

The Art Of Assembly Language, 2nd Edition

汇编语言的编程 艺术(第2版)

(美) Randall Hyde 著 马跃包战译



清华大学出版社



ASSEMBLY FOR THE NON-ASSEMBLY PROGRAMMER

汇编语言是一种低级编程语言，只比计算机本身的机器语言高一级。虽然汇编语言经常用于编写设备驱动程序、模拟器和视频游戏，但是许多程序员认为汇编语言的语法不太友好，很难学习和使用。

1996年以来，Randall Hyde的《汇编语言的编程艺术》一书为非汇编程序员提供了全面、易读和循序渐进的32位x86汇编语言的介绍。Hyde使用的主要教学工具是高级语言汇编器(High Level Assembler, HLA)，其中提供了许多高级语言(如C、C++和Java)的功能，以帮助读者快速掌握汇编语言的基本概念。HLA在允许汇编语言程序员编写真正低级代码的同时，也使他们能够利用高级语言编程的优势。

通过阅读《汇编语言的编程艺术(第2版)》，读者可以学到计算机科学的底层理论基础，并将所学知识转化为真正可以运行的代码。

本书内容

- ◆ 编辑、编译和运行HLA程序
- ◆ 声明和使用常量、标量变量、指针、数组、结构、联合和命名空间
- ◆ 转换算术表达式(整型和浮点型)
- ◆ 转换高级控制结构

本书是汇编语言学习者翘首以盼的《汇编语言的编程艺术》的第2版。与第1版相比，本书新增了反映HLA最新变化的内容，并介绍了如何支持Linux、Mac OS X和FreeBSD。汇编语言是一门复杂的低级语言，但是无论读者是否具有高级语言编程经验，都可以借助本书掌握它。

作者简介

Randall Hyde是*Write Great Code, Volumes 1*和*Volumes 2*的作者，并且与人合著了*MASM 6.0 Bible*。他为Dr. Dobb's Journal、Byte和多种专业刊物撰稿。他在加州大学河滨分校讲授汇编语言已经超过了十年的时间。



上架建议：程序设计/汇编语言、HLA
读者信箱：wkservice@vip.163.com
投稿邮箱：bookservice@263.net

ISBN 978-7-302-26373-9



9 787302 263739 >

定价：69.80元

汇编语言的编程艺术

(第2版)

(美) Randall Hyde 著

马跃 包战 译

清华大学出版社

北 京

Copyright © 2010 by Randall Hyde. Title of English-language original : The Art of Assembly Language, 2nd Edition, ISBN 978-1-59327-207-4, published by No Starch Press, Simplified Chinese-language edition copyright © 2011 by Tsinghua University Press Limited. All rights reserved.

本书中文简体字版由 No Starch Press 授权清华大学出版社出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2010-6326

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。
版权所有,翻印必究。举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

汇编语言的编程艺术(第2版)/(美)海德(Hyde, R.)著;马跃,包战译. —北京:清华大学出版社, 2011.12
书名原文: The Art of Assembly Language, 2nd Edition

ISBN 978-7-302-26373-9

I. 汇… II. ①海…②马…③包… III. 汇编语言—程序设计 IV. TP3F3

中国版本图书馆 CIP 数据核字(2011)第 157314 号

责任编辑:王军 于平

装帧设计:牛艳敏

责任校对:成凤进

责任印制:何芊

出版发行:清华大学出版社

<http://www.tup.com.cn>

社总机:010-62770175

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

地址:北京清华大学学研大厦 A 座

邮编:100084

邮购:010-62786544

印刷者:北京密云胶印厂

装订者:北京市密云县京文制本装订厂

经销:全国新华书店

开本:185×260 印张:37.25 字数:930 千字

版次:2011年12月第2版 印次:2011年12月第1次印刷

印数:1~4000

定价:69.80 元

产品编号:036892-01

目 录

第 1 章 进入汇编语言的世界	1
1.1 HLA 程序的结构	1
1.2 运行第一个 HLA 程序	3
1.3 基本的 HLA 数据声明	4
1.4 布尔值	6
1.5 字符值	6
1.6 Intel 80x86 CPU 系列简介	6
1.7 存储子系统	9
1.8 基本的机器指令	11
1.9 基本的 HLA 控制结构	14
1.9.1 HLA 语句中的布尔表达式	14
1.9.2 HLA 中的 if..then..elseif..else..endif 语句	16
1.9.3 布尔表达式中的逻辑与、 逻辑或以及逻辑非	18
1.9.4 while..endwhile 语句	20
1.9.5 for..endfor 语句	20
1.9.6 repeat..until 语句	21
1.9.7 break 和 breakif 语句	22
1.9.8 forever..endfor 语句	22
1.9.9 try..exception..endtry 语句	23
1.10 HLA 标准库入门	26
1.10.1 stdio 模块中的预定义 常量	28
1.10.2 标准输入和标准输出	29
1.10.3 stdout.newln 例程	29
1.10.4 stdout.putiX 例程	29
1.10.5 stdout.putiXSize 例程	29
1.10.6 stdout.put 例程	31
1.10.7 stdin.getc 例程	32
1.10.8 stdin.getiX 例程	33
1.10.9 stdin.readLn 和 stdin.flushInput 例程	34
1.10.10 stdin.get 例程	35
1.11 关于 try..endtry 的其他细节	35
1.11.1 try..endtry 嵌套语句	36
1.11.2 try..endtry 语句中不受保护 的子句	38
1.11.3 try..endtry 语句中的 anyexception 子句	40
1.11.4 寄存器与 try..endtry 语句	41
1.12 高级汇编语言与低级汇编语 言的比较	42
1.13 更多信息	43
第 2 章 数据表示	45
2.1 数字系统	45
2.1.1 回顾十进制系统	45
2.1.2 二进制数字系统	46
2.1.3 二进制格式	47
2.2 十六进制数字系统	47
2.3 数据结构	49
2.3.1 位	49
2.3.2 半字节	50
2.3.3 字节	51
2.3.4 字	52

2.3.5 双字	53	3.3 HLA 如何为变量分配内存	106
2.3.6 四字和长字	53	3.4 HLA 对数据对齐的支持	107
2.4 二进制数和十六进制数的 算术运算	54	3.5 地址表达式	109
2.5 关于数字及其表示	55	3.6 类型强制转换	111
2.6 位逻辑运算	57	3.7 寄存器类型强制转换	113
2.7 二进制数和位串的逻辑运算	59	3.8 栈段与 push 和 pop 指令	114
2.8 有符号数和无符号数	61	3.8.1 基本的 push 指令	114
2.9 符号扩展、零扩展、 压缩和饱和	65	3.8.2 基本的 pop 指令	115
2.10 移位和循环移位	68	3.8.3 用 push 和 pop 指令保护 寄存器	116
2.11 位域和压缩数据	72	3.9 栈的 LIFO 数据结构	117
2.12 浮点运算简介	76	3.9.1 其他的 push 和 pop 指令	118
2.12.1 IEEE 浮点格式	79	3.9.2 不使用出栈而从栈内移除 数据	119
2.12.2 HLA 为浮点数值提供的 支持	81	3.10 访问已入栈而未出栈的 数据	120
2.13 BCD 数据表示	84	3.11 动态内存分配和堆段	122
2.14 字符	85	3.12 inc 和 dec 指令	125
2.14.1 ASCII 字符编码	85	3.13 获取存储器对象的地址	125
2.14.2 HLA 对 ASCII 字符提供的 支持	88	3.14 更多信息	126
2.15 Unicode 字符集	91	第 4 章 常量、变量与数据类型	127
2.16 更多信息	92	4.1 一些额外的指令: intmul、 bound、into	127
第 3 章 存储器的访问与结构	93	4.2 HLA 常量和数值声明	131
3.1 80x86 的寻址方式	93	4.2.1 常量类型	134
3.1.1 80x86 寄存器寻址方式	94	4.2.2 字符串和字符字面常量	135
3.1.2 80x86 的 32 位存储器寻址 方式	94	4.2.3 const 段中的字符串常量与 文本常量	137
3.2 运行时存储器的结构	100	4.2.4 常量表达式	138
3.2.1 代码段	101	4.2.5 HLA 程序中的多个 const 段 以及它们的顺序	140
3.2.2 静态段	102	4.2.6 HLA 的 val 段	140
3.2.3 只读数据段	103	4.2.7 在程序中的任意位置修改 val 对象	141
3.2.4 存储段	103	4.3 HLA 的 type 段	142
3.2.5 @nostorage 属性	104	4.4 enum 和 HLA 枚举数据类型	143
3.2.6 var 段	104		
3.2.7 程序中声明段的结构	105		

4.5 指针数据类型	144	4.29 对齐记录中的字段	197
4.5.1 在汇编语言中使用指针	145	4.30 记录指针	198
4.5.2 在 HLA 中声明指针	146	4.31 联合	200
4.5.3 指针常量和指针常量 表达式	146	4.32 匿名联合	202
4.5.4 指针变量和动态内存分配	147	4.33 变体类型	203
4.5.5 指针的常见问题	147	4.34 命名空间	203
4.6 复合数据类型	151	4.35 汇编语言中的动态数组	206
4.7 字符串	151	4.36 更多信息	208
4.8 HLA 字符串	154	第 5 章 过程和单元	209
4.9 访问字符串中的字符	159	5.1 过程	209
4.10 HLA 字符串模块和其他与 字符串相关的例程	160	5.2 机器状态的保存	211
4.11 存储器内的转换	170	5.3 过程的提前返回	215
4.12 字符集	171	5.4 局部变量	215
4.13 在 HLA 中实现字符集	172	5.5 其他局部和全局符号类型	220
4.14 HLA 字符集常量和字符集 表达式	173	5.6 参数	220
4.15 HLA 标准库对字符集的支持	175	5.6.1 值传递	221
4.16 在 HLA 程序中使用字符集	177	5.6.2 引用传递	224
4.17 数组	178	5.7 函数和函数的结果	226
4.18 在 HLA 程序中声明数组	179	5.7.1 返回函数结果	227
4.19 HLA 数组常量	180	5.7.2 HLA 中的指令合成	227
4.20 访问一维数组的元素	181	5.7.3 HLA 过程的 @returns 选项	229
4.21 数组排序	182	5.8 递归	231
4.22 多维数组	183	5.9 过程的向前引用	235
4.22.1 以行为主排列	184	5.10 HLA v2.0 的过程声明	236
4.22.2 以列为主排列	187	5.11 过程的底层实现与 call 指令	236
4.23 多维数组的存储空间分配	187	5.12 过程与栈	238
4.24 汇编语言中多维数组元素的 访问	189	5.13 活动记录	240
4.25 记录	190	5.14 标准入口序列	242
4.26 记录常量	192	5.15 标准出口序列	244
4.27 记录数组	193	5.16 自动(局部)变量的底层实现	245
4.28 数组/记录作为记录字段	194	5.17 参数的底层实现	246
		5.17.1 在寄存器中传递参数	247
		5.17.2 在代码流中传递参数	249
		5.17.3 在栈中传递参数	251
		5.18 过程指针	269
		5.19 过程参数	272

5.20 无类型的引用参数	273	6.5.9 超越指令	329
5.21 管理大型程序	274	6.5.10 其他指令	331
5.22 #include 伪指令	274	6.5.11 整数操作	332
5.23 忽略重复的#include 操作	276	6.6 将浮点表达式转换成汇编语言	332
5.24 单元与 external 伪指令	276	6.6.1 将算术表达式转换成后缀表示法	334
5.24.1 伪指令 external 的行为	280	6.6.2 将后缀表达式转换成汇编语言	335
5.24.2 HLA 中的头文件	281	6.7 HLA 标准库对浮点算术运算的支持	336
5.25 命名空间污染	282	6.8 更多信息	337
5.26 更多信息	284		
第 6 章 算术运算	287	第 7 章 低级控制结构	339
6.1 80x86 的整数运算指令	287	7.1 低级控制结构	339
6.1.1 mul 和 imul 指令	287	7.2 语句标号	339
6.1.2 div 和 idiv 指令	290	7.3 无条件控制转移(jmp)	341
6.1.3 cmp 指令	292	7.4 条件跳转指令	343
6.1.4 setcc 指令	296	7.5 “中级”控制结构: jt 和 jf	346
6.1.5 test 指令	298	7.6 使用汇编语言实现常用控制结构	347
6.2 算术表达式	299	7.7 判定	347
6.2.1 简单赋值	300	7.7.1 if..then..else 序列	348
6.2.2 简单表达式	300	7.7.2 将 HLA 的 if 语句翻译成纯汇编语言	351
6.2.3 复杂表达式	302	7.7.3 使用完整布尔求值实现复杂的 if 语句	355
6.2.4 可交换运算符	307	7.7.4 短路布尔求值	356
6.3 逻辑(布尔)表达式	308	7.7.5 短路布尔求值与完整布尔求值	357
6.4 机器特性与运算技巧	309	7.7.6 汇编语言中 if 语句的高效实现	359
6.4.1 不使用 mul、imul 或 intmul 的乘法	310	7.7.7 switch/case 语句	363
6.4.2 不使用 div 或 idiv 的除法	311	7.8 状态机和间接跳转	372
6.4.3 使用 and 实现模 N 计数器	311	7.9 “意大利面条式”代码	375
6.5 浮点运算	312	7.10 循环	375
6.5.1 FPU 寄存器	312	7.10.1 while 循环	376
6.5.2 FPU 的数据类型	317		
6.5.3 FPU 的指令集	318		
6.5.4 FPU 的数据转移指令	318		
6.5.5 换算指令	320		
6.5.6 算术运算指令	322		
6.5.7 比较指令	327		
6.5.8 常量指令	329		

7.10.2	repeat..until 循环	377	8.3.2	80x86 的 daa 指令和 das 指令	441
7.10.3	forever..endfor 循环	378	8.3.3	80x86 的 aaa、aas、aam 和 aad 指令	442
7.10.4	for 循环	378	8.3.4	使用 FPU 的压缩十进制 算术操作	443
7.10.5	break 和 continue 语句	379	8.4	表	445
7.10.6	寄存器的使用与循环	383	8.4.1	通过表查找进行函数计算	445
7.11	性能提高	384	8.4.2	域调节	449
7.11.1	将结束条件判断放在 循环结尾	384	8.4.3	产生表	450
7.11.2	反向执行循环	386	8.4.4	表查找的性能	453
7.11.3	循环不变计算	387	8.5	更多信息	453
7.11.4	循环展开	388	第 9 章	宏与 HLA 编译时语言	455
7.11.5	归纳变量	389	9.1	编译时语言	455
7.12	HLA 中的混合控制结构	390	9.2	#print 和#error 语句	457
7.13	更多信息	392	9.3	编译时常量和变量	458
第 8 章	高级算术运算	393	9.4	编译时表达式和操作符	458
8.1	多精度操作	393	9.5	编译时函数	461
8.1.1	扩展精度操作的 HLA 标准库 支持	394	9.5.1	类型转换编译时函数	462
8.1.2	多精度加法运算	396	9.5.2	数值编译时函数	463
8.1.3	多精度减法运算	398	9.5.3	字符分类编译时函数	463
8.1.4	扩展精度比较操作	399	9.5.4	编译时字符串函数	463
8.1.5	扩展精度乘法操作	403	9.5.5	编译时符号信息	464
8.1.6	扩展精度除法操作	406	9.5.6	其他编译时函数	465
8.1.7	扩展精度 neg 操作	414	9.5.7	编译时文本对象的 类型转换	465
8.1.8	扩展精度 and 操作	415	9.6	条件编译(编译时判定)	467
8.1.9	扩展精度 or 操作	415	9.7	重复编译(编译时循环)	470
8.1.10	扩展精度 xor 操作	416	9.8	宏(编译时过程)	473
8.1.11	扩展精度 not 操作	416	9.8.1	标准宏	473
8.1.12	扩展精度移位操作	416	9.8.2	宏参数	475
8.1.13	扩展精度循环移位操作	419	9.8.3	宏中的局部符号	480
8.1.14	扩展精度 I/O	420	9.8.4	作为编译时过程的宏	482
8.2	对不同长度的操作数进行 操作	437	9.8.5	使用宏模拟函数重载	483
8.3	十进制算术运算	439	9.9	编写编译时“程序”	488
8.3.1	字面 BCD 常量	440	9.9.1	在编译时构造数据表	488

9.9.2 循环展开	492	12.2 HLA 中的类	541
9.10 在不同的源文件中使用宏	493	12.3 对象	543
9.11 更多信息	493	12.4 继承	545
第10章 位操作	495	12.5 重写	546
10.1 位数据	495	12.6 虚拟方法与静态过程	547
10.2 位操作指令	496	12.7 编写类方法和过程	548
10.3 作为位累加器的进位标志	502	12.8 对象实现	552
10.4 位串的压缩与解压缩	503	12.8.1 虚拟方法表	554
10.5 接合位组与分布位串	506	12.8.2 带继承的对象表示	556
10.6 压缩的位串数组	508	12.9 构造函数和对象初始化	560
10.7 搜索位	510	12.9.1 构造函数中的动态对象 分配	561
10.8 位的计数	512	12.9.2 构造函数和继承	563
10.9 倒置位串	515	12.9.3 构造函数的参数和过程 重载	566
10.10 合并位串	517	12.10 析构函数	566
10.11 提取位串	517	12.11 HLA 的_initialize_和_finalize_ 字符串	567
10.12 搜索位模式	519	12.12 抽象方法	572
10.13 HLA 标准库的位模块	520	12.13 运行时类型信息	574
10.14 更多信息	522	12.14 调用基类的方法	576
第11章 字符串指令	523	12.15 更多信息	577
11.1 80x86 字符串指令	523	附录 ASCII 字符集	579
11.1.1 字符串指令的操作过程	524		
11.1.2 rep/repe/repz 和 repnz/repne 前缀	524		
11.1.3 方向标志	525		
11.1.4 movs 指令	527		
11.1.5 cmps 指令	531		
11.1.6 scas 指令	534		
11.1.7 stos 指令	534		
11.1.8 lods 指令	535		
11.1.9 通过 lods 和 stos 构建复杂 的字符串函数	536		
11.2 80x86 字符串指令的性能	536		
11.3 更多信息	536		
第12章 类与对象	539		
12.1 通用原则	539		

第 1 章

进入汇编语言的世界



本章主要介绍汇编语言的入门知识，帮助您快速掌握编写汇编语言程

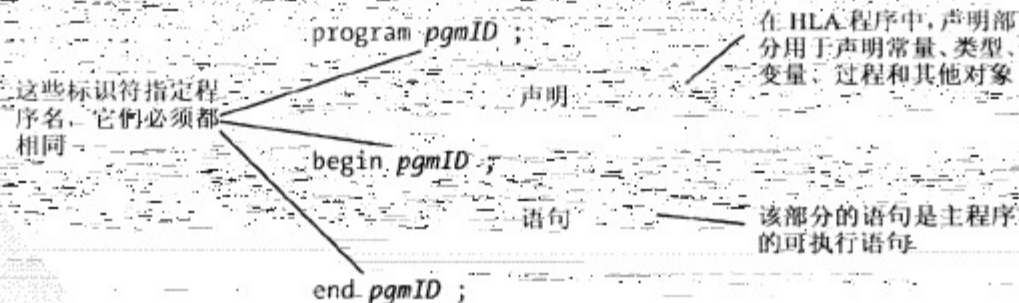
序的基本方法。主要包括以下内容：

- 给出 HLA(High Level Assembly, 高级汇编)程序的基本语法
- 介绍 Intel CPU 的体系结构
- 提供少量的数据声明、机器指令和高级控制语句
- 描述 HLA 标准库中可以调用的一些例程
- 展示如何编写简单的汇编语言程序

到本章结束时，您将会理解 HLA 程序的基本语法和在后面章节中学习汇编语言新特征的先决条件。

1.1 HLA 程序的结构

HLA 程序通常都采用如图 1-1 所示的形式。



program、begin 和 end 都是描述 HLA 程序的保留字。注意程序中分号的位置。

图 1-1 基本的 HLA 程序

上面模板中的 *pgmID* 是用户自定义的程序标识符。必须为程序选取一个适当的、具有描述性的名称。对于实际程序来说, *pgmID* 是一个很糟糕的名字。如果所编写的程序是课堂作业, 那么老师可能会给定主程序的名称。如果是自己编写的程序, 那么就必须为项目选择一个合适的名称。

HLA 中的标识符与大多数高级语言中的标识符非常相似。HLA 标识符可以以下划线或者字母开头, 后面可以接 0 个或者多个字母数字字符或下划线。HLA 的标识符是区分大小写的。这就意味着标识符对大小写是敏感的, 因此程序中的标识符必须使用相同的拼写格式(即使是大小写也必须一致)。但是, 与 C/C++ 这种区分大小写的语言不同的是, HLA 程序中不能声明只存在大小写差异的两个名称。

通常人们编写的第一个示例都是 Kernighan 和 Ritchie 在 *The C Programming Language* 一书中所推广的 Hello World 程序。对于正在学习一门新语言的人来说, 该程序是一个极好的例子。程序清单 1-1 给出了 HLA helloWorld 程序。

程序清单 1-1 helloWorld 程序

```
program helloWorld;
#include( "stdlib.hhf" );

begin helloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end helloWorld;
```

该程序中的 `#include` 语句用于告诉 HLA 编译器包含一组来自 `stdlib.hhf`(标准库, HLA 头文件) 的声明。这个文件包含该程序中使用的 `stdout.put` 代码的声明。

`stdout.put` 语句是 HLA 语言的输出语句。使用它可以将数据写入标准输出设备(一般是控制台)中。对于任何一个熟悉高级汇编语言的 I/O 语句的人来说, 应该确定该语句输出的肯定是“Hello, World of Assembly Language”。语句末尾的 `nl` 是一个常量, 它也是在 `stdlib.hhf` 中定义的, 相当于一个换行序列。

请注意分号跟在 `program`、`begin`、`stdout.put` 以及 `end` 语句之后。从技术上来说, `#include` 语句之后是不跟分号的。如果在 `#include` 语句后面加分号, 就可能导致创建一个出错的 `include` 文件, 所以应该养成在这里不加分号的习惯。

`#include` 是 HLA 声明的前导。事实上 `#include` 本身并不是一个声明, 但是它会让 HLA 编译器用 `stdlib.hhf` 文件直接代替 `#include` 指令, 这样就会在此处插入几个声明。大多数 HLA 程序一般都需要包含一个或多个 HLA 标准库头文件(`stdlib.hhf` 文件实际上把所有的标准库定义都包含到了程序当中)。

编译该程序就可以生成一个控制台应用程序。在命令窗口中运行这个程序就会输出指定的字符串, 然后将控制权返回命令行解释器(或者是 UNIX 平台上的 shell 解释器)。

HLA 语言的格式非常自由。因此可以将语句分开到多行中, 从而提高程序的可读性。例如, 在 Hello-World 程序中, 可以按如下方式编写 `stdout.put` 语句。

```
stdout.put
[
    "Hello, World of Assembly Language",
```

```

        nl
    );

```

从本书的示例代码中，还将看到 HLA 会把它在源文件中发现的邻接字符串常量自动拼接。因此上面的语句也等效于：

```

stdout.put
(
    "Hello, "
    "World of Assembly Language",
    nl
);

```

事实上，nl(换行序列)只是一个字符串常量而已，所以(从技术上讲)nl 与前面字符串之间的逗号是不必要的。您会经常看到上面的代码写成如下形式：

```

stdout.put( "Hello, World of Assembly Language" nl );

```

注意在字符串常量和 nl 之间是没有逗号的；这在 HLA 中是完全合法的。但是只适用于特定的常量，所以一般来说，逗号是不能丢的。第 4 章会详细解释它的原理。这里之所以讨论是因为在正式解释之前，您很可能会发现代码示例中已经使用了这个“技巧”。

1.2 运行第一个 HLA 程序

Hello World 程序的目的是提供一个简单的例子，让初学者领会如何使用该语言的编译工具和运行工具。的确，前面一节给出的 Hello World 程序可以帮助演示 HLA 程序的格式和语法，但是像 Hello-World 这样的程序，其真正的目的在于教您如何从头创建并运行一个完整的程序。虽然前面已经给出了一个 HLA 程序的布局，但并没有讨论如何编辑、编译和运行该程序。本节将会简要地讨论这些细节。

编译和运行 HLA 程序所需的软件都可以通过 <http://www.artofasm.com/> 或 <http://webster.cs.ucr.edu/> 下载。从快速导航面板中选择 High Level Assembly，然后从打开的页面中选择 Download HLA 链接。目前有适用于 Windows、Mac OS X、Linux 和 FreeBSD 的 HLA 版本。选择适合您的系统的 HLA 版本。从 Download HLA Web 页面中，还可以下载到本书使用的所有软件。如果 HLA 下载中不包含它们，那么可能需要在下载 HLA 和本书使用的软件时，下载 HLA 参考手册和 HLA 标准库参考手册。本书不会讨论整个 HLA 语言，也不会描述整个 HLA 标准库。在通过 HLA 学习汇编语言时，手头备有这些参考手册很有帮助。

本节没有介绍如何安装和设置 HLA 系统，因为所用的指令会随着时间的推移而改变。针对每种操作系统的 HLA 下载页面描述了安装和使用 HLA 的方法。请参考这些指示来了解具体的安装过程。

创建、编译和运行 HLA 程序与使用其他语言编写程序时所采用的过程都相似。首先，因为 HLA 不是一种集成开发环境(IDE)，所以无法在同一个程序内编辑、编译、测试与调试，以及运行

应用程序(很多软件开发工具都是这样)。因此, 创建和编辑 HLA 程序时需要一个文本编辑器。¹

Windows、Mac OS X、Linux 和 FreeBSD 都提供了许多文本编辑器供您选择。甚至可以使用其他 IDE 所提供的文本编辑器来创建和编辑 HLA 程序(例如, Visual C++、Borland 的 Delphi、Apple 的 Xcode 等)。唯一的限制就是 HLA 需要使用 ASCII 文本文件, 所以所使用的编辑器必须能够操作和保存文本文件。注意, 在 Windows 下总可以使用记事本来创建 HLA 程序; 而在 Mac OS X 下则可以使用 XCode 或 Text Wrangler 等。

HLA 编译器² 是一个传统的命令行编译器(command-line compiler)。这就意味着需要在 Windows 的命令行提示符(command-line prompt)下或者从 Linux/FreeBSD/Mac OS X 的 shell 中运行它。因此, 需要在命令行提示符或者 shell 窗口中输入如下命令:

```
hla hw.hla
```

该命令告诉 HLA 将 hw.hla(helloWorld)程序编译为可执行文件。假设这里没有错误, 可以把下面的命令输入到命令提示窗口(在 Windows 下)中来运行程序:

```
hw
```

或者在 shell 解释器窗口中输入(在 Linux/FreeBSD/Mac OS X 下):

```
./hw
```

如果不能使程序正确编译和运行, 请参阅 HLA 下载页面中的 HLA 安装指示。这些指令详细描述了如何安装、设置和使用 HLA。

1.3 基本的 HLA 数据声明

HLA 提供了许多常量、类型和数据声明语句。后面的章节会更详细地介绍声明部分, 但是了解如何在 HLA 程序中声明一些简单的变量非常重要。

HLA 预定义了多种不同的带符号整数类型, 包括 int8、int16 和 int32, 分别对应 8 位(一个字节)、16 位(两个字节)和 32 位(四个字节)带符号整数类型。³ 变量声明通常在 HLA 静态变量部分出现。图 1-2 展示了变量声明通常采用的格式。

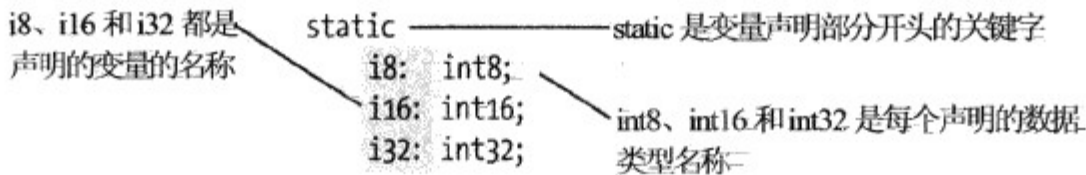


图1-2 静态变量声明

熟悉 Pascal 语言的人对这种声明语法应该比较适应。这个例子展示了如何声明 3 个不同的整

¹ HIDE(HLA Integrated Development Environment)是 Windows 用户可以使用的一种 IDE。更多信息可以在下载 HIDE 时查看 High Level Assembly Web 页面。
² 传统上, 程序员总是将汇编语言的翻译器叫做汇编器, 而不是编译器。但是, 由于 HLA 的高级特征, 把 HLA 称为编译器比汇编器更恰当。
³ 如果您对这些术语不熟悉, 可以参考第 2 章。关于位和字节的知识将在第 2 章介绍。

型数 i8、i16 和 i32。当然，在实际的程序中还是应该使用更具描述性的变量名。像 i8 和 i32 这样的名称只描述对象的类型，并没有描述它们的目的。变量名应该描述对象的目的。

在静态声明部分，也可以赋给变量一个初始值，操作系统把程序载入内存时会把这个初始值赋给变量。图 1-3 给出了这一语法。

```

static
  i8: int8 := 8;
  i16: int16 := 1600;
  i32: int32 := 320000;
  
```

图 1-3 静态变量初始化

赋值操作符(:=)后面的表达式必须为常量表达式。在静态变量声明中无法为其他变量赋值。

熟悉其他高级语言(尤其是 Pascal)的用户应该注意，在每条语句中只能声明一个变量。也就是说，HLA 不允许在用逗号分隔的变量名列表后面出现冒号和类型标识符。每个变量声明都由一个标识符、冒号、类型 ID 和分号组成。

程序清单 1-2 给出了一个简单的 HLA 程序，展示了 HLA 程序中变量的用法。

程序清单 1-2 变量声明与用法

```

program DemoVars;
#include( "stdlib.hhf" )

static
  InitDemo:      int32 := 5;
  NotInitialized: int32;

begin DemoVars;

  // Display the value of the pre-initialized variable:
  stdout.put( "InitDemo's value is:", InitDemo, nl );

  // Input an integer value from the user and display that value:
  stdout.put( "Enter an integer value: " );
  stdin.get( NotInitialized );
  stdout.put( "You entered: ", NotInitialized, nl );

end DemoVars;
  
```

除了静态变量声明以外，这个例子还引入了 3 个新的概念。首先，stdout.put 语句允许有多个参数。如果指定了一个整型值，那么在输出时，stdout.put 会把这个值转换为它的字符串形式。

程序清单 1-2 中引入的第 2 个新特征便是 stdin.get 语句。该语句从标准输入设备(通常是键盘)读入一个值，把它转换为整型数，然后再将该整型值存储到 NotInitialized 变量中。最后，该程序引入了一种 HLA 注释的语法。HLA 编译器忽略从“//”开始直到当前行末尾的所有文本。熟悉 Java、C++ 和 Delphi 的读者应该都认识这些注释。

1.4 布尔值

HLA 与 HLA 标准库为布尔对象提供了有限的支持。可以声明布尔变量,使用布尔字面常量,在布尔表达式中使用布尔变量,并输出布尔变量的值。

布尔字面常量由两个预定义的标识符 `true` 和 `false` 组成。在系统内部,HLA 用数值 1 代表真值;用数值 0 代表假值。大多数程序都把 0 当作假值,而把其他的任意值都当作真值,所以在 HLA 中 `true` 和 `false` 的表示应该是充分的。

为了声明一个布尔变量,可以使用 `boolean` 数据类型。HLA 使用一个字节(它可以分配的最少内存空间)来表示布尔值。下面的例子给出了一些典型的声明:

```
static
    BoolVar:      boolean;
    HasClass:     boolean := false;
    IsCFear:      boolean := true;
```

该示例说明,可以初始化布尔变量。

由于布尔变量是字节对象,因此可以使用能直接操作 8 位值的任何指令来处理它们。而且,只要确信布尔变量只包含 0 和 1(分别代表 `false` 和 `true`),就可以使用 80x86 的 `and`、`or`、`xor` 以及 `not` 指令来处理这些布尔值(第 2 章将介绍这些指令)。

可以调用 `stdout.put` 例程来输出布尔值,例如:

```
stdout.put( BoolVar );
```

该程序输出文本 `true` 还是 `false` 取决于布尔参数的值(0 代表 `false`,其他任何值代表 `true`)。注意 HLA 标准库不允许通过 `stdin.get` 来读取布尔值。

1.5 字符值

HLA 允许使用 `char` 数据类型来声明单字节的 ASCII 字符对象,也可以使用单引号括起来的字面字符值来初始化字符变量。下面的例子展示了在 HLA 中声明和初始化字符变量的方法:

```
static
    c: char;
    LetterA: char := 'A';
```

可以使用 `stdout.put` 例程来输出字符变量,也可以使用 `stdin.get` 过程调用来读取字符变量。

1.6 Intel 80x86 CPU 系列简介

至此,您已经看到了两个可以实际编译和运行的 HLA 程序。然而到目前为止,出现在程序中的所有语句要么是数据声明,要么是对 HLA 标准库例程的调用,还不是真正的汇编语言。在

我们学习真正的汇编语言之前，必须理解 Intel 80x86 CPU 器系列的基本结构，否则机器指令将没有任何意义。

Intel CPU 系列一般都被归为冯·诺依曼式机器。冯·诺依曼计算机系统包括 3 个主要模块：中央处理单元(CPU)、存储器和输入/输出设备(I/O)。这三部分通过系统总线(由地址、数据和控制总线组成)相连。图 1-4 中的模块图表示了这种关系。

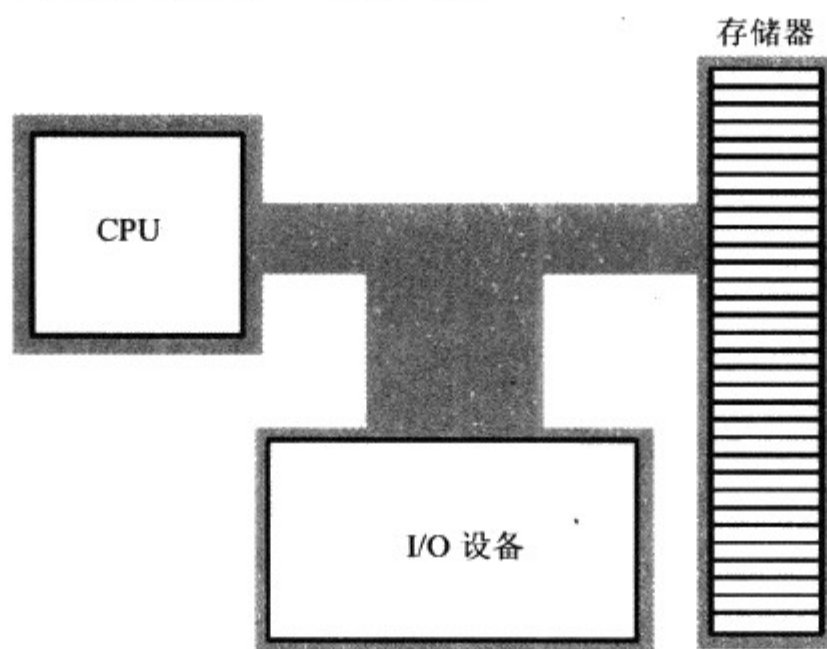


图 1-4 冯·诺依曼计算机系统模块图

CPU 与存储器和 I/O 设备之间的通信方法是在地址总线上放一个数值来选取一个存储单元或者 I/O 设备端口，它们都有唯一的二进制地址。然后，CPU、存储器以及 I/O 设备通过将数据放到数据总线上来彼此传递数据。控制总线则包含用于确定数据传输方向(进出存储器以及进出 I/O 设备)的信号。

80x86 CPU 的寄存器可以分成 4 类：通用寄存器、特殊目的寄存器、段寄存器以及特殊目的的核心模式寄存器。段寄存器在现代的 32 位操作系统(例如 Windows, Mac OS X、FreeBSD 和 Linux)中用得不多；由于本书专门针对 32 位操作系统编写程序，所以不需要讨论段寄存器。特殊目的的核心模式寄存器专门用来编写操作系统、调试器以及其他系统级工具。这些软件的构建远远超出了本书的范围。

80x86(Intel 系列)CPU 提供了几个通用寄存器。其中包含 8 个 32 位寄存器，如下所示：

EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP

每个名称的前缀 E 代表扩展(extended)的意思。该前缀将 32 位寄存器与如下所示的 8 个 16 位寄存器区分开来：

AX、BX、CX、DX、SI、DI、BP 和 SP

最后，80x86 CPU 还提供了 8 个 8 位的寄存器，它们的名称如下所示：

AL、AH、BL、BH、CL、CH、DL 和 DH

这些并非都是分离的寄存器。即 80x86 并未提供 24 个独立的寄存器。实际上，80x86 将 16 位寄存器重叠于 32 位寄存器之上，也将 8 位寄存器重叠于 16 位寄存器之上。图 1-5 给出了这种

关系。

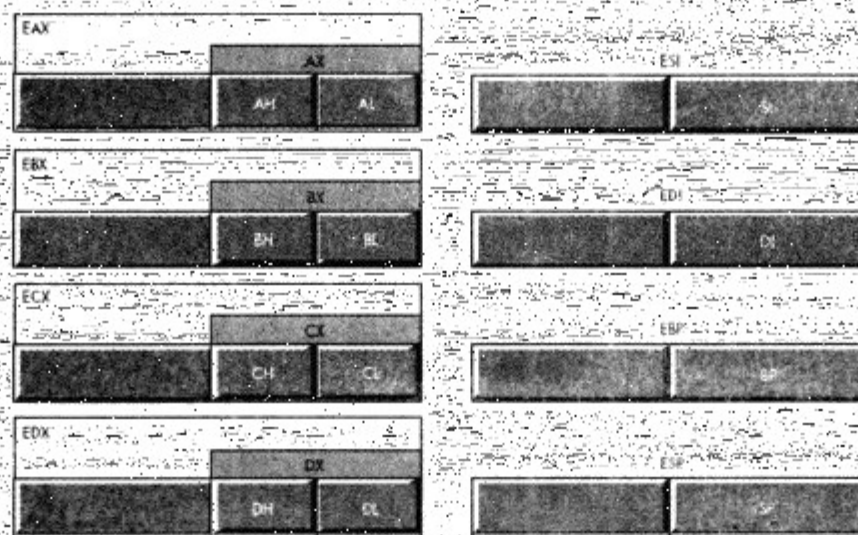


图 1-5 80x86(Intel CPU)的通用寄存器

关于通用寄存器,需要注意的是它们并不独立。修改一个寄存器可能会影响其他三个寄存器。例如,对 EAX 的修改可能会影响寄存器 AL、AH 以及 AX。这一点相当重要。在汇编语言初学者所编写的程序中,一个很常见的错误就是寄存器值的破坏,因为这些编程人员尚未完全理解图 1-5 的含义。

EFLAGS 寄存器是一个 32 位寄存器,它包含几个 1 位的布尔值(true/false)。EFLAGS 寄存器中的大多数位要么是为核心模式(操作系统)函数保留的,要么就是程序员不感兴趣的。其中有 8 位(或标志)是程序员编写汇编语言程序需要用到的。它们分别是溢出标志、方向标志、中断禁止标志⁴、符号标志、零标志、辅助进位标志、奇偶标志以及进位标志。图 1-6 显示了 EFLAGS 寄存器低 16 位中标志的布局情况。

在应用程序员感兴趣的 8 个标志中,有 4 个标志特别重要:溢出标志、进位标志、符号标志以及零标志。这 4 个标志统称为条件码。⁵根据这些标志的状态就可以测试前一次计算的结果。例如,在对两个值进行比较以后,条件码标志就会告诉您其中一个值是小于、等于还是大于另一个值。

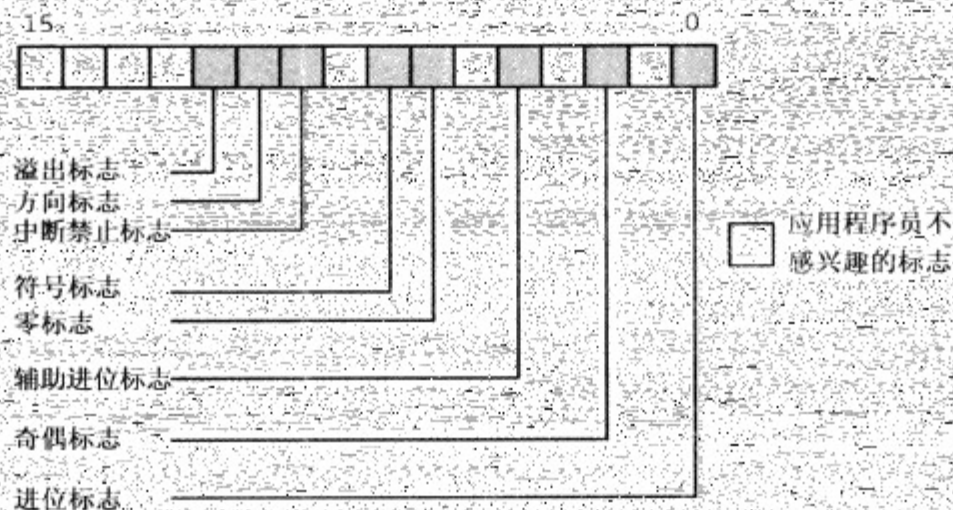


图 1-6 FLAGS 寄存器的布局(EFLAGS 的低 16 位)

令汇编语言初学者惊讶的一个重要事实是,在 80x86 CPU 中进行的所有计算几乎都和寄存器

⁴ 应用程序不能修改中断标志,但是因为我们将在第2章用到这一标志,因此在这里要对它进行讨论。

⁵ 从技术上讲,奇偶标志也是一个条件码,但是我们在本书中不会用到。

有关。例如，要将两个变量相加，并把它们的和存入第3个变量中，必须先将其中一个变量装入一个寄存器中，并将第2个操作数和寄存器中的值相加，然后将寄存器中的值存入目的变量。寄存器在每次计算中都充当媒介。因此，在80x86汇编语言程序中，寄存器是非常重要的。

还有一点很重要，虽然某些寄存器被称为是“通用”的，但并不能就此推断每个寄存器都能随意使用。所有80x86寄存器都有它们自己特殊的目的，因此它们都被限定在一个范围内使用。以寄存器对SP/ESP为例，它就有特殊的目的，而不能用于任何其他的目的(它是栈指针)。同样，寄存器BP/EBP也具有特殊目的，它的使用受到限制而无法当作通用寄存器使用。目前，应该避免将ESP和EBP寄存器用于一般的计算中；还应该记住，余下的寄存器在程序中不可以完全互换。

1.7 存储子系统

运行32位操作系统的80x86处理器可以访问多达 2^{32} 个不同的存储单元，或者40多亿字节。在几年以前，4G字节的存储器是最大的；然而，现代的机器已经远远超出了这一限制。由于在使用像Windows、Mac OS X、FreeBSD和Linux这样的32位操作系统时，80x86体系结构支持最大4G字节的地址空间，所以下面的讨论将假设最大有4G字节的空间。

当然，您应该问的第一个问题是：“确切地说，什么是一个存储单元？”80x86支持字节可寻址存储器(byte addressable memory)。因此基本的存储单元就是一个字节，这足以容纳一个单字符或是一个(非常)小的整数值(第2章将更详细地讨论这一点)。

将存储器看做是字节的线性数组。首字节的地址为0，末字节的地址为 $2^{32}-1$ 。对于80x86处理器来说，下面的伪Pascal数组声明就是一个对存储器非常好的比拟：

```
Memory: array [0..4294967295] of byte;
```

C/C++和Java用户可能偏爱下面这种语法：

```
byte Memory[4294967296];
```

为了执行Pascal语句Memory[125]:=0;的等效语句，CPU将数值0放在数据总线上，将地址125放到地址总线上，并设置写信号线(通常要将该线置0)，如图1-7所示。

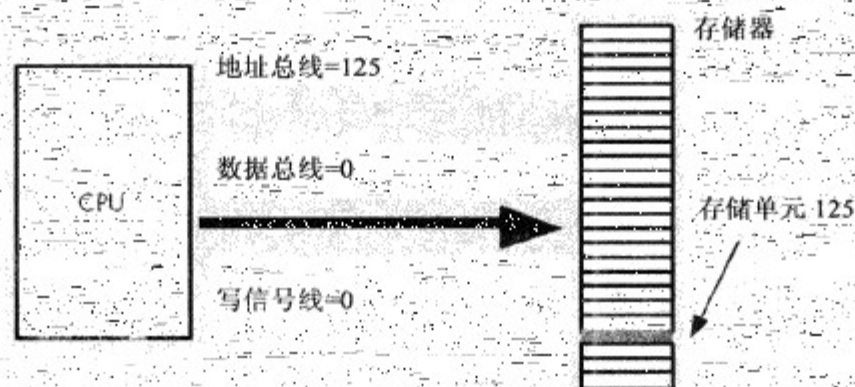


图1-7 存储器的写操作

若执行CPU:=Memory[125];的等效语句，CPU会将地址125放到地址总线上，并设置读信号线(因为CPU正在从存储器读取数据)，然后从数据总线上读取数据结果(如图1-8所示)。

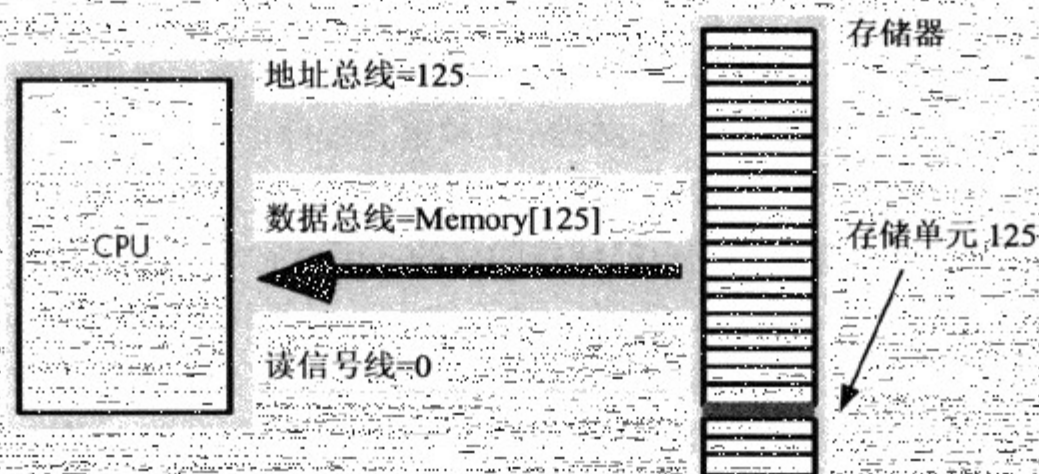


图 1-8 存储器的读操作

这里的讨论只适用于访问存储器中单个字节的情况。那么当处理器访问一个字或者双字时又会如何呢？由于存储器由一个字节数组构成，那么我们应该怎样处理大于一个字节的数呢？很简单，要存储更大的值，80x86 使用连续的存储单元序列就可以了。图 1-9 给出了 80x86 如何在存储器中存储字节、字(双字节)以及双字(四字节)。每个对象的内存地址就是它的首字节地址(最低地址)。

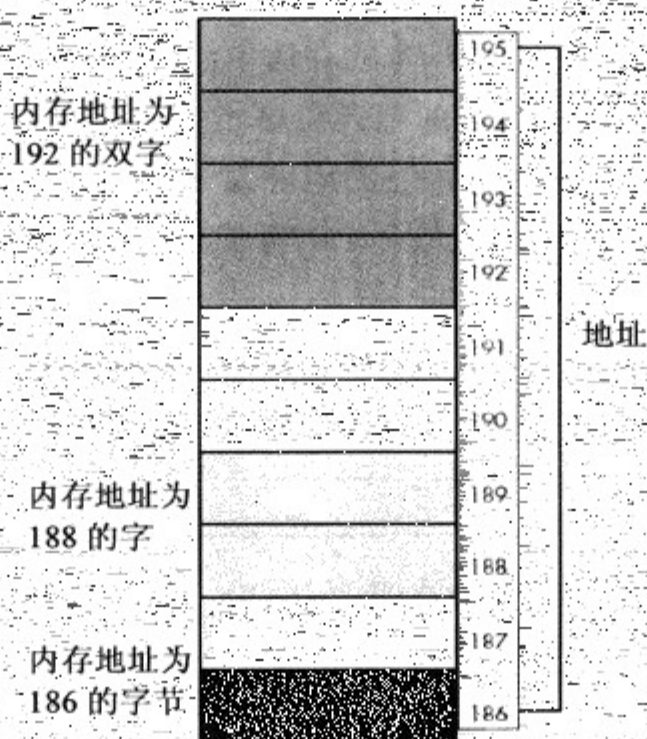


图 1-9 存储器中字节、字和双字的存储

现代的 80x86 处理器实际上并不直接与存储器相连。而是 CPU 上有一个专门的存储缓冲器，称为高速缓存(cache)，它充当 CPU 与主存储器之间的高速媒介。虽然高速缓存可以自动处理细节，但应该明白一个事实，即如果对象的地址为对象长度的整数倍，那么访问内存中的数据对象的效率更高。因此，用 4 的整数倍的地址来对齐四字节(双字)对象是个好主意。同样，让双字节对象与偶数地址对齐是最有效的。而对于单字节对象，则可以在任意地址高效地访问。本书 3.4 节将学习如何设置内存对象的对齐。

在结束讨论内存对象以前，应该理解内存与 HLA 变量之间的相关性。使用像 HLA 这样的汇编器/编译器的一个好处是不必担心内存地址值。只须在 HLA 中声明一个变量，HLA 负责将该变

量与唯一一组内存地址集合相关联。例如，对于下面的声明段：

```
static
    i8 :int8;
    i16 :int16;
    i32 :int32;
```

HLA 将在内存中找到某个未使用的 8 位字节，将其与变量 i8 相关联；还将找到一对连续未使用的字节，并将它们与 i16 相关联。最后，HLA 将找到 4 个连续未使用的字节，并将 i32 的值与这 4 个字节(32 位)相关联。以后就可通过变量的名称来引用它们，而不必关心它们的地址值。但是，您仍然应该知道 HLA 在后台完成了这些工作。

1.8 基本的机器指令

80x86 CPU 系列根据定义机器指令的不同方式提供了大量不同的机器指令。幸运的是，您不必知道所有的机器指令。事实上，大多数汇编语言程序可能只用到大约 30 条不同的机器指令。⁶ 只用少量的机器指令就完全可以编写一些有意义的程序。本节的目的在于提供一些机器指令，由此马上就可以开始编写简单的 HLA 汇编语言程序。

毫无疑问，mov 指令是最常用的语句。在一个程序中，25%~40%的指令都是 mov 指令。顾名思义，该指令是将数据从一个位置移到另一个位置。⁷ 该指令的 HLA 语法如下所示：

```
mov( source_operand, destination_operand );
```

source_operand 可以是一个寄存器、内存变量或者常量。*destination_operand* 可以是一个寄存器或者内存变量。从技术上来讲，80x86 指令集不允许两个操作数都为内存变量。但是，HLA 会自动将一条两字或者说双字的内存操作数的 mov 指令翻译为一对指令，它们将数据从一个位置复制到另一个位置。在 Pascal 或 C/C++ 等高级语言中，mov 指令大致等效于下面的赋值语句：

```
destination_operand = source_operand;
```

也许，对 mov 指令操作数的主要限制是它们必须具有相同的长度。也就是说，可以在两个字节(8 位)，两个字(16 位)或者两个双字(32 位)对象之间移动数据。但两个操作数的长度必须相同。表 1-1 列出了 mov 指令的所有合法组合。

您应该仔细地研究这个表，因为大多数通用 80x86 指令都使用与此相同的语法。

⁶ 不同的程序可能要使用 30 条不同的指令，但很少有用到多于 30 条指令的情况。

⁷ 从技术上来讲，mov 实际上是将数据从一个位置复制到另一个位置。它并不破坏源操作数中的原有数据。也许该指令改名为 copy 更好一些。但现在改已经为时过晚了。

表 1-1 合法的 80x86 mov 指令操作数

源操作数	目的操作数
Reg ₈ *	Reg ₈
Reg ₈	Mem ₈
Mem ₈	Reg ₈
Constant*	Reg ₈
Constant	Mem ₈
Reg ₁₆	Reg ₁₆
Reg ₁₆	Mem ₁₆
Mem ₁₆	Reg ₁₆
constant	Reg ₁₆
constant	Mem ₁₆
Reg ₃₂	Reg ₃₂
Reg ₃₂	Mem ₃₂
Mem ₃₂	Reg ₃₂
Constant	Reg ₃₂
Constant	Mem ₃₂

*后缀表示寄存器或存储单元的长度。

*该常量必须足够小以适合指定的目的操作数。

80x86 的 add 和 sub 指令分别将两个操作数相加和相减。它们的语法与 mov 指令几乎一致：

```
add( source_operand, destination_operand );
sub( source_operand, destination_operand );
```

add 和 sub 指令的操作数采用与 mov 指令相同的格式。⁸add 指令完成如下操作：

```
destination_operand = destination_operand + source_operand ;
destination_operand += source_operand ; // For those who prefer C syntax
```

sub 指令完成如下操作：

```
destination_operand = destination_operand - source_operand ;
destination_operand -= source_operand ; // For C fans.
```

事实上，仅用这 3 条指令加上下一节将要讨论的 HLA 控制结构就可以编写一些复杂的程序。程序清单 1-3 给出的 HLA 程序示例演示了这 3 条指令。

⁸ add 和 sub 指令不支持“内存对内存”的操作。

程序清单 1-3 mov、add 以及 sub 指令

```
program DemoMOVaddSUB;

#include( "stdlib.hhf" )

static
    i8: int8    := -8;
    i16: int16   := -16;
    i32: int32   := -32;

begin DemoMOVaddSUB;

    // First, print the initial values
    // of our variables.

    stdout.put
    (
        nl,
        "Initialized values: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );
    // Compute the absolute value of the
    // three different variables and
    // print the result.
    // Note: Because all the numbers are
    // negative, we have to negate them.
    // Using only the mov, add, and sub
    // instructions, we can negate a value
    // by subtracting it from zero.

    mov( 0, al ); // Compute i8 := -i8;
    sub( i8, al );
    mov( al, i8 );

    mov( 0, ax ); // Compute i16 := -i16;
    sub( i16, ax );
    mov( ax, i16 );

    mov( 0, eax ); // Compute i32 := -i32;
    sub( i32, eax );
    mov( eax, i32 );

    // Display the absolute values:

    stdout.put
    (
        nl,
        "After negation: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    )
```



```

);

// Demonstrate add and constant-to-memory
// operations:
add( 32323200, i32 );
stdout.put(_nl, "After add: i32=", i32, _nl );

end DemoMOVaddSUB;

```

1.9 基本的 HLA 控制结构

mov、add 和 sub 指令虽然很有价值，但是仅用它们编写程序还是不够的。在编写比较复杂的程序之前，需要在 HLA 程序中补充一些能够做决策和创建循环的指令。HLA 提供了几种高级控制结构，它们与高级语言中的控制结构非常相似，包括 if..then..elseif..else..endif、while..endwhile、repeat..until 等。学会这些语句之后就可以编写“真正”的程序了。

在讨论这些高级控制结构以前，有一点非常重要的事实需要指出，这些并非真正的 80x86 汇编语言语句。HLA 会将这些语句编译为一个或者多个汇编语言语句组成的序列。在第 7 章将学习 HLA 如何编译这些语句，以及如何编写一个不使用它们的纯汇编语言代码。但是，在这以前还有很多需要学习，所以现在还是要坚持使用这些高级语言语句。

另一个需要提出的重要事实是 HLA 的高级控制结构并不像它们看起来那么高级。HLA 高级控制结构真正的目的在于让您快速编写汇编程序，而不是让您避免使用汇编语言。很快您就会发现这些语句有一些严重的局限性，不能满足您的需要。一旦您对 HLA 的高级控制结构适应到一定程度，并且认为需要更强大的功能，那么就应该学习这些语句后面隐含的 80x86 指令。

不要被 HLA 中出现的与高级语言类似的语句迷惑。许多人在知道 HLA 中有这些语句后，错误地认为 HLA 只是一种特殊的高级语言，而不是真正的汇编语言。事实并不是这样。HLA 完全是一个低级汇编语言，支持其他任何 80x86 汇编器支持的机器指令。区别在于，HLA 包含了一些附加的语句，提供了比其他 80x86 汇编器更多的功能。在使用 HLA 学习 80x86 汇编语言之后，可以选择忽略这些附加(高级)语句，而只编写低级 80x86 汇编语言代码。

后面的部分假定您至少熟悉一种高级语言。我们将基于这一前提来介绍 HLA 控制语句，而不会劳神去解释怎样使用这些语句来完成程序中的任务。本书假设您已经知道在高级语言中如何使用这些控制语句；在 HLA 程序使用它们时的方式与在高级语言中使用的方式相同。

1.9.1 HLA 语句中的布尔表达式

有几条 HLA 语句需要一个布尔(真或者假)表达式来控制它们的执行。这样的例子包括 if、while 和 repeat..until 语句。这些布尔表达式的语法代表了 HLA 高级控制结构的最大限制。在这里您对高级语言太熟悉反而会起反作用：您更希望使用高级语言中用到的复杂表达式，而 HLA 只支持一些基本的形式。

HLA 布尔表达式总是采用下面的格式：⁹

```
flag_specification
!flag_specification
register
!register
Boolean_variable
!Boolean_variable
mem_reg_relop mem_reg_const
register in LowConst..HiConst
register not in LowConst..HiConst
```

flag_specification 可以是表 1-2 中所描述的符号之一。

表 1-2 flag_specification 所使用的符号

符 号	意 义	解 释
@c	进位	进位标志置位(1)时为真，清零(0)时为假
@nc	无进位	进位标志清零(0)时为真，置位(1)时为假
@z	零	零标志置位时为真，清零时为假
@nz	非零	零标志清零时为真，置位时为假
@o	溢出	溢出标志置位时为真，清零时为假
@no	未溢出	溢出标志清零时为真，置位时为假
@s	符号	符号标志置位时为真，清零时为假
@ns	无符号	符号标志清零时为真，置位时为假

在布尔表达式中使用标志值是很先进的。在第 2 章中，您将看到如何使用这些布尔表达式的操作数。

寄存器操作数可以是任意的 8 位、16 位或者 32 位的通用寄存器。如果寄存器中包含一个零，那么表达式值为假；如果寄存器中是一个非零值，它的值就为真。

如果将一个布尔变量指定为表达式，那么程序就会测试它是零(假)还是非零(真)。HLA 使用值 0 和 1 来分别代表假和真。注意，HLA 要求这种变量是 `boolean` 类型的，它拒绝其他的数据类型。若想测试其他类型是零还是非零，那么可以使用下一部分将讨论的通用布尔表达式。

HLA 布尔表达式最通用的形式就是具有两个操作数和一个关系运算符。表 1-3 列出了一些合法的组合。

⁹ 在第 6 章中，我们会介绍几种附加的格式。

表 1-3 合法的布尔表达式

左 操 作 数	关系运算符	右 操 作 数
内存变量或寄存器	=或==	内存变量、寄存器或常量
	<或!=	
	<	
	<=	
	>	
	>=	

注意，两个操作数不能都是内存操作数。事实上，如果将右操作数看作源操作数，将左操作数看作目的操作数，那么这两个操作数的类型必须与指令 `add` 和 `sub` 允许的类型相同。

两个操作数的长度必须相同，这一点与 `add` 和 `sub` 指令的要求也相同。也就是说，它们必须要么都是字节操作数，要么都是字操作数或者双字操作数。如果右操作数是个常数，它的值必须与左操作数都在相同的范围内。

还有另外一个问题：如果左操作数是一个寄存器，右操作数是一个正的常数或者另一个寄存器，HLA 就采用无符号比较。第 2 章将讨论其中的一个分支；暂时不要将一个存于寄存器中的负数与一个常数或者另一个寄存器相比较。这样得不到直观的结果。

`in` 和 `not in` 的操作数允许通过测试寄存器来判断它的值是否在指定范围内。例如，如果寄存器 `EAX` 中的值在 2000~2099 之间(包括边界值)，那么表达式 `eax in 2000..2099` 的值为 `true`。操作符 `not in`(两个字)检查寄存器中的值是否处于指定范围之外。例如，如果寄存器 `AL` 中的字符不是小写字母，那么 `al not in 'a'..'z'` 的值为 `true`。

这里有一些 HLA 中合法的布尔表达式的例子：

```
@c
Bool_var
al
esi
eax < ebx
ebx > 5
i32 < -2
i8 > 128
al < i8
eax in 1..100
ch not in 'a'..'z'
```

1.9.2 HLA 中的 if..then..elseif..else..endif 语句

HLA 中的 `if` 语句使用如图 1-10 所示的语法。

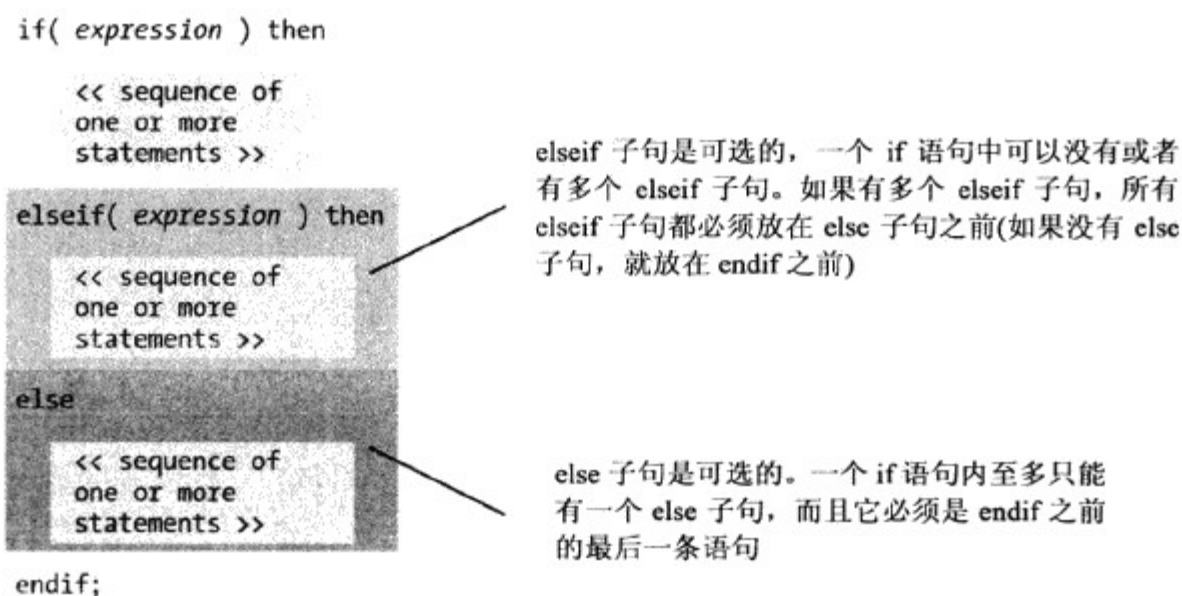


图 1-10 HLA 中 if 语句的语法

if 语句中的表达式必须采用前面所提到的形式之一。如果布尔表达式的值为 **true**，那么就执行 **then** 之后的代码，否则就转向执行语句中的下一个 **elseif** 或者 **else** 子句。

因为 **elseif** 和 **else** 子句是可选的，所以 if 语句可以采取单一的 **if..then** 子句的形式，后面是一个语句序列和一个 **endif** 结束子句。如下面的语句所示：

```

if( eax = 0 ) then
    stdout.put( "error: NULL value", nl );
endif;

```

在程序执行过程中，如果表达式的值为 **true**，那么执行 **then** 与 **endif** 之间的代码。如果表达式的值为 **false**，那么程序就跳过 **then** 与 **endif** 之间的代码执行。

另一种常见的 if 语句只有一个 **else** 子句。下面是带一个可选的 **else** 子句的 if 语句：

```

if( eax == 0 ) then
    stdout.put( "error: NULL pointer encountered", nl );
else
    stdout.put( "Pointer is valid", nl );
endif;

```

如果表达式的值为 **true**，那么执行 **then** 与 **else** 之间的代码。否则，执行 **else** 与 **endif** 之间的代码。

可以将 **elseif** 子句加入到 if 语句中来创建复杂的决策逻辑。例如，如果寄存器 **CH** 包含一个字符值，就可以使用下面的代码来从菜单项中进行选择：

```

if( ch = 'a' ) then
    stdout.put( "You selected the 'a' menu item", nl );
elseif( ch = 'b' ) then

```



```

    stdout.put( "You selected the 'b' menu item", nl );

elseif( ch = 'c' ) then
    stdout.put( "You selected the 'c' menu item", nl );

else
    stdout.put( "Error: illegal menu item selection", nl );

endif;

```

虽然这个简单的例子并不能表明, HLA 中 `elseif` 子句序列的末尾不必非要有一个 `else` 子句。但是, 当进行多路决策时, 为避免发生错误而加入一个 `else` 子句总是一个好主意。即使认为 `else` 子句不可能执行, 但请记住, 以后对代码进行的修改可能会打破这一断言, 所以代码中最好有错误报告语句。

1.9.3 布尔表达式中的逻辑与、逻辑或以及逻辑非

前面列出的操作符中明显省略掉了逻辑与(逻辑 `and`)、逻辑或(逻辑 `or`)以及逻辑非(逻辑 `not`)操作符。本节将描述它们在布尔表达式中的用法(为了能提供实例, 必须等到描述完 `if` 语句之后再讨论)。

在运行时布尔表达式中, HLA 用操作符 `&&` 来表示逻辑与。这是一个二元(两个操作数)操作符, 并且它的两个操作数都必须是合法的运行时布尔表达式。如果两个操作数的值都为 `true`, 那么该操作得到的值就为 `true`。例如:

```

if( eax > 0 && ch = 'a' ) then
    mov( _eax, ebx );
    mov( ' ', ch );

endif;

```

只有当寄存器 `EAX` 大于零, 且 `CH` 等于字符 `a` 时, 上面的两条 `mov` 语句才会执行。如果其中有一个条件为假, 那么程序就会跳过这些 `mov` 指令执行。

注意, 操作符 `&&` 两边的表达式可以是任意合法的布尔表达式。这些表达式不一定是使用关系运算符的比较操作。例如, 下面所示的都是合法的表达式:

```

@z && al in 5..10
a in 'a'..'z' && ebx
boolVar && !eax

```

在编译操作符 `&&` 时, HLA 使用短路求值法(short-circuit evaluation)。如果最左边操作数的值为 `false`, 那么 HLA 生成的代码就不必对第二个操作数求值(因为此时整个表达式的值都必定为 `false`)。因此, 在上面最后一个表达式中, 如果 `boolVar` 的值为 `false`, 代码就无需检查寄存器 `EAX` 的值是否为 0。

注意, 像 `eax < 10 && cbx < eax` 这样的表达式本身就是一个合法的布尔表达式, 所以可以作为操作符 `&&` 的左操作数或者右操作数。因此, 如下所示的表达式是完全合法的:

```
eax < 0 && ebx <> eax && !ecx
```

操作符&&是向左结合的,所以 HLA 生成的代码采用从左到右的方式对上面的表达式进行求值。如果 EAX 小于 0, CPU 就不用再计算余下的任何一个表达式了。同样,如果 EAX 不小于 0,但是 EBX 等于 EAX,那么该代码就不用对第三个表达式求值,因为无论 ECX 的值是什么,整个表达式的值都为 false。

HLA 用操作符||来表示运行时布尔表达式中的逻辑或。就像操作符&&一样,该操作以两个合法的运行时布尔表达式作为操作数。只要一个(或者两个)操作数的值为 true,该操作符就取值为 true。和操作符&&一样,逻辑或操作符也采取短路求值法。如果左操作数的值为 true,那么 HLA 代码就不必检测第二个操作数的值。相反,代码会转向处理布尔表达式为 true 这种情况的位置。使用||操作符的合法表达式如下所示:

```
@z || al = 10
al in 'a'..'z' || ebx
!boolVar || eax
```

与&&操作符相同,逻辑或操作符也是左结合的,所以在同一个表达式中可以出现||操作符的多个实例。如果出现了这种情况,HLA 产生的代码就会从左到右对表达式求值,例如:

```
eax < 0 || ebx <> eax || !ecx
```

如果 EAX 小于 0、EBX 不等于 EAX,或者 ECX 等于 0,上面代码的求值结果都为真。注意,如果第一个比较的结果为真,代码就不需要再测试其他条件。同样,如果第一个比较的结果为假并且第二个比较的结果为真,代码将不会检查 ECX 是否为 0。只有前两个比较的结果都为假时,才会检查 ECX 是否等于 0。

如果逻辑与操作符和逻辑或操作符出现在同一个表达式中,逻辑与操作符&&的优先级比逻辑或操作符||的优先级要高。考虑如下的表达式:

```
eax < 0 || ebx <> eax && !ecx
```

HLA 产生的机器代码将它等价于:

```
eax < 0 || (ebx <> eax && !ecx)
```

如果 EAX 小于 0,那么 HLA 产生的代码就不必检查剩下的表达式了,整个表达式的值就为 true。但是,如果 EAX 不小于 0,那么为了让整个表达式的值为 true,后面两个条件都必须为真。

对于&&和||来说,如果需要调整操作符的优先级,可以使用圆括弧将子表达式括起来。考虑下面的表达式:

```
(eax < 0 || ebx <> eax) && !ecx
```

要使这个表达式的值为 true,ECX 必须包含 0,并且要么 EAX 小于 0,要么 EBX 一定不能等于 EAX。但如果没有括号,产生的结果就与该表达式的结果不同,试进行比较。

HLA 用!操作符来表示逻辑非。但是,!操作符只能做寄存器或者布尔变量的前缀;不能把它作为更大表达式的一部分(例如,!eax<0)。为了得到一个布尔表达式的逻辑非,必须用括号将该表

达式括起来, 并用 ! 操作符作为括号的前缀, 例如:

```
!( eax < 0 )
```

如果 EAX 不小于 0, 那么该表达式的值为 true。

逻辑非操作符主要用于涉及到逻辑与和逻辑或操作符的复杂表达式。当在一个简短的表达式中使用这种操作符时, 直接声明逻辑关系比使用逻辑非操作符要简单一些(可读性更好)。

注意, HLA 也提供了 | 和 & 操作符, 但它们与 || 和 && 的意思完全不同。要想了解更多关于这些(编译时)操作符的信息, 可以参阅 HLA 参考手册。

1.9.4 while..endwhile 语句

while 语句的基本语法如图 1-11 所示。

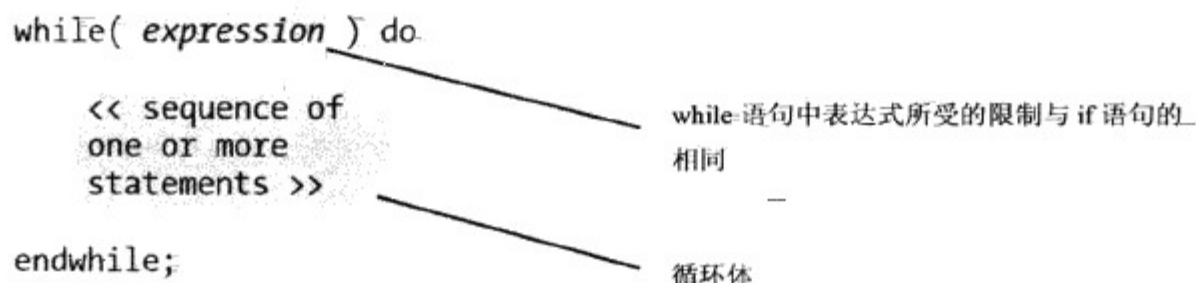


图 1-11 HLA 中 while 语句的语法

该语句对布尔表达式求值。如果为 false, 就立即转向 endwhile 子句之后的第一个语句。如果为 true, 那么 CPU 就执行循环体。循环体执行完之后, 控制权又转回循环起始处, while 语句会重新测试循环控制表达式。该过程重复进行, 直到表达式的值为 false 为止。

注意, 和高级语言中一样, HLA 中的 while 循环是在循环起始处测试循环终止条件的。因此, 循环体中的语句可能不会执行(如果首次执行 while 语句时表达式的值为 false)。还要注意, while 循环体必须在某个时候修改布尔表达式的值, 否则就会导致一个无限循环。

下面是一个 HLA 的 while 循环的例子:

```
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;
```

1.9.5 for..endfor 语句

HLA 的 for 循环通常采用下面的形式:

```
for( Initial Stmt; Termination_Expression; Post_Body_Statement ) do

    << Loop Body >>

endfor;
```

这等效于下面的 while 语句:

```
Initial Stmt;
while( Termination_expression ) do
    << loop_body >>
    Post_Body_Statement;
endwhile;
```

Initial Stmt 可以是任意一条 HLA/80x86 指令。通常情况下, 该语句将寄存器或者存储单元(循环计数器)初始化为 0 或者其他值。*Termination_expression* 是一个 HLA 布尔表达式(与 while 语句的语法相同)。该表达式确定是否执行循环体。*Post-Body-Statement* 在循环末尾执行(正如上面 while 语句的例子中一样)。这是一个单独的 HLA 语句。通常修改循环控制变量的值都是类似于 add 的指令。

下面给出一个完整的例子:

```
for( mov( 0,i ); i < 10; add( 1, i ) ) do
    stdout.put( "i=", i, nl );
endfor;
```

如果将上面的例子改写为 while 循环, 则变成:

```
mov( 0,i );
while( i < 10 ) do
    stdout.put( "i=", i, nl );
    add( 1,i );
endwhile;
```

1.9.6 repeat..until 语句

HLA 的 repeat..until 语句使用如图 1-12 所示的语法。C/C++/C# 和 Java 用户应该注意到 repeat..until 语句和 do..while 语句非常相似。

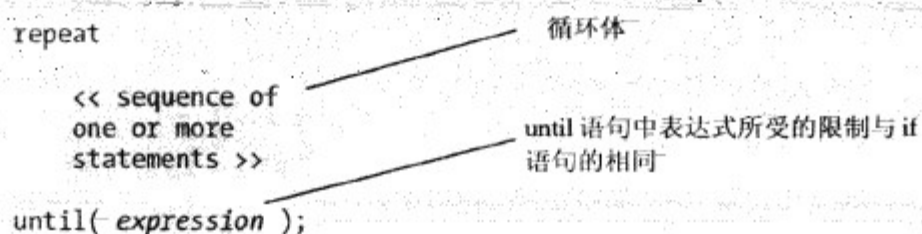


图 1-12 HLA 中 repeat..until 语句的语法

HLA 的 repeat..until 语句在循环末尾处检测循环终止条件。因此, 循环体中的语句至少会执行一次。一旦遇到 until 子句, 程序就会对表达式求值, 如果表达式值为 false, 就重复循环(也就是说, 当值为 false 时就重复)。如果表达式值为 true, 就转向执行 until 子句后面的第一条语句。

下面这个简单的例子演示了 `repeat..until` 语句:

```
mov( 10, ecx );
repeat

    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

如果循环体至少要执行一次,那么使用 `repeat..until` 循环通常会比 `while` 循环更合适。

1.9.7 break 和 breakif 语句

`break` 和 `breakif` 语句的作用是退出循环。图 1-13 给出了这两条语句的语法。

```
break;
breakif( expression );
```

breakif 语句中表达式所受的限制与 if 语句的相同

图 1-13 HLA 中 `break` 和 `breakif` 语句的语法

`break` 语句退出直接包含 `break` 的循环;`breakif` 语句对布尔表达式求值,如果表达式的值为 `true`,就退出直接包含它的循环。

注意, `break` 和 `breakif` 语句不允许跳出多层嵌套的循环。HLA 提供了完成这种功能的语句: `begin..end` 块和 `exit/exitif` 语句。如果想了解更多的细节,请参阅 HLA 参考手册。HLA 还提供了 `continue/continueif` 对,可用于重复执行循环体。同样,详情请参阅 HLA 参考手册。

1.9.8 forever..endfor 语句

图 1-14 给出了 `forever` 语句的语法。

```
forever

    << sequence of
      one or more
      statements >>

endfor;
```

循环体

图 1-14 HLA 中 `forever` 语句的语法

该语句创建了一个无限循环。也可以使用 `break` 和 `breakif` 语句以及 `forever..endfor` 创建一个循环来检测循环终止条件。事实上,这也许是该循环最常见的用法,如下面的例子所示:

```
forever

    stdout.put( "Enter an integer less than 10:" );
    stdin.get( i );
    breakif( i < 10 );
    stdout.put( "The value needs to be less than 10!", nl );

endfor;
```

1.9.9 try..exception..endtry 语句

HLA 的 try..exception..endtry 语句提供了非常强大的异常处理能力。该语句的语法如图 1-15 所示。

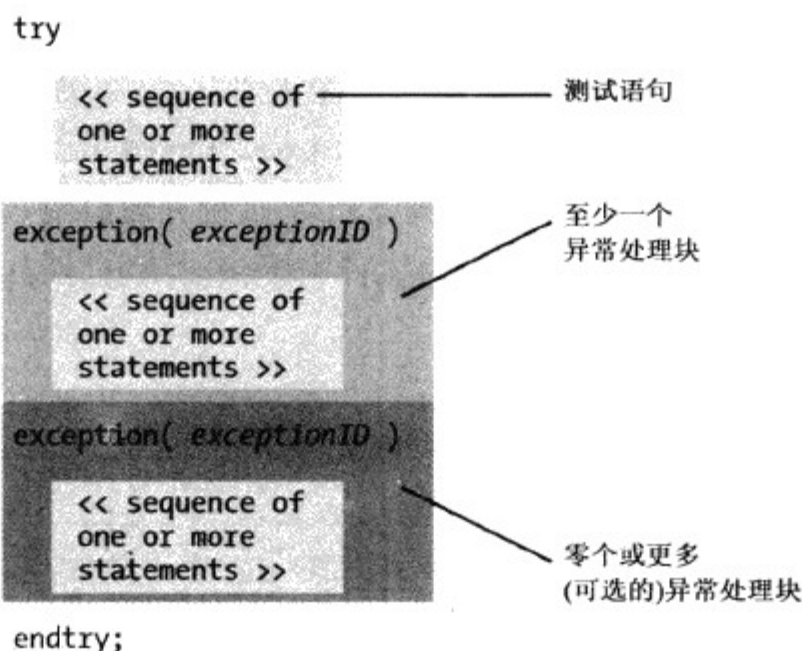


图 1-15 HLA 中 try..exception..endtry 语句的语法

在执行过程中，try..endtry 语句保护一块语句。如果 try 子句和第一个 exception 子句之间的语句(受保护块)在执行过程中没有出现问题，那么在执行完受保护块中的最后一个语句之后，就立即转向 endtry 之后的第一个语句执行。如果发生了一个错误(异常)，那么程序就会在发生异常的点中断控制(也就是说，程序引发一个异常)。每个异常都有一个与它相关联的无符号整型常量，即异常 ID。HLA 标准库中的头文件 excepts.hhf 预定义了几个异常 ID，也可以根据自己的需要创建新的 ID。当有异常发生时，系统就将异常 ID 与受保护代码之后每个异常子句中的值相比较。如果当前的异常 ID 与一个异常值相匹配，那么就继续执行紧跟在该异常之后的语句块。异常处理代码执行完之后，就转向 endtry 之后的第一个语句执行。

如果有异常发生，而且没有激活的 try..endtry 语句，或者激活的 try..endtry 语句不处理指定的异常，程序就会终止并发出一条错误消息。

下面的代码段演示了如何使用 try..endtry 语句来保护程序不接受错误的用户输入：

```

repeat
    mov( false, GoodInteger ); // Note: GoodInteger must be a boolean var.
    try
        stdout.put( "Enter an integer: " );
        stdin.get( i );
        mov( true, GoodInteger );

    exception( ex.ConversionError );
        stdout.put( "Illegal numeric value, please re-enter", nl );

    exception( ex.ValueOutOfRange );
        stdout.put( "Value is out of range, please re-enter", nl );
  
```



```
endtry;  
  
until( GoodInteger );
```

只要输入过程中存在错误，repeat..until 循环就会反复执行该段代码。如果是由于错误的输入而导致异常的发生，控制就转到 exception 子句去看是否存在转换错误(例如，数字当中不合法的字母)或者发生数值溢出。如果这些异常中有一个发生，那么就输出相应的消息，并且跳出 try..endtry 语句。由于 GoodInteger 一直都未被置为 true，repeat..until 循环就重复执行。如果有另外一种异常发生(一个该段代码中没有处理的异常)，那么程序就会输出指定的错误消息并终止。¹⁰

表 1-4 列出了在编写本书时 excepts.hhf 头文件中提供的所有异常。异常的完整列表请参阅 HLA 提供的 excepts.hhf 头文件。

表 1-4 excepts.hhf 中提供的异常

异 常	描 述
ex.StringOverflow	试图将过大的字符串存入一个字符串变量
ex.StringIndexError	试图访问字符串中不存在的某个字符
ex.StringOverlap	试图将字符串复制到自身当中
ex.StringMetaData	损坏的字符串值
ex.StringAlignment	试图在未对齐的地址上存储字符串
ex.StringUnderflow	试图从字符串中提取“负”字符
ex.IllegalStringOperation	不允许在字符串数据上执行此操作
ex.ValueOutOfRange	当前操作的值过大
ex.IllegalChar	操作遇到 ASCII 码不在 0~127 之间的字符代码
ex.TooManyCmdLnParms	命令行包含太多程序参数
ex.BadObjPtr	类对象的指针非法
ex.InvalidAlignment	参数没有在合适的内存地址对齐
ex.InvalidArgument	函数调用(一般是 OS API 调用)包含一个无效的参数值
ex.BufferOverflow	缓冲区或 blob 对象超过了声明的大小
ex.BufferUnderflow	试图从 blob 或者缓冲区获取不存在的数据
ex.IllegalSize	参数的数据大小不正确
ex.ConversionError	字符串到数值的转换含有非法的(非数值)字符
ex.BadFileHandle	程序试图使用非法的文件句柄值访问某个文件
ex.FileNotFound	程序试图访问不存在的文件
ex.FileOpenFailure	操作系统无法打开文件(文件未找到)
ex.FileCloseError	操作系统无法关闭文件

10 经验丰富的程序员可能想知道该代码为什么使用布尔变量而不是 breakif 语句来退出 repeat..until 循环。在 1.11 节中，您会学到其中的一些技术原因。

(续表)

异 常	描 述
ex.FileWriteError	写入文件错误
ex.FileReadError	读取文件错误
ex.FileSeekError	试图查找文件中不存在的位置
ex.DiskFullError	试图向已满的磁盘中写入数据
ex.AccessDenied	用户没有访问文件数据的足够权限
ex.EndOfFile	程序试图读取文件末尾之外的数据
ex.CannotCreateDir	创建目录失败
ex.CannotRemoveDir	删除目录失败
ex.CannotRemoveFile	删除文件失败
ex.CDFailed	改变目录失败
ex.CannotRenameFile	重命名文件失败
ex.MemoryAllocationFailure	系统没有足够的内存满足分配请求
ex.MemoryFreeFailure	不能释放指定的内存块(内存管理系统损坏)
ex.MemoryAllocationCorruption	内存管理系统损坏
ex.AttemptToFreeNULL	调用程序试图释放 NULL 指针
ex.AttemptToDerefNULL	程序试图使用 NULL 指针间接访问数据
ex.BlockAlreadyFree	调用程序试图释放已被释放的内存块
ex.CannotFreeMemory	释放内存操作失败
ex.PointerNotInHeap	调用程序试图释放未在堆上分配的内存块
ex.WidthTooBig	由数值转换为字符串之后的宽度过长
ex.FractionTooBig	由浮点数转换为字符串之后的小数部分过长
ex.ArrayShapeViolation	试图操作两个维度不一致的数组
ex.ArrayBounds	试图访问数组的某个元素, 但下标越界
ex.InvalidDate	试图操作某个非法日期
ex.InvalidDateFormat	由字符串到日期的转换含有非法字符
ex.TimeOverflow	在对时间进行算术运算期间溢出
ex.InvalidTime	试图操作非法时间
ex.InvalidTimeFormat	由字符串到时间的转换含有非法字符
ex.SocketError	网络通信失败
ex.ThreadError	一般性线程(多任务)错误
ex.AssertionFailed	assert 语句遇到一个失败的断言
ex.ExecutedAbstract	试图执行抽象类方法
ex.AccessViolation	试图访问某个非法的存储单元

(续表)

异 常	描 述
ex.InPageError	OS 内存访问错误
ex.NoMemory	OS 内存不足
ex.InvalidHandle	向 OS API 调用传递了无效的句柄
ex.ControlC	在系统控制台中按下了 CTRL-C(功能是 OS 特有的)
ex.Breakpoint	程序执行中断指令(INT3)
ex.SingleStep	在跟踪标志置位情况下运行程序
ex.PrivInstr	程序试图执行内核专用指令
ex.IllegalInstr	程序试图执行某个非法机器指令
ex.BoundsInstr	用“越界”值执行 bound 指令
ex.IntoInstr	在溢出标志置位情况下执行 into 指令
ex.DivideError	程序试图除零或者其他的除法错误
ex.fDenormal	浮点异常(见第 6 章)
ex.fDivByZero	浮点异常(见第 6 章)
ex.flnexactResult	浮点异常(见第 6 章)
ex.fInvalidOperation	浮点异常(见第 6 章)
ex.fOverflow	浮点异常(见第 6 章)
ex.fStackCheck	浮点异常(见第 6 章)
ex.fUnderflow	浮点异常(见第 6 章)
ex.InvalidHandle	操作系统对某个操作报告无效句柄

大多数异常都超出本章讨论的范围，它们在这里出现是出于完整性的考虑。如果需要了解有关这些异常的更多内容，请参阅 HLA 参考手册、HLA 标准库文档以及 HLA 标准库源代码。ex.ConversionError、ex.ValueOutOfRange 和 ex.StringOverflow 是最常用到的几种异常。

在 1.11 节，我们将再次讨论 try..endtry 语句。然而，首先需要介绍一下 HLA 标准库。

1.10 HLA 标准库入门

HLA 之所以比标准的汇编语言好学好用，主要有两个原因。第一个原因是 HLA 在声明和控制结构上的高级语法。这一特征利用了高级语言的知识，从而有助于更有效地学习汇编语言。另一个原因便是 HLA 标准库。HLA 标准库提供了很多常见的且易于使用的汇编语言例程。您可以调用这些例程，而不必自己编写代码(甚至不需要学习如何编写)。这就消除了很多人在学习汇编语言时都遇到的障碍：为了编写基本语句，需要编写复杂的 I/O 代码及其相关的支持代码。在标准化的汇编语言库出现以前，即使显示一个字符串，初学者通常也要花上很长的时间才能学会。有了 HLA 标准库，这个障碍终于消除了，这样就可以集中精力学习汇编语言的概念，而不用学习

一些专门针对某个操作系统的低级 I/O 细节。

大量的库例程只是 HLA 所支持的一部分。毕竟,汇编语言库已经存在很长一段时间了。¹¹HLA 的标准库为这些例程提供了高级语言接口,从而使 HLA 更加完整。事实上,HLA 语言原本就是专门为允许高级库例程集的创建而设计的。与库中许多例程的高级特性相结合,该高级接口将许多功能压缩到一个易用的包中。

HLA 标准库由按类别组织的若干模块构成。表 1-5 列出了很多可利用的模块:¹²

表 1-5 HLA 标准库模块

名 称	描 述
args	命令行参数解析的支持例程
arrays	数组声明和操作
bits	位操作函数
blobs	二进制大对象——对大块二进制数据的操作
bsd	FreeBSD 的 OS API 调用(只用于 FreeBSD 版本的 HLA)
chars	字符数据上的操作
console	可移植的控制台(文本屏幕)操作(光标移动、清屏等)
conv	字符串和其他值之间的各种转换
cset	字符集函数
DataTime	日历、日期和时间函数
env	访问 OS 环境变量
excepts	异常处理例程
fileclass	面向对象文件的输入和输出
fileio	文件输入输出例程
filesys	访问 OS 文件系统
hla	特殊的 HLA 常量和值
Linux	Linux 系统调用(只用于 Linux 版本的 HLA)
lists	用于操作链表的 HLA 类
mac	Mac OS X 的 OS API
math	扩展精度的算术函数、超越函数和其他数学函数
memmap	内存映射文件操作
memory	内存分配、回收及其支持代码
patterns	HLA 的模式匹配库
random	伪随机数生成器及其支持代码
sockets	一组网络通信函数和类

¹¹ 例如, 80x86 汇编语言程序员专用的 UCR 标准库。

¹² 因为 HLA 标准库在不断扩充, 该表可能已经过期。要了解最新的标准库模块列表, 请参阅 HLA 文档。

(续表)

名 称	描 述
stderr	为用户提供输出和其他几个支持函数
stdin	用户输入例程
stdio	stderr、stdin 和 stdout 的支持模块
stdout	为用户提供输出和其他几个支持例程
strings	HLA 强大的字符串库
tables	表(关联数组)支持例程
threads	支持多线程应用程序和进程同步
timers	支持应用程序中的定时事件
win32	Windows 调用中使用的常量(只用于 Windows 版本的 HLA)
x86	专用于 80x86 CPU 的常量和和其他项

本书稍后将更详细地对这些模块进行解释。本节主要介绍最重要的例程(至少对 HLA 初学者来说)——stdio 库。

1.10.1 stdio 模块中的预定义常量

我们首先应该从 stdio 模块所定义的一些普通常量开始。考虑下面的例子：

```
stdout.put( "Hello World", nl);
```

该语句末尾的 nl 表示换行(newline)。nl 标识符既不是专门的 HLA 保留字，也不专用于 stdout.put 语句。相反，它只是一个简单的预定义常量，对应于包含标准行尾序列的字符串(这就像 Windows 下的回车/换行符对或者 Linux、FreeBSD 和 Mac OS X 下的换行符一样)。

除了 nl 常量以外，HLA 标准 I/O 库模块还定义了一些其他一些字符常量，如表 1-6 所示。

表 1-6 HLA 标准 I/O 库定义的字符常量

字 符	定 义
stdio.bell	ASCII 响铃符，输出时发出嘟嘟声
stdio.bs	ASCII 退格符
stdio.tab	ASCII 制表符
stdio.lf	ASCII 换行符
stdio.cr	ASCII 回车符

除了 nl 以外，这些字符都出现在 stdio 命名空间¹³中(因此需要前缀 stdio.)。在 stdio 命名空间中加入这些 ASCII 常量可以避免与自定义的变量产生命名冲突。nl 不会出现在命名空间中，因为会经常用到它，而输入 stdio.nl 很快就会让人感到厌烦。

13 命名空间将在第 5 章中介绍。

1.10.2 标准输入和标准输出

很多 HLA I/O 例程都有 `stdin` 或者 `stdout` 前缀。从技术上来说,这就意味着标准库在命名空间中定义了这些名称。实际上,该前缀表明从哪里(标准输入设备)输入或者输出到哪里(标准输出设备)。默认情况下,标准输入设备是系统键盘。同样,默认的标准输出设备是控制台显示器。所以,一般来说,带有 `stdin` 或者 `stdout` 前缀的语句会在控制台设备上读写数据。

当从命令行窗口(或者 `shell`)运行程序时,可以选择重定向(`redirecting`)标准输入和标准输出设备。`>outfile` 形式的命令行参数将标准输出设备重定向到指定的文件(`outfile`)。`<infile` 形式的命令行参数重定向标准输入,使得它的数据来自指定的输入文件(`infile`)。下面的例子演示了在命令行窗口中运行一个名为 `testpgm` 的程序时,如何使用这些参数¹⁴。

```
testpgm <input.data
testpgm >output.txt
testpgm <in.txt >output.txt
```

1.10.3 `stdout.newln` 例程

`stdout.newln` 例程向标准输出设备输出一个换行序列。这在功能上与 `stdout.put(nl);` 等效。有时调用 `stdout.newln` 会更方便一些。下面给出了一个例子:

```
stdout.newln();
```

1.10.4 `stdout.putiX` 例程

`stdout.puti8`、`stdout.puti16` 和 `stdout.puti32` 库例程将一个参数(分别为一个字节、两个字节或者四个字节)作为带符号整数值输出。该参数可以是一个常量、寄存器或者内存变量,只要实参的长度与形参的长度相同就行。

这些例程将指定参数的值输出到标准输出设备。它们将用尽可能少的打印位置来输出值。如果该值为负,就输出一个前导的减号。这里有一些调用这些例程的例子:

```
stdout.puti8( 123 );
stdout.puti16( dx );
stdout.puti32( i32Var );
```

1.10.5 `stdout.putiXSize` 例程

正如 `stdout.putiX` 例程一样, `stdout.puti8Size`、`stdout.puti16Size` 和 `stdout.puti32Size` 例程将带符号整数值输出到标准输出设备。但是,这些例程对输出提供了更多的控制;它们允许为值指定所要求的(最小的)打印位置数。这些例程还允许指定填充字符(如果打印域大于显示值所需的字符位置数)。这些例程需要下面的参数:

¹⁴ Linux、FreeBSD 和 Mac OS X 用户请注意:根据系统的设置,可能需要在程序名前输入“`/`”才能真正地执行程序,例如,“`./testpgm<input.data`”。

```

stdout.puti8Size( Value8, width, padchar );
stdout.puti16Size( Value16, width, padchar );
stdout.puti32Size( Value32, width, padchar );

```

参数 *Value** 可以是一个常量、寄存器或者指定大小的存储单元。参数 *width* 可以是 -256~+256 之间任意的带符号整型常量，该参数可以是常量、寄存器(32 位)或者存储单元(32 位)。参数 *padchar* 应该是单独一个字符值。

就像 `stdout.putiX` 例程一样，这些例程将指定的值当作带符号整型常量输出到标准输出设备。但是，这些例程要让您为该值指定域宽。所谓域宽，就是在打印该值时，这些例程将要使用的打印位置的最小数目。参数 *width* 定义了最小域宽。如果该数字需要更多的打印位置(例如，如果想在域宽为 2 的空间打印“1234”)，那么这些例程就会打印正确显示其值所需的全部字符。另一方面，如果参数 *width* 大于显示值所需的字符位置数，这些例程就会打印一些填充字符来确保输出结果的宽度至少等于字符位置数。如果 *width* 值为负，那么该数字在打印域中就是左对齐的；如果 *width* 值为正，那么该数字在打印域中就是右对齐的。

如果参数 *width* 的绝对值大于打印位置数的最小值，那么这些 `stdout.putiXSize` 例程就会在该数字之前或者之后打印填充字符。参数 *padchar* 指定这些例程打印哪个字符。大多数情况下，将空格指定为填充字符；在特殊情况下，可能指定其他字符。记住，参数 *padchar* 是一个字符值；在 HLA 中，字符常量通常用单引号括起来，而不是双引号。也可以为该参数指定一个 8 位的寄存器。

程序清单 1-4 提供了一个简短的 HLA 程序，演示了如何使用 `stdout.puti32Size` 例程来以表列形式显示一系列值。

程序清单 1-4 使用 `stdio.puti32Size` 以表列形式输出值

```

program NumsInColumns;

#include( "stdlib.hhf" )

var
    i32:    int32;
    ColCnt:int8;

begin NumsInColumns;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.puti32Size( i32, 5, ' ' );
        sub( 1, i32 );
        add( 1, ColCnt );
    endwhile;
end

```

```

    endwhile;
    stdout.newln();
end NumsInColumns;

```

1.10.6 stdout.put 例程

`stdout.put` 例程¹⁵是标准输出库模块中最灵活的例程之一。它将大部分的其他输出例程合并为一个易用的过程。

`stdout.put` 例程的一般形式如下所示：

```

stdout.put( list_of_values_to_output );

```

`stdout.put` 的参数列表由一个或者多个常量、寄存器或者内存变量组成，每一个之间都用逗号隔开。该例程显示列表中每个参数的值。由于在本章中我们都在使用这一例程，所以您已经看到过很多该例程的基本形式的例子。值得指出的是，该例程还有几个额外的特征在本章的例子中并没有体现出来。尤其是，每个参数都可以采取下面两种形式中的任意一种：

```

value
value:width

```

value 可以是任意合法的常量、寄存器或者内存变量对象。在本章中，您已经看到了出现在 `stdout.put` 参数列表中的字符串常量和内存变量。这些参数与上面的第一种形式相对应。与 `stdout.putiXSize` 例程相似，第二种参数形式允许指定一个最小域宽¹⁶。程序清单 1-5 中程序的输出结果与程序清单 1-4 的相同；但是，程序清单 1-5 使用的是 `stdout.put` 而不是 `stdout.puti32Size`。

程序清单 1-5 `stdout.put` 指定域宽的示例

```

program NumsInColumns2;

#include( "stdlib.hhf" )

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns2;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do
        if( ColCnt = 8 ) then
            stdout.newln();

```

¹⁵ `stdout.put` 实际上是一个宏，而不是一个过程。两者之间的区别超出了本章的范围，但是本书将在第 9 章对它们进行描述。

¹⁶ 注意，当使用 `stdout.put` 例程时，不能指定填充字符；填充字符默认为空格字符。如果需要其他字符作为填充字符，可用 `stdout.putiXSize` 例程。


```

        mov( 0, ColCnt );

endif;
stdout.put( i32:5 );
sub( 1, i32 );
add( 1, ColCnt );

endwhile;
stdout.put( nl );

end NumsInColumns2;

```

`stdout.put` 例程的功能比本节所描述的要多得多。本书将在适当的时候介绍一些额外的功能。

1.10.7 `stdin.getc` 例程

`stdin.getc` 例程从标准输入设备的输入缓冲区¹⁷读取下一个字符。它将该字符返回给 CPU 的 AL 寄存器。程序清单 1-6 演示了该例程的一个简单应用。

程序清单 1-6 `stdin.getc()` 例程示例

```

program charInput;

#include( "stdlib.hhf" );

var
    counter: int32;

begin charInput;

    // The following repeats as long as the user
    // confirms the repetition.
    repeat
        // Print out 14 values.

        mov( 14, counter );
        while( counter > 0 ) do
            stdout.put( counter:3 );
            sub( 1, counter );
        endwhile;

        // Wait until the user enters 'y' or 'n'.
        stdout.put( nl, nl, "Do you wish to see it again? (y/n):" );
        forever
            stdin.readLn();
            stdin.getc();
            breakif( al = 'n' );
        forever
    repeat

```

¹⁷“缓冲区”是描述数组的一个术语。

```

        breakif( al = 'y' );
        stdout.put( "Error, please enter only 'y' or 'n': " );

    endfor;
    stdout.newln();

until( al = 'n' );

end charInput;

```

该程序使用 `stdin.ReadLn` 例程来强制用户的输入换行。1.10.9 节将对 `stdin.ReadLn` 进行描述。

1.10.8 `stdin.getiX` 例程

`stdin.geti8`、`stdin.geti16` 和 `stdin.geti32` 例程分别从标准输入设备读取 8 位、16 位和 32 位的带符号整数值，并将它们分别返回给 AL、AX 以及 EAX 寄存器。它们为 HLA 提供了从用户那里读取带符号整数值的标准机制。

就像 `stdin.getc` 例程一样，这些例程从标准输入缓冲区读取一个字符序列。它们先跳过所有的空白字符(空格、制表符等)，然后将它后面的十进制数字流(带可选的前导负号)转换为相应的整数。如果输入序列不是有效的整数字符串，或者如果用户输入太长，大于指定的整数长度，那么这些例程就会引发异常。注意 `stdin.geti8` 例程读入的值必须在 -128~+127 之间；`stdin.geti16` 例程读入的值必须在 -32 768~+32 767 之间；`stdin.geti32` 例程读入的值必须在 -2 147 483 648~+2 147 483 647 之间。

程序清单 1-7 中的示例程序展示了这些例程的使用。

程序清单 1-7 `stdin.getiX` 示例代码

```

program intInput;

#include( "stdlib.hhf" )

var
    i8:   int8;
    i16:  int16;
    i32:  int32;

begin intInput;
    // Read integers of varying sizes from the user:

    stdout.put( "Enter a small integer between -128 and +127: " );
    stdin.geti8();
    mov( al, i8 );

    stdout.put( "Enter a small integer between -32768 and +32767: " );
    stdin.geti16();
    mov( ax, i16 );

    stdout.put( "Enter an integer between +/- 2 billion: " );
    stdin.geti32();
    mov( eax, i32 );

```



```

    // Display the input values.

    stdout.put
    (
        nl,
        "Here are the numbers you entered:", nl, nl,
        "Eight-bit integer: ", i8:12, nl,
        "16-bit integer:     ", i16:12, nl,
        "32-bit integer:      ", i32:12, nl
    );

end intInput;

```

您应该编译和运行该程序，并检测当输入一个超出指定范围的值或者输入了一个不合法的字符串时会发生什么。

1.10.9 stdin.readLn 和 stdin.flushInput 例程

调用像 `stdin.getc` 或 `stdin.geti32` 这样的输入例程时，程序不必从用户那里读取数据，而是 HLA 标准库会从用户那里读取一整行文本，并将输入内容存入缓冲区。对输入例程的调用会从该输入缓冲区读入数据，直到缓冲区变空为止。虽然这种缓冲模式很有效而且比较方便，但是有时会产生混淆。考虑下面的代码序列：

```

stdout.put( "Enter a small integer between -128 and +127: " );
stdin.geti8();
mov( al, i8 );

stdout.put( "Enter a small integer between -32768 and +32767: " );
stdin.geti16();
mov( ax, i16 );

```

从直觉上来说，您会认为程序先输出第一条提示消息，等待用户输入，然后输出第二条提示消息，再等待第二次用户输入。然而并非如此。例如，如果运行该代码(取自前一节的示例程序)，并输入文本“123 456”以响应第一条提示消息，在第二条提示处，程序不会停下来等待其他用户输入。而是当调用 `stdin.geti16` 时，它会从输入缓冲区中读入第二个整数(456)。

一般情况下，只有当输入缓冲区为空时，`stdin` 例程才会从用户处读取文本。只要输入缓冲区包含了额外的字符，输入例程就会试图从缓冲区中读取它们的数据。可以编写如下所示的代码序列来利用这一行为：

```

stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intValue );
stdin.geti32();
mov( eax, AnotherIntVal );

```

该序列允许用户在同一行输入两个值(用一个或者多个空格隔开)，从而保留屏幕上的空格。

所以，有时候输入缓冲区行为还是令人满意的。但有些时候，输入例程的缓冲行为却和直觉相反。

幸运的是，HLA 标准库提供了 `stdin.readLine` 和 `stdin.flushInput` 这两个例程，所以可以控制标准输入缓冲区。`stdin.readLine` 例程放弃输入缓冲区中的所有数据，直接要求用户输入一行新的文本。而 `stdin.flushInput` 例程只是放弃输入缓冲区中的所有数据。下一次执行输入例程时，系统就需要用户输入新的一行。通常情况下，在某个标准输入例程之前调用 `stdin.readLine`，而在一个标准输入例程调用之后调用 `stdin.flushInput`。

注意：

如果调用 `stdin.readLine` 时发现需要将数据输入两遍，这就说明此时应该调用的是 `stdin.flushInput` 而不是 `stdin.readLine`。一般情况下，应该调用 `stdin.flushInput` 来刷新输入缓冲区，并在下一次输入调用时读入新的一行数据。`stdin.readLine` 例程几乎没有必要，所以应该用 `stdin.flushInput`，除非真的需要马上输入一行新文本。

1.10.10 `stdin.get` 例程

`stdin.get` 例程将许多标准输入例程组合成一个调用，就像 `stdout.put` 将所有的输出例程组合成一个调用一样。事实上，`stdin.get` 比 `stdout.put` 更容易使用，因为该例程唯一的参数是一列变量名称。

我们重写前一节给出的例子：

```
stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

使用 `stdin.get` 例程，我们可以将这段代码重写为：

```
stdout.put( "Enter two integer values: " );
stdin.get( intval, AnotherIntVal );
```

正如您所看到的，`stdin.get` 例程更易于使用。

注意，`stdin.get` 将输入数据直接存储到参数列表指定的内存变量中；它并不将值返回给寄存器，除非指定了一个寄存器作为参数。`stdin.get` 的参数必须都是变量或者寄存器。

1.11 关于 `try..endtry` 的其他细节

为了捕获在语句执行过程中发生的所有异常，通常使用 `try..endtry` 语句包围这块语句。系统通过 3 种方式引发异常：硬件故障(例如除零错误)产生的异常、由操作系统产生的异常，以及执行 HLA 的 `raise` 语句产生的异常。可以使用 `exception` 子句编写一个异常处理程序来截取某个异常。程序清单 1-8 中的程序给出了 `try..endtry` 语句的典型例子。

程序清单 1-8 try..endtry 示例

```

program testBadInput;
#include( "stdlib.hhf" )

static
    u:      int32;

begin testBadInput;

    try
        stdout.put( "Enter a signed integer:" );
        stdin.get( u );
        stdout.put( "You entered: ", u, nl );

        exception( ex.ConversionError )

        stdout.put( "Your input contained illegal characters" nl );

        exception( ex.ValueOutOfRange )

        stdout.put( "The value was too large" nl );

    endtry;

end testBadInput;

```

HLA 将 try 子句和第一个 exception 子句之间的语句作为受保护的语句。如果受保护的语句中发生一个异常,那么程序就会扫描每个异常,并将当前异常的值与每个 exception 子句之后圆括号中的值相比较。¹⁸该异常值仅是一个 32 位的值。因此,每个 exception 子句后圆括号中的值必须是一个 32 位的值。HLA 的 excepts.hhf 头文件预定义了几个异常常量。可以用数字值来取代以上两个 exception 子句,但这是一种不好的编程风格。

1.11.1 try..endtry 嵌套语句

如果程序扫描 try..endtry 语句中的所有 exception 子句,但没有找到与当前的异常值匹配的值,那么为了找到一个合适的异常处理程序,程序会搜索动态嵌套的 try..endtry 块中的 exception 子句。例如,考虑程序清单 1-9 中的代码。

程序清单 1-9 try..endtry 嵌套语句

```

program testBadInput2;
#include( "stdlib.hhf" )

static

```

¹⁸ 注意, HLA 将该值存入 EAX 寄存器。所以一旦进入 exception 子句, EAX 中就会存储该异常号。


```

    u:    int32;

begin testBadInput2;
    try
        try
            stdout.put( "Enter a signed integer: " );
            stdin.get( u );
            stdout.put( "You entered: ", u, nl );

            exception( ex.ConversionError )
                stdout.put( "Your input contained illegal characters" nl );
        endtry;

        stdout.put( "Input did not fail due to a value out of range" nl );
        exception( ex.ValueOutOfRange )
            stdout.put( "The value was too large" nl );
        endtry;
    end testBadInput2;

```

在程序清单 1-9 中，一个 try 语句嵌套在另一个 try 语句中。在执行 stdin.get 语句时，如果用户输入一个大于 40 亿的数，那么 stdin.get 将产生 ex.ValueOutOfRange 异常。当 HLA 运行库系统接收该异常时，它会首先搜索直接包围产生异常的语句的 try..endtry 语句(在上面的例子中，就是嵌套的 try..endtry 语句)中的所有 exception 子句。如果 HLA 运行库系统不能为 ex.ValueOutOfRange 异常找到一个处理程序，那么它就会检查当前的 try..endtry 语句是否嵌套在另一个 try..endtry 语句中(就像程序清单 1-9 中的例子一样)。如果是这样，HLA 运行库系统就会在外层的 try..endtry 语句中搜索一个适当的 exception 子句。由于在程序清单 1-9 中的 try..endtry 块内找到了一个适当的异常处理程序，所以控制权转移到 exception(ex.ValueOutOfRange)子句后面的语句。

在执行完 try..endtry 块之后，HLA 运行库系统就不再认为该块是活动的，当程序发生异常时，就不会搜索其异常列表了。¹⁹这样就允许在程序的其他部分对同样的异常做不同的处理。

如果两个 try..endtry 语句处理相同的异常，其中一个 try..endtry 块嵌套在另一个 try..endtry 语句受保护的部分中，并且当执行最内层的 try..endtry 序列时，程序发生了异常，那么 HLA 就直接将控制权转移到最内层 try..endtry 块所提供的异常处理程序处执行。HLA 不会自动将控制权转移到外层 try..endtry 序列所提供的异常处理程序中。

在前面的例子(程序清单 1-9)中，第二个 try..endtry 语句静态嵌套在外层 try..endtry 语句中²⁰。如前所述，如果最迟激活的 try..endtry 语句不能处理某个异常，那么程序就会搜索所有动态嵌套

¹⁹ 当然，除非程序通过循环或者其他控制结构重新进入 try..endtry 块。

²⁰ “静态嵌套”的意思是在源代码中，一个语句嵌套在另一个语句之内。当我们说一个语句嵌套在另一个语句内时，通常是指该语句静态嵌套于另一个语句内。

`try..endtry` 块中的 `exception` 子句。动态嵌套不要求嵌套的 `try..endtry` 块必须在外层 `try..endtry` 语句中。而是控制权能够从外层 `try..endtry` 受保护块内部转移到程序中的其他地方。在这些地方的 `try..endtry` 语句动态嵌套两个 `try` 语句。虽然有很多方法可以动态嵌套代码，但是有一种方法您在高级语言中可能就已经熟悉了：过程调用。在第5章，当您学会了如何用汇编语言编写过程(函数)以后，就应该记住，如果程序在过程中执行 `try..endtry`，那么在 `try..endtry` 块受保护的部分中对该过程的任意调用都能够创建一个动态嵌套的 `try..endtry`。

1.11.2 `try..endtry` 语句中不受保护的子句

只要程序执行了 `try` 子句，一旦有异常发生，程序就会为当前的异常保存现场，并设置系统以使控制权转移到 `try..endtry` 语句中的异常子句处执行。如果程序成功地执行了 `try..endtry` 受保护的部分，那么程序就会恢复原来的现场，控制权转移到 `endtry` 子句后面的第一个语句处执行。最后一步对于恢复执行现场是非常重要的。如果程序跳过了这一步，那么即使程序已经离开了 `try..endtry` 块，以后所有的异常还是会将控制权转到该 `try..endtry` 语句。程序清单 1-10 演示了该情况：

程序清单 1-10 错误退出 `try..endtry` 语句

```
program testBadInput3;
#include( "stdlib.hhf" )

static
    input: int32;

begin testBadInput3;

    // This forever loop repeats until the user enters
    // a good integer and the break statement below
    // exits the loop.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );
            break;

        exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large, re-enter." nl );

        exception( ex.ConversionError )

            stdout.put( "The input contained illegal characters, re-enter." nl );

    endtry;

endfor;

// Note that the following code is outside the loop and there
// is no try..endtry statement protecting this code.
```

```

        stdout.put( "Enter another number: " );
        stdin.get( input );
        stdout.put( "The new number is: ", input, nl );

end testBadInput3;

```

本例试图通过将 `try..endtry` 语句放入一个循环，当 `stdin.get` 例程发生异常(因为错误的用户输入)时强制用户重新输入数据来创建一个健壮的程序。虽然这是一个好主意，但是在实现时存在一个很大的问题：`break` 语句会立即退出 `forever..endfor` 循环，而不是先恢复异常现场。因此，当程序在其末尾执行第二个 `stdin.get` 语句时，HLA 异常处理代码就认为还在 `try..endtry` 块中。如果发生了一个异常，HLA 就将控制权转回到 `try..endtry` 语句来寻找适当的异常处理程序。假设该异常为 `ex.ValueOutOfRange` 或者 `ex.ConversionError`，程序清单 1-10 中的程序将会输出适当的错误消息，然后强制用户重新输入第一个值。这并非我们想得到的。

将控制权转到错误的 `try..endtry` 异常处理程序只是一个小问题。程序清单 1-10 中另一个必须处理的大问题是 HLA 如何保留和恢复现场：HLA 将旧的执行现场信息保存在存储器中一个被称为栈(stack)的专门区域。如果没有恢复异常现场就退出了 `try..endtry`，就会把旧的执行现场信息遗留在栈中。这些额外的数据会使程序出错。

虽然这里的讨论已经非常清晰地说明了程序不能像程序清单 1-10 那样退出 `try..endtry` 语句，但是如果程序出现了这种情况时，就在 `try..endtry` 语句外通过循环来强制用户重新输入数据，这样也是不错的。为了做到这一点，HLA 的 `try..endtry` 语句提供了一个不受保护的部分。考虑下面程序清单 1-11 所示的代码：

程序清单 1-11 `try..endtry` 不受保护的部分

```

program testBadInput4;
#include( "stdlib.hhf" )

static
    input: int32;

begin testBadInput4;

    // This forever loop repeats until the user enters
    // a good integer and the break statement below
    // exits the loop. Note that the break statement
    // appears in an unprotected section of the try..endtry
    // statement.

    forever

        try
            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );
        unprotected

            break;

    exception( ex.ValueOutOfRange )

```



```

        stdout.put( "The value was too large, re-enter." nl );
    exception( ex.ConversionError )
        stdout.put( "The input contained illegal characters, re-enter." nl );
    endtry;
endfor;

// Note that the following code is outside the loop and there
// is no try..endtry statement protecting this code.

stdout.put( "Enter another number: " );
stdin.get( input );
stdout.put( "The new number is: ", input, nl );
end testBadInput4;

```

只要 `try..endtry` 语句命中了 `unprotected` 子句, 它就会立即恢复异常现场。顾名思义, 不受保护部分的语句的执行不再受到 `try..endtry` 块的保护(但是要注意所有的动态嵌套语句还是活动的; “不受保护”只是关闭了包含 `unprotected` 子句的 `try..endtry` 块的异常处理功能)。因为程序清单 1-11 中的 `break` 语句出现在 `unprotected` 部分, 所以它可以不“执行”`endtry` 子句就能安全地将控制权转出 `try..endtry` 块, 这是因为程序已经恢复了前一次异常的现场。

注意, `unprotected` 关键字必须在受保护块之后的第一个 `try..endtry` 语句中出现。也就是说, 它必须出现在所有的 `exception` 关键字之前。

如果在 `try..endtry` 序列执行期间发生了一个异常, HLA 会自动恢复执行现场。因此, 可以在 `exception` 子句中执行一条 `break` 语句(或者其他可以将控制权转出 `try..endtry` 块的指令)。

由于程序一遇到 `unprotected` 块或者 `exception` 块就会恢复异常现场, 所以在这些区域中发生的异常会立即将控制权转到前一个(动态嵌套)活动的 `try..endtry` 序列。如果没有嵌套的 `try..endtry` 序列, 程序就会终止并输出相应的错误消息。

1.11.3 `try..endtry` 语句中的 `anyexception` 子句

在典型情况下, 会将 `try..endtry` 语句与一组 `exception` 子句一起使用, 这组 `exception` 子句将处理 `try..endtry` 语句受保护部分可能发生的所有异常。通常, 应该确保 `try..endtry` 语句能够处理所有可能的异常, 以防程序由于一个未处理的异常而提前终止, 这是非常重要的。如果编写了受保护部分的所有代码, 就会知道它将引发的异常, 这样就能够处理所有的异常了。但是, 如果正在调用一个库例程(尤其是第三方的库例程)、进行一次 OS API 调用, 或者执行一段没有控制权限的代码, 那么要想预测代码中可能引发的所有异常(尤其是在考虑代码过去的、现在的和未来的版本时)也许是不可能的。如果代码产生了一个没有对应 `exception` 子句的异常, 则会使得程序失败。幸运的是, HLA 的 `try..endtry` 语句提供了 `anyexception` 子句, 它会自动捕获当前 `exception` 子句不能处理的所有异常。

除了不需要异常号参数以外, `anyexception` 子句与 `exception` 子句相似(因为它可以处理任何异常)。如果 `anyexception` 子句与其他 `exception` 子句一起出现在 `try..endtry` 语句中, 那么 `anyexception` 部分必须是 `try..endtry` 语句中的最后一个异常处理程序。`anyexception` 部分可以是 `try..endtry` 语句

中唯一的异常处理程序。

如果一个未处理的异常将控制权转到了 `anyexception` 部分,那么寄存器 `EAX` 就会存储异常号。`anyexception` 块中的代码可以通过检查该数值来确定引发异常的原因。

1.11.4 寄存器与 `try..endtry` 语句

当进入 `try..endtry` 语句时, `try..endtry` 语句保留几个字节的数据。离开 `try..endtry` 块(或者命中 `unprotected` 子句),程序就会恢复异常现场。只要没有异常发生,当进入或者退出 `try..endtry` 语句时, `try..endtry` 语句就不会影响任何寄存器的值。但是,如果异常发生在受保护语句的执行过程中,那么情况就不是这样了。

一旦进入 `exception` 子句, `EAX` 寄存器就会存储异常号,但是其他所有通用寄存器的值都还没有定义。由于操作系统可能已经引发了一个与硬件错误相关的异常(并且已经因此而破坏了寄存器的内容),所以不能假设通用寄存器会包含异常发生时它们包含的值。HLA 为异常生成的代码会根据编译器版本的不同而不同,当然它还会因操作系统而异。因此如果想通过试验来判断在某个异常处理程序中寄存器所存储的值,并且在代码中还要使用这些值,那么这并不是个好主意。

由于进入异常处理程序会破坏寄存器的值,所以如果紧跟在 `endtry` 子句之后的代码假设寄存器存储了某个值(也就是说,在受保护部分设定的值或者执行 `try..endtry` 语句之前设定的值),那么就必须确保重新载入重要的寄存器。如果不能做到这一点,就会在程序中引入一些具有威胁性的缺陷(由于异常通常都很少发生,并且也不总是会破坏某个寄存器的值,所以这些缺陷会时有时无,并且很难检测到)。下面的代码段提供了该程序的一个典型示例及其解决办法:

```
static
    sum: int32;

    .data

    mov( 0, sum );
    for( mov( 0, ebx ); ebx < 8; inc( ebx ) ) do

        push( ebx ); // Must preserve ebx in case there is an exception.
        forever
            try

                stdin.geti32();
                unprotected break;

            exception( _ex.ConversionError )

                stdout.put( "Illegal input, please re-enter value: " );

            endtry;
        endfor;
        pop( ebx ); // Restore ebx's value.
        add( -ebx, eax );
        add( eax, sum );

    endfor;
```


由于 HLA 的异常处理机制搞乱了寄存器,而且异常处理是一个效率相对较低的过程,所以千万不要把 `try..endtry` 语句当作一般的控制结构使用(例如,通过生成一个整型的异常值,并把 `exception` 子句作为要处理的情况,用它来模拟 `switch/case` 语句)。这样做会对程序的性能产生很大的负面影响,并且由于异常处理程序会扰乱寄存器,所以还有可能会引入一些难以察觉的缺陷。

对于正确的操作来说, `try..endtry` 语句假设您只用寄存器 `EBP` 来指向活动记录(本书将在第 5 章讨论活动记录)。默认情况下, HLA 程序会自动使用 `EBP`; 只要不修改 `EBP` 中的值,程序就会自动使用 `EBP` 来保存一个指向当前活动记录的指针。如果试图将 `EBP` 寄存器用作通用寄存器来保存值,并计算算术结果,那么 HLA 的异常处理功能就不再正常(还可能存在一些其他的问题)。因此,千万不要把寄存器 `EBP` 当作通用寄存器来使用。当然,这里的讨论也适用于寄存器 `ESP`。

1.12 高级汇编语言与低级汇编语言的比较

在结束本章以前,有必要提醒您,在本章出现的控制语句中没有一个是“真正的”汇编语言。80x86 CPU 不支持像 `if`、`while`、`repeat`、`for`、`break`、`breakif` 和 `try` 这样的机器指令。HLA 只要一遇到这些语句,就把它们编译成一个或多个真正的机器指令,这种机器指令就像您所用过的高级语句一样完成操作。尽管这些语句用起来很方便,在很多情况下就像 HLA 将它们翻译成低级机器指令序列一样有效,但不要忽略了一个事实,它们并不是真正的机器指令。

本书的目的就是教您进行低级汇编语言的编程;讨论这些高级控制语句只是达到这一目的的途径。请记住,学会 HLA 的高级汇编控制结构可以让您将高级语言的知识早早地应用于学习过程中,这样就不必将关于汇编语言的知识立刻都学会。通过使用高级语言的控制结构,本书就可以推迟介绍在控制流当中经常使用的真正的机器指令。这样一来,就会发现学习汇编语言变得轻松多了。然而,必须记住,这些高级控制语句只是帮助您学会汇编语言的一种教学工具。虽然在掌握了真正的控制流语句后还可以在汇编程序中自由使用它们,但是如果要学会汇编语言的编程,就必须真正地学会低级控制语句。这就是学习本书的原因所在,所以千万不要让高级控制语句成为拐杖。当您真正学会了如何编写低级控制语句时,就要经常使用它们。当具备了使用低级控制语句的经验并了解了它们的优点和缺点时,您就可以很好地判断对于一个给定的应用来说,是高级代码还是低级代码更适合。但是,只有获得了足够多的关于低级控制结构的经验时,才能作出明智的决定。记住,只有在掌握了低级语句之后,您才能真正地称自己为“汇编语言程序员”。

需要记住的另一点是,HLA 标准库函数不是汇编语言的一部分。它们只是为了方便您而预编写的一些函数。虽然调用这些函数没有什么错,但还要记住它们并不是机器指令,而且这些例程没什么特别的;掌握了编写汇编语言代码的方法之后,您就可以编写自己的例程了(甚至可以编写得更加有效)。

如果学习汇编语言是为了编写效率最高的程序(要么最快,要么代码最少),那么应该明白若使用高级控制语句并大量调用 HLA 标准库,就不可能完全达到这一目的。HLA 的代码生成器和 HLA 标准库也并不是效率就非常低下,但是要编写高效的程序,唯一的方法就是使用汇编语言方式进行思考。HLA 的高级控制语句和 HLA 标准库中的许多例程都是很重要的,因为它们可以让您不必用汇编语言来思考。这对于初学汇编语言的用户来说是很重要的,但是如果最终目的是要编写高效率的代码,那么就必须学会用汇编语言。本书将帮您实现这一目标(由于它使用了 HLA

的高级特征,所以要实现这一目标会快得多),但是不要忘记最终的目标是要利用低级编码,从而放弃这些高级特征。

1.13 更多信息

本章介绍了很多基础知识。尽管关于汇编语言编程还有很多知识需要学习,但是本章提供的内容加上您自己所具备的高级语言的知识,足够您开始编写真正的汇编语言程序了。

虽然本章介绍了很多主题,但是其中令人比较感兴趣的三个主题分别是 80x86 CPU 的体系结构、简单 HLA 程序的语法以及 HLA 标准库。如果想要了解更多的信息,可以参阅本书(未删减的)电子版本、HLA 参考手册和 HLA 标准库手册。从 <http://www.artofasm.com/> 和 <http://webster.cs.ucr.edu/> 上可以找到这三个文档。

第 2 章

数 据 表 示



在学习汇编语言的时候，许多初学者遇到的主要障碍都是二进制和十六进制数字系统的用法。虽然十六进制数与通常所用的数的确存在一些差异，但是它们的优点远远大于缺点。理解这些数字系统是很重要的，因为它们简化了许多复杂的问题，包括位运算、有符号数的表示、字符代码以及压缩数据。

本章将讨论几个重要的概念，其中包括：

二进制和十六进制数字系统

- 二进制数据的结构(位、半字节、字节、字以及双字)
- 二进制的有符号和无符号数字系统
- 二进制值的算术、逻辑、移位以及循环移位运算
- 位域和压缩数据

这些都是基础知识，本书剩下的部分将依赖于对这些概念的理解。如果您已经在其他课程或者研究中熟悉了这些术语，那么在学习第 3 章之前至少应该复习一下这些内容。如果您对它们还不熟悉，那么应该认真学习本章内容。本章的内容非常重要，千万不要跳过任何基础知识。

2.1 数字系统

现代的计算机系统大多数都不用十进制数字系统来表示数值。它们一般都采用二进制或者二进制补码数字系统。

2.1.1 回顾十进制系统

由于人们使用十进制数字系统的时间已经很长了，以至于都认为它是理所当然的。当看到数

字“123”时，您不会去考虑数值123，而是在大脑中考虑该数值代表了多少项。实际上，数字123表示的是：

$$1*10^2 + 2*10^1 + 3*10^0$$

或者

$$100+20+3$$

在十进制位置数字系统当中，小数点左边的每一位数字都代表0~9之间的数值乘以10的递增次幂。小数点右边的数字都代表0~9之间的数值乘以10的递减负次幂。例如，数值123.456可表示为：

$$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$$

或者

$$100+20+3+0.4+0.05+0.006$$

2.1.2 二进制数字系统

现代的计算机系统大多都采用二进制逻辑进行操作。计算机使用两个电平(通常是0V和+2.4~5V)来表示数值。有了这两个电平，我们就正好可以表示两个不同的数值。它们可以是任意两个不同的数，但一般都表示0和1。这两个数同时对应于二进制数字系统中所使用的两个数字。

二进制数字系统与十进制系统的原理相同，但有两点差别：二进制只使用数字0和1(而不是0~9)，并且二进制使用2的幂而不是10的幂。因此，将二进制数转换为十进制数非常容易。二进制串中的每个“1”，都要加上 2^n ，其中 n 是从0开始排序的二进制数字的位置数。例如，二进制数值11001010₂表示：

$$\begin{aligned} &1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \\ &= \\ &128 + 64 + 8 + 2 \\ &= \\ &202_{10} \end{aligned}$$

将十进制数转换为二进制数要稍微复杂一些。必须要找到那些加到一起时得到十进制结果的2的幂数。

一种简单的方法叫做“偶/奇-除2”算法。该算法的步骤如下所示：

- (1) 如果是偶数，就得到一个0；如果为奇数，就得到一个1。
- (2) 用2来除这个数并舍弃小数部分或者余数。
- (3) 如果商为0，算法就完成了。
- (4) 如果商不是0而是奇数，就在现有的二进制串之前插入一个1；如果该数为偶数，那么就在二进制串之前附加一个0。

(5) 返回步骤(2)，再重复进行。

虽然在高级语言中，二进制数并不重要，但它们在汇编语言中却处处可见。所以您应该熟悉它们。

2.1.3 二进制格式

从最纯粹的意义上来说，每个二进制数都包含无数个数字(或位(bit))，它是二进制数字(binary digit)的简写)。例如，我们可以用下面任何一种方式来表示数字 5：

101 00000101 0000000000101 ...000000000000101

二进制数前面可以加任意多个前导 0，而不会改变它的值。

我们遵守一个约定，如果在数值中出现了前导 0，就将它们都忽略掉。例如， 101_2 表示数值 5，但是由于 80x86 一般都对 8 位数进行操作，我们发现用 0 将所有的二进制数扩展为 4 位或者 8 位的若干倍，处理起来会容易得多。因此根据这个约定，我们将数字 5 表示为 0101_2 或者 00000101_2 。

在美国，为了能使一些比较大的数更容易读，大多数人都将每三位用逗号隔开。例如，1,023,435,208 就比 1023435208 读起来和理解起来更容易。在本书中我们将对二进制数采用相似的约定，将每四个二进制位分为一组并用下划线来分隔。例如，我们将二进制数 101011110110010 写成 $1010_1111_1011_0010$ 。

我们将采用下面的方式对每一位进行编号：

- (1) 二进制数的最右一位为第 0 位。
- (2) 每向左一位就给一个后继的位号。

8 位二进制数使用位 0~7：

$X_7X_6X_5X_4X_3X_2X_1X_0$

16 位二进制数使用位 0~15：

$X_{15}X_{14}X_{13}X_{12}X_{11}X_{10}X_9X_8X_7X_6X_5X_4X_3X_2X_1X_0$

32 位二进制数使用位 0~31，等等。

第 0 位是低位(L.O.)位(有人把这一位称为最低位)。最左边的位一般叫做高位(H.O.)位(或者最高位)。我们将用位号来引用中间的各个位。

2.2 十六进制数字系统

二进制系统中存在的一大问题就是冗长。数值 202_{10} 需要 8 个二进制数字表示。如果采用十进制表示，3 个数字就足够了。因此，采用十进制表示的数就要比采用二进制紧凑得多。这一事实并没有被设计二进制的工程师遗忘。当要处理一个较大的数值时，二进制数就变得非常不适用了。遗憾的是，计算机是使用二进制进行思考的，所以在大多数时间里，使用二进制数字系统会更加方便一些。虽然我们可以在十进制和二进制之间进行转换，但是这种转换并不简单。十六进制(基数为 16)数字系统解决了二进制系统中固有的很多问题。十六进制数具有我们一直在寻求的

两个特征：它们非常紧凑，并且将它们转换为二进制或者反过来转换都非常简单。因此，大多数计算机系统的工程师都采用十六进制数字系统。

由于十六进制数的基数为 16，所以十六进制小数点左边的每位数字表示该数值与 16 的逐次幂的乘积。例如， 1234_{16} 等于

$$1 \times 16^3 + 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0$$

或者

$$4096 + 512 + 48 + 4 = 4660_{10}$$

每个十六进制数字都可以代表 $0 \sim 15_{10}$ 这 16 个数字之间的某一个。由于只存在 10 个十进制数字，所以我们需要增加六个额外的数字来代表 $10_{10} \sim 15_{10}$ 这几个数。我们并没有为这些数创建新的符号，而是使用字母 A~F 来表示。下面全部都是有效的十六进制数：

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

由于我们经常需要向计算机系统输入十六进制数，所以需要采用不同的机制来表示十六进制数。在大多数计算机上，我们都不能使用下标来表示相关数值的基数。我们将采用如下的约定：

- 所有的十六进制数都以 \$ 字符开头，例如，\$123A4。
- 所有的二进制数都以一个百分号(%)开头。
- 十进制数没有前缀。
- 如果从上下文可以很清楚地知道基数，本书可能会省掉 \$ 或者 % 符号。

下面是一些有效的十六进制数的示例：

\$1234 \$DEAD \$BEEF \$AFB \$FEED \$DEAF

正如您所看到的，十六进制数非常紧凑，且可读性较好。另外，还很容易进行十六进制与二进制之间的转换。考虑如下所示的表 2-1，该表提供了十六进制数与二进制数相互转换时所需的全部信息。

表 2-1 二进制/十六进制转换

二 进 制	十 六 进 制
%0000	\$0
%0001	\$1
%0010	\$2
%0011	\$3
%0100	\$4
%0101	\$5
%0110	\$6
%0111	\$7
%1000	\$8

(续表)

二 进 制	十 六 进 制
%100F	\$9
%1010	\$A
%1011	\$B
%1100	\$C
%110F	\$D
%1110	\$E
%111F	\$F

要将某个十六进制数转换为二进制数，只需要将该数中的每一位十六进制数字替换为4位对应的二进制数。例如，要将\$ABCD转换为二进制数，就只需要参照表 2-1 对每一位十六进制数字进行转换：

A	B	C	D	十六进制
1010	1011	1100	1101	二进制

将二进制数转换为十六进制格式也一样容易。第一步就是将二进制数补零，使得该数的位数变为4的倍数。例如，给定一个二进制数1011001010，第一步就是在该数的左边添加两位使得它包含12位数字。要转换的二进制数就是001011001010。下一步是将二进制数每四位分成一组，例如，0010_1100_1010。最后，在表 2-F 中查这些二进制数并用相应的十六进制数字来替换，也就是\$2CA。将这一过程的难度与十进制与二进制之间的转换或者十进制与十六进制之间的转换进行对比！

由于经常需要在十六进制与二进制之间进行转换，所以最好花上一点时间牢记表 2-1。即使有一个可以完成这种转换的计算器，您也会发现，在进行二进制与十六进制之间的转换时，手动转换会更快、更方便一些。

2.3 数据结构

单纯从数学上来说，一个数值可以取任意多位。而从另一方面来说，计算机一般都只能处理特定位数的值。一般有1位(bit)一组的、4位一组的(半字节, nibble)、8位一组的(字节, byte)、16位一组的(字, word)、32位一组的(双字, double word 或 dword)、64位一组的(四字, quad word 或 qword)、128位一组的(长字, long word 或 lword)等。长度并不是任意的。之所以用这些特定的值，是有充分理由的。本部分将讲述在 Intel 80x86 芯片中通常用到的位组。

2.3.1 位

在二进制计算机上，数据的最小单元是一位。用一位可以表示两个不同的数据项。例如，0或者1、真或者假、开或者关、男或者女、对或者错。但是，位并不局限于表示二进制数据类型(也

就是说,那些只有两个不同数值的对象)。可以用一位来表示数字 723 与 1245,或者表示数值 6254 与 5。也可以用一位来表示红色与蓝色。甚至还可以用一位来表示两个不相关的对象。例如,可以用一位来表示红色与数字 3256。用一位可以表示任意两个不同的数值,但是一位只能表示两个不同的数值。

不同的位可以表示不同的对象。例如,可以用一个位表示数值 0 与 1,而用另一个位表示真与假。那么该如何通过观察这些位来分辨它们呢?当然,答案是您不能分辨。但是这却阐明了计算机数据结构背后的所有思想:数据就是您所定义的对象。如果用一位来表示布尔值(true/false),那么这一位就被定义为表示 true 或者 false。要让一位具有真正的意义,就必须保持前后一致。如果在程序的某一点上,用一位来表示 true 或者 false,那么在后面就不应该用它来表示红或者蓝。

由于试图要表示的内容大多数都需要多于两位,一位数值并不是使用得最多的数据类型。但是,由于其他数据类型都由多位构成的组所组成,所以位将在程序中扮演重要角色。当然,有几种数据类型需要两个不同的数值,所以看起来这些位本身也很重要。但是,您很快就会看到单独的位很难操作,所以我们通常都会用其他数据类型来表示具有两种状态的数值。

2.3.2 半字节

半字节是 4 位一组的数据类型。它并不是特别令人感兴趣,但其中的两种表示法很有用:BCD(binary-coded decimal,二进制编码的十进制数)数¹和十六进制数。它用 4 位来表示一个 BCD 或者十六进制数字。用半个字节,我们可以表示小于 16 的不同数值,因为 4 位串有 16 种不同的组合:

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111

¹ 二进制编码的十进制数是一种用来表示十进制数的方法,它将每一位十进制数字都用 4 位来表示。

对于十六进制数而言,数值 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F 都用 4 位来表示。BCD 只用到了 10 个不同的数字(0、1、2、3、4、5、6、7、8、9),它也需要 4 位(因为用 3 位只能表示 8 个不同的数值,用 4 位可表示的另外 6 个值不在 BCD 表示法中使用)。事实上,任意 16 个不同的数值都可以用一个半字节来表示,但是十六进制和 BCD 数字是我们可以用半字节来表示的主要数据项。

2.3.3 字节

毫无疑问,80x86 微处理器所使用的最重要的数据结构是字节。一个字节由 8 位组成。80x86 的主存和 I/O 地址都是字节地址。这就意味着 80x86 程序能够独立访问的最小单元是 8 位数值。要访问更小的数需要读入包含该数据的字节,并屏蔽掉不需要的位。字节中的位通常都从 0~7 进行编号,如图 2-1 所示。



图 2-1 位编号

位 0 是低位位或者称为最低位,位 7 是字节的高位位或者称为最高位。我们将根据编号来引用它们。

注意一个字节正好包含两个半字节(如图 2-2 所示)。

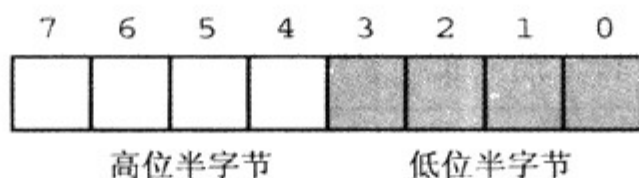


图 2-2 一个字节中的两个半字节

位 0~3 构成了低位半字节,位 4~7 构成了高位半字节。因为一个字节正好包含两个半字节,所以字节数值需要两个十六进制数字。

因为一个字节包含 8 位,所以它可以表示 2^8 (即 256)个不同的数值。一般来说,都使用一个字节来表示 0~255 内的数值、-128~+127 内的有符号数(参阅 2.8 节)、ASCII/IBM 字符代码以及其他一些不需要超过 256 个不同数值的专用数据类型。很多数据类型的数值都少于 256 个,所以 8 位通常已经足够了。

因为 80x86 是字节编址的机器,所以操作整个字节比操作位或者半字节的效率更高。因此,大多数程序员都用一个字节来表示那些不需要超过 256 个数值的数据类型,即使少于 8 位也足够了。例如,我们通常用 00000001_2 与 00000000_2 来分别表示布尔值 true 与 false。

对于一个字节来说,最重要的用途在于保存字符数值。无论是键盘上敲入的字符、屏幕上显示的字符,还是打印机打印的字符,都具有数值。为了让它能与其他数值相通,PC 一般都使用 ASCII 字符集的某种变体。ASCII 字符集定义了 128 个字符代码。

因为字节是 80x86 存储空间中最小的存储单元,字节也正好是 HLA 程序中可以创建的最小变量;正如在第 1 章中所介绍的那样,可以用数据类型 int8 声明一个 8 位的有符号整型变量。因

为 int8 对象是有符号的, 所以可以用 int8 类型的变量表示 -128~+127 之间的数值。int8 类型的变量中应该只存储有符号的数值: 如果想要创建一个任意的字节变量, 就应该使用 byte 数据类型, 如下所示:

```
static
    byteVar:byte;
```

byte 数据类型是一个部分非类型化数据类型。唯一与 byte 对象相关的类型信息就是它的长度(一个字节)。可以将任意一个 8 位数值(较小的有符号整数、较小的无符号整数、字符等)存储为字节变量。您需要自己跟踪存入字节变量的对象类型。

2.3.4 字

字是一个 16 位的组。我们将如图 2-3 中所示那样对一个字中的位从 0~15 进行编号。和字节的编号一样, 位 0 是低位位。对于字来讲, 位 15 是高位位。当要引用字中的其他位时, 我们将使用它们的位置编号。

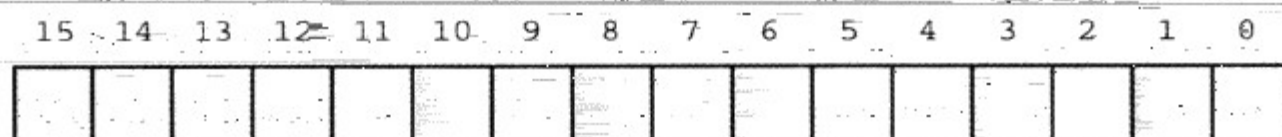


图 2-3 字中的位编号

可注意到, 一个字正好包含两个字节。位 0~7 组成了低位字节, 位 8~15 组成了高位字节(见图 2-4)。

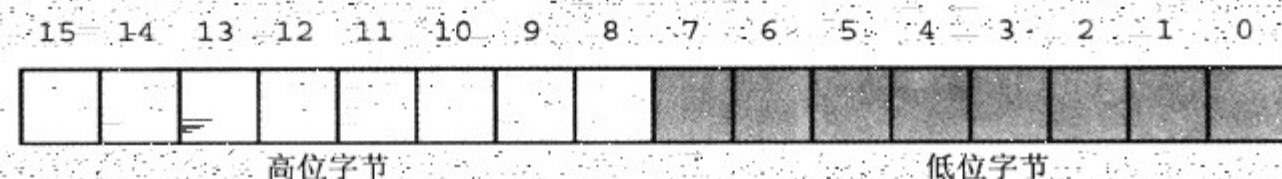


图 2-4 一个字中的两个字节

一个字也可以如图 2-5 那样进一步分解为 4 个半字节。0 号半字节是字中的低位半字节, 3 号半字节是字中的高位半字节。我们将简单地称另两个半字节为“1 号半字节”和“2 号半字节”。

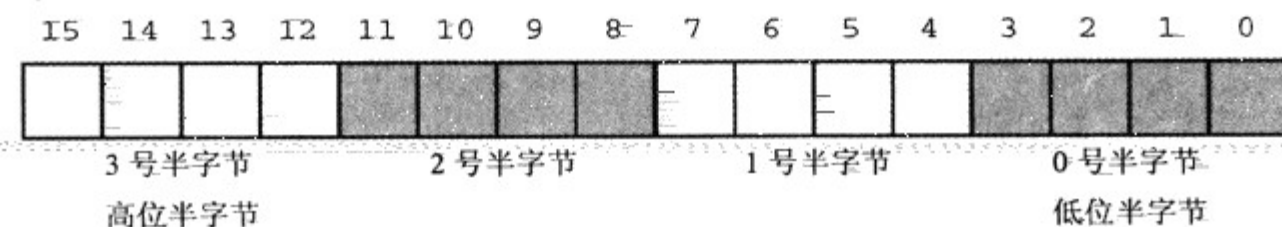


图 2-5 字中的半字节

16 位可以表示 2^{16} (即 65536) 个不同的数值。这些数值可以处于 0~65535 之间, 或者像一般情况下那样, 是 -32768~+32767 之间的有符号数值, 或者是不超过 65536 的任意其他类型的数值。字的三个主要用途分别是有符号短整数值、无符号短整数值以及 Unicode 字符。

字可以表示 0~65535 或者 -32768~+32767 之间的整数值。无符号数用与字中的位相应的二进制值来表示。有符号数使用二进制补码形式来表示(见 2.8 节)。作为 Unicode 字符, 字可以表示多达 65536 个不同的字符, 允许在计算机程序中出现非罗马字符集。和 ASCII 一样, Unicode 是一

种国际标准, 允许计算机处理亚洲语、希腊语以及俄语字符这样的非罗马字符。

与字节相同, 在 HLA 程序中也可以创建字变量。当然, 在第 1 章中您已经看到了如何使用 `int16` 数据类型来创建 16 位的有符号整型变量。下面是一个使用 `word` 数据类型创建字变量的例子:

```
static
    w: word;
```

2.3.5 双字

顾名思义, 双字由两个字组成。因此, 双字有 32 位长, 如图 2-6 所示。

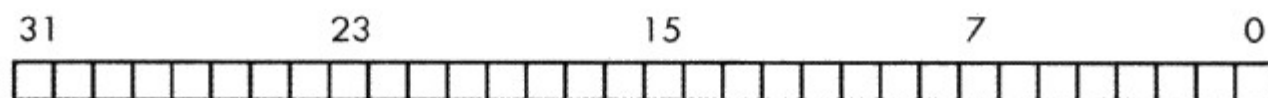


图 2-6 双字中的位编号

双字可以被分成一个高位字和一个低位字、4 个不同的字节或者 8 个不同的半字节(见图 2-7)。

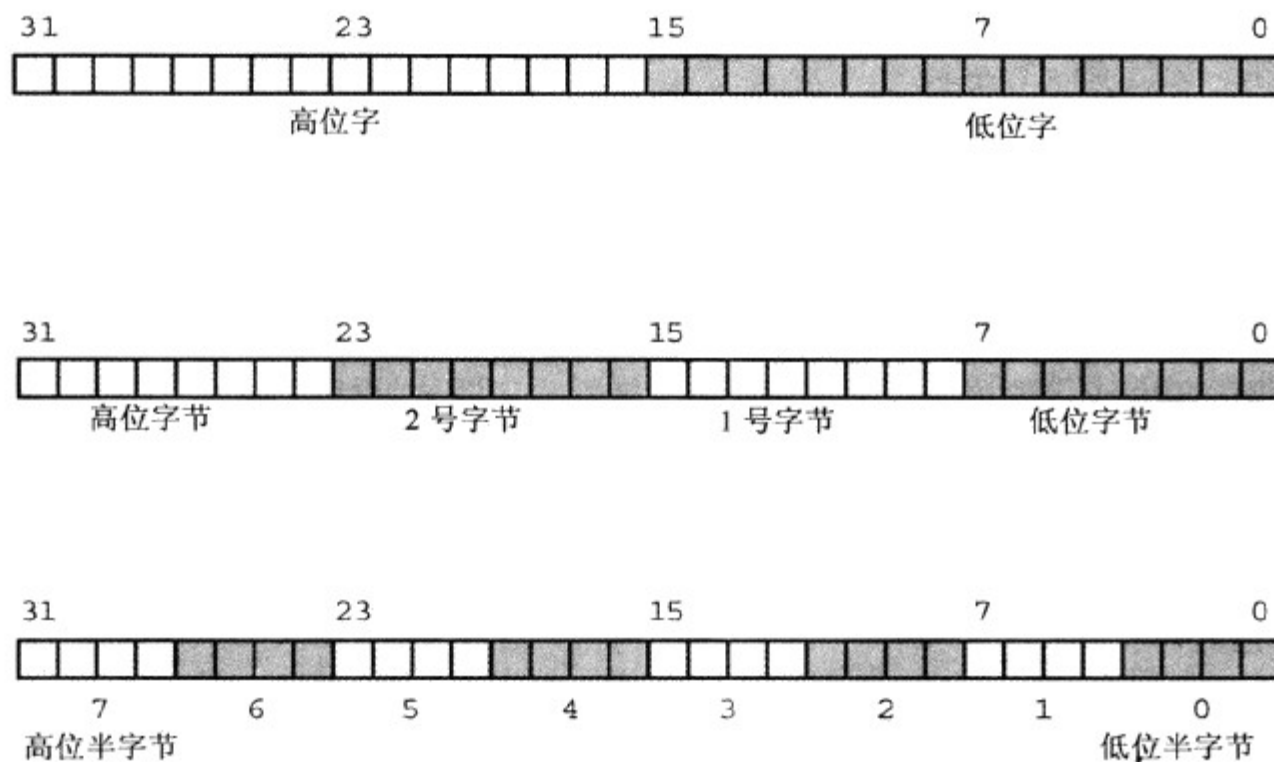


图 2-7 双字中的半字节、字节和字

双字可以表示各种不同的内容。一般双字都用来表示 32 位整型值(允许表示 0~4294967295 之间的无符号数或者 -2147483648~+2147483647 之间的有符号数)。32 位浮点数也可以用双字来表示。双字的另一个常见用法便是存储指针值。

在第 1 章中, 已经看到了如何用 `int32` 数据类型来创建 32 位有符号整型变量。同样也可以用 `dword` 数据类型创建一个任意的双字变量, 如下面的例子所示:

```
static
    d: dword;
```

2.3.6 四字和长字

显然, 我们可以继续定义更长的类型。但是, 80x86 只支持特定的字长, 所以没有理由再继

续定义更长的类型。虽然字节、字以及双字是 80x86 程序中最普遍的字长，四字(64 位)也是非常重要的，因为有些浮点数据类型需要 64 位。同样，现代 80x86 处理器的 SSE/MMX 指令集也可以对 64 位的数值进行操作。长字(128 位)也很重要，因为后来的 80x86 处理器上的 SSE 指令集可以操作 128 位的数值。HLA 允许用 `qword` 和 `lword` 类型声明 64 位和 128 位的数值，如下所示：

```
static
    q    :qword;
    l    :lword;
```

注意，也可以用如下所示的 HLA 声明定义 64 位和 128 位整型值：

```
static
    i64      :int64;
    i128     :int128;
```

但是请注意，不能用像 `mov`、`add` 以及 `sub` 这样的标准指令直接操作 64 位和 128 位的整型数，因为标准的 80x86 整数寄存器每次只处理 32 位。在第 8 章，将看到怎样操作这些扩展精度(extended-precision)的数值。

2.4 二进制数和十六进制数的算术运算

对二进制和十六进制数，我们可以执行若干种运算。例如，我们可以执行加法、减法、乘法、除法以及其他一些算术运算。虽然您不必是这方面的专家，但在一些紧急情况下应该能够用一张纸和一支笔手工完成这些运算。虽然说您应该能够手工完成这些运算，但正确的执行方法便是用计算器来做。市场上有几种这样的计算器，下面列出了十六进制计算器的一些制造商(2010 年)：

- 卡西欧公司(Casio)
- 惠普公司(Hewlett-Packard)
- 夏普公司(Sharp)
- 德州仪器公司(Texas Instruments)

除此之外，其他的计算器制造商也能生产这样的计算器。其中，惠普生产的是最好的，但是他们的产品比其他厂家的更贵。夏普和卡西欧的产品售价都低于 50 美元。如果您打算什么样的汇编程序都做，那么拥有一款这样的计算器是很关键的。

要想明白您为什么应该花钱买计算器，请考虑下面的算术问题：

```
$9
+ $1
---
```

您可能认为这个问题的答案为 \$10。但这并不正确！正确答案是 10，即 \$A，而不是 16(\$10)。减法存在与此相似的问题：

```
$10
- $1
---
```

虽然真正的答案是\$F，但您可能会认为答案为\$9。记住，这个问题问的是“16 与 1 的差”。当然，答案是 15，即\$F。

即使上面的两个问题没有干扰到您，但是当在考虑其他问题时，情急之下您的大脑还是会转回到十进制模式，就会产生不正确的结果。这个例子说明：如果必须手工用十六进制数进行算术运算，那您就需要放松心情认真做。或者可以先将数转换为十进制，再对十进制数进行运算，然后再将它们转换回十六进制。

2.5 关于数字及其表示

许多人都对数字及其表示产生了混淆。初学汇编语言的学生经常会提出这样一个问题：“EAX 寄存器中有一个二进制数，怎样才能将它转换为 EAX 寄存器中的十六进制数呢？”答案是：“不用。”虽然存储器或者寄存器中的数以二进制表示存在很大争议，但是最好将存储在它们当中的数看做是抽象数量(abstract numeric quantity)。像 128、\$80 或者%1000_0000 这样的符号串并不是不同的数；它们只是同一个抽象数量的不同表示，即我们所说的“一百二十八”。在计算机中，数字是不考虑其表示方法的；唯一涉及表示的就是在以人类可读的格式输入或者输出该数值的时候。

数量的可读格式是字符串形式。要用人类可读的格式输出 128，就必须将数值 128 转换为 1 后面跟 2，2 后面跟 8 的三字符序列。这就给出了数量的十进制表示。如果愿意，可以将数值 128 转换为三字符序列\$80。数是相同的，但是我们将它转换成了不同的字符序列，因为我们可能想用十六进制而不是十进制方式表示该数。同样，如果我们想用二进制方式来表示这个数，那么就必须将该数值转换为一个 1 后面跟着七个 0 的串。

默认情况下，当调用 stdout.put 例程时，HLA 以十六进制数字系统显示所有的字节、字、双字、四字以及长字变量。同样，HLA 的 stdout.put 例程也以十六进制来显示所有的寄存器数值。考虑程序清单 2-1 中的程序，它将以十进制形式输入的值转换为它们的十六进制等效值。

程序清单 2-1 十进制到十六进制的转换

```
program ConvertToHex;
#include( "stdlib.hhf" )
static
    value: int32;

begin ConvertToHex;

    stdout.put( "Input a decimal value:" );
    stdin.get( value );
    mov( value, eax );
    stdout.put( "The value ", value, " converted to hex is $", eax, nl );

end ConvertToHex;
```

类似地，寄存器变量以及字节、字、双字、四字或者长字变量的默认输入也是十六进制。程序清单 2-2 中的程序与程序清单 2-1 中的功能相反：它输入十六进制数值，再以十进制形式输出。

程序清单 2-2 十六进制到十进制的转换

```

program ConvertToDecimal;
#include( "stdlib.hhf" )
static
    value: int32;
begin ConvertToDecimal;
    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );
    mov( ebx, value );
    stdout.put( "The value $", ebx, " converted to decimal is ", value, nl );
end ConvertToDecimal;

```

HLA 的 `stdout.put` 例程选择了十进制作为 `int8`、`int16` 和 `int32` 变量的默认输出基数，但并不意味着这些变量都会保存十进制数。记住，存储器和寄存器保存的是数值，而不是十六进制数或者十进制数。`stdout.put` 例程将这些数值转换为串，并输出得到的结果串。在 HLA 语言中，使用十六进制还是十进制进行输出只是一种设计选择而已。可以修改 HLA，使得它将寄存器变量以及字节、字、双字、四字或者长字变量按十进制而不是十六进制输出。如果需要按十进制输出寄存器变量以及字节、字、双字、四字或者长字变量，那么简单地调用 `putiX` 例程中的一个就可以了。`stdout.puti8` 例程按 8 位有符号整数类型输出其参数。任何 8 位参数都有效。所以可以将 8 位的寄存器变量、`int8` 变量或者字节变量作为参数传递给 `stdout.puti8`，其结果总是十进制形式的。`stdout.puti16` 和 `stdout.puti32` 也为 16 位和 32 位对象提供了同样的功能。程序清单 2-3 展示了只使用寄存器 EBX(也就是说，没有使用变量 `value`)的十进制转换程序(与程序清单 2-2 是相对照的)。

程序清单 2-3 没有使用变量的十六进制到十进制的转换

```

program ConvertToDecimal2;
#include( "stdlib.hhf" )
begin ConvertToDecimal2;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );
    stdout.put( "The value $", ebx, " converted to decimal is " );
    stdout.puti32( ebx );
    stdout.newln();

end ConvertToDecimal2;

```

注意，HLA 的 `stdin.get` 例程使用的默认基数与 `stdout.put` 所采用的默认基数相同。也就是说，如果要读入 `int8`、`int16` 或者 `int32` 类型的变量，默认的输入基数为十进制。如果要读入寄存器变量或者字节、字、双字、四字或长字变量，默认的输入基数则为十六进制。在读取寄存器变量或者字节、字、双字、四字或长字变量时，如果要将默认的输入基数改为十进制，那么可以调用 `stdin.geti8`、`stdin.geti16`、`stdin.geti32`、`stdin.geti64` 或 `stdin.geti128`。

如果想按十六进制输入或者输出一个 int8、int16、int32、int64 或者 int128 类型的变量，那么可以调用 stdout.puth8、stdout.puth16、stdout.puth32、stdout.puth64、stdout.puth128、stdin.geth8、stdin.geth16、stdin.geth32、stdin.geth64 或 stdin.geth128 例程。stdout.put8、stdout.put16、stdout.put32、stdout.put64 和 stdout.put128 例程分别按十六进制写 8 位、16 位、32 位、64 位和 128 位的数值。stdin.geth8、stdin.geth16、stdin.geth32、stdin.geth64 和 stdin.geth128 分别读 8 位、16 位、32 位、64 位和 128 位的数值，并将它们返回在寄存器 AL、AX 或者 EAX(或者在一个 64 位或者 128 位的参数位置)中。程序清单 2-4 展示了其中几个例程的使用。

程序清单 2-4 stdin.geth32 和 stdout.puth32 的示例

```
program HexIO;

#include( "stdlib.hhf" )

static
    i32: int32;

begin HexIO;

    stdout.put( "Enter a hexadecimal value: " );
    stdin.geth32();
    mov( eax, i32 );
    stdout.put( "The value you entered was $" );
    stdout.puth32( i32 );
    stdout.newln();

end HexIO;
```

2.6 位逻辑运算

有 4 种基本的逻辑运算用于处理十六进制和二进制数：and、or、xor(异或)和 not。与算术运算不同，要执行这些运算，并不一定需要十六进制的计算器。通常用手工计算比用电子设备更容易一些。逻辑 and 运算是二元(dyadic)²运算(意思是它正好有两个操作数)。这些操作数都是独立的二进制位。and 运算就是：

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

表示逻辑 and 运算的一种简洁方法是用真值表。真值表采用如表 2-2 所示的格式。

² 很多书都把它叫做“二元(binary)”运算。“二元(dyadic)”的意思与之相同，避免了与二进制数字系统发生混淆。

表 2-2 and 运算的真值表

and	0	1
0	0	0
1	0	1

这就如同乘法表一样。左边一列值对应于 and 运算最左边的操作数。表中最上面一行的值对应于 and 运算最右边的操作数。对于特定的一对输入值，位于行列交汇处的值是那两个值一起进行逻辑 and 运算的结果。

用日常用语表述的话，逻辑 and 运算的意思是：“如果第一个操作数为 1，并且第二个操作数也为 1，结果就为 1；否则结果为 0。”我们也可以这样描述：“如果有一个或者两个操作数都为 0，则结果为 0。”

关于 and 运算，请注意一个重要的事实，可以使用它将结果强制为 0。只要操作数中有一个为 0，结果就为 0 而无须考虑其他操作数。例如，在上面的真值表中，标记为 0 的行中只包含 0，标记为 0 的列中也只包含 0。相反，如果一个操作数为 1，那么结果就恰好是第二个操作数的值。and 运算的这些结果是非常重要的，尤其是当我们想将某些位强制为 0 时。我们将在下一节中对逻辑 and 运算的用法深入研究。

逻辑 or 运算也是一个二元运算。它的定义是：

0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1

or 运算的真值表采用了表 2-3 所示的格式。

表 2-3 or 运算的真值表

or	0	1
0	0	1
1	1	1

通俗地讲，逻辑 or 运算就是：“如果第一个操作数或者第二个操作数(或者两个都)为 1，那么结果为 1；否则，结果为 0。”这也被称为同或(inclusive-or)运算。

如果逻辑 or 运算的操作数中有一个为 1，那么结果就总是为 1，而不必考虑第二个操作数的值。如果一个操作数为 0，结果就总是第二个操作数的值。与逻辑 and 运算相同，这是逻辑 or 运算一个很重要的附带结果，后面将证实它是很有用的。

注意，逻辑同或运算的意思与标准英语中的意思是有差异的。考虑这样一个句子：“我将要去商店或者我将要去公园”。这个句子暗示了说话者将要去商店或者去公园而不是两个地方都去。因此，英文 or 的意思与同或运算稍微有点不同；事实上，这是异或运算的定义。

逻辑 xor(异或)运算也是一个二元运算。它的定义如下所示：

```

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

```

xor 运算的真值表如表 2-4 所示。

表 2-4 xor 运算的真值表

xor	0	1
0	0	1
1	1	0

通俗地说，逻辑 xor 运算的意思是：“如果第一个操作数或者第二个操作数为 1，但不是两个都为 1，那么结果就为 1；否则结果就为 0。”注意，异或运算的意思比逻辑 or 运算更接近于单词 or。

如果逻辑异或运算的操作数中有一个值为 1，那么结果就总是另一个操作数的相反值；也就是说，当有一个操作数为 1 时，如果另一个操作数也为 1，结果就为 0；如果另一个操作数为 0，结果就为 1。如果第一个操作数的值为 0，那么结果就正好取第二个操作数的值。这个特征可以让您有选择地将位串中的某些位取反。

逻辑 not 运算是一元运算(意思是它只接受一个操作数)。如下所示：

```

not 0 = 1
not 1 = 0

```

表 2-5 为 not 运算的真值表。

表 2-5 not 运算的真值表

not	0	1
	1	0

2.7 二进制数和位串的逻辑运算

前一节定义的逻辑函数都是为单位操作数定义的。由于 80x86 使用的是由 8 位、16 位或者 32 位构成的组，我们需要对这些函数的定义进行扩展以便对两位以及两位以上的数进行处理。80x86 的逻辑函数是逐位进行操作的。给定两个值，这些函数对第 0 位进行操作就产生第 0 位的结果。它们对输入值的第 1 位进行运算就产生结果的第 1 位，依此类推。例如，如果要对下面两个 8 位数进行逻辑 and 运算，那么可以对每一列进行独立的逻辑 and 运算：

```

%F011_0101
%F110_1110
-----
%1010_0100

```


同样也可以将这种逐位进行的运算应用到其他逻辑函数中。

由于我们已经根据二进制数值定义了逻辑运算，您会发现对二进制数进行逻辑运算要比对其他表示方式进行运算容易得多。因此，如果要对两个十六进制数进行某种逻辑运算，那么应该首先将它们转换为二进制。这一点适用于大多数对二进制进行的基本逻辑运算(例如，`and`、`or`、`xor`等)。

在对位串(例如，二进制数)进行运算时，能够使用逻辑 `and/or` 运算将某些位强行置为 0 或者 1，以及使用 `xor` 运算将某些位取反是非常重要的。这些运算允许您有选择地对位串中的某些位施加运算而不会影响其他位。例如，如果有一个 8 位二进制数 `X`，想要保证第 4~7 位的值为 0，那么可以将该值与二进制数 `%0000_1111` 进行逻辑 `and` 运算。这种按位进行的逻辑 `and` 运算会将 `X` 的高 4 位强行置为 0，而保持低 4 位的值不变。同样，也可以通过将 `X` 与 `%0000_0001` 进行逻辑或运算，将 `X` 与 `%0000_0100` 进行逻辑异或运算来分别将 `X` 的最低位强行置为 1，将 `X` 的第 2 位取反。以这种方式使用逻辑 `and`、`or` 以及 `xor` 运算来操作位串叫做屏蔽(masking)位串。之所以使用这个术语，是因为当我们要将某些位强行置为 0、1 或者对某些位取反时，可以用某个值(对于 `and` 用 1，对于 `or/xor` 用 0)屏蔽其他位。

80x86 CPU 支持四条将这些按位进行的逻辑运算应用到它们的操作数上的指令。这些指令是 `and`、`or`、`xor` 以及 `not`，其中 `and`、`or` 和 `xor` 指令使用的语法与 `add` 和 `sub` 指令的相同，如下所示：

```
and( source, dest );
or( source, dest );
xor( source, dest );
```

这些操作数与 `add` 操作数所受的限制相同。尤其是，`source` 操作数必须是一个常量、存储器或者寄存器操作数，而 `dest` 操作数必须是存储器或寄存器操作数。而且，操作数的长度必须相等，还不能都是存储器操作数。这些指令下列通过等式进行按位的逻辑运算。

```
dest = dest operator source
```

80x86 的逻辑 `not` 指令的语法稍有不同，因为它只有一个操作数。这个指令采用如下所示的形式：

```
not( dest );
```

该指令计算如下所示的结果：

```
dest = not( dest )
```

`dest` 操作数必须是寄存器或者存储器操作数。该指令对指定的目标操作数中的所有位取反。

程序清单 2-5 中的程序接受用户输入的两个十六进制数，并计算它们的逻辑 `and`、`or`、`xor` 以及 `not` 值：

程序清单 2-5 `and`、`or`、`xor` 以及 `not` 的示例

```
program LogicalOp;
#include( "stdlib.hhf" )
begin LogicalOp;
```

```

stdout.put( "Input left operand: " );
stdin.get( eax );
stdout.put( "Input right operand: " );
stdin.get( ebx );

mov( eax, ecx );
and( ebx, ecx );
stdout.put( "$", eax, " and $", ebx, " = $", ecx, nl );

mov( eax, ecx );
or( ebx, ecx );
stdout.put( "$", eax, " or $", ebx, " = $", ecx, nl );

mov( eax, ecx );
xor( ebx, ecx );
stdout.put( "$", eax, " xor $", ebx, " = $", ecx, nl );

mov( eax, ecx );
not( ecx );
stdout.put( "not $", eax, " = $", ecx, nl );

mov( ebx, ecx );
not( ecx );
stdout.put( "not $", ebx, " = $", ecx, nl );

```

end LogicalOp;

2.8 有符号数和无符号数

到目前为止，我们一直都把二进制数当作无符号数。二进制数...00000表示0、...00001表示1、...00010表示2，依此类推直到无穷大。负数又怎样表示呢？有符号数在前面很多地方都用到了，我们也提到了二进制补码数字系统，但是还没有讨论怎样使用二进制数字系统表示负数。这正是本节所要讨论的内容。

为了用二进制数字系统表示有符号数，必须对数施加一个限制：它们的位数必须有限并且位数固定。为此，我们将把位数严格限制为8位、16位、32位、64位、128位或者其他一些较小的位数。

使用固定的位数只能表示一定数目的对象。例如，用8位只能表示256个不同的数值。与正数和0一样，负数本身就是对象；因此，我们必须从256个不同的8位数中拿出一部分来表示负数。换句话说，我们需要占用一些位组合来表示负数。为了公平起见，我们将从所有可能的组合中拿出一半来表示负数，另一半表示正数和0。所以可用一个8位字节来表示-128~-1内的负数和0~127内的非负数。用16位的字可以表示-32768~+32767内的数值，用32位的双字可以表示-2 147 483 648~+2 147 483 647内的数值。一般情况下，用 n 位可以表示 $-2^{n-1} \sim +2^{n-1} - 1$ 内的有符号数。

好了，我们可以表示负数了。但到底应该怎样做呢？其实有很多方法，但是80x86微处理器采用补码形式来表示。在补码系统中，数字的最高位代表符号位(sign bit)。如果最高位为0，那么该数就是正的；如果最高位为1，那么该数就是负的。例如：

对于 16 位数:

\$8000 是负数, 因为最高位为 1。
 \$100 是正数, 因为最高位为 0。
 \$7FFF 是正数。
 \$FFFF 是负数。
 \$FFF (\$0FFF) 是正数。

如果最高位为 0, 那么这个数就为正, 并采用标准的二进制格式表示。如果最高位为 1, 那么这个数就为负数, 并使用补码形式表示。要将一个正数转换为它的负数形式, 即它的补码形式, 可以使用下面的算法:

- (1) 对该数的每一位取反——也就是说, 应用逻辑 not 函数。
- (2) 对取反后的结果加 1, 忽略最高位的溢出。

例如, 计算 5 的 8 位等效值:

%0000_0101	5 (二进制)
%1111_1010	对所有位取反
%1111_1011	加 1 以后得到结果

如果我们取 -5, 并对它进行补码操作, 那么就可以得到原数值 %0000_0101, 这正符合我们的预期。

%1111_1011	-5 的补码
%0000_0100	对所有位取反
%0000_0101	加 1 以后得到结果 (+5)

下面的例子给出了一些正的和负的 16 位有符号数:

\$7FFF: +32767, 最大的 16 位正数
 \$8000: -32768, 最小的 16 位负数
 \$4000: +16384

要将上面的数转换为它们相应的负数(也就是说, 对它们取负值), 可以按照下面的方法进行:

\$7FFF:	%0111_1111_1111_1111	+32767
	%1000_0000_0000_0000	对所有位取反 (8000h)
	%1000_0000_0000_0001	加 1 (8001h 或 -32767)
\$4000h:	%0100_0000_0000_0000	+16384
	%1011_1111_1111_1111	对所有位取反 (\$BFFF)
	%1100_0000_0000_0000	加 1 (\$C000 或 -16,384)
\$8000:	%1000_0000_0000_0000	-32768
	%0111_1111_1111_1111	对所有位取反 (\$7FFF)
	%1000_0000_0000_0000	加 1 (8000h 或 -32768)

\$8000 取反以后就变成了 \$7FFF。再加 1 就得到了 \$8000! 这是怎么回事呢? $-(-32768)$ 与 -32,768 相等吗? 当然不相等了。但是由于 16 位有符号数不能表示 32768, 所以我们不能对最小的负数再取负。

为什么要对这么令人痛苦的数字系统进行纠缠呢？为何不把最高位用作符号标志位，再用剩下的几位来表示该数正的等效值呢(这叫做反码数字系统)？问题在于硬件。当出现这种情况时，对数值取负是最费事的。如果采用补码形式，大多数其他操作都变得与在二进制系统中一样容易。例如，要做加法 $5 + (-5)$ ，其结果为 0。考虑一下当我们在补码系统中将这两个数相加时的情况：

```

% 0000_0101
% 1111_1011
-----
%1_0000_0000

```

结果是第 9 位为进位，而其余位都是 0。当出现这种情况时，如果我们将最高位的进位忽略掉，那么当使用补码数字系统时，将两个有符号数相加就能够得到正确的结果了。这意味着对于有符号数和无符号数的加法和减法，我们可以使用相同的硬件。但是，如果使用其他的数字系统就不行。

通常，补码操作不需要手工完成。80x86 微处理器提供了一条 `neg(negate)` 指令来完成这一操作。而且，只要按符号键(+/- 或者 CHS)，所有的十六进制计算器就都能执行该操作。不过，用手工进行补码操作也很容易，您应该知道怎样做。

另外，应该注意用一组二进制位表示的数据完全依赖于上下文。8 位二进制数值 `%1100_0000` 可以表示一个字符、可以表示无符号十进制数值 192，或者表示有符号数 -64。作为一名程序员，您应该负责定义数据的格式，然后一致使用。

80x86 的取负指令 `neg` 所使用的语法与 `not` 指令相同；也就是说，它只有一个目的操作数：

```
neg(-dest);
```

该指令完成 `dest = -dest` 运算，并且其操作数所受的限制与 `not` 所受的限制相同(它必须是一个存储单元或者寄存器)。`neg` 可以对字节、字、双字对象进行操作。当然，由于这是一个有符号的整数操作，所以它只能对有符号的整数进行操作。程序清单 2-6 中的程序展示了用 `neg` 指令完成的补码操作：

程序清单 2-6 补码操作的示例

```

program twosComplement;
#include( "stdlib.hhf" )

static
    PosValue: int8;
    NegValue: int8;

begin twosComplement;

    stdout.put( "Enter an integer between 0 and 127: " );
    stdin.get( PosValue );

    stdout.put( nl, "Value in hexadecimal: $" );
    stdout.puth8( PosValue );

    mov( PosValue, al );

```

```

not( al );
stdout.put( nl, "Invert all the bits: $", al, nl );
add( 1, al );
stdout.put( "Add one: $", al, nl );
mov( al, NegValue );
stdout.put( "Result in decimal: ", NegValue, nl );

stdout.put
(
    nl,
    "Now do the same thing with the NEG instruction: ",
    nl
);
mov( PosValue, al );
neg( al );
mov( al, NegValue );
stdout.put( "Hex result = $", al, nl );
stdout.put( "Decimal result = ", NegValue, nl );

end twosComplement;

```

正如前面所看到的一样，可以用 `int8`、`int16`、`int32`、`int64` 和 `int128` 数据类型来保存有符号的整型变量。您也看到了像 `stdout.puti8` 和 `stdin.geti32` 这样用于读写有符号整数值的例程。因为本节已经讲得非常清楚，必须对程序中有符号数和无符号数的计算进行区分，可能此时您正在问自己：“我应该如何声明和使用无符号整型变量呢？”

该问题的第一部分“如何声明无符号变量”最容易回答。当声明变量的时候，只需要简单地使用数据类型 `uns8`、`uns16`、`uns32`、`uns64` 和 `uns128` 就可以了，例如：

```

static
    u8:    uns8;
    u16:   uns16;
    u32:   uns32;
    u64:   uns64;
    u128:  uns128;

```

至于这些无符号变量的使用，HLA 标准库为无符号变量的读取和显示提供了一组输入/输出例程。正如您猜到的一样，这些例程包括 `stdout.putu8`、`stdout.putu16`、`stdout.putu32`、`stdout.putu64`、`stdout.putu128`、`stdout.putu8Size`、`stdout.putu16Size`、`stdout.putu32Size`、`stdout.putu64Size`、`stdout.putu128Size`、`stdin.getu8`、`stdin.getu16`、`stdin.getu32`、`stdin.getu64` 和 `stdin.getu128`。除了这些例程用于处理无符号数值以外，它们的使用方法与处理有符号整型数的相应例程一样。程序清单 2-7 中的源代码不但展示了无符号数的 I/O 操作，还展示了在同一个计算中混用有符号和无符号操作时出现的情况。

程序清单 2-7 无符号数的 I/O 操作

```

program UnsExample;
#include( "stdlib.hhf" )

```

```
static
    UnsValue: uns16;

begin UnsExample;

    stdout.put( "Enter an integer between 32,768 and 65,535: " );
    stdin.getu16();
    mov( ax, UnsValue );

    stdout.put
    (
        "You entered ",
        UnsValue,
        ". If you treat this as a signed integer, it is "
    );
    stdout.putu16( UnsValue );
    stdout.newln();

end UnsExample;
```

2.9 符号扩展、零扩展、压缩和饱和

由于采用补码形式表示的整数具有固定的长度，这样就出现了一个小问题。如果要将一个 8 位二进制补码数值转换成 16 位数会怎样呢？这个问题以及它的逆问题(将 16 位数转换成 8 位数)可以通过符号扩展(sign extension)和压缩(contraction)操作来实现。

考虑一下数值 - 64。该数的 8 位二进制补码为\$C0，它的 16 位等效值为\$FFC0。现在来考虑数值+64。该数的 8 位和 16 位等效值分别为\$40 和\$0040。8 位和 16 位数之间的差别可以用这样一条规则来描述：“如果该数为负，那么它的 16 位数字表示中高位字节就应该为\$FF；如果该数为正，那么高位字节就应该为 0。”

将一个有符号数从几位扩展成更多位的数字表示是很容易的，只须将它的符号位复制到新格式的所有附加位中就可以了。例如，要将一个 8 位数符号扩展成为 16 位数，只要简单地将该数的第 7 位数字复制到 16 位数的第 8~15 位中就可以了。如果要将一个 16 位的数符号扩展成为双字，那么将第 15 位复制到双字的第 16~31 位中就可以了。

当要对长度可变的有符号数进行操作时，必须采用符号扩展的方法。您经常需要将一个字节与一个字相加。那么在进行该操作以前必须先将字节符号扩展成为一个字。其他的操作(尤其是乘法和除法)可能需要符号扩展成为 32 位：

符号扩展：

8 位	16 位	32 位
\$80	\$FF80	\$FFFF_FF80
\$28	\$0028	\$0000_0028
\$9A	\$FF9A	\$FFFF_FF9A
\$7F	\$007F	\$0000_007F
	\$1020	\$0000_1020
	\$8086	\$FFFF_8086

要将一个无符号数扩展成为更长的数时，必须对该数进行零扩展。零扩展非常容易：只要将0保存到更长操作数的高位字节中就可以了。例如，要将一个8位数值\$82零扩展成为16位，只需要简单地在高位字节上加0就可得到\$0082。

零扩展：

8 位	16 位	32 位
\$80	\$0080	\$0000_0080
\$28	\$0028	\$0000_0028
\$9A	\$009A	\$0000_009A
\$7F	\$007F	\$0000_007F
	\$1020	\$0000_1020
	\$8086	\$0000_8086

80x86 提供了一些指令可将较短的数符号扩展或者零扩展成为较长的数。表 2-6 列出了一组用于对寄存器 AL、AX 和 EAX 进行符号扩展的指令。

表 2-6 用于扩展 AL、AX 和 EAX 的指令

指 令	解 释
cbw();	通过符号扩展将 AL 中的字节转换为 AX 中的字
cwd();	通过符号扩展将 AX 中的字转换为 DX:AX 中的双字
cdq();	通过符号扩展将 EAX 中的双字转换为 EDX:EAX 中的四字
cwde();	通过符号扩展将 AX 中的字转换为 EAX 中的双字

注意，指令 cwd(将字转换为双字)并不能将 AX 中的字符扩展为 EAX 中的双字。而是将符号扩展后的高位字存储到 DX 寄存器中(符号 DX:AX 告诉您有一个双字，它的高16位保存在 DX 中，低16位保存在 AX 中)。如果想让 AX 符号扩展为 EAX，就应该使用 cwde 指令(将字扩展成为双字)。

上面四条指令是您第一次看到的没有操作数的指令。这些指令的操作数由指令自身隐含。

在下面几章中，您将看到这些指令的重要性，并了解 cwd 和 cdq 指令要使用 DX 和 EDX 寄存器的原因。但是对于简单的有符号扩展操作来说，这些指令存在几个主要的缺点：不能指定源操作数和目的操作数，而且这些操作数都必须是寄存器。

对于一般的符号扩展操作来说，80x86 提供了 mov 指令的扩展指令 movsx(带符号扩展的移动)，在对数据进行复制的同时可对它进行符号扩展。movsx 指令的语法与 mov 指令非常相似：

```
movsx (source, dest);
```

该指令与 mov 指令最大的区别在于目的操作数的数据宽度必须大于源操作数的数据宽度。也就是说，如果源操作数是一个字节，那么目的操作数就必须是字或者双字。同样，如果源操作数是字，目的操作数就必须是双字。另一个区别是目的操作数必须是寄存器；而源操作数可以是存储单元³。movsx 指令不接受常量操作数。

3 这并不能算作是一种限制，因为符号扩展总是在寄存器中的算术操作完成之前进行。

对数值进行零扩展可以使用 `movzx` 指令。它的语法和所受的限制与 `movsx` 指令的相同。将 8 位寄存器(AL、BL、CL 和 DL)零扩展为与之对应的 16 位寄存器, 可以通过向高位寄存器(AH、BH、CH 或者 DH)中载入 0 来实现。显然, 如果要将 AX 零扩展为 DX:AX 或者将 EAX 零扩展为 EDX:EAX, 要做的就是将 DX 或者 EDX 中载入 0⁴。

程序清单 2-8 中的示例程序展示了符号扩展指令的应用。

程序清单 2-8 符号扩展指令

```
program signExtension;
#include( "stdlib.hhf" )

static
  i8:   int8;
  i16:  int16;
  i32:  int32;

begin signExtension;

  stdout.put( "Enter a small negative number: " );
  stdin.get( i8 );

  stdout.put( nF, "Sign extension using CBW and CWDE:", nF, nl );

  mov( i8, aF );
  stdout.put( "You entered ", i8, " ($", al, ")", nl );

  cbw();
  mov( ax, i16 );
  stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

  cwde();
  mov( eax, i32 );
  stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

  stdout.put( nl, "Sign extension using MOVSX:", nl, nl );

  movsx( i8, ax );
  mov( ax, i16 );
  stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

  movsx( i8, eax );
  mov( eax, i32 );
  stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

end signExtension;
```

符号压缩是将一个数转换成一个位数更少、但值相同的数, 该操作要稍微麻烦一些。符号扩展是不会失败的。给定一个 m 位的有符号数, 使用符号扩展总是可以把它转换成 n 位的数(这里的 $n > m$)。遗憾的是, 当给定一个 n 位的数时, 如果 $m < n$, 它就并不总能转换为 m 位的数。例

4 最终您会明白, 零扩展为 DX:AX 或者 EDX:EAX 与使用 `cwde` 和 `cdq` 指令一样必要。

如,考虑一下数值-448。作为一个16位的有符号数,它的十六进制表示为\$FE40。但对于8位数值表示法而言,该数值太大了,所以它不能用8位来表示。这是转换过程中发生上溢的例子。

为了能正确地将一个值符号压缩为另一个值,必须看一下要删除的高位字节。要删除的高位字节只能全都是0或者\$FF。如果遇到其他值,在不发生上溢的情况下就无法对它进行符号压缩。最后,结果数值的高位位必须与从该数中删除的每一位相匹配。这里有一些从16位压缩为8位的例子:

\$FF80 可以被符号压缩为\$80。
 \$0040 可以被符号压缩为\$40。
 \$FE40 不能被符号压缩为8位。
 \$0100 不能被符号压缩为8位。

减小整数长度的另一种方法是用饱和(saturation)来实现。当必须将某个较长的对象转换成较短的对象,并且愿意接受可能的精度损失时,就可以使用饱和操作。用饱和操作对数值进行转换时,如果较大对象没有超出较小对象的表示范围,那么只需要简单地将较大的数复制到较小的数中即可。如果较长数值超出了较短数值的范围,就将它设置为较短对象范围内最大(或者最小)的数值来对该数值进行修剪(clip)。

例如,将一个16位的有符号整型数转换为8位的整型数时,如果该16位数值在-128~+127内,只须简单地将该16位数的低位字节复制到8位数中。如果该16位数的值大于+127,那么就要将该值修剪为+127,并将+127存入8位数中。同样,如果它的值小于-128,那么就要将最终的8位数修剪为-128。将32位的数修剪成为较短的数时,饱和操作的方法相同。如果较长的数值超出较短的数所能表示的范围,就将较短的数简单地设置为该数所能表示的最接近边界的数值。

显然,如果数据宽度较大的数值超出了数据宽度较小数值的范围,那么在进行转换时就会有精度的损失。虽然我们都不希望将数值修剪为较小的界值,但比起引发一次异常,或者拒绝进行计算,有时候这还是可以接受的。对于很多应用,比如音频或者视频处理,修剪后的结果还是可以识别的,所以这是一种合理的转换。

2.10 移位和循环移位

应用于位串的另一组逻辑操作是移位(shift)操作和循环移位(rotate)操作。这两类操作可以进一步细分为左移(left shift)、循环左移(left rotate)、右移(right shift)以及循环右移(right rotate)。这些操作对汇编语言程序员来说是非常有用的。

左移操作将位串中的每一位向左移动一个位置(图2-8给出了一个8位移位操作的例子)。

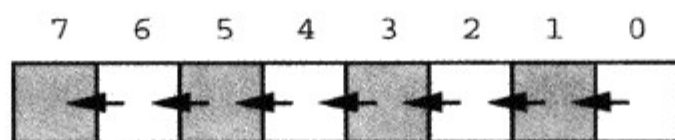


图2-8 左移操作

第0位移到第1位所在的位置,以前位于第1位的值被移动到第2位,依此类推。当然,自然会出现两个问题:“什么数会进入第0位?”和“最高位将移到哪里去?”我们将向第0位移入

一个 0，而最高位中以前的值将变成进位值。

80x86 提供了一个左移指令 `shl` 来完成该操作。`shl` 指令的语法为：

```
shl ( count, dest );
```

操作数 *count* 要么是 CL，要么是 0~*n* 内的一个常数，其中 *n* 比目的操作数的位数小 1(也就是说，对于 8 位的操作数 *n*=7；对于 16 位的操作数 *n*=15；对于 32 位的操作数 *n*=31)。操作数 *dest* 是一个典型的的目的操作数；它可以是存储单元或者寄存器。

当操作数 *count* 为常数 1 时，`shl` 指令完成如图 2-9 所示的操作。

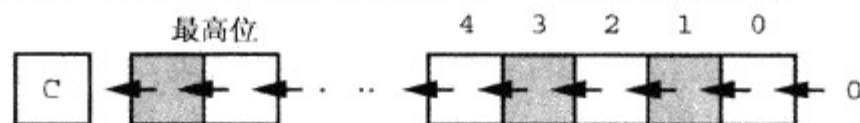


图 2-9 左移操作

在图 2-9 中，C 表示进位标志位。也就是说，从操作数中移出的最高位被移入进位标志位。因此，指令 `shl(1, dest);` 执行完以后可以通过立即对进位标志位进行检测来判断是否发生上溢(例如，可以使用 `if(@c) then ...` 或者 `if(@nc) then ...`)。

Intel 的说明文档指出，如果移动的位数不为 1，那么进位标志位的状态就没有定义。进位标志位通常为移出目的操作数的最后一位，但是 Intel 并不保证这一点。

请注意，将一个数向左移就等同于将它与基数相乘。例如，将一个十进制数向左移一位(在数的右端加一个 0)等效于将它乘以 10(基数)：

```
1234 shl 1 = 12340
```

其中，指令 `shl 1` 的意思是向左移动一位。因为二进制数的基数为 2，将它向左移一位就相当于将它乘以 2。如果要将一个二进制数左移两位，就用 2 与它相乘两次(也就是说，让它与 4 相乘)。如果要将一个二进制数向左移 3 位，就用 8 与之相乘($2*2*2$)。一般来说，如果要将一个数左移 *n* 位，就将它乘以 2^n 。

除了数据移动的方向以外，右移操作与左移操作相同。对于一个字节数，第 7 位移入第 6 位，第 6 位移入第 5 位，第 5 位移入第 4 位，依此类推。在进行右移的时候，我们将 0 移入第 7 位，第 0 位将作为进位位被移出(见图 2-10 所示)。

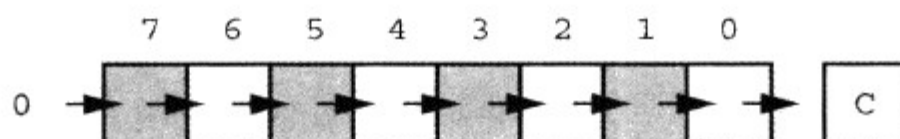


图 2-10 右移操作

正如您所期望的，80x86 提供了 `shr` 指令来完成目的操作数的右移操作。它的语法与 `shl` 指令相同，当然，只是您指定的是 `shr` 而不是 `shl`。

```
shr ( count, dest );
```

该指令向目的操作数的最高位移入 0；将其他的每一位向右移动一个位置(也就是说，从较高位移向较低位)。最后，第 0 位被移入进位标志位中。如果将 *count* 定义为 1，那么 `shr` 指令就进行如图 2-11 所示的操作。



图 2-11 右移操作

Intel 的说明文档中也指出，当移动的位数超过 1 位时，进位位就为未定义状态。

因为左移等效于乘 2，那么右移就应该粗略地等效于除以 2(或者一般情况下，除以该数的基数)。如果右移 n 位，那么就相当于除以 2^n 。

关于除法，右移操作存在一个问题：右移只能等效于除数为 2 的无符号除法。例如，如果将 254(\$FE) 的无符号表示形式向右移动一位，就得到 127(\$7F)，这符合预期。但是，如果将 -2(\$FE) 的二进制表示向右移动一位，就会得到 127(\$7F)，这是错误的结果。之所以出现这个问题，是因为我们向第 7 位移入了一个 0。如果第 7 位以前的值为 1，那么这样做就会把负数变成正数。当除以 2 时，结果就不对了。

为了能够使用右移操作来完成除法运算，我们必须再定义一个移位操作：算术右移。⁵ 算术右移操作与普通的右移操作(逻辑右移)相同，只有一点例外：算术右移操作并不向最高位移入 0，而是将最高位复制回自身；也就是说，在移位的时候并不对最高位进行修改，如图 2-12 所示。

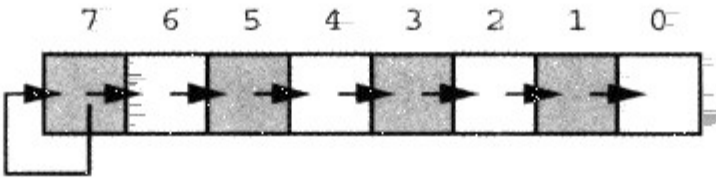


图 2-12 算术右移操作

算术右移操作一般都能够产生希望的结果。例如，如果对 -2(\$FE) 进行算术右移操作，就可以得到 -1(\$FF)。但是关于算术右移操作请记住一点，该操作总是将数值四舍五入为小于或者等于实际结果的最接近的整数值。基于高级语言编程的经验和截取整数的标准规则，大多数人都认为这意味着除法的结果将总是被舍入为 0。但事实并非如此。例如，如果对 -1(\$FF) 进行算术右移，结果是 -1 而不是 0。因为 -1 小于 0，所以算术右移操作将它四舍五入为 -1。这并不是算术右移操作中出现了错误；它只是使用了与整数除法不同(虽然有效)的定义。

80x86 提供了算术右移指令 sar(shift arithmetic right)。该指令的语法大致与 shl 和 shr 相同。其语法为：

```
sar( count, dest );
```

对于操作数 *count* 和 *dest* 的限制通常对它都适用。当 *count* 为 1 时，该指令的操作如图 2-13 所示。

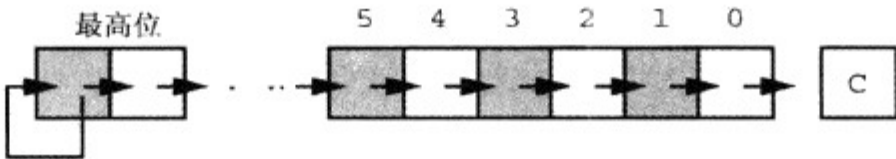


图 2-13 sar(1, dest)操作

Intel 的说明文档中提出当移动的位数超过 1 位时，进位位处于未定义状态。

⁵ 算术左移并不需要。假如没有上溢发生，标准的左移操作对有符号数和无符号数都适用。

另一对有用的操作是循环左移和循环右移。这两种操作与右移和左移操作基本相同，但存在一个主要的差别：从一端移出的位会在另一端被移回。图 2-14 对这两种操作进行了演示。

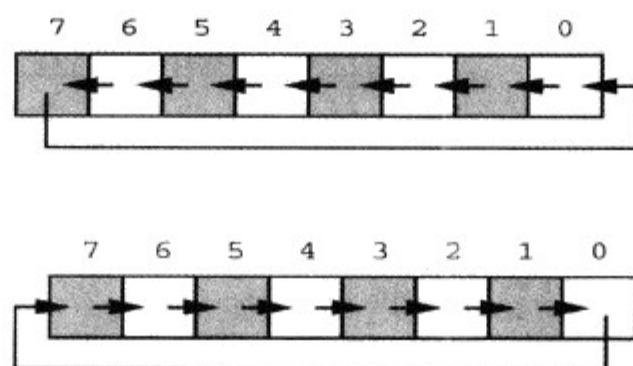


图 2-14 循环左移和循环右移操作

80x86 提供了 `rol`(循环左移)和 `ror`(循环右移)指令来对它们的操作数完成这两种基本的操作。这两条指令的语法与移位指令相似：

```
rol( count, dest );
```

```
ror( count, dest );
```

同样，当移动的位数为 1 时，这些指令也提供了特殊的操作。在此条件下，这两条指令也将移出目的操作数的位复制到进位标志位中，如图 2-15 和图 2-16 所示。

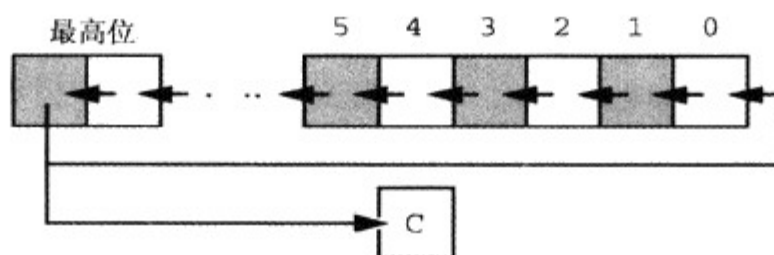


图 2-15 `rol(1, dest)`操作

请注意，Intel 的说明文档提出当循环移动的位数超过 1 位时，进位标志位将处于未定义状态。

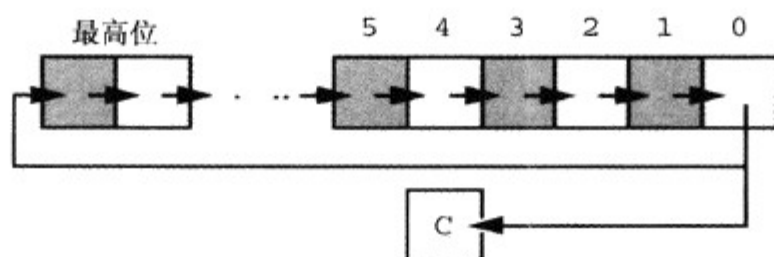


图 2-16 `ror(1, dest)`操作

对于循环移位操作来说，让移位操作中的输出位通过进位标志位移动，并将先前进位标志位的值移回移位操作的输入位通常会更方便一些。80x86 的进位左循环移位指令 `rcl` 和进位右循环移位指令 `rcr` 可以达到这一目的。这两种指令的语法如下所示：

```
rcl( count, dest );
```

```
rcr( count, dest );
```

与其他的移位操作和循环移位操作相同，操作数 `count` 是一个常数或者 CL 寄存器，并且目的操作数 `dest` 为一个存储单元或者寄存器。操作数 `count` 的数值必须小于目的操作数 `dest` 的位数。

对于 count 数值为 1 的情况，这两条指令完成如图 2-17 所示的循环移位。

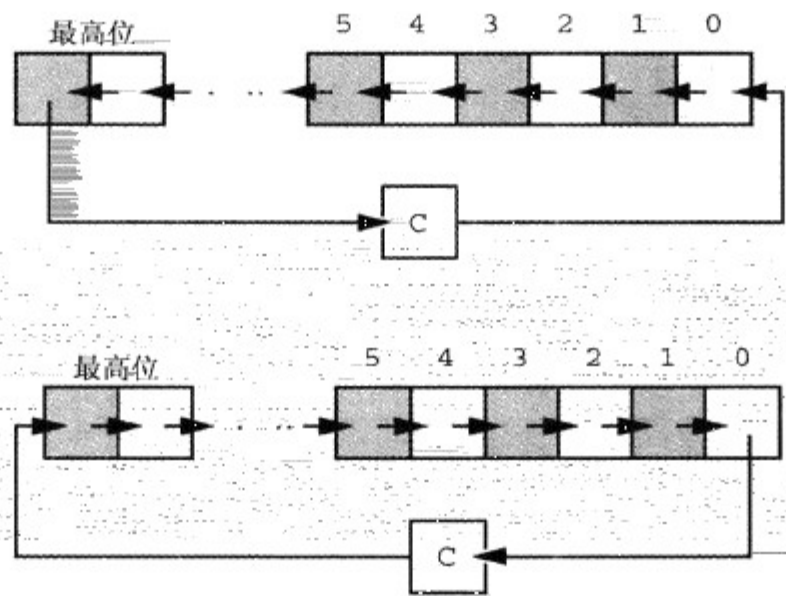


图 2-17 rcl(1, dest)和 rcr(1, dest)操作

Intel 的说明文档提出当循环移动的位数超过 1 位时，进位标志位处于未定义状态。

2.14 位域和压缩数据

虽然 80x86 在对字节、字以及双字数据类型进行处理时效率最高，但偶尔也需要对 8 位、16 位、32 位之外的数据类型进行处理。例如，考虑格式为 04/02/01 的日期。其中用了三个数值来表示这个日期：月份、天数以及年份。当然，月份使用的数值为 1~12。至少需要 4 位(最多 16 个数值)来表示月份。天数的范围为 1~31，所以需要 5 位(最多 32 个不同的数值)来表示。假设我们用 0~99 来表示年份，它需要 7 位(最多可以表示 128 个不同的数值)来表示。4 加 5 加 7 等于 16 位，即两个字节。换句话说就是我们可以将日期数据压缩到两个字节，而不像我们分别用一个字节来表示月份、天数和年份，那样需要三个字节。这样在存储每个日期的时候就节省了一个字节，如果有大量的日期需要存储，就可以节省更多的空间。这些位可以像图 2-18 所示那样分布。

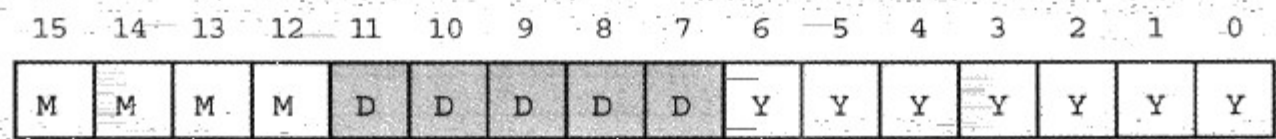
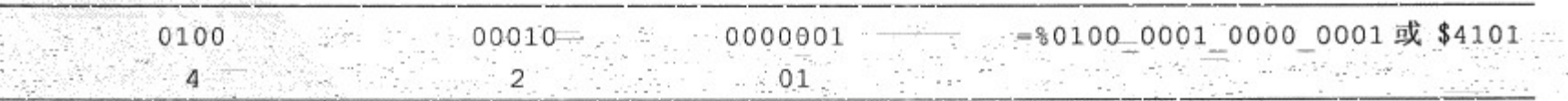


图 2-18 短压缩数据格式(两字节)

MMMM 表示组成月份的 4 位数；DDDDD 表示组成天数的 5 位数；YYYYYYY 表示组成年份的 7 位数。每组表示一个数据项的位的集合就是一个位域(bit field)。例如，2001 年 4 月 2 日表示为 \$4101:



虽然压缩的数值是有空间效率的(也就是说，就存储器的使用来说非常有效)，但在进行计算时效率却非常低(比较慢)。那么是什么原因呢？这是因为它需要额外的指令解压缩不同位域中的数据。这些额外的指令需要消耗额外的时间来执行(并且还需要额外的字节来存储指令)；因此，

必须仔细地考虑压缩后的数据域是否能够达到节省的目的。程序清单 2-9 中的示例程序演示了在这个 16 位的日期格式进行压缩和解压缩的时候需要完成的工作。

程序清单 2-9 压缩和解压缩日期数据

```

program dateDemo;

#include( "stdlib.hhf" )

static
    day:      uns8;
    month:    uns8;
    year:      uns8;

    packedDate: word;

begin dateDemo;
    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    // 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    // m m m m d d d d y y y y y y y y

    mov( 0, ax );
    mov( ax, packedDate ); // Just in case there is an error.
    if( month > 12 ) then
        stdout.put( "Month value is too large", nl );
    elseif( month = 0 ) then
        stdout.put( "Month value must be in the range 1..12", nl );
    elseif( day > 31 ) then
        stdout.put( "Day value is too large", nl );
    elseif( day = 0 ) then
        stdout.put( "Day value must be in the range 1..31", nl );
    elseif( year > 99 ) then
        stdout.put( "Year value must be in the range 0..99", nl );
    else
        mov( month, al );
        shl( 5, ax );
        or( day, al );
        shl( 7, ax );
        or( year, al );
        mov( ax, packedDate );
    end if;
end

```



```
endif;

// Okay, display the packed value:

stdout.put( "Packed data = $", packedDate, nl );

// Unpack the date:

mov( packedDate, ax );
and( $7f, al );           // Retrieve the year value.
mov( al, year );

mov( packedDate, ax );    // Retrieve the day value.
shr( 7, ax );
and( %1_1111, al );
mov( al, day );

mov( packedDate, ax );    // Retrieve the month value.
rol( 4, ax );
and( %1111, al );
mov( al, month );

stdout.put( "The date is ", month, "/", day, "/", year, nl

end dateDemo;
```

当然，在经历“千年虫”的问题后，现在如果再采用一种限制在 100 年(或者甚至于 127 年)以内的日期格式是非常愚蠢的。如果考虑到软件从现在算起能运行 100 年，那么采用三字节的格式可能比采用两字节的格式要明智一些。但是在关于数组的章节中将会看到，应该尽可能地创建长度为 2 的偶数次方的数据对象(单字节、两字节、四字节、八字节等)，否则就会带来性能损失。因此，也许采用四字节将该数据压缩成一个双字变量是一种比较明智的做法。图 2-19 给出了一个四字节日期的数据组织方式。

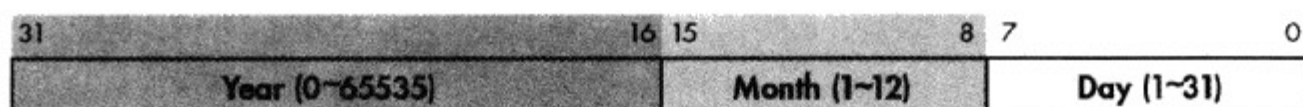


图 2-19 长压缩数据格式(四字节)

在这种长压缩数据格式中，除了对表示年份的位数进行扩展以外，还做了几点变动。首先，因为在 32 位双字变量中存在额外的位，这种格式将额外的位分配给了月份域和天数域。因为这两个域现在分别都由 8 位组成，所以它们可以作为一个字节对象很容易从双字中提取。这样给年份域留下的位数就少了，但是表示 65536 年应该足够了；您几乎可以确定从现在算起 63000 年后，即这一日期格式不再有效的时候，您的软件不可能还在使用。

当然，您可能会争辩说这并不是一种压缩的日期格式。毕竟，我们需要三个数值，其中有两个正好能放到一个字节中，而一个可能至少需要两个字节。因为这种“压缩”的数据格式同没有压缩的版本一样都需要占据四个字节，那么这种格式有什么特别之处呢？您会注意到长压缩数据格式与图 2-18 中的短压缩数据格式之间的另一个差别就是，事实上这种长压缩数据格式对所有的位进行了重新分配，所以年份域到了高位位，月份域处于中间位，而天数域处于低位位。这种排列是非常重要的，因为它使得两个日期很容易进行比较，以判断某个日期是小于、等于还是大于

另一个日期。考虑下面的代码：

```
mov( Date1, eax );           // Assume Date1 and Date2 are dword variables
if( eax > Date2 ) then-      // using the Long Packed Date format.

    << Do something if Date1 > Date2 >>

endif;
```

如果将日期中的各个域分别保存到不同的变量中或者对各个域进行不同的结构划分，就不能用这么直接的方式来对 *Date1* 和 *Date2* 进行比较了。因此，即使没有节省空间，这个例子仍然阐明了对数据进行压缩的另一个原因：它可以使某些运算更加方便或者更加有效(正好与通常进行数据压缩的情况相反)。

在实际应用中，压缩数据类型的例子有很多。可以将 8 个布尔数值压缩到一个字节中；可以将两个 BCD 数压缩到一个字节中，等等。当然，EFLAGS 寄存器是压缩数据的一个典型例子(见图 2-20)。该寄存器将 9 个重要的布尔对象(加上 7 个重要的系统标志)压缩到一个 16 位的寄存器中。其中一些标志需要经常访问。正由于这个原因，80x86 指令集提供了很多方法对 EFLAGS 寄存器中的各个位进行操作。当然还可以在 if 语句或者其他使用布尔表达式的语句中使用 HLA 的 @c、@nc、@z、@nz 等伪布尔变量来测试很多条件码标志。

除了条件码

以外，80x86 还提供了对某些标志直接进行处理的指令(见表 2-7)。

表 2-7 影响某些标志的指令

指 令	解 释
cld();	清除方向标志(设置为 0)
std();	设置方向标志(设置为 1)
cli();	清除中断禁止标志
sti();	设置中断禁止标志
clc();	清除进位标志
stc();	设置进位标志
cmc();	将进位标志取反
sahf();	将 AH 寄存器存储到 EFLAGS 寄存器的低 8 位中
lahf();	从 EFLAGS 寄存器的低 8 位加载 AH

还有其他一些能对 EFLAGS 寄存器产生影响的指令；但是这些指令阐明了如何对 EFLAGS 寄存器中的几种压缩布尔数值进行访问。特别是 lahf 和 sahf 指令为把 EFLAGS 寄存器的低 8 位当作一个 8 位字节来访问提供了一种很便利的途径。图 2-20 为 EFLAGS 寄存器的布局图。

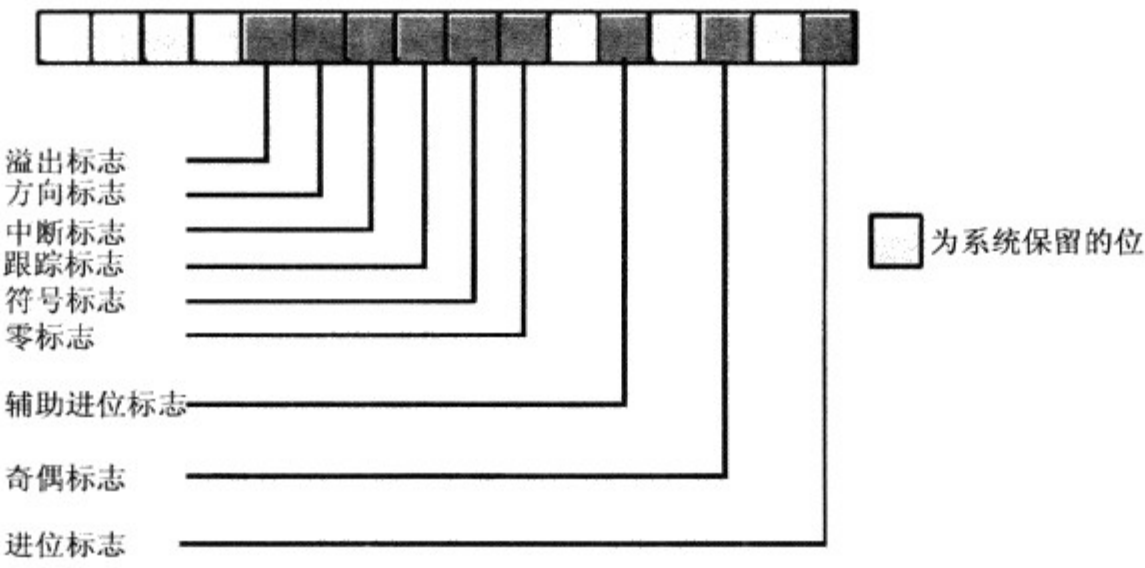


图 2-20 EFLAGS 寄存器的布局

lahf(用 EFLAGS 寄存器的低 8 位装载 AH 寄存器)和 sahf(将 AH 寄存器的 EFLAGS 存储到布局寄存器的低 8 位中)这两条指令的语法如下所示:

```
lahf();  
sahf();
```

2.12 浮点运算简介

整数运算不允许出现小数值。因此, 现在的 CPU 都支持一种实数运算的近似运算: 浮点运算。浮点运算存在的一个大问题就是它不服从代数学的标准规则。但是, 在进行浮点运算时, 很多程序员都采用标准的代数学规则。这就是很多程序发生错误的原因。本节的一个主要目标就是对浮点运算所受到的限制进行描述, 使您明白如何正确使用它。

标准的代数学规则只适用于无限精度(infinite precision)的运算。考虑一个简单的语句 $x := x + 1$, 其中 x 是一个整数。只要不发生上溢, 在现在所有的计算机上, 该语句都遵循代数学的标准规则。也就是说, 该语句只对 x 的某些值有效(最小值 $\leq x <$ 最大值)。大多数的程序员对此并没有疑问, 因为他们都知道这样一个事实, 即程序中的整数并不遵循标准的代数学规则(例如, $5/2 \neq 2.5$)。

整数不服从标准规则是因为计算机只能用有限的位数来表示它们, 所以任何大于最大值或者小于最小值的(整数)数值都表示不出来。浮点数也存在同样的问题, 而且情况只会更糟。毕竟, 整数是实数的子集。因此, 浮点数必须同样能表示整数的无限集合。但是, 在任意两个整数之间都存在无限个实数数值, 所以这个问题就糟糕透顶了。因此, 不但必须将数值限制在最大值和最小值的范围内, 而且并不是处于这个范围内的所有数值都能够表示。

为了表示实数, 大多数的浮点格式都采用科学记数法, 用其中的某几位来表示尾数, 再用更少的位来表示指数。最后的结果是浮点数只能表示有效数字的个数有限的数值。这对浮点运算有很大的影响。为了看到有限精度运算所带来的影响, 我们在例子中将采用一种简化了的十进制浮点格式。我们的浮点格式由带三个有效数字的尾数和一个两位的十进制指数组成。尾数和指数都是有符号数, 如图 2-21 所示。

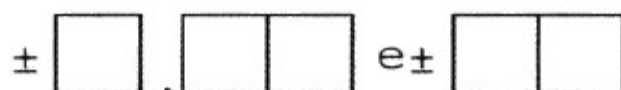


图 2-21 浮点格式

在科学记数法中，对两个数进行加法或者减法操作时必须对这两个数进行调整，使得它们的指数相同。例如，当 $1.23e1$ 与 $4.56e0$ 做加法时，必须对它们的数值进行调整，使它们具有相同的指数。一种方法就是将 $4.56e0$ 转换成 $0.456e1$ ，然后做加法。这样就得到结果为 $1.686e1$ 。遗憾的是，该结果不能满足必须包含三个有效数字这一限制，所以必须将结果四舍五入(round)或者截取(truncate)得到三个有效数字。四舍五入一般能产生最正确的结果，所以对结果进行四舍五入以后就得到 $1.69e1$ 。正如您所看到的，精度(我们在计算中保留的数字或者位的数目)不足会对精确性(计算的正确性)产生影响。

在前面的例子中，我们之所以对结果进行四舍五入，是因为在进行计算的时候，我们保留了四位有效数字。如果在进行运算时我们的浮点运算受到限制，只能保留三位有效数字，那么就不得不将小数部分的最后一位数字截取掉得到 $1.68e1$ ，它就更加不精确了。为了提高浮点运算的精度，有必要在进行计算时增加额外的数字。在运算过程中所增加的额外数字被称为保护数字(guard digit)(或者在二进制格式中称为保护位)。在运算环节中，它们极大地提高了精度。

运算中的精度损失通常不值得担心，除非非常关心运算的精度。但是如果对一组浮点操作进行计算，错误就会发生累积，并对运算本身产生很大的影响。例如，假设要对 $1.23e3$ 与 $1.00e0$ 做加法。首先对这两个数进行调整使得它们的指数相同，然后做加法 $1.23e3 + 0.001e3$ 。即使在进行四舍五入之后，这两个数的和也是 $1.23e3$ 。对于您来说，看起来这似乎是完全合理的：毕竟只能保留三位有效数字，因此对一些较小的数值进行加法时根本就不会影响结果。但是，我们假设要对 $1.23e3$ 累加十次 $1.00e0$ 。第一次将 $1.00e0$ 和 $1.23e3$ 相加得到 $1.23e3$ 。同样，第二次、第三次、第四次、……、第十次将 $1.00e0$ 和 $1.23e3$ 相加时，我们都会得到相同的结果 $1.23e3$ 。另一方面，如果将 $1.00e0$ 自身累加十次，然后再将结果($1.00e1$)和 $1.23e3$ 相加，那么应该会得到一个不同的结果 $1.24e3$ 。对有限精度运算有一定的了解是非常重要的：

求值的顺序会对结果的精度造成影响。

在对浮点数进行加法或者减法操作的时候，如果其中一个数的数量级(即指数)接近于另一个数，那么就能得到更加精确的结果。如果要执行一个涉及加法和减法操作的运算链，那么应该尽量将数据进行适当的分组。

加法和减法的另一个问题是可能最终得到错误的精度。考虑运算 $1.23e0 - 1.22e0$ 。它将产生结果 $0.01e0$ 。虽然这在算术上等效于 $1.00e-2$ ，但是后面这种格式表示后面两位数字精确为 0。遗憾的是，此时我们只得到了一个有效的数字。事实上，某些浮点单元(floating-point unit, FPU)软件包可能会向低位插入随机数字(或者位)。这就又产生了一条关于有限精度运算的重要规则：

两个同号的数相减或者两个异号的数相加时，结果的精度可能小于浮点格式中所能够得到的精度。

乘法和除法不存在与加法和减法同样的问题，因为在操作以前不需要对它们的指数进行调整：需要做的是将指数相加，将尾数相乘(或者将指数相减，将尾数相除)。乘法和除法本身并不会产生精度特别低的结果。但是，它们往往会使得数值中已存在的错误加倍。例如，如果应该将 $1.24e0$ 乘以 2，而您将 $1.23e0$ 乘以 2，那么结果就更加不正确了。这就引入了有限精度运算的又

条重要规则:

在执行一系列涉及加减乘除的运算中,应该先执行乘法和除法运算。

通常,通过应用标准的代数变换就可以安排运算的顺序,使得乘法和除法首先进行。例如,假设要计算 $x*(y+z)$ 。正常顺序是将 y 与 z 相加,再用它们的和乘以 x 。但是如果对 $x*(y+z)$ 进行变换得到 $x*y+x*z$,并通过先执行乘法来计算结果,那么就会得到更高的精度⁶。

并不是乘法和除法本身就不存在问题。将两个非常大或者非常小的数相乘时,很可能会发生上溢或者下溢。当用一个很大的数除以一个很小的数或者用一个很小的数除以一个很大的数时也会出现同样的情况。这就引入了在进行乘法和除法操作时应该尽量遵循的第4条规则:

当对一组数值进行乘法操作和除法操作时,应该尽量安排大数与小数相乘;同样,尽量安排两个具有相同数量级的数值相除。

对浮点数进行比较操作是非常危险的。考虑到任何一个计算都会出现错误(包括将一个输入字符串转换成浮点数值),所以千万不要通过两个浮点数进行比较来判断它们是否相等。在二进制浮点格式中,产生相同(算术)结果的不同运算可能会在最低有效位上发生差异。例如,做加法 $1.31e0+1.69e0$ 应该得到 $3.00e0$ 。同样,做加法 $1.50e0+1.50e0$ 应该得到 $3.00e0$ 。但是,如果对 $(1.31e0+1.69e0)$ 和 $(1.50e0+1.50e0)$ 进行比较就会发现这两个和是互不相等的。当且仅当两个操作数中所有的位(或者数字)都完全相同时,相等性检查才会成功。因为在两个应该产生相同结果的不同浮点运算执行完以后,这一点并不一定会成立,直接测试是否相等可能不会产生正确的结果。

检查两个浮点数是否相等的标准方法是确定在比较中允许存在的误差范围(允差范围),并检查其中一个数值是否在另一个数值的误差范围内。通常的方法就是进行如下所示的检查:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then ...
```

另一种处理这种比较的常用方法就是使用如下格式的句子:

```
if abs(Value1-Value2) <=error then ...
```

在选择 *error* 值时一定要仔细。它应该是比运算中可能出现的最大误差值稍小的数。具体数值取决于所使用的浮点格式,稍后将作详细的阐述。我们要提出的最后一条规则是:

在对两个浮点数进行比较的时候,总是对一个数值进行比较,看它是否处于第二个数值加上或者减去一个小的误差值所得到的范围之内。

在使用浮点数的过程中,还可能发生很多其他的小问题。本书只能指出其中一些主要问题,让您知道不能像对待实数运算一样对待浮点运算这样一个事实——如果不仔细,那么有限精度运算中出现的不准确度将会带来很大的麻烦。关于数字分析或者科学计算的好文章都有助于补充超出本书范围的一些细节知识。如果要使用浮点运算,就应该花时间对运算过程中的有限精度运算所产生的影响进行研究。

HLA 的 *if* 语句不支持涉及浮点操作数的布尔表达式。因此,在程序中不能使用类似于 *if (x < 3.141) then ...* 这样的语句。第6章将介绍如何进行浮点数比较。

⁶ 当然,缺点是现在必须执行两个乘法而不是一个乘法,所以结果会更慢。

2.12.1 IEEE 浮点格式

当 Intel 打算在它的新 8086 微处理器中引入浮点单元(FPU)时,它非常明智地认识到,在为浮点格式选择一种尽可能最好的二进制表示时,设计芯片的电子工程师和固体物理学家可能并不是最佳人选。所以 Intel 公司聘请了一位最好的数字分析专家来为 8087 FPU 设计浮点格式。然后这个专家又聘请了该领域中的另外两位专家,他们三人(Kahn、Coonan 和 Stone)共同设计了 Intel 的浮点格式。他们干得非常漂亮,设计了 KCS 浮点标准,使得 IEEE 组织将该格式作为 IEEE 浮点格式⁷。

为了满足各种性能和精度需求,Intel 实际上引入了 3 种浮点格式:单精度、双精度以及扩展精度。单精度和双精度格式分别对应于 C 语言中的 float 和 double 类型或 FORTRAN 中的 real 和 double precision 类型。Intel 想在较长的运算链中使用扩展精度类型。扩展精度类型包含 16 个额外的位,在存储结果的时候,运算可以在四舍五入到双精度之前将这些额外的位用作保护位。

单精度格式用到一个由反码表示的 24 位尾数和一个 8 位的 excess-127 格式的指数。尾数通常为 1.0~2.0 之间的数值。尾数的最高位总是被假设为 1,表示正好位于二进制小数点(binary point)左边的数值⁸。余下的 23 位尾数位于二进制小数点的右侧。因此,尾数表示为:

1. mmmmmmm mmmmmmm mmmmmmm

mmmm...字符代表尾数中的其他 23 位。记住,我们在这里处理的是二进制数。因此二进制小数点右边的每一个位置表示一个数值(0 或 1)乘以 2 的负逐次幂。隐含的那一位总是乘以 2^0 ,即 1。这就是为什么尾数总是大于或者等于 1 的原因。即使其他的尾数位都为 0,隐含的那一位也总是数值 1⁹。当然,即使在二进制小数点后面存在无限个 1,它们相加仍然不能得到 2。这就是为什么尾数能够表示 1~2 范围内的值的原因。

虽然在 1~2 之间存在无限个数值,但我们只能对其中的 800 万个数进行表示。因为我们使用的是 23 位的尾数(第 24 位总是 1)。这就是浮点运算不精确的原因——在涉及单精度浮点数值运算中,精度被限制为 23 位。

尾数采用的是反码格式而不是补码格式。这就意味着尾数的 24 位数值只是一个无符号二进制数,并且符号位决定了该数值是正还是负。反码数具有不寻常的属性,它对 0 有两种表示(符号位被设置或者清除)。一般情况下,这只对那些设计浮点软件或者硬件系统的人来说很重要。我们假设数值 0 总是将符号位清零。

当要表示 1.0~2.0 这个范围以外的数值时,浮点格式的指数部分才能发挥作用。浮点格式将 2 自乘到指数指定的次幂,再用该数值与尾数相乘。指数为 8 位,以 excess-127 格式存储。此时,指数 2^0 用数值 127(\$7F)表示。因此,要将一个指数转换为 excess-127 格式,只需要简单地将 127 和指数值相加。使用 excess-127 格式简化了浮点数的比较。单精度浮点格式采用图 2-22 所示的形式。

7 处理某些退化运算时所采用的方法有一些小的变化,但是位表示基本上没变。

8 二进制小数点与十进制小数点是相同的,只是它出现在二进制数当中而不是十进制数中。

9 事实上,这并不一定对。IEEE 浮点格式支持高位位不为 0 的非规格化(denormalized)数。但是在我们的讨论当中,将忽略非规格化的数值。

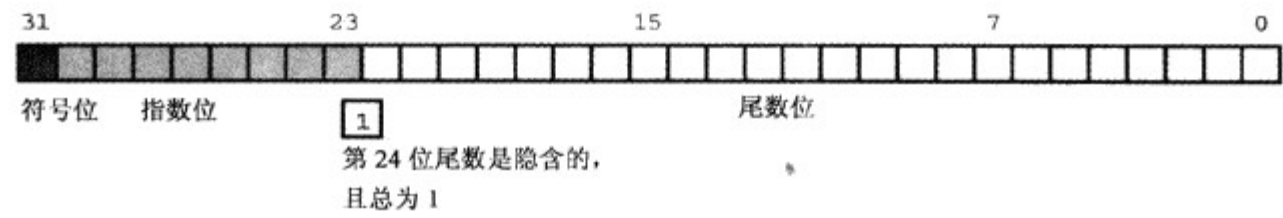


图 2-22 单精度(32 位)浮点格式

采用 24 位尾数所得到的精度近似于 $6\frac{1}{2}$ ($\frac{1}{2}$ 精度的意思是前 6 个数字可以在范围 0~9 之间，但是第 7 个数字只能在范围 0~x 之间，其中 $x < 9$ 且一般都接近 5)。采用 8 位的 excess-127 格式的指数时，单精度浮点数的动态范围大约为 $2^{\pm 128}$ 或者 $10^{\pm 38}$ 。

虽然单精度浮点数适合于很多情况，但是动态范围却受到一些限制，且不适合于金融、科学以及很多其他的应用。而且，在进行长链运算的时候，单精度浮点格式的有限精度可能会引入很严重的错误。

双精度浮点格式能够克服单精度浮点数的缺点。双精度浮点格式占用了两倍的空间，具有一个 11 位的 excess-1023 格式的指数和一个 53 位的尾数(带一个隐含为 1 的高位位)再加上一个符号位。这提供了一个大约为 $10^{\pm 308}$ 的动态范围和 $14\frac{1}{2}$ 精度的数字，它满足大多数的应用。双精度浮点数值采用图 2-23 所示的格式。

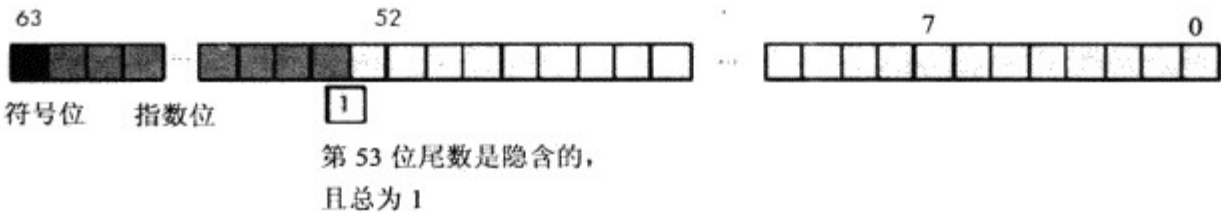


图 2-23 64 位双精度浮点格式

在涉及到双精度浮点数的长链运算时，为了确保精度，Intel 设计了扩展精度格式。扩展精度格式需要占用 80 位。附加的 16 位中有 12 位被附加到尾数上，另外 4 位被添加到指数部分的末尾。与单精度和双精度数值不同，扩展精度格式的尾数不包含隐含的且总是为 1 的高位位。因此扩展精度格式提供了一个 64 位的尾数、一个 15 位的 excess-16383 格式的指数以及一个 1 位的符号位。扩展精度浮点数值的格式如图 2-24 所示。

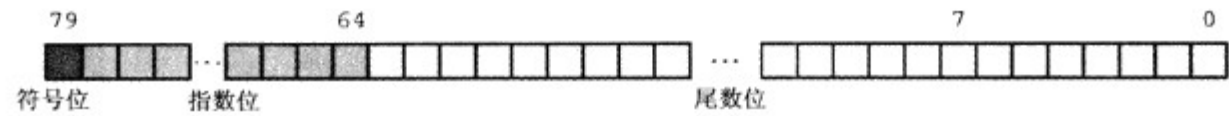


图 2-24 80 位扩展精度浮点格式

在 FPU 上进行的所有运算都采用扩展精度格式。只要加载一个单精度或者双精度的数值，FPU 就会自动将它转换成一个扩展精度的数值。同样，只要向存储器中存入一个单精度或者双精度的数值，FPU 就会在存入以前自动将该数值四舍五入到适当的长度。Intel 公司总是利用扩展精度格式进行操作，因而保证了引入大量的保护位来确保运算的精度。

为了在运算中维持最高的精度，大多数运算都使用规格化的数值。规格化浮点数值是那些尾数最高位为 1 的数。几乎所有的非规格化数都能够被规格化；将尾数的各位向左移，减小指数直到尾数的高位位出现 1 为止。记住，指数是二进制的。每增加一次指数，就是将浮点数值乘以 2。

同样, 每减小一次指数就是将浮点数值除以 2。同理, 将尾数左移一个位置就要将浮点数乘以 2; 将尾数向右移就要将浮点数除以 2。因此, 将尾数左移一个位置并将指数减 1 根本不会改变浮点数的数值。

使浮点数规格化是有好处的, 因为它能保证运算结果最大的精度。如果尾数的高位位都是 0, 那么尾数为运算保持的精确位就要少得多了。因此, 如果一个浮点运算只包含规格化的数值, 那么它就会更加精确。

有两种重要的情况, 浮点数不能规格化。0 是特殊情况之一。0 不能被规格化, 是因为它的浮点表示中尾数不含值为 1 的位。但是, 这并不是问题, 因为我们可以只用一位来表示数值 0。

第二种情况就是当尾数的高位位为 0 但指数也为 0(我们不能减小它来对尾数进行规格化)的时候。对于尾数的高位位和指数都为 0 的那些很小的数, IEEE 标准并没有不接受它们, 而是用一些特殊的非规格化数来表示¹⁰。虽然与发生下溢的情况相比较, 使用非规格化数值可以使 IEEE 浮点运算产生更准确的结果, 但是它提供的精确位更少。

2.12.2 HLA 为浮点数值提供的支持

HLA 提供了几种数据类型和库例程来支持汇编语言程序中浮点数据的使用, 包括用于声明浮点变量的内置类型以及提供浮点输入、输出以及转换的例程。

在对 HLA 提供的浮点支持进行讨论时, 最好从浮点字面常量开始。HLA 浮点常量采用如下所示的语法:

- 一个可选的+号或-号, 代表尾数的符号(如果这个符号没有出现, HLA 就假定尾数为正)
- 后面跟着一位或者多位十进制数
- 后面跟着一个小数点和一位或者多位十进制数(本项是可选项)
- 后面跟着一个 e 或者 E, 再可选地跟着一个符号位(+或者-)和一位或者多位十进制数(本项是可选项)

注意, 为了将该数值与整数或者无符号字面常量进行区分, 必须出现小数点或者 e/E。下面是一些合法的浮点字面常量:

```
1.234 3.75e2 -1.0 1.1e-1 1e+4 0.1 -123.456e+789 +25e0
```

注意浮点字面常量不能从小数点开始; 它必须以十进制数字开头, 所以在程序中必须用“0.1”来表示“1”。

HLA 还允许在浮点字面常量中两个连续的十进制数字之间插入一个下划线符号“_”。可以用下划线符号代替逗号(或者其他语言指定的分隔符)以使得较长的浮点数更容易阅读。下面是一些例子:

```
1_234_837.25 1_000.00 789_934.99 9_999.99
```

可以用数据类型 real32、real64 或者 real80 来声明浮点变量。像整数和无符号数值一样, 这些数据类型声明末尾的那个数指定了每种类型的二进制表示所使用的位数。因此, real32 用于声

¹⁰ 另一种办法是将数值下溢为 0。

明单精度实数，`real64` 用于声明双精度浮点数，`real80` 用于声明扩展精度浮点数。除了使用这些类型来声明浮点数而不是整数这一事实以外，它们的用法基本与 `int8`、`int16`、`int32` 等相同。下面的例子展示了这些声明以及它们的语法：

```
static
    fltVar1:          real32;
    fltVar1a:         real32 := 2.7;
    pi:               real32 := 3.14159;
    DblVar:           real64;
    DblVar2:          real64 := 1.23456789e+10;
    XPVar:            real80;
    XPVar2:           real80 := -1.0e-104;
```

要以 ASCII 格式输出浮点变量，可以使用例程 `stdout.putr32`、`stdout.putr64` 或者 `stdout.putr80`。这些过程以小数点表示法显示数值——也就是说，一个数字串、一个可选的小数点再加上一个数字串结尾。除了它们的名称以外，这三个例程使用完全相同的调用序列。下面显示了对这些例程的调用方法以及相应参数：

```
stdout.putr80( r:real80; width:uns32; decpts:uns32 );
stdout.putr64( r:real64; width:uns32; decpts:uns32 );
stdout.putr32( r:real32; width:uns32; decpts:uns32 );
```

这些过程的第一个参数为要打印的浮点数值。该参数的长度必须与过程的名称相匹配(例如，当调用 `stdout.putr80` 时，参数 `r` 必须是一个 80 位的扩展精度浮点变量)。第二个参数为输出文本指定了域宽；这是过程显示数据时所需要的打印位置数。注意，该宽度必须包含数据的符号位和小数点的打印位置。第三个参数指定了小数点后面的数所需要的打印位置。例如：

```
stdout.putr32( pi, 10, 4 );
```

显示数值：

```
3.14159
```

在本例中下划线表示前导空格。

当然，如果该数非常大或者非常小，那么在浮点数输出时就会想到采用科学记数法而不是小数点表示法。HLA 标准库例程 `stdout.pute32`、`stdout.pute64` 以及 `stdout.pute80` 为此提供了方便。这些例程采用的过程原型如下所示：

```
stdout.pute80( r:real80; width:uns32 );
stdout.pute64( r:real64; width:uns32 );
stdout.pute32( r:real32; width:uns32 );
```

与小数点输出例程不同的是，这些科学记数法输出例程不需要第三个参数来指定小数点后面要显示的数字的位数。参数 `width` 间接指定了该数值，因为所有的尾数位当中总是只有一位出现在小数点左边。这些例程以小数点表示法输出它们的数值，与下面所示的类似：

```
1.23456789e+10  -1.0e-104  1e+2
```

还可以使用 HLA 标准库中的 `stdout.put` 例程来输出浮点数值。如果在 `stdout.put` 的参数列表中指定一个浮点变量的名称, `stdout.put` 程序就会采用科学记数法输出该数值。实际的域宽取决于浮点变量的长度(在这种情况下, `stdout.put` 例程会尝试尽可能多地输出有效数字)。例如:

```
stdout.put( "XPVar2 = ", XPVar2 );
```

如果使用一个冒号并在后面给出一个有符号整数值来指定域宽, 那么 `stdout.put` 例程就会调用适当的 `stdout.putEXX` 例程显示该数值。也就是说, 该数仍然会以科学记数的形式出现, 但是可以控制输出数值的域宽。与整数和无符号数的域宽一样, 域宽为正数表明指定域中的数为右对齐, 域宽为负数说明指定域中的数为左对齐。这里给出了一个使用 10 个打印位置打印变量 `XPVar2` 的例子:

```
stdout.put( "XPVar2 = ", XPVar2:E0 );
```

如果想用 `stdout.put` 以小数形式打印浮点数, 就需要使用下面所示的语法格式:

```
Variable Name : Width : DecPts
```

注意, *DecPts* 域必须是非负的整数值。

当 `stdout.put` 中包含这种格式的参数时, 它就会调用相应的 `stdout.putrXX` 例程显示指定的浮点数值。作为例子, 考虑如下所示的调用:

```
stdout.put( "Pi = ", pi:5:3 );
```

其对应的输出为:

```
3.142
```

HLA 标准库提供了其他几种有用的例程, 可以在输出浮点数值的时候使用这些例程。关于这些例程的更多信息可以参阅 HLA 标准库参考手册。

HLA 标准库提供了几种例程, 它们可以采用不同格式显示浮点数据。与之相比, HLA 标准库只提供了两种例程支持浮点输入: `stdin.getf()` 和 `stdin.get()`。`stdin.getf()` 例程需要用到 80x86 的 FPU 栈, 它是一个硬部件, 本章不准备对它进行讨论。因此我们把对 `stdin.getf()` 的讨论推迟到第 6 章进行。因为 `stdin.get()` 例程提供了 `stdin.getf()` 例程具备的所有功能, 所以这样推迟以后不会出现什么问题。

您已经看到了 `stdin.get()` 例程的语法; 它的参数列表只包含一个变量名列表。`stdin.get()` 函数为用户读入参数列表中出现的每个变量的数值。如果浮点变量的名称已经指定了, 那么 `stdin.get()` 例程就会自动从用户处读入浮点数值, 并将结果存入指定变量中。下面的例子展示了该例程的应用:

```
stdout.put( "Input a double precision floating-point value:" );
stdin.get( DblVar );
```

警告:

本节已经讨论了如何声明浮点变量, 以及如何对它们进行输入和输出, 但没有对算术运算进行讨论。浮点算术运算与整数的算术运算是不同的; 不能使用 80x86 的 `add` 和 `sub` 指令来对浮点数值进行操作。浮点算术运算将在第 6 章讲述。

2.13 BCD 数据表示

虽然整数和浮点格式已经可以满足一般程序的数据表示需求,但是在一些特殊的情况下,其他的数字表示方式更加方便一些。在本节中,我们将对二进制编码的十进制(BCD)格式进行讨论,因为 80x86 CPU 为这种数据表示方式提供了少量的硬件支持。

BCD 数值是由一些半字节组成的序列,每个半字节表示 0~9 中的某个数值。当然,也可以使用半字节来表示 0~15 中的数值;但是在每个半字节所能表示的 16 个不同数值中,BCD 格式只使用了其中的 10 个。

BCD 数值中的每个半字节表示一个十进制数字。因此,我们用一个字节(也就是两个数位)可以表示包含两个十进制数字的数值或者 0~99 内的数值(见图 2-25)。用一个字,我们可以表示具有四个十进制数字的数值或者说 0~9999 内的数值。同样,使用一个双字,我们可以表示具有八个十进制数字的数值(因为在一个双字数值当中存在八个半字节)。

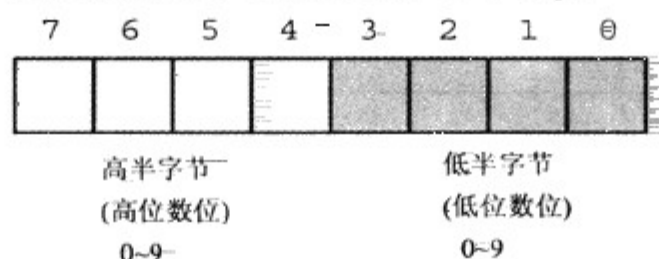


图 2-25 存储器中 BCD 数据表示

正如您所看到的,BCD 存储并没有有效利用存储器。例如,一个 8 位的 BCD 变量可以表示 0~99 内的数值,然而当保存二进制数值的时候,同样的 8 位可以表示 0~255 内的数值。同样,16 位的二进制数值可以表示 0~65535 内的数值,而一个 16 位的 BCD 数值只能表示这些数值中的 1/6(0~9999)。存储效率低还不是唯一的问题。BCD 的计算往往比二进制的计算更慢。

在此,您可能想知道为什么会有人采用 BCD 格式。BCD 格式确实存在两个优点:BCD 数值的内部数字表示和它们的字符串表示之间进行转换是非常容易的;而且,在用硬件(例如,使用微型旋轮或者刻度盘)完成多位十进制数值的编码时,使用 BCD 码比使用二进制要容易得多。正是由于这两个原因,您很可能在嵌入式系统(例如,微波炉、闹钟和核反应堆)中看到人们使用 BCD 码,但是却很少看到它们应用于一般的计算机软件中。

几十年以前,人们错误地认为涉及 BCD 码(或者只是“十进制”)的算术运算比二进制运算更加精确一些。因此,他们通常都采用基于十进制的算术运算来执行那些重要的计算,比如美元和美分(或者其他的金融单位)的计算。虽然某些采用 BCD 码的计算能够产生更加精确的结果,但是这种说法在一般情况下是不正确的。事实上,对于大多数计算(甚至那些涉及定点十进制的算术运算)来说,二进制表示更加精确一些。因此,现在大多数的计算机程序都采用二进制格式来表示所有的数值。例如,Intel 的 80x86 浮点单元(FPU)对两条用于存储和取出 BCD 数值的指令提供了支持。然而在内部,FPU 将这些 BCD 数值转换为二进制表示,并采用二进制来执行所有的计算。它只将 BCD 码用作外部数据格式(也就是说,在 FPU 的外面)。这样一般就能产生更加精确的结果,与专门拥有一个能支持十进制计算的协处理器相比,它占用的硅片面积更小。

2.14 字符

在个人计算机上最重要的数据类型可能就是字符类型。“字符”指的是人或者机器可读的符号，一般都是非数字实体。一般来说，“字符”指的是在键盘上能够正常输入(包括那些需要多个按键才能得到的)或者在显示器上能够显示的字符。很多初学者通常会对“字符”与“字母字符”产生混淆。这些术语是不一样的。标点符号、数字符号、空格符、制表符、回车符、其他的控制字符以及特殊字符也都属于字符。当本书中用到术语“字符”时，指的是这些字符中的任意一种，而不只是字母字符。当提到术语“字母字符”时，本书会使用像“字母字符”、“大写字符”或者“小写字符”之类的术语。

当初学者第一次遇到字符数据类型时，另一个普遍存在的问题是数字字符与数字之间的区别。字符1与数值1是不同的。计算机使用两种不同的内部表示来表示数字字符(0、1、...、9)和数值0~9。一定要小心，不要将两者混淆。

大多数计算机系统都使用单字节或者双字节序列将各种不同的字符编码为二进制形式。Windows、Mac OS X、FreeBSD 和 Linux 也属于这一类，使用 ASCII 码或者 Unicode 来对字符进行编码。本部分将对 HLA 提供的 ASCII 字符集和字符声明方法进行讨论。

2.14.1 ASCII 字符编码

ASCII(American Standard Code for Information Interchange, 美国信息交换标准码)字符集将 128 个文本字符映射为无符号整数值 0~127(\$0~\$7F)。当然，所有的对象在计算机内部都采用二进制数来表示，所以计算机用二进制数值来表示像字符这样的非数字实体就不值得惊讶了。虽然字符到数字的映射是任意的，并且不太重要，但是使用标准化的代码来表示这种映射是很重要的，因为需要与其他程序和外设进行通信，且需要与这些程序和设备使用相同的“语言”。这就该 ASCII 代码发挥作用了；它是一种已经达成一致的标准化代码。因此，如果用 ASCII 代码 65 来表示字符 A，那么就会知道无论何时将这一数据传递给外设(如打印机)，它们都会将该数值解释为字符 A。

不应该认为计算机系统中只使用 ASCII 一种字符集。IBM 在它的很多大型机系统中都使用 EBCDIC 字符集。另一种通用字符集是 Unicode 字符集。Unicode 是对 ASCII 字符集的扩展，它采用 16 位而不是 7 位来表示字符。这就允许在字符集中使用 65536 个字符，将世界上不同语言的大部分符号包含到一种统一的字符集中。

因为 ASCII 字符集只提供了 128 个不同的字符，而一个字节可以表示 256 个不同的数值，这样就产生了一个很有趣的问题：“我们应该如何处理存储到一个字节中的数值 128~255？”一种方法是忽略那些额外的数值。这是本书采用的最主要的方法。另一种可能性是扩展 ASCII 字符集，向字符集中加入 128 个字符。当然，这往往不能得到一种标准化的字符集，除非您能让所有人都赞成这种扩展。这是一项很困难的任务。

当 IBM 首次生产出 IBM-PC 时，定义了这些额外的 128 个字符代码来保存各种非英文字母字符、由线画成的字符、数学符号以及其他几种特殊字符。因为 IBM 公司的 PC 是我们今天所说的一般 PC 的基础，所以它的字符集已经成为所有与 IBM-PC 兼容的机器的一种伪标准。甚至在现代的一些与 IBM-PC 不兼容且不能运行早期 PC 软件的机器上，IBM 的扩展字符集仍然存在。但

是,要注意该 PC 字符集(ASCII 字符集的扩展)并不是通用的。当使用自带的字体时,大多数打印机都不会打印扩展字符,并且很多程序(尤其是在非英语的国家)都不在 8 位数值中使用扩展字符来表示额外的 128 个字符代码。正是因为这些原因,本书一般都坚持使用标准的 128 字符的 ASCII 字符集。

尽管事实上它是一个标准,但如果只是简单地使用标准的 ASCII 字符对数据进行编码,并不能保证各个系统间的兼容性。虽然大多数情况下,一台机器上的 A 很可能就是另一台机器上的 A,但是关于控制字符的使用,不同机器之间的标准化非常少。事实上,32 个控制代码加上 DELETE,只有 4 个控制代码得到了普遍的支持——空格(BS)、制表符、回车符(CR)以及换行符(LF)。更糟糕的是,不同机器通常采用不同的方式来使用这些控制代码。行末是一个特别麻烦的例子。

Windows、MS-DOS、CP/M 以及其他系统采用两字符的序列 CR/LF 来对行末进行标记。Apple 的 Macintosh 和很多其他系统只用一个 CR 字符来标记行末。Linux、Mac OS X、FreeBSD 以及其他 UNIX 系统使用单个 LF 字符来对行末进行标记。无须多说,要在这些系统之间交换文件会遇到很多麻烦。即使这些系统上的所有文件采用的都是标准的 ASCII 字符,在它们之间交换文件时仍然需要对数据进行转换。幸运的是,这样的转换都十分简单。

尽管 ASCII 数据还存在一些主要的缺点,它仍然是计算机系统以及程序之间进行数据交换的标准。大多数程序都能够识别 ASCII 数据;同样,大多数程序都能够生成 ASCII 数据。因为要使用汇编语言来处理 ASCII 字符,所以需对字符集的编排进行研究,记住几个关键的 ASCII 代码(例如,0、A、a 等)是比较明智的。

ASCII 字符集被分成 4 组,每组由 32 个字符组成。前 32 个字符,即 ASCII 码 0~\$1F(31)构成了非打印字符(即控制字符)的特殊字符集。我们之所以把它们称为控制字符,是因为它们执行的是各种打印/显示的控制操作,而不是显示符号。这一类例子包括回车(将光标放到字符所在行的左边)¹¹、换行(将光标移到输出设备的下一行)以及退格(将光标向左移动一个位置)。遗憾的是,不同的控制字符在不同的输出设备上执行不同的操作。在这些输出设备之间存在的统一标准很少。要想了解某个控制字符是如何影响某个特定设备的,需要参阅它的用户手册。

第 2 组的 32 个 ASCII 字符代码由各种标点符号、特殊字符以及数字组成。在该组中最值得注意的字符包括空格符(ASCII 码为 \$20)和数字(ASCII 码为 \$30~\$39)。

第 3 组的 32 个字符包含所有的大写字母字符。字符 A~Z 的 ASCII 码位于 \$41~\$5A(65~90)之间。由于只有 26 个不同的字母字符,所以余下 6 个代码就用于表示特殊符号。

第 4 组,即最后一组 32 个 ASCII 码表示小写字母字符、5 个特殊符号以及一个控制符(DELETE)。注意,小写字母字符使用 \$61~\$7A 之间的 ASCII 码。如果将大写和小写字母字符转换成二进制,您就会注意到大写字母字符的 ASCII 码与它们所对应的小写字母恰好有一位不同。例如,考虑图 2-26 中给出的 E 和 e 的字符代码。

¹¹ 在历史上,回车指的是打印机上面所用的纸架(paper carriage)。回车操作就是将纸架整个都移动到右边,使得要打印的下一个字符出现在纸的左边。

	7	6	5	4	3	2	1	0
E	0	1	0	0	0	1	0	1

	7	6	5	4	3	2	1	0
e	0	1	1	0	0	1	0	1

图 2-26 E 与 e 的 ASCII 码

这两个代码唯一不同的位置就是第 5 位。大写字符的第 5 位总是 0；而小写字符的第 5 位总是 1。这样就可以利用这一事实快速实现大小写之间的转换。如果有一个大写字符，可以通过把第 5 位设置为 1 将它强制转换为小写字符。如果有一个小写字符，并且想将它强制转换成大写形式，那么可以通过将第 5 位设置为 0 实现。因此，只需要通过转换第 5 位就可以实现字符的大小写转换了。

事实上，如表 2-8 所示，第 5 位和第 6 位就决定了字符处于 4 组 ASCII 字符集中的哪一组。

表 2-8 ASCII 组

第 6 位	第 5 位	组
0	0	控制字符
0	1	数字字符和标点符号
1	0	大写字符和特殊字符
1	1	小写字符和特殊字符

例如，可以通过把第 5 位和第 6 位都设置为 0，将任一大写或者小写(或者相应的特殊)字符转换成等效的控制字符。

考虑表 2-9 中给出的数字字符的 ASCII 码。

表 2-9 数字字符的 ASCII 码

字 符	十 进 制	十 六 进 制
0	48	\$30
1	49	\$31
2	50	\$32
3	51	\$33
4	52	\$34
5	53	\$35
6	54	\$36
7	55	\$37
8	56	\$38
9	57	\$39

这些 ASCII 码的十进制表示并没有很大的启发性。但是, 这些 ASCII 码的十六进制表示却反映了一些非常重要的问题: ASCII 码的低位半字节与它所表示的数字的二进制表示相同。只要把某个数字字符的高位半字节去掉(也就是说, 设置为 0), 就可以将该字符代码转换为与其对应的二进制表示。相反, 只要把高位半字节设置成 3 就可以将 0~9 内的某个二进制数值转换为它的 ASCII 码。注意, 这里可以使用逻辑与操作将高位位强制设为 0; 同样, 可以使用逻辑或操作将高位位强制设为 %0011(3)。

注意, 在数字字符串中, 只去掉其中每一个数字的高位半字节不能将该数字串转换为它们等效的二进制表示。如果以这种方式来转换 123(\$31 \$32 \$33)就可以得到三个字节: \$010203; 而 123 对应的正确的数值应该为 \$7B。如果要将数字串转换为整数就比这更加复杂了; 上面所使用的转换方法只适用于单个的数字。

2.14.2 HLA 对 ASCII 字符提供的支持

虽然很容易将字符数值存储到字节变量中, 并且在程序中使用字符字面量时, 也很容易使用其相应的 ASCII 码, 但是这种痛苦是没必要的: HLA 为汇编语言程序中的字符变量和字面量提供了很好的支持。

HLA 中的字符字面常量可采用两种形式来表示: 由单引号括起来的单个字符, 或者在一个“#”符号后面跟上用于指定该字符的 ASCII 码的数字常量, 该数字常量处于范围 0~127 之间。这里有一些例子:

```
'A'    #65    # $41    # %0100_0001
```

注意, 这些例子都表示同一个字符“A”, 因为“A”的 ASCII 码为 65。

除了一种例外的情况以外, 在字面字符常量中, 单引号之间只能出现一个字符。这种例外就是单引号字符本身。如果想创建一个单引号字面常量, 那么可以通过在一行中放置四个单引号来实现(也就是说, 将单引号用单引号括起来):

```
''''
```

“#”操作符必须放在合法的 HLA 数字常量(正如上面例子当中所指出的, 可以是十进制、十六进制或者二进制)的前面。特别要注意的是, “#”符号不是通用的字符转换函数; 它不能放在寄存器或者变量名的前面, 只能放在常量前面。

按照常规, 应该总是使用字符字面常量的单引号形式来处理图形字符(也就是说, 那些可以打印或者可以显示的字符)。采用“#”符号的格式来处理控制字符(它们在打印时不可见或者会实现一些有趣的功能)或者源代码中出现的不能正确显示和打印的扩展 ASCII 字符。

注意程序中的字符字面常量与字符串字面常量之间的区别。字符串是由 0 个或者多个用双引号括起来的字符组成的序列; 而字符是用单引号括起来的。

认识到 ‘A’ ≠ “A” 尤其重要。字符常量 A 与包含单个字符 A 的字符串的内部表示完全不同。如果在 HLA 希望出现字符常量的地方使用包含一个单字符的字符串, HLA 就会报告出错。本书将在第 4 章专门介绍关于字符串和字符串常量的内容。

要在 HLA 程序中声明一个字符变量, 可以使用 char 数据类型。例如, 下面所示的声明展示了如何声明一个名为 UserInput 的变量:

```
static
    UserInput:    char;
```

该声明保留一个字节的存储空间，可以保存任一字符数值(包括 8 位的扩展 ASCII 字符)。也可以对字符变量进行初始化，如下所示：

```
static

    TheCharA:      char := 'A';
    ExtendedChar:  char := #128;
```

因为字符变量是一个 8 位的对象，所以可以使用一个 8 位寄存器来对它们进行操作。可以将字符变量移到 8 位寄存器中，也可以将 8 位寄存器的数值存储到字符变量中。

HLA 标准库提供了一些例程，使用这些例程可以完成字符的输入输出和处理；这些例程包括 `stdout.putc`、`stdout.putcSize`、`stdout.put`、`stdin.getc` 以及 `stdin.get`。

`stdout.putc` 例程使用了下面所示的调用序列：

```
stdout.putc( charvar );
```

该过程将一个作为字符传递给它的单字符输出到标准输出设备。该参数可以是任意的 `char` 常量、变量、字节变量或者寄存器¹²。

`stdout.putcSize` 例程对显示字符变量时的输出宽度提供了控制。该过程的调用序列为

```
stdout.putcSize( charvar, widthInt32, fillchar );
```

该例程使用至少等于 `widthInt32` 的输出位数输出指定的字符(参数 `c`)¹³。如果 `widthInt32` 的绝对值大于 1，那么 `stdout.putcSize` 将输出 `fillchar` 作为填充字符。如果 `widthInt32` 为正数，那么 `stdout.putcSize` 将在输出域中右对齐输出字符；如果 `width` 为负数，`stdout.putcSize` 将在输出域中左对齐输出字符。由于输出的字符通常是左对齐的，所以在该程序调用中，`widthInt32` 的值通常为负。空格符是常用的填充字符。

也可以使用 `stdout.put` 例程输出字符值。如果某个字符变量出现在其参数列表中，则 `stdout.put` 自动将其作为字符值输出，例如：

```
stdout.put( "Character c = ", c, " ", nl );
```

采用 `stdin.getc` 和 `stdin.get` 例程可以从标准输入设备中读入字符。`stdin.getc` 程序没有任何参数。它从标准输入缓冲区中读入单个字符并将该字符返回 AL 寄存器。然后就可以将该字符存往别处或者在 AL 寄存器中操作该字符。程序清单 2-10 中的程序从用户端读入一个单字符，如果为小写，就将其转换为大写形式，然后输出该字符。

12 如果参数被指定为字节变量或者字节大小的寄存器，那么 `stdout.putc` 例程将输出变量或者寄存器中给出的 ASCII 码所对应的字符。

13 只有在指定宽度为 0 的时候，`stdout.putcSize` 才使用更多的输出位数；这时例程使用的输出位数刚好为 1 位。

程序清单 2-10 字符输入示例

```

program charInputDemo;
#include( "stdlib.hhf" )
begin charInputDemo;

    stdout.put( "Enter a character: " );
    stdin.getc();
    if( al >= 'a' ) then

        if( al <= 'z' ) then

            and( $5f, al );
        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,
        "converted to uppercase, was '"
    );
    stdout.putc( al );
    stdout.put( "'", nl );

end charInputDemo;

```

还可以使用 `stdin.get` 例程从用户端读入字符变量。如果某个 `stdin.get` 参数是字符变量, 该例程将从用户端读入一个字符并将其值存入指定的变量。程序清单 2-11 使用 `stdin.get` 例程重写程序清单 2-10。

程序清单 2-11 `stdin.get` 字符输入示例

```

program charInputDemo2;
#include( "stdlib.hhf" )
static
    c:char;

begin charInputDemo2;

    stdout.put( "Enter a character: " );
    stdin.get(c);
    if( c >= 'a' ) then

        if( c <= 'z' ) then

            and( $5f, c );

        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,

```

```

        "converted to uppercase, was '",
        c,
        "'", nl
    );
end charInputDemo2;

```

回想第1章, HLA 标准库将输入内容存入缓冲区。不管何时利用 `stdin.getc` 或 `stdin.get` 从标准输入中获得一个字符, 库例程都会读入该缓冲区中下一个可用的字符; 如果缓冲区为空, 程序读入用户(输入的)新的一行文本并返回该行的首字母。若要保证在读入某个字符变量时, 程序从用户端读入新的一行文本, 在试图读入数据之前, 应当调用 `stdin.flushInput` 例程。这将刷新当前的输入缓冲区并强迫在下一次输入的时候读入新的一行文本(可能是一次 `stdin.getc` 或者 `stdin.get` 调用)。

行末存在问题。不同的操作系统以不同的方式处理输入的行末和输出的行末。从控制台设备来看, 按下 ENTER 键就意味着一行的结束; 但是, 当从文件读入数据时, 得到的行尾序列是换行或回车/换行对(在 Windows 下), 或者仅仅是换行(在 Linux/Mac OS X/FreeBSD 下)。为了解决该问题, HLA 的标准库提供了一个“行尾”函数。如果当前所有的输入字符都用完了, 该程序将 `true(1)` 返回 AL 寄存器中, 反之返回 `false(0)`。程序清单 2-12 的示例程序演示了 `stdin.eoln` 函数。

程序清单 2-12 使用 `stdin.eoln` 测试某行的末尾

```

program eolnDemo;
#include ("stdlib.hhf")
begin eolnDemo;

    stdout.put( "Enter a short line of text: " );
    stdin.flushInput();
    repeat

        stdin.getc();
        stdout.putc( al );
        stdout.put( "=$", al, nl );

    until( stdin.eoln() );
end eolnDemo;

```

HLA 语言和 HLA 标准库对字符对象提供了许多其他的过程以及附加的支持。本书的第4章和第11章以及 HLA 参考文档介绍了如何使用这些特性。

2.15 Unicode 字符集

尽管 ASCII 字符集毫无疑问是计算机上最流行的字符表示, 但它不是唯一的格式。例如, 在许多大型机和小型机系列中, IBM 使用 EBCDIC 码。因为 EBCDIC 主要出现在 IBM 的大型计算机上, 而在个人计算机系统中很少碰到, 本书不考虑该字符集。另一种字符表示在小型计算机系统中变得流行起来, 这就是 Unicode 字符集。Unicode 克服了 ASCII 码的两个最大限制: 对字符

空间的限制(最多只能有128/256个8位字符)和缺少国际字符。

Unicode 使用 16 位的字来表示单个字符。因此,它支持 65536 个不同的字符码。与使用 8 位字节可表示的 256 个字符码相比,这显然是一个极大的进步。Unicode 与 ASCII 是向上兼容的。具体来说,如果一个 Unicode 字符的高 9 位含 0,则其低 7 位表示的字符和用相同字符编码表示的 ASCII 字符相同。如果高 9 位含有非零值,该字符表示其他的含义。如果想知道为什么需要这么多不同的字符码,可以注意一下,一些亚洲字符集含有 4096 个字符(至少它们的 Unicode 子集是这样)。

除了简单提到 Unicode 的地方,本书将一直使用 ASCII 字符集。不过最终,本书还是必须撤下有关 ASCII 的讨论而转向讨论 Unicode,因为许多新的操作系统在内部使用 Unicode (必要时转换为 ASCII)。遗憾的是,许多字符串算法对于 Unicode 没有 ASCII 那么方便(特别是字符集函数),因此本书尽可能坚持使用 ASCII 码。

2.16 更多信息

本书完整的电子版(可从 <http://webster.cs.ucr.edu/>或 <http://artofasm.com/>站点上下载)含有一些额外的数据表示信息,可能对您有用。若要了解数据表示的详细内容,可以参考笔者编写的 *Write Great Code, Volume 1*(No Starch Press,2004)一书,或者其他有关数据结构和算法方面的书。

第 3 章

存储器的访问与结构



第1章和第2章已经讲述了在汇编语言程序中如何声明和访问简单的变量。在本章中，您将会看到 80x86 存储器访问的完整过程；可以学会如何有效地组织变量声明来加速对其数据的访问。除此之外，本章还将阐述关于 80x86 中栈的相关知识以及如何处理栈中的数据；最后，本章将讲述动态存储器分配和堆(heap)的相关知识。

本章将讨论几个重要的概念，包括：

- 80x86 存储器寻址方式
- 变址寻址方式和比例变址寻址方式
- 存储器的结构
- 程序对存储器的分配
- 数据类型强制转换
- 80x86 栈
- 动态存储器分配

本章将介绍如何有效地使用计算机的存储器资源。

3.1 80x86 的寻址方式

80x86 处理器提供了许多不同的存储器访问方式。到目前为止，您只看到过一种简单的变量访问方式，也就是所谓的位移(displacement-only)寻址方式。在本节中，您将看到程序使用 80x86 存储器寻址方式(memory addressing mode)访问存储器的其他一些方法。80x86 存储器寻址方式提供了灵活的存储器访问方法，很容易访问变量、数组、记录、指针以及其他一些复杂的数据类型。掌握 80x86 的寻址方式是掌握 80x86 汇编语言的第一步。

Intel 在设计最初的 8086 处理器时, 为它提供了一套灵活的(虽然是有限的)存储器寻址方式。后来推出 80386 微处理器时, 在保留所有原有寻址方式的基础上又增加了几种新的寻址方式。然而, 在 Windows、Mac OS X、FreeBSD 和 Linux 这样的 32 位操作系统中, 这些早期的寻址方式就并不十分有用了; 事实上, HLA 甚至都不支持这些旧的 16 位寻址方式。幸运的是, 旧的寻址方式能够做到的, 新的寻址方式也能做到(事实上, 甚至可以做得更好)。因此, 如果您要为现在的高性能操作系统编写代码, 就不必再去学习那些旧的寻址方式了。但必须记住, 若想在 MS-DOS 或一些其他的 16 位操作系统上工作, 就必须学习这些旧的方式(<http://webster.cs.ucr.edu/>上提供了本书针对 16 位操作系统改写的版本)。

3.1.1 80x86 寄存器寻址方式

大多数的 80x86 指令都可以操作 80x86 的通用寄存器组。只要将寄存器名指定为指令的操作数, 就可以访问该寄存器的内容了。下面, 我们来考虑一下 80x86 中的 `mov` 指令:

```
mov( source, destination );
```

这条指令将数据从 *source* 操作数复制到 *destination* 操作数中。对于这条指令来说, 无论是 8 位、16 位还是 32 位寄存器都可以作为其有效的操作数。唯一的限制是两个操作数的数据宽度必须相同。下面我们来看一些 80x86 `mov` 指令:

```
mov( bx, ax );           // Copies the value from bx into ax
mov( al, dl );           // Copies the value from al into dl
mov( edx, esi );         // Copies the value from edx into esi
mov( bp, sp );           // Copies the value from bp into sp
mov( cl, dh );           // Copies the value from cl into dh
mov( ax, ax );           // Yes, this is legal!
```

寄存器是保存变量的最佳地方。使用寄存器作为操作数的指令比那些访问存储器的指令更短而且更快。当然了, 大多数计算都至少需要一个寄存器操作数, 所以寄存器寻址方式在 80x86 的汇编代码中是相当普遍的。

3.1.2 80x86 的 32 位存储器寻址方式

80x86 提供了几百种不同的存储器访问方法。也许这显得多了点, 但幸运的是, 在这些寻址方式中, 大多数都只是其他方式的简单变形, 所以很容易学会。而且您也应该学会它们! 对于好的汇编语言程序来说, 关键就在于如何正确使用存储器寻址方式。

80x86 系列处理器所提供的寻址方式包括位移寻址、基址寻址、间接基址寻址、基址变址寻址和间接基址变址寻址。这 5 种寻址方式的变形就构成了 80x86 中所有的寻址方式。将几百种寻址方式最后仅归为 5 种, 看来情况并不像想象的那么可怕!

1. 位移寻址方式

最普遍的也是所有寻址方式中最简单的一种便是位移寻址方式, 或称为直接(`direct`)寻址方式。位移寻址方式包括一个指定目标地址的 32 位常量。假设变量 `j` 是存储在 `$8088` 处的一个 `int8` 变量, 指令 `mov(j, al);` 的意义是将存储单元 `$8088` 中的字节副本加载到寄存器 `AL` 中。同样, 假设 `int8` 变

量 k 位于存储单元 $\$1234$ 中, 那么指令 `mov(dl,k);` 的意义便是将寄存器 DL 中的数据存储在 $\$1234$ 中(见图 3-1)。

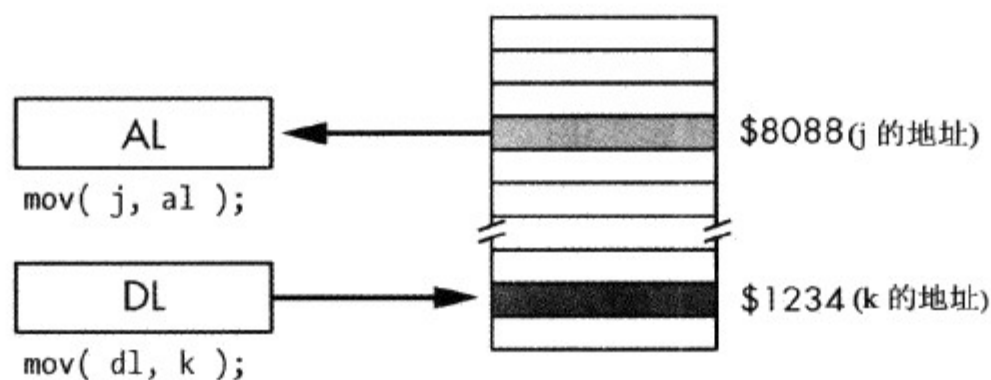


图 3-1 位移(直接)寻址方式

对于访问简单的标量变量来说, 位移寻址方式是很适合的。之所以将这种寻址方式命名为位移寻址方式, 是因为 `mov` 指令的一个操作数是 32 位的常量(位移地址)。在 80x86 系列处理器中, 这个位移地址是相对于存储器起始地址(也就是地址 0 处)的偏移量。本章中的例子将经常访问存储器中的字节。但是别忘了, 在 80x86 系列处理器中, 也可以通过指定第一个字节的地址来访问字和双字(见图 3-2)。

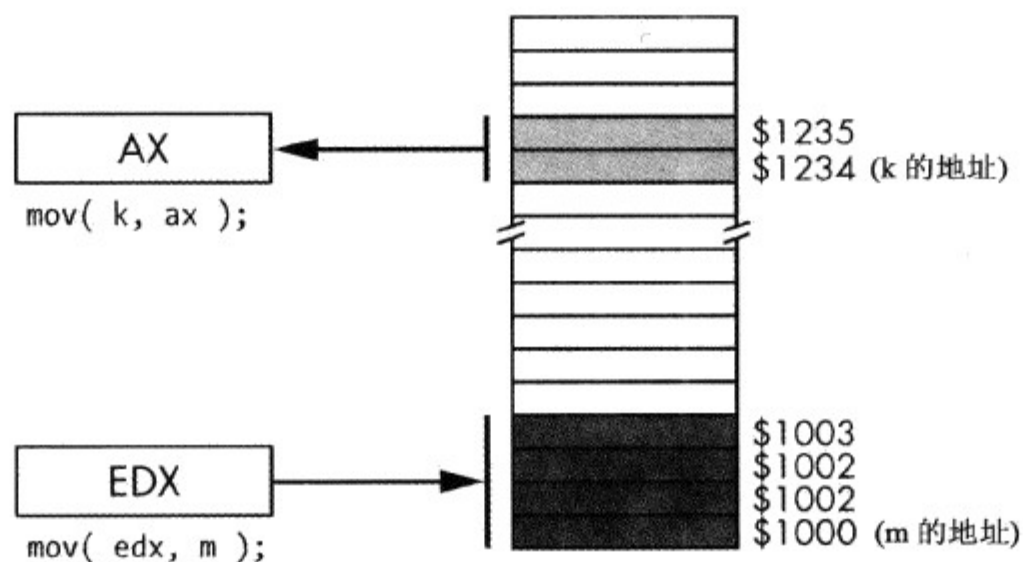


图 3-2 采用位移寻址方式访问字或者双字

2. 寄存器间接寻址方式

在 80x86 系列处理器中, 使用寄存器间接寻址方式可以间接地通过一个寄存器来访问存储器。术语“间接(indirect)”意思就是操作数并不是实际的地址, 操作数的值指定要使用的存储器地址。在寄存器间接寻址方式的例子中, 寄存器中所保存的值是要访问的存储器地址。例如, 指令 `mov(eax,[ebx]);` 告诉 CPU 将 EAX 的值存储在 EBX 值所指定的存储单元中(EBX 外的方括弧告诉 HLA 使用寄存器间接寻址方式)。

80x86 系列处理器总共有 8 种这样的寻址方式, 下面的指令是这 8 种形式的例子:

```
mov([eax], al);
mov([ebx], al);
mov([ecx], al);
mov([edx], al);
mov([edi], al);
```

```
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

这 8 种寻址方式用方括弧括起来的寄存器中的偏移量给出了存储器的地址(分别为 EAX、EBX、ECX、EDX、EDI、ESI、EBP 或者 ESP)。

值得注意的是, 寄存器间接寻址方式需要使用 32 位的寄存器。当使用间接寻址方式时, 不能使用 16 位或者 8 位¹的寄存器。从技术上说, 可以给 32 位的寄存器加载任意一个数值, 并且使用寄存器间接寻址方式间接地访问该地址:

```
mov( $1234_5678, ebx );
mov( [ebx], al );           // Attempts to access location $1234_5678.
```

遗憾的是(或者说幸运的是, 这取决于您如何看), 这可能会引发操作系统的保护故障, 因为并不是所有存储器地址都可以随意访问。有更好的方法可以将某个对象的地址加载到寄存器当中, 稍后就会讨论具体做法。

寄存器间接寻址方式有很多用途。可以用它访问指针所指的数据, 也可以用它遍历数组中的数据。一般来说, 当程序在运行时, 无论何时需要修改变量地址都可以使用它。

寄存器间接寻址方式给出了匿名(anonymous)变量的一个例子。当使用寄存器间接寻址方式的时候, 通常要用由数字表示的存储器地址来引用变量的值(例如, 加载到寄存器中的值), 而不能用该变量的名称。因此称为“匿名变量”。

HLA 提供了一种简单的操作符来获取静态变量的地址, 并将这个地址放到一个 32 位寄存器当中。这就是“&”(取址)操作符(注意, 这与 C/C++中使用的“取址”操作符相同)。下面的例子将变量 j 的地址存在寄存器 EBX 中, 然后采用间接寻址方式将寄存器 EAX 的当前值保存到 j 中:

```
mov( &j, ebx );           // Load address of j into ebx.
mov( eax, [ebx] );        // Store eax into j.
```

当然, 将 EAX 的值直接存入变量 j 中比使用两条指令间接完成这一任务简单。然而, 您应该很容易想象到这样一个代码序列: 在执行指令 `mov(eax, [ebx]);` 之前, 将不同的地址加载到 EBX 中, 这样就可以根据执行路径的不同把 EAX 的值存储到不同的存储单元中。

警告:

与 C/C++中不同, “&”(取址)操作符不是一个通用的操作符。它只能用于静态变量², 不能用于一般的地址表达式或者其他类型的变量。在 3.13 节, 您将学到“加载有效地址”指令, 它为获取存储器中某些变量的地址提供了一种通用的方法。

3. 变址寻址方式

变址寻址方式的语法如下所示:

¹ 事实上, 像前面所提到的一样, 80x86 系列处理器支持寻址方式使用某些特定的 16 位寄存器。然而, HLA 却不支持这些方式, 它们在 32 位的操作系统中没用。

² 注意, 这里的术语“静态”指的是静态的、只读的或者是存储对象。

```

mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );

```

VarName 是程序中某些变量的名称。

变址寻址方式将变量的地址与方括弧中的 32 位寄存器的值相加，从而得到有效地址³。它们的和是指令访问的存储器的实际地址。所以如果 *VarName* 位于存储器的地址 \$1100 中，寄存器 EBX 中的值为 8，那么指令 `mov(VarName[ebx], al);` 将会把地址 \$1108 中的字节加载到寄存器 AL 中(见图 3-3)。

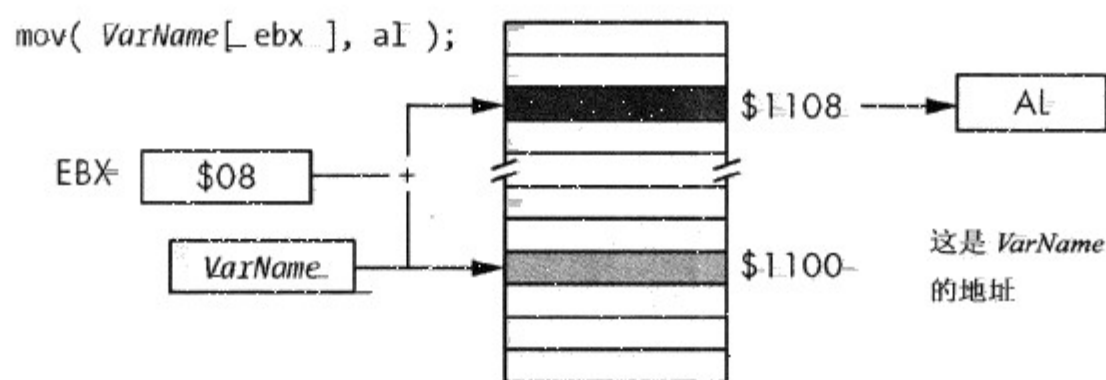


图 3-3 变址寻址方式

变址寻址方式对于访问数组元素来说相当方便。在第 4 章中，您将看到如何使用这种寻址方式来访问数组元素。

4. 变址寻址方式的变形

变址寻址方式有两种重要的语法变形。这两种方式都产生相同的基本机器指令，但是用法不同。

第一种变形使用下面的语法：

```

mov([ ebx + constant ], al );
mov([ ebx - constant ], al );

```

这些例子只用到了寄存器 EBX。但是，可以用其他任意一种 32 位通用寄存器来取代 EBX。这种寻址方式将 EBX 中的值与指定的 *constant* 值相加，或者从 EBX 中减去指定的 *constant* 值，从而得到它的有效地址(见图 3-4 和图 3-5)。

³ 有效地址是所有地址计算都完成以后，指令将访问的存储器的最终地址。

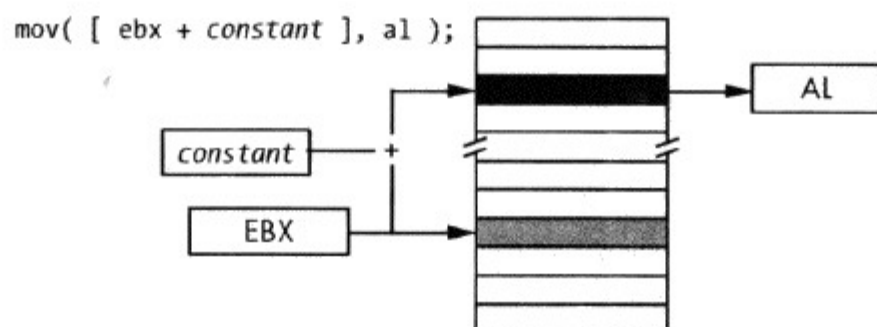


图 3-4 将寄存器值与常量相加的变址寻址方式

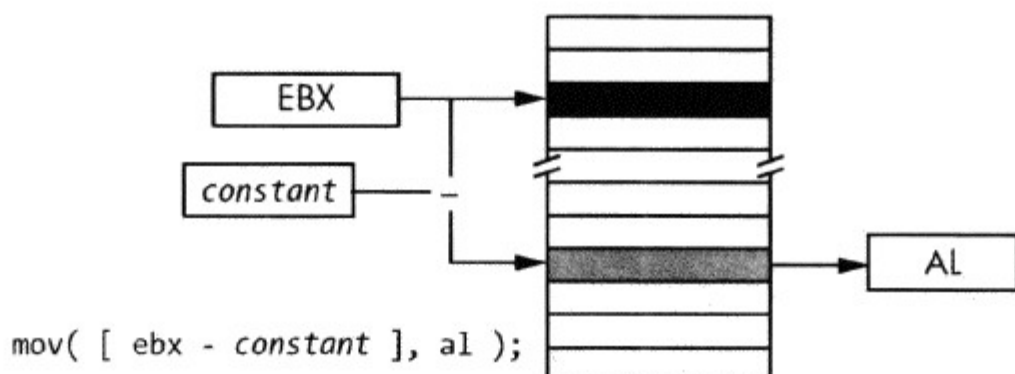


图 3-5 从寄存器值中减去常量值的变址寻址方式

如果一个 32 位寄存器包含一个多字节变量的基址,而您想访问这个地址前面或者后面的单元,那么这种寻址方式的特殊变形是很有用处的。它的一种重要用法就是当有指针指向记录(或结构)数据时,访问其中的某个字段。这种寻址方式对于访问过程中的自动(局部)变量也是非常有用的(详见第 5 章的内容)。

变址寻址方式的第二种变形事实上就是前两种形式的组合。它的语法结构如下所示:

```
mov( VarName[ ebx + constant ], al );
mov( VarName[ ebx - constant ], al );
```

与前面的例子相同,这个例子中只使用了寄存器 EBX。但是在这两个例子当中,可以用任意的 32 位通用寄存器取代 EBX。在汇编语言程序中,要访问记录(结构)数组中的元素时,这种形式的变址寻址方式就十分有用了(对于这一点,第 4 章将会详细阐述)。

这些指令先将 *VarName* 的地址值加上或者减去 *constant* 的值,然后再加上寄存器 EBX 的值便可以得到有效地址值。值得注意的是,计算 *VarName* 的地址与 *constant* 的和或者差的是 HLA,而不是 CPU。上面语句的机器指令中包含一个常量,它在运行时将和寄存器 EBX 中的值相加。因为 HLA 用常量取代 *VarName*,它可将如下形式的指令

```
mov( VarName[ ebx + constant ], al );
```

简化为下面的形式:

```
mov( constant1[ ebx + constant2 ], al );
```

由于这些寻址方式工作的方式,它们在语义上等效于下面的指令:

```
mov( [ ebx + (constant1 + constant2) ], al );
```

编译时, HLA 将把两个常量相加, 实际上生成了下面这条指令:

```
mov( [ ebx + constant_sum ], al );
```

当然, 减法的情况也没有什么特别的, 只要将 32 位的常量取补码, 然后加上这个补码值(而不是去减原码值)。

5. 比例变址寻址方式

比例变址寻址方式与变址寻址方式基本相似, 只存在以下两点不同: ①比例变址寻址方式允许组合两个寄存器的值并加上一个偏移量; ②比例变址寻址方式允许变址寄存器的值乘以比例因子 1、2、4 或者 8。这些寻址方式的语法如下所示:

```
VarName[ IndexReg32*scale ]
VarName[ IndexReg32*scale + displacement ]
VarName[ IndexReg32*scale - displacement ]

[ BaseReg32 + IndexReg32*scale ]
[ BaseReg32 + IndexReg32*scale + displacement ]
[ BaseReg32 + IndexReg32*scale - displacement ]

VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

在上面这些例子当中, *BaseReg32* 代表任意的 32 位通用寄存器; *IndexReg32* 代表除了 ESP 之外的任意 32 位通用寄存器。scale 必须是常数 1、2、4 或者 8 中的一个。

比例变址寻址方式与变址寻址方式主要的差别在于它包含 *IndexReg32*scale* 的计算。这些方式通过加上这一新寄存器的值与指定比例因子之积来得到有效地址(参见图 3-6 所示的例子, 其中 EBX 为基址寄存器, ESI 为变址寄存器)。

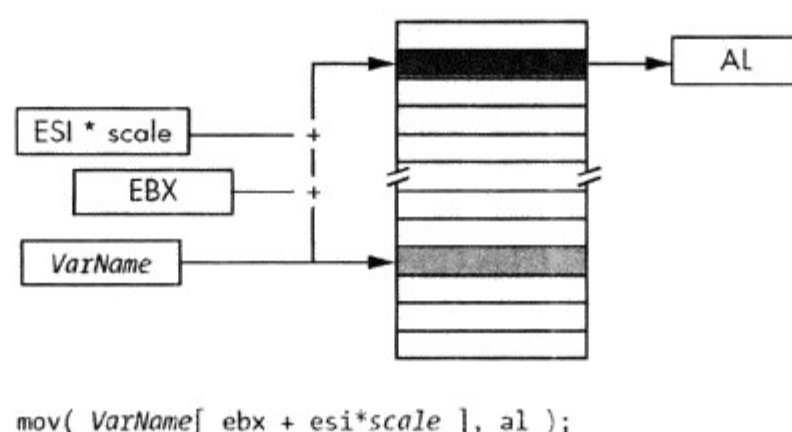


图 3-6 比例变址寻址方式

在图 3-6 中, 假设寄存器 EBX 包含的值为\$100, 寄存器 ESI 包含的值为\$20, *VarName* 位于存储器的基址\$2000 中。那么下面这条指令将地址\$2184(即\$2000+\$100+\$20*4+4)处的字节移到寄存器 AL 当中。

```
mov( VarName[ ebx + esi*4+4 ], al );
```

比例变址寻址方式非常适用于访问大小为 2、4 或者 8 字节的数组元素。当具有指向数组的起始地址的指针时,这种寻址方式也是很有用的。

6. 寻址方式小结

无论您是否相信,到此您已经学会几百种寻址方式了!如果您正在纳闷这些方式究竟从何而来,那么请考虑这样一个事实:寄存器间接寻址方式并不是 1 种寻址方式,而是 8 种不同的寻址方式(涉及 8 个不同的寄存器)。即由寄存器、常量长度以及其他因素的组合乘以系统中可能的寻址方式数目得到。您只需要记住大约 24 种寻址方式,而现在您已经学到了。实际上,任何一个给定的程序真正用到的寻址方式一般不到一半(很多寻址方式也许您永远都用不上)。所以要学会所有的寻址方式比看起来容易得多。

3.2 运行时存储器的结构

像 Mac OS X、FreeBSD、Linux、Windows 这样的操作系统都将不同类型的数据放到存储器的不同段(section 或 segment)中。虽然可以通过运行连接器并指定不同的参数重新配置存储器,但是默认情况下,Windows 使用图 3-7 所示的组织方式将 HLA 程序加载到存储器当中(Linux、Mac OS X 和 FreeBSD 与之类似,但对于某些段会重新分配)。

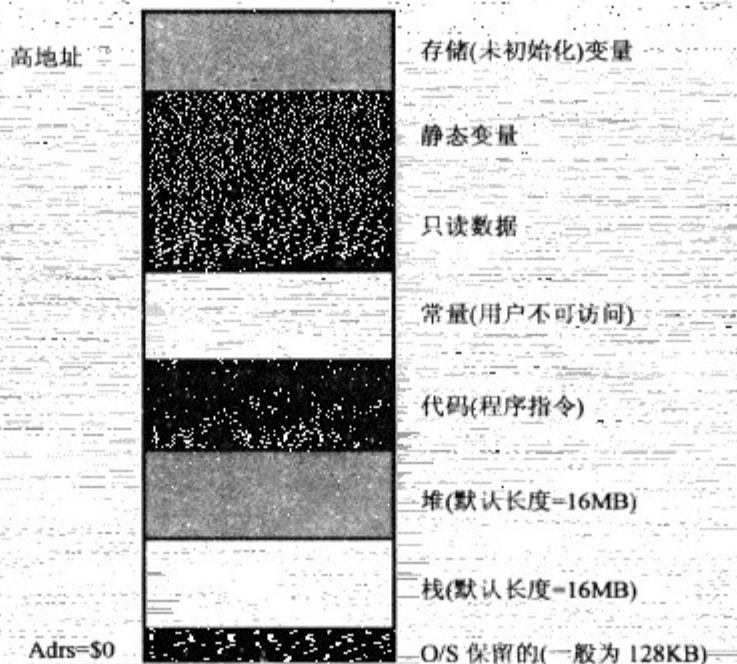


图 3-7 HLA 运行时典型的存储器结构

操作系统保留了最低的存储器地址空间。一般来说,用户的应用程序不能访问这些低地址空间中所保存的数据(或执行指令)。原因之一便是操作系统保留这些空间是为了有助于捕获 NULL 指针引用。如果想访问地址为 0 的存储单元,操作系统会产生一个“通用保护故障”,意思是您访问了存储器中不包含有效数据的空间。因为程序员通常会将指针初始化为 NULL(0)来表明指针没有指向任意位置。访问 0 地址通常意味着程序员出了差错,还没有将指针初始化为一个合法(非空)值。

存储器中剩下的6个区域保存了与程序相关的不同类型的数据，包括栈段、堆段、代码段、只读段、静态段、存储段。每种存储器段都对应一种可以在 HLA 程序中创建的数据类型。下面将详细讨论每一个段。

3.2.1 代码段

代码段包含出现在 HLA 程序中的机器指令。HLA 将您所写的机器指令都翻译为一个或者多个字节值的序列。当程序执行时，CPU 将这些字节值解释为机器指令。

在默认情况下，当 HLA 连接程序时，它会告诉系统您的程序可以在代码段执行指令，并且可以从代码段读取数据。特别要注意的是：不能向代码段写数据。如果试图将数据存入代码段，操作系统会产生一个通用保护故障。

机器指令其实就是一些数据字节而已。从理论上来说，可以编写一个程序将数据存储到存储器中，然后将控制转移给刚刚写入的数据，从而完成一个在执行时可修改自身的程序。这样一来，便产生了人工智能(Artificial Intelligence)程序想要通过自改写来生成期望结果的理想模型。在现实当中，其效果并没有想象的那么美好。一般来说，自修改的程序很难调试，因为指令在后台不停地改变。因为在大多数现代操作系统中都很难编写自修改的程序，所以本书中不会展开讨论这方面的内容。

HLA 将和机器代码有关的数据自动存储到代码段中。除了机器指令以外，还可以采用下面的伪操作码⁴将数据存储到代码段中：

byte	int8
word	int16
dword	int32
uns8	boolean
uns16	char
uns32	

下面的 byte 语句展示了这些伪操作码的语法：

```
byte comma_separated_list_of_byte_constants;
```

下面给出了一些例子：

boolean	true;
char	'A';
byte	0, 1, 2;
byte	"Hello", 0;
word	0, 2;
int8	-5;
uns32	356789, 0;

4. 这不是一个完整的列表。HLA 通常允许将任何标量数据类型名称作为语句存储到代码段中。在第 4 章中，您将学到更多可用的数据类型。

如果在伪操作码后的值列表中出现了多个值,HLA 将这些值依次发送到代码流中。因此,上面的第一个 `byte` 语句发送了 3 个字节到指令流中,分别是 0、1、2。如果在 `byte` 语句中出现一个字符串,HLA 将把字符串中的每个字符都作为一个字节来发送。因此,上面第二个 `byte` 语句发送了 6 个字节:字符 H、e、l、l、o 以及一个 0 字节。

切记,CPU 将把发送到代码流中的数据看作机器指令,除非进行特殊的处理不允许数据执行。例如,如果编写了下面的代码:

```
mov( 0, ax );
byte 0, 1, 2, 3;
add( bx, cx );
```

在执行了 `mov` 语句之后,程序将把 0、1、2、3 这几个字节作为机器指令运行。除非您了解特定指令序列的机器码,将这些值写入代码中一般会使程序崩溃。通常当在程序中放置这样的数据时,都应该执行一些代码来传递对这些数据的控制。

3.2.2 静态段

静态段一般是用于声明变量的。虽然从语法结构上说,静态段是程序或者过程的一部分,但是切记 HLA 会把所有的静态变量都移到存储器的静态段中。因此,HLA 不会将在静态段中声明的变量掺杂到代码段的过程之间。

除了声明静态变量以外,也可以将数据列表保存到静态声明段中。可以使用与将数据嵌入到代码段中的相同技术将数据嵌入静态段中,即使用伪操作码 `byte`、`word`、`dword`、`uns32` 等。考虑下面的例子:

```
static =
  b:  byte := 0;
      byte 1, 2, 3;

  u:  uns32 := 1;
      uns32 5, 2, 10;

  c:  char;
      char 'a', 'b', 'c', 'd', 'e', 'f';

  bn: boolean;
      boolean true;
```

HLA 使用伪操作码向静态存储器段写入的数据被写到前面变量之后的段中。例如,字节值 1、2、3 就会被写到变量 `b` 的字节值 0 之后的静态段中。由于没有标记来标识相关的值,所以在这一程序中,您不能对这些数值直接进行访问。可以使用变址寻址方式来访问这些额外的数值(第 4 章给出了相应的例子)。

在上面的例子中,请注意变量 `c` 和 `bn` 是没有(显式的)初始值的。但是若不给它们指定初始值,HLA 就会将静态段中的变量都初始化为 0,因此 HLA 将 `c` 初始化为 NUL 字符(ASCII 码为 0)。同样,HLA 将 `bn` 初始化为 `false`。应该特别注意,静态段中声明的变量总会消耗存储空间,即使它们没有初始值。

3.2.3 只读数据段

只读数据段中可以存放常量、表以及其他一些在程序执行中不可更改的数据。可以通过在只读段进行声明创建只读对象。只读段与静态段非常相似，但存在以下3点主要的差异：

- 只读段以保留字 `readonly` 开头，而静态段以 `static` 开头。
- 只读段中的声明一般都有初始值设定项。
- 在程序执行过程中，系统不允许将数据存储到只读对象中。

例如：

```
readonly
    pi:      real32 := 3.14159;
    e:      real32 := 2.71;
    MaxU16:  uns16  := 65_535;
    MaxI16:  int16   := 32_767;
```

由于您不能在程序控制下初始化这些值，因此所有的只读对象声明都必须带初始值设定项⁵。不管出于什么样的动机和目的，都可以把只读对象看作常量。但是这些常量是要消耗存储空间的，而且除了不能向只读对象写数据外，它们和静态变量是一样的。正因为它们的行为类似于静态变量，所以不能在每个可以使用常量的地方使用只读对象；特别是，只读对象属于存储器对象，所以指令的操作数不能是只读对象或者其他一些存储器对象。

像静态段一样，可以使用 `byte`、`word`、`dword` 等数据声明将数据值嵌入只读段当中，例如：

```
readonly
    roArray: byte := 0;
           byte 1, 2, 3, 4, 5;
    qwVal:  qword := 1;
           qword 0;
```

3.2.4 存储段

只读段要求声明的对象都要初始化。而静态段则允许有选择地进行初始化(或者不对它们进行初始化，在这种情况下，它们的默认初始值为0)。存储段完成初始化：可以用它来声明当程序运行时总是未初始化的变量。存储段以 `storage` 保留字开头，包含没有初始值设定项的变量声明。这里有一个例子：

```
storage
    UninitUns32: uns32;
    i:          int32;
    character:  char;
    b:          byte;
```

当 Linux、FreeBSD、Mac OS X 和 Windows 将程序加载到存储器中时，它们会把所有的存储对象都初始化为0。但是，依赖于这种隐式的初始化也许并不是什么好主意。如果需要将对象初始化为0，可以在静态段中声明它并显式地将它的值设置为0。

⁵ 第5章将介绍一种例外情况。

对于程序来说,存储段中声明的变量在执行文件中占用的磁盘空间会更少一些。这是因为 HLA 要将只读对象和静态对象的初始值都写到执行文件当中,但是对于在存储段中声明的未初始化的变量却使用紧凑格式保存,不过这种行为取决于操作系统和对象模块的格式。

因为存储段不允许有已初始化的值,所以不能使用 `byte`、`word`、`dword` 等伪操作码将未标记的变量放到存储段中。

3.2.5 @nostorage 属性

`@nostorage` 属性允许在静态数据声明段(即静态段、只读段和存储段)中声明变量,但并不为这些变量分配实际的存储空间。`@nostorage` 选项告诉 HLA 将声明段中的当前地址分配给变量,但并不为这个对象分配任何的存储空间。该变量将与出现在变量声明段中的下一个对象共享相同的存储器地址。下面给出了 `@nostorage` 选项的语法:

```
variableName: varType; @nostorage;
```

注意,在类型名之后紧接着的是 `@nostorage;`,而不是某个初始值或者分号。在只读段中使用 `@nostorage` 选项的例子如下面的代码所示:

```
readonly
    abcd: dword; @nostorage;
        byte= 'a', 'b', 'c', 'd';
```

在这个例子中, `abcd` 属于双字类型,其低位字节为 97(“a”)、1 号字节为 98(“b”)、2 号字节为 99(“c”)、高位字节为 100(“d”)。HLA 不会为变量 `abcd` 保留存储空间,因此 HLA 将内存中接下来的 4 个字节(由 `byte` 指令分配)和 `abcd` 相关联。

注意, `@nostorage` 属性只在静态段、存储段和只读段(所谓的静态变量声明段)中有效。HLA 不允许在 `var` 段中使用该属性,详见 3.2.6 节。

3.2.6 var 段

HLA 提供了另一种变量声明段,即 `var` 段,可用它创建自动(`automatic`)变量。程序单元(即主程序或过程)无论何时开始运行,程序都将为自动变量分配存储空间,而当该程序单元返回其调用程序时,程序又会为自动变量解除分配。当然,在主程序中声明的任何自动变量和所有的静态对象、只读对象、存储器对象拥有相同的生存期⁶。因此,在主程序中, `var` 段的自动分配属性没有发挥作用。一般来说,应该只在过程中使用自动变量对象(详细信息参见第 5 章)。通常,HLA 允许它们在主程序的声明段中声明。

由于在 `var` 段中声明的变量是在运行时创建的,HLA 不允许在该段中对所声明的变量进行初始化。因此, `var` 段的语法和存储段的语法几乎相同;两者语法上唯一的区别在于 `var` 段使用 `var` 保留字,而不是 `storage` 保留字⁷。如下例所示:

```
var
    vInt:    int32;
    vChar:   char;
```

6. 变量的生存期从为该变量分配内存开始直到释放内存。

7. 实际上,还有其他一些较小的差异,但是本文不涉及这些差异。详细信息请参见 HLA 参考手册。

HLA 将 var 段中声明的变量分配到栈存储器段。HLA 不会为 var 对象分配固定的地址；实际上，它在和当前程序单元有关的活动记录中分配这些变量。本书第 5 章将详细讨论活动记录，对现在来说，关键在于认识到 HLA 程序使用 EBP 寄存器作为当前活动记录的指针。因此，无论何时访问 var 对象，HLA 将自动把该变量名换为 “[EBP ± displacement]”。位移量是该对象在活动记录中的偏移量，这意味着无法对 var 对象采用全比例变址寻址方式(一个基址寄存器加上一个比例变址寄存器)，因为 var 对象已经使用 EBP 寄存器作为基址寄存器。尽管通常不会直接使用这两个寄存器寻址方式，var 段的这一限制也让人有充分的理由尽量不在主程序中使用该段。

3.2.7 程序中声明段的结构

静态段、只读段、存储段和 var 段可能在 program 头和主程序的 begin 语句之间出现 0 或多次。在这两点之间，声明段可以任何顺序出现，如下例所示：

```
program demoDeclarations;

static
    i_static:      int32;

var
    i_auto:        int32;

storage
    i_uninit:      int32;

readonly
    i_readonly:    int32 :=5;

static
    j:             uns32;

var
    k:             char;

readonly
    i2:            uns8 :=9;

storage
    c:             char;

storage
    d:             dword;

begin demoDeclarations;
    << Code goes here. >>
end demoDeclarations;
```

除了表明这些段可以按任意顺序出现之外，该段代码也表明一个给定的声明段可能在程序中出现多次。当在程序的声明段中出现多个同类型的声明段时(比如上面的 3 个存储段)，HLA 会将它们合并为一组。

3.3 HLA 如何为变量分配内存

如前所述, 80x86 CPU 不处理诸如 `I`、`Profits` 和 `LineCnt` 等名称的变量。CPU 严格处理其能放置到地址总线上的数字地址, 如 `$1234_5678`、`$0400_1000` 和 `$8000_CC00` 等。另一方面, HLA 并不强迫通过变量对象的地址来引用它们(这一点很方便, 因为名称更容易记忆)。这虽然方便, 但是却模糊了真正发生的操作。本节将介绍 HLA 是如何将数字地址和变量联系起来的, 以便于读者理解那些无法看到的过程。

我们再看下图 3-7。可以看到, 不同的存储器段之间是彼此相连的。因此, 如果改变某个存储器段的大小, 会影响到所有后续存储器段的起始地址。例如, 如果在程序中添加一些附加的机器指令, 代码段加长了, 这会影响存储器中静态段的起始地址, 故而改变所有静态变量的地址。弄清楚变量的地址(而不是它们的名称)已经相当困难; 设想一下, 在程序中添加和删除机器指令时, 地址不停地变动, 情况会更糟糕! 幸运的是, 您不需要清楚这一切, HLA 会替您做记录。

HLA 将当前的位置计数器和三个静态声明段(静态段、只读段和存储段)中的每一段相关联。这些位置计数器初值为零, 无论何时在某一静态段中声明一个变量, HLA 会把该变量和该段位置计数器的当前值联系起来; 并且把所声明对象的长度和位置计数器的值相加。例如, 假定下面是程序中唯一的静态声明段:

```
static
    b      :byte;           // Location counter = 0, size = 1
    w      :word;           // Location counter = 1, size = 2
    d      :dword;          // Location counter = 3, size = 4
    q      :qword;          // Location counter = 7, size = 8
    l      :lword;          // Location counter = 15, size = 16
                        // Location counter is now 31.
```

当然, 上述变量的运行时地址并不是位置计数器的值。首先, HLA 将静态存储段的基址和每个位置计数器的值(称为位移量或者偏移量)相加。其次, 在与程序相链接的模块中可能存在其他静态对象(例如来自 HLA 标准库), 甚至在相同的源文件中可能还存在其他静态段, 连接器必须把静态段合并到一起。因此, 这些偏移量可能使存储器中变量的最终地址有一些小的出入。而且, 关键的一点在于: 在单独的静态声明段中声明的变量被 HLA 分配之后, 其存储单元是彼此相连的。也就是说, 对于上面的声明, 在存储器中, `w` 紧跟着 `b`, `d` 紧跟着 `w`, `q` 紧跟着 `d`, 依此类推。一般来说, 假定系统按上面的方式分配变量并不是好的编码习惯, 但是有时候这样做很方便。

注意, HLA 将在只读段、静态段和存储段中声明的存储器对象分配到存储器中完全不同的区域中。因此, 无法认为下列 3 个存储器对象的存储地址是相邻的(事实上, 很可能不是):

```
static
    b      :byte;
readonly
    w      :word := $1234;
storage
    d      :dword;
```

实际上, HLA 甚至无法保证在分离的静态段(或其他任意段)中声明的变量在存储器中是相邻

的,即使在声明的代码之间没有任何语句(例如,无法肯定下列声明中 b、w 和 d 在存储器中的位置是相邻的,但也无法否定):

```
static
    b      :byte;
static
    w      :word := $1234;
static
    d      :dword;
```

如果代码需要将这些变量分配到相邻的存储单元,就必须在同一个静态段中声明它们。

注意,HLA 处理在 var 段中声明的变量的方式和静态段中稍有不同。var 对象偏移量的分配将在第 5 章进行讨论。

3.4 HLA 对数据对齐的支持

为了编写出运行更快的程序,需要保证存储器中的数据对象能够很好地对齐。合理的对齐意味着对象的起始地址是某个长度的倍数。通常,如果对象的大小是 2 的幂,并且最长可以达到 16 个字节,该宽度就是对象的大小。对于大于 16 个字节的对象,将对象调整到 8 字节或者 16 字节的地址边界上,这就足够了。对于小于 16 个字节的对象,一般将该对象对齐到恰好比其大小大的下一个 2 次幂的地址上是合理的。访问没有对齐到合适地址的数据需要花费额外的时间;所以如果要保证程序运行得足够快,应当尽量依据数据对象的大小来对齐。

任何时候在相邻存储单元上分配大小不同的对象都会导致数据不对齐。例如,如果声明一个字节变量,它将消耗一个字节的存储区,在该声明段中声明的下一个变量的地址就是该字节对象的地址加上 1。如果该字节变量的地址恰好是一个偶数,则后面那个变量的地址就是奇数。如果后面那个变量是一个字对象或者双字对象,则其起始地址就不是最优的。在本小节中,我们将探讨保证依据变量大小来合理安排其起始地址的方法。

考虑下面的 HLA 变量声明:

```
static
    dw:      dword;
    b:       byte;
    w:       word;
    dw2:     dword;
    w2:      word;
    b2:      byte;
    dw3:     dword;
```

程序(运行于 Windows、Mac OS X、FreeBSD、Linux 和绝大多数的 32 位操作系统)中第一个静态声明会把其变量置于 4096 字节的偶数倍地址上。在静态声明中出现的首个变量都能保证被对齐到一个合理的地址上。对于每一个后继变量,静态段起始地址加上其前面所有变量大小的和就是其被分配的地址。因此,在上述例子中,假定 HLA 分配给变量的起始地址为 4096,HLA 将分配如下的地址:

		//	Start Adrs	Length
dw:	dword;	//	4096	4
b:	byte;	//	4100	1
w:	word;	//	4101	2
dw2:	dword;	//	4103	4
w2:	word;	//	4107	2
b2:	byte;	//	4109	1
dw3:	dword;	//	4110	4

除了第 1 个变量(被分配到 4K 边界上)和字节变量(对齐与否无关紧要)之外, 上述所有变量都没有对齐。w、w2 和 dw2 起始于奇地址, dw3 被分配到非 4 倍数的偶地址上。

保证变量合理对齐的一个简单方法是将所有的双字变量放到声明的最前面, 字变量其次, 最后是字节变量:

```
static
    dw:    dword;
    dw2:   dword;
    dw3:   dword;
    w:     word;
    w2:    word;
    b:     byte;
    b2:    byte;
```

这样将在存储器中产生如下的地址:

		//	Start Adrs	Length
dw:	dword;	//	4096	4
dw2:	dword;	//	4100	4
dw3:	dword;	//	4104	4
w:	word;	//	4108	2
w2:	word;	//	4110	2
b:	byte;	//	4112	1
b2:	byte;	//	4113	1

正如您所看到的, 变量都对齐到了合理的地址上。

遗憾的是, 很少能够如此安排变量。除了很多技术上的原因使得无法进行这样的安排之外, 一个很好的理由就是因为您无法按照逻辑功能来组织变量(也就是说, 您很可能不顾其大小而将相关的变量放置在一起)。

为了解决这个问题, HLA 提供了 align 指令。align 指令采用如下的语法:

```
align( integer_constant );
```

该整型常量必须是下列无符号整数值之一: 1、2、4、8 或 16。如果 HLA 在静态段遇到了 align 指令, 它将把紧接其后的变量对齐到指定对齐常量的偶数倍地址上。使用 align 指令, 重写前面的例子如下:

```
static
    align( 4 );
```

```

dw:    dword;
b:      byte;
align( 2 );
w:      word;
align( 4 );
dw2:    dword;
w2:      word;
b2:      byte;
align( 4 );
dw3:    dword;

```

align 指令的原理其实很简单。如果 HLA 求出当前地址(位置计数器的值)不是指定值的偶数倍,它将在前面的变量声明后边填入附加的字节,直到静态段中的当前地址为指定值的偶数倍为止。在快速访问数据的同时,这样做会使得程序稍稍加大(多了一些字节);假定采用这种方法,程序只增加了少量字节,这倒可能是一种不错的选择。

作为一个普遍规律,如果要获得最快的访问速度,必须使得对齐值等于所对齐对象的大小。也就是,必须用 **align(2)**;声明将字对齐到偶数倍边界上;用 **align(4)**;声明将双字对齐到 4 字节整数倍边界上;用 **align(8)**;声明将四字对齐到 8 字节整数倍边界上。如果对象的大小不是 2 的幂,可将其对齐到 2 的更高次幂的边界上(最大为 16 字节)。但是要注意,只需要将 **real80**(以及 **tbyte**)对象对齐到 8 字节边界上。

注意,数据对齐并不是必需的。现代 80x86 CPU 的缓存机制实际上处理了绝大部分的不对齐数据。因此,只有在快速访问相当关键的情况下才使用对齐指令。这是一种合理的空间/时间折中方案。

3.5 地址表达式

在本章前面,我们指出了寻址方式有好几种通用形式,包括:

```

VarName[ Reg32 ]
VarName[ Reg32 + offset ]
VarName[ RegNotESP32*Scale ]
VarName[ Reg32 + RegNotESP32*Scale ]
VarName[ RegNotESP32*Scale + offset ]
VarName[ Reg32 + RegNotESP32*Scale + offset ]

```

还有另一种合法的形式,它不是一种新的寻址方式,只不过是位移寻址模式的扩展。即:

```
VarName [ offset ]
```

在这种寻址模式中,通过将括号中偏移常量和变量地址相加得到有效地址。例如,指令 **mov(Address[3], AL)**;将把存储器中 Address 对象后面第 3 个字节加载到 AL 寄存器(如图 3-8 所示)。

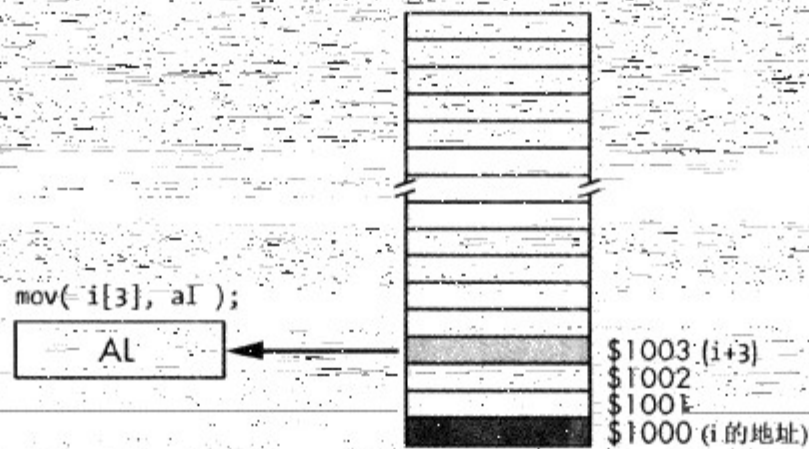


图 3-8 使用寻址表达式访问变量外部的数据

切记，这些例子中的 *offset* 值必须是常量。如果变址是一个 `int32` 变量，那么 `Variable[Index]` 就不是一个合法的寻址表达式。如果希望指定一个运行时可变的变址值，那么必须使用某种变址或者比例变址寻址方式。

还要记住，`Address[offset]` 中的 *offset* 是字节地址。尽管这类似于 C/C++ 或 Pascal 等高级语言的数组索引，但除非 `Address` 是一个字节数组，否则就会在对象数组中产生错位。

本书认为寻址表达式是任何合法的 80x86 寻址方式，其中包括位移量(也就是变量名称)或者偏移量。除了上述形式，还有如下的寻址表达式：

```
[ Reg32 + offset ]
[ Reg32 + RegNotESP32*Scale + offset ]
```

下列寻址表达式将不会被认为是寻址表达式，因为它们没有包含位移量或者偏移量：

```
[ Reg32 ]
[ Reg32 + RegNotESP32*Scale ]
```

寻址表达式的特别之处在于包含寻址表达式的指令总会将一个位移常量编码为机器指令的一部分。换句话说，机器指令包含一些用于保存数值常量的位(通常是 8 或 32 位)。该常量是位移量(即变量的地址或者偏移量)加上偏移量的和。注意，HLA 会自动求取这两个值的和(或者差——如果在寻址方式中使用“-”而不是“+”)。

至此，在所有寻址方式示例中，偏移量总是一个单值常量。但是，只要偏移量合法，HLA 总允许使用常量表达式。一个常量表达式包括一个或者多个常量项。这些常量项间还可以执行加、减、乘、除、取模以及其他多种操作。但是绝大多数寻址表达式只含有加、减、乘法，有时候含有除法操作。考虑如下的例子：

```
mov( X[ 2*4+1 ], al );
```

该指令会把地址 `X+9` 处的字节移到 `AL` 寄存器中。

寻址表达式的值总是在编译的时候计算，决不会在程序运行的时候计算。当 HLA 遇到上面的指令时，它会立即计算 `2*4+1` 并将该结果和 `X` 的基址相加。HLA 将该和(`X` 的基址加上 9)编码为指令的一部分；它不会发出额外的指令在运行时计算该表达式的和(这是合理的，因为如果这么做的话效率相对较低)。由于 HLA 在编译时计算寻址表达式的值，该表达式各部分必须是常量，因为在编译程序时，HLA 并不知道变量运行时的值。

寻址表达式对于访问存储器中某个变量外部的数据是很有效的，特别是在静态段或只读段中使用了 `byte`、`word`、`dword` 等语句在数据声明之后附加一些额外字节的时候。例如，看一下程序清单 3-1 中的程序。

程序清单 3-1 寻址表达式示例

```

program adrsExpressions;
#include( "stdlib.hhf" )
static
    i: int8; @nostorage;
    byte 0, 1, 2, 3;

begin adrsExpressions;

    stdout.put
    (
        "i[0]=", i[0], nl,
        "i[1]=", i[1], nl,
        "i[2]=", i[2], nl,
        "i[3]=", i[3], nl
    );

end adrsExpressions;

```

程序清单 3-1 中的程序将显示 0、1、2、3，就像它们是数组元素一样。这是因为位于 `i` 地址的值为 0(该程序采用 `@nostorage` 选项声明 `i`，所以 `i` 是静态段中下一个对象的地址，该对象作为 `byte` 语句的一部分，其值恰为 0)。寻址表达式 “`i[1]`” 告诉 HLA 取回位于 `i` 的地址加 1 处的那个字节。其值为 1，因为程序中的 `byte` 语句在静态段中紧接值 0 之后产生了值 1。对于 `i[2]`、`i[3]` 依此类推，程序显示它们的值分别为 2 和 3。

3.6 类型强制转换

尽管 HLA 在类型检查上是相当宽松的，但它还是会确保对某个指令所声明的操作数大小是适当的。例如，考虑下面的(错误)程序：

```

program hasErrors;
static
    i8:      int8;
    i16:     int16;
    i32:     int32;
begin hasErrors;

    mov( i8, eax );
    mov( i16, al );
    mov( i32, ax );

end hasErrors;

```

HLA 将对这 3 条 `mov` 指令报错，因为操作数的大小不兼容。第 1 条指令试图将一个字节移

到 EAX, 第2条指令试图将一个字节移到 AL, 第3条指令试图将一个双字移到 AX。而 mov 指令当然需要两个操作数大小一致。

尽管这是 HLA 的一个良好特性⁸, 但有时候它却成为了一种障碍。考虑如下的代码段:

```
static
    byte_values: byte; @nostorage;
                byte 0, 1;
    ...

    mov (byte_values, ax );
```

在该例中, 假设程序员真的想将起始于地址 `byte_values` 的字加载到 AX 寄存器中, 因为他想用一条指令使得 AL 加载 0, 且 AH 加载 1(注意, 0 保存在低位存储字节, 1 保存在高位存储字节)。HLA 将拒绝该语句, 声称此处有一个类型不匹配的错误(因为 `byte_values` 是一个字节对象, 而 AX 是一个字对象)。程序员可以将该指令分成两条指令, 一条将地址 `byte_values` 上的字节加载到 AL 中, 另一条将地址 `byte_values[1]` 上的字节加载到 AH 中。遗憾的是, 这种分解使得程序的效率略有下降(这也可能是使用前面单条 mov 指令的原因所在)。如果能通过某种方法告诉 HLA 我们对自己的行为有清醒的认识, 并且就是要将 `byte_values` 作为字对象来处理, 情况就要好些。HLA 的类型强制转换提供了此功能。

类型强制转换⁹(type coercion)是告知 HLA 您想将某个对象作为显式类型来处理, 而不理会其实际类型的过程。为了强制某个变量的类型, 可使用下面的语法:

```
(type newName addressExpression)
```

newName 项是您希望与某个存储单元相关联的新类型, 该存储单元由 *addressExpression* 指定。只要存储器的地址合法, 就可以使用这种强制操作符。为了纠正上面的例子, 以使得 HLA 不对类型不匹配报错, 可以使用下面语句:

```
mov( (type word byte_values), ax );
```

该指令告诉 HLA 将存储器中起始于地址 `byte_values` 的字加载到 AX 中。假定 `byte_values` 仍然含有其初始值, 该指令会将 0 加载到 AL 中, 1 加载到 AH 中。

在指定某个匿名变量作为某条直接对存储器进行修改的指令(如 `neg`、`shl`、`not` 等)的操作数时, 类型强制转换是必需的。考虑下面的语句:

```
not( [_ebx] );
```

HLA 将对该指令报错, 因为它无法确定操作数的大小。该指令没有提供足够的信息以确定程序是将 EBX 所指字节中的位取反, 或者将 EBX 所指字中的位取反, 还是将 EBX 所指双字中的位取反。必须通过下面的指令, 采用类型强制转换显式指定匿名引用的大小:

```
not( (type byte [ebx]) );
not( (type dword [ebx]) );
```

⁸ 毕竟, 如果两个操作数的大小不一致通常意味着程序中存在错误。

⁹ 在其他语言中也叫 type casting。

警告:

不要使用类型强制转换操作符，除非确实知道自己的行为并完全理解了其对程序的影响。初级的汇编程序员常常使用类型强制转换作为使得编译器对不匹配不报错的工具，而并没有解决潜在的问题。

考虑下面的语句(其中，*byteVar* 是一个 8 位的变量):

```
mov( eax, (type dword byteVar) );
```

如果没有类型强制转换操作符，HLA 会对该指令报错，因为它试图将一个 32 位的寄存器存储到一个 8 位的存储单元上。初级程序员为了程序能通过编译，可能采用一种捷径，像这个指令中一样使用类型强制转换操作符。该做法当然解决了编译器的問題——它不再对不匹配报错，因此初级程序员很满意。但是该程序仍然是不正确的；唯一的区别在于 HLA 不再报告错误。类型强制转换操作符无法修正将 32 位的数存入一个 8 位存储单元的错误，它只不过允许指令将 32 位的数存入以 8 位变量所指定地址为起点的存储单元中。该程序仍然存储了 4 个字节，重写了存储器中 *byteVar* 后面的 3 个字节。这常常会产生无法预期的后果，包括程序中变量会莫名其妙地被修改了¹⁰。另外一种较少出现的可能是，程序会因为通用保护故障而异常中止。如果这 3 个跟在 *byteVar* 后面的字节没有分配到物理存储器，或者这几个字节刚好落在存储器的只读段上，上述的情况就会发生。对于类型强制转换操作符，需要注意的一点是：如果无法明确说明该操作符的影响，就不要使用它。

还要记住的是，类型强制转换操作符不会对存储器中的数据进行任何转换。它只不过告知编译器将存储器中的位作为一个不同的类型来处理。它不会自动将某个 8 位值扩展为 32 位值，也不会将一个整数转换为浮点数。它只不过告知编译器将操作数的位模式作为其他类型来处理。

3.7 寄存器类型强制转换

通过类型强制转换操作符可以将寄存器指定为特定的类型。默认情况下，8 位寄存器的类型是字节，16 位寄存器的类型是字，32 位寄存器的类型是双字。通过类型强制转换，可以将某个寄存器指定为其他类型，只要新类型的大小和寄存器的大小相符。这是一个很重要的限制，在对内存变量采用类型强制转换时却并没有这样的限制。

大多数情况下，无须将寄存器指定为不同的类型。作为字节、字和双字对象，寄存器与所有的单、双和四字节对象兼容。但是，确实有一些寄存器类型强制转换适用的例子。HLA 高级语言语句(如 if 和 while)中的布尔表达式和 `stdout.put` 和 `stdin.get` 以及相关语句中的寄存器 I/O 就是两个很好的示例。

在布尔表达式中，HLA 总是将字节、字和双字对象作为无符号值。因此，没有类型强制转换的话，下面的 if 语句总会为假(因为没有比 0 小的无符号值):

```
if( eax < 0 ) then
    stdout.put( "EAX is negative!", nl );
```

¹⁰ 在该示例中，如果 *byteVar* 后紧跟了一个变量，不管是有意还是无意，`mov` 指令肯定会重写该变量的值。

```
endif;
```

通过将 EAX 强制转换为 int32 变量，可以克服该限制：

```
if( (type int32 eax) < 0 ) then
    stdout.put( "EAX is negative!", nl );
endif;
```

与此类似，HLA 标准库 `stdout.put` 例程总是将字节、字和双字值作为十六进制数输出。因此，如果试图输出某个寄存器，`stdout.put` 例程将把它作为十六进制数输出。若想将该值作为其他类型输出，可以使用寄存器类型强制转换：

```
stdout.put( "AL printed as a char = '", (type char al), "'", nl );
```

对于 `stdin.get` 函数也是同样的道理。它总是将十六进制数读到寄存器中，除非将其类型强制转换为字节、字或双字之外的某种类型。

3.8 栈段与 push 和 pop 指令

本章提到在 `var` 段声明的变量都处于栈存储器段。但是，栈存储器段并不只有 `var` 对象，程序可以以多种方式操作栈段中的对象。本节将讨论栈，并介绍用于操作栈段中的数据的 `push` 和 `pop` 指令。

存储器中的栈段是 80x86 维护栈的地方。栈是一种动态数据结构，它根据程序的需要可变长或者收缩。栈也存储和程序有关的重要信息，包括局部变量、子例程信息和临时数据。

80x86 通过 ESP(栈指针)寄存器控制栈。当程序开始运行，操作系统利用栈存储器段最后一个存储单元的地址初始化 ESP。通过“推”入栈和“弹”出栈将数据写入栈段。

3.8.1 基本的 push 指令

考虑 80x86 `push` 指令的语法：

```
push( reg16 );
push( reg32 );
push( memory16 );
push( memory32 );
pushw( constant );
pushd( constant );
```

这 6 种形式允许将字或双字寄存器、存储单元和常量入栈。应该特别注意不能将字节入栈。`push` 指令进行了如下操作：

```
ESP := ESP - Size_of_Register_or_Memory_Operand (2 or 4)
[ESP] := Operand's_Value
```

pushw 和 pushd 操作数分别是两字节或者四字节常量。

假定 ESP 存放的是\$00FF_FFE8, 那么指令 push(eax);将会把 ESP 设为\$00FF_FFE4, 并将 EAX 的当前值存储到存储单元\$00FF_FFE4 中, 如图 3-9 和图 3-10 所示。

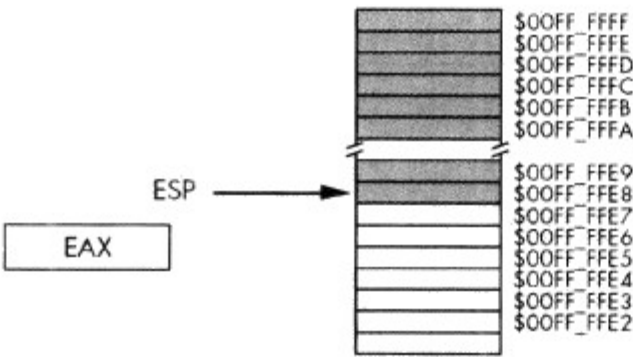


图 3-9 在 push(eax);操作之前的栈段

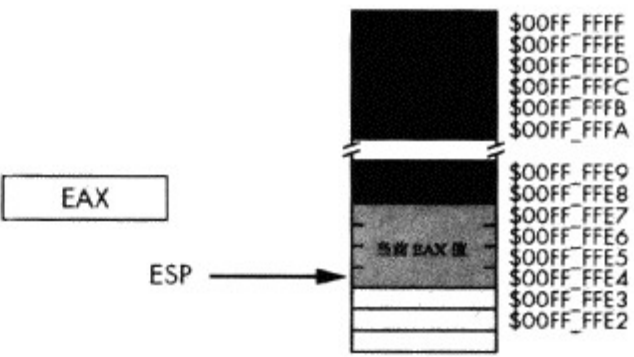


图 3-10 在 push(eax);操作之后的栈段

注意, push(eax);指令不会影响 EAX 寄存器的值。

尽管 80x86 支持 16 位的 push 操作, 这些操作主要用于 16 位的环境, 如 MS-DOS。为了使性能最优, 栈指针的值应该始终是 4 的偶数倍。实际上, 如果 ESP 包含某个不是 4 的倍数的值, 程序在 32 位操作系统环境下可能会出错。在将少于 4 个字节的数入栈的情况中, 唯一可行的是两次将字入栈从而建立一个双字。

3.8.2 基本的 pop 指令

为了取出入栈的数据, 可以使用 pop 指令。基本的 pop 指令允许下面的几种形式:

```
pop( reg16 );
pop( reg32 );
pop( memory16 );
pop( memory32 );
```

和 push 指令相似, pop 指令只支持 16 位和 32 位的操作数; 不能将一个 8 位的值出栈。如同 push 指令, 应该避免将 16 位值出栈(除非连续将两个 16 位的数出栈), 因为 16 位值出栈可能造成 ESP 指针含有非 4 的偶数倍的值。push 和 pop 之间一个主要的区别是不能将一个常数值出栈(这很合理, 因为对于 push 来说是源操作数, 而对 pop 来说却是目的操作数)。

一般, pop 指令采用如下的形式进行操作:

```
Operand := [ESP]
ESP := ESP + Size_of_Operand (2 or 4)
```

如上所示, pop 操作是 push 操作的逆操作。注意, pop 指令是在调整 ESP 中的值之前, 从存储位置[ESP]复制数据的。详见图 3-11 和图 3-12。

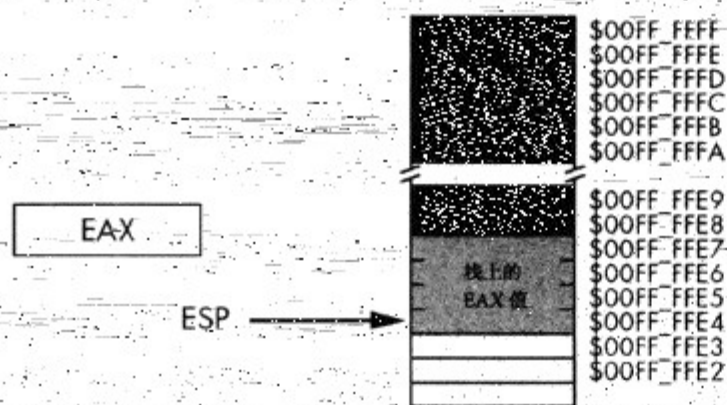


图 3-11 在 pop(cax); 操作之前的存储器

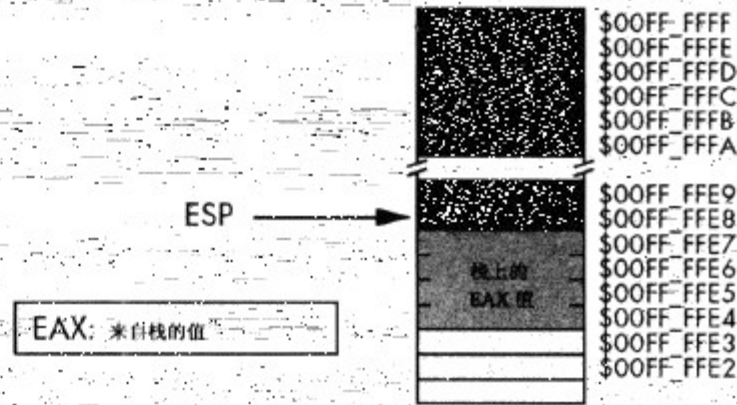


图 3-12 在 pop(cax); 操作之后的存储器

注意，出栈的值仍然存在存储器中。对某个值进行出栈操作并不会将其从存储器中抹掉，只不过是调整栈指针以使其指向已出栈值上方的下一个值。但是，决不能试图访问某个已经出栈的值。下一次入栈的时候，出栈值就会被抹去。由于并不是只有自己的代码在使用栈(例如，操作系统运行子例程要使用栈)，所以无法确保在出栈之后数据仍然保存在栈中。

3.8.3 用 push 和 pop 指令保护寄存器

在中间计算过程中，保存寄存器值可能是 push 和 pop 指令最主要的应用。80x86 体系结构的问题在于它所提供的通用寄存器太少了。由于寄存器是保存临时值的最佳场所，而且各种寻址方式中也需要寄存器，所以在编写那些计算复杂的代码时，很容易就会把所有的寄存器用完。此时 push 和 pop 指令就能帮上忙了。

考虑如下的程序：

```
<< Some sequence of instructions that use the eax register >>

<< Some sequence of instructions that need to use eax, for a
    different purpose than the above instructions >>

<< Some sequence of instructions that need the original value in eax >>
```

对于这样的情况，push 和 pop 指令是很适用的。通过在中间那条语句之前插入一条 push 指令以及在其后插入一条 pop 指令，就可以在计算中保护 EAX 的值。

```
<< Some sequence of instructions that use the eax register >>
push( eax );
<< Some sequence of instructions that need to use eax, for a
    different purpose than the above instructions >>
pop( eax );
<< Some sequence of instructions that need the original value in eax >>
```

上面的 push 指令将第一条语句所计算出来的数据复制到栈内。然后，中间那条指令就可以任意使用 EAX 了。中间指令结束之后，pop 指令又恢复了 EAX 的值。因此，最后那条指令可以使用 EAX 中的初值。

3.9 栈的 LIFO 数据结构

不需要先从栈内弹出数据就可以对多个值进行入栈操作。但是，栈是一个后入先出(Last In First Out, LIFO)的结构，因此在对多个值进行出入栈操作时必须当心。例如，假定在某些指令块中需要保护 EAX 和 EBX。下面的代码展示了如何用显式的方法来处理：

```
push( eax );
push( ebx );
<< Code that uses eax and ebx goes here. >>
pop( eax );
pop( ebx );
```

遗憾的是，该代码不会正常运行！图 3-13~图 3-16 显示了问题所在。因为该代码先将 EAX 入栈后再将 EBX 入栈，栈的指针将停在栈内 EBX 的值上。当运行 pop(eax);指令时，该指令会把开始在 EBX 中的值从栈内移出并放置到 EAX 中！同样，pop(ebx);指令将原来在 EAX 中的值出栈到 EBX 中。通过依照入栈的顺序出栈，最终的结果是将寄存器中的值进行了交换。

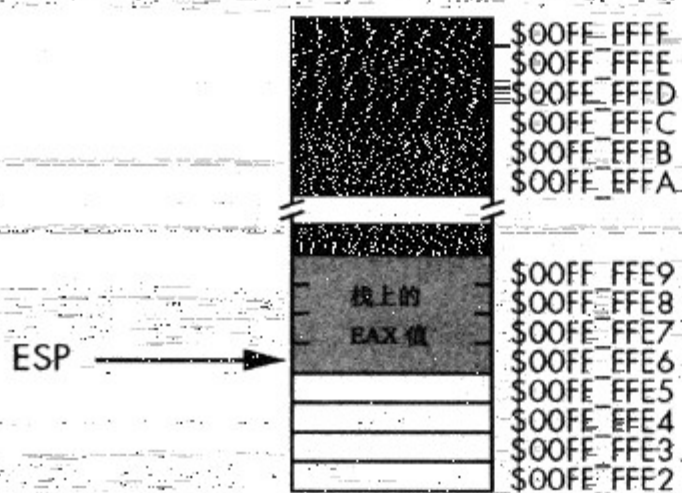


图 3-13 EAX 入栈后的栈

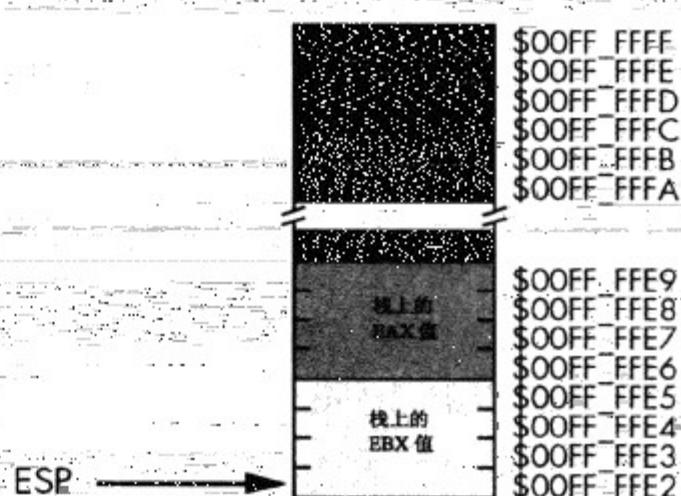


图 3-14 EBX 入栈后的栈

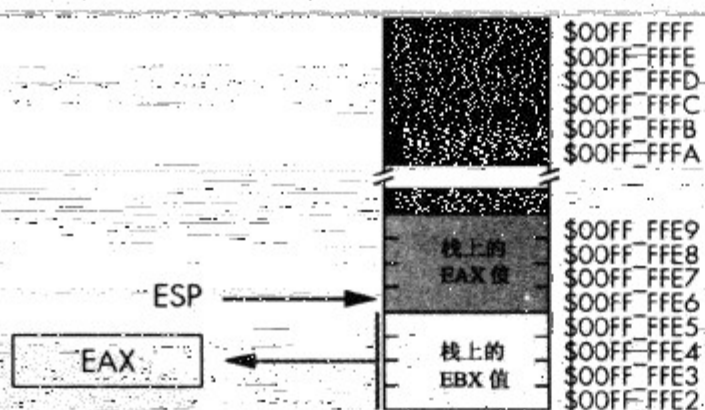


图 3-15 EAX 出栈后的栈

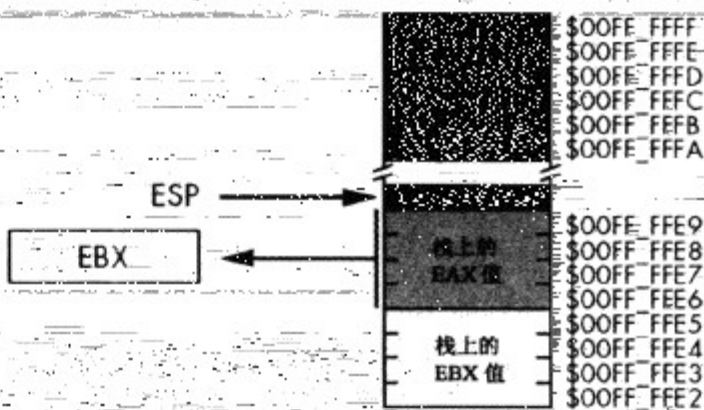


图 3-16 EBX 出栈后的栈

为了纠正该问题，必须注意，栈是 LIFO 的数据结构，因此第一个出栈的必须是最后一个入栈的。所以，必须遵守下列规则：总是按照与入栈相反的顺序出栈。

将前面的代码修正如下：

```
push( eax );
push( ebx );
<< Code that uses eax and ebx goes here. >>
pop( ebx );
pop( eax );
```

另一个要记住的重要规则是：出栈的字节数和入栈的字节数总相等。

这通常意味着入栈和出栈的数目必须一致。如果出栈太少，将会在栈中留下数据，会使运行的程序混乱；如果出栈太多，就会不小心将原来入栈的数据移除，常常会造成灾难性的后果。

由于上面的规则，所以在循环中进行出入栈操作要特别小心。在循环内入栈而在循环外出栈(或与之相反)常常是比较简单的，但会造成栈的不一致。记住，重要的是 `push` 和 `pop` 指令的执行，而不是在程序中出现的 `push` 和 `pop` 指令的次数。在运行时，程序执行的 `push` 指令的数目(顺序)必须和 `pop` 指令的数目(逆序)相符。

3.9.1 其他的 `push` 和 `pop` 指令

除了基本的 `push/pop` 指令之外，80x86 还提供了其他几种 `push` 和 `pop` 指令。这些指令包括如下几种：

<code>pusha</code>	<code>popa</code>
<code>pushad</code>	<code>popad</code>
<code>pushf</code>	<code>opf</code>
<code>pushfd</code>	<code>popfd</code>

`pusha` 指令将所有 16 位通用寄存器入栈。该指令主要用于诸如 MS-DOS 之类的 16 位操作系统中。通常，很少用到该指令。`pusha` 指令按照如下的顺序将寄存器入栈：

```
ax
cx
dx
bx
sp
bp
si
di
```

`pushad` 指令将所有 32 位(双字)寄存器入栈。它依照下列顺序将寄存器入栈：

```
eax
ecx
edx
ebx
esp
ebp
esi
edi
```

由于 `pusha` 和 `pushad` 指令对 SP/ESP 寄存器的固有修改，您可能会对 Intel 将该寄存器入栈的

原因感到疑惑。也许对于硬件来说，直接将 SP/ESP 入栈比把它当成一种特殊情况进行处理要简单一些。在任何情况下，这些指令都会将 SP 或 ESP 入栈，所以不必过于担心——也无法对此采取任何措施。

popa 和 popad 指令针对 pusha 和 pushad 指令提供了相应的“全出栈”操作。这将使得由 pusha 和 pushad 入栈的寄存器依照相应的顺序出栈(即按照与 pusha 或 pushad 入栈相反的顺序，popa 和 popad 将寄存器的值出栈从而恢复其值)。

尽管 pusha/popa 和 pushad/popad 序列简短方便，但它们比相应的 push/pop 指令序列要慢，而且事实上很少需要将大部分寄存器入栈，更不用说全部寄存器了¹¹。所以，如果寻求最快的速度，应当仔细考虑是否使用 pusha(d)/popa(d)指令。

pushf、pushfd、popf 和 popfd 指令将 EFLAGS 寄存器入栈和出栈。这些指令允许在某些指令序列的执行过程当中保护条件码以及其他标志设置。遗憾的是，除非解决很多麻烦，否则很难对单个的标志进行保护。当使用 pushf(d)和 popf(d)指令时，要么保存所有的标志，要么全部不保存：当入栈时，所有的标志位都受到保护；当出栈时，所有的标志位都复原。

正如 pushad 和 popad 指令一样，应当使用 pushfd 和 popfd 指令将 32 位的 EFLAGS 寄存器入栈。尽管在编写应用程序时，出入栈的多余的 16 位会被忽略掉，您仍然想要通过只出栈和入栈双字来保持栈的对齐。

3.9.2 不使用出栈而从栈内移除数据

一旦栈内有不再需要的数据存在，可以将该数据出栈到某个未使用的寄存器或者内存单元上，但是还有更简单的将不需要的数据从栈内移除的方法，即仅仅调整 ESP 寄存器的值来跳过栈内不需要的数据。

考虑如下的难题：

```

push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into eax and ebx >>

if ( Calculation_was_performed ) then

    // Whoops, we don't want to pop eax and ebx!
    // What to do here?

else

    // No calculation, so restore eax,ebx.

    pop( ebx );
    pop( eax );

endif;
```

在 if 语句的 then 段中，代码试图将 EAX 和 EBX 中的旧值移除而不影响任何寄存器或者存储单元。如何实现呢？

¹¹ 例如，极少需要利用 pushad/popad 指令序列将 ESP 寄存器出入栈。

由于 ESP 寄存器含有栈顶项的存储器地址,可以通过往 ESP 寄存器上添加该项的大小而从栈顶移除该项。在前面的例子中,我们试图从栈顶移除两个双字项。可以通过栈指针加上 8 轻松实现(详见图 3-17 和图 3-18)。

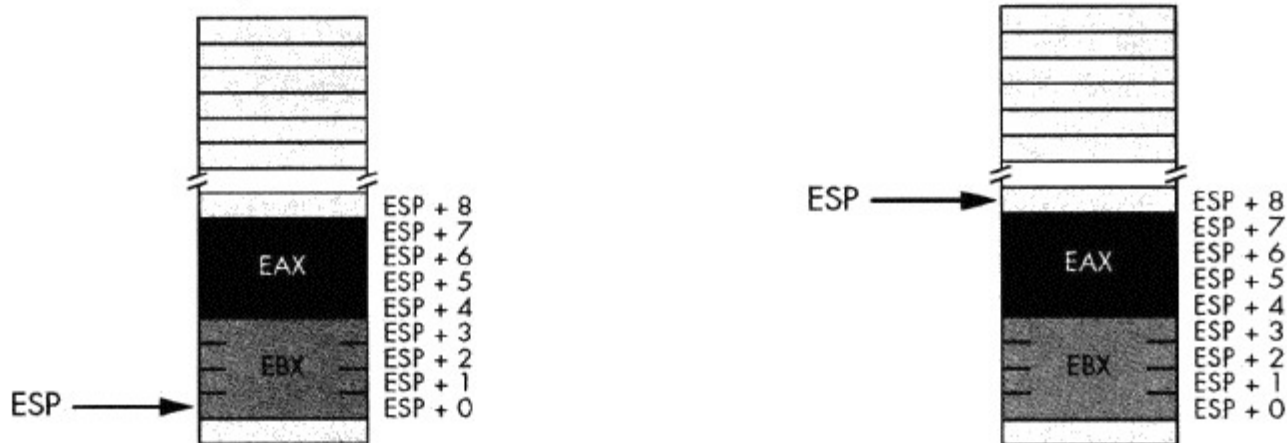


图 3-17 从栈内移除数据(在执行 add(8, esp); 指令之前) 图 3-18 从栈内移除数据(在执行 add(8, esp); 指令之后)

```
push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into eax and ebx >>

if( Calculation_was_performed ) then

    add(8,ESP ); // Remove unneeded eax/ebx values from the stack.

else

    // No calculation, so restore eax, ebx.

    pop( ebx );
    pop( eax );

endif;
```

该代码没有将数据移动到任何地方就有效地将数据弹出了栈。还要注意的,该代码比两条 pop 指令更快,因为它可以用一条 add 指令从栈中删除任意数目的字节。

警告:

记住,要保持栈对准某个双字边界。因此,从栈内移除数据时,应该总是将 4 的倍数的常量添加到 ESP 上。

3.10 访问已入栈而未出栈的数据

有时候,在将数据入栈之后,想要得到该数据值的一个副本,或者想要修改该数据的值,而实际上又不将该数据出栈(想在稍后再出栈该数据)。80x86 的[reg32 + offset]寻址方式提供了这样的机制。

考虑执行了以下两条指令后的栈(见图 3-19):

```
push( eax );
push( ebx );
```

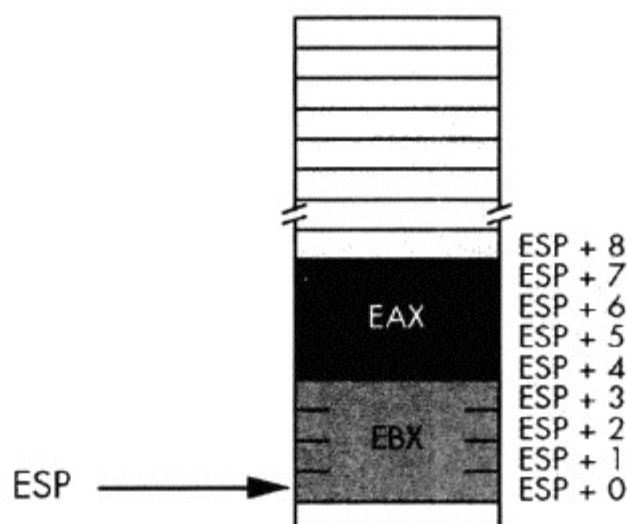


图 3-19 EAX 和 EBX 入栈之后的栈

若要访问初始的 EBX 值而不将其从栈内移除，可以对该值采用出栈并立即再将其入栈的欺骗方式。但是，假定想要访问 EAX 原来的值，或者栈内其他更远一些的值。将所有的中间值出栈然后再将它们入栈，轻则有问题发生，重则不可能实现。然而，如图 3-19 所示，入栈的每一个值都处于 ESP 寄存器的某个偏移位置处。因此，可以使用[ESP + offset]寻址方式来获取所需值的直接地址。在上面的例子中，可以使用一条指令重新加载 EAX 的初值。

```
mov( [esp+4], eax );
```

该代码将从地址 ESP + 4 开始的 4 个字节复制到 EAX 寄存器中。该值恰为入栈的 EAX 的初始值。可以使用同样的方法访问入栈的其他数据值。

警告：

值对 ESP 的偏移量在每次入栈或出栈时都会改变。滥用该特性会创建难以修改的代码；如果代码中从头至尾使用该特性，在第一次入栈数据的这一点到决定使用[ESP + offset]寻址方式再次访问该数据的这一点之间很难入栈和出栈其他的数据项。

前面的章节指出了如何通过将一个常量和 ESP 寄存器相加来从栈内移除数据。该代码或许可以写得更安全，如下所示：

```
push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into eax and ebx >>

if( Calculation_was_performed ) then

    << Overwrite saved values on stack with new eax/ebx values.
        (so the pops that follow won't change the values in eax/ebx.) >>

    mov( eax, [esp+4] );
    mov( ebx, [esp] );
```

```
endif;
pop( ebx );
pop( eax );
```

在该代码序列中，计算结果存储在栈内存储数据的上方。稍后，当程序对该值进行出栈操作时，它将这些计算值加载到 EAX 和 EBX 中。

3.11 动态内存分配和堆段

尽管静态和自动变量可能是简单程序所需要的全部变量，更复杂的程序还需要在程序控制下动态(运行时)分配或回收存储区。在 C 语言中，使用 malloc 和 free 函数。C++ 提供了 new 和 delete 操作符。Pascal 使用 new 和 dispose。其他语言也提供了相应的工具。这些内存分配例程在很多地方有相同之处：它们允许程序员请求所需要分配存储区的字节数，返回新分配存储区的指针，并提供将存储区返还给系统的能力，以便于在将来的分配调用中再次使用该存储区。正如您所猜想到的那样，HLA 的标准库中也提供了一套例程来处理内存分配和回收。

HLA 标准库例程 mem.alloc 和 mem.free 分别处理存储空间的分配和回收。mem.alloc 例程采用下面的调用顺序：

```
mem.alloc( Number_of_Bytes_Requested );
```

这个参数是一个双字值，指定了所需要的存储区字节数。该过程在存储器的堆段中分配存储区。HLA 的 mem.alloc 函数依照您所声明的大小在堆段中找到一个未被使用的存储区，将其标记为“使用中”，因此后面的 mem.alloc 调用就无法分配相同的存储区了。标记该区域为“使用中”之后，mem.alloc 例程返回一个指向该存储区首字节的指针给 EAX 寄存器。

对于许多对象，您必须知晓所需要的字节数目以在存储器中表示该对象。例如，若要为某个 uns32 变量分配存储区，可以这样调用 mem.alloc 例程：

```
mem.alloc( 4 );
```

尽管可以指定如上所示的常量，但在为指定的数据类型分配存储区时，这样做通常并不是个好办法。取而代之的是，我们可以使用 HLA 内置的编译时函数¹²@size 计算某个数据类型的数据宽度。@size 函数使用下面的语法：

```
@size( variable_or_type_name )
```

@size 函数返回一个无符号的整型常量，该常量值就是其参数的字节数目。因此应当重写上面的调用：

```
mem.alloc( @size( uns32 ) );
```

该调用将为指定的对象分配一个足够的存储区，而不管其类型如何。虽然 uns32 对象的字节数很少有可能变化，对于其他的数据类型来说却并不是这样，因此应该总是使用 @size 而不是直

¹² 编译时函数由 HLA 在编译程序时而不是在运行时调用。

接的常量。

一旦 `mem.alloc` 例程返回, `EAX` 寄存器就包含了所请求的存储区的地址(如图 3-20 所示)。

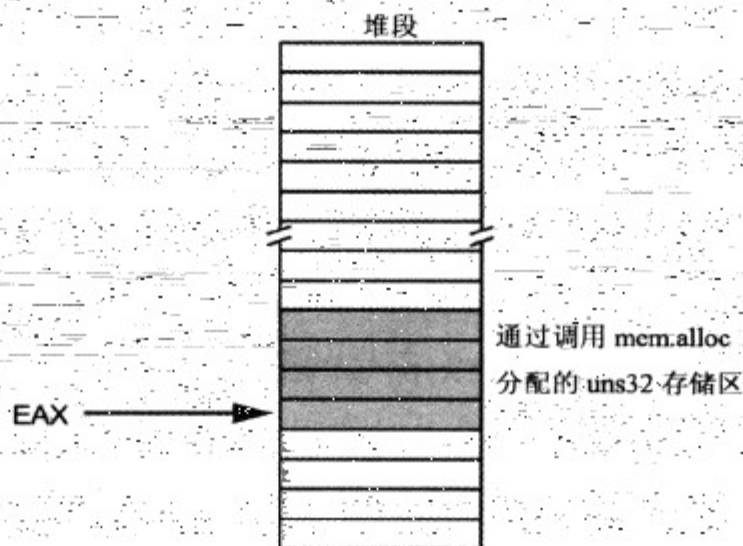


图 3-20 `mem.alloc` 调用返回指针给 `EAX` 寄存器

要访问 `mem.alloc` 分配的存储区, 必须使用一种寄存器间接寻址方式。下面的代码序列展示了如何为 `mem.alloc` 创建的 `uns32` 变量指定值 1234:

```
mem.alloc( @size( uns32 ) );
mov( 1234, (type uns32 [eax]) );
```

注意 `type` 强制转换操作符的使用。在上面的示例中, 这是很必要的, 因为匿名变量没有相关的数据类型, 而常量 1234 可以是字或者双字值。`type` 强制转换操作可以消除这种不确定性。

`mem.alloc` 例程并不总是成功的。如果在堆段中没有足够大的单块连续未使用存储区来满足请求, 那么 `mem.alloc` 例程将引发 `ex.MemoryAllocationFailure` 异常。如果不提供 `try..exception..endtry` 来处理这种情况, 内存分配失败将导致程序中止。因为绝大多数的程序并不使用 `mem.alloc` 来分配大部分的动态内存, 这类异常极少发生。但是, 不能认为内存分配没有任何错误。

用完 `mem.alloc` 在堆上动态分配的某个值之后, 通过调用 `mem.free` 过程可以释放该存储区(即标记其为“不再使用”)。`mem.free` 例程需要一个参数, 该参数必须是前面某个 `mem.alloc` 调用所返回的地址(尚未被释放过)。下面的代码段展示了 `mem.alloc/mem.free` 的本质:

```
mem.alloc( @size(uns32));

<< use the storage pointed at by eax. >>
<< Note: this code must not modify eax. >>

free( eax );
```

该代码说明了很重要的一点: 为了合理释放由 `mem.alloc` 分配的存储区, 必须保存 `mem.alloc` 返回的值。如果 `EAX` 还有别的用处, 可以采用多种方式来保存。可以通过 `push` 和 `pop` 指令在栈上保存指针值, 或者将 `EAX` 的值保存至某个变量直到需要释放时。

所释放的存储区在后面的 `mem.alloc` 调用中可以重新使用。需要时分配内存, 不需要时释放内存的这种能力提高了程序的存储效率。通过一旦用完就释放的方式, 程序能够重新使用该存储区做别的事情。与为单个变量静态分配存储区相比, 这么做程序使用的存储区就要少一些。

使用指针的时候可能会发生某些问题。在使用诸如 `mem.alloc` 和 `mem.free` 等动态存储区分配例程时, 初级程序员会犯下列一些常见错误。

- 错误 1: 释放存储区之后还在使用它。一旦调用 `mem.free` 将存储区还给系统后, 就不应该再访问该存储区。这样做可能导致保护错误, 或者更糟糕的是在没有任何错误警告的情况下, 破坏了程序中的其他数据。
- 错误 2: 对一个存储区调用两次 `mem.free`。这样做可能意外释放某些无意释放的存储区, 或者更糟糕的是可能破坏系统内存管理表。

第 4 章将讨论在处理动态分配存储区时常遇到的另外一些问题。

到目前为止, 本节的例子都是为某个无符号的 32 位对象分配存储区。显然, 通过调用 `mem.alloc` 并指定对象的大小作为其参数, 还可以为别的数据类型分配存储区。调用 `mem.alloc` 还可以为一组连续的对象分配存储区。例如, 下面的代码将为一个 8 字符序列分配存储区:

```
mem.alloc( @size( char ) * 8 );
```

注意, 这里使用常量表达式来计算 8 字符序列所需要的字节数。因为 `@size(char)` 总是返回一个常量值(本例中返回 1), 编译器能计算表达式 `@size(char)*8` 而不产生额外的机器指令。

对 `mem.alloc` 的调用总会在连续的存储单元分配多个字节的存储区。因此, 前面对 `mem.alloc` 的调用情况如图 3-21 所示。

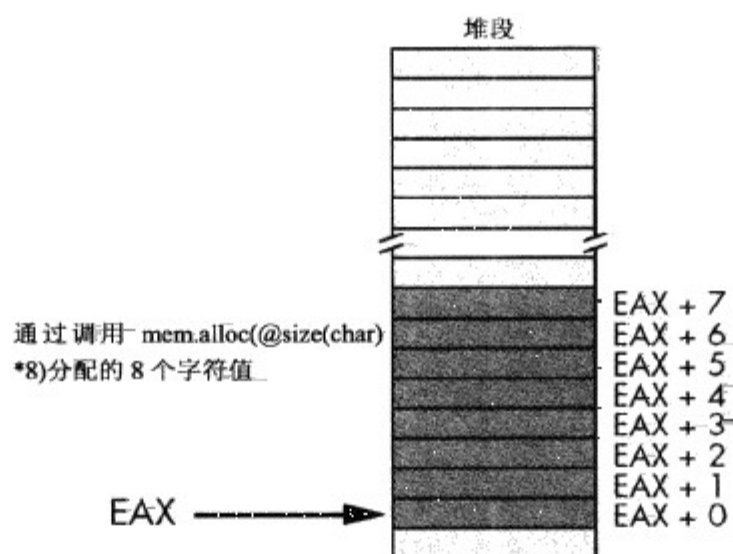


图 3-21 使用 `mem.alloc` 分配 8 个字符

为了访问这些字符值, 可使用基址(由 `mem.alloc` 返回给 `EAX`)的偏移量。例如, `mov(ch, [eax + 2]);` 将 `CH` 中的字符存放到 `mem.alloc` 分配的第三个字节中。可以使用诸如 “[`eax + ebx`]” 的寻址方式在程序控制下访问所有被分配的对象, 下面的代码将设定 128 个字节区域内所有的字符为 `NUL(#0)`:

```
mem.alloc( 128 );
for( mov( 0, ebx ); ebx < 128; add( 1, ebx ) ) do
    mov( 0, (type byte [eax+ebx]) );
endfor;
```

第 4 章将讨论复合数据结构(包括数组)并介绍其他处理存储区的方法。

应当注意, `mem.alloc` 调用实际上分配的存储区比所请求的稍大一些。首先, 内存分配请求的空间通常最小(常常在 4 和 16 之间, 但这要取决于操作系统)。而且, 每次 `mem.alloc` 请求也需要一些多余的字节(通常是 16~32 字节)来跟踪所分配和释放的区域。因此, 为大量的小对象单独调用 `mem.alloc` 分配内存的效率并不高。每次分配时额外分配的字节可能比实际使用的更多。通常情况下, 可以使用 `mem.alloc` 为数组或者大的记录(结构)而不是小的对象分配存储区。

3.12 `inc` 和 `dec` 指令

上节示例或者说到目前为止的几个例子表明, 从一个寄存器或者存储单元加 1 或者减 1 是很常见的操作。实际上, 由于这些操作常用, Intel 的工程师们加入了一对指令: `inc` 和 `dec`。

`inc` 和 `dec` 指令采用如下的语法:

```
inc( mem/reg );
dec( mem/reg );
```

其单个的操作数可以是任何合法的 8 位、16 位或 32 位寄存器或者内存操作数。`inc` 指令将对指定的操作数加 1; `dec` 指令则减 1。

这两条指令比相应的 `add` 或者 `sub` 指令要短一些(也就是说, 它们的编码使用了较少的字节)。它们和 `add` 或 `sub` 指令之间还有细微的区别: `inc` 和 `dec` 指令并不影响进位标志。

作为 `inc` 指令的一个例子, 将上一节的例子改为使用 `inc` (而不是 `add`):

```
mem.alloc( 128 );
for( mov( 0, ebx ); ebx < 128; inc( ebx ) ) do

    mov( 0, (type byte [eax+ebx]) );

endfor;
```

3.13 获取存储器对象的地址

本章前面讨论了如何使用取址操作符“&”来获得静态变量的地址¹³。遗憾的是, 无法使用取址操作符获得自动变量的地址(在 `var` 段声明的变量); 无法用它计算匿名变量的地址, 也不能用它获取使用变址或者比例变址寻址方式的内存引用的地址(即使静态变量作为寻址表达式的一部分)。只能使用取址表达式获得一个简单的静态对象的地址。通常, 还需要获取其他存储器对象的地址; 幸运的是, 80x86 提供了“加载有效地址”的指令 `lea`。

`lea` 指令采用下面的语法:

```
lea( reg32, Memory_operand );
```

第一个操作数必须是一个 32 位的寄存器; 第二个操作数可以是任何合法的内存引用, 可以使用任何合法的寻址方式。该指令将指定存储单元的地址加载到寄存器中。不管以何种方式, 它

¹³ 静态变量是在程序的静态段、只读段或者存储段声明的变量。

都不会访问或修改内存操作数的值。

一旦在 32 位的通用寄存器中加载了有效的存储单元地址,就可以使用寄存器间接、变址、比例变址寻址方式访问指定存储单元中的数据。考虑如下的代码段:

```
static
    b: byte; @nostorage;
        byte 7, 0, 6, 1, 5, 2, 4, 3;
        .
        .
        .
    lea( ebx, b );
    for( mov( 0, ecx ); ecx < 8; inc( ecx ) ) do

        stdout.put( "[ebx+ecx] = ", (type byte [ebx+ecx]), nl );

    endfor;
```

该代码遍历了静态段中 b 标记后的 8 个字节,并将它们的值打印出来。注意[ebx+ecx]寻址方式的使用。EBX 寄存器保存了该列表的基址(即列表中首项的地址),ECX 保存了列表的字节变址。

3.14 更多信息

访问 <http://webster.cs.ucr.edu/> 站点,可发现一个旧的 16 位版本的 *The Art of Assembly Language Programming*。在该文本文件中,可找到有关 80x86 的 16 位寻址方式和分段的信息。关于 HLA 标准库的 mem.alloc 和 mem.free 函数的详细信息,可以在 <http://webster.cs.ucr.edu/> 或 <http://artofasm.com/> 上的 HLA 标准库参考手册中找到。当然,Intel x86 文档(<http://www.intel.com/>上提供)提供了关于 80x86 寻址方式和机器指令编码的完整信息。

第 4 章

常量、变量与数据类型



第2章讨论了内存中数据的基本格式。第3章讲述了计算机系统如何组织内存中的数据。本章通过将数据表示的概念与实际的物理表示相结合来完成这一讨论。正如标题所示，本章旨在介绍3个主题：常量、变量和数据结构。本章假定您还没有正式学习过数据结构课程。

本章讨论了如何声明和使用常量、标量变量、整数、数据类型、指针、数组、记录/结构、联合和命名空间。在进入下一章的学习以前，您必须掌握它们。尤其是，数组的声明和访问似乎给汇编语言初学者提出了一大堆的问题。但是，本书余下的部分都取决于您对这些数据结构以及它们的存储器表示的理解。不要企图跳过这些内容，指望在后面需要的时候再来学习。现在您就需要了解它们，如果将这里的内容和后面的内容一起学习只会让您更加糊涂。

4.1 一些额外的指令：intmul、bound、into

本章介绍数组和其他的概念，它们需要您对80x86指令集的知识进行扩展。尤其需要学会如何将两个数相乘。因此，我们将看到的第一条指令是intmul(整数乘法)指令。访问数组时，通常要检查数组索引是否处于边界以内。80x86的bound指令提供了一种方便的途径来对寄存器值进行检查，看它是否处于某个范围之内。into((上溢中断)指令为有符号算术运算上溢提供了快速的检查。虽然对于数组(或者其他数据类型)的访问来说，into并不真正有必要，但是它的功能非常类似于bound，因此还是在这里对它进行了介绍。

intmul指令可以采用下面任何一种格式：


```
// The following compute destreg = destreg * constant

intmul( constant, destreg16 );
intmul( constant, destreg32 );

// The following compute dest = src * constant

intmul( constant, srcreg16, destreg16 );
intmul( constant, srcmem16, destreg16 );

intmul( constant, srcreg32, destreg32 );
intmul( constant, srcmem32, destreg32 );

// The following compute dest = src * constant

intmul( srcreg16, destreg16 );
intmul( srcmem16, destreg16 );
intmul( srcreg32, destreg32 );
intmul( srcmem32, destreg32 );
```

注意, `intmul` 指令的语法与 `add` 指令和 `sub` 指令是不同的。特别要注意的是, 它的目的操作数必须是寄存器(而 `add` 和 `sub` 都允许将内存操作数作为目的操作数)。还要注意的是当第一个操作数为常量时, `intmul` 允许有三个操作数。另一个重要的差别在于 `intmul` 指令只允许 16 位和 32 位的操作数; 它不对 8 位的操作数做乘法。

`intmul` 为指定的操作数计算乘积, 并将结果存储到目的寄存器。如果发生了上溢(通常都是有符号数的上溢, 因为 `intmul` 只对有符号的整数值进行乘法运算), 那么该指令会设置进位标志和上溢标志。`intmul` 使其他的条件码标志处于未定义状态(所以, 在 `intmul` 执行后不能对符号标志或者零标志进行有意义的检查。

`bound` 指令通过对 16 位或者 32 位的寄存器进行检查来判断它的值是否处于某两个数值之间。如果数值超出了该范围, 程序就会引发异常并终止。在判断数组索引是否处于给定范围内时, 该指令特别有用。`bound` 指令可以采用如下所示的任何一种格式:

```
bound( reg16, LBconstant, UBconstant );
bound( reg32, LBconstant, UBconstant );

bound( reg16, Mem16[2] );
bound( reg32, Mem32[2] );
```

`bound` 指令将它的寄存器操作数与一个无符号下界值和一个无符号上界值进行比较, 以确保寄存器处于该范围之内:

```
lower_bound <= register <= upper_bound
```

带三个操作数的 `bound` 指令将寄存器与第二个和第三个参数进行比较(分别为下界和上界)¹。

¹ 这种形式不是真正的 80x86 指令。HLA 通过创建两个初始化为指定常量的只读内存变量将这种格式的 `bound` 指令转换为带两个操作数的形式。

带两个操作数的 **bound** 指令检查寄存器是否超出下面任意一个范围：

```
Mem16[0] <= register16 <= Mem16[2]
```

```
Mem32[0] <= register32 <= Mem32[4]
```

如果指定的寄存器不在给定的范围内，那么 80x86 就引发一次异常。该异常可以用 HLA 的 `try..endtry` 异常处理语句捕获。头文件 `excepts.hhf` 专门为此目的定义了 `ex.BoundsInstr` 异常。程序清单 4-1 中的程序展示了如何使用 **bound** 指令对用户输入的值进行检查。

程序清单 4-1 **bound** 指令的示例

```
program BoundDemo;
#include( "stdlib.hhf" );

static
    InputValue:int32;
    GoodInput:boolean;

begin BoundDemo;

    // Repeat until the user enters a good value:

    repeat

        // Assume the user enters a bad value.

mov( false, GoodInput );

// Catch bad numeric input via the try..endtry statement.
try

    stdout.put( "Enter an integer between 1 and 10:" );
    stdin.flushInput();
    stdin.geti32();

    mov( eax, InputValue );

    // Use the BOUND instruction to verify that the
    // value is in the range 1..10.

    bound( eax, 1, 10 );

    // If we get to this point, the value was in the
    // range 1..10, so set the boolean GoodInput
    // flag to true so we can exit the loop.

    mov( true, GoodInput );

    // Handle inputs that are not legal integers.

    exception( ex.ConversionError )

    stdout.put( "Illegal numeric format, re-enter", nl );
```



```
// Handle integer inputs that don't fit into an int32.
exception( ex.ValueOutOfRange )

    stdout.put("Value is *way*too big, re-enter", nl );

// Handle values outside the range 1..10 (BOUND instruction)
exception( ex.BoundInstr )

    stdout.put
    (
        "Value was ",
        InputValue,
        ", it must be between 1 and 10, re-enter",
        nl
    );

endtry;

until( GoodInput );
stdout.put( "The value you entered, ", InputValue, "is valid.", nl );

end BoundDemo;
```

与 **bound** 指令一样，**into** 指令在特定的条件下也会产生异常。特别是，如果设置了上溢标志，**into** 就会产生异常。通常，在有符号算术操作(例如 **intmul**)之后都会立即使用 **into** 来检查是否发生了上溢。如果上溢标志没有设置，那么系统就忽略 **into** 指令；但是如果设置了上溢标志，那么 **into** 指令就引发 **ex.IntoInstr** 异常。程序清单 4-2 中的程序展示了 **into** 指令的使用。

程序清单 4-2 into 指令的示例

```
program INTOdemo;
#include( "stdlib.hhf" );

static
    LOperand:int8;
    ResultOp:int8;

begin INTOdemo;
    // The following try..endtry checks for bad numeric
    // input and handles the integer overflow check:

    try

        // Get the first of two operands:

        stdout.put( "Enter a small integer value (-128..+127):" );
        stdin.geti8();
        mov( al, LOperand );

        // Get the second operand:
```

```

stdout.put( "Enter a second small integer value (-128..+127):" );
stdin.geti8();

// Produce their sum and check for overflow:

add( LOperand, al );
into();

// Display the sum:

stdout.put( "The eight-bit sum is ", (type int8 al), nl );

// Handle bad input here:
exception( ex.ConversionError )

    stdout.put( "You entered illegal characters in the number", nl );

// Handle values that don't fit in a byte here:
exception( ex.ValueOutOfRange )

    stdout.put( "The value must be in the range -128..+127", nl );

// Handle integer overflow here:

exception( ex.IntoInstr )

    stdout.put
    (
        "The sum of the two values is outside the range -128..+127",
        nl
    );

endtry;

end= INTOdemo;

```

4.2 HLA 常量和数值声明

HLA 的 `const` 和 `val` 段允许声明符号常量。`const` 段允许声明那些在整个编译和运行过程中值为常量的标识符；`val` 段允许声明那些数值在编译时可以发生改变，但在运行时必须是常量的符号常量(也就是说，在源代码当中，同样的名称可以有不同的值，但是当程序在运行时，`val` 的值在程序中的某个给定点不能改变)。

`const` 段出现在程序中包含静态段、只读段、存储段和 `var` 段的同一个声明段中。它以保留字 `const` 开头，其语法与 `readonly` 段大致相同；也就是说，`const` 段包含了一个标识符列表，每个标识符的后面都跟着类型说明和常量表达式。下面给出了一些 `const` 段的例子：

```

const
    pi:                real32        :=3.14159;

```



```

MaxIndex:    uns32      := 15;
Delimiter:   char       := '/';
BitMask:     byte       := $F0;
DebugActive: Boolean    := true;

```

一旦以这种方式对这些常量进行了声明,在与其相应的字面常量合法的地方就可以使用符号标识符。这些常量被称为明示常量(manifest constant)。明示常量是常量的一种符号表示,它可以在程序中的任意位置替代该常量的字面数值。这一点恰好与只读(readonly)类型的变量相反:只读类型的变量必定是常量,因为在运行时它们的数值不能改变。但是,存储器位置是与只读变量相关联的,并且操作系统(而不是 HLA 编译器)在运行时强制只读属性。虽然像 `mov(eax, ReadOnlyVar)`; 这样的指令在运行的时候肯定会使程序崩溃,但是这条指令却是完全合法的。另一方面, `mov(eax, MaxIndex)`; (使用上面的声明)与 `mov(eax, 15)` 一样不合法。实际上,由于编译器在遇到该明示常量时会用 15 来替代 MaxIndex,所以这两条语句是等效的。

如果对常量的类型一点都不模糊,那么在声明常量时就可以省略它的类型说明,而只指定常量的名称和数值。在前面的例子当中,常量 `pi`、`Delimiter`、`MaxIndex` 和 `DebugActive` 可以使用下面的声明:

```

const
  pi      := 3.14159;      // Default type is real80.
  MaxIndex := 15;          // Default type is uns32.
  Delimiter := '/';       // Default type is char.
  DebugActive := true;    // Default type is boolean.

```

如果具有整数字面常量的符号常量值为 0 或者为正,那么它总是被假定为最小的无符号类型;如果其值为负,那么就会被定义为最小的整数类型(int8、int16 等)。

常量声明对于定义“幻数”是非常重要的,幻数就是那些在对程序进行修改时可能发生变化的数。程序清单 4-3 给出了使用常量来对幻数值进行参数化的例子。在这个例子中,程序为要对测试、对齐和循环及数据重复数分配的存储区大小定义了明示常量。这个程序演示了不对齐的数据访问对程序性能的影响。如果程序过快或过慢,则需要调整 `MainRepetitions` 常量。

程序清单 4-3 使用 const 定义重写的数据对齐程序

```

program ConstDemo;
#include( "stdlib.hhf" );

const
  MemToAllocat := 4_000_000;
  NumDWords    := MemToAllocate div 4;
  MisalignBy   := 62;

  MainRepetitions := 1000;
  DataRepetitions := 999_900;

  CacheLineSize := 16;

begin ConstDemo;

  //console.cls();

```

```

stdout.put
(
    "Memory Alignment Exercise", nl,
    nl,
    "Using a watch (preferably a stopwatch), time the execution of", nl
    "the following code to determine how many seconds it takes to", nl
    "execute.", nl
    nl
    "Press Enter to begin timing the code:"
);

// Allocate enough dynamic memory to ensure that it does not
// all fit inside the cache. Note: the machine had better have
// at least 4 megabytes mem.free or virtual memory will kick in
// and invalidate the timing.

mem.alloc( MemToAllocate );

// Zero out the memory (this loop really exists just to
// ensure that all memory is mapped in by the OS).

mov( NumDWords, ecx );
repeat
    dec( ecx );
    mov( 0, (type dword [eax+ecx*4]) );
until( !ecx ); // Repeat until ecx = 0.

// Okay, wait for the user to press the Enter key.
stdin.readLn();

// Note: As processors get faster and faster, you may
// want to increase the size of the following constant.
// Execution time for this loop should be approximately
// 10-30 seconds.

mov( MainRepetitions, edx );
add( MisalignBy, eax ); // Force misalignment of data.

repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );

    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );

```



```

    dec( edx );

until( !edx );    // Repeat until eax is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );

// Okay, time the aligned access.

stdout.put
(
    "Press Enter again to begin timing access to aligned variable:"
);
stdin.readLn();

// Note: If you change the constant above, be sure to change
// this one, too!

mov( MainRepetitions, edx );
sub( MisalignBy, eax );    // Realign the data.
repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );
    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx );    // Repeat until eax is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );
free( eax );

end ConstDemo;

```

4.2.1 常量类型

明示常量可以是 HLA 基本类型和本章讨论的几种复合类型中的任意一种类型。第 1、第 2、第 3 章讨论了大多数基本类型，包括以下几种²：

- boolean 类型的常量(true 或者 false)
- uns8 类型的常量(0~255)
- uns16 类型的常量(0~65535)
- uns32 类型的常量(0~4294967295)
- int8 类型的常量(-128~+127)

² 这个列表并不完整。HLA 还支持 64 位和 128 位数据类型。第 8 章将作讨论。

- int16 类型的常量(-32768 ~ +32767)
- int32 类型的常量(-2147483648 ~ +2147483647)
- char 类型的常量(字符代码在 0~255 内的任意 ASCII 字符)
- byte 类型的常量(整数类型、布尔类型和字符类型中的任意 8 位数值)
- word 类型的常量(任意的 16 位数值)
- dword 类型的常量(任意的 32 位数值)
- real32 类型的常量(浮点数值)
- real64 类型的常量(浮点数值)
- real80 类型的常量(浮点数值)

除了上面的常量类型以外, const 段还支持另外 6 种常量类型:

- 字符串常量
- 文本常量
- 枚举常量
- 数组常量
- 记录/联合常量
- 字符集常量

这些数据类型是本章的主题, 稍后将对它们进行讨论。但是, 字符串常量和文本常量十分重要, 因此很有必要尽快讨论。

4.2.2 字符串和字符字面常量

与大多数的编程语言一样, HLA 对字符序列(字符串)和单个字符进行区分。字面字符和字符串常量在类型声明和语法上都存在差别。这在类型声明以及字面量字符和字符串常量的语法当中都有所体现。到此为止, 本章还没有很好地对字符和字符串字面常量进行区分, 现在是进行区分的时候了。

字符串字面常量是由 ASCII 双引号括起来的 0 个或者多个字符构成的序列。下面都是字面字符串常量的合法示例:

```
"This is a string"    // String with 16 characters.
""                   // Zero length string.
"a"                  // String with a single character.
"123"                 // String of length 3.
```

长度为 1 的字符串与字符常量是不同的。HLA 对字符和字符串使用了两种完全不同的内部表示。因此, “a” 不是字符值, 而是一个只包含一个字符的字符串值。

字符字面常量采用两种形式, 但最常见的是由一个用 ASCII 单引号括起来的单字符组成:

```
'2'                  // Character constant equivalent to ASCII code $32.
'a'                   // Character constant for lowercase 'A'.
```

正如前面所解释的, “a” 和 ‘a’ 是不等价的。

那些熟悉 C、C++ 或者 Java 的人可能认识这些字面常量格式, 因为它们类似于 C/C++/Java 中的字符常量和字符串常量。实际上, 本书已经假定您对 C/C++ 有了一定程度的了解, 因此基于这

一点,例子中用到字符和字符串常量时都没有给出显示的定义。

C/C++字符串与 HLA 字符串的另一个相似点就是可以自动对程序中相邻的字面字符串常量进行连接。例如,HLA 连接两个字符串常量

```
"First part of string," "second part of string"
```

来组成一个字符串常量

```
"First part of string, second part of string"
```

但是除了这些相似点以外,HLA 字符串与 C/C++字符串是不同的。例如,C/C++字符串允许使用由反斜杠字符后面跟着一个或者多个特殊字符组成的转义字符序列来指定特殊的字符值;HLA 不使用这样的转义字符机制。但是 HLA 提供了其他几种方法来将特殊字符插入字符串常量或者字符常量中。

HLA 不允许在字符串字面常量和字符常量当中出现转义字符序列。因此,您要问的第一个问题可能是“如何将双引号嵌入字符串常量当中以及如何将单引号嵌入字符常量当中”。为了解决这个问题,HLA 使用了与 Pascal 和很多其他语言相同的技术:在字符常量中插入两个双引号来表示单个双引号,或者在字符常量中插入两个单引号来表示一个单引号字符,例如:

```
"He wrote a ""Hello World"" program as an example."
```

这等效于:

```
He wrote a "Hello World" program as an example.
```

第1章指出,要创建单引号字符常量,需要将两个邻接的单引号放到一对单引号中:

```
''''
```

HLA 提供了两个特性来消除对转义字符的需求。除了通过对两个邻接的字符串常量进行连接来组成一个较长的字符串常量以外,HLA 还可以对邻接字符常量和字符串常量的任意组合进行连接来组成一个字符串常量:

```
'1' '2' '3' // Equivalent to "123"
"He wrote a " ''' "Hello World" ''' "program as an example."
```

注意,对于 HLA 来说,这些例子中的两个 He wrote 字符串都是相同的。

HLA 还提供了一种定义字符常量的方法来处理所有其他的 C/C++转义字符序列:ASCII 码字面字符常量。这种字符常量形式的语法如下:

```
#integer_constant
```

这种形式创建了一个字符常量,该字符常量的值是由 integer_constant 指定的 ASCII 码。该常量可以是十进制、十六进制或者二进制数值,例如:

```
#13      #$d      %#1101      // All three are the same character,
                                     // a carriage return.
```

由于可以对字符字面量与字符串进行连接, 并且`#constant`形式代表字符字面量, 下面所示的都是合法的字符串:

```
"Hello World" #13 #10 // #13 #10 is the Windows newline sequence
                        // (carriage return followed by line feed)

"Error: Bad Value" #7  // #7 is the bell character.
"He wrote a " #22 "Hello World" #22 "program as an example."
```

由于`$22`是双引号字符对应的ASCII码, 所以最后一个例子是第三种`He wrote`字符串字面量的形式。

4.2.3 `const` 段中的字符串常量与文本常量

`const` 段当中的字符串常量与文本常量采用如下所示的声明语法:

```
const
  AStringConst: string := "123";
  ATextConst:   text   := "123";
```

除了数据类型以外, 这两个常量的声明是一致的。但是, 它们在 HLA 程序中的行为却迥然不同。

只要 HLA 在程序中遇到了符号字符串常量, 它就会用字符串字面常量替换该字符串的名称。所以像 `stdout.put(AStringConst);` 这样的语句就会向显示器打印字符串 123, 这没什么值得惊讶的。

只要 HLA 在程序中遇到了符号文本常量, 它就会用该字符串的文本(而不是字符串字面常量)代替该标识符。也就是说, HLA 用双引号之间界定的字符代替符号文本常量。因此, 在给定了前面声明的前提下, 下面的语句是完全合法的:

```
mov( ATextConst, al ); // Equivalent to mov( 123, al );
```

注意, 在本例当中用 `AStringConst` 替代 `ATextConst` 是不合法的:

```
mov( AStringConst, al ); // Equivalent to mov( "123", al );
```

后面这个例子是不合法的, 因为不能将字符串字面常量移动到 AL 寄存器中。

只要 HLA 在程序中遇到符号文本常量, 它会立即用文本常量的字符串值取代该文本常量并继续进行编译, 就如同在程序中写的是文本常量的值而不是符号标识符。如果在程序中经常输入一些文本序列, 这可以减少输入字符的量, 并且有助于提高程序的可读性。例如, 考虑 HLA `stdio.hhf` 库的头文件中的 `nl` 文本常量的声明:

```
const
  nl: text := "#$d #a"; // Windows version.

const
  nl: text := " "" "" #a"; // Linux, FreeBSD, and Mac OS X version.
```

只要 HLA 遇到符号 `nl`, 它就会立即用字符串“`#$d #a`”的值替代 `nl` 标识符。当 HLA 遇到`#$d`(回车)字符常量后面跟着`#a`(换行)字符常量时, 它就会将其连接起来组成包含 Windows

换行序列(回车之后换行)的字符串。考虑下面两条语句：

```
stdout.put( "Hello World", nl );
stdout.put( "Hello World" nl );
```

注意，上面的第二条语句没有用逗号将字符串字面量和 `nl` 符号分隔开。在第一个例子当中，HLA 给出了打印字符串 `Hello World` 的代码，然后又给出了一些用于打印换行序列的代码。在第二个例子当中，HLA 又对符号 `nl` 进行了如下所示的扩展：

```
stdout.put( "Hello World" #d #a );
```

现在，HLA 看到了一个后面跟着两个字符常量的字符串字面常量(`Hello World`)。它将三者连接起来形成一个字符串，然后用一个调用来打印这个字符串。因此，去掉字符串字面量与 `nl` 符号之间的逗号可以生成更加有效的代码。请记住，这只对字符串字面常量有效。采用这种技术不能对字符串变量进行连接，也不能将字符串变量与字符串字面量进行连接。

Linux、FreeBSD 和 Mac OS X 用户应该注意 Unix 的行末序列只是一个换行字符。因此，在这些操作系统中，`nl` 的声明稍有不同(以始终保证将 `nl` 扩展为字符串常量而不是字符常量)。

在常量段中，如果只定义一个常量标识符和一个字符串常量(也就是说，没有提供类型)，HLA 就会默认为是字符串(`string`)类型。如果想要声明一个文本(`text`)常量，就必须显式地给出类型。

```
const
  AStrConst := "String Constant";
  ATextConst: text := "mov( 0, eax );";
```

4.2.4 常量表达式

至此，本章已经给出了这样一个印象，符号常量定义是由一个标识符、一个可选类型和一个字面常量组成的。实际上，HLA 的常量声明还可以复杂得多，因为 HLA 允许将符号常量赋值为常量表达式，而不只是字面常量。一般的常量声明都采用下面两种形式：

```
Identifier : typeName := constant_expression ;
Identifier :=constant_expression ;
```

常量表达式采用像 C/C++和 Pascal 这些高级语言所用的形式。它们可以包含字面常量值、前面声明过的符号常量以及各种算术运算符。表 4-1 列出了一些在常量表达式中可能出现的运算。

表 4-1 常量表达式中允许使用的运算

算术运算符	
- (一元取反)	对 - 后面的表达式的结果取反
*	将*两边的整数或实数值相乘
div	将左侧的整数操作数除以右侧的整数操作数，得到一个整数(截断后的)结果
mod	将左侧的整数操作数除以右侧的整数操作数，得到一个整数余数
/	将左侧的操作数除以右侧的右侧的操作数，得到一个浮点数结果
+	将两侧的操作数相加

(续表)

	将左侧的操作数减去右侧的操作数
比较运算符	
=、==	比较两侧的操作数。如果两者相等，则返回 true
<>、!=	比较两侧的操作数。如果两者不等，则返回 true
<	如果左侧的操作数小于右侧的操作数，则返回 true
<=	如果左侧的操作数小于等于右侧的操作数，则返回 true
>	如果左侧的操作数大于右侧的操作数，则返回 true
>=	如果左侧的操作数大于等于右侧的操作数，则返回 true
逻辑运算符*	
&	返回两个布尔型操作数的逻辑与
	返回两个布尔型操作数的逻辑或
^	返回两个布尔型操作数的逻辑异或
!	返回!后面的单个操作数的逻辑非
按位逻辑运算符	
&	返回两个整数操作数的按位与
	返回两个整数操作数的按位或
^	返回两个整数操作数的按位异或
!	返回单个整数操作数的按位取反
字符串运算符	
+	返回左侧字符串操作数和右侧字符串操作数的连接结果

*C/C++和 Java 用户要注意，HLA 的常量表达式使用完整布尔求值，而不是短路布尔值求值。因此，HLA 常量表达式的行为与 C/C++Java 表达式有所不同。

常量表达式运算符服从标准的优先级规则；必要时可以使用圆括号改变优先级。运算符之间正确的优先级关系可以参阅 HLA 参考手册(从<http://webster.cs.ucr.edu/>或<http://artofasm.com/>下载)。通常，如果优先级不明显，可以使用圆括号正确地描述求值的顺序。实际上，HLA 提供了更多的运算符，上面给出的只是使用最普遍的；HLA 文档提供了一个常量表达式运算符的完整列表。

如果常量表达式中出现了一个标识符，那么该标识符必须是已经在程序中的常量段或者 var 段定义过的常量标识符。不可以在常量表达式中使用变量标识符；当编译时 HLA 对该常量表达式求值时，它们的值还没有定义。而且，也不要对编译时和运行时进行的操作产生混淆：

```
// Constant expression, computed while HLA is compiling your program:
```

```
const
    x          := 5;
    y          := 6;
    Sum        := x + y;
```



```
// Run-time calculation, computed while your program is running, long after
// HLA has compiled it:

    mov( x, al );
    add( y, al );
```

在编译的时候,HLA 直接对常量表达式的值进行解释。它不用任何机器指令对上述常量表达式中的 $x+y$ 进行计算,而是直接计算这两个常量的和。从程序中的那个点开始,HLA 就将数值 11 与常量 Sum 相联系,就好像程序已经包含了语句 $\text{Sum}:=11$;而不是 $\text{Sum}:=x+y$ 。另一方面,HLA 不会为上面提到的 mov 和 add 指令预先计算寄存器 AL 中的数值 11;当程序运行时(有时是在编译完成以后),它如实地为这两条指令产生目标代码,并且 80x86 对它们的和进行计算。

一般地,在汇编语言程序当中,常量表达式并不会非常复杂。通常都是对两个整数值进行加法、减法或者乘法操作。例如,下面的 const 段定义了一组数值连续的常量:

```
const
    TapeDAT      := 0;
    Tape8mm      := TapeDAT + 1;
    TapeQIC80     := Tape8mm + 1;
    TapeTravan   := TapeQIC80 + 1;
    TapeDLT      := TapeTravan + 1;
```

上面的常量具有下面的数值: $\text{TapeDAT} = 0$ 、 $\text{Tape8mm} = 1$ 、 $\text{TapeQIC80} = 2$ 、 $\text{TapeTravan} = 3$ 、 $\text{TapeDLT} = 4$ 。

4.2.5 HLA 程序中的多个 const 段以及它们的顺序

虽然 const 段必须出现在 HLA 程序的声明段中(例如,出现在 `program pgmname;` 头和相应的 `begin pgmname;` 语句之间),但它们不一定要出现在声明段中的其他任意一项之前或之后。事实上,就像变量声明段一样,可以在一个声明段当中放置多个 const 段。关于 HLA 常量声明的唯一限制就是对于任意一个常量符号,必须在程序中使用到它之前进行声明。

例如,某些 C/C++ 程序员更喜欢采用下面的形式对常量进行声明(因为这样更贴近于 C/C++ 中声明常量的语法):

```
const    TapeDAT      := 0;
const    Tape8mm      := TapeDAT + 1;
const    TapeQIC80     := Tape8mm + 1;
const    TapeTravan   := TapeQIC80 + 1;
const    TapeDLT      := TapeTravan + 1;
```

在程序中放置 const 段对于程序员来说似乎是个人问题。除了需要在使用之前对所有的变量进行定义以外,可以随意将常量声明段插入到声明段的任意位置。

4.2.6 HLA 的 val 段

在 const 段定义的常量数值是不能改变的。虽然这似乎很合理(毕竟常量就应该是不变的),但是我们有很多种不同的方法都可以用于定义术语“常量”,const 对象只遵从明确定义的规则。HLA 的 val 段允许根据不同的规则定义常量对象。本小节将讨论 val 段以及 val 常量和 const 常量的区别。

“const-ness(常量性)”的概念可以在两种不同的时候存在：当 HLA 对程序进行编译时和程序执行时(HLA 不再运行)。常量所有合理的定义要求，在程序运行时数值不能改变。“常量”的数值在编译时是否可以改变是另外的事情。HLA 的 `const` 对象与 `val` 对象之间的区别在于常量的值在编译时是否发生改变。

只要在 `const` 段定义了常量，该常量的值在运行过程中和 HLA 在对程序进行编译的过程中就都不变了。因此，像 `mov(SymbolicCONST, eax);` 这样一条指令总是将同样的值移到寄存器 EAX 中去，而不必考虑该指令出现在 HLA 主程序中的什么位置。只要在 `const` 段定义了符号 `SymbolicCONST`，此符号从该点开始往后就具有相同的值了。

与 `const` 段相同，HLA 的 `val` 段允许声明符号常量。但是，HLA 的 `val` 常量在程序的整个源代码当中可以改变它们的值。下面的 HLA 声明是完全合法的：

```

val      nitialValue      := 0;
const    SomeVal          := InitialValue + 1;    // = 1
const    AnotherVal       := InitialValue + 2;    // = 2

val      InitialValue     := 100;
const    ALargerVal       := InitialValue;        // = 100
const    LargeValTwo      := InitialValue*2;      // = 200

```

出现在 `const` 段的所有符号都将符号数值 `InitialValue` 作为定义的一部分。但是要注意，`InitialValue` 在该代码序列中不同的地方具有不同的值；在代码序列的开头，`InitialValue` 的值为 0，后来其值又是 100。

请记住 `val` 对象在运行时不是变量；它还是一个明示常量，HLA 将用 `val` 标识符的当前值替代该标识符³。像 `mov(25, InitialValue);` 这样的语句与 `mov(25, 0);` 或 `mov(25, 100);` 一样都不合法。

4.2.7 在程序中的任意位置修改 `val` 对象

如果在声明段对所有的 `val` 对象进行声明，那么看上去就不能改变程序中 `begin` 与 `end` 之间任意一个 `val` 对象的值。毕竟，`val` 段只能出现在程序的声明段中，声明段结束于 `begin` 语句之前。在第 9 章将会看到对 `val` 对象进行的大多数修改都发生在 `begin` 与 `end` 语句之间。因此，HLA 必须提供某种方法在声明段之外改变 `val` 对象的值。完成该任务的机制就是“?”操作符。HLA 不仅允许改变出现在声明段之外的 `val` 对象的值，还允许改变程序中任意位置的 `val` 对象的值。HLA 程序中允许出现空格符的地方都可以插入如下形式的语句：

```
? ValIdentifier := constant_expression;
```

这意味着可以编写一个像程序清单 4-4 所示的简短程序。

程序清单 4-4 用操作符“?”重新定义 `val` 对象

```

program VALdemo;
#include( "stdlib.hhf" )

val

```

³ 在这里，“当前(current)”的意思是源代码中上一次赋给 `val` 对象的值。


```

    NotSoConstant := 0;

begin VALdemo;

    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 10;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 20;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 30;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

end VALdemo;

```

4.3 HLA 的 type 段

假如您不喜欢 HLA 在声明字节、字、双字、实数以及其他变量时所使用的名称，而是喜欢 Pascal 或者 C 的命名方式，想使用 `integer`、`float`、`double` 或者其他的术语。如果 HLA 是 Pascal，那么就可以在 `type` 段对这些名称重新进行定义。对于 C 语言，可以使用 `#define` 或者 `typedef` 语句来定义。与 Pascal 一样，HLA 也有它自己的 `type` 语句，允许为这些名称创建别名。下面的例子展示了如何在 HLA 程序中设置一些与 C/C++/Pascal 兼容的名称：

```

type
    integer:    int32;
    float:      real32;
    double:     real64;
    colors:     byte;

```

现在可以使用意义更加明显的语句来声明变量，例如：

```

static
    i:          integer;
    x:          float;
    HouseColor: colors;

```

如果您是一个 Ada、C/C++ 或者 FORTRAN(或者是其他的语言)程序员，可以选择您个人认为更加习惯的类型名。当然，这丝毫不会影响 80x86 或者 HLA 对这些变量所作出的响应，但是这确实会有助于编写更容易阅读和理解的程序，因为类型名更能表示实际的类型。C/C++ 程序员应该注意：不要兴奋得昏了头而定义一个 `int` 数据类型。`int` 是 80x86 中的一条机器指令(中断)，因此它在 HLA 中是保留字。

`type`段不是只对创建类型同构(也就是给已经存在的类型取一个新的名称)有用。在后面几节将会看到, 在 `type` 段中还可以做许多事情。

4.4 enum 和 HLA 枚举数据类型

前面讨论常量和常量表达式时, 有如下所示的例子:

```

const      TapeDAT      :=      0;
const      Tape8mm      :=      TapeDAT + 1;
const      TapeQIC80     :=      Tape8mm + 1;
const      TapeTravan    :=      TapeQIC80 + 1;
const      TapeDLT       :=      TapeTravan + 1;

```

这个例子展示了如何使用常量表达式创建一组包含唯一的、连续值的常量。然而, 这个方法存在两个问题。首先, 敲键盘的工作量很大(以及在回顾程序的时候需要大量的阅读时间)。其次, 在创建大量只具有唯一值的常量, 以及重用或者跳过一些数据的时候, 很容易出错。HLA 的 `enum` 类型提供了一个较好的方法创建一系列具有唯一值的常量。

`enum` 是一个 HLA 类型声明, 能够将一串名称与一个新的类型关联起来。HLA 将为每一个名称关联一个唯一的数据值(也就是说, 它枚举了列表)。在通常情况下, `enum` 关键字在 `type` 段中出现, 使用方法如下:

```

type
    enumTypeID:      enum { comma_separated_list_of_names };

```

符号 `enumTypeID` 变成了一种新的类型, 它的值是由一组特定的名称来指定的。作为一个例子, 考虑数据类型 `TapeDrives` 和相应的 `TypeDrives` 类型的变量声明:

```

type
    TapeDrives: enum{TapeDAT, Tape8mm, TapeQIC80, TapeTravan, TapeDLT};
static
    BackupUnit:      TapeDrives := TapeDAT;
    .
    .
    .
    mov( BackupUnit, al );
    if( al = Tape8mm ) then
        ...
    endif;
    //etc.

```

默认情况下, HLA 为枚举数据类型保留一个字节的存储空间。所以 `BackupUnit` 变量将占用

一个字节的存储区, 可以用一个 8 位的寄存器访问⁴。对于常量来说, HLA 将从 0 开始的连续的 uns8 常量值与每一个枚举标识符关联起来。在 TapeDrives 的例子中, 磁带驱动标识符含有的数值是: TapeDAT=0、Tape8mm=1、TapeQIC80=2、TapeTravan=3、TapeDLT=4。可以在程序中使用这些常量, 就好像已经在 const 段中将它们定义为这些数值一样。

4.5 指针数据类型

也许您已经在 Pascal、C 或者 Ada 程序语言中与指针有过直接的接触, 并且可能现在已经开始担心了。几乎所有人在高级语言中第一次碰到指针的时候都会遇到麻烦。不过不要惧怕它! 事实上, 汇编语言中的指针要容易处理一些。而且, 所遇到的大多数与指针相关的问题很可能都是与指针本身无关的, 而是与使用指针实现的链表和树数据结构有关。另一方面, 指针在汇编语言中的应用有许多都与链表、树, 以及其他难以处理的数据结构无关。一些简单的数据结构也经常使用指针, 如数组、记录等。因此如果您对指针感到很恐惧, 那么就忘掉一切与它们相关的事情, 而在后面将了解到指针是多么重要。

学习指针最好的起点可能是指针的定义。到底什么是指针呢? 遗憾的是, 高级语言(例如 Pascal)往往将指针的简单性隐藏在抽象之中。这样大大增加了复杂性(但这样做的理由也很充分), 往往会吓倒程序员, 因为他们不知道究竟会发生什么。

如果您害怕指针, 那就暂时忘记它们, 来看一下数组。考虑如下所示的 Pascal 数组声明:

```
M: array [0..1023] of integer;
```

即使不了解 Pascal, 这里的概念也非常容易理解。M 是一个由 1024 个整数组成的数组, 从 M[0]到 M[1023]。该数组中的每一个元素都可以具有一个与其他所有元素相互独立的数值。换句话说, 该数组提供了 1024 个不同的整数变量, 其中的每一个都按编号(数组索引)而不是按名称来引用。

如果遇到一个带有语句 M[0]:=100;的程序, 那么也许根本就不必去思考这条语句会发生什么。它将数值 100 存储到数组 M 的第一个元素当中。现在考虑一下如下所示的两条语句:

```
i := 0; (*Assume "i" is an integer variable.*)
M[i] := 100;
```

这两条语句完成了与 M[0]:=100;相同的操作。事实上, 可以使用范围 0~1023 内的任一整数作为这个数组的索引。如下所示的语句所执行的操作与我们直接给索引为 0 的项赋值的操作相同:

```
i := 5; (*Assume all variables are integers.*)
j := 10;
k := 50;
m[i*j-k] := 100;
```

“那么意义何在?” 您可能会想, “产生 0~1023 之间的整数的任意操作都是合法的。那又怎

⁴HLA 提供了一个机制, 使用这个机制可以指定枚举数据类型占用 2 个或者 4 个字节的存储空间。参阅 HLA 文档可以了解到更加详细的信息。

么样？”来看下面的代码：

```
M [1] := 0;
M [M [1]] := 100;
```

现在花点时间来消化一下这些知识。但是，如果慢慢想一想就会发现，其实这两条指令执行了您一直在进行的同样的操作。第一条语句将 0 存储到数组元素 M[1]中。第二条语句取出 M[1] 的值，这是一个整数，可以用它作为 M 的一个数组索引，并使用这个数值(0)来控制应该将数值 100 存储到哪里。

如果您认为上面所说的合理，那么就说明您已经理解了指针，因为 M[1]就是指针。当然，严格意义上并不能这么说，但是假设将 M 变成“存储器”并将这个数组看作存储器，这就正好是指针的定义。指针只是一个存储单元，它所存储的值是另一个存储单元的地址(或者索引)。指针在汇编语言程序中很容易声明和使用，甚至不用担心数组索引或者其他类似的事情。

4.5.1 在汇编语言中使用指针

HLA 指针是一个可以包含其他变量地址的 32 位数值。如果有一个数值为\$1000_0000 的双字变量 p，那么 p“指向”存储单元\$1000_0000。为了访问 p 指向的双字，可以使用如下所示的代码：

```
mov( p, ebx );           // Load ebx with the value of pointer p.
mov( [ebx], eax );       // Fetch the data that p points at.
```

通过将 p 的数值装载到 EBX 中，这段代码将数值\$1000_0000 装载到了 EBX 中(假设 p 的数值是\$1000_0000，也就是指向存储单元\$1000_0000)。上述的第二条指令将偏移量从 EBX 中数值的位置开始的双字装载到 EAX 寄存器中。因为 EBX 现在的值是\$1000_0000，则这条指令将把从位置\$1000_0000 开始到\$1000_0003 结束的数值装载到 EAX 当中。

为什么不使用类似于 mov(mem,eax); 的指令直接从存储单元\$1000_0000 装载 EAX (假设 mem 的地址是\$1000_0000)呢？原因有许多，但是首要的原因是这条指令将总是从位置 mem 处装载 EAX。该地址无法修改。然而前面的指令总是从 p 所指的位置向 EAX 装载数据，这在程序的控制下是很容易改变的。实际上，简单的指令 mov(&mem2, p); 将使上述两条指令在下一次运行时从 mem2 向 EAX 装载数据。考虑如下所示的指令序列：

```
mov( &i, p );           // Assume all variables are STATIC variables.
.
.
.
if( some_expression ) then
    mov( &j, p );        // Assume the code above skips this instruction
                        // and you get to the next instruction by
                        // jumping to this point from somewhere else.
    .
    .
endif;
mov( p, ebx );           // Assume both of the above code paths
mov( [ebx ], eax );      // wind up down here.
```

这个简短的例子展示了程序中的两条执行路径。第一条路径将变量 *i* 的地址赋给 *p*。第二条路径将变量 *j* 的地址赋给 *p*。这两条路径在最后两条 `mov` 指令处汇合, 这两条指令根据执行的是哪一条路径将 *i* 或者 *j* 装载到 `EAX` 中。这个过程在许多方面都与高级语言(例如 `Pascal`)中过程的参数相似。执行同样的指令来访问不同的变量, 这取决于 *p* 中的数值是哪一个(*i* 或者 *j*)的地址。

4.5.2 在 HLA 中声明指针

因为指针的长度是 32 位, 所以只需要简单地使用双字类型为指针分配存储区。然而, 还有另外一个更好的方法: HLA 专门为声明指针变量提供了短语 `pointer to`。考虑如下所示的例子:

```
static
    b:      byte;
    d:      dword;
    pByteVar: pointer to byte := &b;
    pDWordVar: pointer to dword := &d;
```

这个例子表明, 在 HLA 中不仅可以声明指针变量, 还可以对它们进行初始化。注意, 只能使用取址操作符获取静态变量(`static`、`readonly` 以及 `storage` 对象)的地址, 因此, 指针变量就只能使用静态对象的地址进行初始化。

还可以在 HLA 程序的 `type` 段定义自己的指针类型。例如, 如果经常使用指针指向字符, 可能会想到使用如下所示的 `type` 声明:

```
type
    ptrChar: pointer to char;

static
    cString: ptrChar;
```

4.5.3 指针常量和指针常量表达式

HLA 允许两种字面指针常量形式: 后面跟有静态变量名称的取址操作符或者常量 `NULL`。除了这两种字面指针常量外, HLA 还支持简单的指针常量表达式。

`NULL` 指针指向常量 0。0 是一个非法地址, 在现代操作系统中, 如果试图访问这种地址, 就会引发异常。一般情况下, 程序都将指针初始化为 `NULL`, 表示指针还没有使用有效地址显式初始化。

除了简单的地址字面量和数值 0 以外, HLA 还允许在指针常量合法的任何地方使用非常简单的常量表达式。指针常量表达式可以采用如下所示的 3 种形式:

```
&StaticVarName [ PureConstantExpression ]
&StaticVarName + PureConstantExpression
&StaticVarName - PureConstantExpression
```

PureConstantExpression 表示一个不包含任何指针常量的数字常量表达式。这种类型的表达式会得到一个内存地址, 该地址指向 *StaticVarName* 之前或者之后(“-”或者“+”)指定数量的字节处。注意前两种形式在语法上是等价的: 它们都返回一个指针常量, 这个常量的地址是静态变量和常量表达式的和。

因为可以创建指针常量表达式，所以 HLA 允许在 `const` 段中定义明示指针常量也就不足为奇了。程序清单 4-5 展示了该如何进行。

程序清单 4-5 HLA 程序中的指针常量表达式

```
program PtrConstDemo;
#include( "stdlib.hhf" );

static
    b: byte := 0;
    byte 1, 2, 3, 4, 5, 6, 7;

const
    pb := &b + 1;

begin PtrConstDemo;

    mov( pb, ebx );
    mov( [ebx], al );
    stdout.put( "Value at address pb = $", al, nl );

end PtrConstDemo;
```

该程序一执行就打印出 `b` 后面的那个字节的值(其中存储了数值\$01)。

4.5.4 指针变量和动态内存分配

指针变量是存储 HLA 标准库函数 `mem.alloc` 的返回值的最佳地方。`mem.alloc` 函数将它分配的存储区的地址返回给 `EAX` 寄存器；因此，可以在调用 `mem.alloc` 之后，马上使用一个 `mov` 指令将地址直接存储到一个指针变量中：

```
type
    bytePtr:    pointer to byte;
var
    bPtr: bytePtr;
    .
    .
    .
    mem.alloc( 1024 );    // Allocate a block of 1,024 bytes.
    mov( eax, bPtr );    // Store address of block in bPtr.
    .
    .
    .
    mem.free( bPtr );    // Free the allocated block when done using it.
    .
    .
    .
```

4.5.5 指针的常见问题

在使用指针的时候，程序员可能会碰到 5 种常见的错误。其中的某些错误可能会导致程序立

刻停止运行并弹出一个诊断信息；其他的错误更加细微，只产生错误的结果，而不报告任何错误，或者只是影响程序的性能而不显示错误。这 5 种错误如下所示：

- 使用没有初始化的指针
- 使用具有非法数值的指针(例如 NULL)
- 在存储区释放后仍然使用 `mem.alloc` 分配的存储区
- 在程序使用完后没有使用 `mem.free` 释放存储区
- 使用错误的数据类型访问间接数据

上述第一个问题是，在还没有给指针赋予有效的存储地址之前就使用该指针变量。初学者通常都没有认识到，声明指针变量只为指针本身保留存储区，而并不为指针指向的数据保留存储区。程序清单 4-6 演示了这个问题。

程序清单 4-6 未初始化指针的示例

```
// Program to demonstrate use of
// an uninitialized pointer.Note
// that this program should terminate
// with a Memory Access Violation exception.

program UninitPtrDemo;
#include( "stdlib.hhf" );

static

    // Note: By default,variables in the
    // static section are initialized with
    // zero(NULL)hence the following
    // is actually initialized with NULL,
    // but that will still cause our program
    // to fail because we haven't initialized
    // the pointer with a valid memory address.

    Uninitialized: pointer to byte;

begin UninitPtrDemo;

    mov( Uninitialized, ebx );
    mov( [ebx ], al );
    stdout.put("Value at address Uninitialized:=$", al, nl );

end UninitPtrDemo;
```

虽然静态段中所声明的变量已经被初始化了，但是静态初始化仍然不会为程序中的指针分配一个有效的地址(它将指针初始化为 0，也就是 NULL)。

当然，在 80x86 中不存在真正未被初始化的变量。我们真正具有的变量是已经被显式初始化的变量，和当为变量分配存储空间时恰好继承了存储器中位模式的变量。许多时候，这些废弃的位模式并不对应有效的存储地址。想要废除这样的指针(也就是访问它所指向的存储单元中的数据)，通常会引起内存访问违规(Memory Access Violation)异常。

然而，有时候这些随机位恰好与可以访问的有效内存地址对应。这种情况下，CPU 会访问特

定的存储区而不中断程序。尽管对于初级程序员来说，这似乎比终止程序更可取，但事实上这样会使情况更糟，因为错误的程序将会继续运行而不警告所出现的问题。如果使用一个没有初始化的指针存储数据，很可能会覆盖存储器中其他非常重要的数据。这些错误会在程序中造成很难定位的问题。

第二个关于指针的问题是向指针中存储无效的地址值。前面描述的第一个问题实际上是第二个问题的一个特例(它是由存储器中的废弃位提供无效地址，而不是由于计算错误产生的)。影响是相同的：如果想要废除一个保存了无效地址的指针，那么就会产生内存访问违规异常，或者会访问一个未知的存储区。

第三个问题就是悬空指针问题。为了理解这个问题，来看如下所示的代码段：

```

mem.alloc(-256 );    // Allocate some storage.
mov( _eax, ptr );    // Save address away in a pointer variable.
.
.                    // Code that use the pointer variable ptr.
.
mem.free(ptr );      // Free the storage associated with ptr.
.
.                    // Code that does not change the value in ptr.
.
mov( ptr, ebx );
mov( al, [ebx] );

```

在这个例子中，程序分配了 256 个字节的存储区，并将存储区的地址存储到 ptr 变量中。代码使用完这 256 个字节后释放这部分存储区，还给系统以作他用。注意，对 mem.free 的调用并没有对 ptr 的值进行任何改变：ptr 仍然指向之前由 mem.alloc 分配的存储空间。实际上，mem.free 没有改变存储块中的任何数据，所以在从 mem.free 返回的时候，ptr 仍然指向存储到这个空间中的数据。然而要注意，对 mem.free 的调用告诉系统，程序已经不需要这 256 个字节的存储区了，系统可以将它用作其他用途。mem.free 函数并没有强调这样一个事实，那就是将永远不能再访问这些数据。程序员也只是简单地承诺，不会再访问它们。当然上述代码段打破了这样的承诺。可以看到在最后两条指令中，程序取走了 ptr 中的数据，并且访问它所指向的存储器中的数据。

悬空指针的最大问题是很多时候使用它们不会带来太大的问题。只要系统不再重新使用已经释放的存储区，使用悬空指针不会对程序造成影响。然而，在继续调用 mem.alloc 的时候，系统可能会决定重新使用先前由 mem.free 调用所释放的存储区。当出现这种情况时，试图废除悬空指针可能导致不希望看到的结果。可能会出现的问题包括：读取已经被覆盖的数据(被新的合法数据覆盖)、覆盖新的数据(最糟糕的情况)、覆盖系统的堆管理指针(这样做会导致程序崩溃)。解决方法是很明显的：一旦释放了指针所指向的存储空间以后，就不要再使用该指针值了。

在所有的问题当中，第四个问题(没有释放存储位置)可能对程序正确操作的影响最小。下面的程序代码展示了这个问题：

```

mem.alloc( 256 );
mov( _eax, ptr );
.
.                    // Code that uses the data where ptr is pointing.
.                    // This code does not free up the storage.

```

```

// associated with ptr.
mem.alloc( 512 );
mov( eax, ptr );

// At this point, there is no way to reference the original
// block of 256 bytes pointed at by ptr.

```

在这个例子中, 程序分配了 256 个字节的存储区, 并使用 `ptr` 变量指向这个存储区。后来程序又分配了另外一个存储区并用这个新的存储区的地址覆盖 `ptr` 中的数据。注意, `ptr` 中旧的数据就丢失了。因为程序不再拥有这个地址值, 也就不可能调用 `mem.free` 释放存储区以作他用。结果这部分存储区对于程序来说是不可访问的。当然, 256 个字节的存储块不能访问, 可能对程序来说并不是一个严重的问题, 但是假设这段代码是在一个循环中, 并一次一次地重复。循环的每一次执行都会丢失 256 个字节的存储区。在重复了足够多的循环迭代后, 程序会用完堆上的所有存储区。这个问题通常称为内存泄漏(memory leak), 因为这就好像在程序运行过程中存储单元从计算机中泄漏出去了一样(使得可用的存储区越来越小)。

内存泄漏远不如悬空指针所造成的危害大。事实上, 内存泄漏只会带来两个问题: 用尽堆空间(最终会导致程序退出, 尽管这很少见); 由于虚拟内存页交换所导致的性能问题。尽管如此, 仍然必须养成习惯, 在用完存储区后释放它们。当程序退出时, 操作系统收回所有的存储空间, 包括内存空间泄漏的数据。因此内存泄漏仅仅是对单个的程序, 而不会对整个系统造成泄漏。

指针的最后一个问题缺少类型安全的访问。这种情况的发生是因为 HLA 并不强制进行指针类型检查。例如, 考虑程序清单 4-7 中的程序。

程序清单 4-7 类型不安全的指针访问示例

```

// Program to demonstrate use of
// lack of type checking in pointer
// accesses.

program BadTypePtrDemo;
#include( "stdlib.hhf" );

static
    ptr:    pointer to char;
    cnt:    uns32;

begin BadTypePtrDemo;

    // Allocate sufficient characters
    // to hold a line of text input
    // by the user:

    mem.alloc( 256 );
    mov( eax, ptr );

    // Okay, read the text a character
    // at a time by the user:

    stdout.put( "Enter a line of text:" );
    stdin.flushInput();

```

```

mov( 0, cnt );
mov( ptr, ebx );
repeat

    stdin.getc();           // Read a character from the user.
    mov( al, [ebx] );       // Store the character away.
    inc( cnt );             // Bump up count of characters.
    inc( ebx );             // Point at next position in memory.

until( stdin.eoln());

// Okay, we've read a line of text from the user,
// now display the data:

mov( ptr, ebx );
for( mov( cnt, ecx ); ecx > 0; dec( ecx )) do

    mov( [ebx], eax );
    stdout.put( "Current value is $", eax, nl );
    inc( ebx );

endfor;
mem.free( ptr );

end BadTypePtrDemo;

```

这个程序以字符值的形式从用户处读入数据，并将该数据显示为双字类型的十六进制数据。尽管汇编语言的一个强大特征是可以自由地忽略数据类型和自动强制转换数据类型，但是这种功能有利有弊。如果使用错误的数据类型访问间接数据，HLA 和 80x86 不会发现这个错误，而程序会得到不精确的结果。因此，在使用指针和间接访问的时候要小心，一定要使所使用数据的数据类型保持一致。

4.6 复合数据类型

复合数据类型，也可以称为聚合数据类型，是那些基于其他(通常是标量)数据类型建立的数据类型。本章将介绍几个比较重要的复合数据类型：字符串、字符集、数组、记录，以及联合。字符串是复合数据类型中一个很好的例子：它是由独立的字符序列和其他一些数据组成的数据结构。

4.7 字符串

除了整型数以外，字符串可能是在现代程序中应用最广泛的数据类型。80x86 也确实支持许多字符串指令，但是这些指令实际上是针对存储块的操作，并不是字符串专用的。因此，本节将主要集中在 HLA 对字符串的定义上，以及讨论 HLA 标准库中的字符串处理例程。

一般来说，字符串是具有如下两个主要属性的由 ASCII 字符组成的序列：长度和字符数据。不同的语言使用不同的数据结构表示字符串。为了更好地理解 HLA 字符串，有必要看一下高级

语言中普遍使用的两种字符串表示方法。

毫无疑问,以 0 结尾的字符串(zero-terminated string)可能是现在使用得最普遍的字符串表示方式,因为这是 C、C++、C#、Java,以及其他一些语言中所使用的字符串表示方法。以 0 结尾的字符串由 0 或者更多的 ASCII 字符组成,它以含有 0 的字节结束。例如,在 C/C++ 中,字符串“abc”需要四个字节:三个字符 a、b、c,后面跟着一个包含 0 的字节。很快我们就会看到,HLA 字符串对以 0 结尾的字符串是向上兼容的,但是应该注意到,在 HLA 中创建以 0 结尾的字符串是很容易的。最容易的做法是在静态段中使用如下所示的代码:

```
static
    zeroTerminatedString: char; @nostorage;
                           byte "This is the zero-terminated string", 0;
```

记住,当使用@nostorage 选项时,HLA 并不为变量保留任何空间,所以 zeroTerminatedString 变量在存储区中的地址对应于下面 byte 指令中的第一个字符。每当有字符串出现在 byte 指令中时,HLA 就会将字符串中的每一个字符放置在连续的存储空间中。字符串最后的 0 值作为字符串的结束。

HLA 支持 zstring 数据类型。然而,这些对象是双字指针,包含一个 zstring 的地址,而不是以 0 结尾的字符串本身。下面是 zstring 声明(和静态初始化)的一个例子:

```
static
    zeroTerminatedString: char; @nostorage;
                           byte "This is the zero-terminated string", 0;
    zstrVar:               zstring := &zeroTerminatedString;
```

以 0 结尾的字符串具有两个基本的属性:它们实现起来非常简单,并且字符串的长度任意。另外一方面,以 0 结尾的字符串也有一些缺点。首先,以 0 结尾的字符串不能包含 NUL 字符(其 ASCII 值为 0)。一般来讲这并不是问题,但是有时候确实会造成很大的麻烦。以 0 结尾的字符串存在的第二个问题是对它们进行的许多操作的效率都比较低。例如,要计算以 0 结尾的字符串的长度,就必须搜索整个字符串以寻找值为 0 的字节(计算字符的个数直到遇见 0 为止)。如下所示的程序段展示了如何计算上述字符串的长度:

```
mov( &zeroTerminatedString, ebx );
mov( 0, eax );
while( ( type byte [ebx+eax] ) <> 0 ) do

    inc( eax );

endwhile;

// String length is now in eax.
```

从这段代码中可以看出,计算字符串长度所需的时间与它的长度成正比;如果字符串比较长,则计算长度的时间也会变得更长。

第二种字符串形式,也就是以长度作为前缀的字符串(length-prefixed string),克服了以 0 结尾的字符串的一些缺点。以长度作为前缀的字符串在 Pascal 语言中是很普遍的。它们通常包含一个长度字节,后面是 0 个或者多个字符值。第一个字节指定了字符串的长度,剩下的字节(直到指定

的长度为止)是字符数据本身。在以长度作为前缀的模式下,字符串“abc”将包含四个字节,\$03(字符串的长度)的后面跟上 a、b 和 c。可以用如下的 HLA 代码创建以长度作为前缀的字符串:

```
static
    lengthPrefixedString: char; @nostorage;
    byte 3, "abc";
```

提前计算字符的个数并将它们插入 byte 语句中是比较繁琐的。幸运的是,有方法使得 HLA 可以自动计算字符串的长度。

以长度作为前缀的字符串克服了以 0 结尾的字符串的两个重要缺陷。在以长度作为前缀的字符串中可以插入 NUL 字符,那些对以 0 结尾的字符串运行效率低下的操作(例如计算字符串长度)对于以长度作为前缀的字符串的效率却非常高。然而,以长度作为前缀的字符串也有两个缺点。以长度作为前缀的字符串最主要的缺点是它们的长度最长只能是 255 个字符(假定长度前缀占 1 个字节)。

HLA 对字符串使用了一种扩展的模式,即同时向上兼容以 0 结尾的字符串和以长度作为前缀的字符串。HLA 字符串同时具有以 0 结尾的字符串和以长度作为前缀的字符串的优点,却克服了它们的缺点。实际上与其他字符串相比,HLA 字符串唯一的缺点就是它占用了一些额外的字节(与以 0 结尾的字符串和以长度作为前缀的字符串相比,HLA 字符串占用的额外字节通常是 9~12 个。这些开销主要是在实际字节的开头和结尾所需要的字节数)。

一个 HLA 字符串包含 4 个部分。第一个部分是一个双字的数值,它指定了字符串可以包含的最大字符数。第二个部分也是一个双字数值,指定了字符串的当前长度。第三个部分是字符串中的一系列字符。最后一个部分是一个 0 结束字节。可以用如下代码在静态段中创建一个 HLA 兼容的字符串⁵:

```
static
    align(4);
    dword 11;
    dword 11;

    TheStrng: char; @nostorage;
    byte "Hello there";
    byte 0;
```

注意,与 HLA 字符串相关的地址是第一个字符的地址,而不是最大长度或者当前长度值。

您可能会问:“那么字符串的当前长度和最大长度之间有什么区别呢?”对于字面字符串来说,它们通常是等价的。然而,如果在运行过程中为字符串变量分配存储空间,那么必须指定能够加入到字符串中的最多字符的个数。当字符串中存储实际的字符串数据时,存储的字符数必须小于或者等于这个最大值。如果超过了这个最大长度,HLA 标准库字符串例程将会产生一个异常(C/C++和Pascal不会这样做)。

HLA 字符串最后的 0 结束字节使您在认为确实高效或者方便的情况下,能够像对待以 0 结尾的字符串一样对待 HLA 字符串。例如,对 Windows、Mac OS X、FreeBSD 和 Linux 的大多数调用都需要以 0 结尾的字符串作为它们的字符串参数。在 HLA 字符串的最后放置一个 0 能够确保

⁵ 实际上,HLA 字符串在存储器中的放置是有限制的。本书不会讨论这个问题,查阅 HLA 文档可获得更详细的信息。

与操作系统以及其他一些使用以 0 结尾的字符串的库模块相兼容。

4.8 HLA 字符串

像前面提到的一样, HLA 字符串包含 4 个部分: 最大长度、当前长度、字符数据和一个 0 结束字节。然而, HLA 从不需要手动输入这些部分来创建字符串数据。HLA 非常智能, 当它看到字符串字面常量时, 就能够自动创建这个数据。因此如果要像下面这样使用字符串常量, 那么就必须清楚 HLA 将会在存储器当中创建由 4 个部分组成的字符串:

```
stdout.put("This gets converted to a four-component string by HLA");
```

事实上, HLA 并不直接对前面所描述的字符串数据进行操作。当 HLA 看到一个字符串对象时, 它通常使用指针对这个对象进行操作, 而不是直接对该对象进行操作。毫无疑问, 这是关于 HLA 字符串最重要的一点, 也是 HLA 编程初学者在处理 HLA 字符串时所遇到的最大难题: 字符串是指针! 与指针相同, 一个字符串变量要占用 4 个字节(因为它就是一个指针)。上面说了这么多, 现在让我们来看一个简单的声明字符串变量的例子:

```
static
    StrVariable:    string =
```

因为字符串变量是指针, 所以必须在使用它之前对它进行初始化。一般可以使用 3 种方法来初始化具有合法字符串地址的字符串变量: 使用静态初始值设定项、使用 `str.alloc` 例程, 或者调用其他用于初始化字符串或返回一个指向字符串的指针的 HLA 标准库函数。

在允许已初始化变量的静态声明段(`static` 和 `readonly`)中, 可以使用标准的初始化语法来对字符串变量进行初始化, 例如:

```
static
    InitializedString: string := "This is my string";
```

注意, 这不是使用字符串数据来初始化字符串变量。HLA 在一个特殊的隐藏存储器段中创建了字符串数据结构(参见 4.7 节), 并且使用字符串中第一个字符(This 中的 T)的地址初始化 `InitializedString` 变量。记住, 字符串是指针! HLA 编译器将实际的字符串数据存储在只读存储器段中。因此, 在运行时不能修改这个字符串中的字符。然而因为字符串变量(切记, 它是指针)处于静态段中, 所以可以改变字符串变量, 使之指向不同的字符串数据。

因为字符串变量是指针, 所以可以将其值装载到 32 位的寄存器当中。指针本身指向字符串的第一个字符。可以从这个地址向前 4 个字节的地方发现双字的当前字符串长度, 从这个地址向前 8 个字节的地方发现双字的最长字符串长度。程序清单 4-8 展示了访问这个数据的一种方法⁶。

⁶ 注意, 不推荐使用这种模式。如果需要从字符串中提取长度信息, 可以使用 HLA 字符串库中提供的例程来完成这项任务。

程序清单 4-8 访问字符串的长度域和最大长度域

```
// Program to demonstrate accessing Length and Maxlength fields of a string.

program StrDemo;
#include( "stdlib.hhf" );

static
    theString:string := "String of length 19";

begin StrDemo;

    mov( theString, ebx );    // Get pointer to the string.

    mov( [ebx-4], eax );      // Get current length
    mov( [ebx-8], ecx );      // Get maximum length.

    stdout.put
    (
        "theString = '", theString, "'", nl,
        "length(theString) = ", (type uns32 eax ), nl,
        "maxLength(theString) = ", (type uns32 ecx ), nl
    );

end StrDemo;
```

当对字符串变量的各个域进行访问的时候，像程序清单 4-8 中一样使用固定数字的偏移量进行访问是不明智的。将来，HLA 字符串的定义可能会做一些小的修改。尤其是，最大长度和当前长度域的偏移量是不断变化的。访问字符串数据的一种比较安全的方法是强制字符串指针使用 `str.strRec` 数据类型。`str.strRec` 数据类型是记录数据类型(参见 4.25 节)，它为字符串数据类型中的最大长度以及当前长度域的偏移量定义了符号名称。如果在将来的 HLA 版本中最大长度以及当前长度域的偏移量发生变化，那么 `str.strRec` 中的定义也会发生变化。因此，如果使用的是 `str.strRec`，那么重新编译将自动对程序做必要的修改。

为了正确地使用 `str.strRec` 数据类型，必须首先将字符串指针装载到一个 32 位的寄存器当中，例如 `mov(SomeString, ebx)`。只要指向字符串数据的指针被存储到寄存器当中，就可以使用 HLA 结构 `(type str.strRec [ebx])` 将相应的寄存器强制为 `str.strRec` 数据类型。最后，为了访问当前或者最大长度域，可以分别使用 `(type str.strRec [ebx]).length` 和 `(type str.strRec [ebx]).maxlen`。尽管需要较多的敲击键盘工作量(与只简单地使用偏移量如“-4”或者“-8”相比)，但这些形式的描述性更好，并且比使用使用直接的数字偏移量更加安全。程序清单 4-9 用 `str.strRec` 数据类型更改了程序清单 4-8 中的例子。

程序清单 4-9 访问字符串的长度域和最大长度域的正确方法

```
// Program to demonstrate accessing Length and Maxlength fields of a string

program LenMaxlenDemo;
#include( "stdlib.hhf" );

static
    theString:string := "String of length 19";
```



```

begin LenMaxlenDemo;

    mov(theString, ebx );           // Get pointer to the string.

    mov(( type str.strRec [ebx]).length, eax );   // Get current length
    mov((type str.strRec [ebx]).maxlen, ecx );    // Get maximum length

    stdout.put
    (
        "theString = ", theString, "'", nl,
        "length( theString )= ", (type uns32 eax ), nl,
        "maxLength( theString )= ", (type uns32 ecx ), nl
    );

end LenMaxlenDemo;

```

在 HLA 当中,操作字符串的第二个方法是在堆中分配存储区来存储字符串数据。因为字符串不能直接使用由 `mem.alloc` 返回的指针(因为字符串操作访问该地址之前的 8 个字节),所以不应该使用 `mem.alloc` 为字符串数据分配存储区。然而幸运的是,HLA 标准库内存模块提供了一个为给字符串分配内存而专门设计的内存分配例程:`str.alloc`。与 `mem.alloc` 类似,`str.alloc` 需要一个双字参数。这个数值指定了字符串需要的最大字符数。`str.alloc` 例程将会分配指定数量的存储空间,再加上 9~13 个用来保存额外字符串信息的附加字节⁷。

`str.alloc` 例程将为字符串分配存储区,将最大长度初始化为当作 `str.alloc` 参数传递的值,将当前长度初始化为 0,在字符串第一个字符的位置存储一个 0 结束字节。然后,`str.alloc` 在 EAX 寄存器中返回 0 结束字节的地址(也就是第一个字符元素的地址)。

一旦给字符串分配了存储空间后,就可以调用 HLA 标准库中的各种字符串操作例程对字符串进行操作。下一节将对 HLA 字符串例程进行详细讨论,本节将介绍两个与字符串相关的例程。第一个例程是 `stdin.gets(strvar);`。这个例程从用户端读取字符串,并将字符串数据存储到由字符串参数(如上述表达式的 `strvar`)指定的字符串存储区中。如果用户想要输入比最大字符串长度还要多的字符,`stdin.gets` 将会产生 `ex.StringOverflow` 异常。程序清单 4-10 展示了 `str.alloc` 的用法。

程序清单 4-10 从用户端读入一个字符串

```

// Program to demonstrate str.alloc and stdin.gets

program strallocDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strallocDemo;

    str.alloc( 16 );           // Allocate storage for the string and store
    mov( eax, theString );     // the pointer into the string variable.

    // Prompt the user and read the string from the user:

```

⁷ `str.alloc` 可能会给附加的数据分配多于 9 个的字节,因为分配给 HLA 字符串的存储空间必须是按双字对齐的,数据结构的总长度必须是 4 的倍数。

```

    stdout.put( "Enter a line of text (16 chars, max): " );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

end strallocDemo;

```

如果看得更仔细一些，就会发现上面的程序存在一个缺点。它通过调用 `str.alloc` 给字符串分配存储区，但是却没有释放存储区。尽管程序会在最后一次使用完字符串变量后立刻退出，操作系统会回收存储空间，但是显式释放分配的存储空间还是比较好的做法。这样做可以养成释放存储空间的习惯(从而在必须这样做的时候就不会忘记)，并且随着程序的不断扩展，现在不明显的错误可能在以后的版本中变成严重错误。

必须调用 `str.free` 例程释放通过 `str.alloc` 分配的存储空间，将字符串指针作为唯一的参数进行传递。程序清单 4-11 是对程序清单 4-10 中这个错误的修改。

程序清单 4-11 从用户端读入一个字符串的程序的修改版本

```

// Program to demonstrate str.alloc, str.free, and stdin.gets

program strfreeDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo;

    str.alloc( 16 );          // Allocate storage for the string and store
    mov( eax, theString );    // the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text (16 chars, max): " );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by str.alloc:

    str.free( theString );

end strfreeDemo;

```

阅读这个修改过的程序时，请注意，需要给 `stdin.gets` 例程传递一个字符串参数用于指向已经分配了存储空间的字符串对象。毫无疑问，HLA 程序初学者最容易犯的错误是调用 `stdin.gets`，并给它传递一个没有初始化的字符串变量。旧话重提，一定要记住，字符串就是指针！与指针一样，如果不给字符串初始化一个有效的地址，程序可能会在试图操作那个字符串对象的时候崩溃。上

述程序初始化字符串指针的方法是调用 `str.alloc`，然后使用 `move` 指令。如果想在程序中使用字符串变量，则必须确保在向字符串对象写数据之前给字符串数据分配存储空间。

一般情况下，在给字符串分配存储空间的时候，许多 HLA 标准库例程都会自动完成分配工作，减少程序员的工作量。通常，这样的例程都用“a_”作为它们名称的前缀。例如，`stdin.a_gets` 会将 `str.alloc` 和 `stdin.get` 两个例程的功能集成在一起。这个没有参数的例程从用户输入端读取一行文本以后，分配一个字符串来保存输入数据，然后将该字符串的一个指针返回给 EAX 寄存器。程序清单 4-12 使用 `stdin.a_gets` 改写了程序清单 4-10 和程序清单 4-11 中的两个程序。

程序清单 4-12 使用 `stdin.a_gets` 从用户端读入一个字符串

```
// Program to demonstrate str.free and stdin.a_gets

program strfreeDemo2;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo2;

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by stdin.a_gets:

    str.free( theString );

end strfreeDemo2;
```

注意，像前面一样，仍然必须通过调用 `str.free` 例程释放 `stdin.a_gets` 分配的存储空间。这个例程与前面两个例程的最大区别是，HLA 会自动为从用户端读取的字符串分配足够大的空间。在前面的例程中，对 `str.alloc` 的调用只分配 16 个字节。如果用户输入的字符多于 16 个，那么程序就会产生异常并退出。如果用户输入的字符少于 16 个，那么字符串后面的空间就浪费了。另一方面，`stdin.a_gets` 例程总是为用户输入的字符串分配所需的最小空间。因为它分配存储空间，所以一般不会出现溢出的情况⁸。

⁸ 实际上，对于 `stdin.a_gets` 能够分配的字符的最大个数是有限制的，通常情况下是在 1024~4096 个字节之间。查看 HLA 标准库源代码和您使用的操作系统文档可以找到确切的数值。

4.9 访问字符串中的字符

从字符串中提取单个字符是一项非常普遍的任务。实际上，因为它太简单了，HLA 甚至没有提供任何特定的过程或者语言语法来完成这项任务：只要使用简单的机器指令就能完成。一旦有了指向字符串数据的指针，只须使用一个简单的变址寻址方式就可以做剩下的工作了。

当然，最重要的仍然是要记住：字符串就是指针。因此，不能期望通过对一个字符串变量直接应用变址寻址方式来从字符串中提取字符。也就是说，如果 *s* 是字符串变量，那么 `mov(s[ebx], al)` 不会将字符串 *s* 中 EBX 位置上的字符取出来，并将它存储到 AL 寄存器当中。记住，*s* 仅仅是一个指针变量，类似于 `s[ebx]` 的寻址方式只会将存储器中相对于 *s* 的地址偏移量为 EBX 处的字节取出来(见图 4-1)。

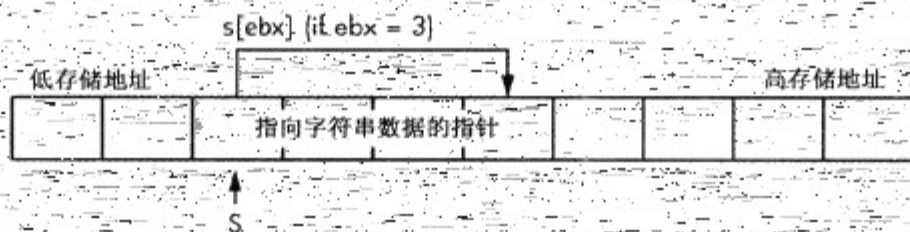


图 4-1 对字符串变量的错误变址

在图 4-1 中，假设 EBX 为 3，`s[ebx]` 不会访问字符串 *s* 中的第 4 个字符，而是取回指向字符串数据的指针的第 4 个字节。这应该不是所需要的。图 4-2 给出了从字符串中取回一个字符所需要的操作：假设 EBX 保存 *s* 的值。

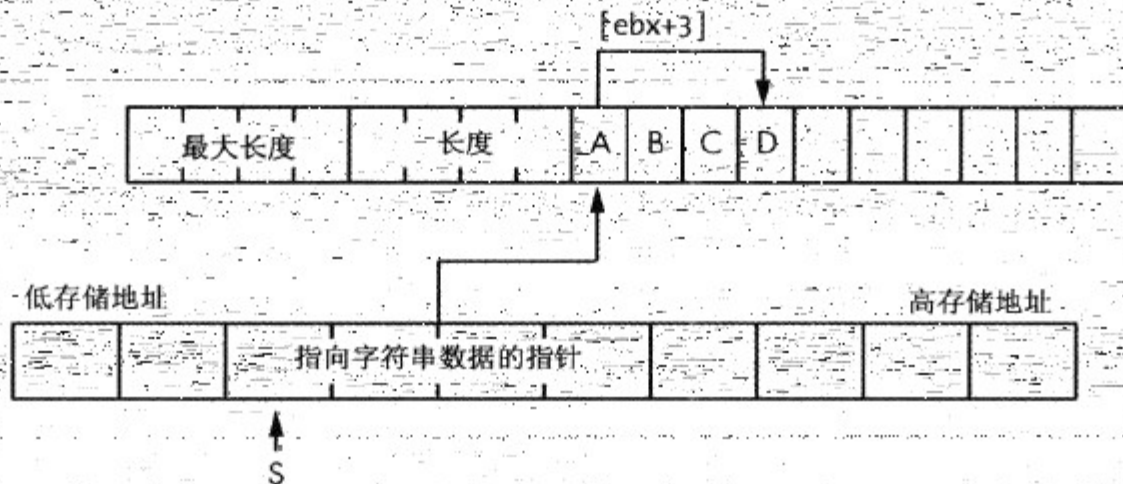


图 4-2 对字符串变量的正确变址

在图 4-2 中，EBX 保存着字符串 *s* 的值。*s* 的值是一个指向存储器中实际字符串数据的指针。因此，当将 *s* 的值装载到 EBX 中时，EBX 会指向字符串的第一个字符。如下所示的代码展示了如何用这种方法来对字符串 *s* 中的第 4 个字符进行访问：

```
mov( s, ebx );           // Get pointer to string data into ebx.
mov( [ebx+3], al );      // Fetch the fourth character of the string.
```

如果想要装载字符串中可变而非固定偏移量处的字符，那么可以使用 80x86 的某个比例变址寻址方式来取字符。例如，如果 `uns32` 变量 `index` 保存着字符串中的某个偏移量，那么可以使用如下所示的代码来访问 `s[index]` 处的字符：


```

mov( s, ebx );           // Get address of string data into ebx.
mov( index, ecx );       // Get desired offset into string.
mov( [ebx+ecx], al );     // Get the desired character into al.

```

上述代码中只存在一个问题：它并不检查偏移量 `index` 处的数据是否存在。如果 `index` 比当前字符串的长度还要大，则这段代码将从存储器中取回废弃字节。这种做法是非常危险的，除非能够事先确定 `index` 的值总是小于字符串的长度。比较好的做法是在访问这个字符之前，将 `index` 的值与字符串的当前长度进行比较。下面的代码就采用了这样的方法：

```

mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).length ) then

    mov( [ebx+ecx], al );

else

    <<Code that handles out-of-bounds string index >>

endif;

```

在 `if` 语句的 `else` 部分中，可以进行一些改正、打印一个错误信息，或者产生异常。如果希望显式地产生异常，则可以使用 HLA 的 `raise` 语句。`raise` 语句的语法如下所示：

```

raise( integer_constant );
raise( reg32 );

```

`integer_constant` 或者 32 位寄存器的值必须是异常号。通常，这是在 `excepts.hhf` 头文件中预定义的常量。当字符串的变址值大于其当前长度时，会产生 `ex.StringIndexError` 异常。下面的代码展示了如果字符串变址超过了上限后就会产生异常：

```

mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).length ) then

    mov( [ebx+ecx], al );

else

    raise( ex.StringIndexError );

endif;

```

4.10 HLA 字符串模块和其他与字符串相关的例程

尽管 HLA 为字符串数据提供了功能强大的定义，但其处理字符串的实际能力在于 HLA 标准库，而不是 HLA 字符串数据的定义。HLA 提供了许多字符串操作例程，远远超过了标准高级语言处理字符串的能力，例如 C/C++、Java 或者 Pascal。HLA 字符串处理能力甚至可以与字符串处理语言进行竞争，例如 Icon 或者 SNOBOL4。本章将对 HLA 标准库提供的一些字符串函数进行讨论。

最基本的字符串处理操作是将一个字符串赋给另外一个字符串。在 HLA 中有 3 种不同的方法可以给字符串赋值，分别是引用、复制字符串，以及重复字符串。在这些方法中，通过引用的方式给字符串赋值是最快、最容易的方法。如果有两个字符串，并想要把其中一个赋给另外一个，比较简单快速的方法是复制字符串指针。如下所示的代码段展示了这个过程：

```
static
    string1:          string := "Some String Data";
    string2:=         string;

    mov( string1, eax );
    mov( eax, string2 );
```

通过引用的方法来完成字符串赋值是非常高效的，因为它只需要执行两个 mov 指令，而不需要考虑字符串的长度。如果在赋值操作后不再修改字符串数据，那么采用通过引用的方法来进行字符串赋值就可以很好地完成工作。然而一定要记住，两个字符串变量(上述例子中的 string1 和 string2)都指向了相同的数据。因此，如果通过一个字符串变量修改了它所指向的数据，那么由第三个字符串变量所指向的数据也被修改了，因为它们所指的是相同的数据。程序清单 4-13 展示了这个问题。

程序清单 4-13 通过复制指针完成字符串赋值的问题

```
// Program to demonstrate the problem with string assignment by reference
program strRefAssignDemo;
#include( "stdlib.hhf" );

static
    string1: string;
    string2: string;

begin strRefAssignDemo;
    // Get a value into string1.

    forever
        stdout.put( "Enter a string with at least three characters:" );
        stdin.a gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars: " nl );
    endfor;
```



```

stdout.put( "You entered: '", string1, "'" nl );

// Do the string assignment by copying the pointer.

mov( string1, ebx );
mov( ebx, string2 );

stdout.put( "String1= '", string1, "'" nl );
stdout.put( "String2= '", string2, "'" nl );

// Okay, modify the data in string1 by overwriting
// the first three characters of the string (note that
// a string pointer always points at the first character
// position in the string and we know we've got at least
// three characters here).

mov( 'a', (type char [ebx]) );
mov( 'b', (type char [ebx+1]) );
mov( 'c', (type char [ebx+2]) );

// Okay, demonstrate the problem with assignment via
// pointer copy.

stdout.put
(
    "After assigning 'abc' to the first three characters in string1:"
    nl
    nl
);
stdout.put( "String1= '", string1, "'" nl );
stdout.put( "String2= '", string2, "'" nl );

str.free( string1 );      // Don't free string2 as well!

end strRefAssignDemo;

```

在这个例子中，由于 `string1` 和 `string2` 都指向同一个字符串数据，对其中一个字符串进行的修改同样也会对另外一个产生影响。尽管有的时候这种情况是可以接受的，但是大多数程序员还是希望通过字符串的副本进行赋值。也就是说，他们希望通过字符串赋值能够创建字符串数据的两个独立副本。

在通过引用进行复制(这种方式意味着复制一个指针)的时候，有一点需要记住，此时已经为字符串数据创建了别名。“别名”意味着存储器中的同一个对象有两个名称(例如，在上述程序中，`string1` 和 `string2` 是同一个字符串数据的两个不同名称)。在阅读程序的时候往往认为不同的变量指向存储器中不同的对象。别名打破了这个规则，使程序很难阅读和理解，因为必须记住，别名并不是指向存储器中不同的对象。如果忘记了这一点，就会在程序当中导致细微的缺陷。例如，在上述例子中，必须记住 `string1` 和 `string2` 之间是别名关系，这样才会在程序结束的时候不至于释放两个对象。还有更糟糕的，必须记住 `string1` 和 `string2` 之间是别名的关系，这样在释放 `string1` 以后才不至于继续使用 `string2`，因为这时候 `string2` 已经是一个悬空引用了。

因为通过引用进行复制会使得程序难以阅读，并且还增加了程序中出现缺陷的可能性，您可能不会理解为什么还是有人采用这种方法。这有两个原因：首先，通过引用的方法进行复制的效

率很高，只需要执行两个 `mov` 指令；其次，有一些算法依赖于通过引用的方法进行复制的语义。然而，在使用这项技术之前，需要认真考虑一下复制字符串指针是否是在程序中完成字符串赋值的合适方法。

将某个字符串赋值给另外一个字符串的第二种方法是复制字符串数据。HLA 标准库 `str.cpy` 例程提供了这个功能。对 `str.cpy` 例程的调用使用的调用语法如下所示⁹：

```
str.cpy( source_string, destination_string );
```

源串和目的串必须是字符串变量(指针)或者是保存字符串数据的存储器地址的32位寄存器。

`str.cpy` 例程首先检查目的字符串的最大长度域，以确保它至少要等于源字符串的当前长度。如果小于，`str.cpy` 将产生 `ex.StringOverflow` 异常。如果目的字符串的长度足够长，那么 `str.cpy` 就会从源字符串中复制字符串长度、字符，以及 0 结束字节到目的字符串。完成这项操作后，两个字符串将指向同样的数据，但是它们并不指向存储器中的同一个数据¹⁰。程序清单 4-14 使用 `srt.cpy` 而不是通过引用进行复制重新编写程序清单 4-13：

程序清单 4-14 使用 `srt.cpy` 复制字符串

```
// Program to demonstrate string assignment using str.cpy

program strcpyDemo;
#include( "stdlib.hhf" );

static
    string1: string;
    string2: string;

begin strcpyDemo;

    // Allocate storage for string2:

    str.alloc( 64 );
    mov( eax, string2 );

    // Get a value into string1.

    forever
        stdout.put( "Enter a string with at least three characters:" );
        stdin.gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars: " & nF );

    endfor;

    // Do the string assignment via str.cpy.
```

9 对 C/C++ 用户的警示：操作数的顺序与 C 标准库中的 `strcpy` 函数是相反的。

10 当然，除非两个字符串包含的起始地址是相同的，在这种情况下，`str.cpy` 将字符串数据复制到自己当中。


```

str.cpy( string1, string2 );

stdout.put( "String1= '" , string1, "'" nl );
stdout.put( "String2= '" , string2, "'" nl );

// Okay, modify the data in string1 by overwriting
// the first three characters of the string (note that
// a string pointer always points at the first character
// position in the string and we know we've got at least
// three characters here).

mov( string1, ebx );
mov( 'a', (type char [ebx]) );
mov( 'b', (type char [ebx+1]) );
mov( 'c', (type char [ebx+2]) );

// Okay, demonstrate that we have two different strings
// because we used str.cpy to copy the data:

stdout.put
(
    "After assigning 'abc' to the first three characters in string1:"
    nl
    nl
);
stdout.put( "String1= '" , string1, "'" nl );
stdout.put( "String2= '" , string2, "'" nl );

// Note that we have to free the data associated with both
// strings because they are not aliases of one another.

str.free( string1 );
str.free( string2 );

end strcpyDemo;

```

在程序清单 4-14 中存在两个非常重要的问题需要注意。首先，这个程序是以给 `string2` 分配存储空间开始的。记住，`str.cpy` 例程并不会为目的字符串分配存储空间，它假设目的字符串已经分配了存储空间。还要记住 `str.cpy` 不对 `string2` 进行初始化，它只是将数据复制到 `string2` 所指向的位置。程序本身要负责在调用 `str.cpy` 之前，通过给字符串分配足够大的空间将字符串初始化。第二个要注意的问题是，程序在退出之前调用 `str.free` 释放了 `string1` 和 `string2` 的存储空间。

在调用 `str.cpy` 之前给字符串变量分配存储空间是非常普遍的做法，HLA 标准库提供了一个例程来给字符串分配存储空间并复制字符串，即 `str.a_cpy`。该例程的语法如下所示：

```
str.a_cpy( source_string );
```

注意，其中没有目的字符串。这个例程查看源字符串的长度，分配足够大的存储空间，复制字符串，然后在 `EAX` 寄存器中返回一个指向新字符串的指针。程序清单 4-15 展示了如何使用 `str.a_cpy` 过程来完成与程序清单 4-14 相同的任务：

程序清单 4-15 使用 str.a_cpy 复制字符串

```
// Program to demonstrate string assignment using str.a_cpy

program stra_cpyDemo;
#include( "stdlib.hhf" );

static
    string1: string;
    string2: string;

begin stra_cpyDemo;

    // Get a value into string1.

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );
        breakif((type str.strRec [eax]).length >=3 );

        stdout.put( "Please enter a string with at least three chars:" nl );

    endfor;

    // Do the string assignment via str.a_cpy.

    str.a_cpy( string1 );
    mov( eax, string2 );

    stdout.put( "String1= ", string1, "" nl );
    stdout.put( "String2= ", string2, "" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we've got at least
    // three characters here).

    mov( string1, ebx );
    mov( 'a', (type char [ebx]) );
    mov( 'b', (type char [ebx+1]) );
    mov( 'c', (type char [ebx+2]) );

    // Okay, demonstrate that we have two different strings
    // because we used str.cpy to copy the data:

    stdout.put
    (
        "After assigning 'abc' to the first three characters in string1:"
        nl
        nl
    )
```



```

);
stdout.put( "String1= ", string1, "' ' nl );
stdout.put( "String2= ", string2, "' ' nl );

// Note that we have to free the data associated with both
// strings because they are not aliases of one another.

str.free( string1 );
str.free( string2 );

end stra_cpyDemo;

```

警告:

只要使用了通过引用进行复制或者 `str.a_cpy` 给字符串赋值, 就不要忘了释放已经(完全)使用完了的字符串数据的存储空间。如果不这样做, 并且没有另外的指针指向那个字符串数据, 就会导致内存泄漏。

获取字符串的长度是一个很普遍的操作, 因此, HLA 标准库提供了 `str.length` 例程专门来完成这项操作。当然, 也可以使用 `str.strRec` 数据类型来直接对长度域进行访问, 以此来取回长度值, 但是由于它所需的代码太多, 所以如果经常使用这个方法是很繁琐的。`str.length` 例程提供了更加简洁和方便的方法来获取长度信息。可以使用如下所示的两种形式之一调用 `str.length`。

```

str.length( Reg32 );
str.length( string variable );

```

这个例程将把当前字符串长度返回到 EAX 寄存器。

另外两个比较有用的字符串例程是 `str.cat` 和 `str.a_cat`。它们的语法如下所示:

```

str.cat( srcRStr, destLStr );
str.a_cat( srcLStr, srcRStr );

```

这两个例程将两个字符串连接起来(也就是说, 它们通过将两个字符串连接在一起创建了新的字符串)。`str.cat` 过程将源串连接到目的串的尾部。在连接实际进行之前, `str.cat` 检查目的字符串的长度以确保它是够保存连接后的结果。如果目的字符串的最大长度太小, 该程序就会产生 `ex.StringOverflow` 异常。

顾名思义, `str.a_cpy` 例程在进行连接之前为结果字符串分配存储空间。这个例程会分配足够大的空间来保存连接后的结果, 然后将 `srcLStr` 复制到分配的存储空间中。最后, 它会将 `srcRStr` 所指向的字符串数据添加到新字符串的尾部, 并在 EAX 寄存器中返回指向新字符串的指针。

警告:

注意, 这里可能会产生混淆。`str.cat` 过程将第一个操作数连接到第二个操作数的尾部。因此, `str.cat` 遵循许多 HLA 语句中的标准的(`src`, `dest`)操作数格式。`str.a_cat` 例程则具有两个源操作数, 而不是一个源操作数和一个目的操作数。该例程将两个操作数以从左到右的形式连接起来, 这与 `str.cat` 相反。在使用这两个例程的时候要记住这个问题。

程序清单 4-16 展示了 `str.cat` 例程和 `str.a_cat` 例程的用法。

程序清单 4-16 str.cat 例程和 str.a_cat 例程的示例

```
// Program to demonstrate str.cat and str.a_cat

program strcatDemo;
#include( "stdlib.hhf" );

static
    UserName:      string;
    Hello:         string;
    a_Hello:       string;

begin strcatDemo;

    // Allocate storage for the concatenated result:
    str.a_malloc( 1024 );
    mov( eax, Hello );

    // Get some user input to use in this example:
    stdout.put( "Enter your name:" );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, UserName );

    // Use str.cat to combine the two strings:
    str.cpy( "Hello ", Hello );
    str.cat( UserName, Hello );

    // Use str.a_cat to combine the string strings:
    str.a_cat( "Hello ", UserName );
    mov( eax, a_Hello );

    stdout.put( "Concatenated string #1 is '", Hello, "' n" );
    stdout.put( "Concatenated string #2 is '", a_Hello, "' n" );

    str.free( UserName );
    str.free( a_Hello );
    str.free( Hello );

end strcatDemo;
```

str.insert 和 str.a_insert 例程与字符串连接过程类似。然而，str.insert 和 str.a_insert 例程可以将一个字符串插入到另外一个字符串的任意位置，而不是只连接到尾部。这两个例程的语法如下所示：

```
str.insert( src, dest, index );
str.a_insert( src, dest, index );
```

这两个例程将源字符串(src)插入目的字符串(dest)中从 index 字符位置开始的地方。str.insert 例程将源字符串直接插入到目的字符串。如果目的字符串没有足够大的空间存储两个字符串，该例

程会产生一个 `ex.StringOverflow` 异常。`str.a_insert` 例程首先在堆中为新的字符串分配存储空间, 并将目的字符串(*dest*)复制到新字符串中, 然后将源字符串(*src*)插入新字符串中的指定偏移量处。`str.a_insert` 在寄存器 `EAX` 中返回一个指向新字符串的指针。

字符串的变址从 0 开始。也就是说, 如果以 0 作为 `str.insert` 或者 `str.a_insert` 的变址, 这两个例程就会将源字符串插入到目的字符串的开头。同样, 如果 `index` 与字符串的长度相等, 它们就会把源字符串连接到目的字符串的尾部。

警告:

如果 `index` 值比字符串的长度大, 那么 `str.insert` 和 `str.a_insert` 将不会产生异常, 而只是将源字符串连接到目的字符串的尾部。

`str.delete` 和 `str.a_delete` 例程可用于从字符串中删除字符。它们的语法如下所示:

```
str.delete( strng, StartIndex, Length );
str.a_delete( strng, StartIndex, Length );
```

两个例程都从字符串 *strng* 的字符位置 *StartIndex* 处开始删除长度为 *Length* 的字符。它们两者之间的区别在于, `str.delete` 直接从 *strng* 字符串中删除字符, 而 `str.a_delete` 则先分配存储空间并复制 *strng*, 然后从新的字符串中删除字符(不改变原来的 *strng*)。 `str.a_delete` 在 `EAX` 寄存器中返回一个指向新字符串的指针。

`str.delete` 和 `str.a_delete` 例程对于传递给 *StartIndex* 和 *Length* 的值的检查是很宽松的。如果 *StartIndex* 比当前字符串的长度大, 则不会从字符串中删除字符。如果 *StartIndex* 比当前字符串的长度小, 但是 *StartIndex*+*Length* 的值大于字符串当前的长度, 那么这两个例程会将从 *StartIndex* 开始一直到字符串最后的字符全部删除。

另外一个比较常用的字符串操作是将一个字符串的一部分复制到另外一个字符串中, 而不对源字符串造成影响。`str.substr` 和 `str.a_substr` 例程提供这一功能。这两个例程的语法如下所示:

```
str.substr( src, dest, StartIndex, Length );
str.a_substr( src, StartIndex, Length );
```

`str.substr` 例程将 *src* 字符串中从 *StartIndex* 开始的, 长度为 *Length* 的字符复制到 *dest* 字符串中。*dest* 字符串必须具有足够的存储空间来存储新的字符串, 否则 `str.substr` 将产生 `ex.StringOverflow` 异常。如果 *StartIndex* 比字符串的长度长, 则 `str.substr` 将产生 `ex.StringIndexError` 异常。如果 *StartIndex*+*Length* 的值大于源字符串当前的长度, 而 *StartIndex* 的值小于字符串当前的长度, 则 `str.substr` 过程将只提取从 *StartIndex* 开始一直到字符串最后的字符。

`str.a_substr` 过程的行为与 `str.substr` 的行为几乎相同, 只是它在堆中为目的字符串分配存储空间。除了任何时候都不会产生字符串溢出异常以外(因为这种情况永远不会发生)¹¹, `str.a_substr` 与 `str.substr` 处理异常的方式是相同的。现在您可能已经猜出来, `str.a_substr` 将在 `EAX` 寄存器中返回一个指向新字符串的指针。

在对字符串数据操作后, 剩下的可能就是需要对两个字符串进行比较。比较字符串的第一种

¹¹ 从技术上讲, 和调用 `mem.alloc` 分配存储空间的所有例程一样, `Str.a_substr` 会产生 `ex.MemoryAllocationFailure` 异常, 但这种情况发生的几率几乎没有。

方法是使用标准的 HLA 关系操作符，这样的代码可以编译，但是产生的结果不理想：

```

mov( s1, eax );
if( eax =s2 ) then

    << Code to execute if the strings are equal >>

else

    << Code to execute if the strings are not equal >>

endif;
```

记住，字符串就是指针。这段代码比较了两个指针，看它们是否相等。如果它们相等，显然两个字符串也是相等的(因为 *s1* 和 *s2* 指向的是同一个字符串数据)。但是问题在于，两个指针不相等并不意味着两个字符串不相等。*s1* 和 *s2* 可能保存不同的数值(也就是说，它们指向存储器中不同的地址)，但是这两个不同的地址中的字符串数据可能是相等的。大多数的程序员都希望，如果两个字符串的数据相等，则字符串比较的结果为 **true**。很显然，指针的比较并不能提供这种比较。为了克服这个问题，HLA 标准库提供了一组字符串比较例程，可以比较两个字符串数据，而不只是它们的指针。这些例程的语法如下所示：

```

str.eq( src1, src2 );
str.ne( src1, src2 );
str.lt( src1, src2 );
str.le( src1, src2 );
str.gt( src1, src2 );
str.ge( src1, src2 );
```

上述的每一个例程都将字符串 *src1* 和 *src2* 进行比较，并依照比较的结果在 EAX 寄存器中返回 **true**(1)或者 **false**(0)。例如，如果 *s1* 等于 *s2*，`str.eq(s1, s2)`；将在 EAX 中返回 **true**。HLA 还进行了一点扩展，可以在 if 语句中使用字符串比较例程¹²。下面的代码展示了在 if 语句中使用这些比较例程的例子：

```

stdout.put( "Enter a single word:" );
stdin.a_gets();
if( str.eq( eax, "Hello" ))then

    stdout.put( "You entered 'Hello'", nl );

endif;
str.free( eax );
```

注意，在这个例子中，用户输入的字符串必须与 **Hello** 完全相同，包括在字符串的开头使用大写 **H**。当处理用户输入的时候，最好是忽略字母大小写，因为不同的用户可能对什么时候按下键盘上的 **Shift** 键有不同的想法。一种简单的解决方法是使用 HLA 中不区分大小写的字符串比较函数。这些例程可以比较两个字符串，但忽略大小写形式。这些例程的语法如下所示：

¹² 这一扩展实际要比这里所描述的内容宽泛些。第7章作了详细介绍。

```
str.ieq( src1, src2 );
str.ine( src1, src2 );
str.ilt( src1, src2 );
str.ile( src1, src2 );
str.igt( src1, src2 );
str.ige( src1, src2 );
```

除了不区分大小写之外,这些例程与前面例程的运行方式是相同的,依据比较的结果在 EAX 寄存器中返回 true 或者 false。

与大多数的高级语言一样,HLA 按字典顺序比较字符串。也就是说,当且仅当两个字符串的长度相等,并且其中相应的字符完全相同时,这两个字符串相等。对于大于或者小于的比较,字典顺序根据每个字母在字典中出现的先后顺序来比较。也就是说,a 比 b 小,b 比 c 小,依此类推。实际上,HLA 使用字符的 ASCII 码比较字符串,因此,如果不确定 a 是否小于逗号,只需要查阅 ASCII 字符表(顺便提一下,在 ASCII 字符集中,a 比逗号大)。

如果两个字符串的长度不同,则只有当较短的字符串与较长字符串的对应字符相同时,字典顺序才考虑长度。如果是这种情况,则较长的字符串大于较短的字符串(相反的,较短的字符串小于较长的字符串)。注意,如果两个字符串中的字符完全不同,HLA 的字符串比较例程就会忽略字符串的长度。例如,z 总是比 aaaaaa 大,尽管它比较短。

str.eq 例程检查两个字符串是否相等。然而有时候可能想知道,某个字符串是否包含另外一个字符串。例如,在游戏中,有时可能想知道某些字符串包含 north 子字符串还是 south 字符串,以此来确定下一步该怎么做。HLA 的 str.index 例程可以检查一个字符串是否是另外一个字符串的子字符串。str.index 例程的语法如下所示:

```
str.index( StrToSearch, SubstrToSearchFor );
```

这个函数在 EAX 中返回 StrToSearch 中的偏移量,SubstrToSearchFor 就出现在这个偏移量开始的地方。如果 SubstrToSearchFor 没有出现在 StrToSearch 中,则该例程就会在 EAX 中返回 -1。注意 str.index 将会进行大小写检查。因此,字符串必须完全相等。str.index 没有不进行大小写检查的变体¹³。

除了本节介绍过的以外,HLA 字符串模块还包含许多其他的例程。由于篇幅限制以及预备知识的限制,这里不会介绍所有的字符串函数。然而,这并不意味着其他的字符串函数不重要。可以查看 HLA 标准库文档了解它们的定义,学习 HLA 字符串库例程所提供的强大功能。

4.11 存储器内的转换

HLA 标准库的字符串模块包含大量在字符串和其他数据格式之间进行转换的例程。尽管在本书中完全描述这些函数还为时过早,但是讨论其中的一个,即 str.put 例程,还是很有必要的。这个例程封装了其他许多字符串转换函数的功能。因此,如果学习了如何使用这个函数,就可以在程序中拥有其他的例程所提供的功能。

¹³ 然而,HLA 提供了可以将字符串中所有的字符转换为另一种大小写形式的例程。因此可以对字符串进行复制,并将两个副本中的字符全都转换为小写形式,然后使用这些经过转换后的字符串进行搜索。这样做得到的结果是一样的。

`str.put` 例程的用法与 `stdout.put` 的用法非常类似。唯一的不同之处在于，`str.put` 例程将它的数
据“写入”一个字符串，而不是输出到标准的输出设备。调用 `str.put` 的语法如下所示：

```
str.put( destString, values_to_convert );
```

下面是调用 `str.put` 的一个例子：

```
str.put( destString, "I= ", i:4, " J= ", j, " s=", s );
```

警告：

一般来说，这种输出与将字符串打印到标准的输出设备不同，在字符串后面不会有换行字符
序列。

`str.put` 参数列表中的第一个参数 `destString` 必须是字符串变量，并且必须已经分配了存储空间。
如果 `str.put` 试图向 `destString` 中存储多于限制数的字符，则这个函数将会产生 `ex.StringOverflow`
异常。

许多时候可能不知道 `str.put` 将要生成的字符串的长度。这种情况下，需要为一个大的字符串
分配足够大的存储空间，并用该字符串数据作为 `str.put` 调用的第一个参数。这样可以防止发生异
常而使程序崩溃。一般来说，如果希望在屏幕上打印一行字符，则可能至少要为目的字符串分配
256 个字符。如果字符串更长，则可以使用默认的 1024 个字符(或者更多，如果确实要生成一个
很大的字符串)。

这里有一个例子：

```
static
    s: string;
    .
    .
    .
    str.alloc( 256 );
    mov( eax, s );
    .
    .
    .
    str.put( s, "R: ", r:16:4, " strval: '", strval: -10, "'" );
```

可以使用 `str.put` 例程将任意数据转换为字符串，然后使用 `stdout.put` 打印。您会发现，对于一
般的数据到字符串的转换来说，这个例程都是非常有用的。

4.12 字符集

字符集是另外一种复合数据类型。它与字符串一样，建立在字符数据类型的基础上。字符集
是一组字符的数学集合，最重要的属性是成员。也就是说，一个字符要么是一个字符集的成员，
要么不是。序列的概念(例如在字符串中，某个字符是否在另外一个字符的前面)不适用于字符集。
此外，成员是一个二元关系，某个字符要么在集合中，要么不在集合中；如果在字符集合中，字

符不能有多个副本。最后,在字符集中可以执行许多操作,包括数学集中的求并集、求交集、求差集以及成员资格测试等操作。

HLA 实现了字符集的受限格式,它的成员可以是 128 个标准 ASCII 字符的任何一个(也就是说,HLA 字符集合功能不支持 128~255 范围内的扩展的字符代码)。尽管有这些限制,HLA 的字符集功能在编写处理字符串数据的程序时是非常强大的,也非常方便。下面的小节描述了 HLA 字符集功能的实现和应用,从而在程序中可以使用字符集功能。

4.13 在 HLA 中实现字符集

在汇编语言程序中有很多方法可以表示字符集。HLA 通过使用一组 128 个布尔值实现字符集。每一个布尔值确定了相对应的字符是否为字符集的某个成员。也就是说,布尔值为真表示相应的字符是集合的成员;布尔值为假表示相应的字符不是集合的成员。为了节约内存,HLA 只为集合中的每个字符分配一位;因此,HLA 字符集占用了 16 个字节的内存空间,因为 16 个字节中有 128 位。这组 128 位的数据在存储器中的组织方式如图 4-3 所示:

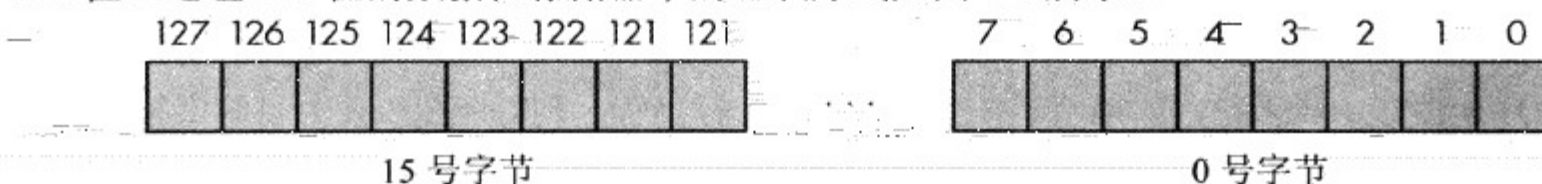


图 4-3 字符集对象的位布局

0 号字节的第 0 位对应 ASCII 码 0(NUL 字符)。如果这一位为 1,则该字符集包含 NUL 字符;如果这一位为假,那么该字符集就不包含 NUL 字符。同样,1 号字节的第 0 位(128 位中的第 8 位)对应于退格符(ASCII 码为 8)。8 号字节的第 1 位对应 ASCII 码 65,是大写的 A。如果 A 是当前字符集的成员,则第 65 位为 1,否则为 0。

尽管还有其他的方法可以实现字符集,这种位向量实现方法还是有许多优点的,因为这样实现集合操作比较容易,如并集、交集、差集比较,以及成员资格测试。

HLA 使用 `cset` 数据类型支持字符集变量。可以使用如下所示的声明方法声明字符集变量:

```
static
  CharSetVar: cset;
```

这个声明将会保留 16 字节的存储区来存储表示 ASCII 字符的 128 位的数据。

尽管可以使用类似 `and`、`or`、`xor` 等指令操作字符集中的位,80x86 指令集包含了一些位测试、置位、复位以及取反指令,非常适合于操作字符集。例如,`bt`(位测试)指令会将存储器中的某个位复制到进位标志中。`bt` 指令的语法格式如下所示:

```
bt( BitNumber, BitsToTest );

bt( reg16, reg16 );
bt( reg32, reg32 );
bt( constant, reg16 );
bt( constant, reg32 );

bt( reg16, mem16 );
```

```

bt( reg32, mem32 );      // HLA treats cset objects as dwords within bt.
bt( constant, mem16 );
bt( constant, mem32 );   // HLA treats cset objects as dwords within bt.

```

第一个操作数是位号，第二个操作数指定了一个寄存器或者存储单元，该寄存器或者存储单元中的位将被复制到进位标志中。如果第二个操作数是寄存器，则第一个操作数必须是 $0 \sim n-1$ 之间的数据，其中 n 是第二个操作数中的位数。如果第一个操作数是常量，第二个操作数是一个存储单元，则常量的值必须在 $0 \sim 255$ 之间。下面是这些指令的一些例子：

```

bt( 7, ax );             // Copies bit 7 of ax into the carry flag (CF).
mov( -20, eax );
bt( eax, ebx );          // Copies bit 20 of ebx into CF.

// Copies bit 0 of the byte at CharSetVar+3 into CF.
bt( 24, CharSetVar );

// Copies bit 4 of the byte at DWmem+2 into CF.
bt( eax, DWmem );

```

bt 指令在测试集合成员的时候非常有用。例如，为了查看字符 A 是否是字符集的成员，可以使用如下所示的代码：

```

bt( 'A', CharSetVar );
if( @c ) then
    << Do something if 'A' is a member of the set. >>
endif;

```

bts(位测试并置位)、btr(位测试并复位)和 btc(位测试并取反)指令在操作字符集变量时也非常有用。与 bt 指令一样，这些指令将指定的位复制到进位标志中。在复制完指定的位后，这些指令将对指定的位进行置位(bts)、复位/清零(btr)或者取反(btc)操作。因此，可以使用 bts 指令通过并集向字符集中添加字符(也就是说，如果字符不是集合的成员，它会将该字符添加到集合中，否则集合不受任何影响)。可以使用 btr 指令通过交集从字符集中删除一个字符(也就是说，当且仅当字符存在于集合中的时候，它会将这个字符从集合中删除；否则集合不受任何影响)。如果字符不在集合中，btc 指令将会把这个字符加入集合中；如果字符在集合中，它会将字符从集合中删除(也就是说，这个函数来回改变字符的成员资格)。

4.14 HLA 字符集常量和字符集表达式

HLA 支持字面字符集常量。这些 cset 类型的常量使得很容易在编译时初始化 cset 变量，也很容易将字符集常量作为过程参数传递。HLA 字符集常量的格式如下所示：

```
[ Comma separated list of characters and character ranges ]
```

下面是一个包含阿拉伯数字字符的简单字符集的例子：


```
{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }
```

当指定一个包含若干相邻数值的字符集字面量时,HLA 允许只使用这个范围的开始值和末尾值简明地表示数据:

```
{ '0'..'9' }
```

可以将字符和各种范围结合在同一个字符集常量中。例如,如下所示的字符集常量是所有字母数字字符:

```
{ '0'..'9', 'a'..'z', 'A'..'Z' }
```

cset 字面常量可以在 const 和 val 段中用作初始值设定项。下面的例子展示了如何使用上述的字符集创建符号常量 AlphaNumeric:

```
const
    AlphaNumeric: cset := { '0'..'9', 'a'..'z', 'A'..'Z' };
```

进行上面的声明后,可以在任何字符集字面量合法的地方使用标识符 AlphaNumeric。

还可以使用字符集字面量(当然还有字符集符号常量)作为 static 或者 readonly 变量的初始值设定项。下面的代码展示了这种情况:

```
static
    Alphabetic: cset := { 'a'..'z', 'A'..'Z' };
```

在任何可以使用字符集字面常量的地方,字符集常量表达式也是合法的。表 4-2 显示了 HLA 在字符集常量表达式中支持的操作符:

表 4-2 HLA 字符集操作符

操 作 符	描 述
CsetConst1 + CsetConst2	计算两个集合的并集。集合的并集是出现在任何一个集合中的字符组成的集合
CsetConst1 * CsetConst2	计算两个集合的交集。集合的交集是同时出现在两个操作数集合中的所有字符组成的集合
CsetConst1 - CsetConst2	计算两个集合的差集。集合的差集是出现在第一个集合中、但是没有出现在第二个集合中的字符组成的集合
- CSetConst	计算集合的补集。集合的补集是没有包含在集合中的所有字符

注意,这些操作符只产生编译时的结果。也就是说,上述表达式的值是由编译器在编译过程中计算出来的。它们并不发出任何机器代码。如果想要在程序运行时对两个不同的集合执行这些操作,HLA 标准库提供了一些例程,可以调用它们得到所需要的结果。HLA 还提供了其他编译时字符集操作符。

4.15 HLA 标准库对字符集的支持

HLA 标准库中提供了若干有用的字符集例程。支持字符集的例程可以分为 4 种：标准字符集函数、字符集测试、字符集转换、字符集 I/O。本节将介绍 HLA 标准库中的这些例程。

首先，我们看一下能够帮助创建字符集的标准库例程。这些例程包括 `cs.empty`、`cs.cpy`、`cs.charToCset`、`cs.unionChar`、`cs.removeChar`、`cs.rangeChar`、`cs.strToCset` 以及 `cs.unionStr`。这些过程可以在运行时使用字符和字符串对象创建字符集。

`cs.empty` 过程通过把字符集中的所有位都置 0 将字符集变量初始化为空集。这个过程调用使用的语法为(*Csvar* 是一个字符集变量)：

```
cs.empty( Csvar );
```

`cs.cpy` 过程将一个字符集复制到另外一个字符集，替换目的字符集中所有的数据。`cs.cpy` 的语法为：

```
cs.cpy( srcCsetValue, destCsetVar );
```

`cs.cpy` 源字符集可以是字符集常量，也可以是字符集变量。而目的字符集必须是字符集变量。

`cs.unionChar` 过程将一个字符添加到三个字符集中，它的调用语法如下所示：

```
cs.unionChar( CharVar, Csvar );
```

这个调用通过并集将第一个参数(一个字符)添加到字符集中。注意，尽管可以使用 `bts` 指令完成同样的操作，但是 `cs.unionChar` 调用通常比较方便。字符值必须在 #0~#127 之间。

`cs.charToCset` 函数产生一个单元元素集合(只包含一个字符的集合)。它的调用语法如下所示：

```
cs.charToCset( CharValue, Csvar );
```

第一个操作数 *CharValue* 是字符值，它可以是一个 8 位的寄存器、一个常量或者是值在 #0~#127 之间的一个字符变量。第二个操作数(*Csvar*)必须是字符集变量。这个函数将目的字符集清零，然后将指定的字符合并到字符集中。

`cs.removeChar` 过程可以从某个字符集中删除一个字符，而不影响字符集中的其他字符。这个函数的语法与 `cs.charToCset` 相同，参数的性质也相同。调用的序列为：

```
cs.removeChar( CharValue, Csvar );
```

注意，如果字符不在 *Csvar* 字符集中，则 `cs.removeChar` 函数对此字符集没有任何影响。这个函数大致对应于 `btr` 指令。

`cs.rangeChar` 函数创建一个字符集，这个字符集中包含两个参数之间的所有字符。这个函数将不在两个参数之间的所有位都置为 0。调用语法为：

```
cs.rangeChar( LowerBoundChar, UpperBoundChar, Csvar );
```

参数 *LowerBoundChar* 和 *UpperBoundChar* 可以是常量、寄存器或者是字符变量。其中保存的

值必须在#0~#127之间。目的字符集 *CSVar* 必须是一个 *cset* 变量。

cs.strToCset 过程创建一个包含某个字符串中所有字符的新字符集。这个过程首先把目的字符集设置为空,然后将字符串中的字符一个一个地加进来,直到字符串中所有的字符都加进来为止。调用序列为:

```
cs.strToCset( StringValue, CSVar );
```

StringValue 参数可以是字符串常量,也可以是字符串变量。然而,这样调用 *cs.strToCset* 是没有什么意义的,因为对于用常量集中的字符初始化一个字符集来说,*cs.cpy* 是一个更加高效的方法。通常,目的字符集必须是一个 *cset* 变量。一般情况下,可能会用这个函数根据用户端的输入创建一个字符集。

cs.unionStr 过程把一个字符串中的字符添加到已经存在的字符集中。与 *cs.strToCset* 相同,通常使用这个函数根据用户端的字符串输入创建字符的并集。调用序列如下所示:

```
cs.unionstr( StringValue, CSVar );
```

标准的集合操作包括并集、交集以及差集。HLA 标准库例程 *cs.setunion*、*cs.intersection* 以及 *cs.difference* 分别提供了这些功能¹⁴。这些函数的调用序列相同:

```
cs.setunion( srcCset, destCset );
cs.intersection( srcCset, destCset );
cs.difference( srcCset, destCset );
```

第一个参数可以是字符集常量或者字符集变量。第二个参数必须是字符集变量。这些过程计算的是 *destCset := destCset op srcCset*。在这里,根据函数调用的不同,*op* 代表的是并集、交集或者差集。

第三种字符集例程以各种方式对字符集进行测试。它们通常返回代表测试结果的布尔值。这一类 HLA 字符集函数包括 *cs.isEmpty*、*cs.member*、*cs.subset*、*cs.psubset*、*cs.psuperset*、*cs.eq* 以及 *cs.ne*。

cs.isEmpty 函数测试字符集是否为空集。该函数在 EAX 寄存器中返回 *true* 或者 *false*。这个函数的调用序列如下所示:

```
cs.isEmpty( CsetValue );
```

其中的参数可以是常量或者是字符集变量,尽管向该过程传递字符集常量并没有什么意义(因为在编译时就可以知道这个集合是否为空)。

cs.member 函数测试一个字符值是否是某个集合的成员。如果给出的字符是指定集合的成员,那么这个函数在 EAX 寄存器中返回 *true*,注意也可以使用 *bt* 指令来对同样的情况进行测试。然而,如果字符参数不是一个常量,那么 *cs.member* 函数使用起来可能更加方便。*cs.member* 的调用序列如下所示:

```
cs.member( CharValue, CsetValue );
```

¹⁴ 使用的是 *cs.setunion* 而不是 *cs.union*, 因为 *union* 是 HLA 的保留字。

第一个参数可以是 8 位寄存器、字符变量或者是常量。第二个参数可以是字符集常量，也可以是字符集变量。两个参数都是常量的情况不太多见。

`cs.subset`、`cs.psubset`(真子集)、`cs.superset` 以及 `cs.psuperset`(真超集)函数可以测试某个字符集是否是另外一个字符集的子集或者超集。这些例程的调用序列几乎相同：

```
cs.subset( CsetValue1, CsetValue2 );
cs.psubset( CsetValue1, CsetValue2 );
cs.superset( CsetValue1, CsetValue2 );
cs.psuperset( CsetValue1, CsetValue2 );
```

这些例程将第一个参数与第二个参数进行比较，根据比较的结果在 EAX 寄存器中返回 `true` 或 `false`。如果第一个字符集的所有成员都存在于第二个字符集当中，那么第一个字符集是第二个字符集的子集。如果第二个字符集中还包含没有出现在第一个字符集中的字符，那么第一个字符集是第二个字符集的真子集。同样地，如果第一个字符集包含第二个字符集的所有字符，那么第一个字符集是第二个字符集的超集。如果第一个字符集中还包含没有出现在第二个字符集中的字符，那么第一个字符集是第二个字符集的真超集。这些参数可以是字符集变量或者字符集常量；然而，两个参数都是字符集常量的情况并不多见(因为这种情况可以在编译时就知道，没有必要在运行时调用一个函数来确定)。

`cs.eq` 和 `cs.ne` 函数检查两个集合是否相等。这些函数根据集合比较的结果在 EAX 中返回 `true` 或者 `false`。调用序列与上述的子集/超集函数相同：

```
cs.eq( CsetValue1, CsetValue2 );
cs.ne( CsetValue1, CsetValue2 );
```

注意，没有测试小于、小于等于、大于、大于等于的函数。子集和真子集分别等效于小于等于和小于。同样，超集和真超集分别等效于大于等于和大于。

`cs.extract` 例程将任意一个字符从字符集中删除，并在 EAX 寄存器中返回该字符¹⁵。调用序列如下所示：

```
cs.extract( CsetVar );
```

该例程的参数必须是字符集变量。注意，因为这个函数需要从字符集中删除某个字符，因此会改变字符集变量的值。如果在调用之前这个字符集为空，则该函数在 EAX 中返回 `$FFFF_FFFF(-1)`。

除了 `cset.hhf`(字符集)库模块中的函数之外，字符串和标准输出模块还提供了其他允许或者需要字符集作为参数的函数。例如，如果在 `stdout.put` 例程中以字符集作为参数，该例程将会打印当前集合中的字符。查阅 HLA 标准库文档可以了解更多关于字符集处理的过程。

4.16 在 HLA 程序中使用字符集

字符集在程序中可以发挥各种各样的作用。例如，字符集的一种常见应用是确认用户输入。

¹⁵ 这个例程在 AL 中返回字符，并将 EAX 中高三个字节清零。

本节将给出字符集的另外两个应用，可以帮助您思考如何在自己的程序中使用字符集。

下面一段代码的作用是从用户端取回 yes 或者 no 类型的回答：

```
static
    answer: char;

    .
    .
    .
    repeat

        stdout.put( "Would you like to play again?" );
        stdin.FlushInput();
        stdin.get( answer );

    until( answer = 'n' );
```

这段代码存在的唯一问题是，只有在用户输入了一个小写的 n 字符时，程序才会结束。如果输入的是任意一个其他的字符(包括大写的 N)，程序会认为这是一个肯定的回答，并将这个回答传递给 repeat..until 循环的开始处。比较好的解决方法是在上述的 until 子句之前确认用户的输入，确保用户只输入了 n、N、y 或者 Y。下面的代码将完成这项功能：

```
repeat
    .
    .
    .
    repeat

        stdout.put( "Would you like to play again?" );
        stdin.FlushInput();
        stdin.get( answer );

        until( cs.member( answer, { 'n', 'N', 'Y', 'y' } ) );
        if( answer = 'N' )then

            mov( 'n', answer );
        endif;

    until( answer = 'n' );
```

4.17 数组

与字符串一样，数组可能是最常用的复合数据。然而编程新手往往对数组的操作过程以及效率代价理解得不够。在他们知道了如何在机器级操作数组之后，许多编程新手(甚至有经验的程序员)就会从一个完全不同的角度看待数组。

理论上，数组是聚合数据类型，它们的成员(元素)都是同一类型的。从数组中选择一个元素

是通过整数索引进行的¹⁶。不同的索引选择数组中不同的元素。本书假设整数索引是连续的(尽管并非必须如此)。也就是说,如果数字 x 是数组的一个有效索引, y 也是一个有效索引,并且 $x < y$,那么所有的 $i(x < i < y)$ 都是数组的有效索引。

当对数组进行索引操作时,得到的结果是通过索引选择的特定数组元素。例如, $A[i]$ 从数组 A 中选择第 i 个元素。注意,在存储器中,第 i 个元素未必与第 $i+1$ 个元素的位置相邻。只要 $A[i]$ 指向的是同一个存储地址,并且 $A[i+1]$ 指向的是相应的存储地址(这两个地址是不同的),就满足数组的定义。

在本书中,我们假设数组元素占用的是存储器中连续的存储空间,图 4-4 所示的是一个具有 5 个元素的数组在存储器中的状态。

数组的基址是数组第一个元素的地址,并总是出现在存储位置的最低位。第二个数组元素在存储器中直接跟在第一个的后面,第三个跟在第二个的后面,依此类推。注意在这里,索引并不一定要从 0 开始。只要它们是连续的,可以从任何一个数字开始。然而,为了讨论方便,本书都将从 0 开始索引数组。

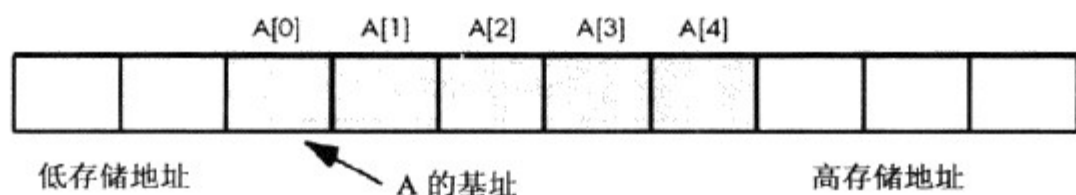


图 4-4 数组在存储器中的排列

为了访问数组中的某个元素,需要有一个将数组索引转换为被索引元素地址的函数。对于一维的数组,这个函数非常简单,如下所示:

$$\text{Element_Address} = \text{Base_Address} + ((\text{index} - \text{Initial_Index}) * \text{Element_Size})$$

在这里, Initial_Index 是数组中第一个索引的值(如果是 0,可以忽略), Element_Size 是数组中元素的数据宽度,以字节为单位。

4.18 在 HLA 程序中声明数组

在访问一个数组的元素之前,必须为这个数组分配存储空间。幸运的是,数组声明以前面看到声明为基础。要在数组中分配 n 个元素,可以在变量声明段中使用如下所示的声明:

```
ArrayName: basetype[n];
```

其中 ArrayName 是数组变量的名称, basetype 是数组中元素的类型。这个语句就为数组分配了存储空间。只要使用 ArrayName , 就得到了数组的基址。

后缀 $[n]$ 的意思是告诉 HLA 将某个对象重复 n 次。下面让我们看一些具体的例子:

```
static
```

```
    CharArray: char[128];    // Character array with elements 0..127.
```

¹⁶ 或者是其底层表示是整数的值,例如字符、枚举以及布尔类型。

```

ByteArray: byte[10];      // Array of bytes with elements 0..9.
PtrArray:  dword[4];     // Array of double words with elements 0..3.

```

这些例子为没有初始化的数组分配存储空间。也可以在静态或者只读段中使用类似于如下所示的声明，指定数组中的元素被初始化为某个数值：

```

RealArray: real32[8]:= [ 1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0 ];
IntegerAry: int32[8]:= [ 1,1,1,1,1,1,1,1 ];

```

这两个定义都创建了具有 8 个元素的数组。第一个定义将每一个 4 字节的 `real32` 值初始化为 1.0；第二个定义将每一个 `int32` 元素初始化为 1。注意，中括号中常量的数量必须与数组的大小相等。

如果数组中所有的元素数值都相同，使用这样的定义机制是正确的。但是，如果要将数组中的元素初始化为(可能)不同的数值该怎么办呢？不用担心，只需要在中括号中指定不同的数据就可以了：

```

RealArray: real32[8]:= [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ];
IntegerAry: int32[8]:= [ 1, 2, 3, 4, 5, 6, 7, 8 ];

```

4.19 HLA 数组常量

前一节的最后几个例子展示了 HLA 数组常量的使用。HLA 数组常量只是一组由一对中括号括起来的值。下面的数组常量都是合法的：

```

[ 1, 2, 3, 4 ]
[ 2.0, 3.14159, 1.0, 0.5 ]
[ 'a', 'b', 'c', 'd' ]
[ "Hello", "world", "of", "assembly" ]

```

注意，最后一个数组常量包含 4 个双字类型的指针，分别指向存储器中的 4 个 HLA 字符串。

在前一节中我们已经看到，可以在静态和只读段中使用数组常量，为数组变量提供初始值。当然，数组常量中由逗号分开的项的数量必须与变量声明中的数组元素数量相等。类似地，数组常量元素的类型也必须与数组变量元素的类型相同。

使用数组常量初始化小的数组是非常方便的。当然，如果数组含有几千个元素，逐个输入将会变得很繁琐。大多数使用这种方法进行初始化的数组都不超过几百个元素，一般都小于 100 个。使用数组常量为这样的变量初始化是可行的。然而，有时候用这种方式初始化数组可能会十分乏味，并且很容易出错。一般不会想要使用数组常量手动初始化一个具有 1000 个不同元素的数组。然而，如果想要将数组中所有的元素都初始化为同样的数值，HLA 提供了一种特殊的数组常量语法来完成这项任务。考虑下面的声明：

```

BigArray: uns32[ 1000 ] := 1000 dup [ 1 ];

```

这个声明创建了一个具有 1000 个元素的数组，将数组中的每一个元素都初始化为 1。1000 `dup[1]` 表达式的意思是告诉 HLA，通过将数值 1 重复 1000 遍创建一个数组常量。甚至可以使用

`dup` 操作符重复一个数值序列(而不是一个数值), 如下面的例子所示:

```
SixteenInts: int32[16] := 4 dup [ 1, 2, 3, 4 ];
```

这个例子将序列(1,2,3,4)重复了 4 遍, 共产生 16 个不同的整数(也就是 1、2、3、4、1、2、3、4、1、2、3、4、1、2、3、4), 然后用这些数值对 `SixteenInts` 进行初始化。

在 4.22 节要介绍的多维数组中, 将会看到 `dup` 操作符的更多用法。

4.20 访问一维数组的元素

访问以 0 为基准的数组的元素, 可以使用下面的简化公式:

$$\text{Element_Address} = \text{Base_Address} + \text{index} * \text{Element_Size};$$

其中, `Base_Address` 可以使用数组名(因为 HLA 将数组中第一个元素的地址与数组的名称相关联)。 `Element_Size` 是每一个数组元素的字节数。如果对象是一组字节, 则 `Element_Size` 的值是 1(一个非常简单的计算)。如果每一个数组元素都是一个字(或者是其他双字节类型), 则 `Element_Size` 的值是 2, 等等。访问上述 `SixteenInts` 数组的元素可以使用如下的公式(`Element_Size` 为 4, 因为每个元素都是一个 `int32` 对象):

$$\text{Element_Address} = \text{SixteenInts} + \text{index} * 4;$$

与 `eax:=SixteenInts[index]` 语句等价的 80x86 代码是:

```
mov( index, ebx );
shl( 2, ebx );           // Sneaky way to compute 4*ebx
mov( SixteenInts[ ebx ], eax );
```

在这里有两个问题必须注意。第一, 这段代码使用的是 `shl` 指令而不是 `intmul` 指令计算 `4*index`。使用 `shl` 的主要原因是它的效率更高。结果表明, 在许多处理器中, `shl` 都要比 `intmul` 更快。

第二个需要注意的问题是, 这个指令序列没有显式计算基址加上 4 倍索引的和, 而是通过变址寻址方式隐式地计算这个和。指令 `mov(SixteenInts[ebx], eax)` 从位置 `SixteenInts+ebx` 向 `EAX` 加载数据, 也就是基址加上 `index*4`(因为 `ebx` 中的值是 `index*4`)。当然也可以使用下列 5 个指令来代替上述的序列。

```
lea( eax, SixteenInts );
mov( index, ebx );
shl( 2, ebx );           // Sneaky way to compute 4*ebx
add( eax, ebx );         // Compute base address plus index*4
mov( [ebx], eax );
```

然而, 如果 3 个指令就可以完成工作, 为什么要使用 5 个指令来完成呢? 这也是为什么应该深刻理解寻址方式的一个很好的例子。选择适当的寻址方式可以减小程序量, 进而使之运行更快。

当然, 在讨论提高效率的时候, 还要指出, 80x86 的比例变址寻址方式可以自动将索引乘以

1、2、4 或者 8。因为这个例子将索引乘以 4，我们还可以通过使用比例变址寻址方式进一步简化代码：

```
mov( index, ebx );
mov( SixteenInts[ ebx*4 ], eax );
```

然而，要注意，如果想要将索引乘以除 1、2、4 或者 8 以外的常量，则不能使用比例变址寻址方式。类似地，如果想要乘以不是 2 的幂值的数值，则不能使用 `shl` 指令将索引乘以这个数值，而是要用 `intmul` 或者其他指令序列完成这样的操作。

80x86 的变址寻址方式是访问一维数组元素的一个很自然的选择。事实上，它的语法也暗示了一个数组访问。唯一需要记住的事情是，必须记住要将索引乘以元素的大小。如果忘记这样做，会产生错误的结果。

4.21 数组排序

当介绍数组的时候，几乎每一本书都会给出排序的例子。因为您可能已经知道如何在高级语言中进行排序，这对我们现在看 HLA 排序具有指导意义。本节的例子将使用冒泡排序，这种方法对于少量的数据或者已经接近排好顺序的数组来说是比较适合的，但是不适合于其他的情况¹⁷。

```
const
    NumElements := 16;
static
    DataToSort: uns32[ NumElements ] :=
    [
        1, 2, 16, 14,
        3, 9, 4, 10,
        5, 7, 15, 12,
        8, 6, 11, 13
    ];

    NoSwap: boolean;

-- // Bubble sort for the DataToSort array:
repeat
    mov( true, NoSwap );
    for( mov( 0, ebx ); ebx <= NumElements-2; inc( ebx ) ) do

        mov( DataToSort[ ebx*4 ], eax );
        if( eax > DataToSort[ ebx*4 + 4 ] ) then
            mov( DataToSort[ ebx*4 + 4 ], ecx );
            mov( ecx, DataToSort[ ebx*4 ] );
```

¹⁷ 不用担心，在第 5 章中，您会看到一些比较好的排序算法。

```

        mov( eax, DataToSort[ ebx*4 + 4 ] ); // Note: eax contains
        mov( false, NoSwap ); // DataToSort[ ebx*4 ]

    endif;

endfor;

until( NoSwap );

```

冒泡排序的工作方法是比较数组中相邻的元素。在这段代码中，值得注意的是它是如何比较相邻的元素的。可以看到 if 语句将 EAX(保存的数据是 DataToSort[ebx*4])与 DataToSort[ebx*4+4] 进行比较。因为这个数组的每个元素是 4 个字节(uns32)，索引[ebx*4+4]所指的是[ebx*4]之后的元素。

冒泡排序法的特点是，当最内层的循环结束，并且没有交换任何数据的时候，该算法就结束。如果数据已经排序过了，则冒泡排序法的效率是很高的，只需要遍历一次数据。遗憾的是，如果数据没有经过排序(最糟糕的情况是，如果数据是按反序排列的)，那么这个算法的效率会相当差。尽管可以对上述的代码进行修改，使它的工作速度平均提高两倍，但是这种优化对于这样一个算法来说也是无济于事的。然而，冒泡排序法的实现比较简单易懂(这也是为什么介绍性的内容依然用它作为例子的原因)。

4.22 多维数组

80x86 的硬件很容易处理一维数组。但是，80x86 没有提供有效的寻址方式，以便容易地访问多维数组的元素。访问多维数组的元素需要若干指令。

在讨论如何声明或访问多维数组之前，有必要介绍在存储器中它们是如何实现的。第一个需要指出的是，如何将多维对象存储到一维存储空间中去。

先考虑一下 A:array[0..3,0..3] of char;形式的 Pascal 数组。这个数组包含 16 个字节，这 16 个字节被组织成为 4 行，每一行有 4 个字符。您可能已经勾画出数组的 16 字节中的每一个字节在存储器中连续的存储空间的布局，如图 4-5 所示。

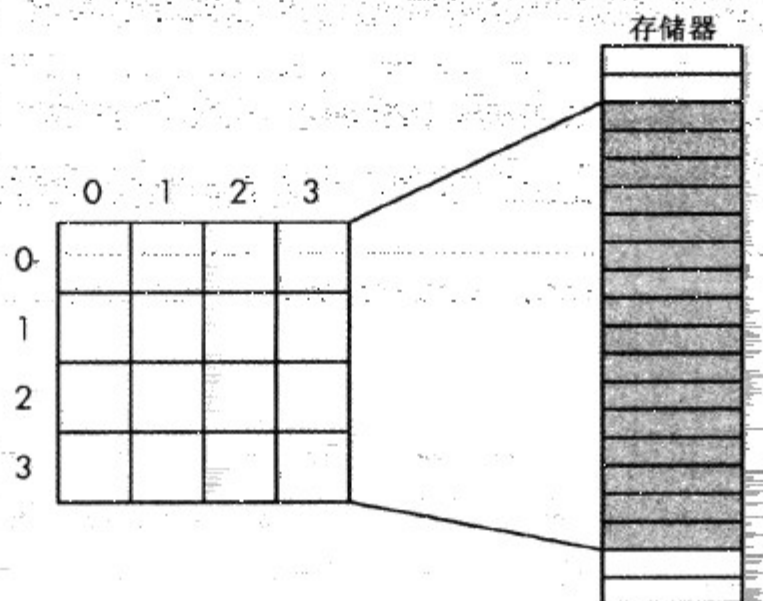


图4-5 将一个 4×4 数组映射到连续的存储空间

只要两件事是正确的，实际的映射方式就不重要了：①每个元素映射唯一的存储地址(也就是说，数组中没有两个元素共用一个存储位置)；②映射是稳定的。也就是说，对于给定的数组元素，它在存储器中的映射位置不变。因此，实际需要的是具有两个输入参数(行和列)的函数，这两个参数是 16 个存储位置的线性数组的偏移量。

任何满足上述限制的函数都可以使用。实际上，只要映射是稳定的，就可以任意选择一种一一对应的映射方法。然而，实际上需要的映射方法是，能够在运行时高效地进行计算，并且能够对任意大小的数组进行映射(不仅仅是 4×4 或者只针对二维数组)。尽管有许多函数都满足这些条件，但有两个函数是大多数的程序员和大多数高级语言所使用的，即以行为主排列和以列为主排列。

4.22.1 以行为主排列

以行为主排列的方法的顺序是，在连续的存储空间里，先将一行的数据排列完，然后再到下一行，直到完成。图 4-6 展示了这种映射方法。

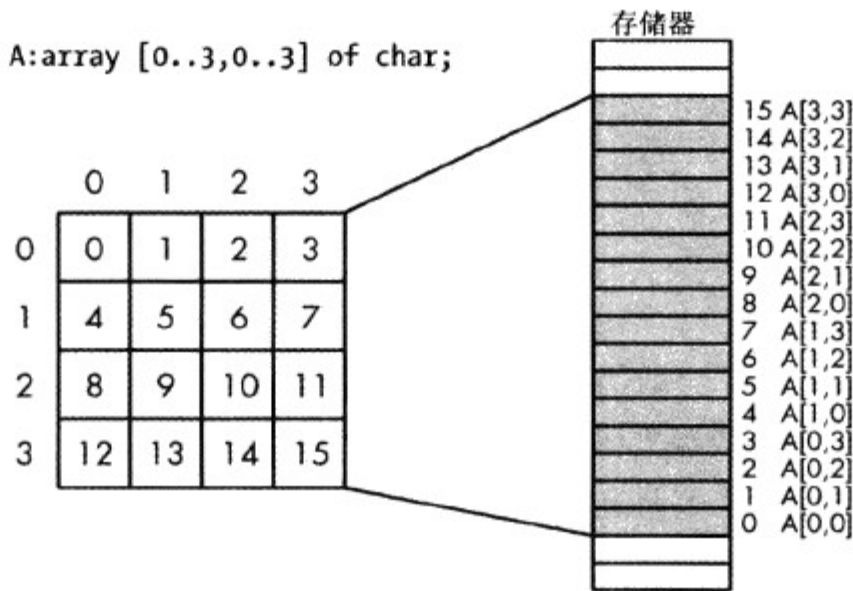


图 4-6 以行为主的数组元素排列

以行为主排列的方法是大多数高级语言所采用的方法。这种方法在机器语言中很容易实现，用起来也很简单。首先从第一行(行号是 0)开始，然后将第二行连接到第一行的后面。然后将第三行连接到第二行的后面，然后第四行，等等(见图 4-7)。

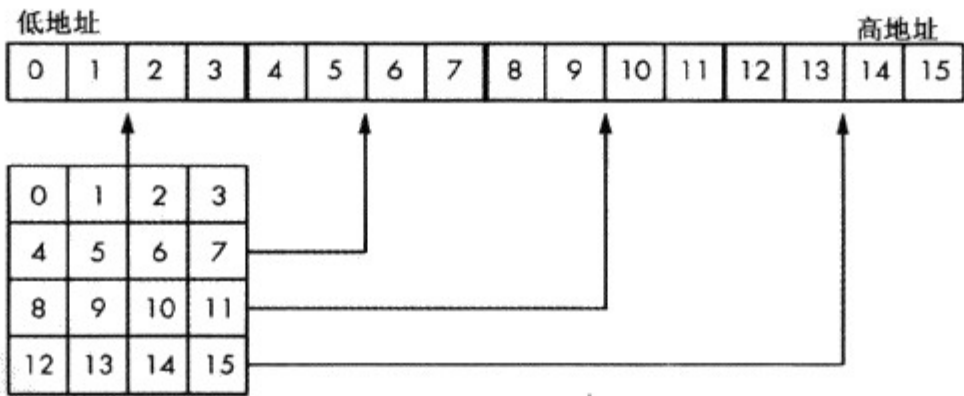


图 4-7 从另一角度看以行为主排列的 4×4 数组

实际上，将一组索引值转换为偏移量的函数是对计算一维数组元素地址的公式稍作修改得来的。计算以行为主排列的二维数组偏移量的公式是：

$$\text{Element_Address} = \text{Base_Address} + (\text{colindex} * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

同样地, *Base_Address* 是数组第一个元素的地址(也就是上述的 *A[0][0]*), *Element_Size* 是数组中元素的大小, 以字节表示。*colindex* 是数组中最左边的索引, *rowindex* 是最右边的索引。*row_size* 是数组中一行的元素数(上述情况中为 4, 因为每一行有 4 个元素)。假设 *Element_Size* 是 1, 这个公式计算得到相对基址的偏移量如下所示:

列索引	行索引	偏移量
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

对于三维数组, 计算存储器中偏移量的公式如下所示:

$$\text{Address} = \text{Base} + ((\text{depthindex} * \text{col_size} + \text{colindex}) * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

col_size 是一列当中的项数, *row_size* 是一行中的项数。在 C/C++ 中, 如果对数组的声明是 *type A[i][j][k]*; , 那么 *row_size* 是 *k*, *col_size* 是 *j*。

对于四维数组, 如果在 C/C++ 中对数组的声明为 *type A[i][j][k][m]*; , 则计算一个数组元素地址的公式是:

$$\text{Address} = \text{Base} + (((\text{LeftIndex} * \text{depth_size} + \text{depthindex}) * \text{col_size} + \text{colindex}) * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

depth_size 是 *j*, *col_size* 是 *k*, *row_size* 是 *m*。*leftindex* 代表的是最左边的索引。

到现在为止, 可以看一个模式了。有一个通用的公式, 可以为任何维数的数组计算存储器中的偏移量; 然而, 一般情况下不会超过四维。

另外一个比较方便的方法是, 可以将以行为主的数组看成是数组的数组。考虑下面的一维 Pascal 数组的定义:

```
A: array [0..3] of sometype;
```

假设 *sometype* 是 *sometype=array[0..3] of char* 类型的。

A 是一维数组。它的每一个元素都是数组, 但是暂时可以忽略这一点。计算一维数组元素地

址的公式为:

$$\text{Element_Address} = \text{Base} + \text{Index} * \text{Element_Size}$$

上述情况中的 *Element_Size* 取值为 4, 因为 A 的每一个元素都是一个 4 字符的数组。那么这个公式计算出来的是什么呢? 它计算出来的是这个 4×4 字符数组的每一行的基址(见图 4-8)。

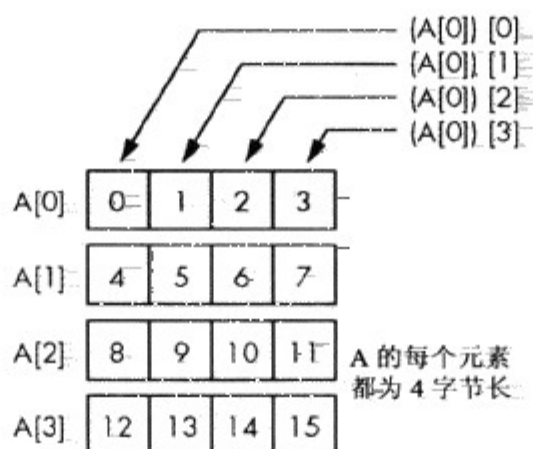


图 4-8 将 4×4 数组看作数组的数组

当然, 在计算出一行的基址后, 可以再使用一维公式来取得特定元素的地址。尽管这样并不会影响计算, 但是, 处理多个一维计算比起计算多维数组元素的地址要简单。

考虑一个定义为 `A:array[0..3][0..3][0..3][0..3][0..3] of char;` 的 Pascal 数组。可以将这个五维数组看做是一维数组的数组。如下所示的 HLA 代码提供了这样的定义:

```
type
    OneD: cChar[4];
    TwoD: OneD[4];
    ThreeD: TwoD[4];
    FourD: ThreeD[4];

var
    A: FourD[4];
```

OneD 的大小是 4 个字节。因为 TwoD 包含 4 个 OneD 数组, 所以它的大小是 16 字节。类似的, ThreeD 是 4 个 TwoD, 因此它的大小是 64 字节。最后, FourD 是 4 个 ThreeD, 因此是 256 字节。为了计算 `A[b,c,d,e,f]` 的地址, 可以采用下面的步骤:

(1) 用 $\text{Base} + b * \text{size}$ 公式计算 `A[b]` 的地址。在这里, *size* 是 256 字节。用这个结果作为进行下一次计算的新基址。

(2) 用 $\text{Base} + c * \text{size}$ 公式计算 `A[b,c]` 的地址, *Base* 是从上一步的结果中得到的, *size* 是 64。使用这个结果作为下一次计算的新基址。

(3) 用 $\text{Base} + d * \text{size}$ 公式计算 `A[b,c,d]` 的地址, *Base* 是从上一步的结果中得到的, *size* 是 16。使用这个结果作为下一次计算的新基址。

(4) 用 $\text{Base} + e * \text{size}$ 公式计算 `A[b,c,d,e]` 的地址, *Base* 是从上一步的结果中得到的, *size* 是 4。使用这个结果作为下一次计算的新基址。

(5) 最后用 $\text{Base} + f * \text{size}$ 公式计算 `A[b,c,d,e,f]` 的地址, *Base* 是从上一步的结果中计算得到的, *size* 是 1(显然可以忽略最后这个乘法)。得到的结果是特定元素的地址。

在汇编语言中不会有维数较多的数组，一个重要原因是，汇编语言强调这样的访问效率会很差。在 Pascal 程序中很容易使用类似于 $A[b,c,d,e,f]$ 的数组，而不清楚编译器对代码的编译。汇编语言程序员并不那么武断：他们会考虑到使用这样的多维数组将带来的麻烦。事实上，优秀的汇编语言程序员一般都会回避二维数组，当必须这样做的时候，他们会使用一定的技巧来访问这种数组中的数据。

4.22.2 以列为主排列

以列为主排列是另外一种经常用于计算数组元素地址的方法。FORTRAN 以及各种版本的 BASIC(例如 Microsoft BASIC 的老版本)都使用这个方法。

在以行为主排列中，在连续的存储空间中，最右边的索引上升得最快。在以列为主排列中，最左边的索引上升得最快。图 4-9 给出了以列为主排列的数组的组织方式的图示：

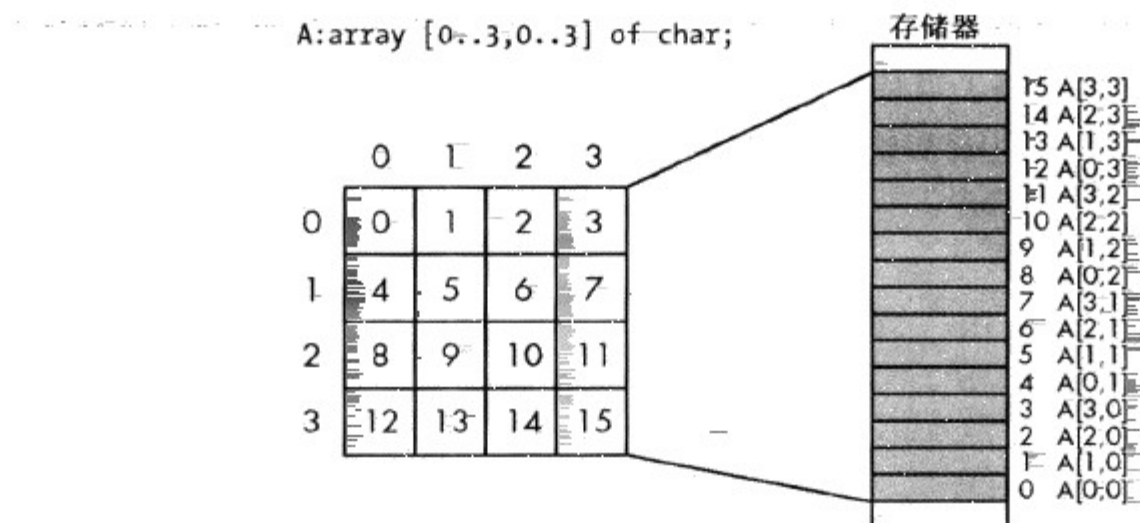


图 4-9 以列为主的数组元素排列

在使用以列为主排列的情况下，用于计算数组元素地址的公式与以行为主排列的情况下所使用的公式非常类似。只需要简单地反转计算中的索引和大小：

对于以列为主排列的二维数组：

$$\text{Element_Address} = \text{Base_Address} + (\text{rowindex} * \text{col_size} + \text{colindex}) * \text{Element_Size}$$

对于以列为主排列的三维数组：

$$\text{Address} = \text{Base} + ((\text{rowindex} * \text{col_size} + \text{colindex}) * \text{depth_size} + \text{depthindex}) * \text{Element_Size}$$

对于以列为主排列的四维数组：

$$\text{Address} = \text{Base} + (((\text{rowindex} * \text{col_size} + \text{colindex}) * \text{depth_size} + \text{depthindex}) * \text{Left_size} + \text{Leftindex}) * \text{Element_Size}$$

4.23 多维数组的存储空间分配

如果有 $m \times n$ 的数组，则总共有 $m * n$ 个元素，需要 $m * n * \text{Element_Size}$ 个字节的存储空间。为了给这样的数组分配存储空间，存储器中必须有足够的空间。通常，有许多不同的方法可以完成这项任务。幸运的是，HLA 的数组声明语法与高级语言的数组声明语法非常相似，因此，C/C++、

Java、BASIC 以及 Pascal 程序员会对此比较熟悉。在 HLA 中，可以使用如下所示的声明方式来声明一个多维数组：

```
ArrayName: elementType [ comma_separated_list_of_dimension_bounds ];
```

例如，下面是一个 4×4 的字符数组的声明：

```
GameGrid: char[4, 4];
```

下面是另外一个三维字符串数组声明的例子：

```
NameItems: string[2, 3, 3];
```

记住，字符串对象实际上是指针，因此这个数组声明为 18 个双字指针分配了存储空间(2*3*3=18)。

与一维数组的情况相同，可以在声明后使用赋值操作符和数组常量，将数组的每一个元素初始化为特定的值。数组常量忽略了维数信息；重要的是数组常量中元素的数量对应于实际数组的元素数量。下面的例子给出了具有初始值设定项的 GameGrid 声明：

```
GameGrid: char[ 4, 4 ] :=
[
    'a', 'b', 'c', 'd',
    'e', 'f', 'g', 'h',
    'i', 'j', 'k', 'l',
    'm', 'n', 'o', 'p'
];
```

注意，HLA 忽略了出现在这个声明中的缩排和空格字符(例如换行符)。使用这些缩排和空格字符是为了提高可读性(通常是一个好办法)。HLA 并不将 4 个分开的行解释成数组中的行，而人们会这样做，这也就是为什么要用这种方法表示初始化数据的原因。事实上，在数组常量中有 16(4*4)个字符即可。您可能也会同意，这样的描述方法比如下所示的描述方法更易读：

```
GameGrid: char[ 4,4 ] :=
[ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
  'o', 'p'];
```

当然，如果数组非常大，行的长度很大，或者深度很大，那么使这个数组易读是很困难的事情。这就需要书写注解，细致地解释所有的问题。

对于一维数组来说，可以使用 dup 操作符将一个非常大的数组中每一个元素初始化为相同的值。如下所示的例子对一个 256×64 的字节数组进行初始化，每个字节的内容都是\$FF：

```
StateValue: byte[256, 64]:= 256*64 dup [$ff];
```

注意，在这里用来计算数组元素个数的是常量表达式，而不是简单地使用常量 16384(256*64)。常量表达式的使用比起简单地使用常量 16384 能够更加清楚地表明，这段代码是对 256×64 的数组元素中的每一个元素进行初始化。

还有另外一个 HLA 技巧可以用来提高程序的可读性，也就是使用嵌套的数组常量。下面是 HLA 嵌套数组常量的例子：

```
[[0, 1, 2], [3, 4], [10, 11, 12, 13]]
```

每当 HLA 遇到某个数组常量嵌套在另外一个数组常量中的时候，会将嵌套的数组常量外层的中括号去掉，将整个常量看作是一个数组常量。例如，HLA 将上述嵌套的数组常量转换成以下的格式：

```
[0, 1, 2, 3, 4, 10, 11, 12, 13]
```

可以利用这个功能使程序更加易读。对于多维数组常量，可以将每一行的常量用中括号括起来，代表每一行的数据为一组，与其他的行分开。作为例子，考虑下面对于 GameGrid 数组的声明，这与前面的声明是完全相同的(就 HLA 而言)：

```
GameGrid: char[ 4,4 ] :=
[
    [ 'a', 'b', 'c', 'd'],
    [ 'e', 'f', 'g', 'h'],
    [ 'i', 'j', 'k', 'l'],
    [ 'm', 'n', 'o', 'p']
];
```

这个声明更清楚地表明了，该数组常量是一个 4×4 的数组，而不仅仅是一个含有 16 个元素，且元素不能放到一行中的一维数组。类似这样的改进就是普通程序员和优秀程序员之间的区别。

4.24 汇编语言中多维数组元素的访问

前面我们已经看过了计算多维数组元素地址的公式。现在就可以来看如何使用汇编语言访问这些数组元素。

mov、shl 和 intmul 指令简化了上述各种计算多维数组偏移量的公式。下面先考虑一个二维数组：

```
static
    i:          int32;
    j:          int32;
    TwoD:       int32[ 4, 8 ];
    .
    .
    .

// To perform the operation TwoD[i, j] := 5; you'd use code like the following.
// Note that the array index computation is (i*8 + j)*4.

    mov( i, ebx );
    shl( 3, ebx );      // Multiply by 8 (shl by 3 is a multiply by 8).
    add( j, ebx );
    mov( 5, TwoD[ ebx*4 ] );
```

注意, 这段代码不需要使用 80x86 中的两个寄存器寻址方式。尽管类似于 `TwoD[ebx][esi]` 的寻址方式好像是访问二维数组的一种很自然的选择, 但这并不是这种寻址方式的目的。

现在让我们来看另外一个使用三维数组的例子:

```
static
    i:          int32;
    j:          int32;
    k:          int32;
    ThreeD:     int32[ 3,4,5 ];
    .
    .
    .

// To perform the operation ThreeD[i,j,k] := esi; you'd use the following code
// that computes  $(i*4 + j)*5 + k$  as the address of ThreeD[i,j,k].

    mov( i, ebx );
    shl( 2, ebx ); // Four elements per column.
    add( j, ebx );
    intmul( 5, ebx ); // Five elements per row.
    add( k, ebx );
    mov( esi, ThreeD[ ebx*4 ] );
```

注意, 这段代码使用 `intmul` 指令将 `EBX` 中的值乘以 5。记住, `shl` 指令只能对寄存器进行 2 的乘方运算。尽管有许多方法可以将寄存器中的值乘以常量而不是计算 2 的乘方, `intmul` 指令的使用还是比较方便的¹⁸。

4.25 记录

另外一种主要的复合数据结构是 Pascal 记录或者 C/C++/C# 结构¹⁹。Pascal 术语可能会比较好, 因为这样做可以避免与更加通用的术语“数据结构”发生混淆。由于 HLA 使用术语“记录”, 所以在这里我们仍然沿用这个术语。

数组是同质的, 它的元素类型都是相同的, 而一个记录中的元素可以是不同的类型。数组可以通过整数索引选择特定的元素。而对于记录, 则必须通过名称(也就是字段)来对元素进行选择。

记录的目的是为了将不同的但是逻辑上有联系的数据封装到一个包中。比较典型的例子就是对 `student` 的 Pascal 记录声明:

```
student =
    record
        Name:      string[64];
        Major:     integer;
        SSN:       string[11];
        Midterm1:  integer;
```

¹⁸ 关于用常量做乘法而不是 2 的乘方的讨论将在第 4 章中进行。

¹⁹ 在其他的语言中可能还会使用其他的名称, 但是大多数人至少认识这些名称中的一个。

```

Midterm2:    integer;
Final:       integer;
Homework:    integer;
Projects:    integer;

end;

```

大多数 Pascal 编译器将记录中的每一个字段分配在连续的存储空间中。这就意味着, Pascal 将用前 65 个字节保存名字²⁰, 随后的两个字节保存主修课代号, 再后面 12 个字节保存社会保险号码, 等等。

在 HLA 中, 还可以使用 `record/endrecord` 声明创建记录类型。在 HLA 中, 可以将上述记录编码如下:

```

type
  student:    record
    Name:     char[65];
    Major:    int16;
    SSN:      char[12];
    Midterm1: int16;
    Midterm2: int16;
    Final:    int16;
    Homework: int16;
    Projects: int16;
  endrecord;

```

正如所看到的一样, HLA 声明与 Pascal 声明非常相似。注意, 这个例子中的 Name 和 SSN(美国社会保险号码)字段使用字符数组而不是字符串的目的是为了与 Pascal 声明相同。在实际的 HLA 记录声明中, 至少要对名字使用字符串类型(一定要记住, 字符串变量仅仅是一个 4 字节的指针)。

记录中的字段名必须是唯一的。也就是说, 在同一个记录中, 一个名字不能出现两次或者两次以上。然而, 所有的字段名对于记录来说都是局部的。因此, 可以在程序的其他地方或者不同的记录中使用相同的字段名。

`record/endrecord` 声明可以出现在某个变量声明段(例如 `static` 或者 `var`)当中, 或者出现在某个 `type` 声明段中。在上述的例子中, Student 声明出现在 `type` 段, 因此这并没有为 Student 变量实际地分配存储空间。所以必须显式声明一个 Student 类型的变量。下面的例子展示了该如何进行:

```

var
  John: Student;

```

这就在存储器中分配了 81 个字节的存储空间, 如图 4-10 所示。

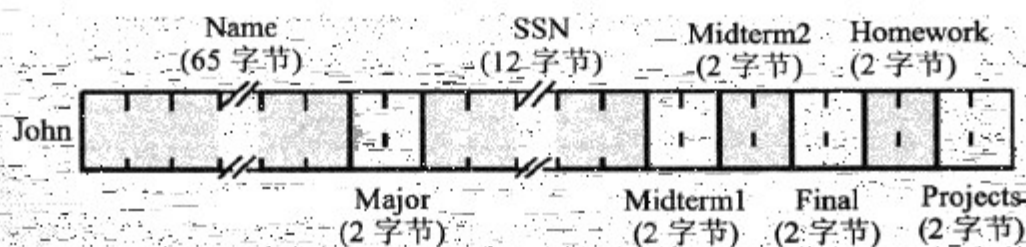


图 4-10 Student 数据结构在存储器中的保存

²⁰ 除了字符串中所有的字符外, 字符串需要一个多余的字节来对长度进行编码。

如果标签 John 对应于这个记录的基址, 则 Name 字段的偏移量是 John+0, Major 字段的偏移量是 John+65, SSN 字段的偏移量是 John+67, 等等。

为了访问结构的某个成员, 需要知道相应的字段相对于结构开始处的偏移量。例如, 在变量 John 中, Major 字段相对于 John 的基址的偏移量是 65。因此, 可以使用如下所示的指令将 AX 中的数据存储到这个字段中:

```
mov( ax, (type word John[65]) );
```

遗憾的是, 如果需要记住一个记录中所有字段的偏移量, 那么就会丧失了使用记录的必要性。毕竟, 如果必须要处理这些数值偏移量, 那么为什么不直接使用字节数组, 而要使用记录呢?

事实上, HLA 使用了与 C/C++/C#和-Pascal 相同的机制引用字段的名称: 圆点操作符。要将 AX 存储到 Major 字段中, 可以使用 `mov(ax, John.Major)`; 而不用使用上述的指令。这样的语句就比较易读, 当然也较容易使用。

注意, 圆点操作符的使用并不是引入了一种新的寻址方式。指令 `mov(ax, John.Major)` 仍然使用的是位移寻址方式。HLA 只是将 John 的基址和 Major 字段的偏移量(65)相加, 得到实际的位移再编码到指令中。

与其他的类型声明相同, HLA 要求, 在使用之前所有的记录类型声明必须出现在程序当中。然而, 并不是必须在 `type` 段中定义所有的记录来创建记录变量。可以在变量声明段中直接使用 `record/endrecord` 声明。如果在程序中只有一个给定记录对象的实例, 这种做法是很方便的。下面的例子展示了这种做法:

```
storage=
    OriginPoint: record
        x: uns8;
        y: uns8;
        z: uns8;
    endrecord;
```

4.26 记录常量

HLA 允许定义记录常量。实际上, HLA 同时支持符号记录常量和字面记录常量。记录常量在初始化静态记录变量的时候非常有用。在使用 HLA 编译时语言的时候, 它们作为编译时的数据结构也很有用(查阅 HLA 参考手册可以找到关于 HLA 编译时语言的详细信息)。本节将要讨论如何创建记录常量。

字面记录常量的形式如下所示:

```
RecordTypeName: [ List_of_comma_separated_constants ]
```

RecordTypeName 是使用常量之前, 已经在 HLA 的 `type` 段中定义的一个记录数据类型的名称。

中括号之间出现的常量列表是指定记录中每一个字段的数据。列表中的第一个项对应的是记录中的第一个字段, 第二个项对应的是记录中的第二个字段, 等等。在这个列表中出现的每一个常量的数据类型必须与各自的字段类型相符。下面的例子展示了如何使用字面记录常量初始化记

录变量:

```
type
    point: record
        x:int32;
        y:int32;
        z:int32;
    endrecord;

static
    Vector: point := point:[1, -2, 3];
```

这个声明分别将 Vector.x、Vector.y、Vector.z 声明为 1、-2、3。

还可以通过在程序的 const 或者 val 段中声明记录对象创建符号记录常量。访问这些符号记录常量中字段的方法与访问记录变量中字段的方法相同，都可以使用圆点操作符。因为对象是常量，可以在任何字段类型常量合法的地方指定这个记录常量的字段。还可以用符号记录常量初始化记录变量。下面的例子展示了这种情况：

```
type
    point: record
        x:int32;
        y:int32;
        z:int32;
    endrecord;

const
    PointInSpace: point := point:[1, 2, 3];

static
    Vector:point :=PointInSpace;
    XCoord:int32 :=PointInSpace.x;
    .
    .
    .

    stdout.put( "Y Coordinate is ", PointInSpace.y, nl );
    .
    .
    .
```

4.27 记录数组

创建记录数组的操作是十分必要的。在创建的过程中，只需要创建一个记录类型，然后使用标准数组声明语法。下面的例子展示了如何声明记录数组：

```
type
    recElement:
        record
            << Fields for this record >>
```



```
endrecord;
```

```
static
```

```
recArray: recElement[4];
```

为了访问这个数组的某个元素,可以使用标准的数组索引技术。因为 *recArray* 是一维数组,所以可以使用公式 $baseAddress + index * @size(recElement)$ 对元素的地址进行计算。例如,为了访问 *recArray* 的某个元素,可以使用下面的代码:

```
// Access element i of recArray:
```

```
intmul( @size(recElement), i, ebx ); // ebx := i * @size( recElement )
mov( recArray.someField[ebx], eax );
```

注意,索引描述出现在变量全名的后面。记住,这是汇编语言,不是高级语言(在高级语言中,可能可以使用 *recArray[i].someField*)。

自然,还可以创建多维记录数组。可以使用以行为主或者以列为主的排列方法计算这样一个记录中的元素地址。与对数组的讨论相比,唯一发生变化的是每个元素的大小是记录对象的大小。

```
static
```

```
rec2D: recElement[ 4, 6 ];
```

```
// Access element [i,j] of rec2D and load someField into eax:
```

```
intmul( 6, i, ebx );
add( j, ebx );
intmul( @size(recElement), ebx );
mov( rec2D.someField[ ebx ], eax );
```

4.28 数组/记录作为记录字段

记录可以包含其他的记录或者数组作为字段。考虑下面的定义:

```
type
```

```
Pixel:
```

```
record
```

```
Pt: point;
```

```
color: dword;
```

```
endrecord;
```

上述的代码定义了一个具有 32 位颜色分量的点。当初始化 *Pixel* 类型的对象时,第一个初始值设定项对应的是 *Pt* 字段,而不是 *x* 坐标字段。如下所示的定义是不正确的:

```
static
    ThisPt: Pixel := Pixel:[ 5, 10 ];           // Syntactically incorrect!
```

第一个字段的值(5)不是 `point` 类型的对象, 因此当遇到这样的语句时, 汇编器就会产生错误。HLA 允许使用类似于如下所示的声明初始化 `Pixel` 字段:

```
static
    ThisPt:Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
    ThatPt:Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];
```

访问 `Pixel` 字段很简单。像高级语言一样, 使用圆点引用 `Pt` 字段, 然后再使用圆点访问 `point` 的 `x`、`y`、`z` 字段:

```
    stdout.put("ThisPt.Pt.x = ", ThisPt.Pt.x, nl);
    stdout.put("ThisPt.Pt.y = ", ThisPt.Pt.y, nl);
    stdout.put("ThisPt.Pt.z = ", ThisPt.Pt.z, nl);
    .
    .
    .
mov( eax, ThisPt.Color );
```

还可以将数组声明为记录字段。下面的代码创建了一个数据类型, 可以使用 8 个点代表一个对象(也就是立方):

```
type
    Object8:
        record
            Pts:    point[8];
            Color: dword;
        endrecord;
```

这个记录为 8 个不同的点分配存储空间。要访问 `Pts` 数组的元素, 必须知道 `point` 类型对象的大小(记住, 必须用数组元素的索引乘以该元素的大小, 对于这个例子是 12)。例如, 假设有一个 `Object8` 类型的变量 `Cube`。可以用如下所示的方法访问 `Pts` 数组的成员:

```
// cube.Pts[i].x := 0;

    mov( i, ebx );
    intmul( 12, ebx );
    mov( 0, Cube.Pts.x[ebx] );
```

在所有这些当中存在一个不足就是必须要知道 `Pts` 数组中每一个元素的大小。幸运的是, 可以使用 `@size` 重写上述的代码:

```
// Cube.Pts[i].x := 0;

    mov( i, ebx );
    intmul( @size(point), ebx );
    mov( 0, Cube.Pts.x[ebx] );
```

注意, 在这个例子中, 尽管数组是 `Pts`, 不是 `x`, 索引描述(`[ebx]`)还跟在对象的全名后。记住, `[ebx]`描述是一种变址寻址方式, 而不是数组的索引。索引总是放在全名的后面, 不要像在高级语言 C/C++ 或者 Pascal 中一样将它们附加到数组成员中。这样就能产生正确的值, 因为加法是可以交换的, 圆点操作符(以及索引操作符)对应着加法。特别地, 表达式 `Cube.Pts.x[ebx]`告诉 HLA 计算 `Cube`(对象的基址)加上 `Pts` 字段的偏移量, 加上 `x` 字段的偏移量, 加上 `EBX` 中的值。我们其实就是计算偏移量(`Cube`)+偏移量(`Pts`)+`EBX`+偏移量(`x`), 但是由于加法的可交换性, 所以我们可以对它重新进行安排。

还可以在记录中定义二维数组。访问这种数组中的元素与访问其他二维数组几乎相同, 唯一的不同之处是必须指定数组字段的名称作为数组的基址。例如:

```

type
    RecW2DArray:
        record
            intField:  int32;
            aField:    int32[4,5];
            .
            .
            .
        endrecord;

static
    recVar: RecW2DArray;
    .
    .
    .
    // Access element [i,j] of the aField field using row-major ordering:

    mov( i, ebx );
    intmul( 5, ebx );
    add( j, ebx );
    mov( recVar.aField[ ebx*4 ], eax );
    .
    .
    .

```

上述代码使用标准的以行为主的方法计算 4×5 的双字数组的索引。这个例子与单独的数组访问之间存在的唯一不同之处在于, 基址是 `recVar.aField`。

有两种方法可以嵌套记录定义。本节前面已经提到, 可以在 `type` 段中创建记录类型, 然后将这个类型名称用作记录中某个字段的数据类型(例如, 上述 `Pixel` 数据类型中的 `Pt:point` 字段)。还可以在其他记录中直接声明记录, 而不需为记录创建单独的数据类型, 下面的例子展示了这种情况:

```

type
    NestedRecs:
        record
            iField: int32;
            sField: string;
            rField:

```

```

        record
            i:int32;
            u:uns32;
        endrecord;
    cField:char;
endrecord;

```

一般来说，创建单独的类型比起将记录直接插入其他记录中要更好一些，但是这种嵌套是完全合法的。

如果存在一个记录数组，并且该记录类型的某个字段是数组，那么就必须对数组的索引分别进行计算，然后将这些索引的和作为最终的索引。下面的例子展示了这种做法：

```

type
    recType:
        record
            arrayField: dword[4,5];
            << Other fields >>
        endrecord;

static
    aryOfRecs: recType[3,3];
    .
    .
    .

    // Access aryOfRecs[i,j].arrayField[k,l]:

    intmul( 5, i, ebx );      // Computes index into aryOfRecs
    add( j, ebx );           // as (i*5 + j)*@size( recType ).
    intmul( @size(recType), ebx );

    intmul( 3, k, eax );      // Computes index into arrayOfRecs
    add( l, eax );           // as (k*3+ j)(*4 handled later).

    mov( aryOfRecs.arrayField[ ebx + eax*4 ], eax );

```

注意，基址加上比例变址寻址方式使这个操作变得更加简单。

4.29 对齐记录中的字段

为了使程序取得最好的性能，或者确保 HLA 的记录正确映射到某个高级语言中的记录或者结构中，可能会经常需要对记录中字段的对齐进行控制。例如，有时候可能会需要确保某个双字字段的偏移量是 4 的偶数倍。可以使用 align 指令做到这一点。下面的例子演示了如何在重要边界上对齐字段：

```

type
    PaddedRecord:
        record
            c: char;

```



```

        align(4);
d:      dword;
b:      boolean;
        align(2);
w:      word;

endrecord;

```

当 HLA 在某个记录声明中遇到 **align** 指令时, 它就会自动调整后面一个字段的偏移量, 使它变成 **align** 指令所指定数值的整数倍。如果有必要的话, 它会通过增加该字段的偏移量来达到目的。在上述的例子中, 各个字段的偏移量为: c:0、d:4、b:8、w:10。注意, HLA 在 c 和 d 之间插入了 3 个填充字节, 在 b 和 w 之间插入了 1 个填充字节。不言而喻, 任何时候都不能假设这些填充字节的存在。如果想要使用这些多余的字节, 则需要为它们声明字段。

注意, 在记录声明中指定对齐方式并不能确保这些字段会在存储器的某一边界上对齐, 它只能保证字段的偏移量是所指定数值的倍数。如果 **PaddedRecord** 类型的变量从存储器中的某个奇数地址开始, 那么 d 字段也将从奇数地址开始(因为任何一个地址加上 4 还是奇数地址)。如果想要确保这些字段被排列在存储器适当的边界上, 则必须在这个记录类型变量的声明前使用 **align** 指令, 例如:

```

static

        align(4);
PRvar: PaddedRecord;

```

align 操作数的值应该是偶数, 可以被记录类型中最大的 **align** 表达式除尽(对应上述例子, 4 是最大值, 它可以被 2 除尽)。

如果想要确保记录的大小是某个数值的倍数, 则只需要简单地将 **align** 指令作为记录声明的最后一项。HLA 将会在记录的最后增加适当数量的填充字节使之成为适当的大小。下面的例子展示了如何确保记录的大小是 4 字节的倍数:

```

type
    PaddedRec:
        record
            << Some field declarations >>
            align(4);
        endrecord;

```

HLA 为记录还提供了其他的对齐指令, 可以很容易地控制记录中所有字段的对齐方式, 以及记录中字段的初始偏移量。如果有兴趣了解关于记录字段对齐的更多信息, 请查阅 HLA 参考手册。

4.30 记录指针

在运行过程当中, 程序可能会使用指针间接引用记录对象。当使用指针访问某个结构的字段

时, 必须将相应记录的地址装载到 80x86 的一个 32 位寄存器当中。假设有如下所示的变量声明(假定是前面小节中的 Object8 结构):

```
static
    Cube:      Object8;
    CubePtr:   pointer to Object8 := &Cube;
```

CubePtr 保存着 Cube 对象的地址(即它是指向该对象的指针)。为了访问 Cube 对象的 Color 字段, 可以使用类似于 `mov(Cube.Color, eax);` 的指令。当通过指针对某个字段进行访问的时候, 需要将对象的地址装载到某个 32 位的寄存器中, 如 EBX。指令 `mov(CubePtr, ebx);` 将会完成这项任务。然后, 可以使用 `[ebx+offset]` 寻址方式访问 Cube 对象的字段。唯一的问题是“如何指定访问哪一个字段”。考虑如下所示错误的代码:

```
mov( CubePtr, ebx );
mov( [ebx].Color, eax );    // This does not work!
```

上述代码中存在一个主要的问题。因为对于一个结构来说字段名是局部的, 因此可以在两个或者多个结构中重用一個字段名, HLA 如何确定 Color 代表哪一个偏移量? 当直接访问结构成员的时候(例如 `mov(Cube.Color, eax);`), 就不会存在含糊的情况, 因为 Cube 具有汇编器可以检查的特定类型。而另外一方面, `[ebx]` 可以指向任何地方。特别地, 它可以指向任何一个包含 Color 字段的结构。因此汇编器本身不能确定应该对 Color 符号使用哪一个偏移量。

为了消除这种不确定性, HLA 要求显式提供一个类型。这样就必須强制 `[ebx]` 为 Cube 类型。只要这样做以后, 就可以使用常规的圆点操作符访问 Color 字段:

```
mov( CubePtr, ebx );
mov( (type Cube [ebx]).Color, eax );
```

如果有一个指向记录的指针, 并且该记录的一个字段是数组, 则访问该字段元素的最简单的方法就是采用基址加上变址的寻址方式。为了达到这个目的, 只需要将指针值装载到一个寄存器中, 在第二个寄存器中计算数组的索引。然后就可以在地址表达式中将这两个寄存器组合起来。在上述例子中, Pts 字段是一个具有 8 个 point 对象的数组。为了访问 Cube.Pts 字段的第 i 个元素的字段 x, 可以使用类似于下面的代码:

```
mov( CubePtr, ebx );
intmul( @size(point), i, esi );    // Compute index into point array.
mov( (type Object8 [ebx]).Pts.x [ esi*4 ], eax );
```

如果在程序中经常使用指向特定记录类型的指针, 用键盘输入一个强制操作符, 如 `(type Object8[ebx])`, 会很快失效。一种减小通过键盘输入来强制 EBX 的方法是使用 text 常量。例如, 考虑如下所示的语句:

```
const
    08ptr: text := "(type Object8 [ebx])";
```

如果程序在一开始就使用这样的语句, 那么可以使用 08ptr 代替类型强制转换操作符, HLA

将会自动替换相应的文本。有了上述文本常量，前面的例子将会变得易读、易写：

```
mov( CubePtr, ebx );
intmul( @size(point), i, esi );    // Compute index into point array.
mov( 08Ptr.Pts.x[ esi*4 ], eax );
```

4.31 联合

记录定义根据各个字段的大小将不同的偏移量赋值给记录中的每一个字段。这种行为十分类似于 `var` 或者静态段中存储器偏移量的分配。HLA 提供了另外一种结构声明，也就是联合(`union`)，它不是将不同的地址赋给每一个对象，而是联合声明中的每一个字段都具有相同的偏移量 0。下面的例子展示了联合声明的语法：

```
type
    unionType:
        union
            << Fields (syntactically identical to record declarations) >>
        endunion;
```

访问联合中的字段与访问记录中字段的方法完全相同：使用圆点符号和字段名。下面是 `union` 类型声明以及 `union` 类型变量的具体例子：

```
type
    numeric:
        union
            i: int32;
            u: uns32;
            r: real64;
        endunion;
    .
    .
    .
static
    number: numeric;
    .
    .
    .
    mov( 55, number.u );
    .
    .
    .
    mov( -5, number.i );
    .
    .
    .
    stdout.put( "Real value = ", number.r, nl );
```

对于联合对象，有一个重要的问题需要注意，联合结构中的所有字段都具有相同的偏移量。在上面的例子中，`number.u`、`number.i` 和 `number.r` 的偏移量都是 0。因此，联合中的各个字段在存储器中是重叠的，这非常类似于 80x86 的 8 位、16 位、32 位寄存器相互重叠的情况。通常，访问联合的各个字段都是相互排斥的，也就是说，不能同时操作联合变量的各个字段，因为对一个字段的写操作会覆盖另外一个字段。在上述例子中，对 `number.u` 的任何修改都会改变 `number.i` 和 `number.r`。

程序员在两种典型的情况下使用联合：保留内存空间或者创建别名。内存保留是这个数据结构的专门职能。我们将上述的 `numeric` 联合与对应的记录结构进行比较，看一下这个数据结构是如何工作的：

```
type
    numericRec:
        record
            i: int32;
            u: uns32;
            r: real64;
        endrecord;
```

如果声明了一个 `numericRec` 类型的变量，假设是 `n`，那么就可以用 `n.i`、`n.u` 和 `n.r` 的形式访问各个字段，就好像已经声明变量类型为 `numeric` 类型。这两者之间的区别是，`numericRec` 变量为记录的每一个字段分配单独的存储空间，而 `numeric`(联合)对象为所有的字段分配相同的存储空间。因此，`@size(numericRec)` 是 16，因为记录中包含两个双字字段和一个四字(`real64`)字段。然而，`@size(numeric)` 是 8。这是因为联合中的所有字段都占用相同的存储空间，则联合对象的大小是对象中最大字段的大小(见图 4-11)。

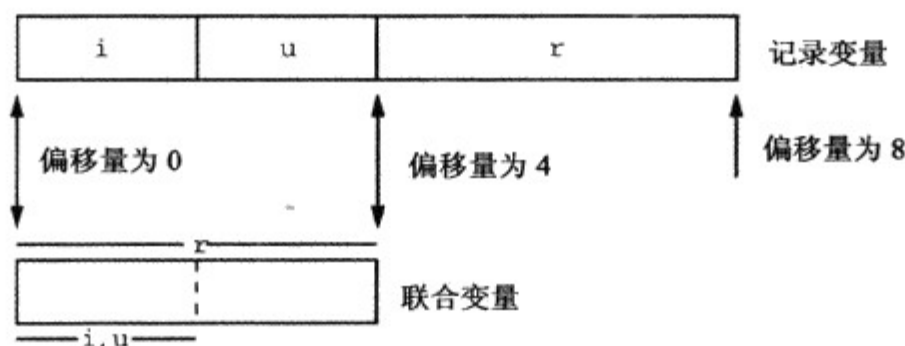


图 4-11 联合变量和记录变量

除了保留内存空间以外，程序员还经常使用联合在代码中创建别名。我们知道，别名是同一个内存对象的另外一个名字。别名经常会在程序中造成混乱，因此使用它们的时候一定要小心。然而，有时使用别名是很方便的。例如，在程序中的某个部分中，可能经常需要使用类型强制来用不同的类型指向一个对象。尽管可以使用 `HLA text` 常量简化这个过程，但另外一个方法就是使用联合变量，这个变量中的字段代表想要对该对象使用的不同类型。作为例子，考虑下面的代码：

```
type
    CharOrUns:
        union
            c:char;
            u:uns32;
        endrecord;
```



```
static
    v:CharOrUns;
```

使用上述声明，可以通过访问 `v.u` 操作 `uns32` 对象。如果在某些时候，需要将这个 `uns32` 变量中的低位字节看作字符，只需要简单地访问 `v.c` 变量。例如：

```
mov( eax, v.u );
stdout.put( "v, as a character, is '", v.c, "'" nl );
```

在 HLA 程序中，可以采用与使用记录完全相同的方法使用联合。联合声明可以作为记录中的字段出现，记录声明可以作为联合中的字段出现，联合中还可以出现数组声明，也可以创建联合数组，等等。

4.32 匿名联合

在记录声明当中，可以放置一个联合，而不用指定该联合对象的字段名。下面的例子展示了这样的语法：

```
type
    HasAnonUnion:
        record
            r:real64;
            union
                u:uns32;
                i:int32;
            endunion;
            s:string;
        endrecord;

static
    v: HasAnonUnion;
```

当记录中出现匿名联合的时候，可以访问联合的字段，就像它们直接是记录的字段一样。例如，在上述例子中，可以分别使用 `v.u` 和 `v.i` 语法访问 `v` 的 `u` 和 `i` 字段。`u` 和 `i` 字段在记录中具有相同的偏移量(8，因为它们前面是一个 `real64` 对象)。V 的字段相对于 `v` 的基址的偏移量为：

<code>v.r</code>	0
<code>v.u</code>	8
<code>v.i</code>	8
<code>v.s</code>	12

`@size(v)`是 16，因为 `u` 和 `i` 字段只占用 4 个字节。

HLA 也允许在联合中出现匿名记录。更多的细节请参考 HLA 文档，语法和用法与记录中的匿名联合相同。

4.33 变体类型

联合在程序中的一个重要的用途就是创建变体类型。在程序的运行过程中，变体变量可以动态地改变它的类型。在程序中，变体对象可以在某个地方是整型，在另外一个地方是字符串类型，然后在其他地方又改为实数类型。许多特高级语言(VHLL)系统都使用动态类型系统(也就是变体对象)来减小整个程序的复杂度。事实上，许多特高级语言的支持者都坚持认为，动态类型系统的使用是仅仅使用短短的几行特高级语言代码就能够编写复杂程序的主要原因之一。当然，如果在特高级语言中可以创建变体对象，那么在汇编语言当中也可以。在本节中，我们来看一下如何使用联合结构创建变体类型。

在程序执行的每一个瞬间，每一个变体对象都有特定的类型，但是在程序的控制下，变体对象可以改为其他的类型。因此，当程序在处理变体对象的时候，必须使用 `if` 语句或者 `switch` 语句(或其他类似的语句)，基于对象的当前类型执行不同的指令序列。特高级语言对这样的处理都是透明的。在汇编语言中，则必须自己提供代码来测试类型。为了做到这一点，除了对象值以外，变体类型还需要一些额外的信息。特别地，变体对象需要一个字段，能够描述对象的当前类型。这个字段(通常称作 `tag` 字段)是一个枚举类型或者整数类型，在任何给定的时刻对对象当前的类型进行描述。下面的代码展示了如何创建变体类型：

```
type
    VariantType:
        record
            tag:uns32;    // 0-uns32, 1-int32, 2-real64
            union
                u:uns32;
                i:int32;
                r:real64;
            endunion;
        endrecord;

static
    v:VariantType;
```

程序将会通过测试 `v.tag` 字段确定 `v` 对象的当前类型。程序基于这个测试操作 `v.i`、`v.u`、`v.r` 字段。

当然，当操作变体对象的时候，程序代码必须经常地测试 `tag` 字段，并对应 `uns32`、`int32` 或者 `real64` 值执行不同的指令序列。如果需要经常使用变体字段，那么编写过程来处理这些操作(例如，`vadd`、`vsub`、`vmul` 以及 `vdiv`)是一个好主意。

4.34 命名空间

记录和联合有一个很好的特性，即对于给定的记录或者联合声明，字段名是局部的。也就是说，可以在不同的记录或者联合中重用这些字段名。这是 HLA 的一个很重要的特点，因为这有助于避免命名空间污染(namespace pollution)。如果在当前的程序内用尽了所有的“好”名称，而不得不开始为一些对象创建描述性很差的名称，因为已经将大多数适当的名称用到其他地方了，

这时候就产生了命名空间污染。我们使用术语“命名空间”描述 HLA 如何将名称与特定的对象相关联。记录的字段名的命名空间仅限于该记录类型的对象。HLA 扩展了这个命名空间机制，使得能够创建任意的命名空间。这些命名空间对象能够对常量、类型、变量以及其他对象的名称进行保护，使它们的名称不受程序中其他声明的影响。

HLA 的 namespace 段封装了一组通用的声明，十分类似于记录封装一组变量声明。namespace 声明的形式如下所示：

```
namespace name;
    << declarations >>
end name;
```

name 标识符为 namespace 提供了名称。end 子句之后的标识符必须与 namespace 之后的标识符完全相符。注意，namespace 声明段是一个独立的段，它没有必要一定要出现在 type 或者 var 段中。namespace 可以出现在 HLA 声明段合法的任意地方。一个程序可以包含任意多个 namespace 声明；实际上，命名空间标识符甚至不必是唯一的，下面将会看到。

出现在 namespace 和 end 子句之间的声明都是标准 HLA 声明段，但是不能嵌套 namespace 声明。然而，可以在 namespace 中放置 const、val、type、static、readonly、storage 段²¹。下面的例子提供了 HLA 程序中的一个典型的 namespace 声明：

```
namespace myNames;

    type
        integer: int32;

    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

为了访问命名空间的某个字段，可以像记录和联合那样使用圆点符号。例如，为了在命名空间外部访问 myName 的字段，可以使用如下所示的标识符：

myNames.integer	类型声明，等值于 int32
myNames.i	integer 变量(int32)
myNames.j	uns32 变量
myNames.pi	real64 常量

这个例子还表明关于 namespace 声明的一个重要问题：在一个命名空间中，可以引用相同 namespace 声明中的其他标识符，而不需要使用圆点符号。例如，i 字段使用来自于 myNames 命名空间中的 integer 类型，但是没有 myNames.前缀。

²¹ 过程声明(第5章的主题)在 namespace 声明段中也是合法的。

例子中没有明显表示出来的是，每当打开命名空间的时候，`namespace` 声明就会创建一个干净的符号表。在 `namespace` 声明中，HLA 唯一认识的外部符号是预定义的类型标识符(例如 `int32`、`uns32` 和 `char`)。当 HLA 处理 `namespace` 声明时，它不认识程序员在 `namespace` 之外声明过的任何符号。当在 `namespace` 内部声明其他符号的时候，如果想要使用 `namespace` 之外的名称，就会产生问题。例如，假设类型 `integer` 已经在 `myNames` 之外被定义如下：

```

type
    integer: int32;

namespace myNames;

    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

如果想要编译这一代码，HLA 将会报错，说明符号 `integer` 没有定义。很显然，`integer` 在这个程序中已经定义了，但是 HLA 在创建一个命名空间时会将所有外部的符号隐藏，以便于在命名空间中重用(以及重新定义)这些符号。当然，如果实际上想要在 `myNames` 命名空间中使用在该命名空间外部定义的名称，这并不会起多大作用。HLA 提供了一个解决该问题的方法：即 `@global:` 操作符。如果在一个 `namespace` 声明段中，将 `@global:` 作为一个名称的前缀，则 HLA 将使用该名称的全局定义，而不是局部定义(即使存在一个局部定义)。下面的代码纠正了上面例子中的问题：

```

type
    integer: int32;

namespace myNames;

    static
        i:@global:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

有了 `@global:` 作为前缀，`i` 变量的类型将是 `int32`，无论在 `myNames` 命名空间中是否有不同的 `integer` 定义出现。

`namespace` 声明不能嵌套。从逻辑上讲，也不需要这样做，因此就从 HLA 语言中删除了这一功能。

在同一个程序中，可以有多个使用相同命名空间标识符的 `namespace` 定义，例如：


```

namespace ns;

    << Declaration group #1 >>

end ns;

namespace ns;

    << Declaration group #2 >>

end ns;

```

对于给定的标识符,当 HLA 遇到第二个 namespace 声明时,它就会将第二组中的声明附加在它为第一组所创建的符号列表后。因此,在处理了两个 namespace 声明后,ns 命名空间将包含两个 namespace 块中定义的所有符号的集合。

可能命名空间最普遍的应用是在库模块中。如果创建了一组库例程,可以在许多项目中使用,或者可以分给其他人使用,那么就必须要为函数和其他对象所选择的名称。如果使用普通的名称,如 get 和 put,则用户就会抱怨这些名称与他们所使用的名称冲突。一个简单的解决方法是将所有代码都放到一个 namespace 块中。那么就只需要考虑 namespace 标识符本身的名称,这是唯一可能与其他用户的标识符产生冲突的名称。当然这有可能会发生,但是,比起不使用命名空间,并且库模块向全局命名空间中引入了几十个,甚至几百个新的名称,发生的几率要小得多²²。HLA 标准库提供了许多使用命名空间的好例子。HLA 标准库定义了许多命名空间,例如 stdout、stdin、str、cs 以及 chars。可以使用类似于 stdout.put、stdin.get、cs.intersection、str.eq 以及 chars.toUpper 的名称来引用这些命名空间中的函数。HLA 标准库中命名空间的使用避免了与程序中相似的名称产生冲突。

4.35 汇编语言中的动态数组

本章所介绍的数组的一个问题是它们的大小是静态的。也就是说,编写程序时,所有例子中的元素数量都是提前定好的,而不是在程序运行时(也就是动态地)选择的。有时候在编写程序时,很难知道究竟需要多大的数组;只有在程序运行的时候才能确定数组的大小。本节将要描述如何为数组动态地分配存储空间,这样就可以在运行时设定数组的大小。

在运行的时候,为一维数组分配存储空间,并访问这个数组的元素,是非常简单的事情。需要做的仅仅是调用 HLA 标准库中的 mem.alloc 例程,以字节的形式指定数组的大小。mem.alloc 将在 EAX 寄存器中返回一个指向新数组基址的指针。一般情况下,可以将这个地址保存到一个指针变量中,并在将来所有的数组访问中作为数组的基址。

为了访问一维动态数组中的成员,一般可以将基址装载到一个寄存器中,并在第二个寄存器中计算变址。然后就可以使用基址变址的寻址方式来访问数组的成员。比起访问静态数组的成员来说,这并不会麻烦很多。下面的代码段展示了如何分配和访问一维动态数组中的元素:

²² 全局命名空间是程序中的全局部分。

```

static
    ArySize:      uns32;
    BaseAdrs:     pointer to uns32;
    .
    .
    .

    stdout.put( "How many elements do you want in your array?" );
    stdin.getu32();
    mov( eax, ArySize );      // Save away the upper bounds on this array.
    shl( 2, eax );           // Multiply eax by 4 to compute the number of bytes.
    mem.alloc( eax );        // Allocate storage for the array.
    mov( eax, BaseAdrs );    // Save away the base address of the new array.
    .
    .
    .

    // Zero out each element of the array:

    mov( BaseAdrs, ebx );
    mov( 0, eax );
    for( mov(0, esi); esi < ArySize; inc( esi ) ) do

        mov( eax, [ebx + esi*4] );
    endfor;

```

为多维数组动态分配存储空间是相当简单的。多维数组中含有的成员数是所有维数的乘积，例如， 4×5 的数组含有 20 个成员。因此，如果从用户那里得到每一个维数的上限，那么所需要做的仅仅是计算所有这些上限值的乘积，并将最后的结果与单个成员的大小相乘。这样就计算出了数组的所有字节数，也就是 `mem.alloc` 所需要的数据。

访问多维数组的元素比较麻烦。问题是必须知道维数的信息(也就是每一维的上限)，因为在计算以行为主(或者以列为主)的数组索引时需要这些数据²³。传统的方法是把这些数据存储到一个静态数组中(一般来说，在编译时会知道维数，因此就有可能静态地为这些数组维数上限组成的数组分配存储空间)。这个动态数组上限组成的数组就是内情向量(dope vector)。下面的代码段给出了如何使用简单的内情向量为二维动态数组分配存储空间。

```

var
    ArrayPtr: pointer to uns32;
    ArrayDims: uns32[2];      // The dope vector
    .
    .
    .

    // Get the array bounds from the user:

    stdout.put( "Enter the bounds for dimension #1:" );
    stdin.get( ArrayDims[0] );

    stdout.put( "Enter the bounds for dimension #2:" );
    stdin.get( ArrayDims[1*4] );

```

23 一般不需要最左边的维数上限来计算数组的索引。然而，如果想要使用 `bound` 指令(或者其他的技术)检查索引上限，则也需要在运行时知道这个值。


```

// Allocate storage for the array:

mov( ArrayDims[0], eax );
intmul( ArrayDims[1*4], eax );
shl( 2, eax ); // Multiply by 4 because each element is 4 bytes.
mem.alloc( eax ); // Allocate storage for the array and
mov( eax, ArrayPtr ); // save away the pointer to the array.

// Initialize the array:

mov( 0, edx );
mov( ArrayPtr, edi );
for( mov(0, ebx); ebx < ArrayDims[0]; inc(ebx)) do

    for( mov(0, ecx); ecx < ArrayDims[1*4]; inc(ecx)) do

        // Compute the index into the array
        // as esi := (ebx * ArrayDims[1*4] + ecx) * 4
        // (Note that the final multiplication by 4 is
        // handled by the scaled indexed addressing mode below.)

        mov( ebx, esi );
        intmul( ArrayDims[1*4], esi );
        add( ecx, esi );

        // Initialize the current array element with edx.

        mov( edx, [edi+esi*4] );
        inc( edx );

    endfor;

endfor;

endfor;

```

4.36 更多信息

在 <http://webster.cs.ucr.edu/> 或 <http://www.artofasm.com/> 上可以找到本书的电子版，在那里可以得到更多关于数据类型的信息。HLA 标准库文档描述了一个 HLA 数组包，该包提供了对动态(和静态)分配的数组、数组元素索引的确定和其他许多数组选项的支持。可以参考 HLA stdlib 文档来了解关于这个数组包的更多细节。关于数据结构在存储器中的表示的更多信息，可以阅读笔者写的 *Write Great Code, Volume 1* (No Starch Press, 2004) 一书。要想对数据类型深入研究，可以阅读关于数据结构和算法方面的书籍。

第 5 章

过程和单元



在基于过程的程序设计语言中，构成代码的基本单元是过程。过程是一组指令，或用于计算某个数值，或完成某种操作(如打印或读取一个字符值)。本章将讨论 HLA 的过程是如何实现的。首先从 HLA 的高级语法开始，这些语法包括过程的声明与调用。同时还会描述过程在机器级低层的实现。到现在为止，您应该已经比较熟悉汇编语言编程了，因此，我们将采用“纯粹”的汇编语言，而不再依赖于 HLA 的高级语法。

5.1 过程

大多数基于过程的程序设计语言使用调用/返回机制实现过程。调用/返回机制就是在代码执行过程中调用某个过程，该过程完成它所定义的功能后返回调用者继续执行。调用和返回指令提供了 80x86 的过程调用机制(procedure invocation mechanism)。调用代码使用 `call` 指令调用一个过程；过程使用 `ret` 指令返回调用代码。例如，下面这条 80x86 指令是对 HLA 标准库例程 `stdout.newln` 的调用¹：

```
call stdout.newln;
```

过程 `stdout.newln` 在控制台设备上输出一个换行序列，然后将控制权立即返回给指令 `call stdout.newln;` 后面的指令。

HLA 的标准库不能提供所有您需要的例程。大多数情况下，必须编写自己的过程。在这个过程中将使用 HLA 的过程声明功能。基本的 HLA 过程声明格式为：

¹ 通常应使用 `newln()`；语法调用 `newln`，但是使用 `call` 指令也能够很好地工作。

```

procedure ProcName;
    << Local declarations >>
begin ProcName;
    << Procedure statements >>
end ProcName;

```

过程的声明出现在程序的声明段中。也就是说，在 `static`、`const`、`type` 或其他声明段中都可以对过程进行声明。在前面的语法示例中，*ProcName* 为所要定义的过程名称，它可以是任何有效(且唯一)的 HLA 标识符。在一个过程里面，无论跟在保留字 `procedure` 后面的是什么标识符，它同样要跟在保留字 `begin` 和 `end` 之后。您可能已经注意到，HLA 中过程的声明看起来与程序的声明非常类似。事实上，就目前来说它们之间唯一的差别就在于用保留字 `procedure` 来代替保留字 `program`。

下面是一个 HLA 过程声明的具体示例。这个过程将 0 存储在进入过程时 `EBX` 所指向的 256 个双字中。

```

procedure zeroBytes;
begin zeroBytes;

    mov( 0, eax );
    mov( 256, ecx );
    repeat

        mov( eax, [ebx] );
        add( 4, ebx );
        dec( ecx );

    until( @z ); // That is, until ecx=0.

end zeroBytes;

```

使用 80x86 的 `call` 指令调用这个过程。在程序执行过程中，当代码进入 `end zeroBytes;` 语句时，过程返回调用者并从 `call` 指令后的第一条指令开始执行。程序清单 5-1 是对上面的 `zeroBytes` 例程进行调用的示例程序。

程序清单 5-1 简单过程的示例

```

program zeroBytesDemo;
#include( "stdlib.hhf" )

procedure zeroBytes;
begin zeroBytes;

    mov( 0, eax );
    mov( 256, ecx );
    repeat

        mov( eax, [ebx] );    // Zero out current dword.
        add( 4, ebx );       // Point ebx at next dword.
        dec( ecx );          // Count off 256 dwords.
    until( @z );
end zeroBytes;

```

```

        until( ecx = 0 );           // Repeat for 256 dwords.

    end zeroBytes;

static
    dwArray:dword[256];

begin zeroBytesDemo;

    lea( ebx, dwArray );
    call zeroBytes;

end zeroBytesDemo;

```

当对 HLA 标准库中的过程进行调用时，并不一定要使用 `call` 指令。自己编写的过程与 HLA 标准库中的过程没有任何差别。虽然 80x86 正式的调用机制是要使用 `call` 指令调用过程，但 HLA 提供了对语法的高级扩展，可以通过一种简单的方法对过程进行调用。这种方法是指定一个过程的名称，并且后面跟着一对内空的括号²。例如，下面这两条语句都是对 HLA 标准库中的过程 `stdout.newln` 的调用。

```

call stdout.newln;
stdout.newln();

```

同样，下面两条语句也都可以对程序清单 5-1 中的过程 `zeroBytes` 进行调用。

```

call zeroBytes;
zeroBytes();

```

选择哪种调用机制完全取决于您自己，但是大多数人一般认为高级语法具有很好的可读性。

5.2 机器状态的保存

我们看一下程序清单 5-2。这段代码要实现的功能是打印 20 行，每行 40 个空格和 1 个星号，但其中有一个错误导致无限循环。主程序使用 `repeat..until` 循环调用 20 次 `PrintSpaces`。`PrintSpaces` 使用寄存器 `ECX` 计数它打印的 40 个空格。当 `PrintSpaces` 返回的时候，`ECX` 的内容为 0。主程序打印 1 个星号后换行，对 `ECX` 执行减 1 操作，因为 `ECX` 此时不为 0（这时它的值总是 `$FFFF_FFFF`），故主程序将重复循环。

问题在于子例程 `PrintSpaces` 没有保存寄存器 `ECX`。寄存器的保存是指在进入子例程之前保存它的值，并在子例程执行后且返回调用者之前将它的值恢复原状。如果程序清单 5-2 中的子例程 `PrintSpaces` 对寄存器 `ECX` 进行了保存，程序将会正确地执行所要实现的功能。

程序清单 5-2 因为不小心造成的无限循环程序

```

program nonWorkingProgram;
#include( "stdlib.hhf" );

```

² 这里假设过程不带任何参数。


```
procedure PrintSpaces;
begin PrintSpaces;

    mov( 40, ecx );
    repeat

        mov( ' ', al );
        stdout.putc( al ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.

    until( ecx = 0 );

end PrintSpaces;

begin nonWorkingProgram;

    mov( 20, ecx );
    repeat

        PrintSpaces();
        stdout.put( '*', nl );
        dec( ecx );

    until( ecx = 0 );

end nonWorkingProgram;
```

当寄存器还有其他用途时,可用 80x86 的 `push` 和 `pop` 指令保存寄存器的值。考虑下面这段 `PrintSpaces` 代码:

```
procedure PrintSpaces;
begin PrintSpaces;

    push( eax );
    push( ecx );
    mov( 40, ecx );
    repeat

        mov( ' ', al );
        stdout.putc( al ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.

    until( ecx = 0 );
    pop( ecx );
    pop( eax );

end PrintSpaces;
```

可以看到, `PrintSpaces` 保存并恢复了寄存器 `EAX` 和 `ECX` 的状态(因为在过程中修改了这些寄存器的值)。并且,寄存器的出栈顺序与入栈顺序相反,这是由于栈的后进先出操作强制要求这样的顺序。

调用程序(包含 `call` 指令的代码)或被调用程序(子例程)都可以对寄存器的值进行保存。在上面的示例中, 由被调用程序保存寄存器。程序清单 5-3 显示的代码使用调用程序保存寄存器。

程序清单 5-3 调用程序保存寄存器的示例

```

program callerPreservation;
#include( "stdlib.hhf" );

  procedure PrintSpaces;
  begin PrintSpaces;

    mov( 40, ecx );
    repeat

      mov( ' ', al );
      stdout.putc( al );  // Print 1 of 40 spaces.
      dec( ecx );         // Count off 40 spaces.

    until( ecx = 0 );

  end PrintSpaces;

begin callerPreservation;

  mov( 20, ecx );
  repeat

    push( eax );
    push( ecx );
    PrintSpaces();
    pop( ecx );
    pop( eax );
    stdout.put( '*', nl );
    dec( ecx );

  until( ecx = 0 );

end callerPreservation;

```

使用被调用程序保存寄存器具有两个优点: 空间开销小和可维护性强。如果用被调用程序(过程)保存被修改的寄存器, 那么相应的 `push` 和 `pop` 指令只在被调用的过程中书写一次; 如果用调用程序保存寄存器的值, 程序中的每个 `call` 指令前后都会出现 `push` 和 `pop` 指令, 导致指令的重复。这些重复的指令不仅增加了程序的空间开销, 还使得它们更难维护, 因为要清楚地记住每个过程调用所要保存的寄存器是很难的事情。

另一方面, 一个子例程可能会保存它所修改的所有寄存器, 但实际上有一些寄存器是没必要保存的。在上面的示例中, 就没有必要保存寄存器 `EAX`。虽然 `PrintSpaces` 改变了寄存器 `AL` 的值, 但并没有影响程序的正常操作。当调用程序保存寄存器时, 它没必要去保存那些根本就不关心的寄存器(参见程序清单 5-4)。

程序清单 5-4 调用程序不需要保存所有寄存器的示例

```

program callerPreservation2;
#include( "stdlib.hhf" );

procedure PrintSpaces;
begin PrintSpaces;

    mov( 40, ecx );
    repeat
        mov( ' ', al );
        stdout.putc( al ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.
    until( ecx = 0 );
end PrintSpaces;

begin callerPreservation2;

    mov( 10, ecx );
    repeat
        push( ecx );
        PrintSpaces();
        pop( ecx );
        stdout.put( '*', nl );
        dec( ecx );
    until( ecx = 0 );

    mov( 5, ebx );
    while( ebx > 0 ) do

        PrintSpaces();

        stdout.put( ebx, nl );
        dec( ebx );

    endwhile;

    mov( 110, ecx );
    for( mov( 0, eax ); eax < 7; inc( eax ) ) do

        PrintSpaces();

        stdout.put( eax, " ", ecx, nl );
        dec( ecx );

    endfor;

end callerPreservation2;

```

程序清单 5-4 中的示例给出了 3 种不同的情况。第一个循环(repeat..until)只对寄存器 ECX 进

行了保存。对寄存器 AL 的修改不会影响循环的正常操作。紧接第一个循环，代码中的一个 `while` 循环再次对 `PrintSpaces` 进行调用。然而，这段代码并没有对寄存器 EAX 或 ECX 进行保存，因为它并不关心 `PrintSpaces` 是否对这两个寄存器的内容进行了修改。

调用程序对寄存器进行保存带来的一个比较大的问题是不利于程序的修改。当对调用程序或过程进行修改时，可能会使用一些附加的寄存器。而这些改动会改变您所要保存的寄存器。更糟糕的是，如果改动是在子例程内部进行，您将必须定位到每一个对子例程的调用，去确定子例程是否改变了调用程序使用的寄存器。

保存寄存器并不是全部，还可以对运行环境进行保存。用同样的压栈、出栈方法对子例程所改变的变量或其他值进行保存。80x86 允许对存储单元进行压栈和出栈的操作，因此很容易实现对这些数值的保存。

5.3 过程的提前返回

使用 HLA 的 `exit` 和 `exitif` 语句可以让程序从一个过程提前返回而不用执行到相应的 `end` 语句。这两条语句实现的功能类似于 `break` 和 `breakif` 语句在循环中所起的作用，差别在于前者把控制权交给过程体后的语句而不是跳出当前循环。这些语句在某些情况下是很有用处的。

这两条语句的语法如下所示：

```
exit procedurename;
exitif( boolean_expression ) procedurename;
```

其中，*procedurename* 操作数是要退出的过程的名称。如果指定的是主程序的名称，则 `exit` 和 `exitif` 语句将会终止程序的执行(无论当时是执行在过程体内还是主程序体内)。

`exit` 语句会立即无条件地将控制权移出所指定的过程或程序。而条件语句 `exitif` 首先检查布尔表达式的值，结果为 `true` 则退出。条件语句 `exitif` 在语义上等价于：

```
if( boolean_expression ) then
    exit procedurename;
endif;
```

虽然 `exit` 和 `exitif` 这两条语句在很多情况下使用起来非常方便，但不能未经仔细考虑就使用它们。如果一个简单的 `if` 语句可以跳过过程中的剩余代码，那么应该选择使用 `if` 语句。包含很多 `exit` 和 `exitif` 语句的过程将比那些没有使用该语句的过程更难阅读、难以理解和不易维护(毕竟，`exit` 和 `exitif` 语句与 `goto` 语句相似，您应该曾经听说过 `goto` 语句所产生的问题)。当想从连续嵌套控制结构的某个过程中返回时，使用 `exit` 和 `exitif` 语句非常方便。而在剩余代码的周围使用 `if..endif` 是不可能的。

5.4 局部变量

HLA 中的过程像很多高级语言的过程和函数一样，可以声明局部变量。局部变量通常只在过程内部是可访问的，而对于调用过程的代码是不可访问的。局部变量的声明与主程序变量的声明

采用同样的方式,当然,局部变量的声明是在过程的声明段而不是在主程序的声明段中。事实上,只要是在主程序声明段内合法的内容,在过程的声明段中都可以同样进行声明,包括常量、类型、甚至其他过程³。在本节中,我们主要讨论局部变量。

局部变量有两个区别于主程序中变量(全局变量)的重要属性,它们是词法作用域(lexical scope)和生存期(lifetime)。词法作用域,简称作用域(scope),确定一个标识符在程序中的什么地方是可用的。生存期确定一个变量在什么时段与存储器相关联,具有存储数据的能力。正因为这两个概念是区分局部变量和全局变量的重要属性,因此要花些时间来对它们进行讨论。

讨论局部变量的作用域和生存期最好从讨论全局变量(在主程序中声明的变量)的作用域和生存期开始。到现在为止,我们对于声明变量需要遵循的唯一原则是“程序中用到的变量必须先声明”。HLA 的声明段与程序语句的位置关系自动施加了另一条主要规则:在使用任何变量之前,必须先声明所有的变量。由于过程的引入,现在有可能违反这条规则,因为:①过程可以访问全局变量;②过程的声明可以出现在声明段的任何位置,甚至可以在一些变量的声明之前。程序清单 5-5 的程序演示了这种源代码的组织。

程序清单 5-5 全局作用域的示例

```

program demoGlobalScope;
#include( "stdlib.hhf" );

static
    AccessibleInProc: char;

    procedure aProc;
    begin aProc;

        mov( 'a', AccessibleInProc );

    end aProc;

static
    InaccessibleInProc: char;

begin demoGlobalScope;

    mov( 'b', InaccessibleInProc );
    aProc();
    stdout.put
    (
        "AccessibleInProc    = '", AccessibleInProc, "' " nl
        "InaccessibleInProc  = '", InaccessibleInProc, "' " nl
    );

end demoGlobalScope;

```

³ 严格地讲,这样说并不准确。因为不可以在过程内声明外部对象。外部对象是 5.24 节要讲述的内容。

这个示例说明只要主程序中全局变量的声明在过程之前，这个过程就可以访问全局变量。在示例中，过程 `aProc` 不能对变量 `InaccessibleInProc` 进行访问，因为变量的声明出现在过程声明之后。然而，`aProc` 可以访问变量 `AccessibleInProc`，因为变量声明出现在 `aProc` 过程之前。

过程可以用简单的名称引用访问任何静态对象、存储对象或只读对象，像主程序对这类变量的访问一样。虽然过程可以访问全局变量对象，但在语法上会有不同，具备必要的知识背景之后您才能更好地理解这些附加的语法(想获得更多信息，请参见 HLA 参考手册)。

对全局对象进行访问既方便又容易。但是，在高级语言的编程中会发现，全局对象的访问会使得程序的可读性、可理解性和可维护性都有所降低。像大多数介绍编程的书籍一样，本书不提倡在过程中访问全局变量。对于某个给定问题，在过程中对全局变量进行访问有时可能是最好的解决办法。然而，这种(合法的)访问通常只出现在高级程序中，包括多线程执行或其他复杂的系统。因为目前还不需要编写这样的代码，也就不必要在过程中访问全局变量。所以，如果在过程中要访问全局变量，应该经过仔细考虑⁴。

在过程中声明局部变量非常简单，使用与主程序同样的声明段就可以：`static`、`readonly`、`storage` 和 `var`。声明段和所声明变量的访问规则及语法同样适用于过程。程序清单 5-6 中的示例代码演示了局部变量的声明方法。

程序清单 5-6 过程中的局部变量示例

```
program demoLocalVars;
#include( "stdlib.hhf" );

// Simple procedure that displays 0..9 using
// a local variable as a loop control variable.

procedure CntTo10;
var
    i:int32;

begin CntTo10;
    for( mov( 0, i ); i < 10; inc( i ) ) do
        stdout.put( "i=", i, nl );
    endfor;
end CntTo10;

begin demoLocalVars;
    CntTo10();
end demoLocalVars;
```

在一个过程中所声明的局部变量，一般只在本过程中是可访问的⁵。因此，在程序清单 5-6 中，

⁴ 注意，关于访问全局变量的争论不牵涉其他全局符号。在程序中访问全局常量、类型、过程和其他对象都是完全合理的。

⁵ 严格地讲，这样说并不准确。但是，访问非局部变量对象的内容不在本书的讨论范围内。可以参阅 HLA 文档了解更多信息。

过程 CntTo10 中的变量 i 对于主程序是不可访问的。

对于局部变量，HLA 放松了程序中变量的标识符必须唯一这条规则。在 HLA 程序中，只要求给定作用域内的标识符必须唯一。因此，所有全局名称必须唯一。类似地，过程内的局部变量也必须有唯一的名称，但这只是相对于该过程中其他的局部符号而言。局部名称可以和全局名称相同。当这种情况发生的时候，HLA 创建两个不同的变量。在过程的作用域内部，所有对这个公共名称的引用会访问局部变量；而在过程以外，对这个名称的引用则访问全局标识符。虽然这种代码的质量值得质疑，但这样做却完全合法。例如，对于全局标识符 MyVar，它同样可以作为两个或者多个过程的局部名称使用。每个过程都有它们自己独立于主程序中 MyVar 的局部变量。程序清单 5-7 演示了这种特征。

程序清单 5-7 局部变量不必具有全局唯一的名称

```
program demoLocalVars2;
#include( "stdlib.hhf" );

static
    i: uns32 := 10;
    j: uns32 := 20;

    // The following procedure declares i and j
    // as local variables, so it does not have access
    // to the global variables by the same name.

    procedure First;
    var
        i: int32;
        j: uns32;

    begin First;

        mov( 10, j );
        for( mov( 0, i ); i < 10; inc( i ) ) do

            stdout.put( "i=", i, " j=", j, nl );
            dec( j );

        endfor;

    end First;

    // This procedure declares only an i variable.
    // It cannot access the value of the global i
    // variable but it can access the value of the
    // global j object because it does not provide
    // a local variant of j.
    .
    procedure Second;
    var
        i: uns32;

    begin Second;
```

```

mov( 10, j );
for( mov( 0, i ); i < 10; inc( i ) ) do

    stdout.put( "i=", i, " j=", j, nl );
    dec( j );

endfor;

end Second;

begin demoLocalVars2;

    First();
    Second();

    // Because the calls to First and Second have not
    // modified variable i, the following statement
    // should print "i=10". However, because the Second
    // procedure manipulated global variable j, this
    // code will print "j=0" rather than "j=20".

    stdout.put( "i=", i, " j=", j, nl );

end demoLocalVars2;

```

在一个过程中，重用全局名称的优缺点是并存的。一方面，这种重用会带来潜在的弊端：可能引起混淆。例如，当使用 `ProfitsThisYear` 作为一个全局符号并在过程中重用这个名称时，程序阅读者可能会认为过程不是对局部符号而是对全局符号进行引用。另一方面，像 `i`、`j`、`k` 这样的简单名称，几乎没有实际意义（几乎所有的编程者都习惯使用它们作为循环控制变量或作为其他的局部变量），因此把它们作为局部对象进行重用是很好的编程理念。从软件工程的角度考虑，应尽量使程序中具有确定意义的名称（如 `ProfitsThisYear`）保持唯一。意义模糊的一般性名称（如 `index`、`counter`、`i`、`j`、`k`）可以作为全局变量被重用。

对于 HLA 程序中标识符的作用域还有一点要说明，在相互独立的过程中所使用的变量也是相互独立的，即使它们具有相同的名称。例如，在程序清单 5-7 中的过程 `First` 和 `Second` 中共用了同名的局部变量 `i`。然而，`First` 中的 `i` 和 `Second` 中的 `i` 是完全不同的两个变量。

生存期是区别局部变量和全局变量的第二个主要特征。变量的生存期是指程序第一次为它分配存储空间开始到释放这些空间为止所跨越的时间。注意，生存期是动态属性（在运行时控制），而作用域是静态属性（在编译时控制）。事实上，一个变量可以具有多个生存期，因为程序可能重复对这个变量进行存储区分配与释放的操作。

全局变量的生存期相对简单，是从主程序首次执行开始到主程序终止所跨越的时间。同样地，所有的静态对象也都具有一个跨越程序执行的生存期（静态对象是 `static`、`readonly`、`storage` 段所声明的名称），这在过程中也同样成立。因此，局部静态对象与全局静态对象的生存期没有任何差别。但对 `var` 段声明的变量却有所不同，`var` 对象使用自动存储分配（automatic storage allocation）原则。所谓自动存储分配，是在进入过程后自动为局部变量分配存储空间。同样地，当过程返回调用者时，主程序自动释放这些存储空间。因此，这种自动对象的生存期是从过程执行第一条语句开始到返回调用者所跨越的时间。

关于自动变量要注意的重要一点是，这种自动变量在过程调用间不能保持它们的数值。一旦

过程返回调用程序，自动分配的存储空间被释放，同时变量的数值丢失。因此，在一个过程的入口点必须总是假设局部的 `var` 对象没有初始化，即使已经调用过这个过程并对变量进行了初始化。无论最后一次调用存储在变量中的值是什么，都将在过程返回调用程序时丢失。要想在过程调用间保持变量的值，应该使用静态变量声明类型。

自动变量在过程调用间不能保持数据的值，但这并不能阻止对它的使用，因为它还具有很多静态变量所没有的优点。使用静态变量的最大弊端是，当引用它的过程没有运行时，仍然消耗存储空间；而对于自动变量，只有与它相关联的过程运行时才消耗存储空间。调用返回时，过程释放所有自动分配的存储空间给系统，以供其他过程重复使用。在本章后面还会讲述自动变量的其他优点。

5.5 其他局部和全局符号类型

从前面的讲述中已经知道，HLA 的过程允许在主程序的声明段中声明诸如常量、数值、类型和其他一切符合语法规定的可声明内容。关于作用域的规则同样适用于这些标识符。因此，在局部声明中可对常量名、过程名、类型名等进行重用。

对全局常量、数值、类型的引用不会出现像对全局变量引用时所产生的那种软件工程问题。对全局变量进行引用所产生的问题是：过程可以通过一种隐式方式改变全局变量的值。这使得程序的可读性、可理解性和可维护性都会降低，因为只观察过程的调用不能确定过程是否对存储器进行了修改。常量、数值、类型等其他非变量对象，由于在运行时不会被修改，因此不存在这样的问题。所以，不必像竭力避免使用全局变量那样避免使用非变量对象。

前面已经讲述过，可以对全局常量、类型等进行访问。需要指出的是，如果程序对这些全局对象的引用仅在某个过程中出现，则可以将这些对象局部声明于这个过程中。这样做将会使程序更易读，因为程序的阅读者不用对整个程序进行搜索来找到这些符号的定义。

5.6 参数

虽然有些过程是完全独立的，但大部分过程还是需要数据的输入，同时返回一些数据给调用程序。参数是过程传入和传出的数值。在纯粹的汇编语言里，参数传递是一件繁琐的事情。幸运的是，HLA 为过程的声明和涉及参数的过程调用提供了一种类似于高级语言的语法。本节将讲述 HLA 关于参数的高级语法。本章后续部分会涉及在纯汇编程序中传递参数的低级机制。

要对参数进行讨论，首先要考虑的是如何将参数传递给过程。如果您熟悉 Pascal 或 C/C++，或许已经知道了传递参数的两种方式：值传递和引用传递。HLA 无疑是支持这两种参数传递机制的。不仅如此，HLA 还支持值/结果传递、结果传递、名称传递和延迟求值传递。当然，因为 HLA 是一门汇编语言，因此在 HLA 中可以使用任何您所能想到的传递参数的方法(至少，这些方法对 CPU 来说是完全可以实现的)。但是，HLA 为值传递、引用传递、值/结果传递、结果传递、名称传递和延迟求值传递提供了专用的高级语法。

值/结果传递、结果传递、名称传递和延迟求值传递是比较高级的方法，本书不对这些参数传递机制进行讨论。如果有兴趣进一步学习这些参数传递方法，可参阅 HLA 参考手册，或者阅读

本书的电子版本，网址为 <http://webster.cs.ucr.edu/>或 <http://www.artofasm.com/>。

所要面临的另一个与处理参数相关的问题是将它们传递到何处。传递参数有很多不同的位置，本节将把过程参数传递到栈上。HLA 已经将它们抽象出来，因此不必关心细节。然而，要记住的是过程调用和过程参数都使用栈。因此，在一个过程调用前，无论压入栈的内容是什么，当执行进入过程后，先前的内容已不在栈的顶部了。

5.6.1 值传递

参数的值传递方式就是调用程序将数值直接传递给过程。通过值传递的参数只是输入的参数。也就是说，可以将这些数值参数传递给过程，但过程不能通过它们返回数值。例如有这样一个 HLA 的过程调用：

```
CallProc(I);
```

如果 I 是通过值传递的参数，那么不管在过程 `CallProc` 内部对参数作了怎样的操作，都不会改变 I 的值。

值传递是将数据的一个副本传递给过程，因此应只对字节、字和双字这种小数据对象采用值传递方式。采用值传递方式传递大规模数组和记录的效率很低(因为每个对象都要产生一个副本，而后再传递给过程)。

类似于 Pascal 和 C/C++，如果不另外特殊指定，HLA 采用值传递方式传递参数。下面是一个通过值传递参数的简单函数：

```
procedure PrintNSpaces( N:uns32 );
begin PrintNSpaces;

    push( ecx );
    mov( N,ecx );
    repeat

        stdout.put( ' '); // Print 1 of N spaces.
        dec( ecx );       // Count off N spaces.

    until( ecx = 0 );
    pop( ecx );

end PrintNSpaces;
```

过程 `PrintNSpaces` 中的 N 是一个形参(formal parameter)。过程体中凡是有 N 出现的地方，程序都会引用调用程序通过 N 所传递的值。

过程 `PrintNSpaces` 的调用序列可以是下面的任意一种：

```
PrintNSpaces( constant );
PrintNSpaces( reg32 );
PrintNSpaces( uns32_variable );
```

下面是一些调用 `PrintNSpaces` 的具体示例：

```
PrintNSpaces( 40 );
PrintNSpaces( eax );
PrintNSpaces( SpacesToPrint );
```

在调用 `PrintNSpaces` 的语句中出现的参数是实参(actual parameter)。上面示例中出现的 `40`、`eax` 和 `SpacesToPrint` 都是实参。

注意,通过值传递的参数与 `var` 段声明的局部变量行为相似,唯一不同是过程的调用者将这些局部变量的控制权传递给过程之前,要先对它们进行初始化。

像大多数高级语言一样,HLA 使用位置参数表示法。因此,当要传递的参数多于一个时,HLA 将根据它们在参数列表中的位置对形参和实参进行关联。下面的过程 `PrintNChars` 是具有两个参数的简单过程示例:

```
procedure PrintNChars( N:uns32; c:char );
begin PrintNChars;

    push( ecx );
    mov( N, ecx );
    repeat

        stdout.put( c ); // Print 1 of N characters.
        dec( ecx );      // Count off N characters.

    until( ecx = 0 );
    pop( ecx );

end PrintNChars;
```

下面是调用过程 `PrintNChars` 打印 20 个星号的语句:

```
PrintNChars( 20, '*' );
```

注意,在过程的声明中,HLA 使用分号分隔形参,而在过程调用的时候则使用逗号分隔实参(使用 Pascal 语言的编程者对这种表示法应该很熟悉)。还可以看到,每一个 HLA 形参的声明都采用下面的形式:

```
parameter_identifier : type_identifier
```

特别要指出,参数的类型必须是一个标识符。下面的参数声明都是非法的,因为数据类型不是一个标识符。

```
PtrVar: pointer to uns32
ArrayVar: uns32[10]
recordVar: record i:int32; u:uns32; endrecord
DynArray: array.dArray( uns32,2 )
```

然而,不要因此而认为不能够传递指针、数组、记录或动态数组变量类型的参数。技巧是在 `type` 段为这些类型声明一个数据类型。然后,就可以使用一个标识符作为参数声明的类型。下面的代码段演示了如何使用前面 4 个数据类型来声明参数:

```

type
  uPtr:      pointer to uns32;
  uArray10:  uns32[10];
  recType:   record i:int32; u:uns32; endrecord
  dtype:    array.dArray( uns32, 2 );

procedure FancyParms
(
  PtrVar:    uPtr;
  ArrayVar:  uArray10;
  recordVar: recType;
  DynArray:  dtype
);
begin FancyParms;
.
.
.
end FancyParms;

```

默认情况下, HLA 假定程序使用值传递方式传递参数。HLA 也允许显式声明一个通过值传递的参数, 方法是在形参前使用关键字 `val` 进行声明。下面是将过程 `PrintNSpaces` 的参数 `N` 显式声明为通过值传递参数的另一版本:

```

procedure PrintNSpaces( val N:uns32 );
begin PrintNSpaces;

  push( ecx );
  mov( N, ecx );
  repeat
    stdout.put( ' ' );    // Print 1 of N spaces.
    dec( ecx );           // Count off N spaces.

  until( ecx = 0 );
  pop( ecx );

end PrintNSpaces;

```

如果同一个过程声明中有多个参数, 并且这些参数使用不同的传递机制, 那么显式声明一个参数通过值传递是一种良好的编程习惯。

当通过值传递方式传递参数, 并且使用 HLA 的高级语言语法调用过程时, HLA 会自动生成对实参值进行复制的代码, 并将这个副本复制到局部存储器中供另一个参数使用(形参)。对于小规模的对象, 用值方式传递参数或许是最高效的。然而, 对于大规模对象, HLA 必须生成将实参的每一个字节都复制到形参中的代码。对于大规模数组或者记录, 这是一个开销很大的操作⁶。除非有特殊的语义关系使得您必须采用值传递方式传递大规模数组或记录, 否则应该使用引用传递或其他参数传递机制传递数组和记录。

6 C/C++编程者注意: HLA 并不自动通过引用传递数组。如果指定形参为数组类型, 那么当调用相关过程的时候, HLA 将会发出为数组中每一个元素的每一个字节产生副本的代码。

向过程传递参数的时候,HLA 会检验每个实参的类型,并将它们和对应形参的类型进行比较。如果不相符合,HLA 会对实参和形参分别进行检验,看其中之一是否为字节、字或双字对象,而另一个为单字节、双字节或四字节对象。如果实参没有满足上面的条件之一,HLA 将报出类型不匹配错误。若由于某些原因而需要传递一个与过程调用所需类型不一致的参数,可以使用 HLA 的类型强制操作符覆盖实参的类型。

5.6.2 引用传递

通过引用传递参数时,所传递的必须是变量的地址而不是数值。换句话说,必须传递一个指向数据的指针,过程通过指针来访问数据。要改变实参的值或传送大规模数据结构的时候,使用引用方式传递参数是很有效的方法。

必须在形参声明前使用关键字 `var` 来声明参数是通过引用传递的。下面的代码段演示了这样的例子:

```
procedure UsePassByReference( var PBRvar: int32 );
begin UsePassByReference;

end UsePassByReference;
```

使用与通过值传递参数的过程相同的语法来调用通过引用传递参数的过程。差别在于,通过引用传递的参数必须是一个存储单元而不能是常量或寄存器。而且,这个存储单元的数据类型必须与形参的类型精确匹配。下列是对上面过程合法的调用(假定 `i32` 是一个 `int32` 变量):

```
UsePassByReference( i32 );
UsePassByReference( (type int32 [ebx]) );
```

下列是对过程 `UsePassbyReference` 的非法调用(假设 `charVar` 是 `char` 类型):

```
UsePassByReference( 40 );           // Constants are illegal.
UsePassByReference( EAX );          // Bare registers are illegal.
UsePassByReference( charVar );      // Actual parameter type must match
                                     // the formal parameter type.
```

与高级语言 Pascal 和 C/C++ 不同的是,HLA 并不隐藏传递指针而非传递数值这个事实。在过程调用时,HLA 会自动计算变量的地址并将这个地址传递给过程。然而在过程内部,不能像对待数值参数那样来对待这些变量(大多数高级语言则允许这么做),而是将这类参数视为双字变量来看待,这些变量含有指向指定数据的指针。在访问了参数的数值后,要显式取消对相应指针的引用。程序清单 5-8 中的程序是这种情况的示例。

程序清单 5-8 访问通过引用传递的参数

```
program PassByRefDemo;
#include( "stdlib.hhf" );
```

```

var
  i:int32;
  j:int32;

procedure pbr( var a:int32; var b:int32);
const
  aa:text := "(type int32 [ebx])";
  bb:text := "(type int32 [ebx])";

begin pbr;

  push( eax );
  push( ebx );    // Need to use ebx to dereference a and b.

  // a = -1;

  mov( a, ebx );  // Get ptr to the "a" variable.
  mov( -1,aa );   // Store -1 into the "a" parameter.

  // b = -2;

  mov( b,ebx );   // Get ptr to the "b" variable.
  mov( -2,bb );   // Store -2 into the "b" parameter.

  // Print the sum of a+b.
  // Note that ebx currently contains a pointer to "b".

  mov( bb, eax );
  mov( a, ebx );  // Get ptr to "a" variable.
  add( aa, eax );
  stdout.put( "a+b=", (type int32 eax), nl );

end pbr;

begin PassByRefDemo;

  // Give i and j some initial values so
  // we can see that pass by reference will
  // overwrite these values.

  mov( 50,i );
  mov( 25,j );

  // Call pbr passing i and j by reference

  pbr( i,j );

  // Display the results returned by pbr.

  stdout.put
  (
    "i= ", i, nl,
    "j= ", j, nl
  );

end PassByRefDemo;

```


通过引用传递参数在某些不常见的情况下会产生奇特的结果。考虑程序清单5-8中的过程 `pbr`，如果用调用语句 `pbr(i,i)` 来替换主程序中的调用 `pbr(i,j)`，则程序会输出下面这样不直观的结果：

```
a+b=-4
i=  -2;
j=  25;
```

代码执行的结果为 `a+b=-4` 而不是直观感觉的 `a+b=-3`，原因在于调用语句 `pbr(i,i)`，把同一个实参同时传递给了 `a` 和 `b`。这样做的结果是使得 `a` 和 `b` 引用参数包含了指向同一个存储单元的指针，而这个存储单元保存着变量 `i` 的值。实际上，在这种情况下，`a` 和 `b` 是互为别名的。因此，当程序将 `-2` 保存到 `b` 所指向的单元时，实际上重写了之前由 `a` 指向时写入的 `-1`。当程序通过 `a` 和 `b` 所指向的位置来读取数据准备计算两数之和时，`a` 和 `b` 指向的是同一个数据：`-2`。两个 `-2` 相加得到的结果是 `-4`。无论什么时候，如果在程序中遇到别名的情况，都有可能产生这种非直观结果。将同一个变量作为两个不同的引用参数进行传递的情况并不常见。但如果一个过程对全局变量进行引用的同时，这个全局变量又被作为引用参数传递给这个过程，这种情况下同样会产生别名（这是另一个能说明为什么要避免在过程中引用全局变量的很好示例）。—

通过引用传递在效率上一般不如通过值传递高。在访问通过引用传递的参数后都必须释放相应指针，这与简单的值传递相比是缓慢的，因为这样做至少需要两条指令。然而，对于大规模数据结构的传递，使用引用方式比较快，因为在过程调用之前不必对大规模数据进行复制。当然，您或许要使用一个指针访问大规模数据结构（如数组）的每一个元素，因此通过引用传递大规模数组的效率很高。

5.7 函数和函数的结果

函数是向调用者返回结果的过程。在汇编语言里，过程与函数在语法上几乎没有差别，这也是 HLA 没有提供专门用于函数声明的语法的原因。虽然汇编语言的过程与函数在语法上没什么差别，但在语义上仍有很多不同。因此，在 HLA 中可以使用相同的方法声明它们，但对它们的使用是不同的。

过程是完成特定任务的机器指令序列，过程的执行结果就是任务行为的完成。与此相对，函数执行机器指令序列的目的是计算某些数值，并将计算结果返回调用程序。当然，函数也可以执行一些行为，过程也能够计算数值。但差别在于，函数的目的是返回计算结果，而对过程则没有这样的要求。

过程 `stdout.puti32` 是典型的过程示例。这个过程需要一个 `int32` 类型的参数。过程的执行目的是在标准输出设备上打印这个整型数值的十进制转换结果。注意，过程 `stdout.puti32` 没有返回任何对调用程序可用的数值。

函数 `cs.member` 是一个典型的函数示例。它需要两个参数：第一个参数是一个字符值，第二个参数是一个字符集值。如果该字符属于给定的字符集，则函数将在寄存器 `EAX` 中返回真(1)；如果字符不是给定字符集的成员，则返回假。

从逻辑上说，函数 `cs.member` 向调用程序返回了一个可用数值（在 `EAX` 中），而过程 `stdout.puti32` 则没有。从这两个示例可以看出函数和过程的主要区别。所以，一般来说，当要求在过程返回的

地方返回一个确定的数值时，过程就演变成为一个函数。函数的声明和使用不需要特殊的语法，因此使用与过程同样的方法编写函数。

5.7.1 返回函数结果

80x86 寄存器是最常见的函数结果返回位置。HLA 标准库中的 `cs.member` 例程是一个典型的示例，说明了函数把结果返回到 CPU 的一个寄存器中。它将返回的真值(1)或假值(0)保存在寄存器 EAX 中。按约定，编程人员习惯于将 8 位、16 位和 32 位(非实数)结果分别返回到寄存器 AL、AX 和 EAX 中⁷。大多数高级语言也是把这些类型的结果分别返回上述相应的寄存器中。

当然，寄存器 AL/AX/EAX 与其他寄存器相比并没有什么特别之处。如果有更方便的选择，可以将函数结果返回到任何寄存器中。然而，若没有其他更好的原因来说明不能使用 AL/AX/EAX 的话，最好还是遵循这个约定。因为程序的阅读者通常会假定函数将较小的返回值保存在寄存器 AL/AX/EAX 中，这样便于阅读者对代码的理解。

当函数返回结果的数据宽度大于 32 位时，显然要把结果返回到其他位置而不能在 EAX 中(因为 EAX 只能保存 32 位宽度的数据)。若数据宽度稍大于 32 位(如 64 位或 128 位)，则可以把数据拆分，并将拆分的结果保存到两个或多个寄存器中。经常可以见到程序将 64 位数据结果返回到寄存器对 EDX:EAX 中(例如，HLA 标准库函数 `stdin.geti64` 将 64 位整数返回到寄存器对 EDX:EAX 中)。

如果确实需要返回一个大规模的函数结果，如含有 1000 个元素的数组。这时显然不能把函数结果返回到寄存器中。有两种常用的方法来处理大规模的函数结果：将返回值作为引用参数传递，或使用 `mem.alloc` 在堆上为对象分配存储空间并将指针返回到一个 32 位寄存器中。当然，若返回一个指针指向堆中所分配的存储空间，调用程序在处理完成后必须释放这些存储空间。

5.7.2 HLA 中的指令合成

一些 HLA 标准库函数允许被作为其他指令的操作数来调用。例如，考虑下面这段代码：

```
if( cs.member( al, {'a'..'z'}) ) then
    .
    .
    .
endif;
```

根据高级语言的编程经验和 HLA 的经验，这段代码的目的是调用函数 `cs.member` 检验 AL 中的字符是否为小写字母字符。如果函数 `cs.member` 返回真，这段代码将顺序执行 `then` 后面的语句；如果函数 `cs.member` 返回假，则代码跳过 `if..then` 代码体。事实上，HLA 不支持函数调用直接作为 `if` 语句的布尔表达式。除此之外，这段代码并无什么特别之处(回顾第 1 章中有效表达式的完整集合)。那么这段代码将如何编译并运行来产生直观结果呢？

下一节的内容将会阐述如何在 HLA 的布尔表达式中使用函数调用。在理解这部分内容之前，首先要学习 HLA 中的指令合成(instruction composition)。

指令的合成使得一条指令可以作为另一条指令的操作数来使用。例如，`mov` 指令有两个操作数：源操作数和目的操作数。指令的合成允许使用一条有效的 80x86 机器指令替代任意一个操作

⁷ 第 6 章将会看到编程人员大多会把实数结果返回到何处。

数或全部替代。下面是一个简单的示例:

```
mov( mov( 0, eax ), ebx );
```

当然,这样做立即就带来了一个问题:这条语句意味着什么?要理解其内容,首先必须了解:大多数指令被编译的时候会“返回”一个值给编译器。对大多数指令而言,这个“返回”的值就是目的操作数。因此,对于指令 `mov(0, eax);`, 由于 `EAX` 为目的操作数,所以在编译时将字符串 `eax` 返回给编译器。多数情况下,当一行代码只出现一条指令时,编译器忽略指令返回的字符串结果。当用一条指令代替某操作数时,HLA 才使用返回的字符串结果。确切地说,HLA 使用这个字符串结果代替指令作为操作数。因此,上面的 `mov` 指令等价于下面的指令序列:

```
mov( 0, eax );          // HLA compiles interior instructions first.
mov( eax, ebx );        // HLA substituted "eax" for "mov( 0, eax )"
```

处理合成指令(含有其他指令作为操作数的指令序列)时,HLA 采用从左至右深度优先(由内而外)的工作方式。为了弄清这种工作方式的含义,考虑下面的指令:

```
add( sub( mov( i, eax ), mov( j, ebx ) ), mov( k, ecx ) );
```

下面从源操作数开始来解释这条指令如何工作的。源操作数为下面这条指令:

```
sub( mov( i, eax ), mov( j, ebx ) )
```

这条指令的源操作数为指令 `mov(i, eax)` 并且该指令不再含有合成成分,所以 HLA 发出该指令,同时将目的操作数(`eax`)返回并作为 `sub` 指令的源操作数使用。实际形式变为:

```
sub( eax, mov( j, ebx ) )
```

接下来 HLA 将编译作为目的操作数出现的指令 `mov(j, ebx)`, 并将目的操作数(`ebx`)返回以代替 `sub` 指令中的 `mov` 指令。产生结果为:

```
sub( eax, ebx )
```

这是一条 HLA 能够直接编译且不含合成成分的完整指令。所以 HLA 编译这条指令,并将目的操作数(`ebx`)作为字符串结果返回,以替代最初的 `add` 指令中的 `sub` 指令。此时,最初的 `add` 指令变为:

```
add( ebx, mov( k, ecx ) );
```

HLA 下一步编译在目的操作数中出现的 `mov` 指令。将目的操作数作为字符串结果返回以代替 `mov` 指令,最后产生一条简单指令:

```
add( ebx, ecx );
```

所以,对最初的 `add` 指令进行编译后产生下面的指令序列:

```
mov( i, eax );
mov( j, ebx );
```

```
sub( eax, ebx );
mov( k, ecx );
add( ebx, ecx );
```

从最初的指令很难直接看出上述指令序列。从这个示例中很容易看出，过多使用指令合成产生的程序的可读性非常差。因此在程序中使用指令合成时应格外小心。除了少数例外情况，一般书写指令合成序列会使程序变得难读。

还要注意的，过多地使用指令合成会使程序中的错误难于解释。考虑下面的 HLA 语句：

```
add( mov( eax, i ), mov( ebx, j ) );
```

这条合成指令产生的 80x86 指令序列为：

```
mov( eax, i );
mov( ebx, j );
add( i, j );
```

当然，编译器能够报错，指出程序将一个存储单元加到另一个存储单元上。然而，合成指令的使用使得这个事实被隐藏起来，这样就很难充分理解错误消息产生的原因。所以要尽量避免使用指令合成，除非确实能使程序更易读。上面所举的几个示例都说明了如何不去使用指令合成。

在两个主要的区域使用合成指令会大大增加程序的可读性。一个是 HLA 高级语言的控制结构；另一个是过程参数。虽然指令合成在这两种情况(或许还有其他一些情况)下很有益处，但这并不是说就可以随意使用类似前面示例中出现的 `add` 指令那样让人极其费解的指令。相反，很多时候应该使用一条指令或一个函数调用代替高级语言的布尔表达式或过程/函数参数中出现的单个操作数。

现在面临的问题是：过程调用将返回什么来作为 HLA 用于代替合成指令中所出现调用的字符串？同样情况下，`if..endif` 语句又将返回什么？没有目的操作数的指令又怎样？函数返回结果是下面将要论述的主题，马上就会看到。对于其他的语句和指令，请参见 HLA 参考手册。其中列出了每一条指令和它们所返回的值。当某条指令作为其他指令的操作数出现时，这个返回值就是 HLA 用于代替这种指令的字符串。需要说明的是，很多 HLA 语句和指令返回空字符串作为它们的返回值(默认情况下，过程调用亦如此)。如果一条指令返回空字符串作为其合成值并试图使用这条指令作为另一条指令的操作数，则 HLA 会报出错误。例如：`if..then..endif` 语句返回空字符串作为返回值，所以不能将 `if..then..endif` 语句嵌入另一条指令中使用。

5.7.3 HLA 过程的 @returns 选项

HLA 为过程声明提供一个特殊选项：`@returns`。使用这个选项可以明确指定一个字符串，当过程调用被作为另一条指令的操作数出现时使用这个指定的字符串。带有 `@returns` 选项的过程声明的语法形式如下：

```
procedure ProcName( optional_parameters ); @returns(string_constant);
    << Local declarations >>
begin ProcName;
    << Procedure statements >>
end ProcName;
```


如果没有出现@returns 选项, HLA 则使用空字符串作为过程的@returns 值。调用这样的过程来作为另一条指令的操作数是非法操作。

选项@returns 需要一个带括号的字符串表达式作为参数。如果一个过程作为另一条指令的操作数出现, HLA 将使用这个字符串常量代替过程调用。一般情况下, 使用一个寄存器的名称作为这个字符串常量, 然而任何可以作为合法的指令操作数的文本都可以用在这里。例如, 可以指定存储地址或常量。为了清晰起见, 应该总是在@returns 参数中指定函数返回值的位置。

考虑下面这个布尔函数的示例, 它将根据单个字符参数是否是一个字母而在寄存器 EAX 中返回 true 或 false⁸。

```
procedure IsAlphabeticChar( c:char ); @returns( "EAX" );
begin IsAlphabeticChar;
    // Note that cs.member returns true/false in eax.
    cs.member( c, { 'a'..'z', 'A'..'Z' } );
end IsAlphabeticChar;
```

一旦在过程声明末尾明确使用了@returns 选项, 就可以合法地调用 IsAlphabeticChar 作为其他 HLA 语句或指令的操作数:

```
mov( IsAlphabeticChar( al ), ebx );

if( IsAlphabeticChar( ch ) ) then

endif;
```

上面最后一个示例演示了如何通过@returns 选项将函数调用嵌入不同 HLA 语句的布尔表达式域中使用。上面的代码等价于:

```
IsAlphabeticChar( ch );
if( eax ) then

endif;
```

不是所有的 HLA 高级语言语句都在该语句之前扩展合成指令。例如, 考虑下面的 while 语句:

```
while( ~IsAlphabeticChar( ch ) ) do
```

⁸ 在程序中实际使用这个函数前, 注意 HLA 标准库提供了函数 char.isAlpha 来实现这种测试。若要了解更多细节请参见 HLA 文档。

```
endwhile;
```

这条语句没有扩展为：

```
IsAlphabeticChar( ch );
while( eax ) do
    .
    .
    .
endwhile;
```

相反，对 `IsAlphabeticChar` 的调用将在 `while` 语句的布尔表达式中扩展，所以程序会在每次循环时对函数进行一次调用。

在输入 `@returns` 参数时要谨慎，因为 HLA 在编译过程声明时并不检查字符串参数的语法(只验证它是否为一个字符串常量)。当用 `@returns` 字符串代替函数调用时，HLA 才进行语法检查。所以，在上面的示例中如果错误地将 `eax` 写为 `eaz` 作为过程 `IsAlphabeticChar` 中 `@returns` 的参数，直到真正使用 `IsAlphabeticChar` 作为一个操作数时，HLA 才会报出一个错误。当然，HLA 会报出非法操作数，但是只观察过程 `IsAlphabeticChar` 的调用并不能确定问题所在。因此要特别小心，不要在 `@returns` 的字符串中引入拼写错误，因为在后面去确定这个错误是很困难的。

5.8 递归

一个过程对自身进行调用，就称为递归(recursion)。下面是一个递归过程的示例：

```
procedure Recursive;
begin Recursive;

    Recursive();

end Recursive;
```

当然，运行这个递归过程后 CPU 将不再返回。进入过程 `Recursive` 后，过程将立即再一次调用自身，从而导致控制权不会被传递到过程末端。在这种特殊的情况下，递归将失控并导致无限循环⁹。

类似于循环结构，递归需要一个终止条件来停止无限递归。上面的递归过程 `Recursive` 增加了终止条件后变为：

```
procedure Recursive;
begin Recursive;

    dec( eax );
    if( @nz )then

        Recursive();

    end if;

end Recursive;
```

⁹ 实际上，并不是真正的无限循环。栈将溢出，同时 Windows、Mac OS X、FreeBSD 或 Linux 操作系统在此处产生异常。


```
endif;

end Recursive;
```

对例程进行的修改使过程 Recursive 调用自身若干次后返回,调用次数保存在寄存器 EAX 中。每调用一次,过程 Recursive 对寄存器 EAX 进行一次减1操作后再次调用自身。最终,Recursive 把寄存器 EAX 中的值减为 0 后返回。这时,每一个顺序的调用都返回上一个 Recursive,直到控制权返回最初的过程。

然而,对于前面这个示例,并不一定要使用递归。因为可以有效地把这个过程编码为:

```
procedure Recursive;
begin Recursive;

    repeat
        dec( eax );
    until( @z );

end Recursive;
```

两段示例过程都把过程体循环执行寄存器 EAX 中所指定的次数¹⁰。实际上,只有少数递归算法不能用循环方式实现。然而,许多递归实现的算法要比用循环方式实现的版本效率高。同时,多数情况下,递归形式的算法更易于理解。

快速排序是递归形式实现的著名算法之一。程序清单 5-9 是 HLA 实现的快速排序算法。

程序清单 5-9 递归的快速排序程序

```
program QSDemo;
#include( "stdFib.hhf" );

type
    ArrayType:    uns32[ 10 ];

static
    theArray:    ArrayType := [1,10,2,9,3,8,4,7,5,6];

procedure quicksort( var a:ArrayType; Low:int32; High:int32 );
const
    i:    text := "(type int32 edi)";
    j:    text := "(type int32 esi)";
    Middle:    text := "(type uns32 edx)";
    ary:    text := "[ebx]";

begin quicksort;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );
    push( esi );
```

¹⁰ 后一版本运行速度快,因为它没有 call/ret 指令的系统开销。


```

push( edi );

mov( a, ebx );           // Load BASE address of "a" into ebx.

mov( Low,edi );          // i :=Low;
mov( High,esi );         // j :=High;

// Compute a pivotal element by selecting the
// physical middle element of the array.

mov( i,eax );
add( j,eax );
shr( 1,eax );
mov( ary[eax*4], Middle );      // Put middle value in edx.

// Repeat until the edi and esi indicies cross one
// another (edi works from the start toward the end
// of the array,esi works from the end toward the
// start of the array).

repeat

    // Scan from the start of the array forward
    // looking for the first element greater or equal
    // to the middle element).

    while( Middle > ary[i*4] ) do

        inc( i );

    endwhile;

    // Scan from the end of the array backward looking
    // for the first element that is less than or equal
    // to the middle element.

    while( Middle < ary[j*4] ) do

        dec( j );

    endwhile;

    // If we've stopped before the two pointers have
    // passed over one another,then we've got two
    // elements that are out of order with respect
    // to the middle element.So swap these two elements.

    if(i <= j )then

        mov( ary[i*4],eax );
        mov( ary[j*4],ecx );
        mov( eax, ary[j*4] );
        mov( ecx, ary[i*4] );
        inc( i );
        dec( j );

    endif;

endrepeat;

```

```

until( i > j );

// We have just placed all elements in the array in
// their correct positions with respect to the middle
// element of the array. So all elements at indexes
// greater than the middle element are also numerically
// greater than this element. Likewise, elements at
// indexes less than the middle (pivotal) element are
// now less than that element. Unfortunately, the
// two halves of the array on either side of the pivotal
// element are not yet sorted. Call quicksort recursively
// to sort these two halves if they have more than one
// element in them (if they have zero or one elements, then
// they are already sorted).
if( Low < j )then
    quicksort( a, Low, j );
endif;
if( i < High )then
    quicksort( a, i, High );
endif;

pop( edi );
pop( esi );
pop( edx );
pop( ecx );
pop( ebx );
pop( eax );

end quicksort;

begin QSDemo;

stdout.put( "Data before sorting:" nl );
for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do
    stdout.put( theArray[ebx*4]:5 );
endfor;
stdout.newln();

quicksort( theArray, 0, 9 );

stdout.put( "Data after sorting:" nl );
for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do
    stdout.put( theArray[ebx*4]:5 );
endfor;
stdout.newln();

end QSDemo;

```

注意，上面这个快速排序过程使用寄存器保存过程中出现的所有非参数局部变量。同时还看到快速排序是如何使用 `text` 常量定义为寄存器提供更易读的名称。这些技巧都使得一个算法更易读，然而在使用这些技巧时要谨慎，不要忘记使用的是寄存器。

5.9 过程的向前引用

作为一条通用的规则，HLA 要求程序在使用符号之前对其进行声明¹¹。因此，对所有的过程来说，要在第一次调用它们之前对其进行定义。实际上，下面两个原因使得并不能保证总是这样：相互递归(两个过程互相调用)和源代码组织(您更愿意将代码中的过程放在第一次调用点之后)。HLA 提供了一种向前引用的过程定义方法来声明一个过程原型。向前引用声明的方法可以做到在真正提供过程的代码之前先定义一个过程。

向前引用的过程声明与普通的过程声明类似，差别在于使用保留字 `forward` 代替过程声明段和过程体。下面是对快速排序过程的向前引用声明：

```
procedure quicksort( var a:ArrayType; Low:int32; High:int32 ); forward;
```

HLA 程序中的向前引用声明实际上是对编译器的一个允诺。意思是说，实际的过程声明将会在源代码的后续某处出现并完全与向前引用声明相符合。“完全符合”的意思是：向前引用声明的参数与实际过程的形参必须相同，并且参数类型和传递方式也相同¹²。

两个相互递归的例程(过程 A 调用过程 B，同时过程 B 调用过程 A)至少需要对其中一个进行向前引用声明，因为这两个过程中总有一个要在其调用点之后出现。实际上，相互递归(直接或间接)的现象并不会经常出现，因此，出于这种目的而使用向前引用声明的情况并不多。

当不存在相互递归的时候，组织源码时就可以将每一个过程的声明放置在对它的第一次调用之前。然而，可能做到的和希望达到的是两件完全不同的事情。在实际编程的时候，您可能想把一组相关的过程分组放在源码中开始的部分，而将另一组过程放置在源码的末端。这种根据功能进行逻辑分组的方法相对于根据调用的分组方法来说，会使程序更易读，更易于理解。然而，这样组织代码就会产生试图在一个过程的声明前对它进行调用的情况。使用向前引用的过程定义可以很轻松地解决这类问题。

向前引用定义与实际过程声明之间存在的一个主要差别是可用的过程选项不同。某些选项，如 `@returns`，只能出现在向前引用声明中(如果有向前引用声明的话)。其他的选项则只能出现在实际的过程声明部分(目前为止还没有涉及其他的过程选项，因此先不必考虑)。如果某个过程需要使用 `@returns` 选项，则 `@returns` 必须出现在保留字 `forward` 之前。例如：

```
procedure IsItReady( valueToTest: dword ); returns( "eax" ); forward;
```

同时，`@returns` 选项不能在源文件后面的实际过程声明中再次出现。

¹¹ 针对这条规则会有一些例外，但对于过程调用是完全正确的。

¹² 事实上，“完全”一词过于绝对。您将在后面看到一些例外。

5.10 HLA v2.0 的过程声明

HLA v2.0 及更新的版本支持另外一种声明语法, 类似于常量、类型和变量声明。虽然本书选择原来的过程声明语法(HLA v2.0 及更新版本仍然支持), 但是在实际代码中会看到使用新语法的地方。因此, 本节将简要讨论新的过程声明语法。

新的 HLA v2.0 过程声明语法使用 `proc` 关键字作为过程声明段的开始(类似于 `var` 或 `static` 关键字是变量声明段的开始)。在 `proc` 段中, 过程声明采用如下形式:

```
procname: procedure( parameters );
begin procname;
    << body >>
end procname;

procname: procedure( parameters ) {options};
begin procname;
    << body >>
end procname;

procname: procedure( parameters ); external;
procname: procedure( parameters ) { options }; external;
```

有关这一种过程声明语法的更多细节, 请参看 HLA v2.0(或更新版本)的参考手册。在阅读从其他来源获得的示例 HLA 代码时, 要记住有这样一种过程声明语法, 这样在遇到时才不会感到迷惑。

5.11 过程的底层实现与 `call` 指令

80x86 的 `call` 指令完成两个功能: 首先将紧跟 `call` 指令之后的指令地址推入栈顶, 然后将控制转向调用过程的地址。`call` 指令推入栈顶的数据称为返回地址。当被调用过程想返回调用程序, 并以 `call` 指令后面的第一条指令开始继续执行时, 过程只须从栈顶把返回地址弹出并跳转(间接地)到相应位置开始执行。多数过程通过执行一条 `ret` 指令来返回调用程序。`ret` 指令从栈顶弹出返回地址并将控制间接地转向这个地址。

默认情况下, HLA 会自动地在程序中每一个过程末尾放置一条 `ret` 指令(随同其他一些指令)。这也是不必显式地使用 `ret` 指令从过程返回的原因。如果不想使用 HLA 对过程的这种默认代码生成, 可以在过程声明的时候指定下面的选项:

```
procedure ProcName; @noframe; @nodisplay;
begin ProcName;

.
.
.

end ProcName;
```

子句 `@noframe` 和 `@nodisplay` 都是 HLA 的过程选项。HLA 过程支持几个这样的选项, 包括 `@returns`、`@noframe`、`@nodisplay` 和 `@noalignstack`。本章 5.14 节会介绍选项 `@noalignstack` 的用途

和另一对过程选项。这些过程选项可以按照任意顺序跟在过程名(和参数)的后面。需要注意的是,选项@noframe 和@nodisplay(也包括@noalignstack)只能出现在实际过程声明中,而不可在向前引用声明里指定这些选项。

选项@noframe 通知 HLA 编译器不必自动为过程生成入口和出口代码。也就是让 HLA 不用自动生成 ret 指令(以及其他一些指令)。

选项@nodisplay 通知 HLA 不必在过程的局部变量区域为显示(display)而分配存储空间。显示是过程用于访问非局部-var 对象的机制。因此,只有把过程嵌入在程序中时,显示才是必须的。本书不考虑显示或嵌入的过程,如果想了解显示和嵌入过程的细节,请参见 HLA 参考手册。目前,可以安全地对所有的过程指定@nodisplay 选项。事实上,对本章中到目前为止出现的所有过程指定@nodisplay 选项是很有意义的,因为这些过程实际上没有一个使用了显示。指定了@nodisplay 选项的过程要比没有这个选项的过程速度快、长度短。

下面是一个最小的过程示例:

```

procedure minimal; @nodisplay; @noframe; @noalignstack;
begin minimal;
    ret();
end minimal;

```

如果使用 call 指令调用这个过程,该过程将立即弹出栈的返回地址并返回调用程序。值得注意的是,如果指定了过程的@noframe 选项,那么 ret 指令是不可缺少的¹³。若没有在过程中写入 ret 指令,那么即使执行到过程的 end minimal;语句,程序也不会返回调用程序。相反,程序会错误地进入存储器中物理位置紧接在过程后面的代码。程序清单 5-10 中的示例程序演示了这种问题。

程序清单 5-10 过程中缺少 ret 指令造成的影响

```

program missingRET;
#include( "stdlib.hhf" );

// This first procedure has the @noframe
// option but does not have a ret instruction.
procedure firstProc; @noframe; @nodisplay;
begin firstProc;
    stdout.put( "Inside firstProc" nl );
end firstProc;

// Because the procedure above does not have a
// ret instruction, it will "fall through" to
// the following instruction. Note that there
// is no call to this procedure anywhere in
// this program.

procedure secondProc; @noframe; @nodisplay;

```

¹³严格地讲,这条指令并不是必需的。但在过程体中必须有某种机制可以从栈顶弹出返回地址并跳转到这个地址。


```
begin secondProc;

    stdout.put( "Inside secondProc" nl );
    ret();

end secondProc;

begin missingRET;

    // Call the procedure that doesn't have
    // a ret instruction.

    call firstProc;

end missingRET;
```

虽然这种行为在某些少见的情况下是合乎要求的，但在多数程序中表现为故障。因此，如果指定了过程的@noframe 选项，要切记使用 ret 指令从过程显式返回。

5.12 过程与栈

过程是使用栈来保存返回地址的，所以在过程内部对栈进行操作时应当格外小心。考虑下面这个简单(但有缺陷)的过程：

```
procedure MessedUp; noframe; nodisplay;
begin MessedUp;

    push( eax );
    ret();

end MessedUp;
```

当程序执行到 ret 指令时，80x86 的栈形式如图 5-1 所示。

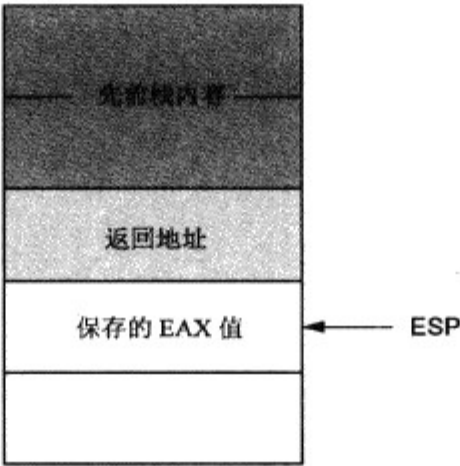


图 5-1 过程 MessedUp 执行到 ret 前的栈内容

此时 ret 指令并不知道栈顶数据不是一个有效的地址。它只是简单地把栈顶数据弹出并跳转到以该数据作为地址所指向的位置。在这个示例中，栈顶存放的是寄存器 EAX 所保存的数值。若要寄存器 EAX 中保存的就是正确的返回地址几乎是不可能的(确切地讲，这种可能性是四十亿分之一)，此时程序很可能会崩溃或出现一些不可知的行为。因此在过程体内向栈顶存放数据的时候

候要格外小心，确保在过程返回前正确地将栈顶数据弹出。

注意：

在编写过程时如果没有指定@noframe 选项，HLA 会自动生成代码在过程开始处向栈顶存放数据。因此，除非您完全理解正在发生的事情并且小心不破坏 HLA 放在栈顶的数据，否则，在没有指定@noframe 选项的过程内部不要执行一条空的 ret 指令。这样做会返回栈顶数据指向的位置(并不是返回地址)，但没有正确地返回调用程序。在没有指定选项@noframe 的过程中，使用 exit 或 exitif 语句返回调用程序。

在过程内部，还没有执行到 ret 指令却从栈顶弹出了数据，同样会对程序造成破坏。考虑下面这段有错误的过程：

```
procedure messedUpToo; @noframe; @nodisplay;
begin messedUpToo;

    pop( eax );
    ret();

end messedUpToo;
```

当过程运行到 ret 指令但还没有执行它的时候，80x86 的栈形式如图 5-2 所示。

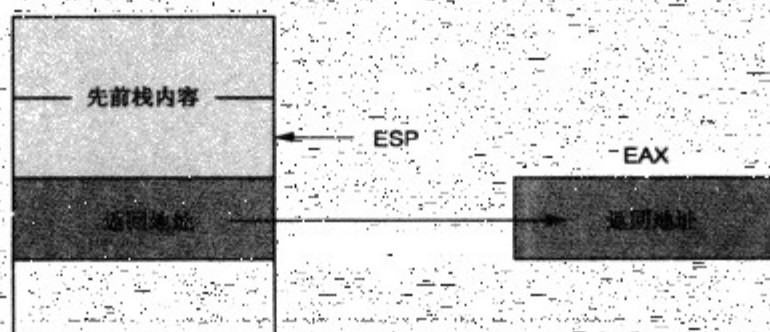


图 5-2 执行到过程 messedUpToo 的 ret 指令前的栈内容

无论栈顶存放的是什么数据，ret 指令都会盲目地再一次将其弹出并试图返回这个数据指向的地址。与前面的示例不同，在前面的示例中，栈顶几乎不可能保存一个有效的返回地址(因为栈顶保存的是寄存器 EAX 中的数值)，而对于这个示例，栈顶却有可能包含了有效地址。然而，这个地址并不是过程 messedUpToo 可以正确返回的地址，而是调用 messedUpToo 的过程应该返回的地址。为便于理解，考虑程序清单 5-11 的程序。

程序清单 5-11 从栈过多地弹出数据造成的影响

```
program extraPop;
#include( "stdlib.hhf" );

// Note that the following procedure pops
// excess data off the stack (in this case,
// it pops messedUpToo's return address).

procedure messedUpToo; @noframe; @nodisplay;
begin messedUpToo;
```



```

    stdout.put( "Entered messedUpToo" nl );
    pop( eax );
    ret();

end messedUpToo;

procedure callsMU2; @noframe; @nodisplay;
begin callsMU2;

    stdout.put( "calling messedUpToo" nl );
    messedUpToo();

    // Because messedUpToo pops extra data
    // off the stack, the following code
    // never executes (because the data popped
    // off the stack is the return address that
    // points at the following code).

    stdout.put( "Returned from messedUpToo" nl );
    ret();

end callsMU2;

begin extraPop;

    stdout.put( "Calling callsMU2" nl );
    callsMU2();
    stdout.put( "Returned from callsMU2" nl );

end extraPop;

```

由于栈顶是一个有效的返回地址，所以程序将继续运行下去。然而，需要注意的是，当从过程 `messedUpToo` 返回时，代码将直接返回主程序而不是 `callsMU2` 过程中的返回地址。因此，在过程 `callsMU2` 中，调用语句 `messedUpToo()` 之后的代码将不会被执行。在阅读源代码的时候，很难判定为什么紧接在过程 `messedUpToo` 调用之后的语句没有被执行。除非仔细阅读代码，才会发现程序在执行的过程中额外地从栈顶弹出了一个返回地址，因此，程序没有返回过程 `callsMU2` 而是直接返回调用 `callsMU2` 的程序段。当然，从这个示例很容易看出程序将如何执行(因为这个示例就是用来说明这种问题的)。然而，在实际编程的时候，确定一个过程是否偶然从栈顶弹出过多的数据是很难的事情。因此，在一个过程中进行压栈出栈操作的时候应多加小心。在编写过程时，应该总是确保过程内部压栈出栈操作的一一对应关系。

5.13 活动记录

当对一个过程进行调用的时候，程序会将某些信息与这个过程调用相关联。返回地址就是这种信息的示例，程序会为过程的某次具体调用而维护这样的信息。参数和自动局部变量(在 `var` 段声明的变量)也是程序为每次过程调用所要维护的信息。这些信息都是程序用来和一个具体的过

程调用相关联的，活动记录(activation record)是用来描述这些信息的术语¹⁴。

活动记录对这种数据结构来说是一个很合适的名字。因为程序在调用(激活)一个过程的时候会创建一个活动记录，并将该结构内的数据按照与记录相同的方式进行组织。活动记录和标准的记录结构相比，唯一的不同是活动记录的基址处在数据结构的中间，因此对活动记录字段的访问必须从正负两个偏移方向进行。

一个活动记录的建立是从代码调用一个过程开始的。调用程序把参数数据放在栈顶。然后，call 指令的执行会把返回地址放在栈顶。这时候，活动记录的建立会继续在过程内部进行。过程把寄存器和其他重要的状态信息放在栈顶，同时在活动记录内为局部变量准备空间。过程还必须对寄存器 EBP 进行更新，让它指向活动记录的基址。

为说明一个典型活动记录的样子，考虑下面的 HLA 过程声明：

```

procedure ARDemo( i:uns32; j:int32; k:dword ); @nodisplay;
var
    a:int32;
    r:real32;
    c:char;
    b:boolean;
    w:word;
begin ARDemo;

.
.
.

end ARDemo;

```

当某个 HLA 程序调用过程 ARDemo 时，首先在栈顶保存参数数据。调用代码将按照参数在列表中从左到右的出现顺序把它们推入栈。因此，调用代码先是把参数 i 的数值推入栈，然后是参数 j，最后把参数 k 的数值也推入栈。参数保存完成，程序开始调用过程 ARDemo。进入过程 ARDemo 的入口点时，栈包含 4 项内容，排列情况如图 5-3 所示。

过程 ARDemo(注意，它没有 @noframe 选项)的最初几条指令把寄存器 EBP 中的数值推入栈顶，同时把寄存器 ESP 的内容复制到 EBP 中。接下来，程序将存储器中的栈指针下移，目的是为局部变量准备空间。这时候的栈结构如图 5-4 所示。

¹⁴ 栈帧(stack frame)是人们描述活动记录的另一个术语。

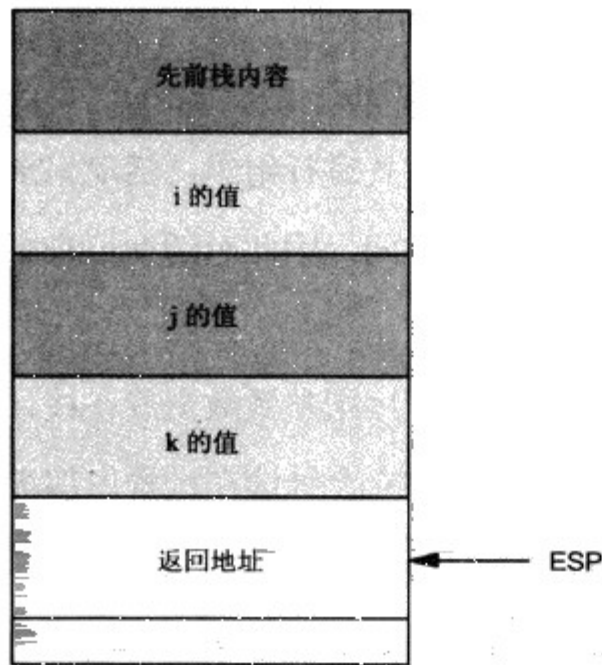


图 5-3 进入过程 ARDem 入口点时的堆栈结构

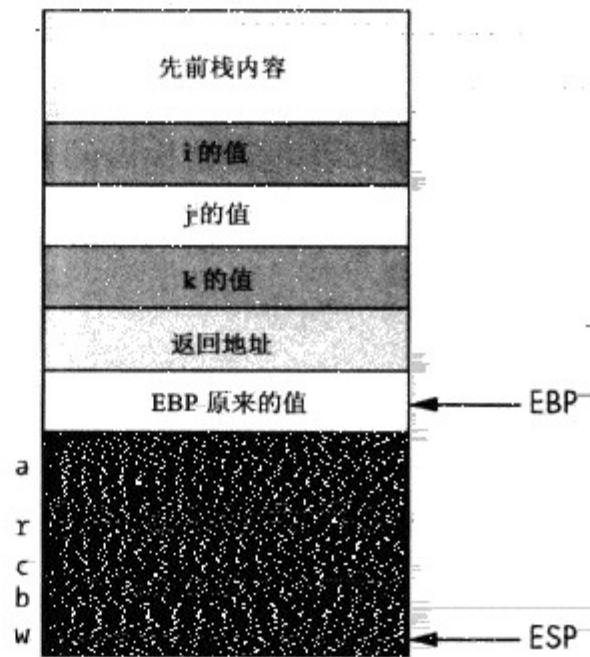


图 5-4 ARDemo 的活动记录

要访问活动记录中的对象，必须采用基址偏移的方法，即用基址寄存器 EBP 的内容加上偏移量来访问所期望的对象。一般只关心参数和局部变量两项。可以通过基址寄存器 EBP 的正向偏移来访问参数，通过负向偏移访问局部变量，如图 5-5 所示。

Intel 公司保留寄存器 EBP(扩展基址指针)作为活动记录的基址指针使用。这也是不能把寄存器 EBP 用于通用计算的原因。如果擅自改变寄存器 EBP 的值，将丧失对当前过程的参数和局部变量的访问权。

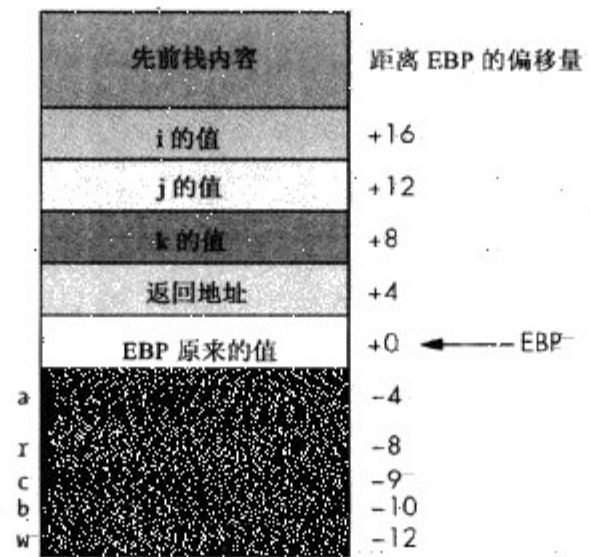


图 5-5 ARDemo 活动记录中对象的偏移量

5.14 标准入口序列

过程的调用者负责把参数推入栈顶。同时，call 指令负责将返回地址推入栈顶，而活动记录剩余部分的构造由过程负责完成。使用下面的标准入口序列代码可以实现上述过程。

```
push( ebp );           // Save a copy of the old ebp value.
mov( esp, ebp );       // Get pointer to base of activation record into ebp.
sub( NumVars, esp );   // Allocate storage for local variables.
```

如果过程不含有任何局部变量,则不必使用上面的第三条指令 `sub(NumVars,esp)`。其中, `NumVars` 表示过程中的局部变量需要占用的字节数,它是一个常量并且是 4 的整数倍(这样可以保证寄存器 ESP 在双字边界上对齐)。如果过程中局部变量使用的字节数不是 4 的整数倍,应先把这个字节数向上舍入为最近的一个 4 的倍数值,再对寄存器 ESP 进行减法操作。这样做会使得过程中局部变量占用的存储空间略有增加,但却是为了保证不影响过程操作的正常进行。

警告:

如果常量 `NumVars` 不是 4 的整数倍,并且从寄存器 ESP(通常包含双字对齐的指针)中减去了该数值,那么,由于程序推入或弹出堆栈的几乎都是双字数值,这会导致后续的栈访问都是非对齐的。这对程序的性能会产生很大的负面影响。更糟糕的是,如果操作系统入口点的栈是非双字对齐的,很多操作系统的 API 调用将会失败。因此,必须确保为局部变量分配的存储空间字节数是 4 的整数倍。

因为存在栈的非对齐问题,默认情况下,HLA 还会发出作为标准入口序列一部分的第 4 条指令。对于前面定义的过程 `ARDemo`,HLA 编译器将生成下面的标准入口序列:

```

push( ebp );
mov( esp, ebp );
sub( 12, esp );           // Make room for ARDemo's local variables.
and( $FFFF_FFC, esp );   // Force dword stack alignment.

```

序列末尾的 `and` 指令强制栈为 4 字节边界对齐的(若 ESP 中的数值不是 4 的整数倍,则该指令把栈指针的数值减去 1、2 或 3)。过程 `ARDemo` 的入口点代码为局部变量分配空间时从 ESP 中减去 12(12 既是局部变量使用的字节数也是 4 的倍数)。若在过程的入口点 ESP 是双字对齐的,那么这样操作后 ESP 仍是双字对齐的。如果调用程序打乱了栈,使得 ESP 包含的数值不是 4 的整数倍,那么,从 ESP 中减去 12 后它包含的仍是非对齐的数值。使用上述序列中的 `and` 指令,无论 ESP 在过程的入口点所包含的是什么数值,都确保 ESP 是双字对齐的。如果 ESP 是非双字对齐的,那么用于执行这条指令所占用的字节数和 CPU 周期将是微不足道的。

尽管在标准入口序列里执行 `and` 指令是安全的,但也可能是不必要的。如果总能够保证 ESP 中包含的数值是双字对齐的,那么标准入口序列中的 `and` 指令就多余了。因此,若指定了过程的 `@noframe` 选项,`and` 指令就不必作为标准入口序列的一部分。

如果没有指定 `@noframe` 选项(也就是说,让 HLA 生成指令来构建标准入口序列),同时能够确定无论何时对过程进行调用栈都是双字对齐的,那么,仍然可以告知 HLA 不发出这条额外的 `and` 指令。使用过程选项 `@noalignstack` 来完成上述过程,例如:

```

procedure NASDemo( i:uns32; j:int32; k:dword ); @noalignstack;
var
    LocalVar:int32;
begin NASDemo;
    .
    .
    .
end NASDemo;

```

HLA 为上面这个过程发出的标准入口序列为:

```
push( ebp );
mov( esp, ebp );
sub( 4, esp );
```

5.15 标准出口序列

当一个过程返回调用程序之前,必须清除活动记录。当然,这种清除任务可以由过程与过程调用者共同承担完成,但 Intel 公司在指令集中所包含的某些功能使得过程本身可以有效地处理所有的清理工作。因此,当过程返回调用程序时,HLA 的标准过程和过程调用者都假定由过程本身负责清除活动记录(包括参数)。

如果一个过程没有任何参数,那么出口序列将非常简单。它只需要 3 条指令:

```
mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret();              // Return to the caller.
```

如果过程含有参数,为了把这些参数移出栈,必须对标准出口序列做相应修改。含有参数的过程使用下面的标准出口序列:

```
mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret( ParmBytes );   // Return to the caller and pop the parameters.
```

上面 `ret` 指令的操作数 *ParmBytes* 是一个常量,当返回指令从栈弹出返回地址后,用这个常量确定从栈移出参数数据的字节数。例如,前面提到的示例 `ARDemo` 有 3 个双字参数。因此,它的标准出口序列为:

```
mov( ebp, esp );
pop( ebp );
ret( 12 );
```

如果是使用 HLA 的语法对参数进行声明的(即在过程声明后使用参数列表),则 HLA 在过程中自动生成一个局部常量 `_parms_`,这个常量等于过程中参数数据的字节数。因此,可以对任何含有参数的过程使用下面的标准出口序列而不必自己计算参数数据的字节数:

```
mov( ebp, esp );
pop( ebp );
ret( _parms_ );
```

当然,如果没有给 `ret` 指令指定一个字节常量的参数,当过程返回时,80x86 不会把参数从栈顶弹出。当开始执行过程调用指令 `call` 的后续指令时,过程的参数仍然保留在栈顶。类似地,如果指定的数值太小,过程返回时仍有部分参数被保留在栈中。相反,若指定的常量数值过大,`ret` 指令将错误地把调用程序的数据从栈顶弹出,这将导致灾难性的后果。

若想从一个不含@noframe 选项的过程提前返回，同时又不想使用 exit 或 exitif 语句，则必须执行标准出口序列来返回调用程序。一条简单的 ret 指令是不够的，因为局部变量和 EBP 的原始数据仍然保存在栈顶。

5.16 自动(局部)变量的底层实现

程序使用活动记录中基址(EBP)的负向偏移访问一个过程的局部变量。考虑下面的 HLA 过程 (只是为了演示局部变量的使用而并无其他功能):

```
procedure LocalVars; @nodisplay;
var
    a:int32;
    b:int32;
begin LocalVars;

    mov( 0,a );
    mov( a,eax );
    mov( eax,b );

end LocalVars;
```

过程 LocalVars 的活动记录如图 5-6 所示。

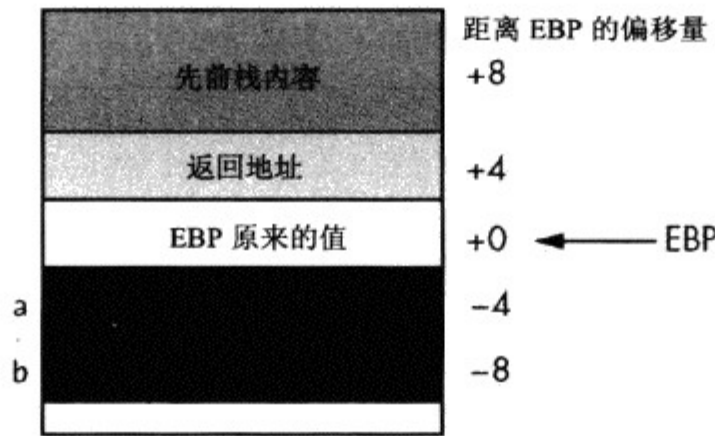


图 5-6 过程 LocalVars 的活动记录

HLA 编译器为过程体发出的代码粗略如下¹⁵:

```
mov( 0, (type dword [ebp-4]));
mov( [ebp-4], eax );
mov( eax, [ebp-8] );
```

事实上，可以自己把这些语句写入过程中来完成相应功能。然而，使用形如[ebp-4]和[ebp-8]的存储器访问代替 a 和 b 会使得程序难读且难理解。因此，应尽量声明并使用 HLA 的符号名称代替 EBP 的偏移量。

15 这里忽略了标准出口序列和入口序列的相关代码。

过程 LocalVars 的标准入口序列为¹⁶:

```
push( ebp );
mov( esp,ebp );
sub( 8,esp );
```

由于过程中有 8 字节的局部变量(2 个双字对象), 所以代码从栈指针减去 8。然而, 当局部变量的数目增加, 尤其是这些变量的字节数不同时, 对局部变量字节数的计算将变得冗长乏味。幸运的是, 对于那些自己编写标准入口序列的人来说, HLA 会自动计算这个值并创建常量 `_vars_`, 它明确指出局部变量包含的字节数¹⁷。因此, 如果打算自己编写标准入口序列, 在为局部变量分配存储空间时, 应该在 `sub` 指令中使用常量 `_vars_`:

```
push( ebp );
mov( esp,ebp );
sub( _vars_,esp );
```

现在, 您已看到汇编语言是如何为局部变量分配并释放存储空间的, 同时也就很容易理解为什么自动变量(var)在同一个过程的两次调用间不保持它们的值。因为与自动变量相关联的内存被放在栈顶, 当一个过程返回时, 调用程序可以继续把其他数据推入栈而删除先前保存在栈顶的局部变量数值。此外, 插入调用其他过程(含有自己的局部变量)将掩盖栈顶的数值。而且, 当再次进入一个过程时, 它所包含的局部变量对应于不同的物理存储单元。因此, 局部变量的数值不会在先前的相应位置。

自动变量有一个很大的优点, 就是可以在多个过程间有效地共享一个固定的存储池。例如, 如果顺序调用 3 个过程:

```
ProcA();
ProcB();
ProcC();
```

第一个过程(上面代码中的 `ProcA`)在栈顶为局部变量分配存储空间。返回时, `ProcA` 释放栈存储空间。进入过程 `ProcB`, 程序使用刚被过程 `ProcA` 释放的存储单元为过程 `ProcB` 的局部变量分配存储空间。同样地, 当过程 `ProcB` 返回并且程序调用 `ProcC` 时, `ProcC` 使用刚被 `ProcB` 释放的栈空间为局部变量分配存储空间。这种存储空间的重用有效地利用了系统资源, 同时也是使用自动变量(var)带来的最大优点。

5.17 参数的底层实现

前面讨论 HLA 的高级参数传递机制时, 有许多与参数相关的问题。其中一些重要的问题是:

- 数据是从哪里来的?
- 使用什么机制传递并返回数据?
- 要传递的数据量是多大?

¹⁶ 该代码假定在入口点 ESP 是双字对齐的, 因此不需要指令 `and($FFFF_FFFC,esp);`。

¹⁷ HLA 同时会把这个常量向上舍入为最近的 4 的偶数倍, 因此不必担心栈的对齐问题。

本节我们将从另一个角度考虑两种最常用的参数传递机制：值传递和引用传递。我们将讨论通过值或引用方式传递参数的 3 个常用地方：寄存器、栈和代码流。参数数据的数量决定使用什么方式将它们传递到何处。下面的章节开始讨论这些问题。

5.17.1 在寄存器中传递参数

本章 5.6 节已经粗略讨论了如何向一个过程传递参数，接下来要讨论的是把参数传递到何处。把参数传递到哪里取决于参数的大小和数量。如果只传递很少的字节数，寄存器是用来向过程传递参数的极好位置。如果只向过程传递一个参数，应该使用下列与相应数据类型对应的寄存器：

数据长度	用来传递的寄存器
字节	al
字	ax
双字	eax
四字	edx:eax

这并不是一个严格的规则。如果发现使用寄存器 SI 或 BX 传递 16 位数值更方便，完全可以这样做。然而，多数程序员习惯于使用上面的寄存器传递参数。

如果使用 80x86 的寄存器向过程传递多个参数，应该按照下面的顺序来使用寄存器：

先	后
eax,edx,ecx,esi,edi,ebx	

通常情况下，应该避免使用寄存器 EBP。如果需要传递的数值超过 6 个双字，应该使用其他的位置传递数值。这种优先选择顺序不是完全任意的。许多高级语言会试图使用寄存器 EAX、EDX 和 ECX 传递参数(通常按这个顺序)。而且，Intel 的 ABI(application binary interface, 应用程序二进制接口)允许高级语言的过程在使用寄存器 EAX、EDX 和 ECX 时不必保存它们的值。因此，这 3 个寄存器是传递参数的主要位置，因为许多代码都假定它们的数值在过程调用的时候被修改。

例如，考虑下面的过程 `strfill(s,c)`，它将字符 `c`(通过值传递到寄存器 AL 中)在字符串 `s`(引用方式传递到寄存器 EDI 中)中的每一个字符位置进行一次复制，直到出现 0 结束字节：

```
// strfill-Overwrites the data in a string with a character.
//
//   EDI- Pointer to zero-terminated string (e.g., an HLA string)
//   AL- Character to store into the string

procedure strfill; @nodisplay;
begin strfill;

    push( edi ); // Preserve this because it will be modified.
    while( (type char [edi] ) <> #0 ) do

        mov( al, [edi] );
        inc( edi );

    endwhile;
    pop( edi );
```



```
end strfill;
```

在调用过程 `strfill` 之前, 应首先将字符串数据的地址写入寄存器 `EDI`, 同时把字符数据保存在 `AL` 中。下面的代码段是对过程 `strfill` 的调用示例:

```
mov( s, edi );      // Get ptr to string data into edi (assumes s:string).
mov( 'F', al );
strfill();
```

不要忘记, `HLA` 的字符串变量是指针。这个示例假定 `s` 是 `HLA` 的字符串变量, 因此包含一个指向 0 结束字符串的指针。因此, 指令 `mov(s,edi)` 将 0 结束字符串的地址写入寄存器 `EDI` (因此, 这句代码将字符串数据的地址传递给 `strfill`, 也就是使用引用方式传递字符串)。

通过寄存器进行参数传递的一种方法是在调用前写入参数的相应值, 而后在过程内部对这些寄存器进行引用。在汇编语言编程中, 这是一种使用寄存器传递参数的传统机制。`HLA` 与传统的汇编语言相比相对高级, 它提供了一种形参声明语法以告知 `HLA` 正在使用通用寄存器传递某个特定参数。这种声明语法为:

```
parmName: parmType in reg
```

其中 `parmName` 是参数名称, `parmType` 是参数类型, `reg` 是一个 80x86 的 8 位、16 位或 32 位的通用寄存器。参数类型的数据宽度必须与寄存器的容量相同, 否则 `HLA` 会报出错误。下面是一个具体示例:

```
procedure HasRegParms( count: uns32 in ecx; charVal:char in al );
```

这种语法有一个很好的特性, 它使您可以像调用其他过程一样使用 `HLA` 的高级语法调用一个含有寄存器参数的过程, 例如:

```
HasRegParms( ecx, bl );
```

如果指定与所声明的形参相同的寄存器作为实参, `HLA` 将不再发出其他任何额外代码, 它假定参数的数值已经保存在相应的寄存器中。例如, 上述调用中的第一个实参就是寄存器 `ECX` 中的值, 因为过程声明指定了第一个参数在 `ECX` 中, 故 `HLA` 将不再发出任何代码。另一方面, 第二个实参在寄存器 `BL` 中, 但过程却希望这个参数在 `AL` 中。因此, `HLA` 会在调用过程之前发出指令 `mov(bl,al)`。这样, 在过程的入口点, 参数的数值就已经保存在相应的寄存器中了。

同样可以通过寄存器使用引用方式传递参数。考虑下面的声明:

```
procedure HasRefRegParm( var myPtr:uns32 in edi );
```

对这个过程的调用需要存储器操作数作为实参。`HLA` 将发出代码把存储器对象的地址写入参数的寄存器(本例中是指 `EDI`)。需要注意的是, 由于地址是 32 位数据宽度的, 因此, 传递引用参数时必须使用 32 位的通用寄存器。下面是一个调用 `HasRefRegParm` 的示例:

```
HasRefRegParm( x );
```

在调用指令 `call` 之前, `HLA` 将发出 `mov(&x,edi)` 指令或 `lea(edi,x)` 指令, 目的是把 `x` 的地址写

入寄存器 EDI 中¹⁸。

如果把一个匿名存储器对象(如[edi]或[ecx])作为参数传递给过程 HasRefRegParm, 同时, 存储器引用所使用的恰好是您为参数声明的那个寄存器(即[edi]), 那么, HLA 将不再生成任何代码。如果不使用寄存器 EDI, 而指定另一个间接寻址方式的寄存器(如[ecx]), HLA 将使用一条简单的 mov 指令把实际地址复制到寄存器 EDI 中。如果使用更加复杂的寻址方式如[edi+ecx*4+2], HLA 将使用 lea 指令来计算匿名存储器操作数的有效地址。

在过程代码内部, HLA 会为寄存器参数创建等价的文字名称, 并把它们映射到相应的寄存器。在 HasRegParms 的示例中, 无论何时引用参数 count, HLA 都会用 ecx 取代 count。同样地, 在过程体内部, HLA 使用 al 取代 charVal。由于这些都是寄存器的别名, 因此需要当心, 不能再独立于这些名称使用 ECX 和 AL。一个很好的编程习惯是在这些参数的每一次使用后都加上一条注释, 提醒程序阅读者: count 等价于 ECX, charVal 等价于 AL。

5.17.2 在代码流中传递参数

传递参数的另一个位置是在紧跟 call 指令之后的代码流中。考虑下面的例程 print, 该例程在标准输出设备上打印一串文字:

```
call print;
byte "This parameter is in the code stream.",0;
```

通常情况下, 子例程将控制权返回给紧跟 call 指令之后的第一条指令。在这里如果仍然这样执行的话, 80x86 将试图把“This...”的 ASCII 码解释为一条指令, 这将产生不合乎要求的结果。幸运的是, 当从子例程返回时可以跳过这个字符串。

那么将如何获得这些参数的访问权呢? 很简单, 栈顶的返回地址是指向它们的。考虑程序清单 5-12 中过程 print 的实现。

程序清单 5-12 过程 Print 的实现(使用代码流参数)

```
program printDemo;
#include( "stdlib.hhf" );

// print-
//
// This procedure writes the literal string
// immediately following the call to the
// standard output device. The literal string
// must be a sequence of characters ending with
// a zero byte (i.e., a C string, not an HLA
// string).

procedure print; @noframe; @nodisplay;
const
    // RtnAdrs is the offset of this procedure's
    // return address in the activation record.
```

¹⁸ 指令的选择由 x 是否为静态变量决定(静态对象使用 mov 指令, 其他对象使用 lea 指令)。

```

    RtnAdrs:text := "(type dword [ebp+4])";

begin print;

    // Build the activation record (note the
    // @noframe option above).

    push( ebp );
    mov( esp,ebp );

    // Preserve the registers this function uses.

    push( eax );
    push( ebx );

    // Copy the return address into the ebx
    // register.Because the return address points
    // at the start of the string to print,this
    // instruction loads ebx with the address of
    // the string to print.

    mov( RtnAdrs,ebx );

    // Until we encounter a zero byte, print the
    // characters in the string.

    forever

        mov( [ebx],al );      // Get the next character.
        breakif( !al );      // Quit if it's zero.
        stdout.putc( al );   // Print it.
        inc( ebx );          // Move on to the next char.

    endfor;

    // Skip past the zero byte and store the resulting
    // address over the top of the return address so
    // we'll return to the location that is one byte
    // beyond the zero terminating byte of the string.

    inc( ebx );
    mov( ebx, RtnAdrs );

    // Restore eax and ebx.

    pop( ebx );
    pop( eax );

    // Clean up the activation record and return.

    pop( ebp );
    ret();

end print;

begin printDemo;

```



```
// Simple test of the print procedure.

call print;
byte "Hello World!", 13, 10, 0 ;

end printDemo;
```

除了说明如何在代码流中传递参数外, 例程 `print` 还引入了另一个概念: 可变长度参数 (variable-length parameter)。紧跟 `call` 指令后面的字符串可以是任何可能的数据宽度。结束字节 0 标志着参数列表的结尾。有两种简单的方法处理可变长度参数: 使用某些特殊的结束数值(如 0) 或传递一个特定长度的数值以告知子例程所传递参数的个数。两种方法各有其优缺点。使用特殊数值结束一个参数列表时, 必须选择一个不会在列表中出现过的数值。例如, `print` 使用 0 作为结束数值, 因此它不能打印字符 NUL(ASCII 码是 0)。但这往往不算是一个限制。指定一个特定长度的参数是传递可变长度参数列表的另一种机制。这种方法不需要任何特殊代码, 同时也不限制向子例程传递的数值的范围。但可变长度参数的设置和结果代码的维护是很困难的事情¹⁹。

尽管在代码流中传递参数提供了很多方便, 它同样存在一些缺陷。首先, 如果没有给出过程所需参数的精确数目, 那么子例程将执行混乱。考虑示例程序 `print`, 它打印一串字符直到遇见 0 结束字节, 然后把控制权交给 0 结束字节后的第一条指令。如果没有 0 结束字节, 例程 `print` 会把后面的操作码作为 ASCII 字符打印, 直到遇见一个 0 字节。由于 0 字节经常出现在一些指令的中间, 例程 `print` 可能把控制权返回到其他某些指令的中部, 这样很可能会使机器崩溃。额外插入 0 字节的操作出现的频繁程度要比想象的高, 这是编程者在使用例程 `print` 的时候出现的另一个问题。在这样的情况下, 例程 `print` 将在遇到第一个 0 字节时返回, 并试图把接下来的 ASCII 字符作为机器码执行, 这同样会使机器崩溃。这些都是 HLA 的 `stdout.put` 代码不在代码流中传递参数的原因。尽管有问题存在, 代码流仍是用于传递不变数值参数的有效位置。

5.17.3 在栈中传递参数

多数高级语言都使用栈传递参数, 因为这种方法相当有效。默认情况下, HLA 也使用栈传递参数。虽然使用栈传递参数与使用寄存器相比在效率上稍有欠缺, 但寄存器集合非常有限, 因此只能通过寄存器传递少量数值或引用参数。而栈可以在没有任何困难的情况下传递大量参数数据, 这也是多数程序都使用栈传递参数的主要原因。

通常情况下, 对于用高级过程调用语法所指定的参数, HLA 使用栈传递它们。例如, 假定用下面的方法定义前面提到的 `strfill`:

```
procedure strfill( s:string; chr:char );
```

形如 `strfill(s, ' ');` 的调用将把 `s` 的值(是一个地址)和一个空格字符放在 80x86 的栈顶部。当通过这种形式指定对 `strfill` 的调用时, HLA 会自动替您把参数推入栈, 而不用自己去做相应的操作。当然, 如果选择自己做, HLA 也允许在调用之前手动地把参数推入栈顶。

若要使用手动方法通过栈传递参数, 应该在子例程调用之前把它们推入栈。这样子例程就可以从栈存储器中读取数据并对它们进行相应操作。考虑下面的 HLA 过程调用:

¹⁹ 尤其是参数列表经常改变的时候。

CallProc(i,j,k);

HLA 将按照参数在参数列表中出现的顺序依次把它们推入栈²⁰。因此,HLA 为这个子例程调用(假设通过值传递参数)发出的 80x86 代码为:

```
push( i );
push( j );
push( k );
call CallProc;
```

在过程 CallProc 的入口点,80x86 的栈如图 5-7 所示。

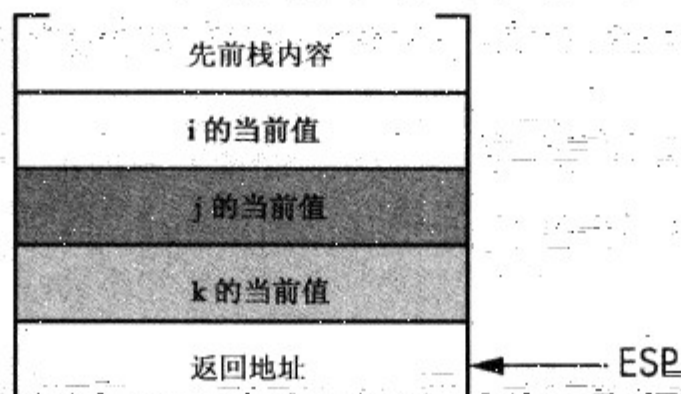


图 5-7 过程 CallProc 入口点处的栈布局

通过下面代码段所演示的方法把数据移出栈,这样就可以获得栈顶参数的访问权。

```
// Note: To extract parameters off the stack by popping, it is very important
// to specify both the @nodisplay and @noframe procedure options.
static
    RtnAdrs: dword;
    p1Parm: dword;
    p2Parm: dword;
    p3Parm: dword;

procedure CallProc( p1:dword; p2:dword; p3:dword );@ nodisplay; @noframe;
begin CallProc;

    pop( RtnAdrs );
    pop( p3Parm );
    pop( p2Parm );
    pop( p1Parm );
    push( RtnAdrs );

    .
    .
    .

    ret();

end CallProc;
```

²⁰ 当然,这里假定不再另外指示 HLA。告知 HLA 反转栈顶的参数顺序是可能的。请参见本书的电子版本了解更多细节。

正如从代码中所看到的，它首先把返回地址从栈顶弹出并保存在变量 `RtnAdrs` 中；然后依次弹出(按照与入栈相反的顺序)参数 `p1`、`p2` 和 `p3` 的值；最后，把返回地址放回栈顶(这样 `ret` 指令才能正确执行)。在过程 `CallProc` 内，通过访问变量 `p1Parm`、`p2Parm` 和 `p3Parm` 使用参数 `p1`、`p2` 和 `p3` 的值。

然而，有一种更好的访问过程参数的方法。如果过程包含标准入口和出口序列，那么可以通过寄存器 `EBP` 的变址在活动记录中直接访问参数的值。考虑使用下面声明的过程 `CallProc` 的活动记录布局：

```

procedure CallProc( p1:dword; p2:dword; p3:dword ); @nodisplay; @noframe;
begin CallProc;

    push( ebp );      // This is the standard entry sequence.
    mov( esp, ebp ); // Get base address of A.R. into ebp.
    .
    .
    .

```

我们看一下在过程 `CallProc` 内，`mov(esp,ebp)` 刚被执行完成时的栈情况。假定已经把 3 个双字的参数放置在栈顶，则栈将如图 5-8 所示。

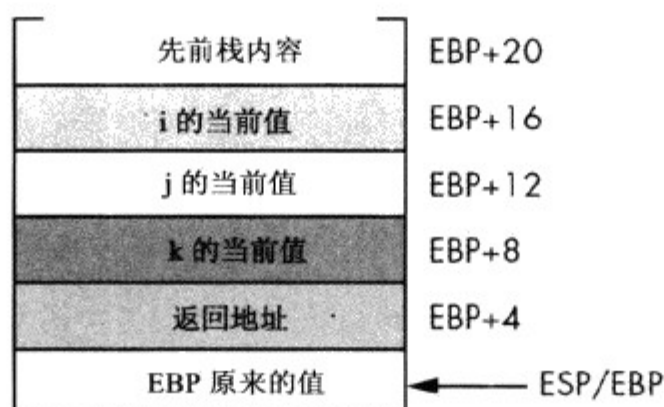


图 5-8 标准入口序列执行后过程 `CallProc` 的活动记录

现在可以通过寄存器 `EBP` 的变址访问参数：

```

mov( [ebp+16], eax ); // Accesses the first parameter.
mov( [ebp+12], ebx ); // Accesses the second parameter.
mov( [ebp+8], ecx );  // Accesses the third parameter.

```

当然，与局部变量类似，不会真正使用这种方法访问参数。可以使用形参名称(`p1`、`p2` 和 `p3`)，这样 HLA 会把它们替换为相应的`[ebp+位移量]`内存地址。尽管不应该使用地址表达式`[ebp+12]`实际访问参数，但对于过程中的参数，理解二者之间的关系是很重要的。

经常出现在活动记录中的其他项是过程保存的寄存器数值。在过程中，对寄存器进行保存的最合理位置是在紧跟标准入口序列的代码中。在一个标准的 HLA 过程里(没有指定 `@noframe` 选项)，这意味着对寄存器进行保存的代码最先出现在过程体中。同样地，恢复这些寄存器值的代码应该出现在过程的 `end` 子句之前²¹。

²¹ 需要注意的是，如果使用 `exit` 语句退出一个过程，那必须复制将寄存器数值弹出栈的代码，并把它放置在 `exit` 子句之前。这个示例说明了可维护性的困难，同时也说明为什么程序中只能有一个出口点。

1. 访问栈中的值参数

访问通过值传递的参数与访问局部 `var` 对象相同。只要在形参列表中对参数进行声明,并且过程在程序入口点执行了标准入口序列,这时只须指定参数名称就可以引用参数的数值。如程序清单 5-13 所提供的示例程序,主程序通过值向过程传递参数,过程对这个参数进行了访问。

程序清单 5-13 值参数示例

```
program AccessingValueParameters;
#include( "stdlib.hhf" )

procedure ValueParm( theParameter: uns32 ); @nodisplay;
begin ValueParm;
    mov( theParameter, eax );
    add( 2, eax );
    stdout.put
    (
        "theParameter + 2 = ",
        {type uns32}eax,
        nl
    );
end ValueParm;

begin AccessingValueParameters;
    ValueParm( 10 );
    ValueParm( 135 );
end AccessingValueParameters;
```

虽然可以在程序中使用匿名地址 `[EBP+8]` 访问 `theParameter` 的值,但并没有很好的理由说明这样做的原因。如果使用 HLA 的高级语言语法声明了参数列表,则可以通过在过程内部指定名称的方法访问值参数。

2. 在栈中传递值参数

如程序清单 5-13 所示,向一个过程传递值参数非常容易。像在高级语言中那样,只须在实参列表中指定相应数值。实际情况会更复杂一些。如果要传递的是常量、寄存器或变量数值,那么传递值参数是很容易的。若要传递某些表达式的结果,情况会相对复杂。本小节将介绍值参数的不同传递方法。

当然,不必使用 HLA 的高级语法向过程传递值参数。可以自己把这些数值推入栈中。因为很多时候采用手动传递参数的方法会更方便、更高效。我们就从如何这样做开始讲述。

从本章前面的内容可看到,当在栈中传递参数时,是按照它们在形参列表中出现的顺序推入栈的(从左至右)。通过值传递参数时,应该把实参的数值放在栈中。程序清单 5-14 演示了如何这样做。

程序清单 5-14 在栈中手动传递参数

```

program ManuallyPassingValueParameters;
#include( "stdlib.hhf" )

    procedure ThreeValueParms( p1:uns32; p2:uns32; p3:uns32 ); @nodisplay;
    begin ThreeValueParms;

        mov( p1, eax );
        add( p2, eax );
        add( p3, eax );
        stdout.put
        (
            "p1 + p2 + p3 = ",
            (type uns32 eax),
            nl
        );

    end ThreeValueParms;

static
    SecondParmValue:uns32 := 25;

begin ManuallyPassingValueParameters;

    pushd( 10 );           // Value associated with p1
    pushd( SecondParmValue ); // Value associated with p2
    pushd( 15 );           // Value associated with p3
    call ThreeValueParms;

end ManuallyPassingValueParameters;

```

注意，如果像示例中那样采用手动方法向栈上推入参数，则必须使用 `call` 指令调用过程。若试图使用形如 `ThreeValueParms()` 的过程调用，HLA 将报出参数列表不匹配的错误。HLA 并不知道参数已经被手动推入栈中(HLA 会认为那些操作是对其他数据的保存)。

一般来说，如果实参是一个常量、寄存器或变量的话，没有必要手动把它们推入栈。HLA 的高级语法会为您处理这些参数。然而，在个别情况下，HLA 的高级语法却不能做到这样。一个典型的示例是把算术表达式的结果作为值参数传递。因为在 HLA 中不存在运行时算术表达式，因此必须手动计算表达式的结果并传递它的值。有两种可行的方法：计算表达式的结果并手动地把结果推入栈，或者把表达式的结果保存在寄存器中，然后把这个寄存器作为参数传递给过程。程序清单 5-15 中的程序演示了这两种机制。

程序清单 5-15 把算术表达式的结果作为参数传递

```

program PassingExpressions;
#include( "stdlib.hhf" )

    procedure ExprParm( exprValue:uns32 ); @nodisplay;
    begin ExprParm;

        stdout.put( "exprValue = ", exprValue, nl );
    end ExprParm;

```

```

    end ExprParm;

static
    Operand1:uns32 := 5;
    Operand2:uns32 := 20;
begin PassingExpressions;

    // ExprParm( Operand1 + Operand2 );
    //
    // Method one: Compute the sum and manually
    // push the sum onto the stack.
    mov( Operand1, eax );
    add( Operand2, eax );
    push( eax );
    call ExprParm;

    // Method two: Compute the sum in a register and
    // pass the register using the HLA high-level
    // language syntax.
    mov( Operand1, eax );
    add( Operand2, eax );
    ExprParm( eax );
end PassingExpressions;

```

本节到目前为止所出现的示例都做了一个很重要的假设：所传递的参数是双字数据。如果传递的参数不是4字节对象，调用序列会做相应改变。当传递非4字节宽度的对象时，HLA会产生效率相对较低的代码，所以如果要想代码具有更快的执行速度，采用手动方式传递这样的数据对象是很好的方法。

HLA要求所有值参数的数据宽度都是4字节的整数倍²²。如果要传递的对象不到4字节宽度，那么HLA要求使用额外的字节填充参数数据，所以所能传递的数据对象至少是4字节宽度的。对于超过4字节的参数，必须保证传递的参数数值是4字节的整数倍，必要时应在数据对象的高位地址端填充额外的字节保证这一点。

考虑下面的过程原型：

```

procedure OneByteParm( b:byte );

```

这个过程的活动记录如图5-9所示。

²² 在过程中，这种要求只针对使用HLA高级语言语法声明并访问参数的时候。所以，如果采用手动方式将参数推入栈，并在过程内使用形如[ebp+8]的寻址方式对参数进行访问，那么可以选择传递任何数据宽度的对象。当然，要注意的是大多数操作系统都认为栈是双字对齐的，因此所传递的参数应该是4字节宽度的整数倍。

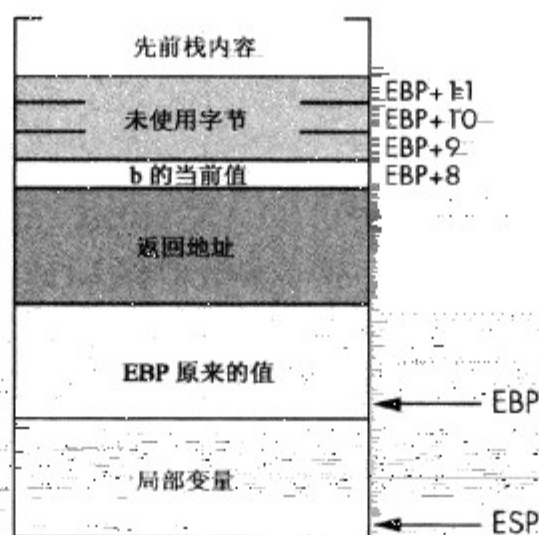


图 5-9 OneByteParm 的活动记录

如您所见,栈中共有 4 个字节与参数 *b* 相关联,但其中只有 1 个字节包含有效数据(低位字节)。其余 3 个字节只是用来填充的,过程会忽略这些字节。特别地,不要假定这些额外字节包含 0 或其他数值。HLA 自动生成的代码是否把栈中的额外字节保存为 0 取决于所传递参数的类型。

当向过程传递一个字节参数时,HLA 会自动生成代码向栈推入 4 个字节的数据。由于 HLA 的参数传递机制总是保证不破坏任何寄存器或其他数值,所以在传送一个字节的参数时,HLA 经常会产生多于实际所需的代码。例如,如果把寄存器 *AL* 作为一个字节参数来传递,HLA 产生的代码将会把寄存器 *EAX* 推入栈。这条简单的 *push* 指令非常有效地把 *AL* 作为 4 字节参数对象传递。另一方面,如果想把寄存器 *AH* 作为字节参数传递,把 *EAX* 推入栈将不起作用,因为这样会把 *AH* 中的数值放在图 5-9 所示活动记录中偏移量为 *EBP+9* 的存储单元中,而过程希望这个数值在偏移量为 *EBP+8* 的存储单元中。所以简单地把 *EAX* 推入栈并不能达到预期的效果。当想把寄存器 *AH*、*BH*、*CH* 或 *DH* 作为字节参数传递时,HLA 会生成如下的代码:

```
sub( 4, esp );      // Make room for the parameter on the stack.
mov( ah, [esp] );   // Store ah into the L.O. byte of the parameter.
```

如您所见,把 *H* 寄存器作为字节参数传递与传递 *L* 寄存器相比是低效的。因此,当把 8 位寄存器作为参数传递时,应尽可能地使用 *L* 寄存器²³。需要注意,对于效率上的差异我们无能为力,即使用手动的方法传递参数也是这样。

如果要传递的字节参数是变量而不是寄存器,HLA 会产生错误的代码。例如,假定对 *OneByteParm* 的调用如下所示:

```
OneByteParm( uns8Var );
```

对于这个调用,HLA 将产生类似于下面的代码传递这个字节参数:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
pop( eax );
```

23 或者更好地,如果是自己编写过程,则可以把参数直接传递到寄存器中。

很显然,对于传递一个字节到栈来说,这段代码过长。HLA之所以生成这样的代码是因为:
①它要确保不破坏任何寄存器;②它并不知道在所分配的存储空间中 *uns8Var* 是否是最后一个变量。在不必强制这两个约束的时候,可以自己产生更好的代码。

如果有空闲的 32 位寄存器可以使用(特别是 EAX、EBX、ECX 或 EDX 之一),那么可以使用两条指令向栈传递一个字节参数。把要传递的字节数值移入(或通过零/符号扩展移入)寄存器,然后把寄存器推入栈。当 EAX 可用时,利用上述方法对 *OneByteParm* 的调用序列如下所示:

```
mov( uns8Var, al );
push( eax );
call OneByteParm;
```

如果只有 ESI 或 EDI 可用,可以使用下面的代码:

```
movzx( uns8Var, esi );
push( esi );
call OneByteParm;
```

还可以强制一个字节变量为双字对象,使用这个技巧可以只用一条 *push* 指令传递参数,例如:

```
push( (type dword uns8Var) );
call OneByteParm;
```

最后这个示例的效率很高。注意,它把存储器中紧跟 *uns8Var* 后的三个字节作为填充数据,而不管这些数据是什么。HLA 不使用这个技巧,因为使用这种方法在某些情况(可能性很小)下会导致程序执行失败。如果碰巧对象 *uns8Var* 是存储器所给页的最后一个字节,而存储器的下一页是不可访问的,这时, *push* 指令将导致存储器访问异常。为了确保安全,HLA 的编译器不采用这种方法。当然,如果能够确定采用这种方式传递的实参不是所声明静态段的最后一个变量,那么使用这种技术的代码可以侥幸成功运行。对于一个字节对象,它出现在栈可访问地址最末尾的情况几乎是不可能的,因此使用这种方法处理 *var* 对象是安全的。

当向栈传递字参数时,同样要确保包含填充字节,使得每一个参数都占用 4 字节整数倍的空间。可以采用与传递字节参数相同的方法传递字参数,当然,差别在于用两个字节的的有效数据代替一个字节。例如,可以使用下面两种方法之一把一个字对象 *w* 传递给过程 *OneWordParm*:

```
mov( w, ax );
push( eax );
call OneWordParm;

push( (type dword w) );
call OneWordParm;
```

当通过值向栈传递大规模数据对象时(如记录和数组),不必保证对象的每个元素或字段都占用 4 字节整数倍的空间,而只须确保整个数据结构在栈中占用的空间为 4 字节的整数倍。例如,如果有一个数组由 10 个元素组成,每个元素占用 3 字节,那么整个数组只需 2 个字节的填充数据(10*3 是 30 字节,它不能被 4 整除;但 10*3+2 是 32,可以被 4 整除)。HLA 可以很好地处理通过值向一个过程传递大规模数据对象的情况。所以,对于大规模数据对象,如果没有特殊要求,

应该使用 HLA 的高级语言过程调用语法完成对它们的传递。当然，若想提高操作效率，应该尽量避免通过值传递大规模数据结构。

默认情况下，HLA 产生代码向过程传递参数时会保证不破坏任何寄存器的值，但有时这样的保证是不必要的。例如，如果把函数结果返回到 EAX 中，而不是通过 EAX 向某个过程传递参数，这时就不必在过程的入口点保存 EAX 的值。相对于产生下面的代码传递一个字节参数：

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
pop( eax );
```

如果知道可以使用 EAX(或其他寄存器)，HLA 可以产生更好的代码：

```
mov( uns8Var, al );
push( eax );
```

可以使用过程选项 `@use`，告知 HLA 在传递参数时可以改变某个寄存器的值，从而改善所生成的代码。这个选项的语法形式为：

```
@use reg32;
```

其中，操作数 `reg32` 可以是 EAX、EBX、ECX、EDX、ESI 或 EDI。如果这个寄存器是 EAX、EBX、ECX 或 EDX 之一，则会获得最佳结果。特别要说明的是，注意不能在这个选项里指定 EBP 或 ESP(因为过程已经使用了这些寄存器)。

过程选项 `@use` 告知 HLA 可以修改指定为操作数的寄存器的数值。这时候，若不保存这个寄存器的值会产生更高效的代码，HLA 会自动选择这样做。例如，如果选项 `@use eax;` 包含在前面提到的过程 `OneByteParm` 中，HLA 会立即产生上面的 2 条指令，而不是保存 EAX 的 5 条指令序列。

指定过程选项 `@use` 时要当心，特别注意不要使用选项 `@use` 指定的寄存器传递任何参数(因为这样 HLA 很可能会把这个参数数值破坏掉)。同样地，必须确保过程对这个寄存器数值的改变不会产生问题。如前所述，当过程通过 EAX 返回一个函数结果时，EAX 是作为 `@use` 寄存器的最佳选择(因为调用者不需要过程对寄存器 EAX 进行保存)。

如果过程有 `forward` 或 `external` 声明(参见本章 5.24 节)，则选项 `@use` 必须出现在 `forward` 或 `external` 的定义里，而不是在实际的过程声明中。如果没有这样的过程原型出现，必须把选项 `@use` 附加在过程声明中。

这里有一个示例：

```
procedure OneByteParm( b:byte ); @nodisplay; @use EAX;
begin OneByteParm;

    << Do something with b. >>

end OneByteParm;
```

```
static
    byteVar:byte;

OneByteParm( byteVar );
```

这个对 OneByteParm 的调用会产生下面的指令:

```
mov( uns8Var, al );
push( eax );
call OneByteParm;
```

3. 访问栈中的引用参数

由于 HLA 是为引用参数传递地址, 在过程中对引用参数的访问要比值参数的访问相对困难, 因为必须解除指针对引用参数的引用。遗憾的是, 关于过程声明和调用的 HLA 高级语法没有(也不能)把这些细节抽象出来, 您必须自己手动解除指针引用。本小节将回顾如何进行这样的操作。

程序清单 5-16 中的过程 RefParm 有一个通过引用传递的参数。通过引用传递的参数总是一个指向某一对象的指针, 这个对象的类型由参数声明指定。因此, theParameter 实际上是一个“指向 uns32 的指针”类型的对象, 而不是一个 uns32 数值。为了访问与 theParameter 相关联的数值, 代码必须把双字地址读取到 32 位寄存器, 同时间接地对数据进行访问。程序清单 5-16 中的指令 mov(theParameter,eax); 把指针读取到寄存器 EAX 中, 然后过程 RefParm 使用寻址方式[eax]访问 theParameter 的实际数值。

程序清单 5-16 访问引用参数

```
program AccessingReferenceParameters;
#include( "stdlib.hhf" )

procedure RefParm( var theParameter: uns32 ); @nodisplay;
begin RefParm;

    // Add 2 directly to the parameter-passed by
    // reference to this procedure.

    mov( theParameter,eax );
    add( 2,(type uns32 [eax]) );

    // Fetch the value of the reference parameter
    // and print its value.

    mov( [eax], eax );
    stdout.put
    (
        "theParameter now equals ",
        (type uns32 eax),
        nl
    );
end RefParm;
```

```

static
    p1: uns32 := 10;
    p2: uns32 := 15;

begin AccessingReferenceParameters;

    RefParm( p1 );
    RefParm( p2 );

    stdout.put( "On return, p1=", p1, "and p2=", p2, nl );

end AccessingReferenceParameters;

```

因为该过程访问的是实参的数据,对该数据进行加2操作影响的是主程序传递给过程 **RefParm** 的变量数值。当然,这并不奇怪,因为这是通过引用传递参数的标准语义。

如您所见,对于较小的对象来说,对引用参数的访问效率会低于值参数的访问,因为需要一条额外的指令把地址读取到一个 32 位指针寄存器中(必须预留一个 32 位的寄存器)。如果经常访问引用参数,这些额外的指令累加起来会降低整个程序的效率。此外,还很容易产生忘记解除引用参数的引用,并在运算中使用该数值的地址(尤其是向过程传递双字参数的情况下更容易出现这种情况,像上面示例中的参数 **uns32**)。因此,除非真的需要改变实参的值,否则应该通过值向过程传递小对象。

使用引用参数传递像数组和记录这样的大规模数据对象是高效的。若通过值传递这样的数据对象,调用代码必须对实参进行复制。如果实参是大规模的数据对象,复制的过程将非常低效。但计算大规模数据对象的地址与计算小标量对象的地址相比,效率相当,所以通过引用传递大规模数据对象没有效率损失。在过程内部必须解除指针引用来访问数据对象,但与复制大规模数据对象相比,这种间接的效率损失是很小的。程序清单 5-17 中的程序演示了如何通过引用传递参数,从而初始化一个记录数组。

程序清单 5-17 通过引用传递一个记录数组

```

program accessingRefArrayParameters;
#include( "stdlib.hhf" )

const
    NumElements := 64;

type
    Pt: record
        x:uns8;
        y:uns8;
    endrecord;

    Pts: Pt[NumElements];

procedure RefArrayParm( var ptArray: Pts ); @nodisplay;
begin RefArrayParm;

    push( eax );
    push( ecx );

```

```

push( edx );

mov( ptArray, edx );    // Get address of parameter into edx.

for( mov( 0, ecx ); ecx < NumElements; inc( ecx ) ) do

    // For each element of the array, set the x field
    // to (ecx div 8) and set the y field to (ecx mod 8).

    mov( cl, al );
    shr( 3, al );        // ecx div 8.
    mov( al, (type Pt [edx+ecx*2]).x );

    mov( cl, al );
    and( %111, al );     // ecx mod 8.
    mov( al, (type Pt [edx+ecx*2]).y );

endfor;

pop( edx );
pop( ecx );
pop( eax );

end RefArrayParm;

static
    MyPts: Pts;

begin accessingRefArrayParameters;

    // Initialize the elements of the array.

    RefArrayParm( MyPts );

    // Display the elements of the array.

    for( mov( 0, ebx ); ebx < NumElements; inc( ebx ) ) do

        stdout.put

        (
            "RefArrayParm[",
            ( type uns32 ebx ):2,
            "].x=",
            MyPts.x[ ebx*2 ],
            "RefArrayParm[",
            ( type uns32 ebx ):2,
            "].y=",
            MyPts.y[ ebx*2 ],
            nl
        );

    endfor;

end accessingRefArrayParameters;

```

从这个示例看出，通过引用传递大规模数据对象是高效的。除了过程中始终占用寄存器

EDX, 并增加了一条为 EDX 加载引用参数地址的指令外, 与通过值传递参数的同一过程相比, 过程 `RefArrayParm` 并不需要其他更多的指令。

4. 在栈中传递引用参数

HLA 的高级语法通常使得传递引用参数成为轻而易举的事情。您所要做的只是在过程的参数列表中指定想要传递的实参名称, HLA 将自动生成代码计算所指定实参的地址, 并把它推入栈中。然而, 像 HLA 为值参数发出代码一样, HLA 生成的用于在栈中传递实参地址的代码可能并不是最高效的。因此, 要想使代码更快捷, 您可能会选择手动编写向过程传递引用参数的代码。本小节将讨论如何这样做。

无论何时, 当把静态对象作为引用参数传递时, HLA 会生成很高效的代码向过程传递参数的地址。考虑下面的代码段:

```
procedure HasRefParm( var d:dword );
```

```

    .
    .
    .

```

```
    static
```

```
        FourBytes:dword;
```

```
    var
```

```
        v: dword[2];
```

```

    .
    .
    .

```

```
HasRefParm( FourBytes );
```

```

    .
    .
    .

```

对于过程 `HasRefParm` 的调用, HLA 生成下面的指令序列:

```
pushed( &FourBytes );
call HasRefParm;
```

如果是在栈上传递引用参数, 也许已经无法比这做得更好。所以, 如果把静态对象作为引用参数传递, HLA 会生成相当不错的代码, 应该坚持使用过程调用的高级语法。

然而, 当把自动(`var`)对象或索引变量作为引用参数传递时, HLA 必须在运行时计算对象的地址。这时通常需要使用 `lea` 指令。遗憾的是, `lea` 指令需要使用一个 32 位的寄存器, 同时 HLA 又要保证在自动生成代码时不破坏任何寄存器的值²⁴。因此, 当 HLA 通过 `lea` 指令计算引用参数的地址时, 必须保存所使用的寄存器的值。下面的示例显示了 HLA 实际发出的代码:

```
// Call to the HasRefParm procedure:

        HasRefParm( v[ebx*4] );
```

²⁴ 这并不完全正确。您将在第 12 章中看到例外情况。同样是使用过程选项 `@use` 来告知 HLA 可以改变某个寄存器的值。

```
// HLA actually emits the following code for the above call:
```

```
push( eax );
push( eax );
lea( eax, v[ebx*4] );
mov( eax, [esp+4] );
pop( eax );
call HasRefParm;
```

如您所见, 这样的代码过长, 特别是存在可用的 32 位寄存器, 而又不需要保存寄存器的值时。下面是在 EAX 可用的情况下给出的更好的代码序列:

```
lea( eax, v[ebx*4] );
push( eax );
call HasRefParm;
```

切记, 当通过引用传递一个实参时, 必须计算这个对象的地址并把地址推入栈。对于简单的静态变量, 可以使用取址操作符(&), 这样很容易就可以计算出对象的地址并把它推入栈。然而, 对于索引或自动对象, 必须使用指令 `lea` 计算对象的地址。下面用过程 `HasRefParm` 说明这种方法:

```
static
    i:      int32;
    Ary:    int32[16];
    iptr:    pointer to int32 := &i;

var
    v:      int32;
    AV:     int32[10];
    vptr:    pointer to int32;

    .
    .
    .

    lea( eax, v );
    mov( eax, vptr );

    .
    .
    .

// HasRefParm( i );

    push( &i );          // Simple static object, so just use &.
    call HasRefParm;

// HasRefParm( Ary[ebx] );    // Pass element of Ary by reference.

    lea( eax, Ary[ ebx*4 ] ); // Must use lea for indexed addresses.
    push( eax );
    call HasRefParm;

// HasRefParm( *iptr );    --- Pass object pointed at by iptr

    push( iptr );          // Pass address(iptr's value) on stack.
    call HasRefParm;
```

```
// HasRefParm( v );

    lea( eax, v );           // Must use lea to compute the address
    push( eax );            // of automatic vars passed on stack.
    call HasRefParm;

// HasRefParm( AV[ esi ] ); -- Pass element of AV by reference.

    lea( eax, AV[ esi*4 ] ); // Must use lea to compute address of the
    push( eax );            // desired element.
    call HasRefParm;

// HasRefParm( *vptr ); -- Pass address held by vptr...

    push( vptr );           // Just pass vptr's value as the specified
    call HasRefParm;        // address.
```

如果有额外的寄存器处于空闲状态，可以通知 HLA 使用这个寄存器计算引用参数的地址(不必发出对这个寄存器的数值进行保存的代码)。选项 `@use` 会告知 HLA 可以使用指定的寄存器而不必保存它的值。与值参数部分提到的相同，这个过程选项的语法为：

```
@use reg32;
```

其中，`reg32` 可以是 EAX、EBX、ECX、EDX、ESI 或 EDI 中任何一个。因为引用参数总是传递 32 位数值，所以这些寄存器对于 HLA 来说是等价的(不同于值参数，后者更倾向于使用寄存器 EAX、EBX、ECX 或 EDX)。如果过程没有在寄存器 EAX 中传递参数而是在其中返回了一个函数结果，那么最好选择 EAX，否则使用当前没有被使用的其他寄存器也可以。

在前面的示例中使用了选项 `@use eax;`，HLA 将生成更简短的代码。那些用于保存寄存器 EAX 值的额外指令将不必全部被生成。这将使您的代码更高效，特别当程序含有许多通过引用传递的参数或多次调用含有引用参数的过程时。

5. 形参作为实参传递

前面两小节的示例介绍了如何通过值或引用把静态变量和自动变量作为参数传递给一个过程。有一种情形这些示例将不能恰当地处理：当在一个过程中传递某一形参，而这个参数被作为另一过程的实参时。下面的示例说明了通过值传递和引用传递会发生的不同情况：

```
procedure p1( val v:dword; var r:dword );
begin p1;
    .
    .
    .
end p1;

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );           // (1) First call to p1
    p1( r2, v2 );           // (2) Second call to p1
```

```
end p2;
```

在标注为(1)的语句中,过程 p2 调用 p1 并把它两个形参作为 p1 的参数传递给 p1。注意,该代码通过值传递两个过程的第一个参数,而通过引用传递两个过程的第二个参数。因此,在语句(1)中,程序通过值方式把参数 v2 传入 p2 后,再通过值方式把它传递给 p1;同样地,程序通过引用方式把 r2 传入后,再次通过引用方式把它传递给 p1。

由于 p2 的调用者是通过值传递 v2 的,同时 p2 也通过值把它传递给 p1,所以代码要做的只是把 v2 的数值进行复制并把它传递给 p1。完成这个功能的代码只需一条简单的 push 指令,例如:

```
push( v2 );
<< Code to handle r2 >>
call p1;
```

正如您所见到的,这段代码与通过值传递一个自动变量的情况相同。事实上,向过程传递值参数的代码与向另一个过程传递局部自动变量的代码相同。

在前面的语句(1)中传递 r2 时需要多思考一下,您没有像传递一个值参数或自动变量那样使用指令 lea 获得 r2 的地址。当把 r2 传递给 p1 时,代码编写者希望 r 形参中包含变量的地址,这个地址就是 p2 的调用者传递给 p2 的。换句话说,这就意味着 p2 必须把 r2 的实参地址传递给 p1。由于参数 r2 是一个包含相应实参地址的双字数值,也就是说代码必须把 r2 的双字数值传递给 p1。对应语句(1)的完整代码如下所示:

```
push( v2 );      // Pass the value passed in through v2 to p1.
push( r2 );      // Pass the address passed in through r2 to p1.
call p1;
```

从这个示例中注意到,当把一个形式引用参数(r2)作为实际引用参数(r)传递时,不涉及获取形参(r2)地址的问题。因为 p2 的调用者已经做了这部分工作, p2 只须简单地把这个地址传递给 p1。

在上面示例对 p1 的第二个调用即语句(2)中,代码交换了实参的位置,这样对 p1 的调用将通过值传递 r2,通过引用传递 v2。明确地说, p1 希望 p2 把与 r2 相关联双字对象的数值传递给它,同样地,它希望 p2 把与 v2 相关联数值的地址传递给它。

要传递与 r2 相关联对象的数值,代码必须解除与 r2 关联的指针,并直接地传递这个数值。下面是 HLA 自动生成的用于把 r2 作为第一个参数传递给语句(2)中 p1 的代码:

```
sub( 4, esp );      // Make room on stack for parameter.
push( eax );        // Preserve eax's value.
mov( r2, eax );     // Get address-of object passed in to p2.
mov( [eax], eax );  // Dereference to get the value of this object.
mov( eax, [esp+4] ); // Put value of parameter into its location on stack.
pop( eax );         // Restore original eax value.
```

通常情况下,HLA 会产生比完成功能所需更多的代码,因为它要保证不破坏寄存器 EAX 中的值(可使用过程选项 @use 告知 HLA 可以使用寄存器 EAX 的值,这样就能够减少它生成的代码)。在这段代码序列中,如果有其他可用的寄存器,则可以写出更高效的代码。如果 EAX 没有被使用,可以把上面的代码简写为:

```

mov( r2, eax );      // Get the pointer to the actual object
pushd( [eax] );      // Push the value of the object onto the stack

```

因为可以像处理局部(自动)变量一样处理值参数,所以可以使用与在 p2 内向 p1 传递局部变量相同的代码向 p1 传递引用参数 v2。特别是可以使用 lea 指令计算 v2 中数值的地址。HLA 自动为语句(2)生成的代码保存所有的寄存器,具体形式如下(与通过引用传递自动变量相同):

```

push( eax );         // Make room for the parameter.
push( eax );         // Preserve eax's value.
lea( eax, v2 );      // Compute address of v2's value.
mov( eax, [esp+4] ); // Store away address as parameter value.
pop( eax );          // Restore eax's value.

```

当然,如果有其他可用的寄存器,可以对这段代码进行优化。下面是与语句(2)一致的完整代码:

```

mov( r2, eax );      // Get the pointer to the actual object.
pushd( [ eax] );     // Push the value of the object onto the stack.
lea( eax, v2 );      // Push the address of v2 onto the stack.
push( eax );
call p1;

```

6. HLA 的混合参数传递功能

与控制结构类似,HLA 为过程调用提供了既使用方便又易于阅读的高级语言语法。然而,这种高级语言语法有时候会很低效,或不能提供所需的功能(例如,不能像在高级语言中那样指定一个算术表达式作为值参数)。HLA 使您通过书写低级(“纯粹”)汇编语言代码克服这种限制。遗憾的是,与使用高级语言语法的过程调用相比,低级代码更难读和难维护。此外,对于某些参数来说,HLA 完全可以生成近乎完美的代码,而只有一两个参数产生问题。幸运的是,HLA 为过程调用提供了混合语法,对于一个给定的实参,允许适当地同时使用高级和低级语法。这种方法使得可以在适当的时候使用高级语法,而对于一些使用 HLA 高级语言语法不能有效处理(或根本不能处理)的特殊参数则向下使用纯粹的汇编语言来传递。

在一个实参列表中(使用高级语言语法),如果遇到#{后面紧跟语句序列和结束符}#,HLA 会使用大括号之间的指令代替它通常会为那些参数生成的代码。例如,考虑下面的代码段:

```

procedure HybridCall( i:uns32; j:uns32 );
begin HybridCall;
    .
    .
    .
end HybridCall;
    .
    .
    .
    // Equivalent to HybridCall( 5, i+j );

HybridCall
(
    5,

```

```

    #{
        mov( i, eax );
        add( j, eax );
        push( eax );
    }#
);

```

上面对过程 `HybridCall` 的调用等价于下面“纯粹”的汇编语言代码：

```

pushd( 5 );
mov( i, eax );
add( j, eax );
push( eax );
call HybridCall;

```

作为第二个示例，考虑前一节提到的示例：

```

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 ); // (1)First call to p1
    p1( r2, v2 ); // (2)Second call to p1

end p2;

```

在这个示例中，HLA 为第二个对 `p1` 的调用产生非常普通的代码。如果效率在对这个过程调用的语境下是一个重要因素，同时具有空闲的寄存器可用，可以重写这段代码如下²⁵：

```

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 ); // (1)First call to p1
    p1 // (2)Second call to p1
    ( // This code assumes eax is free
        #{
            mov( r2, eax );
            pushd( [eax] );
        }#,
        #{
            lea( eax, v2 );
            push( eax );
        }#
    );
end p2;

```

需要注意的是，如果指定了选项 `@use reg`，就等于告诉 HLA 无论在何处对过程进行调用，指定的寄存器都是可用的。如果有这样一种情形，对过程的调用必须保存指定的寄存器，那么不能使用选项 `@use` 生成更好的代码。然而，可以使用混合参数传递机制根据情况对那些特殊调用进

25 当然，在本例中可以使用过程选项 `@use eax` 达到同样的效果。

行性能优化。

7. 寄存器参数与基于栈的参数的混合

可以把寄存器参数和标准(基于栈的)参数混合于同一个高级过程声明中, 例如:

```
procedure HasBothRegAndStack( var dest:dword in edi; count:un32 );
```

在创建活动记录的时候, HLA 会忽略在寄存器中传递的参数, 而只处理那些在栈中传递的参数。因此, 对过程 `HasBothRegAndStack` 的一次调用只在栈顶放入一个参数(`count`)。它将把参数 `dest` 传递到寄存器 `EDI` 中。当这个过程返回调用者时, 只从栈中移出 4 字节的参数数据。

注意, 当在寄存器中传递参数时, 要避免和过程选项 `@use` 指定的寄存器相同。在上例中, HLA 根本不会为参数 `dest` 产生任何代码(因为参数数值已经保存在 `EDI` 中)。如果已经指定了 `@use edi;`, 那么 HLA 认为可以改变寄存器 `EDI` 的值, 这将破坏寄存器 `EDI` 中的参数数值。当然, 在这个特定的示例中并不会发生(因为 HLA 没有使用寄存器传递类似于 `count` 的双字数值参数), 但要谨记这个问题的存在。

5.18 过程指针

80x86 的 `call` 指令提供 3 种基本形式: 通过过程名直接调用、通过一个 32 位通用寄存器间接调用、通过一个双字指针变量间接调用。`call` 指令支持下列(低级)语法:

```
call Procname;           // Direct call to procedure Procname(or Stmt label).
call( Reg32 );           // Indirect call to procedure whose address appears
                           // in the Reg32 general-purpose 32-bit register.
call( dwordVar 1);       // Indirect call to the procedure whose address appears
                           // in the dwordVar double word variable.
```

贯穿本章, 我们使用的都是第一种形式, 因此在这里不必进一步讨论。第二种形式, 寄存器间接调用, 通过保存在指定的 32 位寄存器中的地址调用过程。过程的地址是指过程内部执行的第一条指令的字节地址。应该记得, 在冯·诺依曼体系结构的机器中(如 80x86), 系统把机器指令连同其他数据一起保存在存储器中。CPU 在执行一条指令之前, 首先从存储器中把指令操作码取出。当执行寄存器间接调用指令时, 80x86 首先把返回地址推入栈, 然后通过寄存器中数值所指的地址开始读取下一个操作码字节(指令)。

上面第三种形式的调用指令, 通过存储器中的一个双字变量读取过程第一条指令的地址。虽然这条指令建议调用使用位移寻址方式, 其实任何合法的存储器寻址方式在这里都是可用的; 如 `call(procPtrTable[ebx*4]);` 是完全合法的, 这条语句从双字数组(`procPtrTable`)中读取一个双字, 同时调用由这个双字包含的地址所指向的过程。

HLA 像处理静态对象一样处理过程名。因此, 可以使用取址操作符(`&`)和过程名或使用 `lea` 指令计算过程的地址。例如, `&Procname` 就是过程 `Procname` 的第一条指令的地址。所以下面的 3 段代码序列都是对过程 `Procname` 的调用:

```

    call Procname;
    .
    .
    .
    mov( &Procname, eax );
    call( eax );
    .
    .
    .
    lea( eax, Procname );
    call( eax );

```

因为过程的地址正好是一个 32 位数据对象，所以可以把这个地址保存在一个双字变量中。事实上，可以使用下列代码用一个过程的地址初始化双字变量：

```

    procedure p;
    begin p;
    end p;
    .
    .
    .
    static
        ptrToP: dword := &p;
    .
    .
    .
    call( ptrToP );    // Calls the p procedure if ptrToP has not changed.

```

由于过程指针的使用在汇编语言程序中经常发生，所以 HLA 提供了一个专门的语法来声明过程指针变量，并且可以通过这种指针变量间接地调用过程。在 HLA 的程序中，可以使用如下方法声明一个过程指针：

```

static
    procPtr: procedure;

```

注意，这个语法使用关键字 `procedure` 作为数据类型。它在某一个变量声明段(`static`、`readonly`、`storage` 或 `var`)中紧跟在变量名和一个冒号之后。这个声明准确地为变量 `procPtr` 留出 4 字节的存储空间。可以使用下列两种形式之一调用 `procPtr` 中地址所指向的过程：

```

call( procPtr );    // Low level syntax
procPtr();          // High level language syntax

```

注意，间接过程调用的高级语法与直接过程调用的高级语法相同。HLA 可以通过标识符的类型判断是使用直接调用还是间接调用。如果已经指定了一个变量名称，HLA 假定它需要使用一个间接调用；如果指定一个过程名称，HLA 使用直接调用。

类似于所有的指针对象，除非已经用一个正确的地址对变量进行了初始化，否则不能试图通过一个指针变量间接调用一个过程。有两种方式可以初始化一个过程指针变量：`static` 和 `readonly` 对象允许使用初始值设定项进行初始化，或者计算例程的地址(32 位数值)，并在运行时把这个 32

位地址直接保存到过程指针中。下面的代码段说明了用于初始化一个过程指针的两种方式:

```
static
ProcPointer: procedure := &p; // Initialize ProcPointer with the address of p.
.
.
.
ProcPointer(); // First invocation calls p.
.
mov( &q, ProcPointer ); // Reload ProcPointer with the address of q.
.
.
.
ProcPointer(); // This invocation calls the q procedure.
```

过程指针变量声明还允许声明参数。必须使用如下方法声明一个带有参数的过程指针:

```
static
p: procedure( i: int32; c: char );
```

这个声明规定 *p* 是一个包含过程地址的 32 位指针, 并且这个过程需要两个参数。如果您愿意, 也可以用某个过程的地址静态地初始化这个变量 *p*, 例如:

```
static
p: procedure( i: int32; c: char ) := &SomeProcedure;
```

需要注意的是, 过程 *SomeProcedure* 的参数列表必须与 *p* 的参数列表完全匹配(也就是两个数值参数, 第一个是 *int32* 类型参数, 第二个是 *char* 类型参数)。可以使用下面序列之一间接地调用这个过程:

```
push( Value_for_i );
push( Value_for_c );
call( p );
```

或

```
p( Value_for_i, Value_for_c );
```

间接调用的高级语言语法具有与直接过程调用的高级语言语法相同的特性和约束。唯一的差别在于 HLA 生成的 *call* 指令在调用序列的末尾。

虽然本节的所有示例都使用 *static* 变量声明, 但不要认为只能在 *static* 或其他变量声明段中声明过程指针。也可以在 *type* 段中声明过程指针类型, 同时还可以把过程指针声明为一个记录或一个联合的字段。假设在 *type* 段中为一个过程指针创建了一个类型名, 这时还可以创建过程指针数组。下面的代码段演示了这种声明:

```
type
pptr: procedure;
prec: record
```



```

        p:pptr;
        <<Other fields>>
    endrecord;

static
    p1:pptr;
    p2:pptr[2];
    p3:prec;

    p1();
    p2[ebx*4]();
    p3.p();

```

使用过程指针时,需要注意的一点是,HLA不会对分配给一个过程指针变量的指针值强制进行严格的类型检验。特别地,如果指针变量声明与地址被分配给指针变量的过程参数列表不一致,这时若试图使用高级语法并通过指针间接地调用错误匹配的过程,可能会导致程序崩溃。类似于底层“纯粹”的过程调用,必须确保在调用之前栈中参数类型与个数的正确性。

5.19 过程参数

过程指针在参数列表中是很重要的。采用传递过程地址的方法选择调用许多过程中的某一个很常见。因此,HLA允许把过程指针作为参数来声明。

过程参数的声明并没有什么特别之处。看起来如同过程变量声明一样,只是它出现在参数列表中,而不是在一个变量声明段中。下面是一些典型的过程原型,说明如何声明这样的参数:

```

procedure p1( procparm: procedure ); forward;
procedure p2( procparm: procedure( i:int32 ) ); forward;
procedure p3( val procparm: procedure ); forward;

```

最后一行代码与第一行相同。它只是指出通常是通过值传递过程参数的。这看起来似乎违反直觉,因为过程指针是地址,所以需要传递一个地址来作为实参。然而,通过引用传递过程参数则意味着完全不同的事情。考虑下面的声明(该声明是合法的):

```

procedure p4( var procPtr:procedure ); forward;

```

这个声明告知HLA通过引用向p4传递一个过程变量。在其中,HLA所需的是一个过程指针变量的地址,而不是一个过程的地址。

当通过值传递过程指针时,可以把它指定为过程变量(HLA把它的数值传递给实际的过程)或者过程指针常量。一个过程指针常量由取址操作符(&)后跟一个过程名组成。传递过程常量或许是传递过程参数最便利的方法。例如,下面对例程Plot的调用可以把作为参数传递的函数在-2~+2之间进行分布。

```
Plot( &sineFunc );
Plot( &cosFunc );
Plot( &tanFunc );
```

需要注意, 不能通过简单地指定过程名称把一个过程作为参数传递——也就是 `Plot(sineFunc);` 将不起作用。简单地指定过程名不起作用是因为 HLA 试图直接调用这个指定名称的过程(切记, 一个过程名称出现在参数列表中会引起指令合成)。如果没有在参数/过程名后指定一个参数列表或一对空括号, 则 HLA 将产生一个语法错误信息。所以, 不要忘记在过程参数常量名称前放置取址操作符(&)。

5.20 无类型的引用参数

有时, 会通过引用向过程传递一个通用的存储器对象, 而并不关心这个存储器对象的类型是什么。一个典型的示例是对某些数据结构进行清零操作的过程。这样的过程会有如下所示的原型:

```
procedure ZeroMem( var mem:byte; count:uns32 );
```

这个过程将从第一个参数所指定的地址开始对 `count` 指定的字节数进行清零。这个过程原型存在的问题是, 如果试图传递其他非字节对象作为第一个参数的话, HLA 将报告错误。当然, 使用如下的类型强制操作可以克服这种问题, 但如果使用不同的数据类型调用这个过程多次, 那么这种强制操作符的使用也会很繁琐:

```
ZeroMem( (type=byte MyDataObject ), @size( MyDataObject ));
```

当然, 可以使用混合参数传递机制或手动地把参数推入栈, 但这些解决方法与使用类型强制操作符相比更加麻烦。幸运的是, HLA 提供了一种方便的解决方法: 无类型的引用参数。

准确来说, 无类型的引用参数是指通过引用传递的参数, 同时 HLA 并不对实参的类型和形参的类型进行比较。当含有一个无类型的引用参数时, 上述对 `ZeroMem` 的调用变为下面的形式:

```
ZeroMem( MyDataObject, @size( MyDataObject ));
```

其中, `MyDataObject` 可以是任何数据类型, 同时对 `ZeroMem` 的多次调用可以传递不同类型的对象而不会被 HLA 拒绝。

要声明一个无类型的引用参数, 可以用正常的语法指定这个参数, 只是用保留字 `var` 取代参数的类型。关键字 `var` 告知 HLA 对于这个参数来说, 任何变量对象都是合法的。注意, 必须通过引用传递无类型引用参数, 所以关键字 `var` 也就必须出现在参数声明之前。下面是过程 `ZeroMem` 使用无类型引用参数的正确声明方式:

```
procedure ZeroMem( var mem:var; count:uns32 );
```

有了这个声明, HLA 将计算作为实参传递给 `ZeroMem` 的任何存储器对象的地址, 同时把这个地址推入栈。

5.21 管理大型程序

大多数汇编语言源文件并不是独立存在的程序。通常情况下,您会调用不同的标准库或其他没有在主程序中定义的例程。例如,到目前为止您或许已经注意到 80x86 没有提供如 `read`、`write` 或 `put` 等 I/O 操作的机器指令。当然,可以自己编写相应的过程来完成这些功能。遗憾的是,编写这样的例程是一项很复杂的任务,而作为汇编语言的初学者还没有做好完成这种任务的准备。这时就需要使用 HLA 标准库。它是一个过程包,可以调用这些过程完成简单的 I/O 操作,如 `stdout.put`。

HLA 的标准库包含具有上万行程序的源代码。设想,如果把这些代码并入一个简单的程序将会给编程带来多大困难。对于程序来说,如果必须对这数万行代码进行编译又将是多么缓慢。幸运的是,我们并不用这样做。

对于小程序,使用一个简单的源文件就行了。而对于大型程序,这样就变得非常麻烦(考虑上面示例中所说的,必须把整个 HLA 标准库包含在每一个程序中)。此外,一旦已经调试并验证了很大一部分代码,当您在程序的其他部分做了一个很小的改动后,对整个程序继续汇编将会非常耗时。例如,HLA 的标准库即使在运行很快的机器上进行汇编也要几分钟的时间。想象一下,在一台运行很快的计算机上汇编代码仅作一行改动的程序要等待 20 或 30 分钟将是多么痛苦的事情。

对于高级语言,这种问题的解决方法是分离编译(*separate compilation*)。首先,把大规模的源文件分解为易处理的程序块;然后把这些分离的文件编译成目标代码模块;最后,把这些目标代码模块共同连接成为完整的程序。如果需要对其中的某个模块作微小改动,只须对这个模块重新汇编而不必重新汇编整个程序。

HLA 的标准库可以很好地按照这种方式工作。标准库已经进行了编译,可以直接使用。您只要调用标准库的例程并使用连接器把您的代码和标准库进行连接。使用标准库的代码来开发程序会节省大量的时间。当然,也可以很容易创建自己的对象模块,并把它们和您的程序一起进行连接。甚至还可以把自己写好的例程加入标准库中,这样可以在将来编写程序时使用。

“大规模编程(*programming in the large*)”是软件工程师用于描述处理大型软件项目开发的过程、方法和工具的术语。而每个人对“大规模”都有自己的理解,分离编译是支持“大规模编程”的多种流行技术之一。下面几节将讲述 HLA 提供的用于分离编译的工具,以及如何有效地在程序中使用这些工具。

5.22 #include 伪指令

当在源文件中遇到伪指令 `#include` 时,它会把程序输入从当前文件切换到伪指令 `include` 的参数列表中所指定的文件。这使得您可以构建包含通用常量、类型、源代码和其他 HLA 项的文本文件,同时把这些文件包含在若干分离程序的集合中。伪指令 `#include` 的语法为:

```
#include( "Filename" )
```

Filename 必须是有效的文件名。HLA 把指定的文件合并伪指令 `#include` 的编译位置。注意,可以在所包含的文件里嵌套使用 `#include` 语句。也就是说,在汇编期间,一个文件已经被包含在

另一个文件中，它还可以包含自己的文件。事实上，在许多示例程序中所见到的头文件 `stdlib.hhf` 只是一串 `#include` 语句(程序清单 5-18 列出了 `stdlib.hhf` 的源代码。注意，现在这个文件已经发生了很大的变化，但是基本概念仍然是相同的)。

程序清单 5-18 头文件 `stdlib.hhf`

```
#include( "hla.hhf" )
#include( "x86.hhf" )
#include( "misctypes.hhf" )
#include( "hl.hhf" )

#include( "excepts.hhf" )
#include( "memory.hhf" )

#include( "args.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "cset.hhf" )
#include( "patterns.hhf" )
#include( "tables.hhf" )
#include( "arrays.hhf" )
#include( "chars.hhf" )

#include( "math.hhf" )
#include( "rand.hhf" )

#include( "stdio.hhf" )
#include( "stdin.hhf" )
#include( "stdout.hhf" )
```

通过在源代码中包含 `stdlib.hhf`，就自动包含了所有的 HLA 库模块。在程序中只提供那些实际需要用到模块的 `#include` 语句会更加有效(就编译时间和产生代码的长度而言)。然而，包含 `stdlib.hhf` 非常方便，同时在文本上只占用很少的空间，这也是本书中大多数程序都使用 `stdlib.hhf` 的原因。

注意，伪指令 `#include` 不需要使用分号来结束。如果在 `#include` 后使用了分号，这个分号将成为源文件的一部分，并且在编译时被作为包含的源文件之后紧跟的第一个字符。HLA 通常允许在程序的不同部分出现多余的分号，因此可能会见到一个 `#include` 语句以分号结束。但一般来说，不要在 `#include` 语句后以分号结束，因为在某些特定环境下这可能会产生语法错误。

仅仅使用伪指令 `#include` 并没有提供分离编译。可以使用伪指令 `#include` 把一个大规模源文件分解成为许多分离模块，并在编译时把它们重新组合在一起。下面的示例将在程序编译时把文件 `printf.hla` 和 `putc.hla` 包含进来：

```
#include( "printf.hla" )
#include( "putc.hla" )
```

通过这种途径可以达到程序的模块化设计，程序可以从这种设计中获益，但是并没有节约开发时间。伪指令 `#include` 在编译时把源文件插入 `#include` 所在点，就如同自己键入的代码一样。HLA 仍然要花时间编译这样的代码。若要把标准库例程的所有文件都采用这种方式包含进来的

话,编译将永远进行。

一般来说,不应该像上面所示那样使用伪指令`#include`包含源代码²⁶。而应该使用伪指令`#include`把常量、类型、外部过程声明和其他一些项的通用集合插入程序中。通常情况下,汇编语言所包含的文件不含有任何机器代码(除了宏之外,详见第9章)。当您了解了外部声明是如何工作之后,通过这种方式来使用`#include`文件的目的是将变得更加清晰。

5.23 忽略重复的#include操作

当开发复杂的模块和库时,最终您将发现一个很大的问题:某些头文件还需要包含其他的头文件(例如,头文件`stdlib.hhf`包含HLA标准库的其他所有头文件)。实际上这并不是一个大问题,但当一个头文件包含另一个头文件,同时这第二个头文件又包含另一个,第三个头文件又包含另一个……而最后一个头文件包含第一个头文件时,问题就产生了。现在,它成了一个大问题。

当一个头文件间接地包含自身时会产生两个问题。第一,这会在编译器内部造成无限循环。编译器将一遍遍地对所包含的文件实施它的编译工作,直到存储器溢出或某些其他的错误发生。很明显,这并不是一件好事。产生的第二个问题(通常发生在第一个问题之前)是,当HLA第二次包含一个头文件时,它将针对重复定义的符号提出严重警告。毕竟,它第一次读到头文件时就对其中的所有声明进行了处理,而再一次遇到,它们就会被视为重复出现的符号。

HLA提供一种特殊的`include`伪指令来消除这个问题:`#includeonce`。这条伪指令的使用方法与`#include`完全相同,例如:

```
#includeonce("myHeaderFile.hhf")
```

当使用了`#includeonce`伪指令时,如果`myHeaderFile.hhf`直接或间接地包含了自身,HLA将忽略包含这个文件的新请求。然而,如果不是使用伪指令`#includeonce`而是使用`#include`,HLA将再一次包含这个文件。在某些确实需要包含一个头文件两次的情况下,才会这样做。

最后要说的是:应该总是使用伪指令`#includeonce`包含您创建的头文件。事实上,应该养成使用`#includeonce`的习惯,甚至对于别人创建的头文件也是如此(HLA的标准库已经具有防止递归包含的措施,因此不必担心对标准库的头文件使用`#includeonce`)。

还有一种可以用来防止递归包含的技术——条件编译。第9章在介绍宏和HLA编译时语言的时候会对这种方法进行讨论。

5.24 单元与external伪指令

从技术上讲,伪指令`#include`提供了用于创建模块化程序的所有功能。您可以创建许多模块,其中每个模块包含一些具体的例程,同时必要的时候可以在汇编程序里使用`#include`来包含这些模块。HLA提供了一种更好的方法:外部公有符号。

伴随`#include`机制的一个主要问题是,一旦已经调试了某个例程,当把它包含在编译过程中时

²⁶ 这样做并没有错,但事实上它并没有真正利用分离编译。

仍要耗费时间,因为当每次汇编主程序时,HLA 都必须对已经无误的代码重新进行编译。一个更好的解决方法是对已经调试过的模块进行预编译,而后再把目标代码模块连接起来。这也就是伪指令 `external` 所要做到的。

要使用 `external` 功能,必须至少创建两个源文件。其中第一个文件包含第二个文件使用的一系列变量和过程。第二个文件使用这些变量和过程,但不清楚它们是如何实现的。唯一存在的问题是,如果创建了两个分离的 HLA 程序,当试图把它们组合起来时,连接器将产生混乱。这是因为每个 HLA 程序都有它们自己的主程序。这样,当把程序装入内存后,操作系统将不知道该运行哪个主程序。为了解决这个问题,HLA 使用一个不同类型的编译模块——单元(unit),来编译一个没有主程序的程序。HLA 单元的语法实际上要比 HLA 程序简单,它采用下面的形式:

```
unit unitname;

    << declarations >>

end unitname;
```

只有一个特例(var 段),其他任何能够在 HLA 程序声明段出现的内容都可以在 HLA 单元的声明段出现。需要注意,单元不含有 `begin` 子句,同时也没有程序语句²⁷,它只包含声明。

除了单元不包含主程序段以外,单元与程序之间还有另一个区别:单元不能含有 `var` 段。这是因为 `var` 段用于声明局部隶属于主程序源代码的自动变量。因为不存在与单元相关联的“主程序”,所以 `var` 段是非法的²⁸。

为了便于说明,考虑程序清单 5-19 和程序清单 5-20 中的两个模块。

程序清单 5-19 一个简单的 HLA 单元示例

```
unit Number1;

static
    Var1: uns32;
    Var2: uns32;

    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;

end Number1;
```

程序清单 5-20 引用外部对象的主程序

```
program main;
#include( "stdlib.hhf" );
```

27 当然,单元可以包含过程,而这些过程包含语句,但单元本身并不具有任何与它相联系的可执行指令。

28 当然,单元中的过程可以具有它们自己的 `var` 声明段,但过程的声明段与单元的声明段是相分离的。


```
begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Addland2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

主程序引用 Var1、Var2 和 Addland2，然而这些符号对这个程序来说都是外部的(它们出现在单元 Number1 中)。如果不作修改就编译这个主程序，HLA 将会报警，指出这 3 个符号没有定义。

因此，必须使用选项 **external** 对它们进行外部声明。外部过程声明看起来与向前引用声明很相似，差别是使用保留字 **external** 而不是 **forward**。要声明外部静态变量，只须在这些变量声明后面添加保留字 **external**。程序清单 5-21 是对程序清单 5-20 的改进，它包含了外部声明。

程序清单 5-21 使用外部声明改进的主程序

```
program main;
#include( "stdlib.hhf" );

    procedure Addland2; external;

static
    Var1: uns32; external;
    Var2: uns32; external;

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Addland2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

如果试图对 main 的第二个版本进行编译，使用典型的 HLA 编译命令 **HLA main2.hla** 将会让您略有失望。这个程序将被编译通过，并且没有错误。然而，当 HLA 试图连接代码时，它会报出符号 Var1、Var2 和 Addland2 没有定义。这是因为没有编译相应的单元并把它连接到这个主程序。当您这样做了之后，会发现它仍然不能被编译通过。应该知道，单元中的所有符号在默认情况下对于这个单元都是私有(private)的，这就意味着这个单元以外的代码不能访问这些符号，除非显式声明这些符号为公有(public)符号。要将符号声明为公有的，只须把单元中符号的外部声明放在实际符号声明之前。如果符号的外部声明与实际声明出现在同一个源文件中，HLA 假定存在对这个名称的外部需要，并使它成为一个公有符号(而不是私有的)。程序清单 5-22 中的代码对单元 Number1 作了修正，正确声明了外部对象。

程序清单 5-22 使用外部声明修正单元 Number1

```
unit Number1;

static
```

```

Var1: uns32; external;
Var2: uns32; external;

procedure Add1and2; external;

static
  Var1: uns32;
  Var2: uns32;

  procedure Add1and2;
  begin Add1and2;

    push( eax );
    mov( Var2, eax );
    add( eax, Var1 );

  end Add1and2;

end Number1;

```

在程序清单 5-21 和程序清单 5-22 中，对这些符号的两次声明看起来似乎是多余的，但马上就会看到，通常不必这样编写代码。

如果试图使用典型的 HLA 语句编译程序 `main` 或单元 `Number1`，即：

```

HLA main2.hla
HLA unit2.hla

```

您会发现，连接器仍然返回错误。它是对 `main2.hla` 的编译返回了一个错误，因为您仍然没有告知 HLA 把 `unit2.hla` 的目标代码连接进来。同样地，如果单独编译 `unit2.hla`，连接器同样报错，因为它找不到主程序。解决方法很简单，使用下面的命令把这两个模块共同进行编译：

```

HLA main2.hla unit2.hla

```

这条命令将正确地对两个模块进行编译并把它们的目标代码连接起来。

遗憾的是，上面的命令破坏了分离编译的一个主要好处。当发出这条命令时，它将首先对 `main2` 和 `unit2` 进行编译，而后再进行连接。应该记得，使用分离编译的一个主要原因就是减少大规模项目的编译时间。虽然上面的命令很方便，但却没有达到这个目的。

要分离编译这两个模块，必须分别对它们运行 HLA 命令。当然，前面已经看到，对这两个模块分别编译会产生目标代码连接错误。为了避免这个问题，需要编译这两个模块，但却不对它们进行连接。HLA 的只编译命令行选项 `-c` 做到了这一点。要编译这两个源文件而不运行连接器，可以使用下面的命令：

```

HLA -c main2.hla
HLA -c unit2.hla

```

这样就产生了两个目标代码文件，`main2.obj` 和 `unit2.obj`，可以把它们共同连接生成一个可执行文件。直接运行连接器可以做到这一点，但另一个更简单的方法是使用 HLA 的编译器把目标模块连接起来：

```
HLA main2.obj unit2.obj
```

在 Windows 操作系统下, 这条命令会生成一个名为 `main2.exe` 的可执行文件²⁹; 而在 Linux、Mac OS X 和 FreeBSD 操作系统下, 这条命令生成一个名为 `main2` 的文件。也可以使用下面的命令来编译主程序, 并把它与先前已经编译好的目标模块 `unit2` 进行连接:

```
HLA main2.hla unit2.obj
```

通常情况下, HLA 会查看跟在 HLA 命令后的文件名后缀。如果文件名不带后缀, HLA 会假定它是一个 .HLA 文件。如果文件名带有后缀, 则 HLA 会对文件做如下操作:

- 若后缀是 .HLA, HLA 使用 HLA 编译器编译文件。
- 若后缀是 .ASM, HLA 使用 MASM(或 Windows 操作系统下的其他默认汇编器, 如 FASM、NASM 或 TASM)或 Gas(Linux/Mac OS X/FreeBSD 操作系统)对文件进行汇编。
- 若后缀是 .OBJ 或 .LIB(Windows 操作系统), 或者 .o 或 .a(Linux/Mac OS X/FreeBSD 操作系统), 那么 HLA 把模块与其他的编译文件进行连接。

5.24.1 伪指令 `external` 的行为

无论何时使用伪指令 `external` 声明一个符号, 都要牢记对 `external` 对象的一些限制:

- 在一个给定的源文件里, 一个对象只能出现一次 `external` 声明。也就是说, 不能把同一个符号作为 `external` 对象定义两次。
- 只有 `procedure`、`static`、`readonly` 和 `storage` 变量可以作为外部对象使用。而 `var`、`type`、`const` 和参数对象不能是外部的。
- 外部对象必须出现在全局声明层, 不可以在一个过程或其他嵌套结构内声明 `external` 对象³⁰。
- `external` 对象全局地公布它们的名称。因此, 对 `external` 对象名称的选择要格外小心, 以避免与其他符号相冲突。

最后一点尤为重要, 应该切记。HLA 使用连接器对目标模块进行连接。在这个过程中的每一步, 您所选择的外部名称都可能会产生问题。

考虑下面的 HLA 外部/公有声明:

```
static
    extObj:          uns32; external;
    extObj:          uns32;
    localObject:     uns32;
```

当编译一个包含这些声明的程序时, HLA 会自动为变量 `localObject` 生成一个“特别”的名称, 使得它不会与系统全局外部符号发生任何冲突³¹。然而, 当声明一个外部符号时, HLA 使用这个

29 如果想要显式地指定输出文件的名称, HLA 提供了一个命令行选项来达到这个目的。可以通过键入命令“HLA-?”获得所有合法的命令行选项菜单。

30 有一些例外情况, 但除了在全局层以外您不能声明外部过程或变量。

31 通常情况下, HLA 会产生一个源自 `localObject` 的名称, 如 `001A_localObject`, 这是一个合法的 MASM 标识符。但是, 当 HLA 使用 MASM 编译程序时, 它几乎不会与其他任何全局符号发生冲突。

对象名作为默认的外部名称。如果不注意把某个全局名称作为变量名使用,将会产生问题。

为了避免外部名称冲突问题的发生,HLA 支持选项 `external` 的一个附加语法,让您可以显式地指定外部名称。下面的示例说明了这种扩展语法:

```
static
    c: char; external( "var_c" );
    c: char;
```

如果在关键字 `external` 后面紧跟一个带有括号的字符串常量,在声明该对象的 HLA 源码中,HLA 将继续把声明的名称(本示例中的 `c`)作为标识符来使用。当在外部(也就是在汇编代码中)引用变量 `c` 时,HLA 会用 `var_c` 代替。这种特性可以帮助您有效地避免由于错误地在 HLA 程序中使用汇编保留字或其他全局符号而带来的问题。

还应该注意,选项 `external` 的这个特性允许创建别名。例如,您或许会在一个模块中使用名称 `StudentCount` 引用一个对象,而在另一个模块中则把这个对象作为 `PersonCount` 来引用(这样做可能是因为已经具有能够计算人数的通用库模块,而您只想在程序中使用它计算学生的人数)。使用如下的声明能做到这一点:

```
static
    StudentCount: uns32; external( "PersonCount" );
```

当然,前面已经看到,创建别名可能会遇到一些问题。因此,在程序中应该有节制地使用这种特性。对这种特性一个更合理的使用是使得某些操作系统 API 简单化,例如,Win32 的 API 对某些过程的调用就使用很长的名称。可以通过使用伪指令 `external` 提供一个比操作系统所指定的标准名称更有意义的名称。

5.24.2 HLA 中的头文件

像定义外部符号一样,HLA 使用 `external` 声明定义公有符号,但这样做似乎与直觉相反。为什么不使用关键字 `public` 声明公有符号同时针对外部定义使用关键字 `external` 呢? HLA 外部声明的这种违反直觉特征是建立在多年使用 C/C++ 编程语言的牢固经验基础之上的,在 C/C++ 编程语言中使用很相似的方法处理公有符号和外部符号³²。结合 HLA 的头文件,HLA 的外部声明使得大型程序的维护变成一件轻而易举的事情。

伪指令 `external`(而不是单独的 `public` 和 `external` 伪指令)有一个很重要的好处,它使得源文件中的成果复制达到最小化。例如,假设要建立一个模块,而这个模块包含许多将在其他不同的程序(如 HLA 的标准库)中使用的一系列的支持例程和变量。除了共享某些例程和变量以外,假设还要共享常量、类型和其他一些项。

`#include` 文件机制为这种情况提供了很好的解决办法。您只须创建一个包含常量、宏和 `external` 定义的 `#include` 文件,并在实现自己例程的模块和使用这些例程的模块中包含这个文件,如图 5-10 所示。

³² 事实上,C/C++ 还略有不同。一个模块中所有的全局符号都假定为公有的,除非显式声明为私有的。HLA 的方法(通过 `external` 强制公有项的声明)使用起来会更安全。

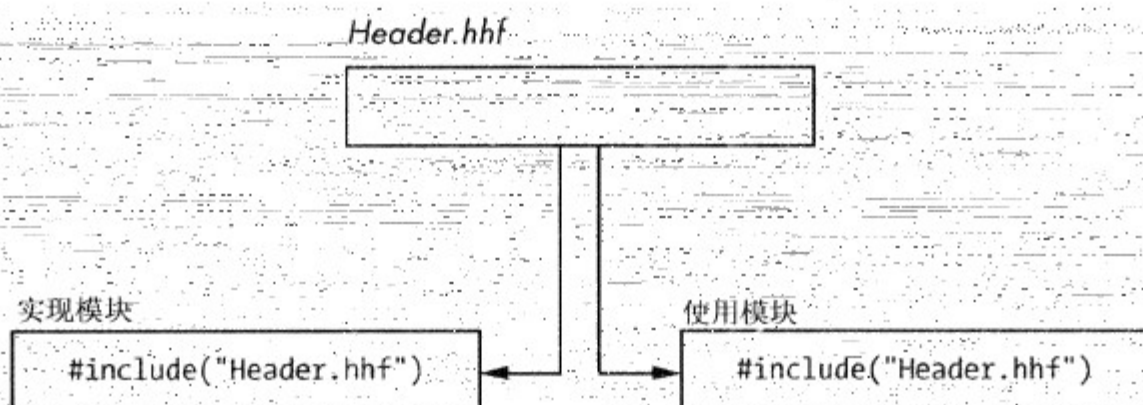


图 5-10 在 HLA 程序中使用头文件

一个典型的头文件只包含 `const`、`val`、`type`、`static`、`readonly`、`storage` 和过程原型(还可以加入一些我们现在还没有见到的结构,如宏)。在 `static`、`readonly` 和 `storage` 段内的对象以及所有的过程声明总是 `external` 对象。特别是不要把任何 `var` 对象放在头文件中,也不能把任何非外部变量或过程体放入头文件。如果这样做,HLA 将在包含这种头文件的源文件里对这些对象进行复制,这样不但使程序规模变大,而且在某些特定情况下会导致执行失败。例如,一般来说,把一个变量放在头文件中的目的是可以在多个其他模块中共享这个变量的值。但在头文件中如果没有把它声明为外部符号,而只是一个标准的变量声明,那么每个包含源文件的模块都将获得它们自己独立的变量,模块之间没有共享一个共同的变量。

如果创建了一个标准头文件,包含 `const`、`val`、`type` 声明和外部对象,要确定保需要使用头文件中的定义的所有模块声明段内包含这个文件。通常情况下,HLA 程序使用紧跟 `program` 或 `unit` 头之后的几条语句来包含所用到的头文件。

本书沿用 HLA 标准库的约定,使用 `.hhf` 作为 HLA 头文件的后缀(`hhf` 指 HLA header file)。

5.25 命名空间污染

命名空间污染(namespace pollution)是用许多不同模块创建库时遇到的一个问题。一个典型的库模块将会有有一个与之相对应的 `#include` 文件,这个文件为库中所出现的例程、常量、变量和其他符号提供外部定义。当要使用库中的例程或其他对象时,典型方法是使用 `#include` 把库的头文件包含在自己的项目中。随着库的不断变大,同时在头文件中也增加了更多的声明,这时很可能会出现库中标识符的名称与当前项目中想要使用的名称相冲突的情况,这就是所谓的命名空间污染。为了更容易访问实际使用的库中的一小部分例程,库的头文件使用了通常不需要使用的名称污染了命名空间。多半时候,这些名称并不会造成任何损害,除非由于自己的目的想要使用它们。

HLA 要求所有的外部符号必须在全局级别(`program/unit`)声明。因此,不能在过程里包含一个具有外部声明的头文件。所以,命名冲突不会发生在外部库符号和过程中的局部符号之间,只有全局符号才有可能和外部符号发生命名冲突。这也是一个很好的论据来说明在程序中要尽可能避免使用全局符号,但事实上,汇编程序中的大多数符号都具有全局作用域。因此需要有另一种解决方法。

HLA 的解决办法是把大多数库名称放在一个命名空间(namespace)声明段中。一个命名空间声明封装了所有的声明,同时只在全局层显示一个简单的名称(namespace 标识符)。可以通过熟知的圆点表示法访问命名空间中的名称(参见 4.34 节对命名空间的讨论)。这种方法通过把大量的名称

转换为一个单一名称的方法减少了命名空间污染所造成的影响。

当然,使用命名空间声明有一个不足之处,就是必须键入一个很长的名称才能访问命名空间中的某个特定标识符(也就是必须键入命名空间标识符,一个圆点和想要使用的特定标识符)。对于那些经常访问的标识符,应该有选择地把它们放置在任何命名空间声明之外。例如,HLA 标准库没有把符号 `nl` 定义在一个命名空间中。尽管如此,您还是想把库中这样的声明达到最少,以避免与自己程序中的其他名称发生冲突。往往可以选择一个命名空间标识符作为例程名称的补充。例如,HLA 标准库的字符串复制例程按照 C 的标准库函数 `strcpy` 的形式命名。HLA 的命名是 `str.cpy`,实际的函数名称为 `cpy`,它恰好是命名空间 `str` 的一个成员,因此整个名称 `str.cpy` 与 C 的函数相比非常类似。HLA 的标准库包含许多采用这种方法的示例。函数 `arg.c` 和 `arg.v` 是另一对这样的标识符(对应于 C 的标识符 `argc` 和 `argv`)。

在一个头文件中使用命名空间与在 `program` 或 `unit` 中使用命名空间相比并没有不同,但是通常不会把实际的过程体放在一个命名空间中。下面是一个包含命名空间声明的典型头文件示例:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file

namespace myLib;

    procedure func1; external;
    procedure func2; external;
    procedure func3; external;

end myLib;
```

通常情况下,您将把每个函数(`func1~func3`)作为独立的单元来编译(所以每个函数都有它们自己的目标文件,连接一个函数并不会把它们都连接起来)。下面是这些函数中一个单元的示例声明:

```
unit func1Unit;
#includeonce( "myHeader.hhf" )

procedure myLib.func1;
begin func1;

    << Code for func1 >>

end func1;
end func1Unit;
```

对于这个单元要注意到两点。第一,不能把 `func1` 的实际过程代码放在一个命名空间声明块中。通过使用标识符 `myLib.func1` 作为过程的名称,HLA 会自动识别这个过程的声明属于一个命名空间。第二,在过程的 `begin` 和 `end` 子句后面的 `func1` 之前不必使用 `myLib.`,HLA 会自动把 `begin` 和 `end` 标识符与过程声明相关联。此时 HLA 知道这些标识符是 `myLib` 命名空间的一部分,这使您不必再键入整个名称。

需要重点注意,当在一个命名空间中声明外部名称时,如在前面的 `func1Unit` 中所实现的,HLA 只使用函数名(本示例中的 `func1`)作为外部名称。这会在外部命名空间中产生命名空间污染问

题。例如，如果有两个不同的命名空间 `myLib` 和 `yourLib`，同时它们都定义一个过程 `func1`，若要同时使用这两个库模块中的函数，目标代码连接器会报出 `func1` 的重复定义错误。这个问题有一个很容易的解决方法：使用伪指令 `external` 的扩展形式为出现在命名空间声明段中的所有外部标识符显式提供一个外部名称。例如，可以对上面的文件 `myHeader.hhf` 做如下的简单修改来解决这种问题：

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file

namespace myLib;

    procedure func1; external( "myLib_func1" );
    procedure func2; external( "myLib_func2" );
    procedure func3; external( "myLib_func3" );

end myLib;
```

本示例展示了一种良好的编程习惯：当要从一个命名空间导出名称时，总是通过把命名空间标识符与对象的内部名称用下划线相连的方法提供一个明确的外部名称。

使用命名空间声明并没有完全消除命名空间污染的问题(毕竟，命名空间标识符仍然是一个全局对象，包含了头文件 `stdlib.hhf` 同时又试图定义一个变量 `cs` 的人可以证明这一点)，但它已经基本上解决了这种问题。因此，在创建自己的库时，应该尽可能地使用命名空间。

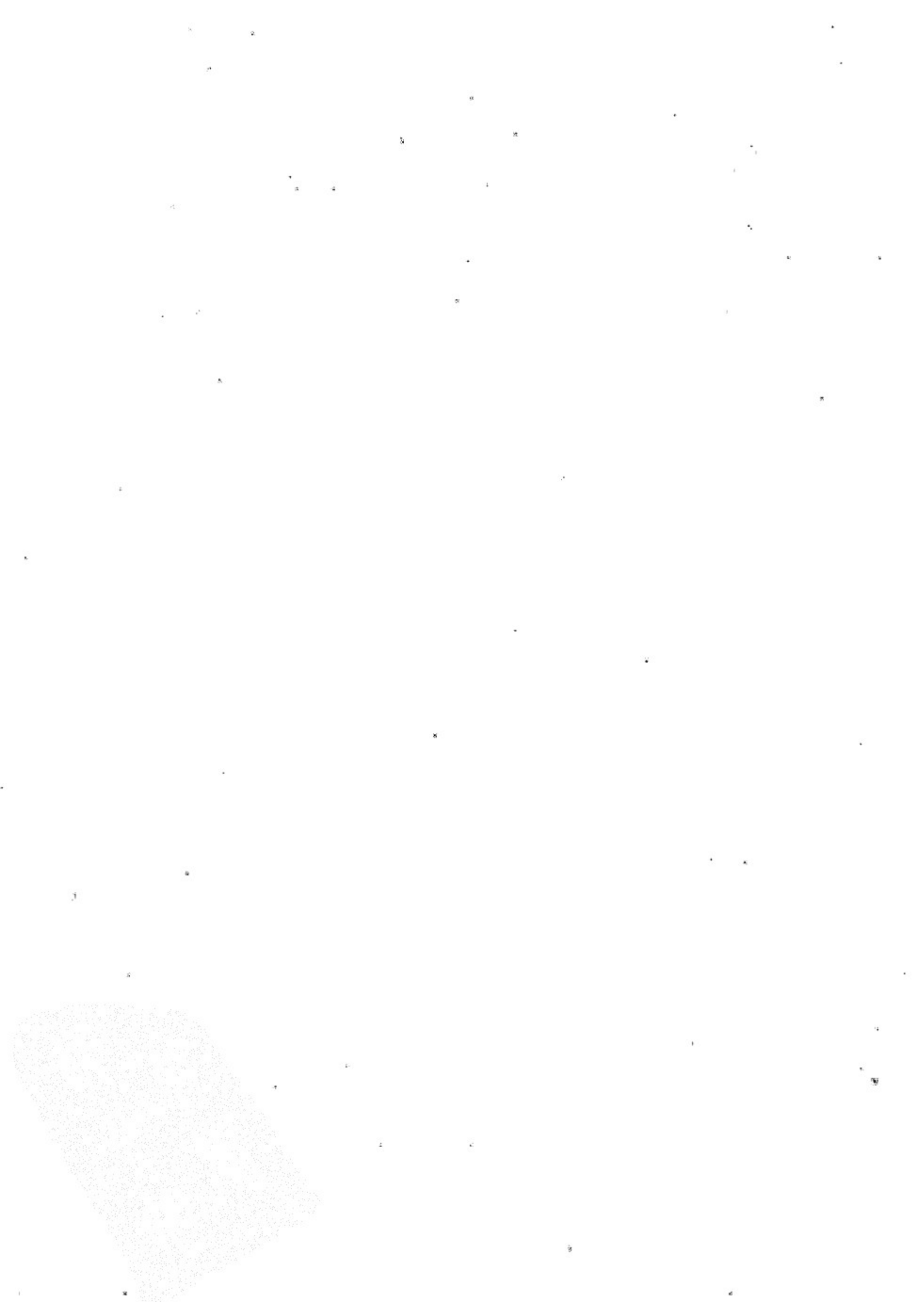
5.26 更多信息

本书的电子版本(地址为 <http://www.artofasm.com/>或 <http://webster.cs.ucr.edu/>)包含了关于高级和中级过程的一整“卷”内容。本章的内容取自电子版本的介绍性和中间章节。虽然本章中出现的信息已经覆盖了汇编程序员可用的 99% 的资源，但您还可以发现另外一些关于过程和参数的信息。特别是，电子版本包含了其他的参数传递机制(值/结果传递、结果传递、名称传递和延迟求值传递)以及参数传递位置的相关内容。其中还包括了迭代器、形实转换程序和其他高级过程类型。还可以通过查阅 HLA 相关文档获得更多关于 HLA 过程特性的细节。一本关于编译器构造的优秀书籍应该会包含关于对过程的运行时支持的更多细节。

本章只讨论了 32 位的本地过程(适合于如 Windows、Mac OS X、FreeBSD 和 Linux 这样的操作系统)。想获得关于 16 位代码(包括本地和远端过程)过程的相关信息，可以参考本书的 16 位版本，网址同样为 <http://www.artofasm.com/>和 <http://webster.cs.ucr.edu/>。

HLA 具有支持过程嵌套的能力，意思是可以在某些其他过程的声明段里再声明一个过程，同时可以在外围过程中通过显示和静态连接的方法访问自动变量。HLA 还支持高级参数指针工具。本文不对这种工具进行讨论是因为它有些超前，很少有汇编语言程序员在他们的程序中利用它。然而，这些特性在某些情况下是非常便捷的。一旦您适应了过程和通用的汇编语言编程，应该继续阅读本书电子版本的有关中级和高级过程的章节中以及 HLA 文档中关于嵌套过程的 HLA 工具的内容。

当使用高级语法传递参数时,HLA 会生成一些代码。本书的电子版本中针对这样的代码给出的示例并不完整。HLA 在逐渐改进把参数传递到栈上时生成代码的质量。如果想要查看 HLA 为特定参数调用序列生成的代码的类型,那么应该向 HLA 提供 `-sourcemode`、`-h` 和 `-s` 命令行参数,并查看 HLA 生成的对应的汇编语言文件(这是一个伪 HLA 源文件,显示了 HLA 产生的低级代码)。



第 6 章

算 术 运 算



本章讨论汇编语言中的算术运算。本章学习结束后，您应该能够把像 Pascal 和 C/C++ 这样的高级语言中的算术表达式和赋值语句转换成 80x86 的汇编语言。

6.1 80x86 的整数运算指令

在讲述如何把算术表达式转换成汇编语言之前，首先讨论 80x86 指令集中剩余的算术运算指令。前面的章节已经介绍了大部分算术运算和逻辑运算指令，所以本章只讨论少数剩余的指令。

6.1.1 mul 和 imul 指令

乘法运算指令会让您感受到 80x86 指令集的另一种不规则性。80x86 指令集中的 `add`、`sub` 和其他一些指令像 `mov` 指令一样支持两个操作数。但 80x86 的操作码字节没有足够的位数支持所有的指令，所以 80x86 把 `mul`(无符号乘法)和 `imul`(有符号整数乘法)作为单操作数指令处理，就像 `inc`、`dec` 和 `neg` 指令一样。

当然，乘法是一个双操作数函数。为了能够基于这个事实正常工作，80x86 总是假定目的操作数是累加器(AL、AX 或 EAX)。这种不规则性使得使用 80x86 的乘法指令比使用其他指令更困难，因为使用乘法指令时，有一个操作数必须在累加器中。Intel 采用这种非正交的方法是因为他们感到编程人员使用乘法要比使用像 `add` 和 `sub` 这样的指令少得多。

伴随 `mul` 指令和 `imul` 指令的另一个问题是，不能使用这两条指令把一个常量和累加器相乘。Intel 很快就发现了支持常量乘法的必要性，因此增加了 `imtmul` 指令解决这个问题。然而，必须明白的一点是，基本的 `mul` 和 `imul` 指令不像 `imtmul` 那样支持全范围的操作数。

乘法指令有两种形式：无符号乘法指令(`mul`)和有符号乘法指令(`imul`)。与加法和减法不同，必须对有符号和无符号操作的乘法指令进行区分。

乘法指令的形式如下所示。

无符号乘法:

```
mul( reg8 );      // returns "ax"
mul( reg16 );     // returns "dx:ax"
mul( reg32 );     // returns "edx:eax"

mul( mem8 );      // returns "ax"
mul( mem16 );     // returns "dx:ax"
mul( mem32 );     // returns "edx:eax"
```

有符号(整数)乘法:

```
imul( reg8 );     // returns "ax"
imul( reg16 );    // returns "dx:ax"
imul( reg32 );    // returns "edx:eax"

imul( mem8 );     // returns "ax"
imul( mem16 );    // returns "dx:ax"
imul( mem32 );    // returns "edx:eax"
```

上面给出的返回值是这些指令返回的用于 HLA 指令合成的字符串。(i)mul 可用于所有 80x86 处理器中的 8 位、16 位或 32 位操作数乘法。

当对两个 n 位数值进行乘法运算时,需要 $2*n$ 位的空间保存结果。因此,如果操作数是 8 位数据,乘法的结果将需要 16 位。同样地,16 位操作数乘法产生 32 位结果,而 32 位操作数乘法则需要 64 位空间保存结果。

带有一个 8 位操作数的指令(i)mul,用这个操作数与 AL 相乘,同时把 16 位结果保存在 AX 中。所以

```
mul( operand8 );
```

或

```
imul( operand8 );
```

计算的是:

```
ax := al * operand8
```

“*”在 mul 中代表一个无符号乘法,而在 imul 中代表一个有符号乘法。

如果指定一个 16 位操作数,则 mul 和 imul 计算的是:

```
dx:ax := ax * operand16
```

“*”的含义与前面的计算相同,而 dx:ax 意味着 DX 保存的是 32 位结果的高位字,而 AX 保存的是 32 位结果的低位字。会让人产生疑问的是为什么 Intel 不把 32 位的结果保存在 EAX 中,这是因为 Intel 是在早期的 80x86 处理器中引入 mul 和 imul 指令的,而直到 80386 的处理器中才出现了 32 位的寄存器。

如果指定一个 32 位操作数, 则 `mul` 和 `imul` 计算的是:

```
edx:eax := eax * operand32
```

“*” 仍然代表与前面相同的意思, 而 `edx:eax` 意味着 EDX 保存 64 位结果的高位双字, 同时 EAX 保存 64 位结果的低位双字。

如果一个 8×8 、 16×16 或 32×32 位的乘积分别大于 8、16 或 32 位, 则 `mul` 和 `imul` 指令将对进位标志和溢出标志进行设置。指令 `mul` 和 `imul` 会修改符号标志和零标志。

注意:

特别要注意的是, 在这两条指令执行完后, 符号标志和零标志不包含有意义的数值。

为了减少使用指令 `mul` 和 `imul` 所带来的语法不规则性, HLA 提供了允许使用双操作数的语法扩展, 形式如下所示。

无符号乘法:

```
mul( reg8, al );
mul( reg16, ax );
mul( reg32, eax );

mul( mem8, al );
mul( mem16, ax );
mul( mem32, eax );

mul( constant8, al );
mul( constant16, ax );
mul( constant32, eax );
```

有符号(整数)乘法:

```
imul( reg8, al );
imul( reg16, ax );
imul( reg32, eax );

imul( mem8, al );
imul( mem16, ax );
imul( mem32, eax );

imul( constant8, al );
imul( constant16, ax );
imul( constant32, eax );
```

这种双操作数形式允许指定(低位)目的寄存器作为第二个操作数。通过指定目的寄存器, 能够使得程序更易读。需要注意的是, 虽然 HLA 在这里允许使用双操作数, 但并不能任意指定一个寄存器。目的操作数必须是 AL、AX 或 EAX, 并根据源操作数的形式决定应该是哪一个。

HLA 提供一种形式来指定一个常量。80x86 实际上并不支持一个含有常量操作数的 `mul` 或 `imul` 指令。HLA 将读取所指定的常量, 同时在存储器的只读段创建一个变量, 并使用读取的数值对这个变量进行初始化。这样 HLA 就把这条指令转换为 `(i)mul(memory)` 指令。需要注意, 当指定

一个常量作为源操作数时,指令需要两个操作数(因为 HLA 要使用第二个操作数确定这次乘法是 8 位、16 位还是 32 位的)。

当在第 8 章学习扩展精度的算术运算时,将频繁使用 `mul` 和 `imul` 指令。除非要完成多倍精度的任务,否则您或许会使用指令 `intmul` 代替指令 `mul` 或 `imul`,因为它是更通用的指令。然而,指令 `intmul` 并不能完全代替这两条指令。除了操作数个数外,在指令 `intmul` 和 `mul/imul` 之间还存在许多不同之处。指令 `intmul` 的特殊之处为:

- 没有一条可用的 8×8 位 `intmul` 指令。
- 指令 `intmul` 并不产生一个 $2 \times n$ 位的结果。也就是说, 16×16 位乘法产生一个 16 位的结果。同样地, 32×32 位乘法产生一个 32 位的结果。如果产生的结果与目的寄存器不符合,这些指令将对进位标志和溢出标志进行置位。

6.1.2 `div` 和 `idiv` 指令

80x86 的除法指令可以完成 64/32、32/16 或 16/8 位的除法。这些指令采用下列形式:

```
div( reg8 );           // returns "al"
div( reg16 );          // returns "ax"
div( reg32 );          // returns "eax"

div( reg8, ax );       // returns "al"
div( reg16, dx:ax );   // returns "ax"
div( reg32, edx:eax ); // returns "eax"

div( mem8 );           // returns "al"
div( mem16 );          // returns "ax"
div( mem32 );          // returns "eax"

div( mem8, ax );       // returns "al"
div( mem16, dx:ax );   // returns "ax"
div( mem32, edx:eax ); // returns "eax"

div( constant8, ax );  // returns "al"
div( constant16, dx:ax ); // returns "ax"
div( constant32, edx:eax ); // returns "eax"

idiv( reg8 );          // returns "al"
idiv( reg16 );         // returns "ax"
idiv( reg32 );         // returns "eax"

idiv( reg8, ax );      // returns "al"
idiv( reg16, dx:ax );  // returns "ax"
idiv( reg32, edx:eax ); // returns "eax"

idiv( mem8 );          // returns "al"
idiv( mem16 );         // returns "ax"
idiv( mem32 );         // returns "eax"

idiv( mem8, ax );      // returns "al"
idiv( mem16, dx:ax );  // returns "ax"
idiv( mem32, edx:eax ); // returns "eax"
```



```

    idiv( constant8, ax );           // returns "al"
    idiv( constant16, dx:ax );       // returns "ax"
    idiv( constant32, edx:eax );     // returns "eax"

```

指令 `div` 进行的是一个无符号除法计算。如果操作数是一个 8 位数据，指令 `div` 用寄存器 `AX` 除以这个操作数，商保存在 `AL` 中，余数(模数)保存在 `AH` 中。如果操作数是 16 位数据，那么指令 `div` 用寄存器对 `dx:ax` 中的 32 位数据除以这个操作数，商保存在 `AX` 中，余数保存在 `DX` 中。对于 32 位的操作数，指令 `div` 用 `edx:eax` 中的 64 位数值除以这个操作数并把商保存在 `EAX` 中，余数保存在 `EDX` 中。

与指令 `mul` 和 `imul` 类似，HLA 为这两条指令提供特殊的语法允许使用常量操作数，即使底层的机器指令实际上并不支持它们。从前面的 `div` 指令列表中可以看到这些扩展。

`idiv` 指令的结果得到的是有符号商和余数。`idiv` 指令的语法类似于 `div`(只是使用的是 `idiv` 助记符)，但是在执行 `idiv` 之前，为 `idiv` 创建有符号操作数所需的指令序列与 `div` 不同。

在 80x86 中，不能简单地使用两个无符号 8 位数值进行除法操作。如果除数是一个 8 位数值，则被除数必须是一个 16 位的数值。如果需要对两个无符号 8 位数值进行除法操作，必须把被除数零扩展为 16 位数值。通过把被除数读入寄存器 `AL` 并在寄存器 `AH` 中移入 0 完成这种零扩展。这样就可以使用 `AX` 作为被除数产生正确的结果。如果在执行指令 `div` 之前没有对 `AL` 进行零扩展，可能会导致 80x86 产生错误的结果！当需要对两个 16 位无符号数值进行除法运算时，必须把寄存器 `AX`(包含被除数)零扩展到寄存器 `DX` 中。只需要在寄存器 `DX` 中装入 0 就可以完成这个操作。如果需要对两个 32 位数值进行除法运算，必须在进行运算之前把寄存器 `EAX` 零扩展到寄存器 `EDX` 中(通过在 `EDX` 中装入 0)。

当处理有符号整型数值时，在执行指令 `idiv` 之前，必须符号扩展 `AL` 到 `AX` 中、`AX` 到 `DX` 中或者 `EAX` 到 `EDX` 中。使用指令 `cbw`、`cwd`、`cdq` 或 `movsx` 完成这些扩展。如果高位字节、字或双字已经不包含有意义的数据，执行 `idiv` 操作之前必须在累加器(`AL/AX/EAX`)中对数值进行符号扩展。否则可能会产生错误的结果。

80x86 的除法指令还有另外一个问题：使用指令时可能会产生致命的错误。首先，可能会使用 0 作除数。第二个问题是，对于寄存器 `EAX`、`AX` 或 `AL` 来说，商的规模可能过大。例如，16/8 位的除法 $8000/2$ 产生的商是 4000，余数是 0。4000 不能放到一个 8 位的寄存器中。如果这样的情况发生或用 0 作除数，80x86 将产生一个 `ex.DivisionError` 异常或整数溢出错误(`ex.IntoInstr`)。通常情况下，这意味着将弹出一个适当的对话框并异常中止程序的执行。如果发生这样的问题，可能是由于在执行除法操作之前没有对被除数进行符号扩展或零扩展。由于这种错误可能会导致程序崩溃，因此，对用于除法操作的数值应该格外当心。当然，可以在程序中对 `ex.DivisionError` 和 `ex.IntoInstr` 使用程序块 `try..endtry` 来捕获这种问题。

在执行了一个除法操作后，80x86 的进位标志、溢出标志、符号标志和零标志都是不确定的。因此，在一个除法操作后，不能通过标志位检验问题。

对于除法操作，80x86 没有提供一条单独用于计算余数的指令。指令 `div` 和 `idiv` 在计算商的同时自动计算余数。然而，HLA 为 `mod` 和 `imod` 指令提供了助记符(指令)。这两条特殊的 HLA 指令被编译成与它们相应的 `div` 和 `idiv` 完全相同的代码。唯一的差别在于指令的返回值(因为这两条指令把余数返回到与商不同的一个位置)。HLA 支持的 `mod` 和 `imod` 指令为：

```

mod( reg8 );           // returns "ah"
mod( reg16 );          // returns "dx"
mod( reg32 );          // returns "edx"

mod( reg8, ax );       // returns "ah"
mod( reg16, dx:ax );   // returns "dx"
mod( reg32, edx:eax ); // returns "edx"

mod( mem8 );           // returns "ah"
mod( mem16 );          // returns "dx"
mod( mem32 );          // returns "edx"

mod( mem8, ax );       // returns "ah"
mod( mem16, dx:ax );   // returns "dx"
mod( mem32, edx:eax ); // returns "edx"

mod( constant8, ax );  // returns "ah"
mod( constant16, dx:ax ); // returns "dx"
mod( constant32, edx:eax ); // returns "edx"

imod( reg8 );          // returns "ah"
imod( reg16 );         // returns "dx"
imod( reg32 );         // returns "edx"

imod( reg8, ax );      // returns "ah"
imod( reg16, dx:ax );  // returns "dx"
imod( reg32, edx:eax ); // returns "edx"

imod( mem8 );          // returns "ah"
imod( mem16 );         // returns "dx"
imod( mem32 );         // returns "edx"

imod( mem8, ax );      // returns "ah"
imod( mem16, dx:ax );  // returns "dx"
imod( mem32, edx:eax ); // returns "edx"

imod( constant8, ax ); // returns "ah"
imod( constant16, dx:ax ); // returns "dx"
imod( constant32, edx:eax ); // returns "edx"

```

6.1.3 cmp 指令

cmp(比较)指令与 sub 指令基本相同,但有一个重要的语义差别:cmp 指令不保留计算结果,而只设置标志寄存器中的条件码位。cmp 指令的语法与 sub 指令相似(但操作数顺序相反,以便于阅读),一般形式为:

```
cmp( LeftOperand, RightOperand );
```

这条指令计算 *LeftOperand - RightOperand*(注意与 sub 指令操作数的顺序相反)的值。具体形式为:

```

cmp( reg, reg );      // Registers must be the same size.
cmp( reg, mem );      // Sizes must match.

```

```
cmp( reg, constant );
cmp( mem, constant );
```

cmp 指令根据减法操作($LeftOperand - RightOperand$)的相应结果更新 80x86 的标志。80x86 采用适当的方式设置标志,因此可以把这条指令读作“比较 $LeftOperand$ 和 $RightOperand$ ”。可以使用条件设置指令(参见 6.1.4 节)或条件跳转指令(参见第 7 章),通过检验标志寄存器中相应的标记查看比较结果。

为了研究 cmp 指令,首先要看看它是如何影响标志的。考虑下面的 cmp 指令:

```
cmp( ax, bx );
```

这条指令完成计算 $AX - BX$ 的操作,并根据计算的结果设置标志。标志的设置情况如下(参见表 6-1):

- Z 当且仅当 $AX=BX$ 时,置位零标志。只有当 $AX - BX$ 的结果为 0 时,这种情况才会发生,因此可以使用零标志检验两个操作数相等或不相等。
- S 当结果是负值时,符号标志被置为 1。由此,您可能认为当 AX 小于 BX 时,这个标志应该被置位,但并不总是这样。如果 $AX=\$7FFF$ 同时 $BX=-1(\$FFFF)$, AX 减去 BX 得到 $\$8000$,是负值(符号标志将置位)。因此,对于有符号数值的比较,符号标志包含的并不是正确的状态。对于无符号操作数,考虑 $AX=\$FFFF$ 和 $BX=1$ 。 AX 大于 BX ,但它们的差 $\$FFFE$ 仍然是一个负数。从中我们可以得出,同时使用符号标志和溢出标志可以对两个有符号数进行比较。
- O cmp 操作后,如果 AX 和 BX 的差产生上溢或下溢,则置位溢出标志。如上所述,符号标志和溢出标志共同用于执行有符号数的比较。
- C cmp 操作,如果 AX 减去 BX 需要借位,则置位进位标志。只有当 AX 和 BX 都是无符号数值,并且 AX 小于 BX 时,这种情况才会发生。

因为 cmp 指令以上面的方式设置标志,所以可以通过下面的标志查看两个操作数的比较结果:

```
cmp( Left, Right );
```

表 6-1 cmp 操作后的条件码设置

无符号操作	有符号操作
Z: 相等/不相等	Z: 相等/不相等
C: $Left < Right (C=1)$ $Left \geq Right (C=0)$	C: 无意义
S: 无意义	S: 参见本节的讨论
O: 无意义	O: 参见本节的讨论

对于有符号数的比较,符号标志 S 和溢出标志 O 共同使用时具有下列的含义:

- 如果 $[(S=0) \text{ 且 } (O=1)]$ 或者 $[(S=1) \text{ 且 } (O=0)]$, 则 $Left < Right$ 。
- 如果 $[(S=0) \text{ 且 } (O=0)]$ 或者 $[(S=1) \text{ 且 } (O=1)]$, 则 $Left \geq Right$ 。

注意, 当操作数 Left 小于操作数 Right 时, (S 异或 O) 为 1。相反, 当操作数 Left 大于等于操作数 Right 时, (S 异或 O) 为 0。

为了理解这两个标志的置位方式, 考虑下面的示例:

Left	减	Right	S	O
-----		-----	-	-
\$FFFF (-1)	-	\$FFFE (-2)	0	0
\$8000	-	\$0001	0	1
\$FFFE (-2)	-	\$FFFF (-1)	1	0
\$7FFF (32767)	-	\$FFFF (-1)	1	1

应该记得, `cmp` 操作实际上是一个减法。因此, 上面第 1 个示例计算 $(-1) - (-2)$, 结果是 $(+1)$ 。计算结果为正同时没有发生溢出, 因此标志 S 和 O 同时为 0。由于 (S 异或 O) 为 0, 所以 Left 大于或等于 Right。

在第 2 个示例中, `cmp` 指令计算 $(-32768) - (+1)$, 结果为 (-32769) 。因为 16 位有符号整数不能表示这个数值, 因此该数值回绕为 $\$7FFF(+32767)$ 同时置位溢出标志。结果为正数(至少作为一个 16 位数值来说如此), 所以 CPU 清除符号标志。这时 (S 异或 O) 为 1, 所以 Left 小于 Right。

在第 3 个示例中, `cmp` 计算 $(-2) - (-1)$, 结果为 (-1) 。没有发生溢出, 所以 O 标志为 0。结果是负数, 所以符号标志为 1。此时 (S 异或 O) 为 1, Left 小于 Right。

在第 4 个(最后)示例中, `cmp` 计算 $(+32767) - (-1)$ 。产生的结果为 $(+32768)$, 置位溢出标志。同时, 由于数值回绕为 $\$8000(-32768)$, 因此符号标志也被置位。这时 (S 异或 O) 为 0, Left 大于等于 Right。

`cmp` 指令执行后, 可以使用 HLA 的高级控制语句和布尔标志表达式检验标志(如 `@c`、`@nc`、`@z`、`@nz`、`@o`、`@no`、`@s`、`@ns` 等)。表 6-2 列出了 HLA 所支持的布尔表达式, 用于检验比较指令执行后的各种条件。

表 6-2 HLA 条件码布尔表达式

HLA 语法	条 件	注 释
<code>@c</code>	进位设置	如果第 1 个操作数小于第 2 个操作数(无符号), 则进位标志被置位。与 <code>@b</code> 和 <code>@nae</code> 的条件相同
<code>@nc</code>	进位清除(无进位)	如果第 1 个操作数大于或等于第 2 个操作数, 则清除进位标志(使用无符号比较)。与 <code>@nb</code> 和 <code>@ae</code> 的条件相同
<code>@z</code>	零标志设置	如果第 1 个操作数与第 2 个操作数相等, 则置位零标志。与 <code>@e</code> 的条件相同
<code>@nz</code>	零标志清除(无零)	如果第 1 个操作数与第 2 个操作数不相等, 则清除零标志。与 <code>@ne</code> 的条件相同
<code>@o</code>	溢出标志设置	如果比较操作的结果有一个有符号的算术溢出, 则置位溢出标志
<code>@no</code>	溢出标志清除(无溢出)	如果在比较操作中没有有符号算术溢出, 则清除溢出标志
<code>@s</code>	符号标志设置	如果比较操作(减法)产生的结果为负, 则置位符号标志
<code>@ns</code>	符号标志清除(无符号)	如果比较操作产生非负(0 或正)结果, 则清除符号标志

(续表)

HLA 语法	条 件	注 释
@a	超过(无符号的大于)	条件@a 检验进位标志和零标志,看是否有@c=0 并且@z=0 成立。如果第 1 个(无符号)操作数大于第 2 个(无符号)操作数,则该条件存在。这与@nbc 的条件相同
@na	不超过	条件@na 查看进位标志(@c)或零标志(@z)是否被置位。这与无符号的“不大于”条件是等价的。注意这个条件与@be 是相同的
@ae	超过或等于(无符号的大于或等于)	如果使用无符号比较得到第 1 个操作数大于或等于第 2 个操作数,则条件@ae 为真。这与条件@nb 和@nc 是等价的
@nae	不超过或等于	如果使用无符号比较得到第 1 个操作数不大于或等于第 2 个操作数,则条件@nae 为真。这与条件@b 和@c 是等价的
@b	低于(无符号的小于)	如果使用无符号比较得到第 1 个操作数小于第 2 个操作数,则条件@b 为真。这与条件@nae 和@c 是等价的
@nb	不低于	如果使用无符号比较得到第 1 个操作数不小于第 2 个操作数,则该条件为真。这与条件@nc 和@ae 是等价的
@be	低于或等于(无符号的小于等于)	如果使用无符号比较得到第 1 个操作数小于或等于第 2 个操作数,则条件@be 为真。该条件等价于@na
@nbe	不低于或等于	如果使用无符号比较得到第 1 个操作数不小于或等于第 2 个操作数,则条件@nbe 为真。该条件等价于@a
@g	大于(有符号的大于)	如果使用有符号比较得到第 1 个操作数大于第 2 个操作数,则条件@g 为真。该条件等价于@nle
@ng	不大于	如果使用有符号比较得到第 1 个操作数不大于第 2 个操作数,则条件@ng 为真。这与条件@le 等价
@ge	大于或等于(有符号的大于或等于)	如果使用有符号比较得到第 1 个操作数大于或等于第 2 个操作数,则条件@ge 为真。这与条件@nl 等价
@nge	不大于或等于	如果使用有符号比较得到第 1 个操作数不大于或等于第 2 个操作数,则条件@nge 为真。这与条件@l 等价
@l	小于(有符号的小于)	如果使用有符号比较得到第 1 个操作数小于第 2 个操作数,则条件@l 为真。这与条件@nge 等价
@nl	不小于	如果使用有符号比较得到第 1 个操作数不小于第 2 个操作数则条件@nl 为真。这与条件@ge 等价
@le	小于或等于(有符号)	如果使用有符号比较得到第 1 个操作数小于或等于第 2 个操作数,则条件@le 为真。这与条件@ng 等价
@nle	不小于或等于	如果使用有符号比较得到第 1 个操作数不小于或等于第 2 个操作数,则条件@nle 为真。这与条件@g 等价
@c	等于(有符号或无符号)	如果第 1 个操作数等于第 2 个操作数,则该条件为真。条件@c 等价于条件@z

(续表)

HLA 语法	条 件	注 释
@ne	不等于(有符号或无符号)	如果第 1 个操作数不等于第 2 个操作数，则该条件为真。条件@ne 等价于条件@nz

可以在 if 语句、while 语句或任何其他允许出现布尔表达式的 HLA 高级控制语句中使用表 6-2 中出现的布尔表达式。在紧跟 cmp 指令后的 if 语句中，通常会使用其中的一个条件，例如：

```
cmp( eax,ebx );
if( @e )then

    << Do something if eax = ebx. >>

endif;
```

上面的示例等价于：

```
if( eax = ebx )then

    << Do something if eax = ebx. >>

endif;
```

6.1.4 setcc 指令

条件设置(setcc)指令根据标志寄存器中的数值把一个单字节操作数(寄存器或存储器)置为 0 或 1。setcc 指令的通用格式为：

```
setcc( reg8 );
setcc( mem8 );
```

表 6-3、表 6-4 和表 6-5 中给出了 setcc 指令的助记符。如果条件为假，这些指令把 0 存入相应的操作数中；如果条件为真，则在 8 位操作数中存入 1。

表 6-3 检验标志的 setcc 指令

指 令	描 述	条 件	注 释
setc	如果有进位则置 1	进位标志=1	与 setb、setnae 相同
setnc	如果没有进位则置 1	进位标志=0	与 setnb、setae 相同
setz	如果有零标志则置 1	零标志=1	与 sete 相同
setnz	如果没有零标志则置 1	零标志=0	与 setne 相同
sets	如果有符号标志则置 1	符号标志=1	
setns	如果没有符号标志则置 1	符号标志=0	
seto	如果有溢出则置 1	溢出标志=1	
setno	如果没有溢出则置 1	溢出标志=0	
setp	如果有奇偶校验则置 1	奇偶标志=1	与 setpe 相同

(续表)

指 令	描 述	条 件	注 释
setpe	如果奇偶校验为偶校验则置 1	奇偶标志=1	与 setp 相同
setnp	如果没有奇偶校验则置 1	奇偶标志=0	与 setpo 相同
setpo	如果奇偶校验为奇校验则置 1	奇偶标志=0	与 setnp 相同

上面的 `setcc` 指令只是简单地对标志进行检验,对于操作不附加其他任何含义。例如,可以在移位、循环移位、位检验或算术运算操作后使用 `setc` 检验进位标志。您可能已经注意到,上面的指令 `setp`、`setpe` 和 `setnp` 是用来检验奇偶标志的。为了完整起见,这些指令都被列了出来,但本文并不对奇偶标志作更多的讨论(它的使用已有些过时)。

`cmp` 指令与 `setcc` 指令联合使用。紧跟一个 `cmp` 操作后,处理器标志提供相关操作数的比较结果信息。它们允许您查看一个操作数是否小于、等于或大于另一个操作数。

在 `cmp` 操作后,还有另外两组非常有用的 `setcc` 指令。第一组处理无符号比较的结果(见表 6-4),第二组处理有符号比较的结果。

表 6-4 用于无符号比较的 `setcc` 指令

指 令	描 述	条 件	注 释
seta	如果超过(>)则置 1	进位标志=0, 零标志=0	与 setnbe 相同
setnbe	如果不低于或等于(不<=)则置 1	进位标志=0, 零标志=0	与 seta 相同
setae	如果超过或等于(>=)则置 1	进位标志=0	与 setnc、setnb 相同
setnb	如果不低于(不<)则置 1	进位标志=0	与 setnc、setae 相同
setb	如果低于(<)则置 1	进位标志=1	与 setc、setnae 相同
setnae	如果不超过或等于(不>=)则置 1	进位标志=1	与 setc、setb 相同
setbe	如果低于或等于(<=)则置 1	进位标志=1 或零标志=1	与 setna 相同
setna	如果不超过(不>)则置 1	进位标志=1 或零标志=1	与 setbe 相同
setc	如果等于(=)则置 1	零标志=1	与 setz 相同
setne	如果不等于(<>)则置 1	零标志=0	与 setnz 相同

表 6-5 列出了对应的有符号比较指令。

表 6-5 用于有符号比较的 `setcc` 指令

指 令	描 述	条 件	注 释
setg	如果大于(>)则置 1	符号标志=溢出标志并且 零标志=0	与 setnle 相同
setnle	如果不小于或等于(不<=)则置 1	符号标志=溢出标志或 0 标志=0	与 setg 相同
setge	如果大于或等于(>=)则置 1	符号标志=溢出标志	与 setnl 相同

(续表)

指 令	描 述	条 件	注 释
setnl	如果不小于(不<)则置1	符号标志=溢出标志	与 setge 相同
setl	如果小于(<)则置1	符号标志<溢出标志	与 setnge 相同
setnge	如果不大于或等于(不>=)则置1	符号标志<溢出标志	与 setl 相同
setle	如果小于或等于(<=)则置1	符号标志<溢出标志或 零标志=1	与 setng 相同
setng	如果不大于(不>)则置1	符号标志<溢出标志或 零标志=1	与 setle 相同
sete	如果等于(=)则置1	零标志=1	与 setz 相同
setne	如果不等于(<>)则置1	零标志=0	与 setnz 相同

注意 setcc 指令和可能出现在布尔指令中的 HLA 标志条件之间的对应关系。

setcc 指令具有其重要价值,因为它们能够把一个比较结果转换为一个布尔数值(false/true 或 0/1)。当把 Pascal 或 C/C++ 这样的高级语言语句转换为汇编语言时,这点尤为重要。下面的示例给出了在这种方式下如何使用这些指令:

```
// bool := a <= b
mov( a, eax );
cmp( eax, b );
setle( bool ); // bool is a boolean or byte variable.
```

由于 setcc 指令总是产生 0 或 1,因此可以把结果用于 and 和 or 指令来计算复杂的布尔数值:

```
// bool := ((a <= b) and (d = e))
mov( a, eax );
cmp( eax, b );
setle( bl );
mov( d, eax );
cmp( eax, e );
sete( bh );
and( bl, bh );
mov( bh, bool );
```

6.1.5 test 指令

80x86 的 test 指令和 and 指令的关系与 cmp 指令和 sub 指令的关系类似。也就是说, test 指令对它的两个操作数进行逻辑与运算,并根据运算的结果对条件码标志置位。然而,它并不把逻辑与运算的结果保存到目的操作数中。test 指令的语法与 and 指令相似,如下所示:

```
test( operand1, operand2 );
```

如果逻辑与运算的结果为 0,则 test 指令设置零标志。如果结果的高位包含 1,则 test 指令

设置符号标志。test 指令总是清除进位标志和溢出标志。

test 指令主要用于检验一个位包含 0 还是 1。考虑指令 test(1,al);, 这条指令对 AL 和数值 1 进行逻辑与运算。如果 AL 的第 0 位数值为 0, 则结果为 0(置位零标志), 因为常量 1 的其他位都是 0。相反, 如果 AL 的第 0 位包含 1, 则结果不是 0, 此时 test 指令清除零标志。因此, 当这条 test 指令执行完, 可以通过检验零标志来查看第 0 位包含的是数值 0 还是 1(如使用指令 setz 或 setnz)。

test 指令还可以用于检验一组指定的位是否都包含 0。例如, 当且仅当 AL 的低 4 位都包含 0 时, 指令 test(\$F,al);才置位零标志。

test 指令的一个主要用途是查看一个寄存器是否包含 0。指令 test(reg,reg);中两个操作数是同一个寄存器, 它使得这个寄存器与自身进行逻辑与运算。如果寄存器包含的是 0, 则运算结果为 0, 同时 CPU 将置位零标志。然而, 如果这个寄存器有一位含有非 0 数值, 这个数值与自身进行逻辑与运算产生同样的非 0 数值, 此时 CPU 清除零标志。因此, 这条指令执行后, 可以立即检验零标志(如使用指令 setz 或 setnz 或者布尔条件 @z 和 @nz)来确定这个寄存器中是否含有 0。例如:

```
test( eax, eax );
setz( bl );           // bl is set to 1 if eax contains 0.

.
.
.

test( bx, bx );
if( @nz )then
    << Do something if bx <> 0. >>
endif;
```

6.2 算术表达式

汇编语言的初学者可能对缺乏算术表达式感到很意外。在大多数高级语言中, 算术表达式与它们的代数形式类似, 例如:

```
x := y * z;
```

在汇编语言中, 则需要使用几条语句来完成同样的任务, 例如:

```
mov( y, eax );
intmul( z, eax );
mov( eax, x );
```

显然, HELL(高级语言)更易于书写、阅读和理解。与其他原因相比, 这点可能是人们不愿学习汇编语言的主要原因。虽然会涉及大量输入工作, 但把一个算术表达式转换成汇编语言程序并不困难。通过一步步地分析问题, 很容易把算术表达式转换成等价的汇编语言语句序列。在这个过程中, 也将学会如何使用相同的方法手工解决这种问题。我们分三步来学习如何把一个算术表达式转换成汇编语言语句, 同时您会发现这并不难。

6.2.1 简单赋值

最容易转换为汇编语言的表达式是简单赋值。简单赋值采用下列两种形式之一把一个数值复制到一个变量中：

```
variable := constant
```

或

```
var1 := var2
```

把第一种形式转换为汇编语言很简单，使用下面这条汇编语言语句：

```
mov( constant, variable );
```

mov 指令把常量复制到变量中。

上面第二种形式的赋值语句就有些复杂，因为 80x86 没有提供存储器到存储器的 mov 指令。因此，必须通过寄存器把一个存储器变量复制到另一个存储器变量中。作为一种约定(同时考虑到效率方面的原因)，大多数编程人员使用 AL、AX 或 EAX 来完成这种功能。例如：

```
var1 := var2;
```

变为

```
mov( var2, eax );  
mov( eax, var1 );
```

当然，此时假定 var1 和 var2 都是 32 位变量。如果是 8 位变量，则使用 AE；如果是 16 位变量，则使用 AX。

如果 AL、AX 或 EAX 已被用于别处，则可以使用其他的寄存器。但不管怎样，都必须使用一个寄存器来完成存储单元间的数值转移。

6.2.2 简单表达式

比简单赋值稍微复杂一点的是简单表达式。简单表达式的形式为：

```
var1 := term1 op term2;
```

其中 var1 是一个变量，term1 和 term2 是变量或常量，op 是算术操作符(加、减、乘等)。大多数表达式都采用这种形式。这也就不难理解，80x86 体系结构对这种类型的表达式进行了优化。

针对这种类型表达式的一个典型变换形式为：

```
mov( term1, eax );  
op( term2, eax );  
mov( eax, var1 );
```

其中 op 是对应指定操作的助记符(例如，+ 为 add，- 为 sub 等)。

注意，简单表达式 var1:=const1 op const2; 很容易通过一个编译时表达式和一条 mov 指令进行

转换。例如,要计算 $var1:=5+3$; 只需要使用指令 `mov(5+3,var1);`。

对于一些不一致的情况必须清楚。当处理 80x86 的(i)mul、(i)div 和(i)mod 指令时,必须使用寄存器 AL/AX/EAX 和寄存器 DX/EDX。不能像对其他操作那样使用任意的寄存器。同样,当执行两个 16/32 位数值的除法操作时,不要忘记使用符号扩展指令。最后,还要注意一些指令可能会导致溢出。当一个算术运算操作完成后,可以检验一下上溢(或下溢)条件。

常用简单表达式的示例如下:

```
x := y + z;

mov( y, eax );
add( z, eax );
mov( eax, x );

x := y - z;

mov( y, eax );
sub( z, eax );
mov( eax, x );

x := y * z; {unsigned}

mov( y, eax );
mul( z, eax );      // Don't forget this wipes out edx.
mov( eax, x );

x := y * z; {signed}

mov( y, eax );
imul( z, eax );      // Does not affect edx!
mov( eax, x );

x := y div z; {unsigned div}

mov( y, eax );
mov( 0, edx );      // Zero extend eax into edx.
div( z, edx:eax );
mov( eax, x );

x := y idiv z; {signed div}

mov( y, eax );
cdq();              // Sign extend eax into edx.
idiv( z, edx:eax );
mov( eax, x );

x := y mod z; {unsigned remainder}

mov( y, eax );
mov( 0, edx );      // Zero extend eax into edx.
mod( z, edx:eax );
mov( edx, x );      // Note that remainder is in edx.

x := y imod z; {signed remainder}
```

```

mov( y, eax );
cdq();                // Sign extend eax into edx.
imod( z, edx:eax );
mov( edx, x );        // Remainder is in edx.

```

某些一元操作也被认为是简单表达式。一元操作的典型示例是取负操作。在高级语言中，取负操作有下列两种形式：

```
var := -var
```

或

```
var1 := -var2
```

需要注意的是，`var := - constant` 是一个简单赋值而不是一个简单表达式。可以指定负数常量作为 `mov` 指令的一个操作数：

```
mov( -14, var );
```

可以使用简单的汇编语言语句处理 `var1 = - var1;`：

```

// var1 = -var1;
neg( var1 );

```

如果涉及两个不同的变量，则使用下列语句序列：

```

// var1 = -var2;

mov( var2, eax );
neg( eax );
mov( eax, var1 );

```

6.2.3 复杂表达式

复杂表达式是指含有两个以上操作数和一个操作符的算术表达式。这种表达式在高级语言程序中很常见。复杂表达式可能会含有圆括号以重写操作符的优先顺序、实现函数调用和对数组的访问等。许多复杂表达式向汇编语言的转换相当直接，而其他的则要费一番功夫。本节内容概述用于转换这种表达式的一些规则。

一个只涉及三个操作数和两个操作符的复杂表达式很容易被转换为汇编语言程序，例如：

```
w := w - y - z;
```

显然，把这条语句直接转换为汇编语言需要两条 `sub` 指令。然而，即使对于像这样简单的表达式，转换也不是轻而易举的。实际上，把上面这条语句转换为汇编语言有两种方式：

```

mov( w, eax );
sub( y, eax );
sub( z, eax );
mov( eax, w );

```

和

```
mov( y, eax );
sub( z, eax );
sub( eax, w );
```

第二个转换看起来更简洁，所以让人感觉更好。然而，它产生的却是错误的结果(假设对原始表达式使用类似 Pascal 的语义)。问题在于运算的结合性。第二个语句序列计算的是 $w := w - (y - z)$ ，这与 $w := (w - y) - z$ 是不同的。如何在子表达式中使用圆括号会影响计算的结果。如果喜欢使用简洁的形式，可以使用下面的序列：

```
mov( y, eax );
add( z, eax );
sub( eax, w );
```

它计算的是 $w := w - (y+z)$ ，这与 $w := (w - y) - z$ 是等价的。

另一个问题是关于运算的优先权。考虑下面的 Pascal 表达式：

```
x := w * y + z;
```

同样可以用两种方法来计算这个表达式的值：

```
x := (w * y) + z;
```

或

```
x := w * (y + z);
```

也许您会认为这再简单不过了。每个人都知道这个表达式的正确计算方法是第一种形式。然而，直观地这样想是错误的。例如，APL 编程语言对表达式的计算是从右向左的，同时没有给出运算符之间的任何优先权。因此，哪种方法是“正确的”完全取决于运算系统的优先权是如何定义的。

多数高级语言使用固定的优先权规则集合来描述包含两个或多个不同运算符的表达式运算顺序。这样的编程语言对乘法和除法的计算通常优先于加法和减法的计算。对于那些支持求幂运算的高级语言(如 FORTRAN 和 BASIC)，求幂运算优先于乘法和除法。这些规则很直观，因为几乎每个人在读中学之前就已经学习了这些规则。考虑一下下面这个表达式：

```
x op1 y op2 z
```

如果 $op1$ 优先于 $op2$ ，则这个表达式等价于 $(x op1 y) op2 z$ ；同样地，如果 $op2$ 优先于 $op1$ ，那么表达式等价于 $x op1 (y op2 z)$ 。依赖于所包含的运算符和操作数，这两种计算形式会产生不同的结果。当把这样的表达式转换成汇编语言时，必须确保首先计算的是具有最高优先权的子表达式。下面的示例对这种技术进行了说明：

```
// w := x + y * z;

mov( x, ebx );
```

```

mov( y, eax );           // Must compute y * z first because "*"
intmul( z, eax );        // has higher precedence than "+".
add( ebx, eax );
mov( eax, w );

```

如果表达式中的两个运算符具有相同的优先权，那么要根据结合性规则决定它们的运算顺序。大多数运算符是左结合的，意思是它们按照从左至右的顺序计算。加法、减法、乘法和除法都是左结合的。右结合的运算符按照从右到左的顺序计算。FORTRAN 语言和 BASIC 语言中的求幂运算符是一个典型的右结合运算符：

2^{2^3} 等价于 $2^{(2^3)}$ 而不是 $(2^2)^3$

优先权和结合性规则决定运算的顺序。这些规则间接地告诉您在一个表达式中的什么位置放置括号来决定运算顺序。当然，可以使用括号来改变默认的优先权和结合性。然而，汇编代码最终必须能够在其他操作之前完成某些特定的操作，以正确地计算出给定表达式的值。下面的示例说明了这种原理：

```

// w := x - y - z

mov( x, eax );           // All the same operator, so we need
sub( y, eax );           // to evaluate from left to right
sub( z, eax );           // because they all have the same
mov( eax, w );           // precedence and are left associative.

// w := x + y * z

mov( y, eax );           // Must compute y * z first because
intmul( z, eax );        // multiplication has a higher
add( x, eax );           // precedence than addition.
mov( eax, w );

// w := x / y - z

mov( x, eax );           // Here we need to compute division
cdq();                   // first because it has the highest
idiv( y, edx:eax );      // precedence.
sub( z, eax );
mov( eax, w );

// w := x * y * z

mov( y, eax );           // Addition and multiplication are
intmul( z, eax );        // commutative, therefore the order
intmul( x, eax );        // of evaluation does not matter.
mov( eax, w );

```

对于结合性规则有一个例外。如果一个表达式同时含有乘法运算和除法运算，先完成乘法运算通常更好。例如，下面的表达式：

$w := x / y * z$ // Note: This is $(x * z) / y$, not $x / (y * z)$.

它通常是首先计算 $x * z$ ，然后再除以 y ，而不是用 x 除以 y ，再用得到的商与 z 相乘。有两个原因可以说明这种方法比较好。首先，应该记得 `imul` 指令会产生一个 64 位的结果(假设操作数是 32 位的)。首先对乘法进行运算，这会自动在 `EDX` 中对结果进行符号扩展，而不必在做除法之前对 `EAX` 进行符号扩展。首先运算乘法的第二个原因是，这样可以提高计算精度。(整数)除法经常产生不精确的结果。例如，如果计算 $5/2$ 将得到结果 2 而不是 2.5。对 $(5/2)*3$ 计算将得到结果 6。然而，如果计算 $(5*3)/2$ 会得到结果 7，更接近结果的精确值(7.5)。因此，如果遇到一个如下形式的表达式：

$$w := x / y * z;$$

可以把它转换为下列汇编代码：

```
mov( x, eax );
imul( z, eax ); // Note the use of imul, not intmul!
idiv( y, edx:eax );
mov( eax, w );
```

当然，如果您的编码算法取决于对除法操作的截断结果，则不能使用这种技巧来改进算法。切记，一定要确保完全理解了您要转换为汇编语言的那个表达式。很明显，如果语义指示必须首先完成除法操作，那么就只能这样做。

考虑下面这条 Pascal 语句：

$$w := x - y * z;$$

它与前面的示例类似，不同之处在于它使用的是减法而不是加法。因为减法是不可交换的，所以不能先计算 $y * z$ ，然后再用这个结果减去 x 。这使得转换过程变得复杂。与一个直接的乘法和加法序列不同，您必须先把 x 读入一个寄存器，然后对 y 和 z 做乘法，并把得到的结果放在另一个寄存器中，最后从 x 中减去这个结果。例如：

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

这个简单的示例说明了在表达式中需要用到临时变量(temporary variable)。这段代码使用寄存器 `EBX` 临时保存 x 的值，直到计算出 y 和 z 相乘的结果。随着表达式复杂性的增加，对临时变量的需要也随之增长。考虑下面这条 Pascal 语句：

$$w := (a + b) * (y + z);$$

按照通常的代数运算规则，将首先计算括号中的子表达式(两个子表达式具有最高的优先权)并把得到的结果放在一边。当两个子表达式的结果都计算出来后，可以对它们的乘积再进行计算。处理这种复杂表达式的一种方法是把它简化为一个简单表达式的序列，它们的中间结果保存在临时变量中。例如，可以把上面这个表达式转换成下面的序列：

```

temp1 := a + b;
temp2 := y + z;
w := temp1 * temp2;

```

把简单表达式转换为汇编语言很容易，现在就可以把前面的复杂表达式转换为汇编语言。其代码为：

```

mov( a, eax );
add( b, eax );
mov( eax, temp1 );
mov( y, eax );
add( z, eax );
mov( eax, temp2 );
mov( temp1, eax );
intmul( temp2, eax );
mov( eax, w );

```

当然，这段代码效率很低，方法是它还需要您在数据段声明两个临时变量。然而，很容易对这段代码进行优化，方法是尽可能地在 80x86 的寄存器中保存临时变量。通过使用 80x86 的寄存器保存临时结果，这段代码变为：

```

mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );

```

还有另一个示例：

```

x := (y + z) * (a - b) / 10;

```

它可以被转换为由 4 个简单表达式组成的序列：

```

temp1 := (y + z)
temp2 := (a - b)
temp1 := temp1 * temp2
X := Temp1 / 10

```

把这 4 个简单表达式转换为汇编语言语句：

```

mov( y, eax );           // Compute eax = y + z
add( z, eax );
mov( a, ebx );           // Compute ebx = a - b
sub( b, ebx );
imul( ebx, eax );        // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );

```

需要谨记的一点是，应该把临时数值保存在寄存器中。应该记得，访问 80x86 的一个寄存器

要比访问一个存储单元效率高得多。只有当可用的寄存器都被用完时，才使用存储单元来保存临时结果。

最后，把一个复杂表达式转换为汇编语言与手工计算这个表达式几乎相同。区别在于，不是每一步都计算出实际结果，而是简单地写出汇编代码来计算它们。因为人们通常每次只计算一个操作，这意味着手工计算针对复杂表达式中的“简单表达式”。把那些简单表达式转换成汇编语言是轻而易举的。因此，能够使用手工方法解决一个复杂表达式的人，只要遵循简单表达式的转换规则，都可以把一个复杂表达式转换成汇编语言。

6.2.4 可交换运算符

假设用 *op* 代表一种运算符，如果下面的关系式总为真，则我们说这个运算符是可交换的 (commutative):

(A op B) = (B op A)

正如在前面的内容中看到的，可交换运算符的操作数顺序是无关紧要的，这就允许对一个计算进行重新排列，从而使得某个计算更容易或更高效。而且，对一个计算重新排列可以减少临时变量的使用。当在某个表达式中遇到一个可交换运算符时，应该检查是否可以使用更好的排序改善代码量或改进代码速度。表 6-6 和表 6-7 分别列出了高级语言中常用的可交换和不可交换运算符。

表 6-6 常用的可交换二元运算符

Pascal	C/C++	描 述
+	+	加法
*	*	乘法
and	&&或&	逻辑与或按位与
or	或	逻辑或或按位或
xor	^	逻辑异或或按位异或
=	==	等于
<>	!=	不等于

表 6-7 常用的不可交换二元运算符

Pascal	C/C++	描 述
-	-	减法
/或 div	/	除法
mod	%	取模或取余
<	<	小于
<=	<=	小于或等于
>	>	大于
>=	>=	大于或等于

6.3 逻辑(布尔)表达式

考虑下面这个Pascal程序中的表达式:

```
b := ((x = y) and (a <= c)) or ((z - a) <> 5);
```

其中b是布尔变量,而其他变量都是整数。

我们如何在汇编语言中表示一个布尔变量呢?虽然只用一位就可以表示一个布尔值,但多数汇编语言编程人员习惯于分配一个整字节或一个字(同样地,HLA也为一个布尔变量分配一个整字节)。对于一个字节,有256个可以用来表示真和假的可能取值,那么我们应该选择哪两个数值(或哪两组数值)表示这些布尔值?由于机器体系结构的原因,检验0或非0、正数或负数这样的条件要比检验一两个特定的布尔值容易得多。多数编程人员(事实上,还有类似于C的编程语言)选择0表示假,而任何其他值都表示真。有些人愿意使用1和0来分别表示真和假,而不允许任何其他数值表示布尔值。也可以选择全1(\$FFFF_FFFF、\$FFFF或\$FF)代表真,0代表假。或者还可以选择正数表示真,负数表示假。这些机制都有其自身的优缺点。

只使用0和1表示表示逻辑真假有两大优点:①setcc指令会产生这样的结果,所以这种方式与setcc指令是兼容的;②80x86的逻辑指令(and、or、xor和not)可以按您所希望的那样对这些逻辑数值进行正确的操作。也就是说,如果有两个布尔变量A和B,则下面的指令将对这两个变量完成基本的逻辑操作:

```
// c = a AND b;
```

```
mov( a, al );
and( b, al );
mov( al, c );
```

```
// c = a OR b;
```

```
mov( a, al );
or( b, al );
mov( al, c );
```

```
// c = a XOR b;
```

```
mov( a, al );
xor( b, al );
mov( al, c );
```

```
// b = NOT a;
```

```
mov( a, al );           // Note that the NOT instruction does not
not( al );              // properly compute al = NOT al by itself.
and( 1, al );           // I.e., (NOT 0) does not equal one. The AND
mov( al, b );           // instruction corrects this problem.

mov( a, al );           // Another way to do b = NOT a;
xor( 1, al );           // Inverts bit 0.
mov( al, b );
```

注意，正如在前述代码中指出的，指令 `not` 不会正确地计算逻辑非。对 0 的按位取反操作得到 \$FF，而对 1 按位取反得到 \$FE。这两个结果都不是 0 或 1。然而，使用 1 和结果进行 `and` 操作可以得到正确的结果。注意，可以使用指令 `xor(1,ax)` 更高效地实现 `not` 操作，因为它只影响低位位。

我们已经看到，用 0 作为逻辑假而其他任何值都作为真的方法有一些独特的优点。特别是在逻辑指令的执行中，对真假值的检验经常是隐式的。然而，这种机制同时还存在一个很大的缺陷：不能使用 80x86 的 `and`、`or`、`xor` 和 `not` 指令实现同名的布尔操作。考虑两个数值 \$55 和 \$AA，这两个值都是非 0 的，因此它们都代表真值。然而，如果使用 80x86 的 `and` 指令对 \$55 和 \$AA 进行逻辑 `and` 操作，结果是 0。真值与真值进行 `and` 操作应该得到真而不是假。虽然可以对这种情况作出一定的解释，但这样通常需要一些额外的指令，而且对布尔操作的计算效率不高。

使用非 0 数值代表真值，使用 0 代表假值的系统称为算术逻辑系统(arithmetic logical system)。而明确使用 1 和 0 分别代表真假值的系统称为布尔逻辑系统(boolean logical system)，或简称布尔系统。可以使用任何一种您认为最方便的系统。我们再来看布尔表达式：

```
b := ((x = y) and (a <= d)) or ((z - a) <> 5);
```

从这个表达式得到的转换结果为：

```
mov( x, eax );
cmp( y, eax );
sete( al );           // al := x = y;

mov( a, ebx );
cmp( ebx, d );
setle( bl );          // bl := a <= d;
and( al, bl );        // bl := (x = y) and (a <= d);

mov( z, eax );
sub( a, eax );
cmp( eax, 5 );
setne( al );
or( bl, al );         // al := ((x = y) and (a <= d)) or ((z - a) <> 5);
mov( al, b );
```

当处理布尔表达式时，不要忘记可以通过对布尔表达式进行简化来优化代码。也可以使用代数变换来简化一个复杂的表达式。在有关控制结构的章节，您将看到如何使用控制流来计算一个布尔结果。这种方法比本节所讲述的完整布尔求值(complete boolean evaluation)更高效。

6.4 机器特性与运算技巧

技巧是一种特性。当编写汇编语言代码时，算术运算操作和 80x86 的指令具有很多可以利用的特性。许多人把使用机器特性和运算技巧的编程方法称为“巧妙编程”，而在良好的程序中应该避免使用这种方法。虽然避免为了技巧本身而使用技巧是明智的，但很多机器特性和运算技巧在汇编语言程序中很知名而且很常见。其中不乏有一些是投机取巧，但大多数只是简单的“技巧运用”。本文不可能阐述当前所有的惯用技巧，它们太多了，而且还在不断地变化。尽管如此，还是

有许多很重要的惯用技巧是您必须知道的, 因此对它们的讨论就变得很有意义。

6.4.1 不使用 mul、imul 或 intmul 的乘法

当计算一个含有常量的乘法时, 可以使用移位、加法和减法来实现乘法操作, 这样可以编写出更快的代码。

记住, 一条 shl 指令计算的结果与指定操作数和 2 所进行的乘法结果相同。同样, 左移 2 位等于对操作数乘 4, 左移 3 位等于对操作数乘 8。一般而言, 把一个操作数左移 n 位与操作数和 2^n 相乘的结果相同。对于任何包含常量的乘法运算都可以使用一系列的移位和加法组合或者移位和减法组合来实现。例如, 如果要计算寄存器 AX 与 10 的乘法, 只须让 AX 先与 8 相乘, 再与两倍的原始数据相加。也就是说, $10*ax=8*ax+2*ax$ 。完成这段操作的代码为:

```
shl( 1, ax );           // Multiply ax by two.
mov( ax, bx );          // Save 2*ax for later.
shl( 2, ax );           // Multiply ax by eight (*4 really,
                        // but ax contains *2).
add( bx, ax );          // Add in ax*2 to ax*8 to get ax*10.
```

多数 x86 处理器使用指令 shl 处理寄存器 AX(或其他任何寄存器)与不同常量的乘法要比使用指令 mul 快很多。这看起来似乎很难理解, 因为乘法指令只使用一条指令计算结果:

```
intmul( 10, ax );
```

然而, 如果仔细观察指令时序您就会发现, 在 80x86 系列的处理器中, 移位和加法所需要的时钟周期要比 mul 指令短。当然, 代码量有所增加(少许字节), 但这与性能的改进相比是值得的。

还可以使用减法与移位相结合的方法实现乘法操作。考虑下面这个与 7 的乘法:

```
mov( eax, ebx );        // Save eax * 1
shl( 3, eax );           // eax = eax * 8
sub( ebx, eax );         // eax * 8 - eax * 1 is eax * 7
```

对于汇编语言的初学者来说, 经常容易出现的错误是对结果加减 1 或 2, 而不是 $eax*1$ 或 $eax*2$ 。下面的代码所计算的就不是 $eax*7$:

```
shl( 3, eax );
sub( 1, eax );
```

它计算的是 $(8*eax) - 1$, 一个完全不同的结果(当然, 除非 EAX=1)。当使用移位、加法和减法实现乘法操作时要当心这种错误发生。

还可以使用 lea 指令计算某些乘积。这种技巧是对比例变址寻址方式的运用。下面的示例说明了一些简单情况:

```
lea( eax, [ecx][ecx] ); // eax := ecx * 2
lea( eax, [eax][eax*2] ); // eax := eax * 3
lea( eax, [eax*4] );     // eax := eax * 4
lea( eax, [ebx][ebx*4] ); // eax := ebx * 5
lea( eax, [eax*8] );     // eax := eax * 8
lea( eax, [edx][edx*8] ); // eax := edx * 9
```

6.4.2 不使用 div 或 idiv 的除法

如同 `shl` 指令可被用来模拟一个与 2 的幂相乘的操作一样，可以使用 `shr` 和 `sar` 指令模拟一个与 2 的幂的除法。不过，虽然很容易使用移位、加法和减法指令实现一个乘法操作，但对于任意常量作除数的除法却不能这样。因此，切记这个技巧只适用于 2 的幂作除数的除法。还有不要忘记，指令 `sar` 舍入负无穷大而不是 0，这与指令 `idiv` 的操作不同(它舍入为 0)。

另一种实现除法的方法是使用乘法指令。可以通过与某个数值的倒数相乘实现对这个数值的除法。由于乘法指令要比除法指令执行速度快，所以与倒数进行乘法通常比除法快。

您马上会产生疑问：“我们处理的都是整数，如何对它们的倒数进行乘法？”答案是我们必须通过很巧妙的方法实现它。如果想要与 1/10 做乘法，没有方法可以在执行乘法操作之前把数值 1/10 装入 80x86 的寄存器中。然而，我们可以对 1/10 和 10 做乘法，执行完以后再用 10 对计算的结果做除法得到最终结果。当然，这样根本没有带来任何好处，事实上，可能使事情更糟，因为这样用 10 做了一次乘法的同时又用它做了一次除法。然而，设想把 1/10 与 65536 相乘(6553)，乘法完成后，让结果被 65536 除。这样仍然完成正确的操作，事实上，如果问题处理正确的话可以省去除法操作。考虑下面 AX 与 10 进行除法运算的代码：

```
mov( 6554, dx );    // 6554 = round( 65536/10 ).
mul( dx, ax );
```

这段代码执行完成后，寄存器 DX 中保存的是 AX/10 的值。

为了理解上面的代码是如何工作的，思考当使用寄存器 AX 与 65536(\$1_0000)相乘时发生了什么。它只是简单地把 AX 中的值移入 DX 中，同时把 AX 置为 0(与 \$10000 做乘法等价于把被乘数左移 16 位)。因此与 6554(65536/10)做乘法就把 AX/10 的结果放在了寄存器 DX 中。由于 `mul` 指令比 `div` 指令执行速度快，这种做法将比直接使用除法操作快。

当需要对一个常量做除法时，与倒数相乘的方法可以很好地完成这种操作。您或许还想使用这种方法来完成对变量的除法，但是只有当多次执行对同一个数值的除法时，计算变量倒数的开销才能够抵消。

6.4.3 使用 and 实现模 N 计数器

如果想要实现一个计数器变量，当计数到 $2^n - 1$ 后复位为 0，可以使用下面的代码：

```
inc( CounterVar );
and( nBits, CounterVar );
```

其中 `nBits` 是一个二进制数值，它含有 n 个右对齐的 1。例如，要创建一个在 0~15($2^4 - 1$)之间循环的计数器，可以使用下面的代码：

```
inc( CounterVar );
and( %00001111, CounterVar );
```

6.5 浮点运算

20 世纪 70 年代末, 8086 CPU 首次出现的时候, 半导体技术还没有达到能够让 Intel 把浮点指令直接放入 8086 CPU 的程度。因此, Intel 设计了一种使用另一块芯片实现浮点运算的方案——浮点运算单元(floating-point unit, FPU)¹。当 Intel Pentium 芯片发布时, 半导体技术已经有了长足的进步, 使得 FPU 可以完全集成到 80x86 CPU 上。因此, 几乎所有现代 80x86 CPU 设备都完全支持在 CPU 上直接进行浮点运算。

6.5.1 FPU 寄存器

80x86 的 FPU 在 80x86 的处理器中增加了 13 个寄存器: 8 个浮点数据寄存器、1 个控制寄存器、1 个状态寄存器、1 个标志寄存器、1 个指令指针和 1 个数据指针。数据寄存器与 80x86 的通用寄存器类似, 只是范围设定为所有浮点运算进行的地方。控制寄存器包含的位可以用来判定 FPU 是如何处理某些退化情况, 如不精确计算的舍入, 还包含用于控制精度的位等。状态寄存器与 80x86 的标志寄存器类似, 它包含条件码位和其他一些用于描述 FPU 状态的浮点标志。标志寄存器包含几组位信息, 这些信息用于确定 8 个浮点数据寄存器中值的状态。指令和数据指针寄存器包含最后执行的浮点指令的某些状态信息。本文不对最后 3 个寄存器进行讨论, 要获得更多详细资料请参见 Intel 文档。

1. FPU 的数据寄存器

FPU 提供 8 个按栈方式组织起来的 80 位寄存器。这与 80x86 CPU 上通用寄存器结构的组织有很大的区别。HLA 把这些寄存器称为 ST0、ST1、...、ST7。

FPU 寄存器组与 80x86 寄存器组之间最大的不同在于栈的结构。在 80x86 CPU 中, 不管如何操作, AX 寄存器就总是 AX 寄存器。而在 FPU 中, 寄存器组是一个 8 元素栈, 栈的每一个元素包含 80 位的浮点数值(参见图 6-1)。

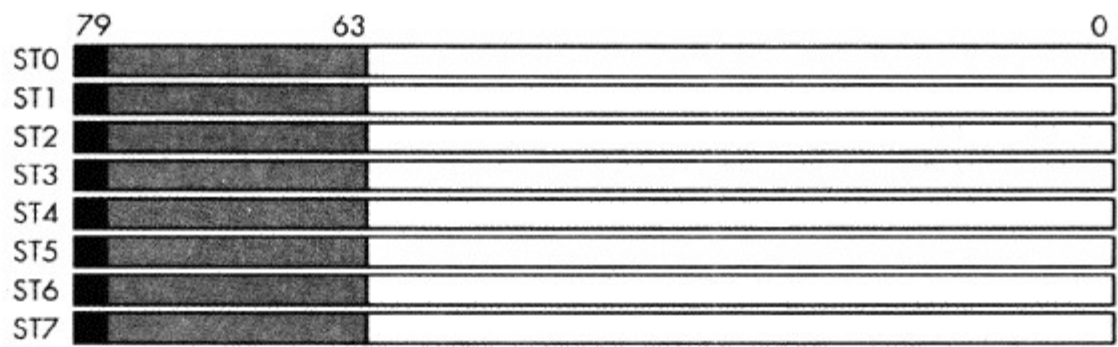


图 6-1 FPU 的浮点寄存器栈

其中 ST0 指示栈顶的项, ST1 指示下一项, 依此类推。许多浮点操作指令会向栈推入或弹出一些项。因此, 当在栈顶推入新的内容时, ST1 将指示先前由 ST0 所指示的内容。需要经过一些思考和实践才会习惯寄存器数改变的事实, 但这是一个容易克服的问题。

¹ Intel 还把这个设备叫做数值数据处理器(Numeric Data Processor, NDP)、数值处理器扩展(Numeric Processor Extension, NPX)和数学协处理器(math coprocessor)。

2. FPU 的控制寄存器

当 Intel 设计 80x87(本质上就是 IEEE 的浮点标准)的时候,还没有出现关于浮点硬件的标准。不同(大型和小型)计算机的生产商都有其各自不同且不兼容的浮点格式。遗憾的是,一些应用程序的编写利用了不同浮点格式特有的性质。Intel 想要设计出一种 FPU 能够用于多数软件(切记,当 Intel 开始设计 8087 时,距离 IBM PC 的发明还有 3~4 年,因此 Intel 不能依靠大量的 PC 可用软件来使它的芯片流行起来)。不过,这些旧的浮点格式中的特性是互不兼容的。例如,在某些浮点系统中,当没有足够的精度时采取舍入的办法,而在另一些浮点系统中则采用截断方式。有些应用软件只能在某个浮点系统中正常工作,但却不能在另一个系统中正常工作。Intel 想达到目的是让尽可能多的应用软件在改动尽可能小的情况下都能够在 80x87 FPU 上工作,因此它增加了一个特殊的寄存器, FPU 控制寄存器,它允许使用者可以从一些可用的操作模式中选择一种用于 FPU。

80x87 的控制寄存器含有 16 位,组织结构如图 6-2 所示。

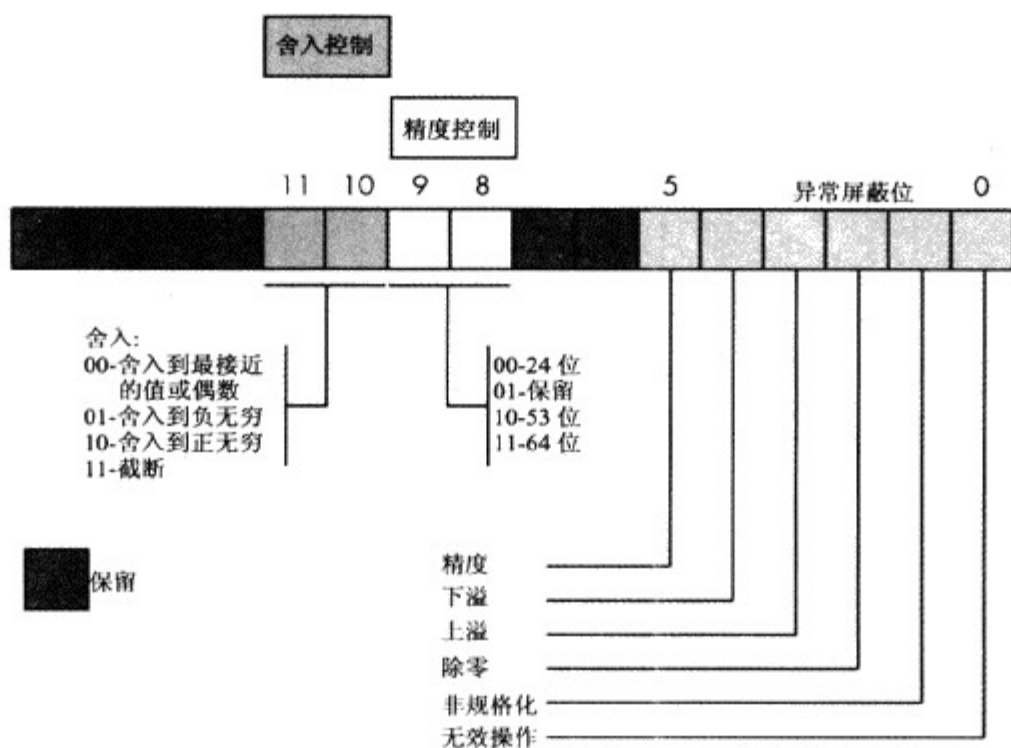


图 6-2 FPU 的控制寄存器

其中, FPU 控制寄存器的第 10 位和第 11 位依照表 6-8 中出现的数值提供舍入控制。

表 6-8 舍入控制

第 10 位和第 11 位	功 能
00	舍入到最接近的值或偶数
01	舍入到负无穷
10	舍入到正无穷
11	截断

默认设置为 00。FPU 以数值的最低有效位的 1/2 为界限,超过这个界限的数值向上舍入,而低于这个界限的数值向下舍入。如果在最低有效位之后的数值恰好是最低有效位数值的 1/2,则 FPU 把这个数值向最低有效位为 0 的那个最接近的数值舍入。对于一长串计算来说,这种做法提

供了一种合理而又自动的方式来维护最大的精度。

如果对计算结果的精度要求很严格，则需要确定是舍入到负无穷还是舍入到正无穷。设置舍入到负无穷后，执行操作，然后重新把舍入控制设置为舍入到正无穷，重复同样的操作，通过这种方式可以获得结果所在区间的上下边界。

截断选项强制所有的计算把多余的位数舍去。如果精度对您来说很重要，则很少使用这个选项。然而，如果要把旧版本的软件移植到 FPU 上，您可能会使用这个选项。当要把一个浮点数值转换为一个整数时，这个选项将非常有用。因为大多数软件选用截断方式将浮点数转换为整数，这时您需要使用截断舍入方式来实现它。

控制寄存器的第 8 位和第 9 位指定计算过程中的精度。像 IEEE754 标准所要求的，提供这种能力是为了允许与旧版软件相兼容。精度控制位使用表 6-9 所示的数值。

表 6-9 浮点尾数精度控制位

第 8 位和第 9 位	精度控制
00	24 位
01	保留
10	53 位
F1	64 位

有些 CPU 对精度为 53 位(如 64 位浮点格式)的浮点数值的操作要比精度是 64 位(如 80 位浮点格式)的快。请参见具体处理器的相关文档获得更详细的信息。通常情况下，CPU 默认使用%11 来选择 64 位的尾数精度。

第 0~5 位为异常屏蔽位(exception mask)。它们与 80x86 标志寄存器的中断使能位类似。如果这些位包含 1，则相应的条件被 FPU 忽略。然而，如果其中任何一位包含 0，并且相应的条件发生，则 FPU 立刻产生中断，以便程序能够处理退化条件(通常情况下，这将产生 HLA 异常，参见头文件 `excepts.hhf` 中的异常数值)。

第 0 位对应于无效操作错误。它通常作为一个编程错误的结果出现。导致无效操作异常(`ex.fInvalidOperation`)的问题包括：向栈中推入多于 8 项的内容或试图从空栈中弹出一项，对一个负数求平方根或者加载一个非空寄存器。

第 1 位屏蔽由于试图操作非规格化(denormalized)的数值而产生的非规格化中断。当把任意扩展精度的数值加载到 FPU 或对超出 FPU 能力范围的过小数值操作时会产生非规格化异常。通常情况，可以不启用这个异常。如果启用了这个异常，并且 FPU 产生了中断，那么 HLA 的运行时系统将产生 `ex.fDenormal` 异常。

第 2 位屏蔽除零异常。当该位为 0 时，如果试图用一个非 0 数值对 0 做除法，则 FPU 产生中断。如果没有启用除零异常，当执行一个 0 作除数的除法时，FPU 将产生 NaN(not a number)。通过把该位编程为 0 来启用这种异常是良好的编程习惯。注意，如果程序产生这种中断，则 HLA 的运行时系统产生 `ex.fDivByZero` 异常。

第 3 位屏蔽上溢异常。如果计算上溢或试图保存一个过大的数值到目的操作数中(如将一个大的扩展精度数值保存到一个单精度变量中)，FPU 将产生上溢异常。如果启用这个异常，同时 FPU 产生这种中断，HLA 的运行时系统将产生 `ex.fOverflow` 异常。

第4位屏蔽下溢异常。如果产生的结果对目的操作数来说太小，则会产生下溢。与上溢类似，当把一个小的扩展精度数值保存到一个更小的变量中(单精度或双精度)或产生的结果对扩展精度来说太小的时候，就会产生这种异常。如果启用这个异常，同时 FPU 产生这种中断，HLA 的运行系统产生 `ex.fUnderflow` 异常。

第5位控制精度异常是否发生。当 FPU 产生一个不精确的结果，通常是一个内部舍入操作产生的结果时，会发生精度异常。虽然很多操作都产生精确的结果，但有更多的结果并不精确。例如，用 1 与 10 做除法将产生一个不精确的结果。因此，该屏蔽位通常为 1，因为不精确的结果很常见。如果启用这种异常，同时 FPU 产生这种中断，HLA 运行系统产生 `ex.InexactResult` 异常。

控制寄存器的第6、7位和第12~15位当前没有定义，保留下来供将来使用(第7位和第12位在较老的 FPU 上是有效的，但现在已不再使用)。

FPU 提供两条指令 `fldcw`(加载控制字)和 `fstcw`(保存控制字)，通过这两条指令可以加载和保存控制寄存器的内容。这两条指令的单一操作数必须是 16 位的存储单元。指令 `fldcw` 从指定的存储单元加载控制寄存器，指令 `fstcw` 把控制寄存器的内容保存到指定的存储单元中。这两条指令的语法为：

```
fldcw( mem16 );
fstcw( mem16 );
```

下面的示例代码设置舍入控制为“截断结果”，同时设置舍入精度为24位：

```
static
    fcw16: word;
    .
    .
    .
    fstcw( fcw16 );
    mov( fcw16, ax );
    and( $f0ff, ax );    // Clears bits 8-11.
    or( $0c00, ax );     // Rounding control=%11, Precision = %00.
    mov( ax, fcw16 );
    fldcw( fcw16 );
```

3. FPU 的状态寄存器

当读取 FPU 的状态寄存器时，它为您提供当前的 FPU 状态。指令 `fstsw` 把 16 位的浮点状态寄存器保存到一个字变量中。状态寄存器是一个 16 位的寄存器，其布局如图 6-3 所示。

第0~5位是异常标志。这几位的出现顺序与控制寄存器中异常屏蔽位的顺序相同。如果相应的条件存在，则该位被置位。这些位与控制寄存器中的异常屏蔽位是相互独立的。FPU 对这些位的置位与清除并不考虑相应屏蔽位的设置情况。

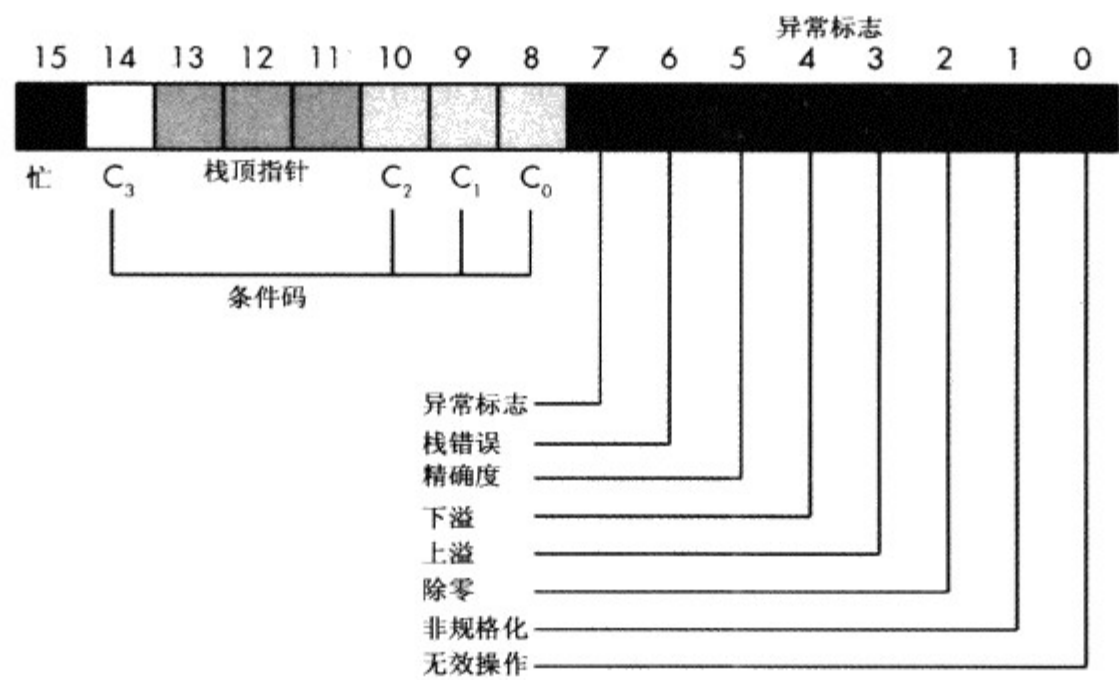


图 6-3 FPU 的状态寄存器

第 6 位指示一个栈错误(stack fault)。当栈上溢或下溢时会产生一个栈错误。当该位被置位后，用条件码位 C₁ 来确定是一个栈上溢(C₁=1)或是栈下溢(C₁=0)。

如果有任何一个错误条件位被置位，则置位状态寄存器的第 7 位。它实际上是第 0~5 位的逻辑 or 操作。程序可以通过测试该位很快确定是否存在一个错误条件。

第 8、9、10 和 14 位是协处理器条件码位。许多不同的指令置位条件码位，如表 6-10 所示。

表 6-10 FPU 的条件码位(X=不关心)

指 令	条 件 码 位				条 件
	C ₃	C ₂	C ₁	C ₀	
fcom	0	0	X	0	ST>源
fcomp	0	0	X	1	ST<源
fcompp	1	0	X	0	ST=源
ficom	1	1	X	1	ST 或源未定义
ficom					
fcomp					
fst	0	0	X	0	ST 为正
	0	0	X	1	ST 为负
	1	0	X	0	ST 为 0(+或-)
	1	1	X	1	ST 是不可比较的
fxam	0	0	0	1	+非规格化
	0	0	1	0	- 非规格化
	0	1	0	0	+规格化
	0	1	1	0	- 规格化
	1	0	0	0	+0
	1	0	1	0	- 0

(续表)

指 令	条 件 码 位				条 件
fxam-	1	1	0	0	+反向规格化
	1	1	1	0	- 反向规格化
	0	0	0	1	+NaN
	0	0	1	1	- NaN
	0	1	0	1	+无穷
	0	1	1	1	- 无穷
	1	X	X	1	空寄存器
fucom	0	0	X	0	ST>源
fucomp	0	0	X	1	ST<源
fucompp	1	0	X	0	ST=源
	1	1	X	1	无序

FPU 状态寄存器的第 11~13 位提供栈顶的寄存器号。在计算的过程中，FPU 把编程人员提供的逻辑寄存器号加到这 3 位(模 8)，来确定运行时的物理寄存器号。

状态寄存器第 15 位是忙位。当 FPU 忙时设置该位。这个位是当 FPU 作为一个单独的芯片时引入的，大多数程序很少去访问这一位。

6.5.2 FPU 的数据类型

FPU 支持 7 种不同的数据类型：3 种整数类型、1 种压缩的十进制类型和 3 种浮点类型。整数类型支持 64 位的整数，尽管使用 CPU 的整数单元来做 64 位算术运算通常会快一些(参见第 8 章)。无疑，使用标准的整数寄存器做 16 位和 32 位的整数算术运算更快。压缩的十进制类型提供 17 位有符号的十进制(BCD)整数。使用 BCD 格式的主要目的是在字符串和浮点数值间进行转换。剩下的 3 种数据类型是 32 位、64 位和 80 位的浮点数据类型。80x87 的数据类型如图 6-4、图 6-5 和图 6-6 所示。

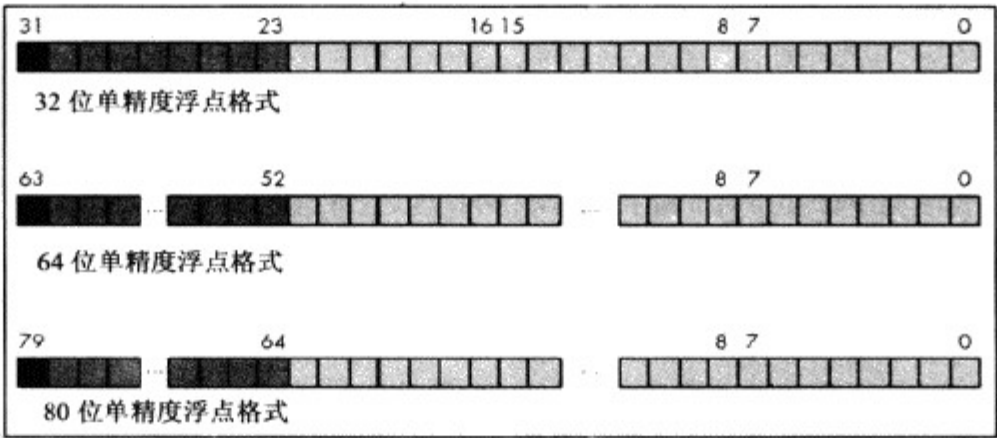


图 6-4 FPU 浮点格式

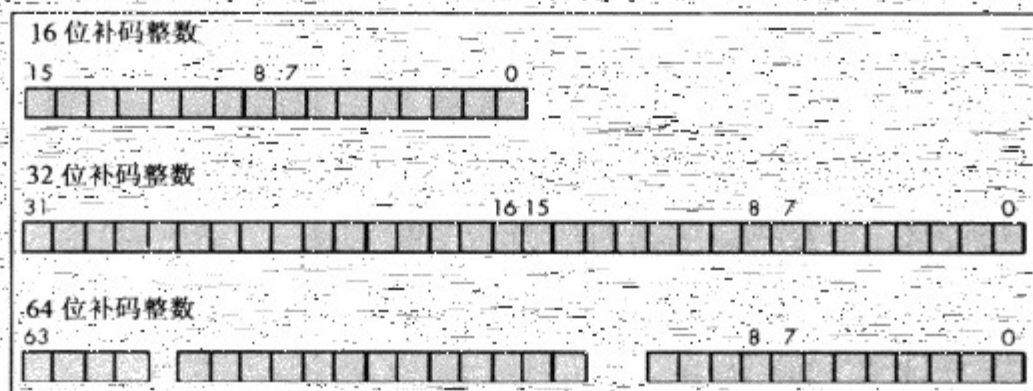


图 6-5 FPU 整数格式



图 6-6 FPU 压缩的十进制格式

FPU 通常把数值保存在一个规格化格式中。当一个浮点数经过规格化时,尾数的高位位总是 1。对于 32 位和 64 位的浮点格式, FPU 实际上并不保存这一位,而总是假定它为 1。因此, 32 位和 64 位的浮点数总是规格化的。对于扩展精度的 80 位浮点格式, FPU 并不假定尾数的高位位为 1, 尾数的高位位作为位串的一部分出现。

规格化数值对于给定数目的位提供最好的精度。然而, 还有大量我们不能用 80 位格式表示的非规格化数值。这些数值非常接近 0, 代表尾数高位位为非 0 的一组数值。FPU 提供一种特殊的 80 位非规格化数值形式。有许多很小的数值用规格化方式是不能对它们编码的, 使用非规格化数值可以做到这一点, 但与规格化数值相比, 精度位却有所减少。因此, 在计算中使用非规格化数值可能导致精度降低。当然, 这总比让非规格化数值下溢为 0 要好(这样的计算更不精确), 但必须记住, 在计算中对很小的数值进行操作时, 很可能会丢失一定的精度。注意, FPU 的状态寄存器中专门提供了一位, 可以用来探测 FPU 在计算中使用的是不是一个非规格化数值。

6.5.3 FPU 的指令集

FPU 在 80x86 的指令集中增加了许多新指令。我们可以把它们分为数据转移指令、换算指令、算术运算指令、比较指令、常量指令、超越指令和其他指令。接下来就按照这种分类方法分别对每种指令进行讲述。

6.5.4 FPU 的数据转移指令

数据转移指令负责在 FPU 的内部寄存器和存储器之间进行数据传递。这类指令包括 fld、fst、fstp 和 fxch。指令 fld 总是把它的操作数推入浮点栈中。指令 fstp 先对栈顶的内容进行保存, 然后再把它弹出栈。其他两条指令并不影响栈中的数据项。

1. fld 指令

指令 fld 把一个 32 位、64 位或 80 位的浮点数值装入栈顶。对于 32 位和 64 位的操作数, 这条指令首先把它们转换成一个 80 位扩展精度的数值, 然后再把转换后的数值推入栈。

指令 fld 首先把栈顶(TOS)指针(状态寄存器的第 1F~13 位)减 1, 然后把 80 位的数值保存在新

的栈顶指针所指向的物理寄存器中。如果 `fld` 指令的源操作数是一个浮点数据寄存器 `sti`, 那么 FPU 实际用于装入操作的寄存器是栈顶指针减 1 之前的那个寄存器号。因此, 语句 `fld(st0)` 把数值复制到栈顶。

如果栈溢出, 则指令 `fld` 置位栈错误位。如果装入的是一个 80 位非规格化数值, 该指令置位非规格化异常位。如果试图使用这条指令在栈顶装入一个空的浮点寄存器(或执行其他无效操作), 它将置位无效操作位。例如:

```
fld( st1 );
fld( real32_variable );
fld( real64_variable );
fld( real80_variable );
fld( (type real64 [ebx]) );
fld( real_constant );
```

注意, 无法在浮点栈中直接装入一个 32 位的整数寄存器, 即使这个寄存器中保存的是一个 `real32` 数值。为了实现这个操作, 必须首先把整数寄存器的内容保存到一个存储单元中, 然后, 就可以使用 `fld` 指令把这个存储单元推入 FPU 栈。例如:

```
mov( eax, tempReal32 ); // Save real32 value in eax to memory.
fld( tempreal32 );      // Push that real value onto the FPU stack.
```

注意, 通过指令 `fld` 装入一个常量是一种 HLA 扩展。FPU 不支持这种指令类型。HLA 在常量段创建一个 `real80` 对象, 然后使用这个存储器对象的地址作为指令 `fld` 的操作数。

2. `fst` 和 `fstp` 指令

指令 `fst` 和 `fstp` 把浮点栈顶中的数值复制到另一个浮点寄存器中, 或复制到一个 32 位、64 位或 80 位的内存变量中。当要把数值复制到一个 32 位或 64 位的内存变量中时, 栈顶的 80 位扩展精度数值将按照 FPU 控制寄存器的舍入控制位所指定的舍入为一个较小的格式。

指令 `fstp` 从栈顶弹出数值并把它转移到目的地址。当该指令访问了 `ST0` 中的数据后, 将对状态寄存器中的栈顶指针执行加 1 操作。如果目的操作数是一个浮点寄存器, FPU 在数值被弹出栈之前就已经把数值保存在了指定编号的寄存器中。

执行指令 `fstp(st0)` 可弹出栈顶数据而没有任何数据移动。例如:

```
fst( real32_variable );
fst( real64_variable );
fst( realArray[ ebx*8 ] );
fst( st2 );
fstp( st1 );
```

上面最后一条指令可弹出 `ST1`, 而把 `ST0` 保留在栈顶。

如果发生栈下溢, 指令 `fst` 和 `fstp` 将对栈异常位(试图从空的寄存器栈中读取数据)进行置位。如果在保存操作中有精度损失(例如, 当把一个 80 位扩展精度的数值保存到一个 32 位或 64 位的内存变量中时, 在数据换算的过程中将丢失一些位), 这两条指令将对精度位进行置位。当把一个 80 位数值保存到一个 32 位或 64 位内存变量中时, 如果换算后的数据太小而不适合目的操作数,

则这两条指令将置位下溢异常位。同样,如果栈顶的数值太大而不适合 32 位或 64 位的内存变量,它们将对上溢异常位进行置位。当试图把一个非规格化数值保存到一个 80 位寄存器或变量中时,指令 `fst` 和 `fstp` 对非规格化标志进行置位²。如果发生无效操作(对空寄存器进行保存操作),它们置位无效操作标志。最后,如果在保存操作时进行了舍入,这两条指令将置位 `C1` 条件位(只有向一个 32 位或 64 位内存变量保存数据时才会有舍入发生,这时,必须舍去尾数以适应目的操作数)。

注意:

由于 FPU 指令集中相关的指令编码特性,因此不能使用指令 `fst` 把数据保存到 `real80` 的内存变量中。然而,可以使用指令 `fstp` 来保存 80 位数据。

3. `fxch` 指令

指令 `fxch` 把栈顶数值与其他某个 FPU 寄存器进行交换。这条指令有两种形式:一种带有一个 FPU 寄存器作为操作数;另一种没有操作数。第一种形式把栈顶数值与指定的寄存器进行交换。指令 `fxch` 的第二种形式把栈顶数值与 `ST1` 进行交换。

许多 FPU 指令,如 `fsqrt`,只能对栈顶的寄存器进行操作。如果想对非栈顶数值执行这种操作,可以先使用指令 `fxch` 把这个寄存器与栈顶数值交换,然后执行相应操作,最后再使用 `fxch` 把栈顶数据与原始数据交换。下面是求 `ST2` 的平方根的示例:

```
fxch( st2 );
fsqrt();
fxch( st2 );
```

如果栈为空,指令 `fxch` 会置位栈异常位。如果指定一个空寄存器作为操作数,它会设置无效操作位。该指令总是会清除 `C1` 条件码位。

6.5.5 换算指令

FPU 在 80 位实数上执行所有的算术操作。在某种意义上,指令 `fild` 和 `fst/fstp` 也可以视为换算指令,因为它们自动在 80 位实数格式和 32 位或 64 位存储器格式之间进行内部换算。尽管如此,我们还是把它们归类为数据转移指令,而不是换算指令,因为它们主要的目的还是向存储器或从存储器中转移数据。当进行数据转移时,FPU 提供了 6 种其他用于对整数或 BCD 码进行换算的指令。它们是: `fild`、`fist`、`fistp`、`fisttp`、`fbld` 和 `fbstp`。

1. `fild` 指令

指令 `fild` 把一个 16 位、32 位或 64 位补码整数换算成 80 位扩展精度格式,同时把换算后的结果推入栈顶。这条指令只需要一个操作数。这个操作数必须是一个字、双字或四字整型变量的地址。不能指定一个 80x86 的 16 位或 32 位通用寄存器作为这个操作数。如果想要把一个 80x86 通用寄存器的值推入 FPU 栈,必须首先把它保存在一个内存变量中,然后再使用指令 `fild` 把这个内存变量的值推入栈。

当向栈中推入换算后的结果时,如果发生溢出,指令 `fild` 将置位栈异常位和 `C1` 条件位。例如:

² 将一个非规格化值存储到 32 位或者 64 位存储器变量总是会置位下溢异常位。

```

fild( word_variable );
fild( dword_val[ ecx*4 ] );
fild( qword_variable );
fild( (type int64 [ebx]) );

```

2. fist、fistp 和 fisttp 指令

指令 `fist`、`fistp` 和 `fisttp` 把栈顶的 80 位扩展精度变量换算成一个 16 位、32 位或 64 位的整数，并把换算的结果保存到操作数所指定的内存变量中。`fistp` 和 `fisttp` 指令把栈顶数值换算成整数时，依照 FPU 控制寄存器中的舍入设置(第 10、11 位)。`fisttp` 指令总是使用截断模式进行换算。与指令 `fild` 类似，指令 `fist`、`fistp` 和 `fisttp` 不允许指定一个 80x86 的 16 位或 32 位的通用寄存器作为目的的操作数。

指令 `fist` 把栈顶的数值换算成一个整数并保存结果，它并不会对浮点寄存器栈有其他的影响。而指令 `fistp` 和 `fisttp` 把换算的结果保存后，还会从浮点寄存器栈顶把换算的数据弹出。

如果浮点寄存器栈为空，这几条指令将置位栈异常位(这也会清除 C_1)。如果有舍入发生(就是说，ST0 中的数值有小数部分)，它们将置位精度(操作不精确)和 C_1 位。如果换算后的结果太小，它们将置位下溢异常位(也就是，大于 0 但小于 1，或者大于 -1 但小于 0)。例如：

```

fist( word_var[ ebx*2 ] );
fist( qword_var );
fisttp( dword_var );
fistp( dword_var );

```

不要忘记，指令 `fist` 和 `fistp` 是根据舍入控制设置来决定它们将如何把浮点数据换算成一个整数，然后再进行存储操作。默认情况下，舍入控制通常设置为“舍入”模式，而大多编程人员都希望指令 `fist/fistp` 在数据换算时截断小数部分。如果想要指令 `fist/fistp` 在把浮点数换算成整数时，对浮点数执行截断操作，则需要置位浮点控制寄存器中相应的舍入控制位(或者使用指令 `fisttp`，不管舍入控制位的设置如何都截断结果)。例如：

```

static
    fcw16:      word;
    fcw16_2:    word;
    IntResult:  int32;
    .
    .
    .
    fstcw( fcw16 );
    mov( fcw16, ax );
    or( $0c00, ax );      // Rounding control=%11 (truncate).
    mov( ax, fcw16_2 );   // Store into memory and reload the ctrl word.
    fldcw( fcw16_2 );

    fistp( IntResult );   // Truncate ST0 and store as int32 object.

    fldcw( fcw16 );      // Restore original rounding control.

```

3. fbld 和 fbstp 指令

指令 fbld 和 fbstp 对 80 位 BCD 数值进行载入和保存。指令 fbld 把一个 BCD 数值换算成与其等价的 80 位扩展精度形式，并把换算的结果放在栈顶。指令 fbstp 把栈顶的扩展精度数值换算成一个 80 位 BCD 数值(按照浮点控制寄存器中的舍入控制位进行舍入)，并把换算后的结果保存在目的存储器操作数指定的地址中。注意，没有 fbst 指令。

如果发生栈溢出，则指令 fbld 置位栈异常位和 C_1 条件位。如果试图载入一个无效的 BCD 数据，指令 fbld 置位无效操作位。如果发生栈下溢(栈为空)，指令 fbstp 置位栈异常位同时清除 C_1 。该指令置位下溢标志的条件与指令 fist 和 fistp 相同。例如：

```
// Assuming fewer than 8 items on the stack, the following
// code sequence is equivalent to an fbst instruction:

    fld( st0 );
    fbstp( tbyte_var );

// The following example easily converts an 80-bit BCD value to
// a 64-bit integer:

    fbld( tbyte_var );
    fist( qword_var );
```

这两条指令对字符串与浮点格式之间的换算特别有用。参见 HLA 标准库中的浮点格式到字符串和字符串到浮点浮点格式换算例程，可获得更多的详细信息。

6.5.6 算术运算指令

算术运算指令是 FPU 指令集的子集，这个子集很小，但却很重要。这些指令大体分为两类——对实数进行操作的指令和对实数、整数都能够操作的指令。

1. fadd 和 faddp 指令

这两条指令采用下面的形式：

```
fadd()
faddp()
fadd( st0, sti );
fadd( sti, st0 );
faddp( st0, sti );
fadd( mem_32_64 );
fadd( real_constant );
```

当没有操作数时，指令 fadd 对 ST0 中的值与 ST1 中的值求和，然后把结果保存到 ST1 中。指令 faddp(没有操作数)把栈顶的两个数据弹出，对它们求和，然后再把结果放回栈顶。

fadd 指令接下来的两种形式带有两个 FPU 寄存器操作数，它们的行为与 80x86 的 add 指令相似。它们把源操作数寄存器中的数值加到目的操作数寄存器中。注意，ST0 必须是寄存器操作数之一。

带有两个操作数的指令 faddp，把 ST0(必须是源操作数)与目的操作数相加，同时把 ST0 从栈

顶弹出。其中，目的操作数必须是另一个 FPU 寄存器。

最后一种形式，带有一个存储器操作数的 `fadd` 指令，把一个 32 位或 64 位的浮点变量与 `ST0` 中的数值相加。在执行加法操作之前，这条指令首先把 32 位或 64 位操作数换算成一个 80 位扩展精度的数值。注意，这条指令不允许使用 80 位的存储器操作数。

在适当情况下，这些指令都可以引发栈、精度、下溢、上溢、非规格化和非法操作异常。如果有栈错误异常产生，`C1` 指示栈上溢或下溢。

与指令 `fld(real_constant)` 类似，指令 `fadd(real_constant)` 是一条 HLA 的扩展指令。注意，对于该指令，HLA 会创建一个保存常量数值的 64 位变量，并发出指令 `fadd(mem64)`，同时，指定它在常量声明段创建的只读对象。

2. `fsub`、`fsubp`、`fsubr` 和 `fsubrp` 指令

这 4 条指令采用下面的形式：

```

fsub()
fsubp()
fsubr()
fsubrp()

fsub( st0, sti )
fsub( sti, st0 );
fsubp( st0, sti );
fsub( mem_32_64 );
fsub( real_constant );

fsubr( st0, sti )
fsubr( sti, st0 );
fsubrp( st0, sti );
fsubr( mem_32_64 );
fsubr( real_constant );

```

对于没有操作数的情况，指令 `fsub` 从 `ST1` 减去 `ST0`，然后把结果留在 `ST1` 中。而指令 `fsubp` 从栈寄存器中弹出 `ST0` 和 `ST1`，计算 `ST1 - ST0`，然后把计算结果放回栈顶。指令 `fsubr` 和 `fsubrp` (反向减法) 几乎按照与前面两条指令相同的方式操作，差别在于它们计算的是 `ST0 - ST1`。

带有两个寄存器操作数(源操作数和目的操作数)的指令 `fsub`，计算的是“目的操作数:=目的操作数 - 源操作数”。`ST0` 必须是这两个寄存器之一。带有两个寄存器操作数的指令 `fsubp` 同样是计算“目的操作数:=目的操作数 - 源操作数”，但计算完成后，它从栈顶把 `ST0` 弹出。因此，对于指令 `fsubp`，源操作数必须是 `ST0`。

带有两个寄存器操作数的指令 `fsubr` 和 `fsubrp` 的工作方式与 `fsub` 和 `fsubp` 类似，差别在于，它们计算的是“目的操作数:=源操作数 - 目的操作数”。

指令 `fsub(mem)` 和指令 `fsubr(mem)` 的操作数是一个 32 位或 64 位的存储器操作数。它们都把存储器操作数换算成一个 80 位扩展精度的数值，对于指令 `fsub` 是用 `ST0` 减去这个数值，而对于指令 `fsubr` 则是用这个数值减去 `ST0`，然后把结果存回 `ST0` 中。

这些指令在相应条件发生时可以引发栈、精度、下溢、上溢、非规格化和非法操作异常。如果发生栈错误异常，`C1` 指示栈上溢或下溢。

注意:

带有实型常量作为操作数的指令并不是真正的 FPU 指令。它们是 HLA 的扩展。HLA 将生成一个常量段存储器对象,并用这个常量数值初始化这个对象。

3. fmul 和 fmulp 指令

指令 **fmul** 和 **fmulp** 对两个浮点数值执行乘法操作。它们采用下面的形式:

```
fmul()
fmulp()

fmul( sti, st0 );
fmul( st0, sti );
fmul( mem_32_64 );
fmul( real_constant );

fmulp( st0, sti );
```

在没有操作数的时候,指令 **fmul** 计算 $st0 \times st1$,并把乘积保存到 ST1 中。而指令 **fmulp** 从栈顶弹出 ST0 和 ST1,对这两个数值做乘法,把结果放回栈顶。带有两个寄存器操作数的指令 **fmul** 计算的是“目的操作数:=目的操作数*源操作数”。ST0 必须是其中一个寄存器(源操作数或目的操作数)。

指令 **fmup(st0,sti)** 计算 $sti := sti \times st0$,然后把 ST0 从栈顶弹出。在把 ST0 从栈顶弹出之前,指令首先使用 STi 的值。指令 **fmul(mem)** 需要一个 32 位或 64 位的存储器操作数。它首先把指定的内存变量换算为一个 80 位扩展精度的数值,然后再把它与 ST0 相乘。

相应条件发生时,这些指令可以引发栈、精度、下溢、上溢、非规格化和非法操作异常。如果计算的时候进行了舍入操作,这些指令将置位 C_1 条件码位。如果有栈错误异常发生, C_1 指示栈下溢或上溢。

注意:

带有一个实型常量作为操作数的指令并不是真正的 FPU 指令。它是 HLA 提供的一个扩展(参见前一小节末尾的“注意”)。

4. fdiv、fdivp、fdivr 和 fdivrp 指令

这 4 条指令采用下面的形式:

```
fdiv()
fdivp()
fdivr()
fdivrp()

fdiv( sti, st0 );
fdiv( st0, sti );
fdivp( st0, sti );

fdivr( sti, st0 );
fdivr( st0, sti );
```

```

fdivrp( st0, sti );

fdiv( mem_32_64 );
fdivr( mem_32_64 );
fdiv( real_constant );
fdivr( real_constant );

```

在没有操作数的时候, 指令 `fdivp` 从栈顶弹出 `ST0` 和 `ST1`, 计算 `st1/st0`, 然后把结果放回栈顶。而没有操作数的指令 `fdiv` 则计算 `st1:=st1/st0`。指令 `fdivr` 和 `fdivrp` 的工作方式与 `fdiv` 和 `fdivp` 相似, 但它计算的是 `st0/st1`, 而不是 `st1/st0`。

带有两个寄存器操作数的时候, 这些指令计算下面的商:

```

fdiv( sti, st0 );      // st0 := st0/sti
fdiv( st0, sti );      // sti := sti/st0
fdivp( st0, sti );     // sti := sti/st0 then pop st0
fdivr( st0, sti );     // st0 := st0/sti
fdivrp( st0, sti );    // sti := st0/sti then pop st0

```

指令 `fdivp` 和 `fdivrp` 在执行完除法操作之后, 还将把 `ST0` 从栈顶弹出。在弹出 `ST0` 之前, 将首先计算这两条指令中 `i` 的值。

相应条件发生时, 这些指令可以引发栈、精度、下溢、上溢、非规格化、除零和非法操作异常。如果计算的时候进行了舍入操作, 这些指令将置位 `C1` 条件码位。如果有栈错误异常发生, `C1` 指示栈下溢或上溢。

注意:

带有一个实型常量作为操作数的指令并不是真正的 FPU 指令。它们是 HLA 提供的扩展指令。

5. fsqrt 指令

指令 `fsqrt` 不允许出现任何操作数。它计算栈顶数值的平方根, 并用计算的结果取代 `ST0` 中的值。栈顶的数值必须是 0 或正数, 否则, 指令 `fsqrt` 将产生一个非法操作异常。

相应条件发生时, 这条指令可以引发栈、精度、非规格化和非法操作异常。如果计算的时候进行了舍入操作, 指令 `fsqrt` 置位 `C1` 条件码位。如果有栈错误异常发生, `C1` 指示栈下溢或上溢。

例如:

```

// Compute z := sqrt(x**2 + y**2);

fld( x );          // Load x.
fld( st0 );         // Duplicate x on TOS.
fmul();            // Compute x**2.

fld( y );          // Load y
fld( st0 );         // Duplicate y.
fmul();            // Compute y**2.

fadd();            // Compute x**2 + y**2.
fsqrt();           // Compute sqrt( x**2 + y**2 ).
fstp( z );         // Store result away into z.

```

6. fprem 和 fprem1 指令

指令 `fprem` 和 `fprem1` 用于计算一个部分余数(partial remainder)。在 IEEE 出台浮点标准之前, Intel 就已经设计出了 `fprem` 指令。在最终的 IEEE 浮点标准草案中, 关于指令 `fprem` 的定义与 Intel 的最初设计有些不同。由于 Intel 必须兼容使用指令 `fprem` 的已有软件, 因此, 它设计了一条新的用于处理 IEEE 部分求余的指令——`fprem1`。在新的软件中, 应该使用指令 `fprem1`, 因此, 这里只讨论 `fprem1`, 当然, 对于指令 `fprem` 可以以相同的方式使用。

指令 `fprem1` 计算 `st0/st1` 的部分余数。如果 `ST0` 和 `ST1` 的指数之差小于 64, 则指令 `fprem1` 可以通过一个操作计算出正确的余数。否则, 必须执行 `fprem1` 两次或多次来得到正确的余数数值。`C2` 条件码指示计算何时完成。注意, `fprem1` 并没有从栈顶把两个操作数弹出。它把部分余数保存在 `ST0` 中, 原始除数在 `ST1` 中。如果想要计算另一个部分余数来完成操作, 仍然可以使用它们。

如果栈顶没有两个数值, 指令 `fprem1` 置位栈异常标志。如果计算的结果太小, 它置位下溢和非规格化异常位。如果栈顶数值与求余操作不匹配, 它置位无效操作位。如果一个部分求余操作没有完成, 它置位 `C2` 条件码位。最后, 它将商的第 0、1、2 位分别装入 `C3`、`C1` 和 `C0`。

例如:

```
// Compute z := x mod y

    fld( y );
    fld( x );
    repeat

        fprem1();
        fstsw( ax );           // Get condition code bits into ax.
        and( 1, ah );         // See if C2 is set.

    until( @z );              // Repeat until C2 is clear.
    fstp( z );                 // Store away the remainder.
    fstp( st0 );               // Pop old y value.
```

7. frndint 指令

指令 `frndint` 依据控制寄存器所指定的舍入算法把栈顶数值舍入为最接近的整数值。

如果栈顶没有数值, 指令 `frndint` 置位栈异常标志(同时清除 `C1`)。如果有精度损失, `frndint` 置位精度和非规格化异常位。如果栈顶保存的不是一个有效数值, `frndint` 置位无效操作标志。注意, 栈顶的数值仍是一个浮点数, 只是它没有小数部分。

8. fabs 指令

指令 `fabs` 通过清除 `ST0` 的尾数符号位来计算 `ST0` 的绝对值。如果栈为空, 则 `fabs` 置位栈异常位和无效操作位。

例如:

```
// Compute x := sqrt(abs(x));

    fld( x );
    fabs();
```

```
fsqrt();
fstp( x );
```

9. fchs 指令

指令 **fchs** 通过反转 ST0 尾数符号位来改变 ST0 数值的符号(就是说, 指令 **fchs** 是浮点数取反指令)。如果栈为空, 则 **fchs** 置位栈异常位和无效操作位。例如:

```
// Compute x := -x if x is positive, x := x if x is negative.
// That is, force x to be a negative value.
    fld( x );
    fabs();
    fchs();
    fstp( x );
```

6.5.7 比较指令

FPU 提供了许多用于实型数值比较的指令。指令 **fcom**、**fcomp** 和 **fcompp** 对栈顶的两个数值进行比较, 并置位相应的条件码。指令 **fst** 把栈顶数值与 0 进行比较。

通常情况下, 多数程序会在一次比较之后立即查看条件码位。遗憾的是, 没有用于测试 FPU 条件码的 FPU 指令。只有先使用指令 **fstsw** 把浮点状态寄存器复制到寄存器 AX 中, 然后再使用指令 **sahf** 把寄存器 AH 的内容复制到 80x86 的条件码位中。这些操作完成后, 可以通过测试标准的 80x86 标志位来查看条件码。这种技术分别把 C_0 复制到进位标志, C_2 复制到奇偶校验标志, C_3 复制到零标志中。指令 **sahf** 没有把 C_1 复制到任何 80x86 标志位中。

由于指令 **sahf** 没有把任何 FPU 状态位复制到符号或溢出标志中, 因此不能使用有符号的比较指令。相反, 要使用无符号操作(如 **seta**、**setb**)来测试一个浮点比较的结果。事实上, 这些指令通常测试的是无符号数值, 而浮点数是有符号数值。但仍然要使用无符号操作, 指令 **fstsw** 和 **sahf** 会如同您使用指令 **cmp** 比较无符号数值一样设置 80x86 标志寄存器。

Pentium II 及其向上兼容的处理器另外提供了浮点比较指令集, 其中的指令可以直接影响 80x86 的条件码标志。有了这些指令, 就不必使用指令 **fstsw** 和 **sahf** 把 FPU 状态复制到 80x86 的条件码中。这些指令包括 **fcomi** 和 **fcomip**。可以像使用指令 **fcom** 和 **fcomp** 一样使用这两条指令, 当然, 差别在于不必手工把状态位复制到 FLAGS 寄存器中。

1. fcom、fcomp 和 fcompp 指令

指令 **fcom**、**fcomp** 和 **fcompp** 把 ST0 与指定的操作数进行比较, 同时根据比较的结果设置相应的 FPU 条件码位。这些指令的合法形式为:

```
fcom()
fcomp()
fcompp()

fcom( sti )
fcomp( sti )

fcom( mem_32_64 )
fcomp( mem_32_64 )
```

```
fcom( real_constant )
fcomp( real_constant )
```

如果不指定操作数,指令 `fcom`、`fcomp` 和 `fcompp` 把 `ST0` 与 `ST1` 进行比较,并置位相应的 FPU 标志。另外,指令 `fcomp` 会把 `ST0` 弹出栈,指令 `fcompp` 把 `ST0` 和 `ST1` 都弹出栈。

若带有一个寄存器操作数,指令 `fcom` 和 `fcomp` 把 `ST0` 与指定的寄存器进行比较。比较完成后,`fcomp` 还将把 `ST0` 从栈顶弹出。

如果带有一个 32 位或 64 位存储器操作数,指令 `fcom` 和 `fcomp` 首先把这个内存变量换算成一个 80 位的扩展精度数值,然后把这个数值与 `ST0` 进行比较,并置位相应的条件码位。`fcomp` 还将把 `ST0` 从栈顶弹出。

如果给定操作数是不可比较的(如 NaN),这两条指令置位 `C2` 条件码位(最终在奇偶标志中)。如果在一个比较操作中可能出现非法浮点数值,应该在查看期望的条件之前,首先查看奇偶标志是否有错(例如,使用 HLA 的 `@p` 和 `@np` 条件,或使用指令 `setp/setnp`)。

如果在寄存器栈顶没有两项,这些指令将置位栈错误位。如果操作数中有一个或者两个都是非规格化数值,这些指令将置位非规格化异常位。如果操作数中有一个不是或两个都不是数字,指令将置位无效操作位。这些指令总是清除 `C1` 条件码。

注意,带有实型常量操作数的指令不是真正的 FPU 指令。它们是 HLA 的扩展。当 HLA 遇到这样的指令时,它在常量段创建一个 `real64` 只读变量,并用指定的常量来初始化这个变量。这样 HLA 就把最初的指令转换成一条指定了一个 `real64` 存储器操作数的指令。

注意

由于精度的差别(64 位与 80 位相比),在一条浮点指令中使用常量操作数所得到的结果将不如预期的精确。

看一下浮点比较指令的示例:

```
fcompp();
fstsw( ax );
sahf();
setb( al );    // al = true if st1 < st0.
```

注意,不能在一个 HLA 运行时的布尔表达式中对浮点数值进行比较(例如在一条 `if` 语句中)。然而,可以在一个浮点比较之后(就像上面的语句序列那样)在这种语句中测试条件,例如:

```
fcompp();
fstsw( ax );
sahf();
if( @b )then

    << Code that executes if st1 < st0 >>

endif;
```


2. fcomi 和 fcomip 指令

指令 `fcomi` 和 `fcomp` 将 `ST0` 与指定的操作数进行比较, 然后根据比较的结果置位相应的 `EFLAGS` 条件码位。这些指令的使用方法类似于 `fcom` 和 `fcomp`, 但是在执行这些指令后, 可以直接测试 CPU 的标志位, 而不需要先把 FPU 的状态位移到 `EFLAGS` 寄存器。指令 `fcomi` 和 `fcomip` 的合法形式如下:

```
fcomi()  
fcomip()  
  
fcomi(sti)  
fcomip(sti)  
  
fcomi(mem_32_64)  
fcomip(mem_32_64)  
  
fcomi(real_constant)  
fcomip(real_constant)
```

3. fst 指令

指令 `fst` 把 `ST0` 中的数值与 0.0 进行比较。它的行为与 `ST1` 中包含 0.0 时的 `fcom` 指令相似。注意, 这条指令不对 -0.0 和 +0.0 进行区分。`ST0` 中的数值是这两者之一, 指令 `fst` 将置位 `C3` 来指示相等。注意, 该指令不把 `ST0` 从栈顶弹出。例如:

```
fstst();  
fstsw(ax);  
sahf();  
sete(al);           // Set al to 1 if TOS = 0.0
```

6.5.8 常量指令

FPU 提供了一些指令可以把通常用到的常量装入 FPU 寄存器栈中。如果栈溢出, 这些指令置位栈错误、无效操作和 `C1` 标志, 它们不会再影响其他的 FPU 标志。这些指令包括:

```
fldz();           // Pushes +0.0  
fldl();           // Pushes +1.0  
fldpi();          // Pushes pi  
fldl2t();          // Pushes log2(10)  
fldr2e();          // Pushes log2(e)  
fldlg2();          // Pushes log10(2)  
fldln2();          // Pushes ln(2)
```

6.5.9 超越指令

FPU 提供了 8 条超越指令(对数和三角函数计算指令), 分别计算正弦、余弦、部分正切(partial tangent)、部分反正切(partial arctangent)、 $2^x - 1$ 、 $y * \log_2(x)$ 和 $y * \log_2(x+1)$ 。通过各种代数恒等变换, 使用这些指令很容易计算出许多常见的超越函数。

1. f2xm1 指令

指令 `f2xm1` 计算 $2^{ST0} - 1$ 。其中 `ST0` 的数值必须在满足 $-1.0 \leq ST0 \leq +1.0$ 。如果 `ST0` 的数值不在这个范围内, 指令 `f2xm1` 将产生一个不确定的结果, 但并不引发任何异常。同时用计算所得到的结果取代 `ST0` 中的值。

下面是使用恒等式 $10^x = 2^{x \cdot \log_2(10)}$ 计算 10^x 的一个示例。这段代码只适用于一部分 x , 它们使 `ST0` 不会落在前面提到的有效范围之外。

```
fld( x );  
fildl2t();  
fmul();  
f2xm1();  
fldl();  
fadd();
```

注意, 指令 `f2xm1` 计算的是 $2^x - 1$, 所以上面的代码最后对结果执行了加1操作。

2. fsin、fcos 和 fsincos 指令

这些指令首先从寄存器栈顶弹出数值, 然后计算正弦、余弦或同时计算两者, 最后把计算的结果放回栈顶。指令 `fsincos` 向栈推入原始数据计算结果的顺序是, 先正弦, 后余弦。因此, 指令执行完成后, `ST0` 中保存的是 $\cos(ST0)$, 而 `ST1` 中保存的是 $\sin(ST0)$ 。

这些指令都假定 `ST0` 中指定的是一个用弧度表示的角, 并且它必须满足 $-2^{63} < ST0 < +2^{63}$ 。如果原始操作数不在给定的范围内, 这些指令置位 C_2 标志, 同时保持 `ST0` 中的数值不变。可以使用 2π 作除数的 `fprem` 指令, 把操作数减小到一个合理的范围内。

这些指令将依据相应的计算结果置位栈错误/ C_1 、精度、下溢、非规格化和无效操作标志。

3. fptan 指令

指令 `fptan` 计算 `ST0` 的正切值并把结果推入栈, 然后再把数值 1.0 也推入栈。与指令 `fsin` 和 `fcos` 类似, `ST0` 的数值必须使用弧度表示, 同时要满足 $-2^{63} < ST0 < +2^{63}$ 。如果数值不在这个范围内, 指令 `fptan` 置位 C_2 指示操作没有发生。像使用指令 `fsin`、`fcos` 和 `fsincos` 一样, 可以使用 2π 作除数的 `fpreml` 指令, 把操作数减小到一个合理的范围内。

如果参数是无效的(如 0 或 π , 它们会导致除零错误), 结果将是一个不确定的值, 但不引发任何异常。指令 `fptan` 将根据操作的情况, 置位栈错误、精度、下溢、非规格化、无效操作、 C_2 和 C_1 位。

4. fpatan 指令

这条指令需要栈顶有两个数值。它把它们弹出并计算 $ST0 = \tan^{-1}(ST1/ST0)$ 。

结果是用弧度表示的栈上两数之比的反正切值。如果想对某个数值计算反正切, 可以使用指令 `fldl` 先创建适当的比率, 然后再执行 `fpatan` 指令。

如果计算过程中产生某些问题, 这条指令会影响栈错误/ C_1 、精度、下溢、非规格化和无效操作位。如果对结果进行了舍入操作, 则置位 C_1 条件码。

5. fyl2x 指令

这条指令需要栈顶有两个操作数: ST1 对应 y , ST0 对应 x 。它计算 $ST0 = ST1 * \log_2(ST0)$ 。
这条指令没有操作数(对指令本身来说)。它使用下面的语法:

```
fyl2x();
```

注意, 该指令计算以 2 为底的对数。当然, 通过与适当的常量相乘的方法能轻而易举地计算任意底的对数。

6. fyl2xp1 指令

这条指令需要栈顶有两个操作数: ST1 对应 y , ST0 对应 x 。它计算 $ST0 = ST1 * \log_2(ST0 + 1.0)$ 。
指令的语法为:

```
fyl2xp1();
```

该指令的其他方面与 fyl2x 相同。

6.5.10 其他指令

FPU 还提供了一些附加指令用于控制 FPU、同步操作、允许测试或置不同的状态位。这些指令包括 finit/fninit、fldcw、fstcw、fclex/fnclex 和 fstsw。

1. finit 和 fninit 指令

指令 finit 为能够进行正确的操作对 FPU 进行初始化。应用程序在执行其他 FPU 指令之前应该首先执行这条指令。这条指令把控制寄存器初始化为 \$37F, 状态寄存器初始化为 0, 标记字初始化为 \$FFFF。其他的寄存器不受影响。例如:

```
finit();  
fninit();
```

这两条指令之间的差别在于, 指令 finit 在初始化 FPU 之前首先检查是否有挂起的浮点异常, 而指令 fninit 不进行这样的检查。

2. fldcw 和 fstcw 指令

指令 fldcw 和 fstcw 需要一个 16 位的存储器操作数:

```
fldcw(mem16);  
fstcw(mem16);
```

这两条指令可以把一个存储单元的数据读入控制寄存器中(fldcw), 或把控制字保存到一个 16 位的存储单元(fstcw)。

当使用指令 fldcw 打开某个异常时, 如果相应的异常标志在被使能的同时被置位, 此时 FPU 在 CPU 执行下一条指令之前会立即产生一个中断。因此, 在改变 FPU 异常使能位之前, 应该使用指令 fclex 清除所有挂起的中断。

3. fclex 和 fnclex 指令

指令 fclex 和 fnclex 清除 FPU 状态寄存器中的所有异常位、栈错误位和忙标志。例如：

```
fclex();
fnclex();
```

这两条指令之间的差别与指令 finit 和 fninit 之间的差别相同。

4. fstsw 和 fnstsw 指令

这些指令把 FPU 的状态寄存器保存到一个 16 位的存储单元或寄存器 AX 中。

```
fstsw( ax )
fnstsw( ax )=
fstsw( mem16 )
fnstsw( mem16 )
```

这些指令与众不同之处在于，它们可以把 FPU 中的数值复制到一个 80x86 的通用寄存器中(明确地说就是 AX)。当然，允许把状态寄存器转移到 AX 中的目的，就是让 CPU 可以通过指令 sahf 查看条件码寄存器。指令 fstsw 和 fnstsw 之间的差别与指令 fclex 和 fnclex 之间的差别相同。

6.5.11 整数操作

FPU 提供一些专门的指令，它们把整数到扩展精度值的换算以及各种算术运算和比较操作结合起来。这些指令的形式如下：

```
fiadd( int_16_32 )
fisub( int_16_32 )
fisubr( int_16_32 )
fimul( int_16_32 )
fidiv( int_16_32 )
fidivr( int_16_32 )
ficom( int_16_32 )
ficomp( int_16_32 )
```

这些指令把 16 位或 32 位的整型操作数换算成一个 80 位的扩展精度浮点数值，然后使用这个换算后的数值作为源操作数来完成指定的操作。这些指令使用 ST0 作为目的操作数。

6.6 将浮点表达式转换成汇编语言

由于 FPU 寄存器的结构与 80x86 整数寄存器组的结构不同，因此，涉及浮点操作数的算术表达式转换与整数表达式的转换相比，也有些差别。因此，花些时间来讨论如何手动地把浮点表达式转换成汇编语言是很有必要的。

从一方面来说，把浮点表达式转换成汇编语言更容易些。Intel FPU 的栈结构使得算术表达式向汇编语言的转换变得容易。如果曾经使用过惠普公司的计算器，那么您对 FPU 的算术运算表示

应该很熟悉。因为，与惠普公司的计算器类似，FPU 使用后缀表示法(postfix notation)表示算术运算，后缀表示法也称为逆波兰表示法(Reverse Polish Notation, RPN)。一旦使用了后缀表示法，对表达式的转换将变得很方便，因为不必担心临时变量的分配问题，它们总是会保存在 FPU 栈中。

后缀表达式与标准的中缀表示法(infix notation)相对，把操作数放在运算符之前。下面给出了一些中缀表示法的简单示例以及相应的后缀表示法：

中缀表示法	后缀表示法
5 + 6	5 6 +
7 - 2	7 2 -
x * y	x y *
a / b	a b /

后缀表达式 5 6 + 的意思是“把 5 推入栈，把 6 推入栈，把栈顶数值(6)弹出并加到新的栈顶数值中”。听起来很熟悉吧！这正是指令 fld 和 fadd 所做的。事实上，可以使用下面的代码来完成这个计算：

```
fld( 5.0 );
fld( 6.0 );
fadd();          // 11.0 is now on the top of the FPU stack.
```

如您所见，后缀表示法是一种很方便的表示法，因为它很容易把这种代码转换成 FPU 指令。

后缀表示法的另一个优点是，它不需要括号。下面的示例演示了一些较复杂的中缀表达式到后缀表达式的转换：

中缀表示法	后缀表示法
(x + y) * 2	x y + 2 *
x * 2 - (a + b)	x 2 * a b + -
(a + b) * (c + d)	a b + c d + *

后缀表达式 x y + 2 * 的意思是“把 x 推入栈，把 y 推入栈，然后把栈顶数值相加(在栈上计算出 x+y)。然后，把 2 推入栈，最后把栈上的两个数值相乘(2 和 x+y)，产生结果 2*(x+y)”。同样，我们可以把这样的后缀表达式直接转换成汇编语言。下面的代码演示了上述每一个表达式的转换：

```
//      x y + 2 *
fld( x );
fld( y );
fadd();
fld( 2.0 );
fmul();

//      x 2 * a b + -
fld( x );
fld( 2.0 );
fmul();
fld( a );
fld( b );
fadd();
fsub();
```

```
//      a b +c d + *
      fld( a );
      fld( b );
      fadd();
      fld( c );
      fld( d );
      fadd();
      fmul();
```

6.6.1 将算术表达式转换成后缀表示法

由于把算术表达式转换成汇编语言涉及后缀表示法(RPN), 因此, 在讨论浮点表达式的转换之前, 我们先来考虑如何把一个算术表达式转换成后缀表达式。本节集中讨论 RPN 的转换。

对于那些只涉及 2 个操作数和 1 个运算符的简单表达式, 这种转换是很容易的。只要把运算符从中缀位置移到后缀位置上(就是说, 把运算符从两个操作数中间移到第二个操作数之后)。例如, $5+6$ 变为 $5\ 6\ +$ 。除了需要分离操作数以避免产生混淆以外(例如, 是 5 和 6 还是 56?), 简单中缀表达式到后缀表达式的转换十分简单。

对于复杂的表达式, 转换的方法是, 首先把简单子表达式转换成后缀表达式, 然后把转换后的子表达式作为剩余表达式的一个操作数来看待。下面的讨论将把已经完成的转换放在一个中括号中, 这样在转换的时候就很容易知道需要把哪些文本作为一个操作数来看待。

像整数表达式的转换那样, 转换最好是从最里面的括号开始, 然后, 在考虑优先权、结合性和其他带括号的子表达式的情况下, 一步步向外扩展。为了具体说明这种过程, 考虑下面的表达式:

```
x = ((y - z) * a) - (a + b * c) / 3.14159
```

最先把子表达式 $(y - z)$ 转换成后缀表达式:

```
x = ([y z -] * a) - (a + b * c) / 3.14159
```

用中括号把转换后的后缀表达式括起来, 这样可以把它与中缀表达式中的其他项区别开。这样只是为了使中间转换更易读。记得, 为了实现我们最终的转换, 我们将把中括号的内容作为一个操作数来看待。因此, 应当把 $[yz -]$ 看作一个简单的变量名或一个常量。

下一步是把子表达式 $([y z -] * a)$ 转换成后缀形式。这样就产生下面的形式:

```
x = [y z - a *] - (a + b * c) / 3.14159
```

接下来, 我们将对带括号的表达式 $(a+b*c)$ 实施这种操作。由于乘法的优先级比加法高, 所以先转换 $b*c$:

```
x = [y z - a *] - (a + [b c *]) / 3.14159
```

转换了“ $b*c$ ”之后, 我们将完成整个括号内表达式的转换:

```
x = [y z - a *] - [a b c * +] / 3.14159
```

这样就只剩下两个中缀运算符: 减法和除法。因为除法的优先级高, 所以我们先转换它:

$$x = [y \ z - a *] - [a - b \ c * + 3.14159 /]$$

最后，我们处理最后一个中缀运算：减法。这样我们就完成了这个表达式到后缀表达式的转换：

$$x = [y \ z - a *] [a - b \ c * + 3.14159 /] -$$

把我们附加的中括号去掉，得到真正的后缀表达式。得到的后缀表达式为：

$$x = y \ z - a * \ a \ b \ c * + 3.14159 / -$$

下面是另一个中缀表达式向后缀表达式转换的示例：

$$a = (x * y - z + t) / 2.0$$

第1步：对括号内的表达式进行转换。因为乘法具有最高的优先级，所以首先转换它：

$$a = ([x \ y *] - z + t) / 2.0$$

第2步：仍然在括号内转换。我们知道加法和减法具有相同的优先权，所以根据结合性来决定下一步该怎么做。这些运算符都是左结合的，因此我们必须按照从左到右的顺序来转换表达式。这意味着我们应该首先转换减法：

$$a = ([x \ y * \ z -] + t) / 2.0$$

第3步：现在转换括号内的加法运算符。由于我们完成了括号内的整个转换，所以可以把括号去掉：

$$a = [x \ y * \ z - \ t +] / 2.0$$

第4步：转换最后的中缀运算符(除法)。产生结果为：

$$a = [x \ y * \ z - \ t + \ 2.0 /]$$

第5步：去掉我们所加的中括号，完成最终的转换：

$$a = x \ y * \ z - \ t + \ 2.0 /$$

6.6.2 将后缀表达式转换成汇编语言

一旦把一个算术表达式转换成后缀表达式，完成它到汇编语言的转换将变得特别简单。您所要做的只是：当遇到一个操作数时，发出一条 fld 指令；当遇到一个运算符时，发出相应的算术运算指令。本节将使用上一节完成的示例来说明此过程多么简单。

$$x = y \ z - a * \ a \ b \ c * + 3.14159 / -$$

第1步：把 y 转换成 fld(y)。

第2步：把 z 转换成 fld(z)。

第3步：把 - 转换成 fsub()。

第4步：把 a 转换成 fld(a)。

第5步：把 * 转换成 fmul()。

第 6 步：继续按照从左到右的方式，为表达式产生下面的代码：

```

    fld( y );
    fld( z );
    fsub();
    fld( a );
    fmul();
    fld( a );
    fld( b );
    fld( c );
    fmul();
    fadd();
    fldpi();      // Loads pi (3.14159)
    fdiv();
    fsub();

    fstp( x );    // Store result away into x.

```

下面是对上一节第 2 个示例的转换：

```

a = x y * z - t + 2.0 /
    fld( x );
    fld( y );
    fmul();
    fld( z );
    fsub();
    fld( t );
    fadd();
    fld( 2.0 );
    fdiv();

    fstp( a );    // Store result away into a.

```

如您所见，如果您已经把中缀表达式转换成后缀表达式，最终的转换将变得非常简单。还应注意，与整数表达式的转换不同，这里不需要任何显式的临时变量。FPU 栈提供了这种临时变量的保存³。由于这些原因，浮点表达式到汇编语言的转换实际上要比整数表达式的转换更容易。

6.7 HLA 标准库对浮点算术运算的支持

第 2 章只简单提到了输入函数 `stdin.getf`，但是没有讨论在从标准输入设备读入浮点数值以后该函数把它们返回到哪里。现在已经讨论了 80x86 的浮点扩展，可以完成对这个标准库函数的讨论了。函数 `stdin.getf` 从标准输入设备读入一串字符，把这些字符转换成 80 位浮点数，然后将结果保存到 FPU 栈上的 ST0 中。

HLA 标准库还提供了 `math.hhf` 模块，其中包含了 FPU 不直接支持的一些数学函数以及 FPU 部分支持的一些函数(如 `sine` 和 `cosine`)。 `math.hhf` 模块提供的一些函数包括 `acos`、`acot`、`acsc`、`asec`、

³ 当然，这是假定了计算没有复杂到超出了 FPU 栈的 8 元素限制。

asin 、 cot 、 csc 、 sec 、 2^x 、 10^x 、 y^x 、 e^x 、 \log 和 \ln 。关于这些函数和 HLA 标准库支持的其他数学函数的更多信息，请参看 HLA 标准库的文档。

6.8 更多信息

Intel/AMD 处理器手册完整地描述了每条整数和浮点算术指令的运算方式，并详细说明了这些指令如何影响 EFLAGS 和 FPU 状态寄存器中的条件码位和其他标志。要想写出最优的汇编语言代码，必须非常熟悉算术运算指令对执行环境的影响，所以应该花一些时间阅读 Intel/AMD 手册。

HLA 标准库提供的许多浮点函数都没有对应的单条机器指令。HLA 标准库还提供了 `math.sin` 和 `math.cos` 这样的函数，它们可以克服处理器本身的机器指令的局限。更多细节可以参看 HLA 标准库参考手册。另外，HLA 标准库提供了源代码形式，所以可以查看这些数学函数的实现来了解浮点编码的更多示例。

第 8 章将讨论多精度整数算术运算。如果想了解如何处理大于 32 位的整数操作数的更多细节，可以阅读该章。

较新型的一些 CPU 上提供了 80x86 SSE 指令集，它们支持使用 SSE 寄存器组的浮点算术运算。可以访问 <http://webster.cs.ucr.edu/> 或参考 Intel/AMD 文档来了解有关 SSE 浮点指令集的更多信息。

第 7 章

低级控制结构



本章讨论“纯”汇编语言的控制语句。必须掌握这些低级控制结构，才能称得上是一名真正的汇编程序员。在学习完本章后，您将学会用低级 80x86 机器指令合成高级控制语句。

本章最后一节讨论混合(hybrid)控制结构，这种结构同时使用了 HLA 高级控制语句和 80x86 控制指令。使用混合控制结构可以同时获得低级控制语句的高效性和高级控制语句的可读性。高级汇编程序员可以在不牺牲程序效率的情况下，使用这种混合语句来提高程序的可读性。

7.1 低级控制结构

到现在为止，您在程序中所看到和使用的大多数控制结构都和一些高级语言如 Pascal、C++、Ada 等的控制结构类似。虽然这些控制结构使得学习汇编语言更加容易，但它们并不是真正的汇编语言语句。这些控制结构通过 HLA 编译器翻译成纯粹的机器指令序列，这些机器指令序列能够获得和高级控制结构相同的结果。本书使用高级控制结构，使您在不需要了解所有内容的情况下就能够学习汇编语言。但是，现在应该将这些高级控制结构放在一边，而使用真正的汇编语言——低级控制结构来编写程序。

7.2 语句标号

汇编语言低级控制结构在代码中大量地使用标号(label)。低级控制结构经常需要将控制权从程序中的一点转移到另一点，转移的目标通常是使用语句标号来指定。语句标号包含一个有效(唯

一)的 HLA 标识符和一个冒号，例如：

```
aLabel:
```

与过程、变量、常量标识符一样，要尽量选择起描述作用的、有意义的名称作为语句标号。在上面的示例中，语句标号 `aLabel` 的描述作用和意义就不是很好。

语句标号不同于其他大多数 HLA 标识符，它有一个重要的特征：在使用标号之前，不需要声明。这一点非常重要。因为在代码中，低级控制结构经常要将控制权转移到后面的标号处，这时该标号还没有定义，但却引用了它。

对标号可以有 3 种操作：使用跳转(`goto`)指令将控制权转移到一个标号、使用 `call` 指令调用一个标号、获得标号的地址。对标号很少有其他直接的操作(当然，您也很少需要对标号进行其他操作，所以这并不是一个限制)。程序清单 7-1 演示了两种获得标号地址的方式(使用 `lea` 指令和使用 `&` 取址操作符)，以及如何打印该地址：

程序清单 7-1 显示程序中语句标号的地址

```
program labelDemo;
#include( "stdlib.hhf" );

begin labelDemo;

    lbl1:
        lea( ebx, lbl1 );
        mov( &lbl2, eax );
        stdout.put( "&lbl1=$", ebx, " &lbl2=", eax, nl );
    lbl2:

end labelDemo;
```

HLA 还允许使用语句标号的地址来初始化双字变量，但是，此时对于标号还有一些限制。最重要的限制是：语句标号必须与变量声明在相同的语法层定义。也就是说，如果在主程序的变量声明的初始值设定项中引用一个语句标号，该语句标号也必须在主程序中。相反，如果在局部变量声明中使用了语句标号的地址，则该语句标号也必须出现在与局部变量相同的过程中。程序清单 7-2 描述了如何使用语句标号初始化变量：

程序清单 7-2 使用语句标号地址初始化双字变量

```
program labelArrays;
#include( "stdlib.hhf" );

static
    labels:dword[2] := [ &lbl1, &lbl2 ];

procedure hasLabels;
    static
        stmtLbls: dword[2] := [ &label1, &label2 ];

    begin hasLabels;
```

```

label1:
    stdout.put
    (
        "stmtLbls[0]= $", stmtLbls[0], nl,
        "stmtLbls[1]= $", stmtLbls[4], nl
    );

label2:

end hasLabels;

begin labelArrays;

    hasLabels();
    lbl1:

        stdout.put("labels[0]= $", labels[0], " labels[1]=", labels[4], nl);

    lbl2:

end labelArrays;

```

当然，有时需要引用一个不在当前过程中的标号，但因为这种需要很少，本书将不描述所有的细节。如果需要，可以查阅 HLA 文档了解更详细的内容。

7.3 无条件控制转移(jmp)

jmp 指令将控制无条件地转移到程序中的另一位置。这条指令有 3 种形式：一种直接跳转和两种间接跳转。指令形式为下列 3 种形式之一：

```

jmp label;
jmp(_reg32_);
jmp(_mem32_);

```

上面的第一条跳转指令是直接跳转指令，通常使用一个语句标号作为目标地址。语句标号通常在可执行机器指令的同一行中出现，或者在可执行机器指令的前一行单独出现。直接跳转指令完全等价于高级语言中的 goto 语句¹。例如：

```

<< statements >>
    jmp laterInPgm;

laterInPgm:

    << statements >>

```

¹ 在高级语言中，通常不提倡使用 goto 语句。但在汇编语言中，您会发现 jmp 指令是非常有必要的。

第二种形式的 `jmp` 指令 `jmp(reg32)` 是寄存器间接跳转指令。这种指令将控制转移给指定 32 位通用寄存器中地址所指的指令。为了使用这种形式的 `jmp` 指令，在执行 `jmp` 指令之前必须将目标地址装入一个 32 位寄存器中。可以利用这种指令实现状态机：首先在寄存器中装入程序中不同位置的标号地址，然后通过程序中公共的一条间接跳转指令，将控制转移到寄存器指定的标号地址处。程序清单 7-3 描述了如何以这种方式使用 `jmp` 指令。

程序清单 7-3 寄存器间接跳转指令的用法

```

program regIndJump;
#include( "stdlib.hhf" );

static
    i:int32;

begin regIndJump;

    // Read an integer from the user and set ebx to
    // denote the success or failure of the input.

    try

        stdout.put( "Enter an integer value between 1 and 10:" );
        stdin.get( i );
        mov( i, eax );
        if( eax in 1..10 )then

            mov( &GoodInput, ebx );

        else

            mov( &valRange, ebx );

        endif;

    exception( ex.ConversionError )

        mov( &convError, ebx );

    exception( ex.ValueOutOfRange )

        mov( &valRange, ebx );

endtry;

// Okay, transfer control to the appropriate
// section of the program that deals with
// the input.

jmp( ebx );

valRange:
    stdout.put( "You entered a value outside the range 1..10" nl );
    jmp Done;

convError:

```

```

        stdout.put( "Your input contained illegal characters" nl );
        jmp Done;

    GoodInput:
        stdout.put( "You entered the value ", i, nl );
    Done:

end regIndJump;

```

第三种形式的 `jmp` 指令是存储器间接跳转指令。这种形式的 `jmp` 指令首先从指定的存储单元取出一个双字数值作为目标地址，然后将控制转移给该目标地址处的指令。它和寄存器间接跳转指令类似，不同之处只是目标地址从存储器获得，而不是从寄存器获得。程序清单 7-4 描述了这种 `jmp` 指令的简单应用：

程序清单 7-4 存储器间接跳转指令的用法

```

program memIndJump;
#include( "stdlib.hhf" );

static
    LabelPtr:dword := &stmtLabel;

begin memIndJump;

    stdout.put( "Before the JMP instruction" nl );
    jmp( LabelPtr );

    stdout.put( "This should not execute" nl );

    stmtLabel:

        stdout.put( "After the LabelPtr label in the program" nl );

end memIndJump;

```

警告：

与 HLA 高级控制结构不同，低级 `jmp` 指令可能带来很多麻烦。尤其是当没有使用有效的指令地址对寄存器进行初始化时，就马上通过该寄存器跳转，这会导致不可预料的结果(虽然这时通常导致一个通用保护错误)。同样地，当没有使用一个合法的指令地址给一个双字变量赋初值，就通过该存储器地址进行间接跳转时，很可能会导致整个程序的崩溃。

7.4 条件跳转指令

虽然 `jmp` 指令提供了控制转移，但是当需要作出判断以实现 `if` 和 `while` 等语句时，它用起来十分麻烦。80x86 的条件跳转指令可以完成这个任务。

条件跳转指令检查一个或多个 CPU 标志，判断它们是否匹配某个模式：如果标志匹配成功，该指令就将控制转移到目标位置；如果匹配失败，CPU 忽略该条件跳转指令而继续执行下一条指令。一些条件跳转指令只是简单测试符号、进位、溢出和零标志的设置。例如，在执行一条 `shl` 指令后，可以测试进位标志，来判断 `shl` 是否从操作数的高位位移出一位。类似地，也可以在一

条 test 指令后测试零标志，来检查结果是否为 0。大多数情况下，在 cmp 指令之后执行条件跳转指令。cmp 指令设置标志，以便判断小于、大于、等于等情况。

条件跳转指令的形式如下：

```
jcc label;
```

其中，jcc 中的 cc 必须用表示测试条件类型的字符序列替换。这些字符和 setcc 指令使用的一样。例如，js 表示根据符号标志是否被置位来决定是否跳转。一个典型的 js 指令如下所示：

```
js ValueIsNegative;
```

在这个示例中，如果符号标志被置位，则 js 指令将控制转移到 ValueIsNegative 标号处；如果符号标志清零，则将控制直接转移给 js 指令后的指令。

与无条件 jmp 指令不同，条件跳转指令不提供间接跳转的形式。唯一允许的形式是跳转到程序中某一标号处。

注意

Intel 文档为许多条件跳转指令定义了多种替代名或指令别名。

表 7-1、表 7-2 和表 7-3 列出了每个指令所有的别名。这些表格也列出了表示相反分支的指令，很快您将明白这些相反分支指令的作用。

表 7-1 测试标志的 jcc 指令

指 令	描 述	条 件	别 名	相 反 指 令
jc	如果进位标志被置位则跳转	进位标志=1	jb、jnae	jnc
jnc	如果进位标志没有置位则跳转	进位标志=0	jnb、jae	jc
jz	如果零标志被置位则跳转	零标志=1	je	jnz
jnz	如果零标志没有置位则跳转	零标志=0	jne	jz
js	如果符号标志被置位则跳转	符号标志=1		jns
jns	如果符号标志没有置位则跳转	符号标志=0		js
jo	如果溢出标志置位则跳转	溢出标志=1		jno
jno	如果溢出标志没有置位则跳转	溢出标志=0		jo
jp	如果奇偶校验标志被置位则跳转	奇偶校验标志=1	jpe	jnp
jpe	如果奇偶校验标志为偶校验则跳转	奇偶校验标志=1	jp	jpo
jnp	如果奇偶校验标志没有置位则跳转	奇偶校验标志=0	jpo	jp
jpo	如果奇偶校验标志为奇校验则跳转	奇偶校验标志=0	jnp	jpe

表 7-2 用于无符号比较的 jcc 指令

指 令	描 述	条 件	别 名	相 反 指 令
ja	如果超过(>)则跳转	进位标志=0，零标志=0	jnb	jna
jnb	如果不低于或等于(不<=)则跳转	进位标志=0，零标志=0	ja	jbe

(续表)

指 令	描 述	条 件	别 名	相 反 指 令
jae	如果超过或等于(\geq)则跳转	进位标志=0	jnc、jnb	jnae
jnb	如果不低于则跳转(不 $<$)	进位标志=0	jnc、jae	jb
jb	如果低于($<$)则跳转	进位标志=1	jc、jnae	jnb
jnae	如果不超过或等于(不 \geq)则跳转	进位标志=1	jc、jb	jae
jbe	如果低于或等于(\leq)则跳转	进位标志=1 或 零标志=1	jna	jnb
jna	如果不超过(不 $>$)则跳转	进位标志=1 或 零标志=1	jbe	ja
je	如果相等($=$)则跳转	零标志=1	jz	jne
jne	如果不相等(\neq)则跳转	零标志=0	jnz	je

表 7-3 用于有符号比较的 jcc 指令

指 令	描 述	条 件	别 名	相 反 指 令
jg	如果大于($>$)则跳转	符号标志=溢出标志或零标志=0	jnle	jng
jnle	如果小于或等于(\leq)则跳转	符号标志=溢出标志或零标志=0	jg	jle
jge	如果大于或等于(\geq)则跳转	符号标志=溢出标志	jnl	jge
jnl	如果不小于(不 $<$)则跳转	符号标志=溢出标志	jge	jl
jl	如果小于($<$)则跳转	符号标志 \neq 溢出标志	jnge	jnl
jnge	如果大于或等于(\geq)则跳转	符号标志 \neq 溢出标志	jl	jge
jle	如果小于或等于(\leq)则跳转	符号标志 \neq 溢出标志或零标志=1	jng	jnle
jng	如果不大于(不 $>$)则跳转	符号标志 \neq 溢出标志或零标志=1	jle	jg
je	如果等于($=$)则跳转	零标志=1	jz	jne
jne	如果不等于(\neq)则跳转	零标志=0	jnz	je

接下来将对“相反指令”一列进行简单的说明。在许多情况下,需要产生与某条分支指令条件相反的分支(在本章后面会给出示例)。除了两个例外,都可以按下面的简单规则(后面统称为“N/No-N”规则)产生相反分支:

- 如果 jcc 的第二个字母不是 n,则在 j 后面插入一个 n。例如: je 变成 jne, jl 变成 jnl。
- 如果 jcc 的第二个字母是 n,则去掉指令中的 n。例如: jng 变成 jg, jne 变成 je。

不遵循这两条规则的两个例外是 jpe 和 jpo。这两个例外并不会导致什么问题,因为:①很少需要测试奇偶标志;②可以使用别名 jp 和 jnp 替代 jpe 和 jpo。而“N/No-N”规则对 jp 和 jnp 是适用的。

虽然 jge 是 jl 的相反指令,但是建议使用 jnl 作为 jl 的相反指令。因为很容易误认为“大于是小于的相反”,从而把 jg 当作 jl 的相反指令。可以坚持使用“N/No-N”规则以避免这种混淆。

80x86 条件跳转指令提供了这样的能力:根据判断条件将程序流分支到两条路径中的某一条。例如,要实现:如果 BX 等于 CX,则寄存器 AX 的值加 1。可以使用下面的代码来完成该功能:

```

    cmp( bx,cx );
    jne SkipStmts;
    inc( ax );
SkipStmts:

```

其中的诀窍是使用相反分支指令来跳过在条件满足的情况下需要执行的指令。请坚持使用前面介绍的“N/No N”规则来选择相反分支指令。

使用条件跳转指令还可以实现循环。例如,下面的代码序列实现了从用户输入读入一串字符,并将字符存储到一个数组的连续元素中,直到用户按下回车键。

```

    mov( 0, edi );
RdLnLoop:
    stdin.getc();           // Read a character into the al register.
    mov( al, Input[ edi ] ); // Store away the character.
    inc( edi );             // Move on to the next character.
    cmp( al, stdio.cr );    // See if the user pressed Enter.
    jne RdLnLoop;

```

与 `setcc` 指令类似,条件跳转指令分为两类——测试特殊处理器标志的条件跳转指令(例如 `jz`、`jc`、`jno`)和测试某些条件(小于、大于等)的条件跳转指令。当测试某个条件时,条件跳转指令通常紧跟在一个 `cmp` 指令之后。`cmp` 指令设置标志后,如果是无符号比较,使用 `ja`、`jae`、`jb`、`jbe`、`je` 或 `jne` 指令测试这些标志来判断是否小于、小于等于、等于、不等于、大于或大于等于;如果是有符号比较,则使用 `jl`、`jle`、`je`、`jne`、`jg`、`jge` 指令。

条件跳转指令测试 80x86 标志,但不影响它们。

7.5 “中级”控制结构: `jt` 和 `jf`

HLA 提供两条特殊的条件跳转指令: `jt`(条件为真则跳转)和 `jf`(条件为假则跳转)。它们的语法如下:

```

jt( boolean_expression ) target_label;
jf( boolean_expression ) target_label;

```

其中, *boolean_expression* 是 `if..endif` 或其他 HLA 高级语言语句允许的标准 HLA 布尔表达式。这些指令先对布尔表达式求值,如果求出的值为真(`jt`)或为假(`jf`),则跳转到指定的标号。

这些并不是真正的 80x86 指令,而是通过 HLA 编译器编译成一条或多条功能等效(能够获得相同结果)的 80x86 机器指令序列。通常,在主程序代码中,不要使用这两种指令。与 `if..endif` 语句相比,它们并不提供更多好处,而且它们的可读性也并不比它们编译后的纯汇编代码序列好多少。HLA 提供这种中级的指令只是为了便于用户使用宏创建自己的高级控制结构(参见第 9 章和 HLA 参考手册)。

7.6 使用汇编语言实现常用控制结构

因为本章的主要目的是教您如何使用低级机器指令实现判定、循环以及其他控制结构，一个明智的做法是首先教会您如何使用纯汇编语言来实现高级语句。下面将介绍这些内容。

7.7 判定

最基本的判定是代码中的某种分支，它根据条件在两条可能的执行路径中选择一条来执行。通常情况(不是所有情况)下，条件指令序列使用条件跳转指令实现。条件指令对应于 HLA 中的 if..then..endif 语句：

```
if( expression ) then
    << statements >>
endif;
```

在处理条件语句时，汇编语言提供了更大的灵活性。考虑下面的 C/C++ 语句：

```
if( (( x < y ) && ( z > t )) || ( a != b ))
    stmt1;
```

通过“蛮力”方式可以把这条语句转换成下面的汇编语言语句：

```
mov( x, eax );
cmp( eax, y );
setl( bl );           // Store x<y in bl.
mov( z, eax );
cmp( eax, t );
setg( bh );           // Store z>t in bh.
and( bh, bl );        // Put (x<y) && (z>t) into bl.
mov( a, eax );
cmp( eax, b );
setne( bh );          // Store a != b into bh.
or( bh, bl );         // Put (x<y) && (z>t) || (a!=b) into bl
je SkipStmt1;         // Branch if result is false.
```

```
<< Code for Stmt1 goes here. >>
```

```
SkipStmt1:
```

正如您所看到的，上面的示例使用了相当多的条件语句来处理表达式。它粗略等价于下面的 C/C++ 语句：

```
bl = x < y;
bh = z > t;
bl = bl && bh;
bh = a != b;
bl = bl || bh;
```

```
if( b1 )
    << Stmt1 >>;
```

现在，请比较一下下面改进了的代码：

```
mov( a, eax );
cmp( eax, b );
jne DoStmt;
mov( x, eax );
cmp( eax, y );
jnl SkipStmt;
mov( z, eax );
cmp( eax, t );
jng SkipStmt;

DoStmt:
    << Place code for Stmt1 here. >>

SkipStmt:
```

观察上面的代码序列，有两件事是显而易见的：第一，C/C++(或其他高级语言)中一条简单的条件语句可能需要多条汇编语言的条件跳转指令才能实现；第二，条件指令序列中，复杂表达式的组织方式会影响代码的效率。因此，在汇编语言中处理条件指令序列时要特别注意。

条件语句可以分为三类：if 语句、switch/case 语句和间接跳转。下面的章节将介绍这些程序结构、如何使用这些结构以及如何把它们转换成汇编语言形式。

7.7.1 if..then..else 序列

最常见的条件语句是 if..then..endif 和 if..then..else..endif 语句。这两种语句的形式如图 7-1 所示。

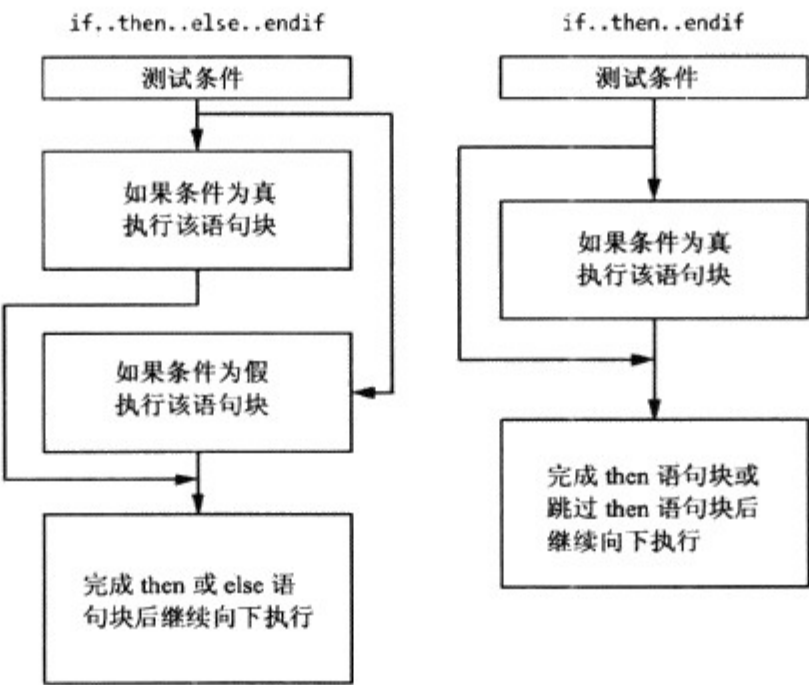


图 7-1 if..then..else..endif 和 if..then..endif 语句的流程图

If..then..endif 语句是 if..then..else..endif 语句的特殊形式(包含一个空 else 块)。因此，这里仅考虑更通用的 if..then..else..endif 形式。使用 80x86 汇编语言实现 if..then..else..endif 语句的基本代码如下：

```
<< Sequence of statements to test some condition >>
    jcc ElseCode;
<< Sequence of statements corresponding to the THEN block >>

    jmp EndOfIf;
```

ElseCode:

```
<< Sequence of statements corresponding to the ELSE block >>
```

EndOfIf:

注意, `jcc` 代表某个条件跳转指令。例如, 要把下面的 C/C++ 语句转换成汇编语言:

```
if( a == b )
    c = d;
else
    b = b + 1;
```

可以用下面的 80x86 代码实现:

```
    mov( a, eax );
    cmp( eax, b );
    jne ElsePart;
    mov( d, c );
    jmp EndOfIf;

ElseBlk:
    inc( b );

EndOfIf:
```

对于像 `(a==b)` 这样的简单表达式, 产生 `if..then..else..endif` 语句的汇编语言代码非常简单。当表达式变复杂时, 对应的汇编语言代码也相应地变得复杂。请考虑下面的 C/C++ 的 `if` 语句:

```
if( (( x > y ) && ( z < t )) || ( a != b ))
    c = d;
```

当要把复杂的 `if` 语句转换成汇编语言代码时, 您会发现, 如果首先将 `if` 语句拆分成下面 3 个不同的 `if` 语句序列, 就可以简化转换。

```
if( a != b ) c = d;
else if( x > y )
    if( z < t )
        c = d;
```

这种转换的依据在于下面等价的 C/C++ 语句:

```
if( expr1 && expr2 ) stmt;
```

等价于

```
if( expr1 ) if( expr2 ) stmt;
```

以及

```
if( expr1 || expr2 ) stmt;
```

等价于

```
if( expr1 ) stmt;
else if( expr2 ) stmt;
```

如果写成汇编语言，前面的 if 语句就变成：

```
// if( (( x > y ) && ( z < t )) || ( a != b ))
//      c = d;

        mov( a, eax );
        cmp( eax, b );
        jne DoIf;
        mov( x, eax );
        cmp( eax, y );
        jng EndOfIf;
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

DoIf:
        mov( d, eax );
        mov( eax, c );

EndOfIf:
```

可能您会发现，用来测试一个条件所需要的代码很可能比真正要执行的 **else** 和 **then** 语句块中的语句还要复杂。测试条件要比实际操作结果花费更大的精力，尽管看上去有些矛盾，但却经常发生。因此，您也必须接受这个现实。

在汇编语言中，关于复杂条件语句最大的一个问题可能就是：弄清代码到底完成什么操作。与汇编语言相比，高级语言的一个很大优势就是提供了可读性好、易理解的表达式。高级版本意味着可以自说明(**self-documenting**)，而汇编语言却隐藏了代码的本质。因此，好的注释成为使用汇编语言实现 **if..then..else..endif** 语句的必要组成成分。上面示例的一个更优雅的实现如下：

```
// if (( x > y ) && ( z < t )) or ( a != b ) c = d;
// Implemented as:
// if (a != b) then goto DoIf;

        mov( a, eax );
        cmp( eax, b );
        jne DoIf;

// if not ( x > y ) then goto EndOfIf;

        mov( x, eax );
        cmp( eax, y );
        jng EndOfIf;

// if not ( z < t ) then goto EndOfIf;
```

```

        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

```

```

// then block:

```

```

DoIf:

```

```

        mov( d, eax );
        mov( eax, c );

```

```

// End of if statement

```

```

EndOfIf:

```

必须承认, 对于这样一个简单示例, 上面一些注释显得有点多余。事实上, 下面的代码就足够了:

```

// if( (( x > y ) && ( z < t )) || ( a != b )) c = d;

```

```

// Test the boolean expression:

```

```

        mov( a, eax );
        cmp( eax, b );
        jne DoIf;
        mov( x, eax );
        cmp( eax, y );
        jng EndOfIf;
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

```

```

// then block:

```

```

DoIf:

```

```

        mov( d, eax );
        mov( eax, c );

```

```

// End of if statement

```

```

EndOfIf:

```

但是, 当您的 if 语句变得复杂时, 注释的多少(与质量)就变得很重要了。

7.7.2 将 HLA 的 if 语句翻译成纯汇编语言

把 HLA 的 if 语句翻译成纯汇编语言是非常简单的。HLA 的 if 语句支持的布尔表达式是经过专门选择的, 很容易被扩展为几条简单的机器指令。下面将讨论如何将这布尔表达式转换成纯机器代码。

```

if(flag_specification) then stmts endif;

```

这种形式的 HLA if 语句可能是最容易转换的。如果某个标志置位(或清零), 为了马上执行紧跟在 then 关键字后的代码, 只须跳过标志清零(或置位)时所执行的代码。这只需要使用一条条

件跳转指令，下面的示例说明了如何实现：

```
// if( @c ) then inc( eax ); endif;

    jnc SkipTheInc;

    inc( eax );

SkipTheInc:

// if( @ns ) then neg( eax ); endif;

    js SkipTheNeg;

    neg( eax );

SkipTheNeg:
```

if(register) then stmts endif;

这种形式的 if 语句使用 **test** 指令检查特定的寄存器是否为 0。如果寄存器值为 0(假)，则使用一条 **jz** 指令跳向 **then** 子句之后的语句。将这条语句转换为汇编语言需要一条 **test** 指令和一条 **jz** 指令，如下面的示例所示：

```
// if( eax ) then mov( false, eax ); endif;

    test( eax, eax );
    jz DontSetFalse;

    mov( false, eax );

DontSetFalse:

// if( al ) then mov( bl, cl ); endif;

    test( al, al );
    jz noMove;

    mov( bl, cl );

noMove:
```

if(!register) then stmts endif;

这种形式的 if 语句使用 **test** 指令检查特定的寄存器是否为 0。如果寄存器值不为 0(真)，则使用一条 **jnz** 指令跳向 **then** 子句之后的语句。把这条语句转换成汇编语言需要使用一条 **test** 指令和一条 **jnz** 指令，方法和上面的示例一样。

if(boolean_variable) then stmts endif;

这种形式的 if 语句比较布尔变量是否为 0(假)，如果布尔变量为 0，则跳向分支语句。HLA 用下面的方法实现这条语句：首先使用 **cmp** 指令来比较布尔变量是否为 0，如果为 0，则使用一条 **jz(je)** 指令跳向 **then** 包含的语句。下面的示例说明了这个转换过程：


```
// if( bool ) then mov( 0, al ); endif;
```

```
    cmp( bool, false );
    je SkipZeroAL;
```

```
    mov( 0, al );
```

```
    SkipZeroAL:
```

if(!boolean_variable) then stmts endif;

这种形式的 if 语句比较布尔变量是否为 0(假)，如果布尔变量为 F，则跳向分支语句(和上一个示例条件相反)。HLA 用下面的方法实现这个语句：首先使用 `cmp` 指令来比较布尔变量是否为 0，如果为 1，使用一条 `jnz(jne)` 指令跳向 `then` 包含的语句。下面的示例说明了这种转换：

```
// if( !bool ) then mov( 0, al ); endif;
```

```
    cmp( bool, false );
    jne SkipZeroAL;
```

```
    mov( 0, al );
```

```
    SkipZeroAL:
```

if(mem_reg_relop mem_reg_const) then stmts endif;

HLA 将这种形式的 if 语句翻译成一条 `cmp` 指令和一条条件跳转指令，如果与 `relop` 运算符指定的条件相反，该条件跳转指令跳向 `then` 包含的语句。表 7-4 列出了运算符和条件跳转指令的对应关系。

表 7-4 if 语句对应的条件跳转指令

关系运算符	两个操作数都为无符号数时的 条件跳转指令	有一个操作数为有符号数时的 条件跳转指令
<code>=</code> 或 <code>==</code>	<code>jnc</code>	<code>jnc</code>
<code><></code> 或 <code>!=</code>	<code>je</code>	<code>je</code>
<code><</code>	<code>jnb</code>	<code>jnl</code>
<code><=</code>	<code>jnb</code>	<code>jnl</code>
<code>></code>	<code>jna</code>	<code>jng</code>
<code>>=</code>	<code>jnae</code>	<code>jnge</code>

下面是将 if 语句翻译成纯汇编语言的一些示例，这些 if 语句使用了包含关系运算符的表达式。

```
// if( al == ch ) then inc( cl ); endif;
```

```
    cmp( al, ch );
    jne SkipIncCL;
```

```
    inc( cl );
```

```

SkipIncCL:
// if( ch >= 'a') then and( $5f, ch ); endif;

    cmp( ch, 'a');
    jnae NotLowerCase
        and( $5f, ch );

NotLowerCase:
// if( (type int32 eax ) < -5 ) then mov( -5, eax ); endif;

    cmp( eax, -5 );
    jnl DontClipEAX;

    mov( -5, eax );

DontClipEAX:
// if( si <> di ) then inc( si ); endif;

    cmp( si, di );
    je DontIncSI;

    inc( si );

DontIncSI:

```

if(reg/mem in *LowConst*..*HiConst*) then *stmts* endif;

HLA 将这种形式的 if 语句翻译成一对 cmp 指令和一对条件跳转指令。它将寄存器或存储器中的值与 *LowConst* 值进行比较, 如果小于(有符号数)或低于(无符号数)则跳过 then 子句之后的语句。如果寄存器或存储器中的值大于或等于 *LowConst*, 则代码进入第二个 cmp/条件跳转对, 在这里寄存器或存储器的值与 *HiConst* 值比较。如果寄存器的值大于(高于)*HiConst* 值, 则条件跳转指令跳过 then 子句包含的语句。请看下面这个示例:

```

// if( eax in 1000..125_000 ) then sub( 1000, eax ); endif;

    cmp( eax, 1000 );
    jb DontSub1000;
    cmp( eax, 125_000 );
    ja DontSub1000;

    sub( 1000, eax );

DontSub1000:

// if( i32 in -5..5 ) then add( 5, i32 ); endif;

    cmp( i32, -5 );
    jl NoAdd5;
    cmp( i32, 5 );
    jg NoAdd5;

    add( 5, i32 );

NoAdd5:

```

```
if(reg/mem not in LowConst..HiConst ) then stmts endif;
```

这种形式的 HLA if 语句测试寄存器或存储器的值是否在指定的范围外。它的实现和上面的代码非常类似,不同的是如果寄存器或存储器的值小于 *LowConst* 值或大于 *HiConst* 值,则分支到 then 子句;如果寄存器或存储器的值处于这两个常数指定的范围之间,则跳过 then 子句。下面的代码说明了如何实现这种转换。

```
// if( eax not in 1000..125_000 ) then add( 1000, eax ); endif;
```

```
    cmp( eax, 1000 );
    jb Add1000;
    cmp( eax, 125_000 );
    jbe SkipAdd1000;
```

```
    Add1000:
        add( 1000, eax );
```

```
    SkipAdd1000:
```

```
// if( i32 not in -5..5 ) then mov( 0, i32 ); endif;
```

```
    cmp( i32, -5 );
    jl Zeroi32;
    cmp( i32, 5 );
    jle SkipZero;
```

```
    Zeroi32:
        mov( 0, i32 );
```

```
    SkipZero:
```

7.7.3 使用完整布尔求值实现复杂的 if 语句

许多布尔表达式都包含逻辑与(and)或逻辑或(or)操作,本节将讨论如何把布尔表达式转换成汇编语言。有两种方法可以将包含逻辑与、逻辑或的复杂布尔表达式转换成汇编语言:使用完整布尔求值或使用短路布尔求值。本节讨论完整布尔求值,下一节讨论短路布尔求值。

使用完整布尔求值实现 if 语句中的布尔表达式的机理,与将算术表达式转换成汇编语言的机理基本相同。事实上,上一章关于算术运算的讨论已经包含了这个转换过程。关于这个过程唯一需要注意的是:不需要将最终的布尔结果存储到某个变量中。一旦表达式求值完成,就可以判断结果是否为真(1 或非 0)或假(0),然后决定是否执行 if 语句的 then 部分。如上一节的示例所示:如果前一条逻辑指令(and/or)执行的结果为假,则设置零标志位;如果结果为真,则清除零标志位。利用这一点,可以避免显式地检查布尔求值的结果。请考虑下面的 if 语句及其使用完整布尔求值转换后的汇编语言代码:

```
// if( ( ( x < y ) && ( z > t ) ) || ( a != b ) )
//    << Stmt1 >>;
//
//    mov( x, eax );
```

```

cmp( eax, y );
setl( bl );      // Store x<y in bl.
mov( z, eax );
cmp( eax, t );
setg( bh );      // Store z>t in bh.
and( bh, bl );   // Put (x<y) && (z>t) into bl.
mov( a, eax );
cmp( eax, b );
setne( bh );     // Store a != b into bh.
or( bh, bl );    // Put (x<y) && (z>t) || (a != b) into bl.
je SkipStmt1;    // Branch if result is false.

```

<< Code for Stmt1 goes here. >>

SkipStmt1:

上面这些代码首先计算 BL 寄存器中的布尔值，在计算结束后，判断结果值是否为真。如果结果为假，则跳过与 Stmt1 关联的代码段。在这个示例中，需要注意的是：每一条计算该布尔值的指令(直到 je 指令)，程序都要执行。

7.7.4 短路布尔求值

如果您愿意花更多的精力研究一条复杂布尔表达式，那么使用短路布尔求值，可以将它转换成更短更快的汇编语言指令序列。短路布尔求值试图通过计算部分指令就确定整个布尔表达式的值。因为只执行部分指令，而且短路布尔求值不需要任何临时寄存器，所以 HLA 在将复杂布尔表达式转换成汇编语言时，使用短路布尔求值。

为了理解短路布尔求值如何工作，考虑表达式 $a \&\& b$ 。一旦确定 a 值为假，就没有必要计算 b 值，因为此时已可以肯定表达式值不会为真。如果 a 、 b 表示子表达式，而不是变量，就可以看出使用短路布尔求值带来的好处。作为一个具体的示例，考虑前面章节提到的子表达式 $((x < y) \&\& (z > t))$ 。一旦确定 x 不小于 y ，就没有必要检查 z 是否大于 t ，因为这时不管 z 和 t 取什么值，表达式始终为假。下面的代码段说明了如何使用短路布尔求值实现这个表达式：

// if ((x<y) && (z>t)) then ...

```

mov( x, eax );
cmp( eax, y );
jnl TestFails;
mov( z, eax );
cmp( eax, t );
jng TestFails;

```

<< Code for THEN clause of IF statement >>

TestFails:

注意上面的示例中确定 x 不小于 y 后代码是如何跳过更多测试的。当然，如果 x 小于 y ，程序还是必须测试 z 是否大于 t ；如果不是，程序跳过 then 子句。只有当满足所有条件时，程序才会执行 then 子句中的代码。

对于逻辑 or 操作, 技巧类似。如果第一个子表达式求值为真, 就没有必要测试第二个操作数。这时, 不管第二个操作数是什么, 整个表达式的值都为真。下面的示例说明了如何使用短路布尔求值实现逻辑 or:

```
// if( ch < 'A' || ch > 'Z' )
//     then stdout.put( "Not an uppercase char" );
// endif;

        cmp( ch, 'A' );
        jb ItsNotUC
        cmp( ch, 'Z' );
        jna ItWasUC;

        ItsNotUC:
            stdout.put("Not an uppercase char");

        ItWasUC:
```

因为逻辑与和逻辑或操作是可交换的, 所以根据情况可以先对左操作数或右操作数中任何一个求值²。本节最后一个示例考虑前一节中的完整的布尔表达式:

```
// if( (( x < y ) && ( z > t )) || ( a != b )) << Stmt1 >>;

        mov( a, eax );
        cmp( eax, b );
        jne DoStmt1;
        mov( x, eax );
        cmp( eax, y );
        jnl SkipStmt1;
        mov( z, eax );
        cmp( eax, t );
jng SkipStmt1;

        DoStmt1:
            << Code for Stmt1 goes here. >>

        SkipStmt1:
```

注意其中的代码如何选择首先计算 $a \neq b$, 然后再计算其余的子表达式。这是汇编程序员编写代码的一个常用技巧。

7.7.5 短路布尔求值与完整布尔求值

完整布尔求值在计算表达式时需要计算指令序列中的每条语句, 而短路布尔求值可能不需要计算布尔表达式对应的指令序列的所有语句。如上一节所述, 基于短路布尔求值的代码通常更短更快。所以, 将复杂布尔表达式转换成汇编语言时, 通常选择短路布尔求值实现。

然而, 有些情况下短路布尔求值可能会产生不正确的结果。如果一条表达式存在副作用(side

² 但是, 要注意有些表达式的值取决于最左边的子表达式, 只有先求出左边表达式的值, 才能确定右边的子表达式是否有效; 例如, C/C++ 中的一个常用测试: `if(x != NULL && x->y)...`

effect), 短路布尔求值会得到与完整布尔求值不同的结果。考虑下面的 C/C++ 示例:

```
if( ( x == y ) && ( ++z != 0 ) ) << Stmt >>;
```

使用完整布尔求值, 可能会得到下面的代码:

```
mov( x, eax );      // See if x == y.
cmp( eax, y );
sete( bl );
inc( z );           // ++z
cmp( z, 0 );        // See if incremented z is zero.
setne( bh );
and( bh, bl );      // Test x == y && ++z != 0.
jz SkipStmt;

<< Code for Stmt goes here. >>
```

SkipStmt:

使用短路布尔求值, 可能会得到下面的代码:

```
mov( x, eax );      // See if x == y.
cmp( eax, y );
jne SkipStmt;
inc( z );           // ++z
cmp( z, 0 );        // See if incremented z is 0.
je SkipStmt;

<< Code for Stmt goes here. >>
```

SkipStmt:

注意这两种转换的一个非常微妙但又很重要的差别: 如果 x 等于 y , 则第一个版本会在执行 Stmt 相关的代码之前先将 z 加 1, 然后与 0 比较; 另一方面, 如果 x 等于 y , 短路版本会跳过将 z 加 1 的代码。因此, 如果 x 等于 y , 这两个代码段中的 z 的行为会不一样。并不能说哪种实现是错误的, 这要视程序员的需要而定, 程序员可能在 x 等于 y 时需要 z 加 1, 也可能不需要 z 加 1。但是, 认识到这两种实现会产生不同的结果是非常重要的。当程序与 z 的行为相关时, 认识到这一点能帮助您选择适当的实现方式。

许多程序利用短路布尔求值的优点, 因为程序可能真的不需要对表达式的某个特定部分求值。下面的 C/C++ 代码段描述了一种最常见的需要短路布尔求值实现的情况:

```
if( Ptr != NULL && *Ptr == 'a' ) << Stmt >>;
```

如果在这条语句中发现 Ptr 是空指针(NULL), 表达式的值为假, 就没有必要对表达式的其他部分求值(这时, 使用短路布尔求值将不计算表达式的其他部分)。这条语句要正确执行, 必须使用短路布尔求值的语义。如果 C/C++ 使用完整布尔求值, 因为变量 Ptr 值为 NULL, 表达式的第二部分将设法解除对空指针的引用(这通常导致许多程序崩溃)。请看下面使用完整布尔求值翻译后的结果:

```
// Complete boolean evaluation:
```

```
mov( Ptr, eax );
test( eax, eax );      // Check to see if eax is 0 (NULL is 0).
setne( bl );
mov( [eax], al );      // Get *Ptr into al.
cmp( al, 'a' );
sete( bh );
and( bh, bl );
jz SkipStmt;

<< Code for Stmt goes here. >>
```

```
SkipStmt:
```

注意在上面的示例中，如果 `Ptr` 值为 `NULL(0)`，程序将试图通过指令 `mov([eax],al)` 访问存储器地址为 0 的数据。在大多数操作系统中，这导致一个存储器访问错误(通用保护错误)。现在请看使用短路布尔求值翻译后的代码：

```
// Short-circuit boolean evaluation
```

```
mov( Ptr, eax );      // See if Ptr contains NULL (0) and
test( eax, eax );      // immediately skip past Stmt if this
jz SkipStmt;          // is the case.

mov( [eax], al );      // If we get to this point, Ptr contains
cmp( al, 'a' );        // a non-NULL value, so see if it points
jne SkipStmt;          // at the character 'a'.

<< Code for Stmt goes here. >>
```

```
SkipStmt:
```

在这个示例中，您可以看到，解除对空指针引用的问题不再存在。当 `Ptr` 值为 `NULL` 时，代码跳过试图访问 `Ptr` 指针包含的存储器地址的语句。

7.7.6 汇编语言中 if 语句的高效实现

使用汇编语言对 if 语句进行高效编码，需要更深入的思考，而不仅仅是简单地选择使用短路布尔求值还是完整布尔求值。为了编写出执行速度更快的汇编语言代码，必须仔细分析实际情况，然后才能正确地生成代码。下面提供了一些建议，帮助您提高程序的性能。

1. 知道自己的数据

程序员经常犯的一个错误是，认为数据是随机的。实际上，数据很少是随机的。如果知道程序通常使用的数据的类型，就可以利用这点编写更高效的代码。请看下面的 C/C++ 语句：

```
if(( a == b ) && ( c < d )) ++i;
```

因为 C/C++ 使用短路布尔求值，上面的代码将测试 `a` 是否等于 `b`。如果等于，然后测试 `c` 是否小于 `d`；如果您期望大多数情况下 `a` 等于 `b`，而 `c` 不小于 `d`，这条语句将执行得比较慢。再看下

面给出的这条语句的 HLA 实现:

```

mov( a, eax );
cmp( eax, b );
jne DontIncI;
mov( c, eax );
cmp( eax, d );
jnl DontIncI;

    inc( i );

DontIncI:

```

从上面的代码中可以看到, 如果大多数情况 *a* 等于 *b* 而 *c* 不小于 *d*, 那么要想确定表达式的值是否为 *false*, 每次都要执行全部 6 条指令。现在考虑该条 C/C++ 语句的如下实现(利用了该知识点, 以及 *&&* 操作符的可交换性):

```

mov( c, eax );
cmp( eax, d );
jnl DontIncI;

mov( a, eax );
cmp( eax, b );
jnl DontIncI;

    inc( i );

DontIncI:

```

在该示例中, 代码首先检查 *c* 是否小于 *d*。如果在大多数情况下 *c* 小于 *d*, 那么该代码决定只执行 3 条指令(在通常情况下), 而不是像上一例那样执行全部 6 条指令, 然后跳到标号 *DontIncI* 包含的语句。在汇编语言中要比在高级语言中更容易看出这一点, 这也是汇编程序一般比对应的高级语言程序运行速度快的主要原因之一: 在汇编语言中更容易知道在哪里优化。当然, 这里的关键是理解数据的行为, 这样才能作出像上面这样的明智决定。

2. 表达式重组

即使数据是随机的(或者您无法根据输入值确定表达式中各项求值的顺序), 通过重新排列表达式中的项也会带来一些好处。有一些计算会比其他计算花费更多的时间: 例如, 除法(*div*)指令就比比比较(*cmp*)指令慢得多。因此, 下面的语句就可以重新组织, 让 *cmp* 首先执行:

```

if( (x % 10 == 0) && (x != y) ) ++x;

```

转换成汇编代码, 上面的 *if* 语句变成:

```

mov( x, eax );           // Compute X %10.
cdq();                   // Must sign extend eax -> edx:eax
imod( 10, edx:eax );      // Remember, remainder goes into edx.
test( edx, edx );         // See if edx is 0.
jnz SkipIf;

```

```

mov( x, eax );
cmp( eax, y );
je SkipIf;

    inc( x );

SkipIf:

```

其中, `imod` 指令非常耗时(通常比其他指令慢 50~100 倍)。除非余数为 0 的概率真的比 `x` 等于 `y` 的概率大 50~100 倍, 否则, 最好先进行比较操作, 然后再进行求余数的操作, 如下面的代码所示:

```

mov( x, eax );
cmp( eax, y );
je SkipIf;

mov( x, eax );           // Compute X %10.
cdq();                  // Must sign extend eax -> edx:eax
imod( 10, edx:eax );     // Remember, remainder goes into edx
test( edx, edx );       // See if edx is 0.
jnz SkipIf;

    inc( x );

SkipIf:

```

当然, 为了能够重新组织表达式, 必须确定代码没有使用短路布尔求值的语义(因为, 如果代码要求某个子表达式必须先于另一个子表达式计算, `&&` 和 `||` 操作符是不可交换的)。

3. 非结构化代码

虽然结构化编程技术有很多好处, 但它仍然有一些不足。尤其是, 比起非结构化代码, 结构化代码的效率有时很低。在很多情况下, 这种现象是可以容忍的, 因为非结构化代码的可读性差, 难以维护; 很多时候牺牲一部分的性能使用易维护的结构化代码是可以接受的。但是, 有些特殊情况, 需要获得尽可能高的性能, 这时, 就需要牺牲代码的可读性来获得更高的性能。

一种典型的方法是代码移动: 将程序中很少执行的代码移到经常执行的代码外面。例如, 考虑下面的伪 C/C++ 语句:

```

if( See_If_an_Error_Has_Occurred )
{
    << Statements to execute if no error >>
}
else
{
    << Error handling statements >>
}

```

在代码正常执行时, 错误情况一般很少出现。因此, 可以很自然地想到在上面的 `if` 语句中,

then 子句比 else 子句更频繁地执行。于是可以将上面的代码转换成下面的汇编语言：

```

cmp( See_If_an_Error_Has_Ocurred, true );
je HandleTheError;

    << Statements to execute if no error >>
    jmp EndOfIf;

HandleTheError:
    << Error handling statements >>
EndOfIf:

```

注意，如果表达式为假，上面的代码执行完正常语句后，就跳过错误处理语句。在程序中，将控制从一点转移到另一点的指令(例如 jmp 指令)通常执行很慢；执行一串顺序指令会比使用跳转指令跳过这些指令还快得多。但是遗憾的是，上面的代码并不允许这样。解决这个问题的一個办法是把 else 子句移到程序中的其他位置。例如，可以将代码按如下方式进行重写：

```

cmp( See_If_an_Error_Has_Ocurred, true );
je HandleTheError;

    << Statements to execute if no error >>

EndOfIf:

```

在程序中相应的位置(通常在 jmp 指令之后)，需要插入下面的代码：

```

HandleTheError:
    << Error handling statements >>
    jmp EndOfIf;

```

注意，程序并没有变短，从原始代码序列中移出的 jmp 指令只是被放到了 else 子句之后。但是，因为 else 子句很少执行，将 jmp 指令从 then 子句(该子句经常执行)中移出并放到 else 子句中，能够获得明显的性能提升，因为此时，then 子句只需要执行一串顺序指令(没有分支)。在许多时间关键的代码段中，使用这种技术是非常有效的。

编写非结构化代码与编写没有结构的代码是不同的。没有结构的代码是一开始就以没有结构的方式编写，通常很难阅读和维护，并包含很多漏洞；但非结构化代码首先是从结构化代码开始，然后有意识地消除结构来获得性能的提升。通常，在消除代码结构之前，已经测试过结构化的原始代码。因此，非结构化代码比没有结构的代码更容易处理。

4. 尽量选择计算而不是分支

在许多 80x86 系列的处理器上，分支指令比起其他的指令非常耗时。因此，有时执行一串较多的顺序指令序列比执行较少的包含分支的指令序列要快得多。例如，考虑简单的赋值语句 `eax=abs(eax);`。并不存在计算整数绝对值的 80x86 指令。要解决这个问题，显然可以使用下面的指令序列：

```

test( eax, eax );
jns ItsPositive;

```

```
neg( eax );
```

```
ItsPositive:
```

从这个示例中，很容易看出，它使用了一个条件跳转指令来跳过 `neg` 指令(如果 `EAX` 为负数，则变成正值放在 `EAX` 中)。现在请看下面的指令序列，它也完成同样的功能：

```
// Set edx to $FFFF_FFFF if eax is negative, $0000_0000 if eax is
// 0 or positive:

cdq();

// If eax was negative, the following code inverts all the bits in eax,
// otherwise it has no effect on eax.

xor( edx, eax );

// If eax was negative, the following code adds 1 to eax, otherwise
// it doesn't modify eax's value.

and( 1, edx );      // edx = 0 or 1 (1 if eax was negative).
add( edx, eax );
```

对于上面的代码来说，在进入该指令序列前，如果 `EAX` 为负数，则将 `EAX` 逐位取反，然后 `EAX` 加 1(也就是说，取 `EAX` 中值的二进制补码)；如果 `EAX` 为 0 或正数，则不改变 `EAX` 的值。

注意，该指令序列使用了 4 条指令，前一个示例只使用了 3 条指令；但是，因为该指令序列中没有控制转移指令，在许多 80x86 系列的 CPU 上，它会执行得更快。

7.7.7 switch/case 语句

HLA 的 `switch` 语句采用如下形式：

```
switch( reg32 )
  case( const1 )
    << Stmt1: code to execute if reg32 equals const1 >>

  case( const2 )
    << Stmt2: code to execute if reg32 equals const2 >>
    .
    .
    .
  case( constn )
    << Stmtn: code to execute if reg32 equals constn >>

  default      // Note that the default section is optional.
    << Stmt_default: code to execute if reg32
      does not equal any of the case values >>

endswitch;
```

当执行这条语句时，它将寄存器的值与常量 `const1~constn` 比较。如果发现与某个常量匹配，则执行相应的语句。HLA 对 `switch` 语句有一些限制：第一，HLA 的 `switch` 语句只允许 32 位的寄

寄存器作为 switch 表达式；第二，所有出现在 case 子句中的常量必须唯一。稍后将说明为什么有这些限制。

许多介绍性的编程教材只是将 switch/case 语句解释成一串 if..then..elseif..else..endif 语句，认为下面两段 HLA 代码是等价的：

```
switch( eax )
  case(0) stdout.put("i=0");
  case(1) stdout.put("i=1");
  case(2) stdout.put("i=2");
endswitch;

if( eax = 0 )then
  stdout.put("i=0");
elseif( eax = 1 )then
  stdout.put("i=1");
elseif( eax = 2 )then
  stdout.put("i=2");
endif;
```

虽然在语义上这两段代码是相同的，但它们的实现却不同。if..then..elseif..else..endif 链对序列中每个条件语句都进行了一次比较操作，而 switch 语句只进行一次计算，然后使用一个间接跳转将控制转移给某个 case 语句。考虑上面的两个示例，它们可以用下面的汇编语言代码实现：

```
// if..then..else form:

mov( i, eax );
test( eax, eax );    // Check for 0.
jnz Not0;
    stdout.put( "i=0" );

jmp EndCase;

Not0:
cmp( eax, 1 );
jne Not1;
    stdout.put( "i=1" );
    jmp EndCase;

Not1:
cmp( eax, 2 );
jne EndCase;
    stdout.put( "i=2" );

EndCase:

// Indirect Jump Version

readonly
  JmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
```



```

mov( i, eax );
jmp( JumpTbl[eax*4] );

    Stmt0:
        stdout.put( "i=0" );
        jmp EndCase;

    Stmt1:
        stdout.put( "i=1" );
        jmp EndCase;

    Stmt2:
        stdout.put( "i=2" );

EndCase:

```

if..then..elseif..else..endif 版本的实现非常明显，不需要太多解释。然而，间接跳转版本对您来说可能显得有些神秘。下面将说明 switch 语句的实现是如何工作的。

请回忆一下 jmp 指令的 3 种通用形式：标准的无条件跳转(jmp)指令，如上面示例中的 jmp EndCase;，将控制直接转移到作为 jmp 指令操作数的语句标号指定的位置处；第二种形式的 jmp 指令(例如 jmp(reg32);)，将控制转移到 32 位寄存器值指定的地址处；第三种形式的 jmp 指令，如上面的示例使用的，将控制转移到一个双字存储器指定的位置处。上面这个示例清楚地说明，存储单元可以使用任何寻址方式。没有限制只能使用偏移寻址方式。现在让我们考虑上面 switch 语句的第二种实现是如何工作的。

首先，switch 语句要求创建一个指针数组，每个指针包含代码中一个语句标号的地址(这些标号必须与 switch 语句中每个 case 要执行的指令序列相关联)。在上面的示例中，JumpTbl 数组完成这部分工作。注意代码使用语句标号 Stmt0、Stmt1 和 Stmt2 的地址初始化 JumpTbl。程序将这个数组放在只读段，因为程序执行时永远不会修改这些值。

警告：

像上面的示例一样，无论什么时候使用一组语句标号的地址初始化一个数组，语句标号必须与声明该数组的段(示例中为只读段)在同一个过程中³。

在这个指令序列的执行过程中，程序将 i 的值装载到 EAX 寄存器中。然后程序使用这个值作为 JumpTbl 数组的索引，再将控制转移到指定位置的四字节地址处。例如，如果 EAX 值为 0，jmp(JumpTbl[eax*4]);指令将去取地址 JumpTbl+0(eax*4=0)处的双字。因为表中第一个双字包含 Stmt0 的地址，jmp 指令将控制转移给紧跟在标号 Stmt0 后的第一条指令。类似地，如果 i 为 1(因此，EAX 的值也为 1)，间接 jmp 指令去取表中偏移值为 4 处的双字，然后将控制转移给标号 Stmt1 后的第一条指令(因为 Stmt1 的地址出现在表中偏移值为 4 的位置)。最后，如果 i/EAX 值为 2，该代码段将控制转移给标号 Stmt2 后的语句，因为它的地址处在 JumpTbl 表中偏移值为 8 的位置。

现在，可以非常明显地看出：和 if/elseif 形式相比，使用的(连续的)case 越多，跳转表实现的效率相对也越高(就空间和速度而言)。除了一些简单的 case，switch 语句通常比 if/elseif 执行要快很多。只要 case 值是连续的，switch 语句的代码量也小得多。

当包含的 case 标号不连续，或者不能确定 switch 的值是否超出范围时，会发生什么情况呢？

³ 如果 switch 语句出现在主程序中，则必须在主程序的声明段声明该数组。

对于 HLA 的 switch 语句,控制会被转移给 endswitch 子句后的第一条语句(或者如果存在 default 子句,则控制会转移到 default 子句)。但是上面的示例并不会这样。如果变量 i 值不为 0、1 或 2,代码的执行结果将不能确定。例如,如果 i 的值为 5, jmp 指令将去取表 JmpTbl 中偏移量为 20(5*4) 的双字,然后将控制转移到该地址。但是, JmpTbl 没有包含 6 个表项,所以程序将去取 JmpTbl 后的第三个双字并作为目标地址。因为该地址没有定义,通常会导致程序崩溃,或者将控制转移到意想不到的地方。

解决的办法是在间接 jmp 指令之前放置几条指令判断 switch 选择值是否在合法的范围之内。在上面的示例中,在执行 jmp 指令前,很可能需要判断 i 的值是否在 0~2 内。如果 i 的值在这个范围之外,程序简单地跳转到 endcase 标号处(即跳转到 endswitch 子句后的第一条语句)。下面的代码是修改后的版本:

```
readonly
    JmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];

    mov( i, eax );
    cmp( eax, 2 );           // Verify that i is in the range
    ja EndCase;             // 0..2 before the indirect jmp.
    jmp( JmpTbl[ eax*4 ] );

    Stmt0:
        stdout.put( "i=0" );
        jmp EndCase;

    Stmt1:
        stdout.put( "i=1" );
        jmp EndCase;

    Stmt2:
        stdout.put( "i=2" );

    EndCase:
```

虽然上面的代码解决了所选择的值在 0~2 范围外的问题,但还是受到下面两个严重限制:

- case 值必须从 0 开始。也就是说,在这个示例中,最小的 case 常量必须为 0。
- case 值必须是连续的。

解决第一个问题很容易,可以分两步解决。第一步,在判断 case 值是否合法之前,将 case 值与下限和上限比较。例如:

```
// SWITCH statement specifying cases 5,6,and 7:
// WARNING: This code does *NOT* work. Keep reading to find out why.

    mov( i, eax );
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );           // Verify that i is in the range
    ja EndCase;             // 5..7 before the indirect jmp.
```

```

    jmp( JmpTbl[ eax*4 ] );

    Stmt5:
        stdout.put( "i=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "i=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "i=7" );

    EndCase:

```

可以看到，这段代码增加了两条额外指令：`cmp` 和 `jb`，用来测试选择值，确保它处于 5~7 内。如果没有，控制直接转到 `EndCase` 标号处；否则，通过间接 `jmp` 指令转移到相应的分支。但是，如注释中提到的，这段代码是有错误的。考虑如果变量 `i` 值为 5 的情况：代码将确定 5 处于 5~7 的范围内，然后跳转到相应的地址处。和以前一样，这将导致从表边界外取回 4 字节数据，从而不会将控制转移到定义的位置。一种的解决方法是，在执行 `jmp` 指令之前，将 `EAX` 的值减去最小的 `case` 选择值。例如：

```

// SWITCH statement specifying cases 5,6,and 7:
// WARNING: There is a better way to do this. Keep reading.

readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];

    .
    .
    .

    mov( i, eax );
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );           // Verify that i is in the range
    ja EndCase;             // 5..7 before the indirect jmp.
    sub( 5, eax );          // 5->0,6->1,7->2.
    jmp( JmpTbl[ eax*4 ] );

    Stmt5:
        stdout.put( "i=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "i=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "i=7" );

    EndCase:

```

通过将 `EAX` 的值减去 5，这段代码使 `EAX` 的值在 `jmp` 指令前变为 0、1 或 2。从而，`case` 选择值 5 对应跳转到 `Stmt5` 处，`case` 选择值 6 对应跳转到 `Stmt6` 处，`case` 选择值 7 对应跳转到 `Stmt7` 处。

可以用一个小小的技巧改进上面的代码。将减法操作放到 `jmp` 指令的地址表达式中, 可以消除 `sub` 指令。请看下面的代码:

```
// SWITCH statement specifying cases 5,6,and 7:
readonly
    JmpTbl:dword[3] := [&Stmt5, &Stmt6, &Stmt7];

    mov( i, eax );
    cmp( eax, 5 );
    jb EndCase;
    cmp( eax, 7 );           // Verify that i is in the range
    ja EndCase;             // 5..7 before the indirect jmp,
    jmp( JmpTbl[ eax*4 - 5*sizeof(dword) ] );

    Stmt5:
        stdout.put( "i=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "i=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "i=7" );

    EndCase:
```

HLA 的 `switch` 语句提供了一个 `default` 子句, 如果 `case` 选择值不与任何一个 `case` 值匹配, 将执行该 `default` 子句。例如:

```
switch( ebx )

    case( 5 ) stdout.put( "ebx=5" );
    case( 6 ) stdout.put( "ebx=6" );
    case( 7 ) stdout.put( "ebx=7" );
    default
        stdout.put( "ebx does not equal 5,6,or 7" );

endswitch;
```

在纯汇编语言中, 实现等价的 `default` 子句是非常容易的。只需要在代码开头的 `jb` 和 `ja` 指令中使用一个不同的目标标号。下面的示例实现了与上面类似的 HLA `switch` 语句:

```
// SWITCH statement specifying cases 5,6,and 7 with a DEFAULT clause:
readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
```

```

mov( i, eax );
cmp( eax, 5 );
jb DefaultCase;
cmp( eax, 7 );           // Verify that i is in the range
ja DefaultCase;         // 5..7 before the indirect jmp.
jmp( JmpTbl[ eax*4 - 5*@size(dword)] );

```

```

Stmt5:
    stdout.put( "i=5" );
    jmp EndCase;

```

```

Stmt6:
    stdout.put( "i=6" );
    jmp EndCase;

```

```

Stmt7:
    stdout.put( "i=7" );
    jmp EndCase;

```

```

DefaultCase:
    stdout.put( "i does not equal 5,6,or 7" );

```

```

EndCase:

```

前面提到的第二个限制，**case** 值必须是连续的，通过在跳转表中插入额外的表项很容易处理。请看下面的 HLA **switch** 语句：

```

switch( ebx )

    case( 1 ) stdout.put( "ebx = 1" );
    case( 2 ) stdout.put( "ebx = 2" );
    case( 4 ) stdout.put( "ebx = 4" );
    case( 8 ) stdout.put( "ebx = 8" );
    default=
        stdout.put( "ebx is not 1,2,4,or 8" );

endswitch;

```

其中，最小的开关值为1，最大的为8。因此，在间接跳转指令之前的代码需要将 **EBX** 的值与1和8比较。即使值在1和8之间，**EBX** 仍然可能含有一个非法的选择值。但是 **jmp** 指令索引的双字表——**case** 选择表——必须包含8个双字表项；为了处理1~8中不是合法 **case** 选择值的值，只需要简单地将 **default** 子句的语句标号(或者，如果没有 **default** 子句，则使用指示 **endswitch** 后的第一条指令的标号)放到这些没有对应 **case** 子句的表项中。下面的代码说明了如何使用这种技术：

```

readonly
JmpTbl2: dword :=
[
    &Case1, &Case2, &dfltCase, &Case4,
    &dfltCase, &dfltCase, &dfltCase, &Case8
];

```



```

cmp( ebx, 1 );
jb dfltCase;
cmp( ebx, 8 );
ja dfltCase;
jmp( JmpTbl2[ ebx*4 - 1*@size(dword)] );

Case1:
    stdout.put( "ebx = 1" );
    jmp EndOfSwitch;

Case2:
    stdout.put( "ebx = 2" );
    jmp EndOfSwitch;

Case4:
    stdout.put( "ebx = 4" );
    jmp EndOfSwitch;

Case8:
    stdout.put( "ebx = 8" );
    jmp EndOfSwitch;

dfltCase:
    stdout.put( "ebx is not 1,2,4,or 8" );

EndOfSwitch:

```

switch 语句的这种实现还有一个问题：如果 case 值包含跨越范围很大的不连续的值，那么跳转表将变得非常大。下面的 switch 语句将产生非常庞大的代码文件：

```

switch( ebx )

    case( 1 )      << Stmt1 >>;
    case( 100 )    << Stmt2 >>;
    case( 1_000 )  << Stmt3 >>;
    case( 10_000 ) << Stmt4 >>;
    default << Stmt5 >>;

endswitch;

```

在这种情况下，如果使用一串 if 语句实现 switch 语句，程序会比用间接跳转语句实现小得多。但是，必须牢记一点：跳转表的大小并不会影响程序的执行速度。不管跳转表包含两个表项还是 2000 个表项，switch 语句执行分支的时间是一样的。而如果使用 if 语句实现，程序执行的时间和 case 语句中 case 标号的数目呈线性增长关系。

与使用 Pascal 或 C/C++ 等高级语言相比，使用汇编语言最大的一个好处是可以自由选择像 switch 这样的语句的实现方法。根据情况，可以用一串 if..then..elseif 语句实现 switch 语句，也可以作为跳转表实现，还可以混合使用这两种方法。例如：

```

switch( eax )

    case( 0 ) << Stmt0 >>;
    case( 1 ) << Stmt1 >>;

```

```

    case( 2 )    << Stmt2 >>;
    case( 100 ) << Stmt3 >>;
    default <<Stmt4>>;

endswitch;

```

上面的代码可以变成:

```

    cmp( eax, 100 );
    je DoStmt3;
    cmp( eax, 2 );
    ja TheDefaultCase;
    jmp( JmpTbl[ eax*4 ] );
    ...

```

当然,也可以使用下面的 HLA 高级控制结构:

```

if( ebx = 100 )then
    << Stmt3 >>;
else
    switch( eax )
        case(0)<< Stmt0 >>;
        case(1)<< Stmt1 >>;
        case(2)<< Stmt2 >>;
        Otherwise<< Stmt4 >>;
    endswitch;
endif;

```

但是这会破坏程序的可读性。在另一方面,上面的汇编语言代码中用来与 100 比较的额外代码并不会降低代码的可读性(也许因为它本身已经够难读了)。因此,许多程序员通过增加额外的代码使他们的程序效率更高。

C/C++的 switch 语句与 HLA 的 switch 语句非常类似。在语义上只有一个主要不同:在 C/C++中,程序员必须显式地在每个 case 语句中加入 break 语句,用来将控制转移到 switch 之后的第一条语句。这个 break 语句对应汇编语言中每个 case 指令序列结尾处的 jmp 指令。如果没有相应的 break 语句,C/C++直接将控制转移到下一个 case 语句。这和去掉 case 指令序列结尾的 jmp 指令是一样的。请看下面的示例:

```

switch (i)
{
    case 0:    << Stmt1 >>;
    case 1:    << Stmt2 >>;
    case 2:    << Stmt3 >>;
                break;
    case 3:    << Stmt4 >>;
                break;
    default:   << Stmt5 >>;
}

```

它可以翻译成下面的 80x86 代码:

```

readonly
    JmpTbl: dword[4] := [ &case0,&case1,&case2,&case3 ];
    .
    .
    .
    mov( i, ebx );
    cmp( ebx, 3 );
    ja DefaultCase;
    jmp( JmpTbl[ ebx*4-1] );

-    case0:
        Stmt1;

    case1:
        Stmt2;

    case2:
        Stmt3;
        jmp EndCase;          // Emitted for the break stmt.

    case3:
        Stmt4;
        jmp EndCase;          // Emitted for the break stmt.

    DefaultCase:
        Stmt5;

    EndCase:-

```

7.8 状态机和间接跳转

在汇编语言中, 状态机(state machine)是另外一个常见的控制结构。状态机使用状态变量(state variable)来控制程序流。FORTRAN 编程语言利用 goto 语句提供这种能力。一些版本的 C 语言(例如, 免费软件基金会的 GNU GCC)也提供类似的特性。在汇编语言中, 通过间接跳转可以实现状态机。

什么是状态机? 简单地说, 它是一段代码, 这段代码通过进入和离开特定的状态来记录其执行历史信息(状态)。考虑到本章要介绍的内容, 我们假定状态机就是一段代码, 它记录了它的执行历史信息, 并基于历史信息执行相应的代码段。

从某种意义上说, 所有的程序都是状态机, CPU 寄存器和存储器中的值组成了该状态机的状态。这里, 我们只使用其中部分值作为状态机状态。实际上, 对于大多数情况, 只需要一个变量(或者, EIP 寄存器的值)来表示当前状态。

我们来考虑一个实际的示例: 假设需要这样一个过程, 在第一次调用它时, 执行第一种操作; 在第二次调用时, 执行第二种操作; 在第三次调用时, 执行第三种操作; 在第四次调用时, 执行第四种操作。在第四次调用以后, 重复上述过程。例如, 在第一次调用过程时, EAX 和 EBX 的值相加, 第二次时相减, 第三次时相乘, 然后第四次时相除。可以用下面的代码实现这个过程:

```

procedure StateMachine;
static
    State:byte := 0;
Begin StateMachine;

    cmp( State, 0 );
    jne TryState1;

    // State 0: Add ebx to eax and switch to State 1:

    add( ebx, eax );
    inc( State );
    exit StateMachine;

TryState1:
    cmp( State, 1 );
    jne TryState2;

    // State 1: Subtract ebx from eax and switch to State 2:

    sub( ebx, eax );
    inc( State ); // State 1 becomes State 2.
    exit StateMachine;

TryState2:
    cmp( State, 2 );
    jne MustBeState3;

    // If this is State 2, multiply eax by ebx and switch to State 3:

    intmul( ebx, eax );
    inc( State ); // State 2 becomes State 3.
    exit StateMachine;

    // If it isn't one of the above states, we must be in state 3
    // So divide eax by ebx and switch back to State 0.

MustBeState3:
    push( edx ); // Preserve this 'cause it gets whacked by div.
    xor( edx, edx ); // Zero extend eax into edx.
    div( ebx, edx:eax );
    pop( edx ); // Restore edx's value preserved above.
    mov( 0, State ); // Reset the state back to 0.
end StateMachine;

```

从技术上说，这个过程并不是状态机，而是变量 State 和 cmp/jne 指令组成了状态机。

这段代码并没有什么非常特别的地方。它只不过是通过 if..then..elseif 结构实现的 switch 语句。唯一值得注意的是，这个过程记录了它被调用的次数⁴，调用次数不同，程序的行为也相应不同。

虽然用这段代码实现状态机是正确的，但效率却不是很高。聪明的读者一定看出这段代码可以使用 switch 语句实现，而不是 if..then..elseif 实现，这样程序也会执行得稍微快一些。但是，还

⁴ 实际上，它记录的是被调用的次数与 4 的余数。

有更好的办法。

使用汇编语言实现状态机更常用的办法是使用间接跳转指令。这时，状态变量包含的不是像 0、1、2、3 这样的值，而是在进入过程时要执行的代码的地址。通过简单地跳转到该地址，状态机可以节省许多上面示例需要的测试操作，而可以执行相应的代码段。请看下面使用间接跳转实现该状态机的示例：

```

procedure StateMachine;
static
    State:dword := &State0;
begin StateMachine;
    jmp( State );

    // State 0: Add ebx to eax and switch to State 1:
State0:
    add( ebx, eax );
    mov( &State1, State );
    exit StateMachine;

State1:
    // State 1: Subtract ebx from eax and switch to State 2:
    sub( ebx, eax );
    mov( &State2, State ); // State 1 becomes State 2.
    exit StateMachine;

State2:
    // If this is State 2, multiply eax by ebx and switch to State 3:
    intmul( ebx, eax );
    mov( &State3, State ); // State 2 becomes State 3.
    exit StateMachine;

    // State 3: Divide eax by ebx and switch back to State 0.
State3:
    push( edx ); // Preserve this 'cause it gets whacked by div.
    xor( edx, edx ); // Zero extend eax into edx.
    div( ebx, edx:eax );
    pop( edx ); // Restore edx's value preserved above.
    mov( &State0, State ); // Reset the state back to 0.

end StateMachine;

```

在过程 StateMachine 开始处的 jmp 指令将控制转移到 State 变量指向的地址。当第一次调用 StateMachine 时，State 指向 State0 标号。此后，每个子代码段设置 State 变量，使它指向正确的后续代码地址。

7.9 “意大利面条式”代码

汇编语言的一个主要问题是需要好几条汇编语句才能实现一条高级语言的语句。很多时候，汇编语言程序员发现通过将程序跳转到一些程序结构的中间，可以节约几个字节或几个周期。在确定这点(并对代码作了相应的修改)后，代码最后会包含一串在代码的一些结构中跳入跳出的指令。如果在每条跳转指令和相应的目标地址间连上一条线，最后代码看上去会像一碗堆起的意大利面条，通常称这样的代码为“意大利面条式”代码。

“意大利面条式”代码存在一个主要缺点：可读性差，很难确定它到底完成什么工作。许多程序开始还是结构化的，但性能优化后都变成了“意大利面条式”代码；但是，实际上“意大利面条式”代码很少是高效的，因为太难确定它到底完成什么功能，从而也很难使用更好的算法来改进。因此，最后导致“意大利面条式”代码比结构化代码的效率更低。

“意大利面条式”代码的确可能会提高程序的性能，但这只是在对程序做完所有其他可能的改进还不能达到要求时不得已而采取的办法。开始编写程序时，要尽量在程序中使用直接的 if 和 switch 语句，当程序正常工作并易于理解后，才开始合并代码段(使用 jmp 指令)。当然，除非真的值得，否则不要消除程序的结构。

在结构化编程界有一条著名的谚语：“除了 goto 语句，指针是编程语言中最危险的元素。”类似的说法有：“指针损害数据结构，goto 损害控制结构。”换句话说，要避免过量使用指针。如果指针和 goto 语句的结构性较差，那么间接跳转就是最差的一种结构，因为它同时包含了指针和 goto！不过话说回来，确实不要随意使用间接跳转指令。这会使程序变得难以阅读。毕竟，(理论上)间接跳转指令可以将控制转移到程序中的任何标号处；可以想象，如果不知道指针包含的内容，又遇到使用该指针的间接跳转指令，要理清程序流程是多么困难的一件事。因此，在使用间接跳转指令时要格外谨慎。

7.10 循环

循环是最后一个组成程序的基本控制结构(顺序、判定和循环)。和许多其他汇编语言结构类似，您会发现使用循环时，会在许多没有想到的地方使用了循环。许多高级语言将一些循环结构隐藏起来。例如，考虑 BASIC 语句“if A\$ = B\$ then 100”。这个 if 语句比较两个字符串，如果相等，则跳转到 100 号语句。如果用汇编语言实现，需要编写一个循环逐个字母地比较 A\$ 和 B\$，只有所有的字母都匹配，程序才跳转到 100 号语句。在 BASIC 程序中，看不到这样的循环。在汇编语言中，这个简单的 if 语句需要使用一个循环来比较字符串中的每个字母⁵。通过这个小示例，说明了循环可能出现在任何地方。

程序循环包含三个部分：可选的初始化部分、可选的循环结束判断部分以及循环体。它们的排列顺序很大程度上影响循环执行的方式。有三种经常出现的排列方式。因为这三种经常出现，在高级语言中分别给了它们不同的名称：while 循环、repeat..until 循环(在 C/C++ 中为 do..while)和无限循环(例如 HLA 中的 forever..endfor)。

⁵ 当然，可以使用 HLA 标准库提供的 str.cmp 例程来比较字符串，从而在汇编程序中也有效地隐藏循环。

7.10.1 while 循环

最通用的循环是 while 循环。在 HLA 中，它的形式如下：

```
while( expression ) do statements endwhile;
```

关于 while 循环有两点非常重要：第一，循环结束的判断出现在循环的开始；第二，由于循环结束判断的位置关系，循环体可能永远不会执行。如果布尔表达式的值总为假，那么循环体永远不会执行。

请看下面的 HLA while 循环：

```
mov( 0,i );
while( i < 100 )do

    inc( i );

endwhile;
```

其中，“mov(0,i);”指令是循环的初始化代码。i 是循环控制变量，因为它控制循环体的执行。“i<100”是循环结束条件。也就是说，如果 i 小于 100，循环就不会结束。循环体只有一条指令“inc(i)”，它是每次循环迭代时需要执行的代码。

注意 HLA while 循环可以用 if 和 jmp 语句简单地实现。例如，可以用下面的代码替换上面的 HLA while 循环：

```
mov( 0,i );
WhileLp:
if( i < 100 )then

    inc( i );
    jmp WhileLp;

endif;
```

更一般地，可以用下面的代码构造任意 while 循环：

```
<< Optional initialization code >>

UniqueLabel:
if( not_termination_condition )then

    << loop-body >>
    jmp UniqueLabel;

endif;
```

因此，可以使用本章前面介绍的技巧，先将 if 语句转换成汇编语言，然后结合一条 jmp 指令来构造 while 循环。例如，将前面的示例翻译成如下的纯 80x86 汇编代码⁶：

⁶ 注意，大多数 while 语句经 HLA 转换后的 80x86 代码与本节介绍的不大一样。在 7.11 节介绍如何编写高效的循环代码时会说明为什么不一样。

```

mov( 0,i );
WhileLp:
    cmp( i,100 );
    jnl WhileDone;
    inc( i );
    jmp WhileLp;

WhileDone:

```

7.10.2 repeat..until 循环

repeat..until(do..while)循环是在循环的结尾而不是在循环开始判断循环结束条件。在 HLA 高级语法中，**repeat..until** 循环的形式如下：

```

<< Optional initialization code >>
repeat

    << loop body >>

until( termination_condition );

```

上面的指令序列执行完初始化代码和循环体之后，再判断结束条件决定是否再次执行循环。如果布尔表达式的值为假，再次执行循环；否则，循环结束。关于 **repeat..until** 循环，有两点要注意：结束条件判断位于循环尾部；由于这个原因，循环体至少执行一次。

和 **while** 循环类似，可以使用 **if** 语句和 **jmp** 语句实现一个 **repeat..until** 循环。例如下面的代码：

```

<< Initialization code >>
SomeUniqueLabel:

    << loop body >>

if( not_the_termination_condition ) then jmp SomeUniqueLabel; endif;

```

基于前面小节介绍的内容，您应该很容易把 **repeat..until** 循环转换成汇编语言。下面是一个简单示例：

```

repeat

    stdout.put( "Enter a number greater than 100:" );
    stdin.get( i );

until( i > 100 );

```

// This translates to the following **if/jmp** code:

```

RepeatLabel:
    stdout.put( "Enter a number greater than 100:" );
    stdin.get( i );

    if( i <= 100 )then jmp RepeatLbl; endif;

```

// It also translates into the following "pure" assembly code:

```
RepeatLabel:

    stdout.put( "Enter a number greater than 100:" );
    stdin.get( i );

    cmp( i, 100 );
    jng RepeatLabel;
```

7.10.3 forever..endfor 循环

判断 **while** 循环结束的位置在循环开始, 判断 **repeat..until** 的循环结束在循环结尾, 因此循环结束判断唯一剩下的位置就是循环中间了。HLA 的 **forever..endfor** 循环, 通过结合 **break** 和 **breakif** 语句, 采取了这种方式。**forever..endfor** 循环的形式如下:

```
forever

    << Loop body >>

endfor;
```

注意, 其中并没有显式的结束条件判断。除非另外提供结束条件, 否则 **forever..endfor** 构成了一个无限循环。通常通过 **breakif** 语句来结束循环。请看下面的 HLA 代码, 它使用了 **forever..endfor** 结构:

```
forever

    stdin.get( character );
    breakif( character = '.' );
    stdout.put( character );

endfor;
```

将 **forever** 循环转换成汇编语言非常容易。所需要的只是一个标号和一条 **jmp** 指令。在这个示例中, **breakif** 语句实际上不过是一条 **if** 指令和一条 **jmp** 指令。转换后的纯汇编语言代码如下:

```
foreverLabel:

    stdin.get( character );
    cmp( character, '.' );
    je ForIsDone;
    stdout.put( character );
    jmp foreverLabel;

ForIsDone:
```

7.10.4 for 循环

for 循环是一种特殊形式的 **while** 循环, 它重复执行循环体直至达到指定的次数。在 HLA 中, **for** 循环的形式如下:

```
for( Initialization_Stmt; Termination_Expression; inc_Stmt ) do

    << statements >>
```

```
endfor;
```

它和下面的代码完全等价:

```
Initialization_Smt;
while( Termination_Expression ) do

    << statements >>

    inc_Smt;

endwhile;
```

一般情况下, for 循环用来处理数组和其他顺序访问的对象。通常首先使用初始化语句初始化循环控制变量, 然后利用循环控制变量作为索引访问数组元素(或其他数据类型)。例如:

```
for( mov( 0, esi ); esi < 7; inc( esi ) ) do

    stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

endfor;
```

为了把上面的代码转换成纯汇编语言, 首先将 for 循环转换成等价的 while 循环, 如下所示:

```
mov( 0, esi );
while( esi < 7 ) do

    stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

    inc( esi );

endwhile;
```

然后, 使用“while 循环”一节介绍的技术, 将上面的代码转换成下面的纯汇编语言:

```
mov( 0, esi );
WhileLp:
cmp( esi, 7 );
jnl EndWhileLp;

    stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

    inc( esi );
    jmp WhileLp;

EndWhileLp:
```

7.10.5 break 和 continue 语句

HLA 的 break 和 continue 语句都被翻译成一条 jmp 指令。break 指令从包含它的循环中马上退出, continue 指令直接重新执行包含它的循环。

将 break 语句转换成汇编语言非常容易。只需要产生一条 jmp 指令, 将控制转移给 endxxxx(或 until)子句后的第一条语句, 退出循环。可以首先在 endxxxx 子句后放置一个标号, 然后跳转到该标号。下面的代码描述了针对不同循环的各种实现方式:


```
// Breaking out of a FOREVER loop:

forever
    << stmts >>
        // break;
        jmp BreakFromForever;
    << stmts >>
endfor;
BreakFromForever:

// Breaking out of a FOR loop:

for( initStmt; expr; incStmt ) do
    << stmts >>
        // break;
        jmp BrkFromFor;
    << stmts >>
endfor;
BrkFromFor:

// Breaking out of a WHILE loop:

while( expr )do
    << stmts >>
        // break;
        jmp BrkFromWhile;
    << stmts >>
endwhile;
BrkFromWhile:

// Breaking out of a REPEAT..UNTIL loop:-

repeat
    << stmts >>
        // break;
        jmp BrkFromRpt;
    << stmts >>
until( expr );
BrkFromRpt:
```

实现 continue 语句比实现 break 语句要稍微困难一些。continue 语句的实现也只包含一条 jmp 指令,但是对不同的循环,目标标号并不在同一个地方。图 7-2、图 7-3、图 7-4 和图 7-5 分别说明了对于每种 HLA 循环,continue 语句控制转移的目标位置。

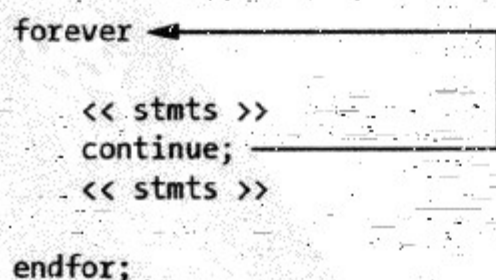


图 7-2 forever 循环的 continue 目标

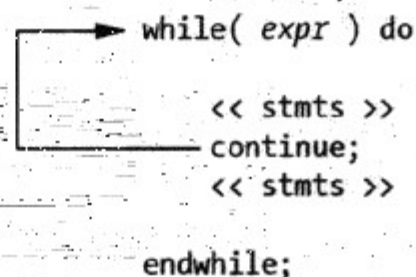


图 7-3 while 循环的 continue 目标

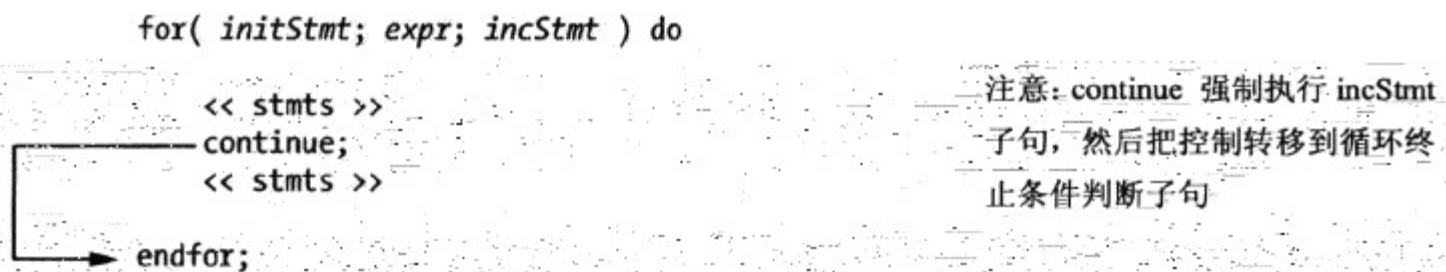


图 7-4 for 循环的 continue 目标

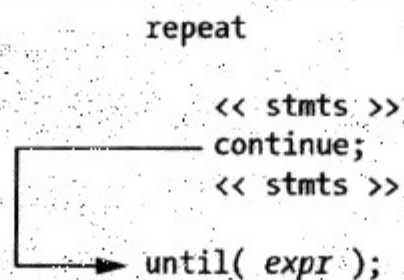


图 7-5 repeat..until 循环的 continue 目标

下面的代码段说明了如何将各种循环的 continue 语句转换成合适的 jmp 指令。

forever..continue..endfor

```
// Conversion of forever loop with continue
// to pure assembly:
forever
    << stmts >>
    continue;
    << stmts >>
endfor;
```

// Converted code:

```
foreverLbl:
    << stmts >>
    // continue;
    jmp foreverLbl;
    << stmts >>
    jmp foreverLbl;
```

while..continue..endwhile

```
// Conversion of while loop with continue
// into pure assembly:
```

```
while( expr )do
    << stmts >>
    continue;
    << stmts >>
endwhile;
```

```
// Converted code:

whlLabel:
<< Code to evaluate expr >>
jcc EndOfWhile;          // Skip loop on expr failure.
    << stmts >
        // continue;
        jmp whlLabel;    // Jump to start of loop on continue.
    << stmts >>
        jmp whlLabel;    // Repeat the code.
EndOfwhile:
```

for..continue..endfor

```
// Conversion for a for loop with continue
// into pure assembly:

for( initStmt; expr; incStmt )do
    << stmts >>
    continue;
    << stmts >>
endfor;

// Converted code:

initStmt
ForLpLbl:
<<Code to evaluate expr >>
jcc EndOfFor;          // Branch if expression fails.
    << stmts >>
        // continue
        jmp ContFor; // Branch to incStmt on continue.
    << stmts >>

ContFor:
    incStmt
    jmp ForLpLbl;
EndOfFor:
```

repeat..continue..until

```
repeat
    << stmts >>
    continue;
    << stmts >>
until( expr );

// Converted Code:

RptLpLbl:
    << stmts >>
        // continue;
        jmp ContRpt;    // Continue branches to loop termination test.
```



```

    << stmts >>
ContRpt:
    << Code to test expr >>
    jcc RptLpLbl;           /* Jumps if expression evaluates false.

```

7.10.6 寄存器的使用与循环

因为 80x86 访问寄存器比访问存储器快得多, 所以寄存器是存放循环控制变量的理想位置(尤其对于小循环)。但是, 在循环中使用寄存器会有一些问题。主要的问题是虽然寄存器可以作为循环控制变量使用, 但寄存器的数目是有限的。下面的代码将不会正常工作, 因为它试图重用一個已经被使用的寄存器(CX):

```

    mov( 8, cx );
loop1:
    mov( 4, cx- );
loop2:
    << stmts >>
    dec( cx );
    jnz loop2;
    dec( cx );
    jnz loop1;

```

程序的意图显然是想创建一个嵌套的循环——也就是说, 一个循环在另一个循环体内。对于外部循环(loop1)的 8 次重复, 内部循环(loop2)每次将重复执行 4 次。但是, 两个循环使用了相同的寄存器作为循环控制变量。这会形成一个无限循环, 因为在第一次循环结束时, CX 值被置为 0; 在执行到第二个 dec 指令时, CX 的值总是 0, 控制将被转移给 loop1 标号(因为对 0 减 1 产生一个非 0 值)。解决的办法是保存并恢复 CX 的值, 或者使用一个不同的寄存器代替外层循环的 CX, 如下面的代码所示:

```

    mov( 8, cx );
loop1:
    push( cx );
    mov( 4, cx );
loop2:
    << stmts >>
    dec( cx );
    jnz loop2;

    pop( cx );
    dec( cx );
    jnz loop1;

```

或

```

    mov( 8, dx );
loop1:
    mov( 4, cx );
loop2:
    << stmts >>

```

```
        dec( cx );
        jnz loop2;

    dec( dx );
    jnz loop1;
```

在汇编语言中，寄存器的数值被破坏是循环产生漏洞的主要原因。请时刻牢记这个问题！

7.11 性能提高

80x86 微处理器以非常高的速度执行指令序列。很少会碰到执行速度很慢、但又不包含循环的程序。因为导致程序性能问题的主要来源就是循环，当想要改善程序性能时，首先要想到改善循环。虽然讨论如何编写高效率的程序不在本章讨论范围之内，但如果注意了下面几点，在程序中使用循环时就可以去掉许多不必要的指令，减少执行一次循环的时间，从而改善程序性能。

7.11.1 将结束条件判断放在循环结尾

前面提到的 3 种循环的流程如下所示：

```
repeat..until loop:
    Initialization code
        Loop body
    Test for termination
    Code following the loop

while loop:
    Initialization code
    Loop termination test
        , Loop body
        Jump back to test
    Code following the loop

forever..endfor loop:
    Initialization code
        Loop body part one
        Loop termination test
        Loop body part two
        Jump back to loop body part one
    Code following the loop
```

可以看出，**repeat..until** 循环是最简单的。这可以从实现这些循环的汇编代码看出。请看下面语义上等价的 **repeat..until** 循环和 **while** 循环：

```
// Example involving a WHILE loop:

mov( edi, esi );
sub( 20, esi );
while( esi <= edi )do

    << stmts >>
```



```

    inc( esi );

endwhile;

// Conversion of the code above into pure assembly language:

mov( edi, esi );
sub( 20, esi );
whlLbl:
    cmp( esi, edi );
    jnle EndOfWhile;

    << stmts >>
    inc( esi );
    << stmts >>
    jmp whlLbl;

EndOfWhile:

```

// Example involving a REPEAT..UNTIL loop:

```

mov( edi, esi );
sub( 20, esi );
repeat
    << stmts >>
    inc( esi );

until( esi > edi );

```

// Conversion of the REPEAT..UNTIL loop into pure assembly:

```

rptLabel:
    << stmts >>
    inc( esi );
    cmp( esi, edi );
    jng rptLabel;

```

仔细研究转换成纯汇编语言的过程，在循环结尾判断结束条件可以在循环中减少一条 `jmp` 指令。如果这个循环嵌套在另一循环中，这将变得非常有意义。在上面的代码中，循环体至少执行一次并没有问题。根据该循环的定义，可以看出循环正好执行 20 次。上面的示例转换成 `repeat.until` 循环是简单也是可行的。但是，有时候这种转换并不容易实现。请看下面的 HLA 代码：

```

while( esi <= edi )do
    << stmts >>
    inc( esi );
endwhile;

```

在这个示例中，循环入口点的 ESI 中包含的内容到底是什么，我们一点也不清楚，所以不能假设循环体至少会执行一次。所以在执行循环体之前，必须检测循环结束条件。检测包含一条单独的 `jmp` 指令，可以放在循环结尾，如下面的代码所示：


```

    jmp WhlTest;
TopOfLoop:
    << stmts >>
    inc( esi );
WhlTest:
    cmp( esi, edi );
    jle TopOfLoop;
    
```

虽然代码和原来的 while 循环一样长,但现在的代码 jmp 指令只执行一次,而原来的代码每次循环都要执行一次。注意,为了性能微弱的提升,代码的可读性有所降低。上面的代码比原来的实现更接近“意大利面条式”代码。这是为了获得性能提高通常要付出的代价。因此,必须仔细分析您的代码,保证性能提升的幅度确实值得牺牲代码的可读性。很多时候,汇编语言程序员牺牲了代码的可读性,产生了难以理解的代码,却没有获得性能的提高。

顺便提一句,HLA 使用了与本节完全相同的技术,将 while 语句翻译成汇编指令序列,其中循环结束条件判断位于循环的结尾。

7.11.2 反向执行循环

因为 80x86 标志的特性,如果循环计数从某个数递减(或递增)到 0,循环执行的效率要比从 0 递增(或递减)到某个值更高一些。比较下面的 HLA for 循环和它产生的代码:

```

for( mov( 1, j ); j <= 8; inc( j ) ) do
    << stmts >>
endfor;
    
```

// Conversion to pure assembly (as well as using a REPEAT..UNTIL form):

```

mov( 1, j );
ForLp:
    << stmts >>
    inc( j );
    cmp( j, 8 );
    jnge ForLp;
    
```

现在请看另一个同样重复 8 次的循环,但是它的控制变量是从 8 递减到 1,而不是从 1 递增到 8:

```

mov( 8, j );
LoopLbl:
    << stmts >>
    dec( j );
    jnz LoopLbl;
    
```

注意,以从 8 递减到 1 的方式执行循环,可以在每次循环重复时节约一次比较操作。

虽然不能让所有的循环都反向执行,但多花一点精力、多动一点脑筋,还是可以让很多循环反向执行的。通过每次迭代都减少一次比较操作,循环代码可以执行得更快。

上面的代码会工作得很好,因为循环控制变量从 8 减少到 1。当循环控制变量变为 0 时,循环结束。但当循环控制变量变为 0,还需要继续执行循环会发生什么情况呢?例如,假设将上面

示例中的循环范围变为从 7 到 0。只要上限是正数，就可以使用 `jns` 指令代替上面的 `jnz` 指令，让循环执行指定的次数。例如：

```
mov( 7, j );
LoopLbl:
    << stmts >>
    dec( j );
    jns LoopLbl;
```

上面的循环会重复 8 次，其中 `j` 值从 7 递减到 0。当 `j` 为 0，再减 1 时，符号标志就会被置位，从而循环结束。

注意，有些值看上去是正数，但实际上是负数。如果循环控制变量是一个字节，那么在补码系统中 128~255 范围的值就是负数。因此使用 129~255(或 0)之间的 8 位数值初始化循环控制变量，都会导致循环执行一次就结束。如果不注意这点的话，会给您带来很多麻烦。

7.11.3 循环不变计算

循环不变计算(loop-invariant computation)就是在循环过程中始终产生相同结果的计算。根本没有必要将这样的计算放在循环中，可以在循环外部计算，然后在循环内部引用计算所得的值。下面的 HLA 代码就是一个包含了循环不变计算的循环：

```
for( mov( 0, eax ); eax < n; inc( eax ) ) do
    mov( eax, edx );
    add( j, edx );
    sub( 2, edx );
    add( _edx, k );
endfor;
```

在循环执行时，`j` 的值保持不变，子表达式 `j - 2` 可以放在循环外部计算：

```
mov( j, ecx );
sub( 2, ecx );
for( mov( 0, eax ); eax < n; inc( eax ) ) do
    mov( eax, edx );
    add( ecx, edx );
    add( _edx, k );
endfor;
```

通过将子表达式 `j - 2` 放到循环外部计算，可以减少一条指令，但循环中还有一个不变成分，即 `ECX`。注意，这个不变成分在该循环中执行了 n 次，因此可以将上面的代码改进如下：

```
mov( j, ecx );
sub( 2, ecx );
intmul( n, ecx );           // Compute n*(j-2) and add this into k outside
add( ecx, k );             // the loop.
for( mov( 0, eax ); eax < n; inc( eax ) ) do
```

```

    add( eax, k );

endfor;

```

可以看到,循环体从4条指令减少到了1条指令。当然,如果您真的对改进这个循环感兴趣,可以根本不用循环来实现(因为对于循环完成的计算,可以使用公式来完成)。但是这个简单的示例只是为了说明如何从循环中去除循环不变计算。

7.11.4 循环展开

对于小循环——也就是循环体只包含少数几条语句的循环,为了处理循环而付出的系统开销可能占整个循环执行时间的很大一部分。例如,看一下下面的Pascal代码及相应的80x86汇编语言代码:

```

for i := 3 downto 0 do A[i] := 0;

mov( 3, i );
LoopLbl:
    mov( i, ebx );
    mov( 0, A[ ebx*4 ] );
    dec( i );
    jns LoopLbl;

```

循环的每次迭代需要4条指令,但只有1条指令执行真正需要的操作(将0移到A的元素中),其他3条指令都是用于循环重复的控制。因此为了完成逻辑上只需要4条指令的操作,该循环使用了16条指令。

可以使用到现在为止介绍的许多方法对这个循环进行改进。仔细观察这个循环完成的操作:仅仅是对A[0]到A[3]赋值0。更有效的实现方式是直接使用4条mov指令来完成。例如,如果A是双字数组,下面的代码就比上面的快得多:

```

mov( 0, A[0] );
mov( 0, A[4] );
mov( 0, A[8] );
mov( 0, A[12] );

```

这只是一个简单的示例,但它说明了循环展开(loop unraveling)的好处。如果这个简单的循环嵌套在一组循环中,4:1的指令减少可能使程序的性能提高一倍。

当然,不可能把所有的循环都展开。执行次数可变的循环就不能展开,因为(在汇编时)没有办法确定循环到底执行了多少次。因此循环展开只适用于循环次数固定(在汇编时知道循环的次数)的循环。

即使对于循环次数固定的循环,有时候循环展开也不是一个好方法。如果控制循环(以及处理其他额外开销的)的指令条数占循环总指令条数相当大的一部分,循环展开能够明显地提高性能;否则就不是这样。例如,对于前面的循环,如果循环体包含了36条指令(不包含4条处理额外开销的指令),程序性能提高的比例为10%(而不是现在的300%~400%)。随着循环体的增大,循环展开带来的性能提升将很快减少,另外循环展开的代价——也就是所有必须在程序中插入的代码——却随循环次数的增加而增大。而且,在程序中输入这些代码是一件琐碎麻烦的事。因此,循

环展开技术最好只对小循环使用。

注意,在超标量 80x86(奔腾或更新的)芯片中,使用了分支预测硬件(branch prediction hardware)和其他技术来提高性能。在这样的系统中,循环展开可能会使代码变得更慢,因为这些处理器已经对小循环的执行进行了优化。

7.11.5 归纳变量

请看下面这个循环:

```
for i := 0 to 255 do csetVar[i] := {};
```

这里,程序将字符集数组中的每个元素设置为空集。可以用下面的汇编代码完成:

```
mov(0, i);
FLP:
    // Compute the index into the array (note that each element
    // of a CSET array contains 16 bytes).

    mov( i, ebx );
    shl( 4, ebx );

    // Set this element to the empty set (all 0 bits).

    mov( 0, csetVar[ ebx ] );
    mov( 0, csetVar[ ebx+4 ] );
    mov( 0, csetVar[ ebx+8 ] );
    mov( 0, csetVar[ ebx+12 ] );

    inc( i );
    cmp( i, 256 );
    jb FLP;
```

虽然展开这样的代码也会提升性能,但需要 1024 条指令,除了对时间要求特别严格的应用程序之外,这对于大多数应用程序来说太多了。这时,可以使用归纳变量(induction variables)来减少循环体执行的时间。归纳变量的值完全取决于其他变量的值。在上面的示例中,索引数组的 csetVar 值隐含了循环控制变量的值(它的值总是循环控制变量值的 16 倍)。因为在循环中没有其他地方使用 i,就没有必要对 i 进行计算。为什么不直接使用数组的索引值呢?下面的代码使用了这种技巧:

```
mov( 0, ebx )
FLP:
    mov( 0, csetVar[ ebx ] );
    mov( 0, csetVar[ ebx+4 ] );
    mov( 0, csetVar[ ebx+8 ] );
    mov( 0, csetVar[ ebx+12 ] );

    add( 16, ebx );
    cmp( ebx, 256*16 );
    jb FLP;
```

在上面的示例中,循环控制变量每次增加 16(为了提高效率,可将控制变量放到 EBX 中),而不是 1。通过将循环控制变量(以及最后终止循环的常量值)增加 16,能够消除在每次循环时都将循环变量乘以 16(也就是,能够消除前一代码中的 shl 指令)。而且,代码不再引用原来的循环控制变量(i),代码可以使循环控制变量总在 EBX 寄存器中。

7.12 HLA 中的混合控制结构

HLA 高级语言控制结构存在一些缺点:①它们不是真正的汇编语言指令;②复杂的布尔表达式只支持短路布尔求值;③HLA 会引入一些不好的编程实践,很多人只有在编写高性能代码时才会这么做。另一方面,80x86 控制结构虽然让您编写高性能的代码,但最终的代码可读性较差,难以维护。因此,HLA 提供了一种混合控制结构,让您可以使用纯汇编语言语句计算布尔表达式,使用高级控制结构描述布尔表达式控制的语句。最后产生的代码可读性比汇编语言好得多,同时也不会有很大的性能损失。

HLA 提供的混合语句有:if..elseif..else..endif、while..endwhile、repeat..until、breakif、exitif 和 continueif(也就是包含布尔表达式的语句)。例如,混合 if 语句的形式如下:

```
if( #{ instructions }# ) then statements endif;
```

注意#{和}#操作符的使用,它们在这个语句中包含了一个指令序列。这也是混合控制结构与高级控制结构不同的地方。其他几种混合控制结构的形式如下:

```
while( #{ statements }# ) statements endwhile;
repeat statements until ( #{ statements }# );
breakif( #{ statements }# );
exitif( #{ statements }# );
continueif( #{ statements }# );
```

大括号包含的语句替代了 HLA 高级控制结构正常情况时的布尔表达式,HLA 在这些特殊语句的上下文中定义了两个伪标号, true 和 false。如果布尔表达式存在且表达式的值为真,HLA 将此时要执行的语句与标号 true 关联;类似地,将布尔表达式值为假时要执行的语句与标号 false 关联。作为一个简单示例,请看下面两个(等价的)if 语句:

```
if( eax < ebx ) then inc( eax ); endif;

if
( #{
    cmp( eax, ebx );
    jnb false;
}# ) then
    inc( eax );
endif;
```

如果 EAX 不小于 EBX, jnb 指令将控制转移到 false 标号,并跳过 inc 语句。注意,如果 EAX 小于 EBX,控制转移到 inc 指令。它基本等价于下面的纯汇编代码:


```

cmp( eax, ebx );
jnl falseLabel;
    inc( eax );
falseLabel:

```

请看下面稍微复杂一点的示例:

```

if( eax >= j && eax <= k ) then sub( j, eax ); endif;

```

下面的混合 if 语句完成相同的功能:

```

if
(#{
    cmp( eax, j );
    jnge false;
    cmp( eax, k );
    jnle false;
}#) then
    sub( j, eax );
endif;

```

最后, 请看下面这个混合 if 语句:

```

// if( ((eax > ebx)&&(eax < ecx)) || (eax = edx)) then
    mov( ebx, eax );
endif;

if
(#{
    cmp( eax, edx );
    je true;
    cmp( eax, ebx );
    jng false;
    cmp( eax, ecx );
    jnl false;
}#) then
    mov( ebx, eax );
endif;

```

因为这些示例非常简单, 并不能说明使用混合语句而不是纯汇编语句会使代码的可读性好多少。但是, 您可能注意到了一件事: 使用混合语句可以不需要在代码中插入标号。这点就可以使程序可读性增加, 易于理解。

对于 if 语句, true 标号对应语句中的 then 子句, false 标号对应于 elseif、else 或 endif 子句(只要紧跟在 then 子句后)。对于 while 语句, true 标号对应于循环体, 而 false 标号对应于紧跟在 endwhile 后的第一条语句。对于 repeat..until 语句, true 标号对应于紧跟在 until 子句后的代码, false 标号对应于循环体的第一条语句。breakif、exitif 和 continueif 语句的 false 标号对应于紧跟在这些语句后的语句, true 标号则分别对应于一般与 break、exit 和 continue 关联的语句。

7.13 更多信息

除了本章介绍的内容外,HLA 还包含其他一些高级控制结构,如 `try..endtry` 和 `foreach` 语句。本章没有讨论这些语句,因为它们是比较高级的控制结构,其实现过程过于复杂,不适合在本书讨论。如果需要了解更多的信息,请参见本书的电子版本(<http://www.artofasm.com/> 或 <http://webster.cs.ucr.edu/>)或 HLA 参考手册。

第 8 章

高级算术运算



本章将讨论一些特别适合使用汇编语言的算术运算。这些内容包括 4 个主题：扩展精度的算术运算、操作数大小不同的算术运算、十进制算术运算和借助表查找的计算。

本章讲述最多的是多精度算术运算，学完本章之后，您将知道如何将算术运算和逻辑运算应用于不同大小的整型操作数。如果需要对超出 ± 2000000000 的整数值(或超出 4000000000 的无符号值)进行运算，也没关系，这一章会讲述如何完成这样的工作。

大小不同的操作数也会给算术运算带来一些特殊的问题。比如说，可能想要将一个 64 位无符号整数和一个 128 位有符号整数相加。这一章将讨论如何把这两种操作数转换成一种兼容的格式，以便运算能够进行。

这一章还将讨论十进制算术运算，该运算利用了 80x86 的 BCD(binary-coded decimal)指令和 FPU(floating-point unit)。这使得在那些极少数必须基于十进制(而不是二进制)进行运算的应用程序中使用十进制算术运算成为可能。

最后，本章的结束部分将讨论如何使用表查找来加快复杂计算的速度。

8.1 多精度操作

汇编语言与高级语言相比，一个很大的优点是汇编语言不限制整数运算的大小。比如说，标准的 C 编程语言定义了 3 种不同的整数大小：short int、int 和 long int¹。在 PC 上，这些常常是 16 位和 32 位整数。虽然 80x86 机器指令限制只能用一条指令处理 8 位、16 位和 32 位整数，但通常可使用多条指令来处理所需要的任何长度的整数。如果要将 256 位整数值加在一起，也没有问题：

¹ 最新的 C 标准还提供了一种 long long int 数据类型，这一般是 64 位的整数。

但使用汇编语言来做这件事相对来说要简单一些。下面一节将讲述如何将各种算术和逻辑运算从 16 或 32 位扩展到任何想要的位数。

8.1.1 扩展精度操作的 HLA 标准库支持

虽然理解如何实现扩展精度的算术运算比较重要,但应该注意,HLA 标准库提供了一整套的 64 位和 128 位算术和逻辑函数。这些例程的应用非常广泛,而且非常便于使用。这一节将简要讲述扩展精度算术运算的 HLA 标准库支持。

正如前面章节中指出的那样,HLA 编译器支持几种不同的 64 位和 128 位数据类型。作为一个回顾,这里列出了 HLA 所支持的扩展数据类型:

- `uns64`: 64 位无符号整数
- `int64`: 64 位有符号整数
- `qword`: 64 位无类型值
- `uns128`: 128 位无符号整数
- `int128`: 128 位有符号整数
- `lword`: 128 位无类型值

HLA 还提供了一种 `tbyte` 类型,但我们在这里不考虑它(参见 8.2 节)。

HLA 完全支持 64 位和 128 位的字面常量和常量的算术运算,这样就可以使用标准十进制、十六进制或二进制表示法初始化 64 位和 128 位静态对象。例如:

```
static
    ul128    :uns128    := 123456789012345678901233567890;
    i64      :int64     := -12345678901234567890;
    lw       :lword     := $1234_5678_90ab_cdef_0000_ffff;
```

为了能简单地运算 64 位和 128 位值,HLA 标准库的 `math.hhf` 模块提供了一组函数来处理所需的大多数算术和逻辑运算。使用这些函数的方式非常类似于 32 位算术指令和逻辑指令。例如,考虑 `math.addq(qword)` 和 `math.addl(lword)` 函数:

```
math.addq( left64, right64, dest64 );
math.addl( left128, right128, dest128 );
```

这些函数进行下列计算:

```
dest64 := left64 + right64;      // dest64, left64, and right64
                                   // must be 8-byte operands
dest128 := left128 + right128;   // dest128, left128, and right128
                                   // must be 16-byte operands
```

这些函数就像一条 `add` 指令执行完毕后那样设置 80x86 标志。特别地,如果(完整的)计算结果为 0,那么这些函数将设置零标志;如果在最高位上有进位,则设置进位标志;如果出现了有符号运算的溢出,则设置溢出标志;如果最高位的结果包含 1,则设置符号标志。

其余的大多数算术和逻辑例程都使用与 `math.addq` 和 `math.addl` 相同的调用序列。下面列出了这些函数:

```

math.andq( left64, right64, dest64 );
math.andl( left128, right128, dest128 );
math.divq( left64, right64, dest64 );
math.divl( left128, right128, dest128 );
math.idivq( left64, right64, dest64 );
math.idivl( left128, right128, dest128 );
math.modq( left64, right64, dest64 );
math.modl( left128, right128, dest128 );
math.imodq( left64, right64, dest64 );
math.imodl( left128, right128, dest128 );
math.mulq( left64, right64, dest64 );
math.mull( left128, right128, dest128 );
math.imulq( left64, right64, dest64 );
math.imull( left128, right128, dest128 );
math.orq( left64, right64, dest64 );
math.ordl( left128, right128, dest128 );
math.subq( left64, right64, dest64 );
math.subl( left128, right128, dest128 );
math.xorq( left64, right64, dest64 );
math.xorl( left128, right128, dest128 );

```

这些函数使用与其相对应的 32 位机器指令同样的方式设置标志，并且对于除法和取模函数的情形，将产生相同的异常。注意，乘法运算函数不会产生扩展精度的结果，其目的值和源操作数值的大小是一样的。如果结果与目的操作数的大小不匹配，这些函数将设置溢出和进位标志。所有这些函数进行下列计算：

```

dest64 := left64 op right64;
dest128 := left128 op right128;

```

其中，*op* 代表特定的运算。

除了上述函数，HLA 标准库的数学运算模块还提供了一些其他的函数，它们的语法稍有不同。这些函数包括 `math.negq`、`math.negl`、`math.notq`、`math.notl`、`math.shlq`、`math.shll`、`math.shrq` 和 `math.shrl`。注意，这里没有循环右移和算术右移函数，但是，我们很快就会看到，可以简单地使用标准指令来合成这些运算。下面给出这些函数的原型：

```

math.negq( source:qword; var dest:qword );
math.negl( source:lword; var dest:lword );
math.notq( source:qword; var dest:qword );
math.notl( source:lword; var dest:lword );
math.shlq( count:uns32; source:qword; var dest:qword );
math.shll( count:uns32; source:lword; var dest:lword );
math.shrq( count:uns32; source:qword; var dest:qword );
math.shrl( count:uns32; source:lword; var dest:lword );

```

同样，这里的函数也是使用与其相对应的支持 64 位或 128 位操作数的机器指令相同的方式设置标志。

HLA 标准库还提供了一些补充的内容，即 64 位和 128 位数值的 I/O 和转换例程。例如，可以使用 `stdout.put` 来显示 64 位和 128 位数值，也可以使用 `stdin.get` 来读取这些数值。而且，在 HLA

转换模块中还有一些例程可以将这些数值和与它们等价的字符串来回转换。一般来说，任何可以对 32 位值进行的运算都可以对 64 位或 128 位值进行。要获取更多细节，请参考 HLA 标准库的文档。

8.1.2 多精度加法运算

80x86 的 add 指令将两个 8 位、16 位或 32 位数相加。在执行完 add 指令后，如果在和的高位位上有溢出，那么 80x86 的进位标志就会被设置。这一信息可以用来进行多精度加法运算。考虑下面的方法，可以手工地进行多位(多精度)加法运算：

第 1 步：将最低位数字相加

289
+456

得到

289
+456

5 进位为 1

第 2 步：将次低位数字加上进位

1 (前一步的进位)
289
+456

得到

289
+456

5 45 进位为 1

第 3 步：将最高位数字加上进位

1 (前一步的进位)
289
+456

得到

289
+456

45 745

80x86 以相同的方式处理扩展精度算术运算，每次将一个字节、一个字或者一个双字相加，而不是像对数字执行加法运算时逐位相加。考虑图 8-1 中 3 个双字(96 位)的加法运算。

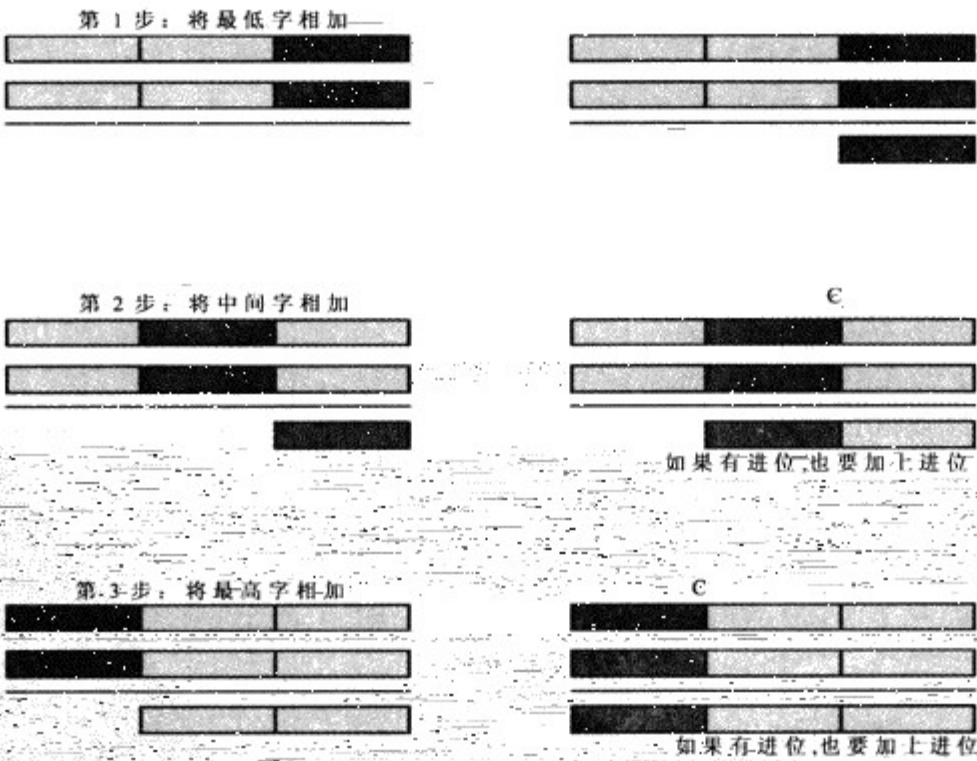


图 8-1 将两个 96 位对象相加

正如在图 8-1 中看到的那样，这里的思想是将一个较大的运算分解成一系列较小的运算。因为 x86 系列处理器一次最多只能将 32 位数值相加，所以第 1 步就是要将两个低位双字相加，正如在将十进制数手工进行相加的算法中，我们将两个低位数位相加一样。这个运算没有什么特别之处，可以使用 `add` 指令来完成。

第 2 步是将两个 96 位数值中的第二对双字相加。在第 2 步中请注意，要将前一步产生的进位(如果有的话)也一起加进来。如果在低位加法过程中有进位，那么 `add` 指令将会把进位标志设置为 1；相反，如果低位加法过程中没有进位，先前的 `add` 指令就会清除进位标志。因此，在第二次相加的时候，我们实际上需要计算两个双字的和，并加上第一条指令产生的进位。所幸的是，x86 CPU 提供了一条指令恰好可以完成这项工作，即 `adc(add with carry)` 指令。`adc` 指令使用与 `add` 指令相同的语法，进行的运算也几乎相同：

```
adc( source, dest );    // dest := dest + source + C
```

`add` 指令和 `adc` 指令唯一的区别是，`adc` 指令将进位标志的值和源操作数与目的操作数一起相加。它还以与 `add` 指令相同的方式设置标志(如果有无符号溢出的话，还包括设置进位标志)。这正是我们将 96 位的中间两个双字相加所需要的。

在图 8-1 的第 3 步中，算法将 96 位值中的高位双字相加。这个加法运算必须也将中间两个双字相加结果的进位合并进来，这样就又需要用到 `adc` 指令。简言之，`add` 指令将低位双字相加，`adc` 指令将所有剩下的双字对相加。在扩展精度加法运算序列结束后，进位标志表示无符号溢出(如果该位被设置)，被设置的溢出标志表示有符号溢出，而符号标志表示结果的符号。在扩展精度加法运算序列结束后，零标志不具有任何实际含义(它只说明两个高位双字的和为 0，而不意味着整个结果为 0)。如果想要知道如何检查扩展精度的 0 结果，请参考 HLA 标准库中 `math.addq` 或 `math.addl` 函数的源代码。

例如，假设有两个 64 位值要相加，就作如下的定义：

```
static
  X: qword;
  Y: qword;
```

再假设想要将加法的和保存在第三个变量 `Z` 里，而且这个变量也是 `qword` 类型的。那么下面的 80x86 代码可以完成这个任务：

```
mov( (type dword X), eax );    // Add together the L.O. 32 bits
add( (type dword Y), eax );    // of the numbers and store the
mov( eax, (type dword Z));     // result into the L.O. dword of Z.

mov( (type dword X[4]), eax ); // Add together (with carry) the
adc( (type dword Y[4]), eax );  // H.O. 32 bits and store the result
mov( eax, (type dword Z[4]));   // into the H.O. dword of Z.
```

记住，这些变量都是 `qword` 对象。因此，编译器不会接受形如 `mov(X, eax);` 的指令，因为这条指令试图将一个 64 位值载入一个 32 位的寄存器。这段代码使用了强制操作符，将符号 `X`、`Y` 和 `Z` 强制为 32 位。其中，前三条指令将 `X` 和 `Y` 的低位双字相加，并将结果存储在 `Z` 的低位双字中。后三条指令将 `X` 和 `Y` 的高位双字以及低位双字加法得到的进位相加，并将结果保存到 `Z` 的高位

双字中。记住，形如 `X[4]` 的地址表达式用于访问一个 64 位数据项的高位双字。这是因为 x86 存储器空间以字节编址，而 4 个连续的字节形成一个双字。

地址的位数可以扩展到任意多位，方法是使用 `adc` 指令将高位数值相加。例如，要将两个 128 位数值相加，就可以使用与下面相似的代码：

```
type
    tBig: dword[4];           // Storage for four dwords fs 128 bits.

static
    BigVal1: tBig;
    BigVal2: tBig;
    BigVal3: tBig;

    .
    .
    .

    mov( BigVal1[0], eax );    // Note there is no need for (type dword BigValx)
    add( BigVal2[0], eax );    // because the base type of BitValx is dword.
    mov( eax, BigVal3[0] );

    mov( BigVal1[4], eax );
    adc( BigVal2[4], eax );
    mov( eax, BigVal3[4] );

    mov( BigVal1[8], eax );
    adc( BigVal2[8], eax );
    mov( eax, BigVal3[8] );

    mov( BigVal1[12], eax );
    adc( BigVal2[12], eax );
    mov( eax, BigVal3[12] );
```

8.1.3 多精度减法运算

像加法一样，80x86 进行多字节减法的运算和手工执行的方式相同，只是每次相减的不是一个十进制位，而是整个字节、字或双字。减法运算的机制类似于 `add` 运算，对于低位字节/字/双字，使用 `sub` 指令；对于高位值，使用 `sbb(subtract with borrow)` 指令。

下面的示例演示了在 80x86 上使用 32 位寄存器的 64 位减法运算：

```
static
    Left:    qword;
    Right:   qword;
    Diff:    qword;

    .
    .
    .

    mov( (type dword Left), eax );
    sub( (type dword Right), eax );
    mov( eax, (type dword Diff) );

    mov( (type dword Left[4]), eax );
```

```
sbb( (type dword Right[4]), eax );
mov( (type dword Diff[4]), eax );
```

下面的示例演示了一次 128 位减法运算：

```
type
    tBig: dword[4]; // Storage for four dwords is 128 bits.

static
    BigVal1: tBig;
    BigVal2: tBig;
    BigVal3: tBig;
    .
    .
    .
    // Compute BigVal3 := BigVal1 - BigVal2

    mov( BigVal1[0], eax ); // Note there is no need for (type dword BigValx)
    sub( BigVal2[0], eax ); // because the base type of BitValx is dword.
    mov( eax, BigVal3[0] );

    mov( BigVal1[4], eax );
    sbb( BigVal2[4], eax );
    mov( eax, BigVal3[4] );

    mov( BigVal1[8], eax );
    sbb( BigVal2[8], eax );
    mov( eax, BigVal3[8] );

    mov( BigVal1[12], eax );
    sbb( BigVal2[12], eax );
    mov( eax, BigVal3[12] );
```

8.1.4 扩展精度比较操作

但是，没有 `compare with borrow` 指令可以进行扩展精度的比较操作。因为至少就标志来说，`cmp` 和 `sub` 指令进行相同的操作，所以通常认为能够使用 `sbb` 指令来合成一次扩展精度的比较操作，但是这并不是总能得到正确的结果。幸好还有一种更好的方法来做这件事。

考虑两个无符号数值\$2157 和\$1293。这两个数值的低位字节对比较的结果不产生影响，只须比较高位字节\$21 和\$12，就可以知道第一个值比第二个值大。实际上，只有高位字节相等的时候，我们才需要考察所有的字节。而在其他任何情况下，只要比较高位字节就可以了。当然，这一事实对于任何数量的值来说都是正确的，而不仅仅针对两个值的情况。下面的代码比较了两个有符号的 64 位整数，先比较它们的高位双字，只有高位双字相等时，才比较它们的低位双字。

```
// This sequence transfers control to location "IsGreater" if
// QwordValue > QwordValue2. It transfers control to "IsLess" if
// QwordValue < QwordValue2. It falls though to the instruction
// following this sequence if QwordValue = QwordValue2. To test for
// inequality, change the "IsGreater" and "IsLess" operands to "NotEqual"
// in this code.
```

```

mov( (type dword QWordValue[4]), eax ); // Get H.O. dword.
cmp( eax, (type dword QWordValue2[4]));
jg IsGreater;
jl IsLess;

mov( (type dword QWordValue[0]), eax ); // If H.O.dwords were equal,
cmp( eax, (type dword QWordValue2[0])); // then we must compare the
jg IsGreater;                          // L.O.dwords.
jl IsLess;

```

```
// Fall through to this point if the two values were equal.
```

要想比较无符号的数值，只要使用 `ja` 和 `jb` 指令替换 `jg` 和 `jl` 即可。

可以从上述指令序列出发简单地合成任何可能的比较操作，下面的示例显示了如何做这件事。这些示例演示了有符号值的比较操作；如果想要进行无符号值的比较，只要使用 `ja`、`jae`、`jb` 和 `jbe` 分别替换 `jg`、`jge`、`jl` 和 `jle` 即可。下面的示例都使用了如下的声明：

```

static
    QW1: qword;
    QW2: qword;

const
    QW1d: text := "(type dword QW1)";
    QW2d: text := "(type dword QW2)";

```

下面的示例测试 `QW1 < QW2` (两者都是 64 位有符号数)。如果 `QW1 < QW2`，则控制转移到 `IsLess` 标号，否则控制转移到下一条语句。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );
jg NotLess;
jl IsLess;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
jl IsLess;

```

```
NotLess:
```

下面的示例测试 `QW1 <= QW2` (两者都是 64 位有符号数)。如果条件为真，则控制转移到 `IsLessEQ` 标号。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );
jg NotLessEQ;
jl IsLessEQ;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
jle IsLessEQ;

```

```
NotLessEQ:
```


下面的示例测试 $QW1 > QW2$ (两者都是 64 位有符号数)。如果条件为真, 则控制转移到 IsGtr 标号。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );
jg IsGtr;
jl NotGtr;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
jg IsGtr;

```

NotGtr:

下面的示例测试 $QW1 \geq QW2$ (两者都是 64 位有符号数)。如果条件为真, 则控制转移到 IsGtrEQ 标号。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );
jg IsGtrEQ;
jl NotGtrEQ;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
jge IsGtrEQ;

```

NotGtrEQ:

下面的示例测试 $QW1 = QW2$ (两者都是 64 位数, 可以有符号或者无符号)。如果 $QW1 = QW2$, 代码会分支到标号 IsEqual 处。如果它们不相等, 则转移到下一条指令。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );

jne NotEqual;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
je IsEqual;

```

NotEqual:

下面的示例测试 $QW1 < > QW2$ (两者都是 64 位数, 可以有符号或者无符号)。如果 $QW1 < > QW2$, 代码会分支到标号 NotEqual 处。如果它们相等, 则转移到下一条指令。

```

mov( QW1d[4], eax ); // Get H.O. dword.
cmp( eax, QW2d[4] );
jne NotEqual;

mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
cmp( eax, QW2d[0] );
jne NotEqual;

```

// Fall through to this point if they are equal.

如果要进行一次扩展精度比较操作的话,还不能直接使用 HLA 高级控制结构。但是,可以使用 HLA 混合控制结构将适当的比较操作隐藏在布尔表达式中,这样做可能会使代码更易于阅读。例如,下面的 if.then.else.endif 语句使用一次 64 位扩展精度无符号值的比较操作检查 QW1>QW2 是否成立:

```
if
  ( # (
    mov( -QW1d[4], eax );
    cmp( eax, QW2d[4] );
    jg true;

    mov( QW1d[0], eax );
    cmp( eax, QW2d[0] );
    jng false;
  )# ) then
  << Code to execute if QW1 > QW2 >>

else
  << Code to execute if QW1 <= QW2 >>

endif;
```

如果需要对大于 64 位的对象进行比较,那么可以简单地将前面针对 64 位操作数的代码进行推广。比较过程也是从对象的高位双字开始的,然后,只要对应的双字相等,就向下进行比较,直至低位双字。下面的示例比较了两个 128 位数值,看第一个数是否小于等于第二个数(无符号比较):

```
static
  Big1: uns128;
  Big2: uns128;

if
  ( # (
    mov( Big1[12], eax );
    cmp( eax, Big2[12] );
    jb true;
    jg false;
    mov( Big1[8], eax );
    cmp( eax, Big2[8] );
    jb true;
    jg false;
    mov( Big1[4], eax );
    cmp( eax, Big2[4] );
    jb true;
    jg false;
    mov( Big1[0], eax );
```

```

    cmp( eax, Big2[0] );
    jnbe false;
}# ) then

    << Code to execute if Big1 <= Big2 >>

else

    << Code to execute if Big1 > Big2 >>

endif;

```

8.1.5 扩展精度乘法操作

虽然 8×8 、 16×16 或 32×32 位的乘法操作通常已经足够, 但有些时候可能想要将更大的数值相乘。这时可以使用 x86 的单操作数指令 `mul` 和 `imul` 来实现扩展精度的乘法操作。

同样(考虑到我们使用 `adc` 和 `sbb` 实现了扩展精度加减操作), 使用与手工实现两个数值乘法运算相同的技术, 我们就可以在 80x86 上进行扩展精度的乘法操作。考虑一下我们手工进行多位相乘操作的简化形式:

1) 将最后两个数位相乘 (5×3):

```

  123
  45
  ---
  15

```

2) 将 5 与 2 相乘:

```

  123
  45
  ---
  15
  10

```

3) 将 5 与 1 相乘:

```

  123
  45
  ---
  15
  10
  5

```

4) 将 4 与 3 相乘:

```

  123
  45
  ---
  15
  10
  5
  12

```

5) 将 4 与 2 相乘:

```

  123
  45
  ---
  15
  10
  5
  12
  8

```

6) 将 4 与 1 相乘:

```

  123
  45
  ---
  15
  10
  5
  12
  8
  4

```

7) 对所有的部分积求和:

```

  123
  45
  ---
  15
  10
  5

```


12
8
4

5535

除了所操作的是字节、字和双字而不是位以外，80x86 进行扩展精度乘法操作的方式与此相同。图 8-2 显示了工作原理。

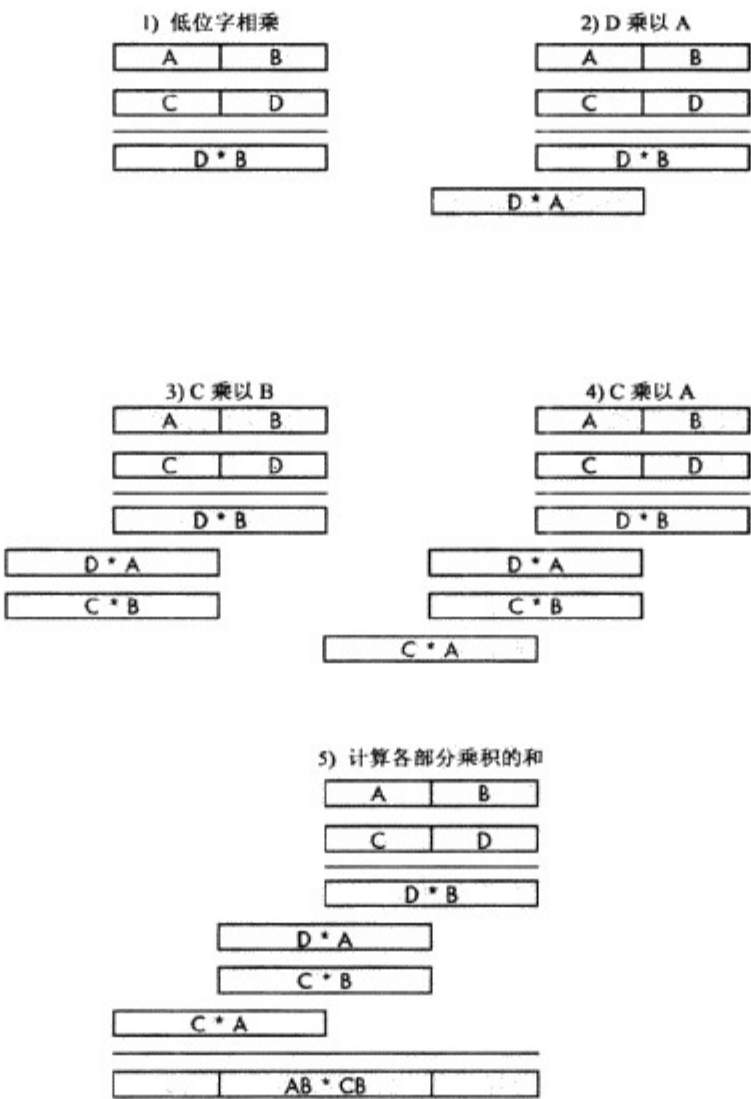


图 8-2 扩展精度乘法操作

切记，在进行扩展精度乘法操作的时候，也必须同时进行多精度加法操作。将所有的部分乘积相加需要几次加法才能得出结果。程序清单 8-1 演示了在一个 32 位处理器上将两个 64 位数值相乘的正确方法：

程序清单 8-1 扩展精度乘法操作

```
program testMUL64;  
#include( "stdlib.hhf" )  
  
procedure MUL64( Multiplier:qword; Multiplicand:qword; var Product:lword );  
const  
    mp: text := "( type dword Multiplier )";  
    mc: text := "( type dword Multiplicand )";  
    prd: text := "( type dword [edi] )";
```

```

begin MUL64;

    mov( Product, edi );

    // Multiply the L.O. dword of Multiplier times Multiplicand.

    mov( mp, eax );
    mul( mc, eax );           // Multiply L.O. dwords.
    mov( eax, prd );          // Save L.O. dword of product.
    mov( edx, ecx );          // Save H.O. dword of partial product result.

    mov( mp, eax );
    mul( mc[4], eax );        // Multiply mp(L.O.) * mc(H.O.)
    add( ecx, eax );           // Add to the partial product.
    adc( 0, edx );            // Don't forget the carry!
    mov( eax, ebx );          // Save partial product for now.
    mov( edx, ecx );

    // Multiply the H.O. word of Multiplier with Multiplicand.

    mov( mp[4], eax );        // Get H.O. dword of Multiplier.
    mul( mc, eax );           // Multiply by L.O. word of Multiplicand.
    add( ebx, eax );           // Add to the partial product.
    mov( eax, prd[4] );        // Save the partial product.
    adc( edx, ecx );           // Add in the carry!

    mov( mp[4], eax );        // Multiply the two H.O. dwords together.
    mul( mc[4], eax );
    add( ecx, eax );           // Add in partial product.
    adc( 0, edx );            // Don't forget the carry!
    mov( eax, prd[8] );        // Save the partial product.
    mov( edx, prd[12] );

end MUL64;

static
    op1: qword;
    op2: qword;
    rslt: lword;

begin testMUL64;

    // Initialize the qword values (note that static objects
    // are initialized with 0 bits).

    mov(1234, ( type dword op1 ));
    mov(5678, ( type dword op2 ));
    MUL64( op1, op2, rslt );

    // The following only prints the L.O. qword, but
    // we know the H.O. qword is 0 so this is okay.

    stdout.put( "rslt=" );
    stdout.putu64( (type qword rslt));

end testMUL64;

```

关于这段代码有一件事必须要牢记,那就是它只对无符号操作数正常工作。要将两个有符号的数值相乘就必须在乘法操作之前记录操作数的符号,取两个操作数的绝对值,做一次无符号乘法操作,然后根据原操作数的符号调整结果积的符号。有符号操作数的乘法留作读者的练习(或者您可以查看 HLA 标准库中的源代码)。

程序清单 8-1 中的示例是相当直接的,因为可以将部分积保存在不同的寄存器中。如果需要将较大的数值相乘,就需要在临时(存储器)变量中保存部分积。除了这点不同以外,程序清单 8-1 中使用的算法可以推广到任意个数的双字乘法。

8.1.6 扩展精度除法操作

使用 `div` 和 `idiv` 指令来合成一般的 n 位/ m 位的除法操作是行不通的,必须使用一系列移位操作和减法操作才能进行这样的除法操作,而且这些操作非常杂乱。但是,有一种不太常见的操作很容易使用 `div` 指令合成,即将一个 n 位的数值除以一个 32 位的数值。本节将给出扩展精度除法操作的两种方法。

在描述如何执行多精度除法操作之前,应该注意的是,一些操作即使它们看上去使用 `div` 或者 `idiv` 指令就可以完成计算,但实际上需要进行一次扩展精度除法操作。将一个 64 位的数值除以一个 32 位的数值比较简单,只要结果商能在 32 位之内,`div` 和 `idiv` 指令就可以直接处理这个操作。但是,如果商不在 32 位以内,那么就不得不将这个问题作为一次扩展精度除法操作来处理。这里的技巧是用除数去除被除数的(零扩展或者符号扩展的)高位双字;然后对余数以及被除数的低位双字重复该过程。下面的序列演示了这一过程:

```
static
dividend: dword[2] := {$1234, 4};      // = $4_0000_1234.
divisor:  dword := 2;                  // dividend/divisor = $2_0000_091A
quotient: dword[2];
remainder: dword;
.
.
.
mov( divisor, ebx );
mov( dividend[4], eax );
xor( edx, edx );                      // Zero extend for unsigned division.
div( ebx, edx:eax );
mov( eax, quotient[4] );              // Save H.O. dword of the quotient (2).
mov( dividend[0], eax );              // Note that this code does *NOT* zero extend
div( ebx, edx:eax );                  // eax into edx before this div instr.
mov( eax, quotient[0] );              // Save L.O. dword of the quotient ($91a).
mov( edx, remainder );               // Save away the remainder.
```

因为将一个值除以 1 是完全合法的,所以当除法操作完成后,商所需的位数当然也有可能和被除数的相同。这就是为什么在这个示例中,`quotient` 变量和 `dividend` 变量的大小相同的原因(注意,应该使用一个由两个双字构成的数组,而不是 `qword` 类型;这就省得将操作数强制转换成双字)。不管被除数和除数操作数的大小如何,余数的大小不会比除法操作本身的还大(本例中是 32 位)。这样一来本例中的 `remainder` 变量就只是一个双字而已。

在分析这段代码如何工作之前，让我们简要地看一看，为什么在这个特殊的示例中，即使 `div` 指令确实计算了一个 64 位/32 位除法操作的结果，而该操作本身却仍然没有工作。这里我们作一个天真的假设，假设 x86 能够完成这个操作，那么请看下面的代码：

```
// This code does *NOT* work!

mov( dividend[0], eax );           // Get dividend into edx:eax
mov( dividend[4], edx );
div( divisor, edx:eax );           // Divide edx:eax by divisor.
```

虽然这段代码在句法上是正确的，而且能够通过编译，但是如果试图运行它的话，就会产生一个 `ex.DivideError2` 异常。原因是商必须被限制在 32 位之内。这里就是因为所产生的商是 `$2_0000_091A`，它不能被存入 EAX 寄存器，这样就导致了异常。

现在让我们再来看一下前面提到的那段正确计算 64 位/32 位商的代码。这段代码首先计算 `dividend[4]/divisor` 的 32 位/32 位商。这个除法操作的商(2)变成了最终商的高位双字。该操作的余数(0)变成了 EDX 的扩展部分，以便进行除法操作的后半部分。这段代码的后半部分将 `edx:dividend[0]` 除以 `divisor`，来产生商的低位双字以及除法操作的余数。注意，这段代码在第二个 `div` 指令之前没有将 EAX 零扩展到 EDX，EDX 已经具有合法的值，而这段代码不能破坏它们。

上述的 64 位/32 位除法操作实际上只是一般除法操作的特例，这些除法操作允许将一个任意大小的值除以一个 32 位的除数。要实现这些，首先要将被除数的高位双字移至 EAX，并对其零扩展为 EDX。接下来，将这一数值除以除数。然后，在不修改 EDX 的情况下，将部分商储存起来，将被除数中次低位上的双字载入 EAX，并将其除以除数。重复这样的操作，直到处理完被除数中所有的双字。这时，EDX 寄存器将包含余数。程序清单 8-2 演示了如何将一个 128 位的量除以一个 32 位的除数，并产生一个 128 位的商和一个 32 位的余数。

程序清单 8-2 无符号 128 位/32 位扩展精度除法操作

```
program testDiv128;
#include( "stdlib.hhf" )

procedure div128
(
    Dividend: lword;
    Divisor:  dword;
    var QuotAdrs: lword;
    var Remainder: dword
); @nodisplay;

const
    Quotient: text := "(type dword=[edi])";

begin div128;
    push( eax );
    push( edx );
    push( edi );
    mov( QuotAdrs, edi );           // Pointer to-quotient storage.
```

² Windows 会将其翻译成 `ex.IntoInstr` 异常。

```

mov( (type dword Dividend[12]), eax ); // Begin division with the H.O. dword.
xor( edx, edx ); // Zero extend into edx.
div( Divisor, edx:eax ); // Divide H.O. dword.
mov( eax, Quotient[12] ); // Store away H.O. dword of quotient.

mov( (type dword Dividend[8]), eax ); // Get dword #2 from the dividend
div( Divisor, edx:eax ); // Continue the division.
mov( eax, Quotient[8] ); // Store away dword #2 of the quotient.

mov( (type dword Dividend[4]), eax ); // Get dword #1 from the dividend.
div( Divisor, edx:eax ); // Continue the division.
mov( eax, Quotient[4] ); // Store away dword #1 of the quotient.

mov( (type dword Dividend[0]), eax ); // Get the L.O. dword of the dividend.
div( Divisor, edx:eax ); // Finish the division.
mov( eax, Quotient[0] ); // Store away the L.O. dword of the quotient.

mov( Remainder, edi ); // Get the pointer to the remainder's value.
mov( edx, [edi] ); // Store away the remainder value.

pop( edi );
pop( edx );
pop( eax );

end div128;

static
op1: lword := $8888_8888_6666_6666_4444_4444_2222_2221;
op2: dword := 2;
quo: lword;
rmndr: dword;

begin testDiv128;

div128( op1, op2, quo, rmndr );

stdout.put
(
  nl
  nl
  "After the division:" nl
  nl
  "Quotient = $",
  quo[12], "_",
  quo[8], "_",
  quo[4], "_",
  quo[0], nl
  "Remainder = ", (type uns32 rmndr )
);

end testDiv128;

```

可以对这段代码进行扩充使其能够处理任意多位，只要向序列中加入其他的 mov/div/mov 指

令即可。像前面一节给出的扩展精度乘法操作一样,这种扩展精度的除法操作只对无符号操作数有用。如果需要将两个有符号量相除,就必须先记录它们的符号,取它们的绝对值,做无符号除法操作,然后根据操作数的符号设置结果的符号。

如果需要使用大于 32 位的除数,那么就需要结合使用移位和减法操作来实现除法操作。但是,这样的算法非常慢。在本节中,我们将开发两个除法算法,这两个算法可以处理任意多的位数。第一个比较慢但易于理解;第二个要快一些(在平均情况下)。

像乘法操作的情况一样,要理解计算机如何进行除法操作,最好的办法就是研究如何手工进行长除(long division)。考虑操作 $3456/12$,以及手工进行这个操作的步骤,如图 8-3 所示。

$\begin{array}{r} 12 \overline{) 3456} \\ \underline{24} \\ 105 \end{array}$ <p>(1) 34 中有 2 个 12</p>	$\begin{array}{r} 2 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \end{array}$ <p>(2) 34 减去 24 再拉下一位得到 105</p>
$\begin{array}{r} 28 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \end{array}$ <p>(3) 105 中有 8 个 12</p>	$\begin{array}{r} 28 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \end{array}$ <p>(4) 105 减去 96 再拉下一位得到 96</p>
$\begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \end{array}$ <p>(5) 96 中正好有 8 个 12</p>	$\begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \end{array}$ <p>(6) 因此, 3456 被 12 整除得到 288</p>

图 8-3 手工逐位执行除法操作

在二进制中,这个算法实际上更简单,因为每一步都不必猜测余数中包含几个 12,也不必将 12 乘以猜测出的值得到供相减的结果。在二进制算法的每一步中,余数恰好包含 0 个或 1 个除数。举个示例来说,考虑除法操作 $27(11011)$ 除以 $3(11)$,如图 8-4 所示。

有一种巧妙的方法可以实现这种二进制除法算法,它可以同时产生商和余数。该算法如下:

```

Quotient := Dividend;
Remainder := 0;
for i := 1 to NumberBits do
    Remainder:Quotient := Remainder:Quotient SHL 1;
    if Remainder >= Divisor then
        Remainder := Remainder - Divisor;
        Quotient := Quotient + 1;
    endif
endfor

```

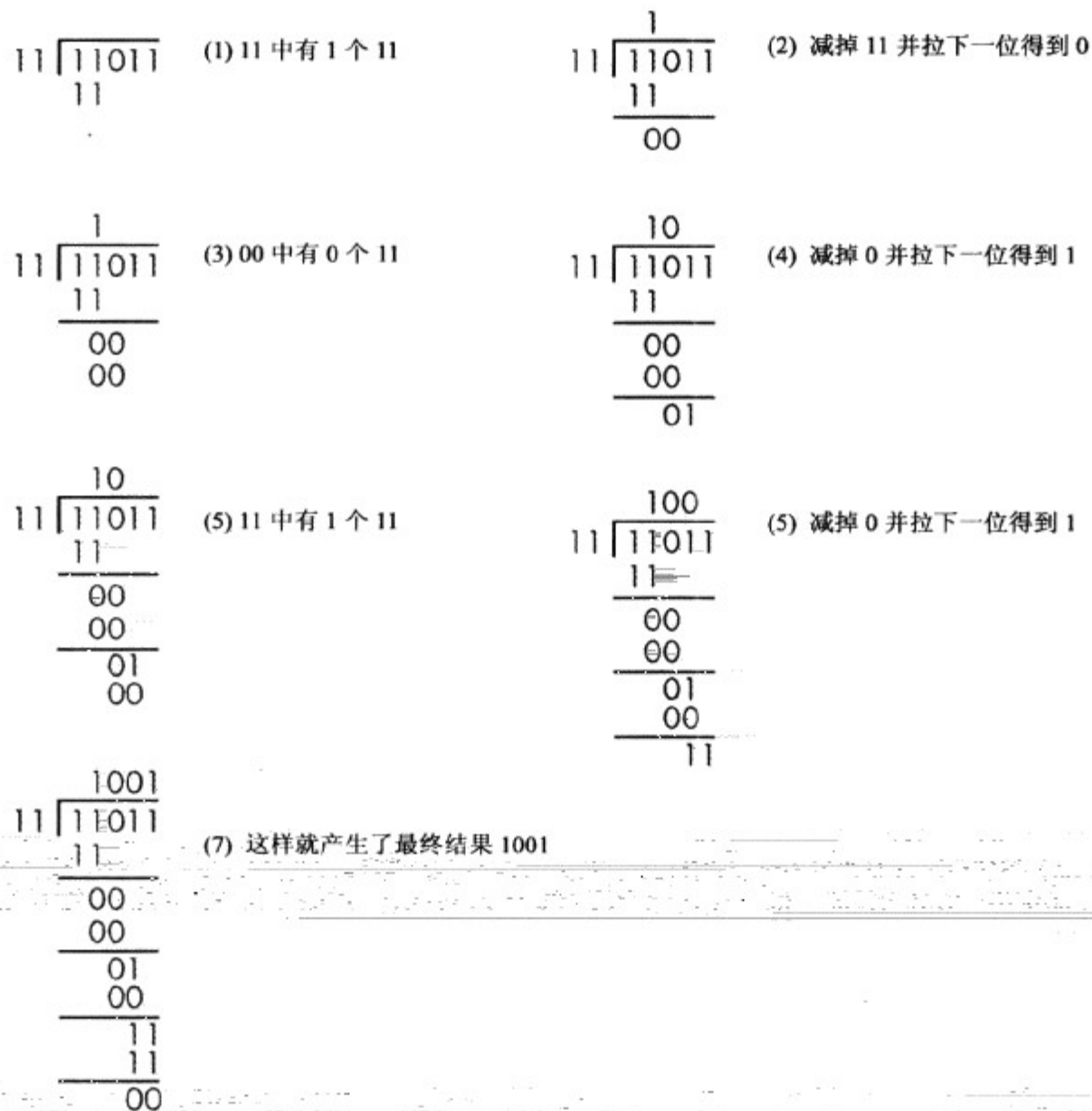



图 8-4 二进制手工长除操作

NumberBits 是 Remainder、Quotient、Divisor 和 Dividend 变量中所包含的位数。注意，Quotient := Quotient + 1; 语句将 Quotient 的最低位设置成 1，因为这个算法先前曾将 Quotient 左移一位。程序清单 8-3 的程序实现了这个算法。

程序清单 8-3 扩展精度除法操作

```
program testDiv128b;
#include( "stdlib.hhf" )

// div128-
//
// This procedure does a general 128/128 division operation using the
// following algorithm (all variables are assumed to be 128-bit objects):
//
// Quotient := Dividend;
// Remainder := 0;
// for i := 1 to NumberBits do
//
// Remainder:Quotient := Remainder:Quotient SHL 1;
// if Remainder >= Divisor then
```

```

//
// Remainder := Remainder - Divisor;
// Quotient := Quotient + 1;
//
// endif
// endfor
//

procedure div128
(
    Dividend:    lword;
    Divisor:     lword;
    var QuotAdrs: lword;
    var RmndrAdrs: lword
); @nodisplay;

const
    Quotient: text := "Dividend"; // Use the Dividend as the Quotient.

var
    Remainder: lword;

begin div128;

    push( eax );
    push( ecx );
    push( edi );

    mov( 0, eax );      // Set the remainder to 0.
    mov( eax, (type dword Remainder[0]) );
    mov( eax, (type dword Remainder[4]) );
    mov( eax, (type dword Remainder[8]) );
    mov( eax, (type dword Remainder[12]) );

    mov( 128, ecx );    // Count off 128 bits in ecx.
    repeat

        // Compute Remainder:Quotient := Remainder:Quotient SHL 1:

        shl( 1, (type dword Dividend[0]) ); // See Section 8.1.12 to see
        rcl( 1, (type dword Dividend[4]) ); // how this code shifts 256
        rcl( 1, (type dword Dividend[8]) ); // bits to the left by 1 bit.
        rcl( 1, (type dword Dividend[12]) );
        rcl( 1, (type dword Remainder[0]) );
        rcl( 1, (type dword Remainder[4]) );
        rcl( 1, (type dword Remainder[8]) );
        rcl( 1, (type dword Remainder[12]) );

        // Do a 128-bit comparison to see if the remainder
        // is greater than or equal to the divisor.
    if
    ( #{
        mov( (type dword Remainder[12]), eax );
        cmp( eax, (type dword Divisor[12]) );

```

```

    ja true;
    jb false;

    mov((type dword Remainder[8]), eax );
    cmp( eax, (type dword Divisor[8]) );
    ja true;
    jb false;

    mov( (type dword Remainder[4]), eax );
    cmp( eax, (type dword Divisor[4]) );
    ja true;
    jb false;

    mov( (type dword Remainder[0]), eax );
    cmp( eax, (type dword Divisor[0]) );
    jb false;
}# ) then

    // Remainder := Remainder - Divisor

    mov( (type dword Divisor[0]), eax );
    sub( eax, (type dword Remainder[0]) );

    mov( (type dword Divisor[4]), eax );
    sub( eax, (type dword Remainder[4]) );

    mov( (type dword Divisor[8]), eax );
    sub( eax, (type dword Remainder[8]) );

    mov( (type dword Divisor[12]), eax );
    sub( eax, (type dword Remainder[12]) );

    // Quotient := Quotient +1;

    add( 1, (type dword Quotient[0]) );
    adc( 0, (type dword Quotient[4]) );
    adc( 0, (type dword Quotient[8]) );
    adc( 0, (type dword Quotient[12]) );

endif;
dec( ecx );

until( @z );

// Okay, copy the quotient (left in the Dividend variable)
// and the remainder to their return locations.

mov( QuotAdrs, edi );
mov((type dword Quotient[0]), eax);
mov( eax, [edi] );
mov((type dword Quotient[4]), eax);
mov( eax, [edi+4] );
mov((type dword Quotient[8]), eax );
mov( eax, [edi+8] );

```



```

    mov((type dword Quotient[12]), eax );
    mov( eax, [edi+12] );

    mov( RmndrAdrs, edi );
    mov((type dword Remainder[0]), eax );
    mov( eax, [edi] );
    mov((type dword Remainder[4]), eax );
    mov( eax, [edi+4] );
    mov((type dword Remainder[8]), eax );
    mov( eax, [edi+8] );
    mov((type dword Remainder[12]), eax );
    mov( eax, [edi+12] );

    pop( edi );
    pop( ecx );
    pop( eax );

end div128;

// Some simple code to test out the division operation:

static
    op1:    lword := $8888_8888_6666_6666_4444_4444_2222_2221;
    op2:    lword := 2;
    quo:    lword;
    rmndr:  lword;

begin testDiv128b;

    div128( op1, op2, quo, rmndr );

    stdout.put
    (
        nl
        nl
        "After the division:" nl
        nl
        "Quotient = $",
        (type dword quo[12]), "_",
        (type dword quo[8]), "_",
        (type dword quo[4]), "_",
        (type dword quo[0]), nl

        "Remainder = ", (type uns32 rmndr )

    );

end testDiv128b;

```

这段代码看上去简单，但其中有几个问题：它没有检测除以 0 的情况(如果试图除以 0，那么将会产生一个\$FFFF_FFFF_FFFF_FFFF 的值)，只能处理无符号值，并且非常慢。处理除以 0 的问题是很简单的：只要在运行代码之前检查除数是否为 0，并在除数为 0 时返回一个适当的错误代码即可(或者产生 `ex.DivisionError` 异常)。处理有符号数值的方法与前面的除法算法相同，先记录

符号,再取操作数的绝对值,做无符号除法操作,然后确定符号。但是,这个算法的性能还是值得商榷的。在 80x86 上,该算法的性能比 `div/div` 指令会差一到两个数量级,而这两条指令已经是 CPU 中最慢的指令了。

有一种技术可以使这种除法操作的性能获得相当程度的提高,即检查除数变量是否只使用 32 位。即使除数是 128 位的变量,其数值本身也常常在 32 位以内(也就是说,Divisor 的高位双字是 0)。在这种特殊却又频繁出现的情况下,使用 `div` 指令往往更快些。这样算法就会稍微复杂一些,因为这样就不得不比较高位双字看其是否为 0,但平均来讲,接下来的将两对数值相除的运行过程要快得多。

8.1.7 扩展精度 `neg` 操作

虽然有好几种方法可以将一个扩展精度的数值取负,但对于较小的数值(96 位或更少)来说,最快捷的方法是使用 `neg` 指令和 `sbb` 指令的组合。该技术利用了 `neg` 指令将用 0 去减它的操作数这一事实。特别地,它设置标志的方法与用 0 减去一个目的值时 `sub` 指令设置标志的方法相同。这段代码采用下面的形式(假设想要将 `EDX:EAX` 中的 64 位数值取负):

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

如果在取负操作的低位字上有借位(这总是会发生的,除非 `EAX` 是 0),那么 `sbb` 指令会将 `EDX` 减 1。

要将这一操作扩展至任意个数的字节、字或双字也很简单;只须从想要取负的对象的高位存储单元开始操作,并依次向低位字节进行。下面的代码计算了一次 128 位的取负操作:

```
static
Value: dword[4];

neg( Value[12] ); // Negate the H.O. double word.
neg( Value[8] ); // Neg previous dword in memory.
sbb( 0, Value[12] ); // Adjust H.O. dword.

neg( Value[4] ); // Negate the second dword in the object.
sbb( 0, Value[8] ); // Adjust third dword in object.
sbb( 0, Value[12] ); // Adjust the H.O. dword.

neg( Value ); // Negate the L.O. dword.
sbb( 0, Value[4] ); // Adjust second dword in object.
sbb( 0, Value[8] ); // Adjust third dword in object.
sbb( 0, Value[12] ); // Adjust the H.O. dword.
```

但是,这段代码显得实在太庞大,并且运行起来很慢,因为在每次取负操作之后都需要在所有高位字上传递进位。要将较大的数值取负,一种简单的方法是用 0 减去要取负的值:

```
static
Value: dword[5]; // 160-bit value.
```

```
mov( 0, eax );
sub( Value, eax );
mov( eax, Value );
```

```
mov( 0, eax );
sbb( Value[4], eax );
mov( eax, Value[4] );
```

```
mov( 0, eax );
sbb( Value[8], eax );
mov( eax, Value[8] );
```

```
mov( 0, eax );
sbb( Value[12], eax );
mov( eax, Value[12] );
```

```
mov( 0, eax );
sbb( Value[16], eax );
mov( eax, Value[16] );
```

8.1.8 扩展精度 and 操作

进行 n 字节的 and 操作非常简单：只要将两个操作数之间对应的字节进行 and 操作，再保存结果即可。例如，要进行一次所有操作数都是 64 位的 and 操作的话，可以使用下面的代码：

```
mov( (type dword source1), eax );
and( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
and( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );
```

这种技术可以简单地扩展到任意位数的情况；只须将操作数中对应的字节、字或双字进行逻辑 and 操作。注意，这一序列将根据最后一次 and 操作的值设置标志。如果最后对高位双字进行 and 操作，就会正确地设置除了零标志以外的所有标志。如果需要在这个序列之后测试零标志，就要对两个结果双字进行逻辑 or 操作(或者分别将二者与 0 作比较)。

8.1.9 扩展精度 or 操作

多字节逻辑 or 操作的进行过程与多字节 and 操作相同。只要简单地对两个操作数中对应的字节进行 or 操作即可。例如，要将两个 96 位值逻辑 or，就使用下面的代码：

```
mov( (type dword source1), eax );
or( (type dword source2), eax );
mov( eax, (type dword dest) );
```



```

mov( (type dword source1[4]), eax );
or( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

mov( (type dword source1[8]), eax );
or( (type dword source2[8]), eax );
mov( eax, (type dword dest[8]) );

```

这段代码没有适当地为整个操作设置零标志。如果想要在多精度 or 操作之后测试零标志,就必须对结果双字进行逻辑 or 操作。

8.1.10 扩展精度 xor 操作

扩展精度 xor 操作的方式与 and/or 相同: 只要对两个操作数中对应的字节进行 xor 操作, 就可以得到扩展精度的结果。下面的代码序列对两个 64 位操作数进行操作, 计算它们的异或值, 并将结果存入一个 64 位变量中:

```

mov( (type dword source1), eax );
xor( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
xor( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

```

在前面两节中关于零标志的论述在这里也适用。

8.1.11 扩展精度 not 操作

not 指令将一个指定操作数的所有位取反。扩展精度的 not 操作只是简单地对所涉及的操作数执行 not 指令。例如, 要对(edx:eax)中的值进行一次 64 位的 not 操作, 就要执行下面的指令:

```

not( eax );
not( edx );

```

要记住, 如果两次执行 not 指令的话, 结果就是原来的值。还要注意, 将一个值与全 1(\$FF、\$FFFF 或 \$FFFF_FFFF)进行异或操作的话, 那么实际进行的操作和 not 指令是一样的。

8.1.12 扩展精度移位操作

扩展精度移位操作需要一条移位指令和一条循环移位指令。考虑一下使用 32 位操作来实现 64 位 shl 操作必须要做的工作(见图 8-5):

- (1) 第 0 位中必须移入一个 0。
- (2) 第 0~30 位被向前移到较高一位中。
- (3) 第 31 位被移到第 32 位中。
- (4) 第 32~62 位必须被向前移到较高一位中。
- (5) 第 63 位被移到进位标志中。

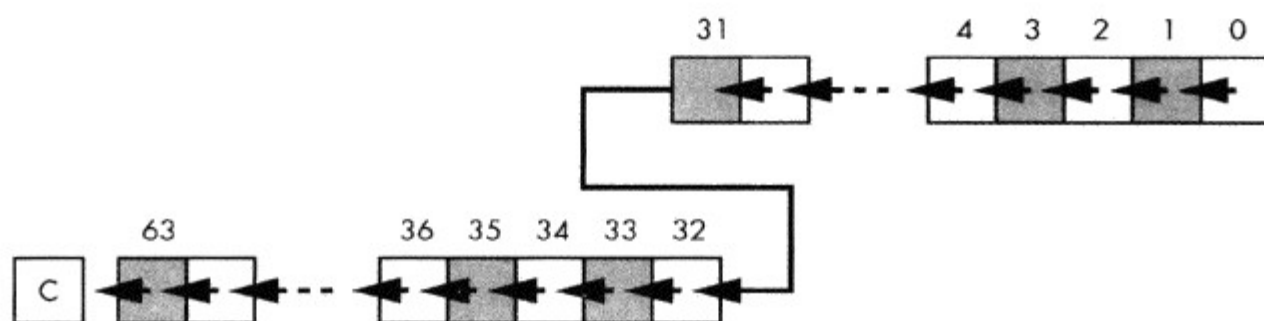


图 8-5 64 位左移操作

用于实现这种 64 位移位操作的指令分别是 `shl` 和 `rcl`。例如，要将一个(EDX:EAX)中的 64 位操作数左移一位的话，就可以使用这些指令：

```
shl( 1, eax );
rcl( 1, eax );
```

注意，对于扩展精度的数值，使用这种技术一次只能移动一位。不能使用 CL 寄存器将一个扩展精度操作数一次移动若干位，也不能指定一个大于 1 的常量值。

为了理解该指令序列的工作过程，请考虑这些指令各自的基本操作。`shl` 指令将 0 移入 64 位操作数的第 0 位，并将第 31 位移入进位标志。然后 `rcl` 指令将进位标志中的内容移入第 32 位，再将第 63 位移入进位标志。最后的结果正是我们所需要的。

要对一个大于 64 位的操作数进行左移操作，只要另外再使用一条 `rcl` 指令即可。一次扩展精度左移操作总是从最低有效双字开始，接下来的每一个 `rcl` 指令都对下一个有效双字进行操作。例如，要在某个存储单元上进行一次 96 位左移操作，就可以使用下面的指令：

```
shl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4]) );
rcl( 1, (type dword Operand[8]) );
```

如果需要每次移动两位或更多位的数据，可以将前述的指令序列重复所需要的次数(常量次数的移位)，或者也可以将这些指令放在一个循环中，让它们重复执行若干次。例如，下面的代码将 96 位操作数 *Operand* 左移由 ECX 指定的位数：

```
ShiftLoop:
    shl( 1, (type dword Operand[0]) );
    rcl( 1, (type dword Operand[4]) );
    rcl( 1, (type dword Operand[8]) );
    dec( ecx );
    jnz ShiftLoop;
```

可以以类似的方法实现 `shr` 和 `sar`，不同的是操作必须从操作数的高位字开始，依次向下进行，直至低位字：

```
// Extended-precision SAR:

sar( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );
```

```
// Double-precision SHR:
```

```
shr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );
```

这里所说的扩展精度移位操作和它所对应的 8/16/32 位版本相比有一个明显的不同之处：扩展精度的移位操作设置标志的方法与单精度操作的设置有区别。这是因为循环移位指令与移位指令二者影响标志的方法不同。所幸的是，在移位操作之后，进位标志是最经常被测试的，而扩展精度移位操作(也就是循环移位指令)的过程中，会对这个标志进行适当的设置。

shld 和 shrd 指令可以有效地实现多精度多位移位指令。这些指令具有如下的语法：

```
shld( constant, Operand1, Operand2 );
shld( cl, Operand1, Operand2 );
shrd( constant, Operand1, Operand2 );
shrd( cl, Operand1, Operand2 );
```

shld 指令的工作过程如图 8-6 所示：

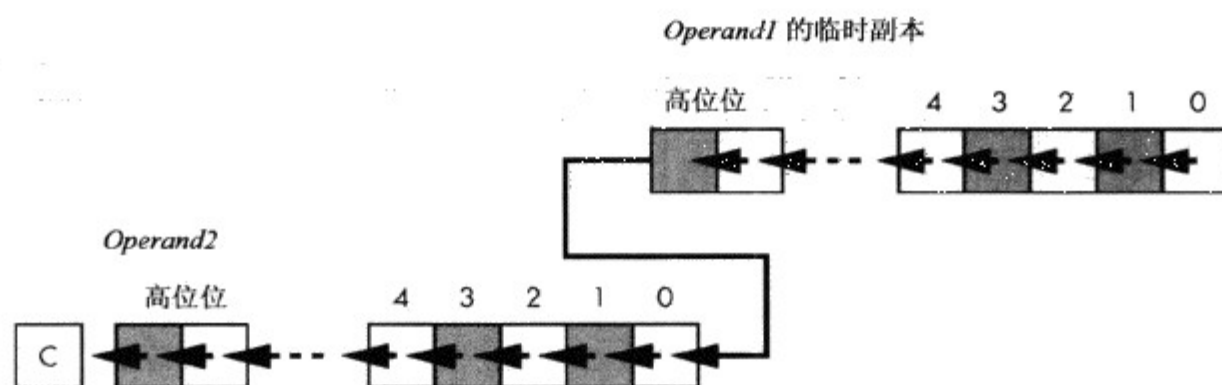


图 8-6 shld 操作

Operand1 必须是一个 16 位或 32 位寄存器，Operand2 可以是一个寄存器，也可以是一个存储单元。两个操作数的长度必须相同。立即操作数可以是 $0 \sim n-1$ 之间的任何一个值，其中 n 是两个操作数的位数；该操作数指出所要移动的位数。

shld 指令将 Operand2 中的若干位左移。其中高位位被移入进位标志，而 Operand1 中的高位位被移入 Operand2 的低位位。注意，这条指令不会修改 Operand1 的值；在移位过程中它使用的是 Operand1 的一个临时副本。立即操作数指出了要移动的位数，如果这个数字为 n ，那么 shld 会将第 $n-1$ 位移入进位标志。并且将 Operand1 中的高 n 位移入 Operand2 的低 n 位。shld 指令将以下面的方法设置标志位：

- (1) 如果要移动的位数是 0，那么 shld 指令将不影响任何标志。
- (2) 进位标志包含从 Operand2 的高位位中移出的最后一位。
- (3) 如果要移动的位数为 1，那么当 Operand2 的符号位在移动过程中发生变化时，溢出标志的值是 1。如果要移动的位数不为 1，那么溢出标志将是未定义的。
- (4) 如果移位操作产生的结果为 0，则零标志将为 1。
- (5) 符号标志将包含结果的高位位。

shrd 指令与 shld 指令类似，当然，除了是向右移动而不是向左移动以外。要想了解 shrd 指令的工作过程，请参见图 8-7。

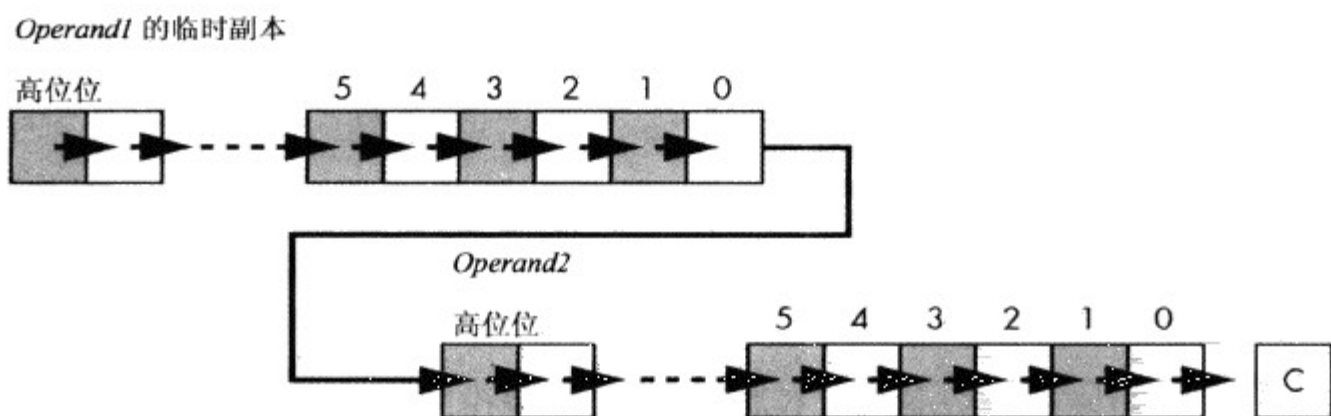


图 8-7 shrd 操作

`shrd` 指令将以下面的方法设置标志位:

- (1) 如果要移动的位数是 0, 那么 `shrd` 指令将不影响任何标志。
- (2) 进位标志包含从 `Operand2` 的低位位中移出的最后一位。
- (3) 如果要移动的位数为 1, 那么当 `Operand2` 的高位位在移动过程中发生变化时, 溢出标志将包含 1。如果要移动的位数不为 1, 那么溢出标志将是未定义的。
- (4) 如果移位操作产生的结果为 0, 则零标志将为 1。
- (5) 符号标志将包含结果的高位位。

考虑下面的代码序列:

```
static
ShiftMe: dword[3] := [ $1234, $5678, $9012 ];

mov( ShiftMe[4], eax )
shld( 6, eax, ShiftMe[8] );
mov( ShiftMe[0], eax );
shld( 6, eax, ShiftMe[4] );
shl( 6, ShiftMe[0] );
```

上述的第一条 `shld` 指令将若干位从 `ShiftMe[4]` 移至 `ShiftMe[8]`, 而且不影响 `ShiftMe[4]` 中的值。第二条 `shld` 指令将若干位从 `ShiftMe` 移至 `ShiftMe[4]`。最后 `shl` 指令将低位双字移动适当的位数。对于这一段代码, 有两个重要之处。首先, 和其他扩展精度左移操作不同, 这个序列从高位双字向低位双字进行操作。其次, 进位标志不包含高位移位操作所产生的进位。如果需要在这一点上保留进位标志, 就要在第一条 `shld` 指令之后将标志压入栈, 并在 `shl` 指令之后将其从栈中弹出。

可以使用 `shrd` 指令来做扩展精度右移操作, 它和前面代码的工作方式几乎相同。区别在于, 右移操作是从低位双字向高位双字进行的。这个问题留给读者作为练习。

8.1.13 扩展精度循环移位操作

`rcl` 和 `rcr` 操作以一种与 `shl` 和 `shr` 相同的方式进行扩展。例如, 要进行 96 位 `rcl` 和 `rcr` 操作的话, 可以使用下面的指令:

```
rcl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4]) );
```

```

rcl( 1, (type_dword Operand[8]) );
rcr( 1, (type_dword Operand[8]) );
rcr( 1, (type_dword Operand[4]) );
rcr( 1, (type_dword Operand[0]) );

```

这一段代码与扩展精度移位操作的代码唯一不同在于, 这段代码中的第一条指令是 `rcl` 或 `rcr`, 而不是 `shl` 或 `shr` 指令。

要实现一条扩展精度 `rol` 或 `ror` 指令不是一个简单的操作, 我们使用 `bt`、`shld` 和 `shrd` 指令来实现。下面的代码显示了如何使用 `shld` 指令来完成一次扩展精度的 `rol`:

```

// Compute rol( 4, edx:eax );
mov( edx, ebx );
shld( 4, eax, edx );
shld( 4, ebx, eax );
bt( 0, eax ); // Set carry flag, if desired.

```

扩展精度 `ror` 指令与之类似; 只是要记住, 应该先操作对象的低位端, 最后操作高位端。

8.1.14 扩展精度 I/O

只要能够使用扩展精度算术操作进行计算, 那么接下来的问题就是如何在程序中获得那些扩展精度的数值, 以及如何将扩展精度的数值显示给用户。HLA 的标准库提供了一些无符号十进制、有符号十进制和十六进制 I/O 操作的例程, 可以对长度为 8 位、16 位、32 位、64 位或 128 位的数值进行 I/O 操作。所以只要所操作的是长度小于或等于 128 位的数值, 就可以使用标准库的代码。如果需要输入或输出长度大于 128 位的数值, 就需要自己编写过程来处理这些操作。这一节将讨论编写这样的例程所需要的一些策略。

本节中的示例是特别针对 128 位数值的。它们的算法极具通用性, 可以扩展至任意位数的情况(实际上, 本节中 128 位的算法只不过是 HLA 标准库针对 128 位数值所使用的算法)。当然, 如果需要一组 128 位无符号 I/O 例程, 那可以使用标准库的代码, 而无须修改。如果需要处理更大的数值, 那么只须简单地修改下面的代码。

后面几节为了避免在每条指令中强制转换 `lword/uns128/int128` 类型的数值, 将使用一组通用的 128 位数据类型, 如下所示:

type	
hl28	:dword[4];
ul28	:dword[4];
il28	:dword[4];

1. 扩展精度十六进制输出

扩展精度的十六进制输出很简单。只要通过调用 `stdout.puth32` 例程, 按照从高位双字到低位双字的顺序, 输出扩展精度数值的每一个双字部分即可。下面的过程正是这样完成一个 `lword` 值的输出:


```

procedure puth128 (b128:h128); @nodisplay;
begin puth128;

    stdout.puth32(-b128[12]);
    stdout.puth32( b128[8] );
    stdout.puth32( b128[4] );
    stdout.puth32( b128[0] );

end puth128;

```

当然，HLA 标准库提供了一个 `stdout.puth128` 过程，可以直接写 `lword` 值，所以输出较大的数值(比如说，一个 256 位的值)时，也可以多次调用 `stdout.puth128`。HLA `stdlib.puth128` 例程的实现方式与上面的 `puth128` 类似。

2. 扩展精度无符号十进制输出

十进制输出比十六进制输出要稍微复杂一些，因为二进制数字的高位位影响其十进制表示的低位数位(对于十六进制值来说不是这样，这就是为什么十六进制输出如此简单的原因)。因此，我们将不得不通过每次从二进制数字中提取一位十进制数字的方法，为一个二进制数字建立十进制表示。

对于无符号十进制输出，最通用的解决方案是将输出值连续除以 10，直到结果变成 0。第一次除后的余数是一个范围在 0~9 之间的数值，这个数值对应着十进制数的低位数位。连续除以 10(和它们对应的余数)可以从整个数字中提取出连续的数位来。

对于这个问题，迭代的解决方案通常是，分配一些存储空间给一个由很多字符构成的字符串，而这个字符串足够容纳整个数字。然后编写代码通过一个循环将所有的十进制位提取出来，并把它们顺次放在字符串当中。在转换过程的最后，例程将逆向打印字符串中的字符(记住，除法算法首先提取低位数位，最后提取高位数位，这正好与它们需要被打印出来的顺序相反)。

在本节中，我们将利用递归的解决方案，因为这样更好一些。这个递归解决方案首先将数值除以 10，并将余数存储在一个局部变量中。如果商不等于 0，那么例程就递归调用自身，首先打印所有前导位。递归调用返回时(这时所有前导位都已经被打印)，递归算法将打印余数的相关位来完成操作。这里给出了打印十进制值 789 时的操作步骤：

- (1) 将 789 除以 10。商是 78，余数是 9。
- (2) 将余数(9)存入一个局部变量中，对商递归调用例程。
- (3) [递归入口 1]将 78 除以 10。商是 7，余数是 8。
- (4) 将余数(8)存入一个局部变量中，对商递归调用例程。
- (5) [递归入口 2]将 7 除以 10。商是 0，余数是 7。
- (6) 将余数(7)存入一个局部变量中。因为商是 0，所以不需要再递归调用例程了。
- (7) 输出局部变量中存储的余数值(7)，返回调用者(递归入口 1)。
- (8) [返回递归入口 1] 输出递归入口 1 局部变量中存储的余数值(8)，返回调用者(过程起始调用处)。
- (9) [起始调用处] 输出起始调用处局部变量中存储的余数值(9)，返回输出例程的起始调用者。

在整个算法中，唯一需要扩展精度计算的地方就是“除以 10”的语句，其他的操作都很简单。就这个算法来讲，我们是幸运的；因为这里使用了一个范围在一个双字之内的值去除一个扩展精

度的数值,所以我们可以使用快速(并且简单)的扩展精度除法算法,而该算法使用了 `div` 指令。程序清单 8-4 中的程序实现了一个使用这种技术的 128 位十进制输出例程。

程序清单 8-4 128 位扩展精度十进制输出例程

```
program out128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    ul28: dword[4];

// DivideBy10-
//
// Divides "divisor" by 10 using fast
// extended-precision division algorithm
// that employs the div instruction.
//
// Returns quotient in "quotient"
// Returns remainder in eax.
// Trashes ebx,edx,and edi.

procedure DivideBy10( dividend:ul28; var quotient:ul28 ); @nodisplay;
begin DivideBy10;

    mov( quotient, edi );
    xor( edx, edx );
    mov( dividend[12], eax );
    mov( 10, ebx );
    div( ebx, edx:eax );
    mov( eax, [edi+12] );

    mov( dividend[8], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+8] );
    mov( dividend[4], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+4] );

    mov( dividend[0], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+0] );
    mov( edx, eax );

end DivideBy10;

// Recursive version of putul28.
// A separate "shell" procedure calls this so that
// this code does not have to preserve all the registers
// it uses (and DivideBy10 uses) on each recursive call.
```

```

procedure recursivePutul28( b128:u128 ); @nodisplay;
var
    remainder: byte;

begin recursivePutul28;

    // Divide by 10 and get the remainder (the char to print).
    DivideBy10( b128, b128 );
    mov( al, remainder );    // Save away the remainder (0..9).

    // If the quotient (left in b128) is not 0, recursively
    // call this routine to print the H.O. digits.
    mov( b128[0], eax );    // If we logically OR all the dwords
    or( b128[4], eax );      // together, the result is 0 if and
    or( b128[8], eax );      // only if the entire number is 0.
    or( b128[12], eax );
    if( -@nz ) then
        recursivePutul28( b128 );

    endif

    // Okay, now print the current digit.

    mov( remainder, al );
    or( '0', al );          // Converts 0..9 -> '0'..'9'.
    stdout.putc( al );

end recursivePutul28;

// Non-recursive shell to the above routine so we don't bother
// saving all the registers on each recursive call.

procedure putul28( b128:uns128 ); @nodisplay;
begin putul28;

    push( eax );
    push( ebx );
    push( edx );
    push( edi );

    recursivePutul28( b128 );

    pop( edi );
    pop( edx );
    pop( ebx );
    pop( eax );

end putul28;

// Code to test the routines above:
static
    b0: u128 := [0, 0, 0, 0];    // decimal = 0

```

```

b1: u128 := [1234567890, 0, 0, 0]; // decimal = 1234567890
b2: u128 := [$8000_0000, 0, 0, 0]; // decimal = 2147483648
b3: u128 := [0, 1, 0, 0]; // decimal = 4294967296

// Largest uns128 value
// (decimal=340,282,366,920,938,463,463,374,607,431,768,211,455);

b4: u128 := [$FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF];

begin out128;

  stdout.put( "b0 = " );
  putu128( b0 );
  stdout.newln();

  stdout.put( "b1 = " );
  putu128( b1 );
  stdout.newln();

  stdout.put( "b2 = " );
  putu128( b2 );
  stdout.newln();

  stdout.put( "b3 = " );
  putu128( b3 );
  stdout.newln();

  stdout.put( "b4 = " );
  putu128( b4 );
  stdout.newln();

end out128;

```

3. 扩展精度有符号十进制输出

只要有了扩展精度无符号十进制输出例程，那么编写一个扩展精度有符号十进制输出例程将是非常简单的。基本的算法采取下面的形式：

(1) 检查数字的符号。

(2) 如果是正号，就调用无符号输出例程输出。如果是负号，就先打印一个减号，然后数字取反，再调用无符号输出例程输出。

要检查扩展精度整数的符号，当然只要简单地测试该数字的高位位即可。而要将一个较大的数值取负，那么最好的解决方法通常是用0去减这个值。这里是 puti128 的快速版本，它使用了前面一节中提到的 putu128 例程。

```

procedure puti128( i128: int128 ); nodisplay;
begin puti128;

  if( (type int32 i128[I2]) < 0 ) then

    stdout.put( '-' );

    // Extended-Precision Negation:

```



```

    push( eax );
    mov( 0, eax );
    sub( i128[0], eax );
    mov( eax, i128[0] );

    mov( 0, eax );
    sbb( i128[4], eax );
    mov( eax, i128[4] );

    mov( 0, eax );
    sbb( i128[8], eax );
    mov( eax, i128[8] );

    mov( 0, eax );
    sbb( i128[12], eax );
    mov( eax, i128[12] );
    pop( eax );

    endEf;
    putu128( i128 );
end puti128;

```

4. 扩展精度格式化输出

前面两节中的代码使用最少量的打印位置来打印有符号和无符号整数。要创建具有良好格式的数值表格，就需要 `puti128Size` 或 `putu128Size` 这样的例程。只要有了这些例程的“非格式化”版本，那么实现它们的格式化版本就很简单了。

第一步是编写一个 `i128Size` 例程和 `u128Size` 例程，来计算显示数值所需要的最少位数。实现它的算法与数值输出例程非常类似。实际上，唯一的不同在于，在进入例程(比如说，非递归 `shell` 例程)之前，要将一个计数器初始化为 0，并在每次递归调用时增加这个计数器的值，而不是输出一位数字(别忘了，如果数字是负数，就要在 `i128Size` 内增加计数器的值；必须考虑输出减号)。计算完成后，这些例程将在 `EAX` 寄存器中返回操作数的长度。

只要有了 `i128Size` 和 `u128Size` 例程，那么编写格式化的输出例程就非常简单了。在 `puti128Size` 或 `putu128Size` 最初的入口处，这些例程将调用相应的 `size` 例程，为要显示的数字确定打印位置的数量。如果 `size` 例程返回的值大于最小长度参数(被传递给 `puti128Size` 或 `putu128Size`)的绝对值，那么所有要做的就是调用 `put` 例程来打印数值，而不再需要其他的格式化工作。如果参数长度的绝对值大于 `i128Size` 或 `u128Size` 返回的值，那么程序就必须计算这两个值的差，并在打印数字之前(如果参数的长度值是正数)或打印数字之后(如果参数的长度值是负数)打印相应数量的空格符(或者其他填充符)。这两个例程的实现留作练习(或者查看 HLA 标准库中 `stdout.putiSize128` 或 `stdout.putuSize128` 例程的源代码)。

HLA 标准库通过一系列连续的扩展精度比较操作来确定数值的位数，从而实现了 `i128Size` 和 `u128Size`。有兴趣的读者可以了解这些例程的源代码，以及 `stdout.puti128` 或 `stdout.putu128` 过程的源代码(这些源代码可以从 <http://webster.cs.ucr.edu/> 或 <http://www.artofasm.com/> 下载)。

5. 扩展精度输入例程

在扩展精度输出例程和扩展精度输入例程之间存在一些基本区别。首先,数值输出一般没有发生错误的可能性³;而另一方面,数值输入必须要处理实际可能出现的输入错误,诸如非法字符以及数值溢出等。同样,HLA 标准库和运行库系统也提供了一种不同的输入转换方案。本节将讨论输入转换与输出转换的区别。

输入转换和输出转换之间最大的区别可能就是,输出转换是一种不与上下文同时进行的转换。也就是说,在将一个数值转换成一个由字符构成的字符串以便输出时,输出例程将不关心输出字符串之前的字符,也不关心输出流中数值后面的字符。数值输出例程将它们的数据转换成字符串并打印该字符串,并不考虑上下文(也就是说,位于数值字符串表示前后的字符)。数值输出例程的要求不能如此宽松,数值字符串周围的上下文信息很重要。

一次典型的数值输出操作包括,从用户那里读取一个由字符构成的字符串,然后将这个字符串翻译成一种内部的数值表示。例如,一个像 `stdin.get(i32);` 这样的语句通常会从用户那里读取一行文本,并把这行文本开始处的一个数字序列转换成一个 32 位有符号整数(假设 `i32` 是一个 `int32` 的对象)。但是要注意, `stdin.get` 例程会跳过字符串中可能出现在实际的数值字符之前的字符。例如, `stdin.get` 会自动跳过字符串中的任何前导空格符。同样,输入字符串中在数值部分之后还可能包含其他的数据(例如,有可能要从同一输入行中读取两个整数值),因此输入转换例程必须能够确定在输入流中数值数据将在哪里结束。所幸的是,HLA 提供了一个简单的机制,可以确定输入数据的开始和结束:这就是 `Delimiters` 字符集。

`Delimiters` 字符集是一个变量,是 HLA 标准库的内部变量,它包含一组能够位于一个合法数值之前或之后的合法字符。默认情况下,这个字符集包括字符串结束标记(一个 0 字节)、制表符、行结束符、回车符、空格符、逗号、冒号和分号。因此,HLA 的数值输入例程可以自动忽略出现在输入的数值字符串之前的这个集合中的任何字符。同样,这个集合中的字符可以合法地跟在一个输入的数值字符串之后(相反地,如果有任何非分隔符跟在数值字符串之后,HLA 就会产生一个 `ex.ConversionError` 异常)。

`Delimiters` 字符集是 HLA 标准库内部的私有变量。虽然我们不能直接访问这些对象,但 HLA 标准库提供了两个存取函数, `conv.setDelimiters` 和 `conv.getDelimiters`, 可以用于访问和修改这个字符集中的值。这两个函数的原型(位于 `conv.hhf` 头文件中)如下:

```
procedure conv.setDelimiters( Delims:cset );
procedure conv.getDelimiters( var Delims:cset );
```

`conv.setDelimiters` 过程将 `Delims` 参数的值复制到内部 `Delimiters` 字符集中。因此,如果想在数值输入中使用一组不同的分隔符,就可以使用这个过程来修改字符集。`conv.getDelimiters` 调用将返回内部 `Delimiters` 字符集的一个副本,将其存储在一个作为参数传递给 `conv.getDelimiters` 过程的变量中。在编写我们自己的扩展精度数值输出例程时,可以使用 `conv.getDelimiters` 的返回值来确定数值输入的开始。

从用户处读取一个数值时,第一步是得到 `Delimiters` 字符集的一个副本。第二步是从用户处

³ 从技术上来说,这并不完全正确,因为某个设备出现错误(比如说,磁盘满)也是有可能的。只不过,这样的可能性很低,以至于我们可以忽略这种可能性。

读取输入字符，并且如果输入的字符是 `Delimiters` 字符集的成员，就将该字符丢弃。只要发现某个字符不在 `Delimiters` 字符集中，输入例程就必须检查这个字符，验证它是不是一个合法的数值字符。如果不是，那么当字符的值超出 `$00~$7f` 的范围时，程序应该产生一个 `ex.IllegalChar` 异常；如果这个字符不是一个合法的数值字符，应该产生 `ex.ConversionError` 异常。一旦例程遇到一个数值字符，那么只要是合法的数值字符，就应该继续读取它们；在读取字符的时候，转换例程应该将它们翻译成数值数据的内部表示。如果在转换过程中出现了溢出，程序就应该产生 `ex.ValueOutOfRange` 异常。

当过程在数值字符串的最后遇到第一个分隔符的时候，数值表示的转换就应该结束了。但是，有一点非常重要，就是该过程并不读入用于结束字符串的分隔符。所以下面的代码是不正确的：

```
static
    Delimiters: cset;

    .
    .
    .

conv.getDelimiters( Delimiters );

// Skip over Feading delimiters in the string:

while( stdin.getc() in Delimiters ) do /* getc did the work */ endwhile;
while( al in '0'..'9' ) do

    // Convert character in al to numeric representation and
    // accumulate result...

    stdin.getc();

endwhile;
if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;
```

第一个 `while` 循环读取了一串分隔符。当这个 `while` 循环结束时，`AL` 中的字符并不是一个分隔符。第二个 `while` 循环处理了一串十进制数字。首先它检查前一个 `while` 循环中读入的字符，看其是否是一个十进制数字；如果是，就处理这个数字并读取下一个字符。这个过程一直继续，直到调用 `stdin.getc`(在循环的底部)读取了一个非数字字符。在第二个 `while` 循环之后，程序检查最后读入的字符，确保它是用于数值输入的一个合法的分隔符。

这个算法的问题在于，它在数值字符串后还使用了分隔符。例如，在默认的 `Delimiters` 字符集中，冒号是一个合法的分隔符。如果用户从键盘输入的是 `123:456`，然后执行上面的代码，那么这段代码通常就会将 `123` 转换成数值 `123`。但是，紧接着下一个从输入流中读取的字符应该是 `4`，而不是冒号(`:`)。这一点虽然在某些情况下是可以接受的，但大多数程序员希望数值输入例程只读入那些前导的分隔符和数值字符，而不希望输入例程读入任何尾随的分隔符(比如说，如果给定字符串 `123:456`，那么很多程序会读取下一个字符并且希望输入的是一个冒号)。因为 `stdin.getc` 会读入一个输入字符，并且没有办法将字符放回输入流中。所以，就要有一些其他的方法来读取用户

的输入字符, 这些方法将不读入那些字符⁴。

HLA 标准库提供了一种解决方法, 即 `stdin.peekc` 函数。像 `stdin.getc` 一样, `stdin.peekc` 例程将从 HLA 内部缓冲区中读取下一个输入的字符。`stdin.peekc` 和 `stdin.getc` 之间有两个主要的不同之处。首先, 如果当前的输入行是空行(或者已经从输入行中读取了所有的文本), 那么 `stdin.peekc` 不会强制用户输入一行新的文本。取而代之的是, `stdin.peekc` 简单地在 AL 寄存器中返回 0, 以表示输入行中不再有更多的字符。因为 #0(NUL 字符)对于数值来说(默认地)是一个合法的分隔符, 并且行结束符也是一种中止数值输入的合法方法, 所以这种机制运行得很好。`stdin.getc` 和 `stdin.peekc` 的第二个不同之处是, `stdin.peekc` 不会读入从输入缓冲区中读取的字符。如果在一行中几次调用 `stdin.peekc`, 那么就总是会返回相同的字符; 同样, 如果在 `stdin.peekc` 之后立即调用 `stdin.getc`, 那么 `stdin.getc` 调用返回的字符一般情况下就会与 `stdin.peekc` 返回的字符相同(唯一的例外就是行结束符)。所以, 虽然我们在使用 `stdin.getc` 读取了字符之后, 不能将它们放回输入流中, 但是我们可以前瞻输入流的下一个字符, 并使处理逻辑基于该字符。前面算法的修正版本如下所示:

```
static
Delimiters:cset;

conv.getDelimiters( Delimiters );

// Skip over leading delimiters in the string:
while( stdin.peekc() in Delimiters ) do
    // If at the end of the input buffer, we must explicitly read a
    // new line of text from the user. stdin.peekc does not do this
    // for us.
    if( al = #0 ) then
        stdin.ReadLn();
    else
        stdin.getc(); // Remove delimiter from the input stream.
    endif;
endwhile;
while( stdin.peekc in '0'..'9' ) do
    stdin.getc(); // Remove the input character from the input stream.
    // Convert character in al to numeric representation and
    // accumulate result...
```

⁴ HLA 标准库例程实际上会在一个字符串中缓冲输入行, 并处理字符串中的字符。这样在寻找一个分隔符来结束输入值的时候, 就易于“前瞻”一个字符。您的代码也可以做到这点; 但是本章中的代码将使用另外一种不同的方案。

```

endwhile;
if( al not in Delimiters )then
    raise( ex.ConversionError );
endif;

```

注意，第二个 while 循环中对 `stdin.peekc` 的调用在表达式的值为假时，没有读入分隔符。这样，在算法结束时，分隔符就会是下一个要读取的字符。

现在关于数值输入只剩下一点需要解释，那就是要指出，HLA 标准库输入例程允许在数值字符串中出现任意多的下划线，而输入例程会忽略这些下划线字符。这就允许用户输入像 `FFFF_F012` 或 `1_023_596` 这样的字符，它们要比 `FFFFFF012` 和 `1023596` 更易读一些。要在数值输入例程中允许下划线(或任选的其他符号)出现是很简单的：只要像下面这样修改第二个 while 循环即可：

```

while( stdin.peekc in {'0'..'9', '_'}) do
    stdin.getc(); // Read the character from the input stream.

    // Ignore underscores while processing numeric input.

    if( al<> '_' )then
        // Convert character in al to numeric representation and
        // accumulate result...

    endif;

endwhile;

```

6. 扩展精度十六进制输入

和数值输出的情况一样，十六进制输入是最容易编写的数值输入例程。对于十六进制字符串的数值转换来说，基本的算法如下：

- (1) 将扩展精度值初始化为 0。
- (2) 对于每一个输入字符，如果它是合法的十六进制数字，就进行如下的操作。
 - a. 将十六进制字符转换为 0~15(\$0~\$F)之间的一个值。
 - b. 如果扩展精度值的高 4 位非 0，就产生一个异常。
 - c. 将当前的扩展精度值乘以 16(也就是，左移 4 位)。
 - d. 将转换后的十六进制值加到累加器中。
 - e. 检查最后输入的字符，确保它是一个合法的分隔符。如果不是就产生一个异常。

程序清单 8-5 中的程序实现了这个 128 位值的扩展精度十六进制输入例程。

程序清单 8-5 扩展精度十六进制输入

```

program Xin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    b128: dword[4];

```



```

procedure getb128( var inValue:b128 ); @node$play;
const
  HexChars := {'0'..'9', 'a'..'f', 'A'..'F', '_'};
var
  Delimiters: cset;
  LocalValue: b128;

begin getb128;

  push( eax );
  push( ebx );

  // Get a copy of the HLA standard numeric input delimiters:

  conv.getDelimiters( Delimiters );

  // Initialize the numeric input value to 0:

  xor( eax, eax );
  mov( eax, LocalValue[0] );
  mov( eax, LocalValue[4] );
  mov( eax, LocalValue[8] );
  mov( eax, LocalValue[12] );

  // By default, #0 is a member of the HLA Delimiters
  // character set. However, someone may have called
  // conv.setDelimiters and removed this character
  // from the internal Delimiters character set. This
  // algorithm depends upon #0 being in the Delimiters
  // character set, so let's add that character in
  // at this point just to be sure.

  cs.unionChar( #0, Delimiters );

  // If we're at the end of the current input
  // line (or the program has yet to read any input),
  // for the input of an actual character.

  if( stdin.peekc() = #0 ) then

    stdin.readLn();

  endif;

  // Skip the delimiters found on input. This code is
  // somewhat convoluted because stdin.peekc does not
  // force the input of a new line of text if the current
  // input buffer is empty. We have to force that input
  // ourselves in the event the input buffer is empty.

  while( stdin.peekc() in Delimiters ) do

    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the

```



```

// delimiter character from the input stream.
if( al = #0 ) then
    stdin.readLn(); // Force a new input line.
else
    stdin.getc(); // Remove the delimiter from the input buffer.
endif;
endwhile;

// Read the hexadecimal input characters and convert
// them to the internal representation:
while( stdin.peekc() in HexChars ) do
    // Actually read the character to remove it from the
    // input buffer.
    stdin.getc();

    // Ignore underscores, process everything else.
    if( al <> '_' ) then
        if( al in '0'..'9' ) then
            and( $f, al ); // '0'..'9' -> 0..9
        else
            and( $f, al ); // 'a'/'A'..'f'/'F' -> 1..6
            add( 9, al ); // 1..6 -> 10..15
        endif;

        // Conversion algorithm is the following:
        //
        // (1) LocalValue := LocalValue * 16.
        // (2) LocalValue := LocalValue + al.
        //
        // Note that "* 16" is easily accomplished by
        // shifting LocalValue to the left 4 bits.
        //
        // Overflow occurs if the H.O. 4-bits of LocalValue
        // contain a nonzero value prior to this operation.
        //
        // First, check for overflow:
        test( $F0, ( type byte LocalValue[15] ) );
        if( @nz ) then
            raise( ex.ValueOutOfRange );
        endif;
    end;
endwhile;

```

```

// Now multiply LocalValue by 16 and add in
// the current hexadecimal digit (in eax).

mov( LocalValue[8], ebx );
shld( 4, ebx, LocalValue[12] );
mov( LocalValue[4], ebx );
shld( 4, ebx, LocalValue[8] );
mov( LocalValue[0], ebx );
shld( 4, ebx, LocalValue[4] );
shl( 4, ebx );
add( eax, ebx );
mov( ebx, LocalValue[0] );

endif;

endwhile;

// Okay, we've encountered a non-hexadecimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.
if( al not in Delimiters ) then
    raise( ex.ConversionError );
endif;

// Okay, this conversion has been a success. Let's store
// away the converted value into the output parameter.

mov( inValue, ebx );
mov( LocalValue[0], eax );
mov( eax, [ebx] );

mov( LocalValue[4], eax );
mov( eax, [ebx+4] );

mov( LocalValue[8], eax );
mov( eax, [ebx+8] );

mov( LocalValue[12], eax );
mov( eax, [ebx+12] );
pop( ebx );
pop( eax );

end getb128;

// Code to test the routines above:

static
    bl:b128;

begin Xin128;
    stdout.put( "Input a 128-bit hexadecimal value:" );

```

```

getb128( b1 );
stdout.put
(
    "The value is: $",
    b1[12], ' ',
    b1[8], ' ',
    b1[4], ' ',
    b1[0],
    nl
);

```

```
end Xin128;
```

要将这段代码进行扩展,以便处理长度大于128位的对象,也很简单。只需修改3处:在getb128例程开始的时候必须将整个对象清零;在检查溢出的时候(test(\$F,(type byte LocalValue[15]));指令),必须测试正在处理的新对象的高4位;还必须修改将LocalValue乘以16(借助shld指令)的代码,将正在处理的对象乘以16(也就是说,向左移动4位)。

7. 扩展精度无符号十进制输入

扩展精度无符号十进制输入的算法与十六进制输入的算法几乎等同。实际上,唯一的不同之处(除了前者只接受十进制数字以外)是,对于每一个输入的字符,应该将扩展精度的数值乘以10,而不是16(一般情况下,对于任何基(base)来说,算法都是相同的;只不过将累加的值乘以对应输入数值的基而已)。程序清单8-6中的代码展示了如何编写一个128位无符号十进制输入例程。

程序清单 8-6 扩展精度无符号十进制输入

```

program Uin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    ul28: dword[4];

procedure getul28( var inValue:ul28 ); @nodisplay;
var
    Delimiters: cset;
    LocalValue: ul28;
    PartialSum: ul28;

begin getul28;
    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

    // Get a copy of the HLA standard numeric input delimiters:

```



```

conv.getDelimiters( Delimiters );

// Initialize the numeric input value to 0:
xor( eax, eax );
mov( eax, LocalValue[0] );
mov( eax, LocalValue[4] );
mov( eax, LocalValue[8] );
mov( eax, LocalValue[12] );

// By default, #0 is a member of the HLA Delimiters
// character set. However, someone may have called
// conv.setDelimiters and removed this character
// from the internal Delimiters character set. This
// algorithm depends upon #0 being in the Delimiters
// character set, so let's add that character in
// at this point just to be sure.
cs.unionChar( #0, Delimiters );

// If we're at the end of the current input
// line (or the program has yet to read any input),
// wait for the input of an actual character.
if( stdin.peekc() = #0 ) then
    stdin.readLn();

endif;

// Skip the delimiters found on input. This code is
// somewhat convoluted because stdin.peekc does not
// force the input of a new line of text if the current
// input buffer is empty. We have to force that input
// ourselves in the event the input buffer is empty.
while( stdin.peekc() in Delimiters ) do
    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the
    // delimiter character from the input stream.
    if( al = #0 ) then
        stdin.readLn(); // Force a new input line.
    else
        stdin.getc(); // Remove the delimiter from the input buffer.
    endif;
endwhile;

// Read the decimal input characters and convert
// them to the internal representation:

```

```

while( stdin.peekc() in '0'..'9' ) do

    // Actually read the character to remove it from the
    // input buffer.

    stdin.getc();

    // Ignore underscores, process everything else.

    if( al <> '_' ) then

        and( $f, al );           // '0'..'9' -> 0..9
        mov( eax, PartialSum[0] ); // Save to add in later.

        // Conversion algorithm is the following:
        //
        // (1) LocalValue := LocalValue * 10.
        // (2) LocalValue := LocalValue + al
        //
        // First, multiply LocalValue by 10:

        mov( 10, eax );
        mul( LocalValue[0], eax );
        mov( eax, LocalValue[0] );
        mov( edx, PartialSum[4] );

        mov( 10, eax );
        mul( LocalValue[4], eax );
        mov( eax, LocalValue[4] );
        mov( edx, PartialSum[8] );

        mov( 10, eax );
        mul( LocalValue[8], eax );
        mov( eax, LocalValue[8] );
        mov( edx, PartialSum[12] );

        mov( 10, eax );
        mul( LocalValue[12], eax );
        mov( eax, LocalValue[12] );

        // Check for overflow. This occurs if edx
        // contains a nonzero value.

        if( edx /* <> 0 */ ) then

            raise( ex.ValueOutOfRange );

        endif;

        // Add in the partial sums (including the
        // most recently converted character).

        mov( PartialSum[0], eax );
        add( eax, LocalValue[0] );

        mov( PartialSum[4], eax );

```

```

        adc( eax, LocalValue[4] );

        mov( PartialSum[8], eax );
        adc( eax, LocalValue[8] );

        mov( PartialSum[12], eax );
        adc( eax, LocalValue[12] );

        // Another check for overflow. If there
        // was a carry out of the extended-precision
        // addition above, we've got overflow.

        if( @c ) then

            raise( ex_ValueOutOfRange );

        endif;

    endif;

endwhile;

// Okay, we've encountered a non-decimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.

if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, this conversion has been a success. Let's store
// away the converted value into the output parameter.

mov( inValue, ebx );
mov( LocalValue[0], eax );
mov( eax, [ebx] );

mov( LocalValue[4], eax );
mov( eax, [ebx+4] );

mov( LocalValue[8], eax );
mov( eax, [ebx+8] );

mov( LocalValue[12], eax );
mov( eax, [ebx+12] );

pop( edx );
pop( ecx );
pop( ebx );
pop( eax );

end getul28;

// Code to test the routines above:

```



```

static
    b1:ul28;

begin Uin128;

    stdout.put( "Input a 128-bit decimal value:" );
    getul28( b1 );
    stdout.put
    (
        "The value is: $",
        b1[12], '_',
        b1[8] , '_',
        b1[4] , '_',
        b1[0],
        nl
    );

end Uin128;

```

同十六进制输入的情况一样，将这个十进制输入程序扩展到 128 位以上的位数也非常简单。所要做的就是修改将 LocalValue 变量清零的代码和将 LocalValue 变量乘以 10 的代码(溢出检查在相同的代码中完成，所以这一代码中只有两个地方需要修改)。

8. 扩展精度有符号十进制输入

只要有了一个无符号十进制输入例程，那么编写有符号十进制输入例程就比较简单了。下面的算法描述了如何完成这个工作：

- (1) 读入位于输入流开始处的分隔符。
- (2) 如果下一个输入的字符是一个减号，就读入该字符，并设置一个代表负数的标志。
- (3) 调用无符号十进制输入例程，将剩下的字符串转换成一个整数。
- (4) 检查返回的结果以确定它的高位位是否已被清除。如果结果的高位位已被设置，就产生一个 ex.ValueOutOfRange 异常。
- (5) 如果代码在上述第(2)步中遇到了一个减号，就将结果取负。

实际代码留作读者的编程练习(或者查看 HLA 标准库中的转换例程，了解具体的示例)。

8.2 对不同长度的操作数进行操作

在某些情况下，可能需要对一对长度不同的操作数进行计算。例如，可能需要将一个字和一个双字相加，或者用一个字值去减一个字节值。对于这些情况的解决方法是很简单的：只要将较小的操作数扩展到与较大操作数相同的长度，然后将两个相同长度的操作数进行操作。对于有符号操作数，就要对较小的操作数进行符号扩展，使它的大小与较大的操作数相同；对于无符号的数值，就对较小的操作数进行零扩展。虽然下面的示例只演示了加法操作的情况，但实际上这种方法对于任何操作都是可行的。

要将一个较小的操作数扩展到与较大操作数相同的长度，就要使用符号扩展或零扩展操作(取决于相加的是有符号数值还是无符号数值)。只要将较小的数值扩展到较大数值的长度，加法操作

就可以继续进行了。考虑下面这段将一个字节值与一个字值相加的代码：

```
static
    var1: byte;
    var2: word;

// Unsigned addition:

    movzx( var1, ax );
    add( var2, ax );

// Signed addition:

    movsx( var1, ax );
    add( var2, ax );
```

在这两种情况下，都是将字节变量载入 AL 寄存器中，并且扩展到 16 位，然后与字操作数相加。如果能够选择操作顺序的话(比如说，将 8 位数值和 16 位数值相加)，那么这段代码将会很好地工作。但有时候，并不能指定操作的顺序，可能 16 位数值已经在 AX 寄存器中，但还想将一个 8 位数值加到它上面。对于无符号加法，可以使用下面的代码：

```
mov( var2, ax );      // Load 16-bit value into ax.
.                    // Do some other operations leaving
.                    // a 16-bit quantity in ax.
add( var1, al );      // Add in the 8-bit value.
adc( 0, ah );         // Add carry into the H.O. word.
```

在这个示例中，第一条 add 指令将 var1 的字节和累加器中数值的低位字节相加。上面的 adc 指令将低位字节上产生的进位和累加器中的高位字节相加。这里必须要确保使用 adc 指令。如果将它遗漏，就可能得不到正确的结果。

将一个 8 位有符号操作数和一个 16 位有符号数值相加稍微困难一些。遗憾的是，不能将一个立即值(像上面这样的)加到 AX 中的高位字上。这是因为高位扩展字节可能是\$00，也可能是\$FF。如果寄存器可用的话，最好使用下面的指令：

```
mov( ax, bx );        // bx is the available register.
movsx( var1, ax );
add( bx, ax );
```

如果额外的寄存器不可用，可能就要尝试下面的代码：

```
push( ax );           // Save word value.
movsx( var1, ax );     // Sign extend 8-bit operand to 16 bits.
add( [esp], ax );      // Add in previous word value.
add( 2, esp );         // Pop junk from stack
```

另一种选择是将累加器中的 16 位数值存储到一个存储单元中，然后像以前那样进行操作：


```

mov( ax, temp );
movsx( var1, ax );
add( temp, ax );

```

上面所有的示例都是将一个字节值和一个字值相加。通过使用将较小操作数进行零扩展或符号扩展,使它的长度与较大操作数的长度相等的方法,就很容易将任意两个不同长度的变量相加。

作为最后一个示例,考虑将一个8位有符号数值和一个四字(64位)数值相加:

```

static
QVal:qword;
BVal:int8;

.
.
.
movsx( BVal, eax );
cdq();
add( (type dword QVal), eax );
adc( (type dword QVal[4]), edx );

```

8.3 十进制算术运算

80x86 CPU 使用二进制数字系统进行内部表示。到目前为止,二进制数字系统是计算机系统中所使用的最通用的数字系统。但是在过去曾经有一些计算机系统是基于十进制(以10为基)的数字系统,而不是基于二进制。因此,它们的算术运算系统也是基于十进制而非二进制。这样的计算机系统在以商业/贸易系统为目的的系统中非常流行⁵。虽然系统设计师们已经发现,对于常见计算,二进制算术操作几乎总是比十进制算术操作要好。但是十进制的神话还是使大家坚持认为对于货币的计算,十进制算术操作要比二进制算术操作好。因此,很多软件系统仍然指定在计算中使用十进制算术操作(更不用说原来遗留下来的代码了,它们的算法都只有在使用十进制算术操作的时候才稳定)。因此,尽管十进制算术操作一般来说要比二进制算术操作差,但十进制算术操作的需求仍然存在。

当然,80x86 并不是一种十进制计算机;因此,为了使用二进制格式来表示十进制数字,我们不得不使用一些技巧。最通用的甚至被很多所谓的十进制计算机所用的技术就是使用二进制编码的十进制表示,或称为BCD表示。这种BCD表示使用4位来表示10个可能的十进制数字(见表8-1)。那些4位的二进制值与对应的0~9之间的十进制值相等。当然,我们实际上可以使用4位表示16个不同的值,但BCD格式将忽略剩下的6种二进制位组合。

⁵ 事实上,直到20世纪60年代中期IBM360发布之前,大多数科学计算机系统是基于二进制的,但多数商业/贸易系统还是基于十进制的。IBM推出它的system/360,将其作为一种同时针对商业和科学应用程序的单用途解决方案。实际上,型号标志(360)起源于罗盘上的360度,这就暗示了system/360可以用于“罗盘上任何一点的”计算(也就是商业和科学计算)。

表 8-1 二进制编码的十进制(BCD)表示

BCD 表示	十进制表示
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	非法
1011	非法
1100	非法
1101	非法
1110	非法
1111	非法

因为每一个 BCD 数字需要 4 位,所以我们可以用一个字节表示两位 BCD 数字。这就意味着我们可以使用一个字节表示 0~99 之间的十进制数值(如果我们将 BCD 数值看成无符号二进制数的话,范围就是 0~255)。显然,以 BCD 形式表示相同的数值比使用二进制表示要多占用存储空间。例如,用一个 32 位数值,可以表示 0~99999999(8 位有效数字)之间的 BCD 值。但是,使用二进制表示的话,就可以表示 0~4294967295(多于 9 位有效数字)之间的数值。

在二进制计算机上,BCD 格式不仅浪费存储空间(因为它使用更多的位来表示一个给定的整数),而且它的十进制算术操作也要慢一些。由于这些原因,我们应该避免使用十进制算术操作,除非是程序对此有特殊的要求。

但是相对于二进制表示来说,二进制编码的十进制表示的确提供了很好的优点,即:在一个十进制数字的字符串表示和 BCD 表示之间作转换是相当简单的。这个特性在对分数值进行操作的时候特别有好处,因为定点和浮点二进制表示都不能准确地表示很多经常使用的 0~1 之间的数值(如 1/10)。因此,在从 BCD 设备读取数据,然后进行简单的算术操作(如一次加法),并将 BCD 数值写入其他设备时,BCD 操作是很高效的。

8.3.1 字面 BCD 常量

HLA 并不提供(且一般也不需要)一个特殊的字面 BCD 常量。因为 BCD 只是十六进制符号中的一种特殊形式,它不使用 \$A~\$F 之间的值,所以可以使用 HLA 十六进制符号简单地建立一个 BCD 常量。当然,必须注意,在 BCD 常量中不能包含符号 A~F,因为它们是非合法的 BCD 数值。作为一个示例,我们考虑下面的 mov 指令,该指令将 BCD 数值 99 复制到 AL 寄存器中:

```
mov( $99, al )
```

有一件重要的事情需要记住，那就是千万不要对 BCD 数值使用 HLA 字面十进制常量。也就是说，`mov(95,al)`不会将 95 的 BCD 表示加载到 AL 寄存器中，而是将\$5F 载入 AL，而这是一个非法的 BCD 数值。试图对非法 BCD 数值进行的任何操作都会产生无用的结果。所以要记住，即使看起来是违反直觉的，我们也要使用十六进制字面常量来表示字面 BCD 数值。

8.3.2 80x86 的 daa 指令和 das 指令

80x86 的整数单元不直接支持 BCD 算术操作。实际情况是，80x86 需要程序员自己使用二进制算术操作进行计算，然后使用一些辅助的指令将二进制结果转换成 BCD。为了支持每个字节表示两位数的压缩 BCD 数值的加法和减法操作，80x86 提供了两条指令：加法的十进制调整指令(`daa`)和减法的十进制调整指令(`das`)。可以在 `add/adc` 或 `sub/sbb` 指令后立即执行这两条指令，来修正 AL 寄存器中的二进制结果。

要将一对两位(也就是一个字节)的 BCD 数值相加，就要使用下面的代码序列：

```
mov( bcd_1, al ); // Assume that bcd1 and bcd2 both contain
add( bcd_2, al ); // value BCD values.
daa();
```

前两条指令使用标准二进制算术操作将两个两字节数值相加，这并不能产生一个正确的 BCD 结果。例如，若 `bcd_1` 包含\$9 且 `bcd_2` 包含\$1，则上面的前两条指令就会产生二进制和\$A，而不是正确的 BCD 值\$10。`daa` 指令可以调整这个不合法的结果，它会检查 BCD 的低位数位上有没有产生进位，并且当出现溢出的时候，它会调整该数值(通过给这个数值加 6)。在调整了低位数位上的溢出之后，`daa` 指令将会对高位数位重复这一过程。如果在操作中，高位数位上有(十进制)进位，那么 `daa` 指令就会设置进位标志。

`daa` 指令仅对 AL 寄存器进行操作。当试图将一个数值加到 AX、EAX 或者其他寄存器中时，它不会对十进制加法作(适当)调整。尤其要注意的是，`daa` 限制了一次只能将两个十进制数字(一个字节)相加，这就意味着如果目的是计算十进制和的话，那么就不得不将 80x86 看作一个 8 位处理器，而且一次相加 8 位数据。如果希望一次相加多于两个的数字，就必须将这种操作看作是一种多精度操作。例如，要将 4 个十进制数相加(使用 `daa`)的话，就必须执行下面的代码序列：

```
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"
```

```
mov( bcd_1[0], al );
add( bcd_2[0], al );
daa();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
adc( bcd_2[1], al );
daa();
mov( al, bcd_3[1], al );
```

```
// Carry is set at this point if there was unsigned overflow.
```

因为两个字的二进制加法(产生一个字的结果)只需要 3 条指令，所以可以看出十进制算术操

作是多么的费事⁶。

das(减法的十进制调整)指令在一次二进制 **sub** 或 **sbb** 指令之后调整十进制结果。使用它的方法和使用 **daa** 指令的方法相同。示例如下:

```
// Two-digit(1-byte) decimal subtraction:

    mov( bcd_1, al ); // Assume that bcd_1 and bcd_2 both contain
    sub( bcd_2, al ); // valid BCD values.
    das();

// Four-digit(2-byte) decimal subtraction.
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

    mov( bcd_1[0], al );
    sub( bcd_2[0], al );
    das();
    mov( al, bcd_3[0] );
    mov( bcd_1[1], al );
    sbb( bcd_2[1], al );
    das();
    mov( al, bcd_3[1], al );

// Carry is set at this point if there was unsigned overflow.
```

遗憾的是, 80x86 只支持使用 **daa** 和 **das** 指令的压缩 BCD 数值的加法和减法操作。它不支持乘法、除法或者其他任何算术操作。因为使用这些指令的十进制算术操作如此有限, 所以很少看到有程序使用这些指令。

8.3.3 80x86 的 **aaa**、**aas**、**aam** 和 **aad** 指令

除了压缩十进制指令(**daa** 和 **das**), 80x86 CPU 还支持 4 种非压缩十进制调整指令。非压缩十进制数字在每个字节中只存储一个十进制位。这种数据表示方法浪费了大量的存储空间。但是, 非压缩十进制调整指令支持乘法和除法操作, 所以它们要稍微有用一些。

指令助记符 **aaa**、**aas**、**aam** 和 **aad** 分别代表 ASCII adjust for Addition、ASCII adjust for Subtraction、ASCII adjust for Multiplication 和 ASCII adjust for Division。尽管它们的名字如此, 但这些指令实际上并不处理任何 ASCII 字符。它们支持 AL 中的非压缩十进制值, 这些数值的低 4 位包含着十进制位, 而高 4 位包含 0。注意, 只要简单地将 AL 和 \$0F 进行 **and** 操作, 就可以将其中的 ASCII 十进制位字符转换成一个非压缩十进制数。

aaa 指令对两个非压缩十进制数的二进制加法的结果进行调整。如果两个数值的和超过 10, 那么 **aaa** 就会从 AL 中减掉 10, 并将 AH 增加 1(同时还将设置进位标志)。**aaa** 假设相加的两个值是合法的非压缩十进制数值。除了 **aaa** 每次只操作一个(而不是两个)十进制位以外, 使用 **aaa** 的方法和使用 **daa** 指令的方法相同。当然, 如果需要将一串十进制位相加, 使用非压缩十进制算术操作就需要双倍的操作量, 并且因此就需要双倍的执行时间。

使用 **aas** 指令的方法和使用 **das** 指令的方法相同, 当然, 除了它操作的是非压缩十进制数而

⁶ 您很快就会看到, 很少会使用这种方法完成十进制算术操作——所以这无关紧要。

不是压缩十进制数以外。和 `aaa` 的情况一样, `aas` 需要双倍的操作量才能相加与 `das` 指令相同个数的十进制位。如果不知道为什么有人会想要使用 `aaa` 或者 `aas` 指令, 请别忘了, 非压缩格式是支持乘法和除法的, 而压缩格式则不支持这两种操作。因为压缩数据和解压缩数据通常比每次只操作一位数据更费事, 所以当不得不对非压缩数据进行操作(因为需要乘法和除法)时, `aaa` 和 `aas` 指令更高效。

`aam` 指令在使用 `mul` 指令将两个非压缩十进制数字相乘之后, 将修改 `AX` 寄存器中的结果, 产生一个正确的非压缩十进制结果。因为能得到的最大乘积是 81($9*9$ 是两个一位数值相乘所能得到的最大乘积), 所以 `AL` 寄存器能够容纳相乘的结果。`aam` 将二进制结果除以 10 来对其进行解压缩, 并将商(高位数位)保留在 `AH` 中, 将余数(低位数位)保留在 `AL` 中。注意, `aam` 将商和余数保留在不同的寄存器中, 这与标准的 8 位 `div` 操作是不同的。

从技术上来讲, 不必使用 `aam` 指令来进行 BCD 乘法操作。`aam` 只是将 `AL` 除以 10, 并把商和余数(分别)保留在 `AH` 和 `AL` 中。如果需要进行这种特殊的操作, 就可以使用 `aam` 指令达到这个目的(实际上, 这大概也是 `aam` 指令在当今程序中的唯一用途)。

如果需要使用 `mul` 和 `aam` 将多于两个的非压缩十进制位相乘, 就需要设计一个多精度乘法, 而它会使用本章前面提到的手工算法。因为那将是一个大量的工作, 所以这一节就不给出算法了。如果需要进行多精度十进制乘法, 请看 8.3.4 节, 其中提供了一种更好的解决方法。

`aad` 指令调整非压缩十进制除法的结果。这条指令的特殊性是必须在 `div` 操作之前执行该指令。它假设 `AL` 包含一个两位数值的最低位, 并且 `AH` 包含着一个两位非压缩十进制数值的最高位。它将这两个数字转换成二进制数, 这样一条标准的 `div` 指令就可以产生正确的非压缩十进制结果。像 `aam` 一样, 这条指令就其设计意图来讲几乎没有什么用处, 因为扩展精度操作(比如说, 多于一到两位的除法)是极其低效的。但是, 这条指令实际上就其本身来讲相当有用。它可以计算 $AX = AH * 10 + AL$ (假设 `AH` 和 `AL` 包含一位的十进制数值)。对于一个 0~99 之间的数值来说, 其 ASCII 表示需要两个字符, 对于这两个字符构成的字符串来说, 使用这条指令就可以将其转换成一个二进制值。例如:

```
mov( '9', al );
mov( '9', ah );          // "99" is in ah:al.
and( $0F0F, ax );        // Convert from ASCII to unpacked decimal.
aad();                   // After this, ax contains 99.
```

十进制和 ASCII 调整指令提供了十进制算术操作的一种极其低效的实现。为了在 80x86 系统上更好地支持十进制算术操作, Intel 将十进制操作整合在 FPU 中。下一节将讨论如何使用 FPU 来达到此目的。但是, 即使有了 FPU 的支持, 十进制算术操作与二进制算术操作相比仍然非常低效, 而且不够精确。因此, 在程序中使用十进制算术操作之前, 应该仔细考虑是否真正需要使用它。

8.3.4 使用 FPU 的压缩十进制算术操作

为了提高依赖十进制算术操作的应用程序的性能, Intel 将对十进制算术的支持直接整合到 FPU 中。与前面讲述的压缩和非压缩十进制格式不同, 就 FPU 的速度来说, 它只支持最多 18 位精度的数值。此外, 除了加法、减法、乘法和除法以外, FPU 还支持所有算术操作(例如, 超越操作)。如果可以接受 18 位精度的限制以及其他一些约束, 那么在程序中必须使用十进制算术操作

时,在 FPU 上进行十进制操作是正确的途径。

使用 FPU 的时候,首先要注意,它实际上不支持十进制算术操作。实际情况是,FPU 提供了两条指令,fbld 和 fbstp。每当将数据移入 FPU 或从 FPU 移出时,这两条指令将在压缩十进制和二进制浮点格式之间进行转换。fbld(float/BCD load,浮点/BCD 载入)指令在将一个 80 位的压缩 BCD 数值转换成 IEEE 二进制浮点格式之后,将其载入 FPU 栈的顶端。同样,fbstp(float/BCD store and pop,浮点/BCD 存入和弹出)指令将这个浮点数值从栈顶弹出,并将其转换成一个压缩 BCD 数值,然后将这个 BCD 数值存储至目的存储单元。

一旦将一个压缩 BCD 数值载入 FPU,它就不再是一个 BCD 数值了,而只是一个浮点数值而已。这样就形成了使用 FPU 作为十进制整数处理器的第一个约束:计算过程是使用二进制算术操作完成的。如果有一个算法,它绝对依赖于十进制算术操作的话,那么使用 FPU 来实现它就会导致失败⁷。

第二个局限是 FPU 只支持一种 BCD 数据类型:一种十字节的、18 位的压缩十进制数值。它不支持更小的数值,也不支持更大的数值。因为 18 位通常是足够的,并且存储器也很便宜,所以这并不算是一种大的约束。

第三点考虑是压缩 BCD 和浮点格式之间的转换是一种开销很高的操作。fbld 和 fbstp 指令可能相当慢(例如,比 fld 和 fstp 慢两个数量级还多)。因此,在做简单的加法和减法操作时,这些指令可能非常耗时;转换操作所耗费的时间要远远超出使用 daa 和 das 指令将数值逐个字节相加所用的时间(但是,乘法和除法在 FPU 上会运行得快一些)。

为什么 FPU 的压缩十进制格式只支持 18 位呢?毕竟使用 10 个字节就应该可以表示 20 位 BCD 数字。实际上,FPU 的压缩十进制格式使用前 9 个字节,以一种标准的压缩十进制格式包含压缩 BCD 数值(第 1 个字节包含两个低位数位,而第 9 个字节包含两个高位数位)。第 10 个字节的高位数位中包含着符号位,而且 FPU 会忽略第 10 个字节中剩下的位。Intel 之所以不愿意再挤出更多的一位数字(比如说,使用第 10 个字节中低 4 位来达到 19 位精度),是因为这样做可能导致一些 FPU 无法使用本身的浮点格式来精确地表示 BCD 数值。这样,就有了 18 位精度的限制。

对于负的 BCD 数值,FPU 使用反码表示法。也就是说,如果数字是负数或者 0,那么符号位就包含一个 1;如果数字是正数或者 0,就包含一个 0(就像二进制反码格式那样,对于 0 来说也有两种不同的表示)。

HLA 的 tbyte 类型是标准的数据类型,可以用于定义压缩 BCD 变量。fbld 和 fbstp 指令需要一个 tbyte 操作数(可以使用十六进制/BCD 值进行初始化)。

因为 FPU 将压缩十进制数值转换成内部的浮点格式,所以就可以在同一次计算中混合使用压缩十进制、浮点和(二进制)整数格式。程序清单 8-7 中的程序演示了如何实现这一目标。

程序清单 8-7 混合模式的 FPU 算术操作

```
program MixedArithmetic;
#include( "stdlib.hhf" )

static
    tb: tbyte := $654321;
```

⁷ 这种算法的一个示例可以是将数字左移一位而实现的乘以 10 的操作。但是,这样的操作不可能出现在 FPU 当中,所以在 FPU 内部,行为不当的算法实际上很少见。


```

begin MixedArithmetic;

    fbld( _tb );
    fmul( 2.0 );
    fiadd( 1 );
    fbstp( _tb );
    stdout.put( "bcd value is" );
    stdout.puth80( _tb );
    stdout.newln();

end MixedArithmetic;

```

FPU 将压缩十进制数值作为整数值来对待。因此，如果计算结果产生了分数，那么 `fbstp` 指令将会根据 FPU 当前的舍入模式对结果进行四舍五入。如果需要操作这些分数值，就需要保留浮点结果。

8.4 表

“表”这个词对于不同的程序员来说有着不同的意义。对于大多数汇编语言程序员来说，一张表无非就是一个由若干数据初始化的数组。汇编语言程序员经常使用表来计算复杂的或者执行很慢的函数。很多非常高级的语言(如 SNOBOL4 和 Icon)直接支持一种称作表的数据类型。在这些语言中，表实质上是联合数组，它们的元素可以使用一个非整数的索引来访问(比如说，浮点、字符串或者任何其他的数据类型)。HLA 提供了一个表模块，从而可以使用一个字符串对一个数组编制索引。不过，在这一章中，我们将采用汇编语言程序员对于表的观点。

一张表就是一个数组，它包含预先初始化了的数值，这些数值在程序执行的过程中是不变的。在汇编语言里，可以使用表来做很多不同的事情：计算函数、控制程序或者只是用来查找。一般来说，表提供了一种快速的机制在程序中进行某些操作，代价是耗费一些空间(额外的这些空间用于存放表格式的数据)。在下面的部分中，我们将研究表在一个汇编语言程序中的一些使用方法。

注意，因为在一般的情况下，表包含着预先定义好的数据，这些数据在程序的执行过程中不会变化，所以只读段是存放表对象的理想场所。

8.4.1 通过表查找进行函数计算

在汇编语言中，表可以做各种各样的事情。在高级语言(如 Pascal)中，创建一个公式进行求值的确很简单。但一个看上去简单的高级语言算术表达式实际上等同于相当数量的 80x86 汇编语言代码，因此其计算代价是很高的。汇编语言程序员会经常预先计算很多数值，并且对于这些数值使用一个表查找过程来加速他们的程序。这样做的优点是可以使操作过程更简单，并且更高效。考虑下面的 Pascal 语句：

```

if (character >= 'a') and (character <= 'z') then character :=
chr(ord(character) - 32);

```

这个 Pascal 的 if 语句当 *character* 是 a~z 之间的字符时，将它们从小写形式转换成大写形式。完成同样工作的 HLA 代码为：


```

mov( character, al );
if( al in 'a'..'z' ) then
    and( $5f, al );    // Same as sub( 32, al ) in this code.
endif;
mov( al, character );

```

注意, 在这个特殊的示例中, HLEA 的高级 if 语句被翻译成 4 条机器指令。这样, 这段代码总共需要 7 条机器指令。

如果把这样的代码放在嵌套的循环中, 那么如果不使用表查找的话, 我们几乎不能减少这段代码的长度。但是, 使用表查找的话, 就可以将这个代码序列减少到只有 4 条指令:

```

mov( character, al );
lea( ebx, CnvrtLower );
xlat
mov( al, character );

```

这段代码如何工作呢? 而这条新的指令 xlat 又是什么呢? xlat 其实就是翻译指令, 它完成这样的操作:

```

mov( [ebx+al*1], al );

```

也就是说, 这条指令使用 AL 寄存器的当前值作为索引访问基址存在于 EBX 中的数组。它在数组的那个索引处取出一个字节, 并把该字节复制到 AL 寄存器中。Intel 将这条指令称作 translate(翻译指令), 因为程序员一般使用这条指令通过表查找将字符从一种形式翻译成另一种形式。这也正是我们在此处的用法。

在前面的示例中, CnvrtLower 是一个 256 字节的表, 它包含索引 0~\$60 上的数值 0~\$60、索引 \$61~\$7A 上的数值 \$41~\$5A、以及索引 \$7B~\$FF 上的数值 \$7B~\$FF。因此, 如果 AL 寄存器包含一个 0~\$60 之间的数值, 那么 xlat 指令就将返回一个 0~\$60 之间的数值, 有效地保持 AL 的值未改变。但是, 如果 AL 中包含的值在 \$61~\$7A 之间(a~z 的 ASCII 码), 那么 xlat 指令就会用一个 \$41~\$5A 的值取代 AL 中的值, 而 \$41~\$5A 之间的值恰好是 A~Z 的 ASCII 码。因此, 如果 AL 原来包含着一个一个小写字符(\$61~\$7A), 那么 xlat 指令就使用对应的 \$41~\$5A 中的值替换 AL 中的值, 这样就有效地将小写字符(\$61~\$7A)转换成大写字符(\$41~\$5A)。表中剩下的项, 比如 \$0~\$60, 只是包含表中特定元素的索引。因此, 如果 AL 原来包含着一个 \$7A~\$FF 之间的值, 那么 xlat 指令将返回对应的包含 \$7A~\$FF 之间值的表项。

随着函数复杂度的增加, 表查找方法带来的性能收益也将大幅度增长。尽管从来不会使用查找表的方法将小写字母转换成大写字母, 但是此处请考虑这种转换形式。例如, 通过计算:

```

mov( character, al );
if( al in 'a'..'z' ) then
    and( $5f, al );
elseif( al in 'A'..'Z' ) then
    or( $20, al );

```

```
endif;
mov( al, character );
```

if 和 elseif 语句分别产生了 4 条和 5 条真正的机器指令，所以这段代码等于 13 条真正的机器指令。

计算该函数的表查找代码为：

```
mov( character, al );
lea( ebx, SwapUL );
xlat();
mov( al, character );
```

在使用表查找过程来计算一个函数的时候，只是表发生了变化，代码保持不变。

表查找的主要问题在于：使用表查找所计算的函数定义域是有限的。一个函数的定义域是能够接受的可能的输入值(参数)的集合。例如，上面的大/小写转换函数的定义域是一个由 256 个字符构成的 ASCII 字符集。

诸如 SIN 或 COS 这样的函数可以接受实数集作为输入的值。显然，SIN 和 COS 的定义域比大/小写转换函数的定义域大得多。如果要通过使用表查找进行计算的话，就必须将函数的定义域限制为一个很小的范围。这是因为在一个函数的定义域内，每个元素在查找表中都对应一项。所以要使用表查找来实现一个定义域为实数集的函数是不太实际的。

大多数的查找表都相当小，通常是 10~256 项，很少能看到有超过 1000 项的查找表。大多数程序员都不会有耐心去创建一个 1000 项的表(并且验证其正确性)。

基于查找表的函数的另一个限制是，函数定义域中的值应该是连续的。表查找接受函数的输入值，使用这个输入值作为表的索引，然后返回表中由这个索引所指定的项的值。如果不给函数传递除了 0、100、1000 和 10000 以外的任何值，那么这似乎是一种理想的使用表查找的候选实现；它的定义域只包含 4 个元素。但是，由于输入值范围的关系，这张表实际上会需要 10001 个不同的元素。因此，不能够通过表查找的方法高效地建立这样一个函数。在本节关于表的部分，我们将假设函数的定义域是一个由连续数值所构成的集合。

可以使用表查找来实现的最好的函数是那些定义域和值域总是在 0~255 之间(或这个范围的某些子集)的函数，通过使用 xlat 指令可以在 80x86 上高效地实现这些函数。前面给出的大/小写转换例程就是这样的函数。这一类的任何函数(定义域和值域都在 0~255 之间的函数)都可以使用两条指令来计算：lea(table,ebx);和 xlat();，其中唯一变化的内容就是查找表。

只要一个函数的值域或者定义域超过 0~255 的范围，那么就不能(方便地)使用 xlat 指令计算这个函数。有 3 种情形需要考虑：

- 定义域超过 0~255，但值域在 0~255 之内。
- 定义域在 0~255 之内，但值域超过 0~255。
- 函数的定义域和值域都超过 0~255 的范围。

我们将分别考虑这 3 种情况。

如果一个函数的定义域超过 0~255，但值域落在 0~255 之内，那么我们的查找表就需要多于 256 项，但我们可以使用一个单独的字节来代表每一项。因此，查找表可以是一个字节数组。除了那些使用 xlat 指令进行的查找以外，归入这一类的函数是最高效的。下面的 Pascal 函数调用


```
B := Func(X);
```

其中, Func 是:

```
function Func(X:dword):byte;
```

可以简单地转换成下面的 HLA 代码:

```
mov( X, ebx );
mov( FuncTable[ ebx ], al );
mov( al, B );
```

这段代码将函数的参数载入 ebx, 使用这个值(在 0 到某个数之间)作为 FuncTable 表的一个索引, 取出该位置上的字节, 然后将结果存入 B。显然, 这张表对于每一个可能的 X 值都必须包含合法的项。例如, 假设要想在屏幕上将一个 0~1999 之间(在一个 80×25 的视频显示器上有 2000 个字符的位置)的光标位置映射为屏幕上的 X 或 Y 坐标, 就可以使用这个函数简单地计算 X 坐标:

```
X := Posn mod 80
```

Y 坐标的计算公式为:

```
Y := Posn div 80
```

(其中 Posn 是屏幕上的光标位置)。这可以使用 80x86 代码简单地计算出来:

```
mov( Posn, ax );
div( 80, ax );
```

```
// X is now in ah, Y is now in al
```

但是, 80x86 上的 div 指令非常慢。如果需要对每一个将要写到屏幕上的字符都做这样的计算, 那么就会严重降低视频显示代码的执行速度。下面的代码通过表查找来实现这两个函数, 可以显著提高代码的性能。

```
movzx( Posn, ebx );           // Use a plain mov instr if Posn is
mov( YCoord[ebx], al );       // uns32 rather than an uns16 value.
mov( XCoord[ebx], ah );
```

如果一个函数的定义域在 0~255 之内, 但值域超过这个范围, 那么查找表就会包含 256 个或者更少的项, 但每一项仍然需要两个或更多的字节。如果函数的值域和定义域都超过 0~255 的范围, 那么不仅每个项都需要两个或更多的字节, 而且整张表将包含多于 256 项。

回想关于数组的内容, 用索引在一维数组(表是这种数组的一个特例)中定位的公式为:

```
Address := Base + index * size
```

如果函数值域中的元素需要两个字节, 那么在定位到表中之前, 必须将这个索引乘以 2。同样, 如果每项需要 3 个、4 个或者更多的字节, 那么在定位到表中之前, 必须将索引乘以每个表项的字节数。例如, 假设有一个函数 F(x), 由下面的(伪)Pascal 语句声明:

```
function F(x:dword):word;
```

可以使用下面的 80x86 代码(当然,表的名称应该是 F)简单地建立这个函数:

```
mov( X, ebx );
mov( F[ebx*2], ax );
```

任何定义域较小并且在多数情况下连续的函数都是使用表查找进行计算的极佳候选。在某些情况下,非连续的定义域也是可以接受的,只要该定义域可以被强制转换为一个恰当的数值集合即可。这样的操作被称为调节(conditioning),并且是下一节将要讨论的主题。

8.4.2 域调节

域调节是指将函数定义域里的一组数值取出来,对它们进行修改,使它们成为该函数可以接受的输入。考虑下面的函数:

$$\sin x = \sin x | (x \in [-2\pi, 2\pi])$$

这就是说,(计算机上的)函数 $\sin(x)$ 与(数学上的)函数 $\sin x$ 在下列定义域上是相同的:

$$-2\pi \leq x \leq 2\pi$$

我们知道,正弦函数是一个三角函数,它可以接受任何实数值。但是计算机上使用的正弦公式只接受这些值当中的一个很小的子集。

值域的限制不会带来任何问题:只要简单地计算 $\sin(X \bmod (2\pi))$,我们就可以得到任何输入值的正弦值。修改一个输入值使我们能够简单地进行计算的过程就叫做“调节输入”。在上面的示例中,我们计算了 $X \bmod 2\pi$,并且使用它的结果作为 \sin 函数的输入。这样就把 X 限制在 \sin 所需的定义域内,而不会对结果产生影响。我们也可以将“调节输入”应用到表查找过程中。实际上,将索引放大,使其能够处理一个字长的数据项,这也是一种“调节输入”的形式。考虑下面的 Pascal 函数:

```
function val(x:word):word; begin
  case x of
    0: val := 1;
    1: val := 1;
    2: val := 4;
    3: val := 27;
    4: val := 256;
    otherwise val := 0;
  end;
end;
```

这个函数计算当 x 在 0~4 之间变化时的取值,并且在 x 超出这个范围的时候返回 0。因为 x 可以取 65536 个不同的值(作为一个 16 位的字),所以建立一个包含 65536 个字但只有前 5 项是非零值的表似乎有些浪费。不过,如果我们使用“调节输入”的话,那么仍然可以使用表查找过程来计算这个函数。下面的汇编语言代码体现了这个原理:


```

mov( 0, ax ); // ax = 0, assume x > 4.
movzx( x, ebx ); // Note that H.O. bits of ebx must be 0!
if( bx <= 4 ) then
    mov( val[ ebx*2 ], ax );
endif;

```

这段代码先检查 x 是否超出 0~4 的范围。如果是，就人为地将 AX 设置为 0；否则就在 val 表中查找函数的值。通过“调节输入”，一些原本不可能通过表查找实现的函数就可以实现了。

8.4.3 产生表

使用表查找有一个很大的问题，那就是首先要建立一张表。当要建立的表含有大量的数据项时，这一问题尤为明显。先计算要放在表里的数据，然后输入数据，最后还要检查这些数据是否合法，这是一个费时而且枯燥的过程。对于很多表来说，并没有能够绕过这个过程的捷径。而对于另外一些表，就存在更好的方法——使用计算机来产生表。有一个示例能够很好地说明这一点，考虑对正弦函数作如下修改：

$$\sin(x) \times r = \left\langle \frac{(r \times (1000 \times \sin x))}{1000} \right\rangle [x \in 0, 359]$$

这里规定 x 是一个在 0~359 之间的整数，而且 r 必须是一个整数。这样计算机就可以使用下面的代码进行简单的计算：

```

movzx( x, ebx );
mov( Sines[ ebx*2 ], eax ); // Get sin(X)*1000
imul( r, eax ); // Note that this extends eax into edx.
idiv( 1000, edx:eax ); // Compute (r*(sin(X)*1000))/1000

```

注意，整数乘法和除法是不满足结合性的，所以不能因为乘以 1000 和除以 1000 看上去相互抵消了就把它们删除。此外，代码还必须以这样的顺序计算该函数。对于我们来说，要完成这个函数所需的全部工作就是构造一张表，这张表里包含 360 个不同的数值，分别对应各种角(以角度为单位)的正弦值乘以 1000 的数字。将这样一张表输入到汇编语言程序中是一件极其枯燥的事，并且在输入和校验数据的过程中可能还会出现若干错误。不过可以让程序来产生这张表，考虑程序清单 8-8 中的 HLA 程序。

程序清单 8-8 产生一个正弦值表的 HLA 程序

```

program GenerateSines;
#include( "stdlib.hhf" );

var
    outFile:   dword;
    angle:     int32;
    r:         int32;

readonly
    RoundMode: uns16 := $23f;

```

```

begin GenerateSines;

    // Open the file:

    mov( fileio.openNew( "sines.hla", outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        "sines: int32[360] := "nl,
        stdio.tab, stdio.tab, stdio.tab, "[ " nl ];

    // Enable rounding control (round to the nearest integer).

    fldcw( RoundMode );

    // Emit the sines table:

    for( mov( 0, angle ); angle < 359; inc( angle ))do

        // Convert angle in degrees to an angle in radians using
        // radians := angle * 2.0 * pi /360.0;

        fild( angle );
        fld( 2.0 );
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

        // Okay, compute the sine of st0.

        fsin();

        // Multiply by 1000 and store the rounded result into
        // the integer variable r.

        fld( 1000.0 );
        fmul();
        fistp( r );

        // Write out the integers eight per line to the source file.
        // Note: If (angle AND %111) is 0, then angle is evenly
        // divisible by 8 and we should output a newline first.

        test( %111, angle );
        if( @z )then

            fileio.put
            (
                outFile,
                nF,

```



```

        stdio.tab,
        stdio.tab,
        stdio.tab,
        stdio.tab,
        r:5,
        ',',
    );

else
    fileio.put( outFile,r: 5, ',' );

endif;

endfor;

// Output sine(359) as a special case (no comma following it).
// Note: This value was computed manually with a calculator.

fileio.put
(
    outFile,
    "-17",
    nl,
    stdio.tab,
    stdio.tab,
    stdio.tab,
    "];",
    nl
);

fileio.close( outFile );

end GenerateSines;

```

上述程序将产生下面的输出(部分节选):

```

sines: int32[360] :=
[
    0,      17,      35,      52,      70,      87,      105,      122,
    139,    156,    174,    191,    208,    225,    242,    259,
    276,    292,    309,    326,    342,    358,    375,    391,
    407,    423,    438,    454,    469,    485,    500,    515,
    530,    545,    559,    574,    588,    602,    616,    629,
    643,    656,    669,    682,    695,    707,    719,    731,
    743,    756,    769,    782,    795,    807,    819,    831,
    843,    856,    869,    882,    895,    907,    919,    931,
    943,    956,    969,    982,    995,   1007,   1019,   1031,
    1043,   1056,   1069,   1082,   1095,   1107,   1119,   1131,
    1143,   1156,   1169,   1182,   1195,   1207,   1219,   1231,
    1243,   1256,   1269,   1282,   1295,   1307,   1319,   1331,
    1343,   1356,   1369,   1382,   1395,   1407,   1419,   1431,
    1443,   1456,   1469,   1482,   1495,   1507,   1519,   1531,
    1543,   1556,   1569,   1582,   1595,   1607,   1619,   1631,
    1643,   1656,   1669,   1682,   1695,   1707,   1719,   1731,
    1743,   1756,   1769,   1782,   1795,   1807,   1819,   1831,
    1843,   1856,   1869,   1882,   1895,   1907,   1919,   1931,
    1943,   1956,   1969,   1982,   1995,   2007,   2019,   2031,
    2043,   2056,   2069,   2082,   2095,   2107,   2119,   2131,
    2143,   2156,   2169,   2182,   2195,   2207,   2219,   2231,
    2243,   2256,   2269,   2282,   2295,   2307,   2319,   2331,
    2343,   2356,   2369,   2382,   2395,   2407,   2419,   2431,
    2443,   2456,   2469,   2482,   2495,   2507,   2519,   2531,
    2543,   2556,   2569,   2582,   2595,   2607,   2619,   2631,
    2643,   2656,   2669,   2682,   2695,   2707,   2719,   2731,
    2743,   2756,   2769,   2782,   2795,   2807,   2819,   2831,
    2843,   2856,   2869,   2882,   2895,   2907,   2919,   2931,
    2943,   2956,   2969,   2982,   2995,   3007,   3019,   3031,
    3043,   3056,   3069,   3082,   3095,   3107,   3119,   3131,
    3143,   3156,   3169,   3182,   3195,   3207,   3219,   3231,
    3243,   3256,   3269,   3282,   3295,   3307,   3319,   3331,
    3343,   3356,   3369,   3382,   3395,   3407,   3419,   3431,
    3443,   3456,   3469,   3482,   3495,   3507,   3519,   3531,
    3543,   3556,   3569,   3582,   3595,   3607,   3619,   3631,
    3643,   3656,   3669,   3682,   3695,   3707,   3719,   3731,
    3743,   3756,   3769,   3782,   3795,   3807,   3819,   3831,
    3843,   3856,   3869,   3882,   3895,   3907,   3919,   3931,
    3943,   3956,   3969,   3982,   3995,   4007,   4019,   4031,
    4043,   4056,   4069,   4082,   4095,   4107,   4119,   4131,
    4143,   4156,   4169,   4182,   4195,   4207,   4219,   4231,
    4243,   4256,   4269,   4282,   4295,   4307,   4319,   4331,
    4343,   4356,   4369,   4382,   4395,   4407,   4419,   4431,
    4443,   4456,   4469,   4482,   4495,   4507,   4519,   4531,
    4543,   4556,   4569,   4582,   4595,   4607,   4619,   4631,
    4643,   4656,   4669,   4682,   4695,   4707,   4719,   4731,
    4743,   4756,   4769,   4782,   4795,   4807,   4819,   4831,
    4843,   4856,   4869,   4882,   4895,   4907,   4919,   4931,
    4943,   4956,   4969,   4982,   4995,   5007,   5019,   5031,
    5043,   5056,   5069,   5082,   5095,   5107,   5119,   5131,
    5143,   5156,   5169,   5182,   5195,   5207,   5219,   5231,
    5243,   5256,   5269,   5282,   5295,   5307,   5319,   5331,
    5343,   5356,   5369,   5382,   5395,   5407,   5419,   5431,
    5443,   5456,   5469,   5482,   5495,   5507,   5519,   5531,
    5543,   5556,   5569,   5582,   5595,   5607,   5619,   5631,
    5643,   5656,   5669,   5682,   5695,   5707,   5719,   5731,
    5743,   5756,   5769,   5782,   5795,   5807,   5819,   5831,
    5843,   5856,   5869,   5882,   5895,   5907,   5919,   5931,
    5943,   5956,   5969,   5982,   5995,   6007,   6019,   6031,
    6043,   6056,   6069,   6082,   6095,   6107,   6119,   6131,
    6143,   6156,   6169,   6182,   6195,   6207,   6219,   6231,
    6243,   6256,   6269,   6282,   6295,   6307,   6319,   6331,
    6343,   6356,   6369,   6382,   6395,   6407,   6419,   6431,
    6443,   6456,   6469,   6482,   6495,   6507,   6519,   6531,
    6543,   6556,   6569,   6582,   6595,   6607,   6619,   6631,
    6643,   6656,   6669,   6682,   6695,   6707,   6719,   6731,
    6743,   6756,   6769,   6782,   6795,   6807,   6819,   6831,
    6843,   6856,   6869,   6882,   6895,   6907,   6919,   6931,
    6943,   6956,   6969,   6982,   6995,   7007,   7019,   7031,
    7043,   7056,   7069,   7082,   7095,   7107,   7119,   7131,
    7143,   7156,   7169,   7182,   7195,   7207,   7219,   7231,
    7243,   7256,   7269,   7282,   7295,   7307,   7319,   7331,
    7343,   7356,   7369,   7382,   7395,   7407,   7419,   7431,
    7443,   7456,   7469,   7482,   7495,   7507,   7519,   7531,
    7543,   7556,   7569,   7582,   7595,   7607,   7619,   7631,
    7643,   7656,   7669,   7682,   7695,   7707,   7719,   7731,
    7743,   7756,   7769,   7782,   7795,   7807,   7819,   7831,
    7843,   7856,   7869,   7882,   7895,   7907,   7919,   7931,
    7943,   7956,   7969,   7982,   7995,   8007,   8019,   8031,
    8043,   8056,   8069,   8082,   8095,   8107,   8119,   8131,
    8143,   8156,   8169,   8182,   8195,   8207,   8219,   8231,
    8243,   8256,   8269,   8282,   8295,   8307,   8319,   8331,
    8343,   8356,   8369,   8382,   8395,   8407,   8419,   8431,
    8443,   8456,   8469,   8482,   8495,   8507,   8519,   8531,
    8543,   8556,   8569,   8582,   8595,   8607,   8619,   8631,
    8643,   8656,   8669,   8682,   8695,   8707,   8719,   8731,
    8743,   8756,   8769,   8782,   8795,   8807,   8819,   8831,
    8843,   8856,   8869,   8882,   8895,   8907,   8919,   8931,
    8943,   8956,   8969,   8982,   8995,   9007,   9019,   9031,
    9043,   9056,   9069,   9082,   9095,   9107,   9119,   9131,
    9143,   9156,   9169,   9182,   9195,   9207,   9219,   9231,
    9243,   9256,   9269,   9282,   9295,   9307,   9319,   9331,
    9343,   9356,   9369,   9382,   9395,   9407,   9419,   9431,
    9443,   9456,   9469,   9482,   9495,   9507,   9519,   9531,
    9543,   9556,   9569,   9582,   9595,   9607,   9619,   9631,
    9643,   9656,   9669,   9682,   9695,   9707,   9719,   9731,
    9743,   9756,   9769,   9782,   9795,   9807,   9819,   9831,
    9843,   9856,   9869,   9882,   9895,   9907,   9919,   9931,
    9943,   9956,   9969,   9982,   9995,   10007,   10019,   10031,
    10043,   10056,   10069,   10082,   10095,   10107,   10119,   10131,
    10143,   10156,   10169,   10182,   10195,   10207,   10219,   10231,
    10243,   10256,   10269,   10282,   10295,   10307,   10319,   10331,
    10343,   10356,   10369,   10382,   10395,   10407,   10419,   10431,
    10443,   10456,   10469,   10482,   10495,   10507,   10519,   10531,
    10543,   10556,   10569,   10582,   10595,   10607,   10619,   10631,
    10643,   10656,   10669,   10682,   10695,   10707,   10719,   10731,
    10743,   10756,   10769,   10782,   10795,   10807,   10819,   10831,
    10843,   10856,   10869,   10882,   10895,   10907,   10919,   10931,
    10943,   10956,   10969,   10982,   10995,   11007,   11019,   11031,
    11043,   11056,   11069,   11082,   11095,   11107,   11119,   11131,
    11143,   11156,   11169,   11182,   11195,   11207,   11219,   11231,
    11243,   11256,   11269,   11282,   11295,   11307,   11319,   11331,
    11343,   11356,   11369,   11382,   11395,   11407,   11419,   11431,
    11443,   11456,   11469,   11482,   11495,   11507,   11519,   11531,
    11543,   11556,   11569,   11582,   11595,   11607,   11619,   11631,
    11643,   11656,   11669,   11682,   11695,   11707,   11719,   11731,
    11743,   11756,   11769,   11782,   11795,   11807,   11819,   11831,
    11843,   11856,   11869,   11882,   11895,   11907,   11919,   11931,
    11943,   11956,   11969,   11982,   11995,   12007,   12019,   12031,
    12043,   12056,   12069,   12082,   12095,   12107,   12119,   12131,
    12143,   12156,   12169,   12182,   12195,   12207,   12219,   12231,
    12243,   12256,   12269,   12282,   12295,   12307,   12319,   12331,
    12343,   12356,   12369,   12382,   12395,   12407,   12419,   12431,
    12443,   12456,   12469,   12482,   12495,   12507,   12519,   12531,
    12543,   12556,   12569,   12582,   12595,   12607,   12619,   12631,
    12643,   12656,   12669,   12682,   12695,   12707,   12719,   12731,
    12743,   12756,   12769,   12782,   12795,   12807,   12819,   12831,
    12843,   12856,   12869,   12882,   12895,   12907,   12919,   12931,
    12943,   12956,   12969,   12982,   12995,   13007,   13019,   13031,
    13043,   13056,   13069,   13082,   13095,   13107,   13119,   13131,
    13143,   13156,   13169,   13182,   13195,   13207,   13219,   13231,
    13243,   13256,   13269,   13282,   13295,   13307,   13319,   13331,
    13343,   13356,   13369,   13382,   13395,   13407,   13419,   13431,
    13443,   13456,   13469,   13482,   13495,   13507,   13519,   13531,
    13543,   13556,   13569,   13582,   13595,   13607,   13619,   13631,
    13643,   13656,   13669,   13682,   13695,   13707,   13719,   13731,
    13743,   13756,   13769,   13782,   13795,   13807,   13819,   13831,
    13843,   13856,   13869,   13882,   13895,   13907,   13919,   13931,
    13943,   13956,   13969,   13982,   13995,   14007,   14019,   14031,
    14043,   14056,   14069,   14082,   14095,   14107,   14119,   14131,
    14143,   14156,   14169,   14182,   14195,   14207,   14219,   14231,
    14243,   14256,   14269,   14282,   14295,   14307,   14319,   14331,
    14343,   14356,   14369,   14382,   14395,   14407,   14419,   14431,
    14443,   14456,   14469,   14482,   14495,   14507,   14519,   14531,
    14543,   14556,   14569,   14582,   14595,   14607,   14619,   14631,
    14643,   14656,   14669,   14682,   14695,   14707,   14719,   14731,
    14743,   14756,   14769,   14782,   14795,   14807,   14819,   14831,
    14843,   14856,   14869,   14882,   14895,   14907,   14919,   14931,
    14943,   14956,   14969,   14982,   14995,   15007,   15019,   15031,
    15043,   15056,   15069,   15082,   15095,   15107,   15119,   15131,
    15143,   15156,   15169,   15182,   15195,   15207,   15219,   15231,
    15243,   15256,   15269,   15282,   15295,   15307,   15319,   15331,
    15343,   15356,   15369,   15382,   15395,   15407,   15419,   15431,
    15443,   15456,   15469,   15482,   15495,   15507,   15519,   15531,
    15543,   15556,   15569,   15582,   15595,   15607,   15619,   15631,
    15643,   15656,   15669,   15682,   15695,   15707,   15719,   15731,
    15743,   15756,   15769,   15782,   15795,   15807,   15819,   15831,
    15843,   15856,   15869,   15882,   15895,   15907,   15919,   15931,
    15943,   15956,   15969,   15982,   15995,   16007,   16019,   16031,
    16043,   16056,   16069,   16082,   16095,   16107,   16119,   16131,
    16143,   16156,   16169,   16182,   16195,   16207,   16219,   16231,
    16243,   16256,   16269,   16282,   16295,   16307,   16319,   16331,
    16343,   16356,   16369,   16382,   16395,   16407,   16419,   16431,
    16443,   16456,   16469,   16482,   16495,   16507,   16519,   16531,
    16543,   16556,   16569,   16582,   16595,   16607,   16619,   16631,
    16643,   16656,   16669,   16682,   16695,   16707,   16719,   16731,
    16743,   16756,   16769,   16782,   16795,   16807,   16819,   16831,
    16843,   16856,   16869,   16882,   16895,   16907,   16919,   16931,
    16943,   16956,   16969,   16982,   16995,   17007,   17019,   17031,
    17043,   17056,   17069,   17082,   17095,   17107,   17119,   17131,
    17143,   17156,   17169,   17182,   17195,   17207,   17219,   17231,
    17243,   17256,   17269,   17282,   17295,   17307,   17319,   17331,
    17343,   17356,   17369,   17382,   17395,   17407,   17419,   17431,
    17443,   17456,   17469,   17482,   17495,   17507,   17519,   17531,
    17543,   17556,   17569,   17582,   17595,   17607,   17619,   17631,
    17643,   17656,   17669,   17682,   17695,   17707,   17719,   17731,
    17743,   17756,   17769,   17782,   17795,   17807,   17819,   17831,
    17843,   17856,   17869,   17882,   17895,   17907,   17919,   17931,
    17943,   17956,   17969,   17982,   17995,   18007,   18019,   18031,
    18043,   18056,   18069,   18082,   18095,   18107,   18119,   18131,
    18143,   18156,   18169,   18182,   18195,   18207,   18219,   18231,
    18243,   18256,   18269,   18282,   18295,   18307,   18319,   18331,
    18343,   18356,   18369,   18382,   18395,   18407,   18419,   18431,
    18443,   18456,   18469,   18482,   18495,   18507,   18519,   18531,
    18543,   18556,   18569,   18582,   18595,   18607,   18619,   18631,
    18643,   18656,   18669,   18682,   18695,   18707,   18719,   18731,
    18743,   18756,   18769,   18782,   18795,   18807,   18819,   18831,
    18843,   18856,   18869,   18882,   18895,   18907,   18919,   18931,
    18943,   18956,   18969,   18982,   18995,   19007,   19019,   19031,
    19043,   19056,   19069,   19082,   19095,   19107,   19119,   19131,
    19143,   19156,   19169,   19182,   19195,   19207,   19219,   19231,
    19243,   19256,   19269,   19282,   19295,   19307,   19319,   19331,
    19343,   19356,   19369,   19382,   19395,   19407,   19419,   19431,
    19443,   19456,   19469,   19482,   19495,   19507,   19519,   19531,
    19543,   19556,   19569,   19582,   19595,   19607,   19619,   19631,
    19643,   19656,   19669,   19682,   19695,   19707,   19719,   19731,
    19743,   19756,   19769,   19782,   19795,   19807,   19819,   19831,
    19843,   19856,   19869,   19882,   19895,   19907,   19919,   19931,
    19943,   19956,   19969,   19982,   19995,   20007,   200
```

显然,编写这样的 HLA 程序来产生数据要比手工输入(并且校验)简单得多。当然,有时甚至都不需要使用 HLA 来编写表产生程序。您还会发现使用 Pascal/Delphi、C/C++ 或者其他高级语言来编写这个程序会更加简单。因为这个程序只执行一次,所以这种表产生程序的性能并不是问题。如果使用高级语言来编写表产生程序会更简单,那么就一定要这样做。注意,HLA 有一个内置的解释器,可以用来简单地建立表,而不需要使用外部程序。要了解更多细节,请参见第 9 章。

只要运行了表产生程序,那么剩下的工作就是从文件(示例中的 `sines.hla`)中将表剪切和粘贴到实际使用表的程序中。

8.4.4 表查找的性能

在早期的 PC 上,表查找是进行高性能计算的一种首选方法。但是,随着新 CPU 的速度极大地超过了存储器,查找表的优点正在消退。今天,对于一个 CPU 来说,比主存快 10~100 倍已经很普遍了。因此,使用表查找不一定比使用机器指令作相同的计算更快。所以,探讨表查找什么时候才能体现更大的优点是值得一做的事情。

虽然 CPU 比主存快得多,但是片内的高速缓冲存储器子系统还是以接近 CPU 的速度运行着。因此,如果把表放在 CPU 上的高速缓冲存储器中,那么表查找还是高效的。这就意味着要使用表查找来获取好的性能,实现的方法只能是使用更小的表(因为在高速缓存中没有很多的空间),并且要使用其中的数据项会被经常访问的表(这样表才会留在高速缓存中)。关于高速缓冲存储器的操作以及优化对高速缓冲存储器的使用方法,请参考 *Write Great Code, Volume 1* (No. Starch Press) 一书或者本书的电子版本(<http://webster.cs.ucr.edu/>或 <http://www.artofasm.com/>)。

8.5 更多信息

HLA 标准库参考手册包含很多关于 HLA 标准库的扩展精度算术运算能力方面的信息。读者也可以查看一些 HLA 标准库例程的源代码,看看不同的扩展精度操作是如何进行的(这些操作在计算完毕后会适当地设置标志)。HLA 标准库源代码还包含扩展精度 I/O 操作的内容,这些操作在本章中并没有出现。

Donald Knuth 的 *The Art of Computer Programming, Volume Two: Seminumerical Algorithms* 一书介绍了关于十进制算术操作和扩展精度算术操作的很多有用信息。但这本书是通用的,并没有描述如何使用 x86 汇编语言来进行这些操作。

宏与 HLA 编译时语言



本章主要讨论 HLA 编译时语言(compile-time language, CTL)。其中包括关于宏(macro)的介绍,宏可能是 HLA 编译时语言中最重要的组件。很多人以宏处理能力的强弱来判断汇编器能力的强弱。当您阅读完这一章之后,可能就会认同 HLA 是世界上最强大的汇编器之一;在所有的计算机语言处理系统中,HLA 拥有最强大的宏处理机制。

9.1 编译时语言

HLA 实际上是混合在一个程序中的两种语言。其中,运行时语言(runtime language)是标准的 80x86/HLA 汇编语言。它被称为运行时语言,是因为所编写的程序是在执行可执行文件时运行的。而 HLA 中还包含另一种语言的解释器,这种语言就是 HLA 编译时语言,它是在 HLA 编译一个程序时运行的。CTL 程序的源代码嵌入在一个 HLA 汇编语言的源文件中;也就是说,HLA 源文件同时包含 HLA CTL 和运行时程序的指令。HLA 在编译过程中执行 CTL 程序,一旦 HLA 完成了编译,CTL 程序就结束了;CTL 应用程序并指令不是 HLA 生成的运行时可执行文件的一部分,尽管 CTL 应用程序也可以编写一部分的运行时程序。实际上,这也正是 CTL 的主要任务,如图 9-1 所示。

将两种不同的语言内置在同一编译器中看上去似乎很混乱,您甚至可能会质疑为什么会需要编译时语言。为了了解编译时语言的优点,我们来考虑下面的语句:

```
stdout.put("i32=", i32, " strVAR=", strVar, " charVar=", charVar, nl );
```

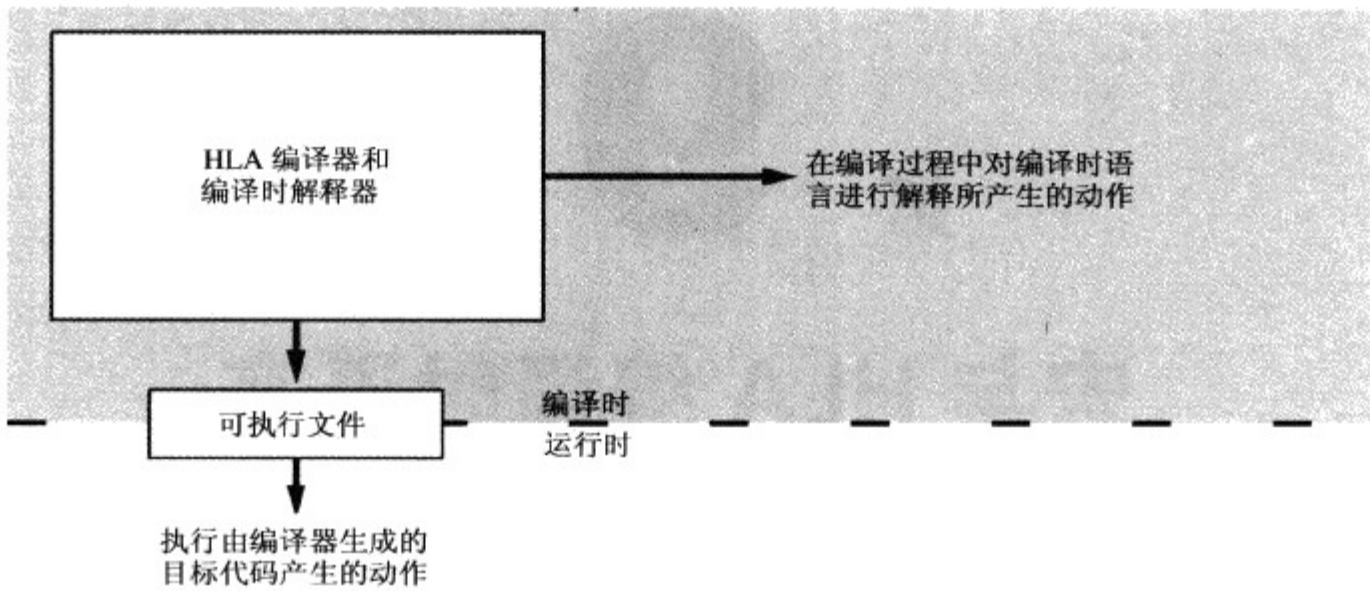


图 9-1 编译时与运行时的程序执行情况

这个语句既不是一个 HLA 语言的语句，也不是对 HLA 标准库过程的调用。stdout.put 实际上是由 HLA 标准库所提供的 CTL 应用程序中的一个语句。stdout.put “应用程序” 处理参数列表，并且发出不同标准库过程的调用；它根据当前处理的参数的类型选择将要调用的过程。例如，上面的 stdout.put “应用程序” 将在运行时可执行文件中生成下列语句：

```
stdout.puts( "i32=" );
stdout.puti32( i32 );
stdout.puts( "strVar=" );
stdout.puts( strVar );
stdout.puts( "charVar=" );
stdout.putc( charVar );
stdout.newln();
```

很明显，stdout.put 相对于它为了响应参数列表而生成的一系列语句来说，读写起来更简单。这正是 HLA 编程语言的一个强大功能：具有修改语言以简化常见编程任务的能力。将许多不同的数据对象以顺序的方式打印出来是一种普通的任务；stdout.put “应用程序” 极大地简化了这个过程。

HLA 标准库提供了很多 HLA CTL 示例。除了标准库的使用以外，HLA CTL 还擅长处理“一次性”或者“只使用一次”的应用程序。一个典型的示例是为一张查找表填写数据。本书第 8 章提示可以使用 HLA CTL 来构建查找表。实际上使用 HLA CTL 来构建查找表非常便捷。

虽然 CTL 本身相对来说是低效的，人们也不会使用它来编写用户端的应用程序，但它的确使时间得到了充分的利用。只要学会了如何使用 HLA CTL，并适当地加以应用，您就能以开发高级语言应用程序的速度来开发汇编语言应用程序(甚至更快，因为 HLA 的 CTL 可以用来创建非常高级的语言结构)。

9.2 #print 和#error 语句

回顾在第1章中，我们曾以一个典型的程序作为开始，它是大多数人在学习一种新的语言时所编写的第一个程序，这就是 Hello World 程序。本章在讨论另一种语言时，也将选用该程序，这是非常合适的。

程序清单 9-1 提供了用 HLA 编译时语言编写的 Hello World 程序。

程序清单 9-1 CTL 的 Hello World 程序

```
program ctlHelloWorld;
begin ctlHelloWorld;

    #print( "Hello, World of HLA/CTL" )

end ctlHelloWorld;
```

在这个程序中，唯一的 CTL 语句是 #print 语句，剩下的部分只是因为要使编译器顺利工作才编写的(尽管我们可以使用一个 unit 声明语句，而不是 program 声明语句，将代码减少到两行)。

在一个 HLA 程序的编译过程中，#print 语句显示了其参数列表的文本表示。因此，如果要使用命令 `hla ctlHW.hla` 来编译上面的程序，HLA 编译器就会立即打印出这些文本：

```
Hello, World of HLA/CTL
```

注意，在 HLA 源文件中，下面这两个语句有很大的不同：

```
#print( "Hello World" )
stdout.puts( "Hello World" nl );
```

第一个语句在编译过程中打印 Hello World(和一个新行)，它对可执行程序不起任何作用。第二个语句在编译过程中不起作用(除了生成可执行文件的代码以外)。然而，当运行可执行文件时，第二个语句将打印字符串 Hello World，并且其后跟随一个换行序列。

HLA/CTL 的 #print 语句使用下面的基本语法：

```
#print( list_of_comma_seperated_constants )
```

注意，这个语句不以分号结尾，分号只作为运行时语句的结尾；它们一般不作为编译时语句的结尾(但有一个明显的例外，这在稍后的内容中会提到)。

#print 语句必须至少有一个操作数；如果多个操作数出现在参数列表中，那么就必须使用逗号将这些参数彼此分开(就像 stdout.put 那样)。如果操作数不是一个字符串常量，那么 HLA 就会将这个常量翻译成对应的字符串表示，并打印这个字符串。例如：

```
#print( "A string Constant ", 45, ' ', 54.9, ' ', true )
```

可以指定已经被命名的符号常量和常量表达式。但是，所有的 #print 操作数都必须是常量(无论是字面常量还是在 const 或者 val 段中定义的常量)，并且这些常量必须在 #print 语句中使用之前就定义好。例如：

```
const
    pi := 3.14159;
    charConst := 'c';

#print( "PI = ", pi, "CharVal=", charConst )
```

HLA 的 `#print` 语句对于调试 CTL 程序来说尤为重要。这个语句对于显示编译进度以及显示假设和默认发生的动作来说,都是很有用的。除了显示与 `#print` 的参数列表相关的文本以外, `#print` 语句对程序的编译过程不起任何作用。

`#error` 语句允许使用一个单独的字符串常量操作数。像 `#print` 一样,这个语句将在编译过程中向控制台显示字符串。但是, `#error` 语句将字符串视为一个错误消息,并将其作为 HLA 错误诊断信息的一部分来显示。另外, `#error` 语句将会增加错误的计数,这会导致 HLA 在处理完当前的源文件后中止编译工作(而不进行汇编和连接)。通常当 CTL 代码发现有东西阻止其建立合法的代码时,就可以使用 `#error` 语句在编译过程中显示一个错误消息。例如:

```
#error( "Statement must have exactly one operand" )
```

像 `#print` 语句一样, `#error` 语句也不以分号结尾。虽然 `#error` 只允许使用一个字符串操作数,但如果使用编译时字符串串联操作符以及其他的 HLA 内置的编译时函数,那么要打印其他的值也是很简单的。本章稍后将介绍这些内容。

9.3 编译时常量和变量

正如运行时语言支持常量和变量一样,编译时语言也支持它们。可以在 `const` 段声明编译时常量,就像运行时语言的情况一样。还可以在 `val` 段声明编译时变量。虽然在运行时语言中, `val` 段声明的对象是常量,但要记住,在整个源文件中都可以修改这些在 `val` 段声明的对象的值,这样就有了术语“编译时变量”。要获得更多细节,请参见本书第 4 章。

CTL 赋值语句(?)将计算位于赋值操作符(`:=`)右边的常量表达式的值,并将结果存入赋值操作符左边的 `val` 对象中¹。这里的示例代码可以出现在 HLA 源文件中的任何地方,而不只是程序的 `val` 段中。

```
?ConstToPrint := 25;
#print( "ConstToPrint = ", ConstToPrint )
?ConstToPrint := ConstToPrint + 5;
#print( "Now ConstToPrint = ", ConstToPrint )
```

9.4 编译时表达式和操作符

HLA CTL 在 CTL 赋值语句中支持常量表达式。与运行时语言不同(必须将代数符号翻译成一系列的机器指令),HLA CTL 允许使用一整套算术操作,这些操作的表达式语法都是非常熟悉的。

¹ 如果赋值操作符左边的标识符是未定义的,那么 HLA 就会在当前作用域级别上自动声明这个对象。

这就赋予HLA CTL相当强的能力，特别是在结合了下一节将要讨论的内置编译时函数之后。

表 9-1 和表 9-2 列出了HLA CTL 在编译时表达式中支持的操作符。

表 9-1 编译时操作符

操 作 符	操作数类型*	描 述
- (一元)	numeric	将特定的数值(int、uns、real)取负
	cset	返回指定字符集的补码
!(一元)	integer	对操作数中的所有位取反(按位 not 操作)
	boolean	操作数的布尔 not 操作
*	numericL * numericR	将两个操作数相乘
	csetL * csetR	计算两个集合的交集
div	integerL div integerR	计算两个整数(int/uns/dword)操作数的整数商
mod	integerL mod integerR	计算两个整数(int/uns/dword)操作数相除的余数
/	numericL / numericR	计算两个数值操作数的实数商。即使两个操作数都是整数，也会返回实数结果
<<	integerL << integerR	将 integerL 操作数左移，integerR 操作数指定移动的位数
>>	integerL >> integerR	将 integerL 操作数右移，integerR 操作数指定移动的位数
+	numericL + numericR	将两个数字操作数相加
	csetL + csetR	计算两个集合的并集
	strL + strR	串联两个字符串
-	NumericL - numericR	计算 numericL 和 numericR 的差
	CsetL - csetR	计算 csetL 和 csetR 的差集
=或=	numericL = numericR	如果两个操作数具有相同的值，就返回 true
	csetL = csetR	如果两个集合相等，就返回 true
	strL = strR	如果两个字符串/字符相等，就返回 true
	typeL = typeR	如果两个数值相等，就返回 true。此时它们必定是相同的类型
<>或!=	typeL <> typeR(与 !=相同)	如果两个(兼容的)操作数(numeric、cset 或 string)彼此不相等，就返回 false
<	numericL < numericR	如果 numericL 小于 numericR，就返回 true
	csetL < csetR	如果 csetL 是 csetR 的真子集，就返回 true
	strL < strR	如果 strL 小于 strR，就返回 true
	booleanL < booleanR	如果左操作数小于右操作数，就返回 true(注意: false < true)
	enumL < enumR	如果 enumL 与 enumR 出现在同一枚举列表中，并且 enumL 出现在 enumR 之前，就返回 true

(续表)

操 作 符	操作数类型	描 述
<=	与<相同	如果左操作数小于或者等于右操作数，就返回 true。对于字符集来说，这就意味着左操作数是右操作数的子集
>	与<相同	如果左操作数大于右操作数，就返回 true。对于字符集来说，这就意味着左操作数是右操作数的真超集
>=	与<=相同	如果左操作数大于或者等于右操作数，就返回 true。对于字符集来说，这就意味着左操作数是右操作数的超集
&	integerL & integerR	计算两个操作数的按位 and
	booleanL & booleanR	计算两个操作数的逻辑 and
	integerL integerR	计算两个操作数的按位 or
	booleanL booleanR	计算两个操作数的逻辑 or
^	integerL ^ integerR	计算两个操作数的按位 xor
	booleanL ^ booleanR	计算两个操作数的逻辑 xor。注意，它与 booleanL <> booleanR 相同
in	charL in csetR	如果 charL 是 csetR 的一个成员，就返回 true

numeric 类型包括 intXX、unsXX、byte、word、dword 和 realXX。cset 是一个字符集操作数。integer 类型包括 intXX、unsXX、byte、word 和 dword。str 是字符串值或字符值。type 代表任意一种 HLEA 类型。其他类型指定一种显式的 HLEA 数据类型。

表 9-2 操作符的优先级和结合性

结 合 性	优先级(从高到低)	操 作 符
从右至左	6	!(一元)
		-(一元)
从左至右	5	*
		div
		mod
		/
		>>
		<<
从左至右	4	+
		-
从左至右	3	=或==
		<>或!=
		<
		<=
		>
		>=

(续表)

结 合 性	优先级(从高到低)	操 作 符
		\geq
从左至右	2	&
		^
无结合性	1	in

当然,在表达式中可以使用圆括号来重写操作符默认的优先级和结合性。

9.5 编译时函数

HLA 提供了使用范围很广的编译时函数。这些函数在编译时计算数值的方法就像高级语言的函数在运行时计算数值的方法一样。HLA 编译时语言包括多种数值、字符串和符号表函数,这些函数有助于编写复杂的编译时程序。

大多数内置的编译时函数的名称都以一个特殊的符号@作为开始,一般的形式就像@sin 或者@length 这样。使用这些特殊的标识符可以防止与那些可能在程序中使用的普通名称(如 length)发生冲突。剩下的编译时函数(不以@作为开始的函数)通常都是数据转换函数,它们使用像 int8 或者 real64 这样的类型名称。甚至可以使用宏来建立自己的编译时函数(9.8 节将会讨论)。

HLA 将编译时函数根据其操作类型组织在不同的类别中。例如,有些函数是将一个常量转换成另一个常量(比如说,字符串到整数的转换),还有很多很有用的字符串函数,并且 HLA 还提供了一整套编译时数值函数。

HLA 编译时函数的完整列表太长,这里难以给出。但是,对于 HLA 参考手册中出现的每一个编译时对象和函数的完整描述都可从 <http://webster.cs.ucr.edu/> 或 <http://www.artofasm.com/> 上获得;本节将着重介绍其中一些函数来演示它们的用法。稍后将大量使用各种编译时函数。

对于编译时函数,要理解的最重要的概念就是,它们在汇编语言代码(也就是,运行时程序)中等同于常量。例如,编译时函数调用@sin(3.14159265358979328)就大致等同于在程序中那一点指定 0.0。² 像@sin(x)这样的函数调用只有在 x 是一个常量并且在源文件中函数调用之前就已经定义过的情况下,才是合法的。特别是,x 不能是一个运行时变量,也不能是那种值存在于运行时(而不是编译时)的对象。因为 HLA 使用编译时函数的常量结果来替换这些函数,所以您可能会问为什么还要使用编译时函数。毕竟在通常情况下,在程序中键入 0.0 还是比键入@sin(3.1415265358979328)方便得多。但是编译时函数对于生成查找表,以及生成那些一旦在程序中修改了某个常量值,它们的值就发生变化的算术结果来说,是相当便利的。9.9 节将深入研究这些内容。

² 实际上,在这个示例中,因为@sin的参数并不正好是 π ,所以作为函数的结果,您会得到一个很小的正数,而不是0,然而理论上您应该得到0。

9.5.1 类型转换编译时函数

最常使用的编译时函数可能是类型转换函数,它们使用一个具有某种类型的参数,将其转换成另外一种特定的类型。这些函数使用一些 HLA 内置数据类型的名称作为函数的名称。这一类函数包括:

- `boolean`
- `int8`、`int16`、`int32`、`int64` 和 `int128`
- `uns8`、`uns16`、`uns32`、`uns64` 和 `uns128`
- `byte`、`word`、`dword`、`qword` 和 `lword`(这些分别等同于 `uns8`、`uns16`、`uns32`、`uns64` 和 `uns128`)
- `real32`、`real64` 和 `real80`
- `char`
- `string`
- `cset`
- `text`

这些函数都接受一个单独的常量表达式参数,并且只要这些参数合法,就将该表达式的值转换成由类型名称所指定的类型。例如,下面的函数调用将返回一个值 `-128`,因为它将字符串常量转换成了对应的整数值:

```
int8( "-128" )
```

一些转换过程是没有意义的,或者说受到与该过程相关的限制。例如, `boolean` 函数可以接受一个字符串参数,但那个字符串必须是 `true` 或者 `false`,否则这个函数就会产生一个编译时错误。同样,数字转换函数(如 `int8`)允许使用一个字符串操作数,但这个字符串操作数必须代表一个合法的数值。一些转换函数(例如,具有一个字符集参数的 `int8`)根本就是没有意义的,或者是非法的。

这一类函数当中最有意义的是 `string` 函数。这个函数几乎可以接受所有常量表达式的类型,并且可以产生一个代表其参数数据的字符串。例如,调用 `string(128)` 就会产生一个字符串 `128` 作为返回的结果。如果希望在 HLA 需要字符串的时候使用某个值,那么使用这个函数十分方便。例如, `#error` 编译时语句只允许一个字符串操作数,但可以使用 `string` 函数和字符串串联操作符(`+`)避开这一限制。例如:

```
#error( "theValue (" + string( theValue ) + ") is out of range" )
```

注意,这些类型函数实际上是进行转换操作的。这就意味着函数返回的位模式可能会与作为参数传递给它们的位模式不相同。例如,考虑下面的 `real32` 函数调用:

```
real32( $3F80_0000 )
```

`$3F80_0000` 是 `real32` 值 `1.0` 的十六进制形式。然而,上面这个函数却不返回 `1.0`,而是将整数值 `$3F80_0000`(`1065353216`)转换成一个 `real32` 值,但结果却失败了。因为这个值太大了,以至于不能够精确地使用 `real32` 对象来表示。与此对应,考虑下面的常量函数:

```
char( 65 )
```

这个 CTL 函数调用返回字符 A(因为 65 是 A 的 ASCII 码)。请注意 char 函数如何使用这个被作为 ASCII 码传递的整数参数的位模式,而 real32 函数则试图将这个整数参数转换成浮点值。虽然这两个函数的语义完全不同,但它们都试图进行直观的操作,甚至以牺牲一致性作为代价。

然而有时候,可能不想让这些函数去做直观的操作。例如,可能希望 real32 函数将一个传递给它的位模式作为 real32 值来对待。为了处理这种情况,HLA 提供了另一组类型函数,它们只是简单地以类型名称和一个@作为前缀来命名,而这些函数将参数作为最终类型的位模式来对待。所以如果真的想从 \$3F80_0000 得到 1.0,就可以使用下面的函数调用:

```
@real32( $3F80_0000 )
```

一般情况下,这种形式的类型强制转换在编译时语言中是高级功能,所以不会经常使用。然而,在需要的时候,了解它们还是很有益的。

9.5.2 数值编译时函数

这一类函数在编译时进行标准的数学运算。这些函数对于产生查找表,以及根据程序起始处定义的常量重新计算函数以“参数化”源代码来说,是非常便利的。这一类函数包括:

- @abs(*n*) 数字参数的绝对值。
- @ceil(*r*)、@floor(*r*) 提取浮点数的整数部分。
- @sin(*r*)、@cos(*r*)、@tan(*r*) 标准三角函数。
- @exp(*r*)、@log(*r*)、@log10(*r*) 标准的对数/指数函数。
- @min(*list*)、@max(*list*) 返回数值列表中的最小/最大值。
- @random、@randomize 返回一个伪随机 int32 数值。
- @sqrt(*n*) 计算其数值参数的平方根(结果为实型)。

关于这些函数的更多细节请参考 HLA 参考手册 (<http://webster.cs.ucr.edu/> 或 <http://www.artofasm.com/>)。

9.5.3 字符分类编译时函数

这一组函数都会返回布尔型的结果。它们对一个字符(或字符串中的所有字符)进行测试,看它是否属于某一类字符。这一类函数包括:

- @isAlpha(*c*)、@isAlphanum(*c*)
- @isDigit(*c*)、@isxDigit(*c*)
- @isLower(*c*)、@isUpper(*c*)
- @isSpace(*c*)

除了这些字符分类函数外,HLA 语言还提供了一组模式匹配函数,也可以使用它们来对字符和字符串数据进行分类。请参见 HLA 参考手册中关于这些例程的讨论。

9.5.4 编译时字符串函数

这一类函数对字符串参数进行操作。它们中的大多数函数会返回字符串的结果,但有一些(例如, @length 和 @index)会返回一个整数结果。这些函数不直接作用于它们的参数;而是返回一个适当的结果,如果愿意,可以将这个结果再次指派给参数。

- @delete、@insert
- @index、@rindex
- @length
- @lowercase、@uppercase
- @strbrk、@strspan
- @strset
- @substr、@tolenize、@trim

要得到关于这些函数及其参数和类型的更多细节,请参考 HLA 参考手册。注意,这些函数是 HLA 标准库中一些字符串函数的编译时等价形式。

@length 函数应该特别注意,因为它是这一类函数中最流行的函数。它返回一个 uns32 常量,指定其字符串参数中的字符个数。语法如下:

```
@length( string expression )
```

其中 *string_expression* 可以代表任何编译时字符串表达式。这个函数以字符个数为单位返回给定表达式的长度。

9.5.5 编译时符号信息

在编译过程中,HLA 保持着一个内部数据库,称为符号表(symbol table)。符号表包含许多有用的信息,用于说明在程序中某个给定的点之前定义的标识符。为了产生机器代码输出,HLA 需要查询这个数据库来确定如何处理特定的符号。在编译时程序中,经常需要查询符号表,来确定如何处理代码中的标识符或者表达式。HLA 编译时符号信息函数可以处理这种任务。

很多编译时符号信息函数都超出了本书所讨论的范围。本章只给出它们中的一部分。要得到编译时符号表函数的完整列表,请参考 HLA 参考手册。在本章中我们要考虑的函数包括:

- @size
- @defined
- @typeName
- @elements
- @elementSize

毫无疑问,@size 函数通常是这一组函数中最重要的函数。实际上,在前面的章节中已经使用过这个函数。@size 函数可以接受一个 HLA 标识符或者常量表达式作为参数。它以字节为单位返回对象(或表达式)所具有的数据类型的大小。如果给它提供一个标识符,那么该标识符可能是一个常量、类型或者变量标识符。正如在前面章节里看到的那样,这个函数对于使用 mem.alloc 来分配存储空间,或者为数组分配存储空间来说,是很有价值的。

另一个非常有用的函数是@defined 函数。这个函数可以接受一个 HLA 标识符作为参数,例如:

```
@defined( MyIdentifier )
```

如果标识符在程序中的某一点已经定义过,这个函数就会返回真;否则返回假。

@typeName 函数返回一个字符串,该字符串指定了作为参数提供给函数的标识符或者表达式的类型名称。例如,如果 i32 是一个 int32 对象,那么@typeName(i32)就会返回字符串 int32。这

个函数对于测试在编译时程序中所处理的对象类型很有用。

`@elements` 函数需要一个数组标识符或者表达式作为参数，它返回数组元素的元素总数作为函数的结果。注意，对于多维数组来说，这个函数将返回数组所有维数的乘积³。

`@elementsSize` 函数以字节为单位返回数组中一个元素的大小，这个数组的名称被作为参数传递给该函数。这个函数对于计算数组索引来说极其有用(也就是，这个函数将完成数组索引计算过程中 `element_size` 的计算工作；请参见第4章以获取更多信息)。

9.5.6 其他编译时函数

HLA 编译时语言包含一些其他的函数，它们并不属于前面所说的任何一类。其中一些非常有用的函数包括：

- `@odd`
- `@lineNumber`
- `@text`

`@odd` 函数将一个顺序值(也就是，非实数的数字或字符)作为参数，当这个值为奇数时返回真，为偶数时返回假。`@lineNumber` 函数不需要参数；它返回源文件中当前的行号。这个函数对于调试编译时(以及运行时)程序来说很有用。

`@text` 函数是这一组函数中最有用的。`@text` 函数需要一个字符串参数，它将其调用位置上的这个字符串以文本形式进行扩展。在结合使用编译时字符串处理函数时，这个函数相当有用。可以使用字符串操作函数建立一条指令(或指令的一部分)，然后使用 `@text` 函数将字符串转换成程序的源代码。下面是这个函数运用的一个简单示例：

```
?id1:string := "eax";
?id2:string := "i32";
@text( "mov( " + id1 + ", " + id2 + " );" )
```

上面这个代码序列将被编译成：

```
mov( eax, i32 );
```

9.5.7 编译时文本对象的类型转换

如果在程序中创建了一个文本常量，那么要操作这个对象就很困难。下面的示例演示了一个程序员想要改变程序中一个文本符号定义的过程：

```
val
    t:text := "stdout.put";
    .
    .
    .
    ?t:text := "fileio.put";
```

³ 有一个 `@dim` 函数，它返回一个数组，指定一个多维数组每一维的边界。如果对这个函数有兴趣，请参考 <http://webster.cs.ucr.edu/> 或 <http://www.artofasm.com/> 上的相关文档以获得更多细节。

这个示例中基本的思想是，在程序的前半部分中将符号 `t` 扩展成 `stdout.put`，在程序的后半部分扩展为 `fileio.put`。但是，这个简单的示例不能正常工作。问题在于，无论在哪里，只要发现了文本符号，HLA 就会对它进行扩展，这就包括在 `?语句` 中出现的 `t`。因此前面的代码被扩展成下面这段(不正确的)文本：

```
val
    t:text := "stdout.put";
    .
    .
    .
    ?stdout.put:text := "fileio.put";
```

HLA 不知道如何处理这个 `?语句`，所以就会产生一个语法错误。

有时可能不想让 HLA 对某个文本对象进行扩展，或者想要用自己的代码来处理由文本对象所持有的字符串数据。HLA 提供了两种方法来处理这两个问题：

- `@string(identifier)`
- `@toString:identifier`

对于 `@string(identifier)`，HLA 将返回与文本对象相关的文本数据所对应的一个字符串常量。换句话说，这个操作符可以处理一个文本对象，就像它是表达式中的一个字符串常量一样。

遗憾的是，`@string` 函数将一个文本对象转换成一个字符串常量，而不是字符串标识符。因此，不能这样使用：

```
?@string(t) := "Hello"
```

这样是不起作用的，因为 `@string(t)` 将用一个与文本对象 `t` 相关的字符串常量替换自己。如果之前给 `t` 进行赋值，这个语句被扩展成：

```
? "stdout.put" := "Hello";
```

该语句仍然是非法的。

`@toString:identifier` 操作符就是用来解决这种情况的。`@toString:`操作符需要一个文本对象作为相关的标识符。它将文本对象转换成一个字符串对象(但仍然保留相同的字符串数据)，并且返回该标识符。因为这个标识符现在已经是一个字符串对象了，所以就可以给它赋一个值(并且把它的类型改成其他某种类型，例如，可以改成 `text` 类型)。因此，要实现这个最初的目标，就要使用下面这样的代码：

```
val
    t:text := "stdout.put";
    .
    .
    .
    ?@toString:t:text := "fileio.put";
```

9.6 条件编译(编译时判定)

HLA 的编译时语言提供了一个 `if` 语句, `#if`, 它可以用来在编译时作出不同的决定。使用 `#if` 语句有两个主要的目的: `#if` 语句的传统用法是用于支持条件编译(conditional compilation), 或者说条件汇编(conditional assembly), 它使您能够在编译的时候, 根据程序中的不同符号或常量值的状态, 将某些代码包括进来或者排除出去。这个语句的另一种用法是用于支持 HLA 编译时语言中标准 `if` 语句的判断过程。本节将讨论 HLA `#if` 语句的这两种用法。

HLA 编译时 `#if` 语句最简单的形式使用下面的语法:

```
#if( constant_boolean_expression )
    << text >>
#endif
```

注意, 在 `#endif` 子句后并没有分号。如果在 `#endif` 语句后面写了一个分号, 那么它就变成了源代码的一部分, 这与在程序中接下来的数据项之前直接插入分号一样。

在编译时, HLA 将对 `#if` 之后括号里的表达式求值。这个表达式必须是一个常量表达式, 并且它的类型必须是布尔型。如果表达式求出的值为真, 那么 HLA 就继续处理源文件中的文本, 就像 `#if` 语句不存在一样。但是, 如果表达式求值为假, HLA 就会将 `#if` 和其对应的 `#endif` 子句之间的文本作为注释来对待(也就是说, 它将忽略这段文本), 如图 9-2 所示。

```
#if( constant_boolean_expression )
```

如果表达式为真, HLA 就会编译此代码。
否则, HLA 就将此代码作为注释来对待。

```
#endif
```

图 9-2 HLA 编译时 `#if` 语句的操作

要记住, HLA 的常量表达式支持一种全表达式语法, 就像在 C 或 Pascal 这样的高级语言中那样。`#if` 表达式的语法不局限于 HLA `if` 语句表达式所允许的语法。因此, 编写下面这样的表达式是完全合理的:

```
#if( @length( someStrConst ) < 10 * i & ( (MaxItems * 2 + 2) < 100 | MinItems - 5 < 10 ) )
    << text >>
#endif
```

还要记住, 编译时表达式中的元素必须都是 `const` 或者 `val` 标识符, 或者是一个 HLA 编译时函数调用(具有适当的参数)。特别要记住, HLA 在编译时对这些表达式进行求值, 所以它们不会包含运行时变量⁴。HLA 的编译时语言使用完整布尔求值, 所以表达式中出现任何一种副作用都会产生无法预料的结果。

⁴ 当然, 除了作为某些 HLA 编译时函数(例如 `@size` 或者 `@typeName`)的参数之外。

HLA `#if` 语句支持可选的 `#elseif` 和 `#else` 子句, 它们以一种直观的方式执行。`#if` 语句的完整语法就像下面这样:

```
#if( constant_boolean_expression_1 )
    << text >>
#elifif( constant_boolean_expression_2 )
    << text >>
#else
    << text >>
#endif
```

如果第一个布尔表达式的值为真, 那么 HLA 就会处理 `#elseif` 子句之前的文本。然后它会跳过接下来的所有文本(也就是说, 将它们作为注释来对待), 直至碰到 `#endif` 子句为止。在常见模式下, HLA 会继续处理 `#endif` 子句之后的文本。

如果上面的第一个布尔表达式求值为假, 那么 HLA 将会跳过接下来的所有文本, 直至碰到一个 `#elseif`、`#else` 或者 `#endif` 子句为止。如果它碰到一个 `#elseif` 子句(就像上面的情况), 那么 HLA 就会对与该子句相关的布尔表达式进行求值。如果求值为真, HLA 就会处理 `#elseif` 和 `#else` 子句之间的文本(或者如果 `#else` 子句不存在的话, 就一直处理到 `#endif` 子句)。在处理这段文本的过程中, 如果 HLA 碰到另一个 `#elseif`, 或者像上面这样碰到了另一个 `#else` 子句, 那么 HLA 就会忽略接下来的所有文本, 直到它发现对应的 `#endif`。

如果在前面的示例中, 第一个和第二个布尔表达式都求值为假, HLA 就会跳过与它们相关的文本, 开始处理 `#else` 子句中的文本。只要了解了 HLA 是如何“执行”这些语句的主体, 就会发现 `#if` 语句是以一种相对直观的方式运作的; `#if` 语句会对文本进行处理, 或者将其作为注释来对待, 这取决于布尔表达式的状态。当然, 通过在语句中加入零个或者更多 `#elseif` 子句, 以及有选择性地提供 `#else` 子句, 就可以建立几乎可以无限变化的各种 `#if` 语句序列。因为这个结构和 HLA 的 `if..then..elseif..else..endif` 语句是相同的, 所以这里没有必要再进一步讨论。

条件编译有一种非常传统的用法, 那就是用来开发可以针对几种不同环境进行简单配置的软件。例如, `fcomip` 指令使得浮点比较操作变得很容易, 但这条指令只在 Pentium Pro 及其以后的处理器上才能使用。如果想要在支持它的处理器上使用这条指令, 而在较老的处理器上使用标准浮点比较操作, 一般就要编写两种版本的程序——一种使用 `fcomip` 指令, 另一种使用传统的浮点比较代码序列。遗憾的是, 维护两种不同的源文件(一个针对新处理器, 一个针对旧处理器)是很困难的。大多数工程师喜欢使用条件编译将不同的代码序列嵌入同一个源文件中。下面的示例演示了如何完成这项工作:

```
const
    // Set true to use FCOMIxx instrs.
    PentProOrLater: boolean := false;

#if( PentProOrLater )

    fcomip();    // Compare st1 to st0 and set flags.
```

```

#else

    fcomp();           // Compare st1 to st0.
    fstsw( ax );       // Move the FPU condition code bits
    sahf();            // into the flags register.

#endif

```

就像现在编写的这样，这个代码段将编译 `#else` 子句中的 3 条指令序列，而忽略 `#if` 和 `#else` 子句之间的代码(因为常量 `PentProOrLater` 为假)。将 `PentProOrLater` 修改为真，就可以告诉 HLA 去编译单独的 `fcomip` 指令，而不是 3 条指令序列。当然，可以在程序的其他 `#if` 子句中使用 `PentProOrLater` 常量来控制 HLA 如何编译代码。

注意，条件编译不允许建立一个单独的可以在所有处理器上高效运行的可执行文件。在使用这种技术的时候，仍然必须将源文件编译两次，来建立两个可执行程序(一个针对 Pentium Pro 及以后的处理器，另一个针对之前的处理器)：在第一次编译的过程中，必须将 `PentProOrLater` 设置为假；在第二次编译的过程中，必须将这个常量再设置成真。虽然必须要建立两个不同的可执行文件，但只要保留一个源文件就可以了。

如果对其他语言(诸如 C/C++ 语言)中的条件编译比较熟悉，就可能想知道 HLA 是否支持类似 C 的 `#ifdef` 这样的语句。答案是否定的，它不支持。但是，可以使用 HLA 编译时函数 `@defined` 来简单地测试一个符号是否在源文件中前面的部分已经定义过。考虑下面这段代码，它对前面的代码作了修改，而且就使用了这种技术：

```

const
    // Note: uncomment the following line if you are compiling this
    // code for a Pentium Pro or later CPU.

    // PentProOrLater := 0; // Value and type are irrelevant.

#if( @defined( PentProOrLater ) )

    fcomip(); // Compare st1 to st0 and set flags.

#else

    fcomp(); // Compare st1 to st0.
    fstsw( ax ); // Move the FPU condition code bits
    sahf(); // into the flags register.

#endif

```

条件编译的另一种常见用法是为程序引入调试和测试代码。很多 HLA 程序员所使用的一种调试技术是在整个代码的关键点上插入“输出”语句；这就使他们能够跟踪代码，并在不同的检查点上显示重要的值。但是这种技术有一个很重要的问题，那就是在完成整个项目之前，必须删除这些调试代码。软件的客户(或者学生的导师)通常不想在程序生成的报表中看见这些调试输出。因此，使用这种技术的程序员倾向于暂时插入代码，然后在运行程序并确定哪里有问题后就删除这些代码。这种技术至少存在两个问题：

- 程序员经常会忘记删除某些调试语句，这将给最终的程序造成缺陷。
- 在删除了一条调试语句之后，程序员常常会发现他们在以后还需要使用相同的语句来调试不同的问题。这样他们会不断地插入、删除，再插入、再删除。

条件编译可以解决这个问题。在程序中定义一个符号(如 `debug`)来控制调试输出，通过简单地修改一行源代码就可以激活或禁用所有的调试输出。下面的代码段演示了这一功能。

```
const

// Set to true to activate debug output.
debug: boolean := false;

.
.
.

#if( debug )

    stdout.put( "At line ", @lineNumber, "i=", i, nl );

#endif
```

只要将所有的调试输出语句用一条 `#if` 语句包围起来，就不必担心调试输出会出现在最终的应用程序中。将 `debug` 符号设置成假，就可以自动禁止所有调试输出。同样，在调试语句完成使命之后，也不必从程序中将它们全部删除。通过使用条件编译，就可以将这些语句留在代码中，因为要使它们变得无效是非常简单的。以后，如果需要在程序编译的过程中显示这些调试信息，并不需要重新输入那些调试语句；只要简单地将 `debug` 符号设置为真就可以激活它们。

虽然程序配置和调试控制是条件编译最常见、最传统的两种用法，但不要忘了 `#if` 语句还为 HLA 编译时语言提供了基本的条件语句。在编译时程序中使用 `#if` 语句的方法和在 HLA 或其他语言中使用 `if` 语句是相同的。本书后面将给出大量使用 `#if` 语句的示例。

9.7 重复编译(编译时循环)

HLA 的 `#while..#endwhile` 和 `#for..#endfor` 语句提供了编译时循环结构。`#while` 语句告诉 HLA 在编译时重复处理相同的语句序列。这对于构造数据表来说是很便利的，它同样也可以为编译时程序提供一个传统的循环结构。虽然对于 `#while` 的使用不会像 `#if` 那样频繁，但在编写高级的 HLA 程序时，这种编译时的控制结构的确十分重要。

`#while` 语句使用下面的语法：

```
#while( constant_boolean_expression )
    <<_text >>
#endwhile
```

当 HLA 在编译过程中遇到 `#while` 语句时，它会对常量布尔表达式进行求值。如果该表达式求值为假，HLA 就会跳过 `#while` 和 `#endwhile` 子句之间的文本(这一行为与 `#if` 语句在表达式求值为假时相似)。如果该表达式求值为真，HLA 就会处理 `#while` 和 `#endwhile` 子句之间的文本，然后“跳回”到源文件中 `#while` 语句的起始处，并重复这个过程，如图 9-3 所示。

```
#while( constant_boolean_expression )
```

只要表达式为真，HLA 就会反复编译此代码，这样就将该语句序列的多个副本有效地插入源文件中(副本的确切数量取决于循环控制表达式的值)。

```
#endwhile
```

图 9-3 HLA 编译时 #while 语句的操作

为了了解这一过程是如何进行的，考虑程序清单 9-2 中的程序。

程序清单 9-2 #while..#endwhile 语句演示

```
program ctWhile;
#include( "stdlib.hhf" )

static
ary: uns32[5] := [ 2,3,5,8,13 ];

begin ctWhile;

    ?i := 0;
    #while( i < 5 )
        stdout.put( "array[", i, "] = ", ary[i*4], nl );
        ?i := i + 1;
    #endwhile
end ctWhile;
```

该程序的输出如下：

```
array[ 0 ] = 2
array[ 1 ] = 3
array[ 2 ] = 4
array[ 3 ] = 5
array[ 4 ] = 13
```

这里有一点非常不明显，就是该程序怎么会产生这样的输出。请记住，#while..#endwhile 结构是编译时语言的一个特性，而不是运行时的控制结构。因此，前面的 #while 循环在编译的过程中重复了 5 次。在循环的每次迭代过程中，HLA 编译器都会处理 #while 和 #endwhile 子句之间的语句。因此，前面所说的程序与程序清单 9-3 中的代码实际上是等同的。

程序清单 9-3 与程序清单 9-2 中的代码等同的程序

```
program ctWhile;
#include( "stdlib.hhf" )

static
```

```

    ary: uns32[5] := [ 2,3,5,8,13 ];

begin ctWhile;

    stdout.put( "array[ ", 0, " ] = ", ary[0*4], nl );
    stdout.put( "array[ ", 1, " ] = ", ary[1*4], nl );
    stdout.put( "array[ ", 2, " ] = ", ary[2*4], nl );
    stdout.put( "array[ ", 3, " ] = ", ary[3*4], nl );
    stdout.put( "array[ ", 4, " ] = ", ary[4*4], nl );

end ctWhile;

```

从以上示例可以看到, `#while` 语句在构造重复的代码序列方面非常方便。这对于循环展开来说特别有用。

HLA 提供了 3 种形式的 `#for..#endfor` 循环。这 3 种循环采取程序清单 9-4 所示的通用形式:

程序清单 9-4 HLA `#for` 循环

```

#for( valObject := startExpr to endExpr )
    .
    .
    .
#endfor

#for( valObject := startExpr downto endExpr )
    .
    .
    .
#endfor

#for( valObject in composite_expr )
    .
    .
    .
#endfor

```

顾名思义, `valObject` 必须是在某个 `val` 声明中定义过的对象。

对于前两种形式的 `#for` 循环来说, `startExpr` 和 `endExpr` 部分可以是任何一个产生整数值的 HLA 常量表达式。这些 `#for` 循环中的第一个循环在语义上等同于下面的 `#while` 代码:

```

?valObject := startExpr;
    #while( valObject <= endExpr )
        .
        .
        .
        ?valObject := valObject + 1;
    #endwhile

```

第二个 `#for` 循环等同于下面的 `#while` 循环:


```
?valObject := startExpr;
  #while( valObject >= endExpr )
    .
    .
    .
  ?valObject := valObject -1;
#endwhile
```

第三个 `#for` 循环(就是使用 `in` 关键字的循环)对于处理某些复合数据类型中的单个项来说特别有用。这个循环对指定给 `composite_expr` 的复合数值中的每个元素、字段、字符等重复一次。它可以是一个数组、字符串、记录,或者字符集表达式。对于数组来说,这个 `#for` 循环为数组的每个元素重复一次,在循环的每次迭代中,循环变量包含当前元素的值。例如,下面的编译时循环显示数值 1、10、100 和 1000:

```
#for(i in [1,10,100,1000])
  #print( i )
#endfor
```

如果 `composite_expr` 常量是一个字符串常量,那么 `#for` 循环就会为这个字符串中的每个字符重复一次,将循环控制变量设置为当前的字符。如果 `composite_expr` 常量表达式是一个记录常量,那么循环就会为记录的每个字段重复执行一次,在每次迭代中,循环控制变量将取得当前字段的类型和值。如果 `composite_expr` 表达式是一个字符集,那么循环就会为该集合中的每个字符重复一次,并将该字符赋给循环控制变量。

`#for` 循环实际上最终还是要比 `#while` 循环更有用,因为所遇到的大量编译时循环都会重复固定的次数(例如,处理一组固定数量的数组元素、宏参数等)。

9.8 宏(编译时过程)

宏是一组对象,在编译的时候,语言处理器会将其替换成其他的文本。宏具有使用简短的文本序列来替换冗长的重复出现的文本序列的强大功能。除了传统的功能(如 C/C++ 中的 `#define`)以外,HLA 的宏还可以充当编译时过程或者函数的等价物。因此,宏在 HLA 编译时语言中很重要——就像函数和过程在其他高级语言中很重要一样。

虽然宏没有什么新鲜的内容,但 HLA 的宏实现远远超过大多数其他编程语言(高级语言或低级语言)的宏处理能力。下面的部分将探讨 HLA 的宏处理机制,以及宏与其他 HLA CTL 控制结构之间的关系。

9.8.1 标准宏

HLA 支持直接的宏机制,可以以一种与过程声明相似的方式来定义宏。简单的宏定义通常采取如下形式:

```
#macro macroname;
  << Macro body >>
#endmacro;
```

虽然宏和过程声明类似,但从这个示例中可以看出,二者还是有一些明显的差异。首先,宏定义使用的是关键字`#macro`,而不是`procedure`。其次,宏的主体并不以一个`begin macroname`子句作为开始。最后,您会注意到宏是以`#endmacro`子句作为结尾的,而不是以`end macroname`结尾。下面的代码是关于宏定义的一个具体示例:

```
#macro neg64;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

#endmacro;
```

这段宏代码的执行将会计算 EDX:EAX 中 64 位数值的补码(请参见 8.1.7 节中关于扩展精度 `neg` 操作的相关描述)。

要执行与 `neg64` 相关的代码,只要在想要执行这些指令的地方简单地指定宏的名称即可。例如:

```
mov( (type dword i64), eax );
mov( (type dword i64[4]), edx );
neg64;
```

注意,宏的使用并不像过程调用那样,在宏的名称后面跟上一对空的圆括号(这样做的原因稍后会进行阐述)。

除了在 `neg64` 的调用⁵之后缺少圆括号之外,剩下的就像一次过程调用一样。可以使用下面的过程声明,将这个简单的宏实现为一个过程:

```
procedure neg64p;
begin neg64p;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

end neg64p;
```

注意,下面的两个语句都会将 EDX:EAX 中的值取负:

```
neg64;           neg64p();
```

二者(也就是宏调用和过程调用)的不同之处在于,宏只是内联地展开它们的文本,而过程调用会启动文本中其他某处的一个相关过程。也就是说,HLA 直接使用下面的文本来替换 `neg64` 调用:

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

⁵ 为了区别宏和过程,本书在描述宏的使用时将使用术语“调用(invocation)”,而在描述过程的使用时,将使用“调用(call)”。

另一方面, HLA 只用一条调用指令来替换过程调用 `neg64p();`:

```
call neg64p;
```

`neg64p` 过程应该是在程序中已经预先定义过的。

在宏和过程调用之间进行选择时要考虑效率。宏比过程调用稍快一些, 因为不需要执行 `call` 和相应的 `ret` 指令。而另一方面, 宏可能会使程序更长, 因为每次宏调用都会将宏扩展成宏主体的文本。而过程调用只是跳转至过程体的一个实例。因此, 如果宏的主体很长, 而且在程序中要调用很多的话, 就会使最终的可执行文件变得更长。另外, 如果宏的主体只不过是一些简单的指令, 那么使用一个 `call/ret` 序列对代码总的执行时间不会产生什么影响, 所以能够节省的执行时间可以忽略不计。另一方面, 如果过程体很短(就像上面的 `neg64` 示例那样), 您就会发现宏的实现会快很多, 而且也不会使程序增大太多。

注意:

对于简短的、对时间要求严格的程序单元, 使用宏。对于较长的代码块, 并且当不严格限制执行时间时, 使用过程。

宏与过程相比还有一些其他的缺点。宏不能使用局部(自动)变量, 宏的参数和过程参数的工作机制不同, 宏不支持(运行时)递归, 并且宏比过程要难以调试一些(这里只给出一些缺点)。因此, 不应该总是使用宏作为过程的替代者, 除非对性能的要求特别严格。

9.8.2 宏参数

像过程一样, 宏也允许定义参数, 使得在每次宏调用的时候能够支持不同的数据。这样就可以编写通用的宏, 其行为可以根据所提供的参数而变化。通过在编译时对这些宏参数的处理, 我们可以就编写出非常复杂的宏。

宏参数声明的语法非常直接, 只要在某个宏声明中的圆括号内提供一个参数名称的列表即可:

```
#macro neg64 ( reg32HO, reg32LO );
```

```
    neg ( reg32HO );
    neg ( reg32LO );
    sbb ( 0, reg32HO );
```

```
#endmacro;
```

注意, 这里并不像对待过程参数那样, 将宏参数与一种数据类型相关联。这是因为, HLA 的宏总是文本类型的对象。

在调用宏的时候, 只要提供实参即可, 就像过程调用一样:

```
neg64 ( edx, eax );
```

注意, 需要参数的宏调用要将整个参数列表都包含在圆括号中。

1. 标准宏参数扩展

正如前一节所说的那样, HLA 会自动将宏的参数与文本(text)类型关联起来。这就意味着在宏

的扩展过程中，HLA 将用所提供的文本来替换任何形参。因为“以文本替换的方式传递”的语义与“通过值传递”或“通过引用传递”的语义有些不同，所以需要解释一下。

考虑下面的宏调用，它使用了前面一节中的 `neg64` 宏：

```
neg64( edx, eax );
neg64( ebx, ecx );
```

这两个调用被扩展成下面的代码：

```
// neg64(edx, eax );

    neg( edx );
    neg( eax );
    sbb( 0, edx );

// neg64(ebx, ecx );

    neg( ebx );
    neg( ecx );
    sbb( 0, ebx );
```

注意，宏调用不会为参数生成本地副本(就像以值方式传递那样)，也不会将实参的地址传递给宏。像 `neg64(edx, eax);` 这种形式的宏调用跟下面的代码是等同的：

```
?reg32HO: text := "edx";
?reg32LO: text := "eax";

neg( reg32HO );
neg( reg32LO );
sbb( 0, reg32HO );
```

当然，文本对象将立即内联地扩展它们的字符串值，产生 `neg64(edx, eax);` 的扩展。

注意，宏的参数不像指令或者过程操作数那样限于存储器、寄存器或者常量操作数，只要文本的扩展在使用形参的地方合法即可。类似地，形参可能出现在宏主体中的任何地方，而不仅仅是那些存储器、寄存器或常量操作数都合法的地方。考虑下面的宏声明及其调用的示例：

```
#macro chkError( instr, jump, target );

    instr;
    jump target;

#endmacro;

chkError( cmp( eax, 0 ), jnl, RangeError );    // Example 1
...
chkError( test( 1, bl ), jnz, ParityError );    // Example 2

// Example 1 expands to

    cmp( eax, 0 );
    jnl RangeError;
```

```
// Example 2 expands to
```

```
test( 1, bl );
jnz ParityError;
```

一般情况下, HLA 假设所有位于逗号之间的文本构成一个单独的宏参数。如果 HLA 碰到开括弧符号(左圆括弧、左花括弧或者左方括弧)时, 它会把到对应的闭括弧符号之前的所有文本都包括进去, 而忽略这些括号之间的所有逗号。这就是为什么上面的 `chkError` 调用会将 `cmp(eax, 0)` 和 `test(1, bl)` 作为单独的参数而不是一对参数对待的原因。当然, HLA 不会将字符串常量中的逗号(和括号)作为实参的结束符号来考虑。所以下面的宏及其调用是完全合法的:

```
#macro print( strToPrint );

    stdout.out( strToPrint );

#endmacro;

.
.
.
print( "Hello, world!" );
```

HLA 将字符串 “Hello, world!” 视为一个单独的参数来对待, 这是因为逗号出现在一个字面字符串常量的内部。

如果对其他语言中文本的宏参数扩展不太熟悉, 也应该知道, 在 HLA 对宏的实参进行扩展时, 会遇到一些问题。考虑下面的宏声明及其调用:

```
#macro Echo2nTimes( n, theStr );
    #for( echoCnt := 1 to n*2 )
        #print( theStr )
    #endfor
#endmacro;

.
.
.
Echo2nTimes( 3+1, "Hello" );
```

这个示例将在编译的过程中显示 Hello 5 次, 而不是凭直觉所认为的 8 次。这是因为上面的 `#for` 语句被扩展成

```
#for( echoCnt := 1 to 3+1*2 )
```

`n` 的实参是 `3+1`; 因为 HLA 在 `n` 的位置上直接对这段文本进行扩展, 所以就得到了这种错误的文本扩展。当然, 在编译时 HLA 将 `3+1*2` 计算得到 5, 而不是 8(这个结果是在 HLA 将参数以值方式传递时得到的, 而不是以文本替换方式得到的)。

在传递可能包含编译时表达式的数值参数时, 通常将宏中的形参用圆括号括起来, 以解决这个问题; 例如, 可以将上面的宏重写如下:

```
#macro Echo2nTimes( n, theStr );
    #for( echoCnt := 1 to (n)*2 )
        #print( theStr )
    #endfor
#endmacro;
```

之前的调用将被扩展成下面的代码:

```
#for( echoCnt := 1 to (3+1)*2 )
    #print( theStr )
#endfor
```

这个版本的宏将产生直觉上预计的结果。

如果实参的个数与形参的个数不匹配,HLA 就会在编译过程中产生一个诊断消息。像过程那样,实参的个数必须同形参的个数保持一致。如果想有一个可选的宏参数,那么请继续往下读。

2. 可变参数个数的宏

现在您可能已经注意到,一些 HLA 宏并不需要固定的参数个数。例如,HLA 标准库里的 `stdout.put` 宏就允许使用一个或多个实参。HLA 使用一种特殊的数组语法来告诉编译器,希望在宏的参数列表中允许出现可变个数的参数。如果在形参列表中最后一个宏参数后面跟上一个[],那么 HLA 就允许可变个数的实参(零个或多个)代替形参。例如:

```
#macro varParms( varying[] );
    << Macro body >>
#endmacro;

varParms( 1 );
varParms( -1, 2 );
varParms( 1, 2, 3 );
varParms();
```

请特别注意最后一个调用。如果宏具有形参,那么就必须在宏调用后面提供一个带括号的参数列表。即使只为具有可变参数列表的宏提供 0 个实参,也必须这么做。要记住,这是没有参数的宏和具有可变参数列表但没有实参的宏的一个重要区别。

当碰到宏的一个形参带有后缀[]时(这肯定是形参列表中的最后一个参数),HLA 会建立一个常量字符串数组,并使用与在宏调用中剩下的实参相关的文本将这个数组初始化。使用 `@elements` 编译时函数可以确定指派给这个数组的实参个数。例如, `@element(varying)` 会返回一些值,0 或者更大的值,来指定与这个参数相关的参数的个数。下面 `varParms` 的声明演示了如何使用这个函数:


```

macro varParms( varying[] );

    #for( vpCnt := 0 to @elements( varying ) - 1 )

        #print( varying[ vpCnt ] )

    #endfor

#endmacro;

.
.

varParms( 1 );           // Prints "1" during compilation.
varParms( 1, 2 );        // Prints "1" and "2" on separate lines.
varParms( 1, 2, 3 );      // Prints "1", "2", and "3" on separate lines.
varParms();              // Doesn't print anything.

```

因为 HLA 不允许文本对象的数组，所以参数 `varying` 必须是一个字符串数组。遗憾的是，这就意味着必须将参数 `varying` 区别于标准的宏参数来对待。如果想让 `varying` 字符串数组中的某些元素扩展为宏主体内部的文本，通常可以使用 `@text` 函数来实现。相反，如果想使用一个不变的形参作为字符串对象，通常可以使用 `@string(name)` 函数。下面的示例作了演示：

```

macro ReqAndOpt( Required, optional[] );
    ?@text( optional[0] ) := @string( ReqAndOpt );
    #print( @text( optional[0] ) )

#endmacro;

.
.

ReqAndOpt( i, j );

// The macro invocation above expands to

    ?@text( "j" ) := @string( i );
    #print( "j" )

// The above further expands to

    j := "i";
    #print( j )

// The above simply prints "i" during compilation.

```

当然，在像上面这样的宏中，在试图引用参数 `optional` 的 0 号元素之前，应该先验证是否至少有两个参数。可以像下面这样简单地完成：

```

macro ReqAndOpt( Required, optional[] );

    #if( @elements( optional ) > 0 )

        ?@text( optional[0] ) := @string( ReqAndOpt );
        #print( @text( optional[0] ) )

    #else

```

```

        #error( "ReqAndOpt must have at least two parameters" )

    #endif

#endmacro;

```

3. 必需的与可选的宏参数

正如前一节中提到的, HLA 对于每一个不变的宏形参来说, 只需要一个实参。如果没有变化的宏参数(最多也只能有一个), 那么实参的个数必须与形参的个数完全匹配。如果出现了变化的形参, 那么实参个数就至少应该和不变(或者必需的)的宏形参一样多。如果只有单独一个变化的实参, 那么一次宏调用就可能有 0 个或多个实参。

没有参数的宏调用和具有单独一个可变的参数但没有实参的宏调用之间一个很大的区别在于: 具有变化的参数列表的宏必须在它的后面有一组空的圆括号, 而没有任何参数的宏调用不允许这样做。如果希望编写一个没有任何参数的宏, 但又想在这个宏的调用之后加上(), 以便与没有参数的过程调用相匹配, 就可以利用上述事实。考虑下面的宏:

```

#macro neg64( JustForTheParens[] );

    #if( @elements( JustForTheParens ) = 0 )

        neg( edx );
        neg( eax );
        sbb( 0, edx );

    #else

        #error( "Unexpected operand(s)" )

    #endif

#endmacro;

```

前面的宏要求像 `neg64()`; 这种形式的调用所使用的语法和过程调用的相同。如果想让无参数宏调用的语法跟无参数过程调用的语法相匹配, 这一特性很有用。这样做也并非不行, 只是需要将宏转换成过程的机会很少(反之亦然)。

9.8.3 宏中的局部符号

考虑下面的宏声明:

```

macro JZC( target );

    jnz NotTarget;
    jc target;

    NotTarget:

endmacro;

```

这个宏的目的是模拟一条指令, 在零标志和进位标志被设置时, 跳转到指定的目标位置。相反, 如果零标志或者进位标志已被清零, 那么这个宏就将控制立刻传递给宏调用下面的指令。

这个宏存在一个严重的问题。考虑在程序中多次使用这个宏时会发生什么：

```
JZC( Dest1 );
.
.
.
JZC( Dest2 );
.
.
.
```

前面的宏调用将扩展成下面的代码：

```
    jnz NotTarget;
    jc Dest1;
NotTarget:
    .
    .
    .
    jnz NotTarget;
    jc Dest2;
NotTarget:
    .
    .
    .
```

这两个宏调用的扩展出现的问题是，它们都会在扩展过程中生成同一个标号 `NotTarget`。当 HLA 处理这段代码时，它就会抱怨说发现了重复的符号定义。因此，在宏内部定义符号时必须要小心，因为宏的多次调用会导致该符号的多重定义。

HLA 对于这个问题的解决方法是允许在宏内部使用局部符号(local symbol)。局部符号对于宏的某次调用来说是独一无二的。例如，前面所说的 JZC 宏调用中，如果 `NotTarget` 是一个局部符号，程序就能够正确编译，因为 HLA 会将 `NotTarget` 的每次出现都视为独一无二的符号来对待。

HLA 不会自动将内部的宏符号定义变成局部的⁶。相反，必须显式告诉 HLA 哪个符号必须是局部的。使用下面这种通用的语法就可以在宏定义中完成这个工作：

```
#macro macroname( optional_parameters ):optional_list_of_local_names ;
    << Macro body >>
#endmacro;
```

局部名称的列表是由一个或多个用逗号隔开的 HLA 标识符组成的序列。每当在某个特定宏调用中碰到这个名称，HLA 都会自动使用一个独一无二的名称来替换这个标识符。对于每次宏调用来说，HLA 都会对局部符号使用不同的名称来替换。

可以使用下面的宏代码来修正 JZC 宏的这个问题：

```
#macro JZC( target ):NotTarget;

    jnz NotTarget;
```

⁶ 有时确实希望这个符号是全局的。


```

        jc target;
NotTarget:

#endmacro;

```

现在每当 HLA 处理这个宏时,都会自动将出现的每个 NotTarget 关联上一个独一无二的符号。如果没有将 NotTarget 声明为局部符号,就能防止出现符号重复错误。

在某次宏调用中,无论何处出现了局部符号,HLA 都会使用 `_nnnn_`(其中 `nnnn` 是一个四位的十六进制数字)形式的符号来替换局部符号。例如,形如 `JZC(SomeLabel)` 的宏调用可以扩展为:

```

        jnz _010A_;
        jc SomeLabel;
_010A_:

```

对于每个出现在宏扩展内部的符号,HLA 将递增这个数值,从而为需要的每个新的局部符号产生一个独一无二的临时标识符。只要不显式地创建形如 `_nnnn_Text_` 的标签(其中 `nnnn` 是一个十六进制数字),程序中就不会有冲突出现。HLA 将所有以下划线开始和结尾的符号保留起来自己使用(也可以被 HLA 标准库使用)。只要遵守了这个限制,在 HLA 局部符号和程序本身存在的标号之间就不会出现冲突,因为所有由 HLA 产生的符号都以下划线作为开始和结束。

HLA 通过有效地将局部符号转换成一个文本常量来实现局部符号,这里的文本常量将扩展成由 HLA 为局部符号所产生的一个独一无二的符号。也就是说,HLA 像下面示例中那样对待局部符号声明:

```

#macro JZC( target );
    ?NotTarget:text := "_010A_Text_";

    jnz NotTarget;
    jc target;

    NotTarget:
#endmacro;

```

HLA 无论何时对这个宏进行扩展,都会使用 `_010A_Text_` 来替换在扩展过程中每次出现的 NotTarget。这样的类比并不恰当,因为在这个例子中,文本符号 NotTarget 在宏扩展后仍然可以访问,而在宏内部定义局部符号时并不是这样。但是这确实体现了 HLA 实现局部符号的思想。

9.8.4 作为编译时过程的宏

虽然程序员在通常情况下使用宏来扩展一些机器指令,但宏的主体并不要求必须包含可执行指令。实际上,很多宏只包含编译时语言的语句(例如, `#if`、`#while`、`#for`、`?赋值语句`等)。如果在宏里只放入编译时语言的语句,那么就可以使用宏高效地编写编译时过程和函数。

下面的宏 `unique` 是一个很好的示例,它是一个编译时函数,返回一个字符串结果。考虑这个宏的定义:

```

#macro unique:theSym;
    @string(theSym)
#endmacro;

```

当代码引用这个宏的时候,HLA 都会使用文本 `@string(theSym)` 来替换宏的调用,当然,这段文本将被扩展成 `021F_Text` 这样的字符串。因此,可以将这个宏作为一个返回字符串结果的编译时函数。

要小心的是,不能将这个函数类比推广太多。要记住,宏总是会在被调用时对其主体的文本进行扩展,而一些扩展可能在程序中是不合法的。所幸的是,程序中凡是空白符可以合法出现的地方,大多数编译时语句都可以合法地出现。因此,在编译时程序的执行过程中,宏的行为可以理解为像函数或者过程那样。

当然,过程和函数之间的唯一区别在于,函数会返回一些显式的值,而过程只是进行一些活动而已。并没有特殊的语法来指定编译时函数返回值,如上例所示,只需要简单地将想要返回的值指定为宏主体中的一个语句就足够了。另一方面,编译时过程不会包含非编译时语言的语句,这样的语句会在宏调用的过程中扩展成某种数据。

9.8.5 使用宏模拟函数重载

C++语言支持一种极好的特性,称为函数重载(function overloading)。函数重载使您能够编写几种不同的函数或者过程,而这些函数和过程具有相同的名称。这些函数之间的区别在于它们的参数类型或者参数个数不同。如果一个过程的参数个数与其他同名函数的参数个数不同,或者它的参数类型和其他同名函数的参数类型不同,那么它的声明在 C++ 中就是独一无二的。HLA 并不直接支持这种函数重载,但可以使用宏来完成同样的事情。本节将说明如何使用 HLA 的宏和编译时语言来实现函数/过程的重载。

过程重载的一种很好的用途就是减少那些必须记住用法的 HLA 标准库例程的个数。例如,HLA 标准库提供了 5 个不同的 `puti` 例程,用于输出一个整数值: `stdout.puti128`、`stdout.puti64`、`stdout.puti32`、`stdout.puti16` 和 `stdout.puti8`。顾名思义,这些例程根据它们的整数参数的大小来输出整数值。在 C++ 语言(或者其他某种支持过程/函数重载的语言)中,设计输入例程的工程师通常会将它们都命名为 `stdout.puti`,而让编译器去基于操作数的大小选择适当的例程⁷。程序清单 9-5 中的宏演示了在 HLA 中如何使用编译时语言计算参数操作数的大小,从而实现重载。

程序清单 9-5 基于操作数大小的简单过程重载

```
// Puti.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti" macro that calls stdout.puti8, stdout.puti16,
// stdout.puti32, or stdout.puti64, depending on the size of the operand.

program putiDemo;
#include( "stdlib.hhf" )

// puti-
//
// Automatically decides whether we have a 64-, 32-, 16-, or 8-bit
```

⁷ 顺便说一下,HLA 标准库能很好地完成这一工作。虽然它不提供 `stdout.puti`,但它提供了 `stdout.put`,这个例程会基于参数类型选择一个适当的输出例程。这比 `puti` 例程要灵活一点。

```
// operand and calls the appropriate stdout.putiX routine to
// output this value.

#define puti( operand );

    // If we have an 8-byte operand, call puti64:

    #if( @size( operand ) = 8 )

        stdout.puti64( operand );

    // If we have a 4-byte operand, call puti32:

    #elseif( @size( operand ) = 4 )

        stdout.puti32( operand );

    // If we have a 2-byte operand, call puti16:

    #elseif( @size( operand ) = 2 )

        stdout.puti16( operand );

    // If we have a 1-byte operand, call puti8:

    #elseif( @size( operand ) = 1 )

        stdout.puti8( operand );

    // If it's not an 8-,4-,2-,or 1-byte operand,
    // then print an error message:

    #else

        #error( "Expected a 64-,32-,16-,or 8-bit operand" )

    #endif

#undef puti;

// Some sample variable declarations so we can test the macro above:

static
    i8:  int8      := -8;
    i16: int16     := -16;
    i32: int32     := -32;
    i64: qword;

begin putiDemo;

    // Initialize i64 since we can't do this in the static section.

    mov( -64, (type dword i64) );
    mov( $FFFF_FFFF, (type dword i64[4]) );

    // Demo the puti macro:

    puti( i8 ); stdout.newln();
```



```

    puti( i16 ); stdout.newln();
    puti( i32 ); stdout.newln();
    puti( i64 ); stdout.newln();

end putiDemo;

```

上面的示例简单地测试了操作数的大小来确定应该使用哪一个输出例程。还可以使用其他的 HLA 编译时函数，例如 `@typename`，来进行更复杂的处理。考虑程序清单 9-6 中的程序，它演示了一个根据操作数的类型重载 `stdout.puti32`、`stdout.putu32` 和 `stdout.putd` 的宏。

程序清单 9-6 基于操作数类型的过程重载

```

// put32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a put32 macro that calls stdout.puti32, stdout.putu32,
// or stdout.putdw depending on the type of the operand.

program put32Demo;
#include( "stdlib.hhf" )

// put32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

#macro put32( operand );

// If we have an int32 operand, call puti32:

    #if( @typename( operand ) = "int32" )

        stdout.puti32( operand );

// If we have an uns32 operand, call putu32:

    #elseif( @typename( operand ) = "uns32" )

        stdout.putu32( operand );

// If we have a dword operand, call putd32:

    #elseif( @typename( operand ) = "dword" )

        stdout.putd32( operand );

// If it's not a 32-bit integer value, report an error:

    #else

        #error( "Expected an int32, uns32, or dword operand" )
    #endif
#endmacro

```

```
#endif

#endmacro;

// Some sample variable declarations so we can test the macro above:

static
    i32: int32    := -32;
    u32: uns32    := 32;
    d32: dword    := $32;

begin put32Demo;

    // Demo the put32 macro:

    put32( d32 ); stdout.newln();
    put32( u32 ); stdout.newln();
    put32( i32 ); stdout.newln();

end put32Demo;
```

可以将这个宏简单地进行扩展,使其不仅能够输出 32 位操作数,还能输出 8 位和 16 位操作数。这个问题留作读者的练习。

实参的个数是决定调用哪个重载过程的另一种途径。如果指定了一个可变个数的宏参数(使用语法[], 参见本章前面的讨论),那么就可以使用@elements 编译时函数来准确地确定有多少个参数,然后调用适当的例程。程序清单 9-7 中的例子使用了这个技巧来确定是应该调用 stdout.puti32 还是调用 stdout.puti32Size。

程序清单 9-7 使用参数个数来决定调用哪个被重载的过程

```
// puti32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a puti32 macro that calls stdout.puti32 or stdout.puti32size
// depending on the number of parameters present.

program puti32Demo;
#include( "stdlib.hhf" )

// puti32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

#macro puti32( operand[] );

    // If we have a single operand, call stdout.puti32:

    #if( @elements( operand ) = 1 )
```

```

        stdout.puti32( @text( operand[0] ));

// If we have two operands, call stdout.puti32size and
// supply a default value of ' ' for the padding character:

#elseif( @elements( operand )= 2 )

        stdout.puti32Size
        (
            @text( operand[0] ),
            @text( operand[1] ),
            ' '
        );

// If we have three parameters, then pass all three of them
// along to puti32size:

#elseif( @elements( operand )= 3 )

        stdout.puti32Size
        (
            @text( operand[0] ),
            @text( operand[1] ),
            @text( operand[2] )
        );

// If we don't have one, two, or three operands, report an error:

#else

        #error( "Expected one, two, or three operands" )

#endif

#endmacro;

// A sample variable declaration so we can test the macro above:

static
    i32: int32 := -32;

begin puti32Demo;

    // Demo the put32 macro:

    puti32( i32 ); stdout.newln();
    puti32( i32, 5 ); stdout.newln();
    puti32( i32, 5, '*' ); stdout.newln();

end puti32Demo;

```

到现在为止，所有的示例都提供了标准库例程的过程重载(特别是整数输出例程)。当然，并不局限于重载 HLA 标准库中的过程，还可以创建自己的重载过程。只须编写一组过程，每个过程赋予一个唯一的名称，然后使用一个宏，并基于宏的参数来决定实际调用哪一个例程。不用去调用每一个具体的例程，只要调用通用的宏，让它来决定实际调用哪个过程即可。

9.9 编写编译时“程序”

HLA 编译时语言提供了一种强有力的机制，可以用来编写“程序”，这种“程序”是在 HLA 编译汇编语言程序的时候执行。虽然可以使用 HLA 编译时语言来编写一些通用程序，但 HLA 编译时语言真正的目的在于编写那些短小的、编写其他程序的程序。特别地，HLA 编译时语言的主要目的是使大型的复杂的汇编语言序列的创建过程变成自动化的。下面的小节提供了这种编译时程序的一些简单示例。

9.9.1 在编译时构造数据表

早些时候，本书曾建议编写程序来为汇编语言程序生成大型复杂的查找表(请参见 8.4.3 节中关于表的讨论)。第 8 章提供了一些 HLA 示例，但指出编写单独的程序是不必要的。的确是这样；仅仅使用 HLA 的编译时语言机制就可以产生大多数所需的查找表。实际上，填写表项是 HLA 编译时语言的一种主要用途。在本节中，我们将看一看如何在编译过程中使用 HLA 的编译时语言来构造数据表。

在 8.4.3 节中，曾有一个 HLA 程序示例，该程序对一个文本文件进行操作，而这个文件包含了一张关于正弦函数的查找表。表中包含 360 项以及它们的索引，这些索引以度数为单位指定了所有的角。表中每个 int32 项都包含值 $\sin(\text{angle}) \times 1000$ ，其中 *angle* 等于索引的值。8.4.3 节曾建议运行这个程序，然后将程序输出的文本包含在实际使用表的程序当中。使用编译时语言，就可以省去许多工作。程序清单 9-8 中的 HLA 程序包含一个简短的编译时代码段，用于直接构造这个正弦函数表。

程序清单 9-8 使用编译时语言产生一张正弦函数查找表

```
// demoSines.hla
//
// This program demonstrates how to create a look-up table
// of sine values using the HLA compile-time language.

program demoSines;
#include( "stdlib.hhf")

const
    pi :real80 := 3.1415926535897;

readonly
    sines: int32[ 360 ] :=
    [
        // The following compile-time program generates
        // 359 entries (out of 360). For each entry
        // it computes the sine of the index into the
        // table and multiplies this result by 1000
        // in order to get a reasonable integer value.

        ?angle := 0;
```

```

        #while( angle < 359 )

            // Note: HLA's @sin function expects angles
            // in radians. radians = degrees*pi/180.
            // The int32 function truncates its result,
            // so this function adds 1/2 as a weak attempt
            // to round the value up.

            int32( @sin( angle * pi / 180.0 ) * 1000 + 0.5 );
            ?angle := angle + 1;

        #endwhile

        // Here's the 360th entry in the table. This code
        // handles the last entry specially because a comma
        // does not follow this entry in the table.

        int32( @sin( 359 * pi / 180.0 ) * 1000 + 0.5 )
    ];

begin demoSines;
    // Simple demo program that displays all the values in the table:

    for( mov( 0, ebx ); ebx < 360; inc( ebx ) ) do

        mov( sines[ ebx * 4 ], eax );
        stdout.put
        (
            "sin(",
            (type uns32 ebx ),
            ")*1000 = ",
            (type int32 eax ),
            nl
        );

    endfor;

end demoSines;

```

编译时语言的另一种常见用途是建立 ASCII 字符查找表，用于运行时的 xlat 指令。常见示例包括字母大小写操作的查找表。程序清单 9-9 演示了如何构造大写转换表和小写转换表⁸。注意，其中将宏作为编译时过程来使用，从而降低了产生表的代码的复杂性。

程序清单 9-9 使用编译时语言生成大小写转换表

```

// demoCase.hla
//
// This program demonstrates how to create a lookup table
// of alphabetic case conversion values using the HLA
// compile-time language.

```

⁸ 注意，在现代的处理器上，使用一张查找表可能并不是进行字母大小写转换最高效的方法。但是，这的确是一个关于使用编译时语言填表的示例。其主要思想是正确的，即使其代码并不是所能达到的最好状态。

```

program demoCase;
#include( "stdlib.hhf" )

const

    // emitCharRange
    //
    // This macro emits a set of character entries
    // for an array of characters. It emits a list
    // of values (with a comma suffix on each value)
    // from the starting value up to, but not including,
    // the ending value.

    #macro emitCharRange( start, last ): index;

        ?index:uns8 := start;
        #while( index < last )

            char( index ),
            ?index := index + 1;

        #endwhile

    #endmacro;

readonly

// toUC:
// The entries in this table contain the value of the index
// into the table except for indices #$61..#$7A (those entries
// whose indices are the ASCII codes for the lowercase
// characters). Those particular table entries contain the
// codes for the corresponding uppercase alphabetic characters.
// If you use an ASCII character as an index into this table and
// fetch the specified byte at that location, you will effectively
// translate lowercase characters to uppercase characters and
// leave all other characters unaffected.

toUC: char[ 256 ] :=
    [
        // The following compile-time program generates
        // 255 entries (out of 256). For each entry
        // it computes toupper( index ) where index is
        // the character whose ASCII code is an index
        // into the table.

        emitCharRange( 0, uns8('a') )

        // Okay, we've generated all the entries up to
        // the start of the lowercase characters. Output
        // uppercase characters in place of the lowercase
        // characters here.

        emitCharRange( uns8('A'), uns8('Z') + 1 )
    ]

```



```

        // Okay, emit the nonalphabetic characters
        // through to byte code #$FE:

        emitCharRange( uns8('z') + 1, $FF )

        // Here's the last entry in the table. This code
        // handles the last entry specially because a comma
        // does not follow this entry in the table.

        #$FF

    ];

    // The following table is very similar to the one above.
    // You would use this one, however, to translate uppercase
    // characters to lowercase while leaving everything else alone.
    // See the comments in the previous table for more details.

    Tolc: char[ 256 ]:=
    [
        emitCharRange( 0, uns8('A'))
        emitCharRange( uns8('a'), uns8('z') + 1 )
        emitCharRange( uns8('Z') + 1, $FF )

        #$FF
    ];

begin demoCase;

    for( mov(uns32( ' ' ), eax ); eax <= $FF; inc( eax )) do

        mov( toUC[ eax ], bl );
        mov( Tolc[ eax ], bh );
        stdout.put
        (
            "toupper(' ",
            (type char al),
            " ')= ' ",
            (type char bl),
            " 'tolower(' ",
            (type char al),
            " ')= ' ",
            (type char bh),
            "' ",
            nl
        );

    endfor;

end demoCase;

```

这个例子中要注意的一个重要地方是，在 `emitCharRange` 宏调用之后并没有以分号结尾。宏调用不需要以分号结束。通常情况下，在宏调用后面加上一个分号，程序也能合法地进行，这是因为 HLA 一般对插入代码中的多余分号的要求非常宽松。但是在这里，多余的分号是非法的，因

为它们会出现在 TOlc 和 toUC 表中相邻的数据项之间。要记住，宏调用是不需要分号的，特别是将宏调用当作编译时过程来使用的时候。

9.9.2 循环展开

在讨论低级控制结构时，本书曾指出，在某些汇编语言程序中，可以将循环展开以提高某些汇编语言程序的性能。打开或展开循环的问题在于，可能需要大量额外的键盘输入工作，特别是当需要展开很多次迭代的时候。所幸的是，HLA 的编译时语言机制，特别是 `#while` 和 `#for` 循环，可以用来解决这一问题。只需要进行少量的键盘输入工作，再复制一次循环体，就可以将一个循环展开任意多次。

如果只是将同一段代码序列重复若干次，那么将这段代码展开并不复杂。此时需要使用一个 HLA 的 `#for..#endfor` 循环将这个代码序列包括进去，然后将 `val` 对象计数到指定的次数。例如，如果要将 Hello World 打印 10 次，就可以像下面这样进行编码：

```
#for( count := 1 to 10 )
    stdout.put( "Hello World", nl );
#endfor
```

虽然上面的代码看起来和写到程序里的 `for` 循环很相似，但是要记住两者基本的区别：前述代码只是由程序中 10 个 `stdout.put` 的直接调用组成。而如果使用一个 HLA 的 `for` 循环来进行编码的话，就只需要调用一次 `stdout.put`，还需要其他的一些逻辑来控制循环，并将这个调用执行 10 次。

如果循环中的任何指令引用了循环控制变量或者其他某个随着循环的每次迭代而变化的值，循环展开就变得稍微有些复杂了。例如，将一个整数数组的所有元素清零的循环：

```
mov( 0, eax );
for( mov( 0, ebx ); ebx < 20; inc( ebx ) ) do
    mov( eax, array[ ebx*4 ] );
endfor;
```

在这个代码段中，循环使用循环控制变量的值(在 `EBX` 中)作为 `array` 的索引。简单地将 `mov(eax,array[ebx*4]);` 复制 20 次并不是将这个循环展开的正确方法。在这个示例中，必须使用一个适当的在 0~76 范围之内(对应的循环索引乘以 4)的常量作为索引来替换 `ebx*4`。正确地展开这个循环将产生下面的代码序列：

```
mov( eax,array[ 0*4 ] );
mov( eax,array[ 1*4 ] );
mov( eax,array[ 2*4 ] );
mov( eax,array[ 3*4 ] );
mov( eax,array[ 4*4 ] );
mov( eax,array[ 5*4 ] );
mov( eax,array[ 6*4 ] );
mov( eax,array[ 7*4 ] );
mov( eax,array[ 8*4 ] );
mov( eax,array[ 9*4 ] );
```

```

mov( eax,array[ 10*4 ] );
mov( eax,array[ 11*4 ] );
mov( eax,array[ 12*4 ] );
mov( eax,array[ 13*4 ] );
mov( eax,array[ 14*4 ] );
mov( eax,array[ 15*4 ] );
mov( eax,array[ 16*4 ] );
mov( eax,array[ 17*4 ] );
mov( eax,array[ 18*4 ] );
mov( eax,array[ 19*4 ] );

```

可以使用下面的编译时代码序列来更加有效地完成这一工作：

```

#for( iteration := 0 to 19 )
    mov( eax, array[ iteration*4 ] );
#endfor

```

如果循环中的语句使用了循环控制变量的值，那么只有在编译时知道这些值，才可能展开这种循环。在用户的输入(或者其他的运行时信息)控制迭代次数的时候，是不能展开循环的。

9.10 在不同的源文件中使用宏

与过程不同，宏没有位于存储器中某地址上固定的代码块。因此，不能创建“外部”宏，并把它们连接到程序中的其他模块上。但是，在不同的源文件之间共享宏是很简单的：只要将想要重用的宏放在一个头文件中，并使用`#include` 伪指令把该文件包含到其他文件中即可。使用这种技巧，就可以使一个宏在任何源文件中都可用。

9.11 更多信息

虽然本章花费了大量时间来描述 HLA 宏支持的不同特性，以及编译时语言的特性，但事实上这一章只描述了 HLA 极少一部分知识。本章宣称 HLA 的宏机制与其他汇编器相比，功能是最强大的；但是，该章对 HLA 的讨论并没有太深入。如果您曾经使用过一种具有良好的宏机制的语言，通常就会对此不以为然。但是实际上，那些更复杂的内容已经超出了这一章的讨论范围，所以这里没有讨论到。如果您有兴趣学习更多关于 HLA 宏机制的知识，就请参考 HLA 参考手册以及本书的电子版本(<http://webster.cs.ucr.edu/>或 <http://www.artofasm.com/>)。您会发现使用 HLA 的宏机制来创建自己的高级语言的确是有可能的。但是，这一章并不假设读者(现在已经)具备进行这种编程的预备知识，所以这部分的内容也可以从提供本书电子版本的网站上下载。

第10章

位 操 作



在存储器中对位进行操作是汇编语言最显著的特点。实际上，人们将C语言称为一种中级语言而不是高级语言，其原因之一就在于C提供了一组种类繁多的位操作运算符。然而，即使有了这样一组位操作运算符，C编程语言还是不能像汇编语言那样，能够提供完整的位操作运算。

本章将讨论如何使用80x86汇编语言操作那些位于存储器和寄存器中的位串。首先复习一下迄今为止我们所涉及的位操作指令，同时也将引入一些新的指令。然后将复习关于存储器中位串的压缩和解压缩的信息，因为这是很多位操作运算的基础。最后，还将讨论几种以位为中心的算法以及它们的汇编语言实现。

10.1 位数据

在描述如何进行位操作之前，首先要给出位数据(bit data)的定义。大多数读者可能认为位操作程序就是在存储器中操纵各个位。虽然做这种事情的程序肯定就是位操作程序，但我们不能把定义只局限在这些程序上。对我们来说，“位操作”指的是对于那些包含位串的数据类型进行操作，这些位串里的位是不相邻的，甚至其长度可以不是8的整数倍。一般来说，这些位对象不会代表整数值，但是我们不会施加这样的限制。

位串(bit string)就是由一个或多个相邻的位构成的序列。注意，位串并不要求在某一特殊点上开始或结束。例如，位串可以从存储器中某字节的第7位开始，一直延续到存储器中下一个字节的第6位。同样，位串也可以从EAX的第30位开始，占用EAX的高两位，然后从EBX的第0位延续到第17位。在存储器中，各个位在物理上必须是连续的(也就是说，位的序号总是递增的，除非跨越了字节边界，而在字节边界上存储器地址将增加1个字节)。在寄存器中，如果位串跨越

了某个寄存器边界,应用程序就会定义连续寄存器,但位串总是从第2个寄存器的第0位上开始延续。

位组(bit set)是一个由位组成的集合,这些位在大型数据结构内部并不需要是相邻的(虽然它们有可能是相邻的)。例如,某个双字中,第0~3、7、12、24和31位就形成了一个位的集合。通常情况下,我们会将位组限制在一个适当的容器对象(container object,一种用来封装位组的数据结构)中,但是位组的定义并不对它的大小作特别限制。一般情况下,我们能够处理那些作为对象一部分的位组,而这个对象大小无非是32位或者64位,而实际上这一限制完全是人为规定的。注意,位串只是位组的一种特殊情况。

同值位(bit run)是一些具有相同值的位构成的序列。全0位(run of zeros)是一个包含全0的位串,而全1位(run of ones)是一个包含全1的位串。在一个位串中,首置1位(first set bit)是位串中第一个包含1的位,也就是,之后可能是全0位的第一个1位。首清零位(first clear bit)也存在相似的定义。末置1位(last set bit)是位串中最后一个包含1的位;位串剩余的部分形成一个不间断的全0位。末清零位(last clear bit)存在相似的定义。

位偏移(bit offset)是从边界位置(通常是字节边界)算起到指定位的位数。正如第2章中所指出的,我们在边界位置从0开始对位编号。

掩码(mask)是一个由位构成的序列,这些位用于操作其他数值中的某些位。例如,位串%0000_1111_0000,如果使用它来做and操作,就会屏蔽(清零)除了第4~7位以外的所有位。同样,如果使用相同的值来做or操作,就会强行将目的操作数中第4~7位变成1。术语“掩码”就来自于这些位串被用于and操作时的用途;此时,1和0位就像是作画时使用的遮蔽胶带;它们让某些位保持不变,而屏蔽(清零)其他的位。

有了这些定义,我们就可以进行位操作了。

10.2 位操作指令

位操作通常由6种行为组成:位置1、位清零、位反转、位测试与位比较、从位串中提取位,以及向位串中插入位。到现在为止,我们对大多数指令应该比较熟悉了;但无论如何,还应该将这些旧指令复习一下,并给出一些我们将要考虑的位操作指令。

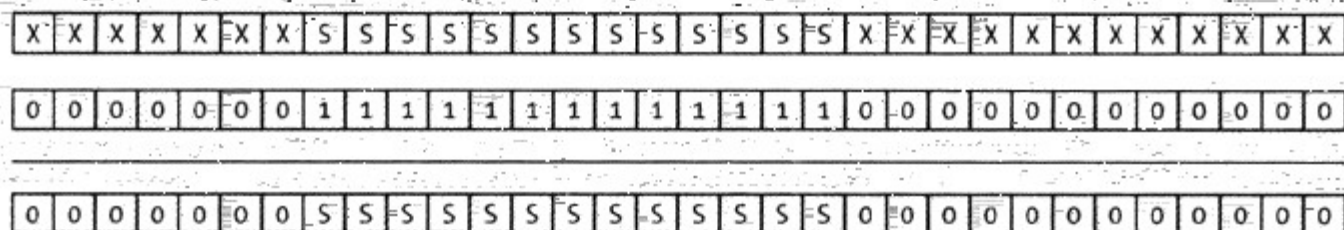
最基本的位操作指令有and、or、xor、not、test,以及移位和循环移位指令。实际上,在早期的80x86处理器上,这些就是全部可用的位操作指令。下面将复习这些指令,并集中讨论如何使用这些指令来操作存储器或寄存器中的位。

and指令提供了一种将不想要的位从位序列中去掉的功能,也就是将那些不想要的位变成0。这个指令对于将一个与其他不相关数据(或至少不是位串或位组的一部分的数据)合并了的位串或位组分离出来的操作来说,是很有用的。例如,假设某个位串占用EAX的第12~24位;我们可以使用下面的指令将EAX中的其他位设置为0,从而将这个位串分离出来:

```
and( %1, EAX, 1111_1111_0000_0000_0000, eax );
```

大多数程序使用and指令将那些不想成为位串组成部分的位清零。理论上讲,也可以使用or指令将所有不想要的位屏蔽为1而不是0,但是如果不需要的位中包含0,那么后者的比较和

操作通常要简单一些(如图 10-1 所示)。



使用位掩码将 EAX 中的第 12~24 位分离出来

上: EAX 中的初始值 中: 位掩码 下: EAX 的终值

图10-1 使用 and 指令分离出一个位串

只要清除了位组中那些不想要的位,通常就可以正确地操作位组了。例如,要检查位于 EAX 寄存器中第 12~24 位上的一个位串是否包含 \$12F3, 可以使用下面的代码:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, %1_0010_1111_0011_0000_0000_0000 );
```

这里还有另一种解决方法,使用常量表达式,更容易理解一些:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, $12F3 << 12 );           // "<<12" shifts $12F3 to the left 12 bits.
```

但是,大多数情况下,在进行所需的任何操作之前,都想(或者需要)将位串与 EAX 的第 0 位对齐。当然,在屏蔽以后,可以使用 shr 指令将其适当地对齐:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
shr( 12, eax );
cmp( eax, $12F3 );
<< Other operations that require the bit string at bit #0 >>
```

现在位串已经与第 0 位对齐,而与该数值结合使用的常量和数值就更易于处理。

还可以使用 or 指令屏蔽不想要的位。但是,or 指令不能用来清零位;它只能将位设置为 1。在某些情况下,可能需要将位组两侧所有的位置 1;但如果只想将两侧的位清零,而不是置 1 的话,那么大多数软件还是比较容易编写的。

or 指令对于将一个位组插入其他位串中的操作来说特别有用。为了做到这一点,需要执行以下步骤:

- 将源操作数中位组两侧的位清零。
- 将目的操作数中将要插入位组的所有位清零。
- 将位组和目的操作数进行 or 操作。

例如,假设在 EAX 的第 0~12 位中有一个值,想将它插入 EBX 的第 12~24 位上,而不影响 EBX 中的其他位。可以首先在 EAX 中将第 13 位及以上的位清零;然后,将 EBX 中的第 12~24 位清零。接下来,移动 EAX 中的位,使得将被插入的位串占据 EAX 的第 12~24 位。最后将 EAX 和 EBX 中的数值进行 or 操作(如图 10-2 所示):

```
and( $1FFF, eax );           // Strip all but bits 0..12 from eax.
and( $00FF_0FFF, ebx );      // Clear bits 12..24 in ebx.
```

```
shl( 12, eax );           // Move bits 0..12 to 12..24 in eax.
or( eax, ebx );           // Merge the bits into ebx.
```

EBX:

X	X	X	X	X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX:

U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第1步: 从 EAX 清除不需要的位(U 位)

EBX:

X	X	X	X	X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第2步: 屏蔽 EBX 中的目的位域

EBX:

X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第3步: 将 EAX 中的 12 个位置左移, 使其与目的位域对齐

EBX:

X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX:

0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第4步: 将 EAX 的值与 EBX 中的值合并

EBX:

X	X	X	X	X	X	X	A	A	A	A	A	A	A	A	A	A	A	A	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX:

0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EBX 中的最终结果

图 10-2 将 EAX 中的第 0~12 位插入 EBX 的第 12~24 位

在这个示例中, 所需要的位(AAAAAAAAAAAAAA)形成了一个位串。然而, 在对不相邻的一组位进行操作时, 这个算法仍然有效。只须创建一个适当的位掩码, 在合适的位置与那些包含 1 的位进行 and 操作。

在使用位掩码的时候, 像前面几个示例中那样使用字面数值常量, 所使用的编程风格差得令人难以置信。通常都应该在 HLA 的 const(或者 val)段为位掩码创建一个符号常量, 然后结合使用一些常量表达式, 就可以产生更便于阅读和维护的代码。当前的示例代码被编写为:

```
const
    StartPosn := 12;
    BitMask:dword := $1FFE<< StartPosn; // Mask occupies bits 12..24.

    shl( StartPosn, eax ); // Move into position.
    and( BitMask, eax );   // Strip all but bits 12..24 from eax.
    and( !BitMask, ebx );  // Clear bits 12..24 in ebx.
    or( eax, ebx );        // Merge the bits into ebx.
```


注意编译时 `not` 操作符(!)的使用,它反转了位掩码,以便将 `EBX` 中的位清零,而这些位就是代码将 `EAX` 中的位插入进来的位置。这就节省了一部分工作,不需要在程序中创建另一个常量,只要 `BitMask` 常量被修改,这个常量就要跟着变化。在程序中,要保留两个数值彼此相关的符号并不是一件好事。

当然,除了将一个位组与另一个位组合并以外,`or` 指令对于将位串中的位强行变成 1 的操作来说,也很有用。将源操作数中的几个位设置成 1,再使用 `or` 指令,就可以将目的操作数中的对应位强行变成 1。

`xor` 指令也提供了将属于某一位组的指定位进行反转的功能。虽然对位的反转需求并不像对它们置 1 和清零那样普遍,但是在位操作程序中,`xor` 指令仍然显示出相当重要的作用。当然,如果要将目的操作数中的所有位都进行反转,使用 `not` 指令会比使用 `xor` 指令更合适一些;但是,要反转指定的位而不影响其他位的话,应该使用 `xor` 指令。

`xor` 操作允许使用任何能够想到的方法来操作已知的数据。例如,如果已知某个域包含 `%1010`,那么就可以将其与 `%1010` 进行 `xor` 操作,将这个域强行变成全 0。类似地,还可以将其与 `%0101` 进行 `xor` 操作,将这个域强行变成 `%1111`。虽然这看起来好像没有用处,因为可以简单地使用 `and/or` 指令来将这个 4 位的位串强行变成全 0 或者全 1,但 `xor` 指令具有两个优点:①它的功能不仅限于将这个域变成全 0 或者全 1;实际上可以使用 `xor` 将这些位设置 16 种合法的组合。②如果需要同时操作目的操作数的其他位,`and/or` 指令就不能满足需要了。例如,假设已知一个域包含 `%1010`,想把它强行变成 0,而另一个域包含 `%1000`,但想把这个域增加 1(也就是将这个域设置成 `%1001`)。使用单独一条 `and` 或者 `or` 指令无法同时实现这两种操作,但可以使用一条 `xor` 指令来完成;只要将第一个域与 `%1010` 进行 `xor` 操作,而将第二个域与 `%0001` 进行 `xor` 操作即可。但是要记住,只有已知目的操作数中位组的当前值,才能使用这个技巧。当然,在调整包含已知值的位域的值时,也可以同时对其他域中的位进行反转。

除了置 1、清零,以及反转目的操作数中的某些位以外,`and`、`or` 和 `xor` 指令还会影响标志寄存器中的条件码。这些指令对标志的影响如下:

- 这些指令总是会将进位和溢出标志清零。
- 当结果的高位位上是 1 时,这些指令就会将符号标志置 1,否则就会将符号标志清零。也就是说,这些指令会将结果的高位位复制到符号标志中。
- 这些指令会根据结果是否为 0 来置 1/清零零标志。
- 如果在目的操作数的低字节位上有偶数个位被置 1,那么这些指令就会将奇偶标志置 1。如果在目的操作数的低字节位上有奇数个位为 1,这些指令就会将奇偶标志清零。

注意,这些指令总是会将进位和溢出标志清零。这就意味着不能指望系统在这些指令的执行期间保持这两个标志的状态。很多汇编语言程序都有一种非常普遍的错误,那就是它们都会假设这些指令是不影响进位标志的。很多人会执行一条置 1/清零进位标志的指令,再执行 `and/or/xor` 指令,然后去测试前一条指令执行之后进位标志的状态,这样做是不正确的。

另外,这些指令会将其结果的高位位复制到符号标志中。这就意味着,只要测试符号标志(使用 `sets/setns`、`js/jns` 指令,或者在一个布尔表达式中使用 `@s/@ns` 标志),就很容易测试出结果的高位位状态。由于这个原因,很多汇编语言程序员都经常将重要的布尔变量放在某个操作数的高位位,这样他们就可以在某次逻辑操作之后,使用符号标志测试出那一位的状态。

在本书中,我们还没有讨论太多关于奇偶标志的问题。我们并不打算对这个标志及其用途展

开大规模的讨论,因为这个标志的主要任务已经被硬件接管了¹。但是,因为这一章是关于位操作的,而奇偶计算又是一种位操作运算,所以此时进行一些关于奇偶标志的简短讨论也很恰当。

奇偶校验是一种非常简单的错误检查方案,它最初被用在电报以及其他的串行通信协议中。奇偶校验的工作原理是,对一个字符中被置1的位进行计数,并在传输过程中另外添加一位,来表示字符中被置1的位有奇数个还是偶数个。传输的接收端也会对位进行计数,并对那个附加的“奇偶”位进行校验,以确认传输是否成功。奇偶标志的目的就是为了帮助计算这个额外位的值,在这里我们并不打算探讨关于这种错误检查方案中涉及的信息理论方面的问题。

对于80x86的and、or和xor指令来说,当操作数的低位字节包含偶数个被置1的位时,它们就会将奇偶标志位置1。有一个重要的事实在这里应该重复说明:奇偶标志只反映目的操作数的低位字节中被置1的位数;它并不包括字、双字或者其他类型操作数的高位字节。这个指令集合只使用低位字节来计算奇偶性,是因为使用奇偶校验的通信程序在通常情况下是面向字符的传输系统(如果一次传输8位以上的信息的话,就会有更好的错误检查方案)。

零标志设置是and/or/xor所产生的一个重要结果。实际上,在and指令之后程序对这个标志的引用是如此频繁,以至于Intel专门加入了一条单独的指令test,它的主要任务就是将两个结果进行逻辑and操作并设置标志,而不影响每一个指令操作数。

在一条and或者test指令执行之后,零标志有3种主要的用途:①检查操作数中的某个位是否被置1;②检查位组的几个位中是否至少有一个位是1;③检查某个操作数是否为0。当位组只包含一位时,①实际上是②的一种特殊情况。下面我们将探讨这3种用途。

and指令通常用于测试在给定的操作数中某一个特定的位是否被置1,这也是在80x86指令集中加入test指令的初衷。进行这样的测试时,要将只包含一个被置1位的常数值与想要测试的操作数进行and/test操作。这会将第二个操作数中其他的所有位都清零,如果被测试的操作数在被测试位的位置上包含一个0,操作过后就会在这个位置上留下一个0。而如果该位置包含一个1,经过与1进行and操作之后就会留下一个1。因为结果的其他位都是0,所以如果被测试位是0的话,整个结果就是0;如果那个位是1,整个结果就是非0的值。80x86将在零标志中反映出这种状态(Z=1表示有一个0位,Z=0表示有一个1位)。下面的指令序列演示了如何测试EAX中的第4位是否被置1:

```
test( %1_000, eax );      // Check bit #4 to see if it is 0/1.
if( !nz ) then

    << Do this if the bit is set. >>

else

    << Do this if the bit is clear. >>

endif;
```

也可以使用and/test指令来测试几位中是否有一位被置1,只要提供一个在所有想要测试的位置上都包含1(其他位上都包含0)的常量即可。将这样一个值与未知量进行and操作,如果被测试的操作数中有一位或者多位包含1,就会产生一个非0值。下面的示例将测试EAX中的值在其第

1 使用奇偶校验来查错的串行通信芯片和其他通信硬件通常情况下是在硬件中计算奇偶性的;您不必使用软件来达到这一目的。

1、2、4 和 7 位上是否包含 1:

```
test( %1001_0010, eax );
if( @nz )then    // At least one of the bits is set.

    << Do whatever needs to be done if one of the bits is set. >>

endif;
```

注意，不能使用一条单独的 **and** 或者 **test** 指令来测试位组中所有相应的位是否等于 1。要实现这一点，必须首先将不在组中的其他位屏蔽掉，然后将结果与掩码本身进行比较。如果结果与掩码相等，那么就说明位组中所有的位都包含 1。对于这种操作必须使用 **and** 指令，因为 **test** 指令不会屏蔽其他任何位。下面的示例将测试位组(**bitMask**)中的所有位是否都等于 1:

```
and( bitMask, eax );
cmp( eax, bitMask );
if( @e )then

    // All the bit positions in eax corresponding to the set
    // bits in bitMask are equal to 1 if we get here.

    << Do whatever needs to be done if the bits match. >>

endif;
```

当然，这里我们只要使用 **cmp** 指令，就不必真正检查该位组中的所有位是否为 1。我们可以将适当的值指定为 **cmp** 指令的操作数，来检查这些值的任何组合。

注意，只有当在 **EAX**(或者其他目的操作数)中与常量操作数中出现 1 的位置所对应的位都是 0 时，**test/and** 指令在上面的代码序列中才会设置零标志。这就给我们提示了另一种方法来检查位组中所有置 1 的位：在使用 **and** 或者 **test** 指令之前，先将 **EAX** 中的值进行反转。然后如果零标志被置 1，就可以知道(原来的)位组是全 1。例如：

```
not( eax );
test( bitMask, eax );
if( @z )then
    // At this point, eax contained all ones in the bit positions
    // occupied by ones in the bitMask constant.

    << Do whatever needs to be done at this point. >>

endif;
```

前面都假设 **bitMask**(也就是源操作数)是一个常量，这只是出于举例方便的目的。事实上，如果喜欢的话，这里也可以使用一个变量或者寄存器。只要在上面的示例中执行 **test**、**and** 或者 **cmp** 指令之前，使用合适的位掩码来加载变量或寄存器即可。

另一组可以用来操作位的指令是位测试指令。这些指令包括 **bt**(位测试)、**bts**(位测试并置 1)、**btc**(位测试并取补)，以及 **btr**(位测试并重置)。我们曾经使用过这些指令来操作 **HLA** 字符集变量中的位；同样可以使用它们来操作一般的位。**btr** 指令允许下面的语法形式：


```

btb( BitNumber, BitsToTest );
btb( reg16, reg16 );
btb( reg32, reg32 );
btb( constant, reg16 );
btb( constant, reg32 );
btb( reg16, mem16 );
btb( reg32, mem32 );
btb( constant, mem16 );
btb( constant, mem32 );

```

btb 指令的第一个操作数是一个位编号,它指定了第二个操作数中所要检查的位。如果第二个操作数是一个寄存器,那么第一个操作数中所包含的值就必须在 0 到寄存器大小(以位为单位)减 1 之间;因为 80x86 的最大寄存器是 32 位的,所以这个值的最大值是 31(在 32 位寄存器的情况下)。如果第三个操作数是一个存储单元,那么位的个数就不会局限于 0~31 之间。如果第一个操作数是一个常量,它可以是在 0~255 之间的任何一个 8 位值。如果第一个操作数是一个寄存器,就没有限制了。

bt 指令将指定的位从第二个操作数复制到进位标志中。例如, bt(8, eax); 指令将 EAX 寄存器的第 8 位复制到进位标志中。在这条指令执行之后,可以测试进位标志,来确定 EAX 中的第 8 位是被置 1 还是被清零。

bts、btc 和 btr 指令在测试某一位时,会对那一位进行操作。这些指令相当慢(这取决于使用的处理器),所以如果性能是主要关心的问题,并且使用的是较老的 CPU,就要避免使用它们。如果性能(相对于便利性来说)很重要,通常就应该尝试两种不同的算法——其中一个使用这些指令,另一个使用 and/test 指令——测量它们性能的差异;然后在这两种不同的方案中选择一种最好的。

移位和循环移位指令是另一组可以用来操作并测试位的指令。当然,所有这些指令都会将高位位(左移/循环左移)或者低位位(右移/循环右移)移动到进位标志中。因此,在执行一条指令之后,就可以测试进位标志,来确定操作数的高位位或低位位的初始设置。当然,移位和循环移位指令对于对齐位串和压缩与解压缩数据的操作来说非常有价值。第 2 章中有这些操作的示例,并且本章前面一些示例也使用了移位指令来达到这些目的。

10.3 作为位累加器的进位标志

btb、移位和循环移位指令都会根据操作和被选择的位来置 1 或者清零进位标志。因为这些指令将它们的“位结果”放在进位标志中,所以将进位标志作为一个 1 位的寄存器或者位操作的累加器来考虑,通常是很方便的。在本节中,我们将探讨对于进位标志中的位结果可能进行的一些操作。

对于操作进位标志中的位结果来说,将进位标志作为某种输入值的指令非常有用。例如:

- adc、sbb
- rcl、rcr
- cmc(我们还会引入 clc 和 stc,即使它们并不使用进位作为输入值)
- jc、jnc

- setc、setnc

adc 和 sbb 指令将它们的操作数与进位标志一起相加或者相减。所以如果已经计算出了位结果并产生进位的话,就可以使用这些指令将结果计算到一次加法或者减法中去。

为了合并进位标志中的位结果,最常用的就是通过进位的循环移位指令(rcl 和 rcr)。当然,这些指令会将进位标志移到目的操作数的低位位或高位位中。这些指令对于将一组位结果压缩成一个字节、字或者双字的操作来说,是很有用的。

cmc(进位取补)指令可以轻松地将某些位操作的结果进行反转。在进行某些涉及进位标志的位串操作之前,也可以使用 clc 和 stc 指令初始化进位标志。

当然,在进行某些计算之后,如果进位标志中留下某个位结果的话,使用测试进位标志的指令很常见。这时,jc、jnc、setc,以及 setnc 指令相当有用。也可以在布尔表达式中使用 HLA 的 @c 和 @nc 操作数来测试进位标志中的结果。

如果有一系列的位计算,并且想要测试这些计算是否产生了一组特定的 1 位结果,那么最简单的方法就是将寄存器或者存储单元清零,然后使用 rcl 或者 rcr 指令将每个结果都移入那个位置。一旦所有的位操作都完成,就可以将该寄存器或者存储单元与一个常量值进行比较,看是否得到一个特定的结果序列。如果想要测试一个涉及逻辑与和逻辑或操作的结果序列(也就是涉及 and 和 or 的结果串),那么可以使用 setc 和 setnc 指令将寄存器设置为 0 或者 1,然后使用 and/or 指令合并该结果。

10.4 位串的压缩与解压缩

将一个位串插入一个操作数中,或者从一个操作数中提取一个位串,都是很普遍的位操作。第 2 章曾提供了对于这种数据进行压缩或者解压缩操作的示例;现在就来正式介绍如何完成这些操作。

为了进行讨论,我们假设此时正在处理的是位串——也就是一个由连续的位构成的序列。在 10.11 节,我们将讨论如何对位组进行提取和插入操作。另一处需要简化的条件是,假设位串的长度与一个字节、字或者双字操作数完全匹配。更大一些的跨越对象边界的位串则需要另外处理;关于跨越双字边界的位串的讨论将在本节稍后进行。

在对一个位串进行压缩和解压缩的时候,有两个属性是我们必须考虑的:起始位的位置和长度。起始位的位置是指在较大的操作数中字串的低位位的位编号,长度当然就是操作数的位数。要将数据插入(压缩)到一个目的操作数中,就要从一个右对齐的(也就是从第 0 个位置开始)具有适当长度的位串开始,然后零扩展到 8 位、16 位或者 32 位。任务就是将这个数据插入另外一个 8 位、16 位或者 32 位操作数的适当起始位置上。这里并不保证目的位置包含有任何特殊的值。

前两个步骤(不分前后顺序)是将目的操作数中的对应位清零,并对位串(的副本)进行移位,使其低位位开始于适当的位置。完成这两步以后,第 3 步就是将移位的结果与目的操作数进行 or 操作。这就将位串插入到了目的操作数中(如图 10-3 所示)。

目的操作数

X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

源操作数

0	0	0	0	0	0	0	0	0	0	0	Y	Y	Y	Y	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第 1 步: 将 YYYY 插入到目的操作数中由 DDDD 占据的位
置上, 首先将源操作数左移 5 位

目的操作数

X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

源操作数

0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第 2 步: 使用 AND 指令清除目的位

目的操作数

X	X	X	X	X	X	X	0	0	0	0	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

源操作数

0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

第 3 步: 将两个数值进行 OR 操作

目的操作数

X	X	X	X	X	X	X	Y	Y	Y	Y	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

源操作数

0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

最终结果在目的操作数中

图 10-3 将位串插入到一个目的操作数中

将一个已知长度的位串插入目的操作数中只需要 3 条指令。下面的 3 条指令演示了如何处理图 10-3 中的插入操作。这些指令假设源操作数在 BX 中, 而目的操作数在 AX 中:

```
shl( 5, bx );
and( %1111110000111111, ax );
or( bx, ax );
```

如果长度和起始位置在编写程序的时候还不知道(也就是说, 必须在运行时计算它们), 那么位串插入就有一些困难。但是, 利用查找表的话, 就仍然很容易实现。我们假设有两个 8 位值: 一个是要插入域的起始位置, 一个是非零的 8 位长度值。假设源操作数在 EBX 中, 而目的操作数在 EAX 中。将一个操作数插入到另一个操作数中的代码会采取下面的形式:

```
readonly
// The index into the following table specifies the length
// of the bit string at each position:

MaskByLen: dword[ 33 ] :=
[
    0, $1, $3, $7, $f, $1f, $3f, $7f,
    $ff, $1ff, $3fff, $7fff, $ffff, $1ffff, $3ffff, $7ffff, $ffffff,
    $1_ffff, $3_ffff, $7_ffff, $f_ffff,
    $1f_ffff, $3f_ffff, $7f_ffff, $ff_ffff,
    $1ff_ffff, $3ff_ffff, $7ff_ffff, $fff_ffff,
    $1fff_ffff, $3fff_ffff, $7fff_ffff, $ffff_ffff
];
```

```

movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
shl( cl, edx );
not( edx );
shl( cl, ebx );
and( edx, eax );
or( ebx, eax );

```

MaskByLen 表的每一项都包含由表索引所指定的 1 位的编号。可以使用 *Length* 值作为这个表的索引来提取一个值, 这个值中的 1 位和 *Length* 值中的一样多。上面的代码取得一个适当的掩码, 然后将其左移, 使得这个全 1 位的低位位与我们要插入数据的域的起始位置相匹配。接下来, 这段代码将掩码进行反转, 然后使用反转后的值将目的操作数中适当的位清零。

从一个较大的操作数中提取一个位串就像在较大的操作数中插入位串一样简单。此时只须将不想要的位屏蔽掉, 然后将结果左移, 直到这个位串的低位位被移到目的操作数的第 0 位为止。例如, 要在 EBX 中从第 5 个位置上开始提取一个 4 位的域, 并将结果放在 EAX 中的话, 就可以使用下面的代码:

```

mov( ebx, eax );           // Copy data to destination.
and( %1_1f10_0000, ebx ); // Strip unwanted bits.
shr( 5, eax );             // Right justify to bit position 0.

```

如果在编写程序的时候不知道位串的长度和起始位置, 仍然可以提取想要的位串。代码和插入操作的很相似(但稍微简单一些)。假设已经有了 *Length* 和 *StartingPosition* 的值, 这两个值是在插入一个位串时使用过的, 然后就可以使用下面的代码提取相应的位串(假设源操作数=EBX, 目的操作数=EAX):

```

movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
mov( ebx, eax );
shr( -cl, eax );
and( edx, eax );

```

到现在为止, 所有示例都假设位串会完整地出现在一个双字(或更小)对象中。这通常是位串的长度小于或者等于 32 位的情况。但是, 如果位串的长度加上它在对象中的起始位置(mod 8)以后大于 32, 那么这个位串就会在对象中跨越一个双字的边界。要提取这样的位串需要三个操作: 提取位串的起始位置(直到第一个双字边界)、复制所有的双字(假设位串非常长, 占用了好几个双字)以及复制位串末尾最后一个双字中剩余的位。这个操作的实现将留作读者的练习。

10.5 接合位组与分布位串

如果插入的位组(或者提取后产生的位组)的“形状”与主对象中的位组相同,那么插入和提取位组就与插入和提取位串稍有不同。位组的形状是指位组中位的分布,而不管位组的起始位置。所以一个包含第 0、4、5、6 和 7 位的位组的形状与一个包含第 12、16、17、18 和 19 位的位组的形状相同,因为二者位的分布是相同的。插入和提取这种位组的代码与前一节中的代码几乎完全相同;唯一的区别在于所用掩码的值。例如,要将 EAX 中从第 0 位开始的这个位组插入到 EBX 中从第 12 位开始的对应位组中,可以使用下面的代码:

```
and( !%1111_0001_0000_0000_0000, ebx ); // Mask out destination bits.
shl( 12, eax ); // Move source bits into position.
or( eax, ebx ); // Merge the bit set into ebx.
```

但是,如果想把 EAX 中第 0~4 位(共有 5 位)合并到 EBX 的第 12、16、17、18 和 19 位上,那么就要在将这些值与 EBX 进行逻辑 or 操作之前,先将它们分布在 EAX 中。只要该位组只拥有两组全 1 位的话,处理过程就简单多了。下面的代码使用一种比较灵活的方式实现了这个操作:

```
and( !%1111_0001_0000_0000_0000, ebx );
shl( 3, eax ); // Spread out the bits:1-4 goes to 4-7 and 0 to 3.
btr( 3, eax ); // Bit 3->carry and then clear bit 3.
rcl( 12, eax ); // Shift in carry and put bits into final position.
or( eax, ebx ); // Merge the bit set into ebx.
```

这个使用 btr(位测试并重置)指令的技巧之所以能够起作用,是因为我们在初始的源操作数中只有一个不适当的 1 位。如果使这些位都处于彼此相对错误的位置上,这个方案就不起作用。稍后我们会看到一种更通用的解决方法。

提取这个位组并将其中的位收集(接合)到一个位串中并不容易。但是,仍然有一些灵活的技巧可以使用。考虑下面的代码,它将从 EBX 中提取位组并将结果放在 EAX 的第 0~4 位中:

```
mov( ebx, eax );
and( %1111_0001_0000_0000_0000, eax ); // Strip unwanted bits.
shr( 5, eax ); // Put bit 12 into bit 7,etc.
shr( 3, ah ); // Move bits 11..14 to 8..11.
shr( 7, eax ); // Move down to bit 0.
```

这段代码将(初始的)第 12 位移到第 7 位上,这是 AL 的高位位。同时,它将第 16~19 位下移到第 11~14 位(AH 的第 3~6 位)。然后这段代码将 AH 的第 3~6 位下移到第 0 位。这样就定位了位组的高位位,使它们与 AL 中最左边的一位相邻。最后,这段代码将所有的位下移到第 0 位。同样,这也不是一种通用的解决方案,但如果仔细考虑,它的确是解决这个问题的一种好方法。

接合和分布算法的问题在于它们不是通用的,只能应用于特殊的位组。一般来讲,特殊的解决方法会提供一种最高效的解决方法。有一种通用的解决方法(指定一个掩码,然后代码就会相应地对位进行分布或者接合),但会更困难一点。下面的代码演示了如何在一个位串中根据位掩码的值对位进行分布:

```

// eax- Originally contains some value into which we insert bits from ebx.
// ebx- L.O.bits contain the values to insert into eax.
// edx- Bitmap with ones indicating the bit positions in eax to insert.
// cl- Scratchpad register.

        mov( 32, cl );    // Count number of bits we rotate.
        jmp DistLoop;

CopyToEAX: rcr( 1, ebx );    // Don't use SHR here, must preserve Z-flag.
          rcr( 1, eax );
          jz Done;
DistLoop: dec( cl );
          shr( 1, edx );
          jc CopyToEAX;
          ror( 1, eax );    // Keep current bit in eax.
          jnz DistLoop;

Done:    ror( cl, eax );    // Reposition remaining bits.

```

在上面的代码中,如果我们用%1100_1001来加载EDX的话,那么这段代码就会将第0~3位复制到EAX中的第0、3、6和7位上。请注意短路测试过程,该测试(通过检验EDX中的0)检查我们是否用尽了EDX中的值。注意,循环移位指令不会影响零标志,而移位指令会影响。这样,上面的shr指令在没有更多位可以分布的时候(也就是,当EDX变成0的时候),就会设置零标志。

接合位的一般算法比分布算法稍微高效一些。这里有一段代码,它使用EDX中的位掩码,从EBX中提取位并将结果保存在EAX中:

```

// eax- Destination register.
// ebx- Source register.
// edx- Bitmap with ones representing bits to copy to eax.
// ebx and edx are not preserved.

        sub( eax, -eax );    // Clear destination register.
        jmp ShiftLoop;

ShiftInEAX:
        rcl( 1, ebx );    // Up here we need to copy a bit from
        rcl( 1, eax );    // ebx to eax.

ShiftLoop:
        shl( 1, edx );    // Check mask to see if we need to copy a bit.
        jc ShiftInEAX;    // If carry set, go copy the bit.
        rcl( 1, ebx );    // Current bit is uninteresting, skip it.
        jnz ShiftLoop;    // Repeat as long as there are bits in edx.

```

这个代码序列利用移位和循环移位指令的一个灵活特点:移位指令会影响零标志,而循环移位指令不会。因此,shl(1,edx);指令在EDX变成0(移位之后)时会设置零标志。如果进位标志被置1,这段代码就会多执行一遍循环,将1位移到EAX中。但是下一次代码将EDX的1位向左移时,EDX仍然为0,因此进位标志也被清零。在这次迭代中,代码退出了循环。

另一种接合位的方法是借助表查找。通过每次获取一个字节的的数据(这样表就不会很大),就

可以使用这个字节的值作为一个查找表的索引。这个查找表接合了直到第 0 位的所有位。最后，就可以将每个字节低端的位合并在一起。这样就会产生一个在某些情况下更高效的接合算法，该算法留给读者实现。

10.6 压缩的位串数组

虽然创建元素字节数为整数的数组的效率非常高，但创建那些元素不是 8 位整数倍的数组也是可能的。这样做的缺陷在于计算一个数组元素的“地址”以及操作数组元素时会涉及大量额外的工作。本节我们将讨论一些对数组元素进行压缩和解压缩的示例，这些数组元素的长度是任意的。

在继续讨论之前，也许有必要讨论一下，为什么要操作位对象数组。答案很简单：因为空间的原因。如果一个对象只占用 3 位，那么如果对该数据进行压缩，而不是给每个对象都分配整个字节的话，就可以在同样大的空间里放下原来的 2.67 倍的元素。对于非常大的数组来说，这可能是一种非常实在的节约。当然，节约空间的代价就是影响速度：必须要执行额外的指令来对数据进行压缩或者解压缩，这就减慢了对数据的访问。

对于在较大的位块中定位一个数组元素的位偏移量来说，它的计算与标准的数组访问几乎是等同的：就是

```
Element_Address_in_bits =
    Base_address_in_bits + index * element_size_in_bits
```

一旦计算出元素的位地址，就需要将其转换成一个字节地址(因为我们在访问存储器的时候，不得不使用字节地址)，并提取特定的元素。因为一个数组元素的基址(几乎)总是从一个字节边界开始，所以我们可以使用下面的等式来简化这个任务：

```
Byte_of_1st_bit =
    Base_Address + (index * element_size_in_bits) / 8

Offset_to_1st_bit =
    (index * element_size_in_bits) % 8 (note "%" = MOD)
```

例如，假设我们有一个数组，包含 200 个 3 位的对象，我们将其声明如下：

```
static
    AO3Bobjects: byte[ int32(200*3)/8 + 2 ] ; // "+2"handles
                                                // truncation.
```

上面数组维数中的常量表达式保留了足够多字节的空间来存放 600 位(200 个元素，每个长度为 3 位)。正如注释中指出的那样，表达式的最后加上 2 个额外的字节是为了确保我们不丢失任何奇数位(在该示例中，这种情况不会发生，因为 600 可以被 8 整除，但一般不能指望这种情况；一个额外的字节通常也不碍事)，以及允许访问数组最后一个元素后面的一个字节(在把数据存储到数组中时)。

现在假设想要访问这个数组中第 i 个 3 位的元素。可以使用下面的代码提取这些位：


```

// Extract the ith group of 3 bits in AO3Bobjects
// and leave this value in eax.

sub( ecx, ecx );           // Put i/8 remainder here.
mov( i, eax );            // Get the index into the array.
lea( eax, [eax+eax*2] );   // eax := eax * 3 (3 bits/element).
shrd( 3, eax, ecx );       // eax/8 -> eax and eax mod 8 -> ecx(H.O.bits).

shr( 3, eax );            // Remember, shrd above doesn't modify eax.
rol( 3, ecx );            // Put remainder into L.O.3 bits of ecx.

// Okay, fetch the word containing the 3 bits we want to
// extract. We have to fetch a word because the last bit or two
// could wind up crossing the byte boundary (i.e., bit offset 6
// and 7 in the byte).

mov( (type word AO3Bobjects[eax]), ax );
shr( cl, ax );            // Move bits down to bit 0.
and( %111, eax );        // Remove the other bits.

```

将一个元素插入数组要更加困难一些。除了计算数组元素的基址和位偏移量以外，还要创建一个掩码将要插入新数据的目的操作数中的某些位清零。下面的代码将 EAX 的低 3 位插入 AO3Bobjects 数组中的第 i 个元素中。

```

// Insert the L.O. 3 bits of ax into the ith element
// of AO3Bobjects:

readonly
Masks:
    word[8] :=
    [
        !%0111,           !%0011_1000,
        !%0001_1100_0000, !%1110,
        !%0111_0000,       !%0011_1000_0000,
        !%0001_1100,       !%1110_0000
    ];

mov( i, ebx );           // Get the index into the array.
mov( ebx, ecx );         // Use L.O. 3 bits as index
and( %111, ecx );        // into Masks table.
mov( Masks[ecx*2], dx );  // Get bit mask.

// Convert index into the array into a bit index.
// To do this, multiply the index by 3:

lea( ebx, [ebx+ebx*2] );

// Divide by 8 to get the byte index into ebx
// and the bit index (the remainder) into ecx:

shrd( 3, ebx, ecx );

```

```

    shr( 3, ebx );
    rol( 3, ecx );

    // Grab the bits and clear those we're inserting.
    and( (type word AO3Bobjects[ ebx ]), dx );

    // Put our 3 bits in their proper location.
    shl( cl, ax );

    // Merge bits into destination.
    or( ax, dx );

    // Store back into memory.
    mov( dx, (type word AO3Bobjects[ ebx ]) );

```

请注意查找表的使用，它用来产生清零数组中适当位置所需的掩码。除了根据给定的位偏移量，我们需要清零的位置上有3个0以外，这个数组的每个元素都包含全1(注意!操作符的使用，它用来反转表中的常量)。

10.7 搜索位

有一种非常普遍的位操作是定位某些同值位的结束。这种操作的常见情况是在一个16位或者32位的数值中，定位首(末)置1或首(末)清零位。在这一节中，我们将探讨实现这种方法。

在描述如何搜索给定值的首位或者末位之前，先来讨论一下这里术语“首”和“末”的意思。术语“首置1位”是指在一个数值中从第0位向高位位扫描，所碰到的第一个包含1的位。“首清零位”有着相似的定义。“末置1位”就是在一个数值中从高位位向第0位扫描，所碰到的第一个包含1的位。“末清零位”有着相似的定义。

扫描首位或末位有一种很明显的方法，那就是在循环中使用一条移位指令，然后计算循环的迭代次数，直到移出一个1(或者0)到进位标志为止。迭代的次数指出了位置信息。这里有一些示例代码，用于检查EAX中的首置1位并在ECX中返回位置信息：

```

    mov( -32, ecx );    // Count off the bit positions in ecx.
TstLp:  shr( 1, eax );   // Check to see if current bit
                        // position contains a 1.

    jc Done             // Exit loop if it does.
    inc( ecx );         // Bump up our bit counter by 1.
    jnz TstLp;         // Exit if we execute this loop 32 times.

Done:   add( 32, cl );   // Adjust loop counter so it holds
                        // the bit position.

// At this point, ecx contains the bit position of the first set bit.
// ecx contains 32 if eax originally contained 0 (no set bits).

```

这段代码中唯一使用技巧的地方就是代码运行所使用的循环计数器是从-32增加到0，而不

是从 32 减到 0。这使得在循环终止的时候, 计算位的位置信息变得容易一些。

这种循环的缺点在于, 它的实现代价很高。这个循环会根据 EAX 中的初始值重复 32 次。如果被检查的值在 EAX 的低位位上经常包含很多 0, 这段代码的运行速度就会相当慢。

搜索首(或者末)置 1 位是一种非常普遍的操作, 以至于 Intel 专门在 80386 上增加了两条指令来加速这个过程。这两条指令是 `bsf`(向前位扫描)和 `bsr`(向后位扫描)。它们的语法如下:

```
bsr( source, destReg );
bsf( source, destReg );
```

源操作数和目的操作数的长度必须是相同, 并且必须都是 16 位或者 32 位的对象。目的操作数必须是一个寄存器, 而源操作数可以是一个寄存器, 也可以是存储单元。

`bsf` 指令扫描源操作数中的首置 1 位(从第 0 位开始), 而 `bsr` 指令从高位位到低位位扫描源操作数的末置 1 位。如果指令在源操作数中找到了一个被置 1 的位, 那么它们就会将零标志清零, 然后将位的位置信息放到目的寄存器中。如果源寄存器包含 0(也就是其中没有被置 1 的位), 那么这些指令就会设置零标志, 然后在目的寄存器中留下一个不确定的值。注意, 在这些指令执行完之后, 应该立即测试零标志, 从而验证目的寄存器的值。例如:

```
mov( SomeValue, ebx );      // Value whose bits we want to check.
bsf( ebx, eax );            // Put position of first set bit in eax.
jz NoBitsSet;               // Branch if SomeValue contains 0.
mov( eax, FirstBit );       // Save location of first set bit.
.
.
.
```

使用 `bsr` 指令的方式基本相同, 只是它计算的是操作数中末置 1 位的位置(也就是说, 在它从高位位向低位位扫描的时候, 所碰到的第一个被置 1 的位)。

80x86 的 CPU 没有提供指令来定位第一个包含 0 的位。但是, 可以首先将源操作数(或者如果一定要保留源操作数的值, 就使用源操作数的副本)反转, 然后就可以简单地扫描出一个 0 位了。如果反转了源操作数, 那么所找到的第一个 1 位就是对应的初始操作数中的第一个 0 位。

`bsf` 和 `bsr` 指令是非常复杂的 80x86 指令。因此, 这些指令会比其他指令慢一些。实际上, 在某些环境下, 使用离散指令定位首置 1 位会更快一些。但是, 因为这些指令的执行时间因 CPU 的不同而变化非常大, 所以在对时间要求严格的代码中使用它们之前, 应该首先测试这些指令的性能。

注意, `bsf` 和 `bsr` 指令不会影响源操作数。有一种很普遍的操作, 就是提取在某个操作数中找到的首(或者末)置 1 位。也就是说, 可能想要在找到一个位之后马上将其清零。如果源操作数是一个寄存器(或者可以将其移到寄存器中), 那么就可以使用 `btr`(或者 `btc`)指令在找到一个位后将它清零。这里是一些用于实现这个结果的代码:

```
bsf( eax, ecx );            // Locate first set bit in eax.
if( @nz )then               // If we found a bit, clear it.
    btr( ecx, eax );        // Clear the bit we just found.
endif;
```

在这个代码序列结束的时候，零标志将指示我们是否找到了一个位(注意，`btr` 不会影响零标志)。或者，也可以给上面的 `if` 语句中加入一个 `else` 部分，来处理在这段代码序列开始时源操作数(`EAX`)中包含 0 的情况。

因为 `bsf` 和 `bsr` 指令只支持 16 位和 32 位操作数，所以必须以不同的方式计算 8 位操作数中首位的位置。有两种较合理的方案。当然，首先可以将一个 8 位操作数零扩展成 16 位或者 32 位，然后对这个操作数使用 `bsf` 或者 `bsr` 指令。另一种选择是创建一个查找表，其中每一项都包含数值中一个位的编号，而该数值用作表的索引；然后就可以使用 `xlat` 指令来“计算”这个数值中首位的位置(注意，必须将数值 0 作为特殊情况来处理)。另一种解决方法是使用本节开始时的移位算法；对于一个 8 位操作数来说，这并不是一个完全无效的解决方法。

`bsf` 和 `bsr` 指令有一种很有趣的用途，那就是用于填充一个字符集，使用的是这个字符集中从最小值字符到最大值字符之间的所有字符。例如，假设一个字符集包含值{ ‘A’, ‘M’, ‘a’, ..., ‘n’, ‘z’ }；如果要填充这个字符集中的空缺，我们的候选值是{ ‘A’, ... ‘z’ }。要计算这个新的集合，我们可以使用 `bsf` 来确定集合中第一个字符的 ASCII 码，使用 `bsr` 来确定这个集合中最后一个字符的 ASCII 码。完成了以后，我们就可以将这两个 ASCII 码作为 `cs.rangeChar` 函数的输入来计算新的集合。

还可以使用 `bsf` 和 `bsr` 指令来确定一个同位值的大小，不过需要假设在操作数中只有一个同位值。只要定位第一位和最后一位(就像上面那样)，就可以计算两个值的差(加 1)。当然，这个方案只有在数值中首置 1 位和末置 1 位之间没有插入 0 的情况下，才是合法的。

10.8 位的计数

在上一节中，最后一个示例演示了一个一般性问题的一种特殊情况：位的计数。遗憾的是，该示例有着严重的局限：它只能对源操作数中的全 1 位进行计数。这一节将讨论该问题更一般的解决方法。

在 Internet 新闻组中，几乎每天都有人询问如何对寄存器中的位进行计数。无疑，这是一种普遍的需求，因为很多汇编语言教师将这个问题作为一个任务分配给学生，教他们使用移位和循环移位指令。这些老师希望以下面的方法解决问题：

```
// BitCount1:
//
// Counts the bits in the eax register, returning the count in ebx.

        mov( 32, cl );           // Count the 32 bits in eax.
        sub( ebx, ebx );         // Accumulate the count here.
CntLoop: shr( 1, eax );          // Shift next bit out of eax and into Carry.
        adc( 0, bl );            // Add the carry into the ebx register.
        dec( cl );              // Repeat 32 times.
        jnz CntLoop;
```

这里有一种“技巧”，就是这段代码使用了 `adc` 指令将进位标志的值加到 `BL` 寄存器中。因为计数的值会比 32 小，所以结果完全可以放在 `BL` 中。

不管它是不是技巧性的代码，这个指令序列并不是特别快。稍作分析就可以看出，上面的循

环总是执行 32 次, 所以这个代码序列会执行 130 条指令(每次迭代 4 条指令, 加上两条额外的指令)。可能有人会问, 是不是还有更高效的解决方法; 答案是肯定的。下面的代码摘自 AMD Athlon 优化指南, 它提供了一种更快捷的解决方法(关于其描述请看算法的注释):

```
// bitCount
//
// Counts the number of "1" bits in a dword value.
// This function returns the dword count value in eax.

procedure bitCount( BitsToCnt:dword ); nodisplay;

const
    EveryOtherBit      := $5555_5555;
    EveryAlternatePair := $3333_3333;
    EvenNibbles        := $0f0f_0f0f;

begin_bitCount;

    push( edx );
    mov( BitsToCnt, eax );
    mov( eax, edx );

    // Compute sum of each pair of bits
    // in eax. The algorithm treats
    // each pair of bits in eax as a
    // z-bit number and calculates the
    // number of bits as follows (description
    // is for bits 0 and 1, it generalizes
    // to each pair):
    //
    // edx = Bit1 Bit0
    // eax = 0-Bit1
    //
    // edx-eax = 00 if both bits were 0.
    // 01 if Bit0=1 and Bit1=0.
    // 01 if Bit0=0 and Bit1=1.
    // 10 if Bit0=1 and Bit1=1.
    //
    // Note that the result is left in edx.
    shr( 1, eax );
    and( EveryOtherBit, eax );
    sub( eax, edx );

    // Now sum up the groups of 2 bits to
    // produces sums of 4 bits. This works
    // as follows:
    //
    // edx = bits 2,3,6,7,10,11,14,15,...,30,31
    // in bit positions 0,1,4,5,...,28,29 with
    // zeros in the other positions.
    //
```



```
// eax = bits 0,1,4,5,8,9,...28,29 with zeros
// in the other positions.
//
// edx+eax produces the sums of these pairs of bits.
//
// The sums consume bits 0,1,2,4,5,6,8,9,10,...28,29,30
// in eax with the remaining bits all containing 0.

mov( edx, eax );
shr( 2, edx );
and( EveryAlternatePair, eax );
and( EveryAlternatePair, edx );
add( edx, eax );

// Now compute the sums of the even and odd nibbles in the
// number. Because bits 3,7,11,etc. in eax all contain
// 0 from the above calculation, we don't need to AND
// anything first, just shift and add the two values.
// This computes the sum of the bits in the 4 bytes
// as four separate value in eax (ah contains number of
// bits in original ah, ah contains number of bits in
// original ah, etc.)

mov( eax, edx );
shr( 4, eax );
add( edx, eax );
and( EvenNibbles, eax );

// Now for the tricky part.
// We want to compute the sum of the 4 bytes
// and return the result in eax. The following
// multiplication achieves this. It works
// as follows:
// (1) the $01 component leaves bits 24..31
// in bits 24..31.
//
// (2) the $100 component adds bits 17..23
// into bits 24..31.
//
// (3) the $1_0000 component adds bits 8..15
// into bits 24..31.
//
// (4) the $1000_0000 component adds bits 0..7
// into bits 24..31.
//
// Bits 0..23 are filled with garbage, but bits
// 24..31 contain the actual sum of the bits
// in eax's original value. The shr instruction
// moves this value into bits 0..7 and zeros
// out the H.O. bits of eax.

intmul( $0101_0101, eax );
```



```

    shr( 24, eax );
    pop( edx );

end bitCount;

```

10.9 倒置位串

另外有一种编程项目是非常普遍的，并且就其功能本身来讲也很有用，这就是倒置一个操作数中所有位的程序。也就是说，它将低位位与高位位互换，将第1位与高位位接下来的那一位互换等。对于这样的任务，通常希望以下面的方法解决：

```

// Reverse the 32-bits in eax, leaving the result in ebx:

    mov( 32, cl );
RvsLoop:  shr( 1, eax );      // Move current bit in eax to the carry flag.
          rcl( 1, ebx );    // Shift the bit back into ebx, backwards.
          dec( cl );
          jnz RvsLoop;

```

和前面的示例一样，这段代码要重复循环 32 次而成为 129 条指令，所以性能会受到一些影响。通过对循环进行展开，可以将指令减少到 64 条，但这样的实现代价仍然很高。

像往常一样，对于一个优化问题，最好的解决方法就是寻求一个更好的算法，而不是去试着选择一些更快的指令来加速一部分代码。但是，在对位进行操作的时候，使用一点点技巧都会节省很多工作。例如，在上一节中，我们使用了一个更复杂的算法来替换那个简单的“移位并计数”的算法，这样就能够加速对位串中位的计数。在上面的示例中，我们再次碰到了一个非常简单的算法，它使用一个循环，重复每个数字中的 1 位。那么现在的问题就是：“我们能不能找到一个算法，不用执行 129 条指令也能够倒置 32 位寄存器中的位？”答案是肯定的，其中的技巧就在于要尽可能多地进行并行处理。

假设我们要交换一个 32 位数值中的偶数位和奇数位。使用下面的代码就可以简单地交换 EAX 中的偶数位和奇数位：

```

mov( eax, edx );      // Make a copy of the odd bits.
shr( 1, eax );        // Move even bits to the odd positions.
and( $5555_5555, edx ); // Isolate the odd bits.
and( $5555_5555, eax ); // Isolate the even bits.
shl( 1, edx );        // Move the odd bits to even positions.
or( edx, eax );       // Merge the bits and complete the swap.

```

当然，交换偶数位和奇数位虽然比较有趣，但并不能解决像倒置一个数字中的所有位这样复杂的问题。但是它的确为我们指出一种途径。例如，如果在执行了前面的代码序列，将相邻的两位交换以后，也就交换了 32 位数值中所有半字节的位。交换相邻两位的方法和上面的方法类似，其代码为：

```

mov( eax, edx );      // Make a copy of the odd-numbered bit pairs.
shr( 2, eax );        // Move the even bit pairs to the odd position.

```

```

and( $3333_3333, edx ); // Isolate the odd pairs.
and( $3333_3333, eax ); // Isolate the even pairs.
shl( 2, edx ); // Move the odd pairs to the even positions.
or( edx, eax ); // Merge the bits and complete the swap.

```

在完成了前面这个代码序列以后，将 32 位寄存器中的相邻半字节交换。同样，唯一的区别在于位掩码和移位的长度，代码如下：

```

mov( eax, edx ); // Make a copy of the odd-numbered nibbles.
shr( 4, eax ); // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx ); // Isolate the odd nibbles.
and( $0f0f_0f0f, eax ); // Isolate the even nibbles.
shl( 4, edx ); // Move the odd pairs to the even positions.
or( edx, eax ); // Merge the bits and complete the swap.

```

您可能会看到这种模式的发展趋势，并且能够推测出接下来的两个步骤是必须对这个对象中的字节进行交换，然后是字。可以使用像上面这样的代码，但是还有更好的方法：使用 **bswap** 指令。**bswap**(字节交换)指令使用下面的语法：

```
bswap( reg32 );
```

这条指令在给定的 32 位寄存器中交换第 0 个字节和第 3 个字节，同时也交换第 1 个字节和第 2 个字节。这条指令的主要用途是将数据在所谓的小尾编址和大尾编址两种格式之间转换²。虽然在这里，并不特别需要这条指令来做这件事，但是 **bswap** 指令的确能够交换一个 32 位对象中的字节和字，其方式正好就是想用它来倒置位的方式。可以不使用另外的 12 条指令来交换字节，然后交换字，而是简单地使用一条 **bswap(eax);** 指令，在上面的指令之后完成这一工作。最后的代码序列为：

```

mov( eax, edx ); // Make a copy of the odd bits in the data.
shr( 1, eax ); // Move the even bits to the odd positions.
and( $5555_5555, edx ); // Isolate the odd bits.
and( $5555_5555, eax ); // Isolate the even bits.
shl( 1, edx ); // Move the odd bits to the even positions.
or( edx, eax ); // Merge the bits and complete the swap.

mov( eax, edx ); // Make a copy of the odd numbered bit pairs.
shr( 2, eax ); // Move the even bit pairs to the odd position.
and( $3333_3333, edx ); // Isolate the odd pairs.
and( $3333_3333, eax ); // Isolate the even pairs.
shl( 2, edx ); // Move the odd pairs to the even positions.
or( edx, eax ); // Merge the bits and complete the swap.

mov( eax, edx ); // Make a copy of the odd numbered nibbles.
shr( 4, eax ); // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx ); // Isolate the odd nibbles.
and( $0f0f_0f0f, eax ); // Isolate the even nibbles.

```

2 在小尾编址系统中，也就是 80x86 的原始格式中，对象的低位字节出现在存储器中最低的地址上。在很多 RISC 处理器所使用的大尾编址系统中，却是对象的高位字节出现在存储器中最低的地址上。**bswap** 指令能够在两种数据格式之间作转换。


```

shl( 4, edx );      // Move the odd pairs to the even positions.
or( edx, eax );     // Merge the bits and complete the swap.

bswap( eax );       // Swap the bytes and words.

```

这个算法只需要 19 条指令，而且它比移位循环执行起来快得多。当然，这个代码序列的确占用了更多的存储空间。如果要设法节省存储空间，而不是时钟周期的话，那么循环可能会是一种更好的解决方案。

10.10 合并位串

另一种常见的位串操作是，通过合并或者交叉插入两个不同源操作数中的位而产生一个单独的位串。下面的示例代码序列通过交替地合并两个 16 位的串，创建了一个 32 位的位串：

```

// Merge two 16-bit strings into a single 32-bit string.
// ax -- Source for even numbered bits.
// bx -- Source for odd numbered bits.
// cl -- Scratch register.
// edx -- Destination register.

mov( 16, cl );
MergeLp: shrd( 1, eax, edx ); // Shift a bit from eax into edx.
         shrd( 1, ebx, edx ); // Shift a bit from ebx into edx.
         dec( cl );
         jne MergeLp;

```

这个特殊的示例将两个 16 位的值合并起来，将它们的位交替放在结果数值中。为了使这段代码能够更快速地实现，展开其中的循环可能是最好的选择，因为这样能够削减一半的指令。

通过一些细微的改动，我们也可以将 4 个 8 位的数值合并起来，或者也可以使用其他位序列来产生结果。例如，下面的代码从 EAX 中复制第 0~5 位，从 EBX 中复制第 0~4 位，从 EAX 中复制第 6~11 位，从 EBX 中复制第 5~15 位，最后从 EAX 中复制第 12~15 位：

```

shrd( -6, eax, edx );
shrd( 5, ebx, edx );
shrd( 6, eax, edx );
shrd( 11, ebx, edx );
shrd( 4, eax, edx );

```

10.11 提取位串

当然，我们可以简单地完成合并两个位流的反操作：也就是，我们可以在一个位串中提取位并将它们分布到多个目的位置上。下面的代码取出 EAX 中的 32 位值，并交替地将每一位分布到 BX 和 DX 寄存器中：


```

        mov( 16, cl );          // Count the loop iterations.
ExtractLp: shr( 1, eax );       // Extract even bits to (e)bx.
        rcr( 1, ebx );
        shr( 1, eax );         // Extract odd bits to (e)dx.
        rcr( 1, edx );
        dec( cl );             // Repeat 16 times.
        jnz ExtractLp;
        shr( 16, ebx );        // Need to move the results from the H.O.
        shr( 16, edx );        // bytes of ebx/edx to the L.O. bytes.

```

这个代码序列执行了 99 条指令。这并不可怕,但是我们也许能够做得更好一些,使用一个更好的算法来并行地提取位。利用我们曾经在倒置寄存器的位时所用到的技术,就可以提出下面这个算法,将偶数位重新分配到 EAX 的低位字上,而奇数位分配到 EAX 的高位字上。

```

// Swap bits at positions (1,2), (5,6), (9,10), (13,14), (17,18),
// (21,22), (25,26), and (29,30).

        mov( eax, edx );
        and( $9999_9999, eax ); // Mask out the bits we'll keep for now.
        mov( edx, ecx );
        shr( 1, edx );          // Move 1st bits in tuple above to the
        and( $2222_2222, ecx ); // correct position and mask out the
        and( $2222_2222, edx ); // unneeded bits.
        shl( 1, ecx );          // Move 2nd bits in tuples above.
        or( edx, ecx );         // Merge all the bits back together.
        or( ecx, eax );

// Swap bit pairs at positions ((2,3), (4,5)), ((10,11), (12,13)), etc.

        mov( eax, edx );
        and( $c3c3_c3c3, eax ); // The bits we'll leave alone.
        mov( edx, ecx );
        shr( 2, edx );
        and( $0c0c_0c0c, ecx );
        and( $0c0c_0c0c, edx );
        shl( 2, ecx );
        or( edx, ecx );
        or( ecx, eax );

// Swap nibbles at nibble positions (1,2), (5,6), (9,10), etc.

        mov( eax, edx );
        and( $f00f_f00f, eax );
        mov( edx, ecx );
        shr( 4, edx );
        and( $0f0f_0f0f, ecx );
        and( $0f0f_0f0f, edx );
        shl( 4, ecx );
        or( edx, ecx );
        or( ecx, eax );

```

```
// Swap bits at positions 1 and 2.
```

```
ror( 8, eax );
xchg( al, ah );
rol( 8, eax );
```

这个代码序列需要 30 条指令。乍一看，好像它就是最好的，因为最初的循环要执行 64 条指令。但是，这段代码并不像看上去那么好。毕竟，如果我们愿意编写这种代码的话，为什么不将上面的循环展开 16 次呢？那个代码序列只需要 64 条指令。因此前一种算法的复杂性在指令数方面并没有带来很大的收益。至于要知道哪个序列更快，就不得不对它们进行计时来断定结果。但是，shrd 指令并不是特别快，其他代码序列中的指令也是一样。这个示例出现在这里并不是为了显示一个更好的算法，而是为了演示编写那些技巧性特别高的代码并不是总能获得很大的性能突破。

提取其他的位组合将留作读者的练习。

10.12 搜索位模式

另一种与位相关的操作是在一个位串中搜索一种特定的位模式。例如，可能想要从位串的某个特殊位置算起，定位%1011 首次出现的位索引。在本节中，我们将探讨一些实现这种任务的简单算法。

要搜索一种特定的位模式，我们需要知道 4 件事：①要搜索的模式(pattern)，②要搜索模式的长度，③将要进行搜索的位串(source)，以及④将要搜索的位串的长度。搜索背后最基本的思想就是基于位模式的长度创建一个掩码，并使用这个值对源操作数的一个副本进行屏蔽。然后我们就可以直接将位模式与被屏蔽的源操作数进行比较，看是否相等。如果它们相等，问题就解决了；如果不相等，那么就将一个位置计数器加 1，将源操作数右移一位，然后再试一次。这种操作将重复 $\text{length}(\text{source}) - \text{length}(\text{pattern})$ 次。如果在这么多次尝试之后，仍然没有检测到那个位模式，算法就失败了(因为我们已经用尽了源操作数中所有能够匹配位模式长度的位)。这里有一种简单的算法，在 EBX 寄存器中搜索一个 4 位的位模式：

```
mov( 28, cl );           // 28 attempts because 32-4 =28
                          // (len(src)-len(pat)).
mov( %1111, ch );        // Mask for the comparison.
mov( pattern, al );       // Pattern to search for.
and( ch, al );            // Mask unnecessary bits in al.
mov( source, ebx );       // Get the source value.
ScanLp: mov( bl, dl );     // Copy the L.O. 4 bits of ebx
and( ch, dl );            // Mask unwanted bits.
cmp( dl, al );            // See if we match the pattern.
jz Matched;
dec( cl );                // Repeat the specified number of times.
shll( 1, ebx );
jnz ScanLp;
```



```
// Do whatever needs to be done if we failed to match the bit string.
```

```
    jmp Done;
```

```
Matched:
```

```
// If we get to this point, we matched the bit string. We can
```

```
// compute the position in the original source as 28-cl.
```

```
Done:
```

位串扫描是串匹配的一种特殊情况。串匹配在计算机科学中已经研究成熟了，并且用于串匹配的很多算法都同样适用于位串匹配。这样的算法超出本章的讨论范围，但为了对其工作过程有个了解，可以先对位模式和当前源位调用一些函数(如 xor 或者 sub)，然后使用其结果作为一个查找表的索引，来确定能够跳过多少位。这样的算法允许您在扫描循环的每次迭代中跳过多位，而不是只移动一次(就像前面的算法中所做的那样)。

10.13 HLA 标准库的位模块

HLA 标准库提供了 bits.hhf 模块，该模块提供了一些与位相关的函数，包括在这一章中我们所学习的很多算法的内置函数。本节将描述 HLA 标准库中的这些函数。

```
procedure bits.cnt( b:dword ); @returns( "eax" );
```

这个过程返回由参数 b 给出的数值中 1 位的数量，它将这个计数值返回在 EAX 中。要对参数值中的 0 位进行计数的话，就要在把参数值传递给 bits.cnt 之前，先对其进行反转。如果想要对一个 16 位操作数中的位进行计数，只要在调用这个函数之前，将其零扩展成 32 位即可。这里有两个示例：

```
// Compute the number of bits in a 16-bit register:
```

```
    pushw( 0 );
```

```
    push( ax );
```

```
    call bits.cnt;
```

```
// If you prefer to use a higher-level syntax, try the following:
```

```
    bits.cnt( #{pushw(0);-push(ax);}# );
```

```
// Compute the number of bits in a 16-bit memory location:
```

```
    pushw( 0 );
```

```
    push( mem16 );
```

```
    call bits.cnt;
```

如果想要计算一个 8 位操作数中的位数，那么这样做可能会快一些，就是编写一个简单的循环，将源操作数中的所有位进行循环移位，并将进位加到累加和中。当然，如果不严格要求性能的话，也可以将这个字节零扩展成 32 位，然后调用 bits.cnt 过程。

```
procedure bits.distribute( source:dword; mask:dword;dest: dword );
    @returns( "eax" );
```

这个函数取 `source` 的低 n 位, 其中 n 是 `mask` 中 1 位的数量, 然后将这些位插入到 `dest` 中由 `mask` 中的 1 位所指定的位置上(也就是, 和这一章前面出现的分布算法相同)。这个函数不会改变 `dest` 中与 `mask` 值中的 0 相对应的位。这个函数也不会影响参数 `dest` 实际的值; 它会在 EAX 寄存器中返回新的数值。

```
procedure bits.coalesce( source:dword; mask:dword );
    @returns( "eax" );
```

这个函数是 `bits.distribute` 的反函数。它将源操作数中对应于 `mask` 中包含 1 的位置上的所有位提取出来。这个函数将这些位(右对齐地)接合起来, 放在结果的低位位, 然后把结果返回给 EAX。

```
procedure bits.extract( var d:dword );
    @returns( "eax" );// Really a macro.
```

这个函数提取 `d` 中的首置 1 位, 它从第 0 位开始搜索, 将这个位的索引返回在 EAX 寄存器中; 在这种情况下, 该函数也会返回被清零的零标志。还会将操作数中的那一位清零。如果 `d` 包含的是 0, 那么这个函数就返回被置 1 的零标志, 而 EAX 中将包含 -1。

注意, HLA 实际上将这个函数作为一个宏而不是过程来实现。这就意味着可以传递任何双字操作数作为参数(也就是, 一个存储器或者寄存器操作数)。但是, 如果将 EAX 作为参数传递给它, 结果就会是未定义的(因为这个函数要计算 EAX 中的位数)。

```
procedure bits.reverse32( d:dword ); @returns( "eax" );
procedure bits.reverse16( w:word ); @returns( "ax" );
procedure bits.reverse8( b:byte ); r@returns( "al" );
```

这 3 个例程都将其参数值中的位倒置, 然后将其值返回到累加寄存器中(AL/AX/EAX)。应根据数据的大小调用适当的例程。

```
procedure bits.merge32( even:dword; odd:dword ); @returns( "edx:eax" );
procedure bits.merge16( even:word; odd:word ); @returns( "eax" );
procedure bits.merge8( even:byte; odd:byte ); @returns( "ax" );
```

这些例程将两个位流合并, 产生一个值, 其大小为两个参数的合并值。参数 `even` 中的位将占用结果的偶数位; 参数 `odd` 中的位将占用结果的奇数位。请注意这些函数会根据字节、字和双字的参数值, 返回 16、32 或者 64 位的结果。

```
procedure bits.nibbles32( d:dword ); @returns( "edx:eax" );
procedure bits.nibbles16( w:word ); @returns( "eax" );
procedure bits.nibbles8( b:byte ); @returns( "ax" );
```

这些例程将参数中的每一个半字节都提取出来, 并将其放在单独的字节中。`bits.nibbles8` 函数从参数 `b` 中提取两个半字节, 将低位半字节放在 AL 中, 高位半字节放在 AH 中。`bits.nibbles16` 函数在 `w` 中提取 4 个半字节, 把它们分别放到 EAX 的 4 个字节中。可以使用 `bswap` 或者 `ror` 指

令获得对 EAX 的高位字中的半字节的访问。`bits.nibbles32` 在 EAX 中提取 8 个半字节，并将它们分布到 EDX:EAX 的 8 个字节中。第 0 个半字节占用 AL，而第 7 个半字节占用 EDX 的高位字节。同样，可以使用 `bswap` 或者循环移位指令来访问 EAX 和 EDX 中的高字节。

10.14 更多信息

在开发位操作算法的时候，可以参考 <http://webster.cs.ucr.edu/> 和 <http://www.artofasm.com/> 上关于本书的信息，或许对您非常有用。特别是，关于数字设计的一章中讨论了布尔代数，在对位进行操作的时候，这个主题是必需的。HLA 标准库参考手册包含了更多关于 HLA 标准库位操作例程的信息。正如在关于位计数的一节中所指出的，AMD Athlon 优化指南中也包含了一些基于位计算的优化算法。最后，如果要学习更多关于位搜索算法的知识，就应该参考关于数据结构和算法的教材，学习其中关于串匹配算法的知识。

第 11 章

字符串指令



字符串是一系列存储在连续存储单元中的值的集合。字符串通常是字节、字或双字(在 80386 或更高级的处理器上)的数组。80x86 微处理器系列支持若干专为处理字符串而设计的指令。本章将介绍这些指令的用法。

80x86 CPU 可以处理 3 种类型的字符串,它们分别是字节字符串、字字符串和双字字符串。这些指令可以移动字符串、进行字符串比较、在一个字符串中搜索指定值、将字符串初始化为一个固定值,以及其他一些与字符串相关的基本操作。80x86 字符串指令还有助于处理数组、表和记录。用户可以使用字符串指令对字符串进行赋值和比较。使用字符串指令可以大幅度提高数组处理代码的运行速度。

11.1 80x86 字符串指令

80x86 系列的所有成员都支持 5 种不同的字符串指令: `movsx`、`cmpsx`、`scasx`、`lodsx` 和 `stosx`¹(x 为 `b`、`w` 或 `d`, 分别代表字节、字或双字。本文中讨论一般意义上的这些指令时,通常将省略后缀 x)。用户可以使用这些字符串基本指令来构建其他更复杂的字符串操作。本节内容的主题是如何使用这 5 种指令。

For `MOVSB`:

```
movsb();  
movsw();  
movsd();
```

¹ 80x86 处理器还支持另外两种字符串指令: `ins` 和 `outs`。这两个指令分别用于从一个输入端口输入数据字符串和向一个输出端口输出数据字符串。我们将不会考虑这些指令,因为这些指令是特权指令,用户无法在标准的 32 位操作系统应用程序中执行它们。


```

For CMPS:
    cmpsb();
    cmpsw();
    cmpsd();

For SCAS:
    scasb();
    scasw();
    scasd();

For STOS:
    stosb();
    stosw();
    stosd();

For LODS:
    lodsb();
    lodsw();
    lodsd();

```

11.1.1 字符串指令的操作过程

字符串指令操作于内存块(线性连续的阵列)上。例如, `movs` 指令将一系列顺序排列的字节从一处存储单元移动到另一处。`cmps` 指令对两块内存块进行比较。`scas` 指令扫描内存块中是否存在特定的值。这些字符串指令通常具有 3 个操作数: 目的块地址、源块地址以及元素计数(可选的)。例如, 当使用 `movs` 指令进行字符串复制时, 用户需要一个源地址、一个目的地址和一个计数(要移动的字符串元素的个数)。

与其他进行存储器操作的指令不同, 字符串指令没有任何显式的操作数。字符串指令的操作数为:

- ESI(源索引)寄存器
- EDI(目的索引)寄存器
- ECX(计数)寄存器
- AL/AX/EAX 寄存器
- FLAGS 寄存器中的方向标志

例如, `movs`(移动字符串)指令的一个变体将一个字符串从 ESI 指定的源地址处复制到由 EDI 指定的目的地址, 该字符串长度为 ECX。类似地, `cmps` 指令将 ESI 所指向的字符串与 EDI 所指向的字符串进行比较, 字符串长度为 ECX。

不是所有的字符串指令都具有源和目的存储器操作数(只有 `movs` 和 `cmps` 支持两个操作数)。例如, `scas` 指令(字符串扫描)就是将累加器(AL、AX 或 EAX)中的值与存储器中的值进行比较。

11.1.2 rep/repe/repz 和 repnz/repne 前缀

字符串指令本身并不对数据字符串进行处理。例如, `movs` 指令只移动一个字节、字或双字。执行 `movs` 指令时, 该指令将忽略 ECX 寄存器中的值。表示重复的前缀告知 80x86 进行一次多字节的字符串操作。这些表示重复的前缀的语法为:

For MOVS:

```
rep.movsb();
rep.movsw();
rep.movsd();
```

For CMPS:

```
repe.cmpsb(); // Note:repz is a synonym for repe.
repe.cmpsw();
repe.cmpsd();

repne.cmpsb(); // Note:repnz is a synonym for repne.
repne.cmpsw();
repne.cmpsd();
```

For SCAS:

```
repe.scasb(); // Note:repz is a synonym for repe.
repe.scasw();
repe.scasd();

repne.scasb(); // Note:repnz is a synonym for repne.
repne.scasw();
repne.scasd();
```

For STOS:

```
rep.stosb();
rep.stosw();
rep.stosd();
```

使用 `lods` 指令时，通常不需要使用表示重复的前缀。

在一个字符串指令前指定表示重复的前缀时，字符串指令将重复 `ECX` 次²。不使用表示重复的前缀时，指令只对一个字节、字或双字进行操作。

可以在一条字符串指令中使用表示重复的前缀来处理整个字符串。也可以只使用字符串指令而不使用表示重复的前缀，用字符串基本操作来合成功能更强大的字符串操作。

11.1.3 方向标志

除了 `ESI`、`EDI`、`ECX` 和 `AL/AX/EAX` 寄存器，还有另一个寄存器控制 80x86 字符串指令的操作——`EFLAGS` 寄存器。特别是标志寄存器中的方向标志(direction flag)控制着 CPU 如何处理字符串。

如果方向标志为 0，那么在处理完每个字符串元素后，CPU 将增加 `ESI` 和 `EDI` 的值。例如，执行 `movs` 指令时将 `ESI` 中的字节、字或双字移动到 `EDI` 中，然后将 `ESI` 和 `EDI` 加 1、2 或 4。当在该指令前指定了 `rep` 前缀时，CPU 将为字符串的每个元素(`ECX` 中的值指明元素的个数)增加 `ESI` 和 `EDI`。结束时，`ESI` 和 `EDI` 寄存器将指向字符串后的第一项。

如果设置了方向标志，80x86 在处理完每个字符串元素后(这里，`ECX` 再次指明字符串元素的个数)将减少 `ESI` 和 `EDI`。在一次重复字符串操作后，如果设置了方向标志，那么 `ESI` 和 `EDI` 寄存器将指向字符串之前的第一个字节、字或双字。

可以使用 `cld`(清零方向标志)与 `std`(置 1 方向标志)指令来改变方向标志的值。在一个过程内部

² `cmps` 指令除外，该指令的重复次数最多为 `ECX` 寄存器中所指定的值。

使用这些字符串指令时, 请记住这些指令将更改机器的状态。因此, 您可能需要在执行该过程时保存方向标志。下面的例子显示了可能会遇到的各种问题:

```
procedure Str2; @nodisplay;
begin Str2;

    std();
    << Do some string operations. >>
    .
    .
    .
end Str2;

    .
    .
    .
    cld();
    << Do some operations. >>
    Str2();
    << Do some string operations requiring D=0. >>
```

这样的代码不能正常工作。调用的代码认为在 Str2 返回时方向标志为 0。然而, 实际情况并非如此。因此, 在调用 Str2 后执行的字符串操作将不可能正确运行。

有两种方法来解决这个问题。第一种也是最明显的一种解决办法是, 在执行一条或多条字符串指令构成的序列前, 总是立即插入 cld 或 std 指令。这可以确保在代码中方向标志的设置总是正确的。另一种方法是使用 pushfd 和 popfd 指令来保存和恢复方向标志。使用了这两种技巧后, 上面的代码就变成下面这样。

总是在一个字符串指令前使用 cld 或 std 指令的情况:

```
procedure Str2; @nodisplay;
begin Str2;

    std();
    << Do some string operations. >>
    .
    .
    .
end Str2;

    .
    .
    .
    cld();
    << Do some operations. >>
    Str2();
    cld();
    << Do some string operations requiring D=0. >>
```

保存并恢复标志寄存器的情况:

```
procedure Str2; @nodisplay;
begin Str2;
```



```

        pushfd();
        std();
    << Do some string operations. >>
    .
    .
    .
    popfd();
end Str2;
.
.
.
    cld();
    << Do some operations. >>
    Str2();
    << Do some string operations requiring D=0. >>

```

如果使用 `pushfd` 和 `popfd` 指令来保存并恢复标志寄存器，那么就要记住，正在保存或恢复的是所有的标志。这样就使得在其他标志位中返回信息变得有些困难。例如，如果使用 `pushfd` 和 `popfd` 来保存过程中的方向标志，就需要做一些工作，以便在进位标志中返回一个错误条件。

对于上述代码的问题，第三种解决方法是，除非要执行需要将方向标志置 1 的特殊序列，否则就要保证在其他情况下方向标志总是被清零。例如，很多库调用和某些操作系统在被调用的时候，总是假设方向标志为空。例如，大部分标准 C 库函数就是以这种方式工作的。可以遵循这种惯例，即总是假设方向标志为空，然后保证在一个需要使用 `std` 的序列后立即将该标志清零。

11.1.4 movs 指令

`movs` 指令使用的语法如下：

```

movsb()
movsw()
movsd()
rep.movsb()
rep.movsw()
rep.movsd()

```

`movsb` 指令将取出地址寄存器 `ESI` 所指向的字节，并将其存储于地址寄存器 `EDI` 所指向的字节中，然后分别将 `ESI` 和 `EDI` 中的值加 1 或减 1。如果存在 `rep` 前缀，CPU 将检查 `ECX` 是否包含 0。如果不包含，则将字节从 `ESI` 移至 `EDI`，并减少 `ECX` 寄存器的值。这样的过程一直重复到 `ECX` 变成 0 为止。如果在初始执行过程中，`ECX` 包含 0 的话，`movs` 指令就不会复制任何数据字节。

`movsw` 指令将取出地址寄存器 `ESI` 所指向的字，并将其存储于地址寄存器 `EDI` 所指向的字中，然后分别将 `ESI` 和 `EDI` 中的值加 2 或减 2。如果有一个 `rep` 前缀，那么 CPU 就会重复这个过程，重复的次数由 `ECX` 所指定。

`movsd` 指令对双字进行一种类似的操作。对于每次数据移动，它都会分别将 `ESI` 和 `EDI` 中的值加 4 或减 4。

如果使用了 `rep` 前缀，`movsb` 指令就会移动一定数量的字节，这个数量由 `ECX` 寄存器所指定。下面的代码段将从 `CharArray1` 中复制 384 个字节到 `CharArray2` 中：

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();

```

如果使用 `movsw` 来替换 `movsb`, 那么前面的代码就会移动 384 个字(768 个字节), 而不是 384 个字节:

```

WordArray1: word[ 384 ];
WordArray2: word[ 384 ];
.
.
.
cld();
lea( esi, WordArray1 );
lea( edi, WordArray2 );
mov( 384, ecx );
rep.movsw();

```

要记住, `ECX` 寄存器包含元素计数, 而不是字节计数。在使用 `movsw` 指令的时候, `CPU` 将移动由 `ECX` 寄存器指定的数量的字。类似地, `movsd` 会移动由 `ECX` 寄存器指定的数量的双字, 而不是那么多的字节。

如果在执行一条 `movsb/movsw/movsd` 指令之前, 已经设置了方向标志, `CPU` 就会在移动每个字符串元素之后, 将 `ESI` 和 `EDI` 寄存器中的值分别减 1。这就意味着在执行一条 `movsb`、`movsw` 或者 `movsd` 指令之前, `ESI` 和 `EDI` 寄存器必须指向它们各自字符串中的最后一个元素。例如:

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1[383] );
lea( edi, CharArray2[383] );
mov( 384, ecx );
rep.movsb();

```

虽然有些时候, 按照从尾部到头部的顺序来处理字符串也是有用的(参见 11.1.5 节中关于 `cmps` 的描述)。但一般来说, 处理字符串都是以正向的顺序进行, 因为这样做更为直接一些。有这么一类字符串操作, 对它们来说, 可以从两个方向来处理字符串的能力是绝对必需的, 那就是: 当源数据块和目的数据块发生重叠时移动字符串。考虑下面代码中所发生的情况:

```

CharArray1: byte;
CharArray2: byte[ 384 ];

```

```

cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();

```

这个指令序列将 CharArray1 和 CharArray2 作为两个包含 384 个字节的字符串来处理。但是, CharArray1 中最后 383 个字节与 CharArray2 中前 383 个字节重叠了。现在让我们来逐字节地跟踪这段代码的操作。

当 CPU 执行 movsb 指令时, 它会将 ESI 所指向的字节(CharArray1)复制到 EDI 所指向的字节(CharArray2)中。然后它将 ESI 和 EDI 中的值加 1, 将 ECX 中的值减 1, 再重复这一过程。现在 ESI 寄存器指向的是 CharArray1+1(这正是 CharArray2 的地址), 而 EDI 寄存器指向的是 CharArray2+1。movsb 指令会把 ESI 指向的字节复制到 EDI 指向的字节中。但是, 这正是最初从 CharArray1 的位置上复制过来的字节。所以 movsb 指令就将最初位于 CharArray1 的位置上的值同时复制到了 CharArray2 和 CharArray2+1 的位置上。再一次地, CPU 将 ESI 和 EDI 中的值加 1, 而将 ECX 中的值减 1, 然后重复这样的操作。现在 movsb 指令就会把字节从 CharArray1+2(CharArray2+1)的位置复制到 CharArray2+2 的位置上。但是同样地, 这个值正是最初出现在 CharArray1 的位置上的值。这个循环的每次重复都会将 CharArray1[0]中的下一个元素复制到 CharArray2 数组中下一个可用的位置上。用图来表示的话, 这个过程看上去就像图 11-1 中那样。

最后的结果就是 movsb 指令将 X 复制到整个字符串中。movsb 指令将源操作数复制到存储单元中, 这个存储单元就变成了下一次移动操作的源操作数, 这样就导致了重复的出现。

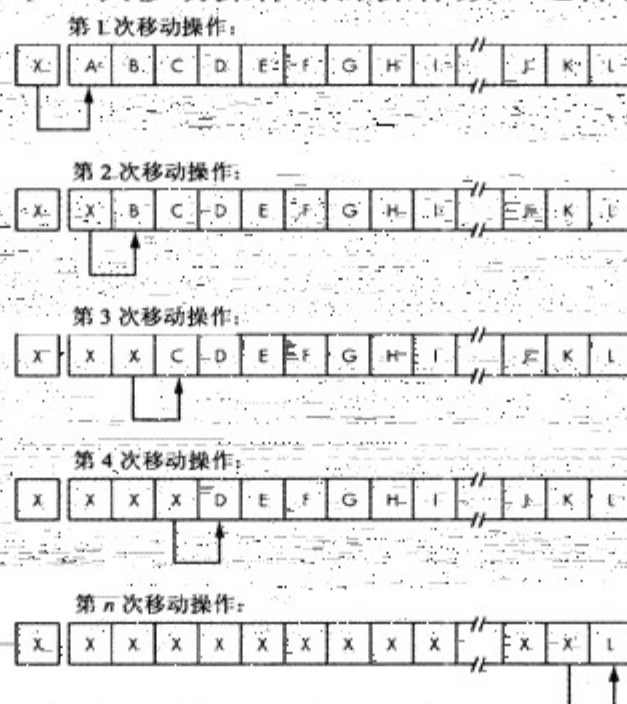


图 H-1 在两个重叠的数组之间(正向)复制数据

如果真的想要将一个数组移动到另一个当中, 并且这两个数组有重叠的话, 就应该将源字符串中的每一个元素都移动到从两个字符串的结束处开始的一个目的字符串中, 如图 11-2 所示。

对方向标志进行置 1 并将 ESI 和 EDI 指向字符串的末端, 那么在两个字符串发生重叠, 而且源字符串的起始地址比目的字符串的地址还低的时候, 这样做就允许您(正确地)将一个字符串移

动到另一个中。如果两个字符串有重叠，并且源字符串的起始地址比目的字符串的地址高，就要将方向标志清零，并且将 ESI 和 EDI 指向两个字符串的起始处。

如果两个字符串没有重叠，那么就可以使用这两种技术中的任何一种在存储器中任意移动字符串。一般来说，在方向标志被清零的时候，操作是最简单的，所以通常应该选择这种技术。

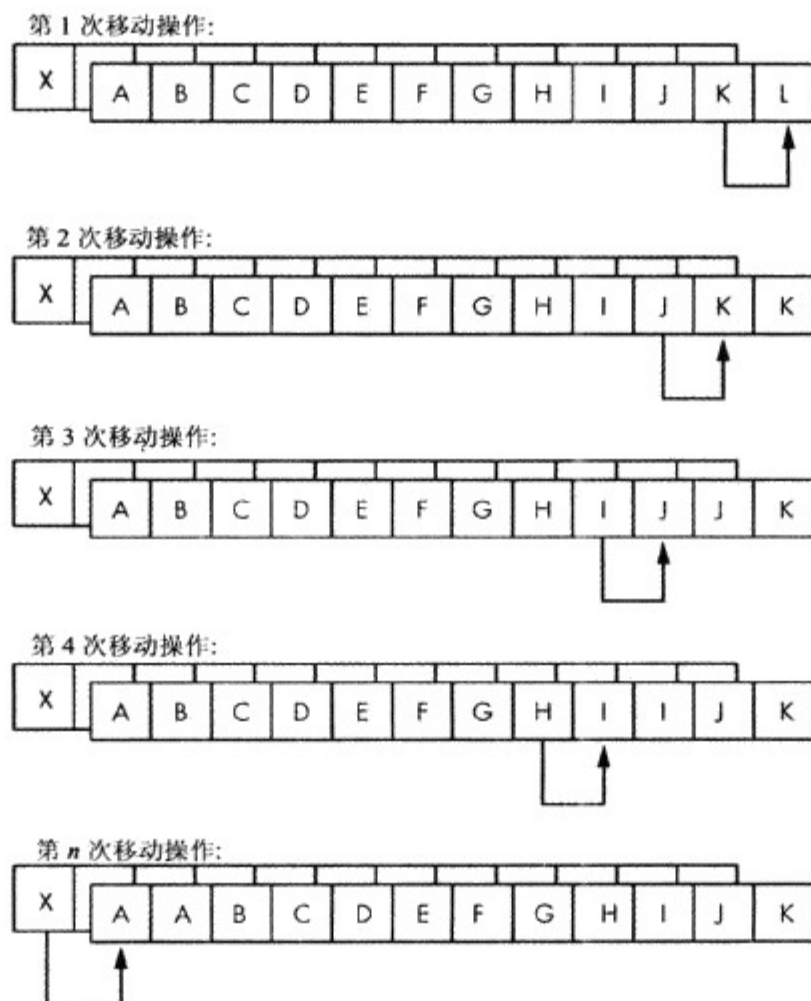


图 11-2 使用反向复制在重叠的数组中复制数据

不应该使用 `movx` 指令以单个字节、字或者双字值填充一个数组。另外一个字符串指令，`stos`，更适合于这个目的。但是，对于那些元素为 1、2 或 4 个字节的数组来说，就可以使用 `movs` 指令将整个数组初始化成第一个元素的内容。

在复制双字的时候，`movs` 指令有时会比复制字节或者字的时候更加高效。在某些系统上，使用 `movsb` 复制一个字节和使用 `movsd` 复制一个双字会耗费相同的时间。因此，如果正在从一个数组向另一个数组移动大量的字节，那么如果可以使用 `movsd` 指令(而不是 `movsb` 指令)，复制操作就能快一些。当然，如果想要移动的字节数是 4 的偶数倍，需要作的修改不大；只要将要复制的字节数除以 4，将得到的值载入 ECX，然后使用 `movsb` 指令即可。如果字节数不能被 4 整除，那么可以使用 `movsd` 指令复制数组中除最后 1 个、2 个或者 3 个字节(也就是将字节数除以 4 以后的余数)以外的所有字节。例如，如果想要高效地移动 4099 个字节，可以使用下面的指令序列来完成：

```
lea( esi, Source );
lea( edi, Destination );
mov( 1024, ecx );    // Copy 1024 dwords = 4096 bytes.
rep.movsd();
movsw();             // Copy bytes 4097 and 4098.
movsb();             // Copy the last byte.
```

使用这种技术来复制数据绝对不会需要多于 3 条的 `movsx` 指令, 因为使用不超过两条的 `movsb` 和 `movsw` 指令, 就可以复制 1 个、2 个或者 3 个字节。如果两个数组是双字边界对齐的话, 上面的方案就是最高效的。如果不是这样, 也许就要将 `movsb` 或 `movsw` 指令(或者二者)移到 `movsd` 指令之前, 让 `movsd` 指令去处理双字对齐的数据。

如果在程序执行之前并不知道要复制的数据块的大小, 也仍然可以使用下面的代码来改善块移动操作的性能:

```
lea( esi, Source );
lea( edi, Dest );
mov( Length, ecx );
shr( 2, ecx );           // Divide by 4.
if( @nz ) then          // Only execute movsd if 4 or more bytes.

    rep.movsd();         // Copy the dwords.
endif;
mov( Length, ecx );
and( %11, ecx );         // Compute (Length mod 4).
if( @nz ) then          // Only execute movsb if #bytes/4 <> 0.

    rep.movsb();         // Copy the remaining 1,2, or 3 bytes.
endif;
```

在很多计算机系统中, `movsd` 指令都提供了可能是最快的方式从一个位置向另一个位置复制大量数据。虽然在某些 CPU 上可能有更快的方式来复制数据, 但是最终, 存储器总线的性能会成为限制因素, 而 CPU 一般要比存储器总线快得多。因此, 除非有一种特殊的系统, 否则编写专门的代码来改善存储器到存储器的传输就是在浪费时间。还要注意, Intel 已经改善了后来的处理器上 `movsx` 指令的性能, 以至于在复制相同数量的字节时, `movsb` 指令和 `movsw` 以及 `movsd` 指令同样高效。因此, 当工作在后来的 80x86 处理器上时, 简单地使用 `movsb` 指令来复制指定数量的字节可能更高效, 并且不用经历上面所述的复杂过程。总之, 如果块移动的速度对您来说很重要, 那么就要尝试不同的方法来选择速度最快的一种(或者如果它们都以相同的速度运行, 就选择最简单的一种, 而这也是有可能发生的)。

11.1.5 cmps 指令

`cmps` 指令用于两个字符串的比较。CPU 将 EDI 所引用的字符串和 ESI 指向的字符串进行比较, 而 ECX 中包含着两个字符串的长度(当使用 `repe` 或 `repne` 前缀时)。就像 `movs` 指令一样, HLA 允许这条指令有以下几种不同的形式:

```
cmpsb();
cmpsw();
cmpsd();

repe.cmpsb();
repe.cmpsw();
repe.cmpsd();
```



```
repne.cmpsb();
repne.cmpsw();
repne.cmpsd();
```

和 `movs` 指令一样, 这些指令需要在 `ESI` 和 `EDI` 寄存器中指定实际的操作数地址。

如果没有表示重复的前缀, `cmps` 指令就会从 `ESI` 所包含的值中减去 `EDI` 所指定位置上的值, 并更新标志。除了更新标志以外, CPU 并不会使用这次减法产生的差。在对两个位置上的值进行了比较之后, `cmps` 会将 `ESI` 和 `EDI` 寄存器分别加减 1、2 或 4(分别对应 `cmpsb/cmpsw/cmpsd`)。如果方向标志被清零, `cmps` 就会将 `ESI` 和 `EDI` 寄存器分别加 1, 否则就将它们减 1。

当然, 如果只使用 `cmps` 指令对存储器中单个的字节、字或者双字进行比较, 就没有真正发挥这条指令的功能。在使用这条指令对整个字符串进行比较时, 它才会大放异彩。有了 `cmps`, 就可以对字符串中的连续元素进行比较, 直到发现了某个匹配项, 或者这些连续的元素都不匹配为止。

要比较两个字符串, 检查它们相等还是不相等, 就必须比较字符串中对应的元素, 直到它们都不匹配为止。考虑下面的字符串:

```
"String1"
"String1"
```

要确定这两个字符串是相等的, 唯一的方法就是将第一个字符串中的每一个字符都与第二个字符串中对应的字符相比较。毕竟第二个字符串可能是 `String2`, 这是绝对不会与 `String1` 相等的。当然, 一旦在目的字符串中遇到了一个字符与源字符串中的对应字符不相等, 比较就可以停止了, 不需要再去比较两个字符串中剩下的字符了。

`repe` 前缀可以完成这种操作。如果字符串中的元素相等, 并且 `ECX` 中的值大于 0, 它会将一个字符串中连续的元素进行比较。我们可以使用下面的 80x86 汇编语言代码来比较上面提到的字符串:

```
cld();
mov( AdrsString1, esi );
mov( AdrsString2, edi );
mov( 7, ecx );
repe.cmpsb();
```

当 `cmpsb` 指令执行完以后, 可以使用标准(无符号)条件跳转指令来测试标志。这样就可以检查各种关系, 如相等、不相等、小于、大于等。

字符构成的串通常使用字典顺序(lexicographical order)来比较。在字典顺序中, 字符串中最低位字符的权重最大。这与标准的整数比较形成了鲜明的对比, 即一个数字的最高位的权重最大。此外, 只有两个字符串在较短字符串的长度以内相等, 字符串的长度才会对比较产生影响。例如, `Zebra` 小于 `Zebras`, 因为它是两个字符串中的较短者; 但是, `Zebra` 大于 `AAAAAAAAAAH!`, 但它比后者要短。字典顺序的比较会比较对应的元素, 直至遇到一个不匹配的字符, 或者到达较短字符串的末尾为止。如果两个对应的字符没有匹配, 那么算法就会基于这两个字符来比较字符串。如果两个字符串在较短字符串的长度之内都是匹配的, 我们就必须比较它们的长度。两个字符串相等的充要条件是它们的长度相等, 并且两个字符串中每一对对应的字符都相等。字典顺序就是

我们所熟知的标准的字母表顺序。

对于字符构成的串来说，要以下面的方式使用 `cmps` 指令：

- 方向标志必须在比较字符串之前被清零。
- 使用 `cmpsb` 指令逐字节地比较字符串。即使字符串包含偶数个字符，也不能使用 `cmpsw` 或者 `cmpsd` 指令。它们不以字典顺序比较字符串。
- 必须将较短字符串的长度载入 `ECX` 寄存器。
- 使用 `repe` 前缀。
- `ESI` 和 `EDI` 寄存器必须指向想要比较的两个字符串的第一个字符。

在 `cmps` 指令执行之后，如果两个字符串相等，就必须比较它们的长度来完成整个比较过程。

下面的代码对两个字符构成的串进行比较：

```
mov( AdrsStr1, esi );
mov( AdrsStr2, edi );
mov( LengthSrc, ecx );
if( ecx > LengthDest )then    // Put the length of the
                               // shorter string in ecx.
    mov( LengthDest, ecx );
endif;
repe.cmpsb();
if( @z )then                  // If equal to the length of the
                               // shorter string, cmp lengths.
    mov( LengthSrc, ecx );
    cmp( ecx, LengthDest );
endif;
```

如果使用字节来存放字符串的长度，就必须适当地调整这段代码(也就是说，使用一条 `movzx` 指令将长度载入 `ECX`)。当然，HLA 字符串使用一个双字来存放当前的长度值，所以在使用 HLA 字符串的时候，这并不是问题。

也可以使用 `cmps` 指令对多字的整数值(也就是扩展精度的整数值)进行比较。因为对于字符串比较来说，需要大量的设置工作，所以对长度小于 6 或 8 个双字的整数来说，这种比较操作是不现实的。但是对于更大的整数值来说，这就是一种非常好的比较方法。与字符串不同，我们不能使用字典顺序来比较整数串，而必须以从最高位字节、字或者双字到最低位字节、字或者双字的顺序来比较数值。所以，要想比较两个 32 字节(256 位)的整数值，在 80x86 上就要使用下面的代码：

```
std();
lea( esi, SourceInteger[28] );
lea( edi, DestInteger[28] );
mov( 8, ecx );
rep.cmpsd();
```

这段代码从最高位双字到最低位双字对整数进行比较。当这两个值不相等或者 `ECX` 减到 0 的时候(表示两个值是相等的)，`cmpsd` 指令就结束了。同样，标志也给出了比较的结果。

只要元素不匹配，`repne` 前缀就会让 `cmps` 指令去比较后续的元素。在这条指令执行完以后，80x86 标志就没有什么用处了。`ECX` 寄存器或者为 0(这种情况就是两个字符串完全不相等)，或者

包含到发现匹配之前两个字符串中被比较的元素个数。虽然这种形式的 `cmps` 指令对字符串比较来说不是很有用,但它对于在两个字节、字或者双字的数组中定位第一对匹配项的操作来说,却是很有用的。不过一般来说,对 `cmps` 指令很少使用 `repne` 前缀。

关于 `cmps` 指令的使用,最后还有一件事要记住: `ECX` 寄存器中的值决定要处理元素的个数,而不是字节的个数。因此,在使用 `cmpsw` 时, `ECX` 指定要比较的字的个数。同样,对于 `cmpsd` 来说, `ECX` 包含要处理的双字的个数。

11.1.6 scas 指令

`cmps` 指令将两个字符串进行比较,但不能用它在某一字符串中搜索特定的元素。例如,不能使用 `cmps` 在字符串中快速扫描出一个 0。不过,可以使用 `scas`(扫描字符串)来完成这个任务。

和 `movs` 与 `cmps` 指令不同, `scas` 指令只需要一个目的字符串(由 `EDI` 所指),而不是一个源字符串和一个目的字符串。源操作数是 `AL`(`scasb`)、`AX`(`scasw`)或者 `EAX`(`scasd`)寄存器中的值。`scas` 指令将累加器(`AL`、`AX` 或者 `EAX`)中的值与 `EDI` 所指向的值相比较,然后将 `EDI` 加(或减)1、2 或 4。CPU 将根据比较的结果设置标志。虽然这条指令只是偶尔才会用到,但 `scas` 指令在使用了 `repe` 和 `repne` 前缀以后会变得更有用。

有了 `repe` 前缀(相等时重复), `scas` 就可以在字符串中搜索某一个元素,而这个元素与累加器中的值是不匹配的。在使用 `repne` 前缀(不相等时重复)时, `scas` 指令会扫描字符串,搜索第一个与累加器中的值相等的字符串元素。

您可能会疑惑:“为什么这些前缀所做的事情正好与它们应该做的事情相反?”在前面的段落中,还没有对 `scas` 指令的操作形成一个恰当的术语。在对 `scas` 使用 `repe` 前缀的时候,如果累加器中的值与字符串操作数相等,80x86 会扫描整个字符串。这等同于在字符串中搜索第一个与累加器中的值不匹配的值。带有 `repne` 前缀的 `scas` 指令在累加器中的值与字符串操作数不相等时扫描整个字符串。当然,这种形式在字符串中搜索第一个与累加寄存器中的值匹配的值。`scas` 指令采取下面的形式:

```
scasb()
scasw()
scasd()

repe.scasb()
repe.scasw()
repe.scasd()

repne.scasb()
repne.scasw()
repne.scasd()
```

像 `cmps` 和 `movs` 指令一样,在使用了表示重复的前缀之后, `ECX` 寄存器中的值指定了要处理的元素个数,而不是字节个数。

11.1.7 stos 指令

`stos` 指令将累加器中的值存储到由 `EDI` 指定的位置上。存储了这个值以后, CPU 会根据方向标志的状态将 `EDI` 加 1 或者减 1。虽然 `stos` 指令有很多用途,但它最主要的用途是将数组和字符

串初始化为常量值。例如，如果有一个共 256 字节的数组，要想将它清零，就要使用下面的代码：

```
cld();
lea( edi, DestArray );
mov( 64, ecx );           // 64 double words = 256 bytes.
xor( eax, eax );          // Zero out eax.
rep.stosd();
```

这段代码写入了 64 个双字，而不是 256 个字节，因为单独一条 stosd 指令要比 4 条 stosb 指令的操作快一些。

stos 指令采用 6 种形式，它们是：

```
stosb();
stosw();
stosd();

rep.stosb();
rep.stosw();
rep.stosd();
```

stosb 指令将 AL 寄存器中的值存储到指定的存储单元中，stosw 指令将 AX 寄存器中的值存储到指定的存储单元中，而 stosd 指令将 EAX 寄存器中的值存储到指定的存储单元中。

要记住，stos 指令只对那些将字节、字或者双字数组初始化为常量值的操作有用。如果需要初始化一个数组，其元素具有不同的值，就不能使用 stos 指令。

11.1.8 lods 指令

lods 指令在字符串指令中是独一无二的。通常不会对这个指令使用重复前缀。lods 指令将 ESI 指向的字节、字或者双字复制到 AL、AX 或者 EAX 寄存器中，然后再将 ESI 寄存器加减 1、2 或 4。使用重复前缀来重复执行这个指令是没有用的，因为每次 lods 指令执行的时候，累加寄存器中的值都会被覆盖。在重复操作结束以后，累加器中就会包含最后从存储器中读出的数值。

但是，可以使用 lods 指令从存储器中取字节(lodsb)、字(lodsw)或者双字(lodsd)，然后作进一步的处理。通过使用 lods 指令和 stos 指令，就可以合成功能强大的字符串操作。

就像 stos 指令一样，lods 指令也采用 6 种形式：

```
lodsb();
lodsw();
lodsd();

rep.lodsb();
rep.lodsw();
rep.lodsd();
```

正如前面所提到的，您很少会对这些指令使用 rep 前缀³。80x86 会根据方向标志以及使用的是 lodsb、lodsw 还是 lodsd 指令，而将 ESI 加减 1、2 或 4。

³它们出现在这里只是因为这是允许的而已。它们并不是很有用，但是允许使用它们。对于这种形式的指令，唯一的用途可能就是去“接触”高速缓存中的项，以便它们可以被预加载到高速缓存当中。但是，还有更好的方法来实现这一目标。

11.1.9 通过 lods 和 stos 构建复杂的字符串函数

80x86 只支持 5 种不同的字符串指令: `movs`、`cmps`、`scas`、`lods` 和 `stos`⁴, 这些可能并不是您想要使用的全部字符串操作。但是, 可以使用 `lods` 和 `stos` 指令简单地产生任何您喜欢的字符串操作。例如, 假设要将一个字符串中的大写字符全部转换成小写, 那么就可以使用下面的代码:

```
mov( StringAddress, esi ); // Load string address into esi.
mov( esi, edi );           // Also point edi here.
mov( (type str.strRec [esi]).length, ecx );

repeat
    lodsb();                // Get the next character in the string.
    if( al in 'A'..'Z' ) then
        or( $20, al );      // Convert uppercase to lowercase.
    endif;
    stosb();                // Store converted char into string.
    dec( ecx );
until( @z );               // Zero flag is set when ecx is 0.
```

因为 `lods` 和 `stos` 指令使用累加器作为中介, 所以可以使用任何累加器操作来快速地操作字符串元素。

11.2 80x86 字符串指令的性能

在早期的 80x86 处理器中, 字符串指令提供了最高效的方式来操作字符串和块数据。但是, 这些指令并不是 Intel 的 RISC 核心指令集的一部分, 因此, 它们比使用不连续的指令完成同样的操作要慢。Intel 已经优化了后来的处理器上的 `movs` 指令, 使它能够尽可能快地操作。但是其他字符串指令的操作还是相当的慢。和往常一样, 应该使用不同的算法(用字符串指令的或者不用字符串指令的)来实现对性能要求严格的算法, 并将它们的性能进行比较, 以决定选用哪种解决方案。

要记住, 字符串指令的运行速度相对于其他指令来说并不相同, 这取决于所使用的处理器。因此, 最好在那些想要代码在上面运行的处理器上进行试验。注意, 在大多数处理器上, `movs` 指令要比对应的不连续指令快。Intel 已经作了很大努力来保持 `movs` 总是被优化过的, 因为有太多对性能要求严格的代码都用到了它。

虽然字符串指令可能比不连续的指令慢, 但毫无疑问, 它一般要比实现同样结果的不连续指令更加紧致。

11.3 更多信息

HLA 标准库包含很多字符串和模式匹配函数, 您会发现它们是很有用的。所有这些函数的源

⁴ 不算 `ins` 和 `outs`, 这里我们将它们忽略。

代码都可以从 <http://www.artofasm.com/> 或 <http://webster.cs.ucr.edu/> 上获得；如果想看一些关于字符串指令实际操作的示例，就应该查看其中的代码。还要注意，有一些 HLA 标准库例程使用不连续的指令来实现某些高性能的算法。可以把那些例程看做这种代码的例子。本书的 16 位版本(也可以从 <http://www.artofasm.com/> 或 <http://webster.cs.ucr.edu/> 上获得)讨论了几种使用 80x86 字符串指令的字符串函数的实现，请查看该版本的内容以获取其他示例(由于字符串指令的性能问题，那些示例在这里并没有出现)。最后，关于一般的字符串函数信息，请参考 HLA 标准库参考手册。它解释了 HLA 标准库中字符串和模式匹配函数的操作过程。

第 12 章

类 与 对 象



许多现代高级编程语言都支持类和对象的概念。C++(C 的面向对象版本)、Java 和 Delphi (Pascal 的面向对象版本)就是一些很好的例子。当然，因为这些高级语言的编译器会将它们的源代码翻译成低级的机器代码，所以在机器代码中存在一些机制来实现类和对象是毫无疑问的。

虽然可以用机器代码实现类和对象，但是大多数汇编器并没有为编写面向对象的汇编语言程序提供很好的支持。当然，HLA 并没有这样的缺点，因为它为编写面向对象的汇编语言程序提供了很好的支持。在本章中，我们将讨论支持面向对象编程(object-oriented programming, OOP)的通用原则以及 HLA 是怎样支持 OOP 的。

12.1 通用原则

在讨论支持 OOP 的机制之前，应该先回头讨论一下运用 OOP 的好处(特别是在汇编语言程序中)。大多数讲述 OOP 优点的教材都会提到诸如“代码重用”、“抽象数据类型”、“提高开发效率”之类的术语。虽然所有这些特征都很好，而且对编程来说它们都是很好的属性，但是一名优秀的软件工程师会对在一个以“提高开发效率”为重要目标的环境中使用汇编语言产生疑问。毕竟，使用高级语言(即使是非面向对象的高级语言)可能会比通过在汇编语言中使用对象获得更高的效率。如果面向对象编程的特点并不能应用到汇编语言编程中，那么为什么还要在汇编语言编程中使用 OOP 呢？本节将会告诉您原因。

首先，应该意识到使用汇编语言并没有否定前面提到的 OOP 的好处。汇编语言中的 OOP 确实提高了代码的重用性；它也为实现抽象数据类型提供了一种好方法，而且也能提高汇编程序的开发效率。换句话说，如果决心使用汇编语言，那么用 OOP 是有好处的。

为了理解 OOP 的一个主要的好处,首先考虑全局变量的概念。大多数的编程教材强烈建议:不要在程序中使用全局变量(就像本书中所做的这样)。通过全局变量建立的过程间的通信十分危险,因为在一个大型程序中,很难跟踪在哪些地方修改了全局对象。更糟糕的是,在对程序作改进的时候,很容易因为别的原因而不是原定意图重用了这个全局变量;这就有可能把缺陷引入系统中。

虽然全局变量具有显而易见的问题,但是全局对象的语义(被扩展的生存周期和从不同过程中访问的能力)在多种情况下都是必需的。对象通过让程序员决定一个对象的生存周期¹,并允许从不同的过程中访问数据域解决了这个问题。与简单的全局变量相比,对象有几个优点,所以它可以控制对它们的数据域的访问(使得过程偶然访问数据变得困难);而且可以创建一个对象的多个实例,从而允许程序的各个段分别使用它们各自唯一的全局对象来避免这些段之间相互干扰。

当然,对象还有很多其他的重要属性。关于对象和 OOP 优点的内容非常多;单单这一章是不可能完全说清楚的。本章剩下的部分将会着眼于在 HLA/汇编程序中使用对象来展示它的优点。然而,如果您还不太熟悉 OOP,或者希望获得更多关于面向对象编程方面的知识,那么应该就这个问题参考一些其他的教材。

类和对象的一个重要用途就是构造抽象数据类型(abstract data type, ADT)。抽象数据类型是数据对象以及操作数据的函数(后面我们将称之为方法)的集合。在一个纯粹的抽象数据类型中,ADT 方法是唯一可以访问 ADT 数据域的代码;外部代码只能通过使用函数调用获得或设置数据域的值来访问数据(这就是 ADT 的访问器方法)。在现实生活中,因为效率的原因,大多数支持 ADT 的语言都至少会允许外部代码受限制地访问一个 ADT 的数据域。

汇编语言不是那种大多数人会将它和 ADT 联系在一起的语言。不过,HLA 提供了几种特性来允许创建基本的 ADT。虽然一些人可能会认为 HLA 的功能并没有像诸如 C++ 或 Java 这类语言那么完善,但请记住,这些差异的存在是因为 HLA 是汇编语言。

真正的 ADT 应该支持信息隐藏(information hiding)。这就意味着 ADT 不会允许 ADT 的用户访问内部数据结构以及使用这些内部数据结构的例程。从本质上说,信息隐藏限制了对 ADT 的访问,使得只能通过 ADT 访问器方法去访问。当然,汇编语言所规定的限制非常少。如果一定要直接访问一个对象,那么 HLA 很难去阻止您这么做。然而,HLA 的某些机制会提供一定形式的信息隐藏。只要稍微注意一些,我们就能够体会到程序中信息隐藏所带来的益处。

HLA 提供的支持信息隐藏的主要机制有分离编译、可链接模块和 `#include/#includeonce` 伪指令。为实现信息隐藏,一个抽象数据类型的定义由两个部分组成:接口部分和实现部分。

接口部分包含那些必须对应用程序可见的定义。通常来说,它不应该包含任何可能导致应用程序违反信息隐藏原则的信息,但是汇编语言的特性使得这一点几乎不可能。不过,我们应该尽量只在接口部分提供必要的信息。

实现部分包含代码、数据结构等真正用来实现 ADT 的内容。虽然实现部分中的一些方法和数据类型可以是公共的(取决于是是否在接口部分出现),但是许多子例程、数据项等对于实现代码都是私有的。实现部分是隐藏应用程序中细节的地方。

如果以后想修改抽象数据类型,那么只须改变接口部分和实现部分。除非删除了原应用程序使用的可见对象,否则无须更改应用程序。

¹ 生存周期就是系统为某个对象分配存储器的时间。

虽然可以直接将接口和实现部分放在应用程序中,但是这不利于信息隐藏或可维护性,尤其是在几个不同的应用程序中需要同时使用这些代码时。最好的方法是将实现部分放在一个包含文件中,任何感兴趣的应用程序都可以通过 HLA `#include` 伪指令读取这个包含文件,还要将实现部分放在一个可以和应用程序连接的模块中。

包含文件包括 `external` 伪指令、必要的宏和一些想要设为公有的定义。通常,除了一些宏会用到 80x86 代码外,包含文件中没有 80x86 代码。当应用程序要使用 ADT 时,需要包含这个文件。

包含实现部分的独立汇编文件包括所有的过程、函数、数据对象等,以真正实现 ADT。那些想要公开的名称应该出现在接口包含文件中,并具有 `external` 属性。在实现文件中也包含接口包含文件,这样就不必维护两组 `external` 伪指令。

在使用过程进行数据访问时出现的一个问题是,许多访问器方法都非常小(例如仅仅有一个 `mov` 指令),而为这些细小操作所进行的调用和返回指令的开销非常大。例如,假设有一个 ADT,其数据对象为一个结构。您不想让域名对应用程序可见,而且也实在不想允许应用程序直接访问数据结构的域(因为将来数据结构可能会改变)。解决这个问题的标准方法是提供一个返回目标域值的 `GetField` 方法。然而,就像上面所指出的,这可能会非常慢。简单访问方法的一个替代方案是用一个宏来生成访问目标域的代码。虽然直接访问数据对象的代码出现在应用程序中(通过宏扩展),但是如果改变接口部分的宏,那么应用程序中的代码会通过简单的重新编译自动更新。

虽然很可能只需要运用单独编译(也许还有记录)就可以创建 ADT,但是 HLA 却提供了一个更好的解决方案:类。下面的内容介绍 HLA 对类和对象的支持以及如何使用这些类和对象来创建 ADT。

12.2 HLA 中的类

从本质上说,类只不过是一个允许定义非数据域(例如过程、常量和宏)的记录声明。在类的定义中包含其他的对象可以显著扩展类的功能。例如,使用类很容易定义 ADT,因为类可以包含数据和对这些数据进行操作的方法(过程)。

在 HLA 中创建抽象数据类型的基本方法是声明一个类数据类型。HLA 中的类总是出现在 `type` 段,并使用下面的语法:

```
classname : class

    << Class declaration section >>

endclass;
```

类的声明部分可以定义 `const`、`val`、`var`、`storage`、`readonly`、`static` 和 `proc` 变量,这和过程的局部声明部分非常相似。但类还可以定义宏、过程、迭代器²和方法原型(只有在类中进行方法声明才是合法的)。显然,这里少了类型定义部分,因为在类中不允许定义新类型。

² 本书不讨论迭代器。请参考 HLA 参考手册,以了解这种函数类型的详细信息。

方法是一种特殊的过程，只出现在类中，我们稍后将会看到过程和方法之间的差别；现在您可以暂时把它们当成是一样的。除了类的初始化和指针的运用不同以外，它们的语义是一样的³。通常情况下，如果不知道在类中应该使用过程还是方法，那么最安全的方式是使用方法。

不要将过程/迭代器/方法的代码放在类中，只需要在类中提供这些例程的原型。例程的原型由保留字(procedure、iterator 或 method)、例程名、参数和几个可选的过程属性(@use、@returns 和 external)组成。而实际的例程定义(也就是说例程体和它所需要的局部声明)出现在类的外部。

下面的示例展示了一个出现在 type 部分的类声明：

```
TYPE
    TypicalClass: class
        Const
            TCconst := 5;

        Val
            TCval := 6;

        var
            TCvar: uns32;    // Private field used only by TCproc.

        static
            TCstatic: int32;

        procedure TCproc( u:uns32 ); @returns( "eax" );
        iterator TCiter( i:int32 ); external;

        method TCmethod (c:char);

    endclass
```

如上所示，类和 HLA 中的记录非常相似。确实，可以把记录看成是一个只允许 var 声明的类。HLA 用一种和记录非常相似的方式实现了类，所以它为连续的数据域分配连续的存储区域。事实上，除了一个微小的差异以外，记录声明和只有一个 var 声明部分的类声明之间几乎没有任何差别。后面我们将看到 HLA 是如何实现类的，但是现在可以认为 HLA 就像实现记录一样地实现类，而且这离实际情况也差不了很多。

可以像访问记录的字段一样访问 TCvar 和 TCstatic 域(在上面的类中)。对 const 的访问方式与 val 域相同。如果一个 TypicalClass 类型的变量名为 obj，那么可以采用下面的方式访问 obj 的域：

```
mov( obj.TCconst, eax );
mov( obj.TCval, ebx );
add( obj.TCvar, eax );
add( obj.TCstatic, ebx );
obj.TCproc( 20 ); // Calls the TCproc procedure in TypicalClass.
etc.
```

如果一个应用程序包含了上面的类声明，那么就可以使用 TypicalClass 类型构造变量并通过上面提到的方法来执行操作。遗憾的是，应用程序也可以毫无阻碍地访问 ADT 数据类型的域。

³ 然而，要注意的是过程和方法之间的差别造成了面向对象编程范式中的所有差别。因此，在 HLA 的类定义中包含了方法。

例如,如果一个程序构造了一个 TypicalClass 类型的变量 MyClass,那么即使域对于实现部分而言是私有的,这个程序也很容易执行诸如 `mov(MyClass.TCvar,eax);` 这样的指令。而且,如果允许应用程序声明一个 TypicalClass 类型的变量,则域名必须可见。虽然在 HLA 的类定义中,我们可以使用一些小技巧来帮助隐藏私有域,但是最好的方法还是完全注释私有域,然后在访问该类的域时实行一些限制。需要明确的是,因为 HLA 允许直接访问数据域,所以这就意味着您用 HLA 的类构造的 ADT 不可能是“纯粹”的 ADT。然而,根据一些小的原则,可以选择不在类的方法、过程和迭代器之外访问这些域而模拟纯粹的 ADT。

在类中出现的原型实际上就是向前引用声明。和通常的向前引用声明一样,在类中定义的所有过程、迭代器和方法都必须在后面的代码中有真正的实现。可以选择把 `external` 选项附在类中过程、迭代器和方法声明的结尾处,用来告诉 HLA 真正的代码将出现在一个单独的模块中。作为一条通用规则,类声明一般出现在头文件中,用来表示 ADT 的接口部分。过程、迭代器和方法体均出现在实现部分,通常是一个单独的源文件,可以单独编译,并且能连接到使用该类的模块上。

下面是一个类的过程的实现示例:

```
procedure TypicalClass.TCproc( u:uns32 ); @nodisplay;
    << Local declarations for this procedure >>
begin TCproc;

    << Code to implement whatever this procedure does >>

end TCProc;
```

在标准过程声明和类过程声明之间存在着一些差别。首先也是最明显的差别是过程名包含了类名(例如 TypicalClass.TCproc)。这就将类的过程定义与仅仅是名称碰巧为 TCproc 的常规过程区别开来。然而,需要注意的是在过程的 `begin` 和 `end` 子句中,并不一定要在过程名之前重复类名(这与在 HLA 命名空间中定义的过程很相似)。

类过程和非类过程的第二个差别不太明显。一些过程属性(`@use`、`external`、`@returns`、`@cdecl`、`@pascal` 和 `@stdcall`)只在类的原型声明中才是合法的,而另外一些属性(`@noframe`、`@nodisplay`、`@noalignstack` 和 `@align`)则只在过程定义而不是类中才是合法的。幸运的是,如果您把选项贴在了错误的地方,那么 HLA 会提供错误帮助信息,所以不一定要记住这些规则。

如果一个类例程的原型没有 `external` 选项,那么包含类声明的编译单元(也就是程序或单元)必须还包含这个例程的定义,否则 HLA 将在编译的结尾产生一个错误。一般情况下,较小的局部类(也就是将类声明和例程的定义放在同一个编译单元中)会把类的过程、迭代器和方法的定义放在紧随类声明之后的源文件中。而在更大的系统中(也就是单独编译一个类的例程时),一般将类的声明单独放在一个头文件中,而将所有的过程、迭代器和方法的定义放在一个独立的 HLA 单元中,并且单独编译它们。

12.3 对象

记住,类的定义只是一种类型。因此,在声明类类型时,还没有创建可以操作它的域的变量。

对象是类的实例；也就是说，对象是一个该类类型的变量。可以像声明其他变量一样在 `var`、`static` 或 `storage` 段中声明对象(也就是类变量)⁴。下面是两个对象声明的示例：

```
var
    T1: TypicalClass;
    T2: TypicalClass;
```

对于给定的类对象，HLA 为类声明中每一个出现在 `var` 部分中的变量分配存储单元。如果有两个类型为 `TypicalClass` 的对象 `T1` 和 `T2`，那么 `T1.TCvar` 和 `T2.TCvar` 一样，都是唯一的。这是直觉上的结果(与记录声明类似)；大多数在类中定义的数据域会出现在类的 `var` 声明部分。

静态数据对象(例如，那些在类声明的 `static` 或 `storage` 部分声明的对象)在该类的对象中不是唯一的；也就是说，HLA 只分配一个静态变量，而类的所有变量共享该静态变量。例如，考虑下面(部分)类的声明和对象声明：

```
type
    sc: class
        var
            i:int32;
        static
            s:int32;
            .
            .
            .
    endclass;

var
    s1:sc;
    s2:sc;
```

在这个示例中，`s1.i` 和 `s2.i` 是不同的变量。然而，`s1.s` 和 `s2.s` 互为别名。因此，像 `mov(5,s1.s)`；这个指令也会将 5 存进 `s2.s` 中。通常用静态类变量保存有关整个类的信息，而用类的 `var` 对象保存有关特定对象的信息。因为跟踪类的信息是相对较少见的，所以可以在 `var` 部分声明大多数的数据域。

也可以创建类的动态实例并通过指针指向那些动态对象。事实上，这可能是对象存储和访问最常见的方式。下面的代码展示了如何为对象构造指针以及如何为对象动态分配存储单元：

```
var
    pSC: pointer to sc;
    .
    .
    .

mem.alloc( @size(sc) );
mov( eax, pSC );
```

⁴ 从技术上说，也可以在 `readonly` 部分声明对象，但是因为 HLA 不允许定义类常量，所以在 `readonly` 部分中声明类对象没有多大的用处。

```

mov( pSC, ebx );
mov( (type sc [ebx]).i, eax );

```

注意，使用强制类型转换将 EBX 中的指针转换为 sc 类型。

12.4 继承

继承是面向对象编程中最根本的思想之一。其基本思想是一个类从某个类中继承或者说复制所有的域，并且还可能在新的数据类型中扩展域的数目。例如，假设构造了一个描述平面(二维)空间中一个点的数据类型 `point`。描述该点的类可能会像下面这样：

```

type
  point: class

    var
      x:int32;
      y:int32;

    method distance;

  endclass;

```

假设想在 3D 空间而不是 2D 空间中构造一个点，就可以简单地构造一个如下所示的数据类型：

```

type
  point3D: class inherits( point )

    var
      z:int32;

  endclass;

```

类声明中的 `inherits` 选项告诉 HLA 将 `point` 的所有域插入类的开头部分。在这种情况下，`point3D` 继承了 `point` 的域。HLA 始终将所继承的域放在类对象的开头。这么做的原因将在稍后进行阐述。如果有一个名为 `P3` 的 `point3D` 实例，那么下面的 80x86 指令是完全合法的：

```

mov( P3.x, eax );
add( P3.y, eax );
mov( eax, P3.z );
P3.distance();

```

注意，在这个示例中，`P3.distance` 方法调用了 `point.distance` 方法。不必为 `point3D` 类编写一个单独的 `distance` 方法，除非您真的想这么做(参见 12.5 节)。就像继承 `x` 和 `y` 域一样，`point3D` 对象也继承了 `point` 的方法。

12.5 重写

重写(overriding)是用一个更适合新类的方法替代继承类中现有方法的过程。在前一节出现的 point 和 point3D 示例中, distance 方法计算从原点到一个指定点的距离。对于二维平面中的一个点, 可以用下面的公式计算距离:

$$d = \sqrt{x^2 + y^2}$$

然而, 在 3D 空间中一个点的距离由下面的公式给定:

$$d = \sqrt{x^2 + y^2 + z^2}$$

显然, 如果为一个 3D 对象调用了 point 的 distance 函数, 就会得到一个错误的答案。但是在 12.4 节中, P3 对象调用了从 point 类中继承的 distance 函数, 所以会产生一个错误的结果。

在这种情况下, point3D 数据类型必须用一个能正确求值的方法重写 distance 方法, 而不能通过增加 distance 方法原型简单地重定义 point3D 类:

```
type
  point3D: class inherits( point )
    var
      z:int32;
    method distance; // This doesn't work!
  endclass;
```

上面的 distance 方法声明所存在的问题是 point3D 已经有了一个 distance 方法——它从 point 类中继承的那个。HLA 将会报错, 因为在一个类中有两个同名的方法。

为解决这个问题, 我们需要一些机制, 重写 point.distance 的声明, 并用一个 point3D.distance 的声明来取代它。这需要在方法声明前面使用 override 关键字:

```
type
  point3D: class inherits( point )
    var
      z:int32;
    override method distance; // This will work!
  endclass;
```

override 前缀告诉 HLA 忽略 point3D 从 point 类中继承了一个名为 distance 的方法的事实。现在, 任何通过 point3D 对象对 distance 方法的调用都将调用 point3D.distance 方法, 而不是 point.distance 方法。当然, 一旦用 override 前缀重写了一个方法, 就必须在代码的实现部分提供这个方法, 例如:


```

method point3D.distance; @nodisplay;

    << local declarations for the distance function >>

begin distance;

    << Code to implement the distance function >>

end distance;

```

12.6 虚拟方法与静态过程

本章前面曾建议将类的方法和类的过程不加区别。事实上，这两者之间存在着很大的差别(毕竟，如果方法和过程一样，那为什么还需要方法呢)。我们将会看到，如果想要开发面向对象程序，那么方法和过程之间的差别是至关重要的。方法提供了支持多态性必需的第三种特性：虚拟过程调用⁵。虚拟过程调用仅仅是间接过程调用(使用一个与对象相关联的指针)的一个别致一点的名称。虚拟函数的关键好处是当用指针指向通用对象时，系统会自动调用正确的方法。

考虑下面使用 `point` 类的声明：

```

var
    P2: point;
    P:  pointer to point;

```

在给定上面的声明的情况下，下面的汇编语句全部是合法的：

```

mov( P2.x, eax );
mov( P2.y, ecx );
P2.distance();    // Calls point3D.distance.
lea( ebx, P2 );   // Store address of P2 into P.
mov( ebx, P );
P.distance();     // Calls point.distance.

```

注意，HLA 是通过一个指向对象的指针来调用方法，而不是直接通过对象变量来调用方法。在 HLA 中，这是对象的一个至关重要的特征，也是实现虚拟方法调用的关键。

隐藏在多态和继承之后的神奇之处是对象的指针是通用的。通常情况下，当程序通过指针间接引用数据时，指针的值是与指针相关联的底层数据类型的某些值的地址。例如，一个指向 16 位无符号整型数据的指针不能正常访问 32 位的有符号整型值。同样，指向某些记录的指针不能正常指向其他的记录类型，也不能访问其他类型的域⁶。然而，使用指向类对象的指针，可以稍微减轻这个限制。指向对象的指针可以包含该对象的类型的地址，或者任何继承了该类型的域的对象地址。考虑下面使用 `point` 和 `point3D` 类型的声明：

⁵ “多态性”字面上的意思是“多面的”。在面向对象编程环境中，多态性意味着相同的方法名称，例如 `distance`，可以指多个不同方法中的某一个方法。

⁶ 当然，汇编语言程序员总是打破这样的规则。现在，我们假设遵守这些规则，并且只用与指针相关联的数据类型来访问数据。

```

var
    P2: point;
    P3: point3D;
    p:  pointer to point;
    .
    .
    .
    lea( ebx, P2 );
    mov( ebx, p );
    p.distance();           // Calls the point.distance method.
    .
    .
    .
    lea( ebx, P3 );
    mov( ebx, p );           // Yes, this is semantically legal.
    p.distance();           // Surprise, this calls point3D.distance.

```

因为 `p` 是一个指向 `point` 对象的指针，所以直觉上看起来好像是 `p.distance` 在调用 `point.distance` 方法。然而，方法是多态的。如果已经获得了一个指向某个对象的指针，并且调用了一个与该对象相关联的方法，那么系统将会调用和对象相关联的实际(被重写)方法，而不是那个只与指针的类类型相关联的方法。

至于重写过程，类过程与方法不一样。当通过对象指针间接调用类的过程时，系统将始终调用与底层类相关联的过程。所以，如果在前面的示例中，`distance` 是一个过程而不是一个方法，那么 `p.distance()` 调用将始终调用 `point.distance`，即使 `p` 是一个指向 `point3D` 对象的指针。12.9 节将解释为什么方法和过程是不同的。

12.7 编写类方法和过程

对于类定义中的每一个类过程和方法原型，都必须有一个相应的过程体或方法体出现在程序中(为了简洁起见，本节之后将用术语“例程(routine)”来代表过程或方法)。如果原型没有包含 `external` 选项，那么这些代码就必须和类的声明出现在同一个编译单元中。如果原型后面跟着 `external` 选项，那么代码可以和类的声明出现在同一个编译单元中，也可以出现在不同的编译单元中(只要将最后所得到的目标文件和包含类声明的代码连接在一起就可以了)。同外部(不是类的)过程一样，如果没有提供代码，那么在构建可执行文件时，连接器将会报错。为了缩短后面示例的长度，我们会在与类的声明相同的源文件中定义它们的例程。

在一个编译单元中，HLA 类例程必须始终紧随在类的声明之后。如果在单独一个单元中编译例程，那么类的声明仍然必须置于例程的实现之前(通常是通过 `#include` 文件来实现)。如果直到定义一个像 `point.distance` 那样的例程时还没有定义类，那么 HLA 就不会知道 `point` 是一个类，也因此不知道如何处理例程的定义。

考虑下面 `point2D` 类的声明：

```

type
    point2D: class

```

```

const
    UnitDistance: real32 := 1.0;

var
    x: real32;
    y: real32;

static
    LastDistance: real32;

method distance
(
    fromX: real32;
    fromY: real32
); @returns( "st0" );
procedure InitLastDistance;

endclass;

```

这个类的 `distance` 函数计算从一个对象的点到(`fromX`,`fromY`)的距离。下面的公式描述了这个计算:

$$d = \sqrt{(x - \text{fromX})^2 + (y - \text{fromY})^2}$$

初次编写的 `distance` 方法可能如下所示:

```

method point2D.distance(_fromX:real32; fromY:real32); @nodisplay;
begin distance;

    fld( x );           // Note: this doesn't work!
    fld( fromX );        // Compute (x-fromX)
    fsubp();
    fld( st0 );          // Duplicate value on TOS.
    fmulp();             // Compute square of difference.

    fld( y );           // This doesn't work either.
    fld( fromY );        // Compute (y-fromY)
    fsubp();
    fld( st0 );          // Compute the square of the difference.
    fmulp();
    faddp();
    fsqrt();

end distance;

```

对于熟悉 C++ 或 Delphi 这类面向对象编程语言的人来说, 这些代码看起来好像可以工作。然而, 就像注释中所显示的那样, 将变量 `x` 和 `y` 压入 FPU 栈的指令将不会执行; HLA 不会自动在类的例程中定义与类的数据域相关的符号。

为了学习如何在类的例程中访问类的数据域, 我们需要回过头来并讨论一些与 HLA 类相关的非常重要的实现细节。考虑下面的变量声明:

```
var
    Origin: point2D;
    PtInSpace: point2D;
```

记住, 无论何时创建两个像 `Origin` 和 `PtInSpace` 这样的对象, HLA 都会为这两个对象的 `x` 和 `y` 数据域保留存储空间。可是, 在存储器中只有 `point2D.distance` 方法的一个副本。因此, 如果要调用 `Origin.distance` 和 `PtInSpace.distance`, 系统将会为这两个方法调用相同的例程。在方法中, 我们想知道像 `fld(x)`: 这样的指令会做些什么工作。它是如何将 `x` 和 `Origin.x` 或 `PtInSpace.x` 关联起来的? 代码如何将数据域 `x` 和一个全局对象 `x` 区别开来? 在 HLA 中, 答案是它没有区分。我们不能像普通的变量那样仅通过它们的名称来区分数据域。

为了在类的例程中将 `Origin.x` 和 `PtInSpace.x` 区分开来, 无论何时调用一个类例程, HLA 都会自动将一个指针传递给一个对象的数据域。因此, 可以用这个指针间接引用该数据域。HLA 在 `ESI` 寄存器中传递这个对象的指针。这是少数几个由 HLA 产生的代码在您不知道的情况下更改 80x86 寄存器的地方之一: 任何时候调用类例程, HLA 都会将对象的地址自动加载至 `ESI` 寄存器中。显然, 不能指望 `ESI` 的值在类例程的调用过程中会被保存下来, 也不能在 `ESI` 寄存器中将参数传递给类例程(虽然指定 `@use esi;` 来让 HLA 在设置其他参数的时候使用 `ESI` 寄存器是完全合理的)。对于类的方法(而不是过程), HLA 也会将类的虚拟方法表(virtual method table)的地址加载至 `EDI` 寄存器中。然而虚拟方法表的地址不像对象地址那样值得注意, 但是一定要记住调用类方法或迭代器时, 由 HLA 产生的代码将重写 `EDI` 寄存器中的任何值。此外, 对于方法的 `@use` 操作数而言, `EDI` 是一个不错的选择, 因为 HLA 无论如何都会清除 `EDI` 中的值。

在类例程的入口处, `ESI` 包含了一个指向与类相关联的(非静态)数据域的指针。因此, 为了访问像 `x` 和 `y`(在我们的 `point2D` 示例中)之类的域, 可能要用到如下的地址表达式:

```
(type point2D [esi]).x
```

因为将 `ESI` 作为对象数据域的基址, 所以最好不要破坏类例程中的 `ESI` 值(或者, 至少在必须使用 `ESI` 时, 也应该在代码中保存 `ESI` 的值)。注意, 如果调用方法, 那么就没必要保存 `EDI`(除非因为某些原因需要访问虚拟方法表, 而这是不太可能的)。

访问类例程中的数据对象的域是很常见的操作, 所以 HLA 提供了一个简写符号将 `ESI` 强制转换为一个类对象的指针: `this`。在 HLA 的类中, 保留字 `this` 自动扩展为一个形式为 `(type classname[esi])` 的字符串, 当然 `classname` 会被替换为合适的类名称。使用 `this` 关键字, 我们就可以(正确地)像下面这样重写前面的 `distance` 方法:

```
method point2D.distance( fromX:real32; fromY:real32 ); @nodisplay;
begin distance;
    fld( this.x );
    fld( fromX );          // Compute (x-fromX).
    fsubp();
    fld( st0 );            // Duplicate value on TOS.
    fmulp();               // Compute square of difference.
    fld( this.y );
    fld( fromY );          // Compute (y-fromY).
```

```

    fsubp();
    fld( .st0 );           // Compute the square of the difference.
    fmulp();
    faddp();
    fsqrt();

end distance;

```

不要忘了调用类例程会清除 ESI 寄存器中的值。从例程的调用语法上来说,这并不明显。当从其他类的例程中调用某些类的例程时,尤其容易忘掉这一点;不要忘记如果这样做了,那么内部调用就会清除 ESI 中的值,在从该调用返回的时候 ESI 不再指向最初的对象。在这种情况下就总是要压入和弹出 ESI(或者用其他方式保存 ESI 的值),例如:

```

    fld( this.x );           // esi points at current object.
    .
    .
    .
    push( esi );             // Preserve esi across this method call.
    SomeObject.SomeMethod();
    pop( esi );
    .
    .
    .
    lea( ebx, this.x );      // esi points at original object here.

```

this 关键字提供了访问在类的 var 段声明的类变量的途径。也可以用 this 去调用与当前对象相关联的其他类的例程,例如:

```

this.distance( 5.0, 6.0 );

```

访问类的常量和静态数据域通常不会使用 this 指针。HLA 将常量和静态数据域与整个类相关联,而不是与一个具体的对象(就像类中的 static 域)相关联。为了访问这些类成员,需要使用类名代替对象名。例如,为了访问 point2D 中的 UnitDistance 常量,可以使用下面的语句:

```

fld( point2D.UnitDistance );

```

如果想要在每次计算距离的时候更新 point2D 类中的 LastDistance 域,那么可以像下面这样重写 point2D.distance 方法:

```

method point2D.distance( fromX:real32; fromY:real32 ); @nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );           // Compute (x-fromX).
    fsubp();
    fld( .st0 );            // Duplicate value on TOS.

```

```

    fmulp();           // Compute square of difference.

    fld( this.y );
    fld( fromY );      // Compute (y-fromY).
    fsubp();
    fld( st0 );        // Compute the square of the difference.
    fmulp();
    faddp();
    fsqrt();

    fst( point2D.LastDistance ); // Update shared (STATIC) field.

end distance;

```

如果想理解为什么在引用常量和静态对象时要用类名而同时却用 `this` 访问 `var` 对象，请阅读下一节。

因为类的过程也是静态对象，所以在过程调用中可以通过指定类名而不是一个对象名来调用类过程。例如，下面这两条指令都是合法的：

```

Origin.InitLastDistance();
point2D.InitLastDistance();

```

然而，在这两个类过程调用之间有一个细小的差别。上面的第一个调用在真正调用 `InitLastDistance` 过程之前，会将 `origin` 对象的地址加载到 `ESI` 中。而第二个调用是对类过程的一个直接调用，没有引用对象；因此，HLA 不知道哪个对象的地址将会加载到 `ESI` 寄存器中。在这种情况下，HLA 在调用 `InitLastDistance` 过程之前先将 `NULL(0)` 载入 `ESI` 中。因为可以用这种方式来调用类过程，所以最好始终检查类过程中的 `ESI` 的值，以确保 HLA 包含有效的对象地址。检查 `ESI` 中的值是确定使用哪种调用机制的好方法。12.9 节将会讨论构造函数和对象初始化；您将看到静态过程和直接调用过程(而不是通过使用对象)的好处。

12.8 对象实现

在 C++ 或 Delphi 这样的高级面向对象语言中，您很可能在还没有真正明白机器是如何实现对象的情况下就掌握了对象的使用。学习汇编语言编程的一个原因是完全领会底层的实现细节，这样在使用对象这样的编程结构时就能做出有根据的决定。此外，因为汇编语言允许在底层随意操作数据结构，所以清楚 HLA 如何实现对象有助于创建某些可靠的算法，而如果没有对象实现的详细知识，则不可能做到这些。因此，本节将解释底层的实现细节。为了编写面向对象的 HLA 程序，我们需要知道这些。

HLA 用一种与记录非常相似的方式实现对象。尤其是，HLA 用与记录一样的连续方式为类中所有的 `var` 对象分配存储空间。事实上，如果一个类仅仅由 `var` 数据域组成，那么该类的存储器表示几乎和相应的记录声明完全一致。考虑第 4 章中的 `student` 记录声明及其相应的类(参见图 12-1 和图 12-2)：

```
type
  student: record
    Name:      char[65];
    Major:     int16;
    SSN:       char[12];
    Midterm1:  int16;
    Midterm2:  int16;
    Final:     int16;
    Homework:  int16;
    Projects:  int16;
  endrecord;
  student2: class
    var
      Name:      char[65];
      Major:     int16;
      SSN:       char[12];
      Midterm1:  int16;
      Midterm2:  int16;
      Final:     int16;
      Homework:  int16;
      Projects:  int16;
  endclass;
```

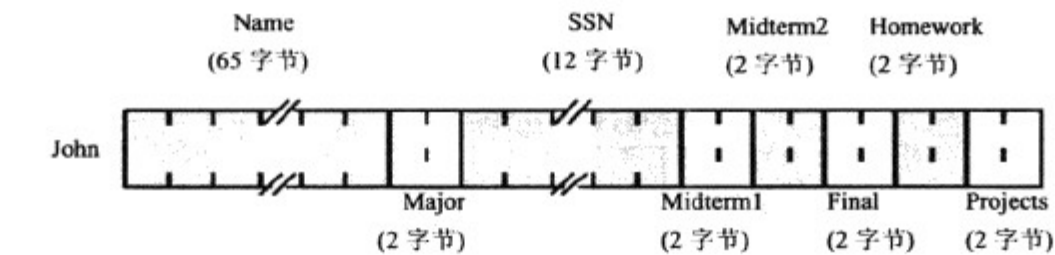


图 12-1 存储器中 student 记录的实现

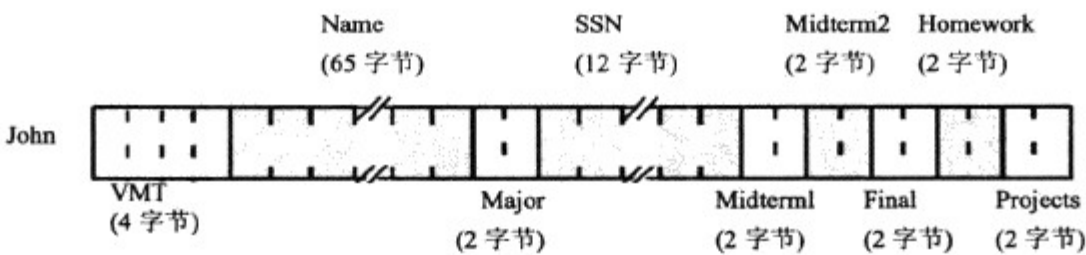


图 12-2 存储器中 student 类的实现

如果仔细查看图 12-1 和图 12-2, 就会发现类的实现和记录的实现之间唯一的差别是在类对象的开头部分包含了 VMT(virtual method table, 虚拟方法表)指针域。这个总是出现在类中的域包含了类的虚拟方法表的地址, 而虚拟方法表又包含了所有类的方法和迭代器的地址。顺便说一下, 即使一个类没有包含任何方法或迭代器, VMT 域也会出现。

就像前面小节所指出的, HLA 不会为对象中的静态对象分配存储空间。实际上, HLA 为每个静态数据域分配了一个为所有对象共享的唯一实例。作为一个示例, 考虑下面的类和对象声明:

```
type
  tHasStatic: class
```



```
var
    i:int32;
    j:int32;
    r:real32;

static
    c:char[2];
    b:byte;

endclass;

var
    hs1:tHasStatic;
    hs2:tHasStatic;
```

图 12-3 显示了在存储器中这两个对象的存储空间的分配情况。

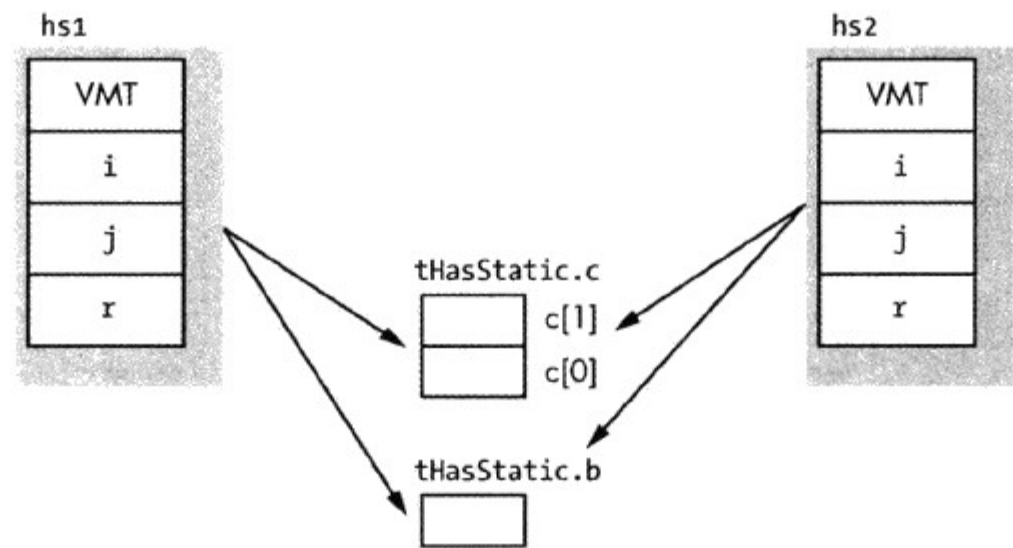


图 12-3 带静态数据域的对象分配

当然，在运行时 `const`、`val` 和 `#macro` 对象对存储器不会有任何需求，所以 HLA 不会为这些域分配任何存储器空间。同静态数据域一样，可以用类的名称或对象的名称访问 `const`、`val` 和 `#macro` 域。因此，即使 `tHasStatic` 拥有这些域类型，`tHasStatic` 对象的存储器的空间分配仍然和图 12-3 中显示的一样。

除了虚拟方法表(VMT)指针的出现外，方法和过程的出现对于一个对象的存储器分配不会有任何影响。当然，与这些例程相关联的机器指令确实会出现在存储器的某个地方。所以从某种意义上说，例程的代码与静态数据域是非常相似的，所以所有的对象共享一个例程实例。

12.8.1 虚拟方法表

当 HLA 调用类过程时，会直接用一个 `call` 指令调用这个过程，就像任何普通过程调用一样。方法则与此不同。系统中的每个对象都有一个指向虚拟方法表的指针，虚拟方法表是一个指向类中所有方法和迭代器的指针数组(如图 12-4 所示)。

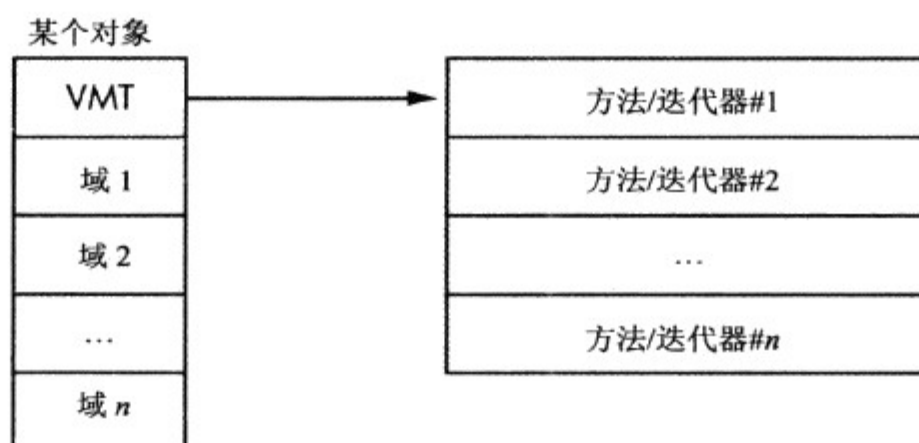


图 12-4 虚拟方法表的组织结构

在类中声明的每一个迭代器或方法在虚拟方法表中都有一个相应的表项。该双字项包含了迭代器或方法中第一个指令的地址。调用类方法或迭代器比调用类过程要稍微复杂一些(它需要一条额外的指令并要使用 EDI 寄存器)。下面是方法的一个典型的调用序列:

```

mov( ObjectAdrs, ESI );           // All class routines do this.
mov( [esi ], edi );               // Get the address of the VMT into edi
call( (type dword [edi+n]));      // "n" is the offset of the method's
                                   // entry in the VMT.

```

对于一个给定的类，在存储器中只有 VMT 的一个副本。因为这是一个静态对象，所以一个给定类类型的所有对象都共享同一个 VMT。这是合理的，因为同一个类类型的所有对象都有完全一样的方法和迭代器(参见图 12-5)。

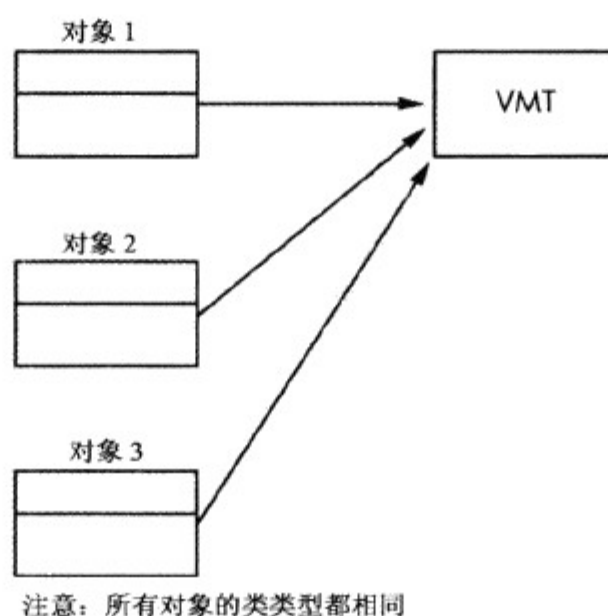


图 12-5 同一个类类型的所有对象共享同一个 VMT

虽然 HLA 就像对待类中的方法和迭代器一样构造了 VMT 记录结构，但是它不会自动产生虚拟方法表。必须在程序中显式声明这个表。这就需要在程序的 static 或 readonly 声明段中包含如下语句，例如：

```

readonly
    VMT( classname );

```

因为在程序执行过程中, 虚拟方法表中的地址应该永不改变, 所以 `readonly` 段可能是用来声明 VMT 的最好选择。通常情况下, 改变 VMT 中的指针不是一个好主意。所以, 最好不要将 VMT 放在 `static` 段中。

上面的声明定义了变量 `classname._VMT_`。在本章 12.9 节中, 我们将看到在初始化对象的变量时需要使用该名称。类的声明自动将 `classname._VMT_` 符号定义为一个外部静态变量。上面的声明恰好提供了这个外部符号的实际定义。

VMT 的声明使用了陌生的语法, 因为我们不会用这个声明实际去声明一个新符号; 只是要简单地给符号提供数据, 这个符号是在前面通过定义类而隐式声明了的。也就是说, 类的声明定义了静态表变量 `classname._VMT_`; 此时需要 VMT 声明告诉 HLA 为这个表发送实际的数据。如果因为某些原因要用一个不是 `classname._VMT_` 的名称来引用这个表, 那么 HLA 允许用一个变量名作为前缀加在声明的上面, 例如:

```
readonly
    myVMT: VMT( classname );
```

在这个声明中, `myVMT` 是 `classname._VMT_` 的一个别名。作为一个通用的原则, 应该在程序中避免使用别名, 因为它们将降低程序的可读性并难以理解。因此, 可能并不真正需要使用这种类型的声明。

像任何其他的全局静态变量一样, 对于在一段程序中的一个给定类应该只有一个 VMT 实例。最好是将 VMT 声明和类的方法、迭代器和过程代码放在同一个源文件中(假设它们都出现在一个单独的文件中)。这样, 无论何时连接一个类的例程, 都会自动连接 VMT。

12.8.2 带继承的对象表示

直到现在为止, 有关类对象实现的讨论都忽略了继承。继承通过增加一些不会在类声明中显式声明的域而影响了一个对象的存储器表示。

将被继承的域从一个基类添加到另一个类中时必须要小心。记住, 从基类中继承域的类的一个重要特性是可以用一个指向基类的指针访问从该基类中继承的域, 即使这个指针包含其他类(从基类中继承了域的类)的地址。作为一个示例, 考虑下面的类:

```
type
    tBaseClass: class
        var
            i:uns32;
            j:uns32;
            r:real32;

        method mBase;
    endclass;

    tChildClassA: class inherits( tBaseClass )
        var
            c:char;
            b:boolean;
            w:word;
```

```

        method mA;
    endclass;

    tChildClassB: class inherits( tBaseClass )
        var
            d:dword;
            c:char;
            a:byte[3];
    endclass;

```

因为 tChildClassA 和 tChildClassB 都继承了 tBaseClass 的域，所以这两个子类包含了 i、j 和 r 域以及它们自己特定的域。而且，只要有一个基类为 tBaseClass 的指针变量，用 tBaseClass 的任何子类的地址加载这个指针就是合法的；因此，用 tChildClassA 或 tChildClassB 变量的指针来加载这样一个指针就是完全合理的。例如：

```

var
    B1: tBaseClass;
    CA: tChildClassA;
    CB: tChildClassB;
    ptr: pointer to tBaseClass;
    .
    .
    .
    lea( ebx, B1 );
    mov( ebx, ptr );
    << Use ptr >>
    .
    .
    .
    lea( eax, CA );
    mov( ebx, ptr );
    << Use ptr >>
    .
    .
    .
    lea( eax, CB );
    mov( eax, ptr );
    << Use ptr >>

```

因为 ptr 指向 tBaseClass 类型的对象，所以可以合法地(从语义的角度)访问 ptr 所指向的对象的 i、j 和 r 域。但是因为任何时候，程序都可能不知道 ptr 指向的确切对象类型，所以访问 tChildClassA 或 tChildClassB 对象的 c、b、w 或 d 域将是非法的。

为了让继承正确地工作，在所有的子类中，i、j 和 r 域都必须和它们在 tBaseClass 中一样出现在相同的偏移位置。这样，即使 EBX 指向 tChildClassA 或 tChildClassB 类型的某个对象，格式为 mov((type tBaseClass [ebx]).i,eax)的指令都将正确地访问 i 域。图 12-6 展示了子类和基类的布局图。

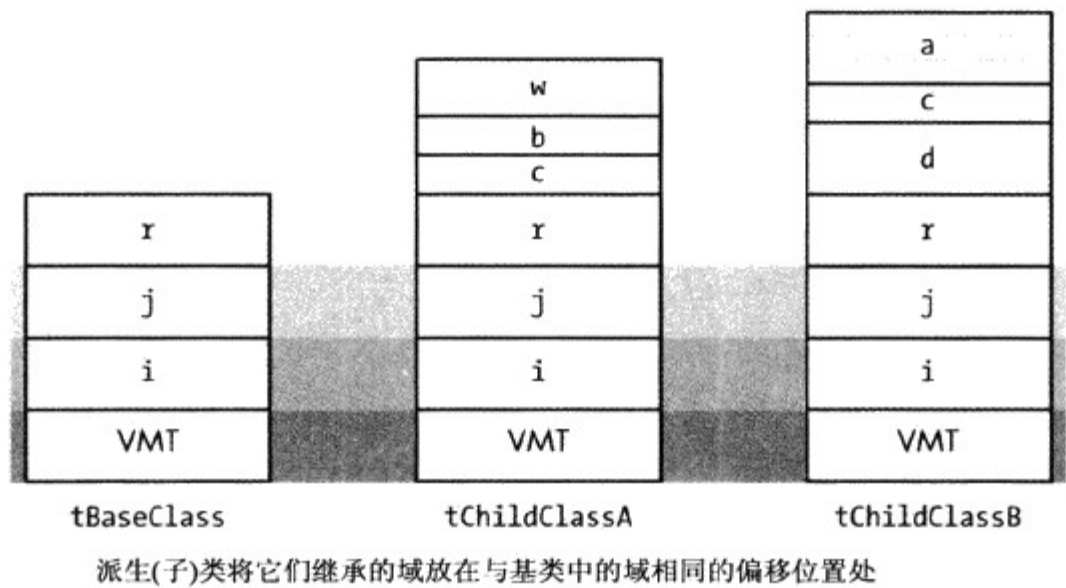


图 12-6 基类和子类的对象在存储器中的布局图

注意，两个子类中的新域相互之间没有关系，即使它们拥有相同的名称(例如，两个子类中的 c 域并没有放在同一偏移量上)。虽然这两个子类共享它们从公共基类中继承的域，但是它们增加的任何新域都是唯一的和单独的。如果域不是从同一个基类中继承的，那么不同类中的两个域只有在偶然的情况下才会使用相同的偏移量。

所有的类(即使那些互不相干的类)都会将指向虚拟方法表的指针放在对象中偏移量为 0 的地方。在程序中有一个单独的 VMT 和每一个类相关联；甚至那些从基类中继承了域的类也会有一个通常和基类的 VMT 不同的 VMT。图 12-7 展示了类型为 tBaseClass、tChildClassA 和 tChildClassB 的对象如何指向它们特定的 VMT。

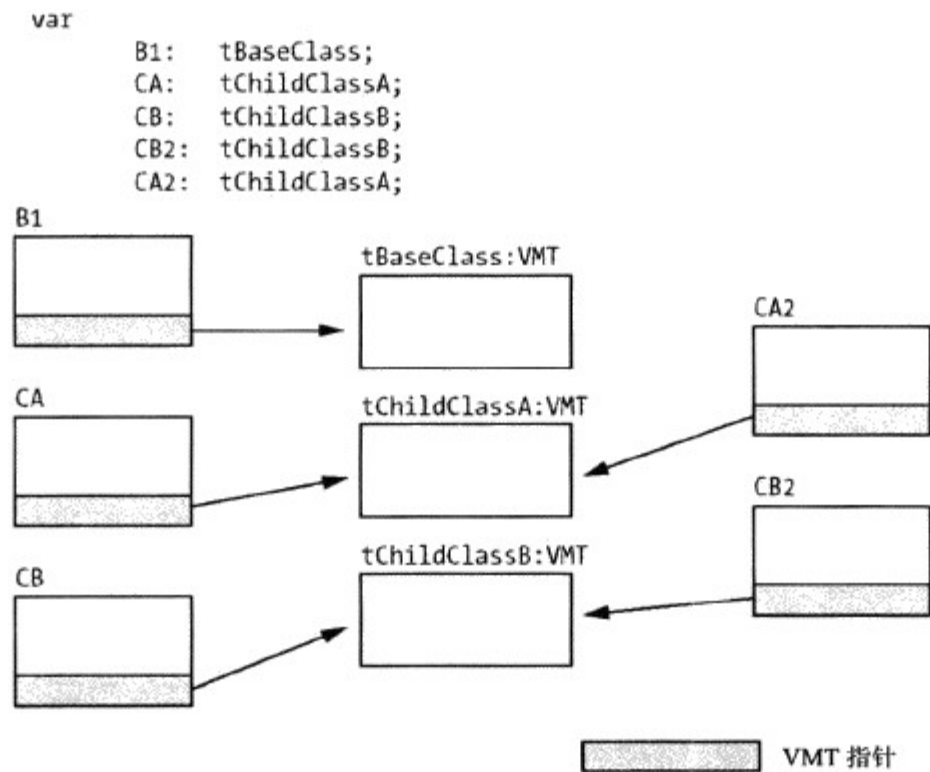


图 12-7 对象中虚拟方法表的引用

虚拟方法表只不过是一个指针数组，指向与类相关的方法和迭代器。出现在类中的第一个方法或迭代器的地址在偏移量为 0 的位置，第二个方法或迭代器的地址出现在偏移量为 4 的位置，依此类推。可以用 @offset 函数来确定一个给定方法或迭代器的偏移量值。如果想要(使用 80x86 语法，而不是 HLA 的高级语法)直接调用某个方法，可以用如下代码：

```

var
    sc: tBaseClass;

    lea( esi, sc );          // Get the address of the object (&VMT).
    mov( [esi], edi );      // Put address of VMT into edi.
    call( type dword [edi+@offset(tBaseClass.mBase)] );

```

当然，如果这个方法有参数，那么必须在执行上面的代码之前将参数压入栈。不要忘记，在对方法作直接调用时必须用对象的地址加载 ESI。方法中的任何域引用都可能依赖于包含这个地址的 ESI。选择 EDI 来包含 VMT 的地址几乎是随意的。除非做的是比较棘手的工作(像用 EDI 去获取运行时的类型信息)，否则可以在这里使用任何自己喜欢的寄存器。作为一个通用的规则，应该在模拟类方法调用时用 EDI，因为这是一个 HLA 采用的惯例，并且大多数程序员都将这样做。

子类从基类中继承域时，子类的 VMT 也会从基类的 VMT 中继承项。例如，tBaseClass 类的 VMT 只包含一个单独的项——一个指向 tBaseClass.mBase 方法的指针。tChildClassA 类的 VMT 包含两个项：一个指向 tBaseClass.mBase 方法的指针和一个指向 tChildClassA.mA 方法的指针。因为 tChildClassB 没有定义任何新的方法或迭代器，所以 tChildClassB 的 VMT 只包含一个单独的项，即一个指向 tBaseClass.mBase 方法的指针。注意，tChildClassB 的 VMT 和 tBaseClass 的 VMT 是一致的。然而，HLA 产生了两个单独的 VMT。这是我们后面将要用到的一个至关重要的事实。图12-8展示了这些 VMT 之间的关系。

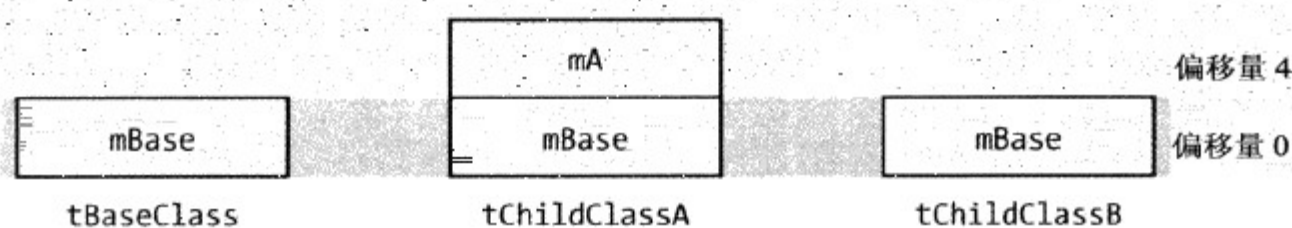


图 12-8 继承类的虚拟方法表

虽然 VMT 指针在一个对象中总是出现在偏移量为 0 的地方(因此，如果 ESI 指向一个对象，那么就可以用地址表达式[ESI]来访问该指针)，但是因为 HLA 实际上将一个符号插入到符号表中，所以可以用符号引用 VMT 指针。符号 `_pVMT_` (指向虚拟方法表的指针)就实现了这种符号引用。所以访问 VMT 指针的一种更合理的方式(就像前面的代码示例所示的那样)如下：

```

lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+@offset( tBaseClass.mBase )]) );

```

如果需要直接访问 VMT，那么有两种方法。声明类对象时，HLA 会自动包含一个名为 `_VMT_` 的域作为该类的一部分。`_VMT_` 是一个由双字对象组成的静态数组。因此可以用一个格式为 `classname._VMT_` 的标识符访问 VMT。通常，不要直接访问 VMT，但是在后面将会看到，需要知道这个对象在内存中的地址。

12.9 构造函数和对象初始化

如果现在就编写一个使用对象的程序,那么在运行程序时,就会发现程序会令人费解地崩溃。虽然迄今为止,我们在本章中已经讨论过许多内容,但是仍然遗漏了关键的内容——如何在使用对象之前合理地初始化它们。本节将会解决这个难题,并可以用类编写程序了。

考虑下面的对象声明和代码段:

```
var
    bc: tBaseClass;
```

```
bc.mBase();
```

记住,在 var 段声明的变量在运行时尚未进行初始化。因此,当包含这些语句的程序执行到 bc.mBase 时,它会执行下面 3 条指令:

```
lea( esi, bc);
mov( [esi], edi );
call( (type_dword [edi+@offset( tBaseClass.mBase )]);
```

这个指令序列的问题是假设前面并没有初始化 bc 对象,它就用一个未定义的值加载 EDI。因为 EDI 包含一个无用的数据值,所以在地址[EDI+@offset(tBaseClass.mBase)]处调用子例程可能会使系统崩溃。因此在使用对象之前,必须用该对象的 VMT 地址初始化 pVMT_域。完成该操作的一种比较容易的方式是使用如下语句:

```
mov( &tBaseClass.VMT, bc.pVMT );
```

始终要记住,使用一个对象之前,要确保已为该对象初始化了虚拟方法表的指针。

虽然必须为使用的所有对象初始化虚拟方法表的指针,但这不是唯一需要在对象中初始化的域。每一个特定的类都可能具有它自己的具体域需要初始化。虽然初始化可能根据类的不同而不同,但是需要对每个特定的类的对象进行同样的初始化。如果曾经从一个给定的类中创建了多个对象,那么创建过程来进行这些初始化工作是一个好主意。面向对象编程人员通常将这个初始化过程称为构造函数(constructor)。

一些面向对象编程语言(例如 C++)用一种特殊的语法来声明构造函数。还有一些语言(例如 Delphi)则简单地用现有的过程声明定义构造函数。使用特殊语法的好处是所使用的语言知道您什么时候定义了构造函数,并且可以自动产生代码来调用该构造函数(当声明对象时)。像 Delphi 这样的语言需要显式调用构造函数;这可能不太方便,也可能会造成程序中的缺陷。HLA 没有用特殊的语法声明构造函数:使用标准的类过程定义构造函数。这样需要在程序中显式调用构造函数;本章的 12.11 节会讲到自动完成这个操作的一种简便方法。

记住,构造函数必须是类过程。不能将构造函数定义为方法。原因非常简单,构造函数的任务之一是初始化指向虚拟方法表的指针,直到已经初始化 VMT 的指针为止都不可能调用类方法或迭代器。因为类过程没有用到虚拟方法表,所以可以在为对象初始化 VMT 指针之前调用类过程。

一般情况下,HLA 程序员为类的构造函数使用名称 `create`。虽然并不一定要使用这个名称,但是这样做可以使程序更易读,以及与其他程序员保持一致。

我们应该还记得,可以通过对象引用或类引用来调用类过程。例如,如果 `clsProc` 是 `tClass` 类的一个类过程,而 `Obj` 是 `tClass` 类型的一个对象,那么下面两个类过程调用都是合法的:

```
tClass.clsProc();
Obj.clsProc();
```

这两个调用之间有很大的区别。第一个调用用包含 0(NULL)的 `ESI` 调用了 `clsProc`,而第二个调用则在调用之前将 `Obj` 的地址加载到 `ESI` 中。我们可以据此确定一个方法中特殊的调用机制。

12.9.1 构造函数中的动态对象分配

事实上,大多数程序用 `mem.alloc` 动态分配对象,并且用指针间接引用这些对象。这为初始化过程增加了一个步骤——为对象分配存储空间。构造函数是分配这个存储空间最好的地方。因为有时不需要动态分配所有的对象,所以有两种类型的构造函数:一种分配存储空间并初始化对象,而另一种则简单地初始化一个已经拥有存储空间的对象。

另外一种有关构造函数的约定是将这两个构造函数合并成一个单独的构造函数,并根据 `ESI` 中的值区分构造函数调用的类型。在进入类的 `create` 过程的入口处,程序通过检查 `ESI` 的值来确定它是否包含 `NULL(0)`。如果是的话,构造函数调用 `mem.alloc` 为对象分配存储空间,并在 `ESI` 中返回一个指向对象的指针。如果在过程的入口处, `ESI` 没有包含 `NULL`,那么构造函数假定 `ESI` 指向一个有效的对象,并跳过内存分配语句。但至少构造函数会初始化指向 `VMT` 的指针;因此最简单的构造函数如下所示:

```
procedure tBaseClass.create; @nodisplay;
begin create;

  if ( ESI = 0 ) then
    push( eax ); // mem.alloc returns its result here, so save it.
    Mem.alloc( @size( tBaseClass ) );
    mov( eax, esi ); // Put pointer into esi;
    pop( eax );
  endif;

  // Initialize the pointer to the VMT:
  // Remember, "this" is shorthand for "(type tBaseClass [esi])"
  mov( &tBaseClass._VMT_, this._pVMT_ );

  // Other class initialization would go here.

end create;
```

编写了构造函数之后,需要根据对象的存储空间是否已经分配来选择合理的调用机制。对于那些预先分配的对象(也就是那些已经在 `var`、`static` 或 `storage` 段⁷中声明的或者已经通过 `mem.alloc`

⁷ 通常不会在 `readonly` 段中声明对象,因为这样做不能对它们进行初始化。

预先分配存储空间的对象), 只需要简单地将对象的地址加载到 ESI 中, 并调用构造函数即可。对于那些作为变量声明的对象, 就非常容易了, 只需要调用适当的 create 构造函数即可:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
    .
    .
    .
    bc0.create(); // Initializes preallocated bc0 object.
    .
    .
    .
    // Allocate storage for bcp object.
    mem.alloc( @size(tBaseClass) );
    mov( eax, bcp );
    .
    .
    .
    bcp.create(); // Initializes preallocated bcp object.
```

注意, 虽然 bcp 是一个指向 tBaseClass 对象的指针, 但是 create 方法并没有自动为这个对象分配存储空间。程序已经在前面分配了存储空间。因此, 当程序调用 bcp.create 时, 它用一个包含在 bcp 中的地址加载 ESI; 因为地址不是 NULL, 所以 tBaseClass.create 过程没有为新的对象分配存储空间。顺便说一下, 对 bcp.create 的调用将发出如下的机器指令序列:

```
mov( bcp, esi );
call tBaseClass.create;
```

到现在为止, 关于一个类过程调用的代码示例总是以 lea 指令开始。这是因为到现在为止, 所有示例使用的都是对象变量而不是指向对象变量的指针。记住, 类过程(方法)调用将在 ESI 寄存器中传递对象的地址。对于对象变量, HLA 发出一条 lea 指令以获得该地址。而对于指向对象的指针, 真正的对象地址是指针变量的值; 因此, 为了将对象的地址加载到 ESI 中, HLA 发出了一条 mov 指令将指针的值复制到 ESI 寄存器中。

在前面的示例中, 程序在调用对象构造函数之前就为对象预先分配了存储空间。虽然预先分配对象的存储空间有很多原因(例如构造的是一个动态对象数组), 但是可以像上面的示例那样通过调用一个标准的 create 过程(也就是如果 ESI 包含 NULL 才分配存储空间)而获得最简单的对象分配。下面的示例说明了这一点:

```
var
    bcp2: pointer to tBaseClass;
    .
    .
    .
    tBaseClass.create(); // calls create with esi=NULL.
    mov( esi, bcp2 );    // Save pointer to new class object in bcp2.
```


记住, 对 `tBaseClass.create` 构造函数的调用会在 `ESI` 寄存器中返回一个指向新对象的指针。将这个函数返回的指针保存到适当的指针变量中是调用者的职责; 构造函数是不会自动完成这些功能的。同样, 在应用程序使用完该对象之后, 释放和这个对象相关的存储空间也是调用者的职责(参见本章 12.10 节关于析构函数的讨论)。

12.9.2 构造函数和继承

从基类中继承域的派生(子)类的构造函数代表了一种特殊的情况。每个类都必须拥有它自己的构造函数, 但是也都需要拥有调用基类的构造函数的能力。本节将解释这样做的原因以及如何才能达到这个目的。

派生类从它的基类中继承了 `create` 过程。可是, 因为派生类可能需要比基类更多的存储空间, 所以必须在派生类中重写这个过程, 并且因此需要用不同的 `mem.alloc` 调用为动态对象分配存储空间。因此, 派生类不重写 `create` 过程的定义很少见。

然而, 重写一个基类的 `create` 过程有它自己的问题。当重写基类的 `create` 过程时, 要为初始化(整个)对象负全部的责任, 包括基类所需要的所有初始化。至少有一点, 那就是这将牵涉到在重写的过程中放置重复的代码来处理通常由基类的构造函数完成的初始化工作。除了使程序变得更大(因为复制了那些已经在基类构造函数出现过的代码)以外, 这也违背了信息隐藏的原则, 因为派生类必须知道基类中的所有域(包含那些对于基类而言逻辑上私有的域)。在这里, 我们需要的是从派生类的构造函数中调用基类的构造函数, 并让该调用进行基类域的低级初始化工作。幸运的是, 在 HLA 中这是很容易实现的。

考虑下面的类声明(它使用一种比较困难的方式来完成声明):

```
type
  tBase: class
    var
      i:uns32;
      j:int32;
    procedure create(); @returns( "esi" );
  endclass;

  tDerived: class inherits( tBase );
    var
      r:real64;
    override procedure create(); @returns( "esi" );
  endclass;

  procedure tBase.create; @nodisplay;
  begin create;

    if( esi = 0 )then

      push( eax );
      mov( mem.alloc( @size( tBase )), esi );
      pop( eax );

    endif;
    mov( &tBase._VMT_, this._pVMT_);
```

```

        mov( 0, this.i );
        mov( -1, this.j );

end create;

procedure tDerived.create; @nodisplay;
begin create;

    if( esi = 0 )then

        push( eax );
        mov( mem.alloc( @size(tDerived) ), esi );
        pop( eax );

    endif;

    // Initialize the VMT pointer for this object:

    mov( &tDerived._VMT_, this._pVMT_ );

    // Initialize the "r" field of this particular object:

    fldz();
    fstp( this.r );

    // Duplicate the initialization required by tBase.create:

    mov( 0, this.i );
    mov( -1, this.j );

end create;

```

仔细观察上面的 `tDerived.create` 过程。就像传统的构造函数一样，该过程首先检查 `ESI`，如果 `ESI` 包含 `NULL` 值，则为新对象分配存储空间。注意，`tDerived` 对象的大小包含被继承的域所需要的大小，因此这恰好为 `tDerived` 对象中的所有域分配了所需要的存储空间。

接下来，`tDerived.create` 过程对对象的 VMT 指针域进行初始化。记住，每个类都有它自己的 VMT，特别是派生类不会使用其基类的 VMT。因此，这个构造函数必须用 `tDerived` 的 VMT 的地址初始化 `_pVMT_` 域。

在初始化 VMT 指针后，`tDerived` 构造函数将 `r` 域的值初始化为 0.0(记住，`fldz` 将 0 加载到 FPU 栈上)。这就结束了 `tDerived` 特定的初始化过程。

`tDerived.create` 中余下的指令需要认真分析。这些语句复制了一些出现在 `tBase.create` 过程中的代码。代码复制的问题在决定更改这些域的初始值时才会真正地显现；如果已经在派生类中复制了初始化代码，那么就需要在多个 `create` 过程中改变初始化代码。然而，通常情况下，这样做将在派生类的 `create` 过程中导致故障，尤其是当这些派生类与基类不在同一个源文件中时。

在派生类构造函数中隐匿基类的初始化将违背信息隐藏原则。基类的一些域可能在逻辑上是私有的。虽然 HLA 没有在类中显式地支持公共域和私有域的概念(就像在 C++ 中那样)，但是有经验的程序员仍然会将域区分为私有或公共，并且只在属于该类的类例程中使用私有域。在派生类中初始化这些私有域对这类程序员而言是不可接受的。这样做会使得在以后改变基类的定义和实现变得非常困难。

幸运的是,HLA 为在派生类的构造函数中调用被继承的构造函数提供了一种方便的机制。所要做的就是用类名语法调用基类构造函数,例如,可以在 `tDerived.create` 中直接调用 `tBase.create`。通过调用基类的构造函数,派生类构造函数可以初始化基类的域,而不用担心基类的具体实现(或初始值)。

然而,每一个(传统的)构造函数都可以完成两种初始化类型,这将影响到调用基类构造函数的方式:所有传统的构造函数在 `ESI` 包含 0 的情况下都将为类分配内存,并且所有传统的构造函数都会初始化 `VMT` 指针。幸运的是,这两个问题很容易处理。

大部分基类对象所需的存储空间通常小于从基类中继承的类的对象所需的存储空间(因为派生类通常增加了更多的域)。因此,从派生类的构造函数内部调用基类构造函数时不允许基类构造函数去分配存储空间。通过在派生类的构造函数中检查 `ESI` 并在调用基类的构造函数之前为对象分配必需的存储空间,就很容易解决这个问题。

第二个难题是 `VMT` 指针的初始化。当调用基类的构造函数时,它将用基类的虚拟方法表的地址初始化 `VMT` 指针。然而,派生类对象的 `_pVMT_` 域必须指向派生类的虚拟方法表。对基类构造函数的调用将导致用错误的指针初始化 `_pVMT_` 域。为了用合理的值正确地对 `_pVMT_` 域进行初始化,派生类的构造函数必须在对基类构造函数进行调用之后将派生类的虚拟方法表的地址保存到 `_pVMT_` 域中(这样,它将重写基类构造函数写入的值)。

下面是被重写的用来调用 `tBase.create` 构造函数的 `tDerived.create` 构造函数:

```
procedure tDerived.create; @noDisplay;
begin create;

    if( esi = 0 )then
        push( eax );
        mov( mem.alloc(@size( tDerived )), esi );
        pop( eax );

    endif;

    // Call the base class constructor to do any initialization
    // needed by the base class. Note that this call must follow
    // the object allocation code above (so esi will always contain
    // a pointer to an object at this point and tBase.create will
    // never allocate storage).

    (type tBase [esi]).create();

    // Initialize the VMT pointer for this object. This code
    // must always follow the call to the base class constructor
    // because the base class constructor also initializes this
    // field and we don't want the initial value supplied by
    // tBase.create.

    mov( &tDerived._VMT_, this._pVMT_ );

    // Initialize the "r" field of this particular object:

    fldz();
    fstp( this.r );
```

```
end create;
```

这个解决方案解决了上面所有与派生类的构造函数相关的问题。注意，对基类的构造函数的调用使用了 `(type tBase[esi]).create()`；而不是 `tBase.create()`；语法。直接调用 `tBase.create` 的问题是，这会把 NULL 加载到 ESI 中，并将指针重写为指向由 `tDerived.create` 分配的存储空间。上面的方案在调用 `tBase.create` 时使用了 ESI 中的已有值。

12.9.3 构造函数的参数和过程重载

到目前为止，所有的构造函数示例都没有任何参数。然而，构造函数也可以使用参数。构造函数是过程；因此，可以指定任意数量和类型的参数。可以用这些参数值初始化特定的域或者控制构造函数初始化域的方式。当然，可以像在其他过程中使用参数那样使用构造函数参数。事实上，唯一要关心的是当拥有派生类时如何使用参数。本节将会讨论这些问题。

在派生类的构造函数中使用参数的第一个、也是最重要的一个问题适用于所有重写的过程和方法：一个重写例程的参数列表必须与基类中对应例程的参数列表完全相匹配。事实上，HLA 甚至都没有给程序员违背这条原则的机会，因为 `override` 例程的原型不允许参数列表的声明，它们自动继承基类例程的参数列表。因此，不能在构造函数原型中为一个类使用特殊的参数列表，而为出现在基类或继承类中的构造函数使用不同的参数列表。虽然能实现这种功能有时会非常方便，但是关于 HLA 为什么不支持这样做有充分的理由⁸。

HLA 支持一种特殊的 `overloads` 声明，允许使用一个标识符(其参数类型的编号指定了要调用的函数)调用不同的过程、方法或迭代器。这样就可以为给定的类(或派生类)创建多个构造函数，并使用参数列表调用相应的构造函数。感兴趣的读者可以阅读 HLA 文档中有关过程的章节，以了解关于 `overloads` 声明的更多细节。

12.10 析构函数

析构函数(`destructor`)是程序使用完对象后用来清除该对象的类例程。像构造函数一样，HLA 没有为析构函数提供特殊的语法，而且 HLA 也不会自动调用析构函数。和构造函数不一样的是，析构函数通常是一个方法而不是过程(因为虚拟析构函数非常有意义，而虚拟构造函数则没有什么意义)。

一个典型的析构函数将关闭任何被对象打开的文件、释放在对象使用期间分配的存储空间，并且如果对象是动态创建的话，就最终释放这个对象。析构函数还处理一些其他的在对象终止并退出之前需要做的清理工作。

一般情况下，大多数 HLA 程序员将他们的析构函数命名为 `destroy`。通常，大多数析构函数都具有的代码是释放和对象相关的存储空间的代码。下面的析构函数演示了如何完成这些工作：

```
procedure tBase.destroy; @nodisplay;
begin destroy;
```

⁸ 调用虚拟方法和迭代器将是一个真正的难题，因为不知道指针所引用的究竟是哪一个例程。因此，不可能知道正确的参数列表。虽然过程的问题不是非常严重，但是如果基类或派生类允许用不同的参数列表重写过程，那么有一些不易察觉的问题可能会引入到代码中。

```

push( eax ); // isInHeap uses this.

// Place any other cleanup code here.
// The code to free dynamic objects should always appear last
// in the destructor.

    /*****/

// The following code assumes that esi still contains the address
// of the object.

if( mem.isInHeap( esi )) then

    free( esi );

endif;
pop( eax );

end Destroy;

```

如果 HLA 标准库例程 `mem.isInHeap` 的参数是一个 `mem.alloc` 返回的地址,那么它将返回 `true`。因此,如果程序最初通过调用 `mem.alloc` 为对象分配存储空间,那么这个代码将自动释放和对象相关的存储空间。显然,在从这个方法调用返回时,如果动态分配了对象的存储空间,那么 `ESI` 将不再指向存储器中合法的对象。注意,这些代码不会影响 `ESI` 中的值,而如果对象不是先前通过调用 `mem.alloc` 分配的,那么它也不会更改对象。

12.11 HLA 的 `_initialize_` 和 `_finalize_` 字符串

虽然 HLA 不会自动调用与类相关的构造函数和析构函数,但是却提供了一种机制,借此可以强迫 HLA 自动进行这些调用:通过使用 HLA 在每个过程中自动声明的 `_initialize_` 和 `_finalize_` 编译时字符串变量(也就是 `val` 常量)。

无论何时编写过程、迭代器或方法,HLA 都会在例程中自动声明几个局部符号。其中两个符号是 `_initialize_` 和 `_finalize_`。HLA 声明这些符号的方法如下:

```

val
    _initialize_: string := "";
    _finalize_: string := "";

```

在例程体的最开始,HLA 发出了作为文本的 `_initialize_` 字符串,也就是,在例程的 `begin` 子句后立即发出⁹。同样,HLA 在例程体的最末尾发出 `_finalize_` 字符串,恰好在 `end` 子句之前。就像下面的代码一样:

```

procedure SomeProc;
    << declarations >>
begin SomeProc;

    @text( _initialize_ );

```

⁹ 如果例程自动发出代码去构建活动记录,那么 HLA 在构建活动记录的代码之后发出 `_initialize_` 的文本。

```
<< Procedure body >>
```

```
@text(_finalize_);
```

```
end SomeProc;
```

因为 `_initialize_` 和 `_finalize_` 最初包含空字符串，所以这些扩展对于 HLA 产生的代码没有任何影响，除非在 `begin` 子句之前显式地改变了 `_initialize_` 的值，或者在过程的 `end` 子句之前更改了 `_finalize_`。所以如果更改了这两个字符串对象中的任何一个去包含机器指令，那么 HLA 将在过程的最开始或最末尾编译该指令。下面的示例演示了如何使用这种技术：

```
procedure SomeProc;
    ?_initialize_ := "mov( 0, eax );";
    ?_finalize_ := "stdout.put( eax );";
begin SomeProc;

    // HLA emits "mov( 0, eax );" here in response to the _initialize_
    // string constant.

    add( 5, eax );

    // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

当然，这些示例对您不会有太多的帮助。在过程的最开始和最末尾输入实际的语句将比把包含这些语句的字符串赋值给 `_initialize_` 和 `_finalize_` 编译时变量要更容易些。然而，如果我们能自动将一些字符串赋给这些变量，从而不需要在每个过程中显式分配它们，那么这个特性也许还有些用。很快，您就会看到我们如何自动地将这些值分配到 `_initialize_` 和 `_finalize_` 字符串中。目前，只考虑将构造函数名称加载到 `_initialize_` 字符串中和将析构函数名称加载到 `_finalize_` 字符串中的情况。通过完成这些，例程将“自动”为特定的对象调用构造函数和析构函数。

前面的示例有一个小问题。如果能自动将一些值赋给 `_initialize_` 或 `_finalize_`，那么当这些变量已经包含了一些值时，会发生什么情况呢？例如，假设我们有两个在同一例程中使用的对象，并且第一个对象将其构造函数名称加载到了 `_initialize_` 字符串中；那么当第二个对象尝试同一操作时，会发生什么情况呢？解决的办法很简单：不要直接将任何字符串赋给 `_initialize_` 和 `_finalize_` 编译时变量；而应该将字符串连接到这些变量中现有字符串的末尾。下面的代码是对上面示例的一个修正，展示了如何完成这些操作：

```
procedure SomeProc;
    ?_initialize_ := _initialize_ + "mov( 0, eax );";
    ?_finalize_ := _finalize_ + "stdout.put( eax );";
begin SomeProc;

    // HLA emits "mov(0, eax );" here in response to the _initialize_
    // string constant.

    add( 5, eax );

    // HLA emits "stdout.put( eax );" here.
```



```
end SomeProc;
```

在将值赋给 `_initialize` 和 `_finalize` 字符串时, HLA 保证将在进入例程时执行 `_initialize` 序列。糟糕的是, 在退出时对 `_finalize` 字符串却不是这样。HLA 只是在例程的结尾, 在代码清理活动记录并返回之前立即为 `_finalize` 字符串发出代码。而且, “直到例程的最后退出”并不是程序从例程中返回的唯一方式。程序可以通过执行 `ret` 指令从代码中间的某个地方返回。因为 HLA 只在例程的末尾发出 `_finalize` 字符串, 所以, 这样从例程中返回就绕过了 `_finalize` 代码。因此, 只能手动发出 `_finalize` 代码¹⁰。幸运的是, 这种退出例程的机制完全在自己的控制之下; 如果除了通过“直到例程的最后退出”以外, 没有其他方式退出例程, 那么就不需要担心这个问题(注意, 如果真想从例程代码中间的某个地方返回, 那么可以使用 `exit` 控制结构将控制转移到该例程的最后)。

另一种提前退出例程的方法是产生一个异常, 但是在这种方式下, 不能进行任何控制。例程可以调用另外一些例程(例如一个标准库例程)而产生一个异常, 并且马上将控制转移到例程的调用者中。幸运的是, 通过在过程中加入 `try..endtry` 块, 很容易捕获异常并处理异常。如下面的示例所示:

```
procedure SomeProc;
  << Declarations that modify _initialize and _finalize >>
begin SomeProc;

  << HLA emits the code for the _initialize string here. >>

  try // Catch any exceptions that occur:
    << Procedure body goes here. >>

  anyexception
    push( eax );           // Save the exception #.
    @text( _finalize );    // Execute the _finalize code here.
    pop( eax );            // Restore the exception #.
    raise( eax );          // Reraise the exception.

  endtry;

  << HLA automatically emits the _finalize code here. >>
end SomeProc;
```

虽然前面的代码解决了一些伴随 `_finalize` 而存在的问题, 但是它绝不可能解决所有的问题。要不断发现自己的程序可能没有执行 `_finalize` 字符串中的代码而意外退出例程的方式。如果遇到了这种情况, 那么就应该显式扩展 `_finalize`。

关于异常, 有一个可能会碰到麻烦的重要地方: 在例程为 `_initialize` 字符串发出的代码内部。如果改变了 `_initialize` 字符串, 以至它所包含的构造函数调用和执行产生了异常, 那么可能会被迫从该例程中退出, 而不执行相关的 `_finalize` 代码。虽然能将 `try..endtry` 语句直接隐藏在 `_initialize` 和 `_finalize` 字符串中, 但是使用这种方法有几个问题。首先, 调用的构造函数中可能会产生异常, 而这个异常会将控制转移到异常处理程序中, 而该异常处理程序将在例程中为所有对象调用构造

¹⁰ 注意, 可以使用 `@text(_finalize);` 语句手动发出 `_finalize` 代码。

函数(包括那些还没有调用过它们的构造函数的对象)。虽然没有简单的解决办法能解决所有问题,但最好是将 `try..endtry` 块放在每个构造函数调用中,前提是构造函数可能产生一些可以进行处理异常(也就是不需要立即终止程序)。

到目前为止,有关 `_initialize_` 和 `_finalize_` 的讨论都没有强调非常重要的一点:为什么要使用这个特性来实现构造函数和析构函数的“自动”调用呢?它明显比简单地直接调用构造函数和析构函数要牵涉更多的工作。显然必定有一种方式可以自动完成 `_initialize_` 和 `_finalize_` 字符串的赋值,否则本节的内容就不应该存在。实现的方式是通过使用一个宏去定义类的类型。所以现在应该讨论 HLA 的另一个特性,即 `forward` 关键字,这个特征使得自动化成为了可能。

我们已经明白了如何使用 `forward` 保留字去构造过程原型(参见 5.9 节的论述),它证明可以提前声明 `const`、`val`、`type` 和变量声明。这种声明的语法采用如下格式:

```
ForwardSymbolName: forward( undefinedID );
```

这个声明和下面的声明是完全等效的:

```
?undefinedID: text := "ForwardSymbolName";
```

特别要注意的是,这个扩展不会真正定义符号 `ForwardSymbolName`。它仅仅是将这个符号转换成一个字符串,并将这个字符串赋给一个指定的 `text` 对象 `undefinedID`。

现在您可能会怀疑像上面的语句怎么会和 `forward` 声明等效呢?其实它们并不等效。然而,`forward` 声明通过将对象类型的实际声明推迟到后面代码中的某个地方,来创建模拟类型名称的宏。考虑下面的示例:

```
type
  myClass: class
    var
      i: int32;

    procedure create; @returns( "esi" );
    procedure destroy;
  endclass;

#macro _myClass: varID;
  forward( varID );
  ?_initialize_ := _initialize_ + @string:varID + ".create()";
  ?_finalize_ := _finalize_ + @string:varID + ".destroy()";
  varID: myClass
#endmacro;
```

注意,在这个宏最后的 `varID: myClass` 声明后面没有跟着分号。稍后我们会发现为什么没有这个分号。

如果在程序中有上面的类和宏的声明,那么现在可以声明 `_myClass` 类型的变量,这些变量会在包含变量声明的例程的入口和出口处自动调用构造函数和析构函数。为了理解这是如何工作的,看一下下面的过程 shell:

```

procedure HasmyClassObject;
var
    mco: _myClass;
begin HasmyClassObject;
    << Do stuff with mco here. >>
end HasmyClassObject;

```

因为 `_myClass` 是一个宏，所以上面的过程在编译时将扩展为下面的文本：

```

procedure HasmyClassObject;
var
    mco:          /* Expansion of the _myClass macro:
    forward(=0103 -); /* -0103 symbol is an HLA-supplied text
                    // symbol that expands to "mco".

    ?_initialize := _initialize + "mco" + ".create()";
    ?_finalize   := _finalize   + "mco" + ".destroy()";
    mco:_myClass;

begin HasmyClassObject;
    mco.create(); /* Expansion of the _initialize_string.
    << Do stuff with mco here. >>
    mco.destroy(); /* Expansion of the _finalize_string.
end HasmyClassObject;

```

您也许注意到了在上面的示例中，有一个分号出现在 `mco:_myClass` 的后面。这个分号实际上不是宏的一部分；而是在最初的代码中跟在 `mco:_myClass;` 声明后的分号。

如果想构造一个对象数组，可以像下面这样合法地声明它：

```

var
    mcoArray: _myClass[10];

```

因为 `_myClass` 宏中最后一条语句并没有以分号结束，所以上面的声明将扩展为如下(接近正确的)代码：

```

mcoArray:          /* Expansion of the _myClass macro:
    forward(=0103 -); /* -0103 symbol is an HLA-supplied text
                    // symbol that expands to "mcoArray".

    ?_initialize := _initialize + "mcoArray" + ".create()";
    ?_finalize   := _finalize   + "mcoArray" + ".destroy()";
    mcoArray:_myClass[10];

```

这个扩展的唯一问题是它只为数组中的第一个对象调用构造函数。有几种方式可以解决这个问题，其中一个是将宏名而不是构造函数名附加到 `_initialize` 和 `_finalize` 的末尾。该宏将检查对象名(这个示例中的 `mcoArray`)来决定它是否是一个数组。如果是，那么宏可以扩展为一个为数组中

的每个元素调用构造函数的循环。

这个问题的另一种解决办法是使用宏的参数为 `myClass` 数组指定维数。虽然这种方法比上面的方法更容易实现,但是它有一个缺点——需要不同的语法来声明对象数组(必须使用圆括号而不是方括号将数组的维数括起来)。

`Forward` 伪指令的功能是非常强大的,可以用来实现多种技巧。但应该要注意一些问题。首先,因为 HLA 会显式地发出 `_initialize_` 和 `_finalize_` 代码,所以如果有任何错误出现在这些字符串中,就很容易被迷惑。如果在例程中得到与 `begin` 或 `end` 语句相关的错误信息,那么需要看一下该例程中的 `_initialize_` 和 `_finalize_` 字符串。在这些字符串后面附加非常简单的语句,这样就可以减少错误发生的可能性。

从本质上说,HLA 不支持自动调用构造函数和析构函数。本节已经给出了一些技巧尝试自动调用例程。然而,自动操作并不完善,而且前面提到的 `_finalize_` 字符串方面的问题限制了这些方法的应用。本节所提供的机制对于简单的类和简单的程序可能是不错的选择。如果代码比较复杂,或者对正确性要求苛刻,建议您最好是手动调用构造函数和析构函数。

12.12 抽象方法

抽象基类(`abstract base class`)的意义在于为它的派生类提供一组公共域。我们永远不会声明类型为抽象基类的变量,而总是使用派生类。抽象基类的用途仅仅是为构造其他类提供一个模板。我们会看到,标准基类和抽象基类在语法上的唯一区别是抽象基类中至少出现一个抽象方法声明。抽象方法是一个特殊的方法,它在抽象基类中没有一个真正的实现。任何调用这个方法的尝试将会产生异常。如果想知道抽象方法有什么作用,那么请继续往下读。

假设要创建一组类去保存数值。其中第一个类可能代表无符号整型,第二个类代表有符号整型,第三个类实现 BCD 值,而第四个类支持 `real64` 值。虽然可以创建 4 个功能独立的类,但是这样做就失去了使得这组类更方便使用的机会。想明白其原因吗?考虑下面的类声明:

```
type
    uint: class
        var
            TheValue: dword;

        method put;
        << Other methods for this class >>
    endclass;

    sint: class
        var
            TheValue: dword;

        method put;
        << Other methods for this class >>
    endclass;

    r64: class
        var
```

```

    TheValue: real64;

    method put;

    << Other methods for this class >>
endclass;

```

这些类的实现是合理的。它们有数据域，并且也有 `put` 方法(看起来，这个方法可能是将数据写到标准输出设备中)，而且可能它们在实现数据的不同操作时有另外的方法和过程。然而，这些类中存在两个问题，一个小问题和一个大问题，两个问题都是因为这些类没有从公共基类中继承任何域而产生的。

第一个问题，也是相对比较小的问题是必须要在这些类中重复声明几个公共域。例如，`put` 方法的声明出现在每个类中¹¹。伴随重复声明的结果是会产生很难维护的程序，因为它不鼓励您为公共函数使用公共名称，而这是因为它很容易在每个类中使用不同的名称。

而比较大的问题是它不通用。也就是说，不能创建一个指向“数值”对象的通用指针，然后对数值执行加、减和输出之类的操作(不考虑数值的底层表示)。

通过将前面的类声明转换成一组派生类，我们很容易解决这两个问题。下面的代码展示了完成该实现的一种简单方式：

```

type
    numeric: class
        method put;
        << Other common methods shared by all the classes >>
    endclass;

    uint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << Other methods for this class >>
    endclass;

    sint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << Other methods for this class >>
    endclass;

    r64: class inherits( numeric )
        var
            TheValue: real64;

        override method put;
        << Other methods for this class >>
    endclass;

```

¹¹ 注意，`TheValue` 不是一个公共域，因为这个域在 `r64` 类中有不同的类型。

这种方案解决了上面提到的两个问题。首先, 通过从 `numeric` 类中继承 `put` 方法, 该代码鼓励派生类始终使用名称 `put`, 从而使得程序更容易维护。其次, 因为这个示例使用了派生类, 所以就可能创建一个指向 `numeric` 类型的指针, 并且用 `uint`、`sint` 或 `r64` 对象的地址来加载这个指针。而这个指针可以调用在 `numeric` 类中找到的方法去完成诸如加、减或数值输出之类的功能。因此, 使用该指针的应用程序不需要知道确切的数据类型, 它只须以一种通用的方式处理数值即可。

使用这种方案的一个问题是可能会声明和使用 `numeric` 类型的变量。这样的 `numeric` 变量不会拥有表示任何数字类型的功能(注意, 数值域的数据存储实际上出现在派生类中)。而且, 因为已经在 `numeric` 类中声明了 `put` 方法, 所以即使从来不会真正调用这个方法, 也必须编写一些代码来实现该方法; 而真正的实现应该只出现在派生类中。虽然您可能已经编写了一个打印错误信息(或者还产生了异常)的虚拟方法, 但是本来应该没有任何必要编写像这样的“虚拟”过程。而如果使用抽象方法, 则不必这样做。

当 `abstract` 关键字跟在方法声明后面时, 会告诉 HLA 您不会为这个类提供方法的实现。相反, 为抽象方法提供一个具体的实现是所有派生类的责任。如果尝试直接调用抽象方法, 那么 HLA 将会产生异常。下面是对 `numeric` 类的修改, 它将 `put` 转换为抽象方法:

```
type
    numeric: class
        method put; abstract;
        << Other common methods shared by all the classes >>
    endclass;
```

抽象基类是至少拥有一个抽象方法的类。注意, 不需要在抽象基类中使所有的方法都是抽象的; 在抽象基类中声明一些标准方法是完全合法的(当然要提供它们的实现)。

抽象方法声明提供了一种机制, 基类通过它可以指定一些派生类必须实现的通用方法。理论上, 所有的派生类必须提供所有抽象方法的具体实现, 否则那些派生类自己也是抽象基类。实际上, 稍微偏离一点是可能的, 并且因为一个稍微不同的目的而使用抽象方法也是可能的。

上面提到永远不要创建类型为抽象基类的变量。因为如果尝试执行一个抽象方法, 那么程序将立即产生一个异常来对这个非法的方法调用报错。而实际上, 可以声明一个抽象基类的变量, 并且只要不调用该类的任何抽象方法就可以避免这个异常。

12.13 运行时类型信息

当使用对象变量(而不是指向对象的指针)时, 该对象的类型是很明显的: 它就是变量声明的类型。因此, 在编译时和运行时, 程序都会知道对象的类型。而当使用指向对象的指针时, 通常情况下不可能确定指针引用的对象类型。然而, 在运行时确定对象的真正类型是可能的。本节讨论如何探测底层对象的类型和如何使用该信息。

如果有一个指向对象的指针并且该指针的类型是某个基类, 那么在运行时该指针可能指向一个基类或任何派生类型的对象。在编译时的任何时刻都不可能确定对象的确切类型。想明白为什么? 请考虑下面的小示例:

```
ReturnSomeObject();           // Returns a pointer to some class in esi.
mov(esi, ptrToObject);
```

例程 *ReturnSomeObject* 返回一个指向 ESI 中某个对象的指针。这可能是某个基类对象或派生类对象的地址。在编译时程序无法知道这个函数返回什么类型的对象。例如, *ReturnSomeObject* 可以询问用户返回什么值, 所以确切的类型只有在程序真正运行并且用户作出选择时才能被确定。

在一个设计完善的程序中, 不需要知道泛型对象的实际类型。毕竟, 面向对象编程和继承的目的就是产生通用程序, 该通用程序可以和许多不同的对象一起工作, 而不需要对程序作实质性的改变。然而在现实中, 程序可能没有一个完善的设计, 并且有时候知道一个指针引用的确切对象类型也是很不错的。运行时类型信息(runtime type information, RTTI)可以帮助您在运行时确定对象的类型, 即使您正在使用指向该对象的某个基类的指针来引用它。

最基本的 RTTI 操作是询问一个指针是否包含某个特定对象类型的地址。许多面向对象语言(例如 Delphi)用一个 *is* 操作符来提供这个功能。*is* 是一个布尔操作符, 如果它的左操作数(一个指针)指向一个类型与右操作数(它必须是一个类型标识符)匹配的对象, 那么就返回 *true*。其语法通常如下所示:

```
ObjectPointerOrVar is ClassType
```

如果变量是一个指定类的变量, 那么这个操作符将会返回 *true*; 否则返回 *false*。下面是这个操作符的一个典型应用(应用于 Delphi 语言中):

```
if( ptrToNumeric is uint )then begin
.
.
.
end;
```

在 HLA 中实现这个功能实际上非常简单。您可能还记得, 每一个类都有其自己的虚拟方法表。无论何时创建一个对象, 都必须使用这个类的 VMT 地址初始化指向 VMT 的指针。因此, 一个给定类类型的所有对象的 VMT 指针域都将包含同样的指针值, 并且这个指针值和所有其他类的 VMT 指针域的值不相同。我们可以运用这个事实查看一个对象是否是某个特定的类型。下面的代码展示了如何在 HLA 中实现上面的 Delphi 语句:

```
mov( ptrToNumeric, esi );
if( (type uint [esi] )._pVMT_ = &uint._VMT_ ) then
.
.
.
endif;
```

这个 *if* 语句简单地比较了对象的 *_pVMT_* 域(指向 VMT 的指针)和目标类的 VMT 地址。如果它们是相等的, 那么 *ptrToNumeric* 变量指向一个 *uint* 类型的对象。

在类的方法体或迭代器中, 有一种简单些的方式可以判断对象是否是特定的类。记住, 在进入一个方法或者迭代器的入口时, EDI 寄存器都会包含虚拟方法表的地址。因此, 假设您没有更

改 EDI 的值, 那么可以像下面这样使用 if 语句来测试该方法或迭代器是否是特定的类类型:

```
if( _edi = &uint._VMT_ ) then
.
.
.
endif;
```

然而要记住, 只有在调用类方法时, EDI 才包含一个指向 VMT 的指针。而当调用类过程的时候, 则并非如此。

12.14 调用基类的方法

在构造函数一节中, 我们知道可以在一个派生类的重写过程中调用一个祖先类的过程。为此, 只须使用 `(type classname[esi]).procedureName(parameters)`; 调用该过程。有时候也许想使用类的方法去做和类过程一样的操作(那就是, 为了做一些您不愿意在派生类的方法中重复的计算而让重写方法调用相应的基类方法)。遗憾的是, HLA 不允许像对过程那样直接调用方法, 而需要使用一个间接的机制来完成这个功能。具体来说, 必须使用基类的虚拟方法表中的地址调用方法。本节描述了如何做到这一点。

无论程序何时调用方法, 它都会使用在方法的类的虚拟方法表中找到的地址间接完成调用。虚拟方法表只是一个由 32 位指针组成的数组, 数组中的每一项都包含类中方法的地址。所以为了调用一个方法, 所需要的就是对该方法的地址的数组进行索引(更准确地说, 是相对数组的偏移量)。HLA 编译时函数 `@offset` 解决了这个问题: 它将返回虚拟方法表中的偏移量。结合 `call` 指令, 很容易调用任何和一个类相关联的方法。下面是一个示例:

```
type
  myCls: class
    .
    .
    .
    method m;
    .
    .
    .
  endclass;
.
.
.
call( myCls._VMT_ [ @offset( myCls.m ) ] );
```

上面的 `call` 指令调用了地址出现在 `myCls` 的虚拟方法表的指定项中的方法。`@offset` 函数调用返回了在虚拟方法表中 `myCls.m` 的地址偏移量(也就是索引乘以 4)。因此, 这段代码通过使用 `m` 的虚拟方法表项间接调用 `m` 方法。

使用这个方案来调用方法有一个比较大的缺陷: 没有过程/方法调用的高级语言语法。相反,

必须使用低级的 `call` 指令。在上面的示例中，因为 `m` 过程没有任何参数，所以这并不是一个太大的问题。如果它确实有参数，那么必须手动将这些参数推入栈中。幸运的是，很少需要从派生类中调用祖先类的方法，所以在实际的程序中这并不是太大的问题。

12.15 更多信息

HLA 参考手册中包含了更多的有关 HLA 类的实现信息。更多低级实现细节也请参考这个文档。本章并没有阐述面向对象编程范例。有关这个主题的细节请参考面向对象设计方面的教材。

附录 ASCII 字符集

二 进 制	十 六 进 制	十 进 制	字 符
0000_0000	00	0	NUL
0000_0001	01	1	CTRL A
0000_0010	02	2	CTRL B
0000_0011	03	3	CTRL C
0000_0100	04	4	CTRL D
0000_0101	05	5	CTRL E
0000_0110	06	6	CTRL F
0000_0111	07	7	bell
0000_1000	08	8	backspace
0000_1001	09	9	TAB
0000_1010	0A	10	line feed
0000_1011	0B	11	CTRL K
0000_1100	0C	12	form feed
0000_1101	0D	13	RETURN
0000_1110	0E	14	CTRL N
0000_1111	0F	15	CTRL O
0001_0000	10	16	CTRL P
0001_0001	11	17	CTRL Q
0001_0010	12	18	CTRL R
0001_0011	13	19	CTRL S
0001_0100	14	20	CTRL T
0001_0101	15	21	CTRL U
0001_0110	16	22	CTRL V
0001_0111	17	23	CTRL W
0001_1000	18	24	CTRL X
0001_1001	19	25	CTRL Y
0001_1010	1A	26	CTRL Z
0001_1011	1B	27	CTRL [
0001_1100	1C	28	CTRL \
0001_1101	1D	29	ESC
0001_1110	1E	30	CTRL ^

(续表)

二 进 制	十 六 进 制	十 进 制	字 符
0001_1111	1F	31	CTRL_
0010_0000	20	32	Space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	\$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2
0011_0011	33	51	3
0011_0100	34	52	4
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A

(续表)

二 进 制	十 六 进 制	十 进 制	字 符
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d

(续表)

二 进 制	十 六 进 制	十 进 制	字 符
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z
0111_1011	7B	123	{
0111_1100	7C	124	[
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	

[G e n e r a l I n f o r m a t i o n]

书名 = 汇编语言的编程艺术

作者 = (美) 海德著

页数 = 5 8 2

出版社 = 北京市：清华大学出版社

出版日期 = 2 0 1 1 . 1 2

S S 号 = 1 2 8 7 5 3 4 3

D X 号 = 0 0 0 0 0 8 2 1 4 8 6 8

URL = <http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000008214868&d=5F39A8861495327EFE5B3DC178F2E01F>