x86-64 Assembly Language Programming with Ubuntu

Ed Jorgensen, Ph.D.

Version 1.1.58

September 2024

Cover image:

Top view of an Intel central processing unit Core i7 Skylake type core, model 6700K, released in June 2015.
Source: Eric Gaba, https://commons.wikimedia.org/wiki/File:Intel_CPU_Core_i7_6700K_Skylake_top.jpg

C背景：由Benjamint444（本人作品）提供 来源：http://commons.wikimedia.org/wiki/File%3ASwirly_belt444.jpg

---

目录

Table of Contents

Table of Contents

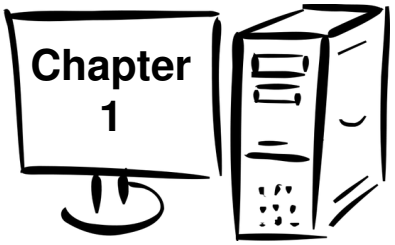
Table of Contents

Table of Contents

插图索引

Table of Contents

If you give someone a program, you will frustrate them for a day; if you teach them to program, you will frustrate them for a lifetime.



1.0 简介

本文本的目的是为大学汇编语言和系统编程课程提供参考。具体来说，本文针对使用Ubuntu 64位操作系统（OS）的流行x86-64[1]指令集进行阐述。虽然提供的代码和各个示例应在任何基于Linux的64位操作系统下工作，但它们仅在Ubuntu 22.04 LTS（64位）下进行了测试。

x86-64是一种复杂指令集计算（CISC[2]）CPU设计。这指的是内部处理器设计理念。CISC处理器通常包括广泛的指令（有时重叠），指令大小各异，以及广泛的寻址模式。该术语是事后相对于精简指令集计算机（RISC[3]）提出的。

1.1 前提条件

必须指出，本文档并非旨在学习如何编程。假设读者已经熟练掌握一门高级编程语言。具体来说，文本主要针对编译型、基于C的高级语言，如C、C++或Java。许多解释和示例都假设读者已经熟悉编程概念，例如声明、算术运算、控制结构、迭代、函数调用、函数、间接引用（即指针）和变量作用域问题。

此外，读者应熟悉使用基于Linux的操作系统，包括使用命令行。如果读者是Linux新手，附加参考文献部分提供了一些有用文档的链接。

1 For more information, refer to: http://en.wikipedia.org/wiki/X86-64
2 For more information, refer to: http://en.wikipedia.org/wiki/Complex_instruction_set_computing
3 For more information, refer to: http://en.wikipedia.org/wiki/Reduced_instruction_set_computing

1.2 汇编语言是什么

学生通常问的问题是“为什么学习汇编？”在回答这个问题之前，让我们先明确汇编语言究竟是什么。

汇编语言是针对特定机器的。例如，为x86-64处理器编写的代码将无法在不同的处理器上运行，例如RISC处理器（在平板电脑和智能手机中很受欢迎）。

汇编语言是一种“低级”语言，并为计算机处理器提供基本的指令接口。汇编语言是程序员能够达到处理器最近的语言。用高级语言编写的程序被翻译成汇编语言，以便处理器执行程序。高级语言是语言和实际处理器指令之间的抽象。因此，“汇编已死”的想法是荒谬的。

汇编语言让您直接控制系统的资源。这包括设置处理器寄存器、访问内存位置以及与其他硬件元素接口。这需要您对处理器和内存的工作原理有更深入的理解。

1.3 为何学习汇编语言

该文本的目的是提供汇编语言编程的全面介绍。学习汇编语言的原因更多是关于理解计算机的工作原理，而不是开发大型程序。由于汇编语言是针对特定机器的，因此其可移植性不足对编程项目非常有限。

实际上学习汇编语言的过程涉及编写非平凡程序以执行特定的低级操作，包括算术运算、函数调用、使用栈动态局部变量以及与操作系统交互进行输入/输出等活动。仅仅查看小的汇编语言程序是不够的。

从长远来看，学习底层原理，包括汇编语言，是区分一个无法应对语言变化的编码技术人员和一个能够适应不断变化技术的计算机科学家之间的关键。

以下部分提供了一些关于学习汇编语言的各种更具体原因的详细信息。

1.3.1 更好地理解架构问题

学习和花一些时间在汇编语言级别工作，可以更深入地理解底层计算机架构。这包括基本指令集、处理器寄存器、内存寻址、硬件接口以及输入/输出。由于最终所有程序都是在这一级别执行，了解汇编语言的能力提供了对可能实现什么、什么容易实现以及什么可能更困难或更慢的有用见解。

1.3.2 理解工具链

工具链是指将人类编写的代码转换为计算机可以直接执行的过程的名称。这包括编译器或汇编器（在我们的情况下），链接器、加载器和调试器。在编译方面，初学者被告知“只需这样做”，而对过程中涉及到的复杂性的解释很少。在底层工作可以帮助提供理解和欣赏工具链细节的基础。

1.3.3 提高算法开发技能

与汇编语言一起工作并编写底层程序有助于程序员通过使用需要更多思考和更多注意细节的语言来练习，从而提高算法开发技能。在极不可能的情况下，如果程序第一次运行不成功，调试汇编语言也提供了练习调试的机会，并且需要更细腻的方法，因为仅仅添加大量输出语句在汇编语言级别上更困难。这通常涉及更全面地使用调试器，这对于任何程序员来说都是一项有用的技能。

1.3.4 提高对函数/过程的了解/{v*}　　dūrès

与汇编语言一起工作可以极大地提高对函数/过程调用工作原理的理解。这包括函数调用帧的内容和结构，也称为活动记录。根据具体实例，活动记录可能包括基于堆栈的参数、保留寄存器和/或堆栈动态局部变量。关于堆栈动态局部变量的一些重要实现和安全影响最好在低级别工作时理解。由于安全影响，提醒读者始终为善是合适的。此外，堆栈及其相关调用帧是递归的基础，也是理解递归函数相对简单实现的基础。

1.3.5 理解I/O缓冲区

在高级语言中，输入/输出指令及其相关的缓冲操作可能看起来很神奇。在汇编语言级别工作并执行一些低级输入/输出操作可以更详细地了解输入/输出和缓冲实际上是如何工作的。这包括交互式输入/输出、文件输入/输出以及相关的操作系统服务之间的差异。

1.3.6 理解编译器作用域

编程使用汇编语言，在已经学习了一种高级语言之后，有助于确保程序员了解编译器的范围和能力。具体来说，这意味着学习编译器在计算机架构方面的作用和作用范围。

1.3.7 多进程概念介绍

此文本还将简要介绍多进程概念。分布式和多核编程的一般概念被提出，重点放在共享内存和线程处理上。作者认为，真正理解与线程相关的微妙问题，如共享内存和竞态条件，在低级别上最容易理解。

1.3.8 介绍中断处理概念

现代多用户计算机工作的基本机制基于中断。在低级别工作是介绍与中断处理、中断服务处理程序和向量中断相关的基本概念的最佳位置。

1.4 附加参考文献

一些关键参考文献以获取更多信息已在以下各节中注明。
这些参考资料提供了更广泛和详细的信息。

如果这些位置中的任何一个发生变化，网络搜索将能够找到新的位置。

1.4.1 Ubuntu 参考

Ubuntu操作系统有大量的文档可用。主要用户指南如下:

- Ubuntu 社区维基
- 开始使用Ubuntu 1 6 .04

此外，还有许多其他网站致力于提供使用Ubuntu（或其他基于Linux的操作系统）的帮助。

1.4.2 BASH 命令行参考

BASH是Ubuntu的默认shell。读者应熟悉基本的命令行操作。以下是一些额外的参考资料:

- Linux 命令行 （在线教程和文本）◦ Linux 命令壳入门指南（pdf）

此外，还有许多其他网站致力于提供有关 BASH 命令壳的信息。

1.4.3 架构参考

一些英特尔发布的重点参考提供了对支持 IA-32 和英特尔 64 架构的英特尔处理器架构和编程环境的详细技术描述。

- Intel® 64 和 IA-32 架构软件开发者手册：基本架构。◦ Intel 64 和 IA-32 架构软件开发者手册：指令集参考。◦ Intel 64 和 IA-32 架构软件开发者手册：系统编程指南。

如果嵌入的链接无法工作，网络搜索可以帮助找到新位置。

1.4.4 工具链引用

工具链包括汇编器、链接器、加载器和调试器。第5章“工具链”提供了本文本中使用的工具链概述。以下参考资料提供了更详细的信息和文档。

1.4.4.1 YASM 参考

YASM汇编器是一个在基于Linux的系统上普遍可用的开源汇编器。YASM引用如下:

- Yasm 网站信息
- Yasm 文档

有关YASM的附加信息可能可在多个汇编语言网站上找到，并通过网络搜索获得。

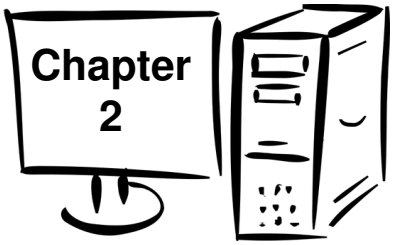
1.4.4.2 DDD调试器参考

DDD 调试器是一个能够支持汇编语言的开源调试器。

- DDD 网站页面
- DDD 文档

有关DDD的更多信息可能存在于多个汇编语言网站上，并且可以通过网络搜索找到。

Warning, keyboard not found. Press enter to continue.



2.0 架构概述

本章介绍了x86-64架构的基本、一般概述。如需更详细的解释，请参阅第1章“引言”中注明的附加参考文献。

2.1 架构概述

计算机的基本组件包括中央处理单元（CPU）、主存储器或随机存取存储器（RAM）、辅助存储器、输入/输出设备（例如，屏幕、键盘、鼠标），以及称为总线的互连。

以下是一个非常基本的计算机架构图：

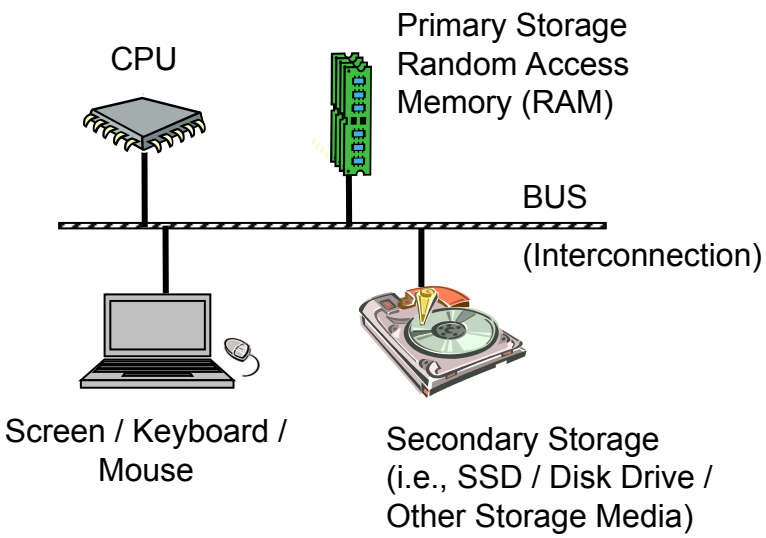


Illustration 1: Computer Architecture

The architecture is typically referred to as the Von Neumann Architecture[4], or the Princeton architecture, and was described in 1945 by the mathematician and physicist John von Neumann.

架构通常被称为冯·诺伊曼架构[4]，或普林斯顿架构，并于1945年由数学家和物理学家约翰·冯·诺伊曼描述。

程序和数据通常存储在二级存储器（例如磁盘驱动器或固态驱动器）上。当程序执行时，它必须从二级存储器复制到主存储器或主存（RAM）。CPU从主存储器或RAM执行程序。

主存储器或主内存也被称为易失性内存，因为当电源移除时，信息不会保留并因此丢失。辅助存储器被称为非易失性内存，因为断电时信息会保留。

例如，考虑将一篇论文存储在二级存储器（即磁盘）中。当用户开始编写或编辑论文时，它将从二级存储介质复制到主存储器（即RAM或内存）。完成后，更新的版本通常会被存储回二级存储器（即磁盘）。如果您在编辑文档时曾经失去过电源（假设没有电池或不间断电源），丢失未保存的工作将肯定能阐明易失性存储器和非易失性存储器之间的区别。

2.2 数据存储大小

x86-64架构支持一组特定的数据存储大小元素，所有这些元素都基于2的幂。支持的存储大小如下：

Storage	Size (bits)	Size (bytes)
Byte	8-bits	1 byte
Word	16-bits	2 bytes
Double-word	32-bits	4 bytes
Quadword	64-bits	8 bytes
Double quadword	128-bits	16 bytes

列表或数组（内存集）可以在这任何一种类型中预留。

这些存储大小与高级语言（例如C、C++、Java等）中的变量声明有直接关联。

4 For more information, refer to: http://en.wikipedia.org/wiki/Von_Neumann_architecture

例如，C/C++声明映射如下：

C/C++ Declaration	Storage	Size (bits)	Size (bytes)
char	Byte	8-bits	1 byte
short	Word	16-bits	2 bytes
int	Double-word	32-bits	4 bytes
unsigned int	Double-word	32-bits	4 bytes
long[5]	Quadword	64-bits	8 bytes
long long	Quadword	64-bits	8 bytes
char *	Quadword	64-bits	8 bytes
int *	Quadword	64-bits	8 bytes
float	Double-word	32-bits	4 bytes
double	Quadword	64-bits	8 bytes

星号表示地址变量。例如，int * 表示整数的地址。其他高级语言通常有类似的映射。

2.3 中央处理器

中央处理单元[6]（CPU）通常被称为计算机的“大脑”，因为实际的计算是在这里进行的。CPU封装在一个单芯片中，有时被称为处理器、芯片或晶圆[7]。封面图像显示了一个这样的CPU。

CPU芯片包括多个功能单元，包括算术逻辑单元[8]（ALU），它是芯片中实际执行算术和逻辑计算的部分。为了支持ALU，处理器寄存器[9]和缓存内存[10]也被包含在“芯片内部”（指芯片内部）。“CPU寄存器和缓存内存将在后续章节中描述。”

5 *Note*, the 'long' type declaration is compiler dependent. Type shown is for **gcc** and **g++** compilers.
6 For more information, refer to: http://en.wikipedia.org/wiki/Central_processing_unit
7 For more information, refer to: http://en.wikipedia.org/wiki/Die_(integrated_circuit)
8 For more information, refer to: http://en.wikipedia.org/wiki/Arithmetic_logic_unit
9 For more information, refer to: http://en.wikipedia.org/wiki/Processor_register
10 For more information, refer to: http://en.wikipedia.org/wiki/Cache_(computing)

现代处理器的内部设计非常复杂。本节提供了一个非常简化的、高级的CPU内部一些关键功能单元的概述。有关更多信息，请参阅脚注或附加参考文献。

2.3.1 CPU寄存器

CPU寄存器，或简称寄存器，是集成在CPU本身（与内存分离）中的临时存储或工作位置。计算通常由CPU使用寄存器执行。

2.3.1.1 通用寄存器（GPRs）

有十六个64位通用寄存器（GPR）。GPR在以下表中描述。GPR寄存器可以通过所有64位或部分或子集进行访问。

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

此外，一些GPR寄存器用于专用目的，具体内容在后续章节中描述。

当使用小于64位的数据元素大小（即32位、16位或8位）时，可以通过使用不同的寄存器名称来访问寄存器的低部分，如表中所示。

例如，当访问64位rax寄存器的低部分时，布局如下：

如图所示，前四个寄存器 rax、rbx、rcx 和 rdx 也允许使用 ah、bh、ch 和 dh 寄存器名访问位 8-15。除了 ah 之外，这些是为了向后兼容而提供的，本文本中不会使用。

访问寄存器部分的能力意味着，如果quadword rax寄存器设置为$50,000,000,000_{10}$（五十亿），则rax寄存器将包含以下十六进制值。

rax = 0000 000B A43B 7400

如果后续操作将 ax 寄存器设置为 $50,000_{10}$（五万，即 $C350_{16}$），rax 寄存器将包含以下十六进制值。

rax = 0000 000B A43B C350

在这种情况下，当64位rax寄存器的低16位ax部分被设置时，高48位不受影响。注意ax的变化（从7400_{16}到$C350_{16}$）。

如果后续操作将字节大小的al寄存器设置为50_{10}（即50，也就是32_{16}），rax寄存器将包含以下十六进制值。

rax = 0000 000B A43B C332

当64位rax寄存器的低8位al部分被设置时，高56位不受影响。注意al的变化（从50_{16}变为32_{16}）。

对于32位寄存器操作，高32位被清除（设置为0）。通常，这不会成问题，因为32位寄存器操作不使用寄存器的高32位。对于无符号值，这可以用于将32位转换为64位。

然而，这不会适用于从32位到64位的有符号转换。具体来说，它可能会对负值提供不正确的结果。有关有符号值的表示的更多信息，请参阅第3章，数据表示。

2.3.1.2 栈指针寄存器（RSP）

CPU寄存器之一，rsp，用于指向栈的当前顶部。rsp寄存器不应用于数据或其他用途。有关栈和栈操作的其他信息请参阅第9章，进程栈。

2.3.1.3 基准指针寄存器（RBP）

CPU寄存器之一，rbp，在函数调用期间用作基指针。rbp寄存器不应用于数据或其他用途。有关函数和函数调用的更多信息，请参阅第12章，函数。

2.3.1.4 指令指针寄存器（RIP）

除了GPRs之外，还有一个特殊的寄存器，rip，它被CPU用来指向***next instruction to be executed***。具体来说，由于rip指向下一条指令，这意味着rip所指向的指令，以及在调试器中显示的指令，尚未执行。这是一个重要的区别，在调试器中查看代码时可能会造成混淆。

2.3.1.5 标志寄存器（rFlags）

标志寄存器 rFlags 用于状态和 CPU 控制信息。rFlag 寄存器在每次指令执行后由 CPU 更新，并且程序无法直接访问。此寄存器存储有关刚刚执行的指令的状态信息。在 rFlag 寄存器的 64 位中，许多位被保留供将来使用。

The follo翼表显示了标志r中的一些状态位　　注册。

Name	Symbol	Bit	Use
Carry	CF	0	Used to indicate if the previous operation resulted in a carry.
Parity	PF	2	Used to indicate if the last byte has an even number of 1's (i.e., even parity).
Adjust	AF	4	Used to support Binary Coded Decimal operations.

Zero	ZF	6	Used to indicate if the previous operation resulted in a zero result.
Sign	SF	7	Used to indicate if the result of the previous operation resulted in a 1 in the most significant bit (indicating negative in the context of signed data).
Direction	DF	10	Used to specify the direction (increment or decrement) for some string operations.
Overflow	OF	11	Used to indicate if the previous operation resulted in an overflow.

存在一些在此文本中未指定的额外位。更多信息可以从第1章引言中注明的附加参考文献中获得。

2.3.1.6 XMM 寄存器

存在一组专用寄存器，用于支持64位和32位浮点运算以及单指令多数据（SIMD）指令。SIMD指令允许单个指令同时应用于多个数据项。如果有效使用，这可以显著提高性能。典型应用包括一些图形处理和数字信号处理。

XMM 寄存器如下所示：

128-bit Registers
xmm0
xmm1
xmm2
xmm3
xmm4
xmm5
xmm6
xmm7
xmm8
xmm9

xmm10
xmm11
xmm12
xmm13
xmm14
xmm15

注意，一些较新的X86-64处理器支持256位XMM寄存器。这在此文本的程序中不会成为问题。

此外，XMM寄存器用于支持Streaming SIMD Extensions（SSE）。SSE指令不在此文本的范围内。更多信息可以从英特尔参考资料中获得（如第1章引言中所述）。

2.3.2 缓存内存

缓存内存是位于CPU芯片中的主存储器或RAM的一个小子集。如果一个内存位置被访问，其值的副本将被放置在缓存中。随后对该内存位置的快速连续访问将从缓存位置（位于CPU芯片内部）检索。内存读取涉及通过总线发送地址到内存控制器，该控制器将获取请求的内存位置上的值，并通过总线将其发送回来。相比之下，如果值在缓存中，访问该值将快得多。

缓存命中发生在请求的数据可以在缓存中找到时，而缓存未命中则发生在无法找到时。缓存命中通过从缓存中读取数据来提供服务，这比从主内存中读取更快。可以从缓存中服务的请求数量越多，系统通常运行得越快。CPU芯片的连续几代都增加了缓存内存并改进了缓存映射策略，以提高整体性能。

典型CPU芯片配置的框图如下:

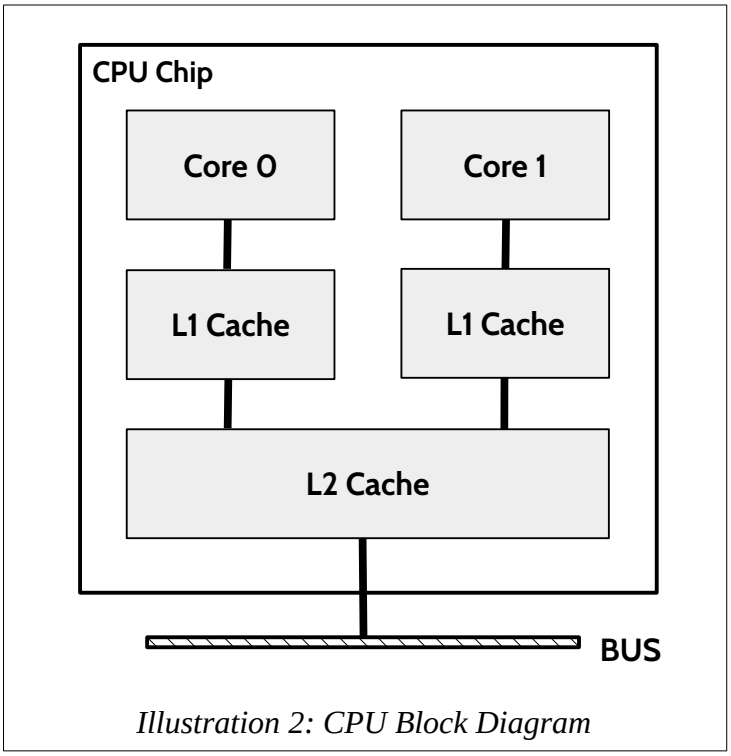


Illustration 2: CPU Block Diagram

当前芯片设计通常每个核心包含一个L1缓存和一个共享的L2缓存。许多较新的CPU芯片将包含一个额外的L3缓存。

从图中可以看出，所有内存访问都通过每个缓存级别。因此，存在多个重复的值副本（CPU寄存器、L1缓存、L2缓存和主内存）。这种复杂性由CPU管理，程序员无法改变。了解缓存及其相关的性能提升有助于理解计算机的工作原理。

2.4 主存储器

内存可以看作是一系列的字节，一个接一个。也就是说，内存是 *byte addressable* 。这意味着每个内存地址都存储一个字节的信息。要存储一个双字，需要四个字节，这需要使用四个内存地址。

此外，架构是 ***little-endian***。这意味着最低内存地址存储的是最低有效字节（LSB），最高内存位置存储的是最高有效字节（MSB）。

对于一个双字（32位），最高有效位（MSB）和最低有效位（LSB）的分配如下所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSB																								**LSB**							

例如，假设值 $5{,}000{,}000_{10}$（$004C4B40_{16}$）要放置在名为 var1 的双字变量中。

对于小端架构，内存图将如下所示：

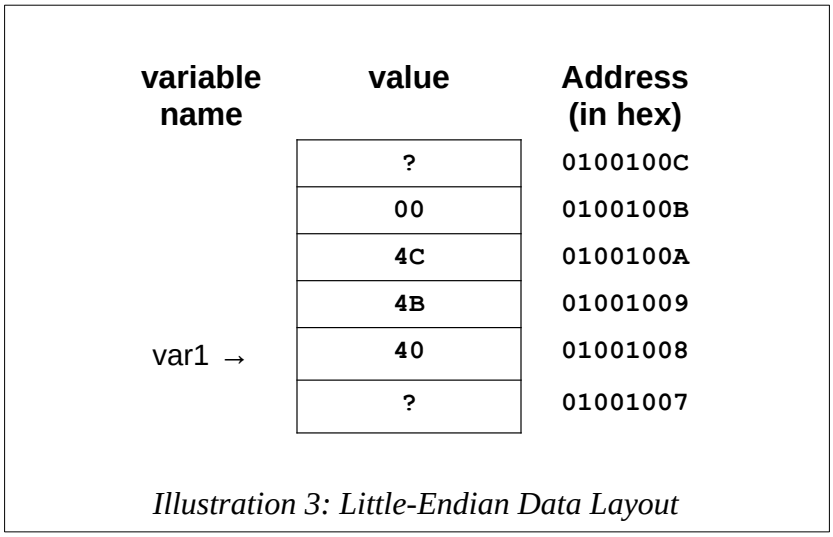


Illustration 3: Little-Endian Data Layout

基于小端架构，最低内存地址存储LSB，最高内存位置存储MSB。

2.5 内存布局

程序的一般内存布局如下所示:

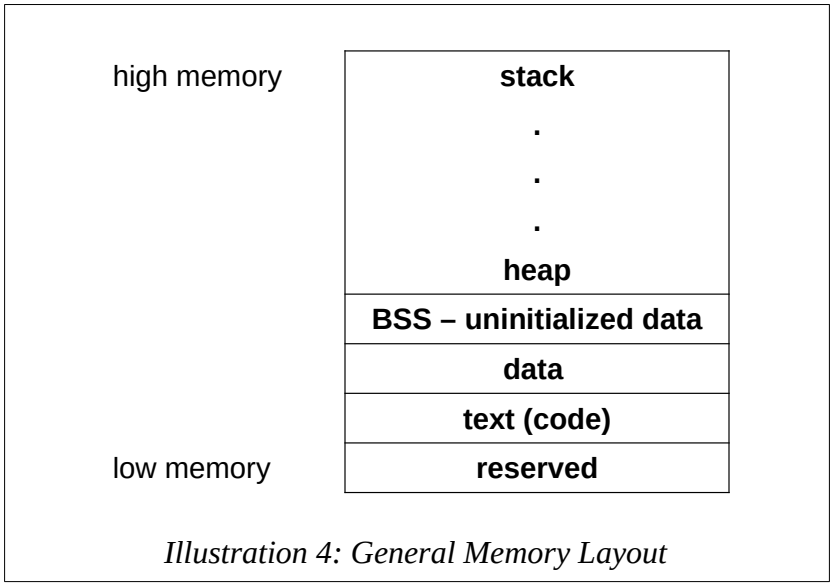


Illustration 4: General Memory Layout

保留部分对用户程序不可用。文本（或代码）部分是存储机器语言[11]（即表示代码的1和0）的地方。数据部分是存储初始化数据的地方。这包括在汇编时已提供初始值的声明变量。未初始化的数据部分，通常称为BSS部分，是存储未提供初始值的声明变量的地方。如果在使用前未设置，则值将没有意义。堆是存储动态分配数据的地方（如果请求）。栈从高内存开始，向下增长。

后续部分将为文本和数据部分提供更多详细信息。

2.6 存储层次结构

为了全面理解各种不同的内存级别及其相关使用，回顾内存层次结构[12]很有用。一般来说，速度更快的内存更昂贵，而速度较慢的内存块则更便宜。CPU寄存器很小，

11 For more information, refer to: http://en.wikipedia.org/wiki/Machine_code
12 For more information, refer to: http://en.wikipedia.org/wiki/Memory_hierarchy

快速、昂贵。二级存储设备，如磁盘驱动器和固态硬盘（SSD），体积更大、速度更慢、价格更低。总体目标是平衡性能与成本。

内存层次结构概述如下:

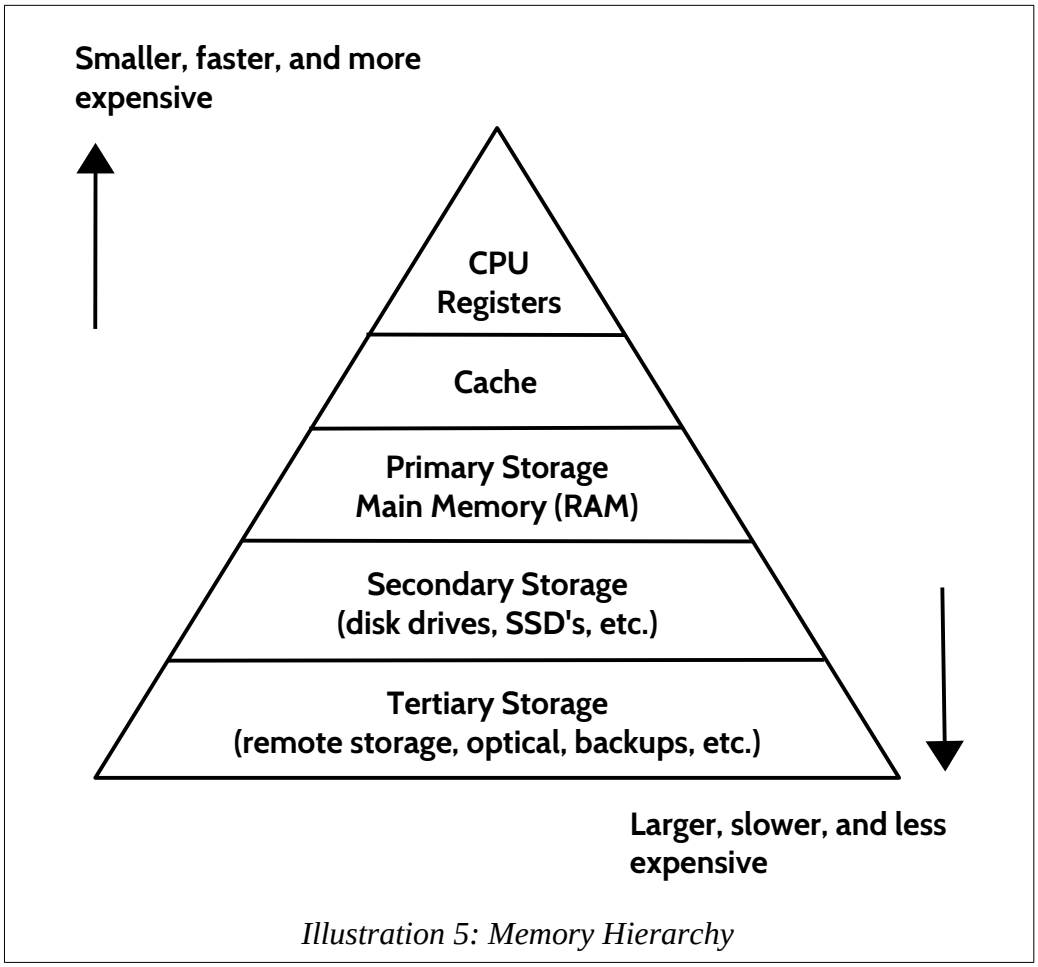


Illustration 5: Memory Hierarchy

顶部的三角形代表最快的、最小的和最昂贵的内存。随着我们向下移动层级，内存变得较慢、较大且价格较低。目标是使用小、快、昂贵的内存和大、慢、便宜的内存之间的有效平衡。

一些典型的性能和尺寸特性如下:

Memory Unit	Example Size	Typical Speed
Registers	16, 64-bit registers	~1 nanoseconds[13]
Cache Memory	4 - 8+ Megabytes[14] (L1 and L2)	~5-60 nanoseconds
Primary Storage (i.e., main memory)	2 – 32+ Gigabytes[15]	~100-150 nanoseconds
Secondary Storage (i.e., disk, SSD's, etc.)	500 Gigabytes – 4+ Terabytes[16]	~3-15 milliseconds[17]

基于此表，主存储器访问时间为100纳秒（100 {v*} 10^{v*}），比二级存储器访问时间快30,000倍，后者为3毫秒（3 {v*} 10^{v*}）。

典型的速度会随着时间的推移而提高（而且这些已经过时了）。关键点是每个存储单元之间的相对差异是显著的。即使使用更新、更快的SSD，存储单元之间的这种差异仍然适用。

2.7 练习

以下是本章的一些问题。

2.7.1 测验问题

以下是几个测验问题。

1) 绘制冯·诺依曼架构的图片。2) 哪个架构组件将内存连接到CPU？3）当计算机关闭时，程序存储在哪里？4）程序执行时必须位于何处？5）缓存内存如何帮助整体性能？6）使用声明int的C++整数占用多少字节?

13 For more information, refer to: http://en.wikipedia.org/wiki/Nanosecond
14 For more information, refer to: http://en.wikipedia.org/wiki/Megabyte
15 For more information, refer to: http://en.wikipedia.org/wiki/Gigabyte
16 For more information, refer to: http://en.wikipedia.org/wiki/Terabyte
17 For more information, refer to: http://en.wikipedia.org/wiki/Millisecond

7) 在Intel X86-64架构上，每个地址可以存储多少个***bytes***?

8) 给定32位十六进制数$004C4B40_{16}$，以下是什么：

1. 最不重要字节 (LSB) 2. 最重要字节 (MSB)

9) 给定32位十六进制数$004C4B40_{16}$，展示小端内存布局，显示内存中的每个字节。

10) 绘制rax寄存器布局图。

11) 以下每个表示多少位：

1. al2. rcx
3. bx4. ed
x5. r116.
r8b7. sil8
. r14w

12) 哪个寄存器指向下一个要执行的指令?

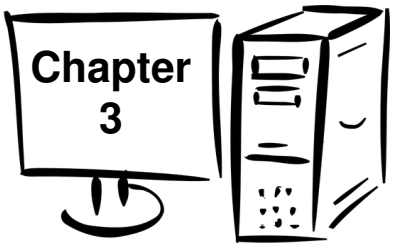
13) 哪个寄存器指向栈顶当前值?

14) 如果al设置为05_{16}，ax设置为0007_{16}，eax设置为00000020_{16}，rax设置为0000000000000000_{16}，则显示完整的rax寄存器内容。

15) 如果rax寄存器设置为$81,985,529,216,486,895_{10}$（$123456789ABCDEF_{16}$），那么以下寄存器在***hex***中的内容是什么?

1. al2. a
x3. eax4
. rax

There are 10 types of people in the world; those that understand binary and those that don't.



3.0 数据表示

数据表示指的是信息在计算机中的存储方式。存储整数有特定方法，与存储浮点数不同，与存储字符也不同。本章简要介绍了整数、浮点数和ASCII表示方案。

它假定读者已经普遍熟悉二进制、十进制和十六进制数制。

应注意的是，如果未指定，则数字为十进制。此外，以0x开头的前面的数字是十六进制值。例如，$19 = 19_{10} = 13_{16} = 0x13$。

3.1 整数表示

表示整数数字指的是计算机在内存中存储或表示数字的方式。计算机使用二进制（1和0）来表示数字。然而，计算机为每个数字或变量可用的空间是有限的。这直接影响了可以表示的数字的大小或范围。例如，一个字节（8位）可以用来表示2^^{8}或256个不同的数字。这256个不同的数字可以是（所有都是正数），在这种情况下，我们可以表示从0到255（包括0和255）之间的任何数字。如果我们选择（正数和负数），那么我们可以表示从-128到+127（包括-128和127）之间的任何数字。

如果该范围不足以处理预期值，则必须使用更大的大小。例如，一个字（16位）可以用来表示2^{16}或65,536个不同的值，而一个双字（32位）可以用来表示2^{32}或4,294,967,296个不同的数字。因此，如果您想存储100,000的值，则需要使用双字。

如您从C、C++或Java中回忆起来，一个整型声明（例如，int <variable>）是一个单双字，可以用来表示介于-2^{31} (−2,147,483,648)和+2^{31} - 1 (+2,147,483,647)之间的值。

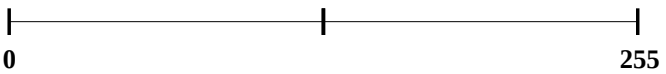
以下表格显示了与典型尺寸相关的范围:

Size	Size	Unsigned Range	Signed Range
Bytes (8-bits)	2^8	0 to 255	-128 to +127
Words (16-bits)	2^{16}	0 to 65,535	−32,768 to +32,767
Double-words (32-bits)	2^{32}	0 to 4,294,967,295	−2,147,483,648 to +2,147,483,647
Quadword	2^{64}	0 to 2^{64} - 1	-(2^{63}) to 2^{63} - 1
Double quadword	2^{128}	0 to 2^{128} - 1	-(2^{127}) to 2^{127} - 1

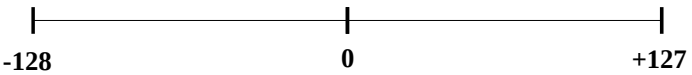
为了确定一个值是否可以表示，您需要知道正在使用的存储元素的大小（字节、字、双字、四字等）以及值是有符号还是无符号。

- 为了表示给定存储大小范围内的 *unsigned* 值，使用标准二进制。
- 为了表示范围内的 *signed* 值，使用二进制补码。具体来说，二进制补码编码过程适用于负数范围内的值。对于正数范围内的值，使用标准二进制。

例如，无符号字节的范围可以用以下数轴表示:



例如，有符号字节范围也可以使用以下数轴表示:



相同的 con　cept适用于半字和具有大的字　　er 范围。

由于无符号值与有符号值的范围不同，仅正数，因此值之间存在重叠。这在检查内存中的变量（使用调试器）时可能会非常令人困惑。

例如，当无符号和有符号值在重叠的正数范围内（0到+127）时：

- 12_{10}的已签名字节表示为$0x0C_{16}$
- 未签名的字节表示 -12_{10} 也是 $0x0C_{16}$

当无符号和有符号值超出重叠范围时：

- -15_{10}的已签名字节表示为$0xF1_{16}$
- 未签名的字节表示241_{10}也是$0xF1_{16}$

这种重叠可能会导致混淆，除非数据类型被明确且正确地定义。

3.1.1 二进制补码

以下描述了如何找到负值（非正值）的二进制补码表示。

要取一个数的二进制补码：

1. 取反码（取负）2. 加1（二进制）

相同的流程用于将十进制值编码为二进制补码，以及从二进制补码转换回十进制。以下章节提供了一些示例。

3.1.2 字节示例

例如，要找到-9和-12的8位字节大小（8位）的二进制补码表示。

9 (8+1) =	**00001001**
Step 1	**11110110**
Step 2	**11110111**
-9 (in hex) =	**F7**

12 (8+4) =	**00001100**
Step 1:	**11110011**
	11110100
-12 (in hex) =	**F4**

*Note*所有给定大小（本例中为字节）的所有位都必须指定。

3.1.3 单词示例

要找到字长（16位），-18和-40的补码表示。

18 (16+2) =	**0000000000010010**
Step 1	**1111111111101101**
Step 2	**1111111111101110**
-18 (hex) =	**0xFFEE**

40 (32+8) =	**0000000000101000**
Step 1	**1111111111010111**
Step 2	**1111111111011000**
-40 (hex) =	**0xFFD8**

*Note*所有给定大小的比特，这些示例中的单词，都必须指定。

3.2 无符号和有符号加法

如前所述，无符号和有符号表示可能对所表示的最终值提供不同的解释。然而，加法和减法操作是相同的。例如：

241	**11110001**
+ 7	**00000111**
248	**11111000**
248 =	**F8**

-15	**11110001**
+ 7	**00000111**
-8	**11111000**
-8 =	**F8**

最终结果0xF8可能解释为无符号表示的248和有符号表示的-8。此外，$0xF8_{16}$是ASCII表中的°（度符号）。

因此，对于正在执行的操作，明确数据的大小（字节、半字、字等）和类型（有符号、无符号）非常重要。

3.3 浮点表示

浮点数的表示问题更为复杂。存在一系列用于不同值范围的浮点数表示。为了简单起见，我们将主要关注IEEE 754 32位浮点标准。

3.3.1 IEEE 32位表示

IEEE 754 32位浮点标准如下定义:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	**biased exponent**								**fraction**																						

s 是符号（0 => 正和1 => 负）。更正式地，这可以写成;

$$\boldsymbol{N} = (-\mathbf{1})^{\boldsymbol{S}} \times \mathbf{1}.\boldsymbol{F} \times \mathbf{2}^{\boldsymbol{E}-\mathbf{127}}$$

当表示浮点值时，第一步是将浮点值转换为二进制。以下表格简要回顾了二进制如何处理小数部分:

	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	
...	8	4	2	1	.	1/2	1/4	1/8	...
	0	**0**	**0**	**0**	**.**	**0**	**0**	**0**	

例如，100.101_2 将是 4.625_{10}。对于循环小数，计算二进制值可能很耗时。然而，由于计算机的存储大小有限（本例中为32位），这有一个限制。

下一步是显示以二进制规范化科学记数法表示的值。这意味着数字应该有一个单独的非零首位数字位于小数点左侧。例如，8.125_{10}是1000.001_2（或$1000.001_2 \times 2^0$），在二进制规范化科学记数法中，这将写成1.000001×2^3（因为小数点向左移动了三位）。当然，如果数字是0.125_{10}，二进制将是0.001_2（或$0.001_2 \times 2^0$），规范化科学记数法将是1.0×2^{-3}（因为小数点向右移动了三位）。在首位1之后的数字，包括首位1，在双字分数部分中左对齐存储。

下一步是计算 *biased exponent*，它是规范化科学记数法中的指数加上偏置。IEEE 754 32位浮点标准的偏置是 127_{10}。结果应转换为字节（8位）并存储在词的偏置指数部分。

Note, 从IEEE 754 32位浮点表示转换为十进制值是反向进行的，但是必须重新添加前导1（因为它没有存储在单词中）。此外，还减去了偏置（而不是添加）。

3.3.1.1 IEEE 32位表示示例

本节提供了几个用于参考的浮点数表示编码和解码的示例。

3.3.1.1.1 示例 → -7.75_{10}

例如，要找到IEEE 754 32位浮点数表示-7.75_{10}：

示例 1：-7.75

Translated Text: E

- 确定符号 -7.75 => 1（因为负数）
- 转换为二进制 $-7.75 = -0111.11_2$
- 标准化科学记数法 = 1.1111×2^2
- 计算有偏指数 $2_{10} + 127_{10} = 129_{10}$ ◦ 并转换为二进制 = 10000001_2
- 二进制表示组件：符号 指数 尾数 1 10000001 11110000000000000000000
- 转换为十六进制（分为4位一组） 110000001111 10000000000000000000 1100 0000 1111 1000 000 0 0000 0000 0000 C 0 F 8 0 0 0 0
- 最终结果：$C0F8\ 0000_{16}$

3.3.1.1.2 示例 → -0.125_{10}

例如，要找到IEEE 754 32位浮点数表示 -0.125_{10}：

示例 2：-0.125

- 确定符号 -0.125 => 1（因为负数）
- 转换为二进制 $-0.125 = -0.001_2$
- 标准化科学记数法 = 1.0×2^{-3}
- 计算有偏指数 $-3_{10} + 127_{10} = 124_{10}$ ◦ 并转换为二进制 = $0111110 0_2$
- 二进制表示组件：符号 指数 尾数 1 01111100 00000000000000000000000

- 转换为十六进制（分为4位一组） 101111100000 00000000000000000000 1011 1110 0000 0000 0000 0000 0000 0000 B E 0 0 0 0 0 0
- 最终结果：BE00 0000_{16}

3.3.1.1.3 示例 → 41440000_{16}

例如，给定IEEE 754 32位浮点表示 41440000_{16}，找到十进制值：

示例 3：41440000_{16}

- convert to binary
  $0100\ 0001\ 0100\ 0100\ 0000\ 0000\ 0000\ 0000_2$
- split into components
  $0\ 10000010\ 10001000000000000000000_2$
- determine exponent $10000010_2 = 130_{10}$
  - and remove bias $130_{10} - 127_{10} = 3_{10}$
- determine sign 0 => positive
- write result $+1.10001 \times 2^3 = +1100.01 =$ **+12.25**

3.3.2 IEEE 64位表示

IEEE 754 64位浮点标准如下定义：

63	62		52	51		0
s	**biased exponent**			**fraction**		

表示过程相同，然而格式允许使用11位偏置指数（支持大值和小值）。11位偏置指数使用偏置±1023。

3.3.3 非数字 (NaN)

当值被解释为浮点值且不符合定义的标准（无论是32位还是64位）时，则不能用作浮点值。这可能会发生，如果整数表示法被当作浮点表示法处理，或者浮点算术运算（加、减、乘或除）的结果太大或太小而无法表示。不正确的格式或无法表示的数字被称为NaN，它是*not a number*的缩写。

3.4 字符和字符串

除了数值数据外，通常还需要符号数据。符号或非数值数据可能包括像“Hello World”[18]这样的重要信息，这是第一个程序的常见问候语。这些符号为英语语言使用者所熟知。计算机内存被设计用于存储和检索数字。因此，通过为每个符号或字符分配数值来表示这些符号。

3.4.1 字符表示

在计算机中，一个字符[19]是信息单位，对应于字母表中的符号，例如字母。字符的例子包括字母、数字、常见的标点符号（如“。”或“！”）和空白。一般概念还包括控制字符，这些控制字符不对应于特定语言中的符号，而是用于处理文本的其他信息。控制字符的例子包括回车符或制表符。

3.4.1.1 美国信息交换标准代码

字符使用美国信息交换标准代码（ASCII[20]）表示。根据ASCII表，每个字符和控制字符都被分配一个数值。使用ASCII时，显示的字符基于分配的数值。这仅在每个人都同意共同值的情况下才有效，这就是ASCII表的目的。例如，字母“A”被定义为65_{10}（0x41）。0x41存储在计算机内存中，当在控制台显示时，字母“A”就会显示出来。有关完整的ASCII表，请参阅附录A。

此外，数字符号也可以用ASCII表示。例如，“9”在计算机内存中表示为57_{10}（0x39）。数字“9”可以显示为控制台输出。如果发送到控制台，整数值9_{10}（0x09）将被解释为ASCII值，在这种情况下将是一个制表符。

非常重要理解字符（如“2”）和整数（如2_{10}）之间的区别。字符可以显示到控制台，但不能用于计算。整数可以用于计算，但不能显示到控制台（不改变表示形式）。

一个字符通常存储在1个字节（8位）的空间中。这很有效，因为内存是按字节寻址的。

18 For more information, refer to: http://en.wikipedia.org/wiki/”Hello,_World!”_program
19 For more information, refer to: http://en.wikipedia.org/wiki/Character_(computing)
20 For more information, refer to: http://en.wikipedia.org/wiki/ASCII

3.4.1.2 Unicode

应注意的是，Unicode[21]是一个包含对不同语言支持的当前标准。Unicode标准提供了一系列不同的编码方案（UTF-8、UTF-16、UTF-32等），以便为每个字符提供一个唯一的数字，无论是什么平台、设备、应用程序或语言。在最常见的编码方案UTF-8中，ASCII英语文本在UTF-8中看起来与在ASCII中完全相同。根据需要使用额外的字节来表示其他字符。有关Unicode表示的详细信息，本文未涉及。

3.4.2 字符串表示

一个字符串[22]是一系列ASCII字符，通常以NULL结尾。NULL是一个不可打印的ASCII控制字符。由于它不可打印，因此可以用来标记字符串的结尾。

例如，字符串“Hello”将如下表示:

Character	“H”	“e”	“l”	“l”	“o”	NULL
ASCII Value (decimal)	72	101	108	108	111	0
ASCII Value (hex)	0x48	0x65	0x6C	0x6C	0x6F	0x0

一个字符串可能部分或全部由数字符号组成。例如，字符串“19653”将如下表示:

Character	“1”	“9”	“6”	“5”	“3”	NULL
ASCII Value (decimal)	49	57	54	53	51	0
ASCII Value (hex)	0x31	0x39	0x36	0x35	0x33	0x0

再次强调，理解字符串“19653”（使用6个字节）和单个整数$19{,}653_{10}$（可以存储在一个2字节的单词中）之间的区别非常重要。

3.5 练习

以下是本章的一些问题。

21 For more information, refer to: http://en.wikipedia.org/wiki/Unicode
22 For more information, refer to: http://en.wikipedia.org/wiki/String_(computer_science)

3.5.1 测验问题

以下是几个测验问题。

1) 提供以下各项的范围：

1. 有符号字节 2. 无符号字节 3. 有符号字 4. 无符号字 5. 有符号双字 6. 无符号双字

2) 提供以下二进制数的十进制值：

1. 0000101_2
2. 0001001_2
3. 0001101_2
4. 0010101_2

3) 提供以下十进制值的十六进制、***byte*** 大小、二进制补码值。*Note*，期望两个十六进制数字。

1. -3_{10} 2. +11_{10} 3. -9_{10}
4. -21_{10}

4) 提供以下十进制值的十六进制、***word*** 大小、二进制补码值。*Note*，期望四个十六进制数字。

1. -17_{10} 2. +17_{10} 3. -1_{10} 4. -138

10

5) 提供以下十进制值的十六进制、***double-word*** 大小、二进制补码值。*Note*，期望八个十六进制数字。

1. -11_{10} 2. 27_{10} 3. $+7_{10}$
4. -261_{10}

6) 提供以下十六进制、双字大小的二进制补码值的十进制数值。

1. $FFFFFFFB_{16}$ 2. $FFFFFFEA_{16}$ 3. $FFFFFF3_{16}$ 4. $FFFF8_{16}$

7) 以下哪个十进制值在二进制中有 ***exact*** 表示?

1. 0.1 2. 0.2 3. 0.3 4. 0.4 5. 0.5

8) 提供以下 IEEE 32 位浮点数的十进制表示。

1. 0xC1440000 2. 0x41440000 3. 0xC0D00000 4. 0xC0F00000

9) 提供以下浮点数的十六进制和IEEE 32位浮点表示。

1. $+11.25_{10}$ 2. -17.125_{10} 3. $+21.875_{10}$ 4. -0.75_{10}

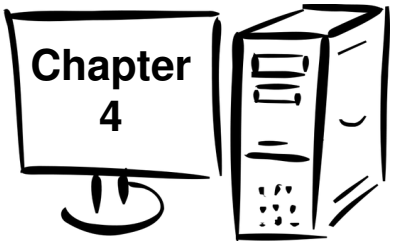
10) 每个以下字符的ASCII码（十六进制）是什么：

1. “A”
2. “a”
3. “0”
4. “8”
5. 制表符

11) 以下字符串的ASCII值（十六进制）分别是：

1. “世界”
2. “123”
3. “是！？”

I would love to change the world, but they won't give me the source code.



4.0 程序格式

本章总结了汇编语言程序的格式要求。格式要求针对yasm汇编器。其他汇编器可能略有不同。提供了一个完整的汇编语言程序示例，以展示适当的程序格式。

一个格式正确的汇编源文件由几个主要部分组成;

- 数据部分，其中声明和定义初始化数据。
- BSS部分，其中声明了未初始化的数据。
- 文本部分，其中放置代码的区域。

以下部分总结了基本的格式要求。 仅展示了基本的格式和汇编器语法。 有关更多信息，请参阅 yasm 参考手册（如第 1 章引言中所述）。

4.1 注释

分号（;）用于标记程序注释。注释（使用;）可以放置在任何地方，包括在指令之后。分号之后的任何字符都将被汇编器忽略。这可以用来解释代码中采取的步骤或注释掉代码的部分。

4.2 数字值

数值可以用十进制、十六进制或八进制指定。

当指定十六进制或基数为16的值时，它们前面有一个0x。例如，要指定127为十六进制，则为0x7f。

当指定八进制或基数为8的值时，它们后面跟着一个q。例如，

要指定511为八进制，它将是777q。

默认的基数（基数）是十进制，因此对于十进制（基数-10）数字不需要特殊符号。

4.3 定义常量

常量使用 equ 定义。一般格式为:

<name**>**等于**<**value**>**

The value 程序执行期间常量的 {v*} 不能更改 执行。

常量在汇编过程中被替换为其定义的值。因此，常量不会被分配内存位置。这使得常量更加灵活，因为它没有被分配特定的类型/大小（字节、字、双字等）。值受预期用途的范围限制。例如，以下常量，

SIZE equ 10000

可以作为单词或双词使用，但不能作为字节使用。

4.4 数据部分

初始化数据必须在".data"节中声明。 "section"一词后必须有一个空格。 所有初始化的变量和常量都放在这个节中。 变量名必须以字母开头，后跟字母、数字，包括一些特殊字符（如下划线，“_”）。 变量定义必须包括变量名、数据类型和变量的初始值。

通用格式为:

<variableName> <dataType> <initialValue>

参考以下部分，以了解使用各种数据类型的一系列示例。

支持的数据类型如下:

Declaration	
db	8-bit variable(s)

dw	16-bit variable(s)
dd	32-bit variable(s)
dq	64-bit variable(s)
ddq	128-bit variable(s) → integer
dt	128-bit variable(s) → float

这些是初始化数据声明的初级汇编指令。其他指令在不同的部分中引用。

初始化数组使用逗号分隔的值定义。

一些简单示例包括:

```
bVar        db      10                  ; byte variable
cVar        db      "H"                 ; single character
strng       db      "Hello World"       ; string
wVar        dw      5000                ; 16-bit variable
dVar        dd      50000               ; 32-bit variable
arr         dd      100, 200, 300       ; 3 element array
flt1        dd      3.14159             ; 32-bit float
qVar        dq      1000000000          ; 64-bit variable
```

指定的值必须能够适应指定的数据类型。例如，如果字节数据类型的变量值被定义为500，将生成汇编器错误。

4.5 BSS部分

未初始化的数据在".bss"节中声明。在单词"section"之后必须有一个空格。所有未初始化的变量都声明在这个节中。变量名以字母开头，后跟字母、数字以及一些特殊字符（如下划线"_"）。变量定义必须包括名称、数据类型和计数。

通用格式为:

```
<variableName>    <resType>     <count>
```

参考以下部分，以了解使用各种数据类型的一系列示例。

支持的数据类型如下:

Declaration	
resb	8-bit variable(s)
resw	16-bit variable(s)
resd	32-bit variable(s)
resq	64-bit variable(s)
resdq	128-bit variable(s)

这些是未初始化数据声明的首选汇编器指令。其他指令在不同的部分中引用。

一些简单示例包括:

```
bArr        resb        10          ; 10 element byte array
wArr        resw        50          ; 50 element word array
dArr        resd        100         ; 100 element double array
qArr        resq        200         ; 200 element quad array
```

分配的数组可能未初始化为任何特定值。

4.6 文本部分

代码放置在“section .text”部分。单词“section”之后必须有一个空格。指令每行指定一个，并且每个指令都必须是有效的指令，并具有适当的所需操作数。

文本部分将包括一些标题或标签，用于定义初始程序入口点。例如，假设使用标准系统链接器的基本程序，必须包含以下声明。

全局 _start _start:

无需特殊标签或指令来终止程序。然而，应使用系统服务来通知操作系统程序应终止，并回收和重新利用资源，如内存。有关示例程序，请参阅下一节。

4.7 示例程序

一个非常简单的汇编语言程序被展示出来，以演示适当的程序格式化。

```
; 简单示例，展示基本程序格式和布局。; 爱德华·约尔根森; 2014年7月18日; **********
**************************************************; 一些基本数据声明



节 .数据

; ----- ; 定义常量

EXIT_SUCCESS equ 0 ; 成功操作SYS_exit equ 60 ; 终止调用代码


; ----- ; 字节（8位）变量声明

bVar1 db 17 bVar2 db
9 bResult db 0

; ----- ; 16位单词变量声明

wVar1 dw 17000wVar2 dw
9000wResult dw 0

; ----- ; 双字（32位）变量声明

dVar1 dd 17000000 dVar2 dd
9000000 dResult dd 0
```

```
; ----- ; 四倍字（64位）变量声明

qVar1          dq      170000000qVar2          dq
90000000qResult        dq      0

; ********************************************************** ; 代码部分

章节     .text global _start
_start:

; 执行一系列非常基本的加法操作 ; 以展示基本程序格式。

; ---------- ; 字节示例 ;   bResult = bVar1 +
bVar2     mov     al, byte [bVar1]     add     al,
byte [bVar2]     mov     byte [bResult], al



; ---------- ; 单词示例 ;   wResult = wVar1 +
wVar2     mov     ax, word [wVar1]     add
ax, word [wVar2]     mov     word [wResult],
ax



; ---------- ; 双词示例 ;   dResult = dVar1 + dV
ar2     mov     eax, dword [dVar1]     add     eax,
dword [dVar2]     mov     dword [dResult], eax
```

```
; ---------- ; 四字词示例 ;  qResult = qVar1 +
qVar2   mov   rax, qword [qVar1]   add   ra
x, qword [qVar2]   mov   qword [qResult], ra
x
```

```
; ********************************************************** ; 完成，终止
程序。

last:   mov   rax, SYS_exit    ; 退出调用代码   mov   rdi, EXIT_SUCCESS  ; 成
功退出程序   syscall
```

此示例程序将在以下章节中引用并进一步解释。

4.8 练习

以下是本章的一些问题。

4.8.1 测验问题

以下是几个测验问题。

1) 本章中使用的是哪个汇编器的名称？2）汇编语言程序中的注释是如何标记的？3）初始化数据声明的部分名称是什么？4）未初始化数据声明的部分名称是什么？5）代码放置的部分名称是什么？6）以下变量的数据声明及其给定值：1. 字节大小的变量 ***bNum*** 设置为 10_{10} 2. 字词大小的变量 ***wNum*** 设置为 $10{,}291_{10}$ 3. 双字大小的变量 ***dwNum*** 设置为 $2{,}126{,}010_{10}$

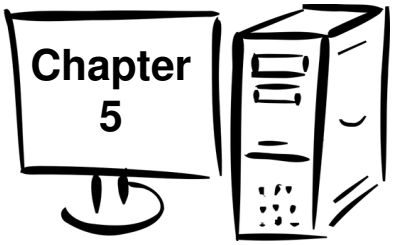
4. quadword 大小的变量 ***qwNum*** 设置为 $10{,}000{,}000{,}000_{10}$

7) 以下各项未初始化数据声明的形式是什么:

1. 命名为 ***bArr*** 的字节大小数组，包含100个元素2. 命名为 ***wArr*** 的字大小数组，包含3000个元素3. 命名为 ***dwArr*** 的双字大小数组，包含200个元素4. 命名为 ***qArr*** 的四字大小数组，包含5000个元素

8) 在文本部分，表示程序开始所需的声明有哪些?

There are two ways to write error-free programs; only the third works.



5.0 工具链

通常，用于创建程序的编程工具集合被称为 ***tool chain***[23]。在本文中，工具链包括以下内容；

- 汇编器
- 链接器
- 加载器
- 调试器

虽然工具有很多选择，但本文使用了一套相当标准的开源工具，这些工具可以很好地协同工作，并完全支持x86 64位环境。

每个的　　e 编程工具在以下部分中解释　　动作。

5.1 汇总/链接/加载概述

在广义上，组装、链接和加载过程是将程序员编写的源文件转换为可执行程序的方式。

人类可读的源文件由汇编器转换为对象文件。在最基本的形式中，对象文件由链接器转换为可执行文件。加载器将可执行文件加载到内存中。

23 For more information, refer to: http://en.wikipedia.org/wiki/Toolchain

流程概述如下图所示。

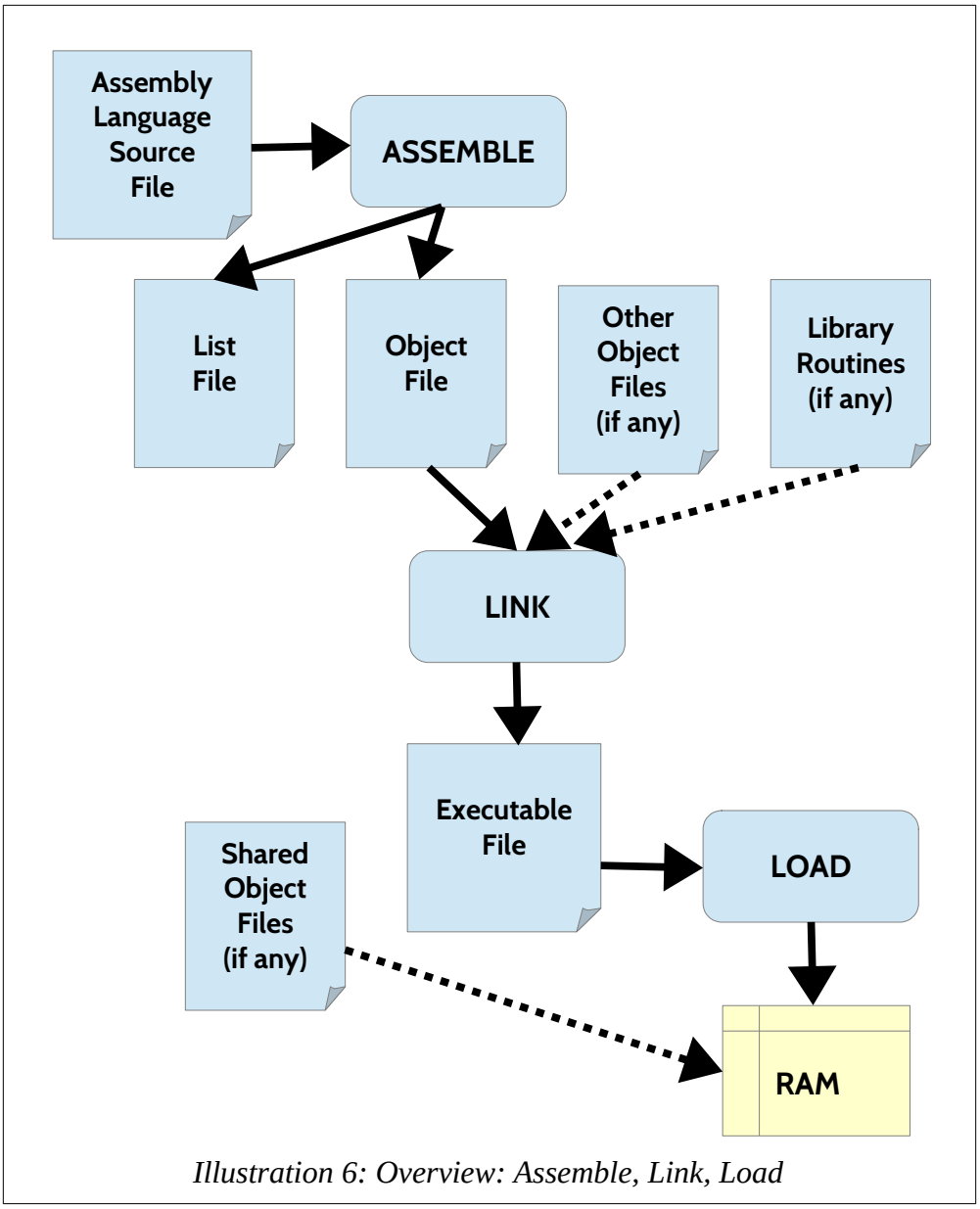


Illustration 6: Overview: Assemble, Link, Load

以下章节中更详细地描述了组装、链接和加载步骤。

5.2 汇编器

汇编器[24]是一个程序，它将读取汇编语言输入文件并将代码转换为机器语言二进制文件。输入文件是一个包含人类可读形式的汇编语言指令的汇编语言源文件。机器语言输出被称为目标文件。在这个过程中，注释被删除，变量名和标签被转换为适当的地址（CPU在执行时所需）。

该文本中使用的汇编器是yasm[25]汇编器。有关yasm网站和文档的链接可以在第1章“简介”中找到。

5.2.1 组装命令

适当的 yasm 汇编器命令用于读取汇编语言源文件，例如前一章中的示例，如下所示：

```
yasm -g dwarf2 -f elf64 example.asm -l example.lst
```

Note，-l 是一个短横线小写字母L（容易与数字1混淆）。

The -g dwarf2[26] 选项用于通知汇编器在最终对象文件中包含调试信息。这会增加对象文件的大小，但允许有效的调试。-f elf64 通知汇编器以 ELF64[27] 格式创建对象文件，这对于基于 Linux 的 64 位系统是合适的。example.asm 是输入的汇编语言源文件的名称。-l example.lst（短横线小写字母 L）通知汇编器创建一个名为 *example.lst* 的列表文件。

如果在组装过程中发生错误，必须在继续到链接步骤之前解决。

5.2.2 列表文件

此外，汇编器可选择创建列表文件。列表文件显示行号、相对地址、指令的机器语言版本（包括变量引用）以及原始源行。列表文件在调试时可能很有用。

---

24 For more information, refer to: http://en.wikipedia.org/wiki/Assembler_(computing)#Assembler
25 For more information, refer to: https://en.wikipedia.org/wiki/Yasm
26 For more information, refer to: https://en.wikipedia.org/wiki/DWARF
27 For more information, refer to: http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

例如，来自上一章示例程序列表文件数据部分的片段如下:

```
36 00000009 40660301          dVar1    dd  17000000
37 0000000D 40548900          dVar2    dd  9000000
38 00000011 00000000          dResult  dd  0
```

在第一行，36是行号。下一个数字，0x00000009，是变量存储的数据区中的相对地址。由于*dVar1*是一个双字，需要四个字节，下一个变量的地址是0x0000000D。*dVar1*变量使用4个字节作为地址0x00000009、0x0000000A、0x0000000B和0x0000000C。该行剩余部分是按原始汇编语言源文件输入的数据声明。

0x40660301是内存中放置的值，十六进制表示。$17{,}000{,}000_{10}$是0x01036640。回忆架构为小端，最低有效字节（0x40）放置在最低内存地址。因此，0x40放置在相对地址0x00000009，下一个字节0x66放置在地址0x0000000A，依此类推。这可能会令人困惑，因为乍一看，数字可能看起来是反的或混乱的（取决于如何查看）。

到 help 可视化，记忆图像将如下所示 ws:

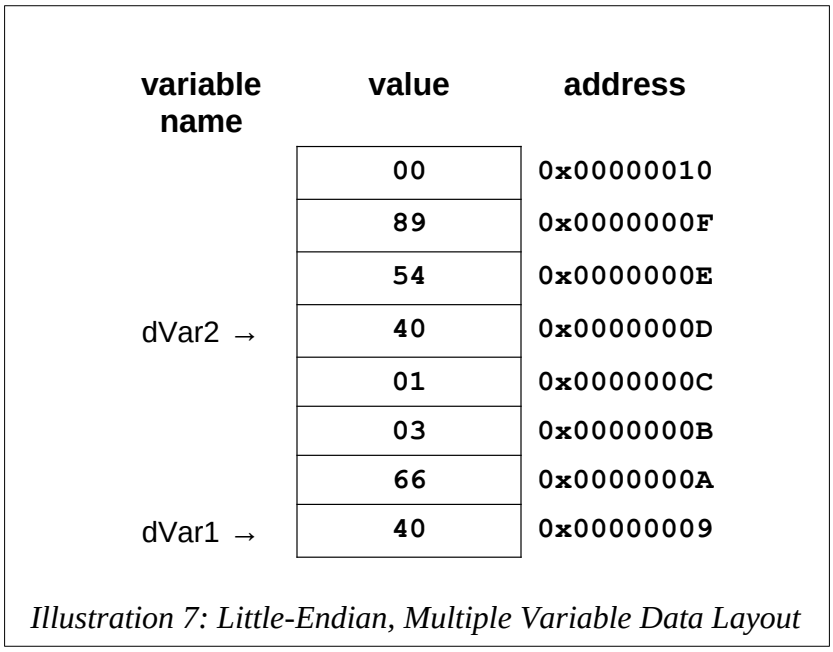


Illustration 7: Little-Endian, Multiple Variable Data Layout

例如，从上一章示例程序中摘录的列表文件文本部分如下：

```
95                                  last:
96 0000005A 48C7C03C000000            mov     rax, SYS_exit
97 00000061 48C7C300000000            mov     rdi, EXIT_SUCCESS
98 00000068 0F05                       syscall
```

再次，左侧的数字是行号。下一个数字，0x0000005A，是代码行将放置的相对地址。

下一个数字，0x48C7C03C000000，是CPU读取并理解的指令的机器语言版本，十六进制表示。该行其余部分是原始汇编语言源指令。

标签 last: 没有机器语言指令，因为标签用于引用特定地址，而不是可执行指令。

5.2.3 双遍汇编器

汇编器[28]将读取源文件，并将程序员键入的每条汇编语言指令转换为CPU所知的1和0的集合，该指令。1和0被称为机器语言。汇编语言指令与二进制机器语言之间存在一一对应关系。这种关系意味着机器语言，以可执行文件的形式，可以转换回人类可读的汇编语言。当然，注释、变量名和标签名将丢失，因此生成的代码可能非常难以阅读。

当汇编器读取汇编语言的每一行时，它为该指令生成机器代码。这对于不执行跳转的指令将很好地工作。然而，对于可能会更改控制流（例如，IF语句、无条件跳转）的指令，汇编器无法转换该指令。例如，给定以下代码片段：

```
        mov     rax, 0
        jmp     skipRest
        ...
        ...
skipRest:
```

28 For more information, refer to: http://en.wikipedia.org/wiki/Assembly_language#Assembler

这被称为前向引用。如果汇编器逐行读取汇编文件，它还没有读取定义 *skipRest* 的那一行。实际上，它甚至不能确定 *skipRest* 是否已经定义。

这种情况可以通过读取汇编源文件两次来解决。整个过程被称为两遍汇编器。每个遍历所需的步骤在以下章节中详细说明。

5.2.3.1 第一次遍历

第一步的步骤取决于特定汇编器的结构。然而，在第一次遍历中执行的一些基本操作包括以下内容：

- 创建符号表
- 展开宏
- 评估常量表达式

宏是一种程序元素，它被扩展成一组程序员预定义的指令。有关更多信息，请参阅第11章，宏。

一个常量表达式是由常量组成的表达式。由于表达式仅包含常量，它可以在汇编时完全评估。例如，假设已定义常量BUFF，以下指令包含一个常量表达式；

```
mov rax, BUFF+5
```

这种常量表达式在大型或复杂程序中常用。

地址分配给程序中的所有语句。符号表是所有程序符号、变量名和程序标签及其在程序中相应地址的列表或表格。

适当的情况下，一些汇编指令在第一次遍历中处理。

5.2.3.2 第二遍

第二次遍历的步骤取决于特定汇编器的结构。然而，第二次遍历中执行的一些基本操作包括以下内容：

- 代码最终生成
- 创建列表文件（如有要求）

- 创建对象文件

术语代码生成指的是将程序员提供的汇编语言指令转换为CPU可执行的机器语言指令。由于一对一的对应关系，这可以用于第一遍或第二遍不使用符号的指令。

应注意的是，根据汇编器设计，大部分代码生成可能在第一次遍历或全部在第二次遍历中完成。无论如何，最终的生成都是在第二次遍历中执行的。这需要使用符号表来检查程序符号并从表中获取适当的地址。

列表文件虽然是可选的，但在调试时可能很有用。如果需要，它将在第二次遍历时生成。

如果没有错误，最终对象文件将在第二次遍历中创建。

5.2.4 汇编指令

汇编指令是给汇编器的指令，指示汇编器执行某些操作。这可能涉及格式或布局。这些指令不会被翻译成CPU的指令。

5.3 链接器

链接器[29]，有时被称为链接编辑器，会将一个或多个目标文件合并成一个可执行文件。此外，根据需要，还会包含用户或系统库中的任何例程。使用的是GNU gold链接器，ld[30]。上一章示例程序的正确链接器命令如下：

```
ld -g -o example example.o
```

Note，-o 是一个短横线小写字母 O，可能会与数字 0 混淆。

-g选项用于通知链接器在最终的执行文件中包含调试信息。这会增加执行文件的大小，但这是允许有效调试所必需的。 -o example指定创建名为*example*（无扩展名）的执行文件。如果省略***-o <fileName>***选项，输出文件将命名为*a.out*（默认）。*example.o*是链接器读取的输入目标文件的名称。需要注意的是，执行文件可以命名为任何名称，不需要与任何输入目标文件具有相同的名称。

29 For more information, refer to: http://en.wikipedia.org/wiki/Linker_(computing)
30 For more information, refer to: http://en.wikipedia.org/wiki/Gold_(linker)

5.3.1 链接多个文件

在编程中，通常通过将大问题分解为更小的问题来解决。这些小问题可以单独解决，可能由不同的程序员完成。

附加的输入目标文件（如有），将按顺序列出，用空格分隔。例如，如果有两个目标文件，*main.o* 和 *funcs.o*，创建包含调试信息的可执行文件名 *example* 的链接命令如下：

```
ld -g -o example main.o funcs.o
```

这通常适用于更大或更复杂的程序。

当使用位于不同外部源文件中的函数时，当前源文件中未定义的任何函数或函数必须声明为 extern。其他源文件中的变量，例如全局变量，也可以通过使用 extern 语句访问，然而数据通常作为函数调用参数进行传递。{v*}

5.3.2 链接过程

链接是将较小的解决方案组合成单个可执行单元的基本过程。如果使用了任何用户或系统库例程，链接器将包含适当的例程。将目标文件和库例程组合成一个单独的可执行模块。机器语言代码从每个目标文件复制到单个可执行文件中。

作为将目标文件合并的一部分，链接器必须根据需要调整可重定位地址。假设有两个源文件，主源文件和包含一些函数的辅助源文件，这两个文件都已汇编成目标文件 *main.o* 和 *funcs.o*。当每个文件被汇编时，对正在汇编的文件外部的例程的调用使用外部汇编器指令声明。代码不可用于外部引用，此类引用在目标文件中标记为外部。列表文件将显示此类可重定位地址的 R。链接器必须满足外部引用。此外，外部引用的最终位置必须放置在代码中。

例如，如果 *main.o* 对象文件在 *funcs.o* 文件中调用一个函数，链接器必须更新调用以适当的地址，如下所示。

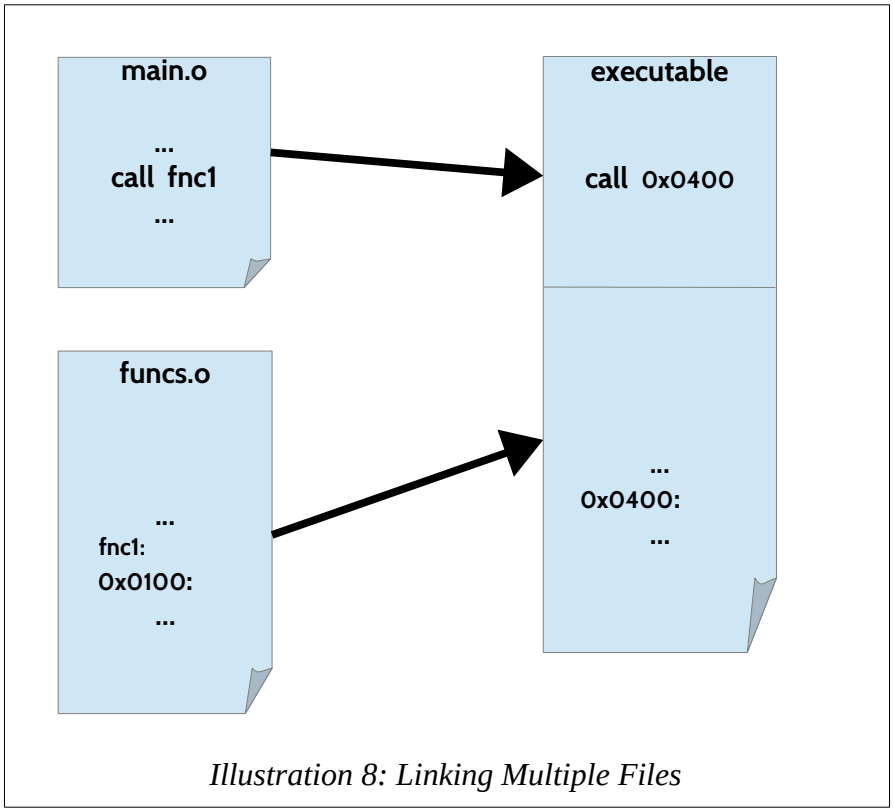


Illustration 8: Linking Multiple Files

这里，函数 ***fnc1*** 对象文件外部，并标记为 R。实际的函数 ***fnc1*** 在 *funcs.o* 文件中，它从 0x0（文本部分）开始相对寻址，因为它不知道主代码。当对象文件合并时，***fnc1*** 的原始相对地址（显示为 0x0100:）被更改为可执行文件中的最终地址（显示为 0x0400:）。链接器必须将此最终地址插入主（显示为 call 0x0400:) 中的调用语句，以完成链接过程并确保函数调用能够正确执行。

这将在代码和数据的所有可重定位地址中发生。

5.3.3 динамическое связывание

Linux操作系统支持动态链接[31]，这允许在程序执行时延迟一些符号的解析。

31 For more information, refer to: http://en.wikipedia.org/wiki/Dynamic_linker

实际指令未放置在可执行文件中，而是在需要时在运行时解析和访问。

虽然更复杂，但这种方法有两个优点：

- 常用库（例如，标准系统库）可以只存储在一个位置，不必在每个二进制文件中重复。
- 如果库函数中的错误得到纠正，所有动态使用它的程序将在下一次执行时从纠正中受益。否则，通过静态链接使用此函数的程序在应用纠正之前必须重新链接。

也存在一些缺点：

- 不兼容的更新库将破坏依赖于该库先前版本行为的可执行文件。

- 一个程序及其使用的库可能作为一个包被认证（例如，关于正确性、文档要求或性能），但如果组件可以被替换，则不会。

在Linux/Unix中，动态链接的对象文件通常具有 .so（共享对象）扩展名。在Windows中，它是 .dll（动态链接库）扩展名。动态链接的更多细节超出了本文的范围。

5.4 组装/链接脚本

当编程时，经常需要多次输入汇编和链接命令以及不同的程序。与其每次都输入汇编（yasm）和链接（ld）命令，可以将它们放在一个文件中，称为脚本文件。然后，可以执行脚本文件，它将仅执行文件中输入的命令。虽然不是必需的，但使用脚本文件可以在处理程序时节省时间并使事情变得更简单。

一个简单的bash[32]汇编/链接脚本如下：

```
#!/bin/bash # 简单的汇编/链接脚本。if [ -z $1 ]; then  echo "用法：  ./asm64 <asmMainFile> (无扩展名)"  exit  fi
```

32 更多信息，请参阅：http://zh.wikipedia.org/wiki/Bash_(Unix_shell)

```
# 验证未输入任何扩展名

如果 [ ! -e "$1.asm" ]; then  echo "错误，未找到 $1.asm。"  echo
"注意，不要输入文件扩展名。"  exit  fi



编译、汇编和链接。

yasm -Worphan-labels -g dwarf2 -f elf64 $1.asm -l $1.lst  ld -g -o $1 $1.o
```

上述脚本应放置在文件中。 对于此示例，文件将命名为 ***asm64*** 并放置在当前工作目录（源文件所在位置）。

一旦创建，需要按照以下方式将执行权限添加到脚本文件中：

```
chmod +x asm64
```

这只需为每个脚本文件做一次。

脚本文件将从命令行读取源文件名。例如，要使用脚本文件组装前一章的示例（命名为 *example.asm*），请输入以下内容：

```
. /asm64 示例
```

".asm" 扩展名不应包含在 *example.asm* 文件中（因为它已在脚本中添加）。脚本文件将汇编和链接源文件，创建列表文件、目标文件和可执行文件。

此或任何脚本文件的使用是可选的。脚本文件的名称可以更改，如下所示期望的。

5.5 加载器

加载器[33]是操作系统的组成部分，它将从辅助存储加载程序到主存储（即主内存）。从广义上讲，加载器将尝试查找，如果找到，则读取格式正确的可执行文件，创建一个新的进程，并将代码加载到内存中，并将程序标记为准备执行。

33 For more information, refer to: http://en.wikipedia.org/wiki/Loader_(computing)

操作系统调度器将决定哪个进程被执行以及何时执行该进程。

加载器通过输入程序名称隐式调用。例如，在先前的名为 *example* 的示例程序中，Linux 命令将是：

./exampleTranslated Text: ./示例

将执行通过之前步骤（汇编和链接）创建的名为 *example* 的文件。由于示例程序没有执行任何输出，控制台将不会显示任何内容。因此，可以使用调试器来检查结果。

5.6 调试器

调试器[34]用于控制程序的执行。这允许进行测试和调试活动。

在之前的示例中，程序计算了一系列计算，但没有输出任何结果。可以使用调试器来检查结果。可执行文件是通过之前描述的汇编和链接命令创建的，并且必须包含-g选项。

GNU DDD调试器，它为GNU命令行调试器gdb提供了一个可视化前端。DDD网站和文档在第一章“简介”的参考文献部分有说明。

由于调试器的复杂性和重要性，专门提供了一章用于调试。

5.7 练习

以下是本章的一些问题。

5.7.1 测验问题

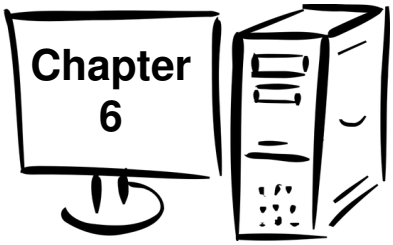
以下是几个测验问题。

1) 汇编语言和机器语言之间的关系是什么？2) 汇编器的第一次遍历执行了哪些操作？3) 汇编器的第二次遍历执行了哪些操作？4) 链接器执行了哪些操作?

34 For more information, refer to: http://en.wikipedia.org/wiki/Debugger

5) 加载器执行哪些操作？6) 提供一个 *constant expression* 的例子。7) 绘制整个组装、链接和加载过程的图。8) 何时将共享对象文件与程序链接？9) 符号表中包含什么（两件事）？

My software never has bugs. It just develops random features.



6.0 DDD 调试器

调试器允许用户控制程序执行、检查变量、其他内存（即堆栈空间）以及显示程序输出（如果有）。开源的GNU数据显示调试器（DDD[35]）是GNU调试器（GDB[36]）的图形前端，并且广泛可用。如果需要，其他调试器也可以轻松使用。

仅在本章中介绍了基本的调试器命令。DDD 调试器具有许多此处未涵盖的功能和选项。随着您经验的积累，回顾第 1 章中引用的 DDD 文档，以了解更多关于附加功能的内容，将有助于提高整体调试效率。

DDD 功能可以通过各种插件进行扩展。插件不是必需的，本章不会涉及。

本章介绍使用GNU DDD调试器作为工具。本章未涉及如何调试程序的逻辑过程。

6.1 开始 DDD

ddd调试器通过可执行文件启动。程序必须使用正确的选项（如前一章所述）进行汇编和链接。例如，使用前一个示例程序*example*，命令将是：

```
ddd 示例
```

启动 DDD/GDB 后，应显示类似于以下屏幕的界面（显示适当的源代码）。

35 For more information, refer to: http://en.wikipedia.org/wiki/Data_Display_Debugger
36 For more information, refer to: http://en.wikipedia.org/wiki/GNU_Debugger

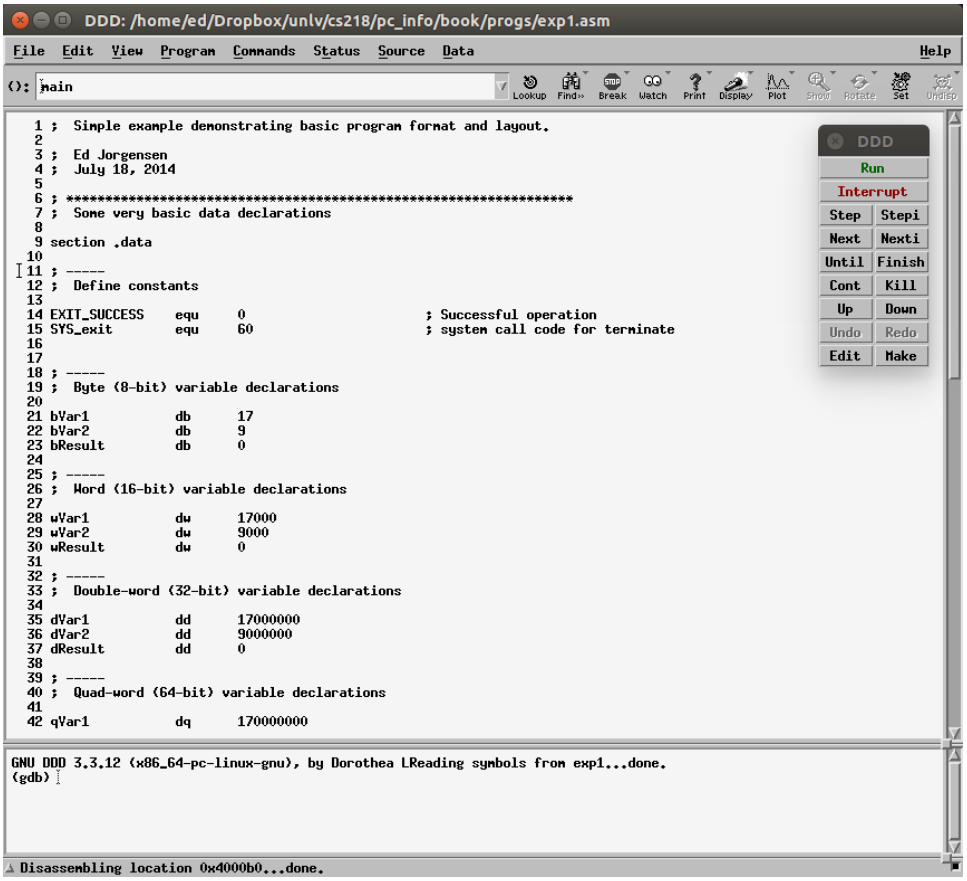


Illustration 9: Initial Debugger Screen

如果代码没有以上所示类似方式显示，应验证汇编和链接步骤。具体来说，必须在汇编和链接步骤中包含 -g 标志。

内置帮助可通过点击帮助菜单项（右上角）获取。DDD 和 GDB 手册可在课程网页上的虚拟图书馆链接中找到。要退出 DDD/GDB，请选择文件 → 退出（从顶部菜单栏）。

6.1.1 DDD 配置设置

一些额外的 DDD/GDB 配置设置建议包括:

> 编辑 → 预设 → 通用 → 禁用 X 警告 编辑 → 预设 → 源 → 显示源行号

这些不是必需的，但可以使使用调试器更容易。如果设置，这些选项将被保存并记住，以便在后续使用调试器时（在同一台机器上）使用。

6.2 使用DDD的程序执行

要执行程序，请从命令工具菜单（如下所示）点击“运行”按钮。或者，您可以在(gdb)提示符（底部GDB控制台窗口）中输入run。然而，这将完全执行程序，完成后，结果将被重置（并丢失）。

6.2.1 设置断点

为了控制程序执行，将需要在用户选择的位置设置一个断点（执行暂停位置），以暂停程序。这可以通过选择源位置（要停止的行）来实现。对于这个例子，我们将停止在第95行。

断点可以以三种方式之一进行设置:

- 在行号上右键单击并选择：*Set Breakpoint*
- 在GDB命令控制台，在(gdb)提示符下，输入：break last
- 在GDB命令控制台，在(gdb)提示符下，输入：break 95

在以下示例中，第94行是一个没有指令的标签。如果在标签上设置断点，它将在下一个可执行指令处停止（本例中的第95行）。

当设置正确时，“停止”图标将出现在行号左侧（如图所示）。

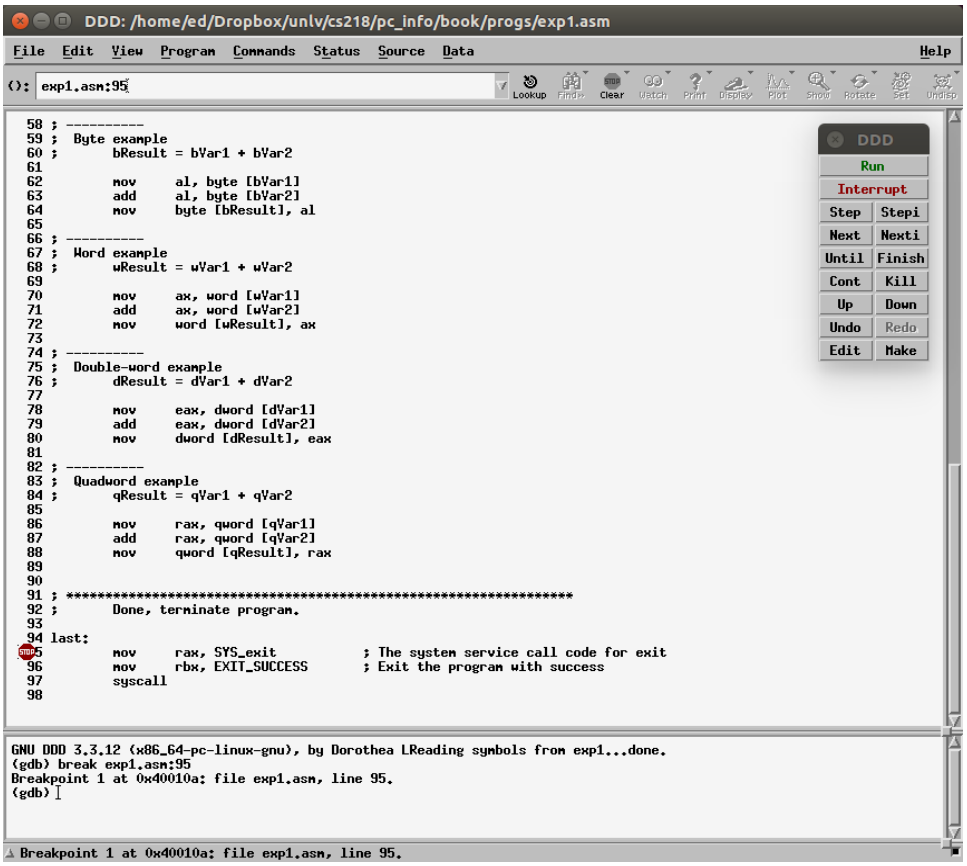


Illustration 10: Debugger Screen with Breakpoint Set

DDD/GDB命令可以在底部窗口（在**(gdb)**提示符下）中随时输入。如果需要，可以设置多个断点。

6.2.2 执行程序

一旦开始调试器，为了有效地使用调试器，必须设置一个初始断点。

一旦设置断点，可以通过点击“运行”菜单窗口或输入“run”在(gdb)提示符下执行运行命令。程序将执行到，*but not including*带有绿色箭头的语句。

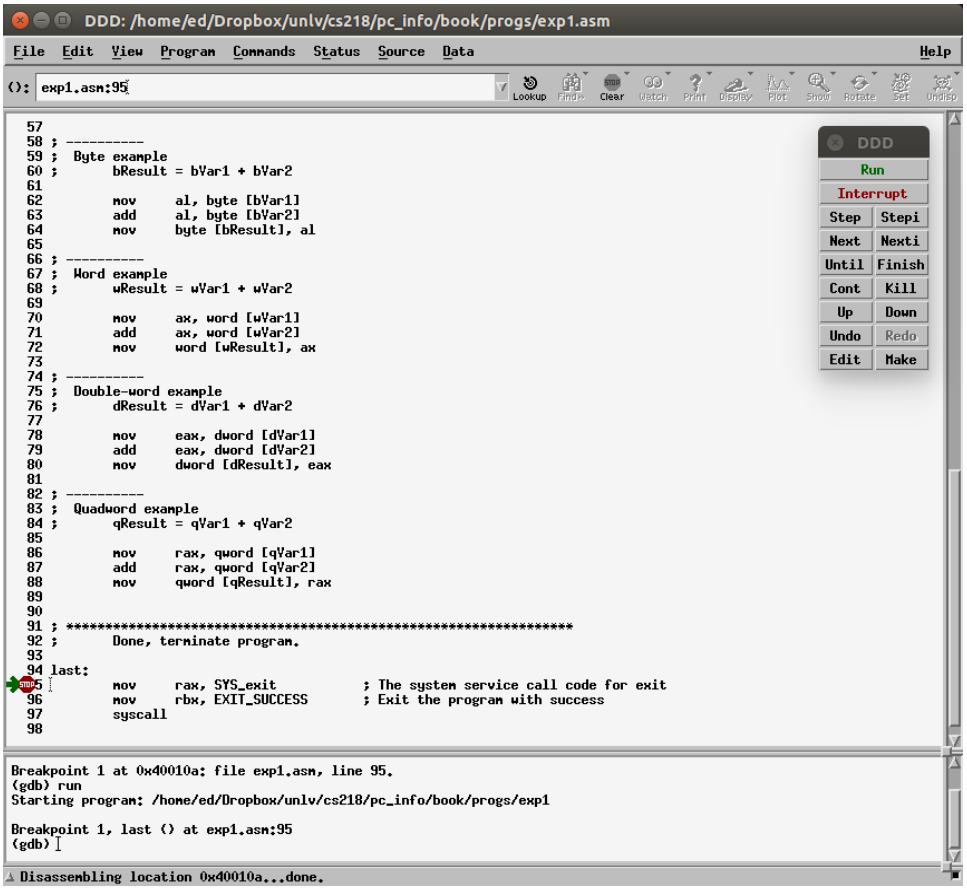


Illustration 11: Debugger Screen with Green Arrow

断点用左侧的停止标志表示，当前位置用绿色箭头表示（见上面示例）。具体来说，绿色箭头指向 *next instruction to be executed*。也就是说，绿色箭头所指的语句尚未执行 ***not***。

6.2.2.1 运行/继续

根据需要，可以设置额外的断点。然而，点击“运行”命令将从头开始重新执行并在初始断点处停止。

Illustration 12: DDD Command Bar

在初始运行命令之后，要继续到下一个断点，必须使用继续命令（通过点击“Cont”菜单窗口或输入(gdb)提示符下的“cont”）。也可以通过逐行输入单步或下一个命令来逐行执行单行（通过点击“Step”或“Next”菜单窗口或输入(gdb)提示符下的“step”或“next”）。

6.2.2.2 下一页 / 步骤

下一个命令将执行到下一个指令。这包括在必要时执行整个函数。步骤命令将执行一步，如果需要则进入函数。对于单个非函数指令，下一个和步骤命令之间没有区别。

6.2.3 显示寄存器内容

最简单查看寄存器内容的方法是使用寄存器窗口。寄存器窗口默认不显示，但可以通过选择状态 → 寄存器（从顶部菜单栏）来查看。当显示时，寄存器窗口将按寄存器名称（左侧列）显示寄存器内容，同时在十六进制（中间列）和无符号十进制（右侧列）中显示。由于右侧列将显示无符号

整个寄存器的值在数据为有符号时可能会令人困惑（因为它将以无符号形式显示）。此外，对于某些寄存器，如rbp和rsp，两列都显示为十六进制（因为它们通常用于地址）。在下一节中描述的检查内存命令将允许对显示值的格式（例如，有符号、无符号、十六进制）进行更具体的控制。

Illustration 13: Register Window

根据机器和屏幕分辨率，注册窗口可能需要调整大小以查看全部内容。

注册窗口的第三列通常显示十进制quadword表示，除非是某些特殊用途的寄存器（rbp和rsp）。有符号quadword的十进制表示可能并不总是有意义的。例如，如果使用无符号数据（如地址），则有符号表示将是错误的。此外，当使用字符数据时，有符号表示将没有意义。

默认情况下，仅显示整数寄存器。点击“所有寄存器”框将添加浮点寄存器到显示中。查看需要在该寄存器窗口内向下滚动。

6.2.4 DDD/GDB 命令摘要

以下表格提供了最常见的DDD命令的小子集。在输入时，大多数命令可以缩写。例如，退出可以缩写为q。命令和缩写显示在表中。

Command	Description
quit | q	Quit the debugger.
break <label/addr> **| b <label/addr>**	Set a break point (stop point) at <label> or <address>.
run <args> | r <args>	Execute the program (to the first breakpoint).
continue | c	Continue execution (to the next breakpoint).
continue <n> | c <n>	Continue execution (to the next breakpoint), skipping ***n***-1 crossing of the breakpoint. This is can be used to quickly get to the n^{th} iteration of a loop.
step | s	Step into next instruction (i.e., steps into function/procedure calls).
next | n	Next instruction (steps through function/procedure calls).
F3	Re-start program (and stop at first breakpoint).
where	Current activation (call depth).
x/<n><f><u> $rsp	Examine contents of the stack.

Command	Description
x/<n><f><u> &<variable>	Examine memory location <variable> <n> number of locations to display, 1 is default. <f> format: d – decimal (signed) x – hex u – decimal (unsigned) c – character s – string f – floating-point <u> unit size: b – byte (8-bits) h – halfword (16-bits) w – word (32-bits) g – giant (64-bits)
source <filename>	Read commands from file <filename>.
set logging file <filename>	Set logging file to <filename>, default is *gdb.txt*.
set logging on	Turn logging (to a file) on.
set logging off	Turn logging (to a file) off.
set logging overwrite	When logging (to a file) is turned on, overwrite previous log file (if any).

更多信息可以通过内置的帮助功能或从ddd网站上的文档（参见第1章）获取。

6.2.4.1 DDD/GDB 命令，示例

例如，给定以下数据声明:

```
bnum1       db      5
wnum2       dw      -2000
dnum3       dd      100000
qnum        dq      1234567890
class       db      "Assembly", 0
twopi       dd      6.28
```

假设 ***signed data***，检查内存命令的命令如下:

x/db &bnum1 x/
dh &wnum2 x/d
w &dnum3 x/dg
&qnum x/s &cla
ss x/f &twopi

如果使用了不适当的内存转储命令（即，不正确的尺寸），*there is no error message* 和调试器将显示所请求的内容（即使它没有意义）。检查变量将需要根据数据声明使用适当的内存转储命令。可以通过屏幕顶部的菜单访问其他选项。

要在DDD中显示一个数组，使用基本检查内存命令。

x/**<**n**><**f**><**u**>** &**<**变量**>**

例如，假设声明为:

```
list1       dd      100001, -100002, 100003, 100004, 100005
```

检查内存命令如下:

x/5dw &list1

在5是数组长度。d表示有符号数据（u将表示无符号数据）。w表示32位大小的数据（这是在源文件中dd、define double定义所声明的）。&list1指的是变量的地址。*Note*，地址指向第一个元素（仅第一个元素）。因此，可以显示比实际在数组中声明的元素少或多的元素。

基本检查内存命令可以直接与内存地址一起使用（而不是变量名）。例如：

```
x/dw 0x600d44
```

地址通常以十六进制显示，因此需要0x才能直接以显示的格式输入十六进制地址。

6.2.5 显示栈内容

在某些情况下，显示堆栈内容可能很有用。堆栈通常由64位无符号元素组成。虽然使用检查内存命令，但地址位于rsp寄存器中（不是变量名）。显示堆栈当前顶部的检查内存命令如下：

```
x/ug $rsp
```

显示栈顶前6项的检查内存命令如下：

```
x/6ug $rsp
```

由于栈的实现，显示的第一个项目始终是栈的当前顶部。

6.2.6 交互式调试器命令文件

由于数据显示命令必须正确输入，这可能会很繁琐。此外，如果命令输入错误，通常没有错误信息，这可能会令人困惑。为了帮助减少错误，可以将正确的执行和显示命令存储在文本文件中。然后，调试器可以从文件中读取命令（而不是手动输入）。虽然结果通常显示在屏幕上，但可以将结果重定向到输出文件。这有助于方便地查看。

例如，一些典型的调试器命令来设置断点、运行程序、显示一些变量以及将输出重定向到日志文件可能如下所示：

```
#---------------------------------- # 调试器输入脚本 #---
---------------------------------- echo \n\n break last run set
pagination off
```

设置日志文件 out.txt 设置日志覆盖 设置日志开启 设置提示
回显 ---------- \n 显示变量 \n \n x/100dw &
list x/dw &length \n x/dw &listMin x/dw &listMid x/dw &listMa
x x/dw &listSum x/dw &listAve \n \n 设置日志关闭 退出

*Note 1;*此示例假定在源程序中定义了一个标签 'last'（如在示例程序中所做的那样）。

*Note 2;*此示例退出调试器。如果不想这样做，可以删除'quit'命令。当从输入文件退出时，调试器可能会请求用户确认退出（是或否）。

这些命令应放置在文件中（例如 *gdbIn.txt*），以便在调试器中读取。

6.2.6.1 调试器命令文件（非交互式）

The debugger command to read a file is "source <filename>".For example, if the command file is named *gdbIn.txt*,

(gdb) 源 gdbIn.txt

基于上述命令，输出将被放置在文件 *out.txt* 中。输出文件名可以按需更改。

每个程序将需要一组自定义的输入命令，这取决于该程序中特定的变量及其相关大小。当程序接近工作状态时，调试器输入命令文件才会非常有用。程序崩溃和其他更严重的错误需要交互式调试来确定具体的错误或错误。

6.2.6.2 非交互式调试器命令文件

可以直接获取输出文件，无需交互式DDD会话。在命令行中输入以下命令，将在给定程序上执行输入文件中的命令，创建输出文件并退出程序。

gdb 程序 **<**gdbIn.txt

这将创建输出文件（如输入文件 *gdbIn.txt* 中指定的）并退出调试器。一旦创建输入文件，这是获取工作程序最终输出文件的最快选项。再次强调，这只有在程序正在运行或非常接近正确运行时才有用。

6.3 练习

以下是本章的一些测验问题。

6.3.1 测验问题

以下是几个测验问题。

1) 调试器是如何启动的（从命令行）？ 2）在汇编和链接步骤中需要哪个选项才能确保程序易于调试？ 3）运行命令具体做什么？ 4）继续命令具体做什么？

5) 注册窗口是如何显示的?

6) 注册窗口中有三列。第一列显示寄存器。其他两列显示什么？ 7）一旦开始调试器，用户如何退出？ 8）描述如何设置断点（多种方法）。9）调试器命令是什么，可以从文件中读取调试器命令？ 10）当DDD显示一个指向指令的绿色箭头时，这意味着什么？ 11）提供显示以下变量的调试器命令，以十进制形式显示。1. bVar1（字节大小变量）

2. wVar1（字大小变量）3. dVar1（双字大小变量）4. qVar1（四字大小变量）5. bArr1（30个字节数组）6. wArr1（50个字数数组）7. dArr1（75个双字数数组）

12) 提供调试器命令以十六进制格式显示以下每个变量。

1. bVar1（字节大小变量）2. wVar1（字大小变量）3. dVar1（双字大小变量）4. qVar1（四字大小变量）5. bArr1（30个元素的字节数组）6. wArr1（50个元素的字数组）7. dArr1（75个元素的双字数组）

13) 调试器命令如何显示栈顶的值?

14) 调试器命令如何显示栈顶的五个（5）值?

6.3.2 建议项目

以下是本章的一些建议项目。

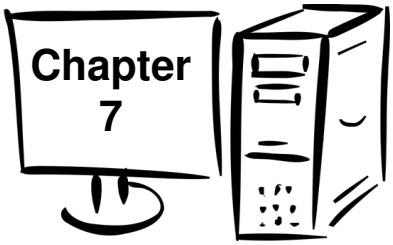
1) 在第4章程序格式中输入示例程序。按照第5章工具链的描述汇编和链接程序。按照本章所述执行调试器。在标签last处设置断点并执行程序（到该断点）。交互式验证所进行的计算结果是否正确。这需要输入适当的调试器检查内存命令（基于变量大小）。

2) 在完成上一个问题后，创建一个调试器输入文件，该文件将输出发送到文本文件，设置一个断点，执行程序，并显示每个变量的结果（基于适当的变量大小）。

执行调试器并读取源文件。 检查输入文件是否正确工作，以及根据输出文件中显示的结果，程序的计算是否正确。

3) 创建一个汇编和链接脚本文件，如第5章工具链中所述。使用该脚本汇编和链接程序。确保脚本正确汇编和链接。

Why are math books sad?
Because they have so many problems.



7.0 指令集概述

本章节为x86-64指令集的一个简单子集提供基本概述，重点关注整数操作。这仅涵盖本文本范围内讨论的主题和程序所需的指令子集。这将排除一些更高级的指令和限制模式指令。有关所有处理器指令的完整列表，请参阅第1章中列出的参考文献。

指示按以下顺序呈现:

- 数据移动
- 转换说明
- 算术指令
- 逻辑指令
- 控制指令

函数调用说明在第12章“函数”中讨论。

本文本中涵盖的指令完整列表位于附录B中供参考。

7.1 符号约定

本节总结了在此文本中使用的符号，这在技术文献中相当常见。一般来说，一条指令将包括指令或操作本身（即，加、减、乘等）和***operands***。操作数指的是数据（要操作的数据）来自哪里以及/或结果要放置在哪里。

7.1.1 操作数表示法

以下表格总结了本文档其余部分使用的符号约定。

Operand Notation	Description
<reg>	Register operand. The operand must be a register.
<reg8>, <reg16>, <reg32>, <reg64>	Register operand with specific size requirement. For example, **reg8** means a byte sized register (e.g., **al**, **bl**, etc.) only and **reg32** means a double-word sized register (e.g., **eax**, **ebx**, etc.) only.
<dest>	Destination operand. The operand may be a register or memory. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
<RXdest>	Floating-point destination register operand. The operand must be a floating-point register. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
<src>	Source operand. Operand value is unchanged after the instruction.
<imm>	Immediate value. May be specified in decimal, hex, octal, or binary.
<imm8>, <imm16>, <imm32>, <imm64>	Immediate value of specified size. May be specified in decimal, hex, octal, or binary.
<imm8/16/32>	Immediate value either 8-bits, 16-bits, or 32-bits. May be specified in decimal, hex, octal, or binary.
<mem>	Memory location. May be a variable name or an indirect reference (i.e., a memory address).
<op> or <operand>	Operand, register or memory.
<op8>, <op16>, <op32>, <op64>	Operand, register or memory, with specific size requirement. For example, **op8** means a byte sized operand only and **reg32** means a double-word sized operand only.
<label>	Program label.

默认情况下，立即值是十进制或基数为10。可以使用十六进制或基数为16的立即值，但必须以0x开头以指示该值是十六进制。例如，15_{10}可以以十六进制形式输入为0x0F。

参考第8章，寻址方式，以获取有关内存位置和间接引用的更多信息。

7.2 数据移动

通常，必须将数据从RAM移动到CPU寄存器中才能进行操作。一旦计算完成，结果可以从寄存器复制并放置到变量中。示例程序中有一些简单的公式执行这些步骤。这种基本的数据移动操作使用移动指令执行。

移动指令的一般形式是:

mov **<**目标**>**，**<**源**>**

源操作数从源操作数复制到目标操作数。源操作数的值保持不变。目标操作数和源操作数必须具有相同的大小（都是字节、都是字等）。目标操作数不能是立即数。两个操作数都不能是内存。如果需要内存到内存的操作，必须使用两条指令。

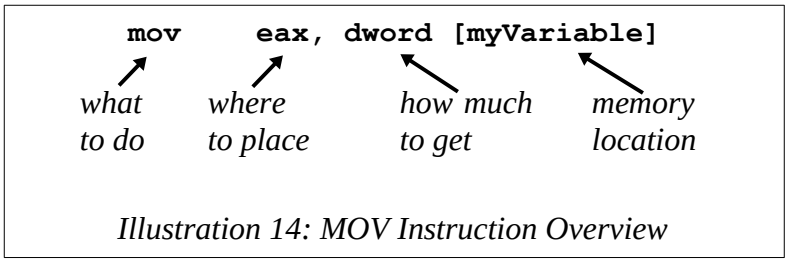


Illustration 14: MOV Instruction Overview

当目标寄存器操作数是双字大小且源操作数是双字大小时，四字寄存器的最高阶双字被设置为零。这仅适用于目标操作数是双字大小的整数寄存器。

具体来说，如果执行以下操作，

```
mov     eax, 100            ; eax = 0x00000064
mov     rcx, -1             ; rcx = 0xffffffffffffffff
mov     ecx, eax            ; ecx = 0x00000064
```

最初，rcx寄存器被设置为-1（即全部为0xF）。当从eax寄存器（100_{10}）的正数移动到rcx寄存器时，quadword寄存器rcx的高阶部分被设置为0，覆盖了之前指令中的1。

移动指令总结如下：

Instruction	Explanation
`mov  <dest>, <src>`	Copy source operand to the destination operand. *Note 1*, both operands cannot be memory. *Note 2*, destination operands cannot be an immediate. *Note 3*, for double-word destination and source operand, the upper-order portion of the quadword register is set to 0.

Examples:	`mov  ax, 42`
	`mov  cl, byte [bvar]`
	`mov  dword [dVar], eax`
	`mov  qword [qVar], rdx`

更完整的指令列表位于附录B中。

例如，假设以下数据声明：

```
dValue      dd      0
bNum        db      42
wNum        dw      5000
dNum        dd      73000
qNum        dq      73000000
bAns        db      0
wAns        dw      0
dAns        dd      0
qAns        dq      0
```

执行以下基本操作:

dValue = 27 bA
ns = bNum wAn
s = wNum dAns
= dNum qAns =
qNum

以下说明可用于:

```
mov     dword [dValue], 27          ; dValue = 27

mov     al, byte [bNum]
mov     byte [bAns], al             ; bAns = bNum

mov     ax, word [wNum]
mov     word [wAns], ax             ; wAns = wNum

mov     eax, dword [dNum]
mov     dword [dAns], eax           ; dAns = dNum

mov     rax, qword [qNum]
mov     qword [qAns], rax           ; qAns = qNum
```

对于某些指令，包括上述指令，可以省略显式类型指定（例如，*byte*，*word*，*dword*，*qword*），因为另一个操作数将明确定义大小。在文本中，为了保持一致性和良好的编程实践，将包含该指定。

7.3 地址和值

仅通过括号([])访问内存。省略括号将无法访问内存，而是获取项目地址。例如:

```
mov     rax, word [wNum]            ; value of wNum in rax
mov     rax, wNum                   ; address of wNum in rax
```

由于省略括号不是错误，汇编器不会生成错误消息或警告。这可能导致混淆。

此外，可以通过加载有效地址或lea指令来获取变量的地址。加载有效地址指令总结如下:

Instruction	Explanation
`lea  <reg64>, <mem>`	Place address of **<mem>** into **reg64**.
Examples:	`lea  rcx, byte [bvar]` `lea  rsi, dword [dVar]`

更完整的指令列表位于附录B中。

第八章 访问模式中提供了更多信息及大量示例。

7.4 转换说明

有时需要将一个大小转换为另一个大小。例如，一个字节可能需要转换为双字，以便在公式中进行某些计算。转换过程取决于操作数的大小和类型。以下各节总结了转换是如何进行的。

7.4.1 狭义转换

窄化转换是将较大类型转换为较小类型（即，从字到字节或从双字到字）。

不需要特殊指令来缩小转换。可以直接访问内存位置或寄存器的下部分。例如，如果将值50（0x32）放入rax寄存器，可以直接访问al寄存器以获取以下值：

```
mov rax, 50mov byte [bVal], a
l
```

这个例子是合理的，因为50的值可以放入一个字节值中。然而，如果将500（0x1f4）的值放入rax寄存器，al寄存器仍然可以访问。

```
mov rax, 500 mov byte [bVal],
al
```

在这个例子中，***bVal***变量将包含0xf4，这可能导致结果不正确。程序员负责确保执行适当的缩窄转换。与编译器不同，不会生成警告或错误消息。

7.4.2 宽化转换

类型扩展是从较小类型到较大类型（例如，字节到字或字到双字）。由于大小正在扩展，最高位必须根据原始值的符号来设置。因此，必须知道数据类型，有符号或无符号，并使用适当的过程或指令。

7.4.2.1 无符号转换

对于无符号扩展转换，内存位置或寄存器的上半部分必须设置为零。由于无符号值只能为正，高位位只能为零。例如，要将al寄存器中的字节值50转换为rbx中的quadword值，可以执行以下操作。

```
mov       al, 50
mov       rbx, 0
mov       bl, al
```

由于rbx寄存器被设置为0，然后将其低8位设置为al的值（本例中为50），因此整个64位的rbx寄存器现在是50。

此 一般 过程 可 在 内存 或 其他 寄存器 上 执行。 确保 值 适用于 正在 使用 的 数据 大小 是 程序员的 责任。

未签名的从小尺寸到较大尺寸的转换也可以使用特殊的移动指令来完成，如下所示：

movzx **<**dest**>**, **<**src**>**

这将用零填充高阶位。movzx指令不允许使用双字源操作数来指定四字目的操作数。如前所述，使用双字寄存器目的操作数和双字源操作数的mov指令将使四字目的寄存器的高阶双字为零。

指令执行无符号扩展转换的摘要如下：

Instruction	Explanation
`movzx  <dest>, <src>`	Unsigned widening conversion. *Note 1*, both operands cannot be memory.

`movzx <reg16>, <op8>`
`movzx <reg32>, <op8>`
`movzx <reg32>, <op16>`
`movzx <reg64>, <op8>`
`movzx <reg64>, <op16>`
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.

Examples:	`movzx cx, byte [bVar]`
	`movzx dx, al`
	`movzx ebx, word [wVar]`
	`movzx ebx, cx`
	`movzx rbx, cl`
	`movzx rbx, cx`

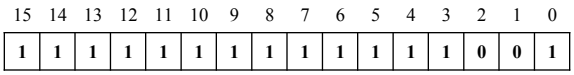
更完整的指令列表位于附录B中。

7.4.2.2 签名转换

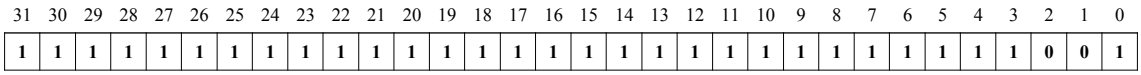
对于有符号的宽转换，最高位必须设置为0或1，具体取决于原始值是正数还是负数。

这是通过符号扩展操作完成的。具体来说，原始值的最高位指示该值是正数（用0表示）还是负数（用1表示）。原始值的最高位被扩展到新值、扩宽数值的高位中。

例如，假设 ax 寄存器设置为 -7（0xfff9），位将如下设置:



由于值是负数，最高位（位15）是1。要将ax寄存器中的字值转换为eax寄存器中的双字值，最高位（本例中的1）被扩展或复制到整个高字（位31-16），结果如下:



存在一系列专用指令，用于将 A 寄存器中的有符号值从较小的尺寸转换为较大的尺寸。这些指令仅在 A 寄存器上工作，有时使用 D 寄存器来存储结果。例如，cwd 指令将 ax 寄存器中的有符号值转换为 dx 寄存器（高位部分）和 ax 寄存器（低位部分）中的双字值。这通常按约定写成 dx:ax。cwde 指令将 ax 寄存器中的有符号值转换为 eax 寄存器中的双字值。

更通用的有符号转换，从小尺寸到较大尺寸，也可以通过一些特殊移动指令来完成，如下所示：

```
movsx       <dest>, <src>
movsxd      <dest>, <src>
```

该操作将对源参数执行符号扩展操作。movsx指令是通用形式，而movsxd指令用于允许双字源操作数与四字目的操作数。

指令执行有符号扩展转换的摘要如下：

Instruction	Explanation
cbw	Convert byte in **al** into word in **ax**. *Note*, only works for **al** to **ax** register.
Examples:	**cbw**
cwd	Convert word in **ax** into double-word in **dx:ax**. *Note*, only works for **ax** to **dx:ax** registers.
Examples:	**cwd**
cwde	Convert word in **ax** into double-word in **eax**. *Note*, only works for **ax** to **eax** register.
Examples:	**cwde**
cdq	Convert double-word in **eax** into quadword in **edx:eax**. *Note*, only works for **eax** to **edx:eax** registers.
Examples:	**cdq**

Instruction	Explanation
`cdqe`	Convert double-word in **eax** into quadword in **rax**. *Note*, only works for **rax** register.
Examples:	`cdqe`
`cqo`	Convert quadword in **rax** into double-quadword in **rdx:rax**. *Note*, only works for **rax** to **rdx:rax** registers.
Examples:	`cqo`

`movsx  <dest>, <src>`
 	
`movsx  <reg16>, <op8>`
`movsx  <reg32>, <op8>`
`movsx  <reg32>, <op16>`
`movsx  <reg64>, <op8>`
`movsx  <reg64>, <op16>`
`movsxd  <reg64>, <op32>`
	Signed widening conversion (via sign extension).
	Note 1, both operands cannot be memory.
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.
	Note 4, special instruction (*movsxd*) required for 32-bit to 64-bit signed extension.

Examples:	`movsx  cx, byte [bVar]`
	`movsx  dx, al`
	`movsx  ebx, word [wVar]`
	`movsx  ebx, cx`
	`movsxd  rbx, dword [dVar]`

更完整的指令列表位于附录B中。

7.5 整数算术指令

整数算术指令执行整数值的加法、减法、乘法和除法等算术运算。以下各节介绍了基本的整数算术运算。

7.5.1 添加

整数加法指令的一般形式如下：

添加 **<**目标**>**，**<**源**>**

操作执行以下操作：

<目标> = <目标> + <源>

具体来说，源操作数和目标操作数相加，结果放在目标操作数中（覆盖之前的值）。源操作数的值保持不变。目标操作数和源操作数必须具有相同的大小（都是字节、都是字等）。目标操作数不能是立即数。两个操作数都不能是内存。如果需要执行内存到内存的加法操作，必须使用两条指令。

例如，假设以下数据声明：

```
bNum1       db      42
bNum2       db      73
bAns        db      0

wNum1       dw      4321
wNum2       dw      1234
wAns        dw      0

dNum1       dd      42000
dNum2       dd      73000
dAns        dd      0

qNum1       dq      42000000
qNum2       dq      73000000
qAns        dq      0
```

执行以下基本操作：

```
bAns = bNum1 + bNum2 wA
ns = wNum1 + wNum2 dAns
= dNum1 + dNum2 qAns = q
Num1 + qNum2
```

以下说明可用于：

```
; bAns = bNum1 + bNum2 mov al
, byte [bNum1] add al, byte [bNu
m2] mov byte [bAns], al
```

```
; wAns = wNum1 + wNum2 mov ax,
word [wNum1] add ax, word [wNum
2] mov word [wAns], ax ; dAns = dN
um1 + dNum2 mov eax, dword [dNu
m1] add eax, dword [dNum2] mov d
word [dAns], eax ; qAns = qNum1 +
qNum2 mov rax, qword [qNum1] add
rax, qword [qNum2] mov qword [qA
ns], rax
```

对于某些指令，包括上述指令，可以省略显式类型指定（例如，*byte*，*word*，*dword*，*qword*），因为其他操作数将明确定义大小。这只是为了保持一致性以及良好的编程实践。

除了基本的加法指令外，还有一个加一指令，该指令会将一个数加到指定的操作数上。加一指令的一般形式如下：

```
inc <操作数>
```

操作如下：

```
<operand> = <operand> + 1
```

结果与使用加法指令（并加一）完全相同。当使用内存操作数时，需要显式指定类型（例如，*byte*，*word*，*dword*，*qword*）以清楚地定义大小。

例如，假设以下数据声明：

```
bNum        db      42
wNum        dw      4321
dNum        dd      42000
qNum        dq      42000000
```

执行以下基本操作：

```
rax = rax + 1 bNum =
bNum + 1
```

```
wNum = wNum + 1 d
Num = dNum + 1 qN
um = qNum + 1
```

以下说明可用于:

```
inc rax ; rax = rax + 1 inc byte [bNum] ; bNum = bNum + 1 inc word [wN
um] ; wNum = wNum + 1 inc dword [dNum] ; dNum = dNum + 1 inc qw
ord [qNum] ; qNum = qNum + 1
```

加法指令对有符号和无符号数据操作相同。程序员负责确保数据类型和大小适用于所进行的操作。

整数加法指令总结如下:

Instruction	**Explanation**

`add  <dest>, <src>`
 	
`add  <reg>, <reg>`
`add  <reg>, <imm8/16/32>`
`add  <reg>, <mem>`
`add  <mem>, <reg>`
`add  <mem>, <imm8/16/32>`
	Add two operands, (<**dest**> + <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

Examples:	`add  cx, word [wVvar]`
	`add  rax, 42`
	`add  dword [dVar], eax`
	`add  qword [qVar], 300`

`inc  <operand>`	Increment <**operand**> by 1. *Note*, <**operand**> cannot be an immediate.

Examples:	`inc  word [wVvar]`
	`inc  rax`
	`inc  dword [dVar]`
	`inc  qword [qVar]`

更完整的指令列表位于附录B中。

7.5.1.1 带进位加法

加法进位是一种特殊的加法指令，它将包括来自先前加法操作的进位。这在添加非常大的数字时很有用，特别是当数字大于机器的寄存器大小时。

使用进位加法是标准的。例如，考虑以下操作。

```
    17 +
    25 ---
    -  42
```

如您所忆，最低有效位（7和5）首先相加。12的结果记为2，并带有1的进位。最高有效位（1和2）与之前的进位（本例中为1）相加，得到4。

因此，需要两个加法操作。由于最低有效位不可能产生进位，因此使用常规加法指令。第二个加法操作需要包括前一个操作可能的进位，并且必须使用带进位加法指令来完成。此外，带进位加法必须紧接在初始加法操作之后，以确保rFlag寄存器不会被无关指令改变（从而可能改变进位位）。

对于汇编语言程序，使用加法指令将最低有效四字长（LSQ）相加，然后立即使用adc指令将最高有效四字长（MSQ）相加，这将相加四字长并包括前一次加法操作的进位。

整数带进加法指令的一般形式如下：

```
adc  <目标>, <源>
```

操作执行以下操作：

```
<目标> = <目标> + <源> + <进位位>
```

具体来说，源操作数和目的操作数以及进位位相加，结果放置在目的操作数中（覆盖之前的值）。进位位是rFlag寄存器的一部分。源操作数的值保持不变。目的操作数和源操作数必须具有相同的大小（都是字节、都是字等）。目的操作数不能是立即数。两个操作数都不能是内存。如果需要执行内存到内存的加法操作，必须使用两条指令。

例如，给定以下声明；

```
dquad1      ddq     0x1A000000000000000
dquad2      ddq     0x2C000000000000000
dqSum       ddq     0
```

每个变量 ***dquad1***、***dquad2*** 和 ***dqSum*** 都是 128 位，因此将超过机器 64 位寄存器的大小。然而，对于每个 128 位值，可以使用两个 64 位寄存器。这需要两个移动指令，每个 64 位寄存器一个。例如，

```
mov rax, qword [dquad1] mov rdx, qword
[dquad1+8]
```

第一次移动到rax寄存器访问了128位变量中的前64位。第二次移动到rdx寄存器访问了128位变量的下一个64位。这是通过使用变量起始地址***dquad1***并加上8字节来实现的，从而跳过了前64位（或8字节）并访问下一个64位。

如果将LSQ相加，然后包括任何进位将MSQ相加，就可以正确地获得128位的结果。例如，

```
mov rax, qword [dquad1]  mov rdx, qword
[dquad1+8]  add rax, qword [dquad2]  adc r
dx, qword [dquad2+8]  mov qword [dqSum
], rax  mov qword [dqSum+8], rdx
```

最初，***dquad1***的LSQ被放置在rax中，MSQ被放置在rdx中。然后，add指令将64位的rax与***dquad2***的LSQ相加，在这个例子中，提供1的进位，并将结果放在rax中。然后，rdx通过adc指令加上***dquad2***的MSQ以及进位，并将结果放在rdx中。

整数带进加法指令总结如下：

Instruction	Explanation
adc <dest>, <src> **adc <reg>, <reg>** **adc <reg>, <imm8/16/32>** **adc <reg>, <mem>** **adc <mem>, <reg>** **adc <mem>, <imm8/16/32>**	Add two operands, (**<dest>** + **<src>**) and any previous carry (stored in the carry bit in the **rFlag** register) and place the result in **<dest>** (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate. *Note 3*, 64-bit immediate values are not allowed.
Examples:	**adc rcx, qword [dVvar1]** **adc rax, 42**

更多　完整指令列表位于Appen　　十个

B.

7.5.2 减法

整数减法指令的一般形式如下：

子 **<**目标**>**，**<**源**>**

操作执行以下操作：

<目标**> = <**目标**> - <**源**>**

具体来说，源操作数从目的操作数中减去，并将结果放置在目的操作数中（覆盖之前的值）。源操作数的值保持不变。目的操作数和源操作数必须具有相同的大小（都是字节、都是字等）。目的操作数不能是立即数。两个操作数都不能是内存。如果需要执行内存到内存的减法操作，必须使用两条指令。

例如，假设以下数据声明：

```
bNum1       db      73
bNum2       db      42
bAns        db      0

wNum1       dw      1234
wNum2       dw      4321
wAns        dw      0
```

```
dNum1       dd      73000
dNum2       dd      42000
dAns        dd      0

qNum1       dq      73000000
qNum2       dq      73000000
qAns        dd      0
```

执行以下基本操作:

```
bAns = bNum1 - bNum2 wA
ns = wNum1 - wNum2 dAns
= dNum1 - dNum2 qAns = q
Num1 - qNum2
```

以下说明可用于:

```
; bAns = bNum1 - bNum2 mov al, by
te [bNum1] sub al, byte [bNum2] mo
v byte [bAns], al ; wAns = wNum1 −
wNum2 mov ax, word [wNum1] sub
ax, word [wNum2] mov word [wAns]
, ax ; dAns = dNum1 − dNum2 mov e
ax, dword [dNum1] sub eax, dword [
dNum2] mov dword [dAns], eax ; qA
ns = qNum1 - qNum2 mov rax, qwor
d [qNum1] sub rax, qword [qNum2]
mov qword [qAns], rax
```

对于某些指令，包括上述指令，可以省略显式类型指定（例如，*byte*，*word*，*dword*，*qword*），因为其他操作数将明确定义大小。这包括在内是为了保持一致性并遵循良好的编程实践。

除了基本的减法指令外，还有一个减一指令，该指令将从指定的操作数中减去一个值。减一指令的一般形式为

如下所示:

十二 **<**操作数**>**
操作执行以下操作:

<operand> = <operand> - 1

结果与使用减法指令（并减去一个）完全相同。当使用内存操作数时，需要显式类型指定（例如，*byte*，*word*，*dword*，*qword*）以清楚地定义大小。

例如，假设以下数据声明:

```
bNum        db      42
wNum        dw      4321
dNum        dd      42000
qNum        dq      42000000
```

执行以下基本操作:

rax **=** rax - 1 bNum **=**
bNum - 1 wNum **=** w
Num - 1 dNum **=** dNu
m - 1 qNum **=** qNum -
1

以下说明可用于:

dec rax ; rax **=** rax - 1 dec byte [bNum] ; bNum **=** bNum - 1 dec wor
d [wNum] ; wNum **=** wNum - 1 dec dword [dNum] ; dNum **=** dNum
- 1 dec qword [qNum] ; qNum **=** qNum - 1

减法指令在有符号和无符号数据上操作相同。确保数据类型和大小适合所执行的操作是程序员的职责。

The in特格减法指令总结如下　　低点:

Instruction	**Explanation**
sub <dest>, <src> **sub <reg>, <reg>** **sub <reg>, <imm8/16/32>** **sub <reg>, <mem>**	Subtract two operands, (<**dest**> - <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory.

Instruction	Explanation
sub <mem>, <reg> **sub <mem>, <imm8/16/32>**	*Note 2*, destination operand cannot be an immediate. *Note 3*, 64-bit immediate values are not allowed.
Examples:	**sub cx, word [wVvar]** **sub rax, 42** **sub dword [dVar], eax** **sub qword [qVar], 300**
dec <operand>	Decrement <**operand**> by 1. *Note*, <**operand**> cannot be an immediate.
Examples:	**dec word [wVvar]** **dec rax** **dec dword [dVar]** **dec qword [qVar]**

更完整的指令列表位于附录B中。

7.5.3 整数乘法

乘法指令用于乘以两个整数操作数。在数学上，处理有符号值乘法有特殊规则。因此，对于无符号乘法（mul）和有符号乘法（imul）使用不同的指令。

乘法通常会产生双倍大小的结果。也就是说，乘以两个 ***n***-位值会产生一个 2***n***-位结果。乘以两个 8 位数字将产生一个 16 位结果。同样，两个 16 位数字的乘积将产生一个 32 位结果，两个 32 位数字的乘积将产生一个 64 位结果，两个 64 位数字的乘积将产生一个 128 位结果。

有许多乘法指令的变体。对于有符号乘法，某些形式会将结果截断到原始操作数的大小。确保使用的值适用于所选特定指令是程序员的职责。

7.5.3.1 无符号乘法

无符号乘法的一般形式如下:

乘 {v*}

源操作数必须是寄存器或内存位置。不允许立即操作数。

对于单操作数乘法指令，必须使用A寄存器（al/ax/eax/rax）作为其中一个操作数（al用于8位，ax用于16位，eax用于32位，rax用于64位）。另一个操作数可以是内存位置或寄存器，但不能是立即数。此外，结果将放置在A寄存器和可能D寄存器中，具体取决于乘法的大小。以下表格显示了字节、字、双字和四字无符号乘法的各种选项。

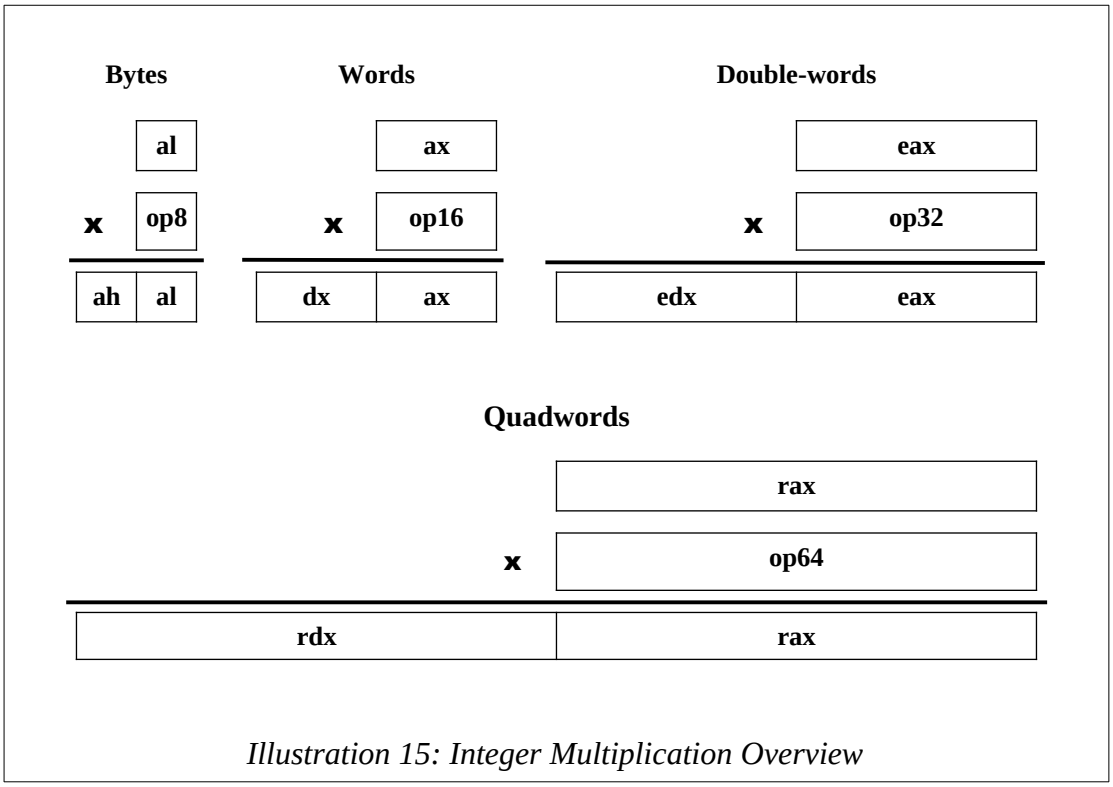


Illustration 15: Integer Multiplication Overview

如图所示，在大多数情况下，整数乘法使用A和D寄存器的组合。这可能会非常令人困惑。

例如，当将rax（64位）乘以一个quadword操作数（64位）时，乘法指令提供双quadword结果（128位）。这在处理非常大的数字时非常有用且重要。由于64位架构只有64位寄存器，因此128位结果必须放在两个不同的quadword（64位）寄存器中，rdx用于高阶结果，rax用于低阶结果，通常写作rdx:rax（按惯例）。

然而，这种使用两个寄存器的做法也适用于较小的尺寸。例如，将ax（16位）乘以一个字操作数（也是16位）的结果是一个双字（32位）的结果。然而，结果不是放在eax（可能更容易）中，而是放在两个寄存器中，dx用于高阶结果（16位），ax用于低阶结果（16位），通常写作dx:ax（按惯例）。由于双字（32位）结果在两个不同的寄存器中，可能需要两个移动来保存结果。

此寄存器配对，即使不是必需的，也是由于对先前早期版本的架构的遗留支持。虽然这有助于确保向后兼容性，但它可能会相当令人困惑。

例如，假设以下数据声明：

```
bNumA       db      42
bNumB       db      73
wAns        dw      0
wAns1       dw      0

wNumA       dw      4321
wNumB       dw      1234
dAns2       dd      0

dNumA       dd      42000
dNumB       dd      73000
qAns3       dq      0

qNumA       dq      420000
qNumB       dq      730000
dqAns4      ddq     0
```

执行以下基本操作：

```
wAns = bNumA2                   ; bNumA平方
bAns1 = bNumA * bNumB w
Ans1 = bNumA * bNumB
```

wAns2 = wNumA * wNumB dAns2 = wNumA * wNumB dAns3 = d NumA * dNumB qAns3 = dNumA * dNumB qAns4 = qNumA * qNumB d qAns4 = qNumA * qNumB 以下指令可能被使用：; wAns = bNumA2 或 bNumA 平方 mov al, byte [bNumA] mul al ; 结果在 ax 中 mov word [wA ns], ax ; wAns1 = bNumA * bNumB mov al, byte [bNumA] mul byte [bNu mB] ; 结果在 ax 中 mov word [wAns1], ax ; dAns2 = wNumA * wNumB mov ax, word [wNumA] mul word [wNumB] ; 结果在 dx:ax 中 mov word [dAns2], ax mov word [dAns2+2], dx ; qAns3 = dNumA * dNumB mov ea x, dword [dNumA] mul dword [dNumB] ; 结果在 edx:eax 中 mov dword [qAns3], eax mov dword [qAns3+4], edx ; dqAns4 = qNumA * qNumB mov rax, qword [qNumA] mul qword [qNumB] ; 结果在 rdx:rax 中 mov qword [dqAns4], rax mov qword [dqAns4+8], rdx

对于某些指令，包括上述指令，需要显式类型说明（例如，*byte*，*word*，*dword*，*qword*）以明确定义大小。

整数无符号乘法指令总结如下:

Instruction	Explanation
mul <src> **mul <op8>** **mul <op16>** **mul <op32>** **mul <op64>**	Multiply **A** register (**al**, **ax**, **eax**, or **rax**) times the <**src**> operand. Byte: **ax** = **al** * <**src**> Word: **dx:ax** = **ax** * <**src**> Double: **edx:eax** = **eax** * <**src**> Quad: **rdx:rax** = **rax** * <**src**> *Note,* <**src**> operand cannot be an immediate.
Examples:	**mul word [wVvar]** **mul al** **mul dword [dVar]** **mul qword [qVar]**

更完整的指令列表位于附录B中。

7.5.3.2 签名乘法

签名乘法允许更广泛的操作数和操作数大小。签名乘法的一般形式如下:

imul **<**source**>** imul **<**dest**>**, **<**src/imm**>**
imul **<**dest**>**, **<**src**>**, **<**imm**>**

在所有情况下，目标操作数必须是一个寄存器。对于多操作数乘法指令，不支持字节操作数。

当使用单操作数乘法指令时，imul 与 mul（如前所述）具有相同的布局。然而，操作数仅解释为有符号的。

当使用两个操作数时，目标操作数和源操作数相乘，结果放置在目标操作数中（覆盖之前的值）。

具体来说，执行的动作是:

<dest> = <dest> * <src/imm>

对于两个操作数，<src/imm>操作数可以是寄存器、内存位置或立即值。<src/imm>值的尺寸限制在源操作数的尺寸内，最多为双字大小（32位），即使对于四字（64位）乘法也是如此。

最终结果被截断到目标操作数的大小。不支持字大小的目标操作数。

当使用三个操作数时，两个操作数相乘，结果放入目标操作数中。具体操作如下：

<目标**> = <**源**> * <**立即**>**

对于三个操作数，<src>操作数必须是寄存器或内存位置，但不能是立即数。<imm>操作数必须是立即数。立即数的大小限制在源操作数的大小内，最多为双字大小（32位），即使对于四字乘法也是如此。最终结果被截断到目标操作数的大小。不支持字大小的目标操作数。

应注意的是，当乘法指令提供更大的类型时，可以使用原始类型。为了使其工作，乘法中的值必须适合较小的尺寸，这限制了数据的范围。例如，当两个双字相乘并提供四字结果时，如果值足够小（这通常是情况），则最低有效双字（四字中的）将包含答案。这通常在高级语言中完成，当一个 int（32 位整数）变量乘以另一个 int 变量并将其赋值给 int 变量时。

例如，假设以下数据声明：

```
wNumA       dw      1200
wNumB       dw      -2000
wAns1       dw      0
wAns2       dw      0

dNumA       dd      42000
dNumB       dd      -13000
dAns1       dd      0
dAns2       dd      0

qNumA       dq      120000
qNumB       dq      -230000
qAns1       dq      0
qAns2       dq      0
```

执行以下基本操作：

```
wAns1 = wNumA * -13 wAns
2 = wNumA * wNumB
```

```
dAns1 = dNumA * 113 dAns2
= dNumA * dNumB qAns1 =
qNumA * 7096 qAns2 = qNum
A * qNumB
```

以下说明可用于:

```
; wAns1 = wNumA * -13 mov ax,
word [wNumA] imul ax, -13        翻译
文本：; wAns1 = wNumA * -13                    ; 结果为 ax
mov ax, word [wNumA] imul ax, -
mov word [wAns1], ax ; wAns2 =
13
wNumA * wNumB mov ax, word
[wNumA] imul ax, word [wNumB
]                                              ; 结果为 ax
mov word [wAns2], ax ; dAns1 = dN
umA * 113 mov eax, dword [dNumA
] imul eax, 113
                                               ; 结果在eax
mov dword [dAns1], eax ; dAns2 = d
NumA * dNumB mov eax, dword [d
NumA] imul eax, dword [dNumB]
                                               ; 结果在eax
mov dword [dAns2], eax ; qAns1 = q
NumA * 7096 mov rax, qword [qNu
mA] imul rax, 7096
                                               ; 结果在rax
mov qword [qAns1], rax ; qAns2 = q
NumA * qNumB mov rax, qword [q
NumA] imul rax, qword [qNumB]
                                               ; 结果在rax
mov qword [qAns2], rax       移动word [qAns2],   rax
```

另一种执行 qAns1 = qNumA * 7096 乘法的方法

将如下所示:

```
; qAns1 = qNumA * 7096 mov rcx, q
word [qNumA] imul rbx, rcx, 7096
                                        ; 结果在rbx
mov qword [qAns1], rbx
```

此示例展示了使用不同寄存器的三操作数乘法指令。

在这些示例中，乘法结果被截断到目标操作数的大小。对于完整大小的结果，应使用单操作数指令（如无符号乘法章节中所述）。

对于某些指令，包括上述指令，可能不需要显式类型指定（例如，*byte*，*word*，*dword*，*qword*）来明确定义大小。

整数有符号乘法指令总结如下:

Instruction	**Explanation**
imul <src> **imul <dest>, <src/imm32>** **imul <dest>, <src>, <imm32>** **imul <op8>** **imul <op16>** **imul <op32>** **imul <op64>** **imul <reg16>,** **<op16/imm8/16/32>** **imul <reg32>,** **<op32/imm8/16/32>** **imul <reg64>,** **<op64/imm8/16/32>** **imul <reg16>, <op16>,** **<imm8/16/32>** **imul <reg32>, <op32>,** **<imm8/16/32>** **imul <reg64>, <op64>,** **<imm8/16/32>**	Signed multiply instruction. For single operand: Byte: **ax** = **al** * **<src>** Word: **dx:ax** = **ax** * **<src>** Double: **edx:eax** = **eax** * **<src>** Quad: **rdx:rax** = **rax** * **<src>** *Note*, **<src>** operand cannot be an immediate. For two operands: **<reg16> = <reg16> * <op16/imm>** **<reg32> = <reg32> * <op32/imm>** **<reg64> = <reg64> * <op64/imm>** For three operands: **<reg16> = <op16> * <imm>** **<reg32> = <op32> * <imm>** **<reg64> = <op64> * <imm>** *Note 1*, 64-bit immediate values are not allowed.
Examples:	**imul ax, 17** **imul al** **imul ebx, dword [dVar]**

Instruction	Explanation

	`imul  rbx, dword [dVar], 791`
	`imul  rcx, qword [qVar]`
	`imul  qword [qVar]`

更完整的指令列表位于附录B中。

7.5.4 整数除法

除法指令将两个整数操作数相除。在数学上，处理有符号值除法有特殊的规则。因此，对于无符号除法（div）和有符号除法（idiv）使用不同的指令。

回忆起 $\frac{dividend}{divisor} = quotient$

除法要求被除数必须比除数大。为了用8位除数进行除法，被除数必须是16位（即更大的大小）。同样，16位除数需要32位被除数。而且，32位除数需要64位被除数。

与乘法类似，在大多数情况下，整数除法使用A和D寄存器的组合。这种寄存器配对是由于对先前早期版本的架构的遗留支持。虽然这有助于确保向后兼容性，但它可能会相当令人困惑。

此外，A寄存器和可能D寄存器必须组合使用以进行被除数。

- 字节除法：ax 为 16 位
- 单词除法：dx:ax for 32位
- 双词除法：edx:eax 用于 64 位
- 四字除法：rdx:rax 用于 128 位

设置除数（最高操作数）正确是问题的关键来源。对于单字、双字和四字除法操作，除数需要D寄存器（用于高阶部分）和A（用于低阶部分）。

设置这些值取决于数据类型。如果之前已执行了乘法，D和A寄存器可能已经正确设置。否则，可能需要将数据项从当前大小转换为具有更高阶的大大小。

部分放入D寄存器。对于无符号数据，高位始终为零。对于有符号数据，现有数据必须按照前述章节中所述进行符号扩展，*Signed Conversions*。

除数可以是内存位置或寄存器，但不能是立即数。此外，结果将放置在A寄存器（al/ax/eax/rax）中，余数将放置在ah、dx、edx或rdx寄存器中。请参考*Integer Division Overview*表以更清楚地了解布局。

使用较大的被除数操作数与单操作数乘法相匹配。对于简单的除法，可能需要适当的转换以确保被除数设置正确。对于无符号除法，被除数的最高阶部分可以设置为零。对于有符号除法，被除数的最高阶部分可以使用相应的转换指令设置。

始终如一，除以零将使程序崩溃并损害时空连续体。因此，请尽量避免除以零。

以下表格概述了字节、字、双字和四字除法指令。

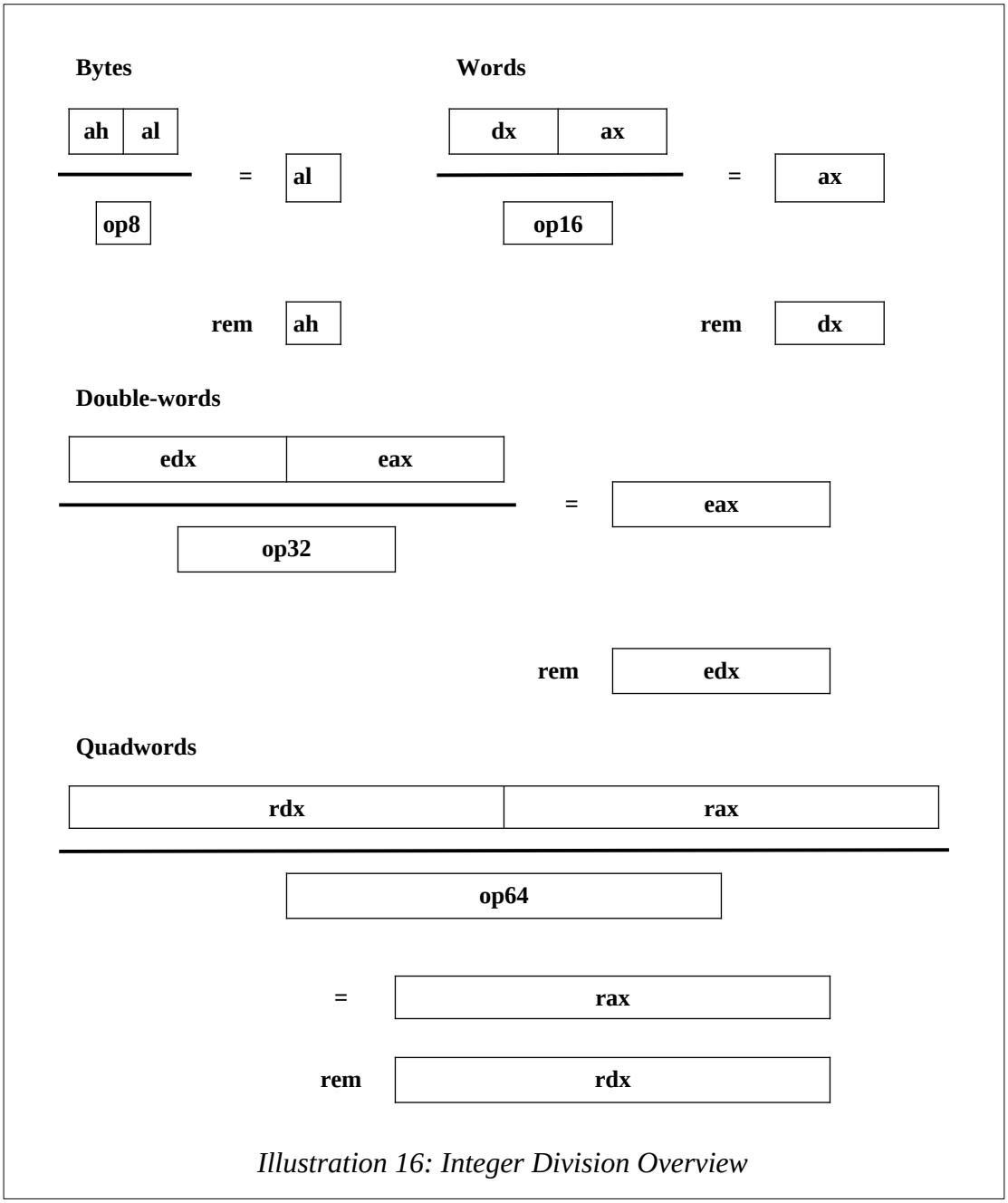


Illustration 16: Integer Division Overview

已签名和无符号除法指令以相同的方式操作。然而，可以除以的值范围不同。程序员负责确保正在除以的值适用于所使用的操作数大小。

无符号和有符号除法的一般形式如下:

div **<**src**>** ; 无符号除法 idiv **<**src**>** ; 有符号除法

源操作数和目标操作数（A和D寄存器）在前面表中进行了描述。

例如，假设以下数据声明:

```
bNumA       db      63
bNumB       db      17
bNumC       db      5
bAns1       db      0
bAns2       db      0
bRem2       db      0
bAns3       db      0

wNumA       dw      4321
wNumB       dw      1234
wNumC       dw      167
wAns1       dw      0
wAns2       dw      0
wRem2       dw      0
wAns3       dw      0

dNumA       dd      42000
dNumB       dd      -3157
dNumC       dd      -293
dAns1       dd      0
dAns2       dd      0
dRem2       dd      0
dAns3       dd      0

qNumA       dq      730000
qNumB       dq      -13456
qNumC       dq      -1279
qAns1       dq      0
qAns2       dq      0
qRem2       dq      0
qAns3       dq      0
```

执行以下基本操作:

```
bAns1 = bNumA / 3                       无符号
bAns2 = bNumA / bNumB                   无符号
bRem2 = bNumA % bNumB                   ; % 是模数
bAns3 = (bNumA * bNumC) / bNumB         无符号

wAns1 = wNumA / 5                       无符号
wAns2 = wNumA / wNumB                   无符号
wRem2 = wNumA % wNumB                   ; % 是模数
wAns3 = (wNumA * wNumC) / wNumB         无符号

dAns = dNumA / 7                        ; 已签
dAns3 = dNumA * dNumB                   ; 已签
dRem1 = dNumA % dNumB                   ; % 是模数
dAns3 = (dNumA * dNumC) / dNumB         ; 已签

qAns = qNumA / 9                        ; 已签
qAns4 = qNumA * qNumB                   ; 已签
qRem1 = qNumA % qNumB                   ; % 是模数
qAns3 = (qNumA * qNumC) / qNumB         ; 已签
```

以下说明可用于:

```
; ----- ; 示例字节操作，无符号


; bAns1 = bNumA / 3 (无符号) mov al, byte
[bNumA] mov ah, 0 mov bl, 3 div bl


                                        ; al = ax / 3
mov byte [bAns1], al

; bAns2 = bNumA / bNumB (无符号) mov ax, 0 mov al, byte [bNumA] div byt
e [bNumB] ; al = ax / bNumB mov byte [bAns2], al mov byte [bRem2], ah


                                        ; ah = ax % bNumB

; bAns3 = (bNumA * bNumC) / bNumB (无符号) mov al, byte [bNumA]
mul byte [bNumC] ; 结果在ax中
```

div byte [bNumB]; al = ax / bNumB mov byte [bAns3], al

; ----- ; 示例单词操作，无符号

; wAns1 = wNumA / 5 (无符号) mov ax, wo
rd [wNumA] mov dx, 0 mov bx, 5 div bx

; ax = dx:ax / 5
mov word [wAns1], ax

; wAns2 = wNumA / wNumB (无符号) mov dx, 0 mov ax, word [wNumA] div wor
d [wNumB] ; ax = dx:ax / wNumB mov word [wAns2], ax mov word [wRem2], dx

; wAns3 = (wNumA * wNumC) / wNumB (无符号) mov ax, wor
d [wNumA] mul word [wNumC]
; 结果为 dx:ax
div 单词 [wNumB] ; ax = dx:ax / wNumB
mov word [wAns3], ax

; ----- ; 示例双词操作，有符号

; dAns1 = dNumA / 7 (有符号) mov eax,
dword [dNumA] cdq
; eax → edx:eax
mov ebx, 7 idiv ebx
; eax = edx:eax / 7
mov dword [dAns1], eax 移动双字 [dAns1], eax

; dAns2 = dNumA / dNumB (有符号) mov eax, dword [dNumA] cdq ; eax → edx:eax
idiv dword [dNumB] ; eax = edx:eax/dNumB mov dword [dAns2], eax mov dword [d
Rem2], edx ; edx = edx:eax%dNumB

```
; dAns3 = (dNumA * dNumC) / dNumB (signed)
mov     eax, dword [dNumA]
imul    dword [dNumC]                   ; result in edx:eax
idiv    dword [dNumB]                   ; eax = edx:eax/dNumB
mov     dword [dAns3], eax

; -----
;  example quadword operations, signed

; qAns1 = qNumA / 9 (signed)
mov     rax, qword [qNumA]
cqo                                     ; rax → rdx:rax
mov     rbx, 9
idiv    rbx                             ; eax = edx:eax / 9
mov     qword [qAns1], rax

; qAns2 = qNumA / qNumB (signed)
mov     rax, qword [qNumA]
cqo                                     ; rax → rdx:rax
idiv    qword [qNumB]                   ; rax = rdx:rax/qNumB
mov     qword [qAns2], rax
mov     qword [qRem2], rdx              ; rdx = rdx:rax%qNumB

; qAns3 = (qNumA * qNumC) / qNumB (signed)
mov     rax, qword [qNumA]
imul    qword [qNumC]                   ; result in rdx:rax
idiv    qword [qNumB]                   ; rax = rdx:rax/qNumB
mov     qword [qAns3], rax
```

对于某些指令，包括上述指令，需要显式类型说明（例如，*byte*，*word*，*dword*，*qword*）以明确定义大小。

整数除法指令总结如下:

Instruction	Explanation
div <src> **div <op8>** **div <op16>** **div <op32>** **div <op64>**	Unsigned divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand. Byte: **al** = **ax** / <**src**>, rem in **ah** Word: **ax** = **dx:ax** / <**src**>, rem in **dx** Double: **eax** = **edx:eax** / <**src**>, rem in **edx** Quad: **rax** = **edx:rax** / <**src**>, rem in **rdx** *Note*, <**src**> operand cannot be an immediate.
Examples:	**div word [wVvar]** **div bl** **div dword [dVar]** **div qword [qVar]**
idiv <src> **idiv <op8>** **idiv <op16>** **idiv <op32>** **idiv <op64>**	Signed divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand. Byte: **al** = **ax** / <**src**>, rem in **ah** Word: **ax** = **dx:ax** / <**src**>, rem in **dx** Double: **eax** = **eax** / <**src**>, rem in **edx** Quad: **rax** = **rax** / <**src**>, rem in **rdx** *Note*, <**src**> operand cannot be an immediate.
Examples:	**idiv word [wVvar]** **idiv bl** **idiv dword [dVar]** **idiv qword [qVar]**

更完整的指令列表位于附录B中。

7.6 逻辑指令

本节总结了编程时可能有用的一些更常见的逻辑指令。

7.6.1 逻辑运算

如您应记得，以下是基本逻辑运算的真值表；

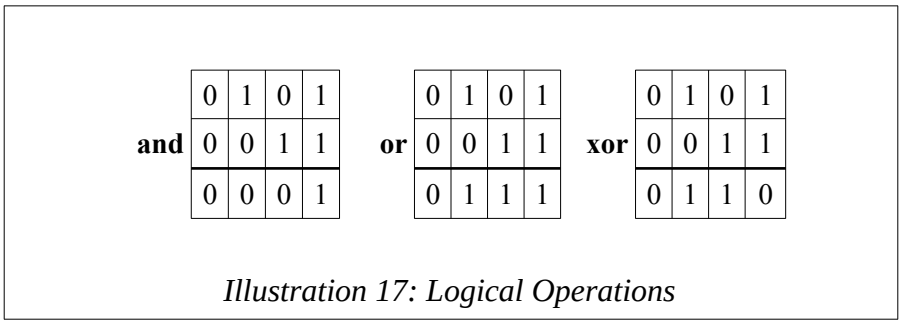


Illustration 17: Logical Operations

T逻辑指令总结如下：

Instruction	**Explanation**

`and <dest>, <src>`
 	
`and <reg>, <reg>`
`and <reg>, <imm8/16/32>`
`and <reg>, <mem>`
`and <mem>, <reg>`
`and <mem>, <imm8/16/32>`
	Perform logical AND operation on two operands, (<**dest**> and <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

Examples:	`and ax, bx`
	`and rcx, rdx`
	`and eax, dword [dNum]`
	`and qword [qNum], rdx`

`or <dest>, <src>`
 	
`or <reg>, <reg>`
`or <reg>, <imm8/16/32>`
`or <reg>, <mem8/16/32>`
`or <mem>, <reg>`
`or <mem>, <imm8/16/32>`
	Perform logical OR operation on two operands, (<**dest**> || <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

Instruction	Explanation

Examples:	`or  ax, bx`
	`or  rcx, rdx`
	`or  eax, dword [dNum]`
	`or  qword [qNum], rdx`

`xor  <dest>, <src>`
 	
`xor  <reg>, <reg>`
`xor  <reg>, <imm8/16/32>`
`xor  <reg>, <mem>`
`xor  <mem>, <reg>`
`xor  <mem>, <imm8/16/32>`
	Perform logical XOR operation on two operands, (<**dest**> ^ <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

Examples:	`xor  ax, bx`
	`xor  rcx, rdx`
	`xor  eax, dword [dNum]`
	`xor  qword [qNum], rdx`

`not  <op>`
 	
`not  <reg>`
`not  <mem8/16/32>`
	Perform a logical not operation (one's complement on the operand 1's→0's and 0's→1's).
	Note 1, operand cannot be an immediate.
	Note 2, 64-bit immediate values are not allowed.

Examples:	`not  bx`
	`not  rdx`
	`not  dword [dNum]`
	`not  qword [qNum]`

The & refers to the logical AND operation, the || refers to the logical OR operation, and t he ^refers to the logical XOR operation as per C/C++ conventions. The ¬ refers to the log ical NOT operation.

更完整的指令列表位于附录B中。

7.6.2 位移操作

位移操作将操作数内的位进行左移或右移。位移动的典型原因包括为了某些特定目的将操作数内的位子集隔离出来，或者可能用于执行乘以或除以2的幂次运算。所有位都移动一个位置。移出操作数的位将丢失，并在另一侧添加一个0位。

7.6.2.1 逻辑移位

逻辑移位是一种位操作，它将源寄存器中的所有位按指定的位数进行移位，并将结果放入目标寄存器。位可以按需左移或右移。源操作数中的每个位都移动指定的位数，新空出的位位置用零填充。

以下图示展示了对于字节大小的操作数，右移和左移操作是如何工作的。

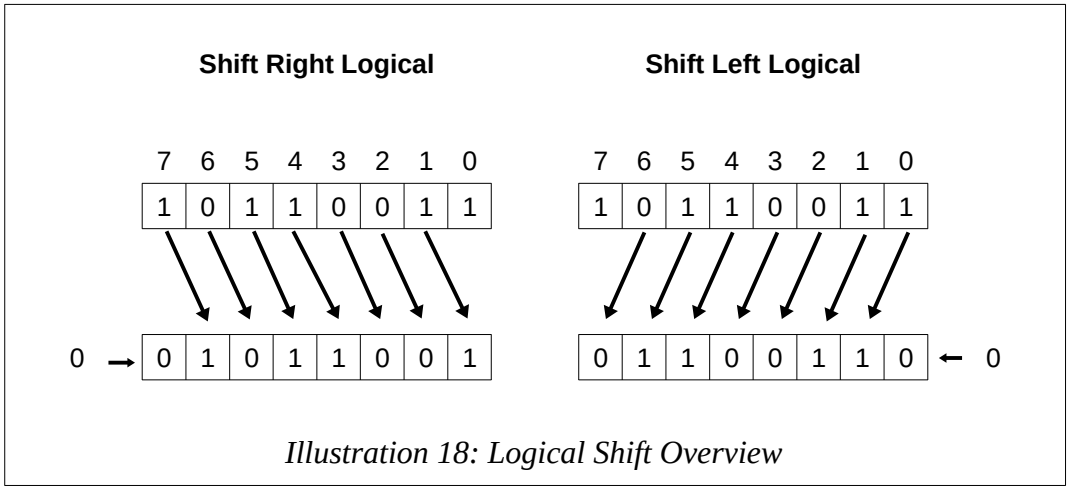


Illustration 18: Logical Shift Overview

逻辑移位将操作数视为一系列比特，而不是一个数字。

位移指令可用于执行2的幂的无符号整数乘法和除法操作。2的幂包括2、4、8等，直到操作数大小的限制（对于寄存器操作数为32位）。

在下面的示例中，23通过执行逻辑右移一位被除以2。得到的结果11以二进制形式显示。接下来，13通过执行逻辑左移两位被乘以4。得到的结果52以二进制形式显示。

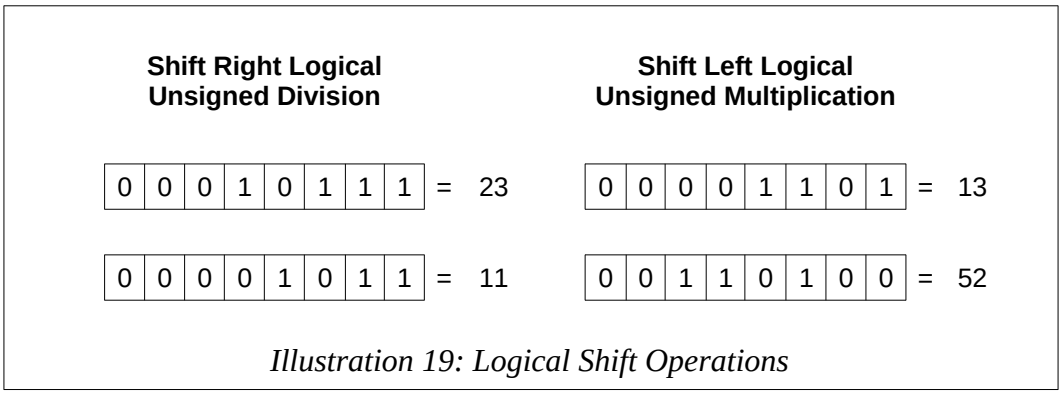


Illustration 19: Logical Shift Operations

在示例中可以看出，在新的空位上（取决于操作，在右侧或左侧）输入了一个0。

逻辑位移指令总结如下:

Instruction	Explanation
shl <dest>, <imm8> **shl <dest>, cl**	Perform logical shift left operation on destination operand. Zero fills from right (as needed). The **<imm>** or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.
Examples:	**shl ax, 8** **shl rcx, 32** **shl eax, cl** **shl qword [qNum], cl**
shr <dest>, <imm8> **shr <dest>, cl**	Perform logical shift right operation on destination operand. Zero fills from left (as needed). The **<imm>** or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

Instruction	Explanation

Examples:	`shr ax, 8`
	`shr rcx, 32`
	`shr eax, cl`
	`shr qword [qNum], cl`

更完整的指令列表位于附录B中。

7.6.2.2 算术移位

算术右移也是一种位操作，它将源寄存器中的所有位按指定的位数右移，并将结果放入目标寄存器。源操作数中的每一位都移动指定的位数，并填充新的空位。对于算术右移，原始最左边的位（符号位）被复制以填充所有空位。这被称为符号扩展。

以下图示展示了字节大小操作数进行左移和右移算术操作的工作原理。

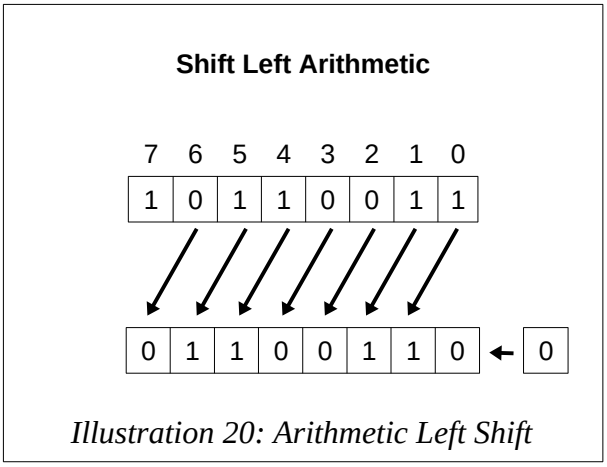


Illustration 20: Arithmetic Left Shift

算术左移将位向左移动指定的位数，并用零填充最低有效位位置（左）。最高符号位不被保留。算术左移可以用于高效地执行以2的幂为因子的乘法。如果结果值不合适，将产生溢出。

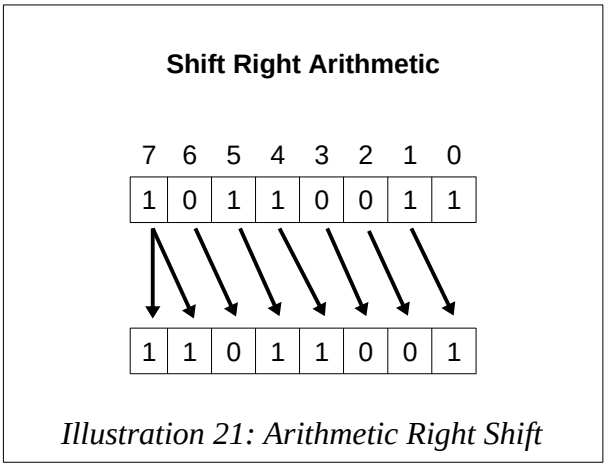


Illustration 21: Arithmetic Right Shift

算术右移将位移动到指定的位置，并将操作数视为有符号数，扩展符号（在这个例子中为负数）。

算术移位总是向下舍入（趋向负无穷）而标准除法指令截断（趋向0）。因此，通常不使用算术移位来替换有符号除法指令。

未提供删除这些指令总结如下 低点:

Instruction	**Explanation**
`sal <dest>, <imm8>` `sal <dest>, cl`	Perform arithmetic shift left operation on destination operand. Zero fills from right (as needed). The **<imm>** or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

Examples:	`sal ax, 8`
	`sal rcx, 32`
	`sal eax, cl`
	`sal qword [qNum], cl`

Instruction	Explanation
`sar <dest>, <imm8>` `sar <dest>, cl`	Perform arithmetic shift right operation on destination operand. Sign fills from left (as needed). The **<imm>** or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

Examples:	`sar ax, 8`
	`sar rcx, 32`
	`sar eax, cl`
	`sar qword [qNum], cl`

更完整的指令列表位于附录B中。

7.6.3 旋转操作

旋转操作将位操作数内的位左移或右移，被移出操作数的位旋转并放置在另一端。

例如，如果一个字节操作数，10010110_2，向右旋转1位，结果将是01001011_2。如果一个字节操作数，10010110_2，向左旋转1位，结果将是00101101_2。

左旋转和右旋转指令总结如下：

Instruction	Explanation
`rol <dest>, <imm8>` `rol <dest>, cl`	Perform rotate left operation on destination operand. The **<imm>** or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

Instruction	Explanation

Examples:	`rol  ax, 8`
	`rol  rcx, 32`
	`rol  eax, cl`
	`rol  qword [qNum], cl`

`ror  <dest>, <imm8>` `ror  <dest>, cl`	Perform rotate right operation on destination operand. The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

Examples:	`ror  ax, 8`
	`ror  rcx, 32`
	`ror  eax, cl`
	`ror  qword [qNum], cl`

更完整的指令列表位于附录B中。

7.7 控制指令

程序控制指的是基本编程结构，如IF语句和循环。

所有高级语言控制结构都必须使用有限的汇编语言控制结构来执行。例如，在汇编语言级别不存在IF-THEN-ELSE语句。汇编语言提供无条件分支（或跳转）和条件分支或IF语句，该语句将跳转到目标标签或不会跳转。

控制指令涉及无条件跳转和条件跳转。跳转对于基本条件语句（即IF语句）和循环是必需的。

7.7.1 标签

程序标签是控制语句的目标，或跳转到的位置。例如，循环的开始可能用“loopStart”这样的标签标记。代码可以通过跳转到标签来重新执行。

通常，标签以字母开头，后跟字母、数字或符号（限于“_”），以冒号（“:”）结尾。标签可以以非字母字符开头（例如，数字、“_”、“$”、“#”、“@”、“~”或“?”）。然而，这些通常具有特殊含义，通常程序员不应使用它们。yasm中的标签是区分大小写的。

例如，

```
循环开始：最
后：
```

有效标签。程序标签只能定义一次。

以下部分描述了标签的使用方式。

7.7.2 无条件控制指令

无条件指令提供对程序中用程序标签表示的特定位置的无条件跳转。目标标签必须恰好定义一次，并且从原始跳转指令可访问且在作用域内。

无条件跳转指令总结如下：

Instruction	**Explanation**
`jmp <label>`	Jump to specified label. *Note,* label must be defined exactly once.
Examples:	**`jmp startLoop`** **`jmp ifDone`** **`jmp last`**

更完整的指令列表位于附录B中。

7.7.3 条件控制指令

条件控制指令根据比较提供条件跳转。这提供了基本 IF 语句的功能。

两个步骤用于比较；比较指令和条件跳转指令。条件跳转指令将根据前一次比较操作的结果跳转到或不会跳转到提供的标签。比较指令将比较两个操作数并将比较结果存储在rFlag寄存器中。条件跳转指令将根据操作（跳转或不跳转）执行。

rFlag寄存器的内容。这要求比较指令立即后跟条件跳转指令。如果在比较和条件跳转之间放置其他指令，rFlag寄存器将被更改，条件跳转可能不会反映正确的条件。

比较指令的一般形式是:

```
cmp    <op1>, <op2>
```

<op1> 和 <op2> 不变，且必须具有相同的大小。两者中任意一个，但不是两者都，可以是内存操作数。<op1> 操作数不能是立即数，但 <op2> 操作数可以是立即数值。

条件控制指令包括跳转相等（je）和跳转不等（jne），它们对有符号和无符号数据都起作用。

已签名的条件控制指令包括比较操作的基本集合；跳转小于（jl），跳转小于等于（jle），跳转大于（jg），以及跳转大于等于（jge）。

未签名的条件控制指令包括比较操作的基本集合；跳转到小于（jb），跳转到小于或等于（jbe），跳转到大于（ja），以及跳转到大于或等于（jae）。

签名条件指令的一般形式及其解释注释如下:

```
je    <label>          ; if <op1> == <op2>
jne   <label>          ; if <op1> != <op2>

jl    <label>          ; signed, if <op1> < <op2>
jle   <label>          ; signed, if <op1> <= <op2>
jg    <label>          ; signed, if <op1> > <op2>
jge   <label>          ; signed; if <op1> >= <op2>

jb    <label>          ; unsigned, if <op1> < <op2>
jbe   <label>          ; unsigned, if <op1> <= <op2>
ja    <label>          ; unsigned, if <op1> > <op2>
jae   <label>          ; unsigned, if <op1> >= <op2>
```

例如，给定以下用于签名数据的伪代码:

```
if (currNum > myMax) myMax
= currNum;
```

假设以下数据声明:

```
currNum     dq      0
myMax       dq      0
```

假设程序内值更新适当（未显示），以下指令可以使用:

```
    mov rax, qword [currNum] cmp rax, qw
    ord [myMax]                                 ; 如果 currNum <= myMax
    jle notNewMax                               ;  跳过设置新最大值
    mov qword [myMax], rax
notNewMax:
```

注意，IF语句的逻辑已被反转。比较和条件跳转提供跳转或不跳转的功能。因此，如果原始IF语句的条件为假，则代码必须不执行。因此，当条件为假时，为了跳过执行，条件跳转将立即跳转到要跳过的代码（不执行）之后的标记。虽然这个例子中只有一行，但可能有许多行代码。

一个更复杂的例子可能如下所示:

```
if (x != 0) { ans = x / y; errFlg
= FALSE; } else { ans = 0; err
Flg = TRUE; }
```

此基本比较和条件跳转不提供典型的 IF-ELSE 结构。它必须创建。假设 ***x*** 和 ***y*** 变量是将在程序执行期间设置的带符号双字，并且以下声明:

```
TRUE equ 1 FALSE equ 0 x dd 0
y dd 0 ans dd 0 errFlg db FALSE
```

以下代码可用于实现上述 IF-ELSE 语句。

```
    比较dword [x] 与 0                                  ; 如果语句
    je doElse mov eax, dword [x] cdq idiv d
word [y] mov dword [ans], eax mov byte [err
Flg], FALSE jmp skpElse doElse: mov dword
[ans], 0 mov byte [errFlg], TRUE skpElse:
```

在这个例子中，由于数据已签名，需要使用有符号除法（idiv）和适当的转换（在这种情况下为cdq）。还应注意的是，即使没有明确出现，edx寄存器也被覆盖了。如果之前已将值放入edx（或rdx），它已被更改。

7.7.3.1 超出范围跳出

目标标签被称为短跳转。具体来说，这意味着目标标签必须位于条件跳转指令的 ±128 字节范围内。虽然这个限制通常不会成问题，但对于非常大的循环，汇编器可能会生成一个关于“跳转超出范围”的错误。无条件跳转（jmp）在范围上没有限制。如果生成了一个“跳转超出范围”，可以通过反转逻辑并使用无条件跳转来实现长跳转来消除它。例如，以下代码：

```
cmp rcx, 0 jne startOfLoop
比较rcx与0，如果不等于
则跳转到循环开始处
```

可能会在标签 ***startOfLoop*** 距离较远时生成“跳转出范围”的汇编器错误。可以通过以下代码消除错误：

```
cmp rcx, 0 je endOfLoop j
mp startOfLoop

endOfLoop:
```

使用无条件跳转进行长跳转，并添加一个跳转到非常接近标签的条件跳转，从而实现相同的功能。

条件跳转指令总结如下:

Instruction	**Explanation**
cmp <op1>, <op2>	Compare <**op1**> with <**op2**>. Results are stored in the **rFlag** register. *Note 1*, operands are not changed. *Note 2*, both operands cannot be memory. *Note 3*, <**op1**> operand cannot be an immediate. *Note 4*, <**op2**> can not be a 64-bit immediate value.
Examples:	**cmp rax, 5** **cmp ecx, edx** **cmp ax, word [wNum]**
je <label>	Based on preceding comparison instruction, jump to <**label**> if <**op1**> == <**op2**>. Label must be defined exactly once.
Examples:	**cmp rax, 5** **je wasEqual**
jne <label>	Based on preceding comparison instruction, jump to <**label**> if <**op1**> **!=** <**op2**>. Label must be defined exactly once.
Examples:	**cmp rax, 5** **jne wasNotEqual**
jl <label>	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> < <**op2**>. Label must be defined exactly once.
Examples:	**cmp rax, 5** **jl wasLess**
jle <label>	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> ≤ <**op2**>. Label must be defined exactly once.

Instruction	Explanation
Examples:	`cmp  rax, 5` `jle  wasLessOrEqual`
jg <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1>** > **<op2>**. Label must be defined exactly once.
Examples:	`cmp  rax, 5` `jg  wasGreater`
jge <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≥ **<op2>**. Label must be defined exactly once.
Examples:	`cmp  rax, 5` `jge  wasGreaterOrEqual`
jb <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** < **<op2>**. Label must be defined exactly once.
Examples:	`cmp  rax, 5` `jb  wasLess`
jbe <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≤ **<op2>**. Label must be defined exactly once.
Examples:	`cmp  rax, 5` `jbe  wasLessOrEqual`
ja <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** > **<op2>**. Label must be defined exactly once.

Instruction	Explanation
Examples:	`cmp rax, 5` `ja wasGreater`
`jae <label>`	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≥ **<op2>**. Label must be defined exactly once.
Examples:	`cmp rax, 5` `jae wasGreaterOrEqual`

更完整的指令列表位于附录B中。

7.7.4 迭代

基本控制指令概述提供了一种迭代或循环的方法。

一个基本的循环可以通过一个计数器实现，该计数器在循环的底部或顶部通过比较和条件跳转进行检查。

例如，假设以下声明：

```
lpCnt dq 15 sum dq 0
```

以下代码将计算从1到30的所有奇数之和：

```
    mov     rcx, qword [lpCnt]      ; loop counter
    mov     rax, 1                  ; odd integer counter
sumLoop:
    add     qword [sum], rax        ; sum current odd integer
    add     rax, 2                  ; set next odd integer
    dec     rcx                     ; decrement loop counter
    cmp     rcx, 0
    jne     sumLoop
```

这是完成奇数求和任务多种不同方法中的一种。在这个例子中，rcx用作循环计数器，rax用于当前的奇数整数（适当地初始化为1，并每次增加2）。

使用rcx作为计数器的显示过程在循环预定的次数时很有用。有一个特殊的指令，loop，提供循环支持。

一般格式如下：

循环 **<标签>**

该操作将执行rcx寄存器的递减、与0的比较，并在rcx ≠ 0的情况下跳转到指定的标签。该标签必须恰好定义一次。

因此，循环指令提供了与上一个示例程序中三行代码相同的功能。以下代码组是等效的：

Code Set 1	**Code Set 2**
`loop  <label>`	`dec  rcx`
	`cmp  rcx, 0`
	`jne  <label>`

例如，上一个程序可以写成以下形式：

```
    mov     rcx, qword [maxN]          ; loop counter
    mov     rax, 1                     ; odd integer counter
sumLoop:
    add     qword [sum], rax           ; sum current odd integer
    add     rax, 2                     ; set next odd integer
    loop    sumLoop
```

两个代码示例以相同的方式产生完全相同的结果。

由于rcx寄存器先递减然后检查，忘记设置rcx寄存器可能会导致循环未知次数。这很可能在循环执行期间产生错误，这在调试时可能会非常误导。

循环指令在编码时可能很有用，但它仅限于rcx寄存器和计数递减。如果需要嵌套循环，除非采取额外措施（即，根据需要保存/恢复rcx寄存器），否则在内外循环中都使用循环指令可能会导致冲突。

虽然本文本中的一些编程示例将使用循环指令，但这不是必需的。

循环指令总结如下:

Instruction	Explanation
`loop <label>`	Decrement **rcx** register and jump to **<label>** if **rcx** is ≠ 0. *Note*, label must be defined exactly once.

Examples:	`loop startLoop`
	`loop ifDone`
	`loop sumLoop`

更多 完整指令列表位于Appen 十个

B.

7.8 示例程序，平方和

以下是一个完整的示例程序，用于计算从1到 {v*} 的平方和。例如，10的平方和如下:

$$1^2 + 2^2 + \cdots + 10^2 = 385$$

此示例主程序将 ***n*** 值初始化为 10 以匹配上述示例。

; 简单示例程序，用于计算从 1 到 n 的平方和。; **********
**********; 数据声明

节数 .data

; ----- ; 定义常量

SUCCESS equ 0 ; 成功操作 SYS_exit equ 60 ; 终止调用代码

; 定义数据。

n dd 10 Σv² dq 0

; **Translated Text: ; ****************

章节 .text 全局 _start _
start:

; ----- ; 计算从1到n（包含）的平方和。; 方法：for (i=1; i<=n; i++) ;su
mOfSquares += i^2;mov rbx, 1 ; imov ecx, dword [n]sumLoop: mov rax, rb
x ; 获取imul rax ; i^2add qword [sumOfSquares], raxinc rbxloop sumLoop

; ----- ; 完成，终止程序。

last:mov rax, SYS_exit ; 退出系统调用代码 mov rdi, SUCCESS ; 成功退出 syscall

调试器可以用来检查结果并验证程序的正确执行。

7.9 练习

以下是本章的一些测验问题和基于此章节的建议项目。

7.9.1 测验问题

以下是本章的一些测验问题。

1) 以下哪条指令是合法的/非法的？如有必要，请提供解释。

1. mov rax, 542. mov ax, 543. mov al, 3544. mov rax, r115. mov rax, r11d6. mov 54, ecx7. mov rax, qword [qVar]8. mov rax, qword [bVar]9. mov rax, [qVar]10. mov rax, qVar11. mov eax, dword [bVar]12. mov qword [qVar2], qword [qVar1]

13. mov qword [bVar2], qword [qVar1] 14. mov r15, 54 15. mov r16, 54 16. mov r11b, 54

2) 解释以下每条指令的作用。

1. movzx rsi, byte [bVar1]
2. movsx rsi, byte [bVar1]

3) 什么指令用于：

1. 将al中的*unsigned*字节转换为ax中的字。2. 将al中的*signed*字节转换为ax中的字。

4) 用来做什么的指令是:

1. 将ax中的*unsigned*字转换为eax中的双字。2. 将ax中的*signed*字转换为eax中的双字。

5) 什么指令用于:

1. 将ax中的*unsigned*词转换为dx中的双词：ax .
2. 将ax中的*signed*词转换为dx中的双词：ax。

6) 解释cwd指令和movsx指令之间的区别。

7) 解释为什么在第一条指令中需要显式指定（本例中的*dword*）而在第二条指令中不需要。

1. 将dword [dVar] 加1
2. 添加 [dVar], eax

8) 给定以下代码片段:

```
mov rax, 9mov rbx,
2add rbx, rax
```

执行后，rax和rbx寄存器中会有什么内容？以十六进制形式显示，完整寄存器大小。

9) 给定以下代码片段:

```
mov rax, 9mov rbx,
2sub rax, rbx
```

执行后，rax和rbx寄存器中会有什么内容？以十六进制形式显示，完整寄存器大小。

10) 给定以下代码片段:

```
mov rax, 9mov rbx,
2sub rbx, rax
```

执行后，rax和rbx寄存器中会有什么内容？以十六进制形式显示，完整寄存器大小。

11) 给定以下代码片段：

```
mov rax, 4mov r
bx, 3imul rbx
```

执行后，rax和rdx寄存器中会有什么？以十六进制形式显示，完整寄存器大小。

12) 给定以下代码片段：

```
mov rax, 5 cqo
mov rbx, 3 idiv r
bx
```

执行后，rax和rdx寄存器中会有什么？以十六进制形式显示，完整寄存器大小。

13) 给定以下代码片段：

```
mov   rax, 11 cqo
mov   rbx, 4 idiv  r
bx
```

执行后，rax和rdx寄存器中会有什么？以十六进制形式显示，完整寄存器大小。

14) 解释为什么以下每个陈述都不会起作用。

1. mov 42, eax2. div 33. mov dword [num1], d
word [num2]

4. 将dword [ax]移动到800

15) 解释为什么以下代码片段将无法正确工作。

```
mov eax, 500mov e
bx, 10idiv ebx
```

16) 解释为什么以下代码片段将无法正确工作。

```
mov eax, -500 cdq m
ov ebx, 10 div ebx
```

17) 解释为什么以下代码片段将无法正确工作。

```
mov ax, -500 cwd mov bx, 10 i
div bx   mov dword [ans], eax
```

在什么情况下可以使用三操作数乘法?

7.9.2 建议项目

以下是本章的一些建议项目。

1) 编写一个程序来计算以下表达式，使用无符号字节变量和无符号操作。*Note*，按照惯例，对于这个问题，变量名的第一个字母表示大小（b → 字节和 w → 字词）以供清晰。{v*}

1. bAns1 = bNum1 + bNum22. b
Ans2 = bNum1 + bNum33. bAns3
= bNum3 + bNum44. bAns6 = bNu
m1 − bNum25. bAns7 = bNum1 −
bNum36. bAns8 = bNum2 − bNum
47. wAns11 = bNum1 * bNum3

8. wAns12 = bNum2 * bNum2
9. wAns13 = bNum2 * bNum4
10. bAns16 = bNum1 / bNum2
11. bAns17 = bNum3 / bNum4

12. bAns18 = wNum1 / bNum4
13. bRem18 = wNum1 % bNum4

使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

2) 使用有符号值和有符号操作重复之前的程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

3) 编写一个程序，使用无符号字大小变量完成以下表达式。*Note*，变量名的第一个字母表示大小（w → 字和 d → 双字）。

1. wAns1 = wNum1 + wNum22.
wAns2 = wNum1 + wNum33. wA
ns3 = wNum3 + wNum44. wAns6
= wNum1 − wNum25. wAns7 = w
Num1 − wNum36. wAns8 = wNu
m2 − wNum47. dAns11 = wNum1
* wNum3

8. dAns12 = wNum2 * wNum2
9. dAns13 = wNum2 * wNum4
10. wAns16 = wNum1 / wNum2
11. wAns17 = wNum3 / wNum4
12. wAns18 = dNum1 / wNum4
13. wRem18 = dNum1 % wNum4

使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

4) 使用有符号值和有符号操作重复之前的程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

5) 编写一个程序，使用无符号双字大小的变量完成以下表达式。*Note*，变量名的第一个字母表示大小（双字d → 和四字q →）。

1. dAns1 = dNum1 + dNum22. d
Ans2 = dNum1 + dNum33. dAns3
= dNum3 + dNum44. dAns6 = dN
um1 − dNum25. dAns7 = dNum1
− dNum36. dAns8 = dNum2 − dN
um47. qAns11 = dNum1 * dNum3

8. qAns12 = dNum2 * dNum2
9. qAns13 = dNum2 * dNum4
10. dAns16 = dNum1 / dNum2
11. dAns17 = dNum3 / dNum4
12. dAns18 = qNum1 / dNum4
13. dRem18 = qNum1 % dNum4

使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

6) 使用有符号值和有符号操作重复之前的程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

7) 实现计算从 1 到 ***n*** 的平方和的示例程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

8) 编写一个程序计算从1到 ***n*** 的和的平方。具体来说，计算从1到 ***n*** 的整数和，然后平方该值。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示十进制和十六进制的结果。

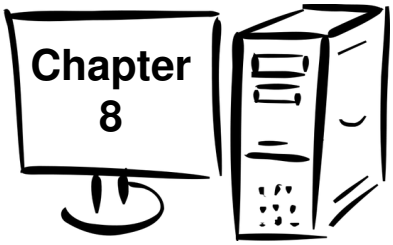
9) 编写一个程序，迭代找到第 ***n*** 个斐波那契数[37]。***n*** 的值应设置为参数（例如，程序员定义的常量）。计算斐波那契数的公式如下：

$$\textbf{\textit{fibonacci}}(\textbf{\textit{n}}) = \begin{cases} \textbf{\textit{n}} & \textbf{\textit{if}}\ \ \textbf{\textit{n}}=\textbf{0}\ \ \textbf{or}\ \ \textbf{\textit{n}}=\textbf{1} \\ \textbf{\textit{fibonacci}}(\textbf{\textit{n}}-\textbf{2}) + \textbf{\textit{fibonacci}}(\textbf{\textit{n}}-\textbf{1}) & \textbf{\textit{if}}\ \ \textbf{\textit{n}}\geq\textbf{2} \end{cases}$$

使用调试器执行程序并显示最终结果。测试程序中 ***n*** 的各种值。创建调试器输入文件以显示十进制和十六进制结果。

37 更多 information, refer to: http://en.wikipedia.org/wiki/Fibonacci_数字

Why did the programmer quit his job?
Because he didn't get arrays.



8.0 寻址模式

本章节提供了一些关于寻址模式和x86-64架构相关地址操作的基本信息。

寻址方式是使用被访问数据项的地址来访问内存中值的支持方法（读取或写入）。这可能包括变量的名称或数组中的位置。

基本寻址模式有:

- 注册
- 立即
- 内存

每个这些模式都在以下各节中用示例进行了描述。此外，还提供了一个访问数组的简单示例。

8.1 地址和值

在64位架构上，地址需要64位。

如前一章所述，访问内存的唯一方式是使用括号（[]'s）。省略括号将无法访问内存，而是获取项的地址。例如:

```
mov rax, qword [var1] ; var1的值存入rax mov rax, var1 ; var1的地址存入rax
```

由于省略括号不是错误，汇编器不会生成错误消息或警告。

当访问内存时，在许多情况下操作数的大小是明确的。例如，该指令

```
mov eax, [rbx]
```

移动内存中的一个双字。然而，对于某些指令，大小可能不明确。例如，

```
inc [rbx]                                    ; 错误
```

由于不清楚所访问的内存是字节、字还是双字，因此是模糊的。在这种情况下，必须使用 *byte*、*word* 或 *dword*、*qword* 大小限定符来指定操作数大小。例如，

```
inc byte [rbx] inc word [rb
x] inc dword [rbx]
```

每条指令都需要大小规格说明，以便清晰且合法。

8.1.1 注册模式寻址

寄存器模式寻址意味着操作数是CPU寄存器（eax、ebx等）。例如：

```
mov eax, ebx
```

eax 和 ebx 都在寄存器模式寻址。

8.1.2 立即模式寻址

立即模式寻址意味着操作数是一个立即值。例如：

```
mov eax, 123
```

目标操作数 eax 是寄存器模式寻址。123 是立即数模式寻址。应该清楚，在这个例子中，目标操作数不能是立即数模式。

8.1.3 内存模式寻址

内存模式寻址意味着操作数是内存中的一个位置（通过地址访问）。这被称为 *indirection* 或 *dereferencing*。

最基本形式的内存模式寻址已在上一章中广泛使用。具体来说，指令：

```
mov rax, qword [qNum]
```

将访问变量 ***qNum*** 的内存位置并检索存储在该处的值。这需要CPU等待检索到值后才能完成操作，因此可能比使用立即值的类似操作稍微慢一些。

当访问数组时，需要一种更通用的方法。具体来说，可以将一个地址放入寄存器中，并通过寄存器（而不是变量名）进行间接寻址。

例如，假设以下声明：

```
lst dd 101, 103, 105, 107
```

十进制数101的十六进制值为0x00000065。内存图如下：

	Value	Address	Offset	Index
	00	0x00000000006000ef	lst + 15	
	00	0x00000000006000ee	lst + 14	
	00	0x00000000006000ed	lst + 13	
	6b	0x00000000006000ec	lst + 12	**lst[3]**
	00	0x00000000006000eb	lst + 11	
	00	0x00000000006000ea	lst + 10	
	00	0x00000000006000e9	lst + 9	
	69	0x00000000006000e8	lst + 8	**lst[2]**
	00	0x00000000006000e7	lst + 7	
	00	0x00000000006000e6	lst + 6	
	00	0x00000000006000e5	lst + 5	
	67	0x00000000006000e4	lst + 4	**lst[1]**
	00	0x00000000006000e3	lst + 3	
	00	0x00000000006000e2	lst + 2	
	00	0x00000000006000e1	lst + 1	
lst →	65	0x00000000006000e0	lst + 0	**lst[0]**

数组的第一个元素可以按以下方式访问：

```
mov eax, dword [lst]
```

另一种访问第一个元素的方法如下：

```
mov rbx, 列表mov eax, dword [rb
x]
```

在这个示例中，列表的起始地址或基本地址被放置在rbx（第一行）中，然后访问该地址的值并将其放置在rax寄存器中（第二行）。这使我们能够轻松访问数组中的其他元素。

回忆一下，内存是“字节寻址的”，这意味着每个地址都是1字节的信息。双字变量是32位或4字节，因此每个数组元素使用4字节内存。因此，下一个元素（103）是起始地址（lst）加4，下一个元素（105）是起始地址（lst）加8，每个后续元素增加4个偏移量。

偏移量根据数据大小增加。例如，字节数组会增加偏移量1，字数组会增加2，双字数组会增加4，四字数组会增加8。

偏移量是加到基本地址上的数量。索引是用于高级语言中的数组元素编号。

有几种方法可以访问数组元素。一种方法是使用基址并加上偏移量。例如，给定以下初始化：

```
mov rbx, lstmov rsi, 8
```

每个以下指令都访问第三个元素（在上面的列表中为105）。

```
mov eax, dword [lst+8]mov eax, dword
[rbx+8]mov eax, dword [rbx+rsi]
```

在每种情况下，起始地址加8被访问，并将105的值放入eax寄存器。位移被添加，访问内存位置，同时不改变任何源操作数寄存器（rbx，rsi）。具体使用的方法由程序员决定。

此外，位移可能以更复杂的方式计算。

内存寻址的一般格式如下：

[基本地址 + (索引寄存器 * 缩放值) + 位移]

baseAddr 是一个寄存器或变量名。***indexReg*** 必须是寄存器。***scaleValue*** 是 1、2、4、8（1 是合法的，但无意义）的立即数。***displacement*** 必须是立即数。总和代表一个 64 位地址。

元素可以以任何组合使用，但必须是合法的，并生成一个有效的地址。

以下是一些源操作数内存寻址的示例：

```
mov eax, dword [var1]mov rax, qword [rbx+rsi]mo
v ax, word [lst+4]mov bx, word [lst+rdx+2]mov rc
x, qword [lst+ (rsi*8) ]mov al, byte [buff-1+rcx]m
ov eax, dword [rbx+ (rsi*4) +16]
```

例如，要访问先前定义的双字数组中的3rd元素（由于索引从0开始，因此索引为2）：

```
mov     rsi, 2                          ; index=2
mov     eax, dword [lst+rsi*4]          ; get lst[2]
```

由于地址始终是 *qword*（在64位架构上），因此使用64位寄存器进行内存模式寻址（即使访问双字值时也是如此）。这允许寄存器更类似于高级语言中的数组索引。

例如，内存操作数 [lst+rsi*4] 与高级语言中的 lst[rsi] 相似。rsi 寄存器乘以数据大小（在这个例子中是 4，因为每个元素是 4 字节）。

8.2 示例程序，列表求和

以下示例程序将计算列表中的数字之和。

```
; 简单示例用于求和和平均值；数字列表。; ****************************
************************** ; 数据声明
```

section .data ; ----- ; 定义常
量

EXIT_SUCCESS equ 0 ; 成功操作 SYS_exit equ 60 ; 终止调用代码

; ----- ; 定义数据。

section .data lst dd 1002, 1004, 1006, 1008, 10010 len dd 5 sum dd 0

; ********** section .text gl
obal _start _start:

求和循环。

mov ecx, dword [len] ; 获取长度值mov rsi, 0 ; 索引=0sumLoop: mov eax, dwor
d [lst+ (rsi*4)] ; 获取 lst[rsi]add dword [sum], eax ; 更新总和inc rsi ; 下一个项目loo
p sumLoop

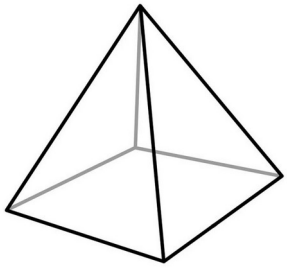
; ----- ; 完成，终止程序。

最后：mov rax, SYS_exit ; 退出调用代码 mov rdi, EXIT_SUCCESS ; 成功退出 syscal
l

括号内的()不是必需的，只是为了清晰起见而添加的。因此，[lst+ (rsi*4)]与[lst+rsi
*4]完全相同。

8.3 示例程序，金字塔面积和体积

这是一个简单的汇编语言程序，用于计算一些几何量。关于一系列正方形棱锥中每个正方形棱锥的信息。具体来说，程序将找到每个正方形棱锥的侧总面积（包括底面）和体积。



一旦计算了值，程序将找到总面积和体积的最小值、最大值、总和和平均值。

所有数据都是无符号值（即，使用mul和div，不使用imul或idiv）。

此示例中使用的基本方法是循环计算表面积和体积数组。第二个循环用于查找每个数组的总和、最小值和最大值。为了找到最小和最大值，最小和最大变量分别初始化为列表中的第一个值。然后，将列表中的每个元素与当前最小和最大值进行比较。如果列表中的当前值小于当前最小值，则将最小值设置为当前值（覆盖先前值）。当检查完所有值后，最小值将代表列表中的真实最小值。如果列表中的当前值大于当前最大值，则将最大值设置为当前值（覆盖先前值）。当检查完所有值后，最大值将代表列表中的真实最大值。

```
; 示例汇编语言程序，用于计算一系列正方形金字塔中每个正方形金字塔
的几何信息。

; 程序计算每个正方锥的总表面积和体积；一旦计算了值，程序将找
到总表面积和体积的最小值、最大值、总和和平均值；-----；公式：
；  totalSurfaceAreas(n) = aSides(n) *；  (2*aSides(n)*sSides(n))
```

; 体积(n) = (aSides(n)^2 * heights(n)) / 3

; **

节数 .data

; ----- ; 定义常量

EXIT_SUCCESS equ 0 ; 成功操作 SYS_exit equ 60 ; 终止调用代码

; ----- ; 提供的数据

aSides db 10, 14, 13, 37, 54 db 31, 13, 20, 61, 36 db 14, 53, 44, 19, 42 db 27, 41, 53, 62, 10 db 19, 18, 14, 10, 15 db 15, 11, 22, 33, 70 db 15, 23, 15, 63, 26 db 24, 33, 10, 61, 15 db 14, 34, 13, 71, 81 db 38, 13, 29, 17, 93 sSides dw 1233, 1114, 1773, 1131, 1675 dw 1164, 1973, 1974, 1123, 1156 dw 1344, 1752, 1973, 1142, 1456 dw 1165, 1754, 1273, 1175, 1546 dw 1153, 1673, 1453, 1567, 1535 dw 1144, 1579, 1764, 1567, 1334 dw 1456, 1563, 1564, 1753, 1165 dw 1646, 1862, 1457, 1167, 1534 dw 1867, 1864, 1757, 1755, 1453 dw 1863, 1673, 1275, 1756, 1353 heights dd 14145, 11134, 15123, 15123, 14123 dd 18454, 15454, 12156, 12164, 12542 dd 18453, 18453, 11184, 15142, 12354 dd 14564, 14134, 12156, 12344, 13142 dd 11153, 18543, 17156, 12352, 15434

```
                dd 18455, 14134, 12123, 15324, 13453 dd 11134, 141
                34, 15156, 15234, 17142 dd 19567, 14134, 12134, 175
                46, 16123 dd 11134, 14134, 14576, 15457, 17142 dd 1
                3153, 11153, 12184, 14142, 17134

长度 dd 50

taMin dd 0 taMax dd 0
taSum dd 0 taAve dd 0


volMin dd 0 volMax d
d 0 volSum dd 0 volAv
e dd 0

; ----- ; 额外变量


ddTwo dd 2 ddThree d
d 3

未初始化数据


节 .bss 总面积 resd 50 卷 resd 50


; ***********************************************

章节 .text 全局 _start _
start:

; 计算          食用体积，侧面积和总面积                              区域

        mov ecx, dword [length]                                 ; 长度计数器
        mov rsi, 0                                              ; 索引
```

计算循环: ; totalAreas(n) = aSides(n) * (2*aSides(n)*sSides(n)) movzx r8d, byt
e [aSides+rsi] ; aSides[i] movzx r9d, word [sSides+rsi*2] ; sSides[i] mov eax, r8d
mul dword [ddTwo] mul r9d mul r8d mov dword [totalAreas+rsi*4], eax

; 体积(n) = (aSides(n)^2 * heights(n)) / 3

movzx eax, byte [aSides+rsi] mul eax mul dword
[heights+rsi*4] div dword [ddThree] mov dword
[volumes+rsi*4], eax inc rsi loop calculationLoop

; ----- ; 查找总面积和体积的最小值、最大值、总和和平均值mov ea
x, dword [totalAreas]mov dword [taMin], eaxmov dword [taMax], eax
mov eax, dword [volumes]mov dword [volMin], eaxmov dword [volMa
x], eaxmov dword [taSum], 0mov dword [volSum], 0mov ecx, dword [l
ength]mov rsi, 0

statsLoop: mov eax, dword [totalAreas+rsi*4] add dword [t
aSum], eax

cmp eax, dword [taMin] jae notNewT
aMin mov dword [taMin], eax 比较
x与dword [taMin]，如果大于或等
于则跳转到notNewTaMin，否则将
eax的值移动到dword [taMin]。

notNewTaMin: cmp eax, dword [taMax] jb
e notNewTaMax mov dword [taMax], eax

notNewTaMax: mov eax, dword [volumes+rsi*4] add
dword [volSum], eax cmp eax, dword [volMin] jae not
NewVolMin mov dword [volMin], eax

notNewVolMin: cmp eax, dword [volMax] j
be notNewVolMax mov dword [volMax], ea
x

notNewVolMax:

增加 rsi 循环统计循环

; ----- ; 计算平均值。

mov eax, dword [taSum]mov edx, 0 di
v dword [length]mov dword [taAve], e
axmov eax, dword [volSum]mov edx,
0 div dword [length]mov dword [volA
ve], eax

; ----- ; 完成，终止程序。

最后:

```
    mov rax, SYS_exit                                   ; 退出代码调用
    mov rdi, EXIT_SUCCESS    移动di, EXIT_SUCCESS       ; 成功退出
    系统调用
```

这是一个示例。解决此问题有多种其他有效方法。

8.4 练习

以下是本章的一些测验问题和基于此章节的建议项目。

8.4.1 测验问题

以下是本章的一些测验问题。

1) 解释以下两个指令之间的区别:

```
1.  mov   rdx, qword [qVar1]2.  mov
rdx, qVar1
```

2) 以下指令中每个源操作数的寻址模式是什么？请回答为*Register*、*Immediate*、*Memory*或*Illegal Instruction*。

Note, mov <dest>, <source> mov ebx, 14
mov ecx, dword [rbx] mov byte [rbx+4],
10 mov 10, rcx mov dl, ah mov ax, word
[rsi+4] mov cx, word [rbx+rsi] mov ax,
byte [rbx]

3) 给定以下变量声明和代码片段:

```
ans1   dd   7 mov rax, 3
mov rbx, ans1
```

```
add     eax, dword [rbx]
```

在执行后，eax 寄存器中会有什么？以十六进制显示，完整寄存器大小。

4) 给定以下变量声明和代码片段：

列表1 dd 2, 3, 4, 5, 6, 7

mov rbx, list1add rbx, 4mov eax
, dword [rbx]mov edx, dword [li
st1]

在执行后，eax 和 edx 寄存器中会有什么内容？以十六进制形式显示，包括完整的寄存器大小。

5) 给定以下变量声明和代码片段：

lst dd 2, 3, 5, 7, 9mov rsi, 4mov eax, 1mov rcx, 2lp:
add eax, dword [lst{v*}rsi]add rsi, 4loop lpmov eb
x, dword [lst]

执行后，eax、ebx、rcx和rsi寄存器中会有什么内容？以十六进制形式显示答案，完整寄存器大小。*Note*，请注意寄存器大小（32位与64位）。

6) 给定以下变量声明和代码片段：

```
        list      dd    8, 6, 4, 2, 1, 0

          mov      rbx, list
          mov      rsi, 1
          mov      rcx, 3
          mov      edx, dword [rbx]
  lp:     mov      eax, dword [list+rsi*4]
          inc      rsi
          loop     lp
```

imul dword [list]

执行后，eax、edx、rcx和rsi寄存器中会有什么内容？以十六进制形式显示答案，完整寄存器大小。*Note*，请注意寄存器大小（32位与64位）。

7) 给定以下变量声明和代码片段：

列表 dd 8, 7, 6, 5, 4, 3, 2, 1, 0

mov rbx, listmov rsi, 0mov rcx, 3mov e
dx, dword [rbx]lp: add eax, dword [list+rsi*4]inc rsi
loop lpcdqidiv dword [list]

执行后，eax、edx、rcx和rsi寄存器中会有什么内容？以十六进制形式显示答案，完整寄存器大小。*Note*，请注意寄存器大小（32位与64位）。

8) 给定以下变量声明和代码片段：

```
list      dd   2, 7, 4, 5, 6, 3

          mov    rbx, list
          mov    rsi, 1
          mov    rcx, 2
          mov    eax, 0
          mov    edx, dword [rbx+4]
lp:       add    eax, dword [rbx+rsi*4]
          add    rsi, 2
          loop   lp
          imul   dword [rbx]
```

执行后，eax、edx、rcx和rsi寄存器中会有什么内容？以十六进制形式显示答案，完整寄存器大小。*Note*，请注意寄存器大小（32位与64位）。

8.4.2 建议项目

以下是本章的一些建议项目。

1) 实现示例程序以计算数字列表的总和。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。2) 将上一个问题中的示例程序更新为查找数字列表的最大值、最小值和平均值。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。

3) 实现示例程序以计算一组正方形棱锥的每个正方形棱锥的侧总面积（包括底面）和体积。计算完这些值后，程序将找出总面积和体积的最小值、最大值、总和和平均值。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

4) 编写一个汇编语言程序，用于找到一组数字的最小值、中间值、最大值、总和和整数平均值。此外，程序还应找到负数的总和、计数和整数平均值。程序还应找到能被3整除的数字的总和、计数和整数平均值。与中位数不同，'中间值'不需要对数字进行排序。对于奇数个元素，中间值定义为中间值。对于偶数个值，它是两个中间值的整数平均值。假设所有数据都是无符号的。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。5) 使用有符号值和有符号操作重复前面的程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

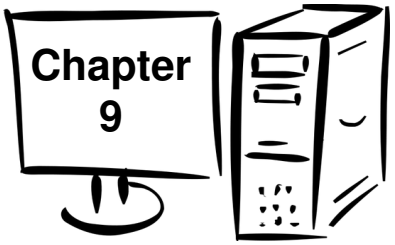
6) 创建一个程序来排序数字列表。使用以下冒泡排序[38]算法:

```
for ( i = (len-1) to 0 ) {      swapped = false      for ( j
= 0 to i-1 )          if ( lst(j) > lst(j+1) ) {              tmp
= lst(j)              lst(j) = lst(j+1)              lst(j+1) = tm
p             swapped = true          }      if ( swapped = f
alse ) exit }
```

使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

38 Fo r more information, refer to: http://en.wikipedia.org/wiki/Bubble_sort

A programmer is heading out to the grocery store, and is asked to "get a gallon of milk, and if they have eggs, get a dozen."
He returns with 12 gallons of milk.



9.0 进程栈

在计算机中，栈是一种数据结构，其中项目以相反的顺序添加和从栈中移除。也就是说，最近添加的项目是第一个被移除的。这通常被称为后进先出（LIFO）。

栈在程序中用于在过程或函数调用期间存储信息。下一章提供了有关栈的信息和示例。

向栈中添加项称为***push***或入栈操作。从栈中移除项称为***pop***或出栈操作。

预期读者将熟悉栈的一般概念。

9.1 栈示例

为了演示栈的一般用法，给定一个数组，一个 = {7, 19, 37}，考虑以下操作：

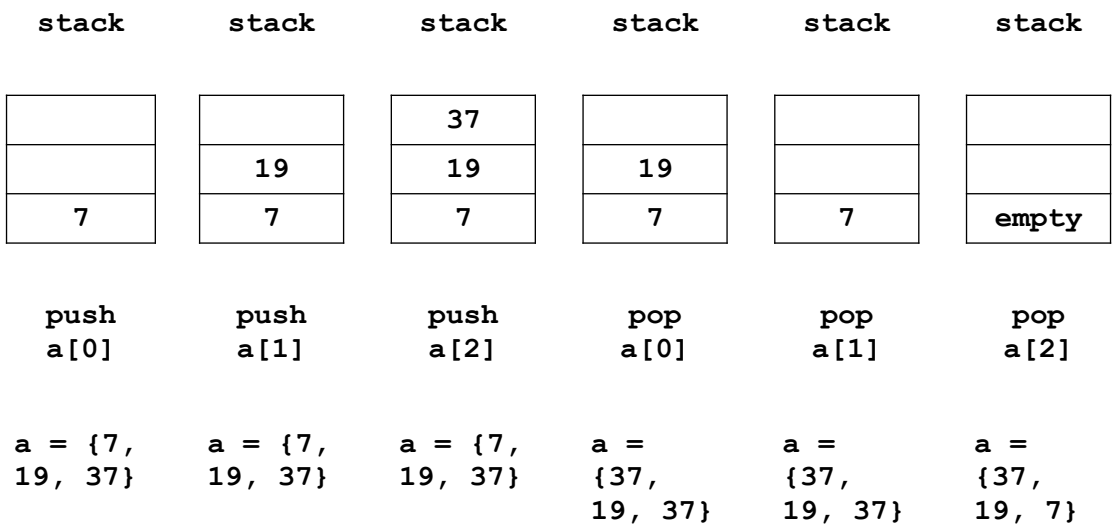
push a[0] push a
[1] push a[2]

随后是以下操作：

```
pop     a[0]
pop     a[1]
pop     a[2]
```

初始推送将推送7，然后是19，最后是37。由于栈是后进先出，因此从栈中弹出的第一个项目将是最后推送的项目，或者在这个例子中是37。37被放置在数组的第一个元素中（覆盖了7）。随着这个过程继续，数组元素的顺序被反转。

以下图表显示了进度和结果。



以下部分提供了有关堆栈实现以及适用堆栈操作和指令的更多详细信息。

9.2 栈指令

一个推操作将事物放入栈中，一个出操作从栈中取出事物。这些命令的格式为:

push <操作数64 pop <操作数64

操作数可以是寄存器或内存，但不允许立即数。一般来说，压栈和出栈操作将压入架构大小。由于架构是64位的，我们将压入和弹出四字组。

栈在内存中是反向实现的。有关原因的详细解释请参阅以下章节。

堆栈指令总结如下:

Instruction	Explanation
push <op64>	Push the 64-bit operand on the stack. First, adjusts **rsp** accordingly (**rsp**-8) and then copy the operand to [**rsp**]. The operand may not be an immediate value. Operand is not changed.
Examples:	**push rax** **push qword [qVal]** ; value **push qVal** ; address
pop <op64>	Pop the 64-bit operand from the stack. Adjusts **rsp** accordingly (**rsp**+8). The operand may not be an immediate value. Operand is overwritten.
Examples:	**pop rax** **pop qword [qVal]** **pop rsi**

如果必须推送超过64位的数据，则需要多个推送操作。虽然可以推送和弹出小于64位的操作数，但并不推荐这样做。

更多　完整指令列表位于Appen　　十个

B.

9.3 栈实现

rsp寄存器用于指向内存中栈的当前顶部。在这个架构中，与大多数架构一样，栈在内存中向下增长。

9.3.1 栈布局

如第2章“架构”中所述，程序的通用内存布局如下:

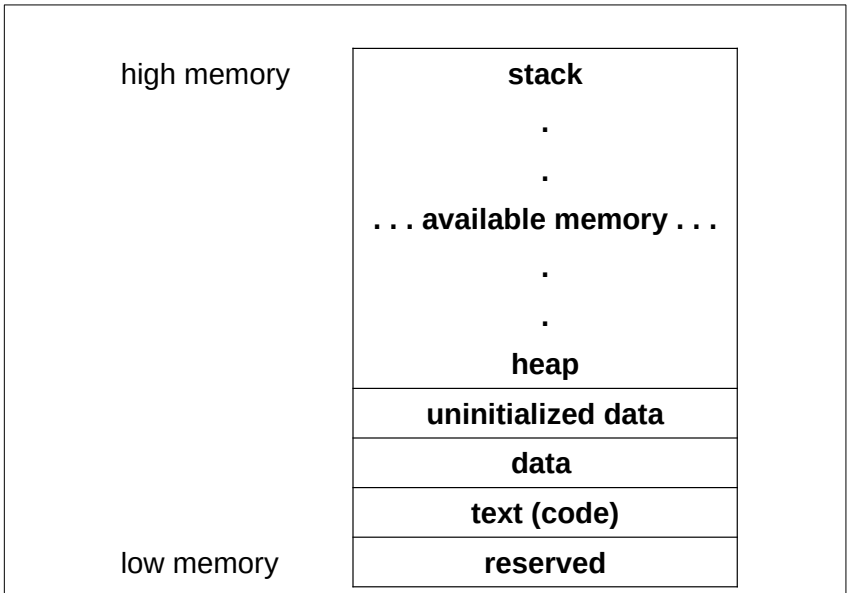


Illustration 22: Process Memory Layout

堆是动态分配的数据将被放置的地方（如果请求）。例如，使用C++ new运算符或C malloc()系统调用的项目。由于动态分配的数据是在运行时创建的，堆通常向上增长。然而，栈从高内存开始，向下增长。栈用于临时存储信息，例如函数调用的调用帧。大型程序或递归函数可能会使用大量的栈空间。

随着堆和栈的扩展，它们向彼此生长。这是为了确保最有效的整体内存使用。

一个使用大量栈空间和最小堆空间的程序（进程A）将能够运行。一个使用最小栈空间和非常大量堆空间的程序（进程B）也将能够运行。

例如:

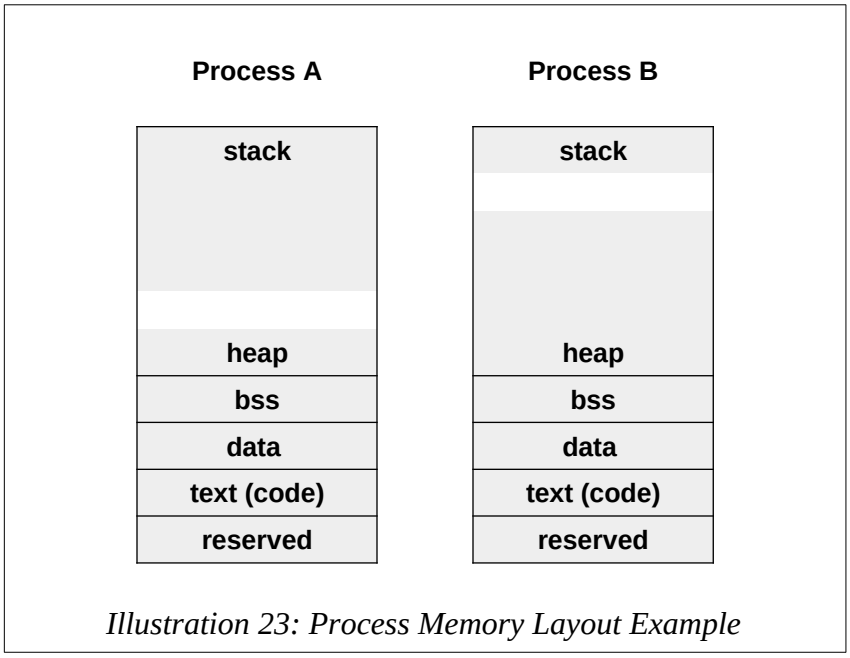


Illustration 23: Process Memory Layout Example

当然，如果栈和堆相遇，程序将会崩溃。如果发生这种情况，将没有可用内存。

9.3.2 栈操作

基本栈操作 push 和 pop 在其操作过程中调整栈指针寄存器 rsp。

对于推操作:

1. rsp寄存器减去8（1个四倍字）。2. 操作数被复制到[rsp]栈中。

操作数未被更改。这些操作的顺序很重要。

对于弹出操作:

1. 当前栈顶，在[rsp]，被复制到操作数中。2. rsp寄存器增加8（1个四倍字）。

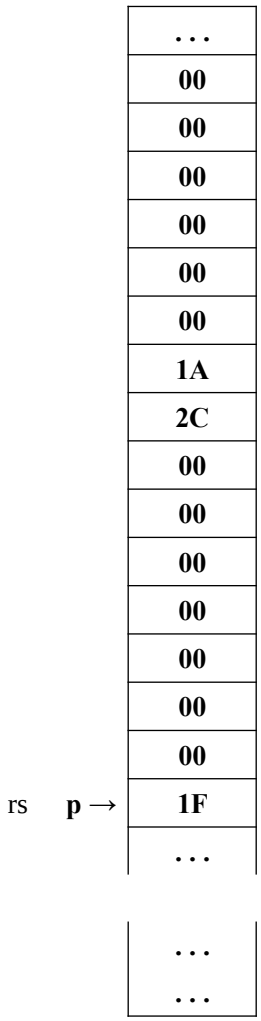
这些操作的顺序与推操作完全相反。弹出的项实际上并未删除。然而，程序员不能指望该项仍然保留在

栈在弹出操作之后。之前已推入但未弹出的项可以访问。

例如：

```
mov rax, 6700                    ; 6700₁₀ = 00001A2C₁₆
push raxmov rax, 31
                                 ;  31₁₀ = 0000001F₁₆
push rax
```

会产生以下堆配置（其中每个框是一个字节）：



布局显示该架构是低位序的，因为最低有效字节被放置在最低的内存位置中。

9.4 栈示例

以下是一个使用栈就地反转四字节数组列表的示例程序。具体来说，在第一个循环中，四字节数组中的每个值都放置在栈中。在第二个循环中，从栈中移除每个元素并将其放回数组中（覆盖）之前的值。

; 简单示例演示基本栈操作。; 在原地反转数字列表；方法：将每个数字放入栈中，然后弹出每个数字；　　再将其放回内存中。; ********** ********** ; 数据声明部分 .data ; ----- ; 定义常量

EXIT_SUCCESS equ 0 ; 成功操作 SYS_exit equ 60 ; 终止调用代码

; ----- ; 定义数据。

数字 dq 121, 122, 123, 124, 125 长度 dq 5

; **********
章节 .text 全局 _start _
start:

; 循环以将数字压入堆栈。

mov rcx, qword [len] mov rbx, nu
mbers mov r12, 0 mov rax, 0

pushLoop: push qword [rbx+r12*8] inc r
12 loop pushLoop

; ----- ; 所有数字都在栈上（顺序相反）。; 循环以将它们取回。; 将它们放回原始列表中... mov rcx, qword [len] mov rbx, numbers mov r12, 0 popLoop: pop rax mov qword [rbx+r12*8], rax inc r12 loop popLoop ; ----- ; 完成，终止程序。

最后：mov rax, SYS_exit ; 退出调用代码 mov rdi, EXIT_SUCCESS ; 成功退出 syscall

有其他方法来完成这个功能（反转列表），然而这是为了演示栈操作。

9.5 练习

以下是本章的一些测验问题和基于此章节的建议项目。

9.5.1 测验问题

以下是本章的一些测验问题。

1) 哪个寄存器指向栈顶？2) 执行 `push rax` 指令后会发生什么（两件事）？ 3) `pop rax` 指令会从栈中移除多少 ***bytes*** 的数据?

4) 给定以下代码片段：

```
mov r10, 1mov r11,
2mov r12, 3push r1
0push r11push r12p
op r10pop r11pop r
12
```

执行后，r10、r11和r12寄存器中会有什么内容？以十六进制形式，显示完整寄存器大小。

5) 给定以下变量声明和代码片段：

```
lst dq 1, 3, 5, 7, 9 mov rsi, 0 mov rcx, 5 lp1: pu
sh qword [lst+rsi*8] inc rsi loop lp1 mov rsi, 0
mov rcx, 5 lp2: pop qword [lst+rsi*8] inc rsi lo
op lp2 mov rbx, qword [lst]
```

解释代码（执行后）的 ***result*** 是什么？

6) 提供一个内存中向下增长的栈的优点。

9.5.2 建议项目

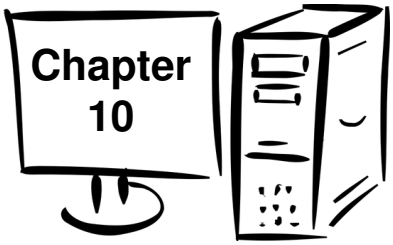
以下是本章的一些建议项目。

1) 实现一个反转数字列表的示例程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

2) 编写一个程序，以确定表示单词的以NULL结尾的字符串是否是回文[39]。回文是指正向和反向读都相同的单词。例如，“anna”，“civic”，“hannah”，“kayak”和“madam”都是回文。这可以通过逐个将字符推入栈中，然后从开始处比较栈项与字符串来实现。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

3) 更新之前的程序以测试一个短语是否是回文。使用栈的一般方法相同，但是必须跳过空格和标点符号。例如，“A man, a plan, a canal – Panama！”是一个回文。程序必须忽略逗号、破折号和感叹号。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

39 Fo r more information, refer to: http://en.wikipedia.org/wiki/Palindrome



10.0 程序开发

编写或开发程序时，遵循明确的方法更容易。该方法的主要步骤是:

- 理解问题
- 设计算法
- 实现程序
- 测试/调试程序

为了详细展示此过程，以下各节将应用这些步骤到一个简单的示例问题中。

10.1 理解问题

在尝试创建解决方案之前，完全理解问题是至关重要的。确保对问题的全面理解可以帮助减少错误并节省时间和精力。第一步是理解需要什么，特别是适用的输入信息和预期的结果或输出。

考虑将单个整数转换为表示该整数的字符串或一系列字符的问题。为了明确，整数可以用于数值计算，但不能以（它本身）的形式显示在控制台上。字符串可以显示在控制台上，但不能用于数值计算。

对于这个例子，只考虑无符号（仅正数）值。处理有符号值的小额外努力留给读者作为练习。

作为一个无符号双字整数，数值1498_{10}在十六进制中表示为0x000005DA（双字大小）。整数1498_{10}（0x000005DA）将用字符串“1”、“4”、“9”、“8”表示，并以空字符结尾。这总共需要5个字节，因为不需要符号或前导空格。

特定示例。因此，字符串“1498”将如下表示：

Character	“1”	“4”	“9”	“8”	NULL
ASCII Value (decimal)	49	52	57	56	0
ASCII Value (hex)	0x31	0x34	0x39	0x38	0x0

目标是将单个整数转换为适当的字符序列，以形成一个以空字符终止的字符串。

10.2 创建算法

算法是解决问题的关键步骤的明确、有序序列的名称。一旦理解了程序，就可以开发一系列步骤来解决该问题。对于给定的问题，可能有，通常也有，多个正确解决方案。

创建算法的过程对不同人来说可能不同。一般来说，应该花一些时间思考可能的解决方案。这可能包括使用一张草稿纸来尝试一些可能的解决方案。一旦选定了方法，该解决方案就可以开发成一个算法。该算法应该被写下、审查和改进。然后，该算法被用作程序的框架。

例如，我们将考虑上一节概述的整数到ASCII转换问题。要将单个数字整数（0-9）转换为字符，可以将48_{10}（或“0”或0x30）加到整数上。例如，0x01 + 0x30是0x31，它是“1”的ASCII值。很明显，这个技巧只适用于单个数字（0-9）。

为了将一个较大的整数（≥ 10）转换为字符串，必须将该整数分解为其组成部分的数字。例如，123_{10}（0x7B）将分解为1、2和3。这可以通过反复执行整数除以10，直到得到0的结果来实现。

例如;

$$\frac{123}{10} = 12 \quad remainder\ 3$$

$$\frac{12}{10} = 1 \quad remainder\ 2$$

$$\frac{1}{10} = 0 \quad remainder\ 1$$

如所见，余数代表个别位置。但是，它们是反转排序得到的。为了解决这个问题，程序可以将余数压入栈中，当分配完毕后，弹出余数并转换为ASCII代码，并存储在字符串中（字符串是字节数组）。
这个过程构成了算法的基础。需要注意的是，有许多开发此算法的方法。以下是一种方法的示例:

```
; A部分 - 连续除法
;           数字计数 = 0
;           获取整数
;     divideLoop:
;           除以10
;           推送余数
;           增加数字计数
;           if (result > 0) 跳转到 divideLoop

; Part B – 转换余数并存储 ; 获取字符串的起始地址（字节数组） ; idx = 0 ;
popLoop: ; 弹出整数字符 ; charDigit = intDigit + “0” (0x030) ; string[idx] =
charDigit ; 增加idx ; 减少digitCount ; 如果 (digitCount > 0) 跳转到popLoop ; s
tring[idx] = NULL
```

算法步骤以程序注释的形式展示，以便于理解。算法通常在纸上开始，然后以如上所示的形式更正式地用伪代码编写。在极不可能的情况下，如果程序第一次运行不成功，注释将是主要的调试清单。

一些程序员会跳过注释，最终会花费更多的时间进行调试。注释代表了算法，而代码则是该算法的实现。

10.3 实现程序

基于算法，可以开发和实现一个程序。算法被扩展，并根据算法中概述的步骤添加代码。这允许程序员专注于当前编码部分的特定问题，包括数据类型和数据大小。此示例仅针对无符号数据，因此使用无符号除法（DIV，而不是IDIV）。由于整数是双字，它必须转换为四字才能进行除法。然而，除法后的结果和余数也将是双字。由于堆栈是四字，整个四字寄存器将被推入。寄存器的最高位部分不会被访问，因此其内容无关紧要。

一种算法的可能实现如下:

```
;  Simple example program to convert an
;  integer into an ASCII string.

; ********************************************************
;  Data declarations

section    .data

; -----
;  Define constants

NULL              equ     0
EXIT_SUCCESS      equ     0             ; successful operation
SYS_exit          equ     60            ; code for terminate

; -----
;  Define Data.

intNum         dd      1498
```

节 .bss strNum resb 10

; **

章节 .text 全局 _start _
start:

; 将整数转换为ASCII字符串。

; ----- ; Part A - 连续除法

mov eax, dword [intNum] ; 获取整数mov rcx, 0 ; 数字计数 = 0mov ebx, 10 ; 设置除以10

divideLoop: mov edx, 0 div ebx ; 除以10 push rdx ; 压入余数 inc rcx ; 增加数字计数 cmp eax, 0 ; 如果 (结果 {v*} 0) jne divideLoop ; 跳转至 divideLoop

; ----- ; Part B - 转换余数并存储

mov rbx, strNum 获取字符串地址
mov rdi, 0 ; idx = 0

popLoop: pop rax
; 弹出 intDigit

添加 al, "0" ; 字符 = 整数 + "0"

mov byte [rbx+rdi], al ; 字符串[idx] = 字符 inc rdi ; 增加idx loop popLoop ; 如果 (digitCount > 0)

; 跳转到 popLoopmov byte [r bx+rdi], NULL ; string[idx] = NULL; -----; 完成，终止程序。last:mov rax, SYS_exit ; 退出代码mov rdi, EXIT_SUCCESS ; 成功退出syscall

有许多不同的有效实现方式适用于此算法。程序应组装以解决任何拼写错误或语法错误。

10.4 测试/调试程序

一旦程序编写完成，应进行测试以确保程序运行正常。测试将基于程序的特定参数进行。

在这种情况下，程序可以使用调试器执行并在程序末尾附近停止（例如，在本例中的标签“last”处）。启动ddd调试器后，可以输入命令b last和run，这将运行程序直到但不执行由标签“last”引用的行。结果字符串***strNum***可以在调试器中使用x/s &strNum查看，将显示字符串地址和内容，应为“1498”。例如；

```
(gdb) x/s &strNum 0x60
0104: "1498"
```

如果字符串显示不正确，可能值得使用 x/5cb &strNum 调试命令检查五个（5）字节数组的每个字符。输出将显示字符串的地址，后跟十进制和ASCII表示。

例如；

```
(gdb) x/5cb &strNum 0x600104: 49 '1' 52 '4' 57 '9' 56 '8' 0 '\000'
```

这个输出的格式最初可能令人困惑 .

如果输出不正确，程序员将需要调试代码。对于这个例子，有两个主要步骤；连续除法和转换/存储余数。第二个步骤需要第一个步骤正常工作，因此第一个步骤应该被验证。这可以通过使用调试器仅关注第一个部分来完成。在

这个示例中，第一步应该恰好迭代4次，因此rcx应该是4。此外，应按顺序将8、9、4和1压入栈中。这可以通过在调试器中查看rdx的寄存器内容在压入时或查看栈顶的4个条目来轻松验证。

如果该部分工作正常，则第二部分可以验证。这里，值1、4、9和8应该从栈中弹出（按此顺序）。如果是这样，则将整数转换为字符，通过添加“0”（0x30）并将它们逐个存储在字符串中。可以逐字符查看字符串，以查看它们是否正确地被输入到字符串中。

这种方式，问题可以相当快地缩小范围。高效的调试是一项关键技能，必须通过实践来磨练。

参考第6章，DDD调试器，以获取有关特定调试器命令的更多信息。

10.5 错误术语

如果程序不起作用，了解一些关于错误可能出现在哪里或是什么的基本术语会有所帮助。使用正确的术语可以确保您能够有效地与他人沟通问题。

10.5.1 编译器错误

汇编程序时会产生汇编错误。这意味着汇编器不理解一个或多个指令。汇编器将提供错误列表以及每个错误的行号。建议从上到下解决错误。解决上面的错误可以清除下面的多个错误。

典型的汇编器错误包括拼写指令错误和/或省略变量声明。

10.5.2 运行时错误

运行时错误是导致程序崩溃的事情。

10.5.3 逻辑错误

逻辑错误是指程序执行时没有产生正确的结果。例如，错误地编写提供的公式或尝试在计算总和之前计算一系列数字的平均值。

如果程序存在逻辑错误，找到错误的一种方法是通过显示中间值。有关查找逻辑错误的建议将在后续章节中提供更多信息。

10.6 练习

以下是本章的一些测验问题和基于此章节的建议项目。

10.6.1 测验问题

以下是本章的一些测验问题。

1) 什么是算法?

2) 算法开发中的四个主要步骤是什么？3) 算法开发中的四个主要步骤是否仅适用于汇编语言编程？4) 如果一个操作数乘法指令使用立即数操作数，会发生什么类型的错误（如果有）？请回答汇编时或运行时。5) 如果汇编语言指令拼写错误（例如，“mv”而不是“mov”），错误将在何时被发现？请回答汇编时或运行时。6) 如果引用了标签但没有定义，错误将在何时被发现？请回答汇编时或运行时。7) 如果一个程序在数组中的值上执行一系列除法，除以0时，错误将在何时被发现？请回答汇编时或运行时。

10.6.2 建议项目

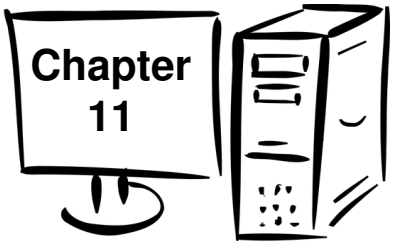
以下是本章的一些建议项目。

1) 实现将整数转换为字符串的示例程序。更改原始整数的值。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。

2) 更新示例程序以处理有符号整数。这需要在字符串中包含前导符号，“+”或“-”。例如，-123_{10} (0xFFFFFF85) 将是“-123”，带有空终止符（总共5个字节）。此外，必须使用有符号除法（IDIV，而不是DIV）和有符号转换（例如，CDQ）。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

3) 编写一个程序将表示数值的字符串转换为整数。例如，给定以NULL结尾的字符串“41275”（总共6个字节），将字符串转换为双字大小的整数（0x0000A13B）。可以假设字符串和结果整数是无符号的。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。4) 更新前面的程序以处理带有前导符号的字符串（“+”或“-”）。这需要在字符串中包含一个符号，“+”或“-”。必须确保最终字符串以NULL结尾。可以假设输入字符串是有效的。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。5) 更新前面的程序，将字符串转换为整数，包括对输入字符串的错误检查。具体来说，符号必须是有效的，并且是字符串的第一个字符，每个数字必须在“0”和“9”之间，并且字符串以NULL结尾。例如，字符串“-321”是有效的，而“1+32”和“+1R3”都是无效的。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。

Why did C++ decide not to go out with C?
Because C has no class.



11.0 宏

一个汇编语言宏是一组预定义的指令，可以轻松地插入到任何需要的地方。一旦定义，该宏就可以根据需要多次使用。当必须多次使用相同的代码集时，它非常有用。宏可以用来减少编码量，简化程序，并减少重复编码的错误。

汇编器包含一个强大的宏处理器，它支持条件汇编、多级文件包含以及两种形式的宏（单行和多行），并具有一个“上下文堆栈”机制以增强宏功能。

在使用宏之前，必须对其进行定义。宏定义应放置在源文件 ***before*** 的数据和代码部分。宏在文本（代码）部分使用。以下部分将提供一个带有定义和使用的详细示例。

11.1 单行宏

有两种关键类型的宏；单行宏和多行宏。这些内容在以下各节中均有描述。

单行宏使用 %define 指令定义。这些定义的工作方式与 C/C++ 类似；因此，您可以执行类似以下操作：

```
%define mulby4(x) shl x, 2
```

然后通过输入使用宏：

```
mulby4 (rax)
```

在源代码中，它将内容乘以4（通过移动两个位）并存储到rax寄存器中。

11.2 多行宏

多行宏可以包含不同数量的行（包括一行）。多行宏更有用，以下章节将主要关注多行宏。

11.2.1 宏定义

在使用多行宏之前，必须首先定义它。其一般格式如下:

```
%macro <name> <参数数量>
    ; [宏体]
%endmacro
```

参数可以通过宏引用为 %<数字>，其中 %1 是第一个参数，%2 是第二个参数，依此类推。

为了使用标签，宏内的标签必须在标签名称前加上 %%。

这将确保多次调用相同的宏时每次都会使用不同的标签。例如，绝对值函数的宏定义如下:

```
%macro abs 1      cmp %
1, 0      jge %%done
neg %1%%done:%endm
acro
```

Refer到样本宏程序以获取完整示例 mple.

11.2.2 使用宏

为了使用或“调用”一个宏，它必须放置在代码段中，并通过名称以及适当的参数数量进行引用。

给定以下数据声明:

```
qVar    dq    4
```

然后，调用“abs”宏（两次）:

```
        mov eax, -3
        绝对值 eax

        绝对 qword [qVar]
```

列表文件将显示如下（对于第一次调用）:

```
27 00000000 B8FDFFFFFF          mov  eax, -3
28                              abs  eax
29 00000005 3D00000000  <1>  cmp %1, 0
30 0000000A 7D02        <1>  jge %%done
31 0000000C F7D8        <1>  neg %1
32                      <1> %%done:
```

宏将从定义复制到代码中，适当的参数将在宏体中替换，*each*次使用时替换。<1>表示从宏定义复制的代码。在这两种情况下，%1参数被替换为给定的参数；在这个例子中是eax。

宏使用更多内存，但不需要控制转移的开销（如函数）。

11.3 宏示例

以下示例程序演示了简单宏的定义和使用。

```
; 示例程序以演示一个简单的宏 ;**********************************
**************** ;; 定义宏 ;   使用三个参数调用 ;aver <lst>, <len>, <av
e> %macro aver 3 mov eax, 0 mov ecx, dword [%2] ; 长度
```

```
    mov r12, 0 lea rbx, [%1]

%%sumLoop: 将 eax 加上 dword [rbx+r12*4] ; 获取 list[n] 增加 r12 循
环 %%sumLoop


    cdq idiv dword [%2] mov dword
    [%3], eax

%endmacro

; ********************************************** ; 数据声明
部分 .data ; ----- ; 定义常量



EXIT_SUCCESS equ 0 ; 成功代码SYS_exit equ 60 ; 终止代码

; 定义数据。

section .datalist1 dd   4, 5, 2, -3, 1len1 dd   5ave1 dd   0
list2 dd   2, 6, 3, -2, 1, 8, 19len2 dd   7ave2 dd   0



; **********************************************

章节 .text 全局 _start _
start:
```

```
; ----- ; 在程序中使用宏

    aver list1, len1, ave1 ; 第1次，数据集1 aver list2, len2, ave2 ; 第2次，数据集2


; ----- ; 完成，终止程序。

last:mov rax, SYS_exit ; 退出 mov rdi, EXIT_SUCCESS ; 成功 syscall
```

在这个例子中，宏被调用了两次。每次使用宏时，它都会从定义中复制到文本部分。因此，宏通常使用更多的内存。

11.4 调试宏

代码宏的代码将不会在调试器源代码窗口中显示。当宏运行正确时，这非常方便。然而，在调试宏时，代码必须是可查看的。

为了查看宏代码，显示机器代码窗口（查看 → 机器代码窗口）。在该窗口中，显示指令的机器代码。步骤和下一个指令将执行整个宏。要执行宏指令，必须使用 stepi 和 nexti 命令。

代码在查看时将是展开后的代码（与原始宏的定义相反）。

11.5 练习

以下是本章的一些测验问题和基于此章节的建议项目。

11.5.1 测验问题

以下是本章的一些测验问题。

1) 宏定义在汇编语言源文件中放置在哪里？2）当调用宏时，代码在代码段中放置了多少次?

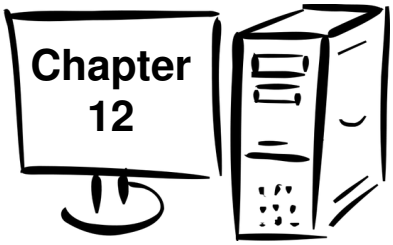
3) 解释为什么在宏中，标签通常由一个 **%%**（双百分号） precede。4) 如果在标签上不包括 **%%**，可能会发生什么？5) 跳转到不包含 **%%** 的标签是否合法？如果不合法，解释原因。如果合法，解释在什么情况下可能有用。6) 宏参数替换何时发生？

11.5.2 建议项目

以下是本章的一些建议项目。

1) 实现一个列表平均值宏的示例程序。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。2) 将上一个问题中的程序更新为包括最小值和最大值。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。3) 创建一个宏，通过将现有列表的每个元素乘以2来更新列表。至少三次调用该宏，每次使用三个不同的数据集。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。4) 从上一章的整数到ASCII转换示例创建一个宏。至少三次调用该宏，每次使用三个不同的数据集。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

Why do programmers mix up Halloween and Christmas?
Because 31 Oct = 25 Dec.



12.0 函数

函数和过程（即无返回值函数）有助于将程序分解成更小的部分，使其更容易编写、调试和维护。函数调用涉及两个主要动作：

- 关联
  - 由于函数可以从代码中的多个不同位置调用，因此该函数必须能够返回到它最初被调用的正确位置。

- 参数传输
  - 函数必须能够访问参数以进行操作或返回结果（即访问按引用传递的参数）。

这些动作如何完成的细节在以下章节中解释。

12.1 更新链接说明

编写和调试函数时，对于C编译器（无论是gcc还是g++）来说，将程序链接起来更容易，因为C编译器知道各种C/C++库的正确位置。

例如，假设源文件名为 *example.asm*，编译、汇编、链接和执行的命令如下：

```
yasm -g dwarf2 -f elf64 example.asm -l example.lst gcc -g -o example exa
mple.o
```

Note, Ubuntu 18 及以上版本需要在 gcc 上启用 no-pie 选项，如下所示：

```
gcc -g -no-pie -o example example.o
```

这将使用GCC编译器调用链接器，读取*example.o*目标文件并创建*example*可执行文件。 “-g”选项以常规方式将调试信息包含在可执行文件中。文件名可以根据需要更改。

12.2 调试器命令

当使用调试器调试具有函数的程序时，查看 ***step*** 和 ***next*** 调试器命令可能会有所帮助。

12.2.1 调试器命令，*next*

关于函数调用，调试器 ***next*** 命令将执行整个函数并跳到下一行。在调试函数时，这有助于快速执行整个函数，然后仅验证结果。它不会显示任何函数代码。

12.2.2 调试器命令，*step*

关于函数调用，调试器 ***step*** 命令将进入函数并跳转到函数代码的第一行。它将显示函数代码。在调试函数时，这有助于调试函数代码。

12.3 栈动态局部变量

在高级语言中，函数中声明的非静态局部变量默认是堆栈动态局部变量。一些C++文本将此类变量称为自动变量。这意味着局部变量是通过在堆栈上分配空间并将这些堆栈位置分配给变量来创建的。当函数完成时，空间被恢复并用于其他目的。这需要少量的额外运行时开销，但使内存的整体使用更加高效。如果一个具有大量局部变量的函数从未被调用，则不会分配局部变量的内存。这有助于减少程序的整体内存占用，通常有助于程序的整体性能。

与静态声明变量不同，静态声明变量在整个程序执行期间分配内存位置。即使相关的函数没有执行，也会使用内存。然而，由于空间分配已经在程序最初加载到内存时完成，因此不需要额外的运行时开销来分配空间。

12.4 函数声明

一个函数必须先编写，才能使用。函数位于代码段中。其一般格式为:

```
        全局 <procName>procName>:
<
            ; 函数体
        ret
```

一个函数只能定义一次。定义函数的顺序没有特定要求。然而，函数不能嵌套。函数定义应在下一个函数定义开始之前开始和结束。

参考示例函数以了解函数声明和使用的示例。

12.5 标准调用约定

编写汇编程序时，需要一个标准的过程来在函数之间传递参数、返回值以及分配寄存器。如果每个函数都以不同的方式执行这些操作，事情会很快变得非常混乱，并要求程序员尝试记住每个函数如何处理参数以及使用了哪些寄存器。为了解决这个问题，定义并使用了一个标准的过程，通常被称为 *standard calling convention*[40]。实际上存在许多不同的标准调用约定。本文件的其余部分描述了64位C调用约定，称为System V AMD64 ABI[41] [42]。

此调用约定默认也用于C/C++程序。这意味着由于使用相同的调用约定，因此轻松实现汇编语言代码和C/C++代码的接口。

必须注意，这里提出的标准调用约定适用于基于Linux的操作系统。Microsoft Windows的标准调用约定略有不同，且未在本文本中介绍。

40 For more information, refer to: http://en.wikipedia.org/wiki/Calling_convention
41 For more information, refer to: https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI
42 For complete details, refer to: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

12.6 链接

链接是关于正确进入和返回函数调用的过程。有两个指令处理链接，即 call **<**funcName**>** 和 ret 指令。

调用将控制权传递给指定的函数，而 ret 将控制权返回给调用例程。

- 调用通过保存函数完成时返回的地址（称为 *return address*）来实现。这是通过将rip寄存器的内容压入栈中完成的。回想一下，rip寄存器指向要执行的下一个指令（即调用之后的指令）。

- ret指令用于过程返回。ret指令将栈（rsp）当前顶部弹出至rip寄存器。因此，适当的返回地址被恢复。

由于堆栈用于支持链接，因此在函数内部必须确保堆栈不被破坏。具体来说，任何推入的项都必须弹出。推入一个值而不弹出会导致该值从堆栈中弹出并放置在rip寄存器中。这会导致处理器尝试在该位置执行代码。很可能会因为无效位置而导致进程崩溃。

函数调用或链接指令总结如下:

Instruction	**Explanation**
`call <funcName>`	Calls a function. Push the 64-bit **rip** register and jump to the <***funcName***>.
Examples:	**`call printString`**
`ret`	Return from a function. Pop the stack into the **rip** register, effecting a jump to the line after the call.
Examples:	**`ret`**

更多　完整指令列表位于Appen　　10 B.

12.7 参数传输

参数传递是指将信息（变量等）发送到函数中，并获取适合特定函数的结果。

函数传递值的标准术语称为 *call-by- value.* 函数传递地址的标准术语称为 *call-by-reference.* 这应该是一个高级语言中熟悉的话题。

有多种方法可以将参数传递给函数或从函数中获取。

- 将值放入寄存器 ◦ 最简单，但有局限性（例如，寄存器的数量）。◦ 用于前六个整数参数。◦ 用于系统调用。

- 全局定义的变量 ◦ 通常是不良实践，可能令人困惑，并且在许多情况下无法工作。◦ 在有限情况下偶尔有用。

- 将值和/或地址放入堆栈 ◦ 没有对可传递参数数量的具体限制。◦ 承担更高的运行时开销。

通常，调用例程被称为 ***caller***，而被调用例程被称为 ***callee***。

12.8 调用约定

函数 *prologue* 是函数开头的代码，函数 *epilogue* 是函数结尾的代码。序言和尾言执行的操作通常由标准调用约定指定，并处理栈、寄存器、传递的参数（如果有），以及栈动态局部变量（如果有）。

一般思路是保存程序状态（即特定寄存器和栈的内容），执行函数，然后恢复状态。当然，函数通常会大量使用寄存器和栈。序言代码有助于保存状态，而尾言代码则恢复状态。

12.8.1 参数传递

如所述，寄存器和堆栈的组合用于向和/或从函数传递参数。

前六个整数参数如下通过寄存器传递：

Argument Number	Argument Size			
	64-bits	32-bits	16-bits	8-bits
1	rdi	edi	di	dil
2	rsi	esi	si	sil
3	rdx	edx	dx	dl
4	rcx	ecx	cx	cl
5	r8	r8d	r8w	r8b
6	r9	r9d	r9w	r9b

第七个和任何额外的参数都通过堆栈传递。标准调用约定要求，在堆栈上传递参数（值或地址）时，参数应按相反顺序压入。也就是说，“someFunc(one, two, three, four, five, six, seven, eight, nine)”将意味着压入顺序为：*nine*，*eight*，然后是*seven*。

对于浮点参数，首先八个浮点参数按照顺序使用浮点寄存器 xmm0 到 xmm7。

此外，当函数完成时，调用例程负责从堆栈中清除参数。而不是执行一系列的弹出指令，根据需要调整堆栈指针 rsp 以清除堆栈上的参数。由于每个参数是 8 个字节，调整将是将 [(参数数量) * 8] 添加到 rsp。

对于返回值的函数，结果根据返回值的大小放置在A寄存器中。

具体来说，值返回如下：

Return Value Size	Location
byte	**al**
word	**ax**
double-word	**eax**
quadword	**rax**
floating-point	**xmm0**

寄存器rax可以在函数中根据需要使用，只要在返回之前适当地设置返回值。

12.8.2 注册使用

标准调用约定指定了在函数调用时寄存器的使用。具体来说，一些寄存器在函数调用期间应保持不变。这意味着如果某个值被放置在 *preserved register* 或 *saved register* (，则必须由 *callee* 或被调用函数) 保存，并且函数必须使用该寄存器，则原始值必须通过将其放置在堆栈上、按需更改，并在返回调用例程之前恢复到其原始值来保留。这种寄存器保留通常在序言中执行，而恢复通常在尾声中执行。

以下表格总结了寄存器使用情况。

Register	Usage
rax	Return Value
rbx	Callee Saved (by function)
rcx	4th Argument
rdx	3rd Argument
rsi	2nd Argument
rdi	1st Argument
rbp	Callee Saved (by function)
rsp	Stack Pointer
r8	5th Argument

r9	6th Argument
r10	Temporary
r11	Temporary
r12	Callee Saved (by function)
r13	Callee Saved (by function)
r14	Callee Saved (by function)
r15	Callee Saved (by function)

临时寄存器（rax、r10 和 r11）以及参数寄存器（rdi、rsi、rdx、rcx、r8 和 r9）在函数调用过程中不被保留。这意味着在函数中可以使用这些寄存器，而无需保留原始值。

此外，浮点寄存器在函数调用过程中均不被保留。有关浮点操作的更多信息，请参阅第18章。

12.8.3 调用帧

函数调用中作为栈上元素的项被称为 *call frame*（也称为 *activation record* 或 *stack frame*）。根据标准调用约定，栈上的项（如果有），将以特定的通用格式存在。

调用帧中可能的项目包括:

- 返回地址（必需）。
- 保留寄存器（如有）。
- 已传递的参数（如果有）。
- 堆栈动态局部变量（如果有）。

其他项可能放置在调用帧中，例如动态作用域语言的静态链接。此类主题超出本文范围，此处不予讨论。

对于某些函数，可能不需要完整的调用帧。例如，如果函数:

- 是一个叶函数（即不调用另一个函数）。
- 仅通过寄存器传递其参数（即，不使用栈）。
- 不改变任何已保存的寄存器。
- 不需要基于栈的局部变量。

这可能在更简单、更小的叶子函数中发生。然而，如果这些条件中的任何一个不成立，则需要完整的调用帧。对于更多非叶子或更复杂的函数，需要更完整的调用帧。

标准调用约定不明确要求使用帧指针寄存器rbp。编译器允许优化调用帧而不使用帧指针。为了简化并明确访问基于堆栈的参数（如果有）和堆栈动态局部变量，本文将使用帧指针寄存器。这与许多其他架构使用帧指针寄存器的方式类似。

因此，如果函数中需要任何基于栈的参数或任何局部变量，则应将帧指针寄存器rbp压入并设置为指向自身。随着执行额外的压入和弹出操作（从而改变rsp），rbp寄存器将保持不变。这允许使用rbp寄存器作为引用来访问传递到栈上的参数（如果有）或栈动态局部变量（如果有）。

例如，假设一个函数调用有八个（8）参数，并且假设该函数使用rbx、r12和r13寄存器（因此必须入栈），调用帧将如下所示：

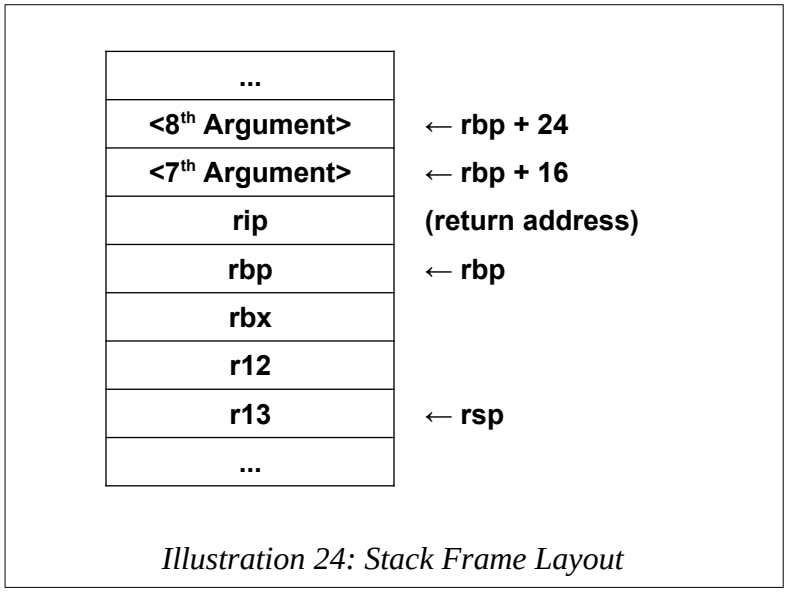


Illustration 24: Stack Frame Layout

基于栈的参数相对于rbp访问。每个压入项是一个四字长度的值，占用8字节。例如，[rbp+16]是第一个传入参数的位置（7th整数参数）和[rbp+24]是第二个传入参数的位置（8th整数参数）。

此外，调用帧将包含局部变量的分配位置（如果有）。关于局部变量的部分详细说明了分配和使用局部变量的具体细节。

12.8.3.1 红区

在Linux标准调用约定中，栈指针rsp之后的第一个128字节被保留。例如，在扩展前面的示例中，调用帧将如下所示:

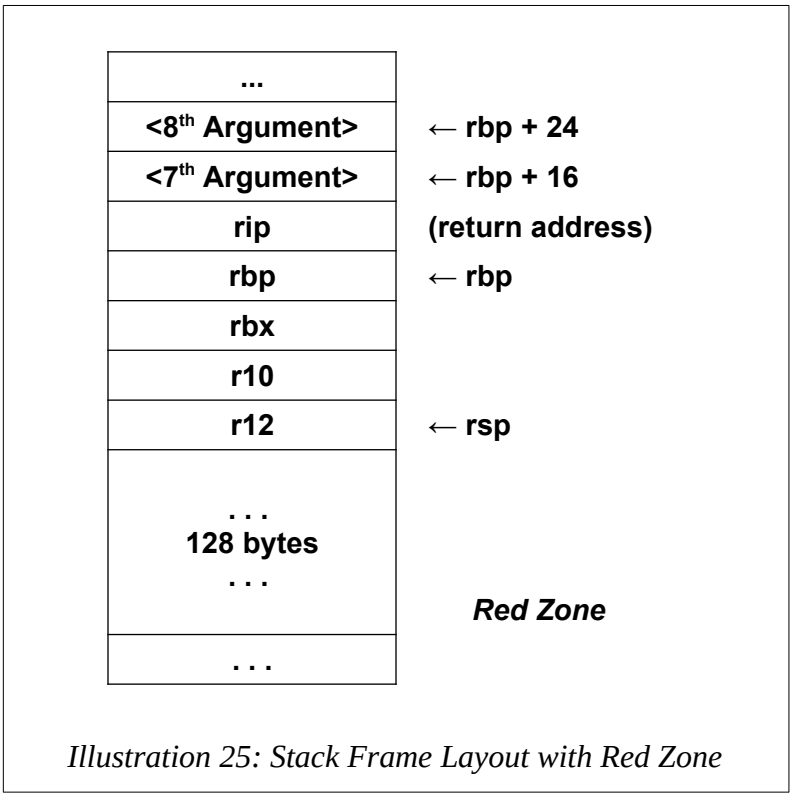


Illustration 25: Stack Frame Layout with Red Zone

此红色区域可由函数使用，无需调整栈指针。其目的是允许编译器对局部变量的分配进行优化。这不会直接影响直接用汇编语言编写的程序。

12.9 示例，统计函数 1（叶子）

这是一个简单的例子，将演示调用一个简单的void函数来找到总和和

数组数字的平均值。C/C++ 的高级语言（HLL）调用如下：

```
stats1(arr, len, &sum, &ave);
```

根据C/C++约定，数组***arr***是按引用调用，长度***len***是按值调用。***sum***和***ave***的参数都是按引用调用（因为还没有值）。对于这个例子，数组***arr***、***sum***和***ave***变量都是有符号双字整数。当然，在上下文中，***len***必须是无符号的。

12.9.1 呼叫者

在这个情况下，有4个参数，并且所有参数都按照标准调用约定传递到寄存器中。调用stats函数的调用例程中的汇编语言代码如下：

```
; stats1(arr, len, &sum, &ave); mov rcx, ave
                                        ; 4th 参数，ave 地址
mov rdx, {v*}                           ; 3rd 参数，sum 的地址
mov esi, dword [len]                    ; 2nd arg，len的值
mov rdi, arr                            ; 1st arg, arr的地址
调用 stats1
```

没有设置参数寄存器的特定顺序要求。此示例以逆序设置它们，为下一个扩展示例做准备。

Note, esi寄存器的设置也将高阶双字设置为0，从而确保rsi寄存器对于这种特定用法被适当地设置，因为长度是无符号的。

此空操作例程不提供返回值。如果该函数是一个返回值的函数，返回值将在A寄存器（适当大小）中。

12.9.2 被调用者

被调用的函数，即被调用者，必须在执行函数目标代码之前和之后执行前导和尾随操作（如标准调用约定所指定）。对于此示例，函数必须执行数组中值的求和，计算整数平均值，并返回总和和平均值。

以下代码实现了stats1示例。

```
; 简单示例函数用于查找并返回；数组的总和和平均值。
```

```
;  HLL 调用：  ; stats1(arr, len, sum, ave); ; ---
-- ;  参数：  ; arr, 地址 – rdi ; len, dword 值 –
esi ; sum, 地址 – rdx ; ave, 地址 - rcx
```

```
全局统计1 统计1：  push r12 ; prologue mov r12, 0 ; counter/index mov rax, 0 ;
运行总和 sumLoop: add eax, dword [rdi+r12*4] ; 总和 += arr[i] inc r12 cmp r
12, rsi jl sumLoop mov dword [rdx], eax ; 返回总和 cdq idiv esi ; 计算平均值
mov dword [rcx], eax ; 返回平均值 pop r12 ; epilogue ret
```

寄存器r12的选择是任意的，然而选择了一个“已保存的寄存器”。

此函数的调用帧如下 s:

最小化使用栈有助于减少函数调用运行时开销。

12.10 示例，统计函数2（非叶节点）

这个扩展示例将演示如何调用一个简单的void函数来找到数字数组的最大值、中位数、最小值、总和和平均值。

高级语言（HLL）对C/C++的调用如下：

```
stats2(arr, len, &min, &med1, &med2, &max, &sum, &ave);
```

对于这个例子，假设数组是按升序排序的。此外，对于这个例子，中位数将是中间值。对于偶数长度的列表，有两个中间值，***med1***和***med2***，两者都返回。对于奇数长度的列表，单个中间值在***med1***和***med2***中返回。

根据C/C++约定，数组***arr***是按引用调用，长度***len***是按值调用。***min***、***med1***、***med2***、***max***、***sum***和***ave***的参数都是按引用调用（因为还没有值）。对于这个例子，数组***arr***、***min***、***med1***、***med2***、***max***、***sum***和***ave***变量都是有符号双字整数。当然，在这种情况下，***len***必须是无符号的。

12.10.1 呼叫者

在这种情况下，有8个参数，只有前六个可以传入寄存器。最后两个参数通过栈传递。调用stats函数的调用例程中的汇编语言代码如下：

```
; stats2(arr, len, &min, &med1, &med2, &max, &sum, &ave);
push    ave                             ; 8th arg, add of ave
push    sum                             ; 7th arg, add of sum
mov     r9, max                         ; 6th arg, add of max
mov     r8, med2                        ; 5th arg, add of med2
mov     rcx, med1                       ; 4th arg, add of med1
mov     rdx, min                        ; 3rd arg, addr of min
mov     esi, dword [len]                ; 2nd arg, value of len
mov     rdi, arr                        ; 1st arg, addr of arr
call    stats2
add     rsp, 16                         ; clear passed arguments
```

7^{th} 和 8^{th} 参数按标准调用约定在栈上传递，并按相反顺序压入。函数完成后，通过调整栈指针寄存器（rsp）清除栈上的参数。由于在栈上传递了两个参数，每个参数8字节，因此栈指针增加了16。

Note, esi寄存器的设置也将高阶双字设置为0，从而确保rsi寄存器对于这种特定用法被适当地设置，因为长度是无符号的。

此空操作例程不提供返回值。如果该函数是一个返回值的函数，返回值将在A寄存器中。

12.10.2 被调用者

被调用的函数，即被调用者，必须执行序言和尾言操作（如标准调用约定所指定）。当然，函数必须执行数组中值的求和、找到最小值、中位数和最大值、计算平均值，并返回所有值。

当通过引用传递参数到栈上时，需要两个步骤来返回值。

- 从堆栈中获取地址。
- 使用该地址返回值。

一个常见的错误是在单步中尝试将值返回到基于栈的位置，这不会改变引用的变量。例如，假设要返回的双字值在eax寄存器中，7th参数是通过引用传递的，并且eax的值要返回，适当的代码如下：

```
mov r12, qword [rbp+16]mov dword [r
12], eax
```

这些步骤不能合并为单个步骤。以下代码

```
mov dword [rbp+16], eax
```

将覆盖通过堆栈传递的地址，不会更改引用变量。

以下代码实现了stats2示例。

```
; 简单示例函数，用于查找并返回数组的最大值、最小值、总和、中位数和平均值。; ----- ; HLL 调用：; stats2(arr, len, min, med1, med2, max, sum, ave); ; 参数：; arr, 地址 – rdi
```

; len, dword值 – esi ; min, 地址 – rdx ; med1, 地址 - rcx ; med2, 地址 - r8 ; max, 地址 – r9 ; sum, 地址 – 栈 (rbp+16) ; ave, 地址 – 栈 (rbp+24)

全局统计2 统计2:

```
push rbp                                                    ; 前言
    mov rbp, rsp push r12
```

获取最小值和最大值。mov eax, dword [rdi]

```
                                                            ; 获取最小值
    mov dword [rdx], eax        移动双字到[rdx]中，  eax    ; 返回最小值

    mov r12, rsi ; 获取 lendec r12 ; 设置 len-1mov eax, dword [rdi+r12*4]
    ; 获取 maxmov dword [r9], eax ; 返回 max
```

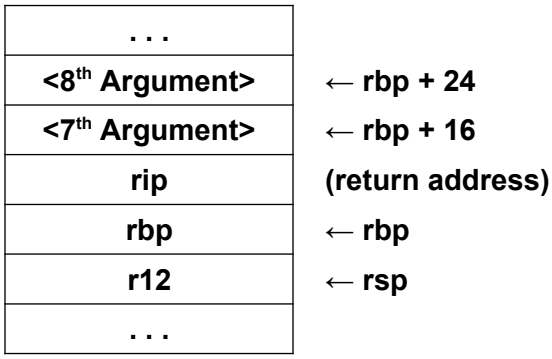
; ----- ; 获取中位数

```
    mov rax, rsimov rdx, 0
    mov r12, 2div r12
                                                            ; rax = 长度/2

    cmp rdx, 0                                              ; 偶数/奇数长度?
    je 偶长 mov r12d, dword [rdi+rax*4]
                                                            ; 获取 arr[len/2]
    mov dword [rcx], r12d                                   ; 返回 med1
    mov dword [r8], r12d                                    ; 返回 med2
    jmp medDone

evenLength: mov r12d, dword [rdi+rax*4]
                                                            ; 获取 arr[len/2]
```

```
mov dword [r8], r12d ; return med2 dec rax mov r12d, dword [rdi+rax*4] ; get arr[
len/2-1] mov dword [rcx], r12d ; return med1 medDone: ; ----- ;  Find sum mov r12, 0 ;
counter/index mov rax, 0 ; running sum sumLoop: add eax, dword [rdi+r12*4] ; sum +=
arr[i] inc r12 cmp r12, rsi jl sumLoop mov r12, qword [rbp+16] ; get sum addr mov dwo
rd [r12], eax ; return sum ; ----- ;  Calculate average. cdq idiv rsi ; average = sum/len mo
v r12, qword [rbp+24] ; get ave addr mov dword [r12], eax ; return ave pop r12 ; epilog
ue pop rbp ret
```

寄存器的选择是任意的，其范围由调用约定决定。

函数的调用帧如下:



在这个示例中，保留的寄存器rbp然后是r12被压入。弹出时，必须按照完全相反的顺序弹出，即先弹出r12然后是rbp，才能正确恢复它们的原始值。

12.11 基于栈的局部变量

如果需要局部变量，它们将在栈上分配。通过调整 rsp 寄存器，为局部变量在栈上分配额外的内存。因此，当函数完成时，用于基于栈的局部变量的内存将被释放（并且不再使用内存）。

进一步扩展前面的例子，如果我们假设所有数组值都在0到99之间，并且我们希望找到众数（出现次数最多的数字），可以使用一个双字变量 ***count*** 和一个包含一百（100）个元素的本地双字数组，***tmpArr[100]***。

与之前一样，帧寄存器rbp被压入栈中并设置为指向自身。帧寄存器加上适当的偏移量可以访问通过栈传递的任何参数。例如，rbp+16是第一个基于栈的参数（7th 整型参数）的位置。

在帧寄存器被推入后，对栈指针寄存器 rsp 进行调整以分配局部变量的空间，本例中为一个 100 个元素的数组。由于计数变量是一个双字，需要 4 个字节。临时数组有 100 个双字元素，需要 400 个字节。因此，总共需要 404 个字节。由于栈在内存中向下增长，从栈指针寄存器中减去 404 个字节。

然后任何已保存的寄存器，例如本例中的rbx和r12，都被推入堆栈。

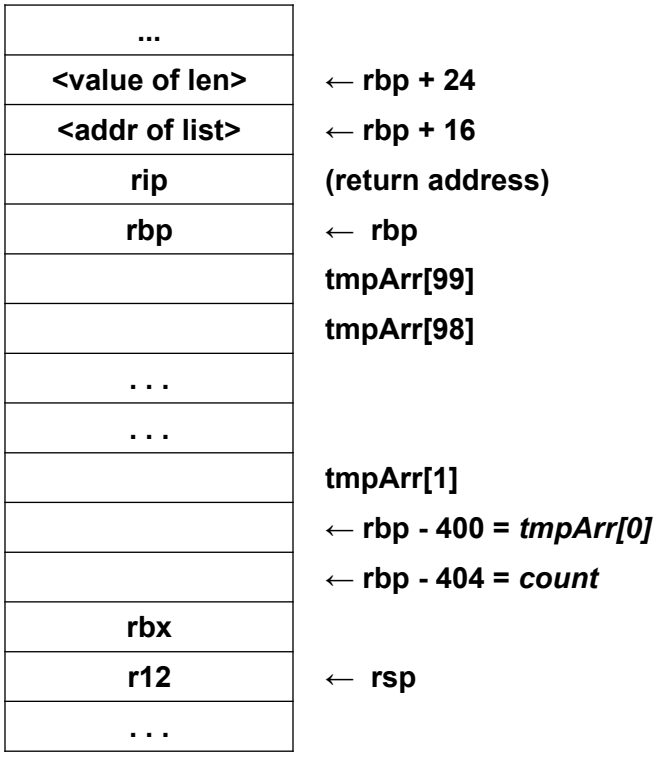
当离开函数时，必须从堆栈中清除保存的寄存器和局部变量。执行此操作的推荐方法是先弹出保存的寄存器，然后将rbp寄存器复制到rsp寄存器中，从而确保rsp寄存器指向堆栈上的正确位置。

```
mov rsp, rbp
```

这通常比将偏移量重新添加到堆栈中要好，因为分配的空间可以根据需要更改，而不需要调整结尾代码。

应清楚，以这种方式分配的变量未初始化。如果函数需要初始化这些变量，可能初始化为0，则必须显式执行这些初始化。

对于这个例子，调用帧的格式如下:



布局和分配的404字节内局部变量的顺序是任意的。

例如，此扩展示例的更新序言代码将是:

```
push rbp                                                    ; 前言
mov rbp, rspsub rsp, 4
04                                                          ; 分配局部变量
push rbx push r
12
```

本地变量可以通过相对于帧指针寄存器rbp进行访问。例如，为了初始化现在分配给rbp-404的计数变量，可以使用以下指令:

```
mov dword [rbp-404], 0
```

要访问 ***tmpArr***，必须获取起始地址，这可以通过 lea 指令来完成。例如，

```
lea rbx, dword [rbp-400]
```

这将设置rbx寄存器中适当的堆栈地址，其中rbx被任意选择。在这个例子中，dword限定符不是必需的，并且可能是误导性的，因为地址始终是64位（在64位架构上）。一旦按上述方式设置，rbx中的***tmpArr***起始地址就按常规方式使用。

例如，一个展示访问基于栈的局部变量的简短不完整函数代码片段如下:

```
; ---------------------------------------- ; 示例全局函数 global  expFunc expFunc: push
rbp ; prologue mov rbp, rsp sub rsp, 404 ; 分配局部变量 push rbx push r12 ; ----- ;
将计数局部变量初始化为0。 mov dword [rbp-404], 0 ; -----
```

; 增加计数变量（例如）...

inc dword [rbp-404] ; count++ ; ----- ; 循环以将tmpArr初始化为所有0。lea rbx, dword [rbp-400] ; tmpArr地址mov r12, 0 ; 索引置零Loop: mov dword [rbx+r12*4], 0 ; tmpArr[index]=0 inc r12 cmp r12, 100 jl zeroLoop ; ----- ; 完成后，恢复所有并返回调用例程。pop r12 ; 尾部pop rbx mov rsp, rbp ; 清除局部变量pop rbp ret

Note, 此 示例 函数 仅 关注 如何 访问 基于栈的 局部 变量 ， 并 不 执行 任何 有用 的 操作。

12.12 摘要

本节简要介绍了标准调用约定要求，具体如下：

呼叫者操作：

- 前六个整数参数通过寄存器 {v*} rdi、rsi、rdx、rcx、r8、r9 传递
- 7th和后续参数通过基于堆栈的◦传递。以相反的顺序（从右到左）将参数推送到堆栈上（因此，在函数调用中指定的第一个堆栈参数最后被推入）。◦推入的参数作为四倍字传递。

- 调用者执行调用指令以将控制权传递给函数（被调用者）。
- 栈基参数从栈中清除。◦ add rsp, <argCount* 8>

调用者操作：

- 函数序言 ◦ 如果参数通过栈传递，被调用者必须将 rbp 保存到栈上并将 rsp 的值移动到 rbp。这允许被调用者使用 rbp 作为帧指针以统一方式访问栈上的参数。▪ 然后，被调用者可以相对于 rbp 访问其参数。在 [rbp] 处的64位字包含 rbp 的上一个值，它是在推送时保存的；下一个64位字，在 [rbp+8] 处，包含由 **`call`** 推送的返回地址。参数从那里开始，在 [rbp+16] 处。◦ 如果需要局部变量，被调用者应进一步减少 rsp 以在栈上为局部变量分配空间。局部变量可以通过从 rbp 的负偏移量访问。◦ 如果被调用者希望向调用者返回一个值，它应该在 al、ax、eax、rax 中留下该值，具体取决于返回值的尺寸。▪ 浮点结果返回在 xmm0 中。◦ 如果改变了 rbx、r12、r13、r14、r15 和 rbp 寄存器，则必须在栈上保存。

- 函数执行 ◦ 函数代码被执行。

- 函数尾声

  ◦ 恢复所有推送的寄存器。◦ 如果使用了局部变量，被调用者将从 rbp 恢复 rsp 以清除基于堆栈的局部变量。◦ 被调用者恢复（即弹出）rbp 的上一个值。◦ 调用通过 ret 指令返回（返回）。

参考示例函数以查看调用约定的具体示例。

12.13 练习

以下是本章的一些测验问题和基于此章节的建议项目。

12.13.1 测验问题

以下是本章的一些测验问题。

1) 函数调用的两个主要动作是什么？2) 实现v1的两个指令是什么？3) 当使用v2传递参数时，它被称为什么？4) 当使用v3传递参数时，它被称为什么？5) 如果一个函数被调用十五（15）次，汇编器将代码放入内存多少次？6) 执行v4指令时会发生什么（两件事）？7) 根据课堂上讨论的标准调用约定，大多数过程中的初始推送和最终弹出有什么目的？8) 如果有六个（6）64位整数参数传递给一个函数，每个参数应该具体传递在哪里？9) 如果有六个（6）32位整数参数传递给一个函数，每个参数应该具体传递在哪里？10) 当一个寄存器被指定为临时寄存器时，这意味着什么？11) 列出两个临时寄存器。12) 函数调用过程中放置在栈上的项目集合被称为什么？13) 当一个函数被称为v5时，这意味着什么？14) 在call语句之后，add rsp, v6immediatev7的作用是什么？15) 如果v8个参数在栈上传递，v9immediatev10的值是多少？16) 如果有七个（7）参数传递给一个函数，并且函数本身按顺序推送rbp、rbx和r12寄存器，使用标准调用约定时基于栈的参数的正确偏移量是多少？

17) 一个函数可以被调用多少次，有什么限制因素吗?

18) 如果一个函数必须为变量 ***sum*** 返回一个结果，应该如何传递（按引用或按值）变量 ***sum*** ？

19) 如果向一个函数传递了八个（8）参数，并且该函数按顺序将rbp、rbx和r12寄存器压入栈中，使用标准调用约定时，两个基于栈的参数（7th和8th）的正确偏移量是多少?

20) 使用堆栈动态局部变量（而不是使用所有全局变量）的优势是什么?

12.13.2 建议项目

以下是本章的一些建议项目。

1) 创建主程序并实现 ***stats1*** 示例函数。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。

2) 创建主程序并实现 ***stats2*** 示例函数。使用调试器执行程序并显示最终结果。创建调试器输入文件以显示结果。

3) 创建一个主程序和一个函数，该函数将数字列表按升序排序。使用以下选择[43]排序算法:

```
beginfor  i = 0 to len-1 small = arr(i) index = i for  j = i
to len-1 if ( arr(j) < small ) then small = arr(j) index = j
end_if end_for arr(index) = arr(i) arr(i) = small end_fo
r end_begin
```

43 For more information, refer to: http://en.wikipedia.org/wiki/Selection_sort

主函数应在至少三个不同的数据集上调用该函数。 使用调试器执行程序并显示最终结果。 创建调试器输入文件以显示结果。

4) 将程序从上一个问题更新，添加一个统计函数，用于找到排序列表的最小值、中位数、最大值、总和和平均值。统计函数应在排序函数之后调用，以便更容易找到最小值和最大值。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

5) 将上一个问题中的程序更新，添加一个整数平方根函数和一个标准差函数。为了估计一个数的平方根，使用以下算法:

$$iSqrt_{est} = iNumber$$

$$iSqrt_{est} = \frac{\left(\frac{iNumber}{iSqrt_{est}}\right) + iSqrt_{est}}{2} \qquad \text{迭代50次}$$

标准差公式如下:

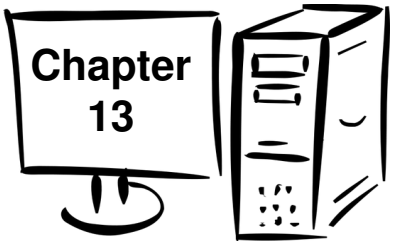
$$iStandardDeviation = \sqrt{\frac{\sum_{i=0}^{length-1} (list[i] - average)^2}{length}}$$

Note, 执行 求和 和 除法 使用 整数 值。 使用 调试器 执行 程序 并 显示 最终 结果。 创建 调试器 输入 文件 以 显示 结果。

6) 将前一章中的整数转换为ASCII宏的函数转换为void函数。该函数应将一个有符号整数转换为给定长度的右对齐字符串。这需要包括任何前导空格、符号（“+”或“-”）、数字和NULL。该函数应接受整数的值、放置NULL终止字符串的地址以及最大字符串长度的值，并按此顺序接受。开发一个主程序来调用一系列不同整数的函数。主程序应包括适当的数据声明。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

7) 创建一个函数，将表示数字的ASCII字符串转换为整数。该函数应读取字符串并执行适当的错误检查。如果发生错误，函数应返回FALSE（一个定义为0的常量）。如果字符串有效，函数应将字符串转换为整数。如果转换成功，函数应返回TRUE（一个定义为1的常量）。开发一个主程序来对一系列不同的整数调用该函数。主程序应包括适当的数据声明和适用的常量。使用调试器执行程序并显示最终结果。创建一个调试器输入文件以显示结果。

Linux is basically a simple operating system, but you have to be a genius to understand the simplicity.



13.0 系统服务

有许多操作是应用程序必须使用操作系统来执行的。这些操作包括控制台输出、键盘输入、文件服务（打开、读取、写入、关闭等）、获取时间或日期、请求内存分配，以及许多其他操作。

访问系统服务是应用程序请求操作系统执行某些特定操作（代表进程）的方式。更具体地说，*system call*是执行进程和操作系统之间的接口。

本节提供了如何使用一些基本系统服务调用的说明。有关其他系统服务调用的更多信息，请参阅附录C，系统服务调用。

13.1 调用系统服务

系统服务调用在逻辑上类似于调用函数，其中函数代码位于操作系统中。该函数可能需要操作权限，这就是为什么控制必须转移到操作系统的原因。

在调用系统服务时，参数放置在标准参数寄存器中。系统服务通常不使用基于堆栈的参数。这限制了系统服务的参数数量为六个（6），这并不构成显著限制。

要调用系统服务，第一步是确定需要哪个系统服务。有许多系统服务（见附录C）。一般过程是将系统服务调用代码放置在rax寄存器中。调用代码是一个为请求的特定系统服务分配的数字。这些是在操作系统部分分配的，不能由应用程序程序更改。为了简化过程，本文将定义一组系统服务调用代码的非常小的子集为常量集。对于本文和相关的示例，子集为

系统调用代码常量在源文件中定义并显示，以帮助为新汇编语言程序员提供完整的清晰度。对于更有经验的程序员，通常在开发较大或更复杂的程序时，常量列表在一个文件中，并包含在源文件中。

如果需要，系统服务的参数放置在 rdi、rsi、rdx、rcx、r8 和 r9 寄存器中（按此顺序）。以下表格显示了与标准调用约定一致的参数位置。

Register	Usage
rax	Call code (see table)
rdi	1st argument (if needed)
rsi	2nd argument (if needed)
rdx	3rd argument (if needed)
rcx	4th argument (if needed)
r8	5th argument (if needed)
r9	6th argument (if needed)

这与标准函数调用约定非常相似，然而，如果需要，4th 参数使用r10寄存器。

每个系统调用将使用不同数量的参数（从无到最多6个）。然而，系统服务调用代码始终是必需的。

在设置好调用代码和任何参数后，执行 syscall 指令。syscall 指令将暂停当前进程并将控制权转移到操作系统，操作系统将尝试执行 rax 寄存器中指定的服务。当系统服务返回时，进程将继续执行。

13.2 新行字符

作为复习，在输出的上下文中，换行符意味着将光标移动到下一行的开头。在包括C在内的许多语言中，它通常被记作字符串的一部分“\n”。C++在cout语句的上下文中使用endl。例如，“Hello World 1”和“Hello\nWorld 2”将显示如下:

```
你好 世界 1 你好
世界 2
```

无内容显示换行，但光标移动到下一行的开头，如图所示。

在 Unix/Linux 系统中，换行符，缩写为 LF，其 ASCII 值为 10（或 0x0A），用作换行字符。在 Windows 系统中，换行符是回车符，缩写为 CR，其 ASCII 值为 13（或 0x0D），后跟 LF。文本中的代码示例使用 LF。

读者可能见过从网页下载文本文件并使用较旧的Windows记事本（Windows 10之前）显示的实例，所有格式都丢失了，看起来像是一行非常长的文本。这通常是由于Unix/Linux格式的文件，它只使用LF，在期望CR/LF对的Windows实用程序中显示时，当只找到LF时无法正确显示。其他Windows软件，如记事本++（开源文本编辑器）将识别并处理不同的换行格式，并正确显示。

13.3 控制台输出

系统服务将字符输出到控制台的是系统写入（SYS_write）。与高级语言中的字符一样，它们被写入标准输出（STDOUT），即控制台。STDOUT 是控制台默认的文件描述符。文件描述符已经打开并可供程序（汇编语言和高级语言）使用。

The arguments for the write system service are as follows: 参数如下:

Register	SYS_write
rax	Call code = SYS_write (1)
rdi	Output location, STDOUT (1)
rsi	Address of characters to output
rdx	Number of characters to output

假设以下声明:

```
STDOUT      equ     1               ; standard output
SYS_write   equ     1               ; call code for write

msg         db      "Hello World"
msgLen      dq      11
```

例如，要将“Hello World”（这是传统的）输出到控制台，系统会使用write（SYS_write）函数。代码如下：

```
mov rax, SYS_writemov rdi, STDOUTmov rsi, msg ; msg地址mov r
dx, qword [msgLen] ; 长度值syscall
```

参考下一节以获取显示上述消息的完整程序。应注意，操作系统不会检查字符串是否有效。

13.3.1 示例，控制台输出

这是一个完整的程序，用于向控制台输出一些字符串。在这个例子中，一个字符串包含换行符，而另一个则不包含。

```
; Example program to demonstrate console output.
; This example will send some messages to the screen.

; ********************************************

section    .data

; -----
;  Define standard constants.

LF              equ     10              ; line feed
NULL            equ     0               ; end of string
TRUE            equ     1
FALSE           equ     0

EXIT_SUCCESS    equ     0               ; success code

STDIN           equ     0               ; standard input
STDOUT          equ     1               ; standard output
STDERR          equ     2               ; standard error

SYS_read        equ     0               ; read
SYS_write       equ     1               ; write
SYS_open        equ     2               ; file open
SYS_close       equ     3               ; file close
SYS_fork        equ     57              ; fork
```

SYS_exit equ 60 ; 终止SYS_creat equ 85 ; 文件打开/创建SYS_time equ 201 ;
获取时间

; ----- ; 定义一些字符串。

message1 db "Hello World.", LF, NULLmessage2 db "Enter Answe
r: ", NULLnewLine db LF, NULL

;----------

章节 .text 全局 _start _
start:

; ----- ; 显示第一条消息。

mov rdi, message1 call printSt
ringTranslated Text: 将 rdi 寄
存器设置为 message1 调用 p
rintString 函数

; ----- ; 显示第二条消息并换行mov rdi, message2 call printStr
ingmov rdi, newLine call printString

; ----- ; 示例程序完成。

exampleDone: mov rax, SYS_exit mov rd
i, EXIT_SUCCESS syscall

; ********** ; 通用函数,
用于在屏幕上显示字符串。 ;
字符串必须以NULL终止。

; 算法: ; 计算字符串中的字符数（不包括NULL）; 使用系统调用输出字符

; 参数: ; 1) 地址, 字符串 ; 返回值: ; 无

全局 打印字符串 打印字符串: 压入 rbx

; ----- ; 计算字符串中的字符数。

```
    mov rbx, rdimov rdx,
    0
strCountLoop: cmp byte [rbx], NULL je
strCountDone inc rdx inc rbx jmp strCo
untLoop

strCountDone:

    cmp rdx, 0 je prtDon
    e
```

; ----- ; 调用操作系统输出字符串mov rax, SYS_write ; write()的系统代码mov rsi, rdi ; 要写入的字符地址mov rdi, STDOUT ; 标准输出; RDX=要写入的计数, 上面已设置 系统调用

字符串已打印, 返回调用例程。

prtDone:

```
pop rbx ret
```

输出结果如下:

```
你好，世界。输入
答案: _
```

换行符（LF）作为第一个字符串（***message1***）的一部分提供，因此将光标放置在下一行的开头。第二条消息将光标留在同一行，这对于从用户读取输入（本例中不包括）是合适的。由于本例中没有获取实际输入，因此打印了一个最后的换行符。

额外的、未使用的常量包含为参考。

13.4 控制台输入

系统服务从控制台读取字符的是系统读取（SYS_read）。类似于高级语言，对于控制台，字符是从标准输入（STDIN）读取的。STDIN是从键盘读取字符的默认文件描述符。文件描述符已经打开并可用于程序（汇编语言和高级语言）中使用。

从键盘交互式读取字符会带来额外的复杂性。当使用系统服务从键盘读取时，就像写入系统服务一样，需要指定要读取的字符数。当然，我们需要声明适当的空间来存储正在读取的字符。如果我们请求读取10个字符，而用户输入超过10个，额外的字符将会丢失，这不是一个重大问题。如果用户输入少于10个字符，例如5个字符，将读取所有五个字符以及换行符（LF），总共六个字符。

如果从文件重定向输入，则会出现问题。如果我们请求10个字符，并且第一行有5个字符，第二行有更多字符，我们将从第一行获得六个字符（5个字符加上换行符）和下一行的前四个字符，总共10个字符。这是不希望的。

为了解决这个问题，在交互式读取输入时，我们将逐个字符读取，直到读取到换行符（回车键）。每个字符将被读取并依次存储在一个适当大小的数组中。

The arguments for the read system service are as follows:

Register	SYS_read
rax	Call code = SYS_read (0)
rdi	Input location, STDIN (0)
rsi	Address of where to store characters read
rdx	Number of characters to read

假设以下声明:

```
STDIN       equ     0               ; standard input
SYS_read    equ     0               ; call code for read

inChar      db      0
```

例如，要从键盘读取单个字符，将使用系统读取（SYS_read）。代码如下:

```
mov rax, SYS_readmov rdi, S
TDINmov rsi, inChar
                                    ; 消息地址
mov rdx, 1                          ; 读取计数
系统调用
```

参考下一节以获取从键盘读取字符的完整程序。

13.4.1 示例，控制台输入

示例是一个完整的程序，用于从键盘读取50个字符的行。由于包括换行符（LF）和最后的空终止符，需要一个允许52个字节的输入数组。

此示例将读取用户输入的前50个字符，然后将输入回显到控制台以验证输入是否正确读取。

```
; 示例程序以演示控制台输出。; 此示例将发送一些消息到屏幕。; ******
**************************************

节数 .data
```

```
; -----
;  Define standard constants.

LF              equ     10              ; line feed
NULL            equ     0               ; end of string

TRUE            equ     1
FALSE           equ     0

EXIT_SUCCESS    equ     0               ; success code

STDIN           equ     0               ; standard input
STDOUT          equ     1               ; standard output
STDERR          equ     2               ; standard error

SYS_read        equ     0               ; read
SYS_write       equ     1               ; write
SYS_open        equ     2               ; file open
SYS_close       equ     3               ; file close
SYS_fork        equ     57              ; fork
SYS_exit        equ     60              ; terminate
SYS_creat       equ     85              ; file open/create
SYS_time        equ     201             ; get time

; -----
;  Define some strings.

STRLEN          equ     50

pmpt            db      "Enter Text: ", NULL
newLine         db      LF, NULL

section   .bss
chr             resb    1
inLine          resb    STRLEN+2        ; total of 52

;-----------------------------------------------------
```

章节 .text 全局 _start _

start:

; ----- ; 显示提示。

```
    mov rdi, pmpt call printStri
    ng
```

从用户读取字符（逐个读取）

mov rbx, inLine ; inLine addrmov r12, 0 ; 字符计数readCharacters: mov rax, SYS_rea d ; 系统代码用于读取mov rdi, STDIN ; 标准输入lea rsi, byte [chr] ; chr的地址mov rdx, 1 ; 计数（读取多少个）syscall ; 执行系统调用mov al, byte [chr] ; 获取刚读取的字符cmp al, LF ; 如果是换行符， 输入完成je readDoneinc r12 ; 计数++cmp r12, STRLEN ; 如果字符数≥等于STRLENjae readCharacters ; 停止放置在缓冲区中mov byte [rbx], al ; inLine[i] = chrinc rbx ; 更新tmpStr地址jmp readCharactersreadDone: mov byte [rbx], NULL ; 添加空终止符

; ----- ; 输出该行以验证成功读取

```
    mov rdi, inLine call 打印
    字符串
```

; ----- ; 示例完成。

exampleDone: mov rax, SYS_exit mov rd
i, EXIT_SUCCESS syscall

; ********** ; 显示字符串到屏幕的通用过程。 ; 字符串必须以NULL结尾。 ; 算法： ； 计算字符串中的字符数（不包括NULL）； 使用系统调用输出字符 ; 参数： ; 1) 地址，字符串 ; 返回值： ; 没有内容

全局 打印字符串 打印字
符串：压入 rbx

; ----- ; 计算字符串中的字符数。

mov rbx, rdimov rdx,
0

strCountLoop: cmp byte [rbx], NULL je
strCountDone inc rdx inc rbx jmp strCo
untLoop

strCountDone:

cmp rdx, 0 je prtDon
e

调用操作系统输出字符串。

mov rax, SYS_write ; 系统代码用于write() mov rsi, rdi ; 要写入的字符地址

```
    mov rdi, STDOUT ; 标准输出 ; RDX=要写入的计数，设置在系统调用之前 ; 系统调用
```

字符串已打印，返回调用例程。

```
prtDone:
    pop rbx ret
```

如果我们在50（STRLEN）个字符处完全停止读取，并且用户输入了更多字符，这些字符可能会在后续的读取操作中引起输入错误。为了处理用户可能输入的额外字符，这些字符将从键盘读取但不放入输入缓冲区（上方的***inLine***）。这确保了额外输入被从输入流中移除，但不会溢出数组。

额外的、未使用的常量包含为参考。

13.5 文件打开操作

为了执行读取和写入等文件操作，必须首先打开文件。 有两种文件打开操作，即打开和打开/创建。 以下各节将解释这两种打开操作中的每一个。

文件打开后，为了执行文件读写操作，操作系统需要关于文件的详细信息，包括完整状态和当前读写位置。这是确保读写操作从上次停止的地方继续进行的必要条件。

如果文件打开操作失败，将返回一个错误代码。如果文件打开操作成功，将返回一个文件描述符。这适用于高级语言和汇编代码。

文件描述符由操作系统用于访问有关文件的完整信息。关于打开文件的完整信息集存储在名为文件控制块（FCB）的操作系统数据结构中。本质上，文件描述符由操作系统用于引用正确的FCB。确保文件描述符被正确存储和使用是程序员的职责。

13.5.1 文件打开

文件打开需要文件存在才能打开。如果文件不存在，则是一个错误。

文件打开操作也需要参数标志来指定访问模式。访问模式必须包含以下之一:

- 只读访问 → O_RDONLY
- 只写访问 → O_WRONLY
- 读取/写入访问 → O_RDWR

必须使用这些访问模式之一。可以通过与这些模式之一进行“或”操作来使用其他附加访问模式。这可能包括追加模式（在本文本中未涉及）。有关文件访问模式的更多信息，请参阅附录C，系统服务。

文件打开系统服务的参数如下:

Register	SYS_open
rax	Call code = SYS_open (2)
rdi	Address of NULL terminated file name string
rsi	File access mode flag

假设以下声明:

```
SYS_open        equ     2           ; file open

O_RDONLY        equ     000000q     ; read only
O_WRONLY        equ     000001q     ; write only
O_RDWR          equ     000002q     ; read and write
```

应注意的是，常量是以八进制或基8（如使用q后缀指定）定义的。这与Linux文件权限有时指定的方式相匹配。

系统调用后，rax寄存器将包含返回值。如果文件打开操作失败，rax将包含一个负值（即< 0）。具体的负值提供了遇到错误类型的指示。有关错误代码的更多信息，请参阅附录C，系统服务。典型错误可能包括无效的文件描述符、文件未找到或文件权限错误。

如果文件打开操作成功，rax包含文件描述符。文件描述符将用于后续的文件操作，并应保存。

参考“示例文件读取”部分以获取一个完整示例，该示例打开一个文件。

13.5.2 文件打开/创建

一个文件打开/创建操作将创建一个文件。如果文件不存在，将创建一个新文件。如果文件已存在，它将被删除并创建一个新文件。因此，文件的前内容将丢失。

文件访问模式必须指定。由于文件正在创建，访问模式必须包括在文件创建时设置的文件权限。这包括指定对*user*、*group*或*world* 的读取、写入和/或执行权限，这与Linux文件权限的典型做法相同。本例中仅涉及文件的所有者或用户的权限。因此，其他用户（即使用其他账户）将无法访问我们程序创建的文件。有关文件访问模式的更多信息，请参阅附录C，系统服务。

文件打开/创建系统服务的参数如下:

Register	SYS_creat
rax	Call code = SYS_creat (85)
rdi	Address of NULL terminated file name string
rsi	File access mode flag

假设以下声明:

```
SYS_creat    equ    85          ; file open

O_CREAT      equ    0x40
O_TRUNC      equ    0x200
O_APPEND     equ    0x400

S_IRUSR      equ    00400q      ; owner, read permission
S_IWUSR      equ    00200q      ; owner, write permission
S_IXUSR      equ    00100q      ; owner, execute permission
```

文件状态标志“S_IRUSR | S_IWUSR”允许同时读取和写入，这是典型的。"|"是一个逻辑或操作，因此组合了选择。

如果文件打开/创建操作不成功，rax寄存器返回负值。如果文件打开/创建操作成功，则返回一个文件描述符。文件描述符用于所有后续的文件操作。

参考示例文件写入部分的完整示例文件打开/创建。

13.6 文件读取

一个文件必须在读取之前使用适当的文件访问标志打开。

文件读取系统服务的参数如下:

Register	SYS_read
rax	Call code = SYS_read (0)
rdi	File descriptor (of open file)
rsi	Address of where to place characters read
rdx	Count of characters to read

假设以下声明:

```
SYS_read     equ     0               ; file read
```

如果文件读取操作不成功，则在rax寄存器中返回一个负值。如果文件读取操作成功，则返回实际读取的字符数。

参考下一节关于示例文件读取的完整文件读取示例。

13.7 文件写入

The parameters for the file write system service are as follows:

Register	SYS_write
rax	Call code = SYS_write (1)
rdi	File descriptor (of open file)
rsi	Address of characters to write
rdx	Count of characters to write

假设以下声明:

```
SYS_write equ 1 ; 文件写入
```

如果文件写入操作不成功，则在rax寄存器中返回一个负值。如果文件写入操作成功，则返回实际写入的字符数。

参考示例文件读取部分的完整文件写入示例。

13.8 文件操作示例

本节包含一些简单的示例程序，以演示非常基本的文件I/O操作。有关文件I/O缓冲的更复杂问题将在后续章节中解决。

13.8.1 示例，文件写入

这个示例程序将一条简短的消息写入文件。创建的文件包含一条简单消息，在这个例子中是一个URL。要写入文件的文件名和消息是硬编码的。这有助于简化示例，但并不现实。

由于使用了打开/创建服务，文件将被创建（即使必须覆盖旧版本）。

```
; 示例程序以演示文件I/O。此示例将打开/创建一个文件，将一些信息写入文
件，并关闭文件。注意，文件名和写入消息在此示例中是硬编码的。


节数 .data

; ----- ;
  定义标准常数。

LF等于10                                    ; 换行符
NULL 等于 0                                 ; 字符串结束
TRUE 等于 1 FALSE 等于 0 EX
IT_SUCCESS 等于 0
                                            ; 成功代码
STDIN 等于 0                                标准输入
STDOUT 等于 1                               ; 标准输出
STDERR 等于 2                               标准误
```

SYS_read 等于 0 ; 读取
SYS_write 等于 1 ; 编写
SYS_open 等于 2 ; 文件打开
SYS_close equ 3 ; 文件关闭
SYS_fork equ 57 分叉
SYS_exit equ 60 ; 终止
SYS_creat equ 85 ; 文件打开/创建
SYS_time equ 201 ; 获取时间

O_CREAT equ 0x40 O_TRUNC equ 0x200 O_APPEND equ 0x400 O_RDO NLY equ 000000q ; 只读 O_WRONLY equ 000001q ; 只写 O_RDWR equ 0 00002q ; 读写 S_IRUSR equ 00400q S_IWUSR equ 00200q S_IXUSR equ 0 0100q

; ----- ; 主变量。

newLine db LF, NULL header db LF, "文件写入示例。" db LF, LF, NULL fileNam e db "url.txt", NULL url db "http://www.google.com" db LF, NULL len dq $-url-1 w riteDone db "写入完成。", LF, NULL fileDesc dq 0 errMsgOpen db "打开文件错 误。", LF, NULL errMsgWrite db "写入文件错误。", LF, NULL ;---------- ----------

章节 .text 全局 _start _
start:

; ----- ; 显示标题行...

```
        mov rdi, header call printSt
        ring
```

; ----- ; 尝试打开文件。; 使用系统服务打开文件

; 系统服务 - 打开/创建
; rax = SYS_creat (文件打开/创建)
; rdi = 文件名字符串地址
; rsi = 属性（即只读等）

; 返回值: ; 如果出错 -> eax < 0 ; 如果成功 -> eax = 文件描述符编号

文件描述符指向文件控制块（FCB）。FCB由操作系统维护。文件描述符用于所有后续的文件操作（读取、写入、关闭）。

```
        mov rax, SYS_creat                                  ; 文件打开/创建
        mov rdi, fileName                                   文件名字符串
        mov rsi, S_IRUSR | S_IWUSR                          ; 允许读写
        系统调用                                            ; 调用内核 cmp rax, 0 ; 检
        查成功 jl errorOnOpen mov qword [fileDesc], rax


                                                            ; 保存描述符
```

; -----
; 写入文件。
; 在这个例子中，要写的字符在一个
; 预定义字符串包含一个URL。

; 系统服务 - 写入 ; rax = SYS_write ; rdi = 文件描述符 ;
rsi = 要写入的字符地址

; rdx = 要写入的字符数 ; 返回值: ; 如果出错 -> rax < 0 ; 如果成功 -> rax = 实际读取的字符数 mov rax, SYS_write mov rdi, qword [fileDesc] mov rsi, url mov rdx, qword [len] syscall cmp rax, 0 jl errorOnWrite mov rdi, writeDone call printString ; ----- ; 关闭文件。 ; 系统服务 - close ; rax = SYS_close ; rdi = 文件描述符 mov rax, SYS_close mov rdi, qword [fileDesc] syscall jmp exampleDone ; ----- ; 打开时出错。 ; 注意，rax包含一个错误码，在此示例中未使用。 errorOnOpen: mov rdi, errMsgOpen call printString jmp exampleDone ; ----- ; 写入时出错。 ; 注意，rax包含一个错误码，在此处未使用

错误写入：mov rdi, errMsgWrite call
printString jmp exampleDone ; ----- ;
示例程序完成。

exampleDone: mov rax, SYS_exit mov rd
i, EXIT_SUCCESS syscall

; ********** ; 通用函数,
用于在屏幕上显示字符串。 ; 字符串必须以NULL结尾。 ; 算法： ； 计算字符串中
的字符数（不包括NULL）； 使用系统调用输出字符

; 参数: ; 1) 地址, 字符串 ; 返回值:
无

全局 打印字符串 打印字
符串：压入 rbp 将 rbp 设
置为 rsp 压入 rbx

; 计算字符串中的字符数。mov rbx, rdi m
ov rdx, 0

strCountLoop: cmp byte [rbx], NULL je
strCountDone inc rdx inc rbx jmp strCo
untLoop

strCountDone: cmp rdx, 0 je prtDone ; 调用操作系统输出字符串. mov rax, SYS_write ; write() 的代码 mov rsi, rdi ; 字符地址 mov rdi, STDOUT ; 文件描述符 ; 设置上述系统调用计数；系统调用

; 字符串已打印，返回调用例程。

prtDone:
pop rbx pop rbp ret ; ********** ********** 这 e

xample 创建文件，该文件由下一个 exa 读取 示例

°13.8.2 示例，文件读取

这个示例将读取一个文件。要读取的文件包含一个简单的消息，来自上一个示例的URL。文件名是硬编码的，这有助于简化示例，但并不现实。使用的文件名与上一个文件写入示例匹配。如果在执行写入示例程序之前执行此示例程序，将生成错误，因为找不到文件。在执行文件写入示例程序之后，此文件读取示例程序将读取文件并显示内容。

; 示例程序，用于演示文件输入/输出。

; 此示例程序将打开一个文件，读取内容，并将内容写入屏幕。; 此例程还提供了一些关于处理系统服务中各种错误的非常简单的示例。; 注意，此示例中文件名是硬编码的。

; ----------

节数 .data

```
; -----
;  Define standard constants.

LF              equ     10              ; line feed
NULL            equ     0               ; end of string

TRUE            equ     1
FALSE           equ     0
EXIT_SUCCESS    equ     0               ; success code
STDIN           equ     0               ; standard input
STDOUT          equ     1               ; standard output
STDERR          equ     2               ; standard error

SYS_read        equ     0               ; read
SYS_write       equ     1               ; write
SYS_open        equ     2               ; file open
SYS_close       equ     3               ; file close
SYS_fork        equ     57              ; fork
SYS_exit        equ     60              ; terminate
SYS_creat       equ     85              ; file open/create
SYS_time        equ     201             ; get time

O_CREAT         equ     0x40
O_TRUNC         equ     0x200
O_APPEND        equ     0x400

O_RDONLY        equ     000000q         ; read only
O_WRONLY        equ     000001q         ; write only
O_RDWR          equ     000002q         ; read and write

S_IRUSR         equ     00400q
S_IWUSR         equ     00200q
S_IXUSR         equ     00100q

; -----
;  Variables/constants for main.

BUFF_SIZE       equ 255

newLine         db      LF, NULL
header          db      LF, "File Read Example."
                db      LF, LF, NULL
fileName        db      "url.txt", NULL
```

文件描述 c dq 0 错误打开文件，换行符，NULL 错误从文件读取，换行符，NULL

; ----------

段 .bss readBuffer resb BUFF_SIZE

; ----------

章节 .text 全局 _start_st
art:

; ----- ； 显示标题行... mov rdi, header call printString ; ----- ； 尝试打开文件 - 使用系统服务打开文件 ； 系统服务 - 打开 ; rax = SYS_open ; rdi = 文件名字符串的地址 ; rsi = 属性（例如，只读等） ；

返回:
; 如果错误 -> eax < 0
; 如果成功 -> eax = 文件描述符编号

; 文件描述符指向文件控制
; 块（FCB）。FCB 由操作系统维护。
; 文件描述符用于所有后续文件
; 操作（读取、写入、关闭）。

```
mov rax, SYS_open                    ; 文件打开
mov rdi, fileName                    文件名字符串
mov rsi, O_RDONLY                    只读访问
系统调用                             ; 调用内核
```

cmp rax, 0 ; 检查成功 jl errorOnOpen mov qword [fileDesc], rax ; 保存描述符

; ----- ; 从文件中读取。; 对于这个例子，我们知道文件只有一行。

; 系统服务 - 读取
; rax = SYS_read
; rdi = 文件描述符
; rsi = 数据放置地址
; rdx = 需要读取的字符数
; 返回:
; 如果错误 -> rax < 0
; 如果成功 -> rax = 实际读取的字符数

mov rax, SYS_readmov rdi, qword [fileDesc]mo v rsi, readBuffermov rdx, BUFF_SIZEsyscallcmp rax, 0 jl errorOnRead; ----- ; 打印缓冲区。; 为打印字符串添加NULLmov rsi, readBuffermov byte [rsi+rax], NULLmov rdi, readBuffercall printStringprintNewLin e; ----- ; 关闭文件。; 系统服务 - 关闭rax = SYS_clo se

```
;               rdi = 文件描述符

        mov rax, SYS_closemov rdi, qword [fileD
        esc]syscalljmp exampleDone



; ----- ; 打开错误。; 注意，eax包含一个错误代码，该代码在此示例中未使
用。


e 错误打开：mov rdi, errMsgOpen ca
  ll printString jmp exampleDone



; ----- ; 读取错误。; 注意，eax包含一个错误代码，该代码在此示例中未使
用。


错误读取：mov rdi, errMsgRead call
printString jmp exampleDone ; ----- ;
示例程序完成。




exampleDone: mov rax, SYS_exit mov rd
i, EXIT_SUCCESS syscall



; ********************************************************** ; 通用过程,
用于在屏幕上显示一个字符串。 ; 字符串必须以NULL结尾。
```

; 算法: ; 计算字符串中的字符数（不包括NULL） ; 使用系统调用输出字符

; 参数: ; 1) 地址, 字符串 ; 返回值: ; 无

全局 打印字符串 打印字符串: 压入 rbp 将 rbp 设置为 rsp 压入 rbx

; ----- ; 计算字符串中的字符数。

```
    mov rbx, rdimov rdx,
    0
strCountLoop: cmp byte [rbx], NULL je
strCountDone inc rdx inc rbx jmp strCo
untLoop

strCountDone:

    cmp rdx, 0 je prtDon
    e
```

调用操作系统输出字符串。

```
    mov rax, SYS_write                  ; write() 的代码
    mov rsi, rdi                        字符地址
    mov rdi, STDOUT                     ; 文件描述符 ; 设置的计数超过
    系统调用                            系统调用
```

字符串已打印, 返回调用例程。

```
prtDone:
    pop     rbx
    pop     rbp
    ret
```

函数 *printString()* 在两个示例中完全相同，仅重复以允许每个程序独立汇编和执行。

13.9 练习

以下是本章的一些测验问题和基于此章节的建议项目。

13.9.1 测验问题

以下是本章的一些测验问题。

1) 当使用系统服务时，*call code* 放置在哪里？2）当执行 syscall 指令时，代码位于何处（用户程序或操作系统中）？3）执行控制台输出的系统服务调用的调用代码和所需参数是什么？4）为什么只读取了一个字符用于交互式键盘输入？5）成功打开文件系统服务调用返回什么？6）不成功打开文件系统服务调用返回什么？7）如果一个系统服务调用需要六个（6）个参数，它们应该具体传递在哪里?

13.9.2 建议项目

以下是本章的一些建议项目。

1) 实现一个 *printString()* 示例 void 函数和一个简单的 main 函数来测试一系列字符串。使用调试器执行程序并显示最终结果。在不使用调试器的情况下执行程序，并验证适当的输出是否显示在控制台上。

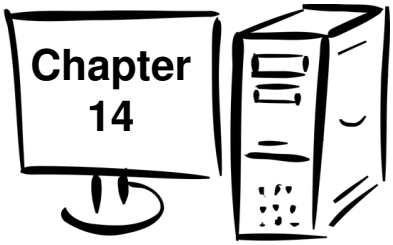
2) 将示例程序转换为从键盘读取输入的 *readString()* 函数。该函数应接受字符串地址和最大字符串长度（按此顺序）的参数。最大长度应包括空格用于空字符（额外一个字节），这意味着该函数不允许

超过最大值减一的字符要存储在字符串中。如果用户输入了额外的字符，应从输入流中清除，但不存储。函数不应包括返回字符串中的换行符。函数应返回字符串中不包括NULL的字符数。应使用前一个问题中的*printString()*函数，无需更改。完成后，创建适当的主函数以测试该函数。根据需要使用调试器来调试程序。当程序运行正确时，从命令行执行程序，将在控制台显示最终结果。

3) 基于文件写入示例，创建一个返回值 *fileWrite()* 函数以将密码写入文件。该函数应接受文件名地址和以 NULL 结尾的密码字符串地址的参数。应创建、打开文件，将密码字符串写入文件，然后关闭文件。如果操作正确，函数应返回 SUCCESS，如果存在问题，则返回 NOSUCCESS。问题可能包括无法创建文件或无法写入文件。创建一个适当的 main 函数来测试该函数。根据需要使用调试器来调试程序。当程序正常工作时，从命令行执行程序，将在控制台显示最终结果。

4) 基于文件读取示例，创建一个返回值 *fileRead()* 的函数，用于从文件中读取密码。该函数应接受文件名地址、存储密码字符串的地址、密码字符串的最大长度以及存储密码长度的地址作为参数。该函数应打开文件，读取表示密码的字符串，关闭文件，并返回密码中的字符数。最大长度应包括空字符的空间，这意味着函数读取时不能在字符串中存储超过最大长度减一的字符。如果操作正确，函数应返回 SUCCESS，如果存在问题，则返回 NOSUCCESS。问题可能包括文件不存在或其他读取错误。创建一个适当的主函数来测试该函数。根据需要使用调试器来调试程序。当程序运行正确时，从命令行执行程序，将在控制台显示最终结果。

My new computer is so fast, it executes an infinite loop in 6 seconds.



14.0 多个源文件

随着程序规模和复杂性的增长，程序的不同部分通常存储在不同的源文件中。这允许程序员确保源文件不会变得过于庞大，同时也使得多个程序员更容易地协同工作于程序的不同部分。

函数调用，即使是不同程序员编写的，也会因为第12章“函数”中概述的标准调用约定而正确协同工作。即使在与高级语言接口时也是如此。

本章将展示一个简单的汇编语言示例，以演示如何创建和使用多文件中的源代码。此外，还提供了一个如何与C/C++源文件接口的示例。

14.1 外部声明

如果从源文件调用函数且函数代码不在当前源文件中，汇编器将生成错误。同样适用于访问不在当前文件中的变量。为了通知汇编器函数代码或变量在另一个文件中，使用extern语句。语法如下：

extern **<**符号名**>**

符号名称将是函数或位于不同源文件中的变量的名称。通常，使用跨多个文件访问的全局变量被认为是较差的编程实践，并且应该谨慎使用（如果使用的话）。数据通常作为函数调用的参数在函数之间传递。

文本中的示例仅关注使用外部函数，而不使用全局声明的变量。

14.2 示例，总和与平均值

以下是一个简单的示例，其中main函数调用汇编语言函数 *stats()* 来计算一组有符号整数的整数总和和平均值。main函数和函数位于不同的源文件中，作为如何使用多个源文件的示例。这个示例本身实际上太小，不需要多个源文件。

14.2.1 组装主要

主要如下:

```
;  Simple example to call an external function.

; ----------------------------------------------------------
;  Data section

section    .data

; -----
;  Define standard constants

LF              equ     10              ; line feed
NULL            equ     0               ; end of string

TRUE            equ     1
FALSE           equ     0

EXIT_SUCCESS    equ     0               ; success code
SYS_exit        equ     60              ; terminate

; -----
;  Declare the data

lst1            dd   1, -2,  3, -4,  5
                dd   7,  9,  11
len1            dd   8

lst2            dd   2, -3,  4, -5,  6
                dd  -7, 10, 12, 14, 16
len2            dd  10
```

节段 .bss sum1 resd 1 ave1 resd 1
sum2 resd 1 ave2 resd 1

; ----------

extern 统计部分 .text
全局 _start _start:

; ---- ; 调用函数 ; HLL 调用: stats(lst, len, &sum, &ave); mov rdi, lst1 ;
数据集 1 mov esi, dword [len1] mov rdx, sum1 mov rcx, ave1 call stats
mov rdi, lst2 ; 数据集 2 mov esi, dword [len2] mov rdx, sum2 mov rcx,
ave2 call stats

; ----- ; 示例程序完成

示例完成：mov rax, SYS_exit mov rdi,
EXIT_SUCCESS syscall

上述主要部分可以使用与第5章“工具链”中描述的相同汇编命令进行组装。e
xtern语句将确保汇编器不会生成错误。

14.2.2 函数源

函数，在另一个源文件中如下所示：

```
; 简单示例 void 函数。

; ********************************************************** ; 数据声明 ;

    注意，本例中无需注意。
; 如有必要，它们将在这里声明，就像往常一样。

节数 .data

; **********************************************************

章节 .text

; ------------------------------------------------ ;
    函数用于查找整数总和和整数平均值
;   对于已通过的有符号整数列表。

; 调用：; stats(lst, len, &sum, &ave);


;   参数传递：
;               1) rdi - 数组地址
;               2) rsi - 已传递数组的长度
;               3) rdx - 求和变量的地址
;               4) rcx - 平均值变量的地址

; 返回值: ; 整数之和（通过引用）; 整数平均值（通过引
用）


全局统计 统计：推
送 r12


; ----- ;  查找并返回总和。


        mov r11, 0 ; i=0mov r12d, 0 ; sum=0
```

```
sumLoop:
    mov     eax, dword [rdi+r11*4]    ; get lst[i]
    add     r12d, eax                 ; update sum
    inc     r11                       ; i++
    cmp     r11, rsi
    jb      sumLoop

    mov     dword [rdx], r12d         ; return sum

; -----
;  Find and return average.

    mov     eax, r12d
    cdq
    idiv    esi

    mov     dword [rcx], eax          ; return average

; -----
;  Done, return to calling function.

    pop     r12
    ret
```

上述源文件可以使用与第5章“工具链”中描述相同的汇编命令进行组装。由于没有调用外部函数，因此不需要extern语句。

14.2.3 组装和链接

假设主源文件命名为 *main.asm*，函数源文件命名为 *stats.asm*，以下命令将执行汇编和链接。

yasm -g dwarf2 -f elf64 main.asm -l main.lst yasm -g dwarf2 -f elf64 stats.asm -l stats.lst ld -g -o main main.o stats.o

文件名可以根据需要更改。

通常，使用 ddd <可执行文件> 命令启动调试器。需要注意的是，使用调试器单步执行命令将进入函数，包括显示函数源代码（即使源代码在另一个文件中）。

调试器的下一个命令将执行整个函数而不显示源代码。如果函数工作正常，下一个命令将最有用。为了调试函数，步进命令将最有用。

如果省略了“-g”选项，调试器将无法显示源代码。

14.3 与高级语言接口

本节提供了关于高级语言如何调用汇编语言函数以及汇编语言函数如何调用高级语言函数的信息。本章提供了这两个方面的示例。

简要来说，如何实现这一点的答案是采用标准调用约定。因此，在与高级语言接口时不需要额外的或特殊的代码。编译器或汇编器需要了解外部例程，以避免出现无法找到非局部例程源代码的错误信息。

多文件链接的一般过程在第五章“工具链”中进行了描述。使用多个源文件的过程在第十四章节“多个源文件”中进行了描述。对象文件是来自高级语言还是汇编语言源代码并不重要。

14.3.1 示例，C{v*} 主/组装功能

当调用在单独源文件中的任何函数时，编译器必须被告知该函数或函数的源代码位于当前源文件外部。这通过C或C++中的extern语句来实现。其他语言将有类似的语法。对于高级语言，extern语句将包括函数原型，这将允许编译器验证函数参数和相关类型。

以下是一个使用C++主程序和汇编语言函数的简单示例。

#include <iostream> 使用命名空间 std; extern "C" void stats(int[], int,
int *, int *);

```
int main()
{
    int lst[] = {1, -2, 3, -4, 5, 7, 9, 11};
```

```
    int len {v*} 8; int sum, ave; stats(lst, len, &sum, &
ave); cout << "Stats:" << endl; cout << "  Sum = " << s
um << endl; cout << "  Ave = " << ave << endl; return
0; }
```

在这个点上，编译器不知道外部函数是用汇编编写的（这也不重要）。

C 编译器在 Ubuntu 上已预安装。然而，C++ 编译器默认未安装。

一个完整的C版本程序也被提供。

```
#include<stdio.h> extern void stats(int[], int, int *, int *);


int main() { int lst[] = {1, -2, 3, -4, 5, 7, 9, 11}; int len = 8; int su
m, ave; stats(lst, len, &sum, &ave); printf ("Stats:\n"); printf ("
Sum = %d \n", sum); printf ("  Ave = %d \n", ave); return 0; }
```

此处引用的 *stats()* 函数应从上一个示例中保持不变使用。

14.3.2 编译、汇编和链接

如函数章节所述，应使用C++编译器。例如，假设C++主函数名为*main.cpp*，汇编源文件名为*stats.asm*，编译、汇编、链接和执行的命令如下：

```
g++ -g -Wall -c main.cpp yasm -g dwarf2 -f elf64 stats.asm -l stats.l
st g++ -g -o main main.o stats.o
```

Note, Ubuntu 18 及以上版本需要在 g++ 上启用 no-pie 选项，如下所示：

```
g++ -g -no-pie -o main main.o stats.o
```

文件名可以根据需要更改。执行后，输出将如下所示：

```
./main 统计：
总计 = 30   平
均 = 3
```

如果使用C主程序，并且假设C主程序命名为 *main.c*，汇编源文件命名为 *stats.asm*，编译、汇编、链接和执行的命令如下：

```
g++ -g -Wall -c main.cpp yasm -g dwarf2 -f elf64 stats.asm -l stats.l
st g++ -g -o main main.o stats.o
```

Note, Ubuntu 18 及以上版本需要在 gcc 上启用 no-pie 选项，如下所示：

```
g++ -g -no-pie -o main main.o stats.o
```

C 编译器，gcc，或 C++ 编译器，g++，用于执行链接，因为它知道各种 C/C++ 库的适当位置。

14.4 练习

以下是本章的一些测验问题和基于此章节的建议项目。

14.4.1 测验问题

以下是本章的一些测验问题。

1) 如何声明函数 *func1()* 和 *func2()* 为外部函数，假设这两个函数都不需要任何参数？2) 如果每个函数使用两个整数参数，如何声明函数 *func1()* 和 *func2()* 为外部函数？3) 如果调用了一个未声明为外部函数的外部函数，会发生什么？4) 如果调用了一个外部声明的函数，但程序员实际上没有编写该函数，错误会在汇编时间、链接时间还是运行时被标记？5) 如果调用了一个外部声明的函数，但程序员实际上没有编写该函数，可能出现的错误是什么？6) 如果在汇编和链接命令中省略了“-g”选项，程序能否执行?

14.4.2 建议项目

以下是本章的一些建议项目。

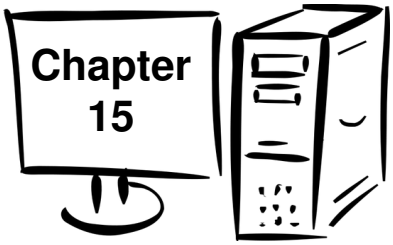
1) 实现汇编语言示例程序，用于查找一组有符号整数的总和和平均值。确保主函数和函数在不同的源文件中。使用调试器执行程序并显示最终结果。

2) 基于示例函数 *stats(),* ，将其拆分为两个返回值的函数，*lstSum()* 和 *lstAverage()*。如第12章“函数”所述，返回值的函数将结果返回到A寄存器。由于这些是双字，结果将返回到eax。确保主函数和这两个函数位于两个不同的源文件中。这两个函数可以位于同一个源文件中。使用调试器执行程序并显示最终结果。

3) 将之前的练习扩展到在控制台显示总和和平均值。将 *printString()* 示例函数（来自多个之前的示例）放置在第三个源文件中（可以在其他练习中使用）。此项目需要一个将整数转换为ASCII的函数（如第10章所述）。根据需要使用调试器来调试程序。在工作时，不使用调试器执行程序，并验证是否正确显示到控制台。

4) 实现C/C++示例主程序之一（任选其一）。另外，实现汇编语言*stats()*函数示例。开发一个简单的bash脚本以执行编译、汇编和链接。链接应使用相应的C/C++编译器执行。根据需要使用调试器调试程序。在工作时，不使用调试器执行程序，并验证正确的结果是否显示在控制台。

My new computer is so fast it requires two HALT instructions to stop it.



15.0 栈缓冲区溢出

栈缓冲区溢出[44]可能发生在程序溢出基于栈的动态变量（如第12.9章所述，基于栈的局部变量）。例如，如果一个程序分配并使用一个包含50个以上元素的基于栈的局部数组，并且数组中存储了超过50个元素，就会发生溢出。这种溢出通常很糟糕，通常会导致程序出现错误，甚至可能使程序崩溃。栈将包含其他重要信息，例如其他变量、保留寄存器、帧指针、返回地址和/或基于栈的参数。如果此类数据被覆盖，可能会引起难以调试的问题，因为症状可能与问题实际发生的位置无关。

如果堆栈缓冲区溢出是作为攻击的一部分故意造成的，则称为堆栈破坏。由于标准调用约定，基于堆栈的调用帧或活动记录的布局相当可预测。恶意个人可以利用这种堆栈缓冲区溢出将可执行代码注入当前运行的程序中，以执行一些不适当的行为。在适当的条件下，这种代码注入可能允许黑帽[45]黑客执行不受欢迎的操作，甚至可能接管系统。

该章节提供了堆栈缓冲区溢出发生的过程以及如何利用它的方法。这是为了使开发者能够清楚地理解问题，从而学习如何保护自己免受此类漏洞的侵害。读者必须熟悉第12章“函数”中概述的标准调用约定细节。

应注意的是，堆栈缓冲区溢出问题存在于高级语言中。在汇编语言中工作可以更容易地更清晰地看到和理解细节。

44 For more information, refer to: http://en.wikipedia.org/wiki/Stack_buffer_overflow
45 For more information, refer to: http://en.wikipedia.org/wiki/Black_hat

15.1 理解堆栈缓冲区溢出

当程序调用一个函数时，标准调用约定提供了关于如何传递参数、如何保存返回地址、必须保留哪些寄存器以及如何分配基于堆栈的局部变量的指导。

例如，考虑函数调用；

```
expFunc(arg1, arg2, arg3);
```

此外，我们假设函数*expFunc()*从用户读取一行文本并将其存储在局部声明的数组中。局部变量包括一个48字节的数组和一个8字节的四倍字局部变量。这可以在高级语言中轻松实现，或者在第13章“系统服务”中的控制台输入中那样实现。

结果调用帧如下：

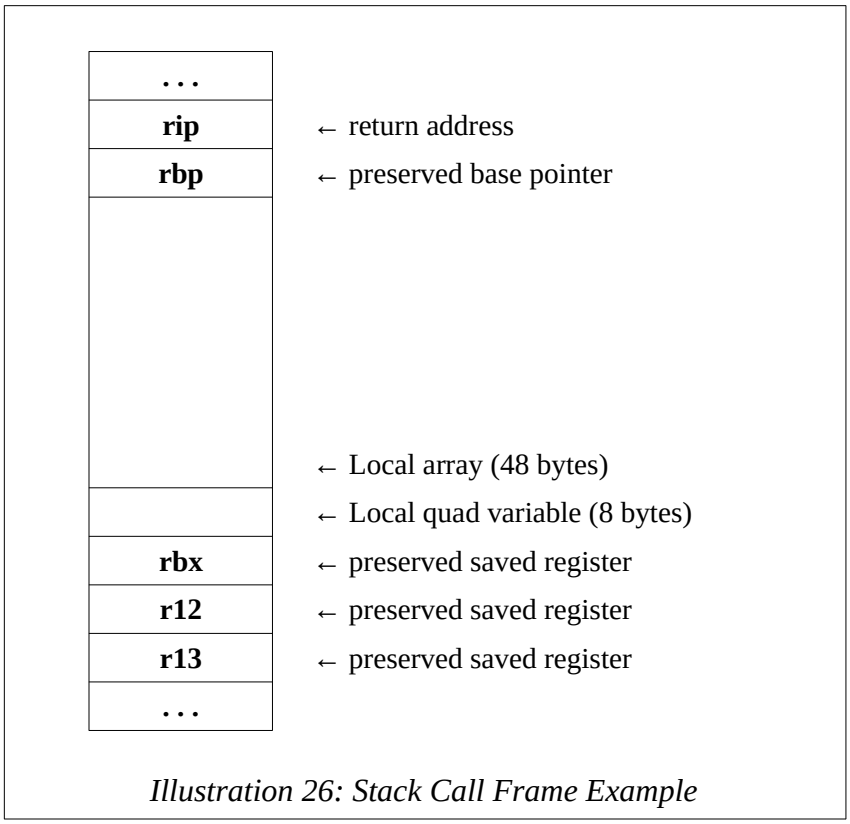


Illustration 26: Stack Call Frame Example

当函数完成时，调用帧的所有元素都被移除。保留的寄存器恢复到其原始内容，局部变量被移除，返回地址从栈中复制出来并放置在rip寄存器中，从而实现跳回到原始调用例程。正如第12章“函数”中所述，这种布局支持包括递归在内的多级函数调用。

第13章中的示例从用户读取字符并将它们显式地输入到一个数组中，明确检查了字符计数以确保计数不超过缓冲区大小。很容易忘记执行这个重要的检查。实际上，一些C函数，如*strcpy()*，不执行大小验证，因此被认为是不安全的[46]或被描述为不安全的函数。

在这个例子中，覆盖48个字符的缓冲区将破坏包含rbp寄存器原始值的栈的内容，以及可能包含rip寄存器的原始值。如果rip寄存器的原始值栈内容被以任何方式更改或损坏，函数将无法返回到调用例程，并会尝试跳转到某个随机位置。如果随机位置在程序作用域之外，这很可能是这种情况，程序将生成段错误（即“段错误”或程序崩溃）。

调试此类错误可能是一个重大挑战。错误出现在函数的非常末尾（在返回时）。问题实际上在函数体中，实际上可能是因为未包含的代码（而不是那里错误的代码）。

测试一个程序是否具有这种漏洞将涉及在提示时输入比预期显著更多的字符。例如，如果提示输入姓名并且输入了200个字符（可能是通过按住键实现的），程序崩溃将是一个这样的漏洞存在的迹象。错误信息将表明这种漏洞不存在。虽然简单，但这种类型的测试往往没有得到彻底检查，甚至完全省略。

15.2 注入代码

在讨论堆栈缓冲区溢出可能被利用的方式之前，我们将回顾可能被注入的代码。要注入的代码可能是许多事物。我们将假设程序在一个受控环境中执行，用户没有控制台访问权限。缺乏控制台访问权限将限制用户理想上能做的事情，仅限于程序允许的范围内。如果程序是基于服务器的，可能会出现这种情况。

46 For more information, refer to: http://en.wikipedia.org/wiki/C_standard_library#Buffer_overflow_vulnerabilities

与用户通过网页或应用程序前端交互。服务器会因明显的安全原因而受到直接用户访问的保护。

因此，一个可能的攻击向量可能是获取对后端服务器（通常不允许此类访问）的控制台访问权限。

存在一种系统服务可以执行另一个程序。使用此系统服务 exec，我们可以执行 sh 程序（shell），这是一个标准的 Linux 程序。这将创建并打开一个 shell，提供控制台访问。新创建的 shell 仅限于启动它的进程的权限。在良好的安全环境中，进程将仅被授予所需的权限。在较差的安全环境中，进程可能具有额外的、不必要的权限，这会使未经授权的黑帽黑客更容易造成问题。

给定以下数据声明：

```
NULL            equ     0
progName        db      "/bin/sh", NULL
```

An个exec系统服务的示例可能如下　　允许以下内容：

; 示例，系统服务调用exec。

```
mov rax, 59
mov rdi, progName
系统调用
```

一个新控制台将出现。鼓励读者尝试此代码示例并查看其工作。

此代码片段的列表文件如下：

40 00000000 48C7C03B000000　　mov rax, 59 41 00000007 48C7C7[0000 0000]　　mov rdi, progName 42 0000000E 0F05　　syscall

回忆一下，第一列是行号，第二列是代码段的相对地址，第三列是机器语言或第四列中显示的指令的十六进制表示。[00000000]代表数据段中字符串的相对地址（在本例中为*progName*）。它是零，因为它是本例数据段中的第一个（也是唯一一个）变量。

如果将显示的机器语言输入内存并执行，将打开一个外壳或控制台。将代码的十六进制版本放入内存只能通过键盘完成（因为没有直接访问文件系统的权限）。这将逐个字符进行。0x48是“0”的ASCII码，因此可以为该字节输入“0”。然而，0x0f和其他许多字符直接作为ASCII输入则更为困难。

命令行将允许以十六进制输入十六进制代码。通过按住控制键和左 Shift 键，然后输入小写字母 u 和四个十六进制数字（必须是十六进制），可以逐个输入十六进制值。例如，要输入 0x3f，它将是；

CTRL SHIFT u 3 f

这可以用于大多数字节，除了0x00。0x00是一个空字符，它是一个不可打印的ASCII字符（用于标记字符串结束）。因此，空字符不能从键盘输入。此外，如果代码被注入到另一个程序中，[00000000]地址就没有意义了。

为了解决这些问题，我们可以重新编写示例代码并消除NULL引用并更改地址引用。可以通过使用不同的指令来消除NULL引用。例如，将rax设置为59可以通过将rax与自身异或并将59放入al（已经通过异或确保了高56位为0）来实现。字符串可以放在栈上，当前rsp用作字符串的地址。字符串“\bin\sh”是7个字节，栈操作将需要推入8个字节。同样，NULL不能输入且不计入。可以额外添加一个不必要的“/”到字符串中，这不会影响操作，只要字符串中有正好8个字节。由于架构是little-endian，为了确保字符串的开始在低内存中，它必须在推入的最不重要字节中。这将使字符串看起来是反的。

修订后的程序片段如下:

```
XOR rax, rax                          ; 清除 rax
push rax                              ; 在栈上放置NULL
mov rbx, 0x68732f6e69622f2f           ; 字符串 -> "//bin/sh"
推动 rbx                              ; 将字符串放入内存
mov al, 59                            ; 调用rax中的代码
mov rdi, rsp                          ; rdi = 字符串地址
系统调用                              系统调用
```

程序片段的列表文件如下:

```
52 00000013 4831C0                  XOR rax, rax
53 00000016 50                      push rax
54 00000017 48BB2F2F62696E2F73      mov rbx, 0x68732f6e69622f2f
55 00000017 68
56 00000021 53                      推动 rbx
57 00000022 B03B                    mov al, 59
58 00000024 4889E7                  mov rdi, rsp
59 00000027 0F05                    系统调用
```

在此修订代码中，没有NULL，地址引用是从栈指针（rsp）获得的，它指向正确的字符串。

存在一个汇编语言指令nop，它执行无操作，机器码为0x90。在这个例子中，nop指令被简单地用来将机器码填充到8字节偶数倍。

需要输入的十六进制值系列如下:

```
0x48 0x31 0xC0 0x50 0x48 0xBB 0x2F 0x2F 0x62 0x69 0
x6E 0x2F 0x73 0x68 0x53 0xB0 0x3B 0x48 0x89 0xE7 0x
0F 0x05 0x90 0x90
```

虽然有些繁琐，但这些字符可以通过手工输入。

15.3 代码注入

如果可用的注入代码可以输入，下一步实际上就是执行该代码。

基于之前的示例调用帧，代码将进入，前面跟着一系列的nop指令（0x90）。在栈中存储rip的确切位置可以通过试错法确定。当8字节地址的第一个字节被更改时，程序将无法返回到调用例程，并且很可能会崩溃。如果rbp的字节被损坏，程序可能会以某种方式失败，但与损坏rip引起的立即崩溃不同。输入的代码将在许多连续尝试的每次尝试中扩展1个字节。以这种方式找到这个确切位置需要耐心。

一旦确定了拆分位置，那里输入的8个字节将需要是注入代码在用户输入存储的堆栈中的地址。这也将通过试错来确定。然而，注入代码开始的精确地址并不需要。从前面任何位置开始

nop的指令就足够了。这被称为NOP滑动[47]，它将有助于将CPU的指令执行流程“滑动”到注入的代码中。

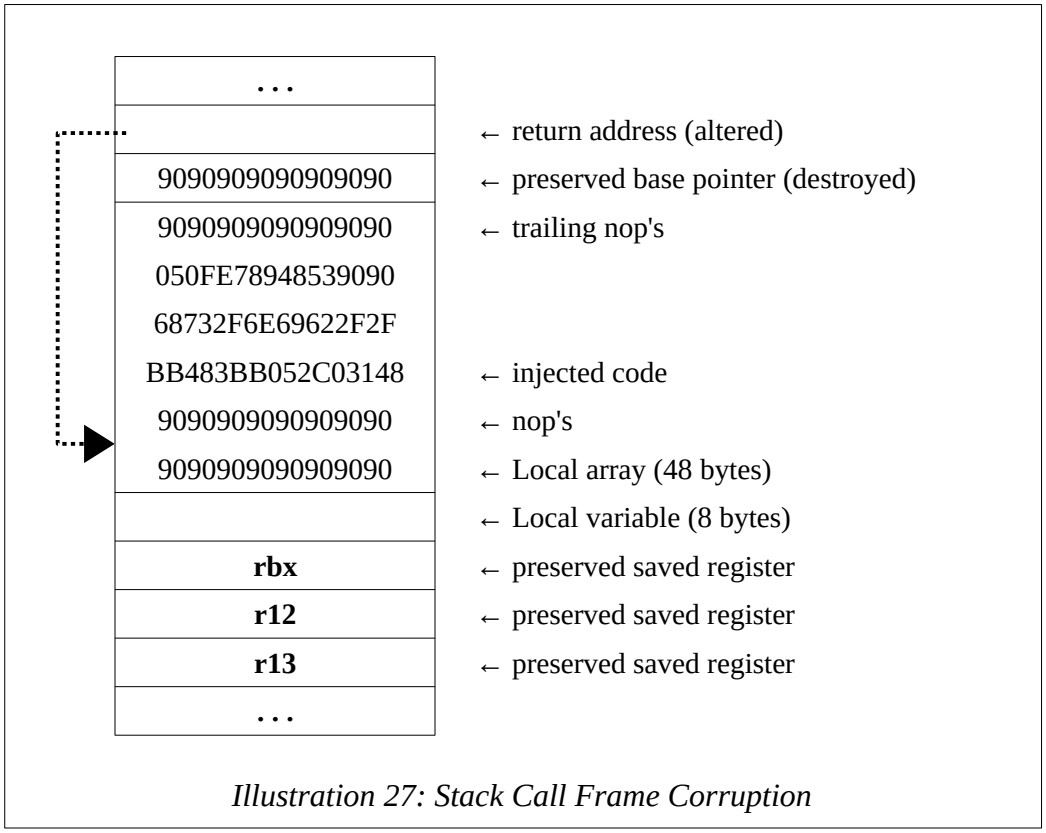


Illustration 27: Stack Call Frame Corruption

更大的局部数组将允许更长的NOP滑动有更多空间。

15.4 代码注入防护

一些方法已被开发和实施以防止堆栈缓冲区溢出。其中一些方法在此总结。必须注意的是，这些方法中没有任何一个是完全完美的。

47 For more information, refer to: http://en.wikipedia.org/wiki/NOP_slide

15.4.1 数据堆栈破坏保护器（或Canaries）

数据堆栈破坏保护器，也称为堆栈卡纳里，用于在恶意代码执行之前检测堆栈缓冲区溢出。这是通过在程序开始时随机选择一个整数，并将其放置在调用帧中返回地址（rip）之前的内存中来实现的。为了覆盖返回地址并执行注入的代码，还必须覆盖卡纳里值。在例程弹出返回地址之前，会检查这个卡纳里值以确保它没有改变。

对于GNU g++编译器，此选项（-f-stack-protector）默认启用。可以使用-fno-stack-protector编译器选项将其关闭。关闭它对于使用本章概述的注入技术进行测试是必要的。

15.4.2 数据执行预防

数据执行保护[48]（DEP）是一种安全功能，将内存区域标记为“可执行”或“不可执行”。只有标记为“可执行”区域的代码允许执行。标记为“不可执行”区域中的代码或注入的代码将不允许执行。这有助于防止堆栈缓冲区溢出代码注入。

15.4.3 数据地址空间布局随机化

地址空间布局随机化（ASLR）是一种防止攻击者可靠地跳转到注入代码的技术。ASLR随机排列进程关键数据区域的地址空间位置，包括可执行文件的基址以及栈、堆和库的位置。

15.5 练习

以下是本章的一些测验问题和基于此章节的建议项目。

15.5.1 测验问题

以下是本章的一些测验问题。

1) 当堆栈缓冲区溢出被故意作为攻击的一部分时，它被称为什么？2）当一个C函数被认为是不安全时，这意味着什么?

48 For more information, refer to: http://en.wikipedia.org/wiki/Data_Execution_Prevention

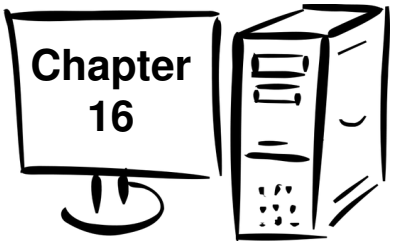
3) 如果输入缓冲区足够大（例如，>1024字节），那么读取用户输入的程序是否仍然容易受到攻击？4) 攻击者如何确定一个交互式程序是否容易受到缓冲区溢出攻击？5) 什么是“NOP滑块”？ 6) 文本示例注入的代码用于打开一个新的shell。请至少提供一个不同的注入代码想法，该代码会导致问题。7) 列举三种旨在防止堆栈缓冲区溢出攻击的技术。

15.5.2 建议项目

以下是本章的一些建议项目。

1) 实现第二个示例程序片段以打开一个新的shell。使用调试器执行程序并显示最终结果。在不使用调试器的情况下执行程序，并验证是否打开了新的shell。2) 实现第13章的控制台输入程序。删除缓冲区大小检查的代码。在不使用调试器的情况下执行程序，并确保读取了适当的输入，并将输出显示到控制台。验证输入过多字符会导致程序崩溃。3) 使用前一个问题中的程序和打开shell的程序片段，尝试将代码注入到正在运行的程序中。为了节省时间，在适当的位置打印rsp的值，以便更容易地猜测目标地址。

The code that is the hardest to debug is the code you know can't possibly be wrong.



16.0 命令行参数

本章节提供了操作系统处理命令行参数以及汇编语言例程如何访问参数的总结。

“命令行参数”这个术语用来指代在程序名称之后输入的字符（如果有）。命令行参数通常用于向程序提供信息或参数，而无需与交互式提示相关的I/O。当然，如果参数不正确，整个行必须重新输入。

命令行参数被汇编器和链接器用于提供有关输入文件、输出文件和各种其他选项的信息。

16.1 解析命令行参数

操作系统负责解析或读取命令行参数并将信息传递给程序。确定什么是正确和错误的是程序的责任。在读取命令行时，操作系统会将一个参数视为一组非空格字符（即字符串）。操作系统会移除或忽略参数之间的空格。此外，程序名称本身被视为第一个，也可能是唯一的参数。如果输入了其他参数，每个参数之间至少需要有一个空格。

所有参数都作为字符字符串传递给程序，即使是数字信息。如果需要，程序必须将字符数据转换为数值（浮点数或整数）。

例如，使用以下命令行参数执行程序 expProg：

```
./expProg  one    42 three
```

将产生以下四个参数:

1. ./expProg2. one
3. 424. three

命令行参数作为参数传递给程序。因此，程序就像是被操作系统调用的函数。所以，使用标准调用约定来传递参数。

16.2 高级语言示例

通常，假设读者已经熟悉基本的C/C{v*}命令行处理。本节简要介绍了C/C{v*}语言如何处理将命令行信息传递给程序的方式。

参数的计数作为第一个整数参数传递给主程序，通常称为 ***argc***。第二个参数，通常称为 ***argv***，是与每个参数相关联的字符串的地址数组。

例如，以下C++示例程序将读取命令行参数并将它们显示到屏幕上。

```
#include <iomanip> #include <iostream> using namespace std; int mai
n(int argc, char* argv[]) { string bars; bars.append(50,'-'); cout << bars
<< endl; cout << "Command Line Arguments Example" << endl << en
dl; cout << "Total arguments provided: " << argc << endl; cout << "Th
e name used to start the program: " << argv[0] << endl; }
```

```
    if (argc > 1) { cout << endl << "The arguments are:" << endl; for (int n
= 1; n < argc; n++) cout << setw(2) << n << ": " << argv[n] << endl; }cou
t << endl; cout << bars << endl; return 0; }
```

假设程序名为 argsExp，执行此程序将产生以下输出：

```
./argsExp   一个 34       三个 ------------------------------------------------- 命
令行参数示例 总共提供的参数数：4 用于启动程序的名称：./argsExp
参数如下：   1: 一个  2: 34  3: 三个 ------------------------------------------------
-----
```

应注意的是，参数 '34' 是一个字符串。此示例只是简单地将字符串打印到控制台。如果要将参数用作数值，程序必须按要求进行转换。这包括所有必要的错误检查。

16.3 参数计数和参数向量表

由于操作系统将主程序作为函数调用，因此适用标准调用约定。根据标准调用约定，参数计数和参数向量地址作为参数传递给 rdi 和 rsi。rdi 中的第一个参数是整数参数计数。在 rsi 中的第二个参数是参数向量表地址。

参数向量表是一个包含每个参数字符串的四字地址的数组。假设前面的例子中有4个总参数，基本的参数向量表布局如下:

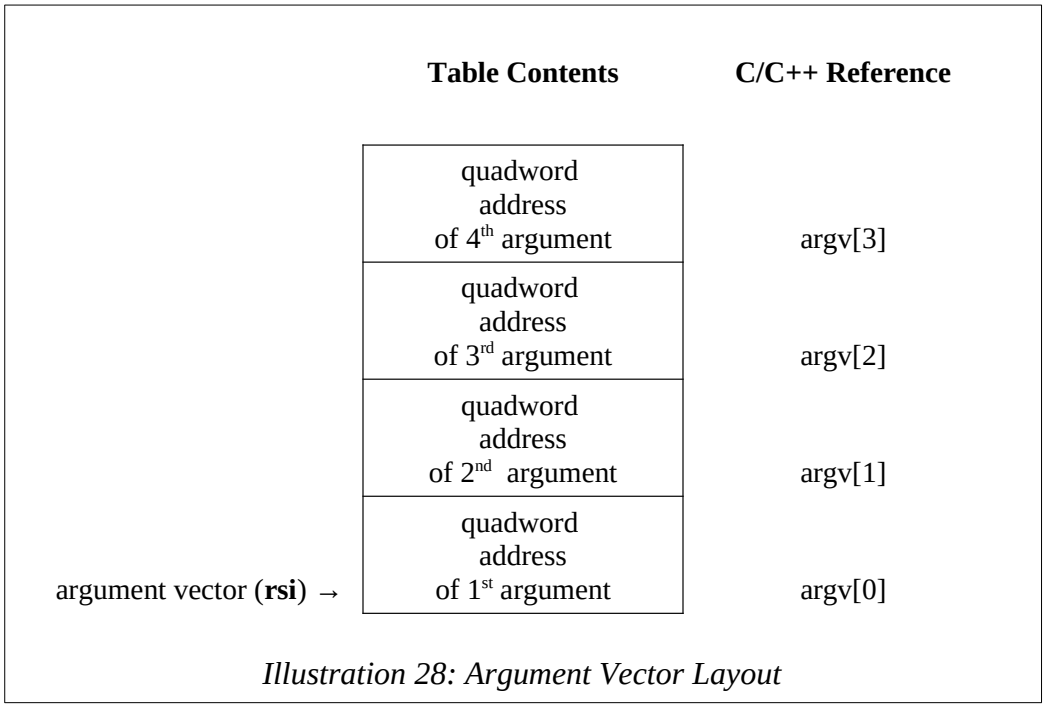


Illustration 28: Argument Vector Layout

每个字符串由操作系统以NULL终止，并且不会包含换行符。

16.4 汇编语言示例

一个用于读取和显示命令行参数的汇编语言程序示例包含供参考。此示例仅读取并显示命令行参数。

```
; 命令行参数示例
; ----------------------------------------------------

节数 .data

; ----- ; 定义标准常数。
```

LF 等于 10 ; 换行 NULL 等于 0 ; 字符串结束 TRUE 等于 1 FALSE 等于 0 E
XIT_SUCCESS 等于 0 ; 成功代码 STDIN 等于 0 ; 标准输入 STDOUT 等于 1
; 标准输出 STDERR 等于 2 ; 标准错误 SYS_read 等于 0 ; 读取 SYS_write 等
于 1 ; 写入 SYS_open 等于 2 ; 文件打开 SYS_close 等于 3 ; 文件关闭 SYS_fo
rk 等于 57 ; 分叉 SYS_exit 等于 60 ; 终止 SYS_creat 等于 85 ; 文件打开/创建
SYS_time 等于 201 ; 获取时间

; ----- ; 主变量。

换行符 db LF，NULL

; --

章节 .text 全局 主 主
:

; ----- ; 获取命令行参数并回显到屏幕。 ; 基于标准调用约定， ; rdi = argc (参
数计数) ; rsi = argv (参数向量起始地址) mov r12, rdi ; 保存以供后续使用... mo
v r13, rsi

; ----- ; 简单循环以将每个参数显示到屏幕上。 ; 每个参数都是一个以NULL结
尾的字符串，因此可以直接打印。

打印参数： mov rdi, newLine call printString m
ov rbx, 0 printLoop: mov rdi, qword [r13+rbx*8]
call printString mov rdi, newLine call printString
inc rbx cmp rbx, r12 jl printLoop

; ----- ； 示例程序完成。

exampleDone: mov rax, SYS_exit mov rd
i, EXIT_SUCCESS syscall

; ** ； 通用过程,
用于在屏幕上显示字符串。 ； 字符串必须以NULL结尾。 ； 算法： ； 计算字符串中
的字符数（不包括NULL） ； 使用系统调用输出字符 ； 参数： ; 1) 地址，字符串 ;
返回值： ; 无

全局打印字符串printString: push rbp mov r
bp, rsp push rbx ; ----- ； 计算字符串中的字
符数。{v*}

```
    mov rbx, rdimov rdx, 0strCountLo
op: cmp byte [rbx], NULL je strCountD
oneinc rdxinc rbxjmp strCountLoop



strCountDone:

    cmp rdx, 0 je prtDon
    e
```

调用操作系统输出字符串。

```
    mov rax, SYS_write ; write() 的代码mov rsi, rdi ; 字符串地址mov edi, STDO
    UT ; 文件描述符count set above syscall ; 系统调用system call
```

字符串已打印，返回调用例程。

```
prtDone: pop rbx po
p rbp ret


; ****************************************************Translated Text: ******************
```

该 *printString()* 函数在本例中重复出现，并且与之前的示例保持不变。

必须注意，为了使这可行，程序应按常规组装，但使用GNU C编译器链接，无论是GCC还是G++。

例如，假设此示例程序命名为cmdLine.asm，汇编和链接过程如下：

```
yasm -g dwarf2 -f elf64 cmdLine.asm -l cmdLine.lst
gcc -g -o cmdLine cmdLine.o
```

Note, Ubuntu 18 将需要在 gcc 命令中使用 no-pie 选项，如下所示：

```
gcc -g -no-pie -o cmdLine cmdLine.o
```

如果使用标准链接器，参数将不会以正确的方式传递。

16.5 练习

以下是本章的一些测验问题和基于此章节的建议项目。

16.5.1 测验问题

以下是本章的一些测验问题。

1) 哪个软件实体负责解析或读取命令行参数？2) 哪个软件实体负责验证或检查命令行参数？3) 第一个命令行参数是什么？4) 解释argc和argv的含义。5) 在汇编语言程序中，argc传递给程序的哪个位置？6) 在汇编语言程序中，argv传递给程序的哪个位置？7) 如果每个命令行参数之间输入了七个空格，当检查命令行参数时，这些空格是如何被移除的？8) 如果期望命令行参数是一个数字，而用户输入了“12x3”（一个无效值），操作系统（即加载器）会生成错误吗？

16.5.2 建议项目

以下是本章的一些建议项目。

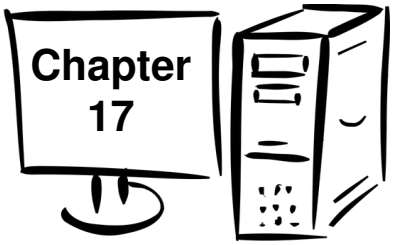
1) 实现示例程序以读取和显示命令行参数。使用调试器执行程序并显示最终结果。在不使用调试器的情况下执行程序，并验证适当的输出是否显示在控制台上。

2) 将命令行示例转换为函数，该函数将显示每个命令行参数到控制台。根据需要使用调试器调试程序。在不使用调试器的情况下执行程序，并验证适当的输出是否显示在控制台。

3) 创建一个汇编语言程序，以在命令行中接受一个文件名并打开该文件，显示文件中包含的一行消息。应创建一系列小的文本文件，每个文件包含本文本每章开头的一条非常重要的消息。程序应对文件名进行错误检查，如果有效，则打开文件。如果文件成功打开，则应从文件中读取消息并在控制台显示。如果无法打开或读取文件，应显示适当的错误消息。程序可以假设每条消息将是 < 256个字符。根据需要使用调试器来调试程序。在不使用调试器的情况下执行程序，并验证适当的输出是否显示在控制台。

4) 创建一个汇编语言程序，该程序将从命令行接受三个无符号整数，将这三个数相加，并显示每个原始数和总和。如果提供的命令行参数过多或过少，应显示错误消息。此程序需要将每个ASCII字符串转换为整数。应包括适当的错误检查。例如，“123”是正确的，而“12a3”是错误的。主程序应根据需要调用函数进行ASCII到整数的转换和输出。根据需要使用调试器调试程序。在不使用调试器的情况下执行程序，并验证适当的输出是否显示在控制台上。

If I had it all to do over again, I'd spell creat with an "e".
- Kernighan



17.0 输入/输出缓冲区

本章提供了有关输入/输出（I/O）缓存的资料。I/O 缓存是一种提高 I/O 操作整体吞吐量和效率的过程。I/O 缓存的一般主题非常广泛，可以包括许多事物，包括硬件、软件和数据网络。本文讨论了从文件中读取信息时的单缓冲区缓存。虽然这是一个非常具体的应用，但关于如何和为什么进行缓存的一般概念普遍适用于许多不同的应用。

高级语言提供了执行I/O缓冲的库函数，并从程序员那里隐藏了相关的复杂性。这在编程时是非常理想的。在低级工作时，缓冲需要被明确实现（由我们）。目标是提供对为什么以及如何执行缓冲的详细理解，以便更深入地了解计算机的工作原理。这应该有助于消除这种I/O是魔法的观念。当然，这也有助于我们欣赏编译器的I/O库函数。

17.1 为什么使用缓冲区?

为了充分理解缓冲的重要性，回顾第2章“架构”中概述的内存层次结构是有用的。正如所提到的，与主存储器访问相比，对二级存储器的访问在运行时成本上要高得多。这会强烈鼓励我们通过在主存储器中使用一些临时存储（称为缓冲区）来限制对二级存储器的访问次数。

这样的缓冲区还可以帮助减少与系统调用相关的开销。例如，一个文件读取系统调用将涉及暂停我们的程序并将控制权转交给操作系统。操作系统将验证我们的请求（例如，确保文件描述符有效），然后通过系统总线将请求传递给辅助存储控制。

控制器会执行必要的操作以获取所需数据，并按照操作系统的指令将其直接放置到主存储器位置，再次访问系统总线以执行传输。*Note*，这被称为直接内存访问[49]（DMA）。一旦辅助存储控制器完成传输，它会通知操作系统。然后操作系统会通知并恢复我们的程序。这个过程代表了系统开销，因为我们的程序只是在等待系统服务请求的完成。在每个系统服务调用中获取更多数据而不是更少数据是有意义的。很明显，限制系统服务请求的数量将有助于提高我们程序的总体性能。

此外，如第13章“系统服务”所述，低级文件读写操作需要读取的字符数。程序员期望的典型操作是读取一行，就像高级语言函数（如C++ *getline()*函数）提供的那样。事先不知道该行可能有多少字符。这使得低级文件读取操作更加困难，因为需要读取的确切字符数。

我们，作为人类，对LF字符（行结束符）赋予特殊的意义和重要性。在计算机中，LF只是一个ASCII字符，与其他字符没有区别。我们将文件视为一系列的行，但在计算机中，文件只是一系列连续的字节。

读取交互式输入所用的过程是逐个字符读取。这可以用来解决事先不知道一行有多少个字符的问题。就像处理交互式输入一样，程序可以逐个读取，直到找到换行符（LF）然后停止。

这样一个过程从文件中读取大量信息将会非常缓慢。交互式输入期望用户输入字符。计算机很容易跟上人类。即使是打字最快的打字员也无法超过计算机（假设代码编写得高效）。此外，交互式输入通常限于相对较小的量。从文件输入不需要用户操作，并且通常不限于小量的数据。对于足够大的文件，单字符I/O将会非常慢。*Note*，测试和量化性能差异留作练习。

为了解决这个问题，输入数据将被缓冲。在这种情况下，我们将读取文件的一部分，例如100,000个字符，到一个大数组中，这个数组被称为缓冲区或输入缓冲区。当需要“一行”文本时，该行将从输入缓冲区中获取。第一次请求行时，将读取文件并填充缓冲区。之后

49 For more information, refer to: http://en.wikipedia.org/wiki/Direct_memory_access

文件读取完成，行从缓冲区返回（包括并止于换行符）。连续调用获取下一行是通过从缓冲区获取字符来满足的，无需再次读取文件。这可以一直发生，直到缓冲区中的所有字符都被返回，此时需要再次读取文件。当没有更多字符时，输入完成，程序可以终止。

此过程有助于提高性能，但更为复杂。高级语言函数，如*getline*，将此复杂性隐藏于用户。当在低级工作时会需要我们自己编写获取下一行的函数。

了解性能下降的原因对于完全理解为什么缓冲可以提高性能很重要。

17.2 缓冲算法

第一步在开发算法时是理解问题。我们将假设文件已经打开并可用于读取。对于我们的缓冲区问题，我们希望向调用例程提供一个 *myGetLine()* 函数。

The routine might be called as follows:

状态 = myGetLine(fileDescriptor, textLine, MAXLEN);

文件打开和相关错误检查是必需的，但将单独处理，并在第 13 章“系统服务”中概述。一旦获得文件描述符，就可以将其提供给 *myGetLine()* 函数。这里将其显示为明确的参数以提高清晰度。

如您可能已经知道，没有特殊的文件结束代码或字符。我们必须根据实际读取的字符数来推断文件结束。文件读取系统服务将返回实际读取的字符数。如果进程请求读取100,000个字符，而读取的少于100,000个，则已到达文件末尾，并且已读取所有字符。进一步尝试读取文件将生成错误。虽然识别文件结束和文件最后需要读取的时间很容易，但还需要处理缓冲区中的剩余字符。因此，尽管程序可能知道已找到文件结束，但程序必须继续，直到缓冲区耗尽。

调用例程不应需要了解有关缓冲、缓冲区管理或文件操作的任何细节。如果文件中的文本行可用，*myGetLine()*函数将返回文件中的行并返回TRUE状态。行通过传递的参数返回。一个用于文本行最大长度的参数

也提供以确保文本行数组不会被覆盖。如果一行不可用，可能是由于所有行都已返回或发生不可恢复的读取错误，函数应返回FALSE。如果存储介质离线（驱动器被移除）、文件从另一个控制台删除或发生硬件错误，可能会发生不可恢复的读取错误。此类错误将使继续读取变得不可能。在这种情况下，将显示错误消息并返回FALSE，这将使程序停止（结果不完整）。

通常，该函数会从大缓冲区读取字符，包括LF，并将它们放置在行缓冲区中，如上述示例调用中名为***textLine***的缓冲区。

这可以在缓冲区中有字符时轻松完成。需要额外一步来确保缓冲区有字符。如果缓冲区为空，则需要读取文件，无论是由于第一次调用还是由于缓冲区中的所有字符都已返回。可以通过将缓冲区中下一个要读取的字符的当前位置与缓冲区大小进行比较来检查是否有字符可用。当前位置不允许超过缓冲区大小（否则我们将访问缓冲区数组末尾之后的地址）。当缓冲区完全填满时，这相当直接。缓冲区可能由于文件大小小于缓冲区大小或一系列文件读取后的最后一个文件读取而部分填满。为了解决这个问题，必须根据实际读取的字符数设置缓冲区的末尾，最初设置为缓冲区大小。当文件中的字符数是缓冲区大小的精确倍数时，存在一个特殊情况。如果发生这种情况，返回的读取字符数是0。需要额外的检查来处理这种情况。

现在，问题及其所有相关细微问题都已理解，我们可以采取下一步开发算法。一般来说，这个过程通常是迭代的。也就是说，首先开发一个初步方案，然后进行审查和改进。这会多次发生，直到获得一个全面的算法。尽管有些人试图快速完成这一步，但这是一种错误。在算法开发步骤上节省的几分钟意味着额外的调试小时。

基于此早期示例，传入的参数包括；

; 文件描述符→打开文件的文件描述符，; 作为读取系统服务所需的
； 文本行→文本行的起始地址；最大长度→文本行数组最大长度

一些变量是必需的，如下定义：

; 缓冲区大小→参数，用于缓冲区大小; currIndex→缓冲区中当前位置的索引，初始设置为 BUFFER_SIZE; buffMaximum→缓冲区的当前最大大小，初始设置为 BUFFER_SIZE; eofFlag→标记是否找到文件末尾的布尔值，初始设置为 false

基于对问题的理解，可能存在多种不同的算法方法。使用参数和局部变量，以下是一种可供参考的方法。

; myGetLine(fileDescriptor, textLine, maxLength) { ; repeat { ; if current index ≥ buffer maximum ; read buffer (buffer size) ; if error ; handle read error ; display error message ; exit routine (with false) ; reset pointers ; if chars read **<** characters request read ; set eofFlag **=** TRUE ; get one character from buffer at current index ; place character in text line buffer ; increment current index ; if character is LF ; exit with true ; } ; }

此算法概述未验证文本行缓冲区未被覆盖，也未处理文件大小是缓冲区大小的精确倍数的情况。优化、实现和测试算法留作读者练习。

此算法将需要静态声明的变量。由于需要在连续调用之间保持信息，因此不能使用堆栈动态变量。

变量和算法作为程序注释提供。算法是程序文档中最重要的部分，不仅有助于编写

代码但在调试过程中。无论你认为通过跳过文档能节省多少时间，在调试时都会花费更多更多的时间。

17.3 练习

以下是本章的一些测验问题和基于此章节的建议项目。

17.3.1 测验问题

以下是本章的一些测验问题。

1) Linux和Windows的行结束符或字符是什么？2) 根据本章的解释，什么是I/O缓冲？3) 在高级语言中，I/O缓冲例程位于何处？4) 隐藏I/O缓冲复杂性对程序员有什么优势？5) 与逐字符读取相比，执行I/O缓冲的关键优势是什么？6) 为什么使用文件读取系统服务从文件中读取一行文本很困难？7) 从内存层次结构的角度来看，为什么缓冲是有利的？8) 从系统开销的角度来看，为什么缓冲是有利的？9) 为什么文件读取缓冲算法需要静态声明的变量？10) 程序如何识别文件结束？11) 提供一个可能的原因，即文件已成功打开，但文件读取请求仍可能返回错误。12) 当文件大小是缓冲区大小的精确倍数时，必须做什么？13) 为什么将文本行的最大长度作为参数传递？14) 所提出的算法如何确保第一次调用*myGetLine()* 将读取缓冲区？

17.3.2 建议项目

以下是本章的一些建议项目。

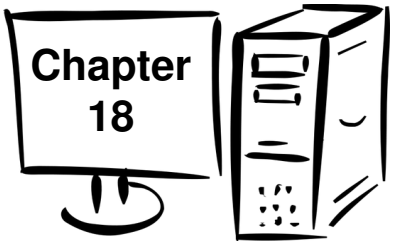
1) 优化所提出的算法以解决最大文本行检查和文件大小是缓冲区大小的偶数倍的可能性。这包括正式化变量名称和循环结构（许多可能的选择）。2) 创建一个简单的主程序，通过打开文件、调用 *myGetLine()* 函数和将行显示到控制台来测试 *myGetLine()* 函数。使用一系列不同大小（包括比所选缓冲区大小小、大和大大）的文件测试程序。捕获程序输出，并将捕获的输出与原始文件进行比较，以确保程序正确。

3) 创建一个程序，该程序将从命令行读取两个文件名，从第一个文件中读取行，添加行号，并将修改后的行（带有行号）写入第二个文件。例如，您的添加行程序可能以以下方式启动：

```
./添加行 inFile.txt newFile.txt
```

在 *inFile.txt* 存在且包含标准 ASCII 文本的地方。如果 *inFile.txt* 文件不存在，应生成错误并终止程序。程序应打开/创建文件 *newFile.txt* 并将带有行号的行写入 *newFile.txt* 文件。输出文件应被创建，如果存在旧版本则删除。使用文本编辑器验证输出文件中的行号是否正确跟踪。

To err is human; to make a real mess, you need a computer.



18.0 浮点指令

本章节为x86-64浮点指令子集提供了一个基本概述。仅涵盖最基本的指令。

文本侧重于x86-64浮点运算，这与32位浮点运算不同。

指示按以下顺序呈现:

- 数据移动
- 整数/浮点数转换指令
- 算术指令
- 浮点控制指令

本文本中涵盖的指令完整列表位于附录B中供参考。

应注意的是，浮点算术运算不需要使用特定的寄存器，并且不会改变类型（大小）。这使得浮点指令更容易使用。

18.1 浮点值

浮点值通常表示为单精度（32位）或双精度（64位）。在C和C++中，单精度浮点变量通常声明为***float***类型，而双精度浮点变量声明为***double***类型。如以下各节所述，汇编语言指令将使用s（小写字母S）修饰符来指代单精度，使用d（小写字母D）修饰符来指代双精度。

18.2 浮点寄存器

存在一组专用寄存器，称为XMM寄存器，用于支持浮点运算。浮点运算必须使用浮点寄存器。XMM寄存器在较晚的处理器上为128位和256位。最初，我们只会使用低32位或64位。

有16个XMM寄存器，命名为**xmm0**至**xmm15**。请参阅第2章以了解CPU寄存器的解释和总结。

18.3 数据移动

通常，必须将数据移动到CPU浮点寄存器中才能进行操作。一旦计算完成，结果可以从寄存器复制并放置到变量中。示例程序中有一些简单的公式执行这些步骤。这些基本数据移动操作是通过移动指令执行的。

移动指令的一般形式是：

```
movss <dest>, <src> movsd <d
est>, <src>
```

对于 **movss** 指令，单个 32 位源操作数被复制到目标操作数。对于 **movsd** 指令，单个 64 位源操作数被复制到目标操作数。源操作数的值保持不变。目标操作数和源操作数必须与指令的正确大小（32 位或 64 位）匹配。两个操作数都不能是立即数。两个操作数不能都是内存，但其中一个可以是。如果需要内存到内存的操作，必须使用两个指令。

指令操作将一个值，使用低32或64位，加载到或从寄存器中。其他指令操作需要加载多个值。

浮点移动指令总结如下：

Instruction	Explanation
movss <dest>, <src>	Move scaler single precision floating point value. Copy 32-bit source operand to the 32-bit destination operand. *Note 1*, both operands cannot be memory. *Note 2*, operands cannot be an immediate.

Instruction	Explanation

Examples:	`movss xmm0, dword [x]`
	`movss dword [fltSVar], xmm1`
	`movss xmm3, xmm2`

`movsd <dest>, <src>`	Move scaler double precision floating point value. Copy 64-bit source operand to the 64-bit destination operand. *Note 1*, both operands cannot be memory. *Note 2*, operands cannot be an immediate.

Examples:	`movsd xmm0, qword [y]`
	`movsd qword [fltDVar], xmm1`
	`movsd xmm3, xmm2`

更完整的指令列表位于附录B中。

例如，假设以下数据声明：

fSVar1 dd 3.14 fSVar2 dd 0.0 fD
Var1 dq 6.28 fDVar2 dq 0.0

执行以下基本操作：

fSVar2 = fSVar2 ; 单精度变量 fDVar2 = fDVar1 ; 双精度变量

以下说明可用于：

```
movss       xmm0, dword [fSVar1]
movss       dword [fSVar2], xmm0      ; fSVar2 = fSVar1

movsd       xmm1, qword [fDVar1]
movsd       qword [fDVar2], xmm1      ; fDVar2 = fDVar1

movss       xmm2, xmm0                ; xmm2 = xmm0 (32-bit)
movsd       xmm3, xmm1                ; xmm3 = xmm1 (64-bit)
```

对于某些指令，包括上述指令，可以省略显式类型指定（例如，*byte*，*word*，*dword*，*qword*），因为另一个操作数将明确定义大小。这只是为了保持一致性以及良好的编程实践。

18.4 整数/浮点数转换指令

如果浮点运算过程中需要整数值，则必须将整数转换为浮点值。如果一系列计算需要单精度和双精度浮点值，则必须将它们转换为单精度或双精度，以便在一致的大小/类型上执行操作。

参考第3章以获取关于浮点值表示细节的更详细解释。假设读者理解表示细节并认识到在执行浮点运算之前确保格式一致性的要求。

这些基本数据转换操作使用转换指令执行。

浮点转换指令总结如下:

Instruction	Explanation
cvtss2sd <RXdest>, <Rxsrc>	Convert scaler single precision to double precision floating point value. Convert 32-bit floating-point source operand to the 64-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtss2sd xmm0, dword [fltSVar]** **cvtss2sd xmm3, xmm2**
cvtsd2ss <RXdest>, <RXsrc> **cvttsd2ss <RXdest>, <RXsrc>**	Convert scaler double precision to single precision floating point value with loss of precision. Convert 64-bit floating-point source operand to the 32-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtsd2ss xmm0, qword [fltDVar]** **cvtsd2ss xmm1, xmm5**

Instruction	Explanation
cvtss2si <reg>, <Rxsrc> **cvttss2si <reg>, <RXsrc>**	Convert scaler single precision to integer value. Convert 32-bit floating-point source operand to the 32-bit or 64-bit integer destination operand. The **cvtss2si** performs rounding and the **cvtss2si** performs truncation. *Note 1*, destination operand must be register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtss2si eax, xmm0** **cvtss2si rbx, dword [fltSVar]** **cvttss2si eax, xmm0** **cvttss2si rcx, dword [fltSVar]**
cvtsd2si <reg>, <Rxsrc> **cvttsd2si <reg>, <RXsrc>**	Convert scaler double precision to integer value. Convert 64-bit floating-point source operand to the 32-bit or 64-bit integer destination operand. The **cvtsd2si** performs rounding and the **cvtsd2si** performs truncation. *Note 1*, destination operand must be register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtsd2si xmm1, xmm0** **cvtsd2si eax, xmm0** **cvttsd2si rbx, xmm0** **cvttsd2si eax, qword [fltDVar]**
cvtsi2ss <RXdest>, <src>	Convert integer value to single precision floating point value. Convert 32-bit or 64-bit integer source operand to the 32-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**cvtsi2ss xmm0, eax** **cvtsi2ss xmm0, ebx** **cvtsi2ss xmm0, dword [fltDVar]**

Instruction	Explanation
`cvtsi2sd  <RXdest>, <src>`	Convert integer value to double precision floating point value. Convert 32-bit integer source operand to the 64-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	`cvtsi2sd  xmm0, eax` `cvtsi2sd  xmm0, dword [fltDVar]`

更完整的指令列表位于附录B中。

18.5 浮点算术指令

浮点算术指令执行加、减、乘、除等算术运算，针对单精度或双精度浮点值。以下章节介绍了基本的算术运算。

18.5.1 浮点数加法

浮点加法指令的一般形式如下：

addss **<**RXdest**>**, **<**src**>** addsd **<**RXdest**>**, **<**src**>**

操作如下：

<RXdest**> = <**RXdest**> + <**src**>**

具体来说，源操作数和目标操作数相加，结果放置在目标操作数中（覆盖之前的值）。目标操作数必须是一个浮点寄存器。源操作数不能是立即数。源操作数的值保持不变。目标操作数和源操作数必须具有相同的大小（双字或四字）。如果需要执行内存到内存的加法操作，必须使用两条指令。

例如，假设以下数据声明：

```
fSNum1      dd      43.75
fSNum2      dd      15.5
fSAns       dd      0.0

fDNum3      dq      200.12
fDNum4      dq      73.2134
fDAns       dq      0.0
```

执行以下基本操作：

fSAns **=** fSNum1 **+** fSNum2 fDA
ns **=** fDNum3 **+** fDNum4

以下说明可用于：

; fSAns **=** fSNum1 **+** fSNum2 movss xmm0, d
word [fSNum1] addss xmm0, dword [fSNum
2] movss dword [fSAns], xmm0 ; fDAns **=** fD
Num3 **+** fDNum4 movsd xmm0, qword [fDN
um3] addsd xmm0, qword [fDNum4] movsd
qword [fDAns], xmm0

对于某些指令，包括上述指令，可以省略显式的类型指定（例如，*dword*，*qword*），因为其他操作数或指令本身明确定义了大小。这包括为了保持一致性和良好的编程实践。

浮点加法指令总结如下：

指令说明 addss **<**RXdest**>**, **<**src**>** 添加单精度浮点值。 将两个32位浮点操作数
（(<RXdest> + <src>）相加，并将结果放入 <RXdest>（覆盖之前值）。*Note 1*，目的操作数必须是一个浮点寄存器。*Note 2*，源操作数不能是立即数。

Instruction	Explanation
Examples:	`addss  xmm0, xmm3` `addss  xmm5, dword [fSVar]`
`addsd  <RXdest>, <src>`	Add double precision floating point values. Add two 64-bit floating-point operands, (<**RXdest**> + <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	`addsd  xmm0, xmm3` `addsd  xmm5, qword [fDVar]`

更多　完整指令列表位于Appen　　10 B.

18.5.2 浮点数减法

浮点数减法指令的一般形式如下：

subss　<RX目标>，<源> subsd
<RX目标>，<源>

操作如下：

<RXdest> = <RXdest> - <src>

具体来说，源操作数和目标操作数相减，结果放置在目标操作数中（覆盖之前的值）。目标操作数必须是一个浮点寄存器。源操作数不能是立即数。源操作数的值保持不变。目标操作数和源操作数的大小必须相同（双字或四字）。如果需要内存到内存的减法操作，必须使用两条指令。

例如，假设以下数据声明：

```
fSNum1      dd      43.75
fSNum2      dd      15.5
fSAns       dd      0.0
```

```
fDNum3      dq      200.12
fDNum4      dq      73.2134
fDAns       dq      0.0
```

执行以下基本操作：

fSAns = fSNum1 - fSNum2 fDAn
s = fDNum3 - fDNum4

以下说明可用于：

; fSAns = fSNum1 - fSNum2 movss xmm0, d
word [fSNum1] subss xmm0, dword [fSNum2
] movss dword [fSAns], xmm0

; fDAns = fDNum3 - fDNum4 movsd xmm0,
qword [fDNum1] subsd xmm0, qword [fDNu
m2] movsd qword [fDAns], xmm0

对于某些指令，包括上述指令，可以省略显式的类型指定（例如，*dword*，*qword*），因为其他操作数或指令本身明确定义了大小。这包括为了保持一致性和良好的编程实践。

浮点数减法指令总结如下：

Instruction	**Explanation**
subss <RXdest>, <src>	Subtract single precision floating point values. Subtract two 32-bit floating-point operands, (**<RXdest>** - **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**subss xmm0, xmm3** **subss xmm5, dword [fSVar]**

Instruction	Explanation
`subsd  <RXdest>, <src>`	Subtract double precision floating point values. Subtract two 64-bit floating-point operands, (<**RXdest**> - <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	`subsd  xmm0, xmm3` `subsd  xmm5, qword [fDVar]`

更完整的指令列表位于附录B中。

18.5.3 浮点数乘法

浮点乘法指令的一般形式如下：

mulss <RXdest>, <src> mulsd <RXdest>, <src>

操作如下：

<RXdest> = <RXdest> * <src>

具体来说，源操作数和目标操作数相乘，结果放置在目标操作数中（覆盖之前的值）。目标操作数必须是一个浮点寄存器。源操作数不能是立即数。源操作数的值保持不变。目标操作数和源操作数必须具有相同的大小（双字或四字）。如果需要内存到内存的乘法操作，必须使用两条指令。

例如，假设以下数据声明：

```
fSNum1      dd      43.75
fSNum2      dd      15.5
fSAns       dd      0.0

fDNum3      dq      200.12
fDNum4      dq      73.2134
fDAns       dq      0.0
```

执行以下基本操作：

```
fSAns = fSNum1 * fSNum2 fDA
ns = fDNum3 * fDNum4
```

以下说明可用于：

```
; fSAns = fSNum1 * fSNum2 movss xmm0, d
word [fSNum1] mulss xmm0, dword [fSNum
2] movss dword [fSAns], xmm0


; fDAns = fDNum3 * fDNum4 movsd xmm0,
qword [fDNum3] mulsd xmm0, qword [fDNu
m4] movsd qword [fDAns], xmm0
```

对于某些指令，包括上述指令，可以省略显式的类型指定（例如，*dword*，*qword*），因为其他操作数或指令本身明确定义了大小。这包括为了保持一致性和良好的编程实践。

浮点乘法指令总结如下:

Instruction	Explanation
mulss <RXdest>, <src>	Multiple single precision floating point values. Multiply two 32-bit floating-point operands, (**<RXdest>** * **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**mulss xmm0, xmm3** **mulss xmm5, dword [fSVar]**
mulsd <RXdest>, <src>	Multiply double precision floating point values. Multiply two 64-bit floating-point operands, (**<RXdest>** * **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**mulsd xmm0, xmm3** **mulsd xmm5, qword [fDVar]**

更完整的指令列表位于附录B中。

18.5.4 浮点除法

浮点除法指令的一般形式如下:

divss **<**RXdest**>**, **<**src**>** divsd **<**RXdest**>**, **<**src**>**

操作如下:

<RXdest**> = <**RXdest**> / <**src**>**

具体来说，源操作数和目标操作数被分开，结果放置在目标操作数中（覆盖之前的值）。目标操作数必须

是一个浮点寄存器。源操作数不能是立即数。源操作数的值保持不变。目的操作数和源操作数必须具有相同的大小（双字或四字）。如果需要执行内存到内存的除法操作，必须使用两条指令。

F例如，假设以下数据声明 ns:

```
fSNum1      dd      43.75
fSNum2      dd      15.5
fSAns       dd      0.0

fDNum3      dq      200.12
fDNum4      dq      73.2134
fDAns       dq      0.0
```

执行以下基本操作:

fSAns = fSNum1 / fSNum2 fDAn
s = fDNum3 / fDNum4

以下说明可用于:

```
; fSAns = fSNum1 / fSNum2 movss xmm0, d
word [fSNum1] divss xmm0, dword [fSNum2
] movss dword [fSAns], xmm0


; fDAns = fDNum3 / fDNum4 movsd xmm0,
qword [fDNum3] divsd xmm0, qword [fDNu
m4] movsd qword [fDAns], xmm0
```

对于某些指令，包括上述指令，可以省略显式的类型指定（例如，*dword*，*qword*），因为其他操作数或指令本身明确定义了大小。这包括为了保持一致性和良好的编程实践。

浮点除法指令总结如下:

Instruction	Explanation
divss <RXdest>, <src>	Divide single precision floating point values. Divide two 32-bit floating-point operands, (**<RXdest>** / **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**divss xmm0, xmm3** **divss xmm5, dword [fSVar]**
divsd <RXdest>, <src>	Divide double precision floating point values. Divide two 64-bit floating-point operands, (**<RXdest>** / **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**divsd xmm0, xmm3** **divsd xmm5, qword [fDVar]**

更完整的指令列表位于附录B中。

18.5.5 浮点平方根

浮点平方根指令的一般形式如下:

sqrtss **<**RXdest**>**, **<**src**>** sqrtsd **<**RX dest**>**, **<**src**>**

操作如下:

<dest> = $\sqrt{\text{<src>}}$

具体来说，源操作数的平方根被放置在目标操作数中（覆盖之前的值）。目标操作数必须是一个浮点寄存器。源操作数不能是立即数。源操作数的值

操作数保持不变。目标操作数和源操作数必须具有相同的大小（双字或四字）。如果需要内存到内存的加法操作，必须使用两条指令。

例如，假设以下数据声明：

```
fSNum1      dd      1213.0
fSAns       dd      0.0

fDNum3      dq      172935.123
fDAns       dq      0.0
```

执行以下基本操作：

$$\texttt{fSAns} = \sqrt{\texttt{fSNum1}}$$
$$\texttt{fDAns} = \sqrt{\texttt{fSNum3}}$$

以下说明可用于：

```
; fSAns = sqrt (fSNum1) sqrtss xmm0, dword
[fSNum1] movss dword [fSAns], xmm0 ; fD
Ans = sqrt(fDNum3) sqrtsd xmm0, qword [fD
Num3] movsd qword [fDAns], xmm0       翻译文
本：  ; fSAns = 平方根 (fSNum1) sqrtss xmm
0, dword [fSNum1] movss dword [fSAns], x
mm0 ; fDAns = 平方根(fDNum3) sqrtsd xm
m0, qword [fDNum3] movsd qword [fDAns],
xmm0
```

对于某些指令，包括上述指令，可以省略显式的类型指定（例如，*dword*，*qword*），因为其他操作数或指令本身明确定义了大小。这包括为了保持一致性和良好的编程实践。

浮点加法指令总结如下：

Instruction	Explanation
`sqrtss <RXdest>, <src>`	Take the square root of the 32-bit floating-point source operand and place the result in destination operand (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.

Instruction	Explanation
Examples:	`sqrtss  xmm0, xmm3` `sqrtss  xmm7, dword [fSVar]`
`sqrtsd  <RXdest>, <src>`	Take the square root of the 64-bit floating-point source operand and place the result in destination operand (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	`sqrtsd  xmm0, xmm3` `sqrtsd  xmm7, qword [fDVar]`

更多　完整指令列表位于Appen　　10 B.

18.6 浮点控制指令

控制指令指的是诸如 IF 语句和循环之类的编程结构。第 7 章中描述的整数比较指令 cmp 对于浮点数将不起作用。

浮点比较指令比较两个浮点值。与整数比较类似，比较的结果存储在rFlag寄存器中，且两个操作数均未改变。比较后立即通过条件跳转指令访问rFlag寄存器以确定结果。虽然所有浮点比较都是带符号的，但使用无符号条件跳转（ja/jae/jb/jbe）。程序标签（即条件跳转的目标）相同。

有两种浮点比较形式，有序和无序。有序浮点比较可能会引发多个异常。无序浮点比较只能引发一个异常，即一个 *S-NaN*（表示非数字），更通用地称为 *NaN*（非数字），如第 3 章数据表示中所述。

GNU C/C++ 编译器青睐无序浮点比较指令。由于它们相似，本文将仅关注无序版本。

18.6.1 浮点比较

浮点比较指令的一般形式如下：

```
ucomiss  <RXsrc>, <src> ucomisd
<RXsrc>, <src>
```

<RXsrc> 和 <src> 作为浮点值进行比较，并且必须具有相同的大小。比较的结果存储在 rFlag 寄存器中。两个操作数均不改变。<RXsrc> 操作数必须是 xmm 寄存器之一。<src> 寄存器可以是 xmm 寄存器或内存位置，但不能是立即值。可以使用其中一个无符号条件跳转指令来读取 rFlag 寄存器。

条件控制指令包括跳转相等（je）和跳转不等（jne）。无符号条件控制指令包括基本的比较操作集；跳转小于（jb），跳转小于等于（jbe），跳转大于（ja），和跳转大于等于（jae）。

签名条件指令的一般形式及其解释注释如下：

```
je   <label> ; 如果 <op1> == <op2> jne   <label> ; 如果 <op1> != <op2> jb
<label> ; 无符号，如果 <op1> < <op2> jbe   <label> ; 无符号，如果 <op1
> <= <op2> ja   <label> ; 无符号，如果 <op1> > <op2> jae   <label> ; 无
符号，如果 <op1> >= <op2>
```

例如，给定以下浮点变量的伪代码：

```
if (fltNum > fltMax) fltMax = fl
tNum;
```

给定以下数据声明：

```
fltNum dq 7.5 fltMax dq 5.25
```

然后，假设值在程序中适当更新（未显示），可以使用以下指令：

```
movsd xmm1, qword [fltNum] ucomisd xmm1, qword [fltMax] ; 如果 fltNum {v*}
大于等于 fltMax
```

jbe notNewFltMax ; 跳过设置新最大值 movsd qword [fltMax], xmm1 notNewFltMax:

与整数比较一样，浮点数比较和条件跳转提供了跳转或不跳转的功能。因此，如果原始 IF 语句的条件为假，则不应执行更新 fltMax 的代码。因此，当条件为假时，为了跳过执行，条件跳转将立即跳转到要跳过的代码（不执行）之后的标记。虽然这个例子中只有一行，但可能有多个代码行。

一个更复杂的例子可能如下所示：

```
if (x != 0.0) { ans = x / y; errFl
g = FALSE; } else { errFlg =
TRUE; }
```

此基本的比较和条件跳转不提供 IF-ELSE 结构。它必须创建。假设 ***x*** 和 ***y*** 变量是将在程序执行期间设置的带符号双字，并且以下声明：

```
TRUE equ 1 FALSE equ 0 fltZero
dq 0.0 x dq 10.1 y dq 3.7 ans dq 0.
0 errFlg db FALSE
```

以下c ode 可以用来实现上述 IF-ELSE 声明。

```
movsd xmm1, qword [x] ucomisd xmm1, qwor
d [fltZero]                                  ; 如果语句
je doElse divsd xmm1, qword [y] movsd dwor
d [ans], eax mov byte [errFlg], FALSE jmp sk
pElse
```

```
doElse: 移
动                  byte [errFlg], 真
skpElse:
```

浮点数比较可能非常棘手，因为浮点数表示的不精确性和舍入误差。例如，0.1这个值加10次应该等于1.0。然而，实现一个执行这个求和并检查结果的程序，将会显示；

$$\sum_{i=1}^{10} \mathbf{0.1} \neq \mathbf{1.0}$$

这可能会让没有经验的程序员感到非常困惑。

有关浮点数细节的更多信息，请参阅流行的文章
What Every Computer Scientist Should Know About Floating-Point Arithmetic[50]。

浮点比较指令总结如下:

Instruction	**Explanation**
ucomiss <RXsrc>, <src>	Compare two 32-bit floating-point operands, (**<RXsrc>** + **<src>**). Results are placed in the **rFlag** register. Neither operand is changed. *Note 1*, <RXsrc> operands must be a floating-point register. *Note 2*, source operand may be a floating-point register or memory, but not be an immediate.
Examples:	**ucomiss xmm0, xmm3** **ucomiss xmm5, dword [fSVar]**
ucomisd <RXsrc>, <src>	Compare two 64-bit floating-point operands, (**<RXsrc>** + **<src>**). Results are placed in the **rFlag** register. Neither operand is changed. *Note 1*, <RXsrc> operands must be a floating-point register. *Note 2*, source operand may be a floating-point register or memory, but not be an immediate.
Examples:	**ucomisd xmm0, xmm3** **ucomisd xmm5, qword [fSVar]**

50 查看：http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

更完整的指令列表位于附录B中。

18.7 浮点调用约定 s

标准调用约定在第12章“函数”中详细说明，仍然完全适用。本节讨论调用浮点函数时浮点寄存器的使用。

当使用浮点寄存器时，在浮点函数调用过程中，没有任何寄存器被保留。

前八个（8）浮点参数通过浮点寄存器xmm0 – xmm7传递。任何额外的参数都按照第12章“函数”中描述的方式以反向顺序放置在堆栈上。返回浮点值的函数将在xmm0中返回结果。

由于没有保留任何浮点寄存器，代码必须仔细编写。

18.8 示例程序，总和与平均值

这是一个简单的汇编语言程序，用于计算浮点值列表的总和和平均值。

; 浮点数示例程序 ; ********** *** section .data

; ----- ; 定义常量。

NULL 等于 0 ; 字符串结束 TRUE 等于 1 FALSE 等于 0 EXIT_SUCCESS 等于 0 ; 成功操作 SYS_exit 等于 60 ; 终止的系统调用代码 ; -----

fltLst dq 21.34, 6.15, 9.12, 10.05, 7.75 dq 1.44, 14.50, 3.32, 75.71, 11.87

```
                dq 17.23, 18.25, 13.65, 24.24, 8.88
长度            dd 15
lstSum          dq 0.0
lstAve          dq 0.0

; ********************************************************** 部分 .text 全局
_start _start:


; ----- ; 循环查找浮点数总和。

    mov ecx, [length]mov rbx, fltLstmov rsi,
    0movsd xmm1, qword [lstSum]


sumLp: movsd xmm0, qword [rbx+rsi*8] ; 获取 fltLst[i] addsd xmm1, xmm0 ; 更新 sum inc
rsi ; i++ 循环 sumLp movsd qword [lstSum], xmm1 ; 保存 sum



计算整个列表的平均值。

    cvtsi2sd xmm0, dword [length] cvtsd2si dwor
    d [length], xmm0 divsd xmm1, xmm0 movsd
    qword [lstAve], xmm1

; ----- ; 完成，终止程序。

last:mov rax, SYS_exit mov rbx, EXIT_S
UCCESS
                                                    ; 退出/成功
    系统调用
```

调试器可以用来检查结果并验证程序的正确执行。

18.9 示例程序，绝对值

这是一个简单的汇编语言程序示例，用于找到浮点数的绝对值，以演示浮点数比较。回想一下，如果一个值是负数，它必须变为正数，如果值已经是正数，则无需进行任何操作。

```
; 浮点数绝对值示例

节数 .data

; ----- ; 定义常量。


TRUE 等于 1 FALSE 等于 0 SUCCESS 等于 0 ; 成功操作 SYS_exit 等于 60 ; 终止
调用代码



; ----- ;
  定义一些测试变量。

dZero dq 0.0  dNegOne dq -1.0  fltVal dq
-8.25


; ********************************************************
章节 .text 全局 _start _
start:

执行绝对值函数于 flt1


      movsd xmm0, qword [fltVal] ucomisd xmm0,
      qword [dZero] jae isPos
```

```
    mulsd xmm0, qword [dNegOne] movsd qword
[fltVal], xmm0 isPos:


; ----- ; 完成，终止程序。


last:mov rax, SYS_exit mov rbx, EXIT_S
UCCESS
                                                    ; 退出/成功
    系统调用
```

在这个例子中，|fltVal|的最终结果被保存到了内存中。根据上下文，这可能不是必需的。

18.10 练习

以下是本章的一些测验问题和基于此章节的建议项目。

18.10.1 测验问题

以下是本章的一些测验问题。

1) 列出浮点寄存器。2) 单精度浮点值和双精度浮点值分别占用多少字节？3) 解释为什么0.1加10次不等于1.0。4) 值返回的浮点函数（如sin(x)）的结果在哪里返回？5) 对于值返回的浮点函数，哪些浮点寄存器必须在函数调用期间保留？

18.10.2 建议项目

以下是本章的一些建议项目。

1) 实现示例程序以找到浮点值列表的总和和平均值。使用调试器执行程序并验证最终结果。

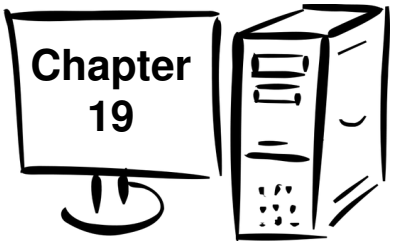
2) 将浮点数的绝对值函数实现为两个宏，一个为fAbsf，用于32位浮点数值，另一个为fAbsd，用于64位浮点数值。创建一个简单的main程序，使用每个宏在各个不同的值上各三次。使用调试器执行程序并验证结果。

3) 实现一个执行求和的程序：

$$\sum_{i=1}^{10} \mathbf{0.1}$$

比较求和结果与值1.0，如果求和结果等于1.0，则显示消息“相同”，如果求和结果不等于1.0，则显示消息“不相同”。如有需要，使用调试器进行程序调试。在工作时，不使用调试器执行程序，并验证预期结果是否显示在控制台。

#define QUESTION ((bb) || !(bb))
- Shakespeare



19.0 并行处理

在计算领域，并行处理[51]，或更普遍地并发[52]，
指多个看似同时执行的过程。

在广义上，并发意味着多个不同的（不一定相关的）进程同时进行。这可以通过多种方式实现。例如，进程A可以在核心0上执行，而进程B同时可以在核心1上执行，从而并行执行。另一种可能性是，进程A和B可以在单个核心上交错执行，其中进程A可能执行一段时间，然后暂停，此时进程B执行一段时间，然后暂停，此时进程A恢复执行。这会一直持续到其中一个或两个都完成。如果这种交错执行经常且足够快，那么它们似乎是在同时执行。

并行处理这一术语意味着进程是同时执行的。这些进程可能是不相关的，或者作为一个协调单位共同解决单个复杂问题。

本章概述了将并行处理应用于单个问题的基本方法。假设一个大问题可以分解成各种子问题，并且这些子问题可以独立执行。子解决方案将被汇集起来，为问题提供一个最终解决方案。如果子问题可以同时执行，最终解决方案可能比子问题依次（一个接一个）执行要快得多。实现并行的潜在回报是显著的，但挑战也同样巨大。不幸的是，并非所有问题都可以轻易地分解成这样的子问题。

51 For more information, refer to: http://en.wikipedia.org/wiki/Parallel_processing
52 For more information, refer to: http://en.wikipedia.org/wiki/Concurrency_(computer_science)

并行处理的基本方法包括分布式计算[53]和多进程[54]（也称为线程计算）。每种方法都侧重于与共享内存多进程相关的技术问题。学习创建并行算法的更大主题超出了本文的范围。

19.1 分布式计算

分布式计算或分布式处理是指将一个大问题分解成多个子问题，并在通过网络连接的不同计算机上执行子问题的通用概念。通常，一个主服务器或主节点会将子问题分配给各种可用的计算节点。完成时，计算节点会将中间结果发送回主节点。如有需要，主节点可以向计算节点发送额外的子问题。主节点还将跟踪和将中间结果组合成最终解决方案。实际上如何执行这一过程的细节差异很大，并且与问题的具体细节直接相关。

有许多大规模的分布式计算项目示例{v*}。其中一个这样的项目是Folding@home[56]，这是一个用于疾病研究的大型分布式计算项目，它模拟蛋白质折叠、计算药物设计和其他类型的分子动力学。该项目使用了超过10万台不同个人电脑的空闲处理资源，这些电脑由安装了软件的志愿者拥有。

这种方法具有扩展到非常大量的分布式计算机的优势。其缺点与网络相关的通信限制有关。

19.2 多进程

如第2章“架构概述”中所述，大多数当前CPU芯片都包含多个核心。CPU核心都平等地访问主内存资源。多处理是一种并行处理形式，特指使用多个核心来执行多个进程的并发执行。

53 For more information, refer to: http://en.wikipedia.org/wiki/Distributed_computing
54 For more information, refer to: http://en.wikipedia.org/wiki/Multiprocessing
55 For more information, refer to: http://en.wikipedia.org/wiki/List_of_distributed_computing_projects
56 For more information, refer to: http://en.wikipedia.org/wiki/Folding@home

关注一个单一的大型项目，一个主要或初始过程可能会生成子过程，称为线程[57]。当初始过程由系统加载器[58]启动并放置在内存中时，操作系统会创建一个新的进程，包括所需的操作系统数据结构、内存分配和页面/交换文件分配。线程通常被称为轻量级进程，因为它将使用初始进程的分配和地址空间，从而使其创建更快。它也将更快地终止，因为不会执行释放操作。由于线程与初始进程和任何其他线程共享内存，这为同时执行的线程之间提供了非常快速的通信潜力。它还有一个附加要求，即必须仔细协调以内存写形式进行的通信，以确保结果不会被破坏。这种问题被称为竞争条件[59]。以下章节将解决竞争条件问题。

线程方法无法扩展到分布式方法的程度。CPU芯片的核心数量有限，这限制了可以同时发生的计算数量。这个限制不适用于分布式计算。关于网络通信速度的限制不适用于线程计算。

19.2.1 POSIX 线程

POSIX 线程[60]，通常称为 pThreads，是 Ubuntu 和许多其他操作系统上广泛可用的线程库。本节中的示例将使用 pThreads 线程库应用程序程序员接口（API）来创建和连接线程。

初始或主进程在加载过程中创建。主进程可能使用 pthread_create() 库函数创建额外的线程。虽然可以创建的线程数量没有具体限制，但可用的核心数量有限，因此为线程数量设置了一个实际限制。

主进程或初始进程可以使用pthread_join()库函数查看线程是否完成。如果线程尚未完成，join函数将等待直到它完成。理想情况下，为了最大化整体并行操作，主进程应在线程或线程执行时执行其他计算，并且仅在完成其他工作后检查线程完成。

57 For more information, refer to: http://en.wikipedia.org/wiki/Thread_(computing)
58 For more information, refer to Chapter 5, Tool Chain
59 For more information, refer to: http://en.wikipedia.org/wiki/Race_condition
60 For more information, refer to: http://en.wikipedia.org/wiki/POSIX_Threads

存在其他pThread函数来处理互斥锁[61]、条件变量以及线程间的同步[62]，这些内容在本简要概述中未涉及。

应注意的是，这里没有涉及其他线程方法。

19.2.2 竞态条件

竞态条件是一个通用术语，指的是多个线程同时向同一位置写入数据的情况。线程编程通常使用高级语言进行。然而，最好在汇编语言级别完全理解竞态条件的具体原因。

一个简单的例子被提供出来，目的是故意创建一个竞争条件并详细检查这个问题。此示例在计算上没有用处。

假设我们希望计算以下公式 MAX 次，其中 MAX 是一个定义的常量。例如；

$$myValue = \left(\frac{myValue}{X} \right) + Y$$

我们可以编写一个高级语言程序，大致如下：

```
for (int i=0; i < MAX; i++)
    myValue = (myValue / X) + Y;
```

如果我们希望加快这个计算，可能是因为MAX非常大，我们可能创建两个线程函数，每个函数执行MAX/2的计算。每个线程函数将共享访问变量***myValue***、***X***和***Y***。因此，代码可能如下；

```
for (int i=0; i < MAX/2; i++)
    myValue = (myValue / X) + Y;
```

此代码将在每个线程函数中重复。这可能不明显，但假设两个线程同时执行，这将在 ***myValue*** 变量上引起竞态条件。具体来说，两个线程中的每一个都试图同时更新该变量，并且可能丢失一些对变量的更新。

61 For more information, refer to: http://en.wikipedia.org/wiki/Mutual_exclusion
62 For more information, refer to: http://en.wikipedia.org/wiki/Synchronization_(computer_science)

为了进一步简化这个例子，我们假设 ***X*** 和 ***Y*** 都设置为 1。因此，每次计算的结果将是将 ***myValue*** 增加 1。如果 ***myValue*** 初始化为 0，重复计算 MAX 次，应该将 ***myValue*** 设置为 MAX。这种简化将允许轻松验证结果。在计算上没有任何有意义的用途。

实现此功能首先需要一个主程序来创建两个线程中的每一个。然后，需要创建每个线程函数。在这个非常简单的例子中，每个线程函数都是相同的（执行 MAX/2 次迭代，其中 MAX 是一个偶数）。

假设其中一个线程函数名为 *threadFunction0()*，并且给定以下 pThread 线程数据结构；

```
pthreadID0 dd 0, 0, 0, 0, 0
```

以下代码片段将创建并启动 *threadFunction0()* 的执行。

```
; pthread_create(&pthreadID0, NULL, ; &threadFunction0, NULL);

    mov rdi, pthreadID0  mov rsi, NULL  mo
    v rdx, threadFunction0  mov rcx, NULL  c
    all pthread_create
```

未提供翻译文本

以下代码片段将检查线程函数是否完成；

```
; pthread_join (pthreadID0, NULL);  mov rdi, qword [
pthreadID0]  mov rsi, NULL  call pthread_join
```

如果线程函数尚未完成，则join调用将等待其完成。

线程函数，*threadFunction0()*，本身可能包含以下代码；

```
; ----- 全局线程函数0  threadFunction0:  ；执行 MAX / 2 次迭代以更
新 myValue。
```

```
    mov rcx, MAX shr rcx,
    1                                                    除以2
    mov r10, qword [x]mov r11, q
    word [y]

incLoop0:     ; myValue = (myValue / x) + y


    mov rax, qword [myValue]  cqo div r10 a
    dd rax, r11 mov qword [myValue], rax lo
    op incLoop0  ret
```

第二个线程函数的代码将类似。

如果两个线程同时执行，它们都在尝试更新 ***myValue*** 变量。例如，假设线程函数 0 在核心 0 上执行，而线程函数 1 在核心 1 上执行（任意选择），可能的执行轨迹如下;

Step	Code: Core 0, Thread 0	Code: Core 1, Thread 1
1	`mov  rax, qword [myValue]`	
2	`cqo`	`mov  rax, qword [myValue]`
3	`div  qword [x]`	`cqo`
4	`add  rax, qword [y]`	`div  qword [x]`
5	`mov  qword [myValue], rax`	`add  rax, qword [y]`
6		`mov  qword [myValue], rax`

作为提醒，每个核心都有自己的寄存器集。因此，核心0的rax寄存器与核心1的rax寄存器不同。

如果变量 ***myValue*** 当前为 730，则两个线程执行应将其增加到 732。在核心 0 代码中，在第 1 步，730 被复制到核心 0 的 rax。在核心 1 中，在第 2 步，730 被复制到核心 1 的 rax。随着执行的进行，步骤 2-4 被执行，核心 0 的 rax 从 730 增加到 731。在此期间，在核心 1 中，步骤 1-3 被执行

完成，并将730递增到731。作为下一步，步骤5在核心0上执行，将731写入变量***myValue***。作为下一步，步骤6在核心1上执行，值731再次写入变量***myValue***。两个执行跟踪应该使变量递增两次。然而，由于在核心0能够写入最终值之前，值已在核心1获得，因此其中一个递增丢失或重复。这种重叠的最终结果是***myValue***的最终值将不正确。此外，由于重叠执行的数量不可预测，错误程度不易预测，并且可能在不同执行之间有所不同。

例如，如果MAX的值相当小，例如10,000，则不太可能出现重叠。每个线程都会迅速启动和完成，因此重叠执行的机会非常小。问题仍然存在，但在这种执行中可能大部分都是正确的，使得偶尔出现的异常输出容易被忽略。随着MAX的增加，重叠执行的概率增加。

此类问题调试起来可能非常具有挑战性，需要了解底层操作和技术架构。

19.3 练习

以下是本章的一些测验问题和基于此章节的建议项目。

19.3.1 测验问题

以下是本章的一些测验问题。

1) 解释并发和并行处理之间的区别。2) 列出两种常见的并行计算方法。3) 在分布式处理中，并行计算可能在哪里进行？4) 提供两个大型分布式计算项目的名称。每个项目提供一句话描述。5) 在多进程处理中，并行计算可能在哪里进行？6) 提供分布式计算方法在并行处理中的一个优点和一个缺点。7) 提供多进程方法在并行处理中的一个优点和一个缺点。8) 解释什么是 ***race condition***。

9) 当多个同时执行的线程读取一个共享变量时，会发生竞态条件吗？解释为什么会或不会发生。10) 当多个同时执行的线程（没有任何协调）写入一个共享变量时，会发生竞态条件吗？解释为什么会或不会发生。

19.3.2 建议项目

以下是本章的一些建议项目。

1) 实现概述的示例程序，并创建两个线程函数，每个线程函数计算公式 MAX/2 次。将 MAX 设置为 1,000,000,000（十亿）。

1. 初始时，将主函数结构化以调用第一个线程函数并等待其完成，然后创建第二个线程函数并等待其完成。这将强制线程顺序执行（而不是并行执行）。包括一个将整数转换为字符串并显示结果到控制台的功能。根据需要使用调试器调试程序。在工作时，不使用调试器执行程序，并验证显示的结果与MAX相同。

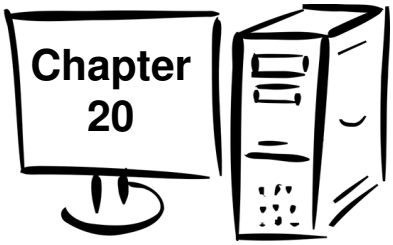
2. 使用Unix时间[63]命令建立基本执行时间。 记录总耗时时间。

3. 重新构建程序，以便同时创建并等待两个线程完成。这将允许线程并行执行。根据需要使用调试器调试程序。在工作时，不使用调试器执行程序，并注意MAX的最终值。确保充分理解为什么MAX显示的值是不正确的。此外，使用修改后的程序上的Unix时间命令来验证它使用的运行时间更少。

2) 将程序从上一个问题更新以解决竞态条件。根据需要使用调试器调试程序。实现此目的的一个非常简单的方法是使用每个线程的临时变量，并在两个线程函数完成后将它们合并，并显示最终的合并结果。在工作时，不使用调试器执行程序，并验证显示的结果对于结果是否与MAX相同。

63 For more information, refer to: http://en.wikipedia.org/wiki/Time_(Unix)

If a program is useful, it must be changed.
If a program is useless, it must be documented.



20.0 中断

在一般意义上，一个中断[64]是当前流程的暂停或保持。例如，如果你正在打电话，门铃响了，电话通话被挂起，然后去开门。销售人员离开后，电话通话继续（从上次谈话的地方开始）。

在计算机编程中，中断也是一种暂停或保持当前执行进程的行为。通常，当前进程被中断以便执行其他工作。中断通常被定义为改变处理器执行指令顺序的事件。这些事件对应于软件和/或硬件生成的信号。例如，大多数输入/输出（I/O）设备生成中断以传输或接收数据。软件程序也可以生成中断以根据需要启动I/O、请求操作系统服务或处理意外情况。

处理中断是一项敏感的任务。中断可以随时发生；内核试图尽快处理中断。此外，一个中断可以被另一个中断中断。

20.1 多用户操作系统

现代多用户操作系统（OS）通过按需共享资源，支持多个程序同时执行或看似同时执行。操作系统负责管理和共享资源。这些资源包括CPU核心、主存储器（即RAM）、辅助存储器（即磁盘或SSD）、显示屏、键盘和鼠标。例如，多个程序必须共享可用的CPU资源（核心或相应核心）。

中断机制是操作系统提供资源共享的主要手段。因此，了解操作系统如何处理中断至关重要。

64 For more information, refer to: http://en.wikipedia.org/wiki/Interrupt

计算机揭示了操作系统如何提供多处理功能。当发生中断时，当前进程被中断（即，暂停），处理中断（这取决于中断的具体原因），然后最终恢复进程。操作系统可能在恢复原始进程之前选择执行其他任务或进程。中断由一个称为中断服务例程的特殊软件例程处理（也称为中断处理程序、设备驱动程序等）。通过使用中断和快速在各个进程之间切换，操作系统能够提供所有进程同时执行的错觉。

并非所有代码都可以中断。例如，由于某些操作系统内核函数的性质，内核中存在一些区域在任何情况下都不能中断。这包括更新一些敏感的操作系统数据结构。

20.1.1 中断分类

为了更好地理解中断和中断处理，了解中断的时序和类别的一些背景信息是有用的。

20.1.2 中断时序

中断的时间可能同步或异步发生。这些术语在计算机处理中相当常见，将在以下章节中解释。

20.1.2.1 异步中断

在计算机中断的上下文中，异步发生的中断意味着中断可能在程序执行过程中任意时刻发生。相对于执行过程中的任何特定位置，异步中断是不可预测的。例如，外部硬件设备可能在不可预测的位置中断当前正在执行的过程。

20.1.2.2 同步中断

同步发生的中断通常在CPU控制下发生，是由当前正在执行的过程引起的或代表该过程。同步性质与中断发生的位置有关，而不是特定的时钟时间或CPU周期时间。同步中断通常在相同的位置重新发生（假设没有改变以解决原始原因）。

20.1.3 中断类别

中断通常被分类为硬件或软件。

20.1.3.1 硬件中断

硬件中断通常由硬件生成。硬件中断可以由以下方式引发

- 输入/输出设备（键盘、网络适配器等）
- 间隔计时器
- 其他CPU（在多处理器系统中）

硬件中断是异步发生的。硬件中断的一个例子是在键盘上按键。操作系统无法提前知道何时，甚至是否，会按下按键。为了处理这种情况，键盘支持硬件将生成一个中断。如果操作系统正在执行一个无关的程序，那么该程序在处理按键时将暂时中断。在这个例子中，具体的处理包括将按键存储在缓冲区中，并返回到被中断的程序。理想情况下，这种短暂的中断将对被中断的程序影响很小。

20.1.3.1.1 异常

异常是指由当前进程引起的中断，需要内核的注意。异常是同步发生的。在这种情况下，同步意味着异常将以可预测或可重复的方式发生。

异常通常按以下类别划分:

- 故障
- 陷阱
- 中止

一个故障的例子是页面故障，它是指从磁盘存储中加载程序部分到内存的请求。中断的进程可以无间断地重新启动。

陷阱是典型的 通常用于调试。进程无重新启动 连续性丢失。

一个中止通常表明发生了严重的错误条件，必须进行处理。这包括除以零、尝试访问无效的内存地址或尝试执行无效/非法指令。非法指令可能是指仅允许由特权/授权进程执行的指令。

无效指令可能是尝试执行一个无意义的数据项（这将没有意义）。根据错误条件的严重程度，进程通常会被终止。如果进程没有被终止，可能会执行另一个例程来尝试解决问题并重新执行原始例程（但不一定是从中断的位置）。C/C++/Java try/catch 块是这种情况的一个例子。

必须指出，这些术语没有一个绝对达成共识的定义。有些文本使用略微不同的术语。

20.1.3.2 软件中断

软件中断是在CPU处理指令时产生的。这通常是由程序员明确请求的程序异常。此类中断通常是同步发生的，常用于从操作系统请求系统服务。例如，请求I/O等系统服务。

20.2 中断类型和级别

中断具有各种类型和与之相关的权限。以下各节提供了类型和权限的解释。被中断的进程可能以比中断处理代码更低的权限执行。为了使中断有效，操作系统必须安全且迅速地处理这种权限升级和降级。

20.2.1 中断类型

两种不同的中断类型或种类是:

- 可屏蔽中断
- 不可屏蔽的中断

可屏蔽中断通常由I/O设备发出。正如“可屏蔽”这个名字所暗示的，可屏蔽中断可以在短时间内被忽略或屏蔽。这允许相关的中断处理被延迟。

不可屏蔽中断（NMI）必须立即处理。这包括一些操作系统函数和关键故障，如硬件故障。不可屏蔽中断始终由CPU处理。

20.2.2 权限级别

权限级别指的是中断代码执行的权限级别。这可能比被中断的代码执行的权限级别更高。处理器在以下四个权限级别之一中执行代码:

Level	Description
Level 0	Full access to all hardware resources (no restrictions). Used by only the lowest level OS functions.
Level 1	Somewhat restricted access to hardware resources. Used by library routines and software that interacts with hardware.
Level 2	More restricted access to hardware resources. Used by library routines and software that has limited access to some hardware.
Level 3	No direct access to hardware resources. Application programs run at this level.

如果第3级执行的应用程序程序因键盘硬件中断而被中断，则键盘中断处理程序必须在第0级执行。

以下图表显示了层级之间的关系。

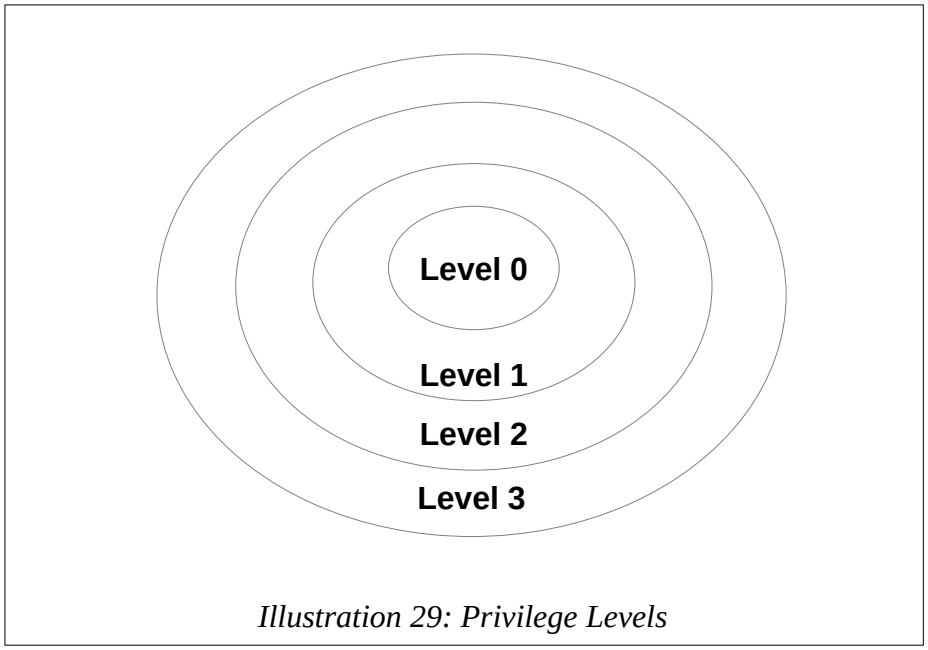


Illustration 29: Privilege Levels

操作系统中断处理机制将以安全的方式处理此权限提升和恢复。这要求将中断源和权限作为中断处理机制的一部分进行验证。

20.3 中断处理

当中断发生时，它必须被安全、快速和正确地处理或处理。一般思路是，当当前正在执行的过程被中断时，它必须被暂停，并找到适当的中断处理代码并执行。具体的中断处理需求取决于中断的原因或目的。一旦中断得到处理，原始过程执行最终将恢复。

20.3.1 中断服务例程（ISR）

响应中断而执行的代码通常被称为中断服务例程或ISR。该代码有时被称为中断处理程序、处理程序、服务例程或ISR代码。为了保持一致性，本文档将使用术语ISR。

ISR代码的开发具有挑战性，因为与并发和竞态条件相关的问题。此外，隔离问题和调试ISR代码也很困难。

20.3.2 处理步骤

中断处理的一般步骤在以下章节中概述。

20.3.2.1 悬挂

当前程序执行已暂停。至少需要将rip和rFlags寄存器保存到系统堆栈。其余寄存器可能（作为进一步步骤）会被保留，具体取决于中断类型。rFlags标志寄存器必须立即保留，因为中断可能已被异步生成，并且这些寄存器会在后续指令执行时发生变化。此多阶段过程确保程序上下文可以完全恢复。

20.3.2.2 获取ISR地址

ISR地址存储在一个称为中断描述符表[65]（IDT）的表中。对于每个ISR，IDT包含ISR地址和一些附加信息，包括ISR的任务门（优先级和权限信息）。IDT中的每个条目总共占用8个字节，每个IDT条目总共占用16个字节。IDT中最多可以有256（0-255）个可能的条目。

要获取ISR的起始地址，将中断号乘以16（因为每个条目是16字节），这用作IDT中的偏移量，从中获取ISR地址（对于该中断）。

IDT的开始由专用寄存器IDTR指向，该寄存器只能由操作系统访问，并且需要0级权限才能访问。

ISR例程的地址针对特定操作系统版本和系统的具体硬件配置。当系统首次启动时创建IDT，并反映特定的系统配置。这对于操作系统在不同系统硬件配置上正确且一致地工作至关重要。

20.3.2.3 跳转到ISR

一旦从中断描述符表（IDT）中获取到ISR地址，就会进行一些验证。这包括确保中断来自合法/有效的源，并且如果需要且允许的话，还要改变特权等级。一旦完成验证

65 *Note,* for Windows this data structure is referred to as the Interrupt Vector Table (IVT).

成功地将来自IDT的中断服务例程（ISR）地址放置在rip寄存器中，从而实现跳转到ISR例程。

20.3.2.4 悬挂执行ISR

在此阶段，根据具体的ISR，可能执行完整的进程上下文切换。进程上下文切换涉及保存被中断进程的整个CPU寄存器集。

在基于Linux的操作系统（OS）中，中断服务例程（ISR）通常分为两部分，分别称为上半部和下半部。其他操作系统将这些称为第一级中断处理程序（FLIH）和第二级中断处理程序（SLIH）。

上半部分或FLIH立即执行，此处执行任何关键活动。这些活动特定于ISR，可能包括确认中断、重置硬件（如有必要）以及记录仅在中断时才可用的任何信息。上半部分可能执行对其他中断的某些阻塞（需要最小化）。

下半部分是执行任何处理活动（如果有）的地方。这有助于确保上半部分快速完成，并将任何非关键处理推迟到更方便的时间。如果存在下半部分，上半部分将创建并安排下半部分的执行。

一旦上半部分完成，操作系统调度器将选择一个新的进程。

20.3.2.5 恢复

当操作系统准备好恢复中断进程时，程序上下文被恢复并执行iret指令（以弹出rFlags和rip寄存器，从而完成恢复）。

20.4 悬挂中断处理摘要

以下图表展示了系统处理中断所使用的通用流程概述。

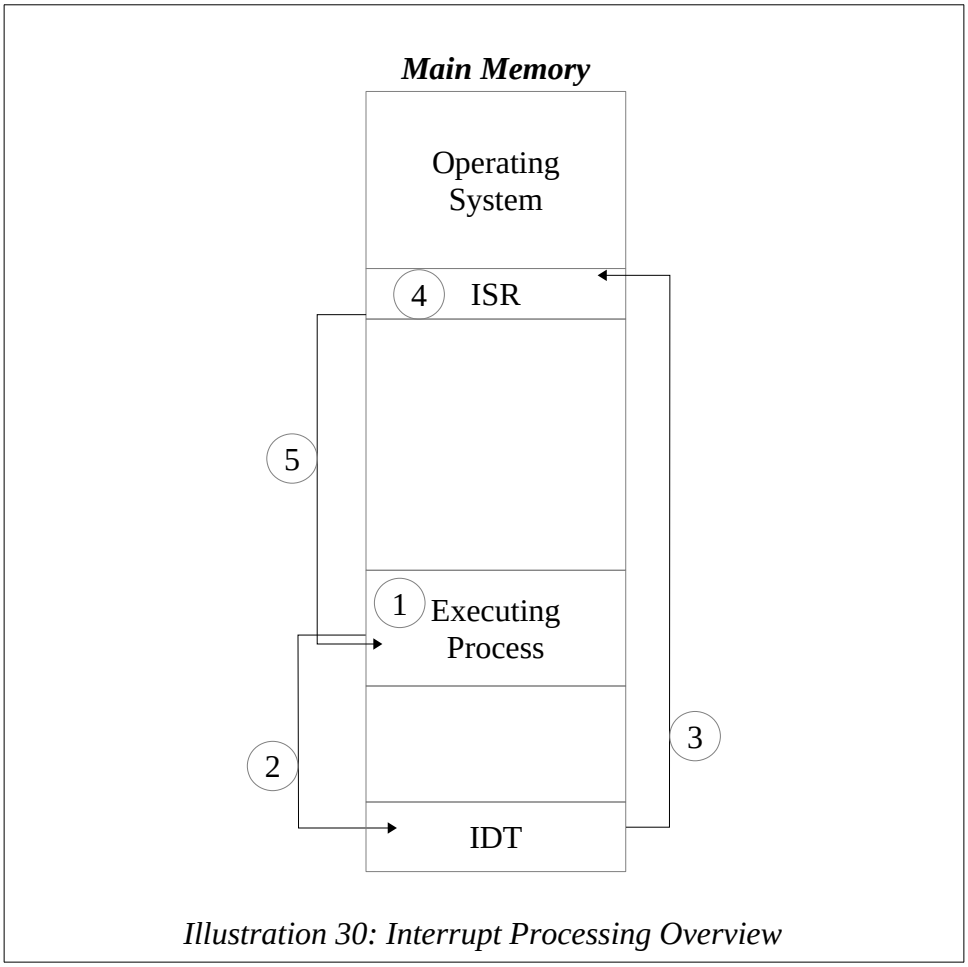


Illustration 30: Interrupt Processing Overview

步骤如下详细说明:

1. 当前程序执行被暂停 - 将rip和rFlags寄存器保存到堆栈
2. 获取中断服务例程（ISR）的起始地址 - 中断号乘以16 - 作为中断描述符表（IDT）的偏移量 - 从IDT中获取（该中断的）ISR地址

3. 跳转到中断服务例程 - 将rip设置为IDT中的地址 4. 中断服务例程执行 - 保存上下文（即任何被更改的额外寄存器） - 处理中断（特定于生成的中断） - 安排任何后续数据处理活动 5. 中断进程恢复 - 根据操作系统调度器恢复调度 - 恢复上下文 - 执行iret（弹出rFlags和rip寄存器）

此中断处理机制允许对ISR地址进行动态、运行时查找。

20.5 练习

以下是一些　e 测验问题和基于 th 的建议项目　　这是第{v *}章。

20.5.1 测验问题

以下是本章的一些测验问题。

1) 操作系统负责什么？列出一些资源。2) 什么是中断？3) 什么是异常？4) 什么是ISR以及它的作用是什么？5）当发生中断时，操作系统从哪里（数据结构名称）获取地址，它包含什么？6）当发生中断时，如何计算IDT中的适当偏移量？7）iret和ret指令之间的区别是什么？8）为什么操作系统使用中断机制而不是仅仅执行标准调用？9）异步发生的中断是什么意思?

10) 同步发生的中断是什么意思?

当发生中断时，rip和rFlags寄存器被压入堆栈。与call语句类似，rip寄存器被压入以保存返回地址。解释为什么rFlag寄存器被压入堆栈。

12) 列出两种硬件中断。

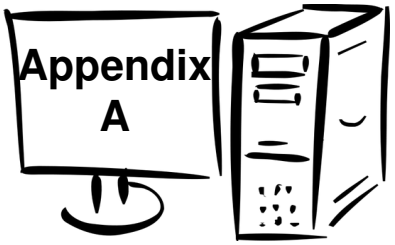
13) 列出一种程序生成异常的方法。

14) 可屏蔽中断与非屏蔽中断之间的区别是什么?

20.5.2 建议项目

以下是本章的一些建议项目。

1) 编写一个程序以获取和列出IDT的内容。这需要一个整数到ASCII/十六进制程序，以便以十六进制显示适用的地址。根据需要使用调试器来调试程序。在工作时，不使用调试器执行程序以显示结果。



21.0 附录A – ASCII表

此附录提供了ASCII表的副本以供参考。

Char	Dec	Hex
NUL	0	0x00
SOH	1	0x01
STX	2	0x02
ETX	3	0x03
EOT	4	0x04
ENQ	5	0x05
ACK	6	0x06
BEL	7	0x07
BS	8	0x08
TAB	9	0x09
LF	10	0x0A
VT	11	0x0B
FF	12	0x0C
CR	13	0x0D
SO	14	0x0E
SI	15	0x0F
DLE	16	0x10
DC1	17	0x11
DC2	18	0x12
DC3	19	0x13

Char	Dec	Hex
spc	32	0x20
!	33	0x21
"	34	0x22
#	35	0x23
$	36	0x24
%	37	0x25
&	38	0x26
'	39	0x27
(	40	0x28
)	41	0x29
*	42	0x2A
+	43	0x2B
,	44	0x2C
-	45	0x2D
.	46	0x2E
/	47	0x2F
0	48	0x30
1	49	0x31
2	50	0x32
3	51	0x33

Char	Dec	Hex
@	64	0x40
A	65	0x41
B	66	0x42
C	67	0x43
D	68	0x44
E	69	0x45
F	70	0x46
G	71	0x47
H	72	0x48
I	73	0x49
J	74	0x4A
K	75	0x4B
L	76	0x4C
M	77	0x4D
N	78	0x4E
O	79	0x4F
P	80	0x50
Q	81	0x51
R	82	0x52
S	83	0x53

Char	Dec	Hex
`	96	0x60
a	97	0x61
b	98	0x62
c	99	0x63
d	100	0x64
e	101	0x65
f	102	0x66
g	103	0x67
h	104	0x68
i	105	0x69
j	106	0x6A
k	107	0x6B
l	108	0x6C
m	109	0x6D
n	110	0x6E
o	111	0x6F
p	112	0x70
q	113	0x71
r	114	0x72
s	115	0x73

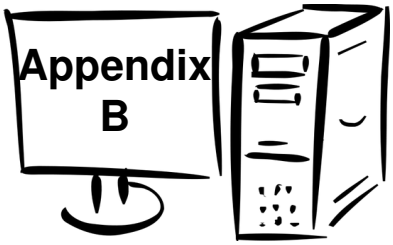
DC4	**20**	**0x14**
NAK	**21**	**0x15**
SYN	**22**	**0x16**
ETB	**23**	**0x17**
CAN	**24**	**0x18**
EM	**25**	**0x19**
SUB	**26**	**0x1A**
ESC	**27**	**0x1B**
FS	**28**	**0x1C**
GS	**29**	**0x1D**
RS	**30**	**0x1E**
US	**31**	**0x1F**

4	**52**	**0x34**
5	**53**	**0x35**
6	**54**	**0x36**
7	**55**	**0x37**
8	**56**	**0x38**
9	**57**	**0x39**
:	**58**	**0x3A**
;	**59**	**0x3B**
<	**60**	**0x3C**
=	**61**	**0x3D**
>	**62**	**0x3E**
?	**63**	**0x3F**

T	**84**	**0x54**
U	**85**	**0x55**
V	**86**	**0x56**
W	**87**	**0x57**
X	**88**	**0x58**
Y	**89**	**0x59**
Z	**90**	**0x5A**
[	**91**	**0x5B**
****	**92**	**0x5C**
]	**93**	**0x5D**
^	**94**	**0x5E**
_	**95**	**0x5F**

t	**116**	**0x74**
u	**117**	**0x75**
v	**118**	**0x76**
w	**119**	**0x77**
x	**120**	**0x78**
y	**121**	**0x79**
z	**122**	**0x7A**
{	**123**	**0x7B**
|	**124**	**0x7C**
}	**125**	**0x7D**
~	**126**	**0x7E**
DEL	**127**	**0x7F**

For additional information and a more complete listing of the ASCII codes (including the extended ASCII characters), refer to http://www.asciitable.com/



22.0 附录B – 指令集摘要

此附录提供了本文本中涵盖的指令列表和简要描述。这组指令是完整指令集的一个子集。要获取指令的完整列表，请参阅第1章中注明的参考文献。

22.1 符号说明

以下表格总结了所使用的符号约定。

Operand Notation	Description
<reg>	Register operand. The operand must be a register.
<reg8>, <reg16>, <reg32>, <reg64>	Register operand with specific size requirement. For example, **reg8** means a byte sized register (e.g., **al**, **bl**, etc.) only and **reg32** means a double-word sized register (e.g., **eax**, **ebx**, etc.) only.
<dest>	Destination operand. The operand may be a register or memory. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
<RXdest>	Floating-point destination register operand. The operand must be a floating-point register. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
<src>	Source operand. Operand value is unchanged after the instruction.
<imm>	Immediate value. May be specified in decimal, hex, octal, or binary.

Operand Notation	Description
<mem>	Memory location. May be a variable name or an indirect reference (i.e., a memory address).
<op> or <operand>	Operand, register or memory.
<op8>, <op16>, <op32>, <op64>	Operand, register or memory, with specific size requirement. For example, **op8** means a byte sized operand only and **reg32** means a double-word sized operand only.
<label>	Program label.

22.2 数据移动指令

以下是摘要　　基本数据移动和寻址指令的ry　　动作。

Instruction	Explanation
mov <dest>, <src>	Copy source operand to the destination operand. *Note 1*, both operands cannot be memory. *Note 2*, destination operands cannot be an immediate.
lea <reg64>, <mem>	Place address of <**mem**> into **reg64**.

22.3 数据转换说明

以下是基本数据转换说明的摘要。

Instruction	Explanation
movzx <dest>, <src> **movzx <reg16>, <op8>** **movzx <reg32>, <op8>** **movzx <reg32>, <op16>** **movzx <reg64>, <op8>** **movzx <reg64>, <op16>**	Unsigned widening conversion. *Note 1*, both operands cannot be memory. *Note 2*, destination operands cannot be an immediate. *Note 3*, immediate values not allowed.
cbw	Convert byte in **al** into word in **ax**. *Note*, only works for **al** to **ax** register.

Instruction	Explanation
`cwd`	Convert word in **ax** into double-word in **dx:ax**. *Note*, only works for **ax** to **dx:ax** registers.
`cwde`	Convert word in **ax** into double-word in **eax**. *Note*, only works for **ax** to **eax** register.
`cdq`	Convert double-word in **eax** into quadword in **edx:eax**. *Note*, only works for **eax** to **edx:eax** registers.
`cdqe`	Convert double-word in **eax** into quadword in **rax**. *Note*, only works for **rax** register.
`cqo`	Convert quadword in **rax** into double-quadword in **rdx:rax**. *Note*, only works for **rax** to **rdx:rax** registers.

`movsx  <dest>, <src>`
 	
`movsx  <reg16>, <op8>`
`movsx  <reg32>, <op8>`
`movsx  <reg32>, <op16>`
`movsx  <reg64>, <op8>`
`movsx  <reg64>, <op16>`
`movsxd  <reg64>, <op32>`
	Signed widening conversion (via sign extension).
	Note 1, both operands cannot be memory.
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.

22.4 整数算术指令

以下是基本整数算术指令的摘要。

Instruction	Explanation

`add  <dest>, <src>`
 	
`add  <reg>, <reg>`
`add  <reg>, <imm8/16/32>`
`add  <reg>, <mem>`
`add  <mem>, <reg>`
`add  <mem>, <imm8/16/32>`
	Add two operands, (<**dest**> + <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

`inc  <operand>`	Increment <**operand**> by 1.

Instruction	Explanation
	Note, <**operand**> cannot be an immediate.
`adc <dest>, <src>`	Add two operands, (<**dest**> + <**src**>) and any previous carry (stored in the carry bit in the **rFlag** register) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate.
Examples:	`adc rcx, qword [dVvar1]` `adc rax, 42`

`sub <dest>, <src>`
 	
`sub <reg>, <reg>`
`sub <reg>, <imm8/16/32>`
`sub <reg>, <mem>`
`sub <mem>, <reg>`
`sub <mem>, <imm8/16/32>`
	Subtract two operands, (<**dest**> - <**src**>) and place the result in <**dest**> (over-writing previous value).
	Note 1, both operands cannot be memory.
	Note 2, destination operand cannot be an immediate.
	Note 3, 64-bit immediate values are not allowed.

`dec <operand>`	Decrement <**operand**> by 1. *Note*, <**operand**> cannot be an immediate.

`mul <src>`
 	
`mul <op8>`
`mul <op16>`
`mul <op32>`
`mul <op64>`
	Multiply **A** register (**al**, **ax**, **eax**, or **rax**) times the <**src**> operand.
	Byte: **ax** = **al** * <**src**>
	Word: **dx:ax** = **ax** * <**src**>
	Double: **edx:eax** = **eax** * <**src**>
	Quad: **rdx:rax** = **rax** * <**src**>
	Note, <**src**> operand cannot be an immediate.

`imul <src>`
`imul <dest>, <src/imm32>`
`imul <dest>, <src>, <imm32>`
 	
`imul <op8>`
`imul <op16>`
`imul <op32>`
`imul <op64>`
`imul <reg16>,`
	Signed multiply instruction.
	
	For single operand:
	Byte: **ax** = **al** * <**src**>
	Word: **dx:ax** = **ax** * <**src**>
	Double: **edx:eax** = **eax** * <**src**>
	Quad: **rdx:rax** = **rax** * <**src**>
	Note, <**src**> operand cannot be an immediate.

Instruction	Explanation
<op16/imm8/16/32> imul <reg32>, <op32/imm8/16/32> imul <reg64>, <op64/imm> imul <reg16>, <op16>, <imm8/16/32> imul <reg32>, <op32>, <imm8/16/32> imul <reg64>, <op64>, <imm8/16/32>	For two operands: **<reg16> = <reg16> * <op16/imm>** **<reg32> = <reg32> * <op32/imm>** **<reg64> = <reg64> * <op64/imm>** For three operands: **<reg16> = <op16> * <imm>** **<reg32> = <op32> * <imm>** **<reg64> = <op64> * <imm>**
div <src> div <op8> div <op16> div <op32> div <op64>	Unsigned divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand. Byte: **al** = **ax** / <**src**>, rem in **ah** Word: **ax** = **dx:ax** / <**src**>, rem in **dx** Double: **eax** = **edx:eax** / <**src**>, rem in **edx** Quad: **rax** = **rdx:rax** / <**src**>, rem in **rdx** *Note*, <**src**> operand cannot be an immediate.
idiv <src> idiv <op8> idiv <op16> idiv <op32> idiv <op64>	Signed divide **A/D** register (**ax**, **dx:ax**, **edx:eax**, or **rdx:rax**) by the <**src**> operand. Byte: **al** = **ax** / <**src**>, rem in **ah** Word: **ax** = **dx:ax** / <**src**>, rem in **dx** Double: **eax** = **edx:eax** / <**src**>, rem in **edx** Quad: **rax** = **rdx:rax** / <**src**>, rem in **rdx** *Note*, <**src**> operand cannot be an immediate.

22.5 逻辑、移位和旋转指令

以下是基本逻辑、移位、算术移位和旋转指令的摘要。

Instruction	Explanation
and <dest>, <src> and <reg>, <reg> and <reg>, <imm8/16/32>	Perform logical AND operation on two operands, (<**dest**> and <**src**>) and place the result in <**dest**> (over-writing previous value).

Instruction	Explanation
and <reg>, <mem> **and <mem>, <reg>** **and <mem>, <imm8/16/32>**	*Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate. *Note 3*, 64-bit immediate values are not allowed.
or <dest>, <src> **or <reg>, <reg>** **or <reg>, <imm8/16/32>** **or <reg>, <mem>** **or <mem>, <reg>** **or <mem>, <imm8/16/32>**	Perform logical OR operation on two operands, (<**dest**> || <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand cannot be an immediate. *Note 3*, 64-bit immediate values are not allowed.
xor <dest>, <src>	Perform logical XOR operation on two operands, (<**dest**> ^ <**src**>) and place the result in <**dest**> (over-writing previous value). *Note 1*, both operands cannot be memory. *Note 2*, destination operand ca not be an immediate.
not <op>	Perform a logical not operation (one's complement on the operand 1's→0's and 0's→1's). *Note*, operand cannot be an immediate.
shl <dest>, <imm8> **shl <dest>, cl**	Perform logical shift left operation on destination operand. Zero fills from right (as needed). The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8 bit immediate values are allowed.
shr <dest>, <imm8> **shr <dest>, cl**	Perform logical shift right operation on destination operand. Zero fills from left (as needed). The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an

Instruction	Explanation
	immediate. *Note 2*, only 8-bit immediate values are allowed.
sal <dest>, <imm8> **sal <dest>, cl**	Perform arithmetic shift left operation on destination operand. Zero fills from right (as needed). The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.
sar <dest>, <imm8> **sar <dest>, cl**	Perform arithmetic shift right operation on destination operand. Sign fills from left (as needed). The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.
rol <dest>, <imm8> **rol <dest>, cl**	Perform rotate left operation on destination operand. The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.
ror <dest>, <imm8> **ror <dest>, cl**	Perform rotate right operation on destination operand. The <**imm**> or the value in **cl** register must be between 1 and 64. *Note 1*, destination operand cannot be an immediate. *Note 2*, only 8-bit immediate values are allowed.

22.6 控制指令

以下是基本控制指令的摘要。

Instruction	Explanation
jmp <label>	Jump to specified label. *Note*, label must be defined exactly once.
Examples:	**jmp startLoop** **jmp ifDone** **jmp last**
cmp <op1>, <op2>	Compare <**op1**> with <**op2**>. Results are stored in the **rFlag** register. *Note 1*, operands are not changed. *Note 2*, both operands cannot be memory. *Note 3*, <**op1**> operand cannot be an immediate. *Note 4*, <**op2**> can not be a 64-bit immediate value.
je <label>	Based on preceding comparison instruction, jump to <**label**> if <**op1**> == <**op2**>. Label must be defined exactly once.
jne <label>	Based on preceding comparison instruction, jump to <**label**> if <**op1**> **!=** <**op2**>. Label must be defined exactly once.
jl <label>	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> < <**op2**>. Label must be defined exactly once.
jle <label>	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> ≤ <**op2**>. Label must be defined exactly once.
jg <label>	For signed data, based on preceding comparison instruction, jump to <**label**> if <**op1**> > <**op2**>. Label must be defined exactly once.

Instruction	Explanation
jge <label>	For signed data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≥ **<op2>**. Label must be defined exactly once.
jb <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** < **<op2>**. Label must be defined exactly once.
jbe <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≤ **<op2>**. Label must be defined exactly once.
ja <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** > **<op2>**. Label must be defined exactly once.
jae <label>	For unsigned data, based on preceding comparison instruction, jump to **<label>** if **<op1>** ≥ **<op2>**. Label must be defined exactly once.
loop <label>	Decrement rcx register and jump to **<label>** if **rcx** is ≠ 0. *Note*, label must be defined exactly once.

22.7 栈指令

以下是基本堆栈指令的摘要 .

Instruction	Explanation
push <op64>	Push the 64-bit operand on the stack. Adjusts **rsp** accordingly. Operand is unaltered.
pop <op64>	Pop the 64-bit operand from the stack. Adjusts **rsp** accordingly. The operand may not be an immediate value. Operand is overwritten.

22.8 函数指令

以下是实现函数调用的基本指令摘要。

Instruction	Explanation
`call <funcName>`	Calls a function. Push the 64-bit **rip** register and jump to the <funcName>.
`ret`	Return from a function. Pop the stack into the **rip** register, effecting a jump to the line after the call.

22.9 浮点数据移动指令

以下是浮点数据移动指令的基本指令摘要。

Instruction	Explanation
`movss <dest>, <src>`	Move scaler single precision floating point value. Copy 32-bit source operand to the 32-bit destination operand. *Note 1*, both operands cannot be memory. *Note 2*, operands cannot be an immediate.

Examples:	`movss xmm0, dword [x]`
	`movss dword [fltVar], xmm1`
	`movss xmm3, xmm2`

`movsd <dest>, <src>`	Move scaler double precision floating point value. Copy 64-bit source operand to the 64-bit destination operand. *Note 1*, both operands cannot be memory. *Note 2*, operands cannot be an immediate.

Examples:	`movsd xmm0, qword [y]`
	`movsd qword [fltdVar], xmm1`
	`movsd xmm3, xmm2`

22.10 浮点数数据转换指令

以下是浮点数据转换指令的基本指令摘要。

Instruction	Explanation
cvtss2sd <RXdest>, <Rxsrc>	Convert scaler single precision to double precision floating point value. Convert 32-bit floating-point source operand to the 64-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtss2sd xmm0, dword [fltSVar]** **cvtss2sd xmm3, xmm2**
cvtsd2ss <RXdest>, <RXsrc> **cvttsd2ss <RXdest>, <RXsrc>**	Convert scaler double precision to single precision floating point value with loss of precision. Convert 64-bit floating-point source operand to the 32-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtsd2ss xmm0, qword [fltDVar]** **cvtsd2ss xmm1, xmm5**
cvtss2si <reg>, <Rxsrc> **cvttss2si <reg>, <RXsrc>**	Convert scaler single precision to integer value. Convert 32-bit floating-point source operand to the 32-bit or 64-bit integer destination operand. The **cvtss2si** performs rounding and the **cvtss2si** performs truncation. *Note 1*, destination operand must be register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.

Instruction	Explanation
Examples:	**cvtss2si eax, xmm0** **cvtss2si rbx, dword [fltSVar]** **cvttss2si eax, xmm0** **cvttss2si rcx, dword [fltSVar]**
cvtsd2si <reg>, <Rxsrc> **cvttsd2si <reg>, <RXsrc>**	Convert scaler double precision to integer value. Convert 64-bit floating-point source operand to the 32-bit or 64-bit integer destination operand. The **cvtsd2si** performs rounding and the **cvtsd2si** performs truncation. *Note 1*, destination operand must be register. *Note 2*, source operand must be a floating-point register and cannot be an immediate.
Examples:	**cvtsd2si xmm1, xmm0** **cvtsd2si eax, xmm0** **cvttsd2si rbx, xmm0** **cvttsd2si eax, qword [fltDVar]**
cvtsi2ss <RXdest>, <src>	Convert integer value to single precision floating point value. Convert 32-bit or 64-bit integer source operand to the 32-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**cvtsi2ss xmm0, eax** **cvtsi2ss xmm0, ebx** **cvtsi2ss xmm0, dword [fltDVar]**
cvtsi2sd <RXdest>, <src>	Convert integer value to double precision floating point value. Convert 32-bit integer source operand to the 64-bit floating-point destination operand. *Note 1*, destination operand must be floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**cvtsi2sd xmm0, eax** **cvtsi2sd xmm0, dword [fltDVar]**

2 2.11 浮点算术指令 ns

以下是浮点算术指令的基本指令摘要。

Instruction	Explanation
addss <RXdest>, <src>	Add single precision floating point values. Add two 32-bit floating-point operands, (**<RXdest>** + **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**addss xmm0, xmm3** **addss xmm5, dword [fSVar]**
addsd <RXdest>, <src>	Add double precision floating point values. Add two 64-bit floating-point operands, (**<RXdest>** + **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**addsd xmm0, xmm3** **addsd xmm5, qword [fDVar]**
subss <RXdest>, <src>	Subtract single precision floating point values. Subtract two 32-bit floating-point operands, (**<RXdest>** - **<src>**) and place the result in **<RXdest>** (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**subss xmm0, xmm3** **subss xmm5, dword [fSVar]**

Instruction	Explanation
subsd <RXdest>, <src>	Subtract double precision floating point values. Subtract two 64-bit floating-point operands, (<**RXdest**> - <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**subsd xmm0, xmm3** **subsd xmm5, qword [fDVar]**
mulss <RXdest>, <src>	Multiply single precision floating point values. Multiply two 32-bit floating-point operands, (<**RXdest**> * <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**mulss xmm0, xmm3** **mulss xmm5, dword [fSVar]**
mulsd <RXdest>, <src>	Multiply double precision floating point values. Multiply two 64-bit floating-point operands, (<**RXdest**> * <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**mulsd xmm0, xmm3** **mulsd xmm5, qword [fDVar]**

Instruction	Explanation
divss <RXdest>, <src>	Divide single precision floating point values. Divide two 32-bit floating-point operands, (<**RXdest**> / <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**divss xmm0, xmm3** **divss xmm5, dword [fSVar]**
divsd <RXdest>, <src>	Divide double precision floating point values. Divide two 64-bit floating-point operands, (<**RXdest**> / <**src**>) and place the result in <**RXdest**> (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**divsd xmm0, xmm3** **divsd xmm5, qword [fDVar]**
sqrtss <RXdest>, <src>	Take the square root of the 32-bit floating-point source operand and place the result in destination operand (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	**sqrtss xmm0, xmm3** **sqrtss xmm7, dword [fSVar]**

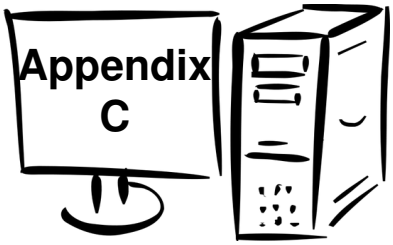
Instruction	Explanation
`sqrtsd <RXdest>, <src>`	Take the square root of the 64-bit floating-point source operand and place the result in destination operand (over-writing previous value). *Note 1*, destination operands must be a floating-point register. *Note 2*, source operand cannot be an immediate.
Examples:	`sqrtsd xmm0, xmm3` `sqrtsd xmm7, qword [fDVar]`

22.12 浮点控制指令

以下是浮点控制指令的基本指令摘要。

Instruction	Explanation
`ucomiss <RXsrc>, <src>`	Compare two 32-bit floating-point operands, (**<RXsrc>** + **<src>**). Results are placed in the **rFlag** register. Neither operand is changed. *Note 1*, <RXsrc> operands must be a floating-point register. *Note 2*, source operand may be a floating-point register or memory, but not be an immediate.
Examples:	`ucomiss xmm0, xmm3` `ucomiss xmm5, dword [fSVar]`
`ucomisd <RXsrc>, <src>`	Compare two 64-bit floating-point operands, (**<RXsrc>** + **<src>**). Results are placed in the **rFlag** register. Neither operand is changed. *Note 1*, <RXsrc> operands must be a floating-point register. *Note 2*, source operand may be a floating-point register or memory, but not be an immediate.

Instruction	Explanation
Examples:	`ucomisd xmm0, xmm3` `ucomisd xmm5, dword [fSVar]`



23.0 附录C – 系统服务

此附录提供了系统服务调用的子集列表及其简要描述。此列表适用于64位Ubuntu系统。更完整的列表可以从多个网络来源获得。

23.1 返回代码

系统调用将在rax寄存器中返回一个代码。如果返回的值小于0，则表示发生了错误。如果操作成功，返回的值将取决于特定的系统服务。有关错误代码值的更多信息，请参阅“错误代码”部分。

23.2 基本系统服务

以下表格总结了更常见的系统服务。

Call Code (rax)	System Service	Description
0	**SYS_read**	**Read characters**
		rdi = file descriptor (of where to read from)
		rsi = address of where to store characters
		rdx = count of characters to read
	If unsuccessful, returns negative value. If successful, returns count of characters actually read.	
1	**SYS_write**	**Write characters**
		rdi = file descriptor (of where to write to)

Call Code (rax)	System Service	Description
		rsi = address of characters to write
		rdx = count of characters to write
	If unsuccessful, returns negative value. If successful, returns count of characters actually written.	
2	**SYS_open**	**Open a file**
		rdi = address of NULL terminated file name
		rsi = file status flags (typically O_RDONLY)
	If unsuccessful, returns negative value. If successful, returns file descriptor.	
3	**SYS_close**	**Close an open file**
		rdi = file descriptor of open file to close
	If unsuccessful, returns negative value.	
8	**SYS_lseek**	**Reposition the file read/write file offset.**
		rdi = file descriptor (of where to write to)
		rsi = offset
		rdx = origin
	If unsuccessful, returns negative value.	
57	**SYS_fork**	**Fork current process.**
59	**SYS_execve**	**Execute a program**
		rdi = Address of NULL terminated string for name of program to execute.
60	**SYS_exit**	**Terminate executing process.**
		rdi = exit status (typically 0)

Call Code (rax)	System Service	Description
85	**SYS_creat**	**Open/Create a file.**
		rdi = address of NULL terminated file name
		rsi = file mode flags
	If unsuccessful, returns negative value. If successful, returns file descriptor.	
96	**SYS_gettimeofday**	**Get date and time of day**
		rdi = address of time value structure
		rsi = address of time zone structure
	If unsuccessful, returns negative value. If successful, returns information in the passed structures.	

23.3 文件模式

当执行文件操作时，文件模式向操作系统提供有关将被允许的文件访问权限的信息。

当打开一个　　在现有文件中，必须使用以下文件模式之一　　未指定。

Mode	Value	Description
O_RDONLY	0	Read only. Allow reading from the file, but to not allow writing to the file. Most common operation.
O_WRONLY	1	Write only. Typically used if information is to be appended to a file.
O_RDWR	2	Allow simultaneous reading and writing.

创建新文件时，必须指定文件权限。以下是完整的文件权限集合。与Linux文件系统标准一致，权限值以八进制指定。

Mode	Value	Description
S_IRWXU	00700q	User (file owner) has read, write, and execute permission.
S_IRUSR	00400q	User (file owner) has read permission.
S_IWUSR	00200q	User (file owner) has write permission.
S_IXUSR	00100q	User (file owner) has execute permission.
S_IRWXG	00070q	Group has read, write, and execute permission.
S_IRGRP	00040q	Group has read permission.
S_IWGRP	00020q	Group has write permission.
S_IXGRP	00010q	Group has execute permission.
S_IRWXO	00007q	Others have read, write, and execute permission.
S_IROTH	00004q	Others have read permission.
S_IWOTH	00002q	Others have write permission.
S_IXOTH	00001q	Others have execute permission.

文本示例仅涉及文件的用户或所有者的权限。

23.4 错误代码

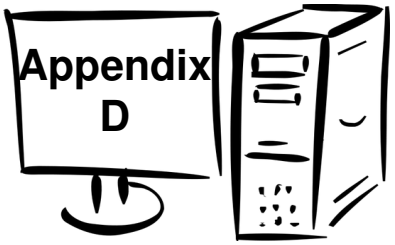
如果系统服务返回错误，返回码的值将是负数。以下是一个错误码列表。提供了代码值以及Linux的符号名称。高级语言通常使用在汇编级别不使用的名称，仅提供参考。

Error Code	Symbolic Name	Description
-1	EPERM	Operation not permitted.
-2	ENOENT	No such file or directory.
-3	ESRCH	No such process.
-4	EINTR	Interrupted system call.

-5	EIO	I/O Error.
-6	ENXIO	No such device or address.
-7	E2BIG	Argument list too long.
-8	ENOEXEC	Exec format error.
-9	EBADF	Bad file number.
-10	ECHILD	No child process.
-11	EAGAIN	Try again.
-12	ENOMEM	Out of memory.
-13	EACCES	Permission denied.
-14	EFAULT	Bad address.
-15	ENOTBLK	Block device required.
-16	EBUSY	Device or resource busy.
-17	EEXIST	File exists.
-18	EXDEV	Cross-device link.
-19	ENODEV	No such device.
-20	ENOTDIR	Not a directory.
-21	EISDIR	Is a directory.
-22	EINVAL	Invalid argument.
-23	ENFILE	File table overflow.
-24	EMFILE	Too many open files.
-25	ENOTTY	Not a typewriter.
-26	ETXTBSY	Text file busy.
-27	EFBIG	File too large.
-28	ENOSPC	No space left on device.
-29	ESPIPE	Illegal seek.
-30	EROFS	Read-only file system.
-31	EMLINK	Too many links.
-32	EPIPE	Broken pipe.

-33	EDOM	Math argument out of domain of function.
-34	ERANGE	Math result not representable.

仅显示最常见的错误代码。 完整列表可通过互联网或查看当前系统包含文件获得。 对于Ubuntu，这通常位于 /usr/include/asm-generic/errno-base.h。



24.0 附录D – 试题答案

本附录提供了每章测验问题的答案。

24.1 测验问题答案，第1章

第一章没有测验问题。

24.2 测验问题答案，第2章

1) 参见第2.1节，插图1，计算机体系结构。2) 总线或互连。

3) 二级存储。

4) 主存储器或主内存（RAM）。

5) 在CPU附近保留数据的副本，消除了通过总线访问RAM所需的额外时间。

6) 4 字节。7) 1 字节。

8) LSB是40_{16}，MSB是00_{16}。

9) 答案如下：

High memory	00_{16}
	$4C_{16}$
	$4B_{16}$
Low memory	40_{16}

10) 布局如下:

11) 答案如下:

1. 82.
643. 1
64. 32
5. 646
. 87. 8
8. 16

12) 撤销寄存器。13) RSP寄存器。14) RAX寄存器是：0000000000000000_{16}。15) 答案如下：1. EF_{16} 2. $CDEF_{16}$ 3. $89ABCDEF_{16}$ 4. $0123456789ABCDEF_{16}$

24.3 测验问题答案，第3章

1) 答案如下:

1. -128 至 +127
2. 0 至 255

3. −32,768 至 +32,767 4. 0 至 65,535
5. −2,147,483,648 至 +2,147,483,647
6. 0 至 4,294,967,295

2) 答案如下：

1. 5 2.
9 3. 1
3 4. 2
1

3) 答案如下：

1. 0xFD2
. 0x0B3.
0xF74. 0
xEB

4) 答案如下：

1. 0xFFEF2
. 0x00113.
0xFFE14. 0
xFF76

5) 答案如下：

1. 0xFFFFFFF52.
0xFFFFFFE53. 0
x000000074. 0xF
FFFFEFB

6) 答案如下：

1. -52.
-22

3. -13
4. -8

7) 0.5_{10}表示为0.1_2 8) 答案如下：

1. -12.25 2. +12.25 3. -6.5 4. -7. 5

9) 答案如下：

1. 0x413400002. 0xC18900003. 0x41AF00004. 0xBF400000

10) 答案如下：

1. 0x41
2. 0x61
3. 0x30
4. 0x38
5. 0x09

11) 答案如下：1. “世界” = 0x57 0x6F 0x72 0x6C 0x64 2. “123” = 0x31 0x32 0x33 3. “Yes!?” = 0x59 0x65 0x73 0x21 0x 3F

24.4 测验问题答案，第4章

1) yasm 2) 使用分号（;）。

3) 段数据。4) 段bss。5) 段text。6) 答案如下：1. bNum db 10 2. wNum dw 10291 3. dwNum dd 2126010 4. qwNum dq 10000000000

7) 答案如下：1. bArr resb 100 2. wArr resw 3000 3. dwArr resd 200 4. qArr resq 5000

8) 声明如下： global _start _start:

24.5 测验问题答案，第五章

1) 关系为1:1（一对一）。2) 创建符号表、宏展开和常量表达式的评估。

3) 代码最终生成，如需则创建列表文件，创建对象文件。

4) 将一个或多个目标文件合并成一个可执行文件，更新所有可重定位地址，搜索用户和系统库，如果需要则创建交叉引用文件，并创建最终的执行文件。

5) 尝试打开可执行文件（验证存在和权限），读取头部信息，请求操作系统创建新进程，如果成功，读取可执行文件的其余部分并将其加载到内存中（由操作系统指定），并在加载完成后通知操作系统。*Note*，加载器不运行进程。

6) 示例可能包括：1. BUFFSIZE + 1 2. MAX + OFFSET *Note, a*ssumes that upper case implies defined constant. Many examples possible. 7) 见第5.1节，图4，概述：组装、链接、加载。8) 在运行时。

9) 符号名称和符号地址。

24.6 测验问题答案，第6章

1) 通过输入： ddd <progName>

2) “-g”选项。

3) 执行程序，始终从开始处开始。4) continue 命令继续到下一个断点。5) 通过菜单选项状态 → 寄存器。6) 第一个是寄存器名称，第二个是值的十六进制表示，第三个是值的十进制表示（排除一些始终以十六进制显示的寄存器，如rip和rsp）。

7) 有多种退出调试器的方法，包括在命令窗口中键入 exit、点击左上角的 x 或使用菜单选项 文件 → 退出。

8) 有多种设置断点的方法，包括双击行、输入 b <lineNumber> 或输入 b <labelName>（如果目标行存在标签）。

9) 调试器命令从文件中读取命令是；source {v*}fileName{v*}。

10) 绿色箭头指向下一个要执行的指令。

11) 答案如下：

1. x/db &bVar12.
x/dh &wVar13. x/
dw &dVar1

4. x/dg &qVar1 5. x/
30db &bArr1 6. x/50
dh &wArr1 7. x/75d
w &dArr1

12) 答案如下:

1. x/xb &bVar1 2. x
/xh &wVar1 3. x/x
w &dVar1 4. x/xg
&qVar1 5. x/30xb
&bArr1 6. x/50xh &
wArr1 7. x/75xw &
dArr1

13) 命令是: x/ug $rsp 14) 命令
是: x/5ug $rsp

24.7 测验问题答案, 第7章

1) 答案如下:

1. 法规 2. 法规 3. 非法, 354 不适合放入一个字节 4. 法规 5. 非法, 大小不匹配 6. 非法, 无法更改值 54 7. 法规 8. 法规, 虽然合法但可能得到一个错误值 9. 法规 10. 法规

11. 法律，虽然合法，但可能会导致不正确的 ECT值

12. 非法，不能将内存移动到内存 13. 非法，不能将内存移动到内存 14. 合法 15. 非法，r16不是一个有效的寄存器 16. 合法

2) 答案如下：

1. 将 ***bVar1*** 处的字节值复制到 rsi 寄存器，将其视为无符号值，因此将高 56 位设置为 0。2. 将 ***bVar1*** 处的字节值复制到 rsi 寄存器，将其视为有符号值，因此对高 56 位进行符号扩展（负数为 1，正数为 0）。

3) 答案如下：

1. 将 ah 寄存器设置为 0 2. 扩展字节到字

4) 答案如下：

1. movzx eax, ax 2. cwde

5) 答案如下：

1. 将dx移动到 0 2. cwd

6) cwd 指令仅将 ax 中的有符号值转换为 dx:ax 中的符号值（以及其他无）。movsx 指令将字源操作数复制到双字目标操作数。

7) 在第一条指令中，必须显式指定目标操作数大小，因为源操作数，即立即值 1，本身没有与之关联的大小。在第二条指令中，目标操作数大小可以从源操作数（在这种情况下，eax寄存器是双字）中确定。

8) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 00092.
rbx = 0x0000 0000 0000 000B

9) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 00072.
rbx = 0x0000 0000 0000 0002

10) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 00092. rb
x = 0xFFFF FFFF FFFF FFF9

11) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 000C2.
rdx = 0x0000 0000 0000 0000

12) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 00012.
rdx = 0x0000 0000 0000 0002

13) 答案如下（为了清晰起见，每4个一组）：

1. rax = 0x0000 0000 0000 00022.
rdx = 0x0000 0000 0000 0003

14) 答案如下:

1. 目标操作数不能是立即数（42）。2. 由于无法确定大小/类型，不允许使用立即数操作数。3. mov指令不允许内存到内存的操作。4. 地址需要64位，无法放入ax寄存器中。

15) idiv指令将除以edx:eax，而edx未设置。

16) 除法的操作数是带符号的（-500），但使用了无符号除法。

17) 使用的单词 divide 将结果放入 dx:ax 寄存器，但使用 eax 寄存器来获取结果。

18) 三操作数乘法指令仅允许用于有限的一组有符号乘法操作。

24.8 测验问题答案，第8章

1) 第一条指令将 qVar1 中的值放入 rdx 寄存器。第二条指令将 qVar1 的地址放入 rdx 寄存器。

2) 答案如下:

1. 立即 2. 内存 3. 立即 4. 非法，目标操作数不能是立即值
5. 寄存器 6. 内存 7. 内存 8. 非法，源操作数和目标操作数大小不同。

3) 答案如下（按每组4个分组以清晰显示）:

1. eax = 0x0000 000A

4) 答案如下（按每组4个分组以清晰显示）:

1. eax = 0x0000 00032.
edx = 0x0000 0002

5) 答案如下（按每组4个分组以清晰显示）:

1. eax = 0x0000 00092. ebx = 0x0
000 00023. rcx = 0x0000 0000 00
00 00004. rsi = 0x0000 0000 0000
000C

6) 答案如下（按每组4个分组以清晰显示）:

1. rax = 0x0000 00102. rcx = 0x00
00 0000 0000 0000

3. edx = 0x0000 0000 4. rsi = 0x
0000 0000 0000 0004

7) 答案如下（按每组4个分组以清晰显示）：

1. eax = 0x0000 00022. rcx = 0x00
00 0000 0000 00003. edx = 0x000
0 00054. rsi = 0x0000 0000 0000
0003

8) 答案如下（为了清晰起见，每4个一组）：

1. eax = 0x0000 00182. edx = 0x0
000 00003. rcx = 0x0000 0000 00
00 00004. rsi = 0x0000 0000 0000
0005

24.9 测验问题答案，第9章

1) rsp寄存器。2) 首先，rsp = rsp - 8，然后rax寄存器被复制到[rsp]（按此顺序）。3) 8字节。4) 答案如下（按4个一组分组以清晰显示）：1. r10 = 0x0000 0000 0000 0003 2. r11 = 0x0000 0000 0000 0002 3. r12 = 0x0000 0000 0000 0001 5) 数组在内存中反转。6) 内存使用更高效。

24.10 测验问题答案，第10章

1) 解决问题所涉及的明确、有序的步骤序列。

2) 答案如下：1. 理解问题

2. 创建算法 3. 实现程序 4.
测试/调试程序

3) 不，这些步骤适用于任何语言或复杂问题（甚至超出编程）。4) 编译时错误。5) 编译时错误。6) 编译时错误。7) 运行时错误。

24.11 测验问题答案，第11章

1) 在顶部（在数据、BSS和文本部分之上）。2) 每次宏被调用时一次。3) %% 将确保每次使用宏时生成一个唯一的标签名称。4) 如果在标签上省略了%%，标签将被原样复制，因此看起来是重复的。5) 是的。这可以用来从宏退出到错误处理代码块（不在宏内部）。6) 宏参数替换发生在汇编时。

24.12 测验问题答案，第12章

1) 链接和论证传输。2) 调用和返回指令。3) 值调用。4) 引用调用。5) 只调用一次，无论调用多少次。6) 将当前rip放置在栈上，并将rip更改为被调用函数的地址。7) 保存和恢复被调用者保留寄存器的内容。

8) 输入：rdi, rsi, rdx, rcx, r8 和 r9。

9) In: edi, esi, edx, ecx, r8d 和 r9d。

10) 函数可以在不保存和恢复的情况下更改寄存器中的值。

11) rax、rcx、rdx、rsi、rdi、r8、r9、r10和r11中的两个。12) 调用帧、函数调用帧或活动记录。

13) 叶函数不调用其他函数。

14) 清除堆栈上的传入参数。

15) 二十四（24）自三个各8字节（8 x 3）的参数。

16) 偏移量是 [rbp+16]，无论哪个保存的寄存器被推入。

17) 可用内存。18) 引用调用。19) 7th参数位于[rbp+16]，8th参数位于[rbp+24]。20) 内存效率，因为栈上的动态局部变量仅在需要时（当函数正在执行时）使用内存。

24.13 测验问题答案，第13章

1) rax寄存器。2) 操作系统。3) 调用代码是SYS_write（1）。The 1st 参数在rdi中是输出位置STDOUT，2nd 参数在rsi中是输出字符的起始地址，3rd 参数在rdx中是要写入的字符数。4) 未知将输入多少个字符。5) rax寄存器将包含文件描述符。6) rax寄存器将包含错误代码。7) 输入：rdi、rsi、rdx、r10、r8和r9。

24.14 测验问题答案，第14章

1) 声明如下：extern func1, func2

2) 声明如下：extern func1, func23) 编译器将生成错误。4) 链接时间。5) 链接器将生成未满足的外部引用错误。6) 是的。然而，在调试器中，代码将不会显示。

24.15 测验问题答案，第15章

1) 缓冲区溢出漏洞通常称为 *stack smashing*。2) C 函数没有检查输入参数的数组边界。3) 是的。4) 当请求输入时输入大量字符，如果程序崩溃。5) 一系列用于使缓冲区溢出漏洞的目标更容易触发的 nop 指令。6) 许多可能的答案。删除文件、打开网络连接、终止进程等。7) 使用 canary、实现数据执行保护（DEP）和使用数据地址空间布局随机化。

24.16 测验问题答案，第16章

1) 操作系统。特别是，加载器。

2) 正在执行的程序。

3) 可执行文件的名称。4) `argc` 指的是参数计数，`argv` 指的是参数向量（表示每个参数的字符串地址表的起始地址）。5) 在 rdi 寄存器中。6) 在 rsi 寄存器中。7) 空格由操作系统移除，因此程序无需做任何事情。

8) 无。程序需要检查并确定是否存在错误。

24.17 测验问题答案，第1章 7

1) Linux的行结束字符是换行符（LF），而Windows的行结束字符是回车符、换行符（CR, LF）。

2) 存储信息的一个子集以快速访问。

它们位于语言I/O库函数中（即cout、cin等）。

4) 简化编程。

5) I/O性能改进。

6) 系统服务函数需要读取特定数量的字符，而这些字符数在读取“一行”文本之前是未知的。7) 在层次结构的下一级中保留信息的一个子集（这比下一级更快）。8) 减少了与过多系统读取相关的总线争用和内存延迟开销。9) 变量值必须在函数调用之间保持。10) 必须从实际读取的字符数推断文件结束。11) 可能有很多原因，包括在打开后另一个窗口中删除文件或打开后移除驱动器（USB）。12）实际读取的字符数将为0，必须显式检查。13）为确保传递的行缓冲区数组不会被覆盖。14）通过初始化变量以指示已读取所有缓冲区字符。

24.18 测验问题答案，第18章

1) 寄存器有：xmm0, xmm1, xmm2, . . ., xmm15。2) 单精度为4字节，双精度为8字节。3) 与二进制中0.1的不精确表示相关的累积舍入误差。4) 浮点函数返回值在xmm0中。5) 没有保留任何浮点寄存器。

24.19 测验问题答案，第19章

1) 并发意味着多个不同的（不一定相关）进程同时进行。并行处理意味着进程是同时执行的。

2) 分布式计算和多进程。

3) 在通过网络连接的不同计算机上。

4) 许多可能答案，包括Folding@Home和SETI@Home。网络搜索可以提供更完整的列表。

5) 在CPU的不同核心上。

6) 分布式计算允许非常大量的计算节点，但需要在具有固有通信延迟的网络上进行通信。7) 多处理允许通过共享内存实现进程之间非常快速的通信，但仅支持与可用核心数量相关的有限数量的同时执行线程。8) 多线程同时向共享变量写入，没有任何控制或协调。9) 不。由于变量没有被更改，不存在问题。10) 是的。由于变量正在被更改，一个线程可能在另一个线程获取值之后更改该值。

24.20 测验问题答案，第20章

1) 操作系统负责管理资源。资源包括CPU核心、主内存、辅助存储、显示屏、键盘和鼠标。2）改变处理器执行指令顺序的事件。3）由当前进程引起并需要内核注意的中断。4）ISR是当发生中断时执行的用于服务（执行所需操作）的中断服务例程。5）中断描述符表（IDT），其中包含中断服务例程（ISR）的地址和门信息。

6) 中断号乘以16。

7) ret指令将从堆栈中弹出返回地址并将其放入rip寄存器。iret指令将从堆栈中弹出返回地址和保留的标志寄存器内容，并将其放入rip寄存器和rFlag寄存器。8) call指令需要目标地址。由于ISR地址可能会因硬件更改或软件更新而更改，中断机制执行运行时查找ISR地址。9) 在执行代码的上下文中，无法预测中断时机，甚至无法预测中断是否可能发生。10) 在执行代码的上下文中，可以预测中断时机。这通常是系统服务调用或异常，例如除以0。11) 每条指令都会更改rFlag寄存器。中断完成后，必须将标志寄存器恢复到其原始值，以确保中断的过程能够恢复。

12) 许多可能的答案，包括键盘和鼠标等I/O设备、网络适配器、辅助存储设备或其他外围设备。

13) 许多可能的答案，包括除以0。

14) 在必须立即处理不可屏蔽中断的情况下，可能暂时忽略一个可屏蔽中断。

Appendix D – Quiz Question Answers

25.0 Alphabetical Index