第3章 指令系统

3.1 指令系统的概述及符号约定

指令是 CPU 执行某种操作的命令。微处理器(MPU)或微控制器(MCU)所能识别全部指令的集合称为指令系统或指令集。指令系统是制造厂家在设计 CPU 时所赋予它的功能，用户必须正确的书写和使用指令。因此学习和掌握指令的功能与应用非常重要，是程序设计的基础。本章将详细的介绍 SPCE061A 指令系统的寻址方式和各种指令。

μ'nSP™单片机指令按其功能可划分为：

1) 数据传送指令，包括立即数到寄存器、寄存器到寄存器、寄存器到存储器存储器到寄存器的数据传送操作；
2) 算术运算，包括加、减、乘运算；
3) 逻辑运算，包括与、或、异或、测试、移位等操作；
4) 转移指令，包括条件转移、无条件转移、中断返回、子程序调用等操作；
5) 控制指令，如开中断、关中断、FIR 滤波器的数据的自由移动等操作。

按寻址方式划分，可分为以下几类：

◆ 立即数寻址

这种寻址方式是操作数以立即数的形式出现，例如：R1 = 0x1234，是把 16 进制数 0x1234 赋给寄存器 R1。

◆ 存储器绝对寻址

这种寻址方式是通过存储器地址来访问存储器中的数据，例如：R1 = [0x2222]，访问 0x2222 单元的数据。

◆ 寄存器寻址

这种寻址方式是操作数在寄存器中，例如：R1 = R2，是把寄存器 R2 中的数据赋给寄存器 R1。

◆ 寄存器间接寻址

这种寻址方式是操作数的地址由寄存器给出，例如：R1 = [BP]，是把由 BP 指向的内存单元的数据送寄存器 R1。

◆ 变址寻址

这种寻址方式下，操作数的地址由基址和偏移量共同给出，例如：R1 = [BP+0x34]。

表 3.1 中的符号是在指令系统叙述过程中所要用到的，在此统一进行约定。

表 3.1 符号约定

R1,R2,R3,R4,R5(BP)	通用寄存器
PC	程序计数器
CS,DS	SR 寄存器中的代码段选择字段和数据段选择字段
NZSC	SR 寄存器中的四个标志位(请参考下面几行)
SR	段寄存器，其中 BIT15～BIT10 对应 DS;BIT9～BIT6 对应 NZSC 标志位; BIT5～BIT0 对应 CS
IM6	6 位（BIT）的立即数
IM16	16 位（BIT）的立即数
A6	6 位地址码
A16	16 位地址码
Rd	目的（destination）寄存器或存储器指针
Rs	源寄存器或存储器指针
→	数据传送符号
MR	由 R4,R3 组成的 32 位结果寄存器（R4 为高字节,R3 为低字节）
&,|,^,	逻辑与记号，逻辑或记号，逻辑异或记号
{}	可选项
[]	寄存器间接寻址标志
++,- -	指针单位字增量，字减量
ss，us	两个有符号数之间的操作，无符号数与有符号数之间的操作
Label	程序标号
FIR	Finite Impulse Response(有限冲击响应),数字信号处理中的一种具有线性相位及任意幅度特性的数字滤波器算法。
N	负标志，N=0 时表示运算前最高有效位为 0，N=1 表示最高有效位为 1。（请参考 2。1 节）
Z	零标志，Z=0 表示运算结果不为 0，Z=1 表示运算结果为 0。
S	符号标志，S=0 表示结果不为负，S=1 表示结果为负数（2 的补数），对于有符号运算，16 位数表示的范围-32768～32768，若结果小于零，则 S＝1。
C	进位标志，C=0 表示运算过程中无进位或有借位产生，C=1 表示有进位或无借位产生。
//	注释符

3.2 数据传送指令

数据传送指令是把源操作数传送到指令所指定的目标地址。数据传送操作属复制性质，而不是搬家性质。指令执行后，源操作数不变，目的操作数为源操作数所代替。通用格式是：

<目的操作数>=<源操作数>

源操作数可以是：立即数、寄存器直接寻址、寄存器间接寻址、直接地址寻址、变址寻址等。

目的操作数可以是：寄存器和直接地址寻址。下面按寻址方式来介绍 SPCE061A 的数据传送指令。各个数据传送指令的执行周期数、指令长度等可参见0。

■ 立即数寻址

【影响标志】 N，Z

【格式】　Rd = IM16　　//16 位的立即数送入目标寄存器 Rd

　　Rd = IM6　　//6 位的立即数扩展成 16 位后送入目标寄存器 Rd

【举例】　设传送前 N=0,Z=1,S=0,C=1

　　R1=0xF001　　//R1 的值变为 0xF001,N=1,Z=0,S=0,C=1

■ 寄存器寻址

【影响标志】 N，Z

【格式】　Rd = Rs　　//将源寄存器 Rs 的数据送给目标寄存器 Rd

【举例】　设传送前 N=0,Z=1,S=0,C=1

　　R2=0xF001　　//R2 的值为 0xF001,N=1,Z=0,S=0,C=1

　　R1=R2　　// R1 的值变为 0xF001,N=1,Z=0,S=0,C=1

■ 直接地址寻址

【影响标志】 N，Z

【格式】　[A6] = Rs

　　//将源寄存器 Rs 中的数据送给以 A6 为地址的存储单元

　　[A16] = Rs　　//把 Rs 数据存储到 A16 指出的存储单元

　　Rd = [A6]　　//把 A6 指定的存储单元数据读到 Rd 寄存器

　　Rd = [A16]　　//把 A16 指定的存储单元数据读到 Rd 寄存器

【举例】　设传送前 N=1,Z=1,S=0,C=1

　　R1=0x0011　　// R1 的值为 0x0011,N=0,Z=0,S=0,C=1

　　[0x0010]=R1　　//[0x0010]单元的值变为 0x0011

■ 变址寻址

【影响标志】 N，Z

【格式】　[BP + IM6] = Rs　　//把 Rs 的值存储到基址指针 BP 与 6 位的立即

　　//数之和指出的存储单元。

　　Rd = [BP +IM6]　　//把基址指针 BP 与 6 位的立即数的和指定的存

　　//储单元数据读到 Rd 寄存器。

【举例】　假设执行前 N=0,Z=1,S=0,C=1

　　R1=0x0010

　　[BP+0x0002]=R1　　//N=0,Z=0,S=0,C=1

■ 寄存器间接寻址

【影响标志】 N，Z

【格式】　[Rd] = Rs　　//把 Rs 的数据存储到 Rd 的值所指的存储单元。

　　//Rd 中存放的是操作数的地址。

【举例】　[++Rd] = Rs　　//首先把 Rd 的值加 1，而后 Rs 的数据存储到 Rd 的

　　//值所指的存储单元间接寻址的存储单元

　　Rd = [Rs++]　　//读取 Rs 的值所指的存储单元的值并存入 Rd，而

　　//后、Rs 的值加 1

[例 3.1]：将 R3 的值保存于 0x25 单元

```
R3 = 0x5678          //把 16 位立即数 0x5678 赋给 R3
```

方法 1：

```
[0x25] = R3          //将 R3 的值存储于 0x25 存储单元，直接地址寻址
```

方法 2：　　//0x25 单元的内容为 0x5678

```
R2 = 0x25            //立即数 0x25 送入 R2 中
[R2] = R3            //将 R3 的值存储于 0x25 存储单元，寄存器间接寻址
```

方法 3：　　//0x25 单元的内容为 0x5678

```
BP = 0x20            //立即数 0x20 送入 BP 中;
[BP + 5] = R3        //将 R3 的值存储于 0x25 存储单元，变址寻址 0x25、单元的
                     //内容为 0x5678
```

[例 3.2]：将 0x25，0x26，0x27 单元清空

方法 1：

```
R1 = 0               //影响标志位：Z=1,N=0
R2 = 0x25            //立即数 0x25 送入 R2 中
[R2++] = R1          //R1 的值存储于 0x25 存储单元，R2=R2 +1
[R2++] = R1          //R1 的值存储于 0x26 存储单元，R2=R2 +1
[R2] = R1            //R1 的值存储于 0x27 存储单元
```

方法 2：

```
R1 = 0               //影响标志位：Z=1,N=0
R2 = 0x27
[R2--] = R1          //R1 的值存储于 0x27 存储单元，R2=R2 - 1
[R2--] = R1          //R1 的值存储于 0x26 存储单元，R2=R2 - 1
[R2] = R1            //R1 的值存储于 0x25 存储单元
```

方法 3：

```
R1 = 0               //影响标志位：Z=1,N=0
R2 = 0x24            //立即数 0x24 送入 R2 中
[++R2] = R1          //R2=R2 +1，R2=0x25,而后 R1 的值存储于 0x25 存储单元
[++R2] = R1          //R2=R2 +1，R2=0x26,而后 R1 的值存储于 0x26 存储单元
[++R2] = R1          //R2=R2 +1，R2=0x27,而后 R1 的值存储于 0x27 存储单元
```

[例 3.3]：用不同方式读取存储器的值。

```
BP = 0x20        //将立即数 0x20 赋给 BP;
R1 = [BP + 5]    //BP + 5=0x25，读取 0x25 单元数据到 R1 中
```

R1 = [0x2345]　//读取 0x2345 单元数据到 R1 中

R1 = [BP]　　//读取 0x20 单元的数据到 R1 中

R1 = [BP++]　//读取 0x20 单元的数据到 R1 中，修改 BP=0x21

R1 = [BP--]　//读取 0x21 单元的数据到 R1 中，修改 BP=0x20

R1 = [++BP]　//修改 BP =0x21，读取 0x21 单元的数据

数据传送指令一览表

<table>
<tr><th>语法</th><th>指令长度
(word)</th><th>影响
标志</th><th>周期数</th></tr>
<tr><td>Rd = IM16</td><td>2</td><td rowspan="13">N,Z</td><td>4</td></tr>
<tr><td>Rd =IM6</td><td rowspan="3">1</td><td>2</td></tr>
<tr><td>Rd = [BP +IM6]</td><td>6</td></tr>
<tr><td>Rd = [A6]</td><td>5</td></tr>
<tr><td>Rd = [A16]</td><td>2</td><td>7</td></tr>
<tr><td>Rd = Rs</td><td rowspan="8">1</td><td>4</td></tr>
<tr><td>Rd = [Rs]</td><td rowspan="4">4</td></tr>
<tr><td>Rd = [Rs++]</td></tr>
<tr><td>Rd = [++Rs]</td></tr>
<tr><td>Rd = [Rs--]</td></tr>
<tr><td>[BP + IM6] = Rs</td><td rowspan="5"></td><td>6</td></tr>
<tr><td>[A6] = Rs</td><td>5</td></tr>
<tr><td>[A16] = Rs</td><td>2</td><td>7</td></tr>
<tr><td>[Rd] = Rs</td><td rowspan="4">1</td><td rowspan="4">6</td></tr>
<tr><td>[++Rd] = Rs</td></tr>
<tr><td>[Rd--] = Rs</td></tr>
<tr><td>[Rd++] = Rs</td></tr>
</table>

注：若目的寄存器 Rd 为 PC,则指令周期数是列表中的后者，且此时运算后所有标志位均不受影响。

除以上介绍的指令外，堆栈（stack）操作也属于一种特殊的数据传送指令下面介绍 SPCE061A 的堆栈操作。堆栈指针 SP 总是指向栈顶的第一个空项，压入一个字后，SP 减 1，将多个寄存器同时压栈总是序号最高的寄存器先入栈，然后依次压入序号较低的寄存器，直到序号最低的寄存器最后入栈。所以，执行指令 PUSH R1,R4 TO [SP] 与 指令 PUSH R4,R1 TO [SP]是等效的。因此，在数据出栈前 SP 加 1，总是先弹出入栈指令中序号最低的寄存器，而后依次弹出序号较高的寄存器。

【格式】 PUSH　　Rx,Ry TO [SP]

POP　　Rx,Ry FROM [SP]

【说明】Rx,Ry 可以是 R1~R4,BP,SP,PC 中的任意两个或一个，执行后将 Rx～Ry 的序列寄存器压栈，或将堆栈中的数据弹入 Rx～Ry 序列寄存器中，压栈操作不影响标志位，出栈操作影响 N,Z 标志，当 Rx,Ry 中含有 SR 时，所有标志位都会改变。压栈、出栈操作的执行周期为 2n + 4，若出栈操作的目的寄存器中含有 PC 时，执行周期为 3n+6。其中 n 是压栈数据的个数。压栈和出栈的指令长度均为 1 字长。

【举例】 PUSH　　R1,R5 to [SP]　　//将 R5,R4,R3,R2,R1 压栈，参见图 3.1

PUSH　　R2,R2 to [SP]　　//将 R2 压栈

PUSH　　R3 to [SP]　　//将 R3 压栈

POP　　R3 from [SP]　　//R3 出栈

POP　　R2,R2 from [SP]　　//R2 出栈

POP　　R1,R5 from [SP]　　//R1,R2,R3,R4,R5 出栈

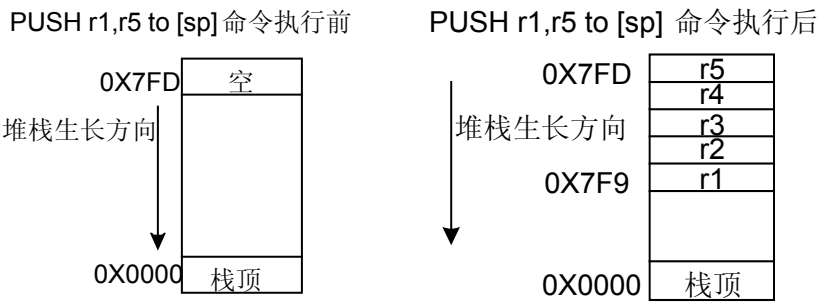


图3.1 堆栈操作

3.3 算术运算

SPCE061A 单片机的算术运算主要包括加,减,乘以及 n 项内积运算。加减运算按是否带进位可分为：不带进位和带进位的加减运算，带进位的加减运算在格式上以及寻址方式与无进位的加减运算类似。这里仍按寻址方式详细介绍不带进位的加减运算，而对带进位的加减运算只作简要说明。

3.3.1 加法运算

加法运算影响标志位：N,Z,S,C。

■立即数寻址（不带进位）

【格式 1】　Rd += IM6　或　Rd = Rd + IM6

【操作】Rd + IM6 → Rd

【说明】Rd 的数据与 6 位(高位扩展成 16 位)立即数相加,结果送 Rd;

【格式 2】　Rd = Rs + IM16

【操作】Rs + IM16 → Rd

【说明】Rd 的数据与 16 位的立即数相加,结果送 Rd;

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0099          //R1 的值为 0x0099，N=0,Z=0,S=0,C=1
R1+=0x0001   //R1 的值变为 0x009A，N=0,Z=0,S=0,C=0
R1+=0xFFFE         //R1 的值变为 0x0098，N=0,Z=0,S=0,C=1
```

■直接地址寻址

【格式 1】　Rd += [A6]　或　Rd = Rd + [A6]

【操作】Rd + [A6] → Rd

【说明】Rd 的数据与 6 位地址指定的存储单元中的数据相加，结果送 Rd。

【格式 2】　Rd = Rs + [A16]

【操作】Rs + [A16] → Rd

【说明】Rs 的数据与 16 位地址指定的存储单元中数据相加，结果送 Rd。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R2=0x0010          //R2 的值为 0x0010,N=0,Z=0,S=0,C=1
[0x0088]=R2        //把 0x0010 送到内存单元 0x0088 中,标志位不变
R1=0xF099          //R1 的值为 0xF099,N=1,Z=0,S=0,C=1
R1+=[0x0088] //R1 的值变为 0xF0A9，N=1,Z=0,S=1,C=0
```

■变址寻址

【格式】Rd += [Bp + IM6]　或　Rd = Rd + [BP + IM6]

【操作】Rd + [Bp + IM6] → Rd

【说明】取基址指针 BP 与 6 位的立即数的和指定的存储单元中的数据与 Rd 相加，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0010          //R1 的值变为 0x0010,N=0,Z=0,S=0,C=1
R2=0x0090          //R2 的值变为 0x0090,N=0,Z=0,S=0,C=1
[0x0015]=r2        //R2 的值送到 0x0015 内存单元,标志位不变
R1+=[BP+0x0015]  //r1 的值变为 0x00A0，N=0,Z=0,S=0,C=0
```

■寄存器寻址

【格式】Rd += Rs

【操作】Rd + Rs → Rd

【说明】Rd 与 Rs 的数据相加，结果送 Rd。

【举例】假设开始时标志位为：N=0,Z=1,S=0,C=1（注：本例为有符号数计算）

```
R1=-8              //R1 的值为-8，实际以补码的形式体现。即 0xFFF8
```

```
                    //N=1,Z=0,S=0,C=1
R2=0xFFFE           //r2 的值为 0xFFFE,N=1, Z=0, S=0, C=1
R1+=R2              //R1 的值变为 0XFFF6 ,N=1,Z=0,S=1,C=1,无溢出。
```

■寄存器间接寻址

【格式 1】　Rd += [Rs];

【操作】　Rd + [Rs] → Rd

【说明】Rd 的数据与 Rs 所指定的存储单元中的数据相加，结果送 Rd

【格式 2】　Rd += [Rs++]

【操作】Rd + [Rs] → Rd, Rs + 1 → Rs

【说明】Rd 的数据与 Rs 所指定的存储单元中的数据相加，结果送 Rd,修改 Rs，Rs=Rs+1

【格式 3】　Rd += [Rs--];

【操作】Rd + [Rs] → Rd, Rs - 1 → Rs

【说明】　Rd 的数据与 Rs 所指定的存储单元中数据相加，结果送 Rd, Rs=Rs- 1

【格式 4】　Rd += [++Rs];

【操作】Rs + 1→ Rs, Rd + [Rs] → Rd

【说明】首先修改 Rs=Rs+1，Rd 的数据与 Rs 所指定的存储单元中的数据相加，结果送 Rd,

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0010           //R1 的值为 0x0010,N=0,Z=0,S=0,C=1
R2=0x0020           //R2 的值为 0x0020,N=0,Z=0,S=0,C=1
[0x0010]=r2         //R2 的值送到内存单元 0x0010 中，标志位不变
R2=0x0010           //R2 的值为 0x0010
R1+=[R2++]          //R1 的值变为 0x0030，N=0,Z=0,S=0,C=0
                    //同时 R2 的值变为 0x0011
```

注意：有符号数的溢出只判断 N 和 S 两位：N!=S 时溢出，N=S 时无溢出。

3.3.2 减法运算

同不带进位的加法运算一样，不带进位的减法运算同样可分为立即数寻址，直接地址寻址，寄存器寻址，寄存器间接寻址等方式。

减法运算影响标志位：N,Z,S,C。

■立即数寻址

【格式 1】　Rd - = IM6　或 Rd = Rd – IM6

【操作】　Rd – IM6 → Rd

【说明】　Rd 的数据减去 6 位(bit)立即数，结果送 Rd。

【格式 2】　Rd = Rs – IM16

【操作】　Rs – IM16 → Rd

【说明】　Rs 的数据减去 16 位(bit)立即数，结果送 Rd。

【举例】　假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0010           //R1 的值为 0x0010，N=0,Z=0,S=0,C=1
R2=0x0001           // R2 的值为 0x0001，N=0,Z=0,S=0,C=1
```

R1-=R2　　// R1 的值为 0x000F，N=0,Z=0,S=0,C=1

■直接地址寻址

【格式 1】　Rd - = [A6]　或　Rd = Rd – [A16]

【操作】　Rd – [A6] → Rd

【说明】　Rd 的数据减去[A6]存储单元中的数据，结果送 Rd。

【格式 2】　Rd = Rs -[A16]

【操作】　Rs – [A16] → Rd

【说明】　Rs 的数据减去[A16]存储单元中的数据，结果送 Rd。

【举例】　假设开始时的标志位为：N=0,Z=1,S=0,C=1

R1=0x0002　　//R1 的值为 0x0002,N=0,Z=0,S=0,C=1

[0x0020]=R1　　//把 R1 的值送到内存单元 0x0020 中,标志位不变。

R2=0x0001　　//R2 的值为 0x0001,N=0,Z=0,S=0,C=1

R2-=[0x0020]　//R2 的值变为 0xFFFF，‘C’为 0，运算过程产生

//借位, N=1,Z=0,S=1

■变址寻址

【格式】　Rd - = [BP + IM6]　或　Rd = Rd – [BP + IM6]

【操作】　Rd – [BP + IM6] → Rd

【说明】　Rd 的值减去基址加变址指定的存储单元的值，结果送 Rd

【举例】　假设开始时的标志位为：N=0,Z=1,S=0,C=1

R1=0x8001　　//R1 的值为 0x8001,N=1,Z=0,S=0,C=1

R2=0x0020　　//R2 的值为 0x0020,N=0,Z=0,S=0,C=1

[0x0010]=R2　　//把 0x0020 送到地址单元 0x0010 中

R1-=[BP+0x0010] //R1 的值变为 0x7FE1,N=0,Z=0,S=1,C=1

■寄存器寻址

【格式】　Rd - = Rs

【操作】　Rd – Rs → Rd

【说明】　寄存器 Rd 的数据减去 Rs 的数据，结果送 Rd 寄存器

【举例】　假设开始时的标志位为：N=0,Z=1,S=0,C=1

R1=32767　　//R1 的初值为 0x7FFF,N=0,Z=0,S=0,C=1

R2=32768　　//R2 的初值为 0x8000, ,N=1,Z=0,S=0,C=1

R1-=R2　　//R1 的值变为 0XFFFF,N=1,Z=0,S=0,C=0

■寄存器间接寻址

【格式 1】　Rd - = [Rs]

【操作】　Rd -[Rs] → Rd

【说明】　Rd 的数据与 Rs 所指定的存储单元中的数据相减，结果送 Rd。

【格式 2】　Rd - = [Rs++]

【操作】　Rd - [Rs] → Rd, Rs+1 → Rs

【说明】　Rd 的数据与 Rs 所指定的存储单元中的数据相减，结果送 Rd, Rs 的值加 1

【格式 3】　Rd - = [Rs--]

【操作】　Rd - [Rs] → Rd, Rs –1 → Rs

【说明】　Rd 的数据与 Rs 所指定的存储单元中的数据相减，结果送 Rd, Rs 的值减 1

【格式 4】　Rd - = [++Rs]
【操作】　Rs + 1 → Rs,Rd - [Rs] → Rd
【说明】　Rs 的值加 1，Rd 的数据与 Rs 所指定单元数据相减，结果送 Rd

3.3.3 带进位的加减运算

由于带进位的加减运算与不带进位的加减运算在寻址方式，周期数，指令长度，影响的标志位均相同，在格式上相似，故这里只给出格式，供读者参考。

- *带进位的加法格式*

Rd += IM6,Carry
Rd = Rd + IM6,Carry
Rd = Rs + IM16,Carry
Rd += [BP + IM6] ,Carry
Rd = Rd + [BP + IM6] ,Carry
Rd += [A6] ,Carry
Rd = Rd + [A6] ,Carry
Rd = Rs + [A16] ,Carry
Rd += Rs,Carry
Rd += [Rs] ,Carry
Rd += [++Rs] ,Carry
Rd += [Rs--],Carry
Rd += [Rs++],Carry

- *带进位的减法格式*

Rd - = IM6,Carry
Rd = Rd - IM6,Carry
Rd = Rs - IM16,Carry
Rd - = [BP + IM6] ,Carry
Rd = Rd - [BP + IM6] ,Carry
Rd - = [A6] ,Carry
Rd = Rd - [A6] ,Carry
Rd = Rs - [A16] ,Carry
Rd - = Rs,Carry
Rd - = [Rs] ,Carry
Rd - = [++Rs] ,Carry
Rd - = [Rs--],Carry
Rd - = [Rs++],Carry

3.3.4 取补运算

取一个数的补码，在计算机中表示为取其反码，再加 1，SPCE061A 也正是这样处理的。取补运算影响标志位 N,Z。

【举例】计算数- 600 与 0x0040 单元数据的差。

R1 = - 600

```
R2=0x0001
[0x0040]=R2
BP = 0x0040        //取该单元地址
R2 = -[BP]         //取此数的补码,-1 的补码为 0xFFFF
R1 += R2       //相加，结果送 R1
```

3.3.5 SPCE061A 的乘法指令

注：Rd,Rs 可用 R1～R4，BP；MR 由 R4、R3 构成，R4 是高位，R3 为低位

【影响标志】无

【格式 1】　MR = Rd * Rs　　或　MR = Rd * Rs,ss

【功能】Rd * Rs → MR

【说明】表示两个有符号数相乘，结果送 MR 寄存器

【格式 2】　MR = Rd * Rs,us

【功能】Rd * Rs → MR

【说明】表示无符号数与有符号数相乘，结果送 MR 寄存器。

【举例】计算一年（365 天）共有多少小时，结果存放 R4(高位)R3(低位)

```
R1 = 365                //R1 的值为 0x016D
R2 = 24                 //R2 的值为 0x0018
MR = R1 * R2,us
                        //计算乘积,结果 R3 的值为 0x2238, R4 的值为 0x0000
```

3.3.6 SPCE061A 的 n 项内积运算指令

【影响标志】无

【格式】　MR = [Rd] * [Rs] {,ss} {,n}

【功能】指针 Rd 与 Rs 所指寄存器地址内有符号数据之间或无符号与有符号字数据之间进行 n 项内积运算，结果存入 MR。符号的缺省选择为 ss，即有符号数据之间的运算。n 的取值为 1～16，缺省值为 1。 内积运算操作如图 3.2所示。

【执行周期】10n＋6

【说明】　当 FIR_MOV ON 时,允许 FIR 运算过程中数据自由移动。为新样本取代旧样本进行数据移动准备：Xn- 4=Xn- 3，Xn- 3=Xn- 2，Xn- 2=Xn- 1。如上图所示当完成一次内积运算后，X1、X2、X3 自动右移，X4 移出。这在数字信号处理中十分有用。比如你要计算连续 4 次采样值的平均值，你即可将采样值放到 X1- - - -X4 中，加权系数放到 C 中，然后完成[Rd] * [Rs] {,ss} {,4}运算，再求平均值。当 FIR_MOV ON 时，运算完后 X1、X2、X3 自动右移，X4 移出，这样我们就可以把下一次的采样值 X0 放到原 X1 的位置，下一次直接[完成 Rd] * [Rs] {,ss} {,4}运算，即 X0-X3 的运算，不必手动移动 Xn。

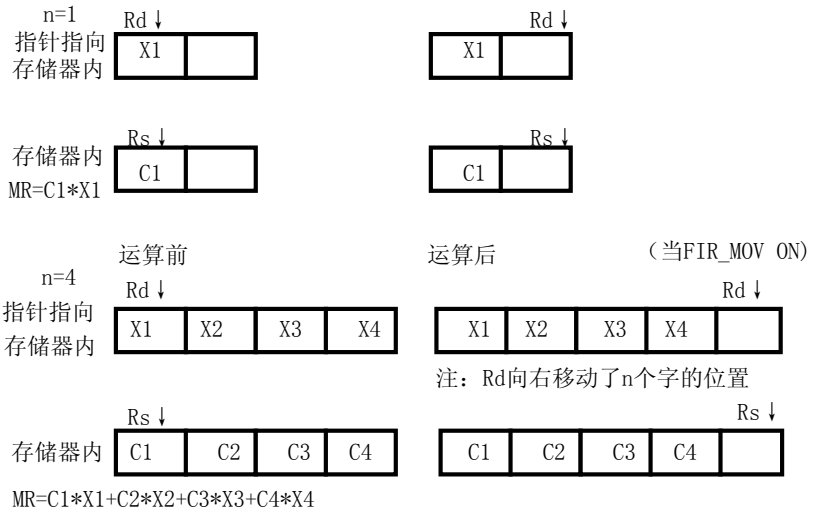


图3.2 内积运算操作示意图示

注意：　(1)Rs 和 Rd 其中任意一个都不能为 R3 或 R4，否则会出错。
(2)Rs 和 Rd 不能同为一个寄存器，否则会出错。

下面为一个内积运算的例子。首先在 IRAM 中定义 8 个变量在程序的第一循环阶段，计算的值为 2*5+3*6 即结果为 1C;第一循环结束后，由于移位的作用，NO_3 的值被 NO_2 的值取代，变为 0x0002，而 NO_2 的值不变，再次循环后的值为：2*5+3*6 结果为 0x001C。该程序比较完整，可在 IDE 内操作。在操作的过程中，通过观察 IDE 中的变量表可以详细了解程序运行的整个流程。

【举例】

```
.IRAM
//*******定义 8 个变量，分别赋初值，NO_7,NO_8 为默认值 0*********//
.VAR      NO_1=0x0001,NO_2=0x0002,NO_3=0x0003, NO_4=0x0004
.VAR      NO_5=0x0005,NO_6=0x0006 ,NO_7,NO_8
 .CODE
.PUBLIC     _MAIN;
_MAIN :
FIR_MOV ON;              //数据自动移动的允许信号
R1=NO_2;                 //定义起始位
R2=NO_5
MR=[R1]*[R2],us,2         //2 项内积运算
[NO_7]=r3                //通过 NO_7,NO_8 可以读出结果
[NO_8]=r4
NOP
JMP     _main            //跳转到主程序
RETF
//*****************************************************************
//*******************************END*****************************
```

//***

3.3.7 比较运算(影响标志位 N,Z,S,C)

比较运算执行两数的减法操作，不存储运算结果，只影响标志位 N、Z、S、C，下面按寻址方式分别介绍比较运算的各条指令。

■立即数寻址

【格式 1】　CMP　Rd,IM6

【说明】　将 Rd 与 6 位（bit）立即数相减。

【格式 2】　CMP　Rd,IM16

【说明】　将 Rd 与和 16 位立即数相减。

■直接地址寻址

【格式 1】　CMP　Rd, [A6]

【说明】此指令将 Rd 的值与 A6 指定地址单元的数据相减。

【格式 2】　CMP　Rd, [A16]

【说明】此指令将 Rd 的值与 A16 指定地址单元的数据相减。

■寄存器寻址

【格式】　CMP　Rd，Rs

【说明】该指令将 Rd 与 Rs 的值相减。

■变址寻址

【格式】CMP　Rd，[BP + IM6]

【说明】该指令将 Rd 与[BP + IM6] 指定地址单元数据相比较。

■寄存器间接寻址

【格式 1】　CMP　Rd，[Rs]

【说明】将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较

【格式 2】　CMP　Rd，[Rs++]

【说明】　将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，并修改 Rs 的值使 Rs 的值加 1。

【格式 3】　CMP　Rd，[Rs--]

【说明】将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，并修改 Rs 的值，使 Rs 值减 1。

【格式 4】　CMP　Rd，[++Rs]

【说明】修改 Rs 的值，使 Rs 加 1，将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，结果影响标志位。

3.4 SPCE061A 的逻辑运算

3.4.1 逻辑与

逻辑与运算影响标志位 N,Z

■立即数寻址

【格式 1】　Rd &= IM6　或　Rd = Rd & IM6

【功能】Rd & IM6 → Rd

【说明】　该指令将 Rd 的数据与 6 位立即数进行逻辑与操作，结果送 Rd 寄存器。

【格式 2】　Rd = Rs & IM16

【功能】Rs & IM16 → Rd

【说明】　该指令将 Rs 的数据与 16 位立即数进行逻辑与操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0010          //R1 的初值为 0x0010，Z=0
R1&=0x000F         //结果为 0，标志位 Z 由 0 变为 1
NOP
```

■直接地址寻址

【格式 1】　Rd &=[A6]　或　Rd ＝ Rd & [A6]

【功能】Rd & [A6] → Rd

【说明】将 Rd 和 A6 指定存储单元数据进行逻辑与操作，结果送 Rd 寄存器

【格式 2】　Rd = Rs & [A16]

【功能】Rs & [A16] → Rd

【说明】Rs 中的数据和 A16 指定存储单元中的数据进行逻辑与操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0010          //R1 赋值为 0x0010，Z=0,N=0
R2=0xFFFF          //R2 赋值为 0xFFFF，Z=0,N=1
[0x000F]=R2
R1&=[0x000F]   //执行后，R1=0x0010，标志位 Z=0,N=0
```

■寄存器寻址

【格式】Rd &= Rs

【功能】Rd & Rs → Rd

【说明】将 Rd 和 Rs 的数据进行逻辑与操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x00FF          //R1 的初值为 0x00FF，Z=0,N=0
R2=0xFFFF          //R1 的初值为 0xFFFF，Z=0,N=1
```

R1&=R2　　//结果为 0x00FF，执行后标志位 N 变为 0，Z=0

■寄存器间接寻址

【格式 1】　Rd &= [Rs]

【功能】Rd & [Rs] → Rd

【说明】　将 Rd 数据与 Rs 指定的存储单元数据进行逻辑与操作结果送 Rd 寄存器。

【格式 2】　Rd &= [Rs++]

【功能】Rd & [Rs] → Rd，Rs + 1 → Rs

【说明】　将 Rd 的数据与 Rs 指定的单元的数据进行逻辑与操作，结果送 Rd 寄存器，修改 Rs 的值，使 Rs 加 1。

【格式 3】　Rd &= [Rs--]

【功能】　Rd & [Rs] → Rd，Rs – 1 → Rs

【说明】　将 Rd 的数据与 Rs 指定的单元的数据进行逻辑与操作，结果送 Rd 寄存器，修改 Rs 的值，使 Rs 减 1。

【格式 4】　Rd &= [++Rs]

【功能】Rs + 1 → Rs，Rd & [Rs] → Rd

【说明】　修改 Rs 的值，Rs 加 1，将 Rd 的数据与 Rs 指定的单元的数据进行逻辑与操作，结果送 Rd 寄存器

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x00FF          //R1 的初值为 0x00FF，Z=0,N=0
R2=0xFFFF          //R1 的初值为 0xFFFF，Z=0,N=1
[0x0001]=R2
R2=0x0001
R1&=[R2]           //结果为 0x00FF，执行后标志位 N 变为 0，Z=0
```

3.4.2 逻辑或

逻辑或影响标志位 N,Z

■立即数寻址

【格式 1】　Rd |= IM6　　或　Rd = Rd | IM6

【功能】Rd | IM6 → Rd

【说明】该指令将 Rd 的数据与 6 位立即数进行逻辑或操作，结果送 Rd 寄存器。

【格式 2】　Rd = Rs | IM16

【功能】Rs | IM16 → Rd

【说明】　该指令将 Rs 的数据与 16 位立即数进行逻辑或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x00FF      //R1 的初值为 0x00FF，Z=0,N=0
R1|=0xF000         //R1 的值变为 0xF0FF, Z=0,N=1
```

■直接地址寻址

【格式 1】　Rd |=[A6]　　或　Rd = Rd | [A6]

【功能】Rd | [A6] → Rd

【说明】将 Rd 和 A6 指定单元数据进行逻辑或操作，结果送 Rd 寄存器

【格式 2】　Rd = Rs | [A16]

【功能】Rs | [A16] → Rd

【说明】Rs 的数据和 A16 指定单元数据进行逻辑或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x00FF       //R1 的初值为 0x00FF，Z=0,N=0
R1|=[0x0009]    //R1 值变为 0x00FF, Z=0,N=0，[0x0009]的值默认为 0。
```

■寄存器寻址

【格式】Rd |= Rs

【功能】Rd | Rs → Rd

【说明】将 Rd 和 Rs 的数据进行逻辑或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0000           //R1 的初值为 0x0000,Z=1,N=0
R2=0xFFFF
R1|=R2              //R1 的值变为 0xFFFF, Z=0,N=1
```

■寄存器间接寻址

【格式 1】　Rd |= [Rs]

【功能】Rd | [Rs] → Rd

【说明】将 Rd 的数据与 Rs 指定的单元的数据进行逻辑或操作，结果送 Rd 寄存器。

【格式 2】　Rd |= [Rs++]

【功能】Rd | [Rs] → Rd，Rs + 1 → Rs

【说明】将 Rd 的数据与 Rs 指定的单元的数据进行逻辑或操作，结果送 Rd 寄存器，修改 Rs 的值，使 Rs 加 1。

【格式 3】　Rd |= [Rs--]

【功能】Rd | [Rs] → Rd，Rs – 1 → Rs

【说明】将 Rd 的数据与 Rs 指定的单元的数据进行逻辑或操作，结果送 Rd 寄存器，修改 Rs 的值，使 Rs 减 1。

【格式 4】　Rd |= [++Rs]

【功能】Rs + 1 → Rs，Rd | [Rs] → Rd

【说明】首先，Rs 的值将加 1，Rd 与 Rs 指定的单元的数据进行逻辑或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0000       //R1 的初值为 0x0000,Z=1,N=0
R2=0xFFFF       //把 0xFFFF 送到地址单元[0x0002]中
[0x0002]=R2
R2=0x0002       //R2 的值为 0x0002 ,Z=0,N=0
R1|=[R2]            //R1 的值变为 0xFFFF, Z=0,N=1
```

3.4.3 逻辑异或

逻辑异或影响标志位：N, Z

■立即数寻址

【格式 1】　Rd ^= IM6　或　Rd = Rd ^ IM6

【功能】Rd ^ IM6 → Rd

【说明】指令将 Rd 的数据与 6 位立即数进行逻辑异或操作，结果送 Rd 寄存器。

【格式 2】　Rd = Rs ^ IM16

【功能】Rs ^ IM16 → Rd

【说明】该指令将 Rs 的数据与 16 位立即数进行逻辑异或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0F00          //R1 的初值为 0x0F00,Z=0,N=0
R1^=0x0FFF             //R1 的值变为 0x00FF, Z=0,N=0
```

■直接地址寻址

【格式 1】　Rd ^=[A6] 或　Rd ＝ Rd ^ [A6]

【功能】Rd ^ [A6] → Rd

【说明】将 Rd 和 A6 指定存储单元中数据进行逻辑异或操作，结果送 Rd 寄存器

【格式 2】　Rd = Rs ^ [A16]

【功能】Rs ^ [A16] → Rd

【说明】Rs 的数据和 A16 指定存储单元中的数据进行逻辑异或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0F00          //R1 的初值为 0x0F00,Z=0,N=0
R2=0x0FF0
[0x0010]=R2
R1^=[0x0010]       //R1 的值变为 0x00F0, Z=0,N=0
```

■寄存器寻址

【格式】Rd ^= Rs

【功能】Rd ^ Rs → Rd

【说明】将 Rd 和 Rs 的数据进行逻辑异或操作，结果送 Rd 寄存器。

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0E01          //R1 的初值为 0x0E01,Z=0,N=0
R2=0x0FF1          // R2 的初值为 0x0FF1,Z=0,N=0
R1^=R2             //R1 的值变为 0x01F0, Z=0,N=0
```

■寄存器间接寻址

【格式 1】　Rd ^= [Rs]

【功能】　Rd ^ [Rs]→ Rd

【说明】将 Rd 的数据与 Rs 指定的存储单元中的数据进行逻辑异或操作，结果送 Rd 寄存器。

【格式 2】　Rd ^= [Rs++]

【功能】Rd ^ [Rs] → Rd，Rs + 1 → Rs

【说明】将 Rd 的数据与 Rs 指定的存储单元中的数据进行逻辑异或操作，结果送

Rd 寄存器，修改 Rs，使 Rs 加 1。

【格式 3】　Rd ^= [Rs--]

【功能】Rd ^ [Rs] → Rd，Rs – 1 → Rs

【说明】将 Rd 的数据与 Rs 指定的存储单元中的数据进行逻辑异或操作，结果送 Rd 寄存器，修改 Rs，使 Rs 减 1

【格式 4】　Rd ^= [++Rs]

【功能】Rs + 1→ Rs ,Rd ^ [Rs] → Rd

【说明】修改 Rs，使 Rs 加 1，Rd 的数据与 Rs 指定的存储单元中的数据进行逻辑异或操作，结果送 Rd 寄存器

【举例】假设开始时的标志位为：N=0,Z=1,S=0,C=1

```
R1=0x0000            //R1 的初值为 0x0000,Z=1，N=0
R2=0xFFFF            //R2 的初值为 0xFFFF，N=1，Z=0
[0x0002]=R2
R2=0x0002
R1^=[R2]             //R1 的值变为 0xFFFF, Z=0,N=1
```

3.4.4 测试（TEST）

测试指令执行指定两个数的逻辑与操作，但不写入寄存器，结果影响 N，Z 标志。

■立即数寻址

【格式】TEST　Rd，IM6

【说明】该指令将 Rd 与 IM6 进行逻辑与操作，不存储结果

【格式】TEST　Rd，IM16

【说明】将 Rd 与 IM16 进行逻辑与操作，不存储结果，只影响 N，Z 标志

【举例】假设初始时标志位分别为：N=0,Z=1,S=0,C=1.

```
R1=0x0E01          //R1 的初值为 0x0E01,N=0,Z=0,S=0,C=1
TEST R1,0x0000     //测试 R1 和 0x0000 相与的结果,N=0,Z=1,S=0,C=1
JZ   loop1         //Z 为 1 跳转到 loop1 处，此时测试结果为 0
NOP
NOP
loop1:             //标号
R1=0x0000
```

■直接地址寻址

【格式 1】　TEST　Rd, [A6]

【说明】该指令将 Rd 与 A6 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响标志 N，Z

【格式 2】　TEST　Rd, [A16]

【说明】该指令将 Rd 与 A16 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志

【举例】假设初始时标志位分别为：N=0,Z=1,S=0,C=1.

```
R1=0x0E01          //R1 的初值为 0x0E01,N=0,Z=0,S=0,C=1
R2=0x0011          //将 0x0011 送到内存单元[0x0000]中
```

```
        [0x0000]=R2
        TEST R1,[0x0000]
                              //测试 R1 和 0x0000 单元的值相与的结果,此时
                              //N=0,Z=0,S=0,C=1
        JNZ   loop1           //Z 为 0 时跳转到 loop1 处，此时测试结果不为 0
        NOP
        NOP
    loop1:                    //标号
        R1=0x0000
```

■变址寻址

【格式】TEST　Rd，[BP+IM6]

【说明】该指令将 Rd 与 BP+IM6 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志。

【举例】假设初始时标志位分别为：N=0,Z=1,S=0,C=1.

```
        R1=0x0E01             //R1 的初值为 0x0E01,N=0,Z=0,S=0,C=1
        R2=0x0011             //将 0x0011 送到内存单元[0x0000]中
        [0x0000]=R2
        TEST R1,[BP+0x0000] //已知 BP 的值为 0
                              //测试‘R1 和 BP+0x0000 单元的值相与的结果
        //测试后 N=0,Z=0,S=0,C=1
        JNZ   loop1           //Z 不为 0 时跳转到 loop1 处，此时测试结果不为 0
        NOP
        NOP
        loop1:
        R1+=1
```

■寄存器寻址

【格式】TEST　Rd，Rs

【说明】该指令将 Rd 与 Rs 的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志位。

【举例】假设初始时标志位分别为：N=0,Z=1,S=0,C=1.

```
        R1=0x0000       //r1 的初值为 0x0000,N=0,Z=1,S=0,C=1
        R2=0x1111       //r2 的初值为 0x1111,N=0,Z=0,S=0,C=1
        TEST R1,R2      //测试‘R1 和 R2 的相与的结果’,N=0,Z=1,S=0,C=1
        JZ   loop1      //‘Z=1’时跳转到 loop1 处，此时测试结果为 0
        NOP
        loop1:              //标号
        R1-=1
```

■寄存器间接寻址

【格式 1】　TEST　Rd，[Rs]

【说明】该指令将 Rd 与[Rs] 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志位

【格式 2】　TEST　　Rd，[Rs++]

【说明】该指令将 Rd 与[Rs] 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志位，并且使 Rs 值加 1。

【格式 3】　TEST　　Rd，[Rs--]

【说明】该指令将 Rd 与 Rs 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志位，并且使 Rs 值减 1。

【格式 4】　TEST　　Rd，[++Rs]

【说明】该指令首先使 Rs 值加 1，而后 Rd 与 Rs 指定存储单元中的数据进行逻辑与操作，不存储结果，只影响 N，Z 标志位。

【举例】假设初始时标志位分别为：N=0,Z=1,S=0,C=1.

```
R1=0x0000       //R1 的初值为 0x0000,N=0,Z=1,S=0,C=1
R2=0x1111       //[0x0001]的初值为 0x1111,N=0,Z=0,S=0,C=1
[0x0001]=R2
R2=0x0001
TEST R1,R2      //测试‘R1 和 R2 相与的结果’,N=0,Z=1,S=0,C=1
JZ  loop1       //‘Z=1’时跳转到 loop1 处，此时测试结果为 0
nop
loop1:                  //标号
R1=0x0000
```

……

3.4.5 SPCE061A 的移位操作

SPCE061A 的移位运算包括逻辑左移、逻辑右移、循环左移、循环右移、算术右移等操作，移位的同时还可进行其他运算，如加、减、比较、取负、与、或、异或、测试等。指令长度 1，指令周期 3/8,影响 N,Z 标志。由于硬件原因对于移位操作，每条指令可以移 1～4 位。

■逻辑左移（LSL）

【格式】Rd ＝ Rs LSL　n

【说明】该指令对 Rs 进行 n（可设为 1～4）位逻辑左移，将 Rs 高 n 位移入 SB 寄存器，同时 Rs 的低 n（1～4）位用 0 补足，结果送 Rd 寄存器。

【举例】逻辑左移 3 位。

寄存器移位前的状态为：

SB	S3	S2	S1	S0

RS	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态为：

SB	S0	B15	B14	B13

Rd	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	0	0	0

程序：

```
R1=0xF00F           //R1 的初值为 0xF00F
R1=R1 LSL 3             //R1 左移 3 位后的值变为 0x8078
```

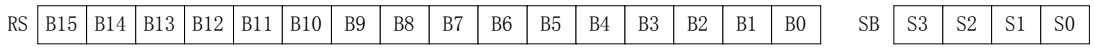
■逻辑右移(LSR)

【格式】Rd ＝ Rs LSR　n

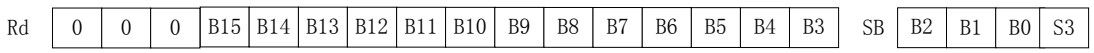
【说明】该指令对 Rs 进行 n（可设为 1～4）位逻辑右移，将 Rs 低 n 位移入 SB 寄存器同时 Rs 的高 n（1～4）位用 0 补足，结果送 Rd 寄存器。

【举例】　逻辑右移 3 位。

寄存器移位前的状态为：



寄存器移位后的状态为：



程序：

```
R1=0xF00F          //R1 的初值为 0xF00F
R1=R1 LSR 3        //R1 右移 3 位后的值变为 0x1E01
```

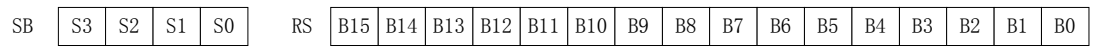
■循环左移(ROL)

【格式】Rd ＝ Rs ROL　n

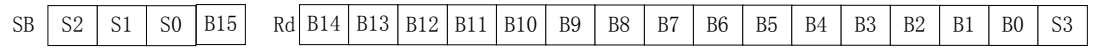
【说明】该指令对 Rs 进行 n（可设为 1～4）位循环左移，将 Rs 的高 n 位移入 S 寄存器，同时移动 SB 寄存器的高 n 位移入 Rs 的低 n 位，结果送 Rd 寄存器。

【举例】循环左移 1 位。

移位前的各位状态如下：



寄存器移位后的状态为：



程序：

```
R1=0x0010          //R1 的初值为 0x0010
R1=R1 ROL 1        //R1 循环左移 1 位后的值变为 0x0020
```

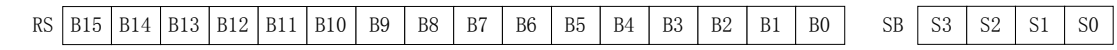
■循环右移(ROR)

【格式】Rd ＝ Rs ROR　n

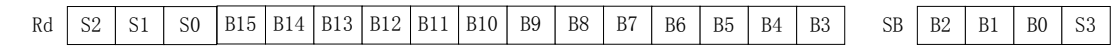
【说明】该指令对 Rs 进行 n（可设为 1～4）位循环右移，将 Rs 的低 n 位移入 SB 寄存器，同时移动 SB 寄存器的低 n 位移入 Rs 的高 n 位，结果送 Rd 寄存器。

【举例】循环右移 3 位。

移位前的各位状态如下：



移位后的各位状态如下：



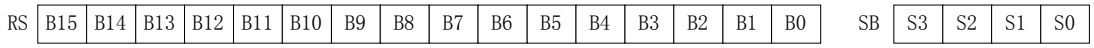
■算术右移（ASR）

【格式】Rd ＝ Rs ASR n

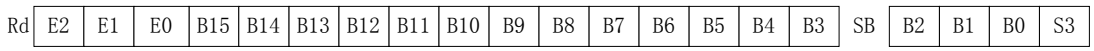
【说明】　该指令将 Rs 算术右移 n（可设为 1～4）位，将 Rs 的低 n 位移入 SB 寄存器，并对最高有效位进行符号扩展，结果送 Rd 寄存器。该指令适合有符号数的移位操作。

【举例】算术右移 3 位。

移位前的各位状态如下：



移位后的各位状态如下：



其中 E2,E1,E0 是 Rs 中最高有效位的符号扩展位。

程序：

```
R1=0xF111          //R1 的初值为 0xF111
R1=R1 ASR 3        // R1 移位后的值为 0xFE22
```

另外，SPCE061A 在进行移位的同时，还可进行其它运算，现以算术右移为例说明如下：

【格式】Rd += Rs ASR n {,Carry}

//将 Rs 移位后的结果与 Rd（带进位）相加，结果送 Rd 寄存器。

Rd - = Rs ASR n {,Carry}

//将 Rs 移位后的结果与 Rd（带进位）相减，结果送 Rd 寄存器。

CMP Rd, Rs ASR n

//将 Rs 移位后的结果与 Rd 相比较。

Rd = - Rs ASR n

//取 Rs 移位后的结果的负值，结果送 Rd 寄存器。

Rd &= Rs ASR n

//将 Rs 移位后的结果与 Rd 逻辑相与，结果送 Rd 寄存器。

Rd |= Rs ASR n

//将 Rs 移位后的结果与 Rd 相或，结果送 Rd 寄存器。

Rd ^= Rs ASR n

//将 Rs 移位后的结果与 Rd 相异或，结果送 Rd 寄存器。

TEST Rd , Rs ASR n

//测试 Rd 中 Rs 移位后结果中为 1 的位。

【举例】

```
R1=0xFF00      //R1 的初值为 0xFF00
R2=0x0001
R2+=R1 ASR 1
```

//R1 算术右移 1 位后的值变为 0xFF80,再与 R2 相加,最后 R2 的值为 0xFF81

3.5 SPCE061A 的控制转移类指令

SPCE061A 的控制转移类指令主要有中断,中断返回,子程序调用,子程序返回,跳转等指令,以下列表说明其用法和功能。

表3.1 条件转移指令列表(指令长度 2,周期 3/5)

助记符	操作数类型	条件	标志位状态
JB	无符号数	小于	C=0
JNB		不小于	C=1
JAE		大于等于	C=1
JNAE		小于	C=0
JA		大于	Z=0 and c=1
JNA		不大于	Not (z =0 and c=1)
JBE		小于等于	Not(z=0 and c=1)
JNBE		大于	Z=0 and c=1
JGE	有符号数	大于等于	S=0
JNGE		小于	S=1
JL		小于	S=1
JNL		大于等于	S=0
JLE	有符号数	小于等于	Not (z=0 and s=0)
JNLE		大于	z=0 and s=0
JG		大于	z=0 and s=0
JNG		小于等于	Not (z=0 and s=0)
JVC		无溢出	N=s
JVS		溢出	N!=s
JCC		进位为 0	C=0
JCS		进位为 1	C=1
JSC		符号位为 0	S=0
JSS		符号位为 1	S=1
JNE		不等于	Z=0
JNZ		非 0	Z=0
JZ		为负	Z=1
JMI		为负	N=1
JPL		为正	N=0
JE		相等	Z=1

下面对表 3.1中的每一种操作类型举一个例子进行说明：

- 无符号数的跳转指令

判 C 的转移指令 JB 和 JNB

```
JB    loop1
JNB   loop1
```

这两条指令都是通过判断标志位’C’的值来决定程序的走向，它们的执行过程见图 3.3

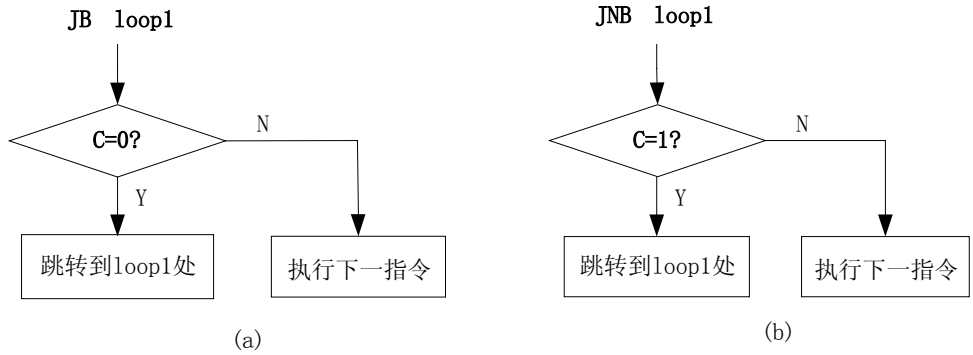


图3.3　指令 JB 和 JNB 的执行过程
(a)JB 的执行过程　（b）JNB 的执行过程

下面举两个例子来说明：
举例(JB)：

```
R1=0x0001          //设初值
R2=0x0006
CMP R1,R2          //比较 R1、R2 的大小
JB loop1
R1=0x0000
 loop1:
      R1=0x0000
      R2=0x0000
        …
```

举例(JNB)：

```
R1=0x0010          //设初值
R2=0x0006
CMP R1,R2          //比较 R1,R2 的大小
JNB loop1
R1=0x0000
  loop1:
      R1=0x0000
      R2=0x0000
      …
```

- 有符号数的跳转指令

判 S 跳转指令 JGE 和 JNGE

```
JGE   loop1
JNGE   loop1
```

JGE 表示“大于等于”通过对标志位 S 的判断来决定程序走向。S 为 0 则产生跳转，若 S 为 1 则继续执行下一语句。JNGE 表示“小于”，当 S 为 1 时产生跳转，S

为 0 时则继续执行下一语句。这两条指令的执行过程如图 3.4

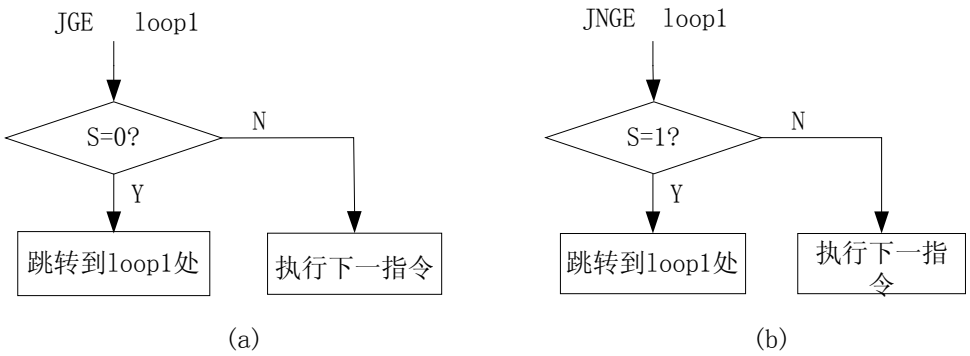


图3.4 指令 JGE 和 JNGE 的执行过程
(a)JGE 的执行过程　（b）JNGE 的执行过程

举例(JGE)：

```
        R1=2            //赋初值,分别为有符号数
        R2=-3
        CMP R1,R2   //比较 R1,R2 的值
       JGE loop1      // S=0 表明 R1 大于等于 R2，跳转到 loop1，否则继续下一条
                       //指令
        R2=0x0000
  loop1:
        R1=0x0000
            …
```

举例(JNGE) ：

```
        R1=-2           //赋初值 ,分别为有符号数
        R2=3
        CMP R1,R2   //比较 R1,R2 的值
        JNGE    loop1
                       //R1 小于 R2 时，跳转到 loop1，此时 S=1 否则继续下条指令。
        R2=0x0000
  loop1:
         R1=0x0000
             …
```

- 其它跳转指令

判 C 跳转指令 JCC 和 JCS

```
    JCC      loop1
    JCS      loop1
```

JCC 是以“C=0”作为判断的标准决定程序的跳转。JCS 是以 “C=1”作为判断的标准决定程序的跳转, 这两条指令的执行过程如图 3.5。

■

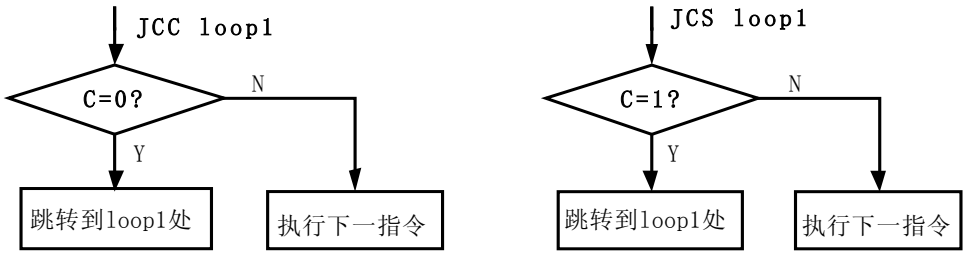


图3.5 指令 JCC 和 JCS 的执行过程
(a)JCC 的执行过程　（b）JCS 的执行过程

举例(JCC) ：

```
        R1=0x0020       //赋初值
        R1+=1
        JCC    loop1
                        //如果 C 为 0，跳转到 loop1，否则继续下一条指令
        R1=0x0000
    loop1:
        R1=0x0000
          …
```

举例(JCS)：

```
        R1=0x0002     //赋初值
        R1-=1
        JCS    loop1
                        //如果 C 为 1，跳转到 loop1，否则继续下一条指令
        R1=0x0000
    loop1:
        R1=0x0000
          …
```

3.6 伪指令

μ'nSP™汇编伪指令与汇编指令不同，它不会被编译，而仅被用来控制汇编器的操作。伪指令的作用有点像语言中的标点符号，它能使语言中的句子所表达意思的结构更加清晰而成为语言中不可缺少的一部分。在汇编语言中正确使用伪指令，不仅能使程序的可读性增强，且使汇编器的编译效率倍增。

3.6.1 伪指令的语法格式及特点

伪指令可以写在程序文件中的任意位置，但在其前面必须用一个小圆点引导，以便与汇编指令区分开。伪指令行中方括弧里的参量是任选项，即不是必须带有的参量。如果某一个参量使用双重方括弧括起来，则说明这个任选项参量本身就必须带着方括弧。例如[[count]]表示引用该任选参量时必须写出[count]才可。
μ'nSP™的汇编器规定的标准伪指令不必区分字母的大小写，亦即书写伪指令时既可全用大写，也可全用小写，甚至可以大小写混用。但所有定义的标号包括宏名、结构名、结构变量名、段名及程序名则一律区分其字母的大小写。

3.6.2 伪指令符号约定

bank	存储器的页单元
ROM	程序存储器
RAM	随机数据存储器
label	程序标号
value	常量数值
IEEE	一种标准的指数格式的实数表达方式
variable	变量名
number	数据的数目
ASCII	数值或符号的 ASCII 代码
argument#	参量表中的参量序号
filename	文件名
[]	任选项

3.6.3 标准伪指令

伪指令依照其用途可分为五类：定义类、存储类、存储定义类、条件类及汇编方式类。它们的具体分类及用途详见表 3.2

表3.2 伪指令的类别一览

类别	用　途	伪指令
定义类	用于对以下内容进行定义的伪指令： 1．程序； 2．程序中所用数据的性质、范围或结构； 3．宏或结构 4．程序 5．其它	1　CODE、DATA、TEXT 2　IRAM、ISRAM、ORAM、OSRAM、RAM、SRAM 3　MACRO、MACEXIT、ENDM 4　PROC、ENDP、STRUCT、ENDS； 5．DEFINE、VAR、PUBLIC、EXTERNAL、EQU、VDEF
存储类	以指定的数据类型存储数据或设定程序地址等	DW、DD、FLOAT、DOUBLE、END
存储定	定义若干指定数据类型的数据存储单元	DUP

义类		
条件类	对汇编指令进行条件汇编	IF、ELSE、ENDIF；IFMA、IFDEF、IFNDEF
汇编方式类	包含汇编文件或创建用户定义段	INCLUDE；SECTION

■ **定义类伪指令**

DEFINE

【类　　别】定义类

【功能描述】定义常量符号

【语法格式】.DEFINE　　variable [value][，…]

【应用解释】给常量符号所赋之值既可是一已定义过的常量符号，亦可是一表达式。切忌符号超前引用，即如果赋值引用的符号不是在引用前定义的，则会出现“非法超前引用”的错误。

【举　　例】
```
.DEFINE      BODY          1;
.DEFINE      IO_PORT       0x7016;
.IFDEF       BODY;
          R1=0xFFFF;
        [IO_PORT]=R1;
.ENDIF
```

PUBLIC

【类　　别】定义类

【功能描述】声明将被引用在其它文件中的全局标号

【语法格式】.PUBLIC　　label[，label][，…]

【应用解释】本伪指令用来在文件中声明将被引用在外部文件中的全局标号。故在外部文件中用伪指令 EXTERNAL 所声明的标号必须是用 PUBLIC 伪指令声明过的。类似地，当要声明多个全局标号时，要用逗号(，)将每一标号分开。

【举　　例】
```
.PUBLIC   sym1
   //声明要引用在其它文件中的全局标号
.PUBLIC    sym1，sym2
  //声明多个全局标号需用逗号将每一标号分开，空格会被忽略。
```

EXTERNAL

【类　　别】定义类

【功能描述】在某文件中声明已在其它文件里定义过的标号、变量或函数

【语法格式】.EXTERNAL label[,label][,...]

【应用解释】这是在已定义过某些标号、变量或函数的文件之外的文件里，要引用这些标号、变量或函数之前需对其进行声明时所要用到的伪指令。以此避免标号、变量或函数在不同的文件里被引用时容易发生的重复定义错误。如果同时要声明多个这样的外部标号，需用逗号(，)将每一个标号分开。本伪指令后禁止将两个或多个标号进行算术或逻辑操作。

【举　　例】
```
.EXTERNAL num_var1，num_var2;
   //声明在其它文件中定义过的标号
.EXTERNAL int SACM_A2000_Initial();
    //声明在其它文件中定义过的函数
```

```
.EXTERNAL _Keycode
                    //声明在其它文件中定义过的变量
```

RAM

【类　　别】定义类

【功能描述】切换定义预定义段 RAM

【语法格式】.RAM

【应用解释】RAM 段用来存放无初始值的变量。RAM 段不能跨 bank 链接，且在链接时所有与其同名或同属性的各段都会被合并在一起而被定位在 RAM 中.。（参见后面「段的定义与使用」内容）。

【举　　例】.RAM

```
        start: .DW   ?       //申请一个整型数据单元
    .VAR num , max;          //定义变量 num、max
```

SRAM

【类　　别】定义类

【功能描述】切换定义预定义段 SRAM

【语法格式】.SRAM

【应用解释】SRAM 段的性质基本与 RAM 段相同。区别在于链接时它只能在 RAM 中占有前面 64（0~63）个字的数据单元（参见后面「段的定义与使用」内容）。

【举　　例】.SRAM

```
        start: .DW    ?      //申请一个 16 位整型数据单元
    .VAR sum,name            //定义变量 sum、name
```

IRAM

【类　　别】定义类

【功能描述】切换定义预定义段 IRAM

【语法格式】.IRAM

【应用解释】IRAM 段用来存放具有初始值的变量。此段亦不可跨 bank 链接，且在链接时所有与其同名或同属性的各段都会被合并在一起，将会同时被分配空间于 ROM 及 RAM 中（参见后面「段的定义与使用」内容）。

【举　　例】.IRAM

```
        .VSR sum=0x0001   //定义变量，并赋初始值为 0x0001
       Storage: .DW  0x20  //在申请的整型数据单元中存放存 0x20
```

ISRAM

【类　　别】定义类

【功能描述】切换定义预定义段 ISRAM

【语法格式】.ISRAM

【应用解释】ISRAM 段与 IRAM 段的性质基本相同。区别是 ISRAM 段在 RAM 中只能被分配在前面 64(0~63)个字的数据单元（参见后面「段的定义与使用」内容）。

【举　　例】.ISRAM

```
        Storage: .DW 0x20 //在申请的整型数据单元中存放存 0x20
    .VAR sum=0x0020      //定义变量，并赋初始值为 0x0020
```

ORAM

【类　　别】定义类

【功能描述】切换定义预定义段 ORAM
【语法格式】.ORAM
【应用解释】ORAM 段用来存放无初始值的变量。ORAM 段也不能跨 bank 链接，且在链接时同一目标文件中所有与其同名或同属性的各段都会被合并在一起而被定位在 RAM 中。但在一个项目里的不同目标文件中所有与 ORAM 段同名或同属性的各段会被重叠在一起。当编程者需要在不同的文件里使用相同的变量空间时，适合用本伪指令（参见后面「段的定义与使用」内容）。
【举　　例】.ORAM

```
.VAR num                //定义变量 num
Buf1: .DW  10  DUP(?)   //申请 10 个整型的数据单元
```

OSRAM
【类　　别】定义类
【功能描述】切换定义预定义段 OSRAM
【语法格式】.OSRAM
【应用解释】OSRAM 段与 ORAM 段的性质基本相同。区别在于链接时 OSRAM 段只能在 RAM 中被分配在前面 64（0~63）个字的数据单元（参见后面「段的定义与使用」内容）。
【举　　例】.OSRAM

```
Buf2: .DW   ?          //申请一个整型的数据单元
.VAR num               //定义变量 num
```

VAR
【类　　别】定义类
【功能描述】定义变量并为变量置初始值
【语法格式】.VAR　variable[=value]
【应用解释】在程序的任何部位都可用本伪指令定义变量并为变量置初始值。若定义多个具有初始值的变量，需用逗号(，)将每一个变量赋值分开（若要为变量赋初始值，则必须声明在 IRAM 或 ISRAM 段中，若在其它段中申明将无效，它将被初始化为 0；同样要申请具有初值的单元也必须在 IRAM 或在 ISRAM 中申请，若在其它段中申请也将无效，它将被初始化为 0）。
【举　　例】.RAM

```
.VAR   s1，s2，s3；     //无赋初值
```

DATA
【类　　别】定义类
【功能描述】切换定义预定义段 DATA
【语法格式】.DATA
【应用解释】程序中所有数据都可以存放在 DATA 段的位置上。DATA 段可以跨 bank 链接。链接时，不同文件中所有与其同名或同属性的各段会被分开置入 ROM 地址中，而同一文件中所有与 DATA 同名或同属性的各段会被合并在一起置入 ROM 地址中。
【举　　例】.DATA

```
tone_table: .DW 5,8,6,9,3,8,0 ;
```

CODE
【类　　别】定义类

【功能描述】切换定义预定义段 CODE
【语法格式】.CODE
【应用解释】所有程序指令和数据都可以存放在 CODE 段下。汇编器对程序指令进行汇编时，遇到各类预定义段会采取相应的方式进行处理。换言之，预定义段的段名是汇编器进行汇编操作的一种地址分类标签。CODE 段不能跨 bank 链接。不同文件中所有与其同名的预定义段或同属性的用户定义段会被分开地定位在 ROM 地址中，而同一文件中所有与 CODE 同名或同属性的各段则会被合并在一起置入在 ROM 地址中。
【举　　例】.CODE
```
        .PUBLIC _MAIN;
_MAIN:
```
TEXT
【类　　别】定义类
【功能描述】切换定义预定义段 TEXT
【语法格式】.TEXT
【应用解释】TEXT 段与 CODE 段的性质基本相同。唯一的区别是它只能被链接到第一个单元即零页 ROM 中，且所有与其同名或同属性的各段均会被合并在一起而被定位到此。该段主要用来写中断程序代码（参见后面「段的定义与使用」内容）。
【举　　例】.TEXT
```
        . PUBLIC _IRQ0
        . PUBLIC _IRQ1
        . PUBLIC _IRQ2
        _IRQ0:                  //中断子程序入口
      PUSH  R1,R5 to [SP]
```
PROC
【类　　别】定义类
【功能描述】开始程序的定义
【语法格式】label:　.PROC
【应用解释】本伪指令用于起始程序的定义，应与结束程序定义的伪指令 ENDP 成对使用（参见后面「过程的定义与调用」内容）。
【举　　例】test1: .PROC
```
          PUSH BP,BP TO [SP];
          BP=SP+1;
          R1 = [0x7015];
          POP BP,BP FROM [SP];
          RETF;
        .ENDP
```
ENDP
【类　　别】定义类
【功能描述】结束程序的定义
【语法格式】ENDP
【应用解释】程序定义的起始要用伪指令 PROC，终止则用本伪指令。两条伪指令应成对使用（参见后面「过程的定义与调用」内容）。
【举　　例】Test1: .PROC

```
PUSH BP to [SP]
R2 = 0x0020;
R2 + = 0x0010;
R1= R2;
POP BP from [SP];
RETF;
.ENDP
```

STRUCT

【类　　别】定义类

【功能描述】起始结构的定义

【语法格式】label:　.STRUCT

【应用解释】本伪指令用于起始结构的定义，应与结束结构定义的伪指令 ENDS 成对使用（参见后面「结构的定义与引用」内容）。

【举　　例】

```
Person: .STRUCT     //定义一个结构‘Person’
    Xm: .DW   ‘joe’      //它包含三个变量 Xm，Nl，Xb
    Nl: .DW    20         //且分别被初始化为’joe’,20,’boy’
    Xb: .DW   ‘boy’
    .ENDS                 //结构定义结束
```

ENDS

【类　　别】定义类

【功能描述】结束结构的定义

【语法格式】.ENDS

【应用解释】本伪指令用来终止结构的定义。起始结构的定义是由伪指令 STRUCT 给出的。二者须成对使用（参见后面「结构的定义与引用」内容）。

【举　　例】同上例

MACRO

【类　　别】定义类

【功能描述】起始宏的定义

【语法格式】label:　.MACRO　　args

【应用解释】本伪指令用来起始宏定义。结束宏定义则用 ENDM 伪指令，二者应成对使用（参见后面「宏的定义与调用」内容）。

【举　　例】

```
Personal_info: .MACRO   arg1,arg2,arg3
    Name:  .DW  arg1
    Age:   .DW  arg2
           .ENDM
```

ENDM

【类　　别】定义类

【功能描述】结束宏的定义

【语法格式】.ENDM

【应用解释】本伪指令用来结束宏定义。它应与起始宏定义的伪指令 MACRO 成对使用（参见后面「宏的定义与调用」内容）。

【举　　例】同上例

MACEXIT

【类　　别】定义类
【功能描述】退出宏定义
【语法格式】.MACEXIT
【应用解释】从宏定义里退出是立即且无条件的。本伪指令并非用来终止宏定义。它常与条件汇编用在一起，意指条件满足时从宏定义里退出，对 ENDM 伪指令后面的指令进行汇编；条件不满足时，继续汇编 MACEXIT 伪指令后的宏定义内的其它指令。不必担心从宏里退出时条件会失衡，因为 MACEXIT 伪指令会自动将条件恢复到原来的均衡状态。
【举　　例】
```
eserve：.MACRO    arg1,arg2
        .VAR   arg1
        .IF    count
        .MACEXIT
        .ENDIF
        .VAR   count -1
    .ENDM
```
EQU
【类　　别】定义类
【功能描述】为标号赋值
【语法格式】label:　.EQU　　value
【应用解释】给标号所赋之值可为在此标号之前已定义过的某一符号常量或为一表达式。同样要注意符号不能超前引用。所有通过 EQU 定义的标号及值将会输出至 Linker 的符号文件。
【举　　例】label:　.EQU　　10
VDEF
【类　　别】定义类
【功能描述】为标号赋值
【语法格式】lable:　.VDEF　value
【应用解释】可以在程序的任何部位用本伪指令为标号重复赋值。
【举　　例】int0：.VDEF　　0x005F

■　**存储类伪指令**

DW
【类　　别】存储类
【功能描述】以 16 位整型数据的形式来存储常量或变量
【语法格式】[label:]　.DW　　[value][,value][,...]
【应用解释】本伪指令是把一系列 16 位整型常量或变量值存入连续的整型数据单元中。整型常量或变量值可以是多种类型的操作数。需用逗号(，)将多个数值分开；特别地，若定义的数值较多，需多行书写时，在一行当中最后一个数值输毕，用回车键结束此行输入之前切莫丢掉逗号。若存储的常量中含有 ASCII 字符串，则必须用引号(' ')将其括起来。如果.DW 后面未输入任何数值，则会自动存入一个整型零常量。
【举　　例】
```
.ISRAM
Label:
.DW  ‘Hello’,0x000D
```

```
//在申请的序列整型数据单元中存放字符串常量'Hello'的
//ASCII 码和整型常量 0x000D
```

DD

【类　　别】存储类

【功能描述】以 32 位长整型数据的形式存储双字常量或变量

【语法格式】label：.DD　[value][，value][，…]

【应用解释】申请若干连续的 32 位长整型数据单元来存放双字常量或变量序列。常量（变量值）与常量（或变量值）之间应当用逗号(，)隔开。存放的常量可以任何一种数制格式输入，但最终由汇编器将其转换成十六进制格式存放。

【举　　例】
```
label1：.DD　0x0A10 //存放一个十六进制数 0x0A10
label2：.DD ‘High’
  //以 4 个 32 位长整型数据单元来存放字符串‘High’的 ASCII
  //码 0x0048，0x0069，0x0067，0x0068
```

DOUBLE

【类　　别】存储类

【功能描述】以双精度浮点型实数数据的形式存储实数常量或变量

【语法格式】label：.DOUBLE　value[，value][，…]

【应用解释】把实数常量或变量的值转换成以 IEEE 格式表示的双精度浮点型数据并存储。存储多个这类常量或变量的值时需用逗号(，)将其分开。以本伪指令存储的常量必须是实数。

【举　　例】
```
label1：.DOUBLE　178.125
//将实数 178.125 转储为 0x0000,0x0000，0x4400，0x4066
label2：.DOUBLE 100.0，-178.125
//将实数 100.0;-178.125 转换成 0x0000,0x0000,0x0000,
//0x4059;0x0000,0x0000,0x4400,0xC066 并存储
```

FLOAT

【类　　别】存储类

【功能描述】以单精度浮点型实数数据单元来存储实数常量或变量

【语法格式】label:　.FLOAT value[,value][,...]

【应用解释】本伪指令用来把实数常量或变量值转换成以 IEEE 格式表示的单精度浮点型数据并存储。当要存储多个这类常量或变量值时需用逗号(，)将其分开。所谓单精度浮点型实数是指数的精度保证 6 位有效位。若超过 6 位，则对第七位进行入舍处理后，其后所有位均会被舍去。

【举　　例】
```
label1：.FLOAT 178.125
  //把数值 178.125 转换成 0x43322000 格式并存储
label2：.FLOAT　10，125，-178.125
  // 把 数 值　100,125,-178.125　转 换 成　0x42C80000，
  //0x42FA0000，0x0C3322000 格式并存储。
```

END

【类　　别】存储类

【功能描述】用来结束程序文件或结束包含文件

【语法格式】.END
【应用解释】本伪指令用在程序文件中意味该文件结束（不可再包含其它文件）。
【举　　例】.END

■ **存储定义类**

DUP
【类　　别】存储定义类
【功能描述】本伪指令与 DW、FLOAT、DD、DOUBLE 存储类伪指令组合在一起用于存放若干个具有相同数值的常量；或者申请若干个备用的整型、单精度浮点型、长整型以及双精度浮点型数据单元。
【语法格式 1】[label:]　.DW　number　DUP(value)
【说　　明 1】存储若干具有相同数值的 16 位的整型数据。
【语法格式 2】[label:]　.FLOAT　number　DUP(value)
【说　　明 2】存储若干具有相同数值的 32 位的单精度浮点型实数数据。
【语法格式 3】[label:]　.DD　number　DUP(value)
【说　　明 3】存储若干具有相同数值的 32 位的长整型数据。
【语法格式 4】[label:]　.DOUBLE　number　DUP(value)
【说　　明 4】存储若干具有相数值的 64 位双精度浮点型实数数据。
【举　　例】

```
label1: .DW  20   DUP(0)
        //存储 20 个值为 0 的整型数据
label2: .DW  20   DUP(0x0FF)
        //存储 20 个值为 0x0FF 的整型数据
label3: .DW  9  DUP(20)
        //存储 9 个值为 20 的整型数据
label4: .DW  11   DUP(0x20)
        //存储 11 个值为 0x20 的整型数据
label5: .FLOAT   20   DUP(10.982)
        //存储 20 个值为 10.982 单精度浮点型数据
lable6: .DOUBLE  5 DUP(5223.29)
        //存储 5 个值为 5223.29 双精度浮点型数据, FLOAT 型或
        //DOUBLE 型浮点数均用 IEEE 格式来表示
label7: .DD  20   DUP(0)
        //存储 20 个值为 0 的长整型数据
label8: .DD  20   DUP(0x12345678)
        //存储 20 个值为 0x12345678 长整型数据
label9: .DW  5   DUP(?)
        //申请 5 个整型的数据单元。
```

■ **条件类伪指令**

IF
【类　　别】条件类
【功能描述】引出在条件汇编结果为真时所要汇编的程序指令

【语法格式】.IF value
【应用解释】所谓条件汇编结果为真是指参量 value 的值不为零。value 既可为在此之前已定义过的某一符号常量或一字符串常量，亦可为一算术表达式。
【举　　例】
```
.DEFINE   var1   0x01
    .IF     var1
        var2 = var1 + 0x10 ;
    .ENDIF
```

ELSE
【类　　别】条件类
【功能描述】引出 IF 伪指令设置的条件汇编结果为假时所要汇编的程序指令。
【语法格式】.ELSE
【应用解释】 若本伪指令前面的 IF 伪指令设置的条件汇编结果为假时引出另一部分汇编程序指令。本伪指令必须与 IF 伪指令结合使用。
【举　　例】
```
.IF (Cond1)
    [0x7016] = R1
      //条件表达式 Cond1 结果为真时的若干汇编指令
  .ELSE
    [0x7016] = R2
      //条件表达式 Cond1 结果为假时的若干汇编指令
  .ENDIF
```

ENDIF
【类　　别】条件类
【功能描述】用来结束条件汇编组合的定义
【语法格式】.ENDIF
【应用解释】本伪指令用来结束条件汇编组合定义。它必须与 IF 伪指令成对使用，否则会产生“条件不匹配”的错误。
【举　　例】
```
.IF (Const1)
        R1 = Const1
  .ENDIF //结束条件汇编，其后的程序指令或数据会接着被汇编
```

IFDEF
【类　　别】条件类
【功能描述】引出若某变量已被赋过值时所要汇编的程序指令
【语法格式】.IFDEF variable
【应用解释】根据本伪指令后面的变量名，汇编器会搜索符号表。若搜索到说明该变量已被赋过值，则会对本伪指令下面的程序指令进行汇编；否则会跳过该段程序指令而直接对 ELSE 或 ENDIF 伪指令后面的程序指令进行汇编。
【举　　例】
```
.Define   name0 = 1;
    .IFDEF    name0
        [0x7010] = R1;
  .ENDIF
```

IFNDEF
【类　　别】条件类
【功能描述】引出某变量还未被赋值时所要汇编的程序指令

【语法格式】.IFNDEF　variable
【应用解释】根据本伪指令后面的变量名，汇编器首先搜索符号表。若在符号表中未查到有该变量名存在，则会对本伪指令下面的汇编指令进行汇编；否则，便会对 ELSE 或 ENDIF 伪指令后面的汇编语句进行汇编。显然，本伪指令与 IFDEF 伪指令的作用相反。
【举　例】
```
.IFNDEF      SPCE
       NOP;
.ELSE
          [0x7010] = R1;
.ENDIF
```

IFMA
【类　别】条件类
【功能描述】引出若指定的宏参量存在时所要汇编的程序指令
【语法格式】.IFMA　argument#
【应用解释】本伪指令必须用在宏内。参量 argument#是指宏定义参量表中的参量序号。汇编器据此序号查找宏调用行上的参量值表，如果在该表内与序号对应的参量值存在，就对本伪指令下面的汇编指令进行汇编；否则便会对 ELSE 或 ENDIF 伪指令后的汇编指令进行汇编。特例情况是当 argument#为零，则汇编器只有在宏调用里不含有参量值时认为条件为真而进行相应的操作。
【举　例】.IFMA 3 //如果宏调用参量值表中的第三个参量值存在，则条件结果为真

■ 汇编方式类

INCLUDE
【类　别】汇编方式类
【功能描述】在汇编文件里包含某个文件
【语法格式】.INCLUDE　filename
【应用解释】本伪指令用来通知汇编器把指定的文件包含在文件中一起进行汇编。其中参量 filename 包括文件的扩展名，且可指明搜索文件所需的路径。注意，每一条 INCLUDE 伪指令只能包含一个文件。已包含某些文件的文件可再被包含在其它文件中。
【举　例】
```
.INCLUDE      hardware.inc
.INCLUDE      key.h
.INCLUDE      hardware.h
```

SECTION
【类　别】汇编方式类
【功能描述】创建用户定义段
【语法格式】label:　.SECTION　.attribute
【应用解释】除了上面提到的一些预定义段用于存放指令代码或数据以外，编程者还可以根据需要用本伪指令定义某些段。其中属性参量 attribute 可以是以下预定义段名当中的任意一个：CODE，DATA，TEXT，RAM，SRAM，IRAM，ISRAM，ORAM，OSRAM（参见后面「段的定义与使用」内容）。
【举　例】
```
section1: .SECTION  .CODE
          //定义一个段名为 section1 的段，其链接属性与预定义段 CODE 相同
```

3.6.4 宏定义与调用

宏定义

所谓宏（Macro）是指在源程序里将一序列的源指令行用一个简单的宏名（Macro Name）所取代。这样做的好处是使程序的可读性增强。

宏在使用之前一定要先经过定义。可分别用伪指令.MACRO 和.ENDM 来起始和结束宏定义；定义的宏名将被存入标号域。在汇编器首次编译通过汇编指令时，先将宏定义存储起来，待指令中遇有被调用的宏名则会用同名宏定义里的序列源指令行取代此宏名。宏定义里可以包括宏参数，这些参数可被代入除注释域之外的任何域内。虚参数不能含有空格。

宏标号

宏定义里可以用显式标号（由用户定义），亦可用隐含标号（由汇编器自动定义）。汇编器不会改变用户定义的显式标号。在宏标号后加上后缀符‘#’则表明该标号为隐含标号，汇编器会自动生成一个后缀数字符号‘_X_XXXX’（X 表示一位数符，XXXX 表示 4 位扩展数符）来取代这个隐含标号中的后缀符‘#’，可见下面程序汇编例子。隐含标号中的字母字符及其后缀数符总共不能超过 32 个字符。

```
instruction：  .MACRO arg，val
              arg
               lab#：  .DW   val ；
               .ENDM
                     //调用前面定义的宏：
               instruction   NOP，7
                     //汇编后会产生以下结果：
               NOP ；
               lab_1_6416：  .DW   7 ；
```

宏调用

在调用宏时，可以使用任何类型的参数：直接型、间接型、字符串型或寄存器型。只有字符串型参数才能含有空格，但必须用引号将此空格括起，即’ ’/” ”。（而字符串参数中的单引号，必须用双引号将其括起，即“ ‘ ”）。只要在宏嵌套中的形参名相同，则这些参数就可以穿过嵌套的宏使用。宏嵌套使用唯一所受的限制是内存空间的容量。宏的多个参数应以逗号隔开，参数前的空格及 Tab 键都将被忽略。单有一个逗号，而后面未带有任何参数会被汇编器表示为参数丢失错误。

宏参数分隔符

在一个宏体中，宏参数的有效分隔为标点符号中的逗号，即“，”。

宏内字符串连接符

符号‘@’（40H）是字符串连接符。注意，字符串连接只能在宏内进行。

助记符搜索顺序

通常，汇编器以表 3.3列出的顺序来搜索各种助记符或符号：

表3.3 汇编器助记符搜索顺序

1	记符表
2	定义表
3	编器伪指令表
4	段名表

宏应用举例

例 1：数的比较

```
cmp_number:   .MACRO    arg1
              .IFMA     0
                 .MACEXIT
              .ENDIF
              .IF    1==arg1
        month:   .DW    1 ;
                 .MACEXIT
              .ENDIF
              .IF    2==arg1
        month:    .DW    2 ;
                  .MACEXIT
              .ENDIF
              .IF    3==arg1
       month:     .DW    3 ;
                  .MACEXIT
              .ENDIF
              .IF    4==arg1
        month:    .DW    4 ;
                  .MACEXIT
              .ENDIF
              .IF    5==arg1
        month:    .DW    5 ;
                  .MACEXIT
              .ENDIF
              .IF    6==arg1
        month:    .DW    6
                  .MACEXIT
              .ENDIF
              .ENDM
```

例 2：将标号名传入程序代码

```
store_label: .MACRO    arg1
```

```
    .DW    @arg1 ;
    .ENDM
       //调用前面定义的宏
    store_label  label1
       //汇编器将宏展开成：
    .DW    label1 ;
```

例 3 ：操作数域内的宏参数替换

```
employee_info1:  .MACRO arg1，arg2，arg3
name:          .DW    arg1 ;
department:    .DW    arg2;
date_hired:    .DD    arg3;
.ENDM
       //调用前面定义的宏：
employee_info1   "John Doe",  personnel，101085
       //汇编器将宏展开成：
name:          .DW    "John Doe" ;
department:    .DW    personnel ;
date_hired:    .DD    101085 ;
```

例 4：将宏参数传入标号域。这样做可以使程序的结构改变。

```
employee_info2:  .MACRO arg1，arg2，arg3
   arg1:  .DW    0x30 ;
   arg2:  .DW    0x10 ;
   arg3:  .DD    1999;
   .ENDM
      //调用前面定义的宏
employee_info2      name，department，date_hired
       //汇编器将宏展开成：
name:        .DW  0x30 ;
department: .DW  0x10 ;
date_hired: .DD  1999;
```

例 5：宏的递归调用。在本例宏的递归调用中，由参数 arg1（计数）控制着递归的次数。宏的每一次递归中保存有 4 个字型数据，其值分别由参数 arg2、arg3、arg4 和 arg5 来指定。每执行一次递归调用计数 arg1 减 1。

```
reserve:  .MACRO arg1，arg2，arg3，arg4，arg5
count:         .VDEF  arg1 ;
               .IF    count==0
                  .MACEXIT
               .ENDIF
count:         .VDEF  count-1
               .DW    arg2，arg3，arg4，arg5 ;
               reserve      count，arg2，arg3，arg4，arg5
```

```
            .ENDM
        //调用前面定义的宏
reserve      10，0x0A，0x0B，0x0C，0x0D
        //汇编后会产生以下结果
count:    .VAR   10 ;
          .IF    count==0;
             .MACEXIT
          .ENDIF
count:    .VAR   count-1 ;
          .DW    0x0A，0x0B，0x0C，0x0D
          reserve   count，0x0A，0x0B，0x0C，0x0D
             . . .
count:    .VAR   count ;
          .IF    count==0
             .MACEXIT
          .ENDM
             . . .
          .ENDM
```

一个递归的宏调用另一个递归的宏是完全合法的，这样的调用可以至多层。不必担心从宏里退出时条件会失衡，汇编器会自动将条件恢复到原来的均衡状态。

3.6.5 段的定义与调用

段其实就是应用在汇编器 Xasm16 中的地址标签。Xasm16 除了定义的预定义段以外，还可由用户自己定义段。

预定义段

在 Xasm16 里共定义有 9 个预定义段：CODE、DATA、TEXT、ORAM、OSRAM、RAM、IRAM、SRAM 及 ISRAM 段。这些预定义段都分别被规定了以下内容：

段内存储数据类型：指令/数据，无初始值的变量/有初始值的变量；

存储介质类型：ROM/RAM（SRAM）；

存储范围：零页（或零页中前 64 个字）或当前页/整个 64 页；

定位排放方式：合并排放/重叠排放。

当用户以这些预定义段的伪指令来定义自己程序的数据块以后，Xasm16 在汇编时会采取相应的措施进行处理，实际上为链接器的链接处理贴好了地址标签。各预定义段的具体规定可参见相应的预定义段的伪指令内容介绍。

用户定义段

为了使程序在链接时具有更大的灵活性，用户可以用伪指令.SECTION 来定义段。定义的段名最多不可超过 32 个字符，且最多可定义 4096 个段，但不可嵌套使用。用户段定义的格式为：

label：　.SECTION .attribute

其中属性参数*.attribute* 可以是上述 9 个预定义段中的任意一个，表明用户定义段的链接属性与这个预定义段相同。定义一个段之后，可以将该段名作为助记符，用来进行

段的转换。详细可参见.SECTION 伪指令内容。
举例：

```
.CODE
        //设置 CODE 预定义段
NOP;
.DATA
        //转换到 DATA 预定义段
.DW  0x20
        //该字节将被存放到 DATA 段
section1: .SECTION .CODE
        // 定义一个新的段，其属性与 CODE 预定义段相同
R3 = R3-0x10 ;     // 该指令将被存入 section1 段
.CODE              // 转换到 CODE 段
R1 = 1;            // 该指令将被存入 CODE 段
.section1          // 转换到用户定义段 section1
R1 = R1 + 2;       // 该指令将被存入 section1
.DW  0x30
        // 任何用户定义段都可包含代码或数据或同时包含二者
.TEXT              // 转换到 TEXT 预定义段
```

3.6.6 结构的定义与调用

像在 ANSI-C 那样，汇编器可以把不同类型的数据组织在一个结构体里，为处理复杂的数据结构提供了手段，并为在过程间传递不同类型的参数提供了便利。
结构的定义
结构作为一种数据构造类型在μ'nSP™汇编语言程序中也要经历定义—说明—使用的过程。在程序中使用结构时，首先要对结构的组成进行描述，即结构的定义。定义的一般格式如下：
结构名：.STRUCT　//定义结构开始
数据存储类型定义
　.*ENDS*　//定义结构结束
结构的定义以伪指令.STRUCT 和 ENDS 作为标识符。结构名由用户命名，命名原则与标号等相同。在两个伪指令之间包围的是组成该结构的各成员项数据存储类型的定义。
例如：

```
test1: .STRUCT        //定义了一个结构 'test1'
ad: .DW 10            //它包括 3 个成员'ad',  'bs',  'gh',
bs: .DW 'abcd'
gh: .DD 0x0FFFC       //且分别被初始化为 10, 'abcd', 0x0FFFC
.ENDS                 //结构定义结束
```

结构变量的定义——结构的说明
某个结构一经定义后，便可指明使用该结构的具体对象，这被称为结构的说明，其一般形式如下：
结构变量名：　.结构名 [结构成员表]

其中[结构成员表]用来存放结构变量中成员的值。例如：

```
Stru_var1:  .test1 [20,'ad',0x7D]
Stru_var2:  .test1 [10,,0x7D] //第二个成员未被存入新值，因此它的初值可被保留。
```

结构变量的引用

结构是不同数据类型的若干数据变量的集合体。在程序中使用结构时不能把结构作为一个整体来参加数据处理，参加各种运算和操作的应是结构中各个成员项数据。结构成员项引用的一般形式为：

结构变量名 . 成员名

例如：

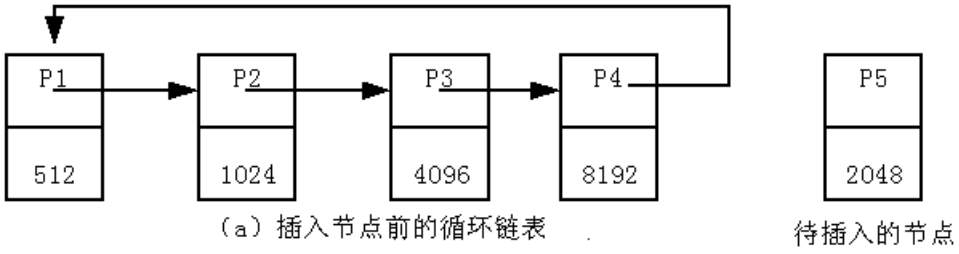
```
R1 + = [stru_var1.ad]
// 'stru_var1'是一个定义过的结构变量，'ad'是它的一个成员。
```

结构应用举例

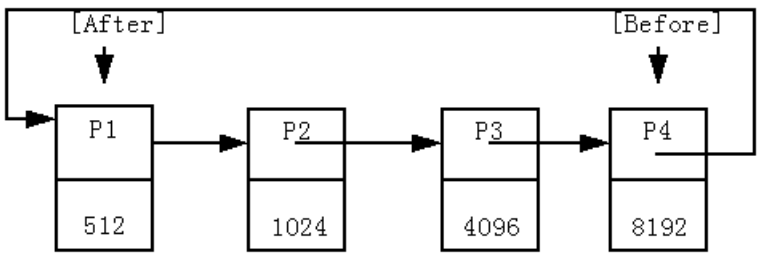
下面举一个在单向循环链表中插入节点的例子

已知一个单向循环链表，每个节点都是一个结构，内含一个编号和结构指针（此指针指向下一个结构针），现在有一个 Id 号为 2048 的结构，要求把该结构插入到如图 3.6所示的一个单向循环链表中，每个结构的 ID 号如图 3.6(a)所示（链表中各节点的成员项 Id 是由小到大的顺序排列的）。

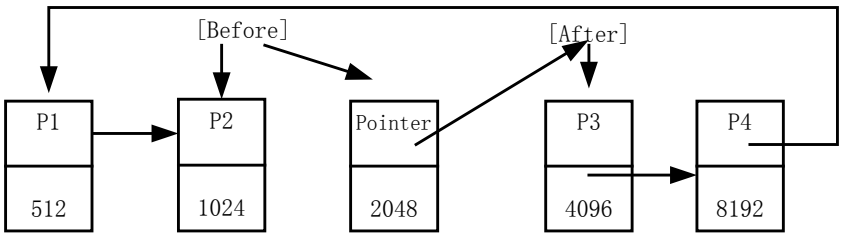
算法说明：初始化时，用变量 Before 存 Id 值最大的结构的地址，用变量 After 存 Id 值最小的结构的地址，Bp 存 Id 值最小的结构的地址，如图 3.6(b)所示。将待插入循环链表的结构的 Id 值(即[Insert.Id])与最大 Id 值（即[Stu_Var4+1]）比较，如果[Insert.Id]> [Stu_Var4+1]，则待插入的节点是插在 Id 值最小的节点和 Id 值最大的节点间。如果[Insert.Id]<[Stu_Var4+1],则[Insert.Id]与[BP+1]比较。如果[Insert.Id]>[BP+1],那么[Before]=BP（BP 存的是前一个结构的地址），BP=[BP]（[BP]存的是结构成员 Pointer，即下一个结构的地址），[After]=BP（After 存的是下一个结构的地址）；如果[Insert.Id]<[BP+1]，则找到插入点的位置如 图 3.6(c)所示。我们可以看到在第二个节点和第三个节点已经间插入一个新的节点，如图 3.6(d)所示。



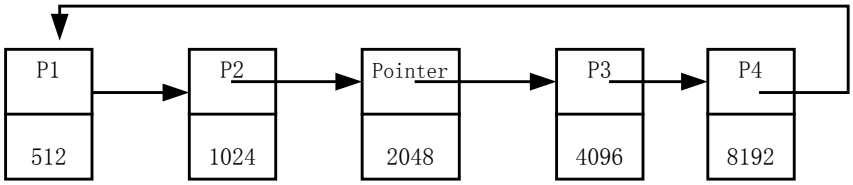
（a）插入节点前的循环链表　　待插入的节点



（b）初始化后的状态



（c）找到插入点位置后的状态



（d）插入节点后的循环链表

图3.6 在单向循环链表中插入节点的过程

程序流程图如图 3.7所示：

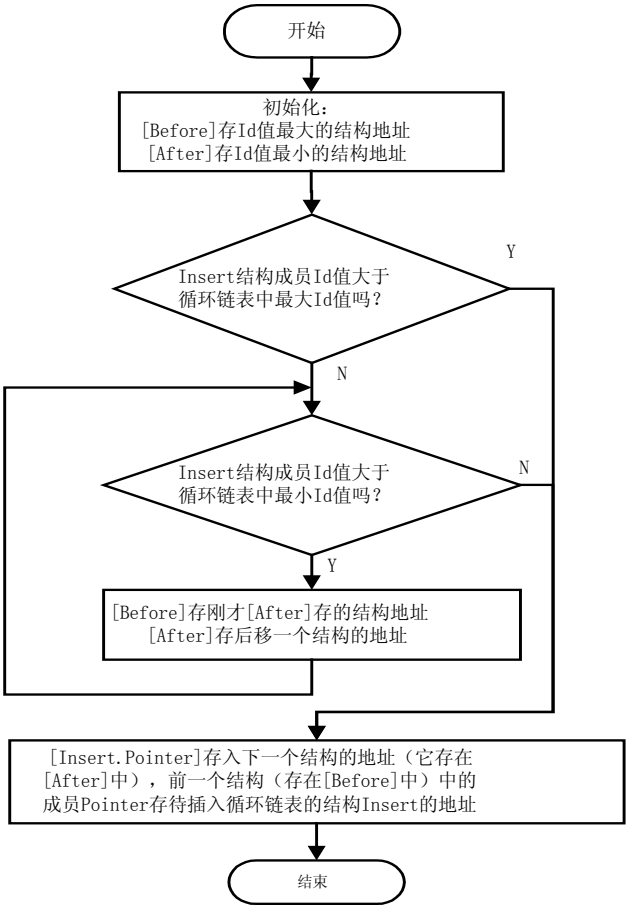


图3.7 程序流程图

程序3-1 在单向循环链表中插入节点

```
.ISRAMPerson: .STRUCT              //定义结构开始
      Pointer: .DW 0               //初始化结构成员 Pointer 为 0
      Id: .DW 100                  //初始化结构成员 Id 为 100
.ENDS                              //定义结构结束

Stu_Var1: .Person[Stu_Var2,512]        //定义结构变量 1
.VAR Before,After;                 //before 存 Id 最大的结构地址,after 存 id 值最小的结构地址
Stu_Var2: .Person[Stu_Var3,1024]       //定义结构变量 2
Stu_Var3: .Person[Stu_Var4,4096]       //定义结构变量 3
Stu_Var4: .Person[Stu_Var1,8192]       //定义结构变量 4
Insert: .Person[,2048];                //定义待插入的结构变量
.CODE
.PUBLIC  _MAIN;
_MAIN:
       R2=Stu_Var1;
       [After]=R2;                 //[After]存 Id 值最小的循环链表节点
       BP=R2                       //BP 存的是结构的地址，[BP]存的是结构成员 Pointer
                                   //的值[BP+1]存的是结构成员 Id 的值
       R2=Stu_Var4;
       [Before]=R2;                //[Before]存 Id 最大的结构的地址
       R1=[Insert.Id];             //R1 存待插入的结构的成员 Id 的值

  cmp R1,[Stu_Var4+1];             //与值最大的 Id 比较大小
       JA first                    //大于则跳转
next:
       R2=[BP+1];                  //BP 为结构地址，R2 存 Id 值
       cmp R1,R2;                  //比较两个 Id 的大小
       JNA first;                  //小于则跳转
       [Before]=BP;                //BP 存的是前一个结构的地址
       BP=[BP];                    //[BP]存的是结构成员 Pointer，即下一个结构的地址
       [After]=BP                  //After 存的是下一个结构的地址
       JMP next;
first:                             //找到插入节点的位置
       R2=[After];
       [Insert.Pointer]=R2;        //[Insert.Pointer]存下一个结构的地址

    R1=[Before];
       R2=Insert;
       [R1]=R2;                    //前一个结构成员 Pointer 存待插入循环链表的结构地
                                   //址 MainLoop:
```

```
JMP  MainLoop;
```

3.6.7 过程的定义与调用

过程实际可以是一个子程序块。它有点类似 ANSI-C 中的函数，可以把一个复杂、规模较大的程序由整化零成一个个简单的过程，以便程序的结构化。

过程的定义
过程的定义就是编写完成某一功能的子程序块，用伪指令.PROC 和.ENDP 作为定义的标识符。定义的一般格式为：
过程名：　.PROC
程序指令列表
RETF
.ENDP
由此格式可以看出，过程的定义主要由过程名和两个伪指令之间的过程体组成。过程名由用户命名，其命名规则同标号。例如：

```
qw:  .PROC                          //定义一个过程‘qw’
label1:
     R1 + = 0x20;
     R2 = R1 ;
     JMP   label1 ;
     RETF ;
     .ENDP                          //过程定义结束
```

过程的调用
在程序中调用一个过程时，程序控制就从调用程序中转移到被调用的过程，且从其起始位置开始执行该过程的指令。在执行完过程体中各条指令并执行到 RETF 指令时，程序控制就返回调用过程时原来断点位置继续执行下面的指令。过程调用的一般格式为：
CALL 过程名
例如：

```
sub1:  .PROC                        //定义一个过程‘sub1’
label1:
    R1 + = 0x0020
    R2 = R1 ;
    JMP   label1 ;
    RETF ;
.ENDP                               //过程定义结束
……;
CALL   sub1                         //调用过程‘sub1’
```

3.6.8 伪指令的应用举例

下面通过例子来讲解μ'nSP™伪指令的用法：

在举例前，我们先来看看 RAM、SRAM、ORAM、OSRAM、IRAM、ISRAM、DATA、TEXT、CODE 几个段的区别，如图 3.8所示。

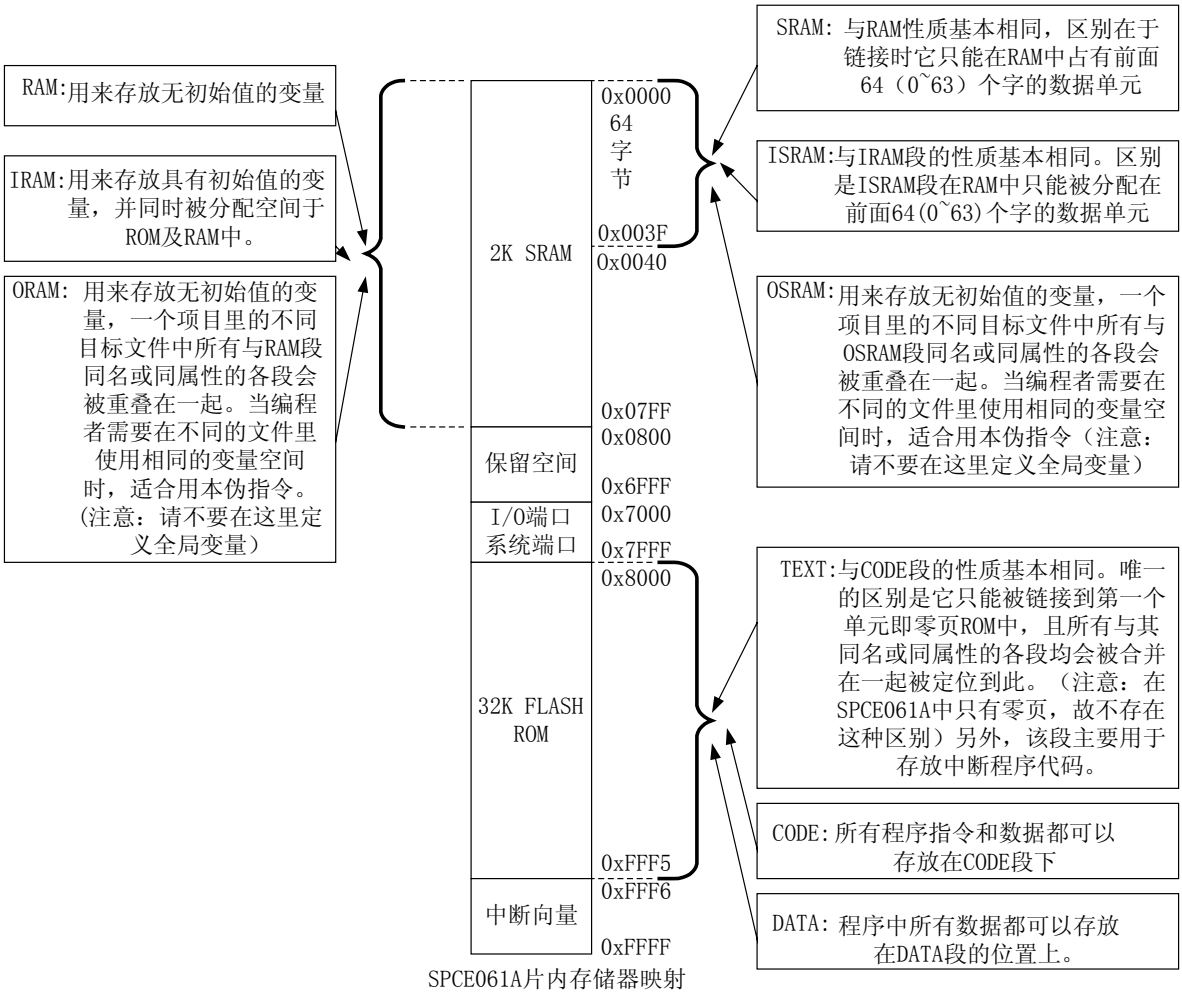


图3.8 SPCE061A 片内存储器映射

例 1：在程序中定义的变量都定位于 IRAM、ISRAM、ORAM、OSRAM、RAM 或 SRAM 段，本例比较它们的用法和区别。

程序3-2 IRAM、ISRAM、ORAM、OSRAM、RAM 和 SRAM 几个段的比较

//主程序 Main.asm 程序：

```
.EXTERNAL Syt;                          //声明 Syt 为外部标号。
.RAM                                    //切换到 RAM 段，该段存放无初始化值的变量。
.VAR Ram1,Ram2=0x0001;                  //在这里初始化 Ram2 将是无效的，系统将它初
                                        //始化为 0。
L_Ram_In_Main: .DW 5 DUP(?);

.SRAM                                   //切换到 SRAM 段，该段存放无初始化值的变量。
.VAR Sram1,Sram2=0x0002;                //在这里初始化 Sram2 将是无效的，系统将它始化为 0

.ORAM                                   //切换到 ORAM 段，该段具有覆盖属性。
.VAR Oram_In_Main;
L_Oram_In_Main: .DW 10 DUP(5);     //在这里申请的 10 个单元并存入 5，也是无效的。

.OSRAM                                  //切换到 OSRAM 段，该段具有覆盖属性。
.VAR Osram_In_Main;
L_Osram_In_Main: .DW 8 DUP(7);     //在这里申请的 8 个单元并存入 7，也是无效的。

.IRAM                                  //切换 IRAM 段，该段存放具有初始值的变量。
.VAR Iram=0x5555;
L_Iram: .DW 0x2222;

.ISRAM                                  //切换 ISRAM 段，该段存放具有初始值的变量。
.VAR Isram=0x0010;

.DATA
L_Data: .DW 10,20,30,40,50,23,43,21;

.CODE
.PUBLIC _main;
_main:
    NOP;
    NOP;
    NOP;
CALL Syt;
L_Wait:
    JMP L_Wait;

//test.asm 文件中的代码
.ORAM                                   //切换到 ORAM 段，该段具有覆盖属性。
.PUBLIC Oram1_In_Test,Oram2_In_Test; //定义全局变量（注意：在这里定义不合适）。
.VAR Oram1_In_Test,Oram2_In_Test;

.OSRAM                                  //切换到 OSRAM 段，该段具有覆盖属性。
.PUBLIC Osram1_In_Test,Osram2_In_Test;//定义全局变量（注意：在这里定义不合适）
```

```
.VAR  Osram1_In_Test,Osram2_In_Test;

.IRAM                                    //切换 IRAM 段，该段存放具有初始值的变量。
.PUBLIC  Iram_In_Test;
.VAR  Iram_In_Test=0x0010;

.TEXT
.PUBLIC  Syt;
Syt:  .PROC
      R1=[Iram_In_Test];
      RETF
.ENDP
```

程序说明：

图 3.9是例 1 程序运行后，各个存储器单元的地址和数据的分布图。从图 3.9中，我们可以证实这一点：一个项目里不同目标文件中所有与 ORAM 段（或 OSRAM 段）同名或同属性的各段会被重叠在一起。从图 3.9中可以看到在 Test.asm 文件里的 ORAM 段中定义了变量 Oram1_In_Test, Oram2_In_Test 的地址与在 Main.asm 中 ORAM 段定义的 Oram_In_Main、L_Oram_In_main 的地址重叠在一起。同样,在 Test.asm 文件里的 OSRAM 段中定义了变量 Osram1_In_Test, Osram2_In_Test 的地址与在 Main.asm 中 ORAM 段定义的 Osram_In_Main、L_Osram_In_main 的地址重叠在一起。因此，需要在不同的文件里使用相同的变量空间时，适合用本伪指令，但不要在这两个段里定义全局变量，因为它很可能会被局部变量覆盖。上面例子中在 ORAM 和 OSRAM 段中定义全局变量是不合适，不可取的。

同时还可以看到，在 RAM,SRAM,ORAM,OSRAM 段中定义有初始值的变量，或者申请有初始值的存储单元都将是无效的，系统会将它们初始化为 0。

Name	Value	Address
Ram1	0	0x00000017
Ram2	0	0x00000018
L_Ram_In_Main	0	0x00000019
Sram1	0	0x00000009
Sram2	0	0x0000000a
Oram_In_Main	0	0x0000000c
L_Oram_In_Main	0	0x0000000d
Osram_In_Main	0	0x00000000
L_Osram_In_Main	0	0x00000001
Iram	21845	0x0000001e
L_Iram	8738	0x0000001f
Isram	16	0x0000000b
L_Data	10	0x00008033
test	" test " not find ...	
Oram1_In_Test	0	0x0000000c
Oram2_In_Test	0	0x0000000d
Osram1_In_Test	0	0x00000000
Osram2_In_Test	0	0x00000001
Iram_In_Test	16	0x00000020

图3.9 各存储单元的分配图

例 2：用户编写的程序代码必须写在 DATA、CODE、或 TEXT 段，下面具体讲解 DATA、CODE、TEXT 三条伪指令的用法及区别。

利用 IOA0—7 的按键唤醒功能确定键码，并将相应键值通过七段数码管显示出来。IOA0—7 设置为具有唤醒功能的输入口，IOB0—7 设置为带缓冲器的输出口，七段数码管采用共阴极接法，见图 3.10。

程序流程如图 3.11所示。

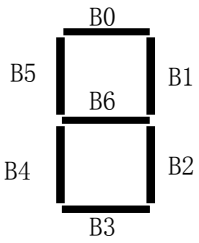


图3.10 7 段数码管

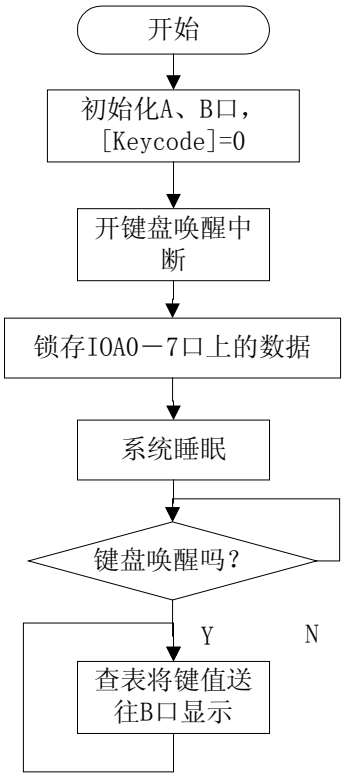


图3.11 程序流程图

程序3-3 DATA、CODE、TEXT 三条伪指令的用法及区别

```
.DEFINE  P_IOA_Data              0x7000;        //定义常量
.DEFINE  P_IOA_Dir               0x7002;
.DEFINE  P_IOA_Attr              0x7003;
.DEFINE  P_IOA_Latch             0x7004;

.DEFINE  P_IOB_Data              0x7005;
.DEFINE  P_IOB_Buffer            0x7006;
.DEFINE  P_IOB_Dir               0x7007;
.DEFINE  P_IOB_Attr              0x7008;

.DEFINE  P_INT_Ctrl              0x7010;
.DEFINE  P_INT_Clear             0x7011;
.DEFINE  P_SystemClock           0x7013;

.RAM
                                 //切换到 RAM 段，RAM 段用于存放无初始化的变量
.VAR  KeyCode

.DATA
                                 //切换到 DATA 段，DATA 段用于存放数据表格
                                 //显示数据表
DispTable:  .DW  0x00FF,0x0006,0x005B,0x004F,0x0066      //全亮、1、2、3、4
            .DW  0x006D,0x007D,0x0007,0x007F,0x006F      //5、6、7、8、9
            .DW  0x003F                                  //0
                                     //键盘表
KeyTable:   .DW  0x0000
            .DW  0x0001,0x0002,0x0004,0x0008
            .DW  0x0010,0x0020,0x0040,0x0080

.CODE
                                     //切换到 CODE 段，CODE 段用于存放程序指令
.PUBLIC  _main;
  _main:
      R1=0xFF00;                     //初始化 IOA 口低 8 位为带下拉电阻的输入
      [P_IOA_Dir]=R1
      [P_IOA_Attr]=R1;
      R1=0x0000;
      [P_IOA_Data]=R1;

      R1=0x00FF;                     //设置 IOB 口低 8 位为同相高电平输出
      [P_IOB_Dir]=R1;
      [P_IOB_Attr]=R1;
      R1=0x00FF;
      [P_IOB_Data]=R1;
```

```
        R1=0x0000                   //初始化 KeyCode 变量
        [KeyCode]=R1

        R1=0x0080;
        [P_INT_Ctrl]=R1;            //开按键唤醒中断
        IRQ  ON;

        R1=[P_IOA_Latch];           //锁存 IOA0-7 口上的数据
        R1=0x0007;
        [P_SystemClock]=R1;         //进入睡眠模式

   L_Wait:
        BP=DispTable                //查表
        BP+  =[KeyCode]
        R1=[BP]
        [P_IOB_Data]=R1;            //送 IOB 口显示
        JMP   L_Wait;

//*************************************************************************************************//
//*************************************中断程序*****************************************************//
//*************************************************************************************************//
   .TEXT
//切换到 TEXT 段，TEXT 段用于存放程序指令，链接时链接到第一个 bank，即零页 ROM 中。
   .PUBLIC  _IRQ3
   _IRQ3:
       PUSH  R1,R5  TO  [SP];
       R1=[P_INT_Ctrl]
       R1=0x0080;                   //按键唤醒中断
       TEST  R1,[P_INT_Ctrl]
       JZ    return
R1=[P_IOA_Data];                    //确定按键值
       R1=R1  and  0x00FF;
       R2=0;
       BP=KeyTable                  //查表换算成顺序值
   LOOP:
       R3=[BP++]
       CMP  R1,R3
       JE  KeyValid
       R2+=1;
       CMP  R2,8
       JBE  LOOP
       R2=0
  KeyValid:
```

```
    [KeyCode]=r2;                    //存有效键码

return:
    R1=0x0380                        //清中断
    [P_INT_Clear]=R1
    POP  R1,R5  FROM  [SP];
    RETI;
.END
```

程序说明：

μ'nSP™的具有 22 位地址线，可以寻址 4M 字的存储器容量，4M 字的存储器分为 64 个 bank，每个 bank 有 64K 字的存储容量。就 SPCE061A 来讲，由于 SPCE061A 具有 32K 字的 FLASH，所以它只有一个零页的 bank。

DATA 段用于存放程序中的数据，DATA 段可以跨 bank 链接。链接时，不同文件（一个工程可以包含几个源程序文件）中所有与其同名或同属性的各段会被分开置入 ROM 地址中，而同一文件中所有与 DATA 同名或同属性的各段会被合并在一起置入 ROM 地址中。例子中将显示数据表 DispTable 和键盘表 KeyTable 都定义在 DATA 段。

CODE 段用于存放程序指令和数据，CODE 段不可以跨 bank 链接。链接时，不同文件中所有与其同名或同属性的各段会被分开置入 ROM 地址中，而同一文件中所有与 DATA 同名或同属性的各段会被合并在一起置入 ROM 地址中。例子中的程序代码都写在 CODE 段。

TEXT 段与 CODE 段的性质基本相同。唯一的区别是它只能被链接到第一个 bank 即零页的程序存储空间中，且所有与其同名或同属性的各段均会被合并在一起而被定位到此。当 CPU 时钟频率达最高时，零页 ROM 的访问速度为 3 个 CPU 时钟周期，非零页的访问速度为 6 个机器周期；在其它 CPU 时钟频率下，零页与非零页的访问速度均为 6 个机器周期。当 CPU 时钟频率达最高时，零页的访问速度快，所以将中断处理程序写在 TEXT 段。

例 3　μ'nSP™的汇编指令包含四种数据类型：DW、DD、FLOAT 和 DOUBLE 型，下例比较它们的用法和区别。

程序3-4 四种数据类型 DW、DD、FLOAT 和 DOUBLE 的用法和区别

```
.IRAM
Label1:  .DW    12.3456789
Label2:  .DD    12.3456789
Label3:  .FLOAT  12.3456789
Label4:  .DOUBLE  12.3456789

Label5:  .DW    0
Label6:  .DD    0
Label7:  .FLOAT  0
Label8:  .DOUBLE  0

.CODE
```

```
.PUBLIC  _main;
  _main:
       R1=0x5555;
       BP=Label5
       [BP]=R1
       BP=Label6
       [BP]=R1
       BP=Label7
       [BP]=R1
       BP=Label8
       [BP]=R1
  L_Wait:
       JMP   L_Wait;
```

程序说明：

1．用 DW 定义的变量以 16 位整型数据的形式来存储。整型变量可以是多种类型的操作数，需用逗号(，)将多个数值分开；若存储的变量值中含有 ascii 字符串，则必须用引号(“”)将其括起来。如果.DW 后面未输入任何数值，则会自动存入一个整型零值。例子中 Label1、Label5 定义为 DW 类型，将 Label1 初始化为 12.3456789 时，只保留整数部分 12（见图 3.12）。向 Label5 送 0x5555 时，Label5 单元存入 0x5555。

2．用 DD 定义的变量以 32 位长整型数据的形式存储。存放的变量可以任何一种数制格式输入，但最终由汇编器将其转换成十六进制格式存放。

例子中 Label2、Label6 定义为 DD 类型，将 Label2 初始化为 12.3456789 时，只保留整数部分 12，扩展成 32 位 0x0000000C 存储，占用 Label2、Label2+1 两个单元，见图 3.12。向 Label6 送 0x5555 时，Label6 单元存入 0x5555，Label6+1 存入 0x0000。

3．用 FLOAT 定义的变量以单精度浮点型实数数据形式来存储。本伪指令用来把实数变量的值转换成以 IEEE 格式表示的单精度浮点型数据并存储。所谓单精度浮点型实数是指数的精度保证 6 位有效位。若超过 6 位，则对第七位进行入舍处理后，其后所有位均会被舍去。

例子中 Label3、Label7 定义为 FLOAT 类型，将 Label3 初始化为 12.3456789 时，只保留 6 位有效位，对第七位进行四舍五入，得到 12.3457，并以 IEEE 格式表示的单精度浮点型数据存储，见图 3.12的 0x00000003、0x00000004 两个地址单元。对变量的操作也应该符合 IEEE 格式，比如 Label7 送 0x5555 时，Label7 单元存入 0x5555，Label7+1 单元存入 0x0000，但不符合 IEEE 格式。

4．用 DOUBLE 定义的变量以双精度浮点型实数数据的形式存储。把实数变量的值转换成以 IEEE 格式表示的双精度浮点型数据并存储，本伪指令存储的变量必须是实数。

例子中 Label4、Label8 定义为 DOUBLE 类型，将 Label4 初始化为 12.3456789 时，以 IEEE 格式表示的双精度浮点型数据存储，见图 3.12的 0x00000005、0x00000006、0x00000007、0x00000008 四个地址单元。对常量的操作也应该符合 IEEE 格式，比如 Label8 送 0x5555 时，Label8 单元存入 0x5555，Label8+1、 Label8+2、 Label8+3 单元存入 0x0000，但不符合 IEEE 格式，见图 3.12的 0x0000000E、0x0000000F、0x00000010、0x00000011 四个地址单元。

DW、DD、FLOAT 和 DOUBLE 型这四种数据类型的字长和值域见表 3.4。

表3.4 μ'nSP™汇编指令中的数据类型

数据类型	字长度（位数）	无符号数值域	有符号数值域
字型（DW）	16	0~65535	-32768~+32767
双字型（DD）	32	0~4294967295	-2147483648~+2147483647
单精度浮点型（FLOAT）	32	无	以 IEEE 格式表示的 32 位浮点数
双精度浮点型（DOUBLE）	64	无	以 IEEE 格式表示的 64 位浮点数

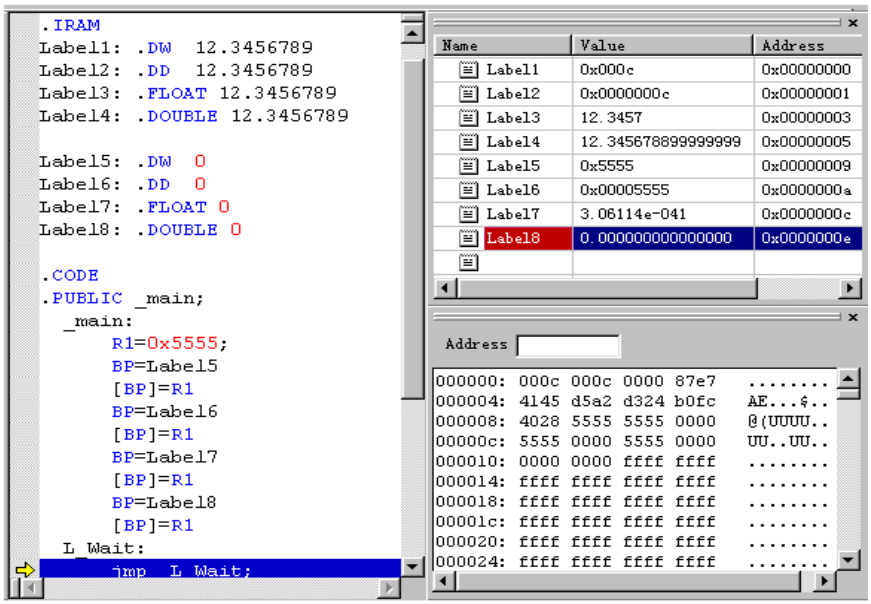


图3.12 各存储单元的分配图

例 4：下例讲解文件包含伪指令 INCLUDE；赋值伪指令 DEFINE；条件汇编伪指令 IF、ELSE、ENDIF；宏 MACRO、ENDM；过程伪指令 PROC、ENDP 的用法。
程序流程图如图 3.13所示

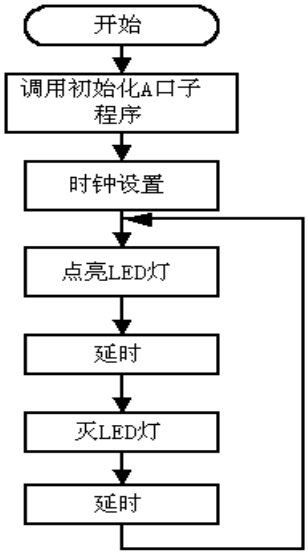


图3.13 主程序流程图

程序3-5 INCLUDE、DEFINE、IF、ENDIF、MACRO 以及 PROC 等伪指令的使用

```
//A 口的 IOA0-I0A7 接 LED(低电平点亮 LED)
.INCLUDE  Hardware.inc;                  //将 Hardware.inc 文件包含进来
//.DEFINE  FoscCLK              0x20;    //Fosc=20.480MHz   为 FoscCLK 赋值
.DEFINE  FoscCLK               0x00;    //Fosc=24.576MHz   为 FoscCLK 赋值
.IF  FoscCLK==0x20;                      //条件汇编
.DEFINE  CPUCLK          0x00;           //CPUClk 选 Fosc
.ELSE
.DEFINE  CPUCLK          0x02;           //CPUClk 选 Fosc/4
.ENDIF                                   //结束条件汇编
Delay: .MACRO   TIM                      //宏定义，有一个宏参数 TIM
      R3=TIM;
      DelayLoop1#:
//隐含标号，程序编译时会将宏展开，为了避免标号重复定义，必须使用隐含标号
      R4=0xFFFF;
DelayLoop2#:                             //隐含标号
      R4-=1;
      JNZ  DelayLoop2#;
      R3-=1;
      JNZ  DelayLoop1#;
.ENDM                                    //结束宏定义

.CODE
.PUBLIC  _main
```

```
_main:
      CALL  Init_IO;                    //调用过程
      R1=FoscCLK;                       //Fosc
      R1|=CPUCLK;                       //CPUClk
      [P_SystemClock]=R1;               //系统时钟选择设置
MainLoop:
     R1=0x00FF;                         //LED 灭（输出低电平亮）
     [P_IOA_Data]=R1;
     Delay  5;                          //宏调用，用实参 5 代替宏定义中的 TIM
     R1=0x00;                           //LED 亮
     [P_IOA_Data]=R1;
     Delay  18;                         //宏调用，用实参 18 代替宏定义中的 TIM
     JMP  MainLoop;

Init_IO:  .PROC                         //过程定义
     R1=0x00FF;
      [P_IOA_Dir]=R1;                   //设置 IOA0-IOA7 为同相低电平输出
      [P_IOA_Attrib]=R1;
      R1=0;
      [P_IOA_Data]=R1;
     RETF;
 .ENDP                                  //结束过程定义
```

程序说明：

INCLUDE　filename 伪指令用来通知汇编器把指定的文件包含在程序文件中一起进行汇编。其中参量 filename 为要指定文件名（含扩展名），且可指明搜索文件所需的路径。例子中将 Hardware.inc 文件包含进来（路径由工程中的 DIRECTORIES 指定，如果没有指定，则在当前目录下查找）。

伪指令 DEFINE 用于定义常量。给常量所赋的值既可是已定义过的常量符号，亦可是表达式。切忌符号超前引用，即如果赋值引用的符号不是在引用前定义的，则会出现“非法超前引用”的错误

条件汇编伪指令 IF 引出在条件为真时所要汇编的程序指令，ELSE 引出 IF 伪指令设置的条件为假时所要汇编的程序指令，END 结束条件汇编。例子中如果前面定义 FoscCLK 的值为 0x20，则定义 CPUCLK 值为 0x00，否则定义 CPUCLK 值为 0x02。

MACRO 伪指令用来起始宏定义。结束宏定义则用 ENDM 伪指令，二者应成对使用。宏定义里可以用显式标号（由用户定义），亦可用隐含标号（由汇编器自动定义）。汇编器不会改变用户定义的显式标号。在宏标号后加上后缀符‘#’则表明该标号为隐含标号，汇编器会自动生成一个后缀数字符号‘_X_XXXX’（X 表示一位数符，XXXX 表示 4 位扩展数符）来取代这个隐含标号中的后缀符‘#’，隐含标号中的字母字符及其后缀数符总共不能超过 32 个字符。在汇编器首次编译通过汇编指令时，先将宏定义存储起来，待指令中遇有被调用的宏名则会用同名宏定义里的序列源指令行取代此宏名，因此为了避免宏展开时出现标号重复，应该用隐含标号，本例中 Delay 宏完成一段延时，延时时间由宏调用时参数 TIM 的实参值确定。

PROC 伪指令用于起始程序的定义，应与结束程序定义的伪指令 ENDP 成对使用，过程就是一段子程序。本例中 Init_IO 过程用来初始化 IOA 口。