第 3 章

TMS320C6000 系列的指令系统

教学提示：TMS320C6000 系列芯片具有许多强大的功能，针对其具体功能以及结构的不同，C6000 具有不同于 TMS320C2000 和 TMS320C5000 的指令系统。本章主要介绍 TMS320C6000 系列的指令系统，包括与指令有关的概念、公共指令集、汇编、线性汇编、伪指令以及 C 语言与汇编的混合编程等。

教学要求：本章要求学生掌握 TMS320C6000 系列芯片指令系统的基本概念，了解公共指令集的特点，熟悉常用的汇编指令，了解其汇编、线性汇编和伪指令，理解 C 语言与汇编的混合编程等内容。

3.1 TMS320C6000 公共指令集概述

3.1.1 指令和功能单元之间的映射

C6000 汇编语言的每一条指令只能在一定的功能单元执行，因此就形成了指令和功能单元之间的映射关系。2.2.1 节，给出了功能单元所能执行的操作。相应地，可以给出指令到功能单元的映射，指出每一条指令可在哪些功能单元运行；也可以给出功能单元到指令的映射，指出每个功能单元可以运行哪些指令。一般而言，与乘法相关的指令都是在.M 单元执行；需要产生数据存储器地址的指令，则要用到.D 功能单元；算术逻辑运算大多在.L 与.S 单元执行。

3.1.2 延迟间隙

C6000 采用流水线结构，从指令进入 CPU 的取指单元到指令执行完毕，需要多个时钟周期。C6000 所宣传的单指令周期是指它最高的流水处理速度。由于指令复杂程度的不同，各种指令的执行周期也不相同，因此程序员就需要了解指令执行的相对延迟，以掌握一条指令的执行结果何时可以被后续指令利用。指令的执行速度可以用延迟间隙(delay slots)来说明。延迟间隙在数值上等于从指令的源操作数被读取直到执行的结果可以被访问所需要的指令周期数。对单周期类型指令(如 ADD)而言，源操作数在第 i 周

期被读取，计算结果在第(i+1)周期即可被访问，等效于无延迟。对乘法指令(MPY)而言，如源操作数在第 i 周期被读取，计算结果在第(i+2)周期才能被访问，延迟周期为 1。表 3.1 给出了各类指令的延迟间隙。

表 3.1 C6000 公共指令集的延迟间隙和功能单元等待时间

指令类型	延迟间隙	功能单元等待时间	读周期	写周期	转移发生
NOP	0	1			
Store	0	1	i	i	
单周期	0	1	i	i	
乘法(16×16)	1	1	i	i+1	
Load	4	1	i	i,i+4	
转移	5	1	i		i+5

表 3.1 中第 4、5 列以进入流水线 E1 节拍为第 i 周期，列出了各类指令要做读写操作所发生的周期号。对于转移类指令，如果是转移到标号地址的指令或是由中断 IRP 和 NRP 引起的转移，没有读操作；对于 Load 指令，在第 i 周期读地址指针并且在该周期内修改基地址，在第 i+4 周期向寄存器写，使用的是不同于 D 单元的另一个写端口。

C6000 所有的公共指令都只有一个功能单元等待时间，这意味着每一个周期功能单元都能够开始一个新指令。单周期功能单元等待时间的另一术语是单周期吞吐量。

3.1.3 指令操作码映射图

C6000 的每一条指令都是 32 位。每一条指令都有自己的代码，详细指明指令相关内容。图 3.1 给出一个使用.L 功能单元指令编码的指令操作码映射图(opcode map)。其中 op 为指令操作代码，creg 指定条件寄存器的代码，z 指定条件，src、dst 分别指定源及目的操作数代码，s 选择寄存器组 A 或 B 作为目的操作数，x 指定源操作数 2(src2)是否使用交叉通道，p 指定是否并行执行，等等。把汇编语句变成代码，由汇编器(assembler)完成；把代码反汇编成汇编语句也是由专用工具程序完成的。

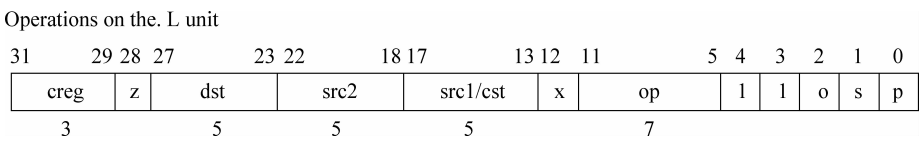


图 3.1 TMS320C6000 .L 指令操作码映射图

3.1.4 并行操作

CPU 运行时，总是一次取 8 条指令，组成一个取指包。取指包的基本格式由图 3.2 给出。取指包一定在地址的 256 位(8 个字)边界定位。

每一条指令的最后一位是并行执行位(P 位)，P 位决定本条指令是否与取指包中的

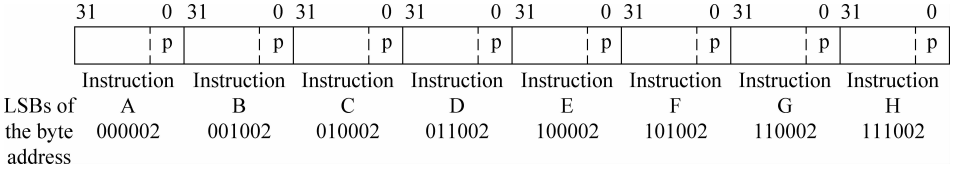


图 3.2　取指包的基本格式

下一指令并行执行。CPU 对 P 位从左至右(从低地址到高地址)进行扫描：如果指令 i 的 P 位是 1，则指令 $i+1$ 就将与指令 i 在同一周期并行执行；如果指令 i 的 P 位是 0，则指令 $i+1$ 将在指令 i 的下一周期执行。所有并行执行的指令组成一个执行包，其中最多可以包括 8 条指令。执行包中的每一条指令使用的功能单元必须各不相同。执行包不能超出 256 位边界，因此，取指包最后一条指令的 P 位必须设定为 0，而每一取指包的开始也将是一个执行包的开始。

一个取指包中 8 条指令的执行顺序可能有几种不同形式：完全串行，即每次执行一条指令；完全并行，即 8 条指令是一个执行包；部分串行，即分成几个执行包。下面给出一个由不同的 P 位设置造成部分串行执行的例子。

例 3.1　一个取指包分成几个执行包时，各个指令的并行执行位(P 位)模式，如图 3.3 所示。

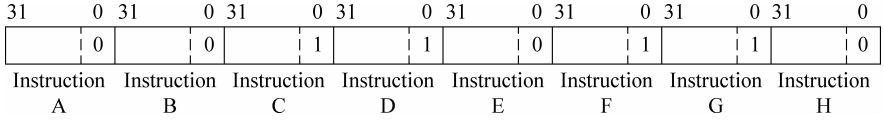


图 3.3　指令包的并行标志位

根据各个指令并行位的模式，此指令包将如表 3.2 所示的顺序执行。

表 3.2　例 3.1 的指令执行包

周期/执行包	指　令
1	A
2	B
3	C　D　E
4	F　G　H

注：指令 C、D 和 E 不能使用相同的功能单元、交叉通路或其他路径资源。这同样适用于指令 F、G 和 H。

此例所示指令包的指令代码描述如下：

```
指令 A
指令 B
指令 C
||指令 D
||指令 E
```

```
指令 F
||指令 G
||指令 H
```

其中，符号||表示本条指令与前一条指令并行执行。

如果有转移指令使程序在执行过程中由外跳转至某一执行包中间的某一条指令，则程序从该条指令继续执行，该执行包中跳转目标之前的指令将被忽略。以例 3.1 为例，如果跳转目标是指令 D，则只有指令 D 和 E 将被执行。虽然指令 C 与 D 处于同一执行包中，它也得不到执行。至于指令 A 和 B，由于处于前一执行包，更不会得到执行。如果程序的运行结果依赖于指令 A、B 或 C 的执行结果，像这样直接向指令 D 的跳转将会产生错误。

3.1.5 条件操作

所有的 C6000 指令都可以是有条件执行的，反映在指令代码的 4 个最高有效位（参见图 3.1）。其中 3 位操作码字段 creg 指定条件寄存器，1 位字段 z 指定是零测试还是非零测试。在流水操作的 E1 节拍，对指定的条件寄存器进行测试：如果 z=1，进行零测试，即条件寄存器的内容为 0 是真；如果 z=0，进行非零测试，即条件寄存器的内容非零是真。如果设置为 creg=0，z=0，则意味着指令将无条件地执行。C62xx/C67xx，可使用 A1、A2、B0、B1 和 B2 这 5 种寄存器做条件寄存器，对 C64xx，还可增加 A0 寄存器作为条件寄存器。

在书写汇编程序时，以方括号对条件操作进行描述，方括号内是条件寄存器的名称。下面所示的执行包中含有两条并行的 ADD 指令。第 1 个 ADD 指令在寄存器 B0 非零时条件执行，第 2 个 ADD 指令在 B0 为零时条件执行。

```
[B0]    ADD     .L1      A1, A2, A3
|| [!B0]  ADD     .L2      B1, B2, B3
```

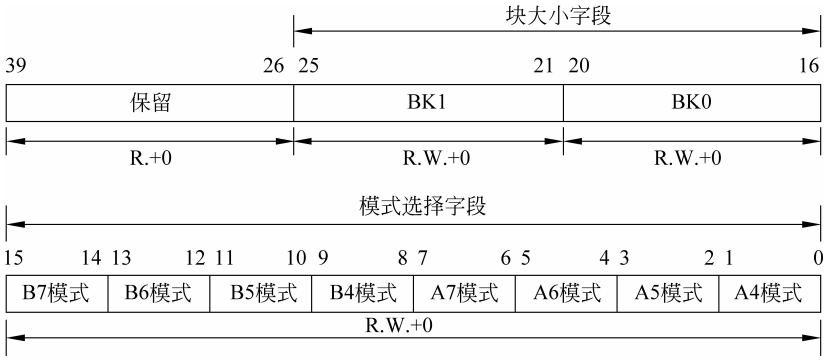
以上两条指令是相互排斥的，也就是说只有 1 条指令将会被执行。互斥指令被安排并行时也有一些限制。

3.1.6 寻址方式

寻址方式指 CPU 是如何访问其数据存储空间。C6000 全部采用间接寻址，所有寄存器都可以做线性寻址的地址指针。A4～A7，B4～B7 等 8 个寄存器还可作为循环寻址的地址指针，由寻址模式寄存器 AMR 控制地址修改方式：线性方式（默认方式）或循环方式。

1. AMR 寄存器

AMR 为 8 个可以执行线性和循环寻址的寄存器（A4～A7，B4～B7）来指定寻址模式。每个寄存器对应两位字段，指定该寄存器的寻址模式：线性（默认模式）或者循环模式。在循环模式下，这两位还要指定哪个 BK（块的大小）段用于循环缓冲器：模式选择字段和 BK 字段如图 3.4 所示，模式选择字段编码如表 3.3 所示。



注：R表示由MVC指令可读；W表示由MVC指令可写；+0表示复位后值为0。

图 3.4　寻址模式寄存器 AMR 各个位域的定义

表 3.3　寻址模式寄存器(AMR)模式选择字段编码

模　式	描　　述	模　式	描　　述
00	线性模式(复位时的默认值)	10	使用 BK1 段循环寻址
01	使用 BK0 段循环寻址	11	保留

AMR 的保留部分常为 0，AMR 在复位时初始化为 0。块大小字段 BK0 和 BK1，包含 5 位值，用于计算循环寻址的块的大小：块大小(以字节为单位)＝2^(*N*＋1)。*N* 为 BK0 或 BK1 的 5 位值。

表 3.4 为 32 种可能的块的计算尺寸。

表 3.4　块尺寸计算

N	块　尺　寸	*N*	块　尺　寸
00000	2	10000	131072
00001	4	10001	262144
00010	8	10010	524288
00011	16	10011	1048576
00100	32	10100	2097152
00101	64	10101	4194304
00110	128	10110	8388608
00111	256	10111	16777216
01000	512	11000	33554432
01001	1024	11001	67108864
01010	2048	11010	134217728
01011	4096	11011	268435456
01100	8192	11100	536870912
01101	16384	11101	1073741824
01110	32768	11110	2147483648
01111	65536	11111	4294967296

注：当 *N* 为 11111 则被认为是线性寻址。

在线性寻址方式下，基地址按照指定的加减量线性修改。在循环寻址方式下，从第 N 位向第 $N+1$ 位的进位/借位被禁止，即第 0～N 位地址在块尺寸范围内循环修改，超出块尺寸字段的高位地址（第 $N+1$ 至 32）不变。块尺寸字段 BK0 和 BK1 含有 5 位数值，用于计算循环寻址循环块的尺寸。例如，设 N 的二进制数为 10000，等于十进制 16，则块尺寸为 2^(16+1)=131 072 字节。

2. 读取/存储（load/store）类指令

表 3.5 列出了 load/store 类指令访问数据存储器地址的汇编语法格式。其中 ucst5 代表一个无符号的二进制 5 位常数偏移量。对寄存器 B14 和 B15 可用 ucst15（无符号的二进制 15 位常数偏移量）。变址计算的符号与常用的 C 语言惯例相同。

表 3.5 load/store 类指令间接地址的产生

寻址类型	不修改地址寄存器	先修改地址寄存器	后修改地址寄存器
寄存器间接寻址	*R	*++R *--R	*R++ *R--
寄存器相对寻址	*+R[ucst5] *-R[ucst5]	*++R[ucst5] *--R[ucst5]	*R++[offsetR] *R--[offsetR]
带 15 位常数偏移量的寄存器相对寻址	*+B14/B15[ucst15]		
基地址+变址	*+R[offsetR] *-R[offsetR]	*++R[offsetR] *--R[offsetR]	*R++[offsetR] *R--[offsetR]

例 3.2 用汇编程序设置循环寻址方式。

下面的汇编程序段将 A4 寄存器设置成循环寻址方式，用 BK0 定义块尺寸为 16 字节。注意，只有用 MVC 指令和.S2 功能单元才能访问控制类寄存器。

```
MVK     .S1     0001h,    A2
MVKLH   .S1     0003h,    A2
MVC     .S2x    A2,       AMR
```

3.2 C6000 公共指令集

C62xx 的所有指令对 C64xx/C67xx 均有效，它的指令集就是本节介绍的所有 C6000 芯片共有的指令集。TMS320C6000 公共指令集是一个定点运算指令集。

TMS320C6000 的指令主要有读取/存储类指令、算术运算类指令、乘法运算指令、逻辑及位操作运算类指令、搬移类指令、跳转（程序转移）类指令 6 种。

3.2.1 读取/存储类指令

load 指令可从数据存储器读取数据送到通用寄存器，store 指令可把通用寄存器的内容送到数据存储器保存。使用此类指令应注意下列问题。

1. 符号扩展指令与无符号扩展指令的区别

LDB/LDH 指令读入的是有符号数(补码数),将字节/半字写入寄存器时,应对高位做符号扩展。LDBU/L、DHU 指令读入的是无符号数,将字节/半字写入寄存器时,应对高位补零。

2. 数据类型对地址偏移量的影响

LDB(U)/LDH(U)/LDW 指令分别读入字节/半字/字,因为地址均以字节为单位,在计算地址修正量时,要分别乘以相应的比例因子 1、2、4。STB/STH/STW 同样计算地址。

例 3.3 线性寻址下的变址计算。

1) LDH .D1 *++A4[A1],A8

此例为先修改地址,地址偏移量按[A1]×2 计算,实际读取第 00000024h 单元的内容。

Before LDH		1 cycle after LDH		5 cycle after LDH	
A1	0000 0002h	A1	0000 0002h	A1	0000 0002h
A4	0000 0020h	A4	0000 0024h	A4	0000 0024h
A8	1103 51FFh	A8	1103 51FFh	A8	FFFF A21Fh
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 24h	A21Fh	mem 24h	A21Fh	mem 24h	A21Fh

2) LDW .D1 *A4++[1],A6

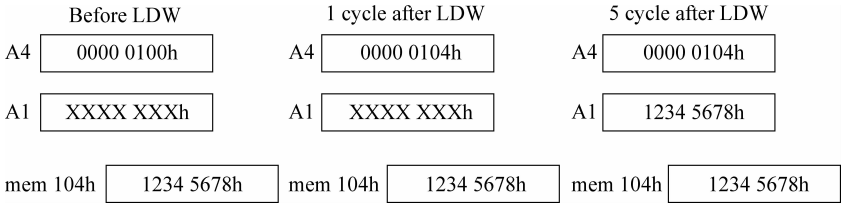
此例为后修改地址,地址偏移量按 1×4 计算。

Before LDW		1 cycle after LDW		5 cycle after LDW	
A4	0000 0100h	A4	0000 0104h	A4	0000 0104h
A6	1234 4321h	A6	1234 4321h	A6	0789 F25Ah
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 100h	0789 F25Ah	mem 100h	0789 F25Ah	mem 100h	0789 F25Ah
mem 104h	1970 19F3h	mem 104h	1970 19F3h	mem 104h	1970 19F3h

例 3.4 循环寻址方式下的地址计算。

```
LDW  .D1    *+ + A4[9],A1
```

此处假设 A4 已按例 3.2 的程序被制定为循环寻址方式,块字节尺寸为 16($N=3$)。因为是以字为单位读取,变址偏移量应乘 4。在线性寻址时,地址为 00000124h;循环寻址时,低 4 位(第 0～3 位)地址以 10h 为模,余数为 4,故实际寻址地址为 00000104h。



3. 数据类型与地址边界调整，终结方式(EN)对数据读取、存放的影响

在C6000的Load/Store指令里，数据长度有单字节、双字节(半字、短型定点数)和4字节(字、定点数)等多种。对C62xx/C67xx双字节型数据的地址必须从偶数开始，即其地址最低位为0；4字节数据地址最低2位必为0，分别称为半字、字边界。在计算或书写地址时，均以它们的最低位地址作为存储单元地址的代表。

在汇编语言或C语言中开辟数据或变量区时，需要根据数据类型调节其地址起始点，称为地址边界调整(alignment)。虽然C64xx的某些指令具有无须边界调整的功能，但其默认工作方式仍是有地址边界调整，仅在特别声明情况(例如LDNW等指令)下，才可以使用无边界调整的地址。

终结方式(EN或Endian)是指多字节数据内部高低有效位的存放顺序。在小端终结方式(Little-Endian)下，数据的高有效位字节存放在地址高位字节(高位高地址)，低有效位放在地址低位字节，这与Intel公司数据存放惯例相同。大端终结方式(Big-Endian)则反过来，与Motorola等公司的数据存放惯例相同。终结方式(EN)由芯片的相应管脚LENDIAN的电平决定，并反映在CSR寄存器的EN位，LENDIAN＝1为小端终结，LENDIAN＝0为大端终结。

表3.6给出了不同终结方式下执行Store指令后存储器的内容(设寄存器内容为BA987654h，存储器原内容为01121970h)。表3.7给出了不同终结方式下执行Load指令后寄存器的内容(设存储器的内容为BA987654h)。

表3.6 寄存器内容为BA987654h，不同终结方式下执行Store指令后存储器内容

指令	地址最低位(1:0)	大端终结时的存储结果	小端终结时的存储结果
STW	00	BA987654h	BA987654h
STH	00	76541970h	01127654h
STH	10	01127654h	76541970h

表3.7 寄存器内容为BA987654h，不同终结方式下执行Load指令后存储器内容

指令	地址最低位(1:0)	大端终结时的存储结果	小端终结时的存储结果
LDW	00	BA987654h	BA987654h
LDH	00	FFFFBA98h	00007654h
LDHU	00	0000BA98h	00007654h
LDH	10	00007654h	FFFFBA98h

4. 按寻址方式的加减运算类指令 ADDA/SUBA

按寻址方式的加减运算类指令在默认方式下做线性加减法运算。但当源操作数 src2 是 A4～A7 或 B4～B7 中的一个时，它按 AMR 寄存器指定的寻址方式做线性或循环计算。指令的操作数有字/半字/字节 3 种。此类指令只在功能单元 D1 或 D2 运行。下面给出两个按循环寻址方式计算的例子。

例 3.5 按寻址方式的加法运算指令。

```
ADDAH   D1  A4,A2,A4
```

假定 AMR 寄存器设定 A4 为循环寻址，块尺寸为 8 字节($N=2$)。ADDAH 指令以半字为单位，A2、A4 高半字都是 0，加法结果仍为 0。低半字运算，仅 A4 的低 3 位(bit0～bit2)改变。计算加法偏移量时，ADDAH 指令以半字为单位，地址以字节为单位，所以偏移量为 A2 低半字乘 2。加法结果的低 3 位以 8 为模，保留余数。

	Before instruction		1 cycle after instruction
A2	0000 000Bh	A2	0000 000Bh
A4	0000 0100h	A4	0000 0106h
AMR	0002 0001h	AMR	0002 0001h

例 3.6 按寻址方式的减法运算指令。

```
SUBAB, D1  A5,A0,A5
```

此处假定 AMR 寄存器指定 A5 为循环寻址，块尺寸为 16 字节($N=3$)。SUBAB 指令以字节为单位，寄存器 A0 高 3 个字节都是 0，故 AS 的高 3 个字节结果不变。最低字节运算，仅 AS 的低 4 位(bit0～bit3)改变。计算减法偏移量时，SUBAB 指令以字节为单位，地址亦以字节为单位，所以偏移量就是 A0 最低字节。减法结果的低 4 位以 10h 为模，保留余数。

	Before instruction		1 cycle after instruction
A0	0000 0004h	A0	0000 0004h
A5	0000 4000h	A5	0000 400Ch
AMR	0003 0004h	AMR	0003 0004h

3.2.2 算术运算类指令

1. 加减运算指令

加减运算指令可分为以下几类：

(1) 有符号数加减运算指令，包括操作数为整型(32 位)或长整型(40 位)的 ADD、SUB 指令，操作数为半字(16 位)的 ADD2/SUB2 指令。ADD2/SUB2 指令的特点是同时进行 2 个 16 位补码数的加减运算，高半字与低半字之间没有进/借位，各自独立进行。

(2) 无符号数加减运算指令 ADDU、SUBU,操作数为 32 位或 40 位无符号数。

(3) 带饱和的有符号数加减运算指令 SADD、SSUB,操作数为 32 位或 40 位有符号数。

(4) 与 16 位常数进行加法操作的指令 ADDK。

加减运算指令操作容易,重点是计算中的溢出问题。

2. 溢出问题

运算结果超出目的操作数字长所能表示数的范围,造成运算结果的高位丢失,使保存的运算结果不正确,称为溢出。由于定点数以全字长表示一个整数,其能覆盖的数据动态范围较小,定点加减运算指令(无论是无符号或有符号加减运算指令)易产生溢出。例如,若目的操作数选定为半字(16 位字长),它所能表示的补码数范围是-32 768～+32 767,能表示的无符号数范围是 0～65 535。如果运算结果超出此范围,它将只保留运算结果的低 16 位,高位丢失,运算结果不正确。有符号数(补码数)在产生溢出时,会改变运算结果的符号。这两种情况都是十分有害的。在 C6000 指令里,普通加减运算指令产生溢出时在 CPU 内不会留下任何标志,所以对此要充分重视。

解决溢出问题的方法通常有以下 3 种:

(1) 用较长的字长来存放运算结果,使目的操作数字长超出源操作数的字长。超出源操作数字长的部分称为保护位(guard bits)。例如 2 个 32 位整型数的加减,用 40 位存放结果,有 8 个保护位。但增加保护位要占用系统资源,还可能会降低运算速度(例如,有些系统存储 40 位整数要比存 32 位整数占用较多时间),使用时必须综合考虑。

(2) 用带饱和的加减运算指令 SADD、SSUB 做补码数加减运算。当产生溢出时,这类指令将使目的操作数置为同符号的最大值(绝对值),即保持运算结果的符号不变,同时使 CPU 的状态寄存器 CSR 内的 SAT 位置 1。这一点需要注意。

(3) 对整个系统乘一个小于 1 的比例因子,亦即将所有参入的数值减小,以保持运算过程不产生溢出,但这种方法会降低计算精度。

一般而言,用与源操作数相同字长的数据类型来保存累加和是非常危险的。通常的选择是在计算过程(循环)内用较长的数据类型保存和数,最后根据具体情况选取适当字长。总之,应根据源操作数及运算次数,慎重选择数据类型和运算方法,防止溢出。

3. 定标和 Q 格式数

定点运算指令里的操作数都是整数,但是数字信号处理中的数值运算涉及小数运算。小数需要乘一个比例因子并取整变成整型数后,才能使用定点运算指令。把实际系统转换成可用定点运算的过程由用户自身完成,在定点指令程序中无法反映。在转换中保证系统的比例因子一致是非常重要的,因为定点运算指令认为操作数都是整数,做加减运算不存在对阶(即小数点位置)问题,只有所有数的比例因子一致才能得到正确结果。通常用 Q 格式数表示小数,以 16 位定点数为例:最高位为符号位,如规定符号位后即是小数点位置,则称为 Q15 格式数,其表示的实际小数范围是$-1 \leqslant x \leqslant 0.9999695$。如转换时确定符号位后有 1 位整数,而后才是小数点位置,则称为 Q14 格式数,其表示的实际小数范围是$-2 \leqslant x \leqslant 1.9999390$。数值动态范围大了,小数点后的有效位就减少。依此可类推

Q13、Q12 等格式数的定义。在字长为 32 位时，则可定义 Q31、Q30……Q15 等格式数。

选定 Q 格式值之后，应将系统的全部数值转换成同一格式的定点数，称为定标，即确定同一标尺。有时为防止溢出，也可以将所有参与操作的数除以同一比例因子。例如，都右移 1 位，即除以 2，然后再做加减运算。这样做的缺点是降低了计算精度。

4. 其他算术运算类指令

C6000 还提供了 ABS(取绝对值)、ADDK(与 16 位有符号常数相加)和 SAT(将 40 位长型有符号数转换为 32 位有符号数)等算术运算指令。SAT 指令在转换时，如果被转换数超出了 32 位有符号数所能表示的范围，则取 32 位的饱和值，并将 CSR 寄存器中的 SAT 位置 1。

3.2.3 乘法运算指令

C6000 公共指令集内的乘法指令以 16×16 位的硬件乘法器为基础，可分为以下两大类：

(1) 适宜于整数乘法的指令。

(2) 适宜于 Q 格式数相乘的指令。

1. 适宜于整数乘法的指令

整数乘法的 2 个源操作数都是 16 位字长，目的操作数为 32 位的寄存器。根据源操作数为有/无符号数以及源操作数是寄存器的低/高半字，可以组合出 16 种不同的乘法指令。除了 2 个无符号源操作数相乘外，只要有 1 个源操作数是有符号数，其结果就认定是有符号数。由于目的操作数为 32 位，乘法指令不存在溢出问题。

2. 适宜于 Q 格式数相乘的 3 条指令

在 2 个 Q 格式数相乘时，用有符号整数乘法指令，其结果将有 2 个符号位，其小数点位置也需重新判定。C6000 提供了一类带左移及饱和的乘法指令 SMPY/SMPYLH/SMPYHL，它将 2 个有符号数相乘的结果左移 1 位，使之只有 1 位符号位。如果原来是 2 个 Q15 格式数，则该类乘法指令运算结果为 Q31 格式数。

SMPY/SMPYLH/SMPYHL 指令有一个特殊情况，那就是当 2 个源操作数都是 8000h 时，按上述处理将出现错误。C6000 规定，当 SMPY/SMPYLH/SMPYHL 指令的 2 个源操作数都是 8000h 时，则将运算结果置为 32 位有符号数的最大正值，并将 CSR 寄存器的饱和位(SAT)置位。

3.2.4 逻辑及位域操作指令

1. 逻辑运算(布尔代数运算)指令

C6000 支持典型的布尔代数运算指令 AND、OR 和 XOR。这类指令都是对两个操作数按位做“与”、“或”和“异或”运算，并将结果写入目的寄存器。

C6000 还提供求补码的指令 NEG，可用于对 32 位、40 位有符号数求补码。

2. 移位指令

C6000 公共指令集共有 4 种移位指令：算术左移指令 SHL、算术右移指令 SHR、逻

辑右移(无符号扩展右移)指令 SHRU 和带饱和的算术左移指令 SSHL。图 3.5 给出了一个长型数据(40 位)执行 SHR 指令操作的示意图。

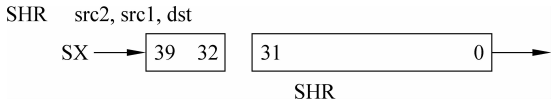


图 3.5 SHR 指令对 40 位长型数据的操作,位移次数由 SCR1 指定

在 SHR 指令里,源操作数 src2 可以是 32 或 40 位有符号数,src1 的低 6 位指定右移位数,将 src2 右移,得数放到 dst。移位时,最高位按符号位扩展。其他移位指令格式与 SHR 相似,其中 SHL 指令在左移时用 0 填补低位,SHRU 指令把 src2 视做无符号数,右移时,用 0 填补最高位。

算术左移指令 SHL 在左移过程中,其符号位有可能改变。带饱和的算术左移指令 SSHL 可用于防止这个问题产生。在 SSHL 指令情况下,src2 是 32 位有符号数,只要被 src1 指定移出的数位中有 1 位与符号位不一致,它就用与 src2 同符号的极大值填入 dst,并使 CSR 寄存器中的 SAT 位置位。

3. 位操作指令

在 C6000 公共指令集内对定点数的位域操作指令可分为 3 类:

(1) 位域清零/置位指令 CLR/SET;

(2) 带符号扩展与无符号扩展的位域提取指令 EXT/EXTU;

(3) LMBD 与 NORM 指令。

CLR、SET、EXT 和 EXTU 等指令的操作及指定操作域的方法相似。下面以 CLR 指令为例说明。CLR 指令的汇编语言有以下两种形式:

```
CLR  (.unit)  src2,  csta,  cstb,  dst
CLR  (.unit)  src2,  src1,  dst
```

前一种形式以常数 csta、cstb 指定位操作的域,后一种形式以 src1 的 bit0～bit4 代替 cstb,以 src1 的 bit5～bit9 代替 csta,这样用 src1 可动态地指定操作域,在指定范围内的位全被清零。CLR 指令操作内容如图 3.6 所示。

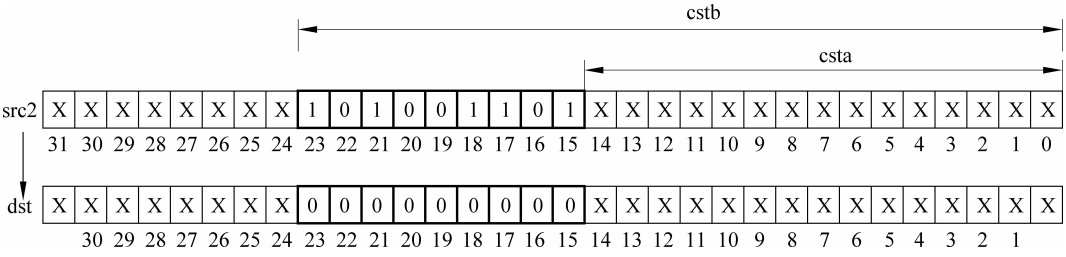


图 3.6 CLR 指令操作内容

LMBD 指令的汇编格式为:

```
LMBD        (.unit)    src1,src2,dst
```

其功能是寻找 src2 中与 src1 最低位(LSB)相同的最高位位置,并将其值(从左计起)返回送入 dst。图 3.7 是两个例子,两例都假设 src1 的 LSB 是 0。src2 分别如图 3.7(a)和图 3.7(b)所示。在图 3.7(a)情况下,返回值为 0;在图 3.7(b)情况下,返回值为 32。

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

(a) LMBD指令举例，返回值为0

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

(b) LMBD指令举例，返回值为32

图 3.7　LMBD 指令举例

NORM 指令用来检测源操作数中有多少个冗余的符号位。其指令格式为:

```
NORM(.unit)  src2,dst
```

图 3.8 中 NORM 指令的执行结果为 3。

src2	0	0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

图 3.8　NORM 指令举例

4. 比较及判别类指令

CMPEQ/CMPGT(U)/CMPLT(U)指令用于比较 2 个有/无符号数是否相等、大于或小于,若为真,则目的寄存器置 1;反之,目的寄存器置 0。应该注意,比较有符号数与无符号数大小的指令是不同的,要根据被比较对象的不同来选择不同的指令。

3.2.5　搬移类指令

搬移指令共有 MV、MVC、MVK 3 类。MV 指令用于在通用寄存器之间传送数据。MVC 指令用于在通用寄存器与控制寄存器之间传送数据,此条指令只能使用.S2 功能单元。MVK 类指令用于把 16 位常数送入通用寄存器。在 C6000 指令集内,只能往寄存器送 16 位常数,可选择 MVKH 或 MVKLH 指令向寄存器的高 16 位送数。

3.2.6　程序转移类指令

在 C6000 公共指令集内控制程序转移的有 4 类转移指令,其汇编语法格式如下:

(1) 用标号 label 表示目标地址的转移指令 B(.unit) label(.unit=.S1 或.S2);

(2) 用寄存器表示目标地址的转移指令 B .S2　src2;

(3) 从可屏蔽中断寄存器取目标地址的转移指令 B .S2 IRP;

(4) 从不可屏蔽中断寄存器取目标地址的转移指令 B .S2 NRP。

这 4 种转移指令只是目标地址不同,其执行过程相同。对用标号 label 表示目标地址的转移指令,在汇编阶段,汇编器(assembler)将计算从当前指令执行包到标号地址的相

对值，并将它填入指令代码。用寄存器表示目标地址的转移指令，其指定的通用寄存器的内容就是目标的绝对地址。第3、第4这两种转移指令与第2种相似，只是它们是从指定的中断寄存器取目标地址，适用于从中断返回的情况。

转移指令有5个指令周期的延迟间隙，即在转移指令进入流水线后，要再等5个周期才发生跳转。所以，转移指令后的5个指令执行包都进入CPU流水线，并相继执行。

例 3.7 转移指令及其延迟间隙的影响。

图3.9是一段汇编程序，最左2列是假设的程序地址。表3.8实际列出了从转移指令进入CPU流水线后每个周期的程序计数器的运行情况。可以看出，在跳转实际发生前，程序已进入LOOP段，所以LOOP段的前4个执行包会重复执行。如果不希望重复执行LOOP段的前4个执行包，也没有别的指令要执行，则可以把连续4个空操作指令NOP写在LOOP语句前，或简写为NOP 4。NOP不执行任何操作，仅用于填补取指包的空白及延迟间隙。多周期的空操作指令会影响转移指令的执行。用户可以有效地利用这一个延迟间隙，执行代码优化，提高代码效率。

```
0000 0000          B      .S1    LOOP
0000 0004          ADD    .L1    A1, A2, A3
0000 0008       || ADD    .L2    B1, B2, B3
0000 000C    LOOP: MPY    .M1X   A3, B3, A4
0000 0010       || SUB    .D1    A5, A6, A6
0000 0014          MPY    .M1    A3, A6, A5
0000 0018          MPY    .M1    A6, A7, A8
0000 001C          SHR    .S1    A4, 15, A4
0000 0020          ADD    .D1    A4, A6, A4
```

图 3.9 含有转移指令的汇编程序段

表 3.8 例 3.7 的汇编程序段执行时，其程序计数器的变化情况

周 期 数	程序计数器数值	操 作
Cycle0	0000 0000h	转移指令开始执行(取得目标地址)
Cycle1	0000 0004h	
Cycle2	0000 000Ch	
Cycle3	0000 0014h	
Cycle4	0000 0018h	
Cycle5	0000 001Ch	
Cycle6	0000 000Ch	转移到目标地址，执行目标代码
Cycle7	0000 0014h	

3.2.7 资源对公共指令集的限制

指令执行过程会占用一定的资源，并行执行的指令所需资源不能冲突。例如，在同一个指令周期，不能有2条指令对同一寄存器执行写操作。指令间资源上的冲突，不仅要考虑同一个执行包内指令间的冲突，还要考虑前后指令间可能的冲突。这是因为C6000采

用流水线结构，某些指令执行需要多个指令周期，前后指令间潜在的冲突有可能发生。

1. 使用相同功能单元的指令的限制

使用相同功能单元的 2 条指令不能被安排在同一执行包中。例如下面的执行包是无效的：

```
ADD    .S1   A0,A1,A2
|| SHR  .S1   A3,15,A4          ;S1 被 2 条指令同时使用，执行包无效
```

如果将上述执行包改成用不同的功能单元，下面的执行包就是有效的：

```
ADD    .L1   A0,A1,A2
|| SHR  .S1   A3,15,A4          ;使用 2 个不同的功能单元，执行包有效
```

2. 使用交叉通路(1X 和 2X)的限制

一个执行包内的一个功能单元(.L、.S 或.M 中的任一个)可以通过交叉通路从另一侧的寄存器组读取一个源操作数。使用同一条交叉通路的 2 条指令不能被安排在同一个执行包中。

下面的执行指令包是无效的：

```
ADD     .L1X   A0,B1,A1
|| MPY   .M1X  A4,B4,A5     ;在一个执行包内被 2 条指令同时使用，无效
```

上述执行包改成如下格式就是有效的：

```
ADD    .L1X    A0, B1, A1
|| MPY  .M2X   B4, A4, B2    ;.L1 使用 1X, .M2 使用 2X，执行包有效
```

3. 对 Load/Store 指令的限制

(1) Load/Store 指令所用的地址指针寄存器必须与所用的.D 功能单元处于同一侧。下面的执行指令包是无效的：

```
LDW    .D1     *A0,    A1
|| LDW  .D2     *A2,    B2     ;D2 必须使用 B 组的寄存器作地址指针
```

执行指令包经过如下的改动成为有效的：

```
LDW    .D1     *A0,    A1
|| LDW  .D2     *B0,    B2     ;D2 使用 B 组地址指针寄存器，正确
```

(2) 使用同一寄存器组作为目的地址/源地址的 2 条 Load/Store 指令不能被安排在同一个执行包中。下面的执行包，LDW 指令要读取数据到 A5，STW 指令要将 A6 数据输出，A5 和 A6 在同一个寄存器组内，是无效的：

```
LDW    .D1   *A4,A5
|| STW  .D2   A6, *B4   ;读取指令目的地址与存储指令的源地址都在 A 寄存器组，指令无效
```

下面 2 组执行包没有冲突，都是有效的执行包：

```
LDW    .D1   * A4, B5
|| STW  .D2   A6, * B4          ;读入到 B5,从 A6 存储,寄存器组不同,执行包有效
LDW    .D1   * A0,B2
|| LDW  .D2   * B0,A1           ;读入到 B2, A1,寄存器组不同,执行包有效
```

4. 使用长型数据(40 位)的限制

C62x/C67x 的.S 和.L 单元共用一套为长型源操作数的读通路和为长型结果的写寄存器通路,所以同一执行包只能容许每个寄存器组处理一个长型数据。

下面的指令执行包是无效的:

```
ADD    .L1   A5: A4,A1, A3: A2
|| SHL  .S1   A8,A9, A7: A6      ;2 个长数据同时写入 A 组寄存器
```

指令执行包经过如下的改动成为有效的:

```
ADD    .L1    A5: A4,A1,A3: A2
|| SHL    .S2    B8. B9. B7: B6    ;每一组寄存器只有 1 个长数据写
```

因为 C62x/C67x 的.S 和.L 单元的长数据读通路与数据存储通路共用,所以涉及同一.S 单元或.L 单元的长数据读操作和存储操作不能安排在同一个执行包中。下面的执行包是无效的:

```
ADD    .L1    A5: A4,A1,A3: A2
|| STW    .D1    A8, * A9         ;长数据读操作与同组存储操作冲突
```

执行包经过如下改动成为有效的:

```
ADD    .L1    A4,A1, A3: A2
|| STW    .D1   A8, * A9          ;没有长数据读操作,无冲突
```

5. 对寄存器读取的限制

对同一寄存器在 1 个指令周期读取多于 4 次是不允许的,条件寄存器不在此限制之列。

下述代码序列在 1 个指令周期内对寄存器 A1 进行了 5 次读取,是无效的:

```
MPY    .M1   A1, A1, A4
|| ADD    .L1   A1, A1, A5
|| SUB    .D1   A1, A2, A3       ;在 1 个指令周期内对寄存器 A1 进行 5 次读取
```

如下的代码序列是有效的:

```
MPY    .M1   A1,A1,A4
|| ADD    .L1   A0,A1,A5
|| SUB    .D1   A1,A2,A3        ;只对寄存器 A1 进行 4 次读取
```

6. 对寄存器存储的限制

在 1 个指令周期,不能同时存在 2 条写入同一寄存器的指令。具有同一目的地址的

2 条指令只要向该目的寄存器的写操作不在同一个指令周期发生，就可以安排并行。例如，第 i 周期的 MPY 指令与其后第 $i+1$ 周期的 ADD 指令不能写入相同的寄存器，因为这 2 条指令的写操作都在第 $i+1$ 周期发生。因此，下面的代码序列是无效的，除非在 MPY 指令后有跳转操作发生，能够阻止 ADD 指令的执行：

```
MPY    .M1   A0,A1,A2
ADD    .L1   A4,A5,A2
```

然而，下面的代码序列却是有效的：

```
ADD    .L1   A4,A5,A2
|| MPY  .M1   A0,A1,A2
```

下面给出了多种可能发生写冲突的例子。汇编器(assembler)可能对某些错误检验不出来，对此，编程人员需特别注意。

```
L1:          ADD   .L2   B5,B6,B7
||           SUB   .S2   B8, B9,B7          ;执行包 L1 有冲突，能被汇编器发现
L2:          MPY   .M2   B0, B1,B2
L3:          ADD   .L2   B3, B4,B2          ;执行包 L2、L3 有冲突，汇编器不能发现
L4: [!B0]    ADD   .L2   B5, B6,B7
||  [B0]     SUB   .S2   B8, B9,B7          ;汇编器能判定本执行包无冲突
L5: [!B1]    ADD   .L2   B5, B6, B7
||  [B0]     SUB   .S2   B8, B9, B7         ;汇编器无法判定本执行包是否有冲突
```

执行包 L1 中，指令 ADD 与 SUB 写入同一寄存器，这个冲突能被汇编器发现。执行包 L2 中的 MPY 指令与 L3 中的 ADD 指令可能同时写入寄存器 B2，如果存在转移指令使执行包 L2 之后是其他执行包而非 L3 的话，则写冲突不会发生，因此 L2 与 L3 之间潜在的写冲突不会被汇编器检测出。L4 中的指令不会发生写冲突，因为它们是互斥的。相比之下，L5 中的指令既可能是互斥的，也可能不是互斥的，汇编器无法判断，也就不会指出有错。如果程序的流程安排确实出现了写冲突，程序的执行结果将不确定。

3.2.8 浮点运算指令集

TMS320C67xx 是 C6000 系列中的浮点芯片系列。除了能执行公共定点指令外，它增加了 30 余条指令。增加的指令主要是单/双精度浮点运算指令，还有双精度数据的读取与存储指令。另外，C67xx 采用与 IEEE 标准相同的浮点数表示法，有 32 位单精度与 64 位双精度两种。

3.3 汇编、线性汇编和伪指令

3.3.1 汇编代码结构

汇编语言必须是 ASCII 码文件，扩展名必须是 sa，用作汇编优化器的输入文件。C6000 任意一行汇编代码可能包括 7 个项目：标号、并行符号、条件、指令、功能单元、操

作数和注释。各项在一行汇编代码中的位置如图 3.10 所示。

```
label: parallel bars[||]  [condition] instruction  unit operands;comments
```

图 3.10 线性汇编语句

1. 标号

标号用来定义一行代码或一个变量,它代表一条指令或数据的存储地址,标号后面的冒号是可选的。标号必须满足下列条件:

(1) 标号的第 1 个字符必须是字母或下划线后跟一个字母。

(2) 标号的第 1 个字符必须在文件的第 1 列。

(3) 标号最多可包含 32 个字母字符。

(4) 并行指令不能使用标号。

2. 并行符号

若一条指令与前面的指令并行执行,这条指令用并行符号||表示。如果一条指令的并行符号处为空白,表示这条指令不与前面的指令并行执行。

3. 条件

C62xx 中有 A1、A2、B0、B1 和 B2 5 个寄存器用于条件寄存器,C64xx 中还可以使用 A0 寄存器。方括号内寄存器的值是指令执行的条件。所有 C6000 指令都是条件指令,其执行规则如下,见表 3.9。

表 3.9 指令条件的使用

指令的条件表示	指令执行条件	指令不执行条件
[A1]	A1!=0	A1=0
[!A1]	A1=0	A1!=0

(1) 如果指令没有指出条件,指令总被执行。

(2) 如果给定条件,当条件为真,指令执行。

(3) 如果给定条件,当条件为假,指令不执行。

4. 指令

汇编代码的指令包括伪指令和命令助记符:

(1) 伪指令用来在汇编语言中控制汇编过程或定义数据结构,所有伪指令都以圆点打头。

(2) 命令助记符代表有效微处理器命令,它执行程序操作。助记符必须在第 2 列或第 2 列以后的位置开始。

5. 功能单元

C6000 有 8 个功能单元,不同类型的功能单元完成不同的功能任务。功能单元都以圆点(.)开始,后跟一个功能单元分类符。与汇编语句相似,功能单元用来说明指令所使用的资源。功能单元也是可选项,其可选方式有 4 种:

(1) 指定具体使用的功能单元(如.D1)。

(2) 指定.D1、.D2 后面再加 T1 或 T2,指定使用哪一侧的寄存器,例如下面的指令用.D1 产生存储器地址,T2 指定 B 侧寄存器(dst)。

```
LDW     .D1T2   *src, dst
```

(3) 可指出功能单元类型(如.M),由汇编器优化器安排特定功能单元(如.M2)。

(4) 不指出功能单元或仅指定数据通道,由汇编优化器根据助记符安排功能单元。

用户指定功能单元,实际上是命令汇编优化器如何分配资源:分配功能单元和寄存器的使用。如果用户的资源使用不当,就限制了汇编优化器发挥作用。所以,用户对是否指定功能单元要很慎重。

6. 操作数

操作数由常数、符号以及常数与符号构成的表达式组成。操作数之间必须用逗号隔开。在线性汇编代码中,指令对操作数有如下要求:

(1) 所有指令都需要 1 个目的操作数。

(2) 多数指令需要 1 个或 2 个源操作数。

(3) 目的操作数必须与 1 个源操作数在同一寄存器组。

(4) 2 个源操作数可以在同一寄存器组,也可在不同寄存器组。

当一条指令中的一个操作数来自另一侧寄存器组时,该指令所使用的功能单元包含一个 X 符号,该符号表示这条指令使用一个交叉通路。例如:

```
ADD  .L1    A0,A1,A3
ADD  .L1X   A0,B1,A3
```

C6000 指令使用 3 种类型操作数访问数据:

(1) 寄存器操作数　指出一个包含数据的寄存器。

(2) 常数操作数　指定汇编代码内的数据。

(3) 指针操作数　包含数据的地址,仅读取和存入指令需要使用指针操作数来实现存储器和寄存器之间的数据传送。

在线性汇编代码中,如果未指定 CPU 寄存器(未使用 C6000 的专用寄存器名),则应以在伪指令.reg 中定义的变量名代替。

7. 注释

与所有编程语言一样,汇编代码中使用注释对代码进行说明。在汇编代码中使用注释应遵循如下规则:

(1) 当前面使用分号(;)时,注释可以在任何一列开始。

(2) 当前面使用星号(*)时,注释必须在第 1 列开始。

(3) 注释不是必需的,但为提高代码可读性,建议使用注释。

3.3.2　线性汇编语言结构

为了提高代码性能,对影响速度的关键 C 代码段可以用线性汇编重新编写。线性汇

编文件是汇编优化器的输入文件。线性汇编代码类似于前面介绍的C6000汇编代码，不同的是在线性汇编代码中不需要给出汇编代码必须指出的所有信息，线性汇编代码对这些信息可进行一些选择，也可以由汇编优化器确定。下面是线性汇编不需要给出的信息：

(1) 使用的寄存器；

(2) 指令的并行与否；

(3) 指令的延迟周期；

(4) 指令使用的功能单元。

如果在代码中没有指定这些信息，汇编优化器会根据代码的情况确定这些信息。与其他代码产生工具一样，有时需要对线性汇编代码进行修改直到其性能令人满意为止。在修改过程中，可能要对线性汇编添加更详细的信息，如指出应该使用哪个功能单元等。

3.3.3 汇编优化器伪指令

1. 汇编优化器伪指令特点

线性汇编文件中必须包含一些汇编优化器伪指令。使用汇编优化器伪指令可以区分线性汇编代码和正规汇编代码，还能为汇编优化器提供有关代码的其他信息。线性汇编伪指令特点如下：

(1) 线性汇编文件的扩展名必须是.sa。

(2) 线性汇编代码应该包括.cproc和.endproc命令。.cproc和.endproc命令限定了需要优化器优化的代码段，.cproc放在这段代码的开始位置，.endproc放在这段代码的结尾。用这种方式可以设置需要优化的汇编代码段，如程序或函数等。

(3) 线性汇编代码中可能包含.reg命令，这个命令允许使用将要存入寄存器的数值的描述名字。当使用.reg时，汇编优化器为数值选择一个寄存器，这个寄存器与对该值进行操作的指令所选择的功能单元是一致的。

(4) 线性汇编代码中还可能包括.trip命令，这个命令用于指出循环的迭代次数。

下面是一段线性汇编语言程序实现的点积运算：

```
_dotp:  .cproc  a_0, b_0                ;优化器开始优化代码

        .reg  a_4, b_4, cnt, tmp        ;定义变量寄存器
        .reg  prod1, prod2, prod3, prod4
        .reg  valA, valB, sum0, sum1, sum

        ADD     4,  a_0, a_4
        ADD     4,  b_0, b_4
        MVK     100,cnt
        ZERO    sum0
        ZERO    sum1
Loop:   .trip  25                       ;设置循环次数

        LDW   *a_0++[2],  valA          ;load a[0-1]
```