换问题我打算放在后面的“高级技巧”一章来讲，因为多数程序并不涉及这个。

了解了内存的格局，我们就可以进入下一节——操作内存了。

3.3 操作内存

前两节中，我们介绍了实模式和保护模式中使用的不同的内存格局。现在开始解释如何使用这些知识。

回忆一下前面我们说过的，寄存器可以用作内存指针。现在，是他们发挥作用的时候了。

可以将内存想象为一个顺序的字节流。使用指针，可以任意地操作（读写）内存。

现在我们需要一些其他的指令格式来描述对于内存的操作。操作内存时，首先需要的就是它的地址。

让我们来看看下面的代码：

```
mov ax,[0]
```

方括号表示，里面的表达式指定的不是立即数，而是偏移量。在实模式中，DS:0 中的那个字（16-bit 长）将被装入 AX。

然而 0 是一个常数，如果需要在运行的时候加以改变，就需要一些特殊的技巧，比如程序自修改。汇编支持这个特性，然而我个人并不推荐这种方法——自修改大大降低程序的可读性，并且还降低稳定性，性能还不一定好。我们需要另外的技术。

```
mov bx,0
mov ax,[bx]
```

看起来舒服了一些，不是吗？BX 寄存器的内容可以随时更改，而不需要用冗长的代码去修改自身，更不用担心由此带来的不稳定问题。

同样的，mov 指令也可以把数据保存到内存中：

```
mov [0],ax
```

在存储器与寄存器之间交换数据应该足够清楚了。

有些时候我们会需要操作符来描述内存数据的宽度：

操作符	意义
byte ptr	一个字节(8-bit, 1 byte)
word ptr	一个字(16-bit)
dword ptr	一个双字(32-bit)

例如，在 DS:100h 处保存 1234h，以字存放：

```
mov word ptr [100h],01234h
```

于是我们将 mov 指令扩展为：

```
mov reg(8,16,32), mem(8,16,32)
mov mem(8,16,32), reg(8,16,32)
mov mem(8,16,32), imm(8,16,32)
```

需要说明的是，加减同样也可以在[]中使用，例如：

```
mov ax,[bx+10]
mov ax,[bx+si]
mov ax,es:[di+bp]
```

等等。我们看到，对于内存的操作，即使使用 MOV 指令，也有许多种可能的方式。下一节中，我们将介绍如何操作串。

3.4 串操作

我们前面已经提到，内存可以和寄存器交换数据，也可以被赋予立即数。问题是，如果我们需要把内存的某部分内容复制到另一个地址，又怎么做呢？

设想将 DS:SI 处的连续 512 字节内容复制到 ES:DI（先不考虑可能的重叠）。也许会有人写出这样的代码：

```
            mov cx,512                              ; 循环次数
NextByte:   mov al,ds:[si]
            mov es:[di],al
            inc si
            inc di
            loop NextByte
```

我不喜欢上面的代码。它的确能达到作用，但是，效率不好。如果你是在做优化，那么写出这样的代码意味着赔了夫人又折兵。

Intel 的 CPU 的强项是串操作。所谓串操作就是由 CPU 去完成某一数量的、重复的内存操作。需要说明的是，我们常用的 KMP 算法（用于匹配字符串中的模式）的改进——Boyer 算法，由于

没有利用串操作，因此在 Intel 的 CPU 上的效率并非最优。好的编译器往往可以利用 Intel CPU 的这一特性优化代码，然而，并非所有的时候它都能产生最好的代码。

某些指令可以加上 REP 前缀（repeat, 反复之意），这些指令通常被叫做串操作指令。

举例来说，STOSD 指令将 EAX 的内容保存到 ES:DI，同时在 DI 上加或减四。类似的，STOSB 和 STOSW 分别作 1 字节或 1 字的上述操作，在 DI 上加或减的数是 1 或 2。

计算机语言通常是不允许二义性的。为什么我要说“加或减”呢？没错，孤立地看 STOS?指令，并不能知道到底是加还是减，因为这取决于“方向”标志(DF, Direction Flag)。如果 DF 被复位，则加；反之则减。

置位、复位的指令分别是 STD 和 CLD。

当然，REP 只是几种可用前缀之一。常用的还包括 REPNE，这个前缀通常被用来比较两个串，或搜索某个特定字符（字、双字）。REPZ、REPE、REPNZ 也是非常常用的指令前缀，分别代表 ZF(Zero Flag)在不同状态时重复执行。

下面说三个可以复制数据的指令：

助记符	意义
movsb	将 DS:SI 的一字节复制到 ES:DI，之后 SI++、DI++
movsw	将 DS:SI 的一字节复制到 ES:DI，之后 SI+=2、DI+=2
movsd	将 DS:SI 的一字节复制到 ES:DI，之后 SI+=4、DI+=4

于是上面的程序改写为

```
cld                                    ; 复位 DF
mov cx, 128                            ; 512/4 = 128，共 128 个双字
rep movsd                              ; 行动！
```

第一句 cld 很多时候是多余的，因为实际写程序时，很少会出现置 DF 的情况。不过在正式决定删掉它之前，建议你仔细地调试自己的程序，并确认每一个能够走到这里的路径中都不会将 DF 置位。

错误（非预期的）的 DF 是危险的。它很可能断送掉你的程序，因为这直接造成**缓冲区溢出**问题。

什么是缓冲区溢出呢？缓冲区溢出分为两类，一类是写入缓冲区以外的内容，一类是读取缓冲区以外的内容。后一种往往更隐蔽，但随便哪一个都有可能断送掉你的程序。

缓冲区溢出对于一个网络服务来说很可能更加危险。怀有恶意的用户能够利用它执行自己希望的指令。服务通常拥有更高的特权，而这很可能会造成特权提升；即使不能提升攻击者拥有的特权，他也可以利用这种问题使服务崩溃，从而形成一次成功的 DoS（拒绝服务）攻击。每年 CERT 的安全公告中，都有 6 成左右的问题是由于缓冲区溢出造成的。

在使用汇编语言，或 C 语言编写程序时，很容易在无意中引入缓冲区溢出。然而并不是所有的语言都会引入缓冲区溢出问题，Java 和 C#，由于没有指针，并且缓冲区采取动态分配的方式，有效地消除了造成缓冲区溢出的土壤。

汇编语言中，由于 REP*前缀都用 CX 作为计数器，因此情况会好一些（当然，有时也会更糟糕，因为由于 CX 的限制，很可能使原本可能改变程序行为的缓冲区溢出的范围缩小，从而更为隐蔽）。避免缓冲区溢出的一个主要方法就是仔细检查，这包括两方面：设置合理的缓冲区大小，和根据大小编写程序。除此之外，非常重要的一点就是，在汇编语言这个级别写程序，你肯定希望去掉所有的无用指令，然而再去掉之前，一定要进行严格的测试；更进一步，如果能加上注释，并通过善用宏来做调试模式检查，往往能够达到更好的效果。

3.5 关于保护模式中内存操作的一点说明

正如 3.2 节提到到的那样，保护模式中，你可以使用 32 位的线性地址，这意味着直接访问 4GB 的内存。由于这个原因，选择器不用像实模式中段寄存器那样频繁地修改。顺便提一句，这份教程中所说的保护模式指的是 386 以上的保护模式，或者，Microsoft 通常称为“增强模式”的那种。

在为选择器装入数值的时候一定要非常小心。错误的数值往往会导致无效页面错误(在 Windows 中经常出现:)。同时，也不要忘记你的地址是 32 位的，这也是保护模式的主要优势之一。

现在假设存在一个描述符描述从物理的 0:0 开始的全部内存，并已经加载进 DS(数据选择器)，则我们可以通过下面的程序来操作 VGA 的 VRAM：

```
mov edi,0a0000h                    ; VGA 显存的偏移量
mov byte ptr [edi],0fh             ; 将第一字节改为 0fh
```

很明显，这比实模式下的程序

```
mov ax,0a000h                      ; AX -> VGA 段地址
mov ds,ax                          ; 将 AX 值载入 DS
mov di,0                           ; DI 清零
mov [di],0fh                       ; 修改第一字节
```

看上去要舒服一些。

3.6 堆栈

到目前为止，您已经了解了基本的寄存器以及内存的操作知识。事实上，您现在已经可以写出很多的底层数据处理程序了。

下面我来说说堆栈。堆栈实在不是一个让人陌生的数据结构，它是一个先进后出(FILO)的线性表，能够帮助你完成很多很好的工作。

先进后出(FILO)是这样一个概念：**最后**放进表中的数据在取出时**最先**出来。**先进后出**(FILO)和**先进先出**(FIFO, 和先进后出的规则相反)，以及**随机存取**是最主要的三种存储器访问方式。

对于堆栈而言，最后放入的数据在取出时最先出现。对于子程序调用，特别是递归调用来说，这是一个非常有用的特性。

一个铁杆的汇编语言程序员有时会发现系统提供的寄存器不够。很显然，你可以使用普通的内存操作来完成这个工作，就像 C/C++中所做的那样。

没错，没错，可是，如果数据段（数据选择器）以及偏移量发生变化怎么办？更进一步，如果希望保存某些在这种操作中可能受到影响的寄存器的时候怎么办？确实，你可以把他们也存到自己的那片内存中，自己实现堆栈。

太麻烦了……

既然系统提供了堆栈，并且性能比自己写一份更好，那么为什么不直接加以利用呢？

系统堆栈不仅仅是一段内存。由于 CPU 对它实施管理，因此你不需要考虑堆栈指针的修正问题。可以把寄存器内容，甚至一个立即数直接放到堆栈里，并在需要的时候将其取出。同时，系统并不要求取出的数据仍然回到原来的位置。

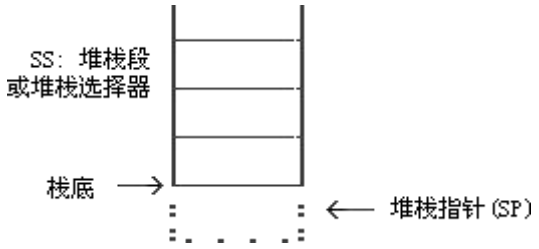
除了显式地操作堆栈（使用 PUSH 和 POP 指令）之外，很多指令也需要使用堆栈，如 INT、CALL、LEAVE、RET、RETF、IRET 等等。配对使用上述指令并不会造成什么问题，然而，如果你打算使用 LEAVE、RET、RETF、IRET 这样的指令实现跳转(比 JMP 更为麻烦，然而有时，例如在加密软件中，或者需要修改调用者状态时，这是必要的)的话，那么我的建议是，先搞清楚它们做的到底是什么，并且，精确地了解自己要做什么。

正如前面所说的，有两个显式地操作堆栈的指令：

助记符	功能
PUSH	将操作数存入堆栈，同时修正堆栈指针
POP	将栈顶内容取出并存到目的操作数中，同时修正堆栈指针

我们现在来看看堆栈的操作。

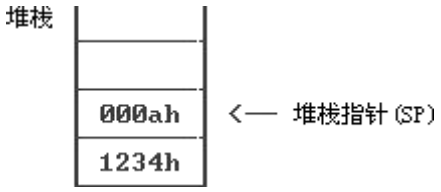
执行之前



执行代码

```
mov ax,1234h
mov bx,10
push ax
push bx
```

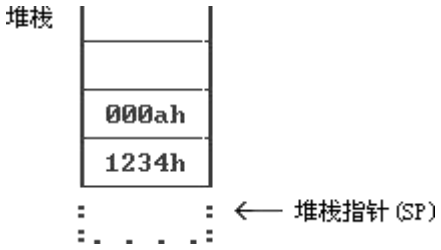
之后，堆栈的状态为



之后，再执行

```
pop dx
pop cx
```

堆栈的状态成为



当然，dx、cx 中的内容将分别是 000ah 和 1234h。

注意，最后这张图中，我没有抹去 1234h 和 000ah，因为 POP 指令并不从内存中抹去数值。不过尽管如此，我个人仍然非常反对继续使用这两个数（你可以通过修改 SP 来再次 POP 它们），然而这很容易导致错误。

一定要保证堆栈段有足够的空间来执行中断，以及其他一些隐式的堆栈操作。仅仅统计 PUSH 的数量并据此计算堆栈所需的大小很可能造成问题。

CALL 指令将返回地址放到堆栈中。绝大多数 C/C++编译器提供了“堆栈检查”这个编译选项，其作用在于保证 C 程序段中没有忘记对堆栈中多余的数据进行清理，从而保证返回地址有效。

本章小结

本章中介绍了内存的操作的一些入门知识。限于篇幅，我不打算展开细讲指令，如 cmps*，lods*，stos*，等等。这些指令的用法和前面介绍的 movs*基本一样，只是有不同的作用而已。

4.0 利用子程序与中断

已经掌握了汇编语言？没错，你现在已经可以去破译别人代码中的秘密。然而，我们还有一件重要的东西没有提到，那就是自程序和中断。这两件东西是如此的重要，以至于你的程序几乎不可能离开它们。

4.1 子程序

在高级语言中我们经常要用到子程序。高级语言中，子程序是如此的神奇，我们能够定义和主程序，或其他子程序一样的变量名，而访问不同的变量，并且，还不和程序的其他部分相冲突。

然而遗憾的是，这种“优势”在汇编语言中是不存在的。

汇编语言并不注重如何减轻程序员的负担；相反，汇编语言依赖程序员的良好设计，以期发挥 CPU 的最佳性能。汇编语言不是结构化的语言，因此，它不提供直接的“局部变量”。如果需要“局部变量”，只能通过堆或栈自行实现。

从这个意义上讲，汇编语言的子程序更像 GWBASIC 中的 GOSUB 调用的那些“子程序”。所有的“变量”(本质上，属于进程的内存和寄存器)为整个程序所共享，高级语言编译器所做的，将局部变量放到堆或栈中的操作，只能自行实现。

参数的传递是靠寄存器和堆栈来完成的。高级语言中，子程序(函数、过程，或类似概念的东西)依赖于堆和栈来传递。

让我们来简单地分析一下一般高级语言的子程序的执行过程。无论 C、C++、BASIC、Pascal，这一部分基本都是一致的。

- 调用者将子程序执行完成时应返回的地址、参数压入堆栈
- 子程序使用 BP 指针+偏移量对栈中的参数寻址，并取出、完成操作
- 子程序使用 RET 或 RETF 指令返回。此时，CPU 将 IP 置为堆栈中保存的地址，并继续予以执行

毋庸置疑，堆栈在整个过程中发挥着非常重要的作用。不过，本质上对子程序最重要的还是返回地址。如果子程序不知道这个地址，那么系统将会崩溃。

调用子程序的指令是 CALL，对应的返回指令是 RET。此外，还有一组指令，即 ENTER 和 LEAVE，它们可以帮助进行堆栈的维护。

CALL 指令的参数是被调用子程序的地址。使用宏汇编的时候，这通常是一个标号。CALL 和 RET，以及 ENTER 和 LEAVE 配对，可以实现对于堆栈的自动操作，而不需要程序员进行 PUSH/POP，以及跳转的操作，从而提高了效率。

作为一个编译器的实现实例，我用 Visual C++编译了一段 C++程序代码，这段汇编代码是使用特定的编译选项得到的结果，正常的 RELEASE 代码会比它精简得多。包含源代码的部分反汇编结果如下(取自 Visual C++调试器的运行结果，我删除了 10 条 int 3 指令，并加上了一些注释，除此之外，没有做任何修改)：

```
1: int myTransform(int nInput){
00401000 push ebp                    ; 保护现场原先的 EBP 指针
00401001 mov ebp,esp
2: return (nInput*2 + 3) % 7;
00401003 mov eax,dword ptr [nInput] ; 取参数
00401006 lea eax,[eax+eax+3]         ; LEA 比 ADD 加法更快
0040100A cdq                         ; DWORD->QWORD(扩展字长)
0040100B mov ecx,7                   ; 除数
00401010 idiv eax,ecx                ; 除
00401012 mov eax,edx                 ; 商->eax(eax 中保存返回值)
3: }
00401014 pop ebp                     ; 恢复现场的 ebp 指针
00401015 ret                         ; 返回
; 此处删除 10 条 int 3 指令，它们是方便调试用的，并不影响程序行为。
4:
5: int main(int argc, char* argv[])
6: {
00401020 push ebp                    ; 保护现场原先的 EBP 指针
00401021 mov ebp,esp
00401023 sub esp,10h                 ; 为取 argc, argv 修正堆栈指针。
7: int a[3];
8: for(register int i=0; i<3; i++){
00401026 mov dword ptr [i],0         ; 0->i
0040102D jmp main+18h (00401038)     ; 判断循环条件
0040102F mov eax,dword ptr [i]       ; i->eax
00401032 add eax,1                   ; eax ++
00401035 mov dword ptr [i],eax       ; eax->i
00401038 cmp dword ptr [i],3         ; 循环条件：i 与 3 比较
0040103C jge main+33h (00401053)     ; 如果不符合条件，则应结束循环
9: a[i] = myTransform(i);
```

```
0040103E mov ecx,dword ptr [i]       ; i->ecx
00401041 push ecx                    ; ecx (i) -> 堆栈
00401042 call myTransform (00401000) ; 调用 myTransform
00401047 add esp,4                   ; esp+=4：在堆中的新单元
                                     ; 准备存放返回结果
0040104A mov edx,dword ptr [i]       ; i->edx
0040104D mov dword ptr a[edx*4],eax  ; 将 eax(myTransform 返回值)
                                     ; 放回 a[i]
10: }
00401051 jmp main+0Fh (0040102f)     ; 计算 i++，并继续循环
11: return 0;
00401053 xor eax,eax                 ; 返回值应该是 0
12: }
00401055 mov esp,ebp                 ; 恢复堆栈指针
00401057 pop ebp                     ; 恢复 BP
00401058 ret                         ; 返回调用者(C++运行环境)
```

上述代码确实做了一些无用功，当然，这是因为编译器没有对这段代码进行优化。让我们来关注一下这段代码中，是如何调用子程序的。不考虑 myTransform 这个函数实际进行的数值运算，最让我感兴趣的是这一行代码：

```
00401003 mov eax,dword ptr [nInput] ; 取参数
```

这里 nInput 是一个简简单单的变量符号吗？Visual C++的调试器显然不能告诉我们答案——它的设计目标是为了方便程序调试，而不是向你揭示编译器生成的代码的实际构造。我用另外一个反汇编器得到的结果是：

```
00401003 mov eax,dword ptr [ebp+8]   ; 取参数
```

这和我们在 main()中看到的压栈顺序是完全吻合的(注意，程序运行到这个地方的时候，EBP=ESP)。main()最终将 i 的值通过堆栈传递给了 myTransform()。

剖析上面的程序只是说明了我前面所提到的子程序的一部分用法。对于汇编语言来说，完全没有必要拘泥于结构化程序设计的框架(在今天，使用汇编的主要目的在于提高执行效率，而不是方便程序的维护和调试，因为汇编不可能在这一点上做得比 C++更好)。考虑下面的程序：

```
void myTransform1(int nCount, char* sBytes){
  for(register int i=1; i<nCount; i++)
    sBytes[i] += sBytes[i-1];
  for(i=0; i<nCount; i++)
    sBytes[i] <<= 1;
}

void myTransform2(int nCount, char* sBytes){
  for(register int i=0; i<nCount; i++)
```

```
    sBytes[i] <<= 1;
}
```

很容易看出，这两个函数包含了公共部分，即

```
  for(i=0; i<nCount; i++)
    sBytes[i] <<= 1;
```

目前，还没有编译器能够做到将这两部分合并。依然沿用刚才的编译选项，得到的反汇编结果是(同样地删除了 int 3)：

```
1: void myTransform1(int nCount, char* sBytes){
00401000 push ebp
00401001 mov ebp,esp
00401003 push ecx
2: for(register int i=1; i<nCount; i++)
00401004 mov dword ptr [i],1
0040100B jmp myTransform1+16h (00401016)
0040100D mov eax,dword ptr [i]
00401010 add eax,1
00401013 mov dword ptr [i],eax
00401016 mov ecx,dword ptr [i]
00401019 cmp ecx,dword ptr [nCount]
0040101C jge myTransform1+3Dh (0040103d)
3: sBytes[i] += sBytes[i-1];
0040101E mov edx,dword ptr [sBytes]
00401021 add edx,dword ptr [i]
00401024 movsx eax,byte ptr [edx-1]
00401028 mov ecx,dword ptr [sBytes]
0040102B add ecx,dword ptr [i]
0040102E movsx edx,byte ptr [ecx]
00401031 add edx,eax
00401033 mov eax,dword ptr [sBytes]
00401036 add eax,dword ptr [i]
00401039 mov byte ptr [eax],dl
0040103B jmp myTransform1+0Dh (0040100d)
4: for(i=0; i<nCount; i++)
0040103D mov dword ptr [i],0
00401044 jmp myTransform1+4Fh (0040104f)
00401046 mov ecx,dword ptr [i]
00401049 add ecx,1
0040104C mov dword ptr [i],ecx
0040104F mov edx,dword ptr [i]
00401052 cmp edx,dword ptr [nCount]
00401055 jge myTransform1+6Bh (0040106b)
```

```
5: sBytes[i] <<= 1;
00401057 mov eax,dword ptr [sBytes]
0040105A add eax,dword ptr [i]
0040105D mov cl,byte ptr [eax]
0040105F shl cl,1
00401061 mov edx,dword ptr [sBytes]
00401064 add edx,dword ptr [i]
00401067 mov byte ptr [edx],cl
00401069 jmp myTransform1+46h (00401046)
6: }
0040106B mov esp,ebp
0040106D pop ebp
0040106E ret
7:
8: void myTransform2(int nCount, char* sBytes){
00401070 push ebp
00401071 mov ebp,esp
00401073 push ecx
9: for(register int i=0; i<nCount; i++)
00401074 mov dword ptr [i],0
0040107B jmp myTransform2+16h (00401086)
0040107D mov eax,dword ptr [i]
00401080 add eax,1
00401083 mov dword ptr [i],eax
00401086 mov ecx,dword ptr [i]
00401089 cmp ecx,dword ptr [nCount]
0040108C jge myTransform2+32h (004010a2)
10: sBytes[i] <<= 1;
0040108E mov edx,dword ptr [sBytes]
00401091 add edx,dword ptr [i]
00401094 mov al,byte ptr [edx]
00401096 shl al,1
00401098 mov ecx,dword ptr [sBytes]
0040109B add ecx,dword ptr [i]
0040109E mov byte ptr [ecx],al
004010A0 jmp myTransform2+0Dh (0040107d)
11: }
004010A2 mov esp,ebp
004010A4 pop ebp
004010A5 ret
12:
13: int main(int argc, char* argv[])
14: {
004010B0 push ebp
```

```
004010B1 mov ebp,esp
004010B3 sub esp,0CCh
15: char a[200];
16: for(register int i=0; i<200; i++)a[i]=i;
004010B9 mov dword ptr [i],0
004010C3 jmp main+24h (004010d4)
004010C5 mov eax,dword ptr [i]
004010CB add eax,1
004010CE mov dword ptr [i],eax
004010D4 cmp dword ptr [i],0C8h
004010DE jge main+45h (004010f5)
004010E0 mov ecx,dword ptr [i]
004010E6 mov dl,byte ptr [i]
004010EC mov byte ptr a[ecx],dl
004010F3 jmp main+15h (004010c5)
17: myTransform1(200, a);
004010F5 lea eax,[a]
004010FB push eax
004010FC push 0C8h
00401101 call myTransform1 (00401000)
00401106 add esp,8
18: myTransform2(200, a);
00401109 lea ecx,[a]
0040110F push ecx
00401110 push 0C8h
00401115 call myTransform2 (00401070)
0040111A add esp,8
19: return 0;
0040111D xor eax,eax
20: }
0040111F mov esp,ebp
00401121 pop ebp
00401122 ret
```

非常明显地，0040103d-0040106e 和 00401074-004010a5 这两段代码存在少量的差别，但很显然只是对寄存器的偏好不同(编译器在优化时，这可能会减少堆栈操作，从而提高性能，但在这里只是使用了不同的寄存器而已)

对代码进行合并的好处是非常明显的。新的操作系统往往使用页式内存管理。当内存不足时，程序往往会频繁引发页面失效(Page faults)，从而引发操作系统从磁盘中读取一些东西。磁盘的速度赶不上内存的速度，因此，这一行为将导致性能的下降。通过合并一部分代码，可以减少程序的大小，这意味着减少页面失效的可能性，从而软件的性能会有所提高?/p>

当然，这样做的代价也不算低——你的程序将变得难懂，并且难于维护。因此，再进行这样的优化之前，一定要注意：

- 优化前的程序必须是正确的。如果你不能确保这一点，那么这种优化必将给你的调试带来极大的麻烦。
- 优化前的程序实现最好是最优的。仔细检查你的设计，看看是否已经使用了最合适(即，对于此程序而言最优)的算法，并且已经在高级语言许可的范围内进行了最好的实现。
- 优化最好能够非常有效地减少程序大小(例如，如果只是减少十几个字节，恐怕就没什么必要了)，或非常有效地提高程序的运行速度(如果代码只是运行一次，并且只是节省几个时钟周期，那么在多数场合都没有意义)。否则，这种优化将得不偿失。

4.2 中断

中断应该说是一个陈旧的话题。在新的系统中，它的作用正在逐渐被削弱，而变成操作系统专用的东西。并不是所有的计算机系统都提供中断，然而在 x86 系统中，它的作用是不可替代的。

中断实际上是一类特殊的子程序。它通常由系统调用，以响应突发事件。

例如，进行磁盘操作时，为了提高性能，可能会使用 DMA 方式进行操作。CPU 向 DMA 控制器发出指令，要求外设和内存直接交换数据，而不通过 CPU。然后，CPU 转去进行起他的操作；当数据交换结束时，CPU 可能需要进行一些后续操作，但此时它如何才能知道 DMA 已经完成了操作呢？

很显然不是依靠 CPU 去查询状态——这样 DMA 的优势就不明显了。为了尽可能地利用 DMA 的优势，在完成 DMA 操作的时候，DMA 会告诉 CPU“这事儿我办完了”，然后 CPU 会根据需要进行处理。

这种处理可能很复杂，需要若干条指令来完成。子程序是一个不错的主意，不过，CALL 指令需要指定地址，让外设强迫 CPU 执行一条 CALL 指令也违背了 CPU 作为核心控制单元的设计初衷。考虑到这些，在 x86 系统中引入了中断向量的概念。

中断向量表是保存在系统数据区(实模式下，是 0:0 开始的一段区域)的一组指针。这组指针指向每一个中断服务程序的地址。整个中断向量表的结构是一个线性表。

每一个中断服务有自己的唯一的编号，我们通常称之为中断号。每一个中断号对应中断向量表中的一项，也就是一个中断向量。外设向 CPU 发出中断请求，而 CPU 自己将根据当前的程序状态决定是否中断当前程序并调用相应的中断服务。

不难根据造成中断的原因将中断分为两类：硬件中断和软件中断。硬件中断有很多分类方法，如根据是否可以屏蔽分类、根据优先级高低分类，等等。考虑到这些分类并不一定科学，并且对于我们介绍中断的使用没有太大的帮助，因此我并不打算太详细地介绍它(在本教程的高级篇中，关于加密解密的部分会提到某些硬件中断的利用，但那是后话)。

在设计操作系统时，中断向量的概念曾经带来过很大的便利。操作系统随时可能升级，这样，通过 CALL 来调用操作系统的服务(如果说每个程序都包含对于文件系统、进程表这些应该由操作

系统管理的数据的直接操作的话，不仅会造成程序的臃肿，而且不利于系统的安全)就显得不太合适了——没人能知道，以后的操作系统的服务程序入口点会不会是那儿。软件中断的存在为解决这个问题提供了方便。

对于一台包含了 BIOS 的计算机来说，启动的时候系统已经提供了一部分服务，例如显示服务。无论你的 BIOS、显示卡有多么的“个性”，只要他们和 IBM PC 兼容，那么此时你肯定可以通过调用 16(10h)号中断来使用显示服务。调用中断的指令是

```
int 中断号
```

这将引发 CPU 去调用一个中断。CPU 将保存当前的程序状态字，清除 Trap 和 Interrupt 两个标志，将即将执行的指令地址压入堆栈，并调用中断服务(根据中断向量表)。

编写中断服务程序不是一件容易的事情。很多时候，中断服务程序必须写成**可重入代码**(或纯代码，pure code)。所谓可重入代码是指，程序的运行过程中可以被打断，并由开始处再次执行，并且在合理的范围内(多次重入，而不造成堆栈溢出等其他问题)，程序可以在被打断处继续执行，并且执行结果不受影响。

由于在多线程环境中等其他一些地方进行程序设计时也需要考虑这个因素，因此这里着重讲一下可重入代码的编写。

可重入代码最主要的要求就是，程序不应使用某个指定的内存地址的内存(对于高级语言来说，这通常是全局变量，或对象的成员)。如果可能的话，应使用寄存器，或其他方式来解决。如果不能做到这一点，则必须在开始、结束的时候分别禁止和启用中断，并且，运行时间不能太长。

下面用 C 语言分别举一个可重入函数，和两个非可重入函数的例子(注. 这些例子应该是在某本多线程或操作系统的书上看到的，遗憾的是我想不起来是哪本书了，在这里先感谢那位作者提供的范例)：

可重入函数：

```
void strcpy(char* lpszDest, char* lpszSrc){
  while(*dest++=*src++);
  *dest=0;
}
```

非可重入函数

```
char cTemp;                                    // 全局变量

void SwapChar(char* lpcX, char* lpcY){
  cTemp = *lpcX; *lpcX = *lpcY; lpcY = cTemp; // 引用了全局变量，在分享内存的多个线程中可能造成问题
```

```
}
```

非可重入函数

```
void SwapChar2(char* lpcX, char* lpcY){
  static char cTemp;                              // 静态变量
  cTemp = *lpcX; *lpcX = *lpcY; lpcY = cTemp; // 引用了静态变量，在分享内存的多个线程中可能造成问题
}
```

中断利用的是系统的栈。栈操作是可重入的(因为栈可以保证“先进后出”)，因此，我们并不需要考虑栈操作的重入问题。使用宏汇编器写出可重入的汇编代码需要注意一些问题。简单地说，干脆不要用标号作为变量是一个不错的主意。

使用高级语言编写可重入程序相对来讲轻松一些。把持住不访问那些全局(或当前对象的)变量，不使用静态局部变量，坚持只适用局部变量，写出的程序就将是可重入的。

书归正传，调用软件中断时，通常都是通过寄存器传进、传出参数。这意味着你的 int 指令周围也许会存在一些“帮手”，比如下面的代码：

```
mov ax, 4c00h
int 21h
```

就是通过调用 DOS 中断服务返回父进程，并带回错误反馈码 0。其中，ax 中的数据 4c00h 就是传递给 DOS 中断服务的参数。

到这里，x86 汇编语言的基础部分就基本上讲完了，《简明 x86 汇编语言教程》的初级篇——汇编语言基础也就到此告一段落。当然，目前为止，我只是蜻蜓点水一般提到了一些学习 x86 汇编语言中我认为需要注意的重要概念。许多东西，包括全部汇编语句的时序特性(指令执行周期数，以及指令周期中各个阶段的节拍数等)、功能、参数等等，限于个人水平和篇幅我都没有作详细介绍。如果您对这些内容感兴趣，请参考 Intel 和 AMD 两大 CPU 供应商网站上提供的开发人员参考。

在以后的简明 x86 汇编语言教程中级篇和高级篇中，我将着重介绍汇编语言的调试技术、优化，以及一些具体的应用技巧，包括反跟踪、反反跟踪、加密解密、病毒与反病毒等等。

5.0 编译优化概述

优化是一件非常重要的事情。作为一个程序设计者，你肯定希望自己的程序既小又快。DOS 时代的许多书中都提到，“某某编译器能够生成非常紧凑的代码”，换言之，编译器会为你把代码尽可能地缩减，如果你能够正确地使用它提供的功能的话。目前，Intel x86 体系上流行的 C/C++ 编译器，包括 Intel C/C++ Compiler, GNU C/C++ Compiler，以及最新的 Microsoft 和 Borland 编译器，都能够提供非常紧凑的代码。正确地使用这些编译器，则可以得到性能足够好的代码。

但是，机器目前还不能像人那样做富于创造性的事情。因而，有些时候我们可能会不得不手工来做一些事情。

使用汇编语言优化代码是一件困难，而且技巧性很强的工作。很多编译器能够生成为处理器进行过特殊优化处理的代码，一旦进行修改，这些特殊优化可能就会被破坏而失效。因此，在你决定使用自己的汇编代码之前，一定要测试一下，到底是编译器生成的那段代码更好，还是你的更好。

本章中将讨论一些编译器在某些时候会做的事情(从某种意义上说，本章内容更像是计算机专业的基础课中《编译程序设计原理》、《计算机组成原理》、《计算机体系结构》课程中的相关内容)。本章的许多内容和汇编语言程序设计本身关系并不是很紧密，它们多数是在为使用汇编语言进行优化做准备。编译器确实做这些优化，但它并不总是这么做；此外，就编译器的设计本质来说，它确实没有义务这么做——编译器做的是等义变换，而不是等效变换。考虑下面的代码：

```
// 程序段 1
int gaussianSum(){
  int i, j=0;

  for(i=0; i<100; i++) j+=i;

  return j;
}
```

好的，首先，绝大多数编译器恐怕不会自作主张地把它“篡改”为

```
// 程序段 1(改进 1)
int gaussianSum(){
  int i, j=0;

  for(i=1; i<100; i++) j+=i;

  return j;
}
```

多数（但确实不是全部）编译器也不会把它改为

```
// 程序段 1(改进 2)
inline int gaussianSum(){
  return 5050;
}
```

这两个修改版本都不同于原先程序的语义。首先我们看到，让 i 从 0 开始是没有必要的，因为 j+=i 时，i=0 不会做任何有用的事情；然后是，实际上没有必要每一次都计算 1+...+100 的和——它可以被预先计算，并在需要的时候返回。

这个例子也许并不恰当(估计没人会写出最初版本那样的代码)，但这种实践在程序设计中确实可能出现。我们把改进 2 称为**编译时表达式预先计算**，而把改进 1 成为**循环强度削减**。

然而，一些新的编译器的确会进行这两种优化。不过别慌，看看下面的代码：

```
// 程序段 2
int GetFactorial(int k){
  int i, j=1;

  if((k<0) || (k>=10)) return -1;

  if((k<=1)) return 1

  for(i=1; i<k; i++) j*=i;

  return j;
}
```

程序采用的是一个时间复杂度为 O(n)的算法，不过，我们可以把他轻易地改为 O(1)的算法：

```
// 程序段 2 (非规范改进)
int GetFactorial(int k){
  int i, j=1;

  static const int FractorialTable[]={1, 1, 2, 6, 24,
    120, 720, 5040, 40320, 362880, 3628800};

  if((k<0) || (k>=10)) return -1;

  return FractorialTable[k];
}
```

这是一个典型的以空间换时间的做法。通用的编译器不会这么做——因为它没有办法在编译时确定你是不是要这么改。可以说，如果编译器真的这样做的话，那将是一件可怕的事情，因为那时候你将很难知道编译器生成的代码和自己想的到底有多大的差距。

当然，这类优化超出了本文的范围——基本上，我把它们归入“算法优化”，而不是“程序优化”一类。类似的优化过程需要程序设计人员对于程序逻辑非常深入地了解和全盘的掌握，同时，也需要有丰富的算法知识。

自然，如果你希望自己的程序性能有大幅度的提升，那么首先应该做的是算法优化。例如，把一个 $O(n^2)$的算法替换为一个 $O(n)$的算法，则程序的性能提升将远远超过对于个别语句的修改。此外，一个已经改写为汇编语言的程序，如果要再在算法上作大幅度的修改，其工作量将和重写相当。因此，在决定使用汇编语言进行优化之前，必须首先考虑算法优化。但假如已经是最优的算法，程序运行速度还是不够快怎么办呢？

好的，现在，假定你已经使用了已知最好的算法，决定把它交给编译器，让我们来看看编译器会为我们做什么，以及我们是否有机会插手此事，做得更好。

5.1 循环优化：强度削减和代码外提

比较新的编译器在编译时会自动把下面的代码：

```
for(i=0; i<10; i++){
  j = i;
  k = j + i;
}
```

至少变换为

```
for(i=0; i<10; i++);
j=i; k=j+i;
```

甚至

```
j=i=10; k=20;
```

当然，真正的编译器实际上是在中间代码层次作这件事情。

原理 如果数据项的某个中间值(程序执行过程中的计算结果)在使用之前被另一中间值覆盖，则相关计算不必进行。

也许有人会问，编译器不是都给咱们做了吗，管它做什么？注意，这里说的只是编译系统中优化部分的基本设计。不仅在从源代码到中间代码的过程中存在优化问题，而且编译器生成的最终的机器语言(汇编)代码同样存在类似的问题。目前，几乎所有的编译器在最终生成代码的过程中都有或多或少的瑕疵，这些瑕疵目前只能依靠手工修改代码来解决。

5.2 局部优化：表达式预计算和子表达式提取

表达式预先计算非常简单，就是在编译时尽可能地计算程序中需要计算的东西。例如，你可以毫不犹豫地写出下面的代码：

```
const unsigned long nGiga = 1024L * 1024L * 1024L;
```

而不必担心程序每次执行这个语句时作两遍乘法，因为编译器会自动地把它改为

```
const unsigned long nGiga = 1073741824L;
```

而不是傻乎乎地让计算机在执行到这个初始化赋值语句的时候才计算。当然，如果你愿意在上面的代码中掺上一些变量的话，编译器同样会把常数部分先行计算，并拿到结果。

表达式预计算并不会让程序性能有飞跃性的提升，但确实减少了运行时的计算强度。除此之外，绝大多数编译器会把下面的代码：

```
// [假设此时 b, c, d, e, f, g, h 都有一个确定的非零整数值，并且，
// a[]为一个包括 5 个整数元素的数组，其下标为 0 到 4]

a[0] = b*c;
a[1] = b+c;
a[2] = d*e;
a[3] = b*d + c*d;
a[4] = b*d*e + c*d*e;
```

优化为(再次强调，编译器实际上是在中间代码的层次，而不是源代码层次做这件事情！)：

```
// [假设此时 b, c, d, e, f, g, h 都有一个确定的非零整数值，并且，
// a[]为一个包括 5 个整数元素的数组，其下标为 0 到 4]

a[0] = b*c;
a[1] = b+c;
a[2] = d*e;
a[3] = a[1] * d;
a[4] = a[3] * e;
```

更进一步，在实际代码生成过程中，一些编译器还会对上述语句的次序进行调整，以使其运行效率更高。例如，将语句调整为下面的次序：

```
// [假设此时 b, c, d, e, f, g, h 都有一个确定的非零整数值，并且，
// a[]为一个包括 5 个整数元素的数组，其下标为 0 到 4]

a[0] = b*c;
a[1] = b+c;
a[3] = a[1] * d;
a[4] = a[3] * e;
a[2] = d*e;
```

在某些体系结构中，刚刚计算完的 a[1]可以放到寄存器中，以提高实际的计算性能。上述 5 个计算任务之间，只有 1, 3, 4 三个计算任务必须串行地执行，因此，在新的处理器上，这样做甚至能够提高程序的并行度，从而使程序效率变得更高。

5.3 全局寄存器优化

[待修订内容] 本章中，从这一节开始的所有优化都是在微观层面上的优化了。换言之，这些优化是不能使用高级语言中的对应设施进行解释的。这一部分内容将进行较大规模的修订。

通常，此类优化是由编译器自动完成的。我个人并不推荐真的由人来完成这些工作——这些工作多半是枯燥而重复性的，编译器通常会比人做得更好(没说的，肯定也更快)。但话说回来，使用汇编语言的程序设计人员有责任了解这些内容，因为只有这样才能更好地驾驭处理器。

在前面的几章中我已经提到过，寄存器的速度要比内存快。因此，在使用寄存器方面，编译器一般会做一种称为全局寄存器优化的优化。

例如，在我们的程序中使用了 4 个变量：i, j, k, l。它们都作为循环变量使用：

```
for(i=0; i<1000; i++){
  for(j=0; j<1000; j++){
    for(k=0; k<1000; k++){
      for(l=0; l<1000; l++)
        do_something(i, j, k, l);
    }
  }
}
```

这段程序的优化就不那么简单了。显然，按照通常的压栈方法，i, j, k, l 应该按照某个顺序被压进堆栈，然后调用 do_something()，然后函数做了一些事情之后返回。问题在于，无论如何压栈，这些东西大概都得进内存(不可否认某些机器可以用 CPU 的 Cache 做这件事情，但 Cache 是写通式的和回写式的又会造成一些性能上的差异)。

聪明的读者马上就会指出，我们不是可以在定义 do_something()的时候加上 inline 修饰符，让它在本地展开吗？没错，本地展开以增加代码量为代价换取性能，但这只是问题的一半。编译器尽管完成了本地展开，但它仍然需要做许多额外的工作。因为寄存器只有那么有限的几个，而我们却有这么多的循环变量。

把四个变量按照它们在循环中使用的频率排序，并决定在 do_something()块中的优先顺序(放入寄存器中的优先顺序)是一个解决方案。很明显，我们可以按照 l, k, j, i 的顺序(从高到低，因为 l 将被进行 1000*1000*1000*1000 次运算！)来排列，但在实际的问题中，事情往往没有这么简单，因为你不知道 do_something()中做的到底是什么。而且，凭什么就以 for(l=0; l<1000; l++)作为优化的分界点呢？如果 do_something()中还有循环怎么办？

如此复杂的计算问题交给计算机来做通常会有比较满意的结果。一般说来，编译器能够对程序中变量的使用进行更全面地估计，因此，它分配寄存器的结果有时虽然让人费解，但却是最优的(因为计算机能够进行大量的重复计算，并找到最好的方法；而人做这件事相对来讲比较困难)。

编译器在许多时候能够作出相当让人满意的结果。考虑以下的代码：

```
int a=0;
```

```
for(int i=1; i<10; i++)
  for(int j=1; j<100; j++){
    a += (i*j);
  }
```

让我们把它变为某种形式的中间代码：

```
00: 0 -> a
01: 1 -> i
02: 1 -> j
03: i*j -> t
04: a+t -> a
05: j+1 -> j
06: evaluate j < 100
07: TRUE? goto 03
08: i+1 -> i
09: evaluate i < 10
10: TRUE? goto 02
11: [继续执行程序的其余部分]
```

程序中执行强度最大的无疑是 03 到 05 这一段，涉及的需要写入的变量包括 a, j；需要读出的变量是 i。不过，最终的编译结果大大出乎我们的意料。下面是某种优化模式下 Visual C++ 6.0 编译器生成的代码(我做了一些修改)：

```
xor eax, eax                ; a=0(eax: a)
mov edx, 1                  ; i=1(edx: i)
push esi                    ; 保存 esi(最后要恢复，esi 作为代替 j 的那个循环变量)
nexti:
mov ecx, edx                ; [t=i]
mov esi, 999                ; esi=999: 此处修改了原程序的语义，但仍为 1000 次循环。
nextj:
add eax, ecx                ; [a+=t]
add ecx, edx                ; [t+=i]
dec esi                     ; j--
jne SHORT nextj             ; jne 等价于 jnz. [如果还需要，则再次循环]
inc edx                     ; i++
cmp edx, 10                 ; i 与 10 比较
jl SHORT nexti              ; i < 10, 再次循环
pop esi                     ; 恢复 esi
```

这段代码可能有些令人费解。主要是因为它不仅使用了大量寄存器，而且还包括了 5.2 节中曾提到的子表达式提取技术。表面上看，多引入的那个变量(t)增加了计算时间，但要注意，这个 t 不仅不会降低程序的执行效率，相反还会让它变得更快！因为同样得到了计算结果(本质上，i*j 即

是第 j 次累加 i 的值)，但这个结果不仅用到了上次运算的结果，而且还省去了乘法(很显然计算机计算加法要比计算乘法快)。

这里可能会有人问，为什么要从 999 循环到 0，而不是按照程序中写的那样从 0 循环到 999 呢？这个问题和汇编语言中的取址有关。在下两节中我将提到这方面的内容。

5.4 x86 体系结构上的并行最大化和指令封包

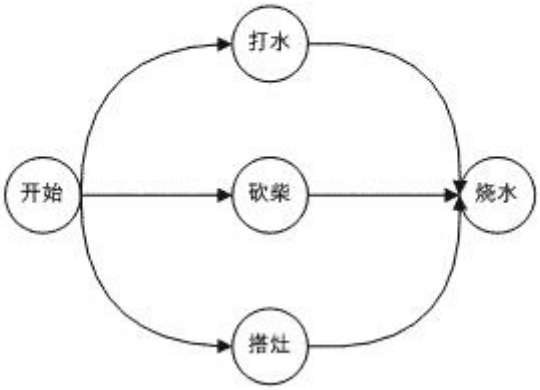
考虑这样的问题，我和两个同伴现在在山里，远处有一口井，我们带着一口锅，身边是树林；身上的饮用水已经喝光了，此处允许砍柴和使用明火(当然我们不想引起火灾:)，需要烧一锅水，应该怎么样呢？

一种方案是，三个人一起搭灶，一起砍柴，一起打水，一起把水烧开。

另一种方案是，一个人搭灶，此时另一个人去砍柴，第三个人打水，然后把水烧开。

这两种方案画出图来是这样：

方案1



方案2

仅仅这样很难说明两个方案孰优孰劣，因为我们并不明确三个人一起打水、一起砍柴、一起搭灶的效率更高，还是分别作效率更高(通常的想法，一起做也许效率会更高)。但假如说，三个人一个只会搭灶，一个只会砍柴，一个只会打水(当然是说这三件事情)，那么，方案 2 的效率就会搞一些了。

在现实生活中，某个人拥有专长是比较普遍的情况；在设计计算机硬件的时候则更是如此。你不可能指望加法器不做任何改动就能去做移位甚至整数乘法，然而我们注意到，串行执行的程序不可能在同一时刻同时用到处理器的所有功能，因此，我们(很自然地)会希望有一些指令并行地执行，以充分利用 CPU 的计算资源。

CPU 执行一条指令的过程基本上可以分为下面几个阶段：取指令、取数据、计算、保存数据。假设这 4 个阶段各需要 1 个时钟周期，那么，只要资源够用，并且 4 条指令之间不存在串行关系(换言之这些指令的执行先后次序不影响最终结果，或者，更严格地说，没有任何一条指令依赖其他指令的运算结果)指令也可以像下面这样执行：

指令 1	取指令	取数据	计　算	存数据			
指令 2		取指令	取数据	计　算	存数据		
指令 3			取指令	取数据	计　算	存数据	
指令 4				取指令	取数据	计　算	存数据

这样，原本需要 16 个时钟周期才能够完成的任务就可以在 7 个时钟周期内完成，时间缩短了一半还多。如果考虑灰色的那些方格(这些方格可以被 4 条指令以外的其他指令使用，只要没有串行关系或冲突)，那么，如此执行对于性能的提升将是相当可观的(此时，CPU 的所有部件都得到了充分利用)。

当然，作为程序来说，真正做到这样是相当理想化的情况。实际的程序中很难做到彻底的并行化。假设 CPU 能够支持 4 条指令同时执行，并且，每条指令都是等周期长度的 4 周期指令，那么，程序需要保证同一时刻先后发射的 4 条指令都能够并行执行，相互之间没有关联，这通常是不太可能的。

最新的 Intel Pentium 4-XEON 处理器，以及 Intel Northwood Pentium 4 都提供了一种被称为超线程(Hyper-Threading ™)的技术。该技术通过在一个处理器中封装两组执行机构来提高指令并行度，并依靠操作系统的调度来进一步提升系统的整体效率。

由于线程机制是与操作系统密切相关的，因此，在本文的这一部分中不可能做更为深入地探讨。在后续的章节中，我将介绍 Win32、FreeBSD 5.x 以及 Linux 中提供的内核级线程机制(这三种操作系统都支持 SMP 及超线程技术，并且以线程作为调度单位)在汇编语言中的使用方法。

关于线程的讨论就此打住，因为它更多地依赖于操作系统，并且，无论如何，操作系统的线程调度需要更大的开销并且，到目前为止，真正使用支持超线程的 CPU，并且使用相应操作系统的人是非常少的。因此，我们需要关心的实际上还是同一执行序列中的并发执行和指令封包。不过，令人遗憾的是，实际上在这方面编译器做的几乎是肯定要比人好，因此，你需要做的只是开启相应的优化；如果你的编译器不支持这样的特性，那么就把它扔掉……据我所知，目前在 Intel 平台上指令封包方面做的最好的是 Intel 的 C++编译器，经过 Intel 编译器编译的代码的性能令人惊异地高，甚至在 AMD 公司推出的兼容处理器上也是如此。

5.5 存储优化

从前一节的图中我们不难看出，方案 2 中，如果谁的动作慢，那么他就会成为性能的瓶颈。实际上，CPU 也不会像我描述的那样四平八稳地运行，指令执行的不同阶段需要的时间(时钟周期数)是不同的，因此，缩短关键步骤(即，造成瓶颈的那个步骤)是缩短执行时间的关键。

至少对于使用 Intel 系列的 CPU 来说，取数据这个步骤需要消耗比较多的时间。此外，假如数据跨越了某种边界(如 4 或 8 字节，与 CPU 的字长有关)，则 CPU 需要启动两次甚至更多次数的读内存操作，这无疑对性能构成不利影响。

基于这样的原因，我们可以得到下面的设计策略：

程序设计中的内存数据访问策略

- **尽可能减少对于内存的访问。在不违背这一原则的前提下，如果可能，将数据一次处理完。**
- **尽可能将数据按 4 或 8 字节对齐，以利于 CPU 存取**
- **尽可能一段时间内访问范围不大的一段内存，而不同时访问大量远距离的分散数据，以利于 Cache 缓存***

第一条规则比较简单。例如，需要求一组数据中的最大值、最小值、平均数，那么，最好是在一次循环中做完。

“于是，这家伙又攒了一段代码”……

```
int a[]={1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0};
int i;
int avg, max, min;

avg=max=min=a[0];

for(i=1; i<(sizeof(a)/sizeof(int)); i++){
  avg+=a[i];
  if(max < a[i])
    max = a[i];
  else if(min > a[i])
    min = a[i];
}

avg /= i;
```

Visual C++编译器把最开始一段赋值语句翻译成了一段简直可以说是匪夷所思的代码：

```
; int a[]={1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0};

mov edi, 2                          ; 此时 edi 没有意义
mov esi, 3                          ; esi 也是！临时变量而已。
mov DWORD PTR _a$[esp+92], edi
mov edx, 5                          ; 黑名单加上 edx
```

```
mov eax, 7                          ; eax 也别跑:)
mov DWORD PTR _a$[esp+132], edi
mov ecx, 9                          ; 就差你了，ecx

; int i;
; int avg, max, min;
; avg=max=min=a[0];

mov edi, 1                          ; edi 摇身一变，现在它是 min 了。
mov DWORD PTR _a$[esp+96], esi
mov DWORD PTR _a$[esp+104], edx
mov DWORD PTR _a$[esp+112], eax
mov DWORD PTR _a$[esp+136], esi
mov DWORD PTR _a$[esp+144], edx
mov DWORD PTR _a$[esp+152], eax
mov DWORD PTR _a$[esp+88], 1        ; 编译器失误？此处 edi 应更好
mov DWORD PTR _a$[esp+100], 4
mov DWORD PTR _a$[esp+108], 6
mov DWORD PTR _a$[esp+116], 8
mov DWORD PTR _a$[esp+120], ecx
mov DWORD PTR _a$[esp+124], 0
mov DWORD PTR _a$[esp+128], 1
mov DWORD PTR _a$[esp+140], 4
mov DWORD PTR _a$[esp+148], 6
mov DWORD PTR _a$[esp+156], 8
mov DWORD PTR _a$[esp+160], ecx
mov DWORD PTR _a$[esp+164], 0
mov edx, edi                        ; edx 是 max。
mov eax, edi                        ; 期待已久的 avg，它被指定为 eax
```

这段代码是最优的吗？我个人认为不是。因为编译器完全可以在编译过程中直接把它们作为常量数据放入内存。此外，如果预先对 a[0..9]10 个元素赋值，并利用串操作指令(rep movsdw)，速度会更快一些。

当然，犯不上因为这些问题责怪编译器。要求编译器知道 a[0..9]和[10..19]的内容一样未免过于苛刻。我们看看下面的指令段：

```
; for(i=1; ...

mov esi, edi                    ; esi: i
for_loop:

; avg+=a[i];
```

```
mov     ecx,     DWORD     PTR ; ecx: 暂存变量, =a[i]
_a$[esp+esi*4+88]              ; eax: avg
add eax, ecx

; if(max < a[i])
                               ; edx: max
cmp edx, ecx
jge SHORT elseif_min

; max = a[i];

mov edx, ecx

; else if(min > a[i])
                               ; 有趣的代码...并不是所有的时候都有用
jmp SHORT elseif_min           ; 但是也别随便删除
elseif_min:                    ; edi: min
cmp edi, ecx
jle SHORT elseif_end

; min = a[i];
mov edi, ecx

elseif_end:

; [for i=1]; i<20; i++){
                               ; i++
inc esi                        ; i 与 20 比较
cmp esi, 20
jl SHORT for_loop

; }
; avg /= i;

cdq                            ; avg /= i
idiv esi
```

上面的程序倒是没有什么惊人之处。唯一一个比较吓人的东西是那个 jmp SHORT 指令，它是否有用取决于具体的问题。C/C++编译器有时会产生这样的代码，我过去曾经错误地把所有的此类指令当作没用的代码而删掉，后来发现程序执行时间没有明显的变化。通过查阅文档才知道，这类指令实际上是“占位指令”，他们存在的意义在于占据那个地方，一来使其他语句能够正确地按 CPU 觉得舒服的方式对齐，二来它可以占据 CPU 的某些周期，使得后续的指令能够更好地并发执行，避免冲突。另一个比较常见的、实现类似功能的指令是 NOP。

占位指令的去留主要是靠计时执行来判断。由于目前流行的操作系统基本上都是多任务的，因此会对计时的精确性有一定影响。如果需要进行测试的话，需要保证以下几点：

计时测试需要注意的问题

- **测试必须在没有额外负荷的机器上完成。例如，专门用于编写和调试程序的计算机**
- **尽量终止计算机上运行的所有服务，特别是杀毒程序**
- **切断计算机的网络，这样网络的影响会消失**
- **将进程优先级调高。对于 Windows 系统来说，把进程(线程)设置为 Time-Critical；对于 *nix 系统来说，把进程设置为实时进程**
- **将测试函数运行尽可能多次运行，如 10000000 次，这样能够减少由于进城切换而造成的偶然误差**
- **最后，如果可能的话，把函数放到单进程的系统(例如 FreeDOS)中运行。**

对于绝大多数程序来说，计时测试是一个非常重要的东西。我个人倾向于在进行优化后进行计时测试并比较结果。目前，我基于经验进行的优化基本上都能够提高程序的执行性能，但我还是不敢过于自信。优化确实会提高性能，但人做的和编译器做的思路不同，有时，我们的确会做一些费力不讨好的事情。