

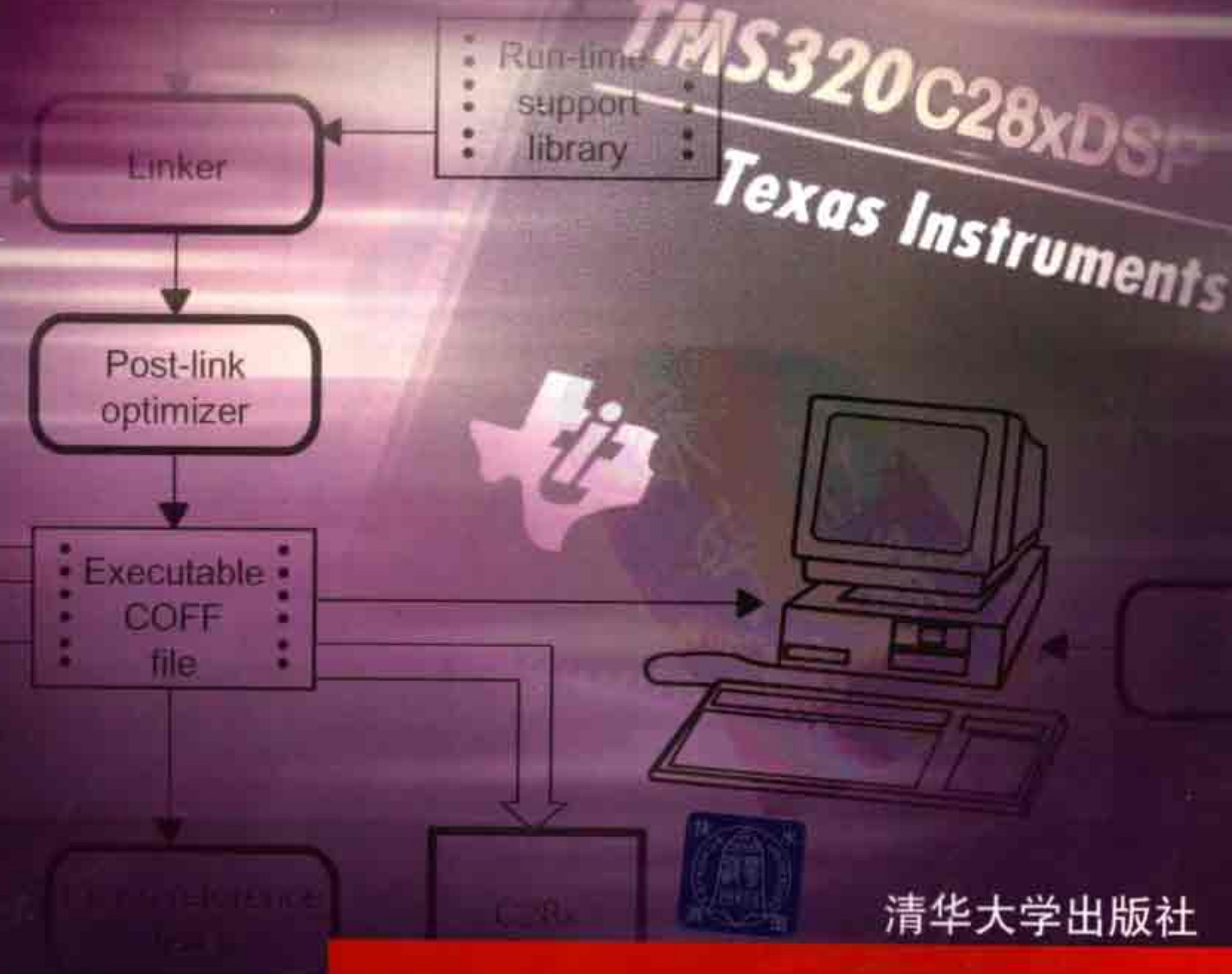
TI DSP 系列中文手册

TMS320C28x 系列 DSP

指令和编程指南

[美] Texas Instruments Incorporated 著

刘和平 张卫宁 刘 林 编译
周有为 邓 力 江 渝



清华大学出版社

TMS320C28x 系列 DSP

指令和编程指南

[美] Texas Instruments Incorporated

著

刘和平 张卫宁 刘 林

编译

周有为 邓 力 江 渝

清华大学出版社

北 京



内 容 简 介

本书由 TI 公司的两个文献编译而成, 编号为 SPRU513 的文献介绍了如何使用汇编语言工具: 汇编器、归档器、目标代码链接器、交叉引用列表程序、绝对地址列表程序、十六进制转换应用程序。编号为 SPRU430B 的文献中的一部分介绍了 C28x 汇编语言指令集。由于这两部分内容紧密相关, 故将其放在一起, 以便读者查阅。

本书主要针对从事 TI 公司 2000 系列 DSP 开发应用的工程技术人员, 也可以作为在校研究生的参考用书。

版权所有, 翻印必究。举报电话: 010-62782989 13501256678 13801310933

本书封面贴有清华大学出版社防伪标签, 无标签者不得销售。

本书防伪标签采用清华大学核研院专有核径迹膜防伪技术, 用户可通过在图案表面涂抹清水, 图案消失, 水干后图案复现; 或将表面膜揭下, 放在白纸上用彩笔涂抹, 图案在白纸上再现的方法识别真伪。

图书在版编目 (CIP) 数据

TMS 320 C28X 系列 DSP 指令和编程指南/TI 公司著; 刘和平等编译. —北京: 清华大学出版社, 2005. 3
(TI DSP 系列中文手册)

ISBN 7-302-10438-7

I. T… II. ①T… ②刘… III. 数字信号—信号处理—数字通信系统, TMS 320 C28X DSP—程序设计—指南 IV. TN911. 72-62

中国版本图书馆 CIP 数据核字 (2005) 第 009219 号

出 版 者: 清华大学出版社

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

社 总 机: 010-62770175

责任编辑: 曾 刚

封面设计: 刘春敏

版式设计: 郑轶文

印 刷 者: 北京密云胶印厂

装 订 者: 北京市密云县京文制本装订厂

发 行 者: 新华书店总店北京发行所

开 本: 185×260 印张: 31.5 字数: 693 千字

版 次: 2005 年 3 月第 1 版 2005 年 3 月第 1 次印刷

书 号: ISBN 7-302-10438-7/TP·7090

印 数: 1~5000

定 价: 46.00 元

地 址: 北京清华大学学研大厦

邮 编: 100084

客户服务: 010-62776969

TI DSP 系列中文手册编译委员会

(按汉语拼音排序)

主任委员:

胡广书	教授	清华大学
彭启琮	教授	电子科技大学
沈洁	经理	TI 中国大学计划

委员:

陈健	教授	上海交通大学
戴逸民	教授	中国科学技术大学
何佩琨	教授	北京理工大学
刘和平	教授	重庆大学
潘亚涛	工程师	TI 中国大学计划
桑恩方	教授	哈尔滨工程大学
王军宁	副教授	西安电子科技大学
张旭东	副教授	清华大学
曾刚	编辑	清华大学出版社



序

经过全体编译老师和编译委员会近一年的努力,“TI DSP 系列中文手册”终于陆续和广大读者见面了。

数字信号处理器(Digital Signal Processing, DSP)是对信号和图像实现实时处理的一类高性能的 CPU。所谓“实时(Real-Time)实现”,是指一个实际的系统能在人们听觉、视觉或按任务要求所允许的时间范围内实现对输入信号的处理并将其输出。目前, DSP 已广泛应用于通信、家电、航空航天、工业测量、控制、生物医学工程及军事等许许多多需要实时实现的领域。

美国德州仪器(Texas Instruments, TI)公司是全球 DSP 研发和生产的领先者。自 1982 年推出第一块 DSP 芯片以来,到 20 世纪 90 年代中期, TI 先后推出了 C10、C20、C30、C40、C50 及 C80 等 6 代 TMS320 系列的 DSP 产品。紧接着又推出了 C2000 系列、C5000 系列和 C6000 系列三大主流产品,并推出了将 DSP 和 ARM 合为一体的 OMAP 系列。这些产品无论是在国外还是在国内都得到了广泛的应用。例如,“TI 中国大学计划”在 2003 年举办的“TI DSP 设计比赛”中,国内高校就有约 90 个队参加,足见 DSP 在我国已经得到普遍的重视。

凡是从事过含有 CPU 的系统设计(单片机或 DSP)的设计人员都知道,为了顺利地实现设计任务,一本或几本好的手册是必不可少的,其中包括该 CPU 的结构手册、指令和汇编语言手册以及开发手册等。

由于 TI 的 DSP 发展迅速,产品更新快,因此其手册自然也非常多。由于手册需要更新和补充,因此,彼此之间难免会出现重复和种类繁多的现象。使用过 TI DSP 文档的用户都感觉到,其手册在使用上是有相当难度的。另外, TI DSP 文档都是用英文写成,这也给部分工程技术人员带来一定的困难。

鉴于此, TI 中国主管提出委托国内的高校老师对其文档进行编译,并授权清华大学出版社正式出版。在“TI 中国大学计划”的建议下,2003 年 6 月通过推荐和报名方式成立了编译委员会。

通过认真讨论,编译委员会首先确定了文档编译的原则,然后确定了编译的书目,最后确定了每一本书的编译者。

关于编译的原则,我们提出了如下两点:

(1) 本文档的定位为“手册”。也就是说,每一位文档的编译者应全面了解和掌握所编译书目的所有英文文档,并了解各个文档之间的关系,在保证文档完整的基础上,选择最新的文档,并去除其中的重复内容和已经淘汰的内容。

(2) 要尽可能地按照 TI 英文文档的“本意”来形成中文,以保证手册的准确性。允

许作者按自己的经验有所发挥，以便于难点的理解。

这次编译的书目包含三大部分：一是各个系列的共用部分，如 CCS、DSP/BIOS、算法标准、C 语言编译器及开发工具等各个手册；二是按 C2000、C5000 和 C6000 三大系列分别编译它们的 CPU 结构及指令手册；三是分别编译它们的应用。

编译计划在“TI 中国大学计划”的相关会议上提出后，得到了国内高校许多老师的热情支持，很快便将要编译的书目一一落实。这些老师都有着从事 DSP 教学和科研的丰富经验，正是由于他们的大力支持，才使这一庞大的工作计划能够付诸实施。在此，谨向参加本系列手册编译工作的全体老师表示衷心的感谢！

“TI 中国大学计划”在本系列手册的编译过程中给予了多方面的大力支持，在此向他们表示衷心的感谢！

由于本系列手册的编译工作量大、时间紧，因此，尽管编译的老师 and 编译委员会都尽了最大的努力，但也难免有不妥，甚至错误之处，编译委员会全体老师恳切地希望广大读者给以批判指正。

清华大学生物医学工程系

胡广书 教授

2004 年 3 月



前 言

TMS320C28x 芯片作为 TI 公司 DSP 系列的新成员,是 TMS320C2000TM 平台下的一款最新推出的 DSP 芯片。C28x 芯片为功能强大的 TMS320TM DSP 结构设计提供了低成本、低功耗和高性能的处理能力,特别适合于电机的数字式运动控制。C28x 具备现有 C2000 DSP 控制器芯片的所有功能且得到了很大的加强,芯片的处理性能更快(150MIPS)、外设集成度更高、程序存储器更大(128 千字)、A/D 转换速度更快、精度更高(12 位)、开发软件集成度更高、开发效率更高、支持 C/C++ 语言等,是电机数字化控制的升级产品。在电机控制、工业控制和电力系统继电保护等方面将会得到大量的应用。

为了满足广大用户的需求,我们将 TI 公司的文献进行选择归类,将相关的部分放在一起,以便读者使用。本译文将两部分文档集成在一起,一部分为设计汇编指令和汇编工具的使用资料,其中使用了编号为 SPRU513 文献,这一部分将涉及如何使用汇编语言工具:汇编器、归档器(档案库存储器)(大容量外存储器)、目标代码链接器、交叉引用列表程序、绝对地址列表器、十六进制转换应用程序;另一部分编号为 SPRU430B 文献将涉及 C28x 汇编语言指令集。

参加本书成书工作的还有尉冰娟、何懋渝、崔晶、吴俊、冼成瑜、杨立勇、张伟、杨利辉、刘钊、李纯、王军生、高苏州等,他们参加了部分资料的翻译、校对、录入等大量工作,在此表示十分的感谢。

本书的成书过程中还得到了重庆大学-美国德州仪器数字信号处理方案实验室、重庆大学电气工程学院电力电子与电力传动系郑群英等许多老师的大力帮助和支持,在此表示衷心的感谢。

在这里还要感谢美国德州仪器公司大学计划所给予的大力支持。

限于编译者的水平,书中难免存在错误和翻译不妥之处,恳请读者批评指正。

作 者

(2004 年 8 月于重庆大学)

资源分享

目 录

第 1 章	软件开发工具.....	1
1.1	软件开发工具概况.....	2
1.2	软件开发工具介绍.....	2
第 2 章	通用目标文件格式介绍.....	4
2.1	段.....	4
2.2	汇编器如何处理段.....	5
2.2.1	未初始化段.....	6
2.2.2	已初始化段.....	6
2.2.3	已命名段.....	7
2.2.4	子段.....	8
2.2.5	段程序计数器.....	8
2.2.6	使用段伪指令实例.....	8
2.3	链接器如何处理段.....	10
2.3.1	默认内存分配.....	11
2.3.2	在存储器映像中存放段.....	12
2.4	重定位.....	12
	运行中的重定位.....	13
2.5	装载程序.....	14
2.6	COFF 文件中的符号.....	14
2.6.1	外部符号.....	14
2.6.2	符号表.....	15
第 3 章	汇编器.....	16
3.1	汇编器功能.....	16
3.2	在软件开发过程中汇编器的作用.....	17
3.3	运行汇编器.....	17
3.4	为汇编器输入的替换目录命名.....	19
3.4.1	使用-i 汇编器选项.....	20
3.4.2	使用 C2000_A_DIR 或 A_DIR 环境变量.....	20
3.5	源程序语句格式.....	20
3.5.1	标号域.....	21
3.5.2	助记符域.....	22
3.5.3	操作数域.....	22
3.5.4	注释域.....	22

3.6	常量.....	22
3.6.1	二进制整数.....	23
3.6.2	八进制整数.....	23
3.6.3	十进制整数.....	23
3.6.4	十六进制整数.....	24
3.6.5	字符常量.....	24
3.6.6	汇编编译过程使用的 (Assembly-Time) 常量.....	24
3.6.7	浮点型常量.....	25
3.7	字符串.....	25
3.8	符号.....	25
3.8.1	标号.....	26
3.8.2	局部标号.....	26
3.8.3	符号常量.....	28
3.8.4	定义符号常量 (-d 选项).....	28
3.8.5	预定义符号常量.....	29
3.8.6	置换符号.....	30
3.9	表达式.....	31
3.9.1	运算符.....	31
3.9.2	表达式的上溢和下溢.....	32
3.9.3	定义明确的表达式.....	32
3.9.4	条件表达式.....	32
3.9.5	合法的表达式.....	33
3.10	内嵌函数.....	34
3.11	源程序列表.....	35
3.12	交叉引用列表.....	36
3.13	灵巧的编码.....	37
3.14	汇编变量的 C 类型符号调试.....	38
3.15	TMS320C28x 汇编器的模式.....	40
3.15.1	C27x 目标模式.....	40
3.15.2	C28x 目标模式.....	41
3.15.3	C28x 目标——兼容 C27x 的语法模式.....	41
3.15.4	C28x 目标——兼容 C2xlp 的语法模式.....	41
第 4 章	汇编伪指令.....	44
4.1	伪指令简介.....	44
4.2	与 TMS320C1x/C2x/C2xx/C5x 汇编伪指令的兼容性.....	47
4.3	定义段的伪指令.....	48
4.4	常数初始化伪指令.....	50
4.5	调准段程序计数器伪指令.....	52

4.6	输出列表格式伪指令	53
4.7	引用其他文件的伪指令	54
4.8	条件汇编伪指令	54
4.9	汇编过程使用的符号的伪指令	55
4.10	汇编器模式伪指令	56
4.11	其他伪指令	56
4.12	伪指令索引表	57
第 5 章	宏语言	99
5.1	宏的使用	99
5.2	定义宏	100
5.3	宏参数/置换符号	101
5.3.1	定义置换符号的伪指令	102
5.3.2	内置置换符号函数	103
5.3.3	递归的置换符号	104
5.3.4	强制置换	104
5.3.5	访问下标置换符号的单个字符	105
5.3.6	在宏中作为局部变量的置换符号	106
5.4	宏库	107
5.5	在宏中使用条件汇编	107
5.6	在宏中使用标号	109
5.7	在宏中产生信息	110
5.8	用伪指令格式化输出列表	111
5.9	递归和嵌套宏的使用	111
5.10	宏伪指令汇总	113
第 6 章	归档器	115
6.1	归档器概述	115
6.2	软件开发流程中归档器的作用	116
6.3	调用归档器	116
6.4	归档器实例	117
第 7 章	链接器	120
7.1	链接器概述	120
7.2	在软件开发流程中链接器的作用	121
7.3	调用链接器	121
7.4	链接器选项	123
7.4.1	重定位 (-a 和 -r 选项)	124
7.4.2	禁止合并符号调试信息	125
7.4.3	C 语言程序选项 (-c 和 -cr 选项)	125

7.4.4	定义入口 (-e 选项)	125
7.4.5	设置默认的填充值 (-f fill_value 选项)	126
7.4.6	全局化符号 (-g symbol 选项)	126
7.4.7	静态化所有的全局符号 (-h 选项)	126
7.4.8	定义堆的大小 (-heap size 选项)	126
7.4.9	改变库的搜索路径 (-l 选项、-i 选项和 C_DIR 环境变量)	127
7.4.10	取消条件链接	128
7.4.11	忽略分配 (-k 选项)	129
7.4.12	创建映像文件 (-m filename 选项)	129
7.4.13	命名输出文件 (-o filename 选项)	130
7.4.14	隐藏运行信息 (-q 选项)	130
7.4.15	删除符号信息 (-s 选项)	130
7.4.16	定义堆栈的大小 (-stack size 选项)	130
7.4.17	引入未定义的符号 (-u symbol 选项)	131
7.4.18	当创建未定义的段时显示信息 (-w 选项)	131
7.4.19	穷举读库 (-x 选项)	131
7.5	链接器命令文件	132
7.5.1	链接器命令文件中的保留名	133
7.5.2	链接器命令文件中的常量	134
7.6	目标库	134
7.7	MEMORY 伪指令	135
7.7.1	默认的存储器模式	135
7.7.2	MEMORY 伪指令格式	135
7.8	SECTIONS 伪指令	138
7.8.1	SECTIONS 伪指令的格式	138
7.8.2	地址分配	140
7.8.3	规定输入段	143
7.9	指定一个段的运行地址	145
7.9.1	指定装载和运行地址	145
7.9.2	未初始化的段	146
7.9.3	用 .label 伪指令访问装载地址	146
7.10	UNION 和 GROUP 语句的使用	147
7.10.1	用 UNION 语句重叠段	147
7.10.2	把输出段在一起分组	149
7.11	重叠页	149
7.11.1	用 MEMORY 伪指令定义重叠页	150
7.11.2	重叠页实例	150
7.11.3	在 SECTIONS 伪指令中使用重叠页	150

7.11.4 对重叠页的存储器分配	151
7.12 特殊段类型	152
7.13 默认分配	152
7.13.1 输出段的形成	153
7.13.2 默认分配算法	153
7.14 链接时给符号赋值	154
7.14.1 赋值语句的格式	154
7.14.2 将 SPC 赋值到一个符号	154
7.14.3 赋值表达式	155
7.14.4 链接器定义的符号	156
7.15 创建和填充空位	156
7.15.1 初始化段和未初始化段	156
7.15.2 创建空位	157
7.15.3 填充空位	158
7.15.4 未初始化段的显式初始化	159
7.16 部分(增量)链接	159
7.17 链接 C 代码	161
7.17.1 运行中的初始化	161
7.17.2 目标库和运行时的支持	161
7.17.3 设置堆栈和堆段的大小	162
7.17.4 运行中变量的自动初始化	162
7.17.5 装载时变量的自动初始化	162
7.17.6 -c 和-cr 链接器选项	163
7.18 链接器举例	163
第 8 章 绝对列表程序	167
8.1 生成绝对列表	167
8.2 调用绝对列表程序	168
8.3 绝对列表程序实例	169
第 9 章 交叉引用列表程序	173
9.1 生成交叉引用列表	173
9.2 调用交叉引用列表程序	174
9.3 交叉引用列表程序举例	175
第 10 章 十六进制转换应用程序	177
10.1 十六进制转换应用程序在软件开发 流程中的作用	178
10.2 调用十六进制转换应用程序	178
10.2.1 从命令行调用十六进制转换应用程序	179
10.2.2 用命令文件调用十六进制转换应用程序	180

10.3	存储器宽度	181
10.3.1	目标宽度	181
10.3.2	存储器宽度	181
10.3.3	数据划分到输出文件	182
10.3.4	规定输出字的字次序	184
10.4	ROMS 伪指令	184
10.4.1	何时使用 ROMS 伪指令	186
10.4.2	ROMS 伪指令举例	186
10.5	SECTIONS 伪指令	188
10.6	分配输出文件名	189
10.7	映像模式和填充 (-fill) 选项	190
10.7.1	生成一个存储器映像	190
10.7.2	填充值	191
10.7.3	在映像模式下所遵循的步骤	191
10.8	控制 ROM 器件地址	191
10.9	目标格式	192
10.9.1	ASCII-Hex 目标格式 (-a 选项)	192
10.9.2	Intel MCS-86 目标格式 (-i 选项)	193
10.9.3	Motorola-S 目标格式 (-m 选项)	194
10.9.4	TI-Tagged SDSMAC 目标格式 (-t 选项)	194
10.9.5	扩展 Tektronix 目标格式 (-x 选项)	195
10.10	十六进制转换应用程序错误信息	196
第 11 章	C28x 汇编语言指令集	197
11.1	指令概述 (按功能分类)	197
11.2	寄存器操作	199
附录 A	通用目标文件格式	428
附录 B	符号调试伪指令	444
附录 C	汇编器错误信息	453
附录 D	链接器错误信息	465
附录 E	术语表	477



第 1 章 软件开发工具

TMS320C28xTM 由一系列软件开发工具支持，包括 C/C++ 优化编译器、汇编器、链接器和各种应用程序。本章将简要说明这些工具在软件开发流程中是如何协调工作的，并对各种工具作简要介绍。

TMS320C28x 汇编语言开发工具有：

- ☐ 汇编器
- ☐ 归档器
- ☐ 目标代码链接器
- ☐ 交叉引用列表程序
- ☐ 绝对地址列表程序
- ☐ 十六进制转换应用程序

1.1 软件开发工具概况

图 1.1 所示为 TMS320C28x 软件开发流程。阴影部分强调最常用的开发路径；其他部分是可选的，它们用于增强开发能力的外围功能。

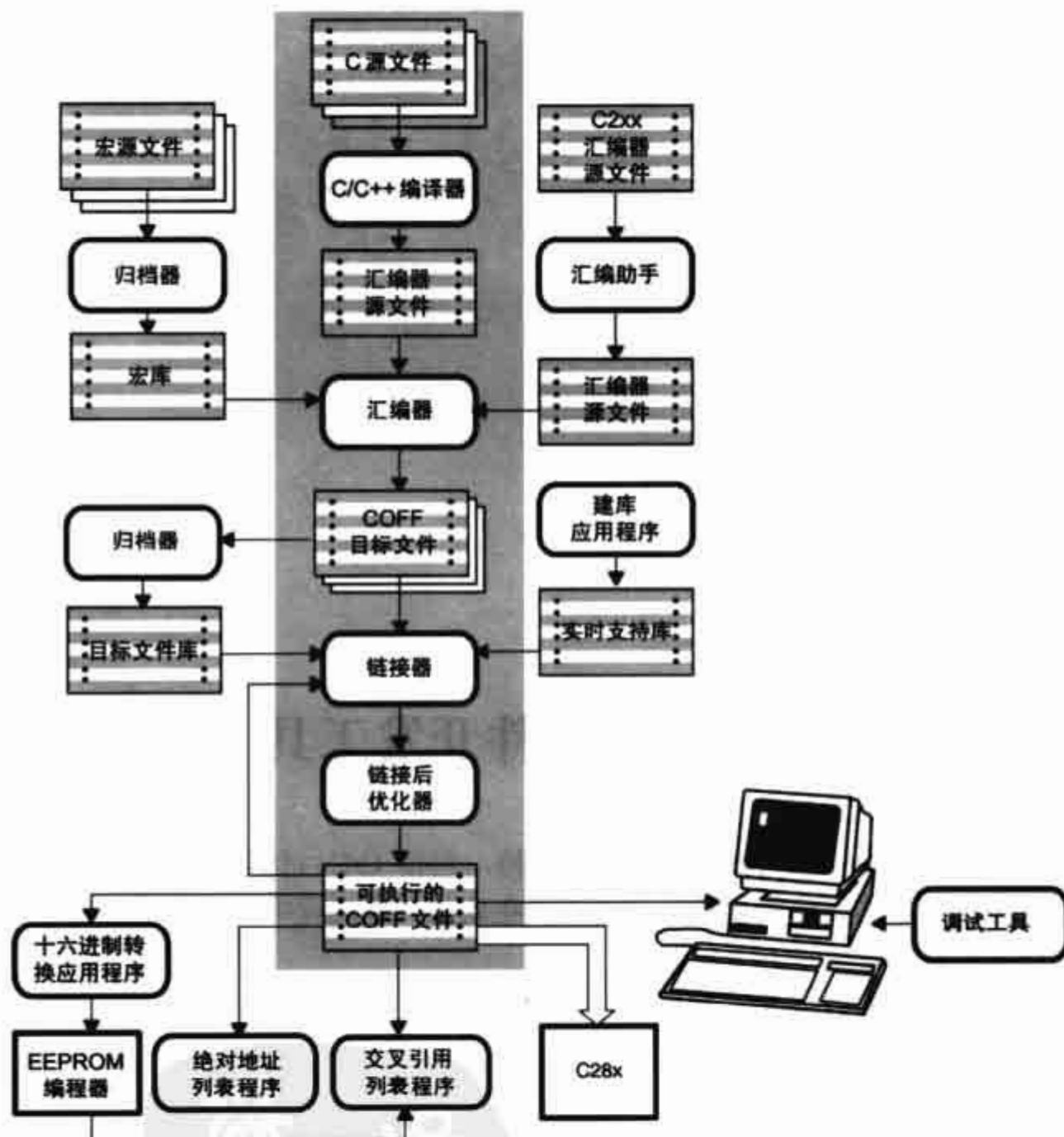


图 1.1 TMS320C28x 软件开发流程图

1.2 软件开发工具介绍

下面将简要介绍图 1.1 中所列出的工具：

□ **C/C++编译器** 接收 C 和 C++源代码并生成 TMS320C28x 汇编语言源代码。编译

程序包中包括集成 (shell) 程序、优化器、交叉引用列表程序:

- 集成程序使用户能够一步完成编译、汇编和链接过程
- 优化器优化代码以提高 C 语言程序的效率
- 交叉引用列表程序使汇编语言输出和 C 源语句进行交互访问, 使其能与经编译器编译后的代码相关联

要想获取更多信息, 请查阅 TMS320C28x C/C++ 优化编译器用户手册。

- **汇编器** 将汇编语言源文件转换成机器语言 COFF 目标文件。源文件中包含指令、汇编伪指令和宏伪指令。用户可以使用汇编伪指令控制汇编过程的各个方面, 如, 源程序列表格式、数据排列以及段内容。可以从第 3 章汇编器介绍到第 5 章宏语言章节中了解到更多信息。若想获得汇编语言指令集的详细内容, 可以参见第 11 章的内容。
- **链接器** 将目标文件组合成一个可执行 COFF 目标模块。在它创建可执行模块的同时, 进行重定位和解决外部引用。链接器接收可重新定位的 COFF 目标文件 (由汇编器创建) 作为输入。它也接收上一次链接器运行所创建的档案库成员和输出模块。链接器伪指令允许用户组合目标文件段, 把段或符号约定在存储器的某些地址范围内, 并定义和重新定义全局符号。要了解更详细的内容, 请参阅第 7 章链接器介绍。
- **归档器** 允许用户把一组文件收集到单个档案文件, 称为库。例如, 用户可以收集几个宏放入宏库中。汇编器搜索库并使用被源文件称作宏的成员。用户也可以收集一组目标文件放入目标库。链接时链接器将确定的外部引用包含到库中。归档器允许用户以删除、替换、提取、增加成员的方式修改库。阅读第 6 章归档器介绍, 可以获得更多信息。
- 用户可以使用 **建库应用程序** 来建立用户自己的实时支持库。要想获取更多信息, 请参阅 TMS320C28x C/C++ 优化编译器用户手册。
- **绝对地址列表程序** 接收目标文件作为输入, 创建 .abs 文件作为输出。用户可以汇编 .abs 文件生成包含绝对地址而不是相对地址的一个列表。如果没有绝对地址列表程序, 生成这种列表的工作将是冗长乏味的, 可能需要许多手工操作。
- **十六进制转换应用程序** 将 COFF 目标文件转换成 TI-Tagged, ASCII-hex, Intel, motorola-S 或 Tektronix 目标格式。这种转换后的文件可以用编程器下载到 EPROM。详情请阅读第 10 章十六进制转换应用程序介绍。
- **交叉引用列表程序** 使用目标文件来生成交叉列表, 显示符号、符号的定义及它们在已链接的源文件中的引用情况。详情请阅读第 9 章交叉引用列表程序介绍。
- 开发过程的主要产物是可以被 TMS320C28x 器件执行的程序模块。
- 用户可以使用下列几种调试工具中的一种来精简和纠正代码。可利用的产品包括:
 - 软件仿真器
 - XDS 仿真器
 - 评估板 (EVM)

要获取有关调试工具的信息, 请阅读 TMS320C28x 代码设计人员工作室用户手册。



第 2 章 通用目标文件格式介绍

汇编器和链接器可以创建 TMS320C28x™ 器件可执行的目标文件。这些目标文件的格式称为通用目标文件格式（COFF）。

COFF 使得程序的模块化编程更容易。因为它鼓励用户在编写汇编语言程序时用代码块和数据块，这些块结构称为段。汇编器和链接器均提供伪指令允许用户创建和生成段。这一章主要叙述汇编语言程序中段的概念及其使用方法。查阅附录 A 通用目标文件格式，可以获取有关 COFF 目标文件格式的更多信息。

2.1 段

目标文件的最小单元称为段。段是代码或数据块的组合，它最终将在存储器映像中占据一个连续的空间。目标文件的每一个段都是各自独立的。COFF 目标文件必须包含 3 个默认段：

- .text 段 通常包含可执行代码
- .data 段 通常包含已初始化的数据
- .bss 段 通常为未初始化变量保留空间

另外，汇编器和链接器允许用户像使用 .data, .text, .bss 段那样创建、命名和链接已命名的段。

段的两种基本类型：

已初始化段：包含数据和代码。 .data 段为已初始化段，用 .sect 汇编伪指令创建的已命名段也是已初始化段。

未初始化段：在存储器映像中为未初始化数据保留空间。 .bss 段是未初始化段，用 .usect 汇编伪指令创建的已命名段也是未初始化段。

某些汇编伪指令允许用户把代码和数据的各部分与相应的段相联系。汇编器在汇编过程中建立这些段，创建如图 2.1 那样的目标文件。

链接器的功能之一是把段重新定位到目标系统的存储器映像中，这一功能称为重定位。因为大多数系统包含几种类型的存储器，使用段可以帮助用户更有效地使用目标系统。所有的段都可以独立地重定位；用户可以把任何段放入目标系统分配的任意块中。例如，用户可以定义一个包含初始化程序的段，然后把程序分配到包括 ROM 在内的存储器映像中。

图 2.1 所示为目标文件中的段与目标存储器之间的关系。

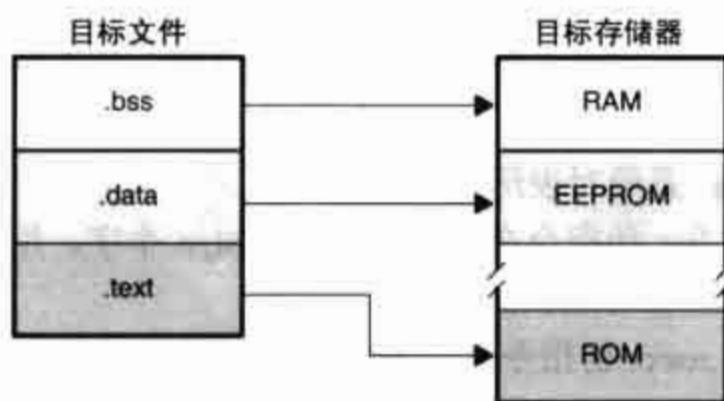


图 2.1 将存储器划分成逻辑块

2.2 汇编器如何处理段

汇编器识别属于一个给定段的部分汇编程序。汇编器有 5 个伪指令支持这一功能：

- ☐ .bss
- ☐ .usect
- ☐ .text
- ☐ .data
- ☐ .sect

.bss 和 .usect 伪指令创建未初始化段；.text, .data 和 .sect 伪指令创建已初始化段。

用户可以创建任意段的子段用来更仔细地分配存储器。子段可以使用 .sect 和 .usect 伪指

令创建。子段用由冒号分开的基段名称和子段名称来识别。

注意：默认段伪指令

如果用户不使用任何段伪指令，那么汇编器将所有内容汇编到.text 段。

2.2.1 未初始化段

未初始化段在 TMS320C28x 存储器中保留空间；它们通常被分配在 RAM 中。这些段在目标文件中没有实际内容，它们仅仅保留存储器空间而已。程序可以在运行时使用这些空间来创建和存储变量。

未初始化数据区使用.bss 和.usect 汇编伪指令建立。

□ .bss 伪指令在.bss 段内保留空间。

□ .usect 伪指令在指定的未初始化的已命名段内保留空间。

每次调用.bss 或.usect 伪指令，汇编器在.bss 或已命名段内保留额外的空间。

这些伪指令的格式如下：

```
.bss symbol, size in words [, blocking flag] [, alignment flag]  
symbol .usect "section name", size in words [, blocking flag] [, alignment flag]
```

Symbol 指向由.bss 或.usect 伪指令所保留的第一个字节。symbol 与为其保留空间的变量名一致。它可以被其他段引用，也可以被定义为全局符号（用.global 汇编伪指令）。

size in words 是绝对表达式

□ .bss 伪指令在.bss 段中保留 size 个字。用户必须指定一个值，这里没有默认值。

□ .usect 伪指令在“section name”段中保留 size 个字。用户必须指定一个值，这里没有默认值。

alignment flag 是可选参数。它指定地址分配所必需的最小队列的字节数。

section name 告诉汇编器保留的空间用于哪一个已命名段。section name 必须用引号引起来。详情请参阅 2.2.3 节已命名段。

已初始化伪指令（.text, .data, .sect）告诉汇编器停止汇编当前段，开始汇编指定段。然而.bss 和.usect 伪指令并不终止对当前段的汇编而开始一个新的段，它们只是使汇编过程暂时从当前段转移开。bss 和.usect 伪指令可以在一个已初始化的段的任何地方出现而不影响它的内容。请参见 2.2.6 节使用段伪指令实例。

由.usect 伪指令创建的未初始化的子段，汇编器将以未初始化段的方式来处理。详情参见 2.2.4 节创建子段。

2.2.2 已初始化段

已初始化段包含可执行代码和已初始化的数据。这些段的内容存放在目标文件内，在装载程序时放入 TMS320C28x 存储器中。每个已初始化段可以独立地重定位并可以引用其

它段中定义的符号。链接器自动解决这些与段有关的问题。

有 3 条伪指令告诉汇编器将代码和数据放入一个段内。这些伪指令的格式如下：

```
.text  
.data  
.sect "section name"
```

当汇编器遇到上述伪指令之一时，停止汇编当前段（相当于隐含结束当前段命令）。然后它把后续的代码汇编到指定的段内，直到遇见另一个 `.text`、`.data` 或 `.sect` 伪指令为止。

段是通过迭代过程建立的。例如，当汇编器第一次遇到 `.data` 伪指令时 `.data` 段是空的。第一个 `.data` 伪指令后面的语句就被汇编入 `.data` 段内（直到汇编器遇到一个 `.text` 或 `.sect` 伪指令）。如果汇编器遇到后面的 `.data` 伪指令，就把 `.data` 伪指令后面的语句添加到已经放到 `.data` 段的语句后面。这样就创建了惟一的 `.data` 段，它可以在存储器中分配连续地址。

已初始化子段用 `.sect` 伪指令创建，汇编器用与处理已初始化段相同的方式处理已初始化子段。有关创建子段的详情请查阅 2.2.4 节。

2.2.3 已命名段

已命名段是用户创建的段。用户可以像使用默认值 `.text`、`.data`、`.bss` 段那样使用它们，但是它们被单独汇编。

例如，重复使用 `.text` 伪指令在目标文件中建立一个 `.text` 段。链接时，`.text` 段作为一个单元在内存中分配存储空间。假设有一个可执行代码段（例如，一个初始化程序），用户不想用 `.text` 段来指定。如果用户将这一段汇编成一个已命名段，它会与 `.text` 段分开汇编，可以在存储器中单独分配空间。用户也可以与 `.data` 段分开汇编已初始化的数据，或与 `.bss` 段分开而为未初始化变量保留空间。

两个用来创建已命名段的伪指令：

- `.usect` 伪指令创建未初始化段，可以像 `.bss` 段那样使用。这些段在 RAM 中为变量保留空间。
- `.sect` 伪指令创建已初始化段，像默认的 `.text` 和 `.data` 段那样，可以包含代码和数据。`.sect` 伪指令用来创建可重定位的已命名段。

这些伪指令的格式是：

```
Symbol .usect "section name", size in words [, blocking flag] [, alignment flag]  
.sect "section name"
```

参数 `section name` 是段的名称。段名可以多至 200 个字符。用户可以创建多至 32767 个单独命名的段。对 `.sect` 和 `.usect` 伪指令而言，一个段名可以指向一个子段。

每当用户用一个新名字调用这些伪指令之一时，就可创建一个新的已命名段。每当用一个已存在的名字调用这些伪指令之一时，汇编器将代码或数据（保留空间）汇编入以此名字命名的段中。不能用不同的伪指令调用相同名字的段。这就是说，不能用 `.usect` 伪指令创建一个段，然后又用同一段名字指定为 `.sect` 段。

2.2.4 子段

子段是较大段内相对较小的段。和段一样，子段也可以由链接器处理。可以利用子段更仔细地分配存储器。子段可以使用 .sect 和 .usect 伪指令创建。这些伪指令的格式如下：

```
symbol .usect "section name:subsection name", size in words [, alignment]
.sect "section name:subsection name"
```

子段用由基段名称加冒号加子段名称来识别。例如，用户用下列代码创建一个 .text 段内的叫做 _func 的子段：

```
.sect ".text:_func"
```

一个子段可以被单独指定地址，或是与用同一基段名称的其他子段组合在一起存放。使用链接器的 SECTIONS 伪指令，可以为 .text:_func 段单独分配地址或是与所有的 .text 段一起分配。

可以创建两种类型的子段：

- 使用 .sect 伪指令创建已初始化子段。见 2.2.2 节已初始化段。
- 使用 .usect 伪指令创建未初始化子段。见 2.2.1 节未初始化段。

子段的地址分配与段的分配方式相同。详见 7.8 节段伪指令。

2.2.5 段程序计数器

汇编器为每一个段使用单独的程序计数器。这些程序计数器称为段程序计数器或 SPCs。

SPC 代表代码段或数据段内的当前地址。开始时，汇编器将每个 SPC 置 0。随着汇编器向段内填充代码或数据，它使相应的 SPC 值增加。如果用户想进入段中继续汇编，汇编器会记住相应的前一次 SPC 的值，然后从这一点开始继续增加 SPC 的值。

汇编器按起始地址为 0 来处理每一个段；链接器按照段最终在存储器映像中的单元重新定位每个段。详细资料见 2.4 节的重定位。

2.2.6 使用段伪指令实例

用户可以使用段伪指令开始第一次汇编到一个段内，或者继续汇编到已经包含代码的段。对后者而言，汇编器只是简单地在段内已存在的代码后面添加新代码。

例 2.1 使用段伪指令在段之间来回交换从而逐渐建立 COFF 段

```
1          ****
2          ** 汇编已初始化表到 .data 段中          **
3          ****
4 00000000          .data
5 00000000 0011  coeff          .word 011h, 022h, 033h
   00000001 0022
```

```

00000002 0033
6
7 *****
8 ** 在.bss 段中为变量保留空间 **
9 *****
10 00000000          .bss buffer, 10
11
12 *****
13 **仍处于.data 段 **
14 *****
15 00000003 0123 ptr      .word 0123h
16
17 *****
18 ** 汇编程序代码到.text 段中 **
19 *****
20 00000000          .text
21 00000000 28A1 add:      mov ar1, #0Fh
   010000001 000F
22 00000002 0BA1 aloop:    dec ar1
23 00000003 0009      banz aloop, ar1--
   00000004 FFFF
24
25 *****
26 ** 汇编另一初始化表到.data 段中 **
27 *****
28 00000004          .data
29 00000004 00AA ivals     word 0AAh, 0BBh, 0CCh
   00000005 00BB
   00000006 00CC
30
31 *****
32 ** 为更多的变量定义其他的段 **
33 *****
34 00000000 var2      .usect "newvars", 1
35 00000001 inbuf     .usect "newvars", 7
36
37 *****
38 ** 汇编更多的程序代码到.text 段中 **
39 *****
40 00000005          .text
41 00000005 28A1 end_mpy:   mov ar1, #0Ah
   00000006 000A
42 00000007 33A1 mloop:    mpy p,t,ar1
43 100000008 28AC      mov t, #0Ah
   00000009 000A
44 0000000a 3FA1      mov ar1, p
45 0000000b 6BFA      sb end_mpy, 0V

```

区域1
 区域2
 区域3
 区域4

例 2.1 是列表文件的格式。例 2.1 说明了汇编过程中 SPCs 是如何被修改的。列表文件的每一行有 4 个区域：

- 区域 1 源代码行计数器。
- 区域 2 段程序计数器。
- 区域 3 目标代码。
- 区域 4 最初的源代码语句。

解释源程序列表区域的详情参见 3.1.1 节源程序列表。

图 2.2 所示为例 2.1 中的文件创建了 4 个段：

- .text** 包含 10 个 32 位字的目标代码。
- .data** 包含 5 个字的目标代码。
- .bss** 在内存中保留 10 个字。
- newvars** 用 .usect 伪指令指定的已命名段，它在存储器中保留 8 个字。

第一列表示生成这些目标代码的源代码行号，第二列表示被汇编到这些段的目标代码。

行号	目标代码	段
5	0011	.data
5	0022	
5	0033	
15	0123	
29	00AA	
29	00BB	
29	00CC	
21	28A1	.text
21	000F	
22	0BA1	
23	0009	
23	FFFF	
41	28A1	
41	000A	
42	33A1	
43	28AC	
43	000A	
44	3FA1	
45	6BFB	
10	无数据 保留 10 个字	.bss
34	无数据 保留 8 个字	newvars
35		

图 2.2 例 2.1 中的文件生成的目标代码

2.3 链接器如何处理段

链接器有两个与段相关的主要功能。第一功能，链接器用 COFF 目标文件中的段作为

装配块。当链接多于一个文件时，它组合输入段，从而在可执行的 COFF 输出模块中创建输出段；第二功能，链接器为输出段选择存储器地址。

两个链接伪指令支持这些功能：

- **MEMORY** 伪指令允许用户定义目标系统的存储器映像。用户可以为存储器的某一部分命名并指定其起始地址和长度。
- **SECTIONS** 伪指令告诉链接器如何将输入段组合到输出段，以及将这些输出段存放在存储器的什么地方。

子段允许用户更仔细地对段进行操作。可以用链接器的 **SECTIONS** 伪指令指定子段。如果用户没有明确指定子段，那么这个子段就和与它共用一个基段名的段组合在一起。

通常不是必须使用链接器伪指令。如果用户不使用它们，那么链接器就使用默认分配算法来处理段。7.13 节默认分配介绍了这部分内容。当用户使用链接器伪指令时，必须在链接器命令文件中指明。

有关链接器命令文件和链接器伪指令的详细介绍见第7章链接器介绍。

2.3.1 默认内存分配

图 2.3 所示为两个文件的链接过程。

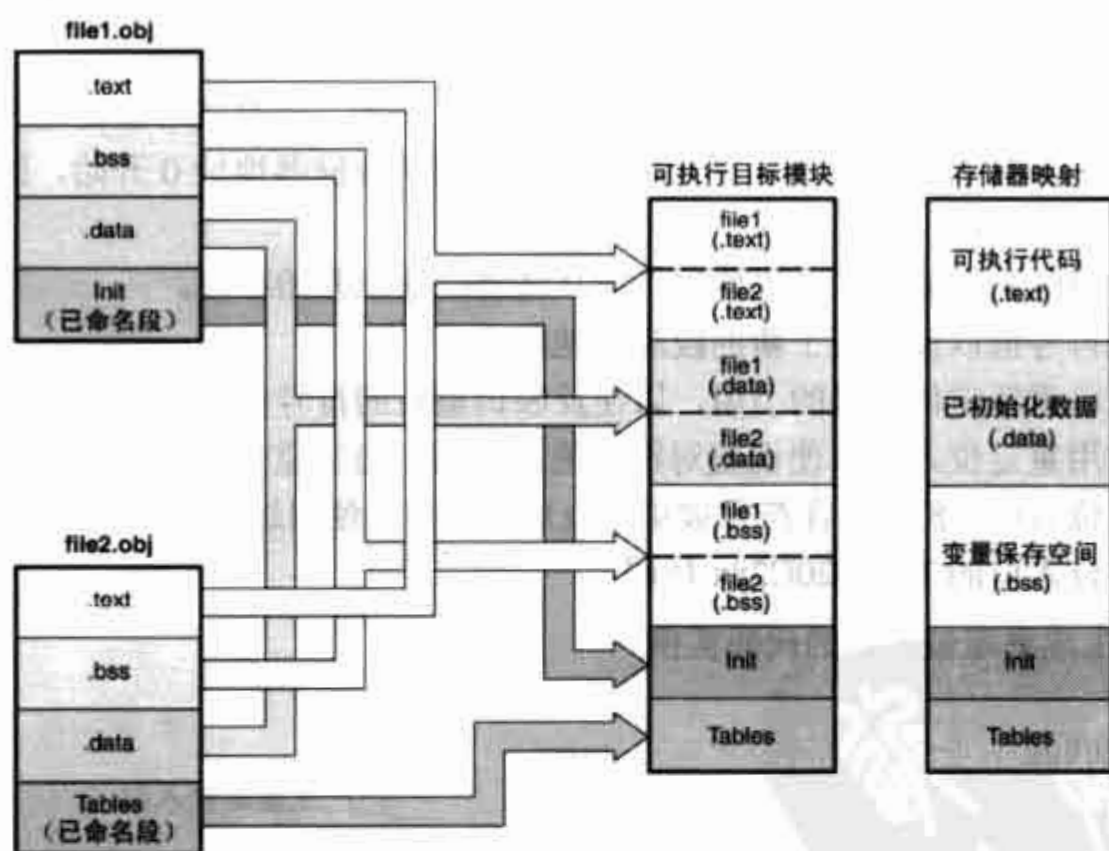


图 2.3 连接输入段组成一个可行执目标模块

在图 2.3 中，`file1.obj` 和 `file2.obj` 已被汇编作为链接器输入。每个文件都包含 `.text`、`.data` 和 `.bss` 默认段；另外，每一个段还包含一个已命名段。可执行目标模块显示了段组合后的情况。链接器将 `file1.obj` 的 `.text` 段和 `file2.obj` 的 `.text` 段组合在一起组成了一个 `.text` 段，然后分别连接两个 `.data` 段和两个 `.bss` 段，再将已命名段放在后面。存储器映像说明每个段在存储器中如何存放。

在默认方式下, 链接器从地址 0h 开始按下列顺序分配段: .text, .const, .data, .bss, .cinit, 已命名段。已命名段按照它们在输入文件中出现的顺序分配。

C/C++ 编译器使用 .const 段来存储定义为常数的变量或数组。编译器为自动初始化的全局变量生成数据表; 这些变量存储在一个叫做 .cinit 的已命名段内。

了解有关 .const 和 .cinit 的更多信息, 参见 TI 公司的 TMS320C28x 优化 C/C++ 编译器用户手册。

2.3.2 在存储器映像中存放段

图 2.3 所示说明了链接器组合段的默认方式。有时用户可能不想使用默认方式。例如, 用户也许不想将所有的 .text 段结合成一个 .text 段; 或者想把一个已命名段放在通常存放 .data 段的地方。大多数使用情况中, 存储器映像都包含大小不一、类型各异的存储器 (RAM, ROM, EPROM 等); 用户有可能想把一个段存放在某种特定的存储器中。

若想更好地了解段存放方面的知识, 请参阅 7.7 节存储器伪指令和 7.8 节段伪指令。

2.4 重定位

汇编器处理每一段的时候都认为段地址从地址 0 开始。所有的可重定位符号 (标号) 是与段内地址 0 相对应的。实际上不可能所有的段都从存储器地址 0 开始, 因此, 链接器用下列方式重新定位段:

- 将段分配到存储器中, 以便每个段从适当的地址开始存放。
- 调整符号值以便对应于新的段起始地址。
- 调整已重新定位符号的引用, 以便反映调整后的符号值。

链接器使用重定位入口以便调整对符号值的引用。每当重定位符号被引用, 汇编器就创建一个重定位入口。然而, 在符号被重定位后, 链接器使用这些入口来修正引用。例 2.2 包含生成重定位入口的 TMS320C28x 代码。

例 2.2 生成重定位入口的代码实例

```

1          .global X
2 00000000 .text
3 00000000 0080' LC Y          ; 产生重定位入口
   00000001 0004
4 00000002 28A1! MOV AR1, #X   ; 产生重定位入口
   00000003 0000
5 00000004 7621 Y:  IDLE

```

在例 2.2 中, 符号 X 和 Y 都是可重定位的。Y 定义在模块的 .text 段中; X 定义在另一模块中。当汇编代码时, X 的值为 0 (汇编器指定所有未定义的外部符号值为 0), Y 的值为 4 (相对段的地址 0 而言)。汇编器为 X 和 Y 生成两个重定位入口。X 是外部引用 (由列表中的 “!” 字符指出)。Y 的引用是内部定义的可重定位符号 (在列表中由 “'” 字符指出)。

当代码链接结束后，假设 X 被重新定位到 0x7100，.text 段重新定位后从地址 0x7200 开始，则 Y 重定位后的值为 0x7204。链接器使用两个重定位入口来修整目标代码中的两个引用：

0080'	LC	Y	becomes	0080'
0004				7204
28A1!	MOV	AR1, #X	becomes	28A1!
0000				7100

有时一个表达式包含多个可重定位的符号，或者表达式的值不能在汇编过程中求得。在这种情况下，汇编器在汇编目标文件时对整个表达式进行编码。决定了符号地址后，链接器再计算表达式的值，如例 2.3 所示。

例 2.3 重定位表达式

```

1          .global sym1, sym2
2  00000000 FF20%    mov ACC, #(sym2-sym1)
3  00000001 0000
```

符号 sym1 和 sym2 都是外部定义的，因此，汇编器不能求出表达式 sym1 和 sym2 的值，汇编器将整个表达式编码到目标文件。“%”字符标明重定位表达式。假设链接器重定位 sym2 为 300h，sym1 为 200h，那么链接器计算表达式的值为 300h-200h=100h。这样，MOV 指令就修改为：

```

00000000    FF20    mov    ACC, #(sym2-sym1)
01          0100
```

注意：表达式的值不能大于为它保留的空间。

如果表达式的值在位数上大于为它所保留的空间，链接器将报告错误信息。

COFF 目标文件的每一个段都有一个重定位入口表。表中包含了段内每一个可重定位引用的重定位入口。链接器通常在使用完后将它清空以防止输出文件再次重定位（当文件再次链接或是装载时）。不包含重定位入口的文件是绝对文件（它的所有地址都是绝对地址）。如果用户想让链接器保留重定位入口，那么可用 -r 选项调用链接器。

运行中的重定位

有时用户可能想将代码放在存储器的某个区域，而在另一个区域运行它。例如，在以外存储器为基础的系统，用户可能有严格性能要求的代码。这些代码必须装入外部存储器，但在内部存储器中运行得更快。

链接器提供了一种简单的方式来处理这种情况。使用 SECTIONS 伪指令，用户可以选择地指示链接器对一个段进行两次重定位：一次设置其装载地址，一次设置运行地址。对装载地址用关键词 load 设定，对运行地址用关键词 run 设定。

装载地址决定了装载器将段的原始数据放在何处。对于段的任何引用（例如，段内标号）都指向它的运行地址。当运行时，在第一次引用符号之前，应用程序应该将段从其装

载地址复制到你运行地址。这一过程不能简单地自动实现，因为运行地址是你单独指定的。运行时如何移动代码请参见例 7.6。

如果你仅为段提供了一种分配（装载或是运行），那么该段只进行一次地址分配而且在同一地址下装载和运行；如果提供了两次分配，那么该段实际分配地址时，就如同对两块同样大小的段分配地址一样。

未初始化段（例如.bss）不作装载，因此惟一有意义的地址是运行地址。链接器只对未初始化段分配一次地址；如果你既指定了装载地址，又指出了运行地址，链接器会发出警告并忽略装载地址。

运行中重定位的一个完整实例见 7.9 节指定一个段在运行中的地址分配情况。

2.5 装 载 程 序

链接器生成 COFF 可执行目标模块。一个可执行目标文件与用做链接器输入的目标文件有共同的 COFF 格式；然而，可执行目标文件的段被组合并重新定位到目标存储器中。

运行一个程序时，可执行目标模块的数据必须传送或装载到目标系统的存储器中。根据运行环境的不同，可以采用几种方法来装载一个程序。两种常用环境如下：

- TMS320C28x 调试工具，包括仿真器、内置装载器。每种工具都有 LOAD 命令，用来调用装载器；装载器读取可执行文件，并将程序复制到目标存储器。
- 用户可以使用十六进制转换应用程序（hex2000，它是作为汇编语言包的一部分提供的）将可执行 COFF 目标模块转换成某种目标文件格式。然后，用户可以用 EPROM 编程器将转换过的程序写入 EPROM。

2.6 COFF 文件中的符号

COFF 文件包含符号表，用来存储程序中的符号信息。

链接器执行重定位时使用这个表。调试工具在提供符号调试时也使用这个表。

2.6.1 外部符号

外部符号是指在某一模块中定义，而在另一模块中引用的符号。使用 .def, .ref, .global 伪指令可以将符号定义为外部符号：

.def	在当前模块中定义并在另一模块中使用；
.ref	在当前模块中引用，但在另一模块中定义；
.global	可以是上面两种情况之一。

举例描述：

x用.def定义，说明它是一个外部符号，在本模块中被定义而在另一模块中可以引用。y

用.ref定义,说明它在本模块中未定义,但在其他模块中定义过。z用.global定义,说明它在本模块中未定义,在其他模块中定义过。a用.global定义,说明它是一个外部符号,在本模块中定义,在其他模块可以引用。

```
.def          x
.ref          y
.global       z
.global       a
x:    ADD     AR1, #56h
      B       y, UNC

a:    ADD     AR1, #56h
      B       z, UNC
```

汇编器将x, y, z, a放在目标文件的符号表中。当该文件与其他文件链接时,关于x和a的入口定义提供给其他文件对未定义的x和a的引用。链接器在其他文件的符号表中寻找y和z的定义以确定y和z的入口。

链接器必须使所有的引用与相应的定义相匹配。如果链接器找不到一个符号的定义,那么它将打印出有关未定义引用的错误信息。这种类型的错误阻止链接器创建一个可执行目标模块。

2.6.2 符号表

当汇编器遇到外部符号(外部符号用2.6.1节中任一条伪指令定义和引用)时,它总是在符号表创建一个入口。汇编器还创建指向每一个段起始的特殊符号;链接器使用这些符号以便重新定位到段内的其他符号。

汇编器通常不为上述符号之外的任何符号创建符号表入口,因为链接器不使用它们。例如,除非标号被声明为.global,否则它们不被包含在符号表内。为了符号调试,在符号表中为程序中每一个符号建立相应的入口有时是很有用的。要实现这一功能,可用-s选项调用汇编器(见-s选项介绍)。





第 3 章 汇 编 器

TMS320C28x 汇编器把汇编语言源程序文件汇编成机器语言目标文件，这些目标文件采用的是通用目标文件格式。上述内容将在通用目标文件格式、附录中进行讨论。源文件包括以下汇编语言要素：

- 汇编伪指令 在第 4 章说明
- 宏伪指令 在第 5 章说明
- 汇编语言指令 在第 11 章说明

3.1 汇编器功能

双扫描汇编器将完成下述功能：

- 处理文本文件格式的源程序语句，从而生成可重定位的目标文件。
- 按要求生成源程序列表，并且可以定制。
- 允许用户把代码分为几段，并为目标代码的每一个段都保存段程序计数器。

- 定义和引用全局符号，并在源程序列表中按要求添加交叉引用列表。
- 允许条件汇编。
- 支持宏定义，允许内嵌或在库中定义宏。

3.2 在软件开发过程中汇编器的作用

图 3.1 所示为在软件开发过程中汇编器的作用，阴影部分表示汇编器常用的开发路径。汇编器接收汇编源程序。

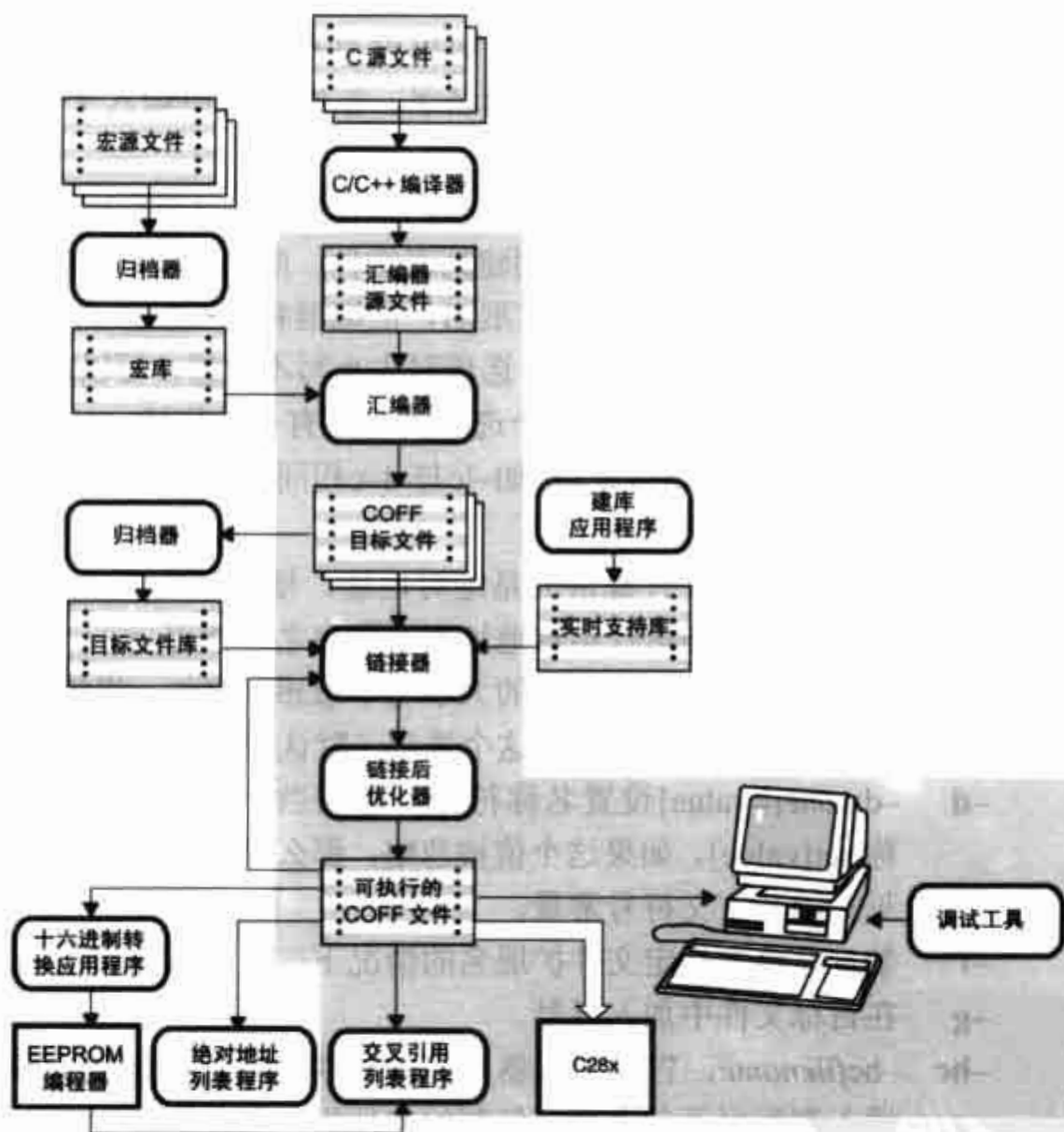


图 3.1 汇编器在 TMS320C28x 软件开发流程中

3.3 运行汇编器

运行汇编器用下面的格式：


```
asm2000 version [input file [object file [listing file ] ] ] [options]
```

- asm2000** 是运行汇编器的命令。
- version** 指对源程序文件进行汇编的目标处理器。有效的版本version为-v28和-v27。-v28表示对应的是TMS320C28x处理器；-v27表示对应的是TMS320C27x处理器。Version必须指定，否则汇编器将报错。如果version规定既是-v27又是-v28，那么汇编器将忽略后者，并提出警告。详情请参见3.15节TMS320汇编器模式。
- input file** 命名汇编语言源程序。如果没有提供扩展名，汇编器将默认扩展名为.asm。若没有提供输入文件的名称，汇编器将提示用户需要输入文件名。
- object file** 命名汇编器创建的目标文件。如果没有指定扩展名汇编器默认扩展名为.obj。若没有提供目标文件的名称，汇编器将用输入文件的名称加.obj扩展名作为目标文件的名称。
- listing file** 命名汇编器创建的可选的列表文件。详情参见3.11节程序列表。若没有提供列表文件名，汇编器不会创建列表文件。除非选择了-l或-x选项。若提供了列表文件的名称而没有扩展名，汇编器将默认扩展名是.lst。
- options** 指定用户想用的汇编器选项。选项对大小写不敏感，可以出现在命令行内命令之后的任何位置。每一个选项的前面有一个短线（-）。没有参数的单个字符的选项可以合并，例如-lc与-l-c相同。具有参数的选项，例如-i 必须单独指定。
- a 创建的列表文件提供的是绝对地址。使用-a选项，汇编器不会生成目标文件。在运行绝对地址列表程序之后使用-a。
 - c 使汇编语言文件中字符的大小写不敏感。例如-c将使字符ABC与abc等价。如果用户不使用这个选项（默认值），那么大小写是敏感的。
 - d -dname[=value]设置名称符号。这相当于在汇编语言的开始插入名称.set[value]。如果这个值被忽略，那么符号将被设置为1。详情请参见3.8.4节定义符号常量。
 - f 禁止在没有指定文件扩展名的情况下，汇编器默认扩展.asm。
 - g 在目标文件中加入行号。
 - hc -hcfilename，告诉汇编器从汇编模块中复制指定的文件。这个文件插入源程序语句之前。复制的文件将出现在汇编列表文件中。
 - hi -hifilename，告诉汇编器在汇编模块中包括了指定的文件。此文件被包含在源程序语句之前。被包含的文件不出现在汇编文件列表中。
 - i 规定汇编器在其中寻找被.copy、.include、.mlib伪指令指定的文件的目录。-i选项的格式是-i 路径名。每个路径名必须由-i引导。详细资料见3.4.1节使用汇编器的-i项。
 - l （小写字母L）生成一个名称与输入文件同名的列表文件.ls。
 - m20接收C2xLP的汇编指令，并把它们当作C28x指令进行汇编。-m20选

项暗含了-v28选项。详细资料见3.15节TMS320C28x汇编模式。

-m27 接收C27的指令格式。-m27选项暗含了-v28选项。详细资料见3.15节TMS320C28x汇编模式。

-mf 允许用大存储器模式代码对16位的代码进行条件汇编。定义LARGE_MODEL的符号并设置为真。

-mg 为汇编变量生成C语言类型调试符号,此汇编变量在汇编源代码中用数据伪指令定义。它支持基本的C语言类型、结构体和数组。

-mw 启动附加的汇编检查。如果.bss分配的空间大于64个字或16位立即数的值超出-32768~+65535的范围将发出警告。

-q 抑制标识和所有进程信息(汇编器在静态模式下运行)。

-s 把所有已定义的符号放入目标文件的符号表中。汇编器通常只把全局符号放入符号表中。当用户使用-s时,定义为标号或汇编编译过程使用的常数符号也被放入此表中。

-u -uname, 取消预定义常量名称,禁止所有-d选项规定的常量。

-x 生成一个交叉引用列表并把它添加在列表文件的后面;并在目标文件中加入交叉引用的信息。如果用户不要求生成列表文件但使用了-x选项,汇编器将自动生成一个列表文件,此列表文件的名称是:输入文件名.lst。

3.4 为汇编器输入的替换目录命名

.copy、.include、.mlib 伪指令告诉汇编器使用来自外部文件的代码。.copy 和.include 伪指令告诉汇编器从其他文件读入源代码语句,.mlib 伪指令指定从包含宏函数的库中读取。第4章汇编伪指令中包含.copy、.include、.mlib 伪指令的例子。伪指令的格式是:

```
.copy "filename"  
.include "filename"  
.mlib "filename"
```

filename 命名了汇编器将从其中读出语句的复制/包含文件以及包含宏定义的宏库。文件名是完整的路径名、部分路径名或没有路径信息的文件名。编译器以下面的顺序寻找文件的位置。

- (1) 包含当前源程序文件的目录。当前源文件是指遇到.copy、.include 或.mlib 伪指令时正在汇编的源程序文件。
- (2) 所有-i 汇编器选项命名的目录。
- (3) 用环境变量 C2000_A_DIR 或 A_DIR 命名的目录。

用户可以通过使用-i 汇编器选项(见3.4.1节)或用 C2000_A_DIR 或 A_DIR 环境变量(见3.4.2节)来增大汇编器目录搜索的范围。

3.4.1 使用-i 汇编器选项

-i 汇编器选项指定另外可选的目录，它包含 .copy 或 .include 的文件或宏。-i 的格式如下：

```
asm2000 -v28 -ipathname source filename [other options]
```

每一个-i 选项指定一个路径名。在汇编源程序中，用户可以使用 .copy、.include 和 .mlib 伪指令而不用指出路径信息。如果汇编器在包含当前源程序的目录中找不到所需的文件，它将按-i 选项指定的路径寻找。

例如，假定当前的目录中有一个名字为 source.asm 的文件，且此文件含有以下伪指令语句：

```
.copy "copy.asm"
```

假设 copy.asm 文件的备选路径是：

UNIX: /320tools/files/copy.asm

Windows: c:\320tools\files\copy.asm

操 作 系 统	输 入
UNIX	asm2000 -v28 -i/320tools/files source.asm
Windows	asm2000 -v28 -ic:/320tools/files source.asm

若在上述系统中调用汇编器，因为 copy、.asm 在当前目录中，所以汇编器首先搜索当前目录。然后搜索-i 选项指定的目录。

3.4.2 使用 C2000_A_DIR 或 A_DIR 环境变量

环境变量是系统符号，用户可以定义并把字符串赋予它。汇编器使用 C2000_A_DIR 或 A_DIR 环境变量来指定包含 .copy、.include 的文件或宏库所在的另外目录。用于为环境变量赋值的命令格式是：

操 作 系 统	输 入
UNIX	tenvA_DIR "pathname1;pathname2;..."
Windows	set A_DIR= pathname1;pathname2;...

路径名字是包含 .copy、.include 的文件或宏库的目录。可以用分号或空格分隔路径名。

3.5 源程序语句格式

TMS320C28x 汇编语言源程序由源程序语句组成，源程序语句包含汇编伪指令、汇编

语言指令、宏伪指令和注释。源语句包括4个域（标号，助记符[操作数列表]；注释）。语句通常的格式是：

[label] [:] [ll] mnemonic [operand list] [;comment]

例：

```

                two .set 2      ; 符号 two=2
Begin:  MOV  AR1,#two      ; AR1=2
                .word 016h    ; 用 016h 初始化一个字

```

C28x汇编器每行最多200个字符，超出200个字符将被截断。为了正确汇编源程序语句（除注释以外的其他部分），必须确保每行不超过200个字符。注释可以超出200个字符的限制，但是超出的部分在列表文件中将被截断。

书写源程序的规则：

- 所有语句必须以标号、空格、星号或分号开头。
- 标号是可选的，若用标号，它必须写在第一列的开始。
- 必须用一个或多个空格分隔每一域。制表符（Tab）等效为空格。
- 注释是可选的。在第一列开始的注释可以用星号或分号（*或;）打头，但在任何其他列开始的注释必须以分号（;）开头。
- 助记符不能在第一列开始，否则将被视为标号。

下面将对每一个域进行说明。

3.5.1 标号域

标号对所有的汇编语言指令和大多数汇编伪指令是可选的。当使用标号时，标号必须从源程序语句的第一列开始。标号可以由128个字符（A~Z，a~z，0~9，_和\$）构成。标号对大小写敏感（除非使用了-c选项），而且标号开头字符不能是数字。标号后面可以跟冒号，冒号并不作为标号的一部分。如果用户不使用标号，那么第一字符位置必须是空格、分号或星号。不能使用指令符作为标号。

当用户使用标号时，它的值就是段程序计数器（SPC）的当前值。标号指向它所对应的语句。例如，用户用.word伪指令初始化几个字，标号将指向第一个字。在下例中标号Start的值是40h。

```

. . . . .
. . . . .
. . . . .
9
10 000040 000A Start:      * 假定前面已汇编了一些程序代码
    000044 0003           .word 0Ah,3,7
    000048 0007

```

单独出现在行中的标号是有效的语句，这将把段程序计数器的当前值赋予标号，即等效于下列伪指令语句。

label .set \$;\$符号提供 SPC 的当前值

当标号单独出现在行中时，它指向下一行中的指令（段程序计数器不增加）。

```
1  000000          Here:
2  000000  0003      .word 3
```

3.5.2 助记符域

助记符域跟随在标号域之后。助记符不能从第一列开始，否则它将被认为是标号。当前面的指令是RPT指令时，助记符域可以用双竖符号(II)开头。双竖符号表示此指令是被重复的指令。

```
      RPT
II  Inst2  ← 重复此指令
```

助记符域可以包含下列操作码之一：

- ☐ 机器码助记符（如 ADD、MOV 或 B）
- ☐ 汇编伪指令（如.data、.list 或.set）
- ☐ 宏伪指令（如.macro、.var 或.mexit）
- ☐ 宏调用

3.5.3 操作数域

操作数域是跟随在助记符域之后的操作数列表，它包含一个或多个操作数。并不是所有的指令或伪指令都需要操作数。操作数由以下项构成：

- ☐ 符号（见 3.8 节）
- ☐ 常量（见 3.6 节）
- ☐ 表达式（常量和符号的混合，见 3.9 节）

操作数必须用逗号分隔。

3.5.4 注释域

注释可以从任何一列开始并扩展到源程序语句行的末尾。注释可以包含任何ASCII字符，包括空格。注释出现在汇编源程序列表中，但是它们不影响汇编。

仅包含注释的源语句也有效。如果它从第一列开始，那么它可以用分号(;)或星号(*)开始。在行的其他任何地方开始的注释必须用分号(;)开始。只有当注释出现在第1列时可以用星号(*)开始。

3.6 常 量

汇编器支持7种类型的常量：

- 二进制整数
- 八进制整数
- 十进制整数
- 十六进制整数
- 字符
- 汇编编译过程使用的 (Assembly-time) 常量
- 浮点数

汇编器在内部把每一个常量当作32位的数存储。常量不进行符号扩展,例如,常量00FFh等于00FF (16进制) 或255 (10进制)。但当使用.byte伪指令时, -1等于00FFh。

3.6.1 二进制整数

二进制整数是一串以B或b为后缀的32位阿拉伯数字 (0或1)。如果指定了少于32位的常量,汇编器将向右对齐,没有值的位填零。下面是有效二进制常量的例:

00000000B	该常量等于 0_{10} 或 0_{16}	(注: 数字的下标为进制标志)
0100000b	该常量等于 32_{10} 或 20_{16}	
01b	该常量等于 1_{10} 或 1_{16}	
11111000B	该常量等于 248_{10} 或 $0F8_{16}$	

3.6.2 八进制整数

八进制整型常量是一个以Q或q为后缀的不超过11位的8个阿拉伯数字 (0至7) 构成的数字串。下面是八进制数的例:

10Q	该常量等于 8_{10} 或 8_{16}
100000Q	该常量等于 32768_{10} 或 8000_{16}
226q	该常量等于 150_{10} 或 96_{16}

或者, 可以对八进制常量使用C语言的符号:

010	该常量等于 8_{10} 或 8_{16}
0100000	该常量等于 32768_{10} 或 8000_{16}
0226	该常量等于 150_{10} 或 96_{16}

注意: C2xlp 句法模式不接收八进制数

当使用-v28 -m20 选项时, asm2000 接收 C2xlp 的源代码。C2xlp 汇编器把以 0 开头的数字作为十进制整数, 如 010 作为 1010。因此在使用-v28 -m20 -mw 调用 asm2000 汇编器时, 当遇到八进制数时汇编器将提出警告。

3.6.3 十进制整数

十进制整型常量是由 10 个阿拉伯数字构成的数字串, 它的取值范围是: -2147483648

到+4294967295。下面是有效的十进制常量的例：

1000 或 1000d	该常量等于 1000_{10} 或 $3E8_{16}$
-32 768	该常量等于 -32768_{10} 或 8000_{16}
25	该常量等于 25_{10} 或 19_{16}

3.6.4 十六进制整数

十六进制整型常量是以 H 或 h 为后缀的十六进制数字构成的数字串，数字串最多不超过 8 个十六进制的数。十六进制数字包括 0~9 以及字母 A~F 或 a~f。十六进制常量必须以十进制的数字（0~9）开头。如果指定了少于 8 位的常量，汇编器将向右对齐，没有值的位填零。下面是有效的十六进制常量的例：

78h	该常量等于 120_{10} 或 0078_{16}
0Fh	该常量等于 15_{10} 或 $000F_{16}$
37Ach	该常量等于 14252_{10} 或 $37AC_{16}$
或者，可以对十六进制数使用 C 语言的符号：	
0x78	该常量等于 120_{10} 或 0078_{16}
0Xf	该常量等于 15_{10} 或 $000F_{16}$
0x37Ac	该常量等于 14252_{10} 或 $37AC_{16}$

3.6.5 字符常量

字符常量是被单引号括起来的单个字符。字符在内部以 8 位 ASCII 码表示。单引号用两个连续的单引号来表示，它是字符串常量的一部分。仅由两个单引号构成的字符常量是有效的，并且被赋值为 0。下面是有效的字符常量：

'a'	定义字符 a 并在内部用 61_{16} 表示
'C'	定义字符 C 并在内部用 43_{16} 表示
'"	定义字符 ' 并在内部用 27_{16} 表示
' '	定义空字符并在内部用 00_{16} 表示

注意：字符常量与字符串的区别，字符常量表示单个的整数，而字符串是一系列的字符。

3.6.6 汇编编译过程使用的（Assembly-Time）常量

如果用户使用 .set 伪指令给符号赋值，那么该符号就变为常量。如果在表达式中要使用该常量，则必须赋给它确定的值。例如：

```
shift3      .set      3
MOV         AR1, #shift3
```

用户也可以用 .set 伪指令为寄存器分配一个符号常量。在这种情况下，此符号是寄存器的别名。

```
myReg    .set    AR1
MOV      myReg,   #3
```

3.6.7 浮点型常量

浮点型常量由十进制数加小数点、小数部分及指数部分构成。浮点型数字的格式是：

[+|-][nnn].[nnn][Ee[+|-]nnn]

用十进制数字串代替nnn，可以在nnn前面加正号或负号，必须指定一个小数点。例如3.e5是有效的，但3e5是无效的。指数部分表示10的幂。下面是有效的浮点数常量例：

```
3.14
.3
-0.314e13
+314.59e-2
```

3.7 字符串

字符串是包含在双引号中的一串字符。双引号也是字符串的一部分，由两个连续的双引号表示。字符串的最大长度由使用字符串的伪指令来定义。字符在内部是以8位ASCII码表示。

下面是有效的字符串的例子：

“sample program” 定义14个字符的字符串**sample program**
“PLAN “C” ” 定义8个字符的字符串**PLAN“C”**

字符串应用在以下几个方面：

- 文件名字，如 .COPY “filename”
- 段名，如 .sect “section name”
- 数据初始化伪指令，如 .byte “charstring”
- .string伪指令中的操作数

3.8 符号

符号（Symbols）用来作为标记、常量和替换符号。符号名是最多可达200个字母或数字符号（A~Z，a~z，0~9，\$和_）的字符串。符号的首字符不能是数字，符号内不能有空格。用户定义的符号对大小写敏感，例如汇编器认为ABC、Abc和abc是不同的符号。用户可以利用汇编器的-c选项去除大小写的敏感性。除非用户使用.global或.def伪指令把符号声明为外部符号，否则符号仅在定义它的汇编程序中有效。

3.8.1 标号

用作标号的符号就成了程序中与它相对应单元的符号地址。在程序中使用的标号必须是惟一的。助记符操作码和汇编伪指令的名字（没有·前缀.）是有效的标号名。

标号也可以用作.global、.ref、.def或.bss伪指令的操作数，如下所示：

```

                .global label1
label2:         NOP
                ADD     AR1      ; label1
                SB      label2   ; UNC

```

3.8.2 局部标号

局部标号是一种特殊标号，它的使用范围和有效期是暂时的。局部标号可以使用两种方式定义：

- \$n 此处n是0~9的十进制数。例如，\$4和\$1是有效的局部标号。参见例3.1。
- name ? 此处的name ?是前面提到的任何合法的符号名。汇编器用后跟特定数字的周期性重复字符代替这个问号。当源代码扩展时，在列表文件中看不到特定的数字。标号以源代码中定义的问号形式出现。用户不能将这种标号声明为全局标号。

通常情况下标号是惟一的（它们只能被定义一次），并且它们可以作为常量使用在操作数字段中。当然，它们可以被取消后再定义。局部标号不能用伪指令定义。

局部标号可以通过以下四种方式之一取消定义或重新设置：

- 使用.newblock伪指令
- 改变段（用.sect、.text 或 .data伪指令）
- 进入一个.include文件（由.incinclude或.copy伪指令规定的）
- 离开一个.include文件

例 3.1 \$n 形式的局部标号

合法命名和使用局部标号的例子：

```

$1:
    ADDB AL, #-7
    B $1, GEQ
    .newblock           ; 取消$1 定义，以便再一次使用
$1
    MOV T, AL
    MPYB ACC, T, #7
    CMP AL, #1000
    B $1, LT

```

非法使用局部标号的例子：

```

$1:
    ADDB AL, #-7
    B $1, GEQ

```

```
$1      MOV T, AL          ; 警告——$1 被多次定义
        MPYB ACC, T, #7
        CMP AL, #1000
        B $1, LT
```

\$1 标号被第 2 个跳转指令重新使用之前没有取消定义，因此对 \$1 重新定义是非法的。

局部标号在宏中尤其有用。如果一个宏包含了一个标准的标号且被多次调用，那么汇编器会报告多次定义的错误。但是，如果用户在宏中使用了局部标号和 `newblock` 伪指令，那么在每一次宏扩展时局部标号就会被应用和重新设置。

一次最多只能使用 10 个局部标号。在取消局部标号的定义后可以重新定义和使用它。局部标号不会出现在目标代码符号表中。

跳转到局部标号的程序不能扩展，因为局部标号只能在局部使用，而某些跳转的偏移量可能超出了范围。

例 3.2 合法声明和使用局部标号的代码

```
*****
**第一次定义局部标号 mylab                               **
*****
        nop
mylab?  nop
        B mylab?, UNC
*****
**包含文件中有 mylab 局部标号的第二次定义               **
*****
        .copy "a.inc"
*****
** 局部标号 mylab 的第三次定义，在从 .include 文件中退出后复位 **
*****
mylab?  nop
        B mylab?, UNC
*****
** 在宏中第四次定义 mylab，为了避免冲突，宏使用不同的空间名 **
*****
mymac   .macro
mylab?  nop
        B mylab?, UNC
        .endm
*****
** 宏调用                                                 **
*****
        mymac
*****
1** 引用 mylab 的第三次定义宏调用不复位局部标号的定义    **
*****
        B mylab?, UNC
*****
**改变段，允许第五次定义 mylab                           **
*****
```

```

        .sect "Sect_One"
        nop
mylab?  .word 0
        nop
        nop
        B mylab?, UNC

*****
** .newblock 伪指令允许第六次定义 mylab                      **
*****

        .newblock
mylab?  .word 0
        nop
        nop
        B mylab?, UNC

```

3.8.3 符号常量

符号常量可以设置为一个常数。用户可以用符号常量代替常数使用。

伪指令 `.set` 和 `.struct`、`.tag`、`.endstruct` 使用户可以把常数赋给符号名。符号常量不能被重新定义。下面的例子说明了这些伪指令的用法：

```

K          .set 1024          ; 常量定义
maxbuf     .set 2*K
item       .struct            ; item 结构体定义
value      .int                ; value 偏移量为 0
delta      .int                ; delta 偏移量为 4
i_len      .endstruct         ; item 长度为 8
array      .tag item
           .bss array, i_len*K ; 定义数组 K "items"
           .text
           MOV array, delta, AR1
                                   ; 访问数组 .delta

```

汇编器也有一些预定义的符号常量，这些将在第3.8.5节预定义符号常量中讨论。

3.8.4 定义符号常量（-d 选项）

`-d` 选项使常数与一个符号等同，然后在汇编源程序中这个符号可用来代替常数使用。`-d` 选项的格式是：

```
asm2000 -v28 -dname=[value]
```

`name` 是用户想要定义的符号名字。`value` 是用户想要赋予符号的数。如果 `value` 被忽略，那么符号将被设置为 1。

一旦用 `-d` 选项定义了一个名字，这个符号就可以代替一个常数或是定义明确的表达式，另外可用汇编伪指令和汇编指令取消定义的符号。例如，在命令行可以输入：

```
asm2000 -v28 -dsym1=1 -dsym2=2 -dsym3=3 -dsym4=4 value.asm
```

自从用户为SYM1、SYM2、SYM3分配值之后，就可以在源代码中使用它们。例3.3 value.asm文件说明了如何使用没有明确定义的符号。

在汇编源程序中，可以使用以下伪指令去测试被-d选项定义的符号。

测试类型	伪指令的用法
Existence	<code>.if \$isdefed ("name")</code>
Nonexistence	<code>.if \$isdefed ("name") = 0</code>
Equal to value	<code>.if name = value</code>
Not equal to value	<code>.if name != value</code>

\$isdefed 内嵌函数的自变量必须包含在引号里面。引号使这个自变量按照字面被直译而不是作为一个置换符。

例 3.3 在命令行中使用定义过的符号常量

```
If_4:  .if      SYM4 = SYM2 * SYM2
        .byte    SYM4                ;等于值
        .else
        .byte    SYM2 * SYM2        ;不等于值
        .endif

IF_5:  .if      SYM1 <= 10
        .byte    10                  ;小于或等于
        .else
        .byte    SYM1                ;大于
        .endif

IF_6:  .if      SYM3 * SYM2 != SYM4 + SYM2
        .byte    SYM3 * SYM2        ;不等于值
        .else
        .byte    SYM4 + SYM4        ;等于值
        .endif

IF_7:  .if      SYM1 = SYM2
        .byte    SYM1
        .elseif SYM2 + SYM3 = 5
        .byte    SYM2 + SYM3
        .endif
```

3.8.5 预定义符号常量

汇编器有几个预定义的符号，它们是：

- \$ 美元符。表示段程序计数器（SPC）的当前值。\$是可重新定位的符号。
- 处理器符号：

符号名称	描 述
TMS320C2700	对 C27x 时设置为 1 (–v27 开关)，其他为 0
.TMS320C2800	对 C28x 时设置为 1 (–v28 开关)，其他为 0
_LARGE_MODEL	对大模式时设置为 1 (–mf 开关)，其他为 0

□ CPU控制寄存器:

寄 存 器	描 述
ACC/AH, AL	累加器/累加器高, 累加器低
DBGIEG	调试中断使能寄存器
DP	数据页指针
IER	中断使能寄存器
IFR	中断标志寄存器
P/PH, PL	乘积寄存器/乘积寄存器高, 乘积寄存器低
ST0	状态寄存器 0
ST1	状态寄存器 1
SP	堆栈指针寄存器
TH	被乘数高寄存器-T 寄存器的别名
XAR0/AR0H, AR0	辅助寄存器 0/辅助寄存器 0 高, 辅助寄存器 0 低
XAR1/AR1H, AR1	辅助寄存器 1/辅助寄存器 1 高, 辅助寄存器 1 低
XAR2/AR2H, AR2	辅助寄存器 2/辅助寄存器 2 高, 辅助寄存器 2 低
XAR3/AR3H, AR3	辅助寄存器 3/辅助寄存器 3 高, 辅助寄存器 3 低
XAR4/AR4H, AR4	辅助寄存器 4/辅助寄存器 4 高, 辅助寄存器 4 低
XAR5/AR5H, AR5	辅助寄存器 5/辅助寄存器 5 高, /辅助寄存器 5 低
XAR6/AR6H, AR6	辅助寄存器 6/辅助寄存器 6 高, /辅助寄存器 6 低
XAR7/AR7H, AR7	辅助寄存器 7/辅助寄存器 7 高, /辅助寄存器 7 低
XT/T, TL	被乘数寄存器/被乘数寄存器高, 被乘数寄存器低

控制寄存器可用大写字母也可用小写字母。如，IER可以用ier输入。

3.8.6 置换符号

可以把一个字符串的值（变量）赋予符号。这使用户能通过让字符串等同于符号名字而为字符串取一个别名。代表字符串的符号被称为置换符号。当汇编器遇到置换符号时，符号名用其对应的字符串值代替。与字符常量不同的是，置换符号可以被重新定义。

在程序中的任何地方，字符串可以被赋予置换符号。例如：

```
.asg "AR1", myReg           ;寄存器 AR1
.asg "**+XAR2 [2]", ARG1     ;第一个 ARG
.asg "**+XAR2 [1]", ARG2     ;第二个 ARG
```

因为宏参数实际上是被赋予宏自变量的置换符号，所以当用户正在使用宏时，置换符号就显得很重要。下面的代码说明了如何在宏中使用置换符号：

```
add2      .macro A, B          ; add2 宏定义
```

```

MOV AL, A
ADD AL, B
.endm
*add2 invocation
add2 LOC1, LOC2      ; 添加“LOC1”变量到第二变量“LOC2”
MOV AL, LOC1
ADD AL, LOC2

```

关于宏的详细介绍请参见第5章宏语言。

3.9 表 达 式

表达式是用算术运算符隔开的常量、符号或常量和符号串。表达式的值为32位，范围是-2147483648~ +2147483647，无符号数的范围是0~4294967295。影响表达式的运算次序的三个主要规则是：

- 圆括号：圆括号中的表达式总是最先被运算。
 $8/(4/2)=4$ ，但 $8/4/2=1$ 。
 不能用大括号（{ }）或中括号（[]）代替圆括号。
- 优先级组：在表3.1中把运算符分为了9个优先级组。当圆括号不能决定表达式的运算次序时，优先级高的运算符首先被执行。
 $8+4/2=10$ （首先执行 $4/2$ ）。
- 自左至右运算：当圆括号和优先级不能决定表达式的运算顺序的时候，表达式从左至右求值，优先级中的第一组（见表3.1）例外，它们是自右至左求值。
 $8/4*2=4$ ，但 $8/(4*2)=1$ 。

3.9.1 运算符

注意：与其他 TMS320 汇编器优先级的区别

- 有些 TMS320 汇编器使用与 TMS320C28x 不同的优先级，因此相同的代码可能产生不同的结果。TMS320C28x 使用与 C 语言程序相同的优先级。
- 当使用 -v28 -m20 调用 asm2000 时，汇编器接收 C2xlp 的源代码。程序员为 C2xlp 写代码时应该先假定它们与 C28x 汇编器具有不同的优先级。因此当使用 -v28 -m20 -mw 选项时，C28x 汇编器若遇到像 $a+b<<c$ 这样的表达式的时候会提出警告。

表3.1列出了在表达式中可用的运算符，它们按优先级分组列出。

表 3.1 表达式中的运算符的优先级顺序

优 先 级	运 算 符	描 述
1	+	一元的加 1
	-	一元的减 1

续表

优 先 级	运 算 符	描 述
	~	按位取反
	!	逻辑非
2	*	乘
	/	除
	%	取模
3	+	加
	-	减
4	<<	左移
	>>	右移
5	<	小于
	<=	不大于
	>	大于
	>=	不小于
6	=	等于
	!=	不等于
7	&	位与
8	^	位异或
9		位或

表注：第一组运算符从右至左求值，所有其他运算符从左至右求值。

3.9.2 表达式的上溢和下溢

在汇编过程中执行算术运算时，汇编器检查上溢和下溢。只要发生上溢或下溢，汇编器发出 value truncated（数被舍入）警告。而数被截断时汇编器会提出警告。汇编器在乘法中不检查上溢或下溢。

3.9.3 定义明确的表达式

某些汇编器指令需要定义明确（Well-Defined）的表达式作为操作数。定义明确的表达式只包括符号和汇编编译过程使用的（assembly-time）常量。在表达式中遇到它们之前，这些符号和汇编编译过程使用的常量已被定义。定义明确的表达式求值必须是绝对值。

下面是一个定义明确的表达式的实例：

0x1000+X

其中，X已在前面定义为绝对符号。

3.9.4 条件表达式

汇编器支持在表达式中使用关系运算符。它们在条件汇编时特别有用。关系运算符包括以下这些：

= 等于	!= 不等于
< 小于	<= 不大于
> 大于	>= 不小于

如果条件表达式为真，那么条件表达式求值得到 1；如果条件为假，那么其值为 0。它们仅可用于等效类型的操作数，即绝对数可以和绝对数比较，但是绝对数不能和可重定位的值相比较。

3.9.5 合法的表达式

除了下文所说的特例之外，不限制表达式、常量、内部定义的符号以及外部定义的符号之间进行组合运算。当一个表达式包含一个或多个可重定位符号或者在汇编时不能确定值时，汇编器就在目标文件中编码一个可重定位的表达式，然后通过链接器对表达式赋值。如果表达式的最后值超出为它保留的字节空间，链接器就会报告错误信息，详见 2.4 节。

当使用寄存器进行相对寻址方式时，中括号或圆括号中的表达式必须是定义明确的表达式，详见 3.9.3 节。例如：

`*+XA4[7]`

下面是使用可重定位和绝对符号表达式的例。这些例使用了在同一段中定义的 4 个符号。

```

                .global extern_1    ; 在外部模块中定义
intern_1:      .word '10'          ; 可重定位的，定义在当前模块中
LAB1:          .set 2              ; LAB1 = 2
intern_2       ; 可重定位的，定义在当前模块中
intern_3       ; 可重定位的，定义在当前模块中

```

例1

此例中使用了绝对符号 LAB1，它上面被定义为具体数 2。第一条语句把数 51 装入累加器。第二条语句把数 27 装入累加器。

```

MOV AL, #LAB1 + ((4+3) * 7),    ; ACC = 51
MOV AL, #LAB1 + 4 + (3*7),      ; ACC = 27

```

例2

以下所有的语句都是合法的：

```

MOV @(extern_1 - 10), AL        ; 合法的
MOV @(10-extern_1), AL         ; 合法的
MOV @(-(intern_1)), AL         ; 合法的
MOV @(extern_1/10), AL         ; / 不是附加操作
MOV @(intern_1 + extern_1), ACC ; 多次重定位

```

例3

下面的第一条语句是合法的；虽然 intern_1 和 intern_2 是可重定位的，但是由于它们在同一段中，所以它们的差值是固定的。从另一个中减去一个可重定位符号把表达式简化为可重定位符号+绝对数。因为两个可重定位符号的和不是绝对数，所以第二条语句是非法的。

```

MOV (intern_1 - intern_2 + extern_1), ACC ; 合法的

```


MOV (intern_1 + intern_2 + extern_1), ACC ; 非法的

例4

可重定位符号在表达式中的位置对表达式的求值非常重要。虽然下面的语句与上面例中的第一条语句相类似，但是由于从左到右的运算符优先次序，汇编器试图把intern_1加到extern_1上，所以此语句是非法的。

MOV (intern_1 + extern_1 - intern_2), ACC ;非法的

3.10 内 嵌 函 数

汇编器支持大量的内嵌函数。内嵌函数总是返回一个值，它们可使用在条件汇编中或任何可以使用常量的地方。

在表3.2中，x、y和z是浮点类型，n是整型。函数\$cvf、\$int和\$sgn返回一个整数，其他函数返回浮点数。三角函数中的角度以弧度表示。

表 3.2 内嵌函数

函 数	描 述
\$trunc(x)	返回 $\cos^{-1}(x)$, $x \in [-1, 1]$
\$asin(x)	返回 $\sin^{-1}(x)$, $x \in [-1, 1]$
\$atan(x)	返回 $\tan^{-1}(x)$
\$atan2(x, y)	返回 $\tan^{-1}(y/x)$
\$ceil(x)	返回不小于 x 的最小整数，x 可以是浮点数
\$cos(x)	返回 x 的余弦
\$cosh(x)	返回 x 的双曲余弦
\$cvf(n)	把整数转化为浮点数
\$cvi(x)	把浮点数转化为整数，返回一个整数
\$exp(x)	返回指数 e^x
\$fabs(x)	返回 x 的绝对值
\$floor(x)	返回不大于 x 的最大整数，作为浮点数
\$fmod(x, y)	返回 x/y 的浮点型余数
\$int(x)	如果 x 是整数返回 1，否则返回 0。返回整型的
\$ldexp(x, n)	返回 x 乘以 2 的整数次幂的值即 $x \times 2^n$ 的值
\$log(x)	返回 x 的自然对数 $\ln(x)$, $x > 0$
\$log10(x)	返回 x 的以 10 为底的对数， $\log_{10}(x)$, $x > 0$
\$max(x, y, ..., z)	返回参数列表中的最大值
\$min(x, y, ..., z)	返回参数列表中的最小值
\$pow(x, y)	返回 x^y 的值
\$round(x)	对 x 取整
\$sgn(x)	返回 x 的符号类型，正数返回 1，0 返回 0，负数返回 -1，返回的值为整型
\$sin(x)	返回 x 的正弦

续表

函 数	描 述
\$sinh(x)	返回 x 的双曲正弦
\$sqrt(x)	返回 x 的平方根
\$tan(x)	返回 x 的正切
\$tanh(x)	返回 x 的双曲正切
\$trunc(x)	返回 x 被截断的值（向 0）

内嵌函数将在 5.3.2 节讨论。

3.11 源程序列表

源程序列表显示源程序语句及它们产生的目标代码。用-l（小写字母L）选项调用汇编器可以获得此列表文件。

在每页源程序列表文件的顶部是两个标识行，一行空行和一个标题行。由伪指令.title 提供的标题在标题行显示，标题的右边是页码。如果用户未使用.title伪指令，标题行显示源程序的名字。汇编器在标题行的下面插入一行空格。

源程序中的每一行可在列表文件中产生一行，它显示源程序语句编号、SPC值、被汇编的目标代码以及源程序语句。例3.4显示了一个实际的列表文件中的各项。

例 3.4 部分汇编器程序列表

```
1          addl    .macro S1, S2, S3, S4
2
3          MOV AL, S1
4          ADD AL, S2
5          ADD AL, S3
6          ADD AL, S4
7          .endm
8
9          .global c1, c2, c3, c4
10         .global _main
11
12         0001 c1          .set 1
13         0002 c2          .set 2
14         0003 c3          .set 3
15         0004 c4          .set 4
16
17         000000          _main:
18         000000          addl #c1, #c2, #c3, #c4
1
1         000000 9A01      MOV AL, #c1
1         000001 9C02      ADD AL, #c2
1         000002 9C03      ADD AL, #c3
1         000003 9C04      ADD AL, #c4
```

19

20

.end

域 1 域 2 域 3

域 4

域1：源程序语句编号**行号**

源程序编号是一个十进制数。汇编器在源程序文件中遇到源程序行时对它编号，某些语句使程序行计数增加但并不在列表文件中列出。（例如，.title 语句和跟在.nolist 之后的语句未被列出）。两个连续程序行编号的差值表示未被列出的源程序中插入的语句数。

包含文件字母

汇编器可以把一个字母放在一行的前面，该字母表示此行从包含文件中被汇编。

嵌套层数

汇编器可以把一个数字放在一行的前面，该数字表示宏扩展或循环块的嵌套层数。

域2：段程序计数器

此域包含十六进制的 SPC 值。所有的段（.text、.data、.bss 以及命名段）保持单独的段程序计数器。有些伪指令不影响 SPC 并使该行为空白。

域3：目标代码

此域包含目标代码的十六进制表示值。所有机器指令和伪指令使用此域来列出目标代码。此域也对这一行源代码的操作数说明了与之相关的重定位类型。如果该行有多个可重定位的操作数，那么此列仅表明第一个操作数的重定位类型。下面列出了在此域能够出现的字符以及它们对应的可重定位类型：

- ! 未定义的外部引用
- ' .text可重定位
- + .sect可重定位
- " .data可重定位
- .bss, .usect可重定位
- % 可重定位的表达式

域4：源语句域

此域包含被汇编器扫描的源程序语句。该域的大小取决于源程序语句的大小。

3.12 交叉引用列表

交叉引用列表显示的是符号及其定义。为了获得交叉引用列表，可以用-x选项调用汇编器或使用.option伪指令生成交叉引用列表。汇编器把交叉引用列表添加到源程序列表的后面。

例 3.5 汇编器的一个交叉引用列表

LABEL	VALUE	DEFN	REF
-------	-------	------	-----

.TMS320C2800	00000001	0	
_func	00000000'	18	
var1	00000000-	4	17
var2	00000004-	5	18

交叉引用列表包含以下几列：

- LABEL**
- 该列包含所有在汇编期间被定义或引用的符号。
- VALUE**
- 该列包含一个分配给符号的8位十六进制的数字或说明符号属性的名字。
数可以后随说明符号属性的字符。表3.3列出了这些字符和名字。
- DEFN**
- 该列包含定义该符号的语句号。对于未定义的符号，此列为空白。
- REF**
- 该列是引用该字符的语句号。该列中的空格表示符号从未被使用。

表 3.3 符号属性

字符或名字	意 义
REF	外部引用（全局符号）
UNDF	未定义
,	在.text 段定义的符号
"	在.data 段定义的符号
+	在.sect 段定义的符号
-	在.bss 段定义的符号

3.13 灵巧的编码

为了提高效率，在可能的情况下汇编器尽量减小指令的长度。例如，若偏移量是8位，则两个字的长转移指令可以转化为一个字的短转移指令。表3.4列出了可互相替换的指令和发生的变化。

表 3.4 灵巧的编码

指 令	编 码
MOV AX, #8bit	MOVB AX, #8Bit
ADD AX, #8BitSigned	ADDB AX, #8BitSigned
CMP AX, #8Bit	CMPB AX, #8Bit
ADD ACC, #8Bit	ADDB ACC, #8Bit
SUB ACC, #8Bit	SUBB ACC, #8Bit
AND AX, #8BitMask	ANDB AX, #8BitMask
OR AX, #8BitMask	ORB AX, #8BitMask
XOR AX, #8BitMask	XORB AX, #8BitMask
B 8BitOffset, cond	SB 8BitOffset, cond
LB 8BitOffset, cond	SB 8BitOffset, cond

续表

指 令	编 码
MOVH <i>loc</i> , ACC << 0	MOV <i>loc</i> , AH
MOV <i>loc</i> , ACC << 0	MOV <i>loc</i> , AL
MOVL XAR _{<i>n</i>} , #8Bit	MOVB XAR _{<i>n</i>} , #8Bit

汇编器在灵巧的编码中也可以自动改变指令格式。例如，把累加器的值压入堆栈，可以使用 MOV *SP++, ACC。由于对于这种操作汇编器自动使用 PUSH ACC，因此汇编器接收 PUSH ACC 而且通过灵巧的编码，把它变为 MOV *SP++, ACC。表 3.5 列出了哪些指令可以在灵巧编码中使用及这些指令的变化。

表 3.5 自动灵巧的编码

指 令	编 码
MOV P, #0	MPY P, T, #0
SUB <i>loc</i> , #16BitSigned	ADD <i>loc</i> , #-16BitSigned
ADDB SP, #-7Bit	SUBB SP, #7Bit
ADDB <i>aux</i> , #-7Bit	SUBB <i>aux</i> , #7Bit
SUBB AX, #8BitSigned	ADDB AX, #-8BitSigned
PUSH IER	MOV *SP++, IER
POP IER	MOV IER, *--SP
PUSH ACC	MOV *SP++, ACC
POP ACC	MOV ACC, *--SP
PUSH XAR _{<i>n</i>}	MOV *SP++, XAR _{<i>n</i>}
POP XAR _{<i>n</i>}	MOV XAR _{<i>n</i>} , *--SP
PUSH #16bit	MOV *SP++, #16bit
MPY ACC, T, #8Bit	MPYB ACC, T, #8Bit

3.14 汇编变量的 C 类型符号调试

当用户使用 -mg 选项时，汇编器将生成汇编程序的调试信息。汇编器为在汇编源代码中用数据伪指令定义的符号，生成 C 类型的符号调试信息。以支持基本的 C 类型、结构体、数组。用户就能够告知汇编器如何用基本的类型信息翻译 C 类型变量的符号。

当按以下方式使用 -mg 选项时，数据伪指令已经被修改用于生成调试信息。

- 初始化数据的数据伪指令 汇编器输出用 .byte、.field、.float、.int 或 .long 等伪指令进行数据初始化的调试信息。在下面的例子中，汇编器忽略调试信息，把 init_sys 作为 C 类型的整数。

```
int_sym      int      10h
```

多个初始值将被作为伪指令指定类型的数组。下面例子是被编译成数组的4个数以及所生成的调试信息。

```
int_sym      .int      10h, 11h, 12h, 13h
```

要生成符号信息，必须用数据伪指令命名一个标号。比较下面所示的第一行与第二行的代码：

```
int_sym      .int      10h
.int         11h --> 不会生成调试信息
```

- **未初始化数据的数据伪指令** `.bss` 和 `.usect` 伪指令接受一类命名作为可选的第5个操作数。这类操作数用来为 `.bss` 伪指令定义的符号生成适当的调试信息。与前面所提到的初始化的数据伪指令生成的调试信息相似。

```
.bss int_sym, 1,1,0,int
```

这类操作数可以是下列中的一种：

CHAR	INT	SCHAR	UINT
DOUBLE	LDOUBLE	SHORT	ULONG
FLOAT	LONG	UCHAR	USHORT

在下面的例子中，参数 `int_sym` 被作为4个整数的数组：

```
.bss int_sym, 4,1,0,int
```

声明的大小必须是规定类型的倍数。如果没有声明类型，不会提出警告。在下列代码中，3不是 `long` 型数据长度的倍数，因此会提出警告。

```
.bss double_sym, 3,1,0,long
```

- **汇编语言结构体的调试信息** 汇编器也会输出在汇编语言中定义的结构体符号信息。下面是一个结构的例子：

```
structlab    .struct
mem1         .int
mem2         .int
struct_len   .endstruct
struct1      .tag structlab
               .bss struct1, 2, 1, 0, structlab
```

在这个例子中，把 `struct1` 作为 C 的结构体生成调试信息：

```
struct struct1{
    int mem1;
    int mem2;
};
```

如果用 `.bss` 伪指令声明的大小是结构体类型大小的倍数，汇编器将输出结构体数组。用非初始化的数据伪指令声明的大小不是结构体类型大小的倍数，将产生警告。

下面的例子说明被成员类型约束的队列：

```
.bss struct1,struct_len * 3, 1, 0, structlab
```

3.15 TMS320C28x 汇编器的模式

TMS320C28x的目标代码与TMS320C27x的目标代码兼容，源代码与TMS320C2xx（C2xlp）兼容。为了支持与C27x和C2xlp向上兼容，C28x支持4种操作模式。这4种模式通过以下选项控制：

-v27	C27x 目标模式
-v28	C28x 目标模式
-v28 -m27	C28x 目标——兼容 C27x 句法模式
-v28 -m20	C28x 目标——兼容 C2xlp 句法模式

-m27选项和-m20选项已暗含-m28选项，因此在这些选项中不需要声明-m28。当声明了多个选项时，汇编器将使用第一种选项并忽略其他选项。例如在C28x目标模式中使用命令：‘asm2000 -v28 -v27’调用汇编器时，汇编器将忽略-v27开关并提出以下警告：

>> Version already specified. -v27 is ignored

由于-m27选项和-m20选项已暗含-m28选项，因此命令‘asm2000 -m20 -v27’与‘asm2000 -v28 -m20 -v27’是等价的，汇编器忽略-v27开关并产生上面的警告。

-m27和-m20开关不能混合使用。命令‘asm2000 -m20 -m27’将产生下面的警告：

>> Target option -m27 cannot be combined with -m20; -m27 is ignored

*TMS320C28x CPU and Instruction Set Reference Guide*详细介绍了C28x支持的目标模式和寻址方式。

3.15.1 C27x 目标模式

在C27x模式下，能够把C27x代码转化为C28x代码并在C28x处理器上运行。因此，在此模式下，C28x汇编器实际上成了支持以下非C27x指令的C27x汇编器。表3.6所列指令用来改变处理器目标模式和寻址方式，详细资料见*TMS320C28x CPU and Instruction Set Reference Guide*。

表 3.6 C27x 目标模式支持的非 TMS320C27x 的指令

指 令		描 述
SETC	OBJMODE	将状态寄存器的 OBJMODE 位置 1，处理器运行在 C28x 目标模式下
CLRC	OBJMODE	将状态寄存器的 OBJMODE 位清 0，处理器运行在 C27x 目标模式下
C28OBJ		与 SETC OBJMODE 同
C27OBJ		与 CLRC OBJMODE 同
SETC	AMODE	将状态寄存器的 AMODE 位置 1，处理器支持 C2xlp 寻址方式
CLRC	AMODB	将状态寄存器的 AMODE 位清 0，处理器支持 C28x 寻址方式

续表

指 令		描 述
LPADDR		与 SETC AMODE 同
C28ADDR		与 CLRC AMODE 同
SETC	MOM1MAP	将状态寄存器的 MOM1MAP 位置 1
CLRC	MOM1MAP	将状态寄存器的 MOM1MAP 清 0
SETC	CNF	将状态寄存器的 CNF 位 (C2xlp 映射模式位) 置 1
CLRC	CNF	将状态寄存器的 CNF 位 (C2xlp 映射模式位) 清 0
SET	XF	将状态寄存器的 XF 位置 1
CLRC	XF	将状态寄存器的 XF 位清 0

在此模式下运作, 当使用非兼容的C27x格式或指令, C28x汇编器将产生错误。下面是在此模式下使用了非法指令的例子:

```

FLIP    AL                ; 在 C27x 模式中不支持 C28x 指令
MOV     AL, *XAR0++       ; 在 C27x 模式中 *XAR0++ 是合法的寻址方式

```

3.15.2 C28x 目标模式

C28x 目标模式支持所有C28x的指令并生成C28x目标代码。新用户设置C28x处理器应使用汇编器的这种模式。当使用了旧的C27x句法时会出错误。下面是这种模式下的非法指令:

```

MOV AL, *AR0++           ; 在 C28x 模式中 *XAR0++ 是合法的寻址方式

```

3.15.3 C28x 目标——兼容 C27x 的语法模式

此模式支持所有C28x指令和C27x指令及寻址方式。该模式生成C28x目标代码。例如: 此模式接受指令语法 'MOV AL, *AR0++', 并把它变成 'MOV AL, *XAR0++'。虽然这种模式支持C27x的语法, 但为了鼓励用户从C27x的语法转移到C28x的语法上来, 如果使用了C27x的语法, 汇编器将提出警告。下面是使用了MOV AL, *AR0+ 指令产生的警告:

```

WARNING! at line 1: [W0000] Full XAR register is modified

```

3.15.4 C28x 目标——兼容 C2xlp 的语法模式

此模式支持所有C28x的指令, 并生成C28x的目标代码, 它也支持C2xlp指令的语法。C28x处理器尽可能的包含了能与C2xlp处理器向下兼容的特征和指令。为了使C28x的源代码与C2xlp的源代码兼容, 译码时汇编器接受C2xlp的指令并把它们当作C28x指令。

详细情况见 *TMS320C2xx User's Guide*。

此模式不支持C27x的语法并将产生错误。不兼容的C2xlp指令也会使汇编器产生错误。

下面是这种模式下的非法指令：

```
MOV AL, *AR0++      ; 在 C28x 模式中*AR0++是合法的寻址方式
TRAP                ; 和 C2XLP 指令不兼容
```

这种模式是假定 LP 寻址方式兼容 (AMODE=1) 而且一页数据是 128 字的长度。

详细情况见 *TMS320C28x CPU and Instruction Set Reference Guide*。

在这种模式下，在文件中 C28x 和 C2xlp 源代码可以随意混合使用，如下所示：

```
; C2xlp 源代码
LDP #VarA
LACL VarA
LAR AR0, *, AR2
SACL *+
|
LC FuncA
|
; 用 LP 寻址 (AMODE = 1) 的 C28x 源代码
FuncA:
MOV DP, #VarB
MOV AL, @@VarB
MOVL XAR0, *XAR0++
MOV *XAR2++, AL
LRET
```

当使用 -mw 开关调用 C28x 汇编器的时候，在下面的情况下将执行附加检查：

- C1x/C2x/C2xx/C5x 汇编器接受以 0 开头的十进制整数，即 010 被作为 10 而不是 8。C28x 汇编器将把以 0 开头的数当作八进制常数。可能有以 0 开头的十进制数的 C2xlp 的汇编代码。当用户使用 C28x 汇编器汇编这些文件的时候，汇编器将把它们作为八进制的常数，因而不会得到预期结果。当使用 -m20 -mw 调用汇编器，这些常数被作为八进制数时，汇编器将提出警告。

下例分析了 -m20 -mw 选项的列表：

```
1 00000000 FF20 lacc #023
"octal.asm", WARNING! at line 1: [W0000] Constant parsed as an octal number
00000001 0013
```

- C1x/C2x/C2xx/C5 汇编器使用了一个操作数优先级不同的表达式。对于 C1x/C2x/C2xx/C5 汇编器，移位运算符 (<< 以及 >>) 的优先级高于二进制的 + 和 - 运算符的优先级。C28x 汇编器遵从 C 语言程序的优先级次序，这与上面的次序恰好相反。在下面的情况下，C28x 汇编器将对使用的优先级提出警告。
 - 使用了 m20 -mw 选项。
 - 源代码含有包括二进制加操作符 (+ 和 -) 的表达式和移位操作符 (<< 和 >>)。
 - 优先级没用圆括号约束。

分析下面用了 -m20 -mw 选项的列表：

```
1 00000000 FF20 lacc #(3 + 4 << 2) ; Warning generated
```

```
"pre.asm", WARNING! at line 1: [W9999] The binary + and - operators have  
higher  
precedence than the shift operators  
00000001 001C  
2 00000002 FF20 lacc #((3 + 4) << 2) ; NO warning  
00000003 001C
```





第 4 章 汇编伪指令

汇编伪指令为程序提供数据并控制汇编过程。汇编伪指令可完成如下工作：

- 将代码和数据汇编到特定的段中
- 在存储器中为未初始化变量保留空间
- 控制列表的显示格式
- 初始化存储器
- 汇编条件块
- 定义全局变量
- 规定汇编器可以从中提取宏的库
- 检查符号调试信息

这一章分成两部分：第一部分（4.1 节到 4.11 节）按功能介绍汇编伪指令，第二部分（4.12 节）是按字母排序的伪指令参考信息。

4.1 伪指令简介

除了这里提到的汇编伪指令，TMS320C28x 软件工具还支持以下伪指令：

- **宏伪指令** 汇编器对宏使用的伪指令。宏伪指令在第 5 章宏语言中讨论，这一章

不讨论。

- 符号调试伪指令 C/C++编译器使用伪指令做符号调试。与其他伪指令不同，大多数汇编语言程序中不使用这种伪指令。这些伪指令将在附录 B 符号调试伪指令中讨论。

表 4.1 总结了汇编伪指令。

注意：包含伪指令的源语句可以有标号和注释。标号从第一列开始（除注释以外，惟一可以从第一列开始的是标号）。如果注释是某一行中的惟一元素，那注释前面必须有分号或星号。为了提高可读性，标号和注释不作为伪指令语法的一部分。

表 4.1 汇编伪指令一览表

(a) 定义段的伪指令

助记符和格式	描 述
.bss	在当前段保留指定数量的位
.bss symbol, size in words, [, blocking] [, alignment]	在.bss 段（未初始化数据段）内保留 size 个字
.data	汇编到.data 段(已初始化数据段)
.sect "section name"	汇编到一个已命名段(已初始化)
.text	汇编到.text 段(可执行代码段)
symbol .usect "section name", size in words[, blocking] [, alignment flag]	在一个已命名段(未初始化段) 内保留 size 个字
.sblock	为块指定段

(b) 初始化常量（数据、存储器）的伪指令

助记符和格式	描 述
.byte value1[, ..., valuen]	初始化当前段内一个或多个连续的字节
.field value[, size]	用 value 初始化 size 位（1~32）的域
.float value1[, ..., valuen]	初始化一个或多个 32 位 IEEE 单精度浮点型常数
.int value1[, ..., valuen]	初始化一个或多个 16 位整数
.long value1[, ..., valuen]	初始化一个或多个 32 位整数
.pstring	将字符串中的 8 位字符放入当前段
.space size	在当前段内保留 size 位；标号指向保留空间的起始处
.string {expr1!"string1"}[, ..., {expren!"stringn"}]	初始化一个或多个字符串
.word value1[, ..., valuen]	初始化一个或多个 16 位整数
.xfloat	将一个或多个浮点型常数用浮点型表示放入当前段
.xlong	将一个或多个 32 位值放到当前段的连续字中

(c) 调整段程序计数器（SPC）的伪指令

助记符和格式	描 述
.align [size in words]	把 SPC 调整到按 size in words（2 的整数次幂）指定的边界；默认值为 64 个字的页边界

(d) 控制输出列表格式的伪指令

助记符和格式	描 述
.drlist	允许列出所有伪指令行（默认设置）
.drnolist	禁止列出某些伪指令行
.fclist	允许列出条件为假代码块（默认设置）
.fcnolist	禁止列出条件为假代码块
.length page length	设置源程序列表的页长度
.list	重新启动源程序列表
.mlist	允许宏和循环块列表（默认设置）
.mnolist	禁止宏和循环块列表
.nolist	停止源程序列表
.option option1[, option2, ...]	输出列表选项，提供的选项有 B, L, M, R, T, W, X
.page	在源程序列表中按页显示
.sslist	允许扩展替换符号列表
.ssnolist	禁止扩展替换符号列表（默认设置）
.tab size	设置制表符（tab）的大小
.title "string"	在列表页头部打印标题
width page width	设置列表的页宽度

(e) 引用其他文件的伪指令

助记符和格式	描 述
.copy ["filename"]	从另一文件复制源语句
.def symbol1[, ..., symbol n]	标识一个或多个在当前模块定义并在其他模块中使用的符号
.global symbol1[, ..., symbol n]	标识一个或多个全局（外部）符号
.include ["filename"]	包含另一个文件的源语句
.mlib ["filename"]	定义宏库
.ref symbol1[, ..., symbol n]	标识一个或多个在其他模块定义在本模块使用的符号

(f) 控制条件汇编的伪指令

助记符和格式	描 述
.break [well-defined expression]	如果条件为真，那么结束.loop 汇编 当使用.loop 结构时，.break 结构为可选
.else	如果.if 条件为假，那么汇编代码块 当使用.if 结构时，.else 结构为可选
.elseif well-defined expression	当.if 条件为假且.elseif 条件为真，则汇编代码块 当使用.if 结构时，.elseif 结构为可选
.endif	结束.if 代码块
.endloop	结束.loop 代码块
.if well-defined expression	如果条件为真，那么汇编代码块
.loop [well-defined expression]	开始一个循环；循环次数由表达式决定

(g) 汇编过程中使用的定义符号的伪指令

助记符和格式	描 述
.asg [""]character string[""], substitution symbol	把字符串赋予替代符号 (substitution symbol)
.endstruct	结束结构的定义
.eval well-defined expression substitution symbol	根据数字替代符号完成运算
.label symbol	在段内定义一个装载时重定位标号
symbol .set value	使 symbol 与 value 相等
.struct	开始结构的定义
.tag	将结构特性赋给一个标号

(h) 汇编模式伪指令

助记符和格式	描 述
.c28_amode	在 C28x 目标模式下汇编
.lp_amode	在 C28x 目标模式下汇编, 但认可 C2xLP

(i) 其他伪指令

助记符和格式	描 述
.clink ["section name"]	允许对当前或指定段条件链接
.emsg string	将把用户定义的错误信息发送到输出设备, 不生成.obj 文件
.end	结束程序
.mmsg string	将用户定义的信息发送到输出设备
.newblock	取消局部标号定义
.wmsg string	将用户定义的警告信息发送到输出设备

4.2 与 TMS320C1x/C2x/C2xx/C5x 汇编伪指令的兼容性

本节讨论 TMS320C28x 汇编伪指令与 TMS320C1x/C2x/C2xx/C5x 汇编伪指令的不同之处。

- C28x 的 **.float**, **.long** 伪指令自动定位 SPC, 使其指向偶数字边界, 而 C1x/C2x/C2xx/C5x 的伪指令不这样做。
- 如果没有参数, 对于 C28x 和 C1x/C2x/C2xx/C5x 汇编器的 **.align** 伪指令都将 SPC 调整到下一页的边界处。然而, C28x 的 **.align** 伪指令也可以接受常数参数, 此参数为 2 的整数次幂, 这个参数将 SPC 定位到指定的边界。对于 C1x/C2x/C2xx/C5x 汇编器 **.align** 伪指令不接受参数。
- C28x 的 **.field** 伪指令处理 1 到 32 位的数, C1x/C2x/C2xx/C5x 的汇编器处理 1 到 16 位的数。使用 C28x 汇编器时, 大于等于 16 位的目标码从字的边界开始存放, 并将最低位放在低地址处。

- C28x 的 .bss、.usect 伪指令有一个附加标志，称作 alignment flag，它说明定位在偶数字边界。C1x/C2x/C2xx/C5x 的 .bss、.usect 伪指令不使用这一标志。
- C28 的 .string 伪指令每个字只初始化一个字符；C1x/C2x/C2xx/C5x 的汇编器 .string 和 C28x 的 .pstring 将伪指令两个字符组合成一个字。
- 下述是 C28x 汇编器的新伪指令，C1x/C2x/C2xx/C5x 汇编器不支持这些伪指令：

伪 指 令	用 途
.xfloat	与 .float 相同，无自动定位
.xlong	与 .long 相同，无自动定位
.pstring	与 .string 相同，两个字符组合成一个字

- C1x/C2x/C2xx/C5x 汇编器支持 .mmregs 和 .port 伪指令，当用 -m20 选项调用 C28x 汇编器时，汇编器忽略这些伪指令，并发出警告——伪指令被忽略。C28x 汇编器不接受这些伪指令。

4.3 定义段的伪指令

这些伪指令把汇编语言程序的各部分与适当的段连接起来。

- .bss 伪指令在 .bss 段内为未初始化变量保留空间。
- .data 伪指令标识 .data 段内的代码部分。 .data 段通常包含已初始化数据。
- .sect 伪指令定义一个已初始化命名段，并将后面的代码和数据与这个段连接起来。一个用 .sect 定义的段可以包含代码或数据。
- .text 伪指令标识 .text 段内部分代码。 .text 段通常包含可执行代码。
- .usect 伪指令在未初始化命名段内保留空间。 .usect 伪指令与 .bss 伪指令相似，但是它允许用户在 .bss 段分开保留空间。

在第 2 章通用目标文件格式中详细讨论了 COFF 段。

例 4.1 说明了如何使用段伪指令将代码和数据同恰当的段连接起来。这是一个输出列表；第一列显示行号，第二列显示 SPC 值（每个段有自己的程序计数器，或称 SPC），当代码是第一次被放到一个段中时，它的 SPC 等于 0。当其他代码被汇编后，用户继续进入这个段进行汇编，这个段的 SPC 继续计数，就好像从未打断过一样。

例 4.1 中的伪指令执行下列任务：

.text	用数 1, 2, 3, 4, 5, 6, 7, 8 初始化字
.data	用数 9, 10, 11, 12, 13, 14, 15, 16 初始化字
.var_defs	用数 17 和 18 初始化字
.bss	保留 10 个字的空间
xy	保留 20 个字的空间

.bss 和 .usect 伪指令不结束当前段或者开始一个新的段；它们保留指定数量的空间，然

后汇编器恢复把代码或数据汇编到当前段中。

例 4.1 段伪指令

```

1          *****
2          * 开始汇编到.text 段中                      *
3          *****
4 000000          .text
5 000000 0001          .word  1, 2
   000001 0002
6 000002 0003          .word  3, 4
   000003 0004
7
8          *****
9          * 开始汇编到.data 段中                      *
10         *****
11 000000          .data
12 000000 0009          .word  9, 10
   000001 000A
13 000002 000B          .word  11, 12
   000003 000C
14
15         *****
16         * 开始汇编到指定的已初始化段, var_def      *
17         *****
18
19 000000          .sect  "var_defs"
20 000000 0011          .word  17, 18
   000001 0012
21
22         *****
23         * 继续汇编到.data 段中                      *
24         *****
25 000004          .data
26 000004 000D          .word  13, 14
   000005 000E
27 000000          .bss   sym, 19      ; 在.bss 中保留空间
28 000006 000F          .word  15, 16   ; 继续在.data 段中
   000007 0010
29
30         *****
31         * 继续汇编到.text 段中                      *
32         *****
33 000004          .text
34 000004 0005          .word  5, 6
   000005 0006
35 000000          usym          .usect  "xy", 20      ; 在 xy 中保留空间
36 000006 0007          .word  7, 8      ; 继续在.text 段中
   000007 0008

```


4.4 常数初始化伪指令

几种伪指令为当前段指定数据。

□ **.bes, .space** 伪指令在当前段中保留指定数目的位。汇编器用 0 填充这些保留位。用户可以用保留字数乘 16 来指定特定数目的位。

■ 当用户与 .space 使用标号时，它指向保留位的第一个字。

■ 当用户与 .bes 使用标号时，它指向保留位的最后一个字。

图 4.1 表明 .space, .bes 伪指令是如何保留空间的。

假设在此例中下列代码已汇编过：

1		** .space 和 .bes 伪指令			
2					
3	000000	0100		.word	100h,200h
	000001	0200			
4	000002		Res_1	.space	17
5	000004	000F		.word	15
6	000006		Res_2	.bes	20
7	000007	00BA		.byte	0BAh

Res_1 指向被 .space 保留的空间中第一个字，Res_2 指向被 .bes 保留的空间的最后一个字。

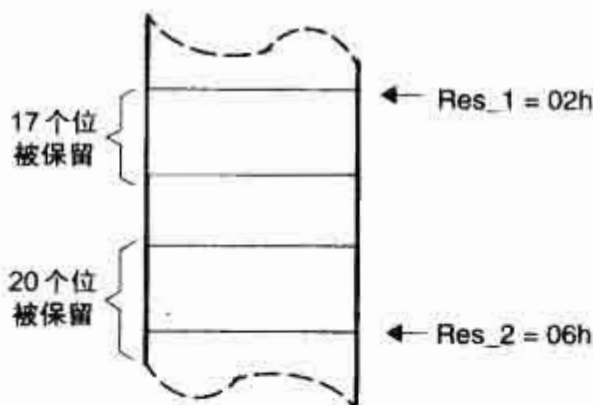


图 4.1 .space 和 .bes 伪指令

□ **.byte, .char** 伪指令将一个或多个 8 位的数放入当前段连续的字中。除了每个数的宽度被限制为 8 位以外，此伪指令与 .word 相似。

□ **.field** 伪指令把单个位域放入当前字规定的位数中。用户可以用 .field 伪指令把多个位域组装到单个字中；在字被填满之后，汇编器才会增加 SPC 的值。

图 4.2 显示了多个位域是如何组合成一个字的。

假设下列代码已汇编过，注意，SPC 不改变（几个位域组合成一个字）：

1	000000 0003	.field	3, 3
2	000000 0008	.field	8, 6
3	000000 0010	.field	16, 5

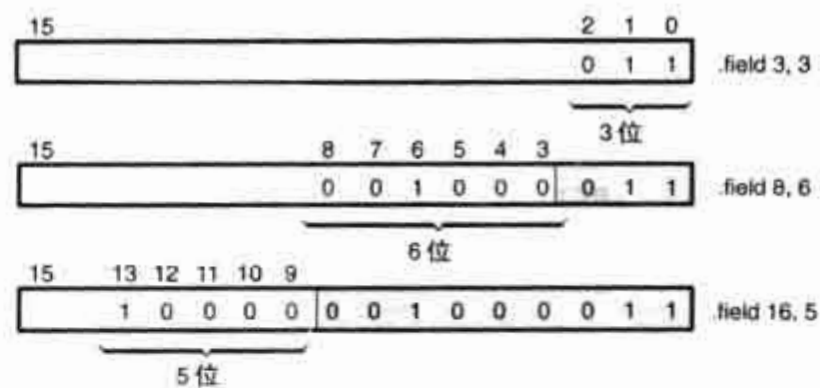


图 4.2 .field 伪指令

- **.float** 和 **.xfloat** 伪指令计算 IEEE 浮点数形式的单精度（32 位）浮点数的值，并将其存入当前段的一个字中。
- **.int** 和 **.word** 伪指令将一个或多个 16 位数放入当前段中连续的 16 位字中。
- **.long** 和 **.xlong** 伪指令将一个或多个 32 位数放入当前段中连续的 32 位的字中。
- **.string** 和 **xstring** 伪指令从一个或多个字符串中将 8 位的字符置入当前段中，每个字中放一个字符。这条伪指令与 **.byte** 相似。

注意：在 **.struct/.endstruct** 中使用常数初始化伪指令

当 **.byte**、**.int**、**.long**、**.word**、**.string**、**.float**、**.field** 伪指令是 **.struct/.endstruct** 的一部分时，它们不能初始化存储器；而只是定义成员的大小。

图4.3比较.byte, .word和.string伪指令的存储格式。

字	内容	代码
1	0 0 A B	.byte 0ABh
2	C D E F	.word 0CDEFh
3	C D E F	.long 089ABCDEFh
4	8 9 A B	
5	0 0 68	.string "help"
	h	
6	0 0 65	
	e	
7	0 0 6C	
	1	
8	0 0 70	
	p	

图 4.3 伪指令的存储格式

假设下列代码已汇编：

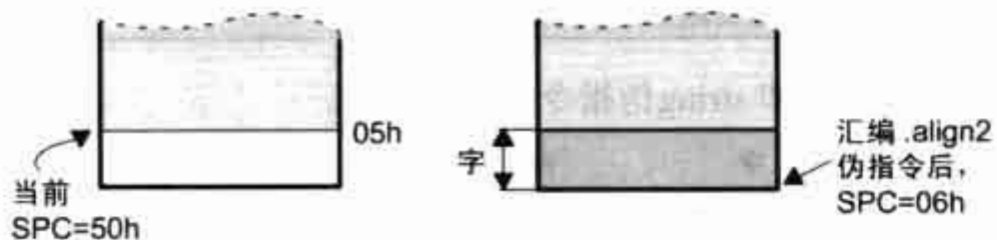
1	000000	00AB	.byte	0ABh
2	000001	CDEF	.word	0CDEFh
3	000002	CDEF	.long	089ABCDEFh
	000003	89AB		
4	000004	0068	.string	"help"
	000005	0065		
	000006	006C		
	000007	0070		

4.5 调准段程序计数器伪指令

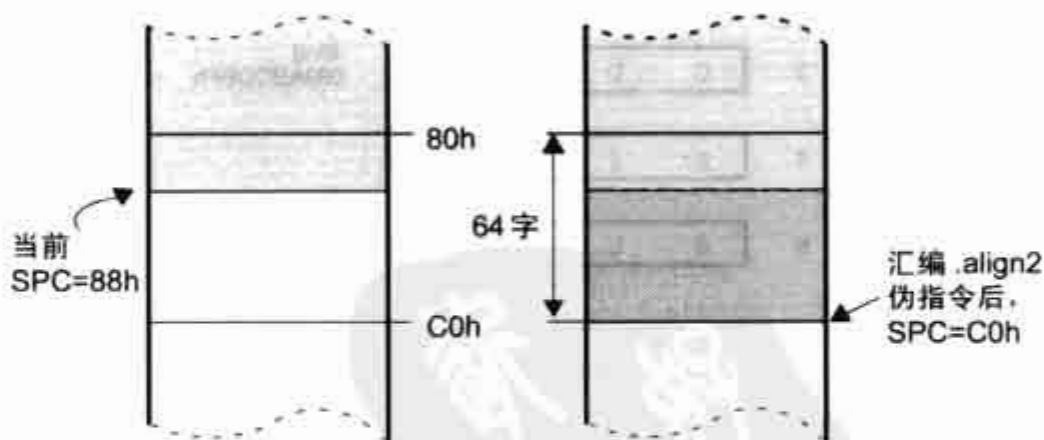
.align 伪指令将 SPC 调准到指定的字边界。这确保伪指令后的代码从该边界开始。如果 SPC 已经调准至所选择的边界，那么它的值就不再增加。**.align** 伪指令的操作数必须等于 1 到 65536 之间 2 的整数次幂。例如：

- 操作数
- 1 将 SPC 定位到字边界；
 - 2 将 SPC 定位到长字/偶数边界；
 - 64 将 SPC 定位到页边界；

不带操作数的 **.align** 伪指令，操作数默认为 64，也就是页边界。



(a) **.align 2** 执行的结果



(b) 没有参数的 **.align** 执行的结果

图 4.4 **.align** 伪指令

图 4.4 说明了 **.align** 伪指令。假设下列代码已汇编：

1	000000	0002	.field	2,3
2	000000	005A	.field	11,8
3			.align	2

```

4  000002 0065      .string "errorcnt"
   000003 0072
   000004 0072
   000005 006F
   000006 0072
   000007 0063
   000008 006E
   000009 0074
5
6  000040 0004      .align
                        .byte 4

```

4.6 输出列表格式伪指令

下列伪指令规定列表文件的格式：

- **.drlist** 伪指令使在列表文件中输出伪指令行；**.drnolist** 伪指令对某些伪指令关闭列表。用户可以使用**.drnolist** 伪指令禁止输出下列伪指令：

.asg	.eval	.length	.mnolist	.var
.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg

可以使用**.drlist**伪指令重新打开列表。

- 源代码包括不能生成代码的条件为假的代码块的列表。**.fclist** 和**.fcnolist** 伪指令打开和关闭此列表。用户可以使用**.fclist** 伪指令对条件为假的代码块列表，其格式与它们在源代码中格式一样。用户也可以使用**.fcnolist** 伪指令只对实际汇编过的条件为真的代码块列表。
- **.length** 伪指令控制列表文件每页的长度。用户可以用这个命令针对不同的输出设备调整列表。
- **.list** 和**.nolist** 伪指令打开和关闭输出列表。用户可以用**.nolist** 伪指令禁止汇编器在列表中输出选定的源语句。用**.list** 伪指令重新打开列表。
- 源代码包含宏扩展和循环块列表。伪指令**.mlist** 和**.mnolist** 打开和关闭这种列表。用户可以用**.mlist** 伪指令在列表中输出所有的宏扩展和循环块，用**.mnolist** 伪指令禁止这种列表。
- **.option** 伪指令控制列表文件中的某些特性。这一伪指令有如下操作数：
 - A** 打开所有伪指令、数据、后续扩展、宏、块的列表；
 - B** 将**.byte** 伪指令的列表限制在一行内；
 - D** 关闭某些伪指令的列表（执行一次**.drnolist**）；
 - L** 将**.long** 伪指令的列表限制在一行内；
 - M** 在列表中禁止宏扩展；
 - N** 关闭列表（执行一次**.nolist**）；
 - O** 打开列表（执行一次**.list**）；
 - R** 重新设置 **B**、**L**、**M**、**T** 和 **W** 伪指令（关闭对 **B**、**L**、**M**、**T** 和 **W** 的限制）；

- T** 将.string 伪指令的列表限制在一行内;
 - W** 将.word 和.int 伪指令的列表限制在一行内;
 - X** 生成符号交叉引用列表。用户也可以用-x 选项调用汇编器来获得交叉引用列表。
- **.page** 伪指令使列表按页输出。
 - 源代码列表包含置换符号属性。**.sslist** 和**.ssnolist** 伪指令打开和关闭这种列表方式。用户可以使用.sslist 伪指令在列表中将所有的置换符号输出, 用.ssnolist 伪指令禁止列表输出置换符号。这些伪指令在调试置换符号时是有用的。
 - **.tab** 伪指令定义 tab (制表符) 的大小。
 - **.title** 伪指令为汇编器提供输出在每页顶部的标题。
 - **.width** 伪指令控制列表的页宽。用户可以用这一伪指令调整列表以适应不同的输出设备。

4.7 引用其他文件的伪指令

这些伪指令为其他文件提供信息或是提供关于其他文件的信息:

- **.copy** 和**.include** 伪指令告诉汇编器从另一文件中读入源程序语句。汇编器读完.copy/.include 文件的源程序语句后, 恢复从当前文件中读入源程序语句。从复制的文件中读入的语句输出到列表文件中; 从包含文件中读入的语句不输出到列表文件中。
- **.def** 伪指令标识在当前模块定义的符号可以在另一模块中使用。汇编器在符号列表中包含此符号。
- **.global** 伪指令声名一个全局符号, 以便在链接时可供其他模块使用。(有关全局符号的详细资料, 见第 2.6.1 节全局符号)。**.global** 伪指令完成双重任务, 对于已定义符号它相当于.def 伪指令, 对于未定义符号, 它相当于.ref 伪指令。只有当符号在程序中使用, 链接器才作为未定义符号引用。
- **.mlib** 伪指令为汇编器提供包含宏定义的归档库的名称。当汇编器在当前模块中遇到没有定义的宏时, 就到由.mlib 指定的宏库中搜索。
- **.ref** 伪指令标识一个在当前模块中使用但在另一模块中定义的符号。汇编器将该符号标记为未定义的外部符号, 并放入目标文件表中, 以便链接器能识别其定义。**.ref** 伪指令强制链接器识别符号引用。

4.8 条件汇编伪指令

条件汇编伪指令指示汇编器根据表达式求值结果的真假来汇编代码的某些段。共有两

组伪指令允许用户有条件的汇编部分代码块：

- **.if/.elseif/.else/.endif** 伪指令告诉汇编器根据表达式的值有条件地汇编部分代码块。
 - .if *well-defined expression*** 标明条件块的开始，如果.if 表达式为真时，那么汇编该代码块。
 - [.elseif *well-defined expression*]** 如果.if 表达式为假，而且.else 表达式为真时，汇编该代码块。
 - [.else]** 如果.if 表达式为假，那么汇编此代码块。
 - .endif** 标明条件块的末尾并结束该块。
- **.loop/.break/.endloop** 伪指令告诉汇编器根据表达式的值重复地汇编程序。
 - .loop [*well-defined expression*]** 标明一个循环代码块的开始。选项定义的表达式的值是循环次数。
 - [.break[*well-defined expression*]]** 告诉汇编器当.break 表达式值为假时继续重复汇编；如果表达式值为真，那么跳转到.endloop 之后的代码处。
 - .endloop** 标明重复块的结束。

汇编器支持几种对条件表达式有用的关系运算符。有关关系运算符的详细资料见第3.9.4节条件表达式。

4.9 汇编过程使用的符号的伪指令

汇编过程使用的（Assembly-Time）符号伪指令使符号名同常数或字符串等同起来。

- **.asg** 伪指令把字符串赋给替代符号。替代值存放在置换符号表中。当汇编器遇到一个置换符号时，它用字符串的值置换这个符号。置换符号可以重复定义。

```
.asg    "10, 20, 30, 40", coefficients
.byte   coefficients
```

- **.eval** 伪指令对表达式求值，把结果转换为字符串。该伪指令常用于操作计数器，如下例所示：

```
.asg    1, x
.loop
.bytex*10h
.break  x = 4
.eval  x+1, x
.endloop
```

- **.label** 伪指令在段内定义一个特殊符号，指向装载地址。当一个段装载在某一个地

址，而又在另一地址运行时，这一伪指令是有用的。例如，用户可能想将一个性能要求严格的代码块放在片外低速存储器以节省空间，又想将代码块移至片内高速存储器运行。

- **.set** 伪指令把常数赋予符号。符号存储在符号表中且不能够重复定义，例如：

```
bval      .set      1000h
          .long      bval, bval*2, bval+12
MOV       AL, #bval
```

.set 伪指令来生成目标代码。

- **.struct/.endstruct** 伪指令建立类似于 C 语言的结构定义，**.tag** 伪指令把类似 C 语言的结构特性赋给标号。

.struct/.endstruct 伪指令允许用户将类似的元素以结构的方式组织起来。元素偏移量的计算留给汇编器来做。**.struct/.endstruct** 伪指令不分配内存。它只是简单地创建可以重复使用的符号模板。

.tag 伪指令把结构特性赋给标号。这样就简化了符号的表示方法，并允许定义包含其他结构的结构。**.tag** 伪指令不分配内存，但结构的名称一定要在使用前定义。

```
COORDT    .struct
X          .int
Y          .int
T_LEN     .endstruct
COORD     .tag      COORDT, T_LEN
ADD      ACC,  @COORD.Y
.bss     COORD
```

4.10 汇编器模式伪指令

这些伪指令设定第3.15节TMS320C28x汇编器模式中讨论的语法检查模式。这些伪指令在C27x目标模式中无效。

- **.c28_amode** 伪指令将汇编器设定为 C28x 目标模式（-v28）。不论命令行中的选项是什么，这一汇编伪指令后的指令都在 C28x 目标模式下汇编。
- **.lp_amode** 伪指令将汇编器设定为 C28x 目标——兼容 C2xlp 伪指令语法（-m20）。本汇编伪指令后的指令在汇编时，就如同在命令行中设定了 -m20 选项一样。

4.11 其他伪指令

这些伪指令具有以下功能或特性：

- **.clink** 伪指令为已命名段在类型域设置 STYP_CLINK 标志。**.clink** 伪指令可用于已初始化段或未初始化段。STYP_CLINK 标志使条件链接有效，如果段内没有发现对任何符号的引用，它就告诉链接器在链接器最终的 COFF 输出中忽略这一段。
- **.end** 伪指令结束汇编。如果用户使用 **.end** 伪指令，它应该放在一个源程序的最后。这一伪指令同一个文件结束字符的效果一样。
- **.newblock** 伪指令重新设置局部标号。局部标号是 **NAME?** 形式的符号，其中 **NAME** 由用户指定。当它们在标号域出现时被定义。局部标号是供跳转指令用做操作数的暂时标号。**.newblock** 伪指令在局部标号使用后，重新设置它的值来限制局部标号的使用范围（详细资料见 3.8.2 节局部标号）。

下面三条伪指令允许用户定义自己的错误和警告信息：

- **.emsg** 伪指令向标准输出设备传送错误信息。**.emsg** 伪指令以汇编器相同的方式产生错误信息，增加错误计数，并阻止汇编器生成目标文件。
- **.mmsg** 伪指令把汇编时的信息传送到标准输出设备。**.mmsg** 伪指令的功能与 **.emsg** 和 **.wmsg** 一样，但它不设置错误计数或警告计数，也不影响目标文件的创建。
- **.wmsg** 伪指令将警告信息传送到标准输出设备。**.wmsg** 伪指令的功能与 **.emsg** 相同，但增加警告计数而不是错误计数。它不影响目标文件的创建。

在宏中使用错误和警告伪指令的详细资料见第 5.7 节在宏中生成信息。

4.12 伪指令索引表

下面是按字母排序的伪指令索引表相关的伪指令（如 **.if/.else/.endif**）放在一页中。

伪 指 令	伪 指 令
.align	.list
.asg	.long
.bes	.loop
.break	.lp_amode
.bss	.mlib
.byte	.mlist
.c28_amode	.mmsg
.char	.mnolist
.clink	.newblock
.copy	.nolist
.data	.option
.def	.page
.drlist	.pstring
.drnolist	.ref
.else	.sblock
.elseif	.sect

续表

伪 指 令	伪 指 令
.emsg	.set
.end	.space
.endif	.sslist
.endloop	.ssnolist
.endstruct	.string
.eval	.struct
.fclist	.tab
.fcnolist	.tag
.field	.text
.float	.title
.global	.usect
.if	.width
.include	.wrnsg
.int	.word
.label	.xfloat
.length	.xlong

语法: **.align**

格式:

.align [*size in words*]

描述: **.align**伪指令将段程序计数器 (SPC) 调准到下一个边界, 边界由参数 *size in words* (字数) 指定。 *size* 可以是2的任意整数次幂, 默认设置为64。汇编器把包括**NOP**的字汇编到下一个x-word字的边界。X为1, 2, 3, 如下所述:

- 操作数
- 1 SPC调准到字边界;
 - 2 SPC 调准到长字/偶数边界;
 - 3 SPC 调准到页边界;

使用**.align**伪指令有两个作用:

- 汇编器把 SPC 调准到当前段内 x-word 字的边界上。
- 汇编器设置标志, 强制链接器调准段, 这样当一个段装载到存储器时, 调准能保持原样。

例: 下例显示了几种类型的调准, 有**.align2**, **.align4**和一个默认的**.align**。

```

1  000000  0004      .byte      4
2                      .align      2
3  000002  0045      .string     "Errorcnt"
   000003  0072
   000004  0072
   000005  006F
   000006  0072

```

```

000007 0063
000008 006E
000009 0074
4
5 000040 0003 .align 3,3
6 000040 002B .field 5,4
7 .align 8
8 000042 0003 .field 3,3
9 .align
10 000048 0005 .field 5,4
11 .align
12 000080 0004 .byte 4

```

语法: **.asg/.eval**

格式:

.asg ["] *character string* ["], *substitution symbol*
.eval *well-defined expression*, *substitution symbol*

描述: **.asg**伪指令把字符串赋给置换符号。置换符号存在置换符号表中。**.asg**伪指令使用时在很多方面与**.set**相同,但是**.set**把常数(不能重新定义)赋给符号,而**.asg**把字符串(可以被重新定义)赋给置换符号。

- 汇编器把 *character string* (字符串) 赋给置换符号。引号是可选择的。如果没有引号,那么汇编器读字符至第一个逗号并去掉前后的空格。如果有引号,则读入字符串,并赋给置换符号。
- *substitution symbol* (置换符号) 是所需的参数,它必须是一个有效的符号名。置换符号可长达 32 位,必须以字母开始,其余字符可以是数字,字母,下划线,美元符号。

.eval 伪指令对存放在置换符号表中的置换符号执行算术计算。这一伪指令计算表达式的值,并指定由计算结果转化来的字符串作为置换符号。**.eval** 伪指令特别适用于**.loop/.endloop** 块中的计数器。

- *well-defined expression* (明确定义的表达式) 是字母和数字表达式,由前面定义的合法值组成。

例: 说明怎样使用**.asg** **.eval** 伪指令。

```

1 .sslist
2 .asg XAR6, FP
3 00000000 0964 ADD ACC, #100
4 00000001 7786 NOP *FP++
# NOP *XAR6++
5 00000002 7786 NOP *XAR6++
6
7 .asg 0, x
8 .loop 5
9 .eval x+1, x

```

```

10      .word      x
11      .endloop
1      .eval      x+1, x
#      .eval      0+1, x
1      00000003    0001      .word      x
#      .word      1
1      .eval      x+1, x
#      .eval      1+1, x
1      00000004    0002      .word      x
#      .word      2
1      .eval      x+1, x
#      .eval      2+1, x
1      00000005      0003      .word      x
#      .word      3
1      .eval      x+1, x
#      .eval      3+1, x
1      00000006      0004      .word      x
#      .word      4
1      .eval      x+1, x
#      .eval      4+1, x
1      00000007      0005      .word      x
#      .word      5

```

语法: **.bss**

格式:

.bss symbol size in words [, blocking flag] [, alignment flag] [, type]

描述: **.bss**伪指令在**.bss**段中为变量保留空间。这一伪指令通常用于在RAM中分配变量。

- **symbol** (符号) 是必需的参数。它定义了一个标号, 该标号指向本伪指令所保留的第一个单元地址。符号名应与所保留空间的变量名一致。
- **size in words** 是必需的参数; 它必须是确定的表达式。汇编器在**.bss**段中分配 **size** 个字数。size 没有默认值。
- **blocking flag** (块标志) 是可选参数。如果用户定义本参数的值大于 0, 那么汇编器连续分配 **size** 个字。这意味着分配的空间不会越过页边界, 除非指定的字数大于一页。在这种情况下, 从页的边界开始。
- **alignment flag** (调准标志) 是可选参数。这一标志使汇编器以长型字为边界来分配 **size** 个字。
- **type** 是可选的参数。指定 **type** 使汇编器为符号生成适当的调试信息。详细资料见第 3.14 节 C-类型汇编变量的符号调试 (-mg 选项)。

汇编器在**.bss**段中分配空间时, 遵循两条准则:

- 准则1 当存储器中留有空位时(如图4.5所示), **.bss**伪指令会尽量填满它。当汇编**.bss**伪指令时, 汇编器搜索前一次**.bss**伪指令留下的空位列表, 并尽量把当前块分配到这些空位中(无论是否设置模块化标志, 这都是标准的过程)。
- 准则2 若汇编器没有找到能放下整个块的空位, 那么它将检查块选项标志是否有效。

- 如果未要求按块存放，那么在当前 SPC 处分配存储器。
- 如果用户要求按块存放，那么汇编器将检查在当前 SPC 和页边界之间的空位是否足够大。如果没有足够的空间，那么汇编器将在下一页上分配空间。

块选项允许用户在.bss段中保留最多64个字，并保证它们放在存储器的同一页中。（当然，用户可以一次保留多于64字的空间，但它们不能放在同一页中）。下列代码在.bss段中保留了两块空间：

```
memptr:    .bss A,32,1
memptr1:   .bss B,35,1
```

每一个块必须包含在同一页中。因此，第一块分配存储空间后，第二块就不能放在当前页了。如图4.5所示，第二块在下一页中分配空间。

用于初始化段的段伪指令（.text, .data, .sect）结束当前段并告诉汇编器开始汇编到另一个段中。但是，.bss伪指令不影响当前段。汇编器汇编到.bss伪指令后，继续将代码汇编到当前段。有关COFF段的详细资料见第2章通用目标文件格式。

例：.bss伪指令用来为两组变量TEMP和ARRAY分配空间，TEMP符号指向4个字的未初始化空间（.bss SPC=0）。ARRAY符号指向100个字的未初始化空间（.bss SPC=040h）；这一空间必须在一页内连续分配。用.bss伪指令定义的符号可以像其他符号一样引用，也可以定义为外部符号。

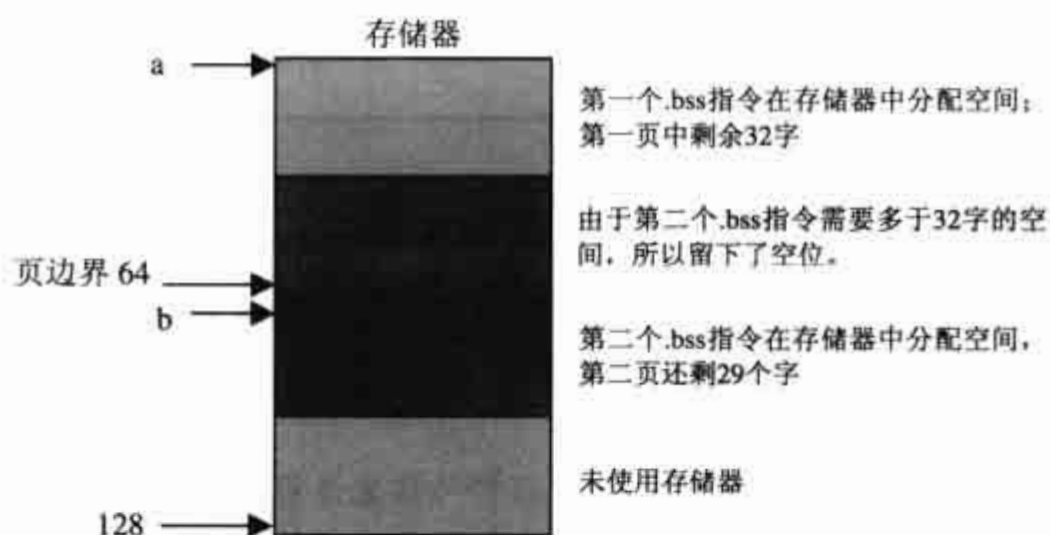


图4.5 在一页内分配.bss块

```
1 *****
2 ** 开始汇编到.text段中 **
3 *****
4 000000 .text
5 000000 2BAC MOV T, #0
6
7 *****
8 ** 在.bss段中分配4个字的空间 **
9 *****
10 000000 .bss Var_1, 2, 0, 1
11
12 *****
```



```

13  ** 仍在.text 段中                                     **
14      ****
15  000001  08AC      ADD T, #56h
16      000002  0056
17  000003  3573 MPY   ACC, T, #73h
18
19
20
21  000040      .bss ARRAY, 100, 1
22
23
24
25  000004  F800-      MOV DP, #Var_1
26  000005  1E00-      MOVL @Var_1, ACC
27  ****
28  ** 声明外部的.bss 符号                                **
29  ****
30      .global ARRAY
31      .end

```

语法: **.byte/char**

格式:

```

.byte value1 [, ... , valuen]
.char value1 [, ... , valuen]

```

描述: **.byte**和 **.char**伪指令将一个或多个字节放入当前段的连续字中。每个字节自己放在一个字中; 8个MSBs (最高位) 填0。value (值) 可以是下列情况中的一种:

- 一个表达式, 汇编器把它当作 8 位有符号数来计算和处理。
- 包含在双引号内的字符串。字符串中每一个字符代表一个单独的数。Value (数) 未被合并或作符号扩展; 每个字节占据一个完整的 16 位字的低 8 位。汇编器将多于 8 位的值截断。每个**.byte** 指令可以使用多达 100 个 value (值) 参数, 但每行长度不能超过 200 个字符。

如果用户使用一个标号, 那么它指向汇编器放置第一个字节的单元地址。

当用户在**.struct/.endstruct** 序列中使用**.byte** 时, **.byte** 定义结构成员的大小, 它不初始化存储器。有关**.struct/.endstruct** 的详细资料见第 4.9 节汇编过程使用符号的伪指令。

下例中 8 位值 (10, -1, abc, a) 被放到存储器的连续字中。标号 STRX 的值为 100h, 是第一个已初始化字的地址:

```

1  000000      .space 100h * 16
2  000100  000A      .byte 10, -1, "abc", 'a'
      000101  00FF

```

```

000102 0061
000103 0062
000104 0063
000105 0061
3 000106 000A      .char 10, -1, "abc", 'a'
000107 00FF
000108 0061
000109 0062
00010a 0063
00010b 0061

```

语法: **.c28_amode/.lp_amode**

格式:

```

.c28_amode
.lp_amode

```

描述: **.c28_amode**和**.lp_amode**伪指令告诉汇编器执行的汇编模式。详细资料见第3.15节TMS320C28x汇编模式。

.c28_amode 伪指令告诉汇编器在 C28x 目标模式 (-v28) 下操作。**.lp_amode** 伪指令告诉汇编器运行在 C28x 目标——兼容 C2xlp 语法格式 (-m20) 的模式下操作。这些伪指令可以在整个源文件中被反复使用。

例如, 如果一个文件用 -m20 选项, 汇编器就在 C28x 目标——兼容 C2xlp 语法格式 (-m20) 的模式下汇编。当它遇到 **.c28_amode** 伪指令后, 它变换到 C28x 目标模式, 并保持这种模式直到它遇到 **.lp_amode** 伪指令或文件结束。

这些伪指令帮助用户通过用 C28x 代码代替部分 C2xlp 代码, 来实现从 C2xlp 到 C28x 的移植。

下例中, 将C28x代码插入到已经存在的C2xlp代码中:

```

; C2xlp 源代码
LDP #VarA
LACL VarA
LAR AR0, *+, AR2
SACL *+
.
.
CALL FuncA
.
.
; 在函数 FuncA 中的 C2xlp 代码被使用 C28x 的寻址方式 (AMODE = 0) 的 C28x 代码替代
.c28_amode      ; 使编辑器模式执行 C28x 的语法
FuncA:
C28ADDR          ; 置 AMODE 为 0, 进入 C28x 寻址模式
MOV DP, #VarB
MOV AL, @VarB
MOVL XAR0, *XAR0++

```

```

MOV *XAR2++, AL
.lp_amode      ; 使汇编器回到 C2x1p
LPADDR        ; AMODE 为 1，恢复 C2x1p 寻址
LRET

```

语法: **.clink**

格式:

.clink [*"section name"*]

描述: **.clink** 伪指令通过在段名 (*section name*) 的类型域设置 **STYP_CLINK** 标志来建立条件链接。已初始化段和未初始化段都可以使用 **.clink** 伪指令。

如果用户使用 **.clink** 时没带段名 (*section name*) 参数, 那它就用于当前已初始化段。如果 **.clink** 要用于未初始化段, 必须有段名 (*section name*)。段名在 200 个字符内是有效的, 且必须用双引号引起来。段名 (*section name*) 可以以: 段名; 子段名 (*section name : subsection name*) 的形式包含子段。

STYP_CLINK 标志告诉链接器如果某段对任何符号都没有引用, 那么就在最后的 COFF 输出中省略该段。

定义了 C 语言入口的段, 不能设置条件链接。

下例中 **Vars** 和 **Counts** 段被设定为条件链接:

```

1  000000      .sect  "Vars"
2                      ; 有条件链接 Vars 段
3                      .clink
4
5  000000 001A  X:      .long  01Ah
6  000001 0000
7  000002 001A  Y:      .word  01Ah
8  000003 001A  Z:      .word  01Ah
9                      ; 有条件链接 Counts 段
10                     .clink
11 000004 001A  XCount:  .word  01Ah
12 000005 001A  YCount:  .word  01Ah
13 000006 001A  ZCount:  .word  01Ah
14                      ; 默认时, 无条件链接 .text 段
15 000000      .text
16
17 000000 97C6      MOV   *XAR6, AH
18                      ; 引用符号 X 使 Vars 段被链接
19                      ; 到 COFF 输出中
20 000001 8500+      MOV   ACC, @X
21 000002 3100      MOV   P, #0
22 000003 0FAB      CMPL  ACC, P

```

语法: **.copy/include**

格式:

```
.copy ["filename"]  
.include ["filename"]
```

描述: **.copy**和**.include**伪指令告诉汇编器从另一文件中读取源程序语句。不管被汇编的**.list/.nolist**伪指令数目如何, 来自复制文件的被汇编语句都打印在汇编列表中, 来自包含文件的被汇编语句不打印在汇编列表中。汇编器:

- 1) 结束汇编当前源程序中的语句
 - 2) 汇编被复制/包含文件的语句
 - 3) 在主源程序文件中, 开始恢复汇编跟随在伪指令**.copy**或**.include**之后的语句
- filename* (文件名) 是命名源程序文件必需的参数。它被包含在双引号之内且必须遵守操作系统的惯例。用户可以指定一个路径名 (如, `c:\dsp\file1.asm`)。如果用户不指定路径, 那么汇编器在下列目录中搜索文件:

- 1) 包含当前源程序文件的目录
- 2) 用**-i** 汇编器选项命名的所有目录
- 3) 用环境变量**C2000_A_DIR**和**A_DIR**指定的所有目录

有关**-i**选项和**C2000_A_DIR**, **A_DIR**的详细资料, 见第3.4节为汇编器输入替换目录名。

.copy和**.include**伪指令可以被嵌套在被复制或被包含的文件中。汇编器将这种类型的嵌套层数限制在10层, 可以设置附加限制。汇编器在被复制文件的行数前放一个字母代码, 用来识别复制的层数。**A**指明第一次被复制的文件, **B**指明第二次被复制的文件, 等等。

例1: **.copy**伪指令用来从其他文件中读入源程序语句并且汇编, 然后汇编器恢复汇编当前文件。

原始文件**copy.asm**中包含一条语句**.copy "byte.asm"**文件。当汇编**copy.asm**时, 汇编器将**byte.asm**的内容复制到列表中。被复制的文件**byte.asm**中包含一个**.copy**至于第二个文件**"word.asm"**语句。

当汇编器汇编**byte.asm**文件的**.copy**语句时, 就转向文件**word.asm**继续复制和汇编。然后返回它在文件**byte.asm**中的位置, 再继续复制和汇编。完成对文件**byte.asm**的汇编后, 汇编器回到文件**copy.asm**, 继续汇编剩余语句。

copy.asm (源文件)	byte.asm (第一次复制文件)	word.asm (第二次复制文件)
.space 29 .copy "byte.asm" **Back in original file .pstring "done"	** In byte.asm .byte 32,1+ 'A' .copy "word.asm" ** Back in byte.asm .byte 67h + 3q	** In word.asm .word 0ABCDh, 56q

列表文件:

1 000000 .space 29


```

2          .copy "byte.asm"
1
2          ; In byte.asm
2 000002 0005 .byte 5
3          .copy "word.asm"
1          ** In word.asm
2 000003 ABCD .word 0ABCDh
4          * Back in byte.asm
5 000004 0006 .byte 6
3
4          **Back in original file
5 000005 646F .pstring "done"
000006 6E65

```

例2: `.include`伪指令用来从其他文件读取源程序语句并汇编, 然后汇编器继续汇编当前文件。除了源语句不输出到列表文件中之外, 其工作方式与`.copy`伪指令相似。

copy.asm (源文件)	byte.asm (第一次复制文件)	word.asm (第二次复制文件)
<pre> .space 29 .include "byte2.asm" **Back in original file .pstring "done" </pre>	<pre> ** In byte2.asm .byte 32,1+ 'A' .include "word2.asm" ** Back in byte2.asm .byte 67h + 3q </pre>	<pre> ** In word2.asm .word 0ABCDh, 56q </pre>

列表文件:

```

1 000000 .space 29
2          .include "byte2.asm"
3
4          ; Back in original file
5 000007 0064 .string "done"
000008 006F
000009 006E
00000a 0065

```

语法: `.data`

格式:

.data

描述: `.data`伪指令告诉汇编器开始将源代码汇编到`.data`段, 且`.data`变为当前段。`.data`段通常用于包括数据表或已初始化变量。

汇编器假设`.text`是默认段。因此, 除非用户使用段控制伪指令, 否则, 在汇编开始时, 汇编器把代码汇编到`.text`段中。

有关COFF段的详细资料见第2章通用目标文件格式。

下例中, 代码被汇编到`.data`和`.text`段中:

```

1 *****
2          ** 在.data 中保留空间          **
3 *****

```

```

4  000000      .data
5  000000      .space 0CCh
6
7      ** 汇编到.text 段中      **
8      ****
9  000000      .text
10 0000      INDEX      .set      0
11 000000      9A00      MOV AL,#INDEX
12
13      ** 汇编到.data 段中      **
14      ****
15 00000c Table:      .data
16 00000d FFFF      .word  -1      ; 汇编 16 位常数到.data
17 00000e 00FF      .byte  0FFh    ; 汇编 8 位常数到.data
18
19      ** 汇编到.text 段中      **
20      ****
21 000001      .text
22 000001 08A9      ADD AL,Table
23 000002 000C
24
25      ** 继续汇编到地址为 0Fh 的.data 的段中      **
26      ****
26 00000f      .data

```

语法: **.drlist/.drnolist**

格式:

.drlist
.drnolist

描述: 这两条伪指令使用户能控制将汇编伪指令输出到列表文件中:

.drlist伪指令使能输出所有的伪指令到列表文件。

.drnolist伪指令禁止在列表中输出下述伪指令:

.asg	.fcnolist	.sslist
.break	.length	.ssnolist
.emsg	.mlist	.var
.eval	.mmsg	.width
.fclist	.mnolist	.wmsg

默认设置为**.drlist**。

下例表示**.drnolist**伪指令是如何禁止列出上述伪指令的:

源文件:

```

.asg 0, x
.loop 2

```

```

.eval x+1, x
.endloop
.drnolist
.asg 1, x
.loop 3
.eval x+1, x
.endloop

```

列表文件:

```

1          .asg 0, x
2          .loop 2
3          .eval x+1, x
4          .endloop
1          .eval 0+1, x
1          .eval 1+1, x
5
6          .drnolist
7
9          .loop 3
10         .eval x+1, x
11        .endloop

```

语法: **.emsg/.mmsg/.wmsg**

格式:

```

.emsg string
.mmsg string
.wmsg string

```

描述: 这些伪指令允许用户定义自己的错误和警告信息。汇编器追踪和报告出现的错误和警告的数目, 并输出到列表文件的最后一行。

.emsg伪指令像汇编器一样发送错误信息到标准输出设备, 增加错误计数并阻止汇编器生成目标文件。

.mmsg伪指令和**.emsg**和**.wmsg**一样, 将汇编中的信息发送到标准输出设备, 但它不增加错误和警告计数, 不阻止汇编器生成目标文件。

.wmsg伪指令和**.emsg**一样将警告信息发送到标准输出设备, 但不增加错误计数, 而是增加警告计数, 不阻止汇编器生成目标文件。

下例中, **ERROR——MISSING PARAMETER** 信息传送到标准输出设备:

源文件:

```

MSG_EX      .global PARAM
             .macro parml
             .if $symlen(parml) = 0
             .emsg "ERROR -- MISSING PARAMETER"
             .else

```

```

        add AL, @parml
        .endif
        .endm
MSG_EX PARAM
MSG_EX
        .emsg/.mmsg/.wmsg Define Messages

```

列表文件:

```

1          .global PARAM
2          MSG_EX .macro parml
3          .if $symlen(parml) = 0
4          .emsg "ERROR -- MISSING PARAMETER"
5          .else
6          add AL, @parml
7          .endif
8          .endm
9
10 000000 MSG_EX PARAM
1          .if $symlen(parml) = 0
1          .emsg "ERROR -- MISSING PARAMETER"
1          .else
1 000000 9400! add AL, @PARAM
1          .endif
11
12 000001 MSG_EX
1          .if $symlen(parml) = 0
1          .emsg "ERROR -- MISSING PARAMETER"
***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1          .else
1          add AL, @parml
1          .endif
1 Error, No Warnings

```

语法: **.end**

格式:

.end

描述: **.end**伪指令是可选的, 它用于结束程序汇编。它应当是源程序的最后一行语句。汇编器将忽略跟在**.end**伪指令后的任何语句。

这一伪指令与文件结束字符有相同的作用。在调试过程中, 如果用户想在代码的某个特定点停下来, 可以使用**.end**。

注意: 终止宏, 使用**.endm**伪指令终止宏。

例: **.end**伪指令结束程序汇编。如果**.end**伪指令后跟有语句, 汇编器将忽略它们。

源文件:

```
START: .space 300
```



```

TEMP    .set 15
        .bss LOC1, 48h
        ABS ACC
        ADD ACC, #TEMP
        MOV @LOC1, ACC
        .end
        .byte 4
        .word CCCh

```

列表文件:

```

1 000000      START:      .space 300
2          000F  TEMP      .set 15
3 000000      .bss LOC1, 48h
4 000013 FF56      ABS ACC
5 000014 090F      ADD ACC, #TEMP
6 000015 9600-     MOV @LOC1, ACC
7 .end

```

语法: **.fclist/.fcnolist**

格式:

.fclist	生成条件为假时的代码块列表
.fcnolist	禁止生成条件为假时的代码块列表

描述: 这两条伪指令使用户能控制条件为假时的代码块产生列表:

.fclist伪指令允许条件为假时的代码块产生列表: (条件不生成执行的代码块)。

.fcnolist伪指令禁止虚假条件块产生列表, 直到遇见**.fclist**伪指令为止。使用**.fcnolist**后, 只有实际汇编过的代码块才出现在列表中, 而**if**, **else**和**endif**伪指令不在列表中出现。

默认设置为所有代码块均被列出; 同使用**.fclist**伪指令时一样。

例: 对于使用和不使用条件的汇编语言文件和列表文件。

源文件:

```

AAA      .set 1
BBB      .set 0
          .fclist
          .if AAA
          ADD ACC, #1024
          .else
          ADD ACC, #1024*4
          .endif
          .fcnolist
          .if AAA

```

```

        ADD ACC, #1024
    .else
        ADD ACC, #1024*10
    .endif

```

列表文件:

```

1      0001 AAA      .set 1
2      0000 BBB      .set 0
3                               .fclist
4
5                               .if AAA
6  000000 FF10      ADD ACC, #1024
   000001 0400
7                               .else
8                               ADD ACC, #1024*4
9                               .endif
10
11                               .fcnolist
12
14  000002 FF10      ADD ACC, #1024
   000003 0400

```

语法: **.field**

格式:

```
.field value [, size in bits]
```

描述: **field** 伪指令初始化单字存储器内的多个位域。此伪指令有两个操作数:

- *value* 是必需的参数,它是放在域中的求值表达式。如果该值是可重定位的,那么 *size* 必须大于或等于 32。
- *size* 是可选参数,它指定一个从 1 到 32 之间的数作为域的位数。如果用户不指定 *size*,那么汇编器按 16 位处理。如果用户指定的位数大于或等于 16,则分配的域从字边界开始。如果用户指定的数 *value* 与 *size* 不相等,汇编器截断 *value* 的位数,并报告错误信息。例如 **.field 3, 1** 汇编器将 3 截为 1,并打印信息:

***warning - value truncated.

连续的 **.field** 伪指令将把数打包放入当前字的指定位数中,并从最低位开始,随域的增加逐渐向最高位移动。如果汇编器遇到域的大小大于当前字的剩余位,那么就增加 SPC 的值,然后开始将该域打包放入下一个字中。用户可以用带操作数 1 的 **.align** 伪指令,迫使下一个 **.field** 开始打包进一个新字中。

当所指定域大于 16 位时,汇编器首先填满低字,然后从高字的最低位开始填入。

如果使用了标号,那么标号指向包含指定域的字。

如果用户在 **.struct/.endstruct** 序列中使用 **.field**,那么 **.field** 只定义一个成员的大小;它不初始化存储器。

例:显示域如何打包进一个字。SPC 在一个字被填满并开始下一个字之前不改变。

```

1      ****
2      **  初始化一个 14 位的域                                **
3      ****
4 000000 0ABC          .field 0ABCh, 14
5
6      ****
7      **  在一个单独的字中初始化一个 5 位的域                **
8      ****
9
10 000001 000A      L_F:  .field 0Ah, 5
11
12      ****
13      ** 在同一字中初始化一个 5 位的域                        **
14      ****
15
16 000001 018A      X:    .field 0Ch, 4
17
18      ****
19      ** 在接下来的 2 个字中初始化一个 22 位的                **
20      ** 可重定位的域                                          **
21      ****
22 000002 0001'      .field X
23
24      ****
25      **  初始化一个 32 位的域                                **
26      ****
27 000003 4321          .field 04321h, 32
    000004 0000

```

图4.6显示伪指令是如何影响存储器的。

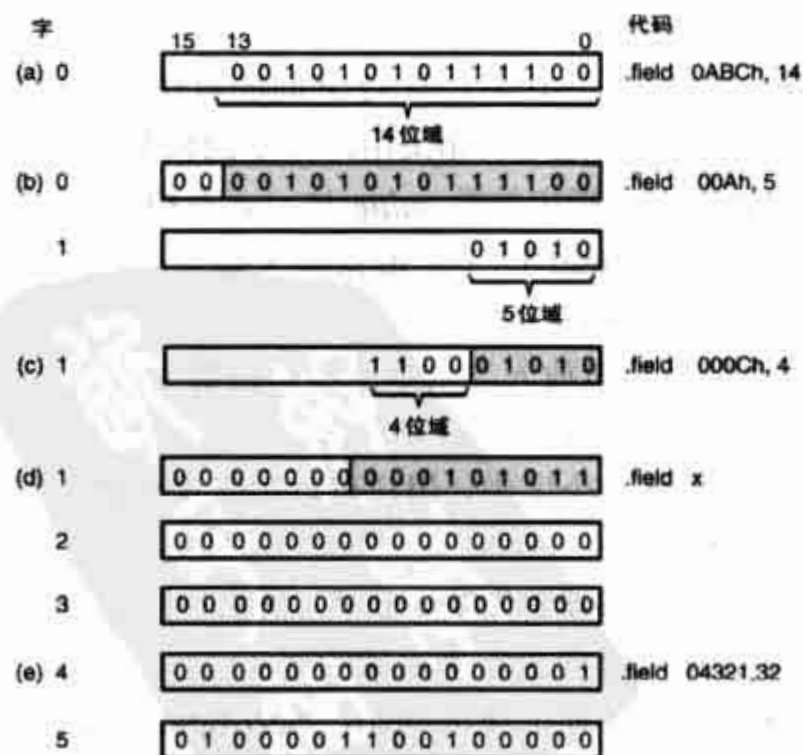


图 4.6 .field 伪指令

语法: **.float/.xfloat**

格式:

```
.float value1 [, ... , valuen]
.xfloat value1 [, ... , valuen]
```

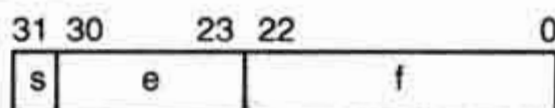
描述: **.float**和**.xfloat**伪指令把一个或多个浮点型常数放入当前段中。*value*必须是浮点常数或是已被赋值为浮点常数的符号。每个常数都被转换成IEEE 32位格式的单精度浮点值。如果不使用**.xfloat**伪指令,那么浮点型常数被调整到长字边界。

.xfloat与**.float**功能相同,但它不将结果调整到长字边界。

32 位值包含 3 个区域:

域	含 义
s	1 位符号域
e	8 位有偏指数
f	23 位小数

值的存放顺序是从字的最低位开始,然后才是字的最高位,格式如下:



在**.struct/.endstruct**序列中使用**.field**时,**.field**定义一个成员的大小;它不初始化存储器。有关**.struct/.endstruct**的详细资料,见第4.9节汇编过程使用符号的伪指令。

例: 说明**.float**和**.xfloat**伪指令。

```
1  00000000 5951      .float -1.0e25
   00000001 E904
2  00000002 0010      .byte 0x10
3  00000003 0000      .xfloat 123.0    ;未调整到长字边界
   00000004 42F6
4  00000006 0000      .float 3          ;调整到长字边界
   00000007 4040
```

语法: **.global/.def/.ref/.globl**

格式:

```
.global symbol1 [, ... , symboln]
.def symbol1 [, ... , symboln]
.ref symbol1 [, ... , symboln]
.globl symbol1 [, ... , symboln]
```

描述: **.global**, **.def**和**.ref**伪指令标识全局符号,全局符号可以在外部定义或者被外部引用。

.globl伪指令为C2xlp提供了向上兼容性。它只在指定-m20选项时才被接受。不鼓励使用**.globl**伪指令。

.def伪指令标识在当前模块中定义并可被其他文件访问的符号。汇编器将此符号放入符号表中。

.ref伪指令标识在当前模块中使用但在另一模块内定义的符号。链接器在链接时解决此符号的定义。

当需要时，**.global** 伪指令起**.ref** 或**.def** 的作用。

全局符号与其他符号的定义方式相同；也就是说，它以标号方式出现或用**.set**、**.bss** 式**.usect** 伪指令定义。像所有的符号一样，如果一个全局符号多次被定义，链接器会报告多次定义的错误信息。无论模块是否使用符号，**.ref** 伪指令总是为符号创建一个符号表入口，但是**.global** 伪指令只在实际使用该符号时才创建符号表入口。

一个符号定义为全局符号有两个原因：

- 如果符号在当前模块中没有定义（包括宏，复制或包含文件），那么**.global** 或**.ref** 伪指令告诉汇编器符号已在外部模块定义过。这样就可防止汇编器发出未定义引用的错误信息。链接时，链接器在其他模块中查找该符号的定义。
- 如果符号在当前模块中定义，那么**.global** 或**.def** 伪指令声明符号及其定义可以被其他模块在外部使用。这些类型的引用在链接时被确定。

例：显示4个文件：

file1.lst 和 **file3.lst** 是等效的。这两个文件都定义了符号 **INIT**，并使它们可供其他模块使用；两个文件都使用外部符号 **X**、**Y** 和 **Z**。**file1.lst** 使用**.global** 伪指令标识这些全局符号；**file3.lst** 文件使用**.ref**，**.def** 标识这些全局符号。

File2.lst 和 **file4.lst** 是等效的。这两个文件都定义了符号 **X**，**Y** 和 **Z**。并使它们可供其他模块使用。两个文件都使用了外部符号 **INIT**。**file2.lst** 用 **.global** 伪指令标识这些全局符号，**file4.lst** 用**.ref** 和**.def** 标识这些全局符号。

file1.lst:

```

1          ; 在文本文件中定义全局符号
2          .global INIT
3          ; 在 file2.lst 中定义全局符号
4          .global X, Y, Z
5 000000    INIT:
6 000000 0956    ADD ACC, #56h
7
8 000001 0000!    .word X
9              ;
10             ;
11             ;
12             .end
```

file2.lst:

```

1          ; 在本文件中定义全局符号
2          .global X, Y, Z
3          ; 在 file1.lst 定义全局符号
```

```

4          .global INIT
5      0001 X:      .set 1
6      0002 Y:      .set 2
7      0003 Z:      .set 3
8 000000 0000!     .word INIT
9          ;      .
10         ;      .
11         ;      .
12         .end

file3.lst:
1          ; 在本文件中定义全局符号
2          .def INIT
3          ; 在 file4.lst 中定义全局符号
4          .ref X, Y, Z
5 000000          INIT:
6 000000 0956      ADD ACC, #56h
7
8 000001 0000!     .word X
9          ;      .
10         ;      .
11         ;      .
12         .end

file4.lst:
1          ; 在本文件中定义全局符号
2          .def X, Y, Z
3          ; 在 file3.lst 中定义全局符号
4          .ref INIT
5      0001 X:      .set 1
6      0002 Y:      .set 2
7      0003 Z:      .set 3
8 000000 0000!     .word INIT
9          ;      .
10         ;      .
11         ;      .
12         .end

```

语法: **.if/.elseif/.else/.endif**

格式:

```

.if well-defined expression
.elseif well-defined expression
.else
.endif

```

描述: 下列伪指令提供条件汇编:

.if伪指令标记条件块的开始。定义明确的表达式 (*well-defined expression*) 是必需的参数。

- 如果表达式值为真（非零），那么汇编器汇编表达式后面的代码（直到遇见.elseif, .else 或.endif 伪指令为止）。
- 如果表达式值为假，那么汇编器汇编.elseif、.else 或.endif 伪指令后的代码。

.elseif 伪指令在.if 表达式为假（为 0 值），且.elseif 表达式为真（为非 0 值）时，汇编后面的代码块。当.elseif 表达式为假时，汇编器继续汇编下一个.elseif, .else 或是.endif 伪指令后的代码。**.elseif** 是可选的，可以使用多个.elseif 语句。如果.elseif 的表达式为假，且没有后跟的.elseif 语句，汇编器继续汇编.else 或.endif 后的代码。

.else 伪指令在.if 表达式和所有.elseif 表达式都为假（为 0 值）时，汇编其后面的代码块。这一伪指令是可选的；如果表达式为假且没有后跟.else 语句，汇编器继续汇编.endif 语句后的代码。

.endif 伪指令结束一个条件汇编代码块。

.elseif 和**.else** 伪指令可以同时使用，**.elseif** 伪指令可以多次使用。

有关关系运算符的资料，见第 3.9.4 节条件表达式。

例：给出条件汇编。

```

1      0001 SYM1      .set 1
2      0002 SYM2      .set 2
3      0003 SYM3      .set 3
4      0004 SYM4      .set 4
5
6      If_4: .if SYM4 = SYM2 * SYM2
7 000000 0004      .byte SYM4          ; 等于数值
8
9      .else
10     .byte SYM2 * SYM2      ; 不等于数值
11     .endif
12
13     If_5: .if SYM1 <= 10
14 000001 030A      .byte 10          ; 小于或等于
15     .else
16     .byte SYM1          ; 大于
17     .endif
18
19     If_6: .if SYM3 * SYM2 != SYM4 + SYM2
20     .byte SYM3 * SYM2      ; 不等于数值
21     .else
22 000002 0008      .byte SYM4 + SYM4      ; 等于数值
23     .endif
24
25     If_7: .if SYM1 = 2
26     .byte SYM1
27     .elseif SYM2 + SYM3 = 5
28 000003 0005      .byte SYM2 + SYM3
29     .endif

```

语法: **.int/.word**

格式:

```
.int value1 [, ... , valuen]  
.word value1 [, ... , valuen]
```

描述: **.int**和**.word**伪指令是等效的。它们将一个或多个值放入当前段中连续的16位域中。

*value*可以是绝对的或是可重定位的表达式。如果一个表达式是可重定位的,那么汇编器将产生重定位入口,它指向合适的符号,从而链接器可以重新调整(重定位)引用。这就允许用户用指向变量的指针或标号来初始化存储器。

用户可以在一行中使用很多的值。如果用户使用标号,那么该标号指向已初始化的第一个字。

在**.struct/.endstruct**序列中使用**.int**或**.word**时,**.int**或**.word**定义成员的大小,但不初始化存储器。有关**.struct/.endstruct**的详细资料,见第4.9节汇编过程使用符号的伪指令。

例1: 用**.int**伪指令来初始化字。

```
1  000000          .space 73h
2  000000          .bss PAGE, 128
3  000080          .bss SYMPTR, 3
4  000008 FF20 INST: MOV ACC, #056h
   000009 0056
5  00000a 000A      .int 10, SYMPTR, -1, 35 + 'a', INST
   00000b 0080-
   00000c FFFF
   00000d 0084
   00000e 0008'
```

例2: 用**.word**伪指令初始化字。符号WORDX指向保留的第一个字。

```
1  000000 0C80 WORDX:      .word 3200, 1 + 'AB', -0AFh, 'X'
   000001 4242
   000002 FF51
   000003 0058
```

语法: **.label**

格式:

```
.label symbol
```

描述: **.label**伪指令定义特定的符号(symbol),它指向当前段内的装载地址,而不是运行地址。汇编器创建的大部分段地址都可重定位。汇编器从0开始汇编每个段,链接器将它重定位到装载和运行时的地址。

在一些应用中,需要将一个段装载到一个地址,而运行在不同的地址。例如,用户可能要把性能要求严格的代码块装载到片外低速存储器以节省空间,然后把代码移到片内高速存储器来运行。

这样的段在链接时被分配两个地址：一个装载地址，一个运行地址。段内定义的所有符号被重定位以指向运行地址，以保证对段的引用（如跳转）是正确的。

`.label` 伪指令创建一个特殊标号指向装载地址。这一功能主要用来为重定位段的代码指定该段装载的位置。

例：表示装载时地址标号的使用。

```

        .sect ".EXAMP"
        .label EXAMP_LOAD    ; 段的装载地址
START:                                ; 段的运行地址
        <code>
FINISH:                                ; 段运行地址结束
        .label EXAMP_END    ; 段装载地址结束

```

有关链接器设定运行和装载地址的详细资料，见第 7.9 节指定一个段的运行地址。

语法：`.length/.width`

格式：

```

.length page length
.width page width

```

描述：`.length` 伪指令设置输出列表文件的页长度。它影响当前页和后续页。用户可以用另一个 `.length` 伪指令重新设置页的长度。

- ☐ 默认长度：60 行
- ☐ 最小长度：1 行
- ☐ 最大长度：32767 行

`.width` 伪指令设置输出列表文件的页宽度。它影响被汇编的下一行以及后续行。用户可以用另一个 `.width` 伪指令重新设置页的宽度。

- ☐ 默认宽度：80 个字符
- ☐ 最小宽度：80 个字符
- ☐ 最大宽度：200 个字符

宽度是指列表文件中的一个完整的行。行计数器值、SPC 值以及目标代码作为行宽度的一部分被计数。注释和源语句的其他部分如果超出页宽，则在列表中被截断。

汇编器不列出 `.width` 和 `.length` 伪指令。

例：页长度和页宽度被改变。

```

*****
** 页长度=65 行                               **
** 页宽度=85 字符                             **
*****
        .length 65
        .width 85
*****

```

```

**页长度=55 行                                     **
**页宽度=100 字符                                   **
*****
                .length 55
                .width 100

```

语法: **.list/.nolist**

格式:

```

.list
.nolist

```

描述: 这两条伪指令使用户能控制源程序列表的打印:

.list伪指令允许输出源程序列表。

.nolist伪指令禁止输出源程序列表, 直至遇到**.list**伪指令为止。**.nolist**伪指令可用于减少汇编编译时间和源程序列表长度。它可以用在宏定义中以禁止宏扩展列表。

汇编器不输出**.nolist**伪指令或出现在**.nolist**伪指令之后的源语句, 但是, 它继续增加行计数器的值。用户可以嵌套使用**.list/.nolist**伪指令。每一个**.nolist**需要一个和它匹配的**.list**来恢复列表。

默认设置, 列表文件中输出源程序列表, 汇编器就好像已使用了**.list**伪指令一样工作。

注意: 创建列表文件 (-l 选项)

当用户调用汇编器时, 如果不请求列表文件, 那么汇编器将忽略**.list**伪指令。

例: 描述**.copy**伪指令怎样插入来自另一个文件的源程序语句。在第一次遇到此伪指令时, 汇编器在列表文件中列出复制的源程序语句。第二次遇到此伪指令时, 因为汇编遇到**.nolist**伪指令, 所以汇编器不再列出复制的源程序语句。

注意: **.nolist**第二个**.copy**以及**.list**伪指令不出现在列表文件中。还要注意的, 即使当源语句未列出时, 行计数器仍增加。

copy.asm (源文件)	copy2.asm (复制文件)
<pre> .copy "copy2.asm" * Back in original file NOP .nolist .copy "copy2.asm" .list * Back in original file .string "Done" </pre>	<pre> *In copy2.asm (copy file) .word 32, 1 + 'A' </pre>

列表文件:

1

.copy "copy2.asm"

```

1          *In copy2.asm (copy file)
2 000000 0020          .word 32, 1 + 'A'
   000001 0042
2          * Back in original file
3 000002 7700 NOP
7          * Back in original file
8 000005 0044          .string "Done"
   000006 006F
   000007 006E
   000008 0065

```

语法: **.long/.xlong**

格式:

```

.long value1 [, ... , valuen]
.xlong value1 [, ... , valuen]

```

描述: **.long**和**.xlong**伪指令把一个或多个32位数放入当前段内连续的字中。最高位在前。**.long**伪指令将结果调整到长字边界, 而**.xlong**不做调整。

操作数`value`可以是绝对的或可重定位的表达式。如果表达式是可重定位的, 那么汇编器产生可重定位入口, 它指向合适的符号, 然后链接器可重定位该引用。这就允许用户用变量指针或标号来初始化存储器。

用户最多可以使用多达100个变量, 但是它们必须放在源语句的一行中。如果用户使用标号, 那么它指向已初始化的第一个字。

当用户在**.struct/.endstruct**序列中使用**.long**时, **.long**定义成员的大小, 但不初始化存储器。有关**.struct/.endstruct**的详细资料, 见第4.9节汇编过程使用符号的伪指令。

例: 表示**.long**和**.xlong**伪指令怎样初始化双字。

```

1 000000 ABCD      DAT1:  .long 0ABCDh, 'A' + 100h, 'g', 'o'
   000001 0000
   000002 0141
   000003 0000
   000004 0067
   000005 0000
   000006 006F
   000007 0000
2 000008 0000'      .xlong DAT1, 0AABBCCDDh
   000009 0000
   00000a CCDD
   00000b AABBB
3 00000c DAT2:

```

语法: **.loop/.break/.endloop**

格式:

```
.loop [well-defined expression]  
.break [well-defined expression]  
.endloop
```

描述: 这3条伪指令使用户能重复汇编代码块:

.loop伪指令开始重复代码块。可选表达式计算循环次数。如果没有表达式, 那么除非汇编器首先遇到表达式值为真(为非0值)或省略**.break**伪指令, 否则循环计数默认值为1024。

.break伪指令和其后表达式是可选的。当表达式为假(为0值)时, 循环继续; 当表达式为真(为非0值)或没有表达式时, 汇编器终止循环并汇编**.endloop**伪指令之后的代码。

.endloop伪指令终止重复代码块。当**.break**伪指令为真(为非0值)或当执行循环次数等于**.loop**所给定的循环次数时, 执行该伪指令。

例: 说明这些伪指令如何与**.eval**伪指令一起使用。

```

1          1          .eval    0,x
2          COEF      .loop
3          .word     x*100
4          .eval     x+1, x
5          .break    x = 6
6          .endloop
1          000000 0000 .word     0*100
1          .eval     0+1, x
1          .break    1 = 6
1          000001 0064 .word     1*100
1          .eval     1+1, x
1          .break    2 = 6
1          000002 00C8 .word     2*100
1          .eval     2+1, x
1          .break    3 = 6
1          000003 012C .word     3*100
1          .eval     3+1, x
1          .break    4 = 6
1          000004 0190 .word     4*100
1          .eval     4+1, x
1          .break    5 = 6
1          000005 01F4 .word     5*100
1          .eval     5+1, x
1          .break    6 = 6
```

语法: **.mlib**

格式:

```
.mlib ["filename"]
```


描述: **.mlib**伪指令向汇编器提供宏库的名字。宏库是包含宏定义的文件集合。这些文件由归档器组合成单个文件（称为库或归档库）。宏库的每一个成员包含一个宏定义，它对应于文件名。宏库成员必须是源文件（不是目标文件）。

宏库成员的文件名必须和宏名一样，它的扩展名必须是.asm。文件名必须遵从操作系统的约定。它可以包含在双引号内。用户可以指定完整的路径名（例如c:\dsp\macs.lib）。如果用户没有指定完整的路径名，那么汇编器在下述目录中搜索文件：

- 1) 包含当前源文件的目录；
- 2) 用汇编器-i 所有用选项命名的目录；
- 3) 用环境变量 A_DIR 指定的目录。

有关-i 选项和环境变量的详细资料，见第 3.4 节汇编器输入的替换目录命名。

当汇编器遇到.mlib 伪指令时，将打开库并创建库的目录表。汇编器把各个库成员的名字作为库的入口，输入到操作代码表中。这将重新定义任何已存在的操作代码或有相同名字的宏。如果这些宏之一被调用，那么汇编器将从库中提取项目并把它装入宏表中。汇编器用与扩展其他宏相同的方式扩展库的入口，但是它不把源代码放入列表之中。只有实际调用的宏才被从库中提取，且它们只被摘录一次。有关宏和宏库的详细资料，见第 5 章宏语言。

例: 创建一个宏库，定义两个宏 incl 和 decl。Incl.asm 文件包含 incl 的定义，decl.asm 包含 decl 的定义。

incl.asm	decl.asm
* Macro for incrementing incl .macro A ADD A, #1 .endm	* Macro for decrementing decl .macro A SUB A, #1 .endm

使用归档器创建宏库：

```
ar2000 -a mac incl.asm decl.asm
```

现在用户就可以用伪指令来引用宏库并定义宏incl和decl：

```

1          .mlib "mac.lib"
2
3          * Macro call
4 000000    incl AL
1 000000 9C01    ADD AL, #1
5
6          * Macro call
7 000001    decl AR1
1 000001 08A9    SUB AR1, #1
000002 FFFF
```

语法: **.mlist/mnolist**

格式:

```
.mlist
.mnolist
```

描述: 这两条伪指令使用户能控制列表文件中宏和可重复块扩展的列表:

.mlist伪指令在列表文件中允许宏和`.loop/.endloop`块的扩展。

.mnolist伪指令在列表文件中禁止宏和`.loop/.endloop`块的扩展。

默认设置为`.mlist`伪指令。

例: 定义一个命名为`STR_3`的宏。第一次调用宏时, 宏扩展没有被列出, 因为`.mnolist`伪指令被汇编。第二次调用宏时, 宏扩展被列出, 因为`.mlist`伪指令被汇编。

```

1          STR_3          .macro P1, P2, P3
2                          .string ":p1: ", ":p2: ", ":p3: "
3                          .endm
4
5 000000          STR_3 "as", "I", "am"
1 000000 003A      .string ":p1: ", ":p2: ", ":p3: "
    000001 0070
    000002 0031
    000003 003A
    000004 003A
    000005 0070
    000006 0032
    000007 003A
    000008 003A
    000009 0070
    00000a 0033
    00000b 003A
6 00000c 003A      .string ":p1: ", ":p2: ", ":p3: "
    00000d 0070
    00000e 0031
    00000f 003A
    000010 003A
    000011 0070
    000012 0032
    000013 003A
    000014 003A
    000015 0070
    000016 0033
    000017 003A
7
8
9 000018          .mnolist
10          STR_3 "as", "I", "am"
11          .mlist
1 000024          STR_3 "as", "I", "am"
    000024 003A      .string ":p1: ", ":p2: ", ":p3: "
    000025 0070
```

```

000026 0031
000027 003A
000028 003A
000029 0070
00002a 0032
00002b 003A
00002c 003A
00002d 0070
00002e 0033
00002f 003A
12 000030 003A      .string ":p1: ", ":p2: ", ":p3: "
000031 0070
000032 0031
000033 003A
000034 003A
000035 0070
000036 0032
000037 003A
000038 003A
000039 0070
00003a 0033
00003b 003A
13

```

语法: **.newblock**

格式:

.newblock

描述: **.newblock**伪指令取消对任何当前已定义的局部标号的定义。局部标号从本质上来说是暂时的;

.newblock伪指令使它们复位并结束其作用域。

当局部标号已被定义且（也许）使用之后，用户应当用**.newblock**伪指令将其复位。**.text**、**.data**和**.sect**伪指令也复位局部标号。定义在包含文件之内的局部标号在包含文件之外是无效的。局部符号的说明，见第3.8.2小节。

例: 说明局部标号\$1是如何被定义、复位，然后再次被定义的。

```

1      .ref ADDRA, ADDRb, ADDRc
2      0076 B      .set 76h
3
4      00000000    F800!      MOV    DP, #ADDRA
5
6      00000001    8500! LABEL1:  MOV    ACC, @ADDRA
7      00000002    1976      SUB    ACC, #B
8      00000003    6403      B      $1, LT
9      00000004    9600!      MOV    @ADDRb, ACC
10     00000005    6F02      B      $2, UNC
11

```

```

12      00000006      8500! $1      MOV      ACC, @ADDRA
13      00000007      8100! $2      ADD      ACC, @ADDRC
14                                     .newblock ;取消$1的定义,以便再一次使用
15
16      00000008      6402          B $1, LT
17      00000009      9600!          MOV @ADDRC, ACC
18      0000000a      7700 $1      NOP

```

语法: **.option**

格式:

.option option list

描述: **.option**伪指令为汇编器的输出列表指定几个选项。参数`option list`是用竖线分隔的选项表,每一个选项选一个列表特征。以下是有效的选项:

- A** 打开所有伪指令、数据、后续展开、宏、块的列表;
- B** 把`.byte`伪指令的列表限制在一行;
- D** 关闭某些伪指令的列表(执行一次`.drnolist`);
- L** 把`.long`伪指令的列表限制在一行内;
- M** 在列表中不输出宏扩展;
- N** 关闭列表(执行一次`.nolist`);
- O** 打开列表(执行一次`.list`);
- R** 重新设置B, L, M, T和W选项;
- T** 把将`.string`伪指令的列表限制在一行内;
- W** 把`.word`伪指令的列表限制在一行内;
- X** 产生符号的交叉引用列表(也可用`-x`选项来,调用汇编器以获得交叉引用列表)。

以上选项不区分字母的大小写。

例:说明如何将`.byte`, `.word`, `.long`和`.string`伪指令的列表限制在一行内。

```

1      *****
2      **  把.byte, .word, .long 和.string 的          **
3      **  伪指令的列表限制在一行                    **
4      *****
5
6      .option B, W, L, T
7      000000 00BD      .byte  -'C', 0B0h, 5
8      000004 CCDD      .long   0AABBCCDDh, 536 + 'A'
9      000008 15AA      .word   5546, 78h
10     00000a 0045      .string "Extended Registers"
11     *****
12     **  恢复到列表选项                              **
13     *****
14     .option R
15     00001c 00BD      .byte  -'C', 0B0h, 5

```

```

00001d 00B0
00001e 0005
16 000020 CCDD      .long    0AABBCCDDh, 536 + 'A'
000021 AABBB
000022 0259
000023 0000
17 000024 15AA      .word    5546, 78h
000025 0078
18 000026 0045      .string  "Extended Registers"
000027 0078
000028 0074
000029 0065
00002a 006E
00002b 0064
00002c 0065
00002d 0064
00002e 0020
00002f 0052
000030 0065
000031 0067
000032 0069
000033 0073
000034 0074
000035 0065
000036 0072
000037 0073

```

语法: **.page**

格式:

.page

描述: **.page**伪指令在列表文件中产生输出页。**.page**伪指令在源程序列表中不打印,但是汇编器在遇到该伪指令时使用行计数器增加。使用**.page**伪指令可把源程序列表分为逻辑段,以提高程序的可读性。

例: 说明**.page**伪指令如何使汇编器开始新的源程序列表页。

源文件:

```

        .title "**** Page Directive Example ****"
;
;
;
        .page

```

列表文件:

```

**** Page Directive Example **** PAGE 1
2
;

```



```

3          ; .
4          ; .
TMS320C2000 COFF Assembler Version x.xx Day Time Year
Copyright (c) xxxx-xxxx Texas Instruments Incorporated
**** Page Directive Example **** PAGE 2

```

语法: **.sblock**

格式:

```
.sblock ["section name"] [, ["section name"], ...]
```

描述: **.sblock**伪指令指定进入块调准的段。块调准是一种类似于页调准的地址调准机制，但是功能较弱。如果块小于一页，那么经过块调准的段保证不跨越页边界（64字）；如果它大于一页，那么将从页边界开始。这一伪指令仅允许对已初始化的段进行规格化，不适用于由**.usect**、**.bss**伪指令定义的未初始化段。段名（*section name*）是可选项，它包含在引号内。

例，**.text** 和 **.data** 段被指定进行块调准。

```

1 *****
2 **为 .tex 和 .data 段指定块调准 **
3 *****
4
5 .sblock .text, .data

```

语法: **.sect**

格式:

```
.sect "section name"
```

描述: **.sect**伪指令定义已命名段，该段类似设置为默认方式的**.text**和**.data**段一样使用。**.sect**伪指令开始把源代码汇编到已命名段。

段名（*section name*）标识汇编器把代码汇编到其中的段。有效的名字最多可以包含200个字符，且必须用双引号引起来。一个段名可以包含子段名，采用段名：子段名的形式。有关COFF段的详细资料，见第2章通用目标文件格式。

例：定义两个特殊用途的段**Sym_Defs**和**Vars**，并把代码汇编到这两个段中。

```

1          **开始汇编到 .text 段中**
2 000000    .text
3 000000 FF20    MOV ACC, #78h    ;汇编到 .text
   000001 0078
4 000002 0936    ADD ACC, #36h    ;汇编到 .text
5
6          ** 开始汇编到 Sym_Defs 段中 **
7 000000    .sect "Sym_Defs"
8 000000 CCCD    .float 0.        ;汇编到 Sym_Defs

```

```

      000001 3D4C
9    000002 00AA X:      .word 0Aah      ;汇编到 Sym_Defs
10   000003 FF10      ADD ACC, #X      ;汇编到 Sym_Defs
      000004 0002+
11
12                                ** 开始汇编到 Vars 段中**
13   000000      .sect "Vars"
14   0010 WORD_LEN      .set 16
15   0020 DWORD_LEN     .set WORD_LEN * 2
16   0008 BYTE_LEN      .set WORD_LEN / 2
17   0053 STR           .set 53h
18
19                                **恢复汇编到.text 段**
20   000003      .text
21   000003 0942 ADD     ACC, #42h      ; 汇编到.text 段
22   000004 0003      .byte 3, 4      ;汇编到.text 段
      000005 0004
23
24                                ** 恢复汇编到 Vars 段**
25   000000      .sect "Vars"
26   000000 000D      .field 13, WORD_LEN
27   000001 000A      .field 0Ah, BYTE_LEN
28   000002 0008      .field 10q, DWORD_LEN
      000003 0000
29

```

语法: **.set/.equ**

格式:

symbol .set value

描述: **.set**伪指令把一个常数赋值给一个符号,然后在汇编语言源文件中就可以用符号代替数。用户可以为常数和其他值取一个有意义的名字。

- **symbol** (符号) 必须出现在标号域。
- **value** (数) 表达式中的符号必须已在前面定义。这就是说,表达式中所有的符号必须是当前源模块中以前定义过的符号。

未定义的外部符号或在模块后面定义的符号不能用在表达式中。如果表达式是可重定位的,那么它指向的符号也是可重定位的。

表达式的值出现在列表的目标域。此值不是实际目标代码的一部分且不能被写到目标文件中。

例: 表示怎样用**.set**为符号赋值。

```

1      ****
2      ** 符号 AUX_R1 赋值给寄存器 AR1,      **
3      ** 用 AUX_R1 代替寄存器使用          **
4      ****

```

```

5          0001    AUX_R1    .set AR1
6 000000    28C1                MOV *AUX_R1, #56h
          000001    0056
7
8          ****
9          **  设置 INDEX 为一整数表达式                      **
10         **  用 INDEX 代替立即数使用                          **
11         ****
12         0035 INDEX    .set 100/2 +3
13 000002    0935                ADD ACC, #INDEX
14
15         ****
16         **  设置符号 SYMTAB 为一可重定位表达式,              **
17         **  用 SYM AB 代替可重定位操作数使用                  **
18         ****
19 000003    000A LABEL    .word 10
20         0004'    SYMTAB .set LABEL + 1
21
22         ****
23         **  设置符号 NSYMS 等同于符号 INDEX,                  **
24         **  用 NSYMS 代替 INDEX 使用                          **
25         ****
26         0035    NSYMS    .set INDEX
27 000004    0035                .word NSYMS

```

语法: **.space/.bes**

格式:

.space size in bits
.bes size in bits

描述: **.space**和**.bes**伪指令在当前段中保留size长度位的空间并对保留的空间赋值为0。
当用户把标号和**.space**伪指令一起使用时,标号指向被保留的第一个字,当用户把标号和**.bes**伪指令一起使用时,标号指向被保留的最后一个字。

例: 显示如何使用**.space**和**.bes**伪指令保留存储器空间。

```

1          ****
2          **  开始汇编到.text 段                                **
3          ****
4 000000    .text
5          ****
6          **  在.text 段中保留 0F0 位(15 字)                      **
7          **                                                                **
8          ****
9 000000    .space 0F0h
10 00000f    0100    .word 100h, 200h
          000010    0200

```

```

11      ****
12      **  开始汇编到.data 段          **
13      ****
14  000000      .data
15  000000 0049      .string "In .data"
      000001 006E
      000002 0020
      000003 002E
      000004 0064
      000005 0061
      000006 0074
      000007 0061
16      ****
17      **  在.data 段中保留 100 位;          **
18      **  RES_1 指向保留空间的第一个字          **
19      **                                          **
20      ****
21  000008      RES_1: .space 100
22  00000f 000F      .word 15
23      ****
24      **  在.data 段中保留 20 位;          **
25      **  RES_2 指向保留空间的最后一个字          **
26      **                                          **
27      ****
28  000011      RES_2: .bss 20
29  000012 0036      .word 36h
30  000013 0011"      .word RES_

```

语法: **.sslist/.ssnolist**

格式:

```

.sslist
.ssnolist

```

描述: 这两条伪指令使用户能控制列表文件中置换符号的扩展:

.sslist伪指令允许在列表文件中扩展置换符号。扩展行出现在实际源程序列表行的下面。

.ssnolist伪指令禁止在列表文件中扩展置换符号。

默认设置为列表文件中禁止所有置换符号的扩展。带#字符的行表示已扩展的置换符号。

例: 显示默认情况下禁止置换符号扩展列表的代码, 也显示被汇编的**.sslist** 伪指令指示汇编器列出置换符号的代码扩展。

```

1  00000000      .bss ADDR_X, 1
2  00000001      .bss ADDR_Y, 1
3  00000002      .bss ADDR_A, 1

```

```

4 00000003 .bss ADDRb, 1
5
6 ADD2 .macro parm1, parm2
7 MOV ACC, @parm1
8 ADD ACC, @parm2
9 MOV @parm2, ACC
10 .endm
11
12 00000000 ADD2 ADDRb, ADDRb
1 00000000 8500- MOV ACC, @ADDRb
1 00000001 8101- ADD ACC, @ADDRb
1 00000002 9601- MOV @ADDRb, ACC
13
14 .sslist
15 00000003 ADD2 ADDRb, ADDRb
1 00000003 8502- MOV ACC, @parm1
# MOV ACC, @ADDRb
1 00000004 8103- ADD ACC, @parm2
# ADD ACC, @ADDRb
1 00000005 9603- MOV @parm2, ACC
# MOV @ADDRb, ACC

```

语法: **.string/.pstring**

格式:

```

.string "string1" [, ... , "stringn"]
.pstring "string1" [, ... , "stringn"]

```

描述: **.string**和**.pstring**伪指令把来自字符串的8位字符放入当前段中。使用**.string**伪指令时, 每个8位字符占用一个16位的字; 使用**.pstring**伪指令时对数据打包, 所以每个字包含两个8位的字符。每一个**string**可以是:

- 由汇编器求值并当作 16 位有符号数来计算和处理的表达式。
- 包含在双引号内的字符串。字符串中的每一个字符代表一个单独的字节。

当使用**.pstring**伪指令时, 数被打包放入字中, 从字的最高位字节开始存放。未使用的字节填零。

任何大于 8 位的数都会被汇编器截断。用户最多可以使用 100 个操作数, 但它们必须能够放在源语句的一行内。

如果用户使用标号, 那么该标号指向被初始化的第一个字。

当用户在**.struct/.endstruct**序列中使用**.string**时, **.string**只定义成员的大小, 而不初始化存储器。有关**.struct/.endstruct**的详细资料, 见第 4.9 节汇编过程使用符号的伪指令。

例: 表示8位数放入当前段内的字中。

```

1 000000 0041 Str_Ptr: .string "ABCD"
  000001 0042
  000002 0043

```



```

    000003 0044
2
3 000004 0041      .string 41h, 42h, 43h, 44h
    000005 0042
    000006 0043
    000007 0044
4
5 000008 4175      .pstring "Austin", "Houston"
    000009 7374
    00000a 696E
    00000b 486F
    00000c 7573
    00000d 746F
    00000e 6E00
6
7 00000f 0030      .string 36 + 12

```

语法: **.struct/.endstruct/.tag**

格式:

[stag]	.struct	[expr]
[mem0]	<i>element</i>	[expr0]
[mem1]	<i>element</i>	[expr1]
.	.	.
.	.	.
.	.	.
[memn]	.tag stag	[exprn]
.	.	.
.	.	.
[memN]	<i>element</i>	[exprN]
[size]	.endstruct	
<i>label</i>	.tag	<i>stag</i>

描述: **.struct** 伪指令为数据结构定义中的元素指定符号偏移量。这使用户能把相似的数据元素组合在一起, 然后让汇编器计算元素的偏移量。这与C的结构体或Pascal的结构体相类似。

注意: **.struct** 伪指令不分配存储器。它仅创建可重复使用的符号模板。

.endstruct 伪指令结束结构的定义。

.tag 伪指令将结构特性赋给一个标号以简化表述, 还能定义包含其他结构的结构。**.tag** 伪指令不分配存储器。**.tag** 伪指令的结构名字 (*stag*) 必须是前面已

定义过的。

Stag 是结构的名称。它的值为结构的起始点。如果没有stag, 那么汇编器将结构中的成员放到全局符号表中, 对应值为它们相对于结构起始点的绝对偏移量。参数stag对.struct伪指令是可选参数, 但对.tag伪指令则是必需的。

Expr 是可选表达式, 表示结构起始点的偏移量。在默认情况下, 结构从0开始。

mem_n 是可选的结构成员标号。此标号是绝对的且等于距离结构起始点的当前偏移量。结构成员的标号不能定义为全局符号。

element 是下列说明符中的一个: .string, .byte, .word, .float, .tag或.field。除.tag外的所有伪指令都是典型的初始化存储器伪指令。这些伪指令如果出现在.struct后, 那么它们不分配存储器。因为使用时必须使用stag (如同定义中指出的那样), 所以.tag伪指令是一种特殊情况。

expr_n 是可选的用于说明被说明元素个数的表达式。默认值为1。通常认为.string型元素的大小是一个字, 而.field型元素是一位。

Size 指明结构总体大小, 为可选项。

注意: 可出现在.struct/.endstruct 序列中的伪指令只能是元素说明符, 条件汇编伪指令, 以及将成员的偏移量调准到字边界的.align 伪指令。空结构是非法的。

下例表示.struct, .tag以及.endstruct伪指令的不同用法。

例1:

```
REAL_REC      .struct           ; 结构体开始
NOM           .int             ; member1 = 0
DEN           .int             ; member2 = 1
REAL_LEN      .endstruct       ; real_len = 4
ADD ACC, @(REAL + REAL_REC.DEN) ; 访问结构体元素
.bss REAL, REAL_LEN           ; 分配 mem rec
```

例2:

```
CPLX_REC      .struct
REALI         .tag REAL_REC     ; stag
IMAGI         .tag REAL_REC     ; member1 = 0
CPLX_LEN      .endstruct       ; rec_len = 4
COMPLEX       .tag CPLX_REC     ; 指定结构体属性
ADD ACC, COMPLEX.REALI         ; 访问结构体
ADD ACC, COMPLEX.IMAGI
.bss COMPLEX, CPLX_LEN         ; 分配空间
```

例3:

```
                .struct         ; 无 stag 则分配为
X              .int            ; 全局符号
Y              .int            ; 创造了维数量
Z              .int
                .endstruct
```

例4:

```

BIT_REC      .struct                ; stag
STREAM        .string 64
BIT7          .field 7                ; bits1 = 64
BIT9          .field 9                ; bits2 = 64
BIT10         .field 10               ; bits3 = 65
X_INT         .int                    ; x_int = 67
BIT_LEN       .endstruct              ; length = 68
BITS          .tag BIT_REC
              ADD AC, @BITS .BIT7    ; 进入 acc
              AND ACC, #007Fh        ; 屏蔽掉无用数据位
              .bss BITS, BIT_REC

```

语法: **.tab**

格式:

.tab size

描述: **.tab**伪指令定义制表符的大小。在源语句输入中遇到的制表符在列表中将被转化为size个空格。默认的制表符大小是8个空格。

例: 下面的每一行中都包括一个标号, 它后面跟随一个NOP指令。

源文件:

```

; default tab size
NOP
NOP
NOP
.tab 4
NOP
NOP
NOP
.tab 16
NOP
NOP
NOP

```

列表文件:

```

1          ; 默认 tab 大小
2 000000 7700 NOP
3 000001 7700 NOP
4 000002 7700 NOP
5
7 000003 7700 NOP
8 000004 7700 NOP
9 000005 7700 NOP
10

```

```

12 000006 7700 NOP
13 000007 7700 NOP
14 000008 7700 NOP

```

语法: **.text**

格式:

.text

描述: **.text**伪指令通知汇编器开始汇编到.text段之中, 该段通常包含可执行代码。如果还没有代码被汇编到.text段之中, 那么段程序计数器被设置为0; 如果段内已经有代码, 那么段程序计数器为该段上一次的计数值。

.text段是默认段。因此, 除非用户规定另外一个段伪指令(.data或.sect), 否则, 在开始汇编时, 汇编器将把代码汇编到.text段中。有关COFF段的详细资料, 参见第2章。

例: 将代码汇编到.text和.data段中。.data段包含整型常数, .text段包含字符串。

```

1          *****
2          **  开始汇编到.data 段                      **
3          *****
4 000000          .data
5 000000 000A          .byte 0Ah, 0Bh
6 000001 000B
7
8          *****
9          **  开始汇编到.text 段                      **
10         *****
11 000000          .text
12 000000 0041 START: .string "A", "B", "C"
13 000001 0042
14 000002 0043
15 000003 0058 END:   .string "X", "Y", "Z"
16 000004 0059
17 000005 005A
18
19 000006 8100'      ADD ACC, @START
20 000007 8103'      ADD ACC, @END
21
22         *****
23         **  恢复汇编到.data 段                      **
24         *****
25 000002          .data
26 000002 000C          .byte 0Ch, 0Dh
27 000003 000D
28
29         *****
30         ** 恢复汇编到.data 段                      **
31         *****

```

```

26 000008      .text
27 000008 0051  .string "Quit"
    000009 0075
    00000a 0069
    00000b 0074
28

```

语法: **.title**

格式:

.title "string"

描述: **.title**伪指令输出印在每一个列表页头部的标题。源程序语句本身不输出, 但行计数器的值增加。

*string*是用双引号引起来的, 最多可达65个字符的标题。如果用户提供多于65个字符的标题, 那么汇编器将截断字符串, 并发出警告。

汇编器在跟随**.title**伪指令之后的页上和后续的页上输出指定标题, 直到另一个**.title**伪指令被处理为止。如果用户想在第一页打印标题, 那么第一条源语句必须包含**.title**伪指令。

例中, 第一页打印了一个标题, 后续页中打印另一个不同的标题。

源文件:

```

.title "**** Fast Fourier Transforms ****"
; .
; .
; .
.title "**** Floating-Point Routines ****"
.page

```

列表文件:

```

TMS320C2000 COFF Assembler Version x.xx   Day   Time   Year
Copyright (c) xxxx-xxxx   Texas Instruments Incorporated
**** Fast Fourier Transforms **** PAGE 1
2           ;           .
3           ;           .
4           ;           .
TMS320C2000 COFF Assembler Version x.xx   Day   Time   Year
Copyright (c) xxxx-xxxx   Texas Instruments Incorporated
**** Floating-Point Routines **** PAGE 2

```

语法: **.usect**

格式:

symbol .usect "section name", size in words [, blocking flag] [, alignment flag] [, type]

描述: `.usect`伪指令在未初始化的已命名段中为变量保留空间。此伪指令与`.bss`伪指令相类似,二者都是为数据保留空间,且不填入内容。`.usect`伪指令定义了可以放到存储器任何地方的附加段,该段与`.bss`段互相独立。

<i>symbol</i> (符号)	指向 <code>.usect</code> 伪指令所保留的第一个单元。它与为其保留空间的变量名相对应。
<i>section name</i> (段名)	必须包含在双引号中。这一参数给未初始化段命名。名字可以长达200个字符。段名可以包含子段的名称。用如下形式表示:段名:子段名。
<i>size in words</i>	定义在段内保留的字数。
<i>blocking flag</i>	可选参数。如果被指定为非零,那么此标志意味着将对该段进行块调准处理。块调准与页调准相似,均为一种地址分配机制,但其功能相对较弱。这意味着如果段的大小不足一页,那么保证不会越过页边界(64字)。如果段的大小超过一页,那么就从页边界开始。这种块调准适用于段,不适用于 <code>.usect</code> 伪指令所定义的内容。
<i>alignment flag</i>	可选参数。该标志使汇编器以长字为边界来分配地址。
<i>type</i>	可选参数。指定一个类型使汇编器产生适当的符号调试信息。参见第3.14节汇编语言变量的C类型符号调试(-mg选项)。

其他段伪指令(`.text`, `.data`, `.sect` 以及`.asect`)结束当前段并通知汇编器开始汇编到其他段中。但是,`.bss`和`.usect`伪指令不影响当前段。汇编器完成汇编`.bss`和`.usect`伪指令后,继续汇编到当前段。

被连续放置在存储器中的变量可以被定义到同一个指定的段中;为了实现这一点,可用相同的段名重复使用`.usect`伪指令。有关COFF段的详细资料,见第2章通用目标文件格式。

例:用`.usect`伪指令定义两个未初始化的已命名段, `var1`和`var2`。符号`ptr`指向`var1`段中保留的第一个字。符号`array`指向`var1`段内保留的100个字块中的第一个字;符号`dflag`指向`var1`段内保留的50个字块中的第一个字。符号`vec`指向`var2`段中保留的第一个字。

```

1      ****
2      **      汇编到.text 段      **
3      ****
4  000000      .text
5  000000 9A03      MOV AL, #03h
6
7      ****
8      **      在 var1 中保留一个字      **
9      ****
10  000000      ptr      .usect  "var1", 1
11
```

```

12      ****
13      **      在 var1 中保留 100 个字      **
14      ****
15 000001      array      .usect  "var1", 100
16
17 000001 9C03      ADD AL, #03h      ; 任然在 .text 段中
18
19      ****
20      **      在 var1 中保留 50 个字      **
21      ****
22 000065      dflag      .usect  "var1", 50
23
24 000002 08A9      ADD AL, #dflag ;任然在 .text 段中
25 000003 0065-
26
27      ****
28      **      在 var2 中保留 100 个字      **
29      ****
29 000000      vec      .usect  "var2", 100
30
31 000004 08A9      ADD AL, #vec      ; 任然在 .text 段中
32 000005 0000-
33
34      ****
35      **      声明外部符号 .usect      **
36      ****
36      global array

```

图 4.7 显示了如何在两个未初始化段 var1, var2 中保留空间。

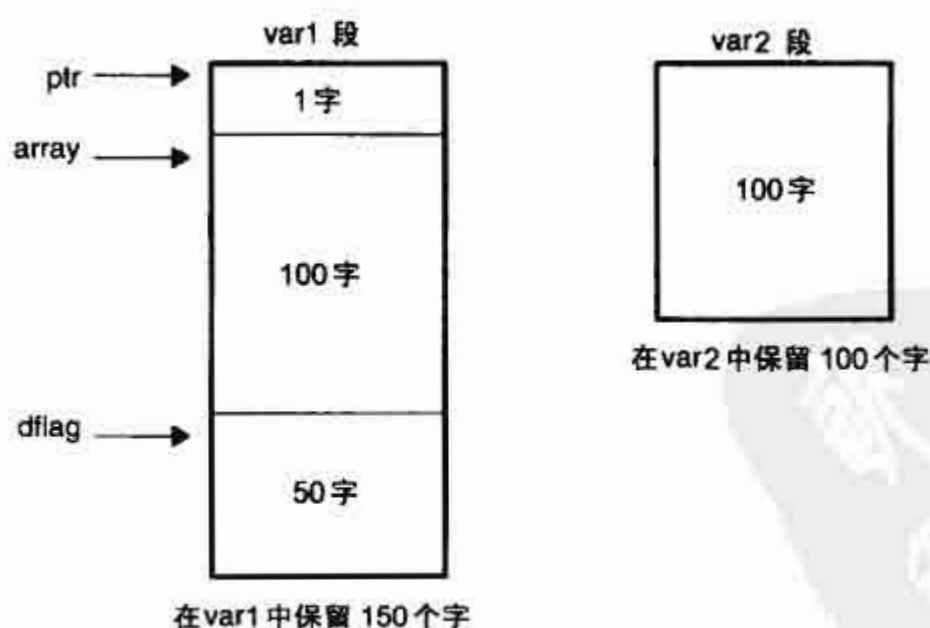


图 4.7 .usect 伪指令



第5章 宏语言

汇编器支持宏语言，它使用户能够创建自己的“指令”。这对一个程序需要多次执行一项特殊任务时特别有用。宏语言可以做到：

- 定义自己的宏或重新定义已有的宏
- 简化长汇编代码和复杂的汇编代码
- 访问归档器创建的宏库
- 在宏中处理字符串
- 控制列表的扩展

5.1 宏的使用

程序中常包含需要执行数次的子程序。对这段用源程序重复书写的程序块可以定义一个宏来代替它，那么就能在需要时重复执行这个程序块的地方通过直接调用宏来完成。这

样可以简化和缩短源程序的长度。

如果想多次调用宏，但每次又需要使用不同的数据。这时可以在宏中指定一些参数，当调用宏时，每次都能把不同的信息传递给宏。宏语言支持一个特殊的符号，称为置换符号，它专门用于宏。详情参见第5.3节宏参数/置换符号。

使用宏的三个步骤：

第一步：定义宏。在程序使用宏之前必须先定义宏。定义宏有如下两种方法：

- 可以在源程序的开始处或在`.copy/include`文件中定义。参见第 5.2 节定义宏。
- 可以在宏库中定义宏。宏库汇集了由归档器创建的归档格式的文件。归档文件（宏库）中的每个成员可以包含一个宏定义且宏的名字与成员名相同。使用`.mlib`伪指令就可以访问一个宏库。详情参见第 5.4 节宏库。

第二步：调用宏。用户定义了宏之后，在源程序中可以用宏的名字作助记符来调用宏。这就是宏调用。

第三步：扩展宏。当源程序调用宏时，汇编器就扩展宏。在扩展时，汇编器通过变量将参量传递给宏参数，用宏定义代替宏调用语句，然后再汇编源程序代码。若为默认设置，宏扩展会在文件列表中输出。也可以使用`.mnolist`伪指令关闭宏扩展列表输出。

当汇编器遇到宏定义时，它把宏名放在操作码表中。这就重新定义了以前定义的、与所遇到的宏具有相同名字的宏、库入口（library entry）、伪指令或指令助记符。这使用户能扩展伪指令和指令的功能，并增加新的指令。

5.2 定 义 宏

用户可以在程序的任何地方定义宏，但是必须在使用之前定义。宏可以在源程序的开始处定义或在`.copy/include`文件中定义，也可以在宏库中定义。详情参见第5.4节宏库。

宏可以嵌套定义，可以调用其他的宏，但一个宏的所有元素必须定义在同一个文件中。宏嵌套将在第 5.9 节递归和嵌套宏的使用中讨论。

宏定义是一串下列格式的源语句。

```
macname .macro [parameter1] [, ... , parametern]
    model statements or macro directives
    [.mexit]
.endm
```

macname	宏名。用户必须把名字放在源程序语句的标号域。有效的宏名最多为128个字符。汇编器把宏名放在内部操作码表中，取代具有相同名字的任何指令或以前的宏定义。
.macro	伪指令，它标识作为宏定义第一行的源程序语句。用户必须把 <code>.macro</code> 放在操作码域。

<i>parameter1,</i>	在macro伪指令中以操作码出现的置换符号, 是可选项。
<i>Parametern</i>	这些参数将在第5.3节宏参数/置换符号中讨论。
<i>model statements</i>	每次执行宏调用时执行的指令或汇编伪指令。
<i>macro directives</i>	用于控制宏扩展。
.mexit	功能与goto .endm相同的伪指令。当测试证实宏扩展失败时, .mexit伪指令起作用。
.endm	结束宏定义伪指令。

例 5.1 一个宏的定义、调用和扩展

```

1          *      add3 arg1, arg2, arg3
2          *      arg3 = arg1 + arg2 + arg3
3
4          add3    .macro P1, P2, P3, ADDR
5
6                      MOV ACC, P1
7                      ADD ACC, P2
8                      ADD ACC, P3
9                      ADD ACC, ADDR
10         .endm
11
12         .global abc, def, ghi, adr
13
14 000000      add3 @abc, @def, @ghi, @adr
1
1          000000 E000!      MOV ACC, @abc
1          000001 A000!      ADD ACC, @def
1          000002 A000!      ADD ACC, @ghi
1          000003 A000!      ADD ACC, @adr
15
16
No Errors, No Warnings

```

如果用户想在宏定义中包含注释, 但又不想让这些注释出现在宏扩展中, 那么在注释之前使用惊叹号“!”。如果用户想让注释出现在宏扩展中, 那么在注释前应使用星号“*”或分号。关于宏注释参见第 5.7 小节在宏中产生信息。

5.3 宏参数/置换符号

如果用户想多次调用宏, 但每次又需要使用不同的数据, 那么用户可以在宏的内部指定参数。宏语言支持被称为置换符号的特殊符号, 它专门用于宏参数。

宏参数就是置换符号, 它是用字符串表示的。这些符号也可以在宏之外使用, 使字符串等同于符号名, 参见第 3.8.6 小节置换符号。

有效的置换符号可长达 128 个字符而且必须以字母开始。其余的字符可以是字母和数

字、下划线 (_)、美元符号 (\$) 的组合。

置换符号在宏定义时作为宏参数放到宏内部。每个宏可以定义多达 32 个局部置换符号，其中包括用 .var 伪指令定义的置换符号。关于 .var 伪指令参见第 5.3.6 在宏中作为局部变量的置换符号。

在宏扩展时，汇编器将自变量传递给宏参数，每个自变量用字符串表示，它被指定给相应的宏参数。如果宏参数没有相应的自变量就放置一个空字符串；如果自变量的数目超过宏参数的数目，那么所有剩余的自变量等效为一个字符串全部赋给最后一个宏参数。

如果用户把自变量表传递给一个宏参数，或者把逗号或分号传递给一个宏参数，那么必须用引号把它们括起来。

在汇编过程使用的符号，汇编器首先用字符串代替宏参数/置换符号，然后再把源代码翻译成机器码。

例 5.2 表示具有可变自变量数目的宏扩展

```
Macro definition:
.Parms .macro a,b,c
;      a = :a:
;      b = :b:
;      c = :c:
      .endm

Calling the macro:
.Parms 100,label
;      a = 100
;      b = label
;      c = " "

.Parms 100, , x
;      a = 100
;      b = " "
;      c = x

.Parms ""string"",x,y
;      a = "string"
;      b = x
;      c = y

.Parms 100,label,x,y
;      a = 100
;      b = label
;      c = x,y

.Parms "100,200,300",x,y
;      a = 100,200,300
;      b = x
;      c = y
```

5.3.1 定义置换符号的伪指令

用户可以用 .asg 和 .eval 伪指令操作置换符号。

□ .asg 伪指令把字符串赋予置换符号

.asg 伪指令格式：

```
.asg [""]character string[""], substitution symbol
```

引号是可选择项，如果没有引号，那么汇编器读第一个逗号前的字符，并去掉前后的空格。无论哪种情况，汇编器读字符串的同时就将其赋予置换符号。

例 5.3 用.asg 伪指令为置换符指定字符串

```
.asg "A4", RETVAL          ; 返回值
.asg "B14", PAGEPTR        ; 全局页指针
.asg ""Version 1.0"", version
.asg "p1, p2, p3", list
```

□ .eval 伪指令完成置换符的数运算

.eval伪指令格式:

.eval well-defined xpression, substitution symbol

.eval 伪指令计算表达式的值并将字符串的求值结果赋值给置换符号。如果表达式未定义, 汇编器会产生错误信息并将一个空字符赋给置换符号。

例 5.4 用.eval 伪指令实现有关置换符的计算

```
.asg 1, counter
.loop 100
.word counter
.eval counter + 1, counter
.endloop
```

在例5.4中用.eval伪指令(.eval 1 counter)代替.asg而不会改变输出。在类似于此的简单情况中, .eval伪指令和.asg伪指令可以互换。如果要计算表达式的值就必须使用.eval。 .asg伪指令仅为置换符指定字符串, 而.eval伪指令计算表达式的值, 然后将所求字符串的结果赋给置换符。

5.3.2 内置置换符号函数

以下的内置置换符号函数可以让用户处理有关置换符号的字符串值的基本问题。这些函数总是返回一个值, 并可以用于表达式中。内置置换符号函数在条件汇编表达式中特别有用。这些函数的参数为置换符号或字符串常量。

函数定义见表 5.1, a 和 b 表示置换符号参数或字符串常量。*String* 项指的是参数的字符串值, ch 表示一个字符串常量。

表 5.1 函数及其返回值

函 数	返 回 值
\$symlen(a)	字符串 a 的长度
\$symcmp(a, b)	若 $a < b$, 则 < 0 ; 若 $a = b$, 则 $= 0$; 若 $a > b$, 则 > 0
\$firstch(a, ch)	在字符串 a 中第一次出现字符常量 ch 的位置
\$lastch(a, ch)	在字符串 a 中最后一次出现字符常量 ch 的位置
\$isdefed(a)	1 字符串 a 已在符号表中被定义 0 字符串 a 未在符号表中被定义
\$ismember(a, b)	把表 b 的第一个元素赋予字符串 0 b 为空字符串

续表

函 数	返 回 值
\$iscons(<i>a</i>)	1 字符串 <i>a</i> 为二进制常量 2 字符串 <i>a</i> 为八进制常量 3 字符串 <i>a</i> 为十六进制常量 4 字符串 <i>a</i> 为字符常量 5 字符串 <i>a</i> 为十进制常量
\$isname(<i>a</i>)	1 字符串 <i>a</i> 为有效符号名 0 无效符号名
\$isreg(<i>a</i>) ⁺	1 字符串 <i>a</i> 为有效预定义寄存器名 0 字符串 <i>a</i> 为无效预定义寄存器名

有关预定义寄存器名，请参见第3.8.5小节预定义符号常量

例 5.5 内置置换符号函数的应用

```

1      global x, label
2      .asg label, ADDR ; ADDR = label
3      .if ($symcmp(ADDR, "label") = 0) ; 计算是否为真
4 000000 8000! SUB ACC, @ADDR
5      .endif
6      .asg "x, y, z", list           ; list = x, y, z
7      .if ($ismember(ADDR, list))   ; ADDR = x list =y,z
8 000001 8000! SUB ACC, @ADDR
9      .endif
No Errors, No Warnings

```

5.3.3 递归的置换符号

当汇编器遇到一个置换符号，它总是试图代之以相应的字符串。如果该字符串也是一个置换符号，汇编器就再作一次代换。汇编器会一直作代换直到遇到一个不是置换符号的标记，或是遇到一个在此求值期间已遇到的置换符号。

例5.6中x被z代替，z被y代替，y被x代替。汇编器认为这是一个无限的递归并停止替换。

例 5.6 递归置换

```

1      .global x
2      .asg "x",z ; 声明 z 和指定 z = "x"
3      .asg "z",y ; 声明 y 和指定 y= "z"
4      .asg "y",x ; 声明 x 和指定 x = "y"
5 000000 FF10 ADD ACC, x
000001 0000!
6

```

5.3.4 强制置换

在某些情况下，汇编器不能识别有些置换符号。强制置换算子就是在符号前后设置一

对冒号，使用户可以强制对一个符号字符串进行置换。只需将需要强制置换的符号用冒号括起，符号和冒号之间不能有空格。

强制置换算子的格式：

```
:symbol:
```

汇编器在扩展其他置换符之前，先扩展用冒号括起的置换符号。

用户只能在宏中使用强制置换算子，但不能在一个强制置换算子中嵌套另一个强制置换算子。

例 5.7 使用强制置换算子

```
force      .macro x
            .loop 8
PORT:x:    .set x*4
            .eval x+1, x
            .endloop
            .endm
            .global portbase

force 0
```

前述的代码产生下列源代码：

```
PORT0      .set 0
PORT1      .set 4
.
.
.
PORT7      .set 28
```

5.3.5 访问下标置换符号的单个字符

在宏内，用户可以用下标置换符号访问置换符号的单个字符（子串）。为了清楚起见，用户必须使用强制置换算子。

有两种方法访问子串：

- :符号（定义明确的表达式）

这种下标方法是对单个字符的字符串求值

- :符号（定义明确的表达式1，定义明确的表达式2）

在这个方法中，定义明确的表达式1表示子串的起始位置，定义明确的表达式2表示子串的长度。用户可以指定子串的起始位置和字符串的长度。子串字符的索引从1开始，而不是0。

例5.8和例5.9用下标置换符号内置置换符号函数。

例 5.8 使用下标置换符号重定义一个指令

```
ADDX      .macro ABC
            .var    TMP
            .asg     :ABC(1): ,TM
```

```

        .if      $symcmp(TMP, "#") = 0
        ADD      ACC, ABC
        .else
        .emsg     "Bad Macro Parameter"
        .endif
        .endm

```

ADDX #100 ;宏调用

在例5.8中，用下标置换符号重新定义ADDX指令以便它能处理立即数。

例 5.9 使用下标置换符号寻找子串

```

substr .macro      start, strg1, strg2, pos
        .var        len1, len2, i, tmp
        .if          $symlen(start) = 0
        .eval        1, start
        .endif
        .eval        0, pos
        .eval        start, i
        .eval        $symlen(strg1), len1
        .eval        $symlen(strg2), len2
        .loop
        .break       i = (len2 - len1 + 1)
        .asg         ":strg2(i, len1): ", tmp
        .if          $symcmp(strg1, tmp) = 0
        .eval        i, pos
        .break
        .else
        .eval        i + 1, i
        .endif
        .endloop
        .endm
        .asg         0, pos
        .asg         "ar1 ar2 ar3 ar4", regs
        substr       1, "ar2", regs, pos
        .word        pos

```

在例 5.9 中使用下标置换符号寻找子串 strg1 在子串 strg2 中的开始单元。置换符号 pos 为子串 strg1 的指定单元。

5.3.6 在宏中作为局部变量的置换符号

如果用户想在一个宏中把置换符号作为局部变量使用，那么可以使用 **.var** 伪指令为每个宏定义多达 32 个局部宏置换符号（包括参数）。**.var** 伪指令用空字符串的初始值创建临时置换符号。这些符号不会作为参数传递，而且会在扩展后丢失。

```
.var sym1 [,sym2, ...,symn]
```

.var 伪指令见例 5.8 和例 5.9 中的应用。

5.4 宏 库

定义宏的一种方法就是创建宏库。宏库是包含宏定义的文件集。用户必须使用归档器把这些文件或成员汇集成一个文件（称为归档文件）。在宏库中的文件必须是未汇编的源程序文件。宏名和成员名必须相同，而且宏文件的文件扩展名必须是.asm。例如：

宏 名	宏库中的文件名
simple	simple.asm
add3	add3.asm

用户可以用.mlib 汇编伪指令访问宏库。

格式：

```
.mlib ["filename"]
```

当汇编器遇到.mlib 伪指令时，就打开由 *filename* 命名的宏库并创建一个库的目录表。汇编器把各成员名放入操作码表作为库入口，这样就重新定义已有的具有相同名字的操作码和宏。如果其中一个宏被访问，汇编器就从库中提取入口信息把它装入宏表。

汇编器扩展宏库入口的方式与扩展宏的方式相同（汇编器如何扩展宏参见第 5.1 节宏的使用）。可以用.mlist 伪指令控制宏库入口扩展列表。关于.mlist 伪指令的详情参见第 5.8 节在宏中使用条件汇编用伪指令格式化输出列表和有关.mlist 的说明。只有真正从宏库中调用宏时宏才被提取，且只被提取一次。

用户可以用归档器创建宏库，宏库所包含的这些文件在一个归档文件中。除了希望宏库包含的宏定义外，宏库与其他任何归档文件没什么不同。汇编器只是要宏库中的宏定义；而把目标代码和各种源程序混在库中可能会产生不合要求的结果。关于创建一个宏库归档文件请参阅第 6 章归档器。

5.5 在宏中使用条件汇编

条件汇编伪指令包括.if/.elseif/.else/.endif 和.loop/.break/.endloop。它们可以嵌套使用，嵌套层最多可达 32 层。条件块的格式为：

```
.if well-defined expression  
[.elseif well-defined expression]  
[.else]  
.endif
```

在条件汇编中,elseif 和.else 伪指令是可选项。在条件汇编块中,elseif 伪指令可以多次使用。当.elseif 和.else 伪指令被省略且.if 表达式为假（为 0 值）时,汇编器继续汇编.endif 伪指令后面的代码。

.loop/.break/.endloop 伪指令使用户能重复汇编一个代码块。重复的块格式为:

```
.loop [well-defined expression]
[.break [well-defined expression]]
.endloop
```

.loop 伪指令的表达式计算循环次数。如果省略表达式,则循环次数的默认值为 1024,除非汇编器遇到.break 伪指令的表达式为真（为非 0 值）。

在重复汇编时,.break 伪指令和它的表达式为可选项。如果表达式求值结果是假,循环继续;当.break 伪指令的表达式求值结果是真或省略表达式,汇编器结束循环。当循环中止后,汇编器继续汇编.endloop 伪指令后面的代码。

例 5.10、例 5.11、例 5.12 中为.loop/.break/.endloop 伪指令的嵌套条件汇编以及在条件汇编代码块中使用内置置换符号函数。

例 5.10 .loop/.break/.endloop 伪指令

```
.asg 1,x
.loop
.break (x == 10)    ; 如果 x == 10, 退出循环
.eval x+1,x
.endloop
```

例 5.11 嵌套条件汇编伪指令

```
.asg 1,x
.loop
.if (x == 10)        ; 如果 x == 10, 退出循环
.break              ; 强制退出
.endif
.eval x+1,x
.endloop
```

例 5.12 在条件汇编代码块中使用内置置换符号函数

```
MACK3 .macro src1, src2, sum, k
; sum = sum + k * (src1 * src2)
.if k = 0
MOV T,#src1
MPY ACC,T,#src2
MOV DP,#sum
ADD @sum,AL
.else
MOV T,#src1
MPY ACC,T,#k
MOV T,AL
MPY ACC,T,#src2
MOV DP,#sum
```

```

        ADD @sum, AL
        .endif
        .endm
.global A0, A1, A2
MACK3 A0, A1, A2, 0
MACK3 A0, A1, A2, 100

```

详细说明参见第4.8节条件汇编伪指令。

5.6 在宏中使用标号

汇编语言程序中的所有标号必须是惟一的。包括宏中的标号。如果宏被多次扩展，那么标号就多次被定义。多次定义一个标号是非法的。为了让标号是惟一的，宏语言提供了一种在宏中定义标号的方法。方法很简单，在标号后接一个问号（?），下一个周期汇编器就用一个惟一的数字代替这个问号（?）。当宏扩展时，在列表文件不会看到这个数字。用户的标号随同（?）一起出现，在宏定义中也一样。不能声明这个标号为全局量。定义惟一标号的格式为：

label?

对于有惟一后缀标号，超长的标号会被截断。若宏被扩展 10 次以下，最长的标号是 126 个字符。如果宏被扩展 10~99 次，最长的标号是 125 个字符。交叉列表文件显示了具有惟一后缀的标号。用带-x 的选项调用汇编器就可得到交叉列表文件。

例 5.13 在宏中生成惟一的标号

1		min	.macro x,y,z
2			
3			MV y,z
4			CMPLT x,y,y
5		[y]	B 1?
6			NOP 5
7			MV x,z
8		1?	
9			.endm
10			
11			
12	00000000		MIN A0,A1,A2
1			
1	00000000	010401A1	MV A1,A2
1	00000004	00840AF8	CMPLT A0,A1,A1
1	00000008	80000292	[A1] B 1?
1	0000000c	00008000	NOP 5
1	00000010	010001A0	MV A0,A2
1	00000014	1?	
	LABEL		VALUE DEFN REF

.TMS320C60	00000001	0	
.tms320C60	00000001	0	
1\$1\$	00000014'	12	12

5.7 在宏中产生信息

汇编器支持的三个伪指令使用户能够在汇编时定义自己的错误和警告信息。当用户需要创建自己的特殊信息时，这三个伪指令尤其有用。列表文件的最后一行显示了错误和警告的个数。警告用户程序代码存在问题，对程序调试特别有用。

.emsg 传送错误信息到列表文件。**.emsg** 伪指令与汇编器产生错误信息的方式相同，增加错误个数并阻止生成目标文件。

.mmsg 传送汇编时的信息到列表文件。**.mmsg** 伪指令的功能与**.emsg** 伪指令相同。但它不设置错误计数，也不阻止生成目标文件。

.wmsg 传送警告信息到列表文件。**.mmsg** 伪指令的功能与**.emsg** 伪指令相同，但它要增加错误个数却不阻止生成目标文件。

宏注释是出现在宏定义中的注释，但不在宏扩展中显示。第一列的惊叹号(!)用来识别宏注释。如果想在宏扩展中出现注释，可以用分号(;)和星号(*)引导注释。

例 5.14 在宏中的用户信息以及在宏扩展中不会出现的宏注释。

例 5.14 在宏中产生信息

```

1      testparam .macro x, y
2      !
3      ! This macro checks for the correct number of parameters.
4      ! It generates an error message if x and y are not present.
5      !
6      ! The first line tests for proper input.
7      !
8      .if ($symlen(x) == 0)
9      .emsg "ERROR --missing parameter in call to TEST"
10     .mexit
11     .else
12     MOV     ACC, #2
13     MOV     AL, #1
14     ADD     ACC, @AL
15     .endif
16     .endm
17
18 000000    testparam 1, 2
1      .if     ($symlen(x) == 0)
1      .emsg "ERROR --missing parameter in call to TEST"
1      .mexit
1      .else
1      000000 FF20 MOV     ACC, #2
1      000001 0002

```



```

1      000002 9A01 MOV    AL, #1
1      000003 A0A9 ADD    ACC, @AL
1      .endif

```

5.8 用伪指令格式化输出列表

宏、置换符号和条件汇编伪指令可以隐藏信息。为了在用户需要的时候看见这些隐藏信息，宏语言支持扩展列表的能力。

默认设置方式，汇编器显示宏扩展和在列表输出文件中的条件为假的代码块。可以在用户的列表文件中关闭或打开这个列表。4 组伪指令使用户能够控制的这些信息列表。

□ 宏和循环扩展列表

.mlist 扩展宏和.loop/.endloop块。**.mlist**伪指令输出在这些块中遇到的所有代码。

.mnolist 禁止扩展宏和.loop/.endloop块。

对于宏和循环扩展列表，默认设置为**.mlist**。

□ 条件为假代码块列表

.fclist 使汇编器在列表文件中包括所有的条件为假代码块。如像在源程序代码中一样，条件块会准确地出现在列表中。

.fcnolist 禁止条件为假代码块列表。只有在条件块中实际汇编的代码才出现在列表中。**.if**、**.elseif**、**.else**和**.endif**不出现在列表中。

对于条件为假代码块列表，默认设置为**.fclist**。

□ 置换符号扩展列表

.sslist 在列表中扩展置换符号。这对调试置换符号很有用。扩展行出现在实际的源程序行的下面。

.ssnolist 在列表中关闭置换符号的扩展。

对于置换符号扩展列表，默认设置为**.ssnolist**。

□ 伪指令列表

.drlist 使汇编器在列表文件中输出所有的伪指令行。

.drnolist 在列表文件中禁止输出某些伪指令。这些伪指令是：**.asg**、**.eval**、**.var**、**.sslist**、**.mlist**、**.fclist**、**.ssnolist**、**.mnolist**、**.fcnolist**、**.emsg**、**.wmsg**、**.mmsg**、**.length**、**.width**、和**.break**。

对于伪指令列表，默认设置为**.drlist**。

5.9 递归和嵌套宏的使用

宏语言支持递归和嵌套宏调用。这意味着在一个宏定义中可以调用其他宏，可以嵌套多达 32 层的宏。当使用递归宏时，用户实际上是从它自己的定义中访问宏(宏的自身调用)。

当创建递归和嵌套宏时，应该密切关注传递给宏参数的自变量，因为汇编器对参数是进行动态观测的。访问宏意味着利用所调用宏的环境。

例 5.15 显示宏的嵌套。在 out_block 宏中的 y 隐藏在 in_block 宏的 y 中，来自 out_block 块中的 x 和 z 是 in_block 可访问的宏。

例 5.15 使用嵌套宏

```
in_block    .macro y,a
            .                                ; y,a 是透明的，x,z 来自于调用宏
            .endm
out_block   .macro x,y,z
            .                                ; x,y,z 是透明的
            .
            in_block x,y    ; x 和 y 作为自变量的宏调用
            .
            .endm
out_block   ; 宏调用
```

例 5.16 递归宏。fact 宏产生计算 n 的阶乘所必需的汇编代码，其中 n 为立即数。其结果存放到 A1 寄存器中。fact 宏通过调用 fact1（也就是递归调用自己）完成这些功能。

例 5.16 使用递归宏

```
1          .fcnolist
2
3          fact    .macro N, loc
4
5              .if N < 2
6              MOV @LOC, #1
7              .else
8              MOV @LOC, #N
9
10
11              .eval N-1, N
12              fact1 temp
13
14              .endif
15              .endm
16
17          fact1   .macro
18              .if N < 1
19              MOV @T, @LOC
20              MPYB @P, @T, #N
21              MOV @LOC, @P
22              MOV ACC, @LOC
23              .eval N - 1, N
24          fact1
25
26              .endif
27              .endm
```

5.10 宏伪指令汇总

以下的伪指令可以与宏一起使用。`.macro`、`.mexit`、`.endm` 和 `.var` 伪指令仅在宏中有效。其余的伪指令就是一般汇编语言伪指令。

表 5.2 创建宏

助记符和格式	描 述
<code>.endm</code>	结束宏定义
<code>macname .macro [parameter₁] [, ... , parameter_n]</code>	由 <code>macname</code> 定义宏
<code>.mexit</code>	跳转到 <code>.endm</code>
<code>.mlib filename</code>	识别包含宏定义的库

表 5.3 处理置换符号

助记符和格式	描 述
<code>.asg ["]character string["], substitution symbol</code>	把字符串分配给置换符号
<code>.eval well-defined expression, substitution symbol</code>	完成有关数字置换符号的算数运算
<code>.var sym1 [, sym2, ... , symn]</code>	定义局部宏符号

表 5.4 条件汇编

助记符和格式	描 述
<code>.break [well-defined expression]</code>	汇编可重复块的选项
<code>.endif</code>	结束条件汇编
<code>.endloop</code>	结束循环块的汇编
<code>.else</code>	条件汇编块选项
<code>.elseif well-defined expression</code>	条件汇编块选项
<code>.if well-defined expression</code>	条件汇编开始
<code>.loop [well-defined expression]</code>	循环块汇编开始

表 5.5 产生汇编时的信息

助记符和格式	描 述
<code>.emsg</code>	传送错误信息到标准输出设备
<code>.mmsg</code>	传送汇编时的信息到标准输出设备
<code>.wmsg</code>	传送警告信息到标准输出设备

表 5.6 格式化列表

助记符和格式	描 述
<code>.fclist</code>	允许条件为假的代码块列表（默认设置）
<code>.fcnolist</code>	禁止条件为假的代码块列表

续表

助记符和格式	描 述
.mlist	允许宏列表（默认设置）
.mnolist	禁止宏列表
.sslist	允许扩展置换符号列表
.ssnolist	禁止扩展置换符号列表（默认设置）

用补版器补版时，补版器会自动识别补版位置，并自动补版。

补版器会自动识别补版位置，并自动补版。



第6章 归档器

TMS320C28x™ 归档器允许用户将几个单独的文件组合在一起，形成一个档案文件。例如，用户可以汇集几个宏组成宏库。汇编器在库中搜索并使用的称为宏的库成员的源文件。用户也可以使用归档器汇集一组目标文件，组成目标库。链接器将链接包含目标库的成员。归档器允许用户用删除、替换、摘录或增加库成员的方式来修改库。

6.1 归档器概述

用户可以建立任何类型的库。汇编器和链接器都接受归档器形成的库作为输入；汇编器可以使用包含源文件的库，而链接器可以使用包含目标文件的库。

归档器最有用的应用之一是建立目标模块库。例如，用户可以编写几个算术子程序，汇编后归档器将目标文件汇集起来，形成一个逻辑组。然后用户就可以指定目标库作为链接器的输入。链接器搜索库和库成员，以确定其成员的外部引用。

用户也可以使用归档器建立宏库。创建几个源文件，每个文件包含一个宏，用归档器

将这些宏集中成一个函数组。用户可以在汇编时使用.mlib 伪指令来指定宏库，并在宏库中搜索用户要调用的宏。在第 5 章中已详细讨论了宏和宏库的内容，本章则解释如何使用归档器来建立宏库。

6.2 软件开发流程中归档器的作用

图 6.1 显示了软件开发过程中归档器的作用。阴影部分强调了最普通的归档器开发途径。汇编器和链接器都接受库作为输入。

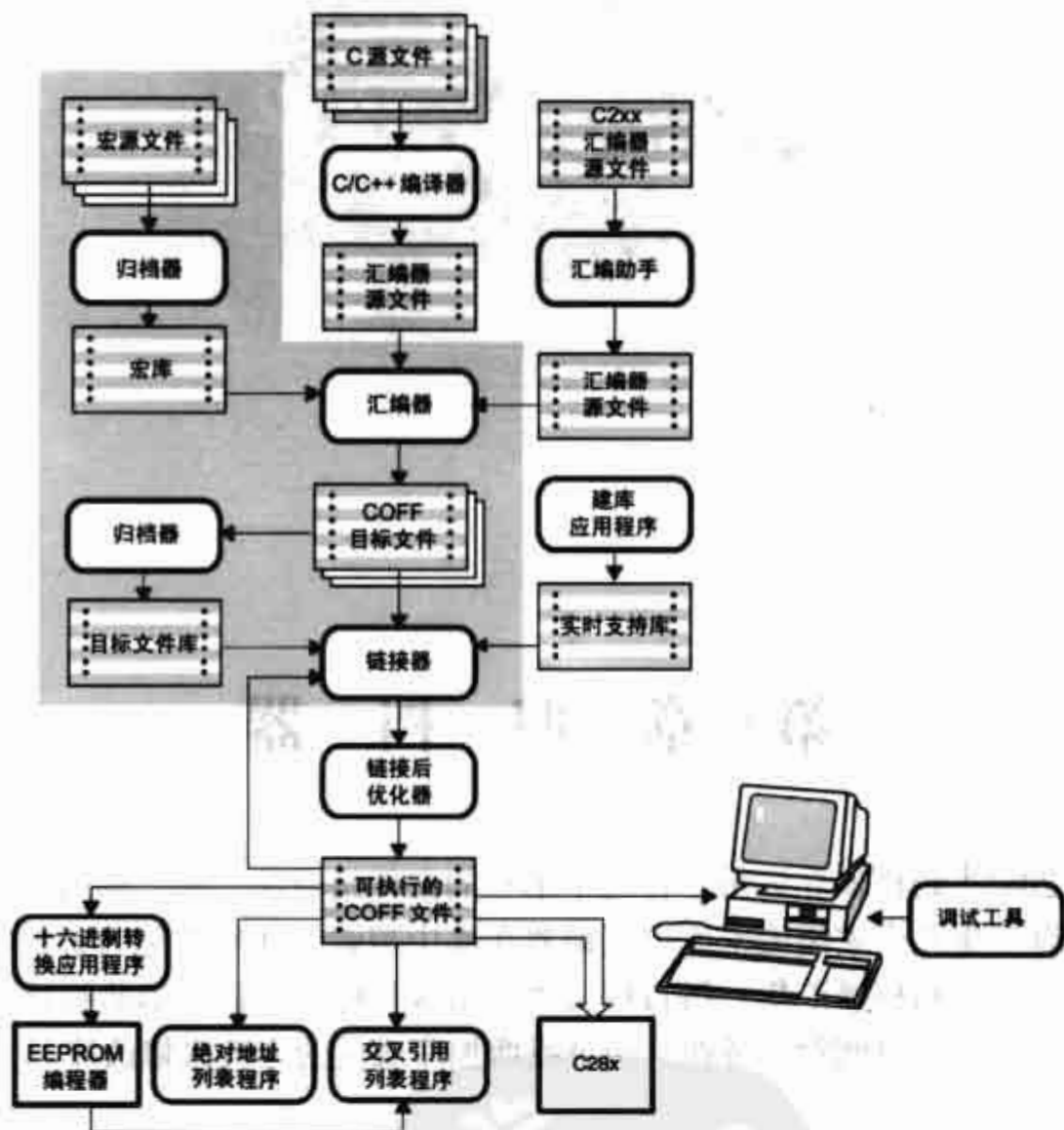


图 6.1 TMS320C28x 软件开发流程中的归档器

6.3 调用归档器

调用归档器时，输入：

```
ar2000 [-]command [options] libname [filename1 ... filenamen]
```

ar2000 是调用归档器的命令。

- [*-*]command** 告诉汇编器如何操作已存在的库成员和指定的文件 *filenames*。命令前可以有一个可选的连字符 (*-*)。调用归档器时，必须使用下述命令之一，但每次只能用一个命令。归档器命令如下：
- @** 用指定文件的内容替换命令行中的条目。用户可以使用这个命令来避免操作系统对命令行长度的限制。
 - a** 将指定文件加到库中。这一命令作为一个添加文件，不替换已经存在的同名成员；只是简单地在档案文件的末尾添加新成员。
 - d** 从库中删除指定成员。
 - r** 替换库中的指定成员。如果用户不指定文件名，归档器用当前目录中有相同名字的文件来替换库成员。如果库中找不到指定成员，归档器就将文件添加到库中，而不作替换。
 - t** 打印库的内容表。如果用户指定了成员名，仅列出指定成员列表。如果用户没有指定成员名，归档器将列出指定库中的所有成员列表。
 - x** 提取指定成员。如果用户没有指定成员的名字，归档器提取所有库成员。归档器提取一个成员时，它简单地将成员复制到当前目录，并不从库中移走这一成员。
- options** 除了使用 *command* 外，用户可以指定选项。使用选项时，将选项与命令结合起来；如，使用 *a* 命令和 *s* 选项，输入 *-as* 或 *as*。连字符 (*-*) 对归档器而言，是可选项。下面是归档器选项：
- q** (*quiet*) 禁止标题和状态信息。
 - s** 打印库中定义的全局符号的列表（这一选项只有与 *a*、*r* 和 *d* 命令连用时有效）。
 - u** 仅替换库成员，假如有一个最新修改数据要替换，用户必须将 *-u* 选项与 *r* 命令连用，来指定替换的成员。
 - v** 从一个旧库和成员中创建新库时，提供对文件的说明。
- libname** 建立或修改的库名字。如果用户不为 *libname* 指定扩展名，归档器使用默认扩展名 *.lib*。
- filenames** 操作的单独文件名。这些文件可以是已经存在的库成员或是要添加到库中的新文件。如果是可应用文件，用户输入文件名时，必须输入包括扩展名的全名。文件名最长可以有 15 个字符，如果文件名超过 15 个字符，归档器将文件名截断。

注意：一个库可能（但不希望）包含几个同名成员。如果用户要删除、替换成员或是提取一个成员作为另一库成员，那归档器就删除、替换或提取库中的第一个具有这个名字的库成员。

6.4 归档器实例

下面是典型的归档器操作实例：

- 如果用户想创建一个叫做 `function.lib` 的库，库中包含 `sine.obj`、`cos.obj` 和 `flt.obj` 文件，请输入：

```
ar2000 -a function sine.obj cos.obj flt.obj
```

归档器作如下回应：

```
TMS320C28x Archiver Version x.xx
Copyright (c) xxxx-xxxx Texas Instruments Incorporated
==> new archive 'function.lib'
==> building archive 'function.lib'
```

- 如果用户想打印 `function.lib` 库的目录表，使用 `-t` 命令，请输入：

```
ar2000 -t function
```

归档器作如下回应：

```
TMS320C28x Archiver Version x.xx
Copyright (c) xxxx-xxxx Texas Instruments Incorporated
```

FILE NAME	SIZE	DATE
sine.obj	300	Wed Apr 4 10:00:24 2001
cos.obj	300	Wed Apr 4 10:00:30 2001
flt.obj	300	Wed Apr 4 09:59:56 2001

- 如果用户想向库中添加新成员，请输入：

```
ar2000 -as function atan.obj
```

归档器作如下回应：

```
TMS320C28x Archiver Version x.xx
Copyright (c) xxxx-xxxx Texas Instruments Incorporated
==> symbol defined: '_sin'
==> symbol defined: '$sin'
==> symbol defined: '_cos'
==> symbol defined: '$cos'
==> symbol defined: '_tan'
==> symbol defined: '$tan'
==> symbol defined: '_atan'
==> symbol defined: '$atan'
==> building archive 'function.lib'
```

因为本例没有为库名指定扩展名，归档器将文件加到 `function.lib` 中，如果 `function.lib` 不存在，归档器创建 `function.lib-s`。选项告诉归档器对库中定义的全局符号列表。

- 如果用户想修改一个库成员，可以提取、编辑、替换成员。假设有一个叫做 `macros.lib` 的库，包含成员 `push.asm`、`pop.asm`、`swap.asm`。

```
ar2000 -x macros push.obj
```

归档器将 `push.asm` 复制到当前目录；但不会从库中移走 `push.asm`。现在，用户可

以编辑复制的文件。要用编辑好的副本来替换库中的push.asm, 请输入:

```
ar2000 -r macros push.obj
```

- 如果用户想使用命令文件, 在@命令后指定命令文件名。如:

```
ar2000 @modules.cmd
```

归档器作如下回应:

```
TMS320C28x Archiver Version x.xx  
Copyright (c) xxxx-xxxx Texas Instruments Incorporated  
==> building archive 'modules.lib'
```

这是 modules.cmd 命令文件:

```
; Command file to replace members of the  
; modules library with updated files  
; Use r command and u option:  
ru  
; Specify library name:  
modules.lib  
; List filenames to be replaced if updated:  
align.obj  
bss.obj  
data.obj  
text.obj  
sect.obj  
clink.obj  
copy.obj  
double.obj  
drnolist.obj  
emsg.obj  
end.obj
```

r命令指定用命令文件给出的文件名替换modules.lib库中的同名文件。-u选项指明, 只有当前文件的最近修改日期比库中的文件晚时, 才作替换。





第7章 链 接 器

TMS320C28x™ 链接器链接 COFF 目标文件生成可执行的模块。COFF 段的概念是链接器操作的基础。在第 2 章通用目标文件格式的介绍中，详细讨论了 COFF 目标文件的格式。

7.1 链接器概述

TMS320C28x 链接器允许通过在系统的存储器映像图中有效地分配段来配置系统存储器。链接器链接目标文件时执行以下任务：

- 把段定位到目标系统配置的存储器中。
- 重定位符号和段，为它们分配最终的地址。
- 在输出文件之间解决未定义的外部引用。

链接器的命令语言可以配置存储器，定义输出段，对地址赋值。这种命令语言支持表达式参数和表达式求值。可通过定义和创造存储器模式来配置系统存储器。两个功能强大的伪指令 MEMORY 和 SECTIONS 可完成以下操作：

- 将段分配到指定的存储器区域。
- 组合目标文件段。
- 在链接时定义和重新定义全局符号。

7.2 在软件开发流程中链接器的作用

图 7.1 说明了链接器在软件开发流程中的作用。链接器接收几种类型的文件，包括目标文件、命令文件、库文件和部分已链接的文件。链接器生成 COFF 目标模块，这些模块可下载到几种开发工具中或 TMS320 器件中执行。

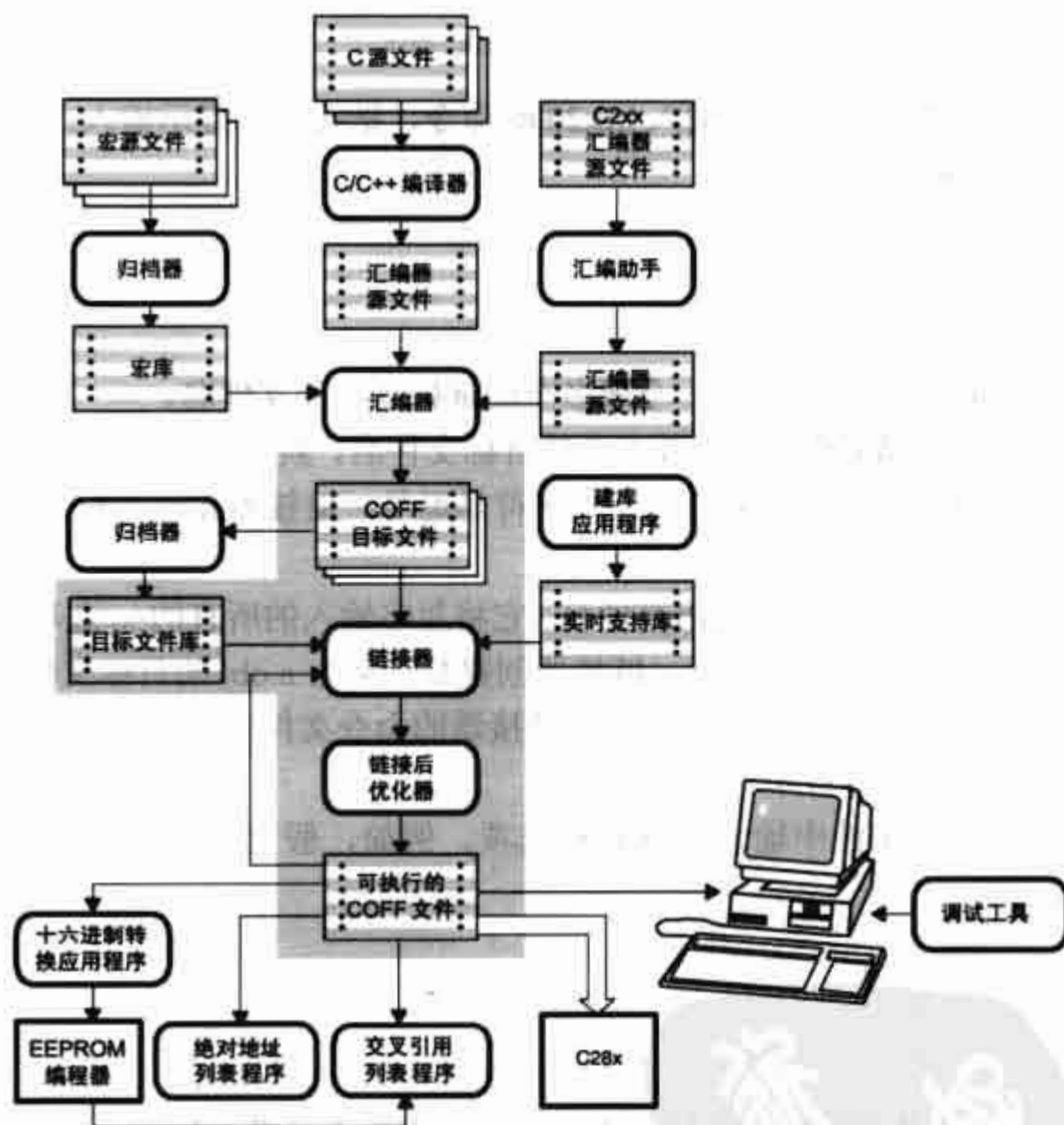


图 7.1 在 TMS320C28x 软件开发流程中的链接器

7.3 调用链接器

调用链接器的一般格式：

```
lnk2000 [options] filename1, ..., [filenamen]
```


lnk2000	调用链接器的命令。
<i>options</i>	可出现在命令行的任何位置，或出现在命令文件中。（选项见第 7.4 节中的链接器选项）
<i>filename</i>	可以是目标文件，链接器命令文件或归档库。所有输入文件的默认扩展名是.obj；其他的扩展名必须明确地指出。链接器能够确定文件是目标文件还是包含链接器命令的 ASCII 文件。输出文件的默认名是 a.out，除非用户用 -o 选项命名了输出文件。

调用链接器有 3 种方法：

- 在命令行中指定选项和文件名。这是链接两个文件 file1.obj 和 file2.obj 的例，它还要创建一个以 link.out 命名的输出文件。

```
lnk2000 file1.obj file2.obj -o link.out
```

- 如果没有文件名和选项，输入 **lnk2000** 命令，链接器将提示输入：

```
Command files :
Object files [.obj] :
Output file [ ] :
Options :
```

- 为 *command files* 输入一个或多个的链接器命令文件名。
 - 为 *object files* 输入一个或多个的目标文件名。默认扩展名是.obj。用空格或逗号分隔文件名。如果最后一个字符是逗号，链接器在附加行将提示输入目标文件名。
 - *output file* 是链接器输出文件名。它将忽略输入的所有的 -o 选项。如果没有 -o 选项而且没有回答提示，链接器创建默认名为 a.obj 的目标文件名。
 - *options* 提示附加选项也可以在链接器的命令文件中输入。用短线输入它们，这就像在命令行中一样。
- 在链接器命令文件中输入文件名和选项。例如，假定文件 linker.cmd 包括以下的行：

```
-o link.out
file1.obj
file2.obj
```

现在用户可以从命令行调用链接器，指定作为输入文件的命令文件名。

```
lnk2000 linker.cmd
```

当使用命令文件时，可以在命令行指定其他的选项和文件。例如可以输入：

```
lnk2000 -m link.map linker.cmd file3.obj
```

链接器一遇到命令行中的文件名就读取和处理这个命令文件，因此链接文件的顺序是：file1.obj, file2.obj, file3.obj。这个例创建了一个 link.out 的输出文件和一个 link.map 的映像文件。

7.4 链接器选项

链接器的选项控制链接操作。它们可以放置在命令行或命令文件中。链接器选项必须以短划线(-)打头。除了-l和-i选项外，其他指定选项的顺序并不重要。因为要从-i选项读入-l选项，所以-i选项必须出现在-l选项前面。如果选项有参数，用空格将选项和参数分开。表7.1为链接器选项一览表。

可以把没有参数的选项串在一起（例如，lnk2000 -ar）或分开输入（例如，lnk2000 -a -r）。有参数的选项必须与其他选项分开（例如，lnk2000-i dsptools -ar）。

表 7.1 链接器选项一览表

选 项	描 述
-a	生成绝对可执行模块。如果既没指定-a也没指定-r选项，链接器默认设置为-a的工作方式
-ar	生成可重定位的可执行目标代码
-b	禁止符号调试信息的合并
-c	在运行时自动初始化变量
-cr	在装载时自动初始化变量
-e <i>global_symbol</i>	定义全局变量，规定输出模块的初始入口
-f <i>fill_value</i>	设置输出段中空位的默认填充值， <i>fill_value</i> 是一个16位的常量
-g <i>symbol</i>	生成全局符号（忽略-h）
-h	使所有的全局符号为静态
-heap <i>size</i>	设置堆（动态存储器进行地址分配时使用堆）的大小为 <i>size</i> 个字，并定义一个全局符号指定堆的大小，默认大小是1K字
-i <i>pathname</i> [†]	改变库搜索算法，在搜索默认目录之前搜索用 <i>pathname</i> 指定的路径。这个选项必须出现在-l选项之前
-j	禁止条件链接
-i <i>filename</i> [†]	命名作为链接器输入的归档库或链接文件名
-m <i>filename</i>	生成输入输出段的映像或列表，包括空位，并把列表放在 <i>filename</i> 文件中
-o <i>filename</i> [†]	命名可执行的输出文件，默认的文件名是 a.out
-q	禁止标志和所有的进程信息
-r	生成不可执行的可重定位的输出文件
-s	从输出文件中删除符号表信息和行号入口
-stack <i>size</i>	设置堆栈的大小为 <i>size</i> 个字，默认大小是1K字
-u <i>symbol</i>	在输出文件符号表中放置非保留的外部符号
-w	当创建未定义的输出段时显示信息
-x	强制重读入后面要引用的库

[†] *pathname* 和 *filename* 必须遵守操作系统的约定。

[‡] 详细的信息见 *TMS320C28x Optimizing C/C++ Compiler User's Guide*。

7.4.1 重定位（-a 和 -r 选项）

链接器执行重定位，就是当符号的地址改变时调整所有对符号的引用过程。链接器支持两种选项（-a 和 -r），通过这两种选项可以生成绝对的或可重定位的输出模块。

1. 生成一个绝对的输出文件（-a 选项）

当用户使用了 -a 选项而没有用 -r 选项时，链接器生成绝对的可执行文件。绝对的可执行文件不包括可重定位的信息。可执行文件包括以下内容：

- 链接器定义的专用符号
- 说明程序入口信息的可选标题
- 没有未定义的引用

下例链接了 file1.obj 和 file2.obj 文件并生成输出文件 a.out：

```
lnk2000 -a file1.obj file2.obj
```

注意：如果用户没有使用 -a 或 -r 选项，链接器指定 -a。

2. 生成可重定位的输出文件（-r 选项）

如果用户使用了 -r 选项而没有使用 -a 选项，链接器在输出文件中保留可重定位的入口。如果输出文件被重定位（在装载时）或重新链接，使用 -r 选项保留可重定位入口。

当用户使用了 -r 选项而没有使用 -a 选项时，链接器将生成不可执行的文件。不可执行的文件不包括特定的链接器符号或可选的标题。这个文件可能包含非保留的引用，但这些引用不会阻止生成输出文件。

例：链接 file1.obj 和 file2.obj 并生成输出文件 a.out。

```
lnk2000 -r file1.obj file2.obj
```

输出文件 a.out 可与其他的目标文件重新链接或在装载时重定位（链接一个可以被其他文件链接的文件称为部分链接）（详见 7.16 节）。

3. 生成一个可执行的、可重定位的输出文件（-ar 选项组合使用）

如果用 -a 和 -r 选项共同调用链接器，链接器将生成可执行的且可重定位的目标模块。输出文件包含了特定的链接器符号、可选标题和所有保留的符号引用，而且还要保留重定位信息。

例：链接 file1.obj 和 file2.obj 文件并生成可执行、可重定位的输出文件 xr.out。

```
lnk2000 -ar file1.obj file2.obj -o xr.out
```

当链接器遇到了包含未重定位的符号或符号表的信息时，就产生警告信息（但仍可执行）。只有所有的输入文件没有包含重定位信息时，才能成功的链接绝对文件（也就是每一个文件没有未定义的引用，当链接器生成它们时，为这些文件的未定义引用分配同一个虚地址）。

7.4.2 禁止合并符号调试信息

默认情况下，链接器清除符号调试信息的副本。这些副本信息通常出现在对 C 语言程序进行汇编与调试的时候。例如：

```
-[ header.h ]-
typedef struct
{
<define some structure members>
} XYZ;
-[ f1.c ]-
#include "header.h"
...
-[ f2.c ]-
#include "header.h"
...
```

当编译这些文件供调试时，f1.obj 和 f2.obj 都有说明 XYZ 类型的符号调试入口。对于最终的输出文件，只需一组这些入口，链接器将自动合并入口副本。

使用 -b 选项可以禁止合并入口副本。当使用 -b 选项时，链接器运行比较快，且使用较少的机器存储器。-b 选项应该放在目标文件之前，否则将被链接器忽略。

7.4.3 C 语言程序选项（-c 和 -cr 选项）

-c 和 -cr 选项使链接器使用 C/C++ 编译器所要求的链接规则。

- -c 选项告诉链接器在运行时自动初始化变量。
- -cr 告诉链接器在装载时自动初始化变量。

详见 7.17 节链接 C 代码，7.17.4 节在运行时自动初始化变量，7.17.5 节在装载时自动初始化变量。

7.4.4 定义入口（-e 选项）

程序开始执行的存储器地址称为入口。当装载器将程序装入目标存储器时，程序计数器（PC）必须初始化为入口地址值，这样 PC 指向程序的开始处。

链接器可以为入口分配 4 个值中的一个。下面以链接器使用它们的先后顺序，列出这些值。如果用户用了前面 3 个中的一个，则它必须是符号表中的一个外部符号。

用 -e 选项指定值，格式：

```
-e global_symbol
```

用来定义入口的 *global_symbol* 必须作为其中一个输入文件的外部符号出现。

(1) 符号 _c_int00 的值（如果存在的话）。如果正在链接代码的是由 C/C++ 编译器生成的，则入口必须是 _c_int00。

(2) 符号 `_main` 的值（如果存在的话）。

(3) 0（默认值）。

例：链接 `file1.obj` 和 `file2.obj` 文件。符号 `begin` 是入口，`begin` 必须是在 `file1` 或 `file2` 中定义的外部符号。

```
lnk2000 -e begin file1.obj file2.obj
```

7.4.5 设置默认的填充值（`-f fill_value` 选项）

`-f` 选项填充在输出文件中形成的空位或填充在未初始化段与初始化段混合使用时初始化未初始化段。这样允许用户不用重新汇编源文件，就可以在进行链接时初始化存储器的区域。参数 `fill_value` 是一个 16 位的常量。如果没有使用 `-f`，链接器把 0 作为默认的填充值。

例：以十六进制数 `ABCDh` 填充。

```
lnk2000 -f 0ABCDh file1.obj file2.obj
```

详见 7.15 节生成和填充空位。

7.4.6 全局化符号（`-g symbol` 选项）

`-h` 选项使所有的全局符号为静态。如果用户用了 `-h` 选项，仍想保留一个符号为全局符号，可以使用 `-g` 选项声明这个符号是全局的。`-g` 选项消除 `-h` 选项对用户指定的符号的作用。`-g` 选项的格式是：

```
-g symbol
```

7.4.7 静态化所有的全局符号（`-h` 选项）

`-h` 选项使所有的全局符号为静态。静态符号在外部的链接模块中是不可见的。把全局符号静态化实际是把全局符号隐藏起来。这样在不同的文件中可使用相同名字的外部符号，而互不影响。

`-h` 选项会使所有的 `.global` 汇编伪指令失效。所有的符号成了定义它的模块中的局部符号，这样就不能使用外部引用了。例如，假定 `file1.obj` 和 `file2.obj` 都定义了全局符号 `EXT`，通过使用 `-h` 选项可以无冲突地链接这些文件，在 `file1.obj` 中定义的 `EXT` 与在 `file2.obj` 中定义的 `EXT` 是各自独立互不相关的。

```
lnk2000 -h file1.obj file2.obj
```

7.4.8 定义堆的大小（`-heap size` 选项）

C/C++ 编译器使用一个称为 `.sysmem` 的未初始化段供 `malloc()` 动态分配存储空间。链接时，可以用 `-heap` 选项设置可动态分配的存储空间的大小。紧接在选项后面的 `size` 为常

量。下例生成一个有 2K 字的堆：

```
lnk2000 -heap 0x0800 /* defines a 2k heap (.sysmem section) */
```

只有在输入文件中存在一个 .sysmem 的段时，链接器才生成 .sysmem 的段。

链接器也生成一个全局符号 `__SYSMEM_SIZE`，并为它分配一个与堆大小一样的值。默认值的大小是 1K 字。

详见 7.17 节链接 C 代码。

7.4.9 改变库的搜索路径（-l 选项、-i 选项和 C_DIR 环境变量）

通常，如果想把一个库指定为链接器的输入时，用户可简单地像输入任何其他输入文件名一样输入库名，链接器就将会在当前目录中寻找文件名。例如，假设当前的目录包括 `object.lib` 库。假设这个库中定义了 `file1.obj` 文件中引用的符号。下面说明如何链接文件：

```
lnk2000 file1.obj object.lib
```

如果用户想使用不在当前目录中的库或命令文件，则使用 -l（小写的 L）链接器选项。这个选项的格式是：

```
-l filename
```

filename 是一个库名或链接器命令文件名，在 -l 和文件名之间的空格是可选的。

可以使用 -i 链接器选项或环境变量设定链接器搜索路径。链接器以下面的顺序搜索指定的目录：

- (1) 搜索 -i 选项指定的目录。命令行或命令文件中的 -i 选项必须出现在 -l 选项之前。
- (2) 搜索用 C_DIR 指定的目录。
- (3) 如果没有设置 C_DIR，将搜索用汇编器 A_DIR 环境变量指定的目录。
- (4) 搜索当前的目录。

1. 指定备选库的目录（-i pathname 选项）

-i 选项用来指定一个包含目标库的目录。此选项的格式是：

```
-i pathname
```

pathname 指定包含目标库的目录，-i 和 *pathname* 之间的空格是可选的。链接器搜索 -l 指定的目标库之前，首先搜索 -i 选项指定的目录。每一个 -i 选项只能指定一个目录，但在一次调用中可使用几个 -i 选项。当用户使用 -i 选项指定备选目录时，-i 选项必须出现在命令行或命令文件中的所有 -l 选项之前。

例：假定有两个库 `r.lib` 和 `lib2.lib`。设库的路径是：

```
UNIX: /ld/r.lib and /ld2/lib2.lib
```

```
Windows: c:\ld\r.lib and c:\ld2\lib2.lib
```

下例说明如何使用 -i 选项和在一次链接中使用两个库文件：

操 作 系 统	输 入
UNIX	lnk2000 f1.obj f2.obj -i/ld -i/ld2 -lr.lib -llib2.lib
Windows	lnk2000 f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib

2. 指定备选库的目录 (C_DIR 环境变量)

环境变量可以是定义和分配一个字符串的系统符号。链接器用 C_DIR 命名的环境变量指定包含目标库的目录。为环境变量赋值的格式是：

操 作 系 统	输 入
UNIX setenv	C_DIR "pathname1;pathname2;..."
Windows set	C_DIR= pathname1;pathname2;...

pathname 是包含目标库的目录。在命令行或命令文件中用 -l (小写的 L) 链接器选项告诉链接器在这些目录中搜索库。

例：假定有两个以 r.lib 和 lib2.lib 命名的库。假定库的路径是：

```
UNIX: /ld/r.lib and /ld2/lib2.lib
Windows: c:\ld\r.lib and c:\ld2\lib2.lib
```

下例说明，如何设置环境变量和在一次链接中使用两个库：

操 作 系 统	输 入
UNIX	setenv C_DIR "/ld /ld2" lnk2000 f1.obj f2.obj -l r.lib -l lib2.lib
Windows	set C_DIR=ld\ld2 lnk2000 f1.obj f2.obj -l r.lib -l lib2.lib

环境变量保持设置直到重新启动系统或通过以下输入重新设置环境变量：

操作系统	输 入
UNIX	unsetenv C_DIR
Windows	set C_DIR=

汇编器用由 A_DIR 命名的环境变量指定包含 .copy/.include 文件或宏库的备选目录。如果没有设置 C_DIR，链接器将在由 A_DIR 指定的目录中搜索目标库。详见 7.6 节目标库。

7.4.10 取消条件链接

-j 选项可取消用汇编器 .clink 伪指令建立的条件链接。默认情况下，所有的段是非条件链接。关于用 .clink 伪指令建立条件链接，详见第 4.13 节。

7.4.11 忽略分配 (-k 选项)

-k 选项强制链接器忽略用 `.align` 伪指令指定的所有分配。关于 `.align` 伪指令的详细情况见第 4.23 节。

7.4.12 创建映像文件 (-m filename 选项)

-m 选项创建一个以 *filename* 命名的链接器映像列表文件。-m 选项的格式为：

```
-m filename
```

此链接器映像文件描述：

- 存储器配置
- 输入和输出段的分配
- 被重定位的外部符号的地址

映像文件包括输出文件的名称和程序的入口地址，它最多可以包括 3 个表：

(1) 显示新的存储器配置的表，所指定的存储器（存储器配置）都是非默认设置。所产生信息是在链接器的命令文件中基于 `MEMORY` 伪指令的信息。这个表包含以下几列：

- **Name** 这是用 `MEMORY` 伪指令指定的存储器区域的名字
- **Origin** 指定存储器区域的开始地址
- **Length** 指定存储器区域的长度
- **Attributes** 指定与区域有关的 1~4 个属性：
 - R** 存储器可读
 - W** 存储器可写
 - X** 存储器包含可执行代码
 - I** 存储器可以初始化
- **Fill** 为存储器指定一个填充字符

关于 `MEMORY` 伪指令的详情见第 7.7 节。

(2) 每个输出段和构成输出段的输入段的链接地址表（段的地址分配图）。所产生信息是在链接器的命令文件中基于 `SECTIONS` 伪指令的信息。此表有以下几列：

- **Output section** 用 `SECTIONS` 伪指令指定的输出文件的名称。
- **Origin** 第一个 origin（起始）值列出的是每一个输出段的起始地址。首行缩进的 origin（初始）值是部分输出段的起始地址。
- **Length** 第一个 length（长度）值列出的是每一个输出段的长度。首行缩进的 length（长度）值列出的是部分输出段的长度。
- **Attributes/input sections** 列出输入文件或一个输出文件有关的值。

(3) 用来显示外部符号和它们地址的两个表，一个用来显示名字，一个用来显示地址。

例：链接 file1.obj 和 file2.obj 文件并生成映像文件 map.out：

```
lnk2000 file1.obj file2.obj -m map.out
```

7.4.13 命名输出文件（-o filename 选项）

当没有遇到错误时，链接器生成一个输出文件。如果用户没有为输出文件指定文件名，链接器的默认文件名是 a.out。如果想为输出文件指定一个不同的名字，使用 -o 选项。-o 选项的格式是：

```
-o filename
```

filename 是输出文件的名称。

例：链接文件 file1.obj 和 file2.obj 并生成输出文件 run.out。

```
lnk2000 -o run.out file1.obj file2.obj
```

7.4.14 隐藏运行信息（-q 选项）

-q 选项隐藏链接器的运行信息，但它必须出现在第一个选项的位置上。否则还是会显示信息。这个选项在批处理中不显示链接器的链接过程信息时有用。

7.4.15 删除符号信息（-s 选项）

-s 选项通过删除符号表信息和行号入口生成较小的输出文件。在生成尽可能小的输出文件时这个选项是有用的。

例：链接 file1.obj 和 file2.obj 并生成删除了行号和符号表信息的以 nosym.out 命名的输出文件。

```
lnk2000 -o nosym.out -s file1.obj file2.obj
```

因为 -s 选项删除了输出文件中的符号信息，-s 选项的使用限制了以后符号调试使用，和重新链接文件。

7.4.16 定义堆栈的大小（-stack size 选项）

C/C++ 编译器用未初始化的段 .stack 为运行时的堆栈分配空间。可以在链接时使用 .stack 选项设置这个段的大小。下例创建了一个 4K 字大小的堆栈：

```
lnk2000 -stack 0x1000 /* defines a 4K stack (.stack section) */
```

如果在输入段中定义了另一个不同的堆栈大小，输入段中的堆栈大小被忽略。在输入段中定义的所有符号保持有效，只是堆栈的大小改变了。

当链接器定义 .stack 段的时候，它也定义一个全局符号 __STACK_SIZE，并把段的大小

赋给这个符号。默认堆栈是 1K 字。

7.4.17 引入未定义的符号（-u symbol 选项）

-u 选项在链接器的符号表中引入未定义的符号。它强迫链接器搜索库并包含定义这个符号的成员。链接器必须在链接定义这个符号的成员之前遇到 -u 选项。-u 选项的格式是：

`-u symbol`

例如，假定一个名字是 `rts2800.lib` 的库包含了定义符号 `symtab` 的成员，且没有被链接的目标文件引用。然而，用户打算重新链接输出文件，并想把包含定义 `symtab` 的成员包含在这个链接中。

如下例使用 -u 选项，使链接器搜索 `rts2800.lib` 以寻找定义 `symtab` 的成员并链接该成员。

```
lnk2000 -u symtab file1.obj file2.obj rts2800.lib
```

如果用户不使用 -u 选项，这个成员将不会被包含在里面，因为在 `file1.obj` 和 `file2.obj` 中没有明确的引用。

7.4.18 当创建未定义的段时显示信息（-w 选项）

-w 选项显示创建存储器段的附加信息。在下列的情况下显示附加信息：

- 在链接器命令文件中，可以用 `SECTIONS` 伪指令说明输入段如何组合到输出段。然而，当链接器遇到输入段，这个输入段没用 `SECTIONS` 伪指令定义一个相应的输出段，链接器把具有相同名字的输入段组合到同样名字的输出段中。默认的情况下，链接器显示信息，告诉用户这种情况什么时候发生的。如果出现这种情况而且使用了 -w 选项，链接器显示什么时候它创建了一个新输出段的信息。
- 当用户没有使用 -heap 和 -stack 选项时链接 C 代码，链接器会为用户生成相应的 `.sysmem` 段和 `.stack` 段。每一段的默认大小是 1K 字。对于一个段或所有的段可能没有足够的存储空间。这种情况下，链接器显示错误信息，指出不能够分配该段。

如果用户使用了 -w 选项，链接器另外显示更多信息，包括用哪个伪指令来分配 `.sysmem` 和 `.stack` 段等。

关于 `SECTIONS` 详见第 7.8 节，关于链接器默认操作详见第 7.13 节。

7.4.19 穷举读库（-x 选项）

链接器对在命令行和命令文件中遇到（包含归档库）的输入文件通常只读一次。当归档文件被读入时，在链接中包含所有要引用的未定义符号的成员。如果以后的输入文件引用了先前读入的归档库中定义的符号（称为向后引用），那么这个引用是不确定的。

使用 -x 选项，可以迫使链接器重复地读所有的库。链接器重复读的库直到所有的引用被确定。例如，如果 `a.lib` 包含了一个在 `b.lib` 中定义的符号引用，`b.lib` 包含了一个在 `a.lib`

中定义的符号引用，可以通过对其中一个库列举两次来解决它们彼此的相关性。如下所示：

```
lnk2000 -la.lib -lb.lib -la.lib
```

或者强制链接器这样做：

```
lnk2000 -x -la.lib -lb.lib
```

使用-x 选项链接可能比每个输入文件读一次要慢，所以只有在必要的时候才用。

7.5 链接器命令文件

链接器命令文件允许用户把链接信息放在一个文件中，这在经常用相同的信息调用链接器时是很有用的。链接器命令文件的另一个用处是，允许用户使用 MEMORY 和 SECTIONS 伪指令定制用户的应用。必须在一个命令文件中使用这些伪指令，而不能在命令行中使用它们。

链接器命令文件是 ASCII 文件，它包含下列各项中的一项或多项：

- 输入文件名：用来指定目标文件、库或其他命令文件（如果一个命令文件调用其他的命令文件作为输入，这条语句必须在调用文件的最后。链接器不会从被调用的命令文件中返回）。
- 链接器选项：在命令文件中的使用方式与在命令行中的使用方式相同。
- MEMORY 和 SECTIONS 链接器伪指令：MEMORY 伪指令定义目标存储器的配置。SECTIONS 伪指令控制如何建立和分配段（见 7.8 节段伪指令）。
- 赋值语句：定义和赋值给全局变量。

用命令文件调用链接器，输入 lnk2000 命令，紧跟着输入命令文件名：

```
lnk2000 command_filename
```

链接器处理输入文件是以遇到输入文件的顺序来处理的。如果链接器确认一个文件是目标文件，就链接这个文件。如果链接器确认一个文件是目标库，就用这个库去确定引用。其他情况下，链接器会确认文件为命令文件，并开始读和处理里面的命令。不管使用的是什么系统，命令文件名都是大小写敏感的。

例 7.1 这是一个名为 link.cmd 的命令文件

```
a.obj          /* 第一个输入文件名 */
b.obj          /* 第二个输入文件名 */
-o prog.out    /* 指定输出文件选项 */
-m prog.map    /* 指定映射文件选项 */
```

例 7.1 中的文件只包含文件名和选项（可以在 /*和*/ 之间放置注释）。用这个命令文件调用链接器：

```
lnk2000 link.cmd
```

当使用命令文件时可以在命令行放置其他的参数：

```
lnk2000 -r link.cmd c.obj d.obj
```

链接器遇到命令文件就立即处理它，所以 a.obj 和 b.obj 在 c.obj 和 d.obj 之前被链接到输出文件中。

可以指定多个命令文件。例如，如果一个名为 names.lst 的文件包含文件名并包含另一个含有链接器伪指令名为 dir.cmd 的命令文件，用户可以输入：

```
lnk2000 names.lst dir.cmd
```

一个命令文件可以调用另一个命令文件，这种类型的嵌套至多允许 16 层。如果一个命令文件调用另一个命令文件作为输入，其调用语句必须在调用文件的最后。

除了作为分隔符之外，空格和空格行在命令文件中无关紧要。例 7.2 是包含有链接器伪指令的典型命令文件。

例 7.2 使用链接器伪指令的命令文件

```
a.obj b.obj c.obj          /* 输入文件名 */
-o prog.out -m prog.map    /* 选项 */
MEMORY                     /* MEMORY 伪指令 */
{
    RAM: origin = 0100h     length = 0100h
    ROM: origin = 01000h    length = 0100h
}
SECTIONS                    /* SECTIONS 伪指令 */
{
    .text: > ROM
    .data: > ROM
    .bss: > RAM
}
```

7.5.1 链接器命令文件中的保留名

以下名字在链接器命令中作为关键字，在命令文件中不能用它们作为符号和段名。

align	group	origin
ALIGN	GROUP	ORIGIN
attr	l (lowercase L)	page
ATTH	len	PAGE
block	length	range
BLOCK	LENGTH	run
COPY	load	RUN
DSECT	LOAD	SECTIONS
f	MEMORY	spare
fill	NOLOAD	type
FILL	o	TYPE
	org	UNION

7.5.2 链接器命令文件中的常量

除了不支持二进制格式外，链接器使用的常量格式与汇编器使用的格式相同。详见第 3.6 节常量。

7.6 目 标 库

目标库是一种归档文件，它包含作为成员的完整目标文件。通常，一组相关联的模块被组合在一起构成一个库。当用户指定一个目标库作为链接器的输入时，链接器就包括进库的所有成员，这个库定义了已有的未确定的符号引用。可以用归档器建立和维护库。详细情况见第 6 章归档器的说明。

使用目标库能减少链接时间和缩小可执行模块的大小。如果链接一个目标文件，不论这个文件中的函数是否使用，这个目标文件都会被链接。然而若同样的函数被放置在归档库中，只有这个函数被引用时，这个库才被包括进来。

指定库的顺序是重要的，因为链接器仅包括那些在本次搜索的库中未定义符号的成员。因此指定相同的库是必须的，每次搜索都会将它包含在内。换句话说，`-x` 选项能用来重新读取库，直到所有的引用均被确定。详见 7.4.19 节穷举读库（`-x` 选项）。每个库均含有一个表，这个表列出了这个库中定义的所有外部符号。链接器通过这个表搜索库，直到不能再用这个库确定更多的引用为止。

下面的例是链接几个文件和库，假定：

- 输入文件 `f1.obj` 和 `f2.obj`，都引用了名为 `clrscr` 的外部函数。
- 输入文件 `f1.obj` 引用符号 `origin`。
- 输入文件 `f2.obj` 引用符号 `fillclr`。
- 库 `libc.lib` 的成员 0 含有 `origin` 的定义。
- 库 `liba.lib` 的成员 3 含有 `fillclr` 的定义。
- 两个库的成员 1 都定义了 `clrscr`。

如果用户输入：

```
lnk2000 f1.obj f2.obj liba.lib libc.lib
```

则有：

- 库 `liba.lib` 的成员 1 满足 `f1.obj` 和 `f2.obj` 对 `clrscr` 的引用，因为库（`liba.lib`）被搜索到而且找到 `clrscr` 的定义。
- 库 `libc.lib` 的成员 0 满足对 `origin` 的引用。
- 库 `liba.lib` 的成员满足对 `fillclr` 的引用。

然而，如果用户输入：

```
lnk2000 f1.obj f2.obj libc.lib liba.lib
```

那么，对 `clrscr` 的引用将由库 `libc.lib` 的成员 1 来满足。如果在一个库中没有定义链接文

件的引用符号，用户可以使用 `-u` 选项强制链接器包括一个库的成员（详见 7.4.17 节）。

下面的例在链接器的全局符号表中创建一个未定义的符号 `route`。

```
lnk2000 -u route libc.lib
```

如果库 `libc.lib` 的任何一个成员定义了 `route`，链接器就包括这个成员。

要控制单个库成员的分配是不可能的，而成员的分配是根据 `SECTIONS` 伪指令默认的分配算法来分配的。详见第 7.8 节。

7.7 MEMORY 伪指令

链接器决定把输出段分配到存储器的什么位置，要完成这些必须选择一个目标存储器模式。`MEMORY` 伪指令允许用户指定目标存储器模式，所以用户可以定义系统包含的存储器的类型和它们占用的地址范围。当链接器分配输出段时保持这种模式，利用模式来决定存储器的哪些单元可被目标代码所用。TMS320C28x 应用系统不同有不同的存储器配置。

详见第 2.3 节链接器如何处理段。

7.7.1 默认的存储器模式

如果用户没有使用 `MEMORY` 伪指令，链接器将根据 TMS320C28x 的体系选择默认的存储器模式。关于存储器默认模式的详见第 7.13 节默认分配。

7.7.2 MEMORY 伪指令格式

`MEMORY` 伪指令指定在目标系统中可被程序使用的物理存储器区域。每一个存储器区域中包括以下特性：

- ☐ Name（名字）
- ☐ Starting address（起始地址）
- ☐ Length（长度）
- ☐ Optional set of attributes（属性选项）
- ☐ Optional fill specification（指定填充选项）

TMS320C28x 器件有分离的存储器空间，它们占用相同地址区域。在默认的存储器映像图中，一部分存储器作为程序区域，一部分作为数据区域。详见 7.11 节重叠页。

在链接器的命令文件中，用户可以用 `MEMORY` 伪指令的 `PAGE` 选项配置单独的地址空间。链接器把每一个页作为单独的存储器空间。TMS320C28x 最多支持 255 个地址空间，但是可用的地址空间数取决于用户所用的器件型号。

当用户使用 `MEMORY` 伪指令时，必须要指明装载代码可能用到的所有的存储器区域。所有用 `MEMORY` 伪指令指定的存储器将会被配置，没用 `MEMORY` 伪指令指定的存储器不会被配置。链接器不会把程序的任何部分放入未配置的存储器中。用户可以通过在 `MEMORY` 伪指令中不包含某些地址区域，来表示不存在的存储器空间。

MEMORY 伪指令在命令文件中使用，以 MEMORY（大写）开头，后跟一系列用大括号括起来的存储器区域。例 7.3 中的 MEMORY 伪指令定义了一个系统，这个系统在程序存储器地址 0C00h 处有 4K 字的 ROM，在数据存储器地址 60h 处有 32 字的 RAM，在数据存储器地址的 200h 处有 512 字。

例 7.3 MEMORY 伪指令

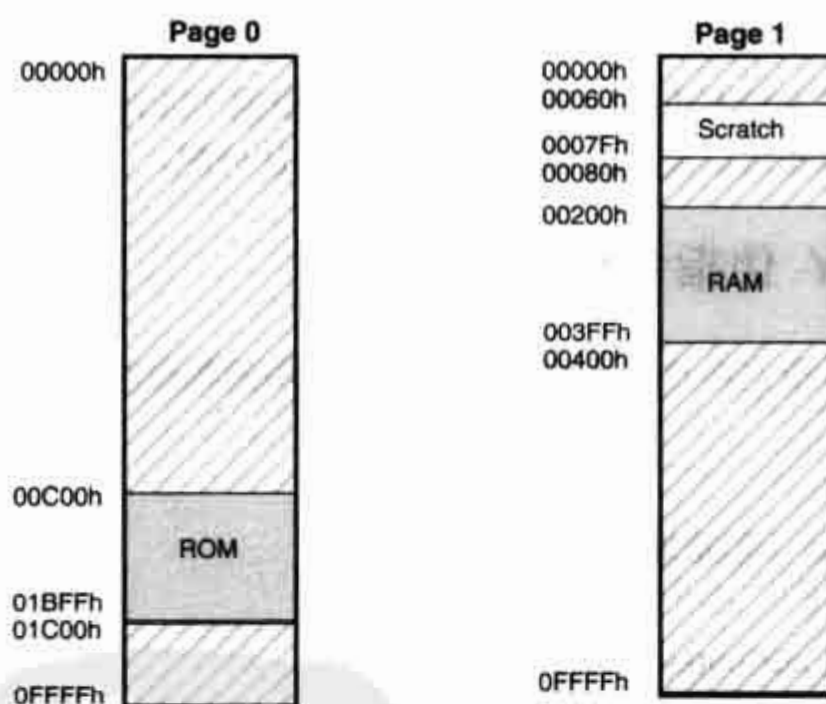
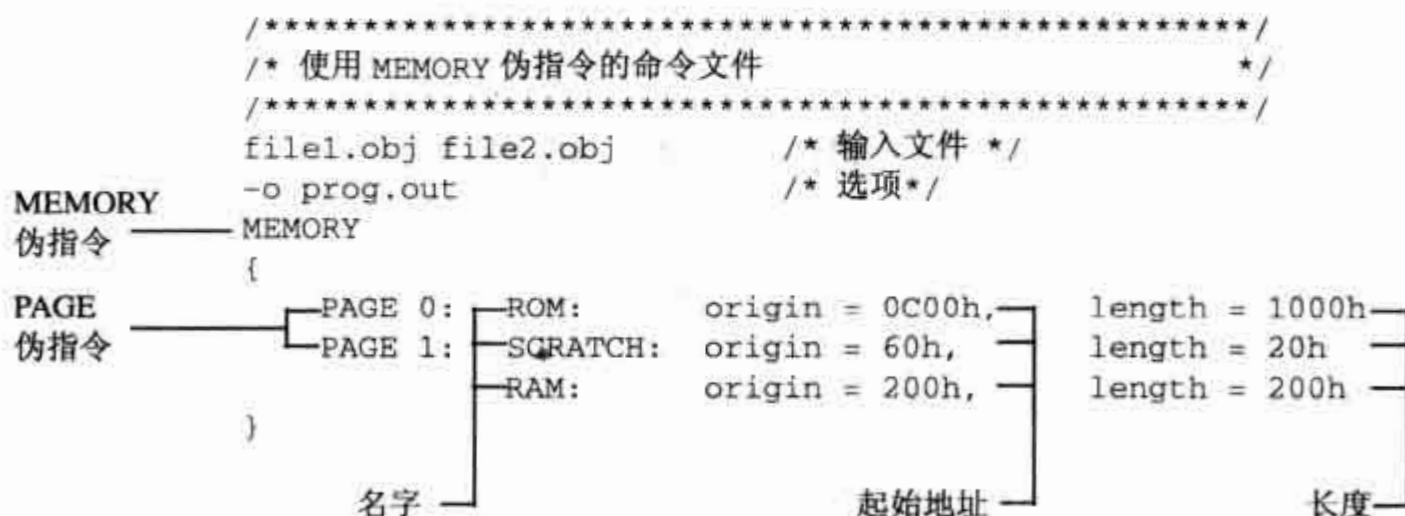


图 7.2 在例 7.3 中定义存储器的映像图

可以使用 SECTIONS 伪指令告诉链接器在何处链接这个段，详见第 7.8 节。MEMORY 伪指令的一般格式是：

```

MEMORY
{
    [PAGE 0 : ] name [(attr)] : origin = constant, length = constant[, fill = constant];
    [PAGE 1 : ] name [(attr)] : origin = constant, length = constant[, fill = constant];
}

```



```
[PAGE n:] name [(attr): origin = constant, length = constant[, fill = constant];
}
```

- PAGE** 标识存储器页。用户最多可以指定32767页。通常，PAGE 0规定程序存储器，PAGE 1规定数据存储器。如果没有指定PAGE选项，链接器使用PAGE 0。每一个PAGE代表一个完全独立的地址空间。PAGE 0配置的存储器可以与PAGE 1配置的存储器重叠，以此类推。
- name** 命名存储器。存储器的名字可以是1~8个字符，有效的字符包括A~Z、a~z、\$、. 和_。这些名字对于链接器没有特殊意义，它们只是用来识别存储器。对于链接器而言，存储器的名字是链接器内部的，在输出文件和符号表中并不保留。所有的存储器必须是惟一的，且存储器区域不能重叠。
- attr** 规定已命名区域有关的1~4个属性。属性是可选的，当被使用时，必须放在圆括号里面。属性限制输出段分配到特定的存储器区域。如果没有使用任何的属性，可以没有限制的把任意输出段放到任何单元。没有规定属性的存储器（包括默认模式的所有存储器）有全部的4个属性。有效的属性包括：
- R** 存储器可读
 - W** 存储器可写
 - X** 存储器包含可执行代码
 - I** 存储器可以初始化
- origin** 存储器区域的起始地址，可以输入origin、org或o。这个值以字节来指定，是一个22位的常量，可以是十进制、八进制或十六进制。
- length** 存储器区域的长度。可输入length、len或l。这个值以字节来指定，是一个22位的常量，可以是十进制、八进制或十六进制。
- fill** 存储器区域的填充字符。fill或f这一项是可选的。这个值是16位的整数常量，可以是十进制、八进制或十六进制。填充值用来填充来分配段的存储器区域。

注意：如果用户为大的存储器区域规定了填充值，用户的输出文件会很大，因为填充存储器区域（即使填0）会导致为存储器区域内没分配的块产生原始数据。

下例指定了一个有R和W属性的存储器区域，其填充值是0FFFh:

```
MEMORY
{
  RFILE (RW) : o = 02h, l = 0FEh, f = 0FFFFh
}
```

通常MEMORY和SECTIONS伪指令联合使用来控制输出段的分配。使用MEMORY指定目标存储器的模式后，可再使用SECTIONS伪指令把输出段分配到指定的存储器区域定了属性的存储器中。

7.8 SECTIONS 伪指令

SECTIONS 伪指令：

- 说明输入段怎样被组合到输出段内。
- 在可执行程序中定义输出段。
- 规定在存储器内何处放置输出段。
- 允许重新命名输出段。

详见 2.3 节链接器如何处理段，2.4 节重定位，第 2.2.4 节子段。子段能使用户更加精确地处理段。

如果用户没有指定 SECTIONS 伪指令，链接器用默认算法组合并定位段。在 7.13 节中将详细讨论这种算法。

7.8.1 SECTIONS 伪指令的格式

SECTIONS 伪指令在命令文件中用 SECTIONS（大写）指定，后跟一系列输出段的说明，说明用大括号括起。

SECTIONS 伪指令的一般格式是：

```
SECTIONS
{
  name : [property [, property] [, property] ...]
  name : [property [, property] [, property] ...]
  name : [property [, property] [, property] ...]
}
```

每一个段的规定是以名字开头，定义一个输出段（输出段即输出文件中的段）。一个段名可以是一个子段的说明。段名的后面是一串用来指定段的内容和说明如何分配段的属性，属性用可选的逗号分隔。段具有的属性有：

- 装载位置 定义将段装入存储器内何处。

格式：

```
load = allocation or
load > allocation or
> allocation
```

allocation 是句法的一部分，它用来说明段如何放置在目标存储器中。

- 运行位置 定义段在存储器内何处运行。

格式：

```
run = allocation or
```

```
run > allocation
```

□ 输入段 定义构成输出段的输入段（即目标文件）。

格式：

```
{ input_sections }
```

□ 段类型 为特殊段的类型定义标志。

格式：

```
type = COPY or
type = DSECT or
type = NOLOAD
```

详见7.12节特殊段的类型。

□ 填充值 定义用来填充未初始化的空位值。

格式：

```
fill = value or
name : [properties] = value
```

详见7.15节创建和填充空位。

例 7.4 一个典型命令文件中 SECTIONS 伪指令的使用

```

/* **** */
/* SECTIONS 伪指令的命令文件例子 */
/* **** */
file1.obj file2.obj          /* 输入文件 */
                             /* 选项 */
SECTIONS -o prog.out
伪指令 SECTIONS
{
    .text: load = ROM, run = 800h
    .const: load = ROM
    .bss: load = RAM
    .vectors: load = 0FF80h
    {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
        endvec = .;
    }
    .data: align = 16
}

```

指令段

图 7.3 为例 7.4 中用 SECTIONS 伪指令定义的 5 个输出段：.vectors、.text、.const、.bss 和 .data。

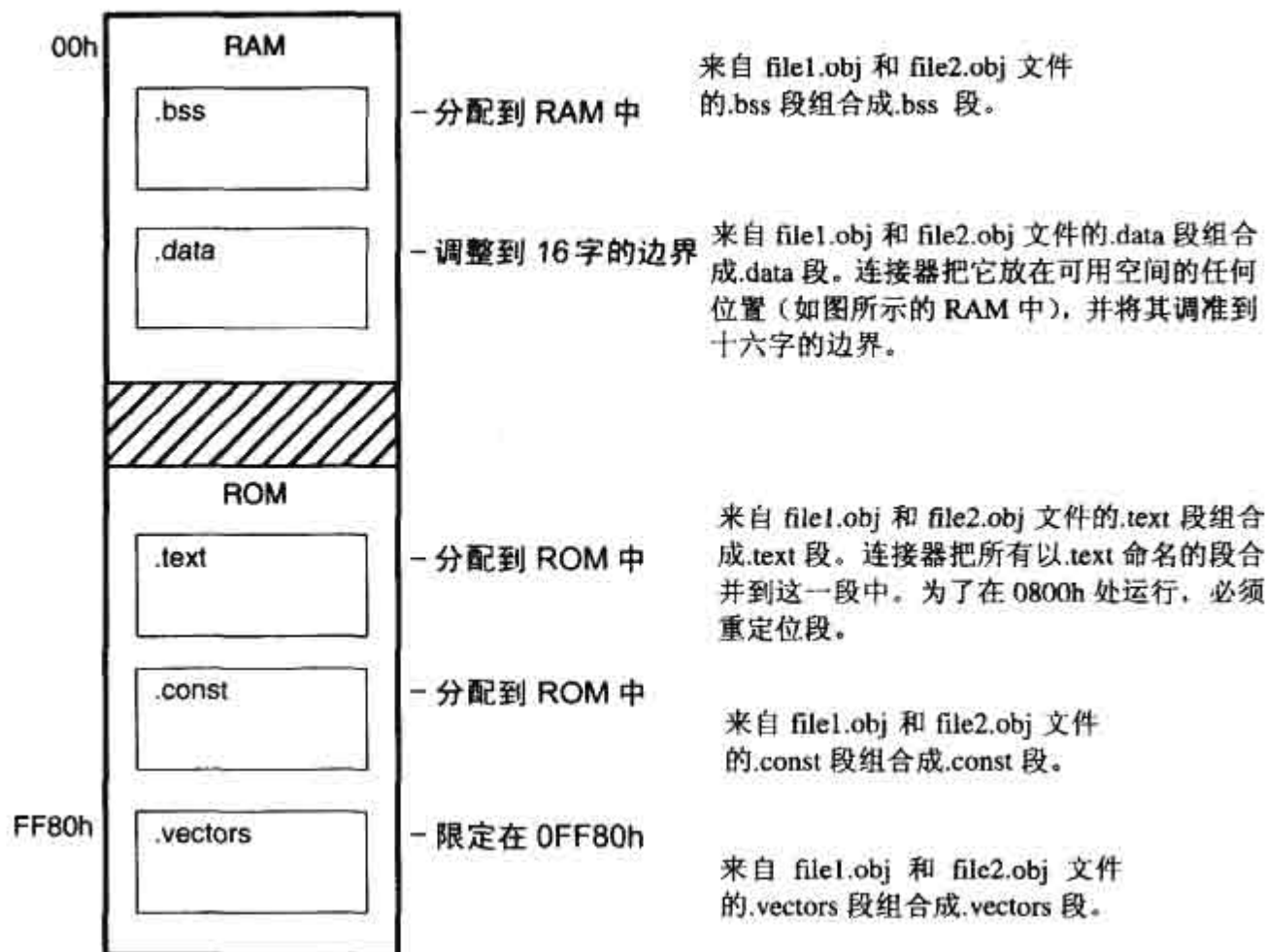


图 7.3 例 7.4 中定义的段的分配

7.8.2 地址分配

链接器为每一个输出段在目标存储器中分配两个地址：一个是段的装载地址，一个是段的运行地址。通常情况下，它们是同一个地址，可以认为每一段只有一个地址。在目标存储器中装载输出段和为它们分配地址的过程称为地址分配。关于装载和运行的地址分配见 7.9 节指定段运行时的地址。

如果没有告诉链接器如何分配一个段，它将使用默认算法为这个段分配地址。通常，链接器将为段配置适合的存储单元。也可以用 SECTIONS 伪指令和为分配段提供的指令取消对一个段的默认分配方式。

通过指定一个或多个定位参数可以控制地址分配。每一个参数包含一个关键字，可选的等号或大于号，以及可包括在圆括号中的选值。如果装载和运行的地址分配是分开来操作的，那么跟在关键字 LOAD 后面的所有参数用来进行装载地址分配，跟在关键字 RUN 后面的所有参数用来进行运行地址分配。

控制地址分配的参数有：

Blinding 把段分配在指定的地址。

```
.text: load = 0x1000
```

Named 把段分配到MEMORY定义的范围中。

memory 用于指定名字或属性的伪指令。

```
.text: load > ROM
```

Alignmen 用align关键字指定的段必须从一个地址边界开始。

```
.text: load = align(128)
```

Blocking 用block关键字指定的段在两个地址边界之间。如果这个段太大，它将从地址的边界开始。

```
.bss: load = block(0x80)
```

Page 指定所使用的存储器页。

```
.text: load = OVR_MEM PAGE 1
```

可以使用大于号和略去装载的关键字来分配装载（通常只有装载）地址。

```
.text: > ROM .text: {...} > ROM
.text: > 0x1000
```

如果使用了多个参数，可把它们串在一起，如下所示：

```
.text: > ROM align 16 PAGE 2
```

或者，使用圆括号来提高可读性，如下：

```
.text: load = (ROM align(16) page (2))
```

用户也可以利用一个指定的输入段来识别来自输入文件的段，这些输入文件被组合后形成一个输出段。详见 7.8.4 小节指定输入段。

以下说明上述分配参数。

1. 绑定 (Binding)

通过在输出段的后面跟一个地址，可以为该段指定一个起始地址。

```
.text: 0x1000
```

指定.text 段必须从 0x1000 开始。

绑定的地址必须是 22 位的常量。

输出段可绑定在所配置存储器的任何单元（假定有足够空间），但是它们不能重叠。如果没有足够的空间来把一个段绑定到指定的地址，链接器将提示错误信息。

注意：绑定与定位和命名寄存器是不兼容的。如果用户使用了定位或命名寄存器，就不能把一个段绑定到一个地址。如果用户这样做了，链接器将提示错误信息。

2. 命名存储器 (Named Memory)

可以把一个段分配到用 MEMORY 伪指令定义的存储区域。详见 7.7 节存储器伪指令。

例：命名存储区域并把段链接在里面

```
MEMORY
```



```

{
    ROM (RIX)    : origin = 0C00h,    length = 1000h
    RAM (RWIX)   : origin = 80h,     length = 1000h
}
SECTIONS
{
    .text : > ROM
    .data : > RAM
    .bss  : > RAM
}

```

上例中，链接器把 .text 段分配到名为 ROM 的区域内。 .data 和 .bss 输出段分配到 RAM 区域内。

同样，可以把一个段链接到一个具有特定属性的存储区中。为此定义了一系列代替存储器名字的属性（用圆括弧括起）。使用与 MEMORY 伪指令声明一样，可指定：

```

SECTIONS
{
    .text: > (X)    /* .text --> 可执行存储区 */
    .data: > (RI)   /* .data --> 可读或初始化存储区 */
    .bss : > (RW)   /* .bss --> 可读写的存储区 */
}

```

在例中，.text 输出段链接到 ROM 或 RAM 区域，因为这两个区域都具有 X 属性。 .data 输出段链接到 ROM 或 RAM 区域，因为这两个区域都具有 R 和 I 属性。 .bss 输出段必须链接到 RAM 区域，因为只有 RAM 区域具有 W 属性。

虽然链接器首先使用存储器的低地址，并且尽可能的避免产生存储器碎片，但是不能控制在一个指定的存储器区域内段具体分配到什么单元。在前面的例中假定不存在分配的冲突，那么 .text 段从地址 0 开始。如果一个段必须在规定的地址开始，那么使用绑定来代替命名存储器是一种办法。

3. 调准和分块 (Alignment and Blocking)

可以用关键字 **align** 告诉链接器把一个输出段放置在某个地址，这个地址落在 n 个字的边界内，其中 n 为 2 的幂次数。例如：

```
.text: load = align(128)
```

分配 .text 段使它落在 128 个字的边界内。

也可以在指定的存储器区域内分配一个段。例如：

```
.data align(128) : > RAM
```

此例中，.data 段落在 RAM 中 128 个字的边界内。

Blocking 是在一个块中任意单元分配一个段的一种较弱的分配方式。所指定的块大小必须是 2 的幂次数。例如：

```
bss: load = block(0x40)
```

分配.bss段使整个段包含在一个单独的64K字的页内,或在某一页的开始处。

可以单独使用 alignment (定位)、blocking (分块) 或者与存储区域一起使用,但是 alignment (定位) 和 blocking (分块) 不能一起使用。

4. 使用页的方法

使用指定地址页的方法,可以把一个段分配到由 MEMORY 伪指令命名的地址空间内。例如:

```
MEMORY
{
    PAGE 0 : PROG      : origin = 0800h, length = 0240h
    PAGE 1 : DATA      : origin = 0A00h, length = 02200h
    PAGE 1 : OVR_MEM    : origin = 02D00, length = 01000h
    PAGE 2 : DATA      : origin = 0A00h, length = 02200h
    PAGE 2 : OVR_MEM    : origin = 02D00, length = 01000h
}
SECTIONS
{
    .text: PAGE = 0
    .data: PAGE = 2
    .cinit: PAGE = 0
    .bss: PAGE = 1
}
```

在这个例中, .text 和 .cinit 分配到 PAGE 0。它们分配在 PAGE 0 内的任意单元。 .data 段被分配在 PAGE 2 内的任意单元。 .bss 段分配在 PAGE 1 内的任意单元。

可以把页方法与其他的方法结合起来使用,共同约束对一个指定地址空间的分配。例如:

```
.text: load = OVR_MEM PAGE 1
```

在这个例中, .text 段分配到指定的 OVR_MEM 存储器区。这里有两个命名为 OVR_MEM 的存储器区,所以必须指出使用的是哪一个。因而加上 PAGE 1,说明使用的 OVR_MEM 是 PAGE 1 地址空间,而不是 PAGE 2 地址空间。

7.3.3 规定输入段

规定输入段用来识别来自输入文件的段,这些输入文件组合成一个输出段。链接器按指定的顺序把输入段链接起来。输出段的大小是它包含的所有输入段大小的总和。

例 7.5 规定段的最通用方法,输入文件没有列出

```
SECTIONS
{
    .text:
    .data:
    .bss:
}
```

在例 7.5，链接器从输入文件中取出所有的.text 段，并把它们组合为.text 输出段。链接器按照在输入文件中遇到它们的顺序来链接它们。链接器对.data 和.bss 执行相似的操作。可以对输出文件使用规定的类型。

可以明确的指定用来形成一个输出段的输入段。每一个输入段用它的文件名和段名来区别。

```

SECTIONS
{
    .text :                /* 组合.text 输出段 */
    {
        f1.obj(.text)      /* 从 f1.obj 中链接.text 段 */
        f2.obj(sec1)       /* 从 f2.obj 中链接 sec1 段 */
        f3.obj             /* 从 f3.obj 中链接 ALL 段 */
        f4.obj(.text,sec2) /* 从 f4.obj 连接.text 和 sec2 */
    }
}

```

组成一个输出段的输入段没有必要使用相同的名字。作为构成输出段的输入段与输出段使用相同的名字也是不必要的。如果一个文件没有被 SECTIONS 列出，这个文件的所有段将被包括在输出文件中。如果附加的输入段与输出段有相同的名字而没有明确的用 SECTIONS 声明，它们被自动的链接在输出段的后面。例如，如果在前面的例中，链接器发现了更多的.text 段，这些段没有在任何单元用 SECTIONS 伪指令声明，链接器将在 f4.obj 的后面链接这些附加段。

在例 7.5 中的指定实际上可以简写成以下形式：

```

SECTIONS
{
    .text: { *(.text) }
    .data: { *(.data) }
    .bss: { *(.bss) }
}

```

指定*(.text) 意味着包含所有输入文件中没有分配地址的.text 段。下述情况下可使用这个格式：

- 希望输出段包含有特定名称的输入段，但是输出段的名称与输入段的名称不同。
- 希望链接器在处理附加输入段或大括号中的命令之前分配输入段。

举例说明上述两种情况：

```

SECTIONS
{
    .text : {
        abc.obj(xqt)
        *(.text)
    }
    .data : {
        *(.data)
    }
}

```

```

        fil.obj(table)
    }
}

```

在这个例中，.text 输出段包含了文件 abc.obj 中的名为 xqt 的段，xqt 段后面是所有的.text 输入段。.data 输出段包含了所有的.data 输入段，它的后面是来自文件 fil.obj 的段表。此方法包含了所有未定义的段。例如，如果在链接器遇到*(.text)之前，在另一个输出段中已包含其中一个.text 输入段，那么在第二个输出段中不会包含第一个.text 输入段。

7.9 指定一个段的运行地址

有时要把代码装入存储器的一个区，而在另一个区运行它。例如，在慢速的外部存储器中，存有对性能影响比较大的代码，而这些代码又不得不装载在慢速的外部存储器中，但它们需要在快速的外部存储器中运行。

链接器提供了一个简单的方法来实现。可以使用 SECTIONS 伪指令使链接器为一个段分配两次地址：一次设置装载地址，一次设置运行地址。例如：

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

装载地址使用 load 关键字，运行地址使用 run 关键字。

详见 2.4.1 节运行时重定位。

7.9.1 指定装载和运行地址

对于段来说，装载地址就是决定原始数据放置在什么地方。对段的任何引用（如其中的表）都是访问它的运行地址。应用时必须把段从它的装载地址复制到它的运行地址；当指定了分开的运行地址时，这个过程不会自动发生。

如果为一个段只指定一次地址分配（要么是装载，要么是运行），这个段将只被分配一次而且在同一个地址上装载和运行。如果为装载和运行都分配了地址，对段的地址分配就好像是分配了两个相同大小的段。意味着这两次分配都在存储器映像图中占有空间，二者不能重叠，也不能与其他的段重叠（UNION 伪指令提供了一种重叠段的方法，见 7.10.1 节）。

如果装载或运行地址有附加的参数，例如 alignment（定位）或 blocking（分块），可以在合适的关键字后面列出它们。关键字 load 后面的所有与分配有关的参数都影响装载地址，关键字 run 后面的参数影响运行地址。装载和运行的分配是完全独立的，其中一个对另一个没有影响。也可以先指定 run，再指定 load。使用圆括号可以提高可读性。

下面的例指定了装载和运行地址：

```

.data: load = ROM, align = 32, run = RAM /*仅指定装载地址 */
.data: load = (ROM align 32), run = RAM /*仅指定装载地址 */
.data: run = RAM, align 32              /*调整到 32，在 RAM 中运行 */
      load = align 16                   /*调整到 16 在任意地方装载*/

```


7.9.2 未初始化的段

未初始化的段（例如.bss）不需装载，所以它们的有效地址只是运行地址。链接器只分配一次未初始化段，如果用户同时指定了运行和装载地址，链接器发出警告并忽略装载地址。如果只指定了一个地址，不管指定的是运行地址，还是装载地址，链接器都把它作为运行地址。这个例为未初始化段指定了装载和运行地址：

```
.bss: load = 0x1000, run = RAM
```

提出一个警告，装载被忽略，在 RAM 中分配空间。以下所有例均有相同的效果，在 RAM 中分配.bss 段。

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

7.9.3 用.label 伪指令访问装载地址

通常在段中对符号的任何引用都是访问它运行时的地址。然而，在运行时访问装载时的地址也是必要的。特别是从装载地址复制一个段的代码到运行地址时必须能寻址装载地址。.label 伪指令定义了一个访问段装载地址的特殊符号。因此，尽管一般符号用运行地址重定位了，而.label 符号用装载地址重定位。详见.label 伪指令。

例 7.6 说明了.label 伪指令的使用。图 7.4 说明了例 7.6 执行时的运行情况。

例 7.6 把段从 ROM 复制到 RAM

(a) 汇编语言文件

```

;-----
; 定义一个段，它从 DATA 中复制到 PROGRAM
;-----
.sect ".fir"
.label fir_src      ; 段的转载地址
fir:                ; 段的运行地址
    <code here>     ; 段的代码
    .label fir_end  ; 段的装载地址结束
;-----
; 把 DATA 中的.fir 段复制到 PROGRAM
;-----
.text
MOV XAR6, fir_src
MOV XAR7, #fir
RPT #(fir_end - fir_src - 1)
_PWRITE *XAR7, *XAR6++
;-----
; 跳转到段，现处于 RAM 中
;-----
B fir

```


(b) 链接器命令文件

```

/*****
/* FIR 例的实际链接命令文件
*****/
MEMORY
{
    PAGE 0 : RAM      : origin = 0800h,   length = 02400h
    PAGE 0 : PROG     : origin = 02C00h,   length = 0D200h
    PAGE 1 : DATA    : origin = 0800h,   length = 0F800h
}
SECTIONS
{
    .text: load = PROG PAGE 0
    .fir: load = DATA PAGE 1, run RAM PAGE 0
}

```

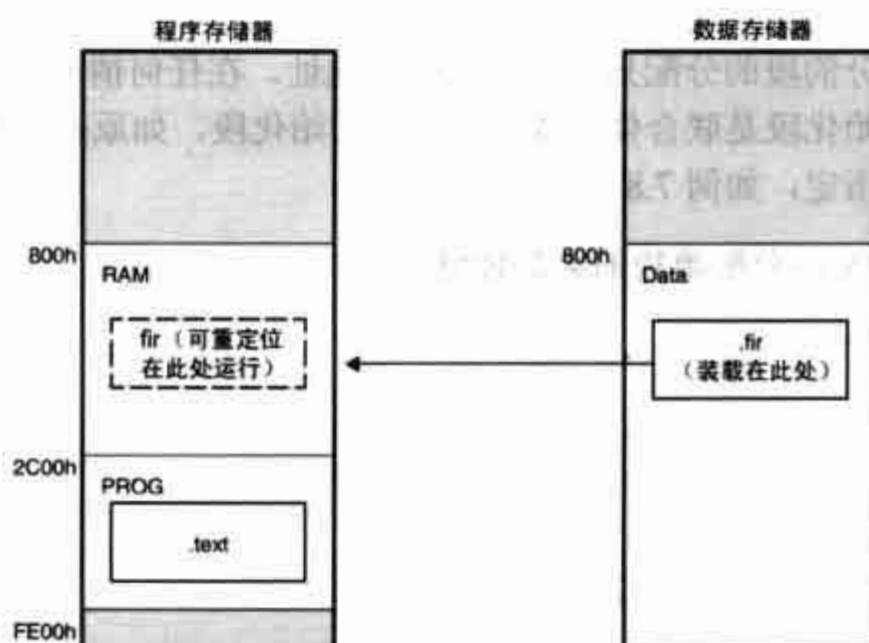


图 7.4 例 7.6 执行时的运行情况

7.10 UNION 和 GROUP 语句的使用

GROUP 和 UNION 两个 SECTIONS 语句允许用户保留存储器。段联合 (Unioning) 可以使链接器为它们分配相同的运行地址，段分组 (Grouping) 可以使链接器为它们在存储器中分配相邻的地址。它们可以访问段、子段或归档库的成员。

7.10.1 用 UNION 语句重叠段

对于某些应用，用户希望把多个段分配在同一个地址上运行。例如，可能在程序执行的不同阶段，在快速的外部存储器中运行几个程序。或者，想分时共享一个存储器块的几个数据块。SECTIONS 伪指令中的 UNION 语句提供了一种方法，即把几个段分配在同一个运行地址上。

在例 7.7 中来自 file1.obj 和 file2.obj 的.bss 段在 RAM 中分配了一个相同的地址。在存储器映像图中，联合体占用的空间大小是它最大成员的大小。联合体的成员保持相互独立，它们被作为一个单元分配。

例 7.7 UNION 语句

```
SECTIONS
{
    .text: load = ROM
    UNION: run = RAM
    {
        .bss1: { file1.obj(.bss) }
        .bss2: { file2.obj(.bss) }
    }
    .bss3: run = RAM { globals.obj(.bss) }
}
```

作为联合体一部分的段的分配只影响它的运行地址。在任何情况下装载时这些段都不能重叠。如果一个初始化段是联合体的成员（一个初始化段，如原始数据的.text 段），它的装载地址必须单独指定，如例 7.8 所示。

例 7.8 为 UNION 段分配单独的装载地址

```
UNION: run = RAM
{
    .text1: load = ROM, { file1.obj(.text) }
    .text2: load = ROM, { file2.obj(.text) }
}
```

例 7.7 和例 7.8 中存储器分配情况如图 7.5 所示。

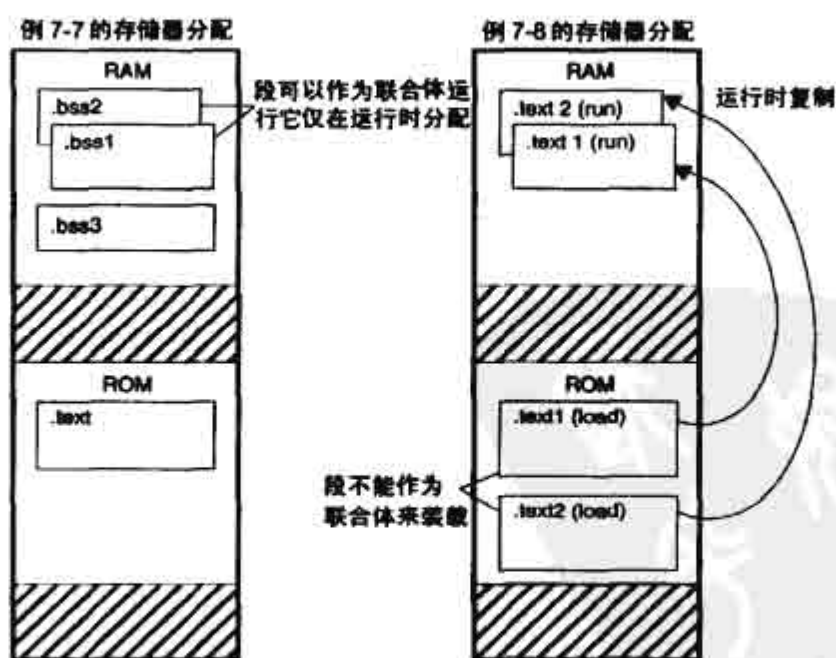


图 7.5 例 7.7 和例 7.8 的存储器分配示意图

因为.text 段包含数据，虽然可以作为一个联合运行，但是不能作为一个联合装载。所以每一个都要有自己的装载地址。如果用户在 UNION 语句中为初始化段错误的提供了一

个装载地址，链接器将提出警告并且在所配置的存储器内任何可能的单元内分配装载空间。

未初始化的段不需装载，因此不需要装载地址。

UNION 语句只提供运行地址的分配，所以用它来为联合体指定一个装载地址是毫无意义的。为了实现分配的目的，联合体处理成一个未初始化的段：任何指定的分配地址都被认为是运行地址，如果指定了运行地址又指定了装载地址，链接器将提出警告，并忽略装载地址。

7.10.2 把输出段在一起分组

SECTIONS 伪指令的 GROUP 选项强制对几个输出段连续分配地址。例如，假定名为 term_rec 的段含有放在 .data 段表中的一个终止记录。用户可以强制链接器把 .data 和 term_rec 分配在一起。见例 7.9。

对单个的输出段，可以使用绑定、调准、命名存储器来分配作用与一个 GROUP 伪指令一样。在下面的例中，GROUP 被绑定到地址 1000h。这意味着，.data 段被分配在存储器中的 1000h，term_rec 紧跟其后。

例 7.9 在一起分配段

```
SECTIONS
{
    .text          /* 正常输出段 */
    .bss           /* 正常输出段 */
    GROUP 1000h :  /* 指定一组段 */
    {
        .data      /* 组中的第一个段 */
        term_rec   /* 在 .data 后紧接着分配空间 */
    }
}
```

注意： GROUP 不能为一个段指定地址。当使用 GROUP 选项时，只能对组（作为整体）使用绑定、调准或命名指定的存储器。不能对一个组内的段使用绑定、调准或命名到指定的存储器。

7.11 重 叠 页

某些器件使用这样的存储器配置，在该配置中，所有或部分存储器空间被影子存储器（shadow memory）重叠。这样允许系统根据硬件的选择信号，把物理存储器的不同存储区分配到单一的地址范围或把一个存储器区分配到不同的地址范围。换句话说，多个物理存储器区在同一个地址范围内相互重叠。用户可能希望，链接器把各个输出段装载到每一个存储区中或在装载时没有被映射到存储区中。

链接器通过提供重叠页支持这一功能。每一页分别表示一个已用 MEMORY 伪指令配置的地址范围。然后就可以用 SECTIONS 伪指令来指定被映射到不同页的段。

注意：重叠段和重叠页是不同的。联合（UNION）和重叠页（overlay page）的功能听起来很相似，因为它们都是处理重叠问题。事实上它们有很大的区别，UNIONS 允许多个段在同一个存储器空间内相互重叠。而重叠页定义了多个存储器空间。

7.11.1 用 MEMORY 伪指令定义重叠页

对于链接器而言，每一个重叠页表示由可寻址地址组成的一个完全独立的存储器空间。用这种方式，可链接两个或多个位于不同页的段。

页从 0 开始顺序编号。如果没有使用 PAGE 选项，链接器把初始化的段分配到 PAGE 0（程序存储器），把未初始化的段分配到 PAGE 1（数据存储器）。

7.11.2 重叠页实例

假定用户的系统可在两个物理存储器的存储区之间选择，数据存储空间 PAGE 1 的地址范围是 A00h~FFFFh；PAGE 2 的地址范围是 0A00h~2BFF。虽然在一个时刻只能选择一个存储器，但是用户可以用不同的数据初始化每一个存储器。

例 7.10 说明了如何用 MEMORY 伪指令来获得这种配置。

例 7.10 使用重叠页的 MEMORY 伪指令

```
MEMORY
{
    PAGE 0 : RAM          :origin = 0800h,   length = 0240h
             : PROG        :origin = 02C00h,  length = 0D200h
    PAGE 1 : OVR_MEM       :origin = 0A00h,   length = 02200h
             : DATA        :origin = 02C00h,  length = 0D400h
    PAGE 2 : OVR_MEM       :origin = 0A00h,   length = 02200h
}
```

例 7.10 定义了 3 个独立的地址空间：

- PAGE 0 定义了 RAM 程序存储器空间的一个区域和剩余的程序存储器空间。
- PAGE 1 定义了第一个重叠的存储器区域和剩余的数据存储器空间。
- PAGE 2 为数据空间定义了另一个重叠存储器区域。

两个 OVR_MEM 范围占用同一个地址范围。这是可能的，因为每一个范围在不同的页上，它们表示不同的存储器空间。

7.11.3 在 SECTIONS 伪指令中使用重叠页

假定在例 7.10 中使用 MEMORY 伪指令。进一步假定代码包含标准的段，用户还想把 4 个代码模块装载到数据存储器空间并在 RAM 中运行。例 7.11 说明了如何通过 SECTIONS 伪指令重叠来实现这些目的。

例 7.11 在例 7.10 中重叠页的 SECTIONS 伪指令

```

SECTIONS
{
    UNION : run = RAM
    {
        S1 : load = OVR_MEM PAGE 1
        {
            s1_load = 0A00h;
            s1_start = .;
            f1.obj (.text)
            f2.obj (.text)
            s1_length = . - s1_start;
        }
        S2 : load = OVR_MEM PAGE 2
        {
            s2_load = 0A00h;
            s2_start = .;
            f3.obj (.text)
            f4.obj (.text)
            s2_length = . - s2_start;
        }
    }

    .text: load = PROG PAGE 0
    .data: load = PROG PAGE 0
    .bss : load = DATA PAGE 1
}

```

f1、f2、f3 和 f4 四个模块中，f1 和 f2 被组合到输出段 S1，f3 和 f4 被组合到输出段 S2。PAGE 为 S1 和 S2 的说明，告诉链接器把这些段链接到相应的页。当程序被装载时，装载器能配置硬件，从而使每个段装载到合适的存储器中。

7.11.4 对重叠页的存储器分配

图 7.6 显示了在例 7.10 中用 MEMORY 定义的重叠页和在例 7.11 中用 SECTIONS 伪指令定义的重叠页。

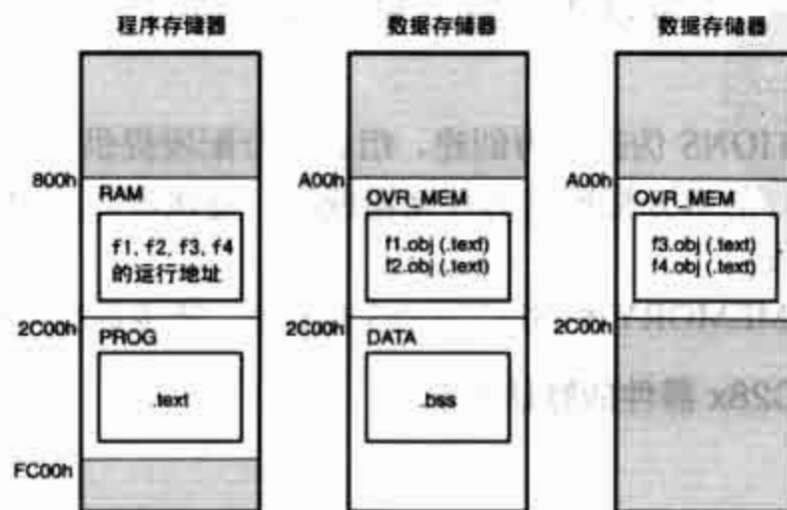


图 7.6 在例 7.10 和例 7.11 中定义的重叠页

7.12 特殊段类型

可以分配 3 种特殊的类型给输出段：DSECT、COPY 和 NOLOAD。这些类型对程序的链接和装载方式有影响，例如：

```
SECTIONS
{
    sec1: load = 0x00002000, type = DSECT {f1.obj}
    sec2: load = 0x00004000, type = COPY {f2.obj}
    sec3: load = 0x00006000, type = NOLOAD {f3.obj}
}
```

□ DSECT 类型创建具有下列特征的虚段：

- 它不包含在输出段的存储器分配中。它不占用任何的存储器，它不包含在存储器映像列表中。
- 它可以覆盖其他的输出段、其他的 DSECT 类型的段或未配置的存储器。
- 通常情况下，它的全局符号要重新定位。如果已经装载了 DSECT 段，它会以相同的值出现在输出文件符号表中。这些符号可被其他的输入模块引用。
- 在这种段中发现未定义的外部符号将会到指定的归档库中搜索。
- 它的内容，重定位信息和行号信息不会在输出文件中出现。

在前例中，虽然这些段被链接在地址 0x00002000 处，但是来自 f1.obj 的段都没有被分配，但却重新定位了所有的符号。其他的段可以引用在 sec1 段中的所有符号。

- COPY 段除了它的内容和相应的信息被写到输出文件中外，其他方面与 DSECT 段相似。包含 TMS320C28x C/C++ 编译器初始化表的 .cinit 段在运行初始化模式下具有这个属性。
- NOLOAD 段与普通的段不同：它的内容，重定位信息，行号信息不在输出文件中出现。链接器为这个段分配空间，它会出现在存储器映像图列表中。

7.13 默认分配

MEMORY 和 SECTIONS 伪指令为创建、组合和分配段提供了灵活的方法。但是链接器仍然会处理用户所选择的未分配地址的存储器或段。链接器在用户提供的规定范围内，使用默认值创建和分配段。

如果用户没有使用 MEMORY 和 SECTIONS 伪指令，链接器会分配输出段，如例 7.12。

例 7.12 TMS320C28x 器件的默认分配

```
MEMORY
{
    PAGE 0: PROG    : origin = 0x000040 length = 0x3fffc0
}
```

```
    PAGE 1: DATA      : origin = 0x000000 length = 0x010000
    PAGE 1: DATA1     : origin = 0x010000 length = 0x3f0000
  }
  SECTIONS
  {
    .text      : PAGE = 0
    .data      : PAGE = 0
    .cinit     : PAGE = 0      /* 仅在 C 程序中使用 */
    .bss       : PAGE = 1
  }
```

所有的.text 输入段组合在一起形成可执行文件中的一个.text 输出段, 所有的.data 输入段组合在一起形成输出段中的一个.data 段。

7.13.1 输出段的形成

如果使用了 SECTIONS 伪指令, 链接器就不执行默认的分配。执行的分配方式取决于 SECTIONS 伪指令指定的规则。常用算法将在 7.13.2 节中讨论。

输出段可由如下两种方式形成:

- (1) SECTIONS 伪指令定义的结果。
- (2) 把具有相同名字的输入段组合到 SECTIONS 伪指令中未定义的输出段中。

如果输出段是 SECTIONS 伪指令产生的结果, 那么这个段的内容完全由伪指令来决定。

如果输出段组合 SECTIONS 伪指令没有规定的输入段, 链接器将组合所有的具有相同名字的输入段形成一个用相同名字命名的输出段。例如, 假定文件 f1.obj 和 f2.obj 都含有名为 Vectors 的段, 并且 SECTIONS 伪指令没有为它们指定输出段。链接器组合来自输入文件的两个 Vectors 段而形成一个名为 Vectors 的输出段, 然后把它们分配到存储器中, 并包括在输出文件中。

默认的情况下, 当链接器创建一个不是 SECTIONS 指定的输出段时不会显示信息。可以使用 -W 链接器选项, 要求链接器创建新的输出段时显示信息, 详见 7.4.18 小节。

链接器确定了输出段的组成以后, 必须把它们分配到配置的存储器中。MEMORY 伪指令指定存储器的配置, 链接器使用的默认配置见例 7.12 (关于配置存储器详见 7.7 节 MEMORY 伪指令)。

7.13.2 默认分配算法

链接器的分配会尽可能地减少存储器碎片。这样可提高存储器的利用率, 使用户的程序更适应存储器。包括以下几步:

- (1) 指定了绑定地址的每一个输出段都会被分配到那个地址中。
- (2) 包含在指定的存储器区域内或在拥有属性的存储器区域内的每一个输出段都会被分配, 且每一个输出段总是分配在指定区域的第一个可用的空间中。
- (3) 其余的段以它们定义的顺序进行分配。SECTIONS 伪指令没有指定的段, 按遇到它们的顺序进行分配。每一个输出段总是分配在第一个可用的空间中。

7.14 链接时给符号赋值

链接器赋值语句允许用户定义外部（全局）符号，在链接时为它们赋值。可以利用这个特点把一个变量或指针初始化为一个与存储器分配相关的值。

7.14.1 赋值语句的格式

链接器中的赋值语句格式与 C 语言程序中的赋值语句相似：

```
symbol = expression      ; 把表达式的值赋予符号  
symbol += expression     ; 把表达式的值加到符号上  
symbol -= expression     ; 从符号减去表达式的值  
symbol *= expression     ; 符号乘以表达式  
symbol /= expression     ; 符号除以表达式
```

符号应在外部定义，否则，链接器将定义一个新的符号并把它加到符号表中。表达式必须遵守 7.14.3 小节赋值表达式中定义的规则。赋值表达式必须以分号结束。

链接器分配完所有的输出段后再处理赋值语句。如果表达式中含有符号，符号的地址反映了这个符号在可执行文件中的地址。

例如，假定一个程序从两个外部符号 Table1 和 Table2 所标识的表中读入数据。程序用 cur_tab 作为当前表的地址。cur_tab 必须指向 Table1 和 Table2 中的一个。这些可以在汇编代码中实现，但当表改变时必须重新汇编程序。用户可以在链接时用链接器的赋值语句来分配 cur_tab：

```
prog.obj          /* 输入文件 */  
cur_tab = Table1; /* 将一个表赋值给 cur_tab */
```

7.14.2 将 SPC 赋值到一个符号

用圆点“.”表示的特殊符号代表在分配期间段程序计数器（SPC）的当前值。在一个段内 SPC 跟踪当前单元。链接器的“.”符号与汇编器的\$符号相似。符号“.”只能用在 SECTIONS 伪指令的赋值语句中，符号“.”只有在赋值期间和 SECTIONS 伪指令的分配过程中才有意义（见 7.8 节 SECTIONS 伪指令）。

符号“.”指当前的运行地址，而不是段的当前装载地址。

假定程序需要知道.data 段的开始地址，可使用.global 伪指令创建一个名为 Dstart 的外部未定义变量，然后为 Dstart 赋值：

```
{  
    .text: {}  
    .data: { Dstart = .; }  
    .bss : {}  
}
```

这样就定义 Dstart 为.data 段的首地址 (Dstart 在.data 段分配之前赋值)。链接器重新定位所有对 Dstart 的引用。

一种特殊类型的赋值方式把值赋予“.”符号, 并将调整输入段内的 SPC 的值, 并在两个输入段之间创建空位。所有分配给符号“.”的值都将创建相对于段首地址的空位, 而不是实际上“.”符号代表的地址。给空位和“.”赋值将在 7.15 节中讨论。

7.14.3 赋值表达式

链接器赋值表达式的规则如下:

- 表达式可以包含全局符号、常量、表 7.2 列出的 C 语言程序运算符。
- 所有的数字被作为 32 位长整型数。
- 链接器对常量的识别与汇编器对常量的标识相同。认为数字是十进制数, 除非它们有后缀 (十六进制 H 或 h; 八进制 Q 或 q)。也认可 C 语言的前缀。十六进制数必须以数字开头。不承认二进制常量。
- 表达式中的符号只有符号的地址值, 不执行类型检查。
- 链接器的表达式可以是绝对的或可重定位的。如果表达式包含任何一个可重定位符号, 那么它就是可重定位的。反之, 这个表达式是绝对的。如果一个符号分配了一个可重定位的表达式值, 那么这个符号就是可重定位的, 如果它被分配了一个绝对表达式的值, 那么它就是绝对的。

链接器以表 7.2 列出的优先级顺序支持 C 语言程序的运算符。同一组中的操作符有相同的优先级。除了表 7.2 的运算符外, 链接器也支持定位运算符, 定位运算符允许在输出段中把符号定位在 n 个字节的边界 (n 是 2 的幂)。例如, 表达式:

```
= align(16);
```

该表达式将使输出段定位在 16 字的边界上。所有用特殊定位运算符对“.”符号的分配, 都会以相同的方式影响段的定位。如果使用多个定位语句, 输出段将定位在指定的最大分配边界上。

表 7.2 在表达式中使用一组算子的优先级

组 1 (最高优先级)	组 6
! 逻辑非	& 按位与
~ 按位非	
- 去反	
组 2	组 7
* 乘	按位或
/ 除	
% 取模	
组 3	组 8
+ 加	&& 逻辑与
- 减	

续表

组 4	组 9
>> 算术右移 << 算术左移	逻辑或
组 5	组 10 (最低优先级)
== 等于 != 不等于 > 大于 < 小于 <= 小于或等于 >= 大于或等于	= 赋值 += $A+=B \rightarrow A=A+B$ -= $A-=B \rightarrow A=A-B$ *= $A*=B \rightarrow A=A*B$ /= $A/=B \rightarrow A=A/B$

7.14.4 链接器定义的符号

链接器根据在汇编源程序中使用的段的不同，自动定义了几个符号。程序可以在运行时使用这些符号来确定一个段在何处被链接。因为这些符号是外部的，他们会出现在链接器的映像图中。用.global 伪指令声明的所有符号可被任何汇编模块调用。为了创建符号，用户必须在源程序模块中使用对应的段。如下所示把数值赋予这些符号：

.text 被赋予.text 段的首地址（可执行代码的开始）。
 .etext 被赋予.text 段结束后的第一个地址（可执行代码的结束）。
 .data 被赋予.data 段的首地址（初始化的数据表的开始）。
 edata 被赋予.data 段结束后的第一个地址（初始化的数据表的结束）。
 .bss 被赋予.bss 段的首地址（未初始化的数据段的开始）。
 end 被赋予.bss 段结束后的第一个地址（未初始化的数据段的结束）。

只为了支持 C 语言而定义的符号（-c 或-cr 选项）：

__STACK_SIZE 赋予.stack 段的大小。
 __SYSMEM_SIZE 赋予.sysmem 段的大小。

7.15 创建和填充空位

链接器允许在输出段中创建一个区域，而没有任何东西链接到里面。这样的区域被称为空位（holes）。特殊的情况下，未初始化的段也被作为空位来对待。本节将讨论链接器如何处理空位和如何用给定的值填充空位（未初始化段）。

7.15.1 初始化段和未初始化段

一个输出段包含如下情况之一：

- 整个段的原始数据。

□ 没有原始数据。

具有原始数据的段被称为初始化段，意味着这个目标文件含有实际存储器映像的内容。当装载这个段时，此映像就被装入存储器中而且由段指定起始地址。如果已有内容被汇编入.text段和.data段中，那么这两个段总是有原始数据。用.sect汇编伪指令命名的段也具有原始数据。

默认的情况下，.bss段和用.usect伪指令指定的段没有原始数据。它们在存储器中占用空间但是没有实际的内容。未初始化的段通常在快速的外部存储器中为变量保留空间。在目标文件中，未初始化的段有一个标准的段标题，并且在段里面定义了符号；虽然没有存储器映像，但可以存储在这个段中。

7.15.2 创建空位

可以在初始化的输出段中创建空位。在输出段中，当用户强制链接器在几个输入段之间保留额外空间的时候就创建了空位。当这个空位被创建时，链接器就必须为空位提供原始的数据。

只能在输出段中创建空位。在输出段之间可以存在空间，但它不是空位。要填充输出段之间的空间，见7.7.2小节MEMEORY伪指令格式。

为了在输出段中创建空位，必须在链接器输出段定义中使用链接器特殊类型的赋值语句。通过加上一个数来赋更大的值或者定位在一个地址边界，用这个赋值语句来修改SPC（用“.”表示）。算符、表达式、赋值语句的格式见7.14节。

例：使用赋值语句在输出段中创建空位。

```
SECTIONS
{
  outsect:
  {
    file1.obj(.text)
    . += 100h;          /* 创建一个长度为 100h 的空位 */
    file2.obj(.text)
    . = align(16);      /* 创建一个空位，调整 SPC */
    file3.obj(.text)
  }
}
```

创建 outsect 输出段，方法如下：

- (1) 链接来自file1.obj的.text段。
- (2) 链接器创建一个 256 (100h) 字的空位。
- (3) 在空位的后面链接来自 file2.obj 的.text 段。
- (4) 通过把 SPC 定位在一个 16 字的边界，创建另一个空位。
- (5) 最后，链接来自 file3.obj 的.text 段。

在一个段中所有赋值给“.”符号的值均引用相对地址。链接器处理赋值“.”符号就像这个段从地址 0 开始一样（即使指定了一个绑定地址）。例中.=align(16)语句把 outsect 段

分配在一个 16 字的边界上。所有使用特殊赋值运算符给“.”符号的赋值都将定位，并以相同的方式影响输出段。如果使用了多个赋值语句，输出段定位在指定的最大分配范围内。

符号“.”是指段的当前运行地址而不是当前装载地址。

“-”号运算在“.”符号表达式中是非法的。例如，在赋值语句中对“.”符号使用“=”运算符是非法的。在赋值语句中对“.”符号的常用运算符是“+=”和“align”。

如果一个输出段中包含有特定类型的所有输入段（例如.text），用户可以在输出段的开始或结束使用下面的语句创建空位：

```
.text: { . += 100h; }    /* 在开始创建处建空位 */
.data: {
*(.data)
. += 100h; }           /* 在结尾处创建空位 */
```

在一个输出段中创建空位的另一个方法是：合并未初始化段和初始化段，形成一个输出段。在这种情况下，链接器把未初始化的段视作空位。下面的例说明了这种方法：

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
        file1.obj(.bss)           /* 它变为空位 */
    }
}
```

因为.text 段有原始数据，所以所有的.outsect 段也包含原始数据。因此，未初始化.bss 段就成了空位。

只有当未初始化段与初始化段合并的时候，未初始化段才成为空位。如果几个未初始化的段链接在一起，生成的输出段仍是未初始化的。

7.15.3 填充空位

当初始化输出段中有空位存在时，链接器必须提供原始数据来填充它。链接器通过存储器用一个 16 位的填充值来反复填充空位，直到填满为止。链接器以下面的方式来决定填充值：

(1) 如果空位是通过合并未初始化段和初始化段形成的，可以为未初始化段指定一个填充值。下面指定了一个 16 位的常量：

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
        file2.obj(.bss) = 00FFh /* 用 0FFh 填充空位 */
    }
}
```

(2) 也可在段定义的后面, 为一个输出段中的所有的空位指定一个填充值:

```
SECTIONS
{
    outsect:fill = 0FF00h
    /* 用 0FF00h 填充空位 */
    {
        . += 10h; /* 创建空位 */
        file1.obj(.text)
        file1.obj(.bss) /* 创建另一空位 */
    }
}
```

(3) 如果没有为空位指定一个初始化的值, 链接器将用 `-f` 选项指定的值来填充空位。例如, 假定名为 `link.cmd` 的文件中包含下面的 `SECTIONS` 伪指令:

```
SECTIONS
{
    .text: { . = 100; } /* 创建一个 100 字的空位 */
}
```

现在用 `-f` 选项调用链接器:

```
lnk2000 -f 0FFFFh link.cmd
```

这时将用 `0FFFFh` 填充空位。

(4) 如果没有使用 `-f` 选项来调用链接器或者另外指定一个填充值, 链接器将用 `0` 来填充空位。

不管在初始化的输出段中的空位是何时创建的, 这个空位都将随同所用的填充值在链接映像中定义。

7.15.4 未初始化段的显式初始化

可以在 `SECTIONS` 伪指令中指定一个明确的填充值, 强制链接器用这个值初始化未初始化的段。这将使得整个段都有原始数据 (填充值)。例如:

```
{
    .bss: fill = 1234h /* 用 1234h 填充 .bss */
}
```

注意: 因为填充段 (即使是填 0) 会导致在输出文件中为整个段产生原始数据, 所以如果用户为大的段或空位指定了填充值, 那么用户的输出文件将变得很大。

7.16 部分 (增量) 链接

已链接的输出文件可以和附加模块再一次链接, 这就是所谓的部分或增量链接。部分

链接允许用户对大的应用程序进行划分，单独链接每一部分，然后把各部分链接起来生成最终的可执行程序。

下面的方法将指导如何产生一个可再链接的文件：

- 链接器产生的中间文件必须有可重定位的信息。除了最后的链接外，所有的链接都要使用-r 选项（见 7.4.1 小节重定位）。
- 中间文件必须有符号信息。默认情况下，链接器在输出文件中保留符号信息。如果想再链接一个文件就不需要使用-s 选项，因为-s 选项将在输出模块中删除符号信息（见 7.4.15 小节删除符号信息（-s 选项））。
- 中间的链接步骤只与输出段的形成有关而与定位无关。所有的定位、绑定和 MEMORY 伪指令应在最后的链接步骤中执行。
- 如果中间文件有与其他文件中名字相同的全局符号，并且用户想把它作为静态符号（只在中间文件中可见），就必须用-h 选项链接它（见 7.4.7 小节所有的静态化全局变量）。
- 如果链接 C 代码，在最后的链接步骤之前不要使用-c 或-cr 选项。用-c 或-cr 选项对链接器的每次调用，都会使链接器创建一个入口（见 7.4.3 小节 C 语言程序选项）。
- 如果每一个输入段都没含有需要重定位的信息，那么就可成功的再次链接一个绝对文件。也就是说，每一个文件都没有未确定的引用，并且链接器创建它们的时候被绑定到同一个虚地址。

下面的例说明了如何使用部分链接：

第一步：链接文件 file1.com；使用-r 选项在 tempout1.out 输出文件中保留重定位的信息。

```
lnk2000 -r -o tempout1 file1.com
```

file1.com 包含：

```
SECTIONS
{
    ssl: {
        f1.obj
        f2.obj
        .
        .
        .
        fn.obj
    }
}
```

第二步：链接 file2.com 文件；使用-r 选项在 tempout2.out 输出文件中保留重定位的信息。

```
lnk2000 -r -o tempout2 file2.com
```

file2.com 包含：

```
SECTIONS
```



```
{
    ss2: {
        g1.obj
        g2.obj
        .
        .
        gn.obj
    }
}
```

第三步：链接 tempout1.out 和 tempout2.out。

```
lnk2000 -m final.map -o final.out tempout1.out tempout2.out
```

7.17 链接 C 代码

C/C++编译器生成可以汇编和链接的汇编语言源代码。例如，一个包含有模块 prog1 和 prog2 等的 C 语言程序可被汇编，链接生成一个名为 prog.out 的可执行文件：

```
lnk2000 -c -o prog.out prog1.obj prog2.obj ... rts.lib
```

-c 选项告诉链接器使用 C 环境中定义的特殊规则。归档库 rts2800.lib 含有 C 运行时支持的函数。

关于 C 的详细情况，包括运行时的环境和运行时支持的函数见 *TMS320C28x Optimizing C/C++ Compiler User's Guide*。

7.17.1 运行中的初始化

所有的 C 语言程序必须与名为 boot.obj 的目标模块链接。当一个程序开始执行的时候，它首先执行 boot.obj。boot.obj 含有符号代码和为运行环境初始化的数据。这个模块完成下面的任务：

- (1) 建立系统的堆栈。
- (2) 处理运行时的初始化表和自动初始化全局变量（当用 -c 选项调用链接器的时候）。
- (3) 调用 _main。

运行时支持的目标库 rts.lib 包括 boot.obj。用户可以：

- 作为输入文件包括 rts.lib（当用户使用 -c 或 -cr 选项的时候链接器会自动提取出 boot.obj）。
- 使用归档器从库中提取出 boot.obj，然后直接链接这个模块。

7.17.2 目标库和运行时的支持

TMS320C28x 优化 C/C++编译器用户手册说明了包含在 rts.src 中运行时支持的附加函数。如果使用了这些函数中某个函数，就必须在用户的目标文件中链接 rts2800.lib。

用户也可以创建自己的目标库并链接它们。链接器只包含和链接未定义的引用库成员。

7.17.3 设置堆栈和堆段的大小

C 使用了两个名为 `.sysmem` 和 `.stack` 的未初始化段分别作为用于 `malloc()` 函数的存储空间（堆）和运行堆栈。可以使用 `-heap` 或 `-stack` 选项来设置它们的大小，在选项的后面直接跟一个 4 字节的常量。这两个段的默认大小均为 1K 字。

见 7.4.8 小节定义堆的大小和 7.4.16 小节定义堆栈的大小。

注意：`.stack` 段链接到数据存储器的低 64K，因为 SP 是一个 16 位的寄存器，所以不能寻址超出 64K 字范围的存储器空间。

7.17.4 运行中变量的自动初始化

在运行时自动初始化变量为默认设置。

使用运行时自动初始化变量时，`.cinit` 段与其他的初始化段一起被装入存储器。链接器定义一个称为 `cinit` 的特殊的符号，用它指向存储器中初始化表的开始。当程序开始执行的时候，C 的引导程序从表（`.cinit` 指向的表）中复制数据到 `.bss` 段指定的变量中。允许初始化的数据存储在 ROM 中，在每一次程序开始执行的时候把初始化数据复制到 RAM 中。

图 7.7 说明了在运行时自动初始化变量。

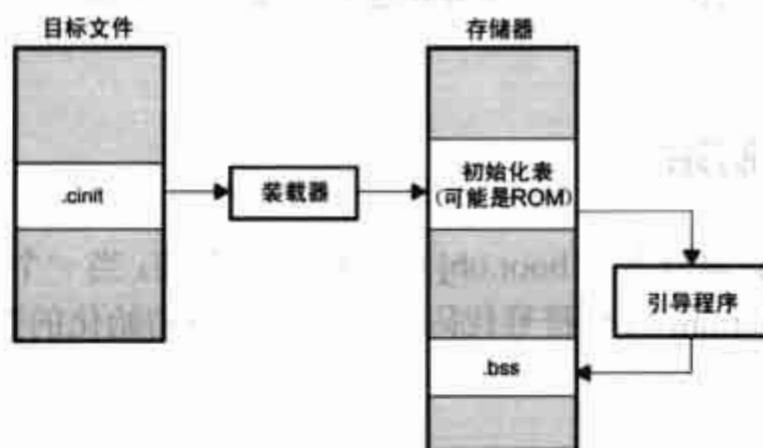


图 7.7 运行时的自动初始化

7.17.5 装载时变量的自动初始化

在装载时初始化变量，能够通过减小导入时间和节省初始化表使用的存储器达到提高性能的目的。要使用这种方法，就要使用 `-cr` 选项来调用链接器。

当使用链接器的 `-cr` 选项的时候，链接器在 `.cinit` 段的头部设定 `STYP_COPY` 位。告诉装载器不要把 `.cinit` 段装入存储器（`.cinit` 段在存储器映像图中不占空间）。链接器也把符号 `cinit` 设定为 `-1`（而通常情况下 `cinit` 指向初始化表的开始）。这个设置告诉引导程序初始化表不出现在存储器中，在导入时不执行初始化。

对于在装载时使用了自动初始化，装载器（它不包含在编译器包中）必须能够完成以

下的任务：

- 检测目标文件中.cinit 段的存在。
- 检测在.cinit 段的标题处设置了 STYP_COPY, 这样就可以知道不用把.cinit 复制到存储器中。
- 推测初始化表的格式。

图 7.8 说明了在装载时初始化变量。

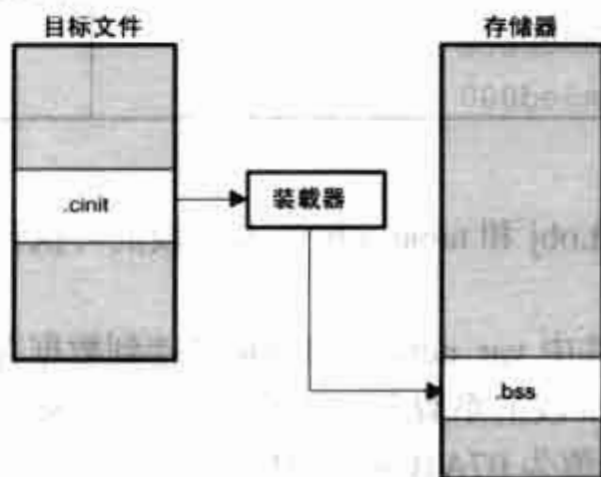


图 7.8 在装载时自动初始化

7.17.6 -c 和-cr 链接器选项

下面列出了当用-c 或-cr 选项调用链接器时所发生的主要情况：

- 符号_c_int00 定义为程序的入口。符号_c_int00 是 C 在 boot.obj 中的引导程序的起点。引用_c_int00 可确保 boot.obj 从运行支持库 rts.lib 中自动链接。
- 用一个终止记录填补.cinit 输出段，这个终止记录指定引导程序（如果在运行时自动初始化）或指定装载器（如果在装载时自动初始化）什么时候停止读入初始化表。
- 装载时初始化的时候（-cr 选项）：
 - 链接器把 cinit 设定为-1。这表明初始化表不在存储器中，所以在运行时不执行初始化。
 - 在 cinit 段的标题处设定 STYP_COPY 标志（0010h）。STYP_COPY 有特殊的属性，它告诉装载器直接执行自动初始化，不把.cinit 段装入存储器。链接器不在存储器中为.cinit 段分配空间。

注意：导入装载器不是 C/C++编译器工具的一部分。可用 TMS320C28x Code Composer Studio 作为装载器。

7.18 链接器举例

例：链接 demo.obj、ctrl.obj 和 tables.obj3 个目标文件合并生成一个名为 demo.out 的

程序。

假定目标存储器有如下配置：

存储器类型	地址范围	内 容
Program	0xf0000 to 0x3fffbf	ROM
	0x3fffc0 to 0x3fffff	中断向量表
Data	0x000040 to 0x0001ff	堆栈
	0x000200 to 0x0007ff	RAM_1
	0x3ed000 to 0x3effff	RAM_2

按以下方式构造输出段：

- 包含在 demo.obj、fft.obj 和 tables.obj 中.text 段的可执行代码链接到程序存储器的 ROM 中。
- 包含在 demo.obj 文件中 var_defs 段的变量链接到数据存储器的块 RAM_2 中。
- 文件 demo.obj 中.data 段的系数表、tables.obj 和 fft.obj 链接到 RAM_1。创建了一个长度为 100，填充值为 07A1Ch 的空位。
- 来自 demo.obj 且包含了缓冲器和变量的 xy 段以默认方式链接到 STACK 块的 Page1。

例 7.13 为该例的链接器命令文件。例 7.14 为映像文件。

例 7.13 链接器命令文件 demo.cmd

```

/*****
/****          指定链接器选项          ****/
/*****
-o demo.out          /* 命名输出文件          */
-m demo.map          /* 创建输出映射          */
/*****
/****          指定输入文件          ****/
/*****
demo.obj
fft.obj
tables.obj
/*****
/****          指定存储器位置          ****/
/*****
MEMORY
{
    PAGE 0:      ROM      (R):  origin=3f0000h length=0ffc0h
                  VECTORS (R):  origin=3fffc0h length=0040h
    PAGE 1:      STACK   (RW): origin=000040h length=01c0h
                  RAM_1   (RW): origin=000200h length=0600h
                  RAM_2   (RW): origin=3ed000h length=3000h
}
/*****
/****          指定输出段          ****/
/*****

```

```

SECTIONS
{
    vectors : { } > VECTORS page=0
    .text   : load = ROM, page = 0 /* 链接.text 到 ROM */
    .data   : fill = 07A1Ch, Load=RAM_1, page=1
{
    tables.obj(.data)      /* .data 输入 */
    ft.obj(.data)          /* .data 输入 */
    += 100h;               /* 创建空位, 用 07A1Ch 填充 */
}
    var_defs : { } > RAM_2 page=1 /* 在 RAM 中定义 */
    .bss: page=1, fill=0ffffh     /* .bss 的填充和链接 */
}
/*****
***          命令文件结束          ***
*****/

```

通过输入下面的命令调用这个文件:

```
lnk2000 demo.cmd
```

此命令产生的映像文件在例 7.14 中。名为 demo.out 的输出文件可在 TMS320C28x 器件上运行。

例 7.14 输出映像文件 demo.map

```

OUTPUT FILE NAME: <demo.out>
ENTRY POINT SYMBOL: 0
MEMORY CONFIGURATION

```

	name	origin	length	attributes	fill
PAGE 0: ROM		003f0000	0000fffc0	R	
	VECTORS	003ffffc0	00000040	R	
PAGE 1: STACK		00000040	000001c0	RW	
	RAM_1	00000200	00000600	RW	
	RAM_2	003ed000	00003000	RW	

```

SECTION ALLOCATION MAP
output attributes/

```

section	page	origin	length	input sections
vectors		0003ffffc0	00000000	UNINITIALIZED
.text		0003f0000	0000001a	
		003f0000	0000000e	demo.obj (.text)
		003f000e	00000000	tables.obj (.text)
		003fo00e	0000000c	fft.obj (.text)
var_defs 1		003ed000	00000002	
		003ed000	00000002	demo.obj (var_defs)
.data 1		00000200	0000010c	
		00000200	00000004	tables.obj (.data)
		00000204	00000000	fft.obj (.data)
		00000204	00000100	--HOLE-- [fill = 7a1c]
		00000304	00000008	demo.obj (.data)

```

.bss      0      00000040      00000069
           00000040      00000068      demo.obj (.bss) [fill=ffff]
           000000a8      00000000      fft.obj (.bss)
           000000a8      00000001      tables.obj (.bss) [fill=ffff]
xy        1      000000a9      00000014      UNINITIALIZED
           000000a9      00000014      demo.obj (xy)

```

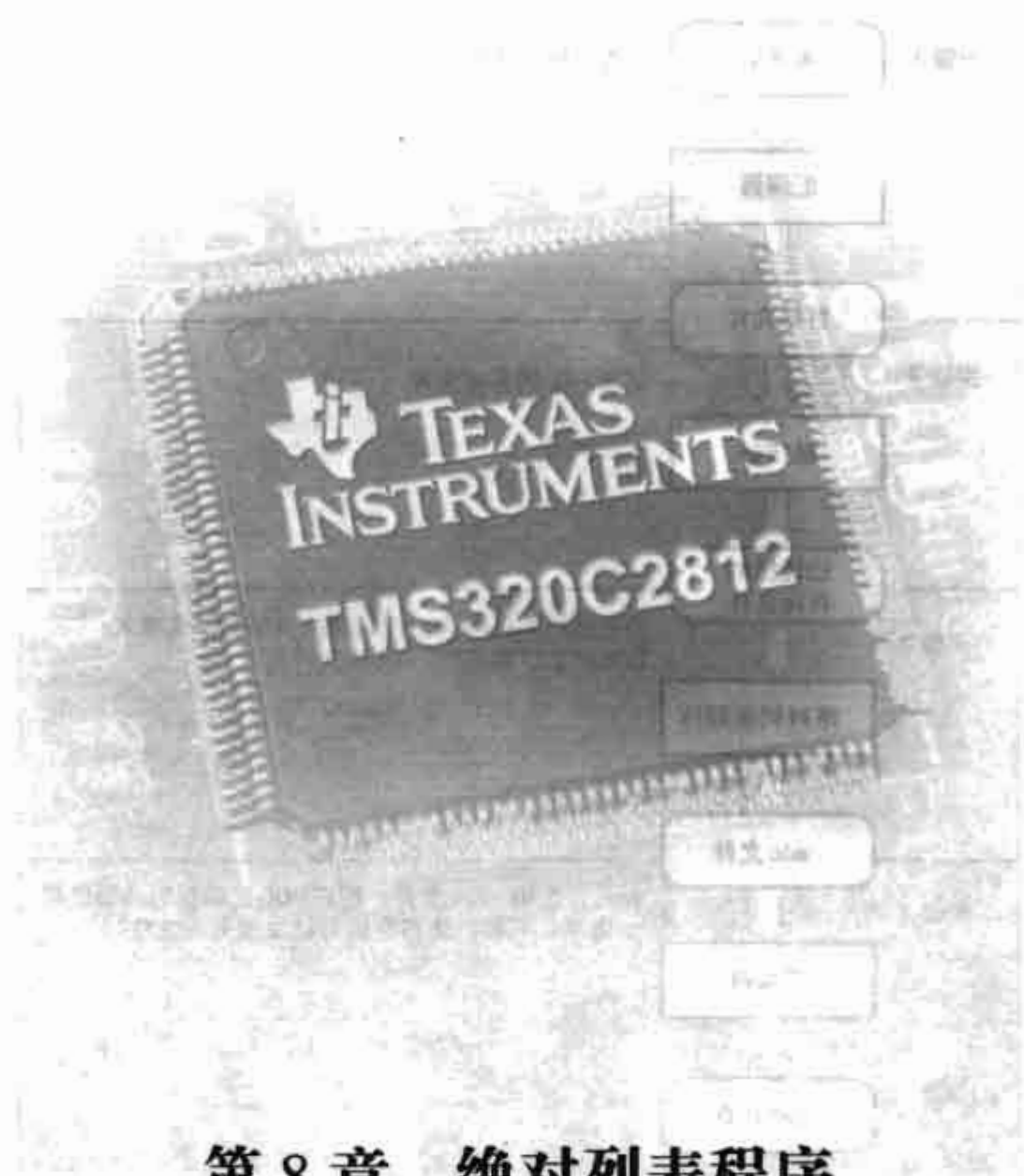
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address	name
-----	----
00000040	.bss
00000200	.data
003f0000	.text
00000040	ARRAY
000000a8	TEMP
00000040	__bss__
00000200	__data__
0000030c	__edata__
000000a9	__end__
003f001a	__etext__
003f0000	__text__
003f000e	_func1
003f0000	_main
0000030c	edata
000000a9	end
003f001a	etext

GLOBAL SYMBOLS: SORTED BY Symbol Address

address	name
-----	----
00000040	ARRAY
00000040	__bss__
00000040	.bss
000000a8	TEMP
000000a9	__end__
000000a9	end
00000200	__data__
00000200	.data
0000030c	edata
0000030c	__edata__
003f0000	_main
003f0000	.text
003f0000	__text__
003f000e	_func1
003f001a	etext
003f001a	__etext__

[16 symbols]



第 8 章 绝对列表程序

TMS320C28x™ 绝对列表程序是一个调试工具，它将已链接的目标文件作为输入并创建.abs 文件作为输出。汇编这些.abs 文件并生成一个列表，列表中显示了目标文件的绝对地址。若用人工操作，这将是一项单调乏味的需要许多操作运算的过程。而绝对列表程序可将这些操作自动完成。

8.1 生成绝对列表

图 8.1 所示为生成一个绝对列表所需要的步骤。

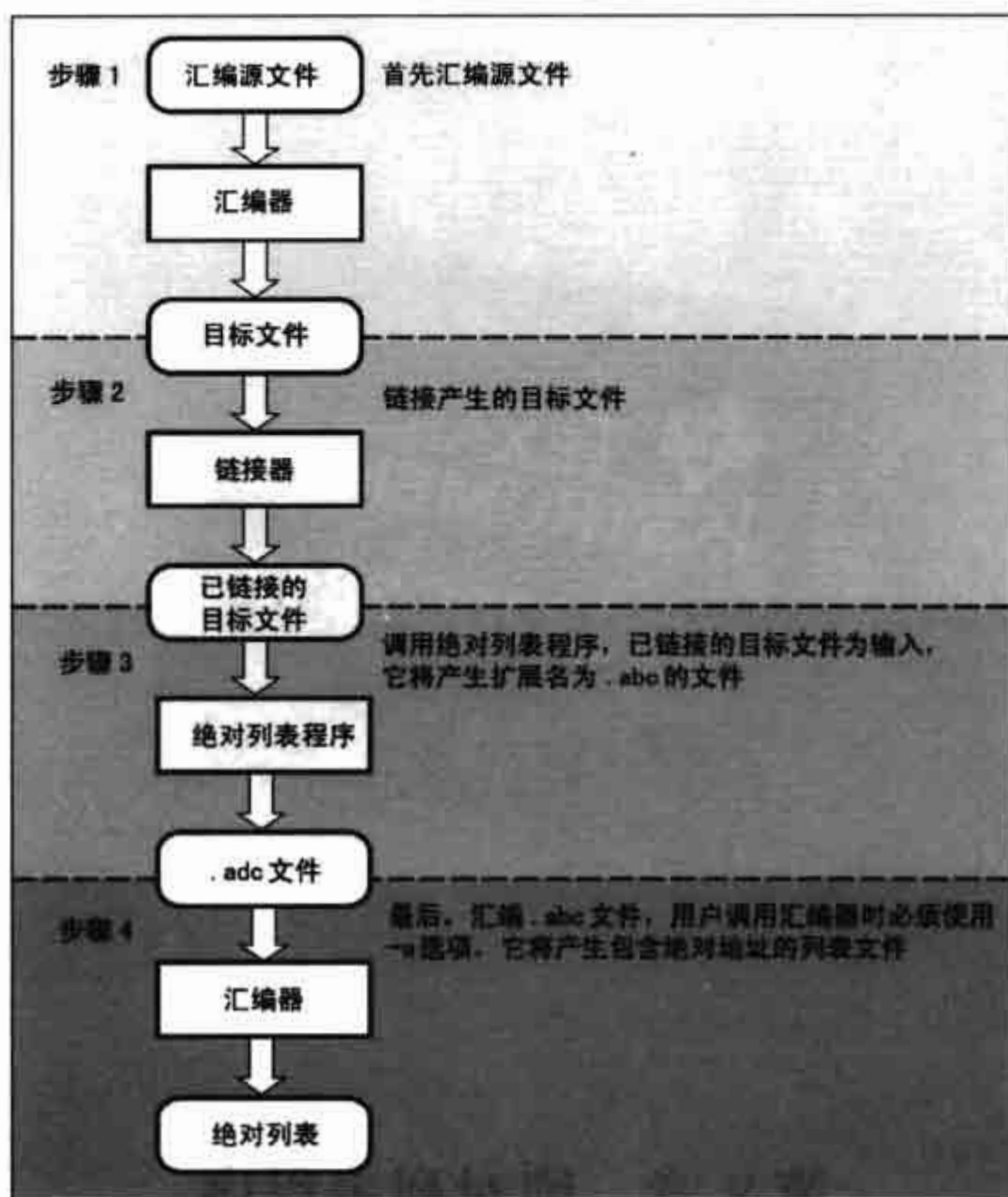


图 8.1 绝对地址列表流程图

8.2 调用绝对列表程序

调用绝对列表程序的指令格式：

abs2000 [-options] input file

abs2000

调用绝对列表程序的命令。

options

表示绝对列表程序的选项，选项对大小写不敏感，它跟在命令的后面，可以出现在命令行的任何地方。每个选项由连字符（-）引导。绝对列表程序的选项如下：

-e 使用户能够改变汇编文件、C文件和C头文件扩展名的命名约定。几个选项如下所示：

-ea[.]asmext 汇编文件（默认设置是.asm）

-ec [.]cext C文件（默认设置是.c）

-eh [.]hext C头文件（默认设置是.h）

-ep [.]cppext C++文件（默认设置是.cpp、.cc和.cxx）

扩展名中的“.”以及选项与扩展名之间的空格是可选的。

-fs 为输出文件指定目录。要把由绝对列表程序生成的.abs文件放置在C:\ABSDIR中，使用命令abs2000 -fs C:\ABSDIR filename.out。如果没有指定-fs选项，绝对列表程序在当前目录下生成.abs文件。

-q (quiet) 禁止产生标题和所有进程信息。

input file 输入链接目标文件名。如果不提供扩展名，绝对列表程序就假设有一个默认设置的扩展名.out。在调用输入文件时若不提供文件名，绝对列表程序将提示用户输入一个文件名。

绝对列表程序为每个链接文件生成一个输出文件。这些文件用输入文件名和扩展名.abs命名。然而头文件不生成相应的.abs文件。

用-a 汇编器选项汇编这些文件并创建如下的绝对列表：

```
asm2000 -v28 -a filename.abs
```

-e选项影响命令行文件名和输出文件名。它总是在命令行的文件名之前。

当已链接的目标文件是由带调试选项编译的C文件创建时，-e选项很有用。当设置了调试选项时，链接目标文件的结果中包含曾经创建它的源文件的文件名。在这种情况下，绝对列表程序不为C头文件生成相应的.abs文件。另外，对应于C源文件的.abs文件将使用由C源文件生成的汇编文件而不是C源文件本身。

例如，假设C源文件hello.csr带调试设置选项被编译，这将生成一个汇编文件hello.s。这个汇编文件包括hello.hsr。假设所创建的可执行文件为hello.out，那么下面命令生成的就是.abs文件。

```
abs2000 -ea s -ec csr -eh hsr hello.out
```

不会为hello.hsr文件（头文件）创建.abs文件，而且hello.abs包括汇编文件hello.s，而不包括C源文件hello.csr。

8.3 绝对列表程序实例

本例使用了3个源文件。文件module1.asm和module2.asm都包括文件globals.def。

```
module1.asm
.text
.bss array,100
.bss dflag, 2
.copy globals.def
MOV ACC, #offset
MOV ACC, #dflag
```

```

module2.asm
    .bss offset, 2
    .copy globals.def
    MOV ACC, #offset
    MOV ACC, #array
    globals.def
    .global dflag
    .global array
    .global offset

```

下面为文件 module1.asm 和 module2.asm 创建绝对列表：

第一步：首先汇编 module1.asm 和 module2.asm。

```

asm2000 -v28 module1
asm2000 -v28 module2

```

按命令顺序创建两个目标文件 module1.obj 和 module2.obj。

第二步：用下面的 bttest.cmd 链接器命令文件链接 module1.obj 和 module2.obj。

```

/*****
/* 文件 bttest.cmd 链接 TMS320C28x 模块的 COFF 链接器 */
/* 命令文件 */
/*****
-o bttest.out          /* 命名输出文件 */
-m bttest.map          /* 创建输出影射 */
/*****
/* 指定输入文件 */
/*****
module1.obj
module2.obj
/*****
/* 指定存储器配置 */
/*****
MEMORY
{
    PAGE 0: ROM: origin=2000h length=2000h
    PAGE 1: RAM: origin=8000h length=8000h
}
/*****
/* 指定输出段 */
/*****
SECTIONS
{
    .data: >RAM
    .text: >ROM
    .bss: >RAM
}

```

调用链接器：

```
lnk2000 bttest.cmd
```

这个命令创建一个 `bttest.out` 可执行文件，绝对列表程序使用这个新文件作为其输入。
 第三步：调用绝对列表程序。

```
abs2000 bttest.out
```

这个命令创建了两个文件 `module1.abs` 和 `module2.abs`。

```
module1.abs:
    .nolist
array  .setsym      000008000h
dflag  .setsym      000008064h
offset .setsym      000008066h
.data  .setsym      000008000h
edata  .setsym      000008000h
.text  .setsym      000002000h
etext  .setsym      000002008h
.bss   .setsym      000008000h
end     .setsym      000008068h
        .setsect    ".text",000002000h
        .setsect    ".data",000008000h
        .setsect    ".pst",000008000h
        .list
        .text
        .copy       "module1.ism"

module2.bass:
    .molest
array  .setsym      000008000h
FDA    .setsym      000008064h
offset .setsym      000008066h
.data  .setsym      000008000h
edata  .setsym      000008000h
.text  .setsym      000002000h
etext  .setsym      000002008h
.bss   .setsym      000008000h
end     .setsym      000008068h
        .setsect    ".text",000002004h
        .setsect    ".data",000008000h
        .setsect    ".bss",000008066h
        .list
        .text
        .copy       "module2.asm"
```

当在第四步调用它时，这些文件包含如下信息：

- 它们包含 `.setsym` 伪指令，这个伪指令给全局符号赋值。两个文件都包括了为符号 `dflag` 赋值的全局等式。在文件 `globals.def` 中定义符号 `dflag`，文件 `globals.def` 包含在 `module1.asm` 和 `module2.asm` 中。
- 它们包含的 `.setsym` 伪指令告诉汇编器包含哪个汇编语言源文件。

第四步：汇编由绝对列表程序创建的 `.abs` 文件（记住，当调用汇编器时，必须用 `-e`

选项)。

```
asm2000 -v28 -a module1.abs
asm2000 -v28 -a module2.abs
```

按这个命令顺序创建两个列表文件 module1.lst 和 module2.lst，不产生目标码。这些列表文件与常规的列表文件相似，但是所示的地址是绝对地址。

所创建的绝对列表文件 module1.lst（见图 8.2）和 module2.lst（见图 8.3）。

```

module1.abs PAGE 1
15 002000      .text
16              .copy      "module1.asm"
1  002000      .text
2  008000      .bss    array,100
3  008064      .bss    dflag,2
4              .copy    globals.def
1              .global dflag
2              .global array
2              .global array
3              .global offset
5  002000 FF20!  MOV     ACC,#offset
      002001 8066
6  002002 FF20-  MOV     ACC,#dflag
      002003 8064
```

图 8.2 module1.lst

```

module1.abs PAGE 1
15 002004      .text
16              .copy    "module2.asm"
1  008006      .bss    offset,2
2              .copy    globals.def
1              .global dflag
2              .global array
3              .global offset
3  002004 FF20-  MOV     ACC,#offset
      002005 8066
4  002006 FF20!  MOV     ACC,#array
      002007 8000
```

图 8.3 module2.lst



第 9 章 交叉引用列表程序

TMS320C28x™ 交叉引用列表程序是一个调试工具。它接受已链接的目标文件作为输入，生成一个交叉引用列表文件作为输出。表中显示了在所链接的源程序中符号、符号定义和符号的引用。

9.1 生成交叉引用列表

如图 9.1 所示，为生成一个交叉引用列表所要求的步骤。

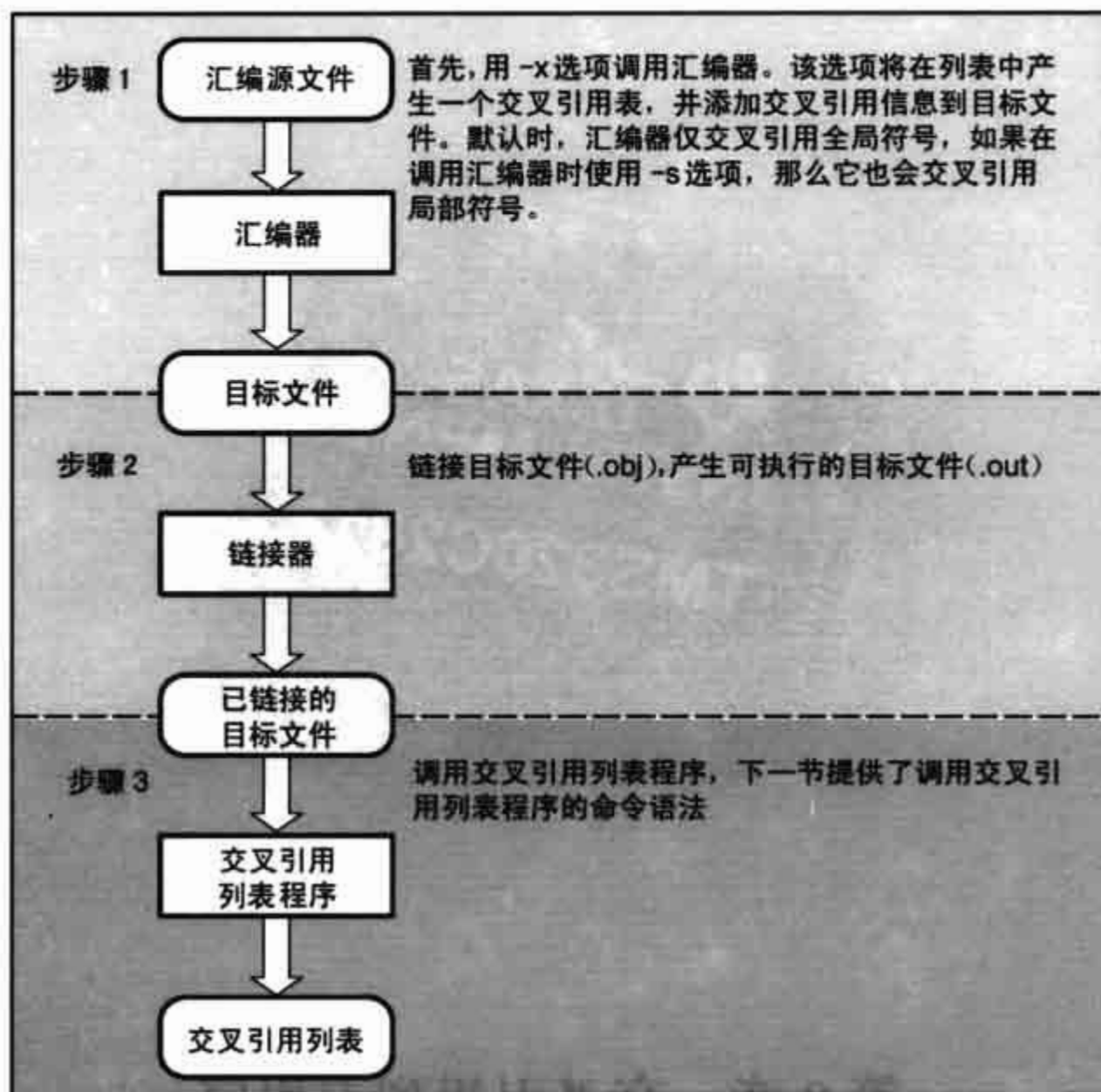


图 9.1 交叉引用列表流程图

9.2 调用交叉引用列表程序

要使用交叉引用列表程序, 就必须用正确的选项汇编文件, 然后把它们链接到一个执行文件中。用 `-x` 选项对汇编语言文件进行汇编。此选项创建一个交叉引用列表并附有对目标文件的交叉引用信息。在默认设置情况下, 汇编器仅交叉引用全局符号。但如果用 `-s` 选项调用汇编器, 也含局部符号, 链接目标文件以获得执行文件。

使用如下格式调用交叉引用列表程序:

```
xref2000 [options] [input filename [output filename] ]
```

xref2000

调用交叉引用列表程序的命令。

options

交叉引用列表程序的选项, 选项对大小写不敏感, 它跟在命令的后面, 可以出现在命令行的任何地方。每个选项由连字号 (-) 引导。

交叉引用列表程序的选项如下：

- l** (小写L) 规定输出文件每页的行数。**-l**选项的格式是**-lnum**，其中num是十进制常数。例如，**-l30**设置输出文件每页为30行。选项与十进制常数之间的空格是可选的。默认设置为每页为60行。
- q** 禁止标题和所有过程信息。
- input filename** 已链接的目标文件。如果省略了输入文件，程序会提示用户需要一个文件名。
- output filename** 交叉引用列表文件的名字。如果省略了输出文件，默认文件名就是输入文件名加一个.xrf扩展名。

9.3 交叉引用列表程序举例

```
=====
Symbol: _SETUP
Filename      RTYP   AsmVal   LnkVal   DefLn   RefLn   RefLn   RefLn
-----
demo.asm  EDEF  '00000018  00000018   18      13      20
=====
Symbol: _fill_tab
Filename      RTYP   AsmVal   LnkVal   DefLn   RefLn   RefLn   RefLn
-----
ctrl.asm  EDEF  '00000000  00000040   10       5
=====
Symbol: _x42
Filename      RTYP   AsmVal   LnkVal   DefLn   RefLn   RefLn   RefLn
-----
demo.asm  EDEF  '00000000  00000000    7       4      18
=====
Symbol: gvar
Filename      RTYP   AsmVal   LnkVal   DefLn   RefLn   RefLn   RefLn
-----
tables.asm EDEF  "00000000  08000000   11      10
=====
```

如下是在上述交叉引用列表中定义的项：

- Symbol** 所列符号的名字。
- Filename** 出现符号的文件名。
- RTYP** 在此文件中符号引用的类型。引用类型有：
- STAT** 在此文件中定义的符号但未声明为全局符号。
 - EDEF** 在此文件中定义的符号并声明为全局符号。
 - EREF** 未在此文件中定义的符号但作为全局符号访问。
 - UNDF** 未在此文件中定义的符号且未声明为全局符号。
- AsmVal** 此十六进制的数字是汇编时赋予符号的数值。数值也可以有一个用以说

明符号属性的前导字符。表9.1列出了这些字符和名字。

LnkVal 此十六进制数字是链接后赋予符号的数值。

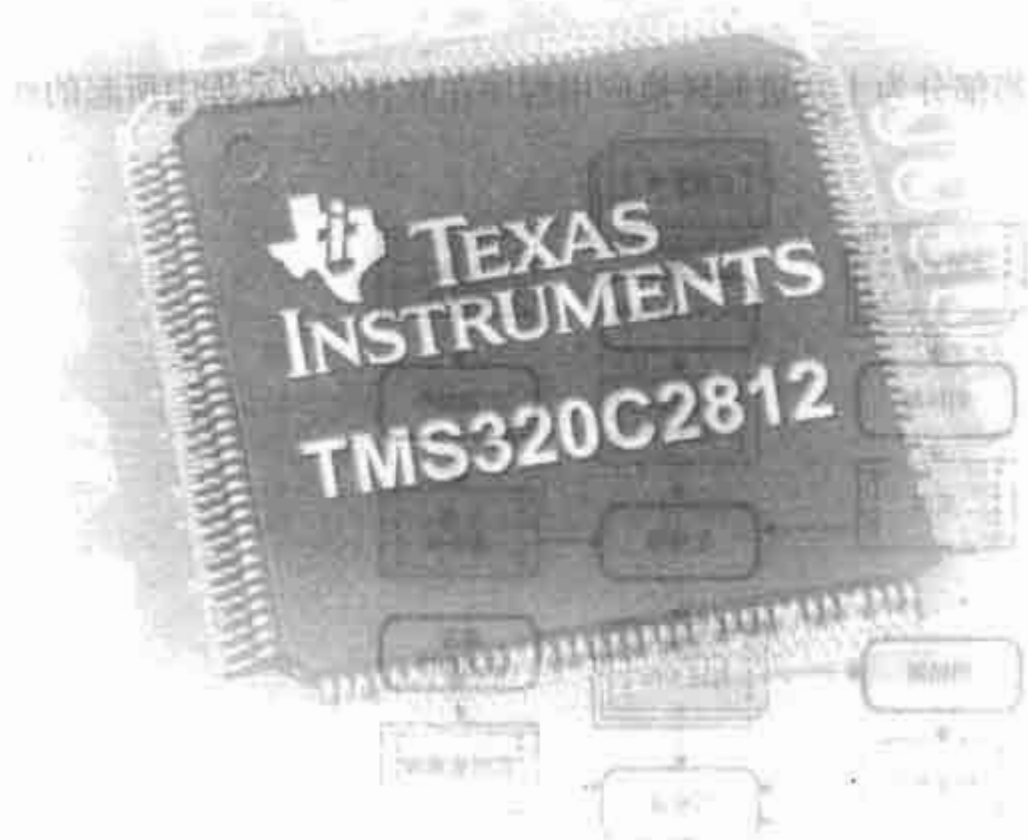
DefLn 符号定义处的语句号。

RefLn 引用符号处的行号。如果此行号后跟着一个星号（*），那么这个引用可以修改目标的内容。空列表示未用的符号。

表 9.1 符号属性

字 符	意 义
,	定义在.text 段中的符号
"	定义在.data 段中的符号
+	定义在.sect 段中的符号
-	定义在.bss 或.usect 段中的符号





第 10 章 十六进制转换应用程序

TMS320C28x™ 汇编器和链接器创建通用目标文件格式 (COFF) 的目标文件。COFF 是一个二进制目标文件，它鼓励模块化编程并提供了强大而灵活的管理代码段和目标系统存储器的方法。

大多数 EPROM 编程器不接受 COFF 目标文件作为输入。为了将目标文件装载到一个 EPROM 编程器中，十六进制转换应用程序将 COFF 目标文件转换成几种标准 ASCII 十六进制格式之一。十六进制转换应用程序也使用在 COFF 目标文件需要进行十六进制转换的其他应用中（例如调试器和装载器的应用）。

十六进制转换应用程序能够生成如下输出格式的文件：

- ☐ ASCII-Hex, 支持 16bit 地址
- ☐ 扩展的 Tektronix (Tektronix)
- ☐ Intel MCS-86 (Intel)
- ☐ Motorola Exorciser (Motorola-S), 支持 16bit 地址
- ☐ Texas Instruments SDSMAC (TI-Tagged), 支持 16bit 地址

10.1 十六进制转换应用程序在软件开发流程中的作用

图 10.1 中阴影部分为十六进制转换应用程序在软件开发过程中所起的作用。

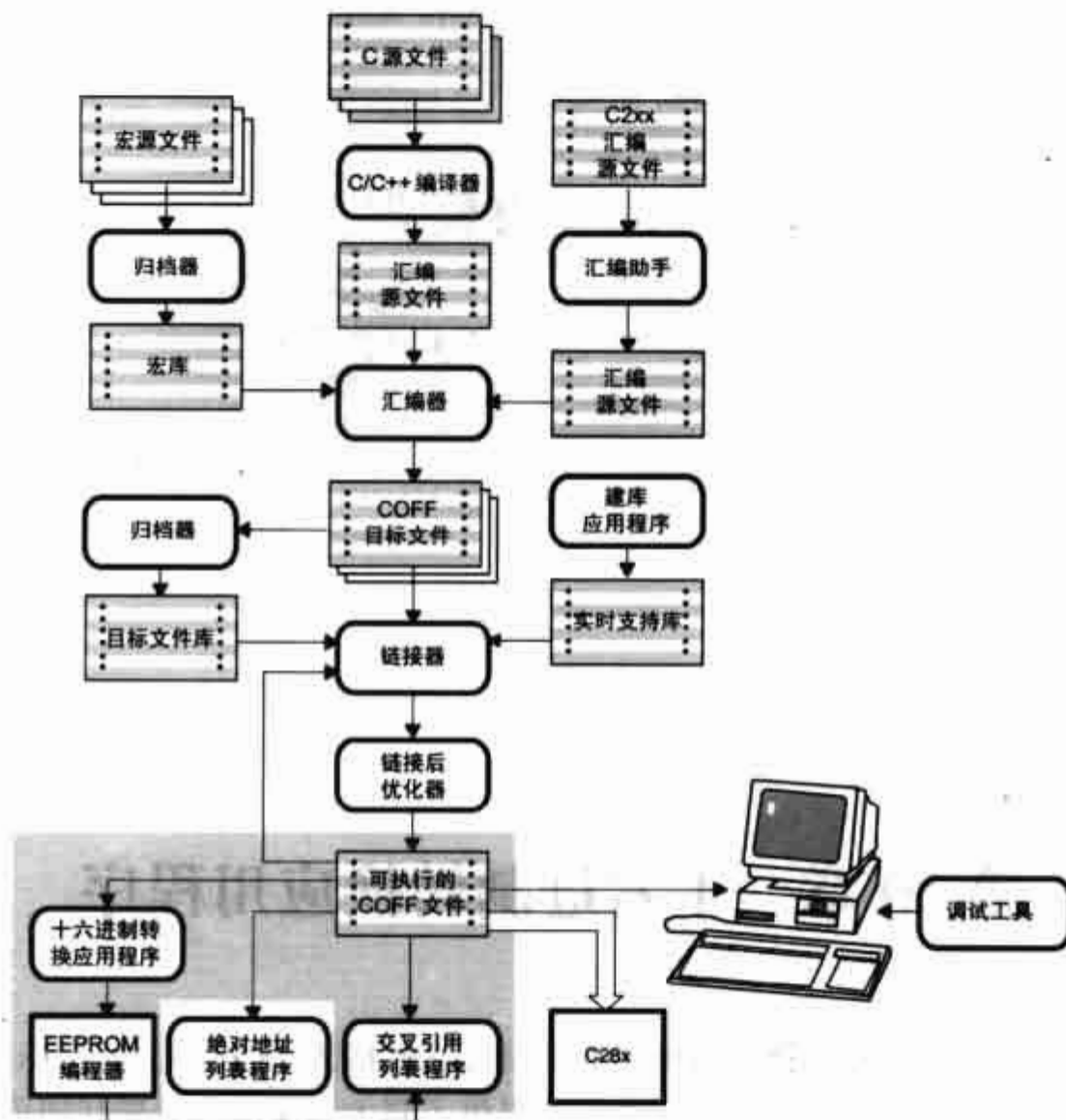


图 10.1 十六进制转换应用程序在 TMS320C28x™ 软件开发中的流程图

10.2 调用十六进制转换应用程序

调用十六进制转换应用程序有两种基本方法：

- 在命令行指定选项和文件名。例如将文件 `firmware.out` 转换为 TI-Tagged 格式，并生成两个输出文件 `firm.lsb` 和 `firm.msb`。

```
hex2000 -t firmware -o firm.lsb -o firm.msb
```

- 在命令文件中指定选项和文件名。为了调用十六进制转换应用程序，用户可以创

建存储命令行选项和文件名的批处理文件。例如用命令文件 hexutil.cmd 调用应用程序。

```
hex2000 hexutil.cmd
```

在命令行中可以用十六进制转换应用程序的 ROMS 和 SECTIONS 伪指令附加常规的命令行信息。

10.2.1 从命令行调用十六进制转换应用程序

调用十六进制转换应用程序，输入：

```
Hex2000 [options] filename
```

hex2000 调用十六进制转换应用程序的命令。

options 提供十六进制转换过程的附加信息。可以在命令行或命令文件中使用选项。

- 所有选项由连字符（-）引导，对大小写不敏感。
- 几个可选项具有附加参数，这些参数必须用至少一个空格与可选项分开。
- 具有多字符名的可选项必须用全称，不允许使用缩写。
- 除了 -q 选项（静态）必须用在其他选项之前而外，可选项不受顺序的影响。

filename 给 COFF 目标文件和命令文件命名，详细情况见 10.2.2 小节用命令文件调用十六进制转换应用程序。

表 10.1 基本选项

通用选项	选 项	描 述
控制十六进制转换 应用程序的总体操作	-map filename	生成映像文件
	-o filename	指定输出文件名
	-q	静态运行（使用时，它必须出现在其它选项之前）
映像选项	选 项	描 述
在目标存储器内 创建连续的映像图。	-fill value	用 value 值填充空位
	-image	指定映像模式
	-zero	在映像模式中重新设置起始地址为 0
存储器选项	选 项	描 述
为输出文件	-memwidth value	定义系统存储器字宽（默认值 32 位）
配置存储器宽度	-romwidth value	指定 ROM 的宽度（默认值与使用格式有关）
输出格式	选 项	描 述
指定输出格式	-a	选择 ASCII-Hex
	-i	选择 Intel
	-m	选择 Motorola-S
	-t	选择 TI-Tagged
	-x	选择 Tektronix

10.2.2 用命令文件调用十六进制转换应用程序

命令文件可以用相同的输入文件和选项多次调用应用程序。当用户使用 ROMS 和 SECTIONS 伪指令来进行十六进制的转换处理时，命令文件也很有用。

命令文件是包含下列各项中的一项或多项的ASCII文件：

- 可选项和文件名。它们在命令文件中指定，指定方式与在命令行中的相同。
- ROMS 伪指令。它定义系统的物理存储器的配置，如一个地址区域表（详情请见第 10.4 节 ROMS 伪指令）。
- SECTIONS 伪指令。SECTIONS 伪指令指定选择来自 COFF 目标文件的哪一个段。
- 注释。用分隔符/*和*/在命令文件中加注释。如：

```
/* 这是注释 */
```

调用应用程序和使用在命令文件中定义的选项，输入：

```
hex2000 command_filename
```

选项和文件名出现的顺序并不重要。在转换之前，应用程序从命令行中读取所有的输入、从命令文件中读取所有的信息。如果要使用-q 选项，-q 必须是命令行和命令文件中的第一个选项。

-q 选项隐藏十六进制转换应用程序的输出信息标和过程信息的显示。

假设，命令文件 firmware.cmd 包含如下几行：

```
firmware.out      /* 输入文件 */
-t               /* TI-Tagged */
-o firm.lsb      /* 输出文件 */
-o firm.msb      /* 输出文件 */
```

调用十六进制转换应用程序，输入：

```
hex2000 firmware.cmd
```

下面显示如何将 appl.out 文件转换成 4 个 Intel 格式的十六进制文件。每个输出文件宽为 1 字节，长为 4K 字节。

```
appl.out          /* 输入文件 */
-i               /* Intel 格式 */
-map appl.mxp     /* 映像文件 */
ROMS
{
    ROW1:  origin=0x00000000 len=0x4000 romwidth=8
           files={ appl.u0 appl.u1 }
    ROW2:  origin=0x00004000 len=0x4000 romwidth=8
           files={ appl.u2 appl.u3 }
}
SECTIONS
{
    .text, .data, .cinit, .sect1, .vectors, .const:
}
```

10.3 存储器宽度

用十六进制转换应用程序允许用户指定存储器和 ROMS 的宽度，使得存储器结构更加灵活。要使用十六进制转换应用程序，必须了解应用程序如何处理字的宽度。在转换过程中有 3 个重要的宽度：

- 目标宽度
- 存储器宽度
- ROM 宽度

图 10.2 所示为十六进制转换应用程序处理流程的两个独立的、截然不同的阶段。

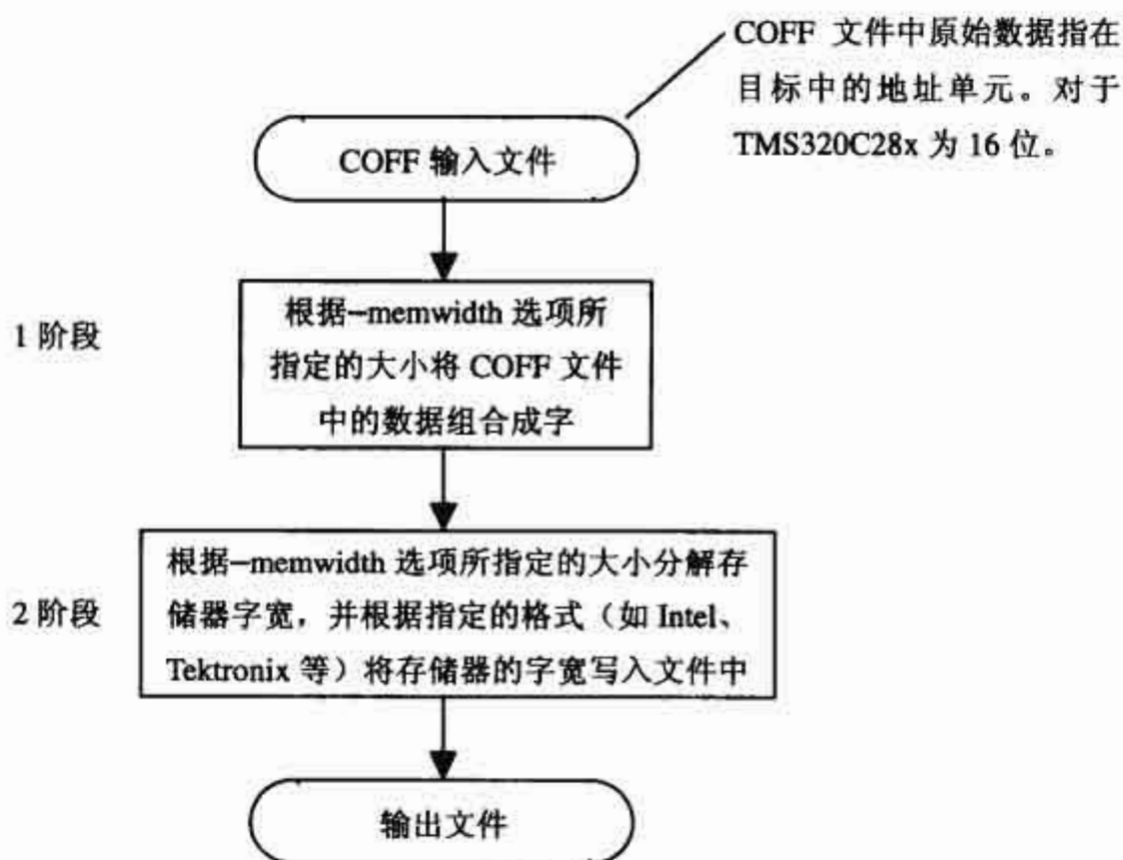


图 10.2 十六进制转换应用程序处理流程图

10.3.1 目标宽度

目标宽度是目标处理器字的位大小。它对应目标处理器数据总线的宽度，对于每一个目标处理器，宽度是固定不变的。TMS320C28x 目标宽度为 32 位。

10.3.2 存储器宽度

存储器宽度是存储器系统的物理宽度。通常情况下，存储器系统的宽度实际上与目标处理器的宽度相同：32 位处理器具有 32 位存储器结构。然而，有些应用要求将目标字分解成窄的多个连续的存储器字。

十六进制转换应用程序默认存储器宽度为目标宽度（在 TMS320C28x 下为 32 位）。可以通过如下方法改变存储器宽度：

- 使用 `-memwidth` 选项。这将改变整个文件的存储器宽度。
- 设置 ROMS 伪指令的 `memwidth` 参数。这将改变 ROMS 伪指令指定的地址范围内存储器的宽度并覆盖这个地址范围内的 `-memwidth` 选项。参见第 10.4 节 ROMS 伪指令。

上述两种方法，所用值都大于等于 8 和 8 的倍数。

仅仅只有在将一个目标字分解成窄的连续的存储器字时，才改变存储器为 16 位的默认宽度。图 10.3 说明存储器宽度与 COFF 数据的关系。

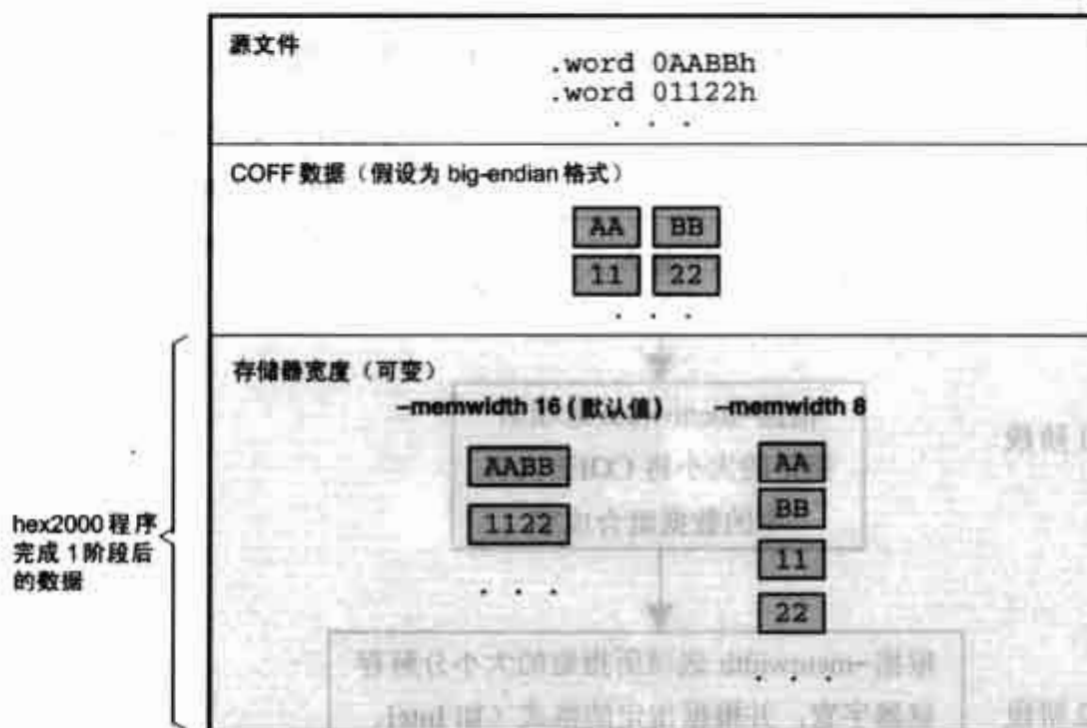


图 10.3 COFF 数据与存储器宽度

10.3.3 数据划分到输出文件

ROM 宽度指的是每个 ROM 器件和对应输出文件（一般是一个字节或 8 位）的物理宽度（位宽）。ROM 的宽度决定十六进制转换应用程序怎样把数据划分到输出文件中。COFF 数据映射到存储器字之后，存储器字被分到一个或多个输出文件。输出文件个数由下述条件决定：

- 如果存储器宽度 \geq ROM 宽度：文件数 = 存储器宽度 \div ROM 宽度
- 如果存储器宽度 $<$ ROM 宽度：文件数 = 1

例如，对于一个宽度为 16 位的存储器，用户可以指定一个 ROM 宽度值为 16 位并设置一个包含 16 位字的输出文件。或者使用一个 8 位的 ROM 宽度值将得到两个文件，每个文件的每个字节为 8 位。

十六进制转换应用程序使用的 ROM 宽度的默认值与输出格式有关：

- 除了 TI-Tagged，所有十六进制格式被配置为 8 位字节的列表；这些格式的 ROM 默认宽度为 8 位。

- TI-Tagged 是 16 位的格式；其 TI-Tagged 的 ROM 默认宽度为 16 位。

注意：TI-Tagged 格式是 16 位宽。不能改变 TI-Tagged 格式的宽度，TI-Tagged 格式仅支持 16 位 ROM 宽度。

除了 TI-Tagged，可以通过以下方法改变 ROM 的宽度：

- 使用 `-romwidth` 选项，此选项改变所有 COFF 文件的 ROM 宽度值。
- 设置 ROMS 伪指令的 `romwidth` 参数，对于指定的 ROM 地址范围内，此参数将改变 ROM 宽度值而且覆盖此地址区域原来的 `-romwidth` 选项。参见第 10.4 节 ROM 伪指令。

上述两种方法，所用值都是大于等于 8 和 8 的倍数。

如果所选择的宽度比输出格式的自然大小还要宽（TI-Tagged 为 16 位，其余为 8 位），应用程序就以多字节的形式写入文件。

图 10.4 说明了 COFF 数据、存储器和 ROM 宽度的相互关系。

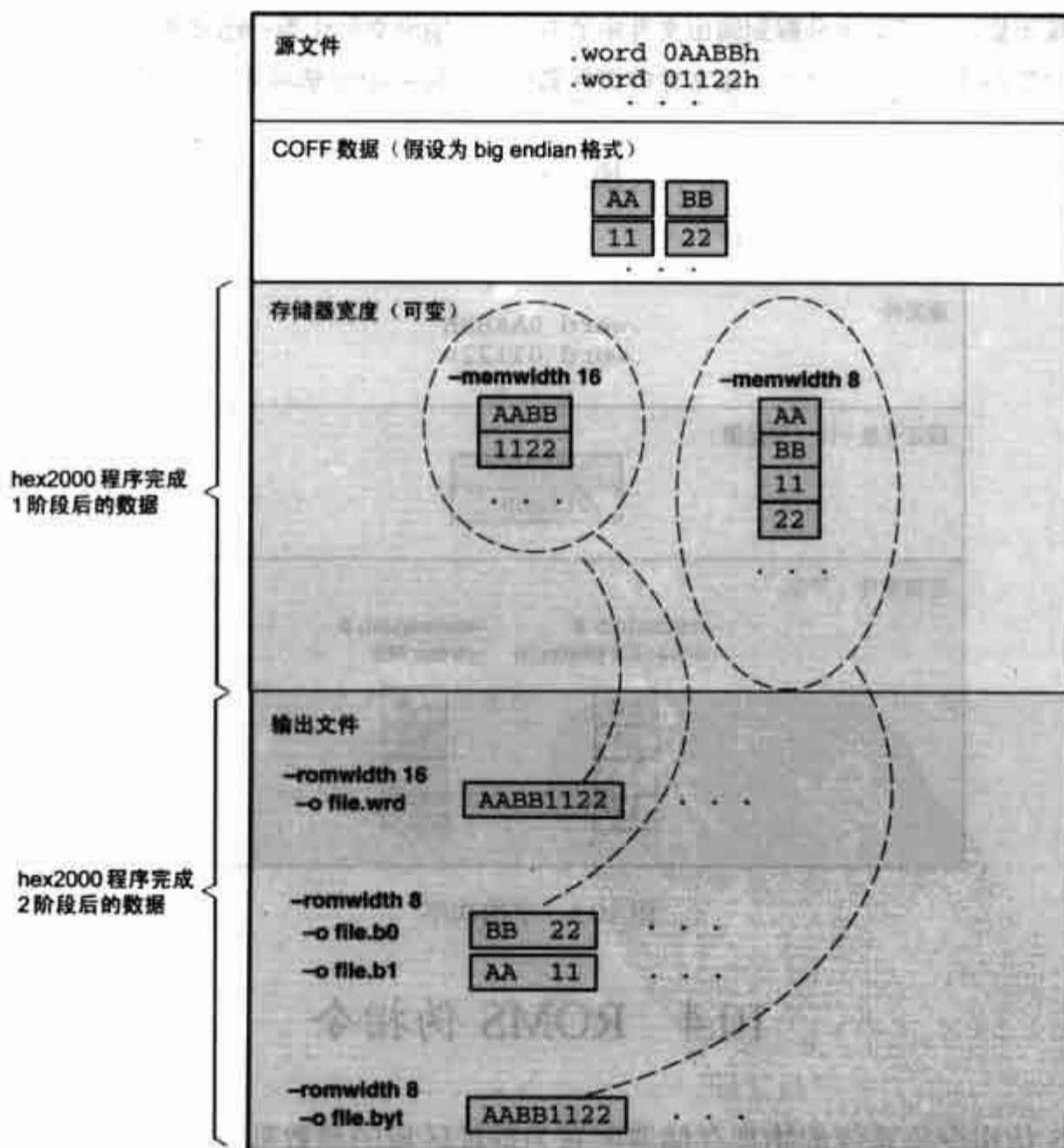
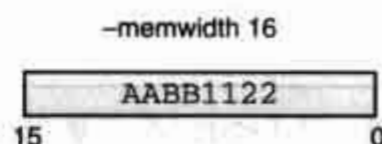


图 10.4 数据、存储器和 ROM 宽度

对于组成 COFF 数据仅仅改变了存储器宽度和 ROM 宽度。因此在整个转换过程中 COFF 数据的字节次序未变。一个存储器字内位的分布，总是从右至左的排列，如右图。



10.3.4 规定输出字的字次序

当存储器字比目标字窄（存储器宽度<16）时，目标字被分解成多个连续的存储器字。有两种方法可以将宽字分解到输出文件的连续存储单元内。

- `-order MS` 指定 big-endian 次序，其中，宽字的最高部分占据连续单元的起始位。
- `-order LS` 指定 little-endian 次序，其中，宽字的最低部分占据连续单元的起始位。

默认方式，应用程序采用 little-endian 格式，因为这种次序是 TMS320C28x 引导装载器希望的数据格式。除非用自己的引导装载程序，否则应避免使用 `-order MS`。

注意：当使用 `-order` 选项时

此选项仅在用户使用小于 16 的存储器宽度时应用，否则，忽略 `-order` 选项。

此选项不影响存储器字分解到输出文件中的方式。把输出文件作为一组考虑：该组包含一个最低字和一个最高字。当用户列出一组文件的文件名时，不管 `-order` 选项如何，总是首先排列最低字。

图 10.5 说明了 `-order` 是如何影响转换过程。此图与前面的图 10.4 解释了十六进制转换应用程序输出文件中数据的状态。

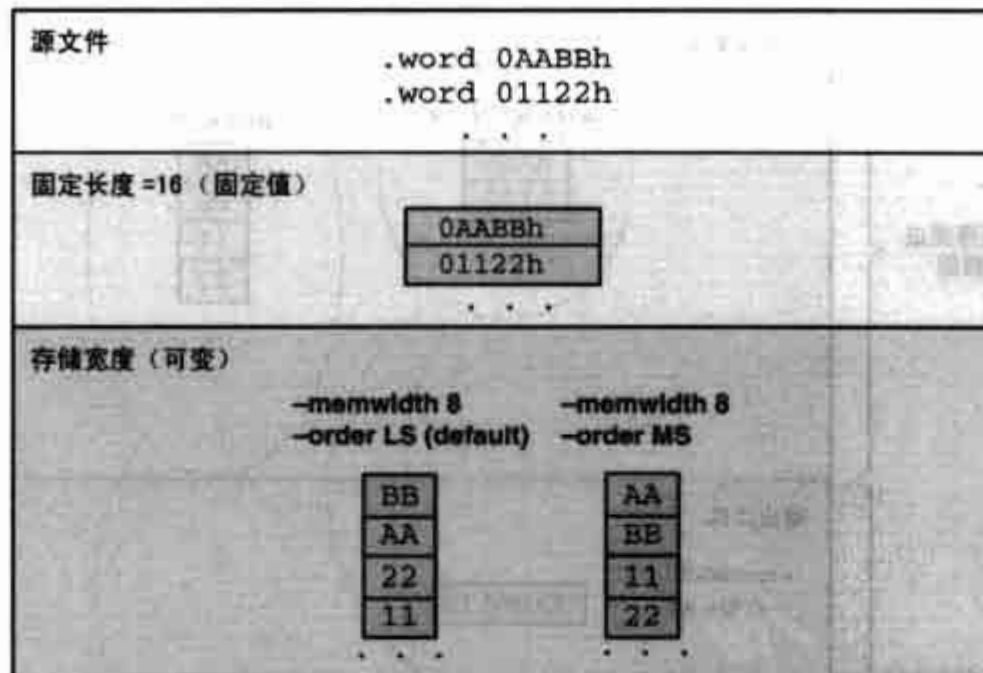


图 10.5 字的次序

10.4 ROMS 伪指令

ROMS 伪指令将系统的物理存储器配置为地址区域的参数列表。

每个地址区域生成一组包含十六进制转换应用程序输出数据的文件，文件与地址区域

相对应。每个文件能够用于单个 ROM 程序存储器。

ROMS 伪指令与 TMS320C28x 链接器的 MEMORY 伪指令相似：两个都定义目标地址空间的存储器映像。ROMS 伪指令的每一行定义一个特定的地址区域。一般格式：

```

ROMS
{
  romname:  [origin=value,] [length=value,] [romwidth=value,]
            [memwidth=value,] [fill=value,]
            [files={filename1, filename2, ...}]
  romname:  [origin=value,] [length=value,] [romwidth=value,]
            [memwidth=value,] [fill=value,]
            [files={filename1, filename2, ...}]
  ...
}

```

ROMS 伪指令定义的开始。

romname 存储器区域名。存储器区域名可以是1~8个字符。名对程序并无意义，仅用于识别存储器区域（允许完全相同的存储区域名）。

origin 存储器区域的开始地址，可以输入origin或缩写org或o，输入值必须是十进制，八进制或十六进制数。如果省略origin的值，默认值为0，如下表所示：

常 数	符 号	例
十六进制	0x 前缀或 h 后缀	0x77 或 077h
八进制	0 前缀	077
十进制	无前缀、后缀	77

length 指定存储器区域的长度为ROM器件的物理长度。可以输入length、len或l，其值必须是一个十六进制、八进制或十进制常数。如果省略长度值，则默认为整个地址空间长度。

romwidth 指定以位计的ROM物理宽度（见第10.3.3小节数据划分为输出文件）。此处指定的任何值将覆盖-romwidth选项。该数值必须是一个十六进制、八进制或十进制常数且为大于等于8和8的倍数。

memwidth 指定以位计的存储器宽度（见第10.3.2小节指定存储器宽度）。此处指定的任何值将覆盖-memwidth选项。该数值必须是一个十六进制、八进制或十进制常数且为大于等于8和8的倍数。当使用memwidth参数时，必须在SECTIONS伪指令中为每个段指定paddr参数（见第10.5节 SECTIONS伪指令）。

fill 指定填充值。在映像模式中，十六进制转换应用程序用此值填充一个区域内段之间的任何空位。空位是输入段之间的区域。填充值必须是一个十六进制、八进制或十进制常数，常数的宽度等于目标宽度。此处指定

files

的任何值将覆盖-fill选项。使用填充时，必须使用-image命令行选项。指定对应于此范围的输出文件名。在大括号内应包含文件名列表，其次序是由低到高。文件名的数目必须等于区域内将产生的输出文件数。计算输出文件产生的数量见10.3.3小节数据划分到输出文件。如果列表的文件名太多或太少，应用程序会提出警告。除非使用-image选项，否则定义区的所有参数都是可选的。命令和等于符号也是可选的。没有起点和长度的区域定义为整个地址空间。在映像模式下，所有的区域都要求有起点和长度。在同一页上的区域互相不能重叠，必须按地址增加的顺序列出。

10.4.1 何时使用 ROMS 伪指令

如果用户未使用 ROMS 伪指令，那么应用程序将定义一个包括全部地址空间的默认区域。这与 ROMS 伪指令有一个没有起点和长度的单个区域完全一样。

当用户想要实现下列各项时，可使用ROMS伪指令：

- 把大量数据编程到尺寸固定的 ROM 中。当用户规定对应于 ROM 长度的存储器区域时，应用程序自动将输出划分成适合的 ROM 块。
- 限制输出到某个段。使用 ROMS 伪指令可以把转换限制到目标地址空间的某个特定的段或某些段。应用程序不会转换 ROMS 伪指令定义区域以外的数据。段可以跨越区域边界，但应用程序将数据在边界处分成多个部分。如果一个段完全落在定义的任何区域之外，应用程序不会转换这个段而且不会发出信息或警告。用这种方法，用户可通过用 SECTIONS 伪指令来排除某些段。然而，如果段的一部分落在区域之内而另一部分落在未配置的存储器中，应用程序将发出警告并只转换处于区域内的那部分。
- 使用映像模式。当使用-image选项时，用户必须使用 ROMS 伪指令。对于整个区域，为了使在一个区域的每个输出文件包含数据，每个区域全部都要填充。段前的空位、段之间的空位或者是段后的空位都要用 ROMS 伪指令中的填充值填充。填充值是-fill选项指定的值或默认值 0。

10.4.2 ROMS 伪指令举例

在例 10.1 中说明了 ROMS 伪指令如何把 16 位存储器的 16KB 划分成 2 个 8K×8 位的 EPROM。图 10.6 所示为需要说明的输入和输出文件。

例 10.1 ROM 伪指令

```
infile.out
-image
-memwidth 16
ROMS
{
```

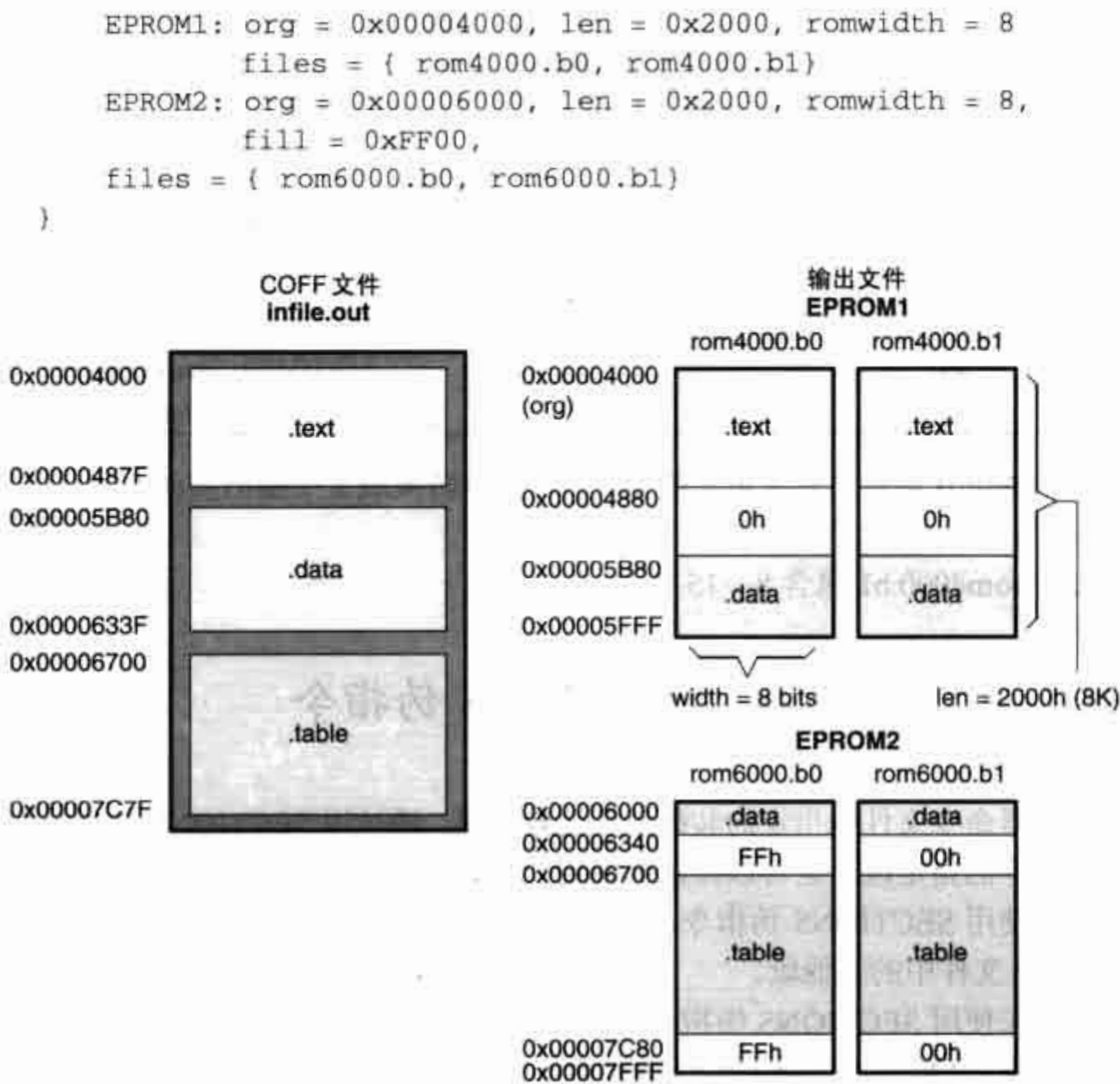



图 10.6 infile.out 文件划分成 4 个输出文件

映像文件（由-map 选项指定）有利于用户使用多个区域的 ROMS 伪指令，它显示出每一个区域、它的参数、有关的输出文件的名字以及按地址划分的内容列表。例 10.2 是例 10.1 的一个段的映像文件结果。

例 10.2 例 10.1 的映像文件输出，显示存储器区

```
-----
00004000..00005fff Page=0 Width=8 "EPROM1"
-----
OUTPUT FILES:  rom4000.b0 [b0..b7]
                rom4000.b1 [b8..b15]

CONTENTS:      00004000..0000487f .text
                00004880..00005b7f FILL = 00000000
                00005b80..00005fff .data
-----
00006000..00007fff Page=0 Width=8 "EPROM2"
-----
OUTPUT FILES:  rom6000.b0 [b0..b7]
```

```

rom6000.b1 [b8..b15]
CONTENTS: 00006000..0000633f .data
           00006340..000066ff FILL = 0000ff00
           00006700..00007c7f .table
           00007c80..00007fff FILL = 0000ff00

```

EPROM1 定义从 0x00004000 到 0x00005FFF 的地址范围。此区域包含以下段：

段	所属范围
.text	0x00004000~0x0000487F
.data	0x00005B80~0x00005FFF

剩余的区域用 0 填充（默认填充值为 0）。数据转换成 2 个输出文件：

- ☐ 文件 rom4000.b0 包含 0~7 位。
- ☐ 文件 rom4000.b1 包含 8~15 位。

10.5 SECTIONS 伪指令

与在链接器命令文件中指定的装载地址比较可见，通过用 SECTIONS 伪指令命名可以转换 COFF 文件的指定段。也可以指定那些用户想放在 ROM 中不同地址的段。

- ☐ 如果使用 SECTIONS 伪指令，那么应用程序仅仅转换用伪指令列出的段，而忽略 COFF 文件中的其他段。
- ☐ 如果未使用 SECTIONS 伪指令，那么应用程序转换所有落在已配置存储器内的初始化段。TMS320C28x C/C++编译器产生的初始化段包括：.text、.const 和.cinit。无论用户是否在 SECTIONS 伪指令中指定未初始化段，它们都不被转换。

注意：TMS320C28x C/C++编译器自动生成以下段。

初始化段：.text、.const 和.cinit。

未初始化段：.bss、.stack 和.system。

在命令文件中使用 SECTIONS 伪指令（详见第 10.2.2 节 用命令文件调用十六进制转换应用程序）。SECTIONS 伪指令的一般格式为：

```

SECTIONS
{
  sname: [paddr=value],
  sname: [paddr=value],
  ...
}

```

SECTIONS 伪指令定义的开始。

sname 标识COFF输入文件中的段。如果用户指定了一个不存在的段，应用程序

发出警告并忽略它的名字。

paddr=value 指定要装载这个段的ROM物理地址。这个值会覆盖链接器给定这个段的装载地址。这个值必须是十六进制、八进制或十进制常数。如果一个段使用了进制选项，所有段都必须使用它。

分隔段名的逗号是可选的。为了与链接器的 SECTIONS 伪指令更相似，可在段名后用冒号。

假设一个 COFF 文件包含 6 个初始化段：.text、.data、.cons、.vectors、.coeff 和 .tables。如果用户仅仅想转换 .text 和 .data，可使用如下的 SECTIONS 伪指令：

```
SECTIONS{.text, .data}
```

10.6 分配输出文件名

当十六进制转换应用程序将用户的 COFF 目标文件转换为一种数据格式时，它将把数据划分成一个或多个输出文件。当通过存储器字分离为 ROM 字时文件被格式化，此时总是以最低到最高的次序分配文件名，其中存储器字中的位是从右自左编号。而不论目标文件或 COFF endian 次序。

当指定输出文件名时，十六进制转换应用程序遵循下列顺序：

(1) **寻找 ROMS 伪指令**。如果一个文件与 ROMS 伪指令中的区域有关，而且该区域还包括了一个文件表 (files = { ... })，那么应用程序就从这个表中读取文件名。

例如，假设正在转换一个 16 位字的目标数据为 8 位宽的两个文件。用 ROMS 伪指令命名输出文件，可以指定：

```
ROMS
{
  RANGE1: romwidth=8, files={xyz.b0 xyz.b1}
}
```

通过写低字节到 xyz.b0 和高字节到 xyz.b1，应用程序创建两个输出文件。

(2) **寻找 -o 选项**。使用 -o 选项可以为输出文件指定文件名。如果在 ROMS 伪指令中没有列出文件名但用户使用了 -o 选项，应用程序将从 -o 选项的列表中读取文件名。下面的例与上例使用 ROMS 伪指令的效果相同：

```
-o xyz.b0 -o xyz.b1
```

如果 ROMS 伪指令和 -o 选项一起使用，ROMS 伪指令将覆盖 -o 选项。

(3) **分配默认文件名**。如果未指定文件名和文件名少于输出文件数，应用程序分配一个默认的文件名。默认文件名是由 COFF 输入文件名作基本名字再加上 2~3 个字符的扩展名。扩展名有 3 部分：

① 基于输出格式的格式符：

a ASCII-Hex

i Intel
t TI-Tagged
m Motorola-S
x Tektronix

② 在 ROMS 伪指令中的区域从 0 开始编号。如果没有 ROMS 伪指令，或仅仅只有一个区域，应用程序忽略这个编号。

③ 在该区域的一组文件中的文件号，最低文件从 0 开始。例如，假设对于 16 位目标处理器 `coff.out` 是一个 COFF 文件，而用户正在创建 Intel 格式的输出。由于没有指定输出文件名，应用程序生成 2 个输出文件名 `coff.i0` 和 `coff.i1`。

```

ROMS
{
    range1: o= 0x00001000 l = 0x1000
    range2: o= 0x00002000 l = 0x1000
}
  
```

输出文件	数据的范围
<code>coff.i00</code> 和 <code>coff.i01</code>	0x00001000 到 0x00001FFF
<code>coff.i10</code> 和 <code>coff.i11</code>	0x00002000 到 0x00002FFF

10.7 映像模式和填充 (-fill) 选项

本节指出在映像模式下运行的优点，并说明用一个精确的、连续的目标存储器区域映像生成输出文件的方法。

10.7.1 生成一个存储器映像

使用 `-image` 选项，通过填充所有在 ROMS 伪指令中指定的映射区域，应用程序生成一个存储器映像图。

COFF 文件由一些分配了存储器单元的存储块组成。典型的情况是所有的段并不是互相邻接的：地址空间内的段与段之间存在空位。当转换这种文件时，如果不使用映射模式，那么十六进制转换应用程序通过使用输出文件中的地址记录跨越这些空位，从下一个段开始。换句话说，在输出文件中可能存在不连续的地址。某些 EPROM 不支持地址的不连续性。

在映像模式中，不存在不连续的地址。每个输出文件包含连续的数据流，它们与目标存储器中的地址区域严格对应。任何段前的空位、段之间的空位和段后的空位都被提供的填充值填充。

使用映像模式转换的输出文件仍具有地址记录，因为大多数十六进制格式要求每一行都有一个地址。而且，在映像模式中，这些地址总是连续的。

注意：定义目标存储区的区域。如果使用映像模式，也就必须使用 ROMS 伪指令。在映像模式中，每个输出文件直接对应一个目标存储器区域。用户必须定义这个区域。如果用户没有提供目标存储器的区域，那么应用程序将建立整个目标处理器地址空间的存储器映像图，这可能创建一个巨大的输出数据。为了避免这种情况的发生，应用程序要求用户使用 ROMS 伪指令明确限定地址空间。

10.7.2 填充值

填充 (-fill) 选项为段之间的空位指定填充值。填充值必须是跟在 -fill 选项后的一个整数。假设常数的宽度是目标处理器的一个字。例如，指定 -fill 0FFh，其填充模式结果是 00FFh。该常数值不进行符号扩展。

如果用户没有用 fill 选项指定填充值，应用程序使用默认填充值 0。-fill 选项仅在用户使用了 -image 时有效，否则它将被忽略。

10.7.3 在映像模式下所遵循的步骤

第一步：用 ROMS 伪指令定义目标存储器区域。详见第 10.4 节 ROMS 伪指令。

第二步：用 -image 选项调用十六进制转换应用程序。可以选择使用 -zero 选项为每一个输出文件重新设置起始地址。如果用户没有在 ROMS 伪指令指定填充值而又想用与默认值不同的数值，那么可使用 -fill 选项。

10.8 控制 ROM 器件地址

十六进制转换应用程序的输出地址对应于 ROMS 伪指令的地址。EPROM 程序编程器在所指定的单元烧写数据，该指定单元在十六进制转换应用程序输出文件的地址区域中指定。十六进制转换应用程序提供某些方法用于控制在 ROM 中每一段的起始地址。然而大多数 EPROM 程序编程器提供直接控制烧写数据的地址。

十六进制转换应用程序输出文件的地址区域由以下几项控制，按从低到高优先级列出：

(1) 链接器命令文件。默认情况下，十六进制转换应用程序输出文件的地址区域是在链接器命令文件中给出的装载地址。

(2) SECTIONS 伪指令的 paddr 选项。对于指定了 paddr 选项的段，十六进制转换应用程序会绕过段的装载地址，而将该段放置在由 paddr 选项指定的地址中。

(3) -zero 选项。当用户使用 -zero 选项时，应用程序为每个输出文件重新将起始地址设置为 0。由于每个文件的起始地址为 0 且向上递增，所以任何地址记录说明的偏移量都是相对于文件的起始地址的（在 ROM 内的地址），而不是数据实际的目标地址。

(4) -byte 选项。某些 EPROM 程序编程器可能要求输出文件地址区域包含字节计数而不是字计数。如果用户使用 -byte 选项，那么输出文件地址每个字节仅增加一次。例如，如果起始地址是 0，第一行包括 8 个字，而用户没有使用 -byte 选项，那么第二行就会从 8

(8h) 开始。如果起始地址是 0，第一行包括 8 个字，而用户使用了 `-byte` 选项，那么第二行就会从 16 (010h) 开始。这两个例中的数据是相同的；`-byte` 影响的仅仅是输出文件地址区域的计算；不影响被转换数据的实际目标处理器地址。

(5) `-byte` 选项使输出文件中的记录指向文件内的字节单元，而不论目标处理器是否是字节地址。

10.9 目标格式

十六进制转换应用程序把 COFF 目标文件转换成大多数 EPROM 程序编程器可接受的五种输入目标格式之一：ASCII-Hex、Intel MCS-86、Motorola-S、扩展 Tektronix 和 TI-Tagged。表 10.2 指定了格式的选项。

- 在命令行中仅需要其中一个选项。如果用户使用了多个选项，那么列出的最后一个选项将覆盖其他选项。
- 默认选项为 Tektronix (`-x` 选项)。

表 10.2 指定十六进制转换格式的选项

选 项	格 式	地 址 位	默 认 宽 度
<code>-a</code>	ASCII-Hex	16	8
<code>-i</code>	Intel	32	8
<code>-m</code>	Motorola-S	24	8
<code>-t</code>	TI-Tagged	16	16
<code>-x</code>	Tektronix	32	8

地址位 决定该格式支持多少位地址信息。16 位地址格式仅支持 64K 地址。应用程序会截取目标地址以满足地址位的大小。

默认宽度 决定格式的默认输出宽度。可以使用 `-romwidth` 选项改变默认宽度，或在 ROMS 伪指令中使用 `romwidth` 参数，但不能改变 Ti-Tagged 格式的默认宽度。它仅仅支持 16 位的宽度。

10.9.1 ASCII-Hex 目标格式 (`-a` 选项)

ASCII-Hex 目标格式支持 16 位地址。此格式是由空格隔开的字节群组成，如图 10.7 所示：



图 10.7 ASCII-Hex 目标格式

10.9.3 Motorola-S 目标格式 (-m 选项)

Motorola-S 格式支持 24 位地址。它由文件起始（头）记录、数据记录 and 文件（终止）结束记录组成。每个记录由 5 部分组成：记录类型、字节计数、地址、数据以及校验和。记录类型为：

记录类型	描 述
S0	文件头部记录
S2	代码/数据记录
S8	文件终止记录

字节计数是记录中除了类型和字节计数本身之外的字符对计数。

校验和是组成字节计数、地址以及代码/数据域的字符对所表示的数值和的补码低有效字节。

图 10.9 所示为 Motorola-S 十六进制目标格式。

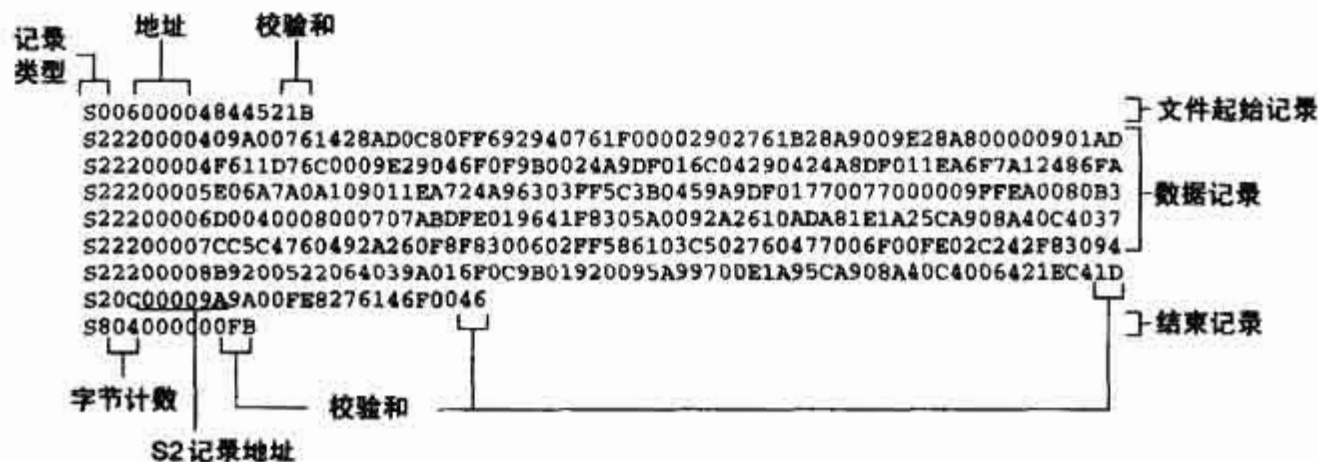


图 10.9 Motorola-S 十六进制目标格式

10.9.4 TI-Tagged SDSMAC 目标格式 (-t 选项)

Texas Instruments SDSMAC (TI-Tagged) 格式支持 16 位地址。它由文件起始记录、数据记录 and 文件结束记录组成。每一个数据记录由一系列域组成并由一个标识字符表示。有意义的标识字符有：

标识字符	描 述
k	后随程序标识符
7	后随校验和
8	后随伪校验和（忽略）
9	后随 16 位装载地址
B	后随数据字（4 个字符）
F	识别数据记录的结束
*	后随数据字节（2 个字符）

10.10 十六进制转换应用程序错误信息

Section mapped to reserved memory message

描述： 段被映射到处理器存储器映像中列出的保留存储器区域。

操作： 修正段和引导装载器地址。对于有效存储器单元参见 *TMS320C28x CPU and Instruction Set Reference Guide*。

Sections overlapping

描述： 两个或更多的COFF段的装载地址相重叠，或引导表地址与另一个段相重叠。

操作： 产生这个问题的可能原因是：当存储器的宽度小于数据宽度时，十六进制转换应用程序执行从装载地址至十六进制输出文件地址的不正确转换。见10.3节存储器宽度和10.8节ROMS伪指令。

Unconfigured memory error

描述： COFF文件包含了装载地址落在ROMS伪指令指定的存储器范围以外的段。

操作： 修正ROMS伪指令指定的ROMS范围，或者修改段的装载地址，以覆盖所需的存储器区域。记住，如果没有使用ROMS伪指令，那么存储器区域默认为整个处理器地址空间。因此，去掉ROMS伪指令也是可行的。





第 11 章 C28x 汇编语言指令集

本章介绍了指令集概要、定义了特殊符号和所使用的符号，按字母顺序详细地说明了每一条指令。

11.1 指令概述（按功能分类）

注意：本章的例程都假定器件已经工作在 C28x 模式（OBJMODE == 1，AMODE == 0）。复位后，若要使器件进入 C28x 模式，必须通过执行指令“C28OBJ”（或“SETC OBJMODE”）对 ST1 寄存器中的 OBJMODE 位置位。

表 11.1 指令一览表（以功能分类）

符 号	描 述
XARn	XAR0~XAR7 寄存器
ARn, ARm	XAR0~XAR7 寄存器的低 16 位
ARnH	XAR0~XAR7 寄存器的高 16 位

续表

符 号	描 述
AR _{Pn}	ARP0~ARP7, 3 位辅助寄存器指针, ARP0 指向 XAR0, ARP7 指向 XAR7
AR (ARP)	ARP 指向的辅助寄存器低 16 位
XAR (ARP)	ARP 指向的辅助寄存器
AX	累加器的高和低寄存器
#	立即数
PM	乘积移位方式 (+4, 1, 0, -1, -2, -3, -4, -5, -6)
PC	程序计数器
~	按位求反码
[loc16]	16 位地址单元的内容
0:[loc16]	零扩展 16 位地址单元的值
S:[loc16]	符号扩展 16 位地址单元的值
[loc32]	32 位地址单元的内容
0:[loc32]	零扩展 32 位地址单元的值
S:[loc32]	符号扩展 32 位地址单元的值
7bit	7 位立即数
0:7bit	零扩展的 7 位立即数
S:7bit	符号扩展的 7 位立即数
8bit	8 位立即数
0:8bit	零扩展的 8 位立即数
S:8bit	符号扩展的 8 位立即数
10bit	10 位立即数
0:10bit	零扩展的 10 位立即数
S:10bit	符号扩展的 10 位立即数
16bit	16 位立即数
0:16bit	零扩展的 16 位立即数
S:16bit	符号扩展的 16 位立即数
22bit	22 位立即数
0:22bit	零扩展的 22 位立即数
LSb	最低有效位
LSB	最低有效字节
LSW	最低有效字
MSb	最高有效位
MSB	最高有效字节
MSW	最高有效字
OBJ	指令的 OBJMODE 位的状态
N	重复计数 (N = 0, 1, 2, 3, 4, 5, 6, 7……)
[]	可选项
=	赋值
==	等于

11.2 寄存器操作

注意：本章的例程都假定器件已经工作在 C28x 模式（OBJMODE == 1，AMODE == 0）。复位后，若要使器件进入 C28x 模式，必须通过执行指令“C28OBJ”（或“SETC OBJMODE”）对 ST1 寄存器中的 OBJMODE 位置位。

表 11.2 寄存器操作

助 记 符	描 述
XARn 寄存器（XAR0~XAR7）操作	
ADDB XARn,#7bit	7 位立即数与辅助寄存器相加，结果保存到辅助寄存器
ADRK #8bit	8 位立即数与当前辅助寄存器相加，结果保存到当前辅助寄存器
CMPR 0/1/2/3	比较辅助寄存器
MOV AR6/7,loc16	装载辅助寄存器
MOV loc16,ARn	保存辅助寄存器的 16 位数
MOV XARn,PC	保存当前程序计数器
MOVB XARn,#8bit	用 8 位数装载辅助寄存器
MOVB XAR6/7,#8bit	用 8 位立即数装载辅助寄存器
MOVL XARn,loc32	装载 32 位辅助寄存器
MOVL loc32,XARn	保存 32 位辅助寄存器
MOVL XARn,#22bit	用 32 位立即数装载辅助寄存器
MOVZ ARn,loc16	装载 XARn 的低 16 位，高 16 位清 0
SBRK #8bit	从当前辅助寄存器中减去 8 位立即数
SUBB XARn,#7bit	从辅助寄存器中减去 7 位立即数
DP 寄存器操作	
MOV DP,#10bit	装载数据页面指针
MOVW DP,#16bit	装载整个数据页面
MOVZ DP,#10bit	装载数据页面和清高位
SP 寄存器操作	
ADDB SP,#7bit	堆栈指针加 7 位立即数
POP ACC	从堆栈中弹出 ACC 累加器
POP AR1:AR0	从堆栈中弹出 AR1 和 AR0
POP AR1H:AR0H	从堆栈中弹出 AR1H 和 AR0H
POP AR3:AR2	从堆栈中弹出 AR3 和 AR2 寄存器
POP AR5:AR4	从堆栈中弹出 AR5 和 AR4 寄存器
POP DBGIER	从堆栈中弹出 DBGIER 寄存器
POP DP:ST1	从堆栈中弹出 DP 和 ST1 寄存器
POP DP	从堆栈弹出 DP 寄存器
POP IFR	从堆栈弹出 IFR 寄存器
POP loc16	从堆栈弹出“loc16”数据

续表

助 记 符	描 述
POP P	从堆栈弹出 P 寄存器
POP RPC	从堆栈弹出 RPC 寄存器
POP ST0	从堆栈弹出 ST0 寄存器
POP ST1	从堆栈弹出 ST1 寄存器
POP T:ST0	从堆栈弹出 T 和 ST0 寄存器
POP XT	从堆栈弹出 XT 寄存器
POP XARn	从堆栈弹出辅助寄存器
PUSH ACC	压 ACC 累加器中的值到堆栈
PUSH ARn:ARn	压 ARn 和 ARn 寄存器中的值到堆栈
PUSH AR1H:AR0H	压 AR1H 和 AR0H 寄存器中的值到堆栈
PUSH DBGIER	压 DBGIER 寄存器中的值到堆栈
PUSH DP:ST1	压 DP 和 ST1 寄存器中的值到堆栈
PUSH DP	压 DP 寄存器中的值到堆栈
PUSH IFR	压 IFR 寄存器中的值到堆栈
PUSH loc16	压 loc16 的地址单元中的数据到堆栈
PUSH P	压 P 寄存器中的值到堆栈
PUSH RPC	压 RPC 寄存器中的值到堆栈
PUSH ST0	压 ST0 寄存器中的值到堆栈
PUSH ST1	压 ST1 寄存器中的值到堆栈
PUSH T:ST0	压 T 和 ST0 寄存器中的值到堆栈
PUSH XT	压 XT 寄存器中的值到堆栈
PUSH XARn	压 XARn 寄存器中的值到堆栈
SUBB SP,#7bit	从堆栈指针中减去 7 位立即数
AX 寄存器 (AH, AL) 操作	
ADD AX,loc16	AX 与指定地址单元的内容相加, 结果保存到 AX 中
ADD loc16,AX	AX 与指定地址单元的内容相加, 结果保存到指定地址单元中
ADDB AX,#8bit	8 位立即数与 AX 相加, 结果保存到 AX 中
AND AX,loc16,#16bit	按位“与”
AND AX,loc16	按位“与”
AND loc16,AX	按位“与”
ANDB AX,#8bit	8 位数按位“与”
ASR AX,1...16	算术右移
ASR AX,T	算术右移, 右移位数由 T (3:0) = 0...15 设置
CMP AX,loc16	与 loc16 地址单元的值比较
CMP AX,#8bit	与 8 位立即数比较
FLIP AX	交换 AX 寄存器中位的次序
LSL AX,1...16	逻辑左移
LSL AX,T	逻辑左移, 由 T (3:0) = 0...15 设置左移位数
LSR AX,1...16	逻辑右移

续表

助 记 符	描 述
LSR AX,T	逻辑右移, 由 T (3:0) = 0...15 设置右移位数
MAX AX,loc16	求最大值
MIN AX,loc16	求最小值
MOV AX,loc16	装载 AX
MOV loc16,X	保存 AX
MOV loc16,AX,COND	条件保存 AX 寄存器
MOVB AX,#8bit	装载 8 位立即数到 AX
MOVB AX.LSB,loc16	装载 AX 寄存器的最低字节, MSB = 0x00
MOVB AX.MSB,loc16	装载 AX 寄存器的最高字节, LSB = 不变
MOVB loc16,AX.LSB	保存 AX 寄存器的最低字节
MOVB loc16,AX.MSB	保存 AX 寄存器的最高字节
NEG AX	求 AX 寄存器中的值的负数
NOT AX	求 AX 寄存器中的值的“非”
OR AX,loc16	按位“或”
OR loc16,AX	按位“或”
ORB AX,#8bit	8 位立即数按位“或”
SUB AX,loc16	从 AX 中减去指定地址单元的内容
SUB loc16,AX	从指定地址单元的值中减去 AX
SUBR loc16,AX	指定地址单元的值与 AX 相减, 结果保存到指定地址单元
SXTB AX	对 AX 寄存器的 MSB 进行符号扩展
XOR AX,loc16	按位“异或”
XORB AX,#8bit	8 位立即数按位“异或”
XOR loc16,AX	按位“异或”
16 位 ACC 累加器操作	
ADD ACC,loc16{<<0...16}	(loc16) 地址单元中的数加到累加器, 结果保存到 ACC
ADD ACC,#16bit{<<0...15}	16 位立即数与累加器相加, 结果保存到 ACC
ADD ACC,loc16<<T	移位后的值与累加器相加, 结果保存到 ACC
ADDB ACC,#8bit	8 位立即数与累加器相加, 结果保存到 ACC
ADDCU ACC,loc16	无符号数与累加器带进位位相加, 结果保存到 ACC
ADDU ACC,loc16	无符号数与累加器相加, 结果保存到 ACC
AND ACC,loc16	按位“与”
AND ACC,#16bit{<<0...16}	按位“与”
MOV ACC,loc16{<<0...16}	指定地址单元的内容移位后装载累加器
MOV ACC,#16bit{<<0...16}	#16bit 移位后装载累加器
MOV loc16,ACC<<1...8	保存累加器移位后的高字
MOV ACC,loc16<<T	loc16 地址单元中的内容移位并装载累加器
MOVB ACC,#8bit	8 位立即数装载累加器
MOVH loc16,ACC<<1...8	保存累加器移位后的高字到 16 位地址单元中
MOVU ACC,loc16	用 16 位地址中的无符号数装载累加器

续表

助 记 符	描 述
SUB ACC,loc16<<T	从累加器中减去 16 位地址的内容移位后的值
SUB ACC,loc16{<<0...16}	从累加器中减去 16 位地址的内容移位后的值
SUB ACC,#16bit{<<0...15}	从累加器中减去移位后的值
SUBB ACC,#8bit	从累加器中减去 8 位数
SBBU ACC,loc16	累加器与 16 位地址的无符号数带借位相减
SUBU ACC,loc16	从累加器中减去无符号 16 位地址的内容
OR ACC,loc16	按位“或”
OR ACC,#16bit{<<0...16}	按位“或”
XOR ACC,loc16	按位“异或”
XOR ACC,#16bit{<<0...16}	按位“异或”
ZALR ACC,loc16	AH = [loc16], AL = 0x8000
32 位 ACC 累加器操作	
ABS ACC	累加器取绝对值
ABSTC ACC	累加器取绝对值, 并装载 TC
ADDL ACC,loc32	32 位地址的内容与累加器相加, 结果保存到 ACC
ADDL loc32,ACC	累加器与指定地址相加, 结果保存到指定地址
ADDCL ACC,loc32	32 位地址的内容与累加器带进位位相加, 结果保存到 ACC
ADDUL ACC,loc32	32 位地址的无符号数与累加器相加, 结果保存到 ACC
ADDL ACC,P<<PM	移位后的 P 与累加器相加, 结果保存到 ACC
ASRL ACC,T	算术右移累加器, 右移位数由 T (4:0) 位设置
CMPL ACC,loc32	比较 32 位地址的内容
CMPL ACC,P<<PM	比较 32 位数
CSB ACC	计算符号位
LSL ACC,1...16	逻辑左移 1~16 位
LSL ACC,T	逻辑左移 T (3:0) = 0...15 位
LSRL ACC,T	逻辑右移 T (4:0) 位
LSLL ACC,T	逻辑左移 T (4:0) 位
MAXL ACC,loc32	求取 ACC 与 loc32 地址中的内容的较大值
MINL ACC,loc32	求取 ACC 与 loc32 地址中的内容的较小值
MOVL ACC,loc32	用 32 位地址的内容装载累加器
MOVL loc32,ACC	累加器值保存到 32 位地址单元中, [loc32] = ACC
MOVL P,ACC	从累加器中装载 P
MOVL ACC,P<<PM	用移位后的 P 装载累加器
MOVL loc32,ACC,COND	条件保存 ACC 到 32 位地址单元中
NORM ACC,XARn+ +/-	规格化 ACC 和修改所选辅助寄存器
NORM ACC,*ind	与 C2XLP 兼容模式规格化 ACC 操作
NEG ACC	求 ACC 的负数
NEGTC ACC	若 TC = 1, 则求 ACC 的负数
NOT ACC	对 ACC 求“非”

续表

助 记 符	描 述
ROL ACC	循环左移 ACC
ROR ACC	循环右移 ACC
SAT ACC	根据 OVC 的值使 ACC 的值为饱和值
SFR ACC,1...16	右移累加器 1~16 位
SFR ACC,T	右移累加器, 移位位数由 T (3:0) = 0...15 位设置
SUBBL ACC,loc32	32 位地址的内容与 ACC 带借位位相减, 结果保存到 ACC
SUBCU ACC,loc16	条件减 16 位地址的内容
SUBCUL ACC,loc32	条件减 32 位地址的内容
SUBL ACC,loc32	减去 32 位地址的内容, 结果保存到 ACC
SUBL loc32,ACC	减去 32 位地址的内容, 结果保存到 32 位地址单元中
SUBL ACC,P<<PM	减去 32 位数
SUBRL loc32,ACC	[loc32] = ACC-[loc32]
SUBUL ACC,loc32	减去 32 位地址单元中的无符号数, 结果保存到 ACC
TEST ACC	测试累加器是否等于零
64 位 ACC:P 寄存器操作	
ASR64 ACC:P,#1...16	64 位数算术右移, 右移位数从 1~16
ASR64 ACC:P,T	64 位数算术右移, 其右移位数由 T (5:0) 位设置
CMP64 ACC:P	比较 64 位数
LSL64 ACC:P,1...16	逻辑左移 1~16 位
LSL64 ACC:P,T	64 位数逻辑左移, 其左移位数由 T (5:0) 位设置
LSR64 ACC:P,#1...16	64 位数逻辑右移 1~16 位
LSR64 ACC:P,T	64 位数逻辑右移, 其右移位数由 T (5:0) 位设置
NEG64 ACC:P	求 ACC:P 的负数
SAT64 ACC:P	根据 OVC 值使 ACC:P 的值为饱和值
P 或 XT 寄存器 (P, PH, PL, XT, T, TL) 操作	
ADDUL P,loc32	32 位地址的无符号数加到 P
MAXCUL P,loc32	条件求取 P 和 32 位地址的内容的无符号最大值
MINCUL P,loc32	条件求取 P 和 32 位地址的内容的无符号最小值
MOV PH,loc16	装载 P 寄存器的高 16 位
MOV PL,loc16	装载 P 寄存器的低 16 位
MOV loc16,P	保存移位后的 P 寄存器中的值的低 16 位
MOV T,loc16	装载 XT 寄存器的高 16 位
MOV loc16,T	保存 T 寄存器
MOV TL,#0	XT 寄存器的低 16 位清 0
MOVA T,loc16	装载 T 寄存器, 并加上先前的乘积
MGVAD T,loc16	装载 T 寄存器
MOVDL XT,loc32	保存 XT, 并装载新的 XT 值
MOVH loc16,P	保存 P 寄存器的高字
MOVL P,loc32	装载 P 寄存器

续表

助 记 符	描 述
MOVL loc32,P	保存 P 寄存器
MOVL XT,loc32	装载 XT 寄存器
MOVL loc32,XT	保存 XT 寄存器
MOVP T,loc16	装载 T 寄存器, 并保存 P 寄存器到累加器
MOVS T,loc16	装载 T, 并从累加器中减去 P
MOVX TL,loc16	用符号扩展装载 XT 的低 16 位
SUBUL P,loc32	减去无符号 32 位数
16X16 乘法操作	
DMAC ACC:P, loc32, *XAR7/++	两个 16 位数相乘并累加
MAC P,loc16,0:pma	乘并累加
MAC P,loc16,*XAR7/++	乘并累加
MPY P,T,loc16	16×16 位乘法
MPY P,loc16,#16bit	16×16 位乘法
MPY ACC,T,loc16	16×16 位乘法
MPY ACC,loc16,#16bit	16×16 位乘法
MPYA P,loc16,#16bit	16×16 位乘法, 并加上先前的乘积
MPY P,T,loc16	16×16 位乘法, 并加上先前的乘积
MPYB P,T,#8bit	有符号数与一个无符号的 8 位立即数相乘
MPYS P,T,loc16	16×16 位乘减
MPYB ACC,T,#8bit	ACC 与一个 8 位立即数相乘
MPYU ACC,T,loc16	16×16 位无符号乘法
MPYU P,T,loc16	无符号 16×16 位乘法
MPYXU P,T,loc16	符号数与无符号数乘
MPYXU ACC,T,loc16	有符号数与无符号数乘
SQRA loc16	对[loc16]作平方, P 与累加器相加结果送累加器
SQRS loc16	对[loc16]作平方, P 与累加器相减结果送累加器
XMAC P,loc16,* (pma)	与 C2XLP 源代码兼容的乘加
XMAC P,loc16,* (pma)	与 C2XLP 源代码兼容且带数据移动乘加
32×32 位乘法操作	
IMACL P,loc32,*XAR7/++	有符号 32×32 位乘加 (低 38 位存于 P)
IMPYAL P,XT,loc32	有符号 32 位乘法 (低 38 位), 并加上先前的 P
IMPYL P,XT,loc32	有符号 32×32 位乘法 (低 38 位)
IMPYL ACC,XT,loc32	有符号 32×32 位乘法 (低 32 位)
IMPYSL P,XT,loc32	有符号 32 位乘法 (低 38 位), 并减去 P
IMPYXUL P,XT,loc32	有符号 32 位×无符号 32 位 (低 38 位)
QMACL P,loc32,*XAR7/++	有符号 32×32 位乘加 (高 32 位)
QMPYAL P,XT,loc32	有符号 32 位乘法 (高 32 位), 并加上先前的 P
QMPYL ACC,XT,loc32	有符号 32×32 位乘法 (高 32 位)
QMPYL P,XT,loc32	有符号 32×32 位乘法 (高 32 位)

续表

助 记 符	描 述
QMPYSL P,XT,loc32	有符号 32 位乘法 (高半部分), 并减去先前的 P
QMPYUL P,XT,loc32	无符号 32×32 位乘法 (高 32 位)
QMPYXUL P,XT,loc32	有符号 32 位×无符号 32 位 (高 32 位)
直接存储器操作	
ADD loc16, #16bitSigned	有符号 16 位立即数与指定地址单元的 16 位数相加, 结果保存在指定地址单元中
AND loc16, #16bitSigned	按位“与”
CMP loc16, #16bitSigned	比较
DEC loc16	减 1
DMOV loc16	移动 16 位地址的内容
INC loc16	加 1
MOV *(0:16bit), loc16	移动数据
MOV loc16, *(0:16bit)	移动数据
MOV loc16, #16bit	保存 16 位立即数
MOV loc16, #0	16 位地址的内容清 0
MOVB loc16, #8bit, COND	条件保存字节
OR loc16, #16bit	按位“或”
TBIT loc16, #bit	位测试
TBIT loc16, T	测试 T 寄存器指定的位
TCLR loc16, #bit	测试并清 0 指定位
TSET loc16, #bit	测试并置位指定位
XOR loc16, #16bit	按位“异或”
I/O 端口操作	
IN loc16, *(PA)	从端口输入数据
OUT *(PA), loc16	输出数据到端口
UOUT *(PA), loc16	未加保护的输出数据到端口
程序存储器操作	
PREAD loc16, *XAR7	读程序存储器
PWRITE *XAR7, loc16	写程序存储器
XPREAD loc16, *AL	与 C2XLP 源代码兼容的程序读
XPREAD loc16, *(pma)	与 C2XLP 源代码兼容的程序读
XPWRITE *AL, loc16	与 C2XLP 源代码兼容的程序写
跳转/调用/返回操作	
B 16bitOff, COND	条件跳转
BANZ 16bitOff, ARn--	若辅助寄存器不等于零, 则跳转
BAR 16bOf, Am, Am, EQ, NEQ	与辅助寄存器比较后跳转
BF 16bitOff, COND	快速跳转
FFC XAR7, 22bitAddr	快速函数调用
IRET	中断返回

续表

助 记 符	描 述
LB 22bitAddr	长跳转 (22 位程序地址)
LB *XAR7	间接长跳转
LC 22bitAddr	直接长调用
LC *XAR7	间接长调用
LCR 22bitAddr	用 RPC 长调用
LCR *XAR7	用 RPC 间接长调用
LOOPZ loc16,#16bit	等于零时, 循环
LOOPNZ loc16,#16bit	不等于零时, 循环
LRET	长返回
LRETE	长返回, 并使能中断
LRETR	用 RPC 长返回
RPT #8bit/loc16	重复下一条指令
SB 8bitOff,COND	条件短跳转
SBF 8bitOff,EQ/NEQ/TC/NTC	条件快速短跳转
XB pma	与 C2XLP 源代码兼容的跳转
XB pma,COND	与 C2XLP 源代码兼容的条件跳转
XB pma,*,ARPN	与 C2XLP 源代码兼容的跳转函数调用
XB *AL	与 C2XLP 源代码兼容函数调用
XBANZ pma,*ind[,ARPN]	若 ARn 不等于零, 则为与 C2XLP 源代码兼容的跳转
XCALL pma	与 C2XLP 源代码兼容的函数调用
XCALL pma,COND	与 C2XLP 源代码兼容的条件函数调用
XCALL pma,*ARPN	与 C2XLP 源代码兼容的函数调用, 并同时修改 ARP
XCALL *AL	与 C2XLP 源代码兼容的间接函数调用
XRET	XRETC UNC 的别名
XRETC COND	与 C2XLP 源代码兼容的条件返回
中断寄存器操作	
AND IER,#16bit	按位“与”, 禁止指定的 CPU 中断
AND IFR,#16bit	按位“与”, 清除悬挂的 CPU 中断
LACK #16bit	中断应答
INTR INT1/...INT14 NMI EMUINT DLOGINT RTOSINT	仿真硬件中断
MOV IER,loc16	装载中断使能寄存器
MOV loc16,IER	保存中断使能寄存器
OR IER,#16bit	按位“或”
OR IFR,#16bit	按位“或”
TRAP #0...31	软件陷阱

续表

助 记 符	描 述
状态寄存器 (ST0, ST1) 操作	
CLRC Mode	状态位清 0
CLRC XF	XF 状态位清 0, 并输出信号
CLRC AMODE	AMODE 位清 0
C28ADDR	AMODE 状态位清 0
CLRC OBJMODE	OBJMODE 位清 0
C27OBJ	OBJMODE 位清 0
CLRC M0M1MAP	M0M1MAP 位清 0
C27MAP	M0M1MAP 位置 1
CLRC OVC	OVC 位清 0
ZAP OVC	溢出计数器清 0
DINT	可屏蔽中断禁止 (INTM 位置 1)
EINT	可屏蔽中断使能 (INTM 位清 0)
MOV PM,AX	装载乘积移位方式位 PM = AX (2:0)
MOV OVC,loc16	装载溢出计数器
MOVU OVC,loc16	用无符号数装载溢出计数器
MOV loc16,OVC	保存溢出计数器
MOVU loc16,OVC	保存无符号的溢出计数器
SETC Mode	乘积状态位置 1
SETC XF	XF 位置 1 和输出信号
SETC M0M1MAP	M0M1MAP 位置 1
C28MAP	M0M1MAP 位置 1
SETC OBJMODE	OBJMODE 位置 1
C28OBJ	OBJMODE 位置 1
SETC AMODE	AMODE 位置 1
LPADDR	SETC AMODE 的别名
SPM PM	设置乘积移位方式位
其他操作	
ABORTI	异常终止中断
ASP	对齐堆栈指针
EALLOW	对保护空间的访问使能
IDLE	处理器进入低功耗 (IDLE) 模式
NASP	不对齐堆栈指针
NOP {*ind}	空操作, 同时可以修改间接地址
ZAPA	零累加 P 寄存器和 OVC
EDIS	禁止对保护空间的访问
ESTOP0	仿真停止 0
ESTOP1	仿真停止 1

ABORTI

异常终止中断

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ABORTI	0000 0000 0000 0001	×	—	2

注：OBJ 模式为生成目标文件的格式，RPT 指可以重复性，CYC 为指令花费周期。

操作数： 无

描述： 异常终止中断。本指令在仿真时很有用。通常程序都使用 IRET 指令从中断返回。IRET 指令将自动恢复保存到堆栈中的所有数据。在恢复状态寄存器 ST1 和仿真状态寄存器（DBGSTST），IRET 指令将恢复中断前的仿真数据。

在某些目标应用软件中，可能不需要 IRET 指令从中断返回。不使用 IRET 指令可能会引起仿真逻辑问题，因为仿真逻辑都是假定需要恢复原来的仿真内容。异常终止中断指令（ABORTI）提供了不需要恢复仿真内容，仿真逻辑复位到默认状态的方法。ABORTI 指令包括以下操作部分：

- 对 ST1 中的 DBGGM 位置 1，禁止仿真事件。
- 修改 DBGSTAT 寄存器中的选择位，复位仿真内容。若在中断发生之前 CPU 处于仿真停止状态，则当中断停止时，CPU 不会停止工作。ABORTI 指令不会修改 DBGIER，IER，INTM 位或任何分析寄存器（例如，用于断点、观察点和数据的寄存器）。

标志与模式：DBGGM DBGGM 位置 1。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

ABS ACC

求累加器的绝对值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ABS ACC	1111 1111 0101 0110	×	—	1

操作数： ACC 累加器

描述： 取累加器内容的绝对值

```

if (ACC = 0x8000 0000)
    V = 1;
If (OVM = 1)
    ACC = 0x7FFF FFFF;
else
    ACC = 0x8000 0000;
else
    if (ACC < 0)
        ACC = -ACC;

```

标志与模式：N 若 ACC 的 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 该操作使进位标志位 C 清 0。

V 若操作开始时, $ACC = 0x8000\ 0000$, 认为是一个溢出值, 溢出标志位 V 置 1; 否则 V 不受影响。

OVM 若 $ACC = 0x8000\ 0000$, 认为是一个溢出值, 则 ACC 值由 OVM 的状态确定: 若 OVM 为 0, 则 ACC 将为 $0x8000\ 0000$; 若 OVM 置位, 则 ACC 将达到饱和值 $0x7FFF\ FFFF$ 。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;取 VarA 的绝对值, 并使其为饱和值

MOVL ACC, @VarA ;把 VarA 的值装入 ACC

SETC OVM ;使能溢出模式

ABS ACC ;取 ACC 的绝对值, 使其为饱和值

MOVL @VarA, ACC ;结果保存到 VarA

ABSTC ACC

求累加器的绝对值并装载 TC

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ABSTC ACC	0101 0110 0101 1111	1	-	1

操作数: ACC 累加器

描述: 取累加器的绝对值, 用符号位与先前测试控制位的值“异或”后装载测试控制位 TC。

```

if(ACC = 0x8000 0000)
{
    if(OVM = 1)
        ACC = 0x7FFF FFFF;
    else
        ACC = 0x8000 0000;
    V = 1;
    TC = TC XOR 1;
}
else
{
    if(ACC < 0)
        ACC = -ACC;
    TC = TC XOR 1;
}
C = 0;

```

标志与模式: **N** 若 ACC 的 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 清进位标志位 C。

V 若操作开始时, $ACC = 0x8000\ 0000$, 则认为是一个溢出值且溢出标志位 V 置位; 否则 V 不受影响。

TC 若操作开始时, $ACC < 0$, 则 $TC = TC \oplus 1$; 否则不影响 TC。
OVM 若操作开始时, $ACC = 0x8000\ 0000$, 它将认为是一个溢出值, 指令执行后的 ACC 值依赖于 OVM 的状态, 若 OVM 清 0 且 $TC == 1$, 则 ACC 将为 $0x8000\ 0000$; 若 OVM 置位且 $TC == 1$, 则 ACC 将为饱和值 $0x7FFF\ FFFF$ 。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;有符号数计算 Quot16 = Num16/Den16, Rem16 = Num16%Den16
CLRC TC                      ;符号标志位 TC 清 0
MOV ACC,@Den16 << 16         ;AH = Den16, AL = 0
ABSTC ACC                    ;取绝对值, TC = sign 异或 TC
MOV T,@AH                    ;在 T 寄存器中临时保存 Den16
MOV ACC,@Num16 << 16         ;AH = Num16, AL = 0
ABSTC ACC                    ;取绝对值, TC = sign 异或 TC
MOVU ACC,@AH                 ;AH = 0, AL = Num16
RPT #15                      ;重复操作 16 次
||SUBCU @T                   ;用 Den16 条件减
MOV @Rem16,AH                ;保存余数到 Rem16 中
MOV ACC,@AL << 16            ;AH = Quot16, AL = 0
NEGTC ACC                    ;若 TC = 1, 则求负
MOV @Quot16,AH               ;保存商到 Quot16

```

ADD ACC, #16bit<<#0...15

移位后的立即数与累加器相加

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD ACC,#16bit<< #0...15	1111 1111 0001 SHFT CCCC CCCC CCCC CCCC	×	-	1

操作数: ACC 累加器

#16bit 16 位立即数

#0...15 移位位数 (未指定时, 默认为 "<<#0")

描述: 左移后的 16 位立即数与 ACC 累加器相加, 结果保存到 ACC。若符号扩展模式使能 ($SXM = 1$), 则移位时为符号扩展; 否则 ($SXM = 0$) 移位时为 0 扩展, 最低位填 0:

```

if(SXM = 1)                // 使能符号扩展模式
    ACC = ACC+S:16bit<<移位位数;
else                        // 禁止符号扩展模式
    ACC = ACC+0:16bit<<移位位数;

```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0

C 若加操作产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。

V 若发生溢出, 则 V 置 1; 否则 V 不受影响。

OVC 若 OVM = 0, 当操作产生正的溢出时, 计算器值增加; 产生负的

溢出时，计数器值减小；若 $OVM = 1$ ，操作不影响计数器。

SXM 若符号扩展模式置位，在作加法之前 16 位立即数进行符号扩展，否则零扩展。

OVM 若溢出模式置位，在操作发生溢出时，ACC 为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;有符号数计算 ACC = (VarB<<10) + (23<<6);
SETC SXM                ;使能符号扩展模式
MOV ACC,@VarB << #10    ;用 VarB 左移 10 位后的值装载 ACC
ADD ACC,#23 << #6       ;立即数 23 左移 6 位后加到 ACC
```

ADD ACC,loc16<<T

数加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD ACC,loc16<<T	0101 0110 0010 0011 0000 0000 LLLL LLLL	1	Y	N+1

操作数： ACC 累加器

loc16 寻址方式

T 被乘数寄存器的高 16 位，XT (31:16)

描述： “loc16” 地址单元的 16 位数左移后加到 ACC 累加器中。由 T 寄存器的最低 4 位 $T(3:0) = 0...15$ 指定移位位数，忽略 T 寄存器的高位。若符号扩展模式使能 ($SXM = 1$)，则移位时进行符号扩展，否则零扩展 ($SXM = 0$)，最低位填 0：

```
if(SXM = 1)                //符号扩展模式使能
    ACC = ACC+S:[loc16]<<T(3:0);
else                        //符号扩展模式禁止
    ACC = ACC+S:[loc16]<<T(3:0);
```

标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。

V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。

OVC 若禁止溢出模式 ($OVM = 0$)，当操作产生正溢出时，计算器值增加；产生负溢出时，计数器值减小；若溢出模式使能 ($OVM = 1$)，操作不影响计数器。

SXM 若符号扩展模式置位，“loc16” 地址单元的 16 位操作数在加之前将符号扩展；否则作零扩展。

OVM 若溢出模式置位，在操作发生溢出时，ACC 的值饱和为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。

重复性： 若重复操作，则执行 N+1 次指令。Z、N、C 的状态为最后的结果。若中

间过程发生溢出，则 V 标志位将置 1。若禁止溢出模式，则 OVC 计数中间溢出。

例：

```
;有符号数计算 ACC = (VarA << SB) + (VarB << SB)
SETC SXM                ;使能符号扩展模式
MOV T,@SA                ;用 SA 移位后的值装载 T
MOV ACC,@VarA << T       ;在 ACC 中装载 VarA 移位后的值
MOV T,@SB                ;用 SB 移位后的值装载 T
ADD ACC,@VarB << T       ;把 VarB 移位后的值加到 ACC
```

ADD ACC,loc16<<#0...16

加数到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD ACC,loc16<<#0	1000 0001 LLLL LLLL	1	Y	N+1
ADD ACC,loc16<<#1...15	0101 0110 0000 0000 0000 SHFT LLLL LLLL	1	Y	N+1
ADD ACC,loc16<<#16	0000 0101 LLLL LLLL	X	Y	N+1
ADD ACC,loc16<<0...15	1010 SHFT LLLL LLLL	0	-	N+1

操作数： ACC 累加器
loc16 寻址方式

#0...16 移位位数（没有指定值时，默认为“<<#0”）

描述：“loc16”地址单元的 16 位数左移后加到 ACC 累加器中。若符号扩展模式使能（SXM = 1），则移位时进行符号扩展，否则进行 0 扩展（SXM = 0），最低位填 0：

```
if(SXM = 1) //符号扩展模式使能
    ACC = ACC+S:[loc16] << 移位位数;
else //符号扩展模式禁止
    ACC = ACC+S:[loc16] << 移位位数;
```

标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若指令执行产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。

V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。

OVC 若禁止溢出模式（OVM = 0），当操作产生正的溢出时，计算器值增加；产生负溢出时，计数器值减小；若使能溢出模式（OVM = 1），操作不影响计数器。

SXM 若符号扩展模式置位，“loc16”地址单元的 16 位操作数在加之前将符号扩展，否则进行 0 扩展。

OVM 若溢出模式置位，在指令执行时发生溢出，ACC 的值饱和为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。

重复性： 若操作重复，则执行 N+1 次指令。Z、N、C 的状态为最后的结果。若中

间过程发生溢出，则 V 标志位将置 1。若溢出模式禁止，则 OVC 计数中间溢出。若操作不可以重复，则指令仅执行一次。

例：

```
;有符号数计算 ACC = VarA << 10 + VarB << 6;
SET     SXM                ;使能符号扩展模式
MOV ACC,@VarA<<#10        ;VarA左移10位后装载到VarA
ADD ACC,@VarB<<#6         ;VarB左移6位后加到ACC
```

ADD AX,loc16 数加到 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD AX,loc16	1001 010A LLLL LLLL	×	-	1

操作数： AX 累加器高 (AH) 或累加器低 (AL) 寄存器
loc16 寻址方式

描述： 加“loc16”地址单元的内容到指定的 AX 寄存器 (AH 或 AL)，并将结果保存到 AX 寄存器：

$AX = AX + [loc16];$

标志与模式： N 若 AX 的第 15 位为 1，则负数标志位 N 置 1，否则负标志位 N 清 0。
Z 若 AX 的值为 0，则零标志位 Z 置 1，否则零标志位 Z 清 0。
C 若加操作产生进位，则进位标志位 C 置 1，否则进位标志位 C 清 0。
V 若发生溢出，则溢出标志位 V 置 1，否则 V 不受影响。若结果在正方向超过了最大正数 (0x7FFF)，则发生了有符号正溢出；若结果在负方向超过了最小负数 (0x8000)，则发生了有符号负溢出。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;VarA 和 VarB 相加，并保存到 VarC 中
MOV AL,@VarA                ;用 VarA 的值装载 AL
ADD AL,@VarB                ;加 VarB 的值到 AL
MOV @VarC,AL                ;结果保存到 VarC
```

ADD loc16,AX 加 AX 到指定地址

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD loc16, AX	0111 001A LLLL LLLL	×	-	1

操作数： loc16 寻址方式
AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述： 指定的 AX 寄存器 (AH 或 AL) 的内容与“loc16”地址单元的内容相加，结果保存到“loc16”地址单元中：

$[loc16] = [loc16] + AX;$

标志与模式： N 测试[loc16]是否为负数，若[loc16]的第 15 位为 1，则负数标志位

N 置 1，否则负标志位 N 清 0。

Z 测试[loc16]是否为 0，若操作导致[loc16] = 0，则零标志位置 Z 为 1，否则零标志位 Z 清 0。

C 若产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。

V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。若结果在正方向超过了最大正数 (0x7FFF)；则发生了有符号正溢出；若结果在负方向超过了最小负数 (0x8000)，则发生了有符号负溢出。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;加 VarA 中的内容到指定的 AR0:

MOV AL,@VarA

;用 VarA 的内容装载 AL

ADD @AR0,AL

;AR0 = AR0 + AL

;加 VarB 到 VarC:

MOV AH,@VarB

;用 VarB 的内容装载 AH

ADD @VarC,AH

;VarC = VarC + AH

ADD loc16,#16bitSigned

有符号立即数加到指定地址单元中

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADD loc16,#16bitSigned	0000 1000 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	1

操作数： loc16 寻址方式

#16bitSigned 16 位有符号立即数

描述： 有符号 16 位立即数与“loc16”地址单元的有符号 16 位数相加，16 位结果保存到“loc16”地址单元中：

[loc16] = [loc16]+16bitSigned;

标志与模式： N 若[loc16]的 15 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若[loc16]的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。

V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;计算 VarA = VarA + 10

,VarB = VarB - 3

ADD @VarA,#10

;VarA = VarA + 10

ADD @VarB,#-3

;VarB = VarB - 3

ADDB ACC, #8bit

8 位立即数加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDB ACC,#8bit	0000 1001 CCCC CCCC	×	-	1

操作数: ACC 累加器
8bit 8 位无符号立即数
描述: 8 位立即数零扩展后加到 ACC 累加器:

$ACC = ACC + 0:8bit;$

标志与模式: N 若 ACC 的 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
OVC 若禁止溢出模式 (OVM = 0), 当操作产生正溢出时, 计数器值增加; 产生负溢出时, 计数器值减小; 若使能溢出模式 (OVM = 1), 操作不影响计数器。
OVM 若溢出模式置位, 在操作发生溢出时, ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 增加 32 位地址 VarA 的内容
MOVL  ACC, @VarA          ; 用 VarA 的值装载 ACC
ADDB  ACC, #1              ; ACC 加 1
MOVL  @VarA, ACC           ; 把结果存回 VarA
```

ADDB AX, #8bitSigned

有符号 8 位立即数加到 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDB AX, #8bitSigned	1001 110A CCCC CCCC	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
#8bitSigned 8 位有符号立即数 (-128~+127)
描述: 8 位立即数符号扩展后加到指定的 AX 寄存器 (AH 或 AL), 并把结果保存到 AX 寄存器:

$AX = AX + S:8bit。$

标志与模式: N 测试 AX 是否为负数, 若 AX 的 15 位为 1, 则负数标志位 N 置 1, 否则负标志位 N 清 0。
Z 若 AX 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。若结果在正方向超过了最大正数 (0x7FFF), 则发生了有符号正溢出; 若结果在负方向超过了最小负数 (0x8000), 则发生了有符号负溢出。
重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;VarA 加 2, VarB 减 3
MOV  AL,@VarA      ;用 VarA 的值装载 AL
ADDB AL,#2          ;AL 加 0x0002
MOV  @VarA,AL       ;结果保存到 VarA 中
MOV  AL,@VarB       ;用 VarB 的值装载 AL
ADDB AL,#-3         ;AL 加 0xFFFD (-3)
MOV  @VarB,AL       ;结果保存到 VarB 中

```

ADDB SP,#7bit

7 位立即数加到堆栈指针上

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDB SP,#7bit	1111 1110 0CCC CCCC	×	-	1

操作数: SP 堆栈指针
#7bit 7 位无符号立即数

描述: 7 位无符号立即数加到 SP, 结果保存到 SP 中:

$$SP = SP + 0:7bit。$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

FuncA:                                ;堆栈中有局部变量的函数
    ADDB SP,#N                        ;在堆栈中为局部变量保留 N 个 16 位字的空间
    .
    .
    .
    SUBB SP,#N                        ;处理保留的堆栈空间
    LRETR                            ;从函数中返回

```

ADDB XARn,#7bit

7 位立即数加到辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDB XARn,#7bit	1101 1nnn 0CCC CCCC	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器

描述: 7 位无符号立即数加到 XARn, 结果保存到 XARn 中:

$$XARn = XARn + 0:7bit;$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

MOVL XAR1,#VarA      ;用 VarA 的地址初始化 XAR1
MOVL XAR2,*XAR1       ;用 VarA 的内容装载 XAR2
ADDB XAR2,#10h        ;XAR2 = VarA + 0x10

```


ADDCL ACC,loc32

加 32 位数和进位位到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDCL ACC,loc32	0101 0110 0100 0000 xxxx xxxx LLLL LLLL	1	-	1

操作数: ACC 累加器
loc32 寻址方式

描述: 加“loc32”寻址方式指向地址的 32 位数和进位位到 ACC 累加器:

$ACC = ACC + [loc32] + C;$

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。

V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。

OVC 若 OVM = 0, 操作产生正溢出时, 计算器值增加; 产生负溢出时, 计数器值减小; 若 OVM = 1, 操作不影响计数器。

OVM 若溢出模式置位, 则操作发生溢出时, ACC 的值饱和为 (0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;两个 64 位数相加 (VarA 和 VarB), 结果保存到 VarC

MOVL ACC, @VarA+0 ;用 VarA 的低 32 位数装载 ACC

ADDUL ACC, @VarB+0 ;加 VarB 的低 32 位数到 ACC

MOVL @VarC+0, ACC ;保存结果的低 32 位到 VarC

MOVL ACC, @VarA+2 ;用 VarA 的高 32 位数装载 ACC

ADDCL ACC, @VarB+2 ;加 VarB 的高 32 位数到 ACC

MOVL @VarC+2, ACC ;保存结果的高 32 位到 VarC

ADDCU ACC,loc16

无符号数和进位位加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDCU ACC,loc16	0000 1100 LLLL LLLL	X	-	1

操作数: ACC 累加器
loc16 寻址方式

描述: “loc16”地址单元的 16 位数零扩展后和进位位一起加到 ACC 累加器:

$ACC = ACC + 0: [loc16] + C;$

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。

V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。

OVC 若 OVM = 0, 操作产生正溢出时, 计算器值增加; 产生负溢出时,

计数器值减小；若 $OVM = 1$ ，操作不影响计数器。

OVM 若溢出模式置位，则操作发生溢出时，ACC 的值饱和为 $(0x7FFFFFFF)$ 或最小负数 $(0x80000000)$ 。

重复性： 本指令不可以重复。若指令跟在 **RPT** 指令之后，它将复位重复计数器 **RPTC**，并且只执行一次。

例：

；16 位加法实现 3 个 32 位无符号变量加

```
MOVU  ACC,@VarA_low      ;AH = 0, AL = VarA 低
ADD   ACC,@VarA_high << 16 ;AH = VarA 高, AL = VarA 低
ADDU  ACC,@VarB_low       ;ACC = ACC + 0:VarB 低
ADD   ACC,@VarB_high << 16 ;ACC = ACC + VarB 高左移 16 位
ADDCU ACC,@VarC_low       ;ACC = ACC + VarC 低 + 进位位
ADD   ACC,@VarC_high << 16 ;ACC = ACC + VarC 高左移 16 位
```

ADDL ACC,loc32

32 位数加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDL ACC,loc32	0000 0111 LLLL LLLL	×	Y	N+1

操作数： ACC 累加器
loc32 寻址方式

描述： “loc32” 地址单元中的 32 位数加到 ACC 累加器

$ACC = ACC + [loc32];$

标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。

V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。

OVC 若 $OVM = 0$ ，当操作产生正溢出时，计算器值增加；产生负溢出时，计数器值减小；若 $OVM = 1$ ，操作不影响计数器。

OVM 若溢出模式置位，则操作发生溢出时，ACC 的值饱和为 $(0x7FFFFFFF)$ 或最小负数 $(0x80000000)$ 。

重复性： 若重复操作，则执行 N+1 次指令。Z、N、C 的状态为最后的结果。若中间过程发生溢出，则 V 标志位将置 1。若禁止溢出模式，则 OVC 计数中间溢出。若操作不可以重复，则指令将仅执行一次。

例：

；计算 32 位数 $VarC = VarA + VarB$

```
MOVL  ACC,@VarA      ;用 VarA 的值装载 ACC
ADDL  ACC,@VarB       ;加 VarB 的值到 ACC
MOVL  @VarC,ACC       ;结果保存到 VarC
```

ADDL ACC,P<<PM

移位后的 P 值加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDL ACC,P<<PM	0001 0000 1010 1100	×	Y	N+1

注意：本指令是使用 loc16 = @T 寻址方式的“MOVA T, loc16”操作的别名。

- 操作数： ACC 累加器
P 乘积寄存器
<<PM 乘积移位方式
- 描述： 加 P 寄存器移位后的值到 ACC 累加器，移位方式由乘积移位方式 (PM) 指定：

$$ACC = ACC + P \ll PM$$
- 标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。
 Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。
 C 若产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。
 V 若发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。
 OVC 若 OVM = 0，当操作产生正溢出时，计算器值增加；产生负溢出时，计数器值减小；若 OVM = 1，操作不影响计数器。
 OVM 若溢出模式置位，在操作发生溢出时，ACC 的值饱和为 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 PM PM 值设定乘积寄存器进行输出操作时的移位方式。若乘积移位位数为正（逻辑左移操作），则最低位填 0；若乘积移位位数为负（逻辑右移操作），则高位进行符号扩展。
- 重复性： 本指令可以重复。将执行 N+1 次指令，Z、N、C 的状态为最后结果。若发生溢出，则溢出标志位 V 置 1。若溢出模式禁止，则 OVC 标志将计数。

例：

```

;计算 Y = ((M×X >> 4) + (B << 11)) >> 10; Y, M, X, B 是 Q15 格式的数据
SPM -4          ;设置乘积右移 4 位
SETC    SXM     ;符号扩展方式使能
MOV     T,@M    ;T = M
MPY     P,T,@X  ;P = M × X
MOV     ACC,@B << 11 ;ACC = S:B << 11
ADDL    ACC,P << PM ;ACC = (M×X >> 4) + (S:B << 11)
MOVH    @Y,ACC << 5 ;保存 ACC 左移 5 位后的结果到 Y 中

```

ADDL loc32,ACC

加累加器的值到指定地址

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDL loc32,ACC	0101 0110 0000 0001 0000 0000 LLLL LLLL	1	-	1

- 操作数： ACC 累加器
loc32 寻址方式
- 描述： ACC 累加器的值加到“loc32”寻址方式指定的地址单元中：

[loc32] = [loc32]+ACC;

这是一个读取—修改—写入操作过程。

- 标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
 Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
 V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
 OVC 若 OVM = 0, 当操作产生正溢出时, 计算器值增加; 产生负溢出时, 计数器减小; 若 OVM = 1, 操作不影响计数器。
 OVM 若溢出模式置位, 则操作发生溢出时, ACC 的值饱和为 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;32 位数 VarA 加 1
MOVB    ACC, #1           ;用 0x00000001 装载 ACC
ADDL     @VarA, ACC        ;VarA = VarA + ACC
```

ADDU ACC,loc16

无符号数加到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDU ACC,loc16	0000 1101 LLLL LLLL	×	Y	N+1

操作数: ACC 累加器
 loc16 寻址方式

描述: “loc16”地址单元的 16 位数加到 ACC 累加器, 寻址地址中的数在加之前被零扩展:

ACC = ACC+0: [loc16];

- 标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
 Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
 V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
 OVC 若 OVM = 0, 当操作产生正溢出时, 计算器值增加; 产生负溢出时, 计数器值减小; 若 OVM = 1, 操作不影响计数器。
 OVM 若溢出模式置位, 则操作发生溢出时, ACC 的值饱和为 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 重复性: 若重复操作, 则执行 N+1 次指令。Z、N、C 的状态为最后的结果。若中间过程发生溢出, 则 V 标志位将置 1。若溢出模式禁止, 则 OVC 计数中间溢出。若操作不可以重复, 则指令将仅执行一次。

例:

```
;16 位加法实现 3 个 32 位无符号变量加
MOVU     ACC,@VarA_low    ;AH = 0, AL = VarA 低
```



```

ADD    ACC,@VarAhigh << 16    ;AH = VarA 高, AL = VarA 低
ADDU   ACC,@VarBlow           ;ACC = ACC + 0:VarB 低
ADD    ACC,@VarBhigh << 16    ;ACC = ACC + VarB 高 << 16
ADDCU  ACC,@VarClow           ;ACC = ACC + VarC 低 + 进位位
ADD    ACC,@VarChigh << 16    ;ACC = ACC + VarC 高 << 16

```

ADDUL P,loc32

加 32 位无符号数到 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDUL P,loc32	0101 0110 0101 0111 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器

loc32 寻址方式

描述: 加“loc32”寻址方式指定的地址单元中的值到 P 寄存器。该加法被作为无符号 ADD 操作对待:

$$P = P + [loc32]; // \text{无符号加法}$$

注意: 有符号和无符号 32 位加法的区别主要在于溢出计数器 (OVC) 的处理。对于有符号 ADD, OVC 计数器检测正/负溢出。对于无符号 ADD, OVC 无符号计数器 (OVCU) 检测进位位。

标志与模式: N 若 P 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 P 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。

V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。

OVCU 当产生无符号进位时, 溢出计数器增计数。OVM 模式不影响 OVCU。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 作 64 位数的加法 VarA + VarB, 结果保存到 VarC 中
MOVL   P,@VarA+0           ;用 VarA 的低 32 位装载 P
MOVL   ACC,@VarA+2         ;用 VarA 的高 32 位装载 ACC
ADDUL  P,@VarB+0           ;将无符号数 VarB 的低 32 位加到 P
ADDCL  ACC,@VarB+2         ;加 VarB 的高 32 位到 ACC
MOVL   @VarC+0,P           ;保存结果的低 32 位到 VarC
MOVL   @VarC+2,ACC         ;保存结果的高 32 位到 VarC

```

ADDUL ACC,loc32

加 32 位无符号数到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADDUL ACC,loc32	0101 0110 0101 0011 xxxx xxxx LLLL LLLL	1	Y	N+1

操作数: ACC 累加器

loc32 寻址方式

描述: 将“loc32”地址单元中的 32 位数加到 ACC 累加器:

$ACC = ACC + [loc32];$ //无符号数加法

注意: 有符号和无符号 32 位加法的区别主要在于溢出计数器 (OVC) 的处理。对于有符号 ADD, OVC 计数器检测正/负溢出。对于无符号 ADD, OVC 无符号计数器 (OVCU) 检测进位位。

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
 Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
 V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
 OVCU 当操作产生无符号进位时, 溢出计数器增计数。OVM 模式不影响 OVCU。

重复性: 若重复操作, 则执行 N+1 次指令。Z、N、C 的状态为最后的结果。若中间过程发生溢出, 则 V 标志位将置 1。OVCU 标志位计数中间进位。

例:

```
;两个 64 位数相加 (VarA 和 VarB), 结果保存在 VarC:
MOVL    ACC, @VarA+0           ;用 VarA 的低 32 位装载 ACC
ADDUL    ACC, @VarB+0           ;VarB 的低 32 位数加到 ACC
MOVL    @VarC+0, ACC           ;保存结果的低 32 位到 VarC
MOVL    ACC, @VarA+2           ;用 VarA 的高 32 位装载 ACC
ADDCL    ACC, @VarB+2           ;VarB 的高 32 位和进位位加到 ACC
MOVL    @VarC+2, ACC           ;保存结果的高 32 位到 VarC
```

ADRK #8bit

加到当前辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ADRK #8bit	1111 1100 1111 1111	×	-	1

操作数: #8bit 8 位立即数

描述: 无符号 8 位立即数加到 ARP 指向的 XARn 寄存器:

$XAR(ARP) = XAR(ARP) + 0:8bit;$

标志与模式: 3 位 ARP 指向当前辅助寄存器, XAR0~XAR7。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
TableA: .word 0x1111
        .word 0x2222
        .word 0x3333
        .word 0x4444

FuncA:
MOVL    AR1, #TableA           ;初始化 XAR1 指针
MOVZ    AR2, *XAR1             ;用 XAR1 (0x1111) 指向的 16 位数装载到
                                ;AR2, 设置 ARP = 1
```

ADRK #2 ;XAR1 加 2
MOVZ AR3, *XAR1 ;用 XAR1 (0x3333) 指向的 16 位数装载到 AR3

AND ACC,#16bit<<#0...16 位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND ACC,#16bit<<#0...15	0011 1110 0000 SHFT CCCC CCCC CCCC CCCC	1	-	1
AND ACC,#16bit<<#16	0101 0110 0000 1000 CCCC CCCC CCCC CCCC	1	-	1

操作数: ACC 累加器

#16bit 16 位立即数

#0...16 移位位数 (未指定时, 默认为“<<#0”)

描述: ACC 累加器和 16 位无符号立即数左移后进行“与”操作。在“与”操作之前, 该立即数零扩展, 最低位填 0。结果保存在 ACC 累加器中:

$ACC = ACC \text{ AND } (0:16\text{bit} \ll \text{移位位数});$

标志与模式: N 测试 ACC 的值是否为负, 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 测试 ACC 的值是否为 0, 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;计算 32 位数 VarA = VarA 与 0x0FFFF000

MOVL ACC, @VarA ;把 VarA 的值装载到 ACC

AND ACC, #0xFFFF << 12 ;ACC 和 0x0FFFF000 进行“与”操作

MOVL @VarA, ACC ;结果保存到 VarA

AND ACC,loc16 位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND ACC,loc16	1000 1001 LLLL LLLL	1	Y	N+1

操作数: ACC 累加器

loc16 寻址方式

描述: ACC 累加器和“loc16”地址单元的内容零扩展后的值进行“与”AND 操作。结果保存在 ACC 累加器中:

$ACC = ACC \text{ AND } 0:[loc16];$

标志与模式: N 清标志位。

Z 测试 ACC 的值是否为 0, 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令可以重复。若指令用在 RPT 后, 则指令将执行 N+1 次。Z、N 的

状态为最后的结果。

例:

```
;计算 32 位数 VarA = VarA 与 0:VarB
MOVL    ACC,@VarA          ;把 VarA 的值装入 ACC
AND      ACC,@VarB          ;ACC 和 0:VarB 进行“与”操作
MOVL     @VarA,ACC          ;结果保存到 VarA
```

AND AX,loc16,#16bit

位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND AX,loc16,#16bit	1100 110A LLLL LLLL CCCC CCCC CCCC CCCC	×	-	1

操作数: ACC 累加器
loc16 寻址方式
#16bit 16 位立即数

描述: “loc16”地址单元的 16 位数和指定的 16 位立即数进行“与”操作。结果保存在 AX 寄存器中:

AX = [loc16] AND 16bit;

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 AX 的值为 0, 则零标志位 Z 置 1; 否则 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;若 VarA 的位 2 和位 7 不为 0, 则跳转
AND AL,@VarA,#0x0084          ;AL = VarA “与” 0x0084
SB Dest,NEQ                    ;若结果不为 0, 则跳转
;合并 VarA 的 0, 1, 2 位和 VarB 的 8, 9, 10 位, 结果保存到 VarC 的 0, 1, 2, 3, 4, 5 位:
AND L,@VarA,#0x0007           ;保留 VarA 的 0, 1, 2 位
AND AH,@VarB,#0x0700          ;保留 VarB 的 8, 9, 10 位
LSR AH,#5                      ;8, 9, 10 位移到 3, 4, 5 位
OR AL,@AH                      ;合并位
MOV @VarC,AL                   ;结果保存到 VarC
```

AND IER,#16bit

位“与”操作以禁止指定的 CPU 中断

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND IER,#16bit	0111 0110 0010 0110 CCCC CCCC CCCC CCCC	×	-	2

操作数: IER 中断使能寄存器
#16bit 16 位立即数 (0x0000~0xFFFF)

描述: 将 IER 寄存器和 16 位立即数进行“与”操作来禁止指定的中断。结果保存到 IER 寄存器:

IER = IER AND #16bit;

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; 仅仅禁止 INT1 和 INT6，不改变其他中断的状态

AND IER, #0xFFBE ; 禁止 INT1 和 INT6

AND IFR, #16bit

位“与”操作以清除悬挂的 CPU 中断

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND IFR, #16bit	0111 0110 0010 1111 CCCC CCCC CCCC CCCC	×	-	2

操作数：IFR 中断标志位寄存器

#16bit 16 位立即数 (0x0000~0xFFFF)

描述：将 IFR 寄存器和 16 位立即数进行“与”操作去清除指定的悬挂中断。“与”操作结果保存到 IFR 寄存器：

IFR = IFR AND #16bit;

注意：在硬件和指令同时修改中断标志位时，硬件中断比 CPU 指令的优先级高。

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; IFR 寄存器清 0，禁止所有悬挂中断

AND IFR, #0x0000 ; IFR 寄存器清 0

AND loc16, AX

位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND loc16, AX	1100 000A LLLL LLLL	×	-	1

操作数：loc16 寻址方式

AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述：“loc16”地址单元的数和指定 AX 寄存器进行“与”操作，结果保存在“loc16”地址单元中：

[loc16] = [loc16] AND AX;

这是一个读取—修改—写入操作过程。

标志与模式：N 若[loc16]的第 15 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若[loc16]的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```
;VarA 和 VarB 进行“与”操作, 结果保存在 VarB
MOV AL,@VarA           ;用 VarA 的值装载 AL
AND @VarB,AL           ;VarB = VarB “与” AL
```

AND AX, loc16

位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND AX,loc16	1100 111A LLLL LLLL	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
loc16 寻址方式

描述: 指定 AX 寄存器和“loc16”地址单元的 16 位数进行“与”操作。结果保存在 AX 中:

AX = AX AND [loc16];

标志与模式: N 测试 AX 的值是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 测试 AX 的值是否为 0, 若 AX 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;VarA 和 VarB 进行“与”操作, 若不为 0, 则跳转
MOV AL,@VarA           ;VarA 的值装入 AL
AND AL,@VarB           ;AL 和 VarB 进行“与”操作
SB Dest,NEQ            ;若不为 0, 则跳转
```

AND loc16,#16bitSigned

位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
AND loc16,#16bitSigned	0001 1000 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	1

操作数: loc16 寻址方式
#16bitSigned 16 位有符号立即数

描述: “loc16”地址单元中 16 位数和指定的 16 位立即数进行位“与”操作。结果保存“loc16”地址单元中:

[loc16] = [loc16]AND16bit;

标志与模式: N 若[loc16]的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若[loc16]的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; VarA 的第 3 位和第 11 位清 0
; VarA = VarA AND #~(1 << 3 | 1 << 11)
AND @VarA, #~(1 << 3 | 1 << 11) ; VarA 的第 3 位和第 11 位清 0
```

ANDB AX,#8bit

8 位数据位“与”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ANDB AX,#8bit	1001 000A CCCC CCCC	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
#8bit 8 位立即数

描述: 指定的 AX 寄存器 (AH 或 AL) 中的值和给定的 8 位无符号立即数零扩展后进行位“与”操作。结果保存在 AX 中:

```
AX = AX AND 0:8bit;
```

标志与模式: N 测试 AX 的值是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 测试 AX 的值是否为 0, 若 AX 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;加 VarA 到 VarB, 保留低字节, 结果保存在 VarC 中
MOV AL,@VarA ;VarA 的值装载到 AL 中
ADD AL,@VarB ;加 VarB 到 AL
ANDB AL,#0xFF ;AL 和 0x00FF 进行“与”操作
MOV @VarC,AL ;结果保存到 VarC
```

ASP

堆栈指针对齐

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASP	0111 0110 0001 1011	×	-	1

操作数: 无

描述: 确保堆栈指针 (SP) 对齐到偶地址处。若 SP 的最低位为 1, SP 指向奇地址, 则 SP 必须加 1 移到偶地址。当有校准记录时, SPA 置 1。ASP 指令发现 SP 已经指向了偶地址, 则 SP 保持不变, SPA 位清 0, 以表示没有发生校准操作。在任何情况, SPA 位的改变都是发生在流水线的译码阶段 2。

```
if(SP = odd)
SP = SP + 1;
SPA = 1;
else
SPA = 0;
```

若你想取消上一次使用 ASP 指令作的校准，则可以用 NASP 指令。

标志与模式：SPA 若指令执行前 SP 保持偶地址，则 SPA 置 1；否则 SPA 清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;在中断服务程序中堆栈指针的校准
;向量表
INTx: .long INTxService      ;INTx 中断向量
.
.
INTxService:
    ASP                      ;校准堆栈指针
    .
    .
    .
    NASP                    ;重新校准堆栈指针
    IRET                    ;从中断中返回

```

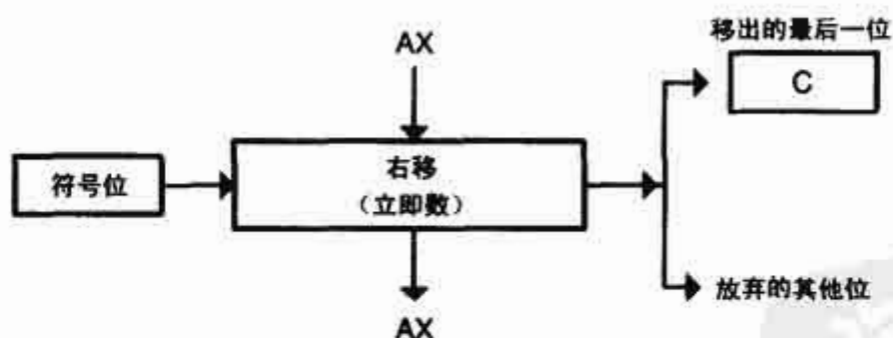
ASR AX,#1...16

算术右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASR AX,#1...16	1111 1111 101A SHFT	×	-	1

操作数：AX 累加器高（AH）或累加器低（AL）寄存器
#1...16 移位位数

描述：指定的 AX 寄存器（AH 或 AL）中的值算术右移给定的位数。在移位时，该数符号扩展，移出 AX 寄存器的最后一位保存在进位状态标志位中：



标志与模式：N 若 AX 的第 15 位为 1，则负标志位 N 置 1，否则负标志位 N 清 0。

Z 若 AX 的值为 0，则零标志位 Z 置 1，否则零标志位 Z 清 0。

C 移出 AH 或 AL 的最后一位保存在 C 中。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;计算有符号数 VarC = (VarA + VarB) 右移 2 位
MOV AL,@VarA      ;VarA 的值装载到 AL 中
ADD AL,@VarB      ;加 VarB 到 AL 中

```

```
ASR AL, #2      ;移位 2 位
MOV @VarC, AL   ;结果保存到 VarC
```

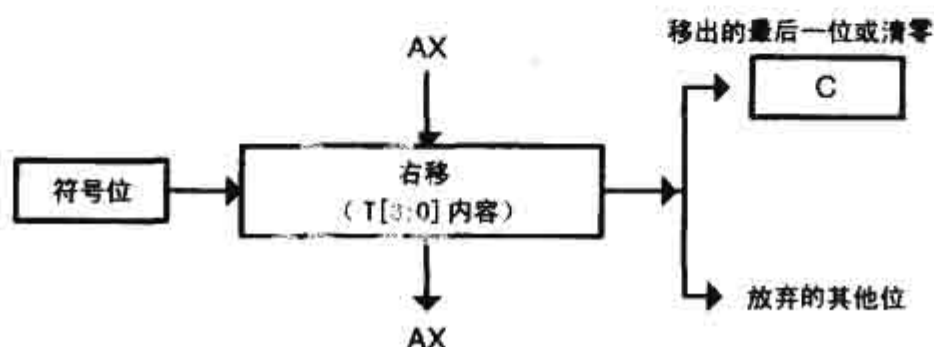
ASR AX,T

算术右移 T 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASR AX,T	1111 1111 0110 010A	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
 T 被乘数 (XT) 寄存器的高 16 位

描述: 指定的 AX 寄存器 (AH 或 AL) 中的值作算术右移, 由 T 寄存器的低 4 位 T (3:0) = 0...15 指定移位位数, T 的高位忽略。在移位时, 进行符号扩展。若 T (3:0) 寄存器指定的移位位数为 0, 则进位标志位 C 清 0; 否则移出 AX 寄存器的最后一位到进位标志位 C 中:



标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置 1, 否则 N 清 0。即使 T (3:0) 寄存器指定的移位位数为 0, 还是要测试 AH 或 AL 的值是否为负, 从而影响负标志位 N。
 Z 若 AX 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。即使 T (3:0) 寄存器指定的移位位数为 0, 也会测试 AH 或 AL 的值是否为 0, 从而影响 Z 标志位。
 C 若 T (3:0) 指定移位位数为 0, 则进位标志位 C 清 0; 否则移出 AH 或 AL 的最后一位到 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;计算有符号数 VarC = VarA >> VarB;
MOV T, @VarB      ;把 VarB 的值装入 T
MOV AL, @VarA     ;把 VarA 的值装入 AL
ASR AL, T         ;AL 移位 T 位
MOV @VarC, AL     ;结果保存在 VarC
```

ASR64 ACC:P,#1...16

64 位数的算术右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASR64 ACC:P,#1...16	0101 0110 1000 SHFT	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)

#1...16 移位位数

描述: 算术右移 ACC:P 寄存器组成的 64 位数, 右移位数由“移位位数”指定。在移位时, 最高位符号扩展, 移出的最后一位存到进位标志位 C 中:



标志与模式: N 若 ACC 累加器的第 31 位为 1, 则 ACC:P 为负, 负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC:P 组合的 64 位数的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 组合 64 位数移出的最后一位到进位标志位 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 64 位数 Var64 作算术右移 10 位
MOVL  ACC, @Var64+2      ; Var64 的高 32 位装入 ACC 中
MOVL  P, @Var64+0        ; Var64 的低 32 位装入 P 中
ASR64  ACC:P, #10        ; ACC:P 算术右移 10 位
MOVL  @Var64+2, ACC      ; 保存高 32 位结果到 Var64
MOVL  @Var64+0, P        ; 保存低 32 位结果到 Var64
  
```

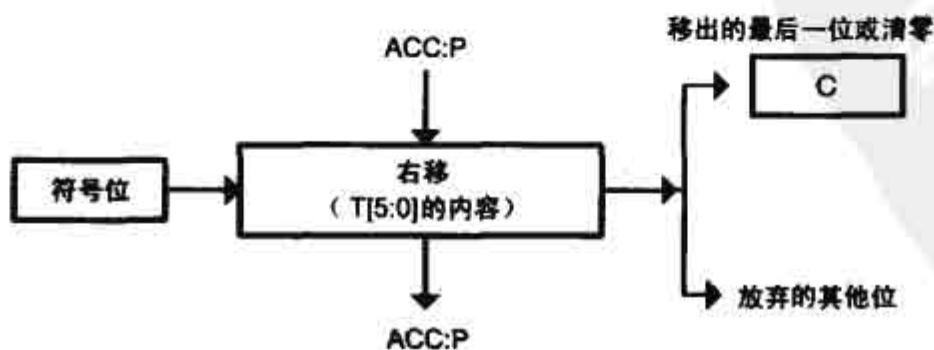
ASR64 ACC:P,T**64 位数的算术右移**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASR64 ACC:P,T	0101 0110 0010 1100	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)

T 被乘数寄存器 (XT) 的高 16 位

描述: 算术右移 ACC:P 寄存器组成的 64 位数, 由 T 寄存器的低 6 位 T (5:0) = 0...63 指定移位位数, 忽略 T 寄存器的高位。在移位时, 最高位符号扩展。若 T 指定的移位位数为 0, 则进位标志位 C 清 0, 否则移出 ACC:P 寄存器的最后位到进位标志位 C 中:



- 标志与模式: N 若 ACC 累加器的第 31 位为 1, 则 ACC:P 为负, 负标志位 N 置 1; 否则负标志位 N 清 0。
- Z 若 ACC:P 组合的 64 位数的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
- C 若 T[5:0] = 0, 则进位标志位 C 清 0; 否则 64 位数移出的最后一位到 C 中。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 64 位数 Var64 作算术右移 Var16 指定位数
MOVL    ACC, @Var64+2      ; Var64 的高 32 位装入 ACC 中
MOVL    P, @Var64+0        ; Var64 的低 32 位装入 P 中
MOV     T, @Var16          ; 移位位数 Var16 装入 T 中
ASR64   ACC:P, T           ; 算术右移 ACC:P T(5:0) 位
MOVL    @Var64+2, ACC      ; 保存结果的高 32 位到 Var64
MOVL    @Var64+0, P        ; 保存结果的低 32 位到 Var64

```

ASRL ACC,T

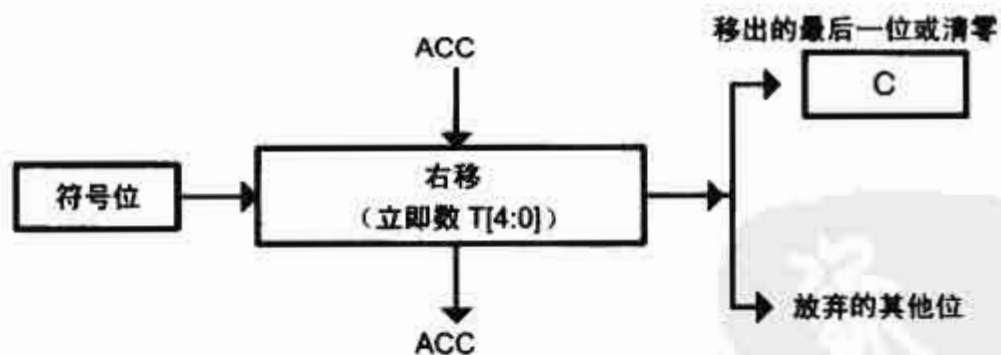
累加器算术右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ASRL ACC,T	0101 0110 0001 0000	1	-	1

操作数: ACC 累加器

T 被乘数寄存器 (XT) 的高 16 位

描述: 算术右移 ACC 寄存器中的数, 由 T 寄存器的低 5 位 T(4:0) = 0...31 指定移位位数, 忽略 T 寄存器的高位。在移位时, 最高位进行符号扩展。若 T 指定的移位位数为 0, 则进位标志位 C 清 0, 否则最后移出 ACC:P 寄存器的位到 C 中:



- 标志与模式: N 若 ACC 累加器的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。即使 T 寄存器指定的移位位数为 0, 还是要测试 ACC 累加器的值是否为负, 会影响 N 的状态。
- Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。即使 T 寄存器指定的移位位数为 0, 还是要测试 ACC 累加器的值是否为 0, 会影响 Z 的状态。
- C 若 T[4:0] = 0, 则进位标志位 C 清 0; 否则移出的最后一位到 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;VarA 算术右移 VarB 位
MOVL    ACC,@VarA          ;ACC = VarA
MOV      T,@VarB            ;T = VarB (移位位数)
ASRL     ACC,T              ;ACC 算术右移 T(4:0)位
MOVL     @VarA,ACC          ;结果保存到 VarA
  
```

B 16bitOffset,COND

条件跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
B 16bitOffset,COND	1111 1111 1110 COND CCCC CCCC CCCC CCCC	×	-	7/4

操作数: 16bitOffset 16 位有符号立即数 (-32768~+32767)

COND

条件代码

COND	语 法	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 和 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	小于	N = 1
0101	LEQ	小于或等于	Z = 1 或 N = 1
0110	HI	更高的	C = 1 和 Z = 0
0111	HIS, C	更高的或相同, 进位位置位	C = 1
1000	LO, NC	更低的, 进位位清 0	C = 0
1001	LOS	更低的或相同	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输入等于零	BIO = 0
1111	UNC	无条件	-

描述: 条件跳转。若条件为真, 则加一个有符号 16 位立即数到当前 PC, 发生跳转; 否则不跳转, 继续执行程序:

```

If (COND = 真) PC = PC + 有符号 16 位偏移量;
If (COND = 假) PC = PC + 2;
  
```

注意: If (COND = 真) 指令执行需花费 7 个周期。

If (COND = 假) 指令执行需花费 4 个周期。

标志与模式: V 若条件测试 V 标志位, 则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

BANZ 16bitOffset, ARn--

辅助寄存器不为 0, 则跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
BANZ 16bitOffset, ARn--	0101 0110 0001 0000 CCCC CCCC CCCC CCCC	1	-	1

操作数: 16bitOffset 16 位有符号立即数

ARn 辅助寄存器 XAR0~XAR7 的低 16 位

描述: 若指定的辅助寄存器的 16 位数不为 0, 则加有符号 16 位的偏移量到 PC 值。强迫控制程序跳到新的地址 (PC+16bitOffset)。16 位偏移量在加之前符号扩展到 22 位, 然后辅助寄存器中的值减 1。在比较中不使用辅助寄存器的高 16 位 (ARnH), 减后也不会影响它:

```
if( ARn != 0 )
PC = PC + 有符号的 16 位偏移量;
ARn = ARn - 1;
ARnH = 不改变;
```

注意: 若发生跳转, 指令需要花费 4 个周期, 若不发生跳转, 指令需要花费 2 个周期。

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;复制 Array1 到 Array2
; int32 Array1[N];
; int32 Array2[N];
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i]
MOVL    XAR2, #Array1      ; XAR2 指针指向 Array1
MOVL    XAR3, #Array2      ; XAR3 指针指向 Array2
MOV     @AR0, #(N-1)        ; 重复 loop 循环 N 次
Loop:
MOVL    ACC, *XAR2++        ; ACC = Array1[i]
MOVL    *XAR3++, ACC        ; Array2[i] = ACC
BANZ    Loop, AR0--         ; 若 AR0 != 0, 则跳到 LOOP, AR0--
```

BAR 16bitOffset, ARn, ARm, EQ/NEQ

辅助寄存器比较跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
BAR 16bitOffset, ARn, ARm, EQ	1000 1111 10nn nnnn CCCC CCCC CCCC CCCC	1	-	4/2
BAR 16bitOffset, ARn, ARm, NEQ	1000 1111 11nn nnnn CCCC CCCC CCCC CCCC	1	-	4/2

操作数: 16bitOffset 16 位有符号立即数偏移量 (−32768~+32767)
 ARn 辅助寄存器 XAR0~XAR7 的低 16 位
 ARm 辅助寄存器 XAR0~XAR7 的低 16 位

语 法	描 述	测 试 条 件
NEQ	不等于	$ARn \neq ARm$
EQ	等于	$ARn = ARm$

描述: 比较两个辅助寄存器 ARn 和 ARm 寄存器的 16 位数, 若条件为真, 则跳转; 否则不跳转, 继续执行程序;

If (测试条件为真) $PC = PC +$ 有符号 16 位偏移量;

If (测试条件为假) $PC = PC + 2$;

注意: If (测试条件 = 真) 指令花费 4 个周期

If (测试条件 = 假) 指令花费 2 个周期

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;字符串比较
MOVL  XAR2, #StringA      ;XAR2 指向字符串 A
MOVL  XAR3, #StringB      ;XAR3 指向字符串 B
MOV   @AR4, #0            ;AR4 = 0
Loop:
MOVZ  AR0, *XAR2+ +       ;AR0 = 字符串 A[i]
MOVZ  AR1, *XAR3+ +       ;AR1 = 字符串 B[i], i++
BAR   Exit, AR0, AR4, EQ   ;若字符串 A[i] = 0, 则退出
BAR   Loop, AR0, AR1, EQ   ;若字符串 A[i] = 字符串 B[i], 则跳到 LOOP
NotEqual:
Exit:                          ;字符串 A 和 B 一样

```

BF 16bitOffset,COND

快速跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
BF 16bitOffset,COND	0101 0110 1100 COND CCCC CCCC CCCC CCCC	1	—	4/4

操作数: 16bitOffset 16 位有符号立即数偏移量 (−32768~+32767)

COND

条件代码:

COND	语法:	描 述	测试标志位
0000	NEQ	不等于	$Z = 0$
0001	EQ	等于	$Z = 1$
0010	GT	大于	$Z = 0$ 和 $N = 0$

续表

COND	语法:	描 述	测试标志位
0011	GEQ	大于或等于	N = 0
0100	LT	小于	N = 1
0101	LEQ	小于或等于	Z = 1 或 N = 1
0110	HI	更高的	C = 1 和 Z = 0
0111	HIS,C	更高的或相同, 进位位置位	C = 1
1000	LO,NC	更低的, 进位位置清 0	C = 0
1001	LOS	更低的或相同	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输入等于零	BIO = 0
1111	UNC	无条件	-

描述: 快速条件跳转。若指定条件为真, 则加一个有符号 16 位立即数到当前 PC, 发生跳转; 否则不跳转, 继续执行程序:

If (COND = 真) PC = PC + 有符号 16 位偏移量 t;

If (COND = 假) PC = PC + 2;

注意: 快速跳转 (BF) 指令在 C28x 内核占用双倍的预取队列, 使指令周期数从 7 减少到 4:

If (COND = 真) 指令执行需花费 4 个周期。

If (COND = 假) 指令执行需花费 4 个周期。

标志与模式: V 若使用了条件来测试 V 标志位, 则 V 标志位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

C27MAP

置 M0M1MAP 位

语法选项	操作码	OBJ 模式	RPT	CYC
C27MAP	0101 0110 0011 1111	×	-	5

注意: 本指令是“CLRC M0M1MAP”指令的别名。

操作数: 无

描述: M0M1MAP 状态位清 0, 配置 M0 和 M1 存储器模块映射位为 C27x 兼容操作模式。存储器模块映射如下:

M0M1MAP 位	数据空间	程序空间
0 (C27x)	M0:0x000~0x3FF M1:0x400~0x7FF	M0:0x400~0x7FF M1:0x000~0x3FF
1 (C28x/C2xLP)	M0:0x000~0x3FF M1:0x400~0x7FF	

注意：当指令执行时，刷新流水线。

标志与模式：M0M1MAP M0M1MAP 位清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

；设置器件模式从复位模式到 C27x 兼容模式

Reset：

```
C27OBJ          ;使能 C27x 模式
C28ADDR         ;使能 C27x/C28x 寻址方式
.c28_amode      ;告诉汇编器我们使用的是 C27x/C28x 寻址
C27MAP          ;使能 M0 和 M1 模块的 C27x 映射
.
.
```

C27OBJ

OBJMODE 位清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
C27OBJ	0101 0110 0011 0110	X	-	5

注意：本指令是“CLRC OBJMODE”操作的别名。

操作数：无

描述：状态寄存器 ST1 中的 OBJMODE 状态位清 0，配置器件到可执行 C27x 目标代码模式。该模式是处理器复位后的默认模式。

注意：当指令执行时，刷新流水线。

标志与模式：OBJMODE 位清 0

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

；设置器件模式从复位到 C27x 模式

Reset：

```
C27OBJ          ;使能 C27x 目标模式
C28ADDR         ;使能 C27x/C28x 寻址方式
.c28_amode      ;告诉汇编器我们正处于 C27x/C28x 寻址方式
C27MAP          ;使能 M0 和 M1 模块的 C27x 映射
.
.
```

C28ADDR

AMODE 状态位清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
C28ADDR	0101 0110 0001 0110	X	-	1

注意：本指令是“CLRC AMODE”操作的别名。

操作数： 无
描述： 状态寄存器 ST1 中的 AMODE 状态位清 0，使器件进入 C27x/C28x 模式。

注意：本指令不刷新流水线操作。

标志与模式： AMODE AMODE 位清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;执行操作 "VarC = VarA + VarB"，用 C2xLP 语法书写
LPADDR                ;进入 C2xLP 兼容的寻址方式
.lp_amode              ;告诉汇编器我们正处于 C2xLP 模式
LDP #VarA              ;初始化 DP（仅低 64K）
LACL VarA              ;ACC = VarA (ACC high = 0)
ADDS VarB              ;ACC = ACC + VarB（无符号）
SACL VarC              ;结果保存到 VarC
C28ADDR               ;返回 C28x 寻址方式
.c28_amode             ;告诉汇编器我们正处于 C28x 模式
```

C28MAP

设置 M0M1MAP 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
C28MAP	0101 0110 0001 1010	×	-	5

注意：本指令是“SETC M0M1MAP”指令的别名。

操作数： 无

描述： 置位状态寄存器 ST1 中的 M0M1MAP 状态位，配置 M0 和 M1 存储器模块映射位为 C28x 兼容操作模式。存储器模块映射如下：

M0M1MAP 位	数 据 空 间	程 序 空 间
0 (C27x)	M0:0x000~0x3FF M1:0x400~0x7FF	M0:0x400~0x7FF M1:0x000~0x3FF
1 (C28x/C2xLP)	M0:0x000~0x3FF M1:0x400~0x7FF	

注意：当指令执行时，重新刷新流水线。

标志与模式： M0M1MAP 置 1M0M1MAP 位。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;设置器件模式从复位模式到 C28x 模式
```

Reset：

```
C28OBJ                ;使能 C28x 目标模式
C28ADDR               ;使能 C28x 寻址方式
.c28_amode            ;告诉汇编器我们处于 C28x 寻址方式
```

C28MAP ;使能 M0 和 M1 的 C28x 映射

C28OBJ

置位 OBJMODE 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
C28OBJ	0101 0110 0001 1111	×	-	5

注意：本指令是“SETC OBJMODE”操作的别名。

操作数： 无

描述： 置位 OBJMODE 的状态位，配置器件到可执行 C28x 目标代码模式（支持 C2xLP 源程序）。

标志与模式： OBJMODE 置位 OBJMODE 位

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;设置器件模式从复位到 C28x 模式

Reset:

C28OBJ ;使能 C28x 目标模式

C28ADDR ;使能 C27x/C28x 寻址方式

.c28_amode ;告诉汇编器我们正处于 C27x/C28x 寻址方式

C28MAP ;使能 M0 和 M1 模块的 C28x 映射

CLRC AMODE

AMODE 位清 0

语法选项	操作码	OBJ 模式	RPT	CYC
CLRC AMODE	0101 0110 0001 0110	×	-	1

操作数： AMODE 状态位

描述： 状态寄存器 ST1 中的 AMODE 状态位清 0，使能 C27x/C28x 寻址模式。

注意：本指令不刷新流水线

标志与模式： OBJMODE OBJMODE 位清 0

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;执行操作 "VarC = VarA + VarB"，用 C2xLP 语法书写

SETC AMODE ;进入 C2xLP 寻址兼容模式

.lp_amode ;告诉汇编器我们处于 C2xLP 模式

LDP #VarA ;初始化 DP (仅低 64K)

LACL VarA ;ACC = VarA (ACC high = 0)

ADDS VarB ;ACC = ACC + VarB (unsigned)

SACL VarC ;结果保存到 VarC
 CLRC AMODE ;返回到 C28x 寻址方式
 .c28_amode ;告诉汇编器我们处于 C28x 模式

CLRC M0M1MAP**M0M1MAP 位清 0**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CLRC M0M1MAP	0101 0110 0011 1111	×	-	5

操作数: M0M1MAP 状态位

描述: 状态寄存器 ST1 中的 M0M1MAP 状态位清 0, 配置 M0 和 M1 存储器模块映射为 C27x 兼容操作模式。存储器模块映射如下:

M0M1MAP 位	数 据 空 间	程 序 空 间
0 (C27x)	M0:0x000~0x3FF M1:0x400~0x7FF	M0:0x400~0x7FF M1:0x000~0x3FF
1 (C28x/C2xLP)	M0:0x000~0x3FF M1:0x400~0x7FF	

注意: 当指令执行时, 刷新流水线。该位提供了用户从 C27x 移植的兼容性。当用户操作在 C28x 模式和 C2xLP 模式时, M0M1MAP 位应该保留置 1。

标志与模式: M0M1MAP M0M1MAP 位清 0。

例:

;设置器件模式从复位模式到 C27x 目标兼容模式

Reset:

CLRC OBJMODE ;使能 C27x 目标模式
 CLRC AMODE ;使能 C27x/C28x 寻址方式
 .c28_amode ;告诉汇编器我们处于 C27x/C28x 寻址方式
 CLRC M0M1MAP ;使能 M0 和 M1 的 C27x 映射
 .
 .

CLRC OBJMODE**OBJMODE 位清 0**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CLRC OBJMODE	0101 0110 0011 0110	×	-	5

操作数: OBJMODE 状态位

描述: OBJMODE 状态位清 0, 使器件执行 C27x 目标代码使能。

注意: 当指令执行时, 会刷新流水线。

标志与模式: OBJMODE OBJMODE 位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 设置器件从复位模式到 C27x 目标兼容模式

Reset:

```
CLRC OBJMODE      ;使能 C27x 目标模式
CLRC AMODE         ;使能 C27x/C28x 寻址方式
.c28_amode         ;告诉汇编器我们处于 C27x/C28x 寻址方式
CLRC M0M1MAP       ;使能 M0 和 M1 的 C27x 映射
.
.
```

CLRC OVC

溢出计数器清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CLRC OVC	0101 0110 0101 1100	1	-	1

注意: 本指令是“ZAP OVC”操作的别名。

操作数: OVC 状态寄存器 0 (ST0) 中的溢出计数器

描述: 6 位溢出计数器清 0。

标志与模式: OBJMODE OBJMODE 位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;计算 VarD = sat (VarA + VarB + VarC)
CLRC OVC           ;溢出计数器清 0
MOVL ACC,@VarA     ;ACC = VarA
ADDL ACC,@VarB     ;ACC = ACC + VarB
ADDL ACC,@VarC     ;ACC = ACC + VarC
SAT ACC            ;若 OVC != 0, 则 ACC 为饱和值
MOVL @VarD, ACC    ;保存饱和值到 VarD
```

CLRC XF

XF 状态位清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CLRC XF	0101 0110 0001 1011	X	-	1

操作数: XF XF 状态位和输出信号

描述: XF 状态位清 0 并将相应的输出信号拉为低电平。

标志与模式: XF XF 状态位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;若没有发生跳转, 则设置 XF 信号输出一个脉冲
MOV AL,@VarA       ;把 VarA 的值装载到 AL
SB Dest,NEQ        ;ACC = VarA
SETC XF            ;XF 位置 1 和信号为高
```

CLRC XF ;XF 位清 0 和信号为低

Dest:

CLRC MODE 状态位清 0

语法选项	操作码	OBJ 模式	RPT	CYC
CLRC mode	0010 1001 CCCC CCCC	×	—	1,2
CLRC SXM	0010 1001 0000 0001	×	—	1
CLRC OVM	0010 1001 0000 0010	×	—	1
CLRC TC	0010 1001 0000 0100	×	—	1
CLRC C	0010 1001 0000 1000	×	—	1
CLRC INTM	0010 1001 0001 0000	×	—	2
CLRC DBGM	0010 1001 0010 0000	×	—	2
CLRC PAGE0	0010 1001 0100 0000	×	—	1
CLRC VMAP	0010 1001 1000 0001	×	—	1

描述： 指定的状态位清 0。“mode”操作数是一个与状态位有关的屏蔽值。

“mode”位	状态寄存器	标 志 位	周 期 数
0	ST0	SXM	1
1	ST0	OVM	1
2	ST0	TC	1
3	ST0	C	1
4	ST1	INTM	2
5	ST1	DBGM	2
6	ST1	PAGE0	1
7	ST1	VMAP	1

注意：汇编器将会接收任何顺序的任意标志位。

标志与模式： SXM 任何指定位都可以用指令清 0。
 OVM
 TC
 C
 INTM
 DBGM
 PAGE0
 VMAP

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```

;修改标志位设置
SETC    INTM,DBGM           ;设置 INTM 和 DBGM 位为 1
CLRC    TC,C,SXM,OVM       ;清 TC, C, SXM, OVM 位为 0
CLRC    #0xFF              ;清所有位为 0
SETC    #0xFF              ;设置所有位为 1
SETC    C,SXM,TC,OVM       ;设置 TC, C, SXM, OVM 为 1
CLRC    DBGM,INTM          ;清 INTM 和 DBGM 位为 0

```

CMP AX,loc16 比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMP AX,loc16	0101 010A LLLL LLLL	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
loc16 寻址方式

描述: 指定 AX 寄存器中的值与 “loc16” 地址单元中的 16 位数进行比较。计算 $(AX - [loc16])$ 的结果, 设置相应的状态位。AX 寄存器和 “loc16” 地址单元的值都保持不变:

根据 $(AX - [loc16])$ 的结果设置标志:

标志与模式: N 若操作结果为负, 则负标志位 N 置 1, 否则负标志位 N 清 0。在指定结果符号时, 假定 CMP 指令为无限位精度。例如, 计算 $0x8000 - 0x0001$ 时, 若精度限定在 16 位, 则结果将会溢出, 得到正数 $0x7FFF$, N 清 0。然而如果认为 CMP 指令无限位精度, 则负标志位 N 会置 1, 以表示 $0x8000 - 0x0001$ 实际上得到一个负数。

Z 测试比较操作是否为 0, 若操作 $(AX - [loc16]) = 0$, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置 1。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;若 VarA 大于 VarB, 则跳转
MOV     AL,@VarA           ;把 VarA 的值装载到 AL
CMPB    AL,@VarB           ;根据 (AL - VarB) 的结果, 设置标志位
SB      Dest,HI            ;若 VarA 大于 VarB, 则跳转

```

CMP loc16,#16bitSigned 16 位有符号数比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMP loc16,#16bitSigned	0101 010A LLLL LLLL	×	-	1

操作数: loc16 寻址方式
#16bitSigned 16 位有符号立即数

描述: “loc16” 地址单元的 16 位数和一个有符号 16 位立即数进行比较。为了

完成比较, 计算 $[\text{loc16}] - \# 16\text{bitSigned}$), 设置相应的状态位。指定地址单元中的值都保持不变:

根据 $([\text{loc16}] - 16\text{bitSigned})$ 方法修改标志位;

- 标志与模式: N 若操作结果为负, 则负标志位 N 置 1; 否则负标志位 N 清 0。在指定结果符号时, 假定 CMP 指令为无限位精度。例如, 计算 $0\text{x}8000 - 0\text{x}0001$ 时, 若精度限定在 16 位, 则结果将会溢出, 得到正数 $0\text{x}7\text{FFF}$, N 清 0。然而由于认为 CMP 指令无限位精度, 则负标志位 N 会置 1, 以表示 $0\text{x}8000 - 0\text{x}0001$ 实际上得到一个负数。
- Z 测试比较操作是否为 0, 若操作 $([\text{loc16} - 16\text{bitSigned}]) = 0$, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
- C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置 1。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

注意: 本章的例都是假定器件已经工作在 C28x 模式 ($\text{OBJMODE} = 1, \text{AMODE} = 0$)。器件复位后, 要使其进入 C28x 模式, 必须通过执行指令 “C28OBJ” (或: SETC OBJMODE) 来设置 ST1 中的 OBJMODE 位。

例:

```
;计算: if( VarA > 20 ) VarA = 0;
CMP @VarA, #20           ;根据 (VarA - 20) 的值, 设置标志位
MOVB @VarA, #0, GT       ;若大于, 则 VarA = 0
```

CMP64 ACC:P

64 位数的比较

语法选项	操作码	OBJ 模式	RPT	CYC
CMP64 ACC:P	0101 0110 0101 1110	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)

描述: ACC:P 寄存器中的 64 位数与零比较, 设置相应的标志位:

```
if((V = 1) & (ACC(bit 31) = 1))
    N = 0;
else
    N = 1;
if((V = 1) & (ACC(bit 31) = 0))
    N = 1;
else
    N = 0;
if(ACC:P = 0x8000 0000 0000 0000)
    Z = 1;
else
    Z = 0;
V = 0;
```


注意：操作如下所示：

CMP64 ACC:P ; V 标志位清 0，实现 64 位比较
 CMP64 ACC:P ; 置位 Z、N 标志位，V = 0，条件跳转

标志与模式： N 测试 ACC 累加器的值，以判定 64 位 ACC:P 的值是否为负。当 ACC 为负时，CMP64 指令将考虑溢出标志位 V 的状态。例如，计算 0x8000 0000 - 0x0000 0001 的减法时，将会导致正溢出 (0x7FFF FFFF) 和标志位 V 置 1。由于 CMP64 指令要考虑溢出，它相应的会得到一个负数，而不是正数。若发现 ACC 为负，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC:P 组合的 64 位数为 0，则零标志位 Z 置 1；否则清 0。

V V 标志位的状态和 ACC 累加器的第 31 位一起决定 ACC:P 寄存器的值是否为负。根据操作结果 V 标志位清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
; 若 64 位数 VarA > 64 位数 VarB，则跳转
CMP64  ACC:P          ; V 标志位清 0
MOVL    P,@VarA+0      ; 用 VarA 的低 32 位装载 P 寄存器
MOVL    ACC,@VarA+2     ; 用 VarA 的高 32 位装载 P 寄存器
SUBUL    P,@VarB+0      ; 从 P 中减去无符号的 VarB 的低 32 位
SUBBL    ACC,@VarB+2     ; 从 ACC 中减去 VarB 的高 32 位和借位
CMP64    ACC:P          ; 根据 ACC:P，相应地设置 Z 和负标志位 N
SB       Dest,GT        ; 若 VarA > VarB，则跳转
```

CMPB AX,#8bit

8 位数比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMPB AX,#8bit	0101 001A CCCC CCCC	×	-	1

操作数： AX 累加器高 (AH) 或累加器低 (AL) 寄存器

#8bit 8 位立即数

描述： 指定的 AX 寄存器中的值和 8 位无符号立即数零扩展后进行比较。计算 (AX - 0:8bit) 的结果，设置相应的状态位。AX 寄存器的值保持不变。

标志与模式： N 若操作结果为负，则负标志位 N 置 1；否则负标志位 N 清 0。在指定结果符号时，假定 CMP 指令为无限位精度。例如，计算 0x8000 - 0x0001 时，若精度限定在 16 位，则结果将会溢出，得到正数 0x7FFF，负标志位 N 清 0。然而由于认为 CMP 指令无限位精度，则负标志位 N 会置 1，以表示 0x8000 - 0x0001 实际上得到一个负数。

Z 测试比较操作是否为 0，若操作 (AX - 0:8bit) = 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置 1。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;检查 VarA 是否在范围内 0x80 <= VarA <= 0xF0
MOV    AL,@VarA           ;装载 VarA 的值到 AL
CMPB   AL,#0xF0           ;根据 (AL - 0x00F0) 的值，设置标志位
SB     OutOfRange,GT      ;若 VarA 大于 0x00F0，则跳转
CMPB   AL,#0x80           ;根据 (AL - 0x0080) 的值，设置标志位
SB     OutOfRange,LT      ;若 VarA 小于 0x0080，则跳转

```

CMPL ACC,loc32

32 位数比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMPL ACC,loc32	0000 1111 LLLL LLLL	×	-	1

操作数： ACC 累加器
loc32 寻址方式

描述： ACC 累加器中的值和“loc32”寻址方式指定的 32 位地址单元中的值进行比较。根据 (ACC-[loc32]) 的结果设置状态标志位，ACC 寄存器和“loc32”指定地址单元中的值都保持不变。

标志与模式： N 若操作结果为负，则负标志位 N 置 1；否则负标志位 N 清 0。在指定结果符号时，假定 CMPL 指令为无限位精度。例如，计算 0x8000 0000 - 0x0000 0001 时，若精度限定在 32 位，则结果将会溢出，得到正数 0x7FFF FFFF，负标志位 N 清 0。由于认为 CMP 指令无限位精度，则负标志位 N 会置 1，以表示 0x8000 0000 - 0x0000 0001 实际上得到一个负数。

Z 若操作 (AX-[loc32]) = 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 若产生借位，则进位标志位 C 清 0，否则进位标志位 C 置 1。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;若 VarB 大于 VarA，则交换 32 位数 VarA 和 VarB
MOVL   ACC,@VarB         ;ACC = VarB
MOVL   P,@VarA           ;P = VarA
CMPL   ACC,@P            ;根据 (VarB - VarA) 的值设置标志位
MOVL   @VarA,ACC,HI      ;若大于，则 VarA = ACC
MOVL   @P,ACC,HI         ;若大于，则 P = ACC
MOVL   @VarA,P           ;VarA = P

```

CMPL ACC,P<<PM

32 位数比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMPL ACC,P<<PM	1111 1111 0101 1001	×	-	1

- 操作数: ACC 累加器
P 乘积寄存器
<<PM 乘积移位方式
- 描述: ACC 累加器中的值和 P 寄存器移位后的值进行比较, 由乘积移位方式 (PM) 指定移位位数。根据 (ACC-[P<<PM]) 的结果设置状态标志位, ACC 累加器和 P 寄存器的值都保持不变。
- 标志与模式: N 若操作结果为负, 则负标志位 N 置 1; 否则负标志位 N 清 0。在指定结果符号时, CMPL 指令假定为无限位精度。例如, 计算 0x8000 0000 - 0x0000 0001 时, 若精度限定在 32 位, 则结果将会溢出, 得到正数 0x7FFF FFFF, 并且 N 清 0。然而由于认为 CMPL 指令是无限位精度, 则标志位 N 置 1, 以表示 0x8000 0000 - 0x0000 0001 实际上得到一个负数。
- Z 测试比较操作是否为 0, 若操作 (ACC - [P<<PM]) = 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
- C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置 1。
- PM 由设置的 PM 值决定乘积寄存器中输出操作时的移位方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;比较 (VarA - VarB >> 4)
MOVL  ACC,@VarA          ;ACC = VarA
SPM    -4                 ;设置右移 4 位移位方式
MOVL  P,@VarB             ;P = VarB
CMPL  ACC,P << PM         ;比较 (VarA - VarB 右移 4 位的值)
```

CMPR 0/1/2/3

辅助寄存器比较

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CMPR 0	0101 0110 0001 1101	1	-	1
CMPR 1	0101 0110 0001 1001	1	-	1
CMPR 2	0101 0110 0001 1000	1	-	1
CMPR 3	0101 0110 0001 1100	1	-	1

操作数: 无

描述: 比较 AR0 和 ARP 指定的 16 位辅助寄存器。比较方式由指令决定, 如下所示:

```
CMPR 0: if (AR0 = AR[ARP]) TC = 1, else TC = 0
CMPR 1: if (AR0 > AR[ARP]) TC = 1, else TC = 0
CMPR 2: if (AR0 < AR[ARP]) TC = 1, else TC = 0
CMPR 3: if (AR0 != AR[ARP]) TC = 1, else TC = 0
```

标志与模式: ARP 3 位宽度的 ARP 指向当前辅助寄存器 XAR0~XAR7。它指定了

AR0 与哪个辅助寄存器进行比较。

TC 若测试为真，则 TC 置 1，否则 TC 清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

TableA: .word 0x1111

.word 0x2222

FuncA:

```

MOVL XAR1, #VarA      ;初始化 XAR1 指针
MOVZ AR0, *XAR1+ +    ;用 0x1111 装载 AR0, AR0H 清 0, ARP = 1
MOVZ AR1, *XAR1      ;用 0x2222 装载 AR1, AR1H 清 0
CMPR 0                ;AR0 是否等于 AR1? 否, 则 TC 清 0
B     Equal, TC        ;不跳转
CMPR 2                ;AR1 是否大于 AR2? 是, 则 TC 置位
B     Less, TC         ;跳转到 "less"

```

CSB ACC 计算符号位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
CSB ACC	0101 0110 0011 0101	1	-	1

操作数： ACC 累加器

描述： 引入 ACC 累加器中 0 或 1 的数目来计算 ACC 累加器中的符号位，结果减 1 保存到 T 寄存器中：

T = 0, 1 符号位

T = 1, 2 符号位

T = 31, 32 符号位

注意：在规格化操作中，常常计算符号位，尤其在运算法则中它特别有用。例如计算平方根、数的相反数，寻找一个字中的第一个“1”。

标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 等于 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

TC TC 位反应了指令执行后符号位的状态（TC = 1 表示为负）。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;规格化 VarA 的值
MOVL ACC, @VarA      ;装载 VarA 的值到 ACC 中
CSB   ACC             ;计算符号位
LSLL  ACC, T          ;ACC 逻辑左移 T(4:0) 位
MOVL  @VarA, ACC      ;结果保存到 VarA

```


DEC loc16

减 1

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
DEC loc16	0000 1011 LLLL LLLL	×	-	1

操作数: loc16 寻址方式。

描述: 把“loc16”地址单元的有符号数减 1。

标志与模式: N 若[loc16]的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若[loc16]为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置 1。

V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;VarA = VarA - 1;
DEC @VarA      ;VarA 的值减 1
```

DINT

禁止可屏蔽中断 (INTM 位置 1)

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
DINT	0011 1011 0001 0000	×	-	2

注意: 本指令是“SETC mode” (mode = INTM) 操作的别名。

操作数: 无

描述: 将 INTM 状态位置 1, 禁止所有可屏蔽 CPU 中断。DINT 指令对复位或 NMI 中断没有影响。

标志与模式: INTM 指令将 INTM 位置 1, 从而禁止中断。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; "VarC = VarA + VarB"
DINT                ;禁止中断 (INTM = 1)
MOVL ACC,@VarA      ;ACC = VarA
ADDL ACC,@VarB       ;ACC = ACC + VarB
MOVL @VarC,ACC       ;结果保存到 VarC
EINT                ;使能中断 (INTM = 0)
```

DMAC ACC:P,loc32,*XAR7/+ +

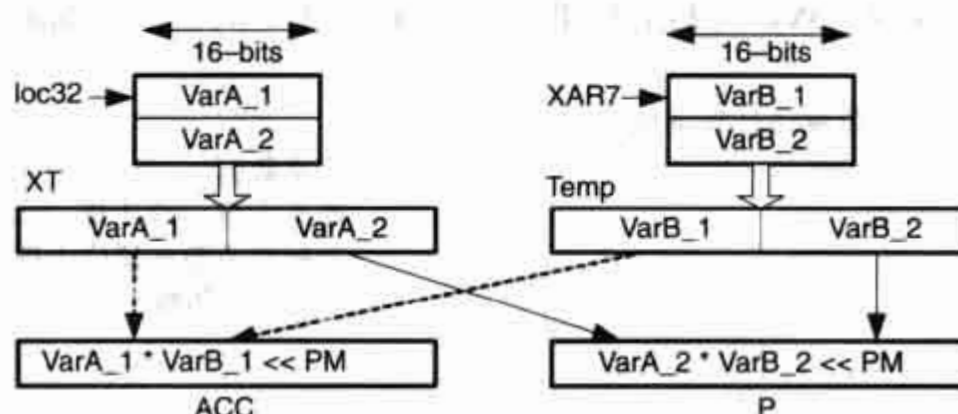
16 位双乘法且累加

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
DMAC ACC:P,loc32,*XAR7	0101 0110 0100 1011 1100 0111 LLLL LLLL	1	Y	N+2
DMAC ACC:P,loc32,*XAR7+ +	0101 0110 0100 1011 1000 0111 LLLL LLLL	1	Y	N+2

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)
 loc32 寻址方式

注意: 不能使用 @ACC 和 @P 寄存器寻址模式, 若使用将不产生非法指令的陷阱 (汇编器仅会标记一个错误)。

***XAR7/++** 使用辅助寄存器 XAR7 间接存储器寻址可以访问整个 4M × 16 程序空间 (0x0000 0000~0x3FFF FFFF)。
描述: 双重 16 位 × 16 位有符号乘法且累加。第一次乘法在 “loc32” 指定 32 位地址与 *XAR7/++ 寻址方式的高字之间进行, 第二次发生在低字之间。



指令执行后, ACC 包含乘法结果加上可寻址的 32 位操作数的高字。P 寄存器包含乘法结果和可寻址的 32 位操作数的低字。

```
XT = [loc32];
Temp = Prog[*XAR7 or *XAR7++];
ACC = ACC + (XT.MSW × Temp.MSW) << PM;
P = P + (XT.LSW × Temp.LSW) << PM;
```

Z、N、V、C 和 OVC 计数器仅仅受到 ACC 操作的影响。PM 移位将影响 ACC 和 P 寄存器。

对于 C28x 器件, 存储器模块映射为程序和数据空间两部分 (存储器统一编址), 因此 “*XAR7/++” 寻址方式可以寻址落入其寻址范围内的数据空间变量。

不同寻址方式组合, 可能得到有冲突的参数。在这种情况下, C28x 将 “loc16/loc32” 区域改变到 XAR7。

例如:

```
DMAC ACC:P, *- -XAR7, *XAR7++ ; - -XAR 给定高优先级
DMAC ACC:P, *XAR7++, *XAR7 ; *XAR7++ 给定高优先级
DMAC ACC:P, *XAR7, *XAR7++ ; *XAR7++ 给定高优先级
```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
 Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 C 若 ACC 产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。
 V 若 ACC 累加器发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
 OVC 若溢出模式禁止, 当操作产生 ACC 累加器的正溢出时, 计算器值

增加；当操作产生 ACC 累加器的负溢出时，计数器减小。

OVM 若溢出模式位置位，在操作发生溢出时，ACC 将为饱和值，即为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。

注意： OVM 只影响 ACC 操作。

PM PM 位设置乘积寄存器的输出操作移位方式。PM 移位方式会影响 ACC 和 P 寄存器累加结果。若乘积移位位数为正（逻辑左移操作），则最低位填 0。若乘积移位位数为负（算术右移操作），则高位进行符号扩展。

重复性： 本指令可以重复。若操作用在 RPT 指令后，则执行 N+1 次指令。Z、N、C 和 OVC 为最后结果。若 ACC 发生中间溢出，则溢出标志位 V 置 1。

例：

```

;使用双重 16 位乘法计算乘积的和
;int16 X[N]
;int16 C[N]
;
;
;sum = 0;
;for(i = 0; i < N; i++)
;sum = sum + (X[i] × C[i]) >> 5;
MOVL  XAR2, #X
MOVL  XAR7, #C
SPM   -5
ZAPA
RPT   #(N/2)-1
||DMAC P, *XAR2+, *XAR7+ +

ADDL  ACC, @P
MOVL  @sum, ACC

```

;定义数据
 ;定义系数（位于低 4MB）
 ;数据和系数必须位于偶地址
 ;N 必须为偶数
 ;XAR2 指针指向 X
 ;XAR7 指针指向 C
 ;设置乘积右移 5 位 ">> 5"
 ;ACC, P, OVC 清 0
 ;重复下一条指令 N/2 次
 ;ACC = ACC + (X[i+1] × C[i+1]) >> 5
 ;P = P + (X[i] × C[i]) >> 5 i++
 ;执行最后计算
 ;保存最后结果到 sum

DMOV loc16

移动 16 位地址单元中的数据

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
DMOV loc16	1010 0101 LLLL LLLL	1	Y	N+1

操作数： loc16 寻址方式

注意： 在该操作中，不能使用寄存器寻址方式，寄存器寻址模式有：@ARn, @AH, @AL, @PH, @PL, @SP, @T。它将会产生一个非法指令陷阱。

描述： 复制“loc16”地址单元的内容到下一个最高地址：

[loc16+1] = [loc16];

标志与模式： 无

重复性： 本指令可以重复。若操作用在 RPT 指令后，则指令将执行 N+1 次。

例:

```

;使用 16 位乘法计算
;int16 X[3];
;int16 C[3];
;Y = (X[0]×C[0] >> 2) + (X[1]×C[1] >> 2) + (X[2]×C[2] >> 2);
;X[2] = X[1];
;X[1] = X[0];
SPM -2                ;设置乘积右移 2 位">> 2"
MOVP T,@X+2           ;T = X[2]
MPYS P,T,@C+2         ;P = T×C[2], ACC = 0
MOVA T,@X+1           ;T = X[1], ACC = X[2]×C[2] >> 2
MPY P,T,@C+1          ;P = T×C[1]
MOVA T,@X+0           ;T = X[0], ACC = ACC + X[1]×C[1] >> 2
MPY P,T,@C+0          ;P = T×C[0]
ADDL ACC,P << PM      ;ACC = ACC + X[0]×C[0] >> 2
DMOV @X+1             ;X[2] = X[1]
DMOV @X+0             ;X[1] = X[0]
MOVL @Y,ACC           ;结果保存到 Y

```

EALLOW

使能对保护空间的写访问

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
EALLOW	0111 0110 0010 0010	×	-	4

操作数: 无

描述: 对访问仿真空间和其他保护寄存器使能。本指令设置状态寄存器 ST1 中的 EALLOW 位。当该位为 1 时, C28xCPU 允许访问存储器映射寄存器和其他保护寄存器。哪些寄存器可以写保护请参考所用器件的数据手册。使用 EDIS 指令, 可以再一次对寄存器写保护。EALLOW 位仅仅控制写访问, 即使没有执行 EALLOW 指令, 仍允许读操作。在中断或陷阱时, EALLOW 位的当前状态保存在堆栈中的 ST1 内, EALLOW 位自动清 0。因此, 开始中断服务程序时, 不允许访问保护寄存器。IRET 指令将恢复保存到堆栈中的 EALLOW 位的当前值。EALLOW 位使 JTAG 接口在 CCS 仿真时, 允许完全控制寄存器访问。

标志与模式: EALLOW EALLOW 标志位置 1。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;具有写保护功能的 RegA 和 RegB 的访问使能
EALLOW                ;使能所选择寄存器的访问
AND @RegA,#0x4000     ;RegA = RegA 与 0x0400
MOV @RegB,#0          ;RegB = 0
EDIS                  ;禁止对所选择寄存器的访问

```

EDIS

禁止保护寄存器的写访问

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
EDIS	0111 0110 0001 1010	×	-	4

- 操作数: 无
- 描述: 禁止对仿真空间和其他保护寄存器的访问。本指令将状态寄存器 ST1 中的 EALLOW 位清 0。当该位清 0 时, C28x CPU 不允许访问存储器映射寄存器和其他保护寄存器。
- 标志与模式: EALLOW EALLOW 标志位清 0。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;具有写保护功能的 RegA 和 RegB 的访问使能
EALLOW           ;使能所选择寄存器的访问
NOP              ;等待两个周期让指令起作用。
                ;等待周期数由所用芯片决定。
NOP
AND @RegA, #0x4000 ;RegA = RegA 与 0x0400
MOV @RegB, #0      ;RegB = 0
EDIS             ;禁止对所选择寄存器的访问
```

EINT

使能可屏蔽中断 (INTM 位清 0)

语法选项	操作码	OBJ 模式	RPT	CYC
EINT	0010 1001 0001 0000	×	-	2

注意: 本指令是“CLRC mode” (mode = INTM) 操作的别名。

- 操作数: 无
- 描述: 将 INTM 状态位清 0, 使能中断。
- 标志与模式: INTM 用指令使 INTM 位清 0, 使能中断。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;实现操作"VarC = VarA + VarB"
DINT                ;禁止中断 (INTM = 1)
MOVL ACC, @VarA     ;ACC = VarA
ADDL ACC, @VarB     ;ACC = ACC + VarB
MOVL @VarC, ACC     ;结果保存到 VarC
EINT                ;使能中断 (INTM = 0)
```

ESTOP0

仿真停止 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ESTOP0	0111 0110 0010 0101	×	-	1

- 操作数: 无
- 描述: 仿真停止 0

本指令在仿真中是很有用的, 它可以产生软件断点。当仿真器连接到 C28x 并仿真使能时, 本指令将使 C28x 停止, 而不管状态寄存器 ST1



附录 C 汇编器错误信息

当汇编器完成第二遍扫描时，它报告在汇编过程中所遇到的所有错误。如果创建了列表文件，那么它还会在列表文件中输出错误；错误在出现错误的源程序行之后输出。在纠正其他错误之前用户应该纠正在代码中出现的第一个错误；因为第一个错误有可能引发其他错误。

如果用户收到汇编器的错误信息，那么可以使用本附录找到解决问题的方法。首先定位错误信息等级号（等级号按字母顺序在本目录中列出）。然后在等级中定位用户得到的错误信息（每个等级号按字母顺序列出错误信息）。每个等级包括问题的描述以及提出可能的修正建议。

E0000

Can't find end of '%s' macro definition starting on line %d—Aborting!

Comma required to separate arguments

FuncA: ;函数 FuncA

.

LB *XAR7

;返回跳转到 XAR7 中的地址处

FLIP AX

改变 AX 寄存器中位的顺序

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
FLIP AX	0101 0110 0111 000A	1	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器。

描述: 指定的 AX 寄存器 (AH 或 AL) 的值进行位颠倒:

```
temp = AX;
AX(bit 0) = temp(bit 15);
AX(bit 1) = temp(bit 14);
.
.
AX(bit 14) = temp(bit 1);
AX(bit 15) = temp(bit 0);
```

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 AX 为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;改变 32 位变量 VarA 的值

```
MOV    AH,@VarA+0      ;把 VarA 的低 16 位装载到 AH
MOV    AL,@VarA+1      ;把 VarA 的高 16 位装载到 AL
FLIP   AL              ;颠倒 AL
FLIP   AH              ;颠倒 AH
MOVL   @VarA,ACC       ;保存 32 位结果到 VarA
```

IACK #16bit

中断应答

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IACK #16bit	0111 0110 0011 1111 CCCC CCCC CCCC CCCC	×	-	1

操作数: #16bit 16 位立即数 (0x0000~0xFFFF)

描述: 在数据总线的低 16 位上输出指定的 16 位立即数去应答中断。某些外设有能力捕获该值, 以提供低功耗功能。细节请参考器件数据手册。

```
Data-bus (15:0) = 16bit;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

IDLE

使处理器进入空闲模式

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IDLE	0111 0110 0010 0001	×	-	5

操作数： 无

描述： 使处理器进入空闲模式，等待使能或非屏蔽中断。使用 C28x CPU 的器件可以使用 IDLE 指令结合外部逻辑完成各种低功耗模式。详情请参考器件的数据手册。IDLE 指令将引起下列事件序列：

- 1) 刷新流水线。
- 2) 完成所有未完成的存储器周期。
- 3) 状态寄存器 ST1 中的 IDLESTAT 位置 1。
- 4) 整个指令缓冲器满后，停止 CPU 时钟，使器件进入空闲状态。在空闲状态中，一旦驱动 3CLKIN，CLKOUT（CPU 输出时钟）和所有 CPU 外部的时钟（包含仿真模块）就会继续操作。PC 继续保持 IDLE 指令的地址，在 CPU 进入空闲模式前，PC 不会增加。
- 5) IDLE 输出 CPU 信号是有效的（驱动为高）。
- 6) 器件等待使能或非屏蔽硬件中断。若发生这样的中断，IDLESTAT 位清 0，PC 值加 1，器件退出空闲状态。

若中断是可屏蔽的，它必须在中断使能寄存器（IER）中使能。然而。不管状态寄存器 ST1 中的中断全局屏蔽位（INTM）为何值，器件都会退出空闲状态。

器件退出空闲模式后，CPU 必须响应中断请求。若使用状态寄存器 ST1 中的 INTM 位禁止中断，下面的事件依赖于 INTM。

- 若中断使能（INTM = 0），CPU 执行相应的中断服务程序。返回中断时，从 IDLE 指令的下一条指令开始执行。
 - 若禁止中断（INTM = 1），程序从 IDLE 的下一条指令继续执行。
- 若中断未用 INTM 位禁止，CPU 执行相应的中断服务程序。返回中断时，从 IDLE 指令的下一条指令开始执行。

标志与模式： IDLESTAT 进入中断模式前，IDLESTAT 置 1；退出空闲模式后，IDLESTAT 清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

IMACL P,loc32,*XAR7/+ +

有符号 32×32 位乘法且累加

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMACL P,loc32,*XAR7	0101 0110 0100 1101 1100 0111 LLLL LLLL	1	Y	N+2
IMACL P,loc32,*XAR7+ +	0101 0110 0100 1101 1000 0111 LLLL LLLL	1	Y	N+2

操作数: P 乘积寄存器
loc32 寻址方式

注意: 当重复指令时, 不能使用 @ACC 寻址方式, 若使用将不产生非法指令的陷阱 (汇编器仅会标记一个错误)。

描述: *XAR7/+ + 使用辅助寄存器 XAR7 间接存储器寻址可以访问整个 $4\text{M} \times 16$ 程序空间 ($0\text{x}0000\ 0000 \sim 0\text{x}3\text{FFF}\ \text{FFFF}$)。
32 位 \times 32 位有符号乘法且累加。首先, 加先前无符号乘积 (保存在 P 寄存器中) 到 ACC 累加器, 忽略乘积移位方式 (PM)。然后, “loc32” 寻址方式指定地址单元中的 32 位有符号值与 XAR7 寄存器指定程序存储器中的 32 位有符号值相乘。乘积移位方式指定 64 位结果的低 38 位的哪一部分保存在 P 寄存器中。若特别指定, 则 XAR7 加 1

```
ACC = ACC + unsigned P;
temp(37:0) = lower_38_bits(signed [loc32]
× signed Prog[*XAR7 or XAR7+ +]);
if( PM = +4 shift )
    P(31:4) = temp(27:0), P(3:0) = 0;
if( PM = +1 shift )
    P(31:1) = temp(30:0), P(0) = 0;
if( PM = 0 shift )
    P(31:0) = temp(31:0);
if( PM = -1 shift )
    P(31:0) = temp(32:1);
if( PM = -2 shift )
    P(31:0) = temp(33:2);
if( PM = -3 shift )
    P(31:0) = temp(34:3);
if( PM = -4 shift )
    P(31:0) = temp(35:4);
if( PM = -5 shift )
    P(31:0) = temp(36:5);
if( PM = -6 shift )
    P(31:0) = temp(37:6);
```

对于 C28x 器件, 存储器模块映射为程序和数据空间两部分 (存储器统一编址), 因此 “*XAR7/+ +” 寻址方式可以寻址落入其寻址范围内的数据空间变量。

不同寻址方式组合, 可能得到有冲突的参数。在这种情况下, C28x 将给定优先级。

例如:

IMACL	P, *- -XAR7, *XAR7+ +	; - -XAR7 给定高优先级
IMACL	P, *XAR7+ +, *XAR7	; *XAR7+ + 给定高优先级
IMACL	P, *XAR7, *XAR7+ +	; *XAR7+ + 给定高优先级

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

- Z** 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。
- C** 若加操作 ACC 累加器产生进位，则进位标志位 C 置 1；否则进位标志位 C 清 0。
- V** 若 ACC 累加器发生溢出，则溢出标志位 V 置 1；否则 V 不受影响。
- OVCU** 当加法操作产生无符号进位时，计算器值增加；OVM 不影响 OVCU 计数器。
- PM** PM 位设置移位方式，它指定 64 位结果的低 38 位如何保存到 P 寄存器。

重复性: 本指令是可以重复。若用在 RPT 指令后，则执行 N+1 次指令。Z、N、C 和 OVC 为最后结果。若发生中间溢出，则溢出标志位 V 置 1。

例:

```

;计算 32 位乘法的乘积与余下的 64 位结果
;int32 X[N];           //数据定义
;int32 C[N];           //系数定义 (位于低 4M)
; int64 sum = 0;
; for(i = 0; i < N; i+ +)
; sum = sum + (X[i] × C[i]) >> 5;
;计算低 32 位
MOVL    XAR2, #X        ;XAR2 指针指向 X
MOVL    XAR7, #C        ;XAR7 指针指向 C
SPM     -5              ;设置乘积右移 5 位
ZAPA    ;ACC, P, OVCU 清 0
RPT     #(N-1)          ;重复下一条指令 N 次
||IMACL  P, *XAR2+ +, *XAR7+ + ;OVCU:ACC = OVCU:ACC + P,
                               ; P = (X[i] × C[i]) << 5,
                               ; i+ +
                               ;OVCU:ACC = OVCU:ACC + P
ADDUL   ACC, @P          ;保存结果的低 32 位到 sum
MOVL    @sum+0, ACC      ;计算高 32 位
MOVU    @AL, OVC         ;ACC = OVCU (进位计数)
MOVB    AH, #0
MPYB    P, T, #0         ;P = 0
MOVL    XAR2, #X        ;XAR2 指针指向 X
MOVL    XAR7, #C        ;XAR7 指针指向 C
RPT     #(N-1)          ;重复下一条指令 N 次
||QMACL  P, *XAR2+ +, *XAR7+ + ;ACC = ACC + P >> 5,
                               ;P = (X[i] × C[i]) >> 32, i+ +
                               ;ACC = ACC + P >> 5
ADDL    ACC, P << PM     ;保存结果的高 32 位到 sum
MOVL    @sum+2, ACC

```

IMPYAL P,XT,loc32

加上先前的 P 中的内容后作有符号 32 位乘法(低半部分)

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMPYAL P,XT,loc32	0101 0110 0100 1100 0000 0000 LLLL LLLL	1	Y	1

操作数: **P** 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

注意: 当指令重复时, 不能使用 @ACC 寻址方式, 若使用将不产生非法指令的陷阱 (汇编器仅会标记一个错误)。

描述: 忽略乘积移位方式 (PM), 加无符号 P 寄存器的内容到 ACC 累加器中。XT 寄存器的 32 位有符号数和 “loc32” 寻址方式指定地址单元中的 32 位有符号数相乘, 乘积移位方式 (PM) 指定 64 位结果中低 38 位的哪一部分保存到 P 寄存器:

```
ACC = ACC + unsigned P;
temp(37:0) = lower_38_bits(signed XT x signed [loc32]);
if( PM = +4 shift )
P(31:4) = temp(27:0), P(3:0) = 0;
if( PM = +1 shift )
P(31:1) = temp(30:0), P(0) = 0;
if( PM = 0 shift )
P(31:0) = temp(31:0);
if( PM = -1 shift )
P(31:0) = temp(32:1);
if( PM = -2 shift )
P(31:0) = temp(33:2);
if( PM = -3 shift )
P(31:0) = temp(34:3);
if( PM = -4 shift )
P(31:0) = temp(35:4);
if( PM = -5 shift )
P(31:0) = temp(36:5);
if( PM = -6 shift )
P(31:0) = temp(37:6);
```

标志与模式: **Z** 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 则负标志位 N 清 0。
 C 若产生进位, 则进位标志位 C 置 1; 则进位标志位 C 清 0。
 V 若发生溢出, 则溢出标志位 V 置 1; 则 V 不受影响。
 OVCU 当产生无符号进位时, 计算器值增加; OVM 不影响 OVCU 计数器。
 PM PM 位设置移位方式, 它指定 64 位结果中 38 位中哪一部分保存到 P 寄存器。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;计算有符号结果
;Y64 = (X0xC0 + X1xC1 + X2xC2) >> 2
SPM      -2                      ;设置移位方式右移 2 位 ">> 2"
ZAPA                      ; ACC, P, OVCU 清 0
MOVL     XT,@X0            ;XT = X0
IMPYAL   P,XT,@C0          ;P = (X0xC0 << 2)的低 32 位
MOVL     XT,@X1            ;XT = X1
IMPYAL   P,XT,@C1          ;OVCU:ACC = OVCU:ACC + P,
                          ;P = (X1xC1 << 2)的低 32 位

MOVL     XT,@X2            ;XT = X2
IMPYAL   P,XT,@C2          ;OVCU:ACC = OVCU:ACC + P,
                          ;P = (X2xC2 << 2)的低 32 位

ADDUL    ACC,@P            ;OVCU:ACC = OVCU:ACC + P
MOVL     @Y64+0,ACC        ;保存结果的低 32 位到 Y64
MOVU     @AL,OVC           ;ACC = OVCU (进位计数)
MOVB     AH,#0
QMPYAL   P,XT,@C2          ;P = (X2xC2)的高 2 位
MOVL     XT,@X1            ;XT = X1
QMPYAL   P,XT,@C1          ;ACC = ACC + P >> 2,
                          ;P = (X1xC1)的高 32 位

MOVL     XT,@X0            ;XT = X0
QMPYAL   P,XT,@C0          ;ACC = ACC + P >> 2,
                          ;P = (X0xC0)的高 32 位

ADDL     ACC,P << PM       ;ACC = ACC + P >> 2
MOVL     @Y64+2,ACC        ;保存结果的高 32 位到 Y64

```

IMPYAL ACC,XT,loc32

有符号 32×32 位乘法（低半部分）

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMPYAL ACC,XT,loc32	0101 0110 0100 0100 0000 0000 LLLL LLLL	1	-	2

操作数: ACC 累加器
 XT 被乘数寄存器
 loc32 寻址方式

注意: 当重复指令时, 不能使用 @ACC 寻址方式, 若使用将不产生非法指令的陷阱 (汇编器仅会标记一个错误)。

描述: XT 寄存器中的 32 位有符号数和“loc32”寻址方式指定地址单元中的 32 位有符号数相乘, 64 位结果的低 32 位保存到 ACC 累加器中:

ACC = 有符号的 XT × 有符号的[loc32]

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;计算      Y32 = M32×X32 + B32
MOVL      XT,@M32                ;XT = M32
IMPYL     ACC,XT,@X32            ;ACC = (M32×X32)的低32位
ADDL      ACC,@B32               ;ACC = ACC + B32
MOVL      @Y32,ACC              ;结果保存到Y32

```

IMPYL P,XT,loc32

有符号 32×32 位乘法（低半部分）

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMPYL P,XT,loc32	0101 0110 0000 0101 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

注意: 当重复指令时, 不能使用@ACC 寻址方式, 若使用将不产生非法指令的陷阱(汇编器仅会标记一个错误)。

描述: XT 寄存器中的 32 位有符号数和“loc32”寻址方式指定地址单元中的 32 位有符号数相乘。乘积移位方式(PM)指定 64 位结果的低 38 位中的哪一部分保存到 P 寄存器:

```

temp(37:0) = lower_38 bits(signed XT × signed [loc32]);
if( PM = +4 shift )
    P(31:4) = temp(27:0), P(3:0) = 0;
if( PM = +1 shift )
    P(31:1) = temp(30:0), P(0) = 0;
if( PM = 0 shift )
    P(31:0) = temp(31:0);
if( PM = -1 shift )
    P(31:0) = temp(32:1);
if( PM = -2 shift )
    P(31:0) = temp(33:2);
if( PM = -3 shift )
    P(31:0) = temp(34:3);
if( PM = -4 shift )
    P(31:0) = temp(35:4);
if( PM = -5 shift )
    P(31:0) = temp(36:5);
if( PM = -6 shift )
    P(31:0) = temp(37:6);

```

标志与模式: PM PM 位设置移位方式, 它指定 64 位结果的低 38 位的哪一部分保存到 P 寄存器中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;计算有符号结果 Y64 = M32×X32
MOVL    XT,@M32                ;XT = M32
IMPYSL   P,XT,@X32              ;P = (M32×X32)的低 32 位
QMPYSL   ACC,XT,@X32            ;ACC = (M32×X32)的高 32 位
MOVL     @Y64+0,P               ;结果保存到 Y64
MOVL     @Y64+2,ACC

```

IMPYSL P,XT,loc32 减先前的 P 中的内容后作有符号 32 位乘法(低半部分)

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMPYSL P,XT,loc32	0101 0110 0100 0011 0000 0000 LLLL LLLL	1	Y	1

操作数: P 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

描述: 忽略乘积移位方式 (PM), 从 ACC 累加器中减去 P 寄存器中无符号的内容。XT 寄存器的 32 位有符号数和“loc32”寻址方式指定地址单元中的 32 位有符号数相乘, 乘积移位方式 (PM) 指定 64 位结果中的低 38 位的哪一部分保存到 P 寄存器:

```

ACC = ACC + unsigned P;
temp(37:0) = lower_38 bits(signed XT × signed [loc32]);
if( PM = +4 shift )
    P(31:4) = temp(27:0), P(3:0) = 0;
if( PM = +1 shift )
    P(31:1) = temp(30:0), P(0) = 0;
if( PM = 0 shift )
    P(31:0) = temp(31:0);
if( PM = -1 shift )
    P(31:0) = temp(32:1);
if( PM = -2 shift )
    P(31:0) = temp(33:2);
if( PM = -3 shift )
    P(31:0) = temp(34:3);
if( PM = -4 shift )
    P(31:0) = temp(35:4);
if( PM = -5 shift )
    P(31:0) = temp(36:5);
if( PM = -6 shift )
    P(31:0) = temp(37:6);

```

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
 C 若减操作产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置 1。
 V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。
 OVCU 当减操作产生无符号借位时, 计数器值减小; OVM 不影响 OVCU

计数器。

PM PM 位设置移位方式，它指定 64 位结果的低 38 位的哪一部分保存到 P 寄存器。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;计算有符号数
;Y64 = (-X0xC0 - X1xC1 - X2xC2) >> 2
SPM    -2                                ;设置移位方式右移 2 位 "<< 2"
ZAPA                                ;ACC, P, OVCU 清 0
MOVL   XT,@X0                        ;XT = X0
IMPYSL P,XT,@C0                       ;P = (X0xC0 << 2) 的低 32 位
MOVL   XT,@X1                        ;XT = X1
IMPYSL P,XT,@C1                       ;OVCU:ACC = OVCU:ACC - P,
                                        ;P = (X1xC1 << 2) 的低 32 位
MOVL   XT,@X2                        ;XT = X2
IMPYSL P,XT,@C2                       ;OVCU:ACC = OVCU:ACC - P,
                                        ;P = (X2xC2 << 2) 的低 32 位
SUBUL   ACC,@P                       ;OVCU:ACC = OVCU:ACC - P
MOVL   @Y64+0,ACC                     ;保存结果的低 32 位到 Y64
MOVU   @AL,OVC                        ;ACC = OVCU (借位计数)
MOVB   AH,#0
NEG     ACC                           ;无借位
QMPYSL P,XT,@C2                       ;P = (X2xC2) 的高 32 位
MOVL   XT,@X1                        ;XT = X1
QMPYSL P,XT,@C1                       ;ACC = ACC - P >> 2, 1
                                        ;P = (X1xC1) 的高 32 位
MOVL   XT,@X0                        ;XT = X0
QMPYSL P,XT,@C0                       ;ACC = ACC - P >> 2,
                                        ;P = (X0xC0) 的高 32 位
SUBUL   ACC,P << PM                   ;ACC = ACC - P >> 2
MOVL   @Y64+2,ACC                     ;保存结果的高 32 位到 Y64

```

IMPYXUL P,XT,loc32

有符号 32 位数与无符号 32 位数乘法(低半部分)

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IMPYXUL P,XT,loc32	0101 0110 0110 0101 0000 0000 LLLL LLLL	1	-	1

操作数： P 乘积寄存器
XT 被乘数寄存器
loc32 寻址方式

描述： XT 寄存器的 32 位有符号数和“loc32”寻址方式指定地址单元中的 32 位无符号数相乘，乘积移位方式 (PM) 指定 64 位结果中低 38 位的哪一部分保存到 P 寄存器：

temp(37:0) = lower_38 bits(signed XT × signed [loc32]);

```

if( PM = +4 shift )
    P(31:4) = temp(27:0), P(3:0) = 0;
if( PM = +1 shift )
    P(31:1) = temp(30:0), P(0) = 0;
if( PM = 0 shift )
    P(31:0) = temp(31:0);
if( PM = -1 shift )
    P(31:0) = temp(32:1);
if( PM = -2 shift )
    P(31:0) = temp(33:2);
if( PM = -3 shift )
    P(31:0) = temp(34:3);
if( PM = -4 shift )
    P(31:0) = temp(35:4);
if( PM = -5 shift )
    P(31:0) = temp(36:5);
if( PM = -6 shift )
    P(31:0) = temp(37:6);

```

标志与模式: PM PM 位设置移位方式, 它指定 64 位结果中低 38 位中的哪一部分保存到 P 寄存器。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;计算 Y64 = M64×X64 + B64
;Y64 = Y1:Y0, M64 = M1:M0, X64 = X1:X0, B64 = B1:B0
MOVL  XT,@X0                ;XT = X0
IMPYL  P,XT,@M0              ;P = (uns M0 × uns X0)的低 32 位
MOVL  ACC,@B0                ;ACC = B0
ADDUL  ACC,@P                 ;ACC = ACC + P
MOVL  @Y0,ACC                ;结果保存到 Y0
QMPYUL P,XT,@M0              ;P = (uns M0 × uns X0)的高 32 位
MOVL  XT,@X1                 ;XT = X1
MOVL  ACC,@P                 ;ACC = P
IMPYXUL P,XT,@M0             ;P = (uns M0 × sign X1)的低 32 位
MOVL  XT,@M1                 ;XT = M1
ADDCL  ACC,@P                 ;ACC = ACC + P + 进位位
IMPYXUL P,XT,@X0             ;P = (sign M1 × uns X0)的低 32 位
ADDUL  ACC,@P                 ;ACC = ACC + P
ADDUL  ACC,@B1                ;ACC = ACC + B1
MOVL  @Y1,P                  ;结果保存到 Y1

```

IN loc16,* (PA)

从口地址单元中读入数据

运 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IN loc16,* (PA)	1011 0100 LLLL LLLL CCCC CCCC CCCC CCCC	1	Y	N+2

操作数: loc16 寻址方式

描述: * (PA) 直接寻址 I/O 空间存储器地址
把* (PA) 指定地址的 I/O 口的值装载到“loc16”寻址方式指定的地址单元中:

[loc16] = IOSpace[PA];

I/O 空间限制在 64K 范围内 (0x0000~0xFFFF)。若器件使用外设接口 (XINTF) 连接, I/O 片选信号 (XIS) 将使能。I/O 地址为 XINTF 地址线的低 16 位 (XA[15:0]), 高位地址线为 0。数据在低 16 位数据线上存取 (XD[15:0])。

注意: I/O 空间不是所有 C28x 器件上都有。请参考器件的数据手册。

标志与模式: N 若 (loc16 = @AX), 测试 AX 是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 (loc16 = @AX), 测试 AX 是否为 0。若 AX 的值为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

重复性: 本指令可以重复。若操作作用在 RPT 指令后, 则执行 N+1 次指令。当重复执行时, “(PA)” I/O 空间的地址在每次重复后加 1。

例:

```

;IORegA address = 0x0300;
;IORegB address = 0x0301;
;IORegC address = 0x0302;
;IORegA = 0x0000;
;IORegB = 0x0400;
;IORegC = VarA;
;if( IORegC = 0x2000 )
;IORegC = 0x0000;
IORegA .set 0x0300      ;定义 IORegA 地址
IORegB .set 0x0301      ;定义 IORegB 地址
IORegC .set 0x0302      ;定义 IORegC 地址
MOV    @AL, #0          ;AL = 0
UOUT   *(IORegA), @AL    ;IOSpace[IORegA] = AL
MOV    @AL, #0x0400     ;AL = 0x0400
UOUT   *(IORegB), @AL    ;IOSpace[IORegB] = AL
OUT     *(IORegC), @VarA ;IOSpace[IORegC] = VarA
IN      @AL, *(IORegC)   ;AL = IOSpace[IORegC]
CMP     @AL, #0x2000     ;根据 (AL - 0x2000) 的值设置标志位
SB      $10, NEQ         ;若不相等, 则跳转
MOV     @AL, #0          ;AL = 0
UOUT   *(IORegC), @AL    ;IOSpace[IORegC] = AL

```

\$10:

INC loc16

加 1

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
INC loc16	0000 1010 LLLL LLLL	×	-	1

操作数: loc16 寻址方式。

描述: “loc16” 地址单元内的数加 1

$[loc16] = [loc16] + 1;$

标志与模式: N 若[loc16]的第 15 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若[loc16] = 0, 则零标志位 Z 置 1; 否则 Z 零标志位清 0。

C 若产生进位, 则进位标志位 C 置 1; 否则进位标志位 C 清 0。

V 若发生溢出, 则溢出标志位 V 置 1; 否则 V 不受影响。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;VarA = VarA+1;
```

```
INC @VarA ;VarA 增 1
```

INTR

仿真硬件中断

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
INTR INTx	0000 0000 0001 CCCC	×	-	8
INTR DLOGINT	0000 0000 0001 CCCC	×	-	8
INTR RTOSINT	0000 0000 0001 CCCC	×	-	8
INTR NMI	0111 0110 0001 0110	×	-	8
INTR EMUINT	0000 0000 0001 1100	×	-	8

操作数: INTx 可屏蔽的 CPU 中断向量名, x = 1~14。

DLOGINT 可屏蔽的 CPU 数据触发中断。

RTOSINT 可屏蔽的 CPU 实时操作系统中断。

NMI 非屏蔽中断。

EMUINT 可屏蔽仿真中断。

描述: 仿真一个中断。INTR 指令可以控制程序流程跳转到指令指定的相应中断向量的中断服务程序处。INTR 指令不受状态寄存器 ST1 中的 INTM 位的影响。它也不受中断使能寄存器 (IER) 或仿真中断使能寄存器 (DBGIER) 中的使能位的影响。一旦 INTR 指令进入了流水线的解码阶段 2, INTR 指令执行完后, 直到中断服务程序开始, 才会响应硬件中断。

INTx, x =	中 断 向 量
0	RESET
1	INT1
2	INT2
3	INT3
4	INT4
5	INT5
6	INT6

续表

INTx, x =	中 断 向 量
7	INT7
8	INT8
9	INT9
10	INT10
11	INT11
12	INT12
13	INT13
14	INT14

可以保存 16 位的 CPU 寄存器对到 SP 寄存器指定的堆栈中。每一寄存器对都保存在一个 32 位操作数中。寄存器对的低字寄存器先保存（偶地址）；形成寄存器对的高字寄存器后保存（随后的奇地址）。例如在保存 T 寄存器和状态寄存器 ST0（T:ST0）时，先保存 ST0，然后保存 T。

当外设中断扩展（PIE）模块使能时，指令不能与向量 1~12 一起使用。

```

if (不是 NMI 向量)
清相应的 IFR 位;
刷新流水线;
temp = PC + 1;
取指定向量;
SP = SP + 1;
[SP] = T:ST0;
SP = SP + 2;
[SP] = AH:AL;
SP = SP + 2;
[SP] = PH:PL;
SP = SP + 2;
[SP] = AR1:AR0;
SP = SP + 2;
[SP] = DP:ST1;
SP = SP + 2;
[SP] = DBGSTAT:IER;
SP = SP + 2;
[SP] = temp;
清相应的 IER 位
INTM = 0;
DBGM = 1;
EALLOW = 0;
LOOP = 0;
IDLESTAT = 0;
PC = 取得向量;

```

```

// 禁止 INT1-INT14, DLOGINT, RTOSINT
// 禁止仿真事件
// 禁止对仿真寄存器的访问
// 清 loop 标志位
// 清 idle 标志位

```

标志与模式: **DBGM**
INTM

置位 **DBGM**，禁止仿真事件。
置位 **INTM** 位，禁止可屏蔽中断。

EALLOW EALLOW 位清 0 以禁止对保护寄存器的访问。

LOOP loop 标志位清 0。

IDLESTAT 空闲标志位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

IRET

中断返回

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
IRET	0111 0110 0000 0010	×	-	8

操作数: 无。

描述: 从中断返回。IRET 指令恢复 PC 值和中断操作时自动保存的其他寄存器的值。这些值恢复的次序是和保存的次序相反的。弹出堆栈的所有值都是 32 位操作。在堆栈恢复操作时, 堆栈指针不会强迫对齐到偶地址处:

```

SP = SP - 2;
PC = [SP];
SP = SP - 2;
DBGSTAT:IER = [SP];
SP = SP - 2;
DP:ST1 = [SP];
SP = SP - 2;
AR1:AR0 = [SP];
SP = SP - 2;
PH:PL = [SP];
SP = SP - 2;
AH:AL = [SP];
SP = SP - 2;
T:ST0 = [SP];
SP = SP - 1;

```

注意: 直到 IRET 指令执行完成后, 才能进入下一个中断服务。

标志与模式: SXM 该操作恢复所有标志位的状态和 ST0 寄存器的模式。

OVM

TC

C

Z

N

V

PM

OVC

INTM 操作恢复指定标志位状态和 ST1 寄存器的模式。

以下的位不受影响: LOOP, IDLESTAT, M0M1MAP。

DBGM

PAGE0
 VMAP
 SPA
 EALLOW
 AMODEOBJ
 XF
 ARP

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 整个中断内容的保存和恢复
; 向量表
INTx: .long INTxService          ; INTx 中断向量
.
.
.
; 中断内容的保存:
INTxService:                     ; ACC, P, T, ST0, ST1, DP, AR0,
; AR1, IER, DPGSTAT 寄存器保存
; 返回堆栈中保存的 PC 值, 禁止 INTx 相应的 IER,
; ST1(EALLOW 位 = 0), ST1(LOOP 位 = 0),
; ST1(DBGM 位 = 1), ST1(INTM 位 = 1)。
PUSH  AR1H:AR0H                  ; 保存余下的寄存器
PUSH  XAR2
PUSH  XAR3
PUSH  XAR4
PUSH  XAR5
PUSH  XAR6
PUSH  XAR7
PUSH  XT
; 中断程序中的用户代码:
.
.
.
; 恢复中断内容
POP   XT                          ; 恢复寄存器
POP   XAR7
POP   XAR6
POP   XAR5
POP   XAR4
POP   XAR3
POP   XAR2
POP   AR1H:AR0H
IRET                             ; 从中断返回

```

LB *XAR7

间接长跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LB *XAR7	0111 0110 0010 0000	×	—	4

操作数: *XAR7 使用辅助寄存器 XAR7 间接程序存储器寻址, 可以寻址整个 4M×16 程序空间范围 (0x000000~0x3FFFFFF)

描述: 间接长跳转。用 XAR7 寄存器的低 22 位装载 PC 指针:

PC = XAR7 (21:0);

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 跳转到开关值选择的开关表中子程序处:
SwitchTable:                                ; 开关地址表
.long Switch0                               ; 开关 0 的地址
.long Switch1                               ; 开关 1 的地址
.
.
MOVL XAR2, #SwitchTable                    ; XAR2 指针指向开关表
MOVZ AR0, @Switch                          ; AR0 开关表指针
MOVL XAR7, *, XAR2[AR0]                    ; XAR7 = SwitchTable[Switch]
LB *XAR7                                    ; 用 XAR7 间接跳转
SwitchReturn:
.
.
Switch0:                                    ; 函数 A
.
.
LB SwitchReturn                            ; 长跳转返回
Switch1:                                    ; 函数 B
.
.
LB SwitchReturn                            ; 长跳转返回

```

LB 22bit

长跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LB 22bit	0000 0000 01CC CCCC CCCC CCCC CCCC CCCC	×	—	4

操作数: 22bit 22 位程序地址 (0x000000~0x3FFFFFF)

描述: 长跳转。用 22 位程序地址装载 PC 指针: PC = 22bit;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计

数器 RPTC，并且只执行一次。

例：

```

;跳转到开关值选择的开关表中子程序处
SwitchTable:
    .long Switch0
    .long Switch1
    .
    .
    MOVL XAR2,#SwitchTable
    MOVZ AR0,@Switch
    MOVL XAR7,*+XAR2[AR0]
    LB *XAR7
SwitchReturn:
    .
    .
Switch0:
    .
    .
    LB SwitchReturn
Switch1:
    .
    .
    LB SwitchReturn

```

;开关地址表
;开关 0 的地址
;开关 1 的地址

;XAR2 指针指向开关表
;AR0 开关表指针
;XAR7 = SwitchTable[Switch]
;用 XAR7 间接跳转

;函数 A

;长跳转返回
;函数 B

;长跳转返回

LC *XAR7

间接长调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LC *XAR7	0111 0110 0000 0100	×	-	4

操作数： *XAR7 使用辅助寄存器 XAR7 间接寻址程序存储器，可以寻址整个 4M×16 程序空间范围 (0x000000~0x3FFFFFF)

描述： 间接长调用。返回的 PC 值压入 SP 寄存器指向的软堆栈的两个 16 位数中。接下来将保存在 XAR7 寄存器中的目标地址装载到 PC 中：

```

temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
[SP] = temp(21:16);
SP = SP + 1;
PC = XAR7(21:0);

```

注意：当 OBJMODE = 1 时，为了更有效地调用函数，使用 LCR 和 LRETR 指令代替 LC 和 LRET 指令。

标志与模式： 无

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;调用开关值选择的开关表中的子程序

```

SwitchTable:                ;开关地址表
    .long Switch0            ;开关 0 的地址
    .long Switch1            ;开关 1 的地址
    .
    .
    MOVL XAR2,#SwitchTable    ;XAR2 指针指向开关表
    MOVZ AR0,@Switch          ;AR0 开关表指针
    MOVL XAR7,*+XAR2[AR0]      ;XAR7 = 开关表[Switch]
    LC *XAR7                  ;用 XAR7 间接调用
    .
    .
Switch0:                    ;开关子函数 0
    .
    .
    LRET                      ;返回
Switch1:                    ;开关子函数 1
    .
    .
    LRET                      ;返回

```

LC 22bit 长调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LC 22bit	0000 0000 10CC CCCC CCCC CCCC CCCC CCCC	X	-	4

操作数: 22bit 22 位程序地址 (0x000000~0x3FFFFFF)

描述: 长调用。返回的 PC 值压入 SP 寄存器指定的软堆栈的两个 16 位中。接下来将 22 位立即数的目标地址装载到 PC 中:

```

temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
[SP] = temp(21:16);
SP = SP + 1;
PC = 22bit;

```

注意: 当 OBJMODE = 1 时, 为了更有效地调用函数, 使用 LCR 和 LRETR 指令代替 LC 和 LRET 指令。

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

    ;函数 A 的标准调用
    LC FuncA                ;调用函数 A, 返回地址保存到堆栈
    .
    .
FuncA:                      ;函数 A

```

LRET ;从堆栈地址单元中返回

LCR #22bit 用 RPC 长调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LCR #22bit	0111 0110 01CC CCCC CCCC CCCC CCCC CCCC	1	-	4

操作数: 22bit 22 位程序地址 (0x000000~0x3FFFFFF)
描述: 使用返回 PC 指针 (RPC) 长调用。当前 RPC 值压入 SP 寄存器指定的软堆栈的两个 16 位数中。接下来将 RPC 寄存器装入返回地址。最后 22 位立即数的目标地址装载到 PC 中:

[SP] = RPC(15:0);
SP = SP + 1;
[SP] = RPC(21:16);
SP = SP + 1;
RPC = PC + 2;
PC = 22bit;

注意: LCR 和 LRETR 操作需要 4 个周期调用和 4 个周期返回。标准的 LC 和 LRET 操作为 4 个周期调用和 8 个周期返回。LCR 和 LRETR 操作可以嵌套和代替 LC 和 LRET 操作。在中断中也一样。只有在任务开关操作时, RPC 需要用户保存和恢复。

标志与模式: 无
重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;函数 A 的 RPC 调用
LCR FuncA ;调用函数 A, 返回地址保存到 RPC

FuncA: ;函数 A

LRETR ;RPC 返回

LCR *XARn 用 RPC 间接长调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LCR *XARn	0011 1110 0110 0RRR	1	-	4

操作数: *XARn 使用辅助寄存器 XAR0 和 XAR7 间接寻址程序存储器, 可以访问整个 4M×16 程序空间范围 (0x000000~0x3FFFFFF)
描述: 使用返回 PC 指针 (RPC) 间接长调用。当前 RPC 值压入 SP 寄存器指定的软堆栈两个 16 位数中。RPC 寄存器装入返回地址。保存在 XARn 寄

寄存器中的目标地址装载到 PC 中:

```
[SP] = RPC(15:0);
SP = SP + 1;
[SP] = RPC(21:16);
SP = SP + 1;
RPC = PC + 1;
PC = XARn(21:0);
```

注意: LCR 和 LRETR 操作需要 4 个周期调用和 4 个周期返回。标准的 LC 和 LRET 操作为 4 个周期调用和 8 个周期返回。LCR 和 LRETR 操作可以嵌套和代替 LC 和 LRET 操作。在中断中也一样。只有在任务选择的开关操作方式时, RPC 需要用户保存和恢复。

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;调用开关值选择的开关表中的子程序
SwitchTable:                                ;开关地址表
    .long Switch0                          ;开关 0 的地址
    .long Switch1                          ;开关 1 的地址
    .
    .
    MOVL  XAR2,#SwitchTable                ;XAR2 指针指向开关表
    MOVZ  AR0,@Switch                      ;AR0 开关表指针
    MOVL  XAR6,*+XAR2[AR0]                 ;XAR6 = SwitchTable[Switch]
    LCR   *XAR6                            ;用 XAR6 间接调用
    .
    .
Switch0:                                    ;开关子函数 0
    .
    .
    LRETR                                  ;RPC 返回
Switch1:                                    ;开关子函数 1
    .
    .
    LRETR                                  ;RPC 返回
```

LOOPNZ loc16,#16bit

不等于零则循环

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LOOPNZ loc16,#16bit	0010 1110 LLLL LLLL CCCC CCCC CCCC CCCC	×	—	5N+5

操作数: loc16 寻址方式
 #16bit 16 位立即数 (0x0000~0xFFFF)

描述: 不为 0 则循环:


```
While ([loc16]&16bit! = 0);
```

LOOPNZ 指令使用“位与”操作去比较“loc16”寻址方式指定的值和 16 位立即数。只要操作结果不为 0，本指令可以重复进行比较。处理过程如下：

- 1) 置位状态寄存器 ST1 中的 LOOP 位。
- 2) 产生“loc16”寻址方式指定值的地址。
- 3) 若“loc16”是间接寻址，则对 SP、指定辅助寄存器、AR_{Pn} 指针进行修改。
- 4) 使用“位与”操作比较指定值和立即数值。
- 5) 若结果为 0，LOOP 位清 0，PC 指针加 2。若结果不为 0，则返回到第 1 步。

1~5 步产生的循环可以被硬件中断。当发生中断时，若 LOOPNZ 指令一直有效，则在堆栈中保存指向 LOOPNZ 指令的返回地址。因此，从中断返回后，LOOPNZ 指令被重新取用。

“与”操作结果不为 0 时，LOOPNZ 指令在流水线的解码阶段 2 每五个周期重新开始。因此每五个周期读一次存储器地址或寄存器。若“loc16”为间接寻址，可以指定指针（SP 或辅助寄存器）的增加或减小。在每次流水线的解码阶段 2 都会修改指针。这就意味着立即数每次都和新的保存数据值比较。

LOOPNZ 指令不会从流水线中刷新预取指令。然而当发生中断时，则会刷新预取指令。

当发生中断时，保存 ST1 的 LOOP 位的当前状态到堆栈中。ST1 中的 LOOP 位清 0。LOOP 位是一个被动的状态位。LOOPNZ 指令改变 LOOP，但是 LOOP 不会影响指令。

在中断服务程序中，可以终止 LOOPNZ 指令。测试保存在堆栈中的 LOOP 位，若为 1，则增加（加 2）堆栈中的返回地址。从中断中返回时，增加后的地址装载到 PC 中，执行 LOOPNZ 的下一条指令。

标志与模式： Z 若“与”操作结果的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

N 若“与”操作结果的第 15 位的值为 1，则负标志位 N 置 1；否则清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;向 RegB 写数之前，必须等待 RegA 的第 3 位清 0
LOOPNZ  @RegA, #0x0004      ;在 (RegA AND 0x0004 != 0) 时，循环
MOV      @RegB, #0x8000      ;RegB = 0x8000
```

LOOPZ loc16,#16bit

等于零循环

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LOOPZ loc16,#16bit	0010 1110 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	5N+5

操作数: loc16 寻址方式

#16bit 16 位立即数 (0x0000~0xFFFF)

描述: 不为 0 则循环。

While ([loc16]&16bit = 0);

LOOPZ 指令使用“位与”操作去比较“loc16”寻址方式指定的值和 16 位立即数。只要操作结果为 0, 本指令可以重复进行比较。处理过程如下:

- 1) 置位状态寄存器 ST1 中的 LOOP 位。
- 2) 产生“loc16”寻址方式指定值的地址。
- 3) 若“loc16”是间接寻址的操作数, 则对 SP 或指定辅助寄存器和 (或) ARPN 指针进行修改。
- 4) 使用“位与”操作比较值和立即数。
- 5) 若结果不为 0, LOOP 位清 0, PC 指针增加 2。若结果为 0, 则返回第 1 步。

1~5 步产生的循环可以被硬件中断。当发生中断时, 若 LOOPZ 指令一直有效, 则在堆栈中保存指向 LOOPZ 指令的返回地址。因此, 从中断返回后, LOOPZ 指令被重新取用。

“与”操作结果不为 0 时, LOOPZ 指令在流水线的解码阶段 2 每五个周期重新开始。因此每五个周期读一次存储器地址或寄存器。若“loc16”间接寻址, 可以指定指针 (SP 或辅助寄存器) 的增加或减小。在每次流水线的解码阶段 2 都会修改指针。这就意味着立即数每次都和新的保存数据值比较。

LOOPNZ 指令不会从流水线中刷新预取指令。然而当发生中断时, 则会刷新预取指令。

当发生中断时, 保存 ST1 的 LOOP 位的当前状态到堆栈中。ST1 中的 LOOP 位清 0。LOOP 位是一个被动的状态位。LOOPZ 指令改变 LOOP, 但是 LOOP 不会影响指令。

在中断服务程序中, 可以终止 LOOPZ 指令。测试保存在堆栈中的 LOOP 位, 若为 1, 则增加 (加 2) 堆栈中的返回地址。从中断中返回时, 增加后的地址装载到 PC 中, 执行 LOOPZ 的下一条指令。

标志与模式: Z 若“与”操作的结果为 0, 则零标志位 Z 置 1; 否则零标志位 Z 清 0。

N 若结果的第 15 位数为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。
LOOP 操作结果为 0 时, LOOP 可以重复置位。当结果不为 0 时, LOOP 清 0。若在 LOOP 指令进入流水线的解码阶段 2 之前发生中断,

指令将会从流水线中终止，因此不会影响 LOOP 位。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; 向 RegB 写数之前，必须等待 RegA 的位 3 置位

LOOPZ @RegA, #0x0004 ; 在 (RegA AND 0x0004 = 0) 时，循环

MOV @RegB, #0x8000 ; RegB = 0x8000

LPADDR

置 AMODE 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LPADDR	0101 0110 0001 1110	×	-	4

操作数： 无

描述： AMODE 状态位置 1，使器件进入 C2xLP 兼容的寻址方式。

注意： 指令不会刷新流水线。

标志与模式： AMODE AMODE 位置 1。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; 用 C2xLP 语法：写程序执行操作 "VarC = VarA + VarB"

LPADDR ; 完全 C2xLP 寻址兼容模式

.lp_amode ; 告诉汇编器处于 C2xLP

LDP #VarA ; 初始化 DP (仅低 64K)

LACL VarA ; ACC = VarA (ACC 高 = 0)

ADDS VarB ; ACC = ACC + VarB (无符号数)

SACL VarC ; 结果保存到 VarC

C28ADDR ; 返回 C28x 寻址方式

.c28_amode ; 告诉汇编器处于 C28x 模式

LRET

长返回

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LRET	0111 0110 0001 0100	×	-	8

操作数： 无

描述： 长返回。返回地址从软堆栈中弹出两个 16 位数到 PC 中：

```
SP = SP - 1;
temp(31:16) = [SP];
SP = SP - 1;
temp(15:0) = [SP];
PC = temp(21:0);
```

标志与模式： 无

注意：当 OBJMODE = 1 时，为了更有效地调用函数，使用 LCR 和 LRETR 指令代替 LC 和 LRET 指令。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;函数 A 的标准调用
LC FuncA                      ;调用函数 A，返回地址保存到堆栈中
.
.
FuncA:                        ;函数 A
.
LRET                          ;从堆栈地址单元中返回

```

LRETE 长返回并中断使能

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LRETE	0111 0110 0001 0000	X	-	8

操作数： 无

描述： 长返回并中断使能。返回地址从软堆栈中弹出两个 16 位数到 PC 中。全局中断标志位 (INTM) 清 0。全局可屏蔽中断使能：

```

SP = SP - 1;
temp(31:16) = [SP];
SP = SP - 1;
temp(15:0) = [SP];
PC = temp(21:0);
INTM = 0;

```

标志与模式： INTM INTM 位清 0，将中断使能。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;函数 A 的标准调用。在进入时禁止中断，退出时中断使能
LC FuncA                      ;调用函数 A，返回地址保存到堆栈中
.
.
FuncA:                        ;函数 A
SETC    INTM                  ;禁止中断
.
.
LRETE                          ;从堆栈地址单元中返回，中断使能

```

LRETR 使用 RPC 长返回

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LRETR	0000 0000 0000 0110	1	-	4

操作数: 无

描述: 使用返回 PC 指针 (RPC) 长返回。将保存在 RPC 寄存器中的返回地址装载到 PC 中。RPC 寄存器装载软堆栈中的两个 16 位数

```

SP = RPC;
SP = SP-1;
temp(31:16) = [SP];
SP = SP - 1;
temp(15:0) = [SP];
RPC = temp(21:0);

```

注意: LCR 和 LRETR 操作需要 4 个周期调用和 4 个周期返回。标准的 LC 和 LRET 操作为 4 个周期调用和 8 个周期返回。LCR 和 LRETR 操作可以嵌套和代替 LC 和 LRET 操作。在中断中也一样。只有在任务开关操作时, RPC 需要用户保存和恢复。

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;函数 A 的 RPC 调用
LCR FuncA                      ;调用函数 A, 返回地址保存到堆栈中
.
.
FuncA:                          ;函数 A
.
.
LRETR                          ;RPC 返回

```

LSL ACC,#1...16

逻辑左移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL ACC,#1...16	1111 1111 0011 SHFT	×	Y	N+1

操作数: ACC 累加器
#1...16 移位位数

描述: ACC 累加器中的值逻辑左移给定的移位位数。在移位时, ACC 的低位作零扩展, 移出的最后一位存到进位标志位 C 中:



标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置 1; 否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则零标志位 Z 清 0。

C 移出 ACC 的最后一位保存在 C 中。

重复性：本指令可以重复。若指令跟在 RPT 指令之后，则 LSL 指令将执行 N+1 次。Z、N、C 的状态为最后的结果。

例：

```

;VarA 逻辑左移 4 位
MOVL  ACC,@VarA          ;ACC = VarA
LSL    ACC,#4             ;VarA 逻辑左移 4 位
MOVL  @VarA,ACC          ;结果保存到 VarA

```

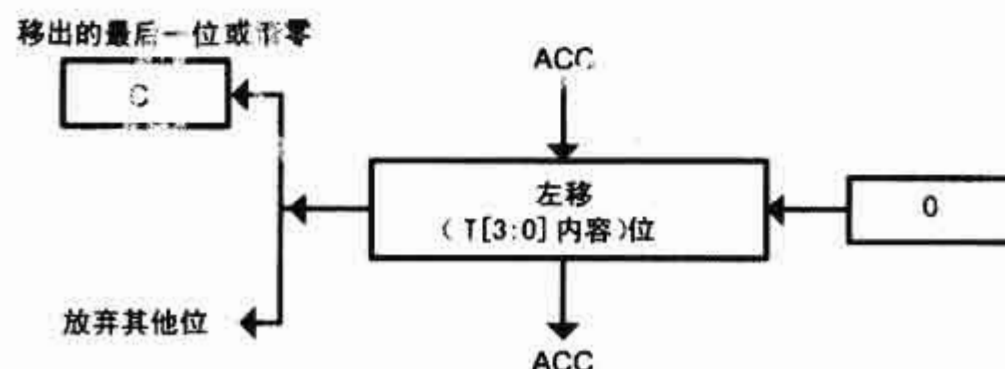
LSL ACC,T 逻辑左移 T (3:0) 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL ACC,T	1111 1111 0101 0000	×	-	1

操作数：ACC 累加器

T 被乘数寄存器 (XT) 的高 16 位

描述：按给定的移位位数移位 ACC 累加器中的值，由 T 寄存器的低 4 位 T(3:0) = 0...15 指定移位位数，忽略 T 寄存器的高位。在移位时，ACC 的最低位填 0。若指定移位位数为 0，则进位标志位 C 清 0；否则最后移出 ACC 累加器的那一位保存到进位标志位 C 中：



标志与模式：N 若 ACC 的第 31 位为 1，则负标志位 N 置 1；否则负标志位 N 清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置 1；否则清 0。即使 T 寄存器指定移位位数为 0，还是会测试 ACC 累加器的值是否为 0，从而影响零标志位 Z。

C 若 T (3:0) = 0，进位标志位 C 清 0；否则移出的最后一位装载到 C 中。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

;VarA 的值逻辑左移 VarB 中的位数
MOVL  ACC,@VarA          ;ACC = VarA
MOV    T,@VarB            ;T = VarB (移位位数)
LSL    ACC,T              ;ACC 逻辑左移 T(3:0) 位

```

MOVL @VarA, ACC

; 结果保存到 VarA

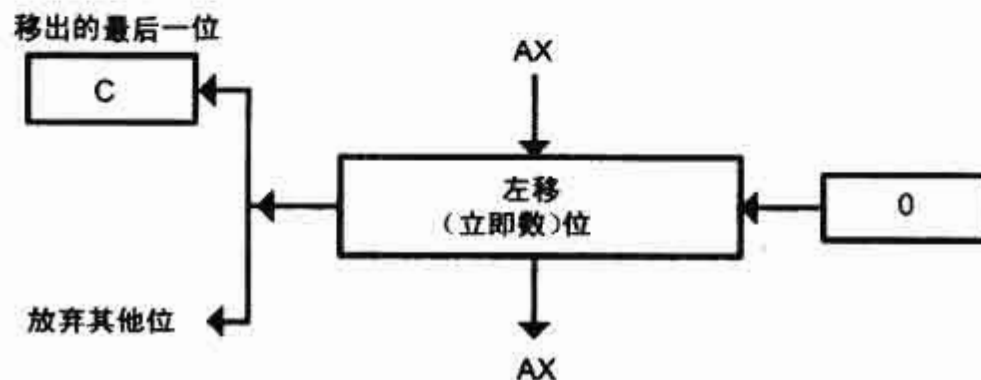
LSL AX, #1...16

逻辑左移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL AX, #1...16	1111 1111 100A SHFT	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
#1...16 移位位数

描述: 由“移位位数”区域给定的数在指定 AX 寄存器 (AH 或 AL) 内容上执行逻辑左移指令。移位时, AX 寄存器的最低位填 0, 移出的最后一位保存在进位标志位 C 中:



标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 AX 为 0。则零标志位 Z 置位; 否则清 0。

C 移出 AH 或 AL 的最后一位保存在 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

; 多指针寄存器 AR0 乘以 2

MOV AL, @AR0 ; AR0 装载到 AL 中

LSL AL, #1 ; 左移 1 位相当于乘 2

MOV @AR0, AL ; 将结果保存回 AR0 中

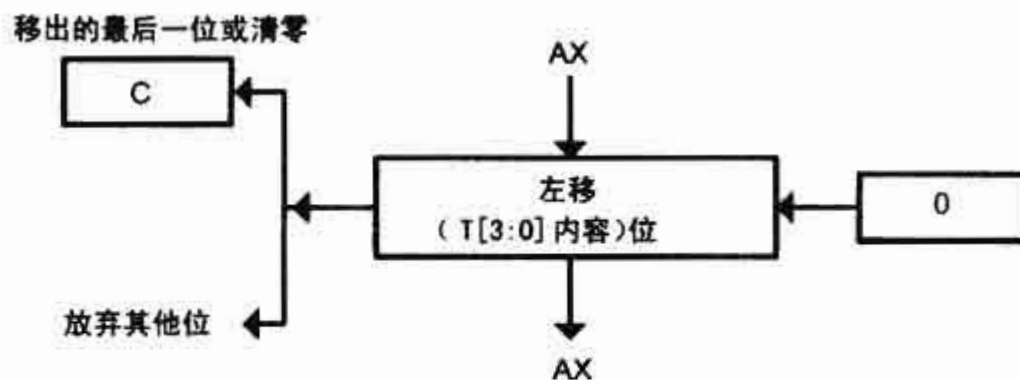
LSL AX, T

逻辑左移 T (3:0) 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL AX, T	1111 1111 0110 011A	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
T 被乘数 (XT) 寄存器的高 16 位

描述: 由 T 寄存器的最低 4 位 T(3:0) 的值指定 AX 寄存器的内容执行逻辑左移的位数, T 寄存器的高位忽略。移位时, AX 寄存器的最低位填 0。若 T(3:0) 指定移位位数为 0, 则进位标志位 C 清 0; 否则 AX 移出的最后一位到进位标志位 C 中:



- 标志与模式:** N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。即使 T(3:0) 寄存器指定移位位数为 0, 还是要测试 AH 或 AL 的值是否为负, 而且会影响负标志位 N。
- Z 若 AX 为 0。则零标志位 Z 置位, 否则清 0。即使 T(3:0) 寄存器指定移位位数为 0, 仍要测试 AH 或 AL 的值是否为 0, 而影响零标志位 Z。
- C 若 T(3:0) 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则从 AH 或 AL 移出的最后一位到 C 中。
- 重复性:** 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

;计算值 VarC = VarA << VarB
MOV     T, @VarB           ;用 VarB 的内容装载 T 寄存器
MOV     AL, @VarA          ;用 VarA 的内容装载 AL 寄存器
LSL     AL, T               ;依据 T 寄存器的 0:3 位的值左移 AL
MOV     @VarC, AL          ;结果保存在 VarC 中

```

LSL64 ACC: P, #1...16

逻辑左移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL64 ACC: P, #1...16	0101 0110 1010 SHFT	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)
#1...16 移位位数

描述: 按“移位位数”逻辑左移 ACC:P 寄存器的 64 位值。移位时, 最低位填 0, 移出的最后一位保存在进位标志位 C 中:



- 标志与模式: N 若 ACC 累加器的第 31 位为 1, 则 ACC:P 为负, 负标志位 N 置位; 否则负标志位 N 清 0。
- Z 若 ACC:P 的 64 位复合值为 0。则零标志位 Z 置位, 否则清 0。
- C 若 $T(5:0) = 0$, 则进位标志位 C 清 0; 否则复合的 64 位数移出的最后一位到进位标志位 C 中。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

;用 VAR16 中的数指定 64 位宽度的 VAR64 逻辑左移位数
MOVL  ACC, @Var64+2      ;由 VAR64 的高 32 位装载 ACC
MOVL  P, @Var64+0        ;由 VAR64 的低 32 位装载 P
MOV   T, @Var16          ;Var16 中的移位位数装载 T
LSL64 ACC:P, T           ;由 T(5:0) 指定 ACC:P 逻辑左移的位数
MOVL  @Var64+2, ACC      ;保存结果的高 32 位到 Var64
MOVL  @Var64+0, P        ;保存结果的低 32 位到 Var64

```

LSL64 ACC: P, T

64 位数逻辑左移 T (5:0) 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSL64 ACC: P, T	0101 0110 0101 0010	1	-	1

操作数: ACC: P 累加器 (ACC) 和乘积寄存器 (P)

T 被乘数 (XT) 寄存器的高 16 位

描述: 逻辑左移 ACC:P 寄存器的 64 位的值。左移位数由 T 寄存器的最低 6 位 $T(5:0) = 0...63$ 指定, T 寄存器的高位忽略, ACC:P 寄存器的最低位填 0。若 T 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则 ACC:P 寄存器移出的最后一位到进位标志位 C 中:



- 标志与模式: N 若 ACC 的第 31 位为 1, 则 ACC:P 为负, 负标志位 N 置位; 否则清 0。
- Z 若 ACC:P 值为 0, 则零标志位 Z 置位; 否则清 0。
- C 若 $T(5:0) = 0$, 则进位标志位 C 清 0, 否则 64 位的值移出的最后一位装载到 C 中。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 64 位的 Var64 逻辑左移 Var16 的值指定的位数
MOVL  ACC, @Var64+2      ; 用 Var64 的高 32 位装载 ACC
MOVL  P, @Var64+0        ; 用 Var64 的低 32 位装载 P
MOV   T, @Var16          ; 用 Var16 中的移位位数装载 T
LSL64 ACC: P, T          ; ACC: P 逻辑左移 T(5:0) 位
MOVL  @Var64+2, ACC      ; 保存结果的高 32 位到 Var64 中
MOVL  @Var64+0, P        ; 保存结果的低 32 位到 Var64 中
    
```

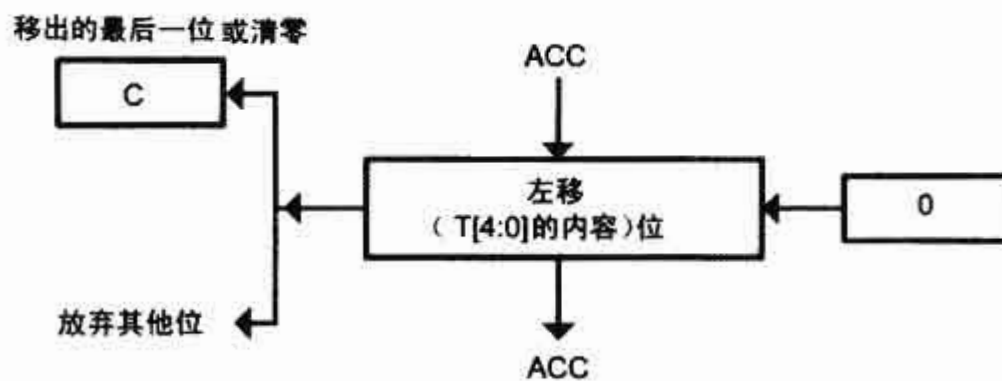
LSLL ACC,T

逻辑左移 T (4:0) 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSLL ACC,T	0101 0110 0011 1011	1	-	1

操作数: ACC 累加器
T 被乘数 (XT) 寄存器的高 16 位

描述: 逻辑左移 ACC 累加器的值。左移位数由 T 寄存器的最低 5 位 T (4:0) = 0...31 指定, T 寄存器的高位忽略, ACC 累加器的最低位填 0。若 T 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则 ACC 累加器移出的最后一位到进位标志位 C 中:



标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。虽然 T 寄存器指定移位位数为 0, 还是要测试 ACC 累加器是否为 0, 且影响零标志位 Z。

N 若 ACC 的第 31 位为 1, 则 ACC 为负, 负标志位 N 置位; 否则清 0。虽然 T 寄存器指定移位位数为 0, 还是要测试 ACC 累加器是否为负, 且影响负标志位 N。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 由 VarB 的值指定 VarA 的内容逻辑左移位
MOVL  ACC, @VarA          ; ACC = VarA
MOV   T, @VarB            ; T = VarB (移位位数)
LSLL  ACC, T              ; ACC 逻辑左移 T(4:0) 位
MOVL  @VarA, ACC          ; 结果保存到 VarA 中
    
```

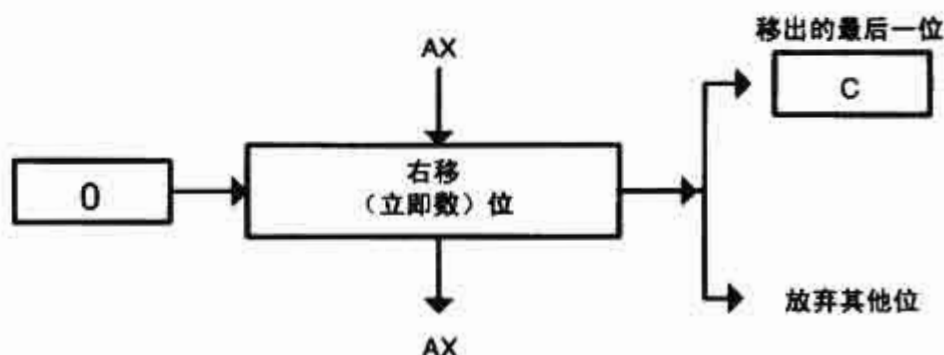

LSR AX, #1...16

逻辑右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSR AX, #1...16	1111 1111 110A SHIFT	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
#1...16 移位位数

描述: 对 AX 寄存器内容执行逻辑右移, 右移位数由“移位位数”给定。移位时, AX 寄存器的高位填充 0, 移出的最后一位保存在进位标志位 C 中:



标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 AX 为 0。则零标志位 Z 置位, 否则清 0。

C 移出 AH 或 AL 的最后一位保存在进位标志位 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

; 指针寄存器 AR0 除以 2

MOV AL, @AR0 ; AR0 的内容装载到 AL 中

LSR AL, #1 ; 除以 2

MOV @AR0, AL ; 将结果存回 AR0 中

LSR AX, T

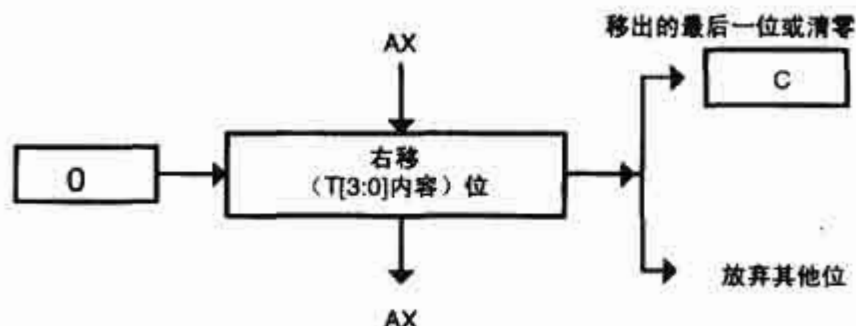
按 T (3:0) 的位数逻辑右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSR AX, T	1111 1111 0110 001A	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器

T 被乘数 (XT) 寄存器的高 16 位

描述: 对 AX 寄存器的内容执行逻辑右移指令, 右移位数由 T 寄存器的最低 4 位 T (3:0) 指定, T 寄存器的高位忽略。移位时, AX 寄存器的高位填充 0。若 T (3:0) 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则 AX 移出的最后一位存到进位标志位 C 中:



标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。即使 T(3:0) 寄存器指定移位位数为 0, 还是要测试 AH 或 AL 的值是否为负, 而且会影响负标志位 N。

Z 若 AX 的值为 0。则零标志位 Z 置位, 否则清 0。即使 T (3:0) 寄存器指定移位位数为 0, 还是要测试 AH 或 AL 的值是否为 0, 而且会影响零标志位 Z。

C 若 T (3:0) 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则 AH 或 AL 移出的最后一位到进位标志位 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```
;计算无符号数 VarC = VarA >>VarB
MOV    T, @VarB      ;用 VarB 的内容装载 T
MOV    AL, @VarA      ;用 VarA 的内容装载 AL
LSR    AL, T          ;依据 T 的 0: 3 位的值右移 AL
MOV    @ VarC, AL     ;结果保存在 VarC 中
```

LSR64 ACC:P,#1...16

64 位逻辑右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSR64 ACC:P,#1...16	0101 0110 1001 SHFT	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)

#1...16 移位位数

描述: 用“移位位数”给定的值逻辑右移 ACC:P 寄存器的 64 位值。移位时, 最高有效位填充 0, 移出的最后一位存到进位标志位 C 中:



标志与模式: N 若 ACC 累加器的第 31 位为 1, 则 ACC:P 为负, 负标志位 N 置位; 否则清 0。

Z 若 ACC:P 的 64 位复合值为 0，则零标志位 Z 置位，否则清 0。

C 复合 64 位数移出的最后一位装载到进位标志位 C 中。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

例：

； 逻辑右移 10 位的 64 位的 VAR64

MOVL ACC,@Var64+2 ; 由 VAR64 的高 32 位装载 ACC

MOVL P,@Var64+0 ; 由 VAR64 的低 32 位装载 P

LSR64 ACC:P,#10 ; ACC:P 逻辑右移 10 位

MOVL @Var64+2,ACC ; 保存结果的高 32 位到 Var64

MOVL @Var64+0,P ; 保存结果的低 32 位到 Var64

LSR64 ACC:P,T

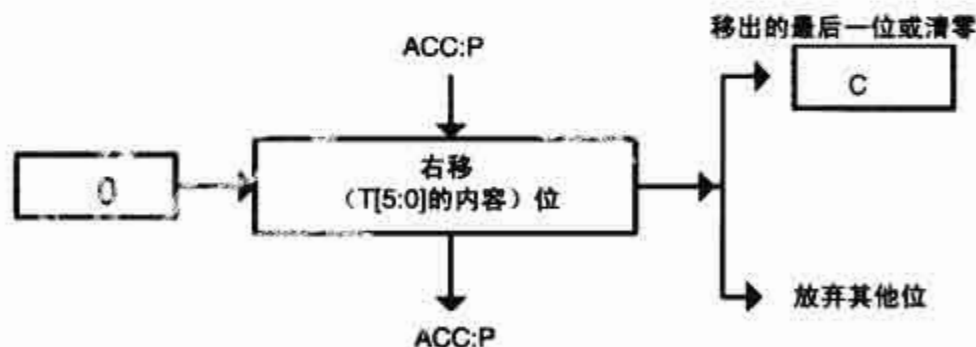
64 位数逻辑右移 T (5:0) 位

语 法 选 项	操 作 码	OBJ 模 式	RPT	CYC
LSR64 ACC: P,T	0101 0110 0101 1011	1	-	1

操作数： ACC:P 累加器 (ACC) 和乘积寄存器 (P)

T 被乘数寄存器 (XT) 的高 16 位

描述： 对 ACC 累加器的值执行逻辑右移，右移位数由 T 寄存器的最低 6 位 T(5:0) = 0...63 指定，T 寄存器的高位忽略。移位时，ACC 累加器的最高有效位填充 0。若 T 寄存器指定移位位数为 0，则进位标志位 C 清 0；否则 ACC: P 寄存器移出的最后一位存到进位标志位 C 中：



标志与模式： **N** 若 ACC 累加器的第 31 位为 1，则 ACC:P 为负，负标志位 N 置位；否则清 0。

Z 若 ACC:P 的 64 位复合值为 0。则零标志位 Z 置位，否则清 0。

C 若 T(5:0) = 0，则进位标志位 C 清 0；否则复合 64 位数移出的最后一位存在进位标志位 C 中。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

例：

； 由 VAR16 的值指定算术右移 64 位的 VAR64 的位数

MOVL ACC,@Var64+2 ; 由 VAR64 的高 32 位装载 ACC

MOVL P,@Var64+0 ; 由 VAR64 的低 32 位装载 P

MOV T,@Var16 ; 用 Var16 的移位位数装载 T

LSR64 ACC:P,T ; ACC:P 逻辑右移 T(5:0) 位

```

MOVL @Var64+2, ACC      ; 保存结果的高 32 位到 Var64
MOVL @Var64+0, P        ; 保存结果的低 32 位到 Var64

```

LSRL ACC, T 逻辑右移 T(4:0)位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
LSRL ACC, T	0101 0110 0010 0010	1	-	1

操作数: ACC 累加器

T 被乘数 (XT) 寄存器的高 16 位

描述: 对 ACC 累加器的值执行逻辑右移, 右移位数由 T 寄存器的最低 5 位 T(4:0) = 0...31 指定, T 寄存器的高位忽略。移位时, ACC 累加器的高位填充 0。若 T 寄存器指定移位位数为 0, 则进位标志位 C 清 0; 否则 ACC 累加器移出的最后一位存到进位标志位 C 中:



标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。即使 T 寄存器指定移位位数为 0, 还是要测试 ACC 累加器的值是否为 0, 而且影响零标志位 Z。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位, 否则清 0。即使 T 寄存器指定移位位数为 0, 还是要测试 ACC 累加器的值是否为负, 而且影响负标志位 N。

C 若 T(4:0) = 0, 则进位标志位 C 清 0; 否则复合 64 位数移出的最后一位存到 C 中。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 由 VarB 的值指定逻辑右移 VarA 内容的位数
MOVL ACC, @VarA      ; ACC = VarA
MOV   T, @VarB        ; T = VarB (移位位数)
LSRL  ACC, T          ; ACC 逻辑右移 T(4:0) 位
MOVL  @VarA, ACC      ; 结果保存到 VarA 中

```

MAC P, loc16, 0:pma

累加且乘积

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MAC P, loc16, 0: pma	0001 0100 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	N+2

- 操作数:** P 乘积寄存器
loc16 寻址模式
0: pma 立即程序存储器地址, 仅访问低 64K 的程序空间区域 (0x000000~0x00FFFF)
- 描述:** 1) 先前的乘积 (保存在 P 寄存器中) 按乘积移位方式 (PM) 指定的移位位数移动后加到 ACC 累加器。
2) 用 “loc16” 寻址模式的地址单元的内容装载 T 寄存器。
3) 用程序存储器指定地址单元的有符号 16 位数乘以 T 寄存器的有符号 16 位数, 结果保存到 32 位的 P 寄存器。
- $$ACC = ACC + P \ll PM;$$
- $$T = [loc16];$$
- $$P = signed\ T \times signed\ Prog[0x00:pma];$$
- 当使用 MAC 指令时, C28x 迫使由 “0: pma” 寻址模式指定的程序存储器地址的高 6 位为 0x00。这限定程序存储器地址为程序地址空间的低 64K (0x000000~0x00FFFF)。在 C28x 器件上, 存储器 (同一物理存储器) 即可以映射到程序也可以映射到数据空间。因此, “0: pma” 寻址模式能够用于存取数据空间的变量。
- 标志与模式:** Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
C 若产生进位, 则进位标志位 C 置位; 否则清 0。
V 若产生溢出, 则溢出标志位 V 置位; 否则清 0。
OVC 若禁止溢出模式, 操作产生正溢出, 计数器值增加; 操作产生负溢出, 计数器值减少。
OVM 若溢出模式位置 1, 当运行发生溢出时, ACC 的值将为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
PM PM 值设置了乘积寄存器输出时的移位操作模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。
- 重复性:** 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。当发生溢出时, 则 V 标志位置 1。在每次重复操作时, 程序存储器地址加 1。

例:

```

;使用 16 位的乘法计算乘积和
;int16 X[N];数据信息
;int16 C[N];系数信息, 位于低 64K
;sum=0;
;for (i=0, I<N; I+ +)
;sum = sum + (X[i] × C[i]) >> 5;
MOVL  XAR2, #X          ;XAR2 = X
SPM   -5                ;设置乘积右移 5 位

```



```

ZAPA                ;ACC, P, OVC 清 0
RPT #N-1            ;重复下一条指令 N 次
||MAC P,*XAR2+ +,0:C ;ACC = ACC + P >> 5,
; P = *XAR2+ + × *C+ +
ADDL ACC,P << PM    ;完成最后的累加
MOVL @sum,ACC       ;保存最终结果到 sum

```

MAC P,loc16,*XAR7/+ +

累加且乘积

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MAC P,loc16,*XAR7	0101 0110 0000 0111 1100 0111 LLLL LLLL	1	Y	N+2
MAC P,loc16,*XAR7+ +	0101 0110 0000 0111 1000 0111 LLLL LLLL	1	Y	N+2

操作数: P 乘积寄存器
loc16 寻址模式
*XAR7/+ + 使用辅助寄存器 XAR7 的间接寻址程序存储器的全部 4M×16 的程序空间 (0x000000~0x3FFFFFF)

描述: 本指令采用以下步骤:
1) 先前的乘积 (保存在 P 寄存器中) 按乘积移位方式 (PM) 指定的移位位数移位后加到 ACC 累加器。
2) 用 “loc16” 寻址地址单元的内容装载 T 寄存器。
3) 用 XAR7 寄存器指向的程序存储器地址单元中的有符号 16 位数乘以 T 寄存器的有符号 16 位数, 32 位结果保存到 P 寄存器中。而且 XAR7 寄存器加 1。

```

ACC = ACC + P << PM;
T = [loc16];
P = signed T × signed Prog[*XAR7 or *XAR7++];

```

在 C28x 器件上, 存储器模块 (同一个物理存储器) 映射到程序和数据空间, 因此, “XAR7/+ +” 寻址模式能够用于存取落入其寻址范围内的数据空间变量。

一些寻址模式组合后, 将得到一些有争议的寻址模式。这种情况下, C28x 将参考 XAR7 而指定 “loc16/loc32” 寻址的优先级。例如:

```

MAC P,*--XAR7,*XAR7+ +; --XAR7 约定的高优先级
MAC P,*XAR7+ +,*XAR7 ; *XAR7+ + 约定的高优先级
MAC P,*XAR7,*XAR7+ + ; *XAR7+ + 约定的高优先级

```

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
C 若产生进位, 则进位标志位 C 置位; 否则清 0。
V 若产生溢出, 则溢出标志位 V 置位; 否则 V 不受影响。
OVC 若禁止溢出模式, 操作产生正溢出, 计数器值增加; 操作产生负

溢出，计数器值减少。

OVM 若溢出模式位置 1，当运行发生溢出时，ACC 值将为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。

PM PM 值设置了乘积寄存器输出时的移位操作模式。若乘积移位位数为正（逻辑左移操作），最低位填 0；若乘积移位位数为负（逻辑右移操作），则高位进行符号位扩展。

重复性： 本指令可以重复。若指令跟在 RPT 指令之后，它将执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。发生溢出时，V 标志位立即置 1。

例：

```

;使用 16 位的乘法计算乘积和
;int16 X[N];数据信息
;int16 C[N];系数信息，位于 4M 低端
;sum=0;
;for (i=0, i<N;i+ +)
;sum = sum + (X[i] × C[i]) >> 5;
MOVL  XAR2, #X           ;XAR2 为指针，指向 X
MOVL  XAR7, #C           ;XAR7 为指针，指向 C
SPM   -5                 ;设置乘积右移 5 位
ZAPA                      ;ACC, P, OVC 清 0
RPT   #N-1               ;重复下一条指令 N 次
||MAC P, *XAR2+ +, *XAR7+ + ;ACC = ACC + P >> 5,
                           ;P = *XAR2+ + × * XAR7+ +
ADDL  ACC, P << PM       ;完成最后的累加
MOVL  @sum, ACC          ;保存最终结果到 sum
```

MAX AX,loc16

求最大值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MAX AX,loc16	0101 0110 0111 001A 0000 0000 LLLL LLLL	1	Y	N+1

操作数： AX 累加器高（AH）或累加器低（AL）寄存器
loc16 寻址模式

描述： 将 AX 寄存器（AH 或 AL）的有符号值与“loc16”寻址地址单元的有符号值进行比较后，将两个值中大的一个值装载到 AX 寄存器。

```

if (AX < [loc16]), AX = [loc16];
if (AX >= [loc16]), AX 不变;
```

标志与模式： N 若 AX 小于指定地址单元的值（AX<[loc16]），则负标志位 N 置位；否则清 0。
Z 若 AX 等于指定地址单元的值（AX=[loc16]），则零标志位 Z 置位；否则清 0。
V 若 AX 大于指定地址单元的值（AX>[loc16]），则溢出标志位置位。但是本指令不能对 V 清 0。

重复性： 本指令可以重复，若指令跟在 RPT 指令之后，它将执行 N+1 次。Z、N 和 V 的状态为最终结果。

例：

```
;使 VarA 为饱和值
;if(VarA > 2000) VarA = 2000;
;if(VarA < -2000) VarA = -2000;
MOV AL,@VarA           ;用 VarA 值装载 AL
MOV @AH,#2000          ;用 2000 装载 AH
MIN AL,@AH             ;if(AL > AH) AL = AH
NEG AH                 ;AH = -2000
MAX AL,@AH             ;if(AL < AH) AL = AH
MOV @VarA,AL           ;结果保存到 VarA 中
```

MAXCUL P,loc32

条件求无符号的最大值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MAXCUL P,loc32	0101 0110 0101 0001 0000 0000 LLLL LLLL	1	-	1

操作数： P 乘积寄存器
loc32 寻址模式

描述： 根据负标志位 N 和零标志位 Z 的状态，有条件地把 P 寄存器的无符号值与“loc32”寻址地址单元的 32 位无符号值进行比较，然后将两个中的最大值装载到 P 寄存器：

```
if( (N = 1) & (Z = 0) )
P = [loc32];
if( (N = 0) & (Z = 1) & (P < [loc32]) )
V = 1, P = [loc32];
if( (N = 0) & (Z = 0) )
P = 不变;
```

注意： “P < [loc32]” 的操作处理与一个 32 位的无符号数比较类似。

本指令是与 MAXL 指令结合形成一个 64 位求最大值的典型指令。假定使用 MAXL 指令去比较 64 位数的高 32 位，负标志位 N 和零标志位 Z 将首先置位。MAXCUL 指令根据高 32 位的比较结果条件比较低 32 位。

标志与模式： N 若 (N = 1 且 Z = 0)，则用[loc32]装载 P。
Z 若 (N = 0 且 Z = 1)，把 P 寄存器的无符号值与无符号值[loc32]进行比较，将两个中最大值装载到 P 寄存器。若 (N = 0 且 Z = 0)，将不作任何处理。
V 若 (N = 0, Z = 1 且 P<[loc32])，则溢出标志位 V 置位；否则 V 不变。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器的值 RPTC，并且只执行一次。

例:

```

;使 64 位 VarA 为饱和值如下:
;if(Var64 > MaxPos64 ) Var64 = MaxPos64
;if(Var64 < MaxNeg64 ) Var64 = MaxNeg64
MOVL ACC,@Var64+2      ;用 Var64 装载 ACC:P
MOVL P,@Var64+0
MINL ACC,@MaxPos64+2    ;if(ACC:P > MaxPos64) ACC:P = MaxPos64
MINCUL P,@MaxPos64+2
SB saturate,OV
MAXL ACC,@MaxNeg64+2    ;if(ACC:P < MaxNeg64) ACC:P = MaxNeg64
MAXCUL P,@MaxNeg64+0
Saturate:
MOVL @Var64+2,ACC      ;结果保存到 Var64 中
MOVL @Var64,P

```

MAXL ACC,loc32

求 32 位最大值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MAXL ACC,loc32	0101 0110 0110 0001 0000 0000 LLLL LLLL	I	Y	N+1

操作数: ACC 累加器
loc32 寻址模式

描述: 把 ACC 累加器的值与 “loc32” 寻址模式指向的 32 位值进行比较, 将最大值装载到 P 寄存器:

```

if(ACC < [loc32]), ACC = [loc32];
if(ACC >= [loc32]), ACC = 不变;

```

标志与模式: Z 若 ACC 等于指定地址中的值 (ACC = [loc32]), 则零标志位 Z 置位, 否则清 0。

N 若 ACC 小于指定地址中的值 (ACC < [loc32]), 则负标志位 N 置位, 否则清 0。当指定结果的符号时, 假设 MAXL 指令有无限位的精度, 例如, 当精度限制在 32 位时, 减法 0x8000 0000-0x0000 0001, 将引起溢出结果为正数 0x7FFF FFFF, 且负标志位 N 清 0。然而, 当假定 MAXL 指令为无限精度, 则将则负标志位 N 置位表明 0x8000 0000-0x0000 0001 事实上将产生一个负数。

C 若 (ACC-[loc32]) 产生借位, 将进位标志位 C 清 0。否则进位标志位 C 置位。

V 若 ACC 大于地址单元中的值 (ACC > [loc32]), 则溢出标志位 V 置位; 否则 V 不受影响。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, MAXL 指令将执行 N+1 次。Z、N、C 的状态为最终结果。若发生溢出时, V 标志位将立即置位。

例:

```

;使 VarA 为饱和值
;if(VarA > MaxPos) VarA = MaxPos

```



```

; if (VarA < MaxNeg) VarA = MaxNeg
MOVL ACC, @VarA          ; ACC = VarA
MINL ACC, @MaxPos        ; if (ACC > MaxPos) ACC = MaxPos
MAXL ACC, @MaxNeg        ; if (ACC < MaxNeg) ACC = MaxNeg
MOVL @VarA, ACC          ; 结果保存到 VarA 中

```

MIN AX, loc16

求最小数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MIN AX, loc16	0101 0110 0111 010A 0000 0000 LLLL LLLL	1	Y	N+1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
loc16 寻址模式

描述: 将寄存器 AX (AH 或 AL) 的有符号值与 “loc16” 地址单元的有符号值进行比较, 最小值装载到 AX 寄存器。

```

if (AX > [loc16]), AX = [loc16];
if (AX <= [loc16]), AX 不变;

```

标志与模式: N 若 AX 小于指定地址中的值 ($AX < [loc16]$), 则负标志位 N 置位; 否则清 0。

Z 若 AX 等于指定地址中的值 ($AX = [loc16]$), 则零标志位 Z 置位; 否则清 0。

V 若 AX 大于指定地址中的值 ($AX > [loc16]$), 则溢出标志位 V 置位。但是本指令不能对 V 清 0。

重复性: 本指令可以重复, 若指令跟在 RPT 指令之后, 将执行 N+1 次。Z、N、V 的状态为最终结果。

例:

```

; 使 VarA 为饱和值
; if (VarA > 2000) VarA = 2000
; if (VarA < -2000) VarA = -2000
MOV AL, @VarA          ; 用 VarA 值装载 AL
MOV @AH, #2000         ; 用 2000 装载 AH
MIN AL, @AH            ; if (AL > AH) AL = AH
NEG AH                 ; AH = -2000
MAX AL, @AH            ; if (AL < AH) AL = AH
MOV @VarA, AL          ; 结果保存到 VarA 中

```

MINCUL P, loc32

条件求无符号数的最小值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MINCUL P, loc32	0101 0110 0101 1001 xxxx xxxx LLLL LLLL	1	-	1

操作数: P 乘积寄存器
loc32 寻址模式

描述: 根据负标志位 N 和零标志位 Z 的状态, 有条件地把 P 寄存器中的无符号值与“loc32”寻址模式指向的 32 位无符号数进行比较, 较小值装载到 P 寄存器:

```
if( (N = 0) & (Z = 0) )
    P = [loc32];
if( (N = 0) & (Z = 1) & (P > [loc32]) )
    V = 1, P = [loc32];
if( (N = 1) & (Z = 0) )
    P = 不变;
```

注意: “P < [loc32]” 的操作处理与一条 32 位无符号数的比较类似。

本指令是与 MINL 指令组合成求 64 位数的最小值的典型指令。在使用 MINL 指令比较 64 位数的高 32 位时, 负标志位 N 和零标志位 Z 置位。MINCUL 指令根据 64 位数的高 32 位的比较的结果条件比较 64 位数的低 32 位数。

标志与模式:

- N 若 (N = 1 且 Z = 0), 用[loc32]的值装载 P。
- Z 若 (N = 0 且 Z = 1), 把 P 寄存器的无符号值与[loc32]的无符号值进行比较, 最小值装载到 P 寄存器。
若 (N = 0 且 Z = 0), 不作任何处理。
- V 若 (N = 0, Z = 1 且 P < [loc32]), 则溢出标志位 V 置位; 否则溢出标志位 V 不变。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器的值 RPTC, 并且只执行一次。

例:

```
;使 64 位的 VarA 为饱和值
;if(Var64 > MaxPos64 ) Var64 = MaxPos64
;if(Var64 < MaxNeg64 ) Var64 = MaxNeg64
MOVL ACC,@Var64+2          ;用 Var64 装载 ACC:P
MOVL P,@Var64+0
MINL ACC,@MaxPos64+2        ;if(ACC:P > MaxPos64) ACC:P = MaxPos64
MINCUL P,@MaxPos64+0
MAXL ACC,@MaxNeg64+2        ;if(ACC:P < MaxNeg64) ACC:P = MaxNeg64
MAXCUL P,@MaxNeg64+0
MOVL @Var64+2,ACC           ;结果保存到 Var64 中
MOVL @Var64+0,P
```

MINL ACC,loc32

求 32 位数的最小值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MINL ACC,loc32	0101 0110 0101 0000 0000 0000 LLLL LLLL	1	Y	N+1

操作数: ACC 累加器
loc32 寻址模式

描述: 把 ACC 累加器的值与“loc32”寻址模式指向的 32 位值进行比较, 最小

值装载到 ACC 累加器:

```
if(ACC <= [loc32]), ACC = 不变;
if(ACC >[loc32]), ACC = [loc32];
```

- 标志与模式: Z 若 ACC 等于指定地址中的值 (ACC = [loc32]), 则零标志位 Z 置位; 否则清 0。
- N 若 ACC 小于指定地址中的值 (ACC < [loc32]), 则负标志位 N 置位; 否则清 0。当指定结果的符号时, 假设 MAXL 指令有无限位的精度, 例如, 当精度限制在 32 位时, 减法 0x8000 0000-0x0000 0001, 将引起溢出结果为正数 0x7FFF FFFF, 且负标志位 N 清 0。然而, 当假定 MAXL 指令为无限精度, 则负标志位 N 置位表明 0x8000 0000-0x0000 0001 事实上将产生一个负数。
- C 若 (ACC-[loc32]) 产生借位, 将进位标志位 C 清 0。否则进位标志位 C 置位。
- V 若 ACC 大于指定地址中的值 (ACC > [loc32]), 则溢出标志位 V 置位, 否则 V 不变。
- 重复性: 本指令可以重复。若指令跟在 RPT 指令之后, MINL 指令将执行 N+1 次。Z、N、C 的状态为最终结果。若中间发生溢出, V 标志位立即置位。

例:

```
;使 VarA 为饱和值
;if(VarA > MaxPos) VarA = MaxPos
;if(VarA < MaxNeg) VarA = MaxNeg
MOVL ACC,@VarA          ;ACC = VarA
MINL ACC,@MaxPos        ;if(ACC > MaxPos) ACC = MaxPos
MAXL ACC,@MaxNeg        ;if(ACC < MaxNeg) ACC = MaxNeg
MOVL @VarA,ACC          ;结果保存到 VarA 中
```

MOV *(0:16bit),loc16

移动值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV *(0:16bit),loc16	1111 0100 LLLL LLLL CCCC CCCC CCCC CCCC	×	Y	N+2

操作数: *(0:16bit) 立即存储器寻址, 仅存取数据空间的低 64K 范围 0x00000000~0x0000FFFF

loc16 寻址模式

描述: 将“loc16”地址单元的值移动到“0:16bit”指定的存储器地址单元中:

```
[0x0000:16bit] = [loc16];
```

标志与模式: 无

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。当重复执行时, “0:16bit”数据存储器地址加 1, 仅地址的低 16 位受影响。

例:

```

; 将 Array1 的值复制到 Array2
; int16 Array1[N];
; int16 Array2[N]; // 位于数据空间的低 64K
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i];
MOVL  XAR2, #Array1          ;XAR2 为指向 Array1 的指针
RPT   #(N-1)                 ;重复下一条指令 N 次
||MOV  *(0:Array2), *XAR2++   ; Array2[i] = Array1[i], i++

```

MOV AX,loc16

装载 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV AX,loc16	1001 001A LLLL LLLL	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
loc16 寻址模式

描述: 用“loc16”地址单元中的 16 位数装载累加器的高寄存器 (AH) 或者累加器的低寄存器 (AL)。累加器的另外一半不变。

AX = [loc16];

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 AX 为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

MOV AH, *-XAR0[0]          ;用 XAR0 指向地址中的 16 位数装载 AH, AL 不变
SB  NotZero, NEQ           ;AH 值不为 0 则翻转

```

MOV ACC,#16bit<<#0...15

用移位后的值装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV ACC,#loc16<<#0...15	1111 1111 0010 SHFT CCCC CCCC CCCC CCCC	×	-	1

操作数: ACC 累加器
#16bit 16 位立即数
#0...15 移位位数 (没有指定值时, 默认为“<<#0”)

描述: 用 16 位立即数左移后的值装载 ACC 累加器。若符号扩展模式 (SXM = 1) 使能, 则移位时作符号扩展, 否则移位时作 0 扩展 (SXM = 0)。最低位填 0。

```

if(SXM = 1) // 符号扩展模式使能
    ACC = S:16bit << 移位位数;
else // 符号扩展模式禁止
    ACC = 0:16bit << 移位位数;

```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置位；否则清 0。

SXM 若符号扩展模式使能，装载前 16 位常数已进行符号扩展；否则该值进行 0 扩展。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且仅执行一次。

例：

```
; 计算有符号数 ACC = -2010 << 10 + VarB << 6;
SETC SXM ;使能符号扩展模式
MOV ACC, #-2010 << #10 ; -2010 左移 10 位后装载到 ACC
ADD ACC, @VarB << #6 ; VarB 左移 6 位后装载到 ACC
```

MOV ACC, loc16 << T

用移位后的值装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV ACC, loc16 << T	0101 0110 0111 0110 0000 0000 LLLL LLLL	1	-	1

操作数： ACC 累加器
loc16 寻址模式

T 乘积寄存器的高 16 位 XT(31:16)

描述： 用“loc16”寻址模式指向的 16 位数左移后的值装载 ACC 累加器。由 T 寄存器的最低 4 位 T(3:0) = 0...15 指定移位位数。忽略 T 寄存器的高位。若符号扩展模式 SXM = 1，移位时进行符号扩展，否则移位时进行 0 扩展 (SXM = 0)，最低位填 0：

```
if(SXM = 1) ;//符号扩展模式使能
    ACC = S:[loc16] << T(3:0);
Else ;//符号扩展模式禁止
    ACC = 0:[loc16] << T(3:0);
```

标志与模式： **N** 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则清 0。

Z 若 ACC 的值为 0，则零标志位 Z 置位；否则清 0。

SXM 若符号扩展模式位置 1；由“loc16”寻址指定的地址单元的 16 位数，在装载前先进行符号扩展；否则进行 0 扩展。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且仅执行一次。

例：

```
; 计算有符号数 ACC = (VarA << SA) + (VarB << SB)
SETC SXM ;使能符号扩展模式
MOV T, @SA ;用 SA 中的值装载 T
MOV ACC, @VarA << T ;用移位后的 VarA 装载到 ACC
MOV T, @SB ;用 SB 中的值装载 T
ADD ACC, @VarB << T ;用移位后的 VarB 装载到 ACC
```


MOV ACC,loc16<<#0...16

用移位后的值装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV ACC,loc16<<#0	1000 0101 LLLL LLLL	1	-	1
	1110 0000 LLLL LLLL	0	-	1
MOV ACC,loc16<<#1...15	0101 0110 0000 0011	1	-	1
	0000 SHFT LLLL LLLL			
	1110 SHFT LLLL LLLL	0	-	1
MOV ACC,loc16<<#16	0010 0101 LLLL LLLL	×	-	1

操作数: ACC 累加器

#16bit 16 位立即数

#0...16 移位位数 (没有指定值时, 默认为 "<<#0")

描述: 用 "loc16" 地址单元的值左移后装载到 ACC 累加器。若符号扩展模式使能 (SXM = 1), 则移位时为符号扩展; 否则移位时为 0 扩展 (SXM = 0), 移位时最低位填 0。

```

if (SXM = 1)                // 符号扩展模式使能
    ACC = S : [loc16] << 移位位数;
Else                          // 符号扩展模式禁止
    ACC = 0: [loc16] << 移位位数;

```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则清 0。

SXM 若符号扩展模式位使能, "loc16" 单元的 16 位常数将在装载前进行符号扩展; 否则该值进行 0 扩展。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且仅执行一次。

例:

```

; 计算有符号值 ACC = -2010 << 10 + VarB << 6;
SETC SXM                ;使能符号扩展模式
MOV ACC, #VarA << #10    ;VarA 左移 10 位后装载 ACC 累加器
ADD ACC, @VarB << #6     ;VarB 左移 6 位后加到 ACC 累加器

```

MOV AR6/7,loc16

装载辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV AR6, loc16	0101 1110 LLLL LLLL	×	-	1
MOV AR7 loc16	0101 1111 LLLL LLLL	×	-	1

操作数: AR6/7 AR6 或 AR7 辅助寄存器

loc16 寻址模式

描述: 用 16 位数装载 AR6 或 AR7。XAR6 或 XAR7 的高 16 位不变。

```

AR6/7 = [loc16]);
AR6/7H = 不变;

```


标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

MOV DP, #10bit

装载数据页指针

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV DP, #10bit	1111 10CC CCCC CCCC	×	-	1

操作数： DP 数据页寄存器
#10bit 10 位立即数

描述：用 10 位常数装载数据页寄存器。其他高 6 位不变。

DP(9:0) = 10bit;

DP(15:10) = 不变;

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

例：

MOV DP, #VarA ;用包含 VarA 的数据页装载 DP
;假设 VarA 在低 0x0000 FFC0 中，DP(15:10)不变

MOV IER, loc16

装载中断使能寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV IER, loc16	0010 0011 LLLL LLLL	×	-	1

操作数： IER 中断使能寄存器
loc16 寻址模式

描述：将“loc16”地址单元的内容装载到 IER 寄存器，可以使能和禁止所选的中断。

IER = [loc16];

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

例：

; 将 IER 的内容压入堆栈并且用 VarA 的值装载 IER
MOV *SP+, IER ;保存 IER 到堆栈中
MOV IER, @VarA ;用 VarA 的值装载 IER

MOV loc16, #16bit

存 16 位常数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, #16 bit	0010 1000 LLLL LLLL CCCC CCCC CCCC CCCC	×	Y	N+1

操作数: loc16 寻址模式

#16bit 16 位立即数

描述: 将 16 位立即数装载到 “loc16” 地址单元中:

[loc16]=16bit;

标志与模式: N 若 (loc16=@AX), 将测试装载到 AX 的值是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 将测试装载到 AX 的值是否为 0。若 AX 寄存器为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且仅执行一次。

例:

```

; 用 0xFFFF 的值初始化 Array1 的内容
; int16 Array1[N];
; for(i = 0; i < N; i++)
; Array1[i] = 0xFFFF;
MOVL  XAR2, #Array1          ;XAR2 指针指向 Array1
RPT   #(N-1)                 ;重复性下一条指令 N 次
||MOV  *XAR2+, #0xFFFF      ;Array1[i] = 0xFFFF, i++

```

MOV loc16, *(0:16bit)

移动值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, *(0:16bit)	1111 0101 LLLL LLLL CCCC CCCC CCCC CCCC	×	Y	N+2

操作数: loc16 寻址模式

*(0:16bit)立即存储器寻址, 仅存取数据空间的低 64K 范围。

描述: 将存储器地址 “0:16bit” 单元的常数移动到 “loc16” 地址单元中。

[loc16]=[0x0000: 16bit];

标志与模式: N 若 (loc16=@AX), 将测试装载到 AX 的值是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 将测试装载到 AX 的值是否为 0。若 AX 寄存器为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。在每次重复时, “0:16bit” 数据存储器地址加 1, 仅仅低 16 位地址受影响。

例:

```

; 将 Array1 的值复制到 Array2
; int16 Array1[N]; // 位于数据空间的低 64k
; int16 Array2 N];
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i];
MOVL  XAR2, #Array2          ; XAR2 指针指向 Array2

```

```

RPT    #(N-1)                ; 重复下一条指令 N 次
||MOV  *XAR2+, *(o:Array1)    ; Array2[i] = Array1[i], i++

```

MOV loc16, #0

16 位地址的区域清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, #0	0010 1011 LLLL LLLL	×	Y	N+1

操作数: loc16 寻址模式
#0 立即数 0

描述: 用 0x0000 装载“loc16”指定的存储器单元。

[loc16]=0x0000;

标志与模式: N 若 (loc16=@AX), 将测试装载到 AX 的值是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 将测试装载到 AX 的值是否为 0。若在运行时 AX 寄存器产生 0 值, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。

例:

```

;初始化 Array1 的值为 0
;int16 Array1[N];
; for(i = 0; i < N; i++ )
;Array1[i] = 0;
MOVL  XAR2, #Array1          ;XAR2 指针指向 Array1
RPT    #(N-1)                ;重复下一条指令 N 次
||MOV  *XAR2+, #0             ;Array1[i] = 0, i++

```

MOV loc16, ARn

存 16 位的辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, ARn	0111 1nnn LLLL LLLL	×	-	1

操作数: loc16 寻址模式
ARn AR0~AR7, 辅助寄存器的低 16 位

描述: 用 ARn 的值装载到“loc16”寻址单元中:

[loc16]=ARn;

若 (loc16=@ARn), 仅修改所选辅助寄存器的低 16 位。高 16 位不变。

标志与模式: N 若 (loc16=@AX), 将测试装载到 AX 的值是否为负。AX 寄存器的第 15 位是符号位, 0 为正, 1 为负。若 AX 寄存器的操作产生负值, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 将测试装载到 AX 的值是否为 0。若操作 AX 寄存器产生 0 值, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。

例:

```

MOV  @AL, AR3      ;用 AR3 的 16 位数装载 AL, 若 AL 的第 15 位为 1,
                   ;负标志位 N 置位, 否则清 0。
MOV  @AR4, AR3      ;若 AL 的值为 0。零标志位 Z 置位。用 AR3 的值装载 AR4
                   ;XAR4 的高 16 位不变。
MOV  *SP+ +, AR3     ;将 AR3 的内容压入堆栈, SP 增 1
MOV  *XAR4+ +, AR4   ;保存 AR4 的值到 XAR4 指定的位置, xAR4 增 1。
MOV  *- -XAR5, AR5   ;XAR5 的值先减 1, 再保存 AR5 的值到 XAR5 指定的位置。

```

MOV loc16, AX

存 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, AX	1001 011A LLLL LLLL	X	Y	N+1

操作数: loc16 寻址模式

AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述: 用指定 AX 寄存器 (AH 或 AL) 的 16 位数装载到 “loc16” 地址单元中。

[loc16] = AX;

标志与模式: N 若 (loc16=@AX), 将测试装载到 AX 的值是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 将测试装载到 AX 的值是否为 0。若操作 AX 寄存器产生了 0 值, 则零标志位 Z 置位; 否则清 0。

重复性: 可以重复, 若指令跟在 RPT 指令之后, 它将执行 N+1 次。N 和零标志位 Z 的状态为最终结果。

例:

```

; 初始化所有的 Array1 元素为值 0xFFFF
MOV  AH, #0xFFFF      ;用 0xFFFF 装载 AH
MOVL  XAR2, #Array1    ;用 Array1 地址装载 XAR2
RPT   #9               ;重复下一条指令 10 次
||MOV *XAR2+ +, AH      ;保存 AH 的值到 XAR2 指向的位置并且 XAR2 加 1。

```

MOV loc16, AX, COND

条件保存 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, AX, COND	1001 011A LLLL LLLL	X	Y	N+1

操作数: loc16 寻址模式

AX 累加器高 (AH) 或累加器低 (AL) 寄存器

COND 条件

COND	语 法	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0

续表

COND	语 法	描 述	测试标志位
0100	LT	大于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于, 进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述: 如果测试的条件为真, 将指定的 AX 寄存器 (AH 或 AL) 的内容装载到 “loc16” 地址单元中:

if (COND = 真) [loc16] = AX;

注意: 无条件地执行寻址模式。因此, 不管条件是否满足, 寻址模式都将执行前修改或后修改。

标志与模式: N 若 (COND=真且 loc16=@AX), 测试 AX 的值是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位。

Z 若 (COND=真且 loc16=@AX), 测试 AX 的值是否为 0, 若 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

V 若发生条件测试, 则 V 标志位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 如果 VarB 的值大于 VarA 的值, 交换 VarA 的值与 VarB 的值

MOV AL,@VarA ; AL = VarA, XAR2 指向 VarB

MOV AH,@VarB ; AH = VarB, XAR2 指向 VarA

CMP AH,@AL ; 比较 AH 和 AL

MOV @VarA,AH,HI ; 如果 AH 更高, 则保存 AH 在 VarA 中

MOV @VarB,AL,HI ; 如果 AL 更高, 则保存 AL 在 VarB 中

MOV loc16, IER 保存中断使能寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, IER	0010 0000 LLLL LLLL	×	—	1

操作数: loc16 寻址模式

IER 中断使能寄存器

描述: 将 IER 寄存器的内容保存到 “loc16” 地址单元中。

[loc16] = IER;

标志与模式: N 若 (loc16=@AX) 且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX) 且 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 将 IER 的值压入堆栈并且用 VarA 的值装载 IER
MOV  *SP+, IER      ; 保存 IER 到堆栈中
MOV  IER, @VarA      ; 用 VarA 值装载 IER
```

MOV loc16, P

保存移位 P 寄存器的低位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, P	0011 1111 LLLL LLLL	×	Y	N+1

操作数: loc16 寻址模式

P 乘积寄存器

描述: P 寄存器的值按移位模式 (PM) 指定的位数移位后的低半部分存在 “loc16” 地址单元中。不修改 P 寄存器的值。

[loc16] = P<<PM;

标志与模式: N 若 (loc16=@AX) 且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX) 且 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0。若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。并且标志位 Z、N 的状态为最终结果。

例:

```
; 计算 Y32 = M16×X16 >> 6
MOV  T, @M16          ; T = M
MPY  P, T, @X16        ; P = T × X
SPM  -6                ; 设置乘积右移 6 位
MOV  @Y32+0, P         ; Y32 = P 右移 6 位
MOVH @Y32+1, P
```

MOV loc16, T

保存 T 寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, P	0011 1111 LLLL LLLL	×	Y	N+1

操作数: loc16 寻址模式

T 被乘数寄存器的高 16 位

描述: 保存 16 位 T 寄存器的值到 “loc16” 寻址指定的地址。

[loc16] = T;

标志与模式: N 若 (loc16=@AX) 且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX) 且 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 16 位的乘积
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
; X2 = X1
; X1 = X0
SPM    -2                ; 设置乘积右移 2 位
MOV     T,@X2             ; T = X2
MPY     P,T,@C2           ; P = T×C2
MOVP    T,@X1             ; T = X1, ACC = X2×C2 >> 2
MPY     P,T,@C1           ; P = T×C1
MOV     @X2,T             ; X2 = X1
MOVA    T,@X0             ; T = X0, ACC = X1×C1 >> 2 + X2×C2 >> 2
MPY     P,T,@C0           ; P = T×C0
MOV     @X1,T             ; X1 = X0
ADDL    ACC,P << PM       ; ACC = X0×C0 >> 2 + X1×C1 >> 2 + X2×C2 >> 2
MOVL    @Y,ACC            ; 结果保存到 Y
```

MOV OVC,loc16

装载溢出计数器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV OVC,loc16	0101 0110 0000 0000 0000 0000 LLLL LLLL	1	-	1

操作数: OVC 6 位溢出计数器

loc16 寻址模式

描述: 用 “loc16” 地址单元的高 6 位数装载溢出计数器 (OVC)。

OVC = [loc16 (15: 10)];

标志与模式: OVC 修改 6 位溢出计数器。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC。并且只执行一次。

例:

```
; 保存和恢复 ACC 的值和 OVC
MOV     *SP+ +,OVC        ;保存 OVC 到堆栈中
MOV     *SP+ +,AL         ;保存 AL 到堆栈中
MOV     *SP+ +,AH         ;保存 AH 到堆栈中
.
```

```

MOV  AH, *- -SP      ;从堆栈中恢复 AH
MOV  AL, *- -SP      ;从堆栈中恢复 AL
MOV  OVC, *- -SP     ;从堆栈中恢复 OVC

```

MOV PH, loc16

装载 P 寄存器的高字

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV PH, loc16	0010 1111 LLLL LLLL	X	-	1

操作数: PH 乘积寄存器的高 16 位
loc16 寻址模式

描述: 用“loc16”地址单元的 16 位数装载 P 寄存器的高 16 位 (PH); 低 16 位 (PL) 不变。

```

PH = [loc16];
PL = 不变;

```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;交换 AH 和 AL 的值
MOV  PH, @AL      ;用 AL 装载 PH
MOV  PL, @AH      ;用 AH 装载 PL
MOV  ACC, @P      ;用 P(AH 与 AL 交换)装载 ACC

```

MOVL PL, loc16

装载 P 寄存器的低字

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL PL, loc16	0010 0111 LLLL LLLL	X	-	1

操作数: PL 乘积寄存器的低 16 位
loc16 寻址模式

描述: 用“loc16”地址单元的 16 位数装载 P 寄存器的低 16 位 (PL); 高 16 位 (PH) 不变。

```

PL = [loc16];
PH = 不变;

```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

;交换 AH 和 AL 的值
MOV  PH, @AL      ;用 AL 装载 PH
MOV  PL, @AH      ;用 AH 装载 PL

```

MOV ACC,@P ;用 P(AH 与 AL 交换)装载 ACC

MOV PM, AX

装载乘积移位模式

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV PM,AX	0101 0110 0011 100A	1	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述: 用 AX 寄存器的最低 3 位装载乘积寄存器 (PM)。

PM = AX(2:0);

标志与模式: PM 将 AX 寄存器的最低 3 位装载到乘积移位模式位 (PM)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 Y32 = (M16×X16 >> Shift) + B32, 移位 0~ 6 位
CLRC AMODE                ; 确保 AMODE = 0
MOV AL,@Shift              ; 用移位位数装载 AL
ADDB AL,#1                 ; 转换“移位位数”编码到 PM
MOV PM,AX                  ; 用“移位位数”编码装载 PM 位
MOV T,@X16                 ; T = X16
MPY P,XT,@M16              ; P = X16×M16
MOVL ACC,@B32              ; ACC = B32
ADDL ACC,P << PM           ; ACC = ACC + (P >>移位位数)
MOVL @Y32,ACC              ; 结果保存到 Y32 中
```

MOV T, loc16

装载 XT 寄存器的高字

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV T,loc16	0010 1101 LLLL LLLL	×	-	1

操作数: T 被乘数寄存器的高 16 位

loc16 寻址模式

描述: 用“loc16”地址单元的 16 位数装载到 T 寄存器。

T = [loc16];

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 用 16 位的乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
; X2 = X1
; X1 = X0
SPM -2                    ; 设乘积右移 2 位
MOV T,@X2                 ; T = X2
MPY P,T,@C2               ; P = T×C2
```

```

MOVP  T,@X1      ; T = X1, ACC = X2×C2 >> 2
MPY   P,T,@C1    ; P = T×C1
MOV   @X2,T      ; X2 = X1
MOVA  T,@X0      ; T = X0, ACC = X1×C1 >> 2 + X2×C2 >> 2
MPY   P,T,@C0    ; P = T×C0
MOV   @X1,T      ; X1 = X0
ADDL  ACC,P << PM ; ACC = X0×C0 >> 2 + X1×C1 >> 2 + X2×C2 >> 2
MOVL  @Y,ACC      ; 结果保存到 Y 中

```

MOV TL,#0

XT 寄存器的低字清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV TL,#0	0101 0110 0101 0110	1	-	1

操作数: T 被乘数寄存器的高 16 位
#0 立即数 0

描述: 被乘数寄存器的低字 TL 为 0, 高字不变。

TL = 0x0000;

T = 不变;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 计算并保留结果的低 32 位 Y32 = M32×X16 >> 32
MOV  TL,#0      ; TL = 0
MOV  T,@X16     ; T = X16
IMPYL P,XT,@M32 ; P = XT × M32 (结果的高 32 位)
MOVL @Y32,P     ; 结果保存到 Y32

```

MOV XARn,PC

保存当前程序计数器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV XARn,PC	0011 1110 0101 1nnn	1	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器

loc32 寻址模式

PC 22 位程序计数器

描述: 将 PC 值保存到 XARn:

XARn = 0:PC;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

TableA:

.long CONST1

; TableA 的位置是相对于当前程序而言的


```

        .long  CONST2
        .long  CONST3
        .
FuncA:
        MOV   XAR5, PC
        SUBB  XAR5, #($-TableA) ;XAR5=当前 PC
        MOVL  ACC, *+XAR5[2]      ;XAR5=TableA 起始位置
        MOVL  @VarA, ACC          ;用 CONST2 装载 ACC 并保存到 VarA

```

MOVA T, loc16

装载 T 寄存器并加上先前的乘积

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVA T, loc16	0001 0000 LLLL LLLL	×	Y	N+1

操作数: T 被乘数寄存器的高 16 位
loc16 寻址模式

描述: 用“loc16”地址单元的 16 位数装载 T 寄存器。而 P 寄存器的内容由被乘积移位模式位 (PM) 指定的移位位数移位后加到 ACC 累加器:

```

T = [loc16];
ACC = ACC + P << PM;

```

标志与模式: N 若 ACC 累加器的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。
C 若产生进位, 则进位标志位 C 置位; 否则清 0。
V 若产生溢出, 则溢出标志位 V 置位; 否则 V 不受影响。
OVC 若溢出模式禁止且操作产生正溢出, 计数器值增加; 操作产生负溢出, 计数器值减少。
OVM 若溢出模式位置位, 当运行出现溢出时, ACC 值将为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号位扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。Z、N、C 和 OVC 的状态为最终的结果。当中间发生溢出时, V 标志位立即置 1。

例:

```

; 用 16 位的乘积进行计算
; Y = (X0xC0) >> 2) + (X1xC1 >> 2) + (X2xC2 >> 2)
; X2 = X1
; X1 = X0
SPM   -2                ; 设置乘积右移 2 位
MOV   T, @X2             ; T = X2
MPY   P, T, @C2          ; P = TxC2
MOVP  T, @X1             ; T = X1, ACC = X2xC2 >> 2
MPY   P, T, @C1          ; P = TxC1
MOV   @X2, T             ; X2 = X1

```

```

MOVA  T,@X0          ; T = X0, ACC = X1×C1 >> 2 + X2×C2 >> 2
MPY   P,T,@C0        ; P = T×C0
MOV   @X1,T          ; X1 = X0
ADDL  ACC,P << PM     ; ACC = X0×C0 >> 2 + X1×C1 >> 2 + X2×C2 >> 2
MOVL  @Y,ACC         ; 结果保存到 Y 中

```

MOVAD T, loc16

装载 T 寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVAD T, loc16	1010 0111 LLLL LLLL	1	N	1

操作数: T 被乘数寄存器的高 16 位
 loc16 寻址模式

注意: 对于该操作, 不能使用寄存器寻址模式。寄存器寻址模式为:

@ARn, @AH, @AL, @PH, @PL, @SP, @T。将产生一个错误指令中断。

描述: 用“loc16”地址单元的 16 位数装载 T 寄存器, 然后将 T 的内容装载到“loc16”地址的下一个地址。而 P 寄存器的内容由被乘积移位模式位 (PM) 指定的移位位数移位后加到 ACC 累加器:

```

T = [loc16];
[loc16+1]=T;
ACC = ACC+P<<PM;

```

标志与模式: N 若 ACC 累加器的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
 Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。
 C 若产生进位, 则进位标志位 C 置位; 否则清 0。
 V 若产生溢出, 则溢出标志位 V 置位; 否则 V 不受影响。
 OVC 若溢出模式禁止, 而操作产生正溢出, 计数器值增加; 操作产生负溢出, 计数器值减少。
 OVM 若溢出模式位置位, 当运行发生溢出时, ACC 值将为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 最高位进行符号扩展。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 用 16 位的乘积进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
; X2 = X1
; X1 = X0
SPM  -2          ; 设置乘积右移 2 位
MOVP  T,@X2      ; T = X2
MPYS  P,T,@C2    ; P = T×C2, ACC = 0

```

```

MOVAD T,@X1      ; T = X1, ACC = X2×C2>>2, X2 = X1
MPY P,T,@C1      ; P = T×C1
MOVAD T,@X0      ; T = X0, ACC = X1×C1>>2 + X2×C2>>2, X1 = X0
MPY P,T,@C0      ; P = T×C0
ADDL ACC,P << PM ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL @Y,ACC      ; 结果保存到 Y 中

```

MOVB ACC, #8bit

用 8 位数装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB ACC, #8bit	0000 0010 CCCC CCCC	1	-	1

操作数: ACC 累加器

#8bit 8 位立即无符号常数

描述: 用指定的 8 位, 0 扩展立即数并装载到 ACC 累加器中。

ACC = 0:8bit;

标志与模式: N 若 ACC 累加器的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 32 位区域 VarA 的增量值

MOVB ACC, #1 ; 用值 0x0000 0001 装载 ACC

ADDL ACC, @VarA ; 将 VarA 的值加到 ACC

MOVL @VarA, ACC ; 将结果保存回 VarA 中

MOVB AR6/7, #8bit

8 位常数装载辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV AR6, #8bit	1101 0110 CCCC CCCC	×	-	1
MOV AR7, #8bit	1101 0111 CCCC CCCC	×	-	1

操作数: XARn XAR6 或 XAR7, 32 位辅助寄存器

#8bit 8 位立即数

描述: 用 8 位无符号常数装载 AR6 或 AR7。XAR6 或 XAR7 的高 16 位不变。

AR6/7 = 0:8bit;

AR6/7H = 不变;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

MOVB AX, #8bit

用 8 位常数装载 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB AX, #8bit	1001 101A CCCC CCCC	×	-	1

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
#8bit 8 位立即数

描述: 用 8 位无符号常数 0 扩展后的值装载累加器高 (AH) 或累加器低 (AL) 寄存器。累加器的其他部分不变。

AX = 0:8bit;

标志与模式: N 该标志位总是置为 0
Z 若 AX 的值为 0, 则零标志位 Z 置位, 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```
MOVB AL, #0xF0      ; 用值 0x00F0 装载 AL
CMP  AL, *+XAR0[0]   ; 用 XAR0 指向的内容与 AL 比较
SB   Dest, EQ        ; 如果相等则跳转
```

MOVB AX.LSB, loc16

装载字节值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB AX.LSB, loc16	1100 011A LLLL LLLL	×	-	1

操作数: AX.LSB 累加器高 (AH.LSB) 或累加器低 (AL.LSB) 寄存器的最低有效字节

loc16 寻址模式

描述: 来自 “loc16” 地址单元的位值装载到指定 AX 寄存器的最低有效字节。AX 寄存器的最高有效字节清 0。“loc16” 操作数的形式指定了哪个 8 位用于装载 AX.LSB。

```
if (loc16 = *+XARn[offset])
{
    if (偏移量为偶数)
        AX.LSB = [loc16.LSB];
    if (偏移量为奇数)
        AX.LSB = [loc16.MSB];
}
else
    AX.LSB = [loc16.LSB];
AX.MSB = 0x00;
```

注意: 3 位偏移量立即数或为 AR0、AR1 的索引模式寻址。

标志与模式: Z 测试 AX 是否为 0, 若 AX=0, 则零标志位 Z 置位; 否则零标志位 Z 清 0。

N 测试 AX 是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则负标志位 N 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器

RPTC, 并且只执行一次。

例:

```
; 在 32 位的 "Var32" 位置交换字节次序
; 运行前: Var32 = B3 | B2 | B1 | B0
; 运行后: Var32 = B0 | B1 | B2 | B3
MOVL  XAR2, #Var32          ; 装载 XAR2 指向 "Var32" 地址
MOVB  AL.LSB, *+XAR2[3]     ; ACC(B0) = Var32(B3), ACC(B1) = 0
MOVB  AH.LSB, *+XAR2[1]     ; ACC(B2) = Var32(B1), ACC(B3) = 0
MOVB  AL.MSB, *+XAR2[2]     ; ACC(B1) = Var32(B2), ACC(B1) = unch
MOVB  AH.MSB, *+XAR2[0]     ; ACC(B3) = Var32(B0), ACC(B1) = unch
MOVL  @Var32, ACC           ; 保存交换结果在 "Var32" 中
```

MOVB AX.MSB, loc16

装载字节值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB AX.MSB, loc16	0011 100A LLLL LLLL	×	-	1

操作数: AX.MSB 累加器高 (AH.MSB) 或累加器低 (AL.MSB) 寄存器的最高有效字节

loc16 寻址模式

描述: 将 "loc16" 地址单元的 8 位值装载到指定 AX 寄存器的最高有效字节。AX 寄存器的最低有效字节不变。"loc16" 操作数的形式指定了哪 8 位用于装载 AX.MSB。

```
if(loc16 = *+XARn[offset])
{
    if(偏移量为偶数)
        AX.MSB = [loc16.LSB];
    if(偏移量为奇数)
        AX.MSB = [loc16.MSB];
}
else
    AX.MSB = [loc16.LSB];
AX.LSB = 0x00;
```

注意: 偏移量=3 位立即数或为 AR0 或 AR1 的索引模式寻址。

标志与模式: N 测试 AX 是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则负标志位 N 清 0。

Z 测试 AX 是否为 0, 若 AX=0 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 在 32 位的 "Var32" 位置交换字节次序
; 运行前: Var32 = B3 | B2 | B1 | B0
; 运行后: Var32 = B0 | B1 | B2 | B3
MOVL  XAR2, #Var32          ; 装载 XAR2 指向 "Var32" 地址
```



```

MOVB  AL.LSB,  *+XAR2[3]      ; ACC(B0) = Var32(B3), ACC(B1) = 0
MOVB  AH.LSB,  *+XAR2[1]      ; ACC(B2) = Var32(B1), ACC(B3) = 0
MOVB  AL.MSB,  *+XAR2[2]      ; ACC(B1) = Var32(B2), ACC(B1) = unch
MOVB  AH.MSB,  *+XAR2[0]      ; ACC(B3) = Var32(B0), ACC(B1) = unch
MOVL  @Var32, ACC              ; 保存交换后的结果到 "Var32" 中

```

MOV loc16, #8bit, COND

条件保存 8 位常数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc16, #8bit, COND	0101 0110 1011 COND CCCC CCCC LLLL LLLL	1	-	1

操作数: loc16 寻址模式
 #8bit 8 位立即数
 COND 条件代码

COND	语 法	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	大于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于, 进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述: 若指定条件测试为真, 0 扩展后的 8 位常数保存到 “loc16” 地址单元中:

```
if (COND = 真) [loc16] = 0:8bit;
```

注意: 无条件地执行寻址模式。因此, 不管条件是否满足, 寻址模式都将执行前修改或后修改。

标志与模式: N 若 (COND=真且 loc16=@AX), 移动后将测试 AX 是否为负。
 并且若 AX 的第 15 位为 1, 则负标志位 N 置位。
 Z 若 (COND=真且 loc16=@AX), 移动后将测试 AX 是否为 0。
 若对 AX 寄存器的操作产生 0 值, 则零标志位 Z 置位; 否则清 0。
 V 若条件测试, 则标志位 V 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 if( VarA > 20 ) VarA = 0;
CMP  @VarA, #20          ; 根据 (VarA - 20) 的值设置标志位
MOVB  @VarA, #0, GT       ; 如果大于时, 置 VarA 为 0
MOV   @VarA, AH, HI       ; 如果 AH 更高时, 保存 AH 在 VarA 中
MOV   @VarB, AL, HI       ; 如果 AL 更高时, 保存 AL 在 VarB 中
```

MOVB loc16, AX.LSB,

保存 AX 寄存器的最低有效字节

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB loc16, AX.LSB	0011 110A LLLL LLLL	×	-	1

操作数: loc16 寻址模式
AX.LSB 累加器高 (AH.LSB) 或累加器低 (AL.LSB) 寄存器的最低有效字节

描述: 用指定的 AX 寄存器 (AH.LSB 或 AL.LSB) 的最低有效字节装载“loc16”寻址模式指定位置的 8 位数。“loc16”操作数的形式决定哪 8 位用于装载 AX, 哪 8 位保持不变。

```
if(loc16 = *+XARn[offset])
{
    if(offset 为偶数)
        [loc16.LSB] = AX.LSB;
    [loc16.MSB] = 不变;
    if(offset 为奇数)
        [loc16.LSB] = 不变;
        [loc16.MSB] = AX.LSB;
}
else
    [loc16.LSB] = AX.LSB;
    [loc16.MSB] = 不变;
```

注意: 偏移量 (offset) = 3 位立即数或为 AR0、AR1 的寻址模式, 这是个读取-修改-写入操作的过程。

标志与模式: N 若 (loc16=@AX), 则装载后将检测 AX 是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 则装载后将检测 AX 是否为 0, 若 AX=0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 以相反的字节次序保存 ACC 的 32 位数到 "Var32" 位置的 32 位数中
; 指令执行前: ACC = B3 | B2 | B1 | B0
; 指令执行后: Var32 = B0 | B1 | B2 | B3
```

```

MOVL  XAR2, #Var32      ; 装载"Var32"地址到 XAR2
MOVB  *+XAR2[0], AH.MSB ; Var32(B0) = ACC(B3)
MOVB  *+XAR2[1], AH.LSB ; Var32(B1) = ACC(B2)
MOVB  *+XAR2[2], AL.MSB ; Var32(B2) = ACC(B1)
MOVB  *+XAR2[3], AL.LSB ; Var32(B3) = ACC(B0)

```

MOVB loc16, AX.MSB,

保存 AX 寄存器的最高有效字节

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB loc16, AX.MSB	1100 100A LLLL LLLL	×	-	1

操作数: loc16 寻址模式

AX.MSB 累加器高 (AH.MSB) 或累加器低 (AL.MSB) 寄存器的最高有效字节

描述: 用指定的 AX 寄存器 (AH.MSB 或 AL.MSB) 的最高有效字节装载“loc16”寻址模式指定位置的 8 位数。“loc16”操作数的形式决定哪 8 位数装载 AX.MSB, 哪 8 位数保持不变。

```

if (loc16 = *+XARn[offset])
{
    if (offset 为偶数)
        [loc16.LSB] = AX.MSB;
    [loc16.MSB] = 不变;
    if (offset 为奇数)
        [loc16.LSB] = 不变;
        [loc16.MSB] = AX.MSB;
}
else
    [loc16.LSB] = AX.MSB;
    [loc16.MSB] = 不变;

```

注意: 偏移量 (offset) = 3 位立即数或为 AR0、AR1 的寻址模式, 这是个读取-修改-写入操作过程。

标志与模式: N 若 (loc16=@AX), 则装载后将检测 AX 是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 则装载后将检测 AX 是否为 0, 若 AX=0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 按相反的字节顺序保存 ACC 中 32 位数到"Var32"寻址的 32 位中
; 指令执行前: ACC = B3 | B2 | B1 | B0
; 指令执行后: Var32 = B0 | B1 | B2 | B3
MOVL  XAR2, #Var32      ; 装载"Var32" 地址到 XAR2
MOVB  *+XAR2[0], AH.MSB ; Var32(B0) = ACC(B3)
MOVB  *+XAR2[1], AH.LSB ; Var32(B1) = ACC(B2)
MOVB  *+XAR2[2], AL.MSB ; Var32(B2) = ACC(B1)
MOVB  *+XAR2[3], AL.LSB ; Var32(B3) = ACC(B0)

```

MOVB XARn, #8bit

用 8 位数装载辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVB XAR0...5, #8bit	1101 0nnn CCCC CCCC	×	—	1
MOVB XAR6, #8bit	1011 1110 CCCC CCCC	1	—	1
MOVB XAR7, #8bit	1011 0110 CCCC CCCC	1	—	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器
#8bit 8 位立即数

描述: 用 8 位无符号立即数装载到 XARn:

XARn = 0:8bit;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

MOVB XAR0, #F2h ; 用 0x0000 00F2 装载 XAR0

MOVDL XT, loc32

保存 XT 并装载新的 XT

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVDL XT, loc32	1010 0110 LLLL LLLL	1	Y	N+1

操作数: XT 被乘数寄存器
loc32 寻址模式

注意: 在该操作中, 不能使用寄存器寻址模式。寄存器寻址模式为:

@XARn, @ACC, @P, @XT。将产生一个错误指令中断。

描述: 将“loc32”地址单元的 32 位数装载到 XT 寄存器, 然后将 XT 的数装载到“loc32”的下一个地址。

XT = [loc32];
[loc32+2] = XT;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 用 32 位的乘法进行计算, 保留高位结果
; Y = (X0xC0) >> 2) + (X1xC1 >> 2) + (X2xC2 >> 2)
; X2 = X1
; X1 = X0
SPM -2                ; 设置乘积右移两位
ZAPA                  ; ACC, P, OVC 清 0
MOVL XT,@X2           ; XT = X2

```

```

QMPYL P,XT,@C2      ; P = XT×C2
MOVDL XT,@X1         ; XT = X1, ACC = X2×C2>>2, X2 = X1
QMPYAL P,XT,@C1      ; P = XT×C1
MOVDL XT,@X0         ; XT = X0, ACC = X1×C1>>2 + X2×C2>>2,
                     ; X1 = X0
QMPYAL P,XT,@C0      ; P = XT×C0
ADDL ACC,P << PM     ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL @Y,ACC          ; 结果保存到 Y 中

```

MOVH loc16,ACC<<1..8

保存移位后累加器的高位字

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVH loc16,ACC<<1	1011 0011 LLLL LLLL	1	Y	N+1
MOVH loc16,ACC<<2..8	0101 0110 0010 1111 0000 0SHF LLLL LLLL	1	Y	N+1
	1011 0SHF LLLL LLLL	0	-	1

操作数: loc16 寻址模式
ACC 累加器
#1..8 移位位数

描述: 将 ACC 累加器的高位字左移指定位数后装载到“loc16”地址单元中, ACC 累加器的值不变。

[loc16] = ACC >> (16 - 移位位数);

标志与模式: N 若 (loc16=@AX), 则装载后将检测 AX 是否为负, 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 则装载后将检测 AX 是否为 0。若 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它复位重复计数器 RPTC, 并且只执行一次。

例:

;两个格式为 Q15 的数相乘 (VarA×VarB) 并以格式 Q15 保存到 VarC 中

```

MOV T,@VarA          ; T = VarA (Q15)
MPY ACC,T,@VarB       ; ACC = VarA × VarB (Q30)
MOVH @VarC,ACC << 1   ; VarC = ACC >> (16-1) (Q15)
                     ; VarC 为一个 Q31 格式的数
MOV T,@VarA          ; T = VarA (T = Q14)
MPY ACC,T,@VarB       ; ACC = VarA × VarB (ACC = Q28)
MOV @VarC+0,ACC << 3   ; VarC 低 = ACC << 3
MOVH @VarC+1,ACC << 3   ; VarC 高 = ACC >> (16-1) (VarC = Q31)

```

MOVH loc16,P

保存 P 寄存器的高字

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVH loc16,P	0101 0111 LLLL LLLL	×	Y	N+1

- 操作数: loc16 寻址模式
P 乘积寄存器
- 描述: P 寄存器的值按乘积移位模式 (PM) 指定的位数移位移位后的高字保存在 “loc16” 地址单元中。P 寄存器不变:
- $$[\text{loc16}] = (\text{P} \ll \text{PM}) \gg 16;$$
- 标志与模式: N 若 (loc16=@AX), 则装载后测试 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。
Z 若 (loc16=@AX), 则装载后测试 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。
PM PM 位的值设置了乘积寄存器输出的移位操作模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号位扩展。
- 重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。Z、N 标志位的状态为最终结果。

例:

```
; 计算 Y32 = M16×X16 >> 6
MOV    T,@M16          ; T = M
MPY    P,T,@X16         ; P = T × X
SPM    -6               ; 设置乘积右移 6 位
MOV    @Y32+0,P         ; Y32 = P >> 6
MOVH   @Y32+1,P
```

MOVL ACC, loc32

用 32 位数装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL ACC, loc32	0000 0110 LLLL LLLL	×	-	1

- 操作数: ACC 累加器
loc32 寻址模式
- 描述: 用 “loc32” 地址单元的数装载 ACC 累加器。
- $$\text{ACC} = [\text{loc32}];$$
- 标志与模式: N 若 ACC 累加器的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
Z 若 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 32 位数 VarC = VarA + VarB;
MOVL   ACC,@VarA      ; 装载 ACC 为 VarA 的值
ADDL   ACC,@VarB      ; 加 VarB 的值到 ACC
MOVL   @VarC,ACC       ; 结果保存到 VarC 中
```

MOVL ACC,P<<PM

用移位后的 P 寄存器的值装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL ACC,P<<PM	0001 0110 1010 1100	×	-	1

注意：本指令为“loc16=@T”寻址模式的“MOVP T, loc16”操作的别名。

操作数： ACC 累加器
 P 乘积寄存器
 <<PM 乘积移位模式
 描述： P 寄存器的值按乘积移位模式(PM)指定的移位位数移位后装载到 ACC。

$$ACC = P \ll PM;$$

标志与模式： N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则清 0。
 Z 若 ACC 为 0，则零标志位 Z 置位；否则清 0。
 PM PM 位的值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正（逻辑左移操作），最低位填 0；若乘积移位位数为负（逻辑右移操作），则高位进行符号位扩展。
 重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

; 计算 Y = Y + (M×X >> 4)
; Y 是 32 位数，M 和 X 是 16 位数
SPM    -4                ; 设置乘积移位位数为右移 4 位
MOV     T,@M              ; T = M
MPY     P,T,@X            ; P = M × X
MOVL    ACC,P << PM      ; ACC = M×X 右移 4 位
ADDL    @Y,ACC            ; Y = Y + ACC

```

MOVL loc32,ACC

保存 32 位累加器的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL loc32,ACC	0001 1110 LLLL LLLL	×	-	1

操作数： ACC 累加器
 loc32 寻址模式
 描述： 将 ACC 累加器的值保存到“loc32”地址单元中。

$$[loc32] = ACC;$$

标志与模式： N 若 (loc32=@ACC)，ACC 第 31 位为 1，则负标志位 N 置位；否则清 0。
 Z 若 (loc32=@ACC)，AX 的值为 0，则零标志位 Z 置位；否则清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;计算 32 位值 VarC = VarA + VarB;
MOVL  ACC,@VarA      ; 装载 ACC 为 VarA 的值
ADDL  ACC,@VarB      ; 加 VarB 的值到 ACC
MOVL  @VarC,ACC       ; 结果保存到 VarC 中
```

MOV loc32,ACC,COND

条件保存累加器的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOV loc32, ACC,COND	0101 0110 0100 1000 0000 COND LLLL LLLL	×	—	1

操作数： loc32 寻址模式
ACC 累加器
COND 条件代码

COND	格 式	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	小于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于，进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述： 如果条件测试为真，则将 ACC 累加器的值装载到“loc32”地址单元中：

```
if (COND=真) [loc32] = ACC;
```

注意：无条件地执行寻址模式。因此，不管条件是否满足，寻址模式都将执行前修改或后修改。

标志与模式： N 若 (COND=真且 loc32=@ACC)，ACC 第 31 位为 1，则负标志位 N 置位；否则清 0。
Z 若 (COND=真且 loc32=@ACC)，ACC 为 0，则零标志位 Z 置位；否则清 0。

V 若发生条件测试，V 标志位清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

; 如果 VarB 的值大于 VarA 的值，交换 32 位的 VarA 值与 VarB 值
MOVL ACC,@VarB      ; ACC = VarB
MOVL P,@VarA         ; P = VarA
CMLP ACC,@P          ; 由 (VarB - VarA) 的值设置标志位
MOVL @VarA,ACC,HI    ; 如果条件为高，VarA = ACC
MOVL @P,ACC,HI       ; 如果条件为高，P = ACC
MOVL @VarA,P         ; VarA = P

```

MOVL loc32, P

保存 P 寄存器的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL loc32, P	1010 1001 LLLL LLLL	1	-	1

操作数： loc32 寻址模式

P 乘积寄存器

描述： P 寄存器的值保存到“loc32”地址单元中：

[loc32]=P;

标志与模式： N 若 (loc32=@ACC)，ACC 第 31 位为 1，则负标志位 N 置位；否则清 0。

Z 若 (loc32=@ACC)，ACC 为 0，则零标志位 Z 置位；否则清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

; 将 64 位的 VarA, VarB 和 VarC 相加，并且将结果保存到 VarD 中
MOVL P,@VarA+0      ; 用 VarA 的低 32 位装载 P
MOVL ACC,@VarA+2     ; 用 VarA 的高 32 位装载 ACC
ADDUL P,@VarB+0      ; 将 VarB 的低 32 位无符号加到 P
ADDCL ACC,@VarB+2    ; 将 VarB 带进位的高 32 位加到 ACC
ADDUL P,@VarC+0      ; 将 VarC 的低 32 位无符号加到 P
ADDCL ACC,@VarC+2    ; 将 VarC 带进位的高 32 位加到 ACC
MOVL @VarD+0,P       ; 保存结果的低 32 位到 VarD 中
MOVL @VarD+2,ACC     ; 保存结果的高 32 位到 VarD 中

```

MOVL loc32, XARn

保存 32 位辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL loc32, XAR0	0011 1010 LLLL LLLL	1	-	1
MOVL loc32, XAR1	1011 0010 LLLL LLLL	1	-	1
MOVL loc32, XAR2	1010 1010 LLLL LLLL	1	-	1
MOVL loc32, XAR3	1010 0010 LLLL LLLL	1	-	1
MOVL loc32, XAR4	1010 1000 LLLL LLLL	1	-	1

续表

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL loc32, XAR5	1010 0000 LLLL LLLL	1	-	1
MOVL loc32, XAR6	1100 0010 LLLL LLLL	×	-	1
MOVL loc32, XAR7	1100 0011 LLLL LLLL	×	-	1

操作数: loc32 寻址模式

XARn XAR0~XAR7, 32 位辅助寄存器

描述: 将 XARn 寄存器的值保存到“loc32”地址单元中

[loc32]=XARn;

标志与模式: N 若 (loc32=@ACC), 则装载后将检测 ACC 是否为负。ACC 累加器的第 31 位是符号位, 0 为正, 1 为负。若操作时 ACC 累加器为负, 则负标志位 N 置位; 否则清 0。

Z 若 (loc32=@ACC), 则装载后将检测 ACC 是否为 0。若操作结果使 ACC 累加器为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

MOVL @ACC, XAR0      ; 将 32 位的 XAR0 值移动到 ACC。若 ACC 的
                      ; 第 31 位为 1, N 置位, 若 ACC 的值为 0, Z 置位。
MOVL *XAR1, XAR7      ; 将 32 位的 XAR7 值移动到 XAR1 指向的地址单元中。
MOVL *XAR6+, XAR6      ; 将 32 位的 XAR6 值移动到 XAR6 指向的地址单
                      ; 元中。并 XAR6 加 1。
MOVL *-XAR5, XAR5      ; 先将 XAR5 的值减 1, 再将 32 位 XAR5 的值移动
                      ; 到 XAR5 指向的地址单元中。

```

MOVL loc32, XT

保存 XT 寄存器的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL loc32, XT	0011 1010 LLLL LLLL	1	-	1

操作数: loc32 寻址模式

XT 被乘数寄存器

描述: 保存 XT 寄存器的值到“loc32”寻址地址单元中:

[loc32]=XT;

标志与模式: N 若 (loc32=@ACC), ACC 第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc32=@ACC), ACC 为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 用 32 位乘法进行计算, 保留高位
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2) (注: ">>2"为右移 2 位)

```



```

; X2 = X1
; X1 = X0
SPM    -2                ; 设置乘积右移 2 位
ZAPA                    ; ACC, P, OVC 清 0
MOVL    XT,@X2           ; XT = X2
QMPYL    P,XT,@C2        ; P = XT×C2
MOVL    XT,@X1           ; XT = X1, ACC = X2×C2>> 2
QMPYAL    P,XT,@C1       ; P = XT×C1
MOVL    @X2,XT           ; X2 = X1
MOVL    XT,@X0           ; XT = X0, ACC = X1×C1 >> 2 + X2×C2 >> 2
QMPYAL    P,XT,@C0       ; P = XT×C0
MOVL    @X1,XT           ; X1 = X0
ADDL    ACC,P << PM      ; ACC = X0×C0 >> 2 + X1×C1 >> 2 + X2×C2 >> 2
MOVL    @Y,ACC           ; 结果保存到 Y

```

MOVL P, ACC

将累加器的值送到 P 乘积寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL P,ACC	1111 1111 0101 1010	×	-	1

操作数: P 乘积寄存器

ACC 累加器

描述: 用 ACC 累加器的值装载 P 寄存器:

P=ACC;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 计算 32 位的值 VarC = abs(VarA) + abs(VarB)
MOVL    ACC,@VarA        ; 用 VarA 值装载 ACC
ABS     ACC               ; 取 VarA 的绝对值
MOVL    P,ACC             ; 临时保存 ACC 在 P 寄存器中
MOVL    ACC,@VarB        ; 用 VarB 的值装载 ACC
ABS     ACC               ; 取 VarB 的绝对值
ADDL    ACC,@P            ; 将 P 值加到 ACC
MOVL    @VarC,ACC        ; 结果保存到 VarC 中

```

MOVL P, loc32

装载 P 寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL P,loc32	1010 0011 LLLL LLLL	1	-	1

操作数: P 乘积寄存器

loc32 寻址模式

描述: 用“loc32”寻址地址单元的内容装载 P 寄存器:

P=[loc32];

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 将 64 位 VarA, VarB 和 VarC 相加, 并将结果保存到 VarD 中:
MOVL P,@VarA+0      ; 用 VarA 的低 32 位装载 P
MOVL ACC,@VarA+2     ; 用 VarA 的高 32 位装载 ACC
ADDUL P,@VarB+0      ; 将无符号的低 32 位的值 VarB 加到 P
ADDCL ACC,@VarB+2    ; 将带进位的高 32 位的值 VarB 加到 ACC
ADDUL P,@VarC+0      ; 将无符号的低 32 位的值 VarC 加到 P
ADDCL ACC,@VarC+2    ; 将带进位的高 32 位的值 VarC 加到 ACC
MOVL @VarD+0,P       ; 保存结果的低 32 位到 VarD 中
MOVL @VarD+2,ACC     ; 保存结果的高 32 位到 VarD 中

```

MOVL XARn, loc32

装载 32 位辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL XAR0,loc32	1000 1110 LLLL LLLL	1	-	1
MOVL XAR1,loc32	1000 1011 LLLL LLLL	1	-	1
MOVL XAR2,loc32	1000 0110 LLLL LLLL	1	-	1
MOVL XAR3,loc32	1000 0010 LLLL LLLL	1	-	1
MOVL XAR4,loc32	1000 1010 LLLL LLLL	1	-	1
MOVL XAR5,loc32	1000 0011 LLLL LLLL	1	-	1
MOVL XAR6,loc32	1100 0100 LLLL LLLL	×	-	1
MOVL XAR7,loc32	1100 0101 LLLL LLLL	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器
loc32 寻址模式

描述: 用“loc32”寻址地址单元的值装载 XARn

XARn=[loc32];

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

MOVL XAR0,ACC        ; 将 ACC 的 32 位数移动到 XAR0
MOVL XAR2,*XAR0+ +    ; 将 XAR0 指向的 32 位数移动到 XAR2
                      ; XAR0 加 2
MOVL XAR3,*XAR3+ +    ; 将 XAR3 指向的 32 位数移动到 XAR3
                      ; XAR3 的寻址模式被忽略
MOVL XAR4,*--XAR4     ; XAR4 先减 2, 再
                      ; 将 XAR4 指向的 32 位数移动到 XAR4

```

MOVL XARn, #22bit

将常数装入 32 位辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL XAR0, #22bit	1000 1101 00CC CCCC CCCC CCCC CCCC CCCC	1	-	1

续表

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL XAR1, #22bit	1000 1101 01CC CCCC CCCC CCCC CCCC CCCC	1	-	1
MOVL XAR2, #22bit	1000 1101 10CC CCCC CCCC CCCC CCCC CCCC	1	-	1
MOVL XAR3, #22bit	1000 1101 11CC CCCC CCCC CCCC CCCC CCCC	1	-	1
MOVL XAR4, #22bit	1000 1111 00CC CCCC CCCC CCCC CCCC CCCC	1	-	1
MOVL XAR5, #22bit	1000 1111 01CC CCCC CCCC CCCC CCCC CCCC	1	-	1
MOVL XAR6, #22bit	0111 0110 10CC CCCC CCCC CCCC CCCC CCCC	×	-	1
MOVL XAR7, #22bit	0111 0110 11CC CCCC CCCC CCCC CCCC CCCC	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器
#22bit 22 位立即数

描述: 用一个 22 位无符号常数装载 XARn:

$XARn = 0:22bit;$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

MOVL XAR4, #VarA ;用 VarA 中的 22 位地址初始化 XAR4

MOVL XT, loc32

装载 XT 寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVL XT, loc32	1000 0111 LLLL LLLL	1	-	1

操作数: XT 被乘数寄存器 (XT) 的高 16 位
loc32 寻址模式

描述: 用“loc32”寻址模式指向的 32 位数装载到 XT 寄存器

$XT = [loc32];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 用 32 位的乘法计算, 保留高位结果

; $Y = (X0 \times C0) \gg 2) + (X1 \times C1 \gg 2) + (X2 \times C2 \gg 2)$

```

; X2 = X1
; X1 = X0
SPM -2           ; 设置乘积右移 2 位
ZAPA             ; ACC, P, OVC 清 0
MOVL XT,@X2      ; XT = X2
QMPYL P,XT,@C2   ; P = XT×C2
MOVL XT,@X1      ; XT = X1, ACC = X2×C2 >> 2
QMPYAL P,XT,@C1  ; P = XT×C1
MOVL @X2,XT      ; X2 = X1
MOVL XT,@X0      ; XT = X0, ACC = X1×C1 >> 2 + X2×C2 >> 2
QMPYAL P,XT,@C0  ; P = XT×C0
MOVL @X1,XT      ; X1 = X0
ADDL ACC,P << PM ; ACC = X0×C0 >> 2 + X1×C1 >> 2 + X2×C2 >> 2
MOVL @Y,ACC      ; 结果保存到 Y 中

```

MOVP T, loc16

装载 T 寄存器并将移位后的 P 装载到累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVP T,loc16	0001 0110 LLLL LLLL	×	-	1

操作数: T 被乘数寄存器(XT)的高 16 位
loc16 寻址模式

描述: 用“loc16”寻址地址单元中的 16 位数装载 T 寄存器。P 寄存器按乘积移位方式 (PM) 位指定的移位位数移位, P 寄存器的内容移位后装载到 ACC 累加器中。

```

T=[loc16];
ACC = P << PM;

```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。
Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。
PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 用 16 位乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
; X2 = X1
; X1 = X0
SPM -2           ; 设置乘积右移 2 位
MOV T,@X2        ; T = X2
MPY P,T,@C2      ; P = T×C2
MOVP T,@X1       ; T = X1, ACC = X2×C2 >> 2
MPY P,T,@C1      ; P = T×C1
MOV @X2,T        ; X2 = X1

```

```

MOVA  T,@X0          ; T = X0, ACC = X1xC1 >> 2 + X2xC2 >> 2
MPY   P,T,@C0         ; P = TxC0
MOV   @X1,T           ; X1 = X0
ADDL  ACC,P << PM     ; ACC = X0xC0 >> 2 + X1xC1 >> 2 + X2xC2 >> 2
MOVL  @Y,ACC          ; 结果保存到 Y 中

```

MOVS T, loc16

装载 T 寄存器并从累加器中减去 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVS T,loc16	0001 0001 LLLL LLLL	×	-	1

操作数: T 被乘数寄存器 (XT) 的高 16 位
loc16 寻址模式

描述: 用“loc16”寻址地址单元的 16 位数装载 T 寄存器。P 寄存器按乘积移位方式 (PM) 位指定的移位位数移位, 从 ACC 累加器中减去 P 寄存器移位后的值。

```

T=[loc16];
ACC = ACC - P << PM;

```

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

C 若产生进位, 则进位标志位 C 置位; 否则清 0。

V 若产生溢出, 则溢出标志位 V 置位; 否则清 0。

OVC 若禁止溢出模式, 且操作产生正溢出, 计数器值增加; 若操作产生负溢出, 计数器值减少。

OVM 若溢出模式位置 1, 当运行发生溢出时, ACC 值将为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 它将执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。发生溢出时, V 标志位立即置 1。在每次重复操作时, 程序存储器地址加 1。

例:

```

; 用 16 位乘法进行计算
; Y = (X0xC0) >> 2) + (X1xC1 >> 2) + (X2xC2 >> 2)
; X2 = X1
; X1 = X0
SPM   -2          ; 设置乘积右移 2 位
MOVPS T,@X2       ; T = X2
MPYS  P,T,@C2     ; P = TxC2, ACC = 0
MOVS  T,@X1       ; T = X1, ACC = -X2xC2 >> 2
MPY   P,T,@C1     ; P = TxC1
MOV   @X2,T       ; X2 = X1

```



```

MOVA  T,@X0          ; T = X0, ACC = -X1xC1 >> 2 - X2xC2 >> 2
MPY   P,T,@C0        ; P = TxC0
MOV   @X1,T          ; X1 = X0
SUBL  ACC,P << PM     ; ACC = -X0xC0 >> 2 - X1xC1 >> 2 - X2xC2 >> 2
MOVL  @Y,ACC         ; 结果保存到 Y 中

```

MOVU ACC, loc16

用无符号字装载累加器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVU ACC,loc16	0000 1110 LLLL LLLL	×	-	1

操作数: ACC 累加器
loc16 寻址模式

描述: 用“loc16”寻址地址单元的 16 位数装载累加器的低位 (AL)。并将累加器的高位 (AH) 填充 0。

```

AL=[loc16];
AH = 0x0000;

```

标志与模式: N 标志位清 0。

Z 若 ACC 值为 0，则零标志位 Z 置位；否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

;用 16 位分部相加方法得到 3 个 32 位无符号变量的和:

```

MOVU ACC,@VarA_low      ; AH = 0, AL = VarA 低
ADD  ACC,@VarA_high << 16 ; AH = VarA 高, AL = VarA 低
ADDU ACC,@VarB_low      ; ACC = ACC + 0:VarB 低
ADD  ACC,@VarB_high << 16 ; ACC = ACC + VarB 高 << 16
ADDU ACC,@VarC_low      ; ACC = ACC + VarC 低 + 进位
ADD  ACC,@VarC_high << 16 ; ACC = ACC + VarC 高 << 16

```

MOVU loc16, OVC

保存无符号溢出计数器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVU loc16,OVC	0101 0110 0010 1000 0000 0000 LLLL LLLL	1	-	1

操作数: loc16 寻址模式
OVC 溢出计数器

描述: 保存溢出计数器 (OVC) 的 6 位到“loc16”寻址地址单元的低 6 位中并且高 10 位置 0。

```

[loc16 (15:6)] = 0;
[loc16 (5:0)] = OVC;

```

标志与模式: N 若 (loc16=@ACC) 且 ACC 累加器的第 15 位为 1，则负标志位 N 置位，否则清 0。

Z 若 (loc16=@ACC) 且 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;保存并恢复 ACC 和 OVC 的值
MOVU *SP+, OVC    ;保存 OVC 到堆栈中
MOV  *SP+, AL      ;保存 AL 到堆栈中
MOV  *SP+, AH      ;保存 AH 到堆栈中
.
.
.
.
MOV  AH, *- -SP    ;从堆栈中恢复 AH
MOV  AL, *- -SP    ;从堆栈中恢复 AL
MOVU OVC, *- -SP   ;从堆栈中恢复 OVC
```

MOVU OVC, loc16

用无符号值装载溢出计数器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVU OVC, loc16	0101 0110 0110 0010 0000 0000 LLLL LLLL	1	-	1

操作数: OVC 6 位溢出计数器

描述: 用“loc16”寻址地址单元的低 6 位装载溢出计数器 (OVC)。

$OVC = [loc16(5:0)]$;

标志与模式: OVC 修改 6 位溢出计数器

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;保存并恢复 ACC 和 OVC 的值
MOVU *SP+, OVC    ;保存 OVC 到堆栈中
MOV  *SP+, AL      ;保存 AL 到堆栈中
MOV  *SP+, AH      ;保存 AH 到堆栈中
.
.
.
.
MOV  AH, *- -SP    ;从堆栈中恢复 AH
MOV  AL, *- -SP    ;从堆栈中恢复 AL
MOVU OVC, *- -SP   ;从堆栈中恢复 OVC
```

MOVW DP, #16bit

装载数据页

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVW DP, #16bit	0111 0110 0001 1111 CCCC CCCC CCCC CCCC	×	-	1

操作数: DP 数据页寄存器
#16bit 16 位立即数
描述: 用 16 位常数装载数据页寄存器
DP(15:0) =16bit;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
MOVW DP, #VarA ; 将 VarA 的值装载到 DP
                ; 假定 VarA 位于存储器低 0x003F FFC0 区域
MOVW DP, #0F012h ; 装载数据页 DP 为 0xF012
```

MOVX TL,loc16

带符号扩展装载 XT 寄存器的低 16 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVX TL,loc16	0101 0110 0010 0001 xxxx xxxx LLLL LLLL	1	-	1

操作数: TL 被乘数寄存器的低 16 位
loc16 寻址模式

描述: 用“loc16”寻址地址单元的值装载被乘数寄存器 (XT) 的低 16 位, 对 XT 的高 16 位进行符号扩展。

TL = [loc16];
T = TL 的符号扩展;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算并保留结果的低 32 位 Y32 = M32×X16
MOVX TL,@X16 ; XT = S:X16
IMPYL P,XT,@M32 ; P = XT × M32 (结果的低 32 位)
MOVL @Y32,P ; 结果保存到 Y32 中
```

MOVZ ARn,loc16

装载 XARn 的低半部分并将高半部分清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVZ AR0...5,loc16	0101 1nnn LLLL LLLL	×	-	1
MOVZ AR6,loc16	1000 1000 LLLL LLLL	1	-	1
MOVZ AR7,loc16	1000 0000 LLLL LLLL	1	-	1

操作数: ARn AR0 ~AR7, 辅助寄存器的低 16 位
loc16 寻址模式

描述: 用“loc16”寻址地址单元的内容装载 ARn 并且 ARnH 清 0;

```
ARn = [loc16];
ARnH = 0;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
MOVL  XAR7, #ArrayA      ; 初始化 XAR2 指针
MOVZ  AR0,  *+XAR2[0]     ; 装载由 XAR2 指向的 16 位数到 AR0 中。
                          ; XAR0(31:16) = 0.
MOVZ  AR7,  *-SP[1]       ; 装载堆栈的第一个 16 位数到 AR7 中
                          ; XAR7(31:16) = 0.
```

MOVZ DP, #10bit

装载数据页并将高 6 位清 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MOVZ DP, #10bit	1011 10CC CCCC CCCC	1	-	1

操作数: DP 数据页寄存器
#10bit 10 位立即数

描述: 用一个 10 位常数装载数据页寄存器并将高 6 位清 0:

```
DP(9:0) = 10bit;
DP(15:10) = 0;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
MOVZ  DP, #VarA          ; 装载 DP 为包含 VarA 的数据页
                          ; 假定 VarA 在存储器的低 0x0000 FFC0 中
MOVZ  DP, #3FFh          ; 装载数据页 DP 为 0x03FF
```

MPY ACC, loc16, #16bit

16×16 位乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPY ACC, loc16, #16bit	0011 0100 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	1

操作数: ACC 累加器
loc16 寻址模式
#16bit 16 位立即数

描述: 用“loc16”寻址地址单元的 16 位数装载 T 寄存器, 接着乘以有符号 16 位立即数。

```
T = [loc16];
ACC = 有符号 T x 有符号 16bit;
```

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 用 16 位乘法计算有符号数
; Y32 = Y32 + X16 × 2000
MPY ACC,@X16,#2000      ; T = X16, ACC = X16 × 2000
ADDL @Y32,ACC            ; Y32 = Y32 + ACC
```

MPY ACC,T,loc16

16×16 位乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPY ACC,T,loc16	0001 0110 LLLL LLLL	×	-	1

操作数: ACC 累加器
T 被乘数寄存器
loc16 寻址模式

描述: 用“loc16”地址单元的 16 位数乘以 T 寄存器中的有符号 16 位数, 结果保存到 ACC 中。

ACC = 有符号 T × 有符号[loc16];

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 用 16 位乘法计算有符号数
; Y32 = Y32 + X16×M16
MOV T,@X16      ; T = X16
MPY ACC,T,@M16  ; ACC = T × M16
ADDL @Y32,ACC   ; Y32 = Y32 + ACC
```

MPY P,loc16,#16bit

16×16 位乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPY P, loc16, #16bit	1000 1100 LLLL LLLL CCCC CCCC CCCC CCCC	1	-	1

操作数: P 乘积寄存器
loc16 寻址模式
#16bit 16 位立即数

描述: 用“loc16”寻址地址单元的有符号的 16 位数乘以有符号的 16 位数, 且 32 位的结果保存在 P 寄存器中。

P = 有符号[loc16] × 有符号 16 位数;

标志与模式 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 用 16 位乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2);
; C0, C1 和 C2 是常数
SPM    -2                      ; 设置乘积右移 2 位
MOVB   ACC, #0                 ; ACC 清 0
MPY     P, @X2, #C2            ; P = X2×C2
MPYA    P, @X1, #C1            ; ACC = X2×C2>>2, P = X1×C1
MPYA    P, @X0, #C0            ; ACC = X1×C1>>2 + X2×C2>>2, P = X0×C0
ADDL    ACC, P << PM           ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL    @Y, ACC                ; 结果保存在 Y 中
```

MPY P,T,loc16

16×16 位乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPY P, T, loc16	0011 0011 LLLL LLLL	×	-	1

操作数: P 乘积寄存器
T 被乘数寄存器
loc16 寻址模式

描述: 用“loc16”寻址地址单元的有符号的 16 位数乘以 T 寄存器的有符号的 16 位数, 且 32 位的结果保存在 P 寄存器中。

$P = \text{有符号 } T \times \text{有符号}[\text{loc16}];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 用 16 位乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
; X2 = X1
; X1 = X0
SPM    -2                      ; 设置乘积右移 2 位
MOVP   T, @X2                 ; T = X2
MPYS    P, T, @C2             ; P = T×C2, ACC = 0
MOVAD   T, @X1                 ; T = X1, ACC = X2×C2>>2, X2 = X1
MPY     P, T, @C1             ; P = T×C1
MOVAD   T, @X0                 ; T = X0, ACC = X1×C1>>2 + X2×C2>>2,
; X1 = X0
MPY     P, T, @C0             ; P = T×C0
ADDL    ACC, P << PM           ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL    @Y, ACC                ; 结果保存在 Y 中
```

MPYA P,loc16,#16bit

16×16 位的乘法并加上先前的乘积

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYA P,loc16,#16bit	0001 0101 LLLL LLLL CCCC CCCC CCCC CCCC	×	-	1

操作数: P 乘积寄存器
loc16 寻址模式
#16bit 16 位立即数

描述: 将先前的乘积（保存在 P 寄存器中）按乘积移位模式（PM）位指定位数移位后加到 ACC 累加器。用“loc16”地址单元的有符号的 16 位数装载 T 寄存器。用有符号的 16 位常数乘以 T 寄存器的有符号 16 位数，且 32 位的结果保存在 P 寄存器中。

$$ACC = ACC + P \ll PM;$$

$$T = [loc16];$$

$$P = \text{有符号}[loc16] \times \text{有符号} 16 \text{ 位数};$$

标志与模式: Z 若 ACC 值为 0，则零标志位 Z 置位；否则清 0。
N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则清 0。
C 若产生进位，则进位标志位 C 置位；否则清 0。
V 若产生溢出，则溢出标志位 V 置位；否则清 0。
OVC 若禁止溢出模式，且操作产生正溢出，计数器值增加；操作产生负溢出，计数器值减少。
OVM 若溢出模式位置 1，当运行发生溢出时，ACC 值将为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。
PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正（逻辑左移操作），最低位填 0；若乘积移位位数为负（逻辑右移操作），则高位进行符号扩展。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```

; 用 16 位乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2),
; C0, C1 和 C2 是常数
SPM    -2                ; 设置乘积右移 2 位
MOVB   ACC,#0            ; ACC 清 0
MPY    P,@X2,#C2         ; P = X2×C2
MPYA   P,@X1,#C1         ; ACC = X2×C2>>2, P = X1×C1
MPYA   P,@X0,#C0         ; ACC = X1×C1>>2 + X2×C2>>2, P = X0×C0
ADDL   ACC,P << PM       ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL   @Y,ACC            ; 结果保存在 Y 中

```

MPYA P,T,loc16**16×16 位的乘法并加上先前的乘积**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYA P, T, #loc16	0001 0111 LLLL LLLL	×	Y	N+1

操作数: P 乘积寄存器
 T 被乘数寄存器
 loc16 寻址模式

描述: 将先前的乘积（保存在 P 寄存器中）按乘积移位模式（PM）位指定位数移位后加到 ACC 累加器。用“loc16”寻址地址单元中的有符号 16 位常数乘以 T 寄存器的有符号 16 位数，32 位结果保存在 P 寄存器中。

ACC = ACC + P << PM;

P = 有符号 T × 有符号[loc16]

标志与模式: Z 若 ACC 值为 0，则零标志位 Z 置位；否则清 0。
 N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则清 0。
 C 若产生进位，则进位标志位 C 置位；否则清 0。
 V 若产生溢出，则溢出标志位 V 置位；否则清 0。
 OVC 若禁止溢出模式，并且操作产生正溢出，计数器值增加；操作产生负溢出，计数器值减少。
 OVM 若溢出模式位置 1，当运行发生溢出时，ACC 值将为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。
 PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正（逻辑左移操作），最低位填 0；若乘积移位位数为负（逻辑右移操作），则高位进行符号扩展。
 重复性: 本指令可以重复。若指令跟在 RPT 指令之后，它将执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。发生溢出时，V 标志位立即置 1。

例:

```

; 用 16 位乘法进行计算
; Y = (X0×C0) >> 2) + (X1×C1 >> 2) + (X2×C2 >> 2)
SPM    -2                ; 设置乘积右移 2 位
MOVP   T,@X2             ; ACC = P, T = X2
MPYS   P,T,@C2           ; ACC = ACC - P = 0, P = T×C2
MOV    T,@X1             ; T = X1
MPYA   P,T,@C1           ; ACC = X2×C2>>2, P = T×C1
MOV    T,@X0             ; T = X0
MPYA   P,T,@C0           ; ACC = X1×C1>>2 + X2×C2>>2, P = T×C0
ADDL   ACC,P << PM      ; ACC = X0×C0>>2 + X1×C1>>2 + X2×C2>>2
MOVL   @Y,ACC            ; 结果保存到 Y 中

```

MPYB ACC,T,#8bit**乘 8 位立即数**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYB ACC,T,#8bit	0011 0101 CCCC CCCC	×	-	1

- 操作数: ACC 累加器
T 被乘数寄存器
#8bit 8 位立即常数
- 描述: 无符号 8 位常数 0 扩展后乘以 T 寄存器的有符号 16 位数, 结果保存到 ACC 累加器:

$$ACC = \text{signed } T \times 0:8\text{bit};$$
- 标志与模式: Z 若 ACC 为 0, 则零标志位 Z 置位; 否则 Z 清 0。
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 使用 16 位乘计算符号
; Y32 = Y32 + (X16×5)
MOV    T,@X16          ; T = X16
MPYB   ACC,T,#5         ; ACC = T × 5
ADDL   @Y32,ACC         ; Y32 = Y32 + ACC
```

MPYB P,T,#8bit

8 位无符号数与有符号数相乘

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYB P,T,#8bit	0011 0001 CCCC CCCC	×	-	1

- 操作数: P 乘积寄存器
T 被乘数寄存器
#8bit 8 位立即常数
- 描述: 无符号 8 位立即常数 0 扩展乘以 T 寄存器的有符号 16 位数, 保存 32 位结果到 P 寄存器:

$$P = \text{signed } T \times 0:8\text{bit};$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 Y32 = X16 × 5;
MOV    T,@X16          ; T = X16
MPYB   P,T,#5          ; P = T × 5
MOVL   @Y,P            ; 结果保存到 Y32
```

MPYS P,T,loc16

16×16 乘减

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYS P,T,loc16	0001 0011 LLLL LLLL	×	Y	N+1

操作数: P 乘积寄存器

T 被乘数寄存器

loc16 寻址方式

描述: 从 ACC 累加器中减去按乘积移位方式 (PM) 指定的移位位数移位后的先前乘积 (保存在 P 寄存器)。用有 “loc16” 寻址地址单元中的符号 16 位数乘以 T 寄存器的有符号 16 位数, 结果保存到 P 寄存器:

$$ACC = ACC - P \ll PM;$$

$$P = \text{signed } T \times \text{signed } [\text{loc16}];$$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 若产生借位, 则进位标志位 C 置位; 否则进位标志位 C 清 0。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVC 若禁止溢出模式, 且操作产生正溢出, 计数器值增加; 操作产生负溢出, 计数器值减少。

OVM 若溢出模式位置 1, 当运行发生溢出时, ACC 值将为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

PM PM 值设置了乘积寄存器输出时移位操作的模式。若乘积移位位数为正 (逻辑左移操作), 最低位填 0; 若乘积移位位数为负 (逻辑右移操作), 则高位进行符号扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则指令执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。若中间产生溢出, 则 V 标志位立即置位。

例:

```

; 使用 16 位乘计算
; Y = (X0xC0) >> 2) + (X1xC1 >> 2) + (X2xC2 >> 2)
SPM -2                ; 设置乘积右移 2 位
MOVP    T,@X2          ; ACC = P, T = X2
MPYS    P,T,@C2        ; ACC = ACC - P = 0, P = TxC2
MOV      T,@X1          ; T = X1
MPYA    P,T,@C1        ; ACC = X2xC2>>2, P = TxC1
MOV      T,@X0          ; T = X0
MPYA    P,T,@C0        ; ACC = X1xC1>>2 + X2xC2>>2, P = TxC0
ADDL    ACC,P << PM    ; ACC = X0xC0>>2 + X1xC1>>2 + X2xC2>>2
MOVL    @Y,ACC         ; 结果保存到 Y

```

MPYU P,T,loc16

无符号 16×16 乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYU P,T,loc16	0011 0111 LLLL LLLL	×	-	1

操作数: P 乘积寄存器
T 被乘数寄存器
loc16 寻址方式

描述: 用“loc16”地址单元的无符号 16 位数乘以 T 寄存器中的无符号 16 位数, 32 位结果保存到 P 寄存器:

$P = \text{unsigned } T \times \text{unsigned } [\text{loc16}];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算无符号值 Y32 = X16 × M16;
MOV    T,@X16      ; T = X16
MPYU   P,T,@M16    ; P = T × M16
MOVL   @Y,P        ; 结果保存到 Y32
```

MPYU ACC,T,loc16

无符号 16×16 乘法

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYU ACC,T,loc16	0011 0110 LLLL LLLL	×	-	1

操作数: ACC 累加器
T 被乘数寄存器
loc16 寻址方式

描述: 用“loc16”地址单元的无符号 16 位数乘以 T 寄存器的无符号 16 位数, 32 位结果保存到 ACC 累加器。

$ACC = \text{unsigned } T \times \text{unsigned } [\text{loc16}];$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 使用 16 位乘法计算无符号数
; Y32 = Y32 + X16×M16
MOV    T,@X16      ; T = X16
MPYU   ACC,T,@M16  ; ACC = T × M16
ADDL   @Y32,ACC    ; Y32 = Y32 + ACC
```

MPYXU ACC,T,loc16

有符号数乘以无符号数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYXU ACC,T,loc16	0011 0000 LLLL LLLL	×	-	1

操作数: ACC 累加器
T 被乘数寄存器
loc16 寻址方式

描述: 用“loc16”地址单元的无符号 16 位数乘以 T 寄存器中的有符号 16 位数,

结果保存到 ACC 累加器:

$ACC = \text{signed } T \times \text{unsigned } [loc16];$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 使用 16 位乘法计算有符号数
; Y32 = Y32 + (有符号) X16 × (无符号) M16
MOV      T,@X16      ; T = X16
MPYXU    ACC,T,@M16   ; ACC = T × M16
ADDL     @Y32,ACC     ; Y32 = Y32 + ACC
```

MPYXU P,T,loc16

有符号数乘无符号数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
MPYXU P, T, loc16	0011 0010 LLLL LLLL	×	-	1

操作数: P 乘积寄存器
T 被乘数寄存器
Loc16 寻址方式

描述: 用“loc16”地址单元的有符号 16 位数乘以 T 寄存器中的有符号 16 位数, 32 位结果保存到 P 寄存器:

$P = \text{signed } T \times \text{unsigned } [loc16];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 使用 16 位分部乘法计算 "Y32 = X32 × M32"
MOV      T,@X32+0     ; T = 无符号低位 X32
MPYU     ACC,T,@M32+0  ; ACC = T × 无符号低位 M32
MOV      @Y32+0,AL     ; 保存低位结果到 Y32
MOVU     ACC,@AH       ; 逻辑右移 ACC 16 位
MOV      T,@X32+1     ; T = 有符号高位 X32
MPYXU    P,T,@M32+0    ; ACC = T × 无符号低位 M32
MOVA     T,@M32+1     ; T = 有符号高位 M32, ACC += P
MPYXU    P,T,@X32+0    ; ACC = T × 无符号低位 X32
ADDL     ACC,@P        ; 加 P 到 ACC
MOV      @Y32+1,AL     ; 保存高位结果到 Y32
```

NASP

不对齐堆栈指针

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NASP	0111 0110 0001 0111	×	-	1

操作数: 无
描述: 若 SPA 位为 1, NASP 指令使堆栈指针 SP 减 1, 然后 SPA 状态位清 0。撤销 ASP 指令先前执行的堆栈指针队列。若 SPA 状态位为 0, 则不执行 NASP 指令。

```
if( SPA = 1 )
{
    SP = SP - 1;
    SPA = 0;
}
```

标志与模式: SPA 若 (SPA = 1), 则 SPA 状态位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 中断服务程序的堆栈指针队列
; 向量表
INTx: .long INTxService      ; INTx 中断向量
.
.
INTxService:
    ASP                      ; 排列堆栈指针
.
.
.
    NASP                    ; 重新排列堆栈指针
    IRET                   ; 从中断返回。
```

NEG ACC

累加器取负

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NEG ACC	1111 1111 0101 0100	×	-	1

操作数: ACC 累加器
描述: ACC 寄存器的内容求负

```
if(ACC = 0x8000 0000)
{
    V = 1;
    if(OVM = 1)
        ACC = 0x7FFF FFFF;
else
    ACC = 0x8000 0000;
}
else
    ACC = -ACC;
if(ACC = 0x0000 0000)
    C = 1;
else
```

C = 0;

- 标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 C 若 ACC 的值为 0, 则进位标志位 C 置位; 否则进位标志位 C 清 0。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 OVM 若操作开始时 (ACC = 0x8000 0000), 这认为是一个溢出值, 指令执行后 ACC 的值取决于 OVM 的状态, 若 OVM 清 0, 则 ACC 的值为 0x80000000。若 OVM 置位, ACC 为 0x7FFFFFFF。
 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 对 VarA 的内容取负, 确保值为饱和值
MOVL  ACC, @VarA      ; 用 VarA 的内容装载 ACC
SETC   OVM             ; 溢出模式使能
NEG ACC                ; 对 ACC 取负并使其为饱和值
MOVL  @VarA, ACC       ; 结果保存到 VarA
```

NEG AX

AX 寄存器的内容求负

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NEG AX	1111 1111 0101 110A	×	-	1

- 操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
 描述: 对 AX 求负

```
if (AX = 0x8000)
{
    AX = 0x8000;
    V = 1;
}
else
    AX = -AX;
if (AX = 0x0000)
    C = 1;
else
    C = 0;
```

- 标志与模式: N 若 AX 的第 15 位为 1, 负标志位 N 置位; 否则清 0。
 Z 若 AX 为 0, 则零标志位 Z 置位; 否则清 0。
 C 若 AX 为 0, 则进位标志位 C 置位; 否则进位标志位 C 清 0。
 V 若 AX 在操作开始时为 0x8000, 则认为发生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 取 VarA 的绝对值
```

```

MOV AL,@VarA          ; 用 VarA 的内容装载 AL
NEG AL                ; 若 AL = 8000h, 则 V = 1
SB NoOverflow,NOV     ; 若无溢出, 则跳转到保存 -AL
MOV @VarA,0x7FFFh     ; 若溢出, 置为 7FFF
NoOverflow:
MOV @VarA,AL          ; 若无溢出, 保存 -AL

```

NEG64 ACC:P

寄存器 ACC:P 的组合值求负

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NEG64 ACC:P	0101 0110 0101 1000	1	-	1

操作数: ACC:P 累加器 (ACC) 和乘积寄存器 (P)

描述: 对 ACC:P 组合寄存器求负数

```

if(ACC:P = 0x8000 0000 0000 0000)
{
    V = 1;
    if(OVM = 1)
        ACC:P = 0x7FFF FFFF FFFF FFFF;
    else
        ACC:P = 0x8000 0000 0000 0000;
}
else
    ACC:P = -ACC:P;
if(ACC:P = 0x0000 0000 0000 0000)
    C = 1;
else
    C = 0;

```

标志与模式: N 若 ACC 的第 31 位为 1, 则 ACC:P 为负值, 负标志 N 被置位; 否则 N 清 0。

Z 若 ACC:P 的组合 64 位数为 0, 则零标志位 Z 置位; 否则 Z 清 0。

C 若 (ACC:P = 0), 则进位标志位 C 置位; 否则进位标志位 C 清 0。

V 若 (ACC:P = 0x8000 0000 0000 0000), 则溢出标志位 V 置位; 否则不变。

OVM 若操作开始时 ACC:P = 0x8000 0000 0000 0000, 代表一个溢出值, 指令执行后 ACC:P 的值取决于 OVM 的状态, 若 OVM = 1, ACC:P 为最大正数 (0x7FFF FFFF FFFF FFFF)。若 OVM = 0, ACC:P 不改变。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 对 64 位的 Var64 内容取负并使其为饱和值

```

MOVL ACC, @Var64+2 ; 用 Var64 的高 32 位装载 ACC
MOVL P, @Var64+0   ; 用 Var64 的低 32 位装载 P
SETC OVM           ; 溢出模式使能

```



```

NEG64  ACC:P          ; 对 ACC:P 取负
MOVL   @Var64+2,ACC   ; 保存结果的高 32 位到 Var64
MOVL   @Var64+0,P     ; 保存结果的低 32 位到 Var64

```

NEGTC ACC

若 TC = 1, 求 ACC 的负数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NEGTC ACC	0101 0110 0011 0011	1	-	1

操作数: ACC 累加器

描述: 根据测试控制位 TC 的状态, 有条件地对 ACC 累加器内容取负数

```

if( TC = 1 )
{
    if(ACC = 0x8000 0000)
    {
        V = 1;
        if(OVM = 1)
            ACC = 0x7FFF FFFF;
        else
            ACC = 0x8000 0000
    }
    else
        ACC = -ACC;
    if(ACC = 0x0000 0000)
        C = 1;
    else
        C = 0;
}

```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 Z 若 ACC 为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 C 若 TC = 1 且 ACC = 0, 则进位标志位 C 置位; 若 TC = 1 且 ACC ≠ 0, 则进位标志位 C 清 0; 否则 C 不改变。
 V 当操作开始时, 若 TC = 1 且 ACC = 0x8000 0000, 这认为是一个溢出值, 则溢出标志位 V 置位。否则不影响 V。
 TC 操作时 TC 位的状态用作测试条件。
 OVM 若操作开始时, ACC = 0x8000 0000 0000 0000, 认为是一个溢出值, 指令执行后 ACC 的值取决于 OVM 的状态。若 OVM 为 0 且 TC = 1, ACC 为 0x8000 0000。若 OVM 置位且 TC = 1, ACC 为 0x7FFF FFFF。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 计算有符号数 Quot16 = Num16/Den16, Rem16 = Num16%Den16
CLRC   TC              ; 标志位 TC 清 0
MOV    ACC,@Den16 << 16 ; AH = Den16, AL = 0

```

ABSTC	ACC	; 取绝对值, TC = sign 异或 TC
MOV	T, @AH	; 保存 Den16 到 T 寄存器
MOV	ACC, @Num16 << 16	; AH = Num16, AL = 0
ABSTC	ACC	; 取绝对值, TC = sign 异或 TC
MOVU	ACC, @AH	; AH = 0, AL = Num16
RPT	#15	; 重复下面操作 16 次
SUBCU	@T	; 用 Den16 作条件减法
MOV	@Rem16, AH	; 保存余数到 Rem16
MOV	ACC, @AL << 16	; AH = Quot16, AL = 0
NEGTC	ACC	; 若 TC = 1 取负
MOV	@Quot16, AH	; 保存商到 Quot16

NOP {*ind}{, ARPn}

可修改间接寻址的空操作

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NOP {*ind}{, ARPn}	0111 0111 LLLL LLLL	X	Y	N+1

操作数: {*ind} 间接寻址方式

ARPn 辅助寄存器指针 (ARP0~ARP7)

描述: 修改指定的间接寻址操作数, 改变辅助寄存器指针 (ARP) 到给定的辅助寄存器。若没有给定操作数, 不操作。

标志与模式: 无

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 将执行 N+1 次。

例:

```

; 复制 Array1 的内容到 Array2
; int32 Array1[N];
; int32 Array2[N];
; for(i = 0; i < N; i+ +)
; Array2[i] = Array1[i];
; 本例只能在程序空间的高 64K 工作
MOVL XAR2, #Array1      ; XAR2 指针指向 Array1
MOVL XAR3, #Array2      ; XAR3 指针指向 Array2
MOV @AR0, # (N-1)       ; 重复 loop 循环 N 次
NOP *, ARP2              ; 指针指到 XAR2 (ARP = 2)
SETC AMODE               ; 全兼容 C2XLP 寻址方式
Loop:
MOVL ACC, *              ; ACC = Array1[i]
NOP *+ +, ARP3           ; 增加 XAR2 并且指针指向 XAR3
RPT #19                  ; 空操作占用 20 个周期
||NOP
MOVL *+ +, ACC, ARP0     ; Array2[i] = ACC, 指针指向 XAR0
XBANZ Loop, *- -, ARP2   ; 若 AR[ARP] != 0, 则 AR[ARP]- -, 跳到 Loop
; 指针指向 XAR2

```

NORM ACC, *ind

规格化 ACC 并且修改辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NORM ACC, *	0101 0110 0010 0100	1	Y	N+4
NORM ACC, *+ +	0101 0110 0101 1010	1	Y	N+4

续表

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NORM ACC,*--	0101 0110 0010 0000	1	Y	N+4
NORM ACC,*0++	0101 0110 0111 0111	1	Y	N+4
NORM ACC,*0--	0101 0110 0011 0000	1	Y	N+4

操作数: ACC 累加器

*ind *,*+ +,*--,*0+ +,*0-- 间接寻址方式

描述: 规格化 ACC 累加器的有符号数, 按间接寻址方式修改辅助寄存器指针 (ARP) 指向的辅助寄存器 (XAR0~XAR7):

注意: 通过查找数的大小, NORM 指令规格化 ACC 累加器中的有符号数。对 ACC 的第 31 位和 30 位进行“异或”。若这两位相同, ACC 累加器的值逻辑左移 1 位, 消除额外的符号位并修改所选择的指针。若这两位不相同, ACC 不移位, 不改变所选择的指针。所选择的指针不会访问任何存储器位置。

标志与模式: Z 若 ACC 为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

TC 若操作置位 TC, 不需要规格化 (ACC 不需要改变)。若操作将 TC 清 0, 第 31 位与第 30 位相同时, ACC 累加器逻辑左移 1 位。

ARP 辅助寄存器指针选择哪一个辅助寄存器参与操作 (XAR0~XAR7)。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则 NORM 指令执行 N+1 次。Z、N、TC 的状态为最终的结果。若只希望 NORM 指令在规格化后执行, 可以建立一个循环, 检查 TC 位。当 TC = 1, 规格化完成。

例:

```

; 规格化 VarA 的内容
; XAR2 将包含操作最后的移位位数
MOVL ACC,@VarA          ; ACC = VarA
MOVB XAR2,#0             ; 初始化 XAR2 为 0
NOP *,ARP2               ; 设置 ARP 指针指向 XAR2
SBF Skip,EQ              ; 若 ACC 的值为 0, 则跳过
RPT #31                  ; 重复下一条指令 32 次
||NORM ACC,*+ +          ; 规格化 ACC 的内容

```

Skip:

NORM ACC,XARn+ +/- -

规格化 ACC 并修改辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NORM ACC,XARn+ +	1111 1111 0111 1nnn	×	Y	N+4
NORM ACC,XARn--	1111 1111 0111 0nnn	×	Y	N+4

操作数: ACC 累加器

XARn XAR0~XAR7, 辅助寄存器增加或减少

+ +/- -

描述: 规格化 ACC 累加器的有符号数并改变指定的辅助寄存器 (XAR0~XAR7)

```

if(ACC != 0x0000 0000)
{
    if((ACC(31) XOR ACC(30)) == 0)
    {
        ACC = ACC << 1, TC = 0;
        if(XARn+ + addressing mode) XARn + = 1;
        if(XARn- - addressing mode) XARn - = 1;
    }
    else
        TC = 1;
}
else
    TC = 1;

```

注意: 通过查找数的大小, NORM 指令规格化 ACC 累加器中的符号数。对 ACC 的第 31 位和 30 位进行“异或”。若这两位相同, ACC 累加器的值逻辑左移 1 位, 消除额外的符号位并修改所选择的指针。若这两位不相同, ACC 不移位, 不改变所选择的指针。所选择的指针不会访问任何存储器。

标志与模式: Z 若 ACC 为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

TC 若操作将 TC 置 1, 则不需要规格化 (即 ACC 不需要修改)。若操作将 TC 清 0, 且第 31 位和第 30 位是相同的, 结果 ACC 累加器逻辑左移 1 位。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则 NORM 指令执行 N+1 次。Z、N 和 TC 的状态为最终结果。若只希望 NORM 指令在规格化后执行, 可以建立一个循环, 检查 TC 位的值。当 TC = 1, 规格化完成。

例:

```

; 规格化 VarA 的内容, 操作最后, XAR2 包含移位位数
MOVL ACC, @VarA      ; ACC = VarA
MOVB XAR2, #0         ; 初始化 XAR2 到 0
SBF Skip, EQ          ; 若 ACC 的值为 0, 跳过
RPT #31               ; 重复下一条指令 32 次
||NORM ACC, XAR2+ +    ; 规格化 ACC 的内容
Skip:

```

NOT ACC

累加器求“非”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NOT ACC	1111 1111 0101 0101	×	-	1

操作数: ACC 累加器

描述: ACC 寄存器的内容求“非”:

ACC = ACC XOR 0xFFFFFFFF;

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 ACC 为 0, 则零标志位 Z 置位; 否则 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; VarA 内容求“非”
MOVL  ACC,@VarA      ; ACC = VarA
NOT    ACC             ; ACC 内容求“非”
MOVL  @VarA,ACC       ; 结果保存到 VarA
```

NOT AX

AX 寄存器内容求“非”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
NOT AX	1111 1111 0101 111A	×	-	1

操作数: AX 累加器高字节 (AH) 或累加器低字节 (AL) 寄存器

描述: 用其“非”取代指定 AX 寄存器 (AH 或 AL) 的内容:

$AX = AX \text{ XOR } 0xFFFF;$

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 AX 为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;对 VarA 的内容求“非”
MOVL  AL,@VarA        ;用 VarA 的内容装载 AL
NOT    AL              ;对 AL 内容求“非”
MOVL  @VarA,AL        ;将结果保存到 VarA 中
```

OR ACC, loc16

按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR ACC, loc16	1010 1111 LLLL LLLL	1	Y	N+1

操作数: ACC 累加器

loc16 寻址方式

描述: “loc16”地址单元的内容零扩展后与 ACC 的值作按位“或”操作。结果保存到 ACC 累加器:

$ACC = ACC \text{ OR } 0: [loc16];$

标志与模式: N 测试 ACC 装载后的值是否为负。若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 测试 ACC 装载后的值是否为 0。若操作结果使得 $ACC = 0$, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则指令将执行 N+1 次。Z、

N 的状态为最后的结果。

例:

```
;计算 32 位的值 VarA = VarA OR 0: VarB
MOVL    ACC,@VarA          ;用 VarA 的内容装载 ACC
OR       ACC,@VarB          ;用 0: VarB 的内容与 ACC 进行“或”操作
MOVL     @VarA,ACC          ;将结果保存到 VarA 中
```

OR ACC, #16bit << #0...16

带移位功能的按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR ACC,#16bit << #0...15	0011 1110 0001 SHFT CCCC CCCC CCCC CCCC	1	-	1
OR ACC,#16bit << #16	0101 0110 0100 1010 CCCC CCCC CCCC CCCC	1	-	1

操作数: ACC 累加器

#16bit 16 位立即常数

#0...16 移位位数 (没有指定值时, 则默认值为“<<#0”)

描述: 在指令执行前对其进行零扩展, 将给定的 16 位无符号数按指定位数左移后在 ACC 中执行按位“或”操作。结果保存到 ACC 累加器:

ACC = ACC OR (0:16bit <<移位位数);

标志与模式: N 测试 ACC 装载后的值是否为负。若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 测试 ACC 装载后的值是否为 0。若操作结果使得 ACC = 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;计算 32 位数 VarA = VarA OR 0x08000000
MOVL    ACC,@VarA          ;用 VarA 的内容装载 ACC
OR       ACC,#0x8000 << 12 ;用 0x08000000 与 ACC 进行“或”操作
MOVL     @VarA,ACC          ;将结果保存到 VarA 中
```

OR AX, loc16

按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR AX, loc16	1100 101A LLLL LLLL	×	-	1

操作数: AX 累加器高字节 (AH) 或累加器低字节 (AL) 寄存器

loc16 寻址方式

描述: “loc16”地址单元的内容与 AX 寄存器的内容按位“或”。结果保存到 AX 寄存器:

AX = AX OR [loc16];

标志与模式: N 测试装载后的 AX 是否为负。若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 测试装载后的 AX 是否为 0。若操作结果使得 AX=0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;对 VarA 和 VarB 的内容进行“或”操作, 将结果保存到 VarC
MOV    AL,@VarA          ;用 VarA 的内容装载 AL
OR     AL,@VarB          ;用 VarB 的内容与 AL 进行“或”操作
MOV    @VarC,AL          ;将结果保存到 VarC 中
```

OR loc16, AX

按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR loc16, AX	1001 100A LLLL LLLL	×	-	1

操作数: loc16 寻址方式

AX 累加器高字节 (AH) 或累加器低字节 (AL) 寄存器

描述: 用指定 AX 寄存器的值与“loc16”地址单元的内容按位“或”操作, 结果保存到“loc16”地址单元中:

$[loc16] = [loc16] \text{ OR } AX;$

本指令执行读取-修改-写入的操作。

标志与模式: N 测试[loc16]的值是否为负。若[loc16]的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 测试[loc16]的值是否为 0。若操作结果使得[loc16]= 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
;VarB 的内容和 VarA 的内容进行“或”操作, 将结果保存到 VarB
MOV    AL,@VarA          ;用 VarA 的内容装载 AL
OR     @VarB,AL          ;VarB = VarB “或” AL
```

OR IER, #16bit

中断使能寄存器 IER 按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR IER, #16bit	0111 0110 0010 0011 CCCC CCCC CCCC CCCC	×	-	2

操作数: IER 中断使能寄存器

#16bit 16 位立即常数

描述: 对 IER 寄存器的值与 16 位立即常数进行逐位“或”操作, 用来使能指定

的中断。结果保存到 IER 寄存器：

```
IER = IER OR #16bit;
```

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;仅使能 INT1 和 INT6。不修改其他中断的状态

```
OR IER, #0x0061 ;使能 INT1 和 INT6
```

OR IFR, #16bit

中断标志位寄存器 IFR 按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR IFR, #16bit	0111 0110 0010 0111 CCCC CCCC CCCC CCCC	×	—	2

操作数：IFR 中断标志位寄存器

#16bit 16 位立即常数

描述：对 IFR 寄存器的值与 16 位立即常数进行逐位“或”操作，用来使能指定的中断。结果保存到 IFR 寄存器：

```
IFR = IFR OR #16bit;
```

注意：中断硬件的优先级高于 CPU 的指令操作，这样可以防止中断标志位被硬件和指令同时修改。当外国中断扩展模块（PIE）使能时，指令不用于 1~12 号中断。

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

;仅触发 INT1 和 INT6 中断。不改变其他中断标志位的状态

```
OR IFR, #0x0061 ;触发 INT1 和 INT6 中断
```

OR loc16, #16bit

按位“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OR loc16, #16bit	0001 1010 LLLL LLLL CCCC CCCC CCCC CCCC	×	—	1

操作数：loc16 寻址方式

#16bit 16 位立即常数

描述：“loc16”地址单元的内容与 16 位立即常数逐位进行“或”操作。结果保存到“loc16”地址单元中：

```
[loc16] = [loc16] OR 16bit;
```

标志与模式：N 若[loc16]的第 15 位为 1，则负标志位 N 置位；否则清 0。

Z 若[loc16]的值为 0，则零标志位 Z 置位；否则清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;置 VarA 的第 4 和第 7 位
;VarA = VarA OR #(1 << 4 | 1 << 7)
OR @VarA, #(1 << 4 | 1 << 7)      ;置 VarA 的第 4 位和第 7 位
```

ORB AX, #8bit

8 位数按位求“或”

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ORB AX, #8bit	0101 000A CCCC CCCC	X	-	1

操作数： **AX** 累加器高字节 (AH) 或累加器低字节 (AL) 寄存器
#8bit 8 位立即常数

描述： 无符号 8 位立即常数进行零扩展后与指定 AX 寄存器的值逐位进行“或”操作。结果保存到 AX 寄存器：

AX = AX OR 0:8bit;

标志与模式： **N** 测试装载到 AX 寄存器的值是否为负。若 AX 的第 15 位为 1，则负标志位 N 置位；否则清 0。

Z 测试装载到 AX 寄存器的值是否为 0。若操作结果使得 AX = 0，则零标志位 Z 置位；否则清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
;置 VarA 的第 7 位并将结果保存到 VarB:
MOV AL, @VarA      ;用 VarA 的内容装载 AL
ORB AL, #0x80      ;用 0x0080 与 AL 的内容进行“或”操作
MOV @VarB, AL      ;将结果保存到 VarB 中
```

OUT *(PA), loc16

输出数据到端口

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
OUT *(PA), loc16	1011 1100 LLLL LLLL CCCC CCCC CCCC CCCC	1	-	4

操作数： ***(PA)** 立即寻址 I/O 空间
loc16 寻址方式

描述： 将“loc16”地址单元的 16 位数存入由*(PA)指向的 I/O 空间：

IOspace[0x0000PA] = [loc16];

I/O 空间限定在 64K 范围内 (0x0000~0xffff)。在操作过程中 I/O 片选信号 XISn 使能。I/O 地址出现在 XINTF 的低 16 位地址线 XA(15:0)，高位地址线为 0。数据出现在低 16 位数据线 XD(15:0)。

注意：UOUT 操作不受流水线保护。因此，若 UOUT 指令后紧跟了一条 IN 指令，则 IN 将在 UOUT 之前发生。要实现确定的操作顺序，使用有流水线保护的 OUT 指令。

注意：I/O 空间不是所有 C28x 芯片上都相同。

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

; IOREgA 地址 = 0x0300;
; IOREgB 地址 = 0x0301;
; IOREgC 地址 = 0x0302;
; IRegA = 0x0000;
; IOREgB = 0x0400;
; IOREgC = VarA;
; 若( IOREgC = 0x2000 )
;   IOREgC = 0x0000;
IORegA .set    0x0300      ; 定义 IOREgA 地址
IORegB .set    0x0301      ; 定义 IOREgB 地址
IORegC .set    0x0302      ; 定义 IOREgC 地址
MOV    @AL, #0             ; AL = 0
UOUT   *(IORegA), @AL      ; Iospace[IORegA] = AL
MOV    @AL, #0x0400        ; AL = 0x0400
UOUT   *(IORegB), @AL      ; Iospace[IORegB] = AL
OUT    *(IORegC), @VarA    ; Iospace[IORegC] = VarA
IN     @AL, *(IORegC)      ; AL = Iospace[IORegC]
CMP    @AL, #0x2000        ; 由(AL - 0x2000)的值置标志位
SB     $10, NEQ            ; 若不等则跳转
MOV    @AL, #0             ; AL = 0
UOUT   *(IORegC), @AL      ; Iospace[IORegC] = AL
$10:

```

POP ACC

栈顶弹出到 ACC

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP ACC	0001 1110 1011 1110	X	-	1

操作数：ACC 累加器

描述：将 SP 预先减掉 2。用 SP 指向的 32 位数装载 ACC：

```

SP - = 2;
ACC = [SP];

```

标志与模式：N 测试装载到 ACC 累加器的值是否为负。若 ACC 累加器的第 31 位为 1，则负标志位 N 置位；否则清 0。

Z 测试装载到 ACC 累加器的值是否为 0。若操作结果使得 ACC 为 0，则零标志位 Z 置位；否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP ARn:ARm

弹出栈顶到 16 位辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP AR1: AR0	0111 0110 0000 0111	×	-	1
POP AR3: AR2	0111 0110 0000 0101	×	-	1
POP AR5: AR4	0111 0110 0000 0110	×	-	1

操作数: ARn: ARm AR1: AR0 或 AR3: AR2 或 AR5: AR4 辅助寄存器

描述: AR1: AR0 或 AR3: AR2 或 AR5: AR4 将 SP 预先减 2。用 SP 和 SP+1 指向的值装载两个 16 位辅助寄存器 (ARn 和 ARm):

```
POP AR1: AR0
    SP - = 2;
    AR0 = [SP];
    AR1 = [SP+1];
    AR1H: AR0H = 不变;

POP AR3: AR2
    SP - = 2;
    AR2 = [SP];
    AR3 = [SP+1];
    AR3H: AR2H = 不变;

POP AR5: AR4
    SP - = 2;
    AR4 = [SP];
    AR5 = [SP+1];
    AR5H: AR4H = 不变;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP AR1H: AR0H

栈顶弹出到辅助寄存器高 16 位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP AR1H: AR0H	0000 0000 0000 0011	×	-	1

操作数: AR1H: AR0H 辅助寄存器 XAR1 和 XAR0 的高 16 位

描述: 将 SP 先减 2。用 SP 指向的值装载 AR0H, 用 SP+1 指向的值装载 AR1H。辅助寄存器 (AR0 和 AR1) 的低 16 位不变:

```
SP - = 2;
AR0H = [SP];
AR1H = [SP+1];
AR1: AR0 = 不变;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 中断和中断陷阱函数恢复断点

```

.
.
POP XT           ; 恢复 32 位 XT
POP XAR7         ; 恢复 32 位 XAR7
POP XAR6         ; 恢复 32 位 XAR6
POP XAR5         ; 恢复 32 位 XAR5
POP XAR4         ; 恢复 32 位 XAR4
POP XAR3         ; 恢复 32 位 XAR3
POP XAR2         ; 恢复 32 位 XAR2
POP AR1H: AR0H   ; 恢复 16 位 AR1H 和 16 位 AR0H
IRET

```

POP DBGIER

栈顶弹出到 DBGIER

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP DBGIER	0111 0110 0001 0010	×	-	5

操作数: DBGIER 调试中断使能寄存器

描述: 将 SP 先减 1。用 SP 指向的值装载 DBGIER:

```

SP - = 1;
DBGIER = [SP];

```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP DP

栈顶弹出到数据页指针

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP DP	0111 0110 0000 0011	×	-	1

操作数: DP 数据页指针寄存器

描述: SP 先减 1。SP 指向的内容装载到 DP:

```

SP - = 1;
DP = [SP];

```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP DP: ST1

栈顶弹出到 DP 和 ST1

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP DP: ST1	0111 0110 0000 0001	×	-	1

操作数: DP: ST1 数据页指针寄存器和状态寄存器 1
 描述: SP 先减 2。用 SP 指向的内容装载 ST1, 用 SP+1 指向的内容装载 DP:

SP- = 2 ;
 ST1 = [SP];
 DP = [SP+1];

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP IFR

栈顶弹出到 IFR

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP IFR	0000 0000 0000 0010	×	-	5

操作数: IFR 中断标志位寄存器

描述: SP 先减 1。用 SP 指向的内容装载 IFR:

SP- = 1;
 IFR = [SP];

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP loc16

弹出栈顶

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP loc16	0010 1010 LLLL LLLL	×	-	2

操作数: loc16 寻址方式

描述: SP 先减 1。用 SP 指向的 16 位数装载到 loc16 中:

SP- = 1;
 [loc16] = [SP];

标志与模式: N 若 (loc16=@AX), 则测试装载到 AX 是否为负数。AX 寄存器的第 15 位是标志位, 0 为正数, 1 为负数。若 AX 寄存器的操作为负数, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16=@AX), 则测试装载到 AX 是否为 0。若 AX 寄存器为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

POP @T ;SP 先减 1。用 SP 指向的位置的内容装载 XT(31: 15), TL 不变
 POP @AL ;SP 先减 1。用 SP 指向的位置的内容装载 AL, AH 不变
 POP @AR4 ;SP 先减 1。用 SP 指向的位置的内容装载 AR4, AR4H 不变

POP *XAR4+ + ;SP 先减 1。用 SP 指向的位置的内容装载到 XAR4 指向的
;16 位地址中, XAR4 增量

POP P

弹出栈顶到 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP P	0111 0110 0001 0001	×	-	1

操作数: P 乘积寄存器

描述: SP 先减 2。用 SP 指向的 32 位数装载 P:

$SP^- = 2;$

$P = [SP];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP RPC

从堆栈中弹出 RPC

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP RPC	0000 0000 0000 0111	×	-	3

操作数: RPC 返回程序计数寄存器

描述: SP 先减 2。用 SP 指向的内容装载到 RPC 中:

$SP^- = 2;$

$RPC = [SP];$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP ST0

弹出栈顶到 ST0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP ST0	0111 0110 0001 0011	×	-	1

操作数: ST0 状态寄存器 0

描述: SP 先减 1。用 SP 指向的内容装载到 ST0 中:

$SP^- = 1;$

$ST0 = [SP];$

标志与模式: C 每个列出的标志位由弹出的值代替

N

V

Z

TC

SXM

OVC
PM

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP ST1

栈顶弹出到 ST1

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP ST1	0111 0110 0000 0000	×	-	5

操作数: ST1 状态寄存器 1

描述: SP 先减 1。用 SP 指向的值装载 ST1 的内容:

SP- = 1;
ST1 = [SP];

标志与模式: DBGM 每个列出的标志位由弹出的值代替

INTM
VMAP
SPA
PAGE0
AMODE
ARP
EAL-
LOW
OBJ-
MODE
XF

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

POP T: ST0

栈顶弹出到 T 和 ST0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP T: ST0	0111 0110 0001 0101	×	-	1

操作数: T: ST0 被乘数寄存器的高 16 位和状态寄存器 0

描述: SP 先减 2。用 SP 指向的内容装载 ST0 和用 SP+1 指向的内容装载 T。
XT 寄存器 TL 的低 16 位不变:

SP - = 2;
ST0 = [SP];
T = [SP+1];
TL = 不变;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器

RPTC, 并且只执行一次。

POP XARn

栈顶弹出到 32 位辅助寄存器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP XAR0	0011 1010 1011 1110	1	-	1
POP XAR1	1011 0010 1011 1110	1	-	1
POP XAR2	1010 1010 1011 1110	1	-	1
POP XAR3	1010 0010 1011 1110	1	-	1
POP XAR4	1010 1000 1011 1110	1	-	1
POP XAR5	1010 0000 1011 1110	1	-	1
POP XAR6	1100 0010 1011 1110	×	-	1
POP XAR7	1100 0011 1011 1110	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器

描述: SP 先减 2。用 SP 指向的 32 位数装载到 XARn 中:

```
SP - = 2;
XARn = [SP];
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 中断或中断陷阱函数恢复断点
.
.
.
POP XT          ; 恢复 32 位 XT
POP XAR7        ; 恢复 32 位 XAR7
POP XAR6        ; 恢复 32 位 XAR6
POP XAR5        ; 恢复 32 位 XAR5
POP XAR4        ; 恢复 32 位 XAR4
POP XAR3        ; 恢复 32 位 XAR3
POP XAR2        ; 恢复 32 位 XAR2
POP AR1H: AR0H  ; 恢复 16 位 AR1H 和 16 位 AR0H
IRET
```

POP XT

栈顶弹出到 XT

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
POP XT	1000 0111 1011 1110	×	-	1

操作数: XT 被乘数寄存器

描述: SP 先减 2。用 SP 指向的 32 位数装载到 XT 中:

```
SP - = 2;
XT = [SP];
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PREAD loc16,*XAR7

读程序存储器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PREAD loc16,*XAR7	0010 0100 LLLL LLLL	×	Y	N+2

操作数: loc16 寻址方式

*XAR7 使用辅助寄存器 XAR7 间接寻址程序存储器, 可以访问全部 4M×16 程序空间范围 (0x000000~0x3FFFFFF)

描述: 将 “*XAR7” 指向的程序存储器地址单元的 16 位数装载到 “loc16” 寻址的数据存储器地址单元中:

```
[loc16] = prog[*XAR7];
```

对于 C28x 器件, 存储器块映射含程序和数据空间 (同一物理存储器), 因此 “*XAR7” 可以寻址访问位于程序空间范围的数据空间。

使用某些组合寻址方式, 可能会使引用发生冲突。在这种情况下, C28x 会将 XAR7 的优先级给定。例如:

```
PREAD *- -XAR7, *XAR7, ; *- -XAR7 约定的高优先级
```

```
PREAD *XAR7+ +, *XAR7, ; * XAR7+ + 约定的高优先级
```

标志与模式: N 若 (loc16 = @AX) 并且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 (loc16 = @AX) 并且 AX 的第 15 位为 0, 则零标志位 Z 置位; 否则 Z 清 0。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则将执行 N+1 次。当每次重复时, “*XAR7” 程序存储器地址复制到一个内部影子寄存器, 地址在每次重复时快速增 1。

例:

```
; Copy the contents of Array1 to Array2;
; int16 Array1[N]
; // 在程序空间定位
; int16 Array [N]
; // 在数据空间定位
; for(i = 0; i N; i+ +)
; Array2[i] = Array1[i];
MOVL XAR7,#Array1 ;XAR7 指向 Array1
MOVL XAR2,#Array2 ;XAR2 指向 Array2
RPT # (N-1) ;重复下一条指令 N 次
||PREAD *XAR2+ +, *XAR7 ;Array2[i] = Array1[i], i+ +
```

PUSH ACC

将累加器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH ACC	0001 1110 1011 1101	×	-	2

操作数: ACC 累加器

描述: 把 ACC 的 32 位数压入堆栈中, SP 增加 2:

```
[SP] = ACC;
SP + = 2;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
MOVL  XAR4, #VarA      ; 用 VarA 的 22 位地址初始化 XAR4 指针
MOVL  ACC,  *+XAR4[0]   ; 装载 VarA 的 32 位数到 ACC
PUSH  ACC               ; 把 ACC 压入堆栈, SP 增加 2
```

PUSH ARn:ARm

将 16 位辅助寄存器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH AR1: AR0	0111 0110 0000 1101	×	-	1
PUSH AR3: AR2	0111 0110 0000 1111	×	-	1
PUSH AR5: AR4	0111 0110 0000 1100	×	-	1

操作数: ARn: ARm AR1: AR0 或 AR3: AR2 或 AR5: AR4 辅助寄存器

描述: 把 2 个 16 位辅助寄存器 (ARn 和 ARm) 的内容压入堆栈。SP 增加 2:

```
PUSH AR1: AR0
    [SP] = AR0;
    [SP+1] = AR1;
    SP + = 2;
PUSH AR3: AR2
    [SP] = AR2;
    [SP+1] = AR3;
    SP + = 2;
PUSH AR5: AR4
    [SP] = AR4;
    [SP+1] = AR5;
    SP + = 2;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH AR1H: AR0H

将 AR1H 和 AR0H 寄存器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH AR1H: AR0H	0000 0000 0000 0101	×	-	1

操作数: AR1H: AR0H XAR1 和 XAR0 辅助寄存器的高 16 位

描述: 把 AR1H 的内容和 AR0H 的内容压入堆栈。SP 增加 2:

```
[SP] = AR0H;
[SP+1] = AR1H;
SP+ = 2;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
IntX:                ;中断或中断陷阱函数的断点保存代码
    PUSH  AR1H: AR0H  ;保存 16 位的 AR1H 和 16 位的 AR0H
    PUSH  XAR2        ;保存 32 位的 XAR2
    PUSH  XAR3        ;保存 32 位的 XAR3
    PUSH  XAR4        ;保存 32 位的 XAR4
    PUSH  XAR5        ;保存 32 位的 XAR5
    PUSH  XAR6        ;保存 32 位的 XAR6
    PUSH  XAR7        ;保存 32 位的 XAR7
    PUSH  XT          ;保存 32 位的 XT
    .
    .
    .
```

PUSH DBGIER

将 DBGIER 寄存器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH DBGIER	0111 0110 0000 1110	×	-	1

操作数: DBGIER 调试中断使能寄存器

描述: 把 DBGIER 的 16 位数压入堆栈。SP 增加 1

```
[SP] = DBGIER;
SP + = 1;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH DP

将 DP 寄存器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH DP	0111 0110 0000 1011	×	-	1

操作数: DP 数据页寄存器

描述: 把 DP 的 16 位数压入堆栈。SP 增加 1

```
[SP] = DP;
SP + = 1;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器

RPTC, 并且只执行一次。

PUSH DP: ST1

将 DP 和 ST1 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH DP: ST1	0111 0110 0000 1001	×	-	1

操作数: DP: ST1 数据页寄存器和状态寄存器 1

描述: 把 DP 的 16 位数和 ST1 的 16 位数压入堆栈。SP 增加 2:

```
[SP] = ST1;
[SP+1] = DP;
SP+ = 2;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH IFR

将 IFR 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH IFR	0111 0110 0000 1010	×	-	1

操作数: IFR 中断标志位寄存器

描述: 把 IFR 的 16 位数压入堆栈。SP 增加 1

```
[SP] = IFR;
SP + = 1;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH loc16

将 16 位数压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH loc16	0010 0010 LLLL LLLL	×	-	2

操作数: loc16 寻址方式

描述: 把“loc16”操作数指向的 16 位数压入堆栈。SP 增加 1

```
[SP] = [loc16];
SP + = 1;
```

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
PUSH @T      ;把 XT(31:15)的内容压入堆栈。SP 增加 1
PUSH @AL      ;把 AL 的内容压入堆栈。SP 增加 1
```


PUSH @AR4 ;把 XAR4 的低 16 位压入堆栈。SP 增加 1
 PUSH *XAR4+ + ;把 XAR4 指向的值压入堆栈, SP 和 XAR4 增加 1

PUSH P

将 P 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH P	0111 0110 0001 1101	×	-	1

操作数: P 乘积寄存器

描述: 把 P 的 32 位数压入堆栈。SP 增加 2:

[SP] = P;
 SP + = 2;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

MOVL XAR5, #VarA ;用 VarA 的 22 位地址初始化 XAR5 指针
 MOVL P, *+XAR5[0] ;装载 VarA 的 32 位数到 P
 PUSH P ;把 32 位 P 压入堆栈。SP 增加 2

PUSH RPC

将 RPC 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH RPC	0000 0000 0000 0010	×	-	1

操作数: RPC 返回程序计数寄存器

描述: 把 RPC 寄存器的内容压入堆栈。SP 增加 2:

[SP] = RPC;
 SP + = 2;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH ST0

将 ST0 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH ST0	0111 0110 0001 1000	×	-	1

操作数: ST0 状态寄存器 0

描述: 把 ST0 的 16 位数压入堆栈。SP 增加 1

[SP] = ST0;
 SP+ = 1;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器

RPTC, 并且只执行一次。

PUSH ST1

将 ST1 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH ST1	0111 0110 0000 1000	×	-	1

操作数: ST1 状态寄存器 1

描述: 把 ST1 的 16 位数压入堆栈。SP 增加 1

[SP] = ST1;

SP + = 1;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH T: ST0

将 T 和 ST0 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH T: ST0	0111 0110 0001 1001	×	-	1

操作数: T: ST0 被乘数寄存器的高 16 位和状态寄存器 0

描述: 将 ST0 的 16 位数和 T 的 16 位数压入堆栈。SP 增加 2:

[SP] = ST0;

[SP+1] = T;

SP + = 2 ;

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

PUSH XARn

将 32 位辅助寄存器压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH XAR0	0011 1010 1011 1101	1	-	1
PUSH XAR1	1011 0010 1011 1101	1	-	1
PUSH XAR2	1010 1010 1011 1101	1	-	1
PUSH XAR3	1010 0010 1011 1101	1	-	1
PUSH XAR4	1010 1000 1011 1101	1	-	1
PUSH XAR5	1010 0000 1011 1101	1	-	1
PUSH XAR6	1100 0010 1011 1101	×	-	1
PUSH XAR7	1100 0011 1011 1101	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器

描述: 把 XARn 的 32 位数压入堆栈。SP 增加 2:

[SP] = XARn;

SP+ = 2;

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
IntX:                                ;中断或中断陷阱函数的断点保存代码
    PUSH  AR1H: AR0H                 ;保存 16 位的 AR1H 和 16 位的 AR0H
    PUSH  XAR2                       ;保存 32 位的 XAR2
    PUSH  XAR3                       ;保存 32 位的 XAR3
    PUSH  XAR4                       ;保存 32 位的 XAR4
    PUSH  XAR5                       ;保存 32 位的 XAR5
    PUSH  XAR6                       ;保存 32 位的 XAR6
    PUSH  XAR7                       ;保存 32 位的 XAR7
    PUSH  XT                         ;保存 32 位的 XT
    .
    .
    .
```

PUSH XT

将 XT 压入堆栈

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PUSH XT	0010 0010 LLLL LLLL	×	—	1

操作数：XT 被乘数寄存器

描述：把 XT 的 32 位数压入堆栈。SP 增加 2：

```
[SP] = XT;
SP + 2 = 2;
```

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
MOVL  XAR1, #VarA                   ;用 VarA 的 22 位地址初始化 XAR1 指针
MOVL  XT, *XAR5[0]                  ;装载 VarA 的 32 位数到 XT
PUSH  XT                           ;把 32 位的 XT 压入堆栈。SP 增加 2
```

PWRITE *XAR7,loc16

写程序存储器

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
PWRITE *XAR7,loc16	0010 0110 LLLL LLLL	×	Y	N+5

操作数：*XAR7 使用辅助寄存器 XAR7 的间接程序存储器寻址方式，可以访问全部 4M×16 程序空间范围 (0x000000 ~ 0x3FFFFFF)

loc16 寻址方式

描述：用“loc16”地址单元内容装载到“*XAR7”指向的程序存储器地址单元中：

```
Prog[*XAR7] = [loc16];
```

对于 C28x 芯片，存储器块映射含程序和数据空间（同一物理存储器），因此“*XAR7”寻址方式可以用来访问程序地址空间范围的数据空间。使用某些组合寻址方式，可能会发生冲突。在这种情况下，C28x 会给定优先级。例如：

```
PWRITE *XAR7,*--XAR7 ;*--XAR7 约定的高优先级
PWRITE *XAR7,*XAR7++ ;*XAR7++ 约定的高优先级
```

标志与模式：无

重复性：本指令可以重复。若指令跟在 RPT 指令之后，它将执行 N+1 次。当每次重复时，“*XAR7”程序存储器地址复制到一个内部影子寄存器，在每次重复时地址增加 1。

例：

```
; 复制 Array1 的内容到 Array2:
; int16 Array1[N];           // 在数据空间
; int16 Array2[N];           // 在程序空间
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i];
MOVL XAR2,#Array1             ; XAR2 指针指向 Array1
MOVL XAR7,#Array2             ; XAR7 指针指向 Array2
RPT  #(N-1)                   ; 重复下一条指令 N 次
||PWRITE *XAR7,*XAR2++ ; Array2[i] = Array1[i], i++
```

QMACL P,loc32,*XAR7/++

32 位×32 位有符号数乘加（高字）

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMACL P,loc32,*XAR7	0101 0110 0100 1111 1100 0111 LLLL LLLL	1	Y	N+2
QMACL P,loc32,*XAR7++	0101 0110 0100 1111 1000 0111 LLLL LLLL	1	Y	N+2

操作数： P 乘积寄存器
loc32 寻址方式

注意：当指令重复时，@ACC 寻址方式不能使用。若使用将不产生（非法指令陷阱汇编程序仅会标记一个错误）。

XAR7/++ 使用辅助寄存器 XAR7 的间接程序存储器寻址，可以寻址全部 4M×16 程序空间范围（0x000000 to 0x3FFFFFF）。

描述：32 位×32 位有符号乘和累加。首先，将 P 按乘积移位方式（PM）指定的移位位数移位后，加先前乘积（保存在 P 寄存器）到 ACC 累加器。然后，用 XAR7 寄存器指向的程序存储器地址单元的 32 位有符号数乘以“loc32”地址单元的 32 位有符号数，64 位结果的高 32 位保存到 P 寄存器。若指定，XAR7 寄存器增加 2：

```
ACC = ACC + P << PM;
P = (signed T × signed Prog[*XAR7 or *XAR7+ +]) >> 32;
```

对于 C28x 芯片，存储器块映射含程序和数据空间（同一物理存储器），因此“*XAR7”寻址方式可以用来访问程序地址空间范围的数据空间。使用某些组合寻址方式，可能会发生冲突。在这种情况下，C28x 会给定优先级。例如：

```
QMACL P, *- -XAR7, *XAR7+ + ; - -XAR7 约定的高优先级
QMACL P, *XAR7+ +, *XAR7 ; *XAR7+ + 约定的高优先级
QMACL P, *XAR7, *XAR7+ + ; *XAR7+ + 约定的高优先级
```

- 标志与模式：**
- Z** 若 ACC 值为 0，则零标志位 Z 置位；否则 Z 清 0。
 - N** 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。
 - C** 若产生进位，则进位标志位 C 置位；否则进位标志位 C 清 0。
 - V** 若产生溢出，则溢出标志位 V 置位；否则不影响 V。
 - OVC** 若溢出模式禁止，操作产生正溢出，则计算器值增加；若操作产生负溢出，则计算器值减少。
 - OVM** 若溢出模式位置位，操作发生溢出，则 ACC 的值为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。
 - PM** PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正（逻辑左移操作），则最低位填 0。若乘积移位位数为负（算术右移操作），则高位进行符号扩展。
- 重复性：** 本指令可以重复。若指令跟在 RPT 指令之后，则将执行 N+1 次。Z、N、C 和 OVC 的状态为最终结果。若中间 ACC 发生溢出，则 V 标志位置位。

例：

```
; 使用 32 位乘积计算乘积的 sum 并保留高位结果
; int32 X[N]; // 数据信息
; int32 C[N]; // 系数信息(定位在 4M 低端)
; int32 sum = 0;
; for(i = 0; i < N; i+ +)
;   sum = sum + ((X[i] × C[i]) >> 32) >> 5;
MOVL XAR2, #X ; XAR2 指针指向 X
MOVL XAR7, #C ; XAR7 指针指向 C
SPM -5 ; 设置乘积右移 5 位
ZAPA ; ACC, P, OVC 清 0
RPT #(N-1) ; 重复下一条指令 N 次
||QMACL P, *XAR2+ +, *XAR7+ + ; ACC = ACC + P >> 5,
; P = (X[i] × C[i]) >> 32, i+ +
; 执行最终的计算
ADDL ACC, P << PM ; 保存最终的结果到 sum
MOVL @sum, ACC
```

QMPYAL P,XT,loc32

有符号数的 32 位×32 位乘法并加上先前乘积 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMPYAL P,XT,loc32	0101 0110 0100 0110 0000 0000 LLLL LLLL	1	~	1

- 操作数: P 乘积寄存器
XT 被乘数寄存器
loc32 寻址方式
- 描述: 将 P 按乘积移位方式 (PM) 指定的移位位数移位后, 加先前有符号乘积 (保存在 P 寄存器) 到 ACC 累加器。用 “loc32” 寻址方式指向的有符号 32 位数乘 XT 的有符号 32 位数, 保存 64 位结果的高 32 位到 P 寄存器:
- $$ACC = ACC + P \ll PM;$$
- $$P = (\text{signed } T \times \text{signed } [\text{loc32}]) \gg 32;$$
- 标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
C 若产生进位, 则进位标志位 C 置位; 否则进位标志位 C 清 0。
V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
OVC 若禁止溢出模式, 且操作产生正溢出, 则计算器值增加。若操作产生负溢出, 则计算器值减少。
OVM 若溢出模式位置位; 且操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
PM PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (算术右移操作), 则高位进行符号扩展。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算有符号数的结果
; Y32 = (X0×C0 + X1×C1 + X2×C2) >> (32 + 2)
SPM    -2                ; 设置乘积右移 2 位
ZAPA                    ; ACC, P, OVC 清 0
MOVL    XT,@X0           ; XT = X0
QMPYAL  P,XT,@C0         ; P = (X0×C0) 的高 32 位
MOVL    XT,@X1           ; XT = X0
QMPYAL  P,XT,@C1         ; ACC = ACC + P >> 2,
                        ; P = (X1×C1) 的高 32 位
MOVL    XT,@X2           ; XT = X0
QMPYAL  P,XT,@C2         ; ACC = ACC + P >> 2,
                        ; (X2×C2) 的高 32 位
ADDL    ACC,P << PM      ; ACC = ACC + P >> 2
MOVL    @Y32,ACC         ; 结果保存到 Y32
```

QMPYL P,XT,loc32

32 位有符号数相乘并加上先前乘积 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMPYL P,XT,loc32	0101 0110 0110 0111 0000 0000 LLLL LLLL	1	-	1

- 操作数: P 乘积寄存器
XT 被乘数寄存器
loc32 寻址方式
- 描述: 用“loc32”地址单元的有符号 32 位数乘以 XT 寄存器的有符号 32 位数, 64 位结果 (Q30 格式) 的高 32 位保存到 P 寄存器:
- $$P = (\text{signed } XT \times \text{signed } [\text{loc32}]) \gg 32;$$
- 标志与模式: 无
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算有符号数结果 Y64 = M32×X32 + B64
MOVL  XT,@M32           ; XT = M32
IMPYL  P,XT,@X32        ; P = (M32×X32) 的低 32 位
MOVL  ACC,@B64+2        ; ACC = B64 的高 32 位
ADDUL  P,@B64+0         ; P = P + B64 的低 32 位
MOVL  @Y64+0,P          ; 保存结果的低 32 位到 Y64
QMPYL  P,XT,@X32        ; P = (M32×X32) 的高 32 位
ADDCL  ACC,@P            ; ACC = ACC + P + 进位
MOVL  @Y64+2,ACC        ; 保存结果的高 32 位到 Y64 中
```

QMPYL ACC,XT,loc32

32×32 有符号数 (高 32 位)

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMPYL ACC,XT,loc32	0101 0110 0110 0011 0000 0000 LLLL LLLL	1	-	2

- 操作数: P 乘积寄存器
XT 被乘数寄存器
loc32 寻址方式
- 描述: 用“loc32”地址单元的有符号 32 位数乘 XT 寄存器的有符号 32 位数, 64 位结果 (Q30 格式) 的高 32 位保存到 ACC 累加器:
- $$ACC = (\text{signed } XT \times \text{signed } [\text{loc32}]) \gg 32;$$
- 标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算有符号结果 Y64 = M32×X32
MOVL  XT,@M32           ; XT = M32
IMPYL  P,XT,@X32        ; P = (M32×X32) 的低 32 位
QMPYL  ACC,XT,@X32      ; ACC = (M32×X32) 的高 32 位
MOVL  @Y64+0,P          ; 结果保存到 Y64
MOVL  @Y64+2,ACC
```

QMPYSL P,XT,loc32**32×32 有符号数相乘（高 32 位）并减先前乘积**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMPYSL P,XT,loc32	0101 0110 0100 0101 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

描述: 有符号 32 位×32 位相乘并减先前的乘积。将 P 按乘积移位方式 (PM) 指定的移位位数移位后, 从 ACC 累加器中减去先前有符号数的乘积 (保存在 P 寄存器)。用有符号 32 位常数乘 XT 寄存器的有符号 32 位数, 64 位结果 (Q30 格式) 的高 32 位保存到 P 寄存器:

$$ACC = ACC - P \ll PM;$$

$$P = (\text{signed } T \times \text{signed } [loc32]) \gg 32;$$

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 OVC 若禁止溢出模式, 且操作产生正溢出, 则计算器值增加。若操作产生负溢出, 则计算器值减少。
 OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 PM PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (算术右移操作), 则高位进行符号扩展。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算有符号结果
; Y32 = -(X0×C0 + X1×C1 + X2×C2) >> (32 + 2)
SPM    - 2          ; 设置乘积右移 2 位
ZAPA    ; ACC, P, OVC 清 0
MOVL   XT,@X0       ; XT = X0
QMPYL   P,XT,@C0     ; P = (X0×C0) 的高 32 位
MOVL   XT,@X1       ; XT = X0
QMPYSL  P,XT,@C1     ; ACC = ACC - P >> 2,
                    ; P = (X1×C1) 的高 32 位
MOVL   XT,@X2       ; XT = X0
QMPYSL  P,XT,@C2     ; ACC = ACC - P >> 2,
                    ; P = (X2×C2) 的高 32 位
SUBL   ACC,P << PM   ; ACC = ACC - P >> 2
MOVL   @Y32,ACC      ; 结果保存到 Y32
```

QMPYUL P,XT,loc32**32×32 无符号数（高 32 位）乘法**

语法选项	操作码	OBJ 模式	RPT	CYC
QMPYUL P,XT,loc32	0101 0110 0100 0111 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

描述: 用“loc32”寻址地址单元的无符号 32 位数乘以 XT 寄存器的无符号 32 位数，64 位结果的高 32 位保存到 P 寄存器：

$$P = (\text{unsigned } XT \times \text{unsigned } [\text{loc32}]) \gg 32;$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```
; 计算无符号结果 Y64 = M32×X32 + B64
MOVL  XT,@M32      ; XT = M32
IMPYL P,XT,@X32    ; P = (M32×X32) 的低 32 位
MOVL  ACC,@B64+2   ; ACC = B64 的高 32 位
ADDUL P,@B64+0     ; P = P + B64 的低 32 位
MOVL  @Y64+0,P     ; 保存结果的低 32 位到 Y64
QMPYUL P,XT,@X32   ; P = (M32×X32) 的高 32 位
ADDCL ACC,@P       ; ACC = ACC + P + 进位
MOVL  @Y64+2,ACC   ; 保存结果的高 32 位到 Y64
```

QMPYXUL P,XT,loc32**32 位无符号数×32 位有符号数乘法**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
QMPYXUL P,XT,loc32	0101 0110 0100 0010 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器
 XT 被乘数寄存器
 loc32 寻址方式

描述: 用“loc32”寻址地址单元的无符号 32 位数乘以 XT 寄存器的有符号 32 位数，64 位结果的高 32 位保存到 P 寄存器：

$$P = (\text{signed } XT \times \text{unsigned } [\text{loc32}]) \gg 32;$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```
; 计算有符号结果 Y64 = (M64×X64) >> 64 + B64
```



```

; Y64 = Y1:Y0, M64 = M1:M0, X64 = X1:X0, B64 = B1:B0
MOVL    XT,@X1      ; XT = X1
QMPYXUL P,XT,@M0     ; P = (无符号 M0 × 有符号 X1)的高 32 位
MOV      @T,#32      ; T = 32
LSL64   ACC:P,T      ; ACC: P = ACC:P << T
ASR64   ACC:P,T      ; ACC: P = ACC:P >> T
MOVL    @XAR4,P      ; XAR5:XAR4 = ACC: P
MOVL    @XAR5,ACC
MOVL    XT,@M1       ; XT = M1
QMPYXUL P,XT,@X0     ; P = (有符号 M1 × 无符号 X0)的高 32 位
MOV      @T,#32      ; T = 32
LSL64   ACC:P,T      ; ACC:P = ACC:P << T
ASR64   ACC:P,T      ; ACC:P = ACC:P >> T
MOVL    @XAR6,P      ; XAR7:XAR6 = ACC:P
MOVL    @XAR7,ACC
IMPYL   P,XT,@X1     ; P = (有符号 M1 × 有符号 X1)的低 32 位
QMPYL   ACC,XT,@X1   ; ACC = (有符号 M1 × 有符号 X1)的高 32 位
ADDUL   P,@XAR4      ; ACC:P = ACC:P + XAR5: XAR4
ADDCL   ACC,@XAR5
ADDUL   P,@XAR6      ; ACC:P = ACC:P + XAR7:XAR6
ADDCL   ACC,@XAR7
ADDUL   P,@B0        ; ACC:P = ACC:P + B64
ADDCL   ACC,@B1
MOVL    @Y0,P        ; 结果保存到 Y64
MOVL    @Y1,ACC

```

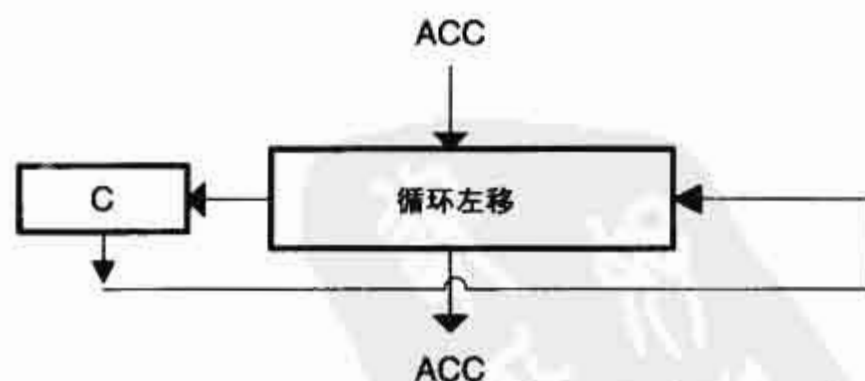
ROL ACC

累加器循环左移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ROL ACC	1111 1111 0101 0011	×	Y	N+1

操作数: ACC 累加器

描述: ACC 累加器的值循环左移一位, 最低位用进位标志位 C 填充, 用移出的位装载进位标志位 C。



标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

C ACC 累加器的第 31 位送到 C。C 的值送到 ACC 的第 0 位。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则 ROL 指令执行 N+1 次。Z、N、C 的状态为最终结果。

例:

```

; 左旋转 VarA 的内容 5 次:
MOVL ACC,@VarA ; ACC = VarA
RPT #4          ; 重复下一条指令 5 次
||ROL ACC       ; 左旋转 ACC
MOVL @VarA,ACC  ; 结果保存到 VarA

```

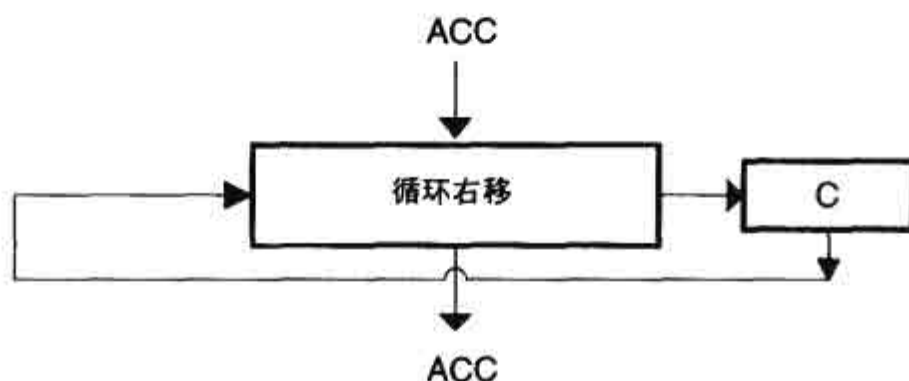
ROR ACC

累加器循环右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
ROR ACC	1111 1111 0101 0010	×	Y	N+1

操作数: ACC 累加器

描述: ACC 累加器的值循环右移一位, 用进位标志位 C 填入第 31 位, 用移出的位装载进位标志位 C:



标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

C ACC 累加器的第 0 位送到 C。C 的值送到 ACC 的第 31 位。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则 ROR 指令执行 N+1 次。Z、N、C 的状态会为最终的结果。

例:

```

; 右旋转 VarA 的内容 5 次:
MOVL ACC,@VarA ; ACC = VarA
RPT #4          ; 重复下一条指令 5 次
||ROR ACC       ; 右旋转 ACC
MOVL @VarA,ACC  ; 结果保存到 VarA

```

RPT #8bit/loc16

重复下一条指令

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
RPT #8bit	1111 0110 CCCC CCCC	×	-	1
RPT loc16	1111 0111 LLLL LLLL	×	-	4

操作数: #8bit 8 位立即数 (0~255 范围)

loc16 寻址方式

描述: 重复下一条指令。用 N 装载内部重复计数器 RPTC, 它既可以是 8 位常数也可以是“loc16”地址单元的值。在 RPT 后的指令将执行 N+1 次。因为 RPTC 不能中途保存, 所以重复运行时被认为是多周期指令并且不

能被中断。

语法描述: 重复指令前的平行线 (||) 用以提醒指令重复并且不能被中断。
当在 C 语言中嵌入汇编程序时, 使用格式:

```
asm(|| RPT #8bit/ loc16 || "instruction");
```

不是所有的指令都可以重复。若 RPT 指令后的指令不能重复, RPTC 重复计数器复位为 0, 指令只执行一次。28x 汇编程序语言工具检查这种条件并发出警告。

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 把 VarA 指定个数的数组元素从 Array1 复制到 Array2;
; int16 Array1[N];                // 定位在程序空间的高 64K
; int16 Array2[N];                // 定位在数据空间
; for(i = 0; i < VarA; i++)
; Array2[i] = Array1[i];
MOVL      XAR2, #Array2           ; XAR2 指针指向 Array2
RPT       @VarA                   ; 重复下一条指令 [VarA] + 1 次
||XPREAD  *XAR2+, *(Array1)      ; Array2[i] = Array1[i], i++
```

SAT ACC

使累加器为饱和值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SAT ACC	1111 1111 0101 0111	×	-	1

操作数: ACC 累加器

描述: 根据 6 位溢出计数器 (OVC) 的值使 ACC 累加器为饱和值:

```
if( OVC > 0 )
    ACC = 0x7FFF FFFF;
V = 1;
if( OVC < 0 )
    ACC = 0x8000 0000;
V = 1;
if( OVC = 0 )
    ACC = 不变;
OVC = 0;
```

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

C C 清 0。

V 若操作开始时 (OVC != 0), 则溢出标志位 V 置位; 否则 V 清 0。

OVC 若 (OVC > 0), 则 ACC 为最大正数。

若 (OVC < 0), 则 ACC 为最小负数。

若 (OVC=0)，则 ACC 不改变。

操作之后，OVC 清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

；将 VarA，VarB 和 VarC 相加，使其为饱和值，结果保存在 VarD：

```
ZAP  OVC          ; 溢出计数器清 0
MOVL ACC,@VarA    ; 用 VarA 的内容装载 ACC
ADDL ACC,@VarB    ; 把 VarB 的内容加到 ACC
ADDL ACC,@VarC    ; 把 VarC 的内容加到 ACC
SAT  ACC          ; 根据 OVC 值使 ACC 为饱和值
MOVL @VarD,ACC    ; 结果保存到 VarD
```

SAT64 ACC:P

使 64 位的 ACC:P 值为饱和值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SAT64 ACC:P	0101 0110 0011 1110	1	-	1

操作数： ACC:P 累加器 (ACC) 和乘积寄存器 (P)

描述： 根据 6 位溢出计数器 (OVC) 的值使 ACC:P 寄存器 64 位数为饱和值：

```
if(OVC > 0)
    ACC:P = 0x7FFF FFFF FFFF FFFF;
V = 1;
if(OVC < 0)
    ACC:P = 0x8000 0000 0000 0000;
V = 1;
if(OVC = 0)
    ACC:P = 不变;
OVC = 0;
```

标志与模式： N 若 ACC 的第 31 位为 1，则 ACC:P 为负数，则负标志位 N 置位；否则清 0。

Z 若 ACC:P 组合 64 位数为 0，则零标志位 Z 置位；否则 Z 清 0。

C 清 0。

V 操作开始时，若 (OVC = 0)，则 V 清 0；否则溢出标志位 V 置位。

OVC 若 (OVC=0)，不发生使其为饱和值的情况：ACC:P 不改变。
若 (OVC>0)，则 ACC:P 为最大正数：

ACC:P = 0x7FFF FFFF FFFF FFFF

若 (OVC<0)，则 ACC:P 为最小负数：

ACC = 0x8000 0000 or ACC:P = 0x8000 0000 0000 0000

操作结束后，OVC 清 0。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```

; 将 64 位的 VarA, VarB 和 VarC 相加, 置为保和值保存到 VarD:
ZAP    OVC                ; 溢出计数器清 0
MOVL   P,@VarA+0          ; 用 VarA 的低 32 位装载 P
ADDUL  P,@VarB+0          ; VarB 的无符号低 32 位加到 P
ADDUL  P,@VarC+0          ; VarC 的无符号低 32 位加到 P
MOVU   @AL,OVC            ; 溢出值存到 ACC 并加 64 位变量的高位
MOVB   AH,#0
ZAP    OVC                ; 溢出计数器清 0
ADDL   ACC,@VarA+2        ; VarA 的高 32 位带进位加到 ACC
ADDL   ACC,@VarB+2        ; VarB 的高 32 位带进位加到 ACC
ADDL   ACC,@VarC+2        ; VarC 的高 32 位带进位加到 ACC
SAT64  ACC:P              ; 根据 OVC 的值使 ACC:P 为饱和值
MOVL   @VarD+0,P          ; 保存结果的低 32 位到 VarD
MOVL   @VarD+2,ACC        ; 保存结果的高 32 位到 VarD

```

SB 8bitOffset,COND

条件短跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SB 8bitOffset, COND	0110 COND CCCC CCCC	×	-	7/4

操作数: 8bitOffset 8 位有符号立即数偏移量 (-128~+127 范围)
 COND 条件代码

COND	格 式	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	大于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于, 进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述: 条件短跳转指令。若条件为真, 加有符号 8 位常数到当前 PC 值并跳转;
 否则继续执行而不跳转:

```

If (COND = 真)    PC = PC + 有符号 8 位偏移量;
If (COND = 假)    PC = PC + 1;

```

标志与模式: V 若用条件测试 V 标志位, 则 V 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

SBBU ACC,loc16

减无符号数再减进位位的反码

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SBBU ACC,loc16	0001 1101 LLLL LLLL	×	-	1

操作数: ACC 累加器

loc16 寻址方式

描述: ACC 累加器减去 0 扩展后的“loc16”地址单元的 16 位数, 再减去进位位的反码:

$$ACC = ACC - 0:[loc16] - \sim C;$$

标志与模式: Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 指令执行前的进位位状态包含在指令中。若减产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVC 若禁止溢出模式 (OVM = 0), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作的影响。

OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;用 16 位分部分减法作 3 个 32 位无符号变量减:

```

MOVU ACC,@VarA_low      ; AH = 0, AL = VarA 低位
ADD  ACC,@VarA_high << 16 ; AH = VarA 高位, AL = VarA 低位
SUBU  ACC,@VarB_low      ; ACC = ACC - 0: VarB 低位
SUB  ACC,@VarB_high << 16 ; ACC = ACC - VarB 高位 << 16
SBBU  ACC,@VarC_low      ; ACC = ACC - VarC 低位 - 进位位取反
SUB  ACC,@VarC_high << 16 ; ACC = ACC - VarC 高位 << 16

```

SBF 8bitOffset, EQ/NEQ/TC/NTC

快速条件短跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SBF 8bitOffset,EQ	1110 1100 CCCC CCCC	1	-	4/4
SBF 8bitOffset,NEQ	1110 1101 CCCC CCCC	1	-	4/4
SBF 8bitOffset,TC	1110 1110 CCCC CCCC	1	-	4/4
SBF 8bitOffset,NTC	1110 1111 CCCC CCCC	1	-	4/4

操作数: 8bitOffset 8 位有符号立即数偏移量 (-128~+127 范围)

- 语法说明 标志位测试
- NEQ 不等于 Z = 0
- EQ 等于 Z = 1
- NTC 测试位不置位 TC = 0
- TC 测试位置位 TC = 1
- 描述: 快速条件短跳转。若条件为真, 加有符号 8 位常数到当前 PC 值并跳转; 否则继续执行而不跳转:
- If (测试条件 = 真) PC = PC + 有符号 8 位偏移量;
If (测试条件 = 假) PC = PC + 1;
- 描述: 快速短跳转 (SBF) 指令利用 C28x 核的复预取序列, 产生的跳转周期从 7 个减少到 4 个:
- 若 (测试条件 = 真) 则指令需要 4 周期。
若 (测试条件 = 假) 则指令需要 4 周期。
- 标志与模式: 无
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

SBRK, #8bit

从当前辅助寄存器中减去 8 位立即数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SBRK, #8bit	1111 1101 CCCC CCCC	X	-	1

- 操作数: #8bit 8 位立即常数
- 描述: 从 ARP 指向的 XARn 寄存器中减去 8 位无符号常数:
- $XAR(ARP) = XAR(ARP) - 0:8bit$;
- 标志与模式: ARP 3 位的 ARP 指向当前辅助寄存器 XAR0~XAR7。这个指针指定操作改变哪一个辅助寄存器。
- 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

.word 0xEEEE
.word 0x0000
TableA: .word 0x1111
        .word 0x2222
        .word 0x3333
        .word 0x4444

```

FuncA:

```

MOVL  XAR1, #TableA ;初始化 XAR1 指针
MOVZ  AR2, *XAR1     ;用 XAR1 (0x1111) 指向的 16 位数装载 AR2
                        ;设置 ARP=1
SBRK   #2             ;XAR1 减 2
MOVZ  AR3, *XAR1     ;用 XAR1 (0xEEEE) 指向的 16 位数装载 AR3

```

SETC Mode

设置多重状态位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SETC Mode	0011 1011 CCCC CCCC	×	—	1, 2
SETC SXM	0011 1011 0000 0001	×	—	1
SETC OVM	0011 1011 0000 0010	×	—	1
SETC TC	0011 1011 0000 0100	×	—	1
SETC C	0011 1011 0000 1000	×	—	1
SETC INTM	0011 1011 0001 0000	×	—	2
SETC DBGM	0011 1011 0010 0000	×	—	2
SETC PAGE0	0011 1011 0100 0000	×	—	1
SETC VMAP	0011 1011 1000 0000	×	—	1

操作数: mode 8 位屏蔽值 (0x00~0xFF)
描述: 设置指定的状态位。“mode”屏蔽值与以下的状态位相关:

“Mode” 位	状态寄存器	标志位	周期
0	ST0	SXM	1
1	ST0	OVM	1
2	ST0	TC	1
3	ST0	C	1
4	ST1	INTM	2
5	ST1	DBGM	2
6	ST1	PAGE0	1
7	ST1	VMAP	1

注意: 汇编程序接受任何顺序的标志位名。例如:
SETC INTM, TC ; 设置 INTM 和 TC 位为 1
SETC TC, INTM, OVM, C ; 设置 TC, INTM, OVM, C 位为 1。

标志与模式: SXM 此指令可以设置任意指定的位。
OVM
TC
C
INTM
DBGM
PAGE0
VMAP

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:
; 改变标志位设置:

```

SETC  INTM,DBGM          ; 设置 INTM 和 DBGM 位为 1
CLRC  TC,C,SXM,OVM       ; TC, C, SXM, OVM 位清 0
CLRC  #0xFF              ; 所有位清 0
SETC  #0xFF              ; 设置所有位为 1
SETC  C,SXM,TC,OVM       ; 设置 TC, C, SXM, OVM 位为 1
CLRC  DBGM,INTM          ; INTM 和 DBGM 位清 0

```

SETC M0M1MAP

设置 M0M1MAP 状态位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SETC M0M1MAP	0101 0110 0001 1010	×	-	5

操作数: M0M1MAP 状态位

描述: 设置 M0M1MAP 状态位, 为 C28x/C2XLP 配置 M0 和 M1 存储器块。存储器块映射如下:

M0M1MAP 位	数据空间	程序空间
0	M0: 0x000 to 0x3FF	M0: 0x400 to 0x7FF
(C27x)	M1: 0x400 to 0x7FF	M1: 0x000 to 0x3FF
1		M0: 0x000 to 0x3FF
(C28x/C2XLP)		M1: 0x400 to 0x7FF

注意: 当指令执行时, 刷新流水线。

标志与模式: M0M1MAP M0M1MAP 位置位。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

;复位时设置到 C28x 芯片工作方式

Reset:

```

SETC  OBJMODE    ; 使能 C28x 目标方式
CLRC  AMODE      ; 使能 C28x 寻址方式
.c28_ amode      ; 告诉汇编程序处于 C28x 寻址方式
SETC  M0M1MAP    ; 使能 C28x 的 M0 和 M1 块映射

```

SETC OBJMODE

设置 OBJMODE 状态位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SETC OBJMODE	0101 0110 0001 1111	×	-	5

操作数: OBJMODE 状态位

描述: 设置 OBJMODE 状态位, 使设备处于 C28x 目标方式 (支持 C2xLP 源程序)。

标志与模式: OBJMODE 设置 OBJMODE 位。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器

RPTC, 并且只执行一次。

例:

;复位时设置到 C28x 芯片工作方式:

Reset:

```
SETC  OBJMODE      ; 使能 C28x 目标方式
CLRC  AMODE        ; 使能 C28x 寻址方式
.c28_amode         ; 告诉汇编程序处于 C28x 寻址方式
SETC  M0M1MAP      ; 使能 C28x 的 M0 和 M1 块映射
```

SETC XF

设置 XF 状态位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SETC XF	0101 0110 0010 0110	×	-	1

操作数: XF 状态位和输出信号

描述: 设置 XF 状态位, 拉高相应的输出信号。

标志与模式: XF XF 状态位置位。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 若无跳转, XF 信号输出一个脉冲:

```
MOV AL, @VarA      ; 用 VarA 的内容装载 AL
SB  Dest, NEQ      ; ACC = VarA
SETC XF           ; XF 位置高, 使输出信号变高
CLRC XF           ; XF 位清 0, 使输出信号变低
```

Dest:

SFR ACC, #1...16

累加器右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SFR ACC, #1...16	1111 1111 0100 SHFT	×	Y	N+1

操作数: ACC 累加器
#1...16 移位位数

描述: ACC 累加器按移位位数右移。移位的类型 (算术或逻辑) 由符号扩展方式 (SXM) 位指定:

```
if (SXM = 1)          // 符号扩展方式使能
    ACC = S:ACC >> 移位位数; // 算术右移
else                  // 禁止符号扩展方式
    ACC = 0:ACC >> 移位位数; // 逻辑右移
```

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。

C 移出的最后一位送到进位标志位 C 中。

SXM 若 (SXM=1)，则操作为算术右移。

若 (SXM=0)，则操作为逻辑右移。

重复性： 本指令可以重复。若指令跟在 RPT 指令之后，则 SFR 指令执行 N+1 次。
Z、N、C 的状态为最终的结果。

例：

```
; VarA 的内容算术右移 10 位
MOVL  ACC,@VarA    ; ACC = VarA
SETC   SXM          ; 使能符号扩展方式
SFR    ACC,#10      ; ACC 算术右移 10 位
MOVL   @VarA,ACC    ; 结果保存到 VarA
```

SFR ACC,T

累加器右移

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SFR ACC,T	1111 1111 0101 0001	×	-	1

操作数： ACC 累加器

T 被乘数 (XT) 寄存器的高 16 位

描述： 按 T 寄存器的最低 4 位指定的数 T(3:0)=0...15 右移 ACC 累加器的值。
忽略 T 寄存器的高位。移位的类型 (算术或逻辑) 由符号扩展方式 (SXM) 位的状态指定：

```
if (SXM = 1)                                // 符号扩展方式使能
    ACC = S:ACC >> T(3:0);                  // 算术右移
else                                          // 禁止符号扩展方式
    ACC = 0:ACC >> T(3:0);                  // 逻辑右移
```

标志与模式： **Z** 若 ACC 的值为 0，则零标志位 Z 置位；否则 Z 清 0。即使 T 寄存器指定移位位数为 0，还是要测试 ACC 累加器的值是否为 0，则同样要影响 Z。

N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则清 0。即使 T 寄存器指定移位位数为 0，还是要测试 ACC 是否为负，将影响 N。

C 若 T(3:0)=0，则进位标志位 C 清 0；否则，移出的最后一位送到进位标志位 C 中。

SXM 若 (SXM=1)，则操作为算术右移。

若 (SXM=0)，则操作为逻辑右移。

重复性： 本指令可以重复。若指令跟在 RPT 指令之后，则 SFR 指令执行 N+1 次。
Z、N、C 的状态为最终的结果。

例：

```
;VarA 的内容算术右移 VarB 位
MOVL  ACC,@VarA    ; ACC = VarA
MOV   T,@VarB      ; T = VarB (移位位数)
```


SETC SXM ; 使能符号扩展方式
 SFR ACC, T ; ACC 算术右移 T(3:0) 位
 MOVL @VarA, ACC ; 结果保存到 VarA

SPM shift

设置乘积移位值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SPM +1	1111 1111 0110 1000	×	—	1
SPM 0	1111 1111 0110 1001	×	—	1
SPM -1	1111 1111 0110 1010	×	—	1
SPM -2	1111 1111 0110 1011	×	—	1
SPM -3	1111 1111 0110 1100	×	—	1
SPM -4 (当 AMODE = 0 有效) SPM +4 (当 AMODE = 1 有效)	1111 1111 0110 1101	×	—	1
SPM -5	1111 1111 0110 1110	×	—	1
SPM -6	1111 1111 0110 1111	×	—	1

操作数: shift 乘积移位方式 (+4, +1, 0, -1, -2, -3, -4, -5, -6)
 描述: 指定一个乘积移位方式。负值表示一个算术右移; 正数表示一个逻辑左移。下面说明“shift”操作数和装载到 ST0 的乘积移位方式 (PM) 的 3 位数之间的关系。寻址方式位 (AMODE) 在下表所示的移位译码的两种类型中选择:

APM 位	AMODE = 1	AMODE = 0
000	SPM +1	SPM +1
001	SPM 0	SPM 0
010	SPM -1	SPM -1
011	SPM -2	SPM -2
100	SPM -3	SPM -3
101	SPM +4	SPM -4
110	SPM -5	SPM -5
111	SPM -6	SPM -6

标志与模式: PM PM 装载“shift”所指定的 3 位值。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则将执行 N+1 次。Z、N、C 的状态为最终的结果。

例:

```
; 计算 Y32 = M16×X16 >> 4 + B32
CLRC AMODE      ; 确保 AMODE = 0
SPM -4          ; 设置乘积右移 4 位
MOV T, @X16     ; T = X16
MPY P, XT, @M16 ; P = X16×M16
MOVL ACC, @B32  ; ACC = B32
ADDL ACC, P << PM ; ACC = ACC + (P >> 4)
MOVL @Y32, ACC  ; 结果保存到 Y32
```

SQRA loc16

平方并加 P 到 ACC

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SQRA loc16	0101 0110 0001 0101 0010 0010 LLLL LLLL	1	Y	N+1

操作数: loc16 寻址方式

描述: 把先前乘积值 (保存在 P 寄存器) 按乘积移位方式 (PM) 指定的值移位后加到 ACC 累加器。把 “loc16” 地址单元的值装载到 T 寄存器并且平方后保存到 P 寄存器:

$$ACC = ACC + P \ll PM;$$

$$T = [loc16];$$

$$P = T \times [loc16];$$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 若产生进位, 则进位标志位 C 置位; 否则进位标志位 C 清 0。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVC 若禁止溢出模式, 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。

OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

PM PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (算术右移操作), 则高位进行符号扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则指令执行 N+1 次。Z、N、C 和 OVC 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。

例:

```

; 使用 16 位乘法计算平方, 保存到 sum
; int16 X[N]           ; 数据信息
; sum = 0;
; for(i = 0; i < N; i++)
; sum = sum + (X[i] * X[i]) >> 5;
MOVL    XAR2, #X        ; XAR2 指针指向 X
SPM     -5              ; 设置乘积右移 5 位
ZAPA                    ; ACC, P, OVC 清 0
RPT     #N-1            ; 重复下一条指令 N 次
||SQRA  *XAR2+ +        ; ACC = ACC + P >> 5, P = (*XAR2+ +)^2
ADDL    ACC, P << PM    ; 执行最终计算
MOVL    @sum, ACC       ; 保存最终结果到 sum

```

SQRS loc16

平方并从 ACC 中减去 P

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SQRS loc16	0101 0110 0001 0001 xxxx xxxx LLLL LLLL	1	Y	N+1

操作数: loc16 寻址方式

描述: 先前乘积值（保存在 P 寄存器）按乘积移位方式（PM）指定的值移位后从 ACC 累加器中减去。把“loc16”地址单元的内容装载到 T 寄存器并且平方后保存在 P 寄存器:

```
ACC = ACC - P << PM;
T = [loc16];
P = T × [loc16];
```

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 OVC 若禁止溢出模式, 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。
 OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 PM PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (算术右移操作), 则高位进行符号扩展。

重复性: 本指令可以重复。若指令跟在 RPT 指令之后, 则指令执行 N+1 次。Z、N、C 和 OVC 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。

例:

```
; 使用 16 位乘计算负数的平方和
; int16 X[N] ; 数据信息
; sum = 0;
; for(i = 0; i < N; i++)
; sum = sum - (X[i] × X[i]) >> 5;
MOVL  XAR2, #X          ; XAR2 指针指向 X
SPM   -5                ; 设置乘积右移 5 位
ZAPA                      ; ACC, P, OVC 清 0
RPT   #N-1              ; 重复下一条指令 N 次
||SQRS *XAR2+ +         ; ACC = ACC - P >> 5,
                        ; P = (*XAR2+ +)^2
SUBL  ACC, P << PM      ; 执行最终的减
MOVL  @sum, ACC         ; 保存最终结果到 sum
```

SUB ACC,loc16 << #0...16

从累加器中减去移位后的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUB ACC,loc16<<#0	1010 1110 LLLL LLLL	1	Y	N+1
	1000 0000 LLLL LLLL	0	-	1
SUB ACC,loc16<<#1...15	0101 0110 0000 0000	1	Y	N+1
	0000 SHFT LLLL LLLL			
SUB ACC,loc16<<#16	1000 SHFT LLLL LLLL	0	-	1
	0000 0100 LLLL LLLL	X	Y	N+1

操作数: ACC 累加器
 loc16 寻址方式
 #0...16 移位位数 (没有指定值时, 默认为 “<<#0”)
 描述: 从 ACC 累加器中减去左移后的 “loc16” 地址单元的值。若符号扩展方式有效 (SXM = 1), 移位时进行符号扩展; 否则移位时为 0 扩展 (SXM = 0), 最低位填 0:

```
if (SXM = 1)          // 符号扩展方式使能
    ACC = ACC - S: [loc16] << 移位位数;
else                  // 禁止符号扩展方式
    ACC = ACC - 0: [loc16] << 移位位数;
```

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 例外: 若使用 16 位移位, SUB 指令将进位标志位 C 清 0 而不是置位。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 OVC 若禁止溢出模式 (OVM = 0), 若操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作影响。
 SXM 若符号扩展方式位置位, 则由 “loc16” 寻址的 16 位数, 在操作前作符号扩展。否则进行 0 扩展。
 OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 重复性: 若本指令可以重复, 则将执行 N+1 次。Z、N、C 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。若禁止溢出模式, OVC 将计数中间溢出次数。若本指令不可以重复, 操作只执行一次。

例:

```
; 计算有符号值 ACC = (VarA << 10) - (VarB << 6);
SETC SXM                      ; 使能符号扩展方式
MOV ACC, @VarA << #10         ; VarA 左移 10 位后装载 ACC
SUB ACC, @VarB << #6          ; 从 ACC 减去 VarB 左移 6 位后的值
```

SUB ACC, loc16<<T

从累加器中减去移位后的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUB ACC, loc16<<T	0101 0110 0010 0111 0000 0000 LLLL LLLL	1	Y	N+1

操作数: ACC 累加器
 loc16 寻址方式
 T 被乘数寄存器, XT(31: 16)高 16 位
 描述: 从 ACC 累加器减去左移后的 “loc16” 地址单元的内容。其左移位数由 T

寄存器的最低 4 位 $T(3:0) = 0...15$ 指定, 忽略 T 寄存器的高位。若符号扩展方式使能 ($SXM = 1$), 移位时进行符号扩展, 否则移位时为 0 扩展 ($SXM=0$), 最低位填 0:

```
if(SXM = 1)           // 使能符号扩展方式
    ACC = ACC - S: [16:0] << T(3:0);
else                  // 禁止符号扩展方式
    ACC = ACC - 0: [16:0] << T(3:0);
```

- 标志与模式:
- Z** 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 - N** 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 - C** 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 - V** 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 - OVC** 若禁止溢出模式 ($OVM = 0$), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 ($OVM = 1$), 则计数器不受操作影响。
 - SXM** 若符号扩展方式位置位; 则由 “loc16” 寻址的 16 位数, 在操作前作符号扩展。否则, 作 0 扩展。
 - OVM** 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 ($0x7FFFFFFF$) 或最小负数 ($0x80000000$)。
- 重复性: 若本指令可以重复, 则指令将执行 N+1 次。Z、N、C 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。若禁止溢出模式, OVC 将计数中间溢出次数。

例:

```
; 计算有符号值 ACC = (VarA << SB) - (VarB << SB)
SETC    SXM                ; 使能符号扩展方式
MOV     T,@SA              ; 用 SA 移位位数装载 T
MOV     ACC,@VarA << T     ; 用 VarA 移位后的内容装载
MOV     T,@SB              ; 用 SB 移位位数装载 T
SUB     ACC,@VarB << T     ; 从 ACC 减去 VarB 移位后的内容
```

SUB ACC,#16bit<<#0...15

从累加器中减去移位后的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUB ACC,#16bit<<#0...15	1111 1111 0000 SHFT CCCC CCCC CCCC CCCC	×	-	1

- 操作数: ACC 累加器
 #16bit 16 位立即常数
 #0...15 移位位数 (没有指定值时, 默认为 “<<#0”)
- 描述: 从 ACC 累加器减去左移后的 16 位立即常数。若符号扩展方式使能 ($SXM = 1$), 移位时进行符号扩展, 否则移位时为 0 扩展 ($SXM=0$), 最低位填 0:

```
if(SXM = 1)           // 符号扩展方式使能
```



```

        ACC = ACC - S: 16bit << 移位位数;
else
        // 禁止符号扩展方式
        ACC = ACC - 0: 16bit << 移位位数;

```

- 标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 OVC 若禁止溢出模式 (OVM = 0), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作影响。
 SXM 若符号扩展方式位置位; 则由 “loc16” 寻址的 16 位数, 在操作前作符号扩展。否则进行 0 扩展。
 OVM 若溢出模式位置位, 当操作发生溢出时, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 计算有符号值 ACC = (VarB << 10) - (23 << 6);
SETC    SXM                                ; 使能符号扩展方式
MOV     ACC, @VarB << #10                  ; 用 VarB 左移 10 位装载 ACC
SUB     ACC, #23 << #6                     ; 从 ACC 减去左移 6 位后的 #23 的值

```

SUB AX,loc16

从 AX 中减去指定单元的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUB AX,loc16	1001 111A LLLL LLLL	×	-	1

- 操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器
 loc16 寻址方式
 描述: 从指定的 AX 寄存器 (AH 或 AL) 中减去 “loc16” 地址单元的 16 位数, 结果保存到 AX;
 $AX = AX - [loc16];$

- 标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。
 Z 若操作产生 AX=0, 则零标志位 Z 置位; 否则清 0。
 C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
 V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。若在正方向结果超出最大正数 (0x7FFF), 产生符号正溢出。若在负方向结果超出最小负数 (0x8000), 产生符号负溢出。
 重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 从 VarA 的内容中减去 VarB 并保存到 VarC
MOV AL,@VarA      ; 用 VarA 的内容装载 AL
SUB AL,@VarB      ; 从 AL 中减去 VarB 的内容
MOV @VarC,AL      ; 结果保存到 VarC
```

SUB loc16,AX

从指定单元的值中减去 AX

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUB loc16,AX	0111 010A LLLL LLLL	×	-	1

操作数: loc16 寻址方式

AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述: 从 “loc16” 地址单元指定的 16 位数中减去 AX 寄存器 (AH 或 AL) 的值, 结果保存到 “loc16” 寻址地址单元中:

$[loc16] = [loc16] - AX;$

标志与模式: N 若[loc16]的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若操作产生[loc16]=0, 则零标志位 Z 置位; 否则清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。若在正方向结果超出最大正数 (0x7FFF), 产生符号正溢出。若在负方向结果超出最小负数 (0x8000), 产生符号负溢出。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 从索引寄存器 AR0 中减去 VarA 的内容
MOV AL,@VarA      ; 用 VarA 的内容装载 AL
SUB @AR0,AL       ; AR0 = AR0 - AL
; 从 arC 减去 VarB 的内容:
MOV AH,@VarB      ; 用 VarB 的内容装载 AH
SUB @VarC,AH      ; VarC = VarC - AH
```

SUBB ACC,#8bit

减去 8 位立即数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBB ACC,#8bit	0001 1001 CCCC CCCC	×	-	1

操作数: ACC 累加器

#8bit 8 位立即数

描述: 从 ACC 累加器中减去 0 扩展后的 8 位立即数:

$ACC = ACC - 0:8bit;$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

- C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。
- V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
- OVC 若禁止溢出模式 (OVM = 0), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作影响。
- OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 32 位 VarA 减 1
MOVL  ACC,@VarA  ; 用 VarA 的内容装载 ACC
SUBB  ACC,#1      ; 从 ACC 减 1
MOVL  @VarA,ACC   ; 结果保存到 VarA
```

SUBB SP,#7bit

SP 减去 7 位无符号常数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBB SP,#7bit	1111 1110 1CCC CCCC	×	-	1

操作数: SP 堆栈指针
#7bit 7 位立即数

描述: 从 SP 中减去一个 7 位无符号常数, 结果保存到 SP:

$SP = SP - 0:7bit;$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
FuncA:                ; 堆栈中有函数的本地变量。
    ADDB SP,#N         ; 在堆栈中为本地变量保留 N 个 16 位字节的空间
    .
    .
    .
    SUBB SP,#N         ; 分配保留的堆栈空间
    LRETR              ; 从函数返回
```

SUBB XARn,#7bit

从辅助寄存器中减去 7 位立即数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBB XARn,#7bit	1101 1nnn 1CCC CCCC	×	-	1

操作数: XARn XAR0~XAR7, 32 位辅助寄存器
#7bit 7 位立即数

描述: 从 XARn 减去 7 位无符号常数, 结果保存到 XARn:

$$XARn = XARn - 0:7bit;$$

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

MOVL    XAR1, #VarA    ;用 VarA 初始化 XAR1 指针
MOVL    XAR2, *XAR1    ;用 VarA 的内容装载 XAR2
SUBB    XAR2, #10h      ; XAR2 = VarA - 0x10

```

SUBBL ACC,loc32

减去 32 位地址单元的数并减进位位的反

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBBL ACC,loc32	0101 0110 0101 0100 0000 0000 LLLL LLLL	1	-	1

操作数: loc32 寻址方式

ACC 累加器

描述: 从 ACC 减去“loc32”寻址方式指向的 32 位数和进位标志位的逻辑反。

$$ACC = ACC - [loc32] - \sim C;$$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 进位位执行前的状态包含在减指令中。若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVC 若禁止溢出模式 (OVM = 0), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作影响。

OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 减去两个 64 位数 (VarA 和 VarB) 并且结果保存到 VarC
MOVL    ACC, @VarA+0    ;用 VarA 低 32 位数装载 ACC
SUBUL    ACC, @VarB+0    ;从 ACC 减去 VarB 的低 32 位数
MOVL    @VarC+0, ACC     ;保存结果的低 32 位到 VarC
MOVL    ACC, @VarA+2    ;用 VarA 的高 32 位数装载 ACC
SUBBL    ACC, @VarB+2    ;从 ACC 带借位减去 VarB 的高 32 位数
MOVL    @VarC+2, ACC     ;保存结果的高 32 位到 VarC

```

SUBCU ACC,loc32

条件减 32 位数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBCU ACC,loc32	0000 1111 LLLL LLLL	×	Y	N+1

操作数: ACC 累加器
 loc32 寻址方式
 描述: 执行 16 位条件减法, 可以用作无符号模数除法:

```
temp(32:0) = ACC << 1 - [loc16] << 16
  if( temp(32:0) >= 0 )
    ACC = temp(31:0) + 1
  else
    ACC = ACC << 1
```

为了执行 16 位无符号模数除法, 在执行 SUBCU 指令前, AH 寄存器为 0, AL 寄存器装载“分子”值。先执行 SUBCU 指令。“loc16”地址单元的值为“分母”。执行 SUBCU 指令 16 次后, AH 寄存器为“余数”, AL 寄存器为“商”。要执行有符号模数除, 在执行 SUBCU 指令前, “分子”和“分母”值必须先转化为无符号量。若“分子”和“分母”值是不同的符号, 最终的“商”结果必须取负, 否则商不变。

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。temp (32:0) 的计算对零标志位 Z 没有影响。
 N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。temp (32:0) 的计算对负标志位 N 没有影响。
 C 若 temp (32:0) 的计算产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

注意: V 和 OVC 不受操作的影响。

重复性: 本指令可以重复, 则指令执行 N+1 次。Z、N、C 的状态为最终的结果。若操作中间产生溢出, 则 V 标志位置位。若禁止溢出模式, OVC 将计数中间溢出次数。

例 1:

```
; 计算无符号数 Quot16 = Num16/Den16, Rem16 = Num16%Den16
MOVU    ACC,@Num16                ; AL = Num16, AH = 0
RPT      #16                      ; 重复操作 16 次
||SUBCU  @Den16                   ; 条件减去 Den16
MOV      @Rem16,AH                ; 保存余数到 Rem16
MOV      @Quot16,AL               ; 保存商到 Quot16
```

例 2:

```
; 计算有符号数 Quot16 = Num16/Den16, Rem16 = Num16%Den16
CLRC     TC                      ; TC 标志位清 0
MOV      ACC,@Den16 << 16        ; AH = Den16, AL = 0
ABSTC    ACC                     ; 取绝对值, TC = sign 异或 TC
MOV      T,@AH                   ; Temp 保存 Den16 到 T 寄存器
MOV      ACC,@Num16 << 16        ; AH = Num16, AL = 0
ABSTC    ACC                     ; 取绝对值, TC = sign 异或 TC
MOVU     ACC,@AH                  ; AH = 0, AL = Num16
```



```

RPT      #15                ; 重复下面的操作 16 次
||SUBCU  @T                  ; 条件减去 Den16
MOV      @Rem16,AH           ; 保存余数到 Rem16
MOV      ACC,@AL<<16         ; AH = Quot16, AL = 0
NEGTC    ACC                 ; 若 TC = 1, 取负
MOV      @Quot16,AH          ; 保存商到 Quot16

```

例 3:

```

; 计算无符号数
; Quot32 = Num32/Den16, Rem16 = Num32%Den16
MOVU     ACC,@Num32+1        ; AH = 0, AL = Num32 的高 16 位
RPT      #15                ; 重复操作 16 次
||SUBCU  @Den16              ; 条件减去 Den16
MOV      @Quot32+1,AL        ; 保存高 16 位到 Quot32
MOV      AL,@Num32+0         ; AL = Num32 的低 16 位
RPT      #15                ; 重复下面的操作 16 次
||SUBCU  @Den16              ; 条件减去 Den16
MOV      @Rem16,AH           ; 保存余数到 Rem16
MOV      @Quot32+0,AL        ; 保存低 16 位到 Quot32

```

例 4:

```

; 计算有符号数
; Quot32 = Num32/Den16, Rem16 = Num32%Den16
CLRC     TC                  ; 标志位 TC 清 0
MOV      ACC,@Den16 << 16   ; AH = Den16, AL = 0
ABSTC    ACC                 ; 取绝对值, TC = sign 异或 TC
MOV      T,@AH               ; 保存 Den16 到 T 寄存器
MOVL     ACC,@Num32          ; ACC = Num32
ABSTC    ACC                 ; 取绝对值, TC = sign 异或 TC
MOV      P,@ACC              ; P = Num32
MOVU     ACC,@PH              ; AH = 0, AL = Num32 的高 16 位
RPT      #15                ; 重复操作 16 次
||SUBCU  @T                   ; 条件减去 Den16
MOV      @Quot32+1,AL        ; 保存高 16 位到 Quot32
MOV      AL,@PL              ; AL = Num32 的低 16 位
RPT      #15                ; 重复操作 16 次
||SUBCU  @T                   ; 条件减去 Den16
MOV      @Rem16,AH           ; 保存余数到 Rem16
MOV      ACC,@AL << 16      ; AH = Quot32 的低 16 位, AL = 0
NEGTC    ACC                 ; 若 TC = 1, 取负
MOV      @Quot32+0,AH        ; 保存低 16 位到 Quot32

```

SUBCUL ACC,loc32**条件减 32 位数**

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBCUL ACC,loc32	0101 0110 0001 0111 0000 0000 LLLL LLLL	I	Y	N+1

操作数: ACC 累加器
 loc32 寻址方式

描述: 执行 32 位条件减法, 可以用作无符号模数除法

```
temp(32:0) = ACC << 1 + P(31) - [loc32];
if( temp(32:0) >= 0 )
    ACC = temp(31:0);
    P = (P << 1) + 1;
else
    ACC:P = ACC:P << 1;
```

为了执行 32 位无符号模数除法, 在执行 SUBCUL 指令前, ACC 累加器为 0, P 寄存器装载“分子”, 先执行 SUBCUL 指令。“loc32”地址单元的值为“分母”。执行 SUBCUL 指令 32 次后, ACC 累加器为“余数”, P 寄存器为“商”。要执行有符号模数除法, 在执行 SUBCUL 指令前, “分子”和“分母”值必须转化为无符号量。若“分子”和“分母”值是不同的符号, 最终的“商”结果必须取负, 否则商不变。

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。temp(32:0) 的计算对零标志位 Z 没有影响。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。temp(32:0) 的计算对负标志位 N 没有影响。

C 若 temp(32:0) 的计算产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

注意: V 和 OVC 不受操作的影响。

重复性: 本指令可以重复, 将执行 N+1 次。Z、N、C 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。若禁止溢出模式, OVC 计数中间溢出次数。

例 1:

```
; 计算无符号数
; Quot32 = Num32/Den32, Rem32 = Num32%Den32
MOVB    ACC, #0                ; ACC 清 0
MOVL    P, @Num32              ; 用 Num32 装载 P
RPT     #31                    ; 重复下面的操作 32 次
||SUBCUL @Den32                 ; 条件减去 Den32
MOVL    @Rem32, ACC            ; 保存余数到 Rem32
MOVL    @Quot32, P             ; 保存商到 Quot32
```

例 2:

```
; 计算有符号数 Quot32 = Num32/Den32, Rem32 = Num32%Den32
CLRC    TC                    ; TC 标志位清 0
MOVL    ACC, @Den32           ; 用 Den32 的内容装载 ACC
ABSTC   ACC                   ; 取绝对值, TC = sign 异或 TC
MOVL    XT, @ACC              ; 保存分母到 XT 寄存器
MOVL    ACC, @Num32           ; 用 Num32 装载 ACC 寄存器
ABSTC   ACC                   ; 取绝对值, TC = sign 异或 TC
MOVL    P, @ACC               ; 用分子装载 P 寄存器
MOVB    ACC, #0               ; ACC 清 0
```

```

    RPT      #31                ; 重复操作 32 次
|| SUBCUL   @XT                ; 用分母条件减
    MOVL     @Rem32, ACC        ; 保存余数到 Rem32
    MOVL     ACC, @P            ; 用商装载 ACC
    NEGTC    ACC               ; 若 TC = 1, ACC 取负
    MOVL     @Quot32, ACC       ; 保存商到 Quot32

```

例 3:

```

; 计算无符号
; Quot64 = Num64Den32, Rem32 = Num64%Den32
MOVB      ACC, #0              ; ACC 清 0
MOVL      P, @Num64+2          ; 用 Num64 的高 32 位装载 P
    RPT      #31                ; 重复操作 32 次
|| SUBCUL   @Den32              ; 用 Den32 条件减
    MOVL     @Quot64+2, P        ; 保存高 32 位商到 Quot64
    MOVL     P, @Num64+0         ; 用 Num64 的低 32 位装载 P
    RPT      #31                ; 重复操作 32 次
|| SUBCUL   @Den32              ; 用 Den32 条件减
    MOVL     @Rem32, ACC         ; 保存余数到 Rem32
    MOVL     @Quot64+0, P        ; 保存低 32 位商到 Quot64

```

例 4:

```

; 计算有符号 Quot64 = Num64Den32, Rem32 = Num64%Den32
MOVL      ACC, @Num64+2        ; 用 64 位分子装载 ACC: P
MOVL      P, @Num64+0
TBIT      @AH, #15             ; TC = 分子标志位
SEF       $10, NTC             ; 分子取绝对值
NEG64     ACC: P
$10:
    MOVL     @XAR3, P           ; Temp 保存分子低位到 XAR3
    MOVL     P, @ACC            ; 用分子高位装载 P 寄存器
    MOVL     ACC, @Den32        ; 用 Den32 的内容装载 ACC
    ABSTC    ACC               ; 取绝对值, TC = sign 异或 TC
    MOVL     XT, @ACC           ; Temp 保存分母到 XT 寄存器
    MOVB     ACC, #0           ; ACC 清 0
    RPT      #31                ; 重复操作 32 次
|| SUBCUL   @XT                ; 用分母条件减
    MOVL     @XAR4, P           ; 保存高位商到 XAR4
    MOVL     P, @XAR3           ; 用低位分子装载 P
    RPT      #31                ; 重复操作 32 次
|| SUBCUL   @XT                ; 用分母条件减
    MOVL     @Rem32, ACC        ; 保存余数到 Rem32
    MOVL     ACC, @XAR4         ; 用 XAR4 的高位商装载 ACC
    SBF      $20, NTC           ; 取商的绝对值
    NEG64     ACC: P
$20:
    MOVL     @Quot64+0, P        ; 保存商的低位到 Quot64
    MOVL     @Quot64+2, ACC      ; 保存商的高位到 Quot64

```

SUBL ACC,loc32

减 32 位数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBL ACC, loc32	0000 0011 LLLL LLLL	1	-	1

操作数: ACC 累加器

loc32 寻址方式

描述: 从 ACC 累加器中减去“loc32”寻址地址单元中的 32 位数:

$$ACC = ACC - [loc32];$$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVC 若禁止溢出模式 (OVM = 0), 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。若溢出模式使能 (OVM = 1), 则计数器不受操作影响。

OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 计算 32 位数 VarC = VarA-VarB

MOVL ACC,@VarA ; 用 VarA 的内容装载 ACC

SUBL ACC,@VarB ; 从 ACC 减去 VarB 的内容

MOVL @VarC,ACC ; 结果保存到 VarC

SUBL ACC,P<<PM

减 32 位数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBL ACC,P<<PM	0001 0001 1010 1100	×	Y	N+1

注意: 本指令是“loc16=@T”寻址方式下的“MOVS T,loc16”指令的一个别名。

操作数: ACC 累加器

P 乘积寄存器

<<PM 乘积移位方式

描述: 从 ACC 累加器的值中减去按乘积移位方式 (PM) 指定移位后的 P 寄存器内容:

$$ACC = ACC - P << PM;$$

标志与模式: Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

- C** 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置位。
- V** 若产生溢出，则溢出标志位 V 置位；否则不影响 V。
- OVC** 若禁止溢出模式（OVM = 0），操作产生正溢出，则计算器值增加；若操作产生负溢出，则计算器值减少。若溢出模式使能（OVM = 1），则计数器不受操作影响。
- OVM** 若溢出模式位置位，操作发生溢出，则 ACC 的值为最大正数（0x7FFFFFFF）或最小负数（0x80000000）。
- PM** PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正（逻辑左移操作），则最低位填 0。若乘积移位位数为负（算术右移操作），则高位进行符号扩展。

重复性： 本指令可以重复，则指令将执行 N+1 次。Z、N、C 的状态为最终的结果。若中间产生溢出，则 V 标志位置位。若禁止溢出模式，OVC 计数中间溢出次数。

例：

```

;计算 Y = ((B << 11) - (M×X >> 4)) >> 10; Y, M, X, B 是 Q15 值
SPM      -4                ; 设置乘积右移 4 位
SETC     SXM               ; 符号扩展方式使能
MOV      T,@M              ; T = M
MPY      P,T,@X            ; P = M × X
MOV      ACC,@B << 11      ; ACC = S: B << 11
SUBL     ACC,P << PM       ; ACC = (S: B << 11) - (M×X >> 4)
MOVH     @Y,ACC << 5       ; 保存 Q15 结果到 Y

```

SUBL loc32,ACC

减 32 位数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBL loc32,ACC	0101 0110 0100 0001 0000 0000 LLLL LLLL	1	-	1

操作数： loc32 寻址方式
ACC 累加器

描述： 从“loc32”寻址地址单元的内容减去 ACC 累加器的值，结果保存到“loc32”寻址地址单元中：

$[loc32] = [loc32] - ACC;$

- 标志与模式：** **Z** 若 ACC 的值为 0，则零标志位 Z 置位；否则 Z 清 0。
- N** 若[loc32]的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。
- C** 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置位。
- V** 若产生溢出，则溢出标志位 V 置位；否则不影响 V。
- OVC** 若禁止溢出模式（OVM = 0），操作产生正溢出，则计算器值增加；若操作产生负溢出，则计算器值减少。若溢出模式使能（OVM = 1），则计数器不受操作影响。
- OVM** 若溢出模式位置位，操作发生溢出时，则 ACC 的值为最大正数

(0x7FFFFFFF) 或最小负数 (0x80000000)。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 32 位 VarA 减 1
MOVB ACC, #1      ; 用 0x00000001 装载 ACC
SUBL @VarA, ACC    ; VarA = VarA - ACC
```

SUBR loc16,AX

从 AX 中减去指定地址单元的值

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBR loc16, AX	1110 101A LLLL LLLL	1	-	1

操作数: loc16 寻址方式

AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述: 从指定的 AX 寄存器 (AH 或 AL) 减去 “loc16” 地址单元的 16 位数, 结果保存到 “loc16” 地址单元:

$[loc16] = AX - [loc16];$

本指令执行一个读取-修改-写入操作。

标志与模式: Z 测试[loc16]是否为负。若[loc16]的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

N 测试[loc16]是否为 0。若 [loc16]=0, 则零标志位 Z 置位; 否则清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。若结果在正方向超出最大正数 (0x7FFF), 产生符号正溢出。若结果在负方向超出最小负数 (0x8000), 产生符号负溢出。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 从 VarA 减去索引辅助寄存器 AR0 并将结果保存在 AR0:
MOV AL, @VarA      ; 用 VarA 的内容装载 AL
SUBR @AR0, AL      ; AR0 = AL - AR0
; 从 VarB 减去 VarC 的内容并保存到 VarC:
MOV AH, @VarB      ; 用 VarB 的内容装载 AH
SUBR @VarC, AH      ; VarC = AH - VarC
```

SUBRL loc32,ACC

从 ACC 中减去指定地址单元的内容

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBRL loc32, ACC	0101 0110 0100 1001 0000 0000 LLLL LLLL	1	-	1

操作数: loc32 寻址方式

ACC 累加器

描述: 从 ACC 累加器减去 “loc32” 地址单元的 32 位数，结果保存到 “loc32” 指向的位置。

$[loc32] = ACC - [loc32];$

标志与模式: Z 若 ACC 的值为 0，则零标志位 Z 置位；否则 Z 清 0。
 N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。
 C 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置位。
 V 若产生溢出，则溢出标志位 V 置位；否则不影响 V。
 OVC 若禁止溢出模式 (OVM = 0)，操作产生正溢出，则计算器值增加；若操作产生负溢出，则计算器值减少。若溢出模式使能 (OVM = 1)，则计数器不受操作影响。
 OVM 若溢出模式位置位，操作发生溢出，则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```
; 计算 32 位数 VarA = VarB - VarA
MOVL  ACC,@VarB    ; 用 VarB 的内容装载 ACC
SUBRL  @VarA,ACC    ; VarA = ACC - VarA
```

SUBU ACC,loc16

从 ACC 中减去 16 位无符号数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBU ACC,loc16	0000 0001 LLLL LLLL	×	Y	N+1

操作数: ACC 累加器
 loc16 寻址方式

描述: 从 ACC 累加器中减去 “loc16” 地址单元的 16 位数。减前 16 位数作 0 扩展：

$ACC = ACC - 0:[loc16];$

标志与模式: Z 若 ACC 的值为 0，则零标志位 Z 置位；否则 Z 清 0。
 N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。
 C 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置位。
 V 若产生溢出，则溢出标志位 V 置位；否则不影响 V。
 OVC 若禁止溢出模式 (OVM = 0)，操作产生正溢出，则计算器值增加；若操作产生负溢出，则计算器值减少。若溢出模式使能 (OVM = 1)，则计数器不受操作影响。
 OVM 若溢出模式位置位，操作发生溢出，则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。
重复性: 本指令可以重复，则指令将执行 N+1 次。Z、N、C 的状态为最终的结果。

若中间产生溢出，则 V 标志位置位。若禁止溢出模式，OVC 将计数中间的溢出次数。

例：

；用 16 位分部减去 3 个 32 位无符号数

```

MOVU    ACC,@VarAlow           ; AH = 0, AL = VarAlow
ADD     ACC,@VarAhigh << 16    ; AH = VarAhigh, AL = VarAlow
SUBU    ACC,@VarBlow           ; ACC = ACC - 0: VarBlow
SUB     ACC,@VarBhigh << 16    ; ACC = ACC - VarBhigh << 16
SBBU    ACC,@VarClow           ; ACC = ACC - VarClow - -C(-C 进位位取反)
SUB     ACC,@VarChigh << 16    ; ACC = ACC - VarChigh << 16

```

SUBUL ACC,loc32

减去 32 位无符号数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBUL ACC,loc32	0101 0110 0101 0101 0000 0000 LLLL LLLL	1	-	1

操作数： loc32 寻址方式

ACC 累加器

描述： 从 ACC 累加器减去“loc32”寻址方式指定的 32 位数，可以看作是无符号 SUBL 操作：

$ACC = ACC - [loc32]; \quad // \text{无符号减}$

注意：有符号和无符号 32 位减的区别是对待溢出计数器（OVC）的不同。对于一个有符号 SUBL，OVC 计数器监视正/负溢出。对一个无符号 SUBL，OVC 计数器（OVCU）监视借位。

标志与模式： Z 若 ACC 的值为 0，则零标志位 Z 置位；否则 Z 清 0。

N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。

C 若产生借位，则进位标志位 C 清 0；否则进位标志位 C 置位。

V 若产生溢出，则溢出标志位 V 置位；否则不影响 V。

OVCU 当减操作产生无符号借位，则溢出计算器值减少。OVM 方式不影响 OVCU 计数器。

重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

；减两个 64 位数 (VarA 和 VarB) 结果保存到 VarC:

```

MOVL    ACC,@VarA+0           ; 用 VarA 的低 32 位数装载 ACC
SUBUL    ACC,@VarB+0           ; 从 ACC 减去 VarB 的低 32 位数
MOVL    @VarC+0,ACC           ; 保存结果的低 32 位到 VarC
MOVL    ACC,@VarA+2           ; 用 VarA 的高 32 位数装载 ACC
SUBBL    ACC,@VarB+2           ; 从 ACC 带借位减去 VarB 的高 32 位数
MOVL    @VarC+2,ACC           ; 保存结果的高 32 位到 VarC

```

SUBUL P,loc32

减去 32 位无符号数

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
SUBUL P,loc32	0101 0110 0101 1101 0000 0000 LLLL LLLL	1	-	1

操作数: P 乘积寄存器

loc32 寻址方式

描述: 从 P 寄存器减去“loc32”地址单元的 32 位数。可以看作无符号 SUB 操作:

$$P = P - [\text{loc32}]; // \text{无符号减}$$

注意: 有符号和无符号 32 位减的区别是对待溢出计数器 (OVC) 的不同。对于一个符号 SUBL, OVC 计数器监视正/负溢出。对一个无符号 SUBL, OVC 计数器 (OVCU) 监视借位。

标志与模式: Z 若 P 的值为 0, 则零标志位 Z 置位; 否则 Z 清 0。

N 若 P 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。

C 若产生借位, 则进位标志位 C 清 0; 否则进位标志位 C 置位。

V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。

OVCU 当减操作产生无符号借位, 则溢出计算器值减少。OVM 方式不影响 OVCU 计数器。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 减 64 位的 VarA - VarB 并将结果保存到 VarC:

MOVL P,@VarA+0 ; 用 VarA 的低 32 位装载 P

MOVL ACC,@VarA+2 ; 用 VarA 的高 32 位装载 ACC

SUBUL P,@VarB+0 ; 从 P 减去 VarB 的无符号低 32 位

SUBBL ACC,@VarB+2 ; 从 ACC 带借位减去 VarB 的高 32 位

MOVL @VarC+0,P ; 保存结果的低 32 位到 VarC

MOVL @VarC+2,ACC ; 保存结果的高 32 位到 VarC

TBIT loc16,#16bit

测试指定位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TBIT loc16,#16bit	0100 BBBB LLLL LLLL	×	-	1

操作数: loc16 寻址方式

#16bit 立即常数位索引从 0~15

描述: 测试“loc16”地址单元中的指定位:

$$TC = [\text{loc16}(\text{bit})];$$

#bit 立即数指定位。例如, 若 #bit=0, 访问第 0 位 (最低位); 若 #bit=15, 访问第 15 位 (最高位)。

标志与模式: TC 若测试位为 1, 则 TC 置位; 若测试位为 0, 则 TC 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; if( VarA.Bit4 = 1 )
; VarB.Bit6 = 1;
; else
; VarB.Bit6 = 0;
TBIT    @VarA, #4      ; 测试 VarA 第 4 位
SB      $10, NTC       ; 若 TC = 0 则跳转
TSET    @VarB, #6      ; 设置 VarB 的第 6 位
SB      $20, UNC       ; 无条件跳转
$10:
TCLR    @VarB, #6      ; VarB 的第 6 位清 0
$20:

```

TBIT loc16,T

测试寄存器中指定的位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TBIT loc16,T	0101 0110 0010 0101 0000 0000 LLLL LLLL	1	-	1

操作数: loc16 寻址方式

T 被乘数寄存器 (XT) 的高 16 位

描述: 测试 T 寄存器的最低 4 位 $T(3:0) = 0 \dots 15$ 指定的 “loc16” 地址单元的位。忽略 T 寄存器的高位:

```

bit = 15 - T(3: 0);
TC = [loc16(bit)];

```

T 寄存器中的第 0 位 (最低位) 对应于 15 位。0 对应于 T 寄存器中的第 15 位 (最高位) 对应于 0 位。忽略 T 寄存器的高 12 位。

标志与模式: TC 若测试位为 1, 则 TC 置位; 若测试位为 0, 则 TC 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; if( VarA.VarB = 1 )
; VarC.Bit6 = 1;
; else
; VarC.Bit6 = 0;
MOV     T, @VarB      ; 用 VarB 的数装载 T
ADD     @T, #15       ; 保留位测试的顺序
TBIT    @VarA, T      ; 测试 VarB 指定的 VarA 中的位
SB      $10, NTC       ; 若 TC = 0 则跳转
TSET    @VarB, #6      ; 置 VarB 第 6 位
SB      $20, UNC       ; 无条件跳转
$10:

```



```

    TCLR    @VarB, #6    ; VarB 第 6 位清 0
$20:

```

TCLR loc16, #bit 测试并清 0 指定位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TCLR loc16, #bit	0101 0110 0000 1001 0000 BBBB LLLL LLLL	1	-	1

操作数: loc16 寻址方式

#bit 立即常数位索引从 0~15

描述: 测试“loc16”地址单元指定的位，然后对相同的位清 0:

```

TC = [loc16(bit)];
[loc16(bit)] = 0;

```

#bit 立即数直接对应指定位数。例如，若 #bit=0，访问第 0 位（最低位）；若 #bit=15，访问第 15 位（最高位）。

TCLR 执行一个读取-修改-写入操作。

标志与模式: N 若 (loc16 = @AX) 且 @AX 的第 15 位为 1，则负标志位 N 置位。

Z 若 (loc16 = @AX) 且 @AX 为 0，则零标志位 Z 置位。

TC 若测试位为 1，则 TC 置位；若测试位为 0，则 TC 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```

    ; if( VarA.Bit4 = 1 )
    ; VarB.Bit6 = 1;
    ; else
    ; VarB.Bit6 = 0;
TBIT    @VarA, #4    ; 测试 VarA 第 4 位
SB      $10, NTC      ; 若 TC = 0 则跳转
TSET    @VarB, #6     ; VarB 第 6 位置位
SB      $20, UNC      ; 无条件跳转
$10:
    TCLR    @VarB, #6    ; VarB 的第 6 位清 0
$20:

```

TEST ACC 测试累加器是否等于 0

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TEST ACC	1111 1111 0101 1000	×	-	1

操作数: ACC 累加器

描述: ACC 累加器和 0 比较，设置相应的状态标志位:

根据 (ACC - 0x00000000) 修改标志位;

标志与模式: N 若 ACC 的第 31 位为 1，则负标志位 N 置位；否则 N 清 0。

Z 若 ACC 为 0，则零标志位 Z 置位；否则 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 测试 ACC 的内容, 若为 0 则跳转:
TEST  ACC          ; 根据 (ACC - 0x00000000) 修改标志位
SB     Zero, EQ     ; 若为 0 则跳转
```

TRAP #VectorNumber

软件陷阱

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TRAP #VectorNumber	0000 0000 001C CCCC	×	-	8

操作数: Vector Number CPU 中断向量 0~31

描述: TRAP 指令控制程序转到指令指定向量对应的中断服务程序。它不影响中断标志位寄存器 (IFR) 或中断使能寄存器 (IER), 不考虑选择的中断在这些寄存器中是否有相应的位。TRAP 指令不受状态寄存器 ST1 的全局中断屏蔽位 (INTM) 的影响。它同样不受 IER 中的使能位或调试中断使能寄存器 (DBGIER) 的影响。一旦 TRAP 指令到达流水线的解码阶段, 直到 TRAP 指令执行完毕 (直到中断服务程序开始), 才会响应硬件中断。

下表表示中断向量与之相对应的 VectorNumber 操作数:

向量数	中断向量	向量数	中断向量
0	RESET	16	RTOSINT
1	INT1	17	Reserved
2	INT2	18	NMI
3	INT3	19	ILLEGAL
4	INT4	20	USER1
5	INT5	21	USER2
6	INT6	22	USER3
7	INT7	23	USER4
8	INT8	24	USER5
9	INT9	25	USER6
10	INT10	26	USER7
11	INT11	27	USER8
12	INT12	28	USER9
13	INT13	29	USER10
14	INT14	30	USER11
15	DLOGINT	31	USER12

操作的部分包括保存 16 位内核寄存器对到 SP 寄存器指向的堆栈。每个寄存器对用一个单独的 32 位操作指令保存。形成寄存器对的低字先保存 (保存到偶地址); 高字后保存 (保存到紧接着的奇地址)。例如, 所保存的第一个值是 T 寄存器和状态寄存器 ST0 的组

合 (T:ST0)。ST0 先保存, 然后是 T。

当外围中断扩展 (PIE) 使能时, 本指令不能用于向量 1~12。

注意: TRAP #0 指令不执行完全的复位过程。它只强制进入执行 RESET 中断向量相应的中断服务程序。

刷新流水线;

temp = PC + 1;

提取向量;

SP = SP + 1;

[SP] = T:ST0;

SP = SP + 2;

[SP] = AH:AL;

SP = SP + 2;

[SP] = PH:PL;

SP = SP + 2;

[SP] = AR1:AR0;

SP = SP + 2;

[SP] = DP:ST1;

SP = SP + 2;

[SP] = DBGSTAT:IER;

SP = SP + 2;

[SP] = temp;

INTM = 0; // 禁止 INT1-INT14, DLOGINT, RTOSINT

DBGM = 1; // 禁止调试事件

EALLOW = 0; // 禁止仿真寄存器的访问

LOOP = 0; // loop 标志位清 0

IDLESTAT = 0; // idle 标志位清 0

PC = 提取向量;

标志与模式: DBGM 设置 DBGM 位禁止调试事件。

INTM 设置 INTM 位禁止可屏蔽中断

EALLOW EALLOW 清 0 禁止保护的寄存器的访问

LOOP loop 标志位清 0

IDLESTAT idle 标志位清 0

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

TSET loc16,#16bit

测试并设置特殊位

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
TSET loc16,#16bit	0101 0110 0000 1101 0000 BBBB LLLL LLLL	1	-	1

操作数: loc16 寻址方式

#16bit 立即常数位索引从 0~15

描述: 测试“loc16”寻址地址单元中的指定位, 然后置位相同的位:

```
TC = [loc16(bit)];
[loc16(bit)] = 1;
```

#bit 立即数指定位数。例如，若 #bit=0，访问第 0 位（最低位）；若 #bit=15，访问第 15 位（最高位）。

TSET 执行一个读取-修改-写入操作。

标志与模式：N 若 (loc16 == @AX) 且 @AX 的第 15 位为 1，则负标志位 N 置位。

Z 若 (loc16 == @AX) 且 @AX 为 0，则零标志位 Z 置位。

TC 若测试位为 1，则 TC 置位；若测试位为 0，则 TC 清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
; if( VarA.Bit4 = 1 )
; VarB.Bit6 = 1;
; else
; VarB.Bit6 = 0;
TBIT    @VarA, #4          ; 测试 VarA 的第 4 位
SB      $10,NTC            ; 若 TC = 0 则跳转
TSET    @VarB, #6          ; 测试 VarB 的第 6 位并置位
SB      $20,UNC            ; 无条件跳转
$10:
TCLR    @VarB, #6          ; VarB 的第 6 位清 0
20:
```

UOUT *(PA),loc16

未保护的输出数据到 I/O 端口

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
UOUT *(PA), loc16	1011 0000 LLLL LLLL CCCC CCCC CCCC CCCC	1	Y	N+2

操作数：*(PA) I/O 空间存储器地址

loc16 寻址方式

描述：输出“loc16”地址单元的 16 位数到“*(PA)”指向的 I/O 端口：

```
IOspace[0x000:PA] = loc16;
```

I/O 空间限制在 64K 范围内 (0x0000~0xFFFF)。在外部接口 (XINTF) 上，若操作外部设备，I/O 片选 (XISn) 使能。I/O 地址出现在低 16 位地址线 XA(15:0) 上，高位地址线为 0。数据出现在低 16 位数据线 XD (15:0) 上。

注意：UOUT 操作不受流水线保护。因此，若一个 IN 指令紧跟在一个 UOUT 指令后，则 IN 会在 UOUT 前发生。为了保证操作的顺序，使用受流水线保护的 OUT 指令。I/O 空间范围与 C28x 芯片有关。详见数据手册。

标志与模式：无

重复性: 本指令可以重复。若操作跟在一个 RPT 指令后, 它执行 N+1 次。每次重复的 “* (PA)” I/O 空间地址先增加 1。

例:

```

; IRegA address = 0x0300;
; IRegB address = 0x0301;
; IRegC address = 0x0302;
; IRegA = 0x0000;
; IRegB = 0x0400;
; IRegC = VarA;
; if( IRegC = 0x2000 )
; IRegC = 0x0000;
IRegA .set      0x0300      ; 定义 IRegA 地址
IRegB .set      0x0301      ; 定义 IRegB 地址
IRegC .set      0x0302      ; 定义 IRegC 地址
MOV @AL, #0                ; AL = 0
UOUT  *(IRegA), @AL         ; Iospace[IRegA] = AL
MOV  @AL, #0x0400           ; AL = 0x0400
UOUT  *(IRegB), @AL         ; Iospace[IRegB] = AL
OUT   *(IRegC), @VarA       ; Iospace[IRegC] = VarA
IN    @AL, *(IRegC)         ; AL = Iospace[IRegC]
CMP   @AL, #0x2000          ; 根据 (AL - 0x2000) 的值设置标志位
SB    $10, NEQ              ; 若不相等则跳转
MOV   @AL, #0               ; AL = 0
UOUT  *(IRegC), @AL         ; Iospace[IRegC] = AL
$10:

```

XB *AL

与 C2 xLP 源代码兼容的直接跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XB *AL	0101 0110 0001 0100	1	-	7

操作数: *AL 使用寄存器 AL 间接寻址程序存储器, 可以访问程序空间范围高端 64K (0x3F0000~0x3FFFFFF)

描述: 无条件跳转, 用寄存器 AL 的内容装载 PC 的低 16 位, 强制 PC 的高 6 位为 0x3F:

PC = 0x3F:AL;

注意: 跳转指令只跳转到程序空间的高 64K 位置 (0x3F0000~0x3FFFFFF)

标志与模式: 无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 跳转到 Switch 值选择的开关表 SwitchTable 的子程序处
; 本例只能分配在程序空间高端 64K 工作:
SwitchTable:                ; Switch 地址表
.word Switch0                ; Switch0 地址

```



```

.word Switch1                ; Switch1 地址
.
.
.
.
MOVL    XAR2,#SwitchTable    ; XAR2 指针指向开关表
MOVZ    AR0,@Switch           ; AR0 = 开关表索引
MOV     AL,*+XAR2[AR0]        ; AL = 开关表[Switch]
XB      *AL                   ; 使用 AL 的间接跳转
SwitchReturn:
.
Switch0:                      ; 子程序 0:
.
.
XB      SwitchReturn,UNC      ; 无条件返回
Switch1:                      ; 子程序 1
.
.
XB      SwitchReturn,UNC      ; 无条件返回

```

XB pma,*,ARPN 用 ARP 修正与 C2 xLP 源代码兼容的跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XB pma,*,ARPN	0011 1110 0111 0nnn CCCC CCCC CCCC CCCC	1	-	4

操作数: pma 16 位立即数程序存储器地址, 可以访问程序空间范围的高端 64K (0x3F0000~0x3FFFFFF)

ARPN 3 位辅助寄存器指针 (ARP0~ARP7)

描述: 用 16 位立即数“pma”装载 PC 的低 16 位以改变 ARP 的无条件跳转位置, 强制 PC 的高 6 位为 0x3F。同样, 按“ARPN”操作数的指定改变辅助寄存器指针:

```
PC = 0x3F: pma;
```

$$\text{ARP} = n;$$

注意：跳转指令只跳转到程序空间的高端 64K 的位置（0x3F0000~0x3FFFFF）。

标志与模式：无

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例：

; 跳转到 SubA 并且设置 ARP。用 ARP 指针指向的内容装载 ACC 并返回到
; ARP 指针指向的位置。本例只能在程序空间高端 64K 工作:

XB SubA, *, ARP1 ; 跳转到 SubA, ARP 指向 XAR1

SubReturn:

SubA: ; 子程序 A:

MOVL ACC, *

XB SubReturn,UNC ; 无条件返回

XB pma,COND

与 C2 xLP 源代码兼容的跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XB pma, COND	0101 0110 1101 COND CCCC CCCC CCCC CCCC	1	—	7/4

操作数: pma 16 位立即数程序存储器地址, 可以访问程序空间范围的高端 64K (0x3F0000~0x3FFFFFF)

COND 条件代码

COND	格 式	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	大于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于, 进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述: 条件跳转。若指定的条件为真, 则用 16 位立即数 “pma” 装载 PC 的低 16 位, 并强制 PC 的高 6 位为 0x3F, 并执行跳转; 否则继续执行:

 If (COND = 真) PC(15:0) = pma;
 If (COND = 假) PC(15:0) = PC(15:0) + 2;
 PC(21:16) = 0x3F;

注意: 若 (COND = 真) 指令执行 7 个周期。
 若 (COND = 假) 指令执行 4 个周期。

标志与模式: V 若条件测试 V 标志位, 则 V 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

 ; 跳转到 Switch 值指定的开关表 SwitchTable 的子程序
 ; 本例只能在程序空间高端 64K 工作:

SwitchTable: ; Switch 地址表
 .word Switch0 ; Switch0 地址

```

.word Switch1                ; Switch1 地址
.
.
MOVL XAR2,#SwitchTable      ; XAR2 指针指向开关表
MOVZ AR0,@Switch            ; AR0 开关表索引
MOV AL,*+XAR2[AR0]          ; AL = 开关表[Switch]
XB *AL                      ; 使用 AL 的间接跳转
SwitchReturn:
.
Switch0:                    ; 子程序 0:
.
.
XB SwitchReturn,UNC         ; 无条件返回
Switch1:                    ; 子程序 1
.
.
XB SwitchReturn,UNC         ; 无条件返回

```

XBANZ pma,*ind{,ARPn}

与 C2 xLP 源代码兼容的 ARn 非零跳转

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XBANZ pma,*	0101 0110 0000 1100 CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*++	0101 0110 0000 1100 CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*--	0101 0110 0000 1011 CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*0++	0101 0110 0000 1110 CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*0--	0101 0110 0000 1111 CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma*,ARPn	0011 1110 0011 0nnn CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*++,ARPn	0011 1110 0011 1nnn CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*--,ARPn	0011 1110 0100 0nnn CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*0++,ARPn	0011 1110 0100 1nnn CCCC CCCC CCCC CCCC	1	—	4/2
XBANZ pma,*0--,ARPn	0011 1110 0101 0nnn CCCC CCCC CCCC CCCC	1	—	4/2

操作数: pma 16 位立即数程序存储器地址, 可以访问程序空间范围的高 64K (0x3F0000~0x3FFFFFF)

ARPn 3 位辅助寄存器指针 (ARP0~ARP7)

描述: 若当前辅助寄存器指针 (ARP) 指向的辅助寄存器的低 16 位不等于 0,

则用 16 位立即数“pma”值装载 PC 的低 16 位执行跳转，并强制 PC 的高 6 位为 0x3F。然后，将 ARP 指向的当前辅助寄存器按间接寻址模式修改。最后，若指定 ARP 指针值，它将指向一个新的辅助寄存器：

```
if( AR[ARP] != 0 )
    PC = 0x3F: pma
if(*+ + 间接模式) XAR[ARP] = XAR[ARP] + 1;
if(*- - 间接模式) XAR[ARP] = XAR[ARP] - 1;
if(*0+ + 间接模式) XAR[ARP] = XAR[ARP] + AR0;
if(*0- - 间接模式) XAR[ARP] = XAR[ARP] - AR0;
if(ARPN 指定的) ARPN = n;
```

注意：本指令只能控制程序转到程序空间的高端 64K 范围位置（0x3F0000~0x3FFFFFF）。这个操作的周期为：若跳转，则指令执行 4 个周期。若不跳转，则指令执行 2 个周期。

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
; 复制 Array1 的内容到 Array2:
; int32 Array1[N];
; int32 Array2[N];
; for(i = 0; i < N; i+ +)
; Array2[i] = Array1[i];
; 本例只能在程序空间高端 64K 工作:
MOVL    XAR2, #Array1      ; XAR2 指针指向 Array1
MOVL    XAR3, #Array2      ; XAR3 指针指向 Array2
MOV      @AR0, # (N-1)      ; 重复 loop 循环 N 次
NOP      *, ARP2            ; 指向 XAR2
SETC     AMODE              ; 全兼容 C2XLP 寻址方式

Loop:
MOVL     ACC, *+ +, ARP3     ; ACC = Array1[i], 指向 XAR3
MOVL     *+ +, ACC, ARP0     ; Array2[i] = ACC, 指向 XAR0
BANZ     Loop, *- -, ARP2    ; 若 AR[ARP] != 0, AR[ARP]- -,
                                ; 指向 XAR2, 跳到 Loop
```

XCALL *AL

与 C2 xLP 源代码兼容的函数调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XCALL *AL	0101 0110 0011 0100	1	-	7

操作数：*AL 使用寄存器 AL 间接寻址程序存储器，可以访问程序空间范围高 64K（0x3F0000~0x3FFFFFF）

描述：用 AL 中的目标地址间接调用。当前 PC 地址的低 16 位保存到软件堆栈。然后，PC 的低 16 位装载 AL 寄存器的内容，PC 的高 6 位装载 0x3F：

```
temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
```

C = 0x3F:AL;

注意：指令只能位于程序空间的高端 64K（0x3F0000~0x3FFFFFF）。从 XCALL 的调用返回，必须使用 XRETC 指令。

标志与模式： 无
重复性： 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```

; 由函数索引值选择的函数表中的函数。
; 本例只能在程序空间高端 64K 工作：
FuncTable:                                ; 函数地址表
    .word FuncA                            ; 函数 A 地址
    .word FuncB                            ; 函数 B 地址
    .
    .
    MOVL XAR2,#FuncTable                  ; XAR2 指针指向函数表 FuncTable
    MOVZ AR0,@FuncIndex                   ; AR0 = 函数表索引 FuncIndex
    MOV AL,*+XAR2[AR0]                    ; AL = 表[FuncIndex]
    XCALL *AL                             ; 使用 AL 的间接调用
    .
    .
FuncA:                                    ; 函数 A:
    .
    .
    XRETC UNC                             ; 无条件返回
FuncB:                                    ; 函数 B
    .
    .
    XRETC UNC                             ; 无条件返回
```

XCALL pma,*,ARPn 与 C2 xLP 源代码兼容的函数调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XCALL pma,*,ARPn	0011 1110 0110 1nnn CCCC CCCC CCCC CCCC	1	-	4

操作数： pma 16 位立即数程序存储器地址，可以访问程序空间范围的高 64K（0x3F0000~0x3FFFFFF）
ARPn 3 位辅助寄存器指针（ARP0~ARP7）
描述： 无条件调用。返回地址的低 16 位压入到软件堆栈中，PC 的低 16 位装载 16 位 “pma” 值，强制 PC 的高 6 位为 0x3F。然后，3 位 ARP 指针设置到 “ARPn” 值：

```

temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
PC = 0x3F:pma;
```


ARP = n;

注意：指令只能控制程序到程序空间范围的高 64K (0x3F0000~0x3FFFFFF)。从 XCALL 的调用返回，必须使用 XRETC 指令。

标志与模式：无

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

```
; 调用函数 FuncA 和设置 ARP。用 ARP 指向的指针装载 ACC
; 本例只能在程序空间高端 64K 工作：
XCALL  FuncA, *, ARP1    ;调用函数 FuncA，ARP 指向 XAR1
```

```
FuncA:                                ; 函数 FuncA
      MOVL  ACC, *              ; 用 XAR(ARP) 指向的内容装载 ACC
      XRETC  UNC                ; 无条件返回
```

XCALL pma,COND

与 C2 xLP 源代码兼容的函数调用

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XCALL pma,COND	0101 0110 1110 COND CCCC CCCC CCCC CCCC	1	—	7/4

操作数：pma 16 位立即数程序存储器地址，可以访问程序空间范围的高 64k (0x3F0000~0x3FFFFFF)

COND 条件代码

COND	格 式	描 述	测试标志位
0000	NEQ	不等于	Z = 0
0001	EQ	等于	Z = 1
0010	GT	大于	Z = 0 与 N = 0
0011	GEQ	大于或等于	N = 0
0100	LT	大于	N = 1
0101	LEQ	小于或等于	Z = 0 或 N = 1
0110	HI	更高	C = 1 与 Z = 0
0111	HIS	更高或等于，进位设置	C = 1
1000	LO, NC	更低或进位位清 0	C = 0
1001	LOS	更低或等于	C = 0 或 Z = 1
1010	NOV	无溢出	V = 0
1011	OV	溢出	V = 1
1100	NTC	测试位未置位	TC = 0
1101	TC	测试位置位	TC = 1
1110	NBIO	BIO 输出等于 0	BIO = 0
1111	UNC	无条件	—

描述: 条件调用。若条件为真，则返回地址的低 16 位压入软件堆栈中，PC 的低 16 位装载立即数“pma”值，强制 PC 的高 6 位为 0x3F；否则继续执行 XCALL 指令后的指令：

```
if(COND = 真)
{
    temp(21:0) = PC + 2;
    [SP] = temp(15:0);
    SP = SP + 1;
    PC = 0x3F:pma;
}
else
    PC = PC + 2;
```

注意: 指令只能适于程序空间的高端 64K (0x3F0000~0x3FFFFFF)。从 XCALL 的调用返回，必须使用 XRETC 指令。这个操作的周期是：若 (COND = 真) 指令执行 7 个周期。若 (COND = 假) 指令执行 4 个周期。

标志与模式: V 若执行了条件测试，则 V 标志位清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```
;若 VarA 不等于 0，调用函数 FuncA。本例只能在程序空间高 64K 工作
MOV AL,@VarA          ; 用 VarA 装载 AL
XCALL FuncA,NEQ        ; 若不等于 0，则调用函数 FuncA
.
.
FuncA:                  ; 函数 FuncA
.
.
XRETC UNC              ; 无条件返回
```

XMAC P,loc16,* (pma)

与 C2 xLP 源代码兼容的乘加

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XMAC P,loc16,* (pma)	1000 0100 LLLL LLLL CCCC CCCC CCCC CCCC	1	Y	N+2

操作数: P 乘积寄存器

loc16 寻址方式

立即数程序存储器地址，只访问程序空间的高 64K 范围 (0x3F0000~0x3FFFFFF)

描述: 先前的乘积 (保存在 P 寄存器) 按乘积移位方式 (PM) 指定的移位位数移位后并加到 ACC 累加器。接着，用“loc16”地址单元的内容装载 T 寄存器。最后，用“* (pma)”寻址程序存储器的有符号 16 位数乘以 T 寄存器的有符号 16 位数，32 位结果保存到 P 寄存器：

```

ACC = ACC + P << PM;
T = [loc16];
P = signed T × signed Prog[0x3F: pma];

```

当使用 MAC 指令格式时, C28x 强制 “*(pma)” 寻址方式指定的程序存储器地址的高 6 位为 0x3F。这限定程序地址空间的高 64K(0x3F0000~0x3FFFFFF)。对于 C28x, 存储器块映射含程序和数据空间(同一物理存储器), 因此 “*(pma)” 寻址方式可以用来访问落入其寻址范围内的数据空间变量。

- 标志与模式:**
- Z** 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。
 - N** 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。
 - C** 若产生进位, 则进位标志位 C 置位; 否则进位标志位 C 清 0。
 - V** 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。
 - OVC** 若禁止溢出模式, 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。
 - OVM** 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数(0x7FFFFFFF)或最小负数(0x80000000)。
 - PM** PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正(逻辑左移操作), 则最低位填 0。若乘积移位位数为负(算术右移操作), 则高位进行符号扩展。
- 重复性:** 本指令可以重复。若操作跟在一个 RPT 指令后, 它执行 N+1 次。Z、N、C 和 OVC 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。当每次重复时, 程序存储器地址增 1。

例:

```

; 使用 16 位乘计算乘积的和 sum:
; int16 X[N]                ; 数据信息
; int16 C[N]                ; 系数信息, 定位在高 64K
; sum = 0;
; for(i = 0; i < N; i++)
; sum = sum + (X[i] × C[i]) >> 5;
MOVL  XAR2, #X              ; XAR2 指针指向 X
SPM   -5                   ; 设置乘积右移 5 位
ZAPA                      ; ACC, P, OVC 清 0
RPT   #N-1                 ; 重复下一条指令 N 次
||XMAC P, *XAR2+, *, *C)    ; ACC = ACC + P >> 5,
                             ; P = *XAR2+ × *C+
ADDL  ACC, P << PM          ; 执行最终累加
MOVL  @sum, ACC             ; 保存最终结果到 sum

```

XMACD P, loc16, *(pma)

带数据移动与 C2 xLP 源代码兼容的乘加

语 法 选 项	操 作 码	OBJ 模式	RPT	CYC
XMACD P, loc16, *(pma)	1010 0100 LLLL LLLL CCCC CCCC CCCC CCCC	1	Y	N+2

- 操作数:** P 乘积寄存器
loc16 寻址方式
- 描述:** 对于这个操作, 寄存器寻址方式不能使用: @ARn, @AH, @AL, @PH, @PL, @SP, @T 方式。会产生一个非法的指令陷阱中断。
*(pma) 立即数程序存储器地址, 只访问程序空间的高 64K 范围 (0x3F0000~0x3FFFFFF)
XMACD 指令函数和 XMAC 的格式一样, 只是多一个数据移动。按乘积移位方式 (PM) 指定的移位位数对 P 寄存器移位, 后加到 ACC 累加器。接着, 用 “loc16” 地址单元的内容装载 T 寄存器。然后, 用 “*(pma)” 寻址程序存储器的有符号 16 位数乘以 T 寄存器的有符号 16 位数, 32 位结果保存到 P 寄存器。最后, T 寄存器的内容保存到 “loc16” 寻址方式指向的下一个高位存储器地址单元中:
- ```
ACC = ACC + P << PM;
T = [loc16];
P = signed T × signed Prog[0x3F:pma];
[loc16 + 1] = T;
```
- 当使用 MAC 指令的这种格式时, C28x 强制 “\*(pma)” 寻址方式指定的程序存储器地址的高 6 位为 0x3F。这限定到程序地址空间的高 64K (0x3F0000~0x3FFFFFF)。对于 C28x, 存储器块映射含程序和数据空间 (同一物理存储器), 因此 “\*(pma)” 寻址方式可以用来访问落入其寻址范围内的数据空间变量。
- 标志与模式:** Z 若 ACC 值为 0, 则零标志位 Z 置位; 否则 Z 清 0。  
N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则 N 清 0。  
C 若产生进位, 则进位标志位 C 置位; 否则进位标志位 C 清 0。  
V 若产生溢出, 则溢出标志位 V 置位; 否则不影响 V。  
OVC 若禁止溢出模式, 操作产生正溢出, 则计算器值增加; 若操作产生负溢出, 则计算器值减少。  
OVM 若溢出模式位置位, 操作发生溢出, 则 ACC 的值为最大正数 (0x7FFFFFFF) 或最小负数 (0x80000000)。  
PM PM 位的值为乘积寄存器的输出时移位操作方式。若乘积移位位数为正 (逻辑左移操作), 则最低位填 0。若乘积移位位数为负 (算术右移操作), 则高位进行符号扩展。
- 重复性:** 本指令可以重复。若操作跟在一个 RPT 指令后, 它执行 N+1 次。Z、N、C 和 OVC 的状态为最终的结果。若中间产生溢出, 则 V 标志位置位。当每次重复时, 程序存储器地址增加 1。

**例:**

```
; 使用 16 位乘计算 FIR 滤波器;
; int16 X[N] ; 数据信息
; int16 C[N] ; 系数信息, 定位在高 64K
```



```

; sum = X[N-1] × C[0];
; for(i = 1; i < N; i++)
; {
; sum = sum + (X[N-1-i] × C[i]) >> 5;
; X[N-i] = X[N-1-i];
; }
; X[1] = X[0];
MOVL XAR2, #X+N ; XAR2 指针指向 X 数组的最后单元
SPM -5 ; 设置乘积右移 5 位
ZAPA ; ACC, P, OVC 清 0
XMAC P, *- -XAR2, *(C) ; ACC = 0, P = X[N-1] × C[0]
RPT #N-2 ; 重复下一条指令 N-1 次
||XMACD P, *- -XAR2, *(C+1) ; ACC = ACC + P >> 5,
; P = X[N-1-i] × C[i],
; i++
MOV *+XAR2[2], T ; X[1] = X[0]
ADDL ACC, P << PM ; 执行最终累加
MOVL @sum, ACC ; 保存最终结果到 sum

```

**XOR ACC, loc16**

按位“异或”

| 语 法 选 项        | 操 作 码               | OBJ 模式 | RPT | CYC |
|----------------|---------------------|--------|-----|-----|
| XOR ACC, loc16 | 1011 0111 LLLL LLLL | 1      | Y   | N+1 |

操作数: ACC 累加器  
loc16 寻址方式

描述: 用“loc16”地址单元中的数 0 扩展后与 ACC 累加器执行位“异或”操作。结果保存到累加器 ACC:

ACC = ACC XOR 0:[loc16];

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若操作跟在一个 RPT 指令后, 它执行 N+1 次。Z 和负标志位 N 的状态为最终的结果。

例:

```

; 计算 32 位数 VarA = VarA XOR 0: VarB
MOVL ACC, @VarA ; 用 VarA 的内容装载 ACC
XOR ACC, @VarB ; 用 0: VarB 的内容“异或”ACC
MOVL @VarA, ACC ; 结果保存到 VarA

```

**XOR ACC, #16bit << #0..16**

按位“异或”

| 语 法 选 项                 | 操 作 码                                      | OBJ 模式 | RPT | CYC |
|-------------------------|--------------------------------------------|--------|-----|-----|
| XOR ACC, #16bit<<#0..15 | 0011 1110 0010 SHFT<br>CCCC CCCC CCCC CCCC | 1      | -   | 1   |
| XOR ACC, #16bit<<#16    | 0101 0110 0100 1110<br>CCCC CCCC CCCC CCCC | 1      | -   | 1   |



操作数: ACC 累加器  
 #16bit 16 位立即常数  
 #0..16 移位位数 (没有指定值时, 默认为 “<<#0”)

描述: 用指定的 16 位无符号数按指定的移位位数左移, 并对 16 位无符号常数进行 0 扩展, 最低位填 0 后与 ACC 累加器位 “异或” 操作。结果保存到 ACC 累加器:

ACC = ACC XOR (0:16bit << 移位位数);

标志与模式: N 若 ACC 的第 31 位为 1, 则负标志位 N 置位; 否则清 0。  
 Z 若 ACC 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 计算 32 位数 VarA = VarA XOR 0x08000000
MOVL ACC,@VarA ; 用 VarA 的内容装载 ACC
XOR ACC,#0x8000 << 12 ; 用 0x08000000 “异或” ACC
MOVL @VarA,ACC ; 结果保存到 VarA
```

#### XOR AX, loc16

按位 “异或”

| 语 法 选 项       | 操 作 码               | OBJ 模式 | RPT | CYC |
|---------------|---------------------|--------|-----|-----|
| XOR AX, loc16 | 0111 000A LLLL LLLL | ×      | -   | 1   |

操作数: AX 累加器高 (AH) 或累加器低 (AL) 寄存器  
 loc16 寻址方式

描述: 对指定的 AX 寄存器 (AH 或 AL) 和 “loc16” 地址单元的内容执行 “异或” 操作。结果保存在指定的 AX 寄存器。

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。  
 Z 若 AX 的值为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```
; 将 VarA 和 VarB 的内容 “异或” 后, 保存到 VarC:
MOV AL,@VarA ; 用 VarA 的内容装载 AL
XOR AL,@VarB ; 用 VarB 的内容 “异或” AL
MOV @VarC,AL ; 结果保存到 VarC
```

#### XOR loc16,AX

按位 “异或”

| 语 法 选 项      | 操 作 码               | OBJ 模式 | RPT | CYC |
|--------------|---------------------|--------|-----|-----|
| XOR loc16,AX | 1111 001A LLLL LLLL | ×      | -   | 1   |

操作数: loc16 寻址方式  
 AX 累加器高 (AH) 或累加器低 (AL) 寄存器

描述: 对 “loc16” 地址单元中的 16 位数与指定的 AX 寄存器 (AH 或 AL) 执

行“异或”操作。结果保存在“loc16”寻址地址单元中：

```
[loc16] = [loc16] XOR AX;
```

指令执行一个读取-修改-写入操作。

标志与模式：N 若[loc16]的第 15 位为 1，则负标志位 N 置位；否则清 0。

Z 若[loc16] = 0，则零标志位 Z 置位；否则清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; 用 VarB OR VarA 的内容，保存到 VarB:

```
MOV AL, @VarA ; 用 VarA 的内容装载 AL
```

```
XOR @VarB, AL ; VarB = VarB “异或” AL
```

### XOR loc16, #16bit

按位“异或”

| 语 法 选 项           | 操 作 码                                      | OBJ 模式 | RPT | CYC |
|-------------------|--------------------------------------------|--------|-----|-----|
| XOR loc16, #16bit | 0001 1100 LLLL LLLL<br>CCCC CCCC CCCC CCCC | ×      | -   | 1   |

操作数：loc16 寻址方式

#16bit 16 位立即常数

描述：将“loc16”地址单元中的内容与 16 位立即常数进行“异或”操作。结果保存到“loc16”指向的地址单元中：

```
[loc16] = [loc16] XOR 16bit;
```

标志与模式：N 若[loc16]的第 15 位为 1，则负标志位 N 置位；否则 N 清 0。

Z 若[loc16] = 0，则零标志位 Z 置位；否则 Z 清 0。

重复性：本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例：

; 置位 VarA 的第 2 位和第 14 位

```
; VarA = VarA XOR #(1 << 2 | 1 << 14)
```

```
XOR @VarA, #(1 << 2 | 1 << 14) ; 置位 VarA 的第 2 位和第 14 位
```

### XORB AX, #8bit

8 位数按位“异或”

| 语 法 选 项        | 操 作 码               | OBJ 模式 | RPT | CYC |
|----------------|---------------------|--------|-----|-----|
| XORB AX, #8bit | 1111 000A CCCC CCCC | ×      | -   | 1   |

操作数：AX 累加器高 (AH) 或累加器低 (AL) 寄存器

#8bit 8 位立即常数

描述：8 位无符号立即常数 0 扩展后与指定的 AX 寄存器执行“异或”操作。结果保存到 AX 寄存器：

```
AX = AX XOR 0:8bit;
```

标志与模式: N 若 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若[loc16] = 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

; 置位 VarA 的第 7 位并将结果保存到 VarB:

```
MOV AL, @VarA ; 用 VarA 的内容装载 AL
XORB AL, #0x80 ; 用 0x0080 “异或” AL 的内容
MOV @VarB, AL ; 结果保存到 VarB
```

### XPREAD loc16,\* (pma)

与 C2 xLP 源代码兼容的读程序存储器

| 语 法 选 项              | 操 作 码                                      | OBJ 模式 | RPT | CYC |
|----------------------|--------------------------------------------|--------|-----|-----|
| XPREAD loc16,* (pma) | 0101 0110 0011 1100<br>0000 0000 LLLL LLLL | 1      | Y   | N+2 |

操作数: loc16 寻址方式

\* (pma) 程序存储器寻址, 只访问程序空间的高 64K 范围 (0x3F0000~0x3FFFFFF)

描述: 用 “\* (pma)” 寻址方式指向的程序存储器地址单元中的 16 位数装载落入其寻址到 “loc16” 寻址的数据存储器单元中:

[loc16] = Prog[0x3F:pma];

当使用 XPREAD 指令的这种格式时, C28x 强制 “\* (pma)” 寻址方式指定的程序存储器地址的高 6 位为 0x3F。这限定到程序地址空间的高 64K (0x3F0000~0x3FFFFFF)。对于 C28x, 存储器块映射含程序和数据空间 (同一物理存储器), 因此 “\* (pma)” 寻址方式可以用来访问落入其寻址范围内的数据空间变量。

标志与模式: N 若 (loc16 = @AX) 并且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则 N 清 0。

Z 若 (loc16 = @AX) 并且 AX 为 0, 则零标志位 Z 置位; 否则 Z 清 0。

重复性: 本指令可以重复。若跟在一个 RPT 指令后, 则将执行 N+1 次。当每次重复时, “\* (pma)” 程序存储器地址复制到一个内部影子寄存器, 每次重复时地址增加 1。

例:

```
; 复制 Array1 的内容到 Array2:
; int16 Array1[N]; // 定位在程序空间的高 64K
; int16 Array2[N]; // 定位在数据空间
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i];
MOVL XAR2, #Array2 ; XAR2 指针指向 Array2
RPT #(N-1) ; 重复性下一条指令 N 次
```

```
||XPREAD *XAR2+ +,*(Array1) ; Array2[i] = Array1[i],
; i+ +
```

**XPREAD loc16,\*AL**

与 C2 xLP 源代码兼容的读程序存储器

| 语 法 选 项          | 操 作 码                                     | OBJ 模式 | RPT | CYC |
|------------------|-------------------------------------------|--------|-----|-----|
| XPREAD loc16,*AL | 0101 0110 0011 110<br>0000 0000 LLLL LLLL | 1      | Y   | N+4 |

操作数: loc16 寻址方式

\*AL 使用 AL 寄存器的间接程序存储器寻址, 只能访问程序空间范围的高 64K (0x3F0000~0x3FFFFFF)

描述: 用“\*AL”寻址方式指向的程序存储器地址单元中的 16 位数装载到“loc16”寻址方式指向的 16 位数据存储器地址单元中:

```
[loc16] = Prog[0x3F:AL];
```

当使用 XPREAD 指令时, C28x 强制“\*AL”寻址方式指定的程序存储器地址的高 6 位为 0x3F。这限定到程序地址空间的高 64K (0x3F0000~0x3FFFFFF)。对于 C28x, 存储器块映射含程序和数据空间 (同一物理存储器), 因此“\*AL”寻址方式可以用来访问程序地址范围内的数据变量。

标志与模式: N 若 (loc16 = @AX) 且 AX 的第 15 位为 1, 则负标志位 N 置位; 否则清 0。

Z 若 (loc16 = @AX) 且 AX 为 0, 则零标志位 Z 置位; 否则清 0。

重复性: 本指令可以重复。若操作跟在一个 RPT 指令后, 则将执行 N+1 次。当每次重复时, “\*AL”程序存储器地址复制到一个内部影子寄存器, 每次重复时地址增加 1。

例:

```
; 复制 Array1 的内容到 Array2:
; int16 Array1[N]; // 定位到程序空间的高 64K
; int16 Array2[N]; // 定位到数据空间
; for(i = 0; i < N; i+ +)
; Array2[i] = Array1[i];
MOV @AL, #Array1 ; AL 指针指向 Array1
MOVL XAR2, #Array2 ; XAR2 指针指向 Array2
RPT #(N-1) ; 重复下一条指令 N 次
||XPREAD *XAR2+ +, *AL ; Array2[i] = Array1[i],
; i+ +
```

**XPWRITE \*AL,loc16**

与 C2 xLP 源代码兼容的写程序存储器

| 语 法 选 项           | 操 作 码                                      | OBJ 模式 | RPT | CYC |
|-------------------|--------------------------------------------|--------|-----|-----|
| XPWRITE *AL,loc16 | 0101 0110 0011 1101<br>0000 0000 LLLL LLLL | 1      | Y   | N+4 |



操作数: \*AL 使用 AL 寄存器的间接程序存储器寻址, 只能访问程序空间范围的高 64K (0x3F0000~0x3FFFFFF)

loc16 寻址方式

描述: 用“loc16”地址单元中的 16 位数装载“\*AL”指向的 16 位程序存储器地址单元:

```
Prog[0x3F:AL] = [loc16];
```

当使用 XPWRITE 指令时, C28x 强制“\*AL”寻址方式指定的程序存储器地址的高 6 位为 0x3F。这限定到程序地址空间的高 64K (0x3F0000~0x3FFFFFF)。对于 C28x, 存储器块映射含程序和数据空间 (同一物理存储器), 因此“\*AL”寻址方式可以用来访问落入其寻址范围内的数据空间变量。

标志与模式: 无

重复性: 本指令可以重复。若操作跟在一个 RPT 指令后, 则将执行 N+1 次。当每次重复时, “\*AL”程序存储器地址复制到一个内部影子寄存器, 每次重复时地址增加 1。

例:

```
; 复制 Array1 的内容到 Array2:
; int16 Array1[N]; // 定位到数据空间
; int16 Array2[N]; // 定位到程序空间的高 64K
; for(i = 0; i < N; i++)
; Array2[i] = Array1[i];
MOVL XAR2, #Array1 ; XAR2 指针指向 Array1
MOV @AL, #Array2 ; AL 指针指向 Array2
RPT #(N-1) ; 重复下一条指令 N 次
||XPWRITE *AL, *XAR2++ ; Array2[i] = Array1[i],
; i++
```

## XRET

与 C2 xLP 源代码兼容的返回

| 语 法 选 项 | 操 作 码               | OBJ 模式 | RPT | CYC |
|---------|---------------------|--------|-----|-----|
| XRET    | 0101 0110 1111 1111 | 1      | -   | 7   |

操作数: 无

描述: 条件返回。若指定的条件为真, 16 位数从堆栈弹出到 PC 的低 16 位, 而 PC 的高 6 位强制为 0x3F; 否则, 继续执行 XRET 指令后的指令:

```
if(COND = 真)
SP = SP - 1;
PC = 0x3F:[SP];
```

**注意:** 指令只能控制程序到程序空间范围的高 64K (0x3F0000~0x3FFFFFF)。从 XCALL 的调用返回, 必须使用 XRET 指令。

标志与模式: V 若条件测试 V 标志位, V 清 0。



**重复性:** 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

**例:**

; 若 VarA 不等于 0, 从函数 A 返回, 否则设置 VarB 为 0 并返回。  
; 本例只能在程序空间的高 64K 工作

XCALL FuncA ;调用函数 FuncA

FuncA: ;函数 FuncA

.

.

.

.

MOV AL, @VarA ; 用 VarA 的内容装载 AL

XRET NEQ ; 若 VarA 不等于 0 返回

MOV @VarA, #0 ; 保存 0 到 VarB

XRETC UNC ; 无条件返回

## XRETC COND

与 C2 xLP 源代码兼容的条件返回

| 语 法 选 项    | 操 作 码               | OBJ 模式 | RPT | CYC |
|------------|---------------------|--------|-----|-----|
| XRETC COND | 0101 0110 1111 COND | 1      | -   | 4/7 |

**操作数:** COND 条件代码

| COND | 格 式    | 描 述         | 测试标志位         |
|------|--------|-------------|---------------|
| 0000 | NEQ    | 不等于         | Z = 0         |
| 0001 | EQ     | 等于          | Z = 1         |
| 0010 | GT     | 大于          | Z = 0 与 N = 0 |
| 0011 | GEQ    | 大于或等于       | N = 0         |
| 0100 | LT     | 大于          | N = 1         |
| 0101 | LEQ    | 小于或等于       | Z = 0 或 N = 1 |
| 0110 | HI     | 更高          | C = 1 与 Z = 0 |
| 0111 | HIS    | 更高或等于, 进位设置 | C = 1         |
| 1000 | LO, NC | 更低或进位位清 0   | C = 0         |
| 1001 | LOS    | 更低或等于       | C = 0 或 Z = 1 |
| 1010 | NOV    | 无溢出         | V = 0         |
| 1011 | OV     | 溢出          | V = 1         |
| 1100 | NTC    | 测试位未置位      | TC = 0        |
| 1101 | TC     | 测试位置位       | TC = 1        |
| 1110 | NBIO   | BIO 输出等于 0  | BIO = 0       |
| 1111 | UNC    | 无条件         | -             |

**描述:** 条件返回。若指定的条件为真, 16 位数从堆栈弹出到 PC 的低 16 位, 而 PC 的高 6 位强制为 0x3F; 否则, 继续执行 XRETC 指令执行后的指令:

```
if(COND = 真)
```

```
{
```

```

 SP = SP - 1;
 PC = 0x3F:[SP];
 }
 else
 PC = PC + 1;

```

**注意：**指令只能适用于程序空间的高 64K (0x3F0000~0x3FFFFFF)。从 XCALL 的调用返回，必须使用 XRET 指令。这个操作的周期为：若 (COND = 真)，则指令执行 7 个周期。若 (COND = 假)，则指令执行 4 个周期。

**标志与模式：** V 若条件测试，则 V 标志位清 0。

**重复性：** 本指令不可以重复。若指令跟在 RPT 指令之后，它复位重复计数器 RPTC，并且只执行一次。

**例：**

； 若 VarA 不等于 0，从函数 FuncA 返回，否则设置 VarB 为 0 并返回。  
； 本例只能在程序空间的高 64K 工作

```

XCALL FuncA ; 调用函数 FuncA
.
FuncA: ; 函数 FuncA
.
.
.
.
MOV AL, @VarA ; 用 VarA 的内容装载 AL
XRETC NEQ ; 若 VarA 不等于 0，则返回
MOV @VarA, #0 ; 保存 0 到 VarB
XRETC UNC ; 无条件返回

```

**ZALR ACC,loc16**

**AL 清 0 并装载 AH**

| 语 法 选 项        | 操 作 码                                      | OBJ 模式 | RPT | CYC |
|----------------|--------------------------------------------|--------|-----|-----|
| ZALR ACC,loc16 | 0101 0110 0001 0011<br>0000 0000 LLLL LLLL | 1      | -   | 1   |

**操作数：** ACC 累加器  
loc16 寻址方式

**描述：** 用 0x8000 装载累加器低 (AL)，用 “loc16” 地址单元中的 16 位数装载累加器高 (AH)。

```

AH = [loc16];
AL = 0x8000;

```

**标志与模式：** N 若 ACC 的第 15 位为 1，则负标志位 N 置位；否则清 0。

Z 若 ACC = 0，则零标志位 Z 置位；否则清 0。

**重复性：** 本指令不可以重复。若指令跟在 RPT 指令之后，它将复位重复计数器 RPTC，并且只执行一次。

例:

```

; 计算 Y = round(M×X << 1 + B << 16) ; Y, M, X, B 都是 Q15 格式的数
SPM +1 ; 设置乘积左移 1 位
MOV T, @M ; T = M (Q15)
MPY P, T, @X ; P = M × X (Q30)
ZALR ACC, @B ; ACC = B << 16 + 0x8000 (Q31)
ADDL ACC, P << PM ; 移位后的 P (Q31 格式) 加到 ACC
MOV @Y, AH ; 保存 AH 到 Y (Q15 格式)

```

**ZAP OVC**

溢出计数器 (OVC) 清 0

| 语 法 选 项 | 操 作 码               | OBJ 模式 | RPT | CYC |
|---------|---------------------|--------|-----|-----|
| ZAP OVC | 0101 0110 0101 1100 | 1      | -   | 1   |

操作数: OVC 状态寄存器 0 (ST0) 的溢出计数器

描述: 状态寄存器 0 (ST0) 的溢出计数器 (OVC) 清 0

标志与模式: OVC 6 位溢出计数器 (OVC) 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 计算 VarD = sat (VarA + VarB + VarC)
ZAP OVC ; 溢出计数器清 0
MOVL ACC, @VarA ; ACC = VarA
ADDL ACC, @VarB ; ACC = ACC + VarB
ADDL ACC, @VarC ; ACC = ACC + VarC
SAT ACC ; 若 OVC != 0, 使为饱和值
MOVL @VarD, ACC ; 保存结果到 VarD

```

**ZAPA**

ACC 和 P 寄存器清 0

| 语 法 选 项 | 操 作 码               | OBJ 模式 | RPT | CYC |
|---------|---------------------|--------|-----|-----|
| ZAPA    | 0101 0110 0011 0011 | 1      | -   | 1   |

操作数: 无

描述: ACC 和 P 寄存器以及溢出计数器 (OVC) 清 0:

ACC = 0;

P = 0;

OVC = 0;

标志与模式: N 负标志位 N 置位。

Z 零标志位 Z 清 0。

重复性: 本指令不可以重复。若指令跟在 RPT 指令之后, 它将复位重复计数器 RPTC, 并且只执行一次。

例:

```

; 使用 32 位乘计算乘积的 sum, 保留高位结果
; int32 X[N]; // 数据信息
; int32 C[N]; // 系数信息 (定位在 4M 低端)

```

```

; int32 sum = 0;
; for(i = 0; i < N; i++)
; sum = sum + ((X[i] * C[i]) >> 32) >> 5;
MOVL XAR2, #X ; XAR2 指针指向 X
MOVL XAR7, #C ; XAR7 指针指向 C
SPM -5 ; 设置乘积右移 5 位
ZAPA ; ACC, P, OVC 清 0
RPT #(N-1) ; 重复下一条指令次
||QMACL P, *XAR2+, *, *XAR7+ + ; ACC = ACC + P >> 5,
; P = (X[i] * C[i]) >> 32
; i++
ADDL ACC, P << PM ; 执行最终累加
MOVL @sum, ACC ; 保存最终结果到 sum

```





## 附录 A 通用目标文件格式

TMS320C28x™ 汇编器和链接器以通用目标文件格式 (COFF) 创建目标文件。COFF 是目标文件格式的一种，该格式具有与 AT&T 公司基于 UNIX 系统开发的目标文件格式相同的名字。因为这种格式鼓励模块化编程，并提供了更为有力和灵活的管理代码段和目标系统存储器的方法，所以采用这种格式。

段 (Section) 是基本的 COFF 概念。第 2 章通用目标文件格式，详细讨论了 COFF 段。如果用户理解了段的操作，那么用户将能更加有效地使用汇编语言工具。

附录 A 包括关于 COFF 目标文件结构的详细技术资料。这些资料许多地方与 C 编译器产生的符号调试资料有关。本附录的目的是提供有关 COFF 目标文件的内部格式的辅助资料。

### A.1 COFF 文件结构

COFF 目标文件的元素说明了文件的段以及符号调试信息。这些元素包括：

- 文件头部
- 可选的头部信息



- 段头部表
- 每一个初始化段的原始数据
- 每一个初始化段的重定位信息
- 每一个初始化段的行号入口
- 符号表
- 字符串表

汇编器和链接器产生具有相同 COFF 结构的目标文件。然而，最后一次链接的程序通常不包含重定位入口。图 A.1 所示为目标文件结构。

图 A.2 所示为一个典型的包含了 3 个默认段 .text、.data、.bss 以及一个命名段的目标文件。在默认情况下，软件工具按下列顺序把段放置在目标文件内：.text、.data 初始化命名段，.bss 未初始化命名段。尽管未初始化段有段的头部，但它们没有原始数据、重定位信息或行号入口。这是因为 .bss 和 .unsect 伪指令只简单地未初始化数据保留空间，未初始化段不包含实际代码。

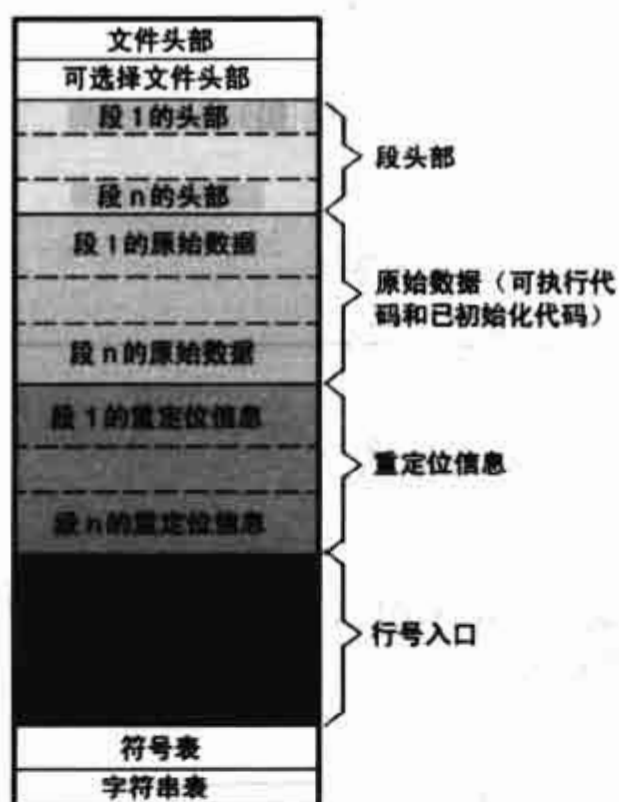


图 A.1 目标文件结构



图 A.2 COFF 目标文件例

## A.2 文件头部的结构

文件头部包含 22 个字节的信息，它说明了目标文件的一般格式。表 A.1 所示为

TMS320C28x COFF 文件头部的结构。

表 A.1 文件头部内容

| 字节号   | 类 型   | 描 述                                                       |
|-------|-------|-----------------------------------------------------------|
| 0~1   | 无符号短型 | ID 版本；说明 COFF 文件结构的版本                                     |
| 2~3   | 无符号短型 | 段头部的个数                                                    |
| 4~7   | 整型    | 时间和日期标记，指示文件创建的时间                                         |
| 8~11  | 整型    | 文件指针；包含符号表的起始地址                                           |
| 12~15 | 整型    | 符号表中入口的个数                                                 |
| 16~17 | 无符号短型 | 可选头部中的字节数。此域为 0 或 28；如果是 0，那么没有可选的文件头部                    |
| 18~19 | 无符号短型 | 标志（见表 A.2）                                                |
| 20~21 | 无符号短型 | 目标 ID；特定标识符（magic number）（00ebh）指示该文件能够在 TMS320C28x 系统中执行 |

表 A.2 列出了可出现在文件头部第 18 和 19 字节的标志。这些标志的值及其组合可同时设置（例如，若字节 18 和 19 设置为 0003h，则 F\_RELFLG 和 F\_EXEC 同时设置）。

表 A.2 文件头部标记（字节 18 和 19）

| 助 记 符    | 标 记   | 描 述                     |
|----------|-------|-------------------------|
| F_RELFLG | 0001h | 重定位信息已从文件中除去            |
| F_EXEC   | 0002h | 可执行文件（包含未确定的外部引用）       |
| F_LNNO   | 0004h | 行号已从文件中除去               |
| F_LSYM   | 0008h | 局部符号已从文件中除去             |
| F_LITTLE | 0100h | 目标是一个 little-endian 型设备 |
| F_BIG    | 0200h | 目标是一个 big-endian 型设备    |

### A.3 可选的文件头部格式

链接器创建可选的文件头部，并利用它在装载时完成重定位。部分链接的文件不包含可选的文件头部。表 A.3 所示为可选的文件头部的格式。

表 A.3 可选的文件头部内容

| 字节号   | 类 型 | 描 述                 |
|-------|-----|---------------------|
| 0~1   | 短型  | 可选的文件头部特定标识符（0108h） |
| 2~3   | 短型  | 版本标记                |
| 4~7   | 整型  | 可执行代码大小（字节）         |
| 8~11  | 整型  | 初始化数据大小（字节）         |
| 12~15 | 整型  | 未初始化数据大小（字节）        |
| 16~19 | 整型  | 入口                  |
| 20~23 | 整型  | 可执行代码起始地址           |
| 24~27 | 整型  | 初始化数据起始地址           |

## A.4 段头部格式

COFF 文件包含段头部表，它定义在目标文件中每一个段在何处起始。每一个段具有自己的段头部。表 A.4 所示为每一个段头部的结构。

表 A.4 段头部内容

| 字节号   | 类 型   | 描 述                                                            |
|-------|-------|----------------------------------------------------------------|
| 0~7   | 字符    | 这部分包括如下之一：<br>□ 8 个字符的段名，不满则用 0 填补<br>□ 如果符号名比 8 个字符长，指针进入字符串表 |
| 8~11  | 整型    | 段的物理地址                                                         |
| 12~15 | 整型    | 段的虚拟地址                                                         |
| 16~19 | 整型    | 以字计的段大小                                                        |
| 20~23 | 整型    | 指向原始数据的指针                                                      |
| 24~27 | 整型    | 指向重定位入口的指针                                                     |
| 28~31 | 整型    | 指向行号入口的指针                                                      |
| 32~35 | 无符号整型 | 重定位入口数                                                         |
| 36~39 | 无符号整型 | 行号入口数                                                          |
| 40~43 | 无符号整型 | 标志（见表 A.5）                                                     |
| 44~45 | 无符号整型 | 保留                                                             |
| 46~47 | 无符号整型 | 存储器页号                                                          |

表 A.5 列出了可出现在段头部的 40 到 43 字节的标志。

表 A.5 段头部标志（40~43 字节）

| 助 记 符       | 标 记       | 描 述                              |
|-------------|-----------|----------------------------------|
| STYP_REG    | 00000000h | 常规段（已分配的、重定位的、装载的）               |
| STYP_DSECT  | 00000001h | 虚段（重定位的、未分配的、未装载的）               |
| STYP_NOLOAD | 00000002h | 未装载段（分配的、重定位的、未装载的）              |
| STYP_GROUP  | 00000004h | 段群（由几个输入段形成）                     |
| STYP_PAD    | 00000008h | 填补段（装载的、未分配的、未装载的）               |
| STYP_COPY   | 00000010h | 复制段（重定位的、装载的，但未全装载；正常处理重定位和行号入口） |
| STYP_TEXT   | 00000020h | 包含可执行代码段                         |
| STYP_DATA   | 00000040h | 包含初始化数据段                         |
| STYP_BSS    | 00000080h | 包含未初始化数据段                        |
| STYP_CLINK  | 00004000h | 要求条件链接段                          |

注意：术语“已装载”指该段的原始数据出现在目标文件中。

表 A.5 中列出的标志可以组合。例如，如果标志字被设置为 024h，那么 TYP\_GROUP 和 STYP\_TEXT 都被设置。

图 A.3 显示了段头部中的指针如何指向目标文件中与 .text 段有关的元素。

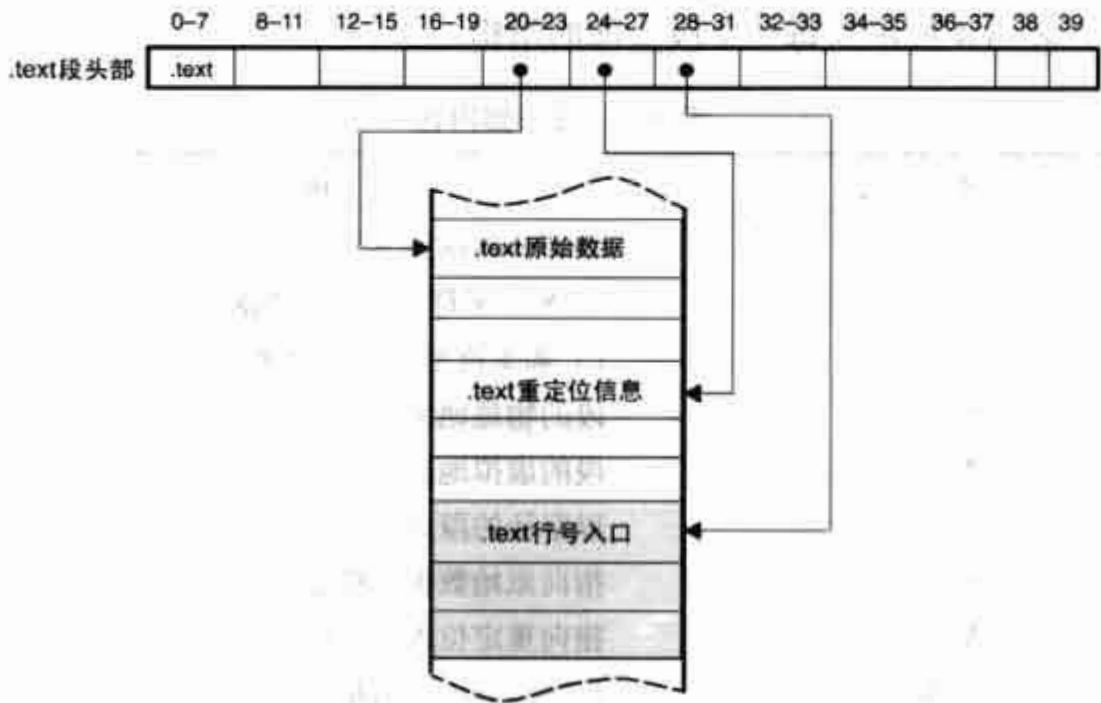


图 A.3 指向 .text 段的段头指针

如图 A.2 所示，未初始化段（由 .bss 和 .usect 伪指令创建的）与此格式不同。虽然未初始化段具有段头部，但它们没有原始数据、可重定位信息和行号信息。它们在目标文件中不占据实际空间。因此对于未初始化段、可重定位入口数、行号入口数以及文件指针都为 0。未初始化段的段头部简单地告诉链接器它应当在存储器映像中为变量保留多大空间。

### A.5 构造可重定位信息

对于每一个可重定位的引用，COFF 目标文件具有一个可重定位入口。汇编器自动产生重定位入口。当链接器在读取每一个输入段时，它读取重定位入口并执行重定位。重定位入口决定在每一个输入段内如何处理引用。

COFF 文件可重定位信息入口使用 10 个字节格式，见表 A.6。说明如下。

表 A.6 重定位入口内容

| 字节号 | 类型    | 描述              |
|-----|-------|-----------------|
| 0~3 | 整型    | 引用的虚地址          |
| 4~5 | 无符号短型 | 符号表索引 (0~65535) |
| 6~7 | 无符号短型 | 保留              |
| 8~9 | 无符号短型 | 重定位类型 (见表 A.7)  |

- 虚地址是重定位之前当前段内符号的地址；它指定重定位必须发生在何处（这是在目标代码中必须被修正的地址）。

以下是产生一个重定位入口的代码例：

```

2 .global x
3 000000ffffl b x
 0000010000

```

在上例中，可重定位的虚地址是 0001。

- 符号表索引是被引用符号的索引。在上例中，可包含表中 X 的索引。重定位的量（amount）是段中符号的当前地址与其汇编时地址之间的差。必须用与被引用符号同样的量来重定位可重定位域。例如，在重定位之前，X 具有数值 0h，假设 X 被重定位到 2000h 地址。这就是重定位的量（2000h-0h），从而通过地址为 1 的重定位域加上 2000h 来修正。

如果用户知道符号定义在哪个段内，那么用户就可以决定符号的重定位地址。例如，如果 X 被定义在 .data 中，而 .data 按 2000h 重定位，那么 X 即按 2000h 重定位。

当重定位入口中的符号表索引是 -1h (0FFFFh) 时，那么这就被称为内部重定位。在这种情况下，重定位量就是当前段被重定位的量。

- 重定位类型规定被修正域的大小并说明应当怎样计算修正值。类型域取决于产生可重定位引用的寻址方式。在前面的例中，引用符号 X 的实际地址是 22 位的地址，但是一个与 16 位成比例的地址被编码到目标代码中。因此，重定位类型是 R\_C27PCR16。表 A.7 列出了重定位类型。

表 A.7 重定位类型（字节 8 和 9）

| 助 记 符       | 标 记   | 重定位类型           |
|-------------|-------|-----------------|
| R_ABS       | 0000h | 无重定位            |
| R_RELBYTE   | 000Fh | 对符号地址的 8 位直接引用  |
| R_RELWORD   | 0010h | 对符号地址的 16 位直接引用 |
| R_RELLONG   | 0011h | 对符号地址的 32 位直接引用 |
| R_PARTLS6   | 05Dh  | 22 地址的 6 位偏移量   |
| R_PARTLS7   | 28h   | 16 地址的 7 位偏移量   |
| R_PARTMID10 | 05Eh  | 22 位地址的中间 10 位  |
| R_REL22     | 05Fh  | 对符号地址的 22 位直接引用 |
| R_PARTMS6   | 060h  | 22 位地址的高 6 位    |
| R_PARTS16   | 061h  | 22 位地址的低 16 位   |
| R_C27PCR16  | 062h  | 16 位 PC         |
| R_C27PCR8   | 063h  | 8 位 PC          |
| R_C27PTR    | 064h  | 22 位指针          |
| R_C27HI16   | 065h  | 地址的高 16 位       |
| R_C27LOPTR  | 066h  | 低 64K 指针        |
| R_C27NWORD  | 067h  | 取消 16 位重定位      |
| R_C27NBYTE  | 068h  | 取消 8 位重定位       |



## A.6 行号表结构

目标文件包含行号入口表，它主要用于符号调试。当 C 编译器产生一些汇编语言代码行时，它创建行号入口，此行号入口把这些行映射回到产生它们的 C 源代码的原始行。每一个行号入口包含 6 个字节的信息。表 A.8 所示为行号入口格式。

表 A.8 行号入口格式

| 字节号 | 类型    | 描述                                                                                                                                                                                              |
|-----|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0~3 | 整型    | 此入口可以具有下列两个值之一： <ul style="list-style-type: none"> <li><input type="checkbox"/> 如果它是行号项入口块中的第一个入口，那么它指向符号表中符号的入口</li> <li><input type="checkbox"/> 如果它不是块中的第一个入口，那么它是字节 4~5 所指行的物理地址</li> </ul> |
| 4~5 | 无符号短型 | 此入口可以具有下列两个值之一： <ul style="list-style-type: none"> <li><input type="checkbox"/> 如果值为 0，那么它指向函数入口的第一行</li> <li><input type="checkbox"/> 如果值不是 0，那么就是 C 源代码行的行号</li> </ul>                        |

图 A.4 显示了行号入口怎样被组合到块中。

如图 A.4 所示，每一项可分为两种：

- ☐ 对于函数的第一行，字节 0~3 指向符号名或在符号表中的函数，字节 4~5 包含 0，它指示块的开始。
- ☐ 对于函数的其余行，字节 0~3 表示物理地址（由 C 语言程序语句行产生的字数），字节 4~5 表示原始 C 语言程序语句的地址。行号入口表可包含许多这种块。

图 A.5 为 XYZ 函数的行号入口。如图所示，函数名在符号表中作为一个符号编码。在 XYZ 块中行号入口的第一部分指向在符号表中的函数名。假设 C 语言程序中的函数包含 3 行代码。此代码与第一行有关，第一行位于相对于函数起始单元偏移量为 0 的字节处。第二行代码从偏移量为 2 的地方开始，与 3 行有关的代码是从函数开始的 6 个字节。



图 A.4 行号块

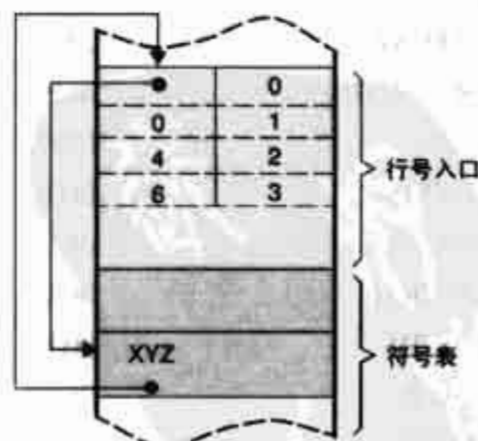


图 A.5 行号入口

注意：XYZ 的符号表入口具有一个指向行号块开始处的域。

因为行号并不是经常需要的，所以链接器提供选项（-s），它可以去除目标文件中的行号信息，这就提供了更紧凑的目标模块（关于-s 选项，详见第 7.4.15 节去除符号信息-s 选项）。

A.7 符号表结构和内容

符号表中符号的次序是非常重要的，它们按图 A.6 所示的顺序出现。

静态变量指 C 语言程序中定义的任何函数之外具有静态存储类型的符号。如果用户有几个模块使用了相同名字的符号，那么令它们为静态将把每一个符号的作用域限制到定义它们的模块中（这就消除多重定义冲突）。

符号表中的每个符号包括符号的如下内容：

- ☐ 名字（或指向字符串表的指针）
- ☐ 类型
- ☐ 数值
- ☐ 对其进行定义的段
- ☐ 存储类型
- ☐ 基本类型（整型、字符等）
- ☐ 派生类型（阵列、结构等）
- ☐ 维数
- ☐ 定义符号的源代码的行号

段名也在符号表中定义。所有符号入口，无论存储类型和基本类型是什么，在符号表中都具有相同的格式。每一个符号表入口包含 18 个字节的信息，如表 A.9 所示。每一个符号也可以有 18 个字节的辅助入口；表 A.10 中所列的专用符号总是有辅助项。某些符号可能不具备表 A.9 中列出的所有特征。如果特定的域未设置，那么该域将设置为空。

|            |
|------------|
| 文件名 1      |
| 函数 1       |
| 函数 1 的局部符号 |
| 函数 2       |
| 函数 2 的局部符号 |
| ⋮          |
| 文件名 2      |
| 函数 1       |
| 函数 1 的局部符号 |
| ⋮          |
| 静态变量       |
| ⋮          |
| 已定义的全局符号   |
| 未定义的全局符号   |

图 A.6 符号表内容

表 A.9 符号表入口内容

| 字节号   | 类 型   | 描 述                                                                                                         |
|-------|-------|-------------------------------------------------------------------------------------------------------------|
| 0~7   | 字符    | 此域包括下列之一：<br><input type="checkbox"/> 8 个字符的符号名，用空字符填充<br><input type="checkbox"/> 如果符号名大于 8 个字符，那么指针指向字符串表 |
| 8~11  | 整型    | 符号值，由存储器类型决定                                                                                                |
| 12~13 | 短型    | 符号的段编号                                                                                                      |
| 14~15 | 无符号短型 | 指定的基本类型和派生类型                                                                                                |
| 16    | 字符    | 符号的存储类型                                                                                                     |
| 17    | 符号    | 辅助项数（总是为 0 或 1）                                                                                             |

## A.7.1 专用符号

符号表包含某些由编译器、汇编器以及链接器产生的专用符号。每一个专用符号包括原始符号表信息和辅助入口。表 A.10 列出了这些专用符号。

这些符号中有几个符号是成对出现的：

- 符号.bb/eb 表示块的开始和结束。
- 符号.bf/ef 表示函数的开始和结束。
- 符号.nfake/.eos 命名和定义未命名的结构、联合以及枚举的范围。符号.eos 也和命名的结构、联合以及枚举成对出现。

表 A.10 符号表中的专用符号

| 符 号                 | 描 述                    |
|---------------------|------------------------|
| .text               | .text 段的地址             |
| .data               | .data 段的地址             |
| .bss                | .bss 段的地址              |
| .bb                 | 块的起始地址                 |
| .eb                 | 块的结束地址                 |
| .bf                 | 函数的起始地址                |
| ef                  | 函数的结束地址                |
| .target             | 指向由函数返回的结构或联合的指针       |
| .nfake <sup>①</sup> | 结构、联合或枚举的虚标记名          |
| .eos                | 结构、联合或枚举的结束            |
| etext               | 在.text 输出段结束之后的下一个可用地址 |
| edata               | 在.data 输出段结束之后的下一个可用地址 |
| end                 | 在.bss 输出段结束之后的下一个可用地址  |

<sup>①</sup>当结构、联合和枚举没有标记名时，编译器为它分配名字以便它能够输入到符号表中。这些名字具有.nfake 形式，其中 n 是整数。编译器从 0 开始对这些符号名编号。

### 1. 符号和块

在 C 语言程序中，块是一个用花括弧开始和结束的复合语句。块总包含符号。在符号表中任何特殊块的符号定义被组合在一起并由.bb/eb 专用符号定界。在 C 语言中，块可以嵌套，因此，它们的符号表入口也可相应地被嵌套。图 A.7 表示在符号表中符号块怎样分组。

### 2. 符号和函数

在符号表中函数的符号定义作为由.bf/ef 专用符号定界的组出现。函数名符号表入口位于.bf 专用符号之前。图 A.8 表示函数的符号表入口的格式。

如果函数返回结构或联合，那么专用符号.target 的符号表入口将出现在函数名入口和.bf 专用符号之间，如图 A.9 所示。



图 A.7 符号块

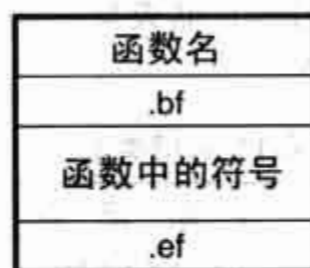


图 A.8 函数的符号



图 A.9 返回结构或联合的函数符号

## A.7.2 符号名格式

符号表入口的首 8 个字节表示符号名：

- 如果符号名等于或小于 8 个字符，那么此域具有字符类型。符号名后由空字符（null）填充并存储在 0~7 字节中。
- 如果符号名大于 8 个字符，那么此域被当作两个整型数处理。整个符号名存储在字符串表中。字节 0~3 包含 0，字节 4~7 是字符串表中的偏移量。

## A.7.3 字符串表结构

长于 8 个字符的符号名存储在字符串表中。符号表入口中通常包含符号名的域，指向字符串表中符号名的指针。符号名被连续地存储在字符串表中，由空（null）字符结束。字符串表的前 4 个字节包含以字节计的字符串表的大小，因此字符串表中的偏移量大于或等于 4。

图 A.10 为包含两个符号名 Adaptive-Filter 和 Fourier-Transform 的字符串表，在字符串表中，Adaptive-Filter 的索引为 4，Fourier-Transform 的索引为 20。

38 个字节

| 4 个字节 |      |     |      |
|-------|------|-----|------|
| 'A'   | 'd'  | 'a' | 'p'  |
| 't'   | 'i'  | 'v' | 'e'  |
| '.'   | 'F'  | 'i' | 'i'  |
| 't'   | 'e'  | 'r' | '\0' |
| 'F'   | 'o'  | 'u' | 'r'  |
| 'i'   | 'e'  | 'r' | '.'  |
| 'T'   | 'r'  | 'a' | 'n'  |
| 's'   | 't'  | 'o' | 'r'  |
| 'm'   | '\0' |     |      |

图 A.10 字符串表格式



## A.7.4 存储类型

符号表入口中第 16 个字节指示符号的存储类型。存储类型指 C 编译器用以访问符号的方法。表 A.11 列出了有效的存储类型。

表 A.11 存储类型

| 助 记 符    | 值  | 存 储 类 型 | 助 记 符     | 值   | 存 储 类 型                  |
|----------|----|---------|-----------|-----|--------------------------|
| C_NULL   | 0  | 无存储类型   | C_USTATIC | 14  | 未定义静态                    |
| C_AUTO   | 1  | 自动变量    | C_ENTAG   | 15  | 枚举标记                     |
| C_EXT    | 2  | 外部定义    | C_MOE     | 16  | 枚举成员                     |
| C_STAT   | 3  | 静态      | C_REGPARM | 17  | 寄存器参数                    |
| C_REG    | 3  | 寄存器变量   | C_FIELD   | 18  | 位域                       |
| C_EXTREF | 5  | 外部引用    | C_UEXT    | 19  | 不确定的外部定义                 |
| C_LABEL  | 6  | 标号      | C_STATLAB | 20  | 静态装载时间标号                 |
| C_ULABEL | 7  | 未定义标号   | C_EXTLAB  | 21  | 外部装载时间标号                 |
| C_MOS    | 8  | 结构成员    | C_BLOCK   | 100 | 块的开始或结束；仅用于专用符号.bb 和.eb  |
| C_ARG    | 9  | 函数自变量   | C_FC      | 101 | 函数的开始或结束；仅用于专用符号.bf 和.ef |
| C_STRTAG | 10 | 结构标记    | C_EOS     | 102 | 结构的结束；仅用于专用符号.eos        |
| C_MOU    | 11 | 联合的成员   | C_FILE    | 103 | 文件名；仅用于文件名符号             |
| C_UNTAG  | 12 | 联合标记    | C_LINE    | 104 | 仅应用于引用程序                 |
| C_TPDEF  | 13 | 定义类型    |           |     |                          |

某些专用符号受某些存储类型的限制。表 A.12 列出了这些符号以及它们的类型。

表 A.12 专用符号及其类型

| 专 用 符 号 | 限制它的存储类型 | 专 用 符 号 | 限制它的存储类型 |
|---------|----------|---------|----------|
| .bb     | C_BLOCK  | .eos    | C_EOS    |
| .eb     | C_BLOCK  | .text   | C_STAT   |
| .bf     | C_FCN    | .data   | C_STAT   |
| .ef     | C_FCN    | .bss    | C_STAT   |

## A.7.5 符号值

符号表入口的字节 8~11 指示符号的值。符号值取决于符号的存储类型。表 A.13 列出了存储类型和相关的数值。

表 A.13 符号值和存储类型

| 存 储 类 型 | 值 的 说 明   | 存 储 类 型 | 值 的 说 明 |
|---------|-----------|---------|---------|
| C_AUTO  | 以位计的栈的偏移量 | C_UNTAG | 0       |
| C_EXT   | C 的重定位地址  | TPDEF   | 0       |



续表

| 存储类型     | 值的说明     | 存储类型      | 值的说明   |
|----------|----------|-----------|--------|
| C_STAT   | 可重定位地址   | C_ENTAG   | 0      |
| C_REG    | 寄存器数     | C_MOE     | 枚举值    |
| C_LABEL  | 可重定位地址   | C_REGPARM | 寄存器数   |
| C_MOS    | 以位计的偏移量  | C_FIELD   | 位位移    |
| C_ARG    | 以位计的栈偏移量 | C_BLOCK   | 可重定位地址 |
| C_STRTAG | 0        | C_FCN     | 可重定位地址 |
| C_MOU    | 以位计的偏移量  | C_FILE    | 0      |

可重定位符号的值是它的虚地址。当链接器重定位段时，可重定位符号的值作相应的改变。

### A.7.6 段编号

符号表入口的字节 12~13 表示定义符号的段编号。表 A.14 列出了这些编号以及它们指示的段。

表 A.14 段编号

| 助记符     | 段号       | 描述              |
|---------|----------|-----------------|
| N_DEBUG | -2       | 专用符号的调试符号       |
| N_ABS   | -1       | 绝对符号            |
| N_UNDEF | 0        | 未定义外部符号         |
| N_SCNUM | 1        | .text 段（典型）     |
| N_SCNUM | 2        | .data 段（典型）     |
| N_SCNUM | 3        | .bss 段（典型）      |
| N_SCNUM | 4-32 767 | 按命名段遇到的次序命名段的段号 |

如果没有 .text、.data 或 .bss 段，那么命名段的编号从 1 开始。

如果符号具有段编号 0、-1 或 -2，那么它不在段中定义。段号 -2 表示符号化的调试符，包括结构、联合、枚举标记名、类型定义以及文件名。段号 -1 表示符号有值但未重定义。段号 0 表示可重定义的外部符号，但它未在当前文件中定义。

### A.7.7 类型项

符号表入口的字节 14~15 定义符号的类型。每一个符号都具有一个基本类型和 1~6 个派生类型。

以下是此 16 位类型入口的格式：

| Size<br>(in bits): | Derived<br>type<br>6 | Derived<br>type<br>5 | Derived<br>type<br>4 | Derived<br>type<br>3 | Derived<br>type<br>2 | Derived<br>type<br>1 | Basic<br>type |
|--------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|---------------|
|                    | 2                    | 2                    | 2                    | 2                    | 2                    | 2                    | 4             |

类型域的 0~3 位是基本类型。表 A.15 列出了有效的基本类型。

表 A.15 基本类型

| 助 记 符     | 值  | 类 型      |
|-----------|----|----------|
| T_VOID    | 0  | 无效类型     |
| T_SCHAR1  | 1  | 字符型（显符号） |
| T_CHAR    | 2  | 字符型（隐符号） |
| T_SHORT   | 3  | 短整型      |
| T_INT     | 4  | 整型       |
| T_LONG    | 5  | 长整型      |
| T_FLOAT   | 6  | 浮点       |
| T_DOUBLE  | 7  | 双精度浮点    |
| T_STRUCT  | 8  | 结构       |
| T_UNION   | 9  | 联合       |
| T_ENUM    | 10 | 枚举       |
| T_LDOUBLE | 11 | 长型双精度浮点  |
| T_UCHAR   | 12 | 无符号字符型   |
| T_USHORT  | 13 | 无符号短整型   |
| T_UINT    | 14 | 无符号整型    |
| T_ULONG   | 15 | 无符号长整型   |

类型域的位 4~15 被安排为 6 个 2 位的域，可以表示 1 至 6 个派生类型。表 A.16 列出了可能的派生类型。

表 A.16 派生类型

| 助 记 符  | 值 | 类 型   |
|--------|---|-------|
| DT_NON | 0 | 无派生类型 |
| DT_PTR | 1 | 指针    |
| DT_FCN | 2 | 函数    |
| DT_ARY | 3 | 数组    |

例如，具有 0000 0000 1101 0011<sub>2</sub> 类型入口的符号就具有几个派生类型。此入口表示符号是指向短型数指针的数组。

### A.7.8 辅助入口

每一个符号表入口可以有 1 个辅助入口或没有辅助入口。辅助符号表入口包含与符号表入口相同的字节数（18 个字节），但是辅助入口取决于符号类型以及存储类型。表 A.17 概括了这些关系。

表 A.17 辅助符号表入口格式

| 名 字                | 存 储 类 型          | 类 型     |                  | 辅 助 格 式                    |
|--------------------|------------------|---------|------------------|----------------------------|
|                    |                  | 派生类型 1  | 基本类型             |                            |
| .text, .data, .bss | C_STAT           | DT_NON  | T_VOID           | 段 (见表 A.18)                |
| tagname            | C_STRTAG         | DT_NON  | T_STRUCT         | 标记名 (见表 A.19)              |
|                    | C_UNTAG          | T_UNION |                  |                            |
|                    | C_ENTAG          | T_ENUM  |                  |                            |
| .eos               | C_EOS            | DT_NON  | T_VOID           | 结束结构 (见表 A.19)             |
| fcname             | C_EXT            | DT_FCN  | 任何               | 函数 (见表 A.21)               |
|                    | C_STAT           |         |                  |                            |
| arrname            | 见注释 <sup>①</sup> | DT_ARY  | 见注释 <sup>②</sup> | 数组 (见表 A.22)               |
| .bb, .eb           | C_BLOCK          | DT_NON  | T_VOID           | 块的开始和结束 (见表 A.23 和表 A.24)  |
| .bf, .ef           | C_FCN            | DT_NON  | T_VOID           | 函数的开始和结束 (见表 A.23 和表 A.24) |
| 与结构、联合或枚举有关的名字     | 见注释 <sup>①</sup> | DT_PTR  | T_STRUCT         | 与结构、联合或枚举有关的               |
|                    |                  | DT_ARR  | T_UNION          | 名字 (见表 A.15)               |
|                    |                  | DT_NON  | T_ENUM           |                            |

注释: ① C\_AUTO, C\_STAT, C\_MOS, C\_MOU, C\_TPDEF, C\_EXT

② 除 T\_VOID 之外的任何类型

在表 A.17 中, tagname 可指任何符号名 (包括专用符号 *nfake*) ; fcname 和 arrname 也可指任何的符号名。一般情况下, tagname 指的是结构, fcname 指的是函数, 而 arrname 指的是数组。

满足表 A.17 中多个条件的符号在其辅助入口中应当具有联合格式。不满足上述任一条件的符号应当没有辅助入口。

## 1. 段

表 A.18 列出了段辅助表入口的格式。

表 A.18 段辅助表入口的格式

| 字 节 号 | 类 型    | 描 述       |
|-------|--------|-----------|
| 0~3   | 整型     | 段的长度      |
| 4~5   | 无符号短整型 | 重定位入口号    |
| 6~7   | 无符号短整型 | 行号入口号     |
| 8~17  | —      | 未用 (填充 0) |

## 2. 标记名

表 A.19 列出了标记名辅助表入口的格式。

表 A.19 标记名辅助表入口的格式

| 字节号   | 类型 | 描述              |
|-------|----|-----------------|
| 0~3   | —  | 未用（填充 0）        |
| 4~7   | 整型 | 结构、联合或枚举大小      |
| 8~11  | —  | 未用（填充 0）        |
| 12~15 | 整型 | 在此函数之外的下一个入口的索引 |
| 16~17 | —  | 未用（填充 0）        |

### 3. 结构结束

表 A.20 列出了结构结束辅助表入口的格式。

表 A.20 结构结束辅助表入口的格式

| 字节号  | 类型 | 描述         |
|------|----|------------|
| 0~3  | 整型 | 标记索引       |
| 4~7  | 整型 | 结构、联合或枚举大小 |
| 8~17 | —  | 未用（填充 0）   |

### 4. 函数

表 A.21 列出了函数辅助表入口的格式。

表 A.21 函数辅助表入口的格式

| 字节号   | 类型 | 描述              |
|-------|----|-----------------|
| 0~3   | 整型 | 标记索引            |
| 4~7   | 整型 | 函数大小（按位）        |
| 8~11  | 整型 | 指向行号的文件指针       |
| 12~15 | 整型 | 除此函数之外的下一个入口的索引 |
| 16~17 | —  | 未用（填充 0）        |

### 5. 数组

表 A.22 列出了数组辅助表入口的格式。

表 A.22 数组辅助表入口的格式

| 字节号   | 类型     | 描述       |
|-------|--------|----------|
| 0~3   | 整型     | 标记索引     |
| 4~7   | 整型     | 数组大小     |
| 8~9   | 无符号短整型 | 一维       |
| 10~11 | 无符号短整型 | 二维       |
| 12~13 | 无符号短整型 | 三维       |
| 14~15 | 无符号短整型 | 四维       |
| 16~17 | —      | 未用（填充 0） |

## 6. 块和函数的结束

表 A.23 列出了块和函数结束辅助表入口的格式。

表 A.23 块和函数结束辅助表入口的格式

| 字 节 号 | 类 型    | 描 述      |
|-------|--------|----------|
| 0~3   | —      | 未用（填充 0） |
| 4~5   | 无符号短整型 | C 语言程序行号 |
| 6~17  | —      | 未用（填充 0） |

## 7. 块和函数的开始

表 A.24 列出了块和函数开始辅助表入口的格式。

表 A.24 块和函数开始辅助表入口的格式

| 字 节 号 | 类 型    | 描 述          |
|-------|--------|--------------|
| 0~3   | 整型     | 寄存器保存屏蔽      |
| 4~5   | 无符号短整型 | 块开始 C 语言程序行号 |
| 6~7   | 无符号短整型 | 函数行入口的数目     |
| 8~11  | 整型     | 函数的局部结构的大小   |
| 12~15 | 整型     | 此块之后下一个入口的序列 |
| 16~17 | —      | 未用（填充 0）     |

## 8. 与结构、联合和枚举有关的名字

表 A.25 列出了与结构、联合和枚举有关的名字辅助表入口的格式。

表 A.25 与结构、联合和枚举有关的名字辅助表入口的格式

| 字 节 号 | 类 型 | 描 述         |
|-------|-----|-------------|
| 0~3   | 整型  | 标记索引        |
| 4~7   | 整型  | 结构、联合或枚举的大小 |
| 8~17  | —   | 未用（填充 0）    |





## 附录 B 符号调试伪指令

汇编器支持几条 TMS320C28x C/C++编译器符号调试的伪指令：

- `.sym` 伪指令定义全局变量、局部变量或函数。该伪指令的几个参数允许用户把各种调试信息与符号或函数联系起来。
- `.stag`、`.etag` 和 `.utag` 伪指令分别定义结构、枚举及联合。`.member` 伪指令规定结构、枚举以及联合的成员。`.eos` 伪指令结束结构、枚举或联合的定义。
- `.func` 和 `.endfuncs` 伪指令规定 C 语言函数的开始和结束。
- `.block` 和 `.endblock` 伪指令规定 C 语言块的边界。
- `.file` 伪指令在符号表中定义识别当前源程序文件名的符号。
- `.line` 伪指令识别 C 语言程序语句的行号。

这些符号调试伪指令通常不列在编译器创建的汇编语言文件中。如果用户想要列出它们且还要保留汇编语言文件，那么可以使用 `-g` 和 `-k` 选项调用编译器命令解释程序，如下所示：

```
c12000 -v28 -gk input file
```

本附录包含按字母目录排列的符号调试伪指令。除 `.file` 伪指令之外，每一个伪指令都包含一个 C 语言程序例以及产生的汇编语言代码。

语法:

```
.block [beginning line number]
.endblock [ending line number]
```

描述: `.block` 和 `.endblock` 伪指令规定 C 语言程序块的开始和结束。行号 (*line numbers*) 是可选项, 它们规定在源程序文件中定义块的位置。

块定义可以嵌套。汇编器可以检测出不正确的块嵌套。

例: 以下是定义 C 语言块的代码和汇编语言代码举例。

C 源代码:

```
main()
{
 int i = 10;
 {
 int y = i + 3;
 foo(y);
 }
}
```

产生的汇编语言代码:

```
_main:
 .sym _i, -1, 4, 1, 16
 .sym _y, -2, 4, 1, 16
 ADDB SP, #2
 .line 3
 MOVB AL, #10 ;|5|
 MOV *-SP[1], AL ;|5|
 .block 7
 .line 5
 ADDB AL, #3 ;|7|
 MOV *-SP[2], AL ;|7|
 .line 6
 LC #_foo ;|8|
 ; call occurs [#_foo] :|8|
 .endblock 9
 SUBB SP, #2
 LRET
 ; return occurs
 .endfunc 11, 000000000h, 2
```

语法:

```
.file "filename"
```

描述: `.file` 伪指令允许调试器把存储器中的存储单元地址映射回到 C 语言程序中的行。

*filename* 是包含 C 语言程序文件的文件名。文件名可以任意长。

在汇编代码中用户也可以使用 `.file` 伪指令以提供文件名可以改善程序的可读性。

例: 名为 `text.c` 的文件包括 C 语言程序, 它产生此伪指令。

```
.file "text.c"
```

语法:

```
.func [beginning line number]
.endfunc [ending line number]
```

描述: .func和.endfunc伪指令规定C语言函数的开始和结束。行号 (*line numbers*) 是可选项, 它们规定源文件中函数的位置。函数定义不能嵌套。

例: 下面是定义 C 语言的函数以及产生的汇编语言代码的例。

C 源代码

```
power(x, n) /* 函数开始 */
{
 int x,n;
 {
 int i, p;
 p = 1;
 for (i =1; i <= n; ++i)
 p = p *x;
 return p; /* 函数结束 */
 }
}
```

产生的汇编语言代码:

```
FP .set XAR2
 .file "b_4.c"
 ; ac2000 -@/var/tmp/caaa003aB
 .sect ".text"
 .global _power
 .sym _power,_power, 36, 2, 0
 .func 1
_power:
 .line 2
 ; * AL 赋与_x
 .sym _x,0, 4, 17, 16
 ; * AH 赋与_n
 .sym _n,1, 4, 17, 16
 .sym _x,-1, 4, 1, 16
 .sym _n,-2, 4, 1, 16
 .sym _i,-3, 4, 1, 16
 .sym _p,-4, 4, 1, 16
 ADDB SP,#4
 MOV *-SP[1],AL ; |2|
 MOV *-SP[2],AH ; |2|
 .line 4
 MOVB AL,#1 ; |4|
 MOV *-SP[4],AL ; |4|
 .line 5
 MOV *-SP[3],AL ; |5|
 MOV AL,*-SP[2] ; |5|
 CMP AL,*-SP[3] ; |5|
```

```

 B L2,LT ; |5|
 ; 跳转; |5|

L1:
 .line 6
 MOV T,*-SP[1] ; |6|
 INC *-SP[3] ; |6|
 MPY ACC,T,*-SP[4] ; |6|
 MOV *-SP[4],AL ; |6|
 MOV AL,*-SP[2] ; |6|
 CMP AL,*-SP[3] ; |6|
 B L1,GEQ ; |6|
 ; 跳转; |6|

L2:
 .line 7
 MOV AL,*-SP[4] ; |7|
 .line 8
 SUBB SP,#4 ; |7|

LRETR
 ; 返回

 .endfunc 8,000000000h,4

```

## 语法:

**.line line number [, address]**

**描述:** .line 伪指令在目标文件中创建行号入口。在符号调试中, 行号入口用于把目标代码中的地址与产生它们的源代码中的行联系在一起。

.line 伪指令有两个操作数:

- **line number** 表示产生部分代码的 C 语言程序的行。行号相对于当前函数起始行。此参数为必要参数。
- **address** 是一个表达式, 它是和行号相联系的地址。这是一个可选参数; 如果用户不规定地址, 那么汇编器将使用当前 SPC 的值。

**例:** .line 伪指令后随汇编语言代码, 它由所指定的 C 语言程序代码行产生。例如, 假设下面的 C 语言程序代码行是原始 C 语言程序中的第 4~6 行, 那么第 5 行产生的汇编语言源程序如下所示:

C 源代码:

```

for (i = 0; i <= 5; ++i)
 sum = sum + i;
return sum;

```

产生的汇编语言代码:

```

_main:
 .sym _i,-1,4,1,16
 .sym _sum,-2,4,1,16
 ADDB SP,#2
 .line 3
 MOV *-SP[2],#0 ;|5|
 .line 5

```

```

MOV *-SP[1],#0 ;|7|
MOV AL,*-SP[1] ;|7|
CMPB AL,#5 ;|7|
B L3,GT ;|7|
 ;跳转;|7|
L2:
 .line 6
MOV AL,*-SP[1] ;|8|
ADD *-SP[2],AL ;|8|
 .line 5
INC *-SP[1] ;|7|
MOV AL,*-SP[1] ;|7|
CMPB AL,#5
B L2,LEQ ;|7|
;branch occurs ;|7|
L3:
 .line 7
MOV AL,*-SP[2] ;|9|
 .line 8
SUBB SP,#2 ;|10|
LRET
 ;返回
 .endfunc 10,000000000h,2

```

语法:

**`.member name, value [, type, storage class, size, tag, dims]`**

描述: `.member` 伪指令定义结构、联合或枚举的成员。它仅在结构、联合或枚举的定义中有效。

- *name* 是放在符号表中成员的名。名的前 128 个字符有效。
- *value* 是与成员有关的值。可接受任何有效的表达式（绝对地址或可重定位地址）。
- *type* 是成员的 C 类型。“附录 A 通用目标文件格式”包括关于 C 类型的详细介绍。
- *storage class* 是成员的 C 存储类型，“附录 A 通用目标文件格式”包括关于 C 存储类型的详细介绍。
- *size* 是包含此成员所需的存储器位数。
- *tag* 是属于成员的类型或结构的名。此名必须在前面由 `.stag`、`.etag` 或 `utag` 伪指令声明。
- *dims* 可以是 1~4 个由逗号隔开的表达式；这些表达式说明的是成员的维数。
- 参数的次序是重要的。*name* 和 *value* 是必须的参数。所有其它参数可以省略或为空（相邻的逗号表示是空项）。这就允许用户跳过个别参数并规定在表中其后出现的参数。被省略的操作数或空操作数被假设为空（`null`）值。

例: 下面是一个 C 语言的结构定义和相应的汇编语言代码的例。

C 源代码:  
 struct doc



```

{
 char title;
 char group;
 int job_number;
}doc_info;

```

产生的汇编语言代码:

```

FP .set AR2
 .file "ex5.c"
 .stag _doc,48
 .member _title,0,2,8,16
 .member _group,16,2,8,16
 .member _job_number,32,4,8,16
 .eos

```

语法:

```

.stag name [, size]
member definitions
.eos
.etag name [, size]
member definitions
.eos
.utag name [, size]
member definitions
.eos

```

描述: .stag伪指令开始结构的定义。 .etag伪指令开始枚举的定义。 .utag伪指令开始联合的定义。 .eos伪指令结束结构、枚举或联合的定义。

- *name* 是结构、枚举或联合的名。名的前 128 个字符有意义。这是必须的参数。
- *size* 是结构、枚举或联合在存储器中占据的位数。这是可选的参数; 如果它被省略, 那么大小就未被指定。

.stag、.etag 或 .utag 伪指令应当后随几个 .member 伪指令, 它们定义结构中的成员。 .member 伪指令是惟一可出现在结构、枚举或联合定义中的伪指令。

汇编器不允许结构、枚举或联合嵌套。 C 编译器通过分别定义嵌套的结构, 然后从它们被引用的结构中进行引用来解开嵌套的结构。

例 1: 以下是定义结构的例。

```

C 源代码:
struct doc
{
 char title;
 char group;
 int job_number;
} doc_info;
产生的汇编语言代码:
FP .set AR2

```

```
.file "ex5.c"
.stag _doc,48
.member_title,0,2,8,16
.member_group,16,2,8,16
.member_job_number,32,4,8,16
.eos
```

例 2: 以下是定义联合的例。

C 源代码:

```
union u_tag {
 int val1;
 float val2;
 char valc;
} valu;
```

产生的汇编语言代码:

```
.utag _u_tag,32
.member_val1,0,4,11,16
.member_val2,0,6,11,32
.member_valc,0,2,11,16
.eos
```

例 3: 以下是定义枚举的例。

C 源代码:

```
{
enum o_ty { reg_1, reg_2, result } otypes;
}
```

产生的汇编语言代码:

```
.etag _o_ty,16
.member_reg_1,0,4,26,26
.member_reg_2,1,4,16,16
.member_result,2,4,16,16
.eos
```

语法:

**.sym name, value [, type, storage class, size, tag, dims]**

描述: .sym 伪指令规定有关的全局变量、局部变量或函数的符号调试信息。

- *name* 是放入目标符号表中的变量名。名的前 128 个字符有意义。
- *value* 是与变量有关的数值。任何合法的表达式都是可接受的（绝对地址或可重定位地址）。
- *type* 是变量的 C 类型。关于 C 类型详见“附录 A 通用目标文件格式”。
- *storage class* 是变量的 C 存储类型。关于 C 存储类型详见“附录 A 通用目标文件格式”。
- *size* 是包含此变量所需的存储器字数。
- *tag* 是变量所属的类型或结构名。此名必须在前面由 .stag、.etag 和 .utag 伪指令声明。

□ `dims` 可以是 1~4 个由逗号隔开的表达式；这些表达式说明的是成员的维数。

参数的次序是重要的。`name` 和 `value` 是必须的参数。其它参数可以省略或为空（相邻的逗号表示空项）。这允许用户跳过个别参数并规定在表中其后出现的参数。省略的操作数或空操作设为空（`null`）值。

例：这些 C 语言程序代码产生下面所示的 `.sym` 伪指令。

C 源代码：

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp();
main(arg1,arg2)
int arg1;
char *arg2;
{
 register r1;
}
```

产生的汇编语言代码：

```
FP .set AR2
 .global _main
 .global _array
 .bss _array,50,1,0
 .global _ptr
 .bss _ptr,1,1,0
 .global _str
 .bss _str,2,1,0
 .global _ext
 .bss _ext,1,1,0
ac27 -q ex6.c ex6.if
 .file "ex6.c"
 .stag _s,32
 .member _member1,0,4,8,16
 .member _member2,16,4,8,16
 .eos
 .sect "text"
 .sym _main,_main,36,2,0
 .func 7
;*****
;* FUNCTION NAME: _main
;*
;* FUNCTION ENVIRONMENT
;* FUNCTION PROPERTIES
;* : 0 Parameter, 2 Auto, 0 SOE
;*****
_main:
;* AL assigned to _arg1
 .sym arg1,0,4,17,16
```

```
 ; * AR4 赋与 _arg2
 .sym arg2,6,18,17,16
 .stm arg1,-1,4,1,16
 .sym _arg2,-2,18,1,16
 ; * AL 赋与 _r1
 .sym _r1,0,4,4,16
 ADDB SP,#2
 .line 4
 MOV *-SP[1],AL ;|10|
 MOV *-SP[2],AR4 ;|10|
 .line 6
 SUBB SP,#2
 LRET
 ;return occurs
 .endfunc 12,000000000h,2
 .sym _array,_array,244,2,800,5,10
 .sym _ptr,_ptr,21,2,16
 .sym _str,_str,8,2,32,_s
 .sym _ext,_ext,4,2,16
```





## 附录 C 汇编器错误信息

当汇编器完成第二遍扫描时，它报告在汇编过程中所遇到的所有错误。如果创建了列表文件，那么它还会在列表文件中输出错误；错误在出现错误的源程序行之后输出。在纠正其他错误之前用户应该纠正在代码中出现的第一个错误；因为第一个错误有可能引发其他错误。

如果用户收到汇编器的错误信息，那么可以使用本附录找到解决问题的方法。首先定位错误信息等级号（等级号按字母顺序在本目录中列出）。然后在等级中定位用户得到的错误信息（每个等级号按字母顺序列出错误信息）。每个等级包括问题的描述以及提出可能的修正建议。

### E0000

**Can't find end of '%s' macro definition starting on line %d—Aborting!**

**Comma required to separate arguments**



**Comma required to separate parameters**  
**Commas must separate directive elements**  
**Expecting operand delimiter**  
**Invalid expression or missing right parenthesis**  
**Invalid register opnd for specified processor version**  
**Left parenthesis expected**  
**Matching right bracket is missing**  
**Missing comma**  
**Missing left parenthesis**  
**Missing right parenthesis**  
**Missing right quote of string constant**  
**No matching right parenthesis**  
**Right parenthesis expected**  
**Syntax error parsing register pair**  
**Too many -h include files (10 max)**  
**Too many files on command line**  
**Unrecognized character type**  
**Unrecognized special character**  
**no input files specified**  
**required switch -v (version) not specified**  
**syntax error**

描述: 常见的语法错误。不满足语法格式要求。

修改: 根据错误信息, 修正源程序。

## E0002

**Invalid directive for specified processor version**  
**Invalid instruction for specified processor version**  
**Incompatible C2XLP instruction**  
**Invalid mnemonic specification**

描述: 无效助记符。不能识别指定的指令、宏或伪指令。

修改: 检查用到的伪指令或指令, 然后修正源程序。

## E0003

**'HI' operand modifier improperly used**  
**'HI16' operand modifier improperly used**  
**.set/equ value cannot be a complex relocation expression**

Cluttered identifier operand encountered  
 Expecting \*XAR7 as operand  
 Expecting ARX++ or ARX as an operand  
 Expecting ARX as operand  
 Expecting XAR6/XAR7  
 Expecting condition code as operand  
 Expecting interrupt vector INT1 thru INT14  
 Expecting long memory operand  
 Illegal condition code  
 Illegal condition code combination  
 Illegal indirect memaddr specification  
 Invalid binary constant specified  
 Invalid constant specification  
 Invalid decimal constant specified  
 Invalid direct operand expression  
 Invalid float constant specified  
 Invalid hex constant specified  
 Invalid immediate value  
 Invalid octal constant specified  
 Invalid operand  
 Invalid processor version:  
 Invalid register used for indirect addr  
 Not expecting ARX/XRL operand  
 Not expecting extended AR as operand  
 Not expecting long memory operand  
 Only labels and comments may begin in the first column. Make sure that the label name is not a reserved keyword.  
 Section %s is not defined  
 Shift value out of range  
 Syntax error -Operand 1  
 Syntax error parsing register pair  
 描述: 无效操作数。不能识别指令、参数或其他指定的操作数。  
 修改: 根据错误信息, 修正源程序。

## E0004

'HI' modifier only valid for moves to AH/AL  
 \*reg is illegal for this instruction  
 Absolute, well-defined integer value expected

**C27x mode '%s' cannot be or'ed with C28x modes**

**Identifier expected**

**Identifier operand expected**

**Illegal character argument specified**

**Illegal mode bit symbol '%s'**

**Illegal offset value, expecting (0-63)**

**Illegal offset value, expecting (0-7) or AR0/AR1**

**Illegal operand combination**

**Illegal operand"**

**Illegal register pair**

**Illegal structure reference**

**Incompatible C2XLP instruction format**

**Invalid data size for relocation**

**Invalid identifier, %s, specified**

**Invalid macro parameter specified**

**No parameters available for macro arguments**

**Not expecting @ operand -Operand %d**

**Not expecting indirect operand -Operand %d**

**Not expecting register addressing for operand**

**Register addressing cannot be used**

**Single character operand expected**

**String constant or substitution symbol expected**

**String operand expected**

**Structure/Union tag symbol expected**

**Substitution symbol operand expected**

**描述:** 非法操作数。指令、参数或其他指定的操作数不符合语法要求。

**修改:** 根据错误信息, 修正源程序。

## E0005

**Missing field value operand**

**Missing operand(s)**

**Operand missing**

**Tag identification operand required**

**Tag symbol identifier required**

**描述:** 缺少操作数。未提供所需要的操作数。

**修改:** 根据错误信息, 修正源程序, 定义所有需要的操作数。

## E0006

**.break must occur within a loop**

**Conditional assembly mismatch****Matching .endloop missing****No matching .endif specified****No matching .endloop specified****No matching .if specified****No matching .loop specified****Open block(s) inside macro****Unmatched .endloop directive****Unmatched .if directive**

描述: 汇编伪指令不匹配所出现的伪指令需要一个与之匹配的伪指令, 但是汇编器不能找到与之匹配的伪指令。

修改: 根据错误信息, 修正源程序。

**E0007****Conditional nesting is too deep****Loop count out of range**

描述: 陷入条件汇编循环中不能退出, 条件程序块的嵌套不能超过 32 级。

修改: 修改源程序中的 macro/.endmacro、.if/.elseif/.else/.endif 或 .loop/.break/.endloop。

**E0008****Bad use of .access directive****Matching .struct directive is not present****Matching .union directive is not present**

描述: 不匹配的结构定义伪指令。采用 .struct/.endstruct 结构中, 所出现的伪指令需要一个与之匹配的伪指令, 但汇编器不能找到与之匹配的伪指令。

修改: 在源程序中检查不匹配结构定义的伪指令, 并改正它。

**E0009****Cannot apply bitwise NOT to floats****Illegal struct/union reference dot operator****Missing "++" operator****Missing right bracket****Missing right bracket****Missing struct/union member or tag****Section %s is not an initialized section**

**Structure or union tag symbol expected****Structure or union tag symbol not found**

描述: 非法操作。该操作对给定的操作数是非法的。

修改: 根据错误信息修正源程序, 定义所有需要的操作数。

**E0100****Label missing****Label required****.setsym requires a label**

描述: 标号遗失。所使用的伪指令需要一个标号, 但用户未指定所需要的标号。

修改: 在源程序中指定伪指令所需的标号。

**E0101****Standalone labels not permitted in structure/union defs**

描述: 无效标号。定义结构和联合时不允许指定标号, 但是程序指定了标号。

修改: 删除无效标号。

**E0102****Local label %d is multiply defined****Local label %d is not defined in this section****Local labels can't be used with directives**

描述: 非法使用局部标号。

修改: 根据错误信息, 修正源程序。使用.newblock 伪指令重新使用局部标号。

**E0200****Bad term in expression****Binary operator can't be applied****Cannot resolve symbol in expression****Difference between segment symbols not permitted****Illegal divide by zero****Illegal remainder by zero****Integer divide by zero****Integer remainder by zero****Offset expression must be integer value****Offset out of range****Operation cannot be performed on given operands**



**Unable to compose expression**

**Unary operator can't be applied**

**Value of expression has changed due to jump expansion**

**Well-defined expression required**

描述: 在一般表达式中, 使用了非法的操作数组合, 或需要进行的计算类型不存在。

修改: 根据错误信息, 修正源程序。

## E0201

**Absolute operands required for FP operations**

**Floating-point divide by zero**

**Floating-point expression required**

**Floating-point overflow**

**Floating-point underflow**

**Illegal floating-point expression**

**Invalid floating-point operation**

描述: 浮点表达式中发生错误。在需要整型表达式的地方使用了浮点表达式、或在需要浮点表达式的地方使用了整型表达式, 或者使用了无效的浮点数。

修改: 根据错误信息, 修正源程序。

## E0300

**%s is not defined in this source file**

**%s is operand to both .ref and .def**

**Can't tag an undefined symbol**

**Cannot equate an external symbol to an external symbol**

**Cannot redefine this section name**

**Empty structure or union definition**

**Illegal structure or union tag**

**Member '%s' was previously defined**

**Redefinition of %s attempted**

**Structure tag can't be global**

**Structure/union member, %s, not found**

**Symbol %s has already been defined**

**Symbol can't be defined in terms of itself**

**Symbol expected**

**Symbol expected in label field**

**Symbol, %s, has already been defined**

**The following symbols are undefined:**



**Union member '%s' previously defined**

**Union tag can't be global**

**local symbol '%s' is missing from this file**

描述： 常规符号发生错误。用户试图重定义一个符号或定义了一个非法符号。

修改： 根据错误信息，修正源程序。

## E0301

**Cannot redefine local substitution symbol**

**Substitution stack overflow**

**Substitution symbol not found**

描述： 一般置换符号错误。用户试图重定义一个符号或定义了一个非法符号。

修改： 要保证用宏参数或使用 .asg、.eval 伪指令来定义置换符号的操作数。

## E0400

**Symbol table entry is not balanced**

描述： 没有与符号调试伪指令配对的伪指令（例如，使用了 .block 却没有 .endblock）。

修改： 在源程序中检查条件不匹配的汇编伪指令，并改正它。

## E0500

**Macro argument string is too long**

**Missing macro name**

**Too many variables declared in macro**

描述： 有关宏的错误。

修改： 根据错误信息，修正源程序。

## E0501

**Macro definition not terminated with .endm**

**Matching .endm missing**

**Matching .macro missing**

**.mexit directive outside macro definition**

**No active macro definition**

描述： 宏定义伪指令发生错误。宏伪指令没有与之配对的伪指令（例如，使用了 .macro 却没有 .endm）。

修改： 根据错误信息，修正源程序。

## E0600

**%s is not in archive format**

**%s macro library not found**

**Bad archive entry for %s**

**Bad archive name**

**Can't read a line from archive entry**

**Macro library is not in archive format**

描述：访问宏库出错。在读或写宏库归档文件时，将会发生错误。原因可能是创建归档文件的方法不正确。

修改：确保宏库是未被汇编的汇编源文件。宏名以及成员名必须相同，文件的扩展名应为.asm。

## E0700

**.sym not allowed inside structure/union**

**Illegal structure/union member**

**No structure/union currently open**

描述：非法使用符号调试伪指令。在不恰当的地方使用了符号调试伪指令。

修改：根据错误信息，修正源程序。

## E0800

**Instructions not permitted in structure/union definitions**

**Parallel operator without instruction**

描述：出现非法操作数。指定的指令、参数或其他操作数不符合语法格式。

修改：根据错误信息，修正源程序。

## E0801

**Too many parallel instructions**

修改：根据错误信息，修正源程序。

## E0900

**.var allowed only within macro definitions**

**Can't include a file inside a loop or macro**

**Cannot change version after 1st instruction**

**Illegal structure definition contents**



**Illegal structure member**

**Illegal union member**

**Specified DIE is undefined**

**Specified parent DIE is undefined**

描述: 非法使用伪指令。在不允许使用伪指令的地方使用了伪指令。(在有些地方, 可能不能使用伪指令, 如果使用了伪指令, 可能会破坏目标文件。) 某些伪指令不允许出现在结构或联合的定义内部。

修改: 根据错误信息, 修正源程序。

## E1000

**Cannot open command file %s: %s**

**Cannot open object file : %s", assy\_obj\_name**

**Cannot read macro library: %s**

**Cannot read temporary macro file**

**Cannot read temporary macro file: %s**

**Cannot write to temporary macro file: %s**

**Include/Copy file not found or opened**

**Unable to open temp macro library: %s**

**Unable to position file pointer in macro invocation**

**cannot open listing file %s: %s**

**cannot open source file %s: %s**

描述: 不能找到指定的文件名。

修改: 检查拼写、路径名、环境变量等。

## E1300

**Copy limit has been reached**

**Exceeded limit for macro arguments**

**Macro nesting limit exceeded**

描述: 超出汇编器的限制。 .copy/.include 文件的嵌套不能超过 10 层。宏变量不超过 32 个参数。宏嵌套不超过 32 层。

修改: 检查源程序中嵌套层数是否超过了最大值。

## W0000

**Delay slot count must be 1 to 9, 1 assumed**

**Half-word offsets must be divisible by 2, truncated**

**Invalid page number specified -gnored**

**No operands expected. Operands ignored**

**Specified alignment is outside accessible memory - ignored**

**Too many operands**

**Trailing Operands Ignored**

**Word offsets must be divisible by 4, truncated**

描述: 有关操作数的警告。汇编器遇到不期望的操作数。

修改: 检查源程序, 查看引起问题的原因, 决定是否修改源程序。

## W0001

**Field width truncated to %d**

**Offset expression -Value out of range**

**Power of 2 required, %ld assumed**

**Section Name is limited to 8 characters**

**Specified value out of 8.bit range**

**String is too long -will be truncated**

**Value out of range**

**Value truncated**

**Value truncated to %d -bit width**

**Value truncated to byte size**

描述: 这个警告包括数截断, 表达式太长, 它不能放入到指令操作码中, 或超出了所需要的位数。

修改: 检查源程序, 确保结果在规定范围内, 如果发生错误, 那么修改源程序。

## W0002

**Address expression will wrap-around**

**Expression will overflow, value truncated**

描述: 有关算术表达式的警告。汇编器执行计算后产生一个结果, 这个结果可能被接受也可能不被接受。

修改: 检查计算结果是否可以被接受; 如果有错误, 那么修改源程序。

## W0003

**.Incorrect size for the type**

**.sym for function name required before .func**

描述: 有关符号调试伪指令的警告。在.func 伪指令之前不能出现.sym 伪指令定义的函数。

修改: 根据错误信息, 改正源程序。



## W0004

### **.Open block(s) at EOF**

描述: 有关结构定义的警告。

修改: 根据错误信息, 修正源程序。

## W9999

**Cannot debug assembly source files that contain > 65535 lines**

**Incompatible C2XLP directive -ignored**

**Possible pipeline hazard: Incorrect value may be transferred**

**Possible AR/BANZ pipeline conflict**

**Pipeline Hazard: PH may not be updated**

**Size specified won't fit on current page, allocating on next page**

**Size is > %d words, allocation will span page boundary**

**The binary + and -operators have higher precedence than the shift operators**

描述: 当出现的错误不属于任何其他错误类型时, 发出警告。

修改: 检查源程序, 决定引起错误的原因和是否需要修改源程序。





## 附录 D 链接器错误信息

本附录按字母表顺序列出了链接器的错误信息。在此列表中，符号（...）代表错误发生时，链接器试图与之相互作用的目标名。

### A

#### **absolute symbol (...) being redefined**

描述： 不能重新定义具有绝对地址值的符号。

修改： 检查所有表达式的语法和检查输入伪指令是否正确。

#### **adding name (...) to multiple output sections**

描述： SECTIONS 伪指令多次使用一个输入段。

修改： 修改用户链接器命令文件中的 SECTIONS 伪指令。

#### **ALIGN illegal in this context**

描述： 使用的符号调准在 SECTIONS 伪指令的有效范围外。

修改： 修改用户链接器命令文件，改变符号调准，使其 SECTIONS 伪指令有效范围内。

**alignment for (...) must be a power of 2**

描述: 段的调准不是 2 的幂。

修改: 确保在十六进制中, 所有 2 的幂由整数 1、2、4、8 后接多个 0。

**alignment for (...) redefined**

描述: 为段提供了多于一次的调准。

修改: 修改用户链接器命令文件。

**attempt to decrement DOT**

描述: 使用了非法语句。如, `.- = value`。对 `.Symbol` 赋值只能用创建空位的方式。

修改: 修改用户链接器命令文件。

**B****bad fill value**

描述: 填充值必须是 16 位常数。

修改: 修改用户链接器命令文件中指定的填充值。

**binding address (...) for section (...) is outside all memory on page (...)**

描述: 每一个段都必须映射到 MEMORY 伪指令分配的存储器中。

修改: 如果用户正在使用链接器命令文件, 那么请检查 MEMORY 和 SECTIONS 伪指令分配的空间是否足够大, 确保在未配置的存储器中没有分配段。

**binding address (...) for section (...) overlays (...) at (...)**

描述: 两个段重叠, 不能为其分配空间。

修改: 如果用户正在使用链接器命令文件, 那么请检查 MEMORY 和 SECTIONS 伪指令分配的空间是否足够大, 确保在未配置的存储器中没有分配段。

**binding address (...) incompatible with alignment for section (...)**

描述: 段的调准由 `align` 伪指令或预先链接产生的校正要求或预先链接。但绑定地址违反了此要求。

修改: 修改用户链接器命令文件。

**binding address for (...) redefined**

描述: 为段提供了多于一个的绑定值。

修改: 修改用户链接器命令文件, 删除多余的绑定值。

**blocking for (...) redefined**

描述: 为段提供了多于一个的分块值。

修改: 修改用户链接器命令文件, 删除多余的分块值。

**C****-c requires fill value of 0 in .cinit (... overridden)**

描述: `.cinit` 表必须用 0 来结束; 因此 `.cinit` 段的填充值必须为 0。

修改: 修改用户链接器命令文件。

**cannot complete output file (...), write error**

描述: 文件系统越界。

修改: 检查磁盘卷标; 删除文件或增加磁盘空间。

**cannot create output file (...)**

描述: 非法文件名。

修改: 检查拼写、路径名、环境变量等; 文件名必须符合操作系统的规定。

**cannot resize (...), section has initialized definition in (...)**

描述: 名为.stack 或.heap 的程序段已经初始化, 链接器不能重新确定段的大小。

修改: 修改用户链接器命令文件。

**cannot specify a page for a section within a GROUP**

描述: 在组内把段规定到了特定页。整个组作为一个单元, 所以组可以被指定到存储器页, 但是构成组的段不能单独处理。

修改: 修改用户链接器命令文件。

**cannot specify both binding and memory area for (...)**

描述: 同时指定的绑定地址和存储器存储空间, 二者是相互排斥的。

修改: 如果用户准备把代码放在规定的地址, 那么应只指定绑定地址; 如果用户准备把代码放在 MEMORY 伪指令规定的范围, 那么应只使用已命名存储器。

**can't align a section within GROUP -(...) not aligned**

描述: 组中的每个段都指定了各自的调准。整个组被当作一个单元, 所以组可以有自己的调准或被绑定到一个地址, 但组成组的段不能有自己的调准。

修改: 修改用户链接器命令文件。

**can't align within UNION -section (...) not aligned**

描述: 为联合中的段指定了一个调准。整个联合被当作一个单元, 所以联合可以有自己的调准或被绑定到一个地址, 但组成联合的段不能有自己的调准。

修改: 修改用户链接器命令文件。

**can't allocate (...), size ... (page ...)**

描述: 不能分配段, 因为在已配置的存储器没有这么大的空间来容纳这个段。

修改: 如果用户正在使用链接器命令文件, 检查 MEMORY 和 SECTIONS 伪指令查看是否有足够大的空间, 确保所有的段都放置在自己分配的存储空间中。

**can't create map file (...)**

描述: 通常指明使用了非法文件名。

修改: 检查拼写、路径名、环境变量等; 文件名必须符合操作系统的规定。

**can't find input file filename**

描述: 文件、文件名拼写错误, 或不在用户路径中等。

修改: 检查拼写、路径名、环境变量等; 文件名必须符合操作系统的规定。

**can't open (...)**

描述: 指定的文件不存在。

修改: 检查拼写、路径名、环境变量等; 文件名必须符合操作系统的规定。

**can't open filename**

描述: 不能打开指定文件名的文件; 原因可能是: 文件不存在, 或未纠正文件类型错误。

修改: 检查拼写、路径名、环境变量等;

**can't read (...)**

描述: 文件可能被破坏。

修改: 重新汇编输入文件。

**can't write (...)**

描述: 磁盘满或写保护。

修改: 检查磁盘卷标并确保磁盘未使用写保护, 磁盘同时能提供所需要的空间。

**command file nesting exceeded with file (...)**

描述: 命令文件嵌套最多为 16 层。

修改: 修改用户链接器命令文件。

## E

**-e flag does not specify a legal symbol name (...)**

描述: 作为 -e 选项操作数的符号名无效。

修改: -e 选项中使用合法的符号名。

**entry point other than \_c\_int00 specified**

描述: 仅在用户使用 -c 或 -cr 选项调用链接器时才出现该错误。程序的入口点不是 \_c\_int00。编译器的运行过程惯例假设 \_c\_int00 是惟一的入口点。

修改: 不需要任何操作。为了避免这种警告, 在使用 -c 或 -cr 选项时不要重新定义程序的入口点。

**entry point symbol (...) undefined**

描述: 与 -e 选项一起使用的符号未定义。

修改: 确保与 -e 选项一起使用的符号名被定义。

**errors in input - (...) not built**

描述: 由于链接错误没有创建输出文件。

修改: 修改链接器列出的其他错误, 然后重新链接文件。

## F

**fail to copy (...)**

描述: 文件可能被损坏。

修改: 重新汇编输入文件。

**fail to read (...)**



描述: 文件可能被损坏。

修改: 重新汇编输入文件。

**fail to seek (...)**

描述: 文件可能被损坏。

修改: 重新汇编输入文件。

**fail to skip (...)**

描述: 文件可能被损坏。

修改: 重新汇编输入文件。

**fail to write (...)**

描述: 磁盘满或写保护。

修改: 检查磁盘卷标; 确保未使用磁盘写保护, 磁盘能够提供所需要的空间。

**file (...) has no relocation information**

描述: 用户企图重新链接未用-r 选项链接过的文件。

修改: 如果打算重新链接所有文件; 链接时要使用-r 选项, 这样可以得到重定位的相关信息。

**file (...) is of unknown type, magic number = (...)**

描述: 二进制输入文件不是 COFF 文件。

修改: 确保所有输入到链接器的文件都是 TMS320C28x 的 COFF 文件格式。

**fill value for (...) redefined**

描述: 为输出段提供了多于一个填充值。在段定义中, 不能用不同的值来填充空位。

修改: 修改用户链接器命令文件。

**-i path too long (...)**

描述: 在-i 路径中的字符数超过了 256 的最大限制。

修改: 使用不大于 256 个字符的路径名。

**illegal input character**

描述: 在命令文件中有控制字符或其他不可识别的字符。

修改: 修改用户链接器命令文件。

**illegal memory attributes for (...)**

描述: 属性不是 R、W、I、X 的组合。

修改: 修改用户链接器命令文件。

**illegal operator in expression**

描述: 链接器检测到一个非法的表达式操作数。

修改: 对照表 7.2 中的合法表达式操作数, 从而修改用户代码。

**illegal option within SECTIONS**

描述: SECTIONS 伪指令使用了非法选项。  
修改: SECTIONS 伪指令中仅使用-l (小写的 L) 选项。

**illegal relocation type (...) found in section(s) of file (...)**

描述: 二进制文件被损坏。  
修改: 检查目标文件, 如有必要, 重新编译这些文件。

**internal error (...)**

描述: 链接器发生内部错误。  
修改: 联系微控制器的服务公司热线咨询。

**invalid archive size for file (...)**

描述: 归档文件被损坏。  
修改: 检查归档文件, 如有必要, 重新编译这些文件。

**invalid path specified with -i flag**

描述: -i 选项 (标志) 的操作数不是有效路径名。  
修改: 确保和-i 选项一起使用的路径名是合法的。

**invalid value for -f flag**

描述: -f 选项 (标志) 的数值不是 4 个字节 (32 位) 的常数。  
修改: -f 选项使用 4 个字节的常数。

**invalid value for -heap flag**

描述: -heap 选项 (标志) 的数值不是一个 4 字节 (32 位) 的常数。  
修改: -heap 选项使用 4 个字节的常数。

**invalid value for -stack flag**

描述: -stack 选项 (标志) 的数值不是一个 4 字节 (32 位) 的常数。  
修改: -stack 选项使用 4 个字节的常数。

**invalid value for -v flag**

描述: -v 选项 (标志) 的数值不是一个常数。  
修改: -v 选项使用的是常数。

**I/O error on output file (...)**

描述: 磁盘满或写保护。  
修改: 检查磁盘卷标并确保磁盘未使用写保护或能够提供所需要的空间。

**L**

**length redefined for memory area (...)**

描述: 在 MEMORY 伪指令中的存储区多于一个长度。  
修改: 修改用户链接器命令文件。

**library (...) member (...) has no relocation information**

描述: 在信息报告中命名的库成员没有重定位信息, 这意味着在链接时, 其他文件不能得到满足的确定信息。  
修改: 这个警告不需要用户作任何修改。没有重定位信息, 以致于库成员不起作

用, 链接器将忽略警告。

**line number entry found for absolute symbol**

描述: 输入文件被损坏。

修改: 重新汇编输入文件。

**load address for uninitialized section (...) ignored**

描述: 为未初始化段提供装载地址。未初始化段没有装载地址, 仅有运行地址。

修改: 修改用户链接器命令文件, 删除为未初始化段指定的装载地址。

**load address for UNION ignored**

描述: UNION 仅引用段的运行地址。

修改: 修改用户链接器命令文件。

**load allocation required for initialized UNION member (...)**

描述: 为联合中初始化段提供装载地址。UNION 只能引用运行地址。

修改: 用户必须为联合中所有段分别指定装载地址。修改用户链接器命令文件。

## M

**-m flag does not specify a valid filename**

描述: 没有为编写文件所对应的输出映像文件指定有效的文件名。

修改: 确保-m 选项使用的文件名是合法的文件名。

**making aux entry filename for symbol n out of sequence**

描述: 输入文件被损坏。

修改: 重新汇编输入文件。

**memory area for (...) redefined**

描述: 为输出段指定了多于一个的已命名的存储器地址。

修改: 修改用户链接器命令文件。

**memory attributes redefined for (...)**

描述: 为输出段提供多于一组的存储器属性。

修改: 修改用户链接器命令文件。

**memory page for (...) redefined**

描述: 为一个段提供多于一页的定位。

修改: 修改用户链接器命令文件。

**memory types (...) and (...) on page (...) overlap**

描述: 在同一页中存储器范围重叠。

修改: 如果用户正在使用链接器命令文件, 那么使用 MEMORY 和 SECTIONS 伪指令为程序段分配足够的空间, 所有的段都分配到已配置的存储器中。

**missing filename on -l; use -l <filename>**

描述: 没有为-l (小写 l) 选型提供文件名操作数。

修改: 使用-l 选项为不在当前目录中的库指定文件名。

**misuse of DOT symbol in assignment instruction**

- 描述: 赋值语句中使用的符号在 SECTIONS 伪指令的有效范围以外。  
修改: 修改用户链接器命令文件。

## N

### **no allocation allowed for uninitialized UNION member**

- 描述: 为联合中未初始化段提供装载地址。联合中未初始化段从 UNION 语句中取得其运行地址但是它没有装载地址, 所以对于成员来说没有有效的装载地址。

- 修改: 修改用户链接器命令文件。

### **no allocation allowed with a GROUP -allocation for section (...) ignored**

- 描述: 组中的段被各自单独分配地址。整个组作为一个单位, 所以一个组可以调准或绑定到一个地址, 但是构成组的段不能有自己的地址。

- 修改: 修改用户链接器命令文件, 删除段的指定的地址。

### **no input files**

- 描述: 未提供 COFF 文件。至少要有一个输入文件的格式为 COFF 格式, 否则链接器将不能工作。

- 修改: 当用户调用链接器时, 至少命名一个 COFF 文件作为其输入。

### **no load address specified for (...); using run address**

- 描述: 没有为初始化段提供装载地址。如果初始化段只有运行地址, 那么运行地址和装载地址为同一个地址。

- 修改: 不需要任何操作。链接器自动认定装载地址和运行地址相同。

### **no run allocation allowed for UNION member (...)**

- 描述: UNION 为联合的所有成员都定义了运行地址; 因此, 每个成员分配单独的运行地址是非法的。

- 修改: 修改用户链接器命令文件。

### **no string table in file filename**

- 描述: 输入文件被损坏。

- 修改: 重新汇编输入文件。

### **no symbol map produced -not enough memory**

- 描述: 可用的存储器不够生成符号表。这是一个非致命的条件, 它将阻止在映像文件中生成符号列表。

- 修改: 增加系统的可用存储器。

## O

### **-o flag does not specify a valid file name : (...)**

- 描述: 选择-o 选项的文件, 其命名不符合操作系统的命名规则。

- 修改: 确保-o 选项指定的文件名遵循操作系统的文件命名规则。



**origin missing for memory area (...)**

描述: 没有采用 MEMORY 伪指令规定起点。

修改: 修改用户链接器命令文件, 在 MEMORY 伪指令中包含一个起点值, 它指向存储器范围的起始地址。

**out of memory, aborting**

描述: 用户系统没有足够的存储器完成所有要求的任务。

修改: 试着把汇编语言文件分解为多个小文件, 然后单独链接它们。见 7.16 节部分 (递增) 连接。

**output file has no .bss section**

描述: 这是一个警告。 .bss 段通常出现在 COFF 文件中, 但是这不是必须的。

修改: 为了避免出现警告, 可以在用户链接器命令文件中指定 .bss 段。

**output file has no .data section**

描述: 这是一个警告。 .data 段通常出现在 COFF 文件中, 但是这不是必须的。

修改: 为了避免出现警告, 可以在用户链接器命令文件中指定 .data 段。

**output file has no .text section**

描述: 这是一个警告。 .text 段通常出现在 COFF 文件中, 但是这不是必须的。

修改: 为了避免出现警告, 可以在用户链接器命令文件中指定 .text 段。

**output file (...) not executable**

描述: 输出文件中可能有不能确定的符号, 或其他错误引起的问题。这种错误情况不是致命的。

修改: 不需要任何操作。该警告告诉用户代码不能完全被链接。

**overwriting aux entry filename of symbol n**

描述: 输入文件被损坏。

修改: 重新汇编输入文件。

**P****PC-relative displacement overflow at address (...) in file (...)**

描述: 在此指令中, 与 PC 有关的操作数的重定位导致太大的置换以至于不能编码。

修改: 修改存储器映像文件, 以在范围内进行置换。

**R****-r incompatible with -s (-s ignored)**

描述: 同时使用 -r 和 -s 选项。因为 -s 选项删出了重定位信息而 -r 选项要求目标文件有可重定位信息, 所以这两个选项是互相冲突的。

修改: 为了避免这类警告, -s 选项和 -r 选项不要一起使用。如果用户同时使用这些选项, 那么 -s 选项将不起作用。



**relocation entries out of order in section (...) of file (...)**

描述: 输入文件被损坏。

修改: 重新汇编输入文件。

**relocation symbol not found: index (...), section (...), file (...)**

描述: 输入文件被损坏。

修改: 重新汇编输入文件。

**S****section (...) at (...) overlays at address (...)**

描述: 两个段重叠, 不能为它们分配空间。

修改: 如果用户使用链接器命令文件, 那么确保 MEMORY 和 SECTIONS 伪指令为段指定了足够的空间, 以确保没有段相互重叠。

**section (...) enters unconfigured memory at address (...)**

描述: 不能为段分配空间, 因为在已分配的存储区中没有足够大的空间来保存该段。

修改: 如果用户使用链接器命令文件, 那么确保 MEMORY 和 SECTIONS 伪指令为段指定足够的空间, 以确保没有段被放置到未配置的存储器中。

**section (...) not built**

描述: 在 SECTIONS 伪指令中有语法错误。

修改: 在用户链接器命令文件中检查和修改定义的 SECTIONS 伪指令。

**section (...) not found**

描述: 在输入文件中没有找到 SECTIONS 伪指令指定的输入段。

修改: 修改用户链接器命令文件, 确保有一个输入文件中含有指定的输入段。

**section (...) won't fit into configured memory**

描述: 不能为段分配空间, 因为在已分配的存储区没有足够的空间来保存该段。

修改: 如果用户使用链接器命令文件, 那么检查 MEMORY 和 SECTIONS 伪指令使其为程序段分配足够的空间, 以确保没有段被放入未配置存储区中。

**seek to (...) failed**

描述: 输入文件可能被损坏。

修改: 重新汇编输入文件。

**semicolon required after assignment**

描述: 命令文件中有语法错误。

修改: 修改用户链接器命令文件。

**statement ignored**

描述: 表达式中有语法错误。

修改: 修改用户链接器命令文件。

**symbol (...) from file (...) being redefined**

- 描述: 在赋值语句中重新定义了已定义的符号。  
修改: 不需要任何操作。为了避免该警告, 删除链接器命令文件中的一个定义符号。

**symbol referencing errors - (...) not built**

- 描述: 引用的符号不能确定。因此, 不能建立目标文件。  
修改: 为了建立可执行文件, 输入文件中所有引用的符号必须能确定。

**T****too many arguments -use a command file**

- 描述: 用户在命令行上或在对提示的响应中使用了太多的变量。  
修改: 创建链接器命令文件以确定要传送给链接器的所有变量。

**too many -i options, 7 allowed**

- 描述: 使用的-i 选项多于 7 个。  
修改: 使用 C\_DIR 或 A\_DIR 环境变量指定附加子目录的搜索路径。

**type flags for (...) redefined**

- 描述: 为一个段指定了多于一个的段类型。注意, 类型 COPY 具有类型 DSECT 的所有属性, 所以不需要另外规定 DSECT。  
修改: 修改用户链接器命令文件。

**type flags not allowed for GROUP or UNION**

- 描述: 为组或联合中的段规定了类型。而特定的段类型仅用于单独的段。  
修改: 修改用户链接器命令文件, 为每一个段仅提供一个段类型。

**U****-u does not specify a legal symbol name**

- 描述: 在使用-u 选项时没有指定符号名。  
修改: 使用-u 选项指定合法的符号名。

**undefined symbol (...) first referenced in file (...)**

- 描述: 没有定义引用的符号, 或没有使用-r 选项。除非使用了-r 选项, 否则链接器要求定义所有引用的符号。出现上述情况, 链接器将不会创建可执行输出文件。  
修改: 链接时使用-r 选项, 或定义符号。

**undefined symbol in expression**

- 描述: 赋值语句包含未定义的符号。  
修改: 修改用户链接器命令文件。

**unexpected EOF(end of file)**

- 描述: 链接器命令文件中有语法错误。  
修改: 修改用户链接器命令文件。

**unrecognized option (...)**

描述: 使用了链接器不能识别的选项。

修改: 检查有效的选项列表, 见表 7.1。

**Z****zero or missing length for memory area (...)**

描述: 用 MEMORY 伪指令定义的存储器区域长度未给定或为零。

修改: 修改用户链接器命令文件。





## 附录 E 术 语 表

### A

**absolute address** (绝对地址): 在TMS320C28x存储器中固定不变的单元地址值。

**alignment** (调准): 链接器将输出段放置对准在一个地址上的一个过程, 此地址落在 $n$ 个字节的边界上, 其中 $n$ 为2的幂。用户可以用SECTIONS链接器伪指令指定调准。

**allocation** (分配): 链接器计算输出段的最终的存储器地址的过程。

**archive library** (归档库): 将多个分离的文件汇集在一起组成一个文件。

**archiver** (归档器): 归档器是一个软件程序, 它将许多单独的文件汇集在一起构成归档库文件。归档器允许用户删除、提取、替换或增加归档库的成员。

**ASCII**: 用于信息交换的美国标准代码。用于表示和交换文字信息的标准计算机代码。

**Assembler** (汇编器): 汇编器是根据源文件创建机器语言程序的软件程序, 其源文件中为汇编指令, 伪指令和宏指令。汇编器用绝对操作码替代符号操作码, 用可重定位地址替代符号地址。

**assembly-time constant** (汇编编译过程中使用的常数): 用.set伪指令给符号赋一个常数。

**assignment statement** (赋值语句): 给变量赋值的语句。

**autoinitialization** (自动初始化): 开始执行程序之前, 对全局C变量(包含在.cinit段中的)进行初始化的过程。

**Auxiliary entry** (辅助入口): 它是符号的额外入口, 在符号表中包含对这个符号的附加说明信息(它可能是一个文件名、一个段名、一个函数名等)。

## B

**big endian** (高字节在低地址, 低字节在高地址): 一种寻址协议, 在该寻址协议中一个字中字节右到左编号。在一个字中高字节的地址值较低。Endian的排序由硬件决定或在复位时指定。(见little endian)

**binding** (绑定): 为输出段或符号指定一个确切地址。

**block** (块): 用括号将一组定义或语句括起来构成的一个整体。

**.bss:** 是默认的COFF段中的一个程序段。可以用.bss伪指令在存储器映射区保留一个指定大小的空间, 此空间可以用于以后的数据存储。.bss段是未初始化段。

**byte:** 由一系列相邻的8位数字构成的单元。

## C

**C/C++ compiler** (C/C++编译器): 将C/C++源程序语句编译成汇编语言源程序的程序。

**command file** (命令文件): 包含选项、文件名、伪指令或链接器与十六进制转换应用程序的文件。

**comment** (注释): 它是源程序语句(或源程序语句的一部分), 此语句用于源程序文件的说明或提高源程序文件的可读性。对注释不进行编译、汇编、链接; 它对目标文件没有影响。

**common object file format (COFF)** (通用目标文件格式): 一种二进制目标文件格式, 它通过支持段概念加强模块化编程。所有的COFF段都可以单独在存储器空间进行重定位; 用户可以将任意段放入分配过的任意目标存储块中。

**conditional processing** (条件处理): 根据指定表达式的计算结果处理一个源代码块或源代码的转换块的方法。

**configured memory** (已配置存储器): 链接器已经分配了地址的存储器。

**constant** (常数): 一个能不改变并可以用作操作数的数值。

**cross-reference listing** (交叉引用列表): 由汇编器创建的一种输出文件, 此文件列出了定义的符号、符号定义的行、引用符号的行和符号的最后值。

## D

**.data:** 默认的COFF段中的一个程序段。.data段是一个初始化段, 它包含初始化的数据。可



以用.data伪指令汇编代码到.data段。

**directives** (伪指令): 是专用命令。它控制一个软件工具的行为和功能。(与汇编语言指令不同, 汇编语言指令控制一个器件的行为。)

## E

**Emulator** (仿真器): 模拟TMS320C2700操作的一种硬件开发系统。

**entry point** (入口点): 目标存储器中可执行程序的起始点。

**executable module** (可执行模块): 已经链接了的, 可以在TMS320C28x系统中执行的一种目标文件。

**expression** (表达式): 由运算符分开的常量、符号或一组常量和一组符号组成的式子。

**external symbol** (外部符号): 用于当前程序模块中的一种符号, 但此符号却在另一个程序模块中定义。

## F

**Field** (域): 对于TMS320C28x, 一种数据类型, 它的长度可编程为1~32位范围的任意值。

**file header** (文件头部): 包含一般信息的COFF目标文件的前头部分, 如: 段头部的数量、能够下载目标文件的系统类型、在符号表中的符号数量、符号表的起始地址。

## G

**global symbol** (全局符号): 具有以下特征之一的一种符号: 1) 在当前模块中定义能在另一个模块访问; 2) 在当前模块中访问但在另一个模块中定义。

**GROUP** (组): SECTIONS伪指令的一个选项。其作用是将输出段强制分配在一个连续的区域(作为一个组)。

## H

**hex-conversion utility** (十六进制转换应用程序): 一个能接受COFF文件, 并且为了将其装载到EPROM编制器中, 将它们转换成几种标准ASCII码十六进制格式之一的程序。

**high-level language debugging** (高级语言调试): 编译器的一种能保留符号信息和高级语言信息的能力(如类型定义和函数定义), 以便调试工具可以使用这些信息。

**hole** (空位): 没有实际代码和数据的区域。该区域位于构成输出段的输入段之间。

## I

**incremental linking** (递增链接): 文件分成几个过程进行链接。递增链接对于大型应用很有用。因为用户可以将应用程序划分成几个部分, 然后单独链接各个部分, 最后把所

有链接好的部分再链接起来。

**initialized section** (初始化段): COFF中的一种段, 它包含可执行代码或初始化数据。可以通过.data、.text或.sect伪指令来建立一个初始化段。

**input section** (输入段): 被链接到可执行文件的目标文件中的一种段。

## L

**Label** (标号): 一个符号, 它起始于源程序语句的第一列, 这个符号与该语句的地址相对应。

**line-number entry** (行号入口): 在COFF输出文件中的一项把汇编代码的行对应其创建代码的C语言程序行。

**linker** (链接器): 一个软件工具。它对目标文件进行组合并形成一个目标块, 此目标块可以分配到TMS320C28x系统存储器中并被器件执行。

**listing file** (列表文件): 一种输出文件, 它由汇编器创建, 它列出了源语句, 语句的符号及它们对SPC的影响。

**little endian**: 一种协议, 在一个字高有效字节为高地址。Endian的排序由硬件决定或在复位时指定。见big endian

**loader** (装载器): 装载可执行文件到TMS32C28X系统存储器中的一种装置。

## M

**Macro** (宏): 用户定义的一种程序, 它可以作为指令使用。

**macro call** (宏调用): 调用一个宏的过程。

**macro definition** (宏定义): 对一段源程序命名, 这段源程序代码就构成了宏。

**macro expansion** (宏表达式): 取代宏调用和宏汇编宏的语句。

**macro library** (宏库): 由宏组成的归档库。库中的每一个文件必须包含一个宏; 文件名必须与文件定义的宏名相同。其扩展名必须是.asm。

**magic number** (编码号): COFF头部入口, 为辨识目标文件的模式以便TMS320C28x可以执行。

**map file** (映像文件): 由链接器创建的一种输出文件, 它显示了存储器配置、段的组成、段的分配以及符号和它们对应的地址。

**member** (成员): 一个结构、联合、归档文件、枚举的元素或变量。

**memory map** (存储器映像图): 目标系统存储空间的分配图, 它反映按功能划分的存储器空间。

**mnemonic** (助记符): 可汇编成机器代码的指令名。

**model statement** (模块语句): 在宏定义中的多个指令或汇编伪指令组成的块, 在每次调用宏时都会被汇编一次。

## N

**named section** (已命名段): 用.sect伪指令定义的一个初始化段。

## O

**object file** (目标文件): 已汇编或链接的文件, 此文件包含机器语言的目标代码。

**object library** (目标库): 由目标文件组成的归档库。

**operands** (操作数): 汇编语言指令、汇编伪指令或宏伪指令的变量或参数。

**optional header** (可选文件头部): 它是COFF目标文件的一部分, 在下载时, 链接器为了完成重定位将会用到。

**options** (选项): 当调用一个软件工具时, 要求用户附加的或热量定用途的命令参数。

**output module** (输出模块): 一个已链接的可执行目标文件, 它能够下载到目标系统中并在目标系统中执行。

**output section** (输出段): 最终的可执行文件中已经链接好的段。

## P

**partial linking** (部分链接): 分几个过程链接文件。递增链接对于大型应用程序很有用。

因为用户可以将应用程序划分成几部分, 然后单独链接好各个部分, 最后把所有已链接好的部分再链接起来。

## Q

**quiet run** (安静运行): 隐藏运行的一般行为和进程信息的选项。

## R

**raw data** (原始数据): 在输出段中的可执行代码或初始化数据。

**relocation** (重定位): 当符号地址改变时, 链接器调整所有引用这个符号地址的过程。

**run address** (运行地址): 段在那里运行的地址。

## S

**Section** (段): 最后将在TMS320C28x存储器映射图中占据连续空间的可重定位的代码或数据块。

**section header** (段头部): COFF目标文件的一部分, 它包含了文件中段的有关信息。每个段有自己的头部; 头部指向段的起始地址, 且包含段的大小, 等等。)

**section program counter** (段程序指针): 见SPC。

**sign extend** (符号扩展): 用数值的一个符号位填充数值未用的最高符号位。

**simulator** (软仿真器): 模拟TMS320C28x操作的软件开发系统。

**source file** (源程序): 包含C代码或汇编语言代码的文件, 文件经过编译或汇编形成目标文件。

**SPC (section program counter)** (段程序计数器): 在段中跟踪当前位置的一个指针, 每个段都有自己的SPC。

**static variable** (静态变量): 一种限制在一个函数或程序内的变量。当退出函数或程序时, 静态变量的值不会被删除; 当再次进入函数或程序时, 恢复变量一上次值。

**storage class** (存储类型): 在符号表中用来说明如何寻址一个符号的相关信息。

**string table** (字符串表): 一种存储大于8个字符的符号名表 (大于或等于8个字符的符号名不存储在符号表中; 而是存储在字符串表中)。符号名入口的指向字符串表中字符串的位置。

**structure** (结构): 将一个或多个变量组合在一起形成的集合, 此集合用一个名字命名。

**subsection** (子段): 最后在TMS320C28x存储器映射图中占据连续空间的可重定位的代码或数据块。在较大段中, 子段大小相当于段。子段使用户对存储器映射图的控制更紧凑。

**symbol** (符号): 表示地址和数值的一串文字字符。

**symbolic debugging** (符号调试): 工具软件能保留符号信息的能力, 以便为仿真器调试工具使用, 例如仿真器。

**symbol table** (符号表): COFF目标文件的一部分, 它包含了目标文件中定义的符号及其使用的信息。

## T

**Tag** (标记): 一种可选类型名, 它能够被赋给一个结构、联合或枚举。

**target memory** (目标存储器): 在TMS320C28x系统中的物理存储器, 它用于装载可执行代码。

**.text**: 默认的COFF段之一。.text是一个包含可执行代码的初始化段。可以用.text伪指令将代码汇编到.text段。

## U

**unconfigured memory** (未配置的存储器): 没有被定义为存储器映射文件的一部分的存储器, 它不能装载代码或数据。

**uninitialized section** (未初始化段): 在存储器映射图中保留空间的COFF段, 但是它没有实际内容, 它不能由.bss和.usect伪指令建立。

**UNION** (联合): SECTIONS伪指令的一种选项, 它使链接器分配同一个地址给多个段。

**union** (联合): 能够保存不同类型和大小的变量。

**unsigned value** (无符号数): 无论实际符号是什么, 都作为一个正数来处理。

## W

**well-defined expression** (定义明确的表达式): 式中仅包含一个或一组符号或汇编过程中使用的常量在表达式之前就已定义。

**word** (字): 在目标存储器中16位可寻址的单元。





## 系列丛书书名

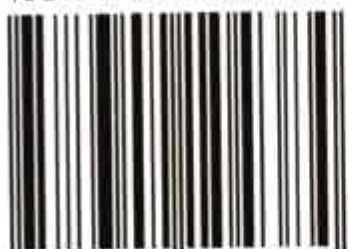
《TMS320LF/LC24 系列 DSP 的 CPU 与外设》  
《TMS320C28x 系列 DSP 的 CPU 与外设(上)》  
《TMS320C28x 系列 DSP 的 CPU 与外设(下)》  
《TMS320C54x 系列 DSP 的 CPU 与外设》  
《TMS320C547x 系列 DSP 的 CPU 与外设》  
《TMS320C55x 系列 DSP 的 CPU 与外设》  
《TMS320C6000 系列 DSP 的 CPU 与外设》

《TMS320LF/LC24 系列 DSP 指令和编程指南》  
《**TMS320C28x 系列 DSP 指令和编程指南**》  
《TMS320C54x 系列 DSP 指令和编程指南》  
《TMS320C55x 系列 DSP 指令和编程指南》  
《TMS320C6000 系列 DSP 编程工具与指南》

《TMS320 DSP 算法标准》  
《集成化开发环境(CCS)使用手册》  
《TI DSP/BIOS 用户手册》  
《TMS320 系列 DSP 硬件开发系统》

《TMS320C2000 在数字式电机控制系统中的应用》

ISBN 7-302-10438-7



9 787302 104384 >

定价: 46.00元