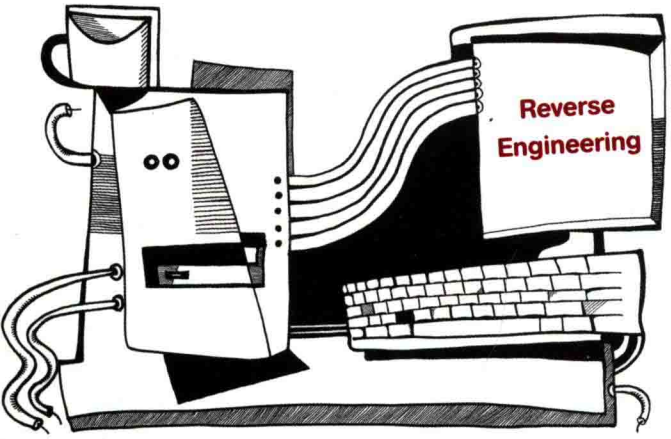
安天 ANTIY
异步图书 www.epubit.com.cn

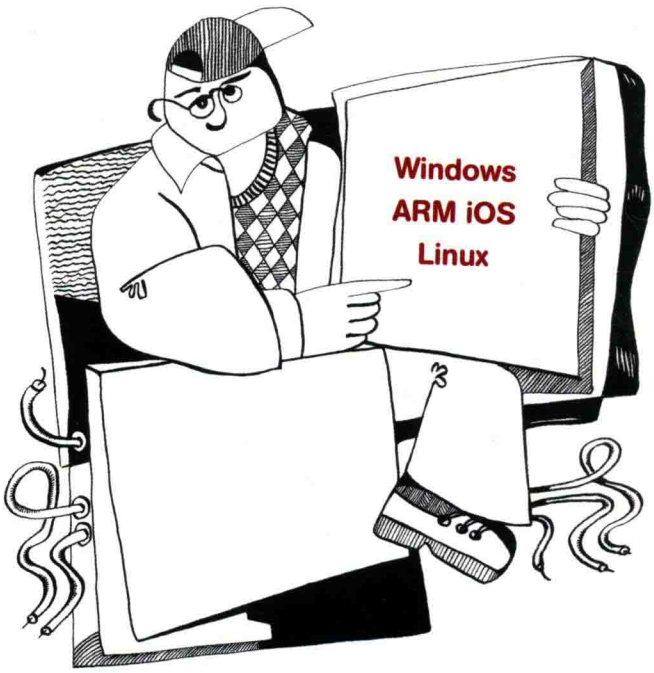


- 了解逆向工程的权威指南
- 初学者必备的大百科全书
- 安天网络安全工程师培训必读书目

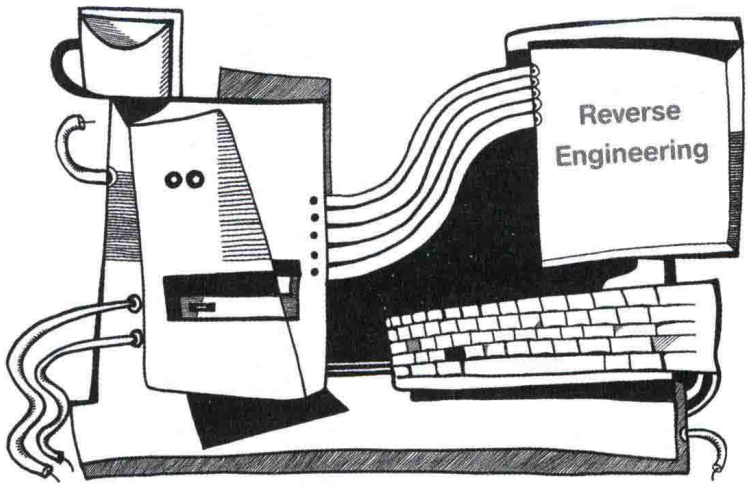
Reverse Engineering for Beginners

逆向工程权威指南 上册

[乌克兰] Dennis Yurichev◎著
Archer 安天安全研究与应急处理中心◎译



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



Reverse Engineering for Beginners

逆向工程权威指南 上册

[乌克兰] Dennis Yurichev◎著
Archer 安天安全研究与应急处理中心◎译

人民邮电出版社
北京

图书在版编目（CIP）数据

逆向工程权威指南 /（乌克兰）丹尼斯（Dennis Yurichev）著；Archer，安天安全研究与应急处理中心译. -- 北京：人民邮电出版社，2017.4
ISBN 978-7-115-43445-6

Ⅰ. ①逆… Ⅱ. ①丹… ②A… ③安… Ⅲ. ①工业产品－计算机辅助设计－指南 Ⅳ. ①TB472-39

中国版本图书馆CIP数据核字(2016)第243654号

版权声明

Simplified Chinese translation copyright ©2017 by Posts and Telecommunications Press
ALL RIGHTS RESERVED
Reverse Engineering for Beginners, by Dennis Yurichev

Copyright © 2016 by Dennis Yurichev

本书中文简体版由作者 Dennis Yurichev 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。
版权所有，侵权必究。

◆ 著　[乌克兰] Dennis Yurichev
译　Archer 安天安全研究与应急处理中心
责任编辑　陈冀康
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京天宇星印刷厂印刷

◆ 开本：787×1092　1/16
印张：61.5　彩插：1
字数：1 990 千字　2017 年 4 月第 1 版
印数：1 – 3 000 册　2017 年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-3227 号

定价：168.00 元（上、下册）

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

内 容 提 要

逆向工程是一种分析目标系统的过程，旨在识别系统的各组件以及组件间关系，以便能够通过其他形式或在较高的抽象层次上，重建系统的表征。

本书专注于软件的逆向工程，是写给初学者的一本权威指南。全书共分为上、下两册，十二个部分，共 102 章，涉及 X86/X64、ARM/ARM-64、MIPS、Java/JVM 等重要内容，详细解析了 Oracle RDBMS、Itanium、软件狗、LD_PRELOAD、栈溢出、ELF、Win32 PE 文件格式、x86-64（critical sections、syscalls、线程本地存储 TLS、地址无关代码（PIC）、以配置文件为导向的优化、C++ STL、OpenMP、SHE 等众多技术问题，堪称是逆向工程技术百科全书。除了详细讲解，本书还给出了很多习题来帮助读者巩固所学的知识，附录部分给出了习题的解答。

本书适合对逆向工程技术、操作系统底层技术、程序分析技术感兴趣的读者阅读，也适合专业的程序开发人员参考。

序　从逆向视角透视代码大厦的那些砖石

相对于本书的中文名字《逆向工程权威指南》，我更喜欢它的英文原名《Reverse Engineering for Beginners》，可以直译为“逆向工程入门者必读”。在国外的程序员圈中，这本书被简称为《RE4B》，作者的 Blog 式连载已经引发了持续关注和好评。

本书的重要特色在于其清晰的章节结构，涵盖了函数、语句、结构体、数组等这些构成代码大厦的基本元素，如显微镜般逐一对比了这些元素在 X86、ARM 和 MIPS 体系架构下的“切片”影像。本书几乎每个章节都按照 X86、ARM、MIPS 三个体系架构分别展开，可以让读者深入对比理解在三种体系架构下，同样的指令、函数及数据结构呈现的异同和特点。无论对分析工程师，还是编码工程师来说，本书都是能更深入理解代码大厦原材料的工作指南。这是基础性、系统性的文献工作，也是令人耳目一新的结构安排。对于在中国传统高校计算机教育中，通过简单的 X86 ASM 来学习和了解体系结构的工程师来说，这也是尽快打通 ARM、MIPS 知识背景的“全栈”指南。对于那些从各种破解教程来切入逆向领域的安全爱好者来说，这更像是一份“正餐”。对于作者来说，我相信这个写作过程是充满激情的，但也充满了艰难和痛苦。对于 Dennis 这样的逆向工程专家来说，这本书的写作过程，更多的不是在探索未知的世界，也不是对高级技巧和经验的总结，而是要对看起来相对枯燥的单元式资料进行整理，将一些对其已经是常识的东西转化为系统而书面化（并易于读者理解）的语言。

逆向工程一直被宣传为一种充满乐趣和带有神秘主义的技能，而逆向分析者的工作看起来更有乐趣的原因，似乎就是在指令奔涌中逆流而上，绕开种种限制和保护，找到代码中的“宝藏”——错误、漏洞或者是被加密算法层层掩盖的数据结构。这个由媒体所打造的形象，也为部分逆向爱好者所自矜。而这也导致逆向领域的书籍和教程，更多地围绕着一些高阶的技巧展开，更多地讲解如何绕过加密和保护，却忘记了逆向工程的初衷是要洞察系统及软件的整体结构和机理。由于之前的逆向领域相关出版物中缺少“代码大全”式的基础读本，本书的完成填补了这一空白，然而，这不仅需要作者有高超的逆向研究能力，也要有正向系统的思维和视野。

本书的译者之一 Archer 提出想邀请安天的工程师们一同完成本书的翻译时，我曾经觉得他是不是“疯了”。本书当时并无定稿的英文版本，只是 Dennis 持续发布的 Blog 式的连载。尽管网络连载已经引发了好评如潮，但其精彩篇章彼时还如散落在海底的珍珠，尚未串成珠链。以 Dennis 天马行空的写作风格和追求完美主义的性格，他必然会在未完成全部章节的期间，不断地增加体系结构的新热点（如 ARM64）和修补既有章节的内容，甚至会改动原有的样例代码，导致本书原版定稿遥遥无期。我觉得，与其说正在养病的 Archer 接手翻译这本书是一份艰难的任务，倒不如说，Dennis 写完本书根本就是一份不可完成的任务。

因此当接近千页的样书摆在我面前时，我被深深震撼了，那种感觉和我之前依靠自己极为蹩脚的英语扫过的英文电子稿不同，我能想象 Archer 是如何一边咳嗽着、一边码字或者校对代码；以及安天 CERT 的同事们是如何在样本分析任务饱和的情况下挤出时间来一点点推动翻译进展的。当我在视频例会上向安天各地的部门负责人展示样书时，我能看到每个人的赞叹和兴奋，那种兴奋不啻于我们自己发布了一份最新的长篇分析报告。

二进制分析，是安全分析工程师，包括有志于投身安全领域的编码工程师的基本功底之一。硅谷一些安全人士都曾提及一个有趣的说法——华人的系统安全天赋。中国安全厂商和安全工作者，在系统安全领域和分析工作中，正在不断取得新的进步。在国际网络安全企业中，华人承担关键系统安全研究和分析工作的占比也很高，国际高校的一些华人研究者在新兴领域展开系统安全研究，同样取得了很大的

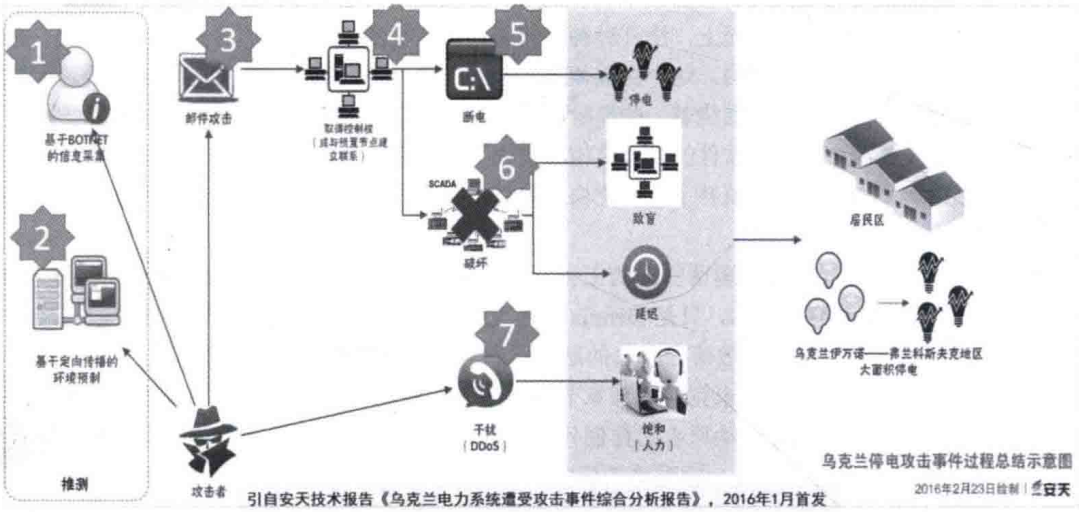
国际影响。遗憾的是，Web 和应用开发的兴起，一定程度上让更多年轻安全从业人员的目光转向 Web 和应用安全，当前新的安全从业者整体的二进制分析能力正在下降，存在着基本功缺失的问题，非常需要系统性的补课。

二进制分析能力，是对安全工程师的基础能力要求。但要对抗 APT 等新兴威胁，仅仅有基本功是不够的，还需要在更开阔的架构视野上，具备从攻击链、体系化等角度逆向分析研判攻击行为的全程、攻击对手的全貌的能力。

当年受到一些破解教程的误导，一些接触逆向工程的年轻人把破解序列号等当成逆向工程的目的，从而导致浅尝辄止，没有真正理解逆向工程的博大精深。即使是安天部分资深的分析工程师，也往往更乐于寻找解析攻击者的个性技巧和那些有“难度”的分析细节，并常以此感到自豪，而不能站到对攻击作业过程完整复盘和对攻击者画像的角度来看待问题，而后者才是一类高阶的逆向工程。

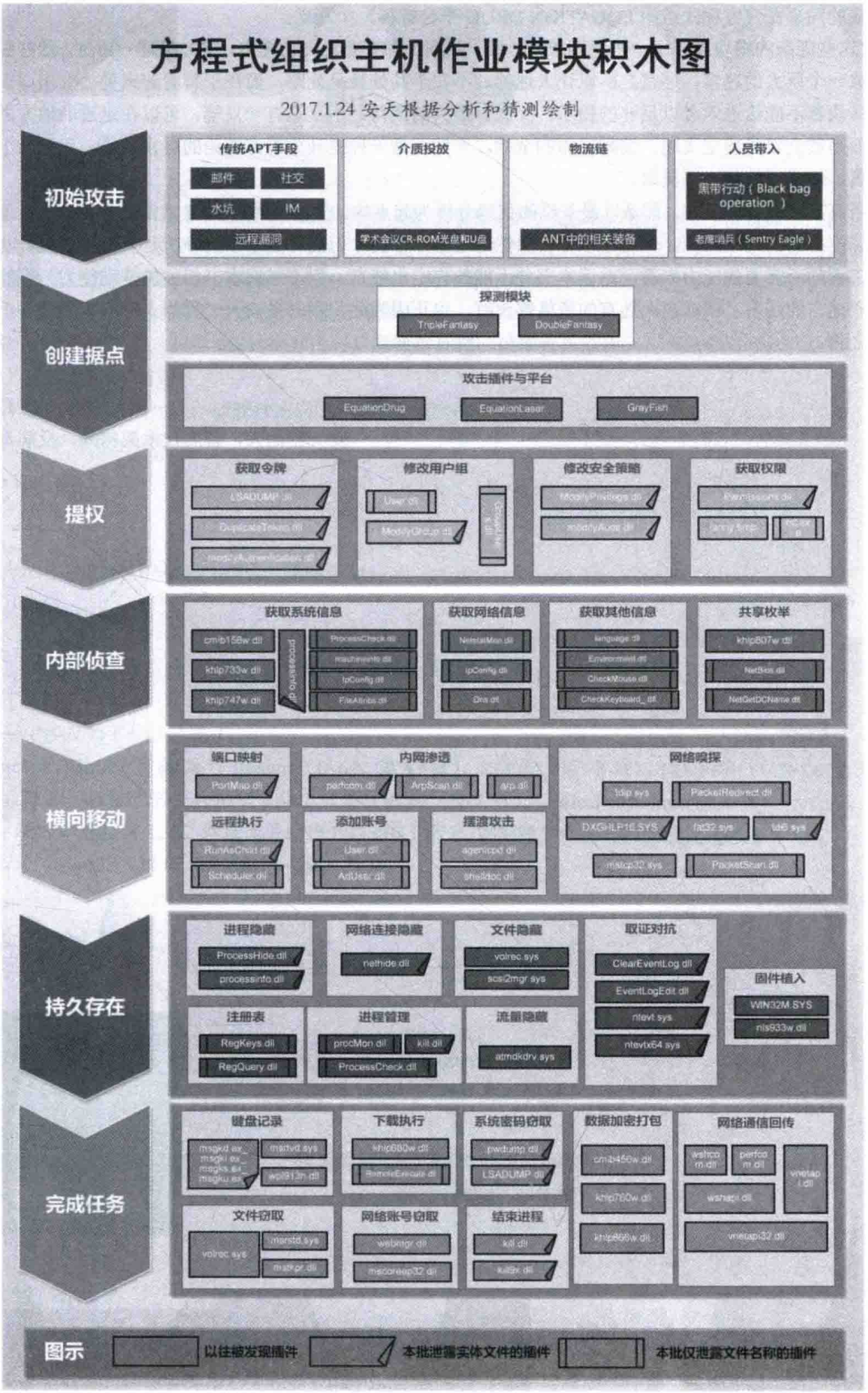
我们是从分析“震网”所遇到的系列挫折中，从对 Flame 蠕虫马拉松式的、收效甚微的模块分析中，认识到了传统分析工程师所面临的这种困境的。在“震网”分析中，由国际知名厂商的架构师和分析工程师所组成的混编分析团队，就能比安天完全由软硬件安全分析工程师组成的分析小组，取得更多的分析成果和进展。因为安全厂商和 APT 攻击之间，不是简单的查杀反制和单点的分析对抗，而是体系化对抗。从认识到这个问题开始，安天的分析团队一方面继续强化二进制分析的基本功底；另一方面自我批判那种“视野从入口点开始”的狭隘分析视角，避免我们自己从透析攻击者载荷的高级技巧中找到太多“快感”，而忘记那些更重要的全局性因素。

过去三年间，安天发布了针对 APT-TOCS、白象、方程式等组织相关的多篇分析报告，也分析了类似乌克兰国家电网遭遇攻击停电等与关键基础设施有关的安全事件。在乌克兰停电事件的分析中，我们已经初步走出了入口点视野，而尝试在更大的时空视角上还原事件。



而即使从微观的分析来看，高阶恶意代码也已经从一个附着在宿主上的代码片段或单体文件的木马，进化为一个庞大的工程体系。对于高级恶意代码来说，不仅有前端的持久化载荷和大量按需投放的“原子化”模块，其后更有一个庞大的支撑工程体系。对于前端，我们还能够依靠代码逆向来分析，而对于后面这个大到无形的“魅影”，我们更多的只能依靠分析恶意代码的指令体系、有限的网络感知捕获、少得可怜的可公开检索信息，以及作为信息安全动物的逻辑和本能来进行猜测。我们所要完成的工作，不仅仅是要逐一进行载荷模块分析，而是要对整个攻击面和攻击链的深入复盘。

2017 年 1 月 25 日，安天分析小组更新了针对方程式攻击组织的第四篇分析报告《方程式组织 EQUATION DRUG 平台解析》，基于对方程式组织大量恶意代码模块的分析，我们初步完成了一份有价值的工作，那就是方程式组织的主机作业模块积木拼图。


方程式组织主机作业模块积木图
2017.1.24 安天根据分析和猜测绘制
初始攻击
传统APT手段
邮件
社交
水坑
IM
远程漏洞
介质投放
学术会议CR-ROM光盘和U盘
物流链
ANT中的相关装备
人员带入
黑带行动（Black bag operation）
老鹰哨兵（Sentry Eagle）
创建据点
探测模块
TripleFantasy
DoubleFantasy
攻击插件与平台
EquationDrug
EquationLaser
GrayFish
提权
获取令牌
LSADUMP.dll
DuplicateToken.dll
修改用户组
User.dll
ModifyGroup.dll
Groups.dll
修改安全策略
ModifyPrivilege.dll
modifyAuser.dll
获取权限
EnumAccounts.dll
Query.dll
Recover.dll
内部侦查
获取系统信息
cmlb158w.dll
khlp733w.dll
khlp747w.dll
ProcessInfo.dll
ProcessCheck.dll
machineInfo.dll
IpConfig.dll
FileAttribs.dll
获取网络信息
NetstatMon.dll
IpConfig.dll
Dns.dll
获取其他信息
language.dll
EnvironMent.dll
CheckMouse.dll
CheckKeyboard_dll
共享枚举
khlp807w.dll
NetBios.dll
NetGetDCName.dll
横向移动
端口映射
PortMap.dll
内网渗透
perfcom.dll
ArpScan.dll
arp.dll
网络嗅探
tdip.sys
PacketRedirect.dll
DXGHLP16.SYS
fat32.sys
ldd6.sys
mslcp32.sys
PacketScan.dll
远程执行
RunAsChild.dll
Scheduler.dll
添加账号
User.dll
AdUser.dll
摆渡攻击
agentcpd.dll
shelldoc.dll
持久存在
进程隐藏
ProcessHide.dll
processinfo.dll
网络连接隐藏
nethide.dll
文件隐藏
volrec.sys
scsi2mgr.sys
取证对抗
ClearEventLog.dll
EventLogEdit.dll
ntevt.sys
ntevtx64.sys
固件植入
WIN32M.SYS
nls933w.dll
注册表
RegKeys.dll
RegQuery.dll
进程管理
procMon.dll
kill.dll
ProcessCheck.dll
流量隐藏
atmdkdrv.sys
完成任务
键盘记录
msgkd.ex_
msgki.ex_
msgks.ex_
msgku.ex_
msltyd.sys
wpl913n.dll
下载执行
khlc680w.dll
RemoteExecute.dll
系统密码窃取
pwdump.dll
LSADUMP.dll
数据加密打包
cmlb456w.dll
khlp760w.dll
khlp868w.dll
网络通信回传
wsltcp
m.dll
perfcp
m.dll
vnetap
i.dll
wshapi.dll
vnetapi32.dll
文件窃取
volrec.sys
mrsrtd.sys
mstkpr.dll
网络账号窃取
webmgr.dll
mscorexp32.dll
结束进程
kill.dll
kill2r.dll
图示
以往被发现插件
本批泄露实体文件的插件
本批仅泄露文件名称的插件

我的同事在《方程式组织 EQUATION DRUG 平台解析》中写道：

“这些庞杂的模块展开了一组拼图碎片，每一张图上都有有意义的图案，但如果逐一跟进，这些图案就会组成一个巨大的迷宫。迷宫之所以让人迷惑，不在于其处处是死路，而在于其看起来处处有出口，但所有的砖块都不能给进入者以足够的提示。此时，最大的期待，不只是有一只笔，可以在走过的地方做出标记，而是插上双翼凌空飞起，俯瞰迷宫的全貌，当然这是一种提升分析方法论的自我期待。我辈当下虽身无双翼，但或可练就一点灵犀。”

所有高阶的分析工作，都是以最基础的代码分析为起点的。无论是面对 APT 攻击的深度分析，还是对高级黑产行为的系统挖掘，仅有基本分析固然是远远不够的，但没有最基本的分析是万万不能的。从未来安全工程师所需要的能力来看，熟读本书并不能使你走出迷宫。但不掌握本书相关的基础能力，不能去透视代码迷宫的砖石，就连走入迷宫的资格都没有。也正因为此，本书是安天工程师人手一本的必备手册和必读之物。

肖新光

安天创始人，首席技术架构师，反病毒老兵

前　言

“逆向工程”一词用在软件工程领域的具体含义历来模糊不清。逆向工程是一种分析目标系统的过程，旨在识别系统的各组件以及组件之间的关系，以便通过其他形式或在较高的抽象层次上，重建系统的表征。按照目标系统的分类划分，人们常说的“逆向工程”大体可分为：

（1）软件逆向工程。研究编译后的可执行程序。

（2）建模逆向工程。扫描 3D 结构并进行后续数据处理，以便重现原物。

（3）重建 DBMS 结构。

本书仅涉及上述第一项，即软件逆向工程的知识范畴。

重点议题

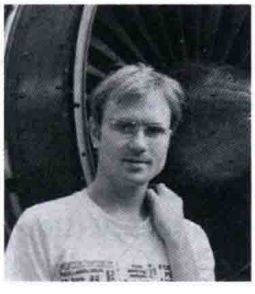
x86/x64、ARM/ARM64、MIPS、Java/JVM。

涉及话题

本书中涉及如下一些话题：

Oracle RDBMS（第 81 章）、Itanium（IA64，第 93 章）、加密狗（第 78 章）、LD_PRELOAD（67.2 节）、栈溢出、ELF、Win32 PE 文件格式（68.2 节）、x86-64（26.1 节）、critical sections（68.4 节）、syscalls（第 66 章），线程本地存储 TLS，地址无关代码 PIC（67.1 节）、以配置文件为导向的优化（95.1 节）、C++ STL（51.4 节）、OpenMP（第 92 章），SEH（68.3 节）。

作者简介



姓名：Dennis Yurichev
特长：逆向工程及计算机编程

联系方式：E-mail dennis(a)yurichev.com
Skype dennis.yurichev

译者简介

安天安全研究与应急处理中心（安天 CERT），是承担安天安全威胁应急处理、恶意代码分析、APT 攻

击分析取证等方面工作的综合研究和服务的部门，由资深安全工程师团队组成。安天 CERT 以“第一时间启动，同时应对两线严重威胁”为自身能力建设目标，在红色代码 II、口令蠕虫、震荡波、冲击波等重大安全疫情的响应中，提供了先发预警、深度分析和解决方案；并在 2010 年后针对震网、毒曲、白象、方程式等 APT 攻击组织和行动，进行了深入的跟踪分析。分析成果有效推动了安天核心引擎和产品能力的成长，获得了主管部门和用户的好评。

致谢

感谢耐心回答我提问的 Andrey “herm1t” Baranovich 和 Slava “Avid” Kazakov。

感谢帮助我勘误的 Stanislav “Beaver” Bobrytskyy、Alexander Lysenko、Shell Rocket、Zhu Ruijin 和 Changmin Heo。

感谢 Andrew Zubinski、Arnaud Patard (rtp on #debian-arm IRC)和 Aliaksandr Autayeu 的鼎力支持。

感谢本书的中文版翻译 Archer 和安天安全研究与应急处理中心。

感谢本书的韩语版翻译 Byungho Min。

感谢校对人员 Alexander “Lstar” Chernenkiy、Vladimir Botov、Andrei Brazhuk、Mark “Logxen” Cooper、Yuan Jochen Kang、Mal Malakov、Lewis Porter 和 Jarle Thorsen。

特别感谢承担最多校对工作的，同时也是补救最多纰漏的朋友 Vasil Kolev。

感谢封面设计 Andy Nechaevsky。

感谢 github.com 的朋友，谢谢他们所发的各种资料以及勘误。

本书使用了 LATEX 的多种工具。在此，我向它们的作者表示敬意。

捐赠

希望鼓励作者继续创作的读者，通过下述网站进行捐赠：

Dennis Yurichevdonate.html

为了表示感谢，每位捐赠者都会获得题名。此外，捐赠者感兴趣的内容将会被优先更新或补充。

捐赠人名录

25 * anonymous, 2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), PawelSzczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), AngeAlbertini (10+50 EUR), Sergey Lukianov (300 RUR), LudvigGislason (200 SEK), Gérard Labadie (40 EUR), Sergey Volchkov (10 AUD), VankayalaVigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (5 EUR), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), BaynaAlZaabi ($75), Redfive B.V. (25 EUR), JoonaOskariHeikkilä (5 EUR), Marshall Bishop ($50), Nicolas Werner (12 EUR), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (50 EUR), Jiarui Hong (100.00 SEK), Jim_Di (500 RUR), Tan Vincent ($30), Sri HarshaKandrakota (10 AUD), Pillay Harish (10 SGD), TimurValiev (230 RUR), Carlos Garcia Prado (10 EUR), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (20 EUR), Hans-Martin Münch (15 EUR), JarleThorsen (100 NOK), VitalyOsipov ($100)。

FAQ

Q：学习汇编语言有何用武之地？

A：除去开发操作系统的研发人员之外，现在几乎没有什么人还要用汇编语言编写程序了。目前，编译程序优化汇编指令的水平已经超越编程人员的脑算水平[①]，而且 CPU 越来越复杂——即使一个人具备丰富的汇编语言的知识，也不代表他有多么了解计算机硬件。但是不可否认的是，汇编语言的知识至少有两大用处：首先，它有助于安全人员进行安全研究、分析恶意软件；其次，它还有助于帮助编程人员调试程序。本书旨在帮助人们理解汇编语言，而不是要指导读者用汇编语言进行编程。所以，作者组织了大量的源代码和对应的汇编指令，供读者研究。

Q：这本书太厚了，有没有精简版？

A：精简版可从网上下载：http://beginners.re/#lite。

Q：逆向工程方面的就业情况如何？

A：在 reddit 等著名网站里，很多著名公司一直在招聘熟悉汇编语言和逆向工程的 IT 专家，甚至专门招聘逆向工程领域的安全专家。有兴趣的读者可以访问以下网址：

http://www.reddit.com/r/ReverseEngineering/

http://www.reddit.com/r/netsec/comments/221xxu/rnetsecs_q2_2014_information_security_hiring

Q：我想要提些问题……

A：作者的邮件地址是：dennis(a)yurichev.com。

读者还可以在我们的网站 forum.yurichev.com 参与互动。

读者点评

- “构思精巧……而且免费……确实不错”——Daniel Bilar, Siege Technologies, LLC。
- “……了不起的免费读物！”——Pete Finnigan, Oracle RDBMS security guru。
- “……引人入胜，值得一读！”——Michael Sikorski,《Practical Malware Analysis: The Hands-On Guide to DissectingMalicious Software》的作者。
- “……谨向这本出色的教材致以个人的敬意！”——Herbert Bos，阿姆斯特丹自由大学教授，《Modern Operating Systems（4th Edition）》作者之一。
- “……令人惊讶、难以置信”。——Luis Rocha, CISSP / ISSAP，技术经理，Verizon 业务部网络及信息安全中心。
- “感谢如此辛劳的作者、感谢如此精彩的书。”——Joris van de Vis, SAP 集成平台及安全专员。
- “……部分内容可圈可点。”——Mike Stay，教员，联邦执法训练中心，Georgia, US。
- “这本书超赞！我的数名学生都在学习这本书，我计划把它当作研究生教材。”——Sergey Bratus，计算机科学系研究助理教授，（美）达特茅斯学院。
- “Dennis Yurichev 发表了一本逆向工程方面的宝典（还免费）！”TanelPoder, Oracle RDBMS 性能调控专家。
- “这本书可谓初学者的百科全书……”——Archer，中文译者，IT 安全研究员。

[①] 可查阅参考文献[Fog13b]。

目　录

第一部分　指令讲解

第 1 章　CPU 简介……3

1.1　指令集架构……3

第 2 章　最简函数……5

2.1　x86……5
2.2　ARM……5
2.3　MIPS……5
MIPS 指令集与寄存器名称……6

第 3 章　Hello，world！……7

3.1　x86……7
3.1.1　MSVC……7
3.1.2　GCC……9
3.1.3　GCC:AT&T 语体……9
3.2　x86-64……11
3.2.1　MSVC-x86-64……11
3.2.2　GCC－x86-64……12
3.3　GCC 的其他特性……12
3.4　ARM……13
3.4.1　Keil　6/2013 —— 未启用优化功能的 ARM 模式……14
3.4.2　Thumb 模式下、未开启优化选项的 Keil……15
3.4.3　ARM 模式下、开启优化选项的 Xcode……15
3.4.4　Thumb-2 模式下、开启优化选项的 Xcode（LLVM）……16
3.4.5　ARM64……18
3.5　MIPS……19
3.5.1　全局指针 Global pointer……19
3.5.2　Optimizing GCC……19
3.5.3　Non-optimizing GCC……21
3.5.4　栈帧……23
3.5.5　Optimizing　GCC:　GDB　的分析方法……23
3.6　总结……24
3.7　练习题……24
3.7.1　题目 1……24
3.7.2　题目 2……24

第 4 章　函数序言和函数尾声……25

递归调用……25

第 5 章　栈……26

5.1　为什么栈会逆增长……26
5.2　栈的用途……27
5.2.1　保存函数结束时的返回地址……27
5.2.2　参数传递……28
5.2.3　存储局部变量……29
5.2.4　x86:alloca()函数……29
5.2.5　（Windows）SEH 结构化异常处理……31
5.2.6　缓冲区溢出保护……31
5.3　典型的栈的内存存储格式……31
5.4　栈的噪音……31
5.5　练习题……34
5.5.1　题目 1……34
5.5.2　题目 2……34

第 6 章　printf()函数与参数调用……36

6.1　x86……36
6.1.1　x86:　传递 3 个参数……36
6.1.2　x64:　传递 9 个参数……41
6.2　ARM……44
6.2.1　ARM 模式下传递 3 个参数……44
6.2.2　ARM 模式下传递 8 个参数……46
6.3　MIPS……50
6.3.1　传递 3 个参数……50
6.3.2　传递 9 个参数……52
6.4　总结……56
6.5　其他……57

第 7 章　scanf()……58

7.1　演示案例……58
7.1.1　指针简介……58
7.1.2　x86……58
7.1.3　MSVC + OllyDbg……60
7.1.4　x64……62
7.1.5　ARM……63
7.1.6　MIPS……64
7.2　全局变量……65

7.2.1 MSVC: x86 ····· 66
7.2.2 MSVC: x86+OllyDbg ····· 67
7.2.3 GCC: x86 ····· 68
7.2.4 MSVC: x64 ····· 68
7.2.5 ARM: Optimizing Keil 6/2013 (Thumb 模式) ····· 69
7.2.6 ARM64 ····· 70
7.2.7 MIPS ····· 70
7.3 scanf()函数的状态监测 ····· 74
7.3.1 MSVC: x86 ····· 74
7.3.2 MSVC: x86: IDA ····· 75
7.3.3 MSVC: x86+OllyDbg ····· 77
7.3.4 MSVC: x86+Hiew ····· 78
7.3.5 MSVC:x64 ····· 79
7.3.6 ARM ····· 80
7.3.7 MIPS ····· 81
7.3.8 练习题 ····· 82
7.4 练习题 ····· 82
题目 1 ····· 82

第 8 章 参数获取 ····· 83

8.1 x86 ····· 83
8.1.1 MSVC ····· 83
8.1.2 MSVC+OllyDbg ····· 84
8.1.3 GCC ····· 84
8.2 x64 ····· 85
8.2.1 MSVC ····· 85
8.2.2 GCC ····· 86
8.2.3 GCC: uint64_t 型参数 ····· 87
8.3 ARM ····· 88
8.3.1 Non-optimizing Keil 6/2013 (ARM mode) ····· 88
8.3.2 Optimizing Keil 6/2013 (ARM mode) ····· 89
8.3.3 Optimizing Keil 6/2013 (Thumb mode) ····· 89
8.3.4 ARM64 ····· 89
8.4 MIPS ····· 91

第 9 章 返回值 ····· 93

9.1 void 型函数的返回值 ····· 93
9.2 函数返回值不被调用的情况 ····· 94
9.3 返回值为结构体型数据 ····· 94

第 10 章 指针 ····· 96

10.1 全局变量 ····· 96
10.2 局部变量 ····· 98
10.3 总结 ····· 100

第 11 章 GOTO 语句 ····· 101

11.1 无用代码 Dead Code ····· 102
11.2 练习题 ····· 102

第 12 章 条件转移指令 ····· 103

12.1 数值比较 ····· 103
12.1.1 x86 ····· 103
12.1.2 ARM ····· 109
12.1.3 MIPS ····· 112
12.2 计算绝对值 ····· 115
12.2.1 Optimizing MSVC ····· 115
12.2.2 Optimizing Keil 6/2013: Thumb mode ····· 116
12.2.3 Optimizing Keil 6/2013: ARM mode ····· 116
12.2.4 Non-optimizng GCC 4.9 (ARM64) ····· 116
12.2.5 MIPS ····· 117
12.2.6 不使用转移指令 ····· 117
12.3 条件运算符 ····· 117
12.3.1 x86 ····· 117
12.3.2 ARM ····· 118
12.3.3 ARM64 ····· 119
12.3.4 MIPS ····· 119
12.3.5 使用 if/else 替代条件运算符 ····· 120
12.3.6 总结 ····· 120
12.4 比较最大值和最小值 ····· 120
12.4.1 32 位 ····· 120
12.4.2 64 位 ····· 123
12.4.3 MIPS ····· 125
12.5 总结 ····· 125
12.5.1 x86 ····· 125
12.5.2 ARM ····· 125
12.5.3 MIPS ····· 126
12.5.4 无分支指令（非条件指令） ····· 126
12.6 练习题 ····· 127

第 13 章 switch()/case/default ····· 128

13.1 case 陈述式较少的情况 ····· 128
13.1.1 x86 ····· 128

13.1.2 ARM: Optimizing Keil 6/2013 (ARM mode) ····· 133
13.1.3 ARM: Optimizing Keil 6/2013 (Thumb mode) ····· 133
13.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9 ····· 134
13.1.5 ARM64: Optimizing GCC (Linaro) 4.9 ····· 134
13.1.6 MIPS ····· 135
13.1.7 总结 ····· 136
13.2 case 陈述式较多的情况 ····· 136
13.2.1 x86 ····· 136
13.2.2 ARM: Optimizing Keil 6/2013 (ARM mode) ····· 140
13.2.3 ARM: Optimizing Keil 6/2013 (Thumb mode) ····· 141
13.2.4 MIPS ····· 143
13.2.5 总结 ····· 144
13.3 case 从句多对一的情况 ····· 145
13.3.1 MSVC ····· 145
13.3.2 GCC ····· 147
13.3.3 ARM64: Optimizing GCC 4.9.1 ····· 147
13.4 Fall-through ····· 149
13.4.1 MSVC x86 ····· 149
13.4.2 ARM64 ····· 150
13.5 练习题 ····· 151
13.5.1 题目 1 ····· 151

第 14 章 循环 ····· 152

14.1 举例说明 ····· 152
14.1.1 x86 ····· 152
14.1.2 x86:OllyDbg ····· 155
14.1.3 x86:跟踪调试工具 tracer ····· 156
14.1.4 ARM ····· 157
14.1.5 MIPS ····· 160
14.1.6 其他 ····· 161
14.2 内存块复制 ····· 161
14.2.1 编译结果 ····· 161
14.2.2 编译为 ARM 模式的程序 ····· 162
14.2.3 MIPS ····· 163
14.2.4 矢量化技术 ····· 164
14.3 总结 ····· 164
14.4 练习题 ····· 165
14.4.1 题目 1 ····· 165
14.4.2 题目 2 ····· 165
14.4.3 题目 3 ····· 166
14.4.4 题目 4 ····· 167

第 15 章 C 语言字符串的函数 ····· 170

15.1 strlen() ····· 170
15.1.1 x86 ····· 170
15.1.2 ARM ····· 174
15.1.3 MIPS ····· 177
15.2 练习题 ····· 178
15.2.1 题目 1 ····· 178

第 16 章 数学计算指令的替换 ····· 181

16.1 乘法 ····· 181
16.1.1 替换为加法运算 ····· 181
16.1.2 替换为位移运算 ····· 181
16.1.3 替换为位移、加减法的混合运算 ····· 182
16.2 除法运算 ····· 186
16.2.1 替换为位移运算 ····· 186
16.3 练习题 ····· 186
16.3.1 题目 1 ····· 186

第 17 章 FPU ····· 188

17.1 IEEE 754 ····· 188
17.2 x86 ····· 188
17.3 ARM、MIPD、x86/x64 SIMD ····· 188
17.4 C/C++ ····· 188
17.5 举例说明 ····· 189
17.5.1 x86 ····· 189
17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode) ····· 193
17.5.3 ARM: Optimizing Keil 6/2013 (Thumb mode) ····· 193
17.5.4 ARM64: Optimizing GCC (Linaro) 4.9 ····· 194
17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9 ····· 195
17.5.6 MIPS ····· 195
17.6 利用参数传递浮点型数据 ····· 196
17.6.1 x86 ····· 196
17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode) ····· 197

17.6.3 ARM + Non-optimizing Keil 6/2013 (ARM mode) ····198
17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9 ····198
17.6.5 MIPS ····199
17.7 比较说明 ····200
17.7.1 x86 ····200
17.7.2 ARM ····216
17.7.3 ARM64 ····219
17.7.4 MIPS ····220
17.8 栈、计算器及逆波兰表示法 ····221
17.9 x64 ····221
17.10 练习题 ····221
17.10.1 题目 1 ····221
17.10.2 题目 2 ····221

第 18 章 数组 ····223

18.1 简介 ····223
18.1.1 x86 ····223
18.1.2 ARM ····225
18.1.3 MIPS ····228
18.2 缓冲区溢出 ····229
18.2.1 读取数组边界以外的内容 ····229
18.2.2 向数组边界之外的地址赋值 ····231
18.3 缓冲区溢出的保护方法 ····234
18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode) ····236
18.4 其他 ····238
18.5 字符串指针 ····238
18.5.1 x64 ····239
18.5.2 32 位 MSVC ····239
18.5.3 32 位 ARM ····240
18.5.4 ARM64 ····241
18.5.5 MIPS ····242
18.5.6 数组溢出 ····242
18.6 多维数组 ····245
18.6.1 二维数组举例 ····246
18.6.2 以一维数组的方式访问二维数组 ····247
18.6.3 三维数组 ····248
18.6.4 更多案例 ····251
18.7 二维字符串数组的封装格式 ····251
18.7.1 32 位 ARM ····253
18.7.2 ARM64 ····254
18.7.3 MIPS ····254
18.7.4 总结 ····255
18.8 本章小结 ····255
18.9 练习题 ····255
18.9.1 题目 1 ····255
18.9.2 题目 2 ····258
18.9.3 题目 3 ····263
18.9.4 题目 4 ····264
18.9.5 题目 5 ····265

第 19 章 位操作 ····270

19.1 特定位 ····270
19.1.1 x86 ····270
19.1.2 ARM ····272
19.2 设置/清除特定位 ····274
19.2.1 x86 ····274
19.2.2 ARM + Optimizing Keil 6/2013 (ARM mode) ····277
19.2.3 ARM + Optimizing Keil 6/2013 (Thumb mode) ····278
19.2.4 ARM + Optimizing Xcode (LLVM)+ ARM mode ····278
19.2.5 ARM: BIC 指令详解 ····278
19.2.6 ARM64: Optimizing GCC(Linaro) 4.9 ····278
19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9 ····279
19.2.8 MIPS ····279
19.3 位移 ····279
19.4 在 FPU 上设置特定位 ····279
19.4.1 XOR 操作详解 ····280
19.4.2 x86 ····280
19.4.3 MIPS ····282
19.4.4 ARM ····282
19.5 位校验 ····284
19.5.1 x86 ····286
19.5.2 x64 ····289
19.5.3 ARM + Optimizing Xcode 4.6.3 (LLVM) + ARM mode ····291
19.5.4 ARM + Optimizing Xcode 4.6.3 （LLVM） + Thumb-2 mode ····292
19.5.5 ARM64 + Optimizing GCC 4.9 ····292
19.5.6 ARM64 + Non-optimizing GCC 4.9 ····292
19.5.7 MIPS ····293

19.6 本章小结……295
19.6.1 检测特定位（编译阶段）……295
19.6.2 检测特定位（runtime 阶段）…295
19.6.3 设置特定位（编译阶段）……296
19.6.4 设置特定位（runtime 阶段）…296
19.6.5 清除特定位（编译阶段）……296
19.6.6 清除特定位（runtime 阶段）…297
19.7 练习题……297
19.7.1 题目 1……297
19.7.2 题目 2……298
19.7.3 题目 3……301
19.7.4 题目 4……301

第 20 章 线性同余法与伪随机函数……304

20.1 x86……304
20.2 x64……305
20.3 32 位 ARM……306
20.4 MIPS……306
MIPS 的重新定位……307
20.5 本例的线程安全改进版……309

第 21 章 结 构 体……310

21.1 MSVC: systemtime……310
21.1.1 OllyDbg……311
21.1.2 以数组替代结构体……312
21.2 用 malloc()分配结构体的空间……313
21.3 UNIX: struct tm……315
21.3.1 Linux……315
21.3.2 ARM……317
21.3.3 MIPS……319
21.3.4 数组替代法……320
21.3.5 替换为 32 位 words……322
21.3.6 替换为字节型数组……323
21.4 结构体的字段封装……325
21.4.1 x86……325
21.4.2 ARM……329
21.4.3 MIPS……330
21.4.4 其他……331
21.5 结构体的嵌套……331
OllyDbg……332
21.6 结构体中的位操作……333
21.6.1 CPUID……333
21.6.2 用结构体构建浮点数……337
21.7 练习题……339
21.7.1 题目 1……339
21.7.2 题目 2……340

第 22 章 共用体（union）类型……345

22.1 伪随机数生成程序……345
22.1.1 x86……346
22.1.2 MIPS……347
22.1.3 ARM (ARM mode)……348
22.2 计算机器精度……349
22.2.1 x86……350
22.2.2 ARM64……350
22.2.3 MIPS……351
22.2.4 本章小结……351
22.3 快速平方根计算……351

第 23 章 函数指针……352

23.1 MSVC……353
23.1.1 MSVC+OllyDbg……354
23.1.2 MSVC+tracer……355
23.1.3 MSVC + tracer（指令分析）……356
23.2 GCC……357
23.2.1 GCC + GDB（有源代码的情况）……358
23.2.2 GCC+GDB（没有源代码的情况）……359

第 24 章 32 位系统处理 64 位数据……362

24.1 64 位返回值……362
24.1.1 x86……362
24.1.2 ARM……362
24.1.3 MIPS……362
24.2 参数传递及加减运算……363
24.2.1 x86……363
24.2.2 ARM……365
24.2.3 MIPS……365
24.3 乘法和除法运算……366
24.3.1 x86……367
24.3.2 ARM……368
24.3.3 MIPS……369
24.4 右移……370
24.4.1 x86……370
24.4.2 ARM……371
24.4.3 MIPS……371
24.5 32 位数据转换为 64 位数据……371
24.5.1 x86……372
24.5.2 ARM……372

24.5.3 MIPS ····372

第 25 章 SIMD ····373

25.1 矢量化····373
25.1.1 用于加法计算····374
25.1.2 用于内存复制····379
25.2 SIMD 实现 strlen() ····383

第 26 章 64 位平台 ····387

26.1 x86-64 ····387
26.2 ARM····394
26.3 浮点数····394

第 27 章 SIMD 与浮点数的并行运算 ···395

27.1 样板程序····395
27.1.1 x64····395
27.1.2 x86····396
27.2 传递浮点型参数····399
27.3 浮点数之间的比较····400
27.3.1 x64····400
27.3.2 x86····401
27.4 机器精····402
27.5 伪随机数生成程序（续） ····402
27.6 总结····403

第 28 章 ARM 指令详解 ····404

28.1 立即数标识（#） ····404
28.2 变址寻址····404
28.3 常量赋值····405
28.3.1 32 位 ARM····405
28.3.2 ARM64····405
28.4 重定位····406

第 29 章 MIPS 的特点 ····409

29.1 加载常量····409
29.2 阅读推荐····409

第二部分 硬件基础

第 30 章 有符号数的表示方法 ····413

第 31 章 字节序 ····415

31.1 大端字节序····415
31.2 小端字节序····415
31.3 举例说明····415
31.4 双模二元数据格式····416
31.5 转换字节序····416

第 32 章 内存布局 ····417

第 33 章 CPU ····418

33.1 分支预测····418
33.2 数据相关性····418

第 34 章 哈希函数 ····419

单向函数与不可逆算法····419

第三部分 一些高级的例子

第 35 章 温度转换 ····423

35.1 整数值····423
35.1.1 x86 构架下 MSVC 2012 优化····423
35.1.2 x64 构架下的 MSVC 2012 优化····425
35.2 浮点数运算····425

第 36 章 斐波拉契数列 ····428

36.1 例子 1····428
36.2 例子 2····430
36.3 总结····433

第 37 章 CRC32 计算的例子 ····434

第 38 章 网络地址计算实例 ····437

38.1 计算网络地址函数 calc_network_address()····438
38.2 函数 form_IP()····439
38.3 函数 print_as_IP()····440
38.4 form_netmask()函数和 set_bit()函数····442
38.5 总结····442

第 39 章 循环：几个迭代 ····444

39.1 三个迭代器····444
39.2 两个迭代器····445
39.3 Intel C++ 2011 实例 ····446

第 40 章 达夫装置 ····449

第 41 章 除以 9 ····452

41.1 x86····452

41.2 ARM……453
41.2.1 ARM 模式下，采用 Xcode 4.6.3（LLVM）优化……453
41.2.2 Thumb-2 模式下的 Xcode 4.6.3 优化（LLVM）……454
41.2.3 非优化的 Xcode 4.6.3(LLVM) 以及 Keil 6/2013……454
41.3 MIPS……454
41.4 它是如何工作的……455
41.4.1 更多的理论……456
41.5 计算除数……456
41.5.1 变位系数#1……456
41.5.2 变位系数#2……457
41.6 练习题……458

第 42 章 字符串转换成数字，函数 atoi()……459

42.1 例 1……459
42.1.1 64 位下的 MSVC 2013 优化……459
42.1.2 64 位下的 GCC 4.9.1 优化……460
42.1.3 ARM 模式下 Keil 6/2013 优化……460
42.1.4 Thumb 模式下 Keil 6/2013 优化……461
42.1.5 ARM64 下的 GCC 4.9.1 优化……462
42.2 例 2……462
42.2.1 64 位下的 GCC 4.9.1 优化……463
42.2.2 ARM 模式下的 Keil6/2013 优化……464
42.3 练习题……465

第 43 章 内联函数……466

43.1 字符串和内存操作函数……467
43.1.1 字符串比较函数 strcmp()……467
43.1.2 字符串长度函数 strlen()……469
43.1.3 字符串复制函数 strcpy()……469
43.1.4 内存设置函数 memset()……470
43.1.5 内存复制函数 memcpy()……471
43.1.6 内存对比函数 memcmp()……473
43.1.7 IDA 脚本……474

第 44 章 C99 标准的受限指针……475

第 45 章 打造无分支的 abs()函数……478

45.1 x64 下的 GCC 4.9.1 优化……478
45.2 ARM64 下的 GCC 4.9 优化……478

第 46 章 变长参数函数……480

46.1 计算算术平均值……480
46.1.1 cdecl 调用规范……480
46.1.2 基于寄存器的调用规范……481
46.2 vprintf()函数例子……483

第一部分

指令讲解

在最初接触 C/C++时，我就对程序编译后的汇编指令十分着迷。按照从易到难的顺序，我循序渐进地研究了 C/C++语言编译器生成汇编指令的模式。经过日积月累的努力，现在我不仅可以直接阅读 x86 程序的汇编代码，而且能够在脑海里将其还原成原始的 C/C++语句。我相信这是学习逆向工程的有效方法。为了能够帮助他人进行相关研究，我把个人经验整理成册，以待与读者分享。

本书包含大量 x86/x64 和 ARM 框架的范例。如果读者熟悉其中某一种框架，可以跳过相关的篇幅。

第 1 章　CPU 简介

CPU 是执行程序机器码的硬件单元。简要地说，其相关概念主要有以下几项。

指令码：CPU 受理的底层命令。典型的底层命令有：将数据在寄存器间转移、操作内存、计算运算等指令。每类 CPU 都有自己的指令集架构（Instruction Set Architecture，ISA）。

机器码：发送给 CPU 的程序代码。一条指令通常被封装为若干字节。

汇编语言：为了让程序员少长白头发而创造出来的、易读易记的代码，它有很多类似宏的扩展功能。

CPU 寄存器：每种 CPU 都有其固定的通用寄存器（GPR）。x86 CPU 里一般有 8 个 GPR，x64 里往往有 16 个 GPR，而 ARM 里则通常有 16 个 GPR。您可以认为 CPU 寄存器是一种存储单元，它能够无差别地存储所有类型的临时变量。假如您使用一种高级的编程语言，且仅会使用到 8 个 32 位变量，那么光 CPU 自带的寄存器就能完成不少任务了！

那么，机器码和编程语言（PL）的区别在哪里？CPU 可不像人类那样，能够理解 C/C++、Java、Python 这类较为贴近人类语言的高级编程语言。CPU 更适合接近硬件底层的具体指令。在不久的将来，或许会出现直接执行高级编程语言的 CPU，不过那种尚未问世的科幻 CPU 必定比现在的 CPU 复杂。人脑和计算机各有所长，如果人类直接使用贴近硬件底层的汇编语言编写程序，其难度也很高——因为那样很容易出现大量的人为失误。可见，我们需要用一种程序把高级的编程语言转换为 CPU 能受理的底层汇编语言，而这种程序就是人们常说的编译器/Compiler。

1.1　指令集架构

在 x86 的指令集架构（ISA）里，各 opcode（汇编指令对应的机器码）的长度不尽相同。出于兼容性的考虑，后来问世的 64 位 CPU 指令集架构也没有大刀阔斧地摒弃原有指令集架构。很多面向早期 16 位 8086 CPU 的指令，不仅被 x86 的指令集继承，而且被当前最新的 CPU 指令集继续沿用。

ARM 属于 RISC[①] CPU，它的指令集在设计之初就力图保持各 opcode 的长度一致。在过去，这一特性的确表现出了自身的优越性。最初的时候，所有 ARM 指令的机器码都被封装在 4 个字节里[②]。人们把这种运行模式叫作“ARM 模式”。

不久，他们就发现这种模式并不划算。在实际的应用程序中，绝大多数的 CPU 指令[③]很少用满那 4 个字节。所以他们又推出了一种把每条指令封装在 2 个字节的“Thumb”模式的指令集架构。人们把采用这种指令集编码的指令叫作“Thumb 模式”指令。然而 Thumb 指令集并不能够封装所有的 ARM 指令，它本身存在指令上的局限。当然，在同一个程序里可以同时存在 ARM 模式和 Thumb 模式这两种指令。

之后，ARM 的缔造者们决定扩充 Thumb 指令集。他们自 ARM v7 平台开始推出了 Thumb-2 指令集。Thumb-2 指令基本都可封装在 2 个字节的机器码之中，2 个字节封装不下的指令则由 4 字节封装。现在，多数人依然错误地认为“Thumb-2 指令集是 ARM 指令集和 Thumb 指令集的复合体”。实际上，它是一种充分利用处理器性能、足以与 ARM 模式媲美的独立的运行模式。在扩展了 Thumb 模式的指令集之后，Thumb-2 现在与 ARM 模式不相上下。由于 Xcode 编译器默认采用 Thumb-2 指令集编译，所以现在主流的 iPod/iPhone/iPad 应用程序都采用了 Thumb-2 指令集。

---

① Reduced instruction computing /精简指令集。

② 这种固定长度的指令集，特别便于计算前后指令的地址。有关特性将在 13.2.2 节进行介绍。

③ 即 MOV/PUSH/CALL/Jcc 等指令。

64 位的 ARM 处理器接踵而至。这种 CPU 的指令集架构再次使用固定长度的 4 字节 opcode，所以不再支持 Thumb 模式的指令。相应地，64 位 ARM 工作于自己的指令集。受到指令集架构的影响，ARM 指令集分为 3 类：ARM 模式指令集、Thumb 模式指令集（包括 Thumb-2）和 ARM64 的指令集。虽然这些指令集之间有着千丝万缕的联系，需要强调的是：不同的指令集分别属于不同的指令集架构；一个指令集绝非另一个指令集的变种。相应地，本书会以 3 种指令集、重复演示同一程序的指令片段，充分介绍 ARM 应用程序的特点。

除了 ARM 处理器之外，还有许多处理器都采用了精简指令集。这些处理器多数都使用了固定长度的 32 位 opcode。例如 MIPS、PowerPC 和 Alpha AXP 处理器就是如此。

第2章 最简函数

返回预定常量的函数，已经算得上是最简单的函数了。

本章围绕下列函数进行演示：

指令清单 2.1 C/C++ 代码

```
int f()
{
        return 123;
};
```

2.1 x86

在开启优化功能之后，GCC 编译器产生的汇编指令，如下所示。

指令清单 2.2 Optimizing GCC/MSVC（汇编输出）

```
f:
        mov   eax, 123
        ret
```

MSVC 编译的程序和上述指令完全一致。

这个函数仅由两条指令构成：第一条指令把数值 123 存放在 EAX 寄存器里；根据函数调用约定[①]，后面一条指令会把 EAX 的值当作返回值传递给调用者函数，而调用者函数（caller）会从 EAX 寄存器里取值，把它当作返回结果。

2.2 ARM

ARM 模式是什么情况？

指令清单 2.3 Optimizing Keil 6/2013 (ARM 模式)

```
f PROC
        MOV    r0,#0x7b ; 123
        BX     lr
        ENDP
```

ARM 程序使用 R0 寄存器传递函数返回值，所以指令把数值 123 赋值给 R0。

ARM 程序使用 LR 寄存器（Link Register）存储函数结束之后的返回地址（RA/ Return Address）。x86 程序使用“栈”结构存储上述返回地址。可见，BX LR 指令的作用是跳转到返回地址，即返回到调用者函数，然后继续执行调用体 caller 的后续指令。

如您所见，x86 和 ARM 指令集的 MOV 指令确实和对应单词“move”没有什么瓜葛。它的作用是复制（copy），而非移动（move）。

2.3 MIPS

在 MIPS 指令里，寄存器有两种命名方式。一种是以数字命名（$0~$31），另一种则是以伪名称（pseudoname）

① Calling Convention，又称为函数的调用协定、调用规范。

命名（$V0~VA0，依此类推）。在 GCC 编译器生成的汇编指令中，寄存器都采用数字方式命名。

指令清单 2.4 Optimizing GCC 4.4.5（汇编输出）

```
j       $31
li      $2,123          # 0x7b
```

然而 IDA 则会显示寄存器的伪名称。

指令清单 2.5 Optimizing GCC 4.4.5（IDA）

```
        jr      $ra
        li      $v0, 0x7B
```

根据伪名称和寄存器数字编号的关系可知，存储函数返回值的寄存器都是$2（即$V0）。此处 LI 指令是英文词组“Load Immediate（加载立即数）”的缩写。

其中，J 和 JR 指令都属于跳转指令，它们把执行流递交给调用者函数，跳转到$31 即$RA 寄存器中的地址。这个寄存器相当于的 ARM 平台的 LR 寄存器。

此外，为什么赋值指令 LI 和转移指令 J/JR 的位置反过来了？这属于 RISC 精简指令集的特性之一——分支（转移）指令延迟槽 (Branch delay slot)的现象。简单地说，不管分支（转移）发生与否，位于分支指令后面的一条指令（在延时槽里的指令），总是被先于分支指令提交。这是 RISC 精简指令集的一种特例，我们不必在此处深究。总之，转移指令后面的这条赋值指令实际上是在转移指令之前运行的。

MIPS 指令集与寄存器名称

习惯上，MIPS 领域中的寄存器名称和指令名称都使用小写字母书写。但是为了在排版风格上与其他指令集架构的程序保持一致，本书采用大写字母进行排版。

第 3 章　Hello，world！

现在，我们开始演示《C 语言编程》一书[①]中著名的程序：

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
};
```

3.1　x86

3.1.1　MSVC

接下来我们将通过下述指令，使用 MSVC 2010 编译下面这个程序。

```
cl 1.cpp /Fa1.asm
```

其中/Fa 选项将使编译器生成汇编指令清单文件（assembly listing file），并指定汇编列表文件的文件名称是 1.asm。

上述命令生成的 1.asm 内容如下。

指令清单 3.1　MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC 生成的汇编清单文件都采用了 Intel 语体。汇编语言存在两种主流语体，即 Intel 语体和 AT&T 语体。本书将在 3.1.3 节中讨论它们之间的区别。

在生成 1.asm 之后，编译器会生成 1.obj 再将之链接为可执行文件 1.exe。

在 hello world 这个例子中，文件分为两个代码段，即 CONST 和_TEXT 段，它们分别代表数据段和代码段。在本例中，C/C++程序为字符串常量“Hello，world”分配了一个指针（const char[]），只是在代码中这个指针的名

① Brian W. Kernighan. The C Programming Language. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Tech- nical Reference, 1988. ISBN: 0131103709。

称并不明显（参照下列 Bjarne Stroustrup. The C++ Programming Language, 4th Edition. 2013 的第 176 页，7.3.2 节）。

接下来，编译器进行了自己的处理，并在内部把字符串常量命名为$SG3830。

因此，上述程序的源代码等效于：

```
#include <stdio.h>

const char *$SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

在回顾 1.asm 文件时，我们会发现编译器在字符串常量的尾部添加了十六进制的数字 0，即 00h。依据 C/C++字符串的标准规范，编译器要为这个字符串常量添加结束标志（即数值为零的单个字节）。有关标准请参照本书的 57.1.1 节。

在代码段_TEXT 只有 1 个函数，即主函数 main()。在汇编指令清单里，主函数的函数体有标志性的函数序言（function prologue）和函数尾声（function epilogue）。实际上所有的函数都有这样的序言和尾声。在函数的序言标志之后，我们能够看到调用 printf()函数的指令: CALL _printf。

通过 PUSH 指令，程序把字符串的指针推送入栈。这样，printf()函数就可以调用栈里的指针，即字符串“hello, world!”的地址。

在 printf()函数结束以后，程序的控制流会返回到 main()函数之中。此时，字符串地址（即指针）仍残留在数据栈之中。这个时候就需要调整栈指针（ESP 寄存器里的值）来释放这个指针。

下一条语句是“add ESP，4”，把 ESP 寄存器（栈指针/Stack Pointer）里的数值加 4。

为什么要加上“4”？这是因为 x86 平台的内存地址使用 32 位（即 4 字节）数据描述。同理，在 x64 系统上释放这个指针时，ESP 就要加上 8。

因此，这条指令可以理解为“POP 某寄存器”。只是本例的指令直接舍弃了栈里的数据而 POP 指令还要把寄存器里的值存储到既定寄存器[①]。

某些编译器（如 Intel C++编辑器）不会使用 ADD 指令来释放数据栈，它们可能会用 POP ECX 指令。例如，Oracle RDBMS（由 Intel C++编译器编译）就会用 POP ECX 指令，而不会用 ADD 指令。虽然 POP ECX 命令确实会修改 ECX 寄存器的值，但是它也同样释放了栈空间。

Intel C++编译器使用 POP ECX 指令的另外一个理由就是，POP ECX 对应的 OPCODE（1 字节）比 ADD ESP 的 OPCODE（3 字节）要短。

指令清单 3.2 Oracle RDBMS 10.2 Linux (摘自 app.o)

```
.text:0800029A     push    ebx
.text:0800029B     call    qksfroChild
.text:080002A0     pop     ecx
```

本书将在讨论操作系统的部分详细介绍数据栈。

在上述 C/C++程序里，printf()函数结束之后，main()函数会返回 0（函数正常退出的返回码）。即 main()函数的运算结果是 0。

这个返回值是由指令“XOR EAX, EAX”计算出来的。

顾名思义，XOR 就是“异或”[②]。编译器通常采用异或运算指令，而不会使用“MOV EAX，0”指令。主要是因为异或运算的 opcode 较短（2 字节:5 字节）。

也有一些编译器会使用“SUB EAX,EAX”指令把 EAX 寄存器置零，其中 SUB 代表减法运算。总之，

① 但是 CPU 标志位会发生变化。

② 参见 http://en.wikipedia.org/wiki/Exclusive_or。

main()函数的最后一项任务是使 EAX 的值为零。

汇编列表中最后的操作指令是 RET，将控制权交给调用程序。通常它起到的作用就是将控制权交给操作系统，这部分功能由 C/C++的 CRT[①]实现。

3.1.2 GCC

接下来，我们使用 GCC 4.4.1 编译器编译这个 hello world 程序。

```
gcc 1.c -o 1
```

我们使用反汇编工具 IDA（Interactive Disassembler）查看 main()函数的具体情况。IDA 所输出的汇编指令的格式，与 MSVC 生成的汇编指令的格式相同，它们都采用 Intel 语体显示汇编指令。

此外，如果要让 GCC 编译器生成 Intel 语体的汇编列表文件，可以使用 GCC 的选项“-S-masm=intel”。

指令清单 3.3 在 IDA 中观察到的汇编指令

```
Main        proc near
var_10      = dword ptr -10h

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 10h
            mov     eax, offset aHelloWorld ; "hello, world\n"
            mov     [esp+10h+var_10], eax
            call    _printf
            mov     eax, 0
            leave
            retn
main        endp
```

GCC 生成的汇编指令，与 MSVC 生成的结果基本相同。它首先把“hello, world”字符串在数据段的地址（指针）存储到 EAX 寄存器里，然后再把它存储在数据栈里。

其中值得注意的还有开场部分的“AND ESP, 0FFFFFFF0h”指令。它令栈地址（ESP 的值）向 16 字节边界对齐（成为 16 的整数倍），属于初始化的指令。如果地址位没有对齐，那么 CPU 可能需要访问两次内存才能获得栈内数据。虽然在 8 字节边界处对齐就可以满足 32 位 x86 CPU 和 64 位 x64 CPU 的要求，但是主流编译器的编译规则规定“程序访问的地址必须向 16 字节对齐（被 16 整除）”。人们还是为了提高指令的执行效率而特意拟定了这条编译规范。[②]

“SUB ESP，10h”将在栈中分配 0x10 bytes，即 16 字节。我们在后文看到，程序只会用到 4 字节空间。但是因为编译器对栈地址（ESP）进行了 16 字节对齐，所以每次都会分配 16 字节的空间。

而后，程序将字符串地址（指针的值）直接写入到数据栈。此处，GCC 使用的是 MOV 指令；而 MSVC 生成的是 PUSH 指令。其中 var_10 是局部变量，用来向后面的 printf()函数传递参数。

随即，程序调用 printf()函数。

GCC 和 MSVC 不同，除非人工指定优化选项，否则它会生成与源代码直接对应的“MOV EAX, 0”指令。但是，我们已经知道 MOV 指令的 opcode 肯定要比 XOR 指令的 opcode 长。

最后一条 LEAVE 指令，等效于“MOV ESP, EBP”和“POP EBP”两条指令。可见，这个指令调整了数据栈指针 ESP，并将 EBP 的数值恢复到调用这个函数之前的初始状态。毕竟，程序段在开始部分就对 EBP 和 EBP 进行了操作（MOVEBP, ESP/AND ESP, ...），所以函数要在退出之前恢复这些寄存器的值。

3.1.3 GCC:AT&T 语体

AT&T 语体同样是汇编语言的显示风格。这种语体在 UNIX 之中较为常见。

---

① C runtime library:sec:CRT，参见本书 68.1 节。

② 参考 Wikipedia：Data structure alignment http://en.wikipedia.org/wiki/Data_structure_alignment。

接下来，我们使用 GCC4.7.3 编译如下所示的源程序。

指令清单 3.4 使用 GCC 4.7.3 编译源程序

```
gcc -S 1_1.c
```

上述指令将会得到下述文件。

指令清单 3.5 GCC 4.7.3 生成的汇编指令

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

在上述代码里，由小数点开头的指令就是宏。这种形式的汇编语体大量使用汇编宏，可读性很差。为了便于演示，我们将其中字符串以外的宏忽略不计（也可以启用 GCC 的编译选项-fno-asynchronous-unwind-tables，直接预处理为没有 cfi 宏的汇编指令），将会得到如下指令。

指令清单 3.6 GCC 4.7.3 生成的指令

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

在继续解读这个代码之前，我们先介绍一下 Intel 语体和 AT&T 语体的区别。

- 运算表达式（operands，即运算单元）的书写顺序相反。

  Intel 格式：<指令><目标><源>。

  AT&T 格式：<指令><源><目标>。

  如果您认为 Intel 语体的指令使用等号（=）赋值，那么您可以认为 AT&T 语法结构使用右箭头（→）进行赋值。应当说明的是，这两种格式里，部分 C 标准函数的运算单元的书写格式确实是相同的，

例如 memcpy()、strcpy()。

- AT&T 语体中，在寄存器名称之前使用百分号（%）标记，在立即数之前使用美元符号（$）标记。AT&T 语体使用圆括号，而 Intel 语体则使用方括号。
- AT&T 语体里，每个运算操作符都需要声明操作数据的类型：

  −9−quad（64 位）

  −l 指代 32 位 long 型数据。

  −w 指代 16 位 word 型数据。

  −b 指代 8 位 byte 型数据。
- 其他区别请参考 Sun 公司发布的《x86 Assembly Language Reference Manual》。

现在再来阅读 hello world 的 AT&T 语体指令，就会发现它和 IDA 里看到的指令没有实质区别。有些人可能注意到，用于数据对齐的 0FFFFFFF0h 在这里变成了十进制的$-16——把它们按照 32byte 型数据进行书写后，就会发现两者完全一致。

此外，在退出 main()时,处理 EAX 寄存器的指令是 MOV 指令而不是 XOR 指令。MOV 的作用是给寄存器赋值（load）。某些硬件框架的指令集里有更为直观的“LOAD”“STORE”之类的指令。

3.2　x86-64

3.2.1　MSVC-x86-64

若用 64 位 MSVC 编译上述程序，则会得到下述指令。

指令清单 3.7　MSVC 2012 x64

```
$SG2989  DB      'hello, world', 00H

main     PROC
         sub     rsp, 40
         lea     rcx, OFFSET FLAT:$SG2989
         call    printf
         xor     eax, eax
         add     rsp, 40
         ret     0
main     ENDP
```

在 x86-64 框架的 CPU 里，所有的物理寄存器都被扩展为 64 位寄存器。程序可通过 R-字头的名称直接调用整个 64 位寄存器。为了尽可能充分地利用寄存器、减少访问内存数据的次数，编译器会充分利用寄存器传递函数参数（请参见 64.3 节的 fastcall 约定）。也就是说，编译器会优先使用寄存器传递部分参数，再利用内存（数据栈）传递其余的参数。Win64 的程序还会使用 RCX、RDX、R8、R9 这 4 个寄存器来存放函数参数。我们稍后就会看到这种情况：printf()使用 RCX 寄存器传递参数，而没有像 32 位程序那样使用栈传递数据。

在 x86-64 硬件平台上，寄存器和指针都是 64 位的，存储于 R-字头的寄存器里。但是出于兼容性的考虑，64 位寄存器的低 32 位，也要能够担当 32 位寄存器的角色，才能运行 32 位程序。

在 64 位 x86 兼容的 CPU 中，RAX/EAX/AX/AL 的对应关系如下。

<table>
<tr><td>7th（字节号）</td><td>6th</td><td>5th</td><td>4th</td><td>3rd</td><td>2nd</td><td>1st</td><td>0th</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

main()函数的返回值是整数类型的零，但是出于兼容性和可移植性的考虑，C 语言的编译器仍将使用 32 位的零。换而言之，即使是 64 位的应用程序，在程序结束时 EAX 的值是零，而 RAX 的值不一定会是零。

此时，数据栈的对应空间里仍留有 40 字节的数据。这部分数据空间有个专用的名词，即阴影空间

（shadow space）。本书将在 8.2.1 节里更详细地介绍它。

3.2.2 GCC - x86-64

我们使用 64 位 Linux 的 GCC 编译器编译上述程序，可得到如下所示的指令。

指令清单 3.8 GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world"
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Linux、BSD 和 Mac OS X 系统中的应用程序，会优先使用 RDI、RSI、RDX、RCX、R8、R9 这 6 个寄存器传递函数所需的头 6 个参数，然后使用数据栈传递其余的参数。[①]

因此，64 位的 GCC 编译器使用 EDI 寄存器（寄存器的 32 位）存储字符串指针。EDI 不过是 RDI 寄存器中地址位较低的 32 位地址部分。为何 GCC 不直接使用整个 RDI 寄存器？

需要注意的是，64 位汇编指令 MOV 在写入 R-寄存器的低 32 位地址位的时候，即对 E-寄存器进行写操作的时候，会同时清除 R 寄存器中的高 32 位地址位[②]。所以，“MOV EAX, 011223344h”能够对 RAX 寄存器进行正确的赋值操作，因为该指令会清除（置零）高地址位的内容。

如果打开 GCC 生成的 obj 文件，我们就能看见全部的 opcode。[③]

指令清单 3.9 GCC 4.4.6 x64

```
.text:00000000004004D0                  main  proc near
.text:00000000004004D0 48 83 EC 08      sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00   mov     edi, offset format ; "hello, world\n"
.text:00000000004004D9 31 C0            xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF   call    _printf
.text:00000000004004E0 31 C0            xor     eax, eax
.text:00000000004004E2 48 83 C4 08      add     rsp, 8
.text:00000000004004E6 C3               retn
.text:00000000004004E6                  main  endp
```

在地址 0x4004D4 处，程序对 EDI 进行了写操作，这部分代码的 opcode 占用了 5 个字节；相比之下，对 RDI 进行写操作的 opcode 则会占用 7 个字节。显然，出于空间方面的考虑，GCC 进行了相应的优化处理。此外，因为 32 位地址（指针）能够描述的地址不超过 4GB，我们可据此判断这个程序的数据段地址不会超过 4GB。

在调用 printf()之前，程序清空了 EAX 寄存器，这是 x86-64 框架的系统规范决定的。在系统与应用程序接口的规范中，EAX 寄存器用来保存用过的向量寄存器(vector registers)。[④]

3.3 GCC 的其他特性

只要 C 语言代码里使用了字符串型常量（可参照 3.1.1 节的范例），编译器就会把这个字符串常量置于常量字段，以保证其内容不会发生变化。不过 GCC 有个有趣的特征：它可能会把字符串拆出来单独使用。

我们来看下面这段程序：

```
#include <stdio.h>
int f1()
{
```

① 参考 Mit13。
② 参考 Int13。
③ 可通过菜单“Options Number of opcode bytes”启用有关选项。
④ 请参考 Mit13。

```
        printf ("world\n");
};

int f2()
{
        printf ("hello world\n");
};

int main()
{
        f1();
        f2();
};
```

多数的 C/C++编译器（包括 MSVC 编译器）会分配出两个直接对应的字符串，不过 GCC 4.8.1 的编译结果则更为可圈可点。

指令清单 3.10　在 IDA 中观察 GCC 4.8.1 的汇编指令

```
f1              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset s ; "world\n"
                call    _puts
                add     esp, 1Ch
                retn
f1              endp

f2              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset aHello ; "hello "
                call    _puts
                add     esp, 1Ch
                retn
f2              endp

aHello          db 'hello'
s               db 'world', 0xa, 0
```

在打印字符串“hello world”的时候，这两个词的指针地址实际上是前后相邻的。在调用 puts()函数进行输出时，函数本身不知道它所输出的字符串分为两个部分。实际上我们在汇编指令清单中可以看到，这两个字符串没有被“切实”分开。

在 f1()函数调用 puts()函数时，它输出字符串“world”和外加结束符（数值为零的 1 个字节），因为 puts()函数并不知道字符串可以和前面的字符串连起来形成新的字符串。

GCC 编译器会充分这种技术来节省内存。

3.4　ARM

根据我个人的经验，本书将通过以下几个主流的 ARM 编译器进行演示。

- 2013 年 6 月版本的 Keil 编译器。
- Apple Xcode 4.6.3 IDE (含 LLVM-GCC 4.2 编译器) 。[①]
- 面向 ARM64 的 GCC 4.9 (Linaro)，其 32 位的 Windows 程序可由下述网址下载：http://www.linaro.org/projects/armv8/。

除非特别标注，否则本书中的 ARM 程序都是 32 位 ARM 程序。在介绍 64 位的 ARM 程序时，本书会

① Apple 公司的 Xcode 4.6.3 使用的前段编译器是开源的 GCC 程序，代码生成程序（code generator）使用的是 LLVM。

称其为 ARM64 程序。

3.4.1 Keil 6/2013——未启用优化功能的 ARM 模式

请使用下述指令，用 Keil 编译器把 hello world 程序编译为 ARM 指令集架构的汇编程序：

```
armcc.exe --arm --c90 -O0 1.c
```

虽然 armcc 编译器生成的汇编指令清单同样采用了 Intel 语体，但是程序所使用的宏却极具 ARM 处理器的特色[①]。眼见为实，我们一起用 IDA 来看看它们的本来面目吧。

指令清单 3.11 使用 IDA 观察 Non-optimizing Keil 6/2013 (ARM 模式)

```
.text:00000000                main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF: main+4
```

在本节的例子里，每条指令都占用 4 个字节。正如您所见到，我们确实要把源程序编译为 ARM 模式指令集的应用程序，而不是把它编译为以 Thumb 模式的应用程序。

现在回顾上面的代码，第一句“STMFD SP!, {R4,LR}”[②]相当于 x86 的 PUSH 指令。它把 R4 寄存器和 LR(Link Register)寄存器的数值放到数据栈中。此处，本文的措辞是“相当于”，而非“完全是”。这是因为 ARM 模式的指令集里没有 PUSH 指令，只有 Thumb 模式里的指令集里才有“PUSH/POP”指令。在 IDA 中可以清除地看到这种差别，所以本书推荐使用 IDA 分析上述程序。

这条指令首先将 SP[③]递减，在栈中分配一个新的空间以便存储 R4 和 LR 的值。

STMFD 指令能够一次存储多个寄存器的值，Thumb 模式的 PUSH 指令也可以这样使用。实际上 x86 指令集中并没有这样方便的指令。STMFD 指令可看作是增强版本的 PUSH 指令，它不仅能够存储 SP 的值，也能够存储任何寄存器的值。换句话说，STMFD 可用来在指定的内存空间存储多个寄存器的值。

接下来的指令是“ADR R0, aHelloWorld”。它首先对 PC[④]进行取值操作，然后把“hello, world”字符串的偏移量（可能题值）与 PC 的值相加，将其结果存储到 R0 之中。有些读者可能不明白此处 PC 寄存器的作用。严谨地说，编译器通常帮助 PC 把某些指令强制变为“位置无关代码/position-independent code”。在（多数）操作系统把程序加载在内存里的时候，OS 分配给程序代码的内存地址是不固定的；但是程序内部既定指令和数据常量之间的偏移量是固定的（由二进制程序文件决定）。这种情况下，要在程序内部进行指令寻址（例如跳转等情况），就需要借助 PC 指针[⑤]。ADR 将当前指令的地址与字符串指针地址的差值（偏移量）传递给 R0。程序借助 PC 指针可找到字符串指针的偏移地址，从而使操作系统确定字符串常量在内存里的绝对地址。

“BL __2printf”[⑥]调用 printf()函数。BL 实施的具体操作实际上是：

- 将下一条指令的地址，即地址 0xC 处“MOV R0, #0”的地址，写入 LR 寄存器。
- 然后将 printf()函数的地址写入 PC 寄存器，以引导系统执行该函数。

当 printf()完成工作之后，计算机必须知道返回地址，即它应当从哪里开始继续执行下一条指令。所以，每次使用 BL 指令调用其他函数之前，都要把 BL 指令的下一个指令的地址存储到 LR 寄存器。

这便是 CISC（复杂指令集）处理器与 RISC（精简指令集）处理器在工作模式上的区别。在拥有复杂

① 例如，ARM 模式的指令集里没有 PUSH/POP 指令。
② STMFD 是 Storage Multiple Full Descending 的缩写。
③ stack pointer，栈指针。x86/x64 框架中的 SP 是 SP/ESP/RSP，而 ARM 框架的 SP 就是 SP。
④ Program Counter，中文叫做指令指针或程序计数器。x86/x64 里的 PC 叫作 IP/EIP/RIP，ARM 里它就叫 PC。
⑤ 本书介绍操作系统的部分有更详细的说明。在不同框架的汇编语言中，PC 很少会是当前指令的指针地址+1，这和 CPU 的流水/pipeline 模式有关。如需完整的官方介绍，请参阅 http://www.arm.com/pdfs/comparison-arm7-arm9-v1.pdf。
⑥ BL 是 Branch with Link 的缩写，相当于 x86 的 call 指令。

指令集的 x86 体系里，操作系统可以利用栈存储返回地址。

顺便说一下，ARM 模式跳转指令的寻址能力确实存在局限性。单条 ARM 模式的指令必须是 32 位/4 字节，所以 BL 指令无法调用 32 位绝对地址或 32 位相对地址（容纳不下），它只能编入 24 位的偏移量。不过，既然每条指令的 opcode 必须是 4 字节，则指令地址必须在 4n 处，即偏移地址的最后两位必定为零，可在 opcode 里省略。在处理 ARM 模式的转移指令时，处理器将指令中的 opcode 的低 24 位左移 2 位，形成 26 位偏移量，再进行跳转。由此可知，转移指令 B/BL 的跳转指令的目标地址，大约在当前位置的±32MB 区间之内[①]。

下一条指令“MOV R0，#0”将 R0 寄存器置零。Hello World 的 C 代码中，主函数返回零。该指令把返回值写在 R0 寄存器中。

最后到了“LDMFD SP!, R4,PC”这一条指令[②]。它与 STMFD 成对出现，做的工作相反。它将栈中的数值取出，依次赋值给 R4 和 PC，并且会调整栈指针 SP。可以说这条指令与 POP 指令很相似。main()函数的第一条指令就是 STMFD 指令，它将 R4 寄存器和 LR 寄存器存储在栈中。main()函数在结尾处使用 LDMFD 指令，其作用是把栈里存储的 PC 的值和 R4 寄存器的值恢复回来。

前面提到过，程序在调用其他函数之前，必须把返回地址保存在 LR 寄存器里。因为在调用 printf()函数之后 LR 寄存器的值会发生改变，所以主函数的第一条指令就要负责保存 LR 寄存器的值。在被调用的函数结束后，LR 寄存器中存储的值会被赋值给 PC，以便程序返回调用者函数继续运行。当 C/C++的主函数 main()结束之后，程序的控制权将返回给 OS loader，或者 CRT 中的某个指针，或者作用相似的其他地址。

数据段中的 DCB 是汇编语言中定义 ASCII 字符数组/字节数组的指令，相当于 x86 汇编中的 DB 指令。

3.4.2　Thumb 模式下、未开启优化选项的 Keil

现在以 Thumb 模式编译前面的源代码：

```
armcc.exe --thumb --c90 -O0 1.c
```

我们会在 IDA 中看到如下指令。

指令清单 3.12　使用 IDA 观察 Non-optimizing Keil 6/2013 (Thumb 模式)

```
.text:00000000                 main
.text:00000000 10 B5           PUSH    {R4,LR}
.text:00000002 C0 A0           ADR     R0, aHelloWorld;"hello, world"
.text:00000004 06 F0 2E F9     BL      _2printf
.text:00000008 00 20           MOVS    R0, #0
.text:0000000A 10 BD           POP     {R4, PC}

.text:00000304 68 65 6C 6C  +aHelloWorld  DCB "hello, world",0    ; DATA XREF: main+2
```

Thumb 模式程序的每条指令，都对应着 2 个字节/16 位的 opcode，这是 Thumb 模式程序的特征。但是 Thumb 模式的跳转指令 BL“看上去”占用了 4 个字节的 opcode，实际上它是由 2 条指令组成的。单条 16 位 opcode 传递的信息太有限，不足以向被调用函数传递 PC 和偏移量信息。所以，上面 BL 指令分为 2 条 16 位 opcode。第一条 16 位指令可以传递偏移量的高 10 位，第二条指令可以传递偏移量的低 11 位。而 Thumb 模式的 opcode 都是固定的 2 个字节长，目标地址位最后一个位必定是 0（Thumb 模式的 opcode 的启始地址位必须是 2n），因而会被省略。在执行 Thumb 模式的转移指令时，处理器会将目标地址左移 1 位，形成 22 位的偏移量。即 Thumb 的 BL 跳转指令将无法跳到奇数地址，而且跳转指令仅仅能偏移到到当前地址 ±2MB（22 位有符号整数的取值区间）附近的范围之内。

程序主函数的其他指令，PUSH 和 POP 工作方式与 STMFD/LDMFD 相似。虽然表面上看不出来，但是实际上它们也会调整 SP 指针。ADR 指令与前文的作用相同。而 MOVS 指令负责把返回值（R0 寄存器）置零。

3.4.3　ARM 模式下、开启优化选项的 Xcode

如果不启用优化选项，Xcode 4.6.3 将会产生大量的冗余代码，所以不妨开启优化选项，让其生成最优

① 这是二进制里 26 位有符号整型数据（26 bits signed int）的数值范围。

② LDMFD 是 Load Multiple Full Descending 的缩写。

的代码。请指定编译选项-O3，使用 Xcode（启用优化选项-O3）编译 Hello world 程序。这将会得到如下所示的汇编代码。

指令清单 3.13 Optimizing Xcode 4.6.3 (LLVM) (ARM 模式)

```
__text:000028C4                 _helloworld
__text:000028C4 80 40 2D E9     STMFD           SP!, {R7, LR}
__text:000028C8 86 06 01 E3     MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1     MOV             R7, SP
__text:000028D0 00 00 40 E3     MOVT            R0, #0
__text:000028D4 00 00 8F E0     ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB     BL              _puts
__text:000028DC 00 00 A0 E3     MOV             R0, #0
__text:000028E0 80 80 BD E8     LDMFD           SP!, {R7, PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello World!", 0
```

我们就不再重复介绍 STMFD/LDMFD 指令了。

第一个 MOV 指令将字符串“Hello World!”的偏移量，0x1686 赋值到 R0 寄存器。

根据 Apple ABI 函数接口规范[①]，R7 寄存器担当帧指针（frame pointer）寄存器。

“MOVT R0, #0”将 0 写到 R0 寄存器的高 16 位地址。在 ARM 模式里，常规的 MOV 指令只能操作寄存器的低 16 位地址，而单条 ARM 指令最多是 32 位/4 字节。当然，寄存器之间传递数据没有这种限制。所以，对寄存器的高位（第 16 位到第 31 位）进行赋值操作的 MOVT 指令应运而生。然而此处的这条 MOVT 指令可有可无，因为在执行下一条指令“MOV R0, #0x1686”时，R0 寄存器的高 16 位本来就会被清零。这或许就是编译器智能方面的缺陷吧。

“ADD R0，PC，R0”将 PC 和 R0 进行求和，计算得出字符串的绝对地址。前文介绍过了“位置无关代码”，我们知道程序运行之后的启始地址并不固定。此处，程序对这个地址进行了必要的修正。

然后，程序通过 BL 指令调用 puts()函数，而没有像前文那样调用 printf()函数。这种差异来自于 GCC 编译器[②]，编译器将第一个 printf()函数替换为 puts()函数（这两个函数的作用几乎相同）。

所谓“几乎”就意味着它们还存在差别事实上，如 printf()函数支持“%”开头的控制符，而 puts()函数则不支持这类格式化字符串。如果参数里有这类控制符，那么这两个函数的输出结果还会不同。

为什么 GCC 编译器会做这种替换？大概是由于这种情况下 puts()的效率更高吧。由于 puts()函数不处理控制符（%）、只是把各个字符输出到 stdout 设备上，所以 puts()函数的运行速度更快[③]。

后面的“MOV R0, #0”指令将 R0 寄存器置零。

3.4.4 Thumb-2 模式下、开启优化选项的 Xcode（LLVM）

默认情况下，Xcode 4.6.3 会启用优化模式，并以 Thumb-2 模式编译源程序。

指令清单 3.14 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 模式)

```
__text:00002B6C                 _hello_world
__text:00002B6C 80 B5           PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30     MOVW            R0, #0x13D8
__text:00002B72 6F 46           MOV             R7, SP
__text:00002B74 C0 F2 00 00     MOVT.W          R0, #0
__text:00002B78 78 44           ADD             R0, PC
__text:00002B7A 01 F0 38 EA     BLX             _puts
__text:00002B7E 00 20           MOVS            R0, #0
__text:00002B80 80 BD           POP             {R7, PC}
...
__cstring:00003E70 48 65 6C     6F 20+aHelloWorld   DCB "Hello word!",0xA,0
```

① 参照参考文献 App10。

② Xcode 4.6.3 是基于 GCC 的编译器。

③ 请参考 http://www.ciselant.de/projects/gcc_printf/gcc_printf.html。

上文提到过，thumb 模式的 BLX 和 BL 指令以 2 个 16 位指令的形式成对出现的。在 Thumb-2 模式下，BL 和 BLX 指令对应的伪 opcode 有明显的 32 位指令特征，其对应的 opcode 都以 0xFx 或者 0xEx 开头。

在显示 Thumb 和 Thumb-2 模式程序的 opcode 时，IDA 会以两个字节为单位对调。在显示 ARM 模式的指令时，IDA 以字节为单位、依次逆序显示其 opcode。这是字节序的排版差异。

简要地说，在 IDA 显示 ARM 平台的指令时，其显示顺序为：

- ARM 及 ARM64 模式的指令，opcode 以 4-3-2-1 的顺序显示。
- Thumb 模式的指令，opcode 以 2-1 的顺序显示。
- Thumb-2 模式的 16 位指令对，其 opocde 以 2-1-4-3 的顺序显示。

在 IDA 中，我们可观察到上述 MOVW 、MOVT.W、BLX 指令都以 0xFx 开头。

之后的“MOVW R0，#0x13D8”将立即数写到 R0 寄存器的低 16 位地址，同时清除寄存器的高 16 位。

“MOVT.W R0, #0”的作用与前面一个例子中 Thumb 模式的 MOVT 的作用相同，只不过此处是 Thumb-2 的指令。

在这两个例子中，最显著的区别是 Thumb-2 模式“BLX”指令。此处的 BLX 与 Thumb 模式的 BL 指令有着根本的区别。它不仅将 puts()函数的返回地址 RA 存入了 LR 寄存器，将控制权交给了 puts()函数，而且还把处理器从 Thumb/Thumb-2 模式调整为 ARM 模式；它同时也负责在函数退出时把处理器的运行模式进行还原。总之，它同时实现了模式转换和控制权交接的功能，相当于执行了下面的 ARM 模式的指令：

```
__symbolstub1:00003FEC _puts          ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5    LDR PC, =__imp__puts
```

聪明的读者可能会问，此处为什么不直接调用 puts()函数？

直接调用的空间开销更大。

几乎所有的程序都会用到动态链接库，详细说来 Windows 的程序基本上都会用到 DLL 文件、Linux 程序差不多都会用到.SO 文件、MacOSX 系统的程序多数也会用到.dylib 文件。常用的库函数通常都放在动态链接库里。本例用到的标准 C 函数——puts()函数也不例外。

可执行的二进制文件（Windows 的 PE 可执行文件，ELF 或 Mach-O）都有一个输入表段（import section）。输入表段声明了该程序需要通过外部模块和加载的符号链接（函数名称和全局变量），并且含有外部模块的名称等信息。

在操作系统执行二进制文件的时候，它的加载程序（OS loader）会依据这个表段加载程序所需要的模块。在它加载该程序主模块的时候，对导入的符号链接进行枚举，逐一分配符号链接的地址。

在本例中，_imp_puts 是操作系统加载程序（OS loader）为 hello world 程序提供的外部函数地址，属于 32 位变量。程序只需要使用 LDR 指令取出这个变量，并且将它赋值给 PC 寄存器，就可以调用 puts()函数。

可见，一次性地给每个符号链接分配独立的内存地址，可以大幅度地减少 OS loader 在加载方面的耗时。

前文已经指出，如果只能靠单条指令、而不借助内存的读取操作，CPU 就无法把 32 位数值（指针或立即数）赋值给寄存器。所以，可以建立一个以 ARM 模式运行的独立函数，让它专门处理动态链接库的接口问题。此后 Thumb 模式的代码就可以跳转到这个处理接口功能的单指令专用函数。这种专用函数称为（运行模式的）形实转换函数（thunk function）。

前面有一个 ARM 模式的编译例子，它就使用 BL 指令实现相同功能的形实转换函数。但是那个程序使用的指令是 BL 而不是 BLX，可见处理器并没有切换运行模式。

形实转换函数（thunk function）的由来

形实转换函数，是“形参与实参互相转换的函数”的缩写。它不仅是缩写词，而且是外来词。这一专用名词的出处可参见：http://www.catb.org/jargon/html/T/thunk.html。

P. Z. Ingerman 在 1961 年首次提出了 thunk 的概念，这个概念沿用至今：在编译过程中，为满足当时的过程（函数）调用约定，当形参为表达式时，编译器都会产生 thunk，把返回值的地址传递给形参。

微软和 IBM 都对“thunk”一词有定义，将从 16 位到 32 位和从 32 位到 16 位的转变叫作“thunk”。

3.4.5 ARM64

GCC

使用 GCC 4.8.1 将上述代码编译为 ARM64 程序，可得到如下所示的代码。

指令清单 3.15 Non-optimizing GCC 4.8.1 + objdump

```
1  0000000000400590 <main>:
2    400590:       a9bf7bfd    stp     x29, x30, [sp,#-16]!
3    400594:       910003fd    mov     x29, sp
4    400598:       90000000    adrp    x0, 400000 <_init-0x3b8>
5    40059c:       91192000    add     x0, x0, #0x648
6    4005a0:       97ffffa0    bl      400420 <puts@plt>
7    4005a4:       52800000    mov     w0, #0x0        //  #0
8    4005a8:       a8c17bfd    ldp     x29, x30, [sp],#16
9    4005ac:       d65f03c0    ret
10
11 ...
12
13 Contents of section .rodata:
14  400640 01000200 00000000 48656c6c 6f210000  ........Hello!..
```

一方面，ARM64 的 CPU 只可能运行于 ARM 模式、不可运行于 Thumb 或 Thumb-2 模式，所以它必须使用 32 位的指令。另一方面，64 位平台的寄存器数量也翻了一翻，拥有了 64 个 X-字头寄存器（请参见附录 B.4.1）。当然，程序还可以通过 W-字头的名称直接访问寄存器的低 32 位空间。

上述程序的 STP（Store Pair）指令把两个寄存器（即 X29,X30）的值存储到栈里。虽然这个指令实际上可以把这对数值存储到内存中的任意地址，但是由于该指令明确了 SP 寄存器，所以它就是通过栈来存储这对数值。ARM64 平台的寄存器都是 64 位寄存器，每个寄存器可存储 8 字节数据。所以程序要分配 16 字节的空间来存储两个寄存器的值。

这条指令中的感叹号标志，意味着其标注的运算会被优先执行。即，该指令先把 SP 的值减去 16，在此之后再把两个寄存器的值写在栈里。这属于“预索引/pre-index”指令。此外还有“延迟索引/post-index”指令与之对应。有关两者的区别，请参见本书 28.2 节。

以更为易懂的 x86 指令来解读的话，这条指令相当于 PUSH X29 和 PUSH X30 两条指令。在 ARM64 平台上，X29 寄存器是帧指针 FP，X30 起着 LR 的作用，所以这两个寄存器在函数的序言和尾声处成对出现。

第二条指令把 SP 的值复制给 X29，即 FP。这用来设置函数的栈帧。

ADRP 和 ADD 指令相互配合，把“Hello!”字符串的指针传递给 X0 寄存器，继而充当函数参数传递给被调用函数。受到指令方面的限制，ARM 无法通过单条指令就把一个较大的立即数赋值给寄存器（可参见本书的 28.3.1 节）。所以，编译器要组合使用数条指令进行立即数赋值。第一条 ADRP 把 4KB 页面的地址传递给 X0，而后第二条 ADD 进行加法运算并给出最终的指针地址。详细解释请参见本书 28.4 节。

0x400000 + 0x648 = 0x400648。这个数是位于.rodata 数据段的 C 字符串“Hello!”的地址。

接下来，程序使用 BL 指令调用 puts()函数。这部分内容的解读可参见 3.4.3 节。

MOV 指令用来给 W0 寄存器置零。W0 是 X0 寄存器的低 32 位，如下图所示。

<table>
<tr><td>高 32 位</td><td>低 32 位</td></tr>
<tr><td colspan="2">X0</td></tr>
<tr><td></td><td>W0</td></tr>
</table>

main()函数通过 X0 寄存器来传递函数返回值 0。程序后续的指令依次制备这个返回值。为什么这里把返回值存储到 X0 寄存器的低 32 位，即 W0 寄存器？这种情况和 x86-64 平台相似：出于兼容性和向下兼容的考虑，

ARM64 平台的 int 型数据仍然是 32 位数据。对于 32 位的 int 型数据来说，X0 寄存器的低 32 位足够大了。

为了进行演示，我对源代码进行了小幅度的修改，使 main()返回 64 位值。

指令清单 3.16　main()返回 uint64_t 型数据

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

返回值虽然相同，但是对应的 MOV 指令发生了变化。

指令清单 3.17　Non-optimizing GCC 4.8.1 + objdump

```
  4005a4:   d2800000   mov    x0, #0x0          // #0
```

在此之后，LDP (Load Pair)指令还原 X29 和 X30 寄存器的值。此处的这条指令没有感叹号标记，这意味着它将率先进行赋值操作，而后再把 SP 的值与 16 进行求和运算。这属于延时索引（post-index）指令。

RET 指令是 ARM64 平台的特色指令。虽然它的作用与 BX LR 相同，但是它实际上是按照寄存器的名称进行跳转的（默认使用 X30 寄存器指向的地址），通过底层指令提示 CPU 此处为函数的返回指令、不属于普通转移指令的返回过程。RET 指令经过了面向硬件的优化处理，它的执行效率较高。

开启优化功能之后，GCC 生成的代码完全一样。本文不在对它进行介绍。

3.5　MIPS

3.5.1　全局指针 Global pointer

全局指针是 MIPS 软件系统的一个重要概念。我们已经知道，每条 MIPS 指令都是 32 位的指令，所以单条指令无法容纳 32 位地址（指针）。这种情况下 MIPS 就得传递一对指令才能使用一个完整的指针。在前文的例子中，GCC 在生成文本字符串的地址时，就采用了类似的技术。

从另一方面来说，单条指令确实可以容纳一组由寄存器的符号、有符号的 16 位偏移量（有符号数）。因此任何一条指令都可以构成的表达式，访问某个取值范围为“寄存器-32768”～“寄存器+32767”之间的地址（总共 69KB）。为了简化静态数据的访问操作，MIPS 平台特地为此保留了一个专用的寄存器，并且把常用数据分配到了一个大小为 64KB 的内存数据空间里。这种专用的寄存器就叫作“全局指针”寄存器。它的值是一个指针，指向 64KB（静态）数据空间的正中间。而这 64KB 空间通常用于存储全局变量，以及 printf()这类由外部导入的的外部函数地址。GCC 的开发团队认为：获取函数地址这类的操作，应当由单条指令完成；双指令取址的运行效率不可接受。

在 ELF 格式的文件中，这个 64KB 的静态数据位于.sbss 和.sdata 之中。“.sbss”是 small BSS（Block Started by Symbol）的缩写，用于存储非初始化的数据。“.sdata”是 small data 的缩写，用于存储有初始化数值的数据。

根据这种数据布局编程人员可以自行决定把需要快速访问的数据放在.sdata、还是.sbss 数据段中。

有多年工作经验的人员可能会把全局指针和 MS-DOS 内存（参见本书第 49 章）、或者 MS-DOS 的 XMS/EMS 内存管理器联系起来。这些内存管理方式都把数据的内存存储空间划分为数个 64KB 区间。

全局指针并不是 MIPS 平台的专有概念。至少 PowerPC 平台也使用了这一概念。

3.5.2　Optimizing　GCC

下面这段代码显示了“全局指针”的特色。

指令清单 3.18 Optimizing GCC 4.4.5 (汇编输出)

```
 1 $LC0:
 2 ; \000 is zero byte in octal base:
 3         .ascii "Hello, world!\012\000"
 4 main:
 5 ; function prologue.
 6 ; set the GP:
 7         lui     $28,%hi(__gnu_local_gp)
 8         addiu   $sp,$sp,-32
 9         addiu   $28,$28,%lo(__gnu_local_gp)
10 ; save the RA to the local stack:
11         sw      $31,28($sp)
12 ; load the address of the puts() function from the GP to $25:
13         lw      $25,%call16(puts)($28)
14 ; load the address of the text string to $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; jump to puts(), saving the return address in the link register:
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; restore the RA:
20         lw      $31,28($sp)
21 ; copy 0 from $zero to $v0:
22         move    $2,$0
23 ; return by jumping to the RA:
24         j       $31
25 ; function epilogue:
26         addiu $sp,$sp,32 ; branch delay slot
```

主函数序言启动部分的指令初始化了全局指针寄存器 GP 寄存器的值，并且把它指向 64KB 数据段的正中央。同时，程序把 RA 寄存器的值存储于本地数据栈。它同样使用 puts()函数替代了 printf()函数。而 puts()函数的地址，则通过 LW（Load Word）指令加载到了$25 寄存器。此后，字符串的高 16 位地址和低 16 位地址分别由 LUI(Load Upper Immediate)和 ADDIU(Add Immediate Unsigned Word)两条指令加载到$4 寄存器。LUI 中的 Upper 一词说明它将数据存储于寄存器的高 16 位。与此相对应，ADDIU 则把操作符地址处的低 16 位进行了求和运算。ADDIU 指令位于 JALR 指令之后，但是会先于后者运行[①]。$4 寄存器其实就是$A0 寄存器，在调用函数时传递第一个参数[②]。

JALR (Jump and Link Register)指令跳转到$25 寄存器中的地址，即 puts()函数的启动地址，并且把下一条 LW 指令的地址存储于 RA 寄存器。可见，MIPS 系统调用函数的方法与 ARM 系统相似。需要注意的是，由于分支延迟槽效应，存储于 RA 寄存器的值并非是已经运行过的、“下一条”指令的地址，而是更后面那条（延迟槽之后的）指令的地址。所以，在执行这条 JALR 指令的时候，写入 RA 寄存器的值是 PC+8，即 ADDIU 后面的那条 LW 指令的地址。

第 19 行的 LW (Load Word)指令，用于把本地栈中的 RA 值恢复回来。请注意，这条指令并不位于被调用函数的函数尾声。

第 22 行的 MOVE 指令把$0（$ZERO）的值复制给$2($V0)。MIPS 有一个常量寄存器，它里面的值是常量 0。很明显，因为 MIPS 的研发人员认为 0 是计算机编程里用得最多的常量，所以他们开创了一种使用$0 寄存器提供数值 0 的机制。这个例子演示了另外一个值得注意的现象：在 MIPS 系统之中，没有在寄存器之间复制数值的（硬件）指令。确切地说，MOVE DST, SRC 是通过加法指令 ADD DST,SRC, $ZERO 变相实现的，即 DST=SRC+0，这两种操作等效。由此可见，MIPS 研发人员希望尽可能地复用 opcode，从而精简 opcode 的总数。然而这并不代表每次运行 MOVE 指令时 CPU 都会进行实际意义上的加法运算。CPU 能够对这类伪指令进行优化处理，在运行它们的时候并不会用到 ALU（Arithmetic logic unit）。

第 24 行的 J 指令会跳转到 RA 所指向的地址，完成从被调用函数返回调用者函数的操作。还是由于分支延迟槽效应，其后的 ADDIU 指令会先于 J 指令运行，构成函数尾声。

① 请参考前文介绍的分支延迟槽（Branch delay slot）效应。
② 有关 MIPS 各寄存器的用途，请参见附录 C.1。

我们再来看看 IDA 生成的指令清单，熟悉一下各寄存器的伪名称。

代码清单 3.19　Opimizing GCC4.4.5(IDA)

```
 1 .text:00000000 main:
 2 .text:00000000
 3 .text:00000000 var_10         = -0x10
 4 .text:00000000 var_4          = -4
 5 .text:00000000
 6 ; function prologue.
 7 ; set the GP:
 8 .text:00000000                lui         $gp, (__gnu_local_gp >> 16)
 9 .text:00000004                addiu       $sp, -0x20
10 .text:00000008                la          $gp, (__gnu_local_gp & 0xFFFF)
11 ; save the RA to the local stack:
12 .text:0000000C                sw          $ra, 0x20+var_4($sp)
13 ; save the GP to the local stack:
14 ; for some reason, this instruction is missing in the GCC assembly output:
15 .text:00000010                sw          $gp, 0x20+var_10($sp)
16 ; load the address of the puts() function from the GP to $t9:
17 .text:00000014                lw          $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018                lui         $a0, ($LC0 >> 16) # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C                jalr        $t9
22 .text:00000020                la          $a0, ($LC0 & 0xFFFF) # "Hello, world!"
23 ; restore the RA:
24 .text:00000024                lw          $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028                move        $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C                jr          $ra
29 ; function epilogue:
30 .text:00000030                addiu       $sp, 0x20
```

第 15 行的指令使用局部栈保存 GP 的值。令人感到匪夷所思的是，GCC 的汇编输出里看不到这条指令，或许这是 GCC 自身的问题[①]。严格地说，此时有必要保存 GP 的值。毕竟每个函数都有着自己的 64KB 数据窗口。

程序中保存 puts()函数地址的寄存器叫作$T9 寄存器。这类 T-开头的寄存器叫作“临时”寄存器，用于保存代码里的临时值。调用者函数负责保存这些寄存器的数值（caller-saved），因为它有可能会被被调用的函数重写。

3.5.3　Non-optimizing GCC

代码清单 3.20　Non-optimizing GCC 4.4.5 (汇编输出)

```
 1 $LC0:
 2         .ascii "Hello, world!\012\000"
 3 main:
 4 ; function prologue.
 5 ; save the RA ($31) and FP in the stack:
 6         addiu  $sp,$sp,-32
 7         sw     $31,28($sp)
 8         sw     $fp,24($sp)
 9 ; set the FP (stack frame pointer):
10         move   $fp,$sp
11 ; set the GP:
12         lui    $28,%hi(__gnu_local_gp)
13         addiu  $28,$28,%lo(__gnu_local_gp)
14 ; load the address of the text string:
```

① 很明显，对于 GCC 的用户来说，查看汇编指令的功能不是那么重要。所以，GCC 输出的汇编指令之中仍然可能存在一些（在生成汇编指令的阶段）未被修正的错误。

```
15         lui   $2,%hi($LC0)
16         addiu $4,$2,%lo($LC0)
17 ; load the address of puts() using the GP:
18         lw    $2,%call16(puts)($28)
19         nop
20 ; call puts():
21         move  $25,$2
22         jalr  $25
23         nop; branch delay slot
24
25 ; restore the GP from the local stack:
26         lw    $28,16($fp)
27 ; set register $2 ($V0) to zero:
28         move $2,$0
29 ; function epilogue.
30 ; restore the SP:
31         move $sp,$fp
32 ; restore the RA:
33         lw    $31,28($sp)
34 ; restore the FP:
35         lw    $fp,24($sp)
36         addiu $sp,$sp,32
37 ; jump to the RA:
38         j     $31
39         nop; branch delay slot
```

未经优化处理的 GCC 输出要详细得多。此处，我们可以观察到程序把 FP 当作栈帧的指针来用，而且它还有 3 个 NOP（空操作）指令。在这 3 个空操作指令中，第二个、第三个指令都位于分支跳转指令之后。

笔者个人认为（虽然目前无法肯定），由于这些地方都存在分支延迟槽，所以 GCC 编译器会在分支语句之后都添加 NOP 指令。不过，在启用它的优化选项之后，GCC 可能就会删除这些 NOP 指令。所以，此处仍然存在这些 NOP 指令。

使用 IDA 程序观察下面这段代码。

指令清单 3.21　Non-optimizing GCC 4.4.5 (IDA)

```
 1 .text:00000000 main:
 2 .text:00000000
 3 .text:00000000 var_10          = -0x10
 4 .text:00000000 var_8           = -8
 5 .text:00000000 var_4           = -4
 6 .text:00000000
 7 ; function prologue.
 8 ; save the RA and FP in the stack:
 9 .text:00000000                 addiu     $sp, -0x20
10 .text:00000004                 sw        $ra, 0x20+var_4($sp)
11 .text:00000008                 sw        $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C                 move      $fp, $sp
14 ; set the GP:
15 .text:00000010                 la        $gp, __gnu_local_gp
16 .text:00000018                 sw        $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C                 lui       $v0, (aHelloWorld >> 16) # "Hello, world!"
19 .text:00000020                 addiu     $a0, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024                 lw        $v0, (puts & 0xFFFF)($gp)
22 .text:00000028                 or        $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C                 move      $t9, $v0
25 .text:00000030                 jalr      $t9
26 .text:00000034                 or        $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038                 lw        $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
```

```
30 .text:0000003C                move      $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040                move      $sp, $fp
34 ; restore the RA:
35 .text:00000044                lw        $ra, 0x20+var_4($sp)
36 ; restore the FP:
37 .text:00000048                lw        $fp, 0x20+var_8($sp)
38 .text:0000004C                addiu     $sp, 0x20
39 ; jump to the RA:
40 .text:00000050                jr        $ra
41 .text:00000054                or        $at, $zero ; NOP
```

在程序的第 15 行出现了一个比较有意思的现象——IDA 识别出了 LUI/ADDIU 指令对，把它们显示为单条的伪指令 LA(Load address)。那条伪指令占用了 8 个字节！这种伪指令（即“宏”）并非真正的 MIPS 指令。通过这种名称替换,IDA 帮助我们这对指令的作用望文思义。

NOP 的显示方法也构成了它的另外一种特点。因为 IDA 并不会自动地把实际指令匹配为 NOP 指令，所以位于第 22 行、第 26 行、第 41 行的指令都是“OR $AT, $ZERO”。表面上看，它将保留寄存器$AT 的值与 0 进行或运算。但是从本质上讲，这就是发送给 CPU 的 NOP 指令。MIPS 和其他的一些硬件平台的指令集都没有单独的 NOP 指令。

3.5.4　栈帧

本例使用寄存器来传递文本字符串的地址。但是它同时设置了局部栈，这是为什么呢？由于程序在调用 printf()函数的时候由于程序必须保存 RA 寄存器的值和 GP 的值，故而此处出现了数据栈。如果此函数是叶函数，它有可能不会出现函数的序言和尾声，有关内容请参见本书的 2.3 节。

3.5.5　Optimizing　GCC:　GDB 的分析方法

指令清单 3.22　GDB 的操作流程

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mips-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:     lui     gp,0x42
0x00400644 <main+4>:     addiu   sp,sp,-32
0x00400648 <main+8>:     addiu   gp,gp,-30624
0x0040064C <main+12>:    sw      ra,28(sp)
0x00400650 <main+16>:    sw      gp,16(sp)
0x00400654 <main+20>:    lw      t9,-32716(gp)
0x00400658 <main+24>:    lui     a0,0x40
0x0040065c <main+28>:    jalr    t9
0x00400660 <main+32>:    addiu   a0,a0,2080
```

```
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

3.6 总结

x64 和 x86 指令的主要区别体现在指针上，前者使用 64 位指针而后者使用 32 位指针。近年来，内存的价格在不断降低，而 CPU 的计算能力也在不断增强，当计算机的内存增加到一定程度时，32 位指针就无法满足寻址的需要了，所以指针也随之演变为 64 位指针。

3.7 练习题

3.7.1 题目 1

请描述下述 32 位函数的功能。

```
main:
    push 0xFFFFFFFF
    call MessageBeep
    xor  eax,eax
    retn
```

3.7.2 题目 2

请描述 Linux 函数的功能，这里使用了 AT&T 汇编语言语法。

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %2, %edi
    call    sleep
    popq    %rbp
    ret
```

第 4 章　函数序言和函数尾声

函数序言（function prologue）是函数在启动的时候运行的一系列指令。其汇编指令大致如下：

```
push  ebp
mov   ebp, esp
sub   esp, X
```

这些指令的功能是：在栈里保存 EBP 寄存器的内容、将 ESP 的值复制到 EBP 寄存器，然后修改栈的高度，以便为本函数的局部变量申请存储空间。

在函数执行期间，EBP 寄存器不受函数运行的影响它是函数访问局部变量和函数参数的基准值。虽然我们也可便中 ESP 寄存器存储局部变量和运行参数，但是 ESP 寄存器的值总是会发生变化，使用起来并不方便。

函数在退出时，要做启动过程的反操作，释放栈中申请的内存，还原 EBP 寄存器的值，将代码控制权还原给调用者函数(callee)。

```
mov   esp, ebp
pop   ebp
ret   0
```

借助函数序言和函数尾声的有关特征，我们可以在汇编语言里识别各个函数。

递归调用

函数序言和尾声都会调整数据栈受硬件 IO 性能影响，所有递归函数的性能都不太理想。

详细内容请参见本书的 36.3 节。

第5章 栈

栈是计算机科学里最重要且最基础的数据结构之一。[①]

从技术上说，栈就是CPU寄存器里的某个指针所指向的一片内存区域。这里所说的“某个指针”通常位于x86/x64平台的ESP寄存器/RSP寄存器，以及ARM平台的SP寄存器。

操作栈的最常见的指令是PUSH和POP，在x86和ARM Thumb模式的指令集里都有这两条指令。PUSH指令会对ESP/RSP/SP寄存器的值进行减法运算，使之减去4（32位）或8（64位），然后将操作数写到上述寄存器里的指针所指向的内存中。

POP指令是PUSH指令的逆操作：它先从栈指针（Stack Pointer，上面三个寄存器之一）指向的内存中读取数据，用以备用（通常是写到其他寄存器里），然后再将栈指针的数值加上4或8。

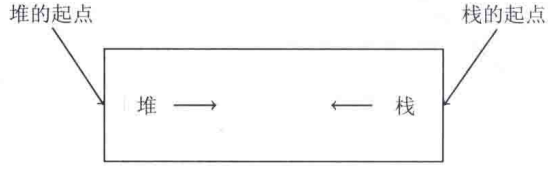
在分配栈的空间之后，栈指针，即Stack Pointer所指向的地址是栈的底部。PUSH将减少栈指针的数值，而POP会增加它的数值。栈的“底”实际上使用的是整个栈的最低地址，即是整个栈的启始内存地址。虽然听起来很奇怪，但是实际上确实如此。

虽然x86/x64的栈已经很难理解了，但是ARM的栈却更为复杂。ARM的栈分为递增栈和递减栈。递减栈（descending stack）的首地址是栈的最高地址，栈向低地址增长，栈指针的值随栈的增长而减少，如STMFD/LDMFD、STMED/LDMED等指令都是递减栈的操作指令。而ARM的递增栈（ascending stack）的首地址则占用栈的最低地址，栈向高地址增长，栈指针的值随栈的增长而增加，如STMFA/LMDFA、STMEA/LDMEA等指令都是递增栈的操作指令。[②]

5.1 为什么栈会逆增长

多数人想象中的“增长”，是栈从低地址位向高地址位增长，似乎这样才符合自然规律。然而研究过栈的人知道，多数的栈是逆增长的，它会从高地址向低地址增长。

这多数还得归功于历史原因。当计算机尚未小型化的时候，它还有数个房间那么大。在那个时候，内存就分为两个部分，即“堆/heap”和“栈/stack”。当然，在程序执行过程中，堆和栈到底会增长到什么地步并不好说，所以人们干脆把它们分开：



有兴趣的读者可以查阅参考文献RT74，其中有这样一段话：

> 程序镜像（进程）在逻辑上分为3个段。从虚拟地址空间的0地址位开始，第一个段是文本段（也称为代码段）。文本段在执行过程中不可写，即使一个程序被执行多次，它也必须共享1份文本段。

---

① 请参见 http://en.wikipedia.org/wiki/Call_stack。

② 这些指令所对应的英文全称，分别是Store Multiple Full Descending、Load Multiple Full Descending、Store Multiple Empty Descending、Load Multiple Empty Descending 、Store Multiple Full Ascending 、Load Multiple Full Ascending 、Store Multiple Empty Ascending、 Load Multiple Empty Ascending。其中的Full/Empty的区别在于：如果指针指向的地址是最新操作的值，那么它就是Full stack；而指针指向的地址没有值、是下一个值将写到的地址，那么这个栈就是Empty stack。

在程序虚拟空间中，文本段 8k bytes 边界之上，是不共享的、可写的数据段。程序可以通过调用系统函数调整其数据段的大小。栈启始于虚拟地址空间的最高地址，它应随着硬件栈指针的变化而自动地向下增长。

这就好比用同一个笔记本给两门课程做笔记：第一门的笔记可以按照第一页往最后一页的顺序写；然而在做第二门的笔记时，笔记本要反过来用，也就是要按照从最后一页往第一页的顺序写笔记。至于笔记本什么时候会用完，那就要看笔记本有多厚了。

5.2　栈的用途

5.2.1　保存函数结束时的返回地址

x86

当程序使用 call 指令调用其他函数时，call 指令结束后的返回地址将被保存在栈里；在 call 所调用的函数结束之后，程序将执行无条件跳转指令，跳转到这个返回地址。

CALL 指令等价于“PUSH 返回地址”和“JMP 函数地址”的指令对。

被调用函数里的 RET 指令，会从栈中读取返回地址，然后跳转到这个地址，就相当于“POP 返回地址”＋“JMP 返回地址”指令。

栈是有限的，溢出它很容易。直接使用无限递归，栈就会满：

```
void f()
{
        f();
};
```

如果使用 MSVC 2008 编译上面的问题程序，将会得到报告：

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717:'f' : recursive on all control paths. Function will cause runtime stack overflow
```

但是它还是会老老实实地生成汇编文件：

```
?f@@YAXXZ PROC                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                          ; f
```

有趣的是，如果打开优化选项“/Ox”，生成的程序反而不会出现栈溢出的问题，而且还会运行得很“好”：

```
?f@@YAXXZ PROC                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                          ; f
```

无论是否开启优化选项，GCC 4.4.1 生成的代码都和 MSVC 生成的代码相似，只是 GCC 不会发布任何警告。

ARM

ARM 程序也使用栈保存返回地址，只是略有不同。在 3.4 节中，我们看到“Hello, World!”程序的返回地址保存在 LR (link register)寄存器里。但是，如果程序还会继续调用其他函数，就需要在调用函数之前保存 LR 寄存器的值。通常，函数会在启动过程中（序言处）保存 LR 寄存器的值。我们通常在函数序言处看到“PUSH R4-R7，LR”，并在尾声处看到“POP R4-R7，PC”。这些指令会对函数自身将要用到的寄存器进行保护，把它们的值存放在栈中——当然，这其中也包括 LR 寄存器。

如果一个函数不调用其他函数，它就像树上的枝杈末端的叶子那样。这种函数就叫作“叶函数（leaf function）”[①]。叶函数的特点是，它不必保存 LR 寄存器的值。如果叶函数的代码短到用不到几个寄存器，那么它也可能根本不会使用数据栈。所以，调用叶函数的时候确实可能不会涉及栈操作。这种情况下，因为这种代码不在外部内存 RAM 进行与栈有关的操作，所以它的运行速度有可能超过 x86 系统[②]。在没有分配栈或者不可能用栈的时候，这类函数就会显现出“寸有所长”的优势。

本书介绍了很多的叶函数。请参见 8.3.2 节、8.3.3 节、15.2 节、15.4 节、17.3 节、19.5.4 节、19.17 节、19.33 节中演示的程序。

5.2.2 参数传递

在 x86 平台的程序中，最常用的参数传递约定是 cdecl[③]。以 cdecl 方式处理参数，其上下文大体是这个样子：

```
push arg3
push arg2
push arg1
call f
add esp, 12 4*3=12
```

被调用方函数（Callee functions）通过栈指针获取其所需的参数。

在运行 f()函数之前，传递给它的参数将以以下格式存储在内存里。

ESP	返回地址
ESP+4	arg1，它在 IDA 里记为 arg_0
ESP+8	arg2，它在 IDA 里记为 arg_4
ESP+0xC	arg3，它在 IDA 里记为 arg_8
……	……

本书第 64 章将会详细介绍有关的调用约定（Calling conventions）。需要注意的是，程序员可以使用栈来传递参数，也可以不使用栈传递参数。参数处理方面并没有相关的硬性规定。

例如，程序员可以在堆（heap）中分配内存并用之传递参数。在堆中放入参数之后，可以利用 EAX 寄存器为函数传递参数。这种做法确实行得通。[④]只是在 x86 系统和 ARM 系统上，使用栈处理参数已经成为了约定俗成的习惯，而且它的确十分方便。

另外，被调用方函数并不知晓外部向它传递了多少个参数。如果函数可处理的参数数量可变，它就需要说明符（多数以%号开头）进行格式化说明、明确参数信息。拿我们常见的 printf()函数来说：

```
printf("%d %d %d", 1234);
```

这个命令不仅会让 printf()显示 1234，而且还会让它显示数据栈内 1234 之后的两个地址的随机数。

---

① 参照 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html。部分文献又称叶函数为“末端函数”

② 多年以前，在 PDP-11 和 VAX 平台上，调用函数的指令效率很低；在运行程序的时候，差不多近半的耗时都花在互相调用的指令上。以至于当时普遍认为、“大规模调用功能较少的函数”的程序就是垃圾程序。

③ 此外还有 stdcall、fastcall、thiscall 等。Windows 上很多程序使用 stdcall。

④ Donald Knuth 在《The Art of Computer Programming》 一书的 14.1 节专门介绍了子程序。读者能够发现他提到了一种方法，通过 JMP 跳转到子程序并在子程序的入口处即刻提取所需参数。Knuth 的这种方法在 System/360 上非常实用。

由此可知，声明 main()函数的方法并不是那么重要。我们可以将之声明为 main(),main(int argc, char *argv[])或 main(int argc, char *argv[], char *envp[])。

实际上 CRT 中调用 main()的指令大体上是下面这个样子的。

```
push envp
push argv
push argc
call main
…
```

即使我们没有在程序里声明 main()函数便用哪些参数，程序还可以照常运行；参数依旧保存在栈里，只是不会被主函数调用罢了。如果将 main()函数声明为 main(int argc, char*argv[])，程序就能够访问到前两个参数，但仍然无法使用第三个参数。除此以外，也可以声明为 main(int argc)，主函数同样可以运行。

5.2.3　存储局部变量

通过向栈底调整栈指针（stack pointer）的方法，函数即可在数据栈里分配出一片可用于存储局部变量的内存空间。可见，无论函数声明了多少个局部变量，都不影响它分配栈空间的速度。

虽然您的确可以在栈以外的任何地方存储局部变量，但是用数据栈来存储局部变量已经是一种约定俗成的习惯了。

5.2.4　x86:alloca()函数

alloca()函数很有特点。[①]

大体来说，alloca()函数直接使用栈来分配内存，除此之外，它与 malloc()函数没有显著的区别。

函数尾声的代码会还原 ESP 的值，把数据栈还原为函数启动之前的状态，直接抛弃由 alloca()函数分配的内存。所以，程序不需要特地使用 free()函数来释放由这个函数申请的内存。

alloca() 函数的实现方法至关重要。

简要地说，这个函数将以所需数据空间的大小为幅度、向栈底调整 ESP 的值，此时 ESP 就成为了新的数据空间的指针。我们一起做个试验：

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

snprint()函数的功能和 printf()函数的功能差不多。printf()将输出结果输出到 stdout（也就是终端 terminal 或 console 一类的输出设备上），而 snprintf()则将结果输出到 buf 数组（人工设定的缓冲区）、我们需要通过 puts()函数才能将 buf 的内容输出到 stdout。当然，printf()函数就足以完成_snprint()和 puts()两个函数的功能。本文意在演示缓冲区的用法，所以刻意把它拆分为 2 个函数。

---

① 在 MSVC 编译环境里，它的实现方式可在目录 C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel 下的 alloca16.asm 和 chkstk.asm 中找到。

MSVC

现在使用 MSVC 2010 编译上面的代码，得到的代码段如下所示。

指令清单 5.1 MSVC 2010

```
...
   mov     eax, 600 ; 00000258H
   call    __alloca_probe_16
   mov     esi, esp

   push    3
   push    2
   push    1
   push    OFFSET $SG2672
   push    600             ; 00000258H
   push    esi
   call    __snprintf

   push    esi
   call    _puts
   add     esp, 28         ; 0000001cH
...
```

由于 alloca()函数是编译器固有函数（参见本书第 90 章），并非常规函数的缘故，这个程序没有使用栈，而是使用 EAX 寄存器来传递 alloca()函数唯一的参数。在调用 alloca()函数之后，ESP 将指向 600 字节大小的内存区域，用以存储数组 buf。

GCC Intel 语体

在编译上述代码时，GCC 4.4.1 同样不会调用外部函数。

指令清单 5.2 GCC 4.7.3

```
.LC0:
        .string  "hi! %d, %d, %d\n"
f:
        push     ebp
        mov      ebp, esp
        push     ebx
        sub      esp, 660
        lea      ebx, [esp+39]
        and      ebx, -16                              ; align pointer by 16-bit border
        mov      DWORD PTR [esp], ebx                  ; s
        mov      DWORD PTR [esp+20], 3
        mov      DWORD PTR [esp+16], 2
        mov      DWORD PTR [esp+12], 1
        mov      DWORD PTR [esp+8], OFFSET FLAT:.LC0   ; "hi! %d, %d, %d\n"
        mov      DWORD PTR [esp+4], 600                ; maxlen
        call     _snprintf
        mov      DWORD PTR [esp], ebx                  ; s
        call     puts
        mov      ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC+ AT&T 语体

接下来让我们看看 AT&T 语体的指令。

指令清单 5.3 GCC 4.7.3

```
.LC0:
        .string  "hi! %d, %d, %d\n"
f:
```

```
pushl   %ebp
movl    %esp, %ebp
pushl   %ebx
subl    $660, %esp
leal    39(%esp), %ebx
andl    $-16, %ebx
movl    %ebx, (%esp)
movl    $3, 20(%esp)
movl    $2, 16(%esp)
movl    $1, 12(%esp)
movl    $.LC0, 8(%esp)
movl    $600, 4(%esp)
call    _snprintf
movl    %ebx, (%esp)
call    puts
movl    -4(%ebp), %ebx
leave
ret
```

它与 Intel 语体的代码没有实质区别。

其中，“movl $3, 20(%esp)”对应 Intel 语体的“mov DWORD PTR [esp+20], 3”指令。在以“寄存器＋偏移量”的方式寻址时，AT&T 语体将这个寻址表达式显示为“偏移量（%寄存器）”。

5.2.5 （Windows）SEH 结构化异常处理

如果程序里存在 SEH 记录，那么相应记录会保存在栈里。

本书将在 68.3 节里进行更详细的介绍。

5.2.6 缓冲区溢出保护

本书将在 18.2 节里进行更详细的介绍。

5.3 典型的栈的内存存储格式

在 32 位系统中，在程序调用函数之后、执行它的第一条指令之前，栈在内存中的存储格式一般如下表所示。

…	……
ESP-0xC	第 2 个局部变量，在 IDA 里记为 var_8
ESP-8	第 1 个局部变量，在 IDA 里记为 var_4
ESP-4	保存的 EBP 值
ESP	返回地址
ESP+4	arg1，在 IDA 里记为 arg_0
ESP+8	arg2，在 IDA 里记为 arg_4
ESP+0xC	arg3，在 IDA 里记为 arg_8
…	……

5.4 栈的噪音

本书会经常使用“噪音”、“脏数据”这些词汇。它们怎么产生的呢？待函数退出以后，原有栈空间里的局部变量不会被自动清除。它们就成为了栈的“噪音”、“脏数据”。我们来看下面这段代码。

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};
```

```
void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

使用 MSVC 2010 编译后可得到如下所示的代码。

指令清单 5.4 Non-optimizing MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

main    PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

编译器会给出提示：

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:st.exe
st.obj
```

可是运行它的结果却是：

```
c:\Polygon\c>st
1, 2, 3
```

天，真奇怪！虽然我们没有给 f2()的任何变量赋值，但是变量自己有自己的值。f2()函数的值就是栈里残存的脏数据。

我们使用 OllyDbg 打开这个程序，如图 5.1 所示。

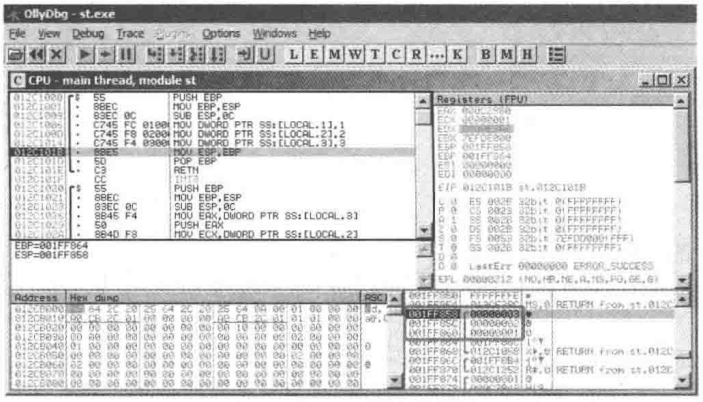


图 5.1　使用 OllyDbg 查看 f1()函数的数据栈

在 f1()函数给变量 *a*、*b*、*c* 赋值后，数值存储于 0x1FF860 开始的连续地址里。

然后，在执行 f2()时，情况如图 5.2 所示。

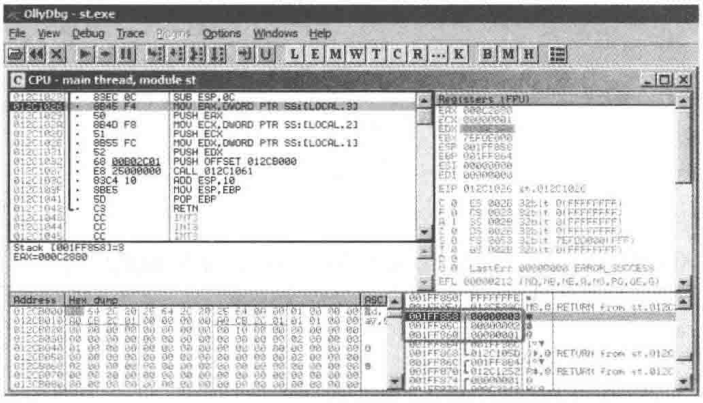


图 5.2　使用 OllyDbg 查看 f2()函数的数据栈

f2()函数的三个变量的地址，和 f1()函数的三个变量的地址相同。因为没有对这个空间进行重新赋值，所以那三个变量会因为地址相同的原因获得前三个变量的值。

在这个特例里，第二个函数在第一个函数之后执行，而第二个函数变量的地址和 SP 的值又与第一个

函数的情况相同。所以，相同地址的变量获得的值相同。

总而言之，在运行第二个函数时，栈中的所有值（即内存中的单元）受前一个函数的影响，而获得了前一个函数的变量的值。严格地说，这些地址的值不是随机值，而是可预测的伪随机值。

有没有办法清除脏数据呢？我们可以在每个函数执行之前清除其开辟的栈空间的数据。不过，即使这是一种技术上可行的方法，但是因为这种方法开销太大、而且必要性很低，所以没有人这样做。

5.5 练习题

5.5.1 题目 1

如果使用 MSVC 编译、运行下列程序，将会打印出 3 个整数。这些数值来自哪里？如果使用 MSVC 的优化选项“/Ox”，程序又会在屏幕上输出什么？为什么 GCC 的情况完全不同？

```
#include <stdio.h>

int main()
{
        printf ("%d, %d, %d\n");

        return 0;
};
```

答案请参见 G1.1。

5.5.2 题目 2

请问描述下述程序的功能。

经 MSVC 2010（启用/Ox 选项）编译而得的代码如下。

指令清单 5.5 Optimizing MSVC 2010

```
$SG3103 DB      '%d', 0aH, 00H

_main   PROC
        push    0
        call    DWORD PTR __imp__time64
        push    edx
        push    eax
        push    OFFSET $SG3103 ; '%d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        xor     eax, eax
        ret     0
_main   ENDP
```

指令清单 5.6 经 Keil 6/2013（启用优化选项）编译而得的 ARM 模式代码

```
main PROC
        PUSH    {r4,lr}
        MOV     r0,#0
        BL      time
        MOV     r1,r0
        ADR     r0,|L0.32|
        BL      __2printf
        MOV     r0,#0
        POP     {r4,pc}
        ENDP
|L0.32|
```

```
        DCB     "%d\n",0
```

指令清单 5.7　经 Keil 6/2013（启用优化选项）编译而得的 Thumb 模式代码

```
main PROC
        PUSH    {r4,lr}
        MOVS    r0,#0
        BL      time
        MOVS    r1,r0
        ADR     r0,|L0.20|
        BL      __2printf
        MOVS    r0,#0
        POP     {r4,pc}
        ENDP

|L0.20|
        DCB     "%d\n",0
```

指令清单 5.8　经 GCC 4.9（启用优化选项）编译而得的 ARM64 模式代码

```
main:
        stp     x29, x30, [sp, -16]!
        mov     x0, 0
        add     x29, sp, 0
        bl      time
        mov     x1, x0
        ldp     x29, x30, [sp], 16
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        b       printf
.LC0:
        .string "%d\n"
```

指令清单 5.9　经 GCC 4.4.5（启用优化选项）编译而得的 MIPS 指令（IDA）

```
main:

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lw      $t9, (time & 0xFFFF)($gp)
                or      $at, $zero
                jalr    $t9
                move    $a0, $zero
                lw      $gp, 0x20+var_10($sp)
                lui     $a0, ($LC0 >> 16)  # "%d\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "%d\n"
                move    $a1, $v0
                jr      $t9
                addiu   $sp, 0x20

$LC0:           .ascii "%d\n"<0>        # DATA XREF: main+28
```

第 6 章　printf()函数与参数传递

现在我们对 Hello, world! 程序稍做修改，演示它的参数传递的过程。

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

6.1　x86

6.1.1　x86：传递 3 个参数

MSVC

使用 MSVC 2010 express 编译上述程序，可得到下列汇编指令：

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H
...
        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                         ; 00000010H
```

这与最初的 Hello World 程序相差不多。我们看到 printf()函数的参数以逆序存入栈里，第一个参数在最后入栈。

在 32 位环境下，32 位地址指针和 int 类型数据都占据 32 位/4 字节空间。所以，我们这里的四个参数总共占用 4×4=16（字节）的存储空间。

在调用函数之后，“ADD ESP, X”指令修正 ESP 寄存器中的栈指针。通常情况下，我们可以通过 call 之后的这条指令判断参数的数量：变量总数＝X÷4。

这种判断方法仅适用于调用约定为 cdecl 的程序。本书将在第 64 章详细介绍各种函数约定[①]。

如果某个程序连续地调用多个函数，且调用函数的指令之间不夹杂其他指令，那么编译器可能把释放参数存储空间的“ADD ESP,X”指令进行合并，一次性地释放所有空间。例如：

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

① calling convention，又有“调用规范”“调用协定”等译法。

如下是一个规定中的例子。

指令清单 6.1　x86

```
.text:100113E7          push 3
.text:100113E9          call sub_100018B0 ; takes one argument (3)
.text:100113EE          call sub_100019D0 ; takes no arguments at all
.text:100113F3          call sub_10006A90 ; takes no arguments at all
.text:100113F8          push 1
.text:100113FA          call sub_100018B0 ; takes one argument (1)
.text:100113FF          add  esp, 8       ; drops two arguments from stack at once
```

使用 OllyDbg 调试 MSVC 编译出的程序

现在我们在 OllyDbg 中加载这个范例。OllyDbg 是非常受欢迎的 user-land（用户空间）win32 debugger。在使用 MSVC 2012 编译这个样本程序的时候，启用/MD 选项，可使可执行程序与 MSVCR*.DLL 建立动态链接。这样，我们就能在 debugger 里清楚地观察到程序从标准库里调用函数的过程。

在 OllyDbg 里调用可执行文件，将第一个断点设为 ntdll.dll，然后按 F9 键执行。然后把第二个断点设置在 CRT-code 里。我们应该能够找到 main()主函数。

因为 MSVC 将 main()函数分配到代码段的开始处，所以只要向下滚屏就能够在底部找到 main()函数体。如图 6.1 所示。

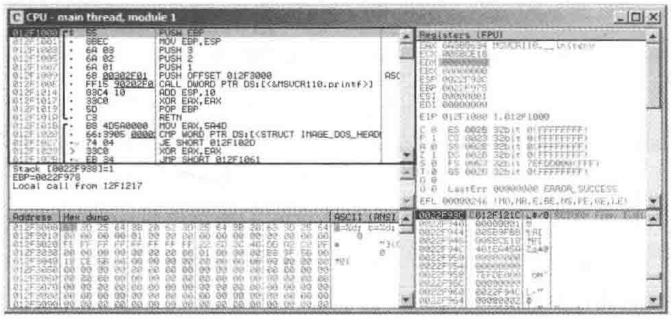


图 6.1　使用 OllyDbg：查看 main()函数的启动部分

单击 PUSH EBP 指令，按 F2 设置断点，再按 F9 键运行。这么做是为了跳过 CRT-code，因为我们的目的不是分析 CRT 代码。

而后按 6 次 F8 键，就是说跳过 6 条指令。如图 6.2 所示。

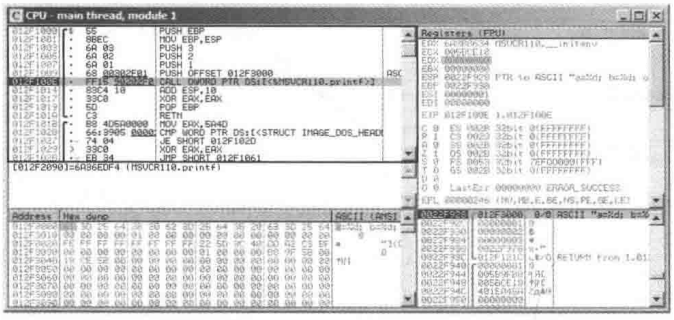


图 6.2　使用 OllyDbg：定位到调用 printf()之前的指令

此时，指令指针 PC 指向 CALL printf 指令。与其他主流的调试器相同，OllyDbg 调试器会加亮显示所有发生过变化的寄存器。每按一次 F8 键，EIP 的值都会发生变化，所以这个寄存器一直被高亮显示。在这个例子中，ESP 同样会被 OllyDbg 高亮显示，因为在这几步调试中 ESP 寄存器的值也发生了变化。

栈里的数值在哪呢？我们看一下 debugger 的右下角，如图 6.3 所示。

此处的内容分为 3 列：栈地址列、数值列及 OllyDbg 的注释列。OllyDbg 能够识别 printf()这样的指令字符串，按照其指令的形式把它所引用的三个值进行了整理。

我们还可以使用鼠标的右键单击 printf()的格式化字符串、单击下拉菜单中的“Follow in dump”，这样屏幕左下方将会显示出格式化字符串的输出结果。调试器左下方的这个区域用于显示内存中的部分数据。我们同样可以编辑这些值。如果修改格式化字符串，那么程序的输出结果就会发生变化。在这个例子中，我们还用不上这样的功能，但是我们可以练练手，从而进一步熟悉调试器。

按下 F8 键，单步执行（Step Over）。

我们将会看到图 6.4 所示的输出结果。

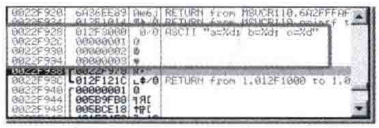


图 6.3 使用 OllyDbg：观察 PUSH 指令造成的栈值变化（红色方块内）

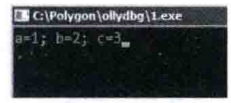


图 6.4 printf() 函数的输出结果

寄存器和栈状态的变化过程如图 6.5 所示。

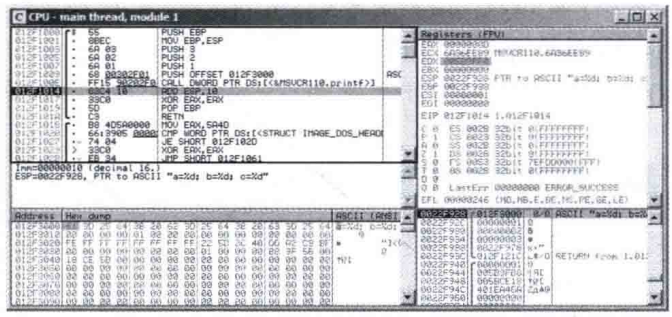


图 6.5 使用 OllyDbg：观察 printf()运行后的数据栈

现在，EAX 寄存器的值是 0xD（即 13）。这个值是 printf()函数所打印的字符总数，所以这个值没有问题。EIP 寄存器的值发生了变化，实际上它是在执行 CALL printf 指令后的 PC。与此同时，ECX 和 EDX 寄存器的值也发生了变化；显然 printf()函数在运行过程中会使用这两个寄存器。

这里最重要的现象是 ESP 的值和栈里的参数都没有发生变化。我们可以观察到数据栈里的、传递给 printf()函数的字符串和其他 3 个参数的值原封未动。这是 cdecl 调用约定的特征：被调用方函数不负责恢复 ESP 的状态；调用方函数（caller function）负责还原参数所用的栈空间。

继续按 F8 键，执行“ADD ESP, 10”指令，如图 6.6 所示。

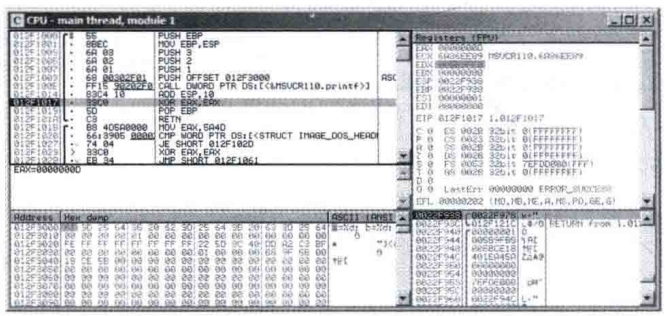


图 6.6 使用 OllyDbg：观察“ADD ESP，10”指令运行之后的状态

ESP 寄存器的值有变化，但是栈中的数据还在那里。因为程序没有把原有栈的数据进行置零（也没有必要清除这些值），所以保留在栈指针 SP 之上的（原有地址）参数值就成为了噪音（noise）或者脏数据（garbage），失去了使用的价值。另外，清除全部噪音的操作十分耗时，程序员完全没有必要刻意地去这么做。

GCC

现在我们使用 GCC 4.4.1 编译这个程序，并使用 IDA 打开可执行文件：

```
main        proc near

var_10      = dword ptr -10h
var_C       = dword ptr -0Ch
var_8       = dword ptr -8
var_4       = dword ptr -4

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 10h
            mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
            mov     [esp+10h+var_4], 3
            mov     [esp+10h+var_8], 2
            mov     [esp+10h+var_C], 1
            mov     [esp+10h+var_10], eax
            call    _printf
            mov     eax, 0
            leave
            retn
main        endp
```

与 MSVC 生成的程序相比，GCC 生成的程序仅在参数入栈的方式上有所区别。在这个例子中，GCC 没有使用 PUSH/POP 指令，而是直接对栈进行了操作。

GCC 和 GDB

GDB 就是 GNU debugger。

在 Linux 下，我们通过下述指令编译示例程序。其中，“-g”选项表示在可执行文件中生成 debug 信息。

```
$ gcc 1.c -g -o 1
```

接下来对可执行文件进行调试。

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

指令清单 6.2　在 printf()函数开始之前设置调试断点

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

然后运行程序。我们的程序里没有 printf() 的源代码，所以 GDB 也无法显示相应源代码。但是，这并不妨碍我们调试程序。

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

继而令 GDB 显示栈里的 10 个数据，其中左边第一列是栈地址。

```
(gdb) x/10w $esp
0xbffff11c:  0x0804844a  0x080484f0  0x00000001  0x00000002
0xbffff12c:  0x00000003  0x08048460  0x00000000  0x00000000
0xbffff13c:  0xb7e29905  0x00000001
```

第一个元素就是返回地址 RA(0x0804844a)。为了验证这点，我们让 GDB 显示出这个地址开始的指令：

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

ret 之后的 XCHG 指令与 NOP 指令等效。x86 平台并没有专用的 NOP 指令，实际上，多数采用 RISC 的 CPU 的指令集里也没有专用的 NOP 指令。

栈中的第二个数据（0x080484f0）是字符串格式的内存地址（指针）：

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

紧接其后的三个数据，是传递给 printf()函数的余下的三个参数。栈中其余的数据，可能是脏数据，也可能是其他函数的数据或局部变量等数据。没有必要进行深究。

此后，我们执行“finish”指令，令调试器“继续执行余下的指令、直到函数结束为止”。在我们的例子里，这条指令将引导 GDB 执行 printf()函数，到它退出为止。

```
(gdb) finish
Run till exit from #0 __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB 显示：在 printf()退出时，EAX 寄存器的值为 13。这个值是函数打印的字符的总数，其结果与 OllyDbg 的结果相同。

GDB 显示出了程序的源代码“return 0;”，它是 1.c 文件里第六行的指令。实际上 1.c 文件就在当前目录里，GDB 在源文件里找到了汇编指令对应的源代码。GDB 又是如何知道当前的指令对应源代码的哪一行呢？这就要归功于编译器所生成的调试信息了。如果启用了保存调试信息的选项，那么编译器在编译程序的时候，会生成源代码的行号与对应的指令地址之间的对应关系表，把它一并保存在可执行文件里。

我们一起检查下 EAX 寄存器里到底是不是 13：

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

使用下述指令反汇编当前的指令。图中箭头指向的是接下来将要运行的指令。

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

默认情况下，GDB 会以 AT&T 语体显示汇编指令。我们可以通过下述指令，令 GDB 采用 Intel 语体：

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp, esp
   0x08048420 <+3>:     and    esp , 0xfffffff0
   0x08048423 <+6>:     sub    esp, 0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR[esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR[esp], 0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax, 0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

单步调试。此处，GDB 用大括号表示这里是函数的结束部分。

```
(gdb) step
7       };
```

在执行过“MOV EAX, 0”指令之后，EAX 会被置零。我们对此进行验证：

```
(gdb) info registers

eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

6.1.2　x64：传递 9 个参数

为了便于演示，我们再次修改代码，将参数的数量增加到 9 个。其中 8 个参数是 int 整型变量，另有 1 个用于控制输出格式的格式化字符串。

```
#include <stdio.h>
int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

前文介绍过，Win64 使用 RCX、RDX、R8、R9 寄存器传递前 4 个参数，使用栈来传递其余的参数。我们将在本例中观察到这个现象。在下面的例子里，编译器使用了 MOV 指令对栈地址进行直接操作而没有使用 PUSH 指令。

指令清单 6.3　MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
```

```
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

在64位系统中，整型数据只占用4字节空间。那么，为什么编译器给整型数据分配了8个字节？其实，即使数据的存储空间不足64位，编译器还是会给它分配8字节的存储空间。这不仅是为了方便系统对每个参数进行内存寻址，而且编译器都会进行地址对齐。所以，64位系统为所有类型的数据都保留8字节空间。同理，32位系统也为所有类型的数据都保留4字节空间。

GCC

x86-64的*NIX系统采用与Win64类似的方法传递参数。它优先使用RDI、RSI、RDX、RCX、R8、R9寄存器传递前六个参数，然后利用栈传递其余的参数。在生成汇编代码时，GCC把字符串指针存储到了EDI寄存器、而非完整的RDI寄存器。在前面的3.2.2节中，64位GCC生成的汇编代码里也出现过这个现象。

在3.2.2节的那个例子里，程序在调用printf()函数之前清空了EAX寄存器。本例中存在相同的操作。

指令清单6.4 Optimizing GCC 4.4.6 x64

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; number of vector registers passed
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; return 0

        xor   eax, eax
        add   rsp, 40
        ret
```

GCC + GDB

在使用GDB调试它之前，我们首先要编译源代码：

```
$ gcc -g 2.c -o 2
```

```
$gdb2
GNU gdb (GDB)7.6.1-ubuntu
Copyright(C)2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version3 or later<http://gnu.org/licenses/gpl.html>
This is free software : you are free to change and redistribute it.
There is NOWARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as"x86_64-linux-gnu".
```

```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

接下来，我们在 printf()运行之前设置断点，然后运行这个程序：

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
29  printf.c: No such file or directory
```

RSI/RDX/RCX/R8/R9 寄存器的值就是传递给函数的参数。RIP 寄存器的值应当是 printf()函数的首地址。我们可以在 GDB 里查看各寄存器的值：

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60    0x7fffffffdf60
rsp            0x7fffffffdf38    0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0    140737488346336
r11            0x7ffff7a65f60    140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040    140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60    0x7ffff7a65f60 <__printf>
...
```

指令清单 6.5　检查格式化字符串

```
(gdb)x/s$rdi
0x400628:       "a=%d;b=%d;c=%d;d=%d;e=%d;f=%d;g=%d;h=%d\n"
```

然后使用 x/g 指令显示栈中的数值。指令中的 g 代表 giant words，即以 64 位 words 型数据的格式显示各数据。

```
(gdb)x/10g$rsp
0x7fffffffdf38: 0x0000000000400576    0x0000000000000006
0x7fffffffdf48: 0x0000000000000007    0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000    0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5    0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048    0x0000000100000000
```

64 位系统的栈与 32 位系统的栈没有太大的区别。栈里的第一个值是返回地址 RA。紧接其后的，是三个保存在栈里的参数 6、7、8。应该注意到数值“8”的高 32 位地址位没有被清零，与之对应的存储空间的值为“0x00007fff00000008”。因为 int 类型只占用（低）32 位，所以这种存储并不会产生问题。另外，我们可以认为这个地方的高 32 位数据是随机的脏数据。

此后，我们通过以下指令查看 printf()函数运行之后的、返回地址开始的有关指令。GDB 将会显示 main()函数的全部指令。

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp, rsp
   0x0000000000400531 <+4>:     sub    rsp, 0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10], 0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8], 0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp], 0x6
   0x000000000040054c <+31>:    mov    r9d, 0x5
```

```
   0x0000000000400552 <+37>:    mov    r8d, 0x4
   0x0000000000400558 <+43>:    mov    ecx, 0x3
   0x000000000040055d <+48>:    mov    edx, 0x2
   0x0000000000400562 <+53>:    mov    esi, 0x1
   0x0000000000400567 <+58>:    mov    edi, 0x400628
   0x000000000040056c <+63>:    mov    eax, 0x0
   0x0000000000400571 <+68>:    call   0x400410, <printf@plt>
   0x0000000000400576 <+73>:    mov    eax, 0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

接下来，通过“finish”指令运行 printf()之后的程序。余下的指令会清空 EAX 寄存器，可以注意到那时 EAX 已经为零。现在，RIP 指针指向 LEAVE 指令，就是 main()函数的倒数第二条指令。

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628"a=%d;b=%d;c=%d;d=%d; e=%d;f=%d;g=%d;h=%d\n") at printf.c:29
a=1;b=2;c=3;d=4;e=5;f=6;g=7;h=8
main () at2.c:6
6               return0;
Value returned is $1=39
(gdb) next
7       };
(gdb) info registers
rax            0x0     0
rbx            0x0     0
rcx            0x26    38
rdx            0x7ffff7dd59f0 140737351866864
rsi            0x7fffffd9     2147483609
rdi            0x0            0
rbp            0x7fffffffdf60 0x7fffffffdf60
rsp            0x7fffffffdf40 0x7fffffffdf40
r8             0x7ffff7dd26a0 140737351853728
r9             0x7ffff7a60134 140737348239668
r10            0x7fffffffd5b0 140737488344496
r11            0x7ffff7a95900 140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040 140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

6.2 ARM

6.2.1 ARM 模式下传递 3 个参数

ARM 系统在传递参数时，通常会进行拆分：把前 4 个参数传递给 R0～R3 寄存器，然后利用栈传递其余的参数。在获取（组装）参数时，遵循的函数调用约定是 fastcall 约定或 win64 约定。有关这两种约定的的详细介绍，请参照 64.3 节和 64.5.1 节。

32 位 ARM 系统

非经优化的 Keil+ARM 模式

指令清单 6.6 非经优化的 Keil 6/2013（ARM 模式）

```
.text:00000000 main
.text:00000000 10 40 2D E9 STMFD  SP!, {R4,LR}
.text:00000004 03 30 A0 E3 MOV    R3, #3
.text:00000008 02 20 A0 E3 MOV    R2, #2
.text:0000000C 01 10 A0 E3 MOV    R1, #1
.text:00000010 08 00 8F E2 ADR    R0, aADBDCD;      "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB BL     __2printf
.text:00000018 00 00 A0 E3 MOV    R0, #0            ; return 0
.text:0000001C 10 80 BD E8 LDMFD  SP!, {R4,PC}
```

可见，R0～R3 寄存器依次负责传递参数。其中，R0 寄存器用来传递格式化字符串，R1 寄存器传递 1，R2 寄存器传递 2，R3 寄存器传递 3。

0x18 处的指令将 R0 寄存器置零，对应着源代码中的“return 0”。

这段汇编指令中规中矩。

即使开启了优化选项，Keil 6/2013 生成的汇编指令也完全相同。

开启了优化选项的 Keil 6/2013 (Thumb 模式)

指令清单 6.7　开启了优化选项的 Keil 6/2013 (Thumb 模式)

```
.text:00000000 main
.text:00000000 10 B5       PUSH   {R4, LR}
.text:00000002 03 23       MOVS   R3, #3
.text:00000004 02 22       MOVS   R2, #2
.text:00000006 01 21       MOVS   R1, #1
.text:00000008 02 A0       ADR    R0, aADBDCD;"a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8 BL     __2printf
.text:0000000E 00 20       MOVS   R0, #0
.text:00000010 10 BD       POP    {R4,PC}
```

这段代码和前面那段 ARM 程序没有太多差别。

开启优化选项的 Keil 6/2013 (ARM 模式) + 无返回值

我们对源代码略做修改，删除“return 0”的语句：

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

相应的汇编指令就会出现显著的差别：

```
.text:00000014 main
.text:00000014 03 30 A0 E3   MOV    R3, #3
.text:00000018 02 20 A0 E3   MOV    R2, #2
.text:0000001C 01 10 A0 E3   MOV    R1, #1
.text:00000020 1E 0E 8F E2   ADR    R0, aADBDCD    ;"a=%d;b=%d;c=%d\n"
.text:00000024 CB 18 00 EA   B      __2printf
```

在其余优化选项（-O3）之后，把源代码编译为 ARM 模式的代码。这次，程序的最后一条指令变成了 B 指令，不再使用之前的 BL 指令。另外，与前面那个没有启用优化选项的例子相比，本例并没有出现保存 R0 和 LR 寄存器的函数序言或函数尾声。这形成了另一个显著的差异。B 指令仅仅将程序跳转到另一个地址，不会根据 LR 寄存器的值进行返回。大体上说，它和 x86 平台的 JMP 指令非常相似。为什么编译器会如此处理呢？实际上，这些指令与前面（未启用优化选项）的运行结果相同。主要原因有两个：

（1）栈和 SP（Stack Pointer）都没有发生变化。

（2）调用 printf()函数是程序的最后一条指令；调用之后程序再无其他操作。

即使未启用优化选项，在完成 printf()函数的作业之后，程序只是要返回到 LR 寄存器里存储的返回地址而已。LR 的值并没有因为调用 printf()函数而发生变化，而且程序也没有调用 printf()函数之外的函数。因为没有指令会修改 LR 的值，所以程序不必保存 LR 的状态。另外，在调用这个函数之后，程序也没有其他操作。故而编译器进行了相应的优化。

当程序最后的语句是调用另外一个函数时，编译器通常都会进行这种优化。

本书 13.1.1 节的“指令清单 13.2”再次出现了这种优化技术。

ARM64

非经优化的 GCC (Linaro) 4.9

指令清单 6.8 非经优化的 GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; save FP and LR in stack frame:
        stp     x29, x30, [sp, -16]!
; set stack frame (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; restore FP and LR
        ldp     x29, x30, [sp], 16
        ret
```

第一条 STP(Store Pair)指令把 FP(X29)和 LR(X30)的值推送入栈。第二条“ADD X29, SP, 0”指令构成栈帧，它只是把 SP 的值复制给 X29。

后面的“ADDRP/ADD”指令对构建了字符串的指针。

在传递给 printf()的格式化字符串里，%d 是 32 位 int 整型数据。所以，程序使用了寄存器的 32 位存储后面的数据 1、2、3。

即使启用了 GCC (Linaro) 4.9 的优化选项，它生成的指令也与此相同。

6.2.2 ARM 模式下传递 8 个参数

我们对 6.1.2 节的例子稍做修改：

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

Optimizing Keil 6/2013：ARM 模式

```
.text:00000028                 main
.text:00000028
.text:00000028                 var_18 = -0x18
.text:00000028                 var_14 = -0x14
.text:00000028                 var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB  BL     __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

程序分为以下几个部分。

- 函数序言：

  第一条指令“STR LR, [SP,#var_4]!”将 LR 寄存器的值推送入栈。在后面调用 printf()函数的时候，程序会修改这个寄存器的值。指令中的感叹号表示这属于预索引（pre-index）指令。具体说来，它会首先将 SP 的值减去 4，然后再把 LR 的值保存在 SP 所指向的地址。这条指令与 x86 平台的 PUSH 指令十分类似。有关 PUSH 指令的详细介绍，请参见本书的 28.2 节。

  第二条指令“SUB SP, SP, #0x14”将修改栈指针 SP，以便在栈内分配 0x14（即 20）字节的存储空间。在后续的操作中，程序会传递 5 个 32 位参数，所以此时需要分配 5×4=20 字节的存储空间。而函数所需的前 4 个参数则是由寄存器负责传递。

- 使用栈传递 5、6、7、8：

  参数 5、6、7、8 分别被存储到 R0、R1、R2、R3 寄存器，然后通过“ADD R12, SP, #0x18+var_14”指令把栈的指针地址写到 R12 寄存器里，以供后续指令进行入栈操作。var_14 是 IDA 创建的汇编宏，其数值等于−0x14。这种“var_?”形式的宏出现在与栈有关的操作指令上时，用于标注栈中的局部变量。最终，R12 寄存器里将会放入 SP＋4。

  后面的“STMIA R12, {R0-R3}”指令把 R0～R3 寄存器中的内容写到 R12 寄存器的值所表示的内存地址上。STMIA 是“Store Multiple Increment After”的缩写。顾名思义，每写入一个寄存器的值，R12 的值就会加 4。

- 通过栈传递数值 4：

  MOV 指令向 R0 寄存器里写入数值“4”。然后，“STR R0, [SP,#0x18+var_18]”指令把 R0 寄存器的值也存储于栈中。由于 var_18 就是−0x18，所以偏移量的值最终为 0。可见，R0 寄存器里的值将会写到 SP 所指向的内存地址。

- 通过寄存器传递 1、2、3：

  传递给 printf()函数的前三个参数 *a*、*b*、*c*（分别为 1、2、3）通过 R1、R2、R3 寄存器传递给 printf()函数。在此之前，其他的 5 个数字都已经推送入栈。在传递格式化字符串之后，被调用的 printf()函数就可以调用它所需的全部参数了。

- 调用 printf()函数。

- 函数尾声：

  “ADD SP, SP, #0x14”指令把栈指针 SP 还原为调用 printf()之前的状态，起到释放栈空间的作用。当然，栈内原有的数据不会被清除或者置零，仍然存在相应地址上。在执行其他函数时，这些数据将被复写。在程序的最后，“LDR PC, [SP+4+var_4],#4”从栈中提取 LR 寄存器的值，把它传递给 PC 寄存器。接下来程序将会跳转到那里。这将结束整个程序。这条指令没有出现感叹号，属于“延迟索引/post-index”指令。也就是说，它先把[SP+4+var_4]（即［SP］）传递给 PC，然后再做 SP=SP−4 的运算。为什么 IDA 以这种风格显示这条指令？IDA 以此充分展现栈的内存存储布局，var_4 是局部栈里保管 LR 的数据空间。这条指令与 x86 平台的 POP 指令十分类似。

Optimizing Keil 6/2013: Thumb 模式

```
.text:0000001C                printf_main2
.text:0000001C
.text:0000001C                var_18 = -0x18
.text:0000001C                var_14 = -0x14
.text:0000001C                var_8  = -8
.text:0000001C
.text:0000001C 00 B5           PUSH    {LR}
.text:0000001E 08 23           MOVS    R3, #8
.text:00000020 85 B0           SUB     SP, SP, #0x14
.text:00000022 04 93           STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22           MOVS    R2, #7
.text:00000026 06 21           MOVS    R1, #6
```

```
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"…
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E              loc_3E  ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

Thumb 模式的代码与 ARM 模式的代码十分相似，但是参数入栈的顺序不同。即，Thumb 模式会在第一批次将 8 推送入栈、第二批次将 7、6、5 推送入栈，而在第三批次将 4 推送入栈。

Optimizing Xcode 4.6.3 (LLVM)：ARM 模式

```
__text:0000290C              _printf_main2
__text:0000290C
__text:0000290C              var_1C = -0x1C
__text:0000290C              var_C  = -0xC
__text:0000290C
__text:0000290C  80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910  0D 70 A0 E1   MOV    R7, SP
__text:00002914  14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918  70 05 01 E3   MOV    R0, #0x1570
__text:0000291C  07 C0 A0 E3   MOV    R12, #7
__text:00002920  00 00 40 E3   MOVT   R0, #0
__text:00002924  04 20 A0 E3   MOV    R2, #4
__text:00002928  00 00 8F E0   ADD    R0, PC, R0
__text:0000292C  06 30 A0 E3   MOV    R3, #6
__text:00002930  05 10 A0 E3   MOV    R1, #5
__text:00002934  00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938  0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C  08 90 A0 E3   MOV    R9, #8
__text:00002940  01 10 A0 E3   MOV    R1, #1
__text:00002944  02 20 A0 E3   MOV    R2, #2
__text:00002948  03 30 A0 E3   MOV    R3, #3
__text:0000294C  10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text:00002950  A4 05 00 EB   BL     _printf
__text:00002954  07 D0 A0 E1   MOV    SP, R7
__text:00002958  80 80 BD E8   LDMFD  SP!, {R7, PC}
```

这段汇编代码与前面的代码十分雷同，不同的是 STMFA (Store Multiple Full Ascending)指令。它是 STMIB (Store Multiple Increment Before)的同义词。它们首先增加 SP 指针的值，然后将数据推送入栈；而不是先入栈，再调整 SP 指针。[①]

虽然这些指令表面看来杂乱无章，但是似乎这就是 xcode 编译出来的程序的一种特点。例如，在地址 0x2918、0x2920、0x2928 处，R0 寄存器的相关操作似乎可以在一处集中处理。不过，这种分散布局是编译器针对并行计算而进行的优化。通常，处理器会尝试着并行处理那些相邻的指令。以“MOVT R0, #0”和“ADD R0, PC, R0”为例——这两条指令都是操作 R0 寄存器的指令，若集中在一起就无法并行计算。另一方面，“MOVT R0, #0”和“MOV R2, #4”之间不存在这种资源冲突，可以同时执行。想必是出于这种设计，编译器才会尽可能地进行这种处理吧。

Optimizing Xcode 4.6.3 (LLVM)：Thumb-2 模式

```
__text:00002BA0              _printf_main2
```

① 指针指向的地址必须有数据，这就是 full stack 的涵义，也是它们与 empty stack 的区别。

```
_text:00002BA0
_text:00002BA0                  var_1C = -0x1C
_text:00002BA0                  var_18 = -0x18
_text:00002BA0                  var_C  = -0xC
_text:00002BA0
_text:00002BA0 80 B5             PUSH      {R7,LR}
_text:00002BA2 6F 46             MOV       R7, SP
_text:00002BA4 85 B0             SUB       SP, SP, #0x14
_text:00002BA6 41 F2 D8 20       MOVW      R0, #0x12D8
_text:00002BAA 4F F0 07 0C       MOV.W     R12, #7
_text:00002BAE C0 F2 00 00       MOVT.W    R0, #0
_text:00002BB2 04 22             MOVS      R2, #4
_text:00002BB4 78 44             ADD       R0, PC ; char *
_text:00002BB6 06 23             MOVS      R3, #6
_text:00002BB8 05 21             MOVS      R1, #5
_text:00002BBA 0D F1 04 0E       ADD.W     LR, SP, #0x1C+var_18
_text:00002BBE 00 92             STR       R2, [SP,#0x1C+var_1C]
_text:00002BC0 4F F0 08 09       MOV.W     R9, #8
_text:00002BC4 8E E8 0A 10       STMIA.W   LR, {R1,R3,R12}
_text:00002BC8 01 21             MOVS      R1, #1
_text:00002BCA 02 22             MOVS      R2, #2
_text:00002BCC 03 23             MOVS      R3, #3
_text:00002BCE CD F8             STR.W     R9, [SP,#0x1C+var_C]
_text:00002BD2 01 F0 0A EA       BLX       _print5
_text:00002BD6 05 B0             ADD       SP, SP, #0x14
_text:00002BD8 80 BD             POP       {R7,PC}
```

与 ARM 模式编译出的代码相比，这段代码存在着明显的 Thumb 指令的特征。除此之外，ARM 模式和 Thumb 模式编译出的代码并无实际区别。

ARM64

Non-optimizing GCC (Linaro) 4.9

指令清单 6.9　Non-optimizing GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; grab more space in stack:
        sub     sp, sp, #32
; save FP and LR in stack frame:
        stp     x29, x30, [sp,16]
; set stack frame (FP=SP):
        add     x29, sp, 16
        adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0, :lo12:.LC2
        mov     w1, 8    ; 9th argument
        str     w1, [sp]  ; store 9th argument in the stack
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; restore FP and LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

X-或 W-寄存器传递函数的前 8 个参数［参见 ARM13c］。字符串指针使用 64 位寄存器，所以它使用整

个 X0 寄存器。所有的其他参数都属于 32 位整型数据，可由寄存器的低 32 位/即 W-寄存器传递。程序使用栈来传递第九个参数（数值 8）。CPU 的寄存器总数有限，所以寄存器往往不足以传递全部参数。

启用优化选项之后，GCC （linaro）4.9 生成的代码与此相同。

6.3 MIPS

6.3.1 传递 3 个参数

Optimizing GCC 4.4.5

在 MIPS 平台上编译“Hello, world!”程序，编译器不会使用 puts()函数替代 printf()函数，而且它会使用$5～$7 寄存器（即$A0～$A2）传递前 3 个参数。

这 3 个寄存器都是“A-”开头的寄存器，因为它们就是负责传递参数（arguments）的寄存器。

指令清单 6.10 Optimizing GCC 4.4.5 (汇编输出)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; load address of printf():
        lw      $25,%call16(printf)($28)
; load address of the text string and set 1st argument of printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; set 2nd argument of printf():
        li      $5,1            # 0x1
; set 3rd argument of printf():
        li      $6,2            # 0x2
; call printf():
        jalr    $25
; set 4th argument of printf() (branch delay slot):
        li      $7,3            # 0x3

; function epilogue:
        lw      $31,28($sp)
; set return value to 0:
        move    $2,$0
; return
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot
```

指令清单 6.11 Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10      = -0x10
.text:00000000 var_4       = -4
.text:00000000
; function prologue:
.text:00000000             lui   $gp, (__gnu_local_gp >> 16)
.text:00000004             addiu $sp, -0x20
.text:00000008             la    $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C             sw    $ra, 0x20+var_4($sp)
.text:00000010             sw    $gp, 0x20+var_10($sp)
; load address of printf():
.text:00000014             lw    $t9, (printf & 0xFFFF)($gp)
```

```
; load address of the text string and set 1st argument of printf():
.text:00000018              la    $a0, $LC0             # "a=%d; b=%d; c=%d"
; set 2nd argument of printf():
.text:00000020              li    $a1, 1
; set 3rd argument of printf():
.text:00000024              li    $a2, 2
; call printf():
.text:00000028              jalr  $t9
; set 4th argument of printf() (branch delay slot):
.text:0000002C              li    $a3, 3
; function epilogue:
.text:00000030              lw    $ra, 0x20+var_4($sp)
; set return value to 0:
.text:00000034              move  $v0, $zero
; return
.text:00000038              jr    $ra
.text:0000003C              addiu #sp, 0x20 ; branch delay slot
```

IDA 没有显示 0x1C 的指令。实际上 0x18 是“LUI”和“ADDIU”两条指令，IDA 把它们显示为单条的伪指令，占用了 8 个字节。

Non-optimizing GCC 4.4.5

如果不启用编译器的优化选项，那么 GCC 输出的指令会详细得多。

指令清单 6.12　Non-optimizing GCC 4.4.5（汇编输出）

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; function prologue:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; load address of the text string
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; set 1st argument of printf():
        move    $4,$2
; set 2nd argument of printf():
        li      $5,1                    # 0x1
; set 3rd argument of printf():
        li      $6,2                    # 0x2
; set 4th argument of printf():
        li      $7,3                    # 0x3
; get address of printf():
        lw      $2,%call16(printf)($28)
        nop
; call printf():
        move    $25,$2
        jalr    $25
        nop

; function epilogue:
        lw      $28,16($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; return
        j       #31
        nop
```

指令清单 6.13 Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10      = -0x10
.text:00000000 var_8       = -8
.text:00000000 var_4       = -4
.text:00000000
; function prologue:
.text:00000000             addiu     $sp, -0x20
.text:00000004             sw        $ra, 0x20+var_4($sp)
.text:00000008             sw        $fp, 0x20+var_8($sp)
.text:0000000C             move      $fp, $sp
.text:00000010             la        $gp, __gnu_local_gp
.text:00000018             sw        $gp, 0x20+var_10($sp)
; load address of the text string:
.text:0000001C             la        $v0, aADBDCD  # "a=%d; b=%d; c=%d"
; set 1st argument of printf():
.text:00000024             move      $a0, $v0
; set 2nd argument of printf():
.text:00000028             li        $a1, 1
; set 3rd argument of printf():
.text:0000002C             li        $a2, 2
; set 4th argument of printf():
.text:00000030             li        $a3, 3
; get address of printf():
.text:00000034             lw        $v0, (printf & 0xFFFF)($gp)
.text:00000038             or        $at, $zero
; call printf():
.text:0000003C             move      $t9, $v0
.text:00000040             jalr      $t9
.text:00000044             or        $at, $zero ; NOP
; function epilogue:
.text:00000048             lw        $gp, 0x20+var_10($fp)
; set return value to 0:
.text:0000004C             move      $v0, $zero
.text:00000050             move      $sp, $fp
.text:00000054             lw        $ra, 0x20+var_4($sp)
.text:00000058             lw        $fp, 0x20+var_8($sp)
.text:0000005C             addiu     $sp, 0x20
; return
.text:00000060             jr        $ra
.text:00000064             or        $at, $zero ; NOP
```

6.3.2 传递 9 个参数

我们再次使用 6.1.2 节中的例子，演示 9 个参数的传递。

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

Optimizing GCC 4.4.5

在传递多个参数时，MIPS 会使用$A0~$A3 传递前 4 个参数，使用栈传递其余的参数。这种平台主要采用一种名为“O32”的函数调用约定。实际上大多数 MIPS 系统都采用这种约定。如果采用了其他的函数调用约定，例如 N32 约定，寄存器的用途则会有不同的设定。

下面指令中的“SW”是“Store Word”的缩写，用以把寄存器的值写入内存。MIPS 的指令集很小，没有把数据直接写入内存地址的那类指令。当需要进行这种操作时，就不得不组合使用 LI/SW 指令。

指令清单 6.14　Optimizing GCC 4.4.5 (汇编输出)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
; pass 5th argument in stack:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; pass 6th argument in stack:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; pass 7th argument in stack:
        li      $2,6                    # 0x6
      sw        $2,24($sp)
; pass 8th argument in stack:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; pass 1st argument in $a0:
        lui     $4,%hi($LC0)
; pass 9th argument in stack:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; pass 2nd argument in $a1:
        li      $5,1                    # 0x1
; pass 3rd argument in $a2:
        li      $6,2                    # 0x2
; call printf():
        jalr    $25
; pass 4th argument in $a3 (branch delay slot):
        li      $7,3                    # 0x3

; function epilogue:
        lw      $31,52($sp)
; set return value to 0:
        move    $2,$0
; return
        j       $31
        addiu   $sp,$sp,56 ; branch delay slot
```

指令清单 6.15　Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28      = -0x28
.text:00000000 var_24      = -0x24
.text:00000000 var_20      = -0x20
.text:00000000 var_1C      = -0x1C
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_4       =-4
.text:00000000
; function prologue:
.text:00000000             lui         $gp, (__gnu_local_gp >> 16)
.text:00000004             addiu       $sp, -0x38
.text:00000008             la          $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C             sw          $ra, 0x38+var_4($sp)
.text:00000010             sw          $gp, 0x38+var_10($sp)
; pass 5th argument in stack:
.text:00000014             li          $v0, 4
.text:00000018             sw          $v0, 0x38+var_28($sp)
; pass 6th argument in stack:
```

```
.text:0000001C             li        $v0, 5
.text:00000020             sw        $v0, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000024             li        $v0, 6
.text:00000028             sw        $v0, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000002C             li        $v0, 7
.text:00000030             lw        $t9, (printf & 0xFFFF)($gp)
.text:00000034             sw        $v0, 0x38+var_1C($sp)
; prepare 1st argument in $a0:
.text:00000038             lui       $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
; pass 9th argument in stack:
.text:0000003C             li        $v0, 8
.text:00000040             sw        $v0, 0x38+var_18($sp)
; pass 1st argument in $a1:
.text:00000044             la        $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
; pass 2nd argument in $a1:
.text:00000048             li        $a1, 1
; pass 3rd argument in $a2:
.text:0000004C             li        $a2, 2
; call printf():
.text:00000050             jalr      $t9
; pass 4th argument in $a3 (branch delay slot):
.text:00000054             li        $a3, 3
; function epilogue:
.text:00000058             lw        $ra, 0x38+var_4($sp)
; set return value to 0:
.text:0000005C             move      $v0, $zero
; return
.text:00000060             jr        $ra
.text:00000064             addiu     $sp, 0x38 ; branch delay slot
```

Non-optimizing GCC 4.4.5

关闭优化选项后，GCC 会生成较为详细的指令。

指令清单 6.16 Non-optimizing GCC 4.4.5（汇编输出）

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; pass 5th argument in stack:
        li      $3,4            # 0x4
        sw      $3,16($sp)
; pass 6th argument in stack:
        li      $3,5            # 0x5
        sw      $3,20($sp)
; pass 7th argument in stack:
        li      $3,6            # 0x6
        sw      $3,24($sp)
; pass 8th argument in stack:
        li      $3,7            # 0x7
        sw      $3,28($sp)
; pass 9th argument in stack:
        li      $3,8            # 0x8
        sw      $3,32($sp)
; pass 1st argument in $a0:
        move    $4,$2
```

```
; pass 2nd argument in $a1:
        li      $5,1            # 0x1
; pass 3rd argument in $a2:
        li      $6,2            # 0x2
; pass 4th argument in $a3:
        li      $7,3            # 0x3
; call printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; function epilogue:
        lw      $28,40($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; return
        j       $31
        nop
```

指令清单 6.17　Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28      = -0x28
.text:00000000 var_24      = -0x24
.text:00000000 var_20      = -0x20
.text:00000000 var_1C      = -0x1C
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_8       = -8
.text:00000000 var_4       = -4
.text:00000000
;  function prologue:
.text:00000000             addiu $sp, -0x38
.text:00000004             sw    $ra, 0x38+var_4($sp)
.text:00000008             sw    $fp, 0x38+var_8($sp)
.text:0000000C             move  $fp, $sp
.text:00000010             la    $gp, __gnu_local_gp
.text:00000018             sw    $gp, 0x38+var_10($sp)
.text:0000001C             la    $v0, aADBDCDDDEDFDGD # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%" ...
; pass 5th argument in stack:
.text:00000024             li    $v1, 4
.text:00000028             sw    $v1, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000002C             li    $v1, 5
.text:00000030             sw    $v1, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000034             li    $v1, 6
.text:00000038             sw    $v1, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000003C             li    $v1, 7
.text:00000040             sw    $v1, 0x38+var_1C($sp)
; pass 9th argument in stack:
.text:00000044             li    $v1, 8
.text:00000048             sw    $v1, 0x38+var_18($sp)
; pass 1st argument in $a0:
.text:0000004C             move  $a0, $v0
; pass 2nd argument in $a1:
.text:00000050             li    $a1, 1
; pass 3rd argument in $a2:
.text:00000054             li    $a2, 2
```

```
; pass 4th argument in $a3:
.text:00000058             li   $a3, 3
; call printf():
.text:0000005C             lw   $v0, (printf & 0xFFFF)($gp)
.text:00000060             or   $at, $zero
.text:00000064             move $t9, $v0
.text:00000068             jalr $t9
.text:0000006C             or   $at, $zero ; NOP
; function epilogue:
.text:00000070             lw   $gp, 0x38+var_10($fp)
; set return value to 0:
.text:00000074             move $v0, $zero
.text:00000078             move $sp, $fp
.text:0000007C             lw   $ra, 0x38+var_4($sp)
.text:00000080             lw   $fp, 0x38+var_8($sp)
.text:00000084             addiu $sp, 0x38
; return
.text:00000088             jr   $ra
.text:0000008C             or   $at, $zero ; NOP
```

6.4 总结

调用函数的时候，程序的传递过程大体如下。

指令清单 6.18 x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; modify stack pointer (if needed)
```

指令清单 6.19 x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)
```

指令清单 6.20 x64 (GCC)

```
MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7th, 8th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)
```

指令清单 6.21 ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; pass 5th, 6th argument, etc, in stack (if needed)
BL function
```

```
; modify stack pointer (if needed)
```

指令清单 6.22　ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; pass 9th, 10th argument, etc, in stack (if needed)
BL CALL function
; modify stack pointer (if needed)
```

指令清单 6.23　MIPS（O32 调用约定）

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; pass 5th, 6th argument, etc, in stack (if needed)
LW temp_reg, address of function
JALR temp_reg
```

6.5　其他

在 x86、x64、ARM 和 MIPS 平台上，程序向函数传递参数的方法各不相同。这种差异性表明，函数间传递参数的方法与 CPU 的关系并不是那么密切。如果付诸努力，人们甚至可以开发出一种不依赖数据栈即可传递参数的超级编译器。

为了方便记忆，在采用 O32 调用约定时，MIPS 平台的 4 号～7 号寄存器也叫作$A0~$A7 寄存器。实际上，编程人员完全可以使用$ZERO 之外的其他寄存器传递数据，也可以采取其他的函数调用约定。

CPU 完全不在乎程序使用何种调用约定。

汇编语言的编程新手可能采取五花八门的方法传递参数。虽然他们基本上都是利用寄存器传递参数，但是传递参数的顺序往往不那么讲究，甚至可能通过全局变量传递参数。不过，这样的程序也能正常运行。

第 7 章　scanf()

本章演示 scanf()函数。

7.1　演示案例

```
#include<stdio.h>

intmain()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

好吧，我承认时下的程序如果大量使用 scanf()就不会有什么前途。本文仅用这个函数演示整形数据的指针的传递过程。

7.1.1　指针简介

在计算机科学里，“指针”属于最基础的概念。如果直接向函数传递大型数组、结构体或数据对象，程序的开销就会很大。毫无疑问，使用指针将会降低开销。不过指针的作用不只如此：如果不使用指针，而是由调用方函数直接传递数组或结构体这种大型数据（同时还要返回这些数据），那么参数的传递过程将会复杂得出奇。所以，调用方函数只负责传递数组或结构体的地址，让被调用方函数处理地址里的数据，无疑是最简单的做法。

在 C/C++的概念中，指针就是描述某个内存地址的数据。

x86 系统使用体系 32 位数字（4 字节数据）描述指针；x64 系统则使用 64 位数字（8 字节数据）。从数据空间来看，64 位系统的指针比 32 位系统的指针大了一倍。当人们逐渐从 x86 平台开发过渡到 x86-64 平台开发的时候，不少人因为难以适应而满腹牢骚。

程序人员确实可在所有的程序里仅仅使用无类型指针这一种指针。例如，在使用 C 语言 memcpy()函数、在内存中复制数据的时候，程序员完全不必知道操作数的数据类型，直接使用 2 个 void*指针复制数据。这种情况下，目标地址里数据的数据类型并不重要，重要的是存储数据的空间大小。

指针还广泛应用于多个返回值的传递处理。本书的第 10 章会详细介绍这部分内容。scanf()函数就可以返回多个值。它不仅能够分析调用方法传递了多少个参数，而且还能读取各个参数的值。

在 C/C++的编译过程里，编译器只会在类型检查的阶段才会检查指针的类型。在编译器生成的汇编程序里，没有指针类型的任何信息。

7.1.2　x86

MSVC

使用 MSVC 2010 编译上述源代码，可得到下述汇编指令：

```
CONST    SEGMENT
```

```
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$=-4            ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X: '
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; return 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main   ENDP
_TEXT   ENDS
```

变量 *x* 是局部变量。

C/C++标准要求：函数体内部应当可以访问局部变量，且函数外部应该访问不到函数内部的局部变量。演变至今，人们不约而同地利用数据栈来存储局部变量。虽然分配局部变量的方法不只这一种，但是所有面向 x86 平台的编译器都约定俗成般地采用这种方法存储局部变量。

在函数序言处有一条“PUSH ECX”指令。因为函数尾声处没有对应的“POP ECX”指令，所以它的作用不是保存 ECX 的值。

实际上，这条指令在栈内分配了 4 字节的空间、用来存储局部变量 *x*。

汇编宏_x$ （其值为−4）用于访问局部变量 *x*，而 EBP 寄存器用来存储栈当前帧的指针。

在函数运行的期间，EBP 一直指向当前的栈帧（stack frame）。这样，函数即可通过 EBP+offset 的方式访问本地变量、以及外部传入的函数参数。

ESP 也可以用来访问本地变量，获取函数所需的运行参数。不过 ESP 的值经常发生变化，用起来并不方便。函数在启动之初就会利用 EBP 寄存器保存 ESP 寄存器的值。这就是为了确保在函数运行期间保证 EBP 寄存器存储的原始 ESP 值固定不变。

在 32 位系统里，典型的栈帧（stack frame）结构如下表所示。

.....	.....
EBP-8	局部变量#2，IDA 标记为 var_8
EBP-4	局部变量#1，IDA 标记为 var_4
EBP	EBP 的值
EBP+4	返回地址 Return address
EBP+8	函数参数#1，IDA 标记为 arg_0

续表

EBP+0xC	函数参数#2，IDA 标记为 arg_4
EBP+0x10	函数参数#3，IDA 标记为 arg_8
……	……

本例中的 scanf()有两个参数。

第一个参数是一个指针，它指向含有“%d”的格式化字符串。第二个参数是局部变量 *x* 的地址。

首先，“lea eax, DWORD PTR _x$[ebp]”指令将变量 *x* 的地址放入 EAX 寄存器。“lea”是“load effective address”的缩写，能够将源操作数（第二个参数）给出的有效地址（offset）传送到目的寄存器（第一个参数）之中。①

此处，LEA 将 EBP 寄存器的值与宏_x$求和，然后使用 EAX 寄存器存储这个计算结果；也就是等同于“ lea eax, [ebp−4]”。

就是说，程序会将 EBP 寄存器的值减去 4，并把这个运算结果存储于 EAX 寄存器。把 EAX 寄存器的值推送入栈之后，程序才开始调用 scanf()函数。

在调用 printf()函数之前，程序传给它第一个参数，即格式化字符串“You entered %d...\n”的指针。

printf()函数所需的第二个参数由“mov ecx,[ebp−4]”指令间接取值。传递给 ecx 的值是 ebp−4 所指向的地址的值（即变量 *x* 的值），而不是 ebp−4 所表达的地址。

此后的指令把 ECX 推送入栈，然后启动 printf()函数。

7.1.3 MSVC + OllyDbg

现在使用 OllyDbg 调试上述例子。加载程序之后，一直按 F8 键单步执行，等待程序退出 ntdll.dll、进入我们程序的主文件。然后向下翻滚滚轴，查找 main()主函数。在 main()里面点中第一条指令“PUSH EBP”，并在此处按下 F2 键设置断点。接着按 F9 键，运行断点之前的指令。

我们一起来跟随调试器查看变量 *x* 的计算指令，如图 7.1 所示。

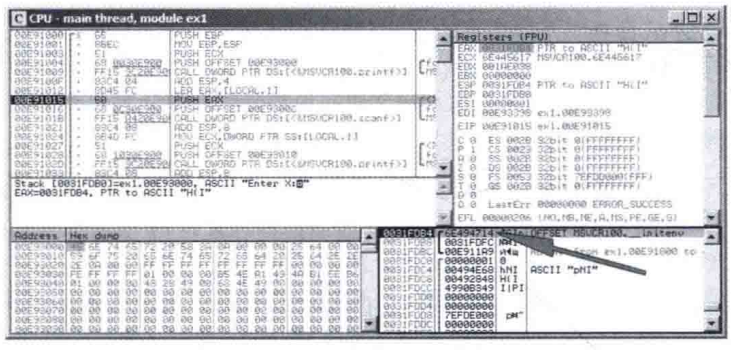


图 7.1 OllyDbg：局部变量 *x* 的赋值过程

在这个界面里，我们在寄存器的区域内用右键单击 EAX 寄存器，然后选择“Follow in stack”。如此一来，OllyDbg 就会在栈窗口里显示栈地址和栈内数据，以便我们清楚地观察栈里的局部变量。图中红箭头所示的就是栈里的数据。其中，在地址 0x6E494714 处的数据就是脏数据。在下一时刻，PUSH 指令会把数据存储到栈里的下一个地址。接下来，在程序执行完 scanf()函数之前，我们一直按 F8 键。在执行 scanf()函数的时候，我们要在运行程序的终端窗口里输入数据，例如 123，如图 7.2 所示。

图 7.2 控制台窗口

scanf()函数的执行之后的情形如图 7.3 所示。

EAX 寄存器里存有函数的返回值 1。这表示它成功地读取了 1 个值。我们可以在栈里找到局部变量的

① 参见附录 A.6.2。

地址，其数值为 0x7B（即数字 123）。

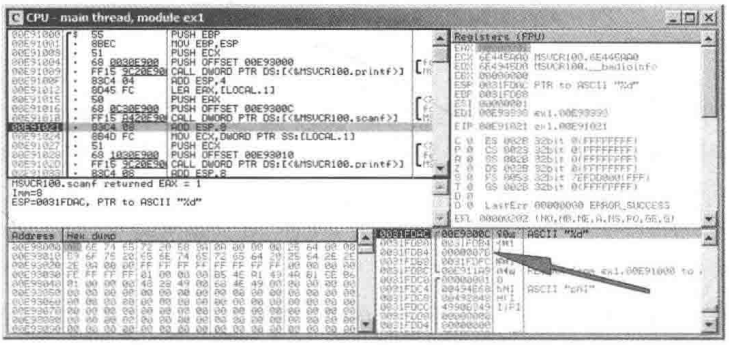


图 7.3　OllyDbg：运行 scanf()之后

这个值将通过栈传递给 ECX 寄存器，然后再次通过栈传递给 printf()函数，如图 7.4 所示。

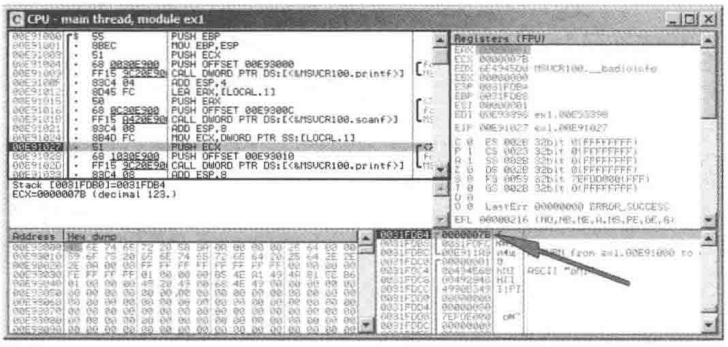


图 7.4　OllyDbg：将数值传递给 printf()

GCC

我们在 Linux GCC 4.4.1 下编译这段程序。

```
main        proc near

var_20      = dword ptr -20h
var_1C      = dword ptr -1Ch
var_4       = dword ptr -4

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 20h
            mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
            call    _puts
            mov     eax, offset aD  ; "%d"
            lea     edx, [esp+20h+var_4]
            mov     [esp+20h+var_1C], edx
            mov     [esp+20h+var_20], eax
            call    __isoc99_scanf
            mov     edx, [esp+20h+var_4]
            mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
            mov     [esp+20h+var_1C], edx
            mov     [esp+20h+var_20], eax
            call    _printf
            mov     eax, 0
            leave
            retn
main        endp
```

GCC 把 printf()替换为 puts()，这和 3.4.3 节中的现象相同。

此处，GCC 通过 MOV 指令实现入栈操作，这点和 MSVC 相同。

其他

这个例子充分证明：在编译过程中，编译器会遵循 C/C++源代码的表达式顺序和模块化结构。在 C/C++源代码中，只要两个相邻语句之间没有其他的表达式，那么在生成的机器码中对应的指令之间就不会有其他的指令，而且其执行顺序也与源代码各语句的书写顺序相符。

7.1.4 x64

在编译面向 x64 平台的可执行程序时，由于这个程序的参数较少，编译器会直接使用寄存器传递参数。除此之外，编译过程和 x86 的编译过程没有太大的区别。

MSVC

指令清单 7.1 MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X: ', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d... ', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X: '
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d... '
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
```

GCC

使用 GCC 4.6（启用优化选项－O3）编译上述程序，可得到如下所示的汇编指令。

指令清单 7.2 Optimizing GCC 4.4.6 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
```

```
call    puts
lea     rsi, [rsp+12]
mov     edi, OFFSET FLAT:.LC1 ; "%d"
xor     eax, eax
call    __isoc99_scanf
mov     esi, DWORD PTR [rsp+12]
mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
xor     eax, eax
call    printf

;return 0
xor     eax, eax
add     rsp, 24
ret
```

7.1.5　ARM

Optimizing Keil 6/2013 (Thumb 模式)

```
.text:00000042            scanf_main
.text:00000042
.text:00000042            var_8           = -8
.text:00000042
.text:00000042 08 B5                      PUSH    {R3,LR}
.text:00000044 A9 A0                      ADR     R0, aEnterX    ; "Enter X:\n"
.text:00000046 06 F0 D3 F8                BL      __2printf
.text:0000004A 69 46                      MOV     R1, SP
.text:0000004C AA A0                      ADR     R0, aD         ; "%d"
.text:0000004E 06 F0 CD F8                BL      __0scanf
.text:00000052 00 99                      LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0                      ADR     R0,aYouEnteredD___ ; "You entered%d…\n"
.text:00000056 06 F0 CB F8                BL      __2printf
.text:0000005A 00 20                      MOVS    R0, #0
.text:0000005C 08 BD                      POP     {R3,PC}
```

scanf()函数同样要借助指针传递返回值。在本例里，编译器给它分配了一个整型变量的指针。整型数据占用 4 个字节的存储空间。但是返回数据的内存地址，可以直接放在 CPU 的寄存器中。在生成的汇编代码里，变量 *x* 存在于数据栈中，被 IDA 标记为 var_8，然而，此时程序完全可以直接使用。栈指针 SP 指向的存储空间，没有必要像上述代码那样机械式地调整 SP 分配栈空间。此后，程序把栈指针 SP（*x* 的地址）存入 R1 寄存器，再把格式化字符串的偏移量存入 R0 寄存器，如此一来，scanf()函数就获得了它所需要的所有参数。在此之后，程序使用 LDR 指令把栈中的返回值复制到 R1 寄存器、以供 printf()调用。

即使使用 Xcode LLVM 程序以 ARM 模式编译这段程序，最终生成的汇编代码也没有实质性的区别，所以本书不再演示。

ARM64

指令清单 7.3　Non-optimizing GCC 4.9.1 ARM64

```
 1 .LC0:
 2         .string "Enter X:"
 3 .LC1:
 4         .string "%d"
 5 .LC2:
 6         .string "You entered %d...\n"
 7 scanf_main:
 8 ; subtract 32 from SP, then save FP and LR in stack frame:
 9         stp     x29, x30, [sp, -32]!
10 ; set stack frame (FP=SP)
11         add     x29, sp, 0
12 ; load pointer to the "Enter X:" string:
13         adrp    x0, .LC0
```

```
14          add x0, x0, :lo12:.LC0
15 ; X0=pointer to the "Enter X:" string
16 ; print it:
17          bl      puts
18 ; load pointer to the "%d" string:
19          adrp    x0, .LC1
20          add     x0, x0, :lo12:.LC1
21 ; find a space in stack frame for "x" variable (X1=FP+28):
22          add     x1, x29, 28
23 ; X1=address of "x" variable
24 ; pass the address to scanf() and call it:
25          bl      __isoc99_scanf
26 ; load 32-bit value from the variable in stack frame:
27          ldr     w1, [x29,28]
28 ; W1=x
29 ; load pointer to the "You entered %d...\n" string
30 ; printf() will take text string from X0 and "x" variable from X1 (or W1)
31          adrp    x0, .LC2
32          add     x0, x0, :lo12:.LC2
33          bl      printf
34 ; return 0
35          mov     w0, 0
36 ; restore FP and LR, then add 32 to SP:
37          ldp     x29, x30, [sp], 32
38          ret
```

第 22 行用来分配局部变量 x 的存储空间。当 scanf()函数获取这个地址之后，它就把用户输入的数据传递给这个内存地址。输入的数据应当是 32 位整型数据。第 27 行的指令把输入数据的值传递给 printf()函数。

7.1.6 MIPS

MIPS 编译器同样使用数据栈存储局部变量 x。然后程序就可以通过$sp+24 调用这个变量。scanf()函数获取地址指针之后，通过 LW（Load Word）指令把输入变量存储到这个地址、以传递给 printf()函数。

指令清单 7.4 Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; call puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; call scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; set 2nd argument of scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot

; call printf():
        lw      $28,16($sp)
```

```
; set 2nd argument of printf(),
; load word at address $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; function epilogue:
        lw      $31,36($sp)
; set return value to 0:
        move    $2,$0
; return:
        j       $31
        addiu   $sp,$sp,40 ; branch delay slot
```

IDA 将本程序的栈结构显示如下。

指令清单 7.5 Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_4       = -4
.text:00000000
; function prologue:
.text:00000000             lui     $gp, (__gnu_local_gp >> 16)
.text:00000004             addiu   $sp, -0x28
.text:00000008             la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C             sw      $ra, 0x28+var_4($sp)
.text:00000010             sw      $gp, 0x28+var_18($sp)
; call puts():
.text:00000014             lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018             lui     $a0, ($LC0 >> 16)  # "Enter X:"
.text:0000001C             jalr    $t9
.text:00000020             la      $a0, ($LC0 & 0xFFFF)  # "Enter X:" ; branch delay slot
; call scanf():
.text:00000024             lw      $gp, 0x28+var_18($sp)
.text:00000028             lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C             lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; set 2nd argument of scanf(), $a1=$sp+24:
.text:00000030             addiu   $a1, $sp, 0x28+var_10
.text:00000034             jalr    $t9 ; branch delay slot
.text:00000038             la      $a0, ($LC1 & 0xFFFF)  # "%d"
; call printf():
.text:0000003C             lw      $gp, 0x28+var_18($sp)
; set 2nd argument of printf(),
; load word at address $sp+24:
.text:00000040             lw      $a1, 0x28+var_10($sp)
.text:00000044             lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048             lui     $a0, ($LC2 >> 16) # "You entered %d...\n"
.text:0000004C             jalr    $t9
.text:00000050             la      $a0, ($LC2 & 0xFFFF) # "You entered %d...\n" ; brach delay slot
; function epilogue:
.text:00000054             lw      $ra, 0x28+var_4($sp)
; set return value to 0:
.text:00000058             move    $v0, $zero
; return:
.text:0000005C             jr      $ra
.text:00000060             addiu   $sp, 0x28 ; branch delay slot
```

7.2 全局变量

在本章前文的那个程序里，如果 x 不是局部变量而是全局变量，那会是什么情况？一旦 x 变量成为了

全局变量，函数内部的指令、以及整个程序中的任何部分都可以访问到它。虽然优秀的程序不应当使用全局变量，但是我们应当了解它的技术特点。

```
#include <stdio.h>

//now x is global variable
int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

7.2.1 MSVC：x86

```
_DATA     SEGMENT
COMM      _x:DWORD
$SG2456      DB    'Enter X: ', 0aH, 00H
$SG2457      DB    '%d', 00H
$SG2458      DB    'You entered %d... ', 0aH, 00H
_DATA     ENDS
PUBLIC      _main
EXTRN     __scanf:PROC
EXTRN     __printf:PROC
; Function compile flags: /Odtp
_TEXT     SEGMENT
_main     PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2456
    call    _printf
    add     esp, 4
    push    OFFSET _x
    push    OFFSET $SG2457
    call    _scanf
    add     esp, 8
    mov     eax, DWORD PTR _x
    push    eax
    push    OFFSET $SG2458
    call    _printf
    add     esp, 8
    xor     eax, eax
    pop     ebp
    ret     0
_main     ENDP
_TEXT     ENDS
```

与前文不同的是，*x* 变量的存储空间是数据段（_data 域），反而没有使用数据栈。因此整个程序的所有指令都可以直接访问全局变量 *x*。在可执行文件中，未经初始化的变量不会占用任何存储空间。

在某些指令在变量访问这种未初始化的全局变量的时候，操作系统会分配一段数值为零的地址给它。这是操作系统 VM（虚拟内存）的管理模式所决定的。

如果对上述源代码稍做改动，加上变量初始化的指令：

```
int x=10; //设置默认值
```

那么对应的代码会变为：

```
_DATA   SEGMENT
_x      DD      0aH
...
```

上述指令将初始化 x。其中 DD 代表 DWORD，表示 *x* 是 32 位的数据。

若在 IDA 里打开对 x 进行初始化的可执行文件，我们将会看到数据段的开头部分看到初始化变量 *x*。紧随其后的空间用于存储本例中的字符串。

用 IDA 打开 7.2 节里那个不初始化变量 *x* 的例子，那么将会看到下述内容。

```
.data:0040FA80 _x              dd ?            ; DATA XREF: _main+10
.data:0040FA80                                 ; _main+22
.data:0040FA84 dword_40FA84    dd ?            ; DATA XREF: _memset+1E
.data:0040FA84                                 ; unknown_libname_1+28
.data:0040FA88 dword_40FA88    dd ?            ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                                 ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem           dd ?            ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                                 ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90    dd ?            ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                                 ; __calloc_impl+72
.data:0040FA94 dword_40FA94    dd ?            ; DATA XREF: ___sbh_free_block+2FE
```

这段代码里有很多带“？”标记的变量，这是未初始化的 *x* 变量的标记。这意味着在程序加载到内存之后，操作系统将为这些变量分配空间、并填入数字零[①]。但是在可执行文件里，这些未初始化的变量不占用内存空间。为了方便使用巨型数组之类的大型数据，人们刻意做了这种设定。

7.2.2　MSVC：x86+OllyDbg

我们可以在 OllyDbg 观察程序的数据段里的变量。如图 7.5 所示。

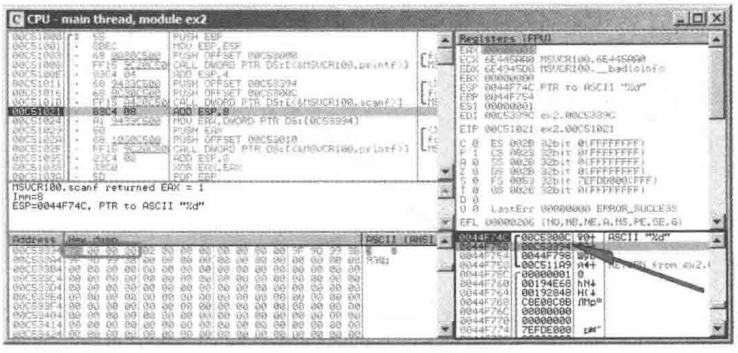


图 7.5　OllyDbug：运行 scanf()函数之后的状态

全局变量 x 出现在数据段里。在调试器执行完 PUSH 指令之后，变量 *x* 的指针推即被推送入栈，我们就可在栈里右键单击 *x* 的地址并选择“Follow in dump”，并在左侧的内存窗口观察它。

在控制台输入 123 之后，栈里的数据将会变成 0x7B（左下窗口的高亮部分）。

您有没有想过，为什么第一个字节是 0x7B？若考虑到数权，此处应该是 00 00 00 7B。可见，这是 x86 系统里低位优先的“小端字节序/LITTLE-ENDIAN”的典型特征。小端字节序属于“字节（顺）序/endianness”的一种，它的第一个字节是数权最低的字节，数权最高的字节会排列在最后。本书将在第 31 章将详细介绍字节序。

此后，EAX 寄存器将存储这个地址里的 32 位值，并将之传递给 printf()函数。

本例中，变量 *x* 的内存地址是 0x00C53394。

在 OllyDbg 里，按下 Alt+M 组合键可查看这个进程的内存映射（process memory map）。如图 7.6 所示，这个地址位于程序 PE 段的.data 域。

① 请参阅 ISO07，6.7.8 节。

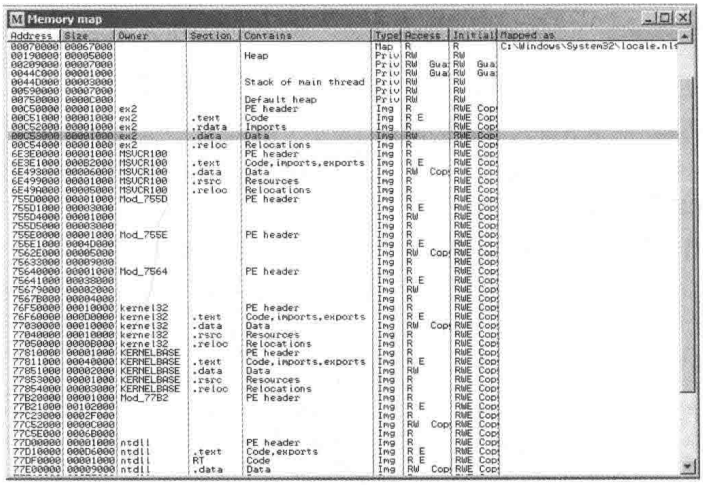


图 7.6 OllyDbg：进程内存映射

7.2.3 GCC：x86

在汇编指令层面，Linux 与 Windows 的编译结果区别不大。它们的区别主要体现在字段（segament）名称和字段属性上：Linux GCC 生成的未初始化变量会出现在_bss 段，对应的 ELF 文件描述了这个字段数据的属性。

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

如果给这个变量分配了初始值，比如说 10，那么这个变量将会出现在_data 段，并且具有下述属性。

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

7.2.4 MSVC：x64

指令清单 7.6 MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X: ', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d... ', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X: '
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d... '
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

这段 x64 代码与 x86 的代码没有明显区别。请注意变量 *x* 的传递过程：scanf()函数通过 LEA 指令获取 *x* 变量的指针；而 printf()函数则是通过 MOV 指令获取 *x* 变量的值。“DWORD PTR”与机器码无关，它是汇编语言的一部分，用来声明后面的变量为 32 位的值，以便 MOV 指令能够正确处理数据类型。

7.2.5　ARM: Optimizing Keil 6/2013 (Thumb 模式)

```
.text:00000000 ; Segment type:  Pure code
.text:00000000                  AREA .text, CODE
...
.text:00000000 main
.text:00000000                  PUSH    {R4,LR}
.text:00000002                  ADR     R0, aEnterX             ; "Enter X:\n"
.text:00000004                  BL      __2printf
.text:00000008                  LDR     R1, =x
.text:0000000A                  ADR     R0, aD                  ; "%d"
.text:0000000C                  BL      __0scanf
.text:00000010                  LDR     R0, =x
.text:00000012                  LDR     R1, [R0]
.text:00000014                  ADR     R0, aYouEnteredD___     ; "You entered %d...\n"
.text:00000016                  BL      __2printf
.text:0000001A                  MOVS    R0, #0
.text:0000001C                  POP     {R4,PC}
...
.text:00000020 aEnterX          DCB "Enter X:",0xA,0            ; DATA XREF: main+2
.text:0000002A                  DCB 0
.text:0000002B                  DCB 0
.text:0000002C off_2C           DCD x                           ; DATA XREF: main+8
.text:0000002C                                                  ; main+10
.text:00000030 aD               DCB "%d",0                      ; DATA XREF: main+A
.text:00000033                  DCB 0
.text:00000034 aYouEnteredD___  DCB "You entered %d...",0xA,0   ; DATA XREF: main+14
.text:00000047                  DCB 0
.text:00000047 ; .text          ends
.text:00000047
...
.data:00000048 ; Segment type:  Pure data
.data:00000048                  AREA .data, DATA
.data:00000048                  ; ORG 0x48
.data:00000048                  EXPORT x
.data:00000048 x                DCD 0xA                         ; DATA XREF: main+8
.data:00000048                                                  ; main+10
.data:00000048 ; .data          ends
```

因为变量 *x* 是全局变量，所以它出现于数据段的“.data”字段里。有的读者可能会问，为什么文本字符串出现在代码段（.text），但是 *x* 变量却出现于数据段（.data）？原因在于 *x* 是变量。顾名思义，变量的值可变、属于一种频繁变化的可变数据。在 ARM 系统里，代码段的程序代码可存储于处理器的 ROM（Read-Only Memory），而可变变量存储于 RAM（Random-Access Memory）中。与 x86/x64 系统相比，ARM 系统的内存寻址能力很有限，可用内存往往十分紧张。在 ARM 系统存在 ROM 的情况下，使用 RAM 内存存储常量则明显是一种浪费。此外，ARM 系统在启动之后 RAM 里的值都是随机数；想要使用 RAM 存储常量，还要单独进行初始化赋值才行。

在后续代码段的指令中，程序给变量 *x* 分配了个指针（即 off_2c）。此后，程序都是通过这个指针对 *x* 变量进行的操作。不这样做的话变量 *x* 可能被分配到距离程序代码段很远的内存空间，其偏移量有可能超过有关寻址指令的寻址能力。Thumb 模式下，ARM 系统的 LDR 指令只能够使用周边 1020 字节之内的变量；即使在 32 位 ARM 模式下，它也只能调用偏移量在±4095 字节之内的变量。这个范围就是变量地址（与调用指令之间）的偏移量的局限。为了保证它的地址离代码段足够近、能被代码调用，就需要就近分配 *x* 变量的地址。由于在链接阶段（linker）*x* 的地址可能会被随意分配，甚至可能被分配到外部内存的地址，所以编译器必须在前期阶段就把 *x* 的地址分配到就近的区域之内。

另外，如果声明某变量为常量/const，Keil 编译程序会把这个变量分配到.constdata 字段。这可能是为

了便于后期 linker 把这个字段与代码段一起放入 ROM。

7.2.6 ARM64

指令清单 7.7 Non-optimizing GCC 4.9.1 ARM64

```
 1          .comm   x,4,4
 2 .LC0:
 3          .string "Enter X:"
 4 .LC1:
 5          .string "%d"
 6 .LC2:
 7          .string "You entered %d...\n"
 8 f5:
 9 ; save FP and LR in stack frame:
10          stp     x29, x30, [sp, -16]!
11 ; set stack frame (FP=SP)
12          add     x29, sp, 0
13 ; load pointer to the "Enter X:" string:
14          adrp    x0, .LC0
15          add     x0, x0, :lo12:.LC0
16          bl      puts
17 ; load pointer to the "%d" string:
18          adrp    x0, .LC1
19          add     x0, x0, :lo12:.LC1
20 ; form address of x global variable:
21          adrp    x1, x
22          add     x1, x1, :lo12:x
23          bl      __isoc99_scanf
24 ; form address of x global variable again:
25          adrp    x0, x
26          add     x0, x0, :lo12:x
27 ; load value from memory at this address:
28          ldr     w1, [x0]
29 ; load pointer to the "You entered %d...\n" string:
30          adrp    x0, .LC2
31          add     x0, x0, :lo12:.LC2
32          bl      printf
33 ; return 0
34          mov     w0, 0
35 ; restore FP and LR:
36          ldp     x29, x30, [sp], 16
37          ret
```

在上述代码里变量 *x* 被声明为全局变量。程序通过 ADRP/ADD 指令对（第 21 行和第 25 行）计算它的指针。

7.2.7 MIPS

无未初始值的全局变量

以变量 *x* 为全局变量为例。我们把它编译为可执行文件，然后使用 IDA 加载这个程序。因为程序在声明变量 *x* 的时候没有对它进行初始化赋值，所以在 IDA 中变量 *x* 出现在.sbss ELF 里（请参见 3.5.1 节的全局指针）。

指令清单 7.8 Optimizing GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10          = -0x10
.text:004006C0 var_4           = -4
.text:004006C0
; function prologue:
.text:004006C0                 lui     $gp, 0x42
.text:004006C4                 addiu   $sp, -0x20
.text:004006C8                 li      $gp, 0x418940
```

```
.text:004006CC                  sw      $ra, 0x20+var_4($sp)
.text:004006D0                  sw      $gp, 0x20+var_10($sp)
; call puts():
.text:004006D4                  la      $t9, puts
.text:004006D8                  lui     $a0, 0x40
.text:004006DC                  jalr    $t9 ; puts
.text:004006E0                  la      $a0, aEnterX  # "Enter X:" ; branch delay slot
; call scanf():
.text:004006E4                  lw      $gp, 0x20+var_10($sp)
.text:004006E8                  lui     $a0, 0x40
.text:004006EC                  la      $t9, __isoc99_scanf
; prepare address of x:
.text:004006F0                  la      $a1, x
.text:004006F4                  jalr    $t9 ; __isoc99_scanf
.text:004006F8                  la      $a0, aD         # "%d" ; branch delay slot
; call printf():
.text:004006FC                  lw      $gp, 0x20+var_10($sp)
.text:00400700                  lui     $a0, 0x40
; get address of x:
.text:00400704                  la      $v0, x
.text:00400708                  la      $t9, printf
; load value from "x"  variable and pass it to printf() in $a1:
.text:0040070C                  lw      $a1, (x - 0x41099C)($v0)
.text:00400710                  jalr    $t9 ; printf
.text:00400714                  la      $a0, aYouEnteredD___ # "You entered %d...\n" ; branch delay slot
; function epilogue:
.text:00400718                  lw      $ra, 0x20+var_4($sp)
.text:0040071C                  move    $v0, $zero
.text:00400720                  jr      $ra
.text:00400724                  addiu   $sp, 0x20  ; branch delay slot
...

.sbss:0041099C  #  Segment type: Uninitialized
.sbss:0041099C                  .sbss
.sbss:0041099C                  .globl x
.sbss:0041099C x:               .space 4
.sbss:0041099C
```

IDA 会精简部分指令信息。我们不妨通过 objdump 观察上述文件确切的汇编指令。

指令清单 7.9　Optimizing GCC 4.4.5 (objdump)

```
 1 004006c0 <main>:
 2 ; function prologue:
 3   4006c0:       3c1c0042        lui     gp,0x42
 4   4006c4:       27bdffe0        addiu   sp,sp,-32
 5   4006c8:       279c8940        addiu   gp,gp,-30400
 6   4006cc:       afbf001c        sw      ra,28(sp)
 7   4006d0:       afbc0010        sw      gp,16(sp)
 8 ; call puts():
 9   4006d4:       8f998034        lw      t9,-32716(gp)
10   4006d8:       3c040040        lui     a0,0x40
11   4006dc:       0320f809        jalr    t9
12   4006e0:       248408f0        addiu   a0,a0,2288 ; branch delay slot
13 ; call scanf():
14   4006e4:       8fbc0010        lw      gp,16(sp)
15   4006e8:       3c040040        lui     a0,0x40
16   4006ec:       8f998038        lw      t9,-32712(gp)
17 ; prepare address of x:
18   4006f0:       8f858044        lw      a1,-32700(gp)
19   4006f4:       0320f809        jalr    t9
20   4006f8:       248408fc        addiu   a0,a0,2300 ; branch delay slot
21 ; call printf():
22   4006fc:       8fbc0010        lw      gp,16(sp)
23   400700:       3c040040        lui     a0,0x40
24 ; get address of x:
25   400704:       8f828044        lw      v0,-32700(gp)
```

```
26  400708:       8f99803c        lw      t9,-32708(gp)
27 ; load value from "x" variable and pass it to printf() in $a1:
28  40070c:       8c450000        lw      a1,0(v0)
29  400710:       0320f809        jalr    t9
30  400714:       24840900        addiu   a0,a0,2304 ; branch delay slot
31 ; function epilogue:
32  400718:       8fbf001c        lw      ra,28(sp)
33  40071c:       00001021        move    v0,zero
34  400720:       03e00008        jr      ra
35  400724:       27bd0020        addiu   sp,sp,32 ; branch delay slot
36 ; pack of NOPs used for aligning next function start on 16-byte boundary:
37  400728:       00200825        move    at,at
38  40072c:       00200825        move    at,at
```

第 18 行处的指令对全局指针 GP 和一个负数值的偏移量求和，以此计算变量 *x* 在 64KB 数据缓冲区里的访问地址。此外，三个外部函数（puts()、scanf()、printf()）在 64KB 数据空间里的全局地址，也是借助 GP 计算出来的（第 9 行、第 16 行、第 26 行）。GP 指向数据空间的正中央。经计算可知，三个函数的地址和变量 *x* 的地址都在数据缓冲区的前端。这并不意外，因为这个程序已经很短小了。

此外，值得一提的是函数结尾处的两条 NOP 指令。它的实际指令是空操作指令“MOVE$AT, $AT”。借助这两条 NOP 指令，后续函数的起始地址可向 16 字节边界对齐。

有初始值的全局变量

我们对前文的例子做相应修改，把有关行改为：

```
int x=10; // default value
```

则可得如下代码段。

指令清单 7.10 Optimizing GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10      = -0x10
.text:004006A0 var_8       = -8
.text:004006A0 var_4       = -4
.text:004006A0
.text:004006A0             lui     $gp, 0x42
.text:004006A4             addiu   $sp, -0x20
.text:004006A8             li      $gp, 0x418930
.text:004006AC             sw      $ra, 0x20+var_4($sp)
.text:004006B0             sw      $s0, 0x20+var_8($sp)
.text:004006B4             sw      $gp, 0x20+var_10($sp)
.text:004006B8             la      $t9, puts
.text:004006BC             lui     $a0, 0x40
.text:004006C0             jalr    $t9 ; puts
.text:004006C4             la      $a0, aEnterX    # "Enter X:"
.text:004006C8             lw      $gp, 0x20+var_10($sp)
; prepare high part of x address:
.text:004006CC             lui     $s0, 0x41
.text:004006D0             la      $t9, __isoc99_scanf
.text:004006D4             lui     $a0, 0x40
; add low part of x address:
.text:004006D8             addiu   $a1, $s0, (x - 0x410000)
; now address of x is in $a1.
.text:004006DC             jalr    $t9 ; __isoc99_scanf
.text:004006E0             la      $a0, aD         # "%d"
.text:004006E4             lw      $gp, 0x20+var_10($sp)
; get a word from memory:
.text:004006E8             lw      $a1, x
; value of x is now in $a1.
.text:004006EC             la      $t9, printf
.text:004006F0             lui     $a0, 0x40
.text:004006F4             jalr    $t9 ; printf
```

```
.text:004006F8              la        $a0, aYouEnteredD___  # "You entered %d...\n"
.text:004006FC              lw        $ra, 0x20+var_4($sp)
.text:00400700              move      $v0, $zero
.text:00400704              lw        $s0, 0x20+var_8($sp)
.text:00400708              jr        $ra
.text:0040070C              addiu     $sp, 0x20

...

.data:00410920              .globl  x
.data:00410920 x:           .word   0xA
```

为何它处没有了.sdata 段？这可能是受到了 GCC 选项的影响。不论如何，变量 *x* 出现在.data 段里。这个段会被加载到常规的通用内存区域，我们可以在此看到变量的处理方法。

MTPS 程序必须使用成对指令处理变量的地址。本例使用的是 LUI(Load Upper Immediate)和 ADDIU (Add Immediate Unsigned Word)指令时。

我们继续借助 objdump 观察确切地操作指令。

指令清单 7.11　Optimizing GCC 4.4.5 (objdump)

```
004006a0 <main>:
  4006a0:       3c1c0042        lui     gp,0x42
  4006a4:       27bdffe0        addiu   sp,sp,-32
  4006a8:       279c8930        addiu   gp,gp,-30416
  4006ac:       afbf001c        sw      ra,28(sp)
  4006b0:       afb00018        sw      s0,24(sp)
  4006b4:       afbc0010        sw      gp,16(sp)
  4006b8:       8f998034        lw      t9,-32716(gp)
  4006bc:       3c040040        lui     a0,0x40
  4006c0:       0320f809        jalr    t9
  4006c4:       248408d0        addiu   a0,a0,2256
  4006c8:       8fbc0010        lw      gp,16(sp)
; prepare high part of x address:
  4006cc:       3c100041        lui     s0,0x41
  4006d0:       8f998038        lw      t9,-32712(gp)
  4006d4:       3c040040        lui     a0,0x40
; add low part of x address:
  4006d8:       26050920        addiu   a1,s0,2336
; now address of x is in $a1.
  4006dc:       0320f809        jalr    t9
  4006e0:       248408dc        addiu   a0,a0,2268
  4006e4:       8fbc0010        lw      gp,16(sp)
; high part of x address is still in $s0.
; add low part to it and load a word from memory:
  4006e8:       8e050920        lw      a1,2336(s0)
; value of x is now in $a1.
  4006ec:       8f99803c        lw      t9,-32708(gp)
  4006f0:       3c040040        lui     a0,0x40
  4006f4:       0320f809        jalr    t9
  4006f8:       248408e0        addiu   a0,a0,2272
  4006fc:       8fbf001c        lw      ra,28(sp)
  400700:       00001021        move    v0,zero
  400704:       8fb00018        lw      s0,24(sp)
  400708:       03e00008        jr      ra
  40070c:       27bd0020        addiu   sp,sp,32
```

这个程序使用 LUI 和 ADDIU 指令对生成变量地址。地址的高地址位仍然存储于$S0 寄存器，而且单条 LW 指令（Load Word）即可封装这个偏移量。所以，单条 LW 指令足以提取变量的值，然后把它交付给 printf()函数。

T-字头的寄存器名称是临时数据寄存器的助记符。此外，这段程序还使用到了 S-字头的寄存器名称。在调用其他函数之前，调用方函数应当保管好自身 S-字头的寄存器的值，避免它们受到被调用方函数的影响，举例来说在 0x4006cc 处的指令对$S0 寄存器赋值，而后程序调用了 scanf()函数，接着地址为 0x4006e8 的指令然继续调用$S0 据此我们可以判断。scanf()函数不会变更$S0 的值。

7.3 scanf()函数的状态监测

大家都知道scanf()不怎么流行了，但是这并不代表它派不上用场了。在万不得已必须使用这个函数的时候，切记检查函数的退出状态是否正确。例如：

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");
        return 0;
};
```

根据这个函数的功能规范[①]，scanf()函数在退出时会返回成功赋值的变量总数。

就本例子而言，正常情况下：用户输入一个整型数字时函数返回 1；如果没有输入的值存在问题（或为 EOF/没有输入数据），scanf()则返回 0。

为此我们可在 C 程序里添加结果检查的代码，以便在出现错误时进行相应的处理。

我们来验证一下：

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

7.3.1 MSVC：x86

使用 MSVC2010 生成的汇编代码如下所示。

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d... ', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh? ', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

① 请参照 http://msdn.microsoft.com/en-us/library/9y6s16x1(VS.71).aspx。

当被调用方函数 callee（本例中是 scanf()函数）使用 EAX 寄存器向调用方函数 caller（本例中是 main()函数）传递返回值。

之后，“CMP EAX，1”指令对返回值进行比对，检查其值是否为 1。

JNE 是条件转移指令其全称是“Jump if Not Equal”。在两值不相同时进行跳转。

就是说，如果 EAX 寄存器里的值不是 1，则程序将跳转到 JNE 所指明的地址（本例中会跳到 $LN1@main）；在将控制权指向这个地址之后，CPU 会执行其后的打印指令，显示“What you entered? Huh?”。另一种情况是 scanf()成功读取指定数据类型的数据，其返回值就会是 1，此时不会发生跳转，而是继续执行 JNE 以后的指令，显示‘You entered %d…’和变量 *x* 的值。

在 scanf()函数成功地给变量赋值的情况下，程序会一路执行到 JMP（无条件转移）指令。这条指令会跳过第二条调用 printf()函数的指令，从“XOR EAX,EAX”指令开始执行，从而完成 return 0 的操作。

可见，“一般地说”条件判断语句会出现成对的“CMP/Jcc”指令。此处 cc 是英文“condition code”的缩写。比较两个值的 CMP 指令会设置处理器的标志位[①]。Jcc 指令会检查这些标志位，判断是否进行跳转。

但是上述的说法容易产生误导，实际上 CMP 指令进行的操作是减法运算。确切地说，不仅是 CMP 指令所有的“数学/算术计算”指令都会设置标志位。如果将 1 与 1 进行比较，1−1=0，ZF 标志位（“零”标识位，最终运算结果是 0）将被计算指令设定为 1。将两个不同的数值进行 CMP 比较时，ZF 标志位的值绝不会是 1。JNE 指令会依据 ZF 标志位的状态判断是否需要进行跳转，实际上此两者（Jump if Not Zero）的同义指令。JNE 和 JNZ 的 opcode 都相同。所以，即使使用减法运算操作指令 SUB 替换 CMP 指令，Jcc 指令也可以进行正常的跳转。不过在使用 SUB 指令时，我们还需要分配一个寄存器保存运算结果，而 CMP 则不需要使用寄存器保存运算结果。

7.3.2　MSVC：x86：IDA

现在来让 IDA 大显身手。对于多数初学者来说，使用 MSVC 编译器的/MD 选项是个值得推荐的好习惯。这个选项会要求编译器“不要链接（link）标准函数”，而是从 MSVCR*.DLL 里导入这些标准函数。总之，使用/MD 选项编译出来的代码一目了然，便于我们观察它在哪里、调用了哪些标准函数。

在使用 IDA 分析程序的时候，应当充分利用它的标记功能。比如说，分析这段程序的时候，我们明白在发生错误的时候会执行 JNE 跳转。此时就可以用鼠标单击跳转的 JNE 指令，按下“n”键，把相应的标签（lable）改名为“error”；然后把正常退出的标签重命名为“exit”。这种修改就可大幅度增强代码的可读性。

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X: \n"
.text:00401009       call    ds:printf
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
```

① processor flags,参见 http://en.wikipedia.org/wiki/FLAGS_register_(computing)。

```
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit:  ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

如此一来，这段代码就容易理解了。虽然重命名标签的功能很强大，但是逐一修改每条指令的标签则无疑是画蛇添足。

此外，IDA 还有如下一些高级用法。

整理代码：

标记某段代码之后，按下键盘数字键的“－”减号，整段代码将被隐藏，只留下首地址和标签。下面的例子中，我隐藏了两段代码，并对整段代码进行了重命名。

```
.text:00401000 _text  segment para public 'CODE' use32
.text:00401000        assume cs:_text
.text:00401000        ;org 401000h
.text:00401000 ;  ask for X
.text:00401012 ;  get X
.text:00401024        cmp    eax, 1
.text:00401027        jnz    short error
.text:00401029 ;  print result
.text:0040103B        jmp    short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D        push  offset aWhat ; "What you entered? Huh?\n"
.text:00401042        call  ds:printf
.text:00401048        add   esp, 4
.text:0040104B
.text:0040104B exit:  ; CODE XREF: _main+3B
.text:0040104B        xor   eax, eax
.text:0040104D        mov   esp, ebp
.text:0040104F        pop   ebp
.text:00401050        retn
.text:00401050 _main endp
```

如需显示先前隐藏的代码，可以直接使用数字键盘上的“＋”加号。

图解模式：

按下空格键，IDA 将会进入图解的显示方式。其效果如图 7.7 所示。

判断语句会分出两个箭头，一条是红色、一条是绿色。当判断条件表达式的值为真时，程序会走绿色箭头所示的流程；如果判断条件表达式不成立，程序会采用红色箭头所标示的流程。

图解模式下，也可以对各分支节点命名和收缩。图 7.8 处理了 3 个模块。

这种图解模式非常实用。逆向工程工作经验不多的人，可使用这种方式来大幅度地减少他需要处理的信息量。

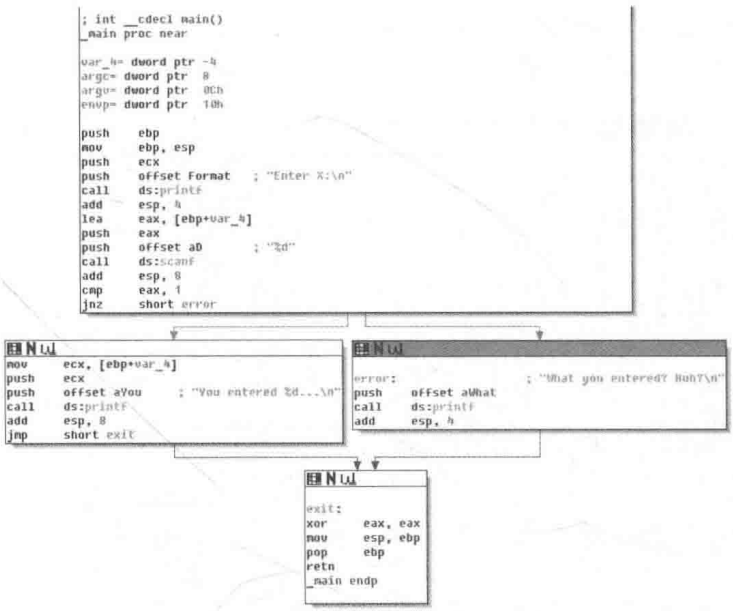


图 7.7　IDA 的图解模式

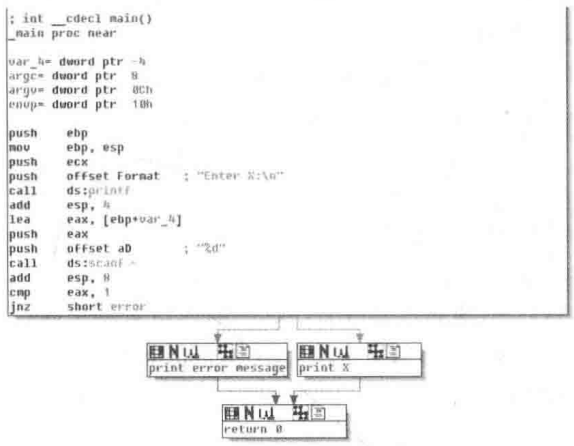


图 7.8　IDA 图解模式下的收缩操作

7.3.3　MSVC：x86+OllyDbg

我们使用 OllyDbg 调试刚才的程序，在 scanf()异常返回的情况下强制其继续运行余下的指令。

在把本地变量的地址传递给 scanf()的时候，变量本身处于未初始化状态，其值应当是随机的噪声数据。如图 7.9 所示，变量 *x* 的值为 0x6E494714。

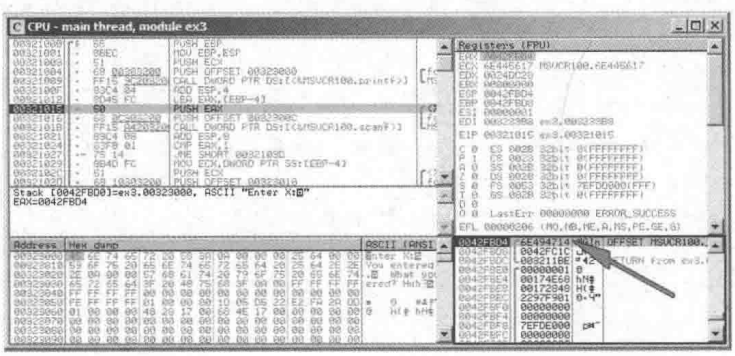


图 7.9　使用 OllyDbg 观察 scanf()的变量传递过程

在执行 scanf()函数的时候，我们输入非数字的内容，例如 “asdasd”。这时候 scanf()会通过 EAX 寄存器返回 0，如图 7.10 所示。这个零意味着 scanf()函数遇到某种错误。

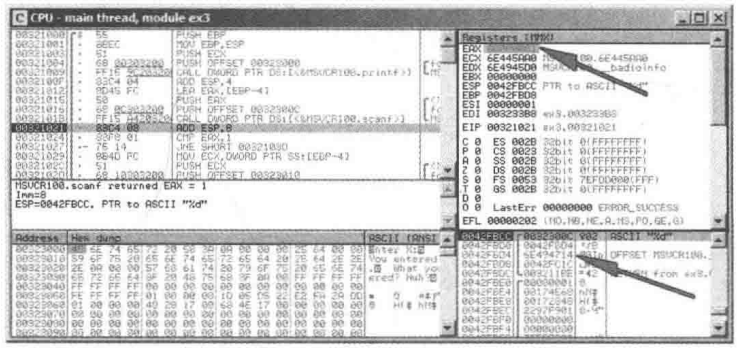


图 7.10 使用 OllyDbg 观察 scanf()的异常处理

执行 scanf()前后，变量 *x* 的值没有发生变化。在上述情况下，scanf()函数仅仅返回 0，而没对变量进行赋值。

现在来“hack”一下。右键单击 EAX，选中“Set to 1”。

在此之后 EAX 寄存器存储的值被人为设定为 1，程序将完成后续的操作，printf()函数会在控制台里显示数据栈里变量 *x* 的值。

图 7.11 控制台窗口

我们使用 F9 键继续运行程序。此后控制台的情况如图 7.11 所示。

1850296084 是十进制值，其十六进制值就是我们刚才看到的 0x6E494714。

7.3.4 MSVC：x86+Hiew

下面，我们一起修改可执行文件。所谓的“修改”可执行文件，就是对其打补丁（即人们常说的“patch”）。通过打补丁的方法，我们可以强制程序在所有情况下都进行输出。

首先，我们要启用 MSVC 的编译选项/MD。这样编译出来的可执行文件将把标准函数链接到 MSVCR*.DLL，以方便我们在可执行文件的文本段里找到 main()函数。此后，我们使用 Hiew 工具打开可执行文件，找到.text 段开头部分的 main()函数（依次使用 Enter, F8, F6, Enter, Enter）。

然后会看到图 7.12 所示的界面。

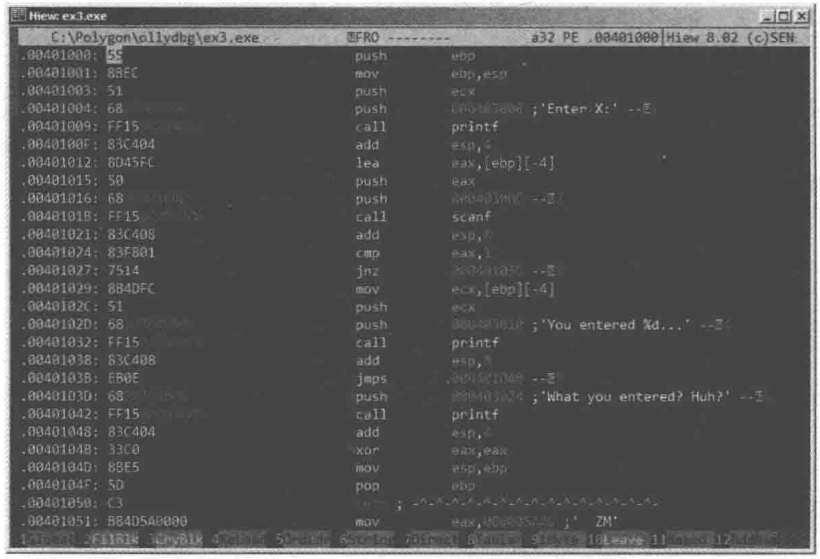


图 7.12 使用 Hiew 观察 main()函数

Hiew 能够识别并显示 ASCIIZ，即以 null 为结束字节的 ASCII 字符串。它还能识别导入函数的函数名称。

如图 7.13 所示，将鼠标光标移至.0040127 处（即 JNZ 指令的所在位置，我们使其失效），按 F3 键，然后输入“9090”。“9090”是两个连续的 NOP（No Operation，空操作）的 opcode。

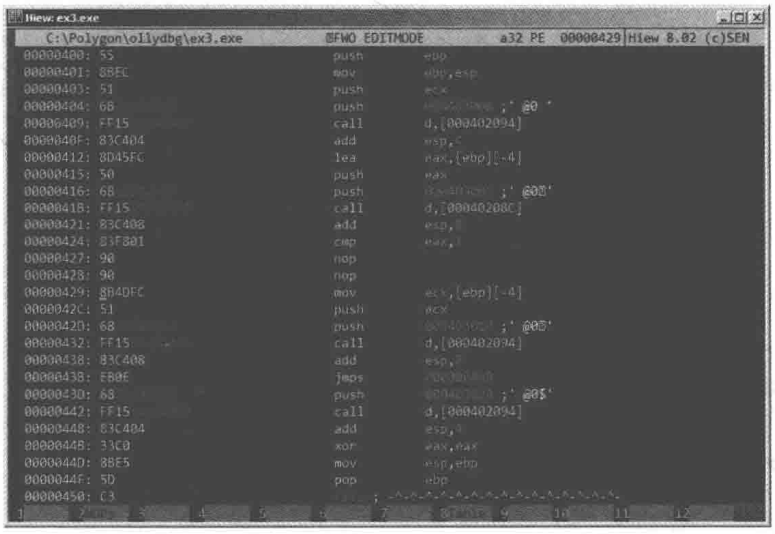


图 7.13　在 Hiew 中把 JNZ 替换为两条 NOP 指令

然后按 F9 键（更新），把修改后的可执行文件保存至磁盘。

连续的 9090 是典型的修改特征，有的人会觉得不甚美观。此外还可以把这个 opcode 的第二个字节改为零（opcode 的两个字节代表 jump offset），令 jcc 指令跳转的偏移量为 0，从而继续运行下一条指令。

刚才的修改方法可使转移指令失效。除此以外我们还可以强制程序进行转我们可以把 jcc 对应的 opcode 的第一个字节替换为“EB”，不去修改第二字节（offset）。这种修改方法把条件转移指令替换为了无条件跳转指令。经过这样的调整之后，本例中的可执行文件都会无条件地显示错误处理信息“What you entered? Huh?”。

7.3.5　MSVC:x64

本例使用的变量 *x* 是 int 型整数变量。在 x64 系统里，int 型变量还是 32 位数据。在 64 位平台上访问寄存器的（低）32 位时，计算机就要使用助记符以 E-头的寄存器名称。然而在访问 x64 系统的 64 位指针时，我们就需要使用 R-字头的寄存器名称、处理完整的 64 位数据。

指令清单 7.12　MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X: ', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d... ', 0aH, 00H
$SG2929 DB      'What you entered? Huh? ', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X: '
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
```

```
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d,.. '
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh? '
        call    printf
$LN1@main:
        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

7.3.6 ARM

ARM: Optimizing Keil 6/2013 (Thumb 模式)

指令清单 7.13 Optimizing Keil 6/2013 (Thumb 模式)

```
var_8           = -8

                PUSH    {R3,LR}
                ADR     R0, aEnterX          ; "Enter X:\n"
                BL      __2printf
                MOV     R1, SP
                ADR     R0, aD               ; "%d"
                BL      __0scanf
                CMP     R0, #1
                BEQ     loc_1E
                ADR     R0, aWhatYouEntered  ; "What you entered? Huh?\n"
                BL      __2printf
loc_1A                                       ; CODE XREF: main+26
                MOVS    R0, #0
                POP     {R3,PC}
loc_1E                                       ; CODE XREF: main+12
                LDR     R1, [SP,#8+var_8]
                ADR     R0, aYouEnteredD___  ; "You entered %d...\n"
                BL      __2printf
                B       loc_1A
```

这里我们见到了未介绍过的 CMP 和 BEQ 指令。

ARM 系统的 CMP 与 x86 系统的同名指令相似。它们都是将两个参数相减，并设置相应的标志位。

BEQ 是条件转移指令（Branch if EQual），在 CMP 操作数相等的情况下进行跳转。如果 CMP 比较的两个值相同，则 ZF 标志寄存器的值就会是 1，那么 BEQ 指令就会跳转到它指定的偏移量上去。它与 x86 的 JZ 指令作用相同。

其余的指令并不难理解：程序流有两个分支，这两个分支最终收敛于 loc_1A 处，通过“MOV R0, #0”指令把返回值保存于 R0 寄存器，然后退出。

ARM64

指令清单 7.14 Non-optimizing GCC 4.9.1 ARM64

```
1 .LC0:
2         .string "Enter X:"
3 .LC1:
4         .string "%d"
5 .LC2:
```

```
 6         .string "You entered %d...\n"
 7 .LC3:
 8         .string "What you entered? Huh?"
 9 f6:
10 ; save FP and LR in stack frame:
11         stp     x29, x30, [sp, -32]!
12 ; set stack frame (FP=SP)
13         add     x29, sp, 0
14 ; load pointer to the "Enter X:" string:
15         adrp    x0, .LC0
16         add     x0, x0, :lo12:.LC0
17         bl      puts
18 ; load pointer to the "%d" string:
19         adrp    x0, .LC1
20         add     x0, x0, :lo12:.LC1
21 ; calculate address of x variable in the local stack
22         add     x1, x29, 28
23         bl      __isoc99_scanf
24 ; scanf() returned result in W0.
25 ; check it:
26         cmp     w0, 1
27 ; BNE is Branch if Not Equal
28 ; so if W0<>0, jump to L2 will be occurred
29         bne     .L2
30 ; at this moment W0=1, meaning no error
31 ; load x value from the local stack
32         ldr     w1, [x29,28]
33 ; load pointer to the "You entered %d...\n" string:
34         adrp    x0, .LC2
35         add     x0, x0, :lo12:.LC2
36         bl      printf
37 ; skip the code, which print the "What you entered? Huh?" string:
38         b       .L3
39 .L2:
40 ; load pointer to the "What you entered? Huh?" string:
41         adrp    x0, .LC3
42         add     x0, x0, :lo12:.LC3
43         bl      puts
44 .L3:
45 ; return 0
46         mov     w0, 0
47 ; restore FP and LR:
48         ldp     x29, x30, [sp], 32
49         ret
```

上述程序通过 CMP/BNE 指令对控制分支语句。

7.3.7 MIPS

指令清单 7.15 Optimizing GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18      = -0x18
.text:004006A0 var_10      = -0x10
.text:004006A0 var_4       = -4
.text:004006A0
.text:004006A0             lui     $gp, 0x42
.text:004006A4             addiu   $sp, -0x28
.text:004006A8             li      $gp, 0x418960
.text:004006AC             sw      $ra, 0x28+var_4($sp)
.text:004006B0             sw      $gp, 0x28+var_18($sp)
.text:004006B4             la      $t9, puts
.text:004006B8             lui     $a0, 0x40
.text:004006BC             jalr    $t9 ; puts
.text:004006C0             la      $a0, aEnterX    # "Enter X:"
```

```
.text:004006C4              lw      $gp, 0x28+var_18($sp)
.text:004006C8              lui     $a0, 0x40
.text:004006CC              la      $t9, __isoc99_scanf
.text:004006D0              la      $a0, aD         # "%d"
.text:004006D4              jalr    $t9 ; __isoc99_scanf
.text:004006D8              addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC              li      $v1, 1
.text:004006E0              lw      $gp, 0x28+var_18($sp)
.text:004006E4              beq     $v0, $v1, loc_40070C
.text:004006E8              or      $at, $zero      # branch delay slot, NOP
.text:004006EC              la      $t9, puts
.text:004006F0              lui     $a0, 0x40
.text:004006F4              jalr    $t9 ; puts
.text:004006F8              la      $a0, aWhatYouEntered  # "What you entered? Huh?"
.text:004006FC              lw      $ra, 0x28+var_4($sp)
.text:00400700              move    $v0, $zero
.text:00400704              jr      $ra
.text:00400708              addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C              la      $t9, printf
.text:00400710              lw      $a1, 0x28+var_10($sp)
.text:00400714              lui     $a0, 0x40
.text:00400718              jalr    $t9 ; printf
.text:0040071C              la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text:00400720              lw      $ra, 0x28+var_4($sp)
.text:00400724              move    $v0, $zero
.text:00400728              jr      $ra
.text:0040072C              addiu   $sp, 0x28
```

scanf()函数通过$V0 寄存器传递其返回值。地址为 0x004006E4 的指令负责比较$V0 和$V1 的值。其中，$V1 在 0x004006DC 处被赋值为 1。BEQ 的作用是“Branch Equal”（在相等时进行跳转）。如果两个寄存器里的值相等，即成功读取了 1 个整数，那么程序将会从 0x0040070C 处继续执行指令。

7.3.8 练习题

JNE/JNZ 指令可被修改为 JE/JZ 指令，而且后者也可被修改为前者。BNE 和 BEQ 之间也有这种关系。不过，在进行这种替代式修改之后，还要对程序的基本模块进行修改。请多进行一些有关练习。

7.4 练习题

7.4.1 题目

这段代码在 Linux x86-64 上用 GCC 编译，运行的时候都崩溃了（段错误）。然而，它在 Windows 环境下用 Msvc 2010 x86 编译后却能工作，为什么？

```
#include <string.h>
#include <stdio.h>

void alter_string(char *s)
{
        strcpy (s, "Goodbye!");
        printf ("Result: %s\n", s);
};

int main()
{
        alter_string ("Hello, world!\n");
};
```

第8章 参数获取

调用方（caller）函数通过栈向被调用方（callee）函数传递参数。本章介绍被调用方函数获取参数的具体方式。

指令清单 8.1 范例

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

8.1 x86

8.1.1 MSVC

使用 MSVC 2010 Express 编译上述程序，可得到汇编指令如下。

指令清单 8.2 MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8                         ; size = 4
_b$ = 12                        ; size = 4
_c$ = 16                        ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3rd argument
        push    2 ; 2nd argument
        push    1 ; 1st argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
```

```
        ; return 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

main()函数把 3 个数字推送入栈，然后调用了 f(int, int, int)。被调用方函数 f()通过_a$=8 一类的汇编宏访问所需参数以及函数自定义的局部变量。只不过从被调用方函数的数据栈的角度来看，外部参考的偏移量是正值，而局部变量的偏移量是负值。可见，当需要访问栈帧（stack frame）以外的数据时，被调用方函数可把汇编宏（例如_a$）与 EBP 寄存器的值相加，从而求得所需地址。

当变量 *a* 的值存入 EAX 寄存器之后，f()函数通过各参数的地址依次进行乘法和加法运算，运算结果一直存储于 EAX 寄存器。此后 EAX 的值就可以直接作为返回值传递给调用方函数。调用方函数 main()再把 EAX 的值当作参数传递给 printf()函数。

8.1.2 MSVC+OllyDbg

本节演示 OllyDbg 的使用方法。当 f()函数读取第一个参数时，EBP 的值指向栈帧，如图 8.1 中的红色方块所示。栈帧里的第一个值是 EBP 的原始状态，第二个值是返回地址 RA，第三个值开始的三个值依次为函数的第一个参数、第二个参数和第三个参数。在访问第一个参数（当需要访问第一个参数）时，计算机需要把 EBP 的值加上 8（2 个 32 位 words）。

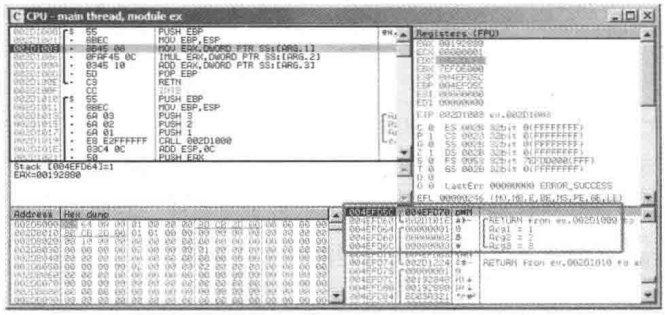


图 8.1 使用 OllyDbg 观察 f()函数

OllyDbg 能够识别出外部传递的参数。它会对栈里的数据进行标注，添加上诸如“RETURN from”“Arg1”之类的标注信息。

故而，被调用方函数所需的参数并不在自己的栈帧之中，而是在调用方函数的栈帧里。所以，被 OllyDbg 标注为 Arg 的数据都存储于其他函数的栈帧。

8.1.3 GCC

我们使用 GCC 4.4.1 编译上述源程序，然后使用 IDA 查看它的汇编指令。

指令清单 8.3 GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1st argument
        imul    eax, [ebp+arg_4] ; 2nd argument
        add     eax, [ebp+arg_8] ; 3rd argument
        pop     ebp
```

```
            retn
f           endp

            public main
main        proc near

var_10      = dword ptr -10h
var_C       = dword ptr -0Ch
var_8       = dword ptr -8

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 10h
            mov     [esp+10h+var_8], 3 ; 3rd argument
            mov     [esp+10h+var_C], 2 ; 2nd argument
            mov     [esp+10h+var_10], 1 ; 1st argument
            call    f
            mov     edx, offset aD  ; "%d\n"
            mov     [esp+10h+var_C], eax
            mov     [esp+10h+var_10], edx
            call    _printf
            mov     eax, 0
            leave
            retn
main        endp
```

GCC 的编译结果和 MSVC 的编译结果十分相似。

不同之处是两个被调用方函数（f 和 printf）没有还原栈指针 SP。这是因为函数尾声的倒数第二条指令——LEAVE 指令（参见附录 A.6.2）能够还原栈指针。

8.2　x64

x86-64 系统的参数传递过程略有不同。x86-64 系统能够使用寄存器传递（前 4 个或前 6 个）参数。就这个程序而言，被调用方函数会从寄存器里获取参数，完全不需要访问栈。

8.2.1　MSVC

启用优化选项后，MSVC 编译的结果如下。

指令清单 8.4　Optimizing MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - 1st argument
        ; EDX - 2nd argument
        ; R8D - 3rd argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
```

```
f       ENDP
```

我们可以看到，f()函数通过寄存器获取了全部的所需参数。此处求址的加法运算是通过 LEA 指令实现的。很明显，编译器认为 LEA 指令的效率比 ADD 指令的效率高，所以它分配了 LEA 指令。在制备 f()函数的第一个和第三个参数时，main()函数同样使用了 LEA 指令。编译器无疑认为 LEA 指令向寄存器赋值的速度比常规的 MOV 指令速度快。

我们再来看看 MSVC 未开启优化选项时的编译结果。

指令清单 8.5 MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1st argument
                ; EDX - 2nd argument
                ; R8D - 3rd argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d,3 ; 3rd argument
                mov     edx,2 ; 2nd argument
                mov     ecx,1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931  ; "%d\n"
                call    printf

                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

比较意外的是，原本位于寄存器的 3 个参数都被推送到了栈里。这种现象叫作“阴影空间/shadow space”[①]。每个 Win64 程序都可以（但非必须）把 4 个寄存器的值保存到阴影空间里。使用阴影空间有以下两个优点：1）通过栈传递参数，可避免浪费寄存器资源（有时可能会占用 4 个寄存器）；2）便于调试器 debugger 在程序中断时找到函数参数（请参阅：https://msdn.microsoft.com/en-us/library/ew5tede7%28v=VS.90%29.aspx）。

大型函数可能会把输入参数保存在阴影空间里，但是小型函数（如本例）可能就不会使用阴影空间了。

在使用阴影空间时，由调用方函数分配栈空间，由被调用方函数根据需要将寄存器参数转储到它们的阴影空间中。

8.2.2 GCC

在启用优化选项后，GCC 生成的代码更为晦涩。

指令清单 8.6 Optimizing GCC 4.4.6 x64

```
f:
```

① 请参阅 MSDN https://msdn.microsoft.com/en-us/library/zthk2dkh%28v=vs.80%29.aspx。

```
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

不开启优化选项的情况下，GCC 生成的代码如下。

指令清单 8.7　GCC 4.4.6 x64

```
f:
        ;EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; number of vector registers passed
        call    printf
        mov     eax, 0
        leave
        ret
```

阴影空间只是微软的概念，System V *NIX［参见参考文献 Mit13］里没有这种规范或约定。GCC 只有在寄存器数量容纳不下所有参数的情况下，才会使用栈传递参数。

8.2.3　GCC: uint64_t 型参数

指令清单 8.1 的源程序采用的是 32 位 int 整型参数，所以以上的汇编指令使用的寄存器都是 64 位寄存器的低 32 位（即 E-字头寄存器）。如果把参数改为 64 位数据，那么汇编指令则会略有不同：

```
#include <stdio.h>
#include <stdint.h>
```

```
uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

指令清单 8.8 Optimizing GCC 4.4.6 x64

```
f           proc near
            imul    rsi, rdi
            lea     rax, [rdx+rsi]
            retn
f           endp

main        proc near
            sub     rsp, 8
            mov     rdx, 3333333344444444h ; 3rd argument
            mov     rsi, 1111111122222222h ; 2nd argument
            mov     rdi, 1122334455667788h ; 1st argument
            call    f
            mov     edi, offset format ; "%lld\n"
            mov     rsi, rax
            xor     eax, eax ; number of vector registers passed
            call    _printf
            xor     eax, eax
            add     rsp, 8
            retn
main        endp
```

相应的汇编指令使用了整个寄存器（R-字头寄存器）。其他部分基本相同。

8.3 ARM

8.3.1 Non-optimizing Keil 6/2013 (ARM mode)

```
.text:000000A4  00 30 A0 E1     MOV         R3, R0
.text:000000A8  93 21 20 E0     MLA         R0, R3, R1, R2
.text:000000AC  1E FF 2F E1     BX          LR
...
.text:000000B0                  main
.text:000000B0  10 40 2D E9     STMFD       SP!, {R4,LR}
.text:000000B4  03 20 A0 E3     MOV         R2, #3
.text:000000B8  02 10 A0 E3     MOV         R1, #2
.text:000000BC  01 00 A0 E3     MOV         R0, #1
.text:000000C0  F7 FF FF EB     BL          f
.text:000000C4  00 40 A0 E1     MOV         R4, R0
.text:000000C8  04 10 A0 E1     MOV         R1, R4
.text:000000CC  5A 0F 8F E2     ADR         R0, aD_0  ; "%d\n"
.text:000000D0  E3 18 00 EB     BL          __2printf
.text:000000D4  00 00 A0 E3     MOV         R0, #0
.text:000000D8  00 00 A0 E3     LDMFD       SP!, {R4,PC}
```

主函数只起到了调用另外 2 个函数的作用。它把 3 个参数传递给了 f()函数。

前文提到过，在 ARM 系统里，前 4 个寄存器（R0～R3）负责传递前 4 个参数。

在本例中，f()函数通过前 3 个寄存器（R0～R2）读取参数。

MLA（Multiply Accumulate）指令将前两个操作数（R3 和 R1 里的值）相乘，然后再计算第三个操作数（R2 里的值）和这个积的和，并且把最终运算结果存储在零号寄存器 R0 之中。根据 ARM 指令的有关规范，返回值就应该存放在 R0 寄存器里。

MLA 是乘法累加指令[①]，能够一次计算乘法和加法运算，属于非常有用的指令。在 SIMD 技术[②]的 FMA 指令问世之前，x86 平台的指令集里并没有类似的指令。

首条指令“MOV R3, R0”属于冗余指令。即使此处没有这条指令，后面的 MLA 指令直接使用有关的寄存器也不会出现任何问题。因为我们没有启用优化选项，所以编译器没能对此进行优化。

BL 指令把程序的控制流交给 LR 寄存器里的地址，而且会在必要的时候切换处理器的运行模式（Thumb 模式和 ARM 模式之间进行模式切换）。被调用方函数 f()并不知道它会被什么模式的代码调用，不知道调用方函数属于 ARM 模式的代码还是 Thumb 模式的代码。所以这种模式切换的功能还是必要的。如果它被 Thumb 模式的代码调用，BX 指令不仅会进行相应的跳转，还会把处理器模式调整为 Thumb。如果它被 ARM 模式的指令[③]调用，则不会进行模式切换。

8.3.2 Optimizing Keil 6/2013 (ARM mode)

```
.text:00000098                 f
.text:00000098 91 20 20 E0     MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1     BX      LR
```

在启用最大幅度的优化功能（-O3）之后，前面那条 MOV 指令被优化了，或者说被删除了。MLA 直接使用所有寄存器的参数，并且把返回值保存在 R0 寄存器里。调用方函数继而可从 R0 寄存器获取返回值。

8.3.3 Optimizing Keil 6/2013 (Thumb mode)

```
.text:0000005E 48 43    MULS    R0, R1
.text:00000060 80 18    ADDS    R0, R0, R2
.text:00000062 70 47    BX      LR
```

因为 Thumb 模式的指令集里没有 MLA 指令，所以编译器将它分为两个指令。第一条 MULS 指令计算 R0 和 R1 的积，把运算结果存储在 R1 寄存器里。第二条 ADDS 计算 R1 和 R2 的和，并且把计算结果存储在 R0 寄存器里。

8.3.4 ARM64

Optimizing GCC (Linaro) 4.9

ARM64 的情况简单一些。MADD 指令可以一次进行乘法和加法的混合运算，与前文的 MLA 指令十分类似。全部 3 个参数由 X-字头寄存器的低 32 位传递。这是因为这些参数都是 32 位整型数据。函数的返回值存储在 W0 寄存器。

指令清单 8.9　Optimizing GCC (Linaro) 4.9

```
f:
        madd    w0, w0, w1, w2
        ret
main:
; save FP and LR to stack frame:
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
```

① 请参见 http://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation。
② “单指令流多数据流”的缩写，请参见 https://en.wikipedia.org/wiki/FMA_instruction_set。
③ ARM12，附录 A2.3.2。

```
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0, :lo12:.LC7
        bl      printf
; return 0
        mov     w0, 0
; restore FP and LR
        ldp     x29, x30, [sp], 16
        ret
.LC7:
        .string "%d\n"
```

我们再来看看参数为 64 位 uint64_t 的情况：

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};
int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};


f:
        madd    x0, x0, x1, x2
        ret
main:
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0, :lo12:.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
.LC8:
        .string "%lld\n"
```

这两段代码的 f()函数部分相同。在采用 64 位参数之后，程序使用了整个 64 位 X 寄存器。程序通过两条指令才能把长数据类型的 64 位值存储到寄存器里。本书在 28.3.1 节会详细介绍 64 位数据的有关操作。

Non-optimizing GCC (Linaro) 4.9

在没有启用优化选项的情况下，编译器生成的代码稍显冗长：

```
f:
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w1, w1, w0
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

函数 f()把传入的参数保存在数据栈里，以防止后期的指令占用 W0～W2 寄存器。这可防止后续指令覆盖函数参数，起到保护传入参数的作用。这种技术叫作“寄存器保护区/Register Save Area”[参见 ARM13c]。但是，本例的这种被调用方函数可以不这样保存参数。寄存器保护区与 8.2.1 节介绍的阴影空间十分相似。

在启用优化选项后，GCC 4.9 会把这部分寄存器存储指令删除。这是因为优化功能判断出后续指令不会再操作函数参数的相关地址，所以编译器不再另行保存 W0～W2 中存储的数据。

此外，上述代码使用了 MUL/ADD 指令对，而没有使用 MADD 指令。

8.4 MIPS

指令清单 8.10　Optimizing GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000              mult    $a1, $a0
.text:00000004              mflo    $v0
.text:00000008              jr      $ra
.text:0000000C              addu    $v0, $a2, $v0 ; branch delay slot
; result in $v0 upon  return

.text:00000010 main:
.text:00000010
.text:00000010 var_10       = -0x10
.text:00000010 var_4        =-4
.text:00000010
.text:00000010              lui     $gp, (__gnu_local_gp >> 16)
.text:00000014              addiu   $sp, -0x20
.text:00000018              la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C              sw      $ra, 0x20+var_4($sp)
.text:00000020              sw      $gp, 0x20+var_10($sp)
; set c:
.text:00000024              li      $a2, 3
; set a:
.text:00000028              li      $a0, 1
.text:0000002C              jal     f
; set b:
.text:00000030              li      $a1, 2      ; branch delay slot
; result in $v0 now
.text:00000034              lw      $gp, 0x20+var_10($sp)
.text:00000038              lui     $a0, ($LC0 >> 16)
.text:0000003C              lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040              la      $a0, ($LC0 & 0xFFFF)
.text:00000044              jalr    $t9
; take result of f()        function and pass it as a second argument to printf():
.text:00000048              move    $a1, $v0        ; branch delay slot
.text:0000004C              lw      $ra, 0x20+var_4($sp)
.text:00000050              move    $v0, $zero
.text:00000054              jr      $ra
.text:00000058              addiu   $sp, 0x20       ; branch delay slot
```

函数所需的前 4 个参数由 4 个 A-字头寄存器传递。

MIPS 平台有两个特殊的寄存器：HI 和 LO。它们用来存储 MULT 指令的乘法计算结果——64 位的积。只有 MFLO 和 MFHI 指令能够访问 HI 和 LO 寄存器。其中，MFLO 负责访问积的低 32 位部分。本例中它把积的低 32 位部分存储到$V0 寄存器。

因为本例没有访问积的高 32 位，所以那半部分被丢弃了。不过我们的程序就是这样设计的：积是 32 位的整型数据。

最终 ADDU（Add Unsigned）指令计算第三个参数与积的和。

在 MIPS 平台上，ADD 和 ADDU 是两个不同的指令。此二者的区别体现在异常处理的方式上，而符号位的处理方式反而没有区别。ADD 指令可以触发溢出处理机制。溢出有时候是必要的[①]，而且被 Ada 和其他编程语言支持。ADDU 不会引发溢出。因为 C/C++不支持这种机制，所以本例使用的是 ADDU 指令而非 ADD 指令。

此后$V0 寄存器存储这 32 位的运算结果。

main()函数使用到了 JAL（Jump and Link）指令。JAL 和 JALR 指令有所区别，前者使用的是相对地址——偏移量，后者则跳转到寄存器存储的绝对地址里。JALR 的 R 代表 Register。由于 f()函数和 main()函数都位于同一个 object 文件，所以 f()函数的相对地址是已知的，可以被计算出来。

① http://blog.regehr.org/archives/1154。

第9章 返 回 值

在x86系统里，被调用方函数通常通过EAX寄存器返回运算结果。[①]若返回值属于byte或char类型数据，返回值将存储于EAX寄存器的低8位——AL寄存器存储返回值。如果返回值是浮点float型数据，那么返回值将存储在FPU的ST(0)寄存器里。ARM系统的情况相对简单一些，它通常使用R0寄存器回传返回值。

9.1 void型函数的返回值

主函数main()的数据类型通常是void而不是int，程序如何处理返回值呢？

调用main()函数的有关代码大体会是这样的：

```
push envp
push argv
push argc
call main
push eax
call exit
```

将其转换为源代码，也就是：

```
exit(main(argc,argv,envp));
```

如果声明main()的数据类型是void，则main()函数不会明确返回任何值（没有return指令）。不过在main()函数退出时，EAX寄存器还会存有数据，EAX寄存器保存的数据会被传递给exit()函数、成为后者的输入参数。通常EAX寄存器的值会是被调用方函数残留的确定数据，所以void类型函数的返回值、也就是主函数退出代码往往属于伪随机数（pseudorandom）。

在我们进行相应的演示之前，请注意main()函数返回值是void型：

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

然后使用Linux系统编译。

3.4.3节处介绍过GCC 4.8.1会使用puts()替换printf()，而且puts()函数会返回它所输出的字符的总数。我们将充分利用这点，观察main()函数的返回值。请注意在main()函数结束时，EAX寄存器的值不会是零；也就是说，此时EAX寄存器存储的值应当是上一个函数——puts()函数的返回值。

指令清单9.1 GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
```

① 请参见MSDN: Return Values (C++): http://msdn.microsoft.com/en-us/library/7572ztz4.aspx。

```
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

我们再通过一段 bash 脚本程序，观察程序的退出状态（返回值）。

指令清单 9.2 tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

执行上述脚本之后，我们将会看到：

```
$ ./tst.sh
Hello, world!
14
```

这个“14”就是 puts()函数输出的字符的总数。

9.2 函数返回值不被调用的情况

printf() 函数的返回值为打印的字符的总数，但是很少有程序会使用这个返回值。实际上，确实有调用运算函数、却不使用运算结果的程序：

```
int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};
```

上述四个 rand()函数都会把运算结果存储到 EAX 寄存器里。但是前三个 rand()函数留在 EAX 寄存器的运算结果都被抛弃了。

9.3 返回值为结构体型数据

我们继续讨论使用 EAX 寄存器存储函数返回值的案例。函数只能够使用 EAX 这 1 个寄存器回传返回值。因为这种局限，过去的 C 编译器无法编译返回值超过 EAX 容量（一般来说，就是 int 型数据）的函数。那个时候，如果要让返回多个返回值，那么只能用函数返回一个值、再通过指针传递其余的返回值。现在的 C 编译器已经没有这种短板了，return 指令甚至可以返回结构体型的数据，只是时下很少有人会这么做。如果函数的返回值是大型结构的数据，那么应由调用方函数（caller）负责分配空间，给结构体分配指针，再把指针作为第一个参数传递给被调用方函数。现在的编译器已经能够替程序员自动完成这种复杂的操作了，其处理方式相当于上述几个步骤，只是编译器隐藏了有关操作。

我们来看：

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
```

```
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;
    return rt;
};
```

使用 MSVC 2010 （启用优化选项/Ox）编译，可得到：

```
$T3853 = 8                          ; size = 4
_a$ = 12                            ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC          ; get_some_values
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, DWORD PTR $T3853[esp-4]
    lea     edx, DWORD PTR [ecx+1]
    mov     DWORD PTR [eax], edx
    lea     edx, DWORD PTR [ecx+2]
    add     ecx, 3
    mov     DWORD PTR [eax+4], edx
    mov     DWORD PTR [eax+8], ecx
    ret     0
?get_some_values@@YA?AUs@@H@Z ENDP          ; get_some_values
```

在程序内部传递结构体的指针就是$T3853。

如果使用 C99 扩展语法来写，刚才的程序就是：

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

经 GCC 4.8.1 编译上述程序，可得到如下所示的指令。

指令清单 9.3 GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

可见，调用方函数（caller）创建了数据结构、分配了数据空间，被调用的函数仅向结构体填充数据。其效果等同于返回结构体。这种处理方法并不会影响程序性能。

第 10 章　指　　针

指针通常用来帮助函数处理返回值（请参阅第 7 章的范例）。当函数需要返回多个值时，它通常都是通过指针传递返回值的。

10.1　全局变量

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

经 MSVC 2010（启用优化选项/Ox /Ob）编译上述程序，可得到如下所示的指令。

指令清单 10.1　Optimizing MSVC 2010 (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H
_x$ = 8                                         ; size = 4
_y$ = 12                                        ; size = 4
_sum$ = 16                                      ; size = 4
_product$ = 20                                  ; size = 4

_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456                             ; 000001c8H
        push    123                             ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
```

```
        mov   ecx, DWORD PTR _sum
        push  eax
        push  ecx
        push  OFFSET $SG2803
        call  DWORD PTR __imp__printf
        add   esp, 28                           ; 0000001cH
        xor   eax, eax
        ret   0
_main ENDP
```

如图 10.1 所示，我们使用 OllyDbg 调试这个程序。

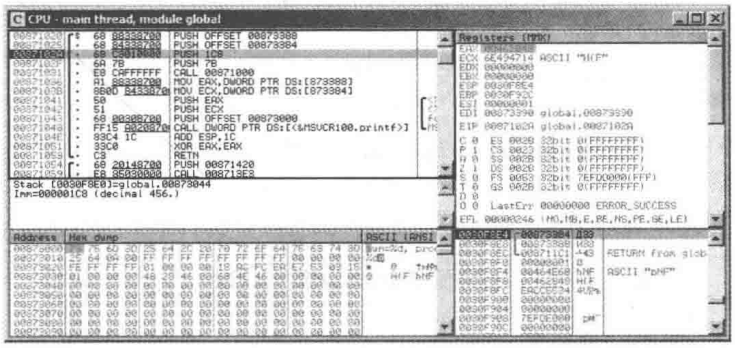


图 10.1　使用 OllyDbg 观察全局变量的地址传递到 f1()函数的过程

首先，全局变量的地址会被传递给 f1()函数。我们用右键单击栈中的元素，选中“Follow in dump”，将会在数据段中看到两个变量的实际情况。

在运行第一条指令之前，在 BSS（Block Started by Symbol）域内未被初始化赋值的数据会置零，所以这些变量会被清空（ISO 可规范，6.7.8 节）。变量驻留在数据段。如图 10.2 所示，我们可以按 Alt-M 组合键调用 memory map 进行确认。

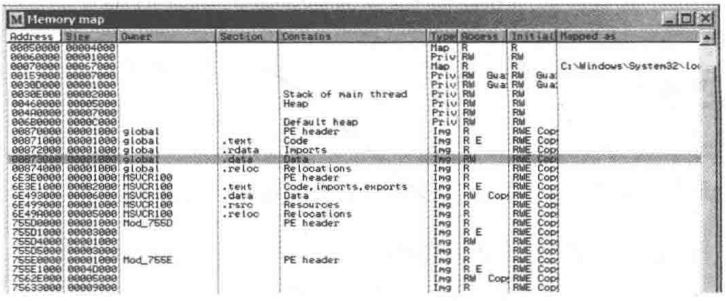


图 10.2　使用 OllyDbg 查看内存映射

按下 F7 键及跟踪调试 f1()函数[①]，如图 10.3 所示。

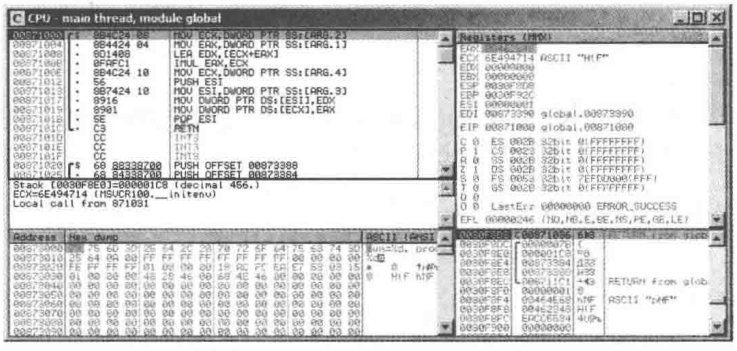


图 10.3　使用 OllyDbug 跟踪调试 f1()

① trace，可以进入被调用的函数里继续进行单步调试。如果使用 F8 键，则可一步就完成 call 调用的函数。

如上图所示，数据栈里会有 0x1C8(456)和 0x7B(123)两个值，以及这两个全局变量的地址（指针）。

继续跟踪调试程序，待其执行到 f1()函数尾部。如图 10.4 所示，我们会在左下方的窗口看到全局变量的计算结果。

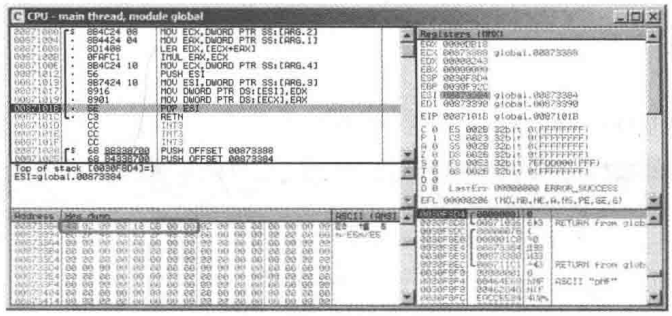


图 10.4 使用 ollyDbg 观察 f1()函数执行完毕

而后，全局变量的值将被加载到寄存器中，通过栈传递给 printf()函数。如图 10.5 所示。

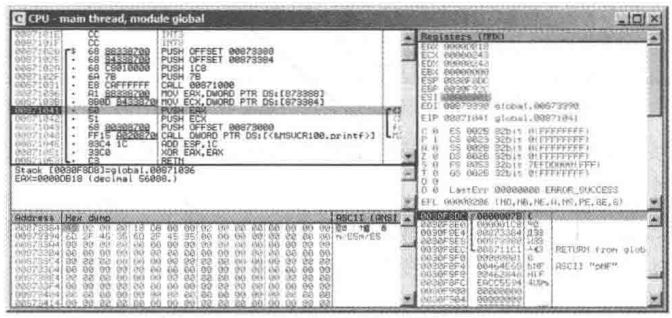


图 10.5 使用 OllyDbg 观察 printf()函数获取全局变量的地址

10.2 局部变量

下面，我们略微调整下程序，使其通过局部变量完成同样的功能。

指令清单 10.2 将和、积变量存储在局部变量里

```
void main()
{
        int sum, product; // now variables are local in this function

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

编译之后，f1()函数的汇编代码与使用全局变量的代码完全相同。而 main()函数的汇编代码会有相应的变化。启用/Ox /Ob0 的优化选项编译后，会得到如下所示的指令。

指令清单 10.3 Optimizing MSVC 2010 (/Ob0)

```
_product$ = -8                                          ; size = 4
_sum$ = -4                                              ; size = 4
_main    PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
```

```
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456                             ; 000001c8H
        push    123                             ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36                         ; 00000024H
        ret     0
```

接下来，我们使用 OllyDbg 来调试这个程序。栈内的两个变量在栈内 0x2EF854 和 0x2EF858 两个地址上。如图 10.6 所示，我们观察下它们入栈的过程。

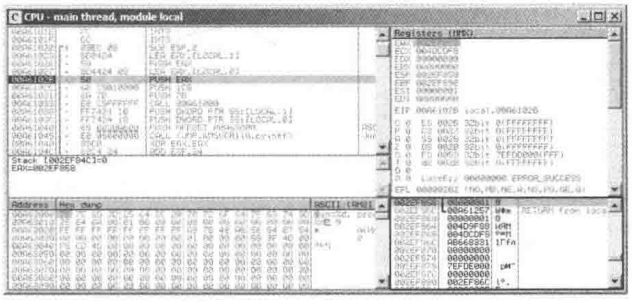


图 10.6　使用 OllyDbg 观察局部变量的入栈过程

在 f1()函数启动时，0x2EF854 和 0x2EF858 地址的数据都是随机的脏数据，如图 10.7 所示。

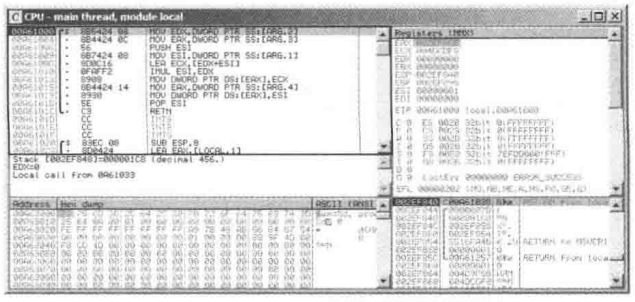


图 10.7　使用 OllyDbg 追踪 f1()的启动过程

f1()函数结束时的情况如图 10.8 所示。

现在 0x2EF854 的值是 0xDB18，0x2EF8588 的值是 0x243。它们都是 f1()函数的运算结果。

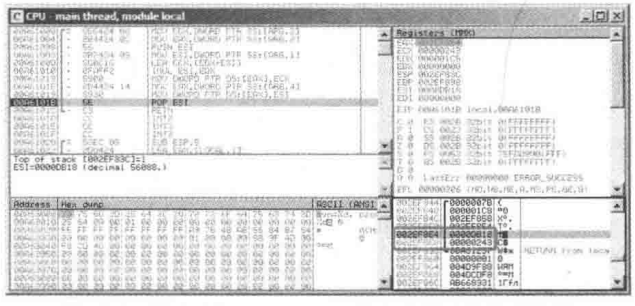


图 10.8　使用 OllyDbg 观察 f1()执行完毕时的栈内数据

10.3 总结

借助指针，f1() 函数可以返回位于任意地址的任意值。这个例子体现了指针的重要作用。C++的引用（指针）/reference 的作用与 C pointer（指针）相同。本书的 51.3 节将详细介绍 C++的引用。

第 11 章　GOTO 语句

人们普遍认为使用 GOTO 语句将破坏代码的结构化构造[①]。即使如此，某些情况下我们还必须使用这种语句[②]。

本章围绕以下程序进行说明：

```
#include <stdio.h>

int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

使用 MSVC 2012 编译上述程序，可得到如下所示的指令。

指令清单 11.1　MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

源代码中的 goto 语句直接被编译成了 JMP 指令。这两个指令的效果完全相同：无条件的转移到程序中的另一个地方继续执行后续命令。

只有在人工干预的情况下，例如使用调试器调整程序、或者对程序打补丁的情况下，程序才会调用第二个 printf()函数。

我们用它练习 patching 技术。首先使用 Hiew 打开刚才编译的可执行文件，如图 11.1 所示将光标移动到 JMP（0x410）处，按下 F3 键（编辑），再输入两个零。这样，我们就把这个地址的 opcode 改为了 EB 00 如图 11.2 所示。

JMP 所在的 opcode 的第二个字节代表着跳转的相对偏移量。把这个偏移量改为 0，就可以让它继续运行后续指令。这样 JMP 指令就不会跳过第二个 printf()函数。

按下 F9 键（保存文件）并退出 Hiew。再次运行这个可执行文件的情况应当如图 11.3 所示。

① 请参阅 Dij68。
② 请参阅 Knu74，以及 Yur13。

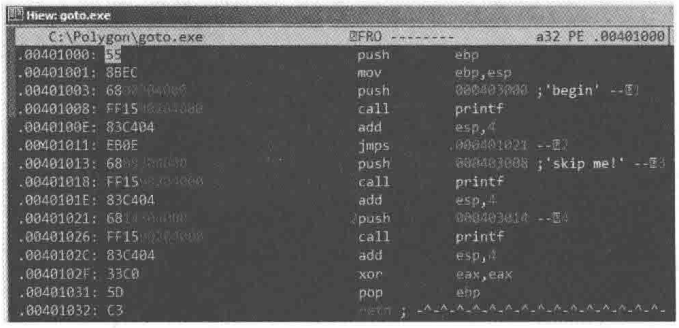


图 11.1 Hiew

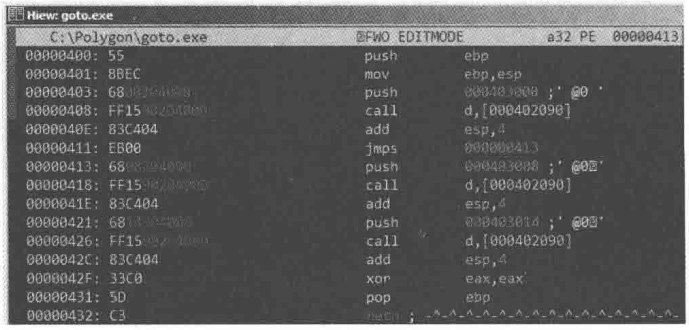


图 11.2 Hiew

当然，把 JMP 指令替换为两个 NOP 指令可以达到同样效果。因为 NOP 的 opcode 是 0x90、只占用一个字节，所以在进行替换时要写上两个 NOP 指令。

```
C:\Polygon>goto.exe

begin
skip me!
end
```

图 11.3 修改后的程序

11.1 无用代码 Dead Code

在编译术语里，上述程序中第二次调用 printf()函数的代码称为“无用代码/dead code”。无用代码永远不会被执行。所以，在启用编译器的优化选项之后，编译器会把这种无用代码删除得干干净净。

指令清单 11.2 Optimizing MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

然而编译器却没能删除字符串“skip me!”。

11.2 练习题

请结合自己的编译器和调试器进行有关练习。

第 12 章　条件转移指令

12.1　数值比较

本章会围绕以下程序进行演示：

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

12.1.1　x86

x86 + MSVC

在关闭优化选项时，使用 MSVC 编译上述源程序，可得到 f_signed()函数。

指令清单 12.1　Non-optimizing MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737          ; 'a>b'
    call   _printf
    add    esp, 4

$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
```

```
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739       ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741       ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

第一个条件转移指令是 JLE，即“Jump if Less or Equal”。如果上一条 CMP 指令的第一个操作表达式小于或等于（不大于）第二个表达式，JLE 将跳转到指令所标明的地址；如果不满足上述条件，则运行下一条指令，就本例而言程序将会调用 printf()函数。第二个条件转移指令是 JNE，“Jump if Not Equal”，如果上一条 CMP 的两个操作符不相等，则进行相应跳转。

第三个条件转移指令是 JGE，即“Jump if Greater or Equal”。如果 CMP 的第一个表达式大于或等于第二个表达式（不小于），则进行跳转。这段程序里，如果三个跳转的判断条件都不满足，将不会调用 printf()函数；不过，除非进行特殊干预，否则这种情况应该不会发生。

现在我们观察 f_unsigned()函数的汇编指令。f_unsigned()函数和 f_signed()函数大体相同。它们的区别集中体现在条件转移指令上：f_unsinged()函数的使用的条件转移指令是 JBE 和 JAE，而 f_signed()函数使用的条件转移指令则是 JLE 和 JGE。

使用 GCC 编译上述程序，可得到 f_unsigned()的汇编指令如下。

指令清单 12.2 GCC

```
_a$ = 8  ; size  = 4
_b$ = 12  ; size  = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
LN4@f_unsigned:
    Pop    ebp
    Ret    0
_f_unsigned ENDP
```

GCC 编译的结果与 MSVC 编译的结果基本相同。

经 GCC 编译后，f_unsigned()函数使用的条件转移指令是 JBE（Jump if Below or Equal，相当于 JLE）和 JAE（Jump if Above or Equal，相当于 JGE）。JA/JAE/JB/JBE 与 JG/JGE/JL/JLE 的区别，在于它们检查的标志位不同：前者检查借/进位标志位 CF（1 意味着小于）和零标志位 ZF（1 意味着相等），后者检查“SF XOR OF”（1 意味着异号）和 ZF。从指令参数的角度看，前者适用于 unsigned（无符号）类型数据的（CMP）运算，而后者的适用于 signed（有符号）类型数据的运算。

本书第 30 章会介绍 signed 类型数据。可见，根据条件转移的指令，我们可以直接判断 CMP 所比较的变量的数据类型。

接下来，我们一起研究 main()函数的汇编代码。

指令清单 12.3　main()

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + OllyDbg

我们可以通过 OllyDbg 直观地观察到指令对标志寄存器的影响。我们先用 OllyDbg 观察 f_unsigned()函数比较无符号数的过程。f_unsigned()函数使用了 CMP 指令，分三次比较了两个相同的 unsigned 类型数据。因为参数相同，所以 CMP 设置的标志位必定相同。

如图 12.1 所示，在运行到第一个条件转移指令时，C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0。OllyDbg 会使用标志位的首字母作为该标志位的简称。

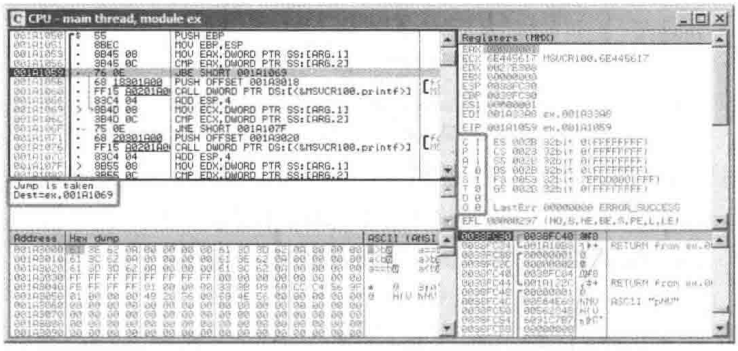


图 12.1　使用 OllyDbg：观察 f_unsigned()的第一个条件转移指令

OllyDbg 在左下窗口进行提示，JBE 条件跳转指令的条件已经达成，下一步会进行相应跳转。这种预测准确无误，JBE 的触发条件是（CF=1 或 ZF=1）。条件表达式为真时，JBE 确实会进行跳转。

如图 12.2 所示，在运行到第二个条件转移指令——JNZ 指令时，ZF=0。所以 OllyDbg 能够判断程序会进行相应跳转。

如图 12.3 所示，运行到第三个条件转移指令——JNB 指令的时候，借/进位标志 CF=0，条件表达式

会为假，所以不会发生跳转，程序将执行第三个 printf()指令。

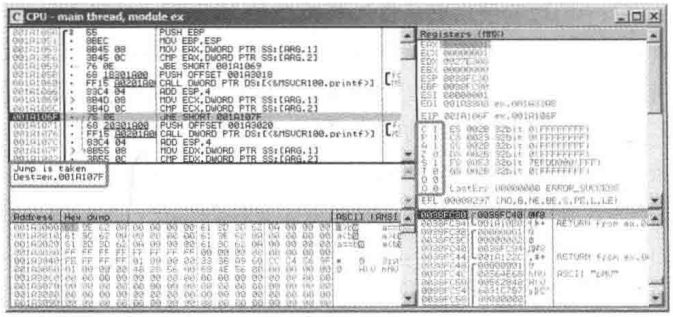


图 12.2 使用 OllyDbg 观察 f_unsigned()函数的第二个条件转移指令

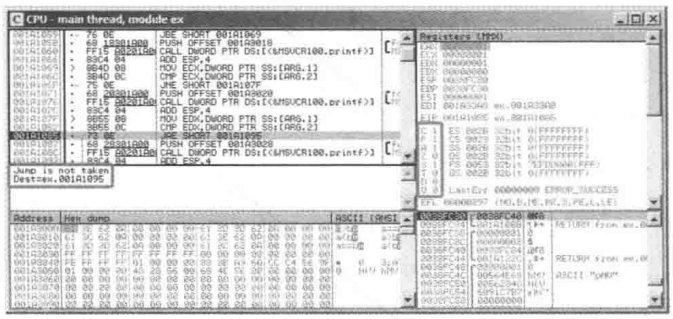


图 12.3 使用 OllyDbg 观察 f_unsigned()函数的第三个条件转移指令

现在来调试下示例程序里的 f_signed()函数，它的参数为 signed 型数据。

在运行 f_signed()函数时，标志位的状态和刚才一样。即，运行 CMP 指令之后，C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0。

第一个条件转移指令——JLE 指令将会被触发，如图 12.4 所示。

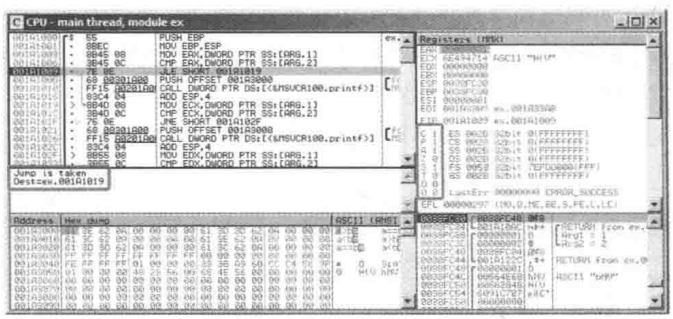


图 12.4 使用 OllyDbg 观察 f_signed()函数的第一个条件转移指令

参照［Int13］，触发 JLE 的条件是 ZF=1 或 SF≠OF。本例满足 SF≠OF 的条件。

由于 ZF＝0，第二个条件转移指令——JNZ 指令会被触发，如图 12.5 所示。

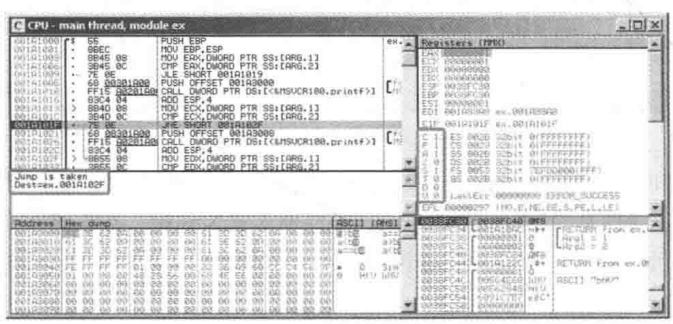


图 12.5 使用 OllyDbg 观察 f_signed()函数的第二个条件转移指令

而第三个条件转移指令——JGE 指令不会被触发。触发 JGE 的条件是 SF=OF，而当前情形不满足这个条件，如图 12.6 所示。

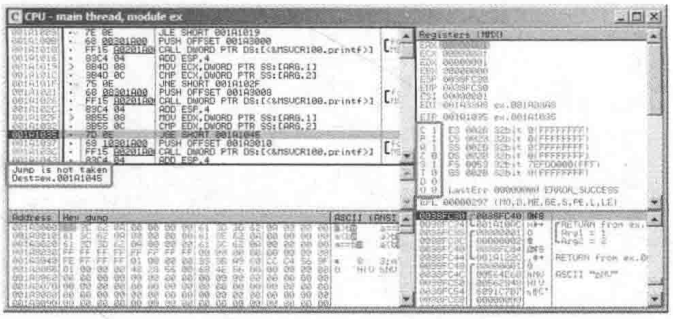


图 12.6　使用 OllyDbg 观察 f_signed()函数的第三个条件转移指令

x86 + MSVC +Hiew

我们还可以用 Hiew 给可执行文件打补丁，强制 f_unsigned()函数永远打印“a==b”、忽略它的输入参数，如图 2.7 所示。

图 12.7　使用 Hiew 打开 f_unsigned() 函数

本文将分 3 次修改上述可执行程序，分别完成下述三个任务：

- 强制触发第一个条件转移指令。
- 强制屏蔽第二个条件转移指令。
- 强制触发第三个条件转移指令。

我们可以直接修改程序，令程序流永远转向第二个 printf()函数并打印“a==b”。

故而需要修改三条指令（3 个字节）：

- 把第一个条件转移指令改为 JMP，并保留原始的转移偏移量(jump offset)。
- 第二个条件转移指令的触发条件不一定成立。无论触发条件是否成立，我们都要它跳转到下一条指令。所以，我们把转移偏移量设置为零。对于条件转移语句来说，跳转的目标地址是下一条地址与转移偏移量的和。把转移偏移量设置为零之后，程序会继续执行下一条指令。
- 第三个条件转移指令的修改方法和第一个条件转移指令的修改方法相同。我们只需把将条件转移指令换成 JMP（无条件转移指令）即可。

修改之后的 f_unsigned()函数如图 12.8 所示。

三个条件转移指令全部都要修改。如果少修改了一个指令，它就可能会多次调用 printf()函数，与我们

的预期——只调用一次 printf()函数的任务目标相悖。

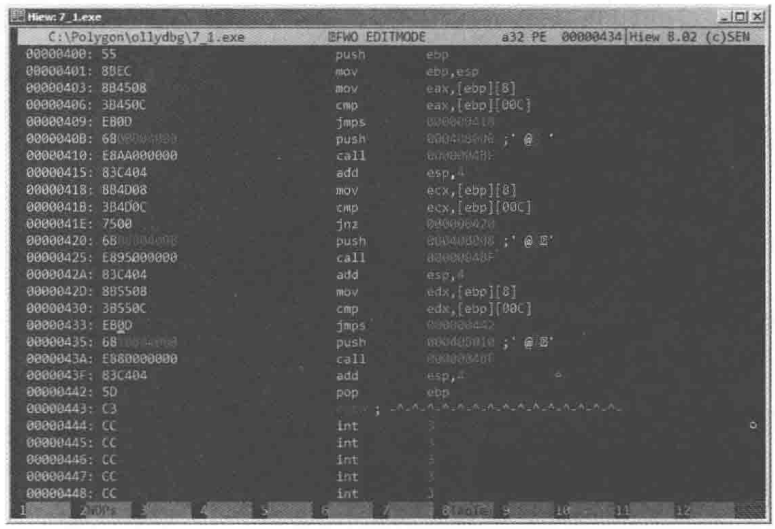


图 12.8 经过修改之后的 f_unsigned() 函数

Non-optimizing GCC

如果关闭了 GCC 的优化选项，那么它编译出来的程序和 MSVC 编译出来的程序没什么区别，只不过就是把 printf()函数替换为了 puts()函数[①]。

Optimizing GCC

聪明的您一定会问，既然 CMP 比较的是相同的值，比较之后的标志位的状态也相同，那么何必要对同样的参数进行多次比较呢？或许 MSVC 真的不能再智能一些了；但是启用优化选项后，GCC 4.8.1 确实能够进行这种深度优化。

指令清单 12.4 GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep     ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

很明显，它使用 jmp 指令替代了臃肿的“CALL ……puts …… RETN”指令。本书将在 13.1.1 节里详细解说这种编译技术。

我们不得不说在 x86 的系统中，这种程序比较少见。MSVC 2012 做不到 GCC 那种程度的深度优化。另一方面，汇编语言的编程人员确实可能学会 Jcc 指令的连用技巧。所以，如果您遇到了这样精简的程序，

① 请参见本书 3.4.3 节的解释。

而且还能够判断出它不是 GCC 编译出来的程序，那么您基本上可以判断它是手写出来的汇编程序。

即使开启了同样的优化选项，f_unsigned()函数对应的指令也没有那么精致。

指令清单 12.5　GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx        ; instruction may be removed
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

程序中只有两条 CMP 指令，至少它优化去了一个 CMP 指令。可见，GCC 4.8.1 的优化算法还有改进的空间。

12.1.2　ARM

32 位 ARM 程序

Optimizing Keil 6/2013 (ARM mode)

指令清单 12.6　Optimizing Keil 6/2013 (ARM mode)

```
.text:000000B8                 EXPORT f_signed
.text:000000B8             f_signed                ; CODE XREF: main+C
.text:000000B8 70 40 2D E9     STMFD    SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1     MOV      R4, R1
.text:000000C0 04 00 50 E1     CMP      R0, R4
.text:000000C4 00 50 A0 E1     MOV      R5, R0
.text:000000C8 1A 0E 8F C2     ADRGT    R0, aAB    ; "a>b\n"
.text:000000CC A1 18 00 CB     BLGT     __2printf
.text:000000D0 04 00 55 E1     CMP      R5, R4
.text:000000D4 67 0F 8F 02     ADREQ    R0, aAB_0; "a==b\n"
.text:000000D8 9E 18 00 0B     BLEQ     __2printf
.text:000000DC 04 00 55 E1     CMP      R5, R4
.text:000000E0 70 80 BD A8     LDMGEFD  SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8     LDMFD    SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2     ADR      R0, aAB_1 ; "a<b\n"
```

```
.text:000000EC 99 18 00 EA        B         __2printf
.text:000000EC             ; End of function f_signed
```

ARM 模式的多数指令都存在着相应的条件执行指令。这些派生出来的条件执行指令仅会在特定标志位为 1 的情况下执行。换句话说，只有当前面存在比较数值的指令时，后面才可能会出现这种派生出来的条件执行指令。

举例来讲，加法指令 ADD 指令实际上是 ADDAL 指令。“AL”就是 always 的缩写，即 ADDAL 总会被无条件执行。在 32 位的 ARM 指令中，条件判断表达式被封装在条件执行指令的前（最高）4 位——条件字段（condition field）里。即使是无条件转移指令 B 指令，其前 4 位还是条件字段。从指令构成上说，B 指令仍然属于条件转移指令，只不过它的条件字段是 AL 而已。顾名思义，AL 的作用就是忽略标志寄存器、永远执行这条指令。

ADRGT 指令中的 GT 代表 greater than（大于）。该指令依据先前 CMP 指令的比较结果，而判断是否执行寻址指令。当且仅当 CMP 比较的第一个值大于第二个值的时候，ADRGT 指令才会执行寻址（ADR）指令。

后面的 BLGT 指令有异曲同工之妙。仅在相同条件下，即当且仅当 CMP 比较的第一个值大于第二个值的时候，BLGT 指令才会执行 BL 指令。在这个条件成立的时候，前面的 ADRGT 指令已经把字符串“a>b /n”的地址赋值给 R0 寄存器，成为了 printf()的参数，而 BLGT 负责调用 printf()。可见，当且仅当在 R0 的值（变量 *a*）大于 R4 的值（变量 *b*）的情况下，计算机才会运行后面那组带有-GT 后缀的指令。很显然，这是一组相互关联的指令。

后面的 ADREQ 和 BLEQ 指令，都在最近一个 CMP 的操作数相等的情况下才会讲行 ADR 和 BL 指令的操作。程序之中连续两次出现“CMP R5, R4”指令，这是因为夹在其间的 printf()函数可能会影响标志位。

LDMGEFD 是“Great or Equal（大于或等于）”的情况下进行 LDMFD (Load Multiple Full Descending) 操作的指令。

依此类推，“LDMGEFD SP!, {R4-R6,PC}”指令起到函数尾声的作用，不过它只会在“a>=b”的时候才会结束本函数。

如果上述条件不成立，即“a<b”的时候，会执行下一条指令“LDMFD SP!, {R4-R6,LR}”。这同样起到函数尾声的作用。该指令将恢复 R4～R6 寄存器、LR 寄存器的值，而不恢复 PC 寄存器的值，且不会退出当前函数。

函数最后的两条指令，分别向 printf()函数传递参数（字符串“a<b\n”），并且调用 printf()函数。本书的 6.2.1 节已经介绍过：当调用方函数调用（跳转到）printf()函数之后，调用方函数可以伴随 printf()函数退出而退出。所以本节不再进行有关解释。

f_unsigned()函数与 f_signed()函数的功能十分类似。不同之处是它用到了 ADRHI、BLHI 和 LDMSFD 指令。指令尾部的-HI 代表 Unsigned Higher，CS 代表 Carry Set (greater than or equal)。因为参数的数据类型有所变化，所以这两个函数的具体指令有所区别。

这个程序的 main()函数的汇编指令如下。

指令清单 12.7 main()函数

```
.text:00000128                 EXPORT main
.text:00000128             main
.text:00000128 10 40 2D E9     STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3     MOV     R1, #2
.text:00000130 01 00 A0 E3     MOV     R0, #1
.text:00000134 DF FF FF EB     BL      f_signed
.text:00000138 02 10 A0 E3     MOV     R1, #2
.text:0000013C 01 00 A0 E3     MOV     R0, #1
.text:00000140 EA FF FF EB     BL      f_unsigned
.text:00000144 00 00 A0 E3     MOV     R0, #0
.text:00000148 10 80 BD E8     LDMFD   SP!, {R4,PC}
.text:00000148             ; End of function main
```

可见，ARM 模式的程序可以完全不依赖条件转移指令。

这样做有什么优点呢？依赖精简指令集（RISC）的 ARM 处理器采用流水线技术（pipeline）。简单地说，这种处理器在跳转指令方面的性能不怎么优越，所以它们的分支预测处理器（branch predictor unites）

决定了整体的性能。对于采用流水线技术的处理器来说，运行其上的程序跳转次数越少（无论是条件转移还是无条件转移），程序的性能就越高。条件执行指令[①]，会受益于其跳跃次数最少的优点，体现出最高的效率。详细介绍请参阅本书的 33.1 节

x86 指令集里只有 CMOVcc 指令，没有其他的条件执行指令了。CMOVcc 指令是仅在特定标志位为 1（通常由 CMP 指令设置）的情况下才会执行 MOV 操作的条件执行指令。

Optimizing Keil 6/2013 (Thumb mode)

指令清单 12.8　Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000072                f_signed ; CODE XREF: main+6
.text:00000072 70 B5            PUSH    {R4-R6,LR}
.text:00000074 0C 00            MOVS    R4, R1
.text:00000076 05 00            MOVS    R5, R0
.text:00000078 A0 42            CMP     R0, R4
.text:0000007A 02 DD            BLE     loc_82
.text:0000007C A4 A0            ADR     R0, aAB     ; "a>b\n"
.text:0000007E 06 F0 B7 F8      BL      __2printf
.text:00000082
.text:00000082                loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42            CMP     R5, R4
.text:00000084 02 D1            BNE     loc_8C
.text:00000086 A4 A0            ADR     R0, aAB_0   ; "a==b\n"
.text:00000088 06 F0 B2 F8      BL      __2printf
.text:0000008C
.text:0000008C                loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42            CMP     R5, R4
.text:0000008E 02 DA            BGE     locret_96
.text:00000090 A3 A0            ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8      BL      __2printf
.text:00000096
.text:00000096                locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD            POP     {R4-R6,PC}
.text:00000096                ; End of function f_signed
```

在 ARM 系统的 Thumb 模式指令集里，只有 B 指令才有派生出来的条件执行指令。所以 Thumb 模式下的汇编指令看上去更为贴近 x86 指令。

上述指令里，条件转移指令有 BLE(Less than or Equal)、BNE（Not Equal）、BGE(Greater than or Equal)。这些指令都可以望文生义。

f_unsigned 函数十分雷同，只是里面出现了新的条件转移指令 BLS(Unsigned Lower or Same)和 BCS（Carry Set(Greater than or equal)）。

64 位 ARM 程序

Optimizing GCC(Linaro)4.9

指令清单 12.9　f_signed()

```
f_signed:
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; Branch if Greater Than (a>b)
        beq     .L20    ; Branch if Equal (a==b)
        bge     .L15    ; Branch if Greater than or Equal (a>=b) (impossible here)
        ; a<b
        adrp    x0, .LC11       ; "a<b"
        add     x0, x0, :lo12:.LC11
        b       puts
.L19:
        adrp    x0, .LC9        ; "a>b"
```

① 即 predicated instructions，泛指 BLGT/ADREQ 这类混合条件判定功能的操作指令。

```
        add     x0, x0, :lo12:.LC9
        b       puts
.L15:   ; impossible here
        ret
.L20:
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0, :lo12:.LC10
        b       puts
```

指令清单 12.10 f_unsigned()

```
f_unsigned:
        stp     x29, x30, [sp, -48]!
; W0=a, W1=b
        cmp     w0, w1
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     w19, w0
        bhi     .L25            ; Branch if HIgher (a>b)
        cmp     w19, w1
        beq     .L26            ; Branch if Equal (a==b)
.L23:
        bcc     .L27            ; Branch if Carry Clear (if less than) (a<b)
; function epilogue, impossible to be here
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
.L27:
        ldr     x19, [sp,16]
        adrp    x0, .LC11       ; "a<b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC11
        b       puts
.L25:
        adrp    x0, .LC9        ; "a>b"
        str     x1, [x29,40]
        add     x0, x0, :lo12:.LC9
        bl      puts
        ldr     x1, [x29,40]
        cmp     w19, w1
        bne     .L23    ; Branch if Not Equal
.L26:
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC10
        b       puts
```

我在程序之中添加了注释。很明显，虽然有些条件表达式不可能成立，但是编译器不能自行判断出这种问题。所以程序留有一些永远不会执行的无效代码。

练习题

上述代码中存在无效代码。请在不添加新指令的情况下删除多余多指令。

12.1.3 MIPS

MIPS 处理器没有标志位寄存器，这是它最显著的特征之一。这种设计旨在降低数据相关性的分析难度。

x86 的指令集中有 SETcc 指令，MIPS 平台也有类似的指令：SLT（Set on Less Than/操作对象为有符号数）和 SLTU（无符号数）。这两个指令会在条件表达式为真的时候设置目的寄存器为 1，否则设置其为零。

随即可用 BEQ（Branch on Equal）或 BEN（Branch on Not Equal）指令检查上述寄存器的值，判断是否进行跳转。总之，在 MIPS 平台上组合使用这两种指令，可完成条件转移指令的比较和转移操作。

我们来看范本程序里处理有符号数的相应函数。

指令清单 12.11　Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 f_signed:                               # CODE XREF: main+18
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000 arg_0           =  0
.text:00000000 arg_4           =  4
.text:00000000
.text:00000000                 addiu     $sp, -0x20
.text:00000004                 sw        $ra, 0x20+var_4($sp)
.text:00000008                 sw        $fp, 0x20+var_8($sp)
.text:0000000C                 move      $fp, $sp
.text:00000010                 la        $gp, __gnu_local_gp
.text:00000018                 sw        $gp, 0x20+var_10($sp)
; store input values into local  stack:
.text:0000001C                 sw        $a0, 0x20+arg_0($fp)
.text:00000020                 sw        $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024                 lw        $v1, 0x20+arg_0($fp)
.text:00000028                 lw        $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C                 or        $at, $zero ; NOP
; this is pseudoinstruction. in fact, "slt $v0,$v0,$v1" is there.
; so $v0 will be set to 1 if $v0<$v1 (b<a) or to 0 if otherwise:
.text:00000030                 slt       $v0, $v1
; jump to loc_5c, if condition is not true.
; this is pseudoinstruction. in fact, "beq $v0,$zero,loc_5c" is there:
.text:00000034                 beqz      $v0, loc_5C
; print "a>b" and finish
.text:00000038                 or        $at, $zero ; branch delay slot, NOP
.text:0000003C                 lui       $v0, (unk_230 >> 16) # "a>b"
.text:00000040                 addiu     $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044                 lw        $v0, (puts & 0xFFFF)($gp)
.text:00000048                 or        $at, $zero ; NOP
.text:0000004C                 move      $t9, $v0
.text:00000050                 jalr      $t9
.text:00000054                 or        $at, $zero ; branch delay slot, NOP
.text:00000058                 lw        $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5C:                                 # CODE XREF: f_signed+34
.text:0000005C                 lw        $v1, 0x20+arg_0($fp)
.text:00000060                 lw        $v0, 0x20+arg_4($fp)
.text:00000064                 or        $at, $zero ; NOP
; check if a==b, jump to loc_90 if its not true':
.text:00000068                 bne       $v1, $v0, loc_90
.text:0000006C                 or        $at, $zero ; branch delay slot, NOP
; condition is true, so print "a==b" and finish:
.text:00000070                 lui       $v0, (aAB >> 16) # "a==b"
.text:00000074                 addiu     $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000078                 lw        $v0, (puts & 0xFFFF)($gp)
.text:0000007C                 or        $at, $zero ; NOP
.text:00000080                 move      $t9, $v0
.text:00000084                 jalr      $t9
.text:00000088                 or        $at, $zero ; branch delay slot, NOP
.text:0000008C                 lw        $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:                                 # CODE XREF: f_signed+68
.text:00000090                 lw        $v1, 0x20+arg_0($fp)
.text:00000094                 lw        $v0, 0x20+arg_4($fp)
.text:00000098                 or        $at, $zero ; NOP
; check if $v1<$v0 (a<b), set $v0 to 1 if condition is true:
.text:0000009C                 slt       $v0, $v1, $v0
```

```
; if condition is not true (i.e., $v0==0), jump to loc_c8:
.text:000000A0                  beqz    $v0, loc_C8
.text:000000A4                  or      $at, $zero ; branch delay slot, NOP
; condition is true, print "a<b" and finish
.text:000000A8                  lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:000000AC                  addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:000000B0                  lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4                  or      $at, $zero ; NOP
.text:000000B8                  move    $t9, $v0
.text:000000BC                  jalr    $t9
.text:000000C0                  or      $at, $zero ; branch delay slot, NOP
.text:000000C4                  lw      $gp, 0x20+var_10($fp)
.text:000000C8
; all 3 conditions were false, so just finish:
.text:000000C8 loc_C8:                                  # CODE XREF: f_signed+A0
.text:000000C8                  move    $sp, $fp
.text:000000CC                  lw      $ra, 0x20+var_4($sp)
.text:000000D0                  lw      $fp, 0x20+var_8($sp)
.text:000000D4                  addiu   $sp, 0x20
.text:000000D8                  jr      $ra
.text:000000DC                  or      $at, $zero ; branch delay slot, NOP
.text:000000DC  # End of function f_signed
```

此处有两条指令是 IDA 的伪指令。“SLT REG0, REG1”的实际指令是“SLT REG0, REG0, REG1”，而 BEQZ 的实际指令是“BEQ REG, $ZERO, LABEL”。

f_unsigned()函数的汇编指令，只是把 f_signed()函数中的 SLT 指令替换为 SLTU（U 是 unsigned 的缩写）。除此之外，处理有符号数和无符号数的两个函数完全相同。

指令清单 12.12 Non-optimizing GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned:                              # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10           = -0x10
.text:000000E0 var_8            = -8
.text:000000E0 var_4            = -4
.text:000000E0 arg_0            =  0
.text:000000E0 arg_4            =  4
.text:000000E0
.text:000000E0                  addiu   $sp, -0x20
.text:000000E4                  sw      $ra, 0x20+var_4($sp)
.text:000000E8                  sw      $fp, 0x20+var_8($sp)
.text:000000EC                  move    $fp, $sp
.text:000000F0                  la      $gp, __gnu_local_gp
.text:000000F8                  sw      $gp, 0x20+var_10($sp)
.text:000000FC                  sw      $a0, 0x20+arg_0($fp)
.text:00000100                  sw      $a1, 0x20+arg_4($fp)
.text:00000104                  lw      $v1, 0x20+arg_0($fp)
.text:00000108                  lw      $v0, 0x20+arg_4($fp)
.text:0000010C                  or      $at, $zero
.text:00000110                  sltu    $v0, $v1
.text:00000114                  beqz    $v0, loc_13C
.text:00000118                  or      $at, $zero
.text:0000011C                  lui     $v0, (unk_230 >> 16)
.text:00000120                  addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124                  lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128                  or      $at, $zero
.text:0000012C                  move    $t9, $v0
.text:00000130                  jalr    $t9
.text:00000134                  or      $at, $zero
.text:00000138                  lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                                 # CODE XREF: f_unsigned+34
.text:0000013C                  lw      $v1, 0x20+arg_0($fp)
```

```
.text:00000140                  lw      $v0, 0x20+arg_4($fp)
.text:00000144                  or      $at, $zero
.text:00000148                  bne     $v1, $v0, loc_170
.text:0000014C                  or      $at, $zero
.text:00000150                  lui     $v0, (aAB >> 16)  # "a==b"
.text:00000154                  addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000158                  lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C                  or      $at, $zero
.text:00000160                  move    $t9, $v0
.text:00000164                  jalr    $t9
.text:00000168                  or      $at, $zero
.text:0000016C                  lw      $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:                                 # CODE XREF: f_unsigned+68
.text:00000170                  lw      $v1, 0x20+arg_0($fp)
.text:00000174                  lw      $v0, 0x20+arg_4($fp)
.text:00000178                  or      $at, $zero
.text:0000017C                  sltu    $v0, $v1, $v0
.text:00000180                  beqz    $v0, loc_1A8
.text:00000184                  or      $at, $zero
.text:00000188                  lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:0000018C                  addiu   $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:00000190                  lw      $v0, (puts & 0xFFFF)($gp)
.text:00000194                  or      $at, $zero
.text:00000198                  move    $t9, $v0
.text:0000019C                  jalr    $t9
.text:000001A0                  or      $at, $zero
.text:000001A4                  lw      $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:                                 # CODE XREF: f_unsigned+A0
.text:000001A8                  move    $sp, $fp
.text:000001AC                  lw      $ra, 0x20+var_4($sp)
.text:000001B0                  lw      $fp, 0x20+var_8($sp)
.text:000001B4                  addiu   $sp, 0x20
.text:000001B8                  jr      $ra
.text:000001BC                  or      $at, $zero
.text:000001BC  # End of function f_unsigned
```

12.2　计算绝对值

本节将围绕以下程序进行演示：

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

12.2.1　Optimizing　MSVC

上述程序的编译结果如下。

指令清单 12.13　Optimizing MSVC 2012 x64

```
i$ = 8
my_abs   PROC
; ECX = input
        test    ecx, ecx
; check for sign of input value
; skip NEG instruction if sign is positive
        jns     SHORT $LN2@my_abs
```

```
; negate value
        neg     ecx
$LN2@my_abs:
; prepare result in EAX:
        mov     eax,    ecx
        ret     0
my_abs  ENDP
```

GCC 4.9 的编译结果几乎相同。

12.2.2 Optimizing Keil 6/2013: Thumb mode

指令清单 12.14 Optimizing Keil 6/2013：Thumb mode

```
my_abs PROC
        CMP       r0,#0
; is input value equal to zero or greater than zero?
; skip RSBS instruction then
        BGE       |L0.6|
; subtract input value from 0:
        RSBS      r0,r0,#0
|L0.6|
        BX        lr
        ENDP
```

ARM 平台没有负数运算指令，所以 Keil 编译器使用了“零减去数值”的减法运算指令“Reverse Subtract”（减数和被减数位置对调的减法运算），同样达到了替换符号的效果。

12.2.3 Optimizing Keil 6/2013: ARM mode

因为 ARM 模式的指令集存在条件执行指令，所以开启优化选项后可得到如下指令。

指令清单 12.15 Optimizing Keil 6/2013：ARM mode

```
my_abs PROC
        CMP       r0,#0
; execute "Reverse Subtract" instruction only if input value is less than 0:
        RSBLT     r0,r0,#0
        BX        lr
        ENDP
```

即使没有使用条件转移指令（请参见 33.1 节），它也实现相同的功能。

12.2.4 Non-optimizng GCC 4.9 (ARM64)

ARM64 的指令集存在求负运算的 NEG 指令。

指令清单 12.16 Optimizing GCC 4.9 (ARM64)

```
my_abs:
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; compare input value with contents of WZR register
; (which always holds zero)
        cmp     w0, wzr
        bge     .L2
        ldr     w0, [sp,12]
        neg     w0, w0
        b       .L3
.L2:
        ldr     w0, [sp,12]
.L3:
        add     sp, sp, 16
        ret
```

12.2.5 MIPS

指令清单 12.17　Optimizing GCC 4.4.5 (IDA)

```
my_abs:
; jump if $a0<0:
                bltz    $a0, locret_10
; just return input value ($a0) in $v0:
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
locret_10:
; negate input value and store it in $v0:
                jr      $ra
; this is pseudoinstruction. in fact, this is "subu $v0,$zero,$a0" ($v0=0-$a0)
                negu    $v0, $a0
```

这里出现了新指令 BLTZ（Branch if Less Than Zero），以及伪指令 NEGU。NEGU 指令计算零减去操作数的差。SUBU 和 NEGU 指令中的后缀 U 代表它的操作数是无符号型数据，并且在整数溢出的情况下不会触发异常处理机制。

12.2.6 不使用转移指令

不使用转移指令同样可以计算绝对值。本书的第 45 章有详细说明。

12.3 条件运算符

C/C++都支持条件运算符：

表达式? 表达式: 表达式

例如：

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

12.3.1 x86

在编译含有条件运算符的语句时，早期无优化功能的编译器会以编译“if/else”语句的方法进行处理。

指令清单 12.18　Non-optimizing MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; compare input value with 10
        cmp     DWORD PTR _a$[ebp], 10
; jump to $LN3@f if not equal
        jne     SHORT $LN3@f
; store pointer to the string into temporary variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; jump to exit
        jmp     SHORT $LN4@f
$LN3@f:
```

```
; store pointer to the string into temporary variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; this is exit. copy pointer to the string from temporary variable to EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

指令清单 12.19 Optimizing MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; compare input value with 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; jump to $LN4@f if equal
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

新编译器生成的程序更为简洁。

指令清单 12.20 Optimizing MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$      = 8
f       PROC
; load pointers to the both strings
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; compare input value with 10
        cmp     ecx, 10
; if equal, copy value from RDX ("it is ten")
; if not, do nothing. pointer to the string "it is not ten" is still in RAX as for now.
        cmove   rax, rdx
        ret     0
f       ENDP
```

启用优化选项后，GCC 4.8 生成的 x86 指令同样使用了 CMOVcc 指令。相比之下，在关闭优化功能的情况下，GCC 4.8 用条件转移指令编译条件操作符。

12.3.2 ARM

启用优化功能之后，Keil 生成的 ARM 代码会应用条件运行指令 ADRcc。

指令清单 12.21 Optimizing Keil 6/2013 (ARM mode)

```
f PROC
; compare input value with 10
        CMP     r0, #0xa
; if comparison result is EQual, copy pointer to the "it is ten" string into R0
        ADREQ   r0,|L0.16| ; "it is ten"
; if comparison result is Not Equal, copy pointer to the "it is not ten" string into R0
        ADRNE   r0,|L0.28| ; "it is not ten"
        BX      lr
        ENDP

|L0.16|
```

```
        DCB     "it is ten",0
|L0.28|
        DCB     "it is not ten",0
```

除非存在人为干预，否则 ADREQ 和 ADRNE 指令不可能在同一次调用期间都被执行。

在启用优化功能之后，Keil 会给编译出的 Thumb 模式代码分配条件转移指令。毕竟在 Thumb 模式的指令之中，没有支持标志位判断的赋值指令。

指令清单 12.22　Optimizing Keil 6/2013 (Thumb mode)

```
f PROC
; compare input value with 10
        CMP     r0,#0xa
; jump to |L0.8| if EQual
        BEQ     |L0.8|
        ADR     r0,|L0.12| ; "it is not ten"
        BX      lr
|L0.8|
        ADR     r0,|L0.28| ; "it is ten"
        BX      lr
        ENDP

|L0.12|
        DCB     "it is not ten",0
|L0.28|
        DCB     "it is ten",0
```

12.3.3　ARM64

启用优化功能之后，GCC（Linaro）4.9 编译出来的 ARM64 程序同样用条件转移指令实现条件运算符。

指令清单 12.23　Optimizing GCC (Linaro) 4.9

```
f:
        cmp     x0, 10
        beq     .L3             ; branch if equal
        adrp    x0, .LC1        ; "it is ten"
        add     x0, x0, :lo12:.LC1
        ret
.L3:
        adrp    x0, .LC0        ; "it is not ten"
        add     x0, x0, :lo12:.LC0
        ret
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
```

ARM64 同样没有能够判断标志位的条件赋值指令。而 32 位的 ARM 指令集[①]，以及 x86 的 CMOVcc 指令都可以根据相应标志位进行条件赋值。虽然 ARM64 存在“条件选择”指令 CSEL（Conditional SELect），但是 GCC 4.9 似乎无法给这种程序分配上这条指令。

12.3.4　MIPS

不幸的是，GCC 4.45 在编译 MIPS 程序方面的智能程度也有待完善。

指令清单 12.24　Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
        .ascii  "it is not ten\000"
$LC1:
        .ascii  "it is ten\000"
f:
```

① 请参阅 ARM13a，p390，C5.5。

```
        li      $2,10                   # 0xa
; compare $a0 and 10, jump if equal:
        beq     $4,$2,$L2
        nop ; branch delay slot

; leave address of "it is not ten" string in $v0 and return:
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2:
; leave address of "it is ten" string in $v0 and return:
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

12.3.5 使用 if/else 替代条件运算符

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

启用优化功能之后，GCC 4.8 在编译 x86 程序时能够应用 CMOVcc 指令。

指令清单 12.25 Optimizing GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
.LFB0:
; compare input value with 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; if comparison result is Not Equal, copy EDX value to EAX
; if not, do nothing
        cmovne  eax, edx
        ret
```

Optimizing Keil 编译的 ARM 程序，与指令清单 12.21 相同。

但是启用优化功能的 MSVC 2012 仍然没有什么起色。

12.3.6 总结

启用优化功能之后，编译器会尽可能地避免使用条件转移指令。本书的 33.1 节将详细讲解这个问题。

12.4 比较最大值和最小值

12.4.1 32 位

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
```

```
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

指令清单 12.26　Non-optimizing MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; compare A and B:
        cmp     eax, DWORD PTR _b$[ebp]
; jump, if A is greater or equal to B:
        jge     SHORT $LN2@my_min
; reload A to EAX if otherwise and jump to exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; this is redundant JMP
$LN2@my_min:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; compare A and B:
        cmp     eax, DWORD PTR _b$[ebp]
; jump if A is less or equal to B:
        jle     SHORT $LN2@my_max
; reload A to EAX if otherwise and jump to exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; this is redundant JMP
$LN2@my_max:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

两个函数的唯一区别就是条件转移指令：第一个函数使用的是 JGE（Jump if Greater or Equal），而第二个函数使用的是 JLE(Jump if Less or Equal)。

上述每个函数里都存在一个多余的 JMP 指令。这可能是 MSVC 的问题。

无分支指令的编译方法

Keil 编译的 Thumb 模式程序与 x86 程序有几分相似。

指令清单 12.27 Optimizing Keil 6/2013 (Thumb mode)

```
my_max PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; branch if A is greater then B:
        BGT      |L0.6|
; otherwise (A<=B) return R1 (B):
        MOVS     r0,r1
|L0.6|
; return
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; branch if A is less then B:
        BLT      |L0.14|
; otherwise (A>=B) return R1 (B):
        MOVS     r0,r1
|L0.14|
; return
        BX       lr
        ENDP
```

两个函数所用的转移指令不同：一个是 BGT，而另一个是 BLT。

在编译 ARM 模式程序时，编译器可能会使用条件执行指令（即“有分支”指令）。这种程序会显得更为短小。在编译条件表达式时，Keil 编译器使用了 MOVcc 指令。

指令清单 12.28 Optimizing Keil 6/2013 (ARM mode)

```
my_max PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A<=B (hence, LE - Less or Equal)
; if instruction is not triggered (in case of A>B), A is still in R0 register
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
        CMP      r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A>=B (hence, GE - Greater or Equal)
; if instruction is not triggered (in case of A<B), A value is still in R0 register
        MOVGE    r0,r1
        BX       lr
        ENDP
```

在启用优化功能的情况下，GCC 4.8.1 和 MSVC 2013 都能使用 CMOVcc 指令。这个指令相当于 ARM 程序里的 MOVcc 指令。

指令清单 12.29 Optimizing MSVC 2013

```
my_max:
```

```
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
        cmp     edx, eax
; if A>=B, load A value into EAX
; the instruction idle if otherwise (if A<B)
        cmovge  eax, edx
        ret
my_min:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
        cmp     edx, eax
; if A<=B, load A value into EAX
; the instruction idle if otherwise (if A>B)
        cmovle  eax, edx
        ret
```

12.4.2　64 位

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
        if (a>b)
                return a;
        else
                return b;
};

int64_t my_min(int64_t a, int64_t b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

虽然编译出来的程序里存在不必要的数据交换，但是代码功能一目了然。

指令清单 12.30　Non-optimizing GCC 4.9.1 ARM64

```
my_max:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        ble     .L2
        ldr     x0, [sp,8]
        b       .L3
.L2:
        ldr     x0, [sp]
.L3:
        add     sp, sp, 16
        ret

my_min:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
```

```
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        bge     .L5
        ldr     x0, [sp,8]
        b       .L6
.L5:
        ldr     x0, [sp]
.L6:
        add     sp, sp, 16
        ret
```

无分支指令的编译方法

既然函数参数就在寄存器里，那么就不必通过栈访问它们。

指令清单 12.31 Optimizing GCC 4.9.1 x64

```
my_max:
; RDI=A
; RSI=B
; compare A and B:
        cmp     rdi, rsi
; prepare B in RAX for return:
        mov     rax, rsi
; if A>=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A<B)
        cmovge  rax, rdi
        ret
my_min:
; RDI=A
; RSI=B
; compare A and B:
        cmp     rdi, rsi
; prepare B in RAX for return:
        mov     rax, rsi
; if A<=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A>B)
        cmovle  rax, rdi
        ret
```

MSVC 2013 的编译方法几乎一样。

ARM64 指令集里有 CSEL 指令。它相当于 ARM 指令集中的 MOVcc 指令，以及 x86 平台的 CMOVcc 指令。它只是名字不同：“Conditional SELect”。

指令清单 12.32 Optimizing GCC 4.9.1 ARM64

```
my_max:
; X0=A
; X1=B
; compare A and B:
        cmp     x0, x1
; select X0 (A) to X0 if X0>=X1 or A>=B (Greater or Equal)
; select X1 (B) to X0 if A<B
        csel    x0, x0, x1, ge
        ret
my_min:
; X0=A
; X1=B
; compare A and B:
        cmp     x0, x1
; select X0 (A) to X0 if X0<=X1 or A<=B (Less or Equal)
; select X1 (B) to X0 if A>B
        csel    x0, x0, x1, le
        ret
```

12.4.3　MIPS

不幸的是，GCC 4.4.5 在编译 MIPS 程序方面的智能化程度有限。

指令清单 12.33　Optimizing GCC 4.4.5 (IDA)

```
my_max:
; set $v1  $a1<$a0,or clear otherwise (if $01>$a0):
                slt     $v1, $a1, $a0
; jump, if $v1 iso (or $a1>$a9):
                beqz    $v1, locret_10
; this is branch delay slot
; prepare $a1 in $v0 in case of branch triggered:
                move    $v0, $a1
; no branch triggered, prepare $a0 in $v0:
                move    $v0, $a0
locret_10:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP

; the min() function is same, but input operands in SLT instruction are swapped:
my_min
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0
locret_28:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

请注意分支延时槽现象：第一个 MOVE 指令“先于”BEQZ 指令运行，而第二个 MOVE 指令仅在不发生跳转的情况下才会被执行。

12.5　总结

条件转移指令的构造大体如下。

12.5.1　x86

指令清单 12.34　x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

12.5.2　ARM

指令清单 12.35　ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

12.5.3 MIPS

指令清单 12.36 遇零跳转

```
BEQZ REG, label
...
```

指令清单 12.37 遇负数跳转

```
BLTZ REG, label
...
```

指令清单 12.38 值相等的情况下跳转

```
BEQ REG1, REG2, label
...
```

指令清单 12.39 值不等的情况下跳转

```
BNE REG1, REG2, label
...
```

指令清单 12.40 第一个值小于第二个值的情况下跳转(signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

指令清单 12.41 第一个值小于第二个值的情况下跳转(unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

12.5.4 无分支指令（非条件指令）

如果条件语句十分短，那么编译器可能会分配条件执行指令：

- 编译 ARM 模式的程序时应用 MOVcc 指令。
- 编译 ARM64 程序时应用 CSEL 指令。
- 编译 x86 程序时应用 CMOVcc 指令。

ARM

在编译 ARM 模式的程序时，编译器可能用条件执行指令替代条件转移指令。

指令清单 12.42 ARM (ARM mode)

```
CMP register, register/value
instr1_cc ; some instruction will be executed if condition code is true
instr2_cc ; some other instruction will be executed if other condition code is true
... etc ...
```

在被执行指令不修改任何标志位的情况下，程序可有任意多条的条件执行指令。

Thumb 模式的指令集里有 IT 指令。它可以把后续四条指令构成一个指令组，并且在条件表达式为真的时候运行这组指令。详细介绍请参见本书的 17.7.2 节。

指令清单 12.43 ARM (Thumb mode)

```
CMP register, register/value
ITEEE EQ ; set these suffixes: if-then-else-else-else
```

```
instr1   ; instruction will be executed if condition is true
instr2   ; instruction will be executed if condition is false
instr3   ; instruction will be executed if condition is false
instr4   ; instruction will be executed if condition is false
```

12.6　练习题

请使用 CSFL 指令替代指令清单 12.23 中所有的条件转移语句。

第 13 章　switch()/case/default

13.1　case 陈述式较少的情况

本节将围绕这个例子进行讲解：

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f(2); //test
};
```

13.1.1　x86

Non-optimizing MSVC

使用 MSVC 2010 编译上述程序，可得到如下指令。

指令清单 13.1　MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
```

```
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

上面这个函数的源程序相当于：

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

如果仅从汇编代码入手，那么我们无法判断上述函数是一个判断表达式较少的 switch()语句、还是一组 if()语句。确实可以认为，switch()语句是一种旨在简化大量嵌套 if()语句而设计的语法糖[①]。

上面的汇编代码把输入参数 *a* 代入了临时的局部变量 tv64，其余部分的指令都很好理解。[②]

若用 GCC 4.4.1 编译器编译这个程序，无论是否启用其最大程度优化的选项“-O3”，生成的汇编代码也和 MSVC 编译出来的代码没有什么区别。

Optimizing MSVC

经指令“cl 1.c /Fa1.asm /Ox”编译上述程序，可得到如下指令。

指令清单 13.2 MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp    _printf
_f    ENDP
```

① 即 syntactic sugar，指代增强代码可读性、降低编程出错率的语法改进措施。

② MSVC 编译器在处理栈内的局部变量时，按照其需要，可能给这些内部变量分配以 tv 开头的宏变量。

我们看到，它有以下两处不同。

第一处：程序把变量 a 存储到 EAX 寄存器之后，又用 EAX 的值减去零。似乎这样做并没有什么道理。但是这两条指令可以检查 EAX 寄存器的值是否是零。如果 EAX 寄存器的值是零，ZF 标志寄存器会被置 1（也就是说 0-0=0，这就可以提前设置 ZF 标志位），并会触发第一条条件转移指令 JE，使程序跳转到 $LN4@f，继而在屏幕上打印“Zero”。如果 EAX 寄存器的值仍然不是零，则不会触发第一条跳转指令、做“EAX=EAX−1”的运算，若计算结果是零则做相应输出；若此时 EAX 寄存器的值仍然不是零，就会再做一次这种减法操作和条件判断。

如果三次运算都没能使 EAX 寄存器的值变为零，那么程序会输出最后一条信息“something unknown”。

第二处：在把字符串指针存储到变量 *a* 之后，函数使用 JMP 指令调用 printf()函数。在调用 printf()函数的时候，调用方函数而没有使用常规的 call 指令。这点不难解释：调用方函数把参数推送入栈之后，的确通常通过 CALL 指令调用其他函数。这种情况下，CALL 指令会把返回地址推送入栈、并通过无条件转移的手段启用被调用方函数。就本例而言，在被调用方函数运行的任意时刻，栈的内存存储结构为：

- ESP——指向 RA。
- ESP+4——指向变量 *a*。

另一方面，在本例程序调用 printf()函数之前、之后，除了制各第一个格式化字符串的参数问题以外，栈的存储结构其实没有发生变化。所以，编译器在分配 JMP 指令之前，把字符串指针存储到相应地址上。

这个程序把函数的第一个参数替换为字符串的指针，然后跳转到 printf()函数的地址，就好像程序没有“调用”过 f()函数、直接“转移”了 printf()函数一般。当 printf()函数完成输出的使命以后，它会执行 RET 返回指令。RET 指令会从栈中读取（POP）返回地址 RA、并跳转到 RA。不过这个 RA 不是其调用方函数——f()函数内的某个地址，而是调用 f()函数的函数即 main()函数的某个地址。换而言之，跳转到这个 RA 地址后，printf()函数会伴随其调用方函数 f()一同结束。

除非每个 case 从句的最后一条指令都是调用 printf()函数，否则编译器就做不到这种程度的优化。某种意义上说这与 longjmp()函数[①]十分相似。当然，这种优化的目的无非就是提高程序的运行速度。

ARM 编译器也有类似的优化，请参见本书的 6.2.1 节。

OllyDbg

本节讲解使用 OllyDbg 调试这个程序的具体方法。

OllyDbg 可以识别 switch()语句的指令结构，而且能够自动地添加批注。最初的时候，EAX 的值、即函数的输入参数为 2，如图 13.1 所示。

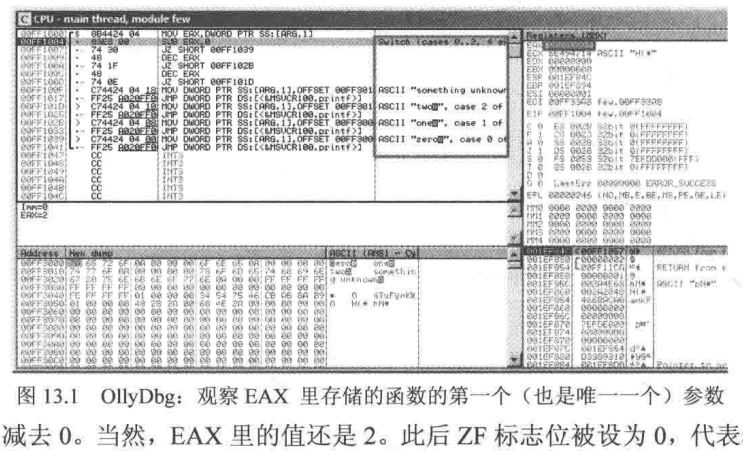


图 13.1 OllyDbg：观察 EAX 里存储的函数的第一个（也是唯一一个）参数

EAX 的值（2）减去 0。当然，EAX 里的值还是 2。此后 ZF 标志位被设为 0，代表着运算结果不是零，如图 13.2 所示。

执行 DEC 指令之后，EAX 里的值为 1。但是 1 还不是零，ZF 标志位还是 0，如图 13.3 所示。

① https://en.wikipedia.org/wiki/Setjmp.h。

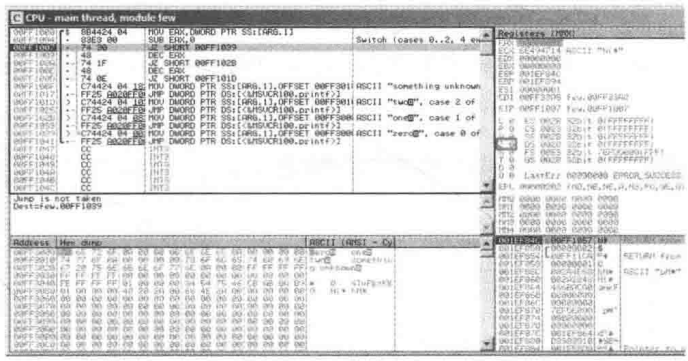


图 13.2　OllyDbg：执行第一个 SUB 指令

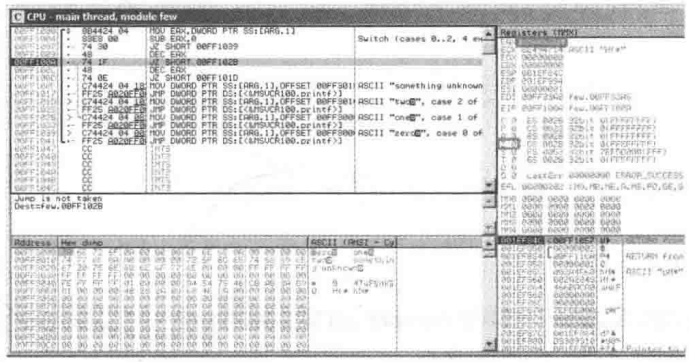


图 13.3　OllyDbg：执行第一条 DEC 指令

再次执行 DEC 指令，此时 EAX 里的值终成为是零了。因为运算结果为零，ZF 标志位被置位为 1，如图 13.4 所示。

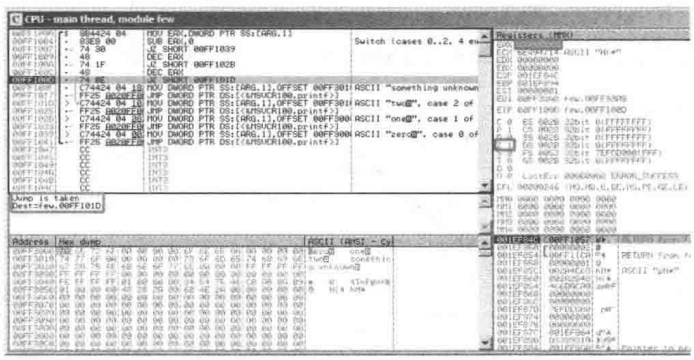


图 13.4　OllyDbg：执行第二条 DEC 指令

OllyDbg 提示将会触发条件转移指令，字符串“two”的指针即刻被推送入栈，如图 13.5 所示。

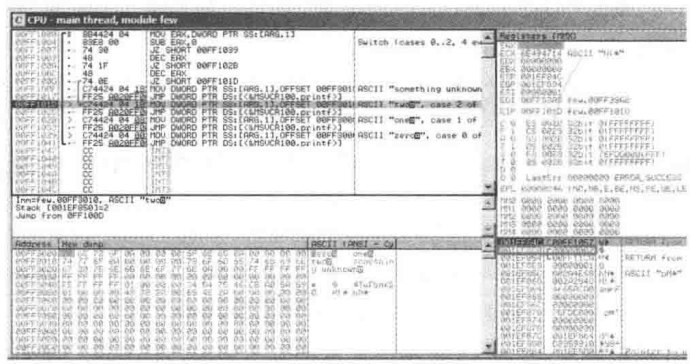


图 13.5　OllyDbg：函数的第一个参数被赋值为字符串指针

请注意：函数的当前参数是 2，它位于 0x0020FA44 处的栈。

MOV 指令把地址 0x0020FA44 的指针放入栈中，然后进行跳转。程序将执行文件 MSVCR100.DLL 里的 printf()函数的第一条指令。为了便于演示，本例在编译程序时使用了/MD 开关，如图 13.6 所示。

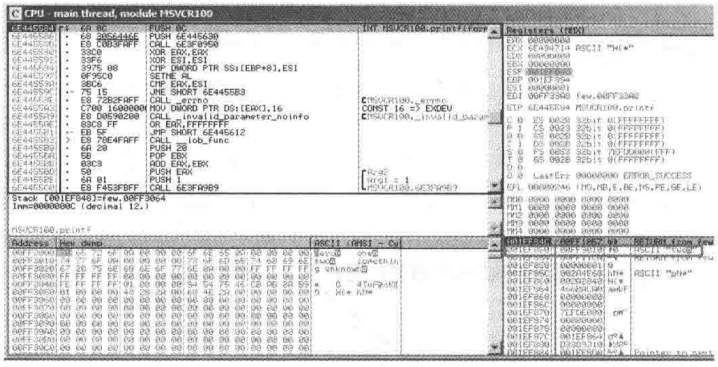


图 13.6 OllyDbg：文件 MSVCR100.DLL 中 printf()函数的第一条指令

之后，printf()函数从地址 0x00FF3010 处读取它的唯一参数——字符串地址。然后函数会 stdout 设备（一般来说，就是屏幕）上在打印字符串。

printf()函数的最后一条指令如图 13.7 所示：

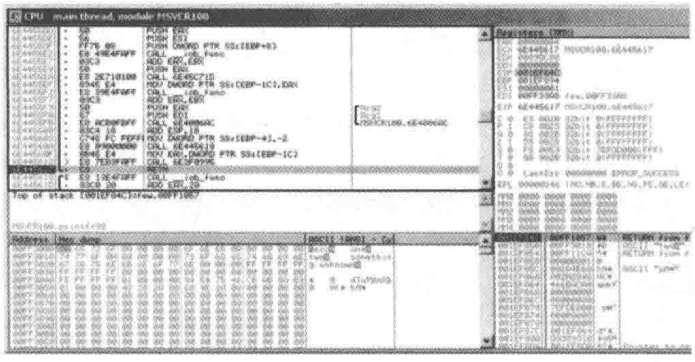


图 13.7 OllyDbg：Printf()函数的最后一条指令

此时，字符串“two”就会被输出到控制台窗口（console）。

接下来，我们使用 F7 或 F8 键、单步执行这条返回指令。然而程序没有返回 f()函数，而是回到了主函数 main()，如图 13.8 所示。

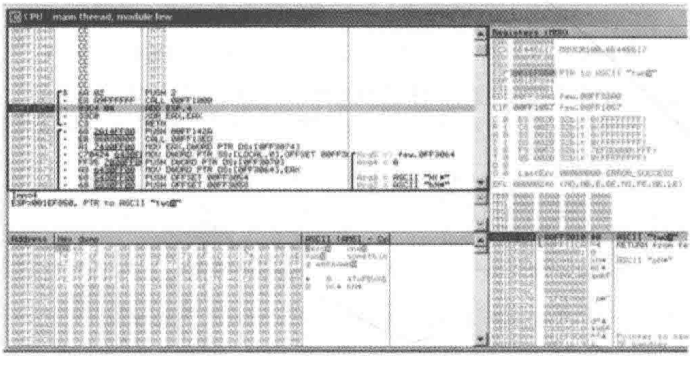


图 13.8 OllyDbg: 返回至 main()函数

正如你所看到的那样，程序从 printf()函数的内部直接返回到 main()函数。这是因为 RA 寄存器里存储

的返回地址确实不是 f()函数中的某个地址，而是 main()函数里的某个地址。请仔细观察返回地址的上一条指令，即“CALL 0X00FF1000”指令。它同样还是调用函数（即调用 f()）的指令。

13.1.2　ARM: Optimizing Keil 6/2013 (ARM mode)

```
.text:0000014C                f1:
.text:0000014C 00 00 50 E3    CMP     R0, #0
.text:00000150 13 0E 8F 02    ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A    BEQ     loc_170
.text:00000158 01 00 50 E3    CMP     R0, #1
.text:0000015C 4B 0F 8F 02    ADREQ   R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A    BEQ     loc_170
.text:00000164 02 00 50 E3    CMP     R0, #2
.text:00000168 4A 0F 8F 12    ADRNE   R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02    ADREQ   R0, aTwo  ; "two\n"
.text:00000170
.text:00000170              loc_170 ; CODE XREF: f1+8
.text:00000170                      ; f1+14
.text:00000170 78 18 00 EA    B       __2printf
```

我们同样无法根据汇编指令判断源代码使用的是 switch()语句还是 if()语句。

此外，这段代码还出现了 ADRQ 指令之类的条件执行指令。第一条 ADREQ 指令会在 R0＝0 的情况下，将字符串“zero \n”的地址传给 R0。紧接其后都 BEQ 指令在相同的条件下把控制流转交给 loc_170。或许有读者会问，ADREQ 之后的 BEQ 指令还能读取到前面由 CMP 设置的标志吗？这当然不是问题。只有少数指令才会修改标志位寄存器的值，而一般的条件执行指令不会重设任何标志位。

其余的指令不难理解。程序只在尾部调用了一次 printf()函数。前面的 6.2.1 节讲解过编译器的这种处理技术。最后，3 条逻辑分支都会收敛于同一个 printf()函数。

最后一条 CMP 指令是“CMP R0, #2”。它的作用是检查 *a* 是否为 2。如果条件不成立，程序将通过 ADRNE 指令把“something unknown \n”的指针赋值给 R0 寄存器。在此之前，程序已经检查过 *a* 是否是 0 或 1；所以运行到这里时，我们已经可以确定变量 *a* 不是这两个值。如果 R0 的值为 2，那么 ADREQ 指令将把“two”的指针传递给 R0 寄存器。

13.1.3　ARM: Optimizing Keil 6/2013 (Thumb mode)

```
.text:000000D4                f1:
.text:000000D4 10 B5          PUSH    {R4,LR}
.text:000000D6 00 28          CMP     R0, #0
.text:000000D8 05 D0          BEQ     zero_case
.text:000000DA 01 28          CMP     R0, #1
.text:000000DC 05 D0          BEQ     one_case
.text:000000DE 02 28          CMP     R0, #2
.text:000000E0 05 D0          BEQ     two_case
.text:000000E2 91 A0          ADR     R0, aSomethingUnkno ; "something unknown \n"
.text:000000E4 04 E0          B       default_case

.text:000000E6                zero_case ; CODE XREF: f1+4
.text:000000E6 95 A0          ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0          B       default_case

.text:000000EA                one_case ; CODE XREF: f1+8
.text:000000EA 96 A0          ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0          B       default_case

.text:000000EE                two_case ; CODE XREF: f1+C
.text:000000EE 97 A0          ADR     R0, aTwo ; "two\n"
.text:000000F0                default_case ; CODE XREF: f1+10
.text:000000F0                                        ; f1+14
.text:000000F0 06 F0 7E F8 BL       __2printf
.text:000000F4 10 BD          POP     {R4,  PC}
```

前文介绍说过，条件执行指令是 ARM 模式指令的特色之一。Thumb 模式的指令集里没有条件执行指令。所以 Thumb 模式的指令更为接近 x86 CISC（复杂指令集）的指令，理解难度相对较低。

13.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9

```
.LC12:
        .string "zero"
.LC13:
        .string "one"
.LC14:
        .string "two"
.LC15:
        .string "something unknown"
f12:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     w0, [x29,28]
        ldr     w0, [x29,28]
        cmp     w0, 1
        beq     .L34
        cmp     w0, 2
        beq     .L35
        cmp     w0, wzr
        bne     .L38            ; jump to default label
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        bl      puts
        b       .L32
.L34:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        bl      puts
        b       .L32
.L35:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        bl      puts
        b       .L32
.L38:
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        bl      puts
        nop
.L32:
        ldp     x29, x30, [sp], 32
        ret
```

因为输入变量属于整型数据，所以 ARM64 编译器没有给它分配 64 位的 X0 寄存器而是为它分配了容量相当的 X0 寄存器。另外，本例用 ADRP/ADD 指令对向 puts()函数传递字符串指针。前文 3.4.5 节中的“Hello,world!”也是这样传递参数的。

13.1.5 ARM64: Optimizing GCC (Linaro) 4.9

```
f12:
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; default case
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        b       puts
.L35:
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        b       puts
.L32:
```

```
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        b       puts
.L31:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        b       puts
```

优化编译的效果显著。CBZ(Compare and Branch on Zero)会在 W0 的值为零的情况下进行跳转。此外，在调用 puts()函数的时候，本例使用的是 JMP 指令而非常规的 call 指令调用，这再现了 13.1.1 节出现过的函数调用方式。

13.1.6 MIPS

指令清单 13.3 Optimizing GCC 4.4.5(IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; is it 1?
                Li      $v0, 1
                beq     $a0, $v0, loc_60
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; is it 2?
                Li      $v0, 2
                beq     $a0, $v0, loc_4C
                or      $at, $zero ; branch delay slot, NOP
; jump, if not equal to 0:
                bnez    $a0, loc_38
                or      $at, $zero ; branch delay slot, NOP
; zero case:
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9 ; branch delay slot, NOP
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot
# ---------------------------------------------------------------
loc_38:                                 # CODE XREF: f+1C
                lui     $a0, ($LC3 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "something unknown" ; branch delay slot
# ---------------------------------------------------------------
loc_4C:                                 # CODE XREF: f+14
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot
# ---------------------------------------------------------------
loc_60:                                 # CODE XREF: f+8
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot
```

在汇编层面，每个 case 分支的最后一条指令都是调用 puts()函数的指令。而且本例的每个 case 分支都通过跳转指令 JR（Jump Register）调用 puts()函数，完全没有使用常规的函数调用指令 JAL (Jump And Link)。有关详细介绍，请参阅 13.1.1 节。

另外，参数 LW 指令之后有一条 NOP 指令。这种指令组合叫作“加载延迟槽/load delay slot”，是 MIPS 平台的另一种延迟槽。在 LW 指令从内存加载数据的时候，下面的那条指令可能和它并发执行。这样一来，LW 后面的那条指令就无法使用 LW 读取的数据了。当今主流的 MIPS CPU 都针对这一问题进行了优化，

在下一条指令 LW 的数据的情况下能够进行自动处理。虽然现在的 MIPS 处理器不再存在加载延时槽，但是 GCC 还是会颇为保守地添加加载延迟槽。总之，我们已经可以忽视这种延迟槽了。

13.1.7 总结

在 case 分支较少的情况下，switch()语句和 if/else 语句的编译结果基本相同。指令清单 13.1 可充分论证这个结论。

13.2 case 陈述式较多的情况

在 switch()语句存在大量 case()分支的情况下，编译器就不能直接套用大量 JE/JNE 指令了。否则程序代码肯定会非常庞大。

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f(2); // test
};
```

13.2.1 x86

Non-optimizing MSVC

使用 MSVC 2010 编译上述程序，可得到如下指令。

指令清单 13.4 MSVC 2010

```
tv64 = -4   ; size=4
_a$ = 8     ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
```

```
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad   2; align next label
$LN11@f:
    DD     $LN6@f ; 0
    DD     $LN5@f ; 1
    DD     $LN4@f ; 2
    DD     $LN3@f ; 3
    DD     $LN2@f ; 4
_f    ENDP
```

这段代码可被分为数个调用 printf()函数的指令组，而且每组指令传递给 printf()函数的参数还各不相同。这些指令组在内存中拥有各自的起始地址，也就被编译器分配到了不同的符号标签（symbolic label）之后。总的来看，程序通过$LN11@f 处的一组数据调派这些符号标签。

函数最初把变量 *a* 的值与数字 4 进行比较。如果 *a* 大于 4，函数则跳转到$LN1@f 处，把字符串“something unknown”的指针传递给 printf()函数。

如果变量 *a* 小于或等于 4，则会计算 *a* 乘以 4 的积，再计算积与$LN11@f 的偏移量的和（表查询），并跳转到这个结果所指向的地址上。以变量 *a* 等于 2 的情况来说，2×4=8（由于 x86 系统的内存地址都是 32 位数据，所以$LN11@f 表中的每个地址都占用 4 字节）。在计算 8 与$LN11@f 的偏移量的和之后，再跳转到这个和指向的标签——即$LN4@f 处。JMP 指令最终跳转到$LN4@f 的地址。

$LN11@f 标签（偏移量）开始的表，叫作“转移表 jumptable”，也叫作“转移（输出）表 branchtable”。[①]

当 *a* 等于 2 的时候，程序分配给 printf()的参数是“two”。实际上，此时的 switch 语句的分支指令等效于“jmp DWORD PTR $LN11@f[ecx*4]”。它会进行间接取值的操作，把指针“PTR｛表达式｝”所指向的数据读取出来，当作 DWORD 型数据传递给 JMP 指令。在这个程序里，表达式的值为$LN11@f+ecx*4。

此处出现的 npad 指令属于汇编宏，本书第 88 章会详细介绍它。它的作用是把紧接其后的标签地址向 4 字节（或 16 字节）边界对齐。npad 的地址对齐功能可提高处理器的 IO 读写效率，通过一次操作即可完成内存总线、缓冲内存等设备的数据操作。

OllyDbg

接下来使用 OllyDbg 调试这个程序。我们在 EAX＝2 的时候进行调试，如图 13.9 所示。

① 这个名称来自于 Fortran 早期的 GOTO 算法。虽然现在保留了这个名称，但是已经和那个概念没什么关系了。详情请参见 http://en.wikipedia.org/wiki/Branch_table。

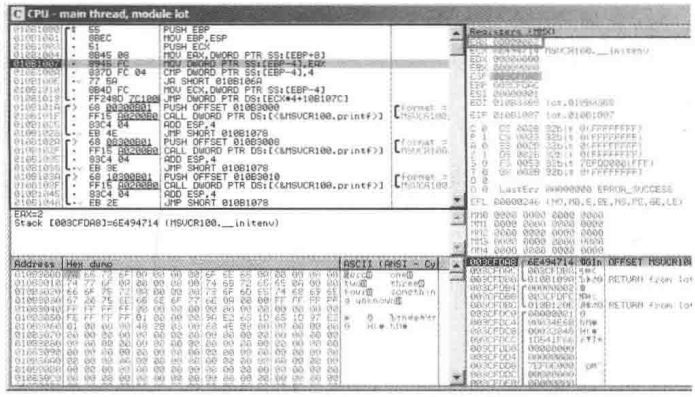


图 13.9 使用 OllyDbg 查看 EAX 获取输入值的情况

程序将检验输入值是否大于 4。因为 2<4，所以不会执行“default”规则的跳转，如图 13.10 所示。

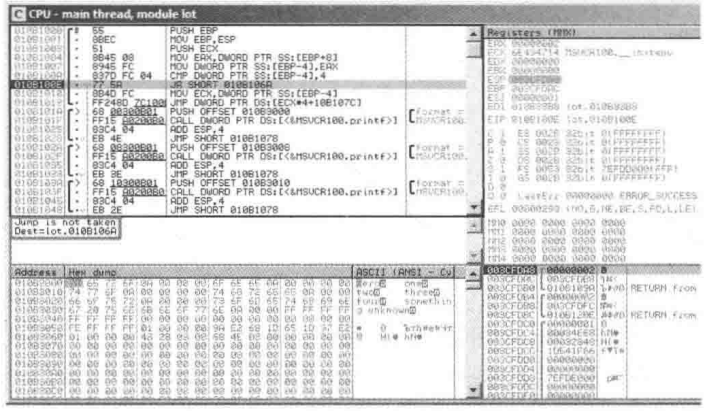


图 13.10 OllyDbgEAX≤4,不会触发 default 规则的跳转

然后就开始处理转移表，如图 13.11 所示。

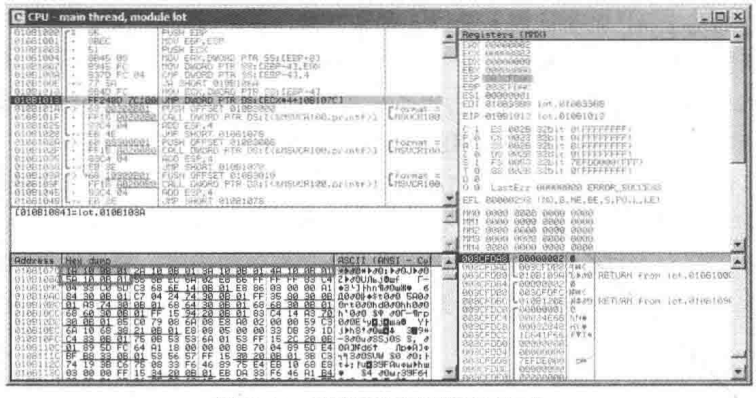


图 13.11 利用转移表计算目标地址

在用鼠标选择“Follow in Dump”→“Address constant”功能之后，即可在数据窗口看见转移表。表里有 5 个 32 位的值[①]。现在 ECX 寄存器的值是 2，所以对应表中的第 2 个元素（从 0 开始数）。另外，您还可以使用 OllyDbg 的“Follow in Dump→Memory address”功能查看 JMP 指令的目标地址。此时，这个目标地址为 0x010B103A。

地址 0x010B103A 处的指令将会打印字符串“two”，如图 13.12 所示。

① OllyDbg 用下画线的格式显示这些值，因为它们也是 FIXUPS。本书的 68.2.6 节会进行详细的解释。

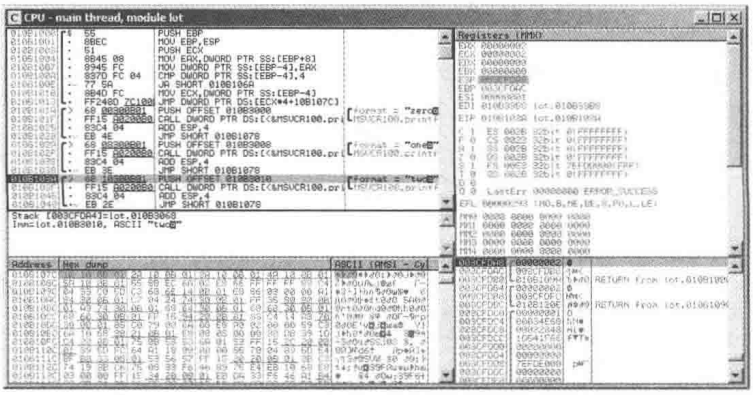


图 13.12　使用 OllyDbg 观察 case：label 的触发过程

Non-optimizing GCC

GCC 4.4.1 编译出的代码如下。

指令清单 13.5　GCC 4.4.1

```
        public  f
f       proc near ; CODE XREF: main+10

var_18  = dword ptr -18h
arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450
```

```
loc_8048444: ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
        call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

这段代码与 MSVC 编译出来的代码几乎相同。参数 arg_0 被左移 2 位（数学上等同于乘以 4，有关指令介绍请参阅 16.2.1 节），然后在转移表 off_804855C 的数组中获取相应地址，并将计算结果存储于 EAX 寄存器。最后通过 JMP EAX 指令进行跳转。

13.2.2 ARM: Optimizing Keil 6/2013 (ARM mode)

指令清单 13.6 Optimizing Keil 6/2013 (ARM mode)

```
00000174                f2
00000174  05 00 50 E3       CMP     R0, #5              ; switch 5 cases
00000178  00 F1 8F 30       ADDCC   PC, PC, R0,LSL#2    ; switch jump
0000017C  0E 00 00 EA       B       default_case        ; jumptable 00000178 default case

00000180
00000180                loc_180 ; CODE XREF: f2+4
00000180  03 00 00 EA       B       zero_case           ; jumptable 00000178 case 0

00000184
00000184                loc_184 ; CODE XREF: f2+4
00000184  04 00 00 EA       B       one_case            ; jumptable 00000178 case 1

00000188
00000188                loc_188 ; CODE XREF: f2+4
00000188  05 00 00 EA       B       two_case            ; jumptable 00000178 case 2

0000018C
0000018C                loc_18C ; CODE XREF: f2+4
0000018C  06 00 00 EA       B       three_case          ; jumptable 00000178 case 3

00000190
00000190                loc_190 ; CODE XREF: f2+4
00000190  07 00 00 EA       B       four_case           ; jumptable 00000178 case 4

00000194
00000194                zero_case ; CODE XREF: f2+4
00000194                          ; f2:loc_180
00000194  EC 00 8f E2       ADR     R0, aZero           ; jumptable 00000178 case 0
00000198  04 00 00 EA       B       loc_1B8

0000019C
0000019C                one_case  ; CODE XREF: f2+4
0000019C                          ; f2:loc_184
0000019C  EC 00 8F E2       ADR     R0, aOne            ; jumptable 00000178 case 1
000001A0  04 00 00 EA       B       loc_1B8

000001A4
```

```
000001A4                two_case  ; CODE XREF: f2+4
000001A4                          ; f2:loc_188
000001A4  01 0C 8F E2       ADR     R0, aTwo          ; jumptable 00000178 case 2
000001A8  02 00 00 EA       B       loc_1B8

000001AC
000001AC              three_case ; CODE XREF: f2+4
000001AC                         ; f2:loc_18C
000001AC  01 0C 8F E2       ADR    R0, aThree         ; jumptable 00000178 case 3
000001B0  00 00 00 EA       B      loc_1B8

000001B4
000001B4               four_case ; CODE XREF: f2+4
000001B4                        ; f2:loc_190
000001B4  01 0C 8F E2       ADR    R0, aFour          ; jumptable 00000178 case 4
000001B8
000001B8                loc_1B8   ; CODE XREF: f2+4
000001B8                          ; f2+2C
000001B8  66 18 10 EA       B      __2printf

000001BC               default_case ; CODE XREF: f2+4
000001BC                          ; f2+8
000001BC D4 00 8F E2       ADR    R0, aSomethingUnkno ; jumptable default case
000001C0 FC FF FF EA       B      loc_1B8
```

这段代码充分体现了 ARM 模式下每条汇编指令占用 4 个字节的特点。

这个程序能够识别出 4 及 4 以下的自然数。当输入值是大于 4 的整数时，程序都会显示“something unknown \n”。

第一条指令是“CMP R0, #5”。它将输入变量与 5 做比较。

“ADDCC PC, PC, R0, LSL#2”会在 R0 寄存器的值小于 5 的时候进行加法计算，其中 CC 代表借位标志 Carry Clear。如果 R0 寄存器的值不小于 5，（即 R0 大于或等于 5），则会直接跳转到标签 default_case 处。

如果 R0 寄存器的值是 5 以下的整数，那么将会触发 ADDCC，并且进行下列运算：

- 将 R0 的值乘以 4。LSL 是左移操作，左移两位（2bits）就相当于乘以 4。
- 把上述积与 PC 的值相加，并会把运算结果存储在 PC 寄存器里。这种调整 PC 指针的操作，等同于运行 B 跳转指令。
- 在执行 ADDCC 指令的时候，PC 寄存器的值会比当前指令的（首）地址提前 8 个字节。此时 ADDCC 的地址是 0x178，PC 的值为 0x180。即，PC 指向当前指令后面的第二条指令。

这是 ARM 处理器的 pipeline/流水线决定的。当 ARM 处理器执行某条指令时，处理器正在处理（fetch 取指）后面的第二条指令。实际上 PC 指向后面第二条（正在被 fetch/取指的）指令的地址。[①]

- 如果 $a==0$，“加零”操作使 PC 寄存器的值不变。所以在 PC 操作之后，CPU 会跳到 8 个字节之后的 loc_180 处继续执行后续命令，开始执行 ADDCC 指令。
- 如果 $a==1$，则 $PC+8+a\times4=PC+16=0x184$。程序会跳转到 loc_184 处。
- 依此类推，变量 a 的值每增加 1，PC 的值就会增加 4。这 4 字节是每个分支语句的唯一一条指令——B 指令的 opcode 的长度。在 ADDCC 之后，总共有 5 个 B 跳转指令。

后面的指令比较容易理解，5 条 B 跳转指令接着完成各自赋值和打印的任务，完成 switch()语句的功能。

13.2.3　ARM: Optimizing Keil 6/2013 (Thumb mode)

指令清单 13.7　Optimizing Keil 6/2013 (Thumb mode)

```
000000F6                EXPORT f2
000000F6            f2
```

① 虽然 pipeline 三级流水的解释较为直观，但是官方手册《ARM architecture reference manual》第 1 章第 2 节否认了这种硬件上的联系，它把 PC 与指令间 opcode 的增量关系解释为历史原因。

```
000000F6 10 B5              PUSH    {R4,LR}
000000F8 03 00              MOVS    R3, R0
000000FA 06 F0 69 F8        BL      __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05                 DCB 5
000000FF 04 06 08 0A 0C 10 DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00                 ALIGN 2
00000106
00000106                zero_case ; CODE XREF: f2+4
00000106 8D A0              ADR     R0, aZero ; jump table 000000FA case 0
00000108 06 E0              B       loc_118

0000010A
0000010A                one_case ; CODE XREF: f2+4
0000010A 8E A0              ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0              B       loc_118

0000010E
0000010E                two_case ; CODE XREF: f2+4
0000010E 8F A0              ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0              B       loc_118

00000112
00000112                three_case ; CODE XREF: f2+4
00000112 90 A0              ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0              B       loc_118

00000116
00000116                four_case ; CODE XREF: f2+4
00000116 91 A0              ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                loc_118 ; CODE XREF: f2+12
00000118                        ; f2+16
00000118 06 F0 6A F8        BL      __2printf
0000011C 10 BD              POP     {R4,PC}

0000011E
0000011E                default_case ; CODE XREF: f2+4
0000011E 82 A0              ADR     R0, aSomethingUnkno ; jumptable 000000FA default case
00000120 FA E7              B       loc_118

000061D0                    EXPORT __ARM_common_switch8_thumb
000061D0                __ARM_common_switch8_thumb ; CODE XREF: example6_f2+4
000061D0 78 47              BX      PC

000061D2 00 00              ALIGN 4
000061D2                ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                __32__ARM_common_switch8_thumb ; CODE XREF__ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

Thumb 和 Thumb-2 程序的 opcode 长度并不固定。这一特征更接近 x86 系统的程序。

它们的程序代码里有一个专门用于存储 case 从句信息（default 以外）的表。这个表负责记录 case 从句的数量、偏移量和标签，以便程序可以进行准确的寻址。程序通过这个表单进行相应的跳转，继而处理相应的分支 case 语句。

因为需要操作转移表并进行后续跳转，所以这个程序也使用了专用函数_ARM_common_ switch8_thumb。这个函数的第一条指令是“BX PC”，它将运行模式切换到 32 位的 ARM 模式，然后在 32 位模式下进行操作。然后函数着手表查询和分支转移单操作。具体指令非常复杂，本文在这里只是简单介绍一下，暂时不进行详解。

比较有趣的是，这个函数使用 LR 寄存器存储表的指针。在调用这个函数之后，LR 寄存器存储着“BL __ARM_common_switch8_thumb ”的后续指令的地址，也就是表开始的地址。

这个程序出现了每个 switch()陈述句都会复用的专用函数，具有显著的结构化特征。可能是编译器为了避免生成重复代码而进行的处理。

IDA 能够自动识别出这个函数和相应的转移表。IDA 还能给相应的条目加上合理的注释。举例来说，IDA 就给本例添加了“jumptable 000000FA case 0”这样的注释。

13.2.4　MIPS

指令清单 13.8　Optimizing GCC 4.4.5 (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; jump to loc_24 if input value is lesser than 5:
                sltiu   $v0, $a0, 5
                bnez    $v0, loc_24
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; input value is greater or equal to 5.
; print "something unknown" and finish:
                lui     $a0, ($LC5 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC5 & 0xFFFF)  # "something unknown" ; branch delay slot

loc_24:                                 # CODE XREF: f+8
; load address of jumptable
; LA is pseudoinstruction, LUI and ADDIU pair are there in fact:
                la      $v0, off_120
; multiply input value by 4:
                sll     $a0, 2
; sum up multiplied value and jumptable address:
                addu    $a0, $v0, $a0
; load element from jumptable:
                lw      $v0, 0($a0)
                or      $at, $zero ; NOP
; jump to the address we got in jumptable:
                jr      $v0
                or      $at, $zero ; branch delay slot, NOP

sub_44:                                 # DATA XREF: .rodata:0000012C
; print "three" and finish
                lui     $a0, ($LC3 >> 16)  # "three"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "three" ; branch delay slot

sub_58:                                 # DATA XREF: .rodata:00000130
; print "four" and finish
                lui     $a0, ($LC4 >> 16)  # "four"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC4 & 0xFFFF)  # "four" ; branch delay slot
sub_6C:                                 # DATA XREF: .rodata:off_120
; print "zero" and finish
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

sub_80:                                 # DATA XREF: .rodata:00000124
```

```
; print "one" and finish
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot

sub_94:                                 # DATA XREF: .rodata:00000128
; print "two" and finish
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

; may be placed in .rodata section:
off_120:        .word sub_6C
                .word sub_80
                .word sub_94
                .word sub_44
                .word sub_58
```

上述代码出现了 SLTIU(Set on Less Than Immediate Unsigned)指令。它和 SLTU（Set on Less Than Unsigned）的功能基本相同。请注意，这两个指令名称里差了一个“立即数（immediate）”字样。这说明前者需要在指令中指定既定的立即数。

BNEZ 是“在非零情况下进行转移/Branche if Not Equal to Zero”的缩写。

上述代码和其他指令集的代码十分相近。SLL（Shift Word Left Logical）是逻辑左移的指令，本例用它进行“乘以 4”的运算。毕竟这是一个面向 32 位 MIPS CPU 的程序，所有转移表里的所有地址都是 32 位指针。

13.2.5 总结

switch()的大体框架参见指令清单 13.9。

指令清单 13.9 x86

```
MOV REG,input
CMP REG,4 ; maximal number of cases
JA default
SHL REG,3 ; find element in table.shift for 3bits in x64.
MOV REG, jump_table[REG]
JMP REG

case1;
    ; do something
    JMP exit
case2;
    ; do something
    JMP exit
case3;
    ; do something
    JMP exit
case4;
    ; do something
    JMP exit
Case5;
    ; do something
    JMP exit

defaule:

    ...

exit:

    ...
```

```
jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

若不使用上述指令，我们也可以在 32 位系统上使用指令 JMP jump_table[REG*4]/在 64 位上使用 JMP jump_table[REG*8]，实现转移表中的寻址计算。

说到底，转移表只不过是某种指针数组它和 18.5 节介绍的那种指针数组十分雷同。

13.3 case 从句多对一的情况

多个 case 陈述从句触发同一系列操作的情况并不少见，例如：

```
#include <stdio.h>

void f (int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
        default:
                printf ("default\n");
                break;
        };
};

int main ()
{
        f(4);
};
```

如果编译器刻板地按照每种可能的逻辑分支逐一分配对应的指令组，那么程序里将会存在大量的重复指令。一般而言，编译器会通过某种派发机制来降低代码的冗余度。

13.3.1 MSVC

使用 MSVC 2010（启用/Ox 选项）编译上述程序，可得到如下指令。

指令清单 13.10 Optimizing MSVC 2010

```
1 $SG2798 DB        '1, 2, 7, 10', 0aH, 00H
2 $SG2800 DB        '3, 4, 5', 0aH, 00H
3 $SG2802 DB        '8, 9, 21', 0aH, 00H
4 $SG2804 DB        '22', 0aH, 00H
```

```
 5 $SG2806 DB        'default', 0aH, 00H
 6
 7 _a$ = 8
 8 _f      PROC
 9         mov       eax, DWORD PTR _a$[esp-4]
10         dec       eax
11         cmp       eax, 21
12         ja        SHORT $LN1@f
13         movzx     eax, BYTE PTR $LN10@f[eax]
14         jmp       DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16         mov       DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17         jmp       DWORD PTR __imp__printf
18 $LN4@f:
19         mov       DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20         jmp       DWORD PTR __imp__printf
21 $LN3@f:
22         mov       DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23         jmp       DWORD PTR __imp__printf
24 $LN2@f:
25         mov       DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26         jmp       DWORD PTR __imp__printf
27 $LN1@f:
28         mov       DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29         jmp       DWORD PTR __imp__printf
30         npad      2 ; align $LN11@f table on 16-byte boundary
31 $LN11@f:
32         DD        $LN5@f ; print '1, 2, 7, 10'
33         DD        $LN4@f ; print '3, 4, 5'
34         DD        $LN3@f ; print '8, 9, 21'
35         DD        $LN2@f ; print '22'
36         DD        $LN1@f ; print 'default'
37 $LN10@f:
38         DB        0 ; a=1
39         DB        0 ; a=2
40         DB        1 ; a=3
41         DB        1 ; a=4
42         DB        1 ; a=5
43         DB        1 ; a=6
44         DB        0 ; a=7
45         DB        2 ; a=8
46         DB        2 ; a=9
47         DB        0 ; a=10
48         DB        4 ; a=11
49         DB        4 ; a=12
50         DB        4 ; a=13
51         DB        4 ; a=14
52         DB        4 ; a=15
53         DB        4 ; a=16
54         DB        4 ; a=17
55         DB        4 ; a=18
56         DB        4 ; a=19
57         DB        2 ; a=20
58         DB        2 ; a=21
59         DB        3 ; a=22
60 _f      ENDP
```

这个程序用到了两个表：一个是索引表$LN10@f；另一个是转移表$LN11@f。

第 13 行的 movzx 指令在索引表里查询输入值。

索引表的返回值又分为 0（输入值为 1、2、7、10）、1（输入值为 3、4、5）、2（输入值为 8、9、21）、3（输入值为 22）、4（其他值）这 5 种情况。

程序把索引表的返回值作为关键字，再在第二个转移表里进行查询，以完成相应跳转（第 14 行指令的作用）。

需要注意的是，输入值为 0 的情况没有相应的 case 从句。如果 $a=0$，则“dec eax”指令会继续进行计算，而$LN10@f 表的查询是从 1 开始的。可见，没有必要为 0 的特例设置单独的表。

这是一种普遍应用的编译技术。

表面看来，这种双表结构似乎不占优势。为什么它不象前文（请参见 13.2.1 节）那样采用一个统一的指针结构呢？在这种双表结构中，索引表采用的是 byte 型数据，所以双表结构比前面那种单表结构更为紧凑。

13.3.2　GCC

在编译这种多对一的 switch 语句时，GCC 会生成统一的转移表。其代码风格和前文 13.2.1 节的风格相同。

13.3.3　ARM64: Optimizing GCC 4.9.1

因为输入值为零的情况没有对应的处理方法，所以 GCC 会从输入值为 1 的特例开始枚举各个分支，以便把转移表压缩得尽可能小。

GCC 4.9.1for ARM64 的编译技术更为优越。它能把所有的偏移量信息编码为 8 位字节型数据、封装在单条指令的 opcode 里。前文介绍过，ARM64 程序的每条指令都对应着 4 个字节的 opcode。在本例这种类型的小型代码中，各分支偏移量的具体数值不会很大。GCC 能够充分利用这一现象，构造出单字节指针组成的转移表。

指令清单 13.11　Optimizing GCC 4.9.1 ARM64

```
f14:
; input value in W0
        sub     w0, w0, #1
        cmp     w0, 21
; branch if less or equal (unsigned):
        bls     .L9
.L2:
; print "default":
        adrp    x0, .LC4
        add     x0, x0, :lo12:.LC4
        b       puts
.L9:
; load jumptable address to X1:
        adrp    x1, .L4
        add     x1, x1, :lo12:.L4
; W0=input_value-1
; load byte from the table:
        ldrb    w0, [x1,w0,uxtw]
; load address of the Lrtx label:
        adr     x1, .Lrtx4
; multiply table element by 4 (by shifting 2 bits left) and add (or subtract) to the address of lrtx
        add     x0, x1, w0, sxtb #2
; jump to the calculated address:
        br      x0
; this label is pointing in code (text) segment:
.Lrtx4:
        .section        .rodata
; everything after ".section" statement is allocated in the read-only data (rodata) segment:
.L4:
        .byte   (.L3 - .Lrtx4) / 4  ;case 1
        .byte   (.L3 - .Lrtx4) / 4  ;case 2
        .byte   (.L5 - .Lrtx4) / 4  ;case 3
        .byte   (.L5 - .Lrtx4) / 4  ;case 4
        .byte   (.L5 - .Lrtx4) / 4  ;case 5
        .byte   (.L5 - .Lrtx4) / 4  ;case 6
        .byte   (.L3 - .Lrtx4) / 4  ;case 7
        .byte   (.L6 - .Lrtx4) / 4  ;case 8
        .byte   (.L6 - .Lrtx4) / 4  ;case 9
        .byte   (.L3 - .Lrtx4) / 4  ;case 10
        .byte   (.L2 - .Lrtx4) / 4  ;case 11
        .byte   (.L2 - .Lrtx4) / 4  ;case 12
        .byte   (.L2 - .Lrtx4) / 4  ;case 13
```

```
        .byte   (.L2 - .Lrtx4) / 4  ;case 14
        .byte   (.L2 - .Lrtx4) / 4  ;case 15
        .byte   (.L2 - .Lrtx4) / 4  ;case 16
        .byte   (.L2 - .Lrtx4) / 4  ;case 17
        .byte   (.L2 - .Lrtx4) / 4  ;case 18
        .byte   (.L2 - .Lrtx4) / 4  ;case 19
        .byte   (.L6 - .Lrtx4) / 4  ;case 20
        .byte   (.L6 - .Lrtx4) / 4  ;case 21
        .byte   (.L7 - .Lrtx4) / 4  ;case 22
        .text
; everything after ".text" statement is allocated in the code (text) segment:
.L7:
; print "22"
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        b       puts
.L6:
; print "8, 9, 21"
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        b       puts
.L5:
; print "3, 4, 5"
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        b       puts
.L3:
; print "1, 2, 7, 10"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        b       puts
.LC0:
        .string "1, 2, 7, 10"
.LC1:
        .string "3, 4, 5"
.LC2:
        .string "8, 9, 21"
.LC3:
        .string "22"
.LC4:
        .string "default"
```

把上述程序编译为 obj 文件，然后再使用 IDA 打开，可看到其转移表如下。

指令清单 13.12 jumptable in IDA

```
.rodata:0000000000000064                AREA .rodata, DATA, READONLY
.rodata:0000000000000064                ; ORG 0x64
.rodata:0000000000000064 $d             DCB    9     ; case 1
.rodata:0000000000000065                DCB    9     ; case 2
.rodata:0000000000000066                DCB    6     ; case 3
.rodata:0000000000000067                DCB    6     ; case 4
.rodata:0000000000000068                DCB    6     ; case 5
.rodata:0000000000000069                DCB    6     ; case 6
.rodata:000000000000006A                DCB    9     ; case 7
.rodata:000000000000006B                DCB    3     ; case 8
.rodata:000000000000006C                DCB    3     ; case 9
.rodata:000000000000006D                DCB    9     ; case 10
.rodata:000000000000006E                DCB 0xF7     ; case 11
.rodata:000000000000006F                DCB 0xF7     ; case 12
.rodata:0000000000000070                DCB 0xF7     ; case 13
.rodata:0000000000000071                DCB 0xF7     ; case 14
.rodata:0000000000000072                DCB 0xF7     ; case 15
.rodata:0000000000000073                DCB 0xF7     ; case 16
.rodata:0000000000000074                DCB 0xF7     ; case 17
.rodata:0000000000000075                DCB 0xF7     ; case 18
.rodata:0000000000000076                DCB 0xF7     ; case 19
```

```
.rodata:0000000000000077          DCB    3     ; case 20
.rodata:0000000000000078          DCB    3     ; case 21
.rodata:0000000000000079          DCB    0     ; case 22
.rodata:000000000000007B ; .rodata ends
```

当输入值为 1 时，目标偏移量的技术方法是：9 乘以 4、再加上 Lrtx4 的偏移量。当输入值为 22 时，目标偏移量为：0 乘以 4、结果为 0。在转移表 Lrtx4 之后就是 L7 的标签的指令了，这部分指令将负责打印数字 22。请注意，转移表位于单独的.rodata 段。编译器没有把它分配到.text 的代码段里。

上述转移表有一个负数 0xF7。这个偏移量指向了打印默认字符串（.L2 标签）的相关指令。

13.4　Fall-through

Switch()语句还有一种常见的使用方法——fall-through。

```
 1 #define R 1
 2 #define W 2
 3 #define RW 3
 4
 5 void f(int type)
 6 {
 7         int read=0, write=0;
 8
 9         switch (type)
10         {
11         case RW:
12                 read=1;
13         case W:
14                 write=1;
15                 break;
16         case R:
17                 read=1;
18                 break;
19         default:
20                 break;
21         };
22         printf ("read=%d, write=%d\n", read, write);
23 };
```

如果 type 为 1（参见第一行可知，这是读取权限 R 为真的情况），则 read 的值会被设置为 1；如果 type 为 2（W），则 write 被设置 1；如果 type 为 3（RW），则 read 和 write 的值都会被设置为 1。

无论 type 的值是 RW 还是 W，程序都会执行第 14 行的指令。type 为 RW 的陈述语句里没有 break 指令，从而利用 switch 语句的 fall through 效应。

13.4.1　MSVC x86

指令清单 13.13　MSVC 2012

```
$SG1305 DB      'read=%d, write=%d', 0aH, 00H

_write$ = -12   ; size= 4
_read$  = -8    ; size= 4
tv64    = -4    ; size= 4
_type$  = 8     ; size= 4
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _read$[ebp], 0
        mov     DWORD PTR _write$[ebp], 0
        mov     eax, DWORD PTR _type$[ebp]
        mov     DWORD PTR tv64[ebp], eax
```

```
        cmp     DWORD PTR tv64[ebp], 1 ; R
        je      SHORT $LN2@f
        cmp     DWORD PTR tv64[ebp], 2 ; W
        je      SHORT $LN3@f
        cmp     DWORD PTR tv64[ebp], 3 ; RW
        je      SHORT $LN4@f
        jmp     SHORT $LN5@f
$LN4@f: ; case RW:
        mov     DWORD PTR _read$[ebp], 1
$LN3@f: ; case W:
        mov     DWORD PTR _write$[ebp], 1
        jmp     SHORT $LN5@f
$LN2@f: ; case R:
        mov     DWORD PTR _read$[ebp], 1
$LN5@f: ; default
        mov     ecx, DWORD PTR _write$[ebp]
        push    ecx
        mov     edx, DWORD PTR _read$[ebp]
        push    edx
        push    OFFSET $SG1305 ; 'read=%d, write=%d'
        call    _printf
        add     esp, 12
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

上述汇编指令与 C 语言源代码几乎一一对应。因为在$LN4@f 和$LN3@f 之间没有转移指令，所以当程序执行了$LN4@f 处的“令 read 的值为 1”的指令之后，它还会执行后面那个标签的 write 赋值指令。这也是“fall through”（滑梯）这个名字的来源：当执行完前面那个陈述语句的指令（read 赋值）之后，继续执行下一个陈述语句的指令（write 赋值）。如果 type 的值为 W，程序会执行$LN3@f 的指令，而不会执行前面那个 read 赋值指令。

13.4.2 ARM64

指令清单 13.14 GCC (Linaro) 4.9

```
.LC0:
        .string "read=%d, write=%d\n"
f:
        stp     x29, x30, [sp, -48]!
        add     x29, sp, 0
        str     w0, [x29,28]
        str     wzr, [x29,44]    ; set "read" and "write" local variables to zero
        str     wzr, [x29,40]
        ldr     w0, [x29,28]     ; load "type" argument
        cmp     w0, 2            ; type=W?
        beq     .L3
        cmp     w0, 3            ; type=RW?
        beq     .L4
        cmp     w0, 1            ; type=R?
        beq     .L5
        b       .L6              ; otherwise...
.L4: ; case RW
        mov     w0, 1
        str     w0, [x29,44]     ; read=1
.L3: ; case W
        mov     w0, 1
        str     w0, [x29,40]     ; write=1
        b       .L6
.L5: ; case R
        mov     w0, 1
        str     w0, [x29,44]     ; read=1
```

```
        nop
.L6: ; default
        adrp    x0, .LC0          ; "read=%d, write=%d\n"
        add     x0, x0, :lo12:.LC0
        ldr     w1, [x29,44]      ; load "read"
        ldr     w2, [x29,40]      ; load "write"
        bl      printf
        ldp     x29, x30, [sp], 48
        ret
```

Arm64 程序的汇编指令与 MSVC x86 的汇编指令大致相同。偏移器同样没有在标签.L4 和标签.L3 之间分配转移指令，从而形成了 Switch()语句的 fall-through 效应。

13.5　练习题

13.5.1　题目 1

13.2 节有一段 C 语言源代码。请改写这个程序，并且在不改变程序功能的前提下，让编译器生成体积更小的可执行程序。

第 14 章　循　　环

14.1　举例说明

14.1.1　x86

x86 指令集里有一条专门的 LOOP 指令。LOOP 指令检测 ECX 寄存器的值是否是 0，如果它不是 0 则将其递减，并将操作权交给操作符所指定的标签处（即跳转）。或许是因为循环指令过于复杂的缘故，我至今尚未见过直接使用 LOOP 指令将循环语句转译成汇编语句的编译器。所以，如果哪个程序直接使用 LOOP 指令进行循环控制，那它很可能就是手写的汇编程序。

C/C++语言的循环控制语句主要分为 for()、while()、do/while()语句。

我们从 for()语句开始演示。

for()语句定义了循环的初始态（计数器的初始值）、循环条件（在变量满足何等条件下继续进行循环），以及循环控制（每次循环后对变量进行什么操作，通常是递增或递减）。当然这种语句也必须声明循环体，即每次循环时要实现什么操作。简而言之，for()语句的使用方法是：

```
for (初始态; 循环条件; 循环控制)
{
    循环体;
}
```

根据 for()语句所代表的各种功能，编译器会把 for()语句在编译为 4 个相应的功能体。

我们一起来编译下面的程序：

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                printing_function(i);
        return 0;
};
```

使用 MSVC 2010 编译上述程序，可得到如下所示的指令。

指令清单 14.1　MSVC 2010

```
_i$ = -4
_main   PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2        ; 初始态
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]      ; 循环控制语句:
    add    eax, 1                       ; i 递增 1
```

```
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10      ; 判断是否满足循环条件
    jge    SHORT $LN1@main             ; 如果 i=10 则终止循环语句
    mov    ecx, DWORD PTR _i$[ebp]     ; 循环体: call f(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main             ; 跳到循环开始处
$LN1@main:                             ; 循环结束
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main   ENDP
```

上面的汇编代码可以说是中规中矩。

使用 GCC 4.4.1（未启用优化选项）编译上述程序，可得到下述汇编指令。

指令清单 14.2　GCC 4.4.1

```
main            proc near

var_20          = dword   ptr -20h
var_4           = dword   ptr -4

                push      ebp
                mov       ebp, esp
                and       esp, 0FFFFFFF0h
                sub       esp, 20h
                mov       [esp+20h+var_4], 2 ; (i) initializing
                jmp       short loc_8048476
loc_8048465:
                mov       eax, [esp+20h+var_4]
                mov       [esp+20h+var_20], eax
                call      printing_function
                add       [esp+20h+var_4], 1 ; (i) increment
loc_8048476:
                cmp       [esp+20h+var_4], 9
                jle       short loc_8048465 ; if i<=9, continue loop
                mov       eax, 0
                leave
                retn
main            endp
```

在开启优化选项（/Ox）后，MSVC 的编译结果如下。

指令清单 14.3　Optimizing MSVC

```
_main   PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main   ENDP
```

在开启优化选项后，ESI 寄存器成为了局部变量 *i* 的专用寄存器；而在通常情况下，局部变量都应当位于栈。可见，编译器会在局部变量为数不多的情况下进行这样的优化。

进行这种优化的前提条件是：被调用方函数不应当修改局部变量专用寄存器的值。当然，在本例中编译器能够判断函数 printing_function ()不会修改 ESI 寄存器的值。在编译器决定给局部变量分配专用寄存器的时候，它会在函数序言部分保存这些专用寄存器的初始状态，然后在函数尾声里还原这些寄存器的原始值。因此，您可以在本例 main()函数的序言和尾声中分别看见 PUSH ESI/POP ESI 指令。

现在启用 GCC 4.4.1 的最深程度优化选项-O3，看看生成的汇编指令。

指令清单 14.4 Optimizing GCC 4.4.1

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    printing_function
                mov     [esp+10h+var_10], 3
                call    printing_function
                mov     [esp+10h+var_10], 4
                call    printing_function
                mov     [esp+10h+var_10], 5
                call    printing_function
                mov     [esp+10h+var_10], 6
                call    printing_function
                mov     [esp+10h+var_10], 7
                call    printing_function
                mov     [esp+10h+var_10], 8
                call    printing_function
                mov     [esp+10h+var_10], 9
                call    printing_function
                xor     eax, eax
                leave
                retn
main            endp
```

GCC 把我们的循环指令给展开（分解）了。

编译器会对选代次数较少的循环进行循环分解（Loop unwinding）对处理。展开循环体以后代码的执行效率会有所提升，但是会增加程序代码的体积。

编译经验表明，展开循环体较大的循环结构并非良策。大型函数的缓存耗费的内存占有量（cache footprint）较大。[①]

我们把变量 *i* 的最大值调整到 100，看看 GCC 是否还会分解循环。

指令清单 14.5 GCC

```
                public main
main            proc near

var_20          = dword   ptr -20h

                push      ebp
                mov       ebp, esp
                and       esp, 0FFFFFFF0h
                push      ebx
                mov       ebx, 2 ; i=2
                sub       esp, 1Ch

; aligning label loc_80484D0 (loop body begin) by 16-byte border:
                nop
```

① 请参阅 Dre07，及 Int14, 3.4.1.7 节。

```
loc_80484D0:
; pass (i) as first argument to printing_function():
                mov     [esp+20h+var_20], ebx
                add     ebx, 1 ; i++
                call    printing_function
                cmp     ebx, 64h ; i==100?
                jnz     short loc_80484D0 ; if not, continue
                add     esp, 1Ch
                xor     eax, eax ; return 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

这就与 MSVC 2010 /Ox 编译出来的代码差不多了。唯一的区别是，GCC 使用 EBX 寄存器充当变量 *i* 的专用寄存器。因为 GCC 能够判断后面的被调用方函数不会修改这个专用寄存器的值，所以它才会这样分配寄存器。在 GCC 不能进行相应判断，但是还决定给局部变量分配专用寄存器的时候，它就会在使用局部变量的那个函数的序言和尾声部分添加相应指令，利用数据栈保存和恢复专用寄存器的原始值。我们可以在 main() 函数里看到这种现象：它在函数的序言和尾声部分分别存储和恢复了局部变量专用寄存器 ebx 的原始值。

14.1.2　x86:OllyDbg

我们启用 MSVC 2010 的优化选项“/Ox”和“/Ob0”来编译上述程序，然后使用 OllyDbg 调试生成的可执行文件。

如图 14.1 所示，OllyDbg 能够识别简单的循环语句，并用方括号进行标注。

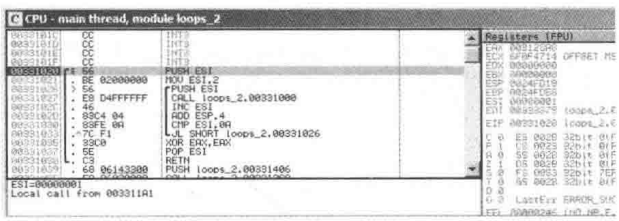


图 14.1　OllyDbg：main()函数的启动代码

我们按 F8 键进行单步调试，将会看到 ESI 寄存器的值不断递增。我们运行到 ESI 的值（变量 *i*）为 6 的时刻，如图 14.2 所示。

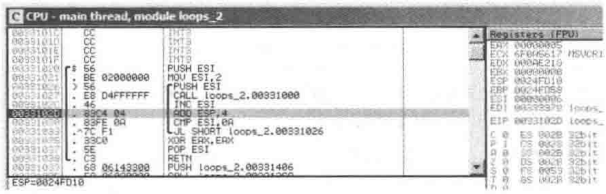


图 14.2　OllyDbg：*i*=6 时的循环体

当 *i*=9 的时候，循环语句会做最后一次迭代。进行了这次迭代之后，*i* 值变为 10，不会再触发条件转移指令 JL。main()函数结束时的情况如图 14.3 所示。

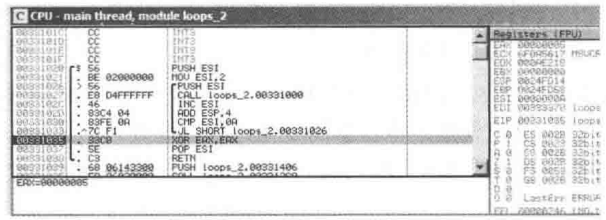


图 14.3　OllyDbg：ESI＝10 之后，循环结束

14.1.3 x86:跟踪调试工具 tracer

您可能也注意到了，直接使用 OllyDbg 这样的调试工具跟踪调试程序并不方便。所以我自己写了个调试工具 tracer。[①]

使用 IDA 打开编译后的可执行文件，找到 PUSH ESI（这条指令把参数传递给 printing_function()函数），记下其地址 0x401026，然后通过下述指令启动 tracer 程序：

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

上述指令会在 BPX 设置的断点地址处中断，输出此时各寄存器的状态。

tracer 工具会把寄存器的状态输出到 tracer.log:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

从中可以看到 ESI 寄存器的值从 2 递增到 9。

tracer 工具能够追查函数中任意地址处的寄存器状态。它名字中的 trace 就强调了其独首的追查功能。它能够在任意指令处设置断点，以记录指定寄存器的变化过程。此外，它还可以生成可被 IDA 调用的.idc 脚本文件，并为指令添加注释。例如，我们知道 main()函数的地址是 0x00401020，就可以执行

① 如果您有兴趣，可以到作者的网站上下载。http://yurichev.com/tracer-en.html。

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF 的意思就是在函数执行前设置断点。

这条指令可以生成两个脚本文件，即 loops_2.exe.idc 和 loops_2.exe_clear.idc。我们使用 IDA 加载脚本文件 loops_2.exe.idc 之后的情形如图 14.4 所示。

```
.text:00401020
.text:00401020 ; ================ S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: __tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

图 14.4　IDA 加载.idc 脚本文件

根据 Tracer 给循环体进行的注释可知：ESI 的值首先会从 2 递增到 9。在递增指令结束之后，ESI 的值再从 3 递增到 0xA（10）。另外，在 main()函数结束的时候，EAX 的值会是零。

在生成注释文件的同时，tracer 还会生成 loops_2.ext.txt。这个文件会统计每条指令被执行的次数，并且标注了各寄存器数值的变化过程。

指令清单 14.6　loops_2.exe.txt

```
0x401020 (.text+0x20), e=     1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=     1 [MOV ESI, 2]
0x401026 (.text+0x26), e=     8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=     8 [CALL 8D1000h] tracing nested maximum level (1) reached, skipping ↙
↳this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=     8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=     8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=     8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=     8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=     1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=     1 [POP ESI]
0x401038 (.text+0x38), e=     1 [RETN] EAX=0
```

我们可以使用 grep 指令搜索特定的关键字。

14.1.4　ARM

Non-optimizing Keil 6/2013 (ARM mode)

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C ; CODE XREF: main+1C
        MOV     R0, R4
        BL      printing_function
        ADD     R4, R4, #1

loc_368 ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
```

```
        LDMFD   SP!, {R4,PC}
```

上述代码中的 R4 寄存器用于存储循环计数器（即变量 *i*）。

“MOV R4, #2”给变量 *i* 进行初始化赋值。

“MOV R0, R4”和“BL printing_function”指令构成循环体。前者负责向被调用方函数传递参数，后者则直接调用 printing_function ()函数。

“ADD R4, R4, #1”指令在每次迭代之后进行 *i*++的运算。

“CMP R4, #0xA”指令会比较变量 *i* 和数字 10（十六进制的 0xA）。下一条指令 BLT（Branch Less Than）会在 *i*<10 的情况下进行跳转；否则执行“MOV R0，#0”（处理返回值），并且退出当前函数。

Optimizing Keil 6/2013 (Thumb mode)

```
_main
            PUSH        {R4,LR}
            MOVS        R4, #2

loc_132                             ; CODE XREF: _main+E
            MOVS        R0, R4
            BL          printing_function
            ADDS        R4, R4, #1
            CMP         R4, #0xA
            BLT         loc_132
            MOVS        R0, #0
            POP         {R4,PC}
```

这段代码似乎不需更多解释。

Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
_main
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOV             R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

实际上，它把 printing_function ()函数内部的指令直接代入了循环体：

```
void printing_function (int i)
{
    printf ("%d\n", i);
};
```

可见，LLVM 不仅把循环控制语句展开为 8 个顺序执行的循环体，而且还在 main()函数的循环体内直接代入了 printing_function ()函数内部的指令。当循环体调用的函数不复杂且循环的次数不高时，LLVM 编译器会把它们都拿出来进行分解和展开。

ARM64: Optimizing GCC 4.9.1

指令清单 14.7 Optimizing GCC 4.9.1

```
printing_function:
; prepare second argument of printf():
        mov     w1, w0
; load address of the "f(%d)\n" string
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; just branch here instead of branch with link and return:
        b       printf
main:
; save FP and LR in the local stack:
        stp     x29, x30, [sp, -32]!
; set up stack frame:
        add     x29, sp, 0
; save contents of X19 register in the local stack:
        str     x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
        mov     w19, 2
.L3:
; prepare first argument of printing_function():
        mov     w0, w19
; increment counter register.
        add     w19, w19, 1
; W0 here still holds value of counter value before increment.
        bl      printing_function
; is it end?
        cmp     w19, 10
; no, jump to the loop body begin:
        bne     .L3
; return 0
        mov     w0, 0
; restore contents of X19 register:
        ldr     x19, [sp,16]
; restore FP and LR values:
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"
```

ARM64: Non-optimizing GCC 4.9.1

指令清单 14.8 Non-optimizing GCC 4.9.1

```
printing_function:
; prepare second argument of printf():
        mov     w1, w0
; load address of the "f(%d)\n" string
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
```

```
; just branch here instead of branch with link and return:
        b       printf
main:
; save FP and LR in the local stack:
        stp     x29, x30, [sp, -32]!
; set up stack frame:
        add     x29, sp, 0
; save contents of X19 register in the local stack:
        str     x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
        mov     w19, 2
.L3:
; prepare first argument of printing_function():
        mov     w0, w19
; increment counter register.
        add     w19, w19, 1
; W0 here still holds value of counter value before increment.
        bl      printing_function
; is it end?
        cmp     w19, 10
; no, jump to the loop body begin:
        bne     .L3
; return 0
        mov     w0, 0
; restore contents of X19 register:
        ldr     x19, [sp,16]
; restore FP and LR values:
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"
```

14.1.5 MIPS

指令清单 14.9 Non-optimizing GCC 4.4.5 (IDA)

```
main:

; IDA is not aware of variable names in local stack
; We gave them names manually:
i               = -0x10
saved_FP        = -8
saved_RA        = -4

; function prologue:
                addiu   $sp, -0x28
                sw      $ra, 0x28+saved_RA($sp)
                sw      $fp, 0x28+saved_FP($sp)
                move    $fp, $sp
; initialize counter at 2 and store this value in local stack
                li      $v0, 2
                sw      $v0, 0x28+i($fp)
; pseudoinstruction. "BEQ $ZERO, $ZERO, loc_9C" there in fact:
                b       loc_9C
                or      $at, $zero ; branch delay slot, NOP
# ---------------------------------------------------------------------------
loc_80:                                 # CODE XREF: main+48
; load counter value from local stack and call printing_function():
                lw      $a0, 0x28+i($fp)
                jal     printing_function
                or      $at, $zero ; branch delay slot, NOP
; load counter, increment it, store it back:
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                addiu   $v0, 1
                sw      $v0, 0x28+i($fp)
```

```
loc_9C:                                 # CODE XREF: main+18
; check counter, is it 10?
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                slti    $v0, 0xA
; if it is less than 10, jump to loc_80 (loop body begin):
                bnez    $v0, loc_80
                or      $at, $zero ; branch delay slot, NOP
; finishing, return 0:
                move    $v0, $zero
; function epilogue:
                move    $sp, $fp
                lw      $ra, 0x28+saved_RA($sp)
                lw      $fp, 0x28+saved_FP($sp)
                addiu   $sp, 0x28
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

上述代码中的“B”指令属于 IDA 生成的伪指令。它原本的指令是 BEQ。

14.1.6　其他

您可能注意到了，在对循环控制变量 i 进行初始化赋值之后，程序首先检查变量 i 是否满足循环条件。在满足循环表达式的情况下，才会运行循环体的指令。这恐怕是有意而为之。如果循环变量的初始值不能满足循环条件，那么整个循环体都不会被执行。例如，下面例子中的循环体就可能不会被执行：

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

如果变量 total_entries_to_process 等于零，就不应当执行循环体。所以，循环控制语句有必要首先检查控制变量的初始值是否满足循环条件。

在启用优化选项之后，如果编译器能够判断循环语句的初始状态不可能满足循环体的执行条件，那么它可能根本不会生成该循环体的条件检测指令和循环体的对应指令。在启用编译器的优化功能之后，在编译本例的源程序时，Keil、Xcode（LLVM）、MSVC 都能做到这种程度的优化。

14.2　内存块复制

目前的计算机多数采取了 SIMD 和矢量化等技术，在内存中复制数据时，能够在每次迭代中复制 4～8 个字节。本节将从一个尽可能简单的例子开始演示。

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

14.2.1　编译结果

指令清单 14.10　GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block

; initialize counter (i) at 0
        xor     eax, eax
```

```
.L2:
; all bytes copied? exit then:
        cmp     rax, rdx
        je      .L5
; load byte at RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; store byte at RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

指令清单 14.11 GCC 4.9 ARM64 optimized for size (-Os)

```
my_memcpy:
; X0 = destination address
; X1 = source address
; X2 = size of block

; initialize counter (i) at 0
        mov     x3, 0
.L2:
; all bytes copied? exit then:
        cmp     x3, x2
        beq     .L5
; load byte at X1+i:
        ldrb    w4, [x1,x3]
; store byte at X1+i:
        strb    w4, [x0,x3]
        add     x3,x3,1         ; i++
        b       .L2
.L5:
        ret
```

指令清单 14.12 Optimizing Keil 6/2013 (Thumb mode)

```
my_memcpy PROC
; R0 = destination address
; R1 = source address
; R2 = size of block

        PUSH    {r4, lr}
; initialize counter (i) at 0
        MOVS    r3,#0
; condition checked at the end of function, so jump there:
        B       |L0.12|
|L0.6|
; load byte at R1+i:
        LDRB    r4,[r1,r3]
; store byte at R1+i:
        STRB    r4,[r0,r3]
; i++
        ADDS    r3,r3,#1
|L0.12|
; i<size?
        CMP     r3,r2
; jump to the loop begin if its so:'
        BCC     |L0.6|
        POP     {r4,pc}
        ENDP
```

14.2.2 编译为 ARM 模式的程序

在编辑 ARM 模式的应用程序时，Keil 能够充分利用相应指令集的条件执行指令：

指令清单 14.13 Optimizing Keil 6/2013 (ARM mode)

```
my_memcpy PROC
; R0 = destination address
```

```
; R1 = source address
; R2 = size of block

; initialize counter (i) at 0
      MOV     r3,#0
|L0.4|
; all bytes copied?
      CMP     r3,r2
; the following block is executed only if "less than" condition,
; i.e., if R2<R3 or i<size.
; load byte at R1+i:
      LDRBCC  r12,[r1,r3]
; store byte at R1+i:
      STRBCC  r12,[r0,r3]
; i++
      ADDCC   r3,r3,#1
; the last instruction of the "conditional block".
; jump to loop begin if i<size
; do nothing otherwise (i.e., if i>=size)
      BCC     |L0.4|
; return
      BX      lr
      ENDP
```

32 位的 ARM 程序只用了一个转移指令，比 thumb 程序少了一个转移指令。

14.2.3 MIPS

指令清单 14.14 GCC 4.4.5 optimized for size (-Os) (IDA)

```
my_memcpy:
; jump to loop check part:
                b       loc_14
; initialize counter (i) at 0
; it will always reside in \$v0:
                move    $v0, $zero ; branch delay slot

loc_8:                                  # CODE XREF: my_memcpy+1C
; load byte as unsigned at address in $t0 to $v1:
                lbu     $v1, 0($t0)
; increment counter (i):
                addiu   $v0, 1
; store byte at $a3
                sb      $v1, 0($a3)

loc_14:                                 # CODE XREF: my_memcpy
; check if counter (i) in $v0 is still less then 3rd function argument ("cnt" in $a2):
                sltu    $v1, $v0, $a2
; form address of byte in source block:
                addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; jump to loop body if counter sill less then "cnt":
                bnez    $v1, loc_8
; form address of byte in destination block (\$a3 = \$a0+\$v0 = dst+i):
                addu    $a3, $a0, $v0 ; branch delay slot
; finish if BNEZ wasnt triggered:'
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

这里序言介绍的两个指令分别是 LBU（Load Byte Unsigned）和 SB（Store Byte）。与 ARM 模式的程序相似，所有的 MIPS 寄存器都是 32 位寄存器。在这两个平台上，我们无法像 x86 平台上那样直接对寄存器的低位进行直接操作。因此在对单个字节进行操作时，我们还是得访问整个 32 位寄存器。LBU 指令加载字节型数据并且会清除寄存器的其他位。另外由于有符号数据存在符号位的问题，所以在处理有符号数的时候还要使用 LB（Load Byte）指令把单字节的有符号数据扩展为等值的 32 位数据、再存储在寄存器里。SB 指令的作用是把寄存器里的低 8 位复制到内存里。

14.2.4 矢量化技术

25.1.2 节会详细介绍 GCC 的优化编译方式基于矢量化技术（Vectorization）的。

14.3 总结

当循环控制变量从 2 递增到 9 时，循环语句对应的汇编代码大体如下。

指令清单 14.15 x86

```
    mov [counter], 2 ; initialization
    jmp check
body:
    ; loop body
    ; do something here
    ; use counter variable in local stack
    add [counter], 1 ; increment
check:
    cmp [counter], 9
    jle body
```

如果没有开启编译器的优化编译功能，那么控制变量的递增操作可能对应着 3 条汇编指令。

指令清单 14.16 x86

```
    MOV [counter], 2 ; initialization
    JMP check
body:
    ; loop body
    ; do something here
    ; use counter variable in local stack
    MOV REG, [counter] ; increment
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

当循环体较为短小时，编译器可能给循环控制变量分配专用的寄存器。

指令清单 14.17 x86

```
    MOV EBX, 2 ; initialization
    JMP check
body:
    ; loop body
    ; do something here
    ; use counter in EBX, but do not modify it!
    INC EBX ; increment
check:
    CMP EBX, 9
    JLE body
```

编译器还可能调换部分指令的顺序。

指令清单 14.18 x86

```
    MOV [counter], 2 ; initialization
    JMP label_check
label_increment:
    ADD [counter], 1 ; increment
label_check:
    CMP [counter], 10
```

```
    JGE exit
    ; loop body
    ; do something here
    ; use counter variable in local stack
    JMP label_increment
exit:
```

通常情况下，程序应当首先判断循环条件是否满足，然后再执行循环体。但是在编译器确定第一次迭代肯定会发生的情况下，它可能会调换循环体和判断语句的顺序。下面这个程序就是个典型的例子。

指令清单 14.19　x86

```
    MOV REG, 2 ; initialization
body:
    ; loop body
    ; do something here
    ; use counter in REG, but do not modify it!
    INC REG ; increment
    CMP REG, 10
    JL body
```

编译器不会使用 LOOP 指令。由 LOOP 控制的循环控制语句比较少见。如果某段代码带有 LOOP 指令，那么您应当认为这是手写出来的汇编程序。

指令清单 14.20　x86

```
    ; count from 10 to 1
    MOV ECX, 10
body:
    ; loop body
    ; do something here
    ; use counter in ECX, but do not modify it!
    LOOP body
```

ARM 平台的 R4 寄存器专门用于存储循环控制变量。

指令清单 14.21　ARM

```
    MOV R4, 2 ; initialization
    B check
body:
    ; loop body
    ; do something here
    ; use counter in R4, but do not modify it!
    ADD R4,R4, #1 ; increment
check:
    CMP R4, #10
    BLT body
```

14.4　练习题

14.4.1　题目 1

为什么现在的编译器不再使用 LOOP 指令了？

14.4.2　题目 2

使用您喜欢的操作系统和编译器编译 14.1.1 节的样本程序，然后修改这个可执行程序，使循环变量 i 可以从 6 递增到 20。

14.4.3 题目 3

请描述下述代码的功能。

指令清单 14.22 由 MSVC 2010 （启用/Ox 选项）编译而得的代码

```
$SG2795 DB      '%d', 0aH, 00H

_main   PROC
        push    esi
        push    edi
        mov     edi, DWORD PTR __imp__printf
        mov     esi, 100
        npad    3; align next label
$LL3@main:
        push    esi
        push    OFFSET $SG2795  ; '%d'
        call    edi
        dec     esi
        add     esp, 8
        test    esi, esi
        jg      SHORT $LL3@main
        pop     edi
        xor     eax, eax
        pop     esi
        ret0
_main   ENDP
```

指令清单 14.23 Non-optimizing Keil 6/2013 (ARM mode)

```
main PROC
        PUSH    {r4,lr}
        MOV     r4,#0x64
|L0.8|
        MOV     r1,r4
        ADR     r0,|L0.40|
        BL      __2printf
        SUB     r4,r4,#1
        CMP     r4,#0
        MOVLE   r0,#0
        BGT     |L0.8|
        POP     {r4,pc}
        ENDP

|L0.40|
        DCB     "%d\n",0
```

指令清单 14.24 Non-optimizing Keil 6/2013 (Thumb mode)

```
main PROC
        PUSH    {r4,lr}
        MOVS    r4,#0x64
|L0.40|
        MOVS    r1,r4
        ADR     r0,|L0.24|
        BL      __2printf
        SUBS    r4,r4,#1
        CMP     r4,#0
        BGT     |L0.4|
        MOVS    r0,#0
        POP     {r4,pc}
        ENDP

        DCW     0x0000
|L0.24|
```

```
        DCB     "%d\n",0
```

指令清单 14.25 Optimizing GCC 4.9 (ARM64)

```
main:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        stp     x19, x20, [sp,16]
        adrp    x20, .LC0
        mov     w19, 100
        add     x20, x20, :lo12:.LC0
.L2:
        mov     w1, w19
        mov     x0, x20
        bl      printf
        subs    w19, w19, #1
        bne     .L2
        ldp     x19, x20, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "%d\n"
```

指令清单 14.26 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
main:

var_18          = -0x18
var_C           = -0xC
var_8           = -8
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x28
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x28+var_4($sp)
                sw      $s1, 0x28+var_8($sp)
                sw      $s0, 0x28+var_C($sp)
                sw      $gp, 0x28+var_18($sp)
                la      $s1, $LC0       # "%d\n"
                li      $s0, 0x64  # 'd'

loc_28:                                 # CODE XREF: main+40
                lw      $t9, (printf & 0xFFFF)($gp)
                move    $a1, $s0
                move    $a0, $s1
                jalr    $t9
                addiu   $s0, -1
                lw      $gp, 0x28+var_18($sp)
                bnez    $s0, loc_28
                or      $at, $zero
                lw      $ra, 0x28+var_4($sp)
                lw      $s1, 0x28+var_8($sp)
                lw      $s0, 0x28+var_C($sp)
                jr      $ra
                addiu   $sp, 0x28

$LC0:           .ascii "%d\n"<0>        # DATA XREF: main+1C
```

14.4.4 题目 4

请描述下述代码的功能。

指令清单 14.27 Optimizing MSVC 2010

```
$SG2795 DB      '%d', 0aH, 00H

_main   PROC
        push    esi
        push    edi
        mov     edi, DWORD PTR __imp_printf
        mov     esi, 1
        npad    3; align next label
$LL3@main:
        push    esi
        push    OFFSET $SG2795  ; '%d'
        call    edi
        add     esi, 3
        add     esp, 8
        cmp     esi, 100
        jl      SHORT $LL3@main
        pop     edi
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

指令清单 14.28 Non-optimizing Keil 6/2013 (ARM mode)

```
main  PROC
        PUSH    {r4,lr}
        MOV     r4,#1
|L0.8|
        MOV     r1,r4
        ADR     r0,|L0.40|
        BL      __2printf
        ADD     r4,r4,#3
        CMP     r4,#0x64
        MOVGE   r0,#0
        BLT     |L0.8|
        POP     {r4,pc}
        ENDP

|L0.40|
        DCB     "%d\n",0
```

指令清单 14.29 Non-optimizing Keil 6/2013 (Thumb mode)

```
main  PROC
        PUSH    {r4,lr}
        MOVS    r4,#1
|L0.4|
        MOVS    r1,r4
        ADR     r0,|L0.24|
        BL      __2printf
        ADDS    r4,r4,#3
        CMP     r4,#0x64
        BLT     |L0.4|
        MOVS    r0, #0
        POP     {r4,pc}
        ENDP

        DCW     0x0000
|L0.40|
        DCB     "%d\n",0
```

指令清单 14.30 Optimizing GCC 4.9 (ARM64)

```
main:
        stp     x29, x30, [sp, -32]!
```

```
        add     x29, sp, 0
        stp     x19, x20, [sp,16]
        adrp    x20, .LC0
        mov     w19, 1
        add     x20, x20, :lo12:.LC0
.L2:
        mov     w1, w19
        mov     x0, x20
        add     w19, w19, 3
        bl      printf
        cmp     w19, 100
        bne     .L2
        ldp     x19, x20, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "%d\n"
```

指令清单 14.31 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
main:

var_18      = -0x18
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4

            lui         $gp, (__gnu_local_gp >> 16)
            addiu       $sp, -0x28
            la          $gp, (__gnu_local_gp & 0xFFFF)
            sw          $ra, 0x28+var_4($sp)
            sw          $s2, 0x28+var_8($sp)
            sw          $s1, 0x28+var_C($sp)
            sw          $s0, 0x28+var_10($sp)
            sw          $gp, 0x28+var_18($sp)
            la          $s2, $LC0       # "%d\n"
            li          $s0, 1
            li          $s1, 0x64  # 'd'
loc_30:                                 # CODE XREF: main+48
            lw          $t9, (printf & 0xFFFF)($gp)
            move        $a1, $s0
            move        $a0, $s2
            jalr        $t9
            addiu       $s0, 3
            lw          $gp, 0x28+var_18($sp)
            bne         $s0, $s1, loc_30
            or          $at, $zero
            lw          $ra, 0x28+var_4($sp)
            lw          $s2, 0x28+var_8($sp)
            lw          $s1, 0x28+var_C($sp)
            lw          $s0, 0x28+var_10($sp)
            jr          $ra
            addiu       $sp, 0x28

$LC0:       .ascii   "%d\n"<0>          # DATA XREF: main+20
```

第 15 章　C 语言字符串的函数

15.1　strlen()

本章是循环控制语句的具体应用。通常，strlen()函数由循环语句 while()语句实现。参照 MSVC 标准库对 strlen()函数的定义方法，我们讨论下述的程序：

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

15.1.1　x86

Non-optimizing MSVC

使用 MSVC 2010 编译上述程序可得到：

```
_eos$ = -4                              ; size = 4
_str$ = 8                               ; size = 4
_strlen PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _str$[ebp] ; 将指针指向字符串
    mov    DWORD PTR _eos$[ebp], eax ; 指向局部变量 eos
$LN2@strlen_:
    mov    ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign extension

    movsx  edx, BYTE PTR [ecx]
    mov    eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add    eax, 1                    ; EAX++
    mov    DWORD PTR _eos$[ebp], eax ; EAX 还原为 EOS
    test   edx, edx                  ; EDX = 0 ?
    je     SHORT $LN1@strlen_        ; yes, then finish loop
    jmp    SHORT $LN2@strlen_        ; continue loop
$LN1@strlen_:
    ; here we calculate the difference between two pointers

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                    ; subtract 1 and return result
    mov    esp, ebp
    pop    ebp
    ret    0
```

```
_strlen_ ENDP
```

这里出现了两个新指令：MOVSX 和 TEST。

此处，MOVSX 指令从内存中读取 8 位（单字节）数据，并把它存储到 32 位寄存器里。MOVSX 是 MOV with Sign-Extend 的缩写。在把小空间数据转换为大空间数据时，存在填充高位数据的问题；本例中，MOVSX 将用原始数据的 8 位数据填充 EDX 寄存器的低 8 位；如果原始数据是负数，该指令将使用 1 填充第 8 到第 31 位（高 24 位），否则使用 0 填充高 24 位。

这是为了保证有符号型数据在类型转换后的数值保持不变。

根据 C/C++标准，char 类型数据是 signed（有符号型）数据。现在设想一下把 char 型数据转换为 int 型数据（都是有符号型数据）的情况：假如 char 型数据的原始值是−2（0xFE），直接把整个字节复制到 int 型数据的最低 8 位上时，int 型数据的值就变成 0x000000FE，以有符号型数据的角度看它被转换为 254 了，而没有保持原始值−2。−2 对应的 int 型数据是 0xFFFFFFFE。所以，在把原始数据复制到目标变量之后，还要使用符号标志位填充剩余的数据，而这就是 MOVSX 的功能。

本书第 30 章会更为详细地介绍有符号型数据的表示方法。

虽然不太确定寄存器是否有必要分配 EDX 寄存器专门保存 char 型数据，看上去它只使用了寄存器的低 8 位空间（相当于 DL）。显然，编译器根据自身的寄存器分配规则进行了相应分配。

您将在后面看到“TEST EDX，EDX”指令。本书会在第 19 章详细介绍 TEST 指令。在本例中，它的功能是检查 EDX 的值是否是零，并设置相应的标志位。

Non-optimizing GCC

使用 GCC 4.4.1（未启用优化选项）编译上述程序，可得到：

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax
loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

GCC 编译的结果和 MSVC 差不多。这里它没有使用 MOVSX 指令，而是用了 MOVZX 指令。MOVZX 是 MOV with Zero-Extent 的缩写。在将 8 位或 16 位数据转换为 32 位数据的时候，它直接复制原始数据到目标寄存器的相应低位，并且使用 0 填充剩余的高位。拆文解字的角度来看，这条指令相当于一步完成了“xor eax, eax”和“mov al，[源 8/16 位数据]”2 条指令。

另一方面，编译器可以生成“mov al, byte ptr [eax] / test al, al”这样的代码，但是这样一来 EAX 寄存器的高

位将会存在随机的噪声。不如说，这就是编译器的短板所在——在它生成汇编代码的时候，它不会照顾汇编代码的（人类）可读性。严格说来，编译器编译出来的代码本来就是给机器运行的，而不是给人阅读的。

本例还出现了未介绍过的 SETNZ 指令。从前面一条指令开始解释：如果 AL 的值不是 0，则“test al, al”指令会设置标志寄存器 ZF=0；而 SETNZ（Not Zero）指令会在 ZF＝0 的时候，设置 AL＝1。用白话解说，就是：如果 AL 不等于 0，则跳到 loc_80483F0 处。编译器转译出来的代码中，有些代码确实没有实际意义，这是因为我们没有开启优化选项。

Optimizing MSVC

使用 MSVC（启用优化选项/Ox /Ob0）编译上述程序，可得到如下所示的指令。

指令清单 15.1 Optimizing MSVC 2012 /Ob0

```
_str$ = 8                                       ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4]     ;用 EDX 作字符串指针
        mov     eax, edx                        ;  复制到 EAX

$LL2@strlen:
        mov     cl, BYTE PTR [eax]              ; CL = *EAX
        inc     eax                             ; EAX++
        test    cl, cl                          ; CL==0?
        jne     SHORT $LL2@strlen               ; no, continue loop
        sub     eax, edx                        ; 计算指针的变化量
        dec     eax                             ; decrement EAX
        ret     0
_strlen ENDP
```

优化编译生成的程序短了很多。不必多说，只有在函数较短且局部变量较少的情况下，编译器才会做出这种程度的优化。

inc/dec 指令就是递增、递减指令，换句话说它们相当于运算符“＋＋”“—”。

Optimizing MSVC + OllyDbg

我们使用 OllyDbg 打开 MSVC 优化编译后的可执行文件。程序在进行第一次迭代时的情况如图 15.1 所示。

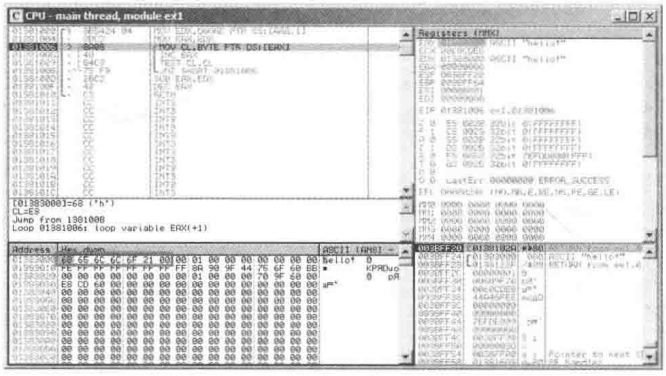


图 15.1 OllyDbg：第一次迭代

OllyDbg 识别出了循环结构，并且用方括号把整组循环体指令标示了出来。如需 OllyDbg 在内存窗口里显示对应的数据，可用右键单击 EAX 寄存器，然后选择“Follow in Dump”，OllyDbg 将自动滚动到对应的地址。在内存数据的显示窗口里，我们可以看到字符串“hello!”。字符串会用数值为零的字节做结尾，在零之后的数据就是随机的噪音数据。如果 OllyDbg 发现某个寄存器的值是指向这片内存地址（某处）的指针，它就会把对应的字符串提示出来。

然后不停地按 F8 键，等待循环体进入下一次循环。

如图 15.2 所示，EAX 寄存器的值是个指针，它指向字符串的第二个字符的地址。

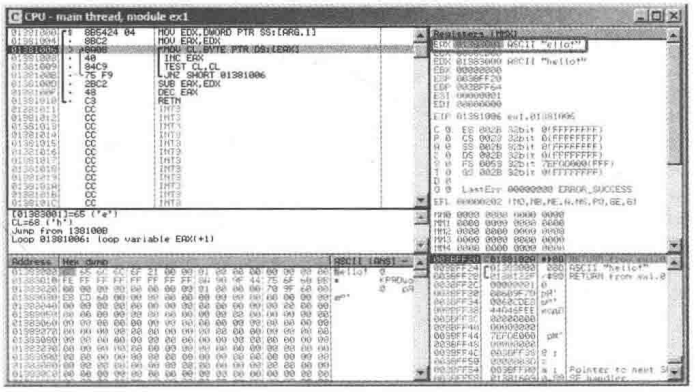


图 15.2　OllyDbg：第二次迭代

继续按 F8 键，等待循环语句执行完毕。

如图 15.3 所示，EAX 寄存器里的指针最终指向了数值为零的字符串终止字符。在循环过程中，EDX 寄存器的值始终没有发生变化，它一直是字符串首地址的指针。在循环语句结束后，程序即将计算两个指针的差值。

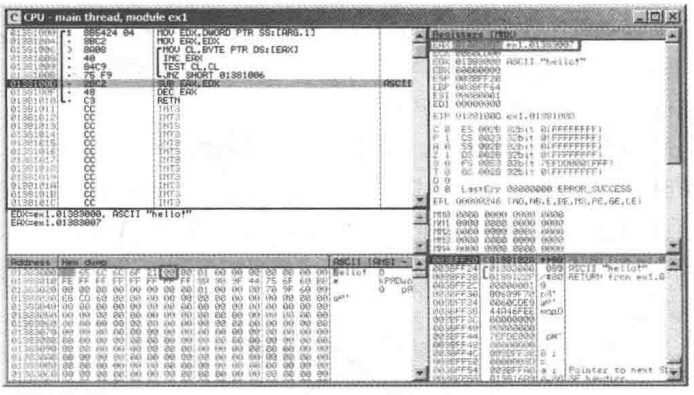


图 15.3　OllyDbg：即将计算指针（地址）之间的差值

把这两个寄存器的值（指针）相减，再减去 1，就可得到字符串的长度，如图 15.4 所示。

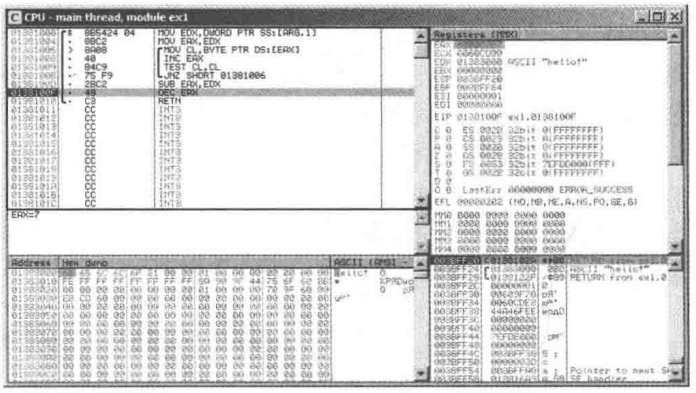


图 15.4　OllyDbg：EAX 递减

经计算，指针之间的差值为 7。实际上我们的字符串“hello!”只有 6 个字节，算上结束标志字节才是 7 个字节。很明显字符串以外的那个数值为零的字节不应当纳入字符串的长度，所以函数最后做了递减运算，把这个字节去掉。

Optimizing GCC

使用 GCC 4.4.1 编译（启用优化选项“-O3”）上述程序，可得到：

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

GCC 编译的结果和 MSVC 差不多，主要区别体现在 MOVZX 指令上。

即使把这条 MOVZX 指令替换为“mov dl, byte ptr [eax]”也没问题。

GCC 编译器的代码生成器这样做，或许是为了便于“记住”整个寄存器已经分配给了 char 型变量，以保证寄存器的高地址位（bits）不会含有噪音数据。

接下来将要介绍的是 NOT 指令。NOT 指令对操作数的所有位（bit）都进行非运算。可以说，这条指令和 XOR ECX, 0xffffffffh 指令等价。“not ecx”的结果与某数相加，相当于某数减去 ECX 和 1。在程序开始的时候，ECX 保存了字符串首个字符的地址（指针），EAX 寄存器存储的是终止符的地址。对 ECX 求非、再与 eax 相加，就是在计算 eax-ecx-1 的值。这种运算可以得到正确的字符串长度。

其中的数学问题，请参见第 30 章。

换句话说，在执行完循环语句之后，函数进行了如下操作：

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

它得到的结果与下述运算完全相同：

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

GCC 编译器这么编译代码的具体原因不明。我们能够确定的是，即使 GCC 和 MSVC 分别选择不同的算法，计算的结果肯定相同。

15.1.2 ARM

32bit ARM

Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

指令清单 15.2 Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
_strlen
```

```
eos  =-8
str  =-4

     SUB    SP, SP, #8 ; allocate 8 bytes for local variables
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; deallocate 8 bytes
     BX     LR
```

如果不指定优化选项，LLVM 编译器所生成的代码会很啰唆。但是这种冗长的代码有助于我们理解它处理局部变量所用的栈结构。本例只用到了 2 个局部变量，即 eos 和 str。

在 IDA 的反编译结果中，我把变量 var_8 重命名为原始变量名 eos，把 var_4 重命名为 str。

程序的前几条指令把输入值传递给变量 str 和 eos。

循环体的起始地址是 loc_2CB8。

循环体内的前 3 条指令（LDR，ADD，STR）把 eos 的值保存在 R0 寄存器里，然后将这个值递增（+1），再把修改后的值直接赋值给栈内的局部变量 eos。

接着“LDRSB RO,[R0]”（Load Register Signed Byte）指令从 R0 所指向的地址处读取 1 个字节，并将之转换为 32 位有符号数据（Keil 也把 Char 型数据视为有符号数据）。这条指令类似于 x86 的 MOVSX 指令（参见 15.1.1 节）。既然在 C 标准里 char 类型数据是 Signed 类型数据，编译器就把这个值当作 Signed 型数据处理。

应当注意的是，虽然 x86 系统可以把一个 32 位寄存器拆分为 8 位寄存器或 16 位寄存器单独调用，但是 ARM 系统没有把寄存器拆解出来、分别使用的助记符。确切地说，x86 的这种特性是兼容性的要求：为了运行 16 位 8086 指令甚至是 8 位 8080 指令，它的指令集必须向下兼容。而 ARM 系统的处理器最初就是 32 位 RISC 处理器。所以，在使用 ARM 系统的寄存器时，只能把它当作完整的 32 位寄存器使用。

此后，LDRSB 指令把字符串中的字符逐字节地传递到 R0 寄存器。其后的 CMP 和 BEQ 指令，会检查这个字节是不是零字节。如果这个字节不是 0，则再次进行循环；如果这个字节是 0，则结束循环语句。

函数在结束以前计算 eos 和 str 的差值，再把这个差值减去 1，然后作为函数的返回值、保存在 R0 寄存器里。

整个函数没有把寄存器推送入栈的操作。按照 ARM 调用函数的约定，R0～R3 寄存器的作用是传递参数的暂存寄存器；函数在退出以后它们已经完成使命了，也不必恢复它们的初始值。在被调用方函数结束之后，再怎么操作它们都行。另外，这个程序没有用到其他的寄存器，也没必要使用栈。所以在函数结束时，唯一需要做的工作就是使用跳转指令（BX）把控制权交给 LR 寄存器保存的返回地址。

Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

指令清单 15.3　Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_strlen
            MOV         R1, R0

loc_2DF6
```

```
LDRB.W    R2, [R1],#1
CMP       R2, #0
BNE       loc_2DF6
MVNS      R0, R0
ADD       R0, R1
BX        LR
```

在进行优化编译的时候，编译器认为寄存器已经足够用了，不必使用栈来保管变量 eos 和 str。在开始循环之前，编译器使用 R0 寄存器存储变量 str、用 R1 寄存器存储变量 eos。

“LDRB.W R2, [R1],#1”从 R1 所指向的地址里读取 1 个字节，将其转换为 32 位有符号数据（signed）之后存储在 R2 寄存器里。指令末端的立即数＃1 将在上述操作之后，把 R1 寄存器的值加 1（递增）。这种指令属于延迟索引寻址（Post-indexed address，又称后变址寻址）指令，便于操作数组。

本书的 28.2 节会详细介绍延迟索引寻址。

循环体的 CMP 和 BNE 指令负责判断循环结束的条件。它们在读取到数值为零的字节之前会保证循环语句处于的迭代状态。

MVNS（MoVe Not）指令相当于 x86 指令集中的 NOT 指令，与后面的 ADD 指令配合完成“eos-str-1”的运算。在 15.1.1 节里，我们详细介绍过其中的各种细节，这里不再复述。

LLVM 编译器与 GCC 都认为这种代码的效率更高。

Optimizing Keil 6/2013 (ARM mode)

使用 Keil（启用优化选项）编译上述程序，可得到如下所示的指令。

指令清单 15.4 Optimizing Keil 6/2013 (ARM mode)

```
_strlen
            MOV   R1, R0

loc_2C8
            LDRB  R2, [R1],#1
            CMP   R2, #0
            SUBEQ R0, R1, R0
            SUBEQ R0, R0, #1
            BNE   loc_2C8
            BX    LR
```

这段代码与前面的 LLVM 优化编译生成的 Thumb 代码相似。区别在于前面的例子在循环结束后才运算 str−eos−1，而本例则是在循环体内进行表达式演算。前文介绍过，条件执行指令中的−EQ 后缀表示其运行条件为“当前面的 CMP 比较的两个值相等/EQ 时，才会执行该指令”。所以，如果 R0 寄存器的值是 0，则会触发 CMP 指令之后的两条 SUBEQ 指令。其运算结果会被保留在 R0 寄存器，迭代结束之后就自然成为函数的返回值。

ARM64

Optimizing GCC (Linaro) 4.9

```
my_strlen:
        mov     x1, x0
        ; X1 is now temporary pointer (eos), acting like cursor
.L58:
        ; load byte from X1 to W2, increment X1 (post-index)
        ldrb    w2, [x1],1
        ; Compare and Branch if NonZero: compare W2 with 0, jump to .L58 if it is not
        cbnz    w2, .L58
        ; calculate difference between initial pointer in X0 and current address in X1
        sub     x0, x1, x0
        ; decrement lowest 32-bit of result
        sub     w0, w0, #1
```

```
        ret
```

本例的算法与指令清单 15.11 的算法相同。它计算首字符和终止符地址的差值，然后再从差里减去 1。指令清单中的注释详细介绍了具体的演算方法。不过，我的源程序有问题：my_strlen()的返回值是 32 位 int 型数据，但是它应当返回 size_t 或者另外一种 64 位数据。理论上讲，在 64 位平台上，strlen()函数的参数地址可能是 4GB 以上的内存地址，所以它在 64 位平台上的返回值应当是 64 位数据。由于源程序存在这个缺陷，所以最后一条 SUB 指令只对寄存器的低 32 位进行操作，但是倒数第二条 SUB 指令依然是对完整的 64 位寄存器进行减法运算。虽说程序存在缺陷，但是出于演示的目的，我还是把这个样本保留了下来。

Non-optimizing GCC (Linaro) 4.9

```
my_strlen:
; function prologue
        sub     sp, sp, #32
; first argument (str) will be stored in [sp,8]
        str     x0, [sp,8]
        ldr     x0, [sp,8]
; copy "str" to "eos" variable
        str     x0, [sp,24]
        nop
.L62:
; eos++
        ldr     x0, [sp,24]  ; load "eos" to X0
        add     x1, x0, 1    ; increment X0
        str     x1, [sp,24]  ; save X0 to "eos"
; load byte from memory at address in X0 to W0
        ldrb    w0, [x0]
; is it zero? (WZR is the 32-bit register always contain zero)
        cmp     w0, wzr
; jump if not zero (Branch Not Equal)
        bne     .L62
; zero byte found. now calculate difference.
; load "eos" to X1
        ldr     x1, [sp,24]
; load "str" to X0
        ldr     x0, [sp,8]
; calculate difference
        sub     x0, x1, x0
; decrement result
        sub     w0, w0, #1
; function epilogue
        add     sp, sp, 32
        ret
```

关闭优化功能之后，编译器生成的代码就长了许多。在操作变量时，程序频繁地访问内存中的栈。此外，由于源程序的设计缺陷，最后一条 SUB 指令只对寄存器的低 32 位数据进行了减法运算。

15.1.3　MIPS

指令清单 15.5　Optimizing GCC 4.4.5 (IDA)

```
my_strlen:
; "eos" variable will always reside in $v1:
                move    $v1, $a0
loc_4:
; load byte at address in "eos" into $a1:
                lb      $a1, 0($v1)
                or      $at, $zero ; load delay slot, NOP
; if loaded byte is not zero, jump to loc_4:
                bnez    $a1, loc_4
; increment "eos" anyway:
                addiu   $v1, 1 ; branch delay slot
```

```
; loop finished. invert "str" variable:
                nor     $v0, $zero, $a0
; $v0=-str-1
                jr      $ra
; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
                addu    $v0, $v1, $v0 ; branch delay slot
```

MIPS 的指令集里没有求非（NOT）运算指令，但是它有非或 NOR（即 OR+NOT）指令。在数字电路领域，非或运算电路十分普遍；但是在计算机编程领域，它还比较冷门。借助非或运算指令，求非的 NOT 运算指令可由“NOR DST,$ZERO,SRC”指令变相实现。

前文介绍过，求非运算相当于“求负、再减 1”的运算。与后面的 ADDU 指令配合，可完成“eos−str−1”的运算，求得正确的字符串长度。

15.2 练习题

15.2.1 题目 1

请描述下列代码的功能。

指令清单 15.6 Optimizing MSVC 2010

```
_s$   = 8
_f    PROC
          mov  edx, DWORD PTR _s$[esp-4]
          mov  cl, BYTE PTR [edx]
          xor  eax, eax
          test cl, cl
          je   SHORT $LN2@f
          npad 4    ; align next label
$LL4@f:
          cmp  cl, 32
          jne  SHORT $LN3@f
          inc  eax
$LN3@f:
          mov  cl, BYTE PTR [edx+1]
          inc  edx
          test cl, cl
          jne  SHORT $LL4@f
$LN2@f:
          ret  0
_f        ENDP
```

指令清单 15.7 GCC 4.8.1 -O3

```
f:
.LFB24:
        push    ebx
        mov     ecx, DWORD PTR [esp+8]
        xor     eax, eax
        movzx   edx, BYTE PTR [ecx]
        test    dl, dl
        je      .L2
.L3:
        cmp     dl, 32
        lea     ebx, [eax+1]
        cmove   eax, ebx
        add     ecx, 1
        movzx   edx, BYTE PTR [ecx]
        test    dl, dl
        jne     .L3
.L2:
        pop     ebx
```

```
        ret
```

指令清单 15.8　Optimizing Keil 6/2013 (ARM mode)

```
f   PROC
        MOV     r1,#0
|L0.4|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        MOVEQ   r0,r1
        BXEQ    lr
        CMP     r2,#0x20
        ADDEQ   r1,r1,#1
        ADD     r0,r0,#1
        B       |L0.4|
        ENDP
```

指令清单 15.9　Optimizing Keil 6/2013 (Thumb mode)

```
f PROC
        MOVS    r1,#0
        B       |L0.12|
|L0.4|
        CMP     r2,#0x20
        BNE     |L0.10|
        ADDS    r1,r1,#1
|L0.10|
        ADDS    r0,r0,#1
|L0.12|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        BNE     |L0.4|
        MOVS    r0,r1
        BX      lr
        ENDP
```

指令清单 15.10　Optimizing GCC 4.9 (ARM64)

```
f:
        ldrb    w1, [x0]
        cbz     w1, .L4
        mov     w2, 0
.L3:
        cmp     w1, 32
        ldrb    w1, [x0,1]!
        csinc   w2, w2, w2, ne
        cbnz    w1, .L3
.L2:
        mov     w0, w2
        ret
.L4:
        mov     w2, w1
        b       .L2
```

指令清单 15.11　Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                lb          $v1, 0($a0)
                or          $at, $zero
                beqz        $v1, locret_48
                li          $a1,0x20#''
                b           loc_28
                move        $v0, $zero
loc_18:                                 # CODE XREF: f:loc_28
                lb          $v1, 0($a0)
                or          $at, $zero
                beqz        $v1, locret_40
                or          $at, $zero
```

```
loc_28:                                 # CODE XREF: f+10
                                        # f+38
                bne     $v1, $a1, loc_18
                addiu   $a0, 1
                lb      $v1, 0($a0)
                or      $at, $zero
                bnez    $v1, loc_28
                addiu   $v0, 1
locret_40:                              # CODE XREF: f+20
                jr      $ra
                or      $at, $zero
locret_48:                              # CODE XREF: f+8
                jr      $ra
                move    $v0, $zero
```

第 16 章　数学计算指令的替换

出于性能优化的考虑，编译器可能会将 1 条数学运算指令替换为其他的 1 条、甚至是一组等效指令。
例如 LEA 指令通常替代其他的简单计算指令：请参见附录 A.6.2 节。
ADD 和 SUB 指令同样可以相互替换。例如，指令清单 52.1 的第 18 行就是如此。

16.1　乘法

16.1.1　替换为加法运算

我们通过下述程序进行演示。

指令清单 16.1　Optimizing MSVC 2010

```
unsigned int f(unsigned int a)
{
        return a*8;
};
```

如果使用 MSVC 2010（启用/Ox）进行编译，编译器会把“乘以 8”的运算指令拆解为 3 条加法指令。很明显 MSVC 认为替换后的程序性能更高。

```
_TEXT   SEGMENT
_a$ = 8                         ; size = 4
_f      PROC
; File c:\polygon\c\2.c
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

16.1.2　替换为位移运算

编译器通常会把“乘以 2”“除以 2”的运算指令处理为位移运算指令：

```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

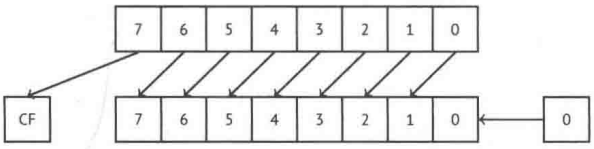
指令清单 16.2　Non-optimizing MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
```

```
_f      ENDP
```

“乘以 4”的运算就是把被乘数左移 2 位、再把位移产生的空缺位添上 0 的运算。就好比在心算计算 3×100 的时候，我们就是在 3 后面的空缺位添加两个零。

位移指令的示意图如下所示。



右侧产生的空缺位都由零填充。

乘以 4 的 ARM 指令如下所示。

指令清单 16.3 Non-optimizing Keil 6/2013 (ARM mode)

```
f  PROC
        LSL      r0,r0,#2
        BX       lr
        ENDP
```

对应的 MIPS 指令如下所示。

指令清单 16.4 Optimizing GCC 4.4.5 (IDA)

```
                jr      $ra
                sll     $v0, $a0, 2 ; branch delay slot
```

其中，SLL 是逻辑左移“Shift Left Logical”的缩写。

16.1.3 替换为位移、加减法的混合运算

即使乘数是 7 或 17，乘法运算仍然可以用非乘法运算指令配合位移指令实现。其算术原理也不难推测。

32 位

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

指令清单 16.5 Optimizing MSVC 2012

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
```

```
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

ARM

ARM 指令集的位移运算指令有 3 个操作符，可在位移运算符后再进行一次加法运算。故而 Keil 生成的代码更为短小。

指令清单 16.6　Optimizing Keil 6/2013 (ARM mode)

```
; a*7
||f1|| PROC
        RSB       r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        BX        lr
        ENDP

; a*28
||f2|| PROC
        RSB       r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        LSL       r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
        BX        lr
        ENDP

; a*17
||f3|| PROC
        ADD     r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
        BX      lr
        ENDP
```

然而 Thumb 模式的位移指令不支持这种运算。编译器无法优化 f2()。

指令清单 16.7 Optimizing Keil 6/2013 (Thumb mode)

```
; a*7
||f1|| PROC
        LSLS     r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS     r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX       lr
        ENDP

; a*28
||f2|| PROC
        MOVS     r1,#0x1c ; 28
; R1=28
        MULS     r0,r1,r0
; R0=R1*R0=28*a
        BX       lr
        ENDP

; a*17
||f3|| PROC
        LSLS     r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS     r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX       lr
        ENDP
```

MIPS

指令清单 16.8 Optimizing GCC 4.4.5 (IDA)

```
_f1:
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
                sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
                jr      $ra
                addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64 位

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};
int64_t f2(int64_t a)
{
```

```
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

指令清单 16.9　Optimizing MSVC 2012

```
; a*7
f1:
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

ARM64

由于 ARM64 模式的位移指令同样可进行加法运算，所以这种程序要短一些。

指令清单 16.10　Optimizing GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
        lsl     x1, x0, 3
; X1=X0<<3=X0*8=a*8
        sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
        ret

; a*28
f2:
        lsl     x1, x0, 5
; X1=X0<<5=a*32
        sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
        ret

; a*17
f3:
        add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
        ret
```

16.2 除法运算

16.2.1 替换为位移运算

本节围绕下述源程序进行演示：

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

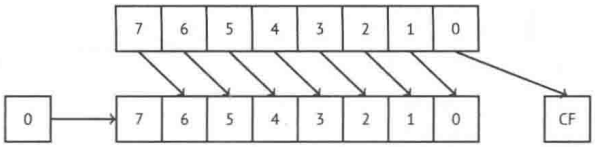
使用 MSVC 2010 编译上述程序，可得到如下所示的指令。

指令清单 16.11 MSVC 2010

```
_a$ = 8                                         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

上述代码中的 SHR（SHift Right）指令将 EAX 寄存器中的数值右移 2 位，使用零填充位移产生的空缺位，并且舍弃那些被右移指令移动出存储空间的比特位。实际上，那些被舍弃的比特位正是除法运算的余数。

SHR 指令和 SHL 指令的运算模式相同，只是位移方向不同。



右移与余数的关系与十进制运算的移动小数点运算相似：当被除数为 23、除数为 10 时，我们将 23 的小数点左移一位，2 为商、3 为余数。

毕竟本例是整数运算，商也应当是整数（不会是浮点数），所以我们舍弃余数完全没有问题。

“除以 4”运算的 ARM 程序如下所示。

指令清单 16.12 Non-optimizing Keil 6/2013 (ARM mode)

```
f  PROC
        LSR     r0,r0,#2
        BX      lr
        ENDP
```

“除以 4”运算的 MIPS 程序如下所示。

指令清单 16.13 Optimizing GCC 4.4.5 (IDA)

```
                jr      $ra
                srl     $v0, $a0, 2 ; branch delay slot
```

上述程序中的 SRL 指令是“逻辑右移指令/Shift-Right Logical”。

16.3 练习题

16.3.1 题目 1

请分析下列程序的功能。

指令清单 16.14　Optimizing MSVC 2010

```
_a$ = 8
_f      PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [ecx*8]
        sub     eax, ecx
        ret     0
_f      ENDP
```

指令清单 16.15　Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
        RSB     r0,r0,r0,LSL #3
        BX      lr
        ENDP
```

指令清单 16.16　Non-optimizing Keil 6/2013 (Thumb mode)

```
f PROC
        LSLS    r1,r0,#3
        SUBS    r0,r1,r0
        BX      lr
        ENDP
```

指令清单 16.17　Optimizing GCC 4.9 (ARM64)

```
f:
        lsl     w1, w0, 3
        sub     w0, w1, w0
        ret
```

指令清单 16.18　Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                sll     $v0, $a0, 3
                jr      $ra
                subu    $v0, $a0
```

第 17 章　FPU

FPU 是专门处理浮点数的运算单元，是 CPU 的一个组件。

在早期的计算机体系中，FPU 位于 CPU 之外的单独的运算芯片上。

17.1　IEEE 754

IEEE 754 标准规定了计算机程序设计环境中的二进制和十进制的浮点数的交换、算术格式以及方法。符合这种标准的浮点数由符号位、尾数（又称为有效数字、小数位）和指数位构成。

17.2　x86

事先接触过 stack machine[①]或 Forth 语言编程基础[②]的读者，理解本节内容的速度会比较快。

在 80486 处理器问世之前，FPU（与 CPU 位于不同的芯片）叫作协作（辅助）处理器。而且那个时候的 FPU 还不属于主板的标准配置；如果想要在主板上安装 FPU，人们还得单独购买它。[③]

80486 DX 之后的 CPU 处理器集成了 FPU 的功能。

若没有 FWAIT 指令和 opcode 以 D8～DF 开头的所谓的“ESC”字符指令（opcode 以 D8～DF 开头），恐怕很少有人还会想起 FPU 属于独立运算单元的这段历史。FWAIT 指令的作用是让 CPU 等待 FPU 运算结束，而 ESC 字符指令都在 FPU 上执行。

FPU 自带一个由 8 个 80 位寄存器构成的循环栈。这些 80 位寄存器用以存储 IEEE 754 格式的浮点数据[④]，通常叫作 ST(0)～ST(7)寄存器。IDA 和 OllyDbg 都把 ST(0)显示为 ST。也有不少教科书把 ST(0)叫作“栈顶/Stack Top”寄存器。

17.3　ARM、MIPD、x86/x64 SIMD

在 ARM 和 MIPS 平台的概念里，FPU 寄存器不构成栈结构，仅仅是一组寄存器。x86/64 构架的 SIMD 扩展（单指令多数据流扩展）也延承了这种理念。

17.4　C/C++

标准 C/C++语言支持两种浮点类型数据，即单精度 32 位浮点数据（float）和双精度 64 位浮点数据（double）。

GCC 编译器还支持 long double 类型浮点，即 80 位增强精度的扩展浮点类型数据（extended precision）。不过 MSVC 编译器不支持这种类型的浮点数据。[⑤]

---

① 中文叫“堆栈机”或“堆栈结构机”。请参见 http://en.wikipedia.org/wiki/Stack_machine。

② 请参见 http://en.wikipedia.org/wiki/Forth_%28programming_language%29。

③ 当初，为了使没有 FPU 的 32 位计算机（例如 80386\80486 SX）兼容其研发的 DOOM 游戏，John Carmack 设计了一套“软”FPU 运算系统。这种系统使用 CPU 寄存器的高 16 位地址存储整数部分、低 16 位地址存储浮点数值的小数部分，仅通过标准 32 位通用寄存器即可实现浮点运算。更多详情请参阅：https://en.wikipedia.org/wiki/Fixed-point_arithmetic。

④ 如需详细了解 IEEE 754 格式规范，请参见 http://en.wikipedia.org/wiki/IEEE_754-2008。

⑤ 如需详细了解这三种不同精度的数据类型，请参见：
http://en.wikipedia.org/wiki/Single-precision_floating-point_format。
http://en.wikipedia.org/wiki/Double-precision_floating-point_format。
http://en.wikipedia.org/wiki/Extended_precision。

虽然单精度浮点（float）型数据和整数（int）型数据在 32 位系统里都是 32 位数据，但是它们的数据格式完全不一样。

17.5 举例说明

本节围绕下述例子进行讲解：

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
  };
```

17.5.1 x86

MSVC

使用 MSVC 编译上述程序，可得到如下所示的指令。

指令清单 17.1 MSVC 2010：f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f PROC
   push    ebp
   mov     ebp, esp
   fld     QWORD PTR _a$[ebp]

; current stack state: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.14

    fld    QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided by 3.14

    fmul   QWORD PTR __real@4010666666666666

; current stack state:
; ST(0) = result of _b * 4.1;
; ST(1) = result of _a divided by 3.14

    faddp ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD 指令从栈中读取 8 个字节，把这个值转换为 FPU 寄存器所需的 80 位数据格式，并存入 ST(0)寄存器。

FDIV 指令把 ST(0)寄存器的值用作被除数，把参数__real@40091eb851eb851f（即 3.14）的值当作除数，进行除法运算。因为汇编语法不支持含有小数点的浮点立即数，所以程序使用 64 位 IEEE 754 格式的 16 进制数 040091eb851eb851fr 表示 3.14 。

在进行 FDIV 运算之后，ST(0)寄存器将保存商。

此外，FDIVP 也是 FPU 的除法运算指令。FDIVP 在进行 ST(1)/ST(0)运算时，先把两个寄存器的值 POP 出来进行运算，再把商推送入(PUSH)FPU 的栈（即 ST(0)寄存器）。这相当于 Forth 语言[①]中的堆栈机[②]。

下一条 FLD 指令，把变量 *b* 的值推送到 FPU 的栈。

此时，ST(1)寄存器里是上次除法运算的商，ST(0)寄存器里是变量 *b* 的值。

接下来的 FMUL 指令做乘法运算。它用 ST(0)寄存器里的值（即变量 *b*），乘以参数__real@4010666666666666（即 4.1），并将运算结果（积）存储到 ST(0)寄存器。

最后一条运算指令 FADDP 计算栈内顶部两个值的和。它先把运算结果存储在 ST(1)寄存器，再 POP ST(1)。所以，运算表达式的运算结果存储在栈顶的 ST(0)寄存器里。

根据有关规范，函数必须使用 ST（0）寄存器存储浮点运算的返回结果。所以在 FADDP 指令之后，除了函数尾声的指令之外再无其他指令。

MSVC + OllyDbg

图 17.1 标记出了两对 32 位数据。这两对数据都是由 main()函数传递过来的、以 IEEE 754 格式存储的双精度浮点数据。我们可看到首条 FLD 指令从栈内读取了 1.2、然后把它推送到 ST(0)。

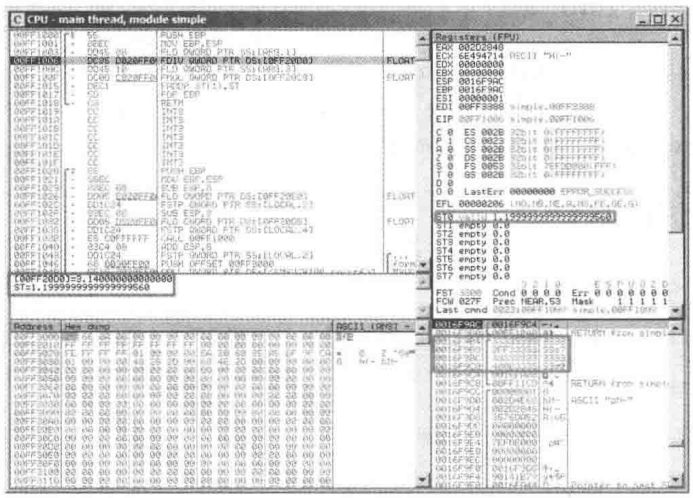


图 17.1 OllyDbg：执行首条 FLD 指令

在把 64 位 IEEE 754 格式的数据转换为 FPU 所用的 80 位浮点数据的过程中，会不可避免地存在误差。图中 1.1999……所要表示的量，就是 1.2 的近似值。此后，EIP 寄存器的值指向下一条指令 FDIV。FDIV 指令会从内存中读取双精度浮点常量。OllyDbg 人性化地显示出了第二个参数的值——3.14。

继续执行 FDIV 指令。如图 17.2 所示，此时 ST(0)寄存器存储着上一次运算的商 0.382…

执行第三条指令即 FLD 指令之后，ST(0)寄存器加载了数值 3.4（3.39999……）。如图 17.3 所示。

在 3.4 入栈的同时，先前运算出来的商被推送到 ST(1)寄存器，而后 EIP 指针的值指向下一条指令 FMUL。如同 OllyDbg 提示的那样，FMUL 指令会从内存中读取因子 4.1。

① https://en.wikipedia.org/wiki/Forth_(programming_language)。

② Stack machines，https://en.wikipedia.org/wiki/Stack_machine。

图 17.2　OllyDbg：执行 FDIV 指令

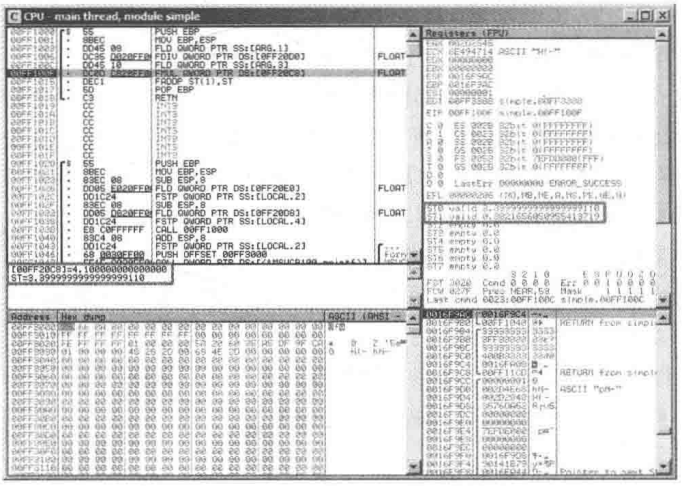


图 17.3　OllyDbg：执行第二个 FLD 指令

在执行 FMUL 指令之后，ST（0）寄存器将存储着乘法运算的积，如图 17.4 所示。

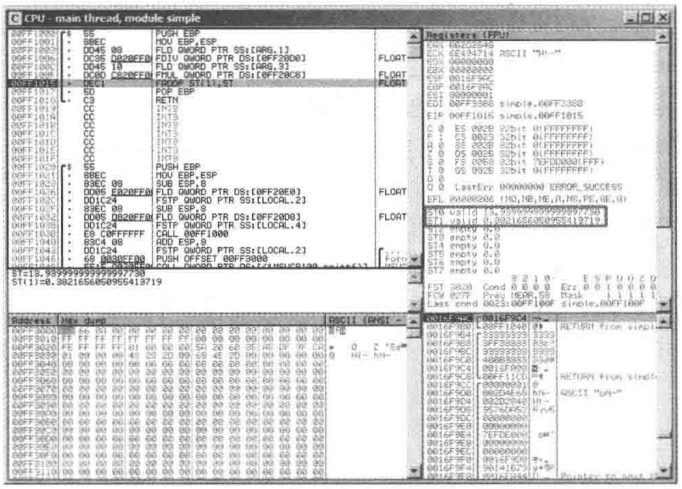


图 17.4　OllyDbg：执行 FMUL 指令

然后运行 FADDP 指令，运算求得的和会被存储在 ST(0)寄存器中。同时，指令会清空 ST(1)寄存器。如图 17.5 所示。

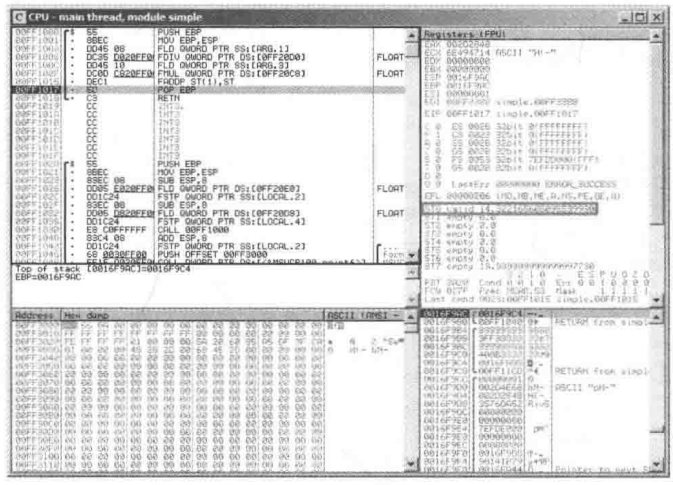


图 17.5 OllyDbg：执行 FADDP 指令

在 FPU 的运算指令结束之后，运算结果存储在 ST(0)寄存器里。main()函数稍后会从这个寄存器提取运算结果。值得注意的是 ST(7)寄存器——它的值是 13.93……这是为什么？

这不难理解。前文介绍过，FPU 的寄存器构成了自己的栈结构。因为需要用硬件直接实现数据栈，所以栈结构也比较简单。在对 FPU 进行出入栈操作的时候，如果每次都要把所有 7 个寄存器的内容转移（或者说复制）到相邻的寄存器，那么开销会非常高。实际的 FPU 只有 8 个寄存器和 1 个栈顶指针（TOP）寄存器。栈顶指针寄存器专门记录“栈顶”寄存器的寄存器编号。在 FPU 进行数据入栈（PUSH）操作时，它首先令栈顶指针寄存器指向下一个寄存器，然后在那个寄存器里存储数据。出栈（POP）指令的过程相反。但是在进行出栈操作时，FPU 不会清空原有寄存器（否则必定影响性能）。所以，在执行完程序的浮点运算指令后，FPU 寄存器的状态就如图 17.5 所示。这种现象可以说是“FADDP 指令把运算结果推送入栈，然后进行了出栈操作”，但是实际上这条指令把和存入寄存器后调整了栈顶指针寄存器的值。所以，确切地说，FPU 的寄存器构成了循环缓冲区（circular buffer）。

GCC

使用 GCC 4.4.1（启用 –O3 选项）编译上述代码，生成的程序略有不同。

指令清单 17.2 Optimizing GCC 4.4.1

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.14
                mov     ebp, esp
                fdivr   [ebp+arg_0]

; stack state now: ST(0) = result of division

                fld     ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division

                fmul    [ebp+arg_8]

; stack state now: ST(0) = result of multiplication, ST(1) = result of division
```

```
                pop     ebp
                faddp   st(1), st

; stack state now: ST(0) = result of addition

                retn
f               endp
```

第一处不同点，同时也是最显著的不同之处是：GCC 把 3.14 送入 FPU 的栈（ST(0)寄存器），用作 arg_0 的除数。

FDIVR 是 Reverse Divide 的缩写。FDIVR 指令的除数和被除数，对应 FDIV 指令的被除数和除数，即位置相反。在乘法运算中，因子的位置不影响运算结果，所以没有 FMULR 指令。

FADDP 指令从栈中 POP 出一个值进行加法运算，并用 ST(0)存储和。

17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

在 ARM 统一浮点运算标准之前，很多厂商都推出了各自的扩展指令以实现浮点运算。后来，VFP（Vector Floating Point）成为了行业的标准。

x86 平台的 FPU 有自己的栈；但是 ARM 平台里没有栈结构，只能操作寄存器。

指令清单 17.3 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
f
                VLDR        D16, =3.14
                VMOV        D17, R0, R1 ; load "a"
                VMOV        D18, R2, R3 ; load "b"
                VDIV.F64    D16, D17, D16 ; a/3.14
                VLDR        D17, =4.1
                VMUL.F64    D17, D18, D17 ; b*4.1
                vVADD.F64   D16, D17, D16 ; +
                VMOV        R0, R1, D16
                BX          LR

dbl_2C98        DCFD 3.14           ; DATA XREF: f
dbl_2CA0        DCFD 4.1            ; DATA XREF: f+10
```

上述程序出现了 D 字头的寄存器。ARM 平台有 32 个 64 位的 D 字头寄存器。这些寄存器即可用来存储（双精度）浮点数，又可用于单指令流多数据流运算/SIMD（ARM 平台下这种运算叫 NEON）。ARM 平台还有 32 个 S 字头的寄存器。S 字头寄存器用于处理单精度浮点数据。简单地说，S 字头寄存器用于存储单精度浮点，而 D 字头寄存器用于处理双精度浮点。详细规格请参见附录 B.3.3。

本例中的两个浮点都以 IEEE 754 的格式存储于内存之中。

望文生义，VLDR 和 VMOV 指令就是操作 D 字头寄存器的 LDR 和 MOV 指令。这些 V 字头的指令和 D 字头的寄存器，不仅能够处理浮点类型数据，而且可以用于 SIMD（NEON）运算。

即使涉及浮点运算，但是它还是 ARM 平台的程序，还会遵循 ARM 规范使用 R 字头寄存器传递参数。双精度浮点数据是 64 位数据，所以每传递一个双精度浮点数据就需要使用 2 个 R 字头寄存器。

“VMOV D17, R0, R1”指令从 R0 和 R1 寄存器读取 64 位数据的 2 个部分，并把最终数值存储在 D17 寄存器中。

“VMOV R0, R1, D16”与上述指令的作用相反。它把 D16 寄存器的值（64 位）分解成两个 32 位数据，并分别存储于 R0 和 R1 寄存器。

后面出现的 VDIV、VMUL、VADD 指令都是浮点运算指令，不再介绍。

使用 Xcode 生成 Thumb-2 模式的代码，会跟这段程序相同。

17.5.3 ARM: Optimizing Keil 6/2013 (Thumb mode)

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
```

```
        MOVS    R4, R3
        MOVS    R5, R0
        MOVS    R6, R1
        LDR     R2, =0x66666666;4.1
        LDR     R3, =0x40106666
        MOVS    R0, R7
        MOVS    R1, R4
        BL      __aeabi_dmul
        MOVS    R7, R0
        MOVS    R4, R1
        LDR     R2, =0x51EB851F;3.14
        LDR     R3, =0x40091EB8
        MOVS    R0, R5
        MOVS    R1, R6
        BL      __aeabi_ddiv
        MOVS    R2, R7
        MOVS    R3, R4
        BL      __aeabi_dadd
        POP     {R3-R7,PC}

; 4.1 in IEEE 754 form:
dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
; 3.14 in IEEE 754 form:
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

Keil 生成的 Thumb 模式程序不支持 NEON 运算和 FPU 浮点运算。Thumb 模式程序使用两个通用的 R 字头寄存器传递双精度浮点型数据。因为不再使用 FPU 的专用指令，所以这类程序必须调用库函数（例如 __aeabi_dmul, __aeabi_ddiv, __aeabi_dadd）“仿真”浮点运算。“仿真”意义上的模拟运算，其速度当然比不上 FPU 处理器的速度，但是聊胜于无。

在早年协作处理器还属于昂贵的奢侈品的时候，x86 平台的浮点仿真运算的库函数曾经盛行一时。

在 ARM 系统里，FPU 仿真处理叫作“软浮点/soft float”或者“armel”，而通过硬件实现的 FPU 指令叫作“硬浮点/hard float”或“armhf”。

17.5.4 ARM64: Optimizing GCC (Linaro) 4.9

这种程序十分短。

指令清单 17.4 Optimizing GCC (Linaro) 4.9

```
f:
; D0 = a, D1 = b
        ldr     d2, .LC25       ; 3.14
; D2 = 3.14
        fdiv    d0, d0, d2
; D0 = D0/D2 = a/3.14
        ldr     d2, .LC26       ; 4.1
; D2 = 4.1
        fmadd   d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
        ret

; constants in IEEE 754 format:
.LC25:
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918      ; 4.1
        .word   1074816614
```

17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9

指令清单 17.5 Non-optimizing GCC (Linaro) 4.9

```
f:
        sub     sp, sp, #16
        str     d0, [sp,8]        ; save "a" in Register Save Area
        str     d1, [sp]          ; save "b" in Register Save Area
        ldr     x1, [sp,8]
; X1 = a
        ldr     x0, .LC25
; X0 = 3.14
        fmov    d0, x1
        fmov    d1, x0
; D0 = a, D1 = 3.14
        fdiv    d0, d0, d1
; D0 = D0/D1 = a/3.14

        fmov    x1, d0
; X1 = a/3.14
        ldr     x2, [sp]
; X2 = b
        ldr     x0, .LC26
; X0 = 4.1
        fmov    d0, x2
; D0 = b
        fmov    d1, x0
; D1 = 4.1
        fmul    d0, d0, d1
; D0 = D0*D1 = b*4.1

        fmov    x0, d0
; X0 = D0 = b*4.1
        fmov    d0, x1
; D0 = a/3.14
        fmov    d1, x0
; D1 = X0 = b*4.1
        fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

        fmov    x0,d0   ;\ redundant code
        fmov    d0,x0   ;/
        add     sp, sp, 16
        ret
.LC25:
        .word   1374389535  ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918  ; 4.1
        .word   1074816614
```

在没有启用优化功能的情况下，GCC 生成的代码比较拖沓。在上述程序中，不仅出现了没有意义的数值交换指令，而且还出现了明显多余的指令（例如最后两条 FMOV 指令）。这可能是 GCC 4.9 在编译 ARM64 程序方面尚有不足。

值得注意的是，ARM64 本身就具备 64 位寄存器，而 D 字头寄存器同样是 64 位寄存器。所以编译器可以调动通用寄存器 GPR 直接存储双精度浮点数，而不必非得使用本地栈来存储这种数据。毫无疑问，在 32 位 CPU 上，编译器无法使用这种寄存器分配方案。

建议读者用这个程序进行练习，在不使用 FMADD 之类的新指令的情况下，手动优化上述函数。

17.5.6 MIPS

MIPS 平台支持多个（4 个及以下）协作处理器。第 0 个协作处理器专门用于调度其他的协作处理器，第 1 个协作处理器就是 FPU。

与 ARM 平台的情形相似，MIPS 的协作处理器不是堆栈机（stack machine），只是 32 个 32 位寄存器（$F0～$F31）。有关 FPU 各寄存器的介绍，请参见附录 C.1.2 节。在处理 64 位双精度浮点数时，必须使用一对 32 位 F 字头寄存器。

指令清单 17.6 Optimizing GCC 4.4.5 (IDA)

```
f:
; $f12-$f13=A
; $f14-$f15=B
                lui     $v0, (dword_C4 >> 16)    ; ?
; load low 32-bit part of 3.14 constant to $f0:
                lwc1    $f0, dword_BC
                or      $at, $zero    ; load delay slot, NOP
; load high 32-bit part of 3.14 constant to $f1:
                lwc1    $f1, $LC0
                lui     $v0, ($LC1 >> 16)        ; ?
; A in $f12-$f13, 3.14 constant in $f0-$f1, do division:
                div.d   $f0, $f12, $f0
; $f0-$f1=A/3.14
; load low 32-bit part of 4.1 to $f2:
                lwc1    $f2, dword_C4
                or      $at, $zero    ; load delay slot, NOP
; load high 32-bit part of 4.1 to $f3:
                lwc1    $f3, $LC1
                or      $at, $zero    ; load delay slot, NOP
; B in $f14-$f15, 4.1 constant in $f2-$f3, do multiplication:
                mul.d   $f2, $f14, $f2
; $f2-$f3=B*4.1
                jr      $ra
; sum 64-bit parts and leave result in $f0-$f1:
                add.d   $f0, $f2        ; branch delay slot, NOP

.rodata.cst8:000000B8 $LC0:             .word 0x40091EB8        # DATA XREF: f+C
.rodata.cst8:000000BC dword_BC:         .word 0x51EB851F        # DATA XREF: f+4
.rodata.cst8:000000C0 $LC1:             .word 0x40106666        # DATA XREF: f+10
.rodata.cst8:000000C4 dword_C4:         .word 0x66666666        # DATA XREF: f
```

需要介绍的指令有：

- LWC1 把一个 32 位 Word 数据传递给第一个协作处理器(Load Word to Coprocessor 1)。可见，指令中的 1 指代协作处理器的编号。成对出现的 LWC1 指令可能会被调试程序显示为伪指令 LD。
- DIV.D、MUL.D、ADD.D 指令是双精度浮点数的除法、乘法和加法运算指令。其后缀“.D”表明数据类型是 double/双精度浮点。顾名思义，后缀为“.S”的指令则是 single/单精度浮点数据的运算指令。

文中用问号“？”标出的 LUI 指令应当没有实际意义，可能是编译器生成的异常指令。如果有读者知道其中奥秘，请发 email 给我。

17.6 利用参数传递浮点型数据

本节围绕下述程序进行演示：

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

17.6.1 x86

使用 MSVC 2010 编译上述程序，可得到如下所示的指令。

指令清单 17.7 MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8  ; 为第 1 个变量分配空间
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; 为第 2 个变量分配空间
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ;单个变量的返回地址

; 栈分配了 8 个字节的空间
; 运算结果存储于 ST(0)寄存器

    fstp   QWORD PTR [esp] ;把 ST(0)的值转移到栈，供 printf()调用
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD 和 FSTP 指令是在数据段（SEGMENT）和 FPU 的栈间交换数据的指令。FLD 把内存里的数据推送入 FPU 的栈，而 FSTP 则把 FPU 栈顶的数据复制到内存中。pow()函数是指数运算函数，它从 FPU 的栈内读取两个参数进行计算，并把运算结果（x 的 y 次幂）存储在 ST(0)寄存器里。之后，printf()函数先从内存栈中读取 8 个字节的数据，再以双精度浮点的形式进行输出。

此外，这个例子里还可以直接成对使用 MOV 指令把浮点数据从内存复制到 FPU 的栈里。内存本身就把浮点数据存储为 IEEE 754 的数据格式，而 pow()函数所需的参数就是这个格式的数据，所以此处没有格式转换的必要。下一节的例子就会用到这个技巧。

17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
_main

var_C           = -0xC

                PUSH        {R7,LR}
                MOV         R7, SP
                SUB         SP, SP, #4
                VLDR        D16, =32.01
                VMOV        R0, R1, D16
                VLDR        D16, =1.54
                VMOV        R2, R3, D16
                BLX         _pow
                VMOV        D16, R0, R1
                MOV         R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
                ADD         R0, PC
                VMOV        R1, R2, D16
                BLX         _printf
                MOVS        R1, 0
                STR         R0, [SP,#0xC+var_C]
                MOV         R0, R1
```

```
                ADD         SP, SP, #4
                POP         {R7,PC}

dbl_2F90        DCFD 32.01      ; DATA XREF: _main+6
dbl_2F98        DCFD 1.54       ; DATA XREF: _main+E
```

前文介绍过，ARM 系统可以在不借助 D 字头寄存器的情况下，通过一对 R 字头寄存器传递 64 位浮点数。但是由于我们没有启用编译器的优化选项，所以它还是用 D 字头寄存器传递浮点数。

从中可以看出，R0 和 R1 寄存器给_pow 函数传递了第一个参数，R2 和 R3 寄存器给函数传递了第二个参数。函数把计算结果存储在 R0 和 R1 寄存器对。而后_pow 的运算结果再通过 D16 寄存器传递给 R1 和 R2 寄存器，以此向 printf()函数传递参数。

17.6.3 ARM + Non-optimizing Keil 6/2013 (ARM mode)

```
_main
            STMFD       SP!, {R4-R6,LR}
            LDR         R2, =0xA3D70A4      ;y
            LDR         R3, =0x3FF8A3D7
            LDR         R0, =0xAE147AE1     ;x
            LDR         R1, =0x40400147
            BL          pow
            MOV         R4, R0
            MOV         R2, R4
            MOV         R3, R1
            ADR         R0, a32_011_54Lf    ; "32.01 ^ 1.54 = %lf\n"
            BL          __2printf
            MOV         R0, #0
            LDMFD       SP!, {R4-R6,PC}

y           DCD 0xA3D70A4                   ; DATA XREF: _main+4
dword_520   DCD 0x3FF8A3D7                  ; DATA XREF: _main+8
; double x
x           DCD 0xAE147AE1                  ; DATA XREF: _main+C
dword_528   DCD 0x40400147                  ; DATA XREF: _main+10
a32_011_54Lf DCB "32.01 ^ 1.54 = %lf",0xA,0
                                            ; DATA XREF: _main+24
```

在没有启用优化功能时，编译器只使用了 R-字头寄存器对，没有使用 D-字头寄存器。

17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9

指令清单 17.8 Optimizing GCC (Linaro) 4.9

```
f:
        stp     x29, x30, [sp, -16]!
        add     x29, sp, 0
        ldr     d1, .LC1 ; load 1.54 into D1
        ldr     d0, .LC0 ; load 32.01 into D0
        bl      pow
; result of pow() in D0
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
.LC0:
; 32.01 in IEEE 754 format
        .word   -1374389535
        .word   1077936455
.LC1:
; 1.54 in IEEE 754 format
```

```
        .word   171798692
        .word   1073259479
.LC2:
        .string"32.01 ^ 1.54 = %lf\n"
```

启用优化功能之后，编译器使用 D0 和 D1 寄存器加载常量、继而传递给 pow()函数。pow()函数的运算结果再由 D0 寄存器传递给 printf()函数。因为 printf()函数不仅可以通过 X-字头寄存器获取整型数据和指针，而且还可以直接访问 D-字头寄存器获取浮点数参数，所以在传递浮点数时不需要修改或转移数据。

17.6.5 MIPS

指令清单 17.9 Optimizing GCC 4.4.5 (IDA)

```
main:

var_10      = -0x10
var_4       = -4

; function prologue:
                lui  $gp, (dword_9C >> 16)
                addiu$sp, -0x20
                la   $gp, (__gnu_local_gp & 0xFFFF)
                sw   $ra, 0x20+var_4($sp)
                sw   $gp, 0x20+var_10($sp)
                lui  $v0, (dword_A4 >> 16) ; ?
; load low 32-bit part of 32.01:
                lwc1 $f12, dword_9C
; load address of pow() function:
                lw   $t9, (pow & 0xFFFF)($gp)
; load high 32-bit part of 32.01:
                lwc1 $f13, $LC0
                lui  $v0, ($LC1 >> 16) ; ?
; load low 32-bit part of 1.54:
                lwc1 $f14, dword_A4
                or   $at, $zero ; load delay slot, NOP
; load high 32-bit part of 1.54:
                lwc1 $f15, $LC1
; call pow():
                jalr $t9
                or   $at, $zero ; branch delay slot, NOP
                lw   $gp, 0x20+var_10($sp)
; copy result from $f0 and $f1 to $a3 and $a2:
                mfc1 $a3, $f0
                lw   $t9, (printf & 0xFFFF)($gp)
                mfc1 $a2, $f1
; call printf():
                lui  $a0, ($LC2 >> 16)  # "32.01 ^ 1.54 = %lf\n"
                jalr $t9
                la   $a0, ($LC2 & 0xFFFF)  # "32.01 ^ 1.54 = %lf\n"
; function epilogue:
                lw   $ra, 0x20+var_4($sp)
; return 0:
                move $v0, $zero
                jr   $ra
                addiu$sp, 0x20

.rodata.str1.4:00000084 $LC2:      .ascii "32.01 ^ 1.54 = %lf\n"<0>

; 32.01:
.rodata.cst8:00000098 $LC0:      .word 0x40400147     # DATA XREF: main+20
.rodata.cst8:0000009C dword_9C: .word 0xAE147AE1     # DATA XREF: main
.rodata.cst8:0000009C                                # main+18
; 1.54:
.rodata.cst8:000000A0 $LC1:      .word 0x3FF8A3D7     # DATA XREF: main+24
.rodata.cst8:000000A0                                # main+30
.rodata.cst8:000000A4 dword_A4: .word 0xA3D70A4      # DATA XREF: main+14
```

这段程序的LUI指令将双精度浮点数的高16位复制到$V0寄存器。其中的（汇编宏>>16）是经IDA整理的伪代码，它的作用是对32位数据右移16位、以得到高16位数。LUI指令只能操作16位立即数。不过两个LWC1之前的LUI指令似乎没有意义，笔者给它们注释上了问号。如果哪位读者知晓其中玄机，还请联系作者本人。

MFC1是“Move From Coprocessor 1”的缩写。在MIPS系统上，第1号协作处理器是FPU。可见，这条指令首先读取协作处理器的寄存器的值，然后再把这个值复制到CPU通用寄存器GPR。不难看出，这条指令把pow()函数的运算结果复制到$A3和$A2寄存器里，然后printf()函数从这对寄存器里提取一对32位数据、再把它输出为64位双精度浮点数。

17.7 比较说明

本节围绕下述程序进行演示：

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

虽然这个函数很短，但是它的汇编代码并不那么简单。

17.7.1 x86

Non-optimizing MSVC

指令清单 17.10 Non-optimizing MSVC 2010

```
PUBLIC   _d_max
_TEXT    SEGMENT
_a$ = 8                 ; size =8
_b$ = 16                ; size = 8
_d_max     PROC
    push  ebp
    mov   ebp, esp
    fld   QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp  QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; we are here only if a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
```

```
    fld    QWORD PTR _b$[ebp]
$LN2@d_max:
    pop    ebp
    ret    0
_d_max    ENDP
```

可见，FLD 指令把汇编宏_b 加载到 ST(0)寄存器。

FCOMP 首先比较 ST（0）与_a 的值，然后根据比较的结果设置 FPU 状态字（寄存器）的 C3/C2/C0 位。FPU 的状态字寄存器是一个 16 位寄存器，用于描述 FPU 的当前状态。

在设置好相应比特位之后，FCOMP 指令还会从栈里抛出（POP）一个值。FCOM 与 FCOMP 的功能十分相似。FCOM 指令只根据数值比较的结果设置状态字，而不会再操作 FPU 的栈。

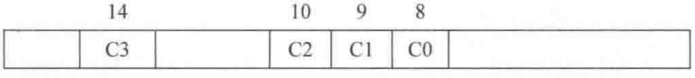
不幸的是，在 Intel P6[①]之前问世的 CPU 上，条件转移指令不能根据 C3/C2/C0 状态位进行条件判断。考虑到那时候的 FPU 在物理上尚与 CPU 分离，所以这种不足在当时大概还算不上是缺陷。

自 Intel P6 问世之后，FCOMI/FCOMIP/FUCOMI/FUCOMIP 指令不仅延续了先前各指令的功能，而且新增了设置 CPU 标志位 ZF/PF/CF 的功能。

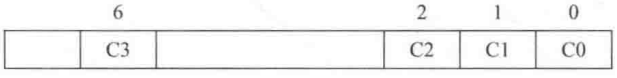
FNSTSW 指令把 FPU 状态寄存器的值复制到 AX 寄存器。C3/C2/C0 标志位对应 AX 的第 14/10/8 位。复制数值并不会改变标志位（bit）的数权（位置）。标志位会集中在 AX 寄存器的高地址位区域——即 AH 寄存器里。

- 如果 *b*>a，则 C3、C2、C0 寄存器的值会分别是 0、0、0。
- 如果 *a*>*b*，则寄存器的值会分别是 0、0、1。
- 如果 *a*=*b*，则寄存器的值会分别是 1、0、0。
- 如果出现了错误（NaN 或数据不兼容），则寄存器的值是 1、1、1。

在 FNSTSW 指令把 FPU 状态寄存器的值复制到 AX 寄存器后，AX 寄存器各个 bit 位与 C0～C3 寄存器的对应关系如下图所示。



若以 AH 寄存器的视角来看，C0～C3 与各 bit 位的对应关系则是：



“test ah, 5”指令把 ah 的值（FPU 标志位的加权求和值）和 0101（二进制的 5）做与（AND）运算，并设置标志位。影响 test 结果的只有第 0 比特位的 C0 标志位和第 2 比特位的 C2 标志位，因为其他的位都会被置零。

接下来，我们首先要介绍奇偶校验位 PF（parity flag）。

PF 标志位的作用是判定运算结果中的“1”的个数，如果“1”的个数为偶数，则 PF 的值为 1，否则其值为 0。

> 检验奇偶位通常用于判断处理过程是否出现故障，并不能判断这个数值是奇数还是偶数。FPU 有四个条件标志位（C0 到 C3）。但是，必须把标志位的值组织起来、存放在标志位寄存器中，才能进行奇偶校验位的正确性验证。FPU 标志位的用途各有不同：C0 位是进位标志位 CF，C2 是奇偶校验位 PF，C3 是零标志位 ZF。在使用 FUCOM 指令（FPU 比较指令的通称）时，如果操作数里出现了不可比较的浮点值（非数值型内容 NaN 或其他无法被指令支持的格式），则 C2 会被设为 1。

如果 C0 和 C2 都是 0 或都是 1，则设 PF 标志为 1 并触发 JP 跳转（Jump on Parity）。前面对 C3/C2/C0 的取值进行了分类讨论，C2 和 C0 的数值相同的情况分为 *b*>*a* 和 *a*=*b* 这两种情况。因为 test 指令把 ah 的值与 5 进行“与”运算，所以 C3 的值无关紧要。

在此之后的指令就很简单了。如果触发了 JP 跳转，则 FLD 指令把变量_b 的值复制到 ST(0)寄存器，

① Pentium Pro, Pentium II 等 CPU。

否则变量_a 的值将会传递给 ST（0）寄存器。

如果需要检测 C2 的状态

如果 TEST 指令遇到错误（NaN 等情形），则 C2 标志位的值会被设置为 1。不过我们的程序不检测这类错误。如果编程人员需要处理 FPU 的错误，他就不得不添加额外的错误检查指令。

使用 OllyDbg 调试本章例一（a=1.2，b=3.4）

使用 OllyDbg 打开编译好的程序，如图 17.6 所示。

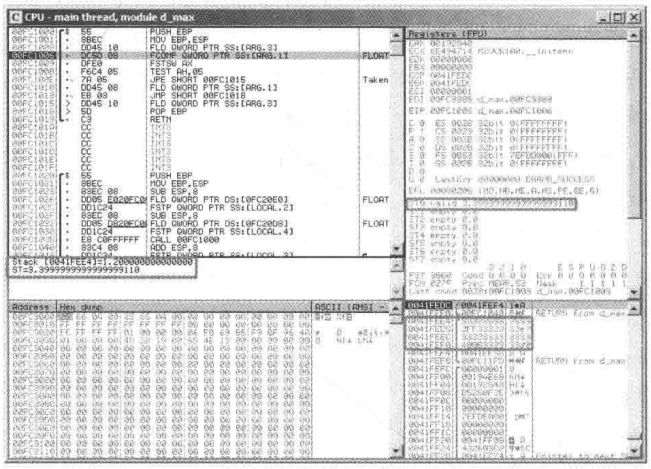


图 17.6 OllyDbg：执行第一条 FLD 指令

我们可以在数据栈中看到两对 32 位的值，它们是当前函数的两个参数：a=1.2, b=3.4。此时 ST(0)寄存器已经读取了变量 b 的值（3.4）。下一步将执行 FCOMP 指令。OllyDbg 会提示 FCOMP 的第二个参数，这个参数也在栈里。

执行 FCOMP 指令，如图 17.7 所示。

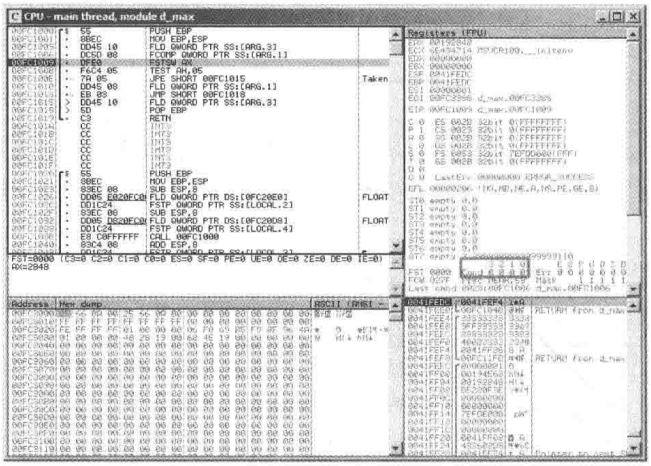


图 17.7 OllyDbg：执行 FCOMP 指令

此时 FPU 的条件标志位都是零。刚才被 POP 的数值已经转移到 ST(7)寄存器里了。本章已经在 5.1 节介绍过 FPU 的寄存器和数据栈，这里不再复述。

然后运行 FNSTSW 指令，如图 17.8 所示。

可见 AX 寄存器的值是零。确实，FPU 的所有标志位目前都是零。OllyDbg 将 FNSTSW 识别为 FSTSW 指令，这两条指令是同一条指令。

接下来运行 TEST 指令，如图 17.9 所示。

PF 标志的值为 1。这是因为 0 里面有偶数个 1，所以 PF 是 1。OllyDbg 将 JP 识别为 JPE 指令，它们是

同一个指令。在下一步里，程序会触发 JP 跳转。

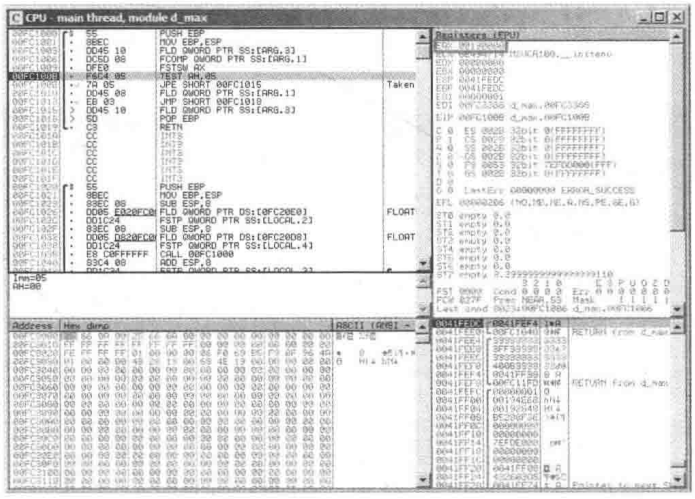


图 17.8　OllyDbg：执行 FNSTSW 指令

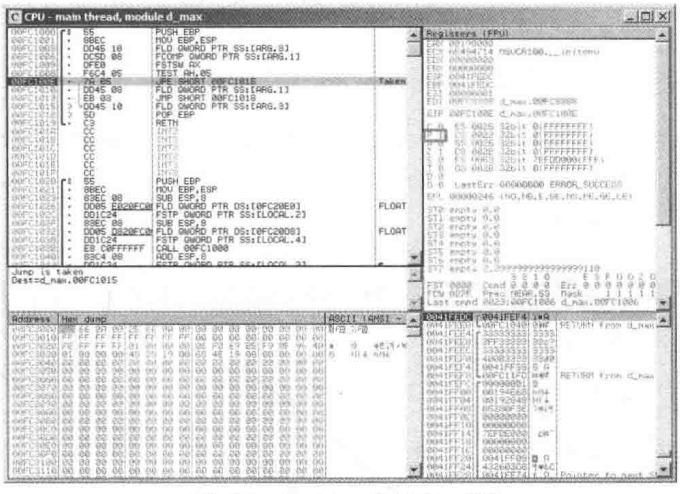


图 17.9　OllyDbg：执行 Test 指令

如图 17.10 所示，程序会触发 JPE 跳转，ST（0）将读取变量 *b* 的值 3.4。

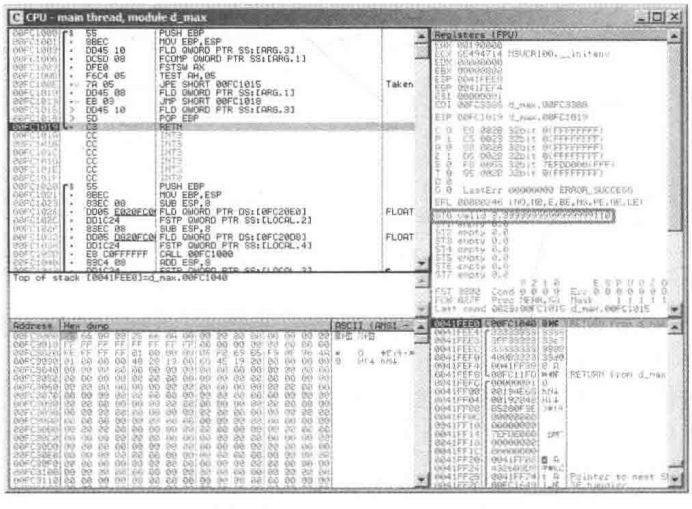


图 17.10　执行第二条 FLD 指令

此后函数结束。

调试本章例二（*a*=5.6, *b*=−4）

首先使用 OllyDbg 加载编译好的可执行程序，如图 17.11 所示。

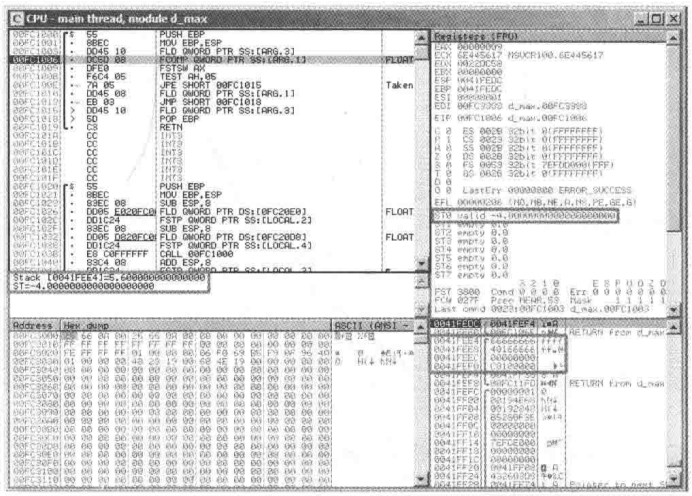


图 17.11 OllyDbg：执行第一条 FLD 指令

这个函数有两个参数，*a* 是 5.6、*b* 是−4。此刻，参数 *b* 已经加载到 ST(0)寄存器，即将执行 FCOMP 指令。OllyDbg 会在栈里显示 FCOMP 的另一个参数。

执行 FCOMP 指令，如图 17.12 所示。

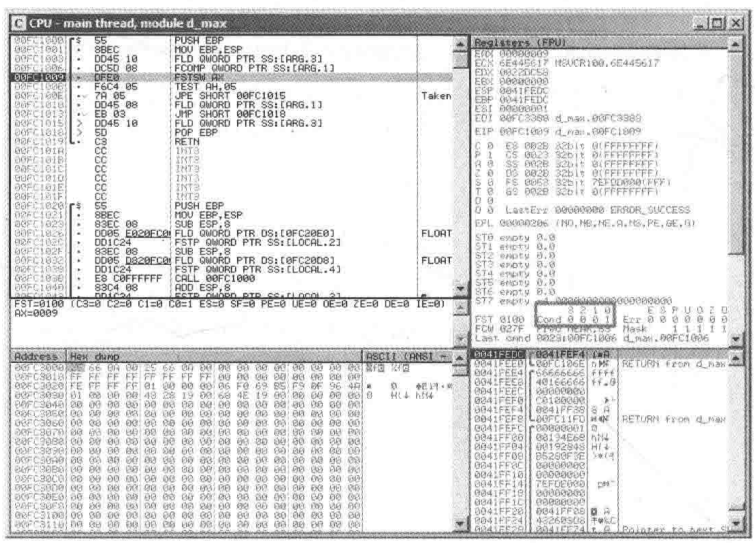


图 17.12 OllyDbg：执行 FCOMP 指令

C0 之外的 FPU 标志位都是 0。

然后执行 FNSTSW 指令，如图 17.13 所示。

此时 AX 寄存器的值是 0x100。C0 标志位是 AX 寄存器的第 8 位（从第零位开始数）。

接下来执行 TEST 指令。

如图 17.14 所示，PF 的值为 0。毕竟，把 0x100 转换为 2 进制数后，里面只有 1 个 1，1 是奇数。此后不会触发 JPE 跳转。

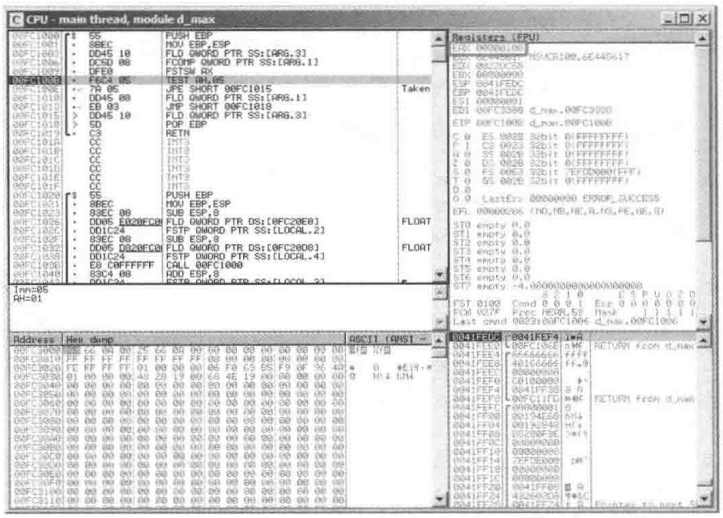


图 17.13　OllyDbg：执行 FNSTSW 指令

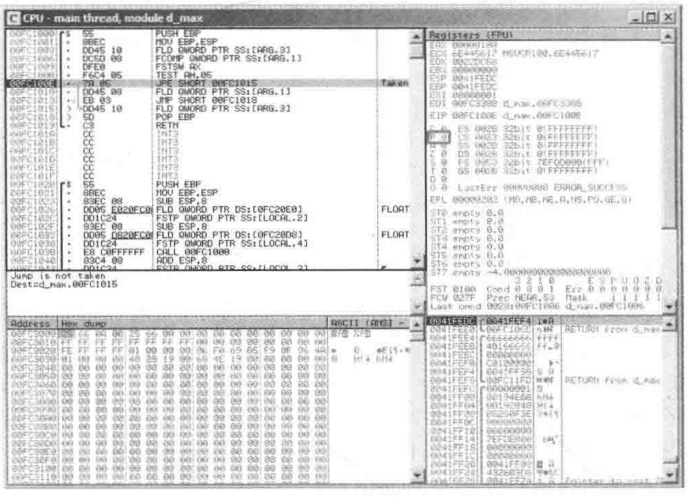


图 17.14　OllyDbg：执行 TEST 指令

因为不会触发 JPE 跳转，所以 FLD 从 *a* 里取值，把 5.6 赋值给了 ST(0)寄存器，如图 17.15 所示。

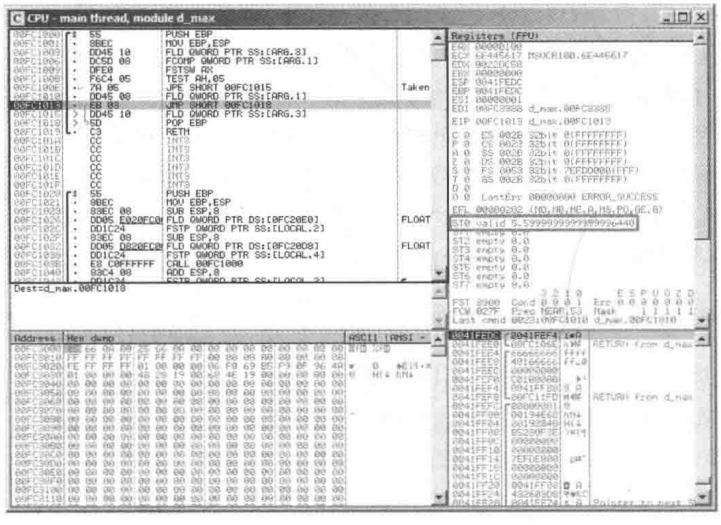


图 17.15　OllyDbg：执行第二条 FLD 指令

Optimizing MSVC 2010

指令清单 17.11 Optimizing MSVC 2010

```
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_d_max      PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]

; current stack state: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; compare _a and ST(1) = (_b)
    fnstsw  ax
    test    ah, 65 ; 00000041H
    jne     SHORT $LN5@d_max
; copy ST(0) to ST(1) and pop register,
; leave (_a) on top
    fstp    ST(1)

; current stack state: ST(0) = _a

    ret     0
$LN5@d_max:
; copy ST(0) to ST(0) and pop register,
; leave (_b) on top
    fstp    ST(0)

; current stack state: ST(0) = _b
    ret 0
_d_max ENDP
```

FCOM 指令和前面用过的 FCOMP 指令略有不同，它不操作 FPU 栈。而且本例的操作数也和前文有区别，这里它是逆序的。所以，FCOM 生成的条件标志位的涵义也与前例不同。

- 如果 *a*>*b*，则 C3、C2、C0 位的值分别为 0、0、0。
- 如果 *b*>*a*，则对应数值为 0、0、1。
- 如果 *a*=*b*，则对应数值为 1、0、0。

就是说，“test ah, 65”这条指令仅仅比较两个标志位——C3（第 6 位/bit）和 C0（第 0 位/bit）。在 *a*>*b* 的情况下，两者都应为 0：这种情况下，程序不会被触发 JNE 跳转，并会执行后面的 FSTP ST(1)指令，把 ST（0）的值复制到操作数中，然后从 FPU 栈里抛出一个值。换句话说，这条指令把 ST（0）的值（即变量_a 的值）复制到 ST（1）寄存器；此后栈顶的 2 个值都是_a。然后，相当于 POP 出一个值来，使 ST(0)寄存器的值为_a，函数随即结束。

在 *b*>*a* 或 *a*==*b* 的情况下，程序将触发条件转移指令 JNE。从 ST(0)取值、再赋值给 ST(0)寄存器，相当于 NOP 操作没有实际意义。接着它从栈里 POP 出一个值，使 ST(0)的值为先前 ST(1)的值，也就是变量_b。然后结束本函数。大概是因为 FPU 的指令集里没有 POP 并舍弃栈顶值的指令，所以才会出现这样的汇报指令。

使用 OllyDbg 调试例一：*a*=1.2/*b*=3.4 的程序

在执行两条 FLD 指令之后，情况如图 17.16 所示。

此后将执行 FCOM 指令。OllyDbg 会显示 ST(0)和 ST(1)的值。

在执行 FCOMP 指令之后，C0 为 1，其他标志全部为 0，如图 17.17 所示。

在执行 FNSTSW 指令之后，AX=0x3100，如图 17.18 所示。

然后运行 TEST，如图 17.19 所示。

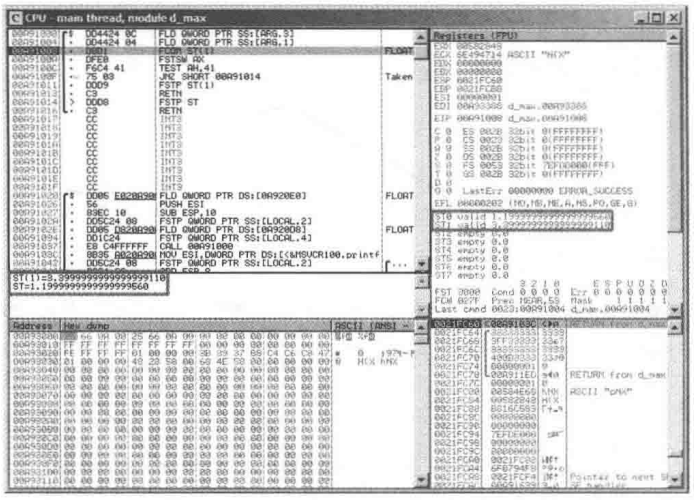


图 17.16　OllyDbg：执行过两条 FLD 指令之后的情况

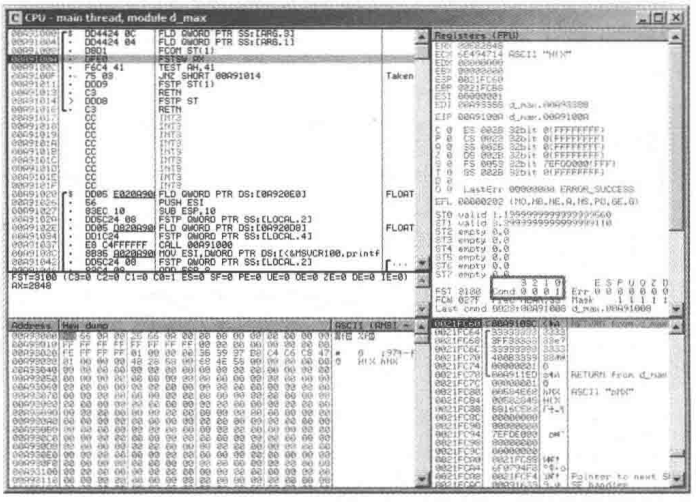


图 17.17　OllyDbg：执行 FCOM 指令

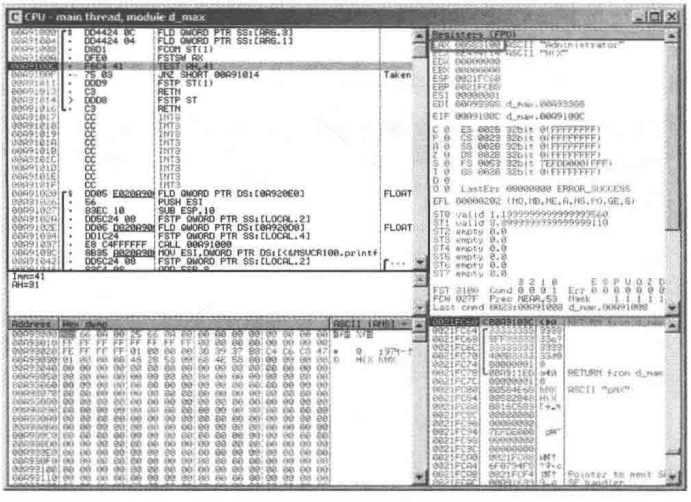


图 17.18　OllyDbg：执行 FNSTSW 指令

此刻 ZF＝0，即将触发条件转移指令。

在执行 FSTP ST（即 ST（0））的时候，FPU 把 1.2 从栈里 POP 了出来，栈顶变为 3.4，如图 17.20 所示。

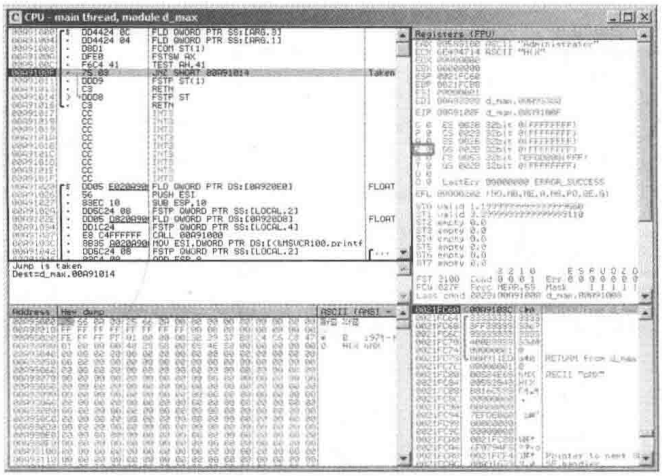


图 17.19 OllyDbg：执行 TEST 指令

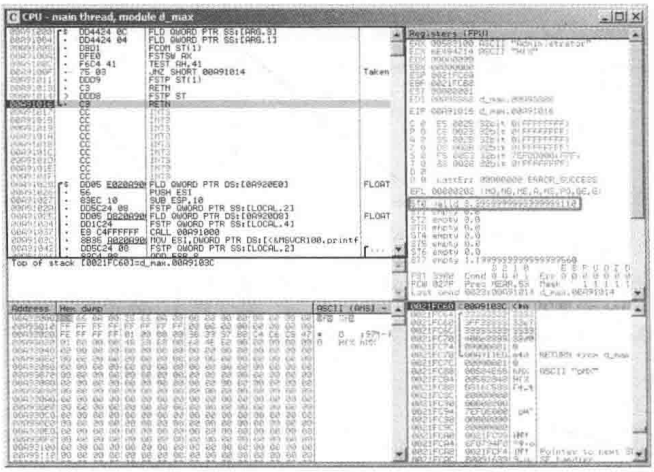


图 17.20 OllyDbg：执行 FSTP 指令

可见“FSTP ST”指令与“POP FPU 栈”指令的作用相似。

使用 OllyDbg 调试例二：a=5.6/b=-4 的程序

在执行两条 FLD 指令之后的情况如图 17.21 所示。

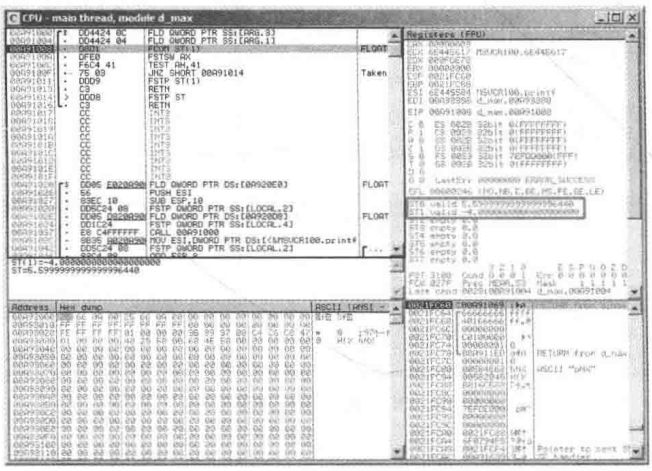


图 17.21 OllyDbg：执行两条 FLD 指令

接下来将要运行 FCOM 指令，如图 17.22 所示。

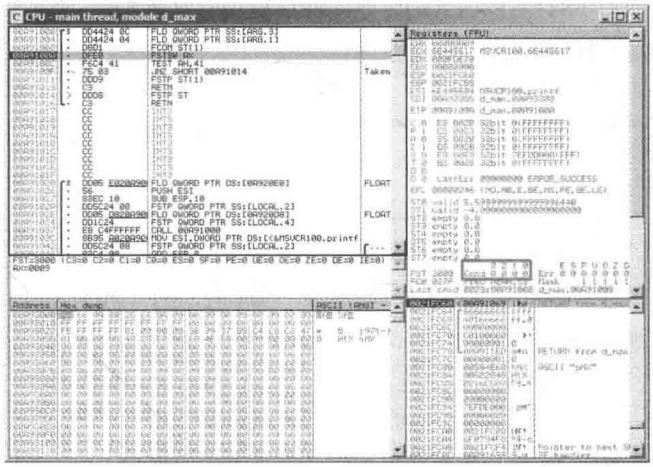


图 17.22　OllyDbg：执行 FCOM 指令

标志位寄存器都被置零。

执行过 FNSTSW 之后，AX＝0x30000，如图 17.23 所示。

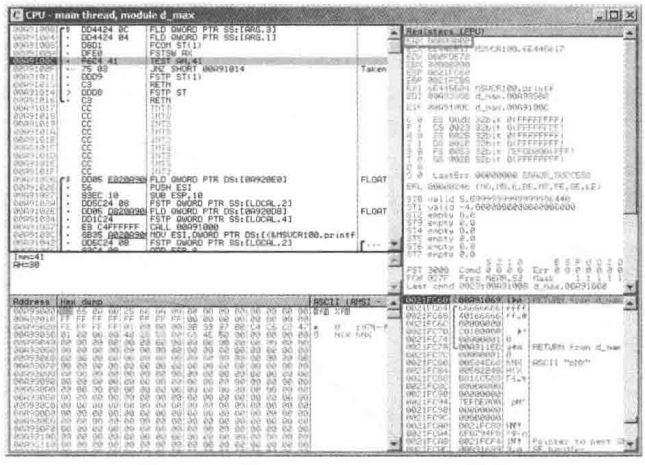


图 17.23　OllyDbg：执行 FNSTSW 指令

此后执行 TEST 指令，如图 17.24 所示。

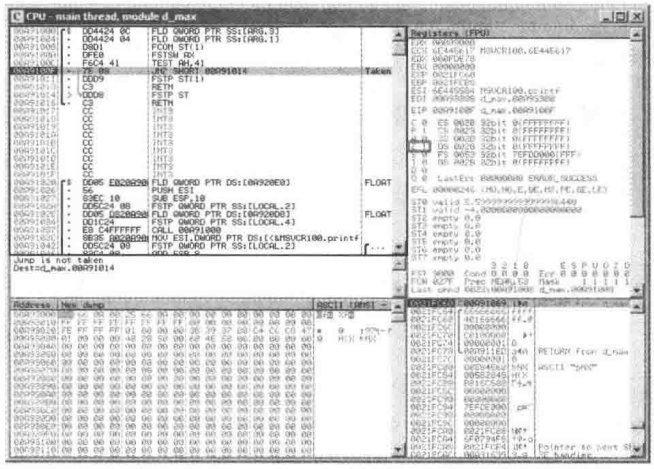


图 17.24　OllyDbg：执行 TEST 指令

在执行 TEST 置零之后，ZF=1，不会触发条件转移指令。

如图 17.25 所示，在执行 FSTP ST（1）的时候，FPU 栈顶的值是 5.6。

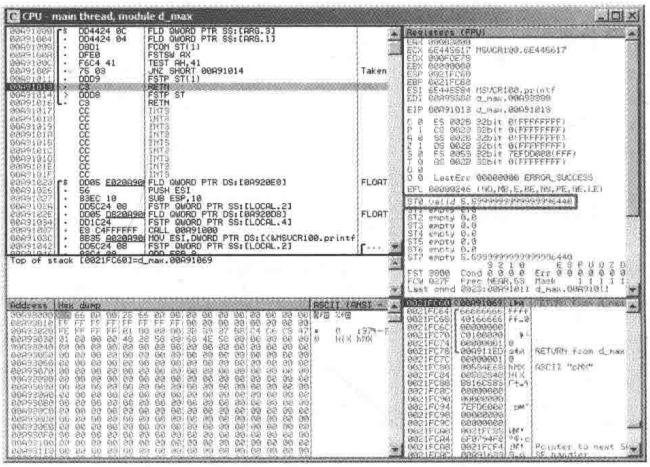


图 17.25 OllyDbg：执行 FSTP 指令

可见，FSTP ST(1)指令不会操作 FPU 栈顶的值，而会清空 ST(1)寄存器的值。

GCC 4.4.1

指令清单 17.12 GCC 4.4.1

```
d_max proc near
b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
    mov     ebp, esp
    sub     esp, 10h

; put a and b to local stack:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; load a and b to FPU stack:
    fld     [ebp+a]
    fld     [ebp+b]

; current stack state: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

    fucompp     ; compare a and b and pop two values from stack, i.e., a and b
    fnstsw  ax ; store FPU status to AX
    sahf       ; load SF, ZF, AF, PF, and CF flags state from AH
    setnbe  al ; store 1 to AL if CF=0 and ZF=0
```

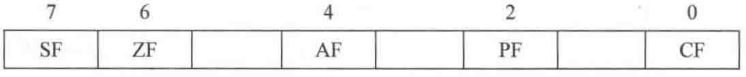
```
    test    al, al          ; AL==0 ?
    jz      short loc_8048453 ; yes
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

FUCOMPP 与 FCOM 指令的功能相似。它的全称是“Floating-Point Unsigned Compare And Pop”，所以它还能够从 FPU 栈中把两个比较的数值 POP 出来。此外，它们处理非数——“not-a-number/NaN”[①]的方式也有所不同。

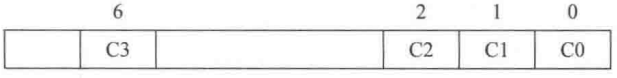
FPU 能够处理特定类型的 NaN，如无限大、除以 0 的结果等。NaN 又分为 Quiet NaN 和 Signaling NaN。对 Quiet NaN 进行操作可能不会出现问题，但是对 Signaling NaN 进行运算将会引发错误（异常处理）。

只要在 FCOM 的操作数中有 NaN，该指令就会引发异常处理机制。而 FUCOM 仅在处理 Signaling NaN（简称为 SNaN）时才会报错。

下一条指令是标志位传送指令 SAHF（Store AH into Flags）。这条指令与 FPU 无关。具体来说，它把 AH 寄存器的 8 个比特位以下列顺序传递到 CPU 的 8 位标志位里：



在前文的例子中，我们关注过 FSNSTSW 指令。它把标志位 C3/C2/C0 以下列顺序复制到 AH 寄存器的第 6、2、0 位里：



换而言之，成对使用 FNSTSW AX /SAHF 这两条指令，可以把 FPU 的 C3/C2/C0 标志位复制到 CPU 的 ZF/PF/CF 标志位。

现在回忆一下 C3/C2/C0 标志位的几种情况：

- 如果 *a>b*，则 C3/C2/C0 依次为 0、0、0。
- 如果 *a<b*，则它们依次为 0、0、1。
- 如果 *a=b*，则它们依次为 1、0、0。

即，在执行过 FUCOMPP/FNSTSW/SAHF 这组指令之后，CPU 的标志位：

- 如果 *a>b*，则 ZF=0、PF=0、CF=0。
- 如果 *a<b*，则 ZF=0、PF=0、CF=1。
- 如果 *a=b*，则 ZF=1、PF=0、CF=0。

SETNBE 可以根据 CPU 标志位和有关限定条件把 AL 寄存器设置为 0 或 1。SETNBE 只在 CF 和 ZF 寄存器都为 0 的情况下，设置 AL 为 1，其他情况下设置 AL=0。SETcc 和 Jcc[②]是孪生兄弟。不过 SETcc 的作用是按条件赋值（0 或 1），而 Jcc 的作用是按条件进行跳转。

在本例中，只有在 *a>b* 的情况下，CF 和 ZF 标志位才同时为 0 。

这种情况下 AL 将会被赋值为 1，程序不会触发 JZ 跳转，函数返回值是_a；否则函数返回值是_b。

Optimizing GCC 4.4.1

经 GCC 4.4.1（启用优化选项－O3）编译上述程序，可得到如下所示的汇编指令。

① http://en.wikipedia.org/wiki/NaN。
② cc 即条件判断指令的通称，如 AE、BE、E 等。

指令清单 17.13 Optimizing GCC 4.4.1

```
                public d_max
d_max           proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; stack state now: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; stack state now: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; compare _a and _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

; store ST(0) to ST(0) (idle operation), pop value at top of stack,
; leave _b at top
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; store _a to ST(1), pop value at top of stack, leave _a at top
                fstp st(1)

loc_804844A:
                pop     ebp
                retn
d_max           endp
```

优化编译的效果集中体现在 SAHF 指令之后的 JA 指令上。实际上，依据无符号类型数据的比较结果进行跳转的条件转移指令（JA/JAE, JB/ JBE, JE/JZ, JNA/ JNAE, JNB/ JNBE, JNE/JNZ），只检测 CF 和 ZF 标志位。

在执行 FSTSW/FNSTSW 指令后，C3/C2/C0 标志位的值将传递给 AH 寄存器。AH 与 Cx 的关系是：

	6		2	1	0
	C3		C2	C1	C0

在执行标志位传送指令 SAHF(Store AH into Flags)后，AH 寄存器的各比特位与 CPU 的 8 位标志位的对应关系就变成了：

7	6		4		2		0
SF	ZF		AF		PF		CF

对照上述两个图表可知在比较数值的一系列操作之后 C3 和 C0 标志位的值被传送到 ZF 和 CF 标志位，以供后续的条件转移指令调用。如果 CF 和 ZF 都为 0，则 JA 跳转将会被触发。

很显然，FPU 的 C3/C2/C0 状态位之所以占用寄存器的相应数权，是为了方便把 FPU 的标志位复制到 CPU 标志位上、以便进行条件判断。这多半是有意而为之。

GCC 4.8.1 –启用优化选项-O3

Intel P6 系列[①]的 FPU 指令组新增加了一组指令。这些指令是 FUCOMI（比较操作数并设置主 CPU 的标志位）和 FCMOVcc（相当于处理 FPU 寄存器的 CMOVcc 指令）。GCC 的维护人员采用了全新的指令集设计 GCC，显然他们决定不再支持 P6 以前的 CPU（也就是奔腾时代以前的 CPU）。

① Pentium Pro, Pentium-II 之后的 CPU。

另外，自 Intel P6 系列 CPU 起，Intel CPU 都整合了 FPU。这使得 FPU 直接修改、检测 CPU 标志位成为可能。经 GCC 4.8.1 优化编译后，可得到如下所示的指令。

指令清单 17.14 Optimizing GCC 4.8.1

```
fld     QWORD PTR [esp+4]       ; load "a"
fld     QWORD PTR [esp+12]      ; load "b"
; ST0=b, ST1=a
fxch    st(1)
; ST0=a, ST1=b
; compare "a" and "b"
fucomi  st, st(1)
; move ST1 (b here) to ST0 if a<=b
; leave a in ST0 otherwise
fcmovbe st, st(1)
; discard value in ST1
fstp    st(1)
ret
```

FXCH 指令把栈寄存器 ST (1)的值与栈顶 ST (0)的值进行交换，并保留栈顶指针。实际上，如果交换前两条 FLD 指令、或者把 FCMOVBE(BE 代表 below or equal)替换为 FCMOVA(A 代表 above)指令，那么就可以不用 FXCH 指令了。或许是因为编译器的优化功能尚未到位。

其后，FUCOMI 比较 ST(0)（即变量 a）和 ST(1)的值（即变量 b），并在 CPU 上设置标志位。接下来 FCOMVBE 指令检查这些标志位，并进行下述操作如果 ST(0)≤ST(1)，即 $a \leqslant b$，就把 ST（1）的值（此时是 a）复制给 ST（0）寄存器。条件不成立，就是 $a>b$ 的情况，它将保持 ST（0）的值不变。

最后一条 FSTP 指令将 ST(0) 寄存器中的值复制到目标操作数 ST(1)，然后弹出寄存器堆栈。为了弹出寄存器堆栈，处理器将 ST(0) 寄存器标记为空，并调整硬件上的堆栈指针（TOP）、便之递增 1。

使用 GDB 调试这个程序，可得到如下所示的指令。

指令清单 17.15 Optimizing GCC 4.8.1 and GDB

```
 1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
 2 dennis@ubuntuvm:~/polygon$ gdb d_max
 3 GNU gdb (GDB) 7.6.1-ubuntu
 4 Copyright (C) 2013 Free Software Foundation, Inc.
 5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
 6 This is free software: you are free to change and redistribute it.
 7 There is NO WARRANTY, to the extent permitted by law. Type "show copying"
 8 and "show warranty" for details.
 9 This GDB was configured as "i686-linux-gnu".
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>...
12 Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
13 (gdb) b d_max
14 Breakpoint 1 at 0x80484a0
15 (gdb) run
16 Starting program: /home/dennis/polygon/d_max
17
18 Breakpoint 1, 0x080484a0 in d_max ()
19 (gdb) ni
20 0x080484a4 in d_max ()
21 (gdb) disas $eip
22 Dump of assembler code for function d_max:
23    0x080484a0 <+0>:     fldl 0x4(%esp)
24 => 0x080484a4 <+4>:     fldl 0xc(%esp)
25    0x080484a8 <+8>:     fxch %st(1)
26    0x080484aa <+10>:    fucomi %st(1),%st
27    0x080484ac <+12>:    fcmovbe %st(1),%st
28    0x080484ae <+14>:    fstp %st(1)
29    0x080484b0 <+16>:    ret
30 End of assembler dump.
31 (gdb) ni
```

```
0x080484a8 in d_max ()
(gdb) info float
  R7: Valid   0x3fff9999999999999800 +1.199999999999999956
=>R6: Valid   0x4000d999999999999800 +3.399999999999999911
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a4
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) ni
0x080484aa in d_max ()
(gdb) info float
  R7: Valid   0x4000d999999999999800 +3.399999999999999911
=>R6: Valid   0x3fff9999999999999800 +1.199999999999999956
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a8
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
   0x080484a0 <+0>:     fldl 0x4(%esp)
   0x080484a4 <+4>:     fldl 0xc(%esp)
   0x080484a8 <+8>:     fxch %st(1)
=> 0x080484aa <+10>:    fucomi %st(1),%st
   0x080484ac <+12>:    fcmovbe %st(1),%st
   0x080484ae <+14>:    fstp %st(1)
   0x080484b0 <+16>:    ret
End of assembler dump.
(gdb) ni
0x080484ac in d_max ()
(gdb) info registers
eax            0x1      1
ecx            0xbffff1c4   -1073745468
edx            0x8048340    134513472
ebx            0xb7fbf000   -1208225792
esp            0xbffff10c   0xbffff10c
ebp            0xbffff128   0xbffff128
esi            0x0      0
edi            0x0      0
eip            0x80484ac    0x80484ac <d_max+12>
eflags         0x203    [ CF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
```

```
 99 es             0x7b     123
100 fs             0x0      0
101 gs             0x33     51
102 (gdb) ni
103 0x080484ae in d_max ()
104 (gdb) info float
105   R7: Valid  0x4000d999999999999800 +3.399999999999999911
106 =>R6: Valid  0x4000d999999999999800 +3.399999999999999911
107   R5: Empty  0x00000000000000000000
108   R4: Empty  0x00000000000000000000
109   R3: Empty  0x00000000000000000000
110   R2: Empty  0x00000000000000000000
111   R1: Empty  0x00000000000000000000
112   R0: Empty  0x00000000000000000000
113
114 Status Word:         0x3000
115                       TOP: 6
116 Control Word:        0x037f IM DM ZM OM UM PM
117                       PC: Extended Precision (64-bits)
118                       RC: Round to nearest
119 Tag Word:            0x0fff
120 Instruction Pointer: 0x73:0x080484ac
121 Operand Pointer:     0x7b:0xbffff118
122 Opcode:              0x0000
123 (gdb) disas $eip
124 Dump of assembler code for function d_max:
125    0x080484a0 <+0>:     fldl 0x4(%esp)
126    0x080484a4 <+4>:     fldl 0xc(%esp)
127    0x080484a8 <+8>:     fxch %st(1)
128    0x080484aa <+10>:    fucomi %st(1),%st
129    0x080484ac <+12>:    fcmovbe %st(1),%st
130 => 0x080484ae <+14>:    fstp %st(1)
131 0x080484b0 <+16>:       ret
132 End of assembler dump.
133 (gdb) ni
134 0x080484b0 in d_max ()
135 (gdb) info float
136 =>R7: Valid   0x4000d999999999999800 +3.399999999999999911
137   R6: Empty   0x4000d999999999999800
138   R5: Empty   0x00000000000000000000
139   R4: Empty   0x00000000000000000000
140   R3: Empty   0x00000000000000000000
141   R2: Empty   0x00000000000000000000
142   R1: Empty   0x00000000000000000000
143   R0: Empty   0x00000000000000000000
144
145 Status Word:         0x3800
146                       TOP: 7
147 Control Word:        0x037f IM DM ZM OM UM PM
148                       PC: Extended Precision (64-bits)
149                       RC: Round to nearest
150 Tag Word:            0x3fff
151 Instruction Pointer: 0x73:0x080484ae
152 Operand Pointer:     0x7b:0xbffff118
153 Opcode:              0x0000
154 (gdb) quit
155 A debugging session is active.
156
157         Inferior 1 [process 30194] will be killed.
158
159 Quit anyway? (y or n) y
160 dennis@ubuntuvm:~/polygon$
```

使用“ni”指令可以执行头两条 FLD 指令。

再使用第 33 行的指令检查 FPU 寄存器的状态。

前文（17.5.1 节）介绍过，FPU 寄存器属于循环缓冲区的逻辑构造，它实际上不是标准的栈结构。所以 GDB

不会把寄存器名称显示为助记符“ST(x)”，而是显示出 FPU 寄存器的内部名称，Rx。第 35 行所示的箭头表示该行的寄存器是当前的栈顶。您可从第 44 行的“Status Word/状态字”里找到栈顶寄存器的编号。本例中栈顶状态字为 6，所以栈顶是 6 号内部寄存器。

在第 54 行处，FXCH 指令交换了变量 *a* 和变量 *b* 的数值。

在执行过第 83 行的 FUCOMI 指令后，我们可在第 95 行看到 CF 为 1。

第 104 行，FCMOVBE 指令复制变量 *b* 的值。

第 136 行的 FSTP 指令会调整栈顶，也会弹出一个值。而后 TOP 的值变为 7，FPU 栈指针指向第 7 寄存器。

17.7.2 ARM

Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

指令清单 17.16 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copy "a" to D16
VMOV            R0, R1, D16
BX              LR
```

这段程序的代码很简短。函数把输入变量存储到 D17、D16 寄存器之后，使用 VCMPE 指令比较这两个变量的值。与 x86 处理器相仿，ARM 处理器也有自己的状态寄存器和标识寄存器、其协作处理器也存在相关的 FPSCR[①]。

虽然 ARM 模式的指令集存在条件转移指令，但是它的条件转移指令都不能直接访问协作处理器的状态寄存器。这个特点和 x86 系统相同。所以，ARM 平台也有专门的指令把协作处理器的 4 个标识位（N、Z、C、V）复制到通用状态寄存器的 ASPR 寄存器里，即 VMRS 指令。

VMOVGT 是 FPU 上的 MOVGT 指令，在操作数大于另一个操作数时进行赋值操作。这个指令的后缀 GT 代表“Greater Than”。

如果触发了 VMOVGT 指令，则会把 D17 里变量 *a* 的值复制到 D16 寄存器里。

否则，D16 寄存器将会保持原来的变量 *a* 的值。

倒数第二条指令 VMOV 的作用是制备返回值，它把 D16 寄存器里 64 位的值拆分为 1 对 32 的值，并分别存储于通用寄存器 R0 和 R1 里。

Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

使用 Xcode 4.6.3（开启优化选项）、以 Thumb-2 模式编译上述程序可得到如下所示的指令。

指令清单 17.17 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Thumb-2 模式的代码和 ARM 模式的程序大体相同。确实，很多 ARM 模式的指令都存在对应的依相应条件才会执行的衍生指令。

但是 Thumb 模式没有这种衍生的执行条件指令。Thumb 模式的 opcode 只有 16 位。这个空间存储不下条件判断表达式所需的那 4 位的存储空间。

① ARM 平台的 Floating-Point Status and Control Register。

而扩充后的 Thumb-2 指令集则没有上述缺陷，它们可以封装 Thumb 模式欠缺的条件判断表达式。

在 IDA 显示的汇编指令清单里，我们可以看到 Thumb-2 模式的代码里也出现 VMOVGT 指令。

此处的实际指令是 VMOV 指令，IDA 为其添加了－GT 后缀。为了直观地体现前面那条指令“IT GT”的条件判断作用，IDA 在此使用了伪指令。

IT 指令与所谓的 if－then 语句存在明确的对应关系。在 IT 指令之后的指令（最多 4 条指令），相当于在 then 语句模块里的一组条件运行指令。在本例中“IT GT”的涵义是：如果前面比较的数值，第一个值“大于/Greater Than”第二个值，则执行后续模块的 1 条指令。

我们来看下“愤怒的小鸟（iOS 版）”里的代码片段。

指令清单 17.18 Angry Birds Classic

```
...
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
BLX             _objc_msgSend ; not prefixed
...
```

“ITE”是“if-then-else”的缩写。这个指令后有两条指令：第一条就是 then 模块，第二条就是 else 模块。

我们再从“愤怒的小鸟”里找段更为复杂的代码。

指令清单 17.19 Angry Birds Classic

```
...
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
BLX             ___stack_chk_fail ; not prefixed
...
```

ITTTT 里有 4 个 T，代表 then 语句有 4 条指令。根据这个信息，IDA 给后续的 4 条指令添加了 EQ 伪后缀。

确实有 ITEEE EQ 这种形式的指令，代表“if-then-else-else-else”。在解析到这条指令后，IDA 会给其后的 5 条指令依次添加下述后缀：

```
-EQ
-NE
-NE
-NE
```

我们继续分析“愤怒的小鸟”里的其他程序片段。

指令清单 17.20 Angry Birds Classic

```
...
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
MOVS            R6, #0          ; not prefixed
CBZ             R0, loc_1E7E32  ; not prefixed
...
```

ITTELE 表示如果“小于或等于/LE(Less or Equal)”的条件成立，则执行 then 模块的 2 条指令，否则（“大于”情况下）执行 else 模块的第 3 条指令。

虽然 I-T-E 类型的指令可以有多个 T 和多个 E，但是编译器还没有聪明到按需分配所有排列组合的程序。以“愤怒的小鸟”（iOS 经典版）为例，那个时候的编译器只能分配“IT,ITE,ITT,ITTE,ITTT,ITTTT”这几种判断语句。调查 IDA 生成的汇编指令清单就可以验证这点。在生成汇编指令清单文件的时候，启用相关选项 IDA 同步输出每条

指令的4字节opcode。因为IT指令的高16位的opcode是0xBF，所以我们应当使用的Linux分析指令是：

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

另外，如果您要使用ARM的汇编语言手工编写Thumb-2模式的应用程序，那么只要您在指令后面添加相应的条件判断后缀，编译器就会自动添加相应的IT指令验证相应标志位。

Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

指令清单 17.21 Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

这段程序使用了栈结构来处理变量 *a* 和变量 *b*，所以操作略微烦琐。其他方面很好理解。

Optimizing Keil 6/2013 (Thumb mode)

使用Keil 6/2013（开启优化选项）、以Thumb模式编译上述程序可得到如下所示的指令。

指令清单 17.22 Optimizing Keil 6/2013 (Thumb mode)

```
            PUSH    {R3-R7,LR}
            MOVS    R4, R2
            MOVS    R5, R3
            MOVS    R6, R0
            MOVS    R7, R1
            BL      __aeabi_cdrcmple
            BCS     loc_1C0
            MOVS    R0, R6
            MOVS    R1, R7
            POP     {R3-R7,PC}

loc_1C0
            MOVS    R0, R4
```

```
        MOVS    R1, R5
        POP     {R3-R7,PC}
```

因为运行 Thumb 指令的硬件平台未必会有 FPU 硬件，所以 Keil 不会生成 FPU 的硬指令。因此在比较浮点数时，Keil 没有直接使用 FPU 的比较指令，而是使用了额外的库函数__aeabi_cdrcmple 进行仿真处理。

需要注意的是，本程序调用的仿真函数会在比较数值后保留 CPU 标志位，所以后面可以直接执行 BCS（B-Carry Set，大于或等于的情况下触发 B 跳转）之类的条件执行指令。

17.7.3　ARM64

Optimizing GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
        fcmpe   d0, d1
        fcsel   d0, d0, d1, gt
; now result in D0
        ret
```

ARM64 处理器的 FPU 指令集，能够不通过 FPSCR 直接设置 APSR。至少，在逻辑上 FPU 不再独立于主处理器。此处的 FCMPE 指令负责比较 D0 和 D1 寄存器中的值（即函数的第一、第二参数），并根据比较结果设置相应的 APSR 标识位（N、Z、C、V）。

FCSEL（Floating Conditional Select）首先判断条件 GT（Greater Than）是否成立，然后会选择性地复制 D0 或 D1 的值到 D0 寄存器。需要强调的是，在进行条件判断的时候，这个指令依据 APSR 寄存器里的标识、而非 FPSCR 里的标识进行判断。相比早期的 CPU 而言，ARM64 平台的这种“可直接访问 APSR”的特性算得上是一种进步。

如果条件表达式（GT）为真，那么 D0 将复制 D0 的值（即不发生值变化）。如果该条件表达式不成立，则 D0 将复制 D1 寄存器的值。

Non-optimizing GCC (Linaro) 4.9

```
d_max:
; save input arguments in "Register Save Area"
        sub     sp, sp, #16
        str     d0, [sp,8]
        str     d1, [sp]
; reload values
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        fmov    d0, x1
        fmov    d1, x0
; D0 - a, D1 - b
        fcmpe   d0, d1
        ble     .L76
; a>b; load D0 (a) into X0
        ldr     x0, [sp,8]
        b       .L74
.L76:
; a<=b; load D1 (b) into X0
        ldr     x0, [sp]
.L74:
; result in X0
        fmov    d0, x0
; result in D0
        add     sp, sp, 16
        ret
```

在关闭优化编译功能之后，编译出来的程序很庞大。首先，函数把输入参数存储于局部数据栈（Register Save Area）。接着把这些值复制到 X0/X1 寄存器，再把它们复制到 D0/D1 中、使用 FCMPE 进

行比较。虽然这种程序的效率不高，不过在非优化模式下，编译器就本来是这样生成程序的。FCMPE在进行比较之后设置相应的 APSR 标识。不过，我们可以看出编译器没有考虑使用更为方便的 FCSEL指令。所以它使用了较为古老的方式进行编译：它在此次分配了 BLE（Branch if Less than or Equal）指令。在 $a>b$ 的情况下，变量 a 的值将传递到 X0 寄存器；否则，变量 b 的值将传递给 X0 寄存器。最终，X0 的值传递给 D0 寄存器，成为函数的返回值。

练习题

请在不使用新指令（包括 FCSEL 指令）的前提下，优化本例的程序。

Optimizing GCC (Linaro) 4.9—float

在把参数的数据类型从 double 替换为 float 之后，再进行编译：

```
float f_max (float a, float b)
{
        if (a>b)
                return a;

        return b;
};

f_max:
; S0 - a, S1 - b
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
; now result in S0
        ret
```

程序使用了 S-字头的寄存器，而不再使用 D-字头寄存器。这是因为 S-字头寄存器的 32 位（即 64 位 D字头寄存器的低 32 位）已经满足单精度浮点的存储需要了。

17.7.4 MIPS

即使是当今最受欢迎的 MIPS 处理器，其协作处理器也只能设置一个条件标识位。供 CPU 访问。早期的 MIPS 处理器只有一个标识条件位（即 FCC0），现已逐步扩展到了 8 个（即 FCC7～FCC0）。这些条件标识位位于浮点条件码寄存器（FCCR）。

指令清单 17.23 Optimizing GCC 4.4.5 (IDA)

```
d_max:
; set FPU condition bit if $f14<$f12 (b<a):
                c.lt.d    $f14, $f12
                or        $at, $zero ; NOP
; jump to locret_14 if condition bit is set
                bc1t      locret_14
; this instruction is always executed (set return value to "a"):
                mov.d     $f0, $f12  ; branch delay slot
; this instruction is executed only if branch was not taken (i.e., if b>=a)
; set return value to "b":
                mov.d     $f0, $f14
locret_14:
                jr        $ra
                or        $at, $zero ; branch delay slot, NOP
```

“C.LT.D”是比较两个数值的指令。在它的名称中，“LT”表示条件为“Less Than”，“D”表示其数据类型为 double。它将根据比较的结果设置、或清除 FCC0 条件位。

“BC1T”检测 FCC0 位，如果该标识位被置位（值为 1）则进行跳转。“T”是 True 的缩写，表示该指令的运行条件是“标识位被置位（True）”。实际上确实存在对应的 BC1F 指令，在判定条件为 False

的时候进行跳转。

无论上述条件转移指令是否发生跳转，它都决定了$F0 的最终取值。

17.8 栈、计算器及逆波兰表示法

现在，我们可以理解部分旧式计算器采取逆波兰表示法[①]的道理了。例如，在计算“12+34”时，这种计算器要依次按下“12”、“34”和加号。这种计算器采用了堆栈机器（stack machine）的构造。逆波兰记法不需要括号来标识操作符的优先级。所以，在计算复杂表达式时，这种构造计算器的操作十分简单。

17.9 x64

有关 x86-64 系统处理浮点数的方法，请参见本书第 27 章。

17.10 练习题

17.10.1 题目 1

请去除 17.7.1 节所示例子中的 FXCH 指令，进行改写并进行测试。

17.10.2 题目 2

请描述下述代码的功能。

指令清单 17.24 Optimizing MSVC 2010

```
__real@4014000000000000 DQ 04014000000000000r  ; 5

_a1$ = 8        ;size=8
_a2$ = 16       ;size=8
_a3$ = 24       ;size=8
_a4$ = 32       ;size=8
_a5$ = 40       ;size=8

_f      PROC
        fld     QWORD PTR _a1$[esp-4]
        fadd    QWORD PTR _a2$[esp-4]
        fadd    QWORD PTR _a3$[esp-4]
        fadd    QWORD PTR _a4$[esp-4]
        fadd    QWORD PTR _a5$[esp-4]
        fdiv    QWORD PTR __real@4014000000000000
        ret     0
_f      ENDP
```

指令清单 17.25 Non-optimizing Keil 6/2013 (Thumb mode/compiled for Cortex-R4F CPU)

```
f PROC
        VADD.F64 d0,d0,d1
        VMOV.F64 d1,#5.00000000
        VADD.F64 d0,d0,d2
        VADD.F64 d0,d0,d3
        VADD.F64 d2,d0,d4
        VDIV.F64 d0,d2,d1
```

① Reverse Polish notation，请参见 https://en.wikipedia.org/wiki/Reverse_Polish_notation。

```
        BX      lr
        ENDP
```

指令清单 17.26 Optimizing GCC 4.9 (ARM64)

```
f:
        fadd    d0, d0, d1
        fmov    d1, 5.0e+0
        fadd    d2, d0, d2
        fadd    d3, d2, d3
        fadd    d0, d3, d4
        fdiv    d0, d0, d1
        ret
```

指令清单 17.27 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:

arg_10          = 0x10
arg_14          = 0x14
arg_18          = 0x18
arg_1C          = 0x1C
arg_20          = 0x20
arg_24          = 0x24

                lwc1   $f0, arg_14($sp)
                add.d  $f2, $f12, $f14
                lwc1   $f1, arg_10($sp)
                lui    $v0, ($LC0 >> 16)
                add.d  $f0, $f2, $f0
                lwc1   $f2, arg_1C($sp)
                or     $at, $zero
                lwc1   $f3, arg_18($sp)
                or     $at, $zero
                add.d  $f0, $f2
                lwc1   $f2, arg_24($sp)
                or     $at, $zero
                lwc1   $f3, arg_20($sp)
                or     $at, $zero
                add.d  $f0, $f2
                lwc1   $f2, dword_6C
                or     $at, $zero
                lwc1   $f3, $LC0
                jr     $ra
                div.d  $f0, $f2

$LC0:           .word 0x40140000  # DATA XREF: f+C
                                  # f+44
dword_6C:       .word 0           # DATA XREF: f+3C
```

第 18 章 数 组

在内存中，数组就是按次序排列的、相同数据类型的一组数据。

18.1 简介

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

18.1.1 x86

MSVC

使用 MSVC 2008 编译上述程序可得如下所示的指令。

指令清单 18.1 MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84               ; size = 4
_a$ = -80               ; size = 80
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84     ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN6@main
$LN5@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax,  1
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
```

```
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20    ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _a$[ebp+ecx*4]
    push    edx
    mov     eax, DWORD PTR _i$[ebp]
    push    eax
    push    OFFSET $SG2463
    call    _printf
    add     esp, 12       ; 0000000cH
    jmp     SHORT $LN2@main
$LN1@main:
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

这里除去两个循环之外没有非常特别之处。第一个循环填充数据，第二个循环打印数据。“shl ecx, 1”指令所做的运算是 ecx=ecx*2，更详细的介绍请参见 16.2.1 节。

程序为数组申请了 80 字节的栈空间，以存储 20 个 4 字节元素。

现在使用 OllyDbg 打开这个执行程序。

如图 18.1 所示，数组中的每个元素都是 32 位的 int 型数据，数组每个元素的值都是其索引值的 2 倍。

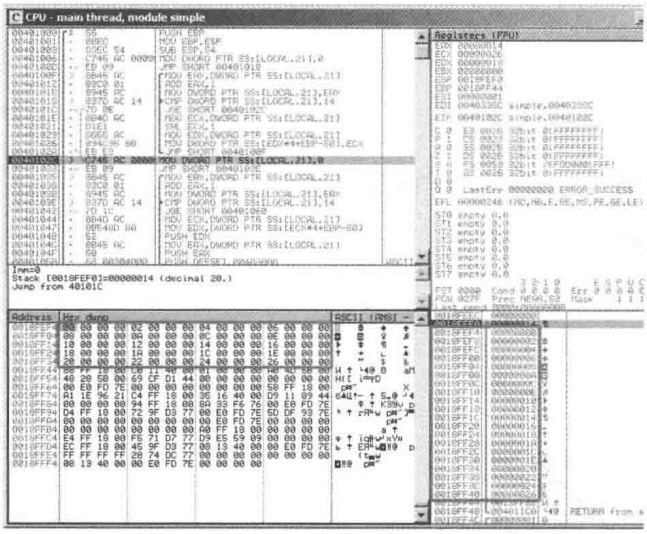


图 18.1 OllyDbg：填充数组

因为全部数组都存储于栈中，所以我们可以在内存数据窗口里看到整个数组。

GCC

若使用 GCC 4.4.1 编译上述程序，可得到如下所示的指令。

指令清单 18.2 GCC 4.4.1

```
                public main
main            proc near               ; DATA XREF: _start+17

var_70          = dword ptr -70h
```

```
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1      ;  i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

实际上变量 *a* 的数据类型是整型指针。严格地说，在把数组传递给函数的时候，传递的数据就是指向第一个元素的指针，我们再根据这个指针就可以轻松地计算出数组每个元素的地址（即指针）。如果使用a[idx]的形式表示数组元素，其中 idx 是数组元素在数组里的排列序号（即索引号），那么就可以通过数组第一个元素的地址、索引号和数据容量求得各个元素的地址。

举个典型的例子：字符串常量“string”是字符型数组，它的每个字符元素都是 const char*型数据。使用索引号之后，我们就可以使用“string”[i]的形式描述字符串中的第 *i* 个字符——这正是 C/C++表达式的表示方法！

18.1.2 ARM

Non-optimizing Keil 6/2013 (ARM mode)

```
                EXPORT _main
_main
                STMFD   SP!, {R4,LR}
                SUB     SP, SP, #0x50       ;分配 20 个 int 的存储空间

; first loop
```

```
                MOV     R4, #0              ; i
                B       loc_4A0

loc_494
                MOV     R0, R4,LSL#1        ; R0=R4*2
                STR     R0, [SP,R4,LSL#2]   ; store R0 to SP+R4<<2 (same as SP+R4*4)
                ADD     R4, R4, #1          ; i=i+1

loc_4A0
                CMP     R4, #20             ; i<20?
                BLT     loc_494             ;yes? loop again

; second loop

                MOV     R4, #0              ; i
                B       loc_4C4
loc_4B0
                LDR R2, [SP,R4,LSL#2]       ; (printf 的第 2 个参数) R2=*(SP+R4<<4) (等同于* (SP+R4*4))
                MOV     R1, R4              ; (printf 的第 1 个参数) R1=i
                ADR     R0, aADD            ; "a[%d]=%d\n"
                BL      __2printf
                ADD     R4, R4, #1          ; i=i+1

loc_4C4
                CMP     R4, #20             ; i<20?
                BLT     loc_4B0             ; yes, run loop body again
                MOV     R0, #0              ; value to return
                ADD     SP, SP, #0x50       ; 释放 20 个 int 的存储空间
                LDMFD   SP!, {R4,PC}
```

单个整型数据占 32 位（4 字节）存储空间。以此类推，要存储 20 个 int 型数据就需要 80（0x50）个字节的空间。函数尾声的“SUB SP, SP, 0x50”指令释放的空间就是存储数组所用的空间。

在两个循环体内，循环控制变量 *i* 都使用 R4 寄存器。

在填充数组元素的时候，编译器用左移 1 位的运算“LSL#1”实现乘法运算“*i*×2”，形成了“MOV R0, R4,LSL#1”指令。

“STR R0，[SP, R4, LSL#2]”实现了把 R0 寄存器的值写入数组。计算数组元素的原理：因为 SP 可代表数组的起始地址、R4 代表变量 *i*，所以可通过 *i* 左移 2 位求得 *i*×4 的相对偏移地址，再计算地址“SP+R4×4”即可推导出数组元素 *a*[*i*]的指针地址。

在第二个循环里，“LDR R2, [SP,R4,LSL#2]”指令读取 *a*[*i*]的值。

Optimizing Keil 6/2013 (Thumb mode)

```
_main
        PUSH    {R4,R5,LR}
; allocate place for 20 int variables + one more variable
        SUB     SP, SP, #0x54

;第一个循环
        MOV     R0, #0          ;i
        MOV     R5, SP          ; pointer to first array element

loc_1CE
        LSLS    R1, R0, #1      ; R1=i<<1 (same as i*2)
        LSLS    R2, R0, #2      ; R2=i<<2 (same as i*4)
        ADDS    R0, R0, #1      ; i=i+1
        CMP     R0, #20         ; i<20?
        STR     R1, [R5,R2]     ; store R1 to *(R5+R2) (same R5+i*4)
        BLT     loc_1CE         ; yes, i<20, run loop body again

;第二个循环
        MOVS    R4, #0          ; i=0
```

```
loc_1DC
        LSLS    R0, R4, #2      ; R0=i<<2 (same as i*4)
        LDR     R2, [R5,R0]     ; load from *(R5+R0) (same as R5+i*4)
        MOVS    R1, R4
        ADR     R0, aADD        ; "a[%d]=%d\n"
        BL      __2printf
        ADDS    R4, R4, #1      ; i=i+1
        CMP     R4, #20         ; i<20?
        BLT     loc_1DC         ; yes, i<20, run loop body again
        MOVS    R0, #0          ; value to return
; deallocate Chunk, allocated for 20 int variables + one more variable
        ADD     SP, SP, #0x54
        POP     {R4,R5,PC}
```

Thumb 模式的程序和 ARM 模式的程序十分相近。在 Thumb 模式里，位移操作有特定的对应指令（如 LSLS）。上述程序借助 Thumb 模式的指令计算每个数组元素的地址。

另外，本例分配的栈空间比实际需求大了一些。这个栈里的最后 4 个字节不会被占用。

Non-optimizing GCC 4.9.1 (ARM64)

指令清单 18.3 Non-optimizing GCC 4.9.1 (ARM64)

```
.LC0:
        .string "a[%d]=%d\n"
main:
; save FP and LR in stack frame:
        stp     x29, x30, [sp, -112]!
; set stack frame (FP=SP)
        add     x29, sp, 0
; setting initial counter variable at 0 (WZR is the register always holding zero):
        str     wzr, [x29,108]
; jump to loop condition checking code:
        b       .L2
.L3:
; load value of "i" variable:
        ldr     w0, [x29,108]
; multiplicate it by 2:
        lsl     w2, w0, 1
; find a place of an array in local stack:
        add     x0, x29, 24
; load 32-bit integer from local stack and sign extend it to 64-bit one:
        ldrsw   x1, [x29,108]
; calculate address of element (X0+X1<<2=array address+i*4) ;  and store W2 (i*2) there:
        str     w2, [x0,x1,lsl 2]
; increment counter (i):
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L2:
; check if we finished:
        ldr     w0, [x29,108]
        cmp     w0, 19
; jump to L3 (loop body begin) if not:
        ble     .L3
; second part of the function begins here.
; setting initial counter variable at 0.
; by the way, the same place in the local stack was used for counter,
; because the same local variable (i) is being used as counter.
        str     wzr, [x29,108]
        b       .L4
.L5:
; calculate array address:
        add     x0, x29, 24
; load "i" value:
        ldrsw   x1, [x29,108]
```

```
; load value from the array at the address (X0+X1<<2 = address of array + i*4)
        ldr     w2, [x0,x1,lsl 2]
; load address of the "a[%d]=%d\n" string:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; load "i" variable to W1 and pass it to printf() as second argument:
        ldr     w1, [x29,108]
; W2 still contains the value of array element which was just loaded.
; call printf():
        bl      printf
; increment "i" variable:
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L4:
; are we finished?
        ldr     w0, [x29,108]
        cmp     w0, 19
; jump to the loop body begin if not:
        ble     .L5
; return 0
        mov     w0, 0
; restore FP and LR:
        ldp     x29, x30, [sp], 112
        ret
```

18.1.3 MIPS

面向 MIPS 的编译器会大量使用 S-字头寄存器。因为 S-字头寄存器不属于临时寄存器，所以函数在序言和尾声部分对这些寄存器的值进行了备份和还原。

指令清单 18.4 Optimizing GCC 4.4.5 (IDA)

```
main:

var_70      = -0x70
var_68      = -0x68
var_14      = -0x14
var_10      = -0x10
var_C       = -0xC
var_8       =-8
var_4       =-4
; function prologue:
                lui         $gp, (__gnu_local_gp >> 16)
                addiu       $sp, -0x80
                la          $gp, (__gnu_local_gp & 0xFFFF)
                sw          $ra, 0x80+var_4($sp)
                sw          $s3, 0x80+var_8($sp)
                sw          $s2, 0x80+var_C($sp)
                sw          $s1, 0x80+var_10($sp)
                sw          $s0, 0x80+var_14($sp)
                sw          $gp, 0x80+var_70($sp)
                addiu       $s1, $sp, 0x80+var_68
                move        $v1, $s1
                move        $v0, $zero
; that value will be used as a loop terminator.
; it was precalculated by GCC compiler at compile stage:
                li          $a0, 0x28  # '('
loc_34:                                             # CODE XREF: main+3C
; store value into memory:
                sw          $v0, 0($v1)
; increase value to be stored by 2 at each iteration:
                addiu       $v0, 2
; loop terminator reached?
```

```
                bne         $v0, $a0, loc_34
; add 4 to address anyway:
                addiu       $v1, 4
; array filling loop is ended
; second loop begin
                la          $s3, $LC0       # "a[%d]=%d\n"
; "i" variable will reside in $s0:
                move        $s0, $zero
                li          $s2, 0x14
loc_54:                                     # CODE XREF: main+70
; call printf():
                lw          $t9, (printf & 0xFFFF)($gp)
                lw          $a2, 0($s1)
                move        $a1, $s0
                move        $a0, $s3
                jalr        $t9
; increment "i":
                addiu       $s0, 1
                lw          $gp, 0x80+var_70($sp)
; jump to loop body if end is not reached:
                bne         $s0, $s2, loc_54
; move memory pointer to the next 32-bit word:
                addiu       $s1, 4
; function epilogue
                lw          $ra, 0x80+var_4($sp)
                move        $v0, $zero
                lw          $s3, 0x80+var_8($sp)
                lw          $s2, 0x80+var_C($sp)
                lw          $s1, 0x80+var_10($sp)
                lw          $s0, 0x80+var_14($sp)
                jr          $ra
                addiu       $sp, 0x80

$LC0:           .ascii "a[%d]=%d\n"<0>  # DATA XREF: main+44
```

编译器对第一个循环使用了代入的技术对变量之进行了等效处理。在第一个循环的循环体中，数组元素的值（$V0 的值）是控制变量 *i* 的 2 倍，即 *i*×2，所以在每次迭代后控制变量的增量都是 2。另外，数组元素的地址增量为 4，编译器单独分配了$V1 寄存器给这个指针使用，也令其增量为 4。

第二个循环体根据数组索引值 *i*、通过 printf()函数依次输出数组元素。编译器首先使用$S0 寄存器存储索引值，$S0 在每次迭代中的增量为 1。与前一个循环体相似，它单独使用$S1 寄存器存储内存地址，并使其在迭代间的增量为 4。

这便是本书第 39 章中会提到的循环优化技术。通过这种技术，编译器可尽量避免效率较低的乘法运算。

18.2 缓冲区溢出

18.2.1 读取数组边界以外的内容

综上所述，编译器借助索引 index、以 array[index]的形式表示数组。若仔细审查二进制程序的代码，那么您可能会发现程序并没有对数组进行边界检查、没有判断索引是否在 20 以内。那么，如果程序访问数组边界以外的数据，又会发生什么情况？C/C++编译器确实不会进行边界检查，这也是它备受争议之处。

编译、并运行下面的程序，会发现整个过程中不会遇到错误提示。

```
#include <stdio.h>

int main()
{
```

```
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[20]=%d\n", a[20]);

        return 0;
};
```

使用 MSVC 2008 编译后可得到如下所示的指令。

指令清单 18.5 Non-optimizing MSVC 2008

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main:
    mov    eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

其执行结果如图 18.2 所示。

图 18.2 OllyDbg：运算结果

这是地址接近栈内数组的数据的值。它的地址距离数组的起始地址有 80 字节。

让我们通过 OllyDbg 看看这个数从哪里来。如图 18.3 所示，使用 OllyDbg 加载这个程序，找到数组最后一个值之后的数据。

这是什么的值？根据栈结构来判断，这个值是先前保存的 EBP 寄存器的值。我们继续跟踪这个程序，如图 18.4 所示，看看它的提取过程。

到底有没有什么办法控制数组的访问边界问题呢？如果需要控制数组边界，就要修改编译器，强制其生成的程序检查索引 index 是否在边界之内。某些高级编程语言，如 Python 和 Java，就会进行边界检查。但是这种控制会严重影响程序的性能。

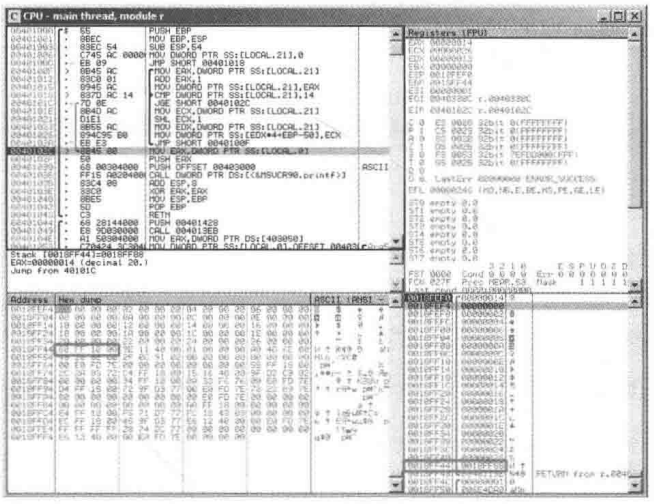


图 18.3　OllyDbg：读取数组中的第 20 个元素、并使用 printf()输出

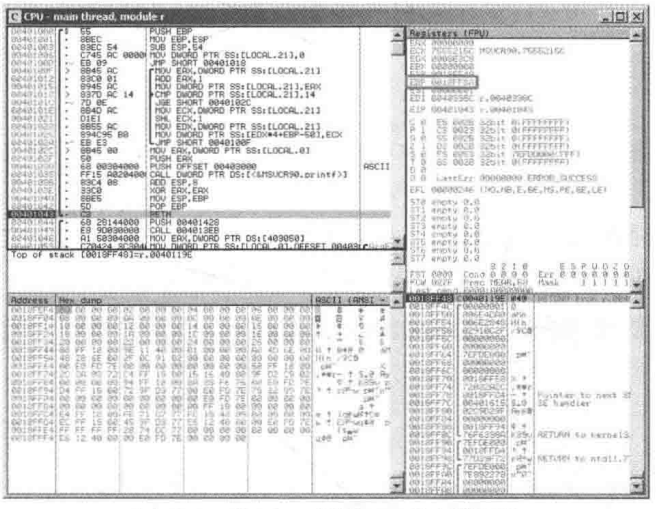


图 18.4　OllyDbg：还原 EBP 寄存器的值

18.2.2　向数组边界之外的地址赋值

既然我们可以“非法地”利用栈来读取数组边界以外的数值，那么我们是否也可以向数组边界以外的地址里写入数据呢？

可通过下面的程序来进行验证。

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

MSVC

使用 MSVC 编译后可得到如下所示的指令。

指令清单 18.6 Non-optimizing MSVC 2008

```
_TEXT   SEGMENT
_i$ = -84                               ; size = 4
_a$ = -80                               ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30          ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]         ; 这条指令用处不大
 mov    DWORD PTR _a$[ebp+ecx*4], edx; 因为也可以用 ECX 取代 edx
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

运行这个可执行程序的时候，程序崩溃了。崩溃并没什么希奇的，但我们要研究一下它是在哪里崩溃的。使用 OllyDbg 加载整个程序，等到写满 30 个元素，如图 18.5 所示。

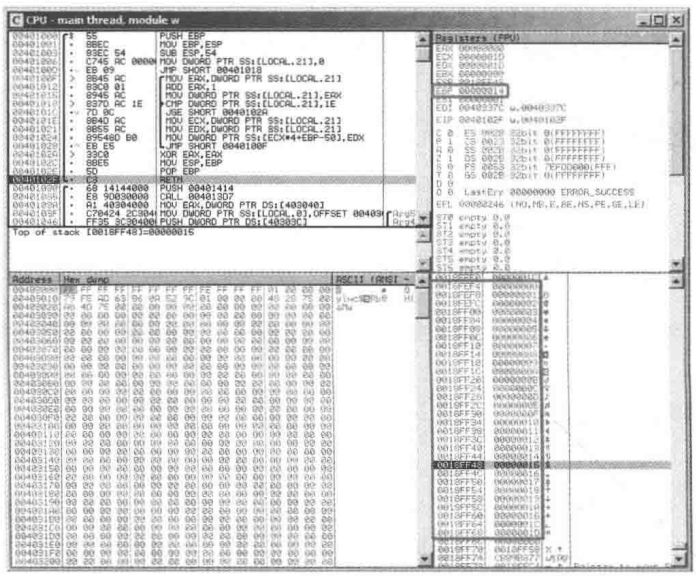


图 18.5 OllyDbg：恢复 EBP 的值之后

然后逐步执行程序，等待函数结束，如图 18.6 所示。

现在，我们再来分析寄存器状态。

现在 EIP 的值（PC）是 0x15。这不是程序的合法地址——至少对于 win32 系统来说这个值有问题！程序在此处发生异常。同样有趣的是，这时 EBP 的值是 0x14，ECX 和 EDX 的值是 0x1D。

我们一起回顾一下栈结构。

待程序进入 main()函数之时，程序使用栈来保存 EBP 寄存器的值。然后程序分配了 84 个字节，用于

存储数组和变量 *i*。计算方法是“(20+1)*sizeof(int)”。栈顶指针 ESP 现在指向栈内的变量_i，在执行 PUSH 入栈操作之后它会指向_i 的下一个地址。

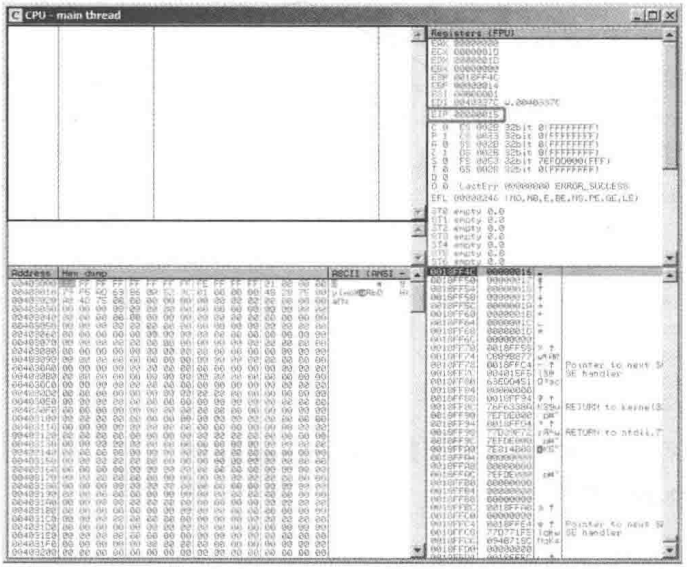


图 18.6　OllyDbg：对 EIP 进行出栈操作，但是 OllyDbg 找不到 0x15 处的程序代码

即，main()函数的栈结构如下：

ESP	*i* 所占用的 4 字节
ESP+4	*a*[20]占用的 80 字节
ESP+84	保存过的 EBP
ESP+88	返回地址

赋值给 *a*[19]的时候，数组 *a*[]已经被全部赋值。

赋值给 *a*[20]，实际修改的是栈里保存的 EBP。

观察程序崩溃时的寄存器状态。本例将数字 20（0x14）赋值给 *a*[19]。在函数退出之前，CPU 会通过栈恢复 EBP 的初始值。本例它会收到的值是 20（0x14）最后运行 RET 指令，相当于执行 POP EIP 指令。

RET 指令将程序的控制权传递给栈里的返回地址（该地址为 CRT 内部调用 main()的地址），不过这个 RA 的值被改为 21（0x15）。CPU 会寻找 0x15 处的代码，以继续执行程序。但是那处地址里没有可执行代码，所以程序就崩溃了。

恭喜！您已经了解了缓冲区溢出[①]的精妙之处。

设想一下：用字符串替代 int 数组，刻意构造个超长字符串，把字符串传递给程序；因为函数不会检查字符串长度，会直接赋值给较短的缓冲区，您就可以强制这个程序跳转到其他程序的地址。纸上谈兵固然简单，但是无论实际情况再怎么复杂，其原理都是一样的。

GCC

使用 GCC 4.4.1 编译上述程序，可得到：

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4
```

① http://en.wikipedia.org/wiki/Stack_buffer_overflow。

```
                push    ebp
                mov     ebp, esp
                sub     esp, 60h ; 96
                mov     [ebp+i], 0
                jmp     short loc_80483D1

loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1

loc_80483D1:
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

在 Linux 系统里执行这个程序会引发段错误/Segmentation fault。

我们使用 GDB Debugger 调试它：

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388    -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0    0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

因为 Linux 和 Windows 的栈结构不完全相同，所以 Linux/Win32 下寄存器的值也存在细微的差别。

18.3 缓冲区溢出的保护方法

无论 C/C++程序员是粗心大意还是严肃认真，我们都可通过编译器的技术手段防止缓冲区溢出。例如，可以启用 MSVC 的编译器选项[①]：

```
/RTCs  启用栈帧的实时检查
/GZ 启用栈检测(/RTCs)
```

另外，还可以在函数启动时构造一些本地变量，在里面写入随即数，在函数结束之前检查这些值是否发生变化。如果这些值发生了变化，就不应当 以 RET 方式结束函数，而是通过其他手段终止（stop/hang）

① 请参见维基百科 https://en.wikipedia.org/wiki/Buffer_overflow_protection。

程序。强行关闭应用程序或许确实不怎么体面，但是总比让他人入侵主机要高明。

有人把这种写入的随机值叫作“百灵鸟 canary”，这个绰号来自于“矿工的百灵鸟 miners' canary”。所谓“矿工的百灵鸟”，确实是指过去矿工下矿时所带的百灵鸟。矿工使用百灵鸟来快速检测矿道里的毒气。百灵鸟对井下气体十分敏感，它们在遇到不良气体的时候就会表现得烦躁不安，甚至会直接死亡。

如果启用 MSVC 的 RTC1 和 RTCs 选项编译本章开头的那段程序，就会在汇编指令里看到函数在退出之前调用@_RTC_CheckStackVars@8，用以检测“百灵鸟”是否会报警。

还是来看看 GCC 预防缓冲区溢出的方法吧。我们借用 5.2.4 节的例子来进行说明：

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

即使不指定任何参数，默认情况下 GCC 4.7.3 也会在代码里加入百灵鸟。

经 GCC 4.7.3 编译前面的程序可得到如下所示的指令。

指令清单 18.7 GCC4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0  ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs:20;
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20;
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5:
        call    __stack_chk_fail
```

随机值位于 gs:20。在函数启动的时候，程序在栈里写入这个百灵鸟，并且在函数退出之前检测它是否

发生了变化、是否与 gs:20 一致。如果其值发生了变化，将会调用__stack_chk_fail，在 Ubuntu 13.04 x86 下，我们会在控制台看见下述报错信息：

```
*** buffer overflow detected ***: ./2_1 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586   /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586   /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586   /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0         [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602   /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602   /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602   /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781   /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781   /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781   /lib/i386-linux-gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0         [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794   /lib/i386-linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794   /lib/i386-linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794   /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0         [stack]
Aborted (core dumped)
```

gs 开头的寄存器就是常说的段寄存器。在 MS-DOS 和基于 DOS 的系统里，段寄存器的作用很广泛。但是，今天它的作用发生了变化。简单地说，Linux 下的 gs 寄存器总是指向 TLS（参见第 65 章）——存储线程的多种特定信息。（Win32 环境下的 fs 寄存器起到 Linux 下 gs 寄存器的作用。Win32 的 fs 寄存器指向 TIB[①]）

如需详细了解 gs 寄存器的作用，请在 3.11 以上版本的 Linux 源代码中查阅文件（arch/x86/include/asm/stackprotector.h）。这个文件的注释详细描述了有关变量的功能。

18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

我们使用 LLVM 编译本章开头的数组程序，看看它是如何使用百灵鸟的：

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
```

① TIB 是 Thread Information Block。请参见 https://en.wikipedia.org/wiki/Win32_Thread_Information_Block。

```
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH        {R4-R7,LR}
    ADD         R7, SP, #0xC
    STR.W       R8, [SP,#0xC+var_10]!
    SUB         SP, SP, #0x54
    MOVW        R0, #aObjc_methtype ; "objc_methtype"
    MOVS        R2, #0
    MOVT.W      R0, #0
    MOVS        R5, #0
    ADD         R0, PC
    LDR.W       R8, [R0]
    LDR.W       R0, [R8]
    STR         R0, [SP,#0x64+canary]
    MOVS        R0, #2
    STR         R2, [SP,#0x64+var_64]
    STR         R0, [SP,#0x64+var_60]
    MOVS        R0, #4
    STR         R0, [SP,#0x64+var_5C]
    MOVS        R0, #6
    STR         R0, [SP,#0x64+var_58]
    MOVS        R0, #8
    STR         R0, [SP,#0x64+var_54]
    MOVS        R0, #0xA
    STR         R0, [SP,#0x64+var_50]
    MOVS        R0, #0xC
    STR         R0, [SP,#0x64+var_4C]
    MOVS        R0, #0xE
    STR         R0, [SP,#0x64+var_48]
    MOVS        R0, #0x10
    STR         R0, [SP,#0x64+var_44]
    MOVS        R0, #0x12
    STR         R0, [SP,#0x64+var_40]
    MOVS        R0, #0x14
    STR         R0, [SP,#0x64+var_3C]
    MOVS        R0, #0x16
    STR         R0, [SP,#0x64+var_38]
    MOVS        R0, #0x18
    STR         R0, [SP,#0x64+var_34]
    MOVS        R0, #0x1A
    STR         R0, [SP,#0x64+var_30]
    MOVS        R0, #0x1C
    STR         R0, [SP,#0x64+var_2C]
    MOVS        R0, #0x1E
    STR         R0, [SP,#0x64+var_28]
    MOVS        R0, #0x20
    STR         R0, [SP,#0x64+var_24]
    MOVS        R0, #0x22
    STR         R0, [SP,#0x64+var_20]
    MOVS        R0, #0x24
    STR         R0, [SP,#0x64+var_1C]
    MOVS        R0, #0x26
    STR         R0, [SP,#0x64+var_18]
    MOV         R4, 0xFDA ; "a[%d]=%d\n"
    MOV         R0, SP
    ADDS        R6, R0, #4
```

```
    ADD         R4, PC
    B           loc_2F1C
; second loop begin

loc_2F14
    ADDS        R0, R5, #1
    LDR.W       R2, [R6, R5, LSL#2]
    MOV         R5, R0

loc_2F1C
    MOV         R0, R4
    MOV         R1, R5
    BLX         _printf
    CMP         R5, #0x13
    BNE         loc_2F14
    LDR.W       R0, [R8]
    LDR         R1, [SP,#0x64+canary]
    CMP         R0, R1
    ITTTT EQ        ;
    MOVEQ       R0, #0
    ADDEQ       SP, SP, #0x54
    LDREQ.W     R8, [SP+0x64+var_64],#4
    POPEQ       {R4-R7,PC}
    BLX         __stack_chk_fail
```

这段 Thumb-2 模式程序最显著的特点就是循环体被逐一展开了。LLVM 编译器判定这样的效率较高。另外，ARM 模式下这种指令的效率可能更高。本书就不再进行有关验证了。

函数尾声检测了“百灵鸟”的状态。如果栈内的值没被修改（和 R8 寄存器所指向的值相等），则会执行 ITTTT EQ 后面的 4 条指令——令 R0 为 0、运行函数尾声并且退出函数。如果“百灵鸟”发生了变化，程序将跳过 ITTTT EQ 后面的 4 条带“－EQ”的指令，转而调用__stack_chk_fail 进行异常处理，实际上会终止个程序。

18.4 其他

现在我们应该可以理解为什么 C/C++编译不了下面的程序了。[①]

```
void f(int size)
{
    int a[size];
...
};
```

在编译阶段，编译器必须确切地知道需要给数组分配多大的存储空间，它需要事先明确分配局部栈或数据段（全局变量）的格局，所以编译器无法处理上述这种长度可变的数组。

如果无法事先确定数组的长度，那么我们就应当使用 malloc()函数分配出一块内存，然后直接按照常规变量数组的方式访问这块内存；或者遵循 C99 标准（参见 ISO07，6.7.5/2）进行处理，但是遵循 C99 标准而设计出来的程序，内部实现的方法更接近 alloca()函数（详情请参阅 5.2.4 节）。

18.5 字符串指针

本节以下述程序为例。

指令清单 18.8 查询月份名称

```
#include <stdio.h>

const char* month1[]=
```

① 但是根据 C99 标准[ISO07，pp 6.7.5/2]这种可变长数组是合法数据，可以被编译。GCC 确实可以使用栈处理动态数组。这时候，它实现数组的方式和 5.2.4 节介绍过的 alloca（）函数的实现方式相近。

```
{
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
        return month1[month];
};
```

18.5.1 x64

指令清单 18.9 Optimizing MSVC 2013 x64

```
_DATA   SEGMENT
month1  DQ      FLAT:$SG3122
        DQ      FLAT:$SG3123
        DQ      FLAT:$SG3124
        DQ      FLAT:$SG3125
        DQ      FLAT:$SG3126
        DQ      FLAT:$SG3127
        DQ      FLAT:$SG3128
        DQ      FLAT:$SG3129
        DQ      FLAT:$SG3130
        DQ      FLAT:$SG3131
        DQ      FLAT:$SG3132
        DQ      FLAT:$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

上述程序的功能如下：

- 第一条指令 MOVSXD 把 ECX 的 32 位整型数值、连同其符号扩展位(sign-extension)扩展为 64 位整型数据，再存储于 RAX 寄存器中。ECX 存储的“月份”信息是 32 位整形数据。因为程序随后还要进行 64 位运算，所以需要把输入变量转换为 64 位值。

- 然后函数把指针表的地址存储于 RCX 寄存器。
- 最后，函数的输入变量(month)的值乘以 8、再与指针表的地址相加。因为是 64 位系统的缘故，每个地址（即指针）的数据需要占用 64 位（即 8 字节）。所以指针表中的每个元素都占用 8 字节空间。因此，最终字符串的地址要加上 month*8。MOV 指令不仅完成了字符串地址的计算，而且还完成了指针表的查询。在输入值为 1 时，函数将返回字符串“February”的指针地址。

编译器的优化编译结果更为直接。

指令清单 18.10 Optimizing GCC 4.9 x64

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

18.5.2 32 位 MSVC

使用 32 位 MSVC 编译器编译上述程序可得如下所示的指令。

指令清单 18.11 Optimizing MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

32 位程序就不用把输入值转化为 64 位数据了。此外，32 位系统的指针属于 4 字节数据，所以相关的计算因子变为了 4。

18.5.3 32 位 ARM

ARM in ARM mode

指令清单 18.12 Optimizing Keil 6/2013 (ARM mode)

```
get_month1 PROC
        LDR     r1,|L0.100|
        LDR     r0,[r1,r0,LSL #2]
        BX      lr
        ENDP

|L0.100|
        DCD     ||.data||

        DCB     "January",0
        DCB     "February",0
        DCB     "March",0
        DCB     "April",0
        DCB     "May",0
        DCB     "June",0
        DCB     "July",0
        DCB     "August",0
        DCB     "September",0
        DCB     "October",0
        DCB     "November",0
        DCB     "December",0

        AREA ||.data||, DATA, ALIGN=2
month1
        DCD     ||.conststring||
        DCD     ||.conststring||+0x8
        DCD     ||.conststring||+0x11
```

```
        DCD     ||.conststring||+0x17
        DCD     ||.conststring||+0x1d
        DCD     ||.conststring||+0x21
        DCD     ||.conststring||+0x26
        DCD     ||.conststring||+0x2b
        DCD     ||.conststring||+0x32
        DCD     ||.conststring||+0x3c
        DCD     ||.conststring||+0x44
        DCD     ||.conststring||+0x4d
```

数据表的首地址存储于 R1 寄存器。其余元素的指针地址则通过 LDR 指令现场计算。实参 month 在左移 2 位之后与 R1 的值（表地址）相加，从而计算出地址表中的指针地址。此后，再用这个指针在 32 位地址表中进行查询，把最终的字符串地址存储在 R0 寄存器里。

Thumb 模式的 ARM 程序

由于在 Thumb 模式里的 LDR 指令不能进行 LSL 运算，所以代码略长一些。

```
get_month1 PROC
        LSLS    r0,r0,#2
        LDR     r1,|L0.64|
        LDR     r0,[r1,r0]
        BX      lr
        ENDP
```

18.5.4　ARM64

指令清单 18.13　Optimizing GCC 4.9 ARM64

```
get_month1:
        adrp    x1, .LANCHOR0
        add     x1, x1, :lo12:.LANCHOR0
        ldr     x0, [x1,w0,sxtw 3]
        ret

.LANCHOR0 = . + 0
        .type   month1, %object
        .size   month1, 96
month1:
        .xword  .LC2
        .xword  .LC3
        .xword  .LC4
        .xword  .LC5
        .xword  .LC6
        .xword  .LC7
        .xword  .LC8
        .xword  .LC9
        .xword  .LC10
        .xword  .LC11
        .xword  .LC12
        .xword  .LC13
.LC2:
        .string "January"
.LC3:
        .string "February"
.LC4:
        .string "March"
.LC5:
        .string "April"
.LC6:
        .string "May"
.LC7:
        .string "June"
.LC8:
        .string "July"
.LC9:
```

```
        .string "August"
.LC10:
        .string "September"
.LC11:
        .string "October"
.LC12:
        .string "November"
.LC13:
        .string "December"
```

表地址由 ADRP/ADD 指令对传送到 X1 寄存器。然后，单条 LDR 指令完成表查询的运算。它把 W0 寄存器（传递实参 month）的值左移 3 位（相当于乘以 8），将其进行有符号数扩展（sxtw 后缀的含义）并与 X0 寄存器的值相加。通过表查询获取的 64 位运算结果最终存储在 X0 寄存器里。

18.5.5 MIPS

指令清单 18.14 Optimizing GCC 4.4.5 (IDA)

```
get_month1:
; load address of table into $v0:
                la      $v0, month1
; take input value and multiply it by 4:
                sll     $a0, 2
; sum up address of table and multiplied value:
                addu    $a0, $v0
; load table element at this address into $v0:
                lw      $v0, 0($a0)
; return
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP

                .data # .data.rel.local
                .globl month1
month1:         .word aJanuary          # "January"
                .word aFebruary         # "February"
                .word aMarch            # "March"
                .word aApril            # "April"
                .word aMay              # "May"
                .word aJune             # "June"
                .word aJuly             # "July"
                .word aAugust           # "August"
                .word aSeptember        # "September"
                .word aOctober          # "October"
                .word aNovember         # "November"
                .word aDecember         # "December"

                .data # .rodata.str1.4
aJanuary:       .ascii "January"<0>
aFebruary:      .ascii "February"<0>
aMarch:         .ascii "March"<0>
aApril:         .ascii "April"<0>
aMay:           .ascii "May"<0>
aJune:          .ascii "June"<0>
aJuly:          .ascii "July"<0>
aAugust:        .ascii "August"<0>
aSeptember:     .ascii "September"<0>
aOctober:       .ascii "October"<0>
aNovember:      .ascii "November"<0>
aDecember:      .ascii "December"<0>
```

18.5.6 数组溢出

本例参数的取值范围是 0～11。如果输入值为 12，会发生什么情况？虽然查询表里没有对应的值，但是程序还是会计算、取值并返回结果，只是返回值不可预料。如果其他函数访问这种超越数据边界的地址，

那么在读取字符串地址时可能引发程序崩溃。

本文通过 MSVC 编译器生成一个 Win64 程序，然后使用 IDA 打开这个可执行程序，观察编译器的 linker 组件在数组后面存储了哪些信息：

指令清单 18.15　在 IDA 里观察问题程序

```
off_140011000    dq offset aJanuary_1 ; DATA XREF: .text:0000000140001003
                                      ; "January"
                 dq offset aFebruary_1     ; "February"
                 dq offset aMarch_1        ; "March"
                 dq offset aApril_1        ; "April"
                 dq offset aMay_1          ; "May"
                 dq offset aJune_1         ; "June"
                 dq offset aJuly_1         ; "July"
                 dq offset aAugust_1       ; "August"
                 dq offset aSeptember_1    ; "September"
                 dq offset aOctober_1      ; "October"
                 dq offset aNovember_1     ; "November"
                 dq offset aDecember_1     ; "December"
aJanuary_1       db 'January',0            ; DATA XREF: sub_140001020+4
                                           ; .data:off_140011000
aFebruary_1      db 'February',0           ; DATA XREF: .data:0000000140011008
                 align 4
aMarch_1         db 'March',0              ; DATA XREF: .data:0000000140011010
                 align 4
aApril_1         db 'April',0              ; DATA XREF: .data:0000000140011018
```

本例的程序很小，所以数据段里的信息以月份名称为主，没有什么其他信息。但是应当注意的是，linker 可能在实际生成的程序中安插“任何信息”。

输入值为 12 时会发生什么情况？程序应当返回数组首地址之后的第 13 个元素。我们一起来看 CPU 是如何处理第“13”个元素的那个 64 位值：

指令清单 18.16　Executable file in IDA

```
off_140011000    dq offset qword_140011060
                                      ; DATA XREF: .text:0000000140001003
                 dq offset aFebruary_1     ; "February"
                 dq offset aMarch_1        ; "March"
                 dq offset aApril_1        ; "April"
                 dq offset aMay_1          ; "May"
                 dq offset aJune_1         ; "June"
                 dq offset aJuly_1         ; "July"
                 dq offset aAugust_1       ; "August"
                 dq offset aSeptember_1    ; "September"
                 dq offset aOctober_1      ; "October"
                 dq offset aNovember_1     ; "November"
                 dq offset aDecember_1     ; "December"
                 qword_140011060 dq 797261756E614Ah ; DATA XREF: sub_140001020+4
                                      ; .data:off_140011000
                 aFebruary_1 db 'February',0 ; DATA XREF: .data:0000000140011008
                 align 4
                 aMarch_1 db 'March',0     ; DATA XREF: .data:0000000140011010
```

那个地址的值是 0x797261756E614A。假如某个函数以字符串指针的格式访问这个地址，那么整个程序多半会立即崩溃。毕竟这个值不太会是一个有效地址。

数组溢出保护

如果指望调用函数的用户能够保证参数不超出正常的取值范围，那么这个程序员真的算得上是很傻很天真了。在编程领域里，成熟的应用程序应当具备“Fail Early, Fail Loudly”或“Fail-fast”这类的早期错误处理功能。

C/C++ assertions 就是处理手段之一。我们可以修改源程序，令其验证输入参数是否在预期范围之内。

指令清单 18.17 assert() added

```
const char* get_month1_checked (int month)
{
        assert (month<12);
        return month1[month];
};
```

只要在函数启动的时候调用一个验证参数有效性的 assert()函数，那么当输入值在取值区间之外时，程序将进行异常处理。

指令清单 18.18 Optimizing MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
        DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
        DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
        push    rbx
        sub     rsp, 32
        movsxd  rbx, ecx
        cmp     ebx, 12
        jl      SHORT $LN3@get_month1
        lea     rdx, OFFSET FLAT:$SG3143
        lea     rcx, OFFSET FLAT:$SG3144
        mov     r8d, 29
        call    _wassert
$LN3@get_month1:
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rbx*8]
        add     rsp, 32
        pop     rbx
        ret     0
get_month1_checked ENDP
```

严格地说，assert()只能算作宏而不能算作函数。它对条件表达式进行判断，然后把行号信息和文件名信息传递给另外一个函数，并由后者向用户进行提示。

在可执行程序里，我们可以看到以 UTF-16 格式封装文件名和条件表达式，以及 assert()在原文件中的行号（29）。

所有编译器在编译 assert 宏时的处理机制基本都大同小异。本文仅演示 GCC 的编译结果。

指令清单 18.19 Optimizing GCC 4.9 x64

```
.LC1:
        .string "month.c"
.LC2:
        .string "month<12"

get_month1_checked:
        cmp     edi, 11
        jg      .L6
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
.L6:
        push    rax
        mov     ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
        mov     edx, 29
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:.LC2
```

```
        call    __assert_fail

__PRETTY_FUNCTION__.2423:
        .string "get_month1_checked"
```

从中可知，GCC 向异常处理函数传递了其调用方函数的函数名称。

这种边界检查其实存在很明显的开销问题。它会降低程序的运行速度——在调用间隔较短的时间敏感函数上使用 assert()宏，性能影响会非常大。以 MSVC 为例，Debug 版的 MSVC 还存在大量的 assert()，而最终发行版的 MSVC 里就不再使用这个宏了。

微软 Windows NT 内核也有“checked”和“free”两个版本。前一个版本还存在输入值的边界检查，而后面的第二版就不再使用 asssert()了。

18.6　多维数组

多维数组和线性数组在本质上是相同的。

计算机内存是连续的线性空间，它可以与一维数组直接对应。在被拆分成多个一维数组之后，多维数组与内栈线性空间同样存在直接对应的存储关系。

举例来说，二维数组 a[3][4]可转化为 a[12]这样的一个 12 个元素一维数组，如表 18.1 所示。

表 18.1　使用一维数组表示二维数组

存储地址	数组元素
0	[0] [0]
1	[0] [1]
2	[0] [2]
3	[0] [3]
4	[1] [0]
5	[1] [1]
6	[1] [2]
7	[1] [3]
8	[2] [0]
9	[2] [1]
10	[2] [2]
11	[2] [3]

在内存之中，3×4 的二维数组将依次存储为连续的 12 个元素，如表 18.2 所示。

表 18.2　二维数组在内存中的存储逻辑

0	1	2	3
4	5	6	7
8	9	10	11

在计算上述数组中某个特定元素的内存存储编号时，可以先将二维索引号的第一个索引号乘以 4（矩阵宽度），而后加上第二个索引号。这种方式就是 C/C++、Python 所用的“行优先的顺序”（row-majororder）。所谓行优先，就是先用第一行排满第一个索引号下的所有元素，然后再依次编排其他各行。

相应地，也有“列优先的顺序”(column-major order)。Fortran、Matlab、R 语言就使用了这种顺序。这种顺序优先使用列的存储空间。

这两种优先的顺序孰优孰劣？从性能及缓存的角度来看，与数据的存储方案（scheme)和组织方式（data organization）匹配的优先顺序最优。只要相互匹配，那么程序就可以连续访问数据，整体性能就会提高。所以，如果程序以“逐行”的方式访问数据，那么就应当以行优先的顺序组织数组；反之亦然。

18.6.1 二维数组举例

本例用 char 类型数据构造一个二维数组，其中每个数组元素都占用 1 字节内存。

以行优先的顺序填充数据

下面这个例子将用数字“0～3”填充数组的第二行。

指令清单 18.20 逐行填充数据

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        //清空数组
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0

        //填充第二行
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

图 18.7 中所示的 3 个红圈代表 3 个不同的行。我们可以看到第二行的值是 0、1、2、3。

Address	Hex dump
00C33370	00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380	02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 18.7 OllyDbg：填充数据后的数组

以列优先的顺序填充数据

下列程序将数组的第三列填充为 0、1、2。

指令清单 18.21 逐列填充数据

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // 清空数组
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        //填充第 3 列
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

图 18.8 中所示的 3 个红圈代表 3 个不同的行。各行的第三列分别是 0、1、2。

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E AA EF 31 1E AA EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 18.8 OllyDbg：填充数据之后的数组

18.6.2 以一维数组的方式访问二维数组

以一维数组的方式访问二维数组的方法至少有两种。

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // treat input array as one-dimensional
        // 4 is array width here
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
        // treat input array as pointer,
        // calculate address, get value at it
        // 4 is array width here
        return *(array+a*4+b);
};

int main() {
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

编译并运行上述程序，可观测到数组元素的相应数值。

MSVC 2013 的编译方法可圈可点，三个函数的指令完全一致。

指令清单 18.22 Optimizing MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=address of array
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+address of array
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=load byte at address RAX+R9*4=b+address of array+a*4=address of array+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP
```

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

GCC 会生成等效的指令，只是各函数的指令略有区别。

指令清单 18.23 Optimizing GCC 4.9 x64

```
; RDI=address of array
; RSI=a
; RDX=b

get_by_coordinates1:
; sign-extend input 32-bit int values "a" and "b" to 64-bit ones
        movsx   rsi, esi
        movsx   rdx, edx
        lea     rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=address of array+a*4
        movzx   eax, BYTE PTR [rax+rdx]
; AL=load byte at address RAX+RDX=address of array+a*4+b
        ret

get_by_coordinates2:
        lea     eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
        cdqe
        movzx   eax, BYTE PTR [rdi+rax]
; AL=load byte at address RDI+RAX=address of array+b+a*4
        ret

get_by_coordinates3:
        sal     esi, 2
; ESI=a<<2=a*4
; sign-extend input 32-bit int values "a*4" and "b" to 64-bit ones
        movsx   rdx, edx
        movsx   rsi, esi
        add     rdi, rsi
; RDI=RDI+RSI=address of array+a*4
        movzx   eax, BYTE PTR [rdi+rdx]
; AL=load byte at address RDI+RDX=address of array+a*4+b
        ret
```

18.6.3 三维数组

多维数组的情况也差不多。

在下面的例子里，我们演示一个由整型数据构成的三维数组，每个整数元素占用 4 字节内存。

我们来看如下所示的指令。

指令清单 18.24 simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

用 MSVC 2010 编译后，这个函数对应的汇编代码如下所示。

指令清单 18.25 MSVC 2010

```
_DATA   SEGMENT
COMM    _a:DWORD:01770H
_DATA   ENDS
PUBLIC  _insert
_TEXT   SEGMENT
_x$ = 8             ; size = 4
_y$=12              ; size = 4
_z$=16              ; size = 4
_value$ = 20        ; size = 4
_insert PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _x$[ebp]
    imul    eax, 2400               ; eax=600*4*x
    mov     ecx, DWORD PTR _y$[ebp]
    imul    ecx, 120                ; ecx=30*4*y
    lea     edx, DWORD PTR _a[eax+ecx]; edx=a + 600*4*x + 30*4*y
    mov     eax, DWORD PTR _z$[ebp]
    mov     ecx, DWORD PTR _value$[ebp]
    mov     DWORD PTR [edx+eax*4], ecx; *(edx+z*4)=value
    pop     ebp
    ret     0
_insert ENDP
_TEXT   ENDS
```

中规中矩。对于三维数组（int a[x][y][z];）来说，计算各元素指针地址的公式是：数组元素地址=600×4x + 30×4y + 4z。32 位系统里 int 类型是 32 位（4 字节）数据，所以要每项都要乘以 4。

使用 GCC 4.4.1 编译上述程序可得如下所示的指令。

指令清单 18.26 GCC 4.4.1

```
        public  insert
insert  proc near

x       = dword ptr 8
y       = dword ptr  0Ch
z       = dword ptr  10h
value   = dword ptr  14h

        push    ebp
        mov     ebp, esp
        push    ebx
        mov     ebx, [ebp+x]
        mov     eax, [ebp+y]
        mov     ecx, [ebp+z]
        lea     edx, [eax+eax]              ; edx=y*2
        mov     eax, edx                    ; eax=y*2
        shl     eax, 4                      ; eax=(y*2)<<4 = y*2*16 = y*32
        sub     eax, edx                    ; eax=y*32 - y*2=y*30
        imul    edx, ebx, 600               ; edx=x*600
        add     eax, edx                    ; eax=eax+edx=y*30 + x*600
        lea     edx, [eax+ecx]              ; edx=y*30 + x*600 + z
        mov     eax, [ebp+value]
        mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=value
        pop     ebx
        pop     ebp
        retn
insert  endp
```

GCC 的处理方法和 MSVC 不同。在计算 30y 的时候，GCC 没有使用乘法指令。它的计算公式是(y + y) << 4 − (y + y) = (2 y)<< 4 − 2 y = 2×16y − 2 y = 32 y − 2 y = 30 y。可见，GCC 组合了加法、位移和减法运算，替代了 MSVC 采用的乘法运算。GCC 算法的运算速度比直接进行乘法运算的速度还要快。

ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

指令清单 18.27 Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_insert

value   = -0x10
z       = -0xC
y       =-8
x       =-4

; allocate place in local stack for 4 values of int type
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9]  ; get pointer to array
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1]   ; R1 - address of array element
; deallocate place in local stack, allocated for 4 values of int type
ADD     SP, SP, #0x10
BX      LR
```

在不启用优化选项的情况下，LLVM 使用栈保存所有变量。当然这样的程序运行效率不怎么好。在数组方面，计算各元素的内存地址的方法和前文相同。

ARM + Optimizing Xcode 4.6.3 (LLVM) + Thumb mode

启用优化选项后，使用 LLVM 以 Thumb 模式编译上述程序可得到如下所示的指令。

指令清单 18.28 Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1,R1,R1,LSL#4     ;R1-y.R1=y<<4-y=y*16-y=y*15
ADD     R9, PC
LDR.W   R9, [R9]           ; R9 = pointer to a array
MLA.W   R0, R0, R12, R9    ; R0 - x, R12 - 2400, R9 - pointer to a. R0=x*2400 + ptr to a
ADD.W   R0, R0, R1,LSL#3   ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ptr to a + y*15*8 =
                           ; ptr to a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2]  ; R2 - z, R3 - value. address=R0+z*4 =
BX      LR                 ; ptr to a + y*30*4 + x*600*4 + z*4
```

LLVM 也用到了 GCC 的地址计算方法，组合了加法、位移、减法三种运算。

其中的“RSB（Reverse Subtract）”指令尚未在前面介绍过。RSB 指令称为逆向减法指令，用于把操作数 2（第三个参数）减去操作数 1（第二个参数），并将结果存放到目的寄存器中。为什么专门从 SUB 指令里派生出 RSB 指令呢？这是因为 SUB 和 RSB 都只能对第二个操作数进行位移运算。如果要使用 SUB 指令，则 LSL #4 就要写

在第 1 个操作数后面，会造成识别上的混乱。而 RSB 指令替换了减数和被减数的位置，从而避免了这个问题。

“LDR.W R9, [R9]”指令和 x86 的 LEA 指令相似。只是这里的这条指令不影响计算结果，所以是冗余代码。很明显，编译器没有进行相应优化。

MIPS

因为源程序很小的缘故，GCC 以全局指针 GP 在 64KB 可寻址空间进行表查询。

指令清单 18.29 Optimizing GCC 4.4.5 (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=value
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = x*32
                sll     $a0, 3
; $a0 = $a0<<3 = x*8
                addu    $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
                sll     $v1, $a1, 5
; $v1 = $a1<<5 = y*32
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
                sll     $a1, 1
; $a1 = $a1<<1 = y*2
                subu    $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
                subu    $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
                la      $gp, __gnu_local_gp
                addu    $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
                addu    $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; load address of table:
                lw      $v0, (a & 0xFFFF)($gp)
; multiply index by 4 to seek array element:
                sll     $a0, 2
; sum up multiplied index and table address:
                addu    $a0, $v0, $a0
; store value into table and return:
                jr      $ra
                sw      $a3, 0($a0)

                .comm a:0x1770
```

18.6.4 更多案例

计算机的显示屏幕是一个 2D 显示空间，但是显存却是一个一维线性数组。

本书第 83 章将详细介绍有关内容。

18.7 二维字符串数组的封装格式

在指令清单 18.8 所示的程序里，如果要制备指向月份名称的指针，就必须要进行一次以上的内存操作。有没有可能略去这步内存加载的内存操作？只要把二维数组进行相应处理，即可省略有关操作。

```
#include <stdio.h>
#include <assert.h>
```

```
const char month2[12][10]=
{
        { 'J','a','n','u','a','r','y',  0,  0,  0 },
        { 'F','e','b','r','u','a','r','y',  0,  0 },
        { 'M','a','r','c','h',  0,  0,  0,  0,  0 },
        { 'A','p','r','i','l',  0,  0,  0,  0,  0 },
        { 'M','a','y',  0,  0,  0,  0,  0,  0,  0 },
        { 'J','u','n','e',  0,  0,  0,  0,  0,  0 },
        { 'J','u','l','y',  0,  0,  0,  0,  0,  0 },
        { 'A','u','g','u','s','t',  0,  0,  0,  0 },
        { 'S','e','p','t','e','m','b','e','r',  0 },
        { 'O','c','t','o','b','e','r',  0,  0,  0 },
        { 'N','o','v','e','m','b','e','r',  0,  0 },
        { 'D','e','c','e','m','b','e','r',  0,  0 }
};

// in 0..11 range
const char* get_month2 (int month)
{
        return &month2[month][0];
};
```

编译上述程序可得如下所示的指令。

指令清单 18.30 Optimizing MSVC 2013 x64

```
month2  DB      04aH
        DB      061H
        DB      06eH
        DB      075H
        DB      061H
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
...
get_month2 PROC
; sign-extend input argument and promote to 64-bit value
        movsxd  rax, ecx
        lea rcx, QWORD PTR [rax+rax*4]
; RCX=month+month*4=month*5
        lea     rax, OFFSET FLAT:month2
; RAX=pointer to table
        lea     rax, QWORD PTR [rax+rcx*2]
; RAX=pointer to table + RCX*2=pointer to table + month*5*2=pointer to table + month*10
        ret     0
get_month2 ENDP
```

上述程序完全不访问内存。整个函数的功能，只是计算月份名称字符串的首字母指针pointer_to_the_table+month*10。它使用单条 LEA 指令，替代了多条 MUL 和 MOV 指令。

上述数组的每个字符串都占用 10 字节空间。最长的字符串由“September”和内容为零的字节构成，其余的字符串使用零字节对齐，所以每个字符串都占用 10 个字节。如此一来，计算字符串首地址的方式变得简单，整个函数的效率也会有所提高。

使用 GCC 4.9 进行优化编译，程序的指令甚至会更少。

指令清单 18.31 Optimizing GCC 4.9 x64

```
        movsx   rdi, edi
        lea     rax, [rdi+rdi*4]
        lea     rax, month2[rax+rax]
        ret
```

它同样使用 LEA 指令进行“乘以 10”的运算。

若由 GCC 进行非优化编译，那么乘法运算的实现方式将会有所不同。

指令清单 18.32 Non-optimizing GCC 4.9 x64

```
get_month2:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
; RDX = sign-extended input value
        mov     rax, rdx
; RAX = month
        sal     rax, 2
; RAX = month<<2 = month*4
        add     rax, rdx
; RAX = RAX+RDX = month*4+month = month*5
        add     rax, rax
; RAX = RAX*2 = month*5*2 = month*10
        add     rax, OFFSET FLAT:month2
; RAX = month*10 + pointer to the table
        pop     rbp
        ret
```

而在不开启优化选项的情况下，MSVC 编译器会直接使用 IMUL 指令。

指令清单 18.33 Non-optimizing MSVC 2013 x64

```
month$ = 8
get_month2 PROC
        mov     DWORD PTR [rsp+8], ecx
        movsxd  rax, DWORD PTR month$[rsp]
; RAX = sign-extended input value into 64-bit one
        imul    rax, rax, 10
; RAX = RAX*10
        lea     rcx, OFFSET FLAT:month2
; RCX = pointer to the table
        add     rcx, rax
; RCX = RCX+RAX = pointer to the table+month*10
        mov     rax, rcx
; RAX = pointer to the table+month*10
        mov     ecx, 1
; RCX = 1
        imul    rcx, rcx, 0
; RCX = 1*0 = 0
        add     rax, rcx
; RAX = pointer to the table+month*10 + 0 = pointer to the table+month*10
        ret     0
get_month2 ENDP
```

不过此处有些令人费解：为什么 RCX 要乘以零，再把零加在最终结果里？虽然说它不影响最终运行结果，但是也足以说是编译器的某种怪癖了。本文刻意演示这种怪癖代码，是希望读者不要拘泥于编译器生成的指令的形式，而要从编程人员的角度理解程序的源代码。

18.7.1 32 位 ARM

由 Keil 优化编译而生成的 Thumb 模式程序，会直接使用乘法运算指令 MULS。

指令清单 18.34 Optimizing Keil 6/2013 (Thumb mode)

```
; R0 = month
        MOVS    r1,#0xa
; R1 = 10
        MULS    r0,r1,r0
; R0 = R1*R0 = 10*month
```

```
        LDR     r1,|L0.68|
; R1 = pointer to the table
        ADDS    r0,r0,r1
; R0 = R0+R1 = 10*month + pointer to the table
        BX      lr
```

而优化编译生成的 ARM 模式程序，会使用加法运算和位移操作指令。

指令清单 18.35 Optimizing Keil 6/2013 (ARM mode)

```
; R0 = month
        LDR     r1,|L0.104|
; R1 = pointer to the table
        ADD     r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = month*5
        ADD     r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = pointer to the table + month*5*2 = pointer to the table + month*10
        BX      lr
```

18.7.2 ARM64

指令清单 18.36 Optimizing GCC 4.9 ARM64

```
; W0 = month
        sxtw    x0, w0
; X0 = sign-extended input value
        adrp    x1, .LANCHOR1
        add     x1, x1, :lo12:.LANCHOR1
; X1 = pointer to the table
        add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
        add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = pointer to the table + X0*10
        ret
```

SXTW 指令把 32 位有符号数据扩展为 64 位有符号数据，并将之存储于 X0 寄存器。ADRP/ADD 指令对则负责数组寻址。ADD 指令中的 LSL 后缀用于进行乘法运算。

18.7.3 MIPS

指令清单 18.37 Optimizing GCC 4.4.5 (IDA)

```
                .globl get_month2
get_month2:
; $a0=month
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = month*8
                sll     $a0, 1
; $a0 = $a0<<1 = month*2
                addu    $a0, $v0
; $a0 = month*2+month*8 = month*10
; load address of the table:
                la      $v0, month2
; sum up table address and index we calculated and return:
                jr      $ra
                addu    $v0, $a0

month2:         .ascii "January"<0>
                .byte 0, 0
aFebruary:      .ascii "February"<0>
                .byte   0
aMarch:         .ascii "March"<0>
                .byte 0, 0, 0, 0
aApril:         .ascii "April"<0>
                .byte 0, 0, 0, 0
```

```
aMay:           .ascii "May"<0>
                .byte 0, 0, 0, 0, 0, 0
aJune:          .ascii "June"<0>
                .byte 0, 0, 0, 0, 0
aJuly:          .ascii "July"<0>
                .byte 0, 0, 0, 0, 0
aAugust:        .ascii "August"<0>
                .byte 0, 0, 0
aSeptember:     .ascii "September"<0>
aOctober:       .ascii "October"<0>
                .byte 0, 0
aNovember:      .ascii "November"<0>
                .byte   0
aDecember:      .ascii "December"<0>
                .byte 0, 0, 0, 0, 0, 0, 0, 0, 0
```

18.7.4 总结

字符串存储技术属于经典的数组技术。Oracle RDBMS 的程序中就大量使用了这种技术。虽然当代计算机程序中可能很少应用这种数组技术了，但是字符串型数据可充分演示数组的各方面特点。

18.8 本章小结

在内存中，数组就是依次排列的一组同类型数据。数组元素可以是任意类型的数据，甚至可以是结构体型的数据。如果要访问数组中的特定元素，首先就要计算其地址。

18.9 练习题

18.9.1 题目 1

请描述下述代码的功能。

指令清单 18.38 MSVC 2010+/O1

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4

?s@@YAXPAN00@Z PROC; s, COMDAT
    mov     eax, DWORD PTR _b$[esp-4]
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _c$[esp-4]
    push    esi
    push    edi
    sub     ecx, eax
    sub     edx, eax
    mov     edi, 200    ; 000000c8H
$LL6@s:
    push    100         ; 00000064H
    pop     esi
$LL3@s:
    fld     QWORD PTR [ecx+eax]
    fadd    QWORD PTR [eax]
    fstp    QWORD PTR [edx+eax]
    add     eax, 8
    dec     esi
    jne     SHORT $LL3@s
    dec     edi
    jne     SHORT $LL6@s
```

```
    pop     edi
    pop     esi
    ret     0
?s@@YAXPAN00@Z  ENDP  ; s
```

/O1 代表 minimize space，即生成最小长度的程序。

通过 Keil 5.03 （启用优化选项-O3）编译而得的 ARM 模式代码如下所示。

指令清单 18.39 Optimizing Keil 6/2013 (ARM mode)

```
        PUSH    {r4-r12,lr}
        MOV     r9,r2
        MOV     r10,r1
        MOV     r11,r0
        MOV     r5,#0
|L0.20|
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        MOV     r4,#0
        ADD     r8,r10,r0,LSL #5
        ADD     r7,r11,r0,LSL #5
        ADD     r6,r9,r0,LSL #5
|L0.44|
        ADD     r0,r8,r4,LSL #3
        LDM     r0,{r2,r3}
        ADD     r1,r7,r4,LSL #3
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADD     r2,r6,r4,LSL #3
        ADD     r4,r4,#1
        STM     r2,{r0,r1}
        CMP     r4,#0x64
        BLT     |L0.44|
        ADD     r5,r5,#1
        CMP     r5,#0xc8
        BLT     |L0.20|
        POP     {r4-r12,pc}
```

指令清单 18.40 Optimizing Keil 6/2013 (Thumb mode)

```
        PUSH    {r0-r2,r4-r7,lr}
        MOVS    r4, #0
        SUB     sp, sp, #8
|L0.6|
        MOVS    r1,#0x19
        MOVS    r0,r4
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#8]
        LDR     r1,[sp,#0xc]
        ADDS    r2,r0,r2
        STR     r2,[sp,#0]
        LDR     r2,[sp,#0x10]
        MOVS    r5,#0
        ADDS    r7,r0,r2
        ADDS    r0,r0,r1
        STR     r0,[sp,#4]
|L0.32|
        LSLS    r6,r5,#3
        ADDS    r0,r0,r6
        LDM     r0!,{r2,r3}
        LDR     r0,[sp,#0]
        ADDS    r1,r0,r6
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADDS    r2,r7,r6
        ADDS    r5,r5,#1
```

```
        STM     r2!,{r0,r1}
        CMP     r5,#0x64
        BGE     |L0.62|
        LDR     r0,[sp,#4]
        B       |L0.32|
|L0.62|
        ADDS    r4,r4,#1
        CMP     r4,#0xc8
        BLT     |L0.6|
        ADD     sp,sp,#0x14
        POP     {r4-r7,pc}
```

指令清单 18.41 Non-optimizing GCC 4.9 (ARM64)

```
s:
        sub     sp, sp, #48
        str     x0, [sp,24]
        str     x1, [sp,16]
        str     x2, [sp,8]
        str     wzr, [sp,44]
        b       .L2
.L5:
        str     wzr, [sp,40]
        b       .L3
.L4:
        ldr     w1, [sp,44]
        mov     w0, 100
        mul     w0, w1, w0
        sxtw    x1, w0
        ldrsw   x0, [sp,40]
        add     x0, x1, x0
        lsl     x0, x0, 3
        ldr     x1, [sp,8]
        add     x0, x1, x0
        ldr     w2, [sp,44]
        mov     w1, 100
        mul     w1, w2, w1
        sxtw    x2, w1
        ldrsw   x1, [sp,40]
        add     x1, x2, x1
        lsl     x1, x1, 3
        ldr     x2, [sp,24]
        add     x1, x2, x1
        ldr     x2, [x1]
        ldr     w3, [sp,44]
        mov     w1, 100
        mul     w1, w3, w1
        sxtw    x3, w1
        ldrsw   x1, [sp,40]
        add     x1, x3, x1
        lsl     x1, x1, 3
        ldr     x3, [sp,16]
        add     x1, x3, x1
        ldr     x1, [x1]
        fmov    d0, x2
        fmov    d1, x1
        fadd    d0, d0, d1
        fmov    x1, d0
        str     x1, [x0]
        ldr     w0, [sp,40]
        add     w0, w0, 1
        str     w0, [sp,40]
.L3:
        ldr     w0, [sp, 40]
        cmp     w0, 99
        ble     .L4
        ldr     w0, [sp,44]
```

```
        add     w0, w0, 1
        str     w0, [sp,44]
.L2:
        ldr     w0, [sp,44]
        cmp     w0, 199
        ble     .L5
        add     sp, sp, 48
        ret
```

指令清单 18.42 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
sub_0:
        li      $t3, 0x27100
        move    $t2, $zero
        li      $t1, 0x64  # 'd'

loc_10:                                 # CODE XREF: sub_0+54
        addu    $t0, $a1, $t2
        addu    $a3, $a2, $t2
        move    $v1, $a0
        move    $v0, $zero

loc_20:                                 # CODE XREF: sub_0+48
        lwc1    $f2, 4($v1)
        lwc1    $f0, 4($t0)
        lwc1    $f3, 0($v1)
        lwc1    $f1, 0($t0)
        addiu   $v0, 1
        add.d   $f0, $f2, $f0
        addiu   $v1, 8
        swc1    $f0, 4($a3)
        swc1    $f1, 0($a3)
        addiu   $t0, 8
        bne     $v0, $t1, loc_20
        addiu   $a3, 8
        addiu   $t2, 0x320
        bne     $t2, $t3, loc_10
        addiu   $a0, 0x320
        jr      $ra
        or      $at, $zero
```

18.9.2 题目 2

请描述下述程序的功能。

通过 MSVC 2010 （启用 /O1 选项）编译而得的代码如下所示。

指令清单 18.43 MSVC 2010 + /O1

```
tv315 = -8              ; size = 4
tv291 = -4              ; size = 4
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_c$ = 16                ; size = 4
?m@@YAXPAN00@Z PROC; m, COMDAT
    push   ebp
    mov    ebp, esp
    push   ecx
    push   ecx
    mov    edx, DWORD PTR _a$[ebp]
    push   ebx
    mov    ebx, DWORD PTR _c$[ebp]
    push   esi
    mov    esi, DWORD PTR _b$[ebp]
    sub    edx, esi
```

```
    push   edi
    sub    esi, ebx
    mov    DWORD PTR tv315[ebp], 100  ; 00000064H
$LL9@m:
    mov    eax, ebx
    mov    DWORD PTR tv291[ebp], 300  ; 0000012cH
$LL6@m:
    fldz
    lea    ecx, DWORD PTR [esi+eax]
    fstp   QWORD PTR [eax]
    mov    edi, 200                   ; 000000c8H
$LL3@m:
    dec    edi
    fld    QWORD PTR [ecx+edx]
    fmul   QWORD PTR [ecx]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [eax]
    jne    HORT $LL3@m
    add    eax, 8
    dec    DWORD PTR tv291[ebp]
    jne    SHORT $LL6@m
    add    ebx, 800                   ; 00000320H
    dec    DWORD PTR tv315[ebp]
    jne    SHORT $LL9@m
    pop    edi
    pop    esi
    pop    ebx
    leave
    ret    0
?m@@YAXPAN00@Z ENDP                   ; m
```

指令清单 18.44 Optimizing Keil 6/2013 (ARM mode)

```
        PUSH    {r0-r2,r4-r11,lr}
        SUB     sp,sp,#8
        MOV     r5,#0
|L0.12|
        LDR     r1,[sp,#0xc]
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        ADD     r1,r1,r0,LSL #5
        STR     r1,[sp,#0]
        LDR     r1,[sp,#8]
        MOV     r4,#0
        ADD     r11,r1,r0,LSL #5
        LDR     r1,[sp,#0x10]
        ADD     r10,r1,r0,LSL #5
|L0.52|
        MOV     r0,#0
        MOV     r1,r0
        ADD     r7,r10,r4,LSL #3
        STM     r7,{r0,r1}
        MOV     r6,r0
        LDR     r0,[sp,#0]
        ADD     r8,r11,r4,LSL #3
        ADD     r9,r0,r4,LSL #3
|L0.84|
        LDM     r9,{r2,r3}
        LDM     r8,{r0,r1}
        BL      __aeabi_dmul
        LDM     r7,{r2,r3}
        BL      __aeabi_dadd
        ADD     r6,r6,#1
        STM     r7,{r0,r1}
        CMP     r6,#0xc8
        BLT     |L0.84|
```

```
        ADD     r4,r4,#1
        CMP     r4,#0x12c
        BLT     |L0.52|
        ADD     r5,r5,#1
        CMP     r5,#0x64
        BLT     |L0.12|
        ADD     sp,sp,#0x14
        POP     {r4-r11,pc}
```

指令清单 18.45 Optimizing Keil 6/2013 (Thumb mode)

```
        PUSH    {r0-r2,r4-r7,lr}
        MOVS    r0,#0
        SUB     sp,sp,#0x10
        STR     r0,[sp,#0]
|L0.8|
        MOVS    r1,#0x19
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#0x10]
        LDR     r1,[sp,#0x14]
        ADDS    r2,r0,r2
        STR     r2,[sp,#4]
        LDR     r2,[sp,#0x18]
        MOVS    r5,#0
        ADDS    r7,r0,r2
        ADDS    r0,r0,r
        STR     r0,[sp,#8]
|L0.32|
        LSLS    r4,r5,#3
        MOVS    r0,#0
        ADDS    r2,r7,r4
        STR     r0,[r2,#0]
        MOVS    r6,r0
        STR     r0,[r2,#4]
|L0.44|
        LDR     r0,[sp,#8]
        ADDS    r0,r0,r4
        LDM     r0!,{r2,r3}
        LDR     r0,[sp,#4]
        ADDS    r1,r0,r4
        LDM     r1,{r0,r1}
        BL      __aeabi_dmul
        ADDS    r3,r7,r4
        LDM     r3,{r2,r3}
        BL      __aeabi_dadd
        ADDS    r2,r7,r4
        ADDS    r6,r6,#1
        STM     r2!,{r0,r1}
        CMP     r6,#0xc8
        BLT     |L0.44|
        MOVS    r0,#0xff
        ADDS    r5,r5,#1
        ADDS    r0,r0,#0x2d
        CMP     r5,r0
        BLT     |L0.32|
        LDR     r0,[sp,#0]
        ADDS    r0,r0,#1
        CMP     r0,#0x64
        STR     r0,[sp,#0]
        BLT     |L0.8|
        ADD     sp,sp,#0x1c
        POP     {r4-r7,pc}
```

指令清单 18.46 Non-optimizing GCC 4.9 (ARM64)

```
m:
        sub     sp, sp, #48
```

```
        str     x0, [sp,24]
        str     x1, [sp,16]
        str     x2, [sp,8]
        str     wzr, [sp,44]
        b       .L2
.L7:
        str     wzr, [sp,40]
        b       .L3
.L6:
        ldr     w1, [sp,44]
        mov     w0, 100
        mul     w0, w1, w0
        sxtw    x1, w0
        ldrsw   x0, [sp,40]
        add     x0, x1, x0
        lsl     x0, x0, 3
        ldr     x1, [sp,8]
        add     x0, x1, x0
        ldr     x1, .LC0
        str     x1, [x0]
        str     wzr, [sp,36]
        b       .L4
.L5:
        ldr     w1, [sp,44]
        mov     w0, 100
        mul     w0, w1, w0
        sxtw    x1, w0
        ldrsw   x0, [sp,40]
        add     x0, x1, x0
        lsl     x0, x0, 3
        ldr     x1, [sp,8]
        add     x0, x1, x0
        ldr     w2, [sp,44]
        mov     w1, 100
        mul     w1, w2, w1
        sxtw    x2, w1
        ldrsw   x1, [sp,40]
        add     x1, x2, x1
        lsl     x1, x1, 3
        ldr     x2, [sp,8]
        add     x1, x2, x1
        ldr     x2, [x1]
        ldr     w3, [sp,44]
        mov     w1, 100
        mul     w1, w3, w1
        sxtw    x3, w1
        ldrsw   x1, [sp,40]
        add     x1, x3, x1
        lsl     x1, x1, 3
        ldr     x3, [sp,24]
        add     x1, x3, x1
        ldr     x3, [x1]
        ldr     w4, [sp,44]
        mov     w1, 100
        mul     w1, w4, w1
        sxtw    x4, w1
        ldrsw   x1, [sp,40]
        add     x1, x4, x1
        lsl     x1, x1, 3
        ldr     x4, [sp,16]
        add     x1, x4, x1
        ldr     x1, [x1]
        fmov    d0, x3
        fmov    d1, x1
        fmul    d0, d0, d1
        fmov    x1, d0
        fmov    d0, x2
```

```
        fmov    d1, x1
        fadd    d0, d0, d1
        fmov    x1, d0
        str     x1, [x0]
        ldr     w0, [sp,36]
        add     w0, w0, 1
        str     w0, [sp,36]
.L4:
        ldr     w0, [sp, 36]
        cmp     w0, 199
        ble     .L5
        ldr     w0, [sp,40]
        add     w0, w0, 1
        str     w0, [sp,40]
.L3:
        ldr     w0, [sp,40]
        cmp     w0, 299
        ble     .L6
        ldr     w0, [sp,44]
        add     w0, w0, 1
        str     w0, [sp,44]
.L2:
        ldr     w0, [sp,44]
        cmp     w0, 99
        ble     .L7
        add     sp, sp, 48
        ret
```

指令清单 18.47 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
m:
                li      $t5, 0x13880
                move    $t4, $zero
                li      $t1, 0xC8
                li      $t3, 0x12C

loc_14:                                 # CODE XREF: m+7C
                addu    $t0, $a0, $t4
                addu    $a3, $a1, $t4
                move    $v1, $a2
                move    $t2, $zero

loc_24:                                 # CODE XREF: m+70
                mtc1    $zero, $f0
                move    $v0, $zero
                mtc1    $zero, $f1
                or      $at, $zero
                swc1    $f0, 4($v1)
                swc1    $f1, 0($v1)

loc_3C:                                 # CODE XREF: m+5C
                lwc1    $f4, 4($t0)
                lwc1    $f2, 4($a3)
                lwc1    $f5, 0($t0)
                lwc1    $f3, 0($a3)
                addiu   $v0, 1
                mul.d   $f2, $f4, $f2
                add.d   $f0, $f2
                swc1    $f0, 4($v1)
                bne     $v0, $t1, loc_3C
                swc1    $f1, 0($v1)
                addiu   $t2, 1
                addiu   $v1, 8
                addiu   $t0, 8
                bne     $t2, $t3, loc_24
```

```
        addiu   $a3, 8
        addiu   $t4, 0x320
        bne     $t4, $t5, loc_14
        addiu   $a2, 0x320
        jr      $ra
        or      $at, $zero
```

18.9.3 题目 3

请描述下列代码的作用。

通过 MSVC 2010（启用/Ox 选项）编译而得的代码如下所示。

指令清单 18.48 Optimizing MSVC 2010

```
_array$ = 8
_x$ = 12
_y$ = 16
_f      PROC
        mov     eax, DWORD PTR _x$[esp-4]
        mov     edx, DWORD PTR _y$[esp-4]
        mov     ecx, eax
        shl     ecx, 4
        sub     ecx, eax
        lea     eax, DWORD PTR [edx+ecx*8]
        mov     ecx, DWORD PTR _array$[esp-4]
        fld     QWORD PTR [ecx+eax*8]
        ret     0
_f      ENDP
```

指令清单 18.49 Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
        RSB     r1,r1,r1,LSL #4
        ADD     r0,r0,r1,LSL #6
        ADD     r1,r0,r2,LSL #3
        LDM     r1,{r0,r1}
        BX      lr
        ENDP
```

指令清单 18.50 Non-optimizing Keil 6/2013 (Thumb mode)

```
f PROC
        MOVS    r3,#0xf
        LSLS    r3,r3,#6
        MULS    r1,r3,r1
        ADDS    r0,r1,r0
        LSLS    r1,r2,#3
        ADDS    r1,r0,r1
        LDM     r1,{r0,r1}
        BX      lr
        ENDP
```

指令清单 18.51 Optimizing GCC 4.9 (ARM64)

```
f:
        sxtw    x1, w1
        add     x0, x0, x2, sxtw 3
        lsl     x2, x1, 10
        sub     x1, x2, x1, lsl 6
        ldr     d0, [x0,x1]
        ret
```

指令清单 18.52 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
        sll     $v0, $a1, 10
        sll     $a1, 6
        subu    $a1, $v0, $a1
        addu    $a1, $a0, $a1
        sll     $a2, 3
        addu    $a1, $a2
        lwc1    $f0, 4($a1)
        or      $at, $zero
        lwc1    $f1, 0($a1)
        jr      $ra
        or      $at, $zero
```

18.9.4 题目 4

请描述下列代码的作用。
请判断数组的维度。

指令清单 18.53 Optimizing MSVC 2010

```
_array$ = 8
_x$ = 12
_y$ = 16
_z$ = 20
_f      PROC
        mov     eax, DWORD PTR _x$[esp-4]
        mov     edx, DWORD PTR _y$[esp-4]
        mov     ecx, eax
        shl     ecx, 4
        sub     ecx, eax
        lea     eax, DWORD PTR [edx+ecx*4]
        mov     ecx, DWORD PTR _array$[esp-4]
        lea     eax, DWORD PTR [eax+eax*4]
        shl     eax, 4
        add     eax, DWORD PTR _z$[esp-4]
        mov     eax, DWORD PTR [ecx+eax*4]
        ret     0
_f      ENDP
```

指令清单 18.54 Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
        RSB     r1,r1,r1,LSL #4
        ADD     r1,r1,r1,LSL #2
        ADD     r0,r0,r1,LSL #8
        ADD     r1,r2,r2,LSL #2
        ADD     r0,r0,r1,LSL #6
        LDR     r0,[r0,r3,LSL #2]
        BX      lr
        ENDP
```

指令清单 18.55 Non-optimizing Keil 6/2013 (Thumb mode)

```
f PROC
        PUSH    {r4,lr}
        MOVS    r4,#0x4b
        LSLS    r4,r4,#8
        MULS    r1,r4,r1
        ADDS    r0,r1,r0
        MOVS    r1,#0xff
        ADDS    r1,r1,#0x41
        MULS    r2,r1,r2
```

```
        ADDS    r0,r0,r2
        LSLS    r1,r3,#2
        LDR     r0,[r0,r1]
        POP     {r4,pc}
        ENDP
```

指令清单 18.56 Optimizing GCC 4.9 (ARM64)

```
f:
        sxtw    x2, w2
        mov     w4, 19200
        add     x2, x2, x2, lsl 2
        smull   x1, w1, w4
        lsl     x2, x2, 4
        add     x3, x2, x3, sxtw
        add     x0, x0, x3, lsl 2
        ldr     w0, [x0,x1]
        ret
```

指令清单 18.57 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
            sll         $v0, $a1, 10
            sll         $a1, 8
            addu        $a1, $v0
            sll         $v0, $a2, 6
            sll         $a2, 4
            addu        $a2, $v0
            sll         $v0, $a1, 4
            subu        $a1, $v0, $a1
            addu        $a2, $a3
            addu        $a1, $a0, $a1
            sll         $a2, 2
            addu        $a1, $a2
            lw          $v0, 0($a1)
            jr          $ra
            or          $at, $zero
```

18.9.5 题目 5

请描述下列代码的作用。

指令清单 18.58 Optimizing MSVC 2012 /GS-

```
COMM    _tbl:DWORD:064H

tv759 = -4      ;size= 4
_main   PROC
        push    ecx
        push    ebx
        push    ebp
        push    esi
        xor     edx, edx
        push    edi
        xor     esi, esi
        xor     edi, edi
        xor     ebx, ebx
        xor     ebp, ebp
        mov     DWORD PTR tv759[esp+20], edx
        mov     eax, OFFSET _tbl+4
        npad    8 ; align next label
$LL6@main:
        lea     ecx, DWORD PTR [edx+edx]
        mov     DWORD PTR [eax+4], ecx
        mov     ecx, DWORD PTR tv759[esp+20]
        add     DWORD PTR tv759[esp+20], 3
```

```
        mov     DWORD PTR [eax+8], ecx
        lea     ecx, DWORD PTR [edx*4]
        mov     DWORD PTR [eax+12], ecx
        lea     ecx, DWORD PTR [edx*8]
        mov     DWORD PTR [eax], edx
        mov     DWORD PTR [eax+16], ebp
        mov     DWORD PTR [eax+20], ebx
        mov     DWORD PTR [eax+24], edi
        mov     DWORD PTR [eax+32], esi
        mov     DWORD PTR [eax-4], 0
        mov     DWORD PTR [eax+28], ecx
        add     eax, 40
        inc     edx
        add     ebp, 5
        add     ebx, 6
        add     edi, 7
        add     esi, 9
        cmp     eax, OFFSET _tbl+404
        jl      SHORT $LL6@main
        pop     edi
        pop     esi
        pop     ebp
        xor     eax, eax
        pop     ebx
        pop     ecx
        ret     0
_main   ENDP
```

指令清单 18.59 Non-optimizing Keil 6/2013 (ARM mode)

```
main PROC
        LDR     r12,|L0.60|
        MOV     r1,#0
|L0.8|
        ADD     r2,r1,r1,LSL #2
        MOV     r0,#0
        ADD     r2,r12,r2,LSL #3
|L0.20|
        MUL     r3,r1,r0
        STR     r3,[r2,r0,LSL #2]
        ADD     r0,r0,#1
        CMP     r0,#0xa
        BLT     |L0.20|
        ADD     r1,r1,#1
        CMP     r1,#0xa
        MOVGE   r0,#0
        BLT     |L0.8|
        BX      lr
        ENDP

|L0.60|
        DCD     ||.bss||
        AREA ||.bss||, DATA, NOINIT, ALIGN=2
tbl
        %       400
```

指令清单 18.60 Non-optimizing Keil 6/2013 (Thumb mode)

```
main PROC
        PUSH    {r4,r5,lr}
        LDR     r4,|L0.40|
        MOVS    r1,#0
|L0.6|
        MOVS    r2,#0x28
        MULS    r2,r1,r2
        MOVS    r0,#0
        ADDS    r3,r2,r4
```

```
|L0.14|
        MOVS    r2,r1
        MULS    r2,r0,r2
        LSLS    r5,r0,#2
        ADDS    r0,r0,#1
        CMP     r0,#0xa
        STR     r2,[r3,r5]
        BLT     |L0.14|
        ADDS    r1,r1,#1
        CMP     r1,#0xa
        BLT     |L0.6|
        MOVS    r0,#0
        POP     {r4,r5,pc}
        ENDP

        DCW     0x0000
|L0.40|
        DCD     ||.bss||
        AREA ||.bss||, DATA, NOINIT, ALIGN=2

tbl
        %       400
```

指令清单 18.61 Non-optimizing GCC 4.9 (ARM64)

```
        .comm   tbl,400,8
main:
        sub     sp, sp, #16
        str     wzr, [sp,12]
        b       .L2
.L5:
        str     wzr, [sp,8]
        b       .L3
.L4:
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w3, w1, w0
        adrp    x0, tbl
        add     x2, x0, :lo12:tbl
        ldrsw   x4, [sp,8]
        ldrsw   x1, [sp,12]
        mov     x0, x1
        lsl     x0, x0, 2
        add     x0, x0, x1
        lsl     x0, x0, 1
        add     x0, x0, x4
        str     w3, [x2,x0,lsl 2]
        ldr     w0, [sp,8]
        add     w0, w0, 1
        str     w0, [sp,8]
.L3:
        ldr     w0, [sp,8]
        cmp     w0, 9
        ble     .L4
        ldr     w0, [sp,12]
        add     w0, w0, 1
        str     w0, [sp,12]
.L2:
        ldr     w0, [sp,12]
        cmp     w0, 9
        ble     .L5
        mov     w0, 0
        add     sp, sp, 16
        ret
```

指令清单 18.62 Non-optimizing GCC 4.4.5 (MIPS) (IDA)

```
main:

var_18      = -0x18
var_10      = -0x10
var_C       = -0xC
var_4       = -4

            addiu       $sp, -0x18
            sw          $fp, 0x18+var_4($sp)
            move        $fp, $sp
            la          $gp, __gnu_local_gp
            sw          $gp, 0x18+var_18($sp)
            sw          $zero, 0x18+var_C($fp)
            b           loc_A0
            or          $at, $zero

loc_24:                                 # CODE XREF: main+AC
            sw          $zero, 0x18+var_10($fp)
            b           loc_7C
            or          $at, $zero

loc_30:                                 # CODE XREF: main+88
            lw          $v0, 0x18+var_C($fp)
            lw          $a0, 0x18+var_10($fp)
            lw          $a1, 0x18+var_C($fp)
            lw          $v1, 0x18+var_10($fp)
            or          $at, $zero
            mult        $a1, $v1
            mflo        $a2
            lw          $v1, (tbl & 0xFFFF)($gp)
            sll         $v0, 1
            sll         $a1, $v0, 2
            addu        $v0, $a1
            addu        $v0, $a0
            sll         $v0, 2
            addu        $v0, $v1, $v0
            sw          $a2, 0($v0)
            lw          $v0, 0x18+var_10($fp)
            or          $at, $zero
            addiu       $v0, 1
            sw          $v0, 0x18+var_10($fp)

loc_7C:                                 # CODE XREF: main+28
            lw          $v0, 0x18+var_10($fp)
            or          $at, $zero
            slti        $v0, 0xA
            bnez        $v0, loc_30
            or          $at, $zero
            lw          $v0, 0x18+var_C($fp)
            or          $at, $zero
            addiu       $v0, 1
            sw          $v0, 0x18+var_C($fp)

loc_A0:                                 # CODE XREF: main+1C
            lw          $v0, 0x18+var_C($fp)
            or          $at, $zero
            slti        $v0, 0xA
            bnez        $v0, loc_24
```

```
or          $at, $zero
move        $v0, $zero
move        $sp, $fp
lw          $fp, 0x18+var_4($sp)
addiu       $sp, 0x18
jr          $ra
or          $at, $zero

.comm tbl:0x64              # DATA XREF: main+4C
```

第19章 位 操 作

有很多程序都把输入参数的某些比特位当作标识位处理。在表面看来，使用布尔变量足以替代标志位寄存器，但是这种替代的做法并不理智。

19.1 特定位

19.1.1 x86

Win32 的 API 中有这么一段接口声明：

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS ,↙
↳FILE_ATTRIBUTE_NORMAL, NULL);
```

经过 MSVC 2010 编译，可得到如下所示的指令。

指令清单 19.1 MSVC 2010

```
push    0
push    128                             ; 00000080H
push    4
push    0
push    1
push    -1073741824                     ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

库文件 WinNT.h 对有关常量的相关位域进行了定义。

指令清单 19.2 WinNT.h

```
#define GENERIC_READ            (0x80000000L)
#define GENERIC_WRITE           (0x40000000L)
#define GENERIC_EXECUTE         (0x20000000L)
#define GENERIC_ALL             (0x10000000L)
```

毫无疑问，在 API 声明里，CreatFile()函数的第二个参数为“GENERIC_READ | GENERIC_WRITE” = 0x80000000 | 0x40000000 = 0xC0000000。

CreatFile()函数如何检测标志位呢？

我们可以在 Windows XP SP3 x86 的 KERNEL32.DLL 中，找到 CreateFileW()函数的相关代码。

指令清单 19.3 KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429  test        byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42   Dmov        [ebp+var_8], 1
.text:7C83D434  jz          short loc_7C83D417
.text:7C83D436  jmp         loc_7C810817
```

比较讲究的是 TEST 指令。在这里，它没有整个使用第二个参数，而是仅仅拿它数权最高的头一个字节（ebp+dwDesiredAccess+3）和 0x40（对应 GENERIC_WRITE 标志）进行与运算。

TEST 与 AND 指令的唯一区别是前者不保存运算结果，CMP 指令和 SUB 指令之间也存在这种差别（请

参见本书 7.3.1 节)。

即，上述可执行的程序源代码逻辑是：

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

如果此处的与运算(&)的结果是 1，则会清零 ZF 标志位，而不会触发 JZ 跳转。只有在 dwDesiredAccess 变量与 0x40000000 的计算结果为 0 的情况下，条件转移指令才会被触发；即与运算（&）的结果是零、ZF 标志位为 1，从而触发条件转移指令。

如果使用 Linux 的 GCC 4.4.1 编译下面的代码：

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;
        handle=open ("file", O_RDWR | O_CREAT);
};
```

得到的汇编指令会如下所示。

指令清单 19.4　GCC 4.4.1

```
            public main
main        proc near

var_20      = dword ptr -20h
var_1C      = dword ptr -1Ch
var_4       = dword ptr -4

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 20h
            mov     [esp+20h+var_1C], 42h
            mov     [esp+20h+var_20], offset aFile ; "file"
            call    _open
            mov     [esp+20h+var_4], eax
            leave
            retn
main        endp
```

在库文件 libc.so.6 里，open()函数只是调用 syscall 而已。

指令清单 19.5　open() (libc.so.6)

```
.text:000BE69B              mov  edx, [esp+4+mode] ; mode
.text:000BE69F              mov  ecx, [esp+4+flags] ; flags
.text:000BE6A3              mov  ebx, [esp+4+filename] ; filename
.text:000BE6A7              mov  eax, 5
.text:000BE6AC              int  80h      ; LINUX - sys_open
```

可见，在执行 Open()函数时，Linux 内核会检测某些标识位。

当然，我们可以下载 Glibc 和 Linux 内核源代码来一窥究竟。但是本文旨在粗略地介绍原理，就不逐行地解释代码了。

在 Linux 2.6 中，当程序通过 syscall 调用 sys_open 的时候，它调用的函数实际上是内核函数 do_sys_open()。在执行函数 do_sys_open()的时候，系统又会调用 do_filp_open() 函数。有兴趣的读者，可以在 Linux 内核源代码里的 fs/namei.c 里找到这部分内容。

编译器不仅会使用栈来传递参数，它还会使用寄存器传递部分参数。编译器最常采用的函数调用约定是 fastcall（参见 6.4.3 节)。这个规范优先使用寄存器来传递参数。这使得 CPU 不必每次都要访问内存里

的数据栈，大大提升了程序的运行效率。此外，我们可以通过 GCC 的 regparm 选项[①]，设置传递参数的寄存器数量。

在 Linux 2.6 内核的编译选项中，设定了寄存器选项“-mregparm=3”。[②]

这个选项决定，编译器首先通过 EAX、EDX 和 ECX 寄存器传递函数所需的头 3 个参数，再通过栈传递其余的参数。当然，如果参数的总数不足 3 个，它只会用到部分寄存器。

在 Ubuntu 里下载 Linux Kernel 2.6.31，并用“make vmlinux”指令编译，然后使用 IDA 打开程序，寻找函数 do_filp_open()。这个函数的开头部分大体是这样的。

指令清单 19.6 do_filp_open() (Linux kernel 2.6.31)

```
do_filp_open    proc near
......
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1th arg)
                mov     [ebp+var_7C], edx ; pathname (2th arg)
                mov     [ebp+var_78], ecx ; open_flag (3th arg)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

函数序言部分，GCC 就把 3 个寄存器里的参数值存储在栈里。若非如此，那么后续程序可能出现寄存器不够使用的资源紧张问题。

我们再来看下 do_filp_open()函数的另外一段代码。

指令清单 19.7 do_filp_open() (Linux kernel 2.6.31)

```
loc_C01EF6B4:                           ; CODE XREF: do_filp_open+4F
                test    bl, 40h         ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

汇编宏 O_CREAT 的值就是 0x40h。TEST 指令检查 open_flag 里是否存在 0x40 的标志位，如果这个位是 1，则会触发下一条 JNZ 指令。

19.1.2 ARM

在 Linux Kernel 3.8.0 平台的系统里，O_CREAT 的检查操作有所不同。

指令清单 19.8 Linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
```

① http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm。

② 参见：http://kernelnewbies.org/Linux_2_6_20#head- 042c62f290834eb1fe0a1942bbf5bb9a4accbc8f 以及源代码文件。

```
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}
static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{ ...
        if (!(open_flag & O_CREAT)) {
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

使用 IDA 打开 ARM 模式的内核程序，可以看到 do_last()函数的代码如下所示。

指令清单 19.9　do_last() (vmlinux)

```
...
.text:C0169EA8        MOV       R9, R3  ; R3 - (4th argument) open_flag
...
.text:C0169ED4        LDR       R6, [R9] ; R6 - open_flag
...
.text:C0169F68        TST       R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C        BNE       loc_C016A128
.text:C0169F70        LDR       R2, [R4,#0x10]
.text:C0169F74        ADD       R12, R4, #8
.text:C0169F78        LDR       R3, [R4,#0xC]
.text:C0169F7C        MOV       R0, R4
.text:C0169F80        STR       R12, [R11,#var_50]
.text:C0169F84        LDRB      R3, [R2,R3]
.text:C0169F88        MOV       R2, R8
.text:C0169F8C        CMP       R3, #0
.text:C0169F90        ORRNE     R1, R1, #3
.text:C0169F94        STRNE     R1, [R4,#0x24]
.text:C0169F98        ANDS      R3, R6, #0x200000
.text:C0169F9C        MOV       R1, R12
.text:C0169FA0        LDRNE     R3, [R4,#0x24]
.text:C0169FA4        ANDNE     R3, R3, #1
.text:C0169FA8        EORNE     R3, R3, #1
.text:C0169FAC        STR       R3, [R11,#var_54]
.text:C0169FB0        SUB       R3, R11, #-var_38
.text:C0169FB4        BL        lookup_fast
...
.text:C016A128 loc_C016A128                       ; CODE XREF: do_last.isra.14+DC
.text:C016A128        MOV             R0, R4
.text:C016A12C        BL              complete_walk
...
```

TST 指令的功能与 x86 平台的 TEST 指令相同。

显而易见，程序在某一条件下会调用 lookup_fast()函数，而在另一条件下会调用 complete_walk()函数。

这种行为符合 do_last()函数的源代码。

这里的汇编宏 O_CREAT 也等于 0x40。

19.2 设置/清除特定位

本节将会围绕下面这个程序进行讨论：

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;
    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);
    return rt;
};

int main()
{
    f(0x12340678);
};
```

19.2.1 x86

Non-optimizing MSVC

经过 MSVC 2010（未启用任何优化选项）编译上述程序，可得到如下所示的指令。

指令清单 19.10 MSVC 2010

```
_rt$ = -4               ; size = 4
_a$ = 8                 ; size = 4

_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384               ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f  ENDP
```

OR 指令就是逐位进行或运算的指令，可用来将指定位置 1。

AND 指令的作用是重置 bit 位。效果上说，如果立即数的某个 bit 位为 1，AND 指令则会保留寄存器里的这个 bit 位；如果立即数的某个 bit 位为 0，AND 指令就会把寄存器里这个 bit 的值设置为 0。按照 bit

位掩码（bitmask）的方式理解，就会非常容易地记住这些指令的作用。

OllyDbg

用 OllyDbg 打开这个程序，可以清楚地看到常量的各个位：

0x200 的二进制数是 00000000000000000001000000000，第 10 位是 1。

0x200 的逻辑非运算结果是 0xFFFFFDFF(11111111111111111110111111111)。

0x4000 的第 15 位是 1。

如图 19.1 所示，输入变量为 0x12340678 (10010001101000000011001111000)。

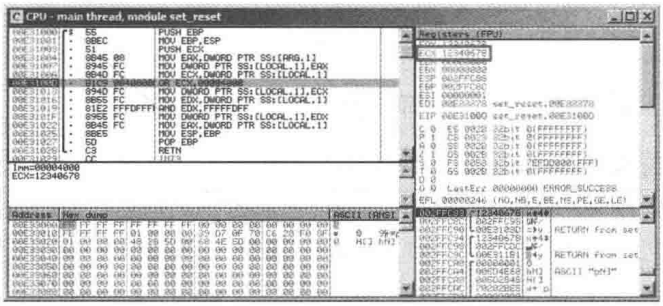


图 19.1　OllyDbg：ECX 寄存器载入输入变量

执行 OR 指令的情形如图 19.2 所示。

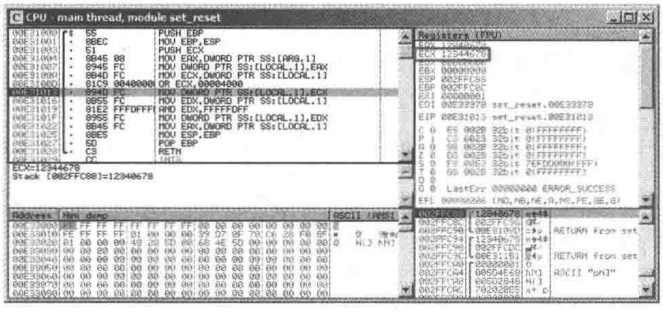


图 19.2　OllyDbg：执行 OR 指令

在执行 OR 指令的时候，第 15 位被设为 1。

因为没有进行优化编译，所以程序中有些无用指令。如图 19.3 所示，它又加载了一次这个值。

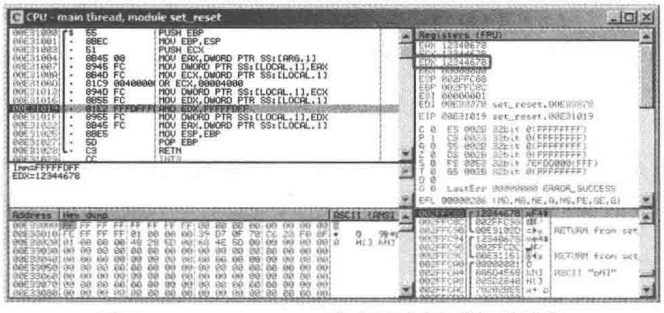


图 19.3　OllyDbg：EDX 寄存器再次加载运算结果

然后进行了 AND 与操作，如图 19.4 所示。

这时寄存器的第 10 位被清零。换句话说，第 10 位之外的所有位都被保留了。最终运算结果是 0x12344478，即二进制的 10010001101000100010001111000。

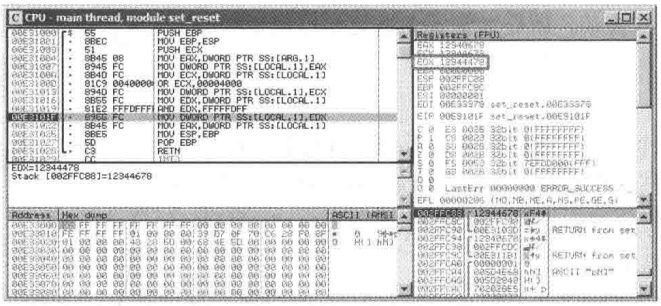


图 19.4 OllyDbg：执行与操作

Optimizing MSVC

如果开启了 MSVC 的优化选项/Ox，生成的代码就会精简很多。

指令清单 19.11 Optimizing MSVC

```
_a$ = 8                 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    and    eax, -513        ; fffffdffH
    or     eax, 16384       ; 00004000H
    ret    0
_f    ENDP
```

Non-optimizing GCC

如果关闭 GCC 4.4.1 的优化选项，则会生成下列代码。

指令清单 19.12 Non-optimizing GCC

```
            public f
f           proc near

var_4       = dword ptr -4
arg_0       = dword ptr  8

            push    ebp
            mov     ebp, esp
            sub     esp, 10h
            mov     eax, [ebp+arg_0]
            mov     [ebp+var_4], eax
            or      [ebp+var_4], 4000h
            and     [ebp+var_4], 0FFFFFDFFh
            mov     eax, [ebp+var_4]
            leave
            retn
f           endp
```

虽然 GCC 的非优化编译结果中存在冗余指令，但是这个程序仍然比 MSVC 的非优化编译程序要短。接下来启用 GCC 的-O3 选项进行优化编译。

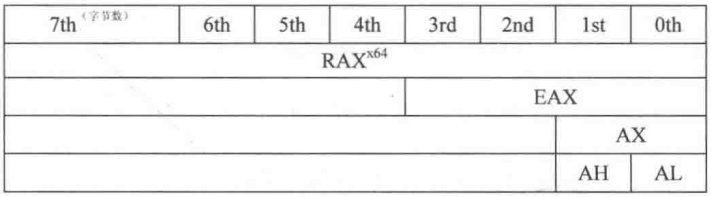
Optimizing GCC

指令清单 19.13 Optimizing GCC

```
            public f
f           proc near

arg_0       = dword ptr  8
```

```
        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0]
        pop     ebp
        or      ah, 40h
        and     ah, 0FDh
        retn
f       endp
```

它比 MSVC 优化编译生成的程序还要短。值得注意的是，它直接使用了 EAX 寄存器的部分空间——EAX 寄存器的第 8 到第 15 位（第 1 个字节），即 AH 寄存器。



在早期的 16 位 8086 CPU 里，AX 寄存器分为 2 个 8 位寄存器——即 AL/AH 寄存器。而 80386 差不多把所有的寄存器都扩展为 32 位，形成了 EAX 寄存器。但是出于兼容性的考虑，后续的 CPU 保留了 AX/AH/AL 寄存器的命名空间。

世上先有的 16 位 8086 CPU，后有的 32 位 x86 CPU。在早期 16 位 CPU 上运行短程序，其 opcode 比在 32 位 CPU 上运行的 opcode 要短。所以，操作 AH 寄存器的“or ah，40h”的指令只占用了 3 字节的 opcode。虽然说似乎“or eax,04000h”这样的指令更合乎情理，但是那样就会占用 5 字节的 opcode；如果第一个操作符不是 EAX 寄存器，同等的指令甚至会使用 6 字节的 opcode。

Optimizing GCC and regparm

如果同时启用优化选项“-O3”和寄存器分配选项“regparm＝3”，生成的指令会更为简短。

指令清单 19.14　Optimizing GCC

```
        public f
f       proc near
        push    ebp
        or      ah, 40h
        mov     ebp, esp
        and     ah, 0FDh
        pop     ebp
        retn
f       endp
```

第一个参数原本就在 EAX 寄存器里（fastcall 规范），所以可直接使用 EAX 寄存器。函数引言的“push ebp / mov ebp,esp”指令，以及函数尾声的“pop ebp”完全可以省略；但是 GCC 还做不到这种程度的智能优化。这种代码完全可以作为内嵌函数（参见第 43 章）来使用。

19.2.2　ARM + Optimizing Keil 6/2013 (ARM mode)

指令清单 19.15　Optimizing Keil 6/2013 (ARM mode)

```
02 0C C0 E3     BIC     R0, R0, #0x200
01 09 80 E3     ORR     R0, R0, #0x4000
1E FF 2F E1     BX      LR
```

BIC 是逐位清除的“（反码）逻辑与”指令，它将第一个操作数与第二个操作数的反码（非求非运算的结果，进行逻辑与运算，比）x86 指令集里“逻辑与”指令多作了一次求非运算，ORR 是“逻辑或指令”，与 x86“逻辑或”的作用相同。

19.2.3 ARM + Optimizing Keil 6/2013 (Thumb mode)

指令清单 19.16 Optimizing Keil 6/2013 (Thumb mode)

```
01 21 89 03     MOVS    R1, 0x4000
08 43ORRS       R0,     R1
49 11ASRS       R1,     R1, #5   ; generate 0x200 and place to R1
88 43           BICS    R0, R1
70 47           BX      LR
```

编译器借助了算术右移指令，由 0x4000>>5 生成 0x200。与直接对寄存器进行立即数赋值（0x200）相比，这种算术指令的 opcode 比较短。

所以，后面的 ASRS（算术右移）指令通过计算 0x4000>>5 得到 0x200。

19.2.4 ARM + Optimizing Xcode (LLVM) + ARM mode

指令清单 19.17 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
42 0C C0 E3     BIC     R0, R0, #0x4200
01 09 80 E3     ORR     R0, R0, #0x4000
1E FF 2F E1     BX      LR
```

这段代码由 LLVM 编译而得。其源代码大体应该是：

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

它确实足以完成所需功能。但是哪里来的 0x4200？或许，LLVM 优化功能对原有指令进行了再加工，或许这是负负得正。总之这段代码没有质量问题。

有时候编译器确实会做这种类型的代码处理，详细情况请参见第 91 章。

使用 Xcode 4.6.3 进行优化编译而生成的代码也非常相似，本文不再介绍。

19.2.5 ARM：BIC 指令详解

把本章开头的源代码略加修改：

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

用 Keil 5.03 进行优化编译[①]，可得 ARM 模式的指令如下：

```
f PROC
        BIC     r0,r0,#0x1000
        BIC     r0,r0,#0x234
        BX      lr
        ENDP
```

上述程序使用了两条 BIC 指令，即它需要通过两次处理才能清除 0x1234 对应的比特位。这是因为立即数 0x1234 无法和 BIC 指令封装在同一条指令里。所以编译器进行了等效处理，通过两条指令实现清空 0x1000 和 0x234 相应的比特位。

19.2.6 ARM64: Optimizing GCC(Linaro) 4.9

面向 ARM64 的 GCC 编译器，可直接使用 AND 指令、而不用 BIC 指令。

① 确切地说，使用的编译器是 LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3。

指令清单 19.18 Optimizing GCC (Linaro) 4.9

```
f:
        addw0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        orrw0, w0, 16384   ; 0x4000
        ret
```

19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9

关闭优化选项之后，GCC 生成的代码中会加载无用指令，但是功能相同。

指令清单 19.19 Non-optimizing GCC (Linaro) 4.9

```
f:
        sub     sp, sp, #32
        str     w0, [sp,12]
        ldr     w0, [sp,12]
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        orr     w0, w0, 16384          ; 0x4000
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        and     w0, w0, -513           ; 0xFFFFFFFFFFFFFDFF
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        add     sp, sp, 32
        ret
```

19.2.8 MIPS

指令清单 19.20 Optimizing GCC 4.4.5 (IDA)

```
f:
; $a0=a
                ori     $a0, 0x4000
; $a0=a|0x4000
                li      $v0, 0xFFFFFDFF
                jr      $ra
                and     $v0, $a0, $v0
; at finish: $v0 = $a0&$v0 = a|0x4000 & 0xFFFFFDFF
```

ORI 无疑还是 OR 指令。指令中的“I”意味着它使用机械码存储数值。

但是后面的 AND 指令可就不能是 ANDI 指令了。这是因为它涉及的立即数 0xFFFFFDFF 太长，无法连同操作指令封装在同一条指令里。所以编译器把这个数赋值给$V0 寄存器，然后让它与其他寄存器进行 AND/与运算。

19.3 位移

C/C++的位移运算符是<<和>>。

x86 指令集里有对应的左移 SHL 和右移 SHR 指令。

位移操作通常用来实现与“2 的 n 次幂”有关的乘法和除法运算。有关介绍请参见本书的 16.1.2 节和 16.2.1 节。

位移指令可用来对特定位进行取值或隔离，用途十分广泛。

19.4 在 FPU 上设置特定位

FPU 存储数据的 IEEE 754 格式如下：

（S—sign）

最高数权位 MSB 为符号位。是否可能在不使用 FPU 指令的前提下变更浮点数的符号呢？

```
#include <stdio.h>

float my_abs (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
        return *(float*)&tmp;
};

float set_sign (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
        return *(float*)&tmp;
};
float negate (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
        return *(float*)&tmp;
};

int main()
{
        printf ("my_abs():\n");
        printf ("%f\n", my_abs (123.456));
        printf ("%f\n", my_abs (-456.123));
        printf ("set_sign():\n");
        printf ("%f\n", set_sign (123.456));
        printf ("%f\n", set_sign (-456.123));
        printf ("negate():\n");
        printf ("%f\n", negate (123.456));
        printf ("%f\n", negate (-456.123));
};
```

使用位操作 C/C++指令之后，我们不必在 CPU 和 FPU 的寄存器之间传递数据、再进行数学运算了。本节列举了三个位操作函数：清除 MSB 符号位的 my_abs()函数、设置符号位的 set_sign()函数及求负函数 negate()。

19.4.1 XOR 操作详解

XOR 指令常用于使一个操作数的某些位取反的场合。若操作数 A 的某个位为 1，那么 XOR 指令将对另一个数的相应位求非。

输入 A	输入 B	输出
0	0	0
0	1	1
1	0	1
1	1	0

若某个参数为零，XOR 则不会进行任何操作。这种指令也就是所谓的空操作。这是 XOR 指令非常重要的属性，建议记住它。

19.4.2 x86

编译生成的指令简单易懂。

指令清单 19.21 Optimizing MSVC 2012

```
_tmp$ = 8
_i$ = 8
_my_abs PROC
```

```
        and     DWORD PTR _i$[esp-4], 2147483647    ; 7fffffffH
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
        or      DWORD PTR _i$[esp-4], -2147483648   ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
        xor     DWORD PTR _i$[esp-4], -2147483648   ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_negate ENDP
```

函数从栈中提取一个浮点类型的数据，但是把它当作整数类型数据进行处理。

AND 和 OR 设置相应的比特位，而 XOR 用来设置相反的符号位。

因为要把浮点数还给 FPU，所以最后把修改后的处理结果保存到 ST0 寄存器。

接下来，使用 64 位的 MSVC 2012 进行优化编译，得到的代码如下所示。

指令清单 19.22 Optimizing MSVC 2012 x64

```
tmp$ = 8
i$ = 8
my_abs  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btr     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
my_abs  ENDP
_TEXT   ENDS

tmp$ = 8
i$ = 8
set_sign PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        bts     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btc     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
negate  END
```

输入值会经 XMM0 寄存器存储到局部数据栈，然后通过 BTR、BTS、BTC 等指令进行处理。

这些指令用于重置（BTR）、置位（BTS）和翻转（BTC）特定比特位。如果从 0 开始数的话，浮点数

的第 31 位才是 MSB。

最终，运算结果被复制到 XMM0 寄存器。在 Win64 系统中，浮点型返回值要保存在这个寄存器里。

19.4.3 MIPS

面向 MIPS 的 GCC 4.4.5 生成的程序几乎没什么区别。

指令清单 19.23 Optimizing GCC 4.4.5 (IDA)

```
my_abs:
; move from coprocessor 1:
                mfc1    $v1, $f12
                li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; do AND:
                and     $v0, $v1
; move to coprocessor 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot

set_sign:
; move from coprocessor 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; do OR:
                or      $v0, $v1, $v0
; move to coprocessor 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot
negate:
; move from coprocessor 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; do XOR
                xor     $v0, $v1, $v0
; move to coprocessor 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot
```

因为 LUI 在传递高 16 位数据时会清除寄存器的低 16 位，所以单条 LUI 指令就可以完成 0x80000000 的赋值，无需再用 ORI 指令。

19.4.4 ARM

Optimizing Keil 6/2013 (ARM mode)

指令清单 19.24 Optimizing Keil 6/2013 (ARM mode)

```
my_abs PROC
; clear bit:
        BIC     r0,r0,#0x80000000
        BX      lr
        ENDP

set_sign PROC
```

```
; do OR:
        ORR     r0,r0,#0x80000000
        BX      lr
        ENDP

negate PROC
; do XOR:
        EOR     r0,r0,#0x80000000
        BX      lr
        ENDP
```

ARM 模式的程序可以使用 BIC 指令直接清除指定的比特位。上述程序中的 EOR 指令是 ARM 模式下进行 XOR（异或）运算的指令。

Optimizing Keil 6/2013 (Thumb mode)

指令清单 19.25 Optimizing Keil 6/2013 (Thumb mode)

```
my_abs PROC
        LSLS    r0,r0,#1
; r0=i<<1
        LSRS    r0,r0,#1
; r0=(i<<1)>>1
        BX      lr
        ENDP

set_sign PROC
        MOVS    r1,#1
; r1=1
        LSLS    r1,r1,#31
; r1=1<<31=0x80000000
        ORRS    r0,r0,r1
; r0=r0 | 0x80000000
        BX      lr
        ENDP

negate PROC
        MOVS    r1,#1
; r1=1
        LSLS    r1,r1,#31
; r1=1<<31=0x80000000
        EORS    r0,r0,r1
; r0=r0 ^ 0x80000000
        BX  lr
        ENDP
```

ARM 平台的 Thumb 模式指令都是 16 位定长指令。因为单条指令无法封装太大的数据，所以编译器使用了 MOVS/LSLS 指令对传递常量 0x80000000。这种赋值操作基于数学机制：1<<31=0x80000000。

然而 my_abs 的指令令人费解，它做的运算是(1<<1)>>1。虽然看上去是画蛇添足，但是在进行“输入变量<<1”的运算时，最高数权位被舍弃了。继而在执行“运算结果>>1”的语句时，根据右移运算规则最高数权位变为零，而其他各位数值回归原位。借助 LSLS/LSRS 指令对，该函数清除了最高数权位 MSB。

Optimizing GCC 4.6.3 (Raspberry Pi, ARM mode)

指令清单 19.26 Optimizing GCC 4.6.3 for Raspberry Pi (ARM mode)

```
my_abs
; copy from S0 to R2:
                FMRS        R2, S0
; clear bit:
                BIC         R3, R2, #0x80000000
; copy from R3 to S0:
                FMSR        S0, R3
                BX          LR

set_sign
```

```
; copy from S0 to R2:
                FMRS        R2, S0
; do OR:
                ORR         R3, R2, #0x80000000
; copy from R3 to S0:
                FMSR        S0, R3
                BX          LR

negate
; copy from S0 to R2:
                FMRS        R2, S0
; do ADD:
                ADD         R3, R2, #0x80000000
; copy from R3 to S0:
                FMSR        S0, R3
                BX          LR
```

在 QEMU 的仿真环境下启动 Raspberry Pi Linux 即可模拟 ARM FPU。所以上述程序使用 S-字头寄存器存储浮点数，而不会使用 R-字头寄存器。

FMRS 指令用于在通用寄存器 GPR 和 FPU 之间交换数据。

上述程序中的 my_abs()函数和 set_sign()函数都中规中矩。但是 negate()函数在应该出现 XOR 指令的地方使用了 ADD 指令。

其实“ADD register, 0x80000000”和“XOR register, 0x80000000”是等效指令，只是不特别直观而已。整个函数用于变更最高数权位的值。在数学上，1000 与任意三位数相加，运算结果的最后 3 位仍然和被加数的最后三位相等。同理，1234567+10000=1244567，和的最后四位与第一个数的最后四位等值。以二进制数的角度来看，0x80000000 就是 10000000000000000000000000000000，只有最高位是 1。因此，当 0x80000000 与任意数值相加时，运算结果的最后 31 位还是被加数的最后 31 位，只是最高数权位会发生变化。再看 MSB：1+0=1，1+1=0（高于 32 位的数值被舍弃了）。因此，对于这个特定加数来说，ADD 指令和 XOR 指令可以互换。虽然替换的必要性不甚明朗，但是整个程序还是没有问题的。

19.5 位校验

我们介绍一个测算输入变量的 2 进制数里有多少个 1 的函数。这种函数叫作“population count/点数”函数。在支持 SSE4 的 x86 CPU 的指令集里，甚至有直接对应的 POPCNT 指令。

```
#include <stdio.h>

#define IS_SET(flag, bit)  ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

for (i=0; i<32; i++)
    if (IS_SET (a, 1<<i))
        rt++;

    return rt;
};
int main()
{
    f(0x12345678); // test
};
```

在这个循环里，循环计数变量 *i* 的取值范围是 0～31。因此“1<<*i*”表达式的取值范围是 0 到 0x80000000，用人类语言解释，它的含义就是“将数字 1 左移 *n* 位”。“1<<*i*”将会使“1”逐一遍历 32 位数字的每一位，同时将其他位置零。

这段程序里，1<<*i* 可能产生的值有：

C/C++表达式	2 的指数	十进制数	十六进制数
1<<0	2^0	1	1
1<<1	2^1	2	2
1<<2	2^2	4	4
1<<3	2^3	8	8
1<<4	2^4	16	0x10
1<<5	2^5	32	0x20
1<<6	2^6	64	0x40
1<<7	2^7	128	0x80
1<<8	2^8	256	0x100
1<<9	2^9	512	0x200
1<<10	2^{10}	1024	0x400
1<<11	2^{11}	2048	0x800
1<<12	2^{12}	4096	0x1000
1<<13	2^{13}	8192	0x2000
1<<14	2^{14}	16384	0x4000
1<<15	2^{15}	32768	0x8000
1<<16	2^{16}	65536	0x10000
1<<17	2^{17}	131072	0x20000
1<<18	2^{18}	262144	0x40000
1<<19	2^{19}	524288	0x80000
1<<20	2^{20}	1048576	0x100000
1<<21	2^{21}	2097152	0x200000
1<<22	2^{22}	4194304	0x400000
1<<23	2^{23}	8388608	0x800000
1<<24	2^{24}	16777216	0x1000000
1<<25	2^{25}	33554432	0x2000000
1<<26	2^{26}	67108864	0x4000000
1<<27	2^{27}	134217728	0x8000000
1<<28	2^{28}	268435456	0x10000000
1<<29	2^{29}	536870912	0x20000000
1<<30	2^{30}	1073741824	0x40000000
1<<31	2^{31}	2147483648	0x80000000

逆向工程的实际工作中经常出现这些常数（bit mask）。逆向工程人员应当常备这个表格。虽然 10 进制常数很难记忆，但是背下十六进制数并不太困难。

它们经常被用作各种标志位。例如，在 Apache 2.4.6 的源代码中，ssl_private.h 就用到了其中的部分常量：

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE            (0)
#define SSL_OPT_RELSET          (1<<0)
#define SSL_OPT_STDENVVARS      (1<<1)
#define SSL_OPT_EXPORTCERTDATA  (1<<3)
#define SSL_OPT_FAKEBASICAUTH   (1<<4)
#define SSL_OPT_STRICTREQUIRE   (1<<5)
#define SSL_OPT_OPTRENEGOTIATE  (1<<6)
#define SSL_OPT_LEGACYDNFORMAT  (1<<7)
```

现在言归正传，看看“点数”程序。

宏 IS_SET 的作用是检测 flag 参数（变量 *a*）的第 *n* 位是否为 1。

如果变量 *a* 对应的比特位不是 1，则宏 IS_SET 返回 0；如果变量 *a* 对应的比特位是 1，则返回 bit mask。因为触发 if()then{}语句的条件是“条件表达式不是零”，所以哪怕条件返回值是 123456，它都会照样执行 then 模块的指令。

19.5.1 x86

MSVC

使用 MSVC2010 编译上述程序，得到的代码如下所示。

指令清单 19.27 MSVC 2010

```
_rt$ = -8               ;size=4
_i$ =-4                 ;size=4
_a$ =8                  ;size=4

_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]    ; 递增
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32     ; 00000020H
    jge    SHORT $LN2@f               ; 循环结束?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                    ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f               ; result of AND instruction was 0?
                                      ; then skip next instructions
    mov    eax, DWORD PTR _rt$[ebp]   ; no, not zero
    add    eax, 1                     ; increment rt
    mov    DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp    SHORT $LN3@f
    $LN2@f:
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

OllyDbg

为了方便使用 OllyDbg 进行演示，我们设置变量 *a* 的值为 0x12345678。

在 *i*=1 时的情况如图 19.5 所示。

图 19.5 OllyDbg：*i*=1 时，*i* 被加载到 ECX 寄存器

此刻 EDX 的值为 1。

执行 SHL 指令的情况，如图 19.6 所示。

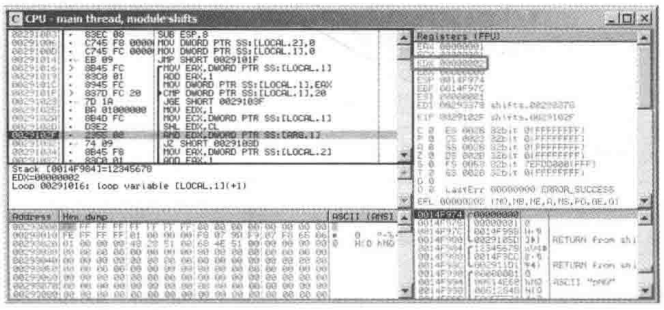


图 19.6　OllyDbg：i = 1,EDX=1<<1 = 2

EDX 寄存器的值是 bit mask ,1<<1=2。

在执行 AND 指令后，ZF＝1。就是说，输入变量 0x12345678 和数字 2 的 AND 计算结果是 0，如图 19.7 所示。

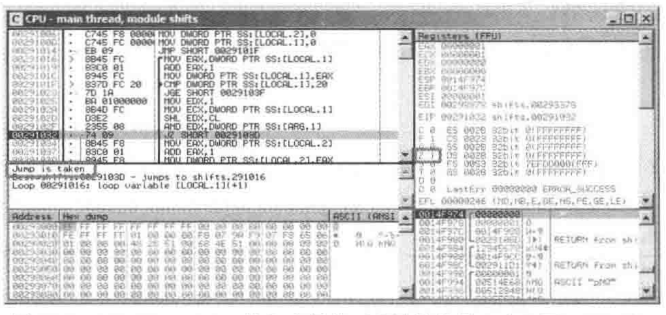


图 19.7　OllyDbg：i=1，输入变量的对应比特位是 1 么？否（ZF=1）

这意味着输入变量的对应比特位是 0。这种情况下，程序通过 JZ 指令进行跳转，绕过了操作点数计数器 rt 的递增指令。

继续执行程序，看看 i=4 的情况。如图 19.8 所示，此时要执行 SHL 指令。

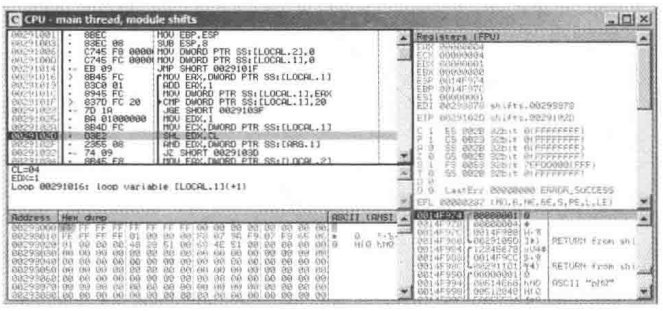


图 19.8　OllyDbg：i=4, ECX 加载 i 的值

此刻 EDX 寄存器的值是 0x10，它是 1<<4 的结果，如图 19.9 所示。

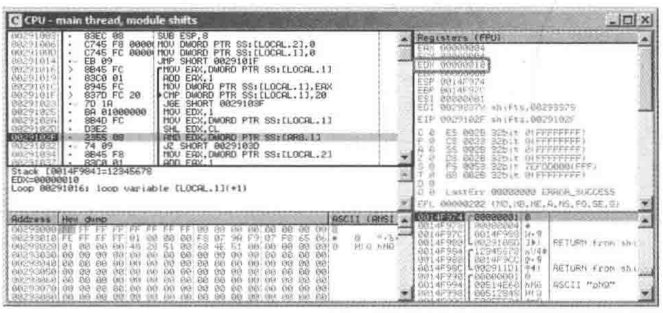


图 19.9　OllyDbg：i = 4, EDX =1≤4 = 0x10

当 i=4 时，对应的 bit mask 存在。

执行 AND 指令的情形如图 19.10 所示。

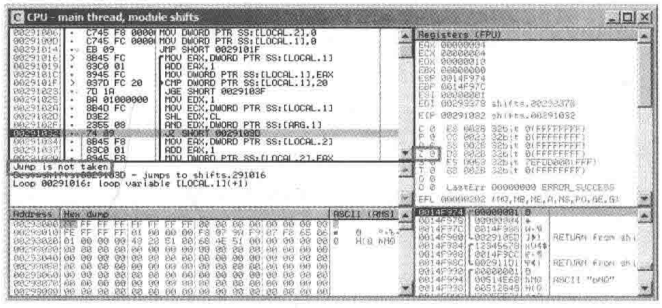


图 19.10 OllyDbg：i=4，输入变量的对应位是否为 1？是（ZF=0）

因为对应 bit 位是 1，所以 ZF=0。确实，0x12345678 & 0x10 = 0x10。这种情况下，将不会触发 jz 转移指令，点数计数器 rt 的值递增。

最终，函数的返回值是 13。输入变量 0x12345678 的二进制数里有 13 个 1。

GCC

使用 GCC 4.4.1 编译上述程序，可得如下所示的指令。

指令清单 19.28 GCC 4.4.1

```
            public f
f           proc near

rt          = dword ptr -0Ch
i           = dword ptr -8
arg_0       = dword ptr  8

            push    ebp
            mov     ebp, esp
            push    ebx
            sub     esp, 10h
            mov     [ebp+rt], 0
            mov     [ebp+i], 0
            jmp     short loc_80483EF
loc_80483D0:
            mov     eax, [ebp+i]
            mov     edx, 1
            mov     ebx, edx
            mov     ecx, eax
            shl     ebx, cl
            mov     eax, ebx
            and     eax, [ebp+arg_0]
            test    eax, eax
            jz      short loc_80483EB
            add     [ebp+rt], 1
loc_80483EB:
            add     [ebp+i], 1
loc_80483EF:
            cmp     [ebp+i], 1Fh
            jle     short loc_80483D0
            mov     eax, [ebp+rt]
            add     esp, 10h
            pop     ebx
            pop     ebp
            retn
f           endp
```

19.5.2　x64

为了进行演示，本节对源程序略加修改，将数据类型扩展为 64 位：

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)   ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

Non-optimizing GCC 4.8.2

指令清单 19.29　Non-optimizing GCC 4.8.2

```
f:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi    ; a
        mov     DWORD PTR [rbp-12], 0      ; rt=0
        mov     QWORD PTR [rbp-8], 0       ; i=0
        jmp     .L2
.L4:
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; the last bit is zero?
; skip the next ADD instruction, if it was so.
        je      .L3
        add     DWORD PTR [rbp-12], 1      ; rt++
.L3:
        add     QWORD PTR [rbp-8], 1       ; i++
.L2:
        cmp     QWORD PTR [rbp-8], 63      ; i< 63?
        jbe     .L4                        ; jump to the loop boday begin, if so
        mov     eax, DWORD PTR [rbp-12]    ; return rt
        pop     rbp
        ret
```

Optimizing GCC 4.8.2

指令清单 19.30　Optimizing GCC 4.8.2

```
1 f:
2       xor     eax, eax            ; rt variable will be in EAX register
3       xor     ecx, ecx            ; i variable will be in ECX register
4 .L3:
```

```
 5      mov     rsi, rdi                ; load input value
 6      lea     edx, [rax+1]            ; EDX=EAX+1
 7 ; EDX here is a "new version of rt", which will be written into rt variable, if the last bit is 1
 8      shr     rsi, cl                 ; RSI=RSI>>CL
 9      and     esi, 1                  ; ESI=ESI&1
10 ; the last bit is 1? If so, write "new version of rt" into EAX
11      cmovne  eax, edx
12      add     rcx, 1                  ; RCX++
13      cmp     rcx, 64
14      jne     .L3
15      rep ret                         ; AKA fatret
```

这段程序简洁又兼具特色。在本节前面介绍过的各个例子里，程序都是先比较特定位、然后在递增“rt”的值。但是这段程序的顺序有所不同，它先递增 rt 再把新的值写到 EDX 寄存器。这样一来，如果最后的比特位是 1，那么程序将使用 CMOVNE[①]指令（等同于 CMOVNZ[②]）把 EDX 寄存器（rt 的候选值）传递给 EAX 寄存器（当前的 rt 值，也是函数的返回值），从而完成 rt 的更新。所以，无论输入值是什么，每次迭代结束之后，循环控制变量的值都要进行递增，它肯定会被递增 64 次。

因为它只含有一个条件转移指令（在循环的尾部），所以这种编译方法独具优势。如果编译的方式过于机械化，那么程序要在递增 rt 和循环尾部进行两次条件转移。现在的 CPU 都具有分支预测的功能（请参加本书 33.1 节）。在性能方面，本例这类程序的效率更高。

最后一则指令是 REP RET（opcode 为 F3 C3）。它就是 MSVC 里的 FATRET。这个指令是由 RET 衍生出来的向优化指令。AMD 建议：如果 RET 指令的前一条指令是条件转移指令，那么在函数最后的返回指令最好使用 REP RET 指令。（请参见 AMD13b，p15）[③]

Optimizing MSVC 2010

指令清单 19.31 MSVC 2010

```
a$ = 8
f       PROC
; RCX = input value
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f:
        test    rdx, rcx
; there are no such bit in input value?
; skip the next INC instruction then.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

本例中，编译器用 ROL 指令替代了 SHL 指令。ROL 的确切作用是“rotate left”，SHL 的含义则是“shift left”。不过在本例中，它们的效果相同。

有关旋转指令的详细介绍，请参见附录 A.6.3。

R8 的值将从 64 逐渐递减为 0，它的变化过程和变量 *i* 完全相反。

在程序的执行过程中，寄存器的关系如下图所示。

① Conditional MOVe if Not Equal。
② Conditional MOVe if Not Zero。
③ 更多介绍请参见 http://repzret.org/p/repzret/。

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
.....	.....
0x4000000000000000	2
0x8000000000000000	1

程序最后使用了 FATRET 指令，我们在上一个例子里已经介绍过它了。

Optimizing MSVC 2012

指令清单 19.32 MSVC 2012

```
a$ = 8
f       PROC
; RCX = input value
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f:
; pass 1 -----------------------------------
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -----------------------------------------
; pass 2 -----------------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax     ; rt++
$LN11@f:
        rol     rdx, 1  ; RDX=RDX<<1
; -----------------------------------------
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

MSVC 2012 优化编译而生成的代码与 MSVC 2010 大体相同。但是 MSVC 2012 把迭代次数为 64 次的循环，拆解为两个迭代 32 次的循环。坦白讲，笔者也不清楚个中缘由。或许是优化的结果，或许循环体更长一些比较好。总之，我特地把这个例子记录下来，希望读者注意：编译器可能生成各种匪夷所思的代码，但是这些代码的功能仍然会忠实于源程序。

19.5.3 ARM + Optimizing Xcode 4.6.3 (LLVM) + ARM mode

指令清单 19.33 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
            MOV             R1, R0
            MOV             R0, #0
            MOV             R2, #1
            MOV             R3, R0
loc_2E54
            TST             R1, R2,LSL R3 ; 根据 R1 & (R2<<R3) 设置标记
            ADD             R3, R3, #1    ; R3++
```

```
        ADDNE           R0, R0, #1    ; if ZF flag is cleared by TST, then R0++
        CMP             R3, #32
        BNE             loc_2E54
        BX              LR
```

ARM 模式的 TST 指令相当于 x86 指令集的 TEST 指令。

ARM 模式的指令里没有单独的位移操作指令（参见本书 41.2.1 节）。但是在数据处理指令——即 MOV、TST、CMP、ADD、SUB、RSB 指令中，可以使用参数调节符（如 LSL、LSR、ASR、ROR、RRX）实现位移运算。

通过“参数调节符（modificators）”进行位移运算的指令，必须明确位移的方法和位移的位数。

此处的“TST R1, R2, LSL R3”，可理解为 R1^(R2<<R3)。

19.5.4 ARM + Optimizing Xcode 4.6.3（LLVM）+ Thumb-2 mode

Thumb-2 模式的代码与 ARM 模式的代码基本相同。只不过在 Thumb-2 模式下，不可能直接在 TST 指令里直接使用参数调节符 LSL，所以它使用 LSL.W 和 TST 指令替代了 ARM 模式下的单条 TST 指令。

```
        MOV             R1, R0
        MOVS            R0, #0
        MOV.W           R9, #1
        MOVS            R3, #0
loc_2F7A
        LSL.W           R2, R9, R3
        TST             R2, R1
        ADD.W           R3, R3, #1
        IT NE
        ADDNE           R0, #1
        CMP             R3, #32
        BNE             loc_2F7A
        BX              LR
```

19.5.5 ARM64 + Optimizing GCC 4.9

本节使用的源程序，是 19.5.2 节中所示程序的 64 位修改版。

指令清单 19.34 Optimizing GCC (Linaro) 4.8

```
f:
        mov     w2, 0           ; rt=0
        mov     x5, 1
        mov     w1, w2
.L2:
        lsl     x4, x5, x1      ; w4 = w5<<w1 = 1<<i
        add     w3, w2, 1       ; new_rt=rt+1
        tst     x4, x0          ; (1<<i) & a
        add     w1, w1, 1       ; i++
; result of TST was non-zero?
; then w2=w3 or rt=new_rt.
; otherwise: w2=w2 or rt=rt (idle operation)
        csel    w2, w3, w2, ne
        cmp     w1, 64          ; i<64?
        bne     .L2             ;yes
        mov     w0, w2          ; return rt
        ret
```

这个程序与 GCC 生成的 x64 程序（请参见指令清单 19.30）十分相似。

其中，CSEL 指令的全称是“Conditional SELect”。它依据 TST 指令设置的标志位，相应地从候选的两个操作符中选取一个数据并复制给 W2。本例中，它传递的值是 rt。

19.5.6 ARM64 + Non-optimizing GCC 4.9

本节再次使用 19.5.2 节中所示程序的 64 位修改版。

由于编译时关闭了优化功能，所以编译器生成的代码十分庞大。

指令清单 19.35 Non-optimizing GCC (Linaro) 4.8

```
f:
        sub     sp, sp, #32
        str     x0, [sp,8]      ; store "a" value to Register Save Area
        str     wzr, [sp,24]    ; rt=0
        str     wzr, [sp,28]    ; i=0
        b       .L2
.L4:
        ldr     w0, [sp,28]
        mov     x1, 1
        lsl     x0, x1, x0      ; X0 = X1<<X0 = 1<<i
        mov     x1, x0
; X1 = 1<<1
        ldr     x0, [sp,8]
; X0 = a
        and     x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 contain zero? then jump to .L3, skipping "rt" increment
        cmp     x0, xzr
        beq     .L3
; rt++
        ldr     w0, [sp,24]
        add     w0, w0, 1
        str     w0, [sp,24]
.L3:
; i++
        ldr     w0, [sp,28]
        add     w0, w0, 1
        str     w0, [sp,28]
.L2:
; i<=63? then jump to .L4
        ldr     w0, [sp,28]
        cmp     w0, 63
        ble     .L4
; return rt
        ldr     w0, [sp,24]
        add     sp, sp, 32
        ret
```

19.5.7 MIPS

Non-optimizing GCC

指令清单 19.36 Non-optimizing GCC 4.4.5 (IDA)

```
f:
; IDA is not aware of variable names, we gave them manually:
rt              = -0x10
i               = -0xC
var_4           = -4
a               = 0

                addiu   $sp, -0x18
                sw      $fp, 0x18+var_4($sp)
                move    $fp, $sp
                sw      $a0, 0x18+a($fp)
; initialize rt and i variables to zero:
                sw      $zero, 0x18+rt($fp)
                sw      $zero, 0x18+i($fp)
; jump to loop check instructions:
                b       loc_68
```

```
                or      $at, $zero ; branch delay slot, NOP
loc_20:
                li      $v1, 1
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; load delay slot, NOP
                sllv    $v0, $v1, $v0
; $v0 = 1<<i
                move    $v1, $v0
                lw      $v0, 0x18+a($fp)
                or      $at, $zero ; load delay slot, NOP
                and     $v0, $v1, $v0
; $v0 = a&(1<<i)
; is a&(1<<i) equals to zero? jump to loc_58 then:
                beqz    $v0, loc_58
                or      $at, $zero
; no jump occurred, that means a&(1<<i)!=0, so increment "rt" then:
                lw      $v0, 0x18+rt($fp)
                or      $at, $zero ; load delay slot, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+rt($fp)

loc_58:
; increment i:
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; load delay slot, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+i($fp)

loc_68:
; load i and compare it with 0x20 (32).
; jump to loc_20 if it is less then 0x20 (32):
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; load delay slot, NOP
                slti    $v0, 0x20 # ' '
                bnez    $v0, loc_20
                or      $at, $zero ; branch delay slot, NOP
; function epilogue. return rt:
                lw      $v0, 0x18+rt($fp)
                move    $sp, $fp ; load delay slot
                lw      $fp, 0x18+var_4($sp)
                addiu   $sp, 0x18 ; load delay slot
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

这段程序十分直观：程序首先把所有的局部变量存储在局部栈里。在使用变量的时候，再把它们从栈里取出来。SLLV 指令的全称是“Shift Word Left Logical Variable”。SLL 指令只能把某个操作位移事先确定的比特位（这个位数是封装在 opcode 里的固定值），而 SLLV 指令从寄存器里读取位移的具体位数（编译时不能确定的变量）。

Optimizing GCC

启用优化功能之后，编译器生成的程序就会简略许多。不过它却使用了 2 条位移指令，比非优化编译而生成的程序还要多用一条位移指令。这是为什么？固然第一条 SLLV 指令可以替换为“跳转到第二个 SLLV 的无条件转移指令”，但是如此一来函数中就需要多用一个转移指令。而编译器的优化功能会刻意减少这种转移指令的使用次数，更多详情请参见本书 33.1 节。

指令清单 19.37 Optimizing GCC 4.4.5 (IDA)

```
f:
; $a0=a
; rt variable will reside in $v0:
                move    $v0, $zero
; i variable will reside in $v1:
                move    $v1, $zero
                li      $t0, 1
                li      $a3, 32
```

```
                sllv    $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i
loc_14:
                and     $a1, $a0
; $a1 = a&(1<<i)
; increment i:
                addiu   $v1, 1
; jump to loc\_28 if a&(1<<i)==0 and increment rt:
                beqz    $a1, loc_28
                addiu   $a2, $v0, 1
; if BEQZ was not triggered, save updated rt into $v0:
                move    $v0, $a2
loc_28:
; if i!=32, jump to loc_14 and also prepare next shifted value:
                bne     $v1, $a3, loc_14
                sllv    $a1, $t0, $v1
; return
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

19.6 本章小结

与 C/C++语言中的位移操作符<<和>>相对应，x86 指令集中有操作无符号数的 SHR/SHL 指令和操作有符号数的 SAR/SHL 指令。

在 ARM 平台上，对无符号数进行位移运算的指令是 LSR/LSL，对有符号数进行位移运算的指令是 ASR/LSL。而且，ARM 指令还可以通过“参数调节符”的形式使用“后缀型”位移运算指令。此类对其他数进行运算的指令，又叫做“数据处理指令”。

19.6.1 检测特定位（编译阶段）

对某数值的二进制 1000000 位(即 16 进制的 0x40)进行检测的指令大体如下。

指令清单 19.38 C/C++

```
if (input&0x40)
    ...
```

指令清单 19.39 x86

```
TEST REG, 40h
JNZ is_set
; bit is not set
```

指令清单 19.40 x86

```
TEST REG, 40h
JZ is_cleared
; bit is set
```

指令清单 19.41 ARM (ARM mode)

```
TST REG, #0x40
BNE is_set
; bit is not set
```

某些情况下，编译器会使用 AND 指令而不使用 TEST 指令。无论编译器选取了哪个指令，都不会影响程序将要检测的标志位。

19.6.2 检测特定位（runtime 阶段）

在 C/C++代码编译而得的程序中，检测特定位的指令（右移 n 位，然后舍弃最低位）大体如下。

指令清单 19.42 C/C++

```
if ((value>>n)&1)
    ...
```

相应的 x86 代码如下所示。

指令清单 19.43 x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

此外还可以把 1 左移 n 次，逐次鉴定各位是否为零。

指令清单 19.44 C/C++

```
if (value & (1<<n))
    ...
```

上述源程序对应的 x86 指令如下所示。

指令清单 19.45 x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

19.6.3 设置特定位（编译阶段）

指令清单 19.46 C/C++

```
value=value|0x40;
```

指令清单 19.47 x86

```
OR REG, 40h
```

指令清单 19.48 ARM (ARM mode) and ARM64

```
ORR R0, R0, #0x40
```

19.6.4 设置特定位（runtime 阶段）

指令清单 19.49 C/C++

```
value=value|(1<<n);
```

对应的 x86 指令大体如下所示。

指令清单 19.50 x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

19.6.5 清除特定位（编译阶段）

如需清除特定位，只需使用 AND 指令操作相应数值即可。

指令清单 19.51 C/C++

```
value=value&(~0x40);
```

指令清单 19.52 x86

```
AND REG, 0FFFFFFBFh
```

指令清单 19.53 x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

上述指令仅仅把被操作数的某个位置零，其他位保持不变。

ARM 平台的 ARM 模式指令集有 BIC 指令。它相当于 NOT 和 AND 指令对。

指令清单 19.54 ARM (ARM mode)

```
BIC R0, R0, #0x40
```

19.6.6 清除特定位（runtime 阶段）

指令清单 19.55 C/C++

```
value=value&(~(1<<n));
```

指令清单 19.56 x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

19.7 练习题

19.7.1 题目 1

请描述下述代码的功能。

指令清单 19.57 Optimizing MSVC 2010

```
_a$ = 8
_f      PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        mov     eax, ecx
        mov     edx, ecx
        shl     edx, 16         ; 00000010H
        and     eax, 65280      ; 0000ff00H
        or      eax, edx
        mov     edx, ecx
        and     edx, 16711680   ; 00ff0000H
        shr     ecx, 16         ; 00000010H
        or      edx, ecx
        shl     eax, 8
        shr     edx, 8
        or      eax, edx
        ret     0
_f      ENDP
```

指令清单 19.58 Optimizing Keil 6/2013 (ARM mode)

```
f PROC
        MOV     r1,#0xff0000
        AND     r1,r1,r0,LSL #8
        MOV     r2,#0xff00
        ORR     r1,r1,r0,LSR #24
        AND     r2,r2,r0,LSR #8
        ORR     r1,r1,r2
        ORR     r0,r1,r0,LSL #24
        BX      lr
        ENDP
```

指令清单 19.59 Optimizing Keil 6/2013 (Thumb mode)

```
f  PROC
        MOVS    r3,#0xff
        LSLS    r2,r0,#8
        LSLS    r3,r3,#16
        ANDS    r2,r2,r3
        LSRS    r1,r0,#24
        ORRS    r1,r1,r2
        LSRS    r2,r0,#8
        ASRS    r3,r3,#8
        ANDS    r2,r2,r3
        ORRS    r1,r1,r2
        LSLS    r0,r0,#24
        ORRS    r0,r0,r1
        BX      lr
        ENDP
```

指令清单 19.60 Optimizing GCC 4.9 (ARM64)

```
f:
        rev     w0, w0
        ret
```

指令清单 19.61 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
            srl     $v0, $a0, 24
            sll     $v1, $a0, 24
            sll     $a1, $a0, 8
            or      $v1, $v0
            lui     $v0, 0xFF
            and     $v0, $a1, $v0
            srl     $a0, 8
            or      $v0, $v1, $v0
            andi    $a0, 0xFF00
            jr      $ra
            or      $v0, $a0
```

19.7.2 题目 2

请描述下述程序的功能。

指令清单 19.62 Optimizing MSVC 2010

```
_a$ = 8                                 ; size = 4
_f      PROC
        push   esi
        mov    esi, DWORD PTR _a$[esp]
        xor    ecx, ecx
        push   edi
        lea    edx, DWORD PTR [ecx+1]
```

```
        xor   eax, eax
        npad  3 ; align next label
$LL3@f:
        mov   edi, esi
        shr   edi, cl
        add   ecx, 4
        and   edi, 15
        imul  edi, edx
        lea   edx, DWORD PTR [edx+edx*4]
        add   eax, edi
        add   edx, edx
        cmp   ecx, 28
        jle   SHORT $LL3@f
        pop   edi
        pop   esi
        ret   0
_f      ENDP
```

指令清单 19.63 Optimizing Keil 6/2013 (ARM mode)

```
f PROC
        MOV   r3,r0
        MOV   r1,#0
        MOV   r2,#1
        MOV   r0,r1
|L0.16|
        LSR   r12,r3,r1
        AND   r12,r12,#0xf
        MLA   r0,r12,r2,r0
        ADD   r1,r1,#4
        ADD   r2,r2,r2,LSL #2
        CMP   r1,#0x1c
        LSL   r2,r2,#1
        BLE   |L0.16|
        BX    lr
        ENDP
```

指令清单 19.64 Optimizing Keil 6/2013 (Thumb mode)

```
f  PROC
        PUSH    {r4,lr}
        MOVS    r3,r0
        MOVS    r1,#0
        MOVS    r2,#1
        MOVS    r0,r1
|L0.10|
        MOVS    r4,r3
        LSRS    r4,r4,r1
        LSLS    r4,r4,#28
        LSRS    r4,r4,#28
        MULS    r4,r2,r4
        ADDS    r0,r4,r0
        MOVS    r4,#0xa
        MULS    r2,r4,r2
        ADDS    r1,r1,#4
        CMP     r1,#0x1c
        BLE     |L0.10|
        POP     {r4,pc}
        ENDP
```

指令清单 19.65 Non-optimizing GCC 4.9 (ARM64)

```
f:
        sub     sp, sp, #32
        str     w0, [sp,12]
        str     wzr, [sp,28]
        mov     w0, 1
```

```
        str     w0, [sp,24]
        str     wzr, [sp,20]
        b       .L2
.L3:
        ldr     w0, [sp,28]
        ldr     w1, [sp,12]
        lsr     w0, w1, w0
        and     w1, w0, 15
        ldr     w0, [sp,24]
        mul     w0, w1, w0
        ldr     w1, [sp,20]
        add     w0, w1, w0
        str     w0, [sp,20]
        ldr     w0, [sp,28]
        add     w0, w0, 4
        str     w0, [sp,28]
        ldr     w1, [sp,24]
        mov     w0, w1
        lsl     w0, w0, 2
        add     w0, w0, w1
        lsl     w0, w0, 1
        str     w0, [sp,24]
.L2:
        ldr     w0, [sp,28]
        cmp     w0, 28
        ble     .L3
        ldr     w0, [sp,20]
        add     sp, sp, 32
        ret
```

指令清单 19.66 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                rl      $v0, $a0, 8
                srl     $a3, $a0, 20
                andi    $a3, 0xF
                andi    $v0, 0xF
                srl     $a1, $a0, 12
                srl     $a2, $a0, 16
                andi    $a1, 0xF
                andi    $a2, 0xF
                sll     $t2, $v0, 4
                sll     $v1, $a3, 2
                sll     $t0, $v0, 2
                srl     $t1, $a0, 4
                sll     $t5, $a3, 7
                addu    $t0, $t2
                subu    $t5, $v1
                andi    $t1, 0xF
                srl     $v1, $a0, 28
                sll     $t4, $a1, 7
                sll     $t2, $a2, 2
                sll     $t3, $a1, 2
                sll     $t7, $a2, 7
                srl     $v0, $a0, 24
                addu    $a3, $t5, $a3
                subu    $t3, $t4, $t3
                subu    $t7, $t2
                andi    $v0, 0xF
                sll     $t5, $t1, 3
                sll     $t6, $v1, 8
                sll     $t2, $t0, 2
                sll     $t4, $t1, 1
                sll     $t1, $v1, 3
                addu    $a2, $t7, $a2
                subu    $t1, $t6, $t1
                addu    $t2, $t0, $t2
```

```
addu    $t4, $t5
addu    $a1, $t3, $a1
sll     $t5, $a3, 2
sll     $t3, $v0, 8
sll     $t0, $v0, 3
addu    $a3, $t5
subu    $t0, $t3, $t0
addu    $t4, $t2, $t4
sll     $t3, $a2, 2
sll     $t2, $t1, 6
sll     $a1, 3
addu    $a1, $t4, $a1
subu    $t1, $t2, $t1
addu    $a2, $t3
sll     $t2, $t0, 6
sll     $t3, $a3, 2
andi    $a0, 0xF
addu    $v1, $t1, $v1
addu    $a0, $a1
addu    $a3, $t3
subu    $t0, $t2, $t0
sll     $a2, 4
addu    $a2, $a0, $a2
addu    $v0, $t0, $v0
sll     $a1, $v1, 2
sll     $a3, 5
addu    $a3, $a2, $a3
addu    $v1, $a1
sll     $v0, 6
addu    $v0, $a3, $v0
sll     $v1, 7
jr      $ra
addu    $v0, $v1, $v0
```

19.7.3　题目 3

请查阅 MSDN 资料，找到 MessageBox()函数使用了哪些标志位。

指令清单 19.67　Optimizing MSVC 2010

```
_main   PROC
        push    278595          ; 00044043H
        push    OFFSET $SG79792 ; 'caption'
        push    OFFSET $SG79793 ; 'hello, world!'
        push    0
        call    DWORD PTR __imp__MessageBoxA@16
        xor     eax, eax
        ret     0
_main   ENDP
```

19.7.4　题目 4

请描述下述代码的功能。

指令清单 19.68　Optimizing MSVC 2010

```
_m$ = 8         ; size = 4
_n$ = 12        ; size = 4
_f      PROC
        mov     ecx, DWORD PTR _n$[esp-4]
        xor     eax, eax
        xor     edx, edx
        test    ecx, ecx
        je      SHORT $LN2@f
```

```
        push    esi
        mov     esi, DWORD PTR _m$[esp]
$LL3@f:
        test    cl, 1
        je      SHORT $LN1@f
        add     eax, esi
        adc     edx, 0
$LN1@f:
        add     esi, esi
        shr     ecx, 1
        jne     SHORT $LL3@f
        pop     esi
$LN1@f:
        ret     0
_f      ENDP
```

指令清单 19.69 Optimizing Keil 6/2013 (ARM mode)

```
f PROC
        PUSH    {r4,lr}
        MOV     r3,r0
        MOV     r0,#0
        MOV     r2,r0
        MOV     r12,r0
        B       |L0.48|
|L0.24|
        TST     r1,#1
        BEQ     |L0.40|
        ADDS    r0,r0,r3
        ADC     r2,r2,r12
|L0.40|
        LSL     r3,r3,#1
        LSR     r1,r1,#1
|L0.48|
        CMP     r1,#0
        MOVEQ   r1,r2
        BNE     |L0.24|
        POP     {r4,pc}
        ENDP
```

指令清单 19.70 Optimizing Keil 6/2013 (Thumb mode)

```
f  PROC
        PUSH    {r4,r5,lr}
        MOVS    r3,r0
        MOVS    r0,#0
        MOVS    r2,r0
        MOVS    r4,r0
        B       |L0.24|
|L0.12|
        LSLS    r5,r1,#31
        BEQ     |L0.20|
        ADDS    r0,r0,r3
        ADCS    r2,r2,r4
|L0.20|
        LSLS    r3,r3,#1
        LSRS    r1,r1,#1
|L0.24|
        CMP     r1,#0
        BNE     |L0.12|
        MOVS    r1,r2
        POP     {r4,r5,pc}
        ENDP
```

指令清单 19.71 Optimizing GCC 4.9 (ARM64)

```
f:
        mov     w2, w0
        mov     x0, 0
        cbz     w1, .L2
.L3:
        and     w3, w1, 1
        lsr     w1, w1, 1
        cmp     w3, wzr
        add     x3, x0, x2, uxtw
        lsl     w2, w2, 1
        csel    x0, x3, x0, ne
        cbnz    w1, .L3
.L2:
        ret
```

指令清单 19.72 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
mult:
                beqz $a1, loc_40
                move $a3, $zero
                move $a2, $zero
                addu $t0, $a3, $a0
loc_10:                                 # CODE XREF: mult+38
                sltu $t1, $t0, $a3
                move $v1, $t0
                andi $t0, $a1, 1
                addu $t1, $a2
                beqz $t0, loc_30
                srl  $a1, 1
                move $a3, $v1
                move $a2, $t1
loc_30:                                 # CODE XREF: mult+20
                beqz $a1, loc_44
                sll  $a0, 1
                b    loc_10
                addu $t0, $a3, $a0
loc_40:                                 # CODE XREF: mult
                move $a2, $zero
loc_44:                                 # CODE XREF: mult:loc_30
                move $v0, $a2
                jr   $ra
                move $v1, $a3
```

第 20 章　线性同余法与伪随机函数

“线性同余法”大概是生成随机数的最简方法。虽然现在的随机函数基本不采用这种技术了[①]，但是它原理简单（只涉及乘法、加法和与运算），仍然值得研究。

```
#include <stdint.h>

// constants from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

上述程序包含两个函数。第一个函数用于初始化内部状态，第二个函数生成伪随机数字。

这个程序中的两个常量来自于参考书目 [Pre+07]。本文直接使用 C/C++的#define 语句，把它们定义为两个宏。C/C++对宏和常量的处理方法有所区别。C/C++编译器的预处理程序会把宏直接替换为相应的值。所以宏并不像变量那样占用内存。相对而言，编译器把常量当作只读变量处理。因为指针的本质是内存地址，所以 C/C++的指针可以指向常量，但是无法指向宏。

函数最后使用了“&”/与操作符。根据 C 语言有关标准，my_rand()函数的返回值应当介于 0～32767 之间。如果您有意生成 32 位的伪随机数，那么就不必在此处进行与运算。

20.1　x86

指令清单 20.1　Optimizing MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
```

① 多数随机函数都采用梅森旋转算法（Mersenne twister）。

```
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767              ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

编译器把源代码中的宏直接替换为宏所绑定的常量。这个例子证明。编译器不会给宏单独分配内存。my_srand()函数直接把输入值传递给内部的 rand_state 变量。

此后，my_rand()函数接收了这个值，以此计算 rand_state。在调整了它的宽度之后，把返回值保留在 EAX 寄存器里。

更为详尽的计算方法可参见如下所示的非优化编译而生成的程序。

指令清单 20.2　Non-optimizing MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767              ;00007 fffH
        pop     ebp
        ret     0
_rand   ENDP

_TEXT   ENDS
```

20.2　x64

x64 的程序与 x86 的程序几乎相同。因为函数的返回值属于 int 型数据，所以它没有使用 64 位寄存器，而是使用了 32 位寄存器的助记符。但是，my_srand()函数从 ECX 寄存器获取的所需参数，没有通过栈读取参数，这构成了 64 位程序的显著特征。

指令清单 20.3　Optimizing MSVC 2013 x64

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = input argument
        mov     DWORD PTR rand_state, ecx
```

```
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525   ; 0019660dH
        add     eax, 1013904223                      ; 3c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767                           ; 00007fffH
        ret     0
my_rand ENDP

_TEXT ENDS
```

GCC 生成的程序与此类似。

20.3 32 位 ARM

指令清单 20.4 Optimizing Keil 6/2013 (ARM mode)

```
my_srand PROC
        LDR     r1,|L0.52|  ; load pointer to rand_state
        STR     r0,[r1,#0]  ; save rand_state
        BX      lr
        ENDP

my_rand PROC
        LDR     r0,|L0.52|  ; load pointer to rand_state
        LDR     r2,|L0.56|  ; load RNG_a
        LDR     r1,[r0,#0]  ; load rand_state
        MUL     r1,r2,r1
        LDR     r2,|L0.60|  ; load RNG_c
        ADD     r1,r1,r2
        STR     r1,[r0,#0]  ; save rand_state
; AND with 0x7FFF:
        LSL     r0,r1,#17
        LSR     r0,r0,#17
        BX      lr
        ENDP

|L0.52|
        DCD     ||.data||
|L0.56|
        DCD     0x0019660d
|L0.60|
        DCD     0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
        DCD     0x00000000
```

ARM 模式的单条指令本来就是 32 位 opcode，所以单条指令不可能容纳得下 32 位常量。因此，Keil 单独开辟了一些空间来存储常量，再通过额外的指令读取它们。

值得关注的是，常量 0x7fff 也无法封装在单条指令之中。Keil 把 rand_state 左移 17 位之后再右移 17 位，这相当于 C/C++程序中的“(rand_state<<17)>>17”语句。虽然看上去这是画蛇添足，但是它清除了寄存器的高 17 位，保留了低 15 位数据，与源代码的功能相符。

Keil 优化编译而生成的程序与此雷同，本文不再单独介绍。

20.4 MIPS

指令清单 20.5 Optimizing GCC 4.4.5 (IDA)

```
my_srand:
```

```
; store $a0 to rand_state:
                lui     $v0, (rand_state >> 16)
                jr      $ra
                sw      $a0, rand_state
my_rand:
; load rand_state to $v0:
                lui     $v1, (rand_state >> 16)
                lw      $v0, rand_state
                or      $at, $zero  ; load delay slot
; multiplicate rand_state in $v0 by 1664525 (RNG_a):
                sll     $a1, $v0, 2
                sll     $a0, $v0, 4
                addu    $a0, $a1, $a0
                sll     $a1, $a0, 6
                subu    $a0, $a1, $a0
                addu    $a0, $v0
                sll     $a1, $a0, 5
                addu    $a0, $a1
                sll     $a0, 3
                addu    $v0, $a0, $v0
                sll     $a0, $v0, 2
                addu    $v0, $a0
; add 1013904223 (RNG_c)
; the LI instruction is coalesced by IDA from LUI and ORI
                li      $a0, 0x3C6EF35F
                addu    $v0, $a0
; store to rand_state:
                sw      $v0, (rand_state & 0xFFFF)($v1)
                jr      $ra
                andi    $v0, 0x7FFF ; branch delay slot
```

在上述程序中，常量只有 0x3C6EF35F （即 1013904223）。另一个值为 1664525 的常量去哪了？编译器通过位移和加法运算实现了“乘以 1664525”的运算。我们假设编译器自行生成了以下函数：

```
#define RNG_a 1664525

int f (int a)
{
        return a*RNG_a;
}
```

那么，上述程序的汇编指令应当如下所示。

指令清单 20.6　Optimizing GCC 4.4.5 (IDA)

```
f:
                sll     $v1, $a0, 2
                sll     $v0, $a0, 4
                addu    $v0, $v1, $v0
                sll     $v1, $v0, 6
                subu    $v0, $v1, $v0
                addu    $v0, $a0
                sll     $v1, $v0, 5
                addu    $v0, $v1
                sll     $v0, 3
                addu    $a0, $v0, $a0
                sll     $v0, $a0, 2
                jr      $ra
                addu    $v0, $a0, $v0 ; branch delay slot
```

我们确实可以在可执行程序中找到对应的指令！

MIPS 的重新定位

本节重点介绍寄存器与内存交换数据的具体方法。在展现 MIPS 程序的细节方面，IDA 略有不足（“智能”整理得太严重）。因此本文使用 objdump 程序，分别观测到汇编层面的指令清单及重定位表（relocation list）。

指令清单 20.7 Optimizing GCC 4.4.5 (objdump)

```
# objdump -D rand_O3.o
...

00000000 <my_srand>:
   0:   3c020000        lui     v0,0x0
   4:   03e00008        jr      ra
   8:   ac440000        sw      a0,0(v0)

0000000c <my_rand>:
   c:   3c030000        lui     v1,0x0
  10:   8c620000        lw      v0,0(v1)
  14:   00200825        move    at,at
  18:   00022880        sll     a1,v0,0x2
  1c:   00022100        sll     a0,v0,0x4
  20:   00a42021        addu    a0,a1,a0
  24:   00042980        sll     a1,a0,0x6
  28:   00a42023        subu    a0,a1,a0
  2c:   00822021        addu    a0,a0,v0
  30:   00042940        sll     a1,a0,0x5
  34:   00852021        addu    a0,a0,a1
  38:   000420c0        sll     a0,a0,0x3
  3c:   00821021        addu    v0,a0,v0
  40:   00022080        sll     a0,v0,0x2
  44:   00441021        addu    v0,v0,a0
  48:   3c043c6e        lui     a0,0x3c6e
  4c:   3484f35f        ori     a0,a0,0xf35f
  50:   00441021        addu    v0,v0,a0
  54:   ac620000        sw      v0,0(v1)
  58:   03e00008        jr      ra
  5c:   30427fff        andi    v0,v0,0x7fff

...
# objdump -r rand_O3.o
...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000000 R_MIPS_HI16       .bss
00000008 R_MIPS_LO16       .bss
0000000c R_MIPS_HI16       .bss
00000010 R_MIPS_LO16       .bss
00000054 R_MIPS_LO16       .bss
...
```

函数 my_srand()两次出现了 relocation 现象。第一次出现在地址 0 处，其类型为 R_MIPS_HI16。第二处出现在地址 8 处，类型为 R_MIPS_L016。这意味着.bss 段的起始地址要写到地址为 0、8（分别为 16 位的高、低地址）的指令中去。

位于.bss 段起始位置的值，是变量 rand_state。

在前几条 LUI 和 SW 的指令中，某些操作符是零。因为编译器在编译阶段(complier)还不能确定这些值，所以此时把它空了出来。编译器将会在链接阶段（linker）处理这些数据，把地址的高地址位传递给 LUI 指令的相应操作符中，并把低地址位传递给 SW 指令。SW 指令会对地址的低地址位和$V0 寄存器的高地址位进行加权求和。

有了上述概念之后我们就不难理解 my_rand()函数中的重定位：重定位标记 R_MIPS_H16 告诉 linker 程序“要把.bss 段地址的高地址位传递给 LUI 指令”。所以变量 rand_state 地址的高地址位将会存储在$V1 寄存器。地址为 0x10 的 LW 指令再把变量地址的低半部分读出来，然后加到$V1 寄存器中。地址为 0x54 的 SW 指令再次进行这种求和，把新的数值传递给全局变量 rand_state。

在读取 MIPS 程序的重定位信息之后，IDA 程序会把数据与指令进行智能匹配。因此我们无法在 IDA 中看到这些原始的汇编指令。不过，无论 IDA 是否显示它们，重定位信息确实存在。

20.5　本例的线程安全改进版

请参见本书 65.1 节查询本例的线程安全（thread-safe）改进版。

第21章 结 构 体

在C/C++的数据结构里结构体（structure）是由一系列数据简单堆积而成的数据类型。结构体中的各项数据元素，可以是相同类型的数据、也可以是不同类型的数据。[①]

21.1 MSVC: systemtime

本节以Win32描述系统时间的SYSTEMTIME结构体为例。

库函数对它的定义如下[②]。

指令清单21.1 WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

根据上述声明，获取系统时间的程序大致如下：

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

使用MSVC 2010（启用/GS-选项）编译这个程序，可得如下所示的代码。

指令清单21.2 MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    lea     eax, DWORD PTR _t$[ebp]
    push    eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   ecx, WORD PTR _t$[ebp+12] ; wSecond
    push    ecx
```

① 又称为“异构容器/heterogeneous container”。

② https://msdn.microsoft.com/en-us/library/ms724950(VS.85).aspx。

```
    movzx   edx, WORD PTR _t$[ebp+10] ; wMinute
    push    edx
    movzx   eax, WORD PTR _t$[ebp+8] ; wHour
    push    eax
    movzx   ecx, WORD PTR _t$[ebp+6] ; wDay
    push    ecx
    movzx   edx, WORD PTR _t$[ebp+2] ; wMonth
    push    edx
    movzx   eax, WORD PTR _t$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call    _printf
    add     esp, 28
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

函数在栈里为这个结构体申请了 16 字节空间。这个结构体由 8 个 WORD 型数据构成，每个 WORD 型数据占用 2 字节，所以整个结构体正好需要 16 字节的存储空间。

这个结构体的第一个字段是 wYear。根据 MSDN 有关 SYSTEMTIME 结构体的相关声明[①]，在使用 GetSystemTime()函数时，传递给函数的是 SYSTEMTIME 结构体的指针。但是换个角度看，这也是 wYear 字段的指针。GetSystemTime()函数首先会在结构体的首地址写入年份信息，然后再把指针调整 2 个字节并写入月份信息，如此类推写入全部信息。

21.1.1　OllyDbg

我们使用 MSVC 2010（指定/GS- /MD 选项）编译上述程序，并用 OllyDbg 打开 MSVC 生成的可执行文件。找到传递给 GetSystemTime()函数的指针地址，然后在数据观察窗口里观察这部分数据。此时数据如图 21.1 所示。

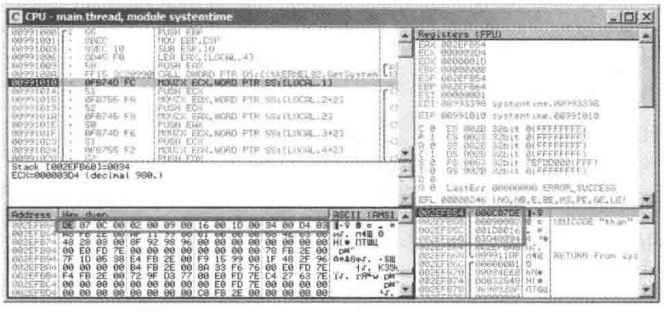


图 21.1　OllyDbg：执行 GetSystemTime()

在执行函数时精确的系统时间是“9 december 2014, 22:29:52”如图 21.2 所示。

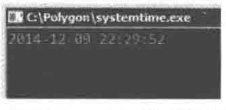


图 21.2　OllyDbg：printf()函数的输出结果

我们在数据窗口看到这个地址开始的 16 字节空间的值是：

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

这段空间的每 2 个字节代表结构体的一个字段。由于采用了小端字节序，所以就同一个 WORD 型数据而言，数权较小的一个字节在前，数权较大的字节在后。我们将其整理一下，看看它们的实际涵义：

十六进制数	十进制含义	字段名
0x07DE	2014	wYear
0x000C	12	wMonth

① https://msdn.microsoft.com/en-us/library/ee488017.aspx。

续表

十六进制数	十进制含义	字段名
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliSeconds

在栈窗口里看到的值与此相同，不过栈窗口以 32 位数据的格式组织数据。

而后 printf()函数从结构体中获取所需数据，并在屏幕上进行相应输出。

printf()函数并没有将所有的字段都打印出来。wDayOfWeek 和 wMilliSeconds 都未被输出，但是内存中确实有它们对应的值。

21.1.2 以数组替代结构体

结构体中的各个元素，在内存里依次排列。为了验证它在内存中的存储状况和数组相同，我用数组替代了 SYSTEMTIME 结构体：

```
#include <windows.h>
#include <stdio.h>
void main()
{
    WORD array[8];
    GetSystemTime (array);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);
    return;
};
```

编译器会提示警告信息：

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

即使如此，MSVC 2010 仍能够进行编译。它生成的代码如下所示。

指令清单 21.3 Non-optimizing MSVC 2010

```
$SG78573 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
_array$ = -16;size=16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
```

```
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

即使调整了数据类型，编译器生成的程序仍然没有什么变化。

这个现象说明，即使我们使用数组替代了原有结构体，编译器生成的汇编指令依旧完全相同。仅仅从汇编指令分析，很难判断出到底源程序使用的是多变量的结构体还是数组。

好在正常人不会做这种别扭的替换。毕竟结构体的可读性、易用性都比数组强，也方便编程人员替换结构体中的字段。

因为这个程序和前面的程序完全相同，本书就不再演示 OllyDbg 的调试过程了。

21.2 用 malloc()分配结构体的空间

在某些情况下，使用堆（heap）来存储结构体要比栈（stack）容易一些：

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;
    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);
    free (t);
    return;
};
```

现在启用 MSVC 的优化选项/Ox，编译上述程序，得到的代码如下所示。

指令清单 21.4 Optimizing MSVC

```
_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
```

```
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

16 就是 sizeof(SYSTEMTIME)，即 malloc()分配空间的确切大小。malloc()函数根据参数指定的大小分配一块空间，并把空间的指针传递给 EAX 寄存器。而后 ESI 寄存器获取了这个指针。Win32 的 GetSystemTime()函数把返回值的各个项存储到 esi 指针指向的对应空间里，接下来几个寄存器依次读取这些返回值并将之依次推送入栈、给 printf()函数调用。

此处的“MOVZX(Move with Zero eXtent)”是前文没有介绍过的指令。这个指令的作用和 MOVSX（参见第 14 章）的相似。与 MOVSX 的不同之处是，MOVZX 在进行格式转换的时候会将其他 bit 位清零。因为 printf()函数需要的数据类型是 32 位整型数据，而我们的结构体 SYSTEMTIME 里对应的字段是 WORD 型数据，所以此处要转换数据类型。WORD 型数据是 16 位无符号数据，因而要把 WORD 型数据照抄到 int 型数据空间的低地址位，并把高地址位（第 16 位到第 31 位）清零。高地址位必须要清零，否则转换的 int 型数据很可能会受到脏数据问题的不良影响。

本例中，我们可以使用 8 个 WORD 型数组重新构造上述结构体：

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

使用 MSVC（启用优化选项）编译上述程序，可得到如下所示的代码。

指令清单 21.5 Optimizing MSVC

```
$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
```

```
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

这个代码和结构体生成的代码完全相同。我再次强调，这种“用数组替代结构体”的做法没有什么实际意义。除非有必要，否则不必做这种替换。

21.3 UNIX: struct tm

21.3.1 Linux

我们研究一下 Linux 源文件 time.h 里的 tm 结构体。

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

使用 GCC 4.4.1 编译，可得如下所示的代码。

指令清单 21.6 GCC 4.4.1

```
main proc  near
     push    ebp
     mov     ebp, esp
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; first argument for time()
     call    time
     mov     [esp+3Ch], eax
     lea     eax, [esp+3Ch]  ; take pointer to what time() returned
     lea     edx, [esp+10h]  ; at ESP+10h struct tm will begin
     mov     [esp+4], edx    ; pass pointer to the structure begin
     mov     [esp], eax      ; pass pointer to result of time()
     call    localtime_r
     mov     eax, [esp+24h]  ; tm_year
     lea     edx, [eax+76Ch] ; edx=eax+1900
     mov     eax, offset format ; "Year: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
```

```
        mov     edx, [esp+20h]      ; tm_mon
        mov     eax, offset aMonthD ; "Month: %d\n"
        mov     [esp+4], edx
        mov     [esp], eax
        call    printf
        mov     edx, [esp+1Ch]      ; tm_mday
        mov     eax, offset aDayD   ; "Day: %d\n"
        mov     [esp+4], edx
        mov     [esp], eax
        call    printf
        mov     edx, [esp+18h]      ; tm_hour
        mov     eax, offset aHourD ; "Hour: %d\n"
        mov     [esp+4], edx
        mov     [esp], eax
        call    printf
        mov     edx, [esp+14h]        ; tm_min
        mov     eax, offset aMinutesD ; "Minutes: %d\n"
        mov     [esp+4], edx
        mov     [esp], eax
        call    printf
        mov     edx, [esp+10h]
        mov     eax, offset aSecondsD ; "Seconds: %d\n"
        mov     [esp+4], edx          ; tm_sec
        mov     [esp], eax
        call    printf
        leave
        retn
main endp
```

可惜的是 IDA 未能在局部栈里将这些局部变量逐一识别出来，因此未能标记变量的别名。但是读到这里的读者都是有经验的逆向工程分析人员了，不再需要辅助信息仍然可以分析出数据的作用。

需要注意的是“lea edx, [eax+76Ch]”指令。它给 EAX 寄存器里的值加上 0x76C（1900），而不会修改任何标志位。有关 LEA 指令的更多信息，请参见附录 A.6.2。

GDB

我们使用 GDB 调试这个程序，可得到如下所示的代码。[①]

指令清单 21.7 GDB

```
dennis@ubuntuvm:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm
Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29     printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc:    0x080484c3  0x080485c0      0x000007de      0x00000000
```

① 为了便于演示，作者略微调整了 date 的返回结果。在实际情况下，手动输入 GDB 指令的速度不会那么快，多次操作的返回结果不可能分毫不差。

```
0xbffff0ec:    0x08048301  0x538c93ed      0x00000025      0x0000000a
0xbffff0fc:    0x00000012  0x00000002      0x00000005      0x00000072
0xbffff10c:    0x00000001  0x00000098      0x00000001      0x00002a30
0xbffff11c:    0x0804b090  0x08048530      0x00000000      0x00000000
(gdb)
```

在数据栈中，结构体的构造非常清晰。首先我们看看 time.h 里的定义。

指令清单 21.8　time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

Linux 的 tm 结构体的每个元素都是 int 型数据。在数据类型上它就和 Windows 的 SYSTEMTIME 结构体采用的 WORD 型数据不同。

我们继续分析局部栈的情况：

```
0xbffff0dc: 0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x00000025 秒   0x0000000a 分
0xbffff0fc: 0x00000012 时   0x00000002 mday 0x00000005 月   0x00000072 年
0xbffff10c: 0x00000001 wday 0x00000098 yday 0x00000001 isdst 0x00002a30
0xbffff11c: 0x0804b090      0x08048530      0x00000000      0x00000000
```

整理一下，结果如下表所示。

十六进制数	十进制	字段
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

这个结构体比 SYSTEMTIME 多了一些字段，例如 tm_wday/ tm_yday/ tm_isdst 字段，不过本例用不到这些字段。

21.3.2　ARM

Optimizing Keil 6/2013 (Thumb mode)

使用 Keil 6（启用优化选项）编译上述结构体程序，可得到 Thumb 模式的程序如下所示。

指令清单 21.9　Optimizing Keil 6/2013 (Thumb mode)

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
```

```
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

        PUSH    {LR}
        MOVS    R0, #0          ; timer
        SUB     SP, SP, #0x34
        BL      time
        STR     R0, [SP,#0x38+timer]
        MOV     R1, SP          ; tp
        ADD     R0, SP, #0x38+timer ; timer
        BL      localtime_r
        LDR     R1, =0x76C
        LDR     R0, [SP,#0x38+var_24]
        ADDS    R1, R0, R1
        ADR     R0, aYearD      ; "Year: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_28]
        ADR     R0, aMonthD     ; "Month: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_2C]
        ADR     R0, aDayD       ; "Day: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_30]
        ADR     R0, aHourD      ; "Hour: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_34]
        ADR     R0, aMinutesD   ; "Minutes: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_38]
        ADR     R0, aSecondsD   ; "Seconds: %d\n"
        BL      __2printf
        ADD     SP, SP, #0x34
        POP     {PC}
```

Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

通过分析被调用方函数的函数名称（例如本例的 localtime_r()函数），IDA 能够“识别”出返回值为结构体型数据，并能给结构体中的字段重新标注名称。

指令清单 21.10 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
var_38 = -0x38
var_34 = -0x34

        PUSH {R7,LR}
        MOV  R7, SP
        SUB  SP, SP, #0x30
        MOVS R0, #0  ; time_t *
        BLX  _time
        ADD  R1, SP, #0x38+var_34 ; struct tm *
        STR  R0, [SP,#0x38+var_38]
        MOV  R0, SP  ; time_t *
        BLX  _localtime_r
        LDR  R1, [SP,#0x38+var_34.tm_year]
        MOV  R0, 0xF44 ; "Year: %d\n"
        ADD  R0, PC  ; char *
        ADDW R1, R1, #0x76C
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
```

```
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_min]
        MOV  R0, 0xF28 ; "Minutes: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34]
        MOV  R0, 0xF25 ; "Seconds: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        ADD  SP, SP, #0x30
        POP  {R7,PC}
......
00000000 tm        struc ; (sizeof=0x2C, standard type)
00000000 tm_sec    DCD ?
00000004 tm_min    DCD ?
00000008 tm_hour   DCD ?
0000000C tm_mday   DCD ?
00000010 tm_mon    DCD ?
00000014 tm_year   DCD ?
00000018 tm_wday   DCD ?
0000001C tm_yday   DCD ?
00000020 tm_isdst  DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone   DCD ? ; offset
0000002C tm        ends
```

21.3.3 MIPS

指令清单 21.11 Optimizing GCC 4.4.5 (IDA)

```
 1 main:
 2
 3 ; IDA is not aware of structure field names, we named them manually:
 4
 5 var_40          = -0x40
 6 var_38          = -0x38
 7 seconds         = -0x34
 8 minutes         = -0x30
 9 hour            = -0x2C
10 day             = -0x28
11 month           = -0x24
12 year            = -0x20
13 var_4           = -4
14
15                 lui     $gp, (__gnu_local_gp >> 16)
16                 addiu   $sp, -0x50
17                 la      $gp, (__gnu_local_gp & 0xFFFF)
18                 sw      $ra, 0x50+var_4($sp)
19                 sw      $gp, 0x50+var_40($sp)
20                 lw      $t9, (time & 0xFFFF)($gp)
21                 or      $at, $zero ; load delay slot, NOP
22                 jalr    $t9
23                 move    $a0, $zero ; branch delay slot, NOP
24                 lw      $gp, 0x50+var_40($sp)
25                 addiu   $a0, $sp, 0x50+var_38
26                 lw      $t9, (localtime_r & 0xFFFF)($gp)
```

```
27              addiu   $a1, $sp, 0x50+seconds
28              jalr    $t9
29              sw      $v0, 0x50+var_38($sp) ; branch delay slot
30              lw      $gp, 0x50+var_40($sp)
31              lw      $a1, 0x50+year($sp)
32              lw      $t9, (printf & 0xFFFF)($gp)
33              la      $a0, $LC0 # "Year: %d\n"
34              jalr    $t9
35              addiu   $a1, 1900 ; branch delay slot
36              lw      $gp, 0x50+var_40($sp)
37              lw      $a1, 0x50+month($sp)
38              lw      $t9, (printf & 0xFFFF)($gp)
39              lui     $a0, ($LC1 >> 16) # "Month: %d\n"
40              jalr    $t9
41              la      $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ; branch delay slot
42              lw      $gp, 0x50+var_40($sp)
43              lw      $a1, 0x50+day($sp)
44              lw      $t9, (printf & 0xFFFF)($gp)
45              lui     $a0, ($LC2 >> 16) # "Day: %d\n"
46              jalr    $t9
47              la      $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ; branch delay slot
48              lw      $gp, 0x50+var_40($sp)
49              lw      $a1, 0x50+hour($sp)
50              lw      $t9, (printf & 0xFFFF)($gp)
51              lui     $a0, ($LC3 >> 16) # "Hour: %d\n"
52              jalr    $t9
53              la      $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ; branch delay slot
54              lw      $gp, 0x50+var_40($sp)
55              lw      $a1, 0x50+minutes($sp)
56              lw      $t9, (printf & 0xFFFF)($gp)
57              lui     $a0, ($LC4 >> 16) # "Minutes: %d\n"
58              jalr    $t9
59              la      $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ; branch delay slot
60              lw      $gp, 0x50+var_40($sp)
61              lw      $a1, 0x50+seconds($sp)
62              lw      $t9, (printf & 0xFFFF)($gp)
63              lui     $a0, ($LC5 >> 16) # "Seconds: %d\n"
64              jalr    $t9
65              la      $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ; branch delay slot
66              lw      $ra, 0x50+var_4($sp)
67              or      $at, $zero ; load delay slot, NOP
68              jr      $ra
69              addiu   $sp, 0x50
70
71              $LC0: .ascii "Year: %d\n"<0>
72              $LC1: .ascii "Month: %d\n"<0>
73              $LC2: .ascii "Day: %d\n"<0>
74              $LC3: .ascii "Hour: %d\n"<0>
75              $LC4: .ascii "Minutes: %d\n"<0>
76              $LC5: .ascii "Seconds: %d\n"<0>
```

这个案例再次演示了延时槽影响人工分析的严重程度。例如，位于第 35 行的“addiu $a1, 1900”指令，把返回值加上 1900。千万别忘记它的执行顺序先于第 34 行的 JALR 指令。

21.3.4 数组替代法

在内存中，结构体是依次排列的一系列数据。为了演示它的这一特色，我们对指令清单 21.8 的程序进行少许修改，以使用数组替代 tm 结构体：

```
#include <stdio.h>
#include <time.h>
void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;
```

```
    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);
    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

注解：指向结构体的指针，实际上指向结构体的第一个元素。

使用 GCC 4.7.3 编译的时候，会看到编译器提示如下所示。

指令清单 21.12 GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [ enabled
↪by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

不过这些问题不影响正常编译，所得可执行程序的具体指令如下所示。

指令清单 21.13 GCC 4.7.3

```
main        proc near

var_30      = dword ptr -30h
var_2C      = dword ptr -2Ch
unix_time   = dword ptr -1Ch
tm_sec      = dword ptr -18h
tm_min      = dword ptr -14h
tm_hour     = dword ptr -10h
tm_mday     = dword ptr -0Ch
tm_mon      = dword ptr -8
tm_year     = dword ptr -4

            push    ebp
            mov     ebp, esp
            and     esp, 0FFFFFFF0h
            sub     esp, 30h
            call    __main
            mov     [esp+30h+var_30], 0 ; arg 0
            call    time
            mov     [esp+30h+unix_time], eax
            lea     eax, [esp+30h+tm_sec]
            mov     [esp+30h+var_2C], eax
            lea     eax, [esp+30h+unix_time]
            mov     [esp+30h+var_30], eax
            call    localtime_r
            mov     eax, [esp+30h+tm_year]
            add     eax, 1900
            mov     [esp+30h+var_2C], eax
            mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
            call    printf
            mov     eax, [esp+30h+tm_mon]
            mov     [esp+30h+var_2C], eax
            mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
            call    printf
            mov     eax, [esp+30h+tm_mday]
            mov     [esp+30h+var_2C], eax
            mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
            call    printf
            mov     eax, [esp+30h+tm_hour]
```

```
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_min]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_sec]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
        call    printf
        leave
        retn
main    endp
```

从汇编代码上看，采用数组的程序与先前使用结构体的程序最终竟会如此一致，以至于我们无法从汇编代码上区分出源程序的区别。

虽然上述程序可以照常运行，但是使用数组替换结构体的做法并不值得推荐。一般来说，“不启用优化编译选项”的编译器通常以代码声明变量的次序，在局部栈里分配变量的空间，但是无法保证每次编译的结果都严丝合缝。

如果使用 GCC 以外的编译器的话，部分编译器可能警告除变量 tm_sec 之外的 tm_year、tm_mon、tm_mday、tm_hour、tm_min 没有被初始化。这是因为编译器并不能分析出这些变量将被 localtime_r()函数赋值。

在这个程序里，结构体各字段都是 int 型数据，所以本例十分直观。如果源程序中结构体的字段都是 16 位 WORD 型数据，且采取了 SYSTEMTIME 那样的数据结构，因为局部变量向 32 位边界对齐的缘故，这将使 GetSystemTime()无法正常赋值。有关结构体的字段封装问题，请参考 21.4 节。

由此可见，结构体就是一连串变量的封装体。在内存中结构体的各字段依次排列。我认为结构体就是语体上的糖块，使其内各个变量像糖分一样粘成一个统一体，以便编译器把它们分配到连续空间里。即使别人可能认为我是编程专家，但是我毕竟不是，所以这种说法很可能不够确切。顺便提一下，早期的（1972 年之前）C 语言不支持结构体 structure。[①]

这个可执行程序完全和前面的程序一样，本书就不演示相关调试过程了。

21.3.5 替换为 32 位 words

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

① 请参见 Dennis M. Ritchie 撰写的<< The development of the c language>>，“SIGPLAN Not”的第 28 章第 3 节:201-208 页，您也可以在作者的网站下载：http://yurichev.com/mirrors/C/dmr-The%20Development%20of%20the%20C%20Language-1993.pdf.。

上述程序把同一地址（指针）的数据分别当作结构体和整型数据、分别进行写和读的访问操作，完全可以正常工作。在当前时间为“23:51:45 26-July-2014”的时刻，内存中的数据如下：

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

这些变量的排列数据，与结构体在源代码中的声明顺序相同。详细内容请参见本书第 21 章第 8 节。

编译而得的程序如下所示。

指令清单 21.14 Optimizing GCC 4.8.1

```
main            proc near
                push    ebp
                mov     ebp, esp
                push    esi
                push    ebx
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; timer
                lea     ebx, [esp+14h]
                call    _time
                lea     esi, [esp+38h]     ;tp
                mov     [esp+4], ebx
                mov     [esp+10h], eax
                lea     eax, [esp+10h]
                mov     [esp], eax         ; timer
                call    _localtime_r
                nop
                lea     esi, [esi+0]       ; nop

loc_80483D8:
; EBX here is pointer to structure, ESI is the pointer to the end of it.
                mov     eax, [ebx]      ; get 32-bit word from array
                add     ebx, 4          ; next field in structure
                mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
                mov     dword ptr [esp], 1
                mov     [esp+0Ch], eax  ; pass value to printf()
                mov     [esp+8], eax    ; pass value to printf()
                call    __printf_chk
                cmp     ebx, esi        ; meet structure end?
                jnz     short loc_80483D8 ; no - load next value then
                lea     esp, [ebp-8]
                pop     ebx
                pop     esi
                pop     ebp
                retn
main            endp
```

栈空间内的数据依次被看作两种数据：先被当作结构体、再被当作数组。

本例表明，我们可以借助指针修改结构体中的各别字段。

本文再次强调：若不是以 hack 目的研究代码，就不必做这种处理。在编写生产环境下的程序时，不建议使用这种处理手段。

21.3.6 替换为字节型数组

更进一步的实验表明，也可以让时间结构体与字节型数组共用一个指针。

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

假如此时的系统时间和 21.3.5 节那个程序同时启动，它肯定会提示“23:51:45 26-July-2014”。需要注意的是，因为采用了小端字节序（参见第 31 章），数权较小的字节反而排在数权较大的字节之前。

指令清单 21.15 Optimizing GCC 4.8.1

```
main            proc near
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; timer
                lea     esi, [esp+14h]
                call    _time
                lea     edi, [esp+38h]  ; struct end
                mov     [esp+4], esi    ; tp
                mov     [esp+10h], eax
                lea     eax, [esp+10h]
                mov     [esp], eax      ; timer
                call    _localtime_r
                lea     esi, [esi+0]    ; NOP
; ESI here is the pointer to structure in local stack. EDI is the pointer to structure end.
loc_8048408:
                xor     ebx, ebx          ; j=0
loc_804840A:
                movzx   eax, byte ptr [esi+ebx] ; load byte
                add     ebx, 1          ; j=j+1
                mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
                mov     dword ptr [esp], 1
                mov     [esp+8], eax    ; pass loaded byte to printf()
                call    __printf_chk
                cmp     ebx, 4
```

```
                jnz     short loc_804840A
; print carriage return character (CR)
                mov     dword ptr [esp], 0Ah ; c
                add     esi, 4
                call    _putchar
                cmp     esi, edi        ; meet struct end?
                jnz     short loc_8048408 ; j=0
                lea     esp, [ebp-0Ch]
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
main            endp
```

21.4 结构体的字段封装

结构体的字段封装方法构成了这种数据类型的一个重要特性。[①]

我们以下面的例子来加以说明。

```
#include <stdio.h>
struct s
{
    char a;
    int b;
    char c;
    int d;
};
void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};
int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

其中有2个单字节型char字段和2个4字节的int型变量。

21.4.1 x86

使用MSVC 2012（启用选项/GS- /Ob0）编译，可得如下所示的代码。

指令清单21.16 MSVC 2012/GS-/O60

```
 1 _tmp$ = -16
 2 _main    PROC
 3     push   ebp
 4     mov    ebp, esp
 5     sub    esp, 16
 6     mov    BYTE PTR _tmp$[ebp], 1       ; set field a
 7     mov    DWORD PTR _tmp$[ebp+4], 2    ; set field b
 8     mov    BYTE PTR _tmp$[ebp+8], 3     ; set field c
 9     mov    DWORD PTR _tmp$[ebp+12], 4   ; set field d
10     sub    esp, 16                      ; allocate place for temporary structure
11     mov    eax, esp
```

① 更多内容请参见 https://en.wikipedia.org/wiki/Data_structure_alignment。

```
12      mov     ecx, DWORD PTR _tmp$[ebp]     ; copy our structure to the temporary one
13      mov     DWORD PTR [eax], ecx
14      mov     edx, DWORD PTR _tmp$[ebp+4]
15      mov     DWORD PTR [eax+4], edx
16      mov     ecx, DWORD PTR _tmp$[ebp+8]
17      mov     DWORD PTR [eax+8], ecx
18      mov     edx, DWORD PTR _tmp$[ebp+12]
19      mov     DWORD PTR [eax+12], edx
20      call    _f
21      add     esp, 16
22      xor     eax, eax
23      mov     esp, ebp
24      pop     ebp
25      ret     0
26 _main   ENDP
27
28 _s$ = 8       ; size = 16
29 ?f@@YAXUs@@@Z PROC      ; f
30      push    ebp
31      mov     ebp, esp
32      mov     eax, DWORD PTR _s$[ebp+12]
33      push    eax
34      movsx   ecx, BYTE PTR _s$[ebp+8]
35      push    ecx
36      mov     edx, DWORD PTR _s$[ebp+4]
37      push    edx
38      movsx   eax, BYTE PTR _s$[ebp]
39      push    eax
40      push    OFFSET $SG3842
41      call    _printf
42      add     esp, 20
43      pop     ebp
44      ret     0
45 ?f@@YAXUs@@@Z ENDP ; f
46 _TEXT    ENDS
```

虽然我们在代码里一次性分配了结构体 tmp，并依次给它的四个字段赋值，但是可执行程序的指令有些不同：它将结构体的指针复制到临时地址（第 10 行指令分配的空间里），然后通过临时的中间变量把结构体的四个值赋值给临时结构体（第 12～19 行的指令），还把指针也复制出来供 f()调用。这主要是因为编译器无法判断 f()函数是否会修改结构体的内容。借助中间变量，编译器可以保证 main()函数里 tmp 结构体的值不受被调用方函数的影响。我们也可以改动源程序、使用指针传递数据，那样编译器生成的汇编指令也基本相同，但是不会再复制数据了。

另外，这个程序里结构体的字段向 4 字节边界对齐。也就是说它的 char 型数据也和 int 型数据一样占 4 字节存储空间。这主要是为了方便 CPU 从内存读取数据，提高读写和缓存的效率。

这样做的缺点是浪费存储空间。

接下来我们启用编译器的/Zp1 (/Zp[n]表示向 *n* 个字节的边界对齐)选项。

使用 MSVC 2012(启用/GS- /Zp1 选项)编译上述程序，可得到如下所示的代码。

指令清单 21.17 MSVC 2012 /GS- /Zp1

```
1 _main    PROC
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 12
5      mov     BYTE PTR _tmp$[ebp], 1        ; set field a
6      mov     DWORD PTR _tmp$[ebp+1], 2     ; set field b
7      mov     BYTE PTR _tmp$[ebp+5], 3      ; set field c
8      mov     DWORD PTR _tmp$[ebp+6], 4     ; set field d
9      sub     esp, 12                       ; allocate place for temporary structure
10     mov     eax, esp
11     mov     ecx, DWORD PTR _tmp$[ebp]     ; copy 10 bytes
12     mov     DWORD PTR [eax], ecx
```

```
13      mov     edx, DWORD PTR _tmp$[ebp+4]
14      mov     DWORD PTR [eax+4], edx
15      mov     cx, WORD PTR _tmp$[ebp+8]
16      mov     WORD PTR [eax+8], cx
17      call    _f
18      add     esp, 12
19      xor     eax, eax
20      mov     esp, ebp
21      pop     ebp
22      ret     0
23 _main    ENDP
24
25 _TEXT    SEGMENT
26 _s$ = 8       ; size = 10
27 ?f@@YAXUs@@@Z PROC      ; f
28      push    ebp
29      mov     ebp, esp
30      mov     eax, DWORD PTR _s$[ebp+6]
31      push    eax
32      movsx   ecx, BYTE PTR _s$[ebp+5]
33      push    ecx
34      mov     edx, DWORD PTR _s$[ebp+1]
35      push    edx
36      movsx   eax, BYTE PTR _s$[ebp]
37      push    eax
38      push    OFFSET $SG3842
39      call    _printf
40      add     esp, 20
41      pop     ebp
42      ret     0
43 ?f@@YAXUs@@@Z ENDP                    ; f
```

经过这种处理之后，结构体只占用 10 字节空间，其中的 char 型数据占用 1 字节。这将提高代码的空间利用效率，不过这样做会同时降低 CPU 的 IO 读取效率。

main()函数里同样使用临时结构体复制了传入参数的信息，再把临时结构体传递给其他函数。不过这 10 字节数据并不是一个字段一个字段地、按变量声明的那样分 4 次复制过去的，编译器分配 3 对 MOV 指令复制它们。为什么不是 4 对赋值指令？这是因为编译器认为，在复制 10 字节数据时，3 个 MOV 指令对的效率，比 4 对 MOV 指令对（按字段赋值）的效率要高。这是编译器常用的优化手段。编译器使用 MOV 指令直接实现(替代)memcpy()函数的情况十分普遍，主要就是因为在复制小型数据时 memcpy()函数没有 MOV 指令的效率高。如需了解这方面详细知识，请参见本书 43.1.5 节。

当然，如果某个结构体被多个源文件、目标文件（object files）调用，那么在编译这些程序时，结构封装格式和数据对其规范(/Zp[n])必须完全匹配。

MSVC 编译器有指定结构体字段对齐标准的/Zp 选项。此外，还可以通过在源文件里设定#pragma pack 的方法来指定这个选项。MSVC 和 GCC 都支持这种代码级的宏指令。[①]

我们回顾一下使用 16 位型数据的结构体 SYSTEMTIME。编译器如何知道要将它的字段向 1 字节边界对齐呢？文件 WinNT.h 有如下声明。

指令清单 21.18 WinNT.h

```
#include "pshpack1.h"
```

而且还有下述声明。

指令清单 21.19 WinNT.h

```
#include "pshpack4.h"//默认情况下进行 4 字节边界对齐
```

① 请参阅 https://msdn.microsoft.com/en-us/library/ms253935.aspx 和 https://gcc.gnu.org/onlinedocs/gcc/Structure-Packing-Pragmas.html。

PshPack1.h 文件有下列内容。

指令清单 21.20 PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

根据#pragma pack 的信息，编译器会在封装结构体时向相应的边界对齐。

OllyDbg + 默认封装格式

现在打开 OllyDbg，加载上述以默认封装格式（4 字节边界对齐）的程序，如图 21.3 所示。

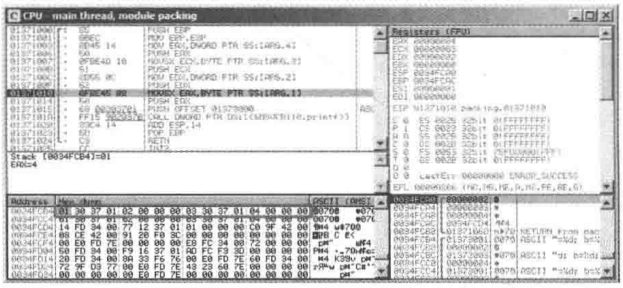


图 21.3 OllyDbg：调用 printf()函数之前

我们在数据窗口可以找到四个字段的值。但是，第一个字段（变量 *a*）和第三个字段（变量 *c*）空间之后的数据工具出现了随机值(0x30,0x37, 0x01)。它们是怎么产生的？在指令清单 21.16 的源程序中，变量 *a* 和变量 *c* 都是 char 型单字节数据。程序的第 6、第 8 行，分别给它们赋值 1、3。它们在内存里占用了 4 个字节，而其他 32 位存储空间里有 3 个字节并没有被赋值！在这 3 个字节空间里的数据，就是随机脏数据。因为在给 printf()函数传递参数时，编译器会使用的是单字节数据赋值指令 MOVSX（参见指令清单 21.16 的第 34 行和第 38 行）传递数据，所以这些脏数据并不会影响 printf()函数的输出结果。

另外，因为 *a* 和 *c* 是 char 型数据，char 型数据属于有符号（signed）型数据，所以复制操作所对应的汇编指令是 MOVSX（SX 是 sign-extending 的缩写）。如果它们是 unsigned char 或 uint8_t 型数据，那么此处就会是 MOVZX 指令。

OllyDbg + 字段向单字节边界对齐

这种情况下，整个结构体的各个变量在内存里依次排列，如图 21.4 所示。

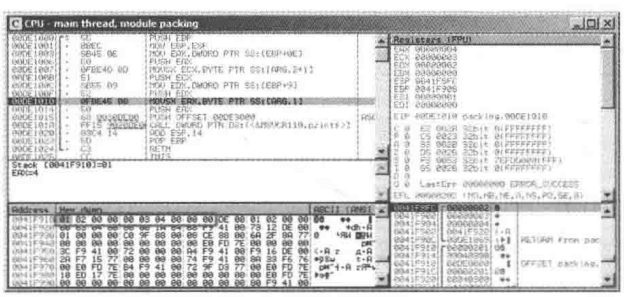


图 21.4 OllyDbg：在调用 printf()函数之前

21.4.2 ARM

Optimizing Keil 6/2013 (Thumb mode)

使用 Keil 6/2013（开启优化选项）、以 Thumb 模式编译上述程序可得如下所示的代码。

指令清单 21.21 Optimizing Keil 6/2013 (Thumb mode)

```
.text:0000003E                exit ; CODE XREF: f+16
.text:0000003E 05 B0                ADD     SP, SP, #0x14
.text:00000040 00 BD                POP     {PC}

.text:00000280                f
.text:00000280
.text:00000280                var_18 = -0x18
.text:00000280                a      = -0x14
.text:00000280                b      = -0x10
.text:00000280                c      = -0xC
.text:00000280                d      =-8
.text:00000280
.text:00000280 0F B5                PUSH    {R0-R3,LR}
.text:00000282 81 B0                SUB     SP, SP, #4
.text:00000284 04 98                LDR     R0, [SP,#16] ; d
.text:00000286 02 9A                LDR     R2, [SP,#8]  ; b
.text:00000288 00 90                STR     R0, [SP]
.text:0000028A 68 46                MOV     R0, SP
.text:0000028C 03 7B                LDRB    R3, [R0,#12] ; c
.text:0000028E 01 79                LDRB    R1, [R0,#4]  ; a
.text:00000290 59 A0                ADR     R0, aADBDCDDD ; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF          BL      __2printf
.text:00000296 D2 E6                B       exit
```

本例向被调用方函数传递的是结构体型数据，不是结构体的指针。因为 ARM 会利用寄存器传递函数所需的前 4 个参数，所以编译器利用 3R0～R3 寄存器向 printf()函数传递结构体的全部字段。

LDRB 从内存中加载 1 个字节并转换为 32 位有符号数据，用来从结构体中读取字段 *a* 和字段 *c*。它相当于 x86 指令集中的 MOVSX 指令。

请注意，在函数退出时，它借用了另一个函数的函数尾声！这种借助 B 指令跳转到其他完全不相干的函数、共用同一个函数尾声的现象，应当是因为两个函数的局部变量的存储空间完全相同。或许因为这两个函数在启动时分配的栈大小相同（都分配了 4×5=0x14 的数据栈），导致退出语句也完全相同的缘故，再加上它们在内存中的地址相近的因素，所以编译器进行了这种处理。确实，使用同一组退出语句不会影响程序的任何功能。这明显是 Keil 编译器出于经济因素而进行的指令复用。JMP 指令只占用 2 个字节，而标准的函数尾声要占用 4 字节。

ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

指令清单 21.22 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```
var_C = -0xC

        PUSH        {R7,LR}
        MOV         R7, SP
        SUB         SP, SP, #4
        MOV         R9,R1;b
        MOV         R1,R0;a
        MOVW        R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
        SXTB        R1, R1 ;制备 a
        MOVT.W      R0, #0
        STR         R3, [SP,#0xC+var_C] ; 推送 d 入栈，供 printf() 调用
        ADD         R0, PC ; 格式化字符串
```

```
SXTB    R3, R2 ; 制备c
MOV     R2,R9;制备b
BLX     _printf
ADD     SP, SP, #4
POP     {R7,PC}
```

SXTB(Signed Extend Byte)对应 x86 的 MOVSX 指令，但是它只能处理寄存器的数据，不能直接对内存进行操作。程序中的其余指令与前例一样，本文不再进行重复说明。

21.4.3 MIPS

指令清单 21.23 Optimizing GCC 4.4.5 (IDA)

```
1 f:
2
3 var_18        = -0x18
4 var_10        = -0x10
5 var_4         = -4
6 arg_0         = 0
7 arg_4         = 4
8 arg_8         = 8
9 arg_C         = 0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15              lui     $gp, (__gnu_local_gp >> 16)
16              addiu   $sp, -0x28
17              la      $gp, (__gnu_local_gp & 0xFFFF)
18              sw      $ra, 0x28+var_4($sp)
19              sw      $gp, 0x28+var_10($sp)
20 ; prepare byte from 32-bit big-endian integer:
21              sra     $t0, $a0, 24
22              move    $v1, $a1
23 ; prepare byte from 32-bit big-endian integer:
24              sra     $v0, $a2, 24
25              lw      $t9, (printf & 0xFFFF)($gp)
26              sw      $a0, 0x28+arg_0($sp)
27              lui     $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d\n"
28              sw      $a3, 0x28+var_18($sp)
29              sw      $a1, 0x28+arg_4($sp)
30              sw      $a2, 0x28+arg_8($sp)
31              sw      $a3, 0x28+arg_C($sp)
32              la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d\n"
33              move    $a1, $t0
34              move    $a2, $v1
35              jalr    $t9
36              move    $a3, $v0 ; branch delay slot
37              lw      $ra, 0x28+var_4($sp)
38              or      $at, $zero ; load delay slot, NOP
39              jr      $ra
40              addiu   $sp, 0x28 ; branch delay slot
41
42 $LC0:        .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>
```

结构体各字段首先被安置于$A0～$A3 寄存器，然后又被重新安放于$A1～$A3 寄存器以传递给 printf()函数。较为特殊的是，上述程序使用了两次 SRA(Shift Word Right Arithmetic)指令，而 SRA 指令用于制备 char 型数据的字段。这是为什么？MIPS 默认采用大端字节序（big-endian，参见本书第 31 章），此外我用的 Debian Linux 也使用大端字节序。当使用 32 位空间存储字节型变量时，数据占用第 31～24 位。因此，当把 char 型数据扩展为 32 位数据时，必须右移 24 位。再加上 char 型数据属于有符号型数据，所以此处必

须用算术位移指令而不能使用逻辑位移指令。

21.4.4　其他

在向被调用方函数传递结构体时（不是传递结构体的指针），传递参数的过程相当于依次传递结构体的各字段。即是说，如果结构体各字段的定义不变，那么 f()函数的源代码可改写为：

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

即使经过上述改动，编译器生成的可执行程序也完全不会发生变化。

21.5　结构体的嵌套

结构体套用另外一个结构体的情况大体如下：

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};
void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

这个程序把结构体 inner_struct 当作另一个结构体 outer_struct 的字段来用，它和 outer_struct 的 a、b、d、e 一样，都是一个字段。

我们使用 MSVC 2010（启用/Ox /Ob0 选项）编译上述程序，可得到如下所示的代码。

指令清单 21.24　Optimizing MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT    SEGMENT
_s$ = 8
_f    PROC
```

```
        mov    eax, DWORD PTR _s$[esp+16]
        movsx  ecx, BYTE PTR _s$[esp+12]
        mov    edx, DWORD PTR _s$[esp+8]
        push   eax
        mov    eax, DWORD PTR _s$[esp+8]
        push   ecx
        mov    ecx, DWORD PTR _s$[esp+8]
        push   edx
        movsx  edx, BYTE PTR _s$[esp+8]
        push   eax
        push   ecx
        push   edx
        push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
        call   _printf
        add    esp, 28
        ret    0
_f      ENDP

_s$ = -24
_main   PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
    pop    ebx
    add    esp, 24
    ret    0
_main   ENDP
```

在汇编代码中，我们找不到内嵌结构体的影子。所以，我们可以断定，嵌套模式的结构体会被编译器展开，最终形成一维结构体。

当然，如果使用“struct iner_struct c”替代源程序中的“struct inter_stuct *c”，汇编指令会大不相同。

OllyDbg

我们使用 OllyDbg 加载上述程序，观测 outer_struct。如图 21.5 所示。

它在内存中的构造如下：

- (outer_struct.a)值为 1 的字节，其后 3 字节是随机脏数据。
- (outer_struct.b) 32 位 word 型数据 2。
- (outer_struct.a)32 位 word 型数据 0x64(100)。

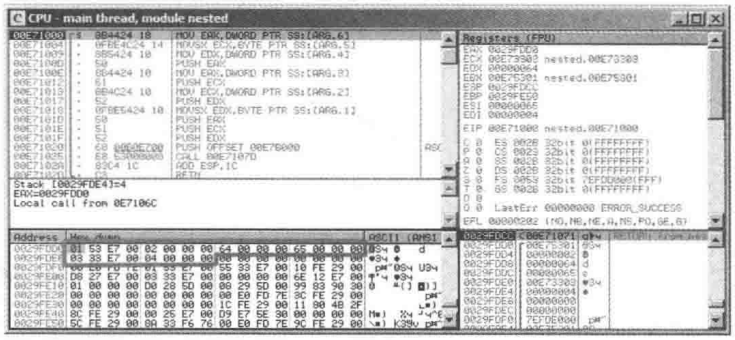


图 21.5 OllyDbg：在执行 printf()之前

- (outer_struct.b)32 位 word 型数据 0x65(101)。
- (outer_struct.d)值为 3 的字节，以及其后 3 字节的脏数据。
- (outer_struct.e)32 位 word 型数据 4。

21.6 结构体中的位操作

21.6.1 CPUID

C/C++语言可以精确操作结构体中的位域。这能够帮助程序员大幅度地节省程序的内存消耗。例如，bool 型数据就需要 1 位空间。但是时间开销和空间效率不可两全，如果要节约内存开销，程序的性能就会下降。

以 CPUID 指令为例。该指令用于获取 CPU 及其特性信息。①

如果在调用指令之前设置 EAX 寄存器的值为 1，那么 CPUID 指令将会按照下列格式在 EAX 寄存器里存储 CPU 的特征信息。

3:0 (4 bits)	Stepping
7:4 (4 bits)	Model
11:8 (4 bits)	Family
13:12(2 bits)	Processor Type
19:16(4 bits)	Extended Model
27:20(8 bits)	Extended Family

MSVC 2010 有 CPUID 宏，而 GCC 4.4.1 没有这个宏。所以，我们自己写一个函数，再用 GCC 编译它。②

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
```

① 请参见 http://en.wikipedia.org/wiki/CPUID。

② 请参见 http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html。

```
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};
int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

在 CPUID 将返回值存储到 EAX/EBX/ECX/EDX 之后，程序使用数组 b[]收集相关信息。然后，我们通过指向结构体 CPUID_1_EAX 的指针，从数组 b[]中获取 EAX 寄存器里的值。

换而言之，我们把 32 位 int 型数据分解为结构体型数据，再从结构体中读取各项数值。

MSVC

使用 MSVC 2008（启用/Ox 选项）编译上述程序，可得如下所示的代码。

指令清单 21.25 Optimizing MSVC 2008

```
b$ = -16 ;size=16
_main   PROC
    sub   esp, 16
    push  ebx
    xor   ecx, ecx
    mov   eax, 1
    cpuid
    push  esi
    lea   esi, DWORD PTR _b$[esp+24]
    mov   DWORD PTR [esi], eax
    mov   DWORD PTR [esi+4], ebx
    mov   DWORD PTR [esi+8], ecx
    mov   DWORD PTR [esi+12], edx

    mov   esi, DWORD PTR _b$[esp+24]
    mov   eax, esi
    and   eax, 15
    push  eax
    push  OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call  _printf

    mov   ecx, esi
```

```
    shr   ecx, 4
    and   ecx, 15
    push  ecx
    push  OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call  _printf

    mov   edx, esi
    shr   edx, 8
    and   edx, 15
    push  edx
    push  OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call  _printf

    mov   eax, esi
    shr   eax, 12
    and   eax, 3
    push  eax
    push  OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call  _printf

    mov   ecx, esi
    shr   ecx, 16
    and   ecx, 15
    push  ecx
    push  OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call  _printf

    shr   esi, 20
    and   esi, 255
    push  esi
    push  OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call  _printf
    add   esp, 48
    pop   esi

    xor   eax, eax
    pop   ebx

    add   esp, 16
    ret   0
_main   ENDP
```

SHR 指令用于过滤位于 EAX 寄存器中右侧那些不参与运算的 bit 位。

而 AND 指令用于过滤位于寄存器左侧且不参与运算的相关位。换句话说，这两条指令用于筛选寄存器的特定 bit 位。

MSVC + OllyDbg

现在使用 OllyDbg 调试这个程序。如图 21.6 所示，在执行 CPUID 之后，EAX/EBX/ECX/EDX 寄存器保存着返回值。

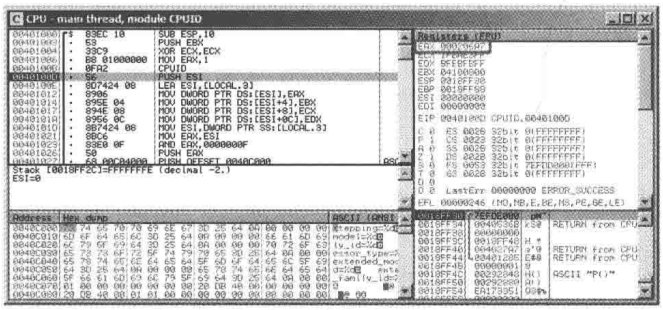


图 21.6　OllyDbg：执行 CPUID 之后的寄存器情况

因为我使用的是 Xeon E3-1200 CPU，所以 EAX 的值是 0x000206A7 。它的二进制数值为 00000000000000010000011010100111。运行结果如图 29.7 所示。

各字段涵义如下表所示。

字段	二进制值	十进制值
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

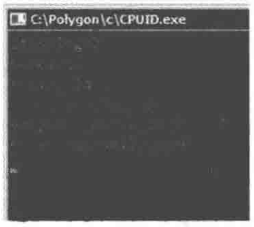


图 21.7 OllyDbg：运行结果

GCC

使用 GCC 4.4.1（启用优化选项–O3）编译上述程序，可得如下所示的代码。

指令清单 21.26 Optimizing GCC 4.4.1

```
main            proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov     dword ptr [esp], 1
    call    __printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    __printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov     dword ptr [esp], 1
    call    __printf_chk
```

```
    mov     eax, esi
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    __printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ; "extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    __printf_chk
    mov     [esp+8], esi
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
    call    __printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

GCC 生成的汇编指令与 MSVC 库函数基本相同。二者的主要区别是：GCC 编译的程序在调用 printf()指令之前把 extended_model_id 和 extended_family_id 放在连续的内存块里一并处理了，而没有像 MSVC 编译的程序那样、在每次调用 printf()之前逐一进行计算。

21.6.2 用结构体构建浮点数

前文已经指出，每个单精度 float 或双精度 double 型浮点数，都由符号、小数和指数三部分组成。到底能否直接利用结构体构建浮点型数据呢？

符号位（第 31 位）	指数部分（23～30 位）	小数部分（0～11 位）

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // 小数
    unsigned int exponent : 8;  // 指数 + 0x3FF
    unsigned int sign : 1;// 符号位
};
float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiply d by 2^n (n here is 2)

    memcpy (&f, &t, sizeof (float));

    return f;
```

```
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

通过结构体构建而成的 float_as_struct 和单精度浮点数据占用相同大小的内存空间，即 32 位/4 字节。在程序里，我们设置符号位为负（1），并把指数增加 2，以进行乘以 4（2 的 n 次方（n=2））的乘法运算。使用 MSVC 2008（不开启任何优化选项）编译上述程序，可得如下所示的代码。

指令清单 21.27 Non-optimizing MSVC 2008

```
_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12
    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - set minus sign
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - drop significand
    and    eax, 255         ; 000000ffH - leave here only exponent
    add    eax, 2           ; add 2 to it
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - shift result to place of bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - drop exponent

; add original value without exponent with new calculated exponent
    or     ecx, eax
    mov    DWORD PTR _t$[ebp], ecx

    push   4
    lea    edx, DWORD PTR _t$[ebp]
    push   edx
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    call   _memcpy
    add    esp, 12

    fld    DWORD PTR _f$[ebp]

    mov    esp, ebp
    pop    ebp
    ret    0
?f@@YAMM@Z ENDP ; f
```

这个程序略微臃肿。如果使用优化选项/Ox 程序就不会调用 memcpy()，转而直接使用变量 f。不过，不启用优化选项而编译出来的代码易于理解。

如果启用 GCC 4.4.1 的-O3 选项又会如何？

指令清单 21.28 Optimizing GCC 4.4.1

```
; f(float)
        public _Z1ff
_Z1ff   proc near

var_4   = dword  ptr -4
arg_0   = dword  ptr 8

        push    ebp
        mov     ebp, esp
        sub     esp, 4
        mov     eax, [ebp+arg_0]
        or      eax, 80000000h  ; set minus sign
        mov     edx, eax
        and     eax, 807FFFFFh  ; leave onlySign and significand in EAX
        shr     edx, 23         ; prepare exponent
        add     edx, 2          ; add 2
        movzx   edx, dl         ; clear all bits except 7:0 in EAX
        shl     edx, 23         ; shift new calculated exponent to its place
        or      eax, edx        ; join new exponent and original value without exponent
        mov     [ebp+var_4], eax
        fld     [ebp+var_4]
        leave
        retn
_Z1ff   endp

        public main
main    proc near
        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        fld     ds:dword_8048614 ; -4.936
        fstp    qword ptr [esp+8]
        mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
        mov     dword ptr [esp], 1
        call    __printf_chk
        xor     eax, eax
        leave
        retn
main    endp
```

f()函数的指令几乎可以自然解释。有趣的是，尽管结构体的各个字段如同大杂烩一样复杂，但是 GCC 能够在编译阶段就计算出函数表达式 f(1.234)的值，并且把这个结果传递给 printf()函数！

21.7 练习题

21.7.1 题目 1

Linux 程序[①]：

请参见 http://beginners.re/exercises/per_chapter/struct_exercise_Linux86.tar

MIPS 程序[②]：

请参见 http://beginners.re/exercises/per_chapter/struct_exercise_MIPS.tar。

这个 Linux x86 程序能够打开文件并在屏幕上打印数字。请问它打印的是什么？

① GCC 4.8.1 -O3。
② GCC 4.4.5-O3。

21.7.2 题目 2

这个函数的输入变量是结构体。请尝试逆向推出结构体的各个字段，不必推敲函数的具体功能。由 MSVC 2010 /Ox 选项编译而得的代码如下所示。

指令清单 21.29 Optimizing MSVC 2010

```
$SG2802 DB      '%f', 0aH, 00H
$SG2803 DB      '%c, %d', 0aH, 00H
$SG2805 DB      'error #2', 0aH, 00H
$SG2807 DB      'error #1', 0aH, 00H

__real@405ec00000000000 DQ 0405ec00000000000r ; 123
__real@407bc00000000000 DQ 0407bc00000000000r ; 444

_s$ = 8
_f          PROC
        push    esi
        mov     esi, DWORD PTR _s$[esp]
        cmp     DWORD PTR [esi], 1000
        jle     SHORT $LN4@f
        cmp     DWORD PTR [esi+4], 10
        jbe     SHORT $LN3@f
        fld     DWORD PTR [esi+8]
        sub     esp, 8
        fmul    QWORD PTR __real@407bc00000000000
        fld     QWORD PTR [esi+16]
        fmul    QWORD PTR __real@405ec00000000000
        faddp   ST(1), ST(0)
        fstp    QWORD PTR [esp]
        push    OFFSET $SG2802 ; '%f'
        call    _printf
        movzx   eax, BYTE PTR [esi+25]
        movsx   ecx, BYTE PTR [esi+24]
        push    eax
        push    ecx
        push    OFFSET $SG2803 ; '%c, %d'
        call    _printf
        add     esp, 24
        pop     esi
        ret     0
$LN3@f:
        pop     esi
        mov     DWORD PTR _s$[esp-4], OFFSET $SG2805 ; 'error #2'
        jmp     _printf
$LN4@f:
        pop     esi
        mov     DWORD PTR _s$[esp-4], OFFSET $SG2807 ; 'error #1'
        jmp     _printf
_f      ENDP
```

指令清单 21.30 Non-optimizing Keil 6/2013 (ARM mode)

```
f  PROC
        PUSH    {r4-r6,lr}
        MOV     r4,r0
        LDR     r0,[r0,#0]
        CMP     r0,#0x3e8
        ADRLE   r0,|L0.140|
        BLE     |L0.132|
        LDR     r0,[r4,#4]
        CMP     r0,#0xa
        ADRLS   r0,|L0.152|
        BLS     |L0.132|
```

```
        ADD     r0,r4,#0x10
        LDM     r0,{r0,r1}
        LDR     r3,|L0.164|
        MOV     r2,#0
        BL      __aeabi_dmul
        MOV     r5,r0
        MOV     r6,r1
        LDR     r0,[r4,#8]
        LDR     r1,|L0.168|
        BL      __aeabi_fmul
        BL      __aeabi_f2d
        MOV     r2,r5
        MOV     r3,r6
        BL      __aeabi_dadd
        MOV     r2,r0
        MOV     r3,r1
        ADR     r0,|L0.172|
        BL      __2printf
        LDRB    r2,[r4,#0x19]
        LDRB    r1,[r4,#0x18]
        POP     {r4-r6,lr}
        ADR     r0,|L0.176|
        B       __2printf
|L0.132|
        POP     {r4-r6,lr}
        B       __2printf
        ENDP
|L0.140|
        DCB     "error #1\n",0
        DCB     0
        DCB     0
|L0.152|
        DCB     "error #2\n",0
        DCB     0
        DCB     0
|L0.164|
        DCB     0x405ec000
|L0.168|
        DCB     0x43de0000
|L0.172|
        DCB     "%f\n",0
|L0.176|
        DCB     "%c, %d\n",0
```

指令清单 21.31 Non-optimizing Keil 6/2013 (Thumb mode)

```
f  PROC
        PUSH    {r4-r6,lr}
        MOV     r4,r0
        LDR     r0,[r0,#0]
        CMP     r0,#0x3e8
        ADRLE   r0,|L0.140|
        BLE     |L0.132|
        LDR     r0,[r4,#4]
        CMP     r0,#0xa
        ADRLS   r0,|L0.152|
        BLS     |L0.132|
        ADD     r0,r4,#0x10
        LDM     r0,{r0,r1}
        LDR     r3,|L0.164|
        MOV     r2,#0
        BL      __aeabi_dmul
        MOV     r5,r0
        MOV     r6,r1
        LDR     r0,[r4,#8]
        LDR     r1,|L0.168|
```

```
        BL      __aeabi_fmul
        BL      __aeabi_f2d
        MOV     r2,r5
        MOV     r3,r6
        BL      __aeabi_dadd
        MOV     r2,r0
        MOV     r3,r1
        ADR     r0,|L0.172|
        BL      __2printf
        LDRB    r2,[r4,#0x19]
        LDRB    r1,[r4,#0x18]
        POP     {r4-r6,lr}
        ADR     r0,|L0.176|
        B       __2printf
|L0.132|
        POP     {r4-r6,lr}
        B       __2printf
        ENDP
|L0.140|
        DCB     "error #1\n",0
        DCB     0
        DCB     0
|L0.152|
        DCB     "error #2\n",0
        DCB     0
        DCB     0
|L0.164|
        DCD     0x405ec000
|L0.168|
        DCD     0x43de0000
|L0.172|
        DCB     "%f\n",0
|L0.176|
        DCB     "%c, %d\n",0
```

指令清单 21.32 Optimizing GCC 4.9 (ARM64)

```
f:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        ldr     w1, [x0]
        str     x19, [sp,16]
        cmp     w1, 1000
        ble     .L2
        ldr     w1, [x0,4]
        cmp     w1, 10
        bls     .L3
        ldr     s1, [x0,8]
        mov     x19, x0
        ldr     s0, .LC1
        adrp    x0, .LC0
        ldr     d2, [x19,16]
        add     x0, x0, :lo12:.LC0
        fmul    s1, s1, s0
        ldr d0, .LC2
        fmul    d0, d2, d0
        fcvt    d1, s1
        fadd    d0, d1, d0
        bl      printf
        ldrb    w1, [x19,24]
        adrp    x0, .LC3
        ldrb    w2, [x19,25]
        add     x0, x0, :lo12:.LC3
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 32
        b       printf
```

```
.L3:
        ldr     x19, [sp,16]
        adrp    x0, .LC4
        ldp     x29, x30, [sp], 32
        add     x0, x0, :lo12:.LC4
        b       puts
.L2:
        ldr     x19, [sp,16]
        adrp    x0, .LC5
        ldp     x29, x30, [sp], 32
        add     x0, x0, :lo12:.LC5
        b       puts
        .size   f, .-f
.LC1:
        .word   1138622464
.LC2:
        .word 0
        .word   1079951360
.LC0:
        .string "%f\n"
.LC3:
        .string "%c, %d\n"
.LC4:
        .string "error #2"
.LC5:
        .string "error #1"
```

指令清单 21.33 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:

var_10          = -0x10
var_8           =-8
var_4           =-4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $s0, 0x20+var_8($sp)
                sw      $gp, 0x20+var_10($sp)
                lw      $v0, 0($a0)
                or      $at, $zero
                slti    $v0, 0x3E9
                bnez    $v0, loc_C8
                move    $s0, $a0
                lw      $v0, 4($a0)
                or      $at, $zero
                sltiu   $v0, 0xB
                bnez    $v0, loc_AC
                lui     $v0, (dword_134 >> 16)
                lwc1    $f4, $LC1
                lwc1    $f2, 8($a0)
                lui     $v0, ($LC2 >> 16)
                lwc1    $f0, 0x14($a0)
                mul.s   $f2, $f4, $f2
                lwc1    $f4, dword_134
                lwc1    $f1, 0x10($a0)
                lwc1    $f5, $LC2
                cvt.d.s $f2, $f2
                mul.d   $f0, $f4, $f0
                lw      $t9, (printf & 0xFFFF)($gp)
                lui     $a0, ($LC0 >> 16)  # "%f\n"
                add.d   $f4, $f2, $f0
                mfc1    $a2, $f5
                mfc1    $a3, $f4
```

```
                jalr    $t9
                la      $a0, ($LC0 & 0xFFFF)  # "%f\n"
                lw      $gp, 0x20+var_10($sp)
                lbu     $a2, 0x19($s0)
                lb      $a1, 0x18($s0)
                lui     $a0, ($LC3 >> 16)  # "%c, %d\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                lw      $s0, 0x20+var_8($sp)
                la      $a0, ($LC3 & 0xFFFF)  # "%c, %d\n"
                jr      $t9
                addiu   $sp, 0x20
loc_AC:                                 # CODE XREF: f+38
                lui     $a0, ($LC4 >> 16)  # "error #2"
                lw      $t9, (puts & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                lw      $s0, 0x20+var_8($sp)
                la      $a0, ($LC4 & 0xFFFF)  # "error #2"
                jr      $t9
                addiu   $sp, 0x20
loc_C8:                                 # CODE XREF: f+24
                lui     $a0, ($LC5 >> 16)  # "error #1"
                lw      $t9, (puts & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                lw      $s0, 0x20+var_8($sp)
                la      $a0, ($LC5 & 0xFFFF)  # "error #1"
                jr      $t9
                addiu   $sp, 0x20

$LC0:           .ascii "%f\n"<0>
$LC3:           .ascii "%c, %d\n"<0>
$LC4:           .ascii "error #2"<0>
$LC5:           .ascii "error #1"<0>

                .data # .rodata.cst4
$LC1:           .word 0x43DE0000

                .data # .rodata.cst8
$LC2:           .word 0x405EC000
dword_134:      .word 0
```

第 22 章　共用体（union）类型

22.1　伪随机数生成程序

我们可以通过不同的方法生成一个介于 0～1 之间的随机浮点数。最简单的做法是：使用 Mersenne twister（马特赛特旋转演算法）之类的 PRNG[①]生成 32 位 DWORD 值，把这个值转换为单精度 float 之后再除以 RAND_MAX（本例中是 0xFFFFFFFF）。这样就可以得到介于 0～1 之间的单精度浮点数。

但是除法的运算速度非常慢。而且从效率的角度考虑，随机函数同样应当尽量少用 FPU 的操作指令。那么，我们是否可以彻底脱离除法运算实现随机函数呢？

单精度浮点数（float）型数据由符号位、有效数字和指数三部分构成。这种数据结构决定，只要随机填充有效数字位就可以生成随机的单精度数！

依据有关规范，随机浮点数的指数部分不可以是 0。那么我们不妨把指数部分直接设置为 01111111（即十进制的 1），用随机数字填充有效数字部分，再把符号位设置为零（表示正数）。瞧！像模像样！这就可以生成一个介于 1～2 之间的随机数。再把它减去 1 就可以得到一个标准的随机函数。

本例利用了线性同余随机数生成随机数的算法[②]。它使用 UNIX 格式的系统时间作为 PRNG 的随机种子，继而生成 32 位数字。

在采用这种算法时，我们可采用共用体类型（union）的数据表示单精度浮点数（float）。这个方法可利用 C/C++ 的数据结构，按照一种与读取方式不同的数据类型存储浮点数据的各个组成部分。本例中，我们创建一个 union 型变量，然后把它当作 float 型或 uint32_t 型数据进行读取。换句话说，这是一种 hack，而且还是比较深度的 hack。

第 20 章的例子介绍过整数型 PRNG 的有关程序。因而本节不再复述其数据格式。

```
include <stdio.h>
#include <stdint.h>
#include <time.h>

// integer PRNG definitions, data and routines:

//constants from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // global variable

void my_srand(uint32_t i)
{
        RNG_state=i;
};
uint32_t my_rand()
{
        RNG_state=RNG_state*RNG_a+RNG_c;
        return RNG_state;
};

// FPU PRNG definitions and routines:

union uint32_t_float
{
        uint32_t i;
```

① Pseudorandom number generator，伪随机数生成函数。

② 参考了网络文章 http://xor0110.wordpress.com/2010/09/24/how-to-generate-floating-point-random-numbers-efficiently。

```
        float f;
};

float float_rand()
{
        union uint32_t_float tmp;
        tmp.i=my_rand() & 0x007fffff | 0x3F800000;
        return tmp.f-1;
};

// test

int main()
{
        my_srand(time(NULL)); // PRNG initialization

        for (int i=0; i<100; i++)
                printf ("%f\n", float_rand());

        return 0;
};
```

22.1.1 x86

使用 MSVC 2010（启用选项/Ox）编译上述程序，可得到如下所示的代码。

指令清单 22.1 Optimizing MSVC 2010

```
$SG4232 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r ;1

tv140 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=pseudorandom value
        and     eax, 8388607       ; 007fffffH
        or      eax, 1065353216    ; 3f800000H
; EAX=pseudorandom value & 0x007fffff | 0x3f800000
; store it into local stack:
        mov     DWORD PTR _tmp$[esp+4], eax
; reload it as float point number:
        fld     DWORD PTR _tmp$[esp+4]
; subtract 1.0:
        fsub    QWORD PTR __real@3ff0000000000000
; store value we got into local stack and reload it:
        fstp    DWORD PTR tv130[esp+4] ; \  these instructions are redundant
        fld     DWORD PTR tv130[esp+4] ; /
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP

_main   PROC
        push    esi
        xor     eax, eax
        call    _time
        push    eax
        call    ?my_srand@@YAXI@Z
        add     esp, 4
        mov     esi, 100
$LL3@main:
        call    ?float_rand@@YAMXZ
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4238
        call    _printf
```

```
        add     esp, 12
        dec     esi
        jne     SHORT $LL3@main
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

在使用 C++编译器编译之后，可执行程序中的函数名称彻底走样。这是编译器对函数的命名规范所决定的。本书 51.1.1 节将详细介绍这种规范。

如果使用 MSVC 2012 编译器进行编译，那么可执行程序则会使用面向 FPU 的 SIMD 指令。本书 27.5 节会对 SMID 指令进行单独介绍。

22.1.2 MIPS

指令清单 22.2　Optimizing GCC 4.4.5

```
float_rand:

var_10          = -0x10
var_4           = -4
                lui         $gp, (__gnu_local_gp >> 16)
                addiu       $sp, -0x20
                la          $gp, (__gnu_local_gp & 0xFFFF)
                sw          $ra, 0x20+var_4($sp)
                sw          $gp, 0x20+var_10($sp)
; call my_rand():
                jal         my_rand
                or          $at, $zero ; branch delay slot, NOP
; $v0=32-bit pseudorandom value
                li          $v1, 0x7FFFFF
; $v1=0x7FFFFF
                and         $v1, $v0, $v1
; $v1=pseudorandom value & 0x7FFFFF
                lui         $a0, 0x3F80
; $a0=0x3F800000
                or          $v1, $a0
; $v1=pseudorandom value & 0x7FFFFF | 0x3F800000
; matter of the following instruction is still hard to get:
                lui         $v0, ($LC0 >> 16)
; load 1.0 into $f0:
                lwc1        $f0, $LC0
; move value from $v1 to coprocessor 1 (into register $f2)
; it behaves like bitwise copy, no conversion done:
                mtc1        $v1, $f2
                lw          $ra, 0x20+var_4($sp)
; subtract 1.0. leave result in $f0:
                sub.s       $f0, $f2, $f0
                jr          $ra
                addiu       $sp, 0x20 ; branch delay slot
main:

var_18          = -0x18
var_10          = -0x10
var_C           = -0xC
var_8           = -8
var_4           = -4
                lui         $gp, (__gnu_local_gp >> 16)
                addiu       $sp, -0x28
                la          $gp, (__gnu_local_gp & 0xFFFF)
                sw          $ra, 0x28+var_4($sp)
                sw          $s2, 0x28+var_8($sp)
                sw          $s1, 0x28+var_C($sp)
                sw          $s0, 0x28+var_10($sp)
                sw          $gp, 0x28+var_18($sp)
                lw          $t9, (time & 0xFFFF)($gp)
                or          $at, $zero ; load delay slot, NOP
```

```
                jalr    $t9
                move    $a0, $zero ; branch delay slot
                lui     $s2, ($LC1 >> 16) # "%f\n"
                move    $a0, $v0
                la      $s2, ($LC1 & 0xFFFF) # "%f\n"
                move    $s0, $zero
                jal     my_srand
                li      $s1, 0x64 # 'd' ; branch delay slot
loc_104:
                jal     float_rand
                addiu   $s0, 1
                lw      $gp, 0x28+var_18($sp)
; convert value we got from float_rand() to double type (printf() need it):
                cvt.d.s $f2, $f0
                lw      $t9, (printf & 0xFFFF)($gp)
                mfc1    $a3, $f2
                mfc1    $a2, $f3
                jalr    $t9
                move    $a0, $s2
                bne     $s0, $s1, loc_104
                move    $v0, $zero
                lw      $ra, 0x28+var_4($sp)
                lw      $s2, 0x28+var_8($sp)
                lw      $s1, 0x28+var_C($sp)
                lw      $s0, 0x28+var_10($sp)
                jr      $ra
                addiu   $sp, 0x28 ; branch delay slot

$LC1:           .ascii "%f\n"<0>
$LC0:           .float 1.0
```

上述程序同样出现了无实际意义的 LUI 指令。17.5.6 节解释过这种情况了，本节不再对其进行复述。

22.1.3 ARM (ARM mode)

指令清单 22.3 Optimizing GCC 4.6.3 (IDA)

```
float_rand
                STMFD   SP!, {R3,LR}
                BL      my_rand
; R0=pseudorandom value
                FLDS    S0, =1.0
; S0=1.0
                BIC     R3, R0, #0xFF000000
                BIC     R3, R3, #0x800000
                ORR     R3, R3, #0x3F800000
; R3=pseudorandom value & 0x007fffff | 0x3f800000
; copy from R3 to FPU (register S15).
; it behaves like bitwise copy, no conversion done:
                FMSR    S15, R3
; subtract 1.0 and leave result in S0:
                FSUBS   S0, S15, S0
                LDMFD   SP!, {R3,PC}
flt_5C          DCFS 1.0

main
                STMFD   SP!, {R4,LR}
                MOV     R0, #0
                BL      time
                BL      my_srand
                MOV     R4, #0x64 ; 'd'
loc_78
                BL      float_rand
; S0=pseudorandom value
                LDR     R0, =aF         ; "%f"
; convert float type value into double type value (printf() will need it):
                FCVTDS  D7, S0
; bitwise copy from D7 into R2/R3 pair of registers (for printf()):
                FMRRD   R2, R3, D7
                BL      printf
```

```
                SUBS        R4, R4, #1
                BNE         loc_78
                MOV         R0, R4
                LDMFD       SP!, {R4,PC}

aF              DCB "%f",0xA,0
```

通过 objdump 工具对上述程序进行分析，可看到 objdump 所显示的 FPU 指令和 IDA 所显示的指令并不一致。这大概是因为 IDA 与 binutils 的研发人员使用的是不同的指令手册。不管原因如何，同一个 FPU 指令会有不同对应名称的情况确实存在。

指令清单 22.4　Optimizing GCC 4.6.3 (objdump)

```
00000038 <float_rand>:
  38:   e92d4008        push    {r3, lr}
  3c:   ebfffffe        bl      10 <my_rand>
  40:   ed9f0a05        vldr    s0, [pc, #20]           ; 5c <float_rand+0x24>
  44:   e3c034ff        bic     r3, r0, #-16777216      ; 0xff000000
  48:   e3c33502        bic     r3, r3, #8388608        ; 0x800000
  4c:   e38335fe        orr     r3, r3, #1065353216     ; 0x3f800000
  50:   ee073a90        vmov    s15, r3
  54:   ee370ac0        vsub.f32        s0, s15, s0
  58:   e8bd8008        pop     {r3, pc}
  5c:   3f800000        svccc   0x00800000

00000000 <main>:
   0:   e92d4010        push    {r4, lr}
   4:   e3a00000        mov     r0, #0
   8:   ebfffffe        bl      0 <time>
   c:   ebfffffe        bl      0 <main>
  10:   e3a04064        mov     r4, #100        ; 0x64
  14:   ebfffffe        bl      38 <main+0x38>
  18:   e59f0018        ldr     r0, [pc, #24]   ; 38 <main+0x38>
  1c:   eeb77ac0        vcvt.f64.f32    d7, s0
  20:   ec532b17        vmov    r2, r3, d7
  24:   ebfffffe        bl      0 <printf>
  28:   e2544001        subs    r4, r4, #1
  2c:   1afffff8        bne     14 <main+0x14>
  30:   e1a00004        mov     r0, r4
  34:   e8bd8010        pop     {r4, pc}
  38:   00000000        andeq   r0, r0, r0
```

在 float_rand()函数里地址 5c 处的指令和 main()函数里地址 38 处的指令都是随机噪音。

22.2　计算机器精度

单精度浮点数的“机器精度/machine epsilon”指的是相对误差的上限，即 FPU 操作的最小值。因此，浮点数的数据位越多，误差越小、精度越高。故而单精度 float 型数据的最高精度为 $2^{-23}=1.19e^{-7}$，而双精度 double 型的最高精度为 $2^{-52}=2.22e^{-16}$。

所以计算某一数值的机器精度是可能的。

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
        uint32_t i;
        float f;
};

float calculate_machine_epsilon(float start)
{
        union uint_float v;
        v.f=start;
        v.i++;
```

```
        return v.f-start;
};
void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

上述程序从 IEEE 754 格式的浮点数中提取小数部分，把这部分当作整数处理并且给它加上 1。运算的中间值是“输入值＋机器精度”。通过测算输入值（单精度型数值）与中间值的差值来测算具体 float 型数据的机器精度。

程序使用 union 型数据结构解析 IEEE 754 格式的 float 型浮点数，并利用这种数据结构把 float 数据的小数部分提取为整数。“1”实际上加到浮点数小数部分中去了。当然，这可能造成溢出，可能把最高位的进位加到浮点数的指数部分。

22.2.1 x86

指令清单 22.5 Optimizing MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
        fld     DWORD PTR _start$[esp-4]
        fst     DWORD PTR _v$[esp-4]  ;冗余代码
        inc     DWORD PTR _v$[esp-4]
        fsubr   DWORD PTR _v$[esp-4]
        fstp    DWORD PTR tv130[esp-4]     ;冗余代码
        fld     DWORD PTR tv130[esp-4]     ;冗余代码
        ret     0
_calculate_machine_epsilon ENDP
```

上述程序的第二个 FST 指令是冗余指令：没有理由把输入值在同一个地址存储两次。这是编译器的问题：它决定把变量 *v* 的存储地址和传递参数所用的栈内地址分配成同一个地址。

接下来的 INC 指令把输入值的小数部分当作整数数据处理。处理结果的中间值再被当作 IEEE 754 型数据传递给 FPU。FSUBR 指令计算差值，最后返回值被存储到 ST0 之中。

程序中最后两条 FSTP/FLD 指令没有实际作用。编译器没能进行相应的优化。

22.2.2 ARM64

把数据类型扩展为 64 位数据的程序如下所示。

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
        uint64_t i;
        double  d;
} uint_double;

double calculate_machine_epsilon(double start)
{
        uint_double v;
        v.d=start;
        v.i++;
        return v.d-start;
}
void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 平台的指令不能直接在 FPU 的 D-字头寄存器里进行加法运算。因此输入值首先从 D0 寄存器复制到 GPR，在那里进行运算并把结果复制给 D1 寄存器，从而在 FPU 里进行减法运算。

指令清单 22.6　Optimizing GCC 4.9 ARM64

```
calculate_machine_epsilon:
        fmov    x0, d0      ; load input value of double type into X0
        add     x0, x0, 1   ; X0++
        fmov    d1, x0      ; move it to FPU register
        fsub    d0, d1, d0 ; subtract
        ret
```

可见上述 x64 程序使用到了 SIMD 指令。有关介绍请参见本书 27.4 节。

22.2.3　MIPS

面向 MIPS 平台的编译器使用 MTC1（Move To Coprocessor 1）指令把 GPR 的数据传递给 FPU 寄存器。

指令清单 22.7　Optimizing GCC 4.4.5 (IDA)

```
calculate_machine_epsilon:
                mfc1    $v0, $f12
                or      $at, $zero ; NOP
                addiu   $v1, $v0, 1
                mtc1    $v1, $f2
                jr      $ra
                sub.s   $f0, $f2, $f12 ; branch delay slot
```

22.2.4　本章小结

本章程序所使用的技巧，在实际程序中出现的几率不大。但是这些技巧可充分演示 IEEE 754 型数据和 C/C++ UNIONS 型数据结构的特点。

22.3　快速平方根计算

浮点数解释为整数的另一个著名的算法，是快速平方根计算。

指令清单 22.8　取自 http://go.yurichev.com/17364 的源代码

```
/* Assumes that float is in the IEEE 754 single precision floating point format
 * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * ((((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m)
     *
     *where
     *
     *b = exponent bias
     * m = number of mantissa bits
     *
     *
     .
     */

    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}
```

作为一个练习，你可以尝试编译这个函数，以理解它是如何工作的。还有一个著名的快速计算 $\frac{1}{\sqrt{x}}$ 的算法。据说这一算法因为用于 QuakeIIIArena 中而变得很流行。

该算法的描述和源代码参见 http://go.yurichev.com/17360。

第23章 函数指针

函数指针和其他指针没有太大区别。它代表的地址为函数代码段的起始地址。

在函数指针（地址）以函数参数的形式传递给另一个函数，继而被用来调用该指针所指向的函数时，这种函数指针的所指向的函数就是“回调函数”/callback function。[①]

此类指针主要应用于：

- 标准 C 函数库里的 qsort()函数和 atexit()函数[②]。
- *NIX 系统里的信号（Signals）[③]。
- 启动线程的 CreateThread()（windows 函数）和 pthread_create()（POSIX 函数）。
- Win32 的多种函数，例如 EnumChildWindows()[④]。
- Linux 内核。例如，Linux 以 callback 的方式调用文件系统的驱动函数：http://lxr.free-electrons.com/source/include/linux/fs.h?v=3.14#L1525。
- GCC 插件：https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API。
- Linux 窗口管理程序定义快捷方式的 dwm 表。当键盘接收到可匹配的特定键时，对应的快捷方式就通过 callback 调用相应函数。请参见 GitHub（https://github.com/cdown/dwm/blob/master/config.def.h#L117），callback 方法要比大量使用 switch()语句的方法更为简单。

其中，qsort()函数是 C/C++编译器函数库自带的快速排序函数。无论待排序的数据是何种数据类型，只要编写出比较两个元素的函数，那么就可以用 qsort()函数以 callback 的方式调用比较函数。

例如，我们可声明比较函数为：

```
int (*compare)(const void *, const void *)
```

我们来看下面这段程序：

```
/* ex3 Sorting ints with qsort */

#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
  const int *a=(const int *)_a;
  const int *b=(const int *)_b;

  if (*a==*b)
    return 0;
  else
    if (*a < *b)
        return -1;
    else
     return 1;
}

int main(int argc, char* argv[])
```

① 又称“回调函数”。确切地说，callback 指的是一种调用方法，而非某个函数，故而本书保留英文名词。请参见 http://en.wikipedia.org/wiki/Callback_(computer_science)。

② 请参见：http://en.wikipedia.org/wiki/Qsort_(C_standard_library)http://pubs.opengroup.org/onlinepubs/009695399/functions/atexit.html。

③ 请参见：http://en.wikipedia.org/wiki/Signal.h。

④ https://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx。

```
21 {
22   int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23   int i;
24
25  /* Sort the array */
26  qsort(numbers,10,sizeof(int),comp) ;
27  for (i=0;i<9;i++)
28    printf("Number = %d\n",numbers[ i ]) ;
29  return 0;
30 }
```

23.1 MSVC

使用 MSVC 2010（启用选项/Ox/GS-/MD）编译上述程序，可得到下述汇编指令（为了凸出重点而进行了精简）。

指令清单 23.1　Optimizing MSVC 2010: /GS- /MD

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_comp   PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     ecx, DWORD PTR _b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     eax, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp ENDP


_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88         ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; fffe7960H
        call    _qsort
        add     esp, 16                                 ; 00000010H
```

...

这个程序没有特殊之处。在传递第四个参数时，传递的是标签_comp 的地址。该地址正是 comp()函数的第一条指令的内存地址。

qsort()函数又是如何调用 comp()函数的？

qsort()函数位于 MSVCR80.DLL（含有 C 函数标准库的 MSVC DLL）中。我们来分析这个文件中的具体指令。

指令清单 23.2 MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const ↙
    ↳ void *, const void *))
.text:7816CBF0                 public _qsort
.text:7816CBF0 _qsort          proc near
.text:7816CBF0
.text:7816CBF0 lo              = dword ptr -104h
.text:7816CBF0 hi              = dword ptr -100h
.text:7816CBF0 var_FC          = dword ptr -0FCh
.text:7816CBF0 stkptr          = dword ptr -0F8h
.text:7816CBF0 lostk           = dword ptr -0F4h
.text:7816CBF0 histk           = dword ptr -7Ch
.text:7816CBF0 base            = dword ptr  4
.text:7816CBF0 num             = dword ptr  8
.text:7816CBF0 width           = dword ptr  0Ch
.text:7816CBF0 comp            = dword ptr  10h
.text:7816CBF0
.text:7816CBF0                 sub     esp, 100h

 ...

.text:7816CCE0 loc_7816CCE0:                           ; CODE XREF: _qsort+B1
.text:7816CCE0                 shr     eax, 1
.text:7816CCE2                 imul    eax, ebp
.text:7816CCE5                 add     eax, ebx
.text:7816CCE7                 mov     edi, eax
.text:7816CCE9                 push    edi
.text:7816CCEA                 push    ebx
.text:7816CCEB                 call    [esp+118h+comp]
.text:7816CCF2                 add     esp, 8
.text:7816CCF5                 test    eax, eax
.text:7816CCF7                 jle     short loc_7816CD04
```

MSVCR80.DLL 中的 comp 参数，是传递给 qsort()函数的第四个参数。在执行 qsort()的过程中，系统会把控制权传递给 comp 参数指向的函数指针的地址。在调用它之前，comp()函数所需的两个参数已经传递到位。在执行它之后，排序已经完成。

可见，函数指针十分危险。首先，如果传递给 qsort()函数的函数指针有误，那么 qsort()函数仍然会把控制权传递给错误的指针地址，届时程序多半将会崩溃，而且人工排错很难发现问题所在。

其次，callback 函数必须严格遵守调用规范。无论是函数不当、参数不当还是数据类型不当，都会引发严重的问题。相比之下，关键问题并不是程序是否会崩溃，而是排查程序崩溃的手段是什么。在编译器处理函数指针的时候，它不会对潜在问题进行任何提示。

23.1.1 MSVC+OllyDbg

我们使用 OllyDbg 加载这个程序，并在首次调用 comp()函数的地址设置断点。

图 23.1 所示为程序首次调用 comp()函数时的情况。OllyDbg 在代码窗口下显示出被比较的两个值。此时 SP 指向 RA，即 qsort()函数的地址（实际上是 MSVCR100.DLL 内部的地址）。

在程序运行完 RETN 指令之前，我们一直按 F8 键，等待它进入 qsort()函数，如图 23.2 所示。

程序将再次调用比较函数。

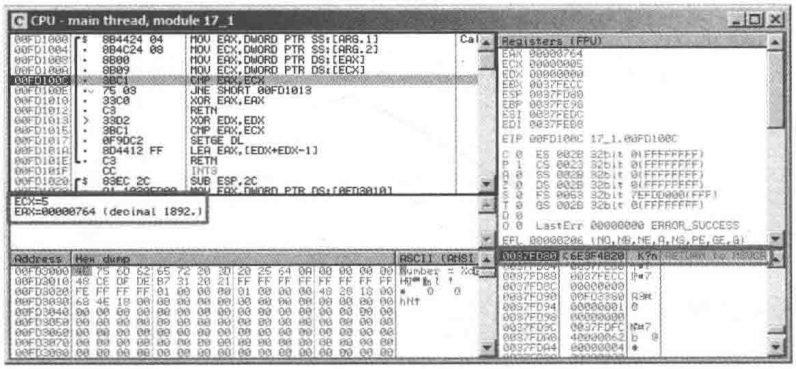


图 23.1　OllyDbg:第一次调用 comp()函数

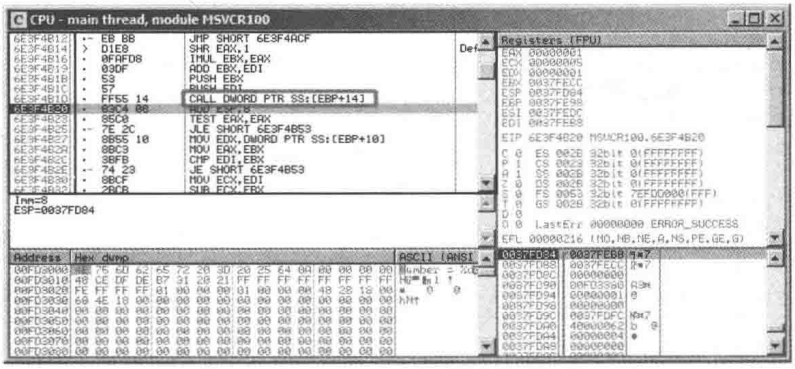


图 23.2　OllyDbg:调用 comp()函数之后返回 qsort()函数

图 23.3 所示的是第二次调用 comp()函数的情形。这一时刻被比较的两个值不相同。

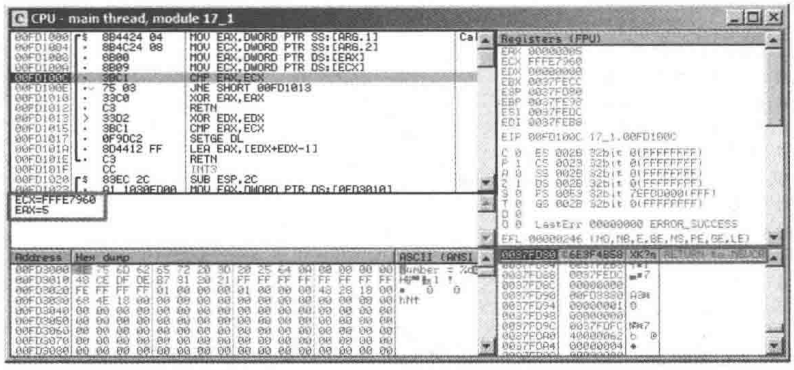


图 23.3　OllyDbg:第二次调用 comp()函数

23.1.2　MSVC+tracer

程序将比较的 10 个数分别是 1892, 45, 200, −98, 4087, 5, −12345, 1087, 88, −100000。

我们在 comp()函数里找到第一个 CMP 指令，它的地址是 0x0040100C。我们在此处设置一个断点：

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

当程序执行到该断点时，各寄存器的情况如下：

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
```

```
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

从中筛选 EAX 和 ECX 寄存器的值：

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xfffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xfffffff9e ECX=0x00000005
EAX=0xfffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xfffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xfffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

得到上面所列的 34 组数据。也就是说，quick sort 算法需要把这 10 个数字进行 34 次比较。

23.1.3 MSVC + tracer（指令分析）

本节利用 tracer 程序收集寄存器里出现过的所有值，稍后在 IDA 里显示它们。

首先使用 tracer 程序追踪 comp()函数里的所有指令：

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

然后再用 IDA 加载刚才生成的.idc-script 文件。如图 23.4 所示。

通过分析 qsort()函数调用的函数指针，IDA 能够显示出相应的函数名称（PtFuncCompare）。

由于数组里存储的是 32 位数据，所以指针 *a* 和指针 *b* 多次指向数组里的不同地方，且它们每次变换之间的地址差是 4 字节。

我们还注意到 0x401010 和 0x401012 处的指令就没有被执行过（所以被标记为白色）。这是因为传递给

comp()函数的值都不相同，函数返回值不会是 0。

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45F7ec..0x45F810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45F7ec..0x45F7f4(step=4), 0x45F7fc
.text:00401008                 mov     eax, [eax]       ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF
.text:0040100A                 mov     ecx, [ecx]       ; [ECX]=5, 0x58, 0xc8, 0x764, 0xFF7, 0xFFFe7960
.text:0040100C                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                              ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43F, 0x764, 0xFF7,
.text:00401017                 setnl   dl               ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                     ; EAX=1, 0xFFFFFFFF
.text:0040101E PtFuncCompare   endp
.text:0040101F
```

图 23.4　tracer 与 IDA 的联动/某些值在屏幕右侧边界之外

23.2　GCC

GCC 的编译方法与 MSVC 的编译方式十分相近。

指令清单 23.3　GCC

```
        lea     eax, [esp+40h+var_28]
        mov     [esp+40h+var_40], eax
        mov     [esp+40h+var_28], 764h
        mov     [esp+40h+var_24], 2Dh
        mov     [esp+40h+var_20], 0C8h
        mov     [esp+40h+var_1C], 0FFFFFF9Eh
        mov     [esp+40h+var_18], 0FF7h
        mov     [esp+40h+var_14], 5
        mov     [esp+40h+var_10], 0FFFFCFC7h
        mov     [esp+40h+var_C], 43Fh
        mov     [esp+40h+var_8], 58h
        mov     [esp+40h+var_4], 0FFFE7960h
        mov     [esp+40h+var_34], offset comp
        mov     [esp+40h+var_38], 4
        mov     [esp+40h+var_3C], 0Ah
        call    _qsort
```

comp()函数对应的代码如下所示。

```
        public comp
comp    proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_4]
        mov     ecx, [ebp+arg_0]
        mov     edx, [eax]
        xor     eax, eax
        cmp     [ecx], edx
        jnz     short loc_8048458
        pop     ebp
        retn
loc_8048458:
```

```
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

qsort()函数的计算过程封装在库文件 libc.so.6 里。因此，Linux 的 qsort()只是 qsort_r()的 wrapper。此处会调用 quicksort()函数，后者再通过函数指针调用我们编写的 comp()函数。

glibc version -2.10.1 中的 lib.so.6 文件有下述指令。

指令清单 23.4 (file libc.so.6, glibc version—2.10.1)

```
.text:0002DDF6              mov     edx, [ebp+arg_10]
.text:0002DDF9              mov     [esp+4], esi
.text:0002DDFD              mov     [esp], edi
.text:0002DE00              mov     [esp+8], edx
.text:0002DE04              call    [ebp+arg_C]
...
```

23.2.1 GCC + GDB（有源代码的情况）

在有源代码的情况下[①]，我们能够针对源代码的行号（第 11 行，第一次调用比较函数）设置断点（b 指令）。这种调试方法有一个前提条件：我们还要在编译源代码时保留调试信息（启用编译选项-g），以便保留地址表和行号之间的对应关系。这样，我们就能根据变量名打印变量的值（p 指令）：调试信息会保留寄存器和（或）数据栈元素与变量之间的对应关系。

我们也可以查看数据栈（bt），并且找出 Glibc 的 msort_with_tmp()函数使用了哪些中间函数。

调试过程如下。

指令清单 23.5 GDB 调试过程

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out
Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11          if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11          if (*a==*b)
```

① 请参阅本章第一个源程序。

```
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d < comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

23.2.2　GCC+GDB（没有源代码的情况）

但是实际情况是，多数情况下我们没有程序的源代码。此时需要反编译 comp()函数（disas 指令），找到第一个 CMP 指令并在该处设置断点。在此之后，我们要查看所有寄存器的值（info registers）。虽然此时还能够查看数据栈（bt），但是所得信息非常有限：程序里没有保存 comp()函数的行号信息。

调试过程如下。

指令清单 23.6　GDB 调试过程

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
   0x08048453 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:     mov    DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:    mov    edx,DWORD PTR [eax]
   0x08048464 <+23>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:    mov    eax,DWORD PTR [eax]
   0x08048469 <+28>:    cmp    edx,eax
   0x0804846b <+30>:    jne    0x8048474 <comp+39>
   0x0804846d <+32>:    mov    eax,0x0
   0x08048472 <+37>:    jmp    0x804848e <comp+65>
   0x08048474 <+39>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:    mov    edx,DWORD PTR [eax]
   0x08048479 <+44>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47>:    mov    eax,DWORD PTR [eax]
   0x0804847e <+49>:    cmp    edx,eax
   0x08048480 <+51>:    jge    0x8048489 <comp+60>
```

```
   0x08048482 <+53>:    mov    eax,0xffffffff
   0x08048487 <+58>:    jmp    0x804848e <comp+65>
   0x08048489 <+60>:    mov    eax,0x1
   0x0804848e <+65>:    leave
   0x0804848f <+66>:    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8     -1073745672
edx            0x764    1892
ebx            0xb7fc0000     -1208221696
esp            0xbfffeeb8     0xbfffeeb8
ebp            0xbfffeec8     0xbfffeec8
esi            0xbffff0fc     -1073745668
edi            0xbffff010     -1073745904
eip            0x8048469      0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x00     0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104     -1073745660
edx            0xffffff9e     -98
ebx            0xb7fc0000     -1208221696
esp            0xbfffee58     0xbfffee58
ebp            0xbfffee68     0xbfffee68
esi            0xbffff108     -1073745656
edi            0xbffff010     -1073745904
eip            0x8048469      0x8048469 <comp+28>
eflags         0x282     [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x00     0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e     -98
ecx            0xbffff100     -1073745664
edx            0xc8     200
ebx            0xb7fc0000     -1208221696
esp            0xbfffeeb8     0xbfffeeb8
ebp            0xbfffeec8     0xbfffeec8
esi            0xbffff104     -1073745660
edi            0xbffff010     -1073745904
eip            0x8048469      0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
```

```
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d < comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main ()
```

第 24 章　32 位系统处理 64 位数据

32 位系统的通用寄存器 GPR 都只能容纳 32 位数据，所以这种平台必须把 64 位数据转换为一对 32 位数据才能进行运算。[①]

24.1　64 位返回值

本节围绕下述程序进行演示。

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

24.1.1　x86

32 位系统用寄存器组合 EDX:EAX 来回传 64 位值。

指令清单 24.1　Optimizing MSVC 2010

```
_f          PROC
            mov     eax, -1867788817        ; 90abcdefH
            mov     edx, 305419896          ; 12345678H
            ret     0
_f          ENDP
```

24.1.2　ARM

ARM 系统用寄存器组合 R0–R1 来回传 64 位值。其中，返回值的高 32 位存储在 R1 寄存器中，低 32 位存储于 R0 寄存器中。

指令清单 24.2　Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
        LDR     r0,|L0.12|
        LDR     r1,|L0.16|
        BX      lr
        ENDP

|L0.12|
        DCD     0x90abcdef
|L0.16|
        DCD     0x12345678
```

24.1.3　MIPS

MIPS 系统使用 V0-V1($2-$3)寄存器对来回传 64 位值。其中，返回值的高 32 位存储在 V0($2)寄存器中，低 32 位存储于$V1($3)寄存器中。

① 16 位系统处理 32 位数据时同样采取了这种拆分数据的处理方法，详情请参见本书 53.4 节。

指令清单 24.3　Optimizing GCC 4.4.5 (assembly listing)

```
        li      $3,-1867841536          # 0xffffffff90ab0000
        li      $2,305397760            # 0x12340000
        ori     $3,$3,0xcdef
        j       $31
        ori     $2,$2,0x5678
```

指令清单 24.4　Optimizing GCC 4.4.5 (IDA)

```
                lui     $v1, 0x90AB
                lui     $v0, 0x1234
                li      $v1, 0x90ABCDEF
                jr      $ra
                li      $v0, 0x12345678
```

24.2　参数传递及加减运算

本节围绕下列程序进行演示。

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

24.2.1　x86

使用 MSVC 2012（启用选项/Ox/Ob1）编译上述程序，可得到如下所示的代码。

指令清单 24.5　Optimizing MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f_add ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
```

```
        push    1942892530      ; 73ce2ff_subH
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436  ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f_sub ENDP
```

在 f1_test()函数中，每个 64 位数据都被拆分为了 2 个 32 位数据。在内存中，高 32 位数据在前（高地址位），低 32 位在后。

加减法运算的处理方法也完全相同。

在进行加法运算时，先对低 32 位相加。如果产生了进位，则设置 CF 标识位。然后 ADC 指令对高 32 位进行运算。如果此时 CF 标识位的值为 1，则再把高 32 位的运算结果加上 1。

减法运算也是分步进行的。第一次的减法运算可能影响 CF 标识位。第二次减法运算会根据 CF 标识位把借位代入计算结果。

在 32 位系统中，函数在返回 64 位数据时都使用 EDX:EAX 寄存器对。当 f_add()函数的返回值传递给 printf()函数时，就可以清楚地观测到这一现象。

使用 GCC 4.8.1（启用选项–O1–fno–inline）编译上述程序，可得到如下所示的代码。

指令清单 24.6 GCC 4.8.1 -O1 -fno-inline

```
_f_add:
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f_add_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889     ; 75939f79H
        mov     DWORD PTR [esp+12], 5461          ; 00001555H
        mov     DWORD PTR [esp], 1942892530       ; 73ce2ff_subH
        mov     DWORD PTR [esp+4], 2874           ; 00000b3aH
        call    _f_add
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0  ; "%lld\12\0"
        call    _printf
        add     esp, 28
        ret

_f_sub:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

GCC 的编译结果和 MSVC 的编译结果相同。

24.2.2 ARM

指令清单 24.7　Optimizing Keil 6/2013 (ARM mode)

```
f_add PROC
        ADDS    r0,r0,r2
        ADC     r1,r1,r3
        BX      lr
        ENDP

f_sub PROC
        SUBS    r0,r0,r2
        SBC     r1,r1,r3
        BX      lr
        ENDP

f_add_test PROC
        PUSH    {r4,lr}
        LDR     r2,|L0.68| ; 0x75939f79
        LDR     r3,|L0.72| ; 0x00001555
        LDR     r0,|L0.76| ; 0x73ce2ff2
        LDR     r1,|L0.80| ; 0x00000b3a
        BL      f_add
        POP     {r4,lr}
        MOV     r2,r0
        MOV     r3,r1
        ADR     r0,|L0.84| ; "%I64d\n"
        B       __2printf
        ENDP

|L0.68|
        DCD     0x75939f79
|L0.72|
        DCD     0x00001555
|L0.76|
        DCD     0x73ce2ff2
|L0.80|
        DCD     0x00000b3a
|L0.84|
        DCB     "%I64d\n",0
```

首个 64 位值被拆分存储到 R0 和 R1 寄存器里，第二个 64 位值则存储于 R2 和 R3 寄存器对。ARM 平台的指令集里有可进行进位加法运算的 ADC 指令和借位减法运算的 SBC 指令。

需要注意的是：在对 64 位数据的低 32 位数据进行加减运算时，需要使用 ADDS 和 SUBS 指令。指令名词中的-S 后缀代表该指令会设置进（借）位标识（Carry Flag）。它们设置过的进借位标识将被高 32 位运算的 ADC/SBC 指令读取并纳入运算结果之中。

没有-S 后缀的 ADD 和 SUB 指令则不会设置借/进位标识位。

24.2.3 MIPS

指令清单 24.8　Optimizing GCC 4.4.5 (IDA)

```
f_add:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
                addu    $v1, $a3, $a1 ; sum up low parts
                addu    $a0, $a2, $a0 ; sum up high parts
; will carry generated while summing up low parts?
; if yes, set $v0 to 1
```

```
                sltu    $v0, $v1, $a3
                jr      $ra
; add 1 to high part of result if carry should be generated:
                addu    $v0, $a0 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_sub:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
                subu    $v1, $a1, $a3 ; subtract low parts
                subu    $v0, $a0, $a2 ; subtract high parts
; will carry generated while subtracting low parts?
; if yes, set $a0 to 1
                sltu    $a1, $v1
                jr      $ra
; subtract 1 from high part of result if carry should be generated:
                subu    $v0, $a1 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_add_test:

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lui     $a1, 0x73CE
                lui     $a3, 0x7593
                li      $a0, 0xB3A
                li      $a3, 0x75939F79
                li      $a2, 0x1555
                jal     f_add
                li      $a1, 0x73CE2FF2
                lw      $gp, 0x20+var_10($sp)
                lui     $a0, ($LC0 >> 16)  # "%lld\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "%lld\n"
                move    $a3, $v1
                move    $a2, $v0
                jr      $t9
                addiu   $sp, 0x20

$LC0:           .ascii "%lld\n"<0>
```

MIPS 处理器没有标识位寄存器。这种平台的加减运算完全不会存储借/进位信息。它的指令集里也没有 x86 指令集里的那种 ADC 或 SBB 指令。当需要存储借/进位信息时，编译器通常使用 SLTU 指令把有关信息（也就是 0 或 1）存储在既定寄存器里，继而在下一步的高数权位运算中纳入借进位信息。

24.3 乘法和除法运算

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
        return a*b;
```

```
};

uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
};
```

24.3.1 x86

使用 MSVC 2012（启用选项/Ox/Ob1）编译上述程序，可得到如下所示的代码。

指令清单 24.9　Optimizing MSVC 2012 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; long long multiplication
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; unsigned long long division
        pop     ebp
        ret     0
_f_div ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
```

```
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; unsigned long long remainder
        pop     ebp
        ret     0
_f_rem  ENDP
```

乘除运算复杂很多。所以编译器通常借助标准库函数来处理乘除运算。

如需了解库函数的各种详细信息，请参见附录 E。

使用 GCC 4.8.1（启用选项-O3-fno-inline）编译上述程序，可得到如下所示的代码。

指令清单 24.10 Optimizing GCC 4.8.1 -fno-inline

```
_f_mul:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f_div:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    __udivdi3 ; unsigned division
        add     esp, 28
        ret

_f_rem:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    __umoddi3 ; unsigned modulo
        add     esp, 28
        ret
```

GCC 把乘法运算进行内部展开处理，大概是它认为这样的效率更高一些。另外它所调用的库函数的名称也和 MSVC 不同（参见附录 D）。除此之外，这段汇编指令和 MSVC 的编译结果之间几乎没有区别。

24.3.2 ARM

在编译 Thumb 模式的程序时，Keil 调用库函数进行仿真运算。

指令清单 24.11　Optimizing Keil 6/2013 (Thumb mode)

```
||f_mul|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_lmul
        POP     {r4,pc}
        ENDP

||f_div|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_uldivmod
        POP     {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_uldivmod
        MOVS    r0,r2
        MOVS    r1,r3
        POP     {r4,pc}
        ENDP
```

相比之下，在编译 ARM 模式的程序时，Keil 能够直接进行 64 位乘法运算。

指令清单 24.12　Optimizing Keil 6/2013 (ARM mode)

```
||f_mul|| PROC
        PUSH    {r4,lr}
        UMULL   r12,r4,r0,r2
        MLA     r1,r2,r1,r4
        MLA     r1,r0,r3,r1
        MOV     r0,r12
        POP     {r4,pc}
        ENDP

||f_div|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_uldivmod
        POP     {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_uldivmod
        MOV     r0,r2
        MOV     r1,r3
        POP     {r4,pc}
        ENDP
```

24.3.3　MIPS

在启用优化选项的情况下编译 MIPS 程序时，GCC 能够直接使用汇编指令进行 64 位乘法运算。但是在进行 64 位除法运算时，编译器还是会用库函数进行处理。

指令清单 24.13　Optimizing GCC 4.4.5 (IDA)

```
f_mul:
                mult    $a2, $a1
                mflo    $v0
                or      $at, $zero  ; NOP
                or      $at, $zero  ; NOP
                mult    $a0, $a3
                mflo    $a0
                addu    $v0, $a0
                or      $at, $zero  ; NOP
```

```
                multu   $a3, $a1
                mfhi    $a2
                mflo    $v1
                jr      $ra
                addu    $v0, $a2

f_div:

var_10          = -0x10
var_4           =-4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lw      $t9, (__udivdi3 & 0xFFFF)($gp)
                or      $at, $zero
                jalr    $t9
                or      $at, $zero
                lw      $ra, 0x20+var_4($sp)
                or      $at, $zero
                jr      $ra
                addiu   $sp, 0x20

f_rem:

var_10          = -0x10
var_4           =-4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lw      $t9, (__umoddi3 & 0xFFFF)($gp)
                or      $at, $zero
                jalr    $t9
                or      $at, $zero
                lw      $ra, 0x20+var_4($sp)
                or      $at, $zero
                jr      $ra
                addiu   $sp, 0x20
```

上述程序中夹杂着大量的 NOP 指令。或许这是因为乘法运算的时间较长，且运算之后需要较长等待时间的缘故；不过这一观点尚无法证实。

24.4 右移

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

24.4.1 x86

指令清单 24.14 Optimizing MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_f      PROC
```

```
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

指令清单 24.15　Optimizing GCC 4.8.1 -fno-inline

```
_f:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
        ret
```

位移运算仍然分为 2 步：第一步处理低 32 位数据，第二步处理高 32 位数据。需要注意的是处理低 32 位数据的指令——SHRD 指令。这个指令不仅可以把 EAX 里的低 32 位数据右移 7 位运算，而且还能从 EDX 寄存器里读取高 32 位中的低 7 位、用它填补到低 32 位数据的高位。如此一来，就可直接使用最普通的 SHR 指令对高 32 位进行位移操作，并用零补充位移产生的空位了。

24.4.2　ARM

ARM 的指令集比 x86 的小，没有 SHRD 之类的指令。因此，Keil 把它拆分为位移和或运算，以进行等效处理。

指令清单 24.16　Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
        LSR     r0,r0,#7
        ORR     r0,r0,r1,LSL #25
        LSR     r1,r1,#7
        BX      lr
        ENDP
```

指令清单 24.17　Optimizing Keil 6/2013 (Thumb mode)

```
||f|| PROC
        LSLS    r2,r1,#25
        LSRS    r0,r0,#7
        ORRS    r0,r0,r2
        LSRS    r1,r1,#7
        BX      lr
        ENDP
```

24.4.3　MIPS

GCC for MIPS 采用了 Keil for Thumb 的编译手段。

指令清单 24.18　Optimizing GCC 4.4.5 (IDA)

```
f:
                sll     $v0, $a0, 25
                srl     $v1, $a1, 7
                or      $v1, $v0, $v1
                jr      $ra
                srl     $v0, $a0, 7
```

24.5　32 位数据转换为 64 位数据

```
#include <stdint.h>
```

```
int64_t f (int32_t a)
{
        return a;
};
```

24.5.1 x86

指令清单 24.19 Optimizing MSVC 2012

```
_a$ = 8
_f        PROC
          mov     eax, DWORD PTR _a$[esp-4]
          cdq
          ret     0
_f        ENDP
```

在这种情况下，我们需要把 32 位有符号数参量扩展为 64 位有符号数。无符号数的转换过程较为直接：高位设置为 0 就万事大吉。但是对于有符号数来说，最高的一位是符号位，不能一概而论地把高位都填零。这里使用了 CDQ 指令进行有符号数的格式转换。它把 EAX 的值扩展为 64 位，将结果存储到 EDX:EAX 寄存器对中。换句话说，CDQ 指令获取 EAX 寄存器中的符号位（最高位），并按照符号位的不同把高 32 位都设为 0 或 1。CDQ 指令的作用和 MOVSX 指令相似。

GCC 生成的汇编指令使用了 inlines 乘法运算。其他部分和 MSVC 的编译结果大体相同。

24.5.2 ARM

指令清单 24.20 Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
        ASR     r1,r0,#31
        BX      lr
        ENDP
```

Keil for ARM 的编译方法与 MSVC 不同：它把参数向右位移 31 位，只用了算术右移指令。MSB 是符号位，而算术位移指令会用符号位填补位移产生的空缺位。所以在执行“ASRr1,r0,#31”指令的时候，如果输入值是负数，那么 R1 就是 0xFFFFFFFF；否则 R1 的值会是零。在生成 64 位值的时候，高 32 位应当存储在 R1 寄存器里。

换句话说，这个指令用最高数权位填充 R1 寄存器以形成 64 位值的高 32 位。

24.5.3 MIPS

GCC for MIPS 的编译方法和 Keil 编译 ARM 程序的方法相同。

指令清单 24.21 Optimizing GCC 4.4.5 (IDA)

```
f:
                sra     $v0, $a0, 31
                jr      $ra
                move    $v1, $a0
```

第 25 章　SIMD

SIMD 意为“单指令流多数据流”，其全称是“Single Instruction, Multiple Data”。顾名思义，这类指令可以一次处理多个数据。在 x86 的 CPU 里，SIMD 子系统和 FPU 都由专用模块实现。

不久之后，x86 CPU 通过 MMX 指令率先整合了 SIMD 的运算功能。支持这种技术的 CPU 里都有 8 个 64 位寄存器，即 MM0～MM7。每个 MMX 寄存器都是 8 字节寄存器，可以容纳 2 个 32 位数据，或者 4 个 16 位数据。使用 SIMD 指令进行操作数的计算时，它可以把 8 个 8 位数据分为 4 组数据同时运算。

平面图像可视为一种由二维数组构成的数据结构。在美工人员调整图像亮度的时候，图像编辑程序得对每个像素的亮度系数进行加减法的运算。简单地说，图像的每个像素都有灰度系数，而且灰度系统是 8 位数据（1 个字节）。因而，每执行一个 SIMD 指令就能够同时调整 8 个像素的灰度（即亮度）。为了满足这种需要，处理器厂商后来专门推出了基于 SIMD 技术的饱和度调整指令。这种饱和度调整指令甚至有越界保护功能，能够避免亮度调整时可能会产生的因子上溢（overflow）和下溢（underflow）等问题。

在刚刚推出 MMX 技术的时候，SIMD 借用了 FPU 的寄存器（别名）。这有利于 FPU 或 MMX 的混合运算。有人说 Intel 这样处理是为了节约晶体管，但是其实际原因更为简单：老式操作系统并不会利用新增的 CPU 寄存器，还是会把数据保存到 FPU 上。所以说，只有同时满足“支持 MMX 技术的 CPU”＋“老式操作系统”＋“利用 MMX 指令集的程序”这 3 个条件，才能充分体现 SIMD 的优势。

SSE 技术是 SIMD 的扩展技术，它把 SIMD 寄存器扩展为 128 位寄存器。支持 SSE 技术的 CPU 已经有了单独的 SIMD 通用寄存器，不再复用 FPU 的寄存器。

AVX 是另外一种 SIMD 技术，它的通用寄存器都是 256 位寄存器。

现在转入正题，看看使用 SIMD 的具体应用。

SIMD 技术当然可用于内存复制（memcpy）和内存比较（memcmp）等用途。

此外它还可用于 DES 的运算。DES（Data Encryption Standard）是分组对称密码算法。它采用了 64 位的分组和 56 位的密钥，将 64 位的输入经过一系列变换得到 64 位的输出。如果在电路的与、或、非门和导线实现 DES 模块，电路规模将会是非常庞大。基于并行分组密码算法的 Bitslice DES[①]应运而生。这种算法可由单指令并行处理技术/SIMD 实现。我们已经知道，x86 平台的 unsigned int 型数据占用 32 位空间。因此，在进行 unsigned int 型数据的 64 位数据和 56 位密钥的演算时，可以以把 64 位中间运算结果和运算密钥都分为 2 个 32 位数据，再进行分步的并行处理。

在 Oracle 存储密码和 hash 的算法就是 DES。为了进行有关演示，您还可以研究一下暴力破解 Oracle RDBMS 密码和 hash 的小程序。这个程序利用了 bitslice DES 算法，针对 SSE2 和 AVX 指令集进行了优化。对其稍加修改，则可用于 128 位/256 位消息模块的并行加密运算。如需查看这个程序的源代码，请参阅 http://conus.info/utils/ops_SIMD/。

25.1　矢量化

矢量化[②]泛指将多个标量数组计算为（转换成）一个矢量数组的技术。进行这种计算时，循环体从输入数组中取值，进行某种运算后生成最终数组。这种算法只对数组中的单个元素进行一次运算。“（并行）矢量化技术”是矢量化处理的并行计算技术。

---

① 请参见 http://www.darkside.com.au/bitslice/。

② Vectorization，又称“向量化”。请参见 http://en.wikipedia.org/wiki/Vectorization_(computer_science)。

矢量化并不是最近才出现的技术。本书的作者在1998年的Cray Y-MP超级计算机系列产品中就见到过“Cray Y-MP EL”的羽量级lite版本。有兴趣的读者可以去他们在线超级计算机博物馆去看看：http://www.cray-cyber.org/。

举例来说，下述循环就采用了矢量化技术：

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

上述代码从数组A和数组B取值，把这两个数组元素进行乘法运算，并把求得的积存储在数组C里。

如果输入数组中的每个元素都是32位int型数据，那么就可以把A的四个元素放在128位的MMX寄存器中，再把数组B的四个元素放在另一个MMX寄存器里，然后通过PMULLD（Multiply Packed Signed Dword Integers and Store Low Result）和PMULHW（Multiply Packed Signed Integers and Store High Result）指令进行并行乘法运算。这可以一次获得4个64位的积。

这样一来，循环体的执行次数不再是1024，而是1024/4。换句话说，程序的性能将提高4倍。

某些编译器具有简单的矢量化自动优化功能。Intel C++编译器就有这样的智能优化功能，详情请参见http://www.intel.com/intelpress/sum_vmmx.htm。

25.1.1 用于加法计算

本节将使用下述程序作为编译对象。

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];
        return 0;
};
```

Intel C++

我们通过下述指令用win32版本的Intel C++编译器编译上述代码：

icl intel.cpp /QaxSSE2 /Faintel.asm /Ox

利用IDA打开上面生成的可执行文件，可看到：

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr  0Ch
ar3    = dword ptr  10h

        push    edi
        push    esi
        push    ebx
        push    esi
        mov     edx, [esp+10h+sz]
        test    edx, edx
        jle     loc_15B
        mov     eax, [esp+10h+ar3]
        cmp     edx, 6
        jle     loc_143
        cmp     eax, [esp+10h+ar2]
        jbe     short loc_36
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
```

```
        lea     ecx, ds:0[edx*4]
        neg     esi
        cmp     ecx, esi
        jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax          ;edi = ar3
        and     edi, 0Fh          ; is ar3 16-byte aligned?
        jz      short loc_9A      ; yes
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
        mov     edx, [ecx+esi*4]
        add     edx, [ebx+esi*4]
        mov     [eax+esi*4], edx
        inc     esi
        cmp     esi, edi
        jb      short loc_C1
        mov     ecx, [esp+10h+var_10]
```

```
        mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
        mov     esi, [esp+10h+ar2]
        lea     esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
        test    esi, 0Fh
        jz      short loc_109   ; yes!
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
        movdqu  xmm1, xmmword ptr [ebx+edi*4]
        movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to XMM0
        paddd   xmm1, xmm0
        movdqa  xmmword ptr [eax+edi*4], xmm1 add edi, 4
        cmp     edi, ecx
        jb      short loc_ED
        jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
        movdqu  xmm0, xmmword ptr [ebx+edi*4]
        paddd   xmm0, xmmword ptr [esi+edi*4]
        movdqa  xmmword ptr [eax+edi*4], xmm0
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
         ; f(int,int *,int *,int *)+164
        cmp     ecx, edx
        jnb     short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp     short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
         ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
         ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
```

```
        pop     esi
        pop     edi
        retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
         ; f(int,int *,int *,int *)+9F
        xor     ecx, ecx
        jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

其中，与 SSE2 有关的指令有：

- MOVDQU（Move Unaligned Double Quadword）是从内存加载 16 字节数据，并复制到 XMM 寄存器的指令。
- PADDD（Add Packed Integers）是对 4 对 32 位数进行加法运算，并把运算结果存储到第一个操作符的指令。此外，该指令不会设置任何标志位，在溢出时只保留运算结果的低 32 位，也不会报错。如果 PADDD 的某个操作数是内存中的某个值，那么这个值的地址必须与 16 字节边界对齐，否则将报错。[①]
- MOVDQA（Move Aligned Double Quadword）的功能 MOVDQU 相同，只是要求存储操作数的内存地址必须向 16 字节边界对齐，否则报错。除了这点区别之外，两个指令完全相同。此外，MOVDQA 的运行速度要比 MOVDQU 快。

所以上述 SSE2 指令的运行条件有 2 个：需要进行加法处理的操作数至少有 4 对；指针 ar3 的地址已经向 16 字节边界对齐。

而且，如果 ar2 的地址也向 16 字节边界对齐，则会执行下述指令：

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

另外，MOVDQU 指令会把 ar2 的值加载到 XMM0 寄存器。虽然这条指令不要求指针地址对齐，但是性能略微慢些：

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to xmm0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

其他代码没有涉及与 SSE2 有关的指令。

GCC

在指定-O3 选项并且启用 SSE2 支持（-msse2 选项）的情况下，GCC 编译器也能够进行简单的矢量化智能处理。[②]

我们使用 GCC 4.4.1 编译上述程序，可得到：

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
```

① 有关地址对齐的详细信息，请参见：http://en.wikipedia.org/wiki/Data_structure_alignment。
② https://gcc.gnu.org/projects/tree-ssa/vectorization.html。

```
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
```

```
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

GCC 生成的代码和 Intel C++十分相似，只是严谨性略差。

25.1.2 用于内存复制

我们回顾一下本书 14.2 节的 memcpy()函数：

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

用 GCC 4.9.1 进行优化编译，可得到如下所示的代码。

指令清单 25.1 Optimizing GCC 4.9.1 x64

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
        test    rdx, rdx
        je      .L41
        lea     rax, [rdi+16]
        cmp     rsi, rax
        lea     rax, [rsi+16]
        setae   cl
        cmp     rdi, rax
        setae   al
        or      cl, al
        je      .L13
        cmp     rdx, 22
        jbe     .L13
        mov     rcx, rsi
        push    rbp
        push    rbx
        neg     rcx
        and     ecx, 15
        cmp     rcx, rdx
        cmova   rcx, rdx
        xor     eax, eax
        test    rcx, rcx
        je      .L4
        movzx   eax, BYTE PTR [rsi]
        cmp     rcx, 1
        mov     BYTE PTR [rdi], al
        je      .L15
        movzx   eax, BYTE PTR [rsi+1]
        cmp     rcx, 2
        mov     BYTE PTR [rdi+1], al
        je      .L16
        movzx   eax, BYTE PTR [rsi+2]
        cmp     rcx, 3
        mov     BYTE PTR [rdi+2], al
        je      .L17
        movzx   eax, BYTE PTR [rsi+3]
        cmp     rcx, 4
        mov     BYTE PTR [rdi+3], al
        je      .L18
        movzx   eax, BYTE PTR [rsi+4]
        cmp     rcx, 5
        mov     BYTE PTR [rdi+4], al
        je      .L19
        movzx   eax, BYTE PTR [rsi+5]
        cmp     rcx, 6
        mov     BYTE PTR [rdi+5], al
        je      .L20
        movzx   eax, BYTE PTR [rsi+6]
        cmp     rcx, 7
        mov     BYTE PTR [rdi+6], al
        je      .L21
        movzx   eax, BYTE PTR [rsi7+]
        cmp     rcx, 8
        mov     BYTE PTR [rdi+7], al
        je      .L22
        movzx   eax, BYTE PTR [rsi+8]
        cmp     rcx, 9
        mov     BYTE PTR [rdi+8], al
        je      .L23
        movzx   eax, BYTE PTR [rsi+9]
        cmp     rcx, 10
```

```
        mov     BYTE PTR [rdi+9], al
        je      .L24
        movzx   eax, BYTE PTR [rsi+10]
        cmp     rcx, 11
        mov     BYTE PTR [rdi+10], al
        je      .L25
        movzx   eax, BYTE PTR [rsi+11]
        cmp     rcx, 12
        mov     BYTE PTR [rdi+11], al
        je      .L26
        movzx   eax, BYTE PTR [rsi+12]
        cmp     rcx, 13
        mov     BYTE PTR [rdi+12], al
        je      .L27
        movzx   eax, BYTE PTR [rsi+13]
        cmp     rcx, 15
        mov     BYTE PTR [rdi+13], al
        jne     .L28
        movzx   eax, BYTE PTR [rsi+14]
        mov     BYTE PTR [rdi+14], al
        mov     eax, 15
.L4:
        mov     r10, rdx
        lea     r9, [rdx-1]
        sub     r10, rcx
        lea     r8, [r10-16]
        sub     r9, rcx
        shr     r8, 4
        add     r8, 1
        mov     r11, r8
        sal     r11, 4
        cmp     r9,14
        jbe     .L6
        lea     rbp, [rsi+rcx]
        xor     r9d, r9d
        add     rcx, rdi
        xor     ebx, ebx
.L7:
        movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
        add     rbx, 1
        movups  XMMWORD PTR [rcx+r9], xmm0
        add     r9, 16
        cmp     rbx, r8
        jb      .L7
        add     rax, r11
        cmp     r10, r11
        je      .L1
.L6:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        lea     rcx, [rax+1]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+1+rax]
        mov     BYTE PTR [rdi+1+rax], cl
        lea     rcx, [rax+2]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+2+rax]
        mov     BYTE PTR [rdi+2+rax], cl
        lea     rcx, [rax+3]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+3+rax]
        mov     BYTE PTR [rdi+3+rax], cl
        lea     rcx, [rax+4]
        cmp     rdx, rcx
```

```
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+4+rax]
        mov     BYTE PTR [rdi+4+rax], cl
        lea     rcx, [rax+5]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+5+rax]
        mov     BYTE PTR [rdi+5+rax], cl
        lea     rcx, [rax+6]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+6+rax]
        mov     BYTE PTR [rdi+6+rax], cl
        lea     rcx, [rax+7]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+7+rax]
        mov     BYTE PTR [rdi+7+rax], cl
        lea     rcx, [rax+8]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+8+rax]
        mov     BYTE PTR [rdi+8+rax], cl
        lea     rcx, [rax+9]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+9+rax]
        mov     BYTE PTR [rdi+9+rax], cl
        lea     rcx, [rax+10]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+10+rax]
        mov     BYTE PTR [rdi+10+rax], cl
        lea     rcx, [rax+11]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+11+rax]
        mov     BYTE PTR [rdi+11+rax], cl
        lea     rcx, [rax+12]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+12+rax]
        mov     BYTE PTR [rdi+12+rax], cl
        lea     rcx, [rax+13]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+13+rax]
        mov     BYTE PTR [rdi+13+rax], cl
        lea     rcx, [rax+14]
        cmp     rdx, rcx
        jbe     .L1
        movzx   edx, BYTE PTR [rsi+14+rax]
        mov     BYTE PTR [rdi+14+rax], dl
.L1:
        pop     rbx
        pop     rbp
.L41:
        rep ret
.L13:
        xor     eax, eax
.L3:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        add     rax, 1
        cmp     rax, rdx
        jne     .L3
        rep ret
```

```
.L28:
        mov     eax, 14
        jmp     .L4
.L15:
        mov     eax, 1
        jmp     .L4
.L16:
        mov     eax, 2
        jmp     .L4
.L17:
        mov     eax, 3
        jmp     .L4
.L18:
        mov     eax, 4
        jmp     .L4
.L19:
        mov     eax, 5
        jmp     .L4
.L20:
        mov     eax, 6
        jmp     .L4
.L21:
        mov     eax, 7
        jmp     .L4
.L22:
        mov     eax, 8
        jmp     .L4
.L23:
        mov     eax, 9
        jmp     .L4
.L24:
        mov     eax, 10
        jmp     .L4
.L25:
        mov     eax, 11
        jmp     .L4
.L26:
        mov     eax, 12
        jmp     .L4
.L27:
        mov     eax, 13
        jmp     .L4
```

25.2 SIMD 实现 strlen()

根据 msdn（http://msdn.microsoft.com/en-us/library/y0dh78ez(VS.80).aspx），在 C/C++源程序中插入特定的宏即可调用 SIMD 指令。在 MSVC 编译器的库文件之中，intrin.h 文件就使用了这种宏。

我们可以把字符串进行分组处理，使用 SIMD 指令实现 strlen()函数。这种算法比常规实现算法快 2～2.5 倍。它每次把 16 个字符加载到 1 个 XMM 寄存器里，然后检测字符串的结束标志——数值为零的结束符。[①]

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
```

① 这种算法借鉴了网上的资料 http://www.strchr.com/sse2_optimised_strlen。

```
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

使用 MSVC 2010（启用/Ox 选项）编译上述程序，可得到如下所示的代码。

指令清单 25.2 Optimizing MSVC 2010

```
_pos$75552 = -4                 ; size = 4
_str$ = 8                       ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    pushebp
    mov     ebp, esp
    and     esp, -16            ; fffffff0H
    mov     eax, DWORD PTR _str$[ebp]
    sub     esp, 12             ; 0000000cH
    push    esi
    mov     esi, eax
    and     esi, -16            ; fffffff0H
    xor     edx, edx
    mov     ecx, eax
    cmp     esi, eax
    je      SHORT $LN4@strlen_sse
    lea     edx, DWORD PTR [eax+1]
    npad    3 ; align next label
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test    eax, eax
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16                         ; 00000010H
    pcmpeqb xmm1, xmm0
    add     edx, 16                         ; 00000010H
```

```
    pmovmskb eax, xmm1
    test     eax, eax
    je       SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf      eax, eax
    mov      ecx, eax
    mov      DWORD PTR _pos$75552[esp+16], eax
    lea      eax, DWORD PTR [ecx+edx]
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
?strlen_sse2@@YAIPBD@Z ENDP                ; strlen_sse2
```

程序首先检查 str 指针是否已经向 16 字节边界对齐。如果这个地址没有对齐，就调用常规的 strlen()函数计算字符串长度。

然后通过 MOVDQA 指令把 16 字节数据加载到 XMM1 寄存器。

部分读者可能会问：为什么不直接使用不要求指针对齐的 MOVDQU 指令？

这种说法似乎有道理：如果指针已经向 16 字节边界对齐了，我们可以使用 MOVDQA 指令；否则，就使用速度慢些的 MOVDQU 指令。

但是本文意在强调：

在 Windows NT 系列操作系统，以及其他一些操作系统里，内存的最小分页单位是 4KB（4096 字节）。虽然每个 win32 程序表面上都有（虚拟）独立的 4GB 内存可以使用，但是实际上部分地址空间受到物理内存和分页方法的限制。如果程序试图访问不存在的内存块，将会引发错误。这是虚拟内存的工作方式决定的，详情请参见 http://en.wikipedia.org/wiki/Page_(computer_memory)。

所以，如果函数一次加载 16 字节数据，可能会跨越虚拟内存的内存块边界。例如说，操作系统可能在地址为 0x00c0000 的物理内存处给这个程序分配了 8192（0x2000）字节的内存块。那么这个程序的内存就在 0x008c0000 与 0x008c1fff 之间。

在这个内存块之后，即位于 0x008c2000 地址以后的物理内存都不可被程序访问。换而言之，操作系统没有给这个程序分配那块内存。如果程序访问这块不可用的内存地址就将引发错误。

我们再来研究一下这个程序。程序可能在内存块的末端存储 5 个字符。这种分配方法是合情合理的。

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

当采用常规算法的 strlen()函数在处理这个字符串时，首先把指针指向“hello”的第一个字节，即地址 0x008c1ff8。strlen()函数会逐字节地读取字符串。在遇到 0x008c1ffd 的 0 字节的时候，函数就结束了。

假如我们编写的 strlen()函数不考虑字符串的地址对齐问题，无论字符串指针地址是否已经向 16 字节边界对齐，每次都读取 16 字节。那么它就可能要从 0x008c1ff8 一直读取到 0x008c2008，这将引发错误。当然，我们应当尽力避免这种错误。

所以仅在字符串地址向 16 字节边界对齐的情况下，才可以使用基于 SIMD 技术的算法。另外，操作系统在进行内存分页时也是向 16 字节边界对齐的，所以在字符串首地址同样向 16 字节边界对齐时，我们的函数不会访问未分配的内存空间。

下面继续介绍我们的函数。

_mm_setzero_si128()：清空 XMM0 寄存器的指令，相当于 pxor xmm0, xmm0。

_mm_load_si128()：MOVDQA 的宏。它从指定地址里加载 16 字节数据到 XMM1 寄存器。

_mm_cmpeq_epi8()：PCMPEQB 的宏。它以字节为单位，比较两个 XMM 专用寄存器的值。如果某个字节相等，该字节返回值为 0xff，否则返回值为 0。举例来说，运行 pcmpeqdb xmm1, xmm0 指令的情况如下表所示。

运行前	XMM1	11223344556677880000000000000000
运行前	XMM0	11ab3444007877881111111111111111
运行后	XMM1	ff0000ff0000ffff0000000000000000

本例先用“pxor xmm0, xmm0”将 xmm0 寄存器清零，然后把从字符串摘取出的 16 字节与 XMM0 寄存器的 16 个 0 相比较。

下一条指令是_mm_movemask_epi8()，即 PMOVVMSKB 指令。这个指令通常与 PCMPEQB 配合使用：

pmovmskb eax, xmm1

在 XMM1 的第一个字节的最高位是 1 的情况下，设置 EAX 寄存器的最高位为 1。也就是说，如果 XMM1 寄存器的值是 0xff，那么 EAX 寄存器的最高位还会被赋值为 1。

它还会在 XMM1 的第二个字节是 0xff 的情况下，设置 EAX 寄存器的次高位为 1。

总而言之，这个指令此处的作用是“判断 XMM1 的每个字节是否是 0xFF”。如果 XMM1 的第 *n* 个字节是 0xFF，则设置 EAX 的第 *n* 位为 1，否则 EAX 的第 *n* 位为 0。

另外，不要忘记我们的算法可能会遇到的实际问题：指针指向的字符串可能包含有效值、结束符和脏数据。例如：

15	14	13	12	11	10	9～3	2	1～0
“h”	“e”	“l”	“l”	“o”	0	脏数据	0	脏数据

一次读取 16 字节到 XMM 寄存器，并将它与 XMM0 里的零进行比较，可能会得到这样的计算结果（从左到右是 LSB 到 MSB 的顺序）：

XMM1: 0000ff00000000000000ff0000000000

也就是说它找到了 2 个以上的零。这是情理之中的结果。

在这种情况下 PMOVMSKB 指令将设置 EAX 为（二进制）：0010000000100000b。

很明显，我们的函数必须以第一个结束符为准，忽略掉脏数据里的结束符。

下一条 BSF（Bit Scan Forward）指令将保留操作符二进制值里的第一个 1，将其余位清零，并把结果存储到第一个操作符里。

在 EAX=0010000000100000b 的情况下，执行“bsf eax, eax”，那么 EAX 将将保留第 5 位的 1（从 0 开始数）。

MSVC 里这条指令对应的宏是_BitScanForward。

代码的其他部分就容易理解了。如果找到了 0 字节，就把它的位置编号累加到计数器里，从而获取字符串长度。

应当注意到 MSVC 编译器优化掉了循环体的部分结构。

Intel Core I7 处理器推出了 SSE 4.2。它提供的新的指令大大简化了字符串的处理。有兴趣的读者可参见：http://www.strchr.com/strcmp_and_strlen_using_sse_4.2。

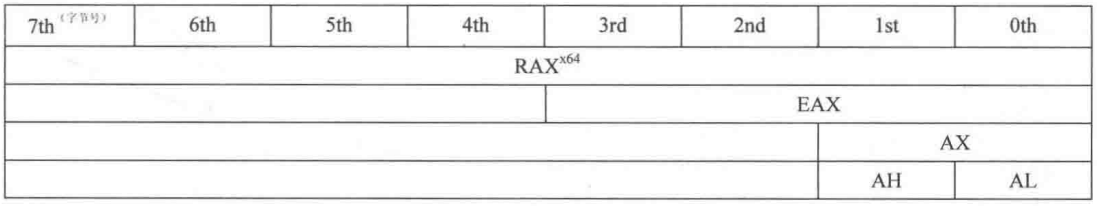
第 26 章　64 位平台

26.1　x86-64

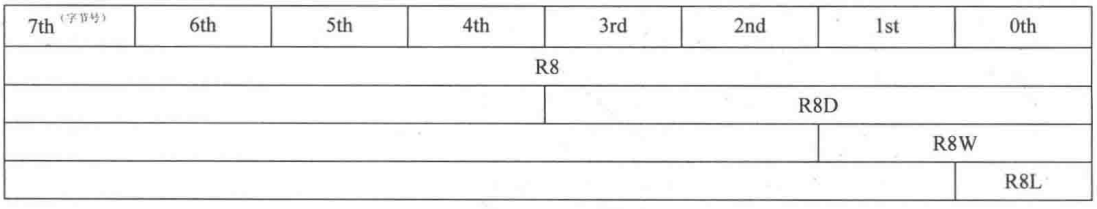
x86-64 框架是一种兼容 x86 指令集的 64 位微处理器架构。

从逆向工程的角度来说，这种框架的主要区别在于：

- 除了位于 FPU 和 SIMD 的寄存器之外，几乎所有的寄存器都变为 64 位寄存器。所有指令都可以通过 R-字头的助记符（寄存器名称）调用 64 位寄存器。而且 x86-64 框架的 CPU 要比 x86 框架的 CPU 多出 8 个通用寄存器。这些通用寄存器分别是：RAX、RBX、RCX、RDX、RBP、RSP、RSI、RDI、R8、R9、R10、R11、R12、R13、R14、R15。
- 它向下兼容，允许程序使用 GPR 的 32 位寄存器的助记符、操作该寄存器的低 32 位数据。例如说，程序可以通过助记符 EAX 访问 RAX 寄存器的低 32 位。



64 位寄存器特有的 R8～R15 寄存器不仅有相应的（低）32 位助记符，即 R8D～R15D（低 32 位），而且还有相应的（低）16 位助记符 R8W～R15W。



x86-64 框架的 CPU 有 16 个 SIMD 寄存器，即 XMM0～XMM15。它的 SIMD 寄存器比 x86 CPU 多出一倍。

- Win64 系统的函数调用约定发生了相应的变化，采取了与 fastcall 规范（请参见本书 64.3 节）相似的参数传递规范。这类程序使用 RCX、RDX、R8、R9 寄存器传递函数所需的前 4 个参数，并用栈传递其余参数。为了保证被调用方函数能够使用前 4 个参数所占的寄存器，调用方函数会单独分配出来 32 字节的栈空间，以便被调用方函数保存 4 个参数寄存器的原始状态。虽然小型函数可能不会用到多少寄存器，但是大型函数可能就要把传入的参数保存到栈里，以尽量充分使用寄存器资源。

System V AMD64 ABI（泛指 Linux、各种 BSD 和 Mac OS X）（参见 Mit13）系统的函数调用约定也与 fastcall 规范相似。它使用 RDI、RSI、RDX、RCX、R8、R9 这 6 个寄存器传递前 6 个参数，再使用栈来传递其余参数。

本书的第 64 章详细介绍了各种调用约定。

- 在 C/C++语言里，x86-64 平台的 int 型数据仍然是 32 位数据。
- x86-64 平台的指针都是 64 位指针。

不过实际情况并非那么理想，x64 CPU 在外部 RAM 的寻址能力只有 48 位。但是存储指针、包括缓存

区的开销却高出 x86 一倍。

因为 64 位平台的寄存器数量是 32 位平台的两倍，编译器在调动资源方面的底气也更足一些，所以它们的寄存器分配方案（register allocation）当然会不同。这就是说，64 位应用程序中的局部变量（栈空间）会更少一些。

第 25 章介绍过一个用 bitslice DES 算法实现并行计算的源程序。我们对其稍加修改，让程序根据 DE_type 数据类型（uint32、uint64、SSE2 或 AVX）处理 32/64/128/256 位数据的 S-Box。

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type      a1,
    DES_type      a2,
    DES_type      a3,
    DES_type      a4,
    DES_type      a5,
    DES_type      a6,
    DES_type      *out1,
    DES_type      *out2,
    DES_type      *out3,
    DES_type      *out4
){
    DES_type      x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type      x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type      x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type      x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type      x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type      x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type      x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
```

```
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

这个源程序里有很多局部变量。当然并非所有的局部变量都要存储到栈里。我们使用 32 位的 MSVC 2008（启用/Ox 选项）编译它，可得到如下所示的代码。

指令清单 26.1　Optimizing MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20              ; size = 4
_x3$ = -16              ; size = 4
_x1$ = -12              ; size = 4
_x8$ = -8               ; size = 4
_x4$ = -4               ; size = 4
_a1$ = 8                ; size = 4
_a2$ = 12               ; size = 4
_a3$ = 16               ; size = 4
_x33$ = 20              ; size = 4
_x7$ = 20               ; size = 4
_a4$ = 20               ; size = 4
_a5$ = 24               ; size = 4
tv326 = 28              ; size = 4
_x36$ = 28              ; size = 4
```

```
_x28$ = 28              ; size = 4
_a6$ = 28               ; size = 4
_out1$ = 32             ; size = 4
_x24$ = 36              ; size = 4
_out2$ = 36             ; size = 4
_out3$ = 40             ; size = 4
_out4$ = 44             ; size = 4
_s1    PROC
    sub    esp, 20                    ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    and    ecx, ebx
    mov    DWORD PTR _x1$[esp+36], eax
    xor    eax, ebx
    mov    esi, ebp
    or     esi, edx
    mov    DWORD PTR _x4$[esp+36], esi
    and    esi, DWORD PTR _a6$[esp+32]
    mov    DWORD PTR _x7$[esp+32], ecx
    mov    edx, esi
    xor    edx, eax
    mov    DWORD PTR _x6$[esp+36], edx
    mov    edx, DWORD PTR _a3$[esp+32]
    xor    edx, ebx
    mov    ebx, esi
    xor    ebx, DWORD PTR _a5$[esp+32]
    mov    DWORD PTR _x8$[esp+36], edx
    and    ebx, edx
    mov    ecx, edx
    mov    edx, ebx
    xor    edx, ebp
    or     edx, DWORD PTR _a6$[esp+32]
    not    ecx
    and    ecx, DWORD PTR _a6$[esp+32]
    xor    edx, edi
    mov    edi, edx
    or     edi, DWORD PTR _a2$[esp+32]
    mov    DWORD PTR _x3$[esp+36], ebp
    mov    ebp, DWORD PTR _a2$[esp+32]
    xor    edi, ebx
    and    edi, DWORD PTR _a1$[esp+32]
    mov    ebx, ecx
    xor    ebx, DWORD PTR _x7$[esp+32]
    not    edi
    or     ebx, ebp
    xor    edi, ebx
    mov    ebx, edi
    mov    edi, DWORD PTR _out2$[esp+32]
    xor    ebx, DWORD PTR [edi]
    not    eax
    xor    ebx, DWORD PTR _x6$[esp+36]
```

```
and     eax, edx
mov     DWORD PTR [edi], ebx
mov     ebx, DWORD PTR _x7$[esp+32]
or      ebx, DWORD PTR _x6$[esp+36]
mov     edi, esi
or      edi, DWORD PTR _x1$[esp+36]
mov     DWORD PTR _x28$[esp+32], ebx
xor     edi, DWORD PTR _x8$[esp+36]
mov     DWORD PTR _x24$[esp+32], edi
xor     edi, ecx
not     edi
and     edi, edx
mov     ebx, edi
and     ebx, ebp
xor     ebx, DWORD PTR _x28$[esp+32]
xor     ebx, eax
not     eax
mov     DWORD PTR _x33$[esp+32], ebx
and     ebx, DWORD PTR _a1$[esp+32]
and     eax, ebp
xor     eax, ebx
mov     ebx, DWORD PTR _out4$[esp+32]
xor     eax, DWORD PTR [ebx]
xor     eax, DWORD PTR _x24$[esp+32]
mov     DWORD PTR [ebx], eax
mov     eax, DWORD PTR _x28$[esp+32]
and     eax, DWORD PTR _a3$[esp+32]
mov     ebx, DWORD PTR _x3$[esp+36]
or      edi, DWORD PTR _a3$[esp+32]
mov     DWORD PTR _x36$[esp+32], eax
not     eax
and     eax, edx
or      ebx, ebp
xor     ebx, eax
not     eax
and     eax, DWORD PTR _x24$[esp+32]
not     ebp
or      eax, DWORD PTR _x3$[esp+36]
not     esi
and     ebp, eax
or      eax, edx
xor     eax, DWORD PTR _a5$[esp+32]
mov     edx, DWORD PTR _x36$[esp+32]
xor     edx, DWORD PTR _x4$[esp+36]
xor     ebp, edi
mov     edi, DWORD PTR _out1$[esp+32]
not     eax
and     eax, DWORD PTR _a2$[esp+32]
not     ebp
and     ebp, DWORD PTR _a1$[esp+32]
and     edx, esi
xor     eax, edx
or      eax, DWORD PTR _a1$[esp+32]
not     ebp
xor     ebp, DWORD PTR [edi]
not     ecx
and     ecx, DWORD PTR _x33$[esp+32]
xor     ebp, ebx
not     eax
mov     DWORD PTR [edi], ebp
xor     eax, ecx
mov     ecx, DWORD PTR _out3$[esp+32]
xor     eax, DWORD PTR [ecx]
pop     edi
pop     esi
```

```
    xor     eax, ebx
    pop     ebp
    mov     DWORD PTR [ecx], eax
    pop     ebx
    add     esp, 20                 ; 00000014H
    ret     0
_s1   ENDP
```

编译器只在栈里存储了 5 个变量。

作为对比，我们再使用 64 位的 MSVC 2008 编译它，得到的代码如下所示。

指令清单 26.2 Optimizing MSVC 2008

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    mov     r15, QWORD PTR a5$[rsp]
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not     rax
    xor     rdx, r9
    not     r10
    mov     r11, rax
    and     rax, r9
    mov     rsi, r10
    mov     QWORD PTR x36$1$[rsp], rax
    and     r11, r8
    and     rsi, r8
    and     r10, r15
    mov     r13, rdx
    mov     rbx, r11
    xor     rbx, r9
    mov     r9, QWORD PTR a2$[rsp]
    mov     r12, rsi
    or      r12, r15
    not     r13
    and     r13, rcx
    mov     r14, r12
    and     r14, rcx
    mov     rax, r14
    mov     r8, r14
    xor     r8, rbx
    xor     rax, r15
    not     rbx
```

```
and     rax, rdx
mov     rdi, rax
xor     rdi, rsi
or      rdi, rcx
xor     rdi, r10
and     rbx, rdi
mov     rcx, rdi
or      rcx, r9
xor     rcx, rax
mov     rax, r13
xor     rax, QWORD PTR x36$1$[rsp]
and     rcx, QWORD PTR a1$[rsp]
or      rax, r9
not     rcx
xor     rcx, rax
mov     rax, QWORD PTR out2$[rsp]
xor     rcx, QWORD PTR [rax]
xor     rcx, r8
mov     QWORD PTR [rax], rcx
mov     rax, QWORD PTR x36$1$[rsp]
mov     rcx, r14
or      rax, r8
or      rcx, r11
mov     r11, r9
xor     rcx, rdx
mov     QWORD PTR x36$1$[rsp], rax
mov     r8, rsi
mov     rdx, rcx
xor     rdx, r13
not     rdx
and     rdx, rdi
mov     r10, rdx
and     r10, r9
xor     r10, rax
xor     r10, rbx
not     rbx
and     rbx, r9
mov     rax, r10
and     rax, QWORD PTR a1$[rsp]
xor     rbx, rax
mov     rax, QWORD PTR out4$[rsp]
xor     rbx, QWORD PTR [rax]
xor     rbx, rcx
mov     QWORD PTR [rax], rbx
mov     rbx, QWORD PTR x36$1$[rsp]
and     rbx, rbp
mov     r9, rbx
not     r9
and     r9, rdi
or      r8, r11
mov     rax, QWORD PTR out1$[rsp]
xor     r8, r9
not     r9
and     r9, rcx
or      rdx, rbp
mov     rbp, QWORD PTR [rsp+80]
or      r9, rsi
xor     rbx, r12
mov     rcx, r11
not     rcx
not     r14
not     r13
and     rcx, r9
or      r9, rdi
and     rbx, r14
xor     r9, r15
xor     rcx, rdx
```

```
    mov     rdx, QWORD PTR a1$[rsp]
    not     r9
    not     rcx
    and     r13, r10
    and     r9, r11
    and     rcx, rdx
    xor     r9, rbx
    mov     rbx, QWORD PTR [rsp+72]
    not     rcx
    xor     rcx, QWORD PTR [rax]
    or      r9, rdx
    not     r9
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR out3$[rsp]
    xor     r9, r13
    xor     r9, QWORD PTR [rax]
    xor     r9, r8
    mov     QWORD PTR [rax], r9
    pop     r15
    pop     r14
    pop     r13
    pop     r12
    pop     rdi
    pop     rsi
    ret     0
s1    ENDP
```

编译器根本没有使用栈。上述代码中的 x36 就是 a5。

顺便说一下，x86-64 CPU 的 GPR 还算不上是最多的。例如，Itanium/安腾处理器就有 128 个 GPR。

26.2 ARM

自 ARM v8 开始，ARM 处理器支持 64 位指令。

26.3 浮点数

有关 64 位平台的浮点数运算，请参见本书第 27 章。

第 27 章　SIMD 与浮点数的并行运算

在 SIMD 功能问世之前，x86 兼容的处理器早就集成了 FPU。自 SSE2 开始，SIMD 拓展指令集提供了单指令多数据浮点计算的指令。这类指令支持的数据格式同样是 IEEE 754。

随着硬件的发展，最近推出的 x86/x86-64 编译器越来越多地使用 SIMD 指令，都不再怎么分配 FPU 指令了。这不得不说是一种好消息。毕竟 SIMD 的指令更为简单。

本章将基于第 17 章的源程序进行讲解。

27.1　样板程序

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

27.1.1　x64

指令清单 27.1　Optimizing MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

程序使用 XMM0～XMM3 传递参数中的前几个浮点数，其余参数都通过栈传递。有关浮点数的参数传递规范，请参见 MSDN：https://msdn.microsoft.com/en-us/library/zthk2dkh.aspx。

变量 *a* 存储在 XMM0 寄存器里，变量 *b* 存储在 XMM1 寄存器里。XMM 系列寄存器都是 128 位寄存器，而双精度 double 型浮点数都是 64 位数据。所以这个程序只使用了寄存器的低半部分。

DIVSD 是 SSE 指令，它的全称是“Divide Scalar Double-Precision Floating-Point Values”，即标量双精度浮点除法。DIVSD 指令以第一个操作数中的低 64 位中的双精度浮点数做被除数，以第二操作数的低 64 位中的双精度浮点数做除数进行除法运算，商将保存到目标地址（第一操作数）的低 64 位中，目标地址的高 64 位保存不变。

在上述程序中可以看到，编译器以 IEEE 754 格式封装浮点数常量。

MULSD 和 ADDSD 分别是乘法和加法的运算指令。

上述函数返回的数据类型是 double，由 XMM0 寄存器回传。

由 MSVC 编译器进行非优化编译，可得如下所示的代码。

指令清单 27.2 MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r  ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr  ; 3.14

a$ = 8
b$ = 16

f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

上述程序还可以进行优化。请注意：传入的参数被保存到了“阴影空间”（可参见本书 8.2.1 节）。但是只有寄存器的低半部分数据——即 double 型 64 位数据会被存在阴影空间里。

GCC 编译出的程序与之相同，本文不再介绍。

27.1.2 x86

在使用 MSVC 2012 编译 x86 平台的可执行程序时，编译器会分配 SSE2 指令。

指令清单 27.3 Non-optimizing MSVC 2012 x86

```
tv70 = -8               ;size=8
_a$ = 8                 ;size=8
_b$ = 16                ;size=8
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        movsd   xmm0, QWORD PTR _a$[ebp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsd   xmm1, QWORD PTR _b$[ebp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        movsd   QWORD PTR tv70[ebp], xmm0
        fld     QWORD PTR tv70[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

指令清单 27.4 Optimizing MSVC 2012 x86

```
tv67 = 8                ; size = 8
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_f      PROC
        movsd   xmm1, QWORD PTR _a$[esp-4]
        divsd   xmm1, QWORD PTR __real@40091eb851eb851f
        movsd   xmm0, QWORD PTR _b$[esp-4]
        mulsd   xmm0, QWORD PTR __real@4010666666666666
        addsd   xmm1, xmm0
        movsd   QWORD PTR tv67[esp-4], xmm1
```

```
        fld     QWORD PTR tv67[esp-4]
        ret     0
_f      ENDP
```

这种 x86 程序与前面介绍的 64 位程序十分相似。它们的区别主要体现在参数的调用规范上：

① x86 程序会像第 17 章的 FPU 例子那样使用栈来传递浮点数，而不是像 64 位程序那样使用 XMM 寄存器传递浮点型参数。

② x86 程序的浮点型数据的运算结果保存在 ST(0)寄存器里——这个值是通过局部变量 tv、从 XMM 寄存器复制到 ST(0)寄存器的。

现在，我们使用 OllyDbg 调试前面那个优化编译的 32 位程序，如图 27.1 到图 27.4 所示。

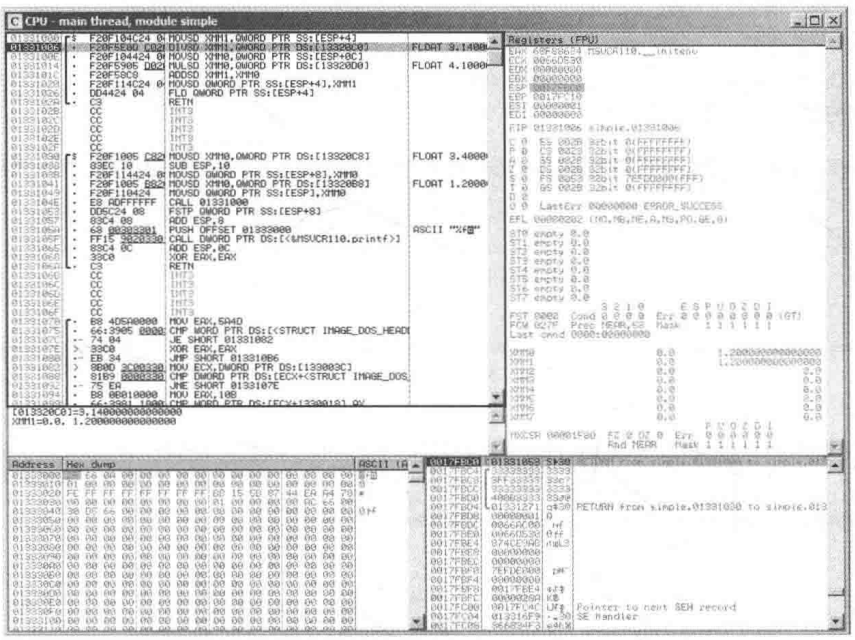


图 27.1　OllyDbg: MOVSD 指令把变量 *a* 传递给 XMM1

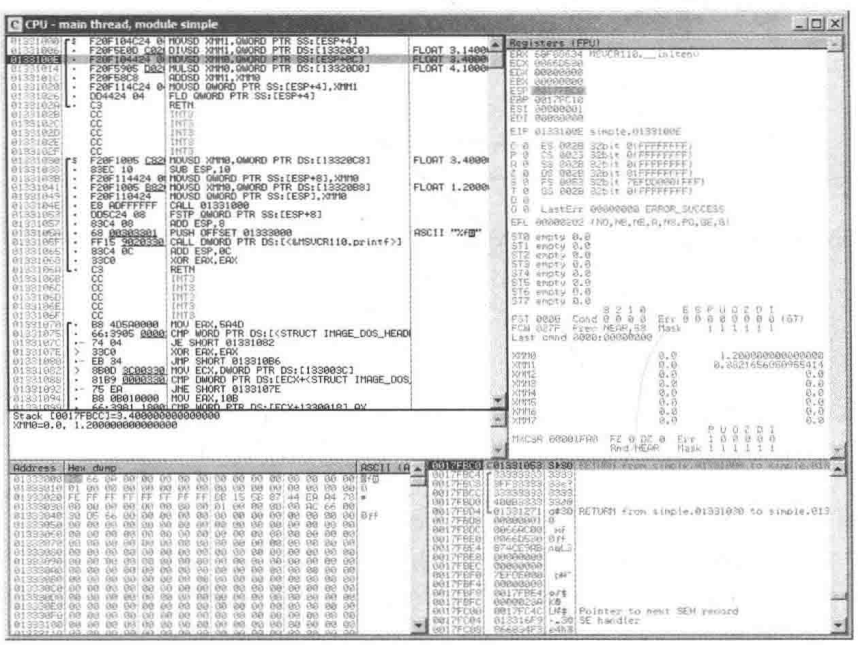


图 27.2　OllyDbg:DIVSD 进行除法运算、把商存储于 XMM1

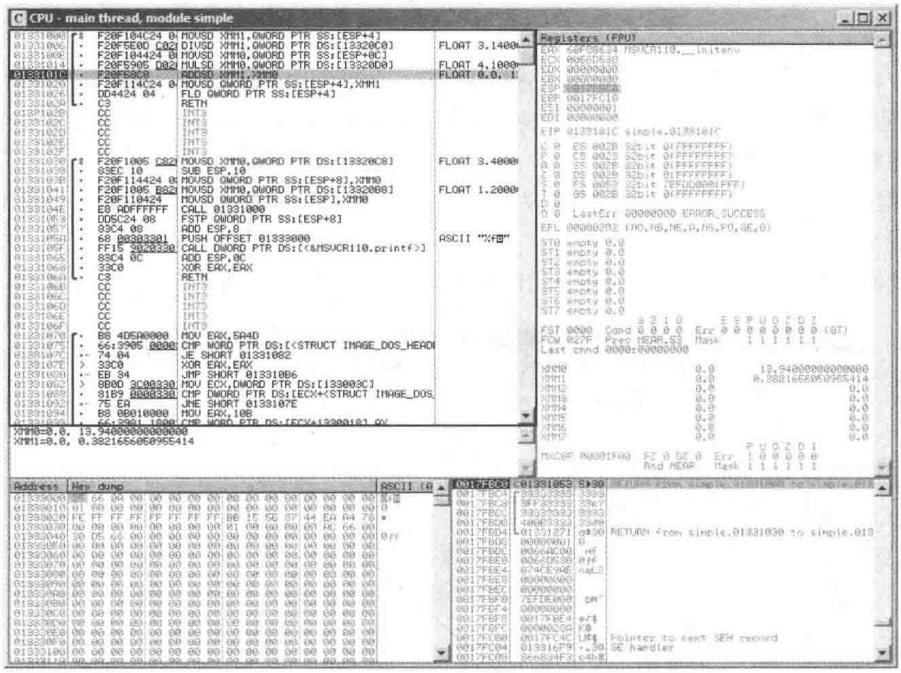


图 27.3 OllyDbg:MULSD 进行乘法运算、把积存储于 XMM0

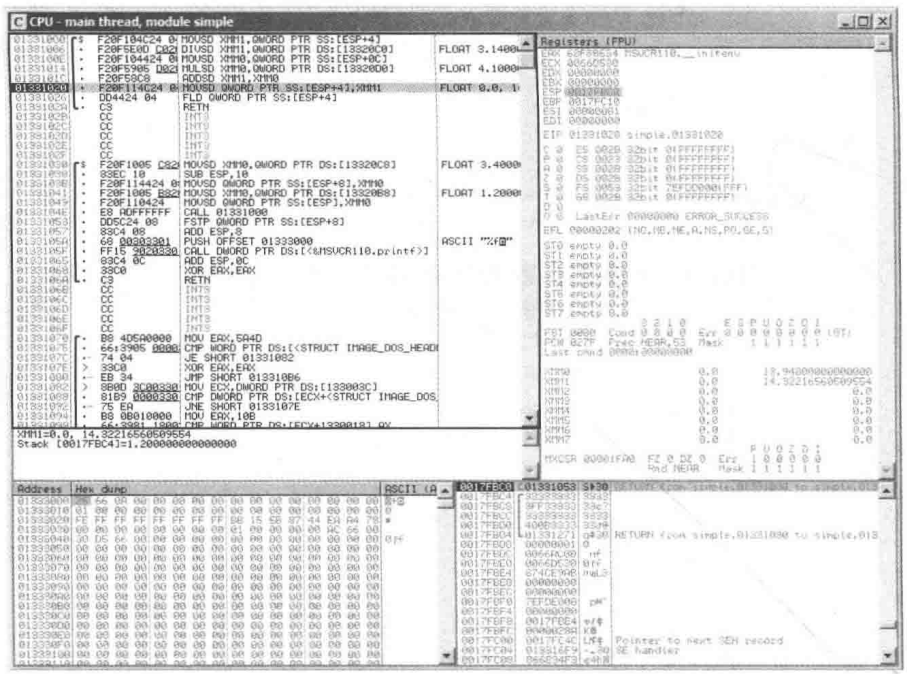


图 27.4 OllyDbg:ADDSD 计算 XMM0 和 XMM1 的和

从图 27.5 可以看到：虽然 OllyDbg 把单个 XMM 寄存器视为一对 double 数据的寄存器，但是它只使用了寄存器的低半部分。很明显，OllyDbg 根据 SSE2 指令的后缀“－SD”判断出了数据类型，并据此进行了数据整理。实际上，OllyDbg 还能够根据 SSE 指令进行判断，把 XMM 寄存器的数据整理为 4 个 float 浮点数、或者是 16 个字节。

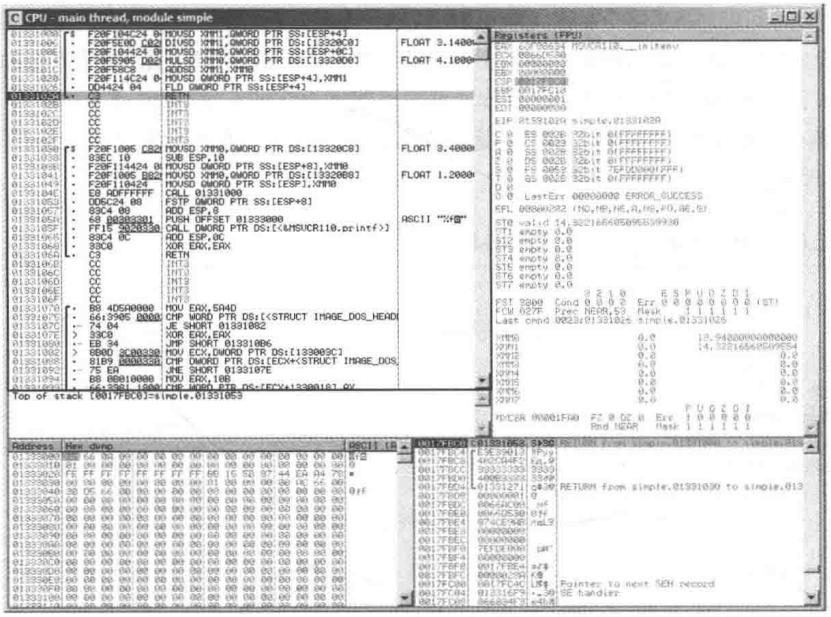


图 27.5　OllyDbg:FLD 把函数返回值传递给 ST（0）

27.2　传递浮点型参数

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
        return 0;
}
```

在函数间负责传递浮点数的寄存器是 XMM0～XMM3 寄存器。

指令清单 27.5　Optimizing MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54

main    PROC
        sub     rsp, 40                                         ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                         ; 00000028H
        ret     0
main    ENDP
```

无论是 Intel 指令白皮书[①]还是 AMD 指令白皮书[②]，都没有记载 MOVSDX 指令。它就是 MOVSD 指令。这就

① 请参阅 Int13。
② 请参阅 AMD13a。

是说 x86 指令集的指令可能对应着多个名称（详情请参阅附录 A.6.2）。而此处出现了这个指令，说明微软的研发人员希望用指令名称来标明操作符的数据类型，避免产生混淆。它是把浮点值传递给 XMM 寄存器的低半部分的指令。

pow()函数从 XMM0 和 XMM1 中读取参数，再把返回结果存储在 XMM0 寄存器。函数返回值又刻意通过 RDX 寄存器传递给 printf()函数，不过这是为什么？坦白地讲，我不知道个中缘由。或许这是 printf()的特性——可受理不同类型参数——造成的。

指令清单 27.6 Optimizing GCC 4.4.6 x64

```
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
main:
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
        call    pow
        ; result is now in XMM0
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 1 ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret

.LC0:
        .long  171798692
        .long  1073259479
.LC1:
        .long  2920577761
        .long  1077936455
```

GCC 的编译手法更易懂一些。它使用 XMM0 寄存器向 printf()函数传递浮点型参数。此外，在向 printf()函数传递参数的时候，程序将 EAX 寄存器的值设置为 1。这就相当于告诉函数“有 1 个参数在矢量寄存器里”。这种特例完全遵循了有关规范（请参阅 Mit13）。

27.3 浮点数之间的比较

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;
        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

27.3.1 x64

指令清单 27.7 Optimizing MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
```

```
        comisd   xmm0, xmm1
        ja       SHORT $LN2@d_max
        movaps   xmm0, xmm1
$LN2@d_max:
        fatret   0
d_max   ENDP
```

MSVC 的优化编译结果简明易懂。

其中，COMISD 的全称是“Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”。它是比较双精度标量并设置标志位的指令。

MSVC 的非优化编译结果同样不难阅读，只是程序的效率低了一些。

指令清单 27.8　MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        movsdx   QWORD PTR [rsp+16], xmm1
        movsdx   QWORD PTR [rsp+8], xmm0
        movsdx   xmm0, QWORD PTR a$[rsp]
        comisd   xmm0, QWORD PTR b$[rsp]
        jbe      SHORT $LN1@d_max
        movsdx   xmm0, QWORD PTR a$[rsp]
        jmp      SHORT $LN2@d_max
$LN1@d_max:
        movsdx   xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret   0
d_max   ENDP
```

然而 GCC 4.4.6 优化得更为彻底。它直接使用 MAXSD 指令（Return Maximum Scalar Double-Precision Floating-Point Value），一步就可完成任务。

指令清单 27.9　Optimizing GCC 4.4.6 x64

```
d_max:
        maxsd   xmm0, xmm1
        ret
```

27.3.2　x86

使用 MSVC 2012 对上述代码进行优化编译，可得到如下所示的代码。

指令清单 27.10　Optimizing MSVC 2012 x86

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_d_max  PROC
        movsd    xmm0, QWORD PTR _a$[esp-4]
        comisd   xmm0, QWORD PTR _b$[esp-4]
        jbe      SHORT $LN1@d_max
        fld      QWORD PTR _a$[esp-4]
        ret      0
$LN1@d_max:
        fld      QWORD PTR _b$[esp-4]
        ret      0
_d_max  ENDP
```

这个程序用栈向函数传递变量 *a* 和 *b*，并且把函数返回值存储在 ST(0)寄存器里。除此之外它和 64 位的程序没有区别。

使用 OllyDbg 调试这个程序，则可以观察 COMISD 指令比较数值、设置/清零 CF/PF 标识位的具体过程，如图 27.6 所示。

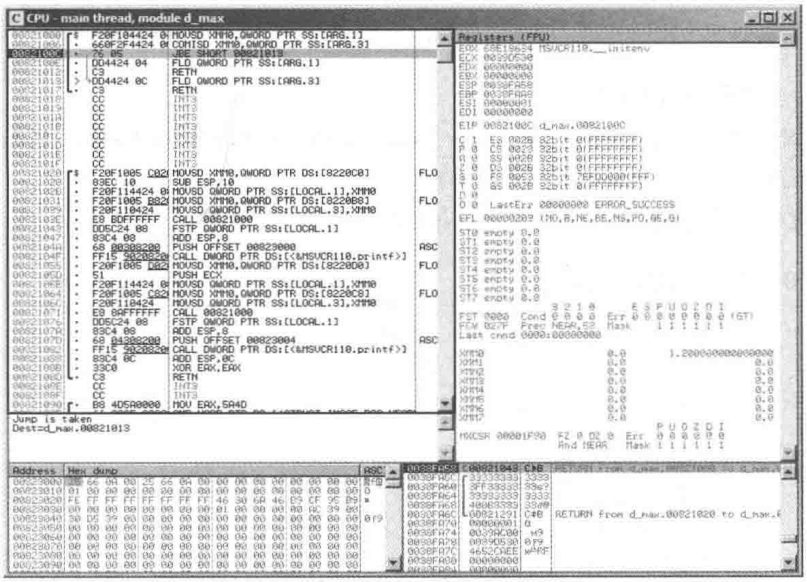


图 27.6 OllyDbg：COMISD 指令调整 CF 和 PF 标识位

27.4 机器精

本书在 22.2.2 节就介绍过机器精度的具体计算算法。在这一节里，我们把它编译为如下所示的 64 位应用程序。

指令清单 27.11 Optimizing MSVC 2012 x64

```
v$ = 8
calculate_machine_epsilon PROC
        movsdx   QWORD PTR v$[rsp], xmm0
        movaps   xmm1, xmm0
        inc      QWORD PTR v$[rsp]
        movsdx   xmm0, QWORD PTR v$[rsp]
        subsd    xmm0, xmm1
        ret      0
calculate_machine_epsilon ENDP
```

由于 INC 指令无法对 128 位的 XMM 寄存器里的值进行操作，程序首先把寄存器的值传递到了内存中的某个地址。

然而这也不是必须如此。如果在此处直接使用 ADDSD（Add Scalar Double-Precision Floating-Point Values）指令，就能够直接对 XMM 寄存器的低 64 位进行加法运算，同时保持寄存器的高 64 位不变。或许，我们不得不说 MSVC 2012 还有改善的余地。

我们再看这个程序。在完成递增运算之后，程序把返回值再次回传给了 XMM 寄存器，然后对 XMM 寄存器进行减法（比较）操作。SUBSD 的全称是“Substract Scalar Double-Precision Floating-Point Values”。它只对 128 位 XMM 寄存器的低 64 位进行操作。最后，减法运算的返回值保存在 XMM0 寄存器中。

27.5 伪随机数生成程序（续）

现在，我们再次利用 MSVC 2012 编译指令清单 22.1 的源程序。这个版本的 MSVC 编译器能够直接分

配 FPU 的 SIMD 指令。

指令清单 27.12　Optimizing MSVC 2012

```
__real@3f800000 DD 03f800000r             ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=pseudorandom value
        and     eax, 8388607                    ; 007fffffH
        or      eax, 1065353216                 ; 3f800000H
; EAX=pseudorandom value & 0x007fffff | 0x3f800000
; store it into local stack:
        mov     DWORD PTR _tmp$[esp+4], eax
; reload it as float point number:
        movss   xmm0, DWORD PTR _tmp$[esp+4]
; subtract 1.0:
        subss   xmm0, DWORD PTR __real@3f800000
; move value to ST0 by placing it in temporary variable...
        movss   DWORD PTR tv128[esp+4], xmm0
; ... and reloading it into ST0:
        fld     DWORD PTR tv128[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP
```

编译器大量使用了带有“-SS”后缀的指令。这个后缀是“Scalar Single”的缩写。“Scalar（标量）”表明寄存器里只有一个值，“Single”则表明该值是单精度 float 型的数据。

27.6　总结

在 XMM 寄存器里存储 IEEE 754 格式格式浮点数的时候，本章的所有程序都只用到了 128 位寄存器空间的低 64 位。

这是由操作指令决定的。本章中的指令都带有-SD（Scalar Double-Precision）后缀。因此，所有指令操作的数据类型都是 IEEE 754 格式的双精度浮点数。这种数据只会占用 XMM 寄存器的低 64 位。

SIMD 指令集比 FPU 的指令更易理解。过去的那些 FPU 指令距离人类语言较远，而且它们还往往涉及 FPU 寄存器的栈结构，远不如 SIMD 指令直观。

如果把本章源程序的数据类型都从 double 改为 float，那么操作浮点数的指令都将改为-SS 后缀（Scalar Single-Precision）。您将看到 MOVSS/COMISS/ADDSS 等指令。

“Scalar”就是“标量”的意思，它表示寄存器里只含有一个值。如果一个寄存器里存有多个值，那么对这些寄存器进行操作的指令的名字里都会带有“packed”字样。

还需要强调的是，SSE2 指令的操作数都是 IEEE 754 格式的 64 位数据，而 FPU 内部则使用 80 位的数据格式存储数据。所以 FPU 的误差较小、精度较高。

第 28 章　ARM 指令详解

28.1　立即数标识（#）

在 Keil 编译器的链接文件里，以及在 IDA 和 objdump 程序显示 ARM 的汇编指令时，立即数（汇编指令中的常量）前面都有“#”标识。然而 GCC 4.9 的编译文件中没有这种立即数标识。请对照一下 14.1.4 节和 39.3 节中的指令清单。

本章列举了很多例子，它们是由多种编译器生成的汇编代码。有些汇编指令的立即数确实没有井号标识，因为它们是 GCC 编译出来的文件。

这种问题毕竟谈不上谁比谁更为正宗。我们也建议读者不必深究这种格式上的问题。

28.2　变址寻址

以 64 位的寻址指令为例：

```
ldr     x0, [x29,24]
```

上述指令从 X29 寄存器中取值，把这个值与 24 相加，再将算术和对应地址的值复制给 X0 寄存器。请注意，X29 和 24 都在方括号之中。如果 24 不在方括号里，指令的涵义就完全不一样了。我们来看：

```
ldr     w4, [x1],28
```

上述指令在 X1 寄存器所表示的地址取值，然后进行“指针 X1 所对应的值+28”的运算。

ARM 可以在取值过程中进行立即数的加减运算。它即可以在取值前进行地址运算，还可以在取值后进行数值运算。

x86 的汇编指令不支持这种操作。但是包括旧式的 PDP-11 平台在内，许多其他处理器平台都支持这种运算。PDP-11 平台的指令集支持前增量（pre-increment）、前减量（pre-decrement）、后增量（post-increment）、后减量（post-decrement）运算。这大体可以归咎于 C 语言（基于 PDP-11 平台开发的）里的*ptr++、*++ptr、*ptr--、*--ptr 一类的指令。在 C 语言中，这些指令确实属于不易掌握的指令。本文将之整理如下。

C 术语	ARM 术语	C 语句	工作原理
后增量	后变址寻址	*ptr++	使用*ptr 的值后，再进行递增运算
后减量	后变址寻址	*ptr--	使用*ptr 的值后，再进行递减运算
前增量	前变址寻址	*++ptr	ptr 先递增，后取*ptr 的值
前减量	前变址寻址	*--ptr	ptr 先递减，后取*ptr 的值

前变址寻址的指令带感叹号标识。例如，指令清单 3.15 的第二条指令“stp x29, x30, [sp,#-16]!”。

Dennis Ritchie（C 语言的作者之一）曾经提到过，变址寻址是 Ken Thompson（另一位 C 语言的作者）根据 PDP-7 平台的特性开发的指令（请参阅［Rit86］[Rit93]）。现在 C 语言编译器仍然保持着这种兼容性，只要硬件处理器支持，编译器就可以进行相应的优化编译。

变址寻址的优越性集中体现在数组的操作方面。

28.3　常量赋值

28.3.1　32 位 ARM

所有的 Thumb 模式的汇编指令都是 2 字节指令，所有的 ARM 模式的汇编指令都是 4 字节指令。ARM 模式指令和 32 位的立即数都占用 4 字节，又如何进行 32 位立即数赋值呢？

我们来看：

```
unsigned int f()
{
        return 0x12345678;
};
```

指令清单 28.1　GCC 4.6.3 -O3 ARM mode

```
f:
        ldr     r0, .L2
        bx      lr
.L2:
        .word   305419896 ; 0x12345678
```

可见，0x12345678 存储于内存之中，供其他指令调用。但是这种指令增加了 CPU 访问内存的次数。我们可以改善这一状况。

指令清单 28.2　GCC 4.6.3 -O3 -march=armv7-a (ARM mode)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660       ; 0x1234
bx      lr
```

上述指令把一个立即数分为 2 个部分，并依次存储到同一个寄存器里。它使用 MOVW 指令先把立即数的低位部分存储到寄存器里，然后再使用 MOVT 存储这个数的高位部分。

这也就是说，ARM 模式的程序需要 2 个指令才能加载一个 32 位的立即数。除了 0 和 1 之外，程序很少用到常量，所以这种分步赋值的方法不会造成实际问题。有人会问，这种一分为二的做法会不会造成性能的下降呢？虽然肯定比单条指令的效率要低，但是现在的 ARM 处理器能够检测到这种指令序列并能够对这种指令进行优化。

另外，IDA 这样的工具能够识别出这种一分为二指令，并在显示的时候把它们合二为一。

```
MOV    R0, 0x12345678
BX     LR
```

28.3.2　ARM64

```
uint64_t f()
{
        return 0x12345678ABCDEF01;
};
```

指令清单 28.3　GCC 4.9.1 -O3

```
mov     x0, 61185    ; 0xef01
movk    x0, 0xabcd, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
ret
```

其中，MOVK 是“MOV Keep”的缩写。它把 16 位数值存储到寄存器里，而保留寄存器里的其他比特

位。实际上这几个 MOVK 指令先使用 LSL 指令依次位移了 16、32、48 位，然后再进行的赋值操作。即，程序通过 4 条指令把一个 64 位数值存储到寄存器里。

浮点数

一条指令就可把浮点型数据存储到 D-存储器里。我们来看：

```
double a()
{
        return 1.5;
};
```

指令清单 28.4 GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
   0:   1e6f1000        fmov    d0, #1.5000000000000000e+000
   4:   d65f03c0        ret
```

上面这个单条 32 位指令如何封装浮点数 1.5 呢？在 ARM64 的 FMOV 指令里，有 8 个特殊的比特位。这 8 位空间用于编排浮点型数据。通过 VFPExpandImm()函数的算法，编译器把浮点数封装在 FMOV 指令的 8 位空间里。这种算法又叫作 minifloat[①]，实现方法请参见[ARM13a]。我测试了几个浮点数，发现编译器能够通过该函数把 30.0 和 31.0 编排在 8 位指令空间里。但是这 8 位空间无法封装 32.0。根据 IEEE 754 规范，32.0 要占用 8 个字节的空间。

```
double a()
{
        return 32;
};
```

上述源程序的汇编指令如下所示。

指令清单 28.5 GCC 4.9.1 -O3

```
a:
        ldr     d0, .LC0
        ret
.LC0:
        .word   0
        .word   1077936128
```

28.4 重定位

我们已经知道 ARM64 的指令都是 4 字节（汇编）指令。4 个字节的容量有限，无法封装很大的数。尽管如此，程序镜像可能会被操作系统加载内存中的任意地址，这就需要（基址）重定位（relocations/relocs）来进行修正。有关重定位的详细介绍，请参见本书的 68.2.6 节。

ARM64 成对使用 ADRP 和 ADD 指令来传递 64 位指针地址。ADRP 指令用来获取标签所在处（本例是 main）的 4KB 分页地址，而 ADD 指令则负责存储偏移量的其余部分。在 Win32 下使用 GCC(Linaro) 4.9 编译了“Hello, word!”程序（即第 6 章的第一个程序），然后使用 objdump 工具分析它的目标文件。

指令清单 28.6 GCC (Linaro) 4.9 and objdump of object file

```
...>aarch64-linux-gnu-gcc.exe hw.c -c

...>aarch64-linux-gnu-objdump.exe -d hw.o

...
```

① https://en.wikipedia.org/wiki/Minifloat。

```
0000000000000000 <main>:
   0:   a9bf7bfd          stp     x29, x30, [sp,#-16]!
   4:   910003fd          mov     x29, sp
   8:   90000000          adrp    x0, 0 <main>
   c:   91000000          add     x0, x0, #0x0
  10:   94000000          bl      0 <printf>
  14:   52800000          mov     w0, #0x0                        // #0
  18:   a8c17bfd          ldp     x29, x30, [sp],#16
  1c:   d65f03c0          ret

...>aarch64-linux-gnu-objdump.exe -r hw.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21  .rodata
000000000000000c R_AARCH64_ADD_ABS_LO12_NC .rodata
0000000000000010 R_AARCH64_CALL26 printf
```

这个目标文件有 3 个地方涉及重定位：

- 第一处首先获取页面地址，把地址的低 12 位舍去，以便在 ADRP 指令里封装地址的高 21 位。ADRP 获取的偏移量的 4KB 地址，其最后 12 位肯定是零，所以可以被忽略；另一方面，ADRP 指令也只有 21 位空间可以封装数据。
- 其后的 ADD 指令则用于保存偏移量的低 12 位。
- 跳转到 printf()函数的 BL 指令。若把 B/BL 指令逐位的展开，会看到其中只有 26 比特可供存储偏移量。实际上 ARM 模式和 ARM64 模式的转移指令只可能跳转到 4 的整数倍的偏移量地址，所以在指令中省略了最后的 2 位（相当于右移 2 位）。在执行转移指令时，相当于先对文件中的偏移量左移两位后再进行相应跳转。如此一来，转移指令的偏移量空间是 28 位，而不是 26 位；即可跳转至 PC±128MB 的地址（即偏移量的取值范围）。

最后生成的可执行文件里并没有再出现重定位。因为在编译过程的后续阶段，“Hello!”字符串的相对地址、分页地址，以及 puts()函数的相对地址都是可被确定的已知数。链接器 linker 可依次计算出 ADRP、ADD 和 BL 指令所需的实际偏移量。

使用 objdump 分析最终的可执行文件，可以看到如下所示的代码。

指令清单 28.7　objdump of executable file

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0                        // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210000 ........Hello!..
```

BL 指令要跳转到的地址可以推算出来。

假如 BL 那条指令的 opcode 是 0x97ffffa0，即二进制的 10010111111111111111111110100000。依据[ARM13a] C5.2.26 描述的有关技术规范，opcode 的最后 26 位是 imm26。因此 imm26 的二进制数值为 11111111111111111110100000，即 imm26 = 0x3FFFFA0。但是 imm26 的最高数权位即符号位是 1，所以它是

负数的补码。要把补码转换为原码，则要进行就取非再加一的运算。由此可得原码为负的 0x5F+1=0x60，即−0x60。ARM 指令里的偏移量是实际偏移量除以 4，所以实际偏移量是其 4 倍，即−0x180。综上，BL 将要跳转到的目标地址是 0x4005a0−0x180 = 0x400420。请注意：要以 BL 指令的偏移量为基数进行计算。如果要对 PC 指针进行计算，那么整个推算过程将完全不同。

如需深入了解 AM64 的重定位，请参见《ELF for the ARM 64-bit Architecture (AArch64)》（2013）。其官方下载地址是 http://infocenter.arm.com/help/topic/com.arm.doc.ihi0056b/IHI0056B_aaelf64.pdf。

第 29 章　MIPS 的特点

29.1　加载常量

```
unsigned int f()
{
        return 0x12345678;
};
```

MIPS 和 ARM 平台的指令都是定长指令。MIPS 程序的指令都是 32 位 opcode。在 MIPS 平台上，我们不可能只使用单条指令就完成对 32 位常量的赋值操作。所以此类操作至少分两步进行：首先传递 32 位数据的高 16 位，然后再通过 ORI 操作向寄存器传递立即数的低 16 位。

指令清单 29.1　GCC 4.4.5 -O3 (assembly output)

```
        li      $2,305397760                    # 0x12340000
        j       $31
        ori     $2,$2,0x5678 ; branch delay slot
```

IDA 能够识别此类组合指令。为了方便阅读，它对后面的 ORi 指令进行了处理：用伪指令 LI 替换了 Ori 指令，而且用完整的 32 位值替换了值的低 16 位。

指令清单 29.2　GCC 4.4.5 -O3 (IDA)

```
        lui     $v0, 0x1234
        jr      $ra
        li      $v0, 0x12345678 ; branch delay slot
```

GCC 直接生成的汇编输出文件（assembly output）同样使用了 LI 伪指令。不过 GCC 的 LI 实际上是 LUI（Load Upper Immediate）指令，把数据的高 16 位传递给寄存器的高半部分。

29.2　阅读推荐

Dominc Sweetman 撰写的《See MIPS Run（第二版）》，2010 年印刷。

第二部分

硬件基础

第 30 章　有符号数的表示方法

有符号数通常以二进制补码[①]的形式存储于应用程序里。维基百科介绍了各种表示方法，有兴趣的读者可参见：http://en. wikipedia.org/wiki/Signed_number_representations。本节摘录了一些典型的单字节数值。

二进制	十六进制	无符号值	有符号值（补码）
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	253	−1
11111110	0xfe	254	−2
...			
10000010	0x82	130	−126
10000001	0x81	129	−127
10000000	0x80	128	−128

如果 0xFFFFFFFE 和 0x0000002 都是无符号数，则第一个数（4294967294）就比第二个数（2）大。如果这两个值表示的是有符号数，那么第一个数（−2）反而比第二个数（2）小了。所以，为了正确操作有符号数和无符号数，条件转移指令（参见本书第 12 章）特地分为 JG/JL 系列（signed 型）和 JA/JBE 系列（unsigned 型）两套指令。

为了便于记忆，我们总结其特点如下。

- 同一个数值即可以表示有符号数，也可以表示无符号数。
- C/C++的有符号型数据有：
  - int64_t（从-9223372036854775806 至 9223372036854775807），即从 0x8000000000000000 至 0x7FFFFFFFFFFFFFFF）
  - int（取值范围从−2147483646 至 2147483647，即 0x80000000 至 0x7FFFFFFF）。
  - char（取值范围从−127 至 128，即从 0x7F 至 0x80）。
  - ssize_t。
- 无符号型数据有：
  - uint64_t（从 0 至 18446744073709551615/0xFFFFFFFFFFFFFFFF）：
  - unsigned int（取值范围从 0 至 4294967295，即从 0 至 0xFFFFFFFF）。
  - unsigned char（取值范围从 0 至 255，即从 0 至 0xFF）。
  - size_t。
- 有符号数的最高数权位是符号位：1 代表负数，0 代表正数。
- 从数据宽度较小的数据转换为数据宽度较大的数据是可行的。可参见前面 24.5 节。
- 负数的补码和原码的双向转换过程是相同的，都是逐位求非再加 1。这种运算过程简单易记：同一个值的正负数是相反的值，所以要求非；求非之后再加 1 则是因为中间的“零”占了一个数的

① two's complement；简称为“补码”。

位置。

- 加减运算不区分有符号数和无符号数，它们的加减运算指令完全相同。但是乘除法运算还是有区别的：在 x86 指令集里，有符号数的乘除指令是 IMUL/IDIV，而无符号数的指令是 MUL/DIV。
- 此外，有符号数的操作指令更多一些。例如 CBW/CWD/CWDE/CDQ/CDQE（附录 A.6.3）、MOVSX（15.1.1 节），SAR（附录 A.6.3）。

第31章 字 节 序

字节序是指多字节类型的数据在内存中的存放顺序，通常可分为小端、大端两种字节序。小端字节序（Little-endian）指低数权字节数据存放在内存低地址处，高数权字节数据存放在内存高地址处的内存分布方式；大端字节序（Big-endian）是高数权字节数据存放在低地址处，低字节数据存放在高地址处的内存分布方式。

31.1 大端字节序

在采用大端字节序时，0x12345678 在内存中的存储方式如下所示。

内存地址	字节值
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Motorola 68k、IBM POWER 系列 CPU 采用大端字节序。

31.2 小端字节序

在采用小端字节序时，0x12345678 在内存中的存储方式如下所示。

内存地址	字节值
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Intel x86 系列 CPU 采用小端字节序。

31.3 举例说明

为了进行演示，我们可以在 QEMU 的虚拟化环境中安装 MIPS Linux。[①]

接下来在 MIPS Linux 里编译下述程序：

```
#include <stdio.h>

int main()
{
        int v, i;

        v=123;

        printf ("%02X %02X %02X %02X\n",
```

① Debian 网站提供虚拟机下载：https://people.debian.org/~aurel32/qemu/mips/。

```
                *(char*)&v,
                *(((char*)&v)+1),
                *(((char*)&v)+2),
                *(((char*)&v)+3));
};
```

然后运行下述指令：

```
root@debian-mips:~# ./a.out
00 00 00 7B
```

其中，0x7B 就是十进制的 123。在采用小端字节序的平台上，例如 x86 或 x86-64 的系统上，第一个字节就是 0x7B。但是 MIPS 采用的是大端字节序，所以数权最高的这个字节排列在最后。

正是因为 MIPS 的硬件平台可能采用两种不同的字节序，所以 MIPS Linux 又分为采用大端字节序的 MIPS Linux 和采用小端字节序的 mipsel Linux。在采取一种字节序的平台上编译出来的程序，不可能在另一种字节序的平台上运行。

本书的 21.4.3 节就介绍过 MIPS 大端字节序的特征。

31.4 双模二元数据格式

ARM、PowerPC、SPARC、MIPS、IA64 等 CPU 采用双模二元数据格式（Bi-endian），它们即可以工作于小端字节序也可以切换到大端字节序。

31.5 转换字节序

BSWAP 指令可在汇编层面转换数据的字节序。

TCP/IP 数据序的封装规范采用大端字节序，所以采用小端字节序平台的系统就需要使用专门的转换字节序的函数。

常用的字节序转换函数是 htonl()和 htons()。

在 TCP/IP 的术语里，大端字节序又称为“网络字节顺序（Network Byte Order）”，网络主机采用的字节序叫作“主机字节顺序”。x86 和其他一些平台的主机字节序是小端字节序，但是 IBM POWER 等著名服务器系列均采用大端字节序。因此，在主机字节顺序为大端字节序的平台上使用 htonl()或 htons()函数转换字节序，其实不会进行真正意义上的字节重排。

第32章 内存布局

C/C++把内存划分为许多区域，主要的内存区域有：

- 全局内存空间，又称为“（全局）静态存储区 static memory allocation”。编程人员不需要为全局变量和静态变量明确划分存储空间凡是由源程序声明的全局变量、全局数组，编译器都能够在数据段或常量段为其分配适当的存储空间。由于整个程序都可以访问这个区域的数据，所以人们认为使用这种存储空间数据会破坏程序的结构化体系。此外，在全局内存区存储数据之前，必须事先声明其确切的容量。因而这个空间不适用于存储缓存或动态数组。在全局内存空间出现的缓冲区溢出问题，往往将覆盖在内存中相邻位置的变量或缓存（请参阅本书 7.2 节的案例）。
- 栈空间，即分配给栈的存储区域。它是由编译器自动分配、释放的存储区域，常用于存放函数的参数和局部变量。在特定情况下，局部变量可被其他函数访问（局部变量的指针作为参数传递给被调用方函数）。在指令层面，“分配和释放栈空间的实质就是调整 SP 寄存器的值，因而分配和释放栈空间的操作速度非常快。编译器只能为那些在编译阶段确定存储空间的局部变量分配栈空间。因此，无法事先预判存储空间的缓冲型数据类型，即缓冲区和动态数组[①]不会被分配到栈空间里。在栈空间发生的缓冲区溢出问题，通常会篡改栈里的重要数据（请参阅本书 18.2 节的案例）案例）。
- 堆空间，即“动态内存分配区。C 语言的 malloc()/free()函数或 C++的 new/delete 语句即可分配、释放堆空间）。“不必事先声明堆空间的大小”即“可在程序启动以后再确定数据块的容量”这一特征构成了其独特的便利性。另外，程序员还可以动态调整（realloc()函数）内存块的大小，只是调整堆空间的操作速度不很理想。在内存分配操作中，申请、释放堆空间的操作是速度最慢的操作：在分配、释放堆空间时，进行这种操作的程序必须支持并且更新所有控制结构。在这个区域发生的缓冲区溢出经常会覆盖堆空间的数据结构体。堆空间管理不当还会发生内存泄露问题：所有被分配的堆空间都应当被明确地释放，否则就会出现内存泄露问题。但是程序员可能会出现“忘记释放堆空间”的问题，还有发生释放不彻底的问题。另外，“在调用 free()函数释放空间之后再次直接使用这块内存区域”的指令同样会带来非常严重的安全问题（请参阅本书 21.2 节的案例）。

① 除非像 5.2.4 节那样使用 alloca()函数。

第 33 章　CPU

33.1　分支预测

现在的主流编译器基本都不怎么分配条件转移指令了。本书的 12.1.2 节、12.3 节和 19.5.2 节的编译结果都体现了这一特性。

虽然目前的分支预测功能并不完美，但是编译器还是在向这一方向发展。

ARM 平台出现的条件执行指令（例如 ADRcc）及 x86 平台出现的 CMOVcc 指令，都是这一趋势的明证。

33.2　数据相关性

当代的 CPU 多数都能并行执行指令（OOE/乱序执行技术）。但是，要充分利用 CPV 的乱序执行功能、尽可能频繁地同期执行多条指令，首先就要降低各指令之间的数据相关性。所以，编译器尽可能地分配那些不怎么影响 CPU 标识的指令。

因为 LEA 指令并不像其他数学运算指令那样影响标识位，所以编译器越来越多地使用这种指令。

第34章 哈希函数

哈希（hash）函数能够生成可靠程度较高的校验和（Checksum），可充分满足数据检验的需要。CRC32算法就是一种不太复杂的哈希算法。哈希值是一种固定长度的信息摘要，不可能根据哈希值逆向“推测”出信息原文。所以，无论CRC32算法计算的信息原文有多长，它只能生成32位的校验和。但是从加密学的角度看，我们可以轻易地伪造出满足同一CRC32哈希值的多个信息原文。当然，防止伪造就是加密哈希函数（Cryptographic hash function）的任务了。

此外，人们普遍使用MD5、SHA1等哈希算法生成用户密码的摘要（哈希值），然后再把密码摘要存储在数据库里。实际上网上论坛等涉及用户密码的数据库，存储的密码信息差不多都是用户密码的哈希值；否则一旦发生数据库泄露等问题，入侵人员将能够轻易地获取密码原文。不仅如此，当用户登录网站的时候，网络论坛等应用程序检验的也不是密码原文，它们检验的还是密码哈希值：如果用户名和密码哈希值与数据库里的记录匹配，它将授予登录用户相应的访问权限。另外，常见的密码破解工具通常都是通过穷举密码的方法，查找符合密码哈希值的密码原文而已。其他类型的密码破解工具就要复杂得多。

单向函数与不可逆算法

单向函数（one-way function）)是一种具有下述特点的单射函数：对于每一个输入，函数值都容易计算（多项式时间）；但是根据函数值对原始输入进行逆向推算却比较困难（无法在多项式时间内使用确定性图灵机计算）。本节着重讲解它的不可逆性。

假设函数的输入值是由10个介于0～9之间的数值构成的一组矢量，且矢量中的标量仅出现一次。例如：

```
4 6 0 1 3 5 7 8 9 2
```

下列算法即可实现最简单的单项函数：

- 取第0位的值作为参数1（本例而言是4）。
- 取第1位的值作为参数2（本例而言是6）。
- 交换位于参数1、2位置处（第4、6位）的值。

本例中的第4位和第6位数字分别是：

```
4 6 0 1 3 5 7 8 9 2
        ^   ^
```

进行最终变换可得：

```
4 6 0 1 7 5 3 8 9 2
```

即使我们知道具体的算法和最终的函数值，我们也无法确定最初的输入值是什么。因为最初的位置参数值可能是0也可能是1，这两个参数可能会被交换位置。

以上只是对单向函数的一种简单说明。实际应用中的单向函数远比本例复杂。

第三部分

一些高级的例子

第35章 温度转换

入门级的编程书籍一般都介绍“华氏温度转换为摄氏温度”的例子。

从华氏度转换成摄氏度的计算公式为

$$C=\frac{5\cdot(F-32)}{9}$$

笔者对程序添加了简单的出错处理：

① 输入的温度数必须正确。

② 核查最终结果，确保不会出现绝对零度（-273℃）以下的摄氏温度值。这是中学物理学课本介绍过的一个常识。

说明：调用函数 exit()会立即退出本程序，而且不会向调用方函数返回任何值。

35.1 整数值

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%d", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %d\n", celsius);
};
```

35.1.1 x86 构架下 MSVC 2012 优化

指令清单 35.1 x86 构架下 MSVC 2012 优化

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H

_fahr$ = -4                                             ; size = 4
_main   PROC
        push    ecx
        push    esi
```

```
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; call printf()
        lea     eax, DWORD PTR _fahr$[esp+12]
        push    eax
        push    OFFSET $SG4230          ; '%d'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        mov     eax, DWORD PTR _fahr$[esp+8]
        add     eax, -32                                ; fffffe0H
        lea     ecx, DWORD PTR [eax+eax*4]
        mov     eax, 954437177                          ; 38e38e39H
        imul    ecx
        sar     edx, 1
        mov     eax, edx
        shr     eax, 31                                 ; 0000001fH
        add     eax, edx
        cmp     eax, -273                               ; fffffeefH
        jge     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        push    eax
        push    OFFSET $SG4234          ; 'Celsius: %d'
        call    esi                     ; call printf()
        add     esp, 8
        ; return 0 - by C99 standard
        xor     eax, eax
        pop     esi
        pop     ecx
        ret     0
$LN8@main:
_main   ENDP
```

必须说明的是：

- 程序首先把 printf()函数的内存地址保存到 ESI 寄存器。在此之后，只要调用“CALL ESI”指令即可调用 prinf()函数了。这是非常常见的编译技术，或许是为了方便后续程序频繁调用这个函数，或许是因为还有“不用白不用”的空闲寄存器。
- 使用加法 ADD 而不使用减法 SUB。我们注意到程序中有一行指令是“ADD EAX,-32”，它用来实现从 EAX 寄存器中减去 32 的目的。程序没有采用指令“SUB EAX,32”，也就是说程序使用了 EAX=EAX+(-32)的算法，而没采用 EAX=EAX-32 的算法。是不是值得这么做，笔者并不能完全确定。
- 为了实现“乘以 5”的运算而使用了 LEA 指令：“lea ecx,DWORD PTR[eax+eax*4]”使得“i+i*4”和“i*5”是相等的。但是指令 LEA 运行速度比 IMUL 快。另外还可以使用指令对——SHL EAX,2 和 ADD EAX,EAX 来代替。确实有些编译器是这样做的。
- 这里也用到了用乘法来代替除法的技巧。参见第 41 章。
- 如果主函数 main()没有明确的返回值，在程序退出时，它的返回值为 0。C99 标准中标明“如果 main()函数没有通过明确的 return 指令声明其返回值，那么它将默认返回 0”。当然这项规则仅仅适用于主

函数 main()。虽然 MSVC 并未声称它完全遵循 C99 标准，但是或许它部分遵循了这一标准吧。这里指的 C99 标准是 ISO07,P.5.1.2.2.3。

35.1.2　x64 构架下的 MSVC 2012 优化

x64 的代码和 x86 的代码大体相同。只是每次调用 exit()函数之后，都有一个 INT 3 指令。

```
        xor     ecx, ecx
        call    QWORD PTR __imp_exit
        int     3
```

INT 3 是调试器 debugger 的断点设置指令。

当程序执行 exit()函数之后，它就不会再返回到原程序，而是直接退出了。编译器大概认为，在发生异常退出的情况下，通常人们应当使用调试器分析异常情况吧。

35.2　浮点数运算

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        double celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%lf", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 采用的是 FPU 指令。

指令清单 35.2　MSVC 2010 x86 优化

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
__real@4040000000000000 DQ 04040000000000000r   ; 32

_fahr$ = -8                                             ; size = 8
_main PROC
      sub     esp, 8
      push    esi
      mov     esi, DWORD PTR __imp__printf
      push    OFFSET $SG4038          ; 'Enter temperature in Fahrenheit:'
      call    esi                     ; call printf()
```

```
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4040          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4041          ; 'Error while parsing your input'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN2@main:
        fld     QWORD PTR _fahr$[esp+12]
        fsub    QWORD PTR __real@4040000000000000 ; 32
        fmul    QWORD PTR __real@4014000000000000 ; 5
        fdiv    QWORD PTR __real@4022000000000000 ; 9
        fld     QWORD PTR __real@c071100000000000 ; -273
        fcomp   ST(1)
        fnstsw  ax
        test    ah, 65                                  ; 00000041H
        jne     SHORT $LN1@main
        push    OFFSET $SG4043          ; 'Error: incorrect temperature!'
        fstp    ST(0)
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN1@main:
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4044          ; 'Celsius: %lf'
        call    esi
        add     esp, 12                                 ; 0000000cH
        ; return 0 - by C99 standard
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN10@main:
_main ENDP
```

但 MSVC 2012 分配的却是 SIMD 指令。

指令清单 35.3 MSVC 2012 x86 优化

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4040000000000000 DQ 04040000000000000r   ; 32
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5

_fahr$ = -8                                             ; size = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; call printf()
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4230          ; '%lf'
```

```
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        movsd   xmm1, QWORD PTR _fahr$[esp+12]
        subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
        movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
        mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
        divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
        comisd  xmm0, xmm1
        jbe     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        sub     esp, 8
        movsd   QWORD PTR [esp], xmm1
        push    OFFSET $SG4234          ; 'Celsius: %lf'
        call    esi                     ; call printf()
        add     esp, 12                                 ; 0000000cH
        ; return 0 - by C99 standard
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN8@main:
_main ENDP
```

当然，x86 的指令集确实支持 SIMD 指令，浮点数运算也毫无问题。大概是这种方式的计算指令比较简单，所以微软的编译器分配了 SIMD 指令。

我们还注意到绝对零度–273，早早地就导入了寄存器 XMM0。这也没关系，编译器不是按照源代码的书写顺序分配的汇编指令。

第 36 章　斐波拉契数列

在计算机编程方面的教科书里，我们通常都能找到 Fibonacci 数列（斐波拉契数列）（以下简称为“Fibonacci”）的生成函数。其实这种数列编排的规则非常简单：从第三项开始，后续数字为前两项数字之和。头两项一般为 0 和 1；也有头两项都是 1 的情况。因此，常见的 Fibonacci 数列大致如下：

0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597; 2584; 4181…

36.1　例子 1

Fibonacci 的实现方法比较简单。举例来说，下面这个程序可以生成数值不超过 21 的数列各项：

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        printf ("%d\n", a+b);
        if (a+b > limit)
                return;
        fib (b, a+b, limit);
};

int main()
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

指令清单 36.1　MSVC 2010 x86

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        push    eax
        push    OFFSET $SG2750 ; "%d"
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     ecx, DWORD PTR _limit$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        add     edx, DWORD PTR _b$[ebp]
        push    edx
        mov     eax, DWORD PTR _b$[ebp]
        push    eax
        call    _fib
        add     esp, 12
        pop     ebp
        ret     0
_fib    ENDP
```

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2753 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

我们重点分析这个程序的栈结构。

我们在 OllyDbg 中加载本例生成的可执行程序，并跟踪到调用 f()函数的那条指令，如图 36.1 所示。

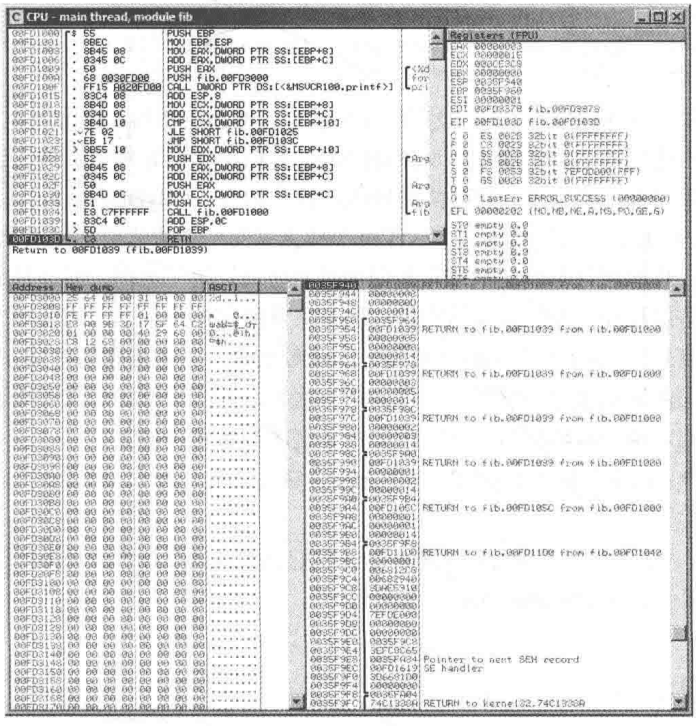


图 36.1　OllyDbg 最后一个函数 f()

让我们来仔细分析栈里的内容。在下面的程序行中，笔者用打括弧的方法加了两个注释。一个是 main()为 f1()做准备，另外一个是 CRT 为 main()做准备[①]。

```
0035F940    00FD1039   RETURN to fib.00FD1039 from fib.00FD1000
0035F944    00000008   1st argument: a
0035F948    0000000D   2nd argument: b
0035F94C    00000014   3rd argument: limit
0035F950   /0035F964   saved EBP register
0035F954   |00FD1039   RETURN to fib.00FD1039 from fib.00FD1000
0035F958   |00000005   1st argument: a
0035F95C   |00000008   2nd argument: b
0035F960   |00000014   3rd argument: limit
0035F964   ]0035F978   saved EBP register
```

① 在 OllyDbg 中，可以选择多项指令，再使用 Ctrl+C 组合键把它们复制到剪切板中。本例就是这样做的。

```
0035F968   |00FD1039   RETURN to fib.00FD1039 from fib.00FD1000
0035F96C   |00000003   1st argument: a
0035F970   |00000005   2nd argument: b
0035F974   |00000014   3rd argument: limit
0035F978   ]0035F98C   saved EBP register
0035F97C   |00FD1039   RETURN to fib.00FD1039 from fib.00FD1000
0035F980   |00000002   1st argument: a
0035F984   |00000003   2nd argument: b
0035F988   |00000014   3rd argument: limit
0035F98C   ]0035F9A0   saved EBP register
0035F990   |00FD1039   RETURN to fib.00FD1039 from fib.00FD1000
0035F994   |00000001   1st argument: a
0035F998   |00000002   2nd argument: b
0035F99C   |00000014   3rd argument: limit
0035F9A0   ]0035F9B4   saved EBP register
0035F9A4   |00FD105C   RETURN to fib.00FD105C from fib.00FD1000
0035F9A8   |00000001   1st argument: a           \
0035F9AC   |00000001   2nd argument: b           | prepared in main() for f1()
0035F9B0   |00000014   3rd argument: limit       /
0035F9B4   ]0035F9F8   saved EBP register
0035F9B8   |00FD11D0   RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC   |00000001   main() 1st argument: argc  \
0035F9C0   |006812C8   main() 2nd argument: argv  | prepared in CRT for main()
0035F9C4   |00682940   main() 3rd argument: envp  /
```

本例属于递归函数[①]。递归函数的栈一般都是这样的“三明治”结构。在上述程序中，limit（阈值）参数总是保持不变（十六进制的 14，也就是十进制的 20），而两个参数 *a* 和 *b* 在每次调用函数的时候都是不同的值。此外栈也存储了 RA 和保存 EBP（扩展堆栈指针）的值。OllyDbg 能基于 EBP 的值判断栈的存储结构，因此它能够用括弧标注栈帧。换而言之，在每个括弧里的一组数值都形成了一个相对独立的栈结构，即栈帧（stack frame）。栈帧就是每次函数调用期间的数据实体。另一方面，即使从纯萃技术方面看每个被调用方函数确实可以访问栈帧之外的栈存储空间，但是正常情况下不应当访问栈帧之外的数据（当然除了获取函数参数的操作以外）。对于没有 bug（缺陷）的函数来说，上述命题的确成立。每一个 EBP 值都是前一个栈帧的地址。因此调试程序能够把数据栈识别为栈帧，并能识别出每次调用函数时传递的参数值。

结合上述指令可知，递归函数应当为下一轮的自身调用制备各项参数。

在程序的最后部分，main()有 3 个参数。其中 argc（参数总数）为 1。确实如此，笔者的确未带参数直接运行并调试本程序。

另外，调整本程序、引发栈溢出的过程并不复杂：我们只需要删除或者注释掉阈值 limit 判断语句，即可导致栈溢出（即错误编号为 0xC00000FD 的异常错误）。

36.2 例子 2

上一节的函数有些冗余。接下来，我们增加一个新的局部变量 next，并用它来代替所有程序中的 *a*+*b*:

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        int next=a+b;
        printf ("%d\n", next);
        if (next > limit)
                return;
        fib (b, next, limit);
};

int main()
```

① 也就是自己调用自己。

```
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

这是非优化 MSVC 的输出，因此 next 变量确实是在本地栈中分配存储空间。

指令清单 36.2　MSVC 2010 x86

```
_next$ = -4     ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _next$[ebp], eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        push    OFFSET $SG2751 ; '%d'
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     edx, DWORD PTR _next$[ebp]
        cmp     edx, DWORD PTR _limit$[ebp]
        jle     SHORT $LN1@fib
        jmp     SHORT $LN2@fib
$LN1@fib:
        mov     eax, DWORD PTR _limit$[ebp]
        push    eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        mov     edx, DWORD PTR _b$[ebp]
        push    edx
        call    _fib
        add     esp, 12
$LN2@fib:
        mov     esp, ebp
        pop     ebp
        ret     0
_fib    ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2753 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

我们再来调用 OllyDbg，如图 36.2 所示。

现在，每个栈帧里都有一个变量 next。

我们来仔细看看堆栈。笔者依然给其中增加了注释。

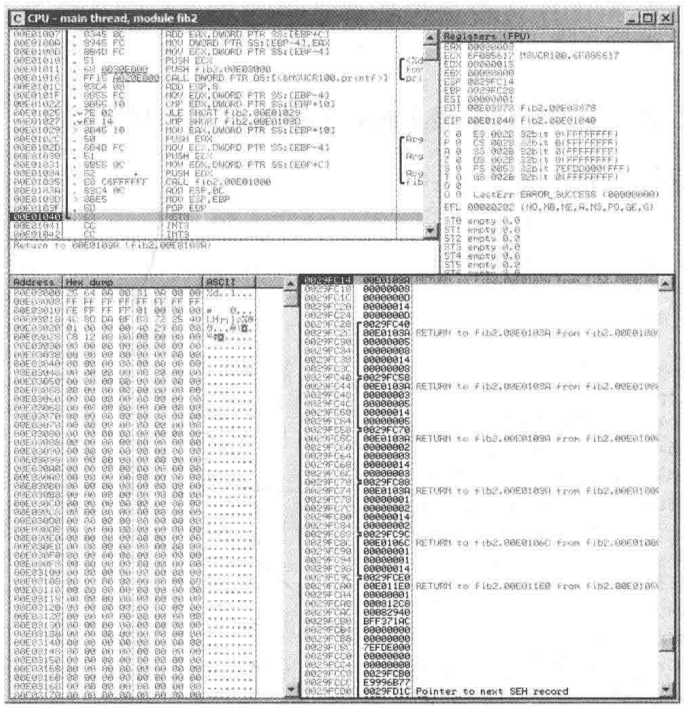


图 36.2 OllyDbg: 最后调用 f()

```
0029FC14   00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC18   00000008  1st argument: a
0029FC1C   0000000D  2nd argument: b
0029FC20   00000014  3rd argument: limit
0029FC24   0000000D  "next" variable
0029FC28  /0029FC40  saved EBP register
0029FC2C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC30  |00000005  1st argument: a
0029FC34  |00000008  2nd argument: b
0029FC38  |00000014  3rd argument: limit
0029FC3C  |00000008  "next" variable
0029FC40  ]0029FC58  saved EBP register
0029FC44  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC48  |00000003  1st argument: a
0029FC4C  |00000005  2nd argument: b
0029FC50  |00000014  3rd argument: limit
0029FC54  |00000005  "next" variable
0029FC58  ]0029FC70  saved EBP register
0029FC5C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC60  |00000002  1st argument: a
0029FC64  |00000003  2nd argument: b
0029FC68  |00000014  3rd argument: limit
0029FC6C  |00000003  "next" variable
0029FC70  ]0029FC88  saved EBP register
0029FC74  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC78  |00000001  1st argument: a             \
0029FC7C  |00000002  2nd argument: b             | prepared in f1() for next f1()
0029FC80  |00000014  3rd argument: limit         /
0029FC84  |00000002  "next" variable
0029FC88  ]0029FC9C  saved EBP register
0029FC8C  |00E0106C  RETURN to fib2.00E0106C from fib2.00E01000
0029FC90  |00000001  1st argument: a             \
0029FC94  |00000001  2nd argument: b             | prepared in main() for f1()
0029FC98  |00000014  3rd argument: limit         /
0029FC9C  ]0029FCE0  saved EBP register
0029FCA0  |00E011E0  RETURN to fib2.00E011E0 from fib2.00E01050
```

```
0029FCA4  |00000001  main() 1st argument: argc  \
0029FCA8  |000812C8  main() 2nd argument: argv  | prepared in CRT for main()
0029FCAC  |00082940  main() 3rd argument: envp  /
```

这里我们看到：递归函数在每次调用期间都会计算并传递下一轮调用所需的函数参数。

36.3　总结

递归函数只是看起来很帅而已。从技术上讲，递归函数在栈方面的开销过大，因而性能不怎么理想。注重性能指标的应用程序，应当避免使用递归函数。

笔者曾经编写过一个遍历二叉树、搜索既定节点的应用程序。把它写成递归函数的时候，整个程序确实又清爽又有条理性。但是每次函数调用都得进行赋值、回调，这使得递归函数比其他类型的函数慢了数倍。

另外，部分 PL[①]编译器会对递归调用采取“尾部调用”的优化方法，以减轻栈的各种开销。

① PL：Program Language（编程语言）。LISP、Python、Lua 等编程语言的编译器能够进行尾部调用优化。详情请参阅 https://en.wikipedia.org/wiki/Tail_call

第 37 章　CRC32 计算的例子

本章介绍一个基于表查询技术实现的 CRC32 校验值的计算程序：[①]

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long ub4;
typedef  unsigned char ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
  0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
  0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
  0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
  0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
  0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
  0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
  0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
  0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
  0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
  0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
  0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
  0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
  0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
  0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
  0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
  0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
  0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
  0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
  0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
  0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
  0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
  0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
  0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
  0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
  0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
  0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
  0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
  0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
  0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
```

① 源代码来源于 http://go.yurichev.com/17327。

```
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n");
  }
}


/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
  ub4 i;
  const ub1 *k = key;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
  return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
  char s[1000];
  while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
  return 0;
}
```

我们这里只关心校验函数 crc()的详细细节。另外，请注意 for()语句的两个初始指令“hash=len”和“i=0”。当然，在 C/C++语言里，我们可以一次指定两条循环初始指令。在最终的汇编指令层面，也就会出现两条初始化指令，而不会是一条指令。

下面我们采用优化的方式（/0x）来编译。为简化起见，这里只列出 crc()函数，同时还增加了笔者的注释。

```
_key$ = 8                ; size = 4
_len$ = 12               ; size = 4
_hash$ = 16              ; size = 4
_crc    PROC
    mov    edx, DWORD PTR _len$[esp-4]
    xor    ecx, ecx ; i will be stored in ECX
    mov    eax, edx
    test   edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
    push   esi
    mov    esi, DWORD PTR _key$[esp+4] ; ESI = key
    push   edi
$LL3@crc:

; work with bytes using only 32-bit registers. byte from address key+i we store into EDI

    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax ; EBX = (hash = len)
    and    ebx, 255 ; EBX = hash & 0xff
```

```
; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits of each register
; but other bits (8-31) are cleared all time, so its OK'
; these are cleared because, as for EDI, it was done by MOVZX instruction above
; high bits of EBX was cleared by AND EBX, 255 instruction above (255 = 0xff)

    xor    edi, ebx

; EAX=EAX>>8; bits 24-31 taken "from nowhere" will be cleared
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] table
    xor    eax, DWORD PTR _crctab[edi*4]
    inc    ecx             ; i++
    cmp    ecx, edx        ; i<len ?
    jb     SHORT $LL3@crc ; yes
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc:
    ret    0
_crc    ENDP
```

在 GCC 4.4.1 环境下，启用优化选项-03 编译，可得到的下述代码。

```
                public crc
crc             proc near

key             = dword ptr 8
hash            = dword ptr 0Ch

                push    ebp
                xor     edx, edx
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+key]
                push    ebx
                mov     ebx, [ebp+hash]
                test    ebx, ebx
                mov     eax, ebx
                jz      short loc_80484D3
                nop                     ; padding
                lea     esi, [esi+0]    ; padding; works as NOP (ESI does not changing here)

loc_80484B8:
                mov     ecx, eax        ; save previous state of hash to ECX
                xor     al, [esi+edx]   ; AL=*(key+i)
                add     edx, 1          ; i++
                shr     ecx, 8          ; ECX=hash>>8
                movzx   eax, al         ; EAX=*(key+i)
                mov     eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
                xor     eax, ecx        ; hash=EAX^ECX
                cmp     ebx, edx
                ja      short loc_80484B8
loc_80484D3:
                pop     ebx
                pop     esi
                pop     ebp
                retn
crc             endp
\
```

GCC 增加了空指令 NOP 以及 LEA esi,[esi+0]（这实际也是一个空指令），以此使得循环语句向 8 字节对齐。另外，编译器通常还会采用 npad 指令进行边界对齐。有关详情请参阅本书第 88 章。

第 38 章　网络地址计算实例

众所周知，IPv4 下的 TCP/IP 地址由 4 个数字组成，每个数字都在 0～255（十进制）之间。所以 IPv4 的地址可以表示为 4 字节数据。4 字节数据就是一个 32 位数据。因此 IPv4 的主机地址、子网掩码和网络地址都可以表示为一个 32 位的整数。

从使用者的角度来看，子网掩码由 4 位数字组成，写出来大致就是 255.255.255.0 这类形式的数字。但是网络工程师或者系统管理员更喜欢使用更为紧凑的表示方法，也就是 CIDR[①]规范的“/8”“/16”一类的表示方法。CIDR 格式的子网掩码从子网掩码的 MSB（最高数权位）开始计数，统计子网掩码里面有多少个 1 并将统计数字转换为 10 进制数。

CIDK 规范的掩码	数字空间	可用地址（个）[②]	十进制子网掩码	十六进制子网掩码	
/30	4	2	255.255.255.252	fffffffc	
/29	8	6	255.255.255.248	fffffff8	
/28	16	14	255.255.255.240	fffffff0	
/27	32	30	255.255.255.224	ffffffe0	
/26	64	62	255.255.255.192	ffffffc0	
/24	256	254	255.255.255.0	ffffff00	C 类网段
/23	512	510	255.255.254.0	fffffe00	
/22	1024	1022	255.255.252.0	fffffc00	
/21	2048	2046	255.255.248.0	fffff800	
/20	4096	4094	255.255.240.0	fffff000	
/19	8192	8190	255.255.224.0	ffffe000	
/18	16384	16382	255.255.192.0	ffffc000	
/17	32768	32766	255.255.128.0	ffff8000	
/16	65536	65534	255.255.0.0	ffff0000	B 类网段
/8	16777216	16777214	255.0.0.0	ff000000	A 类网段

这里举一个简单的例子：将子网掩码应用到主机地址，从而计算的网络地址。

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
        return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
        printf ("%d.%d.%d.%d\n",
                (a>>24)&0xFF,
                (a>>16)&0xFF,
                (a>>8)&0xFF,
                (a)&0xFF);
};
```

① CIDR 是 Classless Inter-Domain Routing 的缩写，即无类域间路由。
② 可用地址二数字空间-2。

```
// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
        return input=input|(1<<bit);
};

uint32_t form_netmask (uint8_t netmask_bits)
{
        uint32_t netmask=0;
        uint8_t i;

        for (i=0; i<netmask_bits; i++)
                netmask=set_bit(netmask, 31-i);

        return netmask;
};

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t   ip4, uint8_t netmask_bits)
{
        uint32_t netmask=form_netmask(netmask_bits);
        uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
        uint32_t netw_adr;

        printf ("netmask=");
        print_as_IP (netmask);

        netw_adr=ip&netmask;

        printf ("network address=");
        print_as_IP (netw_adr);
};

int main()
{
        calc_network_address (10, 1, 2, 4, 24); // 10.1.2.4, /24
        calc_network_address (10, 1, 2, 4, 8);  // 10.1.2.4, /8
        calc_network_address (10, 1, 2, 4, 25);      // 10.1.2.4, /25
        calc_network_address (10, 1, 2, 64, 26);     // 10.1.2.4, /26
};
```

38.1 计算网络地址函数 calc_network_address()

计算网络地址函数 calc_network_address()实现起来非常简单：它将主机地址和网络子网掩码进行 AND 与运算，得到的结果就是网络的实际地址。

指令清单 38.1 MSVC 2012 采用参数/Ob0 优化

```
 1 _ip1$ = 8                ; size = 1
 2 _ip2$ = 12               ; size = 1
 3 _ip3$ = 16               ; size = 1
 4 _ip4$ = 20               ; size = 1
 5 _netmask_bits$ = 24      ; size = 1
 6 _calc_network_address PROC
 7         push    edi
 8         push    DWORD PTR _netmask_bits$[esp]
 9         call    _form_netmask
10         push    OFFSET $SG3045  ; 'netmask='
11         mov     edi, eax
12         call    DWORD PTR __imp__printf
13         push    edi
14         call    _print_as_IP
```

```
15          push    OFFSET $SG3046  ; 'network address='
16          call    DWORD PTR __imp__printf
17          push    DWORD PTR _ip4$[esp+16]
18          push    DWORD PTR _ip3$[esp+20]
19          push    DWORD PTR _ip2$[esp+24]
20          push    DWORD PTR _ip1$[esp+28]
21          call    _form_IP
22          and     eax, edi        ; network address = host address & netmask
23          push    eax
24          call    _print_as_IP
25          add     esp, 36
26          pop     edi
27          ret     0
28  _calc_network_address ENDP
```

在第 22 行，我们可以看到最为重要的运算指令 AND。就是它计算出了网络地址，实现了核心功能。

38.2　函数 form_IP()

form_IP()函数将 IP 的 4 个字节转换成一个 32 位数值。

它的运算流程如下：

- 给返回值分配一个变量，并赋值为 0。
- 取数权最低的第 4 个字节，与返回值 0 进行 OR/或操作，即可得到含有第 4 字节信息的 32 位值。
- 取第 3 个字节，左移 8 位，以生成 0x0000bb00（其中 bb 就是这步读取的第三字节）这种形式的数值。此后与返回值进行 OR/或运算。如果上一步的值如果是 0x000000aa 的话，在执行 OR 或操作后，就会得到 0x0000bbaa 这样的返回值。
- 依此类推。取第 2 个字节，左移 16 位，生成 0x00cc0000 这样一个含有第 2 字节的 32 位值，再进行 OR/或运算。由于以上一步的返回值应当是 0x0000bbaa，因此本次运算的结果会是 0x00ccbbaa。
- 同理。取最高位，左移 24 位，以生成 0xdd000000 这样一个含有第一字节信息的 32 位值，再进行 OR/或运算。由于上一步的返回值是 0x00ccbbaa，因此最终的结果的值就是 0xddccbbaa 这样的 32 位值了。

经 MSVC 2012 进行非优化编译，可得到下述指令：

指令清单 38.2　非优化的 MSVC2012 的实现

```
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8       ; size = 1
_ip2$ = 12      ; size = 1
_ip3$ = 16      ; size = 1
_ip4$ = 20      ; size = 1
_form_IP PROC
        push    ebp
        mov     ebp, esp
        movzx   eax, BYTE PTR _ip1$[ebp]
        ; EAX=000000dd
        shl     eax, 24
        ; EAX=dd000000
        movzx   ecx, BYTE PTR _ip2$[ebp]
        ; ECX=000000cc
        shl     ecx, 16
        ; ECX=00cc0000
        or      eax, ecx
        ; EAX=ddcc0000
        movzx   edx, BYTE PTR _ip3$[ebp]
        ; EDX=000000bb
        shl     edx, 8
        ; EDX=0000bb00
```

```
        or      eax, edx
        ; EAX=ddccbb00
        movzx   ecx, BYTE PTR _ip4$[ebp]
        ; ECX=000000aa
        or      eax, ecx
        ; EAX=ddccbbaa
        pop     ebp
        ret     0
_form_IP ENDP
```

这里操作的顺序不同。当然这不会影响最后的运算结果。

在启用优化选项之后，MSVC 2012 会生成另一种算法的应用程序：

指令清单 38.3 优化的 MSVC2012 带参数/Ob0 的实现

```
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8       ; size = 1
_ip2$ = 12      ; size = 1
_ip3$ = 16      ; size = 1
_ip4$ = 20      ; size = 1
_form_IP PROC
        movzx   eax, BYTE PTR _ip1$[esp-4]
        ; EAX=000000dd
        movzx   ecx, BYTE PTR _ip2$[esp-4]
        ; ECX=000000cc
        shl     eax, 8
        ; EAX=0000dd00
        or      eax, ecx
        ; EAX=0000ddcc
        movzx   ecx, BYTE PTR _ip3$[esp-4]
        ; ECX=000000bb
        shl     eax, 8
        ; EAX=00ddcc00
        or      eax, ecx
        ; EAX=00ddccbb
        movzx   ecx, BYTE PTR _ip4$[esp-4]
        ; ECX=000000aa
        shl     eax, 8
        ; EAX=ddccbb00
        or      eax, ecx
        ; EAX=ddccbbaa
        ret     0
_form_IP ENDP
```

这个实现过程还可以描述为：每个字节都写入到其返回值的最低 8 个比特位，并且每次左移一个字节，并将其与返回值做或操作。重复四次，就能完成函数功能。

就这样了。然而遗憾的是，可能没其他的办法来实现以上的逻辑了。据笔者所知，目前的 CPU 及其 ISA 还不能把既定比特位或者字节直接复制到其他类型数据里。所以一般都是通过位移和 OR 或运算才能把 IP 地址转换为 32 位数据。

38.3 函数 print_as_IP()

函数 print_as_IP()实现的功能与上面函数完全相反，它将一个 32 位的数值切分成 4 个字节。切分过程比较简单：只需要将输入的数值分别位移 24 位、16 位、8 位或者 0 位，取最低字节的 0 到 7 位即可。

指令清单 38.4 非优化 MSVC 2012

```
_a$ = 8                 ; size = 4
_print_as_IP PROC
        push    ebp
        mov     ebp, esp
```

```
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
        and     eax, 255
        ; EAX=000000aa
        push    eax
        mov     ecx, DWORD PTR _a$[ebp]
        ; ECX=ddccbbaa
        shr     ecx, 8
        ; ECX=00ddccbb
        and     ecx, 255
        ; ECX=000000bb
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        ; EDX=ddccbbaa
        shr     edx, 16
        ; EDX=0000ddcc
        and     edx, 255
        ; EDX=000000cc
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
        shr     eax, 24
        ; EAX=000000dd
        and     eax, 255 ; probably redundant instruction
        ; EAX=000000dd
        push    eax
        push    OFFSET $SG2973 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        pop     ebp
        ret     0
_print_as_IP ENDP
```

优化 MSVC 2012 程序做的和上面的一样，但是它不会重新加载输入值。

指令清单 38.5　优化 MSVC 2012 /Ob0

```
_a$ = 8         ; size = 4
_print_as_IP PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        ; ECX=ddccbbaa
        movzx   eax, cl
        ; EAX=000000aa
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 8
        ; EAX=00ddccbb
        and     eax, 255
        ; EAX=000000bb
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 16
        ; EAX=0000ddcc
        and     eax, 255
        ; EAX=000000cc
        push    eax
        ; ECX=ddccbbaa
        shr     ecx, 24
        ; ECX=000000dd
        push    ecx
        push    OFFSET $SG3020 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        ret     0
_print_as_IP ENDP
```

38.4 form_netmask()函数和 set_bit()函数

form_netmask()函数从网络短地址 CIDR 中获取网络子网掩码。当然，或许事先计算出一个查询表、转换的时候进行表查询的速度可能会更快。但是为了演示位移运算的特征，本节特意采用了这种现场计算的转换方法。我们这里还编写了一个函数 set_bit()。虽说格式转换这种底层运算本来不应当调用其他函数了，但是笔者相信 set_bit()函数可以提高代码的可读性。

指令清单 38.6 优化 MSVC 2012 /Ob0

```
_input$ = 8             ; size = 4
_bit$ = 12              ; size = 4
_set_bit PROC
        mov     ecx, DWORD PTR _bit$[esp-4]
        mov     eax, 1
        shl     eax, cl
        or      eax, DWORD PTR _input$[esp-4]
        ret     0
_set_bit ENDP

_netmask_bits$ = 8      ; size = 1
_form_netmask PROC
        push    ebx
        push    esi
        movzx   esi, BYTE PTR _netmask_bits$[esp+4]
        xor     ecx, ecx
        xor     bl, bl
        test    esi, esi
        jle     SHORT $LN9@form_netma
        xor     edx, edx
$LL3@form_netma:
        mov     eax, 31
        sub     eax, edx
        push    eax
        push    ecx
        call    _set_bit
        inc     bl
        movzx   edx, bl
        add     esp, 8
        mov     ecx, eax
        cmp     edx, esi
        jl      SHORT $LL3@form_netma
$LN9@form_netma:
        pop     esi
        mov     eax, ecx
        pop     ebx
        ret     0
_form_netmask ENDP
```

set_bit()函数的功能十分单一。它将输入值左移既定的比特位，接着将位移运算的结果与输入值进行或 OR 运算。而后 form_mask()函数通过循环语句重复调用 set_bit()函数，借助循环控制变量 netmask_bits 设置子网掩码里数值为 1 的各个比特位。

38.5 总结

上述程序的结果如下所示。

```
netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64
```

第 39 章　循环：几个迭代

多数循环语句只有一个迭代器。但是在汇编层面，一个迭代器也可能对应多个数据实体。

下面所示的是一个简单的例子。

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;

        // copy from one array to another in some weird scheme
        for (i=0; i<cnt; i++)
                a1[i*3]=a2[i*7];
};
```

我们可以看到，每次迭代都有两次乘法运算，这是很耗费时间的操作。能不能优化一下呢？答案是肯定的。如果仔细看看程序代码，就会发现这个程序中的矩阵的参数是跳跃的，我们能比较容易地不用乘法就将它计算出来。

39.1　三个迭代器

指令清单 39.1　采用 MSVC 2013 x64 优化的代码

```
f       PROC
; RDX=a1
; RCX=a2
; R8=cnt
        test    r8, r8          ; cnt==0? exit then
        je      SHORT $LN1@f
        npad    11
$LL3@f:
        mov     eax, DWORD PTR [rdx]
        lea     rcx, QWORD PTR [rcx+12]
        lea     rdx, QWORD PTR [rdx+28]
        mov     DWORD PTR [rcx-12], eax
        dec     r8
        jne     SHORT $LL3@f
$LN1@f:
        ret     0
f       ENDP
```

这里有三个迭代变量，它们是 cnt 变量以及 2 个数列参数（索引游标）。数列参数每次迭代都增加 12 或者 28（其实这就是采用加法代替了源程序中的乘法）。因此我们可以采用 C/C++语言重写代码如下。

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;
        size_t idx1=0; idx2=0;

        // copy from one array to another in some weird scheme
        for (i=0; i<cnt; i++)
        {
```

```
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
        };
};
```

这个程序可在每次迭代更新 3 个迭代参数，而不是 1 个。另外，编译器变相实现了两个乘法操作。

39.2　两个迭代器

GCC 4.9 可以做得更好，只有两个迭代。

指令清单 39.2　采用 GCC 4.9 x64 优化

```
; RDI=a1
; RSI=a2
; RDX=cnt
f:
        test    rdx, rdx ; cnt==0? exit then
        je      .L1
; calculate last element address in "a2" and leave it in RDX
        lea     rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
        sal     rdx, 5
; RDX=RDX<<5=cnt*32
        sub     rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
        add     rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3:
        mov     eax, DWORD PTR [rsi]
        add     rsi, 28
        add     rdi, 12
        mov     DWORD PTR [rdi-12], eax
        cmp     rsi, rdx
        jne     .L3
.L1:
        rep ret
```

这里没有计数器变量 counter 了，这是因为 GCC 编译器认为它没有必要。数组 a2 的最后一个参数在循环开始前就已经计算好了（很容易，其实就是 cnt 乘以 7）。而且循环的结束条件也不复杂：第二个索引号指数达到那个可预先计算出来的临界值时，循环语句随即终止迭代。

其中涉及一些采用加/减/位移等办法来替代乘法运算的知识，有兴趣的读者请参阅本书 16.1.3 节。

上述汇编代码与下述 C/C++程序相对应：

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;
        size_t idx1=0; idx2=0;
        size_t last_idx2=cnt*7;

        // copy from one array to another in some weird scheme
        for (;;)
        {
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
                if (idx2==last_idx2)
                        break;
        };
};
```

ARM64 下的 GCC(Linaro) 4.9 采用了同种类型的编译方法。但是它计算的是数组 a1 的最后一个索引值，而不是像上面的程序那样以数组 a2 为边界条件。当然，程序的功能最终还是一样的。

指令清单 39.3 ARM64 下的 GCC(Linaro) 4.9 优化

```
; X0=a1
; X1=a2
; X2=cnt
f:
        cbz     x2, .L1             ; cnt==0? exit then
; calculate last element of "a1" array
        add     x2, x2, x2, lsl 1
; X2=X2+X2<<1=X2+X2*2=X2*3
        mov     x3, 0
        lsl     x2, x2, 2
; X2=X2<<2=X2*4=X2*3*4=X2*12
.L3:
        ldr     w4, [x1],28         ; load at X1, add 28 to X1 (post-increment)
        str     w4, [x0,x3]         ; store at X0+X3=a1+X3
        add     x3, x3, 12          ; shift X3
        cmp     x3, x2              ; end?
        bne     .L3
.L1:
        ret
```

MIPS 下的 GCC 4.4.5 也差不多如此。

指令清单 39.4 MIPS(IDA)下的 GCC 4.4.5 优化

```
; $a0=a1
; $a1=a2
; $a2=cnt
f:
; jump to loop check code:
                beqz    $a2, locret_24
; initialize counter (i) at 0:
                move    $v0, $zero ; branch delay slot, NOP
loc_8:
; load 32-bit word at $a1
                lw      $a3, 0($a1)
; increment counter (i):
                addiu   $v0, 1
; check for finish (compare "i" in $v0 and "cnt" in $a2):
                sltu    $v1, $v0, $a2
; store 32-bit word at $a0:
                sw      $a3, 0($a0)
; add 0x1C (28) to \$a1 at each iteration:
                addiu   $a1, 0x1C
; jump to loop body if i<cnt:
                bnez    $v1, loc_8
; add 0xC (12) to \$a0 at each iteration:
                addiu   $a0, 0xC ; branch delay slot
locret_24:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

39.3 Intel C++ 2011 实例

编译器的优化操作有时候会非常奇怪。但是无论它们采用了何种优化方式，程序的功能肯定忠于源程序。这里列出的是 Intel C++ 2011 编译器如何操作的例子。

指令清单 39.5　Intel C++ 2011 (x64)优化

```
f       PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::                         ; Preds .B1.0
        test      r8, r8                                ;8.14
        jbe       exit          ; Prob 50%              ;8.14
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.2::                         ; Preds .B1.1
        cmp       r8, 6                                 ;8.2
        jbe       just_copy       ; Prob 50%            ;8.2
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.3::                         ; Preds .B1.2
        cmp       rcx, rdx                              ;9.11
        jbe       .B1.5         ; Prob 50%              ;9.11
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.4::                         ; Preds .B1.3
        mov       r10, r8                               ;9.11
        mov       r9, rcx                               ;9.11
        shl       r10, 5                                ;9.11
        lea       rax, QWORD PTR [r8*4]                 ;9.11
        sub       r9, rdx                               ;9.11
        sub       r10, rax                              ;9.11
        cmp       r9, r10                               ;9.11
        jge       just_copy2        ; Prob 50%          ;9.11
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.5::                         ; Preds .B1.3 .B1.4
        cmp       rdx, rcx                              ;9.11
        jbe       just_copy         ; Prob 50%          ;9.11
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.6::                         ; Preds .B1.5
        mov       r9, rdx                               ;9.11
        lea       rax, QWORD PTR [r8*8]                 ;9.11
        sub       r9, rcx                               ;9.11
        lea       r10, QWORD PTR [rax+r8*4]             ;9.11
        cmp       r9, r10                               ;9.11
        jl        just_copy         ; Prob 50%          ;9.11
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 ↙
    ↳ xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
just_copy2::                        ; Preds .B1.4 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
        xor       r10d, r10d                            ;8.2
        xor       r9d, r9d                              ;
        xor       eax, eax                              ;
                                ; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↙
    ↳ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.8::                         ; Preds .B1.8 just_copy2
        mov       r11d, DWORD PTR [rax+rdx]             ;3.6
        inc       r10                                   ;8.2
        mov       DWORD PTR [r9+rcx], r11d              ;3.6
        add       r9, 12                                ;8.2
        add       rax, 28                               ;8.2
        cmp       r10, r8                               ;8.2
        jb        .B1.8           ; Prob 82%            ;8.2
        jmp       exit          ; Prob 100%             ;8.2
                                ; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↙
    ↳ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
```

```
just_copy::                             ; Preds .B1.2 .B1.5 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
        xor       r10d, r10d                                    ;8.2
        xor       r9d, r9d                                      ;
        xor       eax, eax                                      ;
                                ; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 ↙
    ↳ xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.11::                                ; Preds .B1.11 just_copy
        mov       r11d, DWORD PTR [rax+rdx]                     ;3.6
        inc       r10                                           ;8.2
        mov       DWORD PTR [r9+rcx], r11d                      ;3.6
        add       r9, 12                                        ;8.2
        add       rax, 28                                       ;8.2
        cmp       r10, r8                                       ;8.2
        jb        .B1.11        ; Prob 82%                      ;8.2
                                ; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 ↙
    ↳ xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
exit::                          ; Preds .B1.11 .B1.8 .B1.1
        ret                                                     ;10.1
```

上述程序首先根据输入参数确定分支，接着执行相应的例程。虽然它看起来就像是检查数组交叉分叉，但是编译器采用的是一种非常著名的内存块复制例程的优化方法。仔细分析就会发现，它所复制的例程居然完全相同。这或许是 Intel C++编译器的某种不足吧。然而无论怎样，最后生成的程序功能正常。

本书刻意介绍这个例子，旨在让读者认识到：有些时候编译器的输出指令确实会让人感到摸不到头脑。然而只要程序功能正常，我们就不必进行深究了吧。

第40章 达夫装置

达夫装置（Duff's Device）是一种综合了多种控制语句的循环展开技术，可大幅度地减少代码的分支总数。这种循环展开技术巧妙地利用了 swtich 语句的滑梯（fallthrough）效应。

本章对 Tom Duff 的原始程序进行了轻度简化。

现在假设我们要编写一个清除一块连续内存的函数。当然，我们可以使用循环语句、逐个字节的复写数据。但是现代的计算机内存总线都很宽，以字节为单位地清除效率会非常低。若以 4 字节或 8 字节为操作单位进行 io 操作，那么操作效率会高一些。由于本例演示的是 64 位应用程序，所以我们就以 8 字节为单位进行操作。不过，我们又当如何应对那些不足 8 字节的内存空间？毕竟我们的函数也可能清除容量不足 8 字节的内存空间。

因此，合理的算法应当是：

- 首先统计目标空间含有多少个连续的 8 字节空间，继而以 8 字节（64 位）为操作单位将其清除。
- 然后统计那些大小不足 8 字节的尾数、即上一步除法计算的余数，然后逐字节地将之清零。

简单的循环语句即可完成第二步的任务。然而我们更希望把这个循环分解、展开：

```
#include <stdint.h>
#include <stdio.h>

void bzero(uint8_t* dst, size_t count)
{
        int i;

        if (count&(~7))
                // work out 8-byte blocks
                for (i=0; i<count>>3; i++)
                {
                        *(uint64_t*)dst=0;
                        dst=dst+8;
                };

        // work out the tail
        switch(count & 7)
        {
        case 7: *dst++ = 0;
        case 6: *dst++ = 0;
        case 5: *dst++ = 0;
        case 4: *dst++ = 0;
        case 3: *dst++ = 0;
        case 2: *dst++ = 0;
        case 1: *dst++ = 0;
        case 0: // do nothing
                break;
        }
}
```

我们来看看这个计算是如何完成的。待处理内存区域的大小是 64 位数据，它分为如下两个部分。

	7	6	5	4	3	2	1	0
…	B	B	B	B	B	S	S	S

备注：B 代表大小为 8 字节的内存块；S 代表大小不足 8 字节的尾部内存块。

当我们将输入的内存块的大小除以 8，其实就是将该值右移 3 位。然而不足 8 字节的内存块总数，即

这步除法计算的余数，恰恰是刚刚位移出去的那最后的三位。可见，把目标空间大小/即变量 count 右移 3 位，可求得它含有多少个 8 字节的内存块；令变量 counter 与数字 7 进行逻辑与运算可求得它有多少字节的尾部内存块。

当然，我们也得首先看看这块目标空间是否足够进行一次 8 字节的清除操作。因此首先就要检查变量 count 是否大于 7。为此，我们就要将变量 count 的最低三位清零，并将结果和零比较。如果该数大于零，则表示这个数量 count 大于 7，我们就可以进行 8 字节的块操作。当然我们不需要知道它到底比 7 大多少，只需知道它是否比 7 大，即 count 的高位是否为零。

当然之所以能这样做，主要是因为 8 是 2 的 3 次方，而且“某数除以 2 的 *n* 次方”通过位移计算即可实现，其他类型的数字就不能通过位移运算进行判断了。

很难说这些技术是不是值得采用，毕竟这种技巧会明显降低源程序的可读性。然而这已经属于常见技术了。凡是资深的编程人员，不论他愿否使用达夫装置，他都应当能够理解使用了这种技巧的程序。

第一部分其实很简单，用 64 位的零填充所有 8 字节内存块。

第二部分的难点在于其循环展开技术。达夫装置利用的是 switch()函数的滑梯效应。用人类的语言来讲，这段代码的功能就是将变量 count 与数字 7 进行逻辑与运算，得到尾数，然后把它们逐个清零。如果尾数为 0，那么直接跳转至函数尾声，不做任何操作。如果尾数是 1 的话，跳转到 switch()语句中清除单字节的那条语句。如果尾数是 2 的话，跳转到 swith()语句中相应的位置执行清零操作；由于滑梯效应的存在，函数会清除 2 个字节的空间，以此类推。当尾数的值为最大值 7 的时候，它会执行 7 次相同操作。 在这个算法中，不会出现尾数大于 7，也就是 8 的情况，因为第一部分的指令已经清除了所有 8 字节的内存块。

换而言之，达夫装置就是循环展开技术的一种特例。在老式设备上，它显然比普通循环的运行速度更高。然而，对于现代的大多数 CPU 来说，一些体积短小的循环语句反而会比循环展开体的执行速度更快。也许对于目前低成本的嵌入式 MCU（微控单元，例如单片机）处理器而言，达夫设备更有意义一些。

我们下面来看看优化后的 MSVC 2012 的一些行为。

```
dst$ = 8
count$ = 16
bzero   PROC
        test    rdx, -8
        je      SHORT $LN11@bzero
; work out 8-byte blocks
        xor     r10d, r10d
        mov     r9, rdx
        shr     r9, 3
        mov     r8d, r10d
        test    r9, r9
        je      SHORT $LN11@bzero
        npad    5
$LL19@bzero:
        inc     r8d
        mov     QWORD PTR [rcx], r10
        add     rcx, 8
        movsxd  rax, r8d
        cmp     rax, r9
        jb      SHORT $LL19@bzero
$LN11@bzero:
; work out the tail
        and     edx, 7
        dec     rdx
        cmp     rdx, 6
        ja      SHORT $LN9@bzero
        lea     r8, OFFSET FLAT:__ImageBase
        mov     eax, DWORD PTR $LN22@bzero[r8+rdx*4]
        add     rax, r8
        jmp     rax
$LN8@bzero:
        mov     BYTE PTR [rcx], 0
```

```
        inc     rcx
$LN7@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN6@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN5@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN4@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN3@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN2@bzero:
        mov     BYTE PTR [rcx], 0
$LN9@bzero:
        fatret  0
        npad    1
$LN22@bzero:
        DD      $LN2@bzero
        DD      $LN3@bzero
        DD      $LN4@bzero
        DD      $LN5@bzero
        DD      $LN6@bzero
        DD      $LN7@bzero
        DD      $LN8@bzero
bzero   ENDP
```

这个函数的第一部分与源程序一一对应。第二部分的循环展开体也不难理解：switch 语句通过转移指令直接跳到合适的位置。由于 MOV/INC 指令对之间没有其他代码，所以 swtich 语句在转移到指定标签后不会越过后续的转移标签，它会执行完所需的所有指令。

另外，我们注意到 MOV/INC 指令对占用固定的字节数（3+3=6 字节）。在注意到这个问题之后，我们就可以舍去 switch()语句的转移表结构，将输入值乘以 6，并直接跳转到如下地址：目前的 RIP 地址+输入值*6。这样一来，由于省掉了从转移表的查询操作，执行速度会更快。对于乘法来说，数字 6 是一个计算效率不高的乘数因子，或许乘法运算的速度比表查询的转移指令更慢；不过所谓“深度优化”就是这个思路[①]。在介绍循环展开技术时，过去的教科书就是这样介绍“达夫设备”的。

① 作为一个练习，为了去掉跳转表，读者可以尝试重新编写代码。上述 MOV/INC 指令对可以重写成 4 字节或者 8 字节，当然一个字节也可以，比如 STOSB 指令。

第 41 章　除以 9

我们来看一个非常简单的函数：

```
int f(int a)
{
        return a/9;
};
```

41.1　x86

x86 平台编译器的编译方法十分直白：

指令清单 41.1　MSVC

```
_a$ = 8          ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq              ; sign extend EAX to EDX:EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f  ENDP
```

IDIV 指令是除法指令。它会从寄存器对 EDX:EAX 中提取被除数、从 ECX 寄存器中提取除数。计算结束以后，它把计算结果/商存储在 EAX 寄存器里，把余数存储在 EDX 寄存器。除法计算之后，商就位于 EAX 寄存器里，直接成为 f()函数的返回值；因此没有其他值传递的操作。为了通知 IDIV 指令从 EDX:EAX 寄存器对中提取 64 位被除数，编译器在 IDIV 指令之前分派了 CDQ 指令。IDIV 指令就会进行 MOVSX 那样的符号位处理和数据扩展处理。

启用编译器的优化选项之后，可得到下述程序：

指令清单 41.2　采用 MSVC 优化

```
_a$ = 8                        ; size = 4
_f    PROC
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177      ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31             ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

编译器用乘法指令来变相实现除法运算。大家知道乘法会比除法运行快很多。使用这里介绍[1]的方法可以有效提高程序效率、节省时间开销。

① 可以看 War02 pp.10-3 中介绍的用乘法来代替除法的部分。

在编译优化中，我们常常将其称为“强度减轻”的办法。

若使用 GCC 4.4.1 编译此程序，即使我们刻意不启用其优化选项，GCC 的非优化编译结果也足以和 MSVC 的优化编译结果媲美。

指令清单 41.3　不带优化的 GCC 4.4.1

```
        public f
f       proc near

arg_0 = dword ptr  8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     edx, 954437177 ; 38E38E39h
        mov     eax, ecx
        imul    edx
        sar     edx, 1
        mov     eax, ecx
        sar     eax, 1Fh
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        pop     ebp
        retn
f       endp
```

41.2　ARM

ARM 处理器和其他的 RISC 处理器一样，“纯洁”得不支持硬件级别的除法指令。此外，这种 CPU 还不难“直接”进行 32 位常量的乘法运算（32 位 opcode 容纳不下 32 位常量）。因此，在进行除法运算时，编译器会混合加减法运算和位移运算、变相实现除法运算（详情请参阅第 19 章）。

本节引用参考书目 Ltd94（第 3.3 节）的一个例子，介绍一个“32 位数除以 10”的例子，分别计算商和余数。

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10             ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up remainder
    MOV    pc, lr
```

41.2.1　ARM 模式下，采用 Xcode 4.6.3（LLVM）优化

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3 MOV     R1, 0x38E38E39
__text:00002C60 10 F1 50 E7             SMMUL   R0, R0, R1
__text:00002C64 C0 10 A0 E1             MOV     R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0             ADD     R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1             BX      LR
```

这里的代码和采用优化算法时的 MSVC 与 GCC 基本相同。很明显，LLVM 采用了相同的算法来处

理常数。

细心的读者可能会问：既然 ARM 模式的单条指令不能把 32 位立即数赋值给寄存器，那么这个程序又是怎样做到单条 MOV 指令赋值的呢？实际上，原始指令并非是 IDA 显示的那种单条 MOV 指令。仔细观察您就会发现，那“条”指令占用了 8 个字节，而标准的 ARM 指令只有 4 个字节。原始的指令分两步进行 32 位赋值：首先用 MOV 指令将低 16 位（本例是常量 0x8E39）复制到寄存器的低 16 位，再用 MOVT 指令把立即数的高 16 位复制到寄存器的高 16 位。IDA 能够识别出这种指令组合，为了便于读者理解，把两条指令“排版”为一条“伪指令”。这 8 字节的 MOV 指令实际上是 2 条指令。

SMMUL 是 Signed Most Significant Word Multiply 的简称。它是 2 个 32 位有符号数的乘法运算指令，会把 64 位结果的高 32 位保存在寄存器 R0 中，舍弃结果中的低 32 位。

MOV R1,R0,ASR#1 是算术右移 1 位的运算指令。

而指令 ADD R0,R1,R0,LSR#31 的执行结果相当于将 R0 的值右移 31 位，并与 R1 的值相加，其结果保存在 R0 中。也就是：R0=R1+R0>>31。

在 ARM 模式的指令中没有单独的位移指令。不过，它可以在 MOV、ADD、SUB 以及 RSB[①]指令中，使用“后缀”形式的参数调节符对第二个操作数进行位移运算。在使用位移调节符的时候，应当指定位移的确切位数。

ASR 是算术右移 Arithmetic Shift Right 的简称。算术右移需要考虑符号位。

LSR 是逻辑右移 Logical Shift Right 的简称。逻辑右移不考虑符号位。

41.2.2 Thumb-2 模式下的 Xcode 4.6.3 优化（LLVM）

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

Thumb 模式的指令集里有单独的位移运算指令。本例中的 ASRS 就是算术右移指令。

41.2.3 非优化的 Xcode 4.6.3(LLVM) 以及 Keil 6/2013

在没有启用优化选项的情况下，LLVM 编译器不会生成上面那种混合运算指令，它会调用仿真库里的模拟运算函数__divsi3。

然而，无论是否启用优化选项，Keil 编译器都只会调用库函数__aeabi_idivmod。

41.3 MIPS

出于某些原因，优化的 GCC 4.4.5 有除法指令。

指令清单 41.4 优化的 GCC 4.4.5（IDA）

```
f:
                li      $v0, 9
                bnez    $v0, loc_10
                div     $a0, $v0 ; branch delay slot
                break   0x1C00   ; "break 7" in assembly output and objdump

loc_10:
                mflo    $v0
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

① 这些指令也称为“数据处理指令”。

本例出现了新指令 BREAK。它是由编译器产生的异常处理指令，会在除数为零的情况下抛出错误信息。毕竟在正规的数学概念里，除数不可以是零。即使在启用优化选项的情况下，GCC 也未能判断出本例“除数$V0 用于不会是零”的切实情况，机械性地分派了异常处理的检测指令。BREAK 指令只是一个在除数为零的情况下通知操作系统进行异常处理的指令。只要除数不是零，程序将会执行 MFLO 指令，把 LO 寄存器里的商复制到$V0 寄存器。

这里顺便说明一下，乘法指令 MUL 会把积的高 32 位保存在 HI 寄存器中，把积的低 32 位保存在 LO 寄存器中。而除法指令 DIV 则会把商保存在 LO 寄存器，把余数在保存在 HI 寄存器中。

如果把源程序的除法指令修改为“a % 9”、计算余数，那么编译之后的程序就会用 MFHI 指令替换本例中的 MFLO 指令。

41.4　它是如何工作的

在引入 2 的 n 次方之后，除法运算可转换为乘法运算：

$$\text{result}=\frac{\text{input}}{\text{divisor}}=\frac{\text{input}\cdot\frac{2^n}{\text{divisor}}}{2^n}=\frac{\text{input}\cdot M}{2^n}$$

这里的 M 是魔术因子（Magic coefficient）。其计算公式是

$$M=\frac{2^n}{\text{divisor}}$$

最终，除法运算就转换成了

$$\text{result}=\frac{\text{input}\bullet M}{2^n}$$

“除以 2 的 n 次方”的运算可以直接通过右移操作实现。如果 n 小于 32，那么中间运算的积的低 n 位（通常位于 EAX 或者 RAX 寄存器）就会被位移运算直接抹去；如果 n 大于或等于 32，那么积的高半部分（通常位于 EDX 或者 RDX 寄存器）的数值都会受到影响。

可见，参数 n 的取值直接决定了转换运算的计算精度。

在进行有符号数的除法运算时，符号位也对计算精度及 n 的取值产生了显著影响。

下面这个例子将验证符号位的影响。

```
int f3_32_signed(int a)
{
        return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
        return a/3;
};
```

在无符号数的计算过程中，魔术因子是 0xaaaaaaab。乘法的中间结果要除以 2 的 33 次方。

而在有符号数的计算过程中，魔术因子则是 0x55555556，乘法的中间结果要除以 2 的 32 次方。虽然这里没有进行除法运算，但是根据前面的讨论可知：商的有效位取自于 EDX 寄存器。

请别忘记中间一步的乘法计算同样存在符号位的问题：积的高 32 位右移 31 位，将在 EAX 寄存器的最低数权位保留有符号数的符号位（正数为 0，负数为 1）。将符号位加入积的高 32 位值，可实现负数补码的“+1”修正

指令清单 41.5　带优化的 MSVC 2012

```
_f3_32_unsigned PROC
        mov     eax, -1431655765        ; aaaaaaabH
```

```
        mul     DWORD PTR _a$[esp-4] ; unsigned multiply
; EDX=(input*0xaaaaaaab)/2^32
        shr     edx, 1
; EDX=(input*0xaaaaaaab)/2^33
        mov     eax, edx
        ret     0
_f3_32_unsigned ENDP

_f3_32_signed PROC
        mov     eax, 1431655766         ; 55555556H
        imul    DWORD PTR _a$[esp-4] ; signed multiply
; take high part of product
; it is just the same as if to shift product by 32 bits right or to divide it by 2^32
        mov     eax, edx        ; EAX=EDX=(input*0x55555556)/2^32
        shr     eax, 31         ; 0000001fH
        add     eax, edx        ; add 1 if sign is negative
        ret     0
_f3_32_signed ENDP
```

41.4.1 更多的理论

其实，我们都知道，乘法和除法互为逆运算。因此下面的除法可以用乘法来代替。我们可以表述为

$$\frac{x}{c}=x\frac{1}{c}$$

$1/c$ 可以称为乘法的逆运算，是 c 的倒数。可以用编译器来做提前运算。

但是这是为浮点计算用的，有没有整数呢？在模算术计算环境下，是可能会有的。CPU 寄存器的长度是很规整的，要么 32 位要么 64 位，因此几乎所有的寄存器的算术操作都是对 2 的 32 次方或者 2 的 64 次方进行操作。

可以查阅 War02 的第 10 章第 3 节。

41.5 计算除数

41.5.1 变位系数#1

通常我们看到的代码可能就像这样：

```
        mov     eax, MAGICAL CONSTANT
        imul    input value
        sar     edx, SHIFTING COEFFICIENT ; signed division by 2^x using arithmetic shift right
        mov     eax, edx
        shr     eax, 31
        add     eax, edx
```

我们这里用大写字母 M 来代表 32 位的魔术因子，把变位系数记为 C，把除数记为 D。

因此除数可以表示为

$$D=\frac{2^{32+C}}{M}$$

指令清单 41.6 优化的 MSVC 2012 代码

```
        mov     eax, 2021161081                 ; 78787879H
        imul    DWORD PTR _a$[esp-4]
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31                         ; 0000001fH
        add     eax, edx
```

用公式可以表示为

$$D=\frac{2^{32+3}}{2021161081}$$

这个数字超过了 32 位的表达范围。我们可使用 Mathematica 程序计算除数：

指令清单 41.7　Wolfram Mathematica 的计算结果

```
In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.
```

它可算出本例采用的除数是 17。

在 64 位数据的除法运算中，计算除数的方法完全相同。只是不再采用 2 的 32 次方，而采用了 2 的 64 次方。

```
uint64_t f1234(uint64_t a)
{
        return a/1234;
};
```

指令清单 41.8　64 位下的优化 MSVC2012

```
f1234   PROC
        mov     rax, 7653754429286296943            ; 6a37991a23aead6fH
        mul     rcx
        shr     rdx, 9
        mov     rax, rdx
        ret     0
f1234   ENDP
```

指令清单 41.9　WolframMathematica 的计算结果

```
In[1]:=N[2^(64+9)/16^^6a37991a23aead6f]
Out[1]:=1234.
```

41.5.2　变位系数#2

变位系数确实可能为零：

```
        mov     eax, 55555556h ; 1431655766
        imul    ecx
        mov     eax, edx
        shr     eax, 1Fh
```

这样，计算除数的方法就更简单一些了：

$$D=\frac{2^{32}}{M}$$

就本例而言，除数的计算方法是：

$$D=\frac{2^{32}}{1431655766}$$

再次使用 Mathematica 程序计算除数

指令清单 41.10　Wolfram Mathematica 的计算结果

```
In[1]:=N[2^32/16^^55555556]
Out[1]:=3.
```

最终求得除数为 3。

41.6 练习题

请描述下述代码的功能。

指令清单 41.11 采用 MSVC 2010 优化的代码

```
_a$ = 8
_f      PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        mov     eax, -968154503 ; c64b2279H
        imul    ecx
        add     edx, ecx
        sar     edx, 9
        mov     eax, edx
        shr     eax, 31         ; 0000001fH
        add     eax, edx
        ret     0
_f      ENDP
```

指令清单 41.12 ARM64 位下采用 GCC 4.9 优化

```
f:
        mov     w1, 8825
        movk    w1, 0xc64b, lsl 16
        smull   x1, w0, w1
        lsr     x1, x1, 32
        add     w1, w0, w1
        asr     w1, w1, 9
        sub     w0, w1, w0, asr 31
        ret
```

答案请参见 G.1.14。

第 42 章　字符串转换成数字，函数 atoi()

我们来重新实现一下标准的 C 函数 atoi()。这是一个将字符串转换成整数的函数。

42.1　例 1

本例可按照 ASCII 表把数字字符转换为数字。代码没有做容错处理，因此当输入值为非数字型字符时，返回值也就不会正确。

```
#include <stdio.h>

int my_atoi (char *s)
{
        int rt=0;

        while (*s)
        {
                rt=rt*10 + (*s-'0');
                s++;
        };

        return rt;
};

        int main()
        {
                printf ("%d\n", my_atoi ("1234"));
                printf ("%d\n", my_atoi ("1234567890"));
};
```

这个算法实现的是从左到右读取数字，并将每个读取的字符减去数字 0 的 ASCII 值。我们知道，在 ASCII 表中，数字 0～9 是按照递增顺序连续存放的。因此无论字符“0”对应的数值是多少，“0”的值减去“0”的值为 0，“9”的值减去“0”的值为 9。按照这种模式，函数就可把单字节字符转换为相应数值。因此，如果函数读取的字符不是数字型字符的话，计算结果就不会正确。在处理多字符字符串时还要考虑数权问题，本例会把前一步的结果（即变量 rt）乘以 10，再累加本次转换出来的数字。换句话说，在转换单个字符的每次迭代过程中，转换函数都应当给上一次迭代的转换结果分配一次合理的数权。最后一个被转换的字符不会被提高数权。

42.1.1　64 位下的 MSVC 2013 优化

指令清单 42.1　64 位下的 MSVC2013 优化

```
s$ = 8
my_atoi PROC
; load first character
        movzx   r8d, BYTE PTR [rcx]
; EAX is allocated for "rt" variable
; its 0 at start'
        xor     eax, eax
; first character is zero-byte, i.e., string terminator?
; exit then.
        test    r8b, r8b
        je      SHORT $LN9@my_atoi
```

```
$LL2@my_atoi:
        lea     edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
        movsx   eax, r8b
; EAX=input character
; load next character to R8D
        movzx   r8d, BYTE PTR [rcx+1]
; shift pointer in RCX to the next character:
        lea     rcx, QWORD PTR [rcx+1]
        lea     eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=input character + rt*5*2=input character + rt*10
; correct digit  by subtracting 48 (0x30 or '0')
        add     eax, -48                                ; ffffffffffffffd0H
; was last character zero?
        Test    r8b, r8b
; jump to loop begin, if not
        jne     SHORT $LL2@my_atoi
$LN9@my_atoi:
        ret     0
my_atoi ENDP
```

字符串的第一个字符可能就是终止符。在这种情况下，函数不应当进入字符转换的迭代过程。此外，编译器没有分配“乘以 10”的乘法指令，而是使用了第一条、第三条 LEA 指令分步实现了“上次迭代的 rt 值乘以 5”、“输入字符+5 倍原始 rt 值×2”的运算。某些情况下，MSVC 编译器会刻意避开减法运算的 SUB 指令、转而分配“ADD 某个负数”的指令来实现减法运算。本例就发生了这种情况。虽然笔者也不明白 ADD 指令的优越性到底在哪，但是 MSVC 编译器经常会做这种指令替换。

42.1.2 64 位下的 GCC 4.9.1 优化

GCC 4.9.1 优化更精确，但是它会在代码的最后添加一个多余的返回指令 RET。其实一个 RET 就足够了。

指令清单 42.2 64 位下的 GCC 4.9.1 优化

```
my_atoi:
; load input character into EDX
        movsx   edx, BYTE PTR [rdi]
; EAX is allocated for "rt" variable
        xor     eax, eax
; exit, if loaded character is null byte
        test    dl, dl
        je      .L4
.L3:
        lea     eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; shift pointer to the next character:
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
; EAX=input character - 48 + RAX*2 = input character - '0' + rt*10
; load next character:
        movsx   edx, BYTE PTR [rdi]
; goto loop begin, if loaded character is not null byte
        test    dl, dl
        jne     .L3
        rep ret
.L4:
        rep ret
```

42.1.3 ARM 模式下 Keil 6/2013 优化

指令清单 42.3 ARM 模式下 Keil 6/2013 优化

```
my_atoi PROC
```

```
; R1 will contain pointer to character
        MOV     r1,r0
; R0 will contain "rt" variable
        MOV     r0,#0
        B       |L0.28|
|L0.12|
        ADD     r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
        ADD     r0,r2,r0,LSL #1
; R0=input character + rt*5<<1 = input character + rt*10
; correct whole thing by subtracting '0' from rt:
        SUB     r0,r0,#0x30
; shift pointer to the next character:
        ADD     r1,r1,#1
|L0.28|
; load input character to R2
        LDRB    r2,[r1,#0]
; is it null byte? if no, jump to loop body.
        CMP     r2,#0
        BNE     |L0.12|
; exit if null byte.
; "rt" variable is still in R0 register, ready to be used in caller function
        BX      lr
        ENDP
```

42.1.4　Thumb 模式下 Keil 6/2013 优化

指令清单 42.4　Thumb 模式下 Keil 6/2013 优化

```
my_atoi PROC
; R1 will be pointer to the input character
        MOVS    r1,r0
; R0 is allocated to "rt" variable
        MOVS    r0,#0
        B       |L0.16|
|L0.6|
        MOVS    r3,#0xa
; R3=10
        MULS    r0,r3,r0
; R0=R3*R0=rt*10
; shift pointer to the next character:
        ADDS    r1,r1,#1
; correct whole thing by subtracting 0' character from it':
        SUBS    r0,r0,#0x30
        ADDS    r0,r2,r0
; rt=R2+R0=input character + (rt*10 - '0')
|L0.16|
; load input character to R2
        LDRB    r2,[r1,#0]
; is it zero?
        CMP     r2,#0
; jump to loop body if it is not
        BNE     |L0.6|
; rt variable in R0 now, ready to be used in caller function
        BX      lr
        ENDP
```

很有趣的是，我们以前上学时学习到的规则是：加法和减法的运算顺序无所谓先后，完全可以重排。本例就活用了这一原则：编译器首先计算乘/减法的混合运算表达式“rt*10–‘0’”、求得中间差，然后计算输入字符与中间差的和。虽然这种算法的计算结果必定和源代码的计算结果相同，但是请不要忽视编译器的算式重排功能。

42.1.5 ARM64 下的 GCC 4.9.1 优化

ARM64 的编译器可以使用预增量指令后缀。

指令清单 42.5 ARM64 的 GCC 4.9.1 指令优化

```
my_atoi:
; load input character into W1
        ldrb    w1, [x0]
        mov     x2, x0
; X2=address of input string
; is loaded character zero?
; jump to exit if its so'
; W1 will contain 0 in this case.
; it will be reloaded into W0 at L4.
        cbz     w1, .L4
; W0 will contain "rt" variable
; initialize it at zero:
        mov     w0, 0
.L3:
; subtract 48 or '0' from input variable and put result into W3:
        sub     w3, w1, #48
; load next character at address X2+1 into W1 with pre-increment:
        ldrb    w1, [x2,1]!
        add     w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
        add     w0, w3, w0, lsl 1
; W0=input digit + W0<<1 = input digit + rt*5*2 = input digit + rt*10
; if the character we just loaded is not null byte, jump to the loop begin
        cbnz    w1, .L3
; variable to be returned (rt) is in W0, ready to be used in caller function
        ret
.L4:
        mov     w0, w1
        ret
```

42.2 例 2

现在，我们把例 1 的源代码改得更高级一些：令其检测第一个字符是否为负数符号“-”，并且检测输入值里有没有非数字型字符；当输入字符串含有非数字字符的时候，要令程序提示错误信息。

```
#include <stdio.h>

int my_atoi (char *s)
{
        int negative=0;
        int rt=0;

        if (*s=='-')
        {
                negative=1;
                s++;
        };

        while (*s)
        {
                if (*s<'0' || *s>'9')
                {
                        printf ("Error! Unexpected char: '%c'\n", *s);
                        exit(0);
                };
                rt=rt*10 + (*s-'0');
```

```
                s++;
        };

        if (negative)
                return -rt;
        return rt;
};

int main()
{
        printf ("%d\n", my_atoi ("1234"));
        printf ("%d\n", my_atoi ("1234567890"));
        printf ("%d\n", my_atoi ("-1234"));
        printf ("%d\n", my_atoi ("-1234567890"));
        printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

42.2.1　64 位下的 GCC 4.9.1 优化

指令清单 42.6　64 位下的 GCC 4.9.1 优化

```
.LC0:
        .string "Error! Unexpected char: '%c'\n"

my_atoi:
        sub     rsp, 8
        movsx   edx, BYTE PTR [rdi]
; check for minus sign
        cmp     dl, 45 ; '-'
        je      .L22
        xor     esi, esi
        test    dl, dl
        je      .L20
.L10:
; ESI=0 here if there was no minus sign and 1 if it was
        lea     eax, [rdx-48]
; any character other than digit will result unsigned number greater than 9 after subtraction
; so if it is not digit, jump to L4, where error will be reported:
        cmp     al, 9
        ja      .L4
        xor     eax, eax
        jmp     .L6
.L7:
        lea     ecx, [rdx-48]
        cmp     cl, 9
        ja      .L4
.L6:
        lea     eax, [rax+rax*4]
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
        movsx   edx, BYTE PTR [rdi]
        test    dl, dl
        jne     .L7
; if there was no minus sign, skip NEG instruction
; if it was, execute it.
        test    esi, esi
        je      .L18
        neg     eax
.L18:
        add     rsp, 8
        ret
.L22:
        movsx   edx, BYTE PTR [rdi+1]
        lea     rax, [rdi+1]
        test    dl, dl
```

```
        je      .L20
        mov     rdi, rax
        mov     esi, 1
        jmp     .L10
.L20:
        xor     eax, eax
        jmp     .L18
.L4:
; report error. character is in EDX
        mov     edi, 1
        mov     esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '%c'\n"
        xor     eax, eax
        call    __printf_chk
        xor     edi, edi
        call    exit
```

如果字符串的第一个字符是负号，那么就要在转换的最后阶段执行 NEG 指令，把结果转换为负数。

另外值得一提的是“检测字符是否是数字字符”的判断表达式。我们在程序中可以看到代码为：

```
if (*s<'0' || *s>'9')
    ...
```

这是两个比较操作。比较有意思的是，我们完全可以只用一个比较指令就完成这两步比较运算：将输入字符的值减去“0”字符的值，将结果视为无符号数（这是关键）与 9 进行比较。若最后的无符号数比 9 还大，那么输入字符就不是数字字符。

比如说，如果我们在输入的字符串中含有小数点(“.”)，这个符号在 ASCII 表中排在字符零“0”的前 2 位。因此判断语句的减法运算表达式是：46–48=–2。有符号数的减法运算当然会求得有符号数。被减数比减数小，计算的结果肯定是负数。但是，如果我们把这个结果当作无符号数来处理的话，它将是 0xfffffffe（对应的十进制数是 42949672294），显然它比 9 大。举这个例子是想说明按照无符号数处理的重要性。

编译器经常会这样做，因此我们应该重新认识这种技巧。

我们可以本书的其他地方看到这个例子：比如 48.1.2 节。

在编译 64 位应用程序时，MSVC 2013 还好使用这种优化技巧。

42.2.2 ARM 模式下的 Keil6/2013 优化

指令清单 42.7 ARM 模式下的 Keil6/2013 优化

```
1  my_atoi PROC
2          PUSH     {r4-r6,lr}
3          MOV      r4,r0
4          LDRB     r0,[r0,#0]
5          MOV      r6,#0
6          MOV      r5,r6
7          CMP      r0,#0x2d '-'
8  ; R6 will contain 1 if minus was encountered, 0 if otherwise
9          MOVEQ    r6,#1
10         ADDEQ    r4,r4,#1
11         B        |L0.80|
12 |L0.36|
13         SUB      r0,r1,#0x30
14         CMP      r0,#0xa
15         BCC      |L0.64|
16         ADR      r0,|L0.220|
17         BL       __2printf
18         MOV      r0,#0
19         BL       exit
20 |L0.64|
21         LDRB     r0,[r4],#1
22         ADD      r1,r5,r5,LSL #2
23         ADD      r0,r0,r1,LSL #1
```

```
24          SUB         r5,r0,#0x30
25  |L0.80|
26          LDRB        r1,[r4,#0]
27          CMP         r1,#0
28          BNE         |L0.36|
29          CMP         r6,#0
30  ; negate result
31          RSBNE       r0,r5,#0
32          MOVEQ       r0,r5
33          POP         {r4-r6,pc}
34          ENDP
35
36  |L0.220|
37          DCB         "Error! Unexpected char: '%c'\n",0
```

32 位 ARM 的指令集里没有 NEG（取负）指令，因此编译器分配了第 31 行的“反向减法”指令。当第 29 行指令——即 CMP 的比较结果为“不相等”时，才会执行第 31 行的条件执行指令（我们可以看到 NE 后缀，也就是“（运行条件为）Not Equal”的意思）。而后，RSBNE 指令用 0 减去上一步的计算结果。这条指令确实就是减法运算指令，只是被减数和减数的操作符排列位置对调了一下。用 0 来减去任何数，实际的效果就是对该数取负。实现的结果可以用这个来表示：$0-x=-x$。

Thumb 程序的运算模式几乎完全相同。

在编译 ARM64 平台的应用程序时，GCC 4.9 能够分配 NEG（取反）指令。

42.3　练习

这里顺便提一下，安全研究人员经常研究各种异常情况。他们重点关注不合乎预期的输入值，通过这种数据诱使程序进行某种与设计思路相左的行为。因此，他们也会关注模糊测试方法。作为练习，我们可以试图输入非数字字符，看看会发生什么。自己可以尝试着解释一下背后的成因是什么。

第43章 内联函数

在编译阶段，将会被编译器把函数体展开并嵌入到每一个调用点的函数，就是内联函数。

指令清单 43.1 一个简单的例子

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
        return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
        int celsius=atol(argv[1]);
        printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

此函数在汇编层面的具体指令与源代码几乎一一对应。然而，如果在 GCC 编译环境下，我们采用-03参数优化的话，我们会看到如下所示的代码。

指令清单 43.2 GCC 4.8.1 优化

```
_main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        call    __main
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp], eax
        call    _atol
        mov     edx, 1717986919
        mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
        lea     ecx, [eax+eax*8]
        mov     eax, ecx
        imul    edx
        sar     ecx, 31
        sar     edx
        sub     edx, ecx
        add     edx, 32
        mov     DWORD PTR [esp+4], edx
        call    _printf
        leave
        ret
```

值得注意的是，编译器用乘法指令变相地实现了除法运算（可以参见第 41 章）。

您没看错，温度转换函数 celsius_to_fahrenheit()（摄氏温度转换成华氏温度）的函数体被直接展开，放在了函数 printf()（显示字符串）的前面。为什么呢？因为这样运行速度会更快一些。这种代码不需要在调用函数时处理多余的函数调用和返回指令。

现在具有优化功能的编译器一般都能自动的把小型函数的函数体直接“嵌入”到调用方函数的代码里。当然我们也可以借助关键字“inline”强制编译器进行这种“嵌入”处理。

43.1　字符串和内存操作函数

在处理字符串和内存操作的常见函数时，例如：strycpy()、strcmp()、strlen、memset()、memcpy()、memcmp()函数，编译器通常会把这些函数当作内联函数处理。

多数情况下，以内联函数编译的函数会比那些被单独调用的函数运行得更快。

本节将演示一些非常具有特征的内联代码，以供读者研究。

43.1.1　字符串比较函数 strcmp()

指令清单 43.3　字符串比较函数 strcmp()

```
bool is_bool (char *s)
{
        if (strcmp (s, "true")==0)
                return true;
        if (strcmp (s, "false")==0)
                return false;
        assert(0);
};
```

指令清单 43.4　采用 GCC 4.8.1 优化的例子

```
.LC0:
        .string "true"
.LC1:
        .string "false"
is_bool:
.LFB0:
        push    edi
        mov     ecx, 5
        push    esi
        mov     edi, OFFSET FLAT:.LC0
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        repz cmpsb
        je      .L3
        mov     esi, DWORD PTR [esp+32]
        mov     ecx, 6
        mov     edi, OFFSET FLAT:.LC1
        repz cmpsb
        seta    cl
        setb    dl
        xor     eax, eax
        cmp     cl, dl
        jne     .L8
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L8:
        mov     DWORD PTR [esp], 0
        call    assert
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L3:
        add     esp, 20
        mov     eax, 1
        pop     esi
        pop     edi
        ret
```

指令清单 43.5 采用 MSVC 2010 优化的例子

```
$SG3454 DB      'true', 00H
$SG3456 DB      'false', 00H

_s$ = 8         ; size = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
        push    esi
        mov     esi, DWORD PTR _s$[esp]
        mov     ecx, OFFSET $SG3454 ; 'true'
        mov     eax, esi
        npad    4 ; align next label
$LL6@is_bool:
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN7@is_bool
        test    dl, dl
        je      SHORT $LN8@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
        jne     SHORT $LN7@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL6@is_bool
$LN8@is_bool:
        xor     eax, eax
        jmp     SHORT $LN9@is_bool
$LN7@is_bool:
        sbb     eax, eax
        sbb     eax, -1
$LN9@is_bool:
        test    eax, eax
        jne     SHORT $LN2@is_bool

        mov     al, 1
        pop     esi

        ret     0
$LN2@is_bool:

        mov     ecx, OFFSET $SG3456 ; 'false'
        mov     eax, esi
$LL10@is_bool:
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN11@is_bool
        test    dl, dl
        je      SHORT $LN12@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
        jne     SHORT $LN11@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL10@is_bool
$LN12@is_bool:
        xor     eax, eax
        jmp     SHORT $LN13@is_bool
$LN11@is_bool:
        sbb     eax, eax
        sbb     eax, -1
$LN13@is_bool:
        test    eax, eax
```

```
        jne     SHORT $LN1@is_bool

        xor     al, al
        pop     esi

        ret     0
$LN1@is_bool:

        push    11
        push    OFFSET $SG3458
        push    OFFSET $SG3459
        call    DWORD PTR __imp__wassert
        add     esp, 12
        pop     esi

        ret     0
?is_bool@@YA_NPAD@Z ENDP ; is_bool
```

43.1.2　字符串长度函数 strlen()

指令清单 43.6　字符串长度函数 strlen()的例子

```
int strlen_test(char *s1)
{
        return strlen(s1);
};
```

指令清单 43.7　采用 MSVC 2010 优化的例子

```
_s1$ = 8 ; size = 4
_strlen_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        lea     edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
        mov     cl, BYTE PTR [eax]
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strlen_tes
        sub     eax, edx
        ret     0
_strlen_test ENDP
```

43.1.3　字符串复制函数 strcpy()

指令清单 43.8　字符串复制函数 strcpy()的例子

```
void strcpy_test(char *s1, char *outbuf)
{
        strcpy(outbuf, s1);
};
```

指令清单 43.9　采用 MSVC 2010 优化的例子

```
_s1$ = 8        ; size = 4
_outbuf$ = 12   ; size = 4
_strcpy_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        mov     edx, DWORD PTR _outbuf$[esp-4]
        sub     edx, eax
        npad    6 ; align next label
$LL3@strcpy_tes:
        mov     cl, BYTE PTR [eax]
```

```
        mov     BYTE PTR [edx+eax], cl
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strcpy_tes
        ret     0
_strcpy_test ENDP
```

43.1.4 内存设置函数 memset()

例子#1 如下所示。

指令清单 43.10 32 字节的操作

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 32);
};
```

在编译那些操作小体积内存块的 memset()函数时，多数编译器不会分配标准的函数调用指令（call），反而会分配一堆的 MOV 指令，直接赋值。

指令清单 43.11 64 位下的 GCC 4.9.1 优化

```
f:
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+8], 0
        mov     QWORD PTR [rdi+16], 0
        mov     QWORD PTR [rdi+24], 0
        ret
```

这让我们想起了本书 14.1.4 节介绍的循环展开技术。

例子#2 如下所示。

指令清单 43.12 67 字节内存的操作

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 67);
};
```

当内存块的大小不是 4 或者 8 的整数倍时，不同的编译器会有不同的处理方法。

比如说，MSVC 2012 依旧会分配一串 MOV 指令。

指令清单 43.13 64 位下的 MSVC 2012 优化

```
out$ = 8
f       PROC
        xor     eax, eax
        mov     QWORD PTR [rcx], rax
        mov     QWORD PTR [rcx+8], rax
        mov     QWORD PTR [rcx+16], rax
        mov     QWORD PTR [rcx+24], rax
        mov     QWORD PTR [rcx+32], rax
        mov     QWORD PTR [rcx+40], rax
        mov     QWORD PTR [rcx+48], rax
        mov     QWORD PTR [rcx+56], rax
        mov     WORD PTR [rcx+64], ax
        mov     BYTE PTR [rcx+66], al
        ret     0
f       ENDP
```

GCC 还会分配 REP STOSQ 指令。这可能比一堆的 MOV 赋值指令更短，效率更高。

指令清单 43.14　64 位下的 GCC 4.9.1 优化

```
f:
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+59], 0
        mov     rcx, rdi
        lea     rdi, [rdi+8]
        xor     eax, eax
        and     rdi, -8
        sub     rcx, rdi
        add     ecx, 67
        shr     ecx, 3
        rep stosq
        ret
```

43.1.5　内存复制函数 memcpy()

在编译那些复制小尺寸内存块的 memcpy()函数时，多数编译器会分配一系列的 MOV 指令。

指令清单 43.15　内存复制函数 memcpy()

```
void memcpy_7(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 7);
};
```

指令清单 43.16　采用 MSVC 2010 优化

```
_inbuf$ = 8      ; size = 4
_outbuf$ = 12    ; size = 4
_memcpy_7 PROC
        mov     ecx, DWORD PTR _inbuf$[esp-4]
        mov     edx, DWORD PTR [ecx]
        mov     eax, DWORD PTR _outbuf$[esp-4]
        mov     DWORD PTR [eax+10], edx
        mov     dx, WORD PTR [ecx+4]
        mov     WORD PTR [eax+14], dx
        mov     cl, BYTE PTR [ecx+6]
        mov     BYTE PTR [eax+16], cl
        ret     0
_memcpy_7 ENDP
```

指令清单 43.17　采用 GCC 4.8.1 优化

```
memcpy_7:
        push    ebx
        mov     eax, DWORD PTR [esp+8]
        mov     ecx, DWORD PTR [esp+12]
        mov     ebx, DWORD PTR [eax]
        lea     edx, [ecx+10]
        mov     DWORD PTR [ecx+10], ebx
        movzx   ecx, WORD PTR [eax+4]
        mov     WORD PTR [edx+4], cx
        movzx   eax, BYTE PTR [eax+6]
        mov     BYTE PTR [edx+6], al
        pop     ebx
        ret
```

上述指令的操作流程是：首先复制 4 个字节，然后复制一个字（也就是 2 个字节），接着复制最后一个字节。

此外，编译器还会通过赋值指令 MOV 复制结构体（structure）型数据。详情请参见本书的 21.4.1 节。

大尺寸内存块的操作：

不同的编译器会有不同的指令分配方案。

指令清单 43.18 memcpy()内存复制的例子（这里列出了 2 个不同的例子，一个是 128 字节的操作，另外一个则是 123 字节的内存操作）

```
void memcpy_128(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 128);
};

void memcpy_123(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 123);
};
```

MSVC 分配了单条 MOVSD 指令。在循环控制变量 ECX 的配合下，MOVSD 可一步完成 128 个字节的数据复制。其原因显然是 128 能被 4 整除。

指令清单 43.19 MSVC 2010 优化

```
_inbuf$ = 8                ; size = 4
_outbuf$ = 12              ; size = 4
_memcpy_128 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 32
        rep movsd
        pop     edi
        pop     esi
        ret     0
_memcpy_128 ENDP
```

在复制 123 个字节的程序里，编译器首先通过 MOVSD 指令复制 30 个 32[①]字节（也就是 120 字节），然后依次通过 MOVSW 指令和 MOVSB 指令复制 2 个字节和 1 个字节。

指令清单 43.20 采用 MSVC 2010 优化指令

```
_inbuf$ = 8                ; size = 4
_outbuf$ = 12              ; size = 4
_memcpy_123 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 30
        rep movsd
        movsw
        movsb
        pop     edi
        pop     esi
        ret     0
_memcpy_123 ENDP
```

GCC 则分配了一个大型的通用的函数。这个函数适用于任意大小的内存块复制操作。

① 应该为 4 个字节。

指令清单 43.21　采用 GCC 4.8.1 优化

```
memcpy_123:
.LFB3:
        push    edi
        mov     eax, 123
        push    esi
        mov     edx, DWORD PTR [esp+16]
        mov     esi, DWORD PTR [esp+12]
        lea     edi, [edx+10]
        test    edi, 1
        jne     .L24
        test    edi, 2
        jne     .L25
.L7:
        mov     ecx, eax
        xor     edx, edx
        shr     ecx, 2
        test    al, 2
        rep movsd
        je      .L8
        movzx   edx, WORD PTR [esi]
        mov     WORD PTR [edi], dx
        mov     edx, 2
.L8:
        test    al, 1
        je      .L5
        movzx   eax, BYTE PTR [esi+edx]
        mov     BYTE PTR [edi+edx], al
.L5:
        pop     esi
        pop     edi
        ret
.L24:
        movzx   eax, BYTE PTR [esi]
        lea     edi, [edx+11]
        add     esi, 1
        test    edi, 2
        mov     BYTE PTR [edx+10], al
        mov     eax, 122
        je      .L7
.L25:
        movzx   edx, WORD PTR [esi]
        add     edi, 2
        add     esi, 2
        sub     eax, 2
        mov     WORD PTR [edi-2], dx
        jmp     .L7
.LFE3:
```

通用内存复制函数通常的工作原理如下：首先计算块有多少个字（32 位），然后用 MOVSD 指令复制这些内存块，然后逐一复制剩余的字节。

更为复杂的内存复制函数则会利用 SIMD 指令集进行复制，这种复制还会涉及内存地址对齐的问题。有兴趣的读者可以参阅本书第 25 章的第 2 节。

43.1.6　内存对比函数 memcmp()

指令清单 43.22　memcmp()函数的例子

```
void memcpy_1235(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 1235);
};
```

无论内存块的尺寸是多大，MSVC 2010 都会插入相同的通用比较函数。

指令清单 43.23 MSVC 2010 优化程序

```
_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcmp_1235 PROC
        mov     edx, DWORD PTR _buf2$[esp-4]
        mov     ecx, DWORD PTR _buf1$[esp-4]
        push    esi
        push    edi
        mov     esi, 1235
        add     edx, 10
$LL4@memcmp_123:
        mov     eax, DWORD PTR [edx]
        cmp     eax, DWORD PTR [ecx]
        jne     SHORT $LN10@memcmp_123
        sub     esi, 4
        add     ecx, 4
        add     edx, 4
        cmp     esi, 4
        jae     SHORT $LL4@memcmp_123
$LN10@memcmp_123:
        movzx   edi, BYTE PTR [ecx]
        movzx   eax, BYTE PTR [edx]
        sub     eax, edi
        jne     SHORT $LN7@memcmp_123
        movzx   eax, BYTE PTR [edx+1]
        movzx   edi, BYTE PTR [ecx+1]
        sub     eax, edi
        jne     SHORT $LN7@memcmp_123
        movzx   eax, BYTE PTR [edx+2]
        movzx   edi, BYTE PTR [ecx+2]
        sub     eax, edi
        jne     SHORT $LN7@memcmp_123
        cmp     esi, 3
        jbe     SHORT $LN6@memcmp_123
        movzx   eax, BYTE PTR [edx+3]
        movzx   ecx, BYTE PTR [ecx+3]
        sub     eax, ecx
$LN7@memcmp_123:
        sar     eax, 31
        pop     edi
        or      eax, 1
        pop     esi
        ret     0
$LN6@memcmp_123:
        pop     edi
        xor     eax, eax
        pop     esi
        ret     0
_memcmp_1235 ENDP
```

43.1.7 IDA 脚本

笔者编写了一个检索、收缩（folding）常见内联函数的 IDA 脚本。有兴趣的读者请访问：
https://github.com/yurichev/IDA_scripts

第 44 章　C99 标准的受限指针

在某些情况下，用 FORTRAN 系统编译出来的程序会比用 C/C++系统编译出来的程序运行得更快。例如，下面这个例子就是如此：

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

这个程序的功能十分简单，但是里面的指针问题却发人深思：同一块内存可以由多个指针来访问，因此同一个地址的数据可能会被多个指针轮番复写。至少现行标准并不禁止这种情况。

C 语言编译器完全允许上述情况。因此，它分四个阶段处理每次迭代的各类数组：

- 制备 sum[i];
- 制备 product[i];
- 制备 update_me[i];
- 制备 sum_product[i]。在这个阶段，计算机将从内存里重新加载 sum[i]和 product[i]。

第四个阶段是否存在进一步优化的空间呢？既然前面已经计算好了 sum[i]和 product[i]，那么后面我们应该就不必再从内存中读取它们的值了。

答案是肯定的。

只是编译器本身并不能在第三个阶段确定前两个阶段的赋值没有被其他指令覆盖。换而言之，因为编译器不能判断该程序里是否存在指向相同内存区域的指针——即“指针别名（pointer aliasing）”，所以编译器不能确保该指针指向的内存没被改写。

C99 标准中的受限指针［ISO07，6.7.3 节］（部分文献又称“严格别名”）的应运而生。编程人员可通过受限指针的 strict 修饰符向编译器承诺：被该关键字标记的指针是操作相关内存区域的唯一指针，没有其他指针重复指向这个指针所操作的内存区域。

用更为确切、更为正式的语言来说，关键字“restrict”表示该指针是访问既定对象的唯一指针，其他指针都不会重复操作既定对象。从另一个角度来看，一旦某个指针被标记为受限指针，那么编译器就认定既定对象只会被指定的受限指针操作。

下面我们将为每个指针都增加上 restrict 修饰符：

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* ↙
    ↳ restrict sum_product,
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

我们看到的结果如下所示。

指令清单 44.1 x64 下的 GCC 函数 f1()

```
f1:
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6:
        mov     r11, rdi
        mov     rdi, rax
.L4:
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d          ; store to sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d          ; store to product[]
        mov     DWORD PTR [r12+r11*4], ebx     ; store to update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]          ; reload sum[i]
        add     r10d, DWORD PTR [rax]          ; reload product[i]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d   ; store to sum_product[]
        jne     .L6
.L1:
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

指令清单 44.2 x64 下的 GCC 函数 f2()

```
f2:
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
.L11:
        mov     rax, rdi
        mov     rdi, r11
.L10:
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d ; store to update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
```

```
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx ; store to sum[]
        mov     DWORD PTR [r9+rax*4], r11d ; store to product[]
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; store to sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7:
        pop     rbx rsi rdi rbp r12 r13
        ret
```

f1()函数和 f2()函数的不同之处在于：在 f1()函数中，sum[i]和 product[i]数组在循环中会再次加载；而函数 f2()则没有这种重新加载内存数值的操作。在改动后的程序里，因为我们向编译器“承诺”sum[i]和 product[i]的值不会被其他指针复写，所以计算机会重复利用前几个阶段制备好的各项数据，不再从内存加载它们的值了。很明显，改进后的程序运行速度更快一些。

如果我们声明了某个指针是受限指针，而实际的程序又有其他指针操作这个受限指针操作的内存区域，将会发生什么情况？这真的就是程序员的事了，不过程序运行的结果肯定是错误的。

FORTRAN 语言的编译器把所有指针都视为受限指针。因此，在 C 语言不支持 C99 标准的 restrict 修饰符而实际指针属于受限指针的时候，用 FORTRAN 语言编译出来的应用程序会比用 C 语言编译出来的程序运行得更快。

受限指针主要用于哪些领域？它主要用于操作多个大尺寸内存块的应用方面。例如，在超级计算机/HPC 平台上经常进行的线性方程组求解就属于这种类型的应用。或许，这正是这种平台普遍采用 FORTRAN 语言的原因之一吧。

另一方面，在循环语句的迭代次数不是非常高的情况下，受限指针带来的性能提升就不会十分明显。

第 45 章　打造无分支的 abs()函数

请回顾前文第 12 章第 2 节的那个函数。请设想一下，能否用 x86 的汇编指令打造一个无分支的版本？

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

答案当然是肯定的。

45.1　x64 下的 GCC 4.9.1 优化

首先我们来看看，如果我们采用 GCC 4.9 来编译的话，会出现什么情况。

指令清单 45.1　x64 下的 GCC 4.9 优化

```
my_abs:
        mov     edx, edi
        mov     eax, edi
        sar     edx, 31
; EDX is 0xFFFFFFFF here if sign of input value is minus
; EDX is 0 if sign of input value is plus (including 0)
; the following two instructions have effect only if EDX is 0xFFFFFFFF
; or idle if EDX is 0
        xor     eax, edx
        sub     eax, edx
        ret
```

以下是其实现过程的描述：

SAR 算术右[①]移：算术右移 31 位，我们知道算术右移是带符号的操作。有符号数的最高有效位 MSB 就是其符号位。也就是说，如果 MSB 的值为 1（原操作数是负数），那么右移 31 位后，不管原来的数是什么，得到的结果都会是 0xffffffff。当然如果 MSB 的值为 0（原操作数是正数或零），那么最后的结果就会是零。执行完 SAR 后，返回值被存放在 EDX 寄存器中。后面就是 XOR 指令。当原操作数是负数时，这条指令就变成了“XOR｛寄存器｝,0xffffffff”，相当于逐位求非的逻辑非运算；其后的 SUB 指令会完成补码运算的“减去负一（加一）”运算、求得绝对值，因为此时 EDX 寄存器的值是 0xffffffff——十进制的“－1”。有关两个数的逻辑运算和递增运算详解，请参阅本书第 30 章。

因此，当源操作数是负数的时候，最后两条运算指令会对源操作数进行处理。在符号位的值为零，即零和正数的情况下，最后两条运算指令不会调整源操作数的值。

有关本例的详细算法，请查阅参考书目［War02，pp.2-4］的相关介绍。GCC 或许借鉴了现有的成熟算法，或许这正是它自己推演的计算结果。

45.2　ARM64 下的 GCC 4.9 优化

GCC 4.9 for ARM64 的编译方式和 GCC for x64 的编译方式几乎相同，只是它使用的是自己 CPU 平台

① 原文为左移。

的 64 位寄存器而已。由于 ARM64 可以通过参数调节符 ASR 在别的指令里完成位移运算，因此 GCC 4.9 for ARM64 比 GCC for x64 分配的指令少。

指令清单 45.2　ARM64 下的 GCC 4.9 优化

```
my_abs:
; sign-extend input 32-bit value to X0 64-bit register:
        sxtw    x0, w0
        eor     x1, x0, x0, asr 63
; X1=X0^(X0>>63) (shift is arithmetical)
        sub     x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (all shifts are arithmetical)
        ret
```

第 46 章　变长参数函数

像 printf()和 scanf()一类的函数可以处理不同数量的输入参数。这种函数又是如何访问参数的呢？

46.1　计算算术平均值

如果要编写一个计算算术平均值的函数，那么就需要在函数的参数声明部分指定所有的外来参数。但是 C/C++的变长参数函数却无法事先知道外来参数的数量。为了方便起见，我们用“-1”作为最后一个参数兼其他参数的终止符。

C 语言标准函数库的头文件 stdarg.h 定义了变长参数的处理方法（宏）。刚才提到的 printf()函数和 scanf()函数都使用了这个文件提供的宏。

```
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
        va_list args;
        int sum=v, count=1, i;
        va_start(args, v);

        while(1)
        {
                i=va_arg(args, int);
                if (i==-1) // terminator
                        break;
                sum=sum+i;
                count++;
        }

        va_end(args);
        return sum/count;
};

int main()
{
        printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
};
```

变长参数函数按照常规函数参数的方法访问外部传来的第一个参数。而后，程序借助 stdarg.h 提供的宏 var_arg 调用其余参数，依次求得各参数之和，最终计算其平均值。

46.1.1　cdecl 调用规范

指令清单 46.1　MSVC 6.0 优化

```
_v$ = 8
_arith_mean PROC NEAR
        mov     eax, DWORD PTR _v$[esp-4] ; load 1st argument into sum
        push    esi
        mov     esi, 1                    ; count=1
        lea     edx, DWORD PTR _v$[esp]   ; address of the 1st argument
$L838:
```

```
        mov     ecx, DWORD PTR [edx+4]    ; load next argument
        add     edx, 4                    ; shift pointer to the next argument
        cmp     ecx, -1                   ; is it -1?
        je      SHORT $L856               ; exit if so
        add     eax, ecx                  ; sum = sum + loaded argument
        inc     esi                       ; count++
        jmp     SHORT $L838
$L856:
; calculate quotient

        cdq
        idiv    esi
        pop     esi
        ret     0
_arith_mean ENDP

$SG851  DB      '%d', 0aH, 00H

_main   PROC NEAR
        push    -1
        push    15
        push    10
        push    7
        push    2
        push    1
        call    _arith_mean
        push    eax
        push    OFFSET FLAT:$SG851 ; '%d'
        call    _printf
        add     esp, 32
        ret     0
_main   ENDP
```

在 main()函数里，各项参数从右向左依次逆序传递入栈。第一个入栈的是最后一项参数“-1”，而最后入栈的是第一项参数——格式化字符串。

函数 arith_mean()取出第一个参数的值，并将其保存在变量 sum 中。接着，将第二个参数的地址保存在寄存器 EDX 中，并取出其值，与前面的 sum 相加。如此循环往复，直到参数的终止符-1。

当找到了参数串的结尾后，程序再将所有数的算术和 sum 除以参数的个数（当然不包括终止符-1）。按照这种算法计算出来的商就是各参数的算术平均值。

换句话说，在调用变长参数函数时，调用方函数先把不确定长度的参数堆积为数组，再通过栈把这个数组传递给变长函数参数。这就解释了为什么 cdecl 调用规范会要求将第一个参数最后一个推送入栈了。因为如果不这样的话，被调用方函数会找不到第一个参数，这会导致 printf()这样的函数因为找不到格式化字符串的地址而无法运行。

46.1.2　基于寄存器的调用规范

细心的读者也可能会问，那些优先利用寄存器传递参数的调用规范是什么情况？下面我们就来看看。

指令清单 46.2　x64 下的 MSVC 2012 优化

```
$SG3013 DB      '%d', 0aH, 00H

v$ = 8
arith_mean PROC
        mov     DWORD PTR [rsp+8], ecx    ; 1st argument
        mov     QWORD PTR [rsp+16], rdx   ; 2nd argument
        mov     QWORD PTR [rsp+24], r8    ; 3rd argument
        mov     eax, ecx                  ; sum = 1st argument
        lea     rcx, QWORD PTR v$[rsp+8]  ; pointer to the 2nd argument
        mov     QWORD PTR [rsp+32], r9    ; 4th argument
        mov     edx, DWORD PTR [rcx]      ; load 2nd argument
```

```
        mov     r8d, 1                  ; count=1
        cmp     edx, -1                 ; 2nd argument is -1?
        je      SHORT $LN8@arith_mean   ; exit if so
$LL3@arith_mean:
        add     eax, edx                ; sum = sum + loaded argument
        mov     edx, DWORD PTR [rcx+8]  ; load next argument
        lea     rcx, QWORD PTR [rcx+8]  ; shift pointer to point to the argument after next
        inc     r8d                     ; count++
        cmp     edx, -1                 ; is loaded argument -1?
        jne     SHORT $LL3@arith_mean   ; go to loop begin if its not'
$LN8@arith_mean:
; calculate quotient
        cdq
        idiv    r8d
        ret     0
arith_mean ENDP

main    PROC
        sub     rsp, 56
        mov     edx, 2
        mov     DWORD PTR [rsp+40], -1
        mov     DWORD PTR [rsp+32], 15
        lea     r9d, QWORD PTR [rdx+8]
        lea     r8d, QWORD PTR [rdx+5]
        lea     ecx, QWORD PTR [rdx-1]
        call    arith_mean
        lea     rcx, OFFSET FLAT:$SG3013
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
```

在这个程序里，寄存器负责传递函数的前 4 个参数，栈用来传递其余的 2 个参数。函数 arith_mean() 首先将寄存器传递的 4 个参数存放在阴影空间里，把阴影空间和传递参数的栈合并成了统一而连续的参数数组！

GCC 会如何处理参数呢？与 MSVC 相比，GCC 在编译的时候略显画蛇添足。它会把函数分为两部分：第一部分的指令将寄存器的值保存在“红色地带”，并在那里进行处理；而第二部分指令再处理栈的数据。

指令清单 46.3 x64 下的 GCC 4.9.1 的优化

```
arith_mean:
        lea     rax, [rsp+8]
        ; save 6 input registers in "red zone" in the local stack
        mov     QWORD PTR [rsp-40], rsi
        mov     QWORD PTR [rsp-32], rdx
        mov     QWORD PTR [rsp-16], r8
        mov     QWORD PTR [rsp-24], rcx
        mov     esi, 8
        mov     QWORD PTR [rsp-64], rax
        lea     rax, [rsp-48]
        mov     QWORD PTR [rsp-8], r9
        mov     DWORD PTR [rsp-72], 8
        lea     rdx, [rsp+8]
        mov     r8d, 1
        mov     QWORD PTR [rsp-56], rax
        jmp     .L5
.L7:
        ; work out saved arguments
        lea     rax, [rsp-48]
        mov     ecx, esi
        add     esi, 8
        add     rcx, rax
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        je      .L4
```

```
.L8:
        add     edi, ecx
        add     r8d, 1
.L5:
        ; decide, which part we will work out now.
        ; is current argument number less or equal 6?
        cmp     esi, 47
        jbe     .L7             ; no, process saved arguments then
        ; work out arguments from stack
        mov     rcx, rdx
        add     rdx, 8
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        jne     .L8
.L4:
        mov     eax, edi
        cdq
        idiv    r8d
        ret
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        mov     edx, 7
        mov     esi, 2
        mov     edi, 1
        mov     r9d, -1
        mov     r8d, 15
        mov     ecx, 10
        xor     eax, eax
        call    arith_mean
        mov     esi, OFFSET FLAT:.LC1
        mov     edx, eax
        mov     edi, 1
        xor     eax, eax
        add     rsp, 8
        jmp     __printf_chk
```

另外，本书第 64 章第 8 节介绍了阴影空间的另外一个案例。

46.2　vprintf()函数例子

在编写日志(logging)函数的时候，多数人都会自己构造一种与 printf 类似的、处理“格式化字符串＋一系列（但是数量可变）的内容参数”的变长参数函数。

另外一种常见的变长参数函数就是下文的这种 die()函数。这是一种在显示提示信息之后随即退出整个程序的异常处理函数。它需要把不确定数量的输入参数打包、封装并传递给 printf()函数。如何实现呢？这些函数名称前面有一个字母 v 的，这是因为它应当能够处理不确定数量（variable.可变的）的参数。以 die()函数调用的 vprintf()函数为例，它的输入变量就可分为两部分：一部分是格式化字符串，另一部分是带有多种类型数据变量列表 va_list 的指针。

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};
```

仔细观察就会发现，va_list 是一个指向数组的指针。在编译后，这个特征非常明显：

指令清单 46.4 MSVC 2010 下的程序优化

```
_fmt$ = 8
_die    PROC
        ; load 1st argument (format-string)
        mov     ecx, DWORD PTR _fmt$[esp-4]
        ; get pointer to the 2nd argument
        lea     eax, DWORD PTR _fmt$[esp]
        push    eax             ; pass pointer
        push    ecx
        call    _vprintf
        add     esp, 8
        push    0
        call    _exit
$LN3@die:
        int     3
_die    ENDP
```

由此可知，die()函数实现的功能就是：取一个指向参数的指针，再将其传送给 vprintf()函数。变长参数（序列）像数组那样被来回传递。

指令清单 46.5 x64 下的 MSVC 2012 优化

```
fmt$ = 48
die     PROC
        ; save first 4 arguments in Shadow Space
        mov     QWORD PTR [rsp+8], rcx
        mov     QWORD PTR [rsp+16], rdx
        mov     QWORD PTR [rsp+24], r8
        mov     QWORD PTR [rsp+32], r9
        sub     rsp, 40
        lea     rdx, QWORD PTR fmt$[rsp+8] ; pass pointer to the 1st argument
        ; RCX here is still points to the 1st argument (format-string) of die()
        ; so vprintf() will take it right from RCX
        call    vprintf
        xor     ecx, ecx
        call    exit
        int     3
die     ENDP
```

逆向工程权威指南 上册

逆向工程是一种分析目标系统的过程。

本书专注于软件逆向工程，即研究编译后的可执行程序。本书是写给初学者的一本权威指南。全书共分为12个部分，共102章，涉及软件逆向工程相关的众多技术话题，堪称是逆向工程技术百科全书。全书讲解详细，附带丰富的代码示例，还给出了很多习题来帮助读者巩固所学的知识，附录部分给出了习题的解答。

本书适合对逆向工程技术、操作系统底层技术、程序分析技术感兴趣的读者阅读，也适合专业的程序开发人员参考。

“... 谨向这本出色的教程致以个人的敬意!”

—— Herbert Bos，阿姆斯特丹自由大学教授
《Modern Operating Systems (4th Edition)》作者

“... 引人入胜，值得一读!”

—— Michael Sikorski
《Practical Malware Analysis》的作者

作者简介

Dennis Yurichev，乌克兰程序员，安全技术专家。读者可以通过https://yurichev.com/联系他，并获取和本书相关的更多学习资料。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／软件开发／安全
人民邮电出版社网址：www.ptpress.com.cn