第3章 指令系统与汇编语言程序设计

S12X 系列单片机具有强大的指令系统，以高速 16 位 S12X CPU 核为基础。S12X CPU 指令集完全兼容以前的 CPU12 指令集，并增加了一些新指令，支持 16 位数据通道，支持高效快速的算术操作，支持奇数字节指令（包括很多单字节指令），能更有效地利用 ROM 空间，指令缓冲队列技术可以使 CPU 在最短时间内寻址多字节的机器码；寻址方式灵活多样，特别是具有强大的变址寻址能力，允许堆栈指针 SP 和程序计数器 PC 作为变址寄存器，通过累加器 A、B、D 提供偏移量以及自动调整变址指针等，这大大增强了寻址的灵活性。此外，S12X CPU 指令类别齐全、功能丰富，包含数据传送、算术运算、逻辑运算、位操作、移位、控制、全局地址读写、特殊等类别，共有多达几百条不同功能的指令（S12X 完备指令详解参见 S12XCPUV1 手册）。

3.1 CPU 寄存器

在 MCU 的程序设计中，除了需要处理、操作 RAM 数据外，还要使用到大量的各种用途的寄存器。为了区分，本书将 S12X MCU 的寄存器分为两个类别：MCU 寄存器和 CPU 寄存器（如图 3-1 所示）。

MCU 寄存器就是指 S12X 的众多的 I/O 接口、功能模块和配置管理的几百个寄存器，占用 2KB 的存储器映射空间，各个寄存器功能固定、各有侧重，都有各自的有自明含义的寄存器名，用来完成对 MCU 的各个资源模块的管理、设置或写入/读出操作。MCU 寄存器也可称为功能寄存器，包含大量的 I/O 寄存器，如第 2 章所述，它们被安排在存储器映射空间的前 2KB 处。

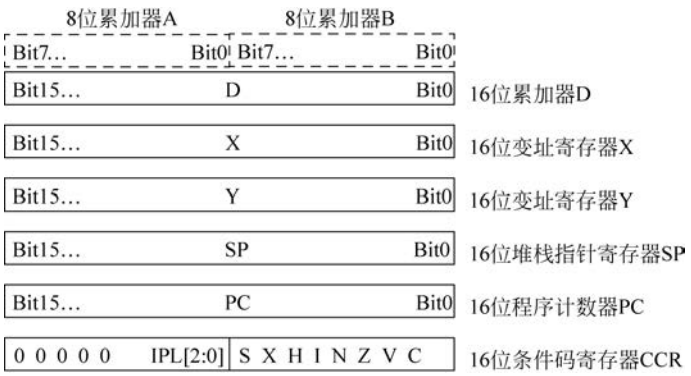


图 3-1 CPU 寄存器

CPU 寄存器专指 S12X CPU 内部寄存器，简称 CPU 寄存器，只有 D、X、Y、SP、PC 和 CCR 这 6 个寄存器（可以看成 7 个）。它们直接与 CPU 的 ALU 相连，具有比 RAM 存储器更快的读/写速度。CPU 寄存器在程序代码中频繁使用，用于直接参与操作、暂存中间数据、存放操作结果、作为寻址指针等。编程时需要使用这些 16 位 CPU 寄存器，如图 3-1 所示。下面分别具体介绍。

1. 累加器 D

S12X CPU 有 2 个 8 位累加器：累加器 A 和累加器 B。累加器 A 和累加器 B 合起来就是一个 16 位累加器 D（A、B，Accumulator），其高 8 位在累加器 A，低 8 位在累加器 B。累加

器 A、B 是指令系统中最灵活、最常用的寄存器，各种寻址方式均可对之寻址，复位时，累加器的内容不受影响。累加器 D、A、B 实际上是同一个寄存器。累加器 D(A、B)主要用于存放操作数和运算结果，对于大多数算术运算指令，累加器用作目的寄存器。

2. 变址寄存器 X、Y

2 个 16 位变址寄存器 X、Y(Index Register X、Y)主要用于寻址操作，作为地址指针使用；它们也可用于临时存放数据并参与运算。在变址寻址方式下，变址寄存器的内容加上 5 位、9 位或 16 位的值或者加上一个累加器中的内容得到指令操作的有效地址。寄存器 X、Y 的内容不受复位影响。

3. 堆栈指针寄存器 SP

S12X CPU 的 16 位堆栈指针寄存器 SP(Stack Pointer)主要用于堆栈管理，服务于中断和子程序调用。堆栈指针 SP 采用递减结构，即进栈时 SP 减 1，出栈时 SP 加 1，堆栈内容后进先出。堆栈指针 SP 总是指向堆栈区的顶部。SP 也可在 8 位或 16 位的偏移量寻址方式中作为变址寄存器。SP 的内容不受复位影响。

4. 程序计数器 PC

程序计数器 PC(Program Counter)是一个 16 位指针，寻址范围 64KB，其内容始终指向程序序列中下一条将要执行的指令地址，包括在执行转移指令时存放转移地址、在执行中断时存放中断子程序入口地址。用户可以读取 PC 指针，但不能直接向 PC 指针进行写入操作。复位后，PC 自动回到默认状态。PC 是特殊的寄存器，一般不能挪作他用，但可以像 SP 一样，作为变址寄存器使用。

5. 条件码寄存器 CCR

在 S12 CPU 中它是 8 位寄存器，在 S12X CPU 中它已被扩展成了 16 位寄存器。为了兼容以前的指令，16 位 CCR 寄存器也可以看成 2 个 8 位寄存器：CCRH 和 CCR。

CCR(Condition Code Register)也称程序状态寄存器，包括 5 个算术特征位：H、N、Z、V、C，它们反映上一条指令执行结果的特征；还有 3 个 MCU 控制位：STOP 指令控制位 S 和中断屏蔽位 X、I，这 3 位通常由软件设定，控制 S12X CPU 的操作。其具体定义如下。

条件码寄存器——CCRH：CCR

复位默认值：0000 0000 1101 0000B

读写	Bit15～11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
R	0	IPL[2:0]			S	X	H	I	N	Z	V	C
W												

IPL[2:0]：记录当前最新中断的较高优先级级别，当 RTI 指令后自动将入栈的原先的中断优先级级别恢复。它是 S12X CPU 较以前 S12 CPU 所增加的。

S：停止模式(STOP 指令)禁止位。该位置 1 将禁止 CPU 执行 STOP 指令。

X：$\overline{\text{XIRQ}}$ 中断屏蔽位。该位置 1 将屏蔽来自 $\overline{\text{XIRQ}}$ 引脚的中断请求，复位默认值为 1。

H：辅助进位标志位。该位 BCD 操作时累加器 A 的 Bit3 向 Bit4 进位。

I：中断屏蔽位。该位置 1 将屏蔽所有的可屏蔽中断源，复位默认值为 1。

N：负标志位。当操作结果为负时，该位置 1。

Z：零标志位。当操作结果为 0 时，该位置 1。

V：溢出标志位。当操作结果出现补码溢出时，该位置 1。

C：进位/借位标志位。当加法运算产生进位或者减法运算产生借位时，该位置 1。一些测试、跳转、移位操作或者直接针对 C 的指令也可改变 C 的值。

3.2 寻址方式

寻址方式是指 CPU 在执行指令时确定操作数所在的单元地址的方式。在 MCU 中，指令是对数据的操作，通常把指令中所要操作的数据称为操作数，CPU 所需的操作数可能来自寄存器、指令代码或存储单元，CPU 在执行指令时（NOP 指令除外），都要先找到操作数的地址，从中得到内容，然后再完成相应的动作。显然，寻址方式越多，指令系统的功能就越强、灵活性越大。

S12X CPU 指令共可综合为 9 种寻址方式。

1. 隐含寻址

指令本身已经隐含了操作数所在地址的寻址方式。指令的操作数隐含在助记符中或无需操作数，这类指令一般为单字节指令。例如，ROLA、PSHB、INX 等指令，操作数隐含是 CPU 寄存器 A、B、X 中；NOP 指令无需操作数。

2. 立即寻址

指令的操作数在指令中立即给出，在汇编语言中用“#”号代表一个立即数。立即寻址类指令常用于给某一寄存器赋值。例如：

```
LDAA   #$8F                    ;将十六进制数 8F 立即装载到 A 中
LDX    #1234                   ;将十进制数 1234 立即装载到 X 中
```

3. 直接寻址

指令中直接给出操作数的地址。这种方式可以直接访问存储器空间中 $0000～$00FF 段的 256 个单元，直接寻址方式默认的地址高 8 位为 $00，指令中只需给出单字节地址。在 S12X 单片机默认的存储器地址分配中，这一段是 MCU 寄存器地址，因此可以使用直接寻址访问这些寄存器。例如：

```
LDAA $55                       ;将 8 位地址 $0055 单元的内容装载到 A 中
```

4. 扩展寻址

扩展寻址与直接寻址类似，指令中给出操作数地址，只是这时的地址是 16 位地址，可以寻址整个 64KB 地址空间，寻址范围远大于立即寻址方式。例如：

```
LDAA $200A                     ;将 16 位地址 $200A 单元的内容装载到 A 中
```

5. 相对寻址

相对寻址只用于转移指令，用于程序跳转和子程序调用。在程序中写出需要跳转的目的地址的标号，汇编语言程序会自动计算出相对转移地址并完成跳转，例如：

```
BRA     LABEL                ;无条件跳转到 LABEL 标号的地址处
BRA     *                    ;无条件跳转到当前地址(*),此语句实现原地等待
BCC     DONE                 ;若 C 标志为 0,则跳转到 DONE 标号的地址处
```

6. 变址寻址

变址寻址方式以变址寄存器 X、Y 或者 SP、PC 寄存器的内容为基址，再加上或减去一个偏移量，作为操作数的最终地址。这个偏移量可以是 5 位(－16～15)、9 位(－256～255，IDX1)或 16 位(－32768～32767，IDX2)常数，也可以是 0，对应指令的占用字节数分别为 2 字节、3 字节和 4 字节，功能相同。例如：

```
STD  7, X                 ;5 位常数偏移量,偏移量为 7
                          ;X 寄存器内容加上 7 作为地址,2 字节的 D 内容存储到这里
                          ;其中,低地址字节存 D 的 A,高地址字节内容存 D 的 B
LDAA 0, X                 ;5 位常数偏移量,偏移量为 0
                          ;X 寄存器内容作为地址,其指向单元的内容装载到 A 中
LDAB  -$FA, PC            ;9 位常数偏移量,偏移量为 -$FA
                          ;PC 寄存器内容减去 $ FA 作为地址,其指向单元内容装载到 B 中
LDAA 1000, X              ;16 位常数偏移量,偏移量为 1000
                          ;X 寄存器内容加 1000 作为地址,其指向单元内容装载到 A 中
```

7. 累加器变址寻址

累加器变址寻址简称为 IDX，这种变址寻址的偏移量来自累加器 A、B 或 D，基址寄存器内容加上这个无符号偏移量构成操作数的地址。例如：

```
LDAA  D,X                 ;将 X 值加上 D 值作为地址,其指向的字节内容装载到 A
LDAD  A,Y                 ;将 Y 值加上 A 值作为地址,其指向的字内容装载到 D
```

8. 自加自减的变址寻址

这是有附带功能的变址寻址，这种寻址方式提供 4 种方式(先加、先减、后加、后减)去自动改变变址寄存器值，加、减数值的范围是 1～8，然后确定操作数的地址。变址寄存器可以是 X、Y 和 SP，这种变址寻址对于连续数据块的操作十分方便，适合字节、字、双字、四字变量的快速定位。例如：

```
STAA  1, - SP             ;SP 寄存器先减 1,然后将 A 内容存储到 SP 指向的单元
                          ;等效于入栈指令 PSHA
LDX   2,SP +              ;SP 指向的字内容先装载到 X 寄存器,然后 SP 寄存器加 2
                          ;等效于出栈指令 PULX
MOVW  2, X + , 4, + Y     ;X 寄存器内容指向的字数据传送到 Y + 4 指向的地址单元
                          ;传送后 X 自动后加 2,传送前 Y 已经自动先加 4
```

9. 间接变址寻址

该寻址方式将变址寄存器的值加上一个 16 位偏移量或累加器 D 的值，形成一个地址，该地址中的内容并不是实际操作数，而是最终操作数的有效地址。例如：

```
LDAA  [1000, X]           ;((1000 + X))→A
                          ;X + 1000 的地址单元内容作为地址,其指向内容装载到 A 中
LDAA  [D, Y]              ;((D + Y))→A
                          ;Y + D 的地址单元内容作为地址,其指向内容装载到 A 中
```

以上寻址方式中，后面的几种变址寻址是 S12X CPU 的重要、高效的寻址方式，相当于

C语言中的指针操作，而间接变址寻址则相当于C语言中的“指针的指针”。变址寻址方式的通常表象是：若指令操作码后面跟的操作数是变址寄存器X、Y或者SP、PC寄存器，则该指令的寻址方式就是变址寻址(LEA指令除外)，X、Y、SP、PC将扮演指针的作用。

另外，若把一个16位数写入存储器，则高8位在存储器低地址处，低8位在存储器高地址处。NXP公司的CPU将16位、32位数据与存储器字节的对应关系规定为高位低地址、低位高地址[即大端(big endian)模式]，这和Intel公司的规定正好相反，Intel公司CPU系列使用高位高地址、低位低地址的方式[即小端(little endian)模式]。

注意：在NXP汇编语言程序中，表示16位数据的前缀是$，在C语言编程环境中则使用0x前缀。“；”是汇编语言编程的注释标志。

3.3 指令概览

S12X CPU的常用指令按照操作类别大致可以分为数据传送类指令、算术运算类指令、逻辑运算类指令、程序控制类指令、CPU控制类指令、中断类指令、全局读写指令和其他指令。每一类别中又有很多小类和多种指令，指令条数繁多，还涉及各种寻址方式，难以尽述，也不需要使用者全面掌握。本节内容旨在帮助使用者对S12X的一般指令集的功能、特点和使用方法有一个基础认识，各指令的使用可以在实际编程应用中随时查阅。其中左边3列是指令助记符、操作功能和寻址方式，第4列是指令编译后形成的机器码，第5列是指令的执行周期数及每个周期的具体动作(每个字母代表1个时钟周期，分别表示读、写、空闲等)，最右边2列是指令影响标志位的情况(△表示影响该位，－表示不影响该位，1表示该位置1，0表示该位清零)。

S12X CPU的指令采用NXP公司自定义的指令助记符方法，指令的英语名称上能够反映指令的基本功能，需要在使用的过程中体会理解，例如：

```
CLI    = CLear I
LDAA   = LoaD Accumulator A
STAB   = STore Accumulator B
TAB    = Transfer A to B
MOVB   = MOVe Byte
BEQ    = Branch EQual zero
PSHA   = PuSH A
RTI    = ReTurn of Interrupt
GLDAD  = Global LoaD Accumulator D
…
```

1. 数据传送类指令

数据传送指令包括寄存器与寄存器之间、寄存器与存储器单元之间、存储器单元与存储器单元之间的传送形式。在后续的指令表中，()表示寄存器或存储器内容，M表示存储器地址，M：M+1表示连续的两个地址；(M：M+1)表示M、M+1两个相邻存储单元的内容组成的一个字内容，(M)为高位字节内容，即高8位数存放在低位地址单元存储器中，H、L分别表示高8位和低8位，其他类同。

1) 寄存器装载指令

寄存器装载(Load)指令将存储器的内容复制到寄存器中，而源存储器中的内容不变。

该类指令(LEA 指令除外)会自动更新 N、Z 标志位,V 标志位清零。寄存器装载指令如表 3-1 所示。

表 3-1　寄存器装载指令

助　记　符	功　　能	操　　作
LDAA	Load A	(M)→A
LDAB	Load B	(M)→B
LDD	Load D	(M：M+1)→(A：B)
LDS	Load SP	(M：M+1)→SPH：SPL
LDX	Load index register X	(M：M+1)→XH：XL
LDY	Load index register Y	(M：M+1)→YH：YL
LEAS	Load effective address into SP	Effective address→SP
LEAX	Load effective address into X	Effective address→X
LEAY	Load effective address into Y	Effective address→Y

指令使用举例如下：

```
LDAA   # $ 3F      ;立即数 $ 3F 装载到累加器 A 中,操作后 N = 0,Z = 0,V = 0
LDAB   $ 20FA      ;把 $ 20FA 单元的内容取到累加器 B 中
LDD    8, X        ;变址寄存器 X 的内容加 8 作为地址,对应单元的内容送 D 的高位(A),
                   ;下一个单元的内容送到 D 的低位(B)
LDS    $ 1000,Y    ;变址寄存器 Y 的内容加 $ 1000 作为地址,对应单元的内容送到 SP 高 8 位
                   ;下一个单元的内容送到 SP 低 8 位
LDX    A, PC       ;将(PC + A)单元的内容送 X 高 8 位,下一个单元的内容送到 X 低 8 位
LDY    2, SP +     ;将 SP、SP + 1 对应单元的内容分别送 Y 的高、低 8 位,然后 SP 内容加 2
LEAS   2, X        ;X 内容加 2 后送到 SP;基址 + 偏移直接形成了有效地址
LEAX   B, Y        ;Y 的内容加上 B 的内容送到 X;基址 + 偏移直接形成了有效地址
LEAY   D, SP       ;SP 的内容加上 D 的内容送到 Y;基址 + 偏移直接形成了有效地址
```

2) 寄存器存储指令

寄存器存储(Store)指令将寄存器的内容复制到存储器中,而源寄存器的内容不变。该类指令会自动更新 N、Z 标志位,V 标志位清零。寄存器存储指令如表 3-2 所示。

表 3-2　寄存器存储指令

助　记　符	功　　能	操　　作
STAA	Store A	(A)→M
STAB	Store B	(B)→M
STD	Store D	(A)→M,(B)→M+1
STS	Store SP	(SPH：SPL)→M：M+1
STX	Store X	(XH：XL)→M：M+1
STY	Store Y	(YH：YL)→M：M+1

指令使用举例如下：

```
STAA   $ 10            ;将累加器 A 的内容保存到 $ 0010 单元
STD    - $ 2000, X     ;将累加器 D 的 A、B 内容保存到 X - $ 2000、X - $ 2000 + 1 单元
STY    2, + SP         ;SP 内容先加 2,然后将 Y 的高、低 8 位分别保存到 SP、SP + 1 单元
```

3) 寄存器传输指令

寄存器传输(Transfer)指令复制一个寄存器内容到另一个寄存器,而源寄存器的内容不变。寄存器传输指令如表 3-3 所示。

S12XCPU 指令主要使用 3 个：TAB、TBA 和 TFR。TAB、TBA 指令会影响 N、Z 和 V 标志位。TFR 是一个寄存器与寄存器之间通用传输指令，TFR 指令不影响标志位。

TFR 指令可以在 16 位和 8 位寄存器之间传送，它的处理方法是：8 位到 8 位或 16 位到 16 位时直接传输；8 位到 16 位时高 8 位补 0 变成 16 位后传输；16 位到 8 位时舍弃高位，只传输低位。

表 3-3　寄存器传输指令

助　记　符	功　　能	操　　作
TAB	Transfer A to B	(A)→B
TBA	Transfer B to A	(B)→A
TFR	Transfer register to register	(A,B,CCR,D,X,Y,SP)→A,B,CCR,D,X,Y,SP

指令使用举例如下：

```
TFR   A, X                    ;将累加器 A 的内容传输到 X 的低 8 位,X 高 8 位清零
```

建议：平常使用时，尽量不使用位数不匹配的方式进行寄存器传输。

4）寄存器交换指令

寄存器交换(Exchange)指令的作用是交换一对寄存器的内容。交换指令的运行不影响标志位。S12X CPU 的指令实际只有 EXG 这一条，其他 2 条指令是为了兼容以前的 S12 CPU。EXG 指令中，当第一个操作数是 8 位，第二个操作数是 16 位，8 位数补 0 扩展后复制到 16 位寄存器中，而 16 位数的低 8 位复制到 8 位寄存器中，其高 8 位被舍弃。寄存器交换指令如表 3-4 所示。

表 3-4　寄存器交换指令

助　记　符	功　　能	操　　作
EXG	Exchange register to register	(A,B,CCR,D,X,Y,SP)←→(A,B,CCR,D,X,Y,SP)
XGDX	Exchange D with X	(D)←→(X)
XGDY	Exchange D with Y	(D)←→(Y)

指令使用举例如下：

```
EXG   X, SP                   ;SP 与 X 内容交换
EXG   A, B                    ;A 与 B 内容交换
EXG   B, X                    ;B 内容送到 X 低 8 位, $00 送 X 高 8 位,X 低 8 位送 B
```

建议：平常使用时，尽量不使用位数不匹配的方式进行寄存器交换。

5）存储器数据传送指令

存储器数据传送(Move)指令将源操作数(1 个字节或字)传送到目的地址，源操作数不变。直接寻址、扩展寻址、变址寻址的 6 种组合允许指定源地址和目的地址。源操作数可为立即数或存储器数，目的地址不能是立即数。存储器数据传送不影响标志位。这类指令共有 2 个助记符，是不经过寄存器的直接存储区数据传送。存储器数据传送指令如表 3-5 所示。

表 3-5　存储器数据传送指令

助　记　符	功　　能	操　　作
MOVB	Move byte (8bit)	(M1)→M2
MOVW	Move word (16bit)	(M1：M1+1)→M2：M2+1

指令使用举例如下：

```
MOVB   # $ 5A,6, Y              ;将立即数 $ 5A 送到(Y + 6)单元
MOVW   # $ 103F, $ 2011         ;将立即数 $ 10、$ 3F 分别送( $ 2011)、( $ 2012)单元
MOVB    $ 103F, $ 2011          ;将( $ 103F)单元内容送到( $ 2011)单元
MOVB   1, X, 1, Y               ;将(X + 1)内容送到(Y + 1)单元
```

6）堆栈操作指令

有关堆栈的指令分为两种：一种是堆栈指针的操作指令，用于特殊形式的数据传送，指令形式与变址寄存器 X、Y 的操作类似；另一种是堆栈操作（入栈、出栈）指令，用来从系统堆栈中保存和恢复信息，遵从“先入后出、后入先出”的堆栈原则。堆栈作为存储器单元的表现形式，本质上就是一个 RAM 存储区，因此也可以通过其他寻址方式直接访问堆栈内的数据。除了 PULC 外，其他均不影响标志位。堆栈操作指令如表 3-6 所示。

表 3-6　堆栈操作指令

助　记　符	功　　能	操　　作
PSHA	Push A	(SP)－1→SP；(A)→M(SP)
PSHB	Push A	(SP)－1→SP；(B)→M(SP)
PSHC	Push CCR	(SP)－1→SP；(CCR)→M(SP)
PSHCW	Push CCRH：CCR	(SP)－2→SP；(CCRH：CCR)→M(SP)：M(SP＋1)
PSHD	Push D	(SP)－2→SP；(A：B)→M(SP)：M(SP＋1)
PSHX	Push X	(SP)－2→SP；(X)→M(SP)：M(SP＋1)
PSHY	Push Y	(SP)－2→SP；(Y)→M(SP)：M(SP＋1)
PULA	Pull A	(M(SP))→A；(SP)＋1→SP
PULB	Pull B	(M(SP))→B；(SP)＋1→SP
PULC	Pull CCR	(M(SP))→CCR；(SP)＋1→SP
PULCW	Pull CCRH：CCR	(M(SP)：M(SP＋1))→CCRH：CCR；(SP)＋2→SP
PULD	Pull D	(M(SP)：M(SP＋1))→A：B；(SP)＋2→SP
PULX	Pull X	(M(SP)：M(SP＋1))→X；(SP)＋2→SP
PULY	Pull Y	(M(SP)：M(SP＋1))→Y；(SP)＋2→SP

指令使用举例如下：

```
PSHB          ;累加器 B 入栈
PSHD          ;累加器 D 入栈,B 先入栈,A 后入栈
PULX          ;从堆栈中弹出 2 字节到寄存器 Y,第 1 字节送到 Y 高 8 位,第 2 字节送到 Y 低 8 位
PULC          ;从堆栈中弹出 1 字节到 CCR
```

2. 算术运算类指令

S12X 算术运算指令不仅包括加、减、乘、除、比较以及十进制调整的 BCD 指令，还支持乘加运算、表插值运算等，并设置了符号扩展指令。算术指令影响标志位。

1）加、减法指令

有符号和无符号的 8 位或 16 位加、减法运算能在寄存器之间或寄存器和存储器之间执行。带进位的加、减法指令能实现多字节的准确运算。加减运算结果仍然保存在助记符所包含的寄存器中。加、减法指令如表 3-7 所示。

表 3-7 加、减法指令

助 记 符	功 能	操 作
加法指令		
ABA	Add B to A	(A)+(B)→A
ABX	Add B to X	(B)+(X)→X
ABY	Add B to Y	(B)+(Y)→Y
ADCA	Add with carry to A	(A)+(M)+C→A
ADCB	Add with carry to B	(B)+(M)+C→B
ADDA	Add without carry to A	(A)+(M)→A
ADDB	Add without carry to B	(B)+(M)→B
ADDD	Add to D	(A：B)+(M：M+1)→A：B
减法指令		
SBA	Subtract B from A	(A)−(B)→A
SBCA	Subtract with borrow from A	(A)−(M)−C→A
SBCB	Subtract with borrow from B	(B)−(M)−C→B
SUBA	Subtract memory from A	(A)−(M)→A
SUBB	Subtract memory from B	(B)−(M)→B
SUBD	Subtract memory from D (A：B)	(A：B)−(M：M+1)→A：B

指令使用举例如下：

```
ADDA   # $35      ;累加器 A 的内容加上立即数 $35,结果存回 A 中
ADCB   8, Y       ;累加器 B 的内容加上(Y+8)单元的内容,再加上进位位,结果存回 B 中
ADDD   $200A      ;累加器 D 的内容加上( $200A)单元的内容,结果存回 D 中
SUBA   $2011      ;累加器 A 的内容减去( $2011)单元的内容,结果存回 A 中
SBCB   $80, X     ;累加器 B 的内容减去(X+ $80)单元的内容,再减去进位位,结果存回 B 中
```

2) 加 1、减 1 指令

加 1 和减 1 指令是优化的 8 位和 16 位加、减法操作，实际上是对寄存器 X、Y、A、B 或存储器字节的加 1、减 1 计算。这类指令通常用来实现计数，一般用于指针的调整或循环控制。它们不会影响 CCR 中的进位位 C 标志。加 1、减 1 指令如表 3-8 所示。

表 3-8 加 1、减 1 指令

助 记 符	功 能	操 作
加 1 指令		
INC	Increment memory	(M)+ $01→M
INCA	Increment A	(A)+ $01→A
INCB	Increment B	(B)+ $01→B
INS	Increment SP	(SP)+ $0001→SP
INX	Increment X	(X)+ $0001→X
INY	Increment Y	(Y)+ $0001→Y
减 1 指令		
DEC	Decrement memory	(M)− $01→M
DECA	Decrement A	(A)− $01→A
DECB	Decrement B	(B)− $01→B
DES	Decrement SP	(SP)− $0001→SP
DEX	Decrement X	(X)− $0001→X
DEY	Decrement Y	(Y)− $0001→Y

指令使用举例如下：

```
INCA                ;累加器 A 自加 1
DECB                ;累加器 B 自减 1
INC X               ;(X)单元内容自加 1
INX                 ;寄存器 X 内容自加 1
DEC 2, SP           ;(SP + 2)单元内容自减 1
DEC $200A           ;$200A 单元内容自减 1
```

3) 清零、取反和求补指令

任何清零、取反和求补指令都是对累加器或存储器中的值进行特定的运算：清零运算把原来的值清除为 0，取反运算把原来的值替换为它的反码，求补运算把原来的值替换为二进制的补码。取反运算实际上是用 $FF 减去累加器或存储器单元中的值的操作；求补运算实际上是用 $00 减去累加器或存储器单元中的值的操作，利用补码的特点，可以用求补运算求操作数的绝对值或将减法运算化为加法运算。清零、取反和求补指令如表 3-9 所示。

表 3-9　清零、取反和求补指令

助　记　符	功　　能	操　　作
CLC	Clear C bit in CCR	0→C
CLI	Clear I bit in CCR	0→I
CLV	Clear V bit in CCR	0→V
CLR	Clear memory	$00→M
CLRA	Clear A	$00→A
CLRB	Clear B	$00→B
COM	Complement memory	$FF-(M)→M
COMA	Complement A	$FF-(A)→A
COMB	Complement B	$FF-(B)→B
NEG	Complement memory	$00-(M)→M
NEGA	Complement A	$00-(A)→A
NEGB	Complement B	$00-(B)→B

4) 十进制调整指令

十进制调整的 BCD 码指令有 1 条：DAA，该指令的作用是将参与运算的数字 BCD 码进行二进制数计算后调整成正确的结果。DAA 指令的操作数只能是累加器 A，标志位 H、C 必须是 BCD 加法运算的结果。十进制调整指令如表 3-10 所示。

DAA 的调整规则是：

(1) 若 A 的低 4 位大于 9 或者标志 H=1，则 A=A+$06。

(2) 若 A 的高 4 位大于 9 或者标志 C=1，则 A=A+$60+H，同时标志 C 置为 1。

表 3-10　十进制调整指令

助　记　符	功　　能	操　　作
DAA	Decimal adjust A	(A)10

指令使用举例如下：

```
LDAA   # $ 75      ;BCD 码形式的被加数 75 装入 A
ADDA   # $ 58      ;A 加上 BCD 码形式的加数 58,CPU 执行 $ 75 + $ 58 = $ CD,C = 0,H = 0
DAA                ;十进制调整.调整前 A = $ CD,经 + $ 06、+ $ 60 的调整后 A = $ 33,且 C = 1
```

5）符号扩展指令

符号扩展指令 SEX 将 8 位有符号数扩展成 16 位，扩展原则是根据源操作数的最高位是 0 还是 1 决定扩展后的高位字节是 $00 还是 $FF。该指令要求源操作数为寄存器 A、B 或 CCR 的内容，扩展结果存放在寄存器 D、X、Y 或 SP 中。SEX 指令的运行不影响标志位。符号扩展指令如表 3-11 所示。

表 3-11 符号扩展指令

助 记 符	功 能	操 作
SEX	Sign extend 8-bit operand	Sign-extended (A,B,CCR)→D,X,Y,SP

6）乘、除法指令

乘法指令用于有符号和无符号的 8 位和 16 位二进制数乘法。8 位二进制数的乘法有 16 位的结果，16 位二进制数的乘法有 32 位的结果。乘法指令共有 3 条，其中 MUL 是 8 位无符号数的乘法，相关的寄存器是 A、B；EMUL、EMULS 分别是 16 位无符号数和有符号数的乘法，相关的寄存器是 D、Y。指令如表 3-12 所示。

整型和小数除法指令有 16 位的被除数、除数、商和余数。扩展的除法指令有 32 位的被除数和 16 位的除数产生 16 位的商和余数。除法指令共有 5 条，其中 EDIV 和 EDIVS 为常规除法指令，它们以 Y：D 为被除数，X 为除数，分别按照无符号和有符号规则运算，商在 Y 中，余数在 D 中；其余 3 条指令是 16 位除以 16 位的除法运算，它们以 D 为被除数，X 为除数，商在 X 中，余数在 D 中。乘、除法指令如表 3-12 所示。

表 3-12 乘、除法指令

助 记 符	功 能	操 作
EMUL	16 by 16 multiply (unsigned)	(D)×(Y)→Y：D
EMULS	16 by 16 multiply (signed)	(D)×(Y)→Y：D
MUL	8 by 8 multiply (unsigned)	(A)×(B)→A：B
EDIV	32 by 16 divide (unsigned)	(Y：D)÷(X)→Y　余数→D
EDIVS	32 by 16 divide (signed)	(Y：D)÷(X)→Y　余数→D
FDIV	16 by 16 fraction divide (unsigned)	(D)÷(X)→X　余数→D
IDIV	16 by 16 integer divide (unsigned)	(D)÷(X)→X　余数→D
IDIVS	16 by 16 integer divide (signed)	(D)÷(X)→X　余数→D

指令使用举例如下：

```
MUL:   LDD # $4001
       LDY # $0080
       EMUL                    ;乘法结果为 $00200080,其中 $0020 在 Y 中, $0080 在 D 中
DIV:   LDD #2468
       LDX #1234
       IDIV                    ;除法结果为 2,在 X 中
```

3. 逻辑运算类指令

S12X 具有丰富的逻辑运算指令，包括布尔逻辑运算、位测试、位操作、移位等操作。单片机主要是面向 I/O 的操作，在嵌入式系统中，这种面向位和字节的操作是最常用的，例如，进行数字输入/输出引脚的读/写、串行数据通信等。

1）布尔逻辑运算指令

布尔逻辑指令包括“与”(AND)、“或”(OR)、“异或”(EOR)等基本运算，与之相关的寄存器是累加器(A 和 B)、条件码寄存器 CCR 和存储器。布尔逻辑运算指令如表 3-13 所示。

表 3-13　布尔逻辑运算指令

助　记　符	功　　能	操　　作
ANDA	And A with memory	(A)&(M)→A
ANDB	And B with memory	(B)&(M)→B
ANDCC	And CCR with memory (clear CCR)	(CCR)&(M)→CCR
EORA	Exclusive OR A with memory	(A)^(M)→A
EORB	Exclusive OR B with memory	(B)^(M)→B
ORAA	OR A with memory	(A)｜(M)→A
ORAB	OR B with memory	(B)｜(M)→B
ORCC	OR CCR with memory (set CCR)	(CCR)｜(M)→CCR

2）比较、测试指令

比较、测试指令是在两个寄存器之间、寄存器与存储器之间执行减法，运算的结果不会保存，但会根据差值影响 CCR 的状态位。这些指令一般用来为转移指令建立条件，主要用于分支、循环等条件判断。比较指令影响 CCR 的 V、C、N、Z 标志；测试操作相当于一次减零运算，因此 V、C 标志清零，N、Z 受影响。比较、测试指令如表 3-14 所示。

表 3-14　比较、测试指令

助　记　符	功　　能	操　　作
比较指令		
CBA	Compare A to B	(A)－(B)
CMPA	Compare A to memory	(A)－(M)
CMPB	Compare B to memory	(B)－(M)
CPD	Compare D to memory (16bit)	(A:B)－(M:M+1)
CPS	Compare SP to memory (16bit)	(SP)－(M:M+1)
CPX	Compare X to memory (16bit)	(X)－(M:M+1)
CPY	Compare Y to memory (16bit)	(Y)－(M:M+1)
测试指令		
TST	Test memory for zero or minus	(M)－$00
TSTA	Test A for zero or minus	(A)－$00
TSTB	Test B for zero or minus	(B)－$00

3）位测试和操作指令

位测试和操作指令用一个掩码值去测试或改变累加器或存储器中单独的一位或几位。BITA、BITB 是一种方便的测试位指令，是将累加器 A、B 与存储器字节或立即数进行“位与”运算。BCLR、BSET 指令是将操作数的某位清 0 和置 1，BCLR 相当于 0 的位与操作，BSET 相当于 1 的位或操作。此类指令结果影响标志位 N、Z、V，只是 BIT 指令并不修改操作数。位测试和操作指令如表 3-15 所示，其中 mm 为 8 位立即数屏蔽字节。

表 3-15 位测试和操作指令

助 记 符	功 能	操 作
BCLR	Clear bits in memory	(M)&(/mm)→M
BITA	Bit test A	(A)&(M)
BITB	Bit test B	(B)&(M)
BSET	Set bits in memory	(M)|(mm)→M

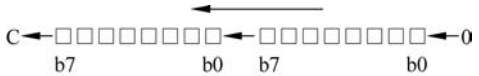
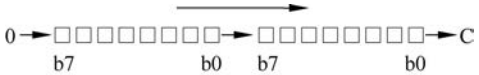
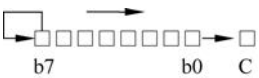
指令使用举例如下：

```
BCLR    $20, #$F0          ;$0020单元内容高4位清0
BSET    $20, #$0F          ;$0020单元内容低4位置1
BITA    #$55               ;测试A的6、4、2、0位是否为0,若是,Z标志置1
```

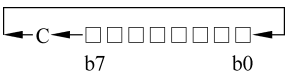
4) 移位指令

移位指令适用于所有的累加器和存储器单元。移位指令分为逻辑移位、算术移位和循环移位 3 种，其中逻辑移位和算术移位在左移时都低位补 0，因此逻辑左移(LSL)和算术左移(ASL)功能相同。逻辑右移时高位补 0，算术右移符号位保持不变，而循环移位又有所不同。移位指令都要借助于进位标志位 C 完成操作，且每次只能移动一位。移位指令如表 3-16 所示。

表 3-16 移位指令

助 记 符	功 能	操 作
逻辑移位		
LSL LSLA LSLB	Logic shift left memory Logic shift left A Logic shift left B	
LSLD	Logic shift left D	
LSR LSRA LSRB	Logic shift right memory Logic shift right A Logic shift right B	
LSRD	Logic shift right D	
算术移位		
ASL ASLA ASLB	Arithmetic shift left memory Arithmetic shift left A Arithmetic shift left B	
ASLD	Arithmetic shift left D	
ASR ASRA ASRB	Arithmetic shift right memory Arithmetic shift right A Arithmetic shift right B	

续表

助 记 符	功 能	操 作
循环移位		
ROL ROLA ROLB	Rotate left memory through carry Rotate left A through carry Rotate left B through carry	
ROR RORA RORB	Rotate right memory through carry Rotate right A through carry Rotate right B through carry	

指令使用举例如下：

```
LSL     1, X+          ;(X)单元左移一位,相当于乘以 2,然后 X 加 1
LSR     $2011          ;$2011 单元里的内容右移一位,相当于除以 2
LSRD                   ;累加器 D 右移一位,高位补 0,低位进 C
ASLD                   ;累加器 D 左移一位,低位补 0,高位进 C
LDX     #$82           ;X 指向一个字节的有符号变量
ASR     8, X           ;(X+8)单元右移一位,最低位进入 C,符号位不变
ROR     0, X           ;循环移位,字节最低位右移一位进入 C,C 进入最高位
```

4. 程序控制类指令

程序控制类指令主要用于控制程序的执行，可以分为转移、循环、调用与返回指令。

转移指令在特定条件存在时引起执行顺序的变化。S12X CPU 有 3 种类型转移指令，分别是短转移指令、长转移指令和按位条件转移指令。转移指令也可按满足转移条件的类型分类，一些指令不只属于一种类别，例如：

(1) 无条件转移指令总是执行。

(2) 先前操作的结果使条件码寄存器产生特定的状态引起简单转移发生。

(3) 当比较和测试无符号值的结果使条件码寄存器特定位变化引起无符号转移。

(4) 当比较和测试有符号值的结果使条件码寄存器特定为变化引起有符号转移。

1) 无条件转移指令

无条件转移指令包括 BRA、BRN、LBRA、LBRN 和 JMP，它们能立即改变指令队列从而使程序无条件转移。BRA、BRN 为短转移指令的特例；LBRA、LBRN 为长转移指令的特例；JMP 跳转指令的转移范围是整个 64 KB 空间，允许直接 16 位寻址和各种形式的变址寻址。指令见表 3-17 和表 3-18 中的无条件转移。

2) 短条件转移指令

短条件转移指令操作过程：当特定的条件满足时，8 位有符号的偏移量加到程序计数器指针上，形成目标地址，程序从新的地址开始执行。短转移指令采用相对寻址，偏移量的范围为－128～＋127。短条件转移指令如表 3-17 所示。

表 3-17 短条件转移指令

助 记 符	功 能	操 作 条 件
无条件转移		
BRA	Branch always	1＝1
BRN	Branch never	1＝0

续表

助 记 符	功 能	操作条件
简单转移		
BCC	Branch if carry clear	C=0
BCS	Branch if carry set	C=1
BEQ	Branch if equal	Z=1
BMI	Branch if minus	N=1
BNE	Branch if not equal	Z=0
BPL	Branch if plus	N=0
BVC	Branch if overflow clear	V=0
BVS	Branch if overflow set	V=1
无符号转移		
BHI	Branch if higher (Result>M)	C | Z=0
BHS	Branch if higher or same (Result≥M)	C=0
BLO	Branch if lower (Result<M)	C=1
BLS	Branch if lower or same (Result≤M)	C | Z=1
有符号转移		
BGE	Branch if greater than or equal (Result≥M)	N^V=0
BGT	Branch if greater than (Result>M)	Z | (N^V)=0
BLE	Branch if less than or equal (Result≤M)	Z | (N^V)=1
BLT	Branch if less than (Result<M)	N^V=1

3) 长条件转移指令

与短条件转移类似,指令助记符中加以“L”(Long)表示。

长条件转移指令操作过程:当特定的条件满足时,16 位的有符号偏移量加到程序计数器指针,形成目标地址,程序从新的地址开始执行。长条件转移指令常用于偏移量大的分支结构之间。长转移指令从下一个存储器地址到转移后的地址的偏移量的范围为-32768~+32767,也即允许转移到 64KB 空间的任意位置。长条件转移指令如表 3-18 所示。

长条件转移指令与短条件转移指令的条件是一样的,只不过短条件转移的偏移值限于 8 位有符号数,长条件转移的偏移值是 16 位有符号数。

表 3-18 长条件转移指令

助 记 符	功 能	操作条件
无条件转移		
LBRA	Long branch always	1=1
LBRN	Long branch never	1=0
简单转移		
LBCC	Long branch if carry clear	C=0
LBCS	Long branch if carry set	C=1
LBEQ	Long branch if equal	Z=1
LBMI	Long branch if minus	N=1
LBNE	Long branch if not equal	Z=0
LBPL	Long branch if plus	N=0
LBVC	Long branch if overflow clear	V=0
LBVS	Long branch if overflow set	V=1

续表

助 记 符	功 能	操 作 条 件
无符号转移		
LBHI	Long branch if higher （Result＞M）	C| Z=0
LBHS	Long branch if higher or same （Result≥M）	C=0
LBLO	Long branch if lower （Result＜M）	C=1
LBLS	Long branch if lower or same （Result≤M）	C| Z=1
有符号转移		
LBGE	Long branch if greater than or equal （Result≥M）	N^V=0
LBGT	Long branch if greater than （Result＞M）	Z |（N^V）=0
LBLE	Long branch if less than or equal （Result≤M）	Z |（N^V）=1
LBLT	Long branch if less than （Result＜M）	N^V=1

4）位条件转移指令

当存储器单元中的某一位处于特定状态时，可以发生按位条件转移的操作。一个掩码操作数用来测试这个地址的值，若所有的位符合该地址的掩码值置位（BRSET）或复位（BRCLR），转移就会发生。从一个存储器地址到转移后的地址的偏移量的范围为－128～＋127。位条件转移指令如表 3-19 所示。

该类指令有 2 条：BRCLR 和 BRSET，指令不依赖标志，也不影响标志。其中 BRCLR 检测存储器字节的某些选定位是否为 0，若是则转移；BRSET 检测存储器字节的某些选定位是否为 1，若是则转移。这里的某些选定位是由指令中的一个立即数（掩码）决定的，表中的 mm 即为 8 位立即数屏蔽字节。

表 3-19　位条件转移指令

助 记 符	功 能	操 作 条 件
BRCLR	Branch if selected bits clear	（M）&（mm）=0
BRSET	Branch if selected bits set	（M）&（mm）!=0

指令使用举例如下：

```
BRCLR   $20, #$81, LP1          ;若$0020单元内容的最高位和最低位为0,转移到LP1
BRSET   $20, #$80, LP2          ;若$0020单元内容的最高位为1,转移到LP2
```

5）循环控制指令

循环控制指令可以看作是计数器转移。这种指令测试寄存器或累加器（A、B、D、X、Y 或 SP）中的计数值是否为 0 作为转移条件。在这些指令中有递减、递增和直接测试 3 种类型。这类指令均不影响标志位，也不依赖标志位。从一个存储器地址到转移后的地址的偏移量的范围为－256～＋255。循环控制指令如表 3-20 所示。

该类指令共有 6 条，其中前 4 条自动调整循环计数器，可以实现类似高级语言中的 for 循环；另外 2 条实际上是条件分支，可以实现 while 循环。

表 3-20　循环控制指令

助 记 符	功 能	操 作
DBEQ	Decrement counter and branch if=0 （counter=A,B,D,X,Y,or SP）	（counter）－1→counter If （counter）=0,then branch; else continue to next instruction

续表

助 记 符	功 能	操 作
DBNE	Decrement counter and branch if ≠ 0 (counter=A,B,D,X,Y,or SP)	(counter)−1→counter If (counter) not=0,then branch; else continue to next instruction
IBEQ	Increment counter and branch if=0 (counter=A,B,D,X,Y,or SP)	(counter)+1→counter If (counter)=0,then branch; else continue to next instruction
IBNE	Increment counter and branch if ≠ 0 (counter=A,B,D,X,Y,or SP)	(counter)+1→counter If (counter) not=0,then branch; else continue to next instruction
TBEQ	Test counter and branch if=0 (counter=A,B,D,X,Y,or SP)	If (counter)=0,then branch; else continue to next instruction
TBNE	Test counter and branch if ≠ 0 (counter=A,B,D,X,Y,or SP)	If (counter) not=0,then branch; else continue to next instruction

指令使用举例如下：

```
    LDAB #4
LOOP: …
    DBNE B, LOOP
```

6) 跳转、子程序调用与返回指令

跳转、子程序调用与返回指令如表 3-21 所示。

跳转指令(JMP)在执行顺序中立即变化。JMP 指令将 64 位的存储器映射地址装入 PC 指针中，程序继续从这个地址执行。这个地址可以是 16 位绝对地址或由多种变址寻址方式决定的地址。

子程序是一段能够完成特定任务的程序代码，是一个优化执行特定任务的转移控制过程。转移指令 BSR、子程序跳转指令 JSR 或扩展调用指令 CALL 均能引起子程序的调用。

BSR、JSR 用于 64KB 以内的调用，调用子程序时，首先在堆栈中保存下一条指令的地址(返回地址)，然后执行调用的子程序，子程序在遇到返回指令 RTS 将结束执行调用，并从堆栈中取得返回地址，回到原来的位置继续运行。BSR 采用相对寻址方式，调用范围为 −128～+127；而 JSR 可支持 7 种寻址方式，其中直接寻址的子程序入口地址必须在 0～255 内，其他寻址方式调用范围为−32768～+32767，即可寻址 64KB 空间。

扩展调用指令 CALL 用于 64KB 以外的扩展存储器地址空间调用。CALL 存储 PPAGE 寄存器中的值到堆栈中并返回地址，然后将子程序所在存储器的页号写入 PPAGE 寄存器中。除直接变址寻址外，页号是一个立即操作数。在这些方式下，在新的页中指向存储器地址指针的值和子程序地址被存储起来。返回调用指令 RTC 用于在扩展存储器地址中结束程序。RTC 将 PPAGE 寄存器的值和返回地址出栈以使恢复执行 CALL 指令的下一条语句。为了软件兼容性，CALL 和 RTC 指令在没有扩展地址能力的器件中能正确执行。

表 3-21　跳转、子程序调用与返回指令

助　记　符	功　　能	操　　作
BSR	Branch to subroutine	SP－2→SP RTNH：RTNL→M(SP)：M(SP＋1) Subroutine address→PC
CALL	Call subroutine in Expanded Memory	SP－2→SP RTNH：RTNL→M(SP)：M(SP＋1) SP－1→SP (PPAGE)→M(SP) Page→PPAGE Subroutine address→PC
JMP	Jump	Address→PC
JSR	Jump to subroutine	SP－2→SP RTNH：RTNL→M(SP)：M(SP＋1) Subroutine address→PC
RTC	Return from call	M(SP)→PPAGE SP＋1→SP M(SP)：M(SP＋1)→PCH：PCL SP＋2→SP
RTS	Return from subroutine	M(SP)：M(SP＋1)→PCH：PCL SP＋2→SP

5. 中断类指令

中断类指令在程序控制中起着一个重要的作用。有 RTI、SWI 和 TRAP，它们控制 CPU 停止当前操作转向去执行一项更重要的任务（SWI 除外）。中断类指令如表 3-22 所示。

中断返回 RTI 指令用来终止所有的异常处理程序，包括一般中断服务程序。RTI 首先从堆栈中返回 CCRH（只在 S12X 中）、CCR、B、A、X、Y 的值和返回地址。若没有其他中断发生，中断恢复到在中断产生前的最后一条指令处。

软件中断 SWI 是一种以软件方式启动的特殊中断类型，通过中断响应方式进入服务例程。首先，返回 PC 指针的值到堆栈中。然后，CPU 中所有寄存器内容保存到堆栈中，程序从 SWI 向量指向的地址执行。执行 SWI 指令会引起中断而不需要中断服务请求。SWI 不受 CCR 寄存器中全局屏蔽位 I 和 X 的控制，执行 SWI 操作会对 I 位置位。一旦 SWI 中断执行，可屏蔽中断被禁止直至 CCR 中的 I 位被清除。在 SWI 结束时中断返回指令 RTI 恢复所有保存的寄存器值。SWI 中断向量地址为 $FFF6～$FFF7。

TRAP 是非法陷阱指令，当 CPU 遇到非法指令时，自动进入 TRAP 中断响应过程。这是 CPU 为无效操作码提供改变的软件中断，若 CPU 试图执行一个无效操作，此时会发生操作码陷阱中断。TRAP 中断向量地址为 $FFF8～$FFF9。

表 3-22 中断类指令

助记符	功能	操作
RTI	Return from interrupt	(M(SP) : M(SP+1))→CCRH : CCR; (SP)－$0000→SP (M(SP) : M(SP+1))→B : A; (SP)－$0002→SP (M(SP) : M(SP+1))→XH : XL; (SP)－$0004→SP (M(SP) : M(SP+1))→PCH : PCL; (SP)－$0006→SP (M(SP) : M(SP+1))→YH : YL; (SP)－$0008→SP
SWI	Software interrupt	SP－2→SP; RTNH : RTNL→M(SP) : M(SP+1) SP－2→SP; YH : YL→M(SP) : M(SP+1) SP－2→SP; XH : XL→M(SP) : M(SP+1) SP－2→SP; B : A→M(SP) : M(SP+1) SP－2→SP; CCRH : CCR→M(SP) : M(SP+1)
TRAP	Unimplemented opcode interrupt	SP－2→SP; RTNH : RTNL→M(SP) : M(SP+1) SP－2→SP; YH : YL→M(SP) : M(SP+1) SP－2→SP; XH : XL→M(SP) : M(SP+1) SP－2→SP; B : A→M(SP) : M(SP+1) SP－2→SP; CCRH : CCR→M(SP) : M(SP+1)

6. CPU 控制类指令

CPU 控制类指令有 STOP、WAI、BGND、BRN、LBRN 和 NOP，如表 3-23 所示。

停止指令 STOP 与等待指令 WAI 为使 CPU 处于闲置状态以降低功耗。STOP 将返回地址和 CPU 的寄存器和累加器中的值入栈，然后停止系统时钟；WAI 将返回地址和 CPU 的寄存器和累加器中的值入栈，然后等待一个中断服务请求，但是系统时钟信号仍然存在。STOP 和 WAI 指令在重新恢复正常指令执行之前，要求有一个中断或者复位的操作。尽管这两条指令在一个中断服务请求产生后恢复到正常程序执行需要相同的时钟周期，但系统从 STOP 模式恢复到正常状态所需时间多于系统从 WAI 模式恢复所需的时钟周期，因为在 STOP 模式下需要额外的时间恢复时钟系统。

STOP 使系统进入停止工作状态，时钟振荡器在内部被关掉，使整个系统停止运行，此时功耗最低，但当 CCR 中的 S=1 时，STOP 操作被禁止，这时 STOP 指令相当于空操作指令。WAI 使系统进入低功耗待机状态，但系统时钟仍然继续运行，可以加速中断响应。

背景调试模式(BDM)是用于系统开发和调试的一种特殊的 CPU 操作模式。在这种模式下当 CPU 的 BDM 使能时，BGND 指令可以使系统进入背景调试模式。

不跳转 BRN、LBRN 指令替代有条件转移指令，例如，使用不跳转指令来调试程序，而不需要影响偏移量。

在软件调试中，空操作 NOP 指令常用来取代其他指令或者凑时钟周期；空操作也可以用作软件延时程序消耗程序执行时间而不影响其他 CPU 寄存器或存储器中的内容。

背景调试模式是一种非常有用的系统开发和调试模式。当 BDM 使能时，BGND 指令可以使系统进入背景调试模式。

表 3-23　CPU 控制类指令

助　记　符	功　　能	操　　作
STOP	Stop	SP－2→SP；RTNH：RTNL→M(SP)：M(SP＋1) SP－2→SP；YH：YL→M(SP)：M(SP＋1) SP－2→SP；XH：XL→M(SP)：M(SP＋1) SP－2→SP；B：A→M(SP)：M(SP＋1) SP－2→SP；CCRH：CCR→M(SP) M(SP＋1) Stop CPU clocks
WAI	Wait for interrupt	SP－2→SP；RTNH：RTNL→M(SP)：M(SP＋1) SP－2→SP；YH：YL→M(SP)：M(SP＋1) SP－2→SP；XH：XL→M(SP)：M(SP＋1) SP－2→SP；B：A→M(SP)：M(SP＋1) SP－2→SP；CCRH：CCR→M(SP)：M(SP＋1)
BGND	Enter background debug mode	If BDM enabled，enter BDM
BRN	Branch never	Does not branch
LBRN	Long branch never	Does not branch
NOP	Null operation	

7. 全局读写类指令

S12X CPU 指令系统新增提供了全局 23 位地址访问的指令包括 GLDAA、GLDAB、GLDD、GLDS、GLDX、GLDY、GSTAA、GSTAB、GSTD、GSTS、GSTX、GSTY 等，S12X 总共新增了 84 条这类全局寻址指令。它们以页面寄存器 GPAGE 的内容为高 7 位地址，以相应寻址方式给出的地址为低 16 位地址，形成全局 8MB 空间访问。全局地址[22:0]即由 GPAGE 寄存器[22:16]和 CPU 本地地址[15:0]联合组成。全局读写类指令的助记符在一般读写类指令前加有“G”标识，寻址方式与一般读写类指令也相同。指令使用举例如下：

```
MOVB    # $ 0F, $ 10     ;将立即数字节 $ 0F 送到 $ 10 单元，即 GPAGE 寄存器的值设为 $ 0F
GLDAB   $ 20FA          ;把全局地址 $ 0F_20FA 单元的内容装载到累加器 B 中
```

8. 其他指令

1) 条件码操作指令

条件码操作指令是针对条件码寄存器 CCR 的特殊访问指令，通常被用来改变 CCR。例如，CLI 用来开中断，SEI 用来屏蔽中断。条件码操作指令如表 3-24 所示。

表 3-24　条件码操作指令

助　记　符	功　　能	操　　作
ANDCC	Logical AND CCR with memory	(CCR) & (M)→CCR
CLC	Clear C bit	0→C
CLI	Clear I bit	0→I
CLV	Clear V bit	0→V
ORCC	Logical OR CCR with memory	(CCR)｜(M)→CCR
PSHC	Push CCR onto stack	(SP)－1→SP；CCR→M(SP)

续表

助 记 符	功 能	操 作
PSHCW	Push CCRH：CCR onto stack	(SP)－2→SP； (CCRH：CCR)→M(SP)：M(SP＋1)
PULC	Pull CCR from stack	(M(SP))→CCR；(SP)＋1→SP
PULCW	Pull CCRH：CCR from stack	(M(SP)：M(SP＋1))→CCRH：CCR； (SP)＋2→SP
SEC	Set C bit	1→C
SEI	Set I bit	1→I
SEV	Set V bit	1→V
TAP	Transfer A to CCR	(A)→CCR
TPA	Transfer CCR to A	(CCR)→A

2) 高级函数指令

S12X 指令系统还提供高级函数指令。高级函数指令可以简化用户程序，减小编程工作量，提高可靠性和可读性，这类指令主要有取大值/小值指令、乘加指令和查表插值指令等。这些指令都是 S12X 的特色指令，实际使用时可自行查阅相关资料。高级函数指令如表 3-25～表 3-27 所示。

表 3-25 取大值/小值指令

助 记 符	功 能	操 作 条 件
取小值指令		
EMIND	MIN of two unsigned 16-bit values to D	MIN ((D),(M：M＋1))→D
EMINM	MIN of two unsigned 16-bit values to memory	MIN ((D),(M：M＋1))→M：M＋1
MINA	MIN of two unsigned 8-bit values to A	MIN ((A),(M))→A
MINM	MIN of two unsigned 8-bit values to memory	MIN ((A),(M))→M
取大值指令		
EMAXD	MAX of two unsigned 16-bit values to D	MAX ((D),(M：M＋1))→D
EMAXM	MAX of two unsigned 16-bit values to memory	MAX ((D),(M：M＋1))→M：M＋1
MAXA	MAX of two unsigned 8-bit values to A	MAX ((A),(M))→A
MAXM	MAX of two unsigned 8-bit values to memory	MAX ((A),(M))→M

表 3-26 乘加指令

助 记 符	功 能	操 作
EMACS	Multiply and accumulate (signed)	((M(X)：M(X＋1))×(M(Y)：M(Y＋1)))＋ (M～M＋3)→M～M＋3

表 3-27 查表插值指令

助 记 符	功 能	操 作
ETBL	16-bit table lookup and interpolate	(M：M＋1)＋[(B)×((M＋2：M＋3)－(M：M＋1))]→D
TBL	8-bit table lookup and interpolate	(M)＋[(B)×((M＋1)－(M))]→A

取大值/小值指令(MAX 和 MIN)用来比较累加器和一个存储器单元。MAX 和 MIN 指令用累加器 A 执行 8 位数的比较，结果(大值或小值)存入累加器 A 或存储器中；而 EMAX 和 EMIN 指令用累加器 D 执行 16 位数的比较，结果(大值或小值)存入累加器 D 或

存储器中。

乘加指令 EMACS 将两个 16 位的操作数相乘，相乘的结果再加上在另一个存储器地址中的 32 位数，最后的乘加结果存入 32 位的存储器地址中。EMACS 可以 16 位操作数来实现简易数字滤波和常规模糊化操作。

查表插值指令（TBL 和 ETBL）将表中的值插入存储器中。任一功能均可被描述为一系列适当表格大小的线性方程。插值可以用作多种用途，包括表格模糊逻辑从属函数。TBL 用 8 位长度的表输入返回一个 8 位的结果，ETBL 用 16 位长度的表输入返回一个 16 位的结果。考虑到存入到表中的每个连续值可以看作一个线段端点的 y 值。在指令执行之前累加器 B 中的值表示 X 的值从线段开始到查找点（从开始到线段末被分割的点）的变化值。B 看作在 MSB 左边有小数点的 8 位二进制小数，所以每个线段被有效地分割成 256 个小线段。在指令执行中，y 值在线段始端和末端变化（TBL 为一个字节，ETBL 为一个字）乘以累加器 B 中的值得到一个中间值 Δy。这个结果（TBL 指令存在累加器 A 中，ETBL 存在累加器 D 中）是从线段始端的 y 值加上有符号的 Δy 值。

3.4 使用汇编语言的程序设计

程序是完成特定任务的指令的集合，多个程序构成系统运行的软件。MCU 的程序设计语言主要有：

（1）机器语言。使用二进制指令代码编写程序，是 MCU 可以直接执行的机器码。

（2）汇编语言。使用特定助记符指令语句编写程序，须经编译后形成机器码。

（3）高级语言。使用通用高级语句如 C 语言编写程序，须经编译后形成机器码。

单片机应用系统的程序多用汇编语言（*.asm 文件）或 C 语言（*.c 文件）编制。程序需要特定的编译程序进行编译，最终仍然生成二进制机器码。对于汇编语言编程，一个汇编语言程序语句对应一条单片机指令，多个汇编语言语句就构成汇编语言程序（源代码）。

汇编语言是一种面向物理界面和硬件接口的系统执行语言，是与 MCU 硬件和指令集密切相关的，因此不易移植；但它以高效、直接面向硬件和代码量小等优点，一直在 MCU 的学习和小型嵌入式应用系统方面占据很重要的地位。

3.4.1 汇编语言的指令格式与伪指令

1. 汇编指令格式

汇编语言程序以行为单位，每行一条指令，一行程序以回车符结束。每行由标号、操作码、操作数和注释 4 部分组成：

[标号:] 操作码 [操作数 1] [,操作数 2] [;注释]

（1）标号。指令所在地址的符号表示，在最终的机器代码中，标号会被编译成本行程序所在存储区的实际物理地址，标号通常是程序转移位置、子程序入口，在需要时标定，标号后面要有“:”符号。

（2）操作码。指令的汇编助记符，例如 LDAA、INC 等。编译后，产生指令的操作码部分，是唯一不可缺少的部分。

（3）操作数。指令的操作对象，根据指令的操作要求不同，可能有一个或两个操作数，

操作数与操作码之间用“,”分开。操作数可以是寄存器、数据、地址等,也可以是常数或表达式。

(4) 注释。对程序的解释说明,在最终的机器代码中,注释不产生任何有意义的信息。注释前加上分号“;”与程序分开。

注意:指令必须在半角字符模式下书写,全角字符不能被汇编语言程序识别。还要注意程序书写的规范和风格,如各部分最好按列对齐、统一用大写字母等,以便阅读和维护。

下面为一个汇编语言程序实例:

```
L1:     LDAA    # $FF          ;赋初值
        STAA    DDRB           ;设置 B 口方向为输出
        LDAA    # $FE          ;赋初值
SHIFT:  STAA    PORTB          ;B 口循环输出;前面必须有标号,表示循环入口地址
        BSR     DELAY          ;调用延时子程序
        ROLA                   ;隐含寻址,不需要操作数
        BRA     SHIFT          ;循环

; …                            ;该行仅有注释,可以是下面子程序的说明
DELAY: …                       ;DELAY 子程序
```

2. 操作数的表示

汇编语言编程时,操作数除了是 CPU 寄存器外,一般以数据或地址的形式出现,也可以是表达式、常数、标号或用伪指令预定义的赋值常量。

常数可采用十进制、十六进制、二进制、八进制 4 种格式。NXP MCU 用下列前缀来表示所使用的常数的格式:

```
无       十进制(Decimal)
$        十六进制(Hexadecimal)
%        二进制(Binary)
@        八进制(Octal)
```

ASCII 码形式的字符串常数(String)用单引号或双引号包含进来表示。若字符串的内容包括单引号,则只能用双引号表示,若字符串的内容包括双引号,则只能用单引号表示。例如,"ABCD"、'ABCD'、"A'B"、'A"B'。

常数通常用来创建参数值、查找表和初始化变量的初始值。编译程序会把汇编语言程序中所有常数变换成二进制码,并在 IDE 的汇编列表时以十六进制格式显示。

3. 汇编伪指令

MCU 汇编语言程序除了指令系统语句以外,还定义了许多汇编管理的语句,即伪指令。伪指令是设计汇编语言程序的一个重要组成部分,是汇编语言程序完成特定操作的一种指示,它们仅为汇编编译程序提供某些汇编命令或操作数。伪指令是汇编语言程序使用的辅助性语句,并不生成机器码。

下面是 NXP MCU 的一些常用伪指令。

1) 起始地址伪指令 ORG

ORG(Original)伪指令给汇编语言程序定义起始地址,指出紧跟在该伪指令后的机器码指令的汇编地址,也就是经编译生成的机器码目标程序或数据块在存储器中的起始存放地址。其语法为:

```
ORG              地址值
```

例如：

```
ORG $C000              ;通知编译器从$C000起始地址处定位下面的程序代码
```

在一个汇编语言源程序中，可以在有效存储器地址的范围内指定子程序或转移程序的存放位置。在程序设计时，某段程序既可以续接前面的程序依次存放，也可以选择特殊的地址存放。因而允许在一段源程序中使用多条 ORG 伪指令，但后一个 ORG 伪指令的操作数应大于前面机器代码已占用的存储地址。当采用多条 ORG 语句时，程序所设定占用的地址不要重叠，否则在汇编编译时会提示出错。

2）赋值伪指令 EQU

EQU(Equal)伪指令的操作含义是将一个符号设置为等同于某值，即使得 EQU 两端的值相等，汇编程序在编译中遇到该符号即代之以赋值语句右端的值。其语法为：

```
符号  EQU  数值
```

符号名一旦被 EQU 赋值，其值就不能再重复定义。这里的符号所代表的数值，既可以是特殊功能寄存器地址、存储器地址、常数，也可以表示一个通用数据或者一个表达式。一般在程序设计中，预定义一些赋值方法对于代码可读性和可修改性是很重要的。用符号代替数值，程序员可对操作的意义一目了然。例如：

```
COUNT  EQU  100
```

这是预定义了一个计数器符号 COUNT。在后面的程序中直接使用符号即可，欲改变计数器值，就直接修改赋值常数，而后面的程序代码不变。

又如，有如下 2 条赋值定义行：

```
PORTB         EQU     $0001          ;PORTB寄存器的实际地址
mPORTB_PB0    EQU     %00000001      ;PORT寄存器第0位的掩码
```

汇编编译程序将用 $0001 代替所有 PORTB，用%00000001 代替所有 mPORTB_PB0。经过这样的预先定义的赋值操作，在后面的程序中就可以使用符号表达：

```
BSET  PORTB, mPORTB_PB0
```

借助于这些符号，代码的意图就很清晰，更好理解。它是将 PORTB 的第 0 位设置为 1。等效于如下形式语句：

```
BSET  $01, %00000001
```

3）字节常量定义伪指令 FCB

FCB(Form Constant Byte)伪指令用于定义一个或多个单字节常量表，依次存放在存储器单元中。其语法为

```
标号:  FCB  字节常数1[,字节常数2][,字节常数2] …
```

若定义多个单字节常数，则各常数之间用逗号分开。例如：

```
BASE    EQU    8
        ORG    $0800
TABLE:  FCB    $5A, 200, BASE+16, $AB
```

上述伪指令 FCB 具有 4 个常量，它们依次存放在存储器的 4 个字节中。其中第 2 行是

ORG 语句，实际上规定了标号 TABLE 所代表的地址为 $0800，所以第 1 字节的存放地址为 $0800，该地址中存放的内容为 $5A，则地址 $0801 中的内容为 200（$C8），地址 $0802 中的内容为 24（$18），地址 $0803 中的内容为 $AB。

4）双字节常量定义伪指令 FDB

FDB（Form Double Byte）伪指令用于定义一个或多个双字节常量（字）表，依次存放在存储器单元中。其语法为

```
标号:  FDB  字常数 1 [,字常数 2] [,字常数 2] …
```

若定义多个双字节常数，则各常数之间用逗号分开。例如：

```
LIST:  FDB  $55AA, 2000
```

上述伪指令 FDB 创建 2 个双字节常量，它们依次存放在存储器的 4 个字节中，起始地址为标号 LIST 所代表的地址。

5）字符常量定义伪指令 FCC

FCC（Form Constant Character）伪指令用于定义 ASCII 字符串常量，一个单字节常量对应一个字符，依次存放在存储器单元中。其语法为

```
标号:  FCC "字符串''
```

例如：

```
LIST: FCC "ABCD"
```

其中，"ABCD"是定义的字符串，单引号（'）或双引号（"）包含均可。本例创建 4 个单字节字符常量，即 A、B、C、D 这 4 个字母的 ASCII 十六进制数值。若没有定义起始地址，则常量按程序代码当前地址位置接续排放，低地址处的内容为 ASCII 字符 A 的十六进制数值 $41，以此类推，各单元内容为 $41、$42、$43、$44。

6）空间保留伪指令 RMB、RMD

RMB（Reserve Memory Byte）和 RMD（Reserve Memory Double）伪指令为变量保留 RAM 存储器空间。例如：

```
VAR1:  RMB  2
VAR2:  RMD  2
```

前者实际上是定义了变量 VAR1，它的数据宽度为 2 字节；后者定义变量 VAR2，它的数据宽度为 4 字节。

7）编译结束伪指令 END

END 伪指令通知编译器后面的程序内容忽略。因此写在 END 之后的指令将不会被编译，也不生成相应的机器码。

8）外部变量定义指令 XDEF

XDEF（eXternal Define）伪指令定义外部变量或符号，表示此处定义的变量或符号可以被其他模块或文件引用。

9）外部参考指令 XREF

XREF（eXternal Reference）伪指令声明外部变量或符号，表示此处声明的变量或符号是在其他模块或文件中定义的。

10）段信息指令 SECTION

SECTION 伪指令用来声明可重定位的段信息，指定某段代码放于什么位置。例如：

MY_EXTENDED_RAM: SECTION

相同名字的段代表后面出现的同名段中的代码将被排放在前一个同名段的最后一条语句之后，也就是通知编译器将具有相同段名字的代码内容接续放在一起。经过 SECTION 指令定义以后，还可以在 CodeWarrior IDE 的 *.prm 文件中重新定位 MY_EXTENDED_RAM 段在存储器中的位置。不过，通常情况下就使用默认的定位方式，如变量数据段在 RAM 区，代码段在 ROM 区，常量也在 ROM 区。

3.4.2 汇编语言编程举例

本节通过几个实例来讨论编写 S12X MCU 的汇编语言程序的基本方法与一般技巧，而涉及 MCU 各功能模块的汇编语言编程应用，本书不做讲述（代之以 C 语言编程）。

1. 基本数据传递与算术运算程序

数据传递与算术运算是实现各种程序功能的基本操作，是汇编语言程序中最基础的代码实现。

【例 3-1】 利用 BCD 运算求十进制数 3275 和 2658 的和，结果存放在寄存器 D 中。

汇编语言程序代码如下：

```
;*******************************************************************
LDD     # $3275          ;$3275→D ((A) = $32,(B) = $75)
ADDB    # $58            ;B + $58,结果(B) = $CD,C = 0,H = 0
EXG     A, B            ;将 B 内容交换到 A,以便进行调整
DAA                     ;结果(A) = $33(由 $CD 先加 $06 再加 $60 调整而得),C = 1
EXG     A, B            ;将 A 内容交换回 B($33),同时 A 取回原来的 $32
ADCA    # $26           ;A + $26 + C,结果(A) = $59,C = 0,H = 0
DAA                     ;结果(A) = $59(未做调整),C = 0
```

上述程序运行后将得到正确的最终结果(D)＝$5933（应看成十进制 BCD 码 5933）。

【例 3-2】 比较 RAM 区内两相邻单元中无符号数的大小，按小数在前、大数在后重新存放（首址在 $2000）。若相等则 Y 寄存器加 1。

汇编语言程序代码如下：

```
;*******************************************************************
        LDX     # $2000        ;数据首地址
        CLC                   ;C 清零
        LDY     #0            ;Y 清零
BEGIN:  LDAA    0,X           ;(0 + X) →A
        LDAB    1,X           ;(1 + X) →B
        CBA                   ;(A) - (B)
        BCS     DONE          ;减有借位(前小后大),无须调整,跳转
        BEQ     FLAG          ;相等,跳转
        STAA    1,X           ;A→(1 + X)
        STAB    0,X           ;B→(0 + X)
        BRA     DONE          ;调整完毕,跳转
FLAG:   INY                   ;Y + 1→Y
DONE:   BRA     *             ;结束,踏步等待
```

2. 循环控制程序

循环是一种基础的程序结构，通过判断指令确定是否满足循环条件。判断功能主要适用于增量和减量的操作，数据存储器中每一个单元都可以作为判断指令的操作对象。当经过增减量操作后，依据单元结果进行跳转，循环计数器设置的初始数值可以控制循环次数。

【例 3-3】 2 个 8 字节数求和，结果保存在被加数所在地址中。

汇编语言程序代码如下：

```
; ***********************************
      …
ADD8: LDX      # $2011
      LEAY     8,X
      LDAB     #8
      CLC
LOOP: LDAA     X
      ADCA     1,Y+
      STAA     1,X+
      DBNE     B, LOOP
      …
```

以上程序中的循环结构中，使用减 1 不为 0 转移指令 DBNE 控制循环次数，在实际应用中是典型、常见的用法。

【例 3-4】 批量数据块传送：将存储器源地址处的若干字数据传送到目的地址处。

汇编语言程序代码如下：

```
; ********************************************************************
BlockMove:   LDX      # SOURCE
             LDY      # TARGE
             LDD      # COUNT   ; 用 D 做循环计数可以超过 256 次
LOOP:        MOVW     2, X+, 2, Y+
             DBNE     D, LOOP
```

其中的符号应在程序代码前进行赋值(EQU)指定。

3. 延时子程序

在程序设计中，延时程序占有很重要的地位。延时功能的实现可以采用两种方式：硬件延时和软件延时。硬件延时由 MCU 的内部定时器实现，软件延时通过循环程序实现。前者适用于精确定时，不占用 CPU；后者常用于粗略定性延时，通过 MCU 多次循环地执行一段程序代码，利用消耗程序指令的周期数完成延时。软件延时显然是占用 CPU 时间的，适合短时间延时或 CPU 空闲时使用，并一般被安排成子程序。

【例 3-5】 5ms 软件延时。

汇编语言程序代码如下：

```
;主程序
        …
        JSR     DELAY0     ;4 T_BUS
        …
; ********************************************************************
; 延时子程序 DELAY0
; ********************************************************************
```

```
TCNT     EQU    9996          ;无
DELAY0:  PSHX                 ;2T_BUS
         LDX    #TCNT         ;3T_BUS
LOOP:    DEX                  ;1T_BUS
         BNE    LOOP          ;3/1T_BUS
         PULX                 ;3T_BUS
         RTS                  ;5T_BUS
```

CPU 执行时间：

$$T = N \times T_{BUS}$$

其中，T 为总执行时间，此例要求 5ms；N 为总的时钟周期数；T_{BUS} 为总线时钟周期（假设为 1/8M＝125ns）；则需：

$$N = T / T_{BUS} = 5\text{ms}/125\text{ns} = 40000$$

例中程序的注释标明了各条指令运行的时钟周期，其中 BNE 指令不发生分支转移时只占 1 个周期，而发生转移即循环时占用 3 个周期。总的运行时间与循环次数有关，程序运行的总的周期数：

$$N = 4 + 2 + 3 + (1 + 3) \times (\text{TCNT} - 1) + (1 + 1) + 3 + 5$$

故得 TCNT＝9996.25≈9996。

该值在实际使用时也可直接粗略设为 10000。不同时间的延时可通过修改 TCNT 的值予以改变。

【例 3-6】 双重循环的 100ms 软件延时子程序。

汇编语言程序代码如下：

```
;*****************************************************************
;子程序 DELAY:利用寄存器 X、Y,执行双重循环实现延时
;*****************************************************************
DELAY: PSHX
       PSHY
       LDX     #100
DL1:   LDY     #400
DL2:   NOP              ;1T_BUS
       NOP              ;1T_BUS
       DBNE    Y,DL2    ;3T_BUS
       DBNE    X,DL1
       PULY
       PULX
       RTS
```

该子程序中空的指令 NOP 的作用为凑时钟周期，总的延时可以忽略配合指令的执行时间误差，进行粗略估算：

$$\text{内循环} = 400 \times (1 + 1 + 3) T_{BUS} = 2000 T_{BUS}$$

$$\text{总时间} = 100 \times 2000 T_{BUS} = 200000 T_{BUS}$$

因此，当 MCU 外接 4MHz 晶振时，则总线频率为 2MHz，则 T_{BUS}＝500ns。上面子程序延时即为 100ms。其他延时可通过更改循环次数而套用上述方法实现。

注意：MCU 总线频率的不同将带来延时的变化。

4. 数据查表程序

数据查表方法在 LED 数码管显示、ASCII 码转换、固定数值查表等程序设计应用中下

较为方便实用。

【例 3-7】 ASCII 码查表转换。将 A 中的两个 4 位十六进制数转换为 ASCII 码，分别存入 $2080、$2081 中。

计算机技术中定义：0～9 的 ASCII 码为 $30～$39，A～F 的 ASCII 码为 $41～$46。本例将这些数值预先存在 Flash 中通过查表获得所需的 ASCII 码，而不是直接硬算。

该实现的汇编语言程序代码的主程序段和子程序段分别如下：

```
;*****************************************************************
; 主程序段
;*****************************************************************
HEXA:     TFR       A,X       ;传 A 给 X 暂存
          ANDA      # $ 0F    ;高 4 位置 0,屏蔽
          JSR       TRANS     ;查表转换
          STAB      $ 2080    ;存储

          TFR       X,A       ;取回原值到 A
          LSRA                ;逻辑右移位
          LSRA
          LSRA
          LSRA                ;高 4 位移至低 4 位,高位被补 0
          JSR       TRANS     ;查表转换
          STAB      $ 2081    ;存储
          …
;*****************************************************************
;子程序 TRANS:十六进制数转换为 ASCII 码
;入口参数:(A) = 十六进制数,高 4 位为 0
;出口参数:(B) = ASCII 码
;*****************************************************************
TRANS:    PSHX                ;X 入栈
          LDX       # TABLE   ;置表地址
          LDAB      A,X       ;查表,(X + A)→B
          PULX                ;X 出栈
          RTS                 ;返回

TABLE: FCC "0123456789ABCDEF" ;伪指令预定义 16 个十六进制数的 ASCII 码表
```

5. 多分支转移程序

分支转移指令实际上是多条件判断指令，条件本身是一个数据或事件，而跳转出口应该是数据信息的返回或事件功能内容的具体表现。例如在 MCU 系统中，键盘扫描程序是最基本的人机输入程序，应用分支转移方式，能方便地实现每一个功能键的程序方向。下列程序中，分支转移子程序与数据查表结构类似，键号输入的转移方向安排在 Jmp_Table 表中。

【例 3-8】 根据 4×4 键盘的键号执行相应子程序。

汇编语言程序代码如下：

```
;*****************************************************************
          …
KeyMain:  JSR     Key16       ;调用键号获取子程序(键号:0～15,16 无键按下)
          LDAA    Key_Numb    ;装载键号到 A,Key_Numb 地址预先有定义
          LSLA                ; (A) = (A) × 2,形成 Jmp_Table 表的偏移
          LDX     # Jmp_Table ;给 X 赋表首地址
```

```
            LDY     A, X          ;查表:给 Y 赋为(X + A),即取得分支地址
            JSR     Y             ;跳转到 Y 指向的分支地址
            …
Jmp_Table:  FDB     Key0Sub       ;各个分支地址列表,占用双字节
            FDB     Key1Sub
            …
            FDB     Key15Sub
            FDB     KeyNo

Key0Sub:    …                     ;键号 0 功能
            RTS
Key1Sub:    …                     ;键号 1 功能
            RTS
…
Key15Sub:   …                     ;键号 15 功能
            RTS
KeyNo:      RTS                   ;无键按下,直接返回
```

注意：因为程序的最终转向的分支地址 Y 应是 16 位的，所以 Jmp_Table 表的每个分支地址标号定义也应是 16 位的(占用双字节地址)，因而上面程序中键号 0，1，2，3…经过乘 2 运算后才能在 Jmp_Table 基地址的 0，2，4，6…偏移位置取得最终地址。

3.4.3 汇编语言编程小提示

NXP MCU 的汇编语言程序设计将在后面进一步阐述和深入实践。要想真正掌握汇编语言编程，应认真进行编程练习、分析体会、上机调试，通过实践来获得训练和提高。

以下是汇编语言编程中的一些小提示：

(1) 程序是被编译成二进制码放在程序存储器(Flash)中的，通过 PC 递加，自动逐条执行；PC 值指向的下一条将要执行的指令。

(2) 程序处理主要是通过访问各种寄存器、数据存储器(RAM)的各单元实现所需功能要求的，而 CPU 寄存器(A、B、X、Y、CCR)是编程中要频繁用到的工作寄存器。

(3) 理解 MCU 的存储器空间分配图，MCU 寄存器、RAM、Flash 是统一编址在 64KB 地址空间的，每单元 8 位；访问寄存器使用其功能符号形式，访问 RAM 区使用 16 位地址形式($xxxx)。而它们的单元内容一般是 8 位的字节数据形式($xx)。

(4) 主程序通常是：初始化以后，循环等待或原地等待。子程序用标号定义开始，用 RTS 结束，主程序用 BSR 或 JSR 调用之；中断服务子程序也用标号定义开始，结束则用 RTI；需要声明中断向量对应的子程序入口，其执行是当中断发生时自动被执行的。

(5) 由于堆栈指针寄存器 SP 在 CPU 复位时并不一定指向 RAM 存储区空余空间，所以在汇编程序的开始必须进行 SP 指针的初始化设定，使其指向 RAM 区的最底部，否则程序在有堆栈操作时的执行会出现异常。

(6) 适当伪指令：辅助编程，便于修改、理解等。

(7) 规范与格式：大写、缩进、对齐、注释；文件名、子程序名和标号等要有自明性。

(8) 编程方法：熟悉指令，理解范例，套用实践，举一反三；由小到大，优化整理，结合硬件，完备应用。